Mitigation Strategy Report



Dimitar Rebranin

College of Engineering and Technology

Utah Valley University

09/26/2024

IT-6780-701 – Secure Coding

Prof. Kodey Crandall

# Table of Contents

Table of Contents

[Table of Contents 3](#_Toc178235869)

[1 Introduction 5](#_Toc178235870)

[2 Threat Summary 5](#_Toc178235871)

[3 Mitigation Strategy for Each Threat 6](#_Toc178235872)

[3.1 Spoofing – Stolen Credentials 6](#_Toc178235873)

[3.1.1 Mitigation Strategy 6](#_Toc178235874)

[3.1.2 Justification 6](#_Toc178235875)

[3.2 Tampering – XSS 8](#_Toc178235876)

[3.2.1 Mitigation Strategy 8](#_Toc178235877)

[3.2.2 Justification 8](#_Toc178235878)

[3.3 Repudiation – Unauthorized access 9](#_Toc178235879)

[3.3.1 Mitigation Strategy 9](#_Toc178235880)

[3.3.2 Justification 9](#_Toc178235881)

[3.4 Information Disclosure – User Data Exposure 11](#_Toc178235882)

[3.4.1 Mitigation Strategy 11](#_Toc178235883)

[3.4.2 Justification 11](#_Toc178235884)

[3.5 Denial of Service – Traffic Overload 13](#_Toc178235885)

[3.5.1 Mitigation Strategy 13](#_Toc178235886)

[3.5.2 Justification 13](#_Toc178235887)

[3.6 Elevation of Privilege - Admin account compromise 15](#_Toc178235888)

[3.6.1 Mitigation Strategy 15](#_Toc178235889)

[3.6.2 Justification 15](#_Toc178235890)

[4 Implementation Plan 17](#_Toc178235891)

[4.1 Spoofing 17](#_Toc178235892)

[3.2 Tampering - XSS 20](#_Toc178235893)

[3.3 Repudiation 20](#_Toc178235894)

[3.4 Information Disclosure 20](#_Toc178235895)

[3.5 Denial of Service 20](#_Toc178235896)

[3.6 Elevation of Privilege 20](#_Toc178235897)

[5 Conclusions 20](#_Toc178235898)

[6 References 20](#_Toc178235899)

# 1 Introduction

The Recipe Hub is a website for cooking enthusiasts to share their love for food and delicious meals, and their recipes with the whole world. It is important to make sure that the website is secure, and people can share safely their recipes, but also have confidence that the recipes are legitimate and safe to use. The idea of this project is to identify the proper mitigation strategy for each threat, and the implementation plan on how to integrate them into our system.

Why are mitigation strategies so important? Mitigation strategies help companies and people to reduce the risk of cyber security threats. Implementing the right strategy for each threat is vital as this will protect our assets, reputation and customer trust. This will also help with resolving vulnerabilities before they have been exploited by the bad actors.

# 2 Threat Summary

In previous weeks, we used the STRIDE model to identify our threats to the Recipe Hub and used the Microsoft threat modeling tool to create level 0 and 1 data flow diagrams, to show the weaknesses and vulnerabilities of the system. STRIDE stands for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege.

1. **Spoofing**: Spoofing attack happens when an attacker attempts to use stolen credentials to impersonate a real user. The consequences could be attacker might publish fake recipes or malicious content, misleading users and potentially harming your reputation.
2. **Tampering**: Cross Site Scripting (XSS) is an attack when the attacker injects a malicious code into a legitimate website. This can be done either with stolen credentials or just vulnerable website components. The attack then can alter existing content, in our case recipes, add harmful ingredients which can lead potentially to death. If the attack is able to compromise an admin user, then they can tamper with personal information, payment details and such.
3. **Repudiation**: Unauthorized access. This type of threat is when the integrity and trust on the website are compromised. This could lead to submitting fake reviews and comments, or even uploading or misusing copyrighted material, claiming it is theirs.
4. **Information Disclosure**: User data exposure, or sensitive information, could be exposed through vulnerabilities or improper access controls.
5. **Denial of Service**: Web server and application uptime. The type of threat is called traffic overload, which in simple words is the attacker generating a flood of requests, overwhelming the server and making it inaccessible to legitimate users.
6. **Elevation of Privilege**: Admin account compromise – if an attacker gets a hold of admin credentials, they could make changes to the settings of the website, access sensitive information and data, and even manipulate the site’s content.

