Vulnerability Report

Archivo: Cliente.php

Code Analyzed:

```
<?php
class Cliente extends Conectar
{
   /*  public function listarCliente()
    {
        $conectar = parent::Conexion();
        $sql = "SP_L_Cliente_01";
        $query = $conectar->prepare($sql);

        $query->execute();
        return $query->fetchAll(PDO::FETCH_ASSOC);
    } */

    public function listarCliente()
    {
        $conectar = parent::Conexion();
        $sql = "CALL SP_L_Cliente_01()"; // Corrección: Usar CALL
        $query = $conectar->prepare($sql);
        $query->execute();
        return $query->fetchAll(PDO::FETCH_ASSOC);
    }


    public function Cliente_id($id)
    {
        $conectar = parent::Conexion();
        $sql = "CALL SP_L_Cliente_02(?)";
        $query = $conectar->prepare($sql);
        $query->bindValue(1, $id);
        $query->execute();
        return $query->fetchAll(PDO::FETCH_ASSOC);
    }


}
```

Analysis: ```html
Security Vulnerabilities

SQL Injection

Type: Potential SQL Injection.
Line: `Cliente_id` method. Although using prepared statements, the stored procedure `SP_L_Cliente_02(?)` might be vulnerable to SQL injection if it doesn't properly sanitize or validate the input `$id` on the server side (inside the stored procedure definition). Relying solely on prepared statements in PHP might not fully protect against stored procedure vulnerabilities.
Mitigation:

- **Review Stored Procedure Definition:** The most crucial step is to thoroughly review the code of the stored procedure `SP_L_Cliente_02` to ensure it sanitizes or validates the input parameter `$id` appropriately to prevent any form of SQL injection.
- **Input Validation:** Even though the code uses bindValue, add client-side input validation on the `$id` variable (before passing it to the method). Check for expected data type (e.g., integer) and range.
- **Principle of Least Privilege:** Ensure the database user account used by PHP has only the minimum required privileges. Avoid using 'root' or similar high-privilege accounts.

Code Quality Metrics

Complexity, Readability, and Coupling

Complexity: The code is relatively simple and not overly complex. The complexity is primarily determined by the complexity of the stored procedures, which are not directly visible in this code.
Duplication: There is some minor duplication in the way database connections and queries are prepared in each method.
Readability: The code is fairly readable, with clear method names and comments (though minimal). Consistent naming conventions are followed.
Coupling: The class `Cliente` is tightly coupled to the `Conectar` class (through inheritance) and the database schema (specifically, the stored procedures). Changes to either `Conectar` or the stored procedures could significantly impact the `Cliente` class.
Improvements:

- **Reduce Duplication:** Consider creating a private helper method within the `Cliente` class to handle the database connection and query preparation logic. This would reduce redundancy.
- **Increase Abstraction:** Abstract the database interaction further by using a Data Access Object (DAO) pattern. This would decouple the `Cliente` class from the specific database implementation (e.g., MySQL via PDO) and the specific stored procedures.
- **Add Comments:** Add more comprehensive comments to explain the purpose of each method and the expected input and output.

Proposed Solution

Addressing Vulnerabilities and Improving Code Quality

The primary focus should be on securing the stored procedures. Review the code within `SP_L_Cliente_01` and `SP_L_Cliente_02` for SQL injection vulnerabilities. Apply proper input validation and sanitization.
To improve code quality:

1. **DAO Pattern:** Introduce a Data Access Object (DAO) for database operations. This will encapsulate the database interaction logic and reduce coupling.
2. **Helper Methods:** Create private helper methods within the `Cliente` class to handle common tasks like database connection and query preparation.
3. **Dependency Injection:** Instead of hardcoding the dependency on the `Conectar` class through inheritance, consider using dependency injection. This allows for easier testing and greater flexibility.
4. **Input Validation:** Implement input validation at multiple layers (client-side, server-side in PHP, and within the stored procedure) to protect against malicious input.

```
```