**Memoria Explicativa.**

**SEGURIDAD EN APLICACIONES Y BASES DE DATOS.**



Jesús Luna García

ÍNDICE:

[Ejercicio 1: Reflejado y no almacenado 3](#_Toc152009585)

[Ejercicio 2: Persistente 6](#_Toc152009586)

[Ejercicio 3: DOM-based 10](#_Toc152009587)

[Ejercicio 4: Reflejado en contexto de JavaScript 13](#_Toc152009588)

[Ejercicio 5: Ruptura de contexto HTML 15](#_Toc152009589)

[Ejercicio 6: Filtro de palabras clave 17](#_Toc152009590)

[Protección contra Vulnerabilidades XSS 19](#_Toc152009591)

### Ejercicio 1: Reflejado y no almacenado

#### Descripción y Diferenciación

* **Descripción**: Este es un ejemplo básico de un ataque XSS reflejado, donde el usuario introduce un texto que se refleja directamente en la página.
* **Diferenciación**: Este ejercicio es una introducción simple, mostrando cómo el contenido no filtrado del usuario puede ser peligroso.

#### Vulnerabilidad:

* **Tipo:** XSS Reflejado (No almacenado)
* **Descripción:** La vulnerabilidad se encuentra en la manipulación de la entrada del usuario, que se refleja directamente en la página.

#### Código Vulnerable:

* **Ubicación:** Parámetro de la URL o campo de entrada.
* **Problema:** La página toma el valor del parámetro y lo inserta directamente en el HTML sin ninguna validación o codificación.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

#### Explotación:

<script>alert('XSS')</script>

Interfaz de usuario gráfica

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### Ejercicio 2: Persistente

#### Descripción y Diferenciación

* **Descripción**: En este ejercicio, la entrada del usuario se refleja en un atributo HTML.
* **Diferenciación**: Diferente del primero, aquí el desafío es inyectar un script en el contexto de un atributo, no directamente en el cuerpo del HTML.

#### Vulnerabilidad:

* **Tipo:** XSS Almacenado (Persistente)
* **Descripción:** La entrada del usuario se almacena en el servidor y luego se muestra a los usuarios, incluyendo scripts maliciosos.

#### Código Vulnerable:

* **Ubicación:** Formulario de comentarios.
* **Problema:** La aplicación guarda y muestra el comentario sin filtrar el contenido HTML o JavaScript.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

El problema de no sanitizar la entrada es que al poner alguna etiqueta HTML como <u>test</u>, la ejecuta.

Interfaz de usuario gráfica, Aplicación, Sitio web

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para este caso, la ejecución de script si que la sanitiza, pero si probamos con la etiqueta <img> no.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

#### Explotación:

<img src="123" onerror="alert('Stored XSS')">

### Ejercicio 3: DOM-based

#### Descripción y Diferenciación

* **Descripción**: Aquí se trata de un XSS almacenado, donde el script inyectado se guarda en el servidor y afecta a todos los usuarios que ven la página.
* **Diferenciación**: A diferencia de los anteriores, este script no solo afecta al usuario que lo inyecta sino a cualquiera que acceda a esa parte del sitio.

#### Vulnerabilidad:

* **Tipo:** XSS Basado en DOM
* **Descripción:** El script en el cliente (JavaScript) escribe datos no confiables en el DOM.

#### Código Vulnerable:

* **Ubicación:** Generalmente en el JavaScript del lado del cliente.
* **Problema:** La aplicación escribe directamente en el DOM con datos controlados por el usuario sin adecuada validación.

Interfaz de usuario gráfica, Texto, Aplicación, Sitio web

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Teams

Descripción generada automáticamente

#### Explotación:

* Cambio en la URL o manipulación del DOM para incluir:

#1’ onerror=”alert(‘XSS')”

### Ejercicio 4: Reflejado en contexto de JavaScript

#### Descripción y Diferenciación

* **Descripción**: Este ejercicio introduce una inyección dentro del contexto de un código JavaScript.
* **Diferenciación**: A diferencia de los anteriores, aquí el reto es inyectar código en un script JavaScript, no directamente en HTML.

#### Vulnerabilidad:

* **Tipo:** XSS Reflejado en contexto de JavaScript
* **Descripción:** La entrada del usuario se inserta en un contexto de código JavaScript.

#### Código Vulnerable:

* **Ubicación:** Dentro de un bloque de script en la página.
* **Problema:** La entrada del usuario no se valida adecuadamente antes de incluirla en el código JavaScript. Esto da pie a que se pueda continuar con el código JavaScript que se espera, mediante la comilla simple y el ;.

Captura de pantalla de computadora

Descripción generada automáticamente

En esta imagen vemos el código de la sentencia JavaScript.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Captura de pantalla de un celular

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

Aquí se muestra la sentencia tras la inyección del código.

#### Explotación:

'); alert('XSS)

### Ejercicio 5: Ruptura de contexto HTML

#### Descripción y Diferenciación

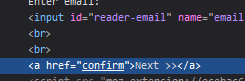
* **Descripción**: Este desafío se centra en el XSS basado en DOM, donde el script malicioso modifica el DOM de la página en tiempo de ejecución.
* **Diferenciación**: A diferencia de los ejercicios anteriores, aquí el foco está en manipular el DOM directamente con JavaScript, no mediante la inserción en HTML.

