Módulo 2. OWASP Top 10 (visión operativa)

CÓRDOBA CREA FUTURO

1. Inyección, XXE y deserialización

Las vulnerabilidades incluidas en esta unidad comparten un principio estructural común: la manipulación del proceso de interpretación de datos por parte del sistema. En lugar de atacar directamente la infraestructura o el canal de comunicación, estos mecanismos explotan la forma en que la aplicación procesa información recibida del exterior. Cuando los datos no son tratados estrictamente como datos, sino que el sistema les otorga capacidad de influir en su propia lógica interna, se produce una ruptura en la separación entre contenido e instrucción.

La inyección representa el ejemplo paradigmático de esta ruptura. Sin embargo, el mismo patrón se observa en el procesamiento de documentos estructurados, como ocurre en XXE, o en la interpretación de objetos serializados. En todos los casos, el sistema confía en que la información recibida cumple un propósito legítimo, sin verificar adecuadamente su impacto en los mecanismos internos de ejecución o parsing.

Estas vulnerabilidades se desarrollan en capas profundas del sistema: motores de base de datos, parsers XML, mecanismos de deserialización o intérpretes internos. No dependen exclusivamente de la interfaz visible, sino de componentes que operan en segundo plano. Esta característica las convierte en amenazas particularmente críticas, ya que muchas veces no son evidentes desde la superficie de la aplicación.

Desde una perspectiva operativa, este tipo de debilidades permite alterar consultas, acceder a archivos internos, ejecutar código no previsto o manipular estructuras de datos internas. El impacto puede afectar la confidencialidad, integridad y disponibilidad del sistema, dependiendo del contexto y del componente comprometido.

La persistencia histórica de estas vulnerabilidades demuestra que el problema no reside únicamente en la falta de conocimiento técnico, sino en supuestos erróneos sobre la confiabilidad de la entrada externa. Cuando el diseño no contempla explícitamente que todo dato proveniente del usuario es potencialmente malicioso, la aplicación se convierte en un intérprete involuntario de instrucciones ajenas.

En términos de arquitectura, la mitigación exige mecanismos de aislamiento claros: consultas parametrizadas, deshabilitación de entidades externas, validaciones estrictas y controles que impidan la ejecución automática de estructuras no confiables. La separación entre lógica y datos no es una recomendación abstracta, sino una condición estructural de seguridad.

Inyección (SQLi fundamentos y evasiones comunes)

La inyección es una de las vulnerabilidades más persistentes en las aplicaciones web y uno de los puntos centrales que OWASP ha sostenido históricamente como un riesgo crítico. Comprenderla exige abandonar la idea de que se trata únicamente de un error técnico y abordarla como un fenómeno que nace en la frontera entre el lenguaje natural del usuario y el lenguaje formal del sistema. En un entorno donde las aplicaciones procesan constantemente información proveniente del exterior, cada campo de entrada, parámetro de URL o estructura interna que dependa de datos enviados por el usuario se convierte en un espacio potencial para que esa frontera se difumine y los límites entre dato e instrucción se diluyan. Desde esta perspectiva, la inyección no es solamente un problema de validación insuficiente: es, sobre todo, un problema de interpretación errónea sobre la confiabilidad de los datos que ingresan al sistema y el modo en que el servidor decide otorgarles una estructura semántica.

Cuando una aplicación utiliza estos datos para construir consultas o instrucciones sin un mecanismo que garantice su neutralidad, el atacante puede insertar fragmentos que el servidor interpretará como comandos legítimos. OWASP ilustra este fenómeno al mostrar cómo las aplicaciones, en lugar de tratar una entrada como un valor inofensivo, la incorporan dentro de una consulta sin delimitar adecuadamente su rol. El resultado es que la consulta final puede incluir instrucciones completas, generadas no por la

lógica del sistema, sino por la voluntad del atacante. La explotación de esta vulnerabilidad ocurre porque la aplicación no distingue entre aquello que proviene del usuario y aquello que proviene de la lógica interna que define la conducta legítima de la herramienta.

SQL Injection, una de las formas más conocidas de inyección, opera precisamente con este mecanismo. Para comprenderla, es necesario visualizar cómo el sistema construye una consulta con varias capas: estructura, condiciones, parámetros y operadores. Los parámetros deberían ser espacios neutros donde el usuario solo aporta contenido que el sistema utilizará como datos. Sin embargo, en aplicaciones vulnerables, los parámetros se insertan de manera directa dentro de la consulta, permitiendo que valores inesperados —como operadores lógicos, comillas estratégicas o comentarios truncantes— modifiquen la intención original. Una consulta pensada para recuperar un solo registro puede transformarse en una instrucción para mostrar todos los registros, eliminar información o modificar permisos. A nivel organizacional, este tipo de intrusión compromete la confidencialidad, integridad y disponibilidad de los datos, afectando procesos críticos como facturación, historial de clientes, reportes internos o autenticación.

