National Institute of Applied Sciences and Technology

CARTHAGE UNIVERSITY

DevSecOps Project

Specialty : **Software Engineering**

Practical SSDLC

Presented by

Racem BENRHAYEM

INSAT Supervisor : **M. KARMOUS Mohamed Aymen**

Presented on : **01/06/2023**

Academic Year : 2022/2023

Table of Contents

List of Figures

General Introduction

Secure software development life cycle (SDLC) is crucial because application security has become a paramount concern. Gone are the days when releasing a product and addressing bugs later was sufficient. Developers now need to proactively address security at every step of the development process. Integrating security into the SDLC is essential to protect against potential vulnerabilities. With source code accessibility, coding with security in mind is imperative. Establishing a robust and secure SDLC process is critical to safeguarding applications from attacks by hackers and malicious users.

This report presents the comprehensive work undertaken for the development of our blog website, with a specific focus on the integration of Secure Software Development Life Cycle (SSDLC) principles. Our primary objective was to create a secure and reliable platform that offers engaging content while prioritizing the protection of user information and interactions.

Throughout the project, we diligently applied SSDLC principles at every stage of development. We initiated the process by incorporating secure design and architecture considerations, identifying potential threats and vulnerabilities, and implementing robust security measures. This included the implementation of strong authentication mechanisms, secure data storage practices, and measures to defend against common attacks like cross-site scripting (XSS) and SQL injection.

During the coding and development phase, our team diligently followed secure coding practices to minimize the introduction of vulnerabilities. We prioritized proper input validation, output encoding, and secure handling of user-generated content. Regular security testing, such as vulnerability scanning, was conducted to identify and rectify any potential security weaknesses.

To enforce stringent access controls and protect against unauthorized access, we implemented granular user account management and meticulous administrative privilege protocols. Additionally, we ensured the timely application of security patches and updates to maintain the website's resilience against emerging threats.

By adhering to SSDLC principles, we successfully created a blog website that not only offers captivating content but also prioritizes the security and privacy of our users. This report provides an overview of the meticulous steps taken to achieve a reliable and secure platform where users can confidently engage, interact, and contribute to the dynamic world of blogging.

Chapter I

Secure Requirement Gathering , Architecture and Design

Summary

Introduction

In this chapter, we start with the first step in SSDLC process which is secure requirement gathering, analysis. It is a security practice that refers to functional and non-functional requirements that need to be satisfied to protect the web application. After that, we move to the second step which is secure architecture and design that should be applied to mitigate the risks of any successful cyber attack coming from any potential threat in a poorly designed architecture. These two steps needs a strong understanding of cyber security.

1 Secure Requirement Gathering

1.1 Goals, Strategies and Metrics

Our goal is to protect the web application against web attacks. For that we have two main strategies :
First, educating the team about secure coding best practices to fix the vulnerabilities during the development process. We defined the number of critical vulnerabilities discovered using a SAST application as a metric.
Second, fixing the vulnerabilities associated with the application's dependencies when the vulnerability gets discovered. The metric in this strategy is the time spent from discovering the vulnerability until fixing it.

1.2 Understanding the policies

The main policy is to apply security controls on data. To reach that we defined two standards : Authentication and Authorization should be well implemented to control posts and comments adding and/or editing on the website , Applying a strict input validation to prevent XSS and injection attacks. We used step-by-step OWASP Cheat Sheet Series to ensure data integrity and safety.

2 Secure Architecture and Design

2.1 Threat Tree Diagram

This model is used to analyze and understand the attack surface of a system and the potential attack paths that an attacker could take to compromise the system.
The first Threat that comes to the mind is that an attacker may modify or delete another user posts or comments as described in Figure I.1.
The second threat is that an attacker could inject malicious content when posting a post or a comment which may lead to an XSS or injection attack. With the capability of adding images to a post , he could send large files to shutdown the system (DDos attack) as shown in Figure I.2.

