Vulnerability Report

Archivo: modificar.php

Code Analyzed:

```php
<?php
if (!empty($_POST["btnactualizar"])) {
    if (
        !empty($_POST["id"]) &&
        !empty($_POST["nombre"]) &&
        !empty($_POST["apellido"]) &&
        !empty($_POST["dni"]) &&

        !empty($_POST["email"])
    ) {
        // Captura de los valores
        $nombre = $_POST["nombre"];
        $apellido = $_POST["apellido"];
        $dni = $_POST["dni"];

        $email = $_POST["email"];

        // Utilizamos consultas preparadas para evitar inyecciones SQL
        $stmt = $conexion->prepare("UPDATE PERSONAS
        SET
        nombre = ?,
        apellido = ?,
        cedula = ?,
        fecha_nacimiento = NOW(),
        correo = ?
        WHERE id = ?");

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

        // Ejecutamos la consulta
        if ($stmt->execute()) {
            // Almacenamos el mensaje en la sesión
            session_start();
            $_SESSION['mensaje'] = 'Persona Actualizada correctamente.';

            // Redirigimos al index.php
            header("Location: index.php");
            exit;  // Es importante llamar a exit después de la redirección
        } else {
            echo '<div class="alert alert-danger" role="alert">
            Persona no Actualizada 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
Security Vulnerabilities

SQL Injection

Type: Potential SQL Injection.
Line: The code uses a prepared statement, but the $id variable is not sanitized or validated and is not being bound to the prepared statement. It is possible that the `$id` variable is being set outside of the provided code snippet, and that could lead to a SQL Injection.

Mitigation: Ensure the `$id` variable is properly sanitized and validated before being used in the WHERE clause. The simplest and most secure mitigation is to add `$id` to the prepared statement parameter list and bind it properly. Since it is an integer identifier, ensure the data type is handled properly.

Code Quality Metrics

Error Handling and User Feedback

Description: The error messages are basic and provide minimal information. The success message relies on session variables.

Improvement: Provide more informative error messages to help debug issues. Avoid relying solely on session variables for displaying success messages. Display the success/error message directly after update, without redirecting first.

Lack of Input Validation

Description: While the code checks for empty fields, there is no validation to ensure the data entered is in the correct format (e.g., email address, DNI).

Improvement: Implement robust input validation to ensure that the data entered by the user matches the expected format and constraints. Use regular expressions for validating email and DNI fields.

Data Type Considerations

Description: The `bind_param` function uses `sssss` for all parameters. This treats all input as strings. While it works for many inputs, ensuring the correct types are specified improves type safety and reduces the risk of unexpected behavior when interacting with the database.

Improvement: Investigate data types in your database schema and use the correct types in bind_param. E.g., using "sssis" if `id` is expected to be an integer. If `cedula` is an integer, that too should be converted to `i`.

Proposed Solution

To address the SQL injection and improve code quality, consider the following revised code snippet:

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

        //Validate email
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
          echo '<div class="alert alert-danger" role="alert">Invalid email format</div>';
          exit;
        }

        // Validate dni (basic check, adjust as needed)
        if (!preg_match('/^[0-9]+$/', $dni)) {
          echo '<div class="alert alert-danger" role="alert">Invalid DNI format</div>';
          exit;
        }
        // Utilizamos consultas preparadas para evitar inyecciones SQL
        $stmt = $conexion->prepare("UPDATE PERSONAS
        SET
        nombre = ?,
        apellido = ?,
        cedula = ?,
        fecha_nacimiento = NOW(),
        correo = ?
        WHERE id = ?");

        // Vinculamos los parámetros.  Assume id is an integer. Change 'i' accordingly if
needed.
        $stmt->bind_param("ssssi", $nombre, $apellido, $dni, $email, $id);


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

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

Key changes:

- The `$id` is now bound to the prepared statement using the 'i' parameter type (assuming it's an integer).
- Added rudimentary email and DNI validation. The DNI regex may need to be adapted to fit the specific format required.
- Error messages include more detail for debugging, specifically the MySQL error message.
- Removed the reliance on session variables for displaying the success message and the redirect. This simplifies the control flow.

```
```