Vulnerability Report

Archivo: factura.php

Code Analyzed:

```php
<?php

require_once("../models/Factura.php");

if(isset($_GET['op'])){
    $op = $_GET['op'];

    switch($op){
        case "guardar":
            // Verifica si se recibió el parámetro 'factura' vía POST
            if(isset($_POST['factura'])){
                // Decodifica el JSON recibido
                $facturaData = json_decode($_POST['factura'], true);

                // Opcional: valida la estructura de $facturaData según tus necesidades

                // Instancia el modelo y llama al método para guardar la factura
                $facturaModel = new Factura();

                try {
                    $resultado = $facturaModel->guardarFactura($facturaData);

                    if($resultado){
                        // Responde con un mensaje de éxito
                        echo json_encode([
                            "status" => "success",
                            "message" => "Factura guardada correctamente."
                        ]);
                    } else {
                        echo json_encode([
                            "status" => "error",
                            "message" => "No se pudo guardar la factura."
                        ]);
                    }
                } catch(PDOException $e) {
                    // En caso de error, envía el mensaje de error
                    echo json_encode([
                        "status" => "error",
                        "message" => "Error al guardar la factura: " . $e->getMessage()
                    ]);
                }
            } else {
                echo json_encode([
                    "status" => "error",
                    "message" => "No se recibieron datos de la factura."
                ]);
            }
            break;

        default:
            echo json_encode([
                "status" => "error",
                "message" => "Operación no reconocida."
            ]);
            break;
    }
} else {
    echo json_encode([
        "status" => "error",
        "message" => "No se especificó ninguna operación."
    ]);
}
?>
```

Analysis: ```html
Security Vulnerabilities

SQL Injection

Approximate Line: Within the `guardarFactura()` method (not visible in the given code, but assumed within the `Factura` model).

Description: If the `guardarFactura()` method in the `Factura` model directly uses data from the `$facturaData` array in raw SQL queries without proper sanitization or parameterization, it is vulnerable to SQL injection attacks. An attacker could inject malicious SQL code into the request, potentially allowing them to read, modify, or delete data in the database.

Mitigation:

1. **Use Prepared Statements:** Always use prepared statements or parameterized queries with placeholders when interacting with the database. This ensures that user-supplied data is treated as data, not as executable SQL code. PDO, used in the exception handling, supports prepared statements.
2. **Input Validation and Sanitization:** Validate and sanitize all input data before using it in SQL queries. Use whitelisting techniques to ensure that only expected characters and data types are allowed. PHP's `filter_var()` function can be used for input validation and sanitization.
3. **Principle of Least Privilege:** Ensure that the database user account used by the application has only the minimum necessary privileges to perform its tasks.

Cross-Site Scripting (XSS)

Approximate Line: Potential vulnerability when echoing any value that is read from user input.

Description: This application is vulnerable to XSS attacks if any data read from $_GET, $_POST or other client side data sources is echoed back to the end-user without sanitization. An attacker could inject malicious JavaScript code into the request, potentially allowing them to steal user sessions, redirect users to malicious websites, or deface the application.

Mitigation:

1. **Output Encoding:** Encode all output data before rendering it in HTML. Use PHP's `htmlspecialchars()` function to escape HTML entities.
2. **Content Security Policy (CSP):** Implement a Content Security Policy (CSP) to restrict the sources from which the browser can load resources.

Code Quality Metrics

Complexity

The code has a relatively low cyclomatic complexity. The switch statement has a limited number of cases, and the conditional logic is straightforward. However, the complexity could increase significantly within the `guardarFactura()` method in the `Factura` model if it contains complex business logic or database interactions.

Duplication

There is some code duplication in the repeated `echo json_encode(...)` blocks for different success/error scenarios. This could be reduced by creating a helper function to generate the JSON response.

Readability

The code is reasonably readable due to consistent indentation and clear variable names. However, adding comments explaining the purpose of each section and complex logic within the `guardarFactura()` method would improve readability further. Consider using docblocks for function and class descriptions.

Coupling

The code is tightly coupled to the `Factura` model. Any changes to the `Factura` model's interface (e.g., the `guardarFactura()` method signature) would require modifications to this code. Consider using dependency injection or interfaces to reduce coupling and improve testability.

Proposed Solution

1. **Implement Prepared Statements:** Modify the `guardarFactura()` method in the `Factura` model to use prepared statements with parameter binding when interacting with the database. This is the most important step to prevent SQL injection.
2. **Input Validation:** Before calling `guardarFactura()`, add validation to the `$facturaData` array. Ensure that required fields are present, and that the data types of the values are correct. Use a whitelisting approach to ensure that only expected fields are accepted.
3. **Output Encoding:** Ensure that all data written back to the end user in HTML is sanitized.
4. **JSON Response Helper:** Create a helper function to generate the JSON response. This will reduce code duplication and improve maintainability. For example:

```
function createJsonResponse(string $status, string $message, array $data = []): string {
  $response = ["status" => $status, "message" => $message];
  if (!empty($data)) {
    $response = array_merge($response, $data); // Avoid modifying the original $response
directly
  }
  return json_encode($response);
}

// Usage example:
echo createJsonResponse("success", "Factura guardada correctamente.");
```

5. **Error Logging:** Implement a robust error logging mechanism. Log detailed error messages, including stack traces, to a file or database. This will help with debugging and identifying potential security vulnerabilities. Avoid exposing sensitive information in the error messages that are displayed to the end user.
6. **Consider Dependency Injection:** If the application grows in complexity, consider using dependency injection to decouple the code from the `Factura` model. This will make the code more testable and maintainable.

```
```