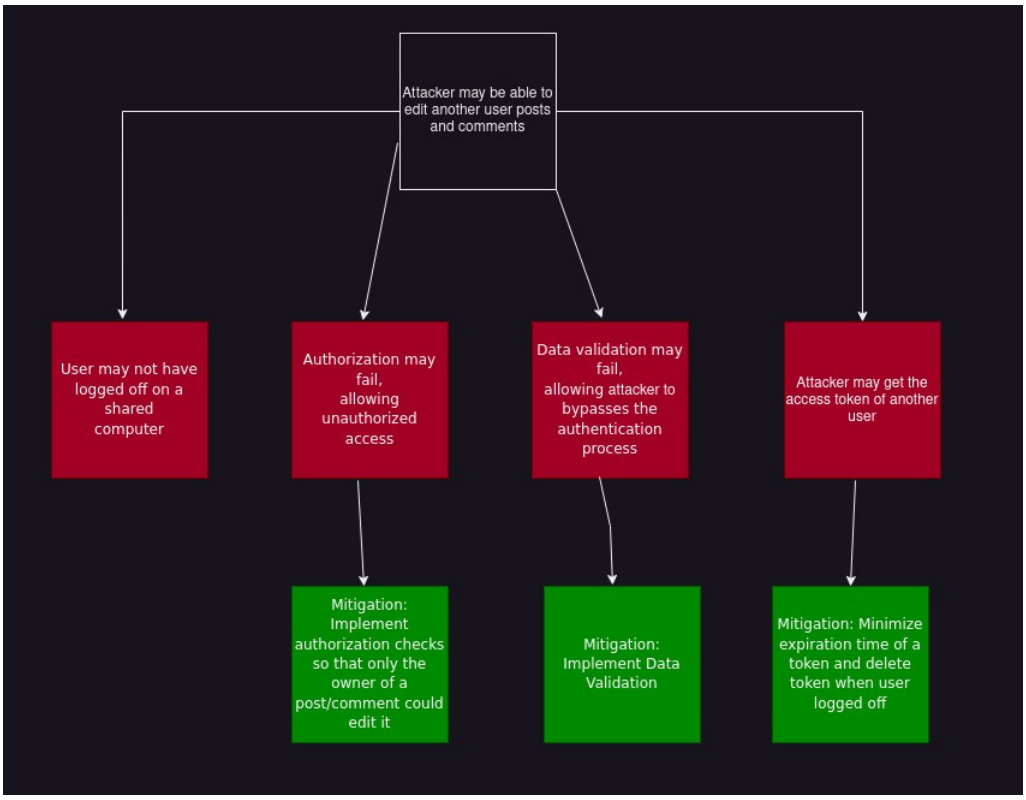


Figure I.1 – Threat Tree Diagram -Authorization

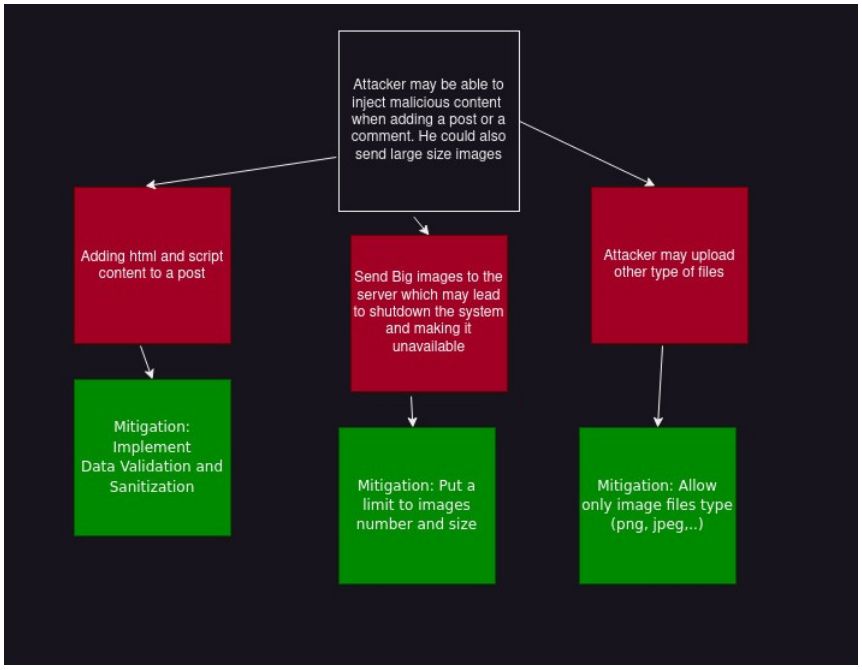


Figure I.2 – Threat Tree Diagram -Injection

2.2 Data Flow Diagram

A data flow diagram (DFD) maps out the flow of information for any process or system. The Figure I.3 models the System and shows data inputs, outputs, storage points and the routes between each destination.

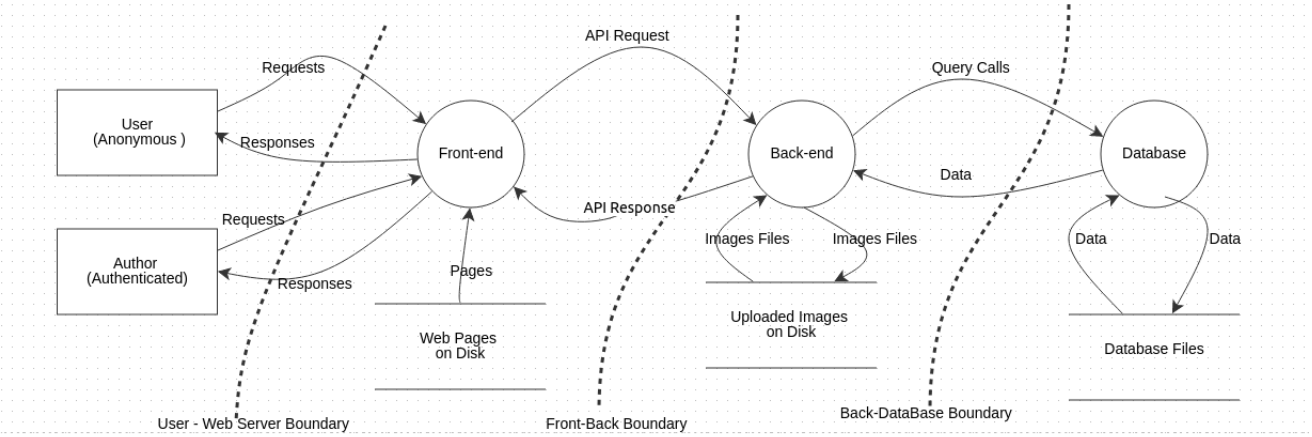


Figure I.3 – Data Flow Diagram

2.3 Applying STRIDE

STRIDE is a threat model that considers potential threats based on: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege.
STRIDE can be applied in conjunction with a Data Flow Diagram to identify and analyze potential security threats in a system.

- Spoofing: A poor authentication mechanism and a poor management of tokens may cause an attacker to spoof or impersonate the user to gain unauthorized access to the system where he can modify or delete posts and comments. That's why we reduced tokens validity.
- Tampering: An attacker could potentially modify content of the database files or uploaded images to inject malicious data or bypass access control. A possible cause of such a threat is a poor validation of data. For that, A validation and sanitization process is implemented to prevent injection attacks. Also, encrypting the authentication requests and responses.
- Repudiation: An Attacker may perform a prohibited actions in the system that are difficult to trace. For that, we keep logs of all API calls and database queries for auditing purposes.

- Information Disclosure: An attacker could potentially modify queries to extract sensitive data (such as passwords and emails) from the database. To remediate such a threat, we encrypted sensitive data in the database and we made sure that no sensitive data transits via API responses (only return what is necessary).
- Denial of Service: An attacker could overwhelm the Front-end with login requests to deny access to legitimate users or cause the system to crash. Also, he could overwhelm the back-end with large Files when uploading images. A possible solution is to implement rate limiting to prevent excessive login attempts from a single user, limit the number and size of files uploaded.
- Elevation of Privilege: An attacker could potentially exploit vulnerabilities to elevate their privileges and gain unauthorized access to the website, to the system or to the database. That's why a good input validation is required to prevent injection attacks.

3 Chosen Architecture and Technologies

The decision was made based on the comprehensive study described above.

3.1 General Architecture

Three-Tier architecture is opted for this system which is is a software architecture pattern that divides an application into three distinct layers:

- Presentation Layer (Front-end): it is Responsible for the user interface and presentation of information to the users. It handles user interactions, displays data, and collects input from users.
- Business Logic Layer (Back-end) : It contains the core business logic and rules of the application. It processes the user requests received from the presentation layer, performs necessary computations, and coordinates data access and manipulation.
- Data Storage Layer: It manages the interaction with databases or other data storage systems, ensuring data integrity, security, and efficiency.

Three-tier architecture offers several advantages for software development. It enhances scalability by allowing independent scaling of each layer, ensuring efficient resource allocation and accommodating increasing user demands. The layered structure promotes maintainability and

modularity, making the system easier to manage, modify, and enhance. Components and modules within each layer can be reused, reducing development time and effort while maintaining consistency. Security measures can be implemented at each layer, ensuring data integrity and mitigating risks.

3.2 Technologies Used

- **Angular**: A popular front-end framework, provides several security advantages for web application development. It automatically sanitizes user input, guarding against cross-site scripting (XSS) attacks. Angular also includes robust mechanisms for handling authentication and authorization, enabling developers to implement secure user authentication and role-based access control. It provides features like route guards and HTTP interceptors to control access to sensitive routes and protect against unauthorized requests. Furthermore, Angular's use of a component-based architecture promotes encapsulation and separation of concerns, reducing the risk of security vulnerabilities through code isolation.

- **NestJs**: A powerful back-end framework that integrates well with popular security libraries, making it easier to implement robust security features. It benefits from the active open-source community, providing regular updates, bug fixes, and security patches to ensure the framework remains secure. Moreover, NestJS follows the principle of modularity, allowing the creation of reusable security modules that can be shared across different parts of the application, improving consistency and reducing the likelihood of security gaps.

- **MongoDB**: A NoSQL database offers several security advantages for data storage and management. It is designed to mitigate traditional SQL injection attacks. It also supports field-level encryption, enabling sensitive data to be encrypted at rest, providing an extra layer of protection.

Conclusion

Together, requirement gathering and architecture designing enable effective planning and decision-making throughout the software development life-cycle. Making those steps secure is crucial to detect potential vulnerabilities and address them early, reducing the risk of malicious attacks and data breaches. Moreover, the three-tier architecture provides a robust framework for building scalable, maintainable, and secure software systems using the appropriate technologies.

Chapter II

Secure Development

Summary

Introduction

This phase includes the actual engineering and writing of the application while attempting to meet all the requirements established during the secure requirement gathering, analysis and planning phase.

1 Following Secure Coding Best Practises

In this section, we will mention the secure coding best practises that we used during the development of the project. We used the recommendations provided in The OWASP Top Ten cheat sheet.

