Guía de seguridad en aplicaciones Web PHP

SEGURIDAD - PHP

Algunos *tips* sobre seguridad en aplicaciones Web bajo PHP, nunca vienen mal. En la primera edición de la revista, hablamos sobre como prevenir inyecciones SQL utilizando mysqli. Si bien cada aplicación "es un mundo", algunas medidas básicas suelen ser más que útiles, inevitables en toda aplicación. Hablaremos de ellas en este artículo.

Escrito por: **Eugenia Bahit** (Arquitecta GLAMP & Agile Coach)

Eugenia es **Arquitecta de Software**, **docente** instructora de tecnologías **GLAMP** (GNU/Linux, Apache, MySQL, Python y PHP) y **Agile coach** (UTN) especializada en Scrum y eXtreme Programming. Miembro de la **Free Software Foundation** e integrante del equipo de **Debian Hackers**.

Webs:
Cursos de programación a Distancia: www.cursosdeprogramacionadistancia.com
Agile Coaching: www.eugeniabahit.com

Redes sociales:
Twitter / Identi.ca: **@eugeniabahit**

Si debe tener permisos de escritura, no debe estar servido

Sin dudas, este es uno de los puntos fundamentales a tener en cuenta al momento de desarrollar nuestra aplicación. Frecuentemente los CMS, plataformas de e-Learning e incluso de comercio electrónico -y me refiero a los más populares-, al momento de su instalación, nos piden asignar un directorio con permisos de escritura a fin de poder cargar archivos de imágenes (o cualquier otro contenido estático), mediante su plataforma de administración. Mayormente, es requisito que este directorio se encuentre servido dentro del `DocumentRoot`.

Un directorio Web con permisos de escritura, es una puerta sin llave ni cerrojo en un barrio lujoso: algo tentador para aquellas personas a quiénes les gusta ir de visita a casas ajenas, sin invitación y tomando como propios aquellos objetos de terceros (por si no se entendió, "ladrón", solo que dicho en plan Neruda de supermercado).

El principal inconveniente que se le presenta a algunos programadores al momento de optar por no "servir" un directorio con permisos de escritura, es **¿cómo haré accesibles**

los archivos estáticos mediante el navegador si el directorio no se encuentra servido[31]? La respuesta a esta pregunta es simple: engañando al navegador.

La forma más eficiente, segura, genérica y reutilizable para lograrlo, es hacer que los archivos estáticos que se encuentran en un directorio no accesible mediante el navegador, se sirvan a través de un único archivo PHP encargado de leer y mostrar dichos archivos modificando previamente el *Content-Type* (tipo de contenido) en los encabezados HTTP.

Desde la versión 5.3 de PHP se dispone del módulo **`fileinfo`**[32], quien mediante sus funciones nos ayudará con este proceso. En versiones anteriores, puede utilizarse la **función obsoleta** `mime_content_type()`. Veamos un ejemplo con `fileinfo`.

Supongamos que nuestro *script* de carga de archivos, guarda los mismos en el directo privado `/miswebs/example.com/uploads/` mientras que el `DocumentRoot` se encuentra en `/miswebs/example.com/www/`. Dentro de **`www`**, crearemos nuestro pequeño *script* encargado de servir archivos estáticos: **`showfile.php`**. La tarea de este *script*, consistirá entonces, en:

```
<?php

# Definimos la ruta del directorio privado.
# Este es el directorio con permisos de escritura.
$ruta = '/miswebs/example.com/uploads';

# El nombre del archivo que se desea servir, lo obtendremos
# mediante el parámetro 'file' que luego será pasado por la URI
$file = isset($_GET['file']) ? "$ruta/{$_GET['file']}" : NULL;

# Verificamos el valor de $file
# Si no es NULL, verificamos si el archivo existe antes de proceder
if(!is_null($file)) {
    if(file_exists($file)) {
        # Creamos un recurso fileinfo para obtener el tipo MIME
        $resource = finfo_open(FILEINFO_MIME_TYPE);
        # Obtenemos el tipo MIME
        $mimetype = finfo_file($resource, $file);
        # Cerramos el recurso
        finfo_close($resource);

        # Modificamos los encabezados HTTP
        header("Content-Type: $mimetype");
        # Leemos y mostramos el archivo
        readfile($file);
    }
}

?>
```

Para ver el archivo (como si en realidad estuviese servido), solo bastará con ingresar al

31 Un directorio “servido” se refiere al hecho de poder ingresar en éste, mediante el navegador Web, a través de una URL simple mediante el protocolo HTTP/HTTPS.

