



---

# **INDIVIDUAL ASSIGNMENT**

**LEVEL 5**

**COMP50011**

**Server Side Programming 2**

**COM2421/SENG2421**

**Name – Thiviru Dilmith Opallage**

**CB number – CB012630**

---

## **INSTRUCTION TO CANDIDATES**

- 1. Students are advised to underpin their answers with the use of references (cited using the Harvard Referencing Style).**
- 2. Late submission will be awarded zero (0) unless extenuating circumstances (EC) are upheld.**
- 3. Cases of plagiarism will be penalized.**
- 4. The assignment should be submitted as a softcopy:**
  - a. The softcopy of the written assignment and source code must be uploaded to given link in the LMS.**

## Table of Contents

1.	Introduction.....	4
2.	System Overview .....	5
2.1.	Overall architecture.....	5
2.2.	Technology stack .....	6
2.3.	Database design .....	7
2.4.	System modules .....	8
3.	Security threats and Mitigation (OWASP Top 10).....	10
3.1.	Broken access control .....	10
3.2.	Cryptographic Failures.....	11
3.3.	SQL Injection.....	11
3.4.	Insecure Design.....	11
3.5.	Security misconfiguration.....	12
3.6.	Vulnerable and Outdated Components .....	13
3.7.	Identification and Authentication Failures.....	13
3.8.	Software and Data Integrity Failures .....	14
3.9.	Security Logging and Monitoring Failures.....	14
3.10.	Server-Side Request Forgery (SSRF) .....	15
4.	Threats and mitigations.....	15
4.1.	SQL injection .....	15
4.2.	Broken authentication .....	16
4.3.	Sensitive data exposure.....	17
4.4.	Cross-site scripting (XSS).....	17
4.5.	Cross-site request forgery (CSRF).....	18
4.6.	Role-based access control (RBAC) .....	19

4.7. Secure error handling and logging .....	19
5. Deployment security considerations .....	20
6. Future enhancements .....	21
7. GitHub link .....	21
8. References.....	21

## List of Figures

1. Laravel folder structure showcasing MVC folder structure .....	5
2. Product.php model showing relationships (hasMany).....	6
3. Livewire component .....	7
4. Use of livewire component in blade file .....	7
5. Database Migration Schema .....	8
6. MongoDB showing products collection .....	8
7. Authenticated routes using Jetstream.....	9
8. Use of Sanctum to generate a login token .....	9
9. Composer audit screenshot .....	13
10. Vulnerable example for SQL injections.....	15
11. Mitigation example for SQL injections.....	16
12. Vulnerable example for the broken authentication .....	16
13. Mitigation example for Broken authentication .....	16
14. Vulnerable example for sensitive data exposure.....	17
15. Mitigation example for sensitive data exposure .....	17
16. Vulnerable example for XSS .....	18
<b>17. Mitigation example for XSS .....</b>	<b>18</b>
18. Vulnerable example for CSRF .....	18
19. Mitigation example for CSRF.....	19
20. env Database configuration.....	20

## 1. Introduction

BiteOasis is a web-based platform developed using PHP, Tailwind CSS and MySQL, that focuses on delivering and showcasing fast food. This web platform is designed to allow customers to seamlessly browse through items, add items to their cart, which are delivered. While remaining lightweight and easy to use locally, it mimics the key features of a modern online food ordering system. The project highlights the core features of web development concepts such as session management, CRUD operations, role based access control, and dynamic page rendering using PHP.