1.1 Performing Input Validation

Input validation is crucial for securing web applications because it helps prevent various types of malicious attacks and vulnerabilities that are already mentioned including SQL injection and XSS attacks. To effectively implement input validation, it's essential to combine server-side validation and client-side validation. Server-side validation is critical as it cannot be bypassed by attackers, while client-side validation helps provide instant feedback to users.

- **Server-Side:** NestJs comes with a plenty of concepts that help developers validate the users' input. Let's start with the combination of Data Transfer Object (DTO) with the recommended packages class-validator and class-transformer. A DTO is an object that is used to transfer data between different layers of an application and provide a standardized format for communication. The class-validator library provides decorators that you can apply to the properties of your DTO classes to define validation rules. Furthermore, Nestjs provide a really powerful ready-to-use validation pipes. Another step of validation is sanitizing HTML input using sanitize-html library. This II.1 shows an example of validating users' content when creating a new account.

Listing II.1 – Input Validation Example

```
  import {
  IsEmail,
  IsNotEmpty,
  IsString,
  MaxLength,
  MinLength,
} from 'class-validator';

export class CreateUserDto {
  @IsNotEmpty()
  @IsString()
  username: string;

  @IsNotEmpty()
  @IsString()
```

```
  @IsEmail()
  email: string;

  @IsNotEmpty()
  @IsString()
  @MinLength(8)
  @MaxLength(16)
  password: string;
}
```

- **Client-Side:** In Angular, we used the built-in form validators to perform client-side validation on forms. These validators help ensure that the data entered by the user meets specific criteria before submitting the form. Adding to that, We implemented a pipe that benefits from Angular built-in DomSanitizer service that sanitize and mark HTML content as safe.

Listing II.2 – Implemented Pipe

```
import { Pipe, PipeTransform, SecurityContext } from '@angular/core';
import { DomSanitizer } from '@angular/platform-browser';

@Pipe({
  name: 'safeHTML',
})

export class SafeHTMLPipe implements PipeTransform {
  constructor(private sanitized : DomSanitizer){}
  transform(value: string) {
    return this.sanitized.sanitize(SecurityContext.HTML, value);
  }
}
```

1.2 File Upload Validation

Our website comes with the feature that a user could add images to his posts. For that, it is primordial that we implement upload verification.
To begin with, we ensured that the uploaded filename uses an expected extension type. In our case, the expected files are images so we allowed only png, jpg, jpeg and gif extensions. In addition, we limited ,not only, the number of allowed images per post to five, but also the file

size limit to 8 megabytes (this is the recommended value by SonarQube That is presented in this section 1.3).

1.3 Upload Storage

As it is recommended to generate new and unique names to the files that are uploaded and the file path should be decided by server side. The purpose of doing it is to prevent the risks of direct file access and ambiguous filename.

1.4 Prevent HTTP Parameter Pollution

In a normal HTTP request, parameters are sent as key-value pairs in the query string or request body. Each parameter has a unique name and value, allowing the application to process the data accurately. However, the attacker intentionally includes multiple parameters with the same name but different values which may lead to unpredictable interpretation of these parameters by the targeted application. This is what called a HTTP Parameter Pollution (HPP) attack. To mitigate to such a threat, we used the hpp module that will ignore all values submitted for a parameter and take only the last value submitted.

1.5 Only Return What Is Necessary

When handling user data in an application, it's crucial to prioritize data security and privacy. User records typically contain sensitive information such as IDs, usernames, email addresses and passwords . To mitigate the risk of personal information disclosure, it's important to limit the retrieval and usage of user objects to only the necessary fields. We returned only specific fields required when querying the database.

Listing II.3 – Do not return password field

```
const user = await this.userModel
      .findOne({ _id: id })
      .select({ password: 0 })
      .populate({
        path: 'posts',
      })
      .populate({
        path: 'comments',
        populate: {
          path: 'post',
          select: 'title',
```

```
    },
})
.exec();
```

1.6 Protecting Sensitive Data

It is crucial to store passwords in a way that prevents them from being obtained by an attacker even if the application or the database is compromised. For that, we opted for password hashing using the bcrypt hashing function.

1.7 Tokens Management

We used JWT as the format for security tokens. JWT, stands for Json Web Token, are JSON data structures containing a set of claims that can be used for access control decisions. Also, we used a secret key to sign the token and ensure its integrity. The secret key is a secret known only to the server that issues and verifies JWTs and it is kept confidential. In addition, we minimized the validity of tokens by specifying the token expiration time. It is important to note that JWT authentication mechanism is stateless, meaning that the server does not need to store any session information or maintain any state about the authenticated user on the server-side. After verifying the token, the server can extract information contained in it. Then, this extracted information is passed to NestJs guards allowing to implement authorization rules and authentication checks to protect specific endpoints or resources.

1.8 Cross-Origin Resource Sharing

CORS (Cross-Origin Resource Sharing) is a W3C standard that allows web applications to specify which domains or origins are permitted to make JavaScript requests to a REST API. By including appropriate CORS headers in the API responses, the server informs the browser about the allowed domains for making cross-domain requests.
In our case, we only allowed the domain where our front-end is deployed.

1.9 Monitoring the event loop

Under heavy network traffic, an application server may become overwhelmed and unable to effectively serve its users. This situation is akin to a Denial of Service (DoS) attack, where the server is rendered unavailable to legitimate users due to the excessive demand on its resources. To mitigate this threat, we used toobusy-js module that allows to monitor the event loop. It

keeps track of the response time, and when it goes beyond a certain threshold, this module can indicate that the server is too busy. In that case, we stopped processing incoming requests and send 503 Server Too Busy message so that our application stay responsive.

1.10 Taking precautions against brute-forcing

As we discussed earlier, in the previous chapter, that a possible threat is overwhelming the system with requests causing it to crash. Consequently, it is crucial for application developers, especially on login pages, to implement measures to mitigate brute-force attacks. NestJs comes with a ready module named ThrottlerModule and we configured it to block an IP address from sending requests if it exceeded ten requests in less than sixty seconds.

1.11 Exception Handling

We tried to cover a good number of exceptions and returning a simple, comprehensible responses with the appropriate code status without revealing any sensitive information. In addition, NestJs comes with a built-in exceptions layer which is responsible for processing all unhandled exceptions across an application. When an exception is not handled by our application code, it is caught by this layer, which then automatically sends an appropriate user-friendly response.

1.12 Logging

NestJS has a built-in text-based logger you can use without needing to install any additional packages. It logs really useful information and errors in controllers and providers. Also, it logs all HTTP requests that hit our server. Good and accurate logs about a user's activity can alert us about malicious actions and when an error happened which are crucial for auditing purposes.

1.13 Database Access

Our database is deployed on MongoDB Cloud. We used the free plan that is more than enough to handle this project. Absolutely, there are many benefits for deploying the database on the cloud. We can mention some of them:

- **Scalability:** MongoDB Cloud provides flexible scalability options, allowing us to easily scale our database infrastructure as our application grows.

- **Managed Service:** MongoDB Cloud is a fully managed service, which means that the operational aspects of database management, such as hardware provisioning, software patching, backups, and monitoring, are taken care of by MongoDB.
- **Security:** MongoDB Cloud includes built-in access controls, authentication mechanisms, and auditing capabilities to help us implement robust security practices.

One of the security techniques in MongoDB cloud is limiting access to the database to only allowed IP addresses. It is clear that we whitelisted only the server IP address that runs our API.