# 3 Mitigation Strategy for Each Threat

## 3.1 Spoofing – Stolen Credentials

### 3.1.1 Mitigation Strategy

The mitigation strategy that I consider the most effective when it comes to protecting from stolen credentials is Multi-Factor Authentication (MFA). What is MFA? Security process that requires more than one step to log in to your account. The user will still need to put their username and password, but then they will be required to enter a second form of authentication.

### 3.1.2 Justification

* **Effectiveness**: Multi-Factor Authentication (MFA) significantly reduces the risk of spoofing attacks, such as credential spoofing or identity spoofing. By requiring an additional verification step beyond just a username and password, MFA makes it extremely difficult for an attacker to gain unauthorized access, even if they have stolen or guessed user credentials.
* **Feasibility:** Implementing MFA is relatively straightforward, especially with many platforms offering built-in support for it. It can be integrated with existing systems through software solutions like Google Authenticator, Microsoft Authenticator, or hardware-based tokens such as YubiKey. Many Identity and Access Management (IAM) systems also support MFA, making integration with existing authentication flows manageable.
* **Impact:**
  + **Performance:** The impact on system performance is minimal, as the additional verification step only occurs during the login process.
  + **Security:** MFA drastically enhances security by ensuring that even if one factor (like a password) is compromised, the attacker still cannot access the account without the second factor. It reduces the effectiveness of spoofing attacks by adding a layer of defense.
* **Cost:**
  + **Tools and Infrastructure:** While there may be some costs associated with acquiring MFA tools or tokens, many solutions are free or low-cost. For example, software-based authenticators can be used without additional expenses.
  + **Operational Costs:** Training and user support might incur some costs, but they are generally outweighed by the security benefits.
  + **Scalability:** MFA solutions are scalable and can be applied to both small and large organizations with minimal incremental costs.

## 3.2 Tampering – XSS

### 3.2.1 Mitigation Strategy

The mitigation strategy that I consider the most effective when it comes to XSS is Output Encoding (OE) and Input Validation (IV).

### 3.2.2 Justification

* **Effectiveness:**
  + **Output Encoding** is highly effective in preventing XSS attacks. It ensures that any user-supplied data displayed in the browser is safely encoded, preventing the execution of malicious scripts. By encoding characters like <, >, &, and " into their HTML entities, the browser treats them as text rather than executable code.
  + **Input Validation** complements output encoding by ensuring that input fields accept only expected data formats. While it is not a replacement for encoding, it can reduce the likelihood of malicious payloads being processed.
* **Feasibility:**
  + **Output Encoding**: It is relatively straightforward to implement using built-in functions in most modern web frameworks and libraries.
  + **Input Validation**: This can be implemented using regular expressions or built-in validation libraries in frameworks like Django, ASP.NET, or Angular. It involves adding checks to ensure inputs conform to expected formats (e.g., no script tags in text fields).
* **Impact:**
  + **Performance Impact**: Minimal, as encoding and validation functions are lightweight and do not significantly affect the system's performance.
  + **Security Impact**: High. Combined, these strategies prevent XSS by ensuring that user inputs are safely handled, and outputs are rendered without executing potentially harmful code. This protects both users and the application from XSS attacks.
* **Cost:**
  + **Development Cost**: Low to moderate, depending on the current state of the codebase. Implementing these strategies typically requires developer time to identify vulnerable areas and apply the appropriate functions.
  + **Tools and Infrastructure**: No additional tools or infrastructure are necessary. Most modern development frameworks have built-in support for encoding and validation.
  + **Maintenance Cost**: Low. Once implemented, the strategies require minimal ongoing maintenance, primarily focused on ensuring that new code adheres to the same standards.

