INFORME TÉCNICO & AUDITORÍA DE CÓDIGO

Proyecto CORDAMI

Evaluación de Vulnerabilidades Lógicas y Arquitectura Backend

Enfoque: Backend PHP & DB **Realizado por:** Equipo de Tecnología de la FIIIDT **Fecha:** Agosto 2026

1. Resumen Ejecutivo

El presente documento detalla el análisis del código fuente del proyecto (específicamente el backend en PHP). Se han identificado vulnerabilidades lógicas críticas relacionadas con el manejo de bases de datos, transacciones y arquitectura general que comprometen seriamente la integridad y estabilidad de los datos.

2. Hallazgos y Errores Críticos

2.1. Ruptura de Transacciones por `ALTER TABLE` (Implicit Commit)

SEVERIDAD: CRÍTICA

Ubicación: `guardar_censo.php` (Línea 89), `guardar_cedula.php` (Línea 60)

Problema: Se inicia una transacción ACID (`$mysqli->begin_transaction();`) para agrupar inserciones complejas. Sin embargo, en medio de la transacción, el código ejecuta un `ALTER TABLE`.

Impacto: En los motores de bases de datos relacionales (como MariaDB/MySQL), ejecutar comandos DDL (*Data Definition Language*) provoca un **commit implícito e irreversible**. Si una consulta posterior falla (por ejemplo, al guardar el predio), el sistema intentará ejecutar un `rollback()`, pero este fallará silenciosamente, dejando datos incompletos y corrompiendo la integridad relacional del sistema.

2.2. Modificación del Esquema en Tiempo de Ejecución

SEVERIDAD: ALTA

Ubicación: Consultas `SHOW COLUMNS` y `ALTER TABLE` en los endpoints de la API

Problema: En cada solicitud POST enviada por los usuarios, el código verifica si existe una columna en la base de datos y, de no existir, intenta crearla dinámicamente.

Impacto:

- **Rendimiento:** Genera una carga innecesaria en la base de datos por cada petición ejecutada.
- **Concurrencia (Condición de Carrera):** Si múltiples usuarios envían el formulario simultáneamente, múltiples hilos intentarán alterar la tabla al mismo tiempo, lo que puede bloquear las tablas o provocar caídas del servidor de base de datos.

2.3. Ausencia de Validación de Datos y Sanitización

SEVERIDAD: ALTA

Problema: El código asume que el objeto JSON entrante tiene el formato y tipo de datos correcto. Utiliza operadores `??` de forma débil (ej. `$edad = (int)($data['edad'] ?? 0);`).

Impacto: Un usuario malintencionado o un error en el frontend podría enviar datos vacíos, textos en lugar de números o formatos inválidos. El sistema guardará ceros (`0`) o cadenas vacías (`''`) en lugar de devolver un código HTTP `400 Bad Request`, llenando la base de datos de información inconsistente e inservible.

2.4. Credenciales Hardcodeadas y Acoplamiento

SEVERIDAD: MEDIA / ALTA

Problema: Las credenciales de la base de datos se encuentran expuestas como valores por defecto en el código fuente dentro de la función `getEnvVar()`. Se observa específicamente el siguiente bloque de código:

```
$host     = getEnvVar('DB_HOST', 'localhost');
$db       = getEnvVar('DB_NAME', 'cordamim_form_carnet');
$user     = getEnvVar('DB_USER', 'cordamim_math');
$password = getEnvVar('DB_PASS', 'Cordamim_math01');
```

Impacto: Dificulta gravemente el mantenimiento y expone credenciales sensibles si el código fuente es compartido. Si la contraseña de la BD cambia, hay que modificar múltiples archivos de forma manual, ya que esta lógica de conexión está duplicada en varios archivos (`guardar_censo.php` y `guardar_cedula.php`).

2.5. Lógica de Actualización Defectuosa (`ON DUPLICATE KEY`)

SEVERIDAD: ALTA

Problema: En la tabla `usuarios`, al detectar un usuario existente, la cláusula `ON DUPLICATE KEY UPDATE` está incompleta: solo ejecuta `id = LAST_INSERT_ID(id)`.

Impacto: Si un usuario existente intenta actualizar sus datos, la lógica del sistema simulará que la operación fue exitosa, pero en la base de datos no se aplicará ningún cambio a campos críticos como la contraseña o información del perfil.

3. Soluciones Inmediatas (Mantenimiento en PHP puro)

Si el objetivo es salir a producción rápidamente con este mismo código, se deben aplicar las siguientes correcciones obligatorias:

1. **Eliminar los `ALTER TABLE` y `SHOW COLUMNS`:** Trasladar cualquier cambio de estructura de base de datos al script `cordamim_form_carnet.sql`. Las tablas deben estar previamente creadas antes de encender la API.
2. **Centralizar la conexión:** Crear un archivo `conexion.php` que maneje el objeto `$mysqli` de forma segura mediante un archivo `.env` real, e incluirlo (`require_once`) en el resto de los scripts.
3. **Validar Entradas:** Validar explícitamente que los campos obligatorios existan y tengan el tipo correcto (usando `filter_var` y `is_numeric`) antes de iniciar cualquier transacción.
4. **Corregir el Update:** Completar todos los campos requeridos en las cláusulas `ON DUPLICATE KEY UPDATE` para garantizar que la información se actualice de forma efectiva.

4. Recomendación Arquitectónica: ¿Arreglar o Refactorizar?

> **Veredicto Tecnológico: Es Altamente Recomendable Refactorizar**
>
> El código actual está construido de forma procedimental usando PHP puro ("Vanilla PHP") como un script monolítico de más de 300 líneas por archivo. A medida que el proyecto CORDAMI escale, este enfoque generará altos costos de mantenimiento e inestabilidad en producción.

Desventajas del enfoque actual:

- **Ausencia de separación de responsabilidades (MVC):** Las peticiones HTTP, la lógica de negocio y la capa de base de datos están mezcladas en un solo archivo.
- **Consultas SQL directas:** La concatenación e inserción directa (`INSERT INTO...`) es propensa a errores humanos, dificulta el mantenimiento y complica el seguimiento de relaciones complejas (Predios, Actividades Dinámicas y Productores).

Opciones de Refactorización

Si el equipo de desarrollo tiene la disponibilidad, se sugiere migrar hacia un framework moderno:

> **Opción A: Framework PHP (Laravel) — Recomendado**
>
> **Ventajas:** Laravel soluciona automáticamente casi todos los problemas encontrados: maneja *Migraciones* (eliminando el uso de `ALTER TABLE` dinámico), incluye *Eloquent ORM* para manejar transacciones e inserciones relacionales de forma segura y cuenta con un motor de validación de datos integrado y robusto.

> **Opción B: Stack JavaScript / TypeScript (Node.js + Express + Prisma)**
>
> **Ventajas:** Si el frontend está basado en JavaScript (`script.js`), migrar el backend a Node.js permite utilizar un solo lenguaje en todo el proyecto. Prisma ORM hace que el manejo de bases de datos relacionales sea fuertemente tipado y sumamente sencillo.

Conclusión Final: Para *pruebas inmediatas*, aplique las soluciones del apartado 3 (eliminar los `ALTER TABLE`). Para *producción y escalabilidad*, se recomienda encarecidamente programar una refactorización hacia Laravel o Node.js; el tiempo invertido se recuperará rápidamente al evitar incidentes recurrentes de mantenimiento y corrupción de datos.