Proyecto NodeGoat

Revisión y solución de vulnerabilidades

Autor:
Pablo Díaz

Índice

1. Introducción

En esta práctica llevaremos a cabo un análisis de seguridad de la aplicación web NodeGoat, un poryecto de OWASP diseñado para comprender y estudiar vulnerabilidades en aplicaciones web. Como parte de esta práctica, desplegamos NodeGoat en un entorno local mediante Docker y luego utilizamos la herramienta ZAP de OWASP para identificar posibles fallos de seguridad.

2. Implantación NodeGoat

En primer lugar, debemos instalar NodeGoat. Para ello, nos dirigimos a su repositorio oficial en github: *https://github.com/OWASP/NodeGoat* donde contamos con varias opciones para instalarlo. En mi caso, opté por la opción 2: Instalación mediante Docker. Por lo tanto, instalé Docker y seguí las instrucciones dadas en el github. Una vez configurado correctamente nos aparece así:

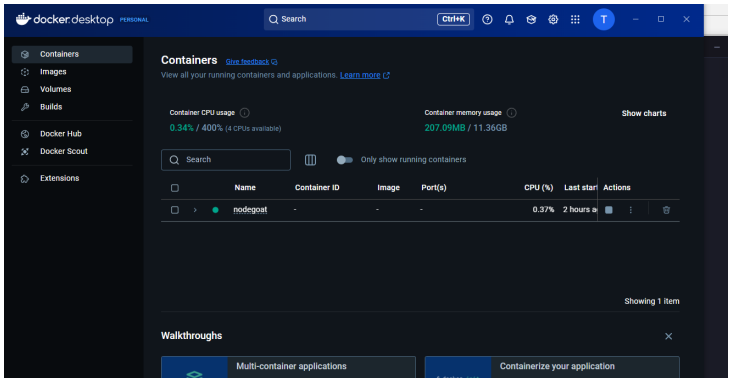


Figura 1: NodeGoat en Docker.

Esta es la aplicación de docker donde podemos arrancar y/o apagar nuestras máquinas.

Para verificar que la aplicación web se ha desplegado correctamente, accedemos a *http://localhost:4000* que es la URL local donde se está ejecutando NodeGoat. Iniciamos sesión con usuario: admin y contraseña: Admin_123

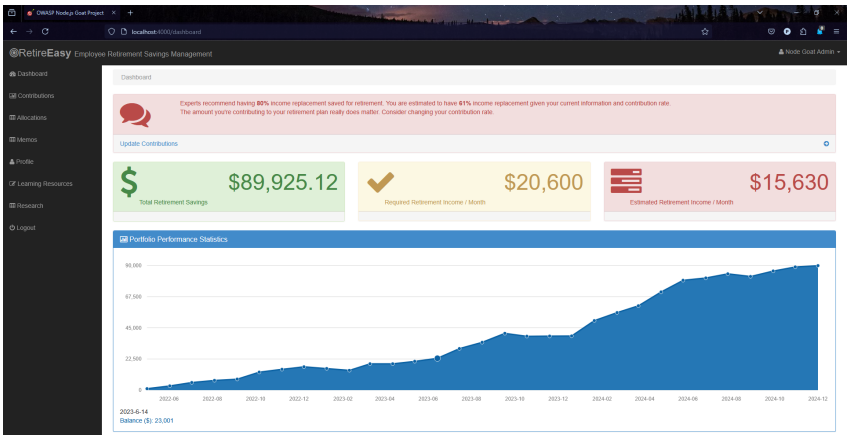


Figura 2: Interfaz de la web.

Ahora que hemos comprobado que está todo configurado, descargamos e instalamos la herramienta de escaneo de vulnerabilidades ZAP. Una vez instalada, la abrimos:

