Final Master's Thesis

Master's Degree in Cybersecurity

Securing a REST API Server

September 1, 2022

Thesis Supervisor :

Silvia Llorente Viejo

Barcelona School of Telecommunication Engineering

&

Computer Science Faculty of Barcelona

at

Polytechnic University of Catalonia

ABSTRACT

Nowadays, there are more sources of cyber-threats and more cyber-attacks that target all kind of victim profiles. From big companies with big architectures, to small businesses that only have a web site as a platform to sell or advertise themselves. Hence, the need of security awareness among users, developers and systems administrators, as well as the application of security measures and best practices, is mandatory. There are a lots of organisations promoting security recommendations or standardisation of procedures, in order to build more robust applications or infrastructures.

The purpose of this thesis is to provide a practical application of those recommendations, sets of best practices, design patterns and standards. This will serve as an example of a minimum standard to achieve regarding security when developing an application or service, as can be an e-commerce platform. This project implemented different security measures and basic infrastructure security designs and configurations in order to create a more robust application. This measures and designs will be considered for all components as well, not only the server. This includes the database, web server, firewall and the virtual machine where the project is hosted.

Contents

Introduction

The main goal of this thesis is to apply security protections and mechanisms to an e-commerce application, in order to provide a useful example of how to check and correct common vulnerabilities. An e-commerce is an example of a typical service we can find in the Internet, hence, the reason behind choosing it. The order of topics that we are going to talk about in this thesis is the following. First, we will introduce some technologies, technical words and ideas regarding the application and the architecture we are going to follow for the development. Furthermore, there will be examples of common methodologies in software development, explaining which have been applied to this project and why.

Afterwards, it will be discussed some examples of different organisations that work on creating and developing standards regarding cybersecurity, protocols, etc, and which references we took in order to tackle the most common vulnerabilities in a server application. To finalise that section, a big appreciation about the importance of these organisations in educating people and developers and how they encourage the community into following those standards and good practices will also be included.

After that, it will be explained in detail how the application will work, the different technologies involved, the functionalities available and all of the security measures implemented, regarding both the server and the other technology components. And last but not least, a summary of all the results and conclusions will be reported along a list of future improvements in order to provide more ideas or alternatives to be implemented in a application.

1 State of the art

In our project, an Application Program Interface (API) has been developed as our application server. An API [1] is a set of definitions and protocols for building and integrating application software. In our case, we used NodeJS to develop it, an asynchronous and event-driven JavaScript run-time, designed to build scalable network applications. In addition to that, we are also going to use ExpressJS, a NodeJS framework for building APIs in JavaScript. The current state of the art of developing an API using ExpressJS, is to use the Model View Controller (MVC) design pattern.

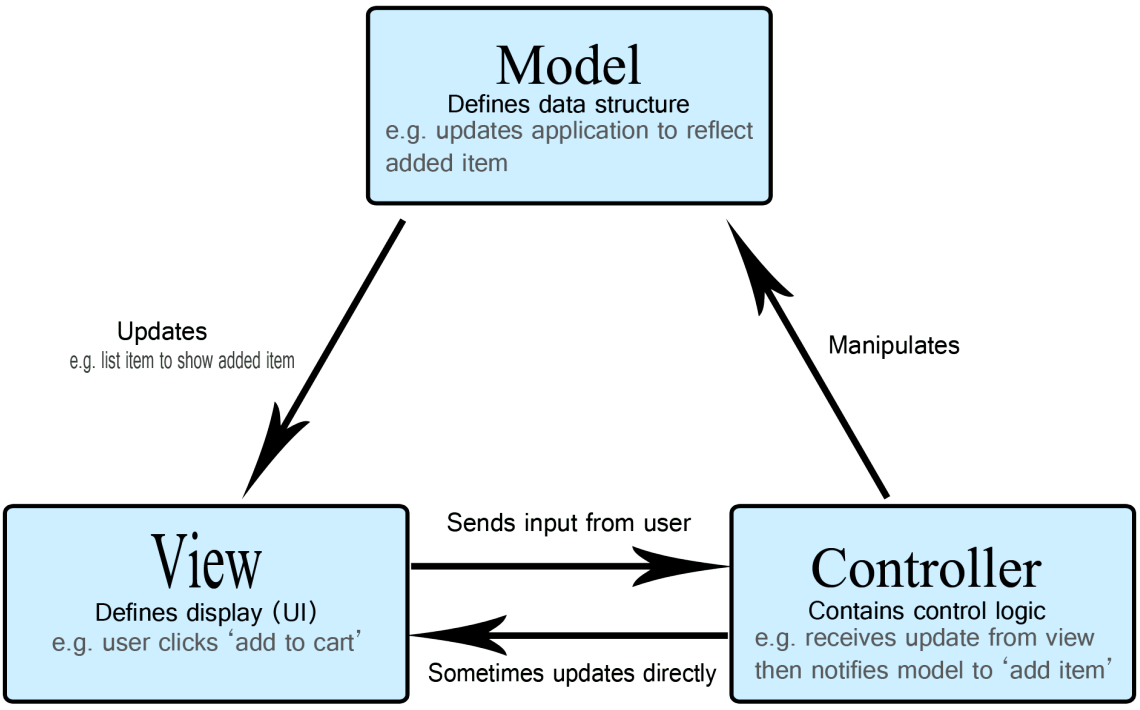

Figure 1: Model-View-Controller design pattern. [2]

The MVC [2] seen in Figure 1, is a pattern design commonly used to implement user interfaces, data, and controlling logic. It emphasises a separation between the business logic and the display of the software, in three components. The Model component, which stores and manage the data (usually a database). The View component, which is a visual representation of the data-like a chart or a diagram (nowadays this goes in a separated component in the frontend, outside of the backend scope). And finally, the Controller component, which provides the brains of the application, and connects the views with the models by converting the inputs from the views to demands to retrieve/update data in the models. The principle of this pattern is that each component is separated into different objects, i.e. components cannot be combined within the same class.

