Seguridad Web

Seguridad para servidores web

- ¿Porqué necesito un WAF?
- ¿Qué es un ataque de inyección SQL y cómo protegerte de él?
- Ataques XSS o cross-site scripting
- Ataques de CORS o Cross-Origin Resource Sharing
- Ataques de CSRF o Cross-Site Request Forgery
- Instalar Modsecurity como WAF

¿Porqué necesito un WAF?

Necesidad de un WAF

Un Web Application Firewall (WAF) es una parte crucial de la seguridad de tu servidor web, y ¿porqué es importante tener un WAF en tu web?

La respuesta es sencilla: **seguridad.** Para explicar un poco mejor los motivos, te desglosamos las principales funciones de un WAF

Detectar y bloquear ataques web comunes

Un WAF puede detectar y bloquear una amplia gama de ataques web, como inyecciones SQL, ataques de cross-site scripting (XSS), ataques de Cross-Origin Resource Sharing (CORS), ataques de denegación de servicio (DoS), entre otros.

Un WAF permite filtrar tráfico malicioso

Puede filtrar el tráfico malicioso antes de que llegue a tu aplicación web, lo que ayuda a prevenir intrusiones y proteger tus datos sensibles.

Cumplimiento normativo

Para ciertas normativas, regulaciones y estándares de seguridad, como por ejemplo el cumplimiento de PCI DSS (Payment Card Industry Data Security Standard) o HIPAA (Health Insurance Portability and Accountability Act) es necesario disponer de esta capa de protección para las webs

Mitigar la explotación de vulnerabilidades

Aunque es importante tener medidas de seguridad en el propio código en el que está desarrollada una web, un WAF añade cierta protección para impedir que ciertas vulnerabilidades del propio lenguaje de programación o de la forma en la que está construida la web (código) puedan ser

explotadas fácilmente.

Monitorización

Un WAF puede proporcionar registros detallados de tráfico web relativos a eventos de posibles ataques, lo que te permite monitorizar y analizar las actividades maliciosas o sospechosas en tu web y de esa forma poder filtrar bien mediante bloqueo de direcciones IP o incluso bloquear ciertas URL mientras se parchea el intento de ataque si es una vulnerabilidad del lenguaje o del código.

Filtrado de Bots

Los bots son procesos que escanean nuestra web en busca de urls bien para indexación, o bien con motivos maliciosos.

Los WAF pueden detectar y bloquear el tráfico generado por los bots maliciosos, como botnets que intentan comprometer tu aplicación web o scrapers que intentan robar tu contenido, o simplemente no queremos que nuestro servidor se sature por peticiones de bots a direcciones URL aleatorias.

¿Qué es un ataque de inyección SQL y cómo protegerte de él?

En un servidor web, la seguridad es muy importante. Uno de los peligros más críticos que afectan a los servidores y a las aplicaciones web es el llamado "ataque de inyección SQL", una táctica utilizada por hackers para comprometer sistemas y acceder a información confidencial.

¿Qué es la inyección SQL?

Imagina que una aplicación web es como una puerta de entrada a una base de datos donde se almacena información valiosa como datos de clientes, facturas, etc. Normalmente, esta puerta está protegida por una serie de cerraduras, pero si un hacker encuentra una forma de "inyectar" código malicioso a través de un formulario de búsqueda o un campo de inicio de sesión, puede "romper" esta cerradura, abrir la puerta y acceder a datos sensibles.

Cómo Funciona

El hacker identifica un campo de entrada en la aplicación web, como un cuadro de búsqueda. Luego, introduce datos manipulados, como un fragmento de código SQL, en ese campo. Cuando la aplicación procesa esta entrada, en lugar de simplemente buscar lo que el usuario solicitó, o devolver un código de error, puede ejecutar también un código malicioso, que puede dar al hacker acceso no autorizado a la base de datos.

