



Secure API Design and Implementation for Shobee

Xin Jie
U2120663H

Tan Choon Wee
U2120106H

SC4013: Application Security
Project Report

College of Computing and Data Science

November 2024

Table of Contents

1. Project Overview	1
1.1 Objective	1
1.2 Scope	1
2. Background	1
2.1 Platform Vision	1
2.2 Security Importance	1
3. Key Security Features to Implement	2
3.1 Authentication	2
JSON Web Tokens (JWT) Implementation	2
Multi-Factor Authentication (MFA)	3
Time-based one-time passwords (TOTPs)	3
Google Authenticator	3
3.2 Authorization and Access Control	4
Principle of Least Privilege (POLP)	4
Role-based access control (RBAC)	4
Authorization and RBAC	4
3.3 Input Validation and Sanitization	5
Input Validation	5
Input Sanitization	5
3.4 Express-Validator in Node.js	5
Validation	5
Sanitization	5
Integration with Express.js	5
3.5 Rate Limiting and Throttling	6
Rate limiting	6
Throttling	6
ThrottlerGuard	6
3.6 Hypertext Transfer Protocol Secure (HTTPS) Enforcement	7
Prevents protocol downgrading	7
Prevents cookie hijacking	7
3.7 Database Security	8
AES-256 encryption	8
3.8 Cross-Site Request Forgery (CSRF) Protection	9
SameSite Attribute	9
3.9 API Versioning	10
	10
3.10 Bug Bounty Program	11

3.11	API Documentation	12
4.	Software Architecture	13
4.1	Dependency Management	13
	Automated Dependency Scanning	13
	Regular Dependency Updates	13
	Vulnerability Management	13
4.2	Secure Development Lifecycle (SDLC)	13
	Requirements Gathering	13
	Design Phase	13
	Development Phase	13
	Testing Phase.....	14
	Secure Deployment	14
4.3	Penetration testing.....	14
	Scope of Testing	14
5.	Future actions for continuous improvement.....	15
6.	Conclusion	15

1. Project Overview

1.1 Objective

Develop secure, robust APIs for Shobee, an e-commerce platform, to handle user authentication, product catalog management, and other core services securely.

1.2 Scope

The project aims to ensure a secure shopping experience through a well-designed API framework with critical security features.

2. Background

2.1 Platform Vision

Shobee aims to create a secure and seamless shopping platform where users feel confident about data safety.

2.2 Security Importance

Protecting user interactions and backend operations is crucial, requiring strict security protocols in all API services.

3. Key Security Features to Implement

3.1 Authentication

JSON Web Tokens (JWT) Implementation

JWT is a secure, open standard for transmitting information between applications and services. JWT are commonly used for authentication and authorization because they can contain information that verifies a user's identity and permissions. JWT are a good practice for maintaining secure sessions because they are digitally signed, stateless, and can be validated locally.

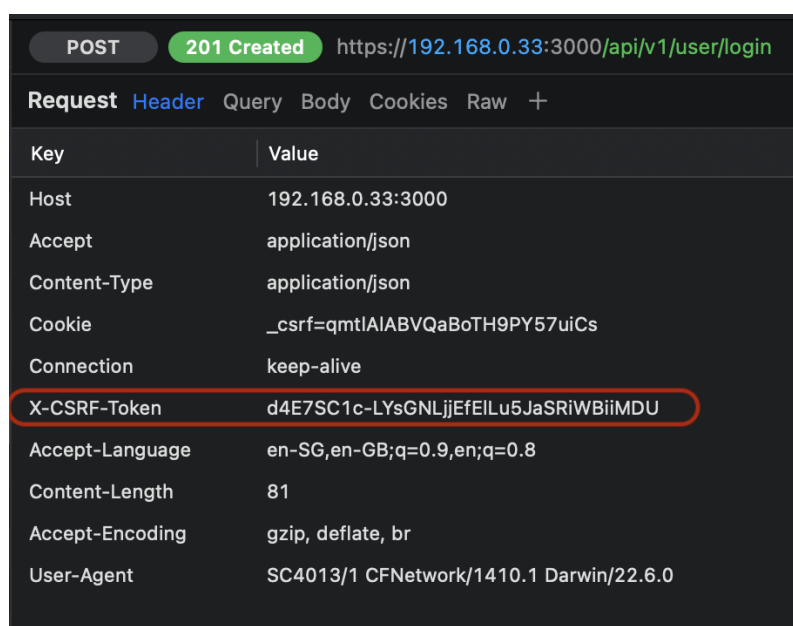
We will implement JSON Web Tokens (JWT) for secure user authentication, ensuring short-lived access tokens to minimize risk. Refresh tokens will be used alongside JWTs to allow session extension without re-authentication.

