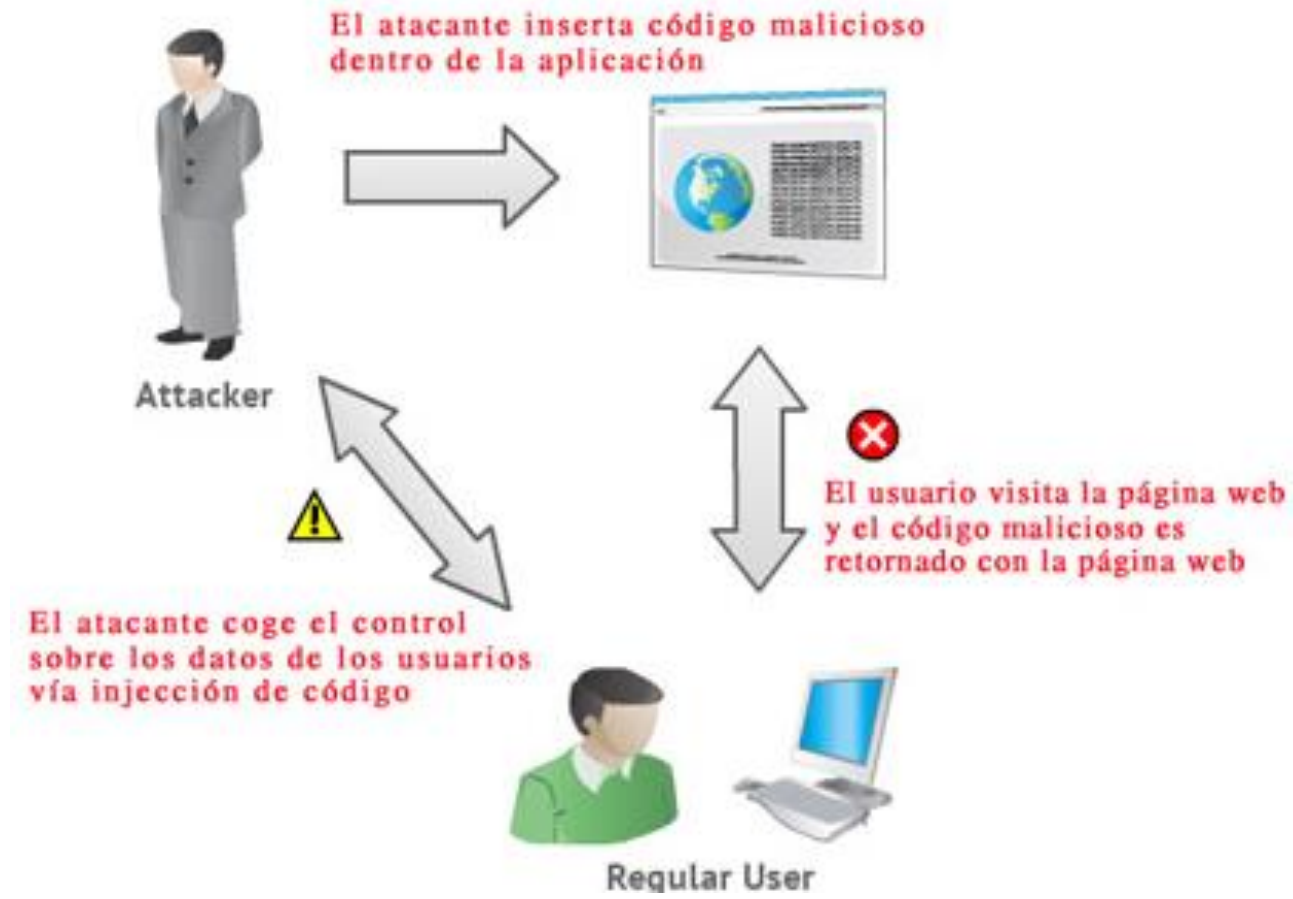
Seguridad en PHP

Cristina Santiago Piris
Miguel Ángel Flores Sirvent

ÍNDICE

1. ¿Qué es Cross-Site Scripting?

