Proyecto: Sistema de Autenticación Web Segura

Memoria técnica — DAW 2.º Curso

Alumno: Aaron Fernández

Módulo: Desarrollo Web en Entorno Servidor

Tecnologías: PHP 8 · PostgreSQL · PDO · JavaScript

Dato	Valor
Alumno	Aaron Fernández
Curso	2.º DAW — 2025/2026
Módulo	Desarrollo Web en Entorno Servidor
Servidor	iac.mooo.com (hosting del instituto)
BD	PostgreSQL 17 — s24aaron_login_aw

1. Introducción

En este proyecto he creado un sistema de login completo para una aplicación web. El objetivo no era solo que el login funcionara, sino que fuera seguro de verdad. He partido de una versión básica en PHP con PostgreSQL y la he reescrito desde cero aplicando las medidas de seguridad que hemos visto en el módulo.

Durante el desarrollo me he encontrado con problemas reales en el servidor del instituto (permisos de base de datos, incompatibilidades de cookies, librerías bloqueadas) que he tenido que ir resolviendo uno a uno. Eso me ha enseñado más que cualquier ejercicio en local.

2. Estructura del Proyecto

Archivo	Función
bbdd.php	Conexión PDO a PostgreSQL
csrf.php	Generación y validación de tokens CSRF
session_check.php	Verificación de sesión en páginas protegidas
auth_login.php	Lógica de autenticación
auth_registro.php	Lógica de registro de usuarios
bienvenida.php	Página protegida tras el login
logout.php	Cierre de sesión completo
login.html	Formulario de login
registro.html	Formulario de registro
style.css	Estilos de la aplicación
schema.sql	Estructura de la base de datos

3. Base de Datos PostgreSQL

Uso PostgreSQL porque es el motor disponible en el servidor del instituto. He creado dos tablas: una para los usuarios y otra para controlar los intentos fallidos de login.

```
CREATE TABLE IF NOT EXISTS usuarios (
id SERIAL PRIMARY KEY,
nombre_usuario VARCHAR(30) NOT NULL UNIQUE,
password VARCHAR(255) NOT NULL, -- hash bcrypt
creado_en TIMESTAMP DEFAULT NOW()
);
```

```
CREATE TABLE IF NOT EXISTS login_intentos (
clave VARCHAR(64) NOT NULL PRIMARY KEY, -- MD5(ip+usuario)
intentos SMALLINT NOT NULL DEFAULT 1,
ultimo_intento TIMESTAMP DEFAULT NOW()
);
```

La tabla **login_intentos** es la que permite el sistema anti-fuerza-bruta. Guardo como clave el MD5 de la IP más el nombre de usuario, así el bloqueo es por combinación IP+usuario y no solo por IP.

4. Conexión Segura con PDO

He migrado de la extensión nativa pg_query_params() a PDO (PHP Data Objects). PDO funciona con cualquier motor de base de datos y obliga a usar consultas preparadas, que son la protección real contra la inyección SQL.

```
// bbdd.php
$dsn = "pgsql:host=localhost;port=5432;dbname=s24aaron_login_aw";
$options = [
PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
PDO::ATTR_EMULATE_PREPARES => false, // Prepara en el servidor, no en PHP
];
$pdo = new PDO($dsn, "s24aaron_loginaw", "250608", $options);
```

El parámetro **ATTR_EMULATE_PREPARES => false** es importante: hace que las queries se preparen realmente en el servidor PostgreSQL. Así el motor trata los valores del usuario siempre como datos, nunca como código SQL.

5. Medidas de Seguridad

Ataque	Solución	Archivo
SQL Injection	Consultas preparadas PDO	bbdd.php + auth_*.php
Fuerza bruta	Bloqueo 5 intentos / 15 min	auth_login.php
Session Fixation	session_regenerate_id(true)	auth_login.php
Session Hijacking	IP + User-Agent en sesión	session_check.php
CSRF	Token aleatorio 32 bytes	csrf.php
XSS	htmlspecialchars() + CSP	bienvenida.php
Password en claro	SHA-256 cliente + bcrypt servidor	login.html + auth_*.php
Sesión abandonada	Timeout 30 min inactividad	session_check.php
Info leakage	Mensajes de error genéricos	Todos los PHP

5.1 Inyección SQL

Con consultas preparadas, el valor del usuario nunca se mezcla con el SQL. Se envían por separado al servidor y PostgreSQL los trata siempre como datos:

```
// Sin preparadas (VULNERABLE):
$query = "SELECT * FROM usuarios WHERE nombre_usuario = '$usuario'";
// Si $usuario = "admin' OR '1'='1" --> entra sin contraseña

// Con PDO (SEGURO):
$stmt = $pdo->prepare("SELECT * FROM usuarios WHERE nombre_usuario = ?");
$stmt->execute([$usuario]); // El ? nunca se interpreta como SQL
```

5.2 Anti Fuerza Bruta

Tras 5 intentos fallidos con la misma combinación IP+usuario, bloqueo el acceso durante 15 minutos. Uso ON CONFLICT de PostgreSQL para actualizar el contador en una sola query:

```
$stmt = $pdo->prepare("
INSERT INTO login_intentos (clave, intentos, ultimo_intento)
VALUES (?, 1, NOW())
ON CONFLICT (clave) DO UPDATE
SET intentos = login_intentos.intentos + 1,
ultimo_intento = NOW()
");
$stmt->execute([$bloqueoClave]);
```

5.3 Protección de Sesión

Aplico tres capas de protección sobre las sesiones PHP:

• **Session Fixation:** llamo a session_regenerate_id(true) justo después del login. Así aunque el atacante conozca el ID anterior, ya no vale.