La inyección opera también como un fenómeno que aprovecha el “punto ciego” de la aplicación: aquella suposición implícita de que una instrucción construida a partir de componentes legítimos debe producir un resultado igualmente legítimo. Sin embargo, el atacante explota la manera en que el sistema concatena elementos. La concatenación carece de comprensión semántica, es decir, no distingue entre una instrucción prevista y una que ha sido injertada. En este sentido, la inyección no solo manipula datos, sino que manipula el proceso de interpretación, haciendo que el servidor ejecute comandos que nunca estuvieron contemplados en el diseño inicial.

Un aspecto relevante para comprender la persistencia de esta vulnerabilidad es la facilidad con la que se pueden aplicar técnicas de evasión. Cuando un sistema implementa filtros parciales, estos filtros suelen actuar sobre patrones estáticos: palabras clave, combinaciones de símbolos o caracteres prohibidos. Los atacantes, en cambio, trabajan con variaciones infinitas. Una palabra clave puede codificarse de

distintas maneras, un símbolo puede representarse de forma alternativa y una cadena aparentemente inocua puede desplegar un comportamiento completamente diferente una vez procesada por el servidor. Así, el filtrado superficial se convierte en una barrera frágil que puede ser evitada mediante pequeñas modificaciones en el texto que el atacante provee. Los motores de bases de datos suelen interpretar correctamente estas variaciones, otorgándoles el mismo significado, de manera que la consulta resultante continúa siendo maliciosa.

La inyección no debe de entenderse como un fallo de un único punto, sino como un problema sistémico en el que confluyen decisiones de diseño, suposiciones sobre el comportamiento del usuario y prácticas deficientes en la separación entre datos e instrucciones. En la práctica, una aplicación afectada por inyección puede permitir que un atacante acceda a información sensible, modifique archivos, altere parámetros internos o manipule transacciones que son críticas para la operación cotidiana. Esto implica consecuencias no solo técnicas, sino también estratégicas. Una organización puede perder la trazabilidad de un proceso, exponer datos personales, enfrentar riesgos legales o sufrir interrupciones en servicios esenciales. La inyección es una puerta de entrada hacia un ecosistema completo de vulnerabilidades secundarias, ya que, una vez comprometida la lógica interna del sistema, se crea una ruta para dominar otras capas de la aplicación.

Para observar de manera ordenada cómo se manifiestan estas variantes y cuál es su impacto operativo, se presenta la siguiente tabla comparativa. Esta tabla no busca describirlas en términos exhaustivos, sino ofrecer una lectura conceptual que permita comprender por qué la inyección adopta múltiples formas pero comparte un mismo

fundamento: la incapacidad del sistema para aislar lo que debería ser un dato de lo que el servidor puede llegar a interpretar como una instrucción.

Tabla 1. Comparación operativa de variantes de inyección y su impacto en aplicaciones web

Variante de inyección	Mecanismo operativo	Qué permite el atacante	Impacto en negocio
SQLi clásica	El servidor interpreta datos como instrucciones	Leer, modificar o borrar información	Pérdida de confidencialidad e integridad
SQLi ciega	Respuestas indirectas permiten inferir datos	Cartografiar bases y extraer información paso a paso	Exfiltración lenta pero segura
Inyección en parámetros de ruta	El motor interpreta segmentos de URL como comandos	Manipular consultas internas	Exposición de registros sensibles
Evasiones (comentarios, codificaciones)	Bypass de validaciones superficiales	Ejecutar la instrucción pese al filtrado	Quiebre del control de entrada

Fuente: elaboración propia.

Como puede observarse, la inyección adopta diversos matices, pero todos responden a un principio común: el atacante transforma la aplicación en un intérprete de instrucciones que nunca debió procesar. Este fenómeno puede comprenderse mejor si se reconoce que las aplicaciones modernas no están diseñadas para “entender” la entrada del usuario, sino para procesarla mecánicamente. Cuando esa entrada ingresa

sin una neutralización adecuada, la aplicación se convierte involuntariamente en ejecutora de acciones elegidas por un tercero.

En síntesis, la inyección no es solo una vulnerabilidad histórica, sino un recordatorio permanente de la importancia de diseñar mecanismos que separen con claridad la lógica del sistema de los datos que el usuario provee. La persistencia de esta amenaza se debe a que aún hoy existen aplicaciones que confían en estructuras frágiles de concatenación, filtrado limitado y supuestos sobre los que se construyeron sistemas que no contemplaban el panorama de riesgo actual. Comprender este fenómeno en su dimensión operativa es fundamental para evaluar adecuadamente el nivel de exposición de una organización y anticipar medidas de protección que reduzcan la superficie de ataque.