Along with the MVC pattern, that defines how the different components in our application are classified and interact with each other, we are also going to follow the REpresentational State Transfer (REST) architecture. REST [3] is a type of API architecture that allows the client and the server to be implemented independently without the knowledge of the other entity. This means that code at either side can be modified without having to worry about the effect of the modification on the other side. As long as both sides agree on the format of communication between them, they can be kept modular.

In order to strengthen the application from the development phase, we are going to code the REST API in Typescript. Typescript [4] is a super set of JavaScript, that allows static typing, class based objects, it is a compiled compiled language instead of an interpreted language, and supports type definitions. There are some advantages in using Typescript instead of pure Javascript. The first one being that it helps preventing errors or bugs before they happen, in an early development phase. This makes the development more consistent, and scalable. Moreover, since it prevents some common (or not so common) errors or bugs, it strengthens security from the very beginning, since less bugs leads to less potential vulnerabilities, increasing the overall level of security of the application. This helps us make our application more robust and bug free from the beginning thanks to these features. It is also imperative to provide a Secure Sockets Layer (SSL)/Transport Layer Security (TLS) certificate, and redirect all connections to Hypertext Transfer Protocol Secure (HTTPS), in order to protect the transmission channel. Alongside securing the channel, in order to authenticate user, it is usually implemented a Token-Based Authentication, using the Open Authorization (OAuth) standard.

2 Justification

As technology evolves over time, so do vulnerabilities, which leads to new emerging threats regarding security. Nowadays there is even a black market in which malware is developed as a service, and it is easier to obtain and to execute. Black Hats [5] are evil hackers that provide and/or perform attacks using malware, to whom may have different motivations than theirs. Some of them just want to make money out of it, and some other just follow an ideology or their own ethical code to justify the attacks. Regarding black hats,there are some of them providing Open Source INTelligence (OSINT) [6] services for clients, which consist in data gathering and infrastructure reconnaissance against targets that can be either competitors or other organisations. Furthermore, many black hats make a living by developing malware using the previous mentioned services as source of information, and developing the code that will exploit those gathered vulnerabilities, being the ransomware one of the most popular nowadays.

This completely changes the view of security since there are more and more frequent attacks that affect not only public services, but private services and all kinds of institutions as well. Hence, the need for platforms, applications and services to increase the minimum security features implemented, in order to be more secure.

Therefore, this project aims to implement as many security features and good practices in a API application and in each of the components of the service, the web server and the database (both connection and information obfuscation). There is a frontend as well for this project, however is designed for visual purposes only, the main purpose of this project is applying the best code practices, security features, and web server configurations to harden the API security. This project aims to be example of the minimum security every application or service should implement.

3 Methodology

To manage a project efficiently, the project manager or the development team must choose the software development methodology that will work best for the project at hand. All methodologies have different strengths and weaknesses and exist for different reasons. By choosing the appropriate methodology we can organise and fulfill the deadlines we set and therefore manage the workload in an optimal way.

The more commonly used software development methodologies [7] are the following:

- **Agile development:** Minimises risks such as bugs, cost overruns, and changing requirements when adding new functionalities. In all agile methods, teams develop software in iterations that contain mini-increments of new functionalities. There are many different variants of the agile development method, including scrum, crystal, Extreme Programming (XP), and Feature-Driven Development (FDD).
- **DevOps deployment:** Focuses on organisational change that enhances collaboration between the departments responsible for different segments of the development life cycle, such as development, quality assurance, and operations.
- **Waterfall development:** Rigid linear model that consists of sequential phases (requirements, design, implementation, verification, maintenance) which focus on different goals. Each phase must be 100% complete before the next phase can start. There is usually no process for going back to modify the project or direction.
- **Rapid application:** Is a condensed development process that produces a high-quality system with low investment costs. The ability to quickly adjust is what allows such a low investment cost.

3.1 Chosen Methodology

The chosen methodology for this project will be a hybrid between the Agile and DevOps methodologies. Agile allows us to develop the application in a fast and comfortable way, given the deadlines of this project and since we need to be aware of bugs and security flaws that we may encounter or detect while developing, and keep up with those changes fast enough. The DevOps methodology is used in addition to the Agile in order to make the deployment and integration of every dependency or version of the app as well as the app itself, faster and fully automated thanks to a good Continuous Integration / Continuous Deployment (CI/CD) infrastructure design.

3.2 Methodology applied to API development

For the DevOps methodology we used Dockers [8], an open platform for developing, shipping, and running applications, along with the Docker-Compose orchestrator. Docker allows us to encapsulate all dependencies in a Docker Image, in order to make it completely portable, and the Docker-Compose orchestrator, allows us to manage the different dockerized services that we need, creating a Docker Container (a virtual machine running one or more services) for each service, those being the web server service, the database, and the API server itself. Regarding CI/CD, we will make use of

Github Actions [9] to easily create a CI/CD pipeline. In this pipeline we will build the Docker images for our services, and push them to DockerHub, a repository platform to store Docker Images. After this step, the pipeline then will connect to our Azure Virtual Machine (VM), pull those images, and start the Docker containers.