32 http://php.net/manual/es/book.fileinfo.php

archivo `showfile.php` pasándole el nombre del estático como parámetro mediante el argumento `file`: **`http://example.com/showfile.php?file=mi_imagen.jpg`**

Vale aclarar que este *script*, servirá para cualquier tipo MIME que pueda ser entendido por el navegador: desde archivos HTML, CSS y JavaScript, hasta imágenes, vídeos, archivos de audio y cualquier otro contenido multimedia.

Si viene por $_GET o $_POST se filtra pero en el usuario nunca se confía

Nada tan conocido, pero nunca está demás recordarlo. No interesa si un dato se guardará o no en una base de datos, en un archivo o nada más se quiere mostrar de forma "volátil" en la pantalla. Eliminar etiquetas HTML y PHP es una buena forma de **evitar los ataques XSS (*Cross Site Scripting*)** y solo lleva menos tiempo que un suspiro:

```
$dato_loco = strip_tags($_GET['maldito_parametro']);
```

Y, convertir caracteres "conflictivos" como comillas dobles y simples, paréntesis angulares y otros caracteres estrafalarios, es solo agregar una función más:

```
$dato_final = htmlentities($dato_loco);
```

Para **validar y sanear datos** (sí, sanear. La palabra "sanitizar" no existe en español), desde la versión 5.2 de PHP, se incorpora la extensión **Filter**[33], la cuál provee funciones de validación y saneamiento, que nos permiten validar si los datos se corresponden a los esperados y limpiarlos, respectivamente.

Esta extensión viene instalada por defecto desde la versión 5.2 de PHP y se puede configurar su aplicación, previamente desde las directivas `filter.default` y `filter.default_flag` del archivo `php.ini`. (Ver más detalles en la Web oficial: http://php.net/manual/es/filter.configuration.php)

Mediante la función **`filter_var($dato, FILTRO[, OPCIONES])`**[34] se puede limpiar y una gran variedad de datos (desde enteros y booleanos hasta correos electrónicos y direcciones IP). Una completa **lista de filtros** posibles, puede encontrarse en http://php.net/manual/es/filter.filters.validate.php.

El saneamiento de los datos también nos provee un gran abanico de filtros que pueden ser consultadas en http://php.net/manual/es/filter.filters.sanitize.php, mientras que las **opciones de filtro y sanación**, pueden obtenerse visitando la URL: http://php.net/manual/es/filter.filters.flags.php

33 http://php.net/manual/es/intro.filter.php
34 http://php.net/manual/es/function.filter-var.php

Vale aclarar que estos filtros y sanadores, son válidos para otras variables superglobales (además de `$_GET` y `$_POST`, como `$_COOKIE` y `$_SERVER`).

Un "Disculpe las molestias" siempre es mejor que un Warning

Nada más agradable para un programador que ver los errores generados por PHP en la pantalla del navegador, como si los logs de Apache y PHP no existieran. Pero eso, no debe hacerse en el servidor de producción.

Los errores de PHP dan demasiada información al usuario. Más información de la que un usuario común puede entender y de la que un usuario "no tan común" debería conocer. ¿Para qué mostrarla entonces?

Los errores pueden ser silenciados con una simple @ delante de una instrucción:

```
@fopen('/home/user/templates/archivo_que_no_existe.html', 'r');
```

Pero mucho más efectivo y cómodo para el desarrollo, es modificar las directivas de `php.ini` en tiempo de ejecución, dependiendo de si se está trabajando en desarrollo u operando en producción:

```
const PRODUCCION = False;  # En producción, se establece en True

if(!PRODUCCION) {
    ini_set('error_reporting', E_ALL | E_NOTICE | E_STRICT);
    ini_set('display_errors', '1');
} else {
    ini_set('display_errors', '0');
}
```

Si es divertido y fascinante para el programador, es una mala idea para el servidor

Sí, es fascinante ponerse a jugar con la función `shell_exec()` y ver como desde un simple *script* Web, puedes hacer cosas como ejecutar locos comandos del sistema operativo. Pero ya. Es un juego... y peligroso. Permitir la ejecución de funciones como `exec()`, `shell_exec()`, `popen()`, `system()` y "compañía" es un claro síntoma del *síndrome del Kamikaze* (sí, es un síndrome que acabo de inventar). Se puede -y debe- evitar que dichas funciones sean ejecutadas, mediante la directiva **`disable_functions`** del archivo `php.ini`.