XML External Entity (XXE): fundamentos y riesgos operativos

Las aplicaciones que procesan archivos o documentos estructurados en formato XML dependen de un componente fundamental para interpretar su contenido: el parser. Este parser cumple una función aparentemente simple, que es leer el documento, identificar sus elementos y construir una representación interna que la aplicación pueda utilizar. Sin embargo, esta operación, que en principio se percibe como una tarea mecánica, puede convertirse en un punto de vulnerabilidad cuando el parser admite el uso de entidades externas. Para comprender el fenómeno de XXE, es necesario reconocer que XML no es solo un lenguaje de marcado, sino también un mecanismo que permite definir referencias que apuntan a recursos externos al documento. La inclusión de estas entidades se diseñó originalmente para facilitar el manejo de grandes estructuras o para permitir la reutilización modular de contenido, pero en el contexto de aplicaciones web se transforma en una puerta de acceso involuntaria a información interna del sistema.

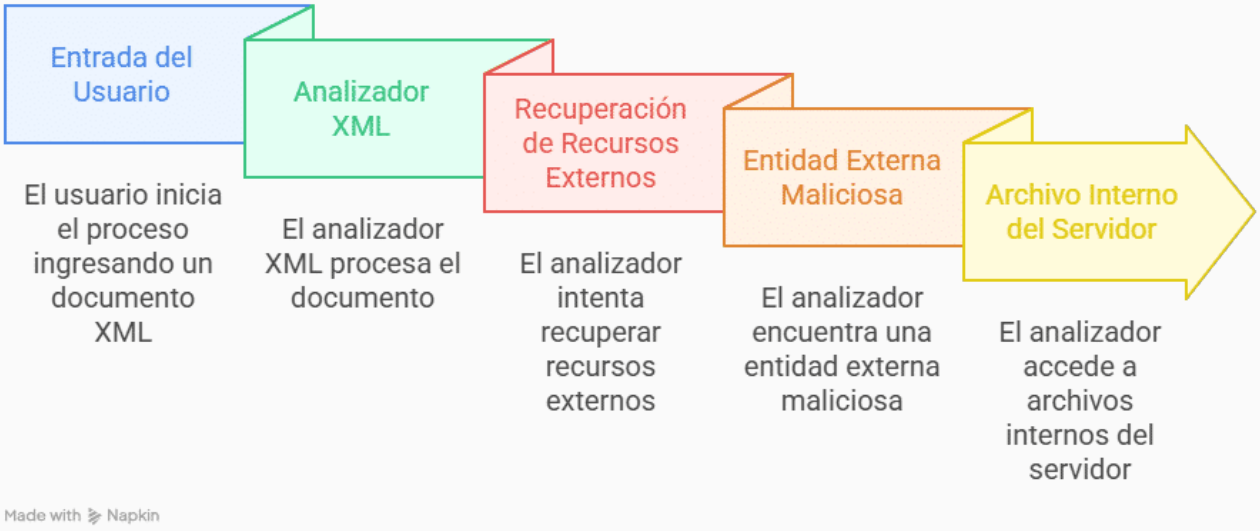
Una entidad externa funciona como una instrucción dentro del documento XML. En lugar de contener contenido directo, la entidad indica que el valor debe obtenerse desde otra ubicación, ya sea un archivo local del servidor o un recurso accesible mediante una URL. Si el parser está configurado para resolver estas entidades —y

muchos lo están por defecto en bibliotecas antiguas o en entornos heredados—, la aplicación termina solicitando un recurso que el usuario define. De esta manera, el atacante no necesita vulnerar la lógica de negocio ni explotar fallos visibles en la interfaz. Le basta con enviar un archivo XML construido de tal forma que, al ser procesado por el servidor, desencadene una consulta interna hacia un recurso que no debió ser expuesto. Este mecanismo convierte a XXE en una vulnerabilidad silenciosa: el ataque se produce antes de que la aplicación siquiera procese los datos del usuario en su lógica principal, ocurriendo íntegramente dentro del motor de *parsing*.

Para entender con claridad el funcionamiento de esta vulnerabilidad, conviene analizar cómo el parser sigue paso a paso las instrucciones del documento. Al recibir el archivo, el parser identifica la presencia de un DTD (Document Type Definition) o conjunto de reglas que indica la existencia de una entidad externa. Mientras la aplicación interpreta el documento como si fuera un simple conjunto de etiquetas, el parser ejecuta la lógica de expansión de la entidad, solicitando el recurso indicado por el atacante. Esta operación puede conducir a la lectura de archivos internos —por ejemplo, registros del sistema, claves privadas, configuraciones del entorno— o incluso a realizar solicitudes a servicios internos que no deberían ser accesibles desde el exterior. Todo esto ocurre sin que la aplicación, en su nivel superior, tenga registro de que se ha producido una acción adicional que no forma parte del flujo esperado.

Con el fin de visualizar este proceso, resulta útil observar el siguiente diagrama conceptual, que sintetiza el flujo que sigue un parser vulnerable al procesar un documento malicioso. El objetivo de esta imagen no es mostrar la sintaxis técnica, sino capturar la estructura operativa del ataque: un documento XML modificado, un parser que confía en sus instrucciones y un recurso interno al que no debería accederse.

