**Puesta en producción Segura**

Proyecto web final de ciberseguridad

### 

[**Cross Site Scripting XSS 4**](#_belz8mllwq4)

[Descripción 4](#_l9yrcamet5l)

[Origen 4](#_neiwg1szwj6)

[Criticidad 4](#_2d12l8p3or5u)

[Tipos 4](#_lobbfer9cf2s)

[Modo de Explotación 5](#_w6gw8vbi9tie)

[Reflejado XSS 5](#_qb2atoxrzkox)

[Almacenado XSS 5](#_bg6xjuolfbm7)

[DOM-based XSS 5](#_e1fhrx1xskby)

[Ejemplo 6](#_fkv1twdhg38t)

[Mitigación 6](#_xslnvktbvop0)

[**SQLi 7**](#_angvp0boo30h)

[Descripción 7](#_xyi6ipth9pe7)

[Origen 7](#_l0810opusmty)

[Criticidad 7](#_frwyo121c7oq)

[Tipos 7](#_37e8xk87enig)

[1. Inyección basada en errores 7](#_om3vomm4ek1v)

[2. Inyección ciega (Blind SQL Injection) 8](#_68phpuaaqbq2)

[3. Inyección de unión (Union-based SQL Injection) 8](#_keifyx86daay)

[4. Inyección inferencial (Inferential SQL Injection) 8](#_bvmd7kixsuky)

[Modo de explotación 8](#_o0pdbe72v6er)

[Inyección basada en errores 8](#_i2vgsp8wolj6)

[Inyección ciega 9](#_64hc84d5stl)

[Inyección de unión 9](#_lol482ujbyt9)

[Ejemplo 9](#_i2vgsp8wolj6)

[Mitigación 9](#_dq0mvuqihx5b)

[**Control de acceso débil (broken access control) 11**](#_e47ltu1jenwb)

[Descripción 11](#_e1e7uw3tsxts)

[Origen 11](#_z9bz1813s3rj)

[Criticidad 11](#_g4iefhhrsoj5)

[Modo de explotación 12](#_z6si5qangd4o)

[Ejemplo 12](#_qm4g251d73sb)

[Mitigación 13](#_cmeidvam4kgr)

[**Glosario 14**](#_eca78ca220z9)

[skimming 14](#_11194r23hfmq)

| Cross Site Scripting XSS |
| --- |
| Descripción |
| Cross-Site Scripting (XSS) es una vulnerabilidad de seguridad que se encuentra comúnmente en aplicaciones web. Ocurre cuando una aplicación permite que los datos ingresados por un usuario sean modificados debido a la falta de validación o saneamiento adecuados.  Esta vulnerabilidad permite a los atacantes ejecutar scripts maliciosos en el navegador de un usuario, lo que puede llevar a una variedad de consecuencias, como el robo de cookies, redirección a sitios maliciosos, y ejecución de acciones no autorizadas en el contexto del usuario afectado. |
| Origen |
| El origen de la vulnerabilidad XSS se debe a la falta de validación y saneamiento de los datos de entrada en las aplicaciones web. Esto puede ocurrir en varias partes de una aplicación, tales como formularios, parámetros de URL, y cabeceras HTTP. Los desarrolladores a menudo no filtran o escapan adecuadamente los datos antes de mostrarlos en una página web, lo que permite la inyección de scripts maliciosos. |
| Criticidad |
| La criticidad de una vulnerabilidad XSS puede variar desde baja hasta alta, dependiendo de varios factores como:   * El tipo de XSS (reflejado, almacenado o DOM-based). * El contexto de ejecución del script inyectado. * La capacidad del atacante para atraer a la víctima a una página vulnerada. |
| Tipos |
| 1. **Reflejado (Reflected XSS)**: El script malicioso se refleja en la respuesta del servidor, generalmente a través de una URL o un formulario. Este tipo suele requerir que el atacante engañe a la víctima para que haga clic en un enlace especialmente diseñado. 2. **Almacenado (Stored XSS)**: El script malicioso se almacena en el servidor y se ejecuta cuando un usuario visita la página afectada. Este tipo es más peligroso porque puede afectar a múltiples usuarios sin interacción adicional del atacante. 3. **Basado en DOM (DOM-based XSS)**: El script malicioso se ejecuta en el cliente mediante la manipulación del Document Object Model (DOM) del navegador. No necesariamente involucra interacción con el servidor. |
| Modo de Explotación |
| Reflejado XSS |
| 1. Un atacante crea un enlace malicioso que contiene un script malicioso en los parámetros de la URL. 2. La víctima hace clic en el enlace. 3. El servidor refleja el script malicioso en la respuesta web. 4. El navegador de la víctima ejecuta el script, que puede robar cookies, redirigir a sitios maliciosos, etc. |
| Almacenado XSS |
| 1. Un atacante ingresa un script malicioso en un campo de entrada que se almacena en el servidor (por ejemplo, un comentario en un foro). 2. El script se almacena en la base de datos del servidor. 3. Cuando otro usuario accede a la página que muestra los datos almacenados, el script se ejecuta en el navegador del usuario. |
| DOM-based XSS |
| 1. Un atacante manipula la URL o la entrada de usuario que la aplicación utiliza para modificar el DOM. 2. El navegador de la víctima procesa el contenido malicioso y ejecuta el script. 3. El script puede llevar a cabo acciones maliciosas en el contexto del navegador de la víctima. |
| Ejemplo |
|  |
| Mitigación |
| Para protegerse contra XSS, se deben implementar las siguientes prácticas de seguridad:   1. **Validación y Escapado de Entradas**: Escapar y validar todas las entradas del usuario antes de mostrarlas en una página web. 2. **Content Security Policy (CSP)**: Implementar CSP para restringir las fuentes desde las cuales se pueden cargar scripts. 3. **Codificación de Salida**: Asegurarse de que los datos dinámicos se codifiquen correctamente antes de ser renderizados en HTML, JavaScript, CSS, etc. 4. **Uso de Bibliotecas Seguras**: Utilizar bibliotecas y frameworks que proporcionen mecanismos de protección contra XSS. 5. **Revisiones y Pruebas de Seguridad**: Realizar revisiones de código y pruebas de seguridad regularmente como auditorías para identificar y corregir vulnerabilidades. |