For the Agile methodology, whenever a bug or feature fix is added, or a feature or security feature is added, we will create a branch in GitHub, and once it is finished, merge it to the main branch. This will allow us to have more control and a better git historic of what we have been doing or implementing, in order to keep up with this low granularity integration.

4 Scope

The aim of this project is not the development itself of a fully functional e-commerce REST API application. It is the research and implementation of the best security measures and features of the current state-of-the-art, regarding API development. This of course, involves not only the API server, but also the components that interact with it, and may be a target as well. Hence, security at code level is far from enough, and is also needed to apply security in the web server level (if used, which is commonly the case), the database (from connection to data protection), and in our case, Docker Images and Containers as well.

4.1 OWASP Top 10 measures implementations

One of the main goals of this project is to implement all the OWASP Top 10 recommendations regarding API security, HTTP security headers, database secure configuration and data protection, authorisation and authentication. Following this measures, and being able to implement protection mechanism against those issues / vulnerabilities, grants a very good base level of security. This is because those vulnerabilities, tend to be the most exploited ones in every application, since the lack of control or protection against them, and many of these vulnerabilities may allow complete control over the business, with relatively ease, if not taken care of.

4.2 Protection applied at all levels

As explained before, given a target, there are many attack vectors, and of course it is impossible to control and protect all of them. However, the main goal of any developer or security engineer is to make the application as much robust as possible. This is done by implementing as many security measures as possible, so potential attackers have to take lots and lots of time and resources to eventually exploit them. Given that, we need to cover the security of the virtual machine as well.

4.3 Docker security implementation

In Docker there are also security improvements and recommendations, in order to properly isolate the services running in the machine, from the machine itself. This will

be covered in the security feature section, but Dockers can indeed be a security flaw and become an attack vector.

4.4 Security by design

The main idea of this application is to build it having security as the main focus, rather than number of functionalities, or to provide an operable e-commerce platform. Hence, there is no payment functionality implemented, since this would require to have an account in any of the typical payment provider platform like Stripe, and develop all the payment process which really is not the main objective. However, we will provide and develop the basic operations available in an e-commerce application, providing all Create Read Update Delete (CRUD) functions, but properly implementing role control.

5 Stakeholders

Since the aim of this thesis is to show tools and follow security standards or recommendations in order to build safer applications, the amount of profiles that might be interested in the topics explained here and the solutions applied, cover from common users to security specialists, auditors or developers.

5.1 Software Developers

As a developer, nowadays is more and more of a requirement to be educated in security, not only in general Internet security, but specific technologies and security patterns or recommendations. In the development takes place the major part of the securisation, testing and functionality implementation, hence, the need of being able to identify possible weaknesses while coding, or be able to audit it and discover vulnerabilities, as well as keep informed about cybersecurity news.

5.2 Internet users / clients

Regular users need to be educated in the matter of cybersecurity as well. However, this project will help them in a more indirect way, since making the Internet a safer place, will affect them as users as well. The more robust the applications, the more protected the user will be against malicious users.

5.3 Service owners / providers

For an e-commerce platform owner, it is very important that the service is as secure as possible. Not only brings more confidence to the users / clients, but it is crucial to avoid major losses in income, data and clients. Moreover, having a higher rating in security also rewards the Search Engine Optimization (SEO) of the service.

5.4 Myself

As a Security IT Engineer, with developer experience and strong interest in applying the maximum security possible in app development or infrastructure (DevOps) and as

a fan of Ethical Hacking, this thesis brings me the perfect opportunity to improve and grow in all those aspects. Being aware of security issues, and being able to tackle them or detect them is a huge factor and something I would like to be able to do, not only to make the Internet more secure but help others detect security weaknesses and be able to know the risk of each vulnerability, understand how they work and help patch them.

Applying Security Measures

6 Organisations

There are a lot of organisations that work towards standardisation and security. Following the standards is very important because it helps to apply a common methodology or to keep a reference in all regarding Information Technology (IT), which at the same time helps enhancing security. Those organisations that develop and promote standards are very often used as reference for good practices regarding code, protocols to use, etc and even for cybersecurity.

There are many organisations that work on this matter, the World Wide Web Consortium (W3C) and the National Institute of Standards and Technology (NIST) are a couple of examples. W3C [10] is an international consortium in which member organisations, full-time staff and the general public work together to develop web standards and guidelines designed to ensure the long-term growth of the web. Moreover, the NIST [11] is an agency of the Technology Administration of the U.S. Department of Commerce that is authorised to provide measurement services, including calibration services, for organisations or individuals located outside the United States.

In the case of NIST, they provide many frameworks, one being the cybersecurity framework [12], in which they develop cybersecurity standards, guidelines, best practices, and other resources to meet the needs of U.S. industry, federal agencies and the broader public.

7 Open Web Application Security Project (OWASP)

The Open Web Application Security Project (OWASP) [13] is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.

In this project we will be using what is called the OWASP recommendations, which are articles summarising the top ten best practices and security measures regarding a topic. This topics are either regarding APIs, databases, web server or security http headers and more. The aim of these articles is to provide information about the most common or/and critical mistakes that can be made in the aspect of code, design, or implementation of

each of the aspects. Thanks to this, developers can be more aware of the most common vulnerabilities and learn about them in order to fix them, hence enhancing the overall security int he Internet.

