Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

Automatización de la mitigación de vulnerabilidades mediante OpenVAS y modelos de lenguaje

Estudante: Daniel Barbeyto Torres

Dirección: Adrián Carballal Mato

A Coruña, enero de 2026.

A mi familia, tanto la que se escoje como la que no.

Resumen

La creciente cantidad de vulnerabilidades e incidentes en los equipos informáticos evidencia la necesidad de herramientas para tanto detectar como mitigar las amenazas que estas provocan. Sin embargo, aunque existen varias soluciones comerciales que realizan esta tarea, su coste es muy elevado. Por otra parte, las herramientas *open source* tradicionales de gestión de vulnerabilidades, como OpenVAS, son muy eficaces en la fase de detección, pero delegan su remediación en el operador humano, lo que ralentiza la respuesta ante incidentes críticos.

Con el objetivo de abordar esta problemática y motivado por el auge de la inteligencia artificial, este documento propone el diseño e implementación de un sistema automatizado orientado a redes internas, que integra OpenVAS con modelos de lenguaje (LLMs) para cerrar el ciclo desde la detección de las vulnerabilidades hasta su mitigación. La herramienta desarrollada actúa como una solución centralizada que permite auditar múltiples equipos dentro de la misma red, realizando escaneos a las máquinas objetivo que generan informes con sus vulnerabilidades presentes para procesarlos y generar, de forma autónoma, scripts de seguridad personalizados para remediar los fallos detectados. La arquitectura de la herramienta usa un agente inteligente, basado en la API de OpenAI, que interpreta los resultados del escáner y genera los scripts Bash ejecutables para su posterior mitigación, mediante reglas de cortafuegos o reconfiguración de servicios, entre otros.

Además del desarrollo, el documento incluye una evaluación detallada centrada en la eficacia de las mitigaciones generadas y el rendimiento del sistema en un entorno virtualizado controlado. Para ello se han realizado varias pruebas de validación tanto unitarias como masivas, probando que la herramienta puede reducir drásticamente la superficie de ataque sin ningún tipo de asistencia humana y logrando una tasa de éxito superior al 90% en la generación y ejecución de los parches de seguridad.

La herramienta ha sido implementada siguiendo una metodología iterativa-incremental, permitiendo refinar el comportamiento del agente que orquesta el *pipeline* y asegurar la robustez de la automatización mediante revisiones continuas. Se ha definido una planificación inicial con seguimiento de hitos, y se realizó un análisis detallado de la viabilidad técnica y económica de la solución propuesta.

Como resultado, se proporcionan pruebas de concepto funcional que demuestran el potencial de los modelos de lenguaje para asistir en tareas complejas de administración de sistemas y ciberseguridad, sentando las bases para futuras investigaciones en defensa de sistemas automatizados.

Abstract

The escalating number of vulnerabilities and incidents in computer systems underscores the need for tools capable of both detecting and mitigating the resulting threats. While several commercial solutions address this task, their cost is often prohibitive. Conversely, traditional open-source vulnerability management tools, such as OpenVAS, are highly effective in the detection phase but delegate remediation to the human operator, slowing down the response to critical incidents.

To address this issue and motivated by the rise of artificial intelligence, this document proposes the design and implementation of an automated system oriented towards internal networks, which integrates OpenVAS with Large Language Models (LLMs) to close the loop from vulnerability detection to mitigation. The developed tool acts as a centralized solution allowing the audit of multiple devices within the same network, performing scans on target machines that generate reports on detected vulnerabilities and process them to autonomously generate personalized security scripts to remediate the flaws. The tool's architecture uses an intelligent agent, based on the OpenAI API, which interprets scanner results and generates executable Bash scripts for subsequent mitigation, using firewall rules or service reconfiguration, among others.

In addition to development, this document includes a detailed evaluation focused on the efficacy of generated mitigations and system's performance in a controlled virtualized environment. For this purpose, several validation tests, both unit and massive, have been conducted, proving that the tool can drastically reduce the attack surface without any human assistance, achieving a success rate exceeding 90% in the generation and execution of security patches.

The tool has been implemented following an iterative-incremental methodology, allowing the refinement of the agent orchestrating the pipeline and ensuring automation robustness through continuous reviews. An initial schedule with milestone tracking was defined, and a detailed analysis of the technical and economic feasibility of the proposed solution was performed.

As a result, functional proofs of concept are provided demonstrating the potential of language models to assist in complex system administration and cybersecurity tasks, laying the groundwork for future research in automated defense systems.

Palabras clave:

- Ciberseguridad
- Gestión de vulnerabilidades
- Automatización
- Modelos de Lenguaje
- OpenVAS

Keywords:

- Cybersecurity
- Vulnerability Management
- Automation
- Large Language Models
- OpenVAS

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo tiene como objetivo introducir el marco general en el que se encuadra este Trabajo de Fin de Grado (TFG). En primer lugar, se contextualiza el entorno y la problemática actual que justifica la necesidad de la solución propuesta. A continuación, se definen los objetivos tanto principales como específicos que se pretenden alcanzar durante el desarrollo de este proyecto. Por último, se detalla el contenido de cada uno de los capítulos de este documento para facilitar la comprensión global del mismo al lector.

1.1 Contexto y motivación

En el contexto tecnológico actual, la ciberseguridad ha dejado de ser una preocupación técnica aislada para convertirse en uno de los pilares más críticos de la estabilidad tanto operativa como económica a nivel global. Esto se debe a que la interconexión masiva de dispositivos y demás sistemas en infraestructuras de red han expandido exponencialmente la superficie de exposición de ataques digitales. Como consecuencia de esto, el volumen, frecuencia y sofisticación de los ataques informáticos está creciendo a un ritmo muy alarmante, comprometiendo la integridad de servicios esenciales tanto en el sector público como el privado.

Para comprender la magnitud de lo tangibles y devastadoras que pueden ser dichas consecuencias podemos fijarnos en un par de ejemplos que demuestran la gravedad del problema planteado. Como primer ejemplo, el 5 de marzo de 2023, el Hospital Clínic de Barcelona (uno de los hospitales más importantes de España) sufrió un ciberataque de *ransomware* que forzó cancelar 150 cirugías no urgentes y más de 3000 consultas externas en solo los primeros días. Además, se secuestraron 4,5 terabytes de datos por los que se pidió un rescate de 4,5 millones de dólares [2].

Otro caso en el ámbito de las infraestructuras críticas es el ciberataque de *ransomware* a Colonial Pipeline. Colonial Pipeline es un sistema vital de oleoductos de casi 9000 kilómetros

que transporta gasolina, diésel y combustible para aviones desde Texas hasta Nueva York. Este ciberataque interrumpió el suministro de combustible en la costa este de los Estados Unidos, obligando a declarar el estado de alerta regional [3].

Como podemos ver, estos eventos demuestran que la falta de una capacidad de reacción rápida no solo genera pérdidas económicas, sino que pone en riesgo el bienestar social.

Además, los datos más recientes de los organismos de referencia corroboran esta tendencia alcista. El Balance de Ciberseguridad publicado por el INCIBE (Instituto Nacional de Ciberseguridad) en el 2024 muestra que se gestionaron 97.000 incidentes de seguridad durante el último año, lo que representa un aumento del 16,6% respecto al 2023 [4].

Esta escalada no es solo un problema nacional. A nivel europeo, la ENISA (Agencia de la Unión Europea para la Ciberseguridad) publicó el informe *Threat Landscape 2024* donde advierte sobre que los conflictos regionales en curso siguen siendo un factor significativo que configura el panorama de la ciberseguridad. Además, también confirma que los ciberciminales muestran un mayor nivel de colaboración y profesionalización [5]. Por otro lado, organismos como la Europol en su informe IOCTA 2024 [6] o el CCN-CERT (Centro Criptológico Nacional) destacan el empleo de la inteligencia artificial para llevar a cabo ciberataques o desarrollar malware, lo que provoca una reducción drástica del tiempo de reacción disponible para los equipos de ciberdefensa.

Para poder entender el impacto de estas amenazas, es necesario mencionar el ciclo de vida estándar de la gestión de vulnerabilidades, el cual es establecido por marcos de trabajo internacionales, entre ellos la norma ISO/IEC 27001 [7]. Este ciclo consta de 4 fases principales:

- **Descubrimiento:** Identificación de los activos críticos, entre otros.
- **Evaluación:** Detección de fallos mediante diversas técnicas de análisis.
- **Priorización:** Análisis de riesgos y clasificación de estos por su gravedad, impacto, etc.
- **Remediación:** Fase dedicada a la aplicación de medidas correctivas.

Mientras que por un lado, las fases de descubrimiento y evaluación han alcanzado un alto grado de madurez, la fase de remediación sigue siendo un cuello de botella crítico por los motivos anteriormente mencionados. El tiempo entre la detección de una vulnerabilidad y su mitigación sigue siendo demasiado alto, generando una ventana de exposición que los ciberdelincuentes pueden aprovechar rápidamente.

A día de hoy, afortunadamente existen soluciones comerciales avanzadas que abordan este ciclo completo, conocidas como plataformas VMDR (*Vulnerability Management, Detection and*

Response) o sistemas SOAR (*Security Orchestration, Automation and Response*). Según Qualys, este tipo de sistemas tienen como fin la combinación de escaneo continuo con agentes, sensores pasivos y contexto de amenazas para reducir el tiempo de respuesta (MTTR (*Mean Time To Repair*)) y asegurar entornos híbridos y dinámicos [8].

Herramientas líderes como Qualys VMDR [8], Tenable One con capacidades de automox [9] o plataformas de orquestación como Palo Alto Cortex XSOAR [10], ofrecen capacidades de remediación automática y parcheo desatendido. Por otro lado, su coste de licenciamiento es demasiado elevado, provocando que sean inaccesibles para muchas organizaciones medianas o redes internas con presupuestos ajustados. Por otro lado, las herramientas *open source* tradicionales como OpenVAS (*Greenbone Vulnerability Management*) [11] tienen la ventaja de ser muy eficaces en la fase de detección, permitiendo identificar vulnerabilidades con precisión, pero su alcance funcional suele limitarse al diagnóstico y reporte, delegando las mitigaciones y correcciones correspondientes a los administradores humanos, obligándolos a analizar manualmente extensos informes técnicos y aplicar correcciones una por una, de forma muy lenta y difícilmente escalable.

Esta desconexión entre la capacidad de detección (muchas veces automática y muy accesible) y la capacidad de respuesta (muy costosa o manual y lenta) es la problemática central que motiva este Trabajo de Fin de Grado, el cual tratará de integrar la potencia de detección de herramientas *open source* como OpenVAS, con la capacidad de razonamiento de los modelos de lenguaje (LLMs), logrando así un acceso consistente a la defensa activa automatizada.

1.2 Objetivos

La problemática que se ha descrito en el apartado anterior, centrada en la brecha entre la capacidad de detección y la capacidad de respuesta, motiva el diseño de una solución de orquestación ágil. Esta carencia justifica el propósito de este proyecto a dar respuesta a dicha necesidad, diseñando e implementando un *pipeline* automatizado que integre modelos de lenguaje (LLMs) en el flujo de trabajo de gestión de vulnerabilidades para que la remediación de fallos sea lo más eficiente, rápida y precisa posible.

1.2.1 Objetivo principal

El objetivo principal de este Trabajo de Fin de Grado es el desarrollo de un sistema automatizado modular que integre la capacidad de análisis de OpenVAS/GVM con la capacidad de generación de scripts de mitigación de un modelo de lenguaje, para solucionar automáticamente las vulnerabilidades de las máquinas de una red interna.

1.2.2 Objetivos específicos

Para poder alcanzar el objetivo principal, se plantean los siguientes objetivos específicos:

1. Desarrollar un entorno de laboratorio virtualizado aislado donde poder probar la solución y realizar las pruebas de validación funcional necesarias.
2. Desarrollar la arquitectura de un pipeline que permita extraer el reporte de seguridad realizado por OpenVAS a una máquina y transformarlo en el formato de entrada necesario para el agente de Inteligencia Artificial.
3. Integrar funciones de análisis semántico mediante la API de OpenAI para generar scripts de mitigación en Bash (ejemplo: reglas de firewall, desactivación de servicios, etc.)
4. Automatizar la realización de la tarea periódicamente mediante programadores de tareas (cron) para poder garantizar la vigilancia continua y desatendida del sistema.
5. Realizar una evaluación de la efectividad de la solución planteada mediante pruebas de validación funcional evaluando una serie de métricas, como el tiempo medio de mitigación, la tasa de éxito de ejecución o la reducción de la superficie de ataque.

1.3 Estructura del documento

Esta memoria está organizada en 7 capítulos y un anexo que abarcan desde la presentación de la problemática y los objetivos, hasta la evaluación de los resultados y las conclusiones extraídas una vez desarrollada e implementada la solución. La estructura es la siguiente:

- **Introducción:** En este capítulo se presenta el marco general del proyecto, detallando la relevancia del trabajo, los retos que pretende abordar y los objetivos principales que se desean alcanzar.
- **Fundamentos teóricos:** Este capítulo ofrece una revisión de los conceptos y teorías fundamentales que sustentan el proyecto, tales como la gestión de vulnerabilidades, el sistema de puntuación CVSS, OpenVAS y los Modelos de Lenguaje (LLMs), entre otros.
- **Estado de la Cuestión:** En este capítulo se presenta un análisis de las soluciones planteadas, evaluando el entorno, las herramientas y estudios previos relacionados para identificar las debilidades que motivan este trabajo.
- **Metodología y plan de proyecto:** Este capítulo detalla la metodología seleccionada para realizar el proyecto, detallando tanto las fases del proyecto como los recursos y costes estimados para poder llevarlo a cabo en un entorno profesional.

- **Desarrollo e Implementación:** Este capítulo es el núcleo de la memoria. En él se detalla cómo ha sido la implementación realizada conforme al plan de proyecto y metodología seleccionados. Se explica cómo se ha llegado a la solución propuesta y se describe su arquitectura paso a paso.
- **Resultados:** En este capítulo se realizan las pruebas de validación funcional sobre la herramienta para poder llevar a cabo una evaluación técnica que permita medir su efectividad, fiabilidad y rendimiento, entre otros.
- **Conclusiones y trabajo futuro:** Este último capítulo contiene las conclusiones extraídas a partir del desarrollo del proyecto. También se mencionan cuáles serían las posibles futuras líneas de desarrollo para la continuación del proyecto, fuera del alcance de este trabajo.

Además, al final del documento se encuentra un breve anexo que facilita al lector la instalación y despliegue local de la solución implementada, llamado "Manual de Instalación y Despliegue".

Capítulo 2

Fundamentos Teóricos

Este capítulo establece el marco conceptual y teórico necesario para la comprensión del sistema desarrollado. En este capítulo se detallan los principios de la ciberseguridad y la gestión de vulnerabilidades, las diferentes fases del ciclo de gestión de vulnerabilidades, los estándares de clasificación y puntuación, las tecnologías de detección usadas durante el proyecto, la explicación teórica de las tecnologías de automatización y de modelos de lenguaje necesarias para implementar el proyecto y, por último, el entorno de pruebas y desarrollo usados para este Trabajo de Fin de Grado.

2.1 Ciberseguridad y la gestión de vulnerabilidades

El contexto operativo de este proyecto se sitúa sobre la auditoría de seguridad ofensiva y la defensa activa de sistemas. Como se mencionó anteriormente, la sofisticación de las amenazas actuales requiere superar los enfoques defensivos estáticos. Para entender esta sofisticación que propone la solución, es necesario definir formalmente los conceptos de auditoría, ciclo de vida de las vulnerabilidades y los estándares de la industria que permiten su automatización.

2.1.1 Auditoría de seguridad y hacking ético

La seguridad absoluta de un sistema es imposible de alcanzar. Es por ello que las empresas adoptan estrategias de defensa en profundidad para poder protegerse de los ciberataques de la mejor manera posible. En este contexto es donde las auditorías de seguridad cobran su verdadera importancia.

El hacking ético consiste en el uso de técnicas de hacking para tratar de descubrir, comprender y mitigar de forma legal y anteriormente pactada vulnerabilidades de seguridad en una red o sistema.

Este Trabajo de Fin de Grado automatiza esta tarea como un auditor que realiza 2 funcio-

nes:

- **Capacidad ofensiva (*Red Team*)**: Detección proactiva de fallos mediante escaneos de puertos y servicios. Esta tarea es delegada en este trabajo al motor de escaneo de vulnerabilidades OpenVAS.
- **Capacidad defensiva (*Blue Team*)**: Propuesta y ejecución de la corrección o mitigación de las vulnerabilidades detectadas. Esta tarea es delegada en este trabajo a un Agente Python basado en LLMs.

2.1.2 El ciclo de gestión de vulnerabilidades

Según el Instituto Nacional de Estándares y Tecnología (NIST), en su publicación especial SP 800-40 Rev. 4, la gestión de vulnerabilidades es un ciclo de vida diseñado para poder realizar una planificación de respuesta ante vulnerabilidades de forma continua [12].

Las fases de este proyecto se han alineado con el procedimiento descrito en la sección 2.2 de dicha publicación. Estas se pueden visualizar en la Figura 2.1 y se describen a continuación:

1. **Identificación**: Se realiza un descubrimiento de los activos presentes en el alcance definido. A dichos activos, se les realiza un reconocimiento para saber cuándo les afectan nuevas vulnerabilidades. En este proyecto, esta fase se ve reflejada en el escaneo de vulnerabilidades realizado por la herramienta OpenVAS.
2. **Planificación de la respuesta**: Se realiza un análisis del riesgo de las vulnerabilidades para poder determinar el orden de atención y la respuesta adecuada a cada una de ellas.
3. **Ejecución e implementación**: Se realiza la acción para mitigar la vulnerabilidad de seguridad. En la solución desarrollada, esta fase se centra en la generación y ejecución automática de scripts de mitigación, que son ejecutados de forma remota en las máquinas objetivo para la mitigación de cada una de las vulnerabilidades.
4. **Verificación:** Se comprueba que las acciones correctivas de los scripts de mitigación han sido aplicadas correctamente. Para ello, realizamos un segundo escaneo de las vulnerabilidades presentes en las máquinas objetivo de nuevo con OpenVAS. Si las mitigaciones han sido realizadas correctamente, OpenVAS nos devolverá un segundo informe que mostrará muchas menos vulnerabilidades presentes en las máquinas objetivo.

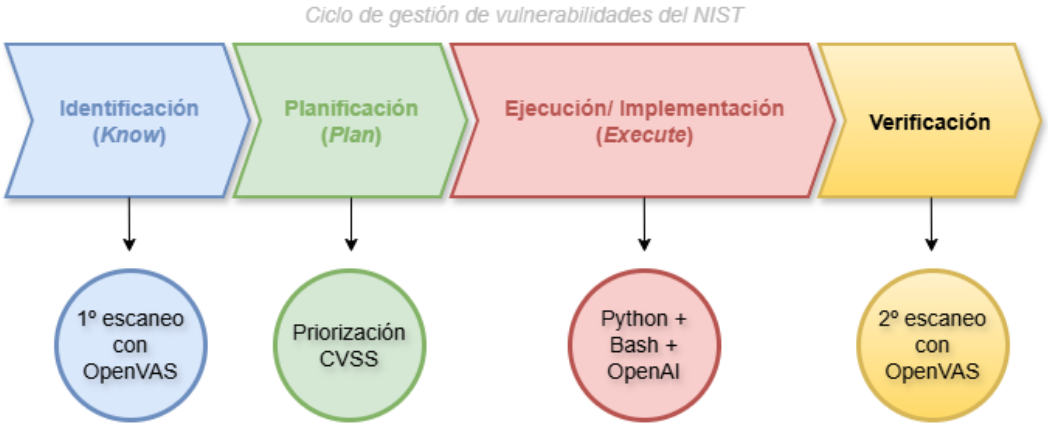


Figura 2.1: Ciclo de vida de gestión de vulnerabilidades. Elaboración propia a partir del NIST SP 800-40r4

2.1.3 Estándares de clasificación y puntuación

Identificadores CVE

Los identificadores CVE son el estándar para la identificación única de vulnerabilidades en el sistema CVE (*Common Vulnerabilities and Exposures*), supervisado por MITRE Corporation [13].

Cada CVE proporciona información única para cada vulnerabilidad que se conozca públicamente. El uso de los CVEs en el pipeline del proyecto permite la interoperabilidad semántica entre el módulo de escaneo de vulnerabilidades (OpenVAS) y el módulo de modelo de lenguaje (LLM) que genera los scripts de mitigación, asegurando que dicho script generado corresponda exactamente con la vulnerabilidad que ha sido detectada.

Sistema de puntuación CVSS

Para la priorización objetiva de las acciones de respuesta, la industria adopta el sistema CVSS (*Common Vulnerability Scoring System*), mantenido por el FIRST (*Forum of Incident Response and Security Teams*). En su versión 3.1, este estándar abierto proporciona un método para capturar las características principales de una vulnerabilidad y producir una puntuación numérica que refleja su severidad [14].

El vector CVSS se compone de tres grupos de métricas detalladas en la guía oficial [15]:

- **Métricas Base:** Representan las características de la vulnerabilidad que son constantes en el tiempo y entre entornos.

- **Métricas Temporales:** Reflejan las características de la vulnerabilidad que no son constantes en el tiempo y entre entornos (por ejemplo, la disponibilidad de un exploit funcional).
- **Métricas de Entorno:** Ajustan la puntuación según el entorno específico.

Podemos visualizar un ejemplo del detalle de una CVE en la Figura 2.2, con su correspondiente puntuación CVSS.

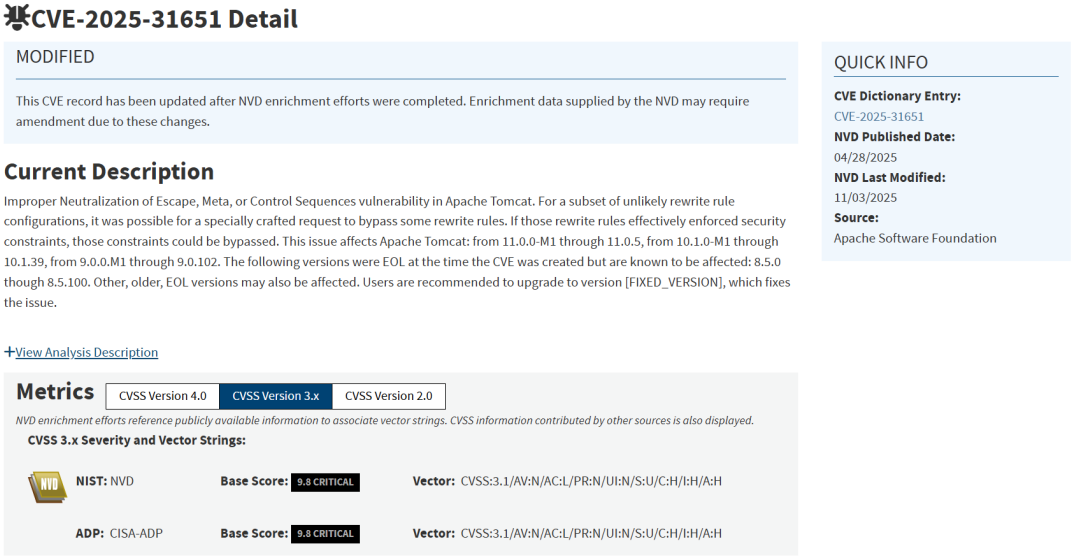


Figura 2.2: Ejemplo de detalle de vulnerabilidad CVE-2025-31651 en el NVD.

