ORM Injection

– BY SAMIR KHAN

---

introduction

Code
ORM
Database
00101011

Object-Relational Mapping (ORM) is a programming paradigm that bridges the gap between object-oriented programming languages and relational databases. It abstracts database interactions by converting data between incompatible systems, allowing developers to manipulate database records as objects within their code. While ORM frameworks like SQLAlchemy, Hibernate, and Django ORM offer convenience and security features, they are not immune to injection attacks.

Overview of ORM Injection

ORM Injection occurs when an attacker manipulates inputs in such a way that they bypass the ORM's security mechanisms and execute arbitrary queries on the database. Unlike traditional SQL injection, ORM injection targets the abstraction layer provided by the ORM framework, making it a more advanced and nuanced form of attack.

Learning Objectives Breakdown

1. Understanding ORM

- **What is ORM?**
  - An abstraction layer that maps database tables to programming language objects.
  - Examples: SQLAlchemy (Python), Hibernate (Java), Entity Framework (.NET).
- **Advantages of ORM:**
  - Simplifies database queries.
  - Reduces boilerplate code.
  - Built-in mechanisms to prevent common vulnerabilities like SQL injection.
- **Potential Risks:**
  - Misuse or improper implementation of ORM frameworks can reintroduce vulnerabilities.
  - Custom query methods or dynamic queries can bypass ORM protections.

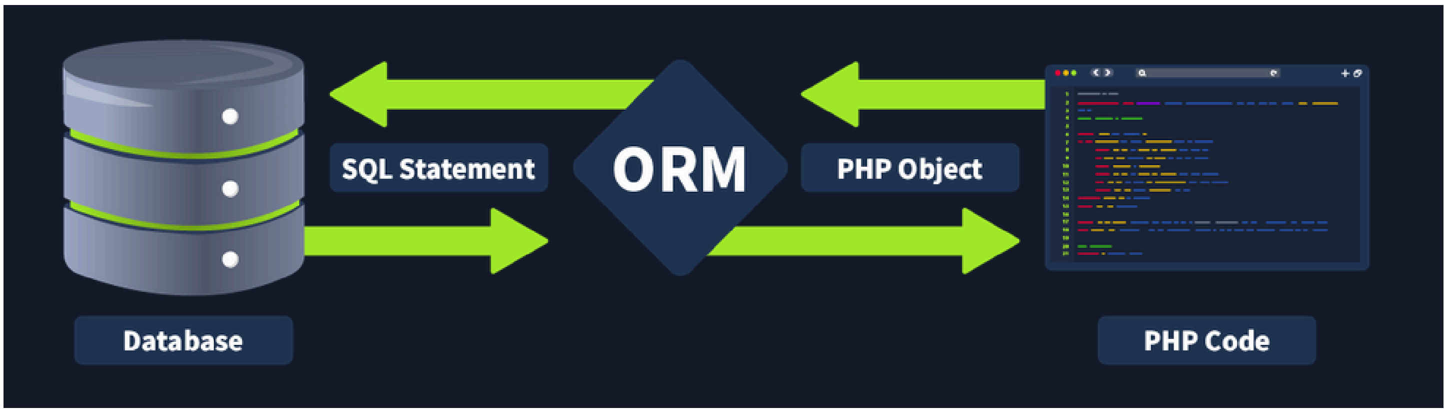
2. Identifying Injection

- **Key Indicators of ORM Injection Vulnerabilities:**
  - Use of **dynamic queries**: Concatenating user input directly into query strings.
  - Insufficient validation or sanitization of input data.
  - Overreliance on ORM's built-in protections without understanding its limitations.

The SQLAlchemy example shows a vulnerable query to SQL injection:

```
query = f"SELECT * FROM users WHERE username = '{user_input}'"
session.execute(query)  # Vulnerable to injection
```

Understanding ORM



What is ORM?

Object-Relational Mapping (ORM) is a programming technique that simplifies the way developers interact with databases. Instead of writing SQL queries to manage database data, ORM allows developers to work with data as objects in their programming language. This makes database operations more intuitive and reduces the need for extensive manual SQL queries.