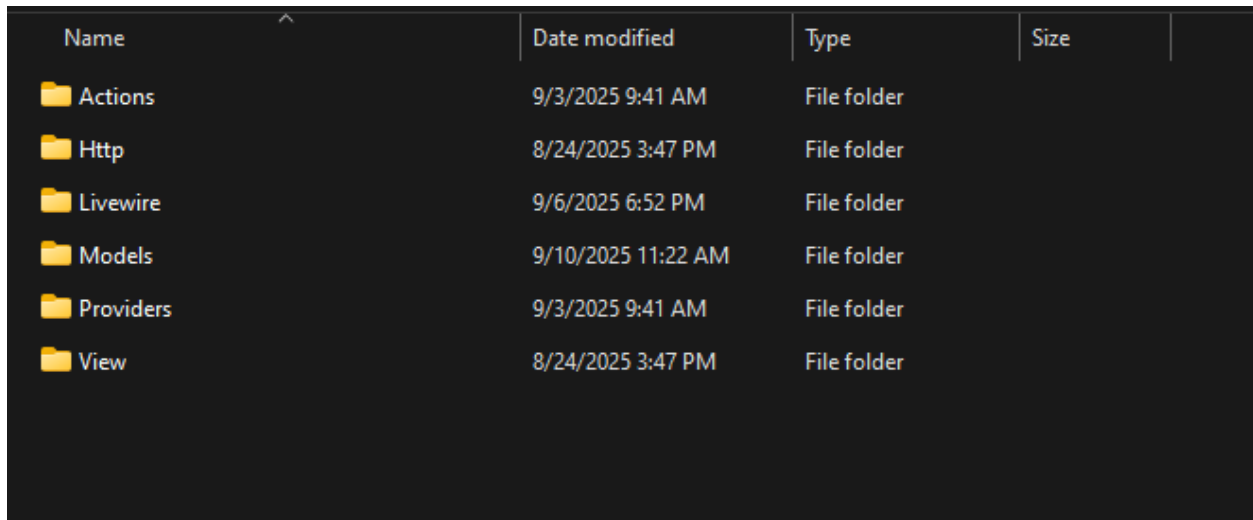
In accordance of the assignment requirements, this web based platform was rebuilt using Laravel 11, showcasing a modern PHP framework. The transition to Laravel 11 allowed this platform to follow Model-View-Controller (MVC) architecture, utilize built-in authentication functions, manage routing and handle database interactions through OMR Eloquent ORM, enhancing the platform's security, maintainability and scalability. This transition showcases the differences between raw PHP development and framework driven development.







Objective of this report is to discuss both generic and Laravel-specific threats that have been identified and addressed, using the OWASP top 10 framework for analysis. It also outlines the security threats implemented during the transition such as CSRF protection, HTTPS enforcement, secure session handling, input validation, role based access control and Laravel policies and gates.

## 2. System Overview

### 2.1.Overall architecture

The redeveloped BiteOasis web platform was developed in accordance with the MVC architecture framework provided by Laravel. All business entities that are associated with the database are managed through Laravel's Eloquent ORM model and represented inside the model's folder. By managing the database through an application layer, Eloquent model provides a simplified interface that allows a clean interaction with the database. Using parameter binding, Laravel's Eloquent ORM and query builder helps to prevent SQL injections. Views folder which is responsible for displaying dynamic content and keeping the presentation layer separate from the business logic acts a central location to store and serve the User Interface. Controllers, containing logic for handling requests, generating requests and ensuring only authorized users access restricted webpages (e.g., Admin Dashboard), act as intermediaries between the models and views.

A screenshot of a file explorer window with a dark theme. It displays a table of folders within a Laravel project. The table has four columns: 'Name', 'Date modified', 'Type', and 'Size'. There are six rows, each representing a folder: 'Actions', 'Http', 'Livewire', 'Models', 'Providers', and 'View'. Each row shows the folder's name with a yellow folder icon, its last modified date and time, its type as 'File folder', and its size, which is currently blank for all entries.

Name	Date modified	Type	Size
 Actions	9/3/2025 9:41 AM	File folder	
 Http	8/24/2025 3:47 PM	File folder	
 Livewire	9/6/2025 6:52 PM	File folder	
 Models	9/10/2025 11:22 AM	File folder	
 Providers	9/3/2025 9:41 AM	File folder	
 View	8/24/2025 3:47 PM	File folder	

*1. Laravel folder structure showcasing MVC folder structure*

```

class Product extends Model
{
    protected $fillable = ['name', 'description', 'price'];

    // A product can appear in many cart items
    public function carts(){
        return $this->hasMany(Cart::class);
    }

    // A product can belong to many order items
    public function orderItems(){
        return $this->hasMany(OrderItem::class);
    }

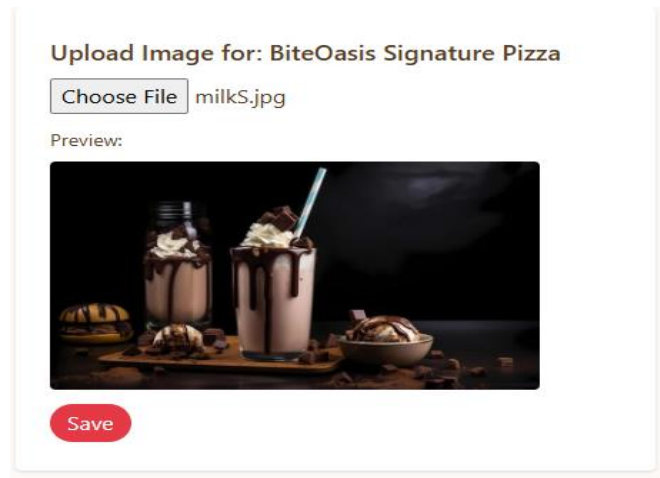
    // A product can have multiple offers
    public function offers(){
        return $this->hasMany(Offers::class);
    }
}