2 Secure Build

Absolutely, our project uses third-party libraries and dependencies to benefit from ready-to-use features without reinventing the wheel. Our policy consists of only using trusted and verified libraries and dependencies that are not vulnerable to known security issues. In this section we will mention the techniques we used to maintain the previous policy.

2.1 Auditing Dependencies

Our project is written with Typescript in nodejs environment. The package manager for our project is NPM. NPM stands for "Node Package Manager" and it is used to install, update, and uninstall packages and their dependencies.
NPM is able to audit the third-party libraries mentioned in package.json file and can fix the known vulnerabilities if there is a known fix. In our project, we have two "package.json" files: one for the Front-end and the other for the Back-end. Let's start auditing the Front-end dependencies as shown in the Figure II.1.
The result of the audit shows that there is 1 high severity vulnerability. Luckily, it comes with a fix available so we did it as mentioned in the Figure II.2
Let's repeat the same process for the Back-end dependencies. The Figure II.3 show that there is no severe vulnerability in packages installed.

2.2 Software Composition Analysis

So, it is important to review third-party libraries and components to ensure that they are up-to-date, secure, and used appropriately. Yet, this is an exhausting task. A possible solution is to use a Software Composition Analysis (SCA) scanner that notify us for any recent vulnerability

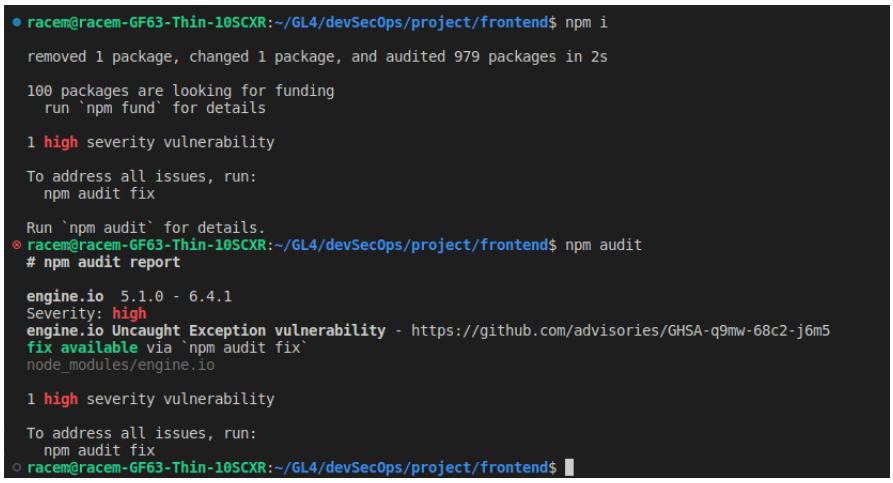


Figure II.1 – Front-end dependencies audit

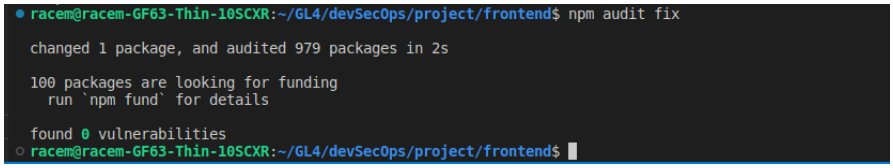


Figure II.2 – Front-end dependencies fix

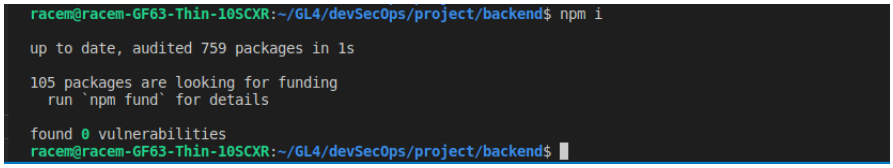


Figure II.3 – Back-end dependencies audit

affecting the actual libraries used in the project and for any new dependencies' versions. In this project, we integrated two commonly used SCA scanners which are Dependabot and Snyk.

2.2.1 Dependabot

Dependabot is a software development tool that automates the process of keeping project dependencies up to date. It continuously monitor them for any available updates, security patches, or bug fixes. When updates are detected, Dependabot can automatically create pull requests or issue notifications, depending on the configured settings. The Figure II.4 shows some of the pull requests that were created by Dependabot and the Figure II.5 shows and example of an email sent to notify us about updates.