Ejemplos

Supongamos que tienes un sitio web WordPress con un formulario de búsqueda que permite a los usuarios buscar publicaciones por título. El campo de búsqueda envía una consulta SQL a la base de datos para recuperar las publicaciones que coincidan con el término de búsqueda.

Ahora, un atacante malintencionado podría intentar explotar una posible vulnerabilidad de inyección SQL en este formulario de búsqueda. Por ejemplo, el atacante podría ingresar lo siguiente en el campo de búsqueda:

```
'; DROP TABLE wp_users; --
```

Cuando se envía esta cadena como consulta de búsqueda, la consulta SQL resultante podría ser algo así:

```
SELECT * FROM wp_posts WHERE post_title LIKE '%'; DROP TABLE wp_users; --%'
```

Aquí está el análisis de cómo funciona esta cadena:

- `';`: Cierra la cadena de búsqueda actual.
- `DROP TABLE wp_users;`: Agrega una nueva consulta SQL maliciosa para eliminar la tabla de usuarios de la base de datos.
- `--`: Comentario en SQL para ignorar el resto de la consulta original y cualquier consulta adicional.

Si la aplicación WordPress no está protegida adecuadamente contra la inyección SQL, ejecutaría esta consulta sin validarla correctamente. Como resultado, la tabla de usuarios `wp_users` se eliminaría de la base de datos, lo que eliminaría todos los usuarios del sitio, esto como puedes suponer sería catastrófico en términos de seguridad y funcionalidad del sitio, sobre todo si en es Wordpress tenemos por ejemplo un WooCommerce con cientos de clientes de nuestro e-commerce almacenado en la tabla de usuarios.

Los Riesgos

Un ataque de inyección SQL que tenga acceso a los datos de nuestra base de datos, puede tener consecuencias graves. Los hackers pueden obtener información delicada, como nombres de usuario, contraseñas, números de tarjetas de crédito o datos personales. Además, podrían manipular o eliminar datos, causando estragos en la integridad de los datos almacenados en la base de datos de la aplicación con los peligros que ello conlleva amén de las sanciones económicas que pueden derivarse del acceso no autorizado a datos confidenciales.

Cómo Protegerte

Una de las mejores formas, es establecer una serie de criterios de buenas prácticas para el desarrollo de aplicaciones web para protegerse contra los ataques de inyección SQL:

1. **Validación de Entradas**: Cuando se desarrolla cualquier aplicación web, los programadores deben implementar medidas para asegurarse de que los datos que los usuarios introducen sean seguros y estén correctamente formateados y validados antes de procesarlos.
2. **Consultas Parametrizadas**: En lugar de concatenar cadenas de texto directamente en las consultas SQL, los programadore pueden utilizar consultas parametrizadas, que separan los datos de las instrucciones SQL, haciendo más difícil para los hackers inyectar código malicioso.

3. **Actualizaciones y Parches**: Mantén tus aplicaciones web y sistemas actualizados con los últimos parches de seguridad para mitigar las vulnerabilidades conocidas.
4. **Auditorías de Seguridad**: Realiza auditorías de seguridad regulares para identificar y corregir posibles vulnerabilidades antes de que los hackers las exploren.

¿Y si no puedo? WAF es tu amigo

En muchos casos lo que tenemos es una aplicación desarrollada por terceros como pueden ser Wordpress, Joomla, Prestashop, Moodle, etc.

En estos casos, debido a que no podemos modificar estos desarrollos, no hay otra forma para realizar esto que es contar con un WAF que permita realizar el filtrado de estos tipos de ataques en la mayor medida posible.

