

21



**Tema:**

SIMULACIÓN Y EXPLOTACIÓN DE VULNERABILIDADES

DE SOFTWARE V1

**Nombre del autor:**

* CRISTIAN CABASCANGO
* JEAN PIERRE SALAZAR

**Tutor:**

ING. ALVARO UYAGUARI

**Latacunga – Ecuador**

2025

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN CARRERA DE INGENIERÍA EN SOFTWARE

ASIGNATURA:

ING. SEGURIDAD DEL SOFTWARE

# INTRODUCCIÓN

En el presente proyecto se ha desarrollado un entorno web controlado cuyo propósito es simular diversas vulnerabilidades comunes en aplicaciones web, con el objetivo de comprender su funcionamiento, alcance y formas de mitigación. Utilizando únicamente tecnologías del lado del cliente (HTML, CSS y JavaScript), se han implementado ejemplos prácticos de ataques como Cross-Site Scripting (XSS), inyección de comandos simulada y Cross-Site Request Forgery (CSRF), en un entorno local gestionado mediante XAMPP y ejecutado en el navegador.

Estas simulaciones permiten observar cómo una aplicación sin mecanismos adecuados de validación, sanitización o control de origen puede quedar expuesta a manipulaciones maliciosas que comprometen la experiencia del usuario, la integridad del sistema o incluso la privacidad de los datos. A través de distintos formularios, se demuestra cómo una simple inyección de código puede alterar la funcionalidad de una página web o ejecutar acciones no autorizadas en nombre de un usuario.

Además, el sistema implementa una bitácora de ataques basada en localStorage, lo cual permite registrar cada intento y visualizar de forma ordenada la evidencia de los ataques ejecutados. Se incluye también una versión segura del sistema, que implementa controles de validación y sanitización como medida preventiva, permitiendo contrastar claramente los efectos de una implementación insegura frente a una segura.

# OBJETIVOS OBJETIVO GENERAL

* Analizar vulnerabilidades comunes en aplicaciones web utilizando tecnologías frontend (HTML, CSS y JavaScript), con el fin de comprender su funcionamiento, impacto y aplicar mecanismos básicos de prevención.

# OBJETIVOS ESPECÍFICOS

* Implementar ejemplos funcionales de vulnerabilidades del tipo XSS, inyección de comandos y CSRF, mediante formularios y scripts manipulables desde el navegador.
* Desarrollar una bitácora de registro local para evidenciar y documentar los intentos de ataque realizados durante la simulación.
* Construir una versión segura de los módulos implementados, aplicando validaciones, sanitización de entradas y restricciones básicas como medida de mitigación.

**DESARROLLO**

| **Concepto** | **Definición** | **Ejemplo** |
| --- | --- | --- |
| Amenaza | Agente que puede explotar una vulnerabilidad. | Un atacante remoto. |
| Vulnerabilidad | Debilidad en el sistema explotable por una amenaza. | Entrada de usuario sin validar. |
| Control físico | Protección del entorno físico. | Acceso restringido a servidores. |
| Control lógico | Medidas técnicas. | Validación de entradas, firewalls. |
| Control administrativo | Procedimientos y políticas. | Política de cambio de contraseñas. |

**Gestión de riesgos:**

1. Identificación
2. Evaluación (probabilidad + impacto)
3. Mitigación
4. Monitoreo

### **Evaluación de Riesgos**

### Clasificación STRIDE

| **Vulnerabilidad** | **Modelo STRIDE** | **CID comprometido** |
| --- | --- | --- |
| OS Command Injection | Tampering, Elevation of Privilege | C, I |
| XSS | Spoofing, Information Disclosure | C, D |

### **Controles de Mitigación Aplicables**

#### 🔐 OS Command Injection

* Validación de entradas.
* Escapar comandos con shlex.quote() o usar subprocess.run() con lista segura.
* Limitar comandos disponibles mediante lista blanca.

