

ICT3103/ICT3203

Secure Software Development

Project Deliverables 2:

Secure Software Requirement Analysis &
Design

Food Ordering System



Lab 7 Group 10

Name	Student ID
Law Jun Hao	2101081
Khor Chai Hong	2103077
Sim Yu Cheng	2102520
Edwin Toh	2100671
Lee Yan Rong	2102608
Olivia Delores Xavier	2102528

Link to repository: <https://github.com/kch-chaihong/food-ordering-system.git>

Table of Contents

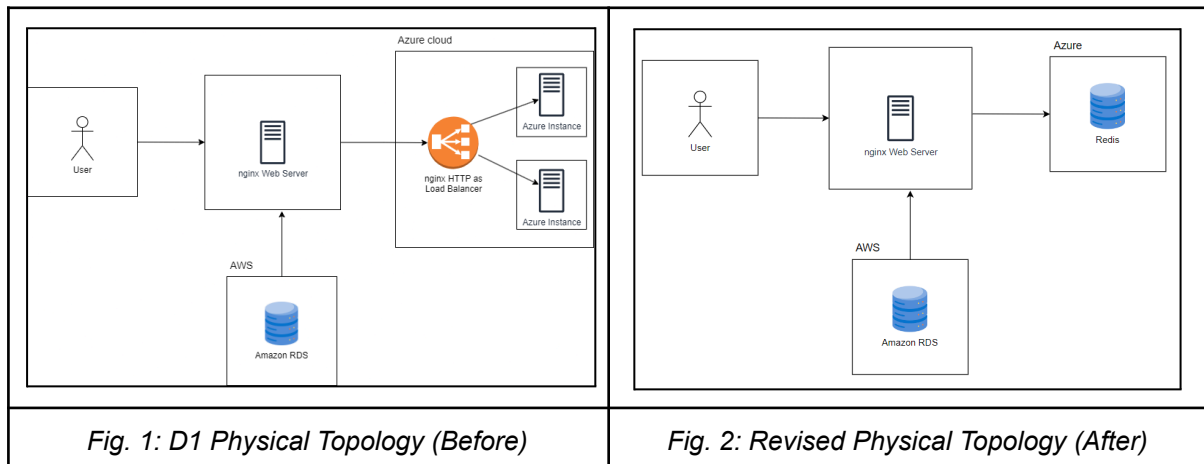
1. Design Review	3
1.1. Security Architecture	3
1.2. Threat Modelling	3
1.2.1. Data Flow Diagram	3
1.3. Account Lockout Policy	4
1.4. Technologies Used	4
1.5. Load balancing	4
1.6. Logging (logrotate)	4
2. Secure Software Implementation	5
2.1. CI/CD	5
2.2. GitHub	7
2.2.1. Files and Directories	7
2.2.2. Implementation Phase	7
2.3. Secure Implementation of Login	9
2.3.1. Sequence diagram	9
2.4. Secure implementation of Session Management	10
2.4.1. Implementation of JWT and Refresh Token	10
2.4.2. Implementation of Session Token	11
2.4.3. Generation of session, JWT and Refresh Token	11
2.4.4. Security aspect of session tokens and JWT	11
2.5. Secure implementation of Access Control	12
2.5.1. Access Control to API endpoints	12
2.5.2. Access Control to pages	12
2.6. Secure Coding	13
2.6.1. OWASP Top 10 Proactive Controls	13
2.6.1.1. C1 - Define Security Requirements	13
2.6.1.2. C2 - Leverage Security Frameworks and Libraries	14
2.6.1.3. C3 - Secure Database Access	15
2.6.1.4. C4 - Encode and Escape Data	16
2.6.1.5. C5 - Validate All Inputs	17
2.6.1.6. C6 - Implement Digital Identity	19
2.6.1.7. C7 - Enforce Access Controls	23
2.6.1.8. C8 - Protect Data Everywhere	24
2.6.1.9. C9 - Implement Security Logging and Monitoring	26
2.6.1.10. C10 - Handle all Errors and Exceptions	27
3. Test Automation	28
3.1. Dependency-Check	28
3.2. End-to-end (e2e) Testing	29

1. Design Review

1.1. Security Architecture

Revised Physical Topology:

The load balancer was removed from the physical diagram and is updated to include the Redis server, which is responsible for storing data related to OTP and password reset.

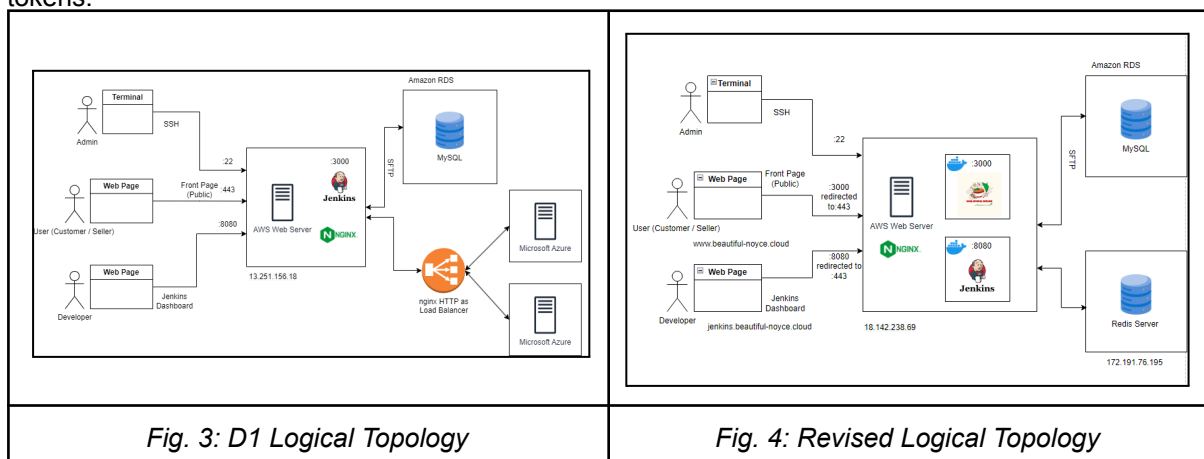


Link to full image of Fig 2 here :

<https://drive.google.com/file/d/1eboRMSrmncqC6kh3SfCQ36cbBa1uLXpl/view?usp=sharing>

Revised Logical Topology:

The new logical topology has been revised to incorporate a new domain name, www.beautiful-noyce.cloud. To maintain the use of protocol 443 for both the main site and Jenkins, the DNS settings have been configured with the addition of the "jenkins" prefix to the domain name. The EC2's IP address has also been updated. The topology has been revised to clearly represent that both Jenkins and the main site have been containerised using Docker. Additionally, a Redis server is included as it communicates with the EC2 server for the storage and management of verification tokens.



Link to full image of Fig 4 here :

https://drive.google.com/file/d/1PQxV7O0CtMDUgwhdgx1P6i__2eDImGNn/view?usp=sharing

1.2. Threat Modelling

1.2.1. Data Flow Diagram

The Level 0 data flow diagram shows how data flows from the source to the destination and back. It includes processes related to entities like the Customer and Seller, as well as components such as

the Web Application, Web Server, SQL Database, and External APIs like Stripe and CAPTCHA.

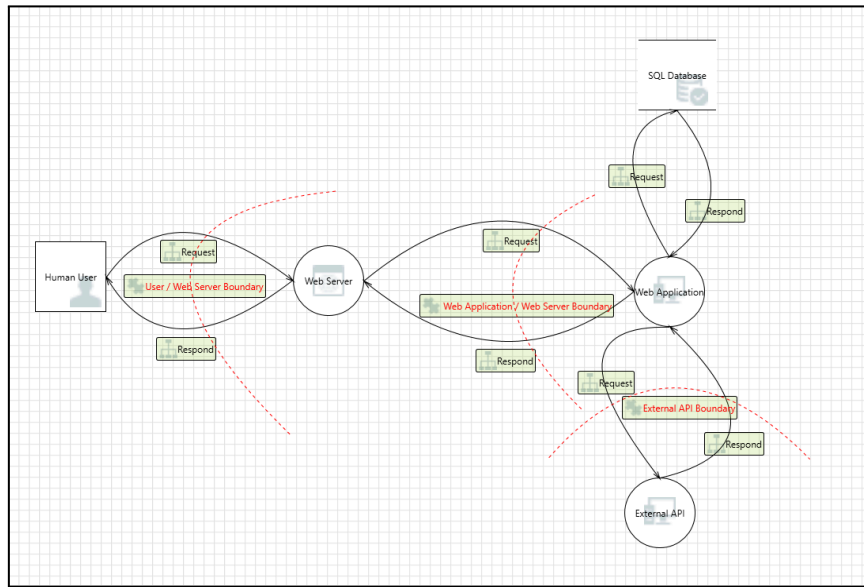


Fig. 5: Data Flow Diagram

1.3. Account Lockout Policy

We did not implement the Account Lockout Policy as stated in D1, as we felt that it was too aggressive and could be exploited by malicious parties. For example, attackers can repeatedly attempt to log in to deliberately trigger the account lockout policy, resulting in legitimate users being locked out of their own accounts, causing disruptions and inconvenience. Thus, we implemented CAPTCHA challenges to deal with brute-force logins.

However, after reviewing the OWASP Web Security Testing Guide, we understood that CAPTCHA should not replace a lockout mechanism. It is possible for a balance between security and usability with the approach of a self-service account unlock mechanism.