Protección de ataques de inyección SQL mediante WAF

1. Un WAF puede inspeccionar el tráfico entrante a tu sitio o a tu aplicación web y detectar patrones de entrada que podrían indicar un intento de inyección SQL. Puede examinar los parámetros de las solicitudes HTTP en busca de cadenas de texto sospechosas que coincidan con los patrones utilizados en ataques de inyección SQL.
2. El WAF puede aplicar reglas específicas para validar y limpiar los datos de entrada antes de pasarlos a tu aplicación. Esto incluye la eliminación o el escape de caracteres especiales que podrían ser utilizados en un ataque de inyección SQL. Por ejemplo, puede bloquear consultas SQL que contengan palabras clave como "SELECT" o "DROP TABLE", o puede escapar automáticamente comillas y otros caracteres que podrían ser parte de un ataque.
3. Los WAFs a menudo incluyen una base de datos de firmas y reglas específicas, que identifican patrones de tráfico malicioso conocidos, incluidos los ataques de inyección SQL. Estas reglas pueden ser actualizadas regularmente para mantenerse al día con las últimas amenazas.
4. Un WAF puede implementar políticas de control de acceso que limiten el acceso a ciertos recursos o funcionalidades de tu aplicación o sitio web, lo que puede ayudar a prevenir ataques de inyección SQL al restringir el acceso a áreas sensibles de tu aplicación.
5. Algunos WAFs pueden realizar un seguimiento del comportamiento normal de tu aplicación web y detectar desviaciones significativas que podrían indicar un ataque de inyección SQL u otras actividades maliciosas.

En resumen, los ataques de inyección SQL representan una amenaza seria para la seguridad de tu web y de los datos que se almacenan en ella, pero con las medidas adecuadas, puedes proteger tus aplicaciones web y datos sensibles contra estos ataques maliciosos.

Ataques XSS o cross-site scripting

Un ataque XSS (Cross-Site Scripting) es una vulnerabilidad de seguridad que ocurre cuando un atacante inyecta código malicioso, generalmente JavaScript, en páginas web vistas por otros usuarios. Este código se ejecuta en el navegador de la víctima, lo que permite al atacante robar información, tomar el control de la sesión del usuario, redirigir a otros sitios maliciosos o incluso modificar el contenido de la página web.

Existen varios tipos de ataques XSS, pero generalmente se dividen en tres categorías:

Reflejado (Reflected XSS): En este tipo de ataque, el código malicioso se inyecta a través de un enlace malicioso o un formulario en la página web. La entrada del usuario se refleja en la página web sin ser sanitizada adecuadamente, lo que permite que el código malicioso se ejecute en el navegador de la víctima cuando visita esa página específica.

En este caso, un atacante crea un enlace malicioso o un formulario que contiene código JavaScript malicioso.

Cuando un usuario hace clic en el enlace o envía el formulario, el código malicioso se envía al servidor web.

El servidor web devuelve la página al usuario, incluyendo el código malicioso en la respuesta.

El navegador del usuario ejecuta el código JavaScript, ya que lo considera parte de la página legítima.

Este tipo de ataque a menudo se utiliza en campañas de phishing, donde el atacante intenta engañar al usuario para que divulgue información confidencial.

Prevención: Los WAF permiten mitigar este riesgo mediante la sanitización y validación adecuada de todas las entradas del usuario antes de mostrarlas en la página. Esto implica codificar o escapar caracteres especiales para que el navegador los interprete como texto plano en lugar de código ejecutable.

Persistente (Stored XSS): En este caso, el código malicioso se almacena en la base de datos de la aplicación web, como en un comentario de un blog o un mensaje en un foro. Cuando otros usuarios acceden a la página que contiene el código malicioso, este se ejecuta en sus navegadores, lo que les expone al ataque.

En este caso, el código malicioso se almacena en la base de datos de la aplicación web, como en un comentario de un blog, un mensaje en un foro o un perfil de usuario.

Cuando otros usuarios acceden a la página que contiene el código malicioso, este se ejecuta en sus navegadores.

Este tipo de ataque puede ser especialmente peligroso porque puede afectar a múltiples usuarios y persistir en la página web durante períodos prolongados.

DOM-based XSS: Este tipo de ataque ocurre cuando la vulnerabilidad XSS reside en el lado del cliente en lugar del servidor. El código malicioso se ejecuta en el navegador del cliente manipulando el Document Object Model (DOM) de la página web.

