Informe de Auditoría WebGoat 8.1.0

Alumno: David Navarro

Curso: Introducción a la Ciberseguridad – KeepCoding

1. Ámbito y alcance

Ámbito

El presente informe documenta una **práctica de auditoría de seguridad con fines exclusivamente formativos**, realizada sobre **WebGoat 8.1.0**, un laboratorio deliberadamente vulnerable desarrollado por OWASP para el aprendizaje práctico del **OWASP Top 10**.

El objetivo de la práctica es identificar, validar y explicar vulnerabilidades representativas de aplicaciones web, así como analizar su impacto y proponer medidas de mitigación adecuadas, siempre dentro de un entorno controlado.

Entorno evaluado

Las pruebas se han realizado bajo las siguientes condiciones:

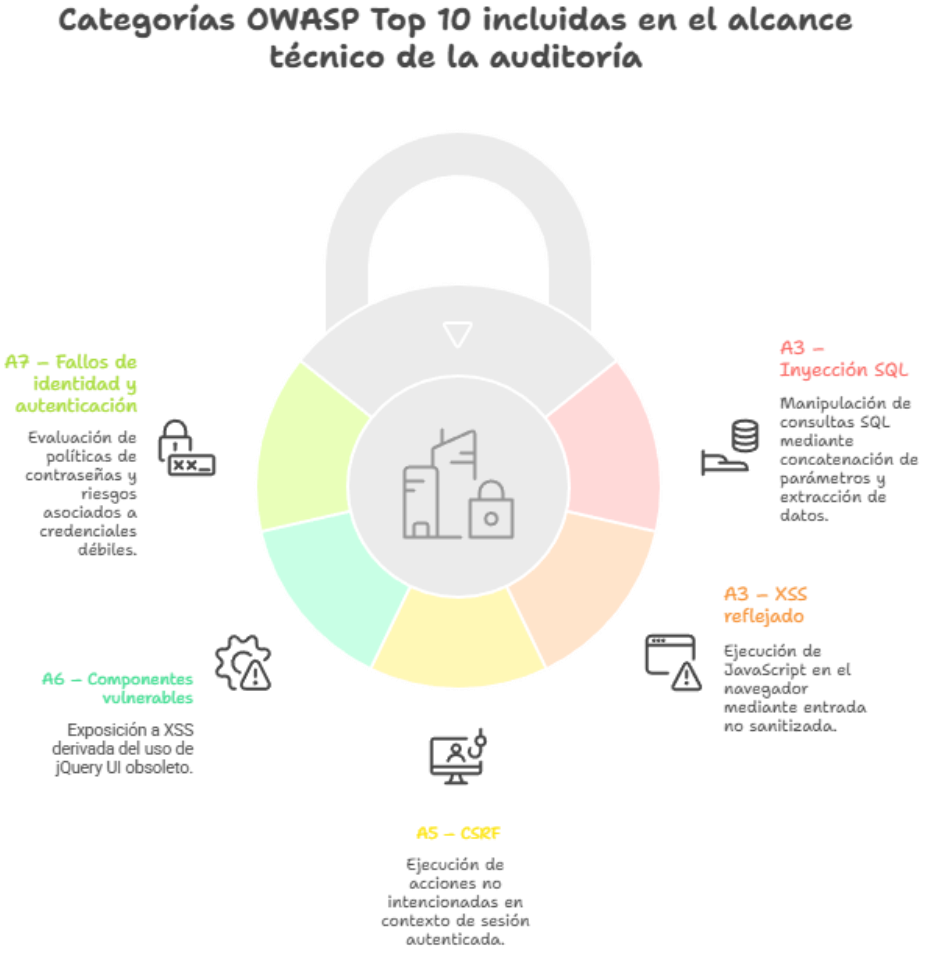
- **Aplicación objetivo:** WebGoat 8.1.0
- **Acceso:** navegador web en entorno local (http://127.0.0.1:8080/WebGoat)
- **Servidor:** Apache Tomcat embebido con Spring Boot
- **Base de datos:** HSQLDB (observada durante la explotación de ejercicios de inyección)
- **Contexto:** laboratorio aislado, sin exposición a Internet y sin impacto sobre terceros

La aplicación se ha desplegado mediante contenedores Docker y ejecutado dentro de una máquina virtual Kali Linux, garantizando el aislamiento del entorno y la reproducibilidad de las pruebas.

Alcance técnico

Dentro del laboratorio WebGoat se han trabajado y documentado ejercicios pertenecientes a las siguientes categorías del OWASP Top 10, según la estructura del propio entorno de aprendizaje:

- **A3 – Injection:**
    - SQL Injection (introducción y extracción de información).
- **A3 – Injection:**
    - Cross-Site Scripting (XSS reflejado).
- **A5 – Security Misconfiguration:**
    - Escenario de Cross-Site Request Forgery (CSRF).
- **A6 – Vulnerable & Outdated Components:**
    - Explotación de comportamiento inseguro derivado de dependencias vulnerables (jQuery UI).
- **A7 – Identity & Authentication Failures:**
    - Evaluación de fortaleza de contraseñas y análisis de riesgos asociados a credenciales débiles.



El gráfico representa únicamente las categorías OWASP Top 10 trabajadas dentro del alcance del laboratorio.

Fuera de alcance

Quedan explícitamente fuera del alcance de este informe:

- Pruebas sobre infraestructura real, redes externas, sistemas productivos o terceros.
- Ataques de denegación de servicio (DoS/DDoS).
- Explotaciones persistentes, movimientos laterales o escaladas fuera del laboratorio.
- Revisión completa del código fuente de la aplicación, más allá del análisis funcional necesario para los ejercicios.

Consideraciones éticas y legales

Todas las pruebas descritas se han realizado en un **entorno deliberadamente vulnerable**, con fines académicos y de aprendizaje en ciberseguridad.
En ningún caso se han llevado a cabo pruebas fuera del entorno autorizado ni con impacto sobre sistemas reales.

2. Informe ejecutivo

2.1 Resumen del proceso

Durante la práctica se accedió a la aplicación WebGoat 8.1.0 y se ejecutaron ejercicios representativos de las principales categorías del OWASP Top 10, siguiendo una metodología homogénea y reproducible.

Para cada ejercicio se aplicó un flujo de trabajo consistente, orientado a simular un proceso real de auditoría técnica:

1. Identificación del vector vulnerable (campo de entrada, endpoint o componente afectado).
2. Validación de la vulnerabilidad mediante una prueba de concepto (PoC) controlada.
3. Obtención de evidencias técnicas (capturas de navegador, consola, tráfico HTTP y terminal).
4. Análisis del impacto potencial sobre la confidencialidad, integridad y disponibilidad.
5. Propuesta de recomendaciones técnicas priorizadas.

En el caso de **Injection (SQL Injection)**, además de la validación manual del fallo, se documentó la automatización del proceso mediante herramientas de auditoría en entorno de laboratorio, demostrando cómo una vulnerabilidad inicial basada en concatenación de parámetros puede escalar rápidamente hacia la enumeración de esquemas y la extracción masiva de información sensible.

En el ejercicio de **Vulnerable & Outdated Components**, se refleja un aspecto relevante del propio diseño del laboratorio: el comportamiento vulnerable no siempre se manifiesta de forma explícita para el usuario final. En este escenario concreto, el exploit no generaba inicialmente un alert visible, mostrando en su lugar un error en la consola JavaScript. Como paso adicional se forzó la definición manual de la función

vulnerable desde la consola del navegador para demostrar el vector de ejecución, reforzando la comprensión del riesgo y aportando evidencia técnica adicional.

2.2 Vulnerabilidades destacadas

Resumen Ejecutivo – Hallazgos Clave (WebGoat 8.1.0)

Categoría OWASP	Vulnerabilidad	Severidad	Impacto principal (CIA)	Mitigación prioritaria
A3 Injection	SQL Injection	Crítica	Confidencialidad/Integridad	Consultas preparadas y mínimos privilegios
A3 Injection	XSS reflejado	Alta	Confidencialidad/Integridad	Sanitización por contexto + CSP
A5 Security Misconfiguration	CSRF	Alta	Integridad	Token ligado a sesión + Origin/Referer + SameSite
A6 Vulnerable & Outdated Components	jQuery UI vulnerable	Alta	Confidencialidad/Integridad	Gestión de dependencias/SCA + actualización
A7 Identity & Authentication Failures	contraseñas débiles/patrones	Media–Alta	Confidencialidad	Passphrases + MFA + rate limit

Las principales debilidades identificadas durante la práctica (en contexto de laboratorio controlado) fueron las siguientes:

- **SQL Injection (A3 – Injection)**
  Riesgo crítico debido a su capacidad para permitir bypass lógico, enumeración de esquemas, extracción de credenciales y exposición masiva de datos sensibles.
- **Cross-Site Scripting reflejado (A3 – Injection / XSS)**
  Riesgo alto por la posibilidad de ejecutar código JavaScript arbitrario en el navegador de la víctima, facilitando robo de sesión, manipulación del contenido y ataques posteriores.
- **Cross-Site Request Forgery (A5 – Security Misconfiguration)**
  Riesgo alto al permitir la ejecución de acciones en nombre de un usuario autenticado sin su interacción consciente ni validación suficiente del origen de la petición.
- **Uso de componentes vulnerables o desactualizados (A6 – Vulnerable & Outdated Components)**
  Riesgo alto derivado de dependencias externas vulnerables (jQuery UI),

evidenciando que la superficie de ataque no se limita al código propio de la aplicación.

- **Política de contraseñas y patrones previsibles (A7 – Identity & Authentication Failures)**
  Riesgo medio–alto, ya que una política basada únicamente en reglas formales puede resultar insuficiente si permite patrones comunes fácilmente explotables mediante ataques automatizados.

2.3 Conclusiones

El conjunto de ejercicios realizados pone de manifiesto un patrón común en la mayoría de fallos de seguridad: la confianza excesiva en entradas, contextos u orígenes sin una validación robusta, así como la falta de control efectivo sobre dependencias externas.

Los ejercicios de **Injection y XSS** evidencian deficiencias en la validación y sanitización de datos, así como en el uso de mecanismos seguros de acceso a bases de datos.
El escenario de **CSRF** demuestra que la presencia de un token por sí sola no es suficiente si no está correctamente vinculado a la sesión y al contexto de la petición.
Finalmente, el ejercicio de **componentes vulnerables** refuerza la idea de que el riesgo también reside en la cadena de suministro de software, y no únicamente en el desarrollo interno.

En conjunto, los resultados reflejan vulnerabilidades ampliamente documentadas y conocidas, pero que continúan apareciendo de forma recurrente en aplicaciones reales.

2.4 Recomendaciones

Como medidas transversales aplicables a entornos reales, se proponen las siguientes recomendaciones, priorizadas por impacto y madurez de seguridad:

1. Eliminar la concatenación directa de entradas en consultas SQL, utilizando siempre consultas preparadas y parametrización.
2. Aplicar validación y sanitización por contexto (HTML, atributos, JavaScript, URL), evitando enfoques genéricos.
3. Implementar protecciones completas frente a CSRF: tokens ligados a sesión, validación de Origin/Referer y cookies con atributo SameSite.
4. Establecer una gestión activa de dependencias (SCA), incluyendo inventario de componentes, revisión de CVEs y políticas de actualización.