```

*2. Product.php model showing relationships (hasMany)*

## 2.2. Technology stack

Laravel 11 framework ensures that the user gets a clean, modern, responsive interface by utilizing blade components along with Tailwind CSS for stylings. For security, Laravel provides built-in features such as routing, authentication, and middleware. MySQL acts as the primary database to store user data, products and order information. Laravel Jetstream handles user authentication and role management, while Laravel sanctum ensures secure interactions by providing token based API authentication. And to enhance the user experience without constantly having to reload the webpage Livewire was utilized to provide a reactive functionality for product management and cart management.



3. Livewire component

```
@error('description')
|   <div class="text-red-500">{{ $message }}</div>
@enderror

<livewire:product-image-upload :productId="$product->id" />

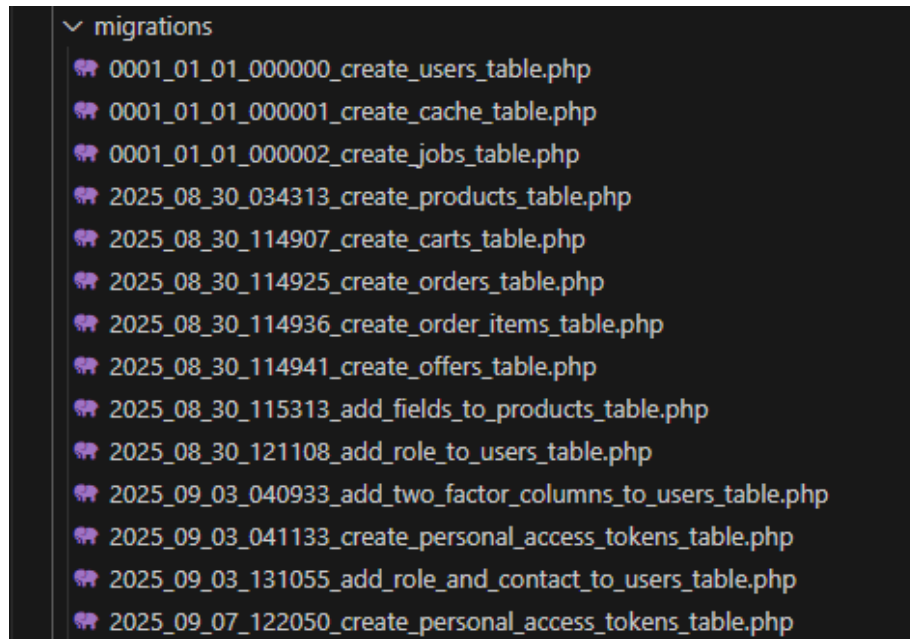
<button class="bg-[#E63946] text-white px-4 py-2 rounded-full
</x-layout>
```

4. Use of livewire component in blade file

### 2.3. Database design

The database follows a relational schema with Users, Products and Orders acting as the key entities. The User entity stores details about the user such as their role identifier, credentials and hashed password. The product entity store details about the product name, price, and category. Order entity uses foreign keys to map users to their associated orders. Eloquent ORM ensures data integrity and protection against SQL injections by enabling relationships such as one-to-many and many-to-many between entities.

Alongside MySQL, MongoDB was integrated for API endpoints allowing flexible NoSQL data handling. To demonstrate dual database usage, product details were stored in MongoDB to provide scalable, document based storage for API driven access.



#### 5. Database Migration Schema



#### 6. MongoDB showing products collection

### 2.4. System modules

Laravel Jetstream differentiates admins from regular users by providing role-based access to product management, allowing admins to add, update or delete products using Eloquent ORM. CSRF protection ensures ordering and cart modules provide users with secure form submissions. Laravel Sanctum enables API functionality, allowing token based access for mobile and third party applications. Based on user roles, middleware such as auth, verified and custom gates ensures controlled access to routes.