En este tipo de ataque, el código malicioso se ejecuta en el navegador del cliente manipulando el DOM de la página web.

El atacante aprovecha las vulnerabilidades en el código JavaScript de la página web para ejecutar código malicioso.

Este tipo de XSS no involucra necesariamente comunicación con el servidor, lo que lo hace más difícil de detectar mediante técnicas tradicionales de seguridad del lado del servidor.

Ataques de CORS o Cross-Origin Resource Sharing

El Cross-Origin Resource Sharing (CORS) es un mecanismo de seguridad utilizado por los navegadores web para permitir o restringir las solicitudes de recursos (como archivos JavaScript, CSS, imágenes, etc.) desde un origen (dominio, protocolo y puerto) a otro origen diferente al de la página web actualmente visualizada. Fue introducido para abordar los problemas de seguridad asociados con las solicitudes de recursos entre orígenes distintos en aplicaciones web.

Cuando un navegador realiza una solicitud de recursos a un servidor web, incluye información sobre el origen desde el cual se originó la solicitud. El servidor web, en función de esa información, puede decidir si permitir o bloquear la solicitud según la política de CORS configurada.

Gran parte de los sitios web operan bajo la política del mismo origen, conocida en inglés como **same-origin policy (SOP)**. Esta política se rige bajo una filosofía muy simple: impedir la carga de datos procedentes de otros servidores. Todos los archivos tienen que compartir la misma fuente.

Al igual que la política del mismo origen (SOP), el **CORS** restringe todas aquellas peticiones cuyo origen difiere del dominio al que se realiza la petición. Pero, al contrario que la SOP, deja abierta la puerta a la colaboración en momentos puntuales.

De este modo, se previene la **infiltración de contenido malicioso** por parte de terceros. Si no se filtrasen las peticiones según su procedencia, se le estarían abriendo las puertas de par en par a los atacantes. Y estos tendrían libertad plena para **introducir malware en las páginas**.

Aquí hay una explicación más detallada de cómo funciona el CORS:

1. **Solicitud de origen cruzado (Cross-Origin Request):** Una solicitud de origen cruzado es una solicitud HTTP realizada por un navegador desde la página web actual a un origen diferente al de la página web actual. Por ejemplo, si una página web alojada en `https://www.ejemplo.com` intenta hacer una solicitud a `https://api.ejemplo.com`, esto se consideraría una solicitud de origen cruzado.
2. **Cabeceras CORS:** Cuando se realiza una solicitud de origen cruzado, el navegador agrega cabeceras CORS a la solicitud HTTP. Estas cabeceras incluyen información sobre el origen desde el cual se originó la solicitud (`Origin`) y el tipo de solicitud que se está realizando (por ejemplo, `GET`, `POST`, `PUT`, `DELETE`, etc.).
3. **Respuesta CORS:** El servidor web al que se envía la solicitud de origen cruzado puede responder con cabeceras CORS específicas que indican si se permite o no la solicitud desde el origen dado. Estas cabeceras incluyen principalmente `Access-Control-Allow-Origin`, que especifica los orígenes permitidos para realizar solicitudes, y otras cabeceras

relacionadas con el control de acceso, como `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`, `Access-Control-Allow-Credentials`, etc.

4. **Política CORS:** La política CORS se configura en el servidor web para determinar qué solicitudes de origen cruzado son permitidas y cuáles son bloqueadas. Esto puede basarse en el origen de la solicitud, los métodos HTTP utilizados, las cabeceras enviadas con la solicitud, entre otros factores.