2.2 Tecnologías de detección

El componente principal y central de la fase de detección de este proyecto es Greenbone Vulnerability Management (GVM). Aunque se sigue usando el término OpenVAS para referirse a la plataforma, OpenVAS es solo el motor de escaneo dentro de un sistema mucho más grande.

2.2.1 Greenbone Vulnerability Management (GVM)

Greenbone Vulnerability Management es una arquitectura que se basa en un modelo de microservicios que separa la lógica de gestión, el almacenamiento de datos y la ejecución de pruebas de seguridad [1]. Los componentes críticos desplegados en el entorno de laboratorio se detallan a continuación y pueden visualizarse en la Figura 2.3:

1. **Greenbone Vulnerability Manager Daemon (*gvmd*):** Servicio central del sistema. Sus funciones entre otras son gestionar la configuración, almacenar los resultados en una base de datos (como por ejemplo PostgreSQL) y gestionar los usuarios y permisos.
2. **OpenVAS Scanner (*openvas-scanner*):** Motor que ejecuta las pruebas de vulnerabilidad contra los objetivos.
3. **Greenbone Security Assistant (*gsad*):** Proporciona la interfaz web que se usa para la configuración manual inicial. Permite la visualización gráfica de reportes. No obstante, no usaremos esta interfaz en el proyecto debido a la intención de automatizar la solución.

2.2.2 *Greenbone Management Protocol* (GMP)

El protocolo *Greenbone Management Protocol* permite la integración con herramientas externas y la orquestación de tareas sin intervención humana. Es un protocolo basado en XML que permite usar todas las funciones del gestor gvmd [16]. GMP usa comandos XML enviados directamente a través de un socket Unix o TLS, lo que lo diferencia de una API REST tal y como la conocemos.

Durante el desarrollo del proyecto, usaremos la herramienta gvm-cli como cliente para realizar las peticiones. Esto permite que el pipeline de automatización desarrollado en Python pueda autenticarse, iniciar tareas de escaneo y extraer reportes de vulnerabilidades directamente en formato XML, lo que elimina la necesidad de interacción manual con la interfaz gráfica de OpenVAS.

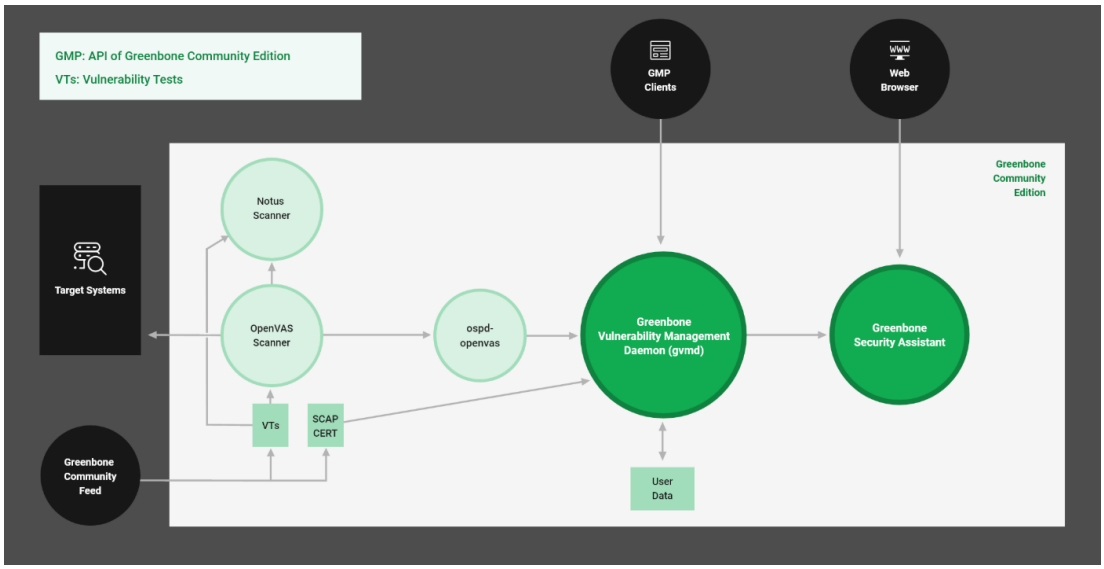


Figura 2.3: Arquitectura de Greenbone en la versión 22.4 [1]

2.3 Inteligencia artificial y modelos de lenguaje

En este trabajo, la capacidad de razonamiento y generación del script de seguridad de remediación de las vulnerabilidades detectadas se sustenta en el uso de Inteligencia Artificial. De esta forma se permite la interpretación de vulnerabilidades en lenguaje natural para la generación de scripts adaptados al contexto especifico.

2.3.1 *Large Language Models* (LLMs)

Los modelos de lenguaje de gran tamaño (LLMs) son una categoría de modelos de *deep learning* entrenados sobre inmensas cantidades de datos, lo que los hace capaces de comprender y generar lenguaje natural y otros tipos de contenidos para realizar una amplia gama de tareas. Los LLMs se basan en un tipo de arquitectura de red neuronal que destaca en la gestión de secuencias de palabras y la captura de patrones de texto. [17].

Para la implementación del Agente en este proyecto, se ha seleccionado el modelo de gpt-5-mini desarrollado por OpenAI. Se ha elegido este modelo debido a que ofrece un equilibrio óptimo entre capacidad de razonamiento y coste económico, lo que lo hace viable para tareas repetitivas de automatización masiva [18].

2.3.2 Ingeniería de prompts

La calidad del código generado por un LLM no depende únicamente de la potencia del modelo, sino de la precisión de las instrucciones que recibe. La ingeniería de prompts es la disciplina técnica de diseñar, refinar y optimizar estas entradas para guiar al modelo hacia una salida precisa, segura y estructurada [19].

En el desarrollo del script `agent.py` (archivo donde se ha desarrollado el agente), se ha implementado una estrategia de ingeniería de prompts robusta que define el rol del asistente (”Experto en Ciberseguridad Defensiva”) y establece restricciones operativas críticas para la seguridad del sistema, como por ejemplo:

- Exigencia de devolver solamente código bash dentro de bloques delimitados, sin ningún tipo de texto explicativo adicional que pudiera romper el flujo.
- Prohibición de uso de comandos interactivos o destructivos que imposibiliten la ejecución del script de mitigación de forma desatendida.
- Inyección de datos específicos y detallados del entorno objetivo del script (por ejemplo, el sistema operativo, la versión de kernel) para evitar la generación de comandos incompatibles con el sistema.

2.3.3 Integración mediante API REST

La comunicación entre el módulo de orquestación (Agente) y el modelo de modelo de lenguaje (OpenAI) se realiza a través de una Interfaz de Programación de Aplicaciones (API) basada en la arquitectura Representational State Transfer (REST).

En el proyecto, el agente Python actúa como un cliente HTTP que realiza solicitudes de tipo POST a la API de OpenAI. Cada una de las solicitudes realizadas contienen el contexto de la vulnerabilidad y el prompt del sistema en formato JSON. Este planteamiento resulta ideal, puesto que permite procesar cada vulnerabilidad independientemente (procesando una vulnerabilidad por solicitud), facilitando la escalabilidad y el manejo de errores. [20].

2.4 Tecnologías de automatización

Para poder automatizar la solución, debemos conseguir integrar las diferentes herramientas en un flujo de trabajo unificado. Esta sección describirá las tecnologías que se han seleccionado para poder realizar la orquestación, el intercambio de datos y la comunicación segura.

2.4.1 Lenguajes de scripting

Para implementar la lógica del sistema, se ha seleccionado Python (versión 3.13.17) como lenguaje de scripting para la programación del agente orquestador debido a su robustez, legibilidad y gran disponibilidad de librerías para tareas de automatización de redes y procesamiento de textos [21].

El agente desarrollado (`agent.py`) actúa como el núcleo de la inteligencia. Entre sus funciones se encuentran:

- Interacción con el sistema de archivos del sistema operativo.
- Gestión de la comunicación síncrona con la API de OpenAI.
- Orquestación de tareas dentro del pipeline propuesto.

Para la gestión de las dependencias necesarias usadas por este script (como por ejemplo, la librería de OpenAI), se ha usado un entorno virtual de Python. Un entorno virtual es un directorio aislado que contiene su propio binario de Python y un conjunto independiente de dependencias instaladas, evitando conflictos con las versiones instaladas en el sistema operativo local [22].

Por otro lado, se usa el lenguaje Bash para la ejecución final de los scripts de mitigación en los equipos objetivo, aprovechando su capacidad para interactuar directamente con el kernel y los servicios del sistema (como por ejemplo, iptables).

2.4.2 Formatos de intercambio de datos

El intercambio de datos entre el módulo de escáner de vulnerabilidades (OpenVAS) y el módulo de modelo de lenguaje (LLM) requiere una transformación de formatos de datos.

- **XML (*eXtensible Markup Language*)**: Formato de salida de los reportes de GVM. Es un estándar robusto y jerárquico [23], pero su alta verbosidad introduce una sobrecarga drástica de ”ruido” que no aporta nada al modelo de lenguaje.
- **JSON (*JavaScript Object Notation*)**: Formato ligero de intercambio de datos elegido para la comunicación con el módulo de modelo de lenguaje. Estandarizado por ECMA-404 [24], su estructura de pares clave-valor reduce drásticamente el número de tokens necesarios para representar la misma información que en XML.

El proceso de parseo y normalización de los datos de cada una de las vulnerabilidades transforma la salida bruta en XML a un objeto JSON optimizado que elimina las etiquetas irrelevantes, optimizando la precisión de la respuesta del modelo de lenguaje y el coste operativo de la llamada a la API. Además, se realiza un procedimiento de fragmentación en el que una vez que el informe de OpenVAS es parseado y normalizado, este se divide en múltiples archivos JSONs independientes (uno por cada vulnerabilidad detectada). Esta división se produce por dos motivos: El primero es que si enviásemos el reporte completo, el volumen de texto sobrepasaría el límite de tokens que podemos enviar al modelo de lenguaje. El segundo motivo, es que de esta manera permitimos que el agente procese cada vulnerabilidad de forma secuencial y aislada.

2.4.3 Protocolos de Comunicación Segura

La ejecución en remoto de los scripts de seguridad generados por el modelo de lenguaje requiere un canal de comunicación seguro que garantice la confidencialidad, integridad y autenticidad de los datos en tránsito.

Secure Shell (SSH)

Secure Shell (SSH) es un protocolo definido en el RFC 4251 [25]. Es el estándar industrial más conocido para la administración remota segura. Proporciona un canal cifrado sobre una red insegura usando una arquitectura cliente - servidor. Este protocolo se ha seleccionado en la solución planteada para realizar la ejecución remota de los scripts de mitigación de vulnerabilidades en la máquina objetivo.

Secure Copy Protocol (SCP)

Secure Copy Protocol es un protocolo que opera sobre el túnel SSH para la transferencia segura de datos. Es el protocolo seleccionado en el proyecto para la transmisión de los scripts desde la máquina escáner hasta cada una de las máquinas objetivo.

Autenticación desatendida mediante Claves Rivest-Shamir-Adleman (RSA)

Para lograr una automatización completa del pipeline sin necesidad de intervención humana para introducir contraseñas, es necesario implementar un mecanismo de autenticación. Para el proyecto, se ha seleccionado un mecanismo de criptografía asimétrica mediante claves RSA para este propósito.

El algoritmo Rivest-Shamir-Adleman (RSA) es un algoritmo criptográfico asimétrico que proporciona confidencialidad de la información en la transmisión de datos entre usuarios basándose en la complejidad de cálculo que tiene encontrar dos factores primos de un número compuesto muy grande. RSA emplea un par de claves en su funcionamiento: Una clave pública, que puede ser distribuida libremente (en este proyecto, a cada una de las máquinas objetivo), y una clave privada, que debe ser secreta y custodiada por el propietario de dicha clave. En nuestro contexto, la clave privada representa la identidad de la máquina orquestadora, y las máquinas objetivo pueden validar las acciones en remoto de la máquina orquestadora con la clave pública [26].

Flujo de despliegue y ejecución

Una vez establecidos los protocolos de transferencia y el mecanismo de autenticación desatendida, es necesario implementar un usuario de administración distinto al usuario 'root' con permisos para poder ejecutar sudo sin contraseña, de forma que la máquina orquestadora y las máquinas objetivo puedan interactuar automáticamente sin la necesidad de introducir contraseñas.

El mecanismo sudo permite la ejecución de comandos con privilegios de otro usuario, el cual por defecto es el usuario 'root' del sistema. Mediante directivas como `NOPASSWD` en su archivo de configuración, permitimos la automatización que el proyecto requiere sin interrumpir el flujo con peticiones de contraseña. Para que esto no suponga un problema de seguridad, es por ello que solo habilitamos esta opción en un usuario de administración diferente a 'root' creado especificamente para este proyecto para cada una de las máquinas objetivo.

2.5 Entorno de pruebas y desarrollo

El desarrollo del proyecto y la validación de herramientas de ciberseguridad ofensiva conlleva posibles riesgos si se realizan sobre sistemas en producción. Es por ello que se ha diseñado una infraestructura virtualizada que permita la ejecución segura de los scripts y la modificación de configuraciones de red sin que haya un impacto real en el entorno local.

2.5.1 Virtualización de sistemas

La virtualización es una tecnología que permite la creación de entornos virtuales llamados máquinas virtuales (VM) a partir de una única máquina física, lo que permite la abstracción de los recursos. [27]. La virtualización proporciona dos capacidades críticas para el desarrollo de la solución:

- **Aislamiento de Red**: Las máquinas virtuales se configuran en un segmento de red privado para garantizar que el tráfico de escaneo y los scripts no afectan a redes externas.
- **Gestión de Estados (Snapshots)**: Una snapshot es una captura del estado de una máquina virtual en un punto del tiempo que permite volver a ese estado si el usuario lo requiere. Esto permite revertir el sistema objetivo a su estado vulnerable original tras cada prueba de mitigación, asegurando la reproducibilidad de los experimentos [28].

Un hipervisor es una capa de software fundamental que permite la virtualización aislada en una única máquina física [29]. Según Gerald J. Popek y Robert P. Goldberg, como se especifica en su artículo Formal Requirements for Virtualizable Third Generation Architectures de 1974 [30], hay dos tipos de hipervisores:

- **Hipervisores de tipo 1**: Se ejecutan directamente en el servidor y gestionan el sistema o sistemas operativos invitados. El software de virtualización se instala directamente en el hardware.
- **Hipervisores de tipo 2**: Se ejecutan como una capa de software por encima del sistema operativo de la máquina host. Son usados para abstraer los sistemas operativos invitados del sistema operativo principal.

Para este proyecto, se ha utilizado el hipervisor de Tipo 2 VMware Workstation Pro.

2.5.2 Componentes del entorno de pruebas

La arquitectura del laboratorio se compone de dos nodos principales con roles diferenciados.

Kali Linux (Máquina Orquestadora/ Escáner)

Kali Linux es una distribución basada en Debian GNU/Linux, desarrollada por Offensive Security. Es considerada el estándar para pruebas de penetración y auditoría de seguridad [31]. En este proyecto, la máquina virtual Kali Linux actúa como el nodo central u ”orquestador”, alojando:

- El módulo de escaneo de vulnerabilidades (OpenVAS).
- El módulo de orquestación Python (`agent.py`).
- Las claves privadas SSH para la conexión con las máquinas objetivo.

Metasploitable 2 (Máquina Objetivo)

Metasploitable 2 es una máquina virtual basada en Ubuntu Linux (exactamente la versión 8.04) diseñada intencionalmente con múltiples vulnerabilidades de seguridad sin parchear [32].

Se ha elegido la imagen de Metasploitable 2 como máquina objetivo por dos razones:

1. Contiene decenas de vulnerabilidades en múltiples capas (como por ejemplo en servicios web, bases de datos o configuraciones de red), lo que permite evaluar la capacidad del módulo de modelo de lenguaje para generar scripts de mitigación heterogéneos (no solo para un tipo de fallo).
2. Al ser un sistema vulnerable conocido, nos permite verificar si el escaneo de vulnerabilidades realizado concuerda con la realidad.

Capítulo 3

Estado de la Cuestión

El propósito de este capítulo es analizar el contexto actual del campo de la ciberseguridad en relación con la propuesta planteada. Para ello, realizaremos un análisis de las herramientas existentes con las que se aborda el problema planteado en este Trabajo de Fin de Grado.

3.1 La ciberseguridad en la actualidad

En la actualidad, la seguridad digital se encuentra con un problema paradójico: A medida que aumentan las inversiones en seguridad sobre los activos tecnológicos de las empresas y corporaciones, el éxito de los ciberataques sobre dichos activos no disminuye. En el informe *Threat Landscape 2023* de la ENISA (Agencia de la Unión Europea para la Seguridad) se destaca que la profesionalización del cibercrimen y la disponibilidad de mercados de *Crime-as-a-Service* (también denominado CaaS) han aumentado las vías de ataque para los atacantes, aumentando tanto el volumen como la sofisticación de los ciberataques [33].

Por otro lado, podemos ver en el Informe de Defensa Digital de Microsoft del 2023 que uno de los factores que han aumentado drásticamente la frecuencia de los ciberataques es la automatización [34]. Esto reduce mucho el tiempo entre que se descubre una vulnerabilidad hasta que esta es explotada masivamente. Esta diferencia de velocidad entre la ofensiva de los atacantes y la defensa de las organizaciones constituye un problema grave.

En el mismo informe podemos ver que los analistas de Microsoft también consideran que la automatización con LLMs tiene un gran potencial para ayudar a las organizaciones a establecer sistemas de recuperación y respuesta ante incidentes o de monitorización y detección de incidentes, entre otros. Esto es un gran motivador para que las empresas de ciberseguridad no se queden atrás, y usen la automatización para aumentar la velocidad de resolución de vulnerabilidades.

3.2 Análisis de las herramientas existentes

El mercado de herramientas para abordar este problema planteado en este documento se divide en dos grandes segmentos:

3.2.1 Plataformas comerciales

Las soluciones líderes del mercado, como son Qualys VMDR o Tenable.io, han evolucionado para integrar con automatización la detección con la respuesta. Estas plataformas permiten visualizar las vulnerabilidades y aplicar parches automáticamente en algunos sistemas operativos [8, 35].

Por otro lado, plataformas SOAR (*Security Orchestration, Automation and Response*) como Palo Alto Cortex XSOAR permiten crear *playbooks* para automatizar respuestas ante incidentes [36].

El problema es que, por muy buenas que suenen estas soluciones, la limitación del elevado coste y complejidad de configuración en la mayoría de casos supone una inversión inasumible para muchas PYMES o redes de laboratorio. Además, estos suelen ser entornos cerrados muy complejos de personalizar fuera de sus integraciones oficiales.

3.2.2 Herramientas de código abierto

En el ámbito *open-source*, OpenVAS (*Greenbone Vulnerability Management*) es el estándar de este tipo de herramientas de detección de vulnerabilidades. Esta herramienta ofrece una capacidad de detección de nivel empresarial, con una base de datos muy completa de NVTs (*Network Vulnerability Tests*) actualizada diariamente con la última información disponible.

En este caso, la limitación es que OpenVAS, tal y como detalla en su documentación oficial, es una herramienta de diagnóstico. Su funcionamiento se basa en reportar en un informe las vulnerabilidades presentes de una máquina objetivo, pero la herramienta carece de módulos para ejecutar acciones correctivas. Esto obliga a un administrador humano a actuar manualmente sobre cada incidencia, ralentizando drásticamente el ciclo de defensa.

3.3 Inteligencia artificial en operaciones de seguridad (SecOps)

El auge de los LLMs (*Large Language Models*) está transformando el sector. Herramientas como Microsoft Security Copilot comenzaron a usar modelos de inteligencia artificial para asistir a los ciberanalistas en la comprensión de incidentes de seguridad o sugiriendo scripts de mitigación [37]. Su uso en la ciberseguridad facilita mucho las tareas de automatización de detección y resolución de incidentes.

No obstante, estas implementaciones siguen formando parte de plataformas comerciales con un coste muy elevado. En este contexto podemos encontrar un vacío entre las plataformas comerciales y las herramientas de código abierto: Combinar el poder de detección de una herramienta de código abierto como OpenVAS con el poder de resolución que un LLM nos podría proporcionar para cerrar el ciclo de remediación completo.

3.4 Justificación de la solución propuesta

Este Trabajo de Fin de Grado cubre una necesidad no resuelta en el ámbito *open-source*: La orquestación de remediación mediante LLMs. Si combinamos la capacidad de detección de la API OpenVAS con la capacidad de generación de código de la API de OpenAI, podemos conseguir una alternativa de defensa activa mucho más accesible y auditable que las costosas suscripciones de plataformas comerciales.

Capítulo 4

Metodología y plan de proyecto

En este capítulo se documenta cuál ha sido la metodología usada durante la realización del proyecto, detallando cual ha sido el modo de trabajo para cada una de las fases en las que se ha dividido el proyecto. Además, se realiza una estimación detallada de los costes y materiales asociados al proyecto y del tiempo empleado (ilustrado mediante un diagrama de Gantt).

4.1 Metodología del proyecto

Dado que el proyecto combina varias herramientas y tecnologías diferentes, se ha optado por seleccionar una metodología iterativo-incremental que permita fragmentar el flujo de la solución en varios módulos entregables que aporten valor desde las primeras iteraciones. Cada uno de los módulos tuvo una fase de análisis previo, diseño, desarrollo e implementación y pruebas de funcionamiento para poder asegurar un incremento fiable antes de avanzar con el siguiente. [38].

Como resultado de la decisión de aplicar esta metodología, el trabajo ha sido fragmentado en varios incrementos que corresponden a las partes más significativas del pipeline. Estos se enumeran a continuación y pueden visualizarse de forma esquemática en la Figura 4.1.

1. **Módulo de modelos de lenguaje**: En esta primera fase, el esfuerzo se centró en desarrollar un agente en Python que trabaje con el API de un LLM (en este caso, de OpenAI). El objetivo es establecer la ingeniería de prompts necesaria para poder interpretar las descripciones en formato JSON de las vulnerabilidades y generar scripts de mitigación en Bash que solucionen cada una de ellas.

2. **Módulo de escaneo**: El objetivo de esta fase era completar toda la configuración relativa al escáner de vulnerabilidades (es decir, OpenVAS) en la máquina escáner. Esto supondría configurar en la interfaz todas las tasks y targets necesarios, desarrollar en

el agente Python un control de OpenVAS en local mediante gvm-cli, obteniendo automáticamente el reporte una vez finalizado el escaneo.

3. **Módulo de normalización e integración**: Esta fase es el nexo entre el módulo de escaneo y el módulo de modelos de lenguaje y mitigación. El objetivo es el desarrollo de un parser avanzado que procese los informes en formato XML generados por OpenVAS filtrando el ruido y estructurando la información crítica para optimizar la entrada al modelo de lenguaje. También se incluye el desarrollo de un splitter, dedicado dividir el informe ya parseado por cada una de las vulnerabilidades que hay en él para que el módulo de modelo de lenguaje genere scripts de mitigación independientes para cada una de ellas.

