Informe de Auditoría de Seguridad Web: WebGoat

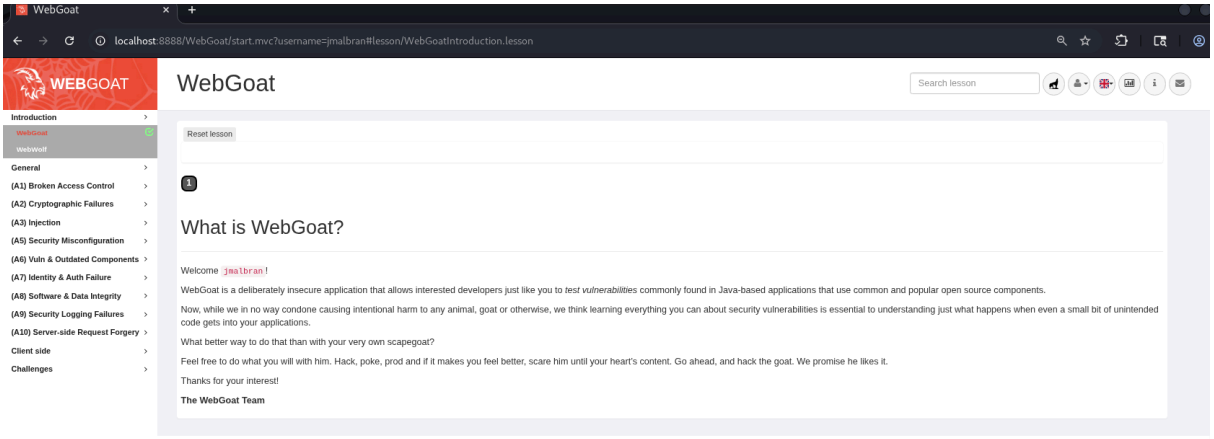
Contenido:

1) Ámbito y alcance de la auditoría

El presente informe documenta una auditoría de seguridad básica realizada sobre la aplicación web vulnerable **WebGoat versión 8.1.0**, desplegada en un entorno local con fines educativos.

El objetivo de la auditoría fue identificar y explotar vulnerabilidades comunes en aplicaciones web, alineadas con el **OWASP Top 10**, utilizando técnicas básicas de reconocimiento y explotación.

El alcance de la auditoría se limitó exclusivamente a la aplicación WebGoat accesible a través de la URL http://loaclhost:8888/WebGoat, sin realizar pruebas sobre otros sistemas, redes o infraestructuras externas.



2.1) Resumen del proceso

Durante la realización de la presente auditoría de seguridad se llevaron a cabo tareas de reconocimiento e identificación de vulnerabilidades sobre la aplicación WebGoat, utilizando un enfoque práctico y guiado.

El proceso incluyó la recopilación de información básica del entorno, así como la explotación de vulnerabilidades previamente definidas dentro de la aplicación, relacionadas con inyección SQL, Cross Site Scripting (XSS), fallos de configuración de seguridad, uso de componentes vulnerables y fallos de autenticación.

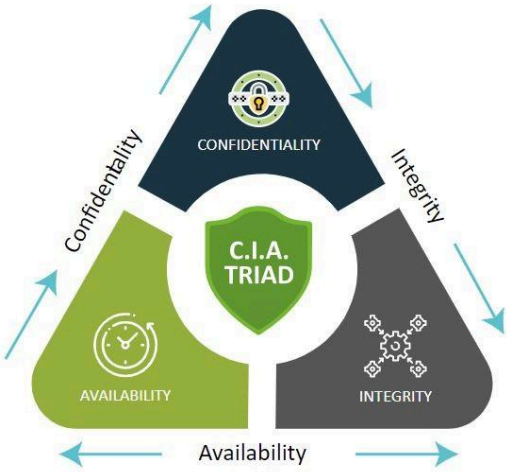
Todas las pruebas fueron realizadas en un entorno controlado, local y con fines exclusivamente educativos.

2.2) Vulnerabilidades destacadas

Durante la auditoría de seguridad se identificaron y explotaron diversas vulnerabilidades comunes en aplicaciones web, entre las que destacan:

- Inyección SQL (SQL Injection)
- Cross Site Scripting (XSS)
- Fallos de configuración de seguridad
- Uso de componentes vulnerables u obsoletos
- Fallos en los mecanismos de autenticación y gestión de contraseñas

Estas vulnerabilidades permiten a un atacante comprometer la confidencialidad, integridad y disponibilidad de la información gestionada por la aplicación.



2.3) Conclusiones

A partir de las pruebas realizadas, se concluye que la aplicación WebGoat presenta múltiples vulnerabilidades de seguridad comunes en aplicaciones web, principalmente relacionadas con una validación inadecuada de las entradas de usuario, configuraciones inseguras y mecanismos de autenticación débiles.

Si bien la aplicación tiene un propósito educativo, los resultados obtenidos reflejan escenarios reales que podrían ser explotados en entornos productivos si no se aplican las medidas de seguridad adecuadas.

2.4) Recomendaciones

Se recomienda implementar mecanismos de validación y sanitización de entradas en todas las interacciones con el usuario, aplicar configuraciones de seguridad adecuadas en el servidor y la aplicación, y mantener actualizados todos los componentes utilizados.

Asimismo, se sugiere reforzar los controles de autenticación y gestión de contraseñas, así como realizar auditorías de seguridad periódicas para identificar y mitigar posibles vulnerabilidades de forma temprana.

3) Proceso de Auditoría de la aplicación WebGoat

En esta sección se describe el proceso de auditoría de seguridad llevado a cabo sobre la aplicación WebGoat, detallando las distintas fases realizadas durante la evaluación.

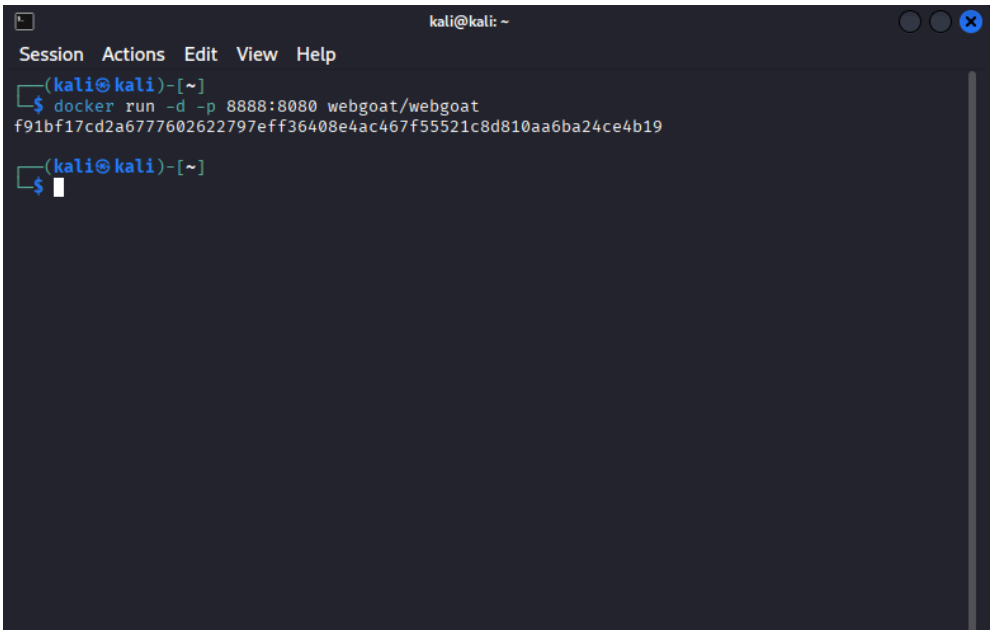
El proceso se estructuró en etapas de reconocimiento, explotación de vulnerabilidades, análisis posterior a la explotación y propuestas de mitigación, siguiendo una metodología básica orientada a la identificación de vulnerabilidades comunes en aplicaciones web.

3.1) Reconocimiento / Information gathering

En esta fase se realizó el análisis técnico previo a la explotación, centrado en el despliegue de la infraestructura y la identificación de servicios activos para definir la superficie de ataque

A) Despliegue de infraestructura mediante Docker

Para el análisis de seguridad, se procedió al despliegue del activo utilizando tecnología de contenedores **(Docker)**. Esto permitió ejecutar la aplicación WebGoat en un entorno aislado, asegurando que los servicios estuvieran correctamente mapeados para el escaneo.

