



PHP – Formularios



Ciclo: DAW
Módulo: DWES
Curso: 2020-2021
Autor: César Guijarro Rosaleny



Introducción.....	3
Leer datos de los formularios.....	4
Ataques XSS.....	9
Tipos de ataque XSS.....	12
Prevenir ataques XSS.....	13
Otros tipos de ataques.....	16
SQL Injection.....	16
CSRF.....	16

Introducción

Una de las características más potente de PHP es la forma de gestionar los formularios. Cuando enviamos los datos de un formulario HTML, lo que hacemos es llamar al archivo del servidor que escribimos en el atributo *action* del formulario.

Dependiendo del método que elijamos en *method*, con PHP podremos acceder a los datos del formulario mediante las variables predefinidas `$_GET` o `$_POST`.

A partir de ahí, podemos tratar los datos como queramos (algo habitual es acceder a una bbdd para gestionar dichos datos).

Una de las cosas que tenemos que tener en cuenta a la hora de trabajar con formularios es el tema de la seguridad. Veremos algunos ejemplos de ataques a webs a través de formularios y como protegernos de ellos.

Leer datos de los formularios

Para entender como leer los datos de un formulario HTML, vamos a crear un archivo en nuestra carpeta `htdocs_apache/DWS/Ejemplos/T1` llamado `formulario.html` con el siguiente código:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Formulario</title>
</head>
<body>
  <form action="http://localhost:8080/DWS/ejemplos/T2/tratarDatos.php"
    method="POST">
    <label for="nombre">Nombre: </label>
    <input type="text" name="nombre">
    <label for="apellidos">Apellidos:</label>
    <input type="text" name="apellidos">
    <label for="edad">Edad:</label>
    <input type="number" name="edad">
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

Si abrimos el fichero y le damos al botón *Enviar* nos saltará un error, ya que intentaremos acceder al archivo `tratarDatos.php` que no existe en nuestro servidor.

NOTA: Si abrimos el archivo html directamente con doble click, el navegador lo tomará como un archivo y deberemos poner la url completa en el *action* del formulario (el puerto 8080 puede variar con respecto al ejemplo): <http://localhost/DWS/ejemplos/T2/tratarDatos.php>

Vamos a crear ahora el archivo *tratarDatos.php* en la misma carpeta con el siguiente código:

```
<?php
var_dump($_POST);
```


Si todo ha funcionado como toca, el navegador debería mostrar algo parecido a ésto:



```
localhost:8080/DWS/ejemplos/T2/tratarDatos.php:3:
array (size=3)
  'nombre' => string 'Pepe' (length=4)
  'apellidos' => string 'Martínez Giménez' (length=18)
  'edad' => string '23' (length=2)
```

Si te das cuenta, la variable `$_POST` almacena en forma de array los campos del formulario y el valor que ha escrito el usuario. Las claves del array son el atributo *name* de los diferentes controles del formulario (en nuestro caso los tres *inputs*).

¿Qué pasa si se me olvida poner el atributo *name* en algún *input* del formulario?. En ese caso, PHP no podrá almacenar ese campo en la variable `$_POST`. Puedes hacer la prueba. Elimina el atributo *name* del *input* *edad* y vuelve a enviar el formulario. En este caso, la salida será:



```
localhost:8080/DWS/ejemplos/T2/tratarDatos.php:3:
array (size=2)
  'nombre' => string 'Pepe' (length=4)
  'apellidos' => string 'Martínez Giménez' (length=18)
```

Olvidarse el atributo *name* de los controles de un formulario es uno de los errores más comunes cuando se comienza a trabajar con PHP.