Here are some best practices for using JWTs:

- Set expiration times
 - Set short expiration times for the tokens, such as minutes or hours. This prevents attackers from using an old JWT to gain access to the application.
- Use HTTPS
 - HTTPS encrypts HTTP path and query parameters, which reduces the risk of someone intercepting the request.

```
const httpsOptions = {
  key: fs.readFileSync(process.env.KEY),
  cert: fs.readFileSync(process.env.CERT),
};
const app = await NestFactory.create(AppModule, {
  httpsOptions: httpsOptions,
});
```

- Send JWT in header
 - Tokens are automatically sent to the server and are not accessible via JavaScript.



POST 201 Created https://192.168.0.33:3000/api/v1/user/login

Key	Value
Host	192.168.0.33:3000
Accept	application/json
Content-Type	application/json
Cookie	_csrf=qmtlAlABVQaBoTH9PY57uiCs
Connection	keep-alive
X-CSRF-Token	d4E7SC1c-LYsGNLjjEfEILu5JaSRiWBiiMDU
Accept-Language	en-SG,en-GB;q=0.9,en;q=0.8
Content-Length	81
Accept-Encoding	gzip, deflate, br
User-Agent	SC4013/1 CFNetwork/1410.1 Darwin/22.6.0

Authentication - JWT

Multi-Factor Authentication (MFA)

MFA is a security method that requires users to provide more than just a password to access an account, system, or application. MFA is an important part of identity and access management (IAM) policies and can help prevent unauthorized access to accounts.

MFA will be added using Time-based One-Time Password (TOTP) to enhance security.

Time-based one-time passwords (TOTPs)

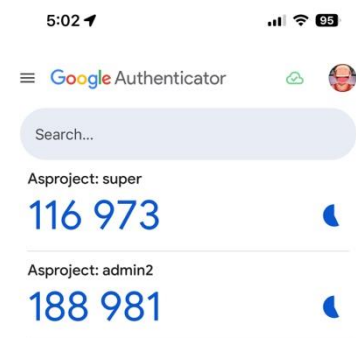
TOTPs are a type of MFA that can add an important layer of security because they are unique to each service and are not susceptible to dictionary attacks or brute-force attempts. We use Google authenticator for our project.

Google Authenticator

A mobile app that generates one-time passcodes (OTPs) for each site or service that signed in to. Can use the app to sign in to the Google account or other services that support using two-factor authentication (2FA). The app generates a new code every 30 seconds.

```
async login(username: string, password: string, twoFactorCode?:string) {
  let user: UserEntity = await this.userRepository.findOne({
    where: { username: username },
  });
  if (!user) {
    throw new UnauthorizedException()
  }

  if (bcrypt.compareSync(password, user.password) == false) {
    throw new UnauthorizedException()
  }
  if (user.twoFactorSecret && user.twoFactorSecret.length>0){
    if (!twoFactorCode) {
      throw new UnauthorizedException("two factor code is required");
    }
    const privateKeyHex = this.configService.get('PRIVATE_KEY');
    const secret = this.decrypt(user.twoFactorSecret, privateKeyHex);
    if (!authenticator.verify({ token: twoFactorCode, secret: secret })) {
      throw new UnauthorizedException("two factor code is invalid");
    }
  }
}
```



Authentication - 2FA

3.2 Authorization and Access Control

Enforce Role-Based Access Control (RBAC) limiting access based on user roles. Implement the principle of least privilege (POLP) to ensure minimal access rights. Implementing network segmentation to minimize the risk of unauthorized access and lateral movement within the system.

Principle of Least Privilege (POLP)

A cybersecurity best practice that limits users' access to only what is required to perform their job duties. It is a fundamental component of RBAC.

Example: Customers access only their orders.

Role-based access control (RBAC)

An access control method that assigns access rights based on roles, rather than individually defining rules for each user. RBAC is considered a better option for large organizations because it is more scalable and easier to manage than access control list (ACL).