## 3.3 Repudiation – Unauthorized access

### 3.3.1 Mitigation Strategy

The mitigation strategy that I consider the most effective when it comes to unauthorized access is Role-Based Access Control (RBAC).

### 3.3.2 Justification

* **Effectiveness:**
  + RBAC effectively minimizes the risk of unauthorized access by defining and enforcing permissions based on user roles. Each user is assigned one or more roles, and each role is associated with a specific set of permissions. This limits access to sensitive data and system functions based on the user's job function or responsibility, ensuring that only authorized users can access certain resources.
  + By providing granular control over who can access what, RBAC significantly reduces the attack surface, preventing both internal and external threats from exploiting unauthorized access.
* **Feasibility:**
  + Implementing RBAC is relatively straightforward, especially in systems that already have a user management framework. Most modern software platforms, databases, and operating systems support RBAC natively or through third-party plugins.
  + For applications without built-in support, RBAC can be implemented programmatically by creating user roles, defining permissions, and updating the application logic to enforce these rules.
* **Impact:**
  + **Performance Impact:** Minimal, as access checks based on roles are usually lightweight operations that do not significantly impact system performance.
  + **Security Impact:** High, as it ensures that users can only access data and perform actions within the boundaries of their roles. This prevents privilege escalation and unauthorized access to sensitive resources.
* **Cost:**
  + **Development and Implementation Costs:** Implementing RBAC may require initial development effort to define roles, permissions, and user assignments. If the system already supports RBAC, the cost is mainly administrative.
  + **Maintenance Costs:** Low to moderate, as ongoing maintenance involves updating roles and permissions as user roles evolve or new roles are created.
  + **Tools and Infrastructure:** No additional tools are typically required if the system supports RBAC. For systems without native support, custom implementation might incur additional costs.

## 3.4 Information Disclosure – User Data Exposure

### 3.4.1 Mitigation Strategy

The mitigation strategy that I consider the most effective when it comes to user data exposure is data encryption.

### 3.4.2 Justification

* **Effectiveness:**
  + **Data Encryption** effectively reduces the risk of user data exposure by converting sensitive information into a coded format that can only be deciphered with the appropriate decryption key. Encrypting data both at rest (e.g., in databases, file storage) and in transit (e.g., during network transmission) ensures that even if data is intercepted or accessed without authorization, it remains unintelligible to unauthorized parties.
  + Encryption addresses various threats such as data breaches, man-in-the-middle attacks, and unauthorized access, making it a comprehensive solution for protecting user data.
* **Feasibility:**
  + **Encryption in Transit**: Implementing encryption for data in transit is straightforward with technologies like HTTPS/TLS for web applications and VPNs for internal network traffic. Most modern frameworks and cloud services support TLS by default.
  + **Encryption at Rest**: For databases and file systems, enabling encryption is usually a matter of configuration. Many modern databases (e.g., MySQL, PostgreSQL, and MongoDB) and cloud storage services provide built-in encryption features.
  + **Key Management**: Proper key management is essential but can be complex. This is simplified by using managed services such as AWS Key Management Service (KMS) or Azure Key Vault.
* **Impact:**
  + **Performance Impact**: Encryption can introduce some performance overhead, especially during data encryption/decryption processes. However, with modern hardware and optimized algorithms, the impact is generally minimal and manageable.
  + **Security Impact**: High. Encryption ensures that even if data is leaked or intercepted, it cannot be accessed or used without the correct decryption keys, thereby providing a robust defense against data exposure.
* **Cost:**
  + **Tools and Infrastructure Costs**: Many encryption solutions are included with existing software or services. For example, database encryption and HTTPS are typically free or low-cost options. However, there may be additional costs for advanced key management systems or hardware security modules (HSMs).
  + **Operational Costs**: Additional costs may be incurred for managing encryption keys and certificates, especially in large-scale deployments. Regular audits and monitoring are also necessary to ensure encryption is properly implemented and maintained.
  + **Implementation Costs**: For systems that do not already use encryption, there may be a moderate initial cost to implement and configure it, including potential modifications to the application code and database schema.

