Vulnerability Report

Archivo: productos.php

Code Analyzed:

```
<?php
include "../../models/conexion.php"; // Asegúrate de que la ruta a este archivo es correcta

// Realizamos la consulta a la base de datos
$sql = $conexion->query("SELECT * FROM productos");

$productos = array();

// Recorremos el resultado de la consulta y lo agregamos al arreglo $productos
while ($producto = $sql->fetch_assoc()) {
    $productos[] = $producto;
}

// Devolvemos los datos en formato JSON
echo json_encode($productos);
?>
```

Analysis: ```html
Security Vulnerabilities

SQL Injection

Type: SQL Injection
Approximate Line: 5: `$sql = $conexion->query("SELECT * FROM productos");`
Description: While the current query might appear safe, if the `productos` table or related parts of the schema are ever constructed based on user input without proper sanitization, SQL injection becomes a risk. More specifically, if the table name "productos" is derived from an external source (e.g., user input, a configuration file susceptible to manipulation), SQL injection is possible.
Mitigation:

1. **Parameterized Queries (Prepared Statements):** The best approach is to use parameterized queries (prepared statements) with proper escaping. Even though the current code directly queries for all products, it is good practice to always use prepared statements when querying data.
2. **Input Validation and Sanitization:** Validate and sanitize all user inputs that influence the database query, or even indirectly influence schema construction or definition.
3. **Principle of Least Privilege:** Database user should only have the minimum necessary permissions required to perform their tasks. Do not use a database user with full administrative privileges to execute routine queries.

Path Traversal/Include Vulnerability

Type: Path Traversal/Include Vulnerability
Approximate Line: 1: `include "../../models/conexion.php";`
Description: If an attacker can control or influence the path passed to the `include` statement, they could potentially include arbitrary files, leading to code execution or information disclosure. This is especially problematic if the included file contains sensitive data or functions. For example, if an attacker could modify the URL or POST data to change the path to a file outside the intended directory, they could potentially include and execute malicious code.
Mitigation:

1. **Whitelist Approach:** Instead of allowing arbitrary paths, maintain a whitelist of allowed include paths. Verify the requested file against this whitelist before including it.

2. **Absolute Paths:** Use absolute paths for includes instead of relative paths. This reduces the risk of path traversal. Use a constant defined at the root of the application to prefix file paths.
3. **Restrict File Access:** Ensure that the web server's user account has only the necessary permissions to access the required files.

Code Quality Metrics

Complexity

Metric: Cyclomatic Complexity
Description: The code has low cyclomatic complexity. There are no branching statements (if, else, switch, for, while) besides the loop.

Duplication

Metric: Code Duplication
Description: There's no obvious code duplication within this small snippet. However, if the database interaction logic is repeated in other files, it should be refactored into a reusable function or class.

Readability

Metric: Readability
Description: The code is generally readable, with clear variable names and comments. However, consider adding more comments to explain the purpose of each section, especially if it's part of a larger application.

Coupling

Metric: Coupling
Description: The code is tightly coupled to the specific database table ('productos') and database connection (`$conexion`). Ideally, database interactions should be abstracted behind an interface or repository pattern to reduce coupling and improve testability. The tight coupling with the global `$conexion` makes unit testing more difficult.
Improvement: Introduce an abstraction layer (e.g., a data access object) to decouple the data access logic from the specific database implementation and the global connection object.

Error Handling

Metric: Error Handling
Description: The code lacks error handling. If the database query fails, or if `json_encode` fails, no error messages are logged or displayed. This can make debugging difficult.
Improvement: Add error handling for database queries using `mysqli_error($conexion)` and handle potential exceptions during the JSON encoding process. Log errors to a file or monitoring system.

Proposed Solution

The following changes address the security vulnerabilities and improve code quality:

1. **Implement Parameterized Queries:** While the current query is static, future queries should use prepared statements to prevent SQL injection.
2. **Validate Include Paths:** Ensure that the included files are whitelisted or use absolute paths.
3. **Improve Error Handling:** Add error checking for database queries and JSON encoding. Log errors to a file or monitoring system.
4. **Abstract Database Access:** Use a Data Access Object (DAO) or Repository pattern to decouple the data access logic from the specific database implementation. This includes abstracting the connection.

```
```