#### 🔐 XSS

* Escapar salida ({{ c }} en lugar de {{ c|safe }}).
* Uso de CSP (Content Security Policy).
* Sanitización de entradas con librerías como bleach o DOMPurify.

### **Simulación de Ataques**

#### 📍 OS Command Injection

* Acceso: http://127.0.0.1:5000/ping?host=google.com && whoami
* Resultado: devuelve nombre del usuario del sistema.
* **Compromete:**
  + **Confidencialidad:** el atacante puede leer datos del sistema.
  + **Integridad:** podría modificar archivos si el comando es más agresivo.

#### XSS

* Entrada: <script>alert("XSS")</script>
* Resultado: el script se ejecuta cada vez que alguien accede a la página.
* **Compromete:**
  + **Confidencialidad:** puede robar cookies.
  + **Disponibilidad:** puede alterar la interfaz o impedir su uso.

## Cross-Site Scripting (XSS) Descripción técnica:

La vulnerabilidad de tipo XSS permite a un atacante inyectar código malicioso (generalmente JavaScript) en una página web que luego es ejecutado por el navegador de otro usuario. Esta ejecución ocurre porque la entrada del usuario no es correctamente filtrada o sanitizada antes de ser insertada en el DOM.

## Implementación en el sistema:

En este proyecto, el XSS fue implementado a través de un formulario en xss.html, que permite a cualquier visitante ingresar texto sin ninguna validación. Este texto se incrusta directamente en el HTML de la página utilizando innerHTML, lo que hace posible ejecutar etiquetas <script> si son ingresadas.

## Código vulnerable (simplificado):

document.getElementById("xss-form").addEventListener("submit", function(e) { e.preventDefault();

const comentario = document.getElementById("comentario").value; const div = document.getElementById("comentarios");

if (comentario.includes("<script>") && comentario.includes("</script>")) { const scriptTag = document.createElement("script");

const scriptContent = comentario.replace("<script>", "").replace("</script>", ""); scriptTag.textContent = scriptContent;

document.body.appendChild(scriptTag);

div.innerHTML += `<p><i>[Script ejecutado con éxito]</i></p>`;

} else {

div.innerHTML += `<p>${comentario}</p>`;

}

});

**Prueba realizada:**

* Se ingresó el siguiente código en el formulario:

## Registro del ataque:

El sistema cuenta con un módulo de bitácora que registra automáticamente el

contenido inyectado, la hora del intento y el tipo de ataque. Estos datos se almacenan en localStorage.

## Medidas de mitigación implementadas:

* En la versión segura (seguro.html), se usa una función escapeHTML() que reemplaza los caracteres especiales como <, >, &, ", ' por sus equivalentes HTML.
* Se implementó una advertencia visual en caso de detección de patrones peligrosos (<script>).

## Función de escape aplicada:

function escapeHTML(str) { return str

.replace(/&/g, "&amp;")

.replace(/</g, "&lt;")

.replace(/>/g, "&gt;")

.replace(/"/g, "&quot;")

.replace(/'/g, "&#039;");

}

**Resultado de la versión protegida:**

Al intentar ingresar código como <script>alert('hack')</script>, el sistema lo bloquea y muestra el mensaje:

Código malicioso detectado. Comentario bloqueado por seguridad.

## Simulación de Inyección de Comandos Descripción técnica:

Aunque las verdaderas inyecciones de comandos ocurren del lado del servidor, en esta simulación se recrea el comportamiento típico de una entrada de texto que representa un intento de ejecutar comandos del sistema, para evaluar cómo una aplicación podría responder si no existiera ningún tipo de validación o filtrado.

## Implementación en el sistema:

La vulnerabilidad se encuentra simulada en el formulario de la página xss.html, donde el usuario puede ingresar comandos como si estuviera interactuando con una terminal. El sistema reacciona de acuerdo con palabras clave detectadas en la entrada, como si ejecutara comandos del sistema operativo.

