# NoSQL Injection Vulnerability in Laravel

## Introduction

This document demonstrates a NoSQL injection vulnerability in a Laravel web application, where user input is improperly sanitized, allowing an attacker to bypass authentication. Additionally, it includes an exploitation script and a secure code fix to mitigate the vulnerability.

## Vulnerable Code

The vulnerable code is a login mechanism that allows a NoSQL injection attack on the email field. By failing to properly sanitize user input, the application allows malicious NoSQL operators to manipulate MongoDB queries.

**Vulnerable Code (LoginController.php):**

    public function vulnerableLogin(Request $request)

    {

        $email = $request->input('email');  *// Directly takes user input*

*// Vulnerable query allowing NoSQL injection*

        $emailQuery = json\_decode($email, true);

        $user = User::whereRaw([

            'email' => $emailQuery  *// Unsanitized user input injected here*

        ])->first();

        if ($user) {

            auth()->login($user);

            return redirect('/home')->with('status', 'Welcome ' . $user->name);

        } else {

            return back()->withErrors('Invalid login credentials.');

        }

    }

**Issue**: The email input is not validated or sanitized and is directly injected into the query. This allows an attacker to exploit the NoSQL database (MongoDB) by passing JSON-based query operators (e.g., $gt) into the input fields, bypassing authentication.

{"$gt": ""}

By exploiting this vulnerability, an attacker can bypass the login process without knowing the correct credentials. The attacker can manipulate the email field using NoSQL injection to match any user in the database.

## Security Enhanced Code

To mitigate this vulnerability, proper input validation and sanitization should be applied. Specifically, ensure that the email field is validated as a valid email and that no direct raw inputs are allowed in MongoDB queries.

**Security Enhanced Code (LoginController.php):**

    public function secureLogin(Request $request)

    {

*// Validate input to prevent NoSQL injection*

        $credentials = $request->validate([

            'email' => 'required|email',

            'password' => 'required'

        ]);

*// Use Laravel's attempt method for secure authentication*

        if (auth()->attempt($credentials)) {

            return redirect('/home')->with('status', 'Login successful');

        }

        return back()->withErrors('Invalid login credentials.');

    }

**Fix**: The input is validated using Laravel's built-in validation rules ('email' => 'required|email'). This ensures that the email input is a valid email address and does not contain any malicious query operators.

**Authentication**: Laravel's auth()->attempt() method is used to handle authentication securely, using hashed passwords and avoiding raw query execution.

# DOM-Based XSS Vulnerability in Laravel

## Introduction

Cross-Site Scripting (XSS) is a common web application vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. This type of attack occurs when a web application includes untrusted data in a web page without proper validation or escaping. In our case, we've created a simple demonstration of a DOM-based XSS vulnerability.

The vulnerable page we've created allows users to input their name, which is then displayed on the page as a greeting. However, due to improper handling of user input, this page is susceptible to XSS attacks. Let's examine why this vulnerability exists, how it can be exploited, and how to fix it.

## Vulnerable Code

The vulnerability arises from the use of the innerHTML property to insert user input directly into the DOM, allowing any HTML or JavaScript within the input to be interpreted and executed. For example, if you input <img src="x" onerror="alert('XSS Attack!')">, the innerHTML property processes this as HTML. The browser attempts to load an image from the source "x", which fails, triggering the onerror event. This event, in turn, executes the JavaScript code alert('XSS Attack!'), leading to a successful XSS attack.

**Vulnerable Code (xss-demo.blade.php):**

<script>

    function greetUser() {

        var userInput = document.getElementById('name').value;

*// Vulnerable line:*

        document.getElementById('greeting').innerHTML = userInput;

    }

</script>

## Security Enhanced Code

In this fixed version, multiple layers of protection have been implemented to prevent XSS attacks. First, input sanitization is applied using the DOMPurify library, which cleanses user input by removing any potentially malicious code while preserving safe HTML if necessary. Second, secure DOM manipulation is used by replacing the vulnerable innerHTML property with safer alternatives such as textContent or sanitized innerHTML. Lastly, HTML escaping is included through a custom escapeHTML function, which replaces special characters with their HTML entity equivalents, preventing them from being interpreted as HTML or JavaScript.

**Security Enhanced Code (xss-demo-fixed.blade.php):**

DOMPurify Integration:

 <script src="https://cdnjs.cloudflare.com/ajax/libs/dompurify/2.3.3/purify.min.js"></script>

Safe DOM Manipulation:

 document.getElementById('greeting').innerHTML = DOMPurify.sanitize(userInput);

HTML Escaping Function:

function escapeHTML(*str*) {

       return str.replace(/[&<>'"]/g,

*tag* => ({

               '&': '&amp;',

               '<': '&lt;',

               '>': '&gt;',

               "'": '&#39;',

               '"': '&quot;'

           }[tag] || tag)

       );

   }

Final Method:

function greetUser() {

            var userInput = document.getElementById('name').value;

            console.log("User input:", userInput); *// Log input*

*// Method 1: Use textContent (safest, but doesn't allow any HTML)*

*// document.getElementById('greeting').textContent = userInput;*

*// Method 2: Use escapeHTML function*

*// document.getElementById('greeting').innerHTML = escapeHTML(userInput);*

*// Method 3: Use DOMPurify (allows safe HTML if needed)*

            document.getElementById('greeting').innerHTML = DOMPurify.sanitize(userInput);

*// Debug output*

            document.getElementById('debug').textContent = "Sanitized input: " + DOMPurify.sanitize(userInput);

        }