5. Reforzar los mecanismos de autenticación mediante contraseñas robustas, listas de contraseñas prohibidas, limitación de intentos y autenticación multifactor (MFA).
6. Implantar sistemas de logging y monitorización orientados a eventos de seguridad, facilitando la detección temprana de intentos de explotación.

3. Metodología

Enfoque metodológico

La auditoría se ha realizado siguiendo una **metodología práctica y estructurada**, adaptada a un entorno de laboratorio formativo, pero alineada con los principios habituales de una auditoría técnica de aplicaciones web.

El proceso aplicado en todos los ejercicios sigue un ciclo iterativo común:

reconocimiento → identificación → validación → evidencia → análisis de impacto → mitigación

Este enfoque permite no solo confirmar la existencia de una vulnerabilidad, sino también comprender su alcance real, su impacto potencial y las medidas necesarias para reducir el riesgo.

Técnicas aplicadas

Durante la ejecución de los ejercicios se emplearon las siguientes técnicas, en función de la naturaleza de cada vulnerabilidad:

- **Análisis funcional en navegador**, observando el comportamiento de los formularios, campos de entrada, respuestas de la aplicación, mensajes de error y cambios de estado tras la interacción del usuario.
- **Inspección de tráfico HTTP y parámetros relevantes** en aquellos escenarios donde el laboratorio lo requería, con el objetivo de identificar campos controlables por el usuario y validar la lógica de las peticiones y respuestas.
- **Validación controlada mediante pruebas de concepto (PoC)**, introduciendo payloads específicos para confirmar la ejecución de código, la alteración de la lógica de la aplicación o el bypass de controles.
- **Automatización en entorno de laboratorio** en los escenarios de inyección SQL, con el fin de demostrar la escalabilidad del riesgo. Esta automatización permitió la enumeración de esquemas, tablas y columnas, así como la extracción controlada de información, siempre con fines formativos.

En determinados ejercicios, como el de **componentes vulnerables**, se complementó el comportamiento esperado del laboratorio con pruebas adicionales desde la consola del navegador, con el objetivo de reforzar la comprensión técnica del vector y aportar evidencias claras de ejecución.

Criterios de severidad

La severidad de las vulnerabilidades se ha estimado de forma cualitativa, teniendo en cuenta su impacto potencial sobre la **tríada CIA**:

- **Confidencialidad**: exposición de datos sensibles, credenciales, cookies o información interna.
- **Integridad**: modificación de datos, ejecución de acciones no autorizadas o manipulación del comportamiento de la aplicación.
- **Disponibilidad**: degradación del servicio o bloqueo, aunque este aspecto no fue el foco principal del laboratorio.

Adicionalmente, se ha considerado:

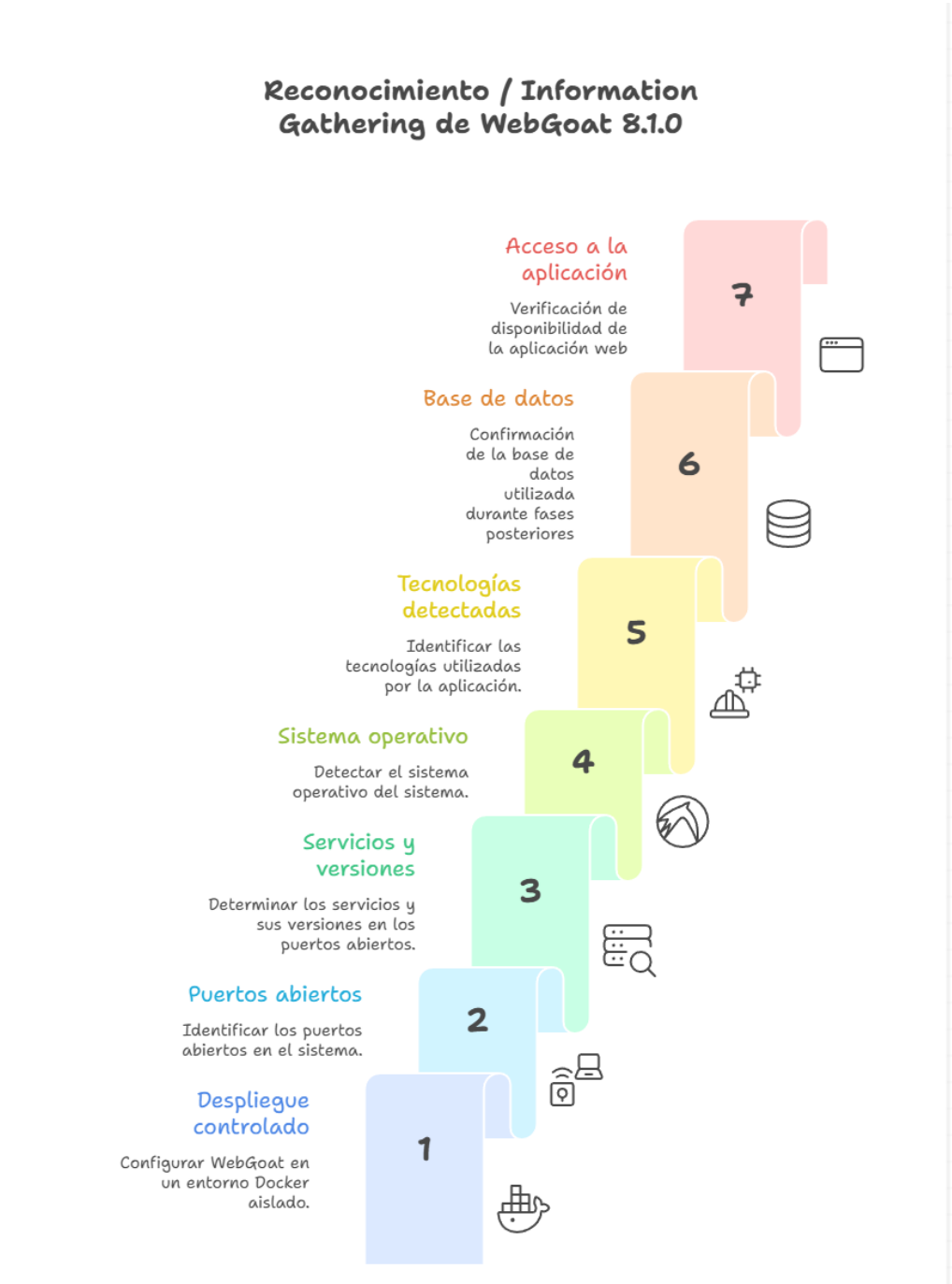
- La **facilidad de explotación** (payloads simples frente a escenarios complejos).
- La **superficie afectada** (impacto limitado frente a escalabilidad a múltiples usuarios, tablas o componentes).

Evidencias

Para garantizar la trazabilidad y reproducibilidad de los resultados, todas las conclusiones se han respaldado mediante evidencias técnicas, incluyendo:

- Capturas de navegador mostrando resultados de los ejercicios, mensajes de la aplicación y comportamientos observados.
- Capturas de la consola JavaScript, reflejando errores de ejecución y pruebas manuales en el contexto del navegador.
- Capturas de terminal, documentando procesos de enumeración, identificación de estructuras y extracción de información en los escenarios de inyección.

4. Reconocimiento / Information Gathering



El objetivo de esta fase es identificar la mayor cantidad posible de información técnica sobre la aplicación WebGoat 8.1.0 y su entorno de ejecución, sin realizar aún una explotación activa, siguiendo un enfoque de reconocimiento pasivo y semi-activo.

4.1 Despliegue del entorno

La aplicación WebGoat 8.1.0 se ha desplegado correctamente dentro de un contenedor Docker, ejecutándose sobre una máquina virtual Kali Linux. Este enfoque permite trabajar en un entorno aislado, reproducible y controlado, adecuado para prácticas de auditoría web.

El contenedor expone los puertos necesarios para el acceso a la aplicación web y sus servicios asociados.

4.2 Identificación de puertos abiertos

Para identificar los servicios accesibles desde el exterior, se realizó un escaneo de puertos sobre el host local utilizando la herramienta **nmap**, centrando el análisis en los puertos habituales de la aplicación.

Se ejecutó el siguiente comando:

nmap -p 8080,9090 127.0.0.1

El resultado del escaneo confirmó que los siguientes puertos se encuentran abiertos:

- **8080/tcp** – Servicio HTTP
- **9090/tcp** – Servicio HTTP adicional

Estos puertos corresponden a los servicios web expuestos por la aplicación WebGoat.

4.3 Detección de servicios y versiones

Con el fin de identificar el software que escucha en los puertos detectados, se realizó un escaneo de detección de servicios y versiones mediante el siguiente comando:

nmap -p 8080,9090 -sV 127.0.0.1

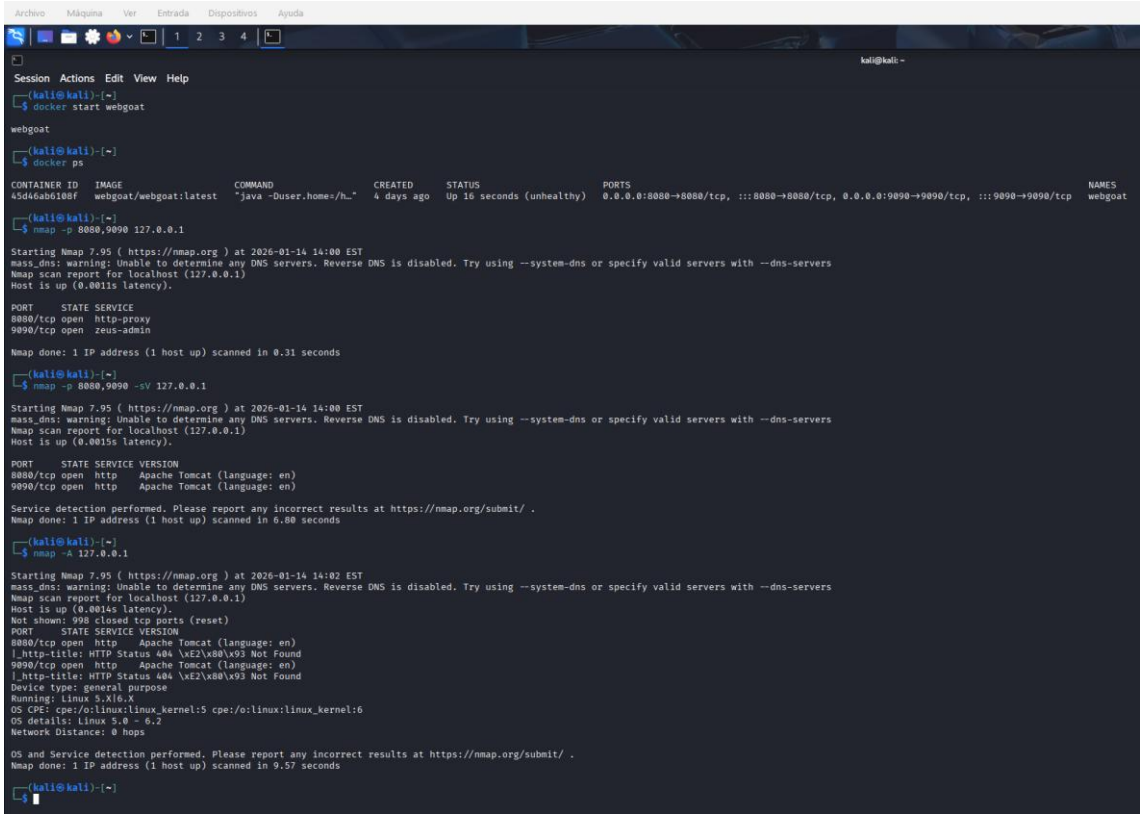
El análisis permitió identificar que ambos puertos están gestionados por un servidor **Apache Tomcat**, utilizado como contenedor de aplicaciones Java. Esta información concuerda con el uso de **Spring Boot** como framework de desarrollo de la aplicación.