7.1 OWASP Top 10 recommendations

The OWASP Top 10 [14] is a standard awareness document for developers and web application security, which represents a broad consensus about the most critical security risks to web applications. The list has been very successful and well received among the community due to the fact that it is easy to understand and master, it helps users prioritise risk and it is litigable. Therefore, when developing an application, or adding security features, this is the first place we can check, and review all vulnerabilities that have to be covered.

7.2 OWASP Top 10 API

The Top 10 vulnerabilities regarding APIs are described as followed in descending order, being the first one the most important vulnerability. This is also and up-to-date with the OWASP Top 10 version of 2021, which implemented some changes regarding the list order. For each vulnerability, a Common Vulnerabilities and Exposures (CVE) and a Common Vulnerability Scoring System (CVSS) is assigned. CVE is a list of publicly disclosed computer security flaws, however, whenever someone refers to CVE, it means a security flaw that has been assigned a CVE identification number. Every CVE has a corresponding CVSS, which is a number score assigned to that vulnerability regarding its criticality and how easy is to be exploited.

7.2.1 A01:2021 – Broken Access Control

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorised information disclosure, modification, or destruction of all data or performing a business function outside the limits of the user.

A couple of examples of this vulnerability are:

- Cross-origin resource sharing (CORS) misconfiguration allows API access from unauthorised/untrustworthy origins.
- Bypassing access control checks by modifying the Uniform Resource Locator (URL) (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool modifying API requests.

7.2.2 A02:2021 – Cryptographic Failures

Failures related to cryptography (or lack thereof), which often lead to exposure of sensitive data. This concerns protocols such as Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP) also using TLS upgrades like STARTTLS, or any protocol or operation that sends sensitive data in clear text, or encrypted data using weak cipher algorithms.

7.2.3 A03:2021 – Injection

An attack perform when user-supplied data is not validated, filtered, or sanitised by the application. Furthermore, dynamic queries or non-parameterised calls without context-aware escaping if used directly in the interpreter, are also vulnerable to this attack. A successful Injection attack may lead into leaking data from columns, tables or whole databases within the system, or even control over the database itself, allowing its modification or deletion.

7.2.4 A04:2021 – Insecure Design

Insecure design is a broad category representing different weaknesses, expressed as "missing or ineffective control design." Meanwhile, a secure design can still have implementation defects leading to vulnerabilities that may be exploited, an insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required.

7.2.5 A05:2021 – Security Misconfiguration

This vulnerability refers to missing appropriate security hardening across any part of the application stack, improperly configured permissions on cloud services, default accounts and their passwords are still enabled and unchanged or unnecessary features enabled or installed. All this misconfigurations may create vulnerabilities in the application, given the lack of control around them, or providing attack entries for malicious attackers.

7.2.6 A06:2021 – Vulnerable and Outdated Components

Lack of knowledge or compatibility test of the versions of all components used, both client-side and server-side, including components nested dependencies. If the software is vulnerable, unsupported, or out of date, an attacker can use the security flaws of those components versions to perform an attack. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, run-time environments, and libraries.

7.2.7 A07:2021 – Identification and Authentication Failures

Confirmation of the identity of the user, authentication, and session management is critical to protect against authentication-related attacks. Hence, if the application permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords, brute force or other automated attacks, default, weak, or well-known passwords, such as "Password1" or "admin/admin", that makes it vulnerable. The authentication process is very delicate, and sometimes is one of the least robust mechanism, allowing attackers to login with session ids that are not verified, etc.

7.2.8 A08:2021 – Software and Data Integrity Failures

Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. For instance, when an application relies upon plugins, libraries, or modules from untrustworthy sources, repositories, and Content Delivery Networks (CDNs). Furthermore, an insecure CI/CD pipeline can also introduce the potential for unauthorised access, malicious code, or system compromise.

7.2.9 A09:2021 – Security Logging and Monitoring Failures

This is not a vulnerability per se, since there is not much CVE or CVSS data for this category. However, the lack of risk or breach detection given by a poor logging and monitoring system is critical. This may occur when auditable events, such as logins, failed logins, and high-value transactions, are not logged, or warnings and errors generate no, inadequate, or unclear log messages.

7.2.10 A10:2021 – Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, a Virtual Private Network (VPN), or another type of network Access Control List (ACL).

8 Other important organisations

There are other organisations that even though are not completely focused on cybersecurity, help the whole community with creating standards, in order to be followed as examples of good practices or good process definitions. A couple of examples of the most important ones are the International Organisation for Standardisation (ISO) and Internet Engineering Task Force (IETF).

8.1 International Organisation for Standardisation (ISO)

The ISO [15] is an independent, non-governmental international organisation that brings together experts to share knowledge and develop voluntary, consensus-based, market relevant International Standards that support innovation and provide solutions to global challenges. They also promote the use of proprietary, industry and commercial standards worldwide.

8.2 Internet Engineering Task Force (IETF)

The IETF [16] is an open international standardisation organisation, which aims to contribute to the engineering of the Internet, acting in several areas, such as transport, routing and security. The technical work of the IETF is done in Working Groups, which are organised by topic into several Areas. For instance, IETF participants have been updating both the core specifications to HTTP, affecting all versions of the protocol, and they have been developing HTTP/3, the latest version of the protocol. The entire

definition of HTTP has been revised, with definitions for HTTP/1.1, HTTP/2, and HTTP/3 either revised or new.