4. **Módulo de despliegue y conectividad**: Como última fase, se abordó la infraestructura de comunicación con los sistemas de la red empresarial virtualizada. En esta fase se incluye la configuración de los protocolos SSH y SCP, implementando un sistema de autenticación mediante pares de claves RSA para asegurar la automatización. Con esta fase se cierra el pipeline, consiguiendo unir desde la extracción de las vulnerabilidades de la red empresarial, hasta la ejecución en remoto por SSH de los scripts de mitigación en las máquinas objetivo.

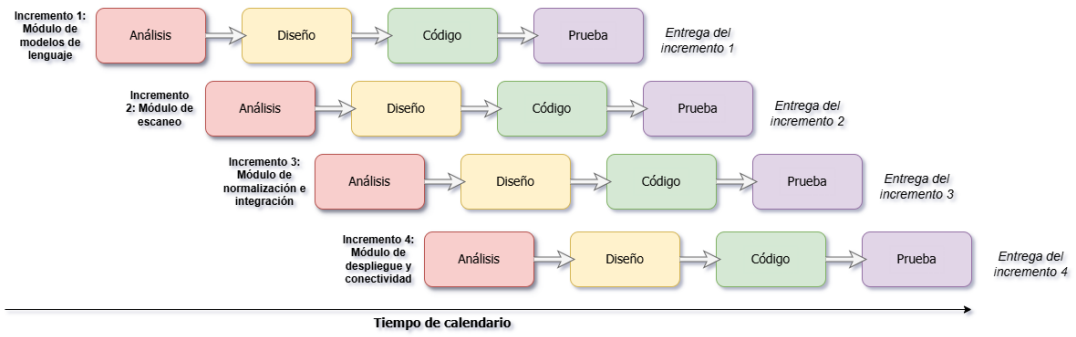


Figura 4.1: Metodología iterativo-incremental aplicada al proyecto. Elaboración propia.

Para poder garantizar la trazabilidad e integridad de los recursos tratados durante el proyecto, se empleó Git como sistema de control de versiones. Se gestionó mediante un repositorio privado local todo el código fuente del proyecto, incluyendo los scripts Python, configuraciones de despliegue y la propia documentación técnica. Esta decisión resultó fundamental para la gestión de errores, ya que permitió revertir cambios inestables donde podría haberse perdido gran parte del trabajo.

Cabe destacar que para la implementación de este proyecto se han usado modelos de lenguaje como herramientas de asistencia en el desarrollo de software. Más concretamente,

se ha usado Gemini para la generación de estructuras base de funciones y múltiples comandos usados en diferentes fragmentos del código, siendo toda la lógica final, revisada, validada y probada de forma manual por el autor principal.

Además, las validaciones de cada uno de los módulos se realizaron mediante pruebas modulares aisladas al finalizar cada incremento. Por ejemplo, para el módulo de escaneo, la correcta devolución por parte de OpenVAS de un reporte XML con todas las vulnerabilidades esperadas en local. Otro ejemplo también, para el módulo de modelos de lenguaje, el introducir una vulnerabilidad cualquiera como entrada y que como salida saliese un script de mitigación que cumpliese todas las condiciones.

4.2 Plan de proyecto

Para poder estimar de manera inicial las horas de trabajo esperadas en cada una de las fases del proyecto, se tomó como referencia la carga de trabajo asignada a la asignatura de Trabajo de Fin de Grado. Dicha carga de trabajo está definida por el número de créditos ECTS (*European Credit Transfer and Accumulation System*), que en el caso de esta materia son 12 [39]. Según la normativa de la Universidade da Coruña y siguiendo el estándar del Espacio Europeo de Educación Superior, 1 crédito ECTS equivale a 25 horas de trabajo del estudiante, lo que nos da un total de 300 horas.

Conociendo este dato, se realizó un reparto de horas entre las fases y se elaboró un diagrama de Gantt (véase la Figura 4.2) para lograr fácilmente una visualización del calendario sencilla, permitiendo establecer márgenes de ajuste en el caso de que se diesen posibles desviaciones, las cuales fueron registradas de manera controlada, procurando mantener igualmente los hitos clave del Trabajo de Fin de Grado [40]. Dicha planificación y sus desvios se pueden consultar en la Tabla 4.1.

Código	Nombre	Estimación inicial (h)	Duración Final (h)
F1	**Estudio inicial**	**75**	**75**
F1.1	Estudio de la metodología y planificación	20	25
F1.2	Estudio del estado de la cuestión	20	15
F1.3	Estudio de los fundamentos teóricos	35	35
F2	**Desarrollo e implementación**	**150**	**155**
F2.1	Incremento de modelos de lenguaje	40	40
F2.2	Incremento de escaneo	45	50
F2.3	Incremento de normalización-integración	25	25
F2.4	Incremento de despliegue y conectividad	40	40
F3	**Análisis de resultados y conclusiones**	**10**	**10**
F4	**Documentación**	**65**	**80**
	TOTAL	**300**	**320**

Tabla 4.1: Estimación inicial de la duración de las fases del proyecto frente a la duración final real.

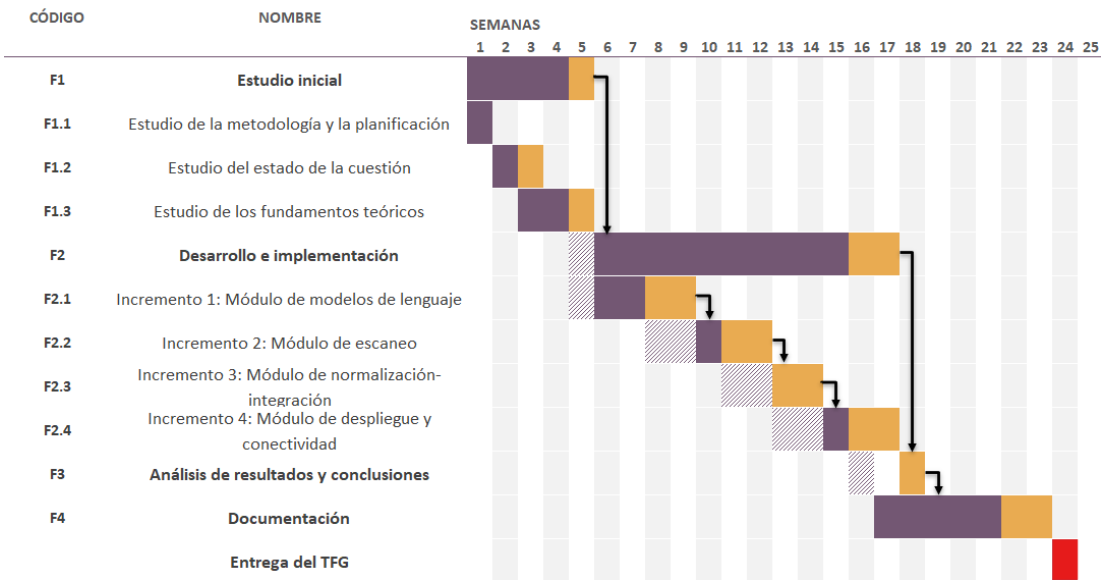


Figura 4.2: Diagrama de Gantt del proyecto. Blanco con diagonales: Duración estimada del plan; Morado: Duración real del plan dentro del plan; Amarillo: Duración real del plan fuera del plan; Rojo: Hitos del proyecto.

4.3 Recursos

Durante la realización del Trabajo de Fin de Grado, se usaron diferentes herramientas, recursos y materiales para poder alcanzar la solución propuesta

4.3.1 Hardware

Portátil

- **Uso**: Entorno hardware de estudio, desarrollo, análisis y documentación del proyecto.
- **Características**:
  - Modelo: *PcCom Revolt*
  - Sistema operativo: *Microsoft Windows 11 Home 10.0.26200*
  - Procesador: *Intel(R) Core(TM) Ultra 7 155H*
  - RAM: *16 GB*
  - Disco: *NVMe WD Blue SN580 1TB*
  - Gráfica: *NVIDIA GeForce RTX 4060*

4.3.2 Software

Lenguajes de programación

- **Python 3.13.17**: Lenguaje de programación usado para el desarrollo de los scripts necesarios para implementación de la solución [21].
- **Bash**: Usado para la transmisión y ejecución en remoto de los scripts de mitigación de la máquina orquestadora a las máquinas objetivo [41].

Entornos de virtualización

- **VmWare Workstation 17.5.2 Pro**: Plataforma de virtualización tipo 2 usada para el despliegue de la red empresarial (lo que incluye las máquinas virtuales mencionadas a continuación), proporcionando un entorno de laboratorio seguro para el desarrollo del proyecto [42].

4.3.3 Imágenes de máquinas virtuales

- **Metasploitable 2**: Máquina virtual con decenas de vulnerabilidades de forma intencionada, seleccionada como objetivo para los escaneos de vulnerabilidades y pruebas de funcionamiento del pipeline implementado [32].

- **Kali Linux**: Máquina virtual con distribución basada en Debian orientada a la ciberseguridad, usada como máquina orquestadora y escáner en el proyecto [31].

4.3.4 Entorno de desarrollo

- **Visual Studio Code**: Principal editor de código que usa las extensiones de Python y Git, para el desarrollo e implementación de los programas necesitados para la solución del proyecto [43].

4.3.5 Servicios

- **OpenAI API Platform**: Plataforma de servicios de IA usado para la configuración de los LLMs usados durante el desarrollo del proyecto [44].
- **Greenbone Vulnerability Manager (GVM)**: Framework usado como escáner de vulnerabilidades sobre las máquinas objetivo [11].

4.3.6 Control de Versiones

- **Github**: Plataforma de alojamiento usada para el control de versiones del código y documentación desarrollados [45].

4.3.7 Librerías

- **openai (1.99.9)**: Librería oficial de Python usada en el entorno virtual para interactuar con los LLMs de OpenAI, permitiendo la entrada de vulnerabilidades y un prompt, y la salida de los scripts de mitigación. [46].
- **gvm-cli (gvm-tools) (25.3.0)**: Interfaz de comando para *Greenbone Vulnerability Manager* (GVM), usada para la comunicación con el framework desde el agente orquestador [47].
- **rich (13.9.4)**: Librería usada para conseguir un entorno de comando profesional durante la ejecución del pipeline del proyecto, permitiendo el uso de barras de estado, tablas de resultado y *spinners* de carga, entre otros. [48]
- **gum (0.17.0)**: Herramienta en línea de comandos que, al igual que Rich, permite el uso de componentes visuales e interactivos para la configuración de un entorno de comando profesional.

4.4 Costes

Aunque el proyecto se ha realizado en un ámbito académico usando licencias educativas u *opensource*, debemos calcular los costes reales que supondría el desarrollo completo en un entorno profesional del proyecto planteado.

4.4.1 Coste del personal

Según el Estudio de la plataforma de empleo Glassdoor [49], el salario medio de un Ingeniero de Software con experiencia en ciberseguridad y automatización es de 42.000 € al año. Considerando una jornada estándar de 40 horas semanales para un total de 1664 horas anuales, esto supone un salario de 25,24 €/h de salario bruto.

Para conseguir el salario real, debemos aplicar los impuestos y gastos adicionales relacionados con la Seguridad Social, para lo cual se ha usado la calculadora de costes del Banco Santander [50]. Según la fuente, aplicando un 23,6% en contingencias comunes, un 5,5% en contingencias profesionales, un 5,5% en prestación por desempleo (se asume un contrato indefinido), un 0,2% al FOGASA (Fondo de Garantía Salarial), un 0,6% a formación y un 0,9% del MEI (Mecanismo de Equidad Intergeneracional), el coste total del empleado para la empresa asciende a 57.245 € al año.

Por otro lado, un *Product Owner* (supervisor técnico encargado de la validación de requisitos) tiene un salario medio de 60.000 € al año [51]. Aplicando los mismos costes empresariales mencionados para el Ingeniero de Software, el coste anual para la empresa asciende a los 81.780 € anuales. Es decir, el Ingeniero de Software tendría un coste real de 34,40 €/h, y el *Product Owner* de 49,14 €/h.

Partiendo de estos cálculos y teniendo en cuenta que el proyecto ha sido realizado siguiendo una metodología iterativa-incremental, con un Ingeniero de Software dedicado a tiempo parcial (estimando unas 20 horas semanales de media para el desarrollo del proyecto) y un *Product Owner* presente en las reuniones de planificación, hitos y validación final, podemos visualizar en la Tabla 4.2 los costes obtenidos:

Personal	**Coste (€)**
Ingeniero de Software (320 h x 34,40 €/h)	11.008,00
Product Owner (10 h x 49,14 €/h)	491,40
Coste total estimado	**11.499,40**

Tabla 4.2: Costes estimados del personal según su salario y dedicación.

4.4.2 Coste de los recursos

Debemos considerar los gastos de los recursos de hardware y de software usados durante el desarrollo en un entorno profesional, los cuales se incluyen en este apartado.

Se ha usado un portátil de alto rendimiento necesario para la virtualización del entorno de pruebas y la ejecución del flujo de trabajo eficazmente. Se trata de un portátil PcCom Revolt con procesador Intel Core i7 Ultra, 16 GB de RAM y 1TB de almacenamiento SSD. Por otro lado, debemos contar con el coste del consumo de tokens de la API de OpenAI durante las fases de desarrollo y validación.

Los costes estimados totales del proyecto, sumando los costes del personal calculados en el apartado anterior y los recursos utilizados, se detallan en la Tabla 4.3.

Concepto	**Coste total (€)**
Coste del personal	11.499,40
Portátil PcCom Revolt	1.600,00
OpenAI API (*gpt-5-mini*)	7,28
Coste total	**13.106,68**

Tabla 4.3: Coste total estimado del proyecto.

Según las estimaciones que hemos realizado, las 24 semanas de desarrollo de la solución planteada en este Trabajo de Fin de Grado alcanzarían un coste de 13.106,68 €.

Capítulo 5

Desarrollo e Implementación

En este capítulo del documento se detalla la implementación de la solución propuesta para el problema planteado en este Trabajo de Fin de Grado. El desarrollo detalla la integración de OpenVAS como herramienta de escaneo de vulnerabilidades en el flujo de trabajo, la implementación de un agente que interactúe con la API de OpenAI para la generación de scripts de mitigación y su transmisión y ejecución remota. Además, se desarrolla la forma en la que se automatizó el pipeline y se conectaron los diferentes módulos para hacer de ellos un único script funcional.

Para ofrecer una visión global antes de profundizar en la implementación técnica de cada módulo, la Figura 5.1 presenta el flujo completo del sistema. Este esquema tiene como objetivo permitir al lector situar la interconexión de cada uno de los componentes (desde el escaneo de vulnerabilidades inicial hasta la ejecución remota de los scripts de mitigación) que desglosaremos a continuación.

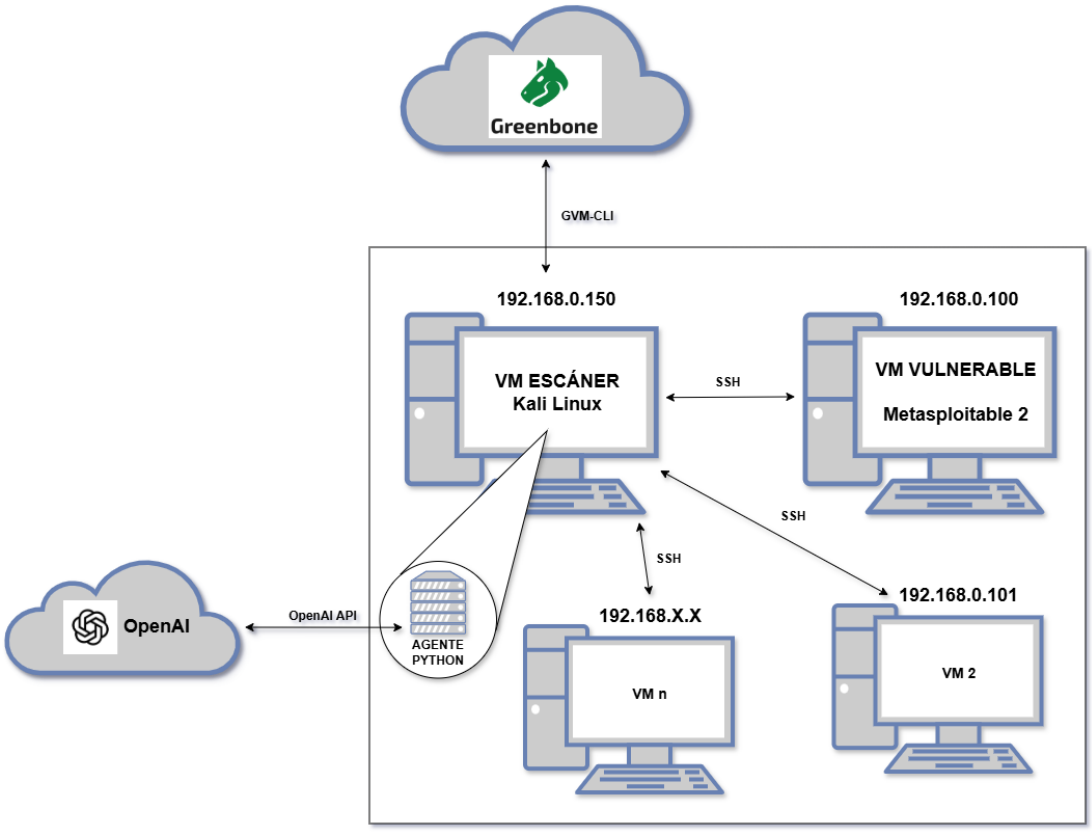


Figura 5.1: Arquitectura del flujo de trabajo del sistema de mitigación automatizado.

5.1 Implementación del entorno de desarrollo

Antes de comenzar con el desarrollo de la solución propuesta, debemos crear un entorno de laboratorio controlado y cerrado en el que poder realizar tanto la implementación como las pruebas de concepto (PoC) de manera segura, garantizando el aislamiento respecto a otras redes externas o de producción.

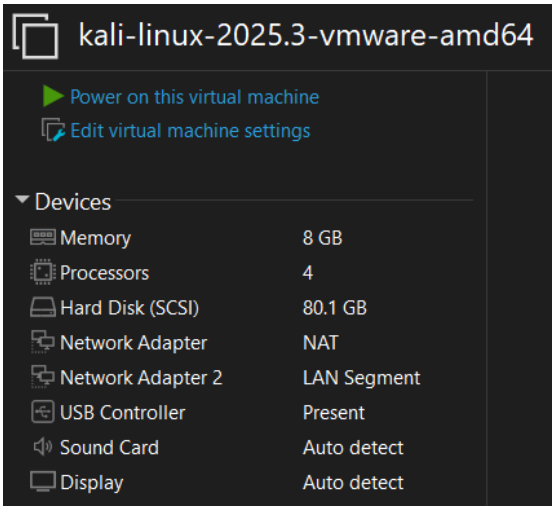
Para ello, crearemos una red empresarial de laboratorio usando el hipervisor VMware Workstation 17 Pro, que estará compuesta de:

- **Máquina orquestadora (*Scanner*)**: Una máquina virtual con la última imagen de Kali Linux. Esta máquina actuará como el nodo de administración de la red, encargada de la orquestación del flujo de trabajo.
- **Máquinas objetivo (Targets)**: Un conjunto de máquinas virtuales vulnerables intencionalmente usando la imagen de Metasploitable 2 que simularán los activos vulnerables de la red empresarial. Se ha seleccionado la imagen de Metasploitable 2 porque, gracias

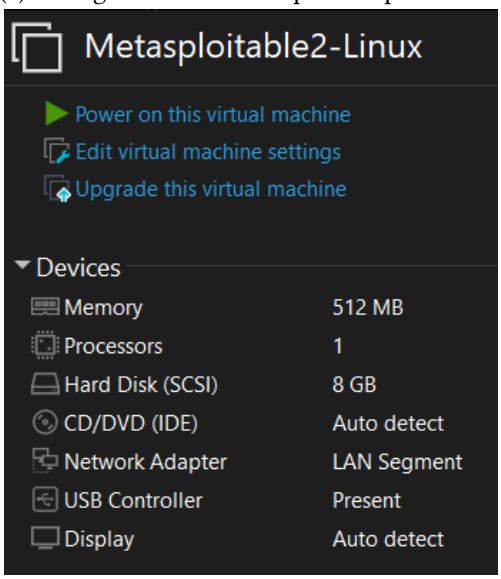
a su naturaleza vulnerable, permitirá validar la eficacia de los scripts de mitigación generados. Para simplificar el desarrollo de la solución inicialmente, usaremos una única máquina objetivo, siendo la arquitectura totalmente escalable a múltiples equipos si se diese el caso.

- **Segmento LAN**: Se ha realizado la interconexión de las máquinas en VMware mediante un segmento LAN privado. Es una solución ideal en nuestro caso, ya que nos permite simular un switch virtual desconectado del equipo local e internet, resultando en un entorno realista y seguro.

A continuación, en las Figuras 5.2a y 5.2b se muestra la configuración aplicada para cada una de las máquinas virtuales de la red.



(a) Configuración de la máquina orquestadora.



(b) Configuración de la máquina objetivo.

Figura 5.2: Configuración de las máquinas del entorno de desarrollo.

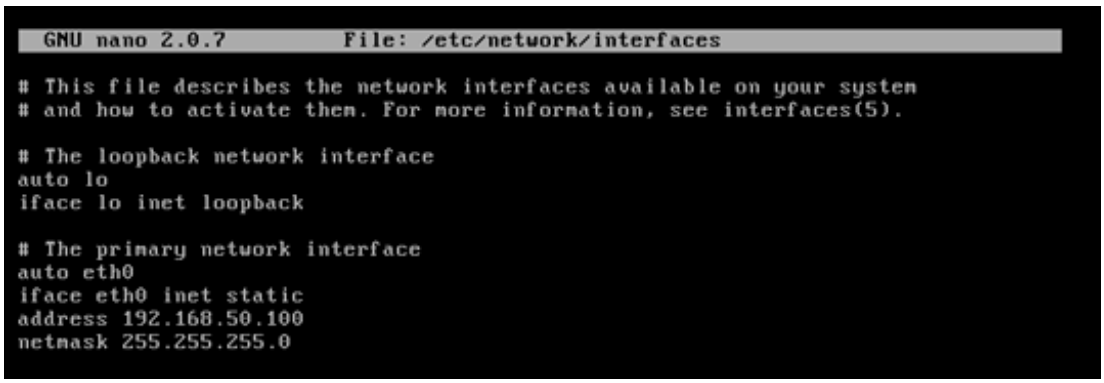
5.1.1 Configuración de red y direccionamiento

Al usar un segmento LAN, no dispondremos de un servidor DHCP, así que se ha diseñado un plan de direccionamiento IP estático para garantizar la conectividad e identificación de las máquinas en la red (véase Tabla 5.1).