Ejemplos de código con CORS

Curl

```
curl -i -X OPTIONS tecnocratica.net/api/dns \
-H 'Access-Control-Request-Method: GET' \
-H 'Access-Control-Request-Headers: Content-Type, Accept' \
-H 'Origin: http://api.tecnocratica.net'
```

Apache

```
<IfModule mod_headers.c>
    <FilesMatch "\.(ttf|ttc|otf|eot|woff|font.css|css|js|gif|png|jpe?g|svg|svgz|ico|webp)$">
        Header set Access-Control-Allow-Origin "*"
    </FilesMatch>
</IfModule>
```

Nginx

```
location ~ \.(ttf|ttc|otf|eot|woff|font.css|css|js|gif|png|jpe?g|svg|svgz|ico|webp)$ {
    add_header Access-Control-Allow-Origin "*";
}
```

El CORS es esencial para proteger las aplicaciones web contra ataques de origen cruzado, como el Cross-Site Request Forgery (CSRF) y el Cross-Site Script Inclusion (XSSI). Sin embargo, es importante configurarlo correctamente para evitar problemas de seguridad, como la exposición accidental de datos sensibles. Los desarrolladores web deben comprender cómo funciona CORS y configurarlo adecuadamente en sus servidores para garantizar la seguridad de sus aplicaciones.

Ataques de CSRF o Cross-Site Request Forgery

El Cross-Site Request Forgery (CSRF), también conocido como ataque de falsificación de solicitudes entre sitios, es un tipo de vulnerabilidad de seguridad en aplicaciones web que explota la confianza de un sistema en las solicitudes realizadas desde el navegador del usuario autenticado. En un ataque CSRF, un atacante engaña a un usuario autenticado para que realice una acción no deseada en un sitio web al aprovechar la sesión activa del usuario como por ejemplo, cambiar su dirección de correo electrónico, su contraseña o realizar una transferencia de dinero.

Mediante una CSRF, un atacante puede eludir parcialmente la política que evita que diferentes sitios web se interfieran entre sí (Same-Origin Policy).

Aquí hay una explicación más detallada de cómo funciona un ataque CSRF:

1. **Autenticación del Usuario:** El usuario autenticado accede a un sitio web legítimo y se autentica con éxito. Durante este proceso, el sitio web establece una sesión válida para el usuario.
2. **Solicitud Maliciosa:** Mientras la sesión del usuario sigue activa, el usuario visita otro sitio web controlado por el atacante, que contiene un código malicioso, como un enlace o un formulario oculto, que realiza una solicitud a la aplicación vulnerable.
3. **Ejecución de la Solicitud:** La solicitud maliciosa se ejecuta en el contexto de la sesión válida del usuario en el sitio web vulnerable. Como resultado, la aplicación web procesa la solicitud como si fuera legítima, ya que proviene de un usuario autenticado.
4. **Acción No Deseada:** Dependiendo de la naturaleza de la solicitud maliciosa, puede llevar a cabo acciones no autorizadas en nombre del usuario, como cambiar la contraseña, realizar compras, eliminar datos, etc.
5. **Impacto:** El atacante puede aprovechar el ataque CSRF para realizar acciones maliciosas sin que el usuario sea consciente de ello. Esto puede comprometer la integridad y la seguridad de los datos del usuario, así como llevar a cabo actividades fraudulentas.

Para prevenir los ataques CSRF, se pueden implementar varias medidas de seguridad, incluyendo:

- **Tokens Anti-CSRF:** Las aplicaciones web pueden generar tokens únicos y aleatorios y asociarlos con las acciones sensibles realizadas por los usuarios. Estos tokens que genera la aplicación del lado del servidor y se transmite al cliente de tal manera que se incluye en la siguiente solicitud realizada por el cliente y se incluyen en los formularios o las solicitudes y se verifican en el servidor para garantizar que la solicitud provenga de una fuente legítima y no de un atacante. Los Tokens previenen estos ataques dado que el atacante no puede predecir el valor del Token CSRF del usuario y no puede construir una

solicitud con todos los parámetros necesarios para que la aplicación cumpla con la solicitud.
- **Cabeceras HTTP:** El uso de cabeceras HTTP como `SameSite` y `Referer` puede ayudar a mitigar los ataques CSRF al restringir el envío de cookies en solicitudes cruzadas y verificar el origen de las solicitudes, respectivamente.
- **Autenticación de Doble Factor (2FA):** La implementación de la autenticación de doble factor puede agregar una capa adicional de seguridad al requerir que los usuarios proporcionen un segundo factor de autenticación, como un código único enviado a su teléfono móvil, antes de realizar acciones sensibles.