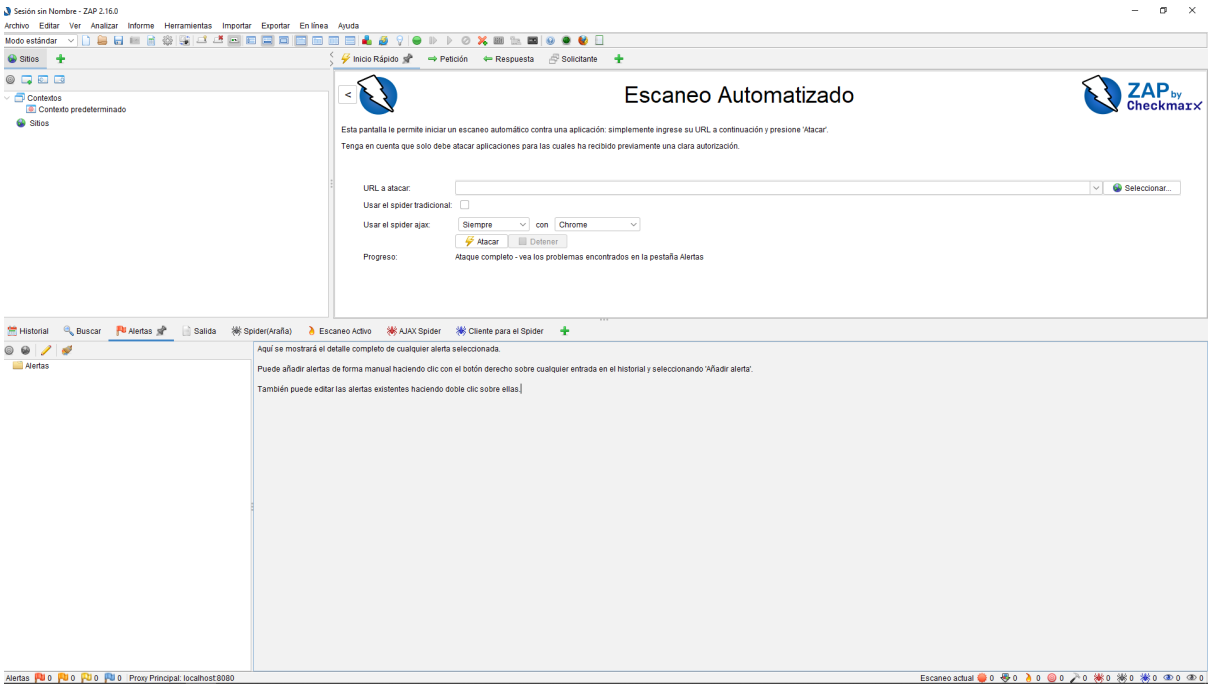


Figura 3: OWASP ZAP.

La forma más sencilla de utilizar ZAP es mediante escaneo automático, y es el que llevaremos a cabo. Hay dos tipos: Traditional Spider y Ajax Spider. Independientemente del tipo, ZAP en primer lugar, rastrea la aplicación web, identificando las direcciones, funcionalidades y parámetros disponibles. Luego, escanea de forma pasiva analizando las solicitudes y respuestas sin modificar el tráfico, con el objetivo de detectar vulnerabilidades en los encabezados, cookies, etc. Por último, utiliza el escáner activo, esto permite realizar pruebas automáticas de ataques, identificando vulnerabilidades como inyeccion SQL, XSS, etc.
Comenzamos con Traditional Spider

3. Exploración automática: Traditional Spider

El modo traditional spider se basa en seguir los enlaces HTML encontrados en el código fuente de las páginas y recorrer todas las rutas accesibles desde ellas. Este método es rápido y efectivo para descubrir páginas estáticas pero no siempre es efectivo para páginas web que tengan mucha dependencia de JavaScript.

Para activar el modo tradicional, lo seleccionamos en el ánlisis automático y ponemos Ajax Spider en nunca para que no escanee con Ajax:

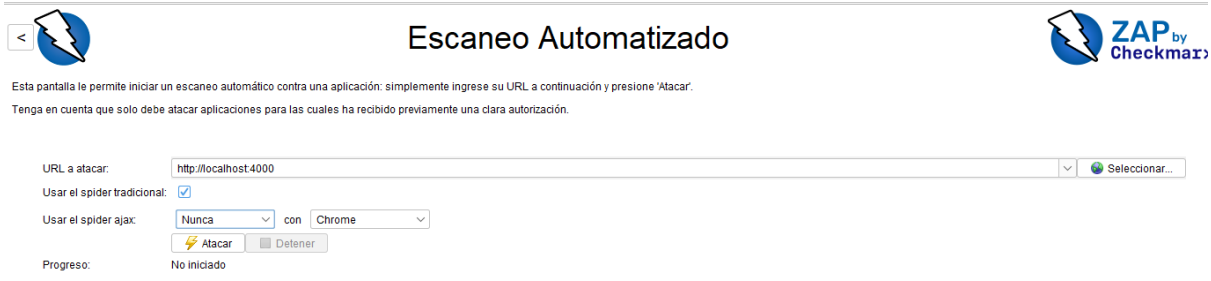


Figura 4: Traditional Spider.