In tandem with CAPTCHA (which will kick in after 5 failed login attempts), the account lockout policy will disable the account after 10 failed login attempts and send a password reset email to the registered email address, allowing the legitimate account owner to unlock their account.

1.4. Technologies Used

We had to switch from Twilio SendGrid to Nodemailer as Twilio SendGrid enforced account creation limits, disallowing us from creating accounts to utilise its service. This happened because we were found to violate their usage policies or terms of service. As a result, Nodemailer is now our chosen email service provider, allowing us to continue sending emails for the OTP and the Reset Password.

1.5. Load balancing

In the previous report (D1), the inclusion of load balancing was initially proposed to enhance the system's availability and provide better fault tolerance. However, due to resource constraints, especially the provisioning of additional server instances for which we lack the necessary credits to do so, we decided that it was infeasible to implement load balancing. This decision allowed us to prioritise other security features of the CIA-AAA model instead.

1.6. Logging (logrotate)

Logrotate is a Linux tool that is designed for ease of administration by allowing users to indicate a way of moving log files around without interrupting their logging process. The services provided range from log rotation, compression to mailing of logs. We decided to use this to configure the logging settings for both front end Jenkins as well as backend nginx services.

In addition, we mentioned in D1 that web access logs will be logged by ModSecurity WAF and Uncomplicated Firewall (UFW). Due to the potential requirement to disable WAF and UFW for D3 QA testing, we decided to seek alternative solutions for logging.

2. Secure Software Implementation

2.1. CI/CD

Continuous Integration and Continuous Deployment (CI/CD) are software development practices that aim to automate and streamline the process of building, testing and deploying of software. The web application uses Jenkins, an open-source automation tool, for its CI/CD pipelines. Jenkins has also been containerised with Docker to optimise space usage and maintain an isolated environment for our CI/CD processes. The Jenkins pipeline, defined in a Jenkinsfile, is linked to the project's GitHub repository via Github webhook integration and is automatically triggered by pushes and pull requests to the repository. All credentials necessary for the CI/CD process are securely stored and retrieved using Jenkins Credentials Manager.

Furthermore, Docker-out-of-Docker (DooD) has been used as an approach to running Docker in Docker containers. This methodology allows us to execute Docker commands from within a Docker container, effectively allowing us to create, manage and run additional Docker containers for our application from within the BlueOcean Jenkins container. In terms of security, this approach offers several distinct advantages.

1. Isolation

Dood offers a high degree of isolation of host resources because Docker commands are run within the Jenkins container. This isolation reduces the risk of a compromised Docker container gaining access to the host system's resources. On the other hand, the Docker-in-Docker (DinD) approach of running Docker daemon within a container, could grant an attacker access to the host's Docker socket, which potentially impacts the host system.

2. Reduced Attack Surface

DooD keeps Docker socket outside of Jenkins container, thereby limiting potential attack vectors.

3. Resource Control

Resources are managed within the Jenkins container, preventing resource contention and enhances stability of the CI/CD environment. DinD, on the other hand, may cause resource exhaustion when it competes with the host system for resources.

4. Clear Security Boundaries

Any security vulnerabilities or issues that arise during the CI/CD process are confined to the Jenkins container, reducing the impact on the host system and other running containers.

Our Jenkins pipeline comprises the following stages as shown in Fig. 6:

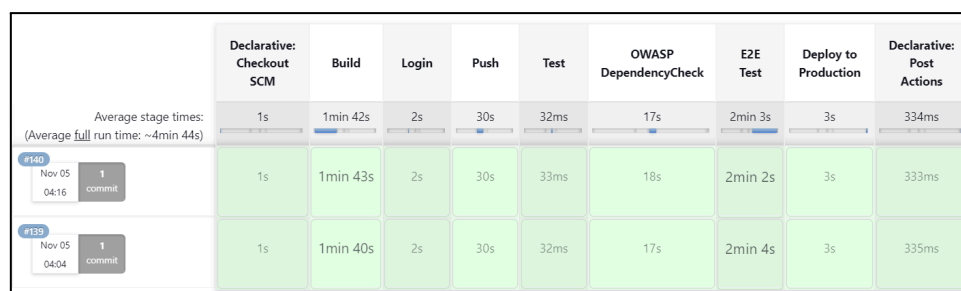


Fig. 6: The stages of the 'food-ordering-pipeline'

Build Stage: This stage is responsible for building the Docker image for the web application and tagging it with the current build ID.

Login Stage: The pipeline securely passes the DockerHub credentials using environment variables in order to log into DockerHub.

Push Stage: This stage tags the built Docker image with the 'latest' tag and pushes it to DockerHub. Once the latest image has been uploaded, it removes the image to save space.

Test Stage: This stage ensures that the web application is secure and functions as expected. Hence, the OWASP Dependency Check tool is used to scan for known vulnerabilities in project dependencies and generates a report in XML format.

Project: food-ordering-pipeline #140	
Scan Information (show all):	
<ul style="list-style-type: none"> • <i>dependency-check version:</i> 8.4.2 • <i>Report Generated On:</i> Sat, 4 Nov 2023 20:19:05 GMT • <i>Dependencies Scanned:</i> 55 (55 unique) • <i>Vulnerable Dependencies:</i> 0 • <i>Vulnerabilities Found:</i> 0 • <i>Vulnerabilities Suppressed:</i> 0 • ... • <i>NVD CVE Checked:</i> 2023-11-04T19:34:42 • <i>NVD CVE Modified:</i> 2023-11-04T18:00:01 • <i>VersionCheckOn:</i> 2023-11-02T15:25:00 • <i>kev.checked:</i> 1699126518 	

Fig. 7: Summary of a scan conducted with OWASP Dependency Check tool

Next, the End-to-End (E2E) testing substage will start, ensuring the application functions correctly as a whole by simulating real user interactions. This is achieved with Cypress, a popular E2E testing framework, within the existing 'jenkins-docker-container' Docker container. Following the completion of this substage, a summary of the test results will be generated as seen in Fig. 8, providing information such as time taken for the tests to run, the number of tests that passed, failed, pending or skipped.

Spec	Tests	Passing	Failing	Pending	Skipped
✓ app.cy.js	00:15	3	3	-	-
✓ loginCustomer.cy.js	00:07	3	3	-	-
✓ loginSeller.cy.js	00:06	3	3	-	-
✓ payment.cy.js	01:02	3	3	-	-
✓ All specs passed!	01:30	12	12	-	-

Fig. 8: Results of the Cypress framework during E2E testing stage

Deploy to Production Stage: This stage first securely connects via SSH, to the designated EC2 instance where Jenkins Docker is running. It then stops and removes any existing containers and images related to the web application to create a clean slate for deployment. Next, the stage deploys the web application using a Docker run command and publishes it on port 3000, ensuring the web application is ready to serve in a production environment. Finally, a Docker system prune command is executed to perform a cleanup to remove dangling Docker resources.

Based on Fig. 9, which was taken from Jenkins Logs, there is clear evidence of the auto-triggering of the CI/CD process through the GitHub webhook. The auto-triggering process occurred multiple times, denoted by the different build numbers. In response to the "Received PushEvent" entries for the GitHub repository, the message "Poked food-ordering-pipeline" followed thereafter, signifying that the CI/CD pipeline was invoked.

```

Received PushEvent for https://github.com/kch-chaihong/food-ordering-system from 140.82.115.241 => http://18.142.238.69:8080/github-webhook/
Nov 04, 2023 7:48:08 PM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber$1 run
Poked food-ordering-pipeline
Nov 04, 2023 7:48:09 PM INFO com.cloudbees.jenkins.GithubPushTrigger$1 run
SCM changes detected in food-ordering-pipeline. Triggering #137
Nov 04, 2023 7:57:10 PM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber onEvent
Received PushEvent for https://github.com/kch-chaihong/food-ordering-system from 140.82.115.29 => http://18.142.238.69:8080/github-webhook/
Nov 04, 2023 7:57:10 PM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber$1 run
Poked food-ordering-pipeline
Nov 04, 2023 7:57:11 PM INFO com.cloudbees.jenkins.GithubPushTrigger$1 run
SCM changes detected in food-ordering-pipeline. Triggering #138
Nov 04, 2023 8:03:54 PM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber onEvent
Received PushEvent for https://github.com/kch-chaihong/food-ordering-system from 140.82.115.35 => http://18.142.238.69:8080/github-webhook/
Nov 04, 2023 8:03:54 PM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber$1 run
Poked food-ordering-pipeline
Nov 04, 2023 8:03:56 PM INFO com.cloudbees.jenkins.GithubPushTrigger$1 run
SCM changes detected in food-ordering-pipeline. Triggering #139
Nov 04, 2023 8:16:26 PM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber onEvent
Received PushEvent for https://github.com/kch-chaihong/food-ordering-system from 140.82.115.34 => http://18.142.238.69:8080/github-webhook/
Nov 04, 2023 8:16:26 PM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber$1 run
Poked food-ordering-pipeline
Nov 04, 2023 8:16:27 PM INFO com.cloudbees.jenkins.GithubPushTrigger$1 run
SCM changes detected in food-ordering-pipeline. Triggering #140
Nov 05, 2023 5:27:39 AM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber onEvent
Received PushEvent for https://github.com/kch-chaihong/food-ordering-system from 140.82.115.34 => http://18.142.238.69:8080/github-webhook/
Nov 05, 2023 5:27:39 AM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber$1 run
Poked food-ordering-pipeline
Nov 05, 2023 5:27:40 AM INFO com.cloudbees.jenkins.GithubPushTrigger$1 run
SCM changes detected in food-ordering-pipeline. Triggering #141
Nov 05, 2023 5:40:44 AM INFO org.jenkinsci.plugins.github.webhook.subscriber.DefaultPushGHEventSubscriber onEvent
Received PushEvent for https://github.com/kch-chaihong/food-ordering-system from 140.82.115.113 => http://18.142.238.69:8080/github-webhook/

```

Fig. 9: Jenkins Log snippet indicating successful auto-triggering of CI/CD pipeline through webhook

2.2. GitHub

This section will describe the directories and file organisation in the project GitHub repository, as well as the data flow of the application.