Figura 1. Proceso de Ataque de Entidad Externa XML



Fuente: elaboración propia.

La potencia del ataque XXE radica en que aprovecha un comportamiento legítimo del parser. Desde su diseño original, XML permitió definir entidades externas como una forma de modularidad. El problema aparece cuando se traslada este diseño a escenarios donde los documentos provienen de usuarios no confiables. En una aplicación moderna, cualquier archivo cargado —como formularios, reportes, plantillas o configuraciones— puede convertirse en un vehículo para activar este mecanismo. La vulnerabilidad se intensifica cuando el servidor posee archivos sensibles cuyo contenido no debería salir nunca de su entorno. Si el parser, siguiendo las instrucciones del documento, permite acceder a estos archivos, la vulnerabilidad deja de ser un problema de interpretación para transformarse en una exposición directa de información crítica.

Las consecuencias de un ataque XXE pueden adoptar diversas formas. En su expresión más básica, permiten a un atacante leer archivos a los que un usuario externo nunca debería tener acceso. Sin embargo, su impacto puede ir más allá: en algunos casos, es posible desencadenar solicitudes internas hacia servicios que se encuentran detrás del

firewall, explorando la infraestructura mediante Server-Side Request Forgery (SSRF). Este es un ejemplo de cómo las vulnerabilidades no se presentan de forma aislada, sino que se conectan entre sí. Una explotación XXE puede servir como punto de partida para ataques más avanzados que comprometan sistemas completos o permitan al atacante realizar movimientos laterales dentro de la red de la organización.

El riesgo de XXE no se limita a la lectura de información. Dependiendo de las características del parser y del entorno en que opera, la expansión de entidades externas puede utilizarse también para producir denegaciones de servicio. Una entidad especialmente construida puede obligar al servidor a procesar estructuras recursivas o demasiado grandes, consumiendo memoria, tiempo de CPU o saturando recursos internos. Aunque este efecto es menos frecuente que el acceso a archivos, sigue siendo una amenaza relevante, sobre todo en aplicaciones que manejan grandes volúmenes de documentos o que operan en servidores compartidos donde los recursos son limitados.

La prevención de XXE exige un cambio en la manera en que las aplicaciones procesan XML. OWASP enfatiza la necesidad de deshabilitar de manera explícita la resolución de entidades externas en todas las bibliotecas y frameworks que las utilicen. Esta medida, que a simple vista parece simple, requiere un conocimiento profundo de las herramientas empleadas en la aplicación. Diferentes lenguajes y librerías poseen configuraciones predeterminadas distintas, y algunas versiones antiguas mantienen comportamientos que son inseguros por defecto. La mitigación también implica considerar la necesidad real de utilizar XML en contextos donde alternativas más seguras, como JSON, podrían cubrir la misma función sin incorporar los riesgos asociados al modelo de entidades externas.

En última instancia, la vulnerabilidad XXE nos recuerda que las aplicaciones suelen confiar en componentes subyacentes cuyo comportamiento no siempre es visible desde la lógica de negocio. El parser opera en un nivel profundo, previo al procesamiento funcional del documento. Cuando una organización evalúa su superficie de ataque, a menudo se concentra en la capa lógica, dejando de lado estos mecanismos internos que pueden convertirse en vectores críticos de exposición.

Comprender XXE es comprender que la seguridad no depende solo de las decisiones visibles, sino también de los mecanismos implícitos que permiten que los sistemas interpreten el mundo exterior.

CONTINUAR

2. XSS, CSRF, IDOR y configuración

Las vulnerabilidades agrupadas en esta unidad se desarrollan en la interacción entre usuario, navegador y servidor. A diferencia de las debilidades asociadas a motores internos o parsers, aquí el punto crítico reside en el modelo de confianza que sostiene la comunicación web. El navegador actúa como intermediario activo, ejecutando scripts, enviando solicitudes y gestionando sesiones en nombre del usuario.

XSS y CSRF explotan precisamente esa relación de confianza. En el primer caso, el navegador interpreta contenido malicioso como parte legítima del sitio; en el segundo, envía solicitudes válidas sin intervención consciente del usuario. Ambos mecanismos demuestran que la seguridad no depende solo del servidor, sino también de cómo el navegador procesa y transmite información.

Por su parte, IDOR revela fallas en la lógica de autorización. El problema no se encuentra en la autenticación del usuario, sino en la ausencia de verificación explícita de permisos al acceder a recursos internos. Esta debilidad pone en evidencia que la identidad por sí sola no garantiza control de acceso adecuado.

Los errores de configuración completan el panorama mostrando que la seguridad no se limita al código de la aplicación. La infraestructura, los encabezados HTTP, los permisos de archivos y las configuraciones por defecto influyen directamente en la superficie de exposición. Un sistema correctamente programado puede resultar vulnerable si su entorno operativo no está adecuadamente endurecido.