Le damos a atacar y esperamos a que termine. Una vez realizado el ataque nos aparecen las vulnerabilidades encontradas a la izquierda:

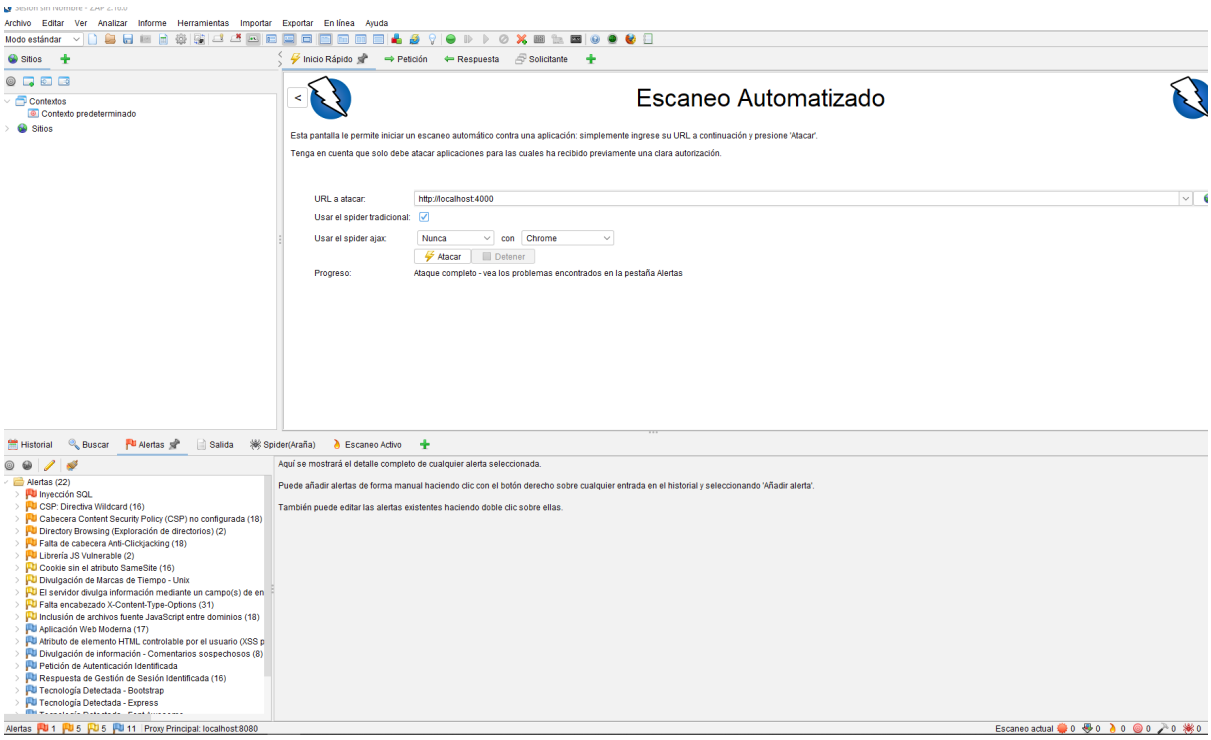


Figura 5: Exploración automática: Traditional Spider.

Hemos encontrado 22 alertas. Hay diferentes tipos dependiendo de la gravedad:

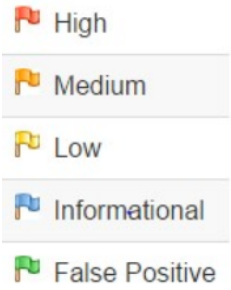


Figura 6: Tipos de alertas ZAP.

Pasamos ahora a usar Ajax Spider. En mi caso, opté por abrir nueva sesión para hacer el escaneo Ajax y poder ver así las diferencias entre ambos. Sin embargo, es muy común también realizar ambos ataques (Ajax y Traditional) de forma conjunta (en la misma sesión) para que se complementen.