How ORM and Databases Work Together

- **Bridge Between Two Worlds:** ORM connects the object-oriented programming world (classes and objects) with the relational database world (tables and rows).
- **Automatic Translation:**ORM frameworks automatically convert data between these systems:
  - Objects → Database rows (saving data).
  - Database rows → Objects (retrieving data).

Purpose of ORM

The main goal of ORM is to make database management easier for developers by:

1. **Reducing Boilerplate Code:** ORM generates SQL queries for common operations, saving developers from writing repetitive code.
2. **Increasing Productivity:** Developers can focus on application logic rather than database-specific code.
3. **Ensuring Consistency:** ORM frameworks handle database interactions in a consistent way, reducing errors.
4. **Enhancing Maintainability:** Updating the database schema is easier, as ORM frameworks link it to the object model.

Popular ORM Frameworks

1. **Doctrine (PHP)**
   - A powerful ORM for PHP.

- Used heavily in Symfony but can work independently.
- Provides tools like a query builder and schema management.
- Great for managing complex object structures.

2. **Hibernate (Java)**
   - A mature ORM for Java applications.
   - Maps Java classes to database tables.
   - Uses Hibernate Query Language (HQL) for powerful database interactions.
   - Features like caching and lazy loading improve performance.
3. **SQLAlchemy (Python)**
   - A flexible ORM for Python.
   - Offers both an SQL toolkit and an ORM system.
   - Allows raw SQL use when needed but supports object-based data handling.
   - Suitable for small projects and large enterprise systems alike.
4. **Entity Framework (C#)**
   - Microsoft's ORM framework for .NET.
   - Simplifies database interaction by allowing developers to use C# objects.
   - Supports multiple database providers and works seamlessly with other .NET technologies.
5. **Active Record (Ruby on Rails)**
   - The default ORM for Ruby on Rails.
   - Maps database tables to classes and rows to instances.
   - Offers built-in methods for querying and manipulating data.

Why Use ORM?

1. **Ease of Use:**
   - Write less code for database operations.
   - Use programming language syntax instead of SQL.
2. **Improved Productivity:**
   - Focus on developing features instead of database management.
3. **Flexibility:**
   - Switch database systems with minimal changes (e.g., moving from MySQL to PostgreSQL).
4. **Security:**
   - Many ORM frameworks include features like parameterized queries to prevent SQL injection.

By using ORM, developers can manage data effectively, save time, and reduce the likelihood of bugs, making it a crucial tool in modern application development.

HOW ORM WORKS

How ORM Connects Code and Database

1. **Mapping**: Classes in the code represent tables in the database. Each class corresponds to a table, properties correspond to columns, and instances correspond to rows.
2. **CRUD Operations**: ORM frameworks handle:
   - **Create**: Insert new data into the database.
   - **Read**: Retrieve data from the database.
   - **Update**: Modify existing records.
   - **Delete**: Remove records.
3. For example, you can retrieve all users or find a specific user by interacting with objects rather than writing SQL queries.

Purpose of ORM

The main goal of ORM is to abstract database interactions and make them easier to handle:

- **Less Repetition**: Automatically generate SQL queries for standard operations.
- **Higher Productivity**: Focus on application logic instead of complex database queries.
- **Consistency**: Unified handling of database operations reduces errors.
- **Maintainability**: Database schema changes are easier to implement.

Popular ORM Frameworks

1. **Laravel Eloquent** (PHP): Maps classes to database tables, providing a simple syntax for database operations.
2. **Hibernate** (Java): Supports advanced features like caching and lazy loading for performance.
3. **SQLAlchemy** (Python): Combines raw SQL flexibility with the simplicity of an ORM.
4. **Entity Framework** (C#): Seamlessly integrates with .NET applications.
5. **Active Record** (Ruby on Rails): Follows the principle of mapping one class to one table.

How Migrations Help in ORM

Migrations act as a version control system for the database. They let you:

- Define the structure of your database (tables and relationships) in code.
- Automatically apply changes to the database schema without writing raw SQL.

Steps in Laravel:

1. Create a migration file using a command.
2. Define the database structure (columns, types) in the up() method.
3. Apply the migration to the database using the php artisan migrate command.