## Código relevante (simplificado):

document.getElementById("cmd-form").addEventListener("submit", function(e) { e.preventDefault();

const comando = document.getElementById("comando").value.toLowerCase(); const salida = document.getElementById("salida");

if (comando.includes("dir") || comando.includes("ls")) { salida.textContent = "Archivo1.txt\nArchivo2.txt\nCarpeta/";

} else if (comando.includes("rm") || comando.includes("del") || comando.includes("format")) {

salida.textContent = "¡ALERTA! Comando peligroso ejecutado.";

} else if (comando.trim() === "") {

salida.textContent = "Por favor ingresa un comando.";

} else {

salida.textContent = "Comando no reconocido.";

}

});

**Pruebas realizadas:**

* Entrada: ls

Resultado: Simulación de listado de archivos.

* Entrada: rm -rf /

Resultado: Advertencia visual: **¡ALERTA! Comando peligroso ejecutado.**

## Registro del ataque:

Cada intento de ejecución de comandos se registra en la bitácora, usando localStorage, con el tipo "Command Injection", el contenido ingresado y la fecha/hora.

## Medidas de mitigación implementadas en la versión segura:

* Solo se permite la ejecución de comandos listados en un arreglo de confianza (["dir", "ls", "help"]).
* Los comandos no reconocidos son bloqueados automáticamente con el mensaje: "Comando no permitido".

## Código en versión protegida:

const comandosPermitidos = ["dir", "ls", "help"];

document.getElementById("seguro-cmd-form")?.addEventListener("submit", function(e) {

e.preventDefault();

const comando = document.getElementById("comando").value.toLowerCase().trim(); const salida = document.getElementById("salida-segura");

if (comando === "") {

salida.textContent = "Por favor ingresa un comando.";

} else if (comandosPermitidos.includes(comando)) { salida.textContent = "Comando permitido ejecutado con éxito.";

} else {

salida.textContent = "Comando no permitido.";

}

});

**Resultado de la versión segura:**

Se evita la ejecución de comandos sospechosos o dañinos. Solo comandos simples como dir, ls o help reciben una respuesta simulada aceptable.

## Simulación de Vulnerabilidad CSRF (Cross-Site Request Forgery) Descripción técnica:

CSRF (Cross-Site Request Forgery) es una vulnerabilidad que permite a un atacante ejecutar acciones no autorizadas en nombre de un usuario autenticado, sin su conocimiento. El ataque se produce cuando el navegador de la víctima envía una solicitud maliciosa a una aplicación donde está autenticada, generalmente mediante un formulario HTML oculto o una URL manipulada.

## Implementación en el sistema:

En esta simulación, se creó un formulario vulnerable ubicado en csrf.html que permite cambiar la contraseña de un usuario. Este formulario acepta solicitudes POST sin verificar su origen, lo que lo hace susceptible a una solicitud maliciosa desde un sitio externo.

## Código vulnerable del formulario:

<form id="csrf-form">

<label for="password">Nueva contraseña:</label><br>

<input type="password" id="password" name="password">

<button type="submit">Cambiar contraseña</button>

</form>

## Script que ejecuta el cambio de contraseña:

document.getElementById("csrf-form").addEventListener("submit", function(e) { e.preventDefault();

const nueva = document.getElementById("password").value; const mensaje = document.getElementById("mensaje");

if (!nueva.trim()) {

mensaje.innerText = " ı. Por favor ingresa una contraseña."; return;

}

mensaje.innerText = " ı. Contraseña cambiada a: " + nueva;

const ataques = JSON.parse(localStorage.getItem("bitacora")) || []; ataques.push({

tipo: "CSRF",

dato: "Nueva contraseña establecida: " + nueva, fecha: new Date().toLocaleString()

});

localStorage.setItem("bitacora", JSON.stringify(ataques));

});

## Simulación del ataque:

Se creó una página externa csrf-attack.html con un formulario oculto que, al cargar, envía automáticamente una solicitud POST para cambiar la contraseña de forma silenciosa:

<body onload="document.getElementById('hack-form').submit();" style="display:none;">

<form action[="http://localhost/vulnerabilidades/csrf.html"](http://localhost/vulnerabilidades/csrf.html) method="POST" id="hack- form">

<input type="hidden" name="password" value="hackeado123">

</form>

</body>

## Prueba realizada:

* Al abrir el archivo csrf-attack.html, se envía automáticamente la solicitud para cambiar la contraseña sin interacción del usuario.
* Resultado visible en csrf.html: contraseña modificada y mensaje mostrado automáticamente.
* Registro del intento de ataque en la bitácora.

## Medidas de mitigación (opcionales para proyectos reales):

* Verificación del origen de la solicitud (Origin o Referer).
* Inclusión de tokens CSRF únicos por sesión en formularios.
* Requiere validación del lado del servidor (no aplicable en esta simulación sólo con frontend).

## Resultado de la simulación:

La contraseña fue cambiada automáticamente sin intervención del usuario, demostrando cómo funciona un ataque CSRF básico.

## Registro de Bitácora y Evidencias de Ataques

**Descripción funcional:**

Para simular un entorno más completo y realista, se implementó una **bitácora de ataques** que permite almacenar localmente cada intento de vulneración detectado por el sistema. Esta bitácora funciona mediante localStorage, permitiendo mantener un historial incluso después de recargar la página.

## Implementación técnica:

La función registrarAtaque() es utilizada en todos los formularios vulnerables para guardar el tipo de ataque, los datos ingresados y la fecha del intento.

function registrarAtaque(tipo, dato) {

const ahora = new Date().toLocaleString();

const ataques = JSON.parse(localStorage.getItem("bitacora")) || []; ataques.push({ tipo, dato, fecha: ahora });

localStorage.setItem("bitacora", JSON.stringify(ataques));

}

Esta función es llamada desde eventos como:

document.getElementById("xss-form")?.addEventListener("submit", function(e) { const comentario = document.getElementById("comentario").value;

registrarAtaque("XSS", comentario);

});

## Visualización de la bitácora:

Se habilitó un enlace en el menú principal (index.html) para acceder a la bitácora y ver el historial de ataques registrados.

window.addEventListener("DOMContentLoaded", () => {

const contenedor = document.getElementById("bitacora-lista"); const ataques = JSON.parse(localStorage.getItem("bitacora")) || []; if (ataques.length === 0) {

contenedor.innerHTML = "<p>No se han registrado ataques aún.</p>"; return;

}

ataques.forEach((entrada, index) => { contenedor.innerHTML += `

<div class="registro">

<p><strong>#${index + 1}</strong> [${entrada.fecha}]</p>

<p> ) Tipo: <b>${entrada.tipo}</b></p>

<p> •¸ ‘ ) Dato ingresado: ${entrada.dato}</p>

<hr>

</div>`;

});

## Cross-Site Scripting (XSS)

El XSS permite a un atacante inyectar código malicioso (generalmente JavaScript) en una página web, de forma que ese código es ejecutado en el navegador del usuario cuando la página es cargada o cuando se interactúa con ella.

* Ejecutó código malicioso directamente en el navegador del usuario (alert(), <h1>,

<img onerror>).

* No hubo sanitización inicial del input, permitiendo la inserción directa de etiquetas <script> y otras etiquetas HTML.
* El código malicioso se reflejaba inmediatamente en la vista del usuario, generando efectos visuales o funcionales (como ventanas emergentes).
* Se registró correctamente en la bitácora, evidenciando el impacto del ataque.