Es importante comentar que realicé estos ataques en diferentes momentos y por ello, los resultados pueden variar debido a factores como cambios en la configuración del servidor, variaciones en la respuesta de la web, latencia de la red en determinados momentos.

4. Exploración automática: Ajax Spider

Ajax Spider explorará rutas dinámicas generadas por JavaScript. Desmarcamos la opción de traditional y seleccionamos siempre en Ajax:

Figura 7: Ajax Spider.

Una vez terminado, revisamos las alertas obtenidas:

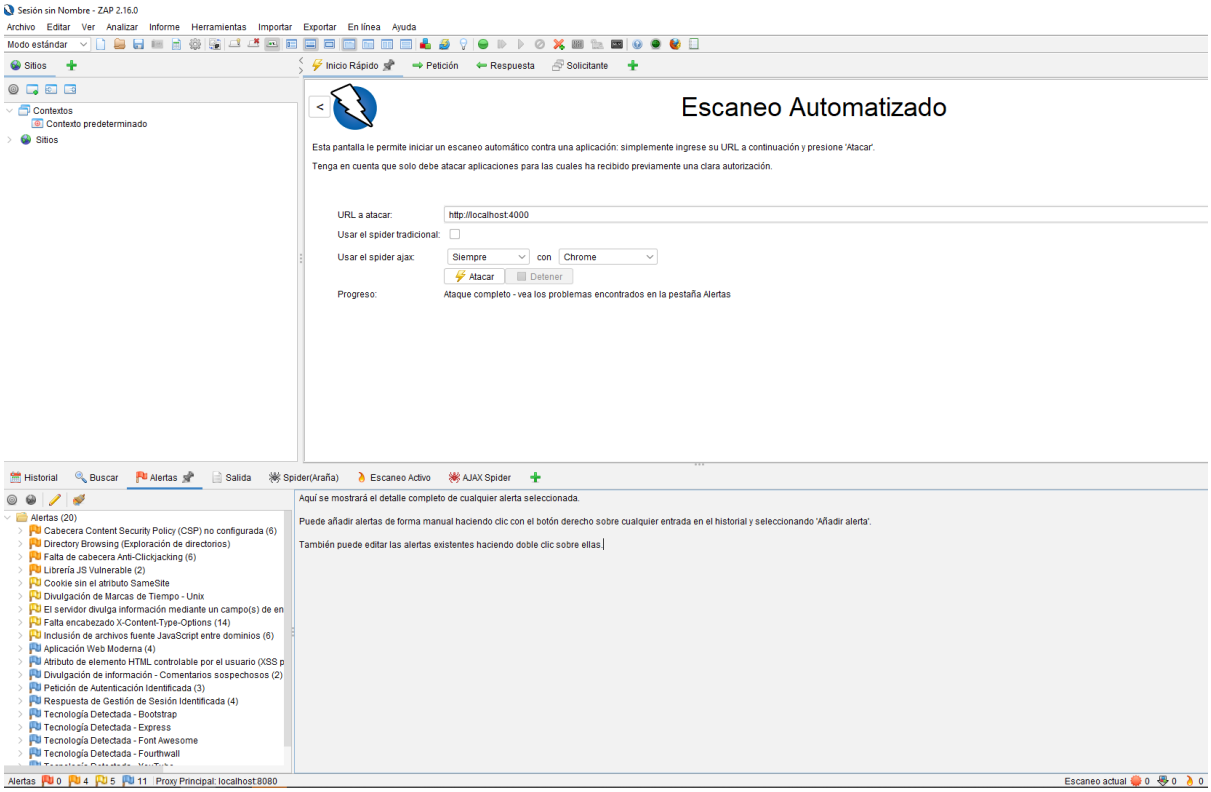


Figura 8: Exploración automática: Ajax Spider.

Obtenemos 20 alertas con Ajax. La diferencia entre el número de alertas puede deberse a que el escaneo tradicional analiza todas las solicitudes HTTP de forma directa, mientras que el escaneo Ajax depende de la ejecución de JavaScript y la interacción dinámica con la página. A parte de lo ya comentado de diferentes resultados en diferentes sesiones.