Security Considerations: SQL Injection vs. ORM Injection

1. **SQL Injection**:
   - Exploits vulnerabilities in raw SQL queries.
   - Example: Input like ' OR '1'='1 can alter query behavior.

2. **ORM Injection**:
   - Exploits ORM query methods by manipulating inputs to generate malicious queries.
   - Example: Supplying admin' OR '1'='1 to an ORM query method.

Mitigation:

- Use parameterized queries and validate inputs.
- Ensure proper ORM configurations to prevent unintended query manipulations.

Testing ORM Injection

Identifying ORM Injection Vulnerabilities

ORM injection vulnerabilities occur when user inputs are embedded into ORM query methods without proper validation or sanitization. These issues often arise from improper coding practices, such as using dynamic queries or directly incorporating untrusted input. Key indicators of potential vulnerabilities include:

- **Dynamic Queries**: Concatenating user inputs directly into queries without safeguards.
- **Raw Query Execution**: Using methods like whereRaw() in Laravel or similar raw query builders.
- **Lack of Parameterized Queries**: Failing to use ORM features that safely handle input parameters, leaving the system open to injection attacks.

Techniques for Testing ORM Injection

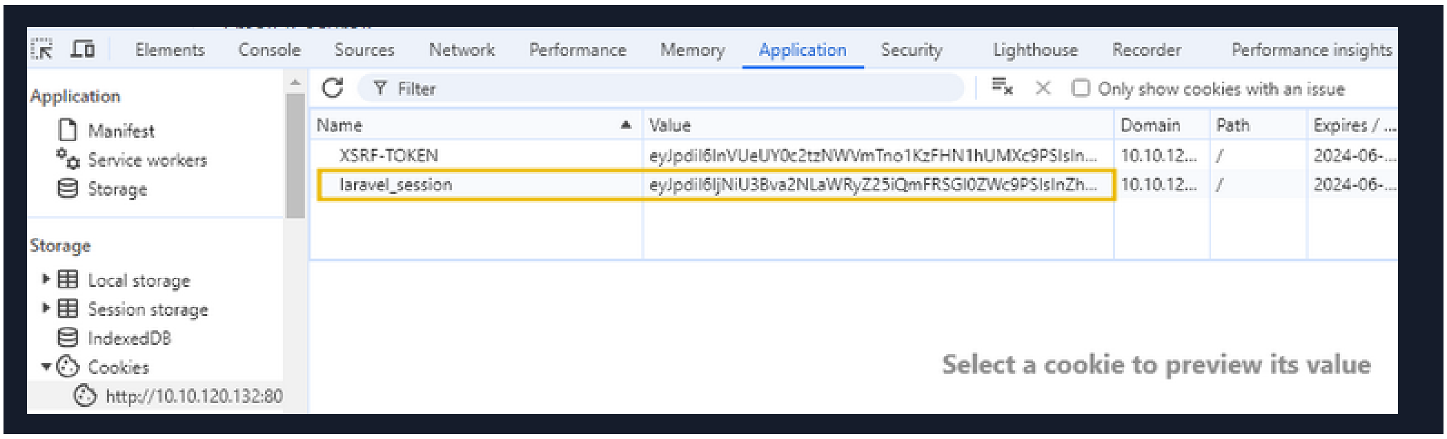
1. **Manual Code Review**

- Inspect the source code for methods like whereRaw() in Laravel or similar ORM methods that execute raw queries.
- Look for concatenated strings or unescaped inputs directly used in ORM query methods, as these are common injection points.
- Example of risky code:

–
```
USER::WHERERAW("EMAIL = '{$EMAIL}'")->GET();
```

Frameworks and ORM Injection Testing

Framework	ORM Library	Common Vulnerable Methods
Laravel	Eloquent ORM	whereRaw(), DB::raw()
Ruby on Rails	Active Record	where("name = '#{input}'")
Django	Django ORM	extra(), raw()
Spring	Hibernate	createQuery() with concatenation
Node.js	Sequelize	sequelize.query()



Name	Value	Domain	Path	Expires / ...
XSRF-TOKEN	eyJpdiI6InVUeUY0c2tzNWVmTno1KzFHN1hUMXc9PSIsIn...	10.10.12...	/	2024-06-...
laravel_session	eyJpdiI6IjNiU3Bva2NLaWRyZ25iQmFRSGI0ZWc9PSIsInZh...	10.10.12...	/	2024-06-...