2.2.1. Files and Directories

The application uses the NextJS app router, where all the pages of the application will be stored in the /app folder in the Github repository. In the app directory, nested folder hierarchy defines route structure. Each folder represents a route segment that is mapped to a corresponding segment in a URL path. The route is publicly accessible if a page.js is existing in the route segment. /app/api consists of all the endpoints of the application. /cypress folder will consist of all the test specs or shortcut commands of the testing. All the test specs will be stored under /cypress/e2e folder. /prisma folder will store the migrations and schema of the application. /public will store all the images needed for the application.

2.2.2. Implementation Phase

Every team member had been contributing to the team GitHub repository during the implementation phase. Upon committing to the main branch of the Github, the CI/CD pipeline will be automatically triggered to the Jenkins in AWS EC2 instance.

Fig. 10 shows the member contribution in the GitHub repository.



Fig. 10: Members' contribution in the GitHub repository

✓	51d325b4-7a7d-11ee-85b2-cee1cbabfaa6	push	2023-09-20 13:15:17	...
✓	32e88ba8-7a7d-11ee-9c18-62032ae749bd	push	2023-09-20 13:14:25	...
✓	e92cd0ce-7a7b-11ee-8f47-f3b5b9365946	push	2023-09-20 13:05:11	...
✓	bf04099a-7a51-11ee-8cdf-573b55e3f3b1	push	2023-09-20 12:03:22	...
✓	5f25bc6e-7a4f-11ee-9482-eca613fb3869	push	2023-09-20 11:46:22	...
✓	82dfc254-7a4e-11ee-9ac2-11c8b6bd6f90	push	2023-09-20 11:40:12	...
✓	8280a210-7a4e-11ee-96c0-86466921f360	pull_request.closed	2023-09-20 11:40:12	...
✓	7deb6280-7a4e-11ee-8c3d-24148aa77250	pull_request.opened	2023-09-20 11:40:05	...
✓	65d80680-7a4e-11ee-9713-0527811ae352	push	2023-09-20 01:39:24	...
✓	b8f5cf6e-7a44-11ee-92e6-10e40a18ad62	push	2023-09-20 00:30:09	...
✓	6993b6a2-7a44-11ee-865d-d858e5d7e829	push	2023-09-19 20:27:55	...
✓	7929ff18-7a30-11ee-9d42-21fad558529	push	2023-09-19 18:05:11	...

Fig. 11: Webhook for Jenkins is set for pushes and pull requests

2.3. Secure Implementation of Login

The server authenticates users with NextAuth as the authentication handler. The use of CredentialsProvider allows for the secure verification of user login credentials. With this configuration, the server ensures the security and accuracy of user roles and access control.

As part of the secure implementation of login, the system uses HTTPS which in turn uses the Transport Layer Security (TLS) protocol to establish encrypted communication between the client and server. When implementing HTTPS with Certbot (LetsEncrypt), the private key is securely stored in a designated directory on the EC2 instance as seen in Fig. 12 below. With the proper access control measures and permissions as part of server hardening, combined with Certbot's automatic renewal of certificate and keys, the private key is effectively safeguarded. For example, the directory the keys are stored in are only accessible by root user and has restrictive file permissions set to 600 (read and write permissions for the owner only).

```
listen 443 ssl; # managed by Certbot
ssl_certificate /etc/letsencrypt/live/www.beautiful-noyce.cloud/fullchain.pem; # managed by Certbot
ssl_certificate_key /etc/letsencrypt/live/www.beautiful-noyce.cloud/privkey.pem; # managed by Certbot
include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
```

Fig. 12: Directory where the server private key is stored by Certbot

The Cipher Suite used is TLS 1.3 X25519 AES_128_GCM.

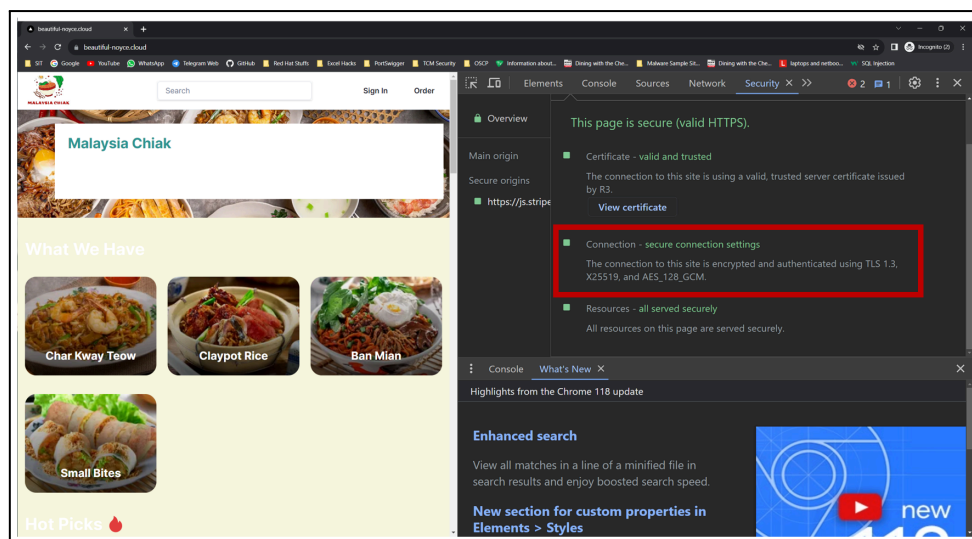


Fig. 13: Cipher Suite used by Certbot / LetsEncrypt

Passwords are stored in a MySQL database located in the AWS RDS and they are hashed using BCrypt with a salt round of 10 which makes the output to be different for each hash. To protect against unauthorised access to the stored passwords, AWS RDS implements Security Groups, which act as virtual firewalls around the RDS database. The Security Groups are finely-tuned to control who can access the RDS instance by permitting only network traffic from authorised sources while blocking all other inbound access.

2.3.1. Sequence diagram

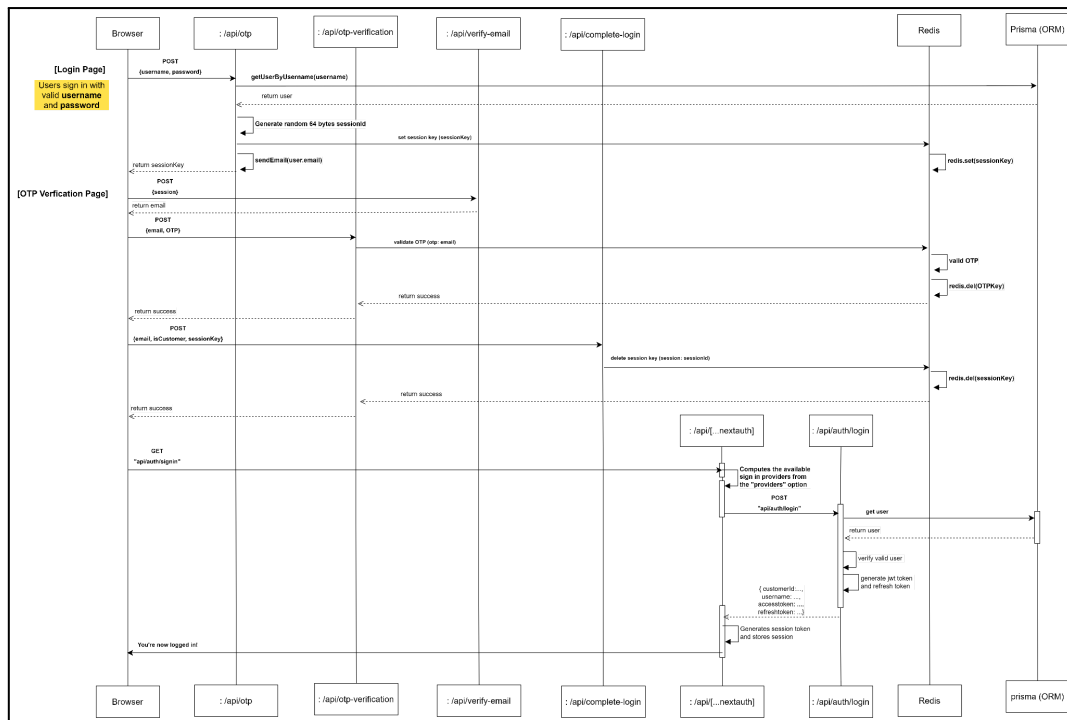


Fig. 14: Login Sequence Diagram

Link to the full image :

<https://drive.google.com/file/d/1Q9-u8tzVNCtVtn6x2hXz1WFEwxyVY5C5/view?usp=sharing>

2.4. Secure implementation of Session Management

The application uses a combination of session and JSON Web Tokens (JWT) to authenticate the user. The team has adopted NextAuth.js to support the JWT and session tokens management for the application. This section will describe the implementation details of the session and JWT in the application.