5. Ataque definiendo credenciales

Si obtenemos credenciales de acceso a la aplicación web, ZAP nos permite realizar ataques autenticados, lo que facilita el acceso a más recursos y la identificación de vulnerabilidades que solo son visibles para usuarios registrados.

Para habilitar la opción autenticada, nos dirigimos a herramientas en la parte superior y luego a tester de autenticación:

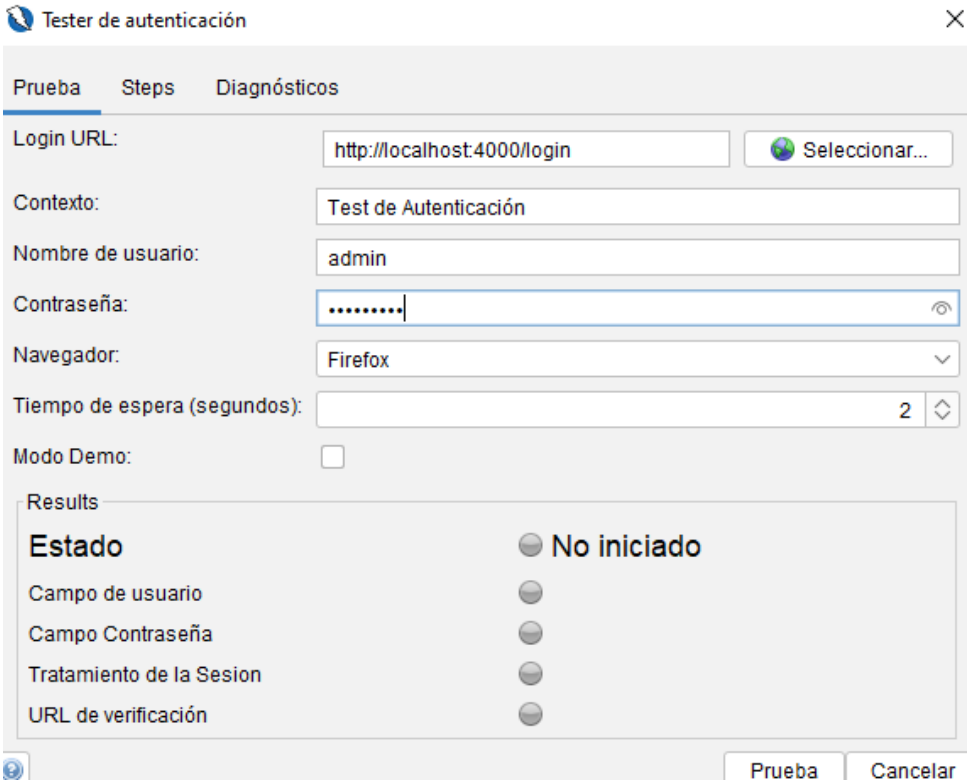
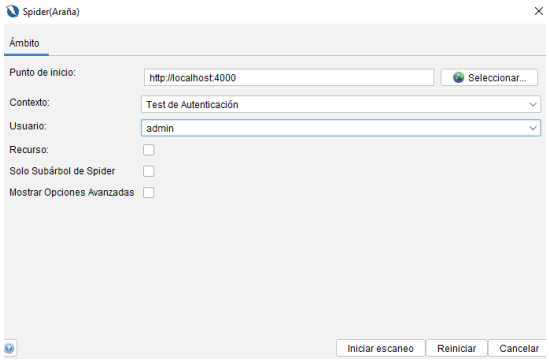
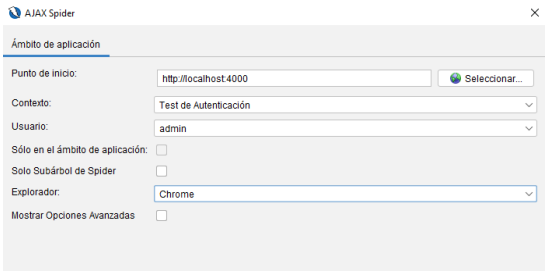