Al aplicar estas medidas de seguridad, las aplicaciones web pueden protegerse de los ataques CSRF y garantizar la integridad y la seguridad de los datos del usuario.

Ejemplo

Vamos a crear un sitio malicioso llamado **bancolsoeduardos.es**

Nuestra víctima visita nuestro sitio malicioso "**bancolsoeduardos.es**", se activa una solicitud POST y se envía a la aplicación legítima de **bancoloseduardos.es**. El JavaScript que se encuentra en las etiquetas "script" garantiza que el formulario se envíe tan pronto como el usuario cargue la página, sin que se requiera ninguna interacción del usuario, ni que el usuario se dé cuenta de lo que está sucediendo.

```
<html>
  <body>
    <form action="https://bancoloseduardos.es/transfer" method="POST">
      <input type="hidden" bsb="421314" accountNo="123456789" amount="100" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

El formulario anterior crea la siguiente solicitud para la aplicación legítima de **bancoloseduardos.es**. La solicitud contiene la cookie de sesión del usuario legítimo, pero contiene nuestro número de cuenta bancaria.

```
POST /transfer HTTP/1.1
Host: bancoloseduardos.es
Content-Length: 42
Content-Type: application/x-www-form-urlencoded
Cookie: session=3PhtXFzhEWzdhFpQfTqrcjW2ItpDAkDm
```

```
bsb=421314&accountNo=1736123125&amount=100
```

Este ataque fue posible debido a algunas condiciones:

El usuario inició sesión en **bancoloseduardos.es**
El usuario que visitó nuestro sitio también inició sesión en la aplicación **bancoloseduardos.es**. Su cookie de sesión se estaba almacenando en su navegador y, como no tenía el atributo SameSite, pudimos robarla para nuestra solicitud.

Instalar Modsecurity como WAF

Modsecurity es una de las aplicaciones que más se usa para proteger los servidores web mediante las funcionalidades WAF que ofrece, aunque se rumorea que va a dejar de existir, todavía tiene validez, ya que las alternativas están poco maduras.

Instalar ModSecurity en Debian 12

Actualizar el sistema

Como siempre comenzaremos por actualizar nuestro sistema.

```
apt update && apt upgrade
```

Instalar ModSecurity

A continuación instalamos el paquete modsecurity

```
apt install libapache2-mod-security2
```

Habilitar ModSecurity

Habilitamos el Modsecurity en el apache, añadiendo el módulo.

```
a2enmod security2
```

Reiniciamos apache

```
systemctl restart apache2
```

Configurar ModSecurity

El archivo de configuración principal de ModSecurity se encuentra en **/etc/apache2/mods-enabled/security2.conf.** Este archivo contiene una gran cantidad de opciones que puede modificar para personalizar el comportamiento de ModSecurity.

- SecRuleEngine: Esta opción define si ModSecurity está en modo de detección (**DetectionOnly**) o si está bloqueando solicitudes (**On**).
- SecDefaultAction: Esta opción define la acción predeterminada que ModSecurity tomará cuando se detecte una violación de las reglas.
- SecRulesFile: Esta opción define la ruta al archivo de reglas de ModSecurity.

Ahora lo preparamos

```
mv /etc/modsecurity/modsecurity.conf-recommended /etc/modsecurity/modsecurity.conf
```

Editamos el fichero y buscamos la línea

```
SecRuleEngine DetectionOnly
```

Y la cambiamos por **SecRuleEngine On**, como hemos comentado antes. Luego buscaa siguiente línea (línea 186), que le indica a ModSecurity qué información debe incluirse en el registro de auditoría.