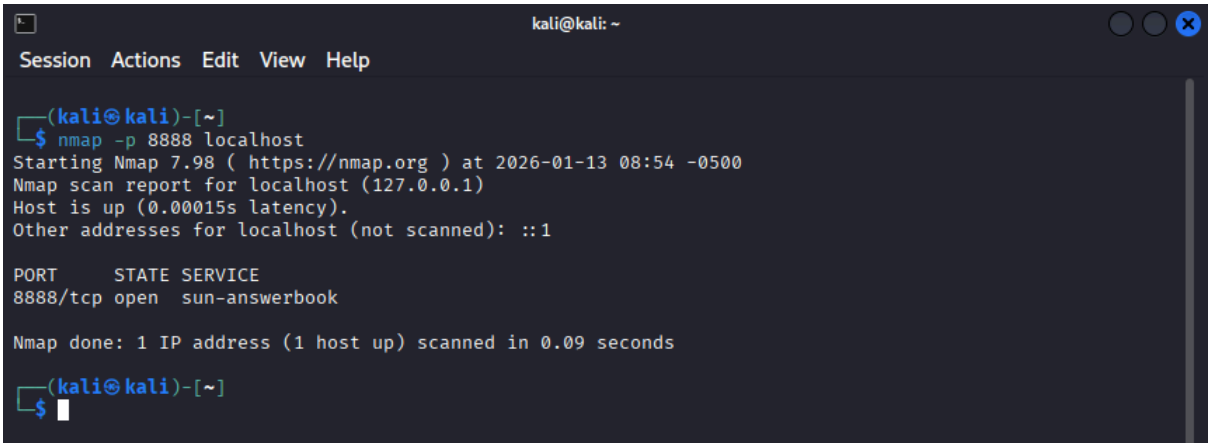


B) Escaneo de servicios y detección de puertos con Nmap

Una vez establecido el entorno, se utilizó la herramienta Nmap para realizar un reconocimiento de red sobre el activo. Este paso permitió validar la disponibilidad de los servicios y confirmar los puntos de entrada expuestos.

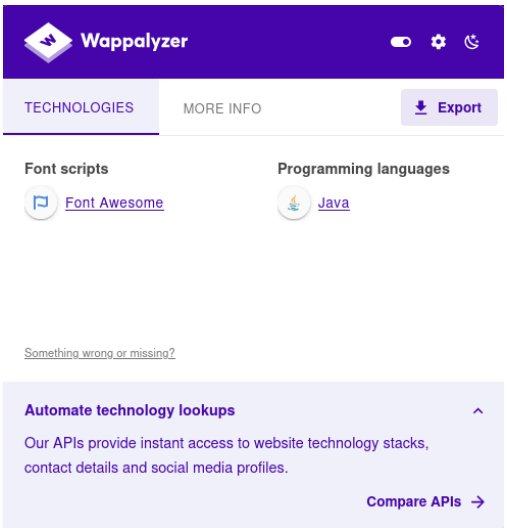
Comando en Terminal:

`nmap -p 8888 localhost`



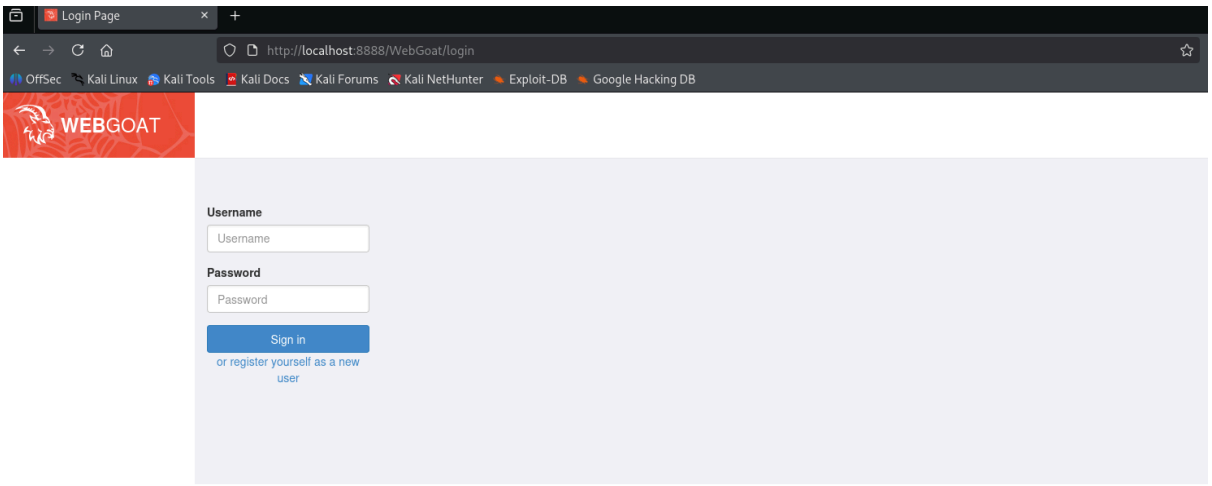
C. Identificación de Tecnologías

Asimismo, mediante Wappalyzer se identificó que la aplicación WebGoat está desarrollada en **Java** sobre un servidor de aplicaciones Java, coincidiendo con los headers HTTP observados. Esta información es relevante para contextualizar las vulnerabilidades y las técnicas de explotación.



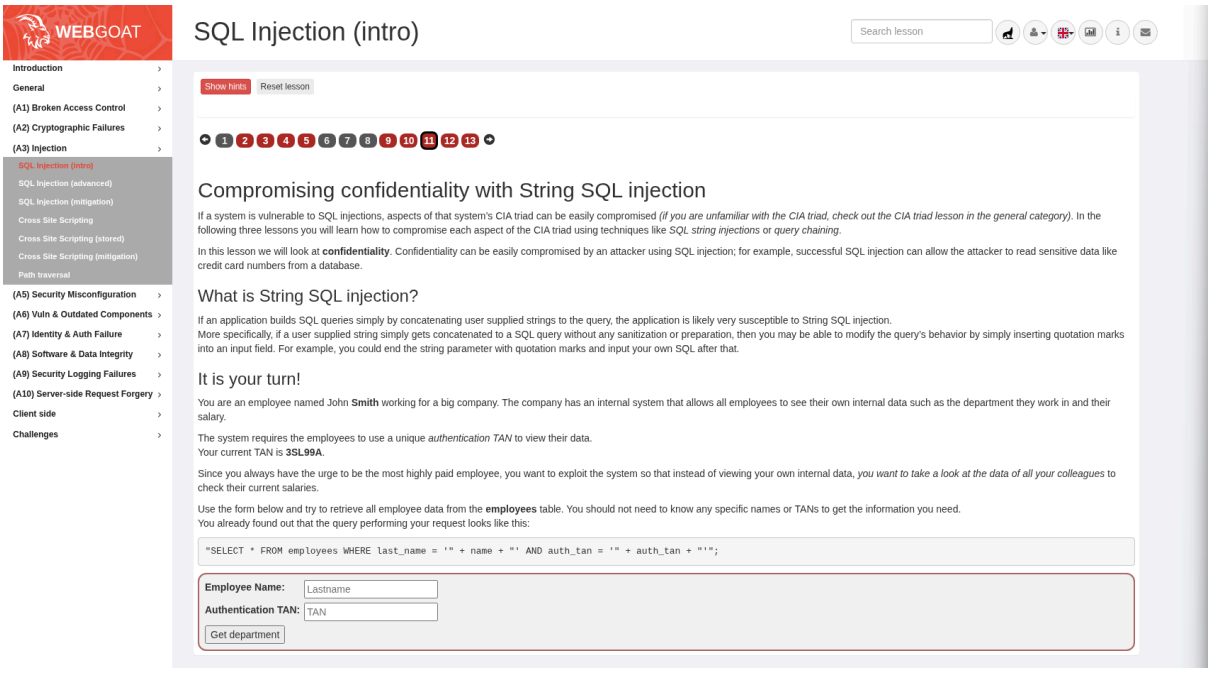
D) Acceso a la aplicación WebGoat

Se abrió la aplicación en el navegador mediante la URL **http://localhost:8888/WebGoat**, verificando su correcto funcionamiento y disponibilidad.

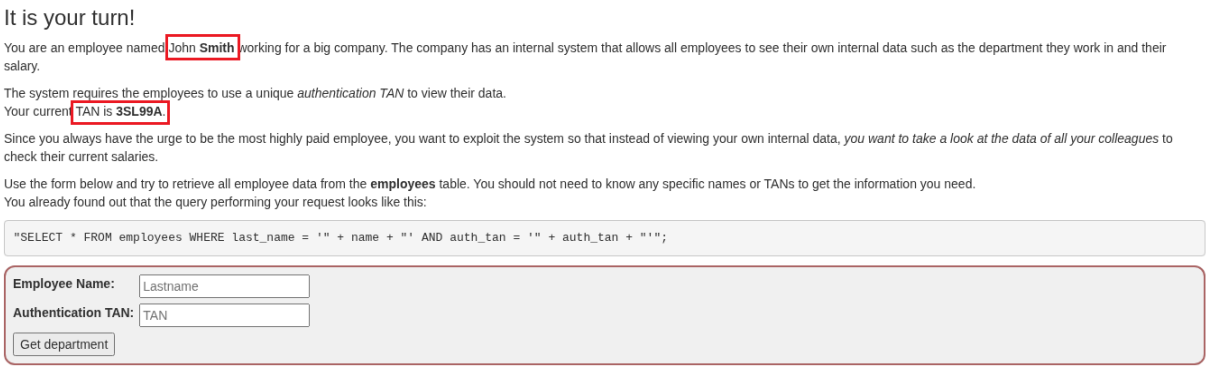


3.2) Explotación de vulnerabilidades detectadas

A3-SQL Injection (Intro)



En la siguiente imagen se muestra el estado inicial del ejercicio “Compromising confidentiality with String SQL injection”, previo a la explotación de la vulnerabilidad, donde el sistema solicita el apellido del empleado y su TAN de autenticación.



