Vulnerability Report

Archivo: Empresa.php

Code Analyzed:

```
<?php
class Empresas extends Conectar
{
   /*  public function listarEmpresas()
    {
        $conectar = parent::Conexion();
        $sql = "SP_L_Empresa_01";
        $query = $conectar->prepare($sql);

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

    public function listarEmpresas()
    {
        $conectar = parent::Conexion();
        $sql = "CALL SP_L_EMPRESA_01()"; // Corrección: Usar CALL
        $query = $conectar->prepare($sql);
        $query->execute();
        return $query->fetchAll(PDO::FETCH_ASSOC);
    }


    public function Empresa_id($id)
    {
        $conectar = parent::Conexion();
        $sql = "CALL SP_L_EMPRESA_02(?)";
        $query = $conectar->prepare($sql);
        $query->bindValue(1, $id);
        $query->execute();
        return $query->fetchAll(PDO::FETCH_ASSOC);
    }


}
```

Analysis: ```html

Security Vulnerabilities

SQL Injection (Potential) - Empresa_id function

Type: SQL Injection (Stored Procedure potentially vulnerable). While the code uses prepared statements with parameter binding, the security ultimately depends on the implementation of the stored procedure `SP_L_EMPRESA_02`.
Approximate Line: Line containing `CALL SP_L_EMPRESA_02(?)` in the `Empresa_id` function.
Description: If the stored procedure `SP_L_EMPRESA_02` does *not* properly sanitize or parameterize the input it receives from the `$id` variable, it is still vulnerable to SQL Injection. The prepared statement only protects the PHP code; it does not magically secure the SQL within the stored procedure.
Mitigation:

1. **Review Stored Procedure:** The most critical step is to carefully review the implementation of the `SP_L_EMPRESA_02` stored procedure. Ensure that all input used within the stored procedure is properly parameterized or sanitized to prevent SQL injection. Avoid dynamic SQL construction within the stored procedure.
2. **Input Validation:** Although parameter binding helps, implement server-side input validation on the `$id` variable *before* passing it to the stored procedure. Verify that it is of the expected type (e.g., integer), and that it falls within an acceptable range. Use a type hint if possible and cast to an integer.
3. **Principle of Least Privilege:** Ensure the database user used by the PHP application has only the minimum necessary privileges. It should not have excessive permissions that could be exploited if an injection vulnerability exists.

Code Quality Metrics & Improvements

Code Quality Analysis

Complexity: The code is relatively simple, with low cyclomatic complexity in each function.
Duplication: There is minor duplication in the connection setup and query preparation steps in each function (`listarEmpresas` and `Empresa_id`). This can be improved by extracting the common parts into a helper function or using dependency injection of the database connection.
Readability: The code is generally readable. Consistent naming conventions and clear comments improve readability.
Coupling: The `Empresas` class is tightly coupled to the `Conectar` class through inheritance. This makes it difficult to test and reuse the `Empresas` class in isolation.
Improvements:

1. **Reduce Duplication:** Create a private method in the `Empresas` class to handle the database connection and query preparation, reducing repetition. This function would accept the SQL query as a parameter.
2. **Reduce Coupling:** Use Dependency Injection (DI) to provide the database connection to the `Empresas` class. Instead of inheriting from `Conectar`, pass an instance of a database connection object (or an interface representing a database connection) to the `Empresas` constructor. This makes the `Empresas` class more flexible and testable.
3. **Consider an ORM:** For larger projects, consider using an ORM (Object-Relational Mapper) such as Doctrine or Eloquent (if using Laravel). ORMs can significantly improve code quality, maintainability, and security by abstracting away the database interaction details.
4. **Error Handling:** Implement more robust error handling. Catch exceptions that might be thrown by the database operations and log them appropriately.
5. **Naming conventions:** Enforce naming conventions for all class, variable, and function names. The camelCase convention is mostly used, but `Empresa_id` violates it.

Proposed Solution

Refactored Code Example (Illustrative)

This example demonstrates reducing duplication and using Dependency Injection (DI). It's important to review the stored procedure for SQL injection vulnerabilities, even with these changes.

```php
<?php
// Define a database connection interface
interface DatabaseConnectionInterface {
    public function prepare(string $sql): PDOStatement;
    public function lastInsertId(): string|false;
}

class Conectar implements DatabaseConnectionInterface {
    private $connection;
    public function __construct() {
        $this->connection = new
PDO("mysql:host=localhost;dbname=your_database", "username", "password");
        $this->connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
    }
    public function prepare(string $sql): PDOStatement {
        return $this->connection->prepare($sql);
    }
    public function lastInsertId(): string|false {
        return $this->connection->lastInsertId();
    }
}

class Empresas
{
    private $db;

    public function __construct(DatabaseConnectionInterface $db)
    {
        $this->db = $db;
    }


    private function executeQuery(string $sql, array $params = []): array
    {
        try {
            $query = $this->db->prepare($sql);
            foreach ($params as $key => $value) {
                $query->bindValue($key + 1, $value); // PDO uses 1-based
indexing
            }
            $query->execute();
            return $query->fetchAll(PDO::FETCH_ASSOC);
        } catch (PDOException $e) {
            // Log the error, handle exception more gracefully
            error_log("Database error: " . $e->getMessage());
            return []; // Or throw a custom exception
        }
    }

    public function listarEmpresas(): array
    {
        $sql = "CALL SP_L_EMPRESA_01()";
        return $this->executeQuery($sql);
    }


    public function getEmpresaById(int $id): array
    {
        $sql = "CALL SP_L_EMPRESA_02(?)";
        return $this->executeQuery($sql, [$id]); // Pass ID as parameter
    }
}

// Example usage (Dependency Injection)
$db = new Conectar();
$empresas = new Empresas($db);
$empresas_list = $empresas->listarEmpresas();
```

```
$empresa_details = $empresas->getEmpresaById(123);
```

Key Changes:

- Dependency Injection: The `Empresas` class now receives a `DatabaseConnectionInterface` in its constructor. This makes it more testable and less coupled to a specific database implementation.
- Helper Function: The `executeQuery` method handles the common database interaction logic (preparing the statement, binding parameters, executing the query, and fetching the results). Error handling is improved.
- Type Hinting: Type hinting is used for the `$id` parameter in `getEmpresaById`.

```
```