Figura 9: Tester autenticación

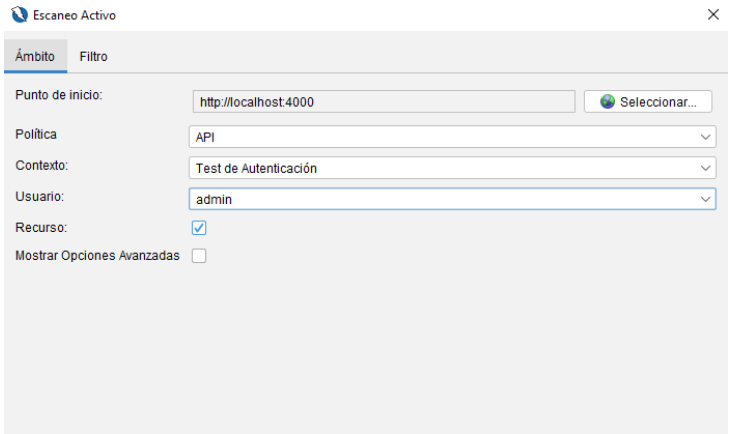
Debemos pulsar en Prueba para confirmar que está todo correcto. En este caso, decidí utilizar traditional, Ajax y escaneo activo todos juntos en la misma sesión de manera autenticada:



(a) Traditional spider autenticado



(b) Ajax autenticado



(c) Activo autenticado

Al poder acceder a más recursos y direcciones encontramos más alertas:

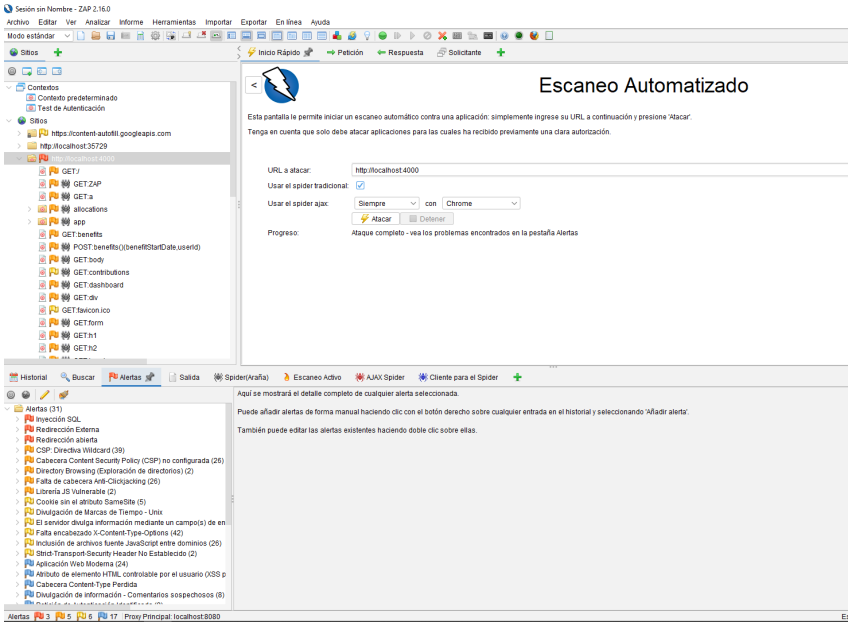


Figura 11: Alertas autenticado

6. Análisis de vulnerabilidades

Ahora que hemos encontrado diferentes vulnerabilidades, debemos elegir tres distintas, explicarlas, comentar el motivo por el que se producen y como podríamos solucionarlas. Elegí **SQL Injection, Redirección abierta y Librería JS vulnerable**.