2.4.1. Implementation of JWT and Refresh Token

The application has implemented the JWT and Refresh Token on top of the session to authenticate the user. The JWT ensures the users can access the website within a particular time frame. It is given a minimal lifetime to ensure that the cybercriminals have minimum time to exploit a user's identity. The refresh token ensures the user can regain a new access token without providing login credentials. The period of the refresh token will be longer than JWT as its purpose is to provide an extended period of access to the application on behalf of a particular user.

The team has allowed the JWT to expire in 15 minutes whereas the refresh token will expire in 48 hours. The usage of the JWT is to authenticate the user when accessing the API endpoints of the application.

The JWT and Refresh Token will consist of the user information such as the customerId, username, full name and the role. Both the tokens will be encrypted by default (JWE) with A256GCM. They are later signed with the NEXTAUTH_SECRET to ensure that it can be verified when authenticating it. The secret key used for signing the JWT is stored securely on the server and never exposed to the client. The generated JWT will result in a unique string that is securely generated to ensure randomness and unpredictability.

```
const customerDetail = {customerId, username, fullname}
const userData = { ...customerDetail, role: 'customer'};
const accessToken = signJwtAccessToken(userData);
const refreshToken = signJwtRefreshToken(userData);
```

Fig. 15: Signing JWT and Refresh Token

2.4.2. Implementation of Session Token

The application has also implemented the session. The rationale of having JWT and session tokens is because the application does not store the session ID in the database as it will have overhead. The JWT is more lightweight than the session token as it stores less user data. The session token will consist of the user information similar to JWT and with the addition of both JWT and refresh token. The usage of the session is to ensure that a user is valid before accessing the pages. The session token is encrypted with the NEXTAUTH_SECRET, which consists of a long random string. This approach ensures that the session token cannot be decrypted to reveal session information without the secret.

```
session.user = {  
  customerId: token.customerId,  
  username: token.username,  
  fullname: token.fullname,  
  role: token.role,  
  accessToken: token.accessToken,  
  refreshToken: token.refreshToken,  
}
```

Fig 16: Session user data

2.4.3. Generation of session, JWT and Refresh Token

The generation of jwt token and refresh token are created on the server side after the user has been authenticated with the credentials and OTP. The API `/api/auth/login` will return a short-lived token(JWT), which expires in 15 minutes, and the refresh token expires in 48 hours, with the user profile returned from that API. The application will generate a session token and store the session. Fig. 17 below shows the detailed flow and object exchange between the classes in the application on the generation of the session, JWT and refresh token.

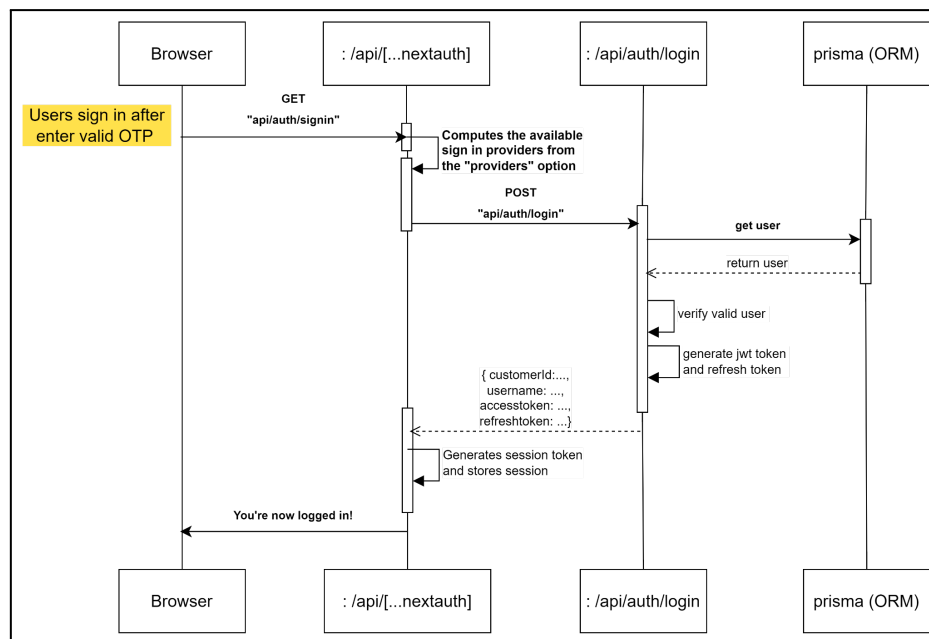


Fig. 17: Generation of session, jwt token and refresh token

Link to the full diagram :

https://drive.google.com/file/d/1725Jjf5W_CiMB-fGkuK1QOj2AudhQ5La/view?usp=drive_link

2.4.4. Security aspect of session tokens and JWT

The team has implemented HTTPS on the server. All the information in the HTTPS is encrypted. The session tokens and JWT will be handled securely over HTTPS to prevent eavesdropping. The session token has been implemented to be HttpOnly flag on the cookie to ensure that it is inaccessible to client-side scripts

2.6. Secure Coding

This section will delve into the secure coding practices implemented by the team, with the relevant code snippets included to showcase how they are integrated into the web application. These practices are aligned with the OWASP Top 10 Proactive Controls.

2.6.1. OWASP Top 10 Proactive Controls

2.6.1.1. C1 - Define Security Requirements

Defining & Choosing Requirements for Software / Security

In the earlier deliverable (D1), we have described the choices made for the project. These choices are broken down into:

- Technologies used
- Functional Requirements, Functional Security Requirements
- Non-Functional Requirements, Non-Functional Security Requirements
- Secure Functional Requirements
- Use Case Diagrams
- Abuse Case Diagrams
- Misuse Case Diagrams
- Potential Risks
- Threat Modelling Based on risks
- Attack surface analysis
- Entire Security Architecture (Logical, Physical, Actors, Data Elements, Access and Control Matrix, External Dependencies)
- Overall Security Design

These written definitions act as a guide and baseline for the development stage of the project, allowing us to work within scope of the requirements needed for the website itself (Both functional and security aspects). With this in mind, changes to the scope as well as choices for the technologies used can still be made during development stages. What is written here in D1 will not be the finalised version, a reworked version (D2) will indicate the newly changes made to the previously stated ones (D1)

Actual Implementation Differences, How does it stack against the planning phase.

Pre-production and Post-production requirements do have some differences in place as seen in [1.0 Design Review](#) of the report. The differences are changes to the ...

- Security Architecture (Physical and Logical)
- Threat modelling
- Policies (Newly added ones, previously missed in D1)
- Replacement Technologies used
- Removal of features (w reasonings)
- New Functions (Logging features & Password Reset)

Stacking both the pre-production and post-production requirements against each other, we can see most of the changes made are minor.

Future Modifications

Future improvements and modifications can be further defined in the D3 report once the QA is conducted based on a further date.

Test Cases & Possible Scenarios

Similar to Future Modifications, Test cases can be further crafted once the first fully functional prototype of the website is up. These test cases would be based on the white box and black box scenario whenever applicable.

2.6.1.2. C2 - Leverage Security Frameworks and Libraries

The web application leverages security frameworks and libraries for the implementation of its security features. Following the best practices, tools like OWASP Dependency Check is used to identify project dependencies and publicly disclosed vulnerabilities.

Bcrypt

Bcrypt is a widely used library for securely hashing passwords. In the web application, Bcrypt is used to hash the passwords of the users to prevent exposing user passwords in plaintext.

Bcrypt Implementation Best Practices:

- 1) Hash user passwords before storing in the database
 - This prevents the actual passwords from being stored in plaintext, making it significantly harder for attackers to access user credentials, even if the database is compromised.

reCAPTCHA

To protect against automated attacks, reCAPTCHA is implemented on the login and signup page. For the login page, reCAPTCHA triggers after five unsuccessful login attempts and is a prerequisite for the web application to process any subsequent login attempts. On the signup page, it is mandatory to solve the reCAPTCHA challenge before account creation is successful. The reCAPTCHA challenge expires after 2 minutes, encouraging prompt user interaction to ensure security.

reCAPTCHA Implementation Best Practices:

1. Trusted Source:

- reCAPTCHA is developed by Google and is a trusted security framework and is widely used in over 15 million websites.
- Actively maintained and supported clear and extensive documentation

2. Use of environment variables:

- Ensures that reCAPTCHA secret key is not exposed

3. Use of 'react-google-recaptcha' wrapper into the client side HTML

- This React reCAPTCHA wrapper is specifically designed for ReactJS, which is the front-end language used in the web application. This specialisation reduces the risk of misconfiguration that can occur as compared to a generic implementation.

zxcvbn-ts

To prevent users from creating common or guessable passwords, zxcvbn-ts, a password strength estimator, is implemented on the signup and password reset page.

zxcvbn-ts goes beyond the traditional password filter as it does not only prevent users from using any common words, but it employs algorithms to recognise patterns to prevent guessable passwords such as 'leekspeak', a technique that replaces similar-looking characters. Additionally, zxcvbn-ts provides an additional HaveIBeenPwned matcher library to prevent users from using data breach passwords. One of the notable features of zxcvbn-ts is the scoring and feedback system, which provides real-time feedback to users about the strength of their passwords. This feedback empowers users to make informed decisions, encouraging them to create strong and unique passwords.

zxcvbn-ts Implementation Best Practices:

1. Trusted Source:

- Actively maintained and supported with clear and extensive documentation

2. zxcvbn-ts in client side HTML

- zxcvbn-ts is a TypeScript of the zxcvbn library which provides full ReactJS. This specialisation not only ensures compatibility but also reduces the risk of misconfigurations. By integrating zxcvbn-ts, a high standard of password security is maintained while minimising potential implementation errors.