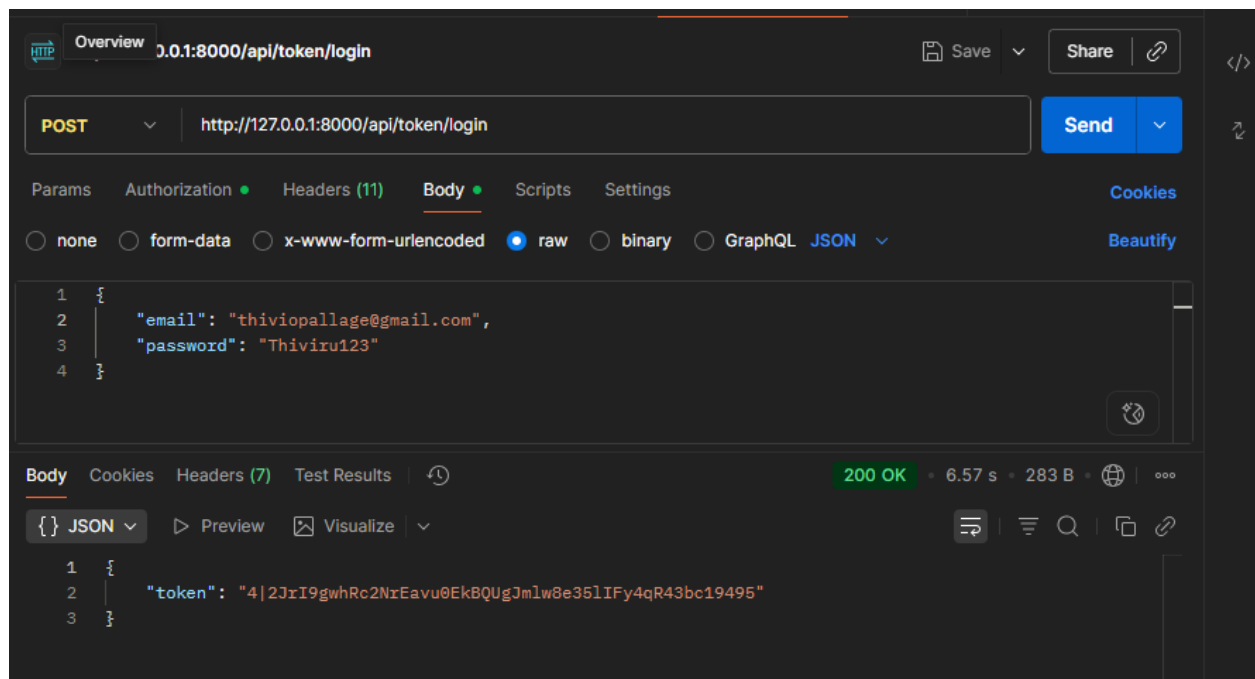
```
//Logged in user routes
Route::middleware(['auth'])->group(function (){
    Route::get('dashboard', function() {
        return view('dashboard');
    })->name('dashboard');

    // Profile
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');

    //Cart
    Route::get('/cart', [CartController::class, 'displayC'])->name('displayC');
    Route::post('/cart/add/{productId}', [CartController::class, 'addC'])->name('cart.addC');
    Route::post('/cart/update/{id}', [CartController::class, 'updateC'])->name('cart.updateC');
    Route::delete('cart/remove/{id}', [CartController::class, 'removeC'])->name('cart.removeC');

    //Orders and checkout
    Route::get('/checkout', [UserOrderController::class, 'checkout'])->name('orders.checkout');
    Route::post('/checkout', [UserOrderController::class, 'placeOrder'])->name('orders.place');
    Route::get('/orders', [UserOrderController::class, 'displayO'])->name('orders.displayO');
});
```

## 7. Authenticated routes using Jetstream



The screenshot shows a web browser interface for a REST client (Postman) with the following details:

- Overview:** The URL is `0.0.1:8000/api/token/login`.
- Method:** `POST`.
- Body:** The request body is a JSON object:
 

```
{
  "email": "thiviopallage@gmail.com",
  "password": "Thiviru123"
}
```
- Response:** The status is `200 OK` with a response time of `6.57 s` and a body size of `283 B`. The response body is a JSON object:
 

```
{
  "token": "4|2JrI9gwhRc2NrEavu0EkBQUgJm1w8e35lIFy4qR43bc19495"
}
```

## 8. Use of Sanctum to generate a login token

### 3. Security threats and Mitigation (OWASP Top 10)

OWASP Top 10 is a list published by the Open Web Application Security Project (OWASP) that highlights the most critical security threats faced by modern web applications and provide a guidance on how to mitigate them. Cyber attackers primarily target web applications due to their widespread use and high availability across the internet. These attacks can lead to data breaches, financial loss for organizations and loss of user trust.