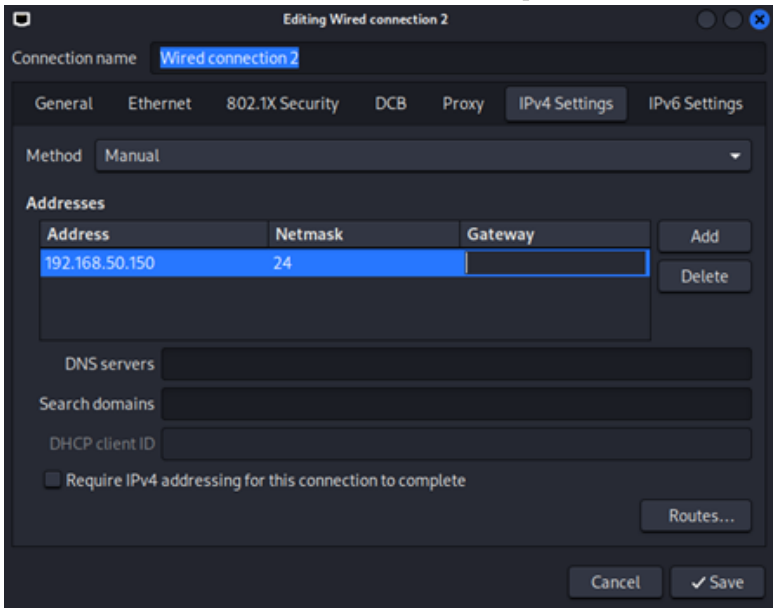
Rol	Sistema Operativo	Dirección IP Asignada
Orquestador	Kali Linux	192.168.50.150
Objetivo	Metasploitable 2	192.168.50.100

Tabla 5.1: Mapa de direccionamiento estático del entorno de laboratorio en VMware.

Para realizar este direccionamiento, se configuró el fichero /etc/network/interfaces en la máquina objetivo (véase la Figura 5.3a y se ajustó la configuración de red manual en la máquina orquestadora (véase la Figura 5.2a).



(a) Direccionamiento estático de la máquina objetivo.



(b) Direccionamiento estático de la máquina orquestadora

Figura 5.3: Configuración del direccionamiento estático en el entorno de desarrollo.

Tal y como podemos ver en la Figura 5.4, el entorno de laboratorio resultante permite una

comunicación directa entre la máquina orquestadora y las máquinas objetivo.

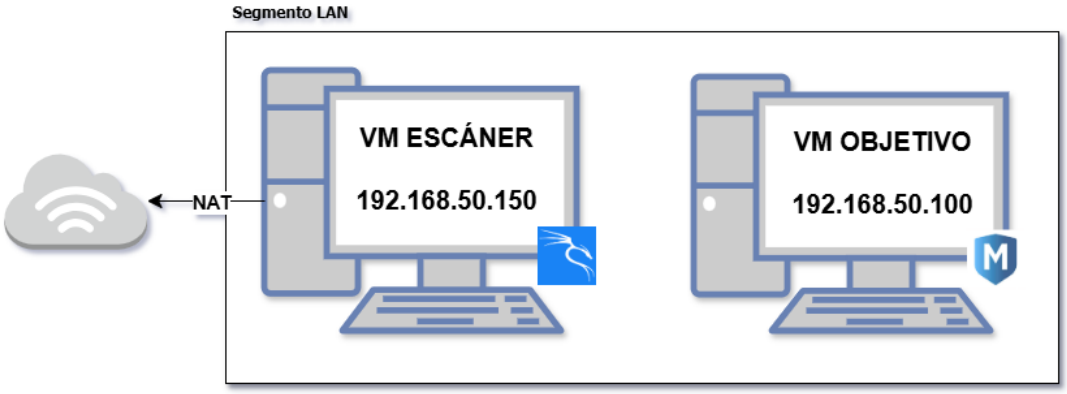


Figura 5.4: Entorno de laboratorio usado para el desarrollo del proyecto.

Por último, es necesario señalar que la máquina orquestadora será la única que dispondrá de acceso directo a Internet mediante una interfaz NAT. Se ha implementado esta decisión de diseño puesto que de esta manera se permite al sistema realizar las peticiones a la API de OpenAI y descargar paquetes o actualizaciones necesarias a la vez que se dejan completamente aisladas del exterior las máquinas objetivo.

5.1.2 Gestión de estados con *snapshots*

Como medida de seguridad y control de seguridad durante el desarrollo del proyecto, se realizó una **snapshot** de la máquina objetivo en el hipervisor VMware, permitiendo capturar el estado de dicha máquina antes de cada ejecución del pipeline para poder revertir el sistema a su estado inicial vulnerable en cualquier momento (véase la Figura 5.5). Esto facilita la validación de las mitigaciones y garantiza que los resultados de las pruebas puedan ser reproducibles e independientes entre sí.

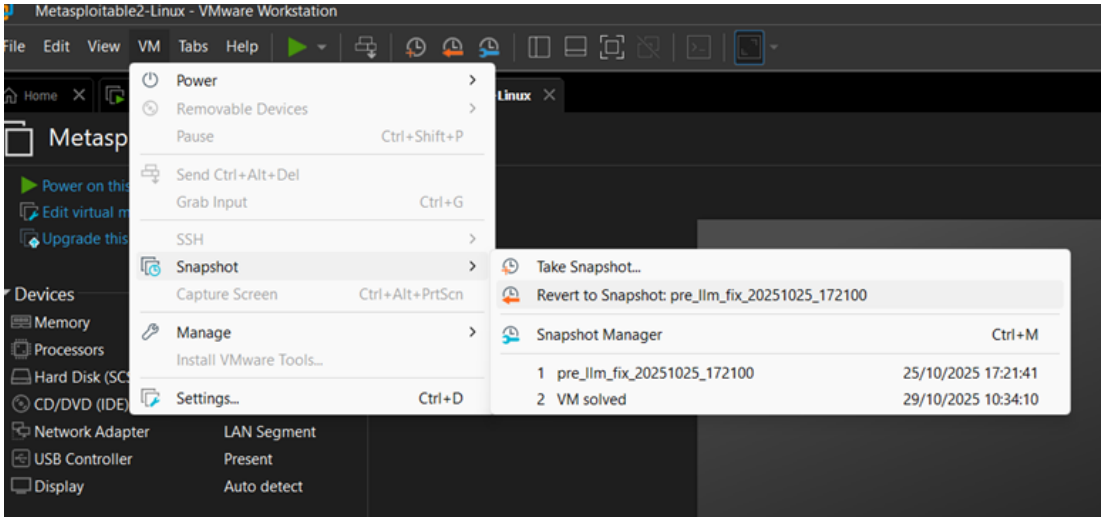


Figura 5.5: Captura de la *snapshot* realizada desde el virtualizador VMware

5.2 Módulo de modelos de lenguaje

El principal objetivo de este primer incremento es desarrollar un agente en Python que pueda trabajar con la API de un LLM. Como LLM se ha seleccionado OpenAI por su gran capacidad de generación de código y, además, por ser el estándar de facto actual.

Debido a que el uso de esta API no es gratuito [52], se asignó un presupuesto inicial de 10,00 € al proyecto (de los cuales solo fueron necesarios 7,28 € para el desarrollo completo del proyecto). Para optimizar el consumo de saldo lo máximo posible, se implementó una estrategia de desarrollo escalonada:

- **Fase de desarrollo**: En esta fase ha usado el modelo ”gpt-4o-mini” de OpenAI [53]. Se eligió este modelo por su equilibrio entre bajo coste y alta velocidad para poder realizar cientos de ejecuciones, acelerando el proceso de implementación.
- **Fase final**: Para la versión definitiva del agente se configuró el modelo ”gpt-5-mini” de OpenAI [18]. Se cambió a este modelo dado que, aunque es casi el doble de caro que el ”gpt-4o-mini”, su capacidad de razonamiento es drásticamente superior. Esto permite al agente generar scripts de mitigación más complejos y seguros, con una tasa de éxito mucho más elevada.

El fichero principal desarrollado en este incremento es el fichero `agent.py`, que contendrá la lógica para desempeñar las siguientes funciones, las cuales también pueden ser visualizadas en el diagrama de secuencia de la Figura 5.6.

1. Recibir como parámetro de entrada un JSON (o un conjunto de JSONs) con la descripción de las vulnerabilidades (un JSON por cada vulnerabilidad).
2. Construcción y envío de una petición POST a la API de OpenAI, donde se incluirá la clave de autenticación API Key (obtenida anteriormente al registrarse en la OpenAI API Platform [44]), el prompt personalizado del sistema y el JSON de la vulnerabilidad como *payload*, entre otros.
3. Generación de un script de mitigación como salida y, dependiendo de los argumentos de ejecución del fichero `agent.py`, guardarlo en un fichero `.sh` o mostrarlo por pantalla.

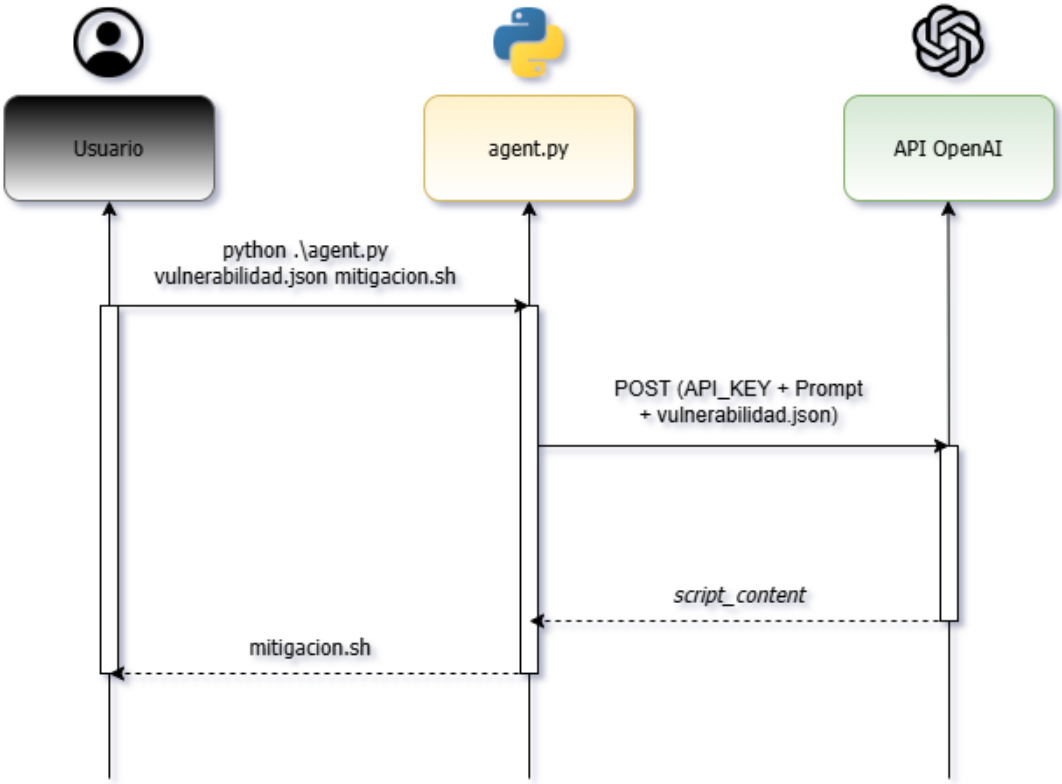


Figura 5.6: Funcionamiento del agente Python (`agent.py`)

5.2.1 Preparación del entorno

Antes de comenzar con la implementación, debemos preparar un entorno virtual de Python [22] para poder aislar las dependencias del proyecto, evitando posibles conflictos con las librerías del sistema operativo local (Kali Linux). A continuación, deberemos instalar las librerías usadas por el agente.

```
# Creación del entorno virtual 'venv'
python -m venv venv

# Activación del entorno virtual
source /venv/bin/activate

# Instalación de las librerías necesarias
pip install openai
pip install rich

```

Una vez activado el entorno, se creó el fichero `agent.py` y se le asignaron permisos de ejecución:

```
touch agent.py
chmod +x agent.py

```

Para poder interactuar con el LLM, es necesario generar desde la plataforma web de OpenAI API una credencial de acceso registrándose en ella [44]. Una vez realizado esto se nos generará una *API Key* con formato `sk-proj-....`

Por motivos de seguridad y para evitar riesgos de filtraciones de claves, nunca debemos almacenar claves en texto plano dentro del código fuente si el código es subido a un repositorio público. Es por ello que se almacenó la credencial en una variable de entorno en la sesión del terminal:

```
export OPENAI_API_KEY="sk-proj-MICLAVE"

```

5.2.2 Implementación del fichero `agent.py`

Una vez preparado el entorno y establecida la autenticación para la API de OpenAI, se procedió a la implementación de la lógica del agente. El código completo del agente está disponible en el repositorio de Github asociado a este Trabajo de Fin de Grado [54].

A continuación, se detallan las partes principales del agente:

Ingeniería del prompt

Para el envío de la petición POST a la API de OpenAI, es necesario diseñar un prompt muy detallado para que se devuelva como salida un script que mitigue la vulnerabilidad con

la mayor tasa de éxito posible en el entorno de Metasploitable 2. A continuación, se muestra una versión abreviada del prompt con las especificaciones más significativas:

```
 SYSTEM_PROMPT = textwrap.dedent("""\
Eres un generador de scripts de ciberseguridad DEFENSIVA para
Metasploitable 2 (Ubuntu 8.04, SysV init).

REGLAS PRINCIPALES:
- Implementa EXACTAMENTE la acción indicada en "solution" si es
automatizable sin interacción humana.
- Si "solution", NO esperes entrada ni inventes contraseñas:
1) Aísla el servicio inseguro de forma no interactiva y
reversible.
2) Añade una línea a `/tmp/mitigation_todo.log` explicando la
acción manual pendiente.
3) Termina con `exit 0`.

ACCESO REMOTO:
- NUNCA parar, matar, deshabilitar ni quitar del arranque el
servicio SSH/sshd.

FIREWALL:
- Puedes usar `iptables` para bloquear puertos inseguros
específicos (telnet, rlogin, VNC sin auth, etc.), siempre
comprobando idempotencia.

FORMATO DE SALIDA: SOLO un único bloque de código tipo ```bash
...``` sin texto ni explicación fuera.

```

El objetivo principal del prompt configurado es la integridad de la configuración del script de mitigación. Entre otros, se restringe la alteración del servicio SSH (para que la máquina orquestadora mantenga el control remoto de la máquina en todo momento), se señala la importancia de la idempotencia (evitando la corrupción de ficheros por ejecuciones múltiples) y se realiza un registro de las vulnerabilidades que requieren ser realizadas manualmente por un administrador (debido a que hay vulnerabilidades que requieren modificar configuraciones que usan sesiones interactivas).

Llamada a la API

La llamada a la API está implementada en la función `call_llm_chatstyle`, la cual crea un cliente autenticado con la *API Key*, enviando una lista de mensajes que contiene el prompt y el JSON con la vulnerabilidad. Por último, también se indica a la hora de realizar la

petición síncrona el modelo de OpenAI que queremos usar (en este caso, gpt-5-mini).

```
def call_llm_chatstyle(vuln_minimal: Dict[str, Any]) -> (str,
float):
# Inicializamos el cliente usando la API Key que generamos
anteriormente.
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# Construimos la lista de mensajes
messages = [
    {"role": "system", "content": SYSTEM_PROMPT},
    {"role": "user", "content": json.dumps(vuln_minimal)}
]

# Realizamos la petición síncrona
resp = client.chat.completions.create(model=MODEL,
messages=messages)

# Extraemos el contenido de la respuesta en CRUDO
return resp.choices[0].message.content

```

Parseo de la respuesta de la API

El contenido que se recibe como respuesta es posible que contenga parte de texto conversacional (típico de IAs como OpenAI) que producirían errores en el código. Para asegurar la integridad del código recibido, se ha implementado la función `extract_script_from_text` que limpia mediante expresiones regulares todo lo que no sea el script de mitigación que hemos solicitado.

```
CODE_BLOCK_RE = re.compile(r"```(?:bash|sh)?\s*(.*?)\s*```",
re.DOTALL | re.IGNORECASE)

def extract_script_from_text(text: str) -> str:
  # Busca el bloque de código Markdown
  m = CODE_BLOCK_RE.search(text)
  if m:
    return m.group(1).strip()
  # Si no hay Markdown, devuelve el texto limpio si parece un
script
  return text.strip()

```

Concurrencia

Dado que un reporte completo de OpenVAS puede contener decenas de vulnerabilidades, el procesamiento secuencial resulta inadecuado debido al tiempo de generación total para todos los scripts de mitigación necesarios. Para acortar ese tiempo, se ha implementado el uso de concurrencia basado en hilos (`ThreadPoolExecutor`) para poder paralelizar las peticiones a la API de OpenAI, reduciendo drásticamente la duración total de generación de scripts.

Por defecto, se ha implementado que el agente genere 5 scripts de mitigación simultáneamente, aunque este valor es modificable con la opción `--workers N`, siendo N el número de scripts que se desean paralelizar. [1]

```
1    with concurrent.futures.ThreadPoolExecutor(max_workers=workers)
     as executor:
2      for f in files:
3        executor.submit(process_file_task, f, out_dir, progress,
     t_id)
4
```

5.3 Módulo de escaneo

El *framework* usado para el escaneo de vulnerabilidades de las máquinas objetivo de la red empresarial es GVM (*Greenbone Vulnerability Management*) [11]. Su instalación se realizó sobre la máquina orquestadora, actualizando previamente los repositorios del sistema y desplegando los servicios necesarios.

5.3.1 Instalación y despliegue

Para la instalación y despliegue de GVM, se realizaron los siguientes comandos, lo que como resultado proporciona acceso a la interfaz web de OpenVAS (Greenbone) como se puede ver en la Figura 5.7.

```
1    sudo apt-get update && sudo apt-get upgrade -y
2    sudo apt install gvm
3    sudo gvm-setup
4    sudo gvm-start
5
```

[1] El número máximo de hilos concurrentes no está limitado por el hardware del equipo en el que se ejecute el proyecto, sino principalmente por *Rate Limits* impuestos por la OpenAI para que no ocurran bloqueos por congestión.

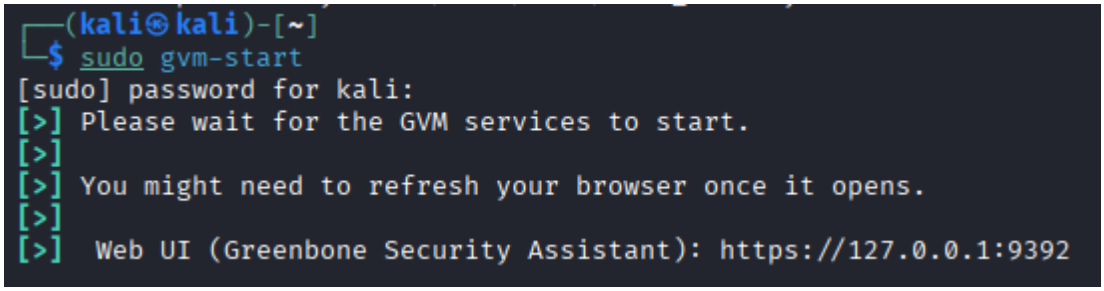


Figura 5.7: Aviso por terminal del despliegue de la interfaz web de OpenVAS

Si entramos en la URL que se nos proporciona por terminal, se nos solicitará un nombre de usuario y una contraseña (los cuales son configurados al realizar el comando `gvm-setup`). Una vez introducidas las credenciales, podremos ver la interfaz web de GVM (véase la Figura 5.8).

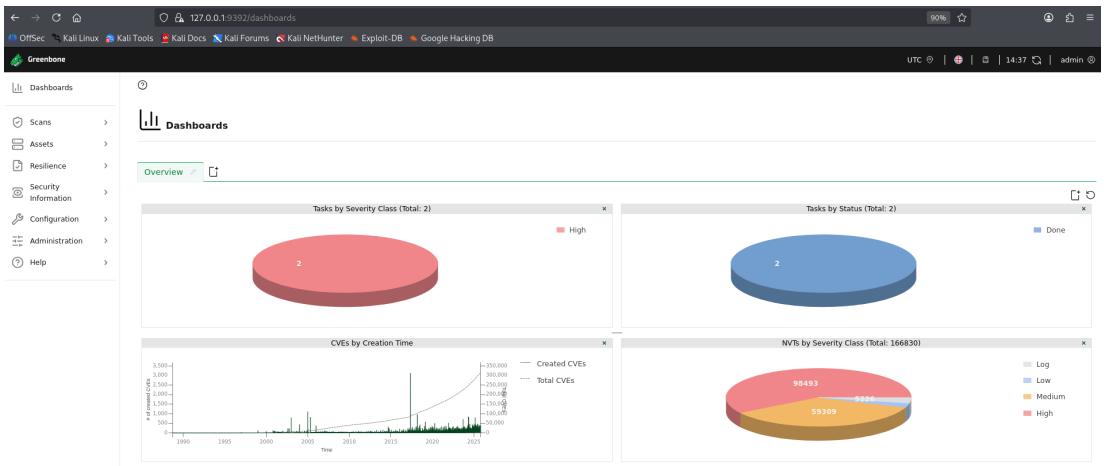


Figura 5.8: Interfaz web de GVM.

5.3.2 Configuración de OpenVAS

Como queremos automatizar el escaneo de vulnerabilidades, debemos crear un *Schedule* para configurar el inicio de las *Tasks* a una hora y período determinados. Como se puede ver en la figura 5.9, para la realización del proyecto se ha establecido que los escaneos de vulnerabilidades se realicen diariamente a las 10 de la mañana hora peninsular.

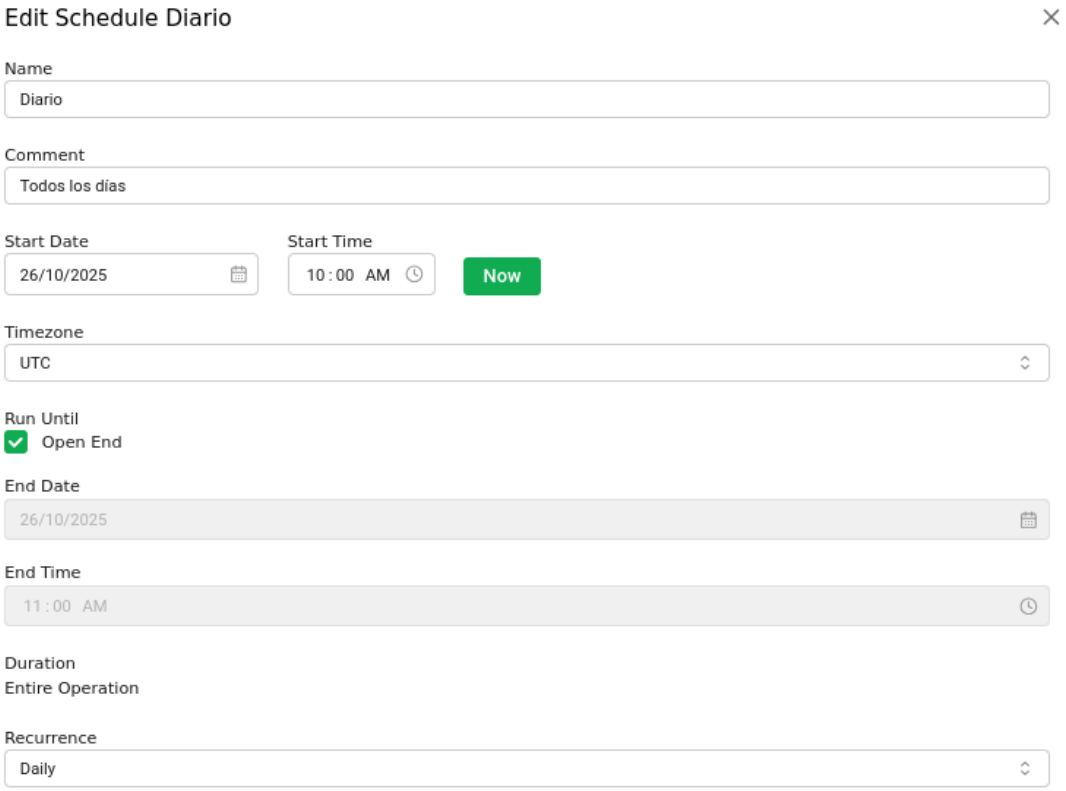


Figura 5.9: Configuración del *Schedule* de los escaneos.