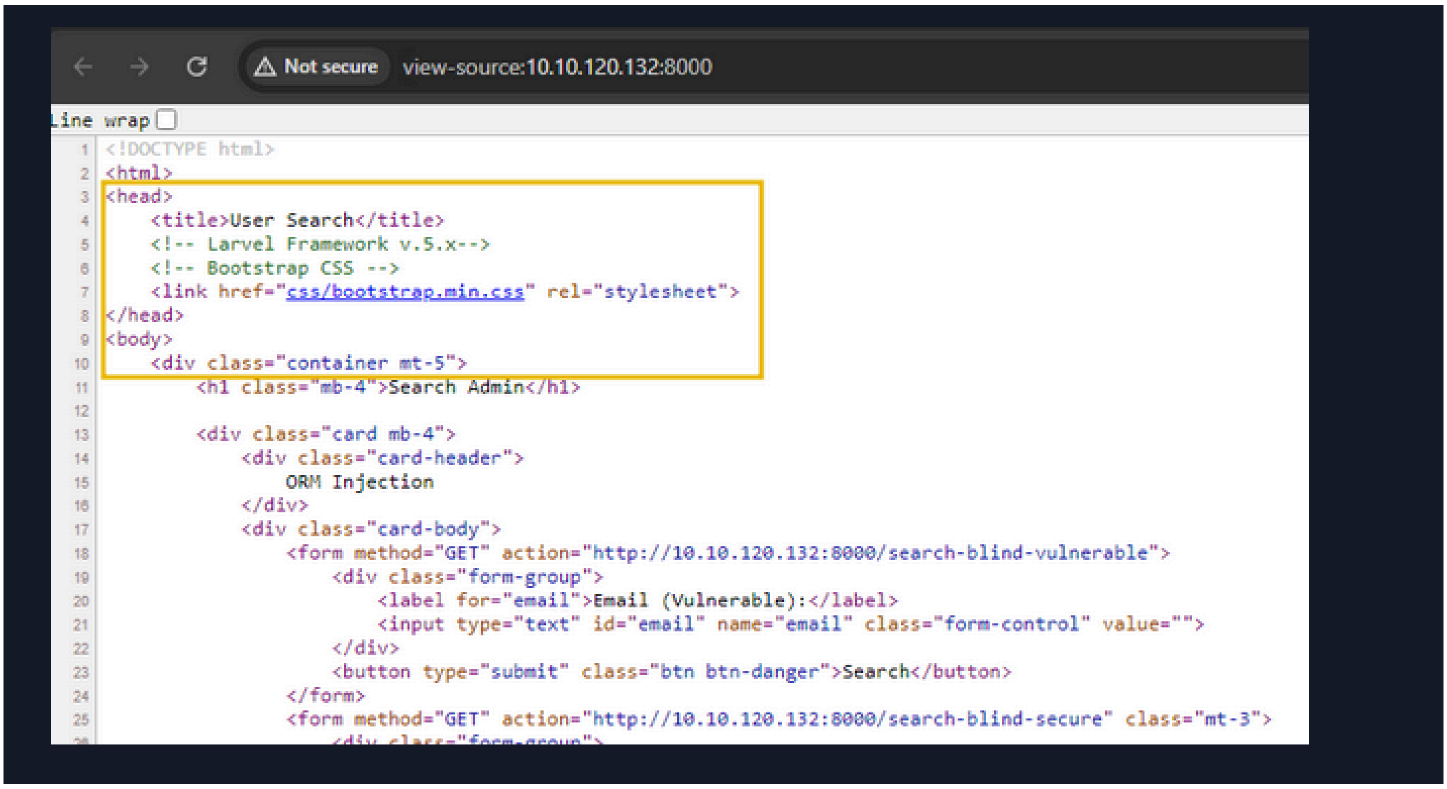
Identifying the Framework

Determining the underlying framework of a web application can reveal its ORM library and potential vulnerabilities. Consider these techniques:

1. **Examining Cookies**
   Many frameworks use unique formats for their session cookies:
   - Example: Laravel cookies often contain prefixes like laravel_session.
2. **Inspecting Source Code**
   Review the HTML source for comments, meta tags, or embedded scripts that might

indicate the framework. While not always definitive, these hints can point you in the right direction.