Nodemailer

Nodemailer provides a reliable solution to ensure the timely delivery of crucial emails such as sending OTP, reset links or notifications of account lockouts.

Nodemailer Implementation Best Practices:

1. Trusted Source:

- Actively maintained and supported with clear and extensive documentation

- Have 16k stars on GitHub
- Supported by Forward Email and Opensense

2. Use of environment variables:

- Google Email and App Password are not exposed

3. Nodemailer in server-side API:

- Nodemailer is a module for Node.JS application which is the back-end framework used in the web application. This minimises the risk of any misconfiguration that could occur during the email delivery process. This approach simplifies email sending as well as reduces the likelihood of vulnerabilities.

Redis & ioredis

Redis is implemented to enhance the security infrastructure by storing Time-Based OTP, a password reset token for validation.

Redis Implementation Best Practices:

1. Trusted Source:

- Actively maintained and supported with clear and extensive documentation
- Have 62.1k stars and 12.4k on GitHub for Redis and ioredis respectively
- Have 4 billion docker pulls for Redis
- Voted as the most-loved database for five years running

2. Use of environment variables:

- Redis IP address, port number and password are not exposed

3. Redis and ioredis in server-side API:

- ioredis is a full-featured Redis client for Node.JS application, which is the back-end framework used in the web application. This minimises the risk of any misconfiguration that could occur.

4. Secure configuration for Redis Server:

- A Dockerised Redis server is installed on the additional Azure VM separately from the Amazon EC2. Strong password authentication is also included in the Redis server to prevent unauthorised access to the Redis server.

5. Separate information storage:

- The OTP and token are also stored in the Redis server, separate from the database.
- The use of the Redis server also ensures that if the Redis server is compromised, the malicious actor will only have access to the OTP and token.

Isomorphic DOMPurify

Isomorphic-dompurify enhances the security of our web application by sanitising user-generated content to prevent cross-site scripting (XSS) attacks and other security vulnerabilities.

Isomorphic DOMPurify Implementation Best Practices:

1. Trusted Source:

- It is an actively maintained library with robust support and extensive documentation
- Has a substantial following in the open-source community, with 12k stars and a strong GitHub presence, demonstrating its popularity and credibility

2. Use of Whitelisting:

- Leverages a whitelist-based approach to content sanitisation, reducing the risk of malicious content infiltration

2.6.1.3. C3 - Secure Database Access

1. Prisma ORM (Object-Relational Mapping)

The web application uses a database library, Prisma, to provide an ORM solution for interacting with the database. To ensure a robust security posture, we follow these best practices:

a. Secure Configuration

Prisma relies on database connection strings and credentials to establish connections to our database hosted at AWS RDS. Configurations are stored securely in environment variables.

```
# format mysql://USER:PASSWORD@HOST:PORT/DATABASE
DATABASE_URL="mysql://admin:foodorderingpw@food-ordering-database.chzsznls9l9b.ap-southeast-1.rds.amazonaws.com:3306/food_ordering"
```

Fig. 21: Database Information in Environment Variable

b. Safe Query Building

Prisma has a built-in support for parameterised queries. Prisma automatically escapes and sanitises user inputs, ensuring that the user input is treated as data and not executable SQL code, protecting the web application against SQL injection attacks.

```
const user = await prisma.customer.create({
  data: {
    firstName: body.firstName,
    lastName: body.lastName,
    username: body.username,
    password: hashedPassword,
    email: body.email,
    phoneNo: body.phoneNo,
    lastLogin: formattedTimestamp
  }
})
```

Fig. 22: Prisma creates a customer in the database with built-in parameterised queries

2. AWS RDS for MySQL

AWS RDS for MySQL is used as the underlying database management system. To enhance the security of our database and the data it contains, we implement the following practices:

a. Data Encryption

AWS RDS offers robust encryption options for data both at rest and in transit. We configure our RDS instances to use SSL encryption for secure connections, ensuring that data exchanged between the application and the database is protected.

b. Automated Backups and Snapshots

We enabled automated backups and snapshots for our database instance, guaranteeing data recovery and the ability to swiftly restore the database in the event of a failure.

<input type="checkbox"/>	Snapshot name	Snapshot creation time	Status	Snapshot type
<input type="checkbox"/>	rds:food-ordering-database-2023-11-02-18-13	November 03, 2023, 02:14 (UTC+08:00)	Available	Automated
<input type="checkbox"/>	rds:food-ordering-database-2023-11-03-18-14	November 04, 2023, 02:14 (UTC+08:00)	Available	Automated
<input type="checkbox"/>	rds:food-ordering-database-2023-11-04-18-14	November 05, 2023, 02:14 (UTC+08:00)	Available	Automated

Fig. 23: Database Snapshots

3. AWS S3

Amazon S3 serves as our storage solution for food image files for sellers to manage the food menu. To maintain data integrity and security, we adhere to the following measures:

a. Data Encryption in S3

Data stored in S3 are encrypted using server-side encryption (SSE) to protect data at rest. This ensures our stored files are safeguarded against unauthorised access.

2.6.1.4. C4 - Encode and Escape Data

In Fig. 24, encoding occurs, converting the original id to base64 encoding to be used as a parameter in database query performed through Prisma, this is essential as it helps to maintain the integrity of the database structure.


```

const originalId = atob(orderId)

const updatedOrder = await prisma.orders.update({
  where: { ordersId : originalId },
  data: {
    status: bodyStatus
  },
});

```

Fig. 24: Encoding the orderId into base64 with atob

2.6.1.5. C5 - Validate All Inputs

Syntax validity

Form input validation is implemented through Chakra UI, which provides built-in support for validating fields based on different input types. For example, in Fig. 25, the email address input field uses the 'type="email"' attribute to ensure that it follows the standard email format, which includes checking for the presence of the '@' symbol. This validation ensures that the email address entered is a single properly-formed email address.

```

</FormControl>
<FormControl isRequired>
  <FormLabel>Email address</FormLabel>
  <Input
    id="email"
    name="email"
    value={formData.email}
    onChange={handleChange}
    type="email"
  />

```

Fig. 25: Input type of email only accepts valid email addresses for user registration

Email address *

⚠ Please include an '@' in the email address. 'abc.com' is missing an '@'.

Fig. 26: Form alerts the user if an invalid email format is provided

Regular Expressions

To enhance security, a specific regex pattern for NRIC is implemented on both the signup page and password reset page. This regex is designed to prevent users from using their NRIC as their password. For example, in Fig. 27, the regex checks if the pattern starts with the capital or small letter 'S', 'T', 'F' or 'G' followed by a 7-digit number and ending with another capital or small letter.

```
const pattern = /^[stfgSTFG]\d{7}[A-Za-z]$/;
```

Fig. 27: Regex to filter NRIC

When users enter their password on the password input field, the system will check whether the input matches the NRIC pattern specified as shown in Fig 28. If the input is identified to be an NRIC it will provide clear feedback as shown in Fig 29.

```

} else if (pattern.test(newPassword)) {
  setPasswordStrength(0);
  setPasswordSuggestions(['Use a different password.'])
  setPasswordReasons('Password should not be NRIC number.')
}

```

Fig. 28: Code to prevent NRIC as the password

Password *

Weak Password, please choose a stronger password.

Password Suggestions:

- Use a different password.

Reason: Password should not be NRIC number.

Fig. 29: Using a password that is an NRIC

Server-Side validation

Server-side validation is an essential component of input validation, as client-side validation can be manipulated or bypassed therefore, it is important not to trust the client side by default.

reCAPTCHA

For reCAPTCHA server-side validation, in Fig. 30 the validation step is performed before allowing successful signup or login. There is an initial check of the reCAPTCHA response for empty or null values before proceeding to use the function `verifyRecaptcha`, as shown in Fig. 31, to validate the reCAPTCHA response sent by the client, ensuring it is indeed valid and not tampered with.

```
// create new user with hashed password
export async function POST(request){
  const body = await request.json();

  try{
    const recaptchaValue = body.recaptchaValue;
    let isRecaptchaValid = false;

    if(recaptchaValue == 'null' || recaptchaValue == '' || recaptchaValue == null){
      isRecaptchaValid = true;
    }else{
      isRecaptchaValid = await verifyRecaptcha(recaptchaValue);
    }
    const hashedPassword = await bcrypt.hash(body.password, 10);
    if (!isRecaptchaValid) {
      return new NextResponse(JSON.stringify({ message: 'reCAPTCHA verification failed' }), { status: 400 });
    }
    const formattedTimestamp = getCurrentTimestamp();
```

Fig. 30: Server-side reCAPTCHA verification during the signup process

```
async function verifyRecaptcha(clientResponse) {
  const secretKey = process.env.RECAPTCHA_SECRET_KEY;

  try {
    const response = await fetch('https://www.google.com/recaptcha/api/siteverify', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8',
      },
      body: new URLSearchParams({
        secret: secretKey,
        response: clientResponse,
      }),
    });

    if (response.status === 200) {
      const data = await response.json();
      if (data.success) {
        console.log('reCAPTCHA verification success');
        return true;
      } else {
        console.log('reCAPTCHA verification failed');
        return false;
      }
    } else {
      console.log('Failed to verify reCAPTCHA');
      return false;
    }
  } catch (error) {
    console.error('Error while verifying reCAPTCHA:', error);
    return false;
  }
}
```

Fig. 31: verifyRecaptcha function makes a POST request to Google to verify client response

Redis

The web application utilises Redis server-side verification for validating session tokens during OTP generation and password reset. When a user forgets their password, a randomised 64-bytes session token is generated and used as a key to store and retrieve the email address which is stored in the redis server.