4.4 Identificación del sistema operativo

Para obtener información adicional sobre el sistema operativo subyacente y el tipo de dispositivo, se ejecutó un escaneo más completo mediante:

nmap -A 127.0.0.1

Los resultados indican que el sistema operativo subyacente corresponde a un entorno **Linux**, coherente con la ejecución de Docker sobre Kali Linux y con el uso de tecnologías habituales en servidores web Java.



4.5 Tecnologías identificadas

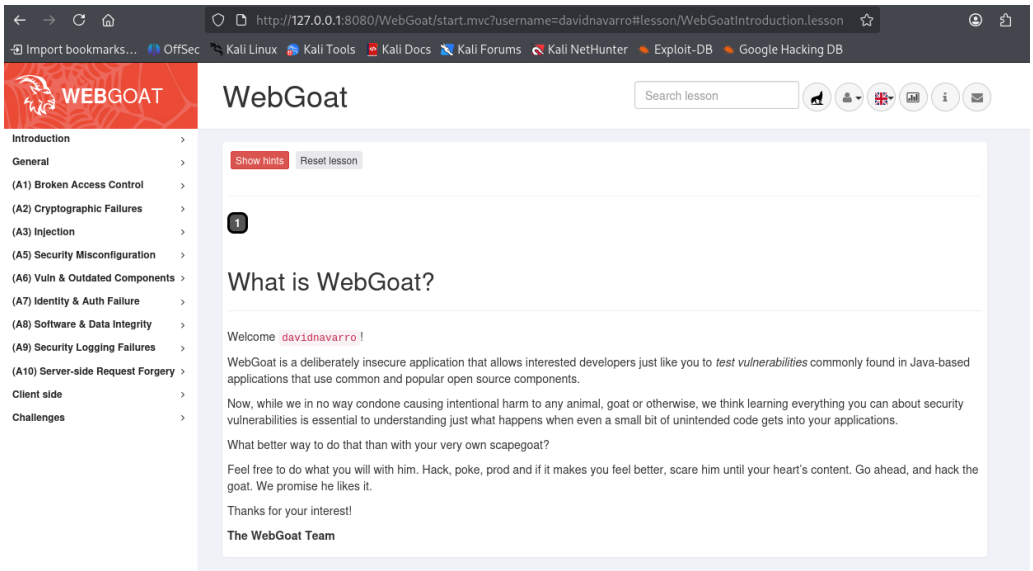
A partir del reconocimiento realizado, se identifican las siguientes tecnologías clave en el entorno auditado:

- **Sistema operativo:** Linux
- **Contenerización:** Docker
- **Servidor web / contenedor de aplicaciones:** Apache Tomcat
- **Framework de desarrollo:** Spring Boot
- **Base de datos:** HSQLDB (versión 2.7.x, observada durante la explotación posterior)
- **Protocolos expuestos:** HTTP
- **Puertos en uso:** 8080 y 9090

4.6 Acceso a la aplicación

La aplicación WebGoat 8.1.0 es accesible correctamente desde un navegador web a través de la siguiente URL:

http://127.0.0.1:8080/WebGoat



Una vez autenticado el usuario, se muestra la interfaz principal del laboratorio, organizada por secciones basadas en el **OWASP Top 10**, que servirán como base para la detección y explotación de vulnerabilidades en las fases posteriores de la auditoría.

5. Explotación de vulnerabilidades

5.1 A3 – SQL Injection (Intro) – Apartado 11

Descripción de la vulnerabilidad

En este ejercicio se ha identificado una vulnerabilidad de tipo **SQL Injection (OWASP A3 – Injection)** provocada por una construcción insegura de consultas SQL mediante concatenación directa de los parámetros introducidos por el usuario, sin ningún tipo de validación ni uso de consultas preparadas.

La aplicación genera dinámicamente la siguiente consulta SQL:

SELECT * FROM employees

WHERE last_name = '<name>'

AND auth_tan = '<auth_tan>';

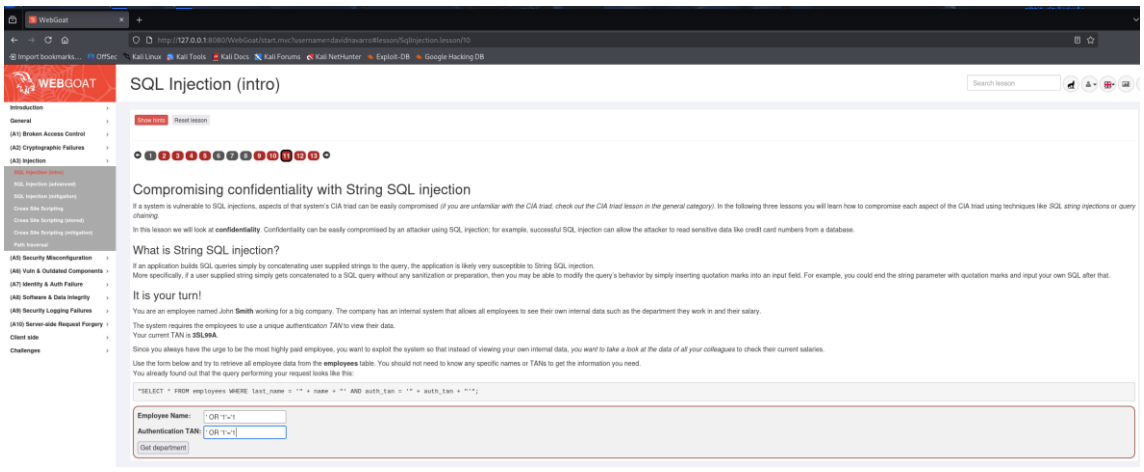
Esta implementación permite que un atacante modifique la lógica de la cláusula WHERE introduciendo código SQL malicioso en los campos de entrada.

Prueba de concepto (PoC)

Para explotar la vulnerabilidad, se introdujo el siguiente payload en ambos campos del formulario (*Employee Name* y *Authentication TAN*):

' OR '1'='1

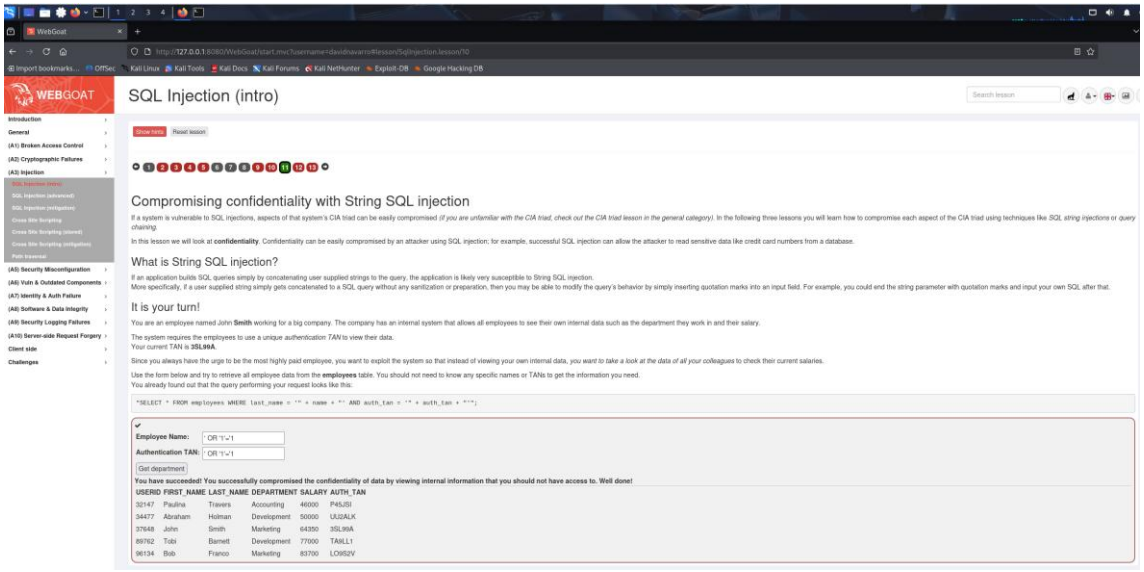
Este payload fuerza a que la condición de la consulta SQL sea siempre verdadera, anulando el mecanismo de autenticación basado en el valor TAN.



Resultado obtenido

Como resultado de la inyección SQL, la aplicación devolvió información de **todos los empleados** almacenados en la base de datos, incluyendo:

- Identificadores de usuario
- Nombres y apellidos
- Departamentos
- Salarios
- Valores de autenticación (TAN)



Impacto

Esta vulnerabilidad permite a un atacante acceder a información sensible de otros usuarios sin necesidad de conocer credenciales válidas, comprometiendo gravemente la **confidencialidad de los datos**, uno de los pilares de la tríada CIA.

En un entorno real, este tipo de fallo podría derivar en filtraciones de información crítica como datos personales o financieros.

5.2 A3 – SQL Injection – Extracción de información

Descripción de la vulnerabilidad

En este ejercicio se ha identificado una vulnerabilidad de tipo **SQL Injection (OWASP A3 – Injection)** en uno de los formularios de la aplicación WebGoat.
La vulnerabilidad se origina por la **construcción insegura de consultas SQL mediante concatenación directa de parámetros introducidos por el usuario**, sin aplicar validación, sanitización ni el uso de consultas preparadas.