#### Vulnerabilidad:

* **Tipo:** Ruptura de contexto HTML a través de XSS
* **Descripción:** La entrada del usuario rompe el contexto HTML y permite la inyección de contenido malicioso.

Interfaz de usuario gráfica

Descripción generada automáticamente



Como se ve el botón Next redirige a confirm que está en la URL, pero si la cambiamos por código lo ejecutará.

Texto

Descripción generada automáticamente con confianza baja

Interfaz de usuario gráfica, Sitio web

Descripción generada automáticamente

Tras darle al botón Next, se ejecuta el código que hemos escrito en la url.

#### Código Vulnerable:

* **Ubicación:** Dentro del HTML.
* **Problema:** No se codifican o escapan los caracteres especiales del HTML, permitiendo la ruptura del contexto.

#### Explotación:

=javascript:alert(‘XSS’)

### Ejercicio 6: Filtro de palabras clave

#### Descripción y Diferenciación

* **Descripción**: Este ejercicio introduce XSS en una aplicación AngularJS, un popular framework de JavaScript.
* **Diferenciación**: Aquí, el reto es explotar una vulnerabilidad en el contexto de un framework JavaScript específico.

#### Vulnerabilidad:

* **Tipo:** XSS con Filtro de Palabras Clave
* **Descripción:** Un filtro intenta bloquear ciertas palabras clave o patrones, pero puede ser evadido.

#### Código Vulnerable:

* **Ubicación:** url, donde se introduce la ruta del gadget.
* **Problema:** El filtro no es exhaustivo y puede ser eludido.

Texto

Descripción generada automáticamente

Como se puede ver, la función includeGadget(url) intenta proteger contra la inyección de scripts maliciosos a través de URLs externas. Esta protección se basa en una verificación de la URL proporcionada antes de crear y añadir un elemento de script al DOM (Document Object Model). Los detalles clave de la función son:

* **Creación de un Elemento de Script**: La función comienza creando un elemento script en el documento. Este es un método común para incluir dinámicamente scripts JavaScript en una página web.
* **Verificación de la URL**: La función luego verifica si la URL proporcionada comienza con http:// o https:// mediante una expresión regular (url.match(/^https?:\/\//)).
* **Bloqueo de Cargas Externas**: Si la URL comienza con http:// o https://, la función asume que es una URL externa y potencialmente maliciosa. En este caso, muestra un mensaje de error y no procede a cargar el script.
* **Prevención de XSS**: El propósito principal es prevenir ataques de Cross-Site Scripting (XSS) donde un atacante podría intentar incluir un script malicioso desde un origen externo. Al bloquear las URLs que comienzan con http:// o https://, la función intenta asegurarse de que sólo se carguen scripts de origen seguro (presumiblemente, el propio sitio web).

**Limitaciones y Consideraciones Adicionales**

* **Control de Protocolo, No de Contenido**: La función solo verifica el protocolo de la URL (HTTP o HTTPS), pero no analiza el contenido real del script que se va a cargar. Esto significa que, si bien bloquea los scripts externos, no necesariamente bloquea el contenido malicioso de las URLs que no comienzan con http:// o https://.
* **Posibles Falsos Positivos**: URLs legítimas que comienzan con http:// o https:// también serán bloqueadas, lo que podría limitar la funcionalidad si hay una razón legítima para cargar scripts de estas fuentes.
* **XSS Basado en DOM**: Esta función no protege contra todas las formas de XSS, especialmente el XSS basado en DOM, donde el ataque se realiza a través de manipulaciones del DOM después de que la página se haya cargado.
* **Otras Formas de Inyección**: No protege contra otras formas de inyección, como inyecciones SQL, inyecciones de código de servidor, etc.

Por lo que podemos probar esta vulnerabilidad.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

#### Interfaz de usuario gráfica, Aplicación Descripción generada automáticamente

#### Explotación:

* Utilizar técnicas de ofuscación o métodos alternativos para evadir el filtro (por ejemplo, usando codificación URL o caracteres de escape).

#data:text/plain,alert('XSS')

### Protección contra Vulnerabilidades XSS

Para protegerse de las vulnerabilidades XSS, es esencial:

1. **Validación de Entradas:** Verificar y restringir lo que los usuarios pueden ingresar.
2. **Codificación de Salida:** Codificar los datos de salida para asegurar que cualquier entrada del usuario se trate como datos y no como código.
3. **Uso de Listas Blancas:** Permitir solo un conjunto específico de caracteres en las entradas del usuario.
4. **Contenido Seguro:** Implementar políticas de seguridad de contenido (Content Security Policy - CSP) para ayudar a prevenir XSS.
5. **Pruebas y Herramientas:** Utilizar herramientas automatizadas y pruebas de penetración para identificar vulnerabilidades XSS.
6. **Educación y Conciencia:** Capacitar a los desarrolladores sobre prácticas seguras de codificación y estar al tanto de las últimas técnicas de explotación y mitigación.