## 3.5 Denial of Service – Traffic Overload

### 3.5.1 Mitigation Strategy

The mitigation strategy that I consider the most effective when it comes to traffic overload is auto-scaling with load balancing.

### 3.5.2 Justification

* **Effectiveness:**
  + **Auto-Scaling** dynamically adjusts the number of active servers based on traffic demand, ensuring that the system can handle increased traffic without degrading performance. It automatically provisions additional resources when traffic surges and scales down when traffic decreases, maintaining optimal performance.
  + **Load Balancing** distributes incoming traffic across multiple servers to prevent any single server from being overwhelmed. This ensures that the system remains responsive and available even during periods of high demand.
* **Feasibility:**
  + **Auto-Scaling and Load Balancing** are relatively easy to implement, especially in cloud environments such as AWS, Azure, or Google Cloud, which provide built-in support for these features. Many cloud platforms offer auto-scaling groups and load balancers that can be configured with minimal effort.
  + For on-premise or hybrid environments, software-based load balancers (e.g., HAProxy, Nginx) and infrastructure automation tools (e.g., Kubernetes, Terraform) can be used to set up auto-scaling and load balancing.
* **Impact:**
  + **Performance Impact:** Both strategies ensure that the system remains performant under high load. Auto-scaling can introduce some latency when scaling up or down, but this is generally minimal and can be mitigated by pre-warming instances.
  + **Security Impact:** Load balancers can also act as a first line of defense against DDoS attacks by distributing malicious traffic and absorbing bursts, reducing the risk of system overload. Auto-scaling ensures that resources are available, preventing denial of service due to legitimate traffic surges.
* **Cost:**
  + **Infrastructure Costs:** Auto-scaling incurs costs based on the additional resources used during traffic surges. However, costs are proportional to usage, and resources scale down when not needed, optimizing overall expenses.
  + **Tools and Infrastructure:** Most cloud providers offer auto-scaling and load balancing as part of their standard services, so no additional tools are needed.
  + **Implementation Costs:** Initial configuration and setup can be moderate, depending on the complexity of the environment. However, ongoing maintenance is minimal once implemented.

## 3.6 Elevation of Privilege - Admin account compromise

### 3.6.1 Mitigation Strategy

The mitigation strategy that I consider the most effective when it comes to elevation of privilege is Least Privilege Principle with RBAC.

### 3.6.2 Justification

* **Effectiveness:**
  + **Least Privilege Principle:** Ensures that users, including administrators, are granted only the minimum permissions necessary to perform their tasks. This significantly reduces the attack surface, as even if an attacker compromises an account, they are limited in what they can do. It prevents users from having excessive privileges that could be exploited to elevate their access rights.
  + **Role-Based Access Control (RBAC):** Helps enforce the least privilege principle by defining roles with specific permissions and assigning users to these roles based on their responsibilities. This structured approach minimizes the risk of privilege escalation by preventing users from acquiring unnecessary or unauthorized access rights.
* **Feasibility:**
  + **RBAC:** Is relatively straightforward to implement in most modern systems, including operating systems, cloud platforms, and enterprise applications. Many systems support RBAC natively or through integration with identity and access management (IAM) solutions. It involves defining roles and permissions, which can be configured based on the organization’s existing structure and policies.
  + **Least Privilege Implementation:** Requires reviewing and adjusting current access rights to ensure that users only have the permissions they need. This process can be time-consuming, especially in environments with poorly documented access controls, but tools like IAM solutions can streamline this process.
* **Impact:**
  + **Performance Impact:** Minimal. RBAC and least privilege configurations are access control measures that do not typically affect system performance. Access checks are lightweight and integrated into the authentication and authorization processes.
  + **Security Impact:** High. By minimizing the permissions available to users and limiting access to critical systems and data, the strategy significantly reduces the risk of privilege escalation. It also helps ensure that if an account is compromised, the damage is contained.
* **Cost:**
  + **Tools and Infrastructure Costs:** Most modern platforms and systems include support for RBAC and least privilege management at no additional cost. For more complex environments, additional IAM tools or solutions like Microsoft Azure AD, AWS IAM, or third-party PAM tools may be required.
  + **Operational Costs:** The primary cost is associated with the initial setup and ongoing management of access rights, including periodic reviews and updates as roles and responsibilities change.
  + **Implementation Costs:** Moderate. Establishing RBAC and least privilege controls can be time-intensive, especially if existing access policies are complex or poorly defined. However, once implemented, the ongoing management is relatively straightforward.