Example: Customers can only access their own orders, while admins have broader access to manage the platform.

Authorization and RBAC

Authorizations are typically included in a rights profile, which is then included in a role, and the role is assigned to a user.

```
import {
  Injectable,
  ExecutionContext,
  CanActivate,
  Logger,
} from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { RoleAdmin, RoleSuperAdmin, UserEntity } from 'src/entities/user.entity';
import { Repository } from 'typeorm';

@Injectable()
export class AdminAuthGuard implements CanActivate {
  private readonly logger: Logger = new Logger(AdminAuthGuard.name);
  constructor(
    @InjectRepository(UserEntity)
    private userRepository: Repository<UserEntity> {
  }

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();

    const user = await this.userRepository.findOne({
      where: { id: request.user.id },
    });

    return user.role == RoleAdmin || user.role == RoleSuperAdmin;
  }
}
```

RBAC - AdminAuthGuard

3.3 Input Validation and Sanitization

Input validation and sanitization are crucial security measures to protect web applications from various vulnerabilities, including injection attacks.

Input Validation

This process involves checking if the user-provided input conforms to a specific format or set of rules. It ensures that the data is of the expected type and length, and it meets specific criteria.

Input Sanitization

This process involves removing or modifying harmful characters from the user input to prevent malicious code execution.



3.4 Express-Validator in Node.js

Express-validator is a popular library in the Node.js ecosystem that helps with both input validation and sanitization. Here are how it can help prevent injection attacks:

Validation

Express-validator provides a set of middleware functions to validate user input against specified criteria. For example, can check if an email is in the correct format or if a password meets certain strength requirements.

Sanitization

The library also includes functions to sanitize input, such as removing HTML tags or escaping special characters to prevent attacks like SQL Injection and Cross-Site Scripting (XSS).

Integration with Express.js

Express-validator integrates seamlessly with Express.js, allowing to define validation and sanitization rules directly in the route handlers.

```
"node_modules/class-validator": {
  "version": "0.14.1",
  "resolved": "https://registry.npmjs.org/class-validator/-/class-validator-0.14.1.tgz",
  "integrity": "sha512-2VEG9JICxIqTpoK1eMzZqaV+u/EiwEJkMGzTrZf6sU/fwsn0ITVgYJ8yojSy6CaXt09V0Cc6ZQZ8h8m4UBuLwQ==",
  "license": "MIT",
  "dependencies": {
    "@types/validator": "^13.11.8",
    "libphonenumber-js": "^1.10.53",
    "validator": "^13.9.0"
  }
},
```


3.5 Rate Limiting and Throttling

Rate limiting and throttling are both methods for limiting access to APIs (Application Programming Interface), but they have different purposes and approaches:

Rate limiting

It is a proactive strategy that protects APIs from abuse by malicious users. Rate limiting can be used to control data flow, for example by merging data streams into a single service. Implementing rate limiting to prevent abuse of critical endpoints like login and checkout, mitigating brute force attacks and ensuring fair usage.

Throttling

A more severe method that completely stops clients from making requests for a certain period. It is often used as a last resort to stop bad behaviour or attacks on the server.

ThrottlerGuard

ThrottlerGuard is a common tool in web development frameworks, particularly in environments like NestJS, to help prevent abuse and brute-force attacks by limiting the number of requests a user can make within a specified timeframe.

ThrottlerGuard enforces a maximum number of requests from a particular user or IP within a defined period.

For example, if you set a limit of 5 requests per minute, a user cannot exceed this threshold. This prevents attackers from bombarding the server with requests, slowing down the system or trying to brute-force login attempts.

```
UserModule,
ThrottlerModule.forRoot({
  throttlers: [
    {
      limit: 5,
      ttl: 1000,
    },
  ],
}),
],
controllers: [],
providers: [
  {
    provide: APP_GUARD,
    useClass: ThrottlerGuard,
  },
],
```

ThrottlerGuard

3.6 Hypertext Transfer Protocol Secure (HTTPS) Enforcement

All data transmitted will be encrypted using HTTPS, with HTTP Strict Transport Security (HSTS) enforcing secure connections. HTTP Strict Transport Security (HSTS) is a standard that helps ensure secure data transmission by requiring that browsers always use HTTPS to connect to a website.