Al introducir un apellido válido junto con el TAN de autenticación proporcionado por el propio ejercicio, la aplicación devuelve correctamente la información asociada al empleado, estableciendo así el comportamiento legítimo del sistema previo a la explotación

A) Análisis de la vulnerabilidad

La funcionalidad analizada construye dinámicamente una consulta SQL utilizando los valores introducidos por el usuario sin aplicar mecanismos de validación ni sanitización de entradas.

La consulta SQL o Query utilizada por la aplicación es la siguiente:

```
SELECT * FROM employees

WHERE last_name = '<Employee Name>'

AND auth_tan = '<Authentication TAN>';
```

Este comportamiento permite a un atacante modificar la lógica de la consulta mediante la inserción de caracteres especiales, como comillas simples, dando lugar a una vulnerabilidad de tipo SQL Injection basada en cadenas (String SQL Injection).

B) Explotación de la vulnerabilidad

A partir del análisis de la consulta SQL utilizada por la aplicación, se identifican múltiples puntos de entrada controlados por el usuario que permiten la explotación de una vulnerabilidad de tipo **SQL Injection**.

La consulta SQL empleada por la aplicación es la siguiente:

```
SELECT * FROM employees WHERE last_name = '<Employee Name>' AND auth_tan = '<Authentication TAN>';
```

Los parámetros last_name y auth_tan reciben directamente valores introducidos por el usuario y son concatenados a la consulta SQL sin ningún tipo de validación, escape de caracteres ni uso de consultas preparadas.

En particular, ambas variables se encuentran delimitadas por comillas simples ('), lo que permite al atacante cerrar la cadena original e inyectar condiciones SQL adicionales que modifican la lógica de la consulta.

¿En cuál de los campos puede realizarse la inyección?

Dado que ambos campos, **Employee Name** y **Authentication TAN**, forman parte de la cláusula **WHERE** y son tratados como cadenas de texto, cualquiera de ellos podría ser utilizado como punto de inyección SQL.

Para esta explotación se opta por inyectar código SQL en el campo **Employee Name**, mientras que en el campo **Authentication TAN** se introduce el valor legítimo proporcionado por la aplicación (**3SL99A**), aunque este último en este caso particular no era necesario de ingresar.

Esto demuestra que en algunos casos no es necesario manipular ambos parámetros, ya que la inyección en un único punto es suficiente para alterar la lógica de la consulta y comprometer la confidencialidad de la información.

Payload utilizado y explotación

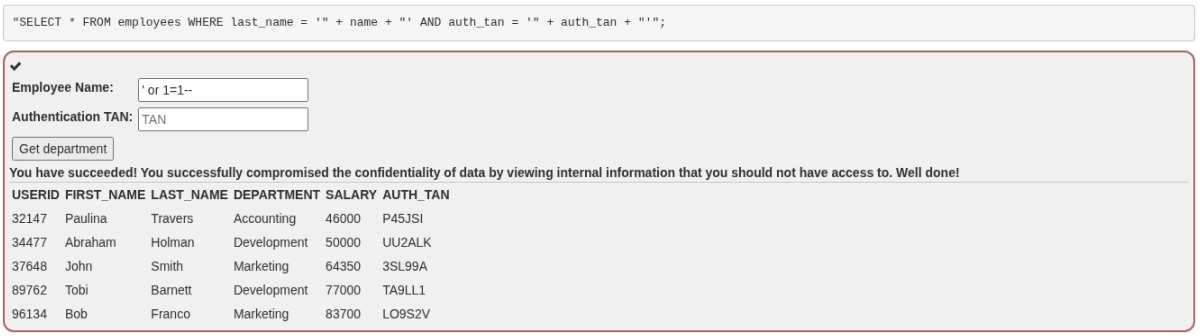
Con el objetivo de forzar una condición lógica siempre verdadera, se introduce el siguiente payload en el campo **Employee Name**:

OR '1' = '1'--

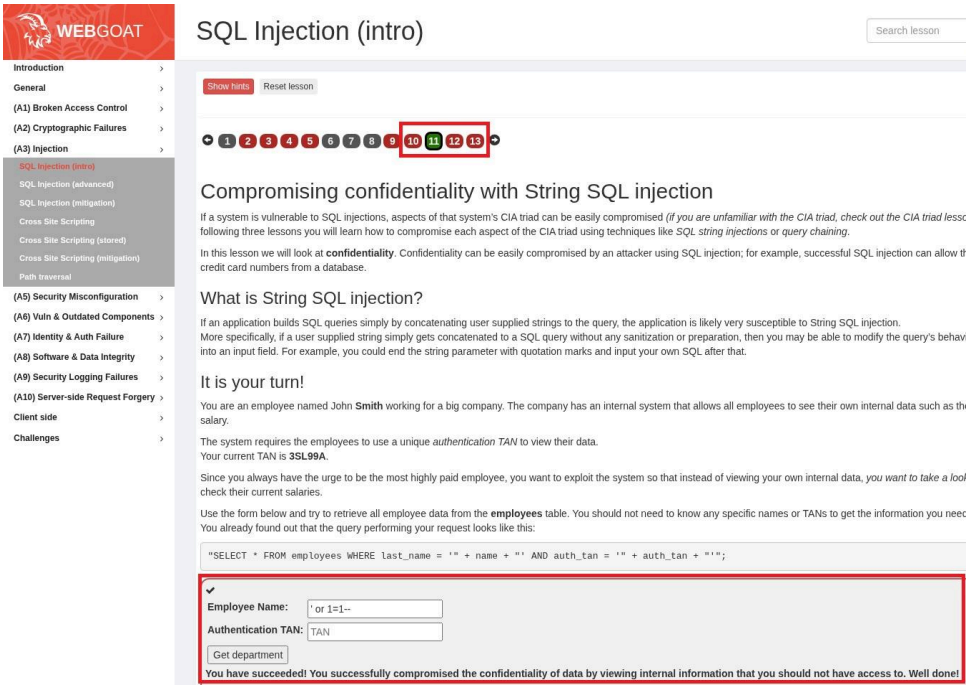
Este payload cierra la cadena original y añade una condición booleana que siempre se evalúa como verdadera, anulando los filtros previstos por la aplicación. El uso del comentario **(--)** permite ignorar el resto de la consulta SQL original.

```
SELECT * FROM employees

WHERE last_name = '' OR '1'='1'--'

AND auth_tan = '3SL99A';
```

Como consecuencia, el sistema devuelve la totalidad de los registros de la tabla **employees**, permitiendo el acceso no autorizado a información sensible de todos los empleados.



Finalmente, la plataforma WebGoat confirma la correcta explotación de la vulnerabilidad mediante la finalización satisfactoria del ejercicio.



C) <u>Obtención de información adicional mediante SQL Injection</u>

Tras confirmar la vulnerabilidad, se procedió a realizar una fase de post-explotación para determinar el alcance real del compromiso sobre la base de datos. El proceso se estructuró en las siguientes etapas:

1- Validación de la estructura de la consulta

Como paso previo a la exfiltración, fue necesario determinar el número exacto de columnas que devuelve la consulta original para asegurar la compatibilidad del operador UNION. Se utilizó una técnica de prueba y error mediante valores NULL, confirmando que el sistema maneja **6 columnas**.