Estas categorías comparten un elemento transversal: la confianza implícita. Confianza en que el navegador ejecutará solo contenido legítimo, en que el usuario no modificará identificadores internos, en que la configuración por defecto es adecuada. Cuando

estas suposiciones no se revisan críticamente, emergen vulnerabilidades que pueden explotarse con relativa simplicidad.

Desde el punto de vista arquitectónico, la mitigación exige controles en múltiples niveles: sanitización de entradas, políticas de seguridad del navegador, verificación sistemática de autorizaciones y revisión periódica de configuraciones. La protección no se concentra en un único punto, sino que se distribuye en toda la cadena de interacción.

En síntesis, esta unidad analiza vulnerabilidades que afectan la capa visible de la aplicación y su entorno inmediato. Comprenderlas permite reconocer que la seguridad web no depende únicamente de algoritmos o consultas, sino de cómo se gestiona la confianza en cada interacción entre sistema y usuario.

Cross-Site Scripting (XSS) y Falsificación de Solicitud entre Sitios (CSRF)

La interacción entre los usuarios y las aplicaciones web se sostiene en un supuesto fundamental: que el navegador actúa como mediador confiable entre ambos. Este rol del navegador, que consiste en interpretar instrucciones, organizar contenido y ejecutar ciertas acciones en representación del usuario, es indispensable para el funcionamiento de la Web. Sin embargo, su capacidad para ejecutar contenido dinámico y mantener sesiones activas también lo convierte en un blanco estratégico para los atacantes. Tanto XSS como CSRF son vulnerabilidades que se desarrollan en este espacio intermedio, donde la lógica del servidor y la actividad del usuario convergen en un entorno que, por diseño, otorga al navegador permisos amplios para interpretar, mostrar y transmitir información. Comprender estas vulnerabilidades exige observar la manera en que el navegador administra contenido, mantiene estados y toma decisiones sin intervención directa del usuario.

El Cross-Site Scripting, o XSS, es una vulnerabilidad que surge cuando una aplicación permite que contenido controlado por un atacante sea interpretado por el navegador como si fuera parte del contenido legítimo del sitio. La clave de este fenómeno reside en la ausencia de neutralización adecuada de los datos que la aplicación presenta. Cuando un sistema inserta contenido enviado por el usuario dentro de una página web sin sanitizarlo, el navegador puede ejecutar ese contenido como si fuese un fragmento legítimo de la interfaz. Esto significa que el atacante puede influir en el comportamiento del navegador: puede inyectar scripts que manipulen el contenido visible, capturen información sensible, modifiquen la interfaz o accedan a elementos internos del documento. La vulnerabilidad opera no en el servidor, sino en la relación entre servidor y navegador, donde el navegador actúa obedientemente ante cualquier instrucción que perciba como parte del documento recibido.

XSS puede adoptar diferentes formas, entre las cuales se destacan el XSS reflejado y el XSS almacenado. En el primero, el atacante induce al usuario a visitar una URL especialmente construida, de modo que los datos enviados en la solicitud vuelven inmediatamente en la respuesta, sin pasar por procesos de validación o neutralización. El navegador recibe estos datos incrustados en el documento y les otorga interpretación activa. En el segundo caso, el contenido malicioso no se devuelve de inmediato, sino que queda almacenado en el sistema —por ejemplo, en un comentario, un perfil, un mensaje o un registro— y se activa cada vez que un usuario visualiza la sección afectada. Ambas modalidades comparten un mismo principio: la aplicación no distingue entre contenido inocuo y contenido con intención de modificar el comportamiento del navegador. Así, un espacio destinado a texto termina funcionando como un vector para ejecutar instrucciones arbitrarias.

Las consecuencias de un ataque XSS son especialmente relevantes debido a la manera en que las aplicaciones modernas gestionan la identidad del usuario. Muchas aplicaciones basan la autenticación en sesiones que se mantienen activas mediante *cookies*. Si un *script* inyectado puede leer estas *cookies* o enviar solicitudes en el contexto del usuario, las implicancias son profundas: robo de sesión, acciones fraudulentas, manipulación del perfil, acceso a información privada o incluso desfiguración del sitio. En este sentido, XSS se convierte en un puente que permite al

atacante realizar acciones que, desde la perspectiva del servidor, parecen provenir de un usuario legítimo. La vulnerabilidad reside en el modelo mismo de confianza implícita entre el navegador y la aplicación.

Si XSS explota la capacidad del navegador para interpretar contenido activo, CSRF explota la capacidad del navegador para enviar solicitudes de manera automática en nombre del usuario. La falsificación de solicitud entre sitios, o Cross-Site Request Forgery, se basa en un principio operativo diferente: el navegador envía información al servidor, incluyendo *cookies* de sesión, cada vez que se realiza una solicitud a un dominio determinado. Cuando el usuario está autenticado y su navegador mantiene una sesión activa, cualquier solicitud dirigida al servidor queda asociada automáticamente a su identidad, aun cuando la solicitud no haya sido iniciada conscientemente por él.