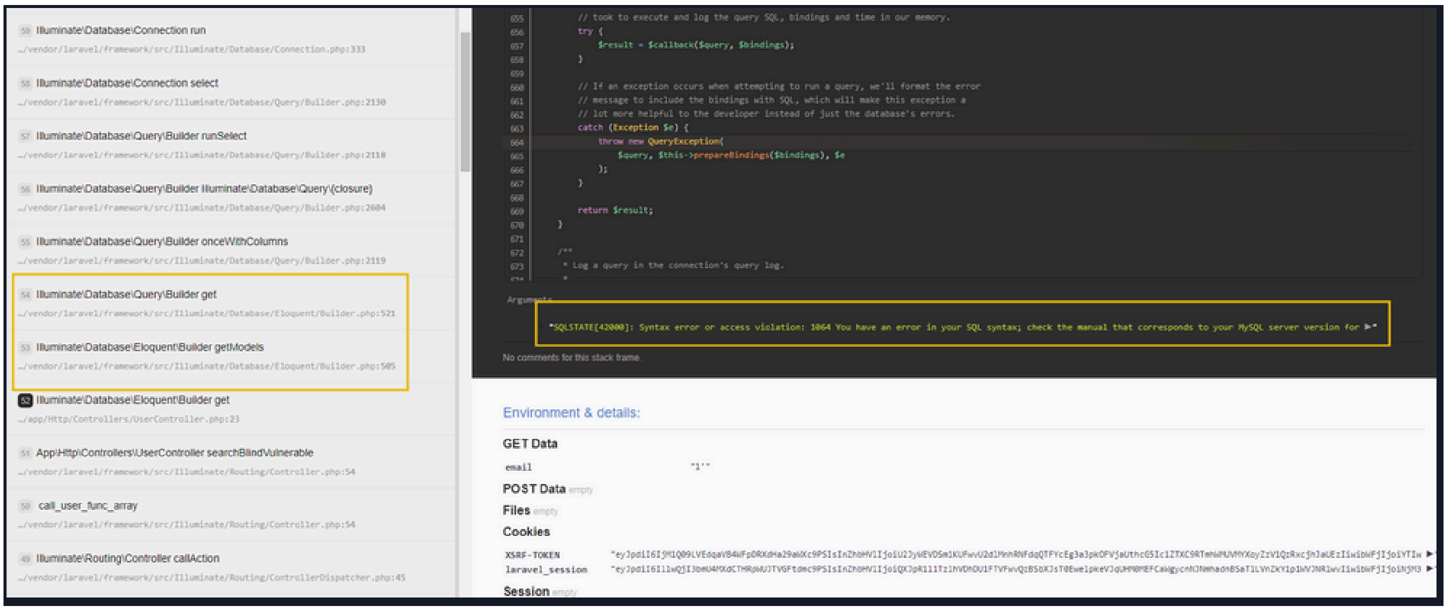
3. **Analyzing HTTP Headers**
   Use tools like **Burp Suite** or browser developer tools to examine HTTP headers. Server technologies, framework-specific versions, or other identifiers might be revealed.
4. **Reviewing URL Structures**
   Frameworks often have distinct URL patterns. For example:
   - Laravel: /users/edit/1
   - Django: /users/edit?id=1
5. **Observing Login and Error Pages**
   Frameworks may have distinctive error messages or authentication pages:
   - Example: Laravel's errors might include phrases like SQLSTATE.



```
<!DOCTYPE html>
<html>
<head>
    <title>User Search</title>
    <!-- Larvel Framework v.5.x-->
    <!-- Bootstrap CSS -->
    <link href="css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container mt-5">
        <h1 class="mb-4">Search Admin</h1>

        <div class="card mb-4">
            <div class="card-header">
                ORM Injection
            </div>
            <div class="card-body">
                <form method="GET" action="http://10.10.120.132:8000/search-blind-vulnerable">
                    <div class="form-group">
                        <label for="email">Email (Vulnerable):</label>
                        <input type="text" id="email" name="email" class="form-control" value="">
                    </div>
                    <button type="submit" class="btn btn-danger">Search</button>
                </form>
                <form method="GET" action="http://10.10.120.132:8000/search-blind-secure" class="mt-3">
```