Similarly, during OTP issuance, a session token and a 6-digit OTP are generated. The session token serves as a key to store and retrieve the user's email address, while the email address which is retrieved from the session token, acts as a key to store and retrieve the OTP.

File upload

File upload is validated on the client-side to ensure that the files uploaded are 'jpeg', 'jpg' or 'png', to ensure appropriate file types are being uploaded. In addition, file sizes are also validated to ensure that the file size uploaded do not exceed more than 5MB, preventing large or potentially malicious files from overloading the server and causing performance issues.

```
<Input type='file' accept="image/jpeg, image/jpg, image/png" onChange={handleFileInput} />
```

Fig. 32: Input Validation to only accept image files

```
const handleFileInput = (e) => {
  const file = e.target.files[0];

  if (file.size > maxFileSize) {
    setFileErrorMessage('File size exceeds the maximum allowed size (5MB).');
    setSelectedFile();
  }
  else {
    setFileErrorMessage('');
    setSelectedFile(file);
  }
};
```

Fig. 33: Input Validation on file size limit

2.6.1.6. C6 - Implement Digital Identity

Level 1: Passwords

Password Requirements

The password requirements for the web application comply with the OWASP Top 10 Proactive Control - C6: Implement Digital Identity with additional requirements. If those requirements are not met, the user will not be able to create their account or reset their password. The password requirements are as follows:

- Passwords must be a minimum of 8 characters in length.
- All printing ASCII characters, as well as space characters, are accepted.
- Password complexity requirements are not included. Instead, the use of common and guessable passwords is prevented.
- Leetspeak is implemented to prevent predictability
- HavelBeenPwned checker to prevent the use of compromised passwords from data breaches
- Passwords cannot be NRIC numbers.
- Passwords cannot contain user-specific information such as first name, last name, email address or username.
- The passwords are scored on a scale of 0 to 4. Only passwords with a score of 3 or above are eligible for use. The higher the score, the stronger the password.

The implementation for the password requirements is as shown:

Passwords must be a minimum of 8 characters in length. If the input is identified to be less than 8 characters in length, it will be flagged as a weak password.

```
if(newPassword.length < 8){
  setPasswordStrength(0)
  setPasswordSuggestions(['Password is too short.']);
  setPasswordReasons('Password must have at least 8 characters.');
```

Fig. 34: Code to prevent passwords with less than 8 characters

Password *

1234567

Weak Password, please choose a stronger password.

Password Suggestions:

- Password is too short.

Reason: Password must have at least 8 characters.

Fig. 35: Using passwords that are less than 8 characters

All printing ASCII characters, as well as space characters, are accepted.

Figure 36 shows a password input field with the text "best password 10 out of 10" and a "Very Strong Password" status indicator.

Fig. 36: Using passwords with space characters

Passwords that have common names in them and are extremely predictable are not allowed and will be flagged as weak passwords.

Figure 37 shows a password input field with the text "r@ymondt@n!". The status is "Weak Password, please choose a stronger password." The suggestions are: "Add more words that are less common." and "Avoid predictable letter substitutions like '@' for 'a'." The reason is: "Common names and surnames are easy to guess."

Fig. 37: Using common password

Figure 38 shows a password input field with the text "P@\$w0rd1245". The status is "Weak Password, please choose a stronger password." The suggestions are: "Add more words that are less common.", "Capitalize more than the first letter.", and "Avoid predictable letter substitutions like '@' for 'a'." The reason is: "This is similar to a commonly used password."

Fig. 38: Using a predictable password

HavelBeenPwned checker is included to prevent compromised passwords from data breaches. If the password has been compromised before, it will be flagged as a weak password.

Figure 39 shows the code to prevent data-breached password:

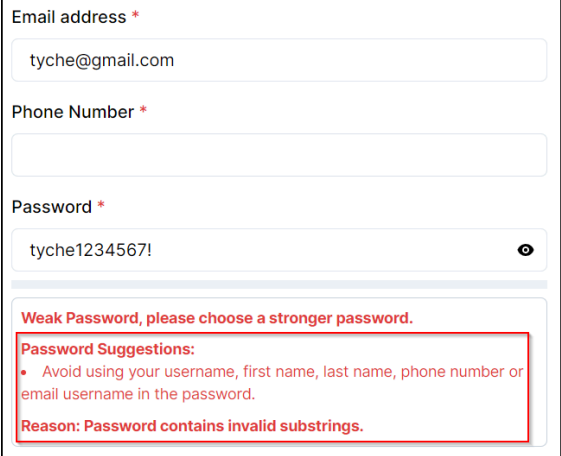
```
const matcherPwned = matcherPwnedFactory(fetch, zxcvbnOptions)
zxcvbnOptions.addMatcher('pwned', matcherPwned)
```

Fig. 39: Code to prevent data-breached password

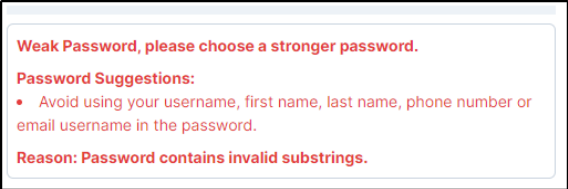
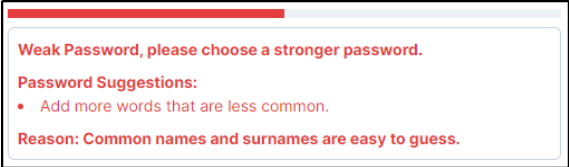
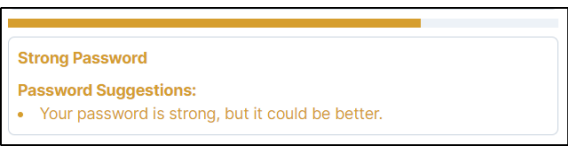

Figure 40 shows a password input field with the text "MYpassword123#". The status is "Weak Password, please choose a stronger password." The suggestions are: "Add more words that are less common." and "If you use this password elsewhere, you should change it." The reason is: "Your password was exposed by a data breach on the Internet."

Fig.40: Using data breached password

Passwords that have the exact substring as the Names or Email Address entered will also be flagged as weak passwords and will not be allowed.

<pre>else if (containsInvalidSubstring) { setPasswordStrength(0) setPasswordSuggestions(['Avoid using your username, first name, last name, phone number or email username in the password.']) setPasswordReasons('Password contains invalid substrings.')</pre>	
<p><i>Fig. 41: Code to prevent the use of substring as other information as password</i></p>	<p><i>Fig. 42: Using passwords that have the same substring as other information</i></p>

The passwords are scored on a scale of 0 to 4. Only passwords with a score of 3 or above are eligible for use. The higher the score, the stronger the password.

<pre><Progress mt={2} value={passwordStrength * 25} // assuming passwordStrength ranges from 0 to 4 colorScheme={ passwordStrength >= 4 ? 'green' : passwordStrength >= 3 ? 'yellow' : 'red' } // adjust colors based on strength size="sm" /></pre>	<pre>setPasswordStrength(result.score); setPasswordSuggestions(result.feedback.suggestions); setPasswordReasons(result.feedback.warning)</pre>
<p><i>Fig. 43: Code to display the password strength</i></p>	<p><i>Fig. 44: Code to display the password strength</i></p>
	
<p><i>Fig. 45: Password with score of 0</i></p>	<p><i>Fig. 46: Password with score of 2</i></p>
	
<p><i>Fig. 47: Password with score of 3</i></p>	<p><i>Fig. 48: Password with score of 4</i></p>

Implement Secure Password Recovery Mechanism

A randomised 64-bytes hexadecimal session token is generated which will be used as a unique URL link that lasts for 15 minutes. The session token will be stored in the Redis server as the key with the email address as the value.

```
const resetToken = randomBytes(64).toString('hex');
const tokenKey = `reset:${resetToken}`;
await redis.set(tokenKey, email, 'EX', 15 * 60);
const transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: process.env.GMAIL_EMAIL,
    pass: process.env.GMAIL_PASSWORD
  }
});
```

Generate a randomized session token

Set the session token key with the email address as value

Set session token to be valid for 15 minutes

Fig. 49: Code to generate and a randomised token that last for 15 mins in Redis server

Single use and expires after an appropriate period

Account will be locked out indefinitely until a password reset request has been issued. If the user had too many attempts to log in and is required to reset his password, the account would have been set to a state of deactivation. This means nothing will change until the password has been successfully reset.

```
updatePassword(storedEmail, hashedPassword)
await redis.del(resetToken);
return new NextResponse(JSON.stringify({
  success: true
}), {
  status: 200,
});
```

Fig. 50: Code to set delete the token after resetting the password

```
// update user with login attempts
updatedUser = await db.update({
  where: {
    username: body.username
  },
  data: {
    loginAttempts: newFailedAttempt, // increment login attempts
    lastLogin: getCurrentTimestamp(),
    active: newFailedAttempt >= 10 ? 0 : 1 // deactivate user's account
  }
})
```

Fig. 51: Code to lockout account after 10 attempts

Upon successful verification, the user would have to contact the customer service to activate their account.