```
SecAuditLogParts ABDEFHIJZ
```

La cambiamos por la siguiente

```
SecAuditLogParts ABCEFHJKZ
```

Reiniciamos apache

```
systemctl restart apache2
```

Ejemplo de configuración

Vamos a ver como configurar ModSecurity con OWASP CRS v3

Instalar ModSecurity y OWASP CRS v3

```
wget https://github.com/coreruleset/coreruleset/archive/v3.3.0.tar.gz
tar xvf v3.3.0.tar.gz
mkdir /etc/apache2/modsecurity-crs/
mv coreruleset-3.3.0/ /etc/apache2/modsecurity-crs/
```

Vamos a la carpeta

```
cd /etc/apache2/modsecurity-crs/coreruleset-3.3.0/
mv crs-setup.conf.example crs-setup.conf
```

Editar el archivo /etc/apache2/mods-enabled/security2.conf

```
nano /etc/apache2/mods-enabled/security2.conf
```

Buscamos la línea

```
IncludeOptional /usr/share/modsecurity-crs/*.load
```

Y la sustituimos por:

```
IncludeOptional /etc/apache2/modsecurity-crs/coreruleset-3.3.0/crs-setup.conf
IncludeOptional /etc/apache2/modsecurity-crs/coreruleset-3.3.0/rules/*.conf
```

Guardamos y probamos la configuración de apache

```
apache2ctl -t
```

Si todo es correcto. reiniciamos apache

```
systemctl restart apache2
```

Reglas personalizadas

Podemos editar el archivo de configuración de OWASP CRS /etc/apache2/modsecurity-crs/coreruleset-3.3.0/crs-setup.conf para configurar reglas personalizadas.

Reiniciar Apache

```
systemctl restart apache2
```

Probar ModSecurity

Puedes probar ModSecurity visitando la web con un navegador web y ejecutando una herramienta de prueba de vulnerabilidades como OWASP ZAP.

Puedes ver que OWASP CRS se puede ejecutar en dos modos:

self-contained. Este es el modo tradicional utilizado en CRS v2.x. Si una solicitud HTTP coincide con una regla, ModSecurity bloqueará la solicitud HTTP inmediatamente y dejará de evaluar las reglas restantes.
anomaly scoring mode. Este es el modo predeterminado utilizado en CRS v3.x. ModSecurity comparará una solicitud HTTP con todas las reglas y agregará una puntuación a cada regla coincidente. Si se alcanza un umbral, la solicitud HTTP se considera un ataque y se bloqueará. La puntuación predeterminada para las solicitudes entrantes es 5 y para la respuesta saliente es 4.

Cuando se ejecuta en modo de puntuación de anomalías, hay 4 niveles de paranoia.

- Nivel de paranoia 1 (predeterminado)
- Paranoia nivel 2
- Paranoia nivel 3
- Paranoia nivel 4

Con cada aumento del nivel de paranoia, el CRS habilita reglas adicionales que le brindan un mayor nivel de seguridad. Sin embargo, los niveles más altos de paranoia también aumentan la posibilidad de bloquear parte del tráfico legítimo debido a falsas alarmas.

Ejemplos de reglas de OWASP CSR v3

```
SecRule REQUEST_METHOD "^(TRACE|TRACK)$" "phase:1,deny,log,status:405"
SecRule REQUEST_HEADERS:Content-Type "application/x-www-form-urlencoded"
"phase:2,deny,log,status:403"
SecRule REQUEST_URI "@rx (..|~) " "phase:2,deny,log,status:403"
```

A continuación por ejemplo tenemos un ejemplo de un conjunto de reglas de hardening de OWASP CSR v3 para Wordpress. Lo puedes encontrar en este enlace de GitHub

Aquí ponemos 3 ejemplos del código anterior para bloquear accesos no autorizados a la carpeta wp-includes para subir archivos del tipo php, el acceso a la ruta wp-admin y el acceso al XML-RPC