Prevents protocol downgrading

HSTS prevents attacks that redirect browsers from HTTPS to an attacker-controlled server.

Prevents cookie hijacking

HSTS encrypts all communication between the browser and the web server, making it difficult for attackers to access or alter sensitive data.

```
const ctx = host.switchToHttp();
const response = ctx.getResponse<Response>();
let status = exception.getStatus();
response.status(HttpStatus.OK).json({
  code: status,
  message: msg,
  data: null,
});
}

private isExpectedException(exception: HttpException): boolean {
  switch (exception instanceof HttpException && exception.getStatus()) {
    case HttpStatus.BAD_REQUEST:
      return true
    case HttpStatus.UNAUTHORIZED:
      return true
  }
}
```

HTTPS

3.7 Database Security

Database security protects data from unauthorized access, theft, and other security threats.

AES-256 encryption

AES-256 encryption is a highly secure encryption algorithm that uses a 256-bit key to scramble data into an unreadable format. This makes it difficult for attackers to guess or crack the key, and protects data even if the security of the infrastructure is compromised.

```
200 // AES-256 encrypt and decrypt
201 private encrypt(text: string, privateKeyHex: string): string {
202     const iv = crypto.randomBytes(16);
203     const cipher = crypto.createCipheriv('aes-256-cbc', Buffer.from(privateKeyHex, 'hex'), iv);
204     let encrypted = cipher.update(text, 'utf8', 'hex');
205     encrypted += cipher.final('hex');
206
207     return iv.toString('hex') + ':' + encrypted;
208 }
209 private decrypt(encryptedText: string, privateKeyHex: string): string {
210     const [ivHex, encrypted] = encryptedText.split(':');
211     const iv = Buffer.from(ivHex, 'hex');
212
213     const decipher = crypto.createDecipheriv('aes-256-cbc', Buffer.from(privateKeyHex, 'hex'), iv);
214     let decrypted = decipher.update(encrypted, 'hex', 'utf8');
215     decrypted += decipher.final('utf8');
216
217     return decrypted;
218 }
219 }
220
```

	id	password	createdAt	updatedAt
	2	\$2b\$10\$S9OpndNJ7u75E..pNXiHjuvPeVVyJ9f8ldLm/cegHkyWYJW9/73/.	2024-09-19 20:40:12.151963	2024-11-12 15:41:39.655794
	3	\$2b\$10\$fWjwr.w.FtsxzW6NhrLWoOndH8CMucdwJNPHsYPDZvbp3hD/R5...	2024-10-27 22:24:10.774892	2024-11-12 15:41:39.669711
	4	\$2b\$10\$IzzF0yeqBjt/NLvVCpbdc.RzDs5EAP8UvflLidyMvnk18lv7WTGwO	2024-11-12 15:35:59.858905	2024-11-12 15:35:59.858905

Database Security - AES-256

3.8 Cross-Site Request Forgery (CSRF) Protection

Cross-Site Request Forgery (CSRF) is a type of attack that tricks a user into performing actions they did not intend to on a web application where they are authenticated. This can lead to unauthorized actions like changing account details, transferring funds, or other state-changing operations

SameSite Attribute

SameSite is an attribute of HTTP cookies that helps by ensuring that cookies (which often include session tokens) are not sent with cross-site requests. This means that even if a malicious site tries to trick a user's browser into making a request to your site, the browser will not include the session cookie, thus preventing the CSRF attack from succeeding.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalInterceptors(new ResponseInterceptor());
  app.useGlobalPipes(new ValidationPipe({ stopAtFirstError: true, transform: true }));
  app.useGlobalFilters(new AllExceptionsFilter());