Project architecture design

9 Architecture design

For the architecture of the project, we wanted to design something that follows the common way of structuring an Application of the sort we are building. This consists in having a frontend, a backend and a web server. The web server will grant SSL to our communication channel with the clients, and will perform as a reverse proxy for our frontend, and our backend. The backend must only be dependent of the Database, since without that service, it cannot run. The frontend however, must be able to be launched or loaded regardless if the backend is available or not. In case it is not available, error messages may appear in the frontend to notify the user. Even though the web server will act as a reverse proxy, it will only do so for the frontend and backend, since only the backend needs to be able to connect to the database, it shall not provide reverse proxying for the database service, it must only be accessed internally.

9.1 CI/CD flow and description

The whole environment will be hosted on a VM in Azure, a Cloud Computing service provided by Microsoft , and we will be using Github Actions, a Github tool to build pipelines for deployments, to automate the CI / CD of our project. In this pipeline, whenever we make a push to the master branch, the pipeline will trigger. There are three jobs to be performed, tow of them can be run in parallel, whereas the third requires both of the previous jobs to be finalised before starting. In all jobs, our secrets will be available to be retrieved from Githubs secret storing system, and their value are not shown anywhere, not even in the pipeline logs.

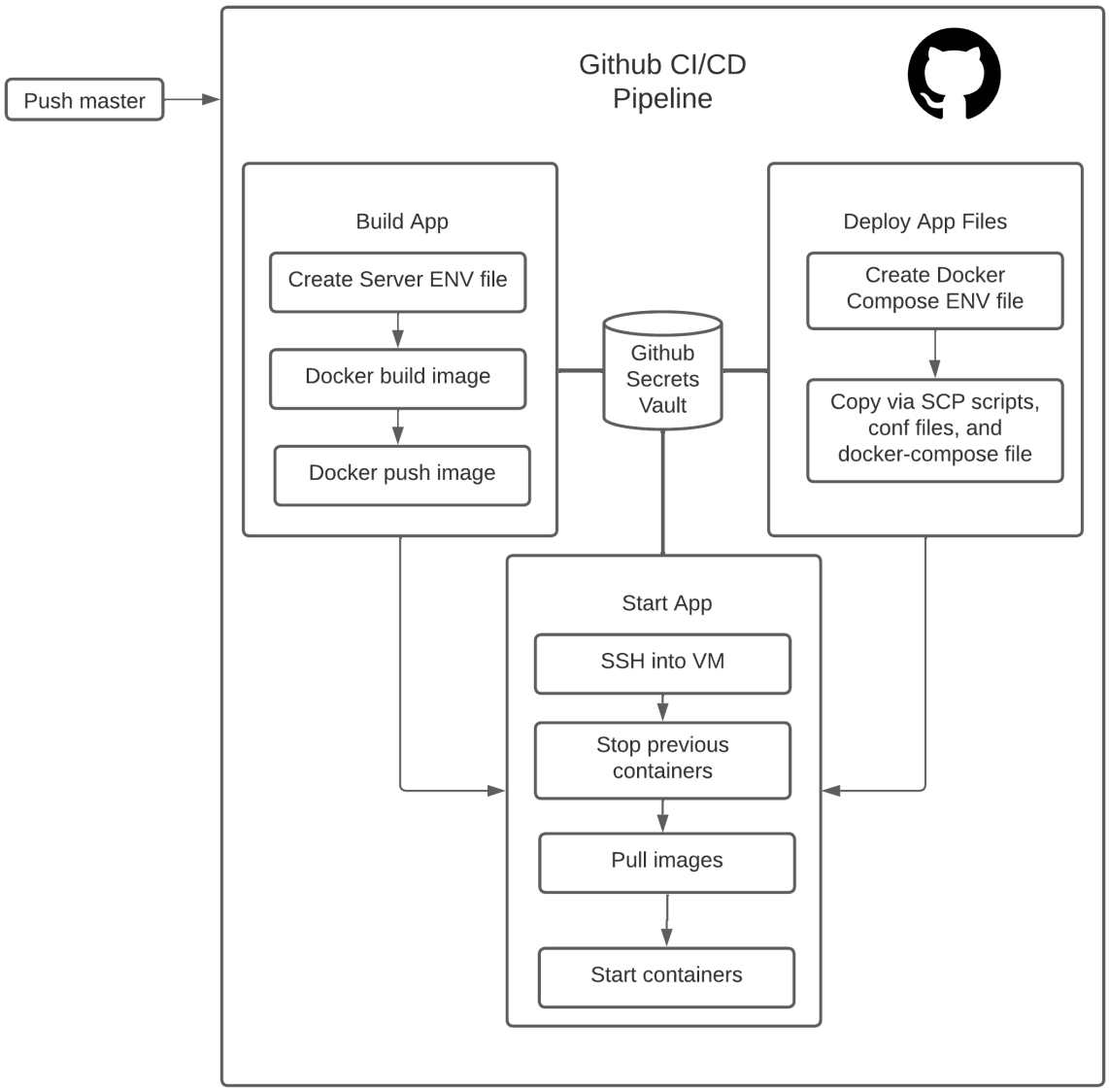

Figure 2: CI/CD Pipeline flow. Source: Own compilation.

The *Build APP* job performs tasks to build the app image, and push it to the image repository in Docker Hub. On the other hand, the *Deploy App Files* job deploys the scripts, docker compose file and configuration files required for starting the application in the final job. This job is required since we do not want to copy all the application repository files to the server, just the files needed. The last job only connects to the server in Azure, and perform a series of shell commands to start the app again.