It serves as a reference for secure coding practices for developers, while acting as a risk management tool for organizations. By addressing these security concerns, organizations can improve their application security, increase awareness among developers and organizations, strengthens user trust and encourage organizations to prevent attacks rather than simply reacting to them.

The risks covered in the list include, Broken Access Control, Cryptographic Failures, Injection, Insecure Design, Security Misconfigurations, Vulnerable and Outdated Components, Identification and Authentication Failures, Software and Data Integrity Failures, Security Logging and Monitoring Failures, and Server-Side Request Forgery (SSRF).

#### 3.1. Broken access control

Broken access control in a web platform occurs when users are able to perform functions or access data that they aren't allowed to or are not authorized to. Broken access control in the OWASP Top 10 refers to how an application enforces user permissions to control which actions each user can perform within the platform. For an example, regular user should not be able to access the admin dashboard and perform admin functionalities or modify URLs to view or edit other user's data.

Laravel provides middleware, such as auth, to restrict access to specific routes based on user roles. Instead of solely relying on hiding buttons or links in the frontend, access rules should be enforced and validated on the backend. A user's role should be checked and authenticated before performing any sensitive actions. Laravel offers policies and gates for fine-grained control over user actions on models or resources. Furthermore, using route model binding together with policies ensures that users can only access the resources they are authorized to use. By implementing these policies broken access control in web applications can be mitigated.

### 3.2.Cryptographic Failures

Sensitive information such as passwords, credit card information, personal data or authorization tokens can be stolen or manipulated from attackers when cryptography is not properly imposed or used. In such cases the web applications fails to protect sensitive data both at rest and in transit.

Cryptography can be effectively mitigated by implementing the following measures:

- Instead of plaintext storing passwords using Laravel's hashing functions.
- Sensitive credentials such as API keys and database passwords should be stored inside the .env file without hardcoding them.
- Use Laravel's built-in encryption functionalities to protect sensitive fields.
- To ensure data in transit is encrypted, enforce HTTPS in routes and server configurations.

### 3.3.SQL Injection

SQL injections is a vulnerability that allows attackers to insert malicious input (e.g., email = ' OR '1'='1) into user input fields, URL parameters, HTTP headers, API requests, attackers have the ability to read or modify data, execute unintended commands or even take control of the application. This is referred to as SQL injection in OWSAP top 10 list. For example, harmful SQL code entered into a login form can be used to steal database information or bypass authentication.

To mitigate this issue, developers must ensure that user input is not directly passed to OS commands or other interpreters. To prevent SQL injections, the platform must be built using Laravel's Eloquent ORM and query builder (e.g., User::where('email', \$email)→first()), which automatically escapes user input. When raw SQL queries are unavoidable developers should employ bound parameters and additional safeguard practices such as input validation, restricted database privileges and web application firewall should be adopted.

### 3.4.Insecure Design

Threats that arise when secure workflows, threat modeling, and proper access control are not incorporated during the development stage are referred to as Insecure Design in the OWASP Top 10 list. Even if developers follow the best coding standards, neglecting these security concerns leaves the application vulnerable, as the issue lies within the overall system design.

To mitigate this issue the following practices can be followed:

- Security measures should be planned from the early stages of the development.
- Using Laravel's built-in mechanism such as middleware, gate and policies to enforce authentication and authorization.
- Using Laravel's built-in resources such as Jetstream to enhance authentication.
- Incorporating threat modeling during the development phase to anticipate possible threats and strength the system architecture.
- Alongside framework features adopt secure design principles and patterns.

### 3.5. Security misconfiguration

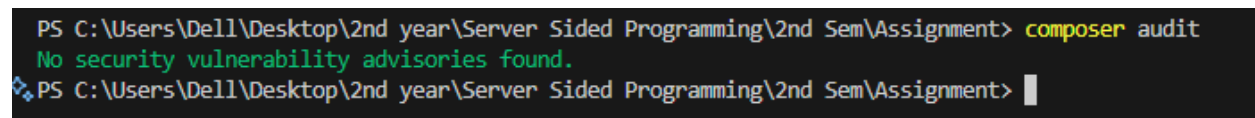
In OWASP top 10 list, security misconfiguration is defined as a system that is not securely configured. This issue arises in the system when unnecessary services, ports or features are enabled, increasing the attack surface. It may also result from the use of outdated and unpatched systems, exposure of detailed error messages about the system to the users, unchanged default configurations such as admin credentials unchanged, or missing and inconsistently applied security settings. These misconfigurations leave gaps in the system that attackers can exploit.