La aplicación genera dinámicamente una consulta SQL similar a la siguiente:

SELECT * FROM employees

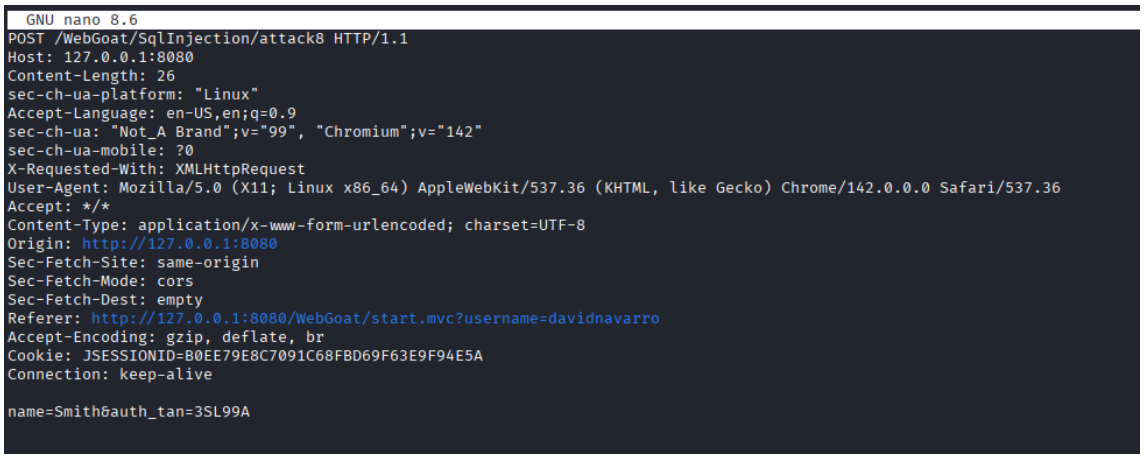
WHERE last_name = '<name>'

AND auth_tan = '<auth_tan>';

Ambos parámetros (name y auth_tan) se insertan directamente en la consulta SQL, permitiendo a un atacante **alterar la lógica de la cláusula WHERE** y ejecutar consultas arbitrarias sobre la base de datos.

Identificación de la vulnerabilidad

Para identificar el punto vulnerable se interceptó la petición HTTP correspondiente al ejercicio *SqlInjection/attack8*, observando que los valores introducidos por el usuario se envían directamente al backend.



Se muestra la petición POST /WebGoat/SqlInjection/attack8 con los parámetros:

name=Smith

auth_tan=3SL99A

El análisis de la petición HTTP generada por el formulario permitió confirmar que **ambos parámetros (name y auth_tan) son controlables por el usuario y se insertan directamente en la consulta SQL sin validación ni sanitización**.

Las pruebas iniciales demostraron que **los dos campos son susceptibles de inyección SQL**, ya que ambos se concatenan como cadenas dentro de la cláusula WHERE.

No obstante, para la explotación y documentación del ejercicio se seleccionó el parámetro **auth_tan como vector principal de inyección**, al permitir un control más directo de la lógica de autenticación y facilitar la enumeración y extracción de información de la base de datos.

Posteriormente, se introdujo un payload lógico en el parámetro auth_tan para modificar el comportamiento de la consulta, confirmando que el backend no valida ni filtra correctamente la entrada del usuario, y que el campo es vulnerable a SQL Injection.

Prueba de concepto (PoC) – Explotación manual (UNION-based SQL Injection)

Antes de automatizar la explotación con herramientas, se validó manualmente la vulnerabilidad para comprender en detalle el funcionamiento interno de la inyección SQL y confirmar el control total sobre la consulta ejecutada por el backend.

A partir de la consulta generada por la aplicación:

SELECT * FROM employees

WHERE last_name = '<name>'

AND auth_tan = '<auth_tan>';

se observa que ambos parámetros (name y auth_tan) se insertan como cadenas de texto, ya que se encuentran delimitados por comillas simples. Esto confirma que ambos campos son potencialmente explotables mediante SQL Injection.

Determinación del número de columnas

Al introducir el payload lógico:

' OR '1'='1

en el parámetro auth_tan, la aplicación devuelve resultados válidos, confirmando la vulnerabilidad.
Adicionalmente, se comprobó que la consulta devuelve **6 columnas**, requisito imprescindible para la construcción de consultas UNION SELECT.

Como alternativa metodológica, el número de columnas podría haberse determinado mediante la técnica:

' ORDER BY 1 --

' ORDER BY 2 --

...

incrementando el índice hasta provocar un error, aunque en este caso no fue necesario.

Enumeración manual de tablas

Una vez conocido el número de columnas, se procedió a enumerar las tablas de la base de datos utilizando el esquema information_schema:

' UNION SELECT table_name, NULL, NULL, NULL, NULL, NULL

FROM information_schema.tables --

Esta consulta devuelve un listado amplio de tablas, incluyendo tanto tablas del sistema como tablas propias del laboratorio.

Enumeración de columnas de tablas específicas

Seleccionando una tabla concreta, se enumeraron sus columnas mediante:

' UNION SELECT column_name, NULL, NULL, NULL, NULL, NULL

FROM information_schema.columns

WHERE table_name = 'SALARIES' --

Lo que permitió identificar las columnas SALARY y USERID.

Extracción de datos y manejo de tipos

Al intentar extraer los datos directamente:

' UNION SELECT SALARY, USERID, NULL, NULL, NULL, NULL

FROM SALARIES --

se produjo un error de incompatibilidad de tipos, ya que el campo SALARY es numérico y la columna de salida esperada es de tipo texto.

Este problema se resolvió concatenando los valores mediante CONCAT, forzando el resultado a tipo string:

' UNION SELECT CONCAT(SALARY,'|', USERID), NULL, NULL, NULL, NULL, NULL

FROM SALARIES --

Esta técnica permitió visualizar correctamente los datos extraídos.
El mismo enfoque se aplicó posteriormente a otras tablas accesibles del esquema del usuario, como USER_DATA, USER_DATA_TAN y USER_SYSTEM_DATA.

Estas pruebas manuales confirman que la vulnerabilidad permite no solo el bypass lógico, sino también la enumeración y extracción controlada de información sensible, demostrando un entendimiento completo del vector de ataque antes de su automatización.

Prueba de concepto (PoC) – Automatización con sqlmap

Una vez confirmada la vulnerabilidad, se procedió a su explotación mediante la herramienta **sqlmap**, utilizando la petición HTTP capturada previamente como entrada (-r at8.txt) y especificando el parámetro vulnerable auth_tan.

Detección automática de la inyección

Se ejecutó el siguiente comando:

sqlmap -r at8.txt -p auth_tan --tables --batch

sqlmap identificó correctamente:

- El parámetro auth_tan como vulnerable.
- El motor de base de datos **HSQLDB (v2.7.2)**.
- Técnicas de inyección basadas en:

- Boolean-based blind
- UNION-based

A continuación, sqlmap comenzó la enumeración de tablas accesibles agrupadas por base de datos.

La opción --batch se utilizó para ejecutar sqlmap en modo no interactivo, aceptando automáticamente las opciones por defecto.
Esto permite una ejecución reproducible del ataque, evita interrupciones durante la enumeración y garantiza que los resultados obtenidos sean consistentes y fácilmente documentables en el contexto del laboratorio.

Enumeración de tablas por base de datos

INFORMATION_SCHEMA

Se muestra el listado de **98 tablas** pertenecientes al esquema INFORMATION_SCHEMA, accesibles mediante la inyección SQL.

Este esquema contiene metadatos del sistema, como definiciones de tablas, columnas, restricciones y privilegios.

SYSTEM_LOBS y base de datos del usuario

```
Database: SYSTEM_LOBS
[4 tables]
+-----------------------------------+
| BLOCKS                            |
| LOBS                              |
| LOB_IDS                           |
| PARTS                             |
+-----------------------------------+

Database: davidnavarro
[13 tables]
+-----------------------------------+
| ACCESS_CONTROL_USERS              |
| ACCESS_LOG                        |
| CHALLENGE_USERS                   |
| EMPLOYEES                         |
| GRANT_RIGHTS                      |
| JWT_KEYS                          |
| SALARIES                          |
| SERVERS                           |
| SQL_CHALLENGE_USERS               |
| USER_DATA                         |
| USER_DATA_TAN                     |
| USER_SYSTEM_DATA                  |
| flyway_schema_history             |
+-----------------------------------+
```

Se observa la enumeración de:

- SYSTEM_LOBS con **4 tablas** (BLOCKS, LOBS, LOB_IDS, PARTS)
- davidnavarro con **13 tablas**, entre ellas:
    - EMPLOYEES
    - SALARIES
    - USER_DATA
    - USER_SYSTEM_DATA
    - ACCESS_LOG
    - SQL_CHALLENGE_USERS

Esta enumeración confirma que la inyección permite acceder a **esquemas de aplicación y de usuario**, no únicamente a metadatos.

Volcado masivo de información (dump-all)

Se ejecutó:

sqlmap -r at8.txt -p auth_tan --dump-all --batch

sqlmap procedió a volcar información de múltiples tablas, exportando los resultados a ficheros CSV.
Entre los datos extraídos se incluyen tablas relacionadas con el progreso de lecciones y otros datos internos de la aplicación.

Explotación dirigida por base de datos

SYSTEM_LOBS

Se ejecutó:

sqlmap -r at8.txt -p auth_tan -D SYSTEM_LOBS --tables --batch

Obteniéndose el listado de tablas del esquema SYSTEM_LOBS.

Posteriormente:

sqlmap -r at8.txt -p auth_tan -D SYSTEM_LOBS --dump-all --batch

sqlmap volcó el contenido de las tablas disponibles, aunque algunas de ellas aparecían sin registros.

Consideración técnica: limitación por mayúsculas/minúsculas en el esquema

Durante la explotación se detectó un **comportamiento anómalo relacionado con el nombre del esquema de base de datos del usuario**.

WebGoat crea un esquema por usuario con el mismo nombre que el identificador de usuario. Sin embargo:

- HSQLDB gestiona internamente los nombres de esquemas en mayúsculas.
- WebGoat **no permite crear usuarios con letras en mayúscula**.

Como consecuencia, sqlmap presentaba dificultades para enumerar correctamente el esquema asociado al usuario davidnavarro.

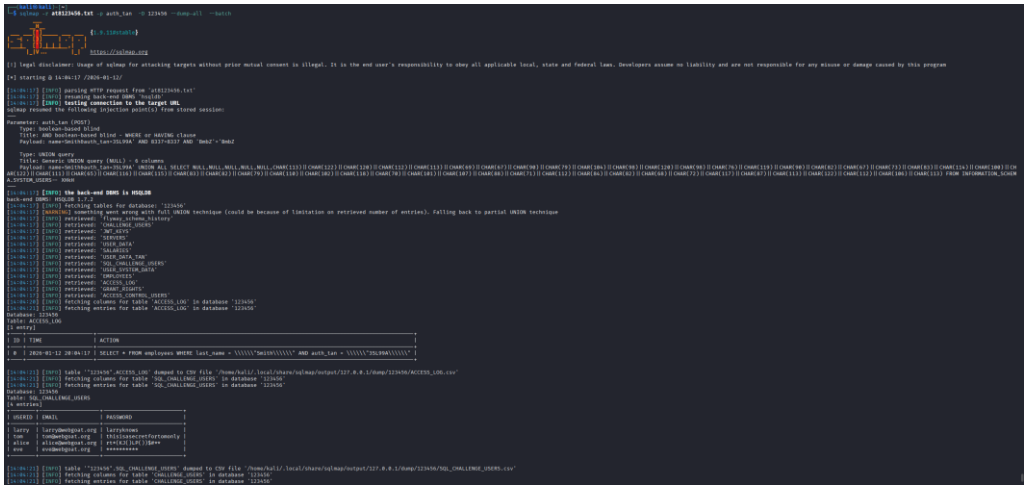
Solución aplicada

Para evitar esta limitación, se creó un **usuario compuesto únicamente por caracteres numéricos**:

Usuario: 123456

Al no verse afectado por el tratamiento de mayúsculas/minúsculas, sqlmap pudo enumerar y explotar correctamente el esquema asociado.