Una buena configuración de esta directiva, podría ser la siguiente:

```
disable_functions = proc_open, popen, disk_free_space, diskfreespace,
set_time_limit, leak, tmpfile, exec, system, shell_exec, passthru,
show_source, system, phpinfo, pcntl_alarm, pcntl_fork, pcntl_waitpid,
pcntl_wait, pcntl_wifexited, pcntl_wifstopped, pcntl_wifsignaled,
pcntl_wexitstatus, pcntl_wtermsig, pcntl_wstopsig, pcntl_signal,
pcntl_signal_dispatch, pcntl_get_last_error, pcntl_strerror,
```

```
pcntl_sigprocmask, pcntl_sigwaitinfo, pcntl_sigtimedwait, pcntl_exec,
pcntl_getpriority, pcntl_setpriority
```

Si lo recibes desde un formulario, no hagas nada hasta no estar seguro

Te matas filtrando datos con JQuery y haciéndole miserable la vida a los usuarios que no entienden que todos los malditos campos del maldito formulario son obligatorios. Y ni te cuento si esto mismo lo venías haciendo desde la época en la que para *John Resig*, los Pitufos eran más importantes e imprescindibles que un ordenador y, todo lo que hoy hace JQery en una sola instrucción, a ti te demandaba unas 744 mil 288 coma 73 líneas de código (no es que yo tenga años para repartir, solo me lo han contado...).

JavaScript por aquí, alertas por allá, bonitos *layers* que se despliegan lentamente con unos mágicos *fadeIn*, *fadeOut* y *fadeWTF*. Todo muy lindo, muy mágico, muy *Alicia en el País de las Maravillas*, pero basta con un simple clic derecho > inspeccionar elemento > editar... para que tu mágico arte se diluya cual chocolate en leche hirviendo (perdón, me apetece una rica *chocolatada*) y te envíen el formulario como se les de la maldita gana. Y peor aún cuando el mismo formulario se copia, edita y sube en un servidor que no es el tuyo o más frustrante aún, ni siquiera se sube a un servidor y se le hace un simple "doble click" al maldito HTML editado.

Los datos importantes, se filtran y validan desde PHP, no desde JavaScript. JavaScript se ejecuta en el cliente y éste, es quien tiene todo el control sobre aquel. Y nunca está demás, verificar desde la URL de la cual proviene dicho formulario:

```
$uri_donde_muestras_tu_form = 'http://example.com/index.php?page=contacto';

if(isset($_SERVER['HTTP_REFERER'])) {
    if($_SERVER['HTTP_REFERER'] != $uri_donde_muestras_tu_form) {
        // El form no se está enviando desde el archivo que creías
    }
}
```

En caso de tener accesible el formulario en más de una URI, bastará con definir las URI permitidas en un *array* (una **lista blanca**) y validar el *referer* con la función `in_array()`:

```
$form_uris = array(
   'http://example.org/contacto.php',
   'http://www.example.org/contacto.php',
   'https://example.org:8000/contacto.php'
   'http://otrodominio.com/contacto'
);

if(isset($_SERVER['HTTP_REFERER'])) {
    if(!in_array($_SERVER['HTTP_REFERER'], $form_uris)) {
        // El form no se está enviando desde el archivo que creías
    }
}
```

Vale aclarar que las listas blancas son también útiles cuando se requieren llamadas de retorno (callback) dinámicas:

```
$funcion = $_GET['pagina'];
$allowed_functions = array('mostrar_form', 'enviar_form');

if(in_array($funcion, $allowed_functions)) {
   # función permitida
   call_user_func($funcion);
}
```

Otra forma de mantener a salvo los formularios, es cerciorándote de que que los datos recibidos (nombres de tus campos de formulario) son los esperados. Si se recibe algún campo extra o menos cantidad de los esperados, algo no huele bien.

```
$campos_esperados = array('nombre', 'apellido', 'edad');

foreach($campos_esperados as $campo) {
    # Obtenemos una variable con el mismo nombre de campo
    $$campo = isset($_POST[$campo]) ? $_POST[$campo] : NULL;
}
```

También puedes aprovechar el paso anterior, para "matar dos pájaros de un tiro", validar y sanear los datos con `filter_var()`.

Si es una contraseña, se hashea. Si no está hasheado, no es una contraseña.

Nunca guardes ni tus contraseñas ni las de tus usuarios en texto plano. Las contraseñas deben guardarse *hasheadas*, vayan o no a estar almacenadas en una base de datos, pues **una contraseña debe ser irrecuperable**.

Guardar contraseñas *hasheadas* es tan simple como recurrir, por ejemplo, a la función `md5()`:

```
$clave = md5($_POST['clave']);
```

Luego, para comparar la contraseña del usuario con la almacenada, simplemente se vuelve a *hashear* el *input* de la misma forma que se hizo para guardarlo.

En un próximo artículo, intentaremos centrar el foco en la prevención de ataques CSRF (Cross Site Request Fogery) y la captura de sesiones implementando identificadores de sesión secretos (tokens) y regenerando identificaciones únicas de sesión (session ID) de forma más profunda.

Ahora **¡a asegurar tu aplicación!**