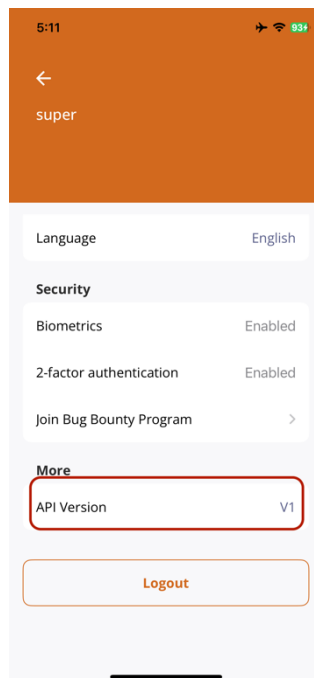
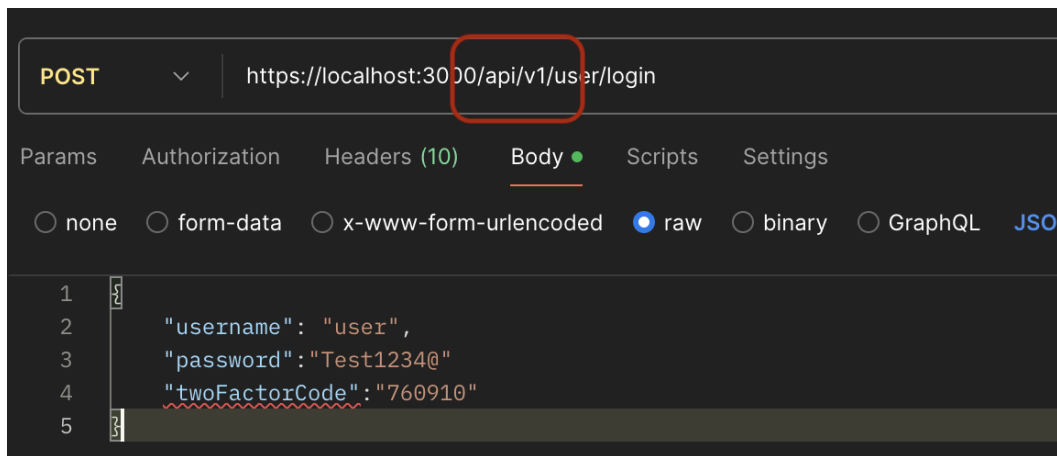
  // add cookie-parser middleware for parsing cookie
  app.use(cookieParser());
  // use csurf middleware for enabling CSRF protection, Post request should add X-CSRF-TOKEN for header
  // /api/v1/user/get-new-csrf-token is used to get new csrf token
  app.use(csurf({ cookie: {
    httpOnly: true,
    sameSite: "strict",
  }}}));