# 4 Implementation Plan

## 4.1 Spoofing

### 4.1.1 Steps to Implement MFA

**Step 1: Select an MFA Provider**

* Choose a third-party service like **Auth0**, **AWS Cognito**, or **Firebase Authentication** that supports MFA.
* Ensure the provider supports OTP via authenticator apps (e.g., Google Authenticator) and SMS.

**Step 2: Integrate MFA into the Website**

* **Client-side Integration**:
  + Use a JavaScript library provided by the MFA provider to handle user registration and login flows.
  + Modify the registration and login pages to include additional steps for MFA, such as entering an OTP.
* **Server-side Integration**:
  + Update authentication endpoints to handle MFA tokens.
  + Store user MFA preferences and statuses in the database securely.

**Step 3: Update User Registration Process**

* Collect and verify the user's phone number or email for MFA during registration.
* Allow users to choose their preferred MFA method (e.g., SMS, authenticator app).

**Step 4: Update Login Process**

* After the user enters their username and password, prompt them to enter the OTP.
* Verify the OTP with the MFA provider.
* Only grant access if both the password and OTP are valid.

**Step 5: Secure the MFA Setup Process**

* Require users to re-authenticate before setting up or modifying their MFA settings to prevent unauthorized changes.
* Send notifications to the user whenever their MFA settings are updated.

**Step 6: Recovery Options**

* Provide options for users to recover access if they lose their MFA device, such as backup codes or verifying identity through email.

### 4.1.2 Tools and Technologies

* **Auth0** or **AWS Cognito**: For MFA and overall user authentication.
* **Google Authenticator**: For OTP generation.
* **Node.js** and **Express**: For server-side implementation of MFA verification endpoints.
* **React.js**: For front-end modifications to the login and registration processes.

## 4.2 Tampering

### 4.2.1 Steps to Implement Input Validation

**Step 1: Define Validation Rules**

* **Identify Inputs**: List all user inputs, including forms, query parameters, cookies, and headers.
* **Set Rules**: Define rules for each input, such as data type (e.g., string, integer), format (e.g., email, phone number), length limits, and allowed characters.

**Step 2: Implement Client-Side Validation**

* Use a validation library like **Yup** or **Formik** with React.js.
* Validate input fields in real-time to provide immediate feedback to the user.

**Step 3: Implement Server-Side Validation**

* Always validate data on the server to prevent bypassing client-side validation.
* Use a validation library like **Joi** in Node.js/Express.
* Create schemas for each API endpoint to enforce validation rules.

**Step 4: Use Input Sanitization**

* Use libraries like **validator.js** for common input sanitization tasks such as stripping HTML tags or normalizing email addresses.
* Sanitize user inputs before further processing to remove potentially malicious content.

**Step 5: Implement Rate Limiting**

* Implement rate limiting on form submissions and API requests using tools like **express-rate-limit** to prevent automated attacks and spamming.

**Step 6: Test and Monitor**

* Use automated tools like **OWASP ZAP** to test for input validation vulnerabilities.
* Implement logging for validation errors to monitor and identify unusual patterns of misuse or attacks.

### 4.2.2 Tools and Technologies for Input Validation

* **React.js + Yup/Formik**: For client-side validation.
* **Node.js + Joi**: For server-side validation.
* **validator.js**: For input sanitization.
* **express-rate-limit**: For rate limiting.

### 4.2.3 Steps to Implement Output Encoding

**Step 1: Identify All User-Generated Content**

* Identify areas of the website where user-generated content is displayed, such as recipe descriptions, comments, or user profiles.

**Step 2: Use Encoding Libraries**

* Use a library like **DOMPurify** in React.js to encode and sanitize any HTML content before rendering it on the client side.
* For server-side rendering, use libraries like **he** or **express-validator** to encode data before sending it to the client.