CHECKING THE SOURCE CODE



```
59 Illuminate\Database\Connection run
…/vendor/laravel/framework/src/Illuminate/Database/Connection.php:333

58 Illuminate\Database\Connection select
…/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php:2130

57 Illuminate\Database\Query\Builder runSelect
…/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php:2118

56 Illuminate\Database\Query\Builder Illuminate\Database\Query\{closure}
…/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php:2604

55 Illuminate\Database\Query\Builder onceWithColumns
…/vendor/laravel/framework/src/Illuminate/Database/Query/Builder.php:2119

54 Illuminate\Database\Query\Builder get
…/vendor/laravel/framework/src/Illuminate/Database/Eloquent/Builder.php:521

53 Illuminate\Database\Eloquent\Builder getModels
…/vendor/laravel/framework/src/Illuminate/Database/Eloquent/Builder.php:505

52 Illuminate\Database\Eloquent\Builder get
…/app/Http/Controllers/UserController.php:23

51 App\Http\Controllers\UserController searchBlindVulnerable
…/vendor/laravel/framework/src/Illuminate/Routing/Controller.php:54

50 call_user_func_array
…/vendor/laravel/framework/src/Illuminate/Routing/Controller.php:54

49 Illuminate\Routing\Controller callAction
…/vendor/laravel/framework/src/Illuminate/Routing/ControllerDispatcher.php:45
```

```
        // took to execute and log the query SQL, bindings and time in our memory.
        try {
            $result = $callback($query, $bindings);
        }

        // If an exception occurs when attempting to run a query, we'll format the error
        // message to include the bindings with SQL, which will make this exception a
        // lot more helpful to the developer instead of just the database's errors.
        catch (Exception $e) {
            throw new QueryException(
                $query, $this->prepareBindings($bindings), $e
            );
        }

        return $result;
    }

    /**
     * Log a query in the connection's query log.
     *
```

```
Arguments
  "SQLSTATE[42000]: Syntax error or access violation: 1064 You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for ▶"

No comments for this stack frame.
```

Environment & details:

```
GET Data
email                 "1'"
POST Data empty
Files empty
Cookies
XSRF-TOKEN        "[illegible] ▶"
laravel_session   "[illegible] ▶"
Session empty
```

INSPECTING ERROR MESSAGES

Test Input

Enter **1'** into the "Email (Vulnerable)" input field to test how the application handles this data. This payload is designed to disrupt the query structure.

Observing the Response

If the application generates an error such as:

SQLSTATE[42000]: SYNTAX ERROR OR ACCESS VIOLATION

It indicates improper handling and concatenation of user input in the query. Such messages, along with specific query patterns, are characteristic of Laravel's Eloquent ORM.