| SQLi |
| --- |
| Descripción |
| La inyección de SQL (SQL Injection, SQLi) es una vulnerabilidad de seguridad que permite a los atacantes interferir con las consultas que una aplicación realiza a su base de datos. Ocurre cuando los datos proporcionados por el usuario se incluyen directamente en una consulta SQL sin la adecuada sanitización o validación. Esto permite al atacante modificar la consulta SQL para ejecutar comandos arbitrarios en la base de datos, acceder a datos no autorizados, manipular o destruir datos, y en algunos casos, escalar privilegios dentro del sistema. |
| Origen |
| El origen de la vulnerabilidad SQL Injection radica en la inclusión directa de datos de entrada del usuario en las consultas SQL sin un adecuado proceso de sanitización. Las aplicaciones web que construyen dinámicamente las consultas SQL a partir de la entrada del usuario son especialmente vulnerables si no utilizan prácticas de codificación seguras, como el uso de sentencias preparadas o consultas parametrizadas. |
| Criticidad |
| La criticidad de una vulnerabilidad de SQL Injection puede variar desde alta hasta crítica, dependiendo de varios factores:   * **Acceso a la Base de Datos**: Si el atacante obtiene acceso completo a la base de datos, puede extraer, modificar o borrar datos sensibles. * **Escalación de Privilegios**: Un atacante puede obtener privilegios elevados dentro de la base de datos o el sistema operativo subyacente. * **Persistencia del Ataque**: La capacidad del atacante para establecer un acceso persistente a través de shell o rootkits. |
| Tipos |
| Inyección basada en errores Se aprovecha de los mensajes de error devueltos por la base de datos para obtener información sobre la estructura de la base de datos y el contenido de sus tablas. Inyección ciega (Blind SQL Injection) El atacante no recibe directamente mensajes de error de la base de datos, pero puede inferir la respuesta observando los comportamientos de la aplicación. Inyección de unión (Union-based SQL Injection) El atacante utiliza la cláusula UNION para combinar los resultados de varias consultas SELECT en una sola respuesta. Inyección inferencial (Inferential SQL Injection) Similar a la ciega, el atacante hace preguntas booleanas (verdadero/falso) a la base de datos, deduciendo la información con base en las respuestas. |
| Modo de explotación |
| Inyección basada en errores  1. Un atacante ingresa una cadena maliciosa en un campo de entrada que se utiliza en una consulta SQL. 2. Si la aplicación devuelve un mensaje de error detallado, el atacante puede usar esta información para entender la estructura de la base de datos y crear inyecciones más complejas.  Inyección ciega  1. El atacante envía inyecciones SQL que modifican el comportamiento de la aplicación (por ejemplo, retrasando la respuesta). 2. La falta de un mensaje de error directo obliga al atacante a inferir la información basándose en los tiempos de respuesta o las diferencias en la salida de la aplicación.  Inyección de unión  1. El atacante utiliza la cláusula UNION para combinar resultados legítimos con datos inyectados. 2. Esto puede permitir la visualización de datos de diferentes tablas en la misma respuesta, revelando información sensible. |
| Ejemplo |
|  |
| Mitigación |
| Para protegerse contra SQL Injection, se deben implementar las siguientes prácticas de seguridad:   * **Consultas Parametrizadas**: Utilizar consultas parametrizadas en lugar de concatenar directamente los datos del usuario en las consultas SQL. |
| $stmt = $pdo->prepare('SELECT \* FROM usuarios WHERE nombre = :nombre AND contraseña = :contraseña');  $stmt->execute(['nombre' => $usuario, 'contraseña' => $contraseña]); |
| * **Uso de ORM (Object-Relational Mapping)**: Utilizar bibliotecas ORM que abstraen y sanitizan las consultas SQL. * **Validación y Saneamiento de Entrada**: Validar y sanear todas las entradas del usuario antes de utilizarlas en consultas SQL. * **Control de Privilegios**: Asegurar que las cuentas de base de datos utilizadas por la aplicación tengan los mínimos privilegios necesarios. * **Mensajes de Error Genéricos**: Evitar mostrar mensajes de error detallados al usuario final. En su lugar, registrar errores de forma segura para su revisión por parte del personal técnico. * **Monitoreo y Auditoría**: Implementar sistemas de monitoreo y auditoría para detectar y responder a posibles intentos de inyección SQL. |