One of the biggest risks that developers face is security misconfiguration. These risks can be avoided by developing these applications using Laravel framework and properly setting up configurations during development. An important step in addressing this issue is disabling all the unnecessary features or services. For example, APP\_DEBUG mode must be disabled in production environments. On top of that, developers should keep all third-party packages up to date, since outdated or unpatched software often becomes a target for attackers. Rather than displaying detailed error messages to users, errors should be logged securely using Laravel's built-in error handling tools. It is also crucial to avoid common mistakes such as leaving default admin credentials unchanged or exposing the .env file to be publicly assessable. Security can be further strengthened by enforcing HTTPS, applying proper file permissions and taking advantage of Laravel's access control tools such as middleware, gates, and policies.

### 3.6. Vulnerable and Outdated Components

When an application depends on outdated or insecure libraries, frameworks or software modules with known vulnerabilities it creates an opening for attackers to exploit. This is referred to as use of vulnerable and outdated components. By exploiting these flaws attackers can compromise the system and gain access to sensitive data even if the application's code is secure. Use of outdated Laravel, PHP, third party packages or JavaScript libraries are some common examples for vulnerabilities that are reported publicly.

Eliminating unwanted and unmaintained libraries and regularly updating Laravel framework and all third-party libraries helps to mitigate this issue. Composer audit command helps developers to quickly identify known vulnerabilities within the system and resolve them properly. Combining automated testing along with secure coding standards minimizes the likelihood of new risks being introduced through dependencies.



```
PS C:\Users\Dell\Desktop\2nd year\Server Sided Programming\2nd Sem\Assignment> composer audit
No security vulnerability advisories found.
❖ PS C:\Users\Dell\Desktop\2nd year\Server Sided Programming\2nd Sem\Assignment> |
```

*9. Composer audit screenshot*

### 3.7. Identification and Authentication Failures

When authentication is not properly enforced, attackers impersonate themselves as legitimate users to gain system access. Passwords that are stored in plain text rather than hashed passwords often results in insecure session handling, inadequate recovery features, predictable passwords.

This issue can be addressed by following the strategies below:

- Utilizing Laravel's built-in hashing features to securely store passwords.
- Implementing multi-factor authentication through Jetstream.
- Through the use of secure cookies, session timeouts, appropriate session handling and invalidating sessions, any unauthorized access to the system can be prevented.
- Setting up secure password recovery mechanisms.
- Restricting login attempts through middleware.

### 3.8. Software and Data Integrity Failures

When an application rely on untrusted or unverified components or does not properly verify software updates, it creates an opportunity for attackers to insert hidden malicious code into a software package, which then becomes a part of the system. If integrity checks are missing, attackers may target critical application data, which could compromise the system security. This can lead to software and data integrity failures.

To mitigate this issue the below practices can be followed:

- Keeping all Laravel dependencies up to date.
- Avoid unverified or unknown third party packages and install composer packages only from trusted and verified sources.
- Commit composer.lock file to version control to unauthorized changes and control dependency changes.
- Restrict access to CI/CD pipelines to trusted personal only.
- Protect sensitive data through Laravel's built-in encryption features.

### 3.9. Security Logging and Monitoring Failures

When applications fails to log security related events regarding unsuccessful login attempts, unauthorized attempts to access restrict resources or any other persistent suspicious activity about the system creates an opportunity to function inside the system undetected.

The following the strategies address the problem below:

- Restrict permissions and ensure log files are securely stored and not accessible to the public.
- Set up alerts to log failed login attempts, or any other abnormal activities.
- Regularly log reviews that detect attacks early.

In addition to Monolog, developers can enhance visibility through Laravel Telescope to monitor requests and exceptions by integrating with third party tools such as Sentry or ELK stack to deliver real time alerts and intrusion detection.

### 3.10. [Server-Side Request Forgery \(SSRF\)](#)

Without connecting to trusted external services, attackers supply malicious URL that tricks the application to send requests to internal systems from its own server. For example, if the application allow users to fetch an image through a URL, an attacker could provide a link to an internal server, gaining access to unauthorized to sensitive data.

The following the strategies address the problem below:

- Validate and sanitize user input.
- Block internal network access.
- Leverage Laravel security features to enforce.
- Monitor and log unusual requests.
- Disable unnecessary HTTP requests.
- Permit requests to trusted and approved external domains only.