Obviamente, si hubiésemos elegido el metodo *GET* en lugar de *POST* en el atributo *method* del formulario, deberíamos acceder a la variable `$_GET` en vez de `$_POST`.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Formulario</title>
</head>
<body>
  <form action="http://localhost:8080/DWS/ejemplos/T2/tratarDatos.php"
    method="GET">
    <label for="nombre">Nombre: </label>
    <input type="text" name="nombre">
    <label for="apellidos">Apellidos:</label>
    <input type="text" name="apellidos">
    <label for="edad">Edad:</label>
    <input type="number" name="edad">
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

```
<?php
var_dump($_GET);
```

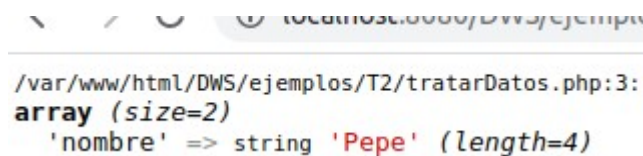
También podemos acceder a la variable `$_REQUEST` para acceder a los campos. Esta variable es un array asociativo que por defecto contiene el contenido de `$_GET`, `$_POST` y `$_COOKIE`, con la ventaja de que no tenemos que saber si el método elegido en el formulario es `$_GET` o `$_POST`.

Por otro lado, la principal desventaja es, precisamente, que no podemos diferenciar si las variables han sido enviadas por `$_GET` o `$_POST`.

Veamos un ejemplo de esto último. Deja el formulario como al principio, para que envíe los datos por `$_POST`, y cambia el *action* a *tratarDatos.php? nombre='juan'*.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Formulario</title>
</head>
<body>
  <form action="http://localhost:8080/DWS/ejemplos/T2/tratarDatos.php?
nombre=Juan" method="POST">
    <label for="nombre">Nombre: </label>
    <input type="text" name="nombre">
    <label for="apellidos">Apellidos:</label>
    <input type="text" name="apellidos">
    <label for="edad">Edad:</label>
    <input type="number" name="edad">
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

Si rellenamos el campo nombre con el valor *Pepe* y le damos a enviar, la salida es:



```
/var/www/html/DWS/ejemplos/T2/tratarDatos.php:3:
array (size=2)
  'nombre' => string 'Pepe' (length=4)
```

Como podemos observar, el valor que toma `$_REQUEST` es primero el valor enviado por POST y después el enviado por GET (si no viniera en POST).

Otra cosa que podría hacer un usuario si utilizamos `$_REQUEST` es “añadir” campos adicionales que no existen en el formulario original. Por ejemplo, si cambiamos el *action* del ejemplo anterior por `tratarDatos.php?anyo=1965`, rellenamos todos los datos del formulario y lo enviamos, la salida será:

```
/var/www/html/DWS/ejemplos/T2/tratarDatos.php:4:
array (size=7)
  'anyo' => string '1965' (length=4)
  'nombre' => string 'Pepe' (length=4)
  'apellidos' => string 'Martínez Giménez' (length=18)
  'edad' => string '34' (length=2)
  'pma_lang' => string 'es' (length=2)
  'pmaUser-1' => string '{"iv":"Q1nEyTSWYfN9xzoySniS9g:'
  'MOODLEID1_' => string '%EF%F5%A7%E5g%28%3A%23' (len
```

Ahora parece que nuestro formulario tiene 4 campos: *anyo*, *nombre*, *apellidos* y *edad*, cuando en realidad tiene sólo los 3 últimos (olvídate de los 3 últimos elementos de `$_REQUEST`).

Ten en cuenta que cualquier usuario puede manipular los archivos HTML, ya que el código es visible desde el cliente, con lo que si usamos `$_REQUEST` podría “añadir” tantos campos como quisiera a nuestro formulario. Por lo tanto, siempre es mejor usar `$_POST` o `$_GET`, según el método que elijamos para nuestro formulario.

Ataques XSS

A la hora de procesar los datos de un formulario desde nuestro archivo PHP hay una regla de oro que nunca se debe olvidar: **Nunca hay que fiarse de los datos que vienen de un usuario o cualquier otra fuente externa.**

Veamos un ejemplo de vulnerabilidad en nuestro formulario.