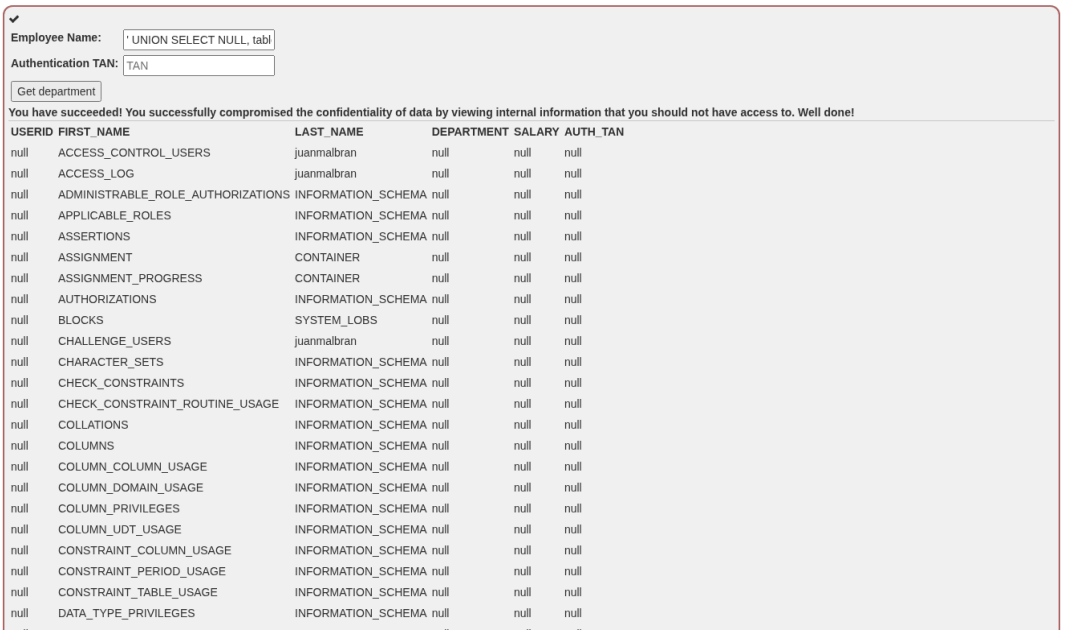
Payload: ' UNION SELECT NULL, NULL, NULL, NULL, NULL, NULL FROM user_system_data–

2- Descubrimiento de tablas y columnas (Reconocimiento interno)

Sin conocimiento previo del esquema, se consultó el diccionario de datos del sistema (**INFORMATION_SCHEMA**) para identificar las tablas disponibles y la ubicación exacta de información sensible como los correos electrónicos.

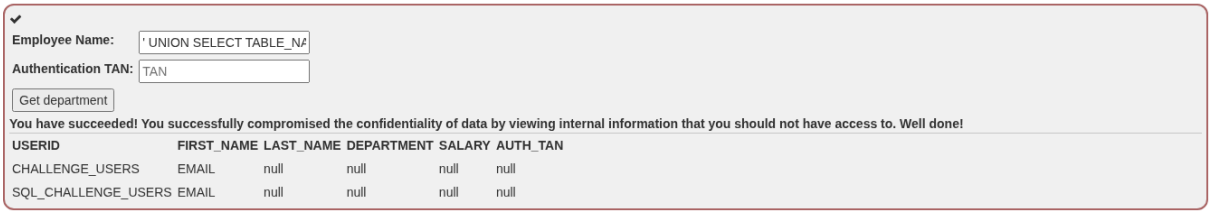
Identificación de tablas: Se listaron todas las tablas del esquema para localizar posibles objetivos.

Payload: ' UNION SELECT NULL, table_name, table_schema, NULL, NULL, NULL FROM information_schema.tables–



Localización de campos de email: Al no encontrar direcciones de correo en la tabla de empleados, se realizó una búsqueda por patrones en el catálogo de columnas.

Payload: ' UNION SELECT TABLE_NAME, COLUMN_NAME, NULL, NULL, NULL, NULL FROM INFORMATION_SCHEMA.COLUMNS WHERE COLUMN_NAME LIKE '%MAIL%'--

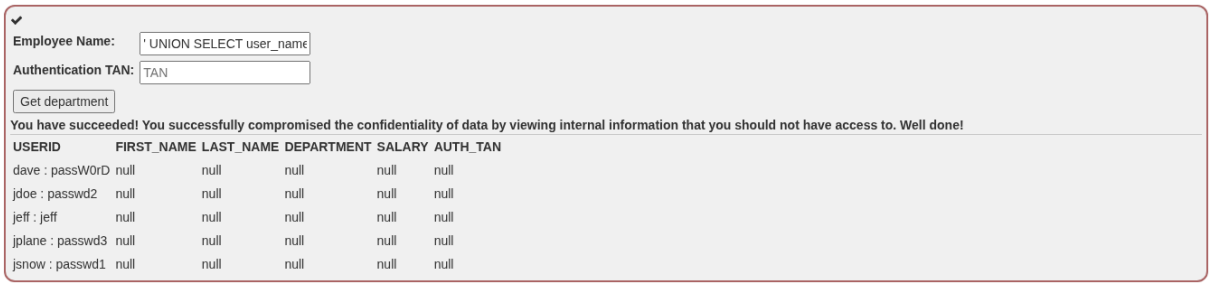


3. Exfiltración selectiva de datos

Una vez mapeada la estructura, se ejecutaron los payloads finales para extraer la información. Se empleó el operador de concatenación **||** para agrupar múltiples campos y facilitar su lectura en la interfaz.

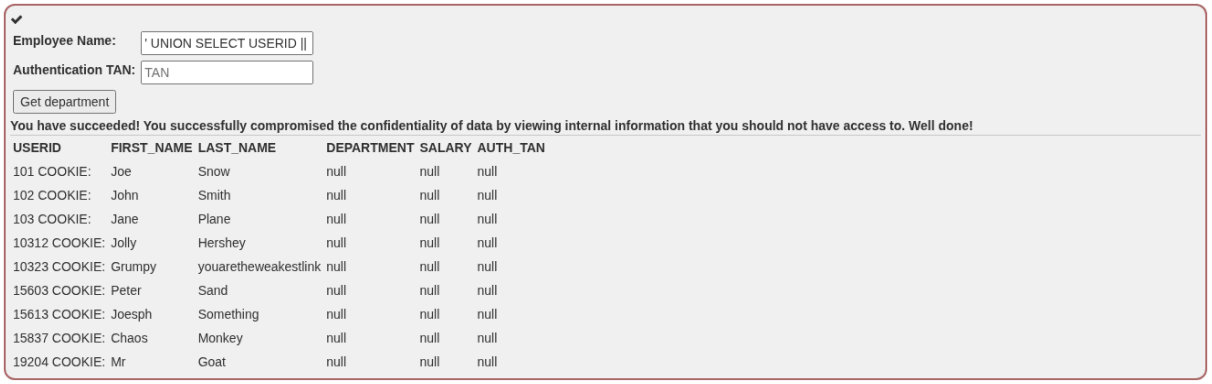
Extracción de credenciales de acceso: Se recuperaron los registros de la tabla **USER_SYSTEM_DATA**, obteniendo pares de usuario y contraseña.

> **Payload: ' UNION SELECT user_name || ' : ' || password, NULL, NULL, NULL, NULL, NULL FROM user_system_data–**



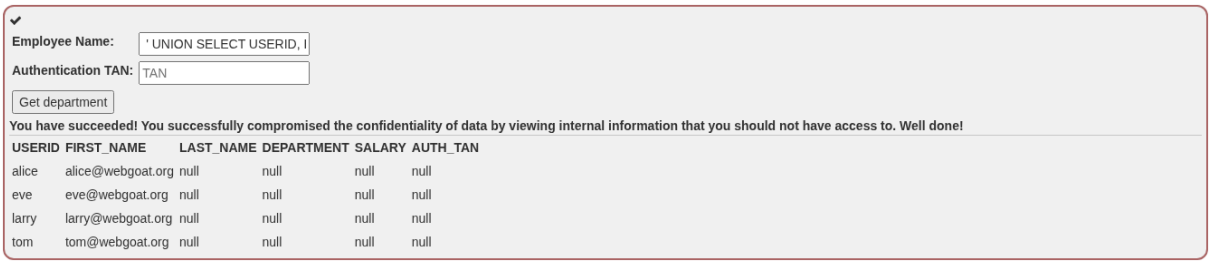
Extracción de tokens de sesión (Cookies): Se identificó y extrajo el campo **COOKIE** de la tabla **USER_DATA**, revelando tokens como **youaretheweakestlink** del usuario **Grumpy**. Este hallazgo confirma la viabilidad de ataques de secuestro de sesión (*Session Hijacking*).

> **Payload: ' UNION SELECT USERID || ' COOKIE: ' || COOKIE, FIRST_NAME, LAST_NAME, NULL, NULL, NULL FROM USER_DATA–**



Extracción de direcciones de correo: Con la ruta identificada en el paso anterior, se procedió a extraer los correos electrónicos de la tabla **CHALLENGE_USERS**.

