**Informe Técnico**

**Seguridad en APIs RESTful y Mitigación de Vulnerabilidades OWASP – Lección 6**

**1. Escenario**

Se analizó un fragmento de código en **PHP** correspondiente a una API que permite eliminar usuarios mediante un endpoint vulnerable. El código recibía los parámetros id y role desde la URL sin validaciones, ejecutando directamente la consulta SQL. Este diseño expone al sistema a múltiples vulnerabilidades críticas. El objetivo del laboratorio fue **detectar los fallos de seguridad** y **reescribir el código siguiendo buenas prácticas OWASP** para mitigar los riesgos.

**2. Análisis de Vulnerabilidades**

1. **Inyección SQL (SQLi):**
   * Uso directo de $\_GET['id'] en la consulta SQL.
   * Riesgo: ejecución de consultas arbitrarias como id=1 OR 1=1 que borrarían múltiples usuarios.
2. **Broken Access Control:**
   * El control de rol se basa en un parámetro manipulable ($\_GET['role']).
   * Riesgo: un atacante puede cambiar role=admin en la URL y obtener privilegios.
3. **Falta de Autenticación Robusta:**
   * No existe sesión o verificación de usuario autenticado.
   * Riesgo: cualquier cliente puede acceder al endpoint y eliminar usuarios.

**3. Código Inseguro Original**

<?php

$id = $\_GET['id'];

if ($\_GET['role'] === 'admin') {

$conexion->query("DELETE FROM usuarios WHERE id = $id");

echo "Usuario eliminado";

} else {

echo "Acceso denegado";

}

?>

**4. Reescritura Segura del Código**

Se aplicaron medidas de ciberseguridad alineadas con **OWASP Top 10**:

<?php

session\_start();

require 'conexion.php';

// Verificar que el usuario esté autenticado y tenga rol admin

if (!isset($\_SESSION['user']) || $\_SESSION['role'] !== 'admin') {

die("Acceso denegado");

}

// Validar entrada

$id = filter\_input(INPUT\_GET, 'id', FILTER\_VALIDATE\_INT);

if ($id === false) {

die("ID inválido");

}

// Uso de consultas preparadas

$stmt = $conexion->prepare("DELETE FROM usuarios WHERE id = ?");

$stmt->bind\_param("i", $id);

$stmt->execute();

echo "Usuario eliminado de forma segura";

?>

**Mejoras implementadas:**

* Verificación de sesión/autenticación en vez de parámetros GET.
* Validación estricta del parámetro id.
* Uso de **prepared statements** para prevenir inyección SQL.
* Aplicación del principio de **mínimo privilegio**: solo admins autenticados pueden ejecutar la acción.

**5. Pruebas de Seguridad (simulación con Postman/curl)**

| **Escenario de prueba** | **Resultado inseguro** | **Resultado seguro** |
| --- | --- | --- |
| No hay sesión activa | Acción permitida ❌ | Acceso denegado ✅ |
| Rol manipulado en la URL | Acción permitida ❌ | Acceso denegado ✅ |
| id manipulado (ej. 1 OR 1=1) | Elimina múltiples usuarios ❌ | Petición rechazada ✅ |
| Rol admin válido y sesión activa | Elimina usuario | Elimina usuario |

**6. Reflexión**

* **Ventaja de roles en sesión sobre parámetros GET:** evita manipulación directa de la URL y asegura que los privilegios sean gestionados por el servidor.
* **Peligro de concatenar valores de entrada:** abre la puerta a **inyección SQL** y ejecución de código arbitrario que compromete la base de datos.
* La práctica mostró la importancia de aplicar controles de acceso robustos, sanitización de entradas y consultas parametrizadas en cualquier API RESTful.

**7. Conclusión**

El análisis evidenció vulnerabilidades críticas como **SQLi, Broken Access Control y falta de autenticación** en el código original. La reescritura segura demostró cómo aplicar buenas prácticas OWASP mitiga eficazmente estos riesgos. Este ejercicio refuerza que el desarrollo de APIs debe incluir desde el inicio: **gestión segura de sesiones, validación de entradas y consultas parametrizadas**, garantizando la confidencialidad e integridad de los datos.