Una vez realizada la instalación y el despliegue de GVM, debemos realizar las configuraciones necesarias para poder automatizar el escaneo de vulnerabilidades en las máquinas objetivo. Para ello, definiremos un *Target* y una *Task* para cada máquina objetivo que tengamos en nuestra red.

Para la configuración del *Target* debemos especificar el nombre y la dirección IP de la máquina objetivo. En la Figura 5.10 podemos ver la configuración de *Target* para la primera máquina objetivo.

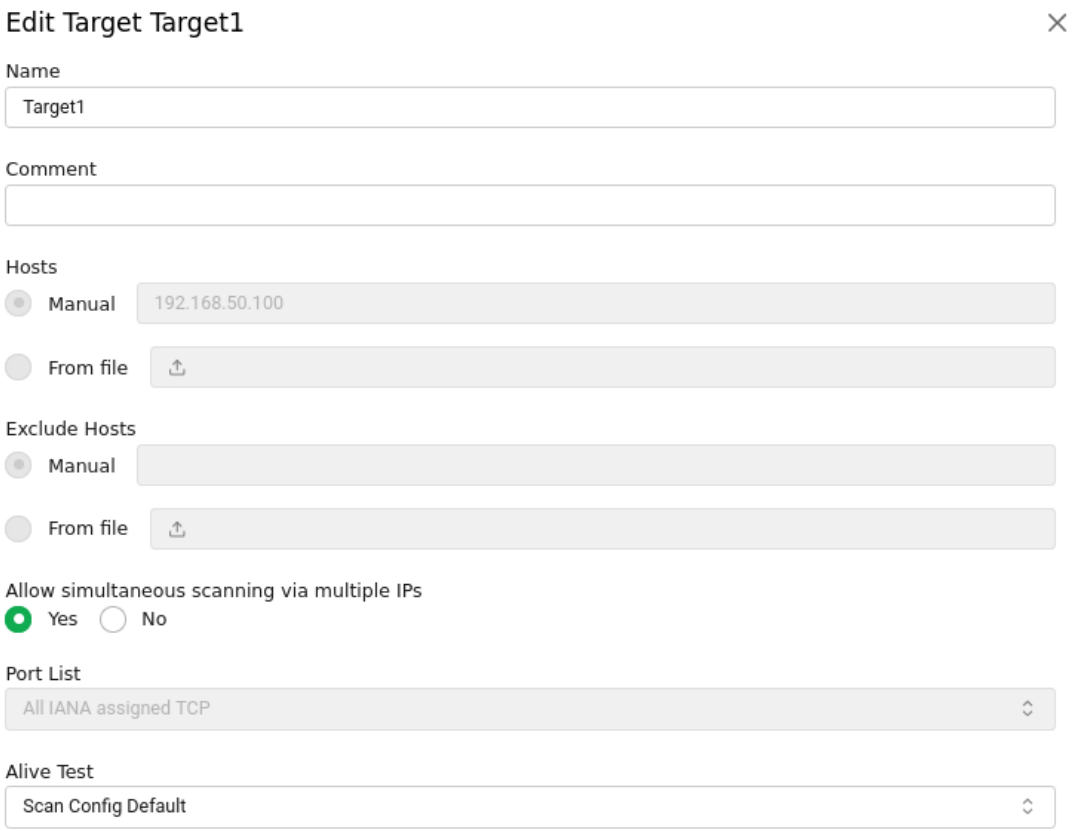


Figura 5.10: Configuración del *Target* para la primera máquina objetivo.

Una vez configurado el *Target* para una máquina objetivo, debemos configurar su *Task*, para la cual debemos especificar el nombre, el *Target* escaneado, el tipo de escaneo (siempre usaremos el tipo "*Full and fast*", ya que es el estándar para detectar vulnerabilidades conocidas [55]) y el horario (*Schedule*) en el que queremos que automáticamente se ejecute el escaneo. En la Figura 5.11 podemos ver la configuración realizada de la *Task* del *Target* de la primera máquina objetivo.

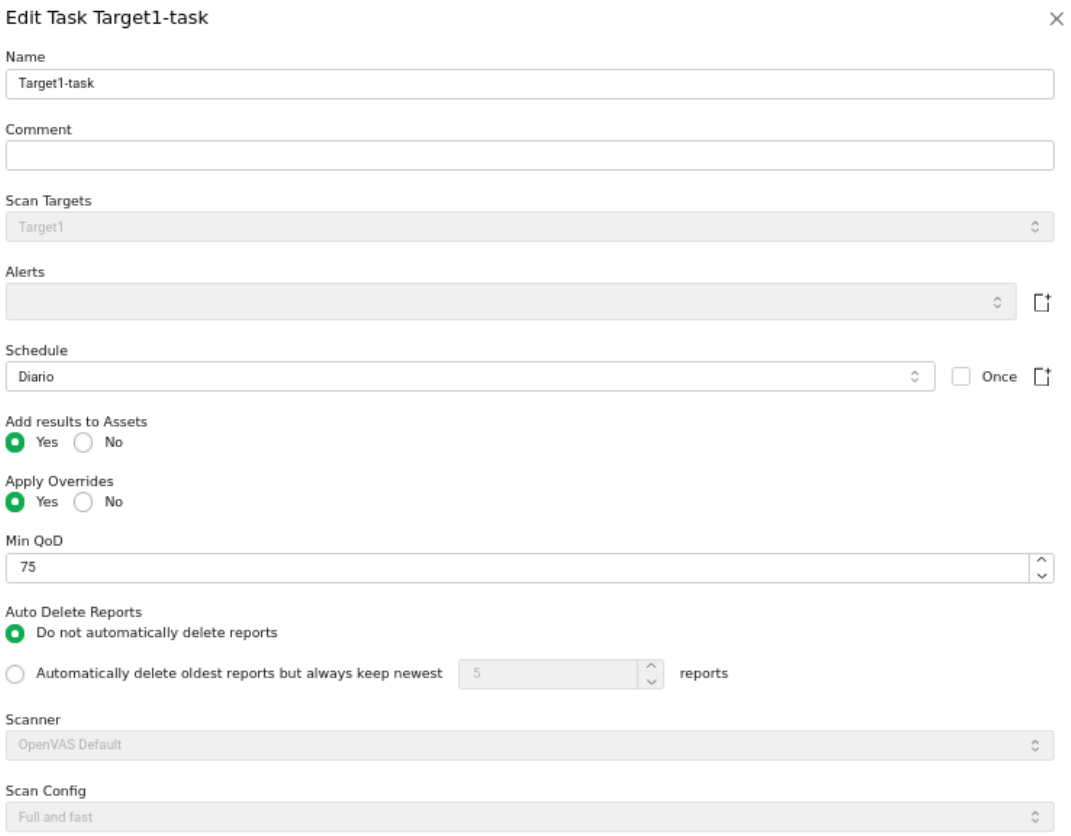


Figura 5.11: Configuración del *Task* para la primera máquina objetivo.

5.3.3 Extracción automatizada del reporte de GVM

Una vez automatizado el escaneo de vulnerabilidades sobre las máquinas objetivo, debemos implementar un script que de forma automática extraiga el último informe de vulnerabilidades de GVM a local para su posterior procesado.

Primero crearemos el script (`pipeline.sh`) que almacenará todo el pipeline que queremos automatizar y le daremos permisos de ejecución. Este será el programa principal que orquestará la total funcionalidad del proyecto.

```
1  touch pipeline.sh
2  chmod +x pipeline.sh
3
```

Como buscamos automatizar la funcionalidad completamente, usaremos cron. Para ello, realizaremos el comando `crontab -e` y añadiremos la siguiente línea al final del archivo:

```
0 13 * * * /usr/bin/flock -n /tmp/pipeline.lock
/home/kali/TFG/pipeline.sh >> /home/kali/TFG/pipeline.log 2>&1

```

Añadiendo esta línea y guardando, `pipeline.sh` será ejecutado todos los días a las 13:00 (es decir, 3 horas después de comenzar el escaneo de vulnerabilidades que configuramos anteriormente en el *Schedule* de OpenVAS). Además, se ha implementado bloqueo con la herramienta *flock*, que garantiza que solo se ejecute una instancia del script a la vez. Por último, cabe destacar que la salida proporcionada por el script es almacenada en un log (`pipeline.log`) a modo de registro para un usuario de administración.

Alerts

Para ello GVM cuenta con un sistema de configuración de alertas (*Alerts*) que permite enviar a la máquina orquestadora de diferentes maneras (Email, HTTP Get, *SCP*, *Send to host*, etc.) el informe una vez haya sido finalizado el escaneo. No obstante, después de varios intentos, no se ha logrado implementar esta configuración, por lo que se ha descartado.

gvm-cli

Como opción alternativa, se ha usado gvm-cli [47], una herramienta de línea de comandos para poder interactuar de forma remota con GVM mediante el protocolo GMP (*Greenbone Management Protocol*).

El siguiente código detalla de forma esquemática la lógica de la extracción de los informes haciendo uso de los IDs atómicos que usan tanto las *Tasks* (`task_id`) como los *Reports* (`report_id`). [2]:

```
# Obtenemos el ID de todas las tasks que hemos configurado
sobre las máquinas objetivo:
TASK_IDS=$(gvm-cli --gmp-username USUARIO_GVM --gmp-password
CONTRASEÑA_GVM --xml "<get_tasks/>" 2>/dev/null | grep -oP
'(?<=task id=")[^"]+')

# Si TASK_ID corresponde a solo un ID de una Task en concreto,
podemos extraer el ID su último reporte de la siguiente manera:
TEMP_ID_FILE=$(mktemp)
gvm-cli --gmp-username USUARIO_GVM --gmp-password
CONTRASEÑA_GVM socket --xml \"<get_tasks task_id='$TASK_ID'
```

[2] Es necesario sustituir los valores "*USUARIO_GVM*" y "*CONTRASEÑA_GVM*" para usar dichos comandos. Las credenciales usadas para este proyecto han sido omitidas por seguridad.

```
    details='1'/>\" | grep -oP '(?<=<report id=\")[^\"]+' | tail -1
    > $TEMP_ID_FILE
7   LAST_REPORT_ID=$(cat "$TEMP_ID_FILE")
8   rm "$TEMP_ID_FILE"
9
10  # Una vez tenemos almacenado en LAST_REPORT_ID el ID del último
    reporte de esa task, podemos extraer el informe en formato XML:
11  gvm-cli --gmp-username USUARIO_GVM --gmp-password
    CONTRASEÑA_GVM socket --xml \"<get_reports
    report_id='$LAST_REPORT_ID' format_id='$FORMAT_ID' details='1'
    filter='apply_overrides=0 levels=hmlg rows=100 min_qod=70
    first=1 sort-reverse=severity'/>\" > \"$XML_FILE\"
12
```

Estas líneas de código permiten la extracción de datos mediante consultas XML. Su funcionamiento se basa en identificar las *Tasks* creadas para cada máquina objetivo y recuperar el identificador del reporte más reciente de cada una de ellas (mediante el uso del parámetro `tail -1`). Cabe destacar la implementación de calidad de detección usando la opción QoD (*Quality of Detection*), de manera que solo sean incluidas en el informe las vulnerabilidades confirmadas con una certeza mayor al 70%.

Usando la lógica de estos comandos, obtendremos en local el reporte de vulnerabilidades realizado por OpenVAS de cada máquina objetivo en formato XML de forma automatizada.

5.4 Módulo de normalización e integración

El reporte generado por OpenVAS es extraído en formato XML. Este tipo de formato presenta una estructura jerárquica ineficiente para su procesamiento por un LLM debido a su complejidad y alta verbosidad. El procesamiento en bruto de este reporte consume un número muy elevado de *tokens*, lo que aumenta el coste de procesamiento del modelo, incluso pudiendo llegar a no procesarlo por excederse de tamaño.

Es por ello que se ha diseñado un módulo intermedio situado entre el módulo de escaneo y el módulo de modelos de lenguaje que realice una normalización e integración de cada uno de los reportes extraídos. Este módulo será orquestado por el archivo principal anteriormente creado (`pipeline.sh`), convirtiendo los reportes a un formato mucho más ligero y optimizado, el JSON. Posteriormente, el reporte será subdividido en varios objetos JSON, donde cada uno de ellos almacenará la información de una variable para su posterior procesamiento independiente.

5.4.1 Parseo del informe (`parseador.py`)

Para realizar el parseo del informe, se ha desarrollado e implementado un archivo Python denominado `parseador.py`. Este archivo tiene como función la deserialización de la estructura jerárquica del XML y su conversión a un formato JSON simplificado en unos pocos campos (como *target*, *name*, *port*, *severity* o *solution*, entre otros).

A continuación, se presenta la lógica principal del fichero `parseador.py`. El código completo del parseador (`parseador.py`) está disponible en el repositorio de Github asociado a este Trabajo de Fin de Grado [56]:

```
        # Campo name
        vuln["name"] = (res.findtext("name") or "").strip()
        # Campo host
        vuln["host"] = (res.findtext("host") or "").strip()
        # Campo port
        vuln["port"] = (res.findtext("port") or "").strip()
        # Campo threat
        vuln["threat"] = (res.findtext("threat") or "").strip()
        # Campo severity
        vuln["severity"] = (res.findtext("severity") or "").strip()

        nvt = res.find("nvt")
        if nvt is not None:
            # Campo family
            vuln["family"] = (nvt.findtext("family") or "").strip()

            # Campo CVSS
            vuln["cvss"] = (nvt.findtext("cvss_base") or "").strip()

            cves = []
            for ref in nvt.findall(".//ref[@type='cve']"):
                cves.append(ref.get("id"))
            if cves:
                # Campo CVE
                vuln["cve"] = cves

            tags = nvt.findtext("tags")
            if tags:
                tags_dict = {}
                for part in tags.split("|"):
                    if "=" in part:
                        k, v = part.split("=", 1)
                        tags_dict[k.strip()] = v.strip()
                vuln.update(tags_dict)
```

```
35
36              # Campo solution
37              vuln["solution"] = (nvt.findtext("solution") or
    "").strip()
38
39          # Función de limpieza que elimina los campos vacíos.
40          vuln = {k: v for k, v in vuln.items() if v}
41          results.append(vuln)
42
```

El código implementado busca elementos específicos dentro del reporte mediante la utilidad `.findtext` de la librería `xml.etree.ElementTree`, y para los campos de family, CVSS y CVE se accede al nodo NVT (*Network Vulnerability Test*), recorriendo su estructura. Por último, se realiza un filtro de limpieza para eliminar posibles campos vacíos presentes en el reporte en crudo y así evitar enviárselos al LLM.

5.4.2 Fragmentación del reporte (`spliteador.py`)

Como siguiente paso, es necesario realizar una división del reporte ya parseado en diferentes vulnerabilidades. En entornos como Metasploitable 2, el número total de vulnerabilidades que pueden aparecer en un informe generado por OpenVAS puede ser de más de una centena. Un informe de este tamaño es casi seguro que supere la ventana de contexto que un LLM pudiese soportar, provocando errores fatales en la generación.

Por este motivo y para maximizar la eficiencia (implementando un sistema de concurrencia en la posterior generación de scripts para cada una de las vulnerabilidades), se ha implementado un programa denominado `spliteador.py` que divide el reporte en cada una de las vulnerabilidades presentes en él para poder ser procesadas independientemente por el agente.

A continuación, se muestran las partes más significativas del código presente en este archivo. El código completo del fragmentador (`spliteador.py`) está disponible en el repositorio de Github asociado a este Trabajo de Fin de Grado [57].

```
1       results = data.get("results", [])
2
3       for vuln in results:
4         if not vuln or not vuln.get("name"):
5           continue
6
7         name = vuln.get("name", "unknown")
8         safe_name = re.sub(r'[^A-Za-z0-9_]+', '_',
    name)[:40].strip('_')
```

```
            vuln_id = uuid.uuid4().hex[:8]

            filename = f"{count:03d}_{safe_name}_{vuln_id}.json"
            filepath = os.path.join(out_dir, filename)

            with open(filepath, "w", encoding="utf-8") as f_out:
                json.dump({
                    "target": target,
                    "result": vuln
                }, f_out, indent=2, ensure_ascii=False)

```

Cabe destacar el uso de la librería `uuid` para la generación de identificadores únicos para cada una de las vulnerabilidades, debido a un problema detectado en el desarrollo donde una misma vulnerabilidad puede estar presente en diferentes puertos, produciendo colisiones de nombres al almacenar todos los subarchivos en un mismo directorio.

La salida de este proceso, como podemos ver en la Figura 5.12 (la cual ya incluye los scripts generados) genera un directorio de vulnerabilidades en formato JSON, preparado para ser procesado por el LLM.

```
┌──(venv_openai)─(kali㉿kali)-[~/TFG/reports_example]
└─$ tree
.
└── 2025-12-06
    └── 5e6840f0-b60a-4d5d-8c55-86d527fff229
        ├── report.json
        ├── report.xml
        ├── scripts
        │   └── 2025-12-06
        │       ├── 204547_samba_3_0_0_3_0_25rc3_ms-rpc_remote_shell_command_execution_vulnerability_-_active_check.sh
        │       ├── 204550_operating_system_os_end_of_life_eol_detection.sh
        │       ├── 204550_weak_encryption_algorithm_s_supported_ssh.sh
        │       ├── 204556_weak_key_exchange_kex_algorithm_s_supported_ssh.sh
        │       └── 204557_weak_host_key_algorithm_s_ssh.sh
        └── split
            ├── 000_Operating_System_OS_End_of_Life_EOL_Dete_70f51007.json
            ├── 001_Samba_3_0_0_3_0_25rc3_MS_RPC_Remote_Shel_6e79ea34.json
            ├── 002_Weak_Host_Key_Algorithm_s_SSH_3583175f.json
            ├── 003_Weak_Key_Exchange_KEX_Algorithm_s_Suppor_13cbe96a.json
            ├── 004_Weak_Encryption_Algorithm_s_Supported_SS_1ec27c6a.json
            └── 005_Weak_MAC_Algorithm_s_Supported_SSH_6bd87935.json
```

Figura 5.12: Ejemplo de organización del directorio de trabajo para un reporte.

5.5 Módulo de despliegue y conectividad

Una vez generados todos los scripts de mitigación necesarios, es necesario implementar un canal de comunicación seguro y automatizado entre la máquina orquestadora y las máquinas objetivo. Para ello, se optó por desarrollar un mecanismo de autenticación basada en claves SSH RSA y un usuario de administración denominado `centinela` para la ejecución de scripts remotos.

Más detalladamente, se debe generar un par de claves RSA de 4096 bits en la máquina orquestadora, y la clave pública resultante se deposita en el archivo `authorized_keys` del usuario de administración de dicha máquina objetivo, permitiendo que la máquina orquestadora se autentique criptográficamente sin necesidad de intervención humana. Además, este método es superior en seguridad al mecanismo de autenticación por contraseña, ya que no es vulnerable a los ataques de fuerza bruta tradicionales y permite la revocación selectiva de credenciales.

Por otro lado, a pesar de que emplear certificados digitales como mecanismo de autenticación sería una solución más robusta y escalable para la gestión de identidades en un entorno empresarial, su implementación tuvo que ser descartada en este proyecto por motivos de incompatibilidad. Las máquinas objetivo implementadas usan la imagen de Metasploitable 2, cuya versión heredada de OpenSSH (en concreto, la 4.7p1) carece de soporte para los protocolos de autenticación mediante certificados, los cuales no fueron introducidos en el estándar hasta la versión 5.4, publicada en marzo de 2010 [58]. Para facilitar la implementación del mecanismo de autenticación, se ha determinado que el uso de claves RSA es la mejor solución disponible.

5.5.1 Creación del usuario de administración

Para garantizar la ejecución remota de los scripts de mitigación generados, es imperativo configurar dicho usuario de autenticación (`centinela`) en cada máquina objetivo de la red para lograr una ejecución automatizada. Dicho usuario debe tener privilegios para usar `sudo` sin contraseña para evitar las solicitudes interactivas de contraseña durante la ejecución.

Para la correcta configuración de dicho usuario, es imperativo modificar dos ficheros de configuración del sistema:

- `/etc/ssh/sshd_config`: Archivo de configuración principal de OpenSSH. Deberemos ajustar la seguridad del servicio, configurando, entre otros, el parámetro *PubkeyAuthentication* para permitir el uso de un mecanismo de autenticación por clave pública.
- `/etc/sudoers`: Archivo del sistema que define las políticas de delegación de privilegios y permisos. Modificándolo lograremos otorgar al usuario de administración la capacidad de poder ejecutar comandos de administrador sin necesidad de intervención humana para introducir credenciales, lo cual es imperativo para la automatización del proyecto.