`. VULNERABILIDADES-FRONTEND

├── )‘ index.html ← Página principal con menú de navegación

├── ) ‘ xss.html ← Página vulnerable a XSS

├── ‘)) comandos.html ← Página vulnerable a inyección de comandos

├── )‘ csrf.html ← Simulación del sistema atacado por CSRF

├── ‘) csrf-attack.html ← Página de ataque externo simulando CSRF

├── )‘ seguro.html ← Versión protegida de XSS y comandos

├── ‘) bitacora.html ← Visualización de ataques registrados

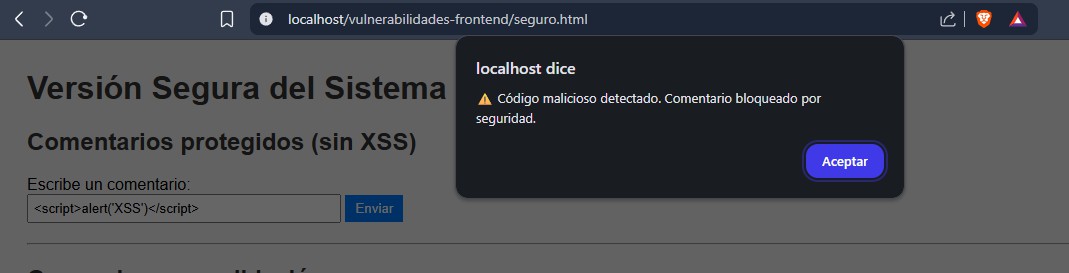
├── ‘) style.css ← Estilos compartidos para toda la interfaz

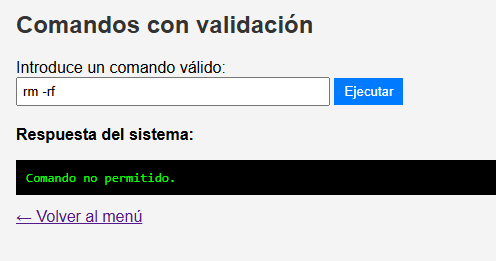
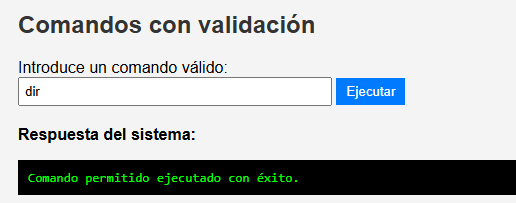
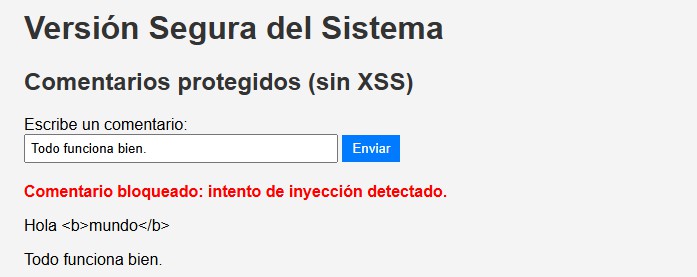
├── ‘) app.js ← Lógica JS para páginas vulnerables

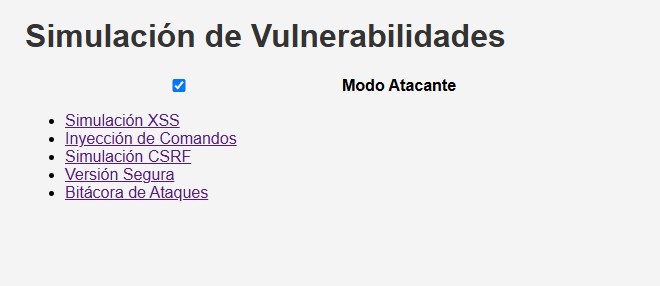
├── ) ‘ utils.js ← Lógica JS para páginas seguras