  app.setGlobalPrefix('api/v1');
  let port = parseInt(process.env.PORT, 10) || 3000;
  await app.listen(port);
  console.log(`Application is running on: ${await app.getUrl()}`);
}
bootstrap();
```

CSRF Protection - SameSite Attribute

3.9 API Versioning

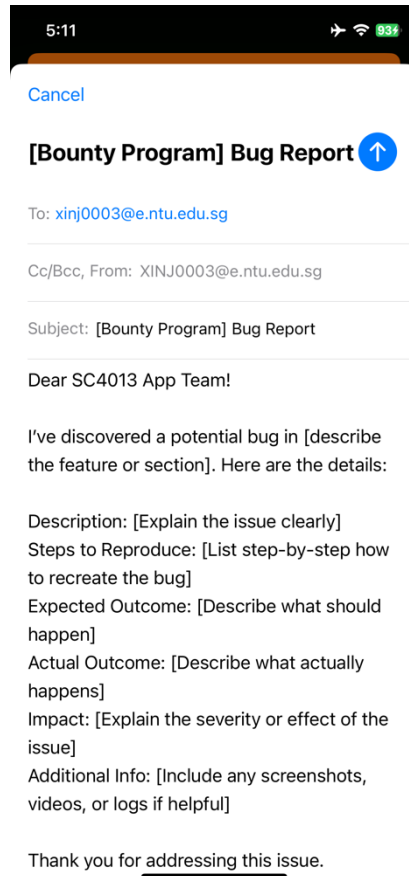
The use of API versioning control to API versioning to manage future updates securely.



API Versioning

3.10 Bug Bounty Program

Implementing a bug bounty program to encourage responsible reporting of security issues.



5:11 93%

Cancel

[Bounty Program] Bug Report ↑

To: xinj0003@e.ntu.edu.sg

Cc/Bcc, From: XINJ0003@e.ntu.edu.sg

Subject: [Bounty Program] Bug Report

Dear SC4013 App Team!

I've discovered a potential bug in [describe the feature or section]. Here are the details:

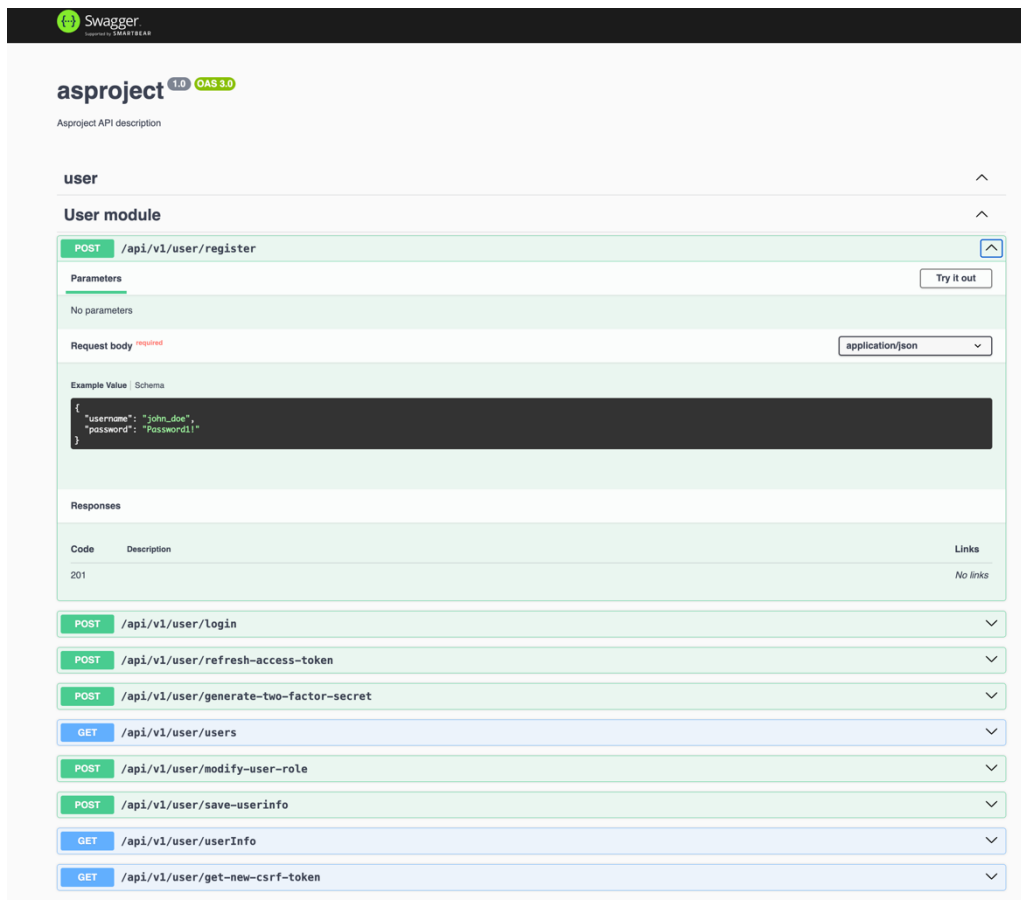
Description: [Explain the issue clearly]
Steps to Reproduce: [List step-by-step how to recreate the bug]
Expected Outcome: [Describe what should happen]
Actual Outcome: [Describe what actually happens]
Impact: [Explain the severity or effect of the issue]
Additional Info: [Include any screenshots, videos, or logs if helpful]

Thank you for addressing this issue.

Bug Bounty Program

3.11 API Documentation

Implementing a comprehensive API documentation using Swagger.



API documentation - Swagger

4. Software Architecture

4.1 Dependency Management

The use of third-party libraries accelerates development but introduces potential security vulnerabilities if dependencies are outdated or not properly managed. Ensuring secure, up-to-date dependencies is critical for maintaining the API's integrity and resilience.

Automated Dependency Scanning

We will integrate automated tools, such as GitHub Dependabot or npm audit, to continuously monitor dependencies for vulnerabilities. These tools provide alerts when vulnerabilities are detected, allowing timely updates.

Regular Dependency Updates

Periodic updates will be scheduled for all third-party libraries, particularly security-sensitive ones, like authentication and encryption libraries. Regular updates help reduce the risk of security gaps due to outdated code.

Vulnerability Management

When vulnerabilities are identified in dependencies, immediate action will be taken to patch or replace the affected libraries. In cases where updates are not immediately available, compensating controls will be implemented to mitigate risks temporarily.

4.2 Secure Development Lifecycle (SDLC)

Security will be embedded throughout the entire development lifecycle of Shobee's APIs, ensuring that vulnerabilities are minimized at every stage rather than addressed only in the final implementation. This proactive approach reduces security risks, enhances code quality, and promotes secure practices across the team.

Requirements Gathering

Security requirements will be clearly defined at the start, based on industry standards (e.g., OWASP API Security Top Ten) and regulatory needs. Security-specific requirements, such as authentication, authorization, and data protection, will be identified as foundational components of the API design.

Design Phase

Security will be incorporated into the architectural design by choosing secure communication protocols, access control strategies, and data encryption methods. Apply secure design principles such as least privilege, defence in depth, and secure defaults. Perform security architecture reviews to ensure that the design meets security requirements and mitigates identified threats.

Development Phase

Conduct regular code reviews with a focus on security to catch vulnerabilities early. Follow secure coding guidelines to prevent common vulnerabilities like SQL injection, cross-site scripting (XSS), and buffer overflows.

Testing Phase

