Vulnerability Report

Archivo: transaccion.php

Code Analyzed:

```
<!doctype html>
<html lang="en">

<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC" crossorigin="anonymous">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.5.2/css/all.min.css">
    <title>Hello, world!</title>
</head>

<body>

    <div class="container-fluid-row ">

        <div class="row px-4">
            <h1>Prueba Tecnica</h1>
            <div class="col">
                <div class="card">
                    <div class="card-body">
                        <form action="" method="POST">
                            <!-- Mostrar Error     -->
                            <?php
                            include "models/conexion.php";
                            include "controller/transaccion.php";

                            ?>
                            <div class="mb-3">
                                <label for="producto_id" class="form-label">ID del
Producto</label>
                                <input type="number" class="form-control" id="producto_id"
name="producto_id" required>
                            </div>
                            <div class="mb-3">
                                <label for="cantidad" class="form-label">Cantidad
Vendida</label>
                                <input type="number" class="form-control" id="cantidad"
name="cantidad" required>
                            </div>
                            <div class="mb-3">
                                <label for="precio" class="form-label">Precio</label>
                                <input type="number" class="form-control" id="precio"
name="precio" step="0.01" required>
                            </div>
                            <button type="submit" name="btntransaccion" class="btn btn-
success" value="ok">Registrar Venta</button>
                        </form>
                    </div>
                </div>
            </div>

        </div>
    </div>

    <!-- Optional JavaScript; choose one of the two! -->

    <!-- Option 1: Bootstrap Bundle with Popper -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJoMp4YLEuNSfAP+JcXn/tWtIaxVXM"
crossorigin="anonymous"></script>

    <!-- Option 2: Separate Popper and Bootstrap JS -->
    <!--
```

```
    <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min.js"
integrity="sha384-IQsoLXl5PILFhosVNubq5LC7Qb9DXgDA9i+tQ8Zj3iwWAwPtgFTxbJ8NT4GN1R8p"
crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.min.js"
integrity="sha384-cVKIPhGWiC2Al4u+LWgxfKTRIcfu0JTxR+EQDz/bgldoEyl4H0zUF0QKbrJ0EcQF"
crossorigin="anonymous"></script>
    -->
</body>

</html>
```

Analysis: ```html
Security Vulnerabilities

SQL Injection

Type: SQL Injection
Approximate Line: Inside `controller/transaccion.php` (exact line depends on implementation). Likely within the database interaction code.
Description: The code includes user-provided input (`producto_id`, `cantidad`, and `precio`) directly in SQL queries without proper sanitization or parameterization. This makes the application vulnerable to SQL injection attacks. An attacker could inject malicious SQL code via the input fields to read, modify, or delete data in the database.
Mitigation: Use prepared statements with parameterized queries (PDO or mysqli) to ensure that user input is treated as data, not as part of the SQL query. Never concatenate user input directly into SQL queries. Implement proper input validation and sanitization on the server-side (e.g., checking data types, ranges, and escaping special characters) as a defense-in-depth measure.
Improvement: SQL injection vulnerability is critical. Address it immediately by implementing parameterized queries.

Code Quality Issues

Lack of Input Validation

Metric: Readability, Reliability, Security
Description: While the HTML form includes the `required` attribute, this is client-side validation and can be easily bypassed. The PHP code needs to perform server-side validation to ensure data integrity and prevent errors or malicious input. For instance, checking that `producto_id` and `cantidad` are positive integers, and `precio` is a valid numeric value.
Approximate Line: `controller/transaccion.php`, where the values from `$_POST` are processed.
Mitigation: Implement server-side input validation. Use functions like `filter_var` with appropriate filters (e.g., `FILTER_VALIDATE_INT`, `FILTER_VALIDATE_FLOAT`) to validate data types. Check for minimum and maximum values, and sanitize inputs to remove potentially harmful characters.
Improvement: Add thorough server-side validation logic within `controller/transaccion.php`. This improves reliability and security. Good validation practices also make the code more readable because the expected types are clearly defined.

Tight Coupling

Metric: Coupling, Maintainability
Description: The `include "models/conexion.php";` and `include "controller/transaccion.php";` statements inside the HTML directly couple the presentation layer (HTML/PHP) with the data access and business logic. This makes the code harder to maintain, test, and reuse.
Approximate Line: PHP block within the HTML form.
Mitigation: Consider using a more structured approach like MVC (Model-View-Controller). The view (HTML) should only be responsible for presentation. Move the logic related to database connection and transaction processing to a separate controller or service layer.
Improvement: Adopting MVC or a similar pattern drastically reduces coupling, making the code easier to understand, test, and modify.

Proposed Solution

Addressing Vulnerabilities and Improving Code Quality

1. **SQL Injection Prevention:** Replace direct query construction with prepared statements and parameterized queries using PDO or mysqli. Ensure that the `controller/transaccion.php` file uses this approach.
2. **Server-Side Input Validation:** Implement robust server-side validation in `controller/transaccion.php` using appropriate PHP functions (e.g., `filter_var`) to validate and sanitize input data.
3. **Decoupling:** Refactor the code to separate concerns by implementing a Model-View-Controller (MVC) pattern or a similar architectural pattern. This would involve: * Moving database connection logic and data access to a Model layer. * Moving transaction processing logic to a Controller layer. * Keeping the HTML as a View, responsible only for presentation. The view will call controller methods, and the controller will interact with the model.
4. **Error Handling:** Implement proper error handling, especially around database interactions. Catch exceptions and display user-friendly error messages (avoid displaying sensitive information directly to the user).
5. **Data Sanitization:** Sanitize data to remove potentially harmful characters.

```
```