El atacante aprovecha este comportamiento mediante la creación de un enlace, imagen o formulario que, al ser cargado o ejecutado por el navegador del usuario, envía una solicitud válida al servidor vulnerable. El navegador, siguiendo sus reglas internas, adjunta las *cookies* de sesión del usuario y el servidor interpreta la solicitud como una acción legítima. Así, el atacante puede desencadenar operaciones que el usuario nunca autorizó: transferencias, cambios de contraseña, publicación de información o modificaciones de configuración. Lo notable es que el ataque ocurre sin que el usuario realice ninguna acción explícita, más allá de cargar contenido externo como una página o un correo.

La relación entre XSS y CSRF es estrecha, porque ambos ataques dependen del modelo de confianza entre navegador y servidor. En el caso de XSS, el navegador ejecuta instrucciones maliciosas; en el de CSRF, el navegador envía solicitudes involuntarias pero válidas. Esta relación puede visualizarse con claridad comparando ambos mecanismos, lo cual permite entender cómo cada uno afecta distintas partes de la interacción entre usuario, navegador y servidor. Con este propósito, se introduce la siguiente tabla conceptual, que sintetiza sus diferencias y vínculos operativos.

Tabla 2. Relación operativa entre XSS y CSRF: diferencias, vínculos y modos de explotación

Aspecto	XSS	CSRF
Qué explota	Interpretación de contenido malicioso	Confianza del servidor en la sesión
Requisito para el atacante	Inyectar contenido en el navegador	Lograr que el usuario ejecute una acción
Consecuencia típica	Robo de sesión, manipulación visual	Transferencias, cambios de perfil, operaciones
Prevención	Escaping, sanitización, CSP	Tokens, verificación de intención, SameSite

Fuente: elaboración propia.

La tabla permite observar que, aunque ambos ataques se dirigen al navegador, lo hacen desde perspectivas distintas. XSS introduce contenido que altera el comportamiento del navegador; CSRF induce al navegador a actuar sin intervención del usuario. Ambos fenómenos surgen porque el navegador opera como un puente entre dos mundos: por un lado, la lógica del sitio que el usuario visita; por otro, los contenidos externos que puede llegar a cargar desde distintos orígenes.

Para mitigar estos riesgos, OWASP subraya la importancia de combinar políticas de neutralización de contenido, controles de origen y mecanismos de verificación de intención. En el caso de XSS, la sanitización rigurosa de entradas y la aplicación de políticas de seguridad como Content Security Policy resultan esenciales. En el caso de CSRF, los tokens antifraude que incorporan valores únicos y no reutilizables constituyen la medida más efectiva para asegurar que cada acción sea iniciada voluntariamente por el usuario. Sin embargo, estas medidas no pueden considerarse de manera aislada; deben integrarse dentro de una arquitectura de seguridad que reconozca que el

navegador es, al mismo tiempo, una herramienta indispensable y una superficie potencial de ataque.

En términos organizacionales, tanto XSS como CSRF representan amenazas que comprometen la confianza en la interacción digital. Una aplicación afectada por estas vulnerabilidades puede perder control sobre las operaciones que ejecuta y sobre la identidad de quienes las realizan. Esto implica riesgos para la integridad de los datos, para la reputación institucional y para la continuidad de los procesos. La comprensión detallada de estos mecanismos no solo permite detectar fallas, sino también anticipar escenarios en los que la interacción entre usuario, navegador y sistema se convierte en un espacio sensible que debe protegerse con rigor.

IDOR (*Insecure Direct Object Reference*)

El control de acceso es uno de los pilares fundamentales en la seguridad de cualquier aplicación, ya que define qué usuario puede acceder a qué información y qué operaciones puede realizar. En la práctica, este control se implementa mediante una combinación de verificaciones en el servidor, reglas de negocio, estructuras de permisos y restricciones asociadas a cada recurso. Sin embargo, muchas aplicaciones, especialmente aquellas desarrolladas con rapidez o que evolucionaron con el tiempo sin una planificación adecuada, delegan parte de estas restricciones en la suposición implícita de que los identificadores utilizados para acceder a los recursos no serán manipulados por los usuarios. Esta suposición genera una de las vulnerabilidades más comunes y, a la vez, más subestimadas: el Insecure Direct Object Reference, o IDOR.