Use static analysis security tools (SAST) to automatically detect vulnerabilities in the code. Perform dynamic application security testing (DAST) to identify potential vulnerabilities from an attacker's perspective. Integrate security testing into the continuous integration/continuous deployment (CI/CD) pipeline to ensure ongoing security checks. Automated scanning for vulnerabilities in third-party libraries will be part of each build, ensuring dependencies remain secure.

Secure Deployment

Implement secure configuration management practices to prevent misconfigurations. Post-deployment, security monitoring will be continuous. Logs will be monitored for suspicious activity using the ELK Stack, and intrusion detection systems (IDS) will help identify potential breaches. Develop and maintain an incident response plan to quickly address security breaches.

4.3 Penetration testing

Regular penetration testing is essential for identifying vulnerabilities within Shobee's API before attackers can exploit them. By simulating real-world attacks, penetration testing provides insights into potential weaknesses in the API's security posture and enables proactive mitigation.

Scope of Testing

- Testing the robustness of JWT and refresh tokens, MFA implementation, and Role-Based Access Control (RBAC).
- Testing for common injection attacks, such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).
- Assessing the effectiveness of rate limiting and throttling mechanisms to prevent abuse and brute-force attacks.
- Identifying exposed endpoints and testing their response to various types of malicious input.

5. Future actions for continuous improvement

1. Perform a comprehensive threat modelling exercise tailored to our e-commerce platform to identify specific risks and vulnerabilities unique to our application.
2. Create a robust plan for secure key management, detailing methods for storing and rotating API and encryption keys.
3. Research and document additional security headers we intend to implement beyond HSTS.
4. Establish a schedule for regular security audits and updates for both our system and its dependencies.
5. Develop project-specific secure coding guidelines tailored to our team.
6. Integrate security-focused code reviews into our development process.
7. Plan secure data backup and recovery procedures, with a focus on protecting sensitive user information.
8. Explore real-time threat detection and response options, such as integrating with a SIEM (Security Information and Event Management) system.
9. Draft an incident response plan specific to our API and e-commerce platform.

6. Conclusion

In building Shobee's API infrastructure, we have prioritized security at every stage of the development lifecycle to ensure a resilient and trustworthy platform for users and developers alike. By implementing a comprehensive suite of security features, including robust authentication and authorization, input validation, rate limiting, and secure error handling, we aim to protect against a range of threats that commonly target e-commerce applications. Key protections, such as HTTPS enforcement, database encryption, and CSRF defences, enhance data integrity and user privacy, forming a strong foundation for Shobee's operations.

Beyond technical features, our strategy includes systematic security practices like dependency management, thorough logging and monitoring, and regular penetration testing. Together, these initiatives help Shobee stay vigilant against potential vulnerabilities and adapt to emerging threats.

Shobee not only to meet current security standards but to maintain an adaptable, secure API infrastructure that can evolve with the platform's growth. By embedding security into the core of our API design, Shobee is equipped to deliver a seamless and secure shopping experience, protecting user data and fostering trust among our user base. This project highlights our commitment to delivering a secure, resilient, and scalable platform, ensuring that Shobee remains prepared to meet future challenges as a secure e-commerce solution.