Vulnerability Report

Archivo: registro.php

Code Analyzed:

```
<?php
if (!empty($_POST["btnregistrar"])) {
    if (
        !empty($_POST["nombre"]) &&
        !empty($_POST["apellido"]) &&
        !empty($_POST["dni"]) &&
        !empty($_POST["fecha"]) &&
        !empty($_POST["email"])
    ) {
        // Captura de los valores
        $nombre = $_POST["nombre"];
        $apellido = $_POST["apellido"];
        $dni = $_POST["dni"];
        $fecha = $_POST["fecha"];
        $email = $_POST["email"];

        // Utilizamos consultas preparadas para evitar inyecciones SQL
        $stmt = $conexion->prepare("INSERT INTO PERSONAS (nombre, apellido,
cedula, fecha_nacimiento, correo)
                                    VALUES (?, ?, ?, ?, ?)");

        // Vinculamos los parámetros
        $stmt->bind_param("sssss", $nombre, $apellido, $dni, $fecha, $email);

        // Ejecutamos la consulta
        if ($stmt->execute()) {
            echo '<div class="alert alert-success" role="alert">
                    Persona registrada correctamente.
                  </div>';
        } else {
            echo '<div class="alert alert-danger" role="alert">
                    Persona no registrada correctamente.
                  </div>';
        }

        // Cerramos el statement
        $stmt->close();
    } else {
        echo '<div class="alert alert-danger" role="alert">
                Alguno de los campos está vacío.
              </div>';
    }
}
?>
```

Analysis: ```html

Vulnerability Analysis

SQL Injection (Mitigated)

Description

Although the code uses prepared statements, it's crucial to ensure that the connection object `$conexion` is properly initialized and configured to use prepared statements by default. Some database extensions might not enable prepared statements by default or might allow emulation, which could open the door to SQL injection vulnerabilities if the underlying database library is bypassed.

Approximate Line

Line 14, 18

Mitigation

1. **Verify Prepared Statement Configuration:** Double-check that the database connection `$conexion` is properly configured to disable emulated prepared statements. This typically involves setting an attribute on the connection object. 2. **Parameter Binding Types:** Ensure the `bind_param` types accurately reflect the data types of the columns in the database. Inconsistent types may introduce unexpected behavior (although it's less of a direct vulnerability).

Improvement

N/A - this vulnerability is mostly avoided with the use of prepared statements.

Cross-Site Scripting (XSS)

Description

The code directly echoes user-provided data in alert messages (success or failure). If the `$nombre`, `$apellido`, `$dni`, `$fecha`, or `$email`

variables contain malicious JavaScript code, it will be executed in the user's browser, potentially leading to XSS vulnerabilities.

Approximate Line

Lines 23, 27

Mitigation

1. **Output Encoding:** Use proper output encoding or escaping when displaying user-provided data. For HTML output, use `htmlspecialchars()` to encode special characters. Example: `echo htmlspecialchars($nombre, ENT_QUOTES, 'UTF-8');`

Improvement

N/A

Code Quality Issues

Description

Lack of Input Validation: Although empty field checks are present, there is no validation on the *format* of the data being submitted. For example, the code doesn't verify if the 'dni' field contains only numeric characters, or if the 'email' field has a valid email format.
Error Handling: The error messages in the `else` blocks are generic. More specific error messages would help in debugging. The PHP error log should be monitored.
Readability and Maintainability: Although the code is reasonably readable, a more structured approach (e.g., using functions) could improve maintainability.

Approximate Line

Lines 3, 23, 27

Mitigation

1. **Input Validation:** Implement robust input validation to check the format and type of data received from the user. Use regular

expressions or built-in PHP functions (e.g., `filter_var()` for email validation) to validate each field. 2. **Specific Error Messages:** Provide more informative error messages to help identify the source of the problem. Log errors to the PHP error log. 3. **Code Structure:** Refactor the code into functions to improve readability and maintainability. For instance, create a function to validate the input, and another function to handle database insertion.

Improvement

* **Complexity:** Refactoring into functions reduces overall complexity. * **Readability:** Improved by using functions and meaningful variable names. * **Maintainability:** Enhanced by separating concerns into functions, making the code easier to modify and debug.

Proposed Solution

Below is an example showing input sanitization, and more complete input validation:

```
<?php
function sanitize_input($data) {
    $data = trim($data);
    $data = stripslashes($data);
    $data = htmlspecialchars($data, ENT_QUOTES, 'UTF-8');
//Sanitize for XSS
    return $data;
}

function validate_email($email) {
    return filter_var($email, FILTER_VALIDATE_EMAIL);
}

function validate_dni($dni) {
    return preg_match('/^[0-
9]{8}[TRWAGMYFPDXBNJZSQVHLCKE]$/i', $dni); //Example Spanish
DNI validation. Adapt for your needs.
}

if (!empty($_POST["btnregistrar"])) {
    if (
        !empty($_POST["nombre"]) &&
        !empty($_POST["apellido"]) &&
```

```
        !empty($_POST["dni"]) &&
        !empty($_POST["fecha"]) &&
        !empty($_POST["email"])
    ) {
        // Sanitize and capture values
        $nombre = sanitize_input($_POST["nombre"]);
        $apellido = sanitize_input($_POST["apellido"]);
        $dni = sanitize_input($_POST["dni"]);
        $fecha = sanitize_input($_POST["fecha"]);
        $email = sanitize_input($_POST["email"]);

        // Validate input
        if (!validate_email($email)) {
            echo '<div class="alert alert-danger"
role="alert">
                    Email inválido.
                  </div>';
            exit; //Stop execution
        }

        if (!validate_dni($dni)) {
           echo '<div class="alert alert-danger"
role="alert">
                    DNI inválido.
                  </div>';
            exit; //Stop execution
        }


        // Prepared statement
        $stmt = $conexion->prepare("INSERT INTO PERSONAS
(nombre, apellido, cedula, fecha_nacimiento, correo)
                                    VALUES (?, ?, ?, ?,
?)");
        $stmt->bind_param("sssss", $nombre, $apellido, $dni,
$fecha, $email);

        if ($stmt->execute()) {
            echo '<div class="alert alert-success"
role="alert">
                    Persona registrada correctamente.
                  </div>';
        } else {
            error_log("Database error: " . $stmt->error); //
Log the error
            echo '<div class="alert alert-danger"
role="alert">
                    Error al registrar la persona.  Consulte
los registros del servidor.
                  </div>';
        }

        $stmt->close();
```

```
    } else {
        echo '<div class="alert alert-danger" role="alert">
                Alguno de los campos está vacío.
              </div>';
    }
}
?>
```

Key changes:

- Introduced `sanitize_input` function to escape data for XSS prevention.
- Added input validation functions like `validate_email` and `validate_dni`.
- Used `error_log` to log database errors for debugging.
- Stopped script execution if validation fails (using `exit`).
- Proper html encoding.

```
```