Explotación completa del esquema del usuario numérico



Se ejecutó:

sqlmap -r at8123456.txt -p auth_tan -D 123456 --dump-all --batch

Se extrajeron datos sensibles de múltiples tablas, incluyendo:

- SQL_CHALLENGE_USERS (usuarios y contraseñas)
- ACCESS_LOG
- EMPLOYEES
- SALARIES
- USER_SYSTEM_DATA

Enumeración de columnas

Comando utilizado:

sqlmap -r at8123456.txt -p auth_tan -D 123456 --columns --batch

Se obtuvieron las columnas de todas las tablas del esquema del usuario.

Comando:

sqlmap -r at8123456.txt -p auth_tan -D 123456 -T salaries --columns --batch

Columnas identificadas:

- SALARY (INTEGER)
- USERID (VARCHAR)

Dump selectivo de una tabla concreta

Finalmente se realizó un volcado selectivo:

sqlmap -r at8123456.txt -p auth_tan -D 123456 -T salaries --dump --batch

Se extrajeron registros con identificadores de usuario y salarios asociados.

Resultado obtenido

Mediante la explotación de esta vulnerabilidad fue posible:

- Confirmar la existencia de SQL Injection.
- Enumerar tablas en múltiples esquemas.
- Extraer información sensible de usuarios.
- Automatizar completamente la explotación mediante sqlmap.
- Superar una limitación técnica de la aplicación creando un usuario alternativo.

Impacto

En un entorno real, esta vulnerabilidad permitiría a un atacante:

- Acceder a información sensible sin autenticación válida.
- Exfiltrar credenciales y datos personales.
- Obtener conocimiento completo de la estructura de la base de datos.
- Facilitar ataques posteriores como escaladas de privilegios.

El impacto sobre la **confidencialidad e integridad de la información es crítico**.

Recomendaciones

- Uso exclusivo de **consultas preparadas (Prepared Statements)**.
- Eliminación de concatenación directa de parámetros SQL.
- Validación y sanitización estricta de todas las entradas.
- Revisión del manejo de esquemas y mayúsculas/minúsculas.
- Aplicación del principio de **mínimos privilegios** en la base de datos.
- Monitorización y detección de intentos de inyección SQL.

5.3 A3 – Cross Site Scripting – Apartado 7

Descripción de la vulnerabilidad

En este ejercicio se ha identificado una vulnerabilidad de tipo **Cross Site Scripting reflejado (Reflected XSS)** en uno de los campos del formulario del carrito de compra de la aplicación WebGoat. Este tipo de vulnerabilidad se produce cuando la aplicación incorpora directamente en la respuesta HTTP datos introducidos por el usuario sin aplicar una validación o sanitización adecuada.

Como consecuencia, un atacante puede inyectar código JavaScript que será ejecutado en el navegador de la víctima cuando esta interactúe con la aplicación.

Prueba de concepto (PoC)

Para identificar qué campo era vulnerable a XSS, se introdujo código JavaScript de prueba en los distintos campos del formulario. Como payload se utilizó el siguiente script:

<script>alert(1)</script>

El payload fue introducido en el campo **“Enter your credit card number”**, manteniendo valores válidos en el resto de campos, y posteriormente se envió el formulario mediante el botón *Purchase*.

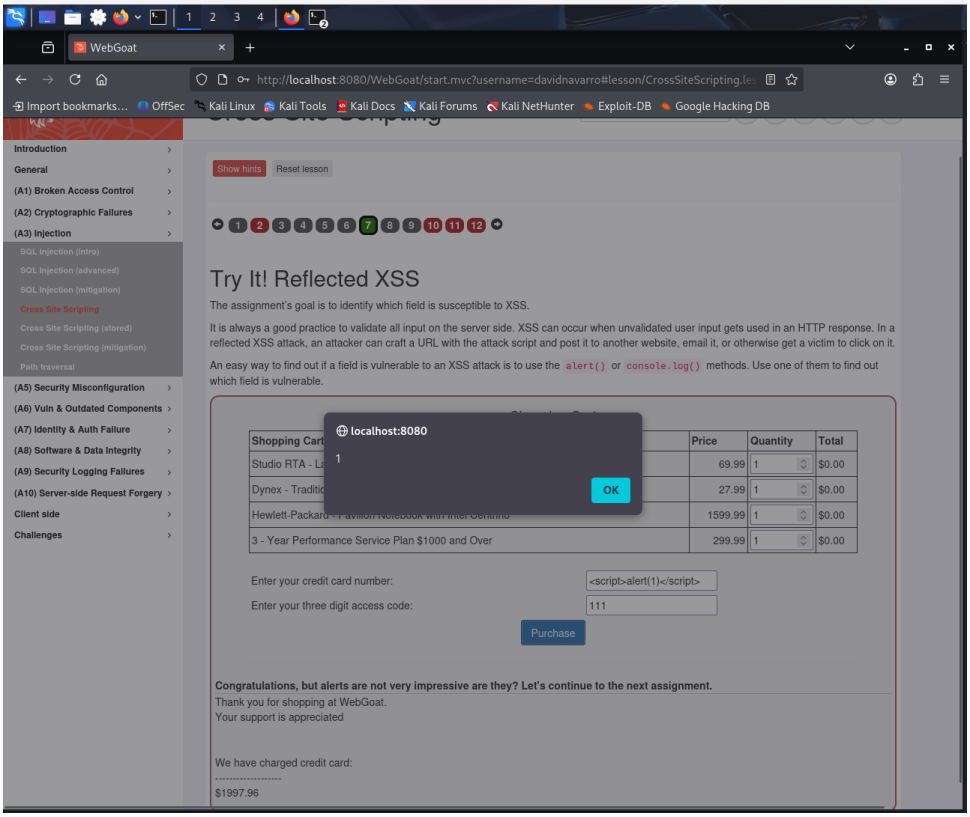
Resultado obtenido

Tras el envío del formulario, el navegador ejecutó el código JavaScript introducido, mostrando un cuadro de diálogo (alert). Este comportamiento confirma que la entrada del usuario es reflejada en la respuesta sin ser correctamente filtrada, validando la existencia de una vulnerabilidad de tipo XSS reflejado en dicho campo.

Como evidencia de la explotación se adjunta una captura de pantalla en la que se observa la ejecución del alert(1) tras el procesamiento del formulario.

Impacto

La presencia de esta vulnerabilidad permitiría a un atacante ejecutar código JavaScript arbitrario en el navegador de otros usuarios. En un entorno real, este tipo de ataque podría utilizarse para el robo de cookies de sesión, la manipulación del contenido de la página o la redirección de los usuarios a sitios web maliciosos, comprometiendo la seguridad de la aplicación y la información de los usuarios.



5.4 A5 – Security Misconfiguration – Apartado 4

Descripción de la vulnerabilidad

En este ejercicio se ha identificado una vulnerabilidad de tipo **Cross-Site Request Forgery (CSRF)**, incluida dentro de OWASP Top 10, que permite a un atacante forzar a un usuario autenticado a ejecutar una acción sin su consentimiento explícito.

La aplicación WebGoat no valida correctamente el **origen real de la solicitud**, permitiendo que una petición POST legítima sea enviada desde un sitio externo, siempre que el usuario mantenga una sesión activa en el navegador.

En este escenario concreto, la acción vulnerable consiste en **publicar una reseña (review)** en nombre del usuario autenticado, sin que este interactúe directamente con el formulario original de la aplicación.

Análisis técnico

Durante el análisis del formulario de envío de reseñas, se observa que la aplicación realiza una petición HTTP POST al endpoint:

/WebGoat/csrf/review

Dicha petición contiene los siguientes parámetros relevantes:

- reviewText – Texto de la reseña
- stars – Valor de puntuación
- validateReq – Token de validación CSRF

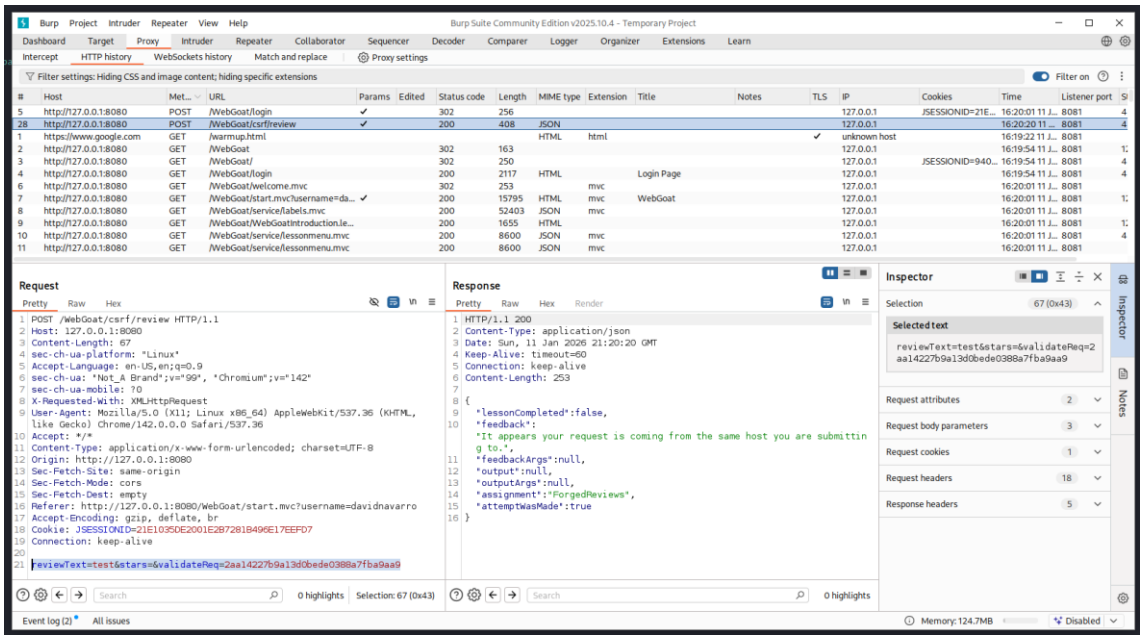
Aunque existe un parámetro de validación (validateReq), este **no está correctamente protegido**, ya que puede ser reutilizado desde un origen externo sin comprobaciones adicionales del contexto o del origen de la petición.

Prueba de concepto (PoC)

Paso 1 – Captura de la petición legítima

Desde el navegador autenticado (Firefox), se envió una reseña de prueba utilizando el formulario original de WebGoat.
La petición fue interceptada y analizada mediante Burp Suite, identificando el cuerpo completo de la solicitud POST:

reviewText=test&stars=&validateReq=2aa14227b9a13d0bede0388a7fba9aa9



Paso 2 – Construcción del exploit CSRF

Con la información obtenida, se creó un archivo HTML malicioso (csrf.html) que envía automáticamente la misma petición POST al servidor de WebGoat, simulando el comportamiento de un ataque CSRF real.