9.2 Design details and description

We have each service running in its own container, and all containers will run in a specific network built for them. The network is configured with a mask of /28, since we do not have many services and do not need many IPs. Using a private network with docker has many advantages compared on using the default network bridge created by Docker. The first is that with a custom network, we can isolate our components from

other networks in the machine. The second one, is that we can configure our database to only accept connections from that network IP range, rejecting any other device that may try to connect to our database. Furthermore, having control of our network, allows us to specify the number of IPs available in that sub-network, hence allowing only as many containers as we need.

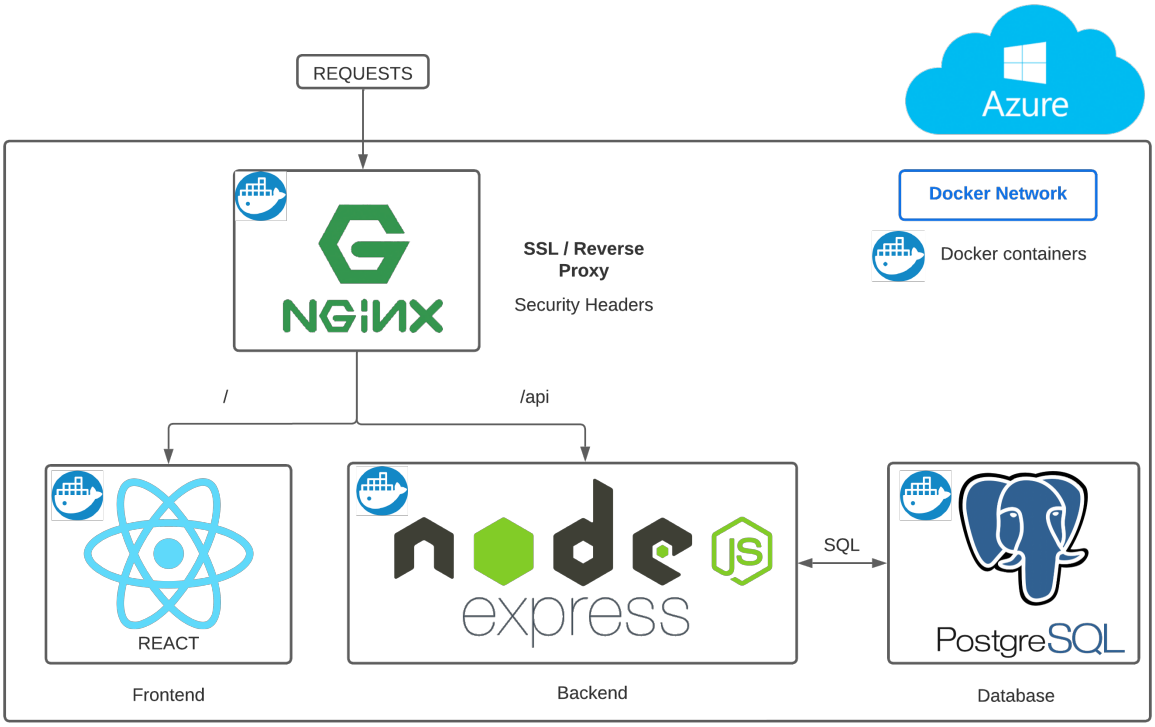

Figure 3: Final API architecture design. Source: Own compilation.

As seen in Figure 3, our NGINX web server will provide TLS for our connections, loading our SSL certificates, and provide security headers for all HTTP responses. This will harden the communication between the client and the server, providing an encrypted channel, removing unwanted HTTP headers that may reveal too much information, and allowing only the HTTP methods that we want, hence, rejecting vulnerable methods like the OPTION HTTP method. The reason behind applying everything regarding TLS and HTTP header configurations in our web server rather than in the API server, is because the web server is supposed to abstract those settings from the server, and leave the server to only perform the business logic. A reverse proxy configuration, allows us to redirect the incoming requests to the appropriate service. For instance, if the request is asking for the */api* resource, it will redirect that request internally, to our express server container. Hence, the server can provide the actual functionalities, and perform the operations to the database. On the other hand, if the client wants to access the home (/) resource, the NGINX will return the frontend web page to the client.

The backend consists of a REST API server built with ExpressJS and written in Typescript, from which we can highlight two important parts. On the one hand we will have the definitions of all our models, which will be in charge of operating each of the database tables, and represent them in the code as a class, like the User model or the Product

model, for instance. On the other hand, we will have the definitions of all our controllers, which have the function definitions that will handle each request, and will have the methods that provide the functionality for each endpoint. All operations needed to manage the database are declared in repositories, which are classes that take a specific model and provide methods to manage the table represented by that model. Therefore, in our controllers we only need an instance of the repositories that the controller may need to fulfill the function of the endpoint. This allows us to have the model and controller components completely modular, since the do not share dependencies between them, but through the repositories, as explained in the MVC pattern in section 1.

For the database we chose PostgreSQL [17], a powerful, open source object-relational database system that has earned it a strong reputation for reliability, feature robustness, and performance. We chose PostgreSQL rather than other solutions, first because it is a Structured Query Language (SQL) [18] database, and second because is one of the most active in applying and promoting the standards, regarding databases or data types.

Finally, for the frontend, we will use a JavaScript library called React. React [19] is the current most popular JavaScript technology used for building User Interface (UI)s. We chose react, because it has the biggest active community, and the biggest number of technology related package repositories. Furthermore, given that it is a library and not a framework, it allows us to only install the packages that we need, and avoid any unnecessary boilerplate. It allows Typescript as well, which of course we will be using in order to write the code of the frontend.