Copia el contenido del archivo *formulario.html* en otro archivo llamado *formulario2.html* y cambia el fichero PHP destino por *tratarDatos2.php*.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Formulario</title>
</head>
<body>
  <form
    action="http://localhost:8080/DWS/ejemplos/T2/tratarDatos2.php"
    method="POST">
    <label for="nombre">Nombre: </label>
    <input type="text" name="nombre">
    <label for="apellidos">Apellidos:</label>
    <input type="text" name="apellidos">
    <label for="edad">Edad:</label>
    <input type="number" name="edad">
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

Ahora crea el archivo *tratarDatos2.php* con el siguiente código:

```
<?php  
  
echo "Nombre:" . $_POST['nombre'] . "<br>";  
echo "Apellidos:" . $_POST['apellidos'] . "<br>";  
echo "Edad:" . $_POST['edad'] . "<br>";
```

Lo único que estamos haciendo es mostrar los datos introducidos por el usuario por pantalla.

```
Nombre:Pepe  
Apellidos:Martínez Giménez  
Edad:34
```

Vuelve a cargar *formulario2.php* en el navegador y escribe lo siguiente en el campo apellidos:

```
<script>alert('Hola')</script>
```

Nombre: Apellidos: Edad:

Dale al botón enviar y observa que ocurre:

← → × ⓘ localhost:8080/DWS/ejemplos/T2/tratarDatos2.php

localhost:8080 dice
Hola

Aceptar

¿Qué ha pasado? Acabamos de inyectar código javascript en nuestro archivo PHP.

Como seguramente ya sabrás, para insertar código javascript en nuestros HTML, tenemos que encerrarlo en las etiquetas **<script>...</script>**. La instrucción **alert('Hola')** muestra por pantalla el mensaje 'Hola'.

Vale, pero ¿porqué se ejecuta el código javascript en nuestro archivo PHP? Ni siquiera hay nada de código HTML en el archivo. La razón es muy simple. Acuérdate que el comando **echo** de PHP imprime **en el fichero** el texto. Si vemos el código en el navegador (botón derecho → ver código fuente o similar, dependiendo del navegador):

```
Nombre:Pepe<br>Apellidos:<script>alert('Hola')</script><br>Edad:34<br>
```

El intérprete PHP crea un archivo con código HTML, por lo tanto ejecutará el código javascript que haya en él.

En este caso el código no hace nada especialmente dañino, sólo muestra un mensaje por pantalla. Pero se pueden hacer cosas mucho más peligrosas ejecutando código javascript (o algún otro similar), como el robo de cookies para después poder hacerse pasar por él, o redirigir al usuario a una página creada por él donde pueda robarle la contraseña....

Este tipo de ataques reciben el nombre de **XSS (Cross-site scripting)**, y son una de las vulnerabilidades más explotadas en los sitios webs. Consiste en inyectar código malicioso en nuestra web mediante algún lenguaje del lado cliente (habitualmente javascript).

Tipos de ataque XSS

Existen varios tipo de ataque XSS, pero hay tres que son los principales:

- **No persistente o reflejado:** Un usuario envía datos al servidor a través, por ejemplo, de un formulario y el servidor genera una página dinámica usando esos datos. Si los datos contienen algún literal HTML, el servidor los inyectará en la página que le devuelve al usuario. Es el caso de nuestro ejemplo.

Puede parecer un ataque inútil, ya que, como en nuestro ejemplo, es el propio atacante el que inyecta código javascript en su HTML. Pero supongamos que el formulario enviamos los datos mediante GET. Un atacante podría sustituir uno de esos datos por el código malicioso y hacer que la víctima pinche en ese enlace mediante engaños.

- **Persistente o almacenado:** El código queda almacenado en el servidor (en una bbdd, por ejemplo), ejecutándose cada vez que un usuario accede a alguna página que muestre los datos. Por ejemplo, se puede inyectar código javascript en los comentarios de algún artículo de una tienda online. Cada vez que alguien accediera a ese artículo y cargara los comentarios se ejecutaría el código atacante.