**Step 3: Encode All Output Based on Context**

* **HTML Encoding**: Encode characters like <, >, and & to their HTML entities (e.g., &lt;, &gt;, &amp;).
* **JavaScript Encoding**: Use backslash encoding to prevent injection in JavaScript contexts (e.g., \x3C for <).
* **URL Encoding**: Encode user inputs in URLs using functions like encodeURIComponent.

**Step 4: Implement Content Security Policy (CSP)**

* Set up a CSP header to restrict the sources of scripts, styles, and other resources. This helps prevent the execution of untrusted code.

**Step 5: Sanitize Inputs Before Storing in the Database**

* Use server-side sanitization libraries to strip any malicious scripts or HTML tags from user inputs before saving them to the database.

**Step 6: Regular Security Testing and Code Reviews**

* Perform regular security testing using tools like **Burp Suite** and **OWASP ZAP** to identify any missed encoding or sanitization issues.
* Conduct code reviews to ensure encoding is applied consistently and correctly across all parts of the application.

### 4.2.4 Tools and Technologies for Output Encoding

* **DOMPurify**: For client-side and server-side HTML sanitization.
* **he**: For HTML entity encoding.
* **express-validator**: For server-side input validation and encoding.
* **CSP Headers**: For enforcing Content Security Policy.
* **Burp Suite** and **OWASP ZAP**: For security testing.

## 4.3 Repudiation

### 4.3.1 Steps to Implement RBAC

**Step 1: Define roles and permissions**

* **Identify Roles**
  + **Admin**: Full control over the website, including managing users, roles, and all content.
  + **Editor**: Can create, edit, and delete recipes, but cannot manage users or roles.
  + **User**: Can view and comment on recipes, create favorite lists, and access personalized content.
  + **Guest**: Can only view public recipes and content.
* **Map Permissions to Roles**
  + Create matrix mapping actions (e.g., create, read, update, delete) to roles.
  + Example:

| **Action** | **Admin** | **Editor** | **User** | **Guest** |
| --- | --- | --- | --- | --- |
| View Recipes | Yes | Yes | Yes | Yes |
| Create Recipes | Yes | Yes | No | No |
| Edit Recipes | Yes | Yes | No | No |
| Delete Recipes | Yes | Yes | No | No |
| Manage Users | Yes | No | No | No |
| Comment | Yes | Yes | Yes | No |

**Step 2: Design the database schema - create tables for roles and permissions**

* **Roles Table**: Stores role information.
* **Permissions Table**: Lists all possible actions.
* **Role-Permissions Table**: Maps roles to permissions.
* **Users Table**: Stores user information and their assigned role.

**Step 3: Implement backend logic for RBAC**

* **Assign and Manage Roles in the Backend**
  + Use an administrative interface to assign roles to users.
  + Update user roles in the database as needed.
* **Check Permissions in Middleware -** Implement middleware to check permissions based on user roles before accessing a route.

**Step 4: Frontend Integration**

* **Show/Hide UI Elements Based on Roles**
  + Use state management to manage user roles and permissions in the front end.
  + Conditionally render UI elements (e.g., buttons, links) based on the user's role.
* **Restrict Access to Routes -** Use client-side routing libraries like **React Router** to restrict access based on roles.

**Step 5: Implement access control for APIs**

* Add RBAC middleware to all API routes to enforce permissions server-side.

**Step 6: Logging and Auditing**

* Implement logging for all access control decisions.
* Log failed access attempts to detect and respond to potential security threats.

**Step 7: Testing and Validation**

* Perform unit and integration testing to ensure that the RBAC implementation works as expected.
* Test edge cases, such as unauthorized users attempting to access restricted routes.

**Step 8: Documentation and Training**

* Document the roles, permissions, and access control rules.
* Train administrators and developers on how to manage roles and permissions.

### 4.3.2 Tools and Technologies

* **Node.js** with **Express**: For backend API development.
* **PostgreSQL** or **MySQL**: For relational database management.
* **Joi**: For request validation and schema enforcement.
* **React.js**: For front-end development and conditional rendering.
* **Redux**: For managing user state and roles in the frontend.
* **Passport.js** or **JWT**: For user authentication and role management.
* **Winston** or **Morgan**: For logging and auditing.

