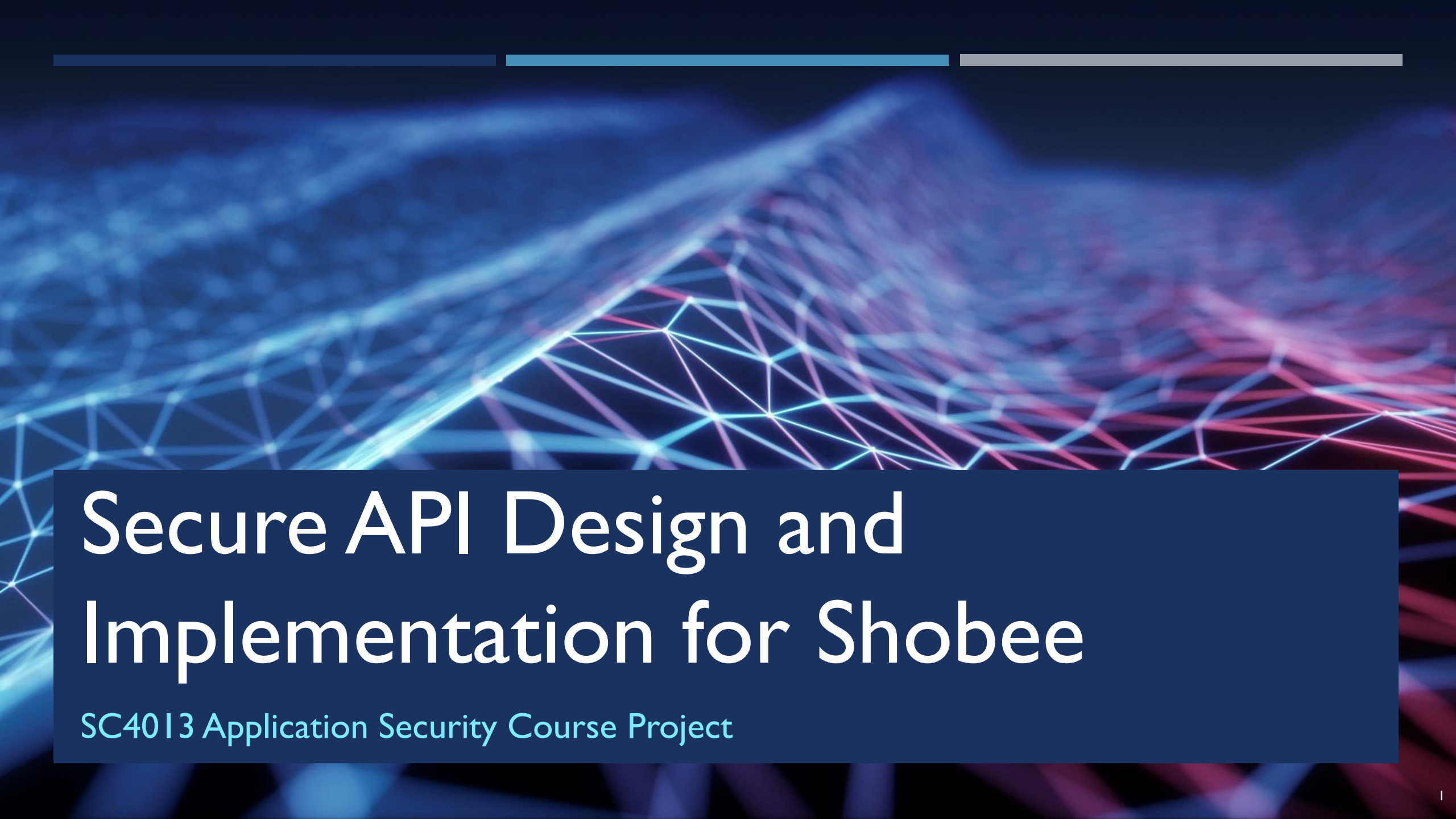

Secure API Design and Implementation for Shobee

SC4013 Application Security Course Project

Team Members

Xin Jie

U2120663H

Tan Choon Wee

U2120106H

Background

- Shobee, a new e-commerce platform, aims to deliver a secure and seamless shopping experience.
- Challenges: protecting payment information, preventing fraud, and ensuring the integrity of user accounts and transactions.
- To achieve these, we are developing secure APIs to handle user interactions and backend services, including user authentication, and product catalog management.