6.1. SQL Injection

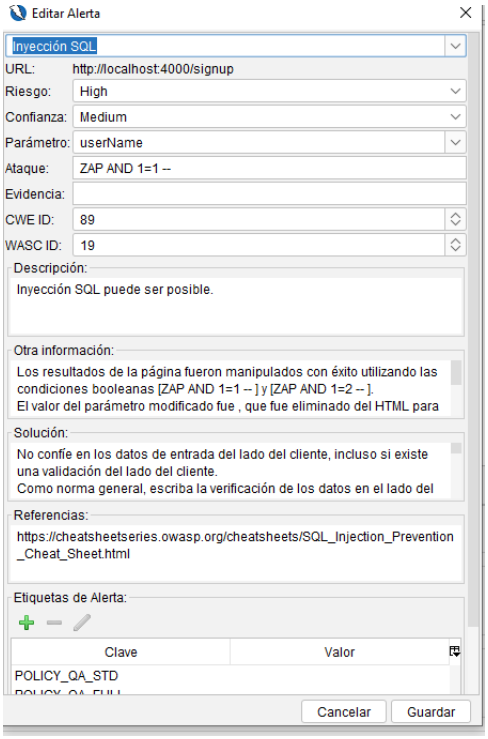


Figura 12: Inyección SQL

Una vulnerabilidad de inyección SQL permite a un atacante manipular consultas SQL enviando valores maliciosos en los parámetros de entrada, pudiendo permitir accesos no autorizados, modificacion de datos o incluso eliminación de información en la base de datos.

Se produce cuando la aplicacion concatena directamente las entradas del usuario en consultas SQL sin validarlas o parametrizarlas. Para prevenirlo, es fundamental no confiar en los datos proporcionados por el usuario y permitir únicamente entradas controladas mediante consultas parametrizadas o mecanismos de escape seguros. Vamos a modificar el código para evitar esta vulnerabilidad.

La vulnerabilidad fue encontrada en el apartado signup. Como la máquina del servidor es nuestra, tenemos acceso al backend. Accedemos al código fuente dentro del contenedor Docker y localizamos en session.js la función handleSignup, encargada de gestionar el proceso de registro. Allí identificamos que se llamaba a userDAO.getUserByUserName. Seguimos rastreando esta función hasta user-dao.js, donde encontramos que realizaba la consulta en MongoDB sin aplicar ninguna validación sobre el dato recibido. Esto podría permitir que un atacante pueda manipular la consulta mediante operadores especiales y potencialmente acceder a información no autorizada.

```
this.getUserByUserName = (userName, callback) => {
    usersCol.findOne({
        userName: userName
    }, callback);
};
```

Figura 13: Parte código vulnerabilidad inyección SQL

Para la correción, implementamos un control previo que verifica que el nombre de usuario solo contenga caracteres alfanuméricos, evitando así la ejecución de operadores especiales en la consulta:

```
this.getUserByUserName = (userName, callback) => {
    // Validar que userName sea un string y solo contenga letras y números
    if (typeof userName !== "string" || !/^[a-zA-Z0-9]+$/.test(userName)) {
        return callback(new Error("Invalid username format"), null);
    }

    usersCol.findOne({
        userName: userName
    }, callback);
};
```

Figura 14: Corrección inyección SQL

Continuamos con redirección abierta.

6.2. Redirección abierta

La vulnerabilidad detectada es una redirección no validada (Open Redirect - CWE-601). Esto ocurre cuando una aplicación permite a los usuarios proporcionar una URL como parámetro y la usa directamente para redirigir sin validación. Esto puede ser aprovechado por atacantes para engañar a los usuarios y enviarlos a sitios maliciosos, facilitando ataques como phishing, robo de credenciales o descargas de malware. La vulnerabilidad se debe a que el código de la aplicación acepta cualquier URL proporcionada por el usuario y la usa en una redirección sin comprobar su origen.

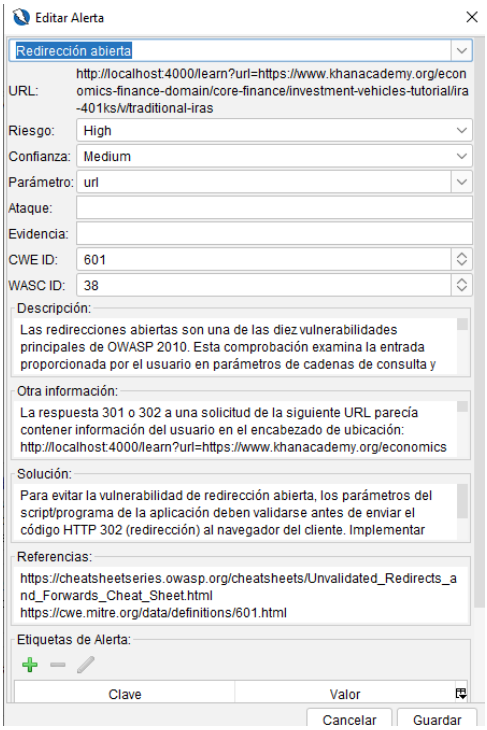


Figura 15: Vulnerabilidad redirección abierta

Buscamos de nuevo en el código fuente de nuestra aplicación web. Localizamos la implementación en routes/index.js, donde se utilizaba directamente el parámetro url proporcionado por el usuario sin aplicar validaciones.

```
// Handle redirect for learning resources link
app.get("/learn", isLoggedIn, (req, res) => {
    // Insecure way to handle redirects by taking redirect url from query string
    return res.redirect(req.query.url);
});
```