## 4. [Threats and mitigations](#)

This section provides sample codes demonstrating how Laravel handles common risks, on selected OWSP Top 10 vulnerabilities such as SQL injection, Broken authentication, and sensitive data exposure. These examples highlight how Laravel's built-in practical mechanisms features protect against such issues in practice.

### 4.1. [SQL injection](#)

**Vulnerable example:**

```
Route::get('/vuln-sql', function (Illuminate\Http\Request $request) {  
    $email = $request->query('email');  
  
    $user = DB::select("SELECT * FROM users WHERE email = '$email'");  
  
    return $user ?: 'No user found';  
});
```

*10. Vulnerable example for SQL injections*

## Mitigation example:

```
// Update order status
public function updateStatus(Request $request, $id){
    $validated = $request->validate([
        'status' => 'required|in:pending,processing,completed,cancelled'
    ]);

    $order = Order::findOrFail($id);
    $order->update($validated);

    return redirect()->route('admin.orders.show0')->with('Successful', 'Order status updated successfully'
}
}
```

11. Mitigation example for SQL injections

## 4.2. Broken authentication

### Vulnerable example:

```
Route::post('/logout', function(Request $request) {
    auth()->logout();
    return redirect('/login');
})->name('logout');
```

12. Vulnerable example for the broken authentication

### Mitigation example:

```
Route::post('/logout', function(Request $request) {
    $user = $request->user();

    //If user has an API token, delete it
    if ($user && $user->currentAccessToken()){
        $user->currentAccessToken()->delete();
    }

    auth()->logout();
    $request->session()->invalidate();
    $request->session()->regenerateToken();
    return redirect('/login');
})->name('logout');
```

13. Mitigation example for Broken authentication



#### 4.3. [Sensitive data exposure](#)

##### Vulnerable example:

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:aH8s9Hcmp5yQ1yHihR4izVda8nc0JvxYhX7tHoovPvU=
APP_DEBUG=true
APP_TIMEZONE=UTC
APP_URL=http://localhost
```

14. Vulnerable example for sensitive data exposure

##### Mitigation example:

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:aH8s9Hcmp5yQ1yHihR4izVda8nc0JvxYhX7tHoovPvU=
APP_DEBUG=false
APP_TIMEZONE=UTC
APP_URL=http://localhost
```

15. Mitigation example for sensitive data exposure

#### 4.4. [Cross-site scripting \(XSS\)](#)

Cross-site scripting (XSS) occurs when attackers inject client side scripts into webpages which compromises the user's browser and sensitive data. This can lead to sessions hijacking, manipulation of the DOM and forced navigations to malicious pages. Laravel prevents XSS attacks by auto escaping outputs through blade components (`{{ }}`) which converts HTML special characters in harmless entities. Additionally, developers should avoid using raw output (`{!! !!`) and validate user inputs to reduce the risk of XSS attacks.

### Vulnerable example:

```
<p class="text-[#583E25]">Customer: {!! $order->user->name !!}</p>
<li>{!! $item->product->name !!} (x{{ $item->pivot->quantity ?? 1 }})</li>
```

16. Vulnerable example for XSS

### Mitigation example:

```
</ul>
<p class="text-[#F72C01]">Total: LKR {{ $order->total_amount ?? '0.00' }}</p>
<p class="mt-2 text-[#583E25]">Status: {{ ucfirst($order->status) }}</p>
```

17. Mitigation example for XSS

## 4.5. [Cross-site request forgery \(CSRF\)](#)

Cross-site request forgery (CSRF) occurs when attackers trick users into performing unintended actions on web application without their knowledge by submitting forged requests. This typically happen when an authenticated user's browser is induced to send malicious requests such as submitting hidden forms to modify data, transfer funds, or perform any other sensitive functions. CSRF can be mitigated in Laravel by utilizing the framework's CSRF middleware and including @csrf in Blade directive in HTML forms which automatically generates and validates tokens for all state changing request.