El archivo contiene un formulario oculto que se envía automáticamente al cargarse la página:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>CSRF PoC</title>

</head>

<body onload="document.forms[0].submit()">


<form action="http://127.0.0.1:8080/WebGoat/csrf/review" method="POST">

    <input type="hidden" name="reviewText" value="CSRF attack successful">

    <input type="hidden" name="stars" value="5">
```

<input type="hidden" name="validateReq" value="2aa14227b9a13d0bede0388a7fba9aa9">

</form>

</body>

</html>

Paso 3 – Ejecución del ataque

El archivo csrf.html fue abierto desde el sistema local mientras el usuario permanecía autenticado en WebGoat en el mismo navegador.

Al cargarse la página, el formulario se envió automáticamente al endpoint vulnerable, publicando la reseña sin interacción directa del usuario con la aplicación.

Resultado obtenido

Tras la ejecución del exploit, la aplicación respondió indicando que la solicitud había sido aceptada correctamente desde un origen externo.

La respuesta del servidor confirmó la finalización del ejercicio:

{

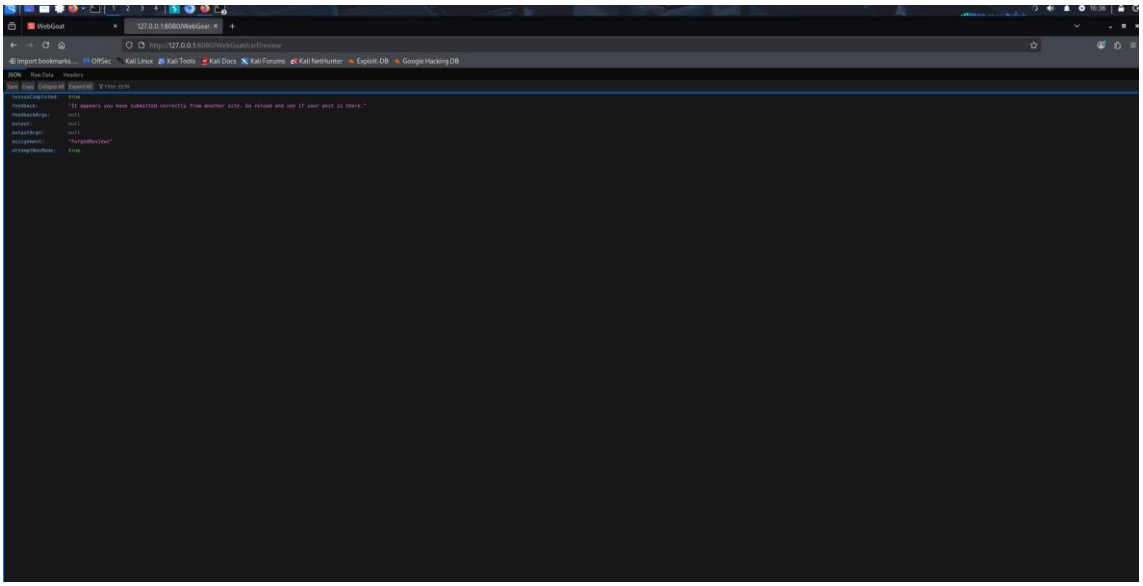
"lessonCompleted": true,

"feedback": "It appears you have submitted correctly from another site. Go reload and see if your post is there.",

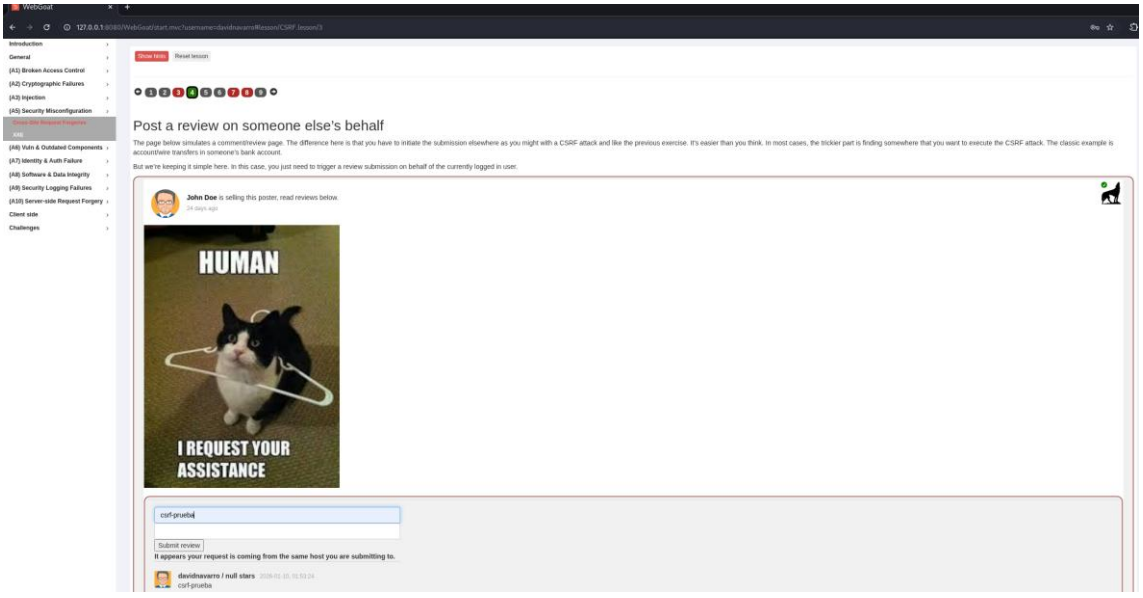
"assignment": "ForgedReviews",

"attemptWasMade": true

}



Además, al recargar la página del ejercicio en WebGoat, se observa que la reseña ha sido publicada correctamente en nombre del usuario autenticado.



Impacto

Esta vulnerabilidad permite a un atacante forzar acciones críticas en nombre de usuarios legítimos, siempre que estos mantengan una sesión activa. En un entorno real, este tipo de fallo podría permitir:

- Publicación de contenido no autorizado
- Transferencias de fondos

- Cambios de configuración
- Acciones administrativas involuntarias

Todo ello sin que la víctima sea consciente del ataque, comprometiendo gravemente la **integridad** y **confianza** en la aplicación.

Recomendaciones

Para mitigar este tipo de vulnerabilidades se recomienda:

- Implementar tokens CSRF **ligados a la sesión y al origen**
- Validar el encabezado Origin y/o Referer
- Utilizar cookies con atributos SameSite=Strict
- Requerir interacción explícita del usuario en acciones sensibles
- Evitar la reutilización de tokens CSRF

Notas finales

Este ejercicio demuestra de forma práctica cómo una protección CSRF mal implementada puede ser fácilmente burlada, incluso cuando existe un token de validación aparente, reforzando la importancia de una correcta implementación de controles de seguridad defensivos.

5.5 A6 – Vulnerable & Outdated Components – Apartado 5

Descripción de la vulnerabilidad

En este ejercicio se analiza una vulnerabilidad de tipo **Cross-Site Scripting (XSS)** derivada del uso de **componentes externos vulnerables**, concretamente una versión obsoleta de la librería **jQuery UI 1.10.4**.

La vulnerabilidad no reside en el código desarrollado por la propia aplicación, sino en el **comportamiento inseguro de una dependencia externa**, lo que encaja con la categoría **OWASP A6 – Vulnerable and Outdated Components**.

El ejercicio pone de manifiesto que, aunque el código de la aplicación no haya sido modificado, el uso de librerías JavaScript desactualizadas puede introducir vulnerabilidades críticas que permitan la ejecución de código en el navegador del usuario.

Identificación de la vulnerabilidad

WebGoat presenta dos escenarios comparativos utilizando el mismo código funcional, diferenciándose únicamente en la versión de la librería empleada:

- **jQuery UI 1.10.4** (versión vulnerable)
- **jQuery UI 1.12.0** (versión corregida)

En el escenario vulnerable, el componente permite que el contenido del botón del diálogo jQuery se renderice sin un filtrado adecuado, lo que abre la puerta a la inyección y ejecución de código JavaScript.

El payload propuesto por el ejercicio es el siguiente:

OK<script>alert("XSS")</script>

Este payload debería ejecutarse al interactuar con el diálogo cuando se utiliza la versión vulnerable de la librería.

Prueba de concepto (PoC)

Durante la ejecución del ejercicio se observó que, al pulsar el botón **Go**, el comportamiento esperado no se producía directamente. En su lugar, la consola del navegador mostraba el siguiente error:

Uncaught TypeError: webgoat.customjs.jqueryVuln is not a function

Este mensaje indica que la función JavaScript responsable del flujo vulnerable no se encontraba disponible o correctamente enlazada en ese contexto concreto del laboratorio.

Este comportamiento no invalida la vulnerabilidad en sí, sino que evidencia un **problema interno en la ejecución del ejercicio**, relacionado con el uso del componente vulnerable, y no con la validación o sanitización del input.

Como validación adicional y con fines estrictamente demostrativos, se definió manualmente en la consola del navegador la función a la que hace referencia el ejercicio:

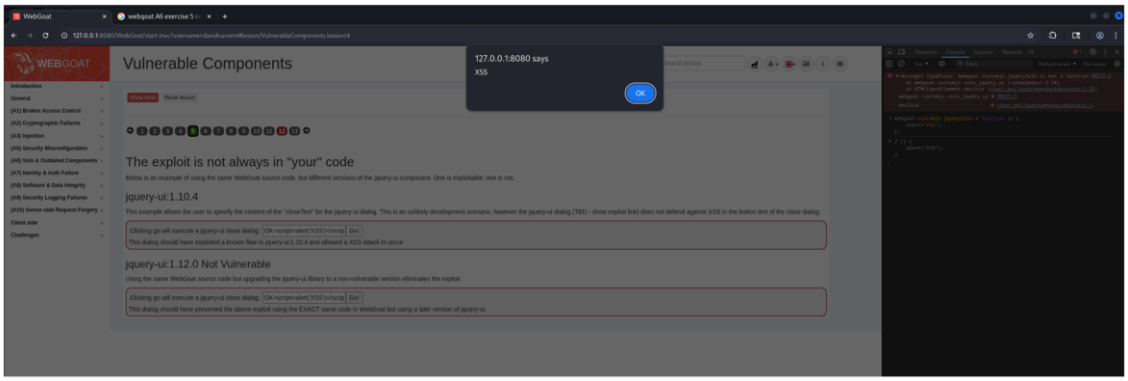
```
webgoat.customjs.jqueryVuln = function() {
  alert("XSS");
};
```

Posteriormente, al ejecutar dicha función desde la consola, se produjo correctamente la ejecución del código JavaScript y la aparición del popup correspondiente:

127.0.0.1:8080 says

XSS

Este paso **no constituye por sí mismo una explotación de XSS**, ya que la definición manual de funciones en la consola es una capacidad estándar de cualquier navegador y no implica persistencia ni control del flujo desde entradas externas. No obstante, permitió **verificar el comportamiento esperado del ejercicio** y confirmar que el problema observado está vinculado al uso del componente vulnerable, y no a una restricción del entorno de ejecución JavaScript.



En la captura asociada se puede observar:

- El contexto del ejercicio en WebGoat
- El error JavaScript mostrado en la consola
- La definición manual de la función vulnerable
- La ejecución del alert("XSS") como demostración controlada

Resultado obtenido

El ejercicio confirma que:

- El uso de **jQuery UI 1.10.4** introduce una vulnerabilidad XSS conocida.
- La versión **jQuery UI 1.12.0** corrige este comportamiento sin necesidad de modificar el código de la aplicación.
- Incluso cuando el exploit no se ejecuta automáticamente debido a un fallo interno del laboratorio, el análisis del comportamiento y la validación manual permiten demostrar que el **vector de riesgo existe**.

Esto refuerza el mensaje principal del ejercicio: el riesgo no siempre reside en el código propio, sino en las **dependencias externas utilizadas por la aplicación**.

Impacto

En un entorno real, una vulnerabilidad de este tipo permitiría a un atacante:

- Ejecutar código JavaScript en el navegador de las víctimas.
- Robar cookies de sesión o tokens de autenticación.
- Realizar acciones en nombre del usuario autenticado.
- Facilitar ataques más avanzados como phishing, keylogging o redirecciones maliciosas.

El impacto afecta directamente a la **confidencialidad, integridad y confianza** en la aplicación web.