Este tipo de ataque es más peligroso que el anterior, ya que las víctimas potenciales serían cualquier usuario que ingresara en el sitio web.

- **Basado en el DOM o XSS local:** En este caso son los scripts del cliente, el JavaScript por ejemplo en el navegador del usuario, el que construye elementos dinámicamente insertándolos en el DOM de la página. Con Javascript podemos acceder a ese árbol para insertar un elemento `<script>` con código malicioso. Su código se ejecutará cuando se visualice la página en el navegador de la víctima.

Prevenir ataques XSS

Como hemos dicho, nunca hay que fiarse de los datos que vienen de un usuario o cualquier otra fuente externa. Todos los datos deben ser validados en el servidor y escapados.

Afortunadamente, PHP (y todos los lenguajes de servidor) nos ofrecen herramientas sencillas para evitar este tipo de ataque (de todas formas, en el tema de seguridad nunca se puede estar del todo protegido. Siempre habrá alguna vulnerabilidad que pueda ser aprovechada por un atacante).

Para protegerse del XSS existen varias medidas que se pueden tomar, como **validación de datos, sanitización de dato y escapar las salidas de datos**.

Validación de datos

La validación de datos es el proceso de asegurarse que tu aplicación analiza el tipo de datos correctos. Si estas esperando un dato de tipo entero, debes rechazar cualquier otro tipo de datos.

Por ejemplo, si en el formulario existe un campo teléfono, deberíamos rechazar cualquier *string* que contenga caracteres que no sean dígitos. También podemos tener en cuenta la longitud que deberán tener esos dígitos (9, en este caso).

Sanitización de datos



En este caso, lo que hacemos es filtrar los datos de entrada. Podemos usar la función **strip_tags()** de PHP para realizar el filtrado. Esta función retira las etiquetas HTML y PHP de un *string*.

```
<?php  
  
$nombre = strip_tags($_POST['nombre']);  
$apellidos = strip_tags($_POST['apellidos']);  
$edad = strip_tags($_POST['edad']);  
  
echo "Nombre: $nombre<br>";  
echo "Apellidos: $apellidos<br>";  
echo "Edad: $edad<br>";
```

También podríamos crearnos nuestras propio código para eliminar las expresiones no deseadas, aunque es más sencillo utilizar las que nos ofrece PHP.

Escapar la salida de datos

Otro método de prevención es escapar los valores a la hora de mostrarlos. PHP tiene una función que convierte caracteres especiales en entidades HTML: **htmlspecialchars()**.

```
<?php  
  
$nombre = strip_tags($_POST['nombre']);  
$apellidos = strip_tags($_POST['apellidos']);  
$edad = strip_tags($_POST['edad']);  
  
echo "Nombre: $nombre<br>";  
echo "Apellidos: $apellidos<br>";  
echo "Edad: $edad<br>";
```

Existen librerías desarrolladas por terceros que nos pueden ayudar a automatizar estas tareas. Más adelante utilizaremos algunas de ellas para proteger nuestras webs de este tipo de ataques.

Otros tipos de ataques

SQL Injection

Parecido al XSS, pero en este caso estamos inyectando código SQL para manipular la bbdd del servidor. Se trata de otro de los ataques más frecuentes a sitios webs.

Veremos algunos ejemplos de ataque y cómo prevenirlos en el tema dedicado a la conexión con las bbdd.

CSRF

En este caso, el atacante realiza acciones sin estar logueado en la web a través de la víctima.

Este tipo de ataque es más difícil de realizar, pero mediante engaño se puede hacer que la víctima realice acciones en las webs vulnerables sin ser consciente de ello. Si la víctima tiene privilegios de administrador, el ataque puede hacer bastante daño.

Cuando lleguemos al tema de control de usuarios mediante sesiones, veremos este tipo de ataques y cómo prevenirlos.