Para ello, se deben ejecutar los siguientes comandos en cada máquina objetivo para configurar el usuario de administración de la forma descrita:

```
1
2     # Creamos el usuario 'centinela' y los añadimos al grupo sudo y
      admin para permitir la elevación de privilegios
3     useradd -m -s /bin/bash centinela
4     echo 'centinela:centinela' | chpasswd
5     usermod -aG sudo centinela || true
6     usermod -aG admin centinela || true
7
8     # Preparamos el directorio para las claves SSH que
      configuraremos en la siguiente sección
9     mkdir -p /home/centinela/.ssh
10    chmod 700 /home/centinela/.ssh
11    chown centinela:centinela /home/centinela/.ssh
12
13    # Debemos asegurarnos que las siguientes opciones están
      habilitadas en el fichero de configuracuón /etc/ssh/sshd_config:
14    # PasswordAuthentication yes
15    # PermitRootLogin no
16    # PubkeyAuthentication yes
17    cat /etc/ssh/sshd_config
18
19    /etc/init.d/ssh restart
20
21    # Para desactivar el sudo del usuario de administración
      centinela, deberemos añadir las siguientes líneas al fichero de
      configuración /etc/sudoers:
22    # Defaults:centinela !authenticate
23    # centinela ALL=(ALL) NOPASSWD: ALL
24    nano /etc/sudoers
25
```

Posteriormente, se procede a la generación del par de claves RSA en la máquina orquestadora y a copiar dichas claves al fichero `authorized_keys` del usuario de administración de cada máquina objetivo. A continuación, se detallan los comandos necesarios para realizar lo descrito en la primera máquina objetivo (IP = `192.168.50.100`):

```
1     # Generación del par de claves en la máquina orquestadora
2     ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa_auto -N ''
3
4     # Comando para la transmisión de la clave pública a la máquina
      objetivo
5     ssh-copy-id -o HostKeyAlgorithms=+ssh-rsa -i
      ~/.ssh/id_rsa_auto.pub centinela@192.168.50.100
6
```

```
7   # Forzamos los algoritmos RSA antiguos para garantizar la
    compatibilidad de Kali Linux (2026) con el servidor SSH de
    Metasploitable 2 (2008):
8   ssh -o HostKeyAlgorithms=+ssh-rsa -o
    PubKeyAcceptedAlgorithms=+ssh-rsa -i ~/.ssh/id_rsa_auto
    centinela@192.168.50.100
9
10  # Limpiamos el comando introduciendo la siguiente configuración
    en el fichero de configuración ~/.ssh/config de la máquina
    orquestadora:
11  # Host 192.168.50.100
12  # User centinela
13  # IdentityFile ~/.ssh/id_rsa_auto
14  # IdentitiesOnly yes
15  # HostKeyAlgorithms +ssh-rsa
16  # PubKeyAcceptedAlgorithms +ssh-rsa
17  nano ~/.ssh/config
18
19  # Probamos la conexión ssh de nuevo:
20  ssh centinela@192.168.50.100
21
```

Durante el desarrollo de este incremento, el establecimiento de la relación de confianza vía SSH presentó un gran desafío debido a que el cliente OpenSSH de la máquina orquestadora (perteneciente a la imagen de Kali Linux, 2026) implementa por defecto políticas que inhabilitan algoritmos considerados inseguros por este. Debido a que la máquina objetivo usa un servidor SSH del 2008 que depende únicamente de algoritmos basados en SHA-1 (considerados inseguros por el orquestador), el conflicto de versiones impide una conexión automática sencilla, devolviendo constantemente errores en la negociación del algoritmo.

Para resolver esta incompatibilidad, se debe establecer manualmente la aceptación de algoritmos heredados mediante los parámetros *HostKeyAlgorithms* y *PubKeyAcceptedAlgorithms*. Una vez establecida la relación de confianza, podemos ”limpiar” el comando SSH trasladando los parámetros al fichero de configuración `/.ssh/config`.

Como podemos ver en la Figura 5.13, si probamos la conexión via SSH con la máquina objetivo 192.168.50.100, podremos ver que nos permite establecerla sin introducir contraseña.

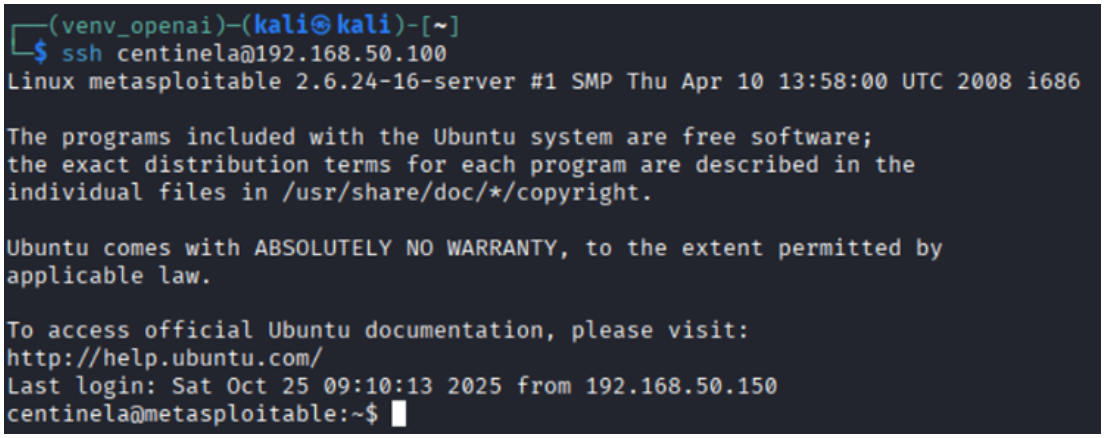


Figura 5.13: Conexión SSH *passwordless* con la máquina objetivo desde la máquina orquestadora.

5.5.2 Implementación del flujo de despliegue y ejecución

El paso final consiste en la implementación en el script de orquestación (`pipeline.sh`) del código necesario para poder tanto transferir como ejecutar en remoto cada uno de los scripts de mitigación en las máquinas objetivo correspondientes. Como se muestra en la Figura 5.14, el flujo resultaría en el siguiente:

1. El agente inicia la conexión y se autentica mediante las claves RSA configuradas.
2. Se transfieren todos los scripts de mitigación mediante SCP al directorio temporal `/tmp` de la máquina objetivo correspondiente.
3. Se ejecuta cada uno de los scripts vía SSH, donde el usuario de administración eleva privilegios automáticamente para aplicar los parches necesarios.

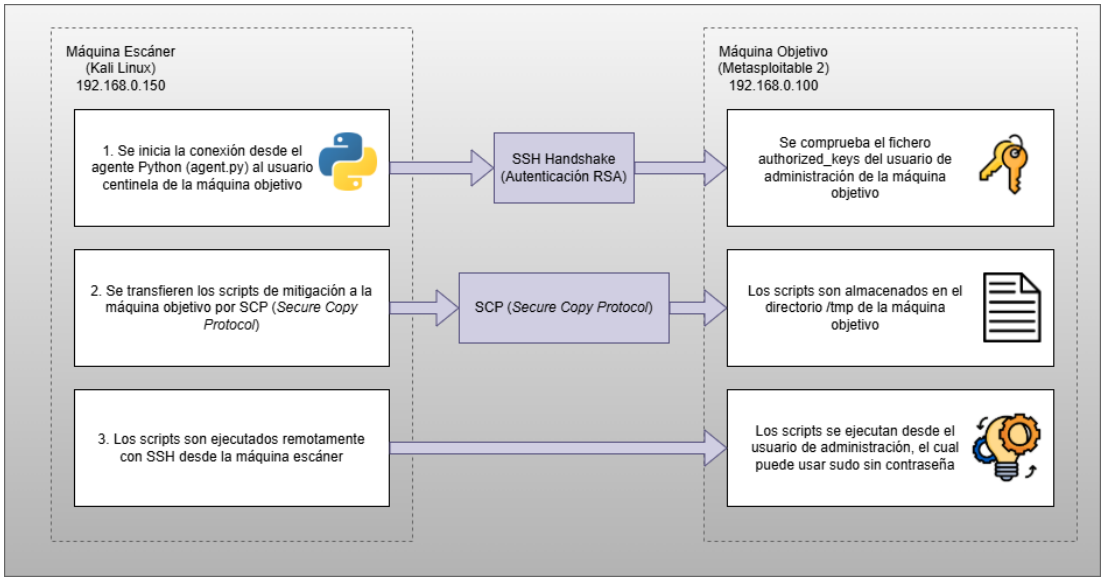


Figura 5.14: Flujo de autenticación, transferencia y ejecución remota entre la máquina orquestadora y las máquinas objetivo.

A continuación se muestra el fragmento de código del orquestador (`pipeline.sh`) encargado de esta funcionalidad. Con esto, el código completo descrito del orquestador está disponible en el repositorio de Github asociado a este Trabajo de Fin de Grado [59].

```
    # Se consulta la IP de la máquina objetivo a través de la
tarea (consultamos mediante gvm-cli).
    TARGET_ID=$(gvm-cli --gmp-username USUARIO_GVM --gmp-password
CONTRASENA_GVM socket --xml "<get_tasks task_id=\"$TASK_ID\"
details='1'/>" | grep -oP '(?<=<target id=\")[^\"]+')

    SSH_HOST=$(gvm-cli --gmp-username USUARIO_GVM --gmp-password
CONTRASENA_GVM socket --xml "<get_targets
target_id=\"$TARGET_ID\"/>" | grep -oP '(?<=<hosts>)[^<]+')

    # Recolección de todos los scripts para su posterior envío
    declare -a FULL_FILES=()
    declare -a BASE_FILES=()
    while IFS= read -r -d '' f; do
        FULL_FILES+=("$f")
        BASE_FILES+=("$(basename "$f")")
    done < <(find "$SCRIPTS_DIR" -type f -name '*.sh' -print0)

    # Se envían todos los scripts al directorio temporal  \tmp de
la máquina objetivo
    scp -P "$SSH_PORT" -i "$SSH_KEY_PATH" -o
StrictHostKeyChecking=no "${FULL_FILES[@]}"
```

```
    "$SSH_USER"@"$SSH_HOST":"$SSH_DEST_DIR"/
16
17      # Ejecución secuencial de cada script + Registro
18      for base in "${BASE_FILES[@]}"; do
19          remote_path="$SSH_DEST_DIR/$base"
20          local_log="$LOGS_DIR/${base%.sh}.log"
21
22          ssh -n -p "$SSH_PORT" -i "$SSH_KEY_PATH" -o
    StrictHostKeyChecking=no \
23              "$SSH_USER"@"$SSH_HOST" \
24              "chmod +x '$remote_path' && sudo /bin/bash
    '$remote_path'" \
25              > "$local_log" 2>&1
26
27          # Eliminamos el script en la máquina objetivo tras su
    ejecución.
28          ssh -n -p "$SSH_PORT" -i "$SSH_KEY_PATH" -o
    StrictHostKeyChecking=no "$SSH_USER"@"$SSH_HOST" "rm -f
    '$remote_path'"
29      done
30
```

Con este último módulo implementado, queda cerrado el ciclo de automatización del flujo de trabajo planteado. El módulo descrito permite que el sistema aplique correcciones masivas sin intervención humana, reduciendo drásticamente el tiempo de respuesta en comparación con una corrección manual humana.

5.6 Entorno de comando profesional

Para mejorar la experiencia del usuario durante la ejecución del pipeline y darle a la solución propuesta un acabado más profesional, se ha implementado una interfaz de línea de comandos con un estilo más moderno. Esto facilita la comprensión de los logs emitidos por pantalla, evitando la fatiga visual del administrador.

Para implementar este entorno mejorado, se han combinado librerías y herramientas especializadas según el lenguaje de programación.

5.6.1 Uso de Gum en el orquestador (`pipeline.sh`)

El orquestador del proyecto (`pipeline.sh`) usa la herramienta Gum para estructurar el flujo de ejecución [60]. Como este archivo es un script Bash, se ha seleccionado esta herramienta por su capacidad de introducir elementos gráficos que normalmente requerirían programación compleja.

Principalmente se han implementado tres componentes:

- **Cabeceras**: Se ha hecho uso del comando "*gum style*" para separar las fases del proyecto (en fase de generación y fase de despliegue) de forma visualmente atractiva.
- **Spinners**: Se ha usado el comando "*gum spin*" para indicar que una tarea está en curso mediante un *spinner* de carga, evitando dar la idea al administrador de que la pantalla está congelada.
- **Tablas de resultados:** Uso de "*gum table*" para la representación final de los datos una vez finalizada la ejecución.

En el código a continuación, se muestra un ejemplo del uso de estos comandos:

```
# Ejemplo de comando de Cabeceras
gum style \
          --border normal \
          --margin "1" \
          --padding "0 1" \
          --border-foreground 33 \
          --foreground 33 \
          "DESPLIEGUE Y EJECUCIÓN REMOTA"

# Ejemplo de comando de Spinners
if gum spin --spinner dot --title "Descargando reporte..." -- \
  bash -c "gvm-cli ... > reporte.xml"; then
  gum style --foreground 82 "□ Reporte descargado"
fi

# Ejemplo de comando de Tabla de resultados
echo "MÉTRICA,VALOR
  - Fecha,$DATE_TAG
  - Scripts Generados,$NUM_GENERADOS
  - Ejecutados OK,$OK_COUNT" | \
  gum table --border double --border-foreground 33
```

Para finalizar, con el fin de ilustrar la evolución de la interfaz del proyecto, a continuación se muestra una comparativa entre la salida por pantalla de la primera iteración del sistema y su versión actual. La Figura 5.15a corresponde a la salida de la primera versión del orquestador, la cual presentaba una salida de texto plano estándar, funcional pero con muy baja legibilidad. Por el contrario, en la Figura 5.15b podemos ver la salida para su última versión, permitiendo al administrador de seguridad identificar de un solo vistazo cada fase de la ejecución del flujo y los resultados críticos del proceso.

```
daniel@daniel-VirtualBox:~$ ./run-gvm-pipeline.sh
[2025-09-26 04:07:38] Pipeline START
[2025-09-26 04:07:38] 1/3 Scan & export...
[+] Report format id: a994b278-1f62-11e1-96ac-406186ea4fc5
[*] Lanzando Target1-task (185d691b-d6af-4d00-bb6b-73d2fe6a0cec)…
  -> report_id = 0b4f078c-38de-499a-a323-b6b2de880d6e
  -> Estado: Doneerando)o)
  -> Exportando a /opt/pipeline/out/report-Target1-task.xml
[*] Lanzando Target2-task (dd8d6bdf-bf44-4fb0-9dac-796595e0e005)…
  -> report_id = bca241a3-59dc-4f03-8bbc-70b45b605600
  -> Estado: Doneerando)o)
  -> Exportando a /opt/pipeline/out/report-Target2-task.xml
[*] Lanzando Target3-task (174e684f-d6d8-430c-80f2-8e065ae48324)…
  -> report_id = c731e448-c24d-4b61-9be7-5c01dbbc7e37
  -> Estado: Doneerando)o)
  -> Exportando a /opt/pipeline/out/report-Target3-task.xml
[✓] Exportación completada. Archivos en /opt/pipeline/out
[2025-09-26 04:13:31] 2/3 Parse reports to JSON...
[✓] report-Target1-task.json  ←  report-Target1-task.xml  (findings=0)
[✓] report-Target2-task.json  ←  report-Target2-task.xml  (findings=0)
[✓] report-Target3-task.json  ←  report-Target3-task.xml  (findings=0)
[2025-09-26 04:13:31] 3/3 Agent dispatch...
[2025-09-26 04:13:31] -> agent /opt/pipeline/json/report-Target1-task.json
```

(a) Entorno de comando de la versión inicial del orquestador.

```
PROCESADO DIARIO DE VULNERABILIDADES

📅 FECHA DE EJECUCIÓN: 2026-01-19
📂 DIRECTORIO BASE:    /home/kali/TFG/reports/2026-01-19

Procesando la Task ID: 26112521-3873-4c18-b26c-e3b4e6260767

✔ Obtenido último report ID con éxito: 5fdc918e-bc7a-4fcc-a6ca-5cd12b293766

✔ Reporte guardado con éxito: /home/kali/TFG/reports/2026-01-19/26112521-3873-4c18-b26c-e3b4e6260767/report.xml

✔ JSON generado con éxito: /home/kali/TFG/reports/2026-01-19/26112521-3873-4c18-b26c-e3b4e6260767/report.json

✔ JSON dividido en vulnerabilidades con éxito: /home/kali/TFG/reports/2026-01-19/26112521-3873-4c18-b26c-e3b4e6260767/split

GENERACIÓN DE SCRIPTS DE SEGURIDAD

  📂 Directorio de vulnerabilidades: /home/kali/TFG/reports/2026-01-19/26112521-3873-4c18-b26c-e3b4e6260767/split
  📂 Directorio destino: /home/kali/TFG/reports/2026-01-19/26112521-3873-4c18-b26c-e3b4e6260767/scripts

✔ Entorno virtual activado.
```

(b) Entorno de comando de la última versión del orquestador.

Figura 5.15: Evolución del entorno de comando del orquestador.

5.6.2 Uso de Rich en el agente (`agent.py`)

Como el módulo de modelos de lenguaje está escrito en Python, se ha optado por el uso de la librería Rich para la profesionalización de su entorno, permitiendo renderizar barras de progreso y tablas dinámicas fácilmente, entre otros.

Con Rich, podremos visualizar el progreso de los hilos usados para la concurrencia de la generación de los scripts mediante:

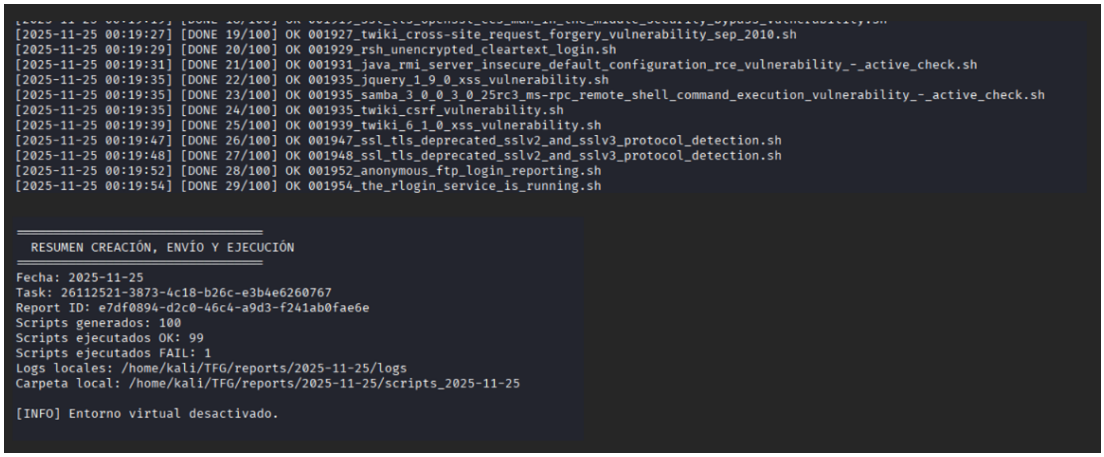
- **Barras de progreso**: Uso del objeto "*Progress*" para mostrar el porcentaje de generación de un script en concreto.
- **Tablas de resumen**: Generación de una tabla informativa al final de la ejecución del agente.

Podemos ver la configuración de la tabla informativa usada por el agente en el siguiente bloque de código:

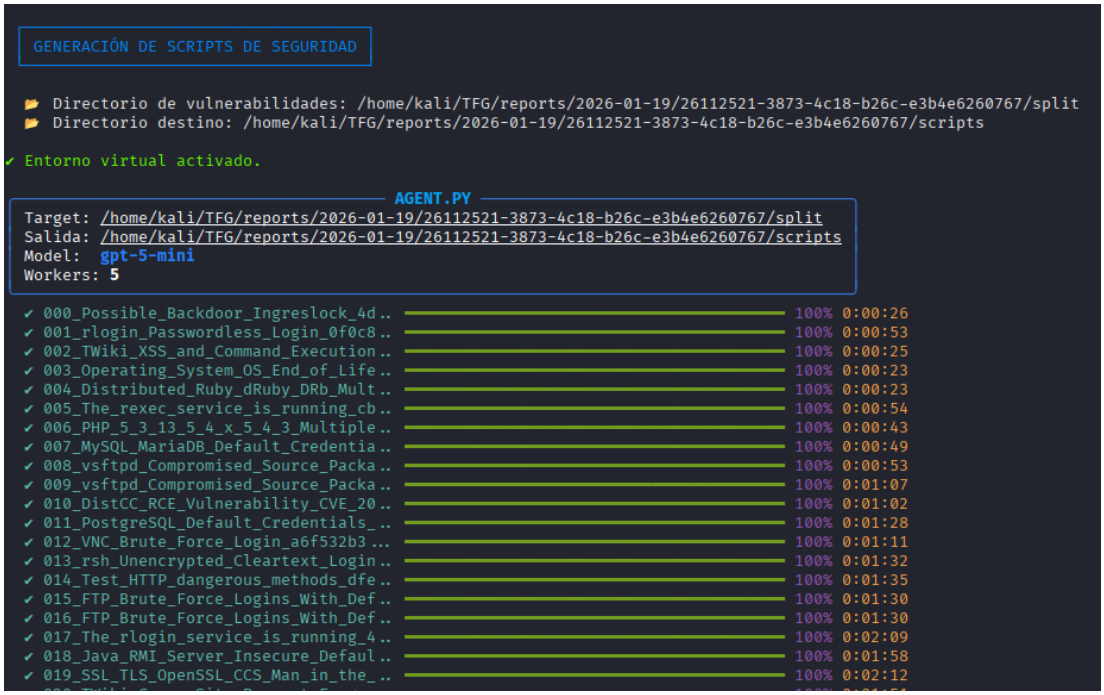
```
    from rich.table import Table
    from rich.console import Console

    # Configuración de la tabla
    table = Table(title="Resumen de Ejecución", box=box.ROUNDED)
    table.add_column("Archivo JSON", style="cyan")
    table.add_column("Estado", justify="center")
    table.add_column("Tiempo", justify="right")

    # Añadido de filas dinámico según resultados
    for res in results:
        status = "[bold green]OK[/bold green]" if res['ok'] else
"[bold red]FAIL[/bold red]"
        table.add_row(res['file'], status,
f"{res['duration']:.1f}s")

    console.print(table)

```

Listing 5.1: Implementación de tabla de resultados con Rich en agent.py

A modo de ilustración de la evolución de la interfaz del agente, como en la sección anterior, se muestra en las Figuras 5.16a y 5.16b una comparativa entre la salida por pantalla antes y después del uso de la librería Rich.



(a) Entorno de comando del agente antes del uso de la librería Rich.



(b) Entorno de comando del agente con la librería Rich implementada.

Figura 5.16: Evolución del entorno de comando del agente.

Capítulo 6

Resultados

El propósito de este capítulo es probar y analizar la eficacia del sistema de mitigación automatizada desarrollado. Se presentan los resultados obtenidos tras la ejecución del pipeline en la máquina objetivo implementada, y se evalúa el rendimiento de la solución basándose en métricas cuantitativas como la tasa de éxito de mitigación y la reducción de la superficie de ataque.

6.1 Metodología de la prueba de validación

Para poder garantizar tanto la reproducibilidad de las pruebas de validación realizadas como la integridad del entorno de trabajo, se ha establecido el siguiente protocolo de validación:

1. **Uso de *snapshots***: Antes de realizar la primera ejecución del pipeline, se generó una instantánea del estado inicial de la máquina objetivo, permitiendo revertir su sistema a su estado vulnerable inicial para la realización de múltiples pruebas iterativas.

2. **Escaneo inicial**: Se ejecuta manualmente un primer escaneo de vulnerabilidades desde la interfaz web de OpenVAS para conocer el estado de seguridad de partida de la máquina objetivo y generar el reporte XML con el que se trabajará sin esperar a la ejecución con `cron`.

3. **Ejecución del pipeline**: Se ejecuta flujo de trabajo implementado (`pipeline.sh`).

4. **Escaneo de validación**: Se realiza un segundo escaneo de la misma manera para cuantificar el número de vulnerabilidades mitigadas por el sistema de mitigación implementado.

6.2 Prueba de validación

6.2.1 Escaneo inicial

La ejecución de la tarea de escaneo inicial (*Target1-task*) sobre la máquina objetivo reveló un escenario de alto riesgo (véase la Figura 6.1) con vulnerabilidades de severidad máxima (*High* 10.0). Este primer escaneo confirma la idoneidad de Metasploitable 2 como sistema vulnerable para la validación de las mitigaciones. En la Tabla 6.1 podemos ver el número de vulnerabilidades encontrado según su severidad.