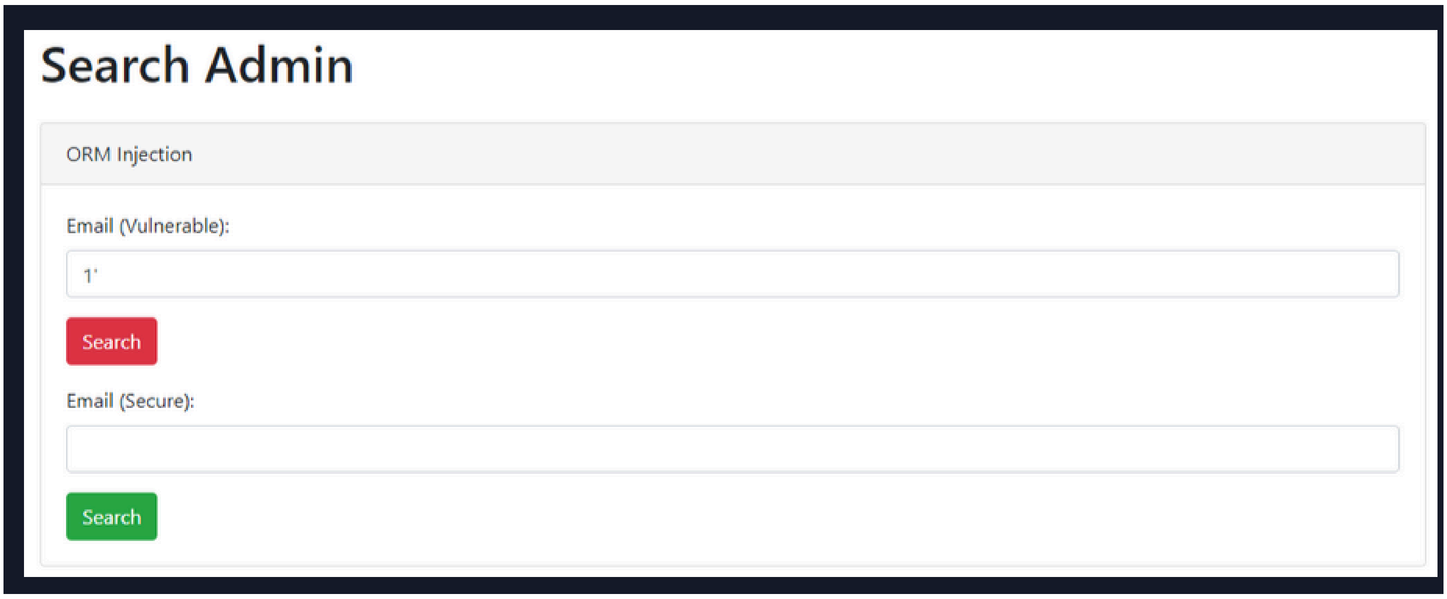
Insights for Red Teamers

For red teamers, uncovering ORM injection vulnerabilities is a vital step in assessing an application’s security posture. Exploiting these vulnerabilities can allow unauthorized database access or manipulation.

Key Techniques

1. **Manual Code Review**: Examine the source code for risky query methods like whereRaw() or concatenation of inputs.
2. **Automated Scanning**: Use tools like **Burp Suite**, **SonarQube**, or **SQLMap** to identify injection-prone patterns.
3. **Input Validation Testing**: Inject test payloads (1' OR '1'='1 or 1'; DROP TABLE users; --) to disrupt query execution and reveal vulnerabilities.
4. **Error-Based Testing**: Trigger errors intentionally to gather information about the underlying ORM and query construction.

ORM INJECTION - WEAK IMPLEMENTATION


Search Admin

ORM Injection

Email (Vulnerable):

1'

Email (Secure):

Reviewing the Vulnerable Source Code

The developer's original function, searchBlindVulnerable(), demonstrates how improper handling of user inputs can lead to vulnerabilities, specifically ORM injection. Here’s a detailed breakdown:

```
public function searchBlindVulnerable(Request $request)
{
   $email = $request->input('email');
   $users = Admins::where('email', $email)->get();

   if ($users->isNotEmpty()) {
      return view('user', ['users' => $users]);
   } else {
        return view('user', ['message' => 'User not found']);
    }
}
```

What This Function Does:

