CONSEJOS DE SEGURIDAD CON PHP

Cuando desarrollamos aplicaciones web en entorno servidor con PHP y bases de datos como MySQL, no basta con que el código “funcione”: también debe ser seguro.

Cada formulario, cada conexión con la base de datos y cada sesión de usuario representa una posible puerta de entrada para ataques si no se gestionan correctamente.

Los fallos de seguridad más comunes, como la exposición de contraseñas, las inyecciones de código o la manipulación de sesiones, suelen tener su origen en una falta de validación o en una mala práctica de programación.

Por eso, la seguridad no debe entenderse como una fase final del desarrollo, sino como una parte integral del diseño y la escritura del código desde el primer momento.

En PHP, esto significa aprender a tratar los datos del usuario con desconfianza, proteger la comunicación con la base de datos, y controlar con precisión qué información se almacena, se muestra o se transmite. Adoptar una mentalidad de “seguridad por defecto” permite construir aplicaciones más robustas, resistentes a errores humanos y preparadas frente a los ataques más habituales en la web actual.

Algunas de las implementaciones de seguridad más recomendables son las siguientes:

1. Almacén seguro de contraseñas

¡NO ALMACENEMOS LAS CONTRASEÑAS EN TEXTO PLANO EN LA BASE DE DATOS!

Si lo hiciéramos, cualquier persona con acceso a la base de datos podría leer directamente las contraseñas de los usuarios, algo que constituye una brecha de seguridad grave.
En su lugar, debemos **almacenar un *hash* de la contraseña**, no la contraseña en sí.

Para ello, PHP nos ofrece funciones específicas que se encargan de aplicar un algoritmo seguro de hash:

password_hash($contrasena, PASSWORD_DEFAULT)

Esta función genera un hash seguro de la contraseña que luego guardaremos en la base de datos.

$hash = password_hash($contrasena, PASSWORD_DEFAULT);

- El primer parámetro es la contraseña original (sin procesar).
- El segundo parámetro indica el algoritmo de hash.
  El valor PASSWORD_DEFAULT usa el algoritmo más seguro disponible en tu versión de PHP (actualmente bcrypt o Argon2).

- PHP se encarga automáticamente de generar un *salt* aleatorio (una cadena aleatoria que refuerza el hash) y de incluirlo dentro del resultado, así que no necesitamos hacerlo manualmente.

El valor devuelto por password_hash() es una cadena que representa el hash seguro, y es lo que debemos almacenar en la base de datos.

Nunca almacenamos la contraseña original ni tratamos de "desencriptarla": el proceso no es reversible.

---

password_verify($contrasena, $hash)

Cuando el usuario intenta iniciar sesión, debemos comprobar si la contraseña introducida coincide con el hash almacenado.

```
if (password_verify($contrasenaIntroducida, $hashGuardado)) {
    echo "Contraseña correcta";
} else {
    echo "Contraseña incorrecta";
}
```

Esta función compara la contraseña original (la que introduce el usuario) con el hash almacenado y devuelve true si coinciden o false en caso contrario.

2. Ataque de Inyección de Código (XSS)

El ataque de inyección de código (conocido también como Cross-Site Scripting o XSS) ocurre cuando una aplicación web no controla adecuadamente los datos que recibe de los usuarios y acaba interpretándolos como código.

Supongamos que tenemos un formulario de comentarios en una página web. Si el desarrollador no filtra ni escapa los datos introducidos, un usuario malintencionado podría enviar un código JavaScript en lugar de un simple comentario:

```
<script>alert('¡Hola, soy un atacante!');</script>
```

Una vez hecho esto, el comentario se guardará sin más en la base de datos, y si otro usuario pide por ejemplo mostrar todos los comentarios, su navegador ejecutará ese código JavaScript al visualizar la página.

Aunque el ejemplo muestra solo un mensaje de alerta, el atacante podría hacer cosas mucho peores, como robar cookies de sesión o redirigir al usuario a una página falsa.

Prevención con htmlspecialchars()

La manera más sencilla y eficaz de evitar este problema es **escapar los caracteres especiales** antes de mostrar cualquier dato introducido por el usuario.

La función htmlspecialchars() en PHP es una función de saneamiento que convierte caracteres especiales en entidades HTML. Esto significa que convierte caracteres como <, >, &, " y ' en sus entidades HTML equivalentes, lo que evita que el navegador interprete estos caracteres como parte del código HTML o JavaScript.

Por ejemplo, si un usuario malintencionado ingresa el siguiente texto en un campo de formulario:

<script>alert('¡Hola, soy un atacante!');</script>

La función ***htmlspecialchars()*** convertirá los caracteres especiales en entidades HTML, lo que resultará en:

<script>alert('¡Hola, soy un atacante!');</script>

Cuando este texto escapado se muestra en la página web, se verá como texto plano y no se ejecutará como código JavaScript.

Este mecanismo es fundamental cuando mostramos datos en HTML que provienen de fuentes no confiables (como formularios, parámetros de URL o datos de una base de datos).

3. Evitar Inyección SQL (SQL Injection) -> La estudiamos solo de forma teórica

Otro tipo de ataque muy común ocurre cuando el atacante intenta manipular una consulta SQL a través de los datos enviados por un formulario o una URL.
Por ejemplo, imagina este código:

$usuario = $_POST['usuario'];

$resultado = mysqli_query($conexion, "SELECT * FROM usuarios WHERE nombre = '$usuario'");

Si el atacante introduce en el campo usuario algo como:

' OR '1'='1

La consulta generada sería:

SELECT * FROM usuarios WHERE nombre = '' OR '1'='1';

Esto hace que la condición siempre sea verdadera y que se devuelvan **todos los registros de la tabla**. En casos peores, el atacante podría modificar o eliminar datos.

Prevención con consultas preparadas (PDO)

La forma correcta de evitarlo es **no concatenar variables directamente en las consultas SQL**, sino usar **consultas preparadas**, donde PHP envía los datos y la estructura de la consulta por separado.

Ejemplo usando PDO:

$pdo = new PDO('mysql:host=localhost;dbname=mi_db;charset=utf8mb4', 'usuario', 'clave');

$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

$stmt = $pdo->prepare("SELECT * FROM usuarios WHERE nombre = :nombre");*

$stmt->execute(['nombre' => $_POST['usuario']]);

$usuario = $stmt->fetch();

En este caso, el valor de $_POST['usuario'] **nunca se interpreta como código SQL**, sino como un simple dato.

Se envía al SGBD sólo la *plantilla*** de la consulta con marcadores (placeholders), p. ej. SELECT * FROM usuarios WHERE nombre = ? o ... WHERE nombre = :nombre. El SGBD analiza y compila esa plantilla sin conocer aún los valores concretos.

De esta manera, incluso si alguien intenta inyectar ' OR '1'='1, se tratará como texto y no afectará la consulta.