## 4.4 Information Disclosure

### 4.4.1 Steps to Implement Data at Rest Encryption

**Step 1: Encrypt Sensitive Data in the Database**

* **Password Encryption**
  + Use a strong hashing algorithm, such as **bcrypt** or **Argon2**, to securely store passwords.
* **Encrypting Other Sensitive Data**
  + Use a symmetric encryption algorithm like **AES-256** to encrypt sensitive fields, such as email addresses or payment details.
  + Store the encryption key securely, using a service like **AWS KMS** or **Azure Key Vault**.
* **Database-Level Encryption**
  + For sensitive columns, use database-level encryption features, such as **Transparent Data Encryption (TDE)** in SQL Server or **pgcrypto** in PostgreSQL.

**Step 2: Encrypt Files and Backups**

* Encrypt file storage using tools like **VeraCrypt** or built-in OS encryption features.
* Automate backup encryption with tools like **AWS S3 Server-Side Encryption** or **Azure Blob Storage Encryption**.

**Step 3: Secure Key Management**

* Store encryption keys in a secure key management service, such as **AWS KMS**, **Google Cloud KMS**, or **Azure Key Vault**.
* Rotate keys regularly and log key usage for auditing purposes.

### 4.4.2 Steps to Implement Data in Transit Encryption

**Step 1: Enforce HTTPS**

* Obtain and install an SSL/TLS certificate from a trusted certificate authority (e.g., **Let's Encrypt**).
* Redirect all HTTP traffic to HTTPS and configure HSTS (HTTP Strict Transport Security) headers to prevent protocol downgrade attacks.

**Step 2: Use Secure Communication for APIs**

* Use HTTPS for all internal and external API calls.
* Validate server certificates to prevent man-in-the-middle attacks.

**Step 3: Secure WebSocket Connections**

* Use secure WebSocket connections (wss://) for real-time features.
* Verify origin and authentication tokens for all WebSocket connections.

### 4.4.3 Tools and Technologies

* **Node.js + bcrypt/crypto**: For password hashing and data encryption.
* **PostgreSQL pgcrypto** or **MySQL TDE**: For database-level encryption.
* **AWS KMS**, **Azure Key Vault**, **Google Cloud KMS**: For secure key management.
* **Let's Encrypt** or **DigiCert**: For SSL/TLS certificates.
* **HashiCorp Vault**, **Doppler**: For secure configuration management.
* **Splunk**, **ELK Stack**, **AWS CloudTrail**: For monitoring and auditing.

## 4.5 Denial of Service

### 4.5.1 Steps to Implement Auto-scaling and Load Balancer

**Step 1:** **Choose a Cloud Provider and Set Up Load Balancer**

* Select a cloud provider like AWS, Google Cloud, or Azure based on existing infrastructure and preferences.
* Deploy a load balancer service (e.g., AWS Elastic Load Balancer, Google Cloud Load Balancer, Azure Load Balancer).
* Configure the load balancer to distribute traffic based on health checks, round-robin, or least connection algorithms.

**Step 2: Implement Auto-Scaling Policies**

* **Define Auto-Scaling Groups**:
  + Create auto-scaling groups for different tiers of the application, such as web servers and API servers.
* **Set Scaling Policies**:
  + Define policies based on metrics like CPU utilization, memory usage, or custom application metrics.
  + Set minimum and maximum instance limits to control scaling behavior.
* **Configure Cooldown Periods**:
  + Set cooldown periods to avoid unnecessary scaling actions when traffic spikes momentarily.

**Step 3: Database Scaling and Load Balancing**

* **Set Up Read Replicas**:
  + Create read replicas of the primary database to handle read-heavy traffic.
* **Database Load Balancer**:
  + Use a database proxy or load balancer like AWS RDS Proxy or ProxySQL to distribute read queries across replicas.
* **Vertical Scaling for Writes**:
  + Increase the size of the primary database instance or use a sharding strategy for high write throughput.