10 Project Description

Our project will simulate a common e-commerce application. This application is owned by a business owner, and its purpose it is to show the products and allow the purchase of those products. Hence, only the owner or designated employees are allowed to add more products to the platform, whereas clients are only allowed to authenticate against the application, browse products, and place orders. However, and as mentioned before, since the purpose of this project is not to develop a complete application ready for be used for business, the payment process it is not implemented, but only simulated. This means that there is no payment gateway whatsoever, only by pressing the pay button, the order will be marked as payed. The reason behind that is because implementing a real payment system, requires several steps that are non-related to the application, and it does fall out of the scope of the project.

10.1 Authentication, Authorisation and Roles

The authentication of this project will be performed via local user accounts, and via Google using OpenID Connect [20]. OpenID Connect is a simple identity protocol and open standard that is built on the OAuth 2.0 [21] protocol, which is the industry-standard protocol for authorisation. OAuth 2.0 verifies the clients identity via third party, provides consented access on the users resources on the third party applications by the client app on behalf of the user and restricts actions of what the client app can perform on those resources on behalf of the user, without ever sharing the users credentials.

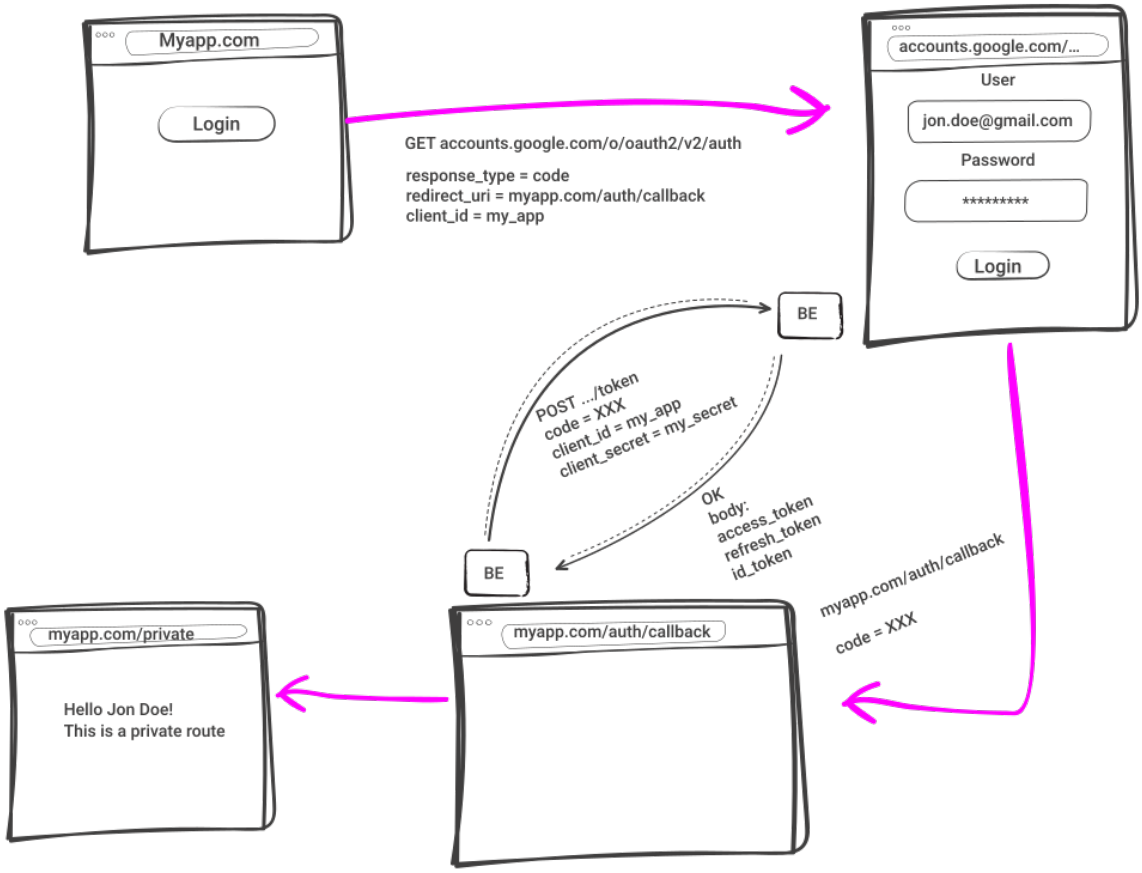

Figure 4: OpenID Connect authentication flow [22].

OpenID Connect enables client applications to rely on authentication that is performed by an OpenID Connect Provider or Relaying Party (RP), which is a server that is capable of providing claims to a client, to verify the identity of a user. The reason of using OpenID Connect instead of OAuth2.0, is because we only want to provide authentication to the user, not authorisation, since we do not need to access any of the Google protected resources of the user. In fact, authorisation will be managed by our application, to allow or restrict access to the endpoints that provide the different functionalities summarised in previous sections.

Nevertheless, once the user has been authenticated successfully, by either local or Google authentication, we get the ID of that user, and store it in a signed JSON Web Token (JWT), following our Token-Based Authentication mechanism mentioned in the introduction of this thesis. A JWT [23] is an open standard (RFC 7519) [24] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC [25] algorithm) or a public/private key pair using RSA [26] or ECDSA [27]. We chose to use the first the secret method since symmetric encryption is faster and easy to manage, since that key has not to be send to anyone.

