Reporte de Vulnerabilidad

Archivo: conexion.php

Código Analizado:

```
<?php
$host = "localhost";
$usuario = "root";
$password = "";
$base_datos = "crud_php";

// Crear conexión
$conexion = new mysqli($host, $usuario, $password, $base_datos);

// Verificar errores de conexión
if ($conexion->connect_error) {
    die("Error de conexión: " . $conexion->connect_error);
}

// Establecer conjunto de caracteres a UTF-8
if (!$conexion->set_charset("utf8mb4")) {
    die("Error al establecer el conjunto de caracteres: " . $conexion->error);
}

// Opcional: mensaje de éxito (solo en desarrollo)
/* echo "? Conexión exitosa a la base de datos."; */
?>
```

Análisis: ```html
Análisis de Vulnerabilidades de Seguridad

Posibles Vulnerabilidades

El código proporcionado, aunque aparentemente sencillo, presenta algunas vulnerabilidades de seguridad, especialmente en un contexto de producción. La ausencia de validación y sanitización de entradas en queries posteriores es una vulnerabilidad potencial, aunque no se refleja directamente en este fragmento.

- **Vulnerabilidad:** Credenciales de la base de datos expuestas.
  Líneas: `$usuario = "root";`, `$password = "";`
  Descripción: Almacenar el nombre de usuario y la contraseña de la base de datos directamente en el código es una práctica extremadamente insegura. Si el código es accedido por personas no autorizadas, la base de datos entera está comprometida. El uso de "root" sin contraseña agrava el problema.
  Mitigación:
    - Nunca usar "root" para las aplicaciones. Crea un usuario con los mínimos privilegios necesarios.
    - Almacena las credenciales de la base de datos en variables de entorno o en un archivo de configuración externo al directorio público del servidor web. Estos archivos deben tener permisos restringidos (por ejemplo, 600).
    - Utiliza un sistema de gestión de secretos (Secret Manager) para entornos de producción.
  Mejora métricas: No aplicable. Esta es una práctica de seguridad, no de calidad del código.
- **Vulnerabilidad:** Falta de manejo de excepciones robusto.
  Líneas: Manejo de errores básico con `die()`.
  Descripción: Si bien el código incluye comprobaciones de errores para la conexión y el conjunto de caracteres, el uso de `die()` interrumpe la ejecución del script y muestra mensajes de error directamente en la página. En un entorno de producción, esto puede revelar información sensible y ofrecer una mala experiencia al usuario.
  Mitigación:
    - Implementa un sistema de registro de errores (logging) para registrar los errores en un archivo.
    - Maneja las excepciones de forma más elegante, mostrando mensajes de error genéricos al usuario y registrando los detalles del error internamente.

    - Utiliza bloques `try-catch` para capturar excepciones y realizar acciones correctivas.

  Mejora métricas: No aplicable. Esta es una práctica de seguridad, aunque una mejora en el manejo de errores incrementaría la mantenibilidad.
- **Vulnerabilidad (Potencial):** Inyección SQL (SQLi).
  Líneas: El código actual no muestra la vulnerabilidad, pero es vulnerable si se usan variables externas (GET, POST) sin validación en queries.
  Descripción: Si este código se utiliza para realizar consultas a la base de datos utilizando datos proporcionados por el usuario (por ejemplo, a través de formularios), es vulnerable a ataques de inyección SQL. Los atacantes pueden manipular las consultas para acceder a datos no autorizados, modificar la base de datos o incluso ejecutar comandos en el servidor.
  Mitigación:
    - **Utiliza consultas preparadas (prepared statements) con parámetros en lugar de concatenar cadenas SQL.** Las consultas preparadas escapan automáticamente los caracteres especiales, previniendo la inyección SQL.
    - Valida y sanitiza todas las entradas del usuario antes de utilizarlas en consultas SQL. Utiliza funciones como `filter_var()` y `htmlspecialchars()` para eliminar o escapar caracteres peligrosos.
    - Aplica el principio de mínimo privilegio: el usuario de la base de datos utilizado por la aplicación debe tener solo los permisos necesarios para realizar las operaciones requeridas.

  Mejora métricas: No aplicable. Esta es una práctica de seguridad.

Métricas de Calidad del Código

Análisis de la calidad del código

El código proporcionado es bastante sencillo, por lo que las métricas son generalmente buenas, pero se pueden mejorar.

- **Complejidad:** Baja.
  Descripción: El código tiene una baja complejidad ciclomática ya que solo realiza tareas básicas de conexión a la base de datos.
  Mejora: No requiere mejora en este caso, dado que la función del código es simple.
- **Duplicación:** Nula.
  Descripción: No hay duplicación de código en este fragmento.
  Mejora: No requiere mejora.
- **Legibilidad:** Buena.
  Descripción: El código es fácil de leer y entender, con nombres de variables descriptivos y comentarios claros.
  Mejora: Se puede agregar más documentación y comentarios si el código se vuelve más complejo. Consistencia en la documentación (estilo phpDoc).
- **Acoplamiento:** Alto.
  Descripción: El código está fuertemente acoplado a la base de datos. Cualquier cambio en la base de datos requerirá modificaciones en el código.
  Mejora:
    - Implementa una capa de abstracción de datos (Data Access Layer - DAL) para desacoplar el código de la base de datos.
    - Utiliza un patrón de diseño como Repository o Data Mapper para aislar la lógica de acceso a datos.

Solución Propuesta

Recomendaciones para un código más seguro y mantenible

A continuación, se presenta un código revisado que aborda las vulnerabilidades y mejora las métricas de calidad.

```
<?php
```

```
// Cargar configuración desde variables de entorno o archivo de configuración
$host = $_ENV["DB_HOST"] ?? "localhost";
$usuario = $_ENV["DB_USER"] ?? "mi_usuario";
$password = $_ENV["DB_PASSWORD"] ?? "mi_password";
$base_datos = $_ENV["DB_NAME"] ?? "crud_php";

// Intentar conexión con manejo de excepciones
try {
    $conexion = new mysqli($host, $usuario, $password, $base_datos);

    // Verificar errores de conexión
    if ($conexion->connect_error) {
        throw new Exception("Error de conexión: " . $conexion->connect_error);
    }

    // Establecer conjunto de caracteres a UTF-8
    if (!$conexion->set_charset("utf8mb4")) {
        throw new Exception("Error al establecer el conjunto de caracteres: " . $conexion->error);
    }

    // Opcional: mensaje de éxito (solo en desarrollo, nunca en producción)
    // error_log("Conexión exitosa a la base de datos.");

} catch (Exception $e) {
    // Registrar el error en un archivo de log
    error_log($e->getMessage(), 0);

    // Mostrar un mensaje genérico al usuario
    die("Error al conectar con la base de datos. Por favor, contacte al administrador.");
} finally {
    // No cerramos la conexión aquí, ya que probablemente se necesite más adelante.
    // La conexión se debe cerrar en el lugar donde ya no se necesite.
}

?>
```

- **Cambios:**
    - Uso de variables de entorno para las credenciales de la base de datos.
    - Manejo de excepciones con `try-catch` para una mejor gestión de errores.
    - Registro de errores en un archivo de log en lugar de mostrar mensajes sensibles al usuario.
    - Eliminación del mensaje de éxito en producción.
- **Importante:** Este código solo mejora la seguridad de la conexión. Para mitigar la inyección SQL, es fundamental usar consultas preparadas y validar las entradas en el código que *usa* esta conexión.

```
```