Severidad	Número de vulnerabilidades
Alta (*High*)	20
Media (*Medium*)	40
Baja (*Low*)	6
Total	**66**

Tabla 6.1: Vulnerabilidades encontradas en la máquina objetivo según su severidad en el primer escaneo.

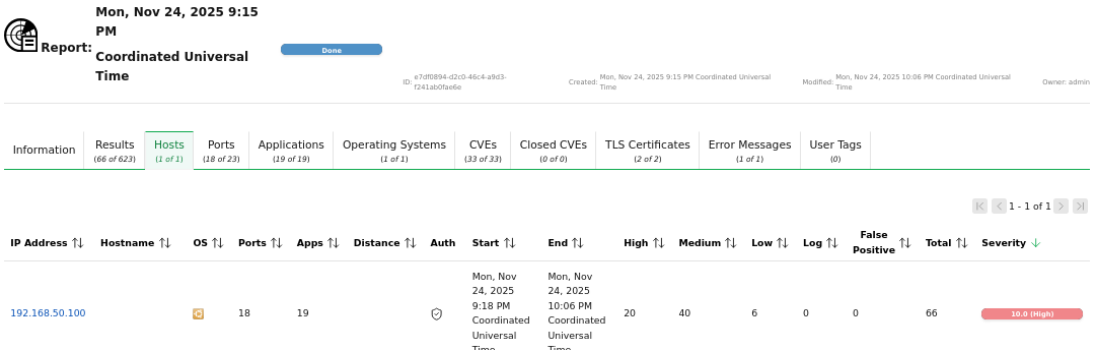


Figura 6.1: Estado de seguridad inicial de la máquina objetivo.

6.2.2 Ejecución del pipeline

Una vez realizado el escaneo, se ejecuta el sistema de mitigación automatizado desarrollado para su posterior validación. Al hacerlo, como podemos ver en la Figura 6.2, un entorno de comando profesional informa al administrador de seguridad de las fases preliminares del

procesamiento del informe: La extracción del reporte de la base de datos de GVM y la normalización de dicho reporte para su posterior procesado.

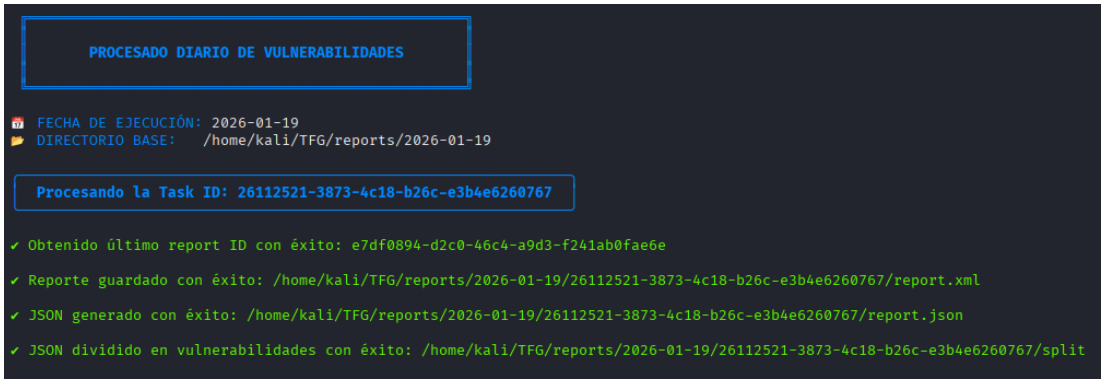


Figura 6.2: Fase preliminar del flujo de trabajo.

Tras ello, se procede a la fase de generación de scripts de mitigación. En la Figura 6.3, podemos ver información relevante sobre dicha fase, donde se nos informa del modelo utilizado, el número de scripts que se pueden generar concurrentemente (*Workers*), la ubicación de los directorios y, sobre todo, del porcentaje de progreso de generación de cada uno de los scripts.

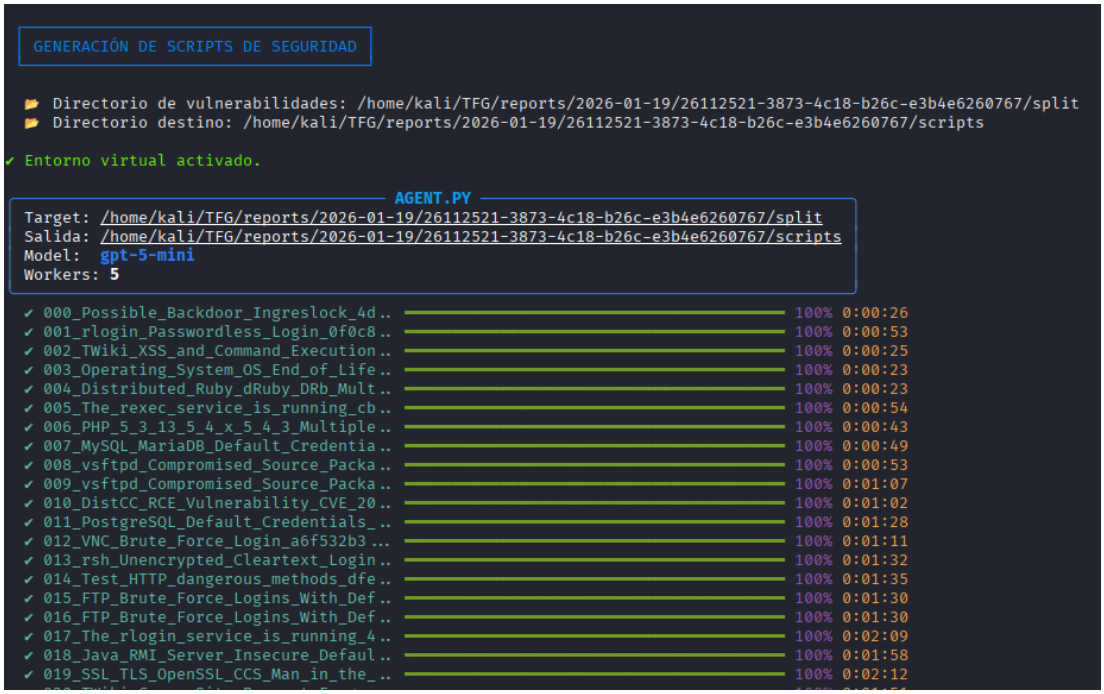


Figura 6.3: Captura del comienzo de la fase de generación de scripts.

Para poder mostrar el funcionamiento interno del agente, se ha extraído una muestra.

Como podemos ver en la Figura 6.4, para la primera vulnerabilidad detectada tenemos la siguiente descripción estructurada en formato JSON correspondiente a la vulnerabilidad crítica detectada (*Ingreslock Backdoor*).

```
┌──(kali㉿kali)-[~/…/reports/2026-01-19/26112521-3873-4c18-b26c-e3b4e6260767/split]
└─$ cat 000_Possible_Backdoor_Ingreslock_4d2c1cea.json
{
  "target": "Target1",
  "result": {
    "name": "Possible Backdoor: Ingreslock",
    "host": "127.0.0.1",
    "port": "1524/tcp",
    "threat": "High",
    "severity": "10.0",
    "family": "Gain a shell remotely",
    "cvss": "10.0",
    "cvss_base_vector": "AV:N/AC:L/Au:N/C:C/I:C/A:C",
    "summary": "A backdoor is installed on the remote host.",
    "impact": "Attackers can exploit this issue to execute arbitrary commands in the\n  context of the application. Successful atta
    "solution": "A whole cleanup of the infected system is recommended.",
    "solution_type": "Workaround"
  }
}
```

Figura 6.4: Descripción de la vulnerabilidad estructurada en formato JSON.

Una vez analizada la descripción, la API de OpenAI genera el script de mitigación para dicha vulnerabilidad (véase Figura 6.5) que como podemos ver, mitiga con éxito la vulnerabilidad verificando la seguridad del entorno y aplicando reglas de iptables.

```
┌──(kali㉿kali)-[~/…/2026-01-19/26112521-3873-4c18-b26c-e3b4e6260767/scripts/2026-01-19]
└─$ cat 182639_possible_backdoor_ingreslock.sh
#!/bin/bash
set -euo pipefail

# Ensure SSH remains allowed through the firewall
iptables -C INPUT -p tcp --dport 22 -j ACCEPT 2>/dev/null || iptables -I INPUT -p tcp --dport 22 -j ACCEPT

# Attempt to identify a process listening on TCP/1524 and stop it if it's not sshd
pid=""
if netstat -tulpn 2>/dev/null | grep -E '[: ]1524\b' >/dev/null 2>&1; then
  pid=$(netstat -tulpn 2>/dev/null | grep -E '[: ]1524\b' | awk '{print $7}' | awk -F'/' '{print $1}' | head -n1)
fi

if [ -n "${pid:-}" ]; then
  pname=$(ps -p "$pid" -o comm= 2>/dev/null | tr -d ' ')
  if [ "$pname" != "sshd" ]; then
    # attempt a graceful stop; ignore failure (process may exit or permissions may vary)
    kill "$pid" 2>/dev/null || true
  fi
fi

# Block only the specific malicious port (1524) in a reversible, idempotent way
iptables -C INPUT -p tcp --dport 1524 -j DROP 2>/dev/null || iptables -I INPUT -p tcp --dport 1524 -j DROP

# Record that a full system cleanup/reinstallation is recommended (manual action required)
echo "$(date +%F_%T) ACTION REQUIRED: Full system cleanup/reinstallation recommended for possible Ingreslock backdoor on 127.0.0.1:1524. Port 1524 block
e -a /tmp/mitigation_todo.log > /dev/null

exit 0
```

Figura 6.5: Script de mitigación generado a partir de la vulnerabilidad.

Una vez finalizada la generación, podemos ver en la Figura 6.6 que la interfaz muestra una tabla con los resultados de esta fase, indicando si la generación de cada script fue exitosa y el tiempo que se tardó en generar. Además, se muestra que el agente Python procesó un total de 100 archivos de vulnerabilidad, generando un script para cada una. El proceso de generación de scripts duró un total de 559.7 segundos (aproximadamente, 9 minutos y medio), lo que valida la alta eficiencia del modelo seleccionado (gpt-5-mini).

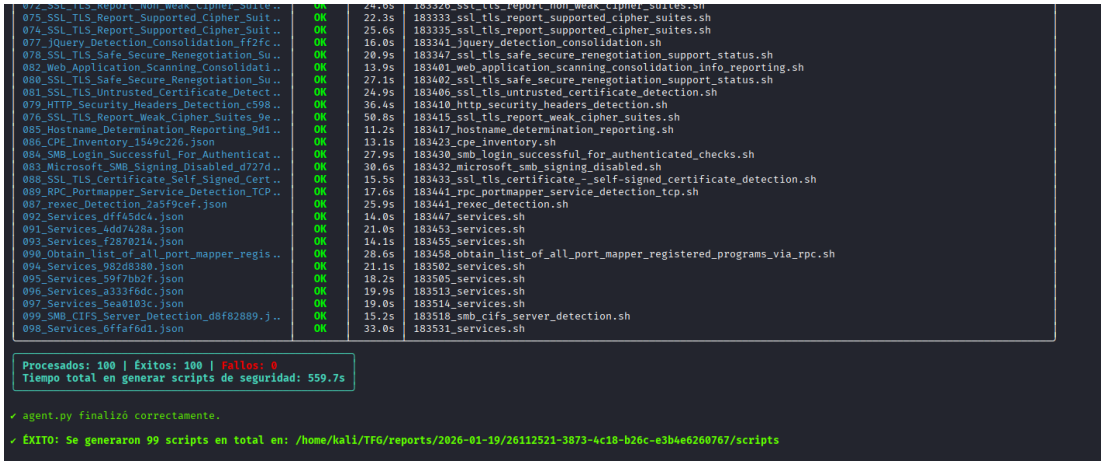


Figura 6.6: Captura del final de la fase de generación de scripts.

De los 100 scripts de seguridad que se solicitaron a la API, los 100 fueron ”creados” correctamente (es decir, la API de OpenAI respondió con *Status 200* el 100% de las veces), pero hubo 1 caso en el que el LLM respondió con una respuesta que no contenía un bloque de código válido, por lo que el pipeline descartó ese script (”*(...) Se generaron 99 scripts en total en: (...)*”). Realizando una segunda prueba de funcionalidad se obtuvo que de los 100 scripts solicitados, 97 contenían código válido y en una tercera, la totalidad de los scripts contenía código con un formato aceptado. Esto nos deja con una media de 98.6% de tasa de éxito de la API al solicitar un script de código.

A continuación, el orquestador transfirió y ejecutó en remoto los scripts en la máquina objetivo. Como podemos ver en la Figura 6.7, la interfaz muestra la fase de transmisión y ejecución en remoto de los scripts de mitigación. El registro muestra una tasa de éxito elevada en la aplicación de los parches, aunque se identificaron fallos puntuales en scripts que usaban comandos no disponibles en el sistema, o que si estaban disponibles, pero fallaron en su ejecución.

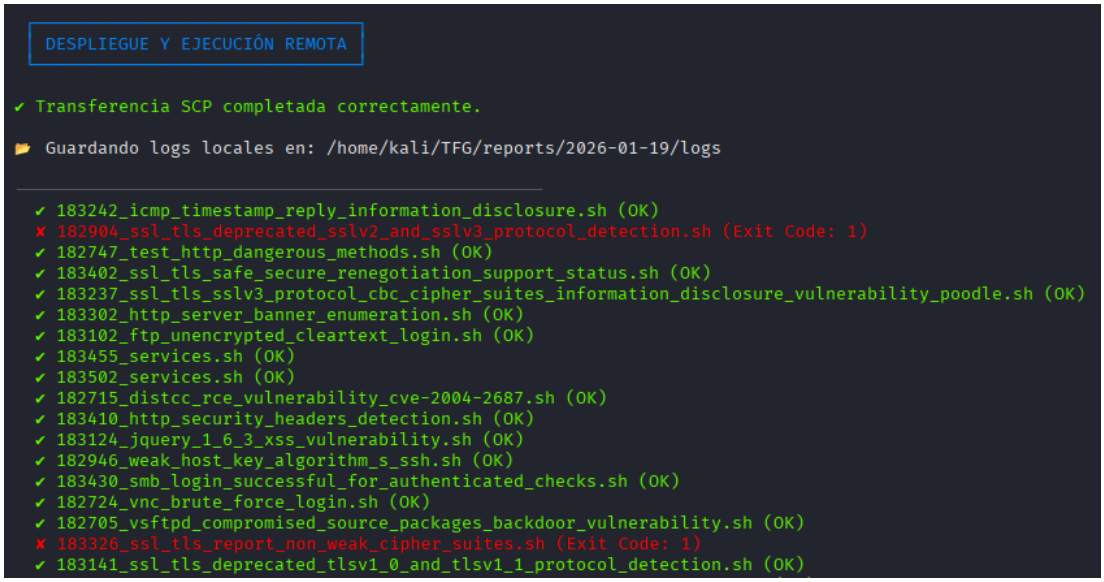


Figura 6.7: Captura del comienzo de la fase de transmisión y ejecución remota de los scripts en la máquina objetivo

Por último, la interfaz muestra una tabla informativa que muestra datos sobre el éxito de la ejecución de los scripts, entre otros (véase Figura 6.8).

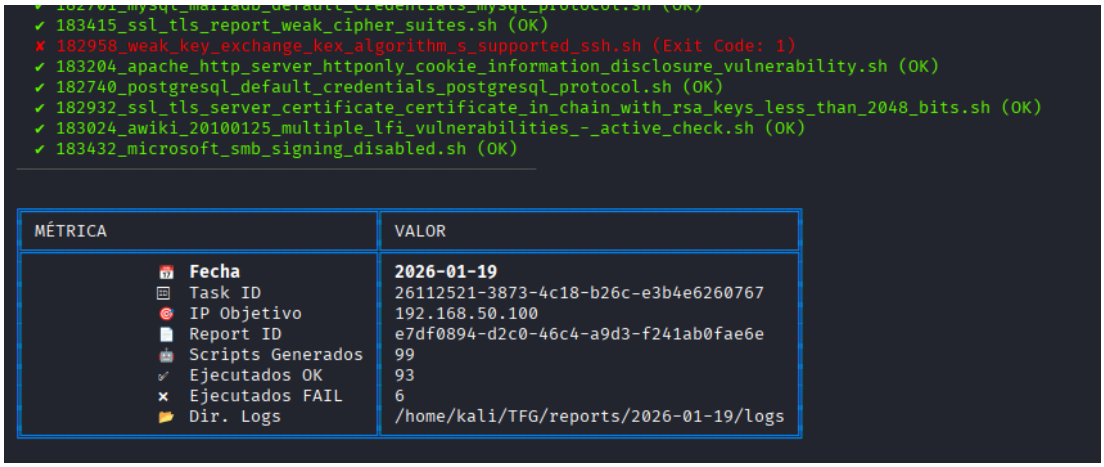


Figura 6.8: Captura del final de la fase de ejecución y final del flujo de trabajo.

Durante esta sección, hemos recorrido la interfaz generada por el orquestador en cada una de las fases del sistema de mitigación, lo que, por lo que podemos ver por pantalla, asegura el funcionamiento del sistema.

6.2.3 Verificación Post-Mitigación

Para validar la efectividad de las acciones aplicadas, se lanzó un segundo escaneo de verificación. Como podemos ver en la Figura 6.9, los resultados muestran una reducción drástica en la superficie de ataque. Podemos ver la comparación de las vulnerabilidades restantes en la máquina en la tabla 6.2.

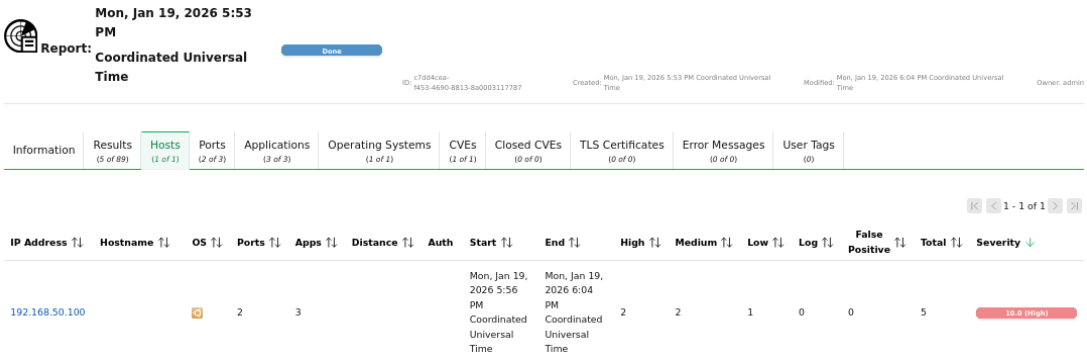


Figura 6.9: Resultado del segundo escaneo tras la ejecución del sistema de mitigación.

Severidad	Vulnerabilidades (1º Escaneo)	Vulnerabilidades (Escaneo final)
Alta (*High*)	20	2
Media (*Medium*)	40	2
Baja (*Low*)	6	1
Total	**66**	**5**

Tabla 6.2: Vulnerabilidades encontradas en la máquina objetivo según su severidad.

La comparación entre el escaneo inicial y el final evidencia que la mayoría de las vulnerabilidades críticas fueron mitigadas eficazmente, transformando un sistema altamente expuesto en uno con un perfil de riesgo drásticamente menor sin intervención humana.

6.3 Evaluación de Rendimiento

A partir de los datos recolectados, se han calculado las siguientes métricas de rendimiento para evaluar la solución:

6.3.1 Tasa de Éxito de Ejecución (TEE)

La Tasa de Éxito de Ejecución (TEE) mide la fiabilidad de los scripts generados por el LLM a la hora de ejecutarse sin errores en la máquina objetivo. Para calcularla, debemos realizar la siguiente operación:

$$TEE = \frac{\text{Scripts Ejecutados}}{\text{Scripts Transferidos}} \times 100 \tag{6.1}$$

Basándonos en los datos recopilados durante la prueba de validación:

- **Scripts Transferidos:** 99
- **Ejecuciones Exitosas (OK):** 93
- **Ejecuciones Fallidas (FAIL):** 6

$$TEE = \frac{93}{99} \times 100 = 93.93\% \tag{6.2}$$

Este resultado nos indica que los scripts en la gran mayoría de los casos podrán ejecutarse sin ningún problema en el sistema de la máquina objetivo.

6.3.2 Tasa de Éxito de Mitigación (TEM)

Como hemos podido ver en la prueba de validación realizada antes, la superficie de ataque se reduce drásticamente. La Tasa de Éxito de Mitigación mide para un nivel de severidad concreto, el porcentaje de vulnerabilidades que se han solucionado una vez se ha ejecutado sobre la máquina objetivo el sistema de mitigación automatizada desarrollado. Para calcular dicha tasa, debemos realizar la siguiente operación:

$$TEM = \left(1 - \frac{\text{Vulnerabilidades totales en el escaneo final}}{\text{Vulnerabilidades totales en el escaneo inicial}}\right) \times 100 \tag{6.3}$$

Basándonos en los datos recopilados durante el escaneo inicial y final:

- **Vulnerabilidades *High***: De las 20 vulnerabilidades que había, solo 2 permanecen aún en el análisis hecho a posteriori. Si calculamos el TEM, obtenemos que un 90% de las vulnerabilidades han sido mitigadas correctamente.
- **Vulnerabilidades *Medium***: De las 40 vulnerabilidades que había, solo 2 permanecen aún en el análisis hecho a posteriori. Si calculamos el TEM, obtenemos que un 95% de las vulnerabilidades han sido mitigadas correctamente.

- **Vulnerabilidades *Low***: De las 6 vulnerabilidades que había, solo 1 permanece aún en el análisis hecho a posteriori. Si calculamos el TEM, obtenemos que un 83.3% de las vulnerabilidades han sido mitigadas correctamente.

Una vez analizados y medidos los datos recopilados, podemos concluir que los scripts mitigan con una muy alta tasa de éxito las vulnerabilidades en el sistema objetivo. Estos resultados validan la viabilidad de delegar la mitigación de vulnerabilidades conocidas mediante el escaneo de una herramienta *open-source* a un LLM de bajo coste. La transformación de un entorno de riesgo con múltiples vulnerabilidades críticas a uno mucho más reducido en cuestión de minutos confirma que la orquestación automatizada mediante LLMs es una solución robusta para reducir drásticamente los fallos de seguridad en redes empresariales.

Por último, es necesario mencionar que la discrepancia entre el número de hallazgos en el reporte resumen (66 vulnerabilidades) y el número de scripts generados (100) se debe a que, como podemos ver en la Figura 6.10, el sistema trata cada combinación única de Vulnerabilidad y Puerto como una tarea de mitigación diferente e independiente. Por ejemplo, si una vulnerabilidad de SSL afecta tanto al puerto 80 como al 8080, el agente generará dos scripts diferentes, asegurando que la mitigación se aplique en todos los puntos de exposición, y no solo en uno.