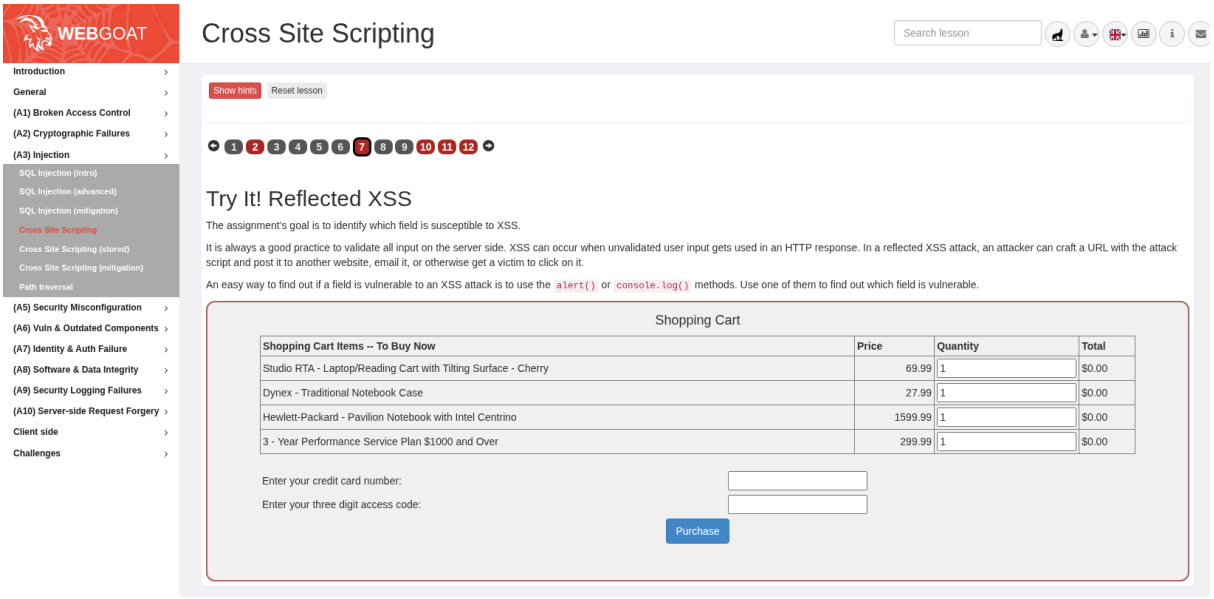
> **Payload: ' UNION SELECT USERID, EMAIL, NULL, NULL, NULL, NULL FROM CHALLENGE_USERS–**



En definitiva, los hallazgos presentados en este informe subrayan la necesidad de adoptar un enfoque de 'seguridad por diseño', donde la protección de los datos y la validación de las interacciones del usuario sean prioridades fundamentales desde las primeras etapas del desarrollo

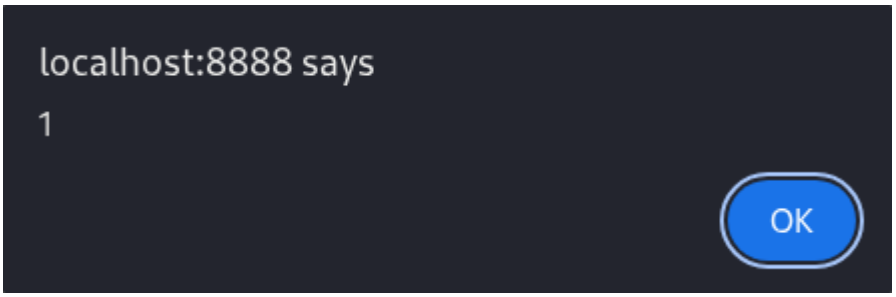
A3- Cross-Site Scripting Reflejado (XSS)

En el ejercicio ***A3 Injection – Cross Site Scripting – Apartado 7***, se analizó un escenario de XSS reflejado cuyo objetivo era identificar qué campo de entrada resultaba vulnerable a la inyección de código JavaScript.

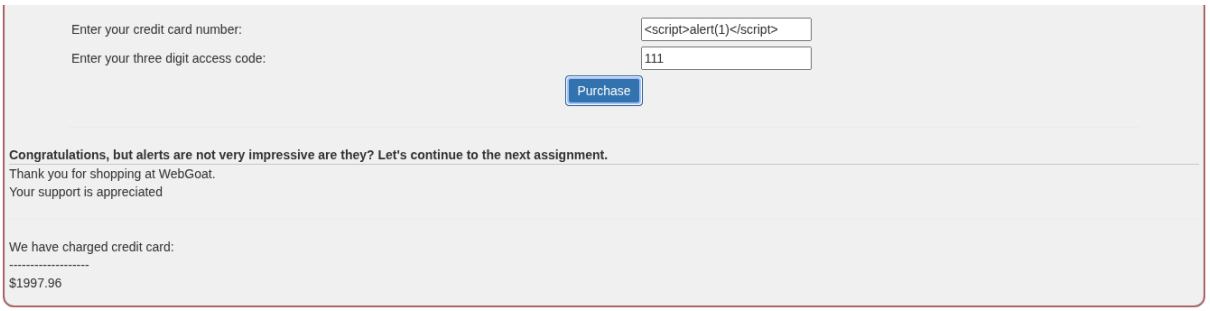


Como primer paso, se procedió a realizar pruebas básicas de inyección utilizando payloads de comprobación, tales como:

<script>alert(1)</script>



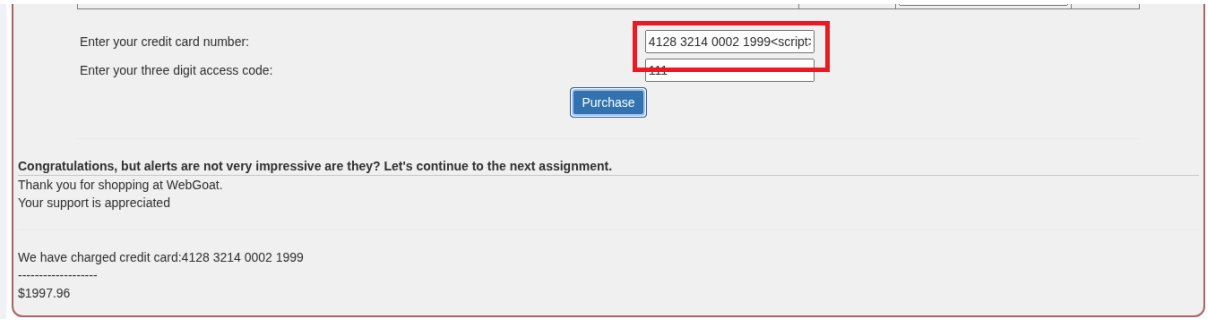
El uso de funciones como **alert()** o **console.log()** constituye una técnica estándar para verificar la presencia de vulnerabilidades XSS, ya que permite confirmar de forma inmediata si el código introducido por el usuario es reflejado en la respuesta HTTP y ejecutado por el navegador sin ser sanitizado.



Durante las pruebas se observó que el campo analizado no aplicaba ningún tipo de validación o filtrado previo sobre el contenido introducido, permitiendo la ejecución del código JavaScript incluso cuando el payload era introducido sin cumplir ningún formato específico (por ejemplo, sin introducir un número de tarjeta válido).

Adicionalmente, se comprobó que el payload también se ejecutaba correctamente cuando se anteponía un valor numérico al código JavaScript, como por ejemplo:

1<script>alert(1)</script>



Este comportamiento confirma que el campo carece tanto de validaciones de formato como de mecanismos de sanitización del lado del servidor, lo que facilita la explotación de una vulnerabilidad de tipo XSS reflejado.

Cabe destacar que, en aplicaciones reales, este tipo de campos suele implementar validaciones adicionales (como longitud o formato numérico), lo que obligaría a un atacante a cumplir dichas restricciones antes de poder inyectar el payload malicioso. Sin embargo, en este caso concreto, la ausencia de estas medidas permitió confirmar la vulnerabilidad de forma directa.

La ejecución satisfactoria del payload demuestra que el campo es susceptible a ataques XSS reflejados, lo que podría permitir a un atacante ejecutar código arbitrario en el navegador de la víctima, comprometiendo la confidencialidad de la sesión y la información del usuario.

La explotación realizada cumple con los objetivos del ejercicio, demostrando de forma práctica la existencia de una vulnerabilidad XSS reflejada mediante la ejecución de código JavaScript arbitrario en el navegador del usuario.

A5 – Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) es una vulnerabilidad que permite a un atacante forzar a un usuario autenticado a ejecutar acciones no deseadas dentro de una aplicación web. El ataque se produce cuando la aplicación confía únicamente en la sesión del usuario (cookies) y no valida el origen ni la intención de la solicitud.

En un ataque CSRF, el navegador de la víctima envía una petición válida al servidor utilizando su sesión activa, pero dicha petición es originada desde un contexto externo controlado por el atacante.