Implement Secure Password Storage

Passwords are hashed using BCrypt with a salt round of 10 on the client side before storing in the database.

Level 2: Multi-Factor Authentication (MFA)

MFA is integrated into various website points. The areas with them mainly require enhanced security to prevent unauthorised and unwanted access to personal information and accounts.

The **Login MFA** works on 2 out of 3 of the conditions ...

- **Something You Know** - Email / Password: Users will start by entering their credentials on the login page. This requires personal knowledge, only known to the user, serving as the first layer of authentication.
- **Something You Have** - Time-based OTP : Subsequently, a unique Time-based OTP is generated and sent directly to the user's designated email address. This possession of this email account will serve as the second layer of authentication. It works because only the authorised user has access to his email account with the OTP attached to it. OTP is set to expire in 4 minutes.

```
const email = customerInfo.email;
const sessionId = randomBytes(64).toString('hex');
const sessionKey = `session:${sessionId}`;
await redis.set(sessionKey, email, 'ex', 240);
const emailSent = await sendEmail(email)
```

Fig. 52: Code to set the time limit for the OTP sent to the authorised user

The **Password Reset MFA** also works on 2 out of 3 of the conditions ..

- **Something You Know** - Email : Users would enter their emails that are linked to the existing account. This requires personal knowledge of knowing what email is linked to the current account, serving the first layer of authentication.
- **Something You Have** - Time-based Magic Link : A unique random 64-byte length in hexadecimal magic URL link would be sent to the user email, which serves as the second layer of authentication as it is something only authorised users would have. The magic link expires in 15 minutes.

```
async function customerInfo(email) {
  try {
    const customerInfo = await prisma.customer.findUnique({
      where: {
        email: email,
      },
      select: {
        password: true,
      },
    });

    if (!customerInfo) {
      throw new Error('Customer not found');
    }

    return customerInfo;
  } catch (error) {
    if (error instanceof Prisma.PrismaClientKnownRequestError) {
      // Handle specific known errors from Prisma
      throw new Error(`Prisma error: ${error.message}`);
    } else {
      // Handle other types of errors
      throw new Error(`Error retrieving customer information: ${error.message}`);
    }
  } finally {
    await prisma.$disconnect();
  }
}
```

Fig. 53: Function “customerInfo” to check if email exists within Database

Level 3: Cryptographic Based Authentication

Cryptographic Based Authentication was implemented to enhance the security of user authentication further. This also enhances the foundation built from Level 2.

Magic Link

- When a user forgets their password, they can provide their email address to receive a unique magic link.
- The application generates a random token using RandomBytes and stores it in the Redis server
- User will then receive a magic link with the token to verify their email
- The link is valid for 15 minutes and, when clicked, directs and prompts the user to reset their password

OTP

- When the user signs in, the application generates a random OTP which will be hashed using SHA-256 and stored in the Redis server.
- The application then creates a token to associate the OTP with the user's account and sets an expiration time of 4 minutes.
- The application then sends the OTP to the user's email address.
- When the user enters the OTP, the application retrieves the OTP from the Redis server and verifies that it is associated with the user's account and has not expired.
- If the OTP is valid, the application authenticates the user and logs them in.

2.6.1.7. C7 - Enforce Access Controls

Avoid hardcoding roles

The code avoids hardcoding roles and instead dynamically sets the role based on the type of user (customer or seller) as shown in Fig. 54 during the authentication process. When roles are inflexibly predefined, attackers can take advantage of this and assume roles they should not have or perform actions that were not intended for their role. This approach reduces the attack surface as there are no fixed roles that could be exploited.

```
const handler = NextAuth({
  providers: [
    // user login credentials
    CredentialsProvider({
      id: 'customer-credentials',
      name: 'Customer-Credentials',
      credentials: {},

      async authorize(credentials, req) {
        return credentials;
      }
    }),

    // seller login credentials
    CredentialsProvider({
      id: 'seller-credentials',
      name: 'Seller-Credentials',
      credentials: {},

      async authorize(credentials, req) {
        return credentials;
      }
    })
  ]
});
```

Fig. 54: Role assignment is determined during the authentication process

Principle of Least Privilege

With distinct roles assigned to customers and sellers during the login process, this allows role-specific actions that grant the minimum required access, permissions, or privileges for carrying out their respective tasks and functions. For example, order creation is role-specific. Customers can create orders, but sellers cannot. This distinction is made to ensure the integrity of the ordering process aligns with the appropriate role, which is the customer's role. As shown in Fig. 55, the POST route for order creation is based on the provided 'customerID' parameter in the request, ensuring that only customers, and not sellers, can place orders.

```
export async function POST(request) {
  const { customerId, address, items } = await request.json();

  try {
    const formattedTimestamp = getCurrentTimestamp();
    const order = await prisma.orders.create({
      data: {
        customerId: customerId,
        timestamp: formattedTimestamp,
        status: 'pending',
        deliveryAddress: address
      }
    });
  }
}
```

Fig. 55: Creation of the order is based on provided 'customerID'

2.6.1.8. C8 - Protect Data Everywhere

Encrypting Data in Transit

The web application ensures the security of data during transmission through the use of Hypertext Transfer Protocol Secure (HTTPS) and security headers.

HTTPS uses the TLS protocol to encrypt data during transmission, ensuring its confidentiality and integrity. First, we register the fully qualified domain names 'www.beautiful-noyce.cloud' and 'jenkins.beautiful-noyce.cloud' with the IP address of the EC2 server, our web application's production environment. Next, URL forwarding is done to redirect all HTTP traffic to the HTTPS protocol.



Current Forwards					
Host	Destination	Type	Include Path	Wildcard	
http://www.beautiful-noyce.cloud	https://www.beautiful-noyce.cloud	temporary	no	yes	
www.beautiful-noyce.cloud	https://www.beautiful-noyce.cloud	temporary	no	yes	

Fig. 56: Domain name is configured with URL forwarding

On the Nginx server, with the use of Certbot (LetsEncrypt), a trusted certificate authority, we are able to issue SSL/TLS certificates to our registered domains. These certificates assure users that they are indeed connecting to a legitimate and secure web application server. It also enables the encryption of data in transit.

```
listen 443 ssl; # managed by Certbot
ssl_certificate /etc/letsencrypt/live/www.beautiful-noyce.cloud/fullchain.pem; # managed by Certbot
ssl_certificate_key /etc/letsencrypt/live/www.beautiful-noyce.cloud/privkey.pem; # managed by Certbot
include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
```

Fig. 57: SSL/TLS certificate configuration generated within site configuration by Certbot

Security headers are HTTP response headers that provide an additional layer of security by controlling how web resources are accessed, displayed and interacted with. By ensuring secure communication, controlling content sources and reducing the risk of data exposure, these headers collectively contribute to data protection at various levels within web applications. Within the global Nginx configuration file, security headers are defined on both domains as follows:

Header	Description
Strict-Transport-Security (HSTS)	This header instructs the browser to only connect to the web application via HTTPS, preventing potential downgrade attacks.
Content-Security-Policy	This header defines rules that restrict which sources of content are considered safe to load and execute on a webpage. Configured to allow resources to be loaded only from ' www.beautiful-noyce.cloud ' and 'jenkins.beautiful-noyce.cloud'.
X-Frame-Options	Configured with "SAMEORIGIN", this header prevents web browsers from allowing the web application to be embedded in an 'iframe' on another domain, thereby protecting from clickjacking attacks.
X-Content-Type-Options	Configured with "nosniff", this header prevents the browser from attempting to guess the content type of files, reducing the risk of MIME-sniffing attacks.
Referrer-Policy	Configured with "strict-origin", this policy restricts the amount of information disclosed by the 'Referer' header, minimising the exposure of navigation information to other websites.
Permissions-Policy	This header defines which permissions are allowed or denied for the web application, which includes denying of geolocation information, synchronous XMLHttpRequest (XHR) usage and MIDI device access.

Application Secrets Management

Jenkins Credential Manager is used to securely store and manage sensitive information, such as SSH keys and credentials. The Jenkins Credentials Manager provides a centralised and encrypted repository for storing these secrets, and allows the capability to retrieve and use these stored secrets within code, such as in the Jenkinsfile.

Credentials						
T	P	Store	Domain	ID	Name	
		System	(global)	github	kch-chaihong/*****	
		System	(global)	docker-hub	khorkch/*****	
		System	(global)	ec2-server-key	student59	

Fig. 58: Secrets securely stored within Jenkins Credentials Manager

```

pipeline {
    agent any
    environment {
        DOCKERHUB_CREDENTIALS = credentials('docker-hub')
        SSH_CREDENTIALS = credentials('ec2-server-key')
    }
}

```

Fig. 59: Usage of stored credentials from Jenkins Credentials Manager in Jenkinsfile

2.6.1.9. C9 - Implement Security Logging and Monitoring

Nginx internal Server Logs

By default, the Error and Access logs generated from using Nginx proxy service would be stored in /var/log/nginx/access.log or /var/log/nginx/error.log.