IDOR ocurre cuando una aplicación permite acceder directamente a un recurso utilizando un identificador —un número, un código, una clave— sin verificar que el usuario tenga permiso para hacerlo. El sistema confía en que el usuario solo manipulará aquellos identificadores que le pertenecen y no contempla la posibilidad de que alguien intente modificar estos valores. La vulnerabilidad surge porque la aplicación expone rutas o mecanismos que hacen referencia a objetos internos, tales como archivos, reportes, registros, formularios o perfiles, utilizando un identificador que la aplicación no protege de manera adecuada. El atacante puede cambiar este valor y

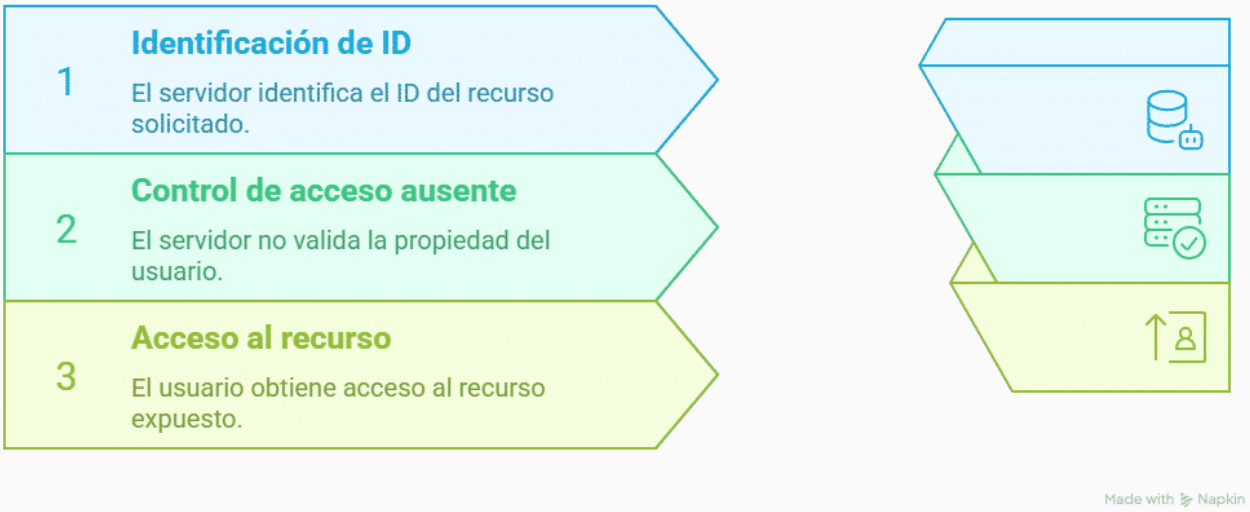
acceder a recursos que no debería ver, revelando información sensible o incluso modificándola.

Una de las razones por las que IDOR persiste es que, en apariencia, se trata de un funcionamiento normal del sistema. Los desarrolladores acostumbran a utilizar identificadores secuenciales o predecibles para identificar elementos dentro de la base de datos; estas decisiones, útiles a nivel interno, pueden convertirse en un problema cuando se exponen directamente a los usuarios. Lo que para la aplicación es un mecanismo simple de organización, para un atacante es una oportunidad: basta con sustituir un identificador por otro para acceder a un recurso distinto. En muchos casos, no se requiere conocimiento técnico ni herramientas avanzadas; basta con modificar un número en la URL para comprometer datos que la organización consideraba protegidos.

El impacto de un IDOR puede ser profundo. Un atacante podría visualizar informes financieros, historiales médicos, registros académicos, perfiles de clientes, comprobantes de pago o cualquier otro tipo de información asociada a un identificador. La vulnerabilidad es especialmente grave en entornos donde los recursos representan operaciones sensibles: aprobaciones, solicitudes de cambio, órdenes de compra o procesos internos. Si el sistema permite modificar estos recursos sin verificar la identidad o los permisos del usuario, el atacante puede alterar operaciones críticas, generando consecuencias operativas y legales. En este sentido, IDOR no es solo una falla técnica, sino una falla en la lógica institucional de control de acceso.

La comprensión de este mecanismo se facilita al observar el patrón conceptual que lo caracteriza: un usuario solicita un recurso utilizando un identificador; la aplicación recibe la solicitud y, sin verificar la legitimidad del usuario, entrega el recurso. Para ilustrar este proceso, se propone el siguiente esquema visual, que muestra la estructura del intercambio y el punto exacto donde se produce la falla. Este diagrama representa la simplicidad aparente del problema, pero también su profundidad, ya que expone cómo decisiones mínimas en la construcción de la interfaz pueden producir brechas significativas.

Figura 2. Flujo de acceso a objetos en aplicación web



Fuente: elaboración propia.

La claridad con la que puede modelarse el flujo de un IDOR refleja también la facilidad con la que puede producirse en sistemas reales. La vulnerabilidad se vuelve más frecuente cuando la aplicación fue diseñada inicialmente para un contexto de uso limitado —por ejemplo, un sistema interno— y luego se reconvirtió en una plataforma accesible a múltiples usuarios con niveles de permisos distintos. En estos escenarios, las decisiones originales de diseño no se ajustan al nuevo modelo de riesgo, y la exposición se incrementa de manera invisible. Este fenómeno también se presenta en aplicaciones que utilizan APIs sin autenticación o sin verificaciones internas adecuadas, ya que las rutas expuestas pueden manipularse desde herramientas externas sin pasar por validaciones de interfaz.