Key Security Features to Implement

1. Authentication
   - JWT
   - MFA

2. Authorization and Access Control
   - Role-Based Access Control
   - Principle of Least Privilege

3. Input Validation and Sanitization
   - Express-Validator
   - Node.js
   - Express,js

4. Rate Limiting and Throttling
   - ThrottlerGuard

5. HTTPS Enforcement
   - HSTS

6. Database Security
   - AES-256

7. CSRF Protection
   - SameSite Attribute

JWT (JSON Web Tokens)

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

- **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.
  - **Store JWT in cookies**
    - Cookies are automatically sent to the server and are not accessible via JavaScript.

MFA (Mult-Factor Authentication)

MFA (Mult-Factor Authentication)

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

Time-based one-time passwords (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.

Authentication - JWT

src > auth > auth.service.ts > ...

```ts
import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { AuthUserInfo } from './auth.dto';

@Injectable()
export class AuthService {
    constructor(private readonly jwtService: JwtService) {}

    // generate JWT token
    async generateUserToken(user: AuthUserInfo): Promise<string> {
        return this.jwtService.sign(user);
    }

    async generateRefreshToken(user: AuthUserInfo): Promise<string> {
        return this.jwtService.sign(user, {
            expiresIn: '7d',  }
        );
    }

    //  varify JWT token
    async validateUser(token: string): Promise<AuthUserInfo | null> {
        try {
            return this.jwtService.verify<AuthUserInfo>(token);
        } catch (error) {
            // return null if meet invalid JWT token
            return null;
        }
    }
}
```

Authentication - 2FA

```ts
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");
        }
    }
```

Authorization and Access Control

- Authorization and access control are related to the principle of least privilege (POLP) and role-based access control (RBAC) in the following ways:

  - **Principle of Least Privilege (POLP)**
    - A cybersecurity best practice that limits users' access to only what is required to perform their job duties. It's a fundamental component of RBAC.

  - **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's more scalable and easier to manage than access control list (ACL).

  - **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.

Authorization and Access Control

```
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
    }
}
```

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.

Express-Validator and Node.js

- **Using express-validator and Node.js to Prevent Injection Attacks**
  - Express-validator is a popular library in the Node.js ecosystem that helps with both input validation and sanitization. Here's 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 XSS attacks.

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

Input Validation and Sanitization - Validator

```
"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/fwsnOITVgYJ8yojSy6CaXtO9V0Cc6ZQZ8h8m4UBuLwQ==",
  "license": "MIT",
  "dependencies": {
    "@types/validator": "^13.11.8",
    "libphonenumber-js": "^1.10.53",
    "validator": "^13.9.0"
  }
},
```

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**
    - A general term that limits the number of requests made to a network, server, or resource. It's 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.
  - **Throttling**
    - A more severe method that completely stops clients from making requests for a certain period. It's 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.

  - **Rate Limiting**
    - ThrottlerGuard enforces a maximum number of requests from a particular user or IP within a defined time 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.

Rate Limiting and Throttling - ThrottlerGuard

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

HTTPS (Hypertext Transfer Protocol Secure) Enforcement

- 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.

HTTPS Enforcement

```
    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
    }
```

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.

Data Security - AES-256 Encryption

```
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     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 
```

CSRF (Cross-Site Request Forgery) Protection

- Cross-Site Request Forgery (CSRF) is a type of attack that tricks a user into performing actions they didn't 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.

CSRF Protection – SameSite Attribute

```
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();
```

Future actions for more comprehensive security:

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. Consider implementing a bug bounty program or vulnerability disclosure policy to promote responsible security issue reporting.
6. Develop project-specific secure coding guidelines tailored to our team.
7. Integrate security-focused code reviews into our development process.
8. Plan secure data backup and recovery procedures, with a focus on protecting sensitive user information.
9. Explore real-time threat detection and response options, such as integrating with a SIEM (Security Information and Event Management) system.
10. Draft an incident response plan specific to our API and e-commerce platform.

THANK YOU

2024/2025 Yr 4