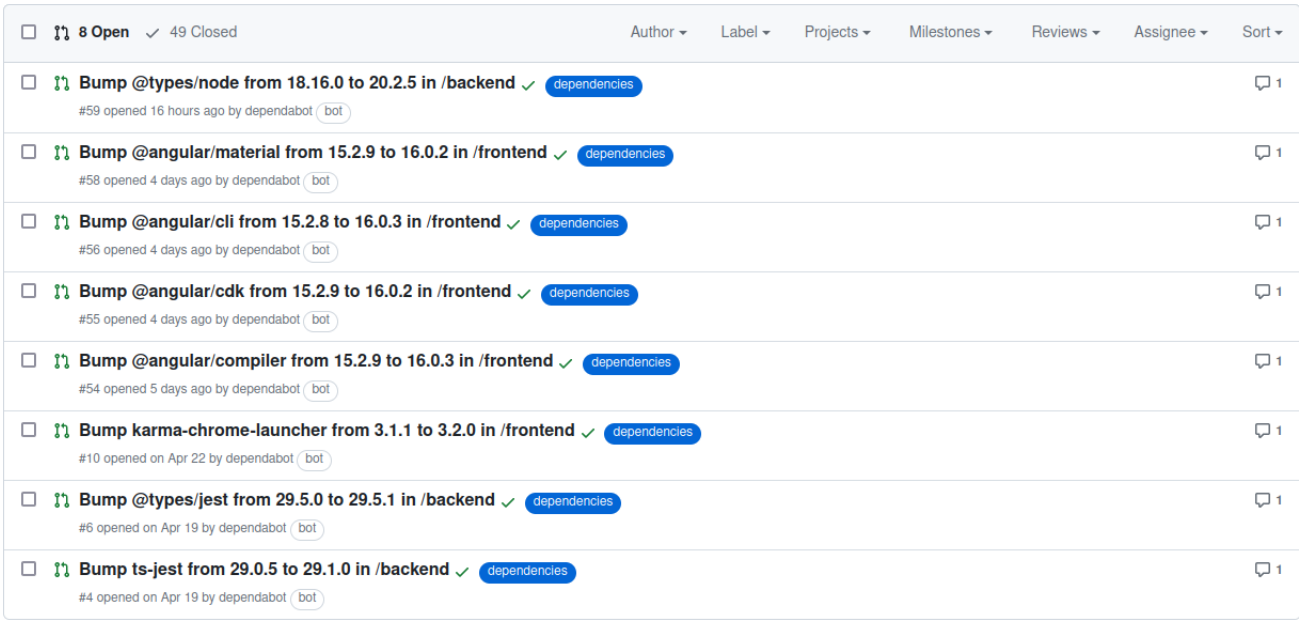


Figure II.4 – Dependabot pull requests

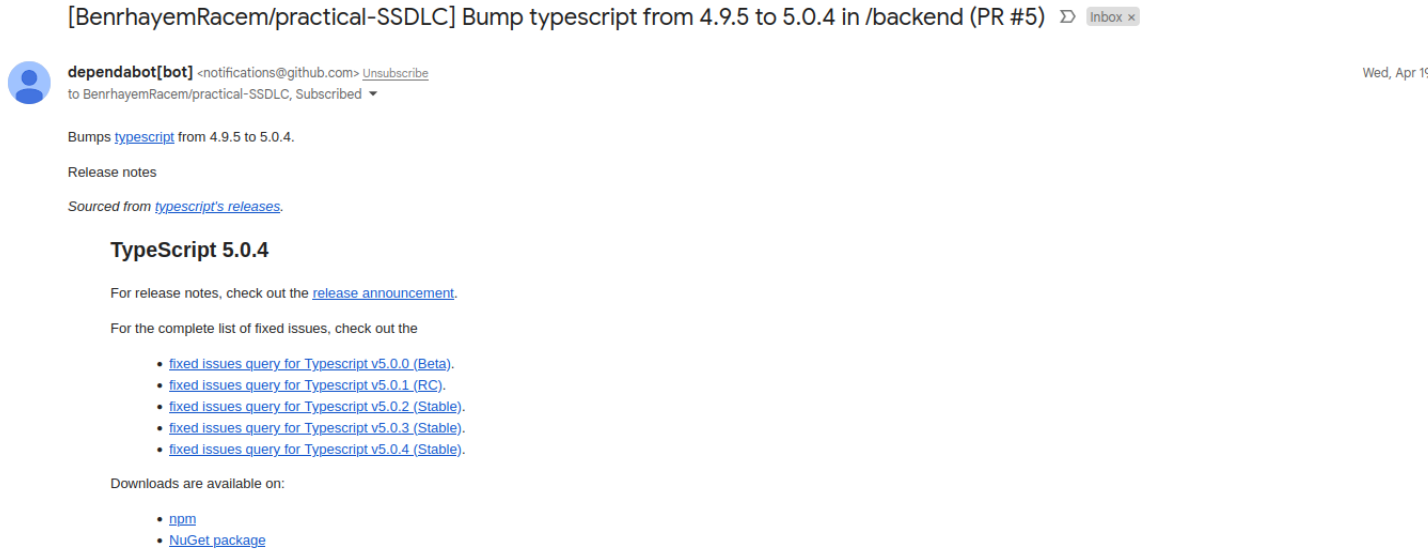


Figure II.5 – Dependabot notification

2.2.2 Snyk

Snyk is a developer security platform. Integrating directly into development tools, workflows, and automation pipelines, Snyk makes it easy for teams to find, prioritize, and fix security vulnerabilities in code, dependencies, containers, and infrastructure as code. Supported by industry-leading application and security intelligence, Snyk puts security expertise in any de-

veloper's toolkit.
We added our project to Snyk platform to allow him to create pull requests when a security patch is available. Here is an example in the Figure II.6 of a merged pull request created by snyk-bot.

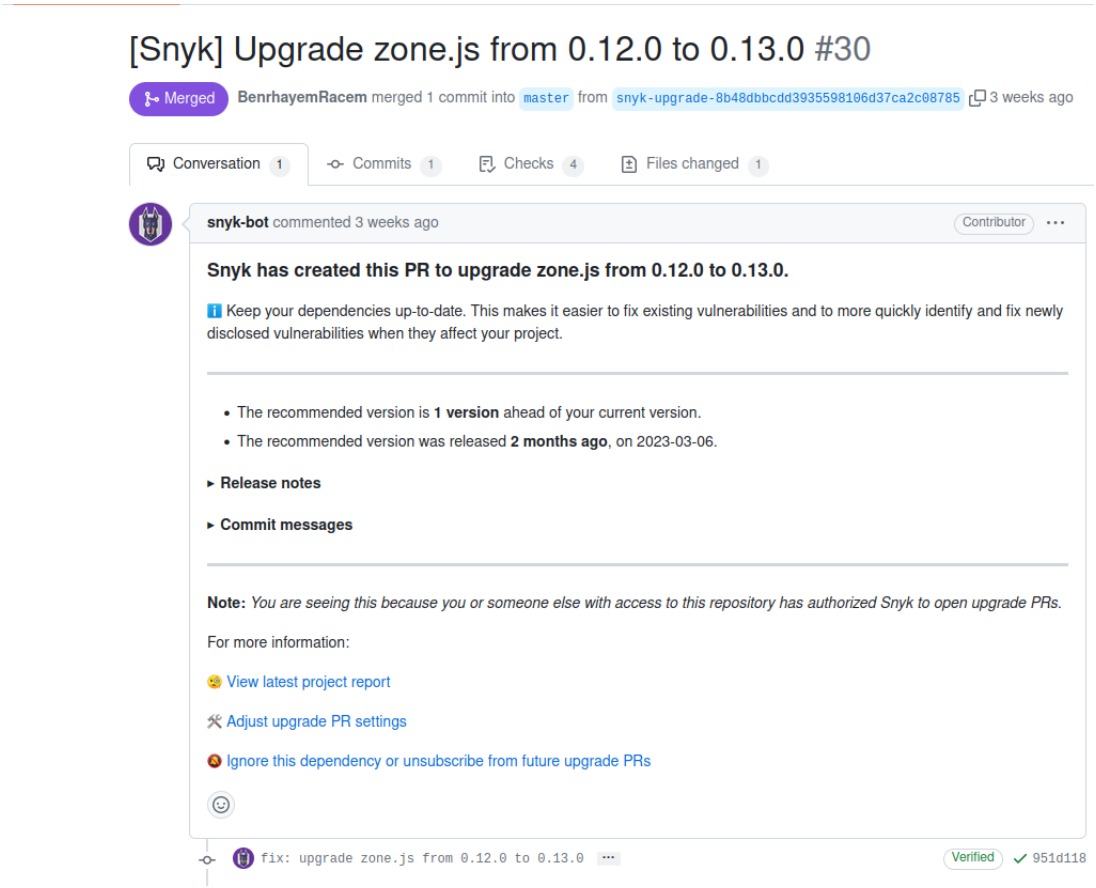


Figure II.6 – Snyk pull request

Conclusion

In software development, it is important to uphold coding best practices, adhere to security recommendations, and exclusively rely on trusted dependencies. Following these principles diligently guarantees the creation of a robust, secure, and reliable application, fortified against vulnerabilities and potential risks. Ultimately, by adhering to these practices, developers can ensure the development of high-quality software that delivers a reliable and secure experience to its users.

Chapter III

Secure Testing

Summary

Introduction

Secure testing refers to the practice of conducting software testing with a focus on identifying and addressing security vulnerabilities and risks. It involves evaluating the security posture of an application or system through various testing techniques to ensure that it is robust against potential attacks and adequately protects sensitive data. We configured all of our test tools with Github workflows so they are automatically triggered on push and on pull requests to the master branch.

1 Static Application Security Testing

Static Application Security Testing (SAST) is a type of security testing that focuses on analyzing the source code of an application to identify potential security vulnerabilities and weaknesses.

1.1 CodeQL analysis

CodeQL is a powerful static analysis engine developed by Semmle, a company acquired by GitHub. It allows developers to query codebases and find potential security vulnerabilities, bugs, or other code quality issues.