XSS ocurre cuando un atacante es capaz de inyectar un **script**, normalmente **JavaScript**, en el ***output*** de una aplicación web de forma que se ejecuta en el **navegador del cliente**. Los ataques se producen principalmente por **validar incorrectamente datos de usuario**, y se suelen inyectar mediante un **formulario web** o mediante un **enlace alterado**.



1.2 Ejemplo de ataque XSS.

Supongamos por ejemplo el siguiente **formulario**:

```
<form action="post.php" method="post">
        <input type="text" name="comment" value="">
        <input type="submit" name="submit" value="Submit">
<form>
```

En el formulario hay una caja de texto para datos de entrada y un botón de enviar. Una vez que el formulario es enviado, enviará los datos a post.php para procesarlos. Supongamos que lo único que se hará con los datos en post.php será mostrarlos con un echo:

```
echo $_POST['comment'];
```

1.2 Ejemplo de ataque XSS.

Sin ningún tipo de filtrado, el atacante puede enviar el siguiente script a través del formulario, lo que generará un **popup** en el navegador con el mensaje "Hackeado":

```
<script>alert("hacked")</script>
```

El script anterior tan sólo envía un mensaje de alerta, pero con Javascript ya hemos visto que se pueden hacer multitud de cosas que pueden dañar al usuario, especialmente relacionadas con el robo de cookies y sesiones o **usar una web para minar criptomonedas** usando la **API** de **CoinHive**!

1.3 Prevenir ataques XSS.

En PHP disponemos de algunas **funciones** para realizar la **prevención** de este tipo de ataques XSS: htmlspecialchars(), htmlentities() y strip_tags()

```
echo htmlspecialchars('<script>alert("El niño atacó esta web");</script>');
// resultado: &lt;script&gt;alert(&quot;El niño atacó esta web&quot;);&lt;/script&gt;

echo htmlentities('<script>alert("El niño atacó esta web");</script>');
// resultado: &lt;script&gt;alert(&quot;El ni&ntilde;o atac&oacute; esta
web&quot;);&lt;/script&gt;
```

Si queremos proteger el contenido que se almacena por ejemplo en una tabla, podemos hacer **limpieza de las etiquetas html** antes de almacenarlas utilizando la función strip-tags.

```
echo strip_tags('<script>alert("El niño atacó esta web");</script>');
// resultado que se almacenaría: alert("El niño atacó esta web");
```

2. Encriptado de contraseñas.

2.1 Importancia.

Cuando almacenamos en el servidor una contraseña sin encriptar, esta queda expuesta a que cualquier persona pueda leerla y usarla. Por lo que es un imperativo **encriptar** todo tipo de credenciales.

De esta manera, en caso de que alguien acceda a la base de datos solo dispondrá de una cadena de caracteres con la que no podrá hacer login a no ser que sepa la **clave de cifrado**.

Esto no es solo una cuestión técnica. También es una **cuestión legal**, ya que la **LOPD** impone la obligación de **almacenar las contraseñas de forma ininteligible**. (Art. 93.3).

2.2 Encriptado y Desencriptado.

El procedimiento que recomienda la **documentación oficial de PHP 7**, es usando la librería Hash.

```
$hash = password_hash('micontraseña', PASSWORD_DEFAULT, [15]);
```

Parámetros de la función **password_hash():**

1. Contraseña a cifrar.

2. Algoritmo a usar. **PASSWORD_DEFAULT** se actualiza siempre que se añada un algoritmo nuevo más fuerte.
   PASSWORD_BCRYPT emplea el algoritmo CRYPT_BLOWFISH.