Therefore, in order to set custom configuration suitable for our scenarios we would input custom settings into the logrotate configuration file, which can be found at /etc/logrotate.conf.

Note that the log settings for both now have a “daily” and “rotate 14”. “Daily” means that the logs are maintained every 24 hours to ensure the log size does not fill up the storage too fast and slow down the server. The “rotate 14” indicates that log files are switched out every 24 hours, occurring 14 times, resulting in log files being retained for a duration of 2 weeks. We do the same for the file at /etc/logrotate.d/nginx as well.

```

/var/log/nginx/access.log {
    daily
    rotate 14
    compress
    missingok
    notifempty
    create 0640 www-data adm
}

/var/log/nginx/error.log {
    daily
    rotate 14
    compress
    missingok
    notifempty
    create 0640 www-data adm
}

```

Fig. 60: /etc/logrotate.conf file settings

```

/var/log/nginx/*.log {
    daily
    missingok
    rotate 14
    compress
    delaycompress
    notifempty
    create 640 nginx adm
    sharedscripts
    postrotate
        if [ -f /var/run/nginx.pid ]; then
            kill -USR1 `cat /var/run/nginx.pid`
        fi
    endscript
}

```

Fig. 61: /etc/logrotate.d/nginx file settings

We can verify the setting with the command below :

```
$ logrotate -d /etc/logrotate.d/nginx
```

How the output of the settings would look like if it is successful:

```
rotating pattern: /var/log/nginx/*.log after 1 days (14 rotations)
empty log files are not rotated, old logs are removed
considering log /var/log/nginx/access.log
Now: 2023-11-05 06:44
Last rotated at 2023-11-03 00:00
log does not need rotating (log is empty)
considering log /var/log/nginx/error.log
Now: 2023-11-05 06:44
Last rotated at 2023-11-03 00:00
log does not need rotating (log is empty)
considering log /var/log/nginx/nginx-access.log
Now: 2023-11-05 06:44
Last rotated at 2023-10-21 00:00
log does not need rotating (log is empty)
not running postrotate script, since no logs were rotated
```

Fig. 62: Verifying nginx log configurations

Jenkins Docker Frontend Logs

Similar to nginx path, we would create a custom logger config file for logrotate to base it off.

This is a newly created file with the path of /etc/logrotate.d/jenkins-docker-container (Change to your container name) with the content seen below

```
/var/lib/docker/containers/*/*-json.log {
    rotate 2
    weekly
    missingok      # Don't produce an error if log file is missing
    notifempty     # Do not rotate the log file if it's empty
    compress       # Compress old log files
    create 0644 root root # Set permissions and ownership for newly created log files
}
```

Fig. 63: /etc/logrotate.d/jenkins-docker-container file settings

Similar to the Nginx, the logs are set to have a lifespan of 2 weeks before getting rotated out as old logs, making the timeline consistent.

2.6.1.10. C10 - Handle all Errors and Exceptions

Avoiding information leakage

The web application ensures that there are no information leakage vulnerabilities in the error handling process, which can unintentionally assist potential attackers. For instance, in scenarios where a user might input an existing username during the signup process, the web application responds with a generic error message: "Error signing up! Please try again!" This practice ensures that specific error details are not disclosed, preventing attackers from gaining insights.

```
<LoginLayout >
{isOpen ? (
    <div className="absolute top-0 maxw="md">
        <AlertItem
            status='error'
            message='Error signing up! Please try again!'
            onClose={closeAlert}
        />
    </div>
}
```

Fig. 64: Generic error message to avoid divulging information to potential attacker

Centralised exception handling

The code avoids duplicated try/catch blocks and instead uses a single try/catch block to manage exceptions in a centralised manner. This is considered a best practice as it simplifies debugging and ensures that errors are consistent across the web application, and enables the capture of exceptions/errors resulting from various outcomes for proper handling. For example, in JWT access token verification, the operation is wrapped within one try/catch block. If the verification is unsuccessful or if certain conditions are not met, it catches the error and handles it by responding with the appropriate error messages and status codes.

```

try{
  const decodedToken = verifyJwt(accesstoken);

  if(!accesstoken || !decodedToken){
    console.log("verify unsuccess")
    return new NextResponse(JSON.stringify({
      error: "unauthorized",
    })),
    {
      status: 401,
    }
  }
}
}

return new NextResponse(JSON.stringify({
  error: "error request",
})),
{
  status: 400,
}
} catch (error){
  return new NextResponse(JSON.stringify({
    error: "unauthorized",
  })),
  {
    status: 401,
  }
}
}

```

Fig. 65: Catching and handling of exceptions/errors within a single try/catch block

3. Test Automation

3.1. Dependency-Check

Inventory of dependency check

The dependencies in the project are all the npm packages and files in the projects.

AlertItem.js	postcss.config.js	"dependencies": {
CartItem.js	prisma.js	"@chakra-ui/next-js": "^2.1.5",
CartOrderSummary.js	providers.js	"@chakra-ui/react": "^2.8.0",
CartProduct.js	route.js	"@emotion/react": "^11.11.1",
CategoryCard.js	route.js	"@emotion/styled": "^11.11.0",
CheckoutForm.js	route.js	"@prisma/client": "^5.2.0",
LoginForm.js	route.js	"@stripe/react-stripe-js": "^2.3.0",
LoginLayout.js	route.js	"@stripe/stripe-js": "^2.1.6",
MenuCard.js	route.js	"autoprefixer": "10.4.15",
MenuGrid.js	route.js	"axios": "^1.5.0",
Navigation.js	route.js	"bcrypt": "^5.1.1",
Order.js	route.js	"dotenv": "^16.3.1",
OrderCard.js	route.js	"es6-promise": "^4.2.8",
QuantityPicker.js	route.js	"eslint": "8.48.0",
axios.js	route.js	"eslint-config-next": "13.3.4",
helpers.js	route.js	"express": "^4.18.2",
jwt.js	route.js	"framer-motion": "^10.16.4",
layout.js	route.js	"isomorphic-dompurify": "^1.9.0",
middleware.js	route.js	"isomorphic-fetch": "^3.0.0",
next.config.js	route.js	"jsonwebtoken": "^9.0.2",
page.js	route.js	"log4js": "^6.9.1",
page.js	route.js	"next": "^13.3.4",
page.js	route.js	"next-auth": "^4.23.1",
page.js	route.js	"postcss": "8.4.29",
page.js	route.js	"react": "18.2.0",
page.js	route.js	"react-aws-s3": "^1.5.0",
page.js	route.js	"react-dom": "18.2.0",
page.js	route.js	"react-google-recaptcha": "^3.1.0",
page.js	route.js	"react-icons": "^4.11.0",
postcss.config.js	tailwind.config.js	"requests": "^0.3.0",
prisma.js	useAxiosAuth.js	"stripe": "^13.7.0",
providers.js	useRefreshToken.js	"tailwindcss": "3.3.3",
route.js		"winston": "^3.11.0"
route.js		}

Fig. 66: Project dependencies list

The team has decided to choose OWASP Dependency Check and 'npm audit' to detect the publicly disclosed vulnerabilities contained within the project's dependencies.

<div> Project: food-ordering-pipeline #140 </div> <div> Scan Information (show all): <ul style="list-style-type: none"> • <i>dependency-check version:</i> 8.4.2 • <i>Report Generated On:</i> Sat, 4 Nov 2023 20:19:05 GMT • <i>Dependencies Scanned:</i> 55 (55 unique) • <i>Vulnerable Dependencies:</i> 0 • <i>Vulnerabilities Found:</i> 0 • <i>Vulnerabilities Suppressed:</i> 0 • ... • <i>NVD CVE Checked:</i> 2023-11-04T19:34:42 • <i>NVD CVE Modified:</i> 2023-11-04T18:00:01 • <i>VersionCheckOn:</i> 2023-11-02T15:25:00 • <i>kev.checked:</i> 1699126518 </div>	<div> 5 vulnerabilities (1 low, 2 moderate, 2 critical) </div>
Fig. 67: OWASP Dependency Check	Fig. 68: npm dependency check

3.2. End-to-end (e2e) Testing

The team has implemented end-to-end testing using the Cypress framework to simulate the real user interaction on the browser. Testing has been done on three pages which is the payment page, login page and main page. The testing stage has been integrated into the CI/CD pipeline for test automation.

	Declarative: Checkout SCM	Build	Login	Push	Test	OWASP DependencyCheck	E2E Test	Deploy to Production	Declarative: Post Actions
Average stage times: (Average full run time: ~4min 44s)	1s	1min 42s	2s	30s	32ms	17s	2min 3s	3s	334ms
Nov 05 04:16 1 commit	1s	1min 43s	2s	30s	33ms	18s	2min 2s	3s	333ms
Nov 05 04:04 1 commit	1s	1min 40s	2s	30s	32ms	17s	2min 4s	3s	335ms

Fig. 69: Automated Testing

Spec	Tests	Passing	Failing	Pending	Skipped
✓ app.cy.js	00:15	3	3	-	-
✓ loginCustomer.cy.js	00:07	3	3	-	-
✓ loginSeller.cy.js	00:06	3	3	-	-
✓ payment.cy.js	01:02	3	3	-	-
✓ All specs passed!	01:30	12	12	-	-

Fig. 70: Automated Testing Result