### Vulnerable example:

```
<form action="{{ route('admin.products.update', $product->id) }}" method="POST" class="space-y-4">
  <input type="text" name="name" value="{{ old('name', $product->name) }}"
    placeholder="Product Name" class="text-[#583E25] w-full border border-black p-2 rounded-xl" />

  <input type="text" name="price" value="{{ old('price', $product->price) }}"
    placeholder="Product Price" class="text-[#583E25] w-full border border-black p-2 rounded-x" />

  <input type="text" name="description" value="{{ old('description', $product->description) }}"
    placeholder="Product Description" class="text-[#583E25] w-full border border-black p-2 rou" />
</form>
```

18. Vulnerable example for CSRF

## Mitigation example:

```
4      <form action="{{ route('admin.products.update', $product->id) }}" method="POST" class="space-y-4">
5          @csrf
6          @method('PUT')
7
8          <input type="text" name="name" value="{{ old('name', $product->name) }}"
9              placeholder="Product Name" class="text-[#583E25] w-full border border-black p-2 rounded-xl
10
11          <input type="text" name="price" value="{{ old('price', $product->price) }}"
12              placeholder="Product Price" class="text-[#583E25] w-full border border-black p-2 rounded-x
13
14          <input type="text" name="description" value="{{ old('description', $product->description) }}"
15              placeholder="Product Description" class="text-[#583E25] w-full border border-black p-2 rou
16      </form>
17
```

19. Mitigation example for CSRF

### 4.6. [Role-based access control \(RBAC\)](#)

Through a combination of gates, policies and middleware, Role-based access control is implemented in Laravel to enforce fine-grained authorization rules. In this platform, users are assigned roles that define their level of access in the system. For example, a customer is limited to browse products, add items to the cart, and placing an order, while the admin can manage users, manage products and view order history and alter order status. These assigned role checks are enforced through Laravel's gates for routes or actions, while policies define the complex authorization logic tied specific models. Middleware such as auth enforces authentication and consistency verifies access to routes. This prevents unauthorized users from accessing elevated privileges and accessing sensitive information.

### 4.7. [Secure error handling and logging](#)

Secure error handling is a vital part of the system since it prevents sensitive information being exposed to end users. To ensure that error logs are not displayed, in production debug mode should be disabled (APP\_DEBUG=false). Instead Laravel logs errors through a Monolog which stores log files outside the publicly accessible directories, this enables developers to safely debug errors without exposing any sensitive data to the attackers or end users.

## 5. Deployment security considerations

In Laravel security deployment plays a crucial role in protecting applications once they go live. Sensitive credentials which are stored in the .env files such as API keys and database credentials, must never be exposed to the public and should be readable by the application itself. To minimize the attack surface at the server level unnecessary ports and services are disabled, while strict file permissions are enforced to restrict access to required users. HTTPS is enabled across the application to safeguard the data in transit as well as to ensure all client-server communications are encrypted. Furthermore, outdated or vulnerable packages are updated and maintained to patch known security patches and risks.

```
LOG_CHANNEL=stack
LOG_STACK=single
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=debug

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=BiteOasis2_db
DB_USERNAME=root
DB_PASSWORD=

MONGO_DB_CONNECTION=mongodb
MONGO_DB_HOST=devcluster.rbjsawi.mongodb.net
MONGO_DB_PORT=27017
MONGO_DB_DATABASE=BiteOasis
MONGO_DB_USERNAME=admin
MONGO_DB_PASSWORD=y1F50qmef7LwKaoy
MONGO_DB_DSN=mongodb+srv://admin:y1F50qmef7LwKaoy@devcluster.rbjsawi.mongodb.net/BiteOasis?retryWrites=true
```

20. env Database configuration

## 6. Future enhancements

Although the system effectively addresses and mitigates the OWASP top 10 risks, to further future improvements could enhance the security against advanced threats. Introducing ThrottleRequests middleware would mitigate the brute-force login attempts and API abuse by restricting excessive requests. By enabling email verification and multi-factor authentication through Jetstream authentication it would ensure that the accounts remains safe even if the passwords are compromised. Enforcing HTTP Strict Transport Security (HSTS) headers at the session and the transport layer along with applying secure cookie attributes such as HttpOnly, SameSite and Secure session security would strengthen defenses against SSL stripping and session hijacking. To provide real-time visibility into malicious activities such as unauthorized access attempts, logging and monitoring can be implemented through Laravel Telescope, Monolog or third party tools such as Sentry. Collectively, these measures significantly improve the resilience and strengthen its security standards.

## 7. GitHub link

**GitHub repo:** <https://github.com/thiviru-opallage/BiteOasis>

## 8. References

- OWASP Foundation (2021) *OWASP Top 10: The Ten Most Critical Web Application Security Risks*. Available at: <https://owasp.org/Top10> (Accessed: 20 September 2025)
- PHP (2024) *PHP Manual*. Available at: <https://www.php.net/manual/en/> (Accessed: 18 September )
- Laravel (2024) *Laravel 11 Documentation*. Available at: <https://laravel.com/docs> (Accessed: 18 October 2025).