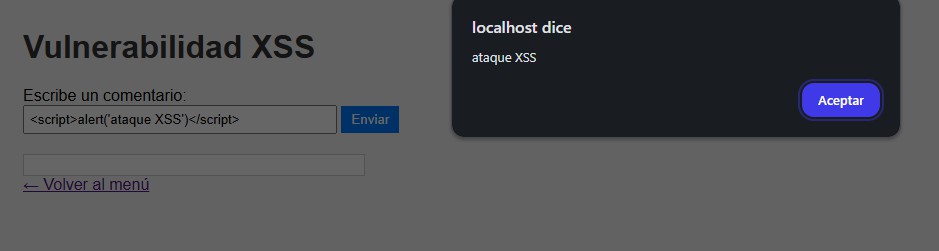
└── ) ‘ README.txt / informe.docx ← (Opcional) documentación técnica



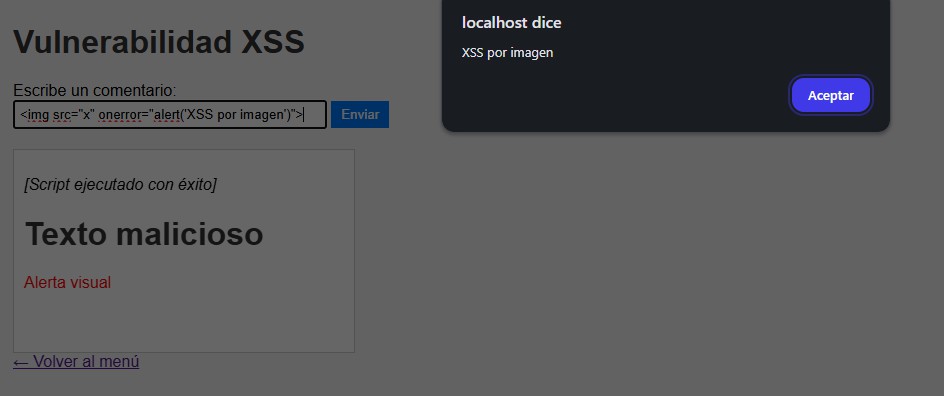


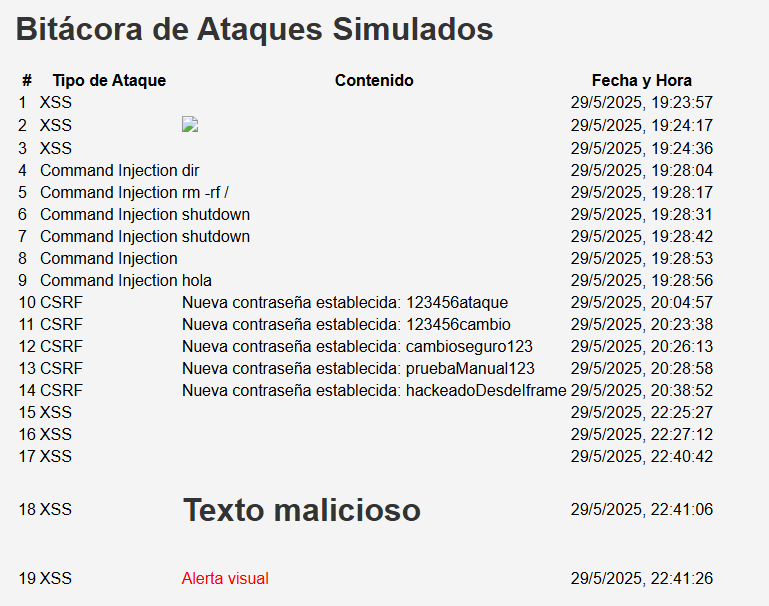




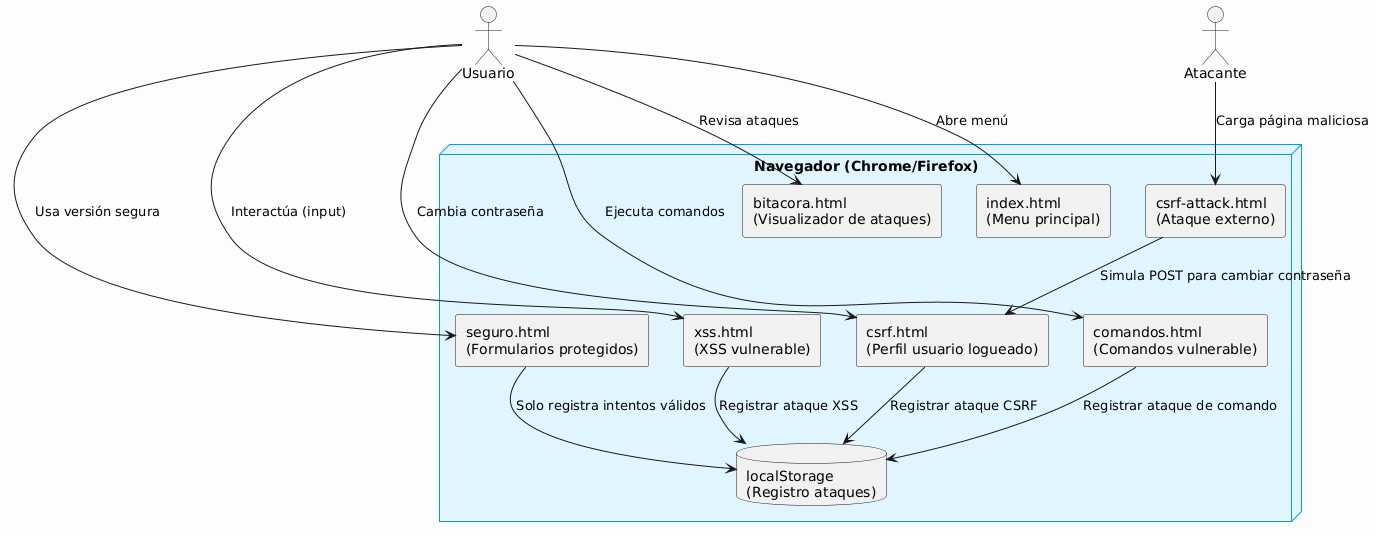












# CONCLUSIÓN

La simulación práctica de vulnerabilidades web mediante tecnologías frontend permitió evidenciar el impacto real que tienen las malas prácticas de desarrollo en la seguridad de los sistemas. A través de la implementación de ataques como Cross-Site Scripting (XSS), inyección de comandos y Cross-Site Request Forgery (CSRF), se demostró cómo una aplicación puede ser fácilmente manipulada cuando no se validan ni filtran adecuadamente las entradas del usuario.

Durante el desarrollo, se construyó una interfaz vulnerable y posteriormente una versión segura, aplicando técnicas como la sanitización de datos, restricciones de comandos válidos y el aislamiento de acciones críticas. Además, la incorporación de una bitácora de ataques permitió registrar de manera persistente los intentos de explotación, proporcionando un valor adicional como evidencia y análisis.

La experiencia no solo fortaleció la comprensión técnica de las vulnerabilidades más comunes en aplicaciones web, sino que también resaltó la importancia de adoptar una cultura preventiva y responsable en el desarrollo de software, incluso desde las capas más superficiales como el frontend.

## Referencias bibliográficas

1. OWASP Foundation. (2021). OWASP Top 10: The Ten Most Critical Web Application Security Risks – 2021. https://owasp.org/Top10/
2. Stuttard, D., & Pinto, M. (2011). The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws (2nd ed.). Wiley.
3. Mozilla Developer Network (MDN). (s.f.). Cross-Site Scripting (XSS). <https://developer.mozilla.org/es/docs/Glossary/Cross-site_scripting>
4. PortSwigger. (s.f.). Cross-Site Request Forgery (CSRF). https://portswigger.net/web- security/csrf
5. Acunetix. (2020). Web Application Security: A Beginner's Guide. https:/[/www.acunetix.com/websitesecurity/web-application-security/](http://www.acunetix.com/websitesecurity/web-application-security/)