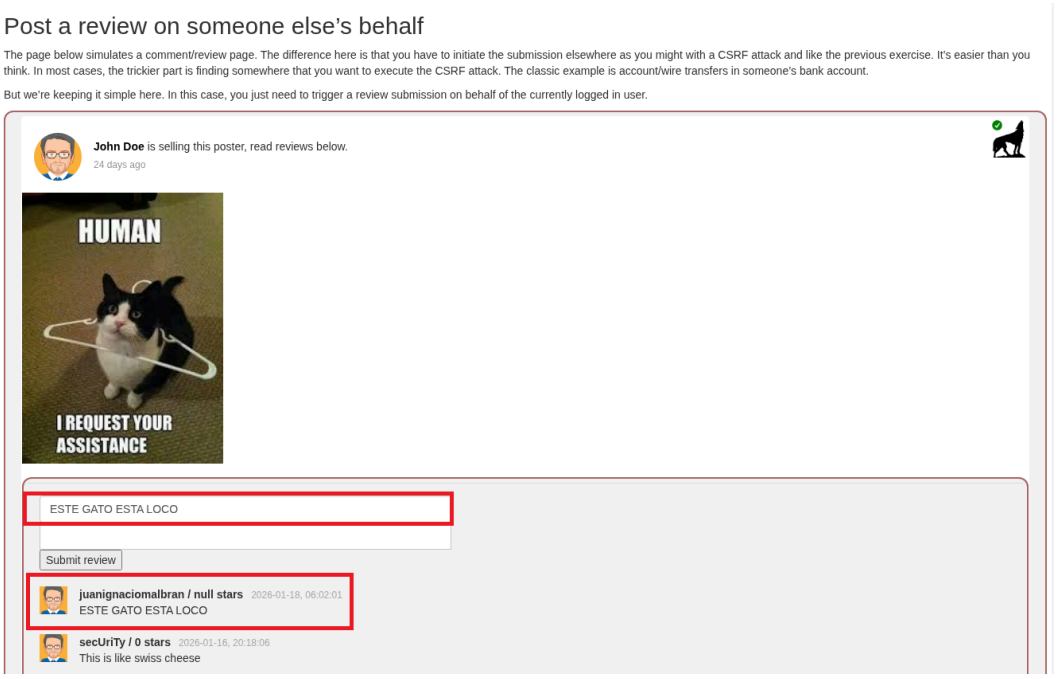
Escenario del ejercicio

En el ejercicio *“Post a review on someone else 's behalf”*, la aplicación simula una página de publicación de reseñas sobre un producto. El objetivo del ejercicio es demostrar que es posible publicar una review en nombre del usuario autenticado sin utilizar el formulario legítimo de la aplicación.

La aplicación no implementa mecanismos de protección contra CSRF, como tokens anti-CSRF o validación del origen de la solicitud.

Metodología utilizada

1. Se publicó inicialmente una reseña de forma legítima para observar el funcionamiento de la aplicación.



2. Se interceptó la solicitud HTTP generada para identificar:
    - el endpoint vulnerable (`/WebGoat/csrf/review`)
    - el método HTTP utilizado (POST)
    - los parámetros enviados (texto de la reseña y puntuación).

3. Se comprobó que la aplicación aceptaba la solicitud basándose únicamente en la sesión activa del usuario.

4. Posteriormente, se reprodujo la misma solicitud desde un contexto externo a la aplicación, simulando un ataque CSRF.

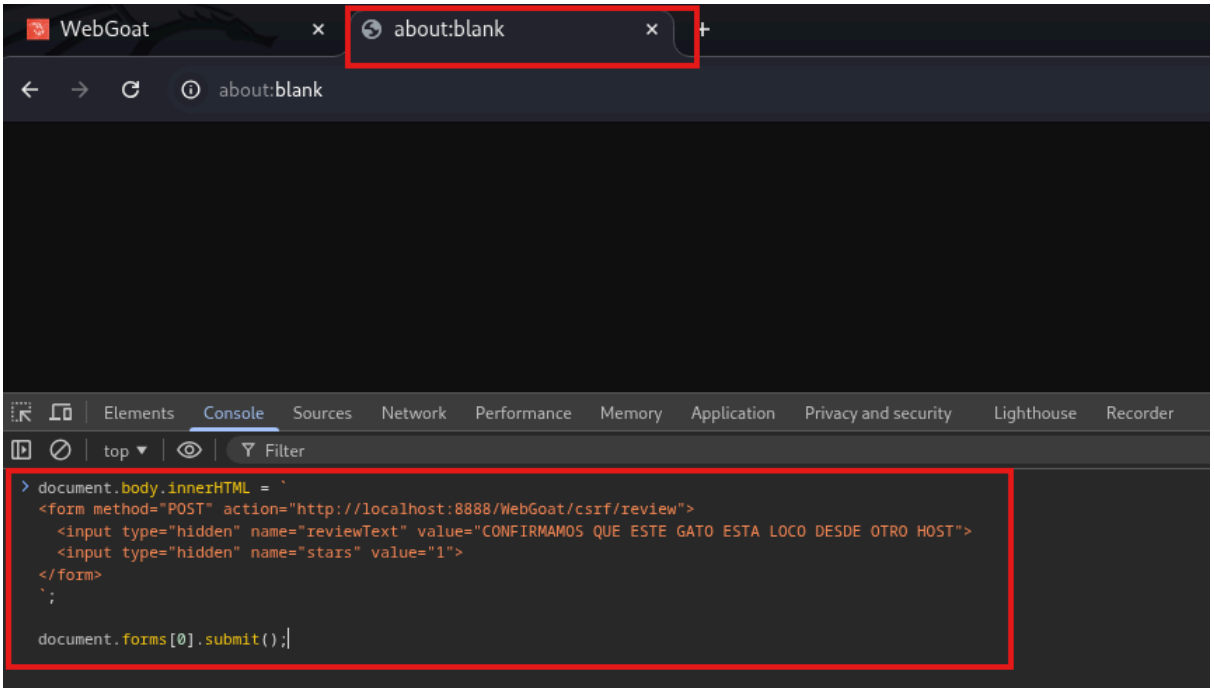
Código utilizado en el ataque

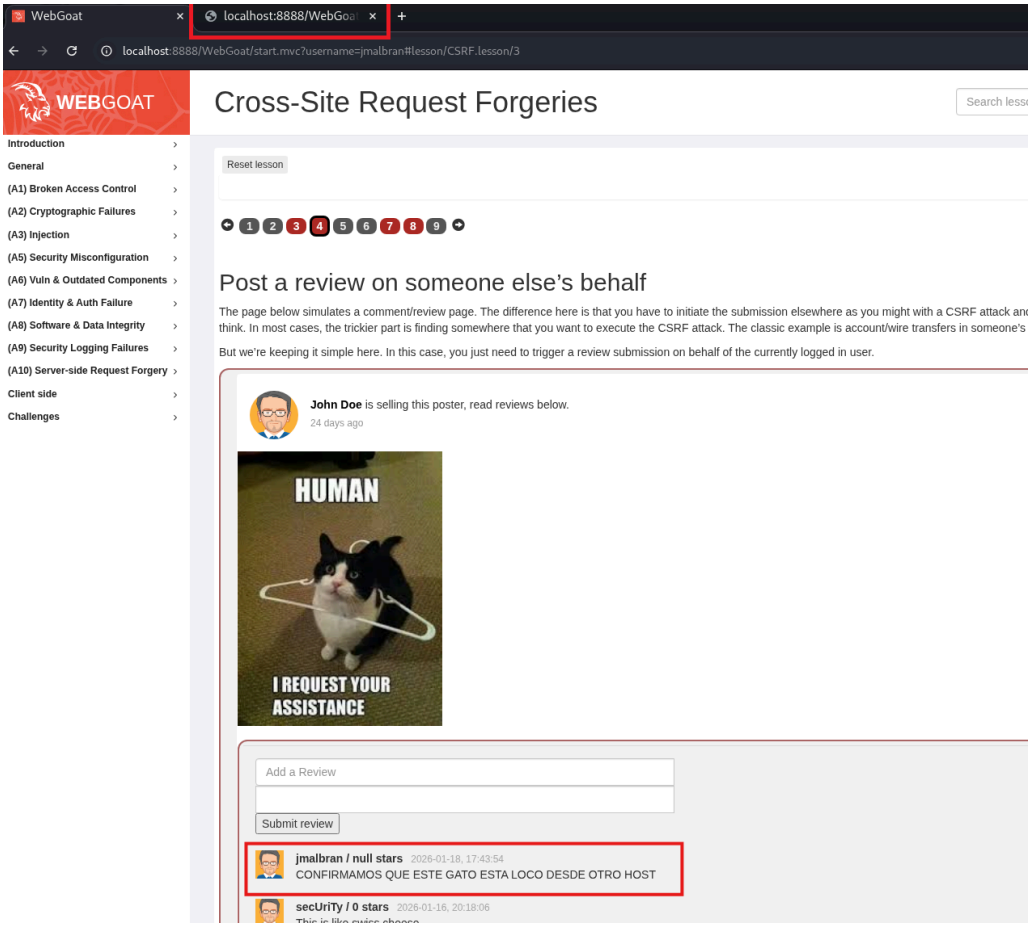
Para simular el envío de la solicitud desde un origen externo, se utilizó una nueva pestaña del navegador (**`about:blank`**) y la consola de desarrollador.
Dado que la consola del navegador solo acepta código JavaScript y no HTML directo, se utilizó JavaScript únicamente como medio para inyectar y enviar automáticamente un formulario HTML.