| Control de acceso débil (broken access control) |
| --- |
| Descripción |
| El control de acceso roto (Broken Access Control) es una vulnerabilidad de seguridad que se produce cuando las aplicaciones no implementan correctamente las políticas de control de acceso, permitiendo a los usuarios ejecutar acciones para las que no tienen permiso. Esto puede resultar en el acceso no autorizado a datos, funciones y recursos sensibles de la aplicación, poniendo en riesgo la integridad, confidencialidad y disponibilidad de la información. |
| Origen |
| El origen de la vulnerabilidad de control de acceso roto se debe a una implementación incorrecta o incompleta de las políticas de control de acceso en las aplicaciones. Esto puede incluir la falta de validación adecuada de las credenciales del usuario, configuraciones incorrectas de permisos, o la exposición inadvertida de endpoints que deberían estar restringidos. Algunas causas comunes incluyen:   * Falta de verificación de permisos en el servidor. * Uso de controles de acceso basados en el cliente, como lógica de autorización implementada en JavaScript. * Errores en la configuración de permisos de archivos y directorios. * Uso inapropiado de tokens de autenticación. |
| Criticidad |
| La criticidad de una vulnerabilidad de control de acceso roto puede variar de media a crítica, dependiendo de varios factores:   * **Impacto en los datos**: El acceso no autorizado a datos sensibles, como información personal, financiera o de salud, puede tener graves consecuencias. * **Impacto en la funcionalidad**: Permitir a los usuarios ejecutar acciones no autorizadas, como cambiar configuraciones, eliminar datos, o realizar operaciones administrativas. * Escalación de privilegios: Si un usuario puede obtener privilegios elevados, puede comprometer completamente la aplicación y los datos relacionados. |
| Modo de explotación |
| La vulnerabilidad agrupa gran variedad de escenarios, alguno de ellos pudieran ser:   * **Acceso a Recursos Sin Autenticación**: Un atacante intenta acceder a una URL o recurso que debería estar restringido. * **Modificación de Parámetros de URL**: Un atacante modifica un parámetro de la URL para obtener acceso a datos de otro usuario. * **Controles de Acceso en el Cliente**: Si la lógica de control de acceso está implementada en el lado del cliente (por ejemplo, en JavaScript), un atacante puede modificar el código para saltarse las restricciones. |
| Ejemplo |
| Aplicación web Wordpress que permite por defecto la enumeración de usuarios mediante un parámetro en la url llamado author, por consiguiente, manipulando la url se podrá enumerar a los usuarios que tengan un perfil público habilitado.  [**https://www.unapagina.eu/?author=1**](https://www.unapagina.eu/?author=1)  Nos devolvería:  [**https://www.unapagina.eu/author/admin/**](https://www.unapagina.eu/author/admin/) |
| Mitigación |
| Para protegerse contra vulnerabilidades de control de acceso roto, se deben implementar las siguientes prácticas de seguridad:   1. **Verificación de Autorización en el Servidor**: Asegurarse de que todas las solicitudes de acceso a recursos protegidos se verifiquen en el servidor, independientemente de cualquier verificación realizada en el cliente. 2. **Principio de Menor Privilegio**: Asignar los permisos mínimos necesarios a cada usuario y recurso, evitando otorgar más permisos de los necesarios. 3. **Revisiones y Pruebas de Seguridad**: Realizar revisiones periódicas de las políticas de control de acceso y pruebas de penetración para identificar y corregir posibles fallos. 4. **Control de Acceso Basado en Roles (RBAC)**: Implementar un sistema de control de acceso basado en roles que defina claramente los permisos para cada rol y usuario. 5. **Uso de Tokens Seguros**: Utilizar tokens de autenticación y autorización seguros que se verifiquen en cada solicitud para asegurar que el usuario tiene permiso para realizar la acción solicitada. 6. **Registros y Auditoría**: Mantener registros detallados de accesos y acciones realizadas por los usuarios, y auditar regularmente estos registros para detectar accesos no autorizados. |

| Glosario |
| --- |
| skimming |
| El "skimming" es una técnica de fraude que implica la captura y el uso no autorizado de información de tarjetas de crédito o débito. Este término puede referirse a varias prácticas diferentes, pero todas tienen en común el robo de datos financieros. |