Vulnerability Report

Archivo: conexion.php

Code Analyzed:

```
<?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."; */
?>
```

Analysis: ```html
Security Vulnerabilities

Hardcoded Credentials

Type: Sensitive Information Exposure

Approximate Line: 2-4

Description: The code contains hardcoded database credentials (username "root" and an empty password). This is a major security risk. Anyone with access to the code can use these credentials to access and potentially modify or delete the database. The localhost host is also a bad practice in production.

Mitigation: Store database credentials securely outside the codebase, such as in environment variables or a configuration file with restricted access. Retrieve these credentials using functions like `getenv()` or by parsing the configuration file. Use unique, strong passwords for each database user. Never commit sensitive information to version control. Avoid "root" user when it is not required.

Improvement: Replace hardcoded values with environment variables or configuration file values loaded at runtime. For example, use `getenv("DB_USER")` after setting the `DB_USER` environment variable.

Missing Input Validation & Prepared Statements

Type: SQL Injection (Potential)

Approximate Line: This code connects to the database, but does not show any query generation. A SQL injection vulnerability is present if this code were to use user-provided data to build SQL queries (which is common with CRUD operations).

Description: While this code snippet only handles the database connection, the typical CRUD operations it supports are highly susceptible to SQL Injection vulnerabilities if user input is not properly validated and sanitized, or if parameterized queries/prepared statements are not used.

Mitigation: Always use prepared statements (parameterized queries) to execute SQL queries. Never directly concatenate user input into SQL queries. Validate and sanitize all user input before using it in any SQL query. Use appropriate encoding functions (e.g., `htmlspecialchars()` for HTML output) to prevent Cross-Site Scripting (XSS) vulnerabilities.

Improvement: Refactor the code to use prepared statements for all database queries. Implement robust input validation to check data types, lengths, and formats.

Code Quality Metrics

Readability

Description: The code is relatively readable due to its clear and concise structure, well-chosen variable names (in Spanish), and comments. However, the comments could be more informative about the purpose of each section.

Improvement: Add more detailed comments explaining the overall purpose of the script and the functionality of each code block. Follow consistent naming conventions. Consider using English for variable and function names in a collaborative setting.

Coupling

Description: The current code snippet has low coupling, as it only deals with database connection. If other parts of the application directly use this connection code, it can increase coupling.

Improvement: Encapsulate the database connection logic into a separate class or function. This allows other parts of the application to access the connection through a well-defined interface, reducing direct dependencies and improving maintainability. Consider using a Database Abstraction Layer (DAL) for improved decoupling and portability.

Complexity

Description: The code has low cyclomatic complexity, as it consists of simple conditional statements and function calls.

Improvement: For this specific snippet, no immediate improvement is necessary. If more logic is added later, consider refactoring complex conditional blocks into separate functions or using design patterns to manage complexity.

Duplication

Description: The code snippet itself does not contain significant duplication. However, if the database connection code is repeated across multiple files, it represents duplication.

Improvement: As mentioned previously, encapsulating the connection logic in a function will avoid such code duplication.

Proposed Solution

1. **Remove Hardcoded Credentials:** Replace `$host`, `$usuario`, `$password`, and `$base_datos` with environment variables (e.g., `getenv('DB_HOST')`).
2. **Database Abstraction:** Encapsulate the connection logic within a database class or function. ```php host = getenv('DB_HOST'); $this->usuario = getenv('DB_USER'); $this->password = getenv('DB_PASS'); $this->base_datos = getenv('DB_NAME'); } public function connect() { $this->conexion = new mysqli($this->host, $this->usuario, $this->password, $this->base_datos); if ($this->conexion->connect_error) { die("Error de conexión: " . $this->conexion->connect_error); } if (!$this->conexion->set_charset("utf8mb4")) { die("Error al establecer el conjunto de caracteres: " . $this->conexion->error); } return $this->conexion; } public

function disconnect() { if ($this->conexion) { $this->conexion->close(); } } public function prepareAndExecute($sql, $params = []) { $stmt = $this->conexion->prepare($sql); if (!$stmt) { die("Error preparing statement: " . $this->conexion->error); } // Bind parameters if (!empty($params)) { $types = str_repeat('s', count($params)); // Assuming all parameters are strings for simplicity $stmt->bind_param($types, ...$params); } // Execute statement $stmt->execute(); return $stmt; } } // Usage $db = new Database(); $conn = $db->connect(); // $stmt = $conn->prepare("SELECT * FROM users WHERE username = ?"); // DON'T USE THE DIRECT CONNECTION LIKE THIS. // $stmt->bind_param("s", $username); // Use Prepared statements //To avoid SQL Injection, call $db->prepareAndExecute $sql = "SELECT * FROM users WHERE username = ?"; $params = ["your_username"]; // Retrieve username from validated POST data $result = $db->prepareAndExecute($sql, $params); $db->disconnect(); ?> ```

3. **Use Prepared Statements:** Always use prepared statements for all database queries. Avoid concatenating user input directly into SQL.
4. **Input Validation:** Implement thorough input validation to sanitize all user-provided data before using it in queries.

```
```