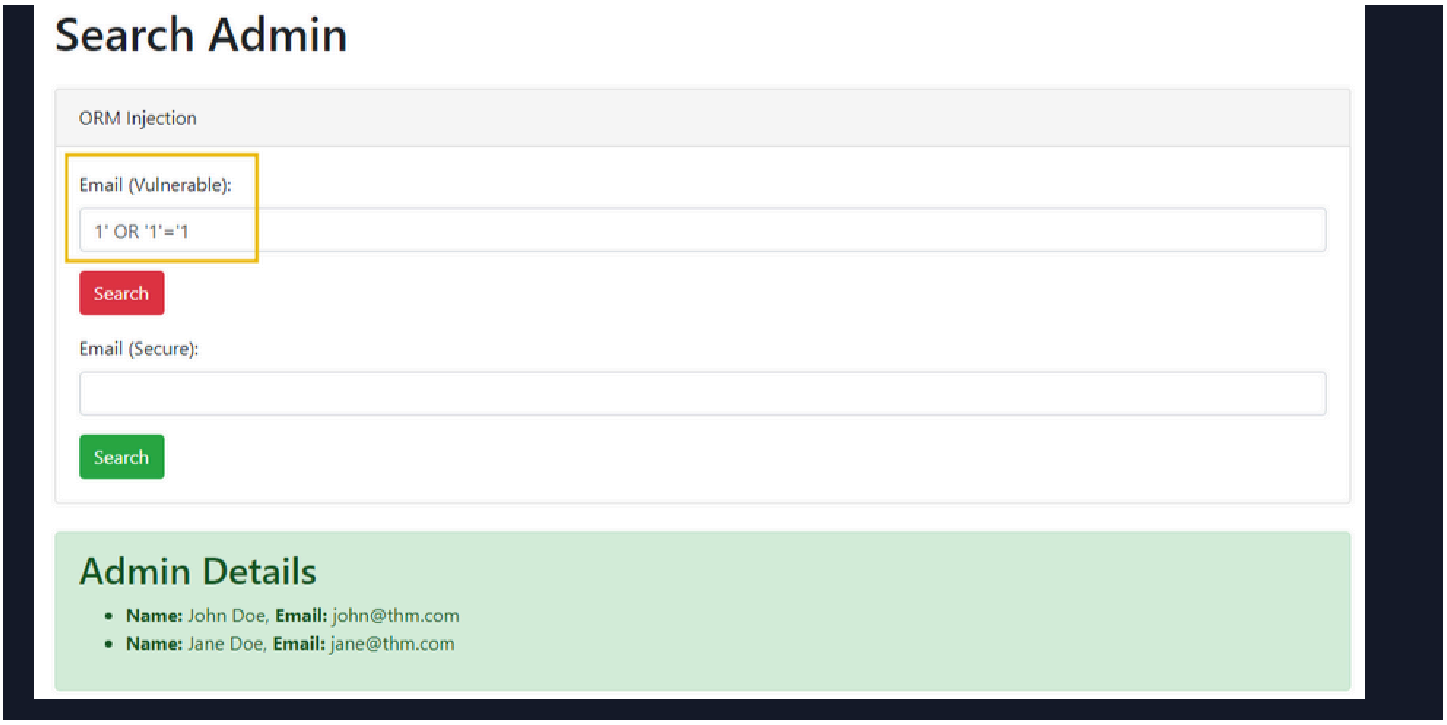
1. **Input Retrieval**: It extracts the email parameter from the HTTP request using $request->input('email').
2. **Query Construction**: The whereRaw() method creates a raw SQL query that incorporates the email input directly into the query string.
3. **Query Execution**: Executes the SQL query and stores the results in the $users array.
4. **Response**: If users are found, it renders a view with the user data. If not, it displays a "User not found" message.

Why It's Vulnerable:

The use of whereRaw() allows direct injection of user input into the SQL query. This exposes the application to ORM injection attacks, as user-provided inputs are not sanitised or escaped.

For example, inputting 1' OR '1'='1 results in the query:

```
SELECT * FROM users WHERE email = '1' OR '1'='1';
```


Search Admin

ORM Injection

Email (Vulnerable):

1' OR '1'='1

Email (Secure):

Admin Details

- **Name:** John Doe, **Email:** john@thm.com
- **Name:** Jane Doe, **Email:** jane@thm.com

Detailed Exploitation

1. **Input Malicious Value**: An attacker submits 1' OR '1'='1 in the email input field.
2. **Query Construction**: The raw SQL query becomes:

```
$users = User::whereRaw("email = '1' OR '1'='1'")->get();
```

1. **Query Execution**: The constructed SQL executes and retrieves all user records.
2. **Outcome**: The attacker gains access to sensitive user data, potentially compromising the application's security.

This scenario underscores the danger of raw query methods combined with unsanitised user input.