Errores de configuración

Los errores de configuración, por su parte, representan un tipo de vulnerabilidad que refleja cómo, más allá de la lógica de la aplicación, el entorno donde esta se ejecuta constituye un componente crítico de seguridad. OWASP identifica la configuración incorrecta como uno de los riesgos más frecuentes en sistemas modernos. Estos errores pueden incluir permisos demasiado amplios, servicios innecesarios habilitados, componentes desactualizados, encabezados de seguridad ausentes, configuraciones por defecto o valores heredados de entornos de prueba. La crítica reside en que estas configuraciones suelen darse por sentadas y no se revisan a medida que la aplicación evoluciona. Los sistemas crecen, se integran con nuevos servicios, migran a infraestructuras diferentes y adoptan frameworks que traen sus propias configuraciones predeterminadas. En este proceso continuo, los puntos vulnerables se acumulan de manera silenciosa.

La interacción entre IDOR y errores de configuración es particularmente significativa. Una aplicación podría tener una política de acceso sólida a nivel conceptual, pero fallar en la aplicación concreta de verificaciones internas. Podría también implementar controles de acceso estrictos, pero ejecutar la aplicación en un servidor que permite exploración de directorios, que expone información del stack o que utiliza versiones vulnerables de servicios esenciales. Estos fallos, que parecen pequeños, se combinan para generar un ecosistema de riesgo que habilita accesos indebidos, manipulaciones no autorizadas o exposición de datos. Un IDOR, por ejemplo, puede convertirse en una brecha mayor si el servidor expone rutas internas o proporciona mensajes de error detallados que facilitan el reconocimiento del entorno y permiten al atacante ajustar sus intentos.

Comprender los errores de configuración implica reconocer que la seguridad no depende solo de decisiones visibles, como formularios, rutas o botones, sino también de los componentes invisibles: servidores, contenedores, proxies, frameworks, librerías y permisos de archivos. Cada uno de estos elementos puede convertirse en un punto de entrada si no se administra adecuadamente. En entornos de infraestructura moderna, donde los componentes se actualizan y reemplazan de manera continua, la revisión periódica de configuraciones se vuelve indispensable. El principio fundamental para mitigar estos riesgos radica en adoptar una postura que considere la configuración

como parte integral del proceso de diseño, y no como una etapa residual posterior al desarrollo.

En suma, tanto IDOR como los errores de configuración revelan que la seguridad de una aplicación depende menos de la complejidad de su código y más de la solidez de sus decisiones estructurales. La vulnerabilidad emerge cuando se confía en supuestos implícitos que no contemplan el comportamiento creativo o malintencionado de un atacante. La revisión constante de los controles de acceso y de las configuraciones del entorno permite reducir la superficie de exposición y anticipar escenarios críticos. Estas prácticas no solo protegen datos sensibles, sino que aseguran la continuidad de las operaciones y fortalecen la confianza en los sistemas que una organización utiliza como parte de su actividad cotidiana.

CONTINUAR

Referencias

OWASP. (2021). *Cross-Site Request Forgery (CSRF).* https://owasp.org/www-community/attacks/csrf

OWASP. (2021). *Cross-Site Scripting (XSS).* https://owasp.org/www-community/attacks/xss/

OWASP. (2021). *CSRF Prevention Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

OWASP. (2021). *Deserialization Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

OWASP. (2021). *OWASP Top 10 – Official Documentation.* https://owasp.org/Top10/

OWASP. (2021). *OWASP Top 10 – Security Misconfiguration.* https://owasp.org/www-project-top-ten/

OWASP. (2021). *OWASP WebGoat Project.* https://owasp.org/www-project-webgoat/

OWASP. (2021). *Secure Configuration Guide.* https://cheatsheetseries.owasp.org/

OWASP. (2021). *SQL Injection.* https://owasp.org/www-community/attacks/SQL_Injection

OWASP. (2021). *Web Security Testing Guide – Testing for Injection.* https://owasp.org/www-project-web-security-testing-guide/

OWASP. (2021). *XSS Prevention Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

OWASP. (2021). *XML External Entity (XXE) Processing*. https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing

OWASP. (2021). *XML External Entity Prevention Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

PortSwigger. (2021). *Bypassing filters (SQL Injection cheat sheet)*. https://portswigger.net/web-security/sql-injection/cheat-sheet

PortSwigger. (2021). *Cross-Site Request Forgery (CSRF)*. https://portswigger.net/web-security/csrf

PortSwigger. (2021). *IDOR (Insecure Direct Object Reference)*. https://portswigger.net/web-security/access-control/idor

PortSwigger. (2021). *Insecure Deserialization*. https://portswigger.net/web-security/deserialization

PortSwigger. (2021). *SQL Injection*. https://portswigger.net/web-security/sql-injection

PortSwigger. (2021). *XSS (Cross-Site Scripting)*. https://portswigger.net/web-security/cross-site-scripting

PortSwigger. (2021). *XXE Injection*. https://portswigger.net/web-security/xxe

CONTINUAR