Recomendaciones

Para mitigar este tipo de vulnerabilidades se recomienda:

- Mantener todas las dependencias externas actualizadas, especialmente librerías JavaScript.
- Evitar el uso de versiones obsoletas con vulnerabilidades conocidas.
- Implementar procesos de **gestión de dependencias** y revisión periódica de CVEs.
- Utilizar herramientas de **Software Composition Analysis (SCA)**.
- No asumir que el código propio es seguro si se utilizan componentes de terceros sin control.
- Complementar estas medidas con políticas de **Content Security Policy (CSP)** para reducir el impacto de posibles XSS.

5.6 A7 – Identity & Auth Failure – Secure Passwords – Apartado 4

Descripción de la vulnerabilidad

En este ejercicio se analiza el impacto que tiene el uso de contraseñas débiles frente a ataques de fuerza bruta. La aplicación evalúa la fortaleza de las contraseñas introducidas estimando el número de combinaciones necesarias para su descifrado y el tiempo aproximado requerido para romperlas.

El objetivo del ejercicio no es explotar una vulnerabilidad de forma activa, sino **demostrar cómo una política de contraseñas débil facilita ataques de fuerza bruta**, constituyendo un fallo de seguridad de tipo **OWASP A7 – Identity & Authentication Failure**.

Prueba de concepto (PoC)

Para completar el ejercicio se introdujo una contraseña diseñada para cumplir criterios reales de seguridad, evitando patrones triviales o diccionarios comunes.

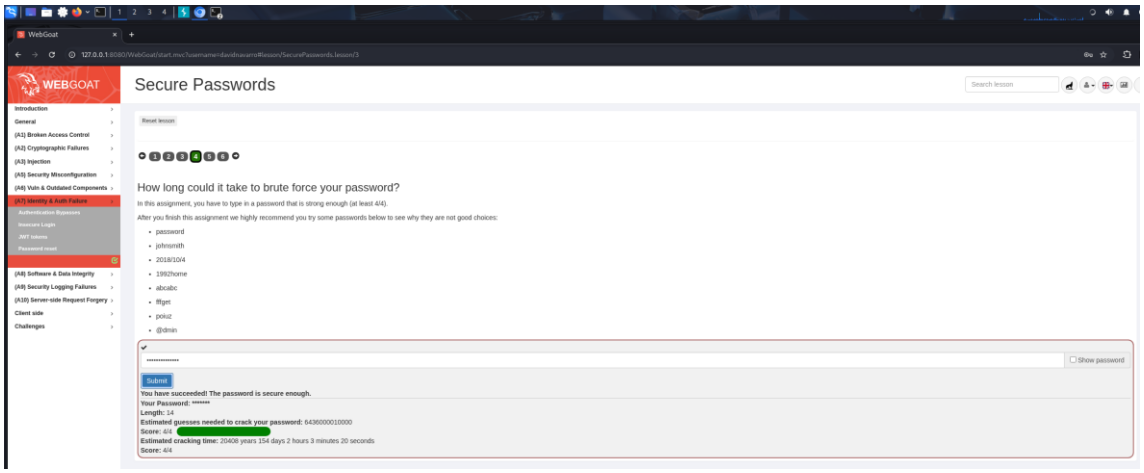
La contraseña utilizada fue:

meGust@el4nime

Tras enviarla, la aplicación realizó el análisis automático de fortaleza y devolvió los siguientes resultados:

- Longitud: **14 caracteres**
- Puntuación de seguridad: **4/4**
- Número estimado de intentos necesarios: **6.436.000.010.000**
- Tiempo estimado de crackeo: **más de 20.000 años**

La aplicación marcó el ejercicio como completado correctamente, indicando que la contraseña introducida es suficientemente robusta frente a ataques de fuerza bruta.



Resultado obtenido

El sistema confirma que una contraseña con suficiente longitud, combinando letras mayúsculas, minúsculas, números y caracteres especiales, incrementa de forma exponencial el coste computacional de un ataque de fuerza bruta.

Este comportamiento evidencia que el uso de contraseñas simples como *password*, *johnsmith* o fechas comunes, tal y como propone el propio ejercicio, supone un riesgo crítico para la seguridad de las cuentas de usuario.

Impacto

En un entorno real, el uso de contraseñas débiles permitiría a un atacante comprometer cuentas de usuario mediante ataques automatizados en un tiempo reducido, facilitando accesos no autorizados y escaladas posteriores.

La falta de políticas de contraseñas robustas compromete directamente la confidencialidad de la información y la seguridad de los sistemas de autenticación.

Recomendaciones

Para mitigar los riesgos asociados a contraseñas débiles y reducir la viabilidad de ataques de fuerza bruta o diccionario, es imprescindible aplicar una política de contraseñas robusta y bien definida.

En primer lugar, se recomienda exigir que las contraseñas cumplan con unos **requisitos mínimos de complejidad**, tales como:

- Una **longitud suficiente** (preferiblemente igual o superior a 12 caracteres).
- Inclusión de **letras mayúsculas y minúsculas**.
- Uso de **números**.
- Incorporación de **caracteres especiales**.

No obstante, cumplir únicamente estos requisitos básicos no siempre garantiza una contraseña realmente segura. En la práctica, muchos usuarios siguen patrones predecibles al construir contraseñas que cumplen formalmente con dichas reglas, como:

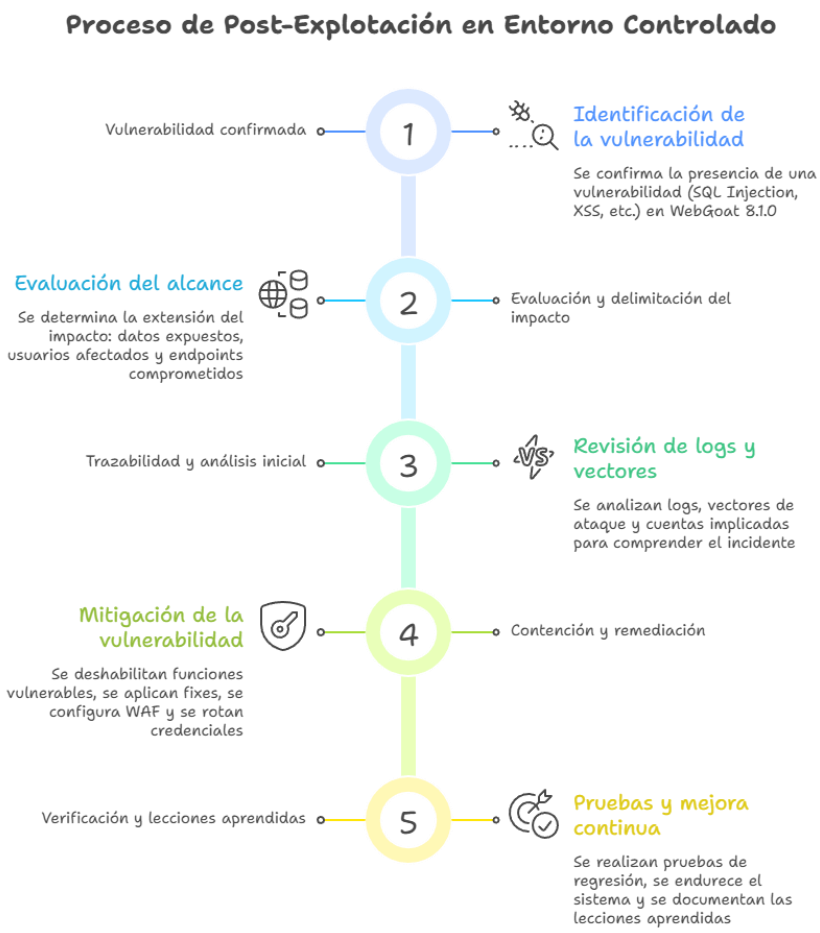
- Colocar la **única letra mayúscula al inicio** de la contraseña.
- Añadir los **números o caracteres especiales al final**.
- Basar la contraseña en palabras comunes o fácilmente reconocibles.

- Estos patrones son bien conocidos por los atacantes y suelen ser priorizados durante ataques automatizados, reduciendo de forma significativa el espacio efectivo de búsqueda y, por tanto, el tiempo necesario para comprometer una contraseña.

Por este motivo, se recomienda además:

- Promover contraseñas en las que mayúsculas, números y caracteres especiales estén **distribuidos de forma no predecible** a lo largo de toda la cadena.
- Evitar estructuras comunes como *Password123!* o *Empresa2026!*.
- Fomentar el uso de **frases de paso (passphrases)** modificadas de manera no trivial, que ofrezcan un buen equilibrio entre seguridad y facilidad de memorización.
- Complementar las contraseñas robustas con medidas adicionales como **autenticación multifactor (MFA)**, limitación de intentos y detección de comportamientos automatizados.

6. Post-explotación



Nota: Dado que WebGoat es un laboratorio deliberadamente vulnerable y controlado, la fase de post-explotación se documenta desde un enfoque **analítico y conceptual**, describiendo qué implicaciones tendría la explotación en un entorno real y cómo debería gestionarse de forma responsable.
No se han realizado acciones destructivas ni persistentes más allá de las necesarias para demostrar cada vulnerabilidad.

Objetivo de la fase de post-explotación

Una vez confirmada la existencia de las vulnerabilidades, la fase de post-explotación tiene como finalidad:

- Determinar el **alcance real del impacto**, identificando qué datos, funcionalidades o usuarios podrían verse afectados.
- Obtener la **evidencia técnica mínima necesaria**, evitando la sobre-extracción de información.
- Analizar qué **medidas de contención y remediación** serían necesarias para prevenir la explotación en un entorno productivo.
- Extraer **lecciones técnicas** aplicables a escenarios reales de seguridad ofensiva y defensiva.

Consideraciones de post-explotación en un entorno real

En un sistema productivo, tras la validación de vulnerabilidades similares a las identificadas en este laboratorio, deberían aplicarse las siguientes acciones:

- **Delimitación del impacto**
  Identificación precisa de los endpoints afectados, roles implicados y tipos de datos expuestos (personales, credenciales, financieros, etc.).
- **Trazabilidad y análisis forense inicial**
  Revisión de logs de aplicación y base de datos para detectar intentos de explotación, vectores utilizados y posibles cuentas comprometidas.
- **Contención inmediata**
  Deshabilitación temporal de funcionalidades vulnerables, aplicación de reglas de WAF o rate-limiting y rotación de credenciales cuando proceda.
- **Remediación estructural**
  Corrección del código vulnerable (consultas preparadas, validación de entradas, protección CSRF, actualización de dependencias) y ajustes de configuración.

- **Verificación y pruebas de regresión**
  Validación de que las correcciones aplicadas eliminan la vulnerabilidad sin introducir nuevos fallos o impactos funcionales.

Lecciones clave extraídas del laboratorio