El código utilizado fue el siguiente:

```
document.body.innerHTML = `

<form method="POST" action="http://localhost:8888/WebGoat/csrf/review">

  <input type="hidden" name="reviewText" value="CSRF REVIEW DESDE OTRO ORIGEN">

  <input type="hidden" name="stars" value="1">

</form>

`;
```

```
document.forms[0].submit();
```





Este código:

- crea dinámicamente un formulario HTML externo,
- envía la solicitud automáticamente sin interacción del usuario,
- Utiliza la sesión activa del navegador para autenticar la petición.

Uso de JavaScript

El uso de JavaScript en este ejercicio no es un requisito del ataque CSRF en sí, sino una limitación del entorno de pruebas (WebGoat).
La consola del navegador solo permite la ejecución de JavaScript, por lo que fue necesario utilizarlo para inyectar y enviar el formulario HTML.

Ejemplo de ataque CSRF utilizando solo HTML (escenario real)

En un escenario real, el ataque podría realizarse sin JavaScript mediante un simple formulario HTML alojado en una página externa, por ejemplo:

```
<form method="POST" action="http://victima.com/csrf/review">
  <input type="hidden" name="reviewText" value="CSRF SIMPLE">
  <input type="hidden" name="stars" value="1">
  <input type="submit" value="Enviar">
</form>
```

Resultado

La reseña fue publicada correctamente en nombre del usuario autenticado sin utilizar el formulario legítimo de la aplicación.
La acción fue ejecutada desde un origen externo y aceptada por el servidor sin validación adicional, confirmando la existencia de la vulnerabilidad.

Impacto

Un atacante podría explotar esta vulnerabilidad para:

- publicar contenido no autorizado,
- modificar información del usuario,
- ejecutar acciones sensibles sin el consentimiento de la víctima.

Medidas de mitigación

Para prevenir ataques CSRF se recomienda:

- implementar tokens CSRF únicos por sesión o petición,
- validar los encabezados **Origin** y **Referer**,
- utilizar cookies con atributo **SameSite**,
- Requerir autenticación para acciones críticas.

El ejercicio demuestra una vulnerabilidad CSRF clásica, donde una acción sensible puede ser ejecutada desde un origen externo aprovechando la sesión activa del usuario. La ausencia de mecanismos de protección adecuados permite la ejecución de acciones sin conocimiento ni consentimiento del usuario afectado.

A6 - Vulnerable Components - Librerías de Terceros

Este apartado analiza el riesgo derivado de la inclusión de componentes externos que no han sido debidamente actualizados o que presentan fallos de seguridad documentados. Una aplicación puede ser vulnerable si utiliza dependencias con fallos conocidos que un atacante puede aprovechar para comprometer la integridad o confidencialidad de los datos.

El objetivo principal consiste en identificar una librería vulnerable cargada por la aplicación y demostrar la explotación de un fallo de **Cross-Site Scripting (XSS)** presente en dicha versión específica

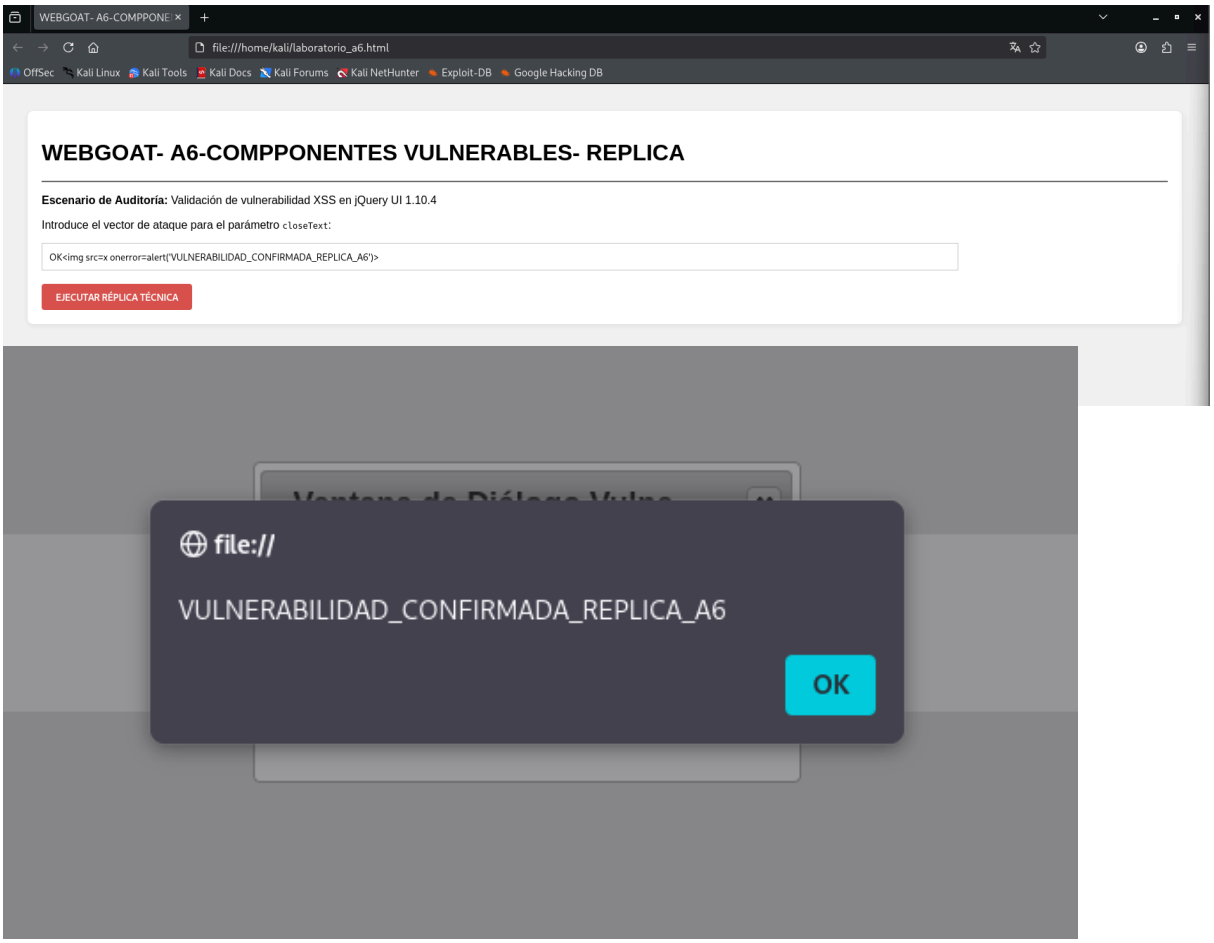
1. **Reconocimiento de dependencias:** A través de las herramientas de desarrollo del navegador, se inspecciona el árbol de directorios de la aplicación, localizando en la carpeta de librerías el archivo **jquery-ui-1.10.4.js.**

2. **Análisis de vulnerabilidades conocidas:** Se verifica que la versión **1.10.4** de jQuery UI posee una vulnerabilidad reportada que permite la ejecución de scripts arbitrarios a través de la propiedad **closeText** en los diálogos de la interfaz.

3. **Ataque:** se introdujo un payload de JavaScript en el campo de entrada. Al procesar la solicitud, la librería no realiza una limpieza (sanitización) adecuada del contenido, permitiendo que el navegador interprete y ejecute el código malicioso.

Durante la fase de explotación, se detectó un comportamiento anómalo en el entorno de laboratorio que impedía la ejecución correcta de la librería dentro del contenedor de

WebGoat. Para confirmar la viabilidad del ataque y descartar un falso negativo, se procedió a realizar una prueba de concepto en un entorno local controlado, recreando fielmente las condiciones de la aplicación



Como se ve en las capturas, al no funcionar correctamente la página de WebGoat, creamos una copia del ejercicio en nuestra computadora. Al probar el código **OK<img src=x onerror=alert('VULNERABILIDAD_CONFIRMADA_REPLICA_A6')>**, confirmamos que la ventana de alerta aparece sin problemas. Esto sucede porque la librería **jQuery UI** es antigua y "se deja engañar", ejecutando el código que escribimos en lugar de mostrarlo como simple texto.