When scanning the source code, codeQL reports that there is a database query built from user controlled-sources in two different parts of our Back-end as described in the Figure III.1. Both issues marked with high severity potential vulnerability.
To resolve these two issues, we verified again our implementation and we marked them as a false alarm because we already implemented input validation and sanitization as we discussed earlier in section 1.1

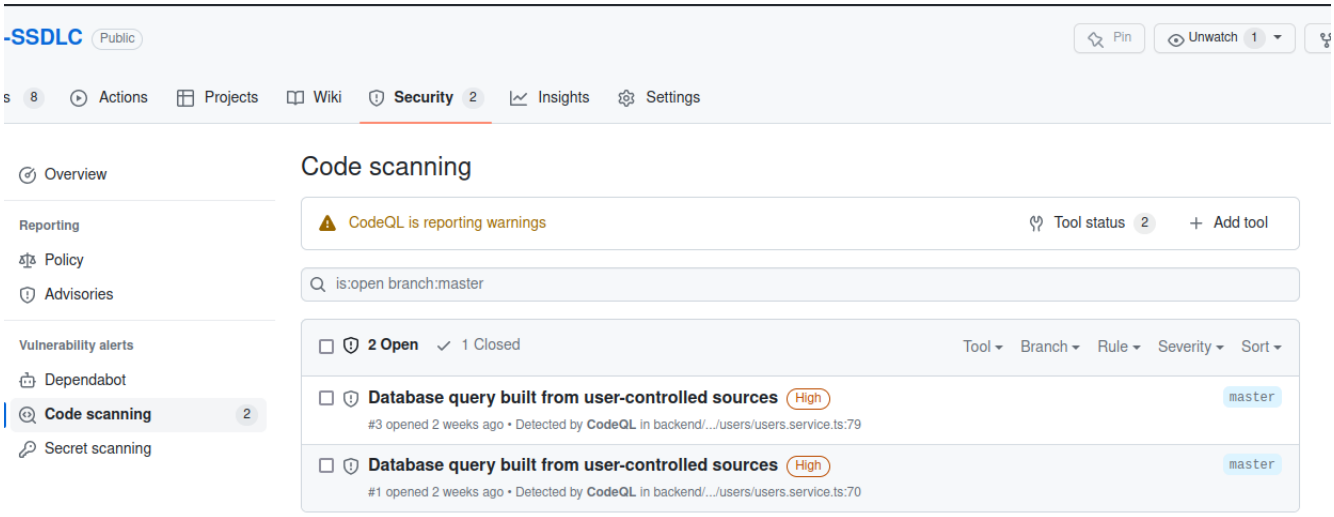


Figure III.1 – CodeQL analysis result

1.2 Github Secret Scanning

GitHub Secret Scanning is a security feature provided by GitHub to help detect and prevent the exposure of sensitive information, such as API keys, tokens, and passwords, within public and private repositories. It automatically scans repositories for known secret patterns and notifies repository administrators when potential secrets are identified.
In the project repository we enabled Github Secret Scanning and we have no sensitive information that was exposed or hard-coded in the source code.

1.3 SonarQube

SonarQube is an open-source platform that provides continuous code quality management and static code analysis. It helps developers and teams to track, manage, and improve the quality of their source code throughout the software development lifecycle.
In our project, we used the SonarQube Cloud that integrated directly with Github with no installation needed.

The First analysis of the code source as mentioned in the Figure III.2 shows a really good results. Our code has zero Duplication, Bugs and Vulnerabilities. However, if we look carefully,

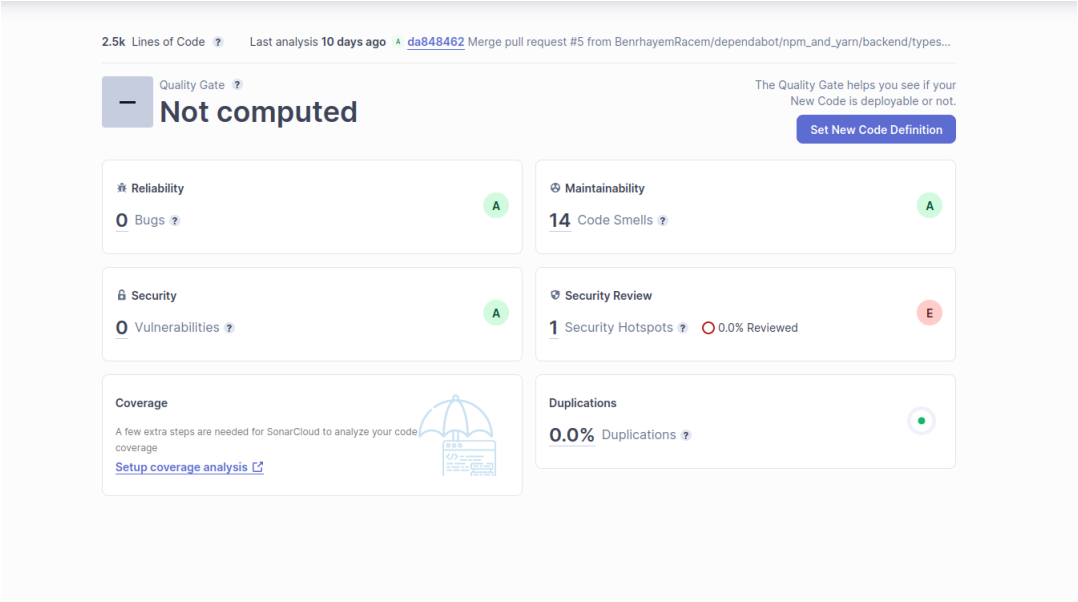


Figure III.2 – SonarQube analysis result

we notice that there is a critical security issue. We have to remember that in the section 1.2 we limited the file size in upload process, that solution was recommended by SonarQube. The figure III.3 describes the risk identified by this tool.

2 Software Composition Analysis

We already introduced the concept of the Software Composition Analysis in section 2.2. This time we used the SCA from another perspective to complete the secure testing phase.

2.1 Snyk

We had already defined what Snyk is and what it does in the following section 2.2.2.
This time we used Snyk in a different task. When opening a new pull request to the master branch, Snyk detects new dependencies introduced to the project and verifies that the new pull request does not contain any known vulnerable dependency. The Figure III.4 shows a pull request that successfully passed Snyk security tests.

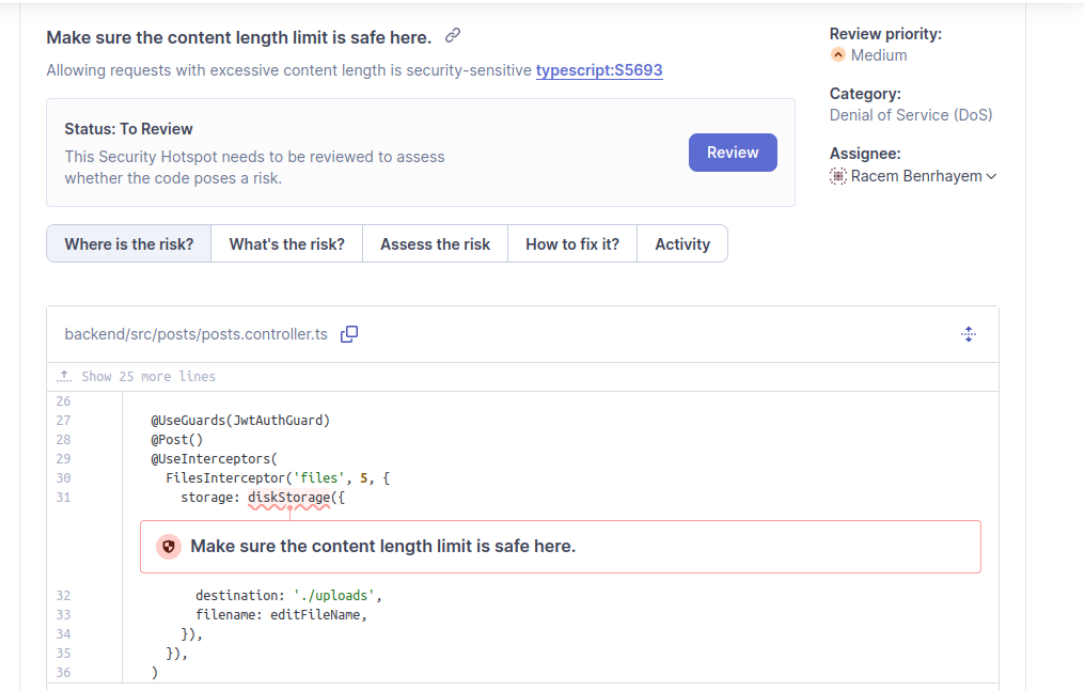


Figure III.3 – SonarQube identifying a potential risk and suggesting a possible solution

Conclusion

Secure testing is an essential component of the software development life-cycle, enabling developers and organizations to identify and address security weaknesses early in the development process. By conducting thorough security testing, vulnerabilities can be remediated, reducing the risk of successful attacks and ensuring the overall security and integrity of the software.