Los ejercicios realizados permiten extraer conclusiones relevantes desde el punto de vista de la post-explotación:

- **SQL Injection**
  Una vez confirmada la inyección, el riesgo escala rápidamente desde un simple bypass lógico hasta la enumeración completa del esquema y la extracción masiva de información sensible, especialmente en ausencia de controles de privilegios y monitorización.
- **Componentes vulnerables**
  El laboratorio evidencia que una vulnerabilidad puede existir aunque el comportamiento "esperado" no se manifieste de forma inmediata (por ejemplo, un alert que no se dispara). La observación de errores en consola y la validación manual del vector demuestran que la superficie de ataque sigue presente, lo que refuerza la importancia de analizar dependencias y no confiar únicamente en efectos visibles.

7. Herramientas utilizadas

Durante la realización de la práctica se emplearon las siguientes herramientas y entornos, todos ellos en un **contexto de laboratorio controlado y con fines exclusivamente formativos**:

WebGoat 8.1.0

Laboratorio de entrenamiento diseñado para la práctica de vulnerabilidades incluidas en el **OWASP Top 10**, utilizado como aplicación objetivo durante toda la auditoría.

Permite reproducir escenarios reales de fallos de seguridad de forma controlada, facilitando la validación de vulnerabilidades, el análisis de impacto y la comprensión de medidas de mitigación.

Navegador web (DevTools)

Uso intensivo de las herramientas de desarrollo del navegador para la validación y análisis de los ejercicios, incluyendo:

- Inspección de la **consola JavaScript**.
- Identificación de **errores de ejecución** y comportamientos anómalos.
- Validación de **payloads XSS** y su ejecución en el contexto del navegador.
- Ejecución manual de pruebas controladas y análisis del comportamiento del lado cliente.

Nmap

Herramienta de escaneo y reconocimiento de red utilizada durante la fase de *Information Gathering* para identificar información técnica sobre el entorno de la aplicación WebGoat, sin realizar explotación activa.

Se empleó para:

- Identificar puertos abiertos expuestos por la aplicación.
- Detectar servicios activos y sus versiones asociadas.
- Obtener información sobre el sistema operativo subyacente.
- Correlacionar los resultados obtenidos con el despliegue del entorno en Docker sobre Kali Linux.

Las principales técnicas utilizadas incluyen escaneos de puertos específicos, detección de servicios y versiones, y análisis básico del sistema operativo, permitiendo construir una visión inicial del entorno antes de las fases de análisis y explotación.

Burp Suite

Herramienta de interceptación y análisis de tráfico HTTP/HTTPS utilizada para:

- Captura y análisis de **peticiones y respuestas HTTP**.
- Identificación de **parámetros controlables por el usuario**.
- Verificación de vulnerabilidades de inyección y manipulación de lógica.
- Reproducción y modificación manual de peticiones durante las pruebas.
- Obtención de **evidencias técnicas** para la documentación del informe.

Sqlmap

Herramienta de automatización de auditorías de **SQL Injection**, utilizada en un entorno de laboratorio con fines exclusivamente didácticos para:

- Confirmar automáticamente la existencia de vulnerabilidades de inyección SQL.
- Identificar el **motor de base de datos subyacente**.
- Enumerar esquemas, tablas y columnas accesibles.
- Realizar volcados controlados de información con el objetivo de demostrar el impacto real de la vulnerabilidad.

Su uso permitió evidenciar la **escalabilidad del riesgo**, mostrando cómo una inyección SQL inicialmente simple puede derivar en la enumeración y extracción masiva de datos sensibles.

Entorno de terminal Linux

Utilizado como soporte técnico para:

- Ejecución de herramientas de auditoría.
- Validación de resultados obtenidos.
- Análisis técnico de la información extraída durante los ejercicios de explotación y verificación.

Docker

Empleado para el despliegue de WebGoat 8.1.0 en un contenedor aislado, garantizando:

- Reproducibilidad del entorno de pruebas.
- Aislamiento respecto al sistema anfitrión.
- Ejecución consistente del laboratorio independientemente del host utilizado.

Máquina virtual Kali Linux

Entorno de trabajo dedicado a las pruebas de seguridad, proporcionando:

- Sistema operativo especializado en auditoría y pentesting.
- Herramientas preinstaladas orientadas a seguridad ofensiva.
- Separación clara entre el entorno de pruebas y el sistema personal del usuario.

8. Conclusión final

La realización de esta práctica demuestra que muchas de las vulnerabilidades más críticas en aplicaciones web no requieren técnicas avanzadas o exploits complejos, sino que surgen de fallos sistemáticos en aspectos básicos del desarrollo seguro, como la validación de entradas, el diseño de los mecanismos de autenticación y la gestión de dependencias externas.

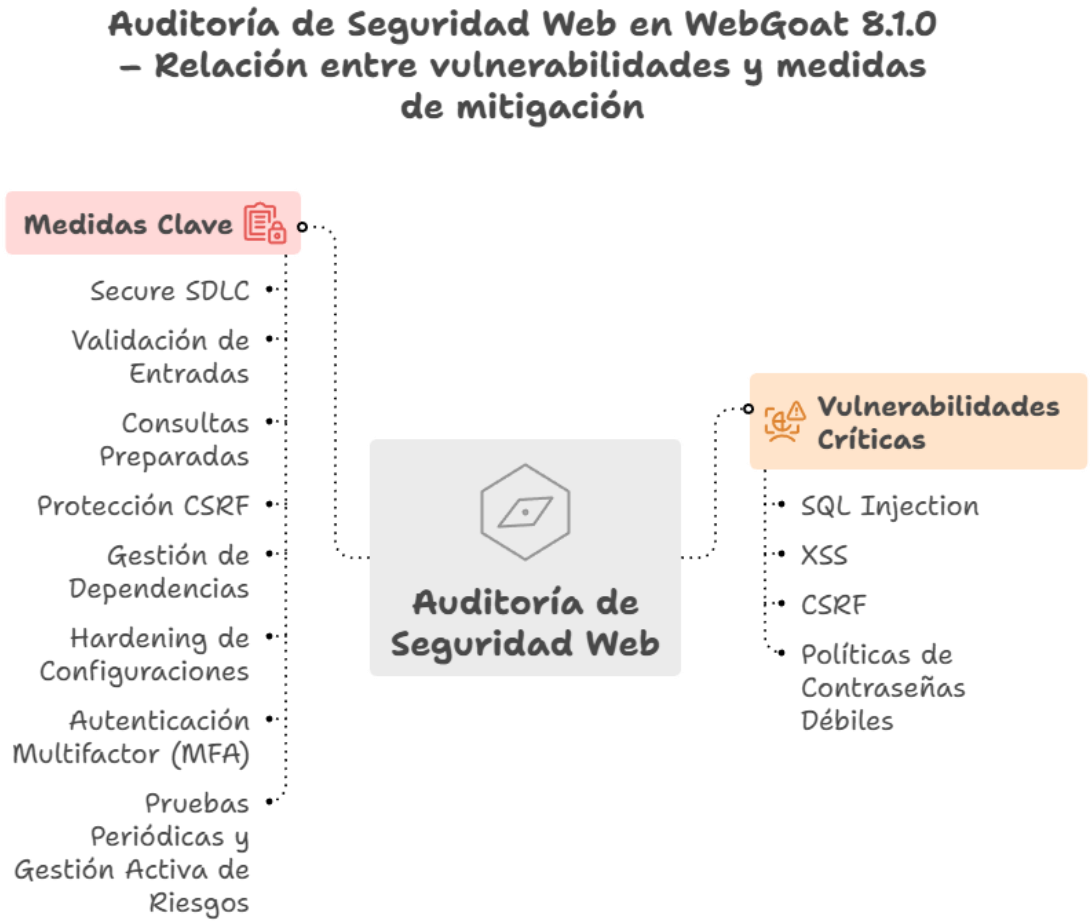
Los ejercicios de **Injection (SQL Injection)** evidencian que este tipo de vulnerabilidad representa uno de los mayores riesgos de seguridad, ya que permite escalar rápidamente desde un simple bypass lógico hasta la enumeración completa de la base de datos y la extracción masiva de información sensible, comprometiendo gravemente la confidencialidad e integridad de los datos.

Las vulnerabilidades de **Cross-Site Scripting (XSS)** y de **componentes vulnerables o desactualizados** refuerzan la idea de que el navegador es un vector crítico de ataque. La ejecución de código JavaScript en el contexto de un usuario legítimo abre la puerta al robo de sesiones, la manipulación de acciones y la explotación de funcionalidades legítimas de la aplicación, incluso cuando el código propio no parece directamente vulnerable.

El ejercicio de **CSRF** demuestra que no basta con que una acción sea “válida” desde el punto de vista funcional: si no se valida adecuadamente el contexto de origen y la intención del usuario, es posible forzar acciones en nombre de un usuario autenticado sin su conocimiento.

Por último, el análisis de **políticas de contraseñas** confirma que cumplir reglas formales de complejidad no garantiza por sí mismo una autenticación segura. La seguridad moderna exige un enfoque en capas que incluya contraseñas robustas y no predecibles, limitación de intentos, detección de comportamientos anómalos y mecanismos adicionales como la autenticación multifactor (MFA).

En conjunto, este laboratorio refuerza una conclusión operativa clara: la seguridad efectiva no depende de un único parche o control aislado, sino de una disciplina continua de **desarrollo seguro (SDLC)**, pruebas periódicas, hardening de configuraciones y una gestión activa de dependencias y riesgos. La integración temprana de la seguridad en el ciclo de vida del software es clave para reducir de forma sostenida la superficie de ataque.



Relación conceptual entre las vulnerabilidades críticas identificadas en WebGoat 8.1.0 y las principales medidas de seguridad necesarias para lograr una defensa web efectiva.