3. Coste del algoritmo. Se puede considerar como el **número de veces que se encripta.**

2.2 Encriptado y Desencriptado.

Para poder verificar la contraseña que inserta el usuario al hacer login, se usa la función **password_verify(),** que devuelve un valor booleano.

```
if(password_verify($password, $hash)){
        // Password correcto!
}
```

Es **importante** destacar, que aunque la contraseña se almacene encriptada, se puede seguir ¨leyendo¨ en el momento en el que se envía al servidor, haciendo un ataque **Man In The Middle.** Por lo que para complementar esta medida de seguridad, sería recomendable implementar HTTPS en nuestro servidor. Y **forzar el uso de HTTPS** desde el fichero .htaccess insertando estas tres líneas.


```
RewriteEngine On
RewriteCond %{SERVER_PORT} 80
RewriteRule ^(.*)$ https://www.susitioweb.com/$1 [R,L]
```

3. SQL Injection.
3.1 ¿Qué es?.

Los **ataques SQL Injection** normalmente comienzan con el atacante introduciendo su código malicioso en un campo de **formulario específico en una aplicación.**

En lugar de introducir un dato válido, lo que haría el atacante sería **inyectar una sentencia SQL** que extraiga información, destruya o cambie algunos datos en la BD.

Esto no es solo una cuestión técnica. También es una **cuestión legal**, ya que la **LOPD** impone la obligación de **almacenar las contraseñas de forma ininteligible**. (Art. 93.3).

3.2 Demostración usando Kali Linux.

Damn Vulnerable Web Application (**DVWA**) es un conjunto de demostraciones e información sobre las vulnerabilidades más comunes de las aplicaciones Web.

Para probarlo, solo tenemos que descargar la maquina virtual de Metasploitable2.

Una vez arrancada, podemos acceder a las instrucciones poniendo la ip de la maquina en el navegador.

3.3 Como prevenir un ataque SQLi

Usar prepared statements y parameterized queries. Esto son sentencias SQL preparadas que se envían a la base de datos de forma separada a cualquier parámetro. De esta forma es imposible para un atacante **inyectar SQL malicioso.** Es la forma más recomendable y segura de evitar este tipo de ataques.

```
 $stmt = $pdo->prepare('SELECT * FROM usuarios WHERE nombre = :nombre');
$stmt->execute(array('nombre' => $nombre));

foreach ($stmt as $row) {
        // Hacer algo con $row
}
```

4. Subida de archivos mediante formulario.

Cuando se permite la **subida de archivos a usuarios en una aplicación web**, se abre la puerta a nuevas **posibilidades de hackeo**, pero suele ser un requisito en las aplicaciones de hoy en día. Subir archivos está disponible tanto en redes sociales como en cualquier aplicación con perfiles de usuario, blogs, foros, sitios de banca, etc. Los usuarios pueden subir imágenes, vídeos y muchos otros tipos de archivos más. Cuanta más libertad se deja al usuario más **aumentan los riesgos**. Aun así, **un gran número de sitios web no disponen de las medidas de seguridad adecuadas.**

4. Subida de archivos mediante formulario.

La siguiente lista de medidas ayuda a prevenir ataques en la subida de archivos:

- Define un archivo .htaccess que sólo permita el acceso a archivos con extensiones específicas.
- No pongas el .htaccess en la misma carpeta donde se suben los archivos, se debe situar en el directorio padre.

Ejemplo de .htacces para permitir solo imagenes **gif, png, jpg, jpeg y png** en la carpeta de subidas:

```
deny from all
        <Files ~ “^w+.(gif|jpe?g|png)$”>
                order deny,allow
                allow from all
        </Files>
```

5. Bibliografía.

https://manuais.iessanclemente.net/index.php/Evitar_ataques_XSS_y_CSRF_con_PHP

https://diego.com.es/

https://portal.hostingdepago.com/knowledgebase/126/Forzar-el-uso-de-HTTPS-desde-htaccess.html

https://www.youtube.com/watch?v=KmbMtiOfTnQ