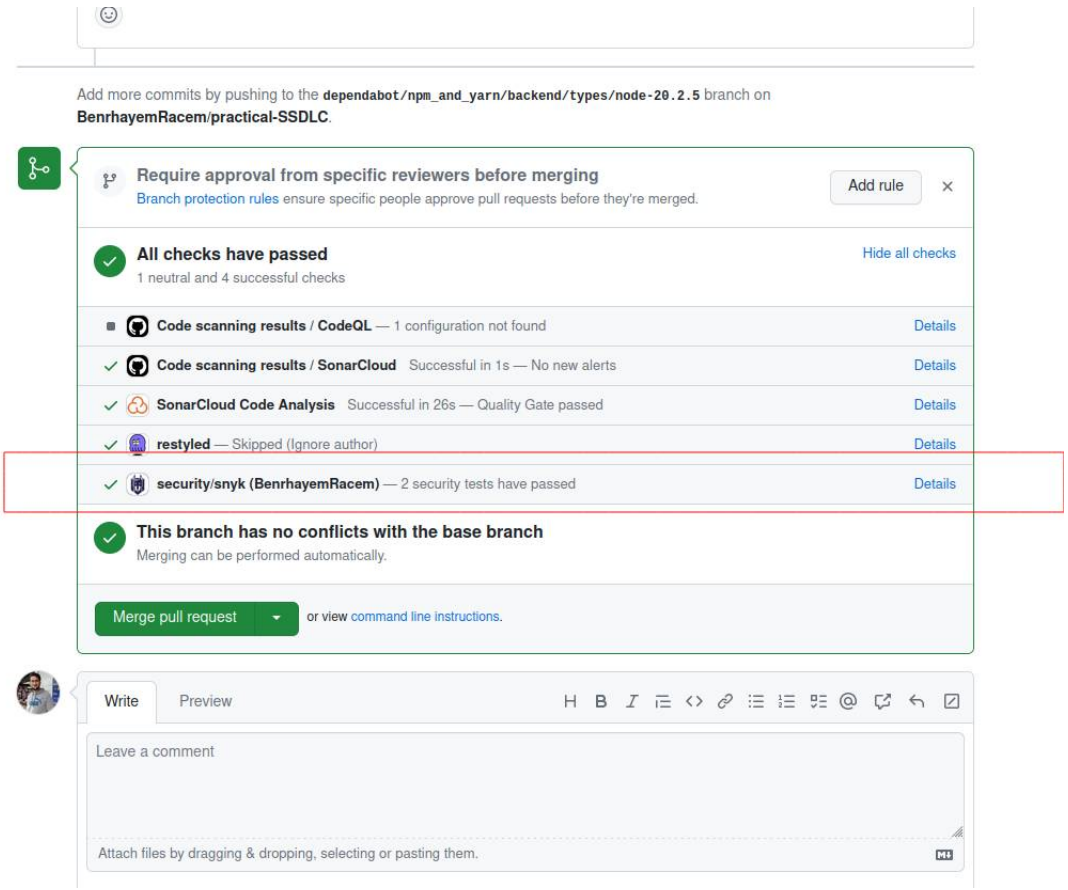


Figure III.4 – Snyk analyzing new dependencies

Chapter IV

Secure Release and Deployment

Summary

Introduction

In software development, deployment refers to the process of making a software application available for use. It involves taking the developed software and installing or configuring it on the target environment where it will be used by end-users. In this chapter we are talking about the steps we did to deploy our project.
Consideration: All following tasks were done locally.

1 Containerization

1.1 Containerization And Docker

Containerization is a method of packaging an application, its dependencies, and its configuration into a standardized, lightweight unit called a container. Docker is an open platform for developing, shipping, and running applications. It provides tools to manage the life-cycle of containers and it is one of the most widely used containerization platforms available today.

1.2 Creating Dockerfiles

Our project is divided into two different deployable components: the Front-end and the Back-end. We created a separate dockerfile for each of them. We need to keep in mind that this step should be done securely, for that we followed the OWASP recommendations about docker security. In addition, SonarQube also scanned these two dockerfiles and helped us fixing potential vulnerabilities.
In the following part we are going to enumerate some of them.

- **Copying recursively:** When creating a dockerfile, we had the habit to use the "**COPY . .**" command. This may put a risk as SonarQube mentions that copying recursively might inadvertently add sensitive data to the container as shown in the Figure IV.1 . To remediate to such a problem, we copied only necessary files to run our project. The Figure IV.2 shows an example of this solution.

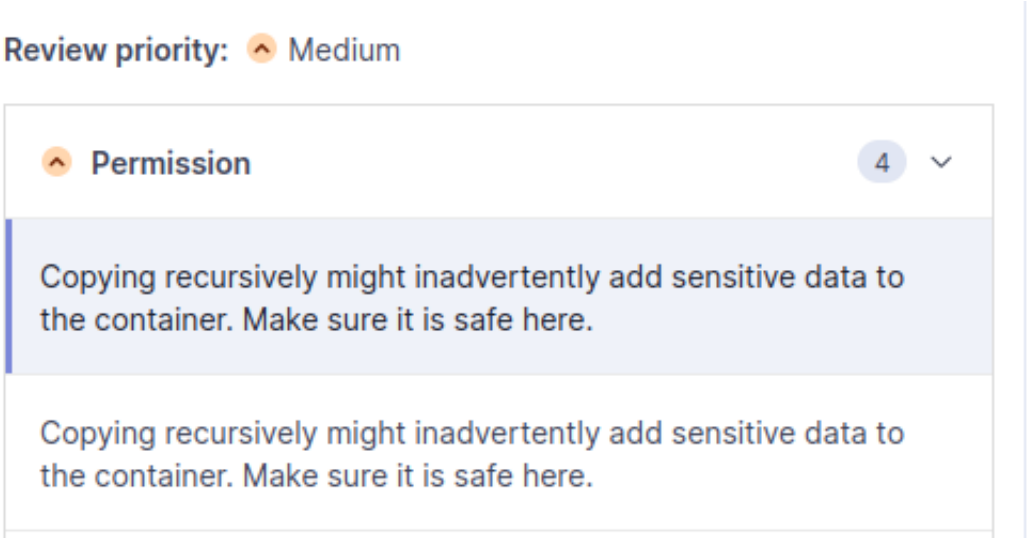
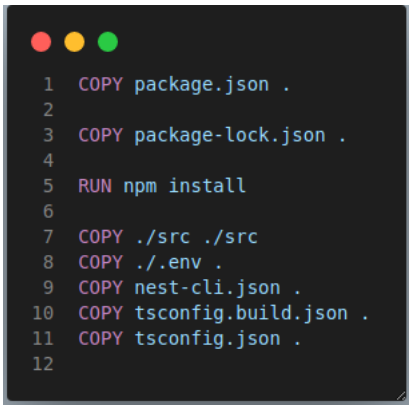