```
SecRule REQUEST_FILENAME "^/wp\-includes(/.*\.php(|[\/].*)|(|\/))$" "phase:1,id:22200001,\
  t:lowercase,t:normalizePath,t:trim,\
  block,\
  log,\
  rev:'1',\
  severity:'6',\
  maturity:'9',\
  accuracy:'9',\
  ver:'%{tx.wprs_version}',\
  tag:'wordpress',\
  tag:'includes',\
  logdata:'Request Filename %{REQUEST_FILENAME}',\
  msg:'WordPress: /wp-includes access attempt'"
SecRule REQUEST_FILENAME "^/wp-admin/(?:install|includes)" "phase:1,id:22200003,\
  t:lowercase,t:normalizePath,t:trim,\
  block,\
  log,\
  rev:'1',\
  severity:'6',\
  maturity:'9',\
```

```
    accuracy:'9',\
    ver:'%{tx.wprs_version}',\
    tag:'wordpress',\
    tag:'includes',\
    logdata:'Request Filename %{REQUEST_FILENAME}',\
    msg:'WordPress: File /wp-admin access attempt'"
  SecRule tx:wprs_allow_xmlrpc "@eq 1" \
    "phase:1,\
    id:22200013,\
    pass,\
    nolog,\
    skipAfter:END_WPRS_XMLRPC"

  SecMarker BEGIN_WPRS_XMLRPC

  SecRule REQUEST_FILENAME "^/xmlrpc\.php" "phase:1,id:22200015,\
    t:lowercase,t:normalizePath,t:trim,\
    block,\
    log,\
    rev:'1',\
    severity:'6',\
    maturity:'9',\
    accuracy:'9',\
    ver:'%{tx.wprs_version}',\
    tag:'wordpress',\
    tag:'xmlrpc',\
    logdata:'Request Filename %{REQUEST_FILENAME}',\
    msg:'WordPress: /xmlrpc.php access attempt'"

  SecMarker END_WPRS_XMLRPC
```

Interpretar los registros de ModSecurity

Es importante analizar los registros de ModSecurity para saber qué tipo de ataques se dirigen a sus aplicaciones web y tomar mejores medidas para defenderse de los actores de amenazas. Existen principalmente dos tipos de registros en ModSecurity:

registro de depuración: deshabilitado de forma predeterminada.
registro de auditoría: /var/log/apache2/modsec_audit.log
Para comprender los registros de auditoría de ModSecurity, necesita conocer las 5 fases de procesamiento en ModSecurity, que son:

- Fase 1: inspeccionar los encabezados de las solicitudes
- Fase 2: inspeccionar el cuerpo de la solicitud
- Fase 3: inspeccionar los encabezados de respuesta
- Fase 4: inspeccionar el cuerpo de respuesta
- Fase 5: Acción (registro/bloqueo de solicitudes maliciosas)

También hay dos tipos de archivos de registro.

Serial / Serie : un archivo para todos los registros. Este es el valor predeterminado.
Concurrent /Simultáneo: múltiples archivos de log. Esto puede proporcionar un mejor rendimiento de escritura. Si nota que sus páginas web se ralentizan después de habilitar ModSecurity, puede optar por utilizar el tipo de registro simultáneo.
Los eventos del registro se dividen en varias secciones.

- sección A: encabezado del registro de auditoría
- sección B: encabezado de solicitud
- sección C: cuerpo de la solicitud
- sección D: reservada
- sección E: intermediary response body
- sección F: encabezados de respuesta final
- sección G: reservada
- sección H: audit log trailer
- sección I: compact request body alternative, which excludes files
- sección J: información sobre archivos cargados
- sección K: cada regla que coincide con un evento, en orden de coincidencia
- sección Z: límite final

Si ejecuta un sitio web con mucho tráfico, el registro de auditoría de ModSecurity puede volverse demasiado grande muy rápidamente, por lo que debemos configurar la rotación de registros para el registro de auditoría de ModSecurity. Cree un archivo de configuración logrotate para ModSecurity.