**Step 4: Implement Caching Mechanism**

* **Integrate Caching Layer**:
  + Use a caching service like Redis or Memcached to store frequently accessed data such as popular recipes.
* **Set Up Cache Invalidation**:
  + Implement cache invalidation strategies to ensure that data remains fresh.
* **Configure Auto-Scaling for Cache**:
  + Set up auto-scaling for caching clusters to handle high request loads dynamically.

**Step 5: Testing and Optimization**

* **Load Testing**:
  + Use load testing tools like Apache JMeter, Gatling, or Locust to simulate high traffic and monitor scaling behavior.
* **Chaos Engineering**:
  + Introduce controlled failures (e.g., killing instances) to test the resilience of the auto-scaling and load balancing setup.
* **Optimization**:
  + Analyze test results and fine-tune scaling policies, cooldown periods, and load balancer configurations.

**Step 6: Deployment and Continuous Monitoring**

* **Deployment**:
  + Deploy the auto-scaling and load balancing setup to the staging environment first, then to production after testing.
* **Monitoring and Alerts**:
  + Use monitoring tools to track performance metrics, instance counts, and user experience.
  + Set up alerts for anomalies like unexpected scaling, high latency, or high error rates.
* **Logging and Auditing**:
  + Implement logging and auditing to track scaling events and application performance.

### 4.5.2 Tools and Technologies

* **Cloud Platforms**: AWS, Azure and GCP
* **Automation**: Terraform or CloudFormation for Infrastructure as Code (IaC) to automate configuration
* **Database Tools**: AWS RDS/Aurora, Google Cloud SQL, Azure SQL Database.
* **Load Testing**: Apache JMeter, Gatling, Locust.
* **Monitoring**: Prometheus, Grafana, CloudWatch, or Stackdriver.
* **Logging**: ELK Stack, Fluentd, or Cloud-native logging services.

## 4.6 Elevation of Privilege

### 4.6.1 Steps to Implement Least Privilege Principle with RBAC

**Step 1: Implement RBAC**

* I have already drawn out the process in a step-by-step in [4.3.1](#_4.3.1_Steps_to) above.

**Step 2: Implement Granular Permissions for Resources**

* Ensure permissions are fine-grained enough to differentiate between similar actions.
* Define Resource-Specific Permissions - Create specific permissions like edit\_own\_recipe vs edit\_any\_recipe.
* Modify Authorization Logic - Update controllers to check not just the role but also the ownership of the resource when modifying or deleting it.

**Step 3: Implement Security Measures for Least Privilege**

* **Input Validation and Sanitization:**
  + Ensure all inputs are validated and sanitized to prevent attacks like SQL injection**.**
* **Token-Based Authentication:**
  + Use JWT (JSON Web Tokens) for secure and stateless authentication.
  + Include role information in the token payload to reduce server calls**.**
* **Encryption:**
  + Encrypt sensitive data in the database such as user roles and permissions**.**

### 4.6.2 Tools and Technologies

* **Security Libraries**: bcrypt.js for password hashing, JWT for token-based authentication.
* **Input Validation**: Libraries like Joi or Yup for Node.js.
* [4.3.2](#_4.3.2_Tools_and)

# 5 Conclusions

Following the STRIDE model for threat identification and after implementing the above mitigation strategies, at the end at minimum, we will have a website:

* With user-friendly MFA experience and enhanced overall site security, protected user accounts against unauthorized access
* With enhanced managed user permissions and roles, granulated access policies, and allowing access on a least privilege model.
* Encryption enabled and applied to data at rest and in transit, and using secure key management practices.
* Able to assess current input handling, implementing robust validation and encoding strategies, testing thoroughly, and maintaining vigilance, the website can significantly enhance its security posture against common web vulnerabilities
* Able to handle traffic variations efficiently while maintaining high availability and performance.

# 6 References

The primary source of the information I have found was google search and ChatGPT. I will be honest – I am using genAI tool for first time ever. I took me some time to figure out how to ask it properly and what information is useful. Most of the information for the implementation portion of this project I used ChatGPT for the step-by-step instructions.