Figure IV.1 – SonarQube recursive copy warning

- **Execution of shell scripts:** In dockerfile, we usually run commands defined in package.json file. Those commands may contain malicious shell scripts. For that, we have to add "**–ignore-scripts**" flag to the command to make sure that no shell script is being executed with NPM commands. This is what SonarQube warned us about in the Figure IV.3



```
1  COPY package.json .
2
3  COPY package-lock.json .
4
5  RUN npm install
6
7  COPY ./src ./src
8  COPY ./.env .
9  COPY nest-cli.json .
10 COPY tsconfig.build.json .
11 COPY tsconfig.json .
12
```

Figure IV.2 – Copying only necessary files

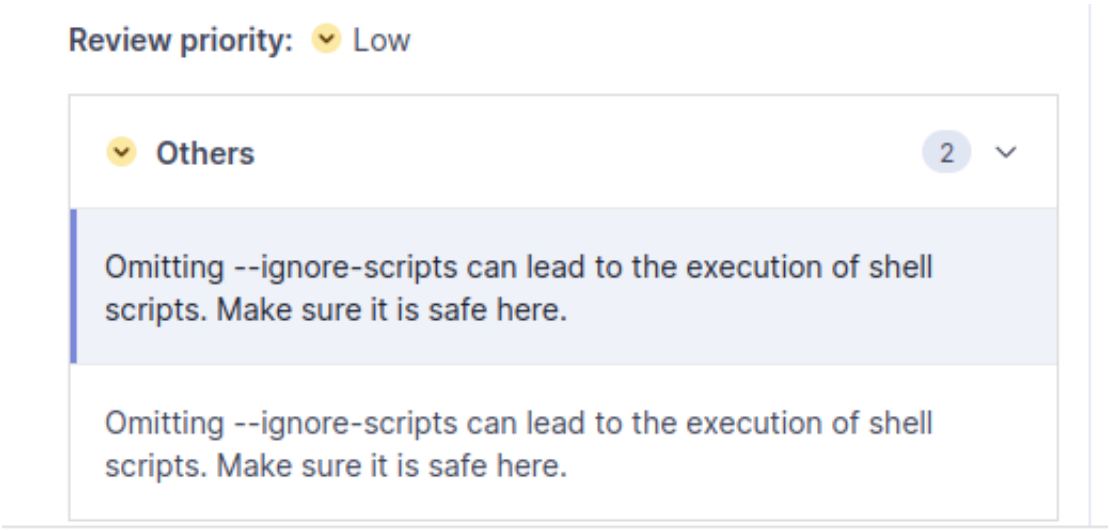


Figure IV.3 – SonarQube scripts warning

2 NGINX Web Server

We have a ubuntu machine where we are going to deploy our project. For now, we have two running containers. We need to setup a web server in front of them and we used NGINX for its beneficial key uses.

2.1 Key Uses and Advantages

Nginx is a popular web server that is known for its high performance, scalability, and versatility. It is commonly used as a reverse proxy server, load balancer, and HTTP cache.

- **Web server:** Nginx can serve static and dynamic content, making it suitable for host-

ing websites, web applications, and APIs. It efficiently handles concurrent connections, optimizing resource utilization and response times.

- **Reverse Proxy:** Nginx can act as an intermediary between clients and backend servers, distributing incoming requests and forwarding them to appropriate destinations. It provides load balancing and improves the overall performance and reliability of the system.
- **Load balancer:** Nginx's load balancing capabilities allow us to distribute incoming traffic across multiple backend servers, improving scalability and preventing any single server from being overwhelmed. It supports various load balancing algorithms and can automatically adapt to changing server conditions.
- **SSL/TLS termination:** Nginx can handle SSL/TLS encryption and decryption, offloading the CPU-intensive task from backend servers. It simplifies the configuration and management of SSL/TLS certificates, enhancing security and performance.

2.2 Nginx Serving Content

We used the Angular CLI to build our Angular application. This will generate a set of static files that can be served by Nginx. So this is how the Front-end docker image is built. It contains two stages: the first one for building the application and the second one is configuring Nginx to serve the static files to optimize the delivery of our application and provide a smooth and efficient user experience.

2.3 Nginx as a Reverse Proxy

We configure NGINX to route incoming requests to the Front-end container. To accomplish that, we edited the default server configuration to route incoming requests to the port where our Front-end is running.

Listing IV.1 – NGINX as a Reverse Proxy

```
location / {
        proxy_pass http://localhost:5000; #whatever port your app runs on
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
  }
```

2.4 HTTPS

HTTPS (Hypertext Transfer Protocol Secure) is the secure version of HTTP, the protocol used for transmitting data over the internet. It adds an extra layer of encryption and authentication to ensure secure communication between a web browser and a web server and this communication cannot be intercepted by unauthorized parties.
To ensure that our web server uses HTTPS we need to generate SSL/TLS certificates. To do that we used Certbot.
Certbot is a free and open-source software tool designed to automate the process of obtaining and managing SSL/TLS certificates for websites. It simplifies the process of enabling HTTPS encryption on web servers by providing an easy-to-use command-line interface.
Certbot integrates easily with Nginx helping us obtaining certificates in few commands. It comes also with auto-renewal feature.

3 Docker SWARM

Docker SWARM is a native clustering and orchestration solution provided by Docker. It uses two types of nodes: managers and workers. Docker SWARM provides built-in load balancing for services. Incoming traffic to a service is automatically distributed among the running containers on different worker nodes. In addition, it keeps a state of running containers and works on keeping that state.
For the simplicity, we used only one manager node which will play both roles at once: a manager and a worker. We configured the manager to run one replication of each service, Front-end and Back-end. When a container stoppes unexpectedly, the manager creates a new one to ensure that there is one container of each service running. As a result, we make sure that our application availability is at its highest level.

Conclusion

The deployment phase in the software life cycle is of paramount importance as it marks the release of the product to customers, ensuring customer satisfaction and validating development efforts. It provides an opportunity for real-world testing, gathering user feedback, and iterating on the software to improve its quality.

Conclusion and Perspectives

In conclusion, this report showcases the extensive implementation of Secure Software Development Life Cycle (SSDLC) principles in the development of our blog website. We have effectively integrated security considerations throughout the entire project, demonstrating our commitment to ensuring a secure and trustworthy platform. By comprehensively implementing SSDLC principles, we have achieved a blog website that not only offers engaging content but also instills confidence in its security. Users can confidently explore, interact, and contribute to the platform, knowing that their information is protected.

One notable limitation is the absence of an automatic deployment process. Currently, the deployment of the website requires manual intervention, which can be time-consuming and prone to human error. Automating the deployment process would streamline operations, enhance efficiency, and reduce the risk of inconsistencies during deployment. By automating the deployment, we can ensure a more seamless and reliable process, enabling quicker updates and enhancements while maintaining the high-security standards established throughout the development cycle. Implementing an automated deployment solution should be considered as a future enhancement to further enhance the efficiency and agility of the project.

Overall, this report highlights our dedication to delivering a secure and reliable blog website, demonstrating our commitment to user privacy and data protection. The successful implementation of SSDLC principles ensures that our platform meets high-security standards, creating a trusted environment for users to enjoy the content and engage with the blogging community.