• **Session Hijacking:** guardo la IP y el User-Agent del navegador al hacer login. En cada página protegida compruebo que coinciden. Si alguien roba la cookie y la usa desde otro ordenador, la sesión se destruye automáticamente.

• **Timeout:** si el usuario lleva 30 minutos sin actividad, la sesión expira.

```
// session_check.php
if ($_SESSION['ip'] !== $_SERVER['REMOTE_ADDR']) {
session_destroy();
header("Location: login.html?error=ip");
exit();
}
if ((time() - $_SESSION['ultima_actividad']) > 1800) {
session_destroy();
header("Location: login.html?error=timeout");
exit();
}
```

5.4 CSRF

Genero un token aleatorio de 32 bytes al cargar el formulario. El JavaScript lo pide a csrf.php y lo incluye en cada petición POST. Si una web maliciosa intenta enviar un formulario en nombre del usuario, no tendrá el token y el servidor rechazará la petición:

```
// csrf.php
$_SESSION['csrf_token'] = bin2hex(random_bytes(32));

// registro.html (JavaScript)
const res = await fetch('csrf.php?action=get_token', {credentials: 'same-origin'});
const { csrf_token } = await res.json();
formData.append('csrf_token', csrf_token); // Se envía como campo POST

// auth_registro.php
if (!hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'])) {
echo json_encode(["status" => "error", "message" => "Token inválido"]);
exit;
}
```

5.5 Cifrado de Contraseñas

La contraseña nunca viaja en texto plano ni se guarda sin cifrar. Uso dos capas:

Capa	Dónde	Por qué
SHA-256	Navegador (JS, CryptoJS)	La contraseña viaja hasheada. Ni con sniffing se ve en claro.
bcrypt coste 12	Servidor (PHP)	El hash SHA-256 se re-hashea con bcrypt. Es lento a propósito: dificulta ataques offline.

```
// registro.html — SHA-256 en cliente
const passHash = CryptoJS.SHA256(password).toString();
```

```
// auth_registro.php — bcrypt en servidor
$hash = password_hash($passHash, PASSWORD_BCRYPT, ['cost' => 12]);

// auth_login.php — verificar sin desencriptar
if (password_verify($passHash, $user['password'])) { /* OK */ }
```

6. Problemas Encontrados en el Servidor

Durante el despliegue en el servidor del instituto me encontré con varios problemas que no aparecen cuando desarrollas en local. Los explico aquí porque creo que son la parte más interesante del proyecto.

Sin permisos para crear la BD

El usuario del hosting no tenía permisos para ejecutar CREATE DATABASE. La base de datos la tuvo que crear el administrador. Yo solo pude crear las tablas dentro.

Archivo db.php en caché

Subí un db.php actualizado pero el servidor seguía usando el antiguo. Lo resolví renombrando el archivo a bbdd.php y actualizando todos los includes.

Hash de integridad de CryptoJS incorrecto

El atributo integrity del script tenía un hash SHA-512 erróneo. El navegador bloqueaba la librería y el SHA-256 nunca se ejecutaba, por lo que las validaciones del servidor fallaban. Lo resolví quitando el atributo integrity.

Validaciones de contraseña en servidor fallaban

El servidor validaba que la contraseña tuviera mayúsculas, pero recibía el hash SHA-256 (todo minúsculas hexadecimales). Lo corregí moviendo esas validaciones solo al cliente y en el servidor solo verifico que sea un hash válido de 64 chars.

Cookies con SameSite=Strict

En el hosting del instituto las cookies con SameSite=Strict causaban que la sesión no se mantuviera entre peticiones. Lo cambié a SameSite=Lax, que es compatible con más configuraciones de servidor.

7. Conclusiones

Este proyecto me ha enseñado que la seguridad no es una sola cosa, sino muchas capas que trabajan juntas. Puedes tener bcrypt y aun así ser vulnerable al CSRF. Puedes tener CSRF y aun así tener session hijacking. Hay que pensar en todo a la vez.

Lo que más me ha sorprendido es que los problemas más difíciles no fueron los de seguridad, sino los del servidor real: permisos, cachés, configuraciones distintas a local. Eso es algo que no se aprende hasta que no lo despliegas de verdad.

Si tuviera que añadir algo más al proyecto, implementaría verificación por email al registrarse y autenticación de dos factores (2FA), que son los siguientes pasos naturales después de lo que ya hay.