Vulnerability Report

Archivo: usuario.php

Code Analyzed:

```
<?php
    require_once("ClassUsuario.php");
    $objUsuario = new Usuario("Andres Cardona","andres@info.com","Admin");
    $objUsuario1 = new Usuario("Andresa","andresa@info.com","Cliente");
    echo Usuario::$strEstado;


    echo $objUsuario->gerPerfil();
    echo $objUsuario1->gerPerfil();
    echo $objUsuario1->setCambiarClave("123456");

    echo $objUsuario1->gerPerfil();

?>
```

Analysis: ```html
Security Vulnerabilities

Lack of Input Validation & Security Context

Vulnerability Type: Information Disclosure & Missing Security Controls.
Approximate Line: All lines where the `Usuario` class is instantiated and used.
Description: The provided code directly instantiates and manipulates user objects without any form of input sanitization, validation, or authentication. This could lead to several security issues if the underlying `ClassUsuario.php` has vulnerabilities. There's also a lack of context. We don't know where this code snippet fits within a larger application (e.g., is it part of an admin panel accessible without proper authorization?).

Mitigation:

1. **Implement Authentication and Authorization:** Restrict access to sensitive user information and functions based on user roles and permissions. The `Admin` role should be handled carefully.
2. **Input Validation:** Before creating `Usuario` objects, thoroughly validate all input parameters (name, email, role). Use techniques such as:
    - **Data Type Validation:** Ensure that the input is of the expected data type (e.g., string, integer).
    - **Length Validation:** Limit the length of input strings to prevent buffer overflows.
    - **Regular Expression Validation:** Use regular expressions to enforce specific patterns for email addresses and other fields.
    - **Whitelist Validation:** If possible, validate the "role" against a predefined whitelist of allowed roles.
3. **Output Encoding:** Before displaying user data (e.g., through `gerPerfil`), properly encode the output to prevent cross-site scripting (XSS) vulnerabilities. Use `htmlspecialchars()` for HTML output.
4. **Password Handling:** The `setCambiarClave` method likely handles passwords. It's crucial to:
    - **Hash Passwords:** Never store passwords in plain text. Use a strong hashing algorithm like bcrypt or Argon2.
    - **Salt Passwords:** Add a unique salt to each password before hashing to prevent rainbow table attacks.
    - **Password Complexity:** Enforce password complexity requirements (minimum length, special characters, etc.).
5. **Principle of Least Privilege:** Ensure that the code runs with the minimum necessary privileges. Avoid running code as root or with excessive permissions.

Code Quality Metrics

Code Quality Issues

Complexity: Low in this specific snippet, but the complexity of the `ClassUsuario.php` is unknown and potentially high if it contains complex logic for user management, authentication, and authorization.
Duplication: Minimal in this snippet, but could be present within the `ClassUsuario.php` file if similar code is repeated across methods.
Readability: Moderate. The code is relatively easy to understand, but using more descriptive variable names (instead of `objUsuario`, `objUsuario1`) would improve readability. Adding comments to explain the purpose of each section would also help.
Coupling: The code is highly coupled to the `ClassUsuario.php` file. Any changes to the `Usuario` class will directly affect this script. Consider using interfaces or abstract classes to reduce coupling.

Improvements:

1. **Reduce Coupling:** Use interfaces or abstract classes to define a contract for user management. This would allow you to switch to a different user management implementation without modifying the main script.
2. **Improve Readability:** Use more descriptive variable names and add comments to explain the purpose of each section of code.
3. **Address Class Complexity:** If the `ClassUsuario.php` file is complex, break it down into smaller, more manageable classes or methods. Use design patterns (e.g., Strategy, Factory) to simplify the code.
4. **Abstract Configuration:** Store roles and other configuration parameters in a configuration file or database, rather than hardcoding them in the PHP code.

Proposed Solution

Enhanced Code Example (Illustrative)

This code snippet demonstrates improved input validation, password hashing and salting, and output encoding. Note: This is a simplified example and might require further modifications depending on the actual implementation of `ClassUsuario.php`.

```
<?php
require_once("ClassUsuario.php");

// Input Validation and Sanitization
$name = filter_var($_POST['name'], FILTER_SANITIZE_STRING);
$email = filter_var($_POST['email'], FILTER_SANITIZE_EMAIL);
$role = $_POST['role']; // Validate against a whitelist!
$allowedRoles = ['Admin', 'Cliente']; // Example
if (!in_array($role, $allowedRoles)) {
    die("Invalid role"); //Or throw an exception, or display an error
}
if (empty($name) || empty($email)) {
    die("Name and email are required.");
}

// Secure Password Handling (Illustrative) - Assuming a password is being set initially.
$password = $_POST['password'];
$hashedPassword = password_hash($password, PASSWORD_ARGON2I); // Use Argon2i or Bcrypt

// Create the User Object (Only if Validation Passed!)
$objUsuario = new Usuario($name, $email, $role, $hashedPassword);  // Assuming the class
constructor allows for a password to be passed

// Example of updating password
if(isset($_POST['new_password'])) {
    $newPassword = $_POST['new_password'];
    $newHashedPassword = password_hash($newPassword, PASSWORD_ARGON2I);

    // Assuming a method exists to update the password in the Usuario class.
    $objUsuario->setCambiarClave($newHashedPassword);
```

```
}

// Output Encoding (Important for Preventing XSS)
echo htmlspecialchars(Usuario::$strEstado, ENT_QUOTES, 'UTF-8');

// Outputting a profile (Example - Encode attributes before echoing them)
echo htmlspecialchars($objUsuario->gerPerfil(), ENT_QUOTES, 'UTF-8'); //assuming gerPerfil
returns text to be printed directly

//If the class does not allow for password setting on creation, consider a new function to set
password and hash it
```

Explanation:

- Uses `filter_var` for sanitizing input, although specific validation like length or regex is not demonstrated, those can be implemented as well.
- Password handling now includes proper hashing with `password_hash` and salting (handled implicitly by `password_hash`).
- Output encoding is done using `htmlspecialchars` to prevent XSS attacks.

Note: This is a simplified example and might need to be adapted to the specifics of your `ClassUsuario.php` and the broader application context. It also assumes the existence of POST data, demonstrating how input data should be validated before interacting with the objects. Exception handling or custom error handling should be implemented for a production environment.

```
```