¿Qué significa esto? Si un atacante logra que un usuario caiga en este fallo, podría:

- **Robar su sesión:** Entrar en la cuenta de la víctima sin saber su contraseña.
- **Engañar al usuario:** Mostrar mensajes falsos o redirigirlo a páginas peligrosas.
- **Controlar el navegador:** Ejecutar órdenes invisibles mientras el usuario navega.

¿Cómo se soluciona?

- **Actualizar:** Cambiar la librería vieja por una versión moderna (1.12.0 o más reciente) que ya tiene este error corregido.
- **Revisión constante:** Usar herramientas que nos avisen automáticamente cuando una pieza de nuestro software se queda vieja o es peligrosa.
- **No confiar en lo que el usuario escribe:** Programar la web para que siempre limpie o ignore códigos extraños antes de mostrarlos en pantalla.

Este ejercicio demuestra que no basta con que nuestro código esté bien escrito; si usamos "piezas" (librerías) de otros que están rotas o viejas, nuestra seguridad también estará en riesgo. La réplica local confirmó que el peligro es real y no un error del sistema de prácticas.

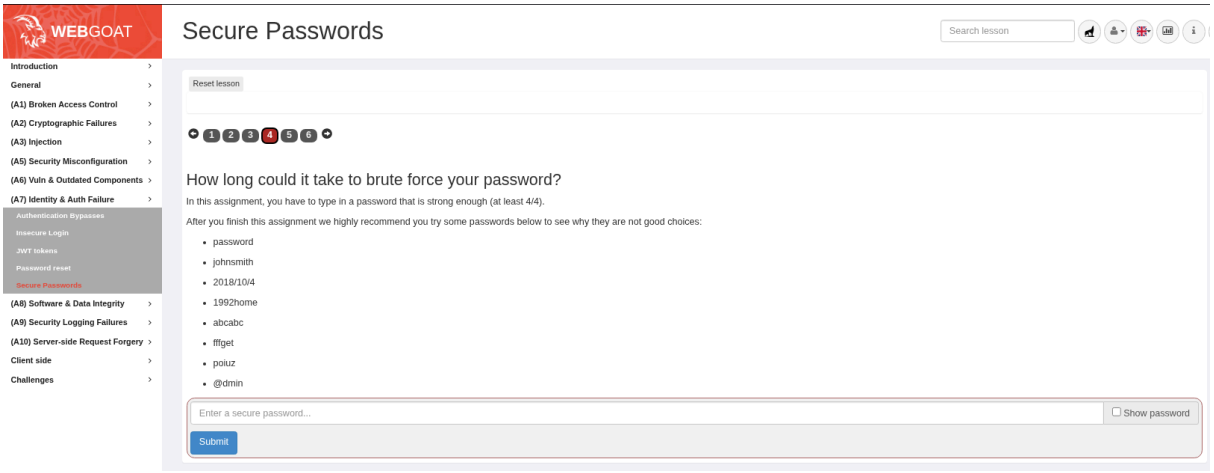
A7- Password Strength

Esta vulnerabilidad se manifiesta cuando los mecanismos de autenticación no son lo suficientemente robustos para impedir el uso de credenciales débiles. Un sistema de autenticación seguro debe obligar al usuario a utilizar contraseñas complejas que resistan ataques de fuerza bruta o diccionarios

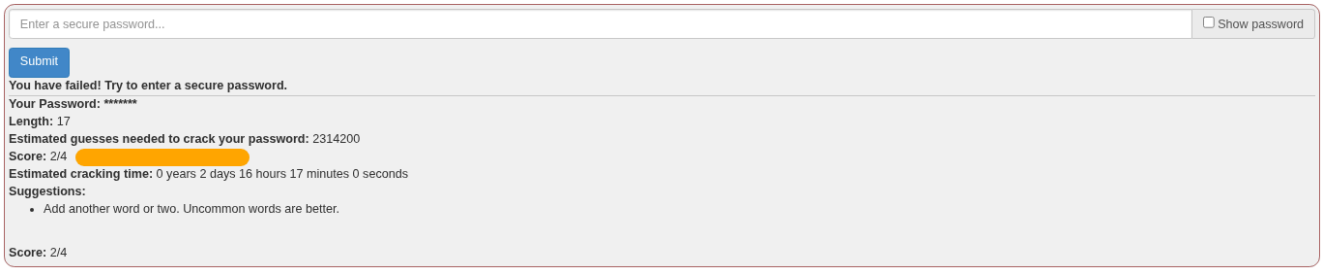
Objetivo: Validar la eficacia de los controles de seguridad implementados por la aplicación para medir la robustez de las contraseñas y asegurar que solo se permitan credenciales que cumplan con un estándar mínimo de seguridad.

Proceso de auditoría:

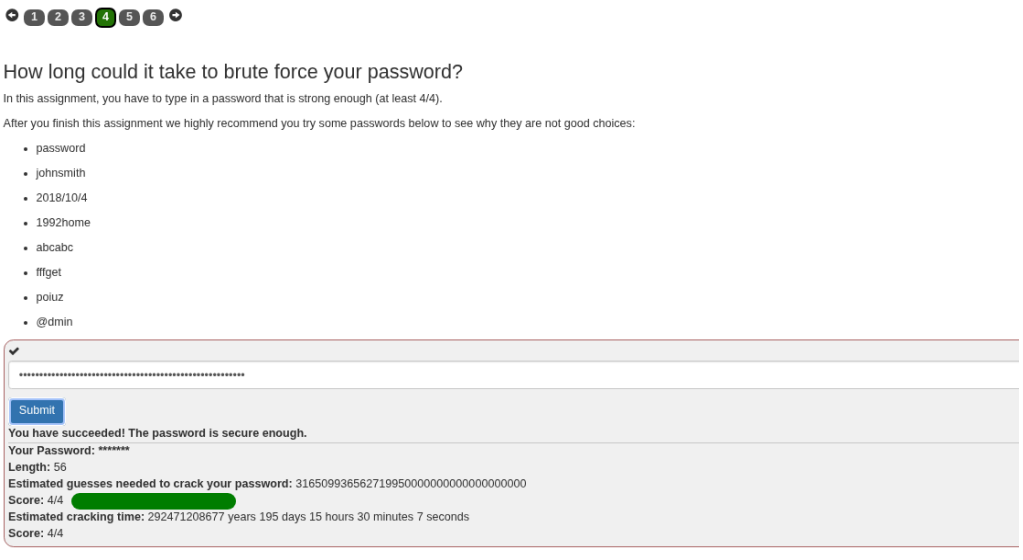
1. **Estado Inicial:** Se accede al módulo de cambio o creación de contraseña. El sistema presenta un medidor de fuerza vacío o en estado crítico, indicando que no hay ninguna entrada válida aún.



2. **Prueba de Seguridad Media:** Se introduce una contraseña corta o predecible. El sistema identifica que la contraseña es débil (representada por una barra de color amarillo y un progreso de 2/4), lo que demuestra que existe un filtro básico, pero que aún permite credenciales vulnerables.



3. **Cumplimiento de Política :** Se introduce una contraseña de alta complejidad (larga, con caracteres especiales, números y mayúsculas). El sistema valida la robustez total y permite completar el ejercicio satisfactoriamente.



La aplicación demuestra tener un control de seguridad activo que evalúa en tiempo real la fortaleza de la clave. Sin embargo, el riesgo de "Identity Failure" persiste si los requisitos mínimos no son lo suficientemente estrictos para bloquear contraseñas de nivel "amarillo" en entornos críticos.

3.3 Post-Explotación/Posibles mitigaciones/ Herramientas utilizadas

Luego de completar las distintas fases de explotación, se pudo comprobar que un ataque coordinado contra WebGoat permitiría comprometer el sistema de forma casi total. La combinación de vulnerabilidades de SQL Injection junto con fallos de autenticación (A7) hace posible escalar privilegios y acceder a cuentas con permisos administrativos. A su vez, el uso de XSS y la presencia de componentes desactualizados (A6) facilitan que el atacante mantenga persistencia dentro de las sesiones de los usuarios.

Además, las malas configuraciones de seguridad detectadas (A5) exponen información interna del sistema, lo que podría ser aprovechado para realizar movimientos laterales dentro de la infraestructura.

Para reducir este tipo de riesgos, resulta fundamental aplicar un control más estricto sobre las entradas del usuario, utilizando consultas parametrizadas y un correcto filtrado y codificación de las salidas. También es necesario mantener actualizadas las librerías de terceros y reforzar las políticas de contraseñas para evitar accesos indebidos. Por último, el endurecimiento de la configuración del servidor es clave para minimizar la exposición de información sensible.

El desarrollo de esta auditoría fue posible gracias al uso de un entorno controlado mediante Docker, el empleo de Nmap para tareas de reconocimiento, las herramientas de desarrollo del navegador para el análisis de scripts y un laboratorio local para validar los distintos ataques, lo que permitió realizar un análisis completo y práctico de la seguridad del sistema.