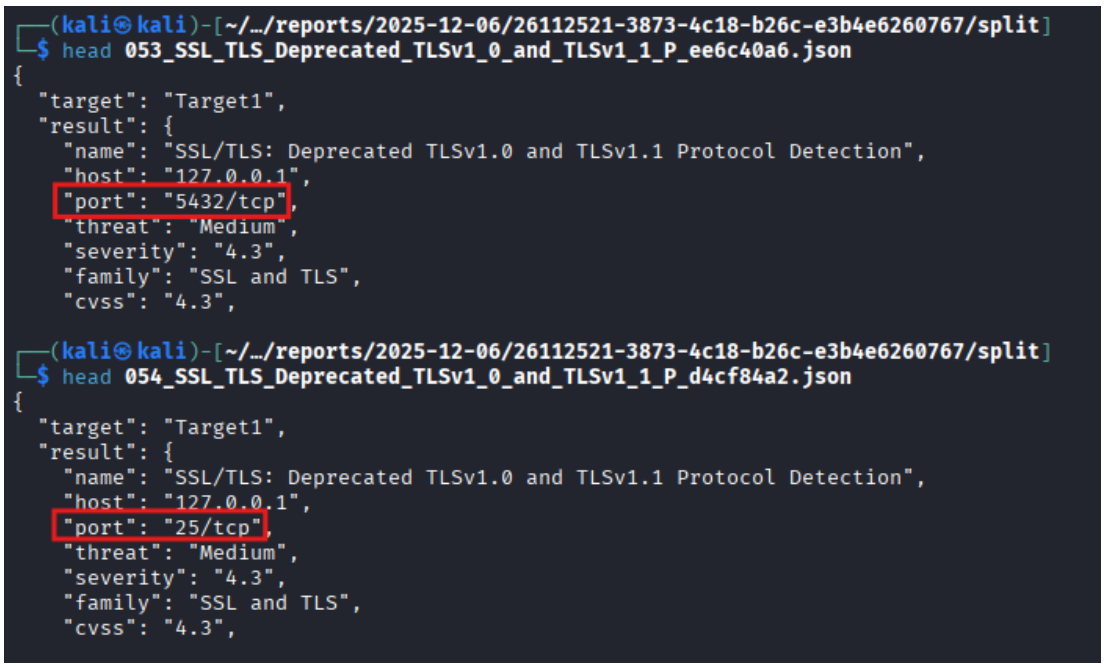


Figura 6.10: Captura donde se aprecia la multiplicidad de vulnerabilidades por puerto.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se exponen las conclusiones finales del proyecto. En él, se detalla una verificación del cumplimiento de los objetivos planteados inicialmente, se analizan las limitaciones y desafíos encontrados y se proponen líneas de evolución futura para el desarrollo del proyecto.

7.1 Conclusiones generales

El desarrollo de este Trabajo de Fin de Grado ha validado la posibilidad de la convergencia de dos tecnologías actualmente en auge: La ciberseguridad defensiva y los LLMs (Modelos de lenguaje). Durante todo este documento, se ha diseñado y evaluado un sistema automatizado capaz de cerrar la brecha existente entre la detección y la mitigación de vulnerabilidades en entornos de redes internas empresariales.

La principal conclusión que podemos desprender responde a un problema planteado al inicio del documento: Este estudio demuestra que la automatización de la mitigación de vulnerabilidades mediante LLMs es posible y viable económicamente. Se ha demostrado que es posible desarrollar un sistema que delegue a un LLM la tarea de interpretación de un reporte de vulnerabilidades técnico y complejo, y de él genere código de mitigación en forma de scripts bash con una tasa de éxito muy elevada.

Como segunda conclusión, hemos demostrado que el uso de LLMs en la mitigación de vulnerabilidades reduce drásticamente el tiempo de respuesta ante incidentes. Una máquina es capaz de procesar scripts de mitigación y ejecutar la aplicación de parches para 100 vulnerabilidades en pocos minutos, a diferencia de un administrador de seguridad que tardaría días.

Como tercera conclusión, se ha validado que es posible el desarrollo de una herramienta con capacidad de respuesta similar a las herramientas comerciales costosas usando solamente

una herramienta de escaneo *open-source* (OpenVAS) y un LLM de bajo coste (la API de OpenAI), creando una tecnología accesible para PYMES y entornos académicos que no se pudiesen permitir una suscripción de alto coste.

Como última conclusión, las pruebas de validación realizadas al sistema confirman que la aplicación de los scripts de mitigación generados reduce drásticamente la criticidad del sistema, mitigando de media el 90% de las vulnerabilidades.

7.2 Revisión del cumplimiento de los objetivos

A continuación, se realiza una revisión del grado de cumplimiento de los objetivos definidos en el **Capítulo 1: Introducción**.

Objetivo 1: Desarrollar un entorno de laboratorio virtualizado

Se desplegó una red privada y aislada con una máquina orquestadora con la imagen de Kali Linux y una serie de máquinas objetivo con la imagen de Metasploitable 2, configurando un canal de comunicación segura mediante ellas usando pares de claves RSA de 4096 bits. El objetivo se considera cumplido.

Objetivo 2: Desarrollar un pipeline que permita extraer el reporte realizado por OpenVAS y normalizarlo

Se ha desarrollado e implementado con éxito un módulo de escaneo de vulnerabilidades usando GVM para el escaneo de vulnerabilidades y creación del informe, y gvm-cli para la extracción del informe en formato XML. Mediante el desarrollo de un parseador (`parseador.py`) y un programa que fragmenta el reporte ya parseado en cada una de las vulnerabilidades para su procesamiento independiente (`spliteador.py`) se ha conseguido normalizar el reporte a un formato JSON procesable por el LLM. El objetivo se considera cumplido.

Objetivo 3: Integrar funciones de análisis semántico mediante la API de OpenAI para generar scripts de mitigación en Bash

Se ha creado un agente en Python (`agent.py`) que integra la API de OpenAI, y mediante el procesado del informe normalizado en formato JSON y uso de un *prompt* muy específico, genera scripts seguros, no interactivos y con una tasa de generación exitosa del 100%. El objetivo se considera cumplido.

Objetivo 4: Automatizar la realización de la tarea periódicamente

Se ha conseguido la automatización de la ejecución del flujo de trabajo de manera periódica mediante el uso de la opción *Schedule* en la interfaz web de OpenVAS (para el escaneo diario de vulnerabilidades) y el uso de *cron* en la máquina orquestadora (para la ejecución diaria del flujo de trabajo). El objetivo se considera cumplido.

Objetivo 5: Realizar una evaluación de la efectividad de la solución planteada mediante pruebas de validación

En el **Capítulo 6: Resultados** se detalló una prueba de validación realizada al flujo, y se presentaron métricas cuantitativas que muestran la validez de la utilidad real de la herramienta. El objetivo se considera cumplido.

7.3 Limitaciones

A pesar de las conclusiones en positivo que podemos extraer una vez finalizado el desarrollo del proyecto, se han identificado ciertas limitaciones relacionadas con el entorno de pruebas y a la simplicidad del enfoque adoptado.

El uso de Metasploitable 2 como imagen para las máquinas objetivo supuso un gran desafío, configurando un prompt muy detallado para el correcto funcionamiento de los scripts de mitigación. Los sistemas operativos modernos tienden a generar otro tipo de comandos, dejando obsoletos los utilizados por esta versión de Ubuntu. Esto provoca que si quisiéramos enfocar el flujo de trabajo a otro sistema operativo, el prompt debería ser modificado.

Por otro lado, es importante tener en cuenta que los entornos de defensa con políticas de seguridad estricta no podrían usar este flujo de trabajo debido a la necesidad de conectarse a la API de OpenAI en la nube para la generación de scripts, lo que se convierte en una limitación en este tipo de escenarios.

Por último, para poder automatizar la mitigación de las vulnerabilidades, las correciones que requerían interactuar con sesiones interactivas no han podido ser mitigados. Esto es un límite puesto que las vulnerabilidades de este tipo no podrán ser solucionadas por este flujo de trabajo.

7.4 Trabajo futuro

Aunque el sistema desarrollado ha cumplido con los objetivos planteados, se han encontrado limitaciones y oportunidades de mejora que evolucionarían la herramienta hacia un nuevo nivel de producción.

- **Soporte para múltiples sistemas operativos**: Mejorar la ingeniería de prompts planteada para que el agente detecte el sistema operativo, y luego adapte la sintaxis de los comandos a este, aumentando drásticamente el alcance del proyecto.
- **Implementación de Certificados SSH**: Usar una imagen más moderna para las máquinas objetivo que permita usar un mecanismo de autenticación de Certificados SSH, mejorando la escalabilidad y seguridad de la red.

- **Integración de LLMs locales**: Eliminar la dependencia de OpenAI por modelos *open-source* ejecutados de manera local permitiría la privacidad total de los datos de las vulnerabilidades de las máquinas objetivo, y permitiría al flujo de trabajo operar en redes con las políticas de seguridad más estrictas.

Apéndices

Apéndice A

Manual de Instalación y despliegue

Este manual proporciona las instrucciones técnicas necesarias para replicar la infraestructura virtualizada del proyecto y desplegar el pipeline de mitigación de vulnerabilidades.

A.1 Requisitos del entorno

Para garantizar la correcta ejecución de los componentes, es recomendable disponer de los recursos enumerados a continuación:

1. **Hipervisor:** VMware Workstation Pro.
2. **VM Orquestadora / Escáner:** Kali Linux (2023.3 o posterior) [61].
3. **VM Objetivo:** Metasploitable 2 [62].
4. **Recursos Kali:** Mínimo 8 GB de RAM y 4 vCPU.
5. **Token de IA:** Clave activa de la API de OpenAI.

A continuación se describen los pasos de instalación y configuración de red [63].

A.2 Configuración de red

La infraestructura en su forma más simplificada se basa en dos máquinas virtuales conectadas a una red interna *host-only* para simular un entorno de red empresarial seguro y aislado de otras redes externas, permitiendo únicamente a la máquina orquestadora el acceso a internet para la descarga de actualizaciones y conexión con la API de OpenAI.

A.2.1 Configuración de las máquinas objetivo

La máquina objetivo debe tener una IP estática definida en su interfaz de red de forma persistente. A continuación se detallan los pasos a seguir para la configuración de la primera máquina objetivo:

1. Inicie sesión con las credenciales por defecto (`msfadmin` / `msfadmin`).
2. Edite el archivo de configuración de interfaces de red mediante el siguiente comando:

```
sudo nano /etc/network/interfaces

```

3. Añada la siguiente configuración:

```
    auto eth1
    iface eth1 inet static
    address 192.168.50.100
    netmask 255.255.255.0

```

4. Reinicie el servicio de red:

```
    sudo /etc/init.d/networking restart

```

A.2.2 Configuración de la máquina orquestadora

El nodo orquestador debe usar una IP estática en el mismo segmento de red para comunicarse con las máquinas objetivo.

1. Configure manualmente la interfaz de red correspondiente con la IP `192.168.50.150`.
2. Verifique la conectividad:

```
    ping -c 4 192.168.50.100

```

A.2.3 Configuración del Hipervisor (VMware)

Para que ambas máquinas se puedan comunicar en un entorno aislado, se debe configurar un adaptador de red específico. La explicación detallada y visual de como configurar estos adaptadores puede ser encontrada en la sección 5.1.1 del **Capítulo 5: Desarrollo e Implementación**.

A.3 Instalación del módulo de escaneo

En la máquina orquestadora, es necesario instalar y configurar el *framework* GVM.

1. Actualice el sistema e instale los paquetes de GVM:

```
sudo apt-get update && sudo apt-get upgrade -y
sudo apt install gvm -y

```

2. Ejecute el asistente de *setup* inicial, que descarga las bases de datos de vulnerabilidades y crea el usuario administrador. Es muy importante que anote la contraseña generada para el administrador al finalizar el proceso.

```
sudo gvm-setup

```

3. Verifique que el despliegue se ha realizado con éxito:

```
sudo gvm-check-setup

```

4. Acceda a la interfaz web en `https://127.0.0.1:9392`.

5. Una vez haya iniciado sesión con su usuario y contraseña, deberá realizar la configuración descrita en la sección 5.3.2 del **Capítulo 5: Desarrollo e Implementación**

A.4 Despliegue del flujo de trabajo

1. Descargue el código del proyecto en el directorio de trabajo donde quiera implementar el proyecto:

```
git clone https://github.com/danielbarbeytotorres/TFG.git

```

2. Exporte las variables en su sesión de terminal para definir los parámetros operativos:

```
export OPENAI_API_KEY="sk-proj-..."
export TARGET_IP="192.168.50.100"
export SSH_KEY_PATH="/home/kali/.ssh/tfg_rsa"

```

A.5 Instalación de Dependencias del Orquestador

Para el correcto funcionamiento del flujo de trabajo, se deben instalar las dependencias necesarias dentro de un entorno virtual Python. Podremos realizar lo descrito siguiendo los siguientes comandos:

```
# Nos situamos en el directorio raíz del proyecto
# Configuración del entorno virtual de Python
cd TFG
python3 -m venv tools/venv_openai
source tools/venv_openai/bin/activate

# Instalación de las dependencias necesarias
pip install openai rich gvm-tools

# Instalación de gum
curl -sL
https://github.com/charmbracelet/gum/releases/download/v0.14.1/gum_0.14.1_amd64.de
-o gum.deb
sudo apt install ./gum.deb && rm gum.deb

```

A.6 Mecanismo de autenticación desatendido

El pipeline requiere capacidad de ejecución remota de comandos sin necesidad de interacción humana, lo que requiere claves SSH y permisos sudo específicos. Debido a la naturaleza crítica y complejidad de los pasos para realizar esta configuración, las instrucciones detalladas se encuentran en la sección 5.5 del **Capítulo 5: Desarrollo e Implementación**.

A.7 Configuración de cron

Para que el flujo de trabajo pueda ser ejecutado periódicamente sin intervención humana, es necesario añadir la tarea al programador del sistema (`cron`):

```
crontab -e

# Añadir la siguiente línea al final del fichero:
0 13 * * * /usr/bin/flock -n /tmp/pipeline.lock
/home/kali/TFG/pipeline.sh >> /home/kali/TFG/pipeline.log 2>&1

```

Con los pasos descritos, cualquier usuario puede desplegar y reproducir el código desarrollado, disponiendo de un sistema automatizado de mitigación de vulnerabilidades completamente funcional.

Bibliografía

[1] Greenbone, "Greenbone community documentation: Architecture," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://greenbone.github.io/docs/latest/architecture.html

[2] INCIBE, "Ciberataque ransomware paraliza actividad del hospital clínic de barcelona," 2023, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.incibe.es/incibe-cert/publicaciones/bitacora-de-seguridad/ciberataque-ransomware-paraliza-actividad-del-hospital

[3] Cibersecurity and Infrastructure Security Agency, "The attack on colonial pipeline: What we've learned & what we've done over the past two years," America's Cyber Defense Agency, Tech. Rep., 2023, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.cisa.gov/news-events/news/attack-colonial-pipeline-what-weve-learned-what-weve-done-over-past-two-years

[4] Instituto Nacional de Ciberseguridad, "Incibe presenta su balance de ciberseguridad 2024 con más de 97.000 incidentes gestionados," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.incibe.es/incibe/sala-de-prensa/incibe-presenta-su-balance-de-ciberseguridad-2024-con-mas-de-97000-incidentes

[5] Agencia de la Unión Europea para la Ciberseguridad, "Enisa threat landscape 2024," Agencia de la Unión Europea para la Ciberseguridad, Tech. Rep., 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024

[6] Europol, "Internet organised crime threat assessment (iocta) 2024," European Union Agency for Law Enforcement Cooperation, Tech. Rep., 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.europol.europa.eu/publication-events/main-reports/internet-organised-crime-threat-assessment-iocta-2024

[7] International Organization for Standardization (ISO/IEC), "Iso 27001 - planificación en iso 27001," 2022, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.normaiso27001.es/planificacion-en-iso-27001/

[8] Qualys, "Vulnerability management detection & response (vmdr) faqs & resources," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.qualys.com/fundamentals/what-is-vulnerability-management-detection-response

[9] Tenable, "Tenable vulnerability management," 2025, documentación oficial. Accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.tenable.com/products/vulnerability-management

[10] Palo Alto Networks, "Cortex xsoar," 2025, documentación oficial. Accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.paloaltonetworks.com/cortex/cortex-xsoar

[11] Greenbone Networks, "Greenbone technical documentation (techdoc portal)," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://docs.greenbone.net/

[12] Murugiah Souppaya (NIST), Karen Scarfone (Scarfone Cybersecurity), "Guide to enterprise patch management planning: Preventive maintenance for technology," NIST, Tech. Rep., 2022, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://csrc.nist.gov/pubs/sp/800/40/r4/final

[13] The MITRE Corporation, "Cve - common vulnerabilities and exposures," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.cve.org/

[14] FIRST.org, "Common vulnerability scoring system v3.1: Specification document," accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.first.org/cvss/v3.1/specification-document

[15] ——, "Common vulnerability scoring system v3.1: User guide," accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.first.org/cvss/v3.1/user-guide

[16] Greenbone, "Greenbone management protocol (gmp)," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://docs.greenbone.net/GSM-Manual/gos-22.04/en/gmp.html

[17] Cole Stryker (IBM), "¿qué son los llm?" 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.ibm.com/es-es/think/topics/large-language-models

[18] OpenAI, "Gpt-5-mini: A faster, cost-efficient version of gpt-5 for well-defined tasks," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://platform.openai.com/docs/models/gpt-5-mini

[19] ——, "Prompt engineering," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://platform.openai.com/docs/guides/prompt-engineering

[20] ——, "Openai api reference: Chat completions," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://platform.openai.com/docs/api-reference/chat

[21] Python Software Foundation, "Python qualys.11 documentation," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://docs.python.org/3.13/

[22] P. S. Foundation, "venv — creation of virtual environments," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://docs.python.org/3/library/venv.html

[23] World Wide Web Consortium (W3C), "Extensible markup language (xml) 1.0 (fifth edition)," 2008, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.w3.org/TR/xml/

[24] Ecma International, "The json data interchange syntax," 2017, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

[25] T. Ylonen and C. Lonvick, "The secure shell (ssh) protocol architecture," 2006, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.rfc-editor.org/rfc/rfc4251

[26] Camilo Gutiérrez Amaya. (2013) Funcionamiento del algoritmo rsa. WeLiveSecurity - ESET. Accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.welivesecurity.com/la-es/2013/01/18/funcionamiento-del-algoritmo-rsa/

[27] IBM, "¿qué es la virtualización?" 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.ibm.com/es-es/think/topics/virtualization

[28] Broadcom, "Snapshots," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://techdocs.broadcom.com/es/es/vmware-cis/desktop-hypervisors/fusion-pro/13-0/using-vmware-fusion/protecting-your-virtual-machines/snapshots.html

[29] Stackscale, "Hipervisores: definición, tipos y soluciones," 2023, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.stackscale.com/es/blog/hipervisores/

[30] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," 1974, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://doi.org/10.1145/361011.361073

[31] Offensive Security, "Kali docs | kali linux documentation," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.kali.org/docs/

[32] Rapid7, "Metasploitable 2 exploitability guide," 2012, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://docs.rapid7.com/metasploit/metasploitable-2-exploitability-guide/

[33] European Union Agency for Cybersecurity (ENISA), "Enisa threat landscape 2023," ENISA, Tech. Rep., 2023, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2023

[34] Microsoft, "Microsoft digital defense report 2023," Microsoft Security, Tech. Rep., 2023, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.microsoft.com/en-us/security/security-insider/microsoft-digital-defense-report-2023

[35] Tenable, "Tenable vulnerability management," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.tenable.com/products/tenable-io

[36] Palo Alto Networks, "Cortex xsoar: Security orchestration, automation, and response," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.paloaltonetworks.com/cortex/cortex-xsoar

[37] Microsoft, "Microsoft security copilot," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.microsoft.com/en-us/security/business/ai-machine-learning/microsoft-security-copilot

[38] PMBC, "¿qué es desarrollo iterativo e incremental?" 2021, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://pmbc.es/que-es-desarrollo-iterativo-e-incremental/

[39] Universidade da Coruña, "Grado en ingeniería informática. planificación de la enseñanza y plan de estudios," https://estudos.udc.es/es/study/detail/614g01v01, 2025, accedido 22 de enero de 2026.

[40] Asana, "Diagrama de gantt: qué es y cómo crear uno con ejemplos," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://asana.com/es/resources/gantt-chart-basics

[41] Hostinger, "Bash script: qué es, cómo escribir uno y ejemplos," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.hostinger.com/es/tutoriales/bash-script-linux

[42] Broadcom, "Vmware by broadcom: Fusion and workstation," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion

[43] Microsoft, "Documentation for visual studio code," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://code.visualstudio.com/docs

[44] OpenAI, "Overview | openai api platform," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://platform.openai.com/docs/overview

[45] Github, "Github," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://github.com/

[46] OpenAI, "The official python library for the openai api," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://github.com/openai/openai-python

[47] Greenbone, "gvm-tools: Remote Control of Your Greenbone Vulnerability Manager (GVM)," 2024, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://greenbone.github.io/gvm-tools/

[48] Will McGugan, "Rich 14.1.0 documentation," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://rich.readthedocs.io/en/latest/introduction.html

[49] Glassdor, "Sueldos para el puesto de ingeniero de software en españa," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.glassdoor.es/Sueldos/ingeniero-de-software-sueldo-SRCH_KO0,21.htm

[50] Santander, "Calculadora de coste de un trabajador para la empresa en 2025," 2025, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.impulsa-empresa.es/coste-trabajador-empresa/#COSTE

[51] Glassdor, "Sueldos para el puesto de product owner en españa," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.glassdoor.es/Sueldos/product-owner-sueldo-SRCH_KO0,13.htm

[52] OpenAI, "Openai api: Pricing," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://platform.openai.com/docs/pricing

[53] ——, "OpenAI API: GPT-4o mini Model," 2023, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://platform.openai.com/docs/models/gpt-4o-mini

[54] Daniel Barbeyto Torres, "agent.py," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://github.com/danielbarbeytotorres/TFG/blob/main/tools/agent.py

[55] Greenbone, "Managing the performance," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://docs.greenbone.net/GSM-Manual/gos-22.04/en/performance.html

[56] Daniel Barbeyto Torres, "parseador.py," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://github.com/danielbarbeytotorres/TFG/blob/main/tools/parseador.py

[57] ——, "spliteador.py," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://github.com/danielbarbeytotorres/TFG/blob/main/tools/spliteador.py

[58] , "Openssh 5.4 release," 2010, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.openssh.org/txt/release-5.4

[59] Daniel Barbeyto Torres, "pipeline.sh," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://github.com/danielbarbeytotorres/TFG/blob/main/pipeline.sh

[60] Charmbracelet, "Charm," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://charm.land/

[61] OffSec, "Get kali | kali linux," 2026, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.kali.org/get-kali/#kali-platforms

[62] SOURCEFORGE, "Metasploitable download | sourceforge," 2019, accedido: 22 de enero de 2026. [En línea]. Disponible en: https://sourceforge.net/projects/metasploitable/

[63] MyDFIR. (2024, May) How to setup: Metasploitable 2 & openvas (tutorial). Tutorial de instalación y configuración inicial. Accedido: 22 de enero de 2026. [En línea]. Disponible en: https://www.youtube.com/watch?v=O8FQO17yEKw