Vulnerability Report

Archivo: ecommerce.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 Con JavaScript</h1>
            <div class="col-4">
                <div class="card">
                    <div class="card-body">
                        <form id="formProducto">
                            <input type="hidden" id="id">
                            <div class="mb-3">
                                <label for="nombre" class="form-label">Nombre</label>
                                <input type="text" class="form-control" id="nombre"
name="nombre" required>
                            </div>
                            <div class="mb-3">
                                <label for="cantidad" class="form-label">Cantidad</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" class="btn btn-success">Guardar</button>
                        </form>
                    </div>
                </div>
            </div>
            <div class="col-8">
                <div class="card">
                    <div class="card-body">
                        <h2>Lista de Productos</h2>

                        <table id="productosTable" class="table table-success table-striped">
                            <thead>
                                <tr>
                                    <th>ID</th>
                                    <th>Nombre</th>
                                    <th>Cantidad</th>
                                    <th>Precio</th>
                                    <th>Acciones</th>
                                </tr>
                            </thead>
                            <tbody>
                                <!-- Los productos se cargarán aquí mediante JavaScript -->
                            </tbody>
                        </table>
                    </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>
    -->
    <script src="ecommerce.js"></script>
</body>

</html>
```

Analysis: ```html
Security Vulnerabilities & Code Quality Analysis

Vulnerabilities

Cross-Site Scripting (XSS)

Type: Stored/Persistent XSS

Approximate Line: Potentially throughout `ecommerce.js` where data from the form is displayed in the table.

Description: If the application stores the product data (name, etc.) without proper sanitization, and then displays that data on the page (e.g., in the `productosTable`), an attacker could inject malicious JavaScript code into the form fields. This code will then be executed in the browsers of other users who view the product list.

Mitigation:

1. **Input Validation:** Validate user inputs on both the client-side (JavaScript) and server-side (if applicable) to ensure they conform to expected formats and lengths.
2. **Output Encoding/Escaping:** Encode/escape user-provided data before rendering it in the HTML. This converts potentially dangerous characters into their safe HTML entities (e.g., < becomes `&lt;`).
3. **Content Security Policy (CSP):** Implement a CSP to restrict the sources from which the browser can load resources. This can help prevent the execution of injected scripts.
4. **HTML Sanitization:** Use a library specifically designed for HTML sanitization to remove potentially harmful tags and attributes from user-provided input.

Cross-Site Request Forgery (CSRF)

Type: CSRF

Approximate Line: The `formProducto` form submission.

Description: If the application uses cookies for authentication, an attacker could potentially forge requests to add/modify products if CSRF protection isn't in place. An attacker could create a malicious website that, when visited by an authenticated user, sends unauthorized requests to the application.

Mitigation:

1. **CSRF Tokens:** Implement CSRF tokens. Include a unique, unpredictable token in the form, which is then validated on the server-side.
2. **SameSite Cookie Attribute:** Set the `SameSite` attribute of the authentication cookies to `Strict` or `Lax` to prevent the browser from sending the cookie with cross-site requests.

Integer Overflow/Underflow

Type: Potential Integer Overflow/Underflow

Approximate Line: Input fields for 'cantidad' and 'precio'.

Description: Although less likely in JavaScript due to its number type, there's a theoretical risk that extremely large or small numbers entered into the 'cantidad' or 'precio' fields could lead to unexpected behavior if these values are later used in calculations without proper validation.

Mitigation:

1. **Input Validation:** Enforce reasonable maximum and minimum values for the 'cantidad' and 'precio' input fields.
2. **Data Type Considerations:** While JavaScript uses double-precision floating-point numbers, be mindful of potential precision issues when dealing with very large numbers, especially if calculations involve integers on the server-side (if applicable).

Code Quality Metrics

- **Readability:** The HTML structure is generally readable due to the use of Bootstrap classes, but the absence of comments can reduce maintainability.
- **Complexity:** The HTML itself is relatively simple. Complexity is likely concentrated in the `ecommerce.js` file, depending on its implementation of data handling, DOM manipulation, and validation.
- **Coupling:** The HTML is tightly coupled to Bootstrap due to the extensive use of Bootstrap classes. This makes it easier to style the page, but reduces flexibility if you wanted to switch to a different CSS framework. The coupling with \`ecommerce.js\` is also high, as the javascript fully manages the table of products.
- **Duplication:** There is no evident duplication in the HTML snippet provided. Duplication might be present in \`ecommerce.js\`.

Proposed Solution

To address the identified security vulnerabilities and improve code quality, the following steps are recommended:

1. **Implement Server-Side Logic:** The current code appears to rely heavily on client-side JavaScript. Moving data storage and processing to the server-side improves security and data integrity. Consider using a backend framework (e.g., Node.js with Express, Python with Flask/Django) and a database to store product data.
2. **Sanitize User Input:** Implement server-side input validation and sanitization to prevent XSS attacks. Libraries such as DOMPurify can be used to sanitize HTML input.
3. **Implement CSRF Protection:** Use CSRF tokens in the form to prevent CSRF attacks. Most backend frameworks have built-in support for CSRF protection.

4. **Validate Data Types and Ranges:** Validate that the data entered by the user is of the expected type (e.g., number) and within reasonable ranges. This helps to prevent integer overflows and other unexpected behavior.
5. **Consider Rate Limiting:** Implement rate limiting on the API endpoints to prevent abuse.
6. **Improve Code Structure:** Refactor the `ecommerce.js` file to improve its structure and readability. Use modular JavaScript to separate concerns and make the code easier to maintain. Consider using a JavaScript framework (e.g., React, Vue.js, Angular) to manage the application's state and UI.
7. **Add Comments and Documentation:** Add comments to the code to explain its functionality and make it easier for others (or yourself in the future) to understand. Generate API documentation.
8. **Decouple where reasonable** Refactor the javascript code to be decoupled from the HTML, this will improve the maintainability of the whole application.

By implementing these changes, the application will be more secure, robust, and maintainable.

```
```