Figura 16: Parte código redirección abierta

Para mitigar la vulnerabilidad, implementamos una validación que restringe las redirecciones únicamente a dominios de confianza, evitando así el uso de enlaces externos no autorizados:

```
const allowedDomains = ["example.com", "owasp.org", "khanacademy.org"];

app.get("/learn", isLoggedIn, (req, res) => {
    const url = new URL(req.query.url, "http://default.com"); // Evita errores con URLs
    relativas
    if (allowedDomains.some(domain => url.hostname.endsWith(domain))) {
        return res.redirect(req.query.url);
    }
    return res.status(400).send("Redirección no permitida.");
});
```

Figura 17: Corrección redirección abierta

Hemos entonces creado una lista de dominios de confianza (allowedDomains) que serán los únicos a los que permitiremos redirigir.

Continuamos con la última vulnerabilidad.

6.3. Librería JS Vulnerable

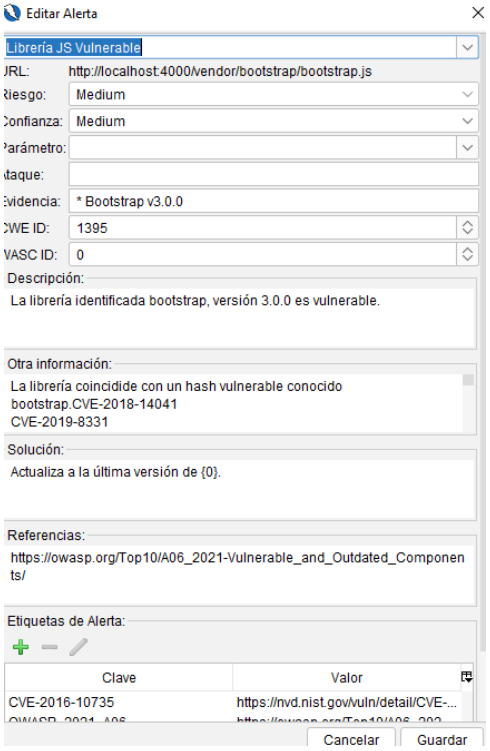


Figura 18: Librería JS vulnerable

ZAP ha detectado que la aplicación está utilizando una versión vulnerable de Bootstrap (3.0.0), la cual presenta varias vulnerabilidades conocidas asociadas a distintos CVE. Estas podrían ser explotadas para realizar ataques como Cross-Site Scripting (XSS), manipulación de la interfaz o ejecución de código malicioso en el navegador del usuario. La presencia de una versión obsoleta de esta librería representa un riesgo significativo para la seguridad de la aplicación, ya que puede permitir que atacantes exploten fallos previamente documentados y comprometidos.

```
/*!
 * Bootstrap v3.0.0
 *
 * Copyright 2013 Twitter, Inc
 * Licensed under the Apache License v2.0
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Designed and built with all the love in the world @twitter by @mdo and @fat.
 */
```

Figura 19: ZAP identifica librería vulnerable

Esta vulnerabilidad se debe a que la aplicación emplea una versión obsoleta de Bootstrap que contiene estos fallos de seguridad. Para mitigar el riesgo, se recomienda actualizar Bootstrap a una versión más reciente y segura, reemplazando la referencia en el código HTML y asegurándose de que la nueva versión esté correctamente implementada en el sistema.
Además, se recomienda verificar que no existan dependencias en el código que puedan forzar el uso de versiones antiguas de Bootstrap. Es importante realizar pruebas funcionales tras la actualización para confirmar que no se generan problemas de compatibilidad con otros elementos de la aplicación. Asimismo, se sugiere utilizar un sistema de gestión de dependencias que permita mantener actualizadas las librerías y recibir alertas ante futuras vulnerabilidades.