Documento de Proyecto: GHA (Gemini Hack Assist)

Versión: 1.0 **Autor:** Diego Thomas Valdés Villar, Gemini **Fecha:** 7 de Octubre de 2025

1.0 Resumen Ejecutivo

El proyecto **Gemini Hack Assist (GHA)** introduce un agente de software avanzado diseñado para optimizar los flujos de trabajo en operaciones de ciberseguridad ofensiva (hacking ético). GHA funciona como un orquestador inteligente que traduce las intenciones de un operador, expresadas en lenguaje natural, en comandos ejecutables para un conjunto de herramientas de pentesting.

La solución ejecuta estas operaciones en un entorno de ejecución seguro y aislado (sandbox) y, posteriormente, utiliza inteligencia artificial para analizar y resumir los resultados. El objetivo principal de GHA es reducir la barrera de entrada a herramientas complejas, minimizar errores de sintaxis y acelerar el ciclo de evaluación de seguridad, permitiendo que los operadores se concentren en el análisis estratégico en lugar de en la mecánica de la ejecución.

2.0 Alcance y Objetivos

2.1 Objetivo Principal

- Desarrollar un orquestador en Python que sirva de interfaz entre un usuario, la API de Google Gemini y un entorno de hacking contenerizado para ejecutar tareas de pentesting mediante comandos en lenguaje natural.

2.2 Objetivos Secundarios

- **Seguridad:** Garantizar que todas las herramientas de hacking se ejecuten en un entorno aislado (contenedor Docker) para prevenir cualquier impacto en el sistema anfitrión.
- **Usabilidad:** Implementar un ciclo de "comando y retroalimentación" donde la IA no solo traduce comandos, sino que también analiza los resultados para ofrecer resúmenes claros.
- **Control del Operador:** Integrar un mecanismo de confirmación humana ("Human-in-the-Loop") como paso de seguridad indispensable antes de la ejecución de cualquier comando.
- **Modularidad:** Diseñar el código de manera estructurada y funcional para facilitar futuras expansiones y el mantenimiento.

3.0 Arquitectura de la Solución

La arquitectura de GHA se fundamenta en tres componentes especializados que operan de forma sinérgica:

- **3.1 Componente de Inteligencia (El Cerebro): API de Google Gemini** Responsable de las capacidades cognitivas del sistema. Su función es doble: primero, interpretar la intención del usuario y traducirla a la sintaxis precisa de un comando de herramienta; segundo, procesar la salida de datos crudos de dichas herramientas y generar un análisis ejecutivo.
- **3.2 Entorno de Ejecución Aislado (Las Manos): Contenedor Docker/Kali Linux** Constituye el "sandbox" operacional. Un contenedor Docker, basado en una imagen de Kali Linux, aloja todo el arsenal de herramientas de pentesting. Este enfoque de contenerización asegura un aislamiento completo, protegiendo el sistema anfitrión y garantizando un entorno de pruebas limpio y reproducible.
- **3.3 Orquestador Central (El Sistema Nervioso): Script `gha.py`** Es el núcleo lógico del sistema. Este script Python gestiona el flujo completo de la operación: interactúa con el usuario a través de una CLI, construye y envía los prompts a la API de Gemini, gestiona el ciclo de vida de la ejecución de comandos vía el SDK de Docker, y presenta los resultados finales.

4.0 Flujo Operacional Detallado

El proceso operativo de GHA sigue un ciclo de vida estructurado en 8 fases:

1. **Recepción de Instrucción:** El operador introduce una tarea en lenguaje natural.
2. **Formulación de Prompt de Comando:** El orquestador contextualiza la instrucción y solicita a Gemini su traducción a un comando ejecutable.
3. **Llamada a API (Traducción):** Se consume el endpoint de Gemini para obtener la traducción.
4. **Extracción y Validación:** El orquestador parsea la respuesta para extraer el comando de las etiquetas `<CMD>`.
5. **Confirmación Humana (Checkpoint de Seguridad):** El comando propuesto se presenta al operador, quien debe autorizar explícitamente su ejecución.
6. **Ejecución Contenerizada:** Tras la autorización, el orquestador invoca a Docker para ejecutar el comando en el contenedor Kali y captura la salida (`stdout`).
7. **Llamada a API (Análisis):** La salida cruda se envía a Gemini para su análisis y resumen.
8. **Presentación de Resultados:** El análisis procesado se entrega al operador, finalizando el ciclo.

5.0 Especificaciones Técnicas

- **Lenguaje de Desarrollo:** Python 3.11+
- **Entorno de Ejecución:** Contenedor Docker
- **Dependencias Python Clave:**
  - `google-generativeai`: para la interacción con la API de Gemini.
  - `docker`: para la orquestación de contenedores.
  - `python-dotenv`: para la gestión segura de claves de API.
- **Imagen Base del Contenedor:** `kalilinux/kali-rolling`
- **Herramientas de Pentesting Integradas:** Nmap, Dirb, theHarvester, Metasploit Framework, SQLmap, John the Ripper, Hashcat, Hydra.

6.0 Metodología de Desarrollo y Lecciones Aprendidas

El desarrollo se abordó con una metodología iterativa, permitiendo la identificación y resolución de desafíos técnicos clave que informaron el diseño final.

- **7.1 Diseño Modular y Prototipado** El proyecto se inició con una definición clara de la arquitectura y la construcción de un prototipo funcional para cada componente (Dockerfile, script de orquestación), asegurando la viabilidad del concepto desde las primeras etapas.
- **7.2 Desafíos Técnicos Resueltos**
    1. **Gestión de Versiones de la API de IA:** Se encontró un error `404 Not Found` persistente al interactuar con la API de Gemini. Se determinó que era causado por una discrepancia entre la versión de la librería y los nombres de los modelos disponibles. **Solución:** Se implementó un script de diagnóstico (`check_models.py`) para realizar una introspección de la API y listar los modelos accesibles, permitiendo una configuración dinámica y precisa.
    2. **Aislamiento y Gestión de Dependencias:** Durante las pruebas, surgieron errores de tipo `ImportError`. La causa raíz fue una incorrecta gestión del entorno virtual de Python. **Solución:** Se reforzó el protocolo de desarrollo para requerir la activación explícita del entorno (`venv`) antes de la instalación de dependencias (`pip install -r requirements.txt`), garantizando la reproducibilidad del entorno.
    3. **Orquestación y Nomenclatura de Contenedores:** El orquestador fallaba al no encontrar el contenedor Docker. El análisis (`docker ps -a`) reveló que los contenedores se estaban creando sin un nombre estático. **Solución:** Se estandarizó el uso del flag `--name` en el comando `docker run` para asignar un identificador predecible (`kali-mcp-env`), permitiendo que el script de Python lo localizara de manera fiable.

7.0 Protocolos de Seguridad y Consideraciones Éticas

El diseño de GHA se rige por principios de seguridad y ética.

- **Principio de Aislamiento (Sandboxing):** La ejecución contenerizada es un requisito no negociable del diseño, previniendo cualquier tipo de desbordamiento de las operaciones hacia el sistema anfitrión.
- **Principio de Supervisión Humana (Human-in-the-Loop):** La funcionalidad de confirmación manual es una salvaguarda deliberada que asegura que el operador humano mantiene la autoridad final sobre cada acción ejecutada.
- **Uso Designado:** GHA es una herramienta concebida para fines educativos y para su uso en auditorías de seguridad autorizadas. Su aplicación en entornos no autorizados constituye una violación de los principios éticos del hacking y de la legislación vigente.