The JWT will be signed with HMAC using SHA-256 hash algorithm, set with an expiration time of five hours, because low expiration times are recommended in case of stolen tokens, and with the issuer and audience set to **"https://tfm.jediupc.com"** and **"https://tfm.jediupc.com/api"**, respectively.All this configuration information is stored in environment variables, through an ENV file. Moreover, below we have the JWT creation function that receives a parameter, with the data to store inside the token, and creates a signed token with that data, the secret for the signature and the configuration mentioned before.

```
1  public static createNewJWTToken = (info: JWTAccessSignInfo):
       string => {
2      const algorithm = (process.env.JWT_ALG! as Algorithm) ||
         undefined;
3      const expiresIn = parseInt(process.env.JWT_EXPIRATION!)
         || 5 * 60 * 60;
4      const token = jwt.sign( info, process.env.JWT_SECRET!, {
5          algorithm,
6          expiresIn,
7          issuer: process.env.JWT_ISSUER!,
8          audience: process.env.JWT_AUDIENCE!
9      });
10
11     return token;
12 };
```

Listing 1: JWT creation and configuration.

The ***info*** variable contains the ID of the existing or recently created user, which is the only information that we need to store. The secret used for the signature is a 32 byte key generated manually with OpenSSL [28], a full-featured toolkit for general-purpose cryptography and secure communication. Since this secret (among others) will be stored in a Github variable secret, to be written afterwards in an environment file for our application, we do not need to think about storage or how to hide this value.

Since JWTs are stateless, we cannot "*end*" the user session per se. However, we can simulate a logout process, by blacklisting the current access token when the user wants to log out. The JWT blacklisting process inside the logout endpoint consists in banning that access token, storing into a Blacklist Token table within the database. In all requests that require authentication, we do not only check if an access token is present, but if exists an entry in the Blacklisted Tokens table, with that token. If the token is in the database, or in other words, blacklisted, we will return a 401 HTTP code, as if the user was not logged in, thus, forcing the user to authenticate again.

In order to send the token to the user, so it can be used for authentication and authorisation purposes, we used Signed HTTP Cookies [29]. An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to a users web browser. The browser may store the cookie for sending it back to the same server in future requests. In our case, we will used signed cookies, which are cookies which value is still visible

but have a signature, so the server can detect if the client modified the cookie in any way.

```
// using signed cookies
this.app.use(cookieParser(process.env.COOKIE_SIGNATURE));
```

Listing 2: Cookies configuration with secret

```
// signed cookies configuration
public readonly login = (req: Request, res: Response, next:
   NextFunction): Response | void => {
    logger.info("In [POST] - /login");
    try {
        ...

        return res.status(200).cookie('access_token', token,
           {
            secure: true,
            signed: true,
            httpOnly: true,
            maxAge: parseInt(process.env.JWT_EXPIRATION! || "
               0") || 5 * 60 * 60 * 1000
        }).send(user).end();
    } catch (error: unknown) {
        next(error);
    }
};
```

Listing 3: Cookies configuration parameters

Our cookies have the *signed* attribute to true, in order to sign them with the secret passed in Listing 2. Moreover, the cookies have the *httpOnly* and *secure* flags set to true. The first flag being set to true, forbids any JavaScript code on the client side to read the cookie, whereas the second flag, only allows to send the cookie via a secure channel using HTTPS. Finally, we set the cookies expiration time the same as with the access token, in milliseconds format.

Regarding authorisation, we designed a role based user administration consisting of three user roles. The most basic one is designed for the clients, which will be able to authenticate itself, browse all products and categories, and make a purchase (place an order). The second level of privilege is the Administrator role, which will be able to do all operations mentioned before, with the addition of being able to create, delete, or edit everything regarding products, categories, and user orders. Last but not least, the Super Admin role, which can perform all that the Admin can, but with the addition of managing users as well. This means, creating, editing (which allows editing other users privileges) or deleting users.

10.2 API Endpoints description

In order to have better control over the different functionalities available for the three different user roles, the endpoints have been divided in two groups. The first group

are the public endpoints, available for the Client role, and then the admin endpoints, available for the Admin and Super Admin roles. To check if a user has access to an endpoint, we use a JWT, in which we store the database ID of that user. In the database, all identifiers are defined as a string with Universally Unique Identifier (UUID) format. An UUID [30] is a 128-bit (32 alphanumeric characters) unique label used for identifying information. Once a user attempts a HTTP request against an endpoint, we extract the token, we obtain the users id from that token, and check in the database its privileges.

10.2.1 Authentication and Authorisation endpoints

Here we group all endpoints regarding authentication for the user, this means being able to create an account, logging in, or logging out.

- **[POST] /auth/login**: Allows the user to log into the application with a local created account.
- **[GET] /auth/google**: Allows the user to log into the application via a Google account using OpenID Connect protocol.
- **[GET] /auth/google/callback**: If the Google authentication has succeeded, the user is redirected by Google to this route, where we use that information to create a JWT token for the users session.
- **[POST] /auth/signin**: Allows the user to create a new local account in the application.
  The data expected is an object with the following properties:
  - **username:** A string.
  - **password:** A string.
- **[GET] /auth/logout**: Stores the users current valid JWT in a blacklist database, whenever a user accesses an endpoint that requires authentication, if that JWT is stored in the blacklist, the user wont be granted access.

At the end of the login, google callback and signin endpoints, when the user has authenticated successfully, is where we create a JWT with the users ID as payload. Once we have that token, we send back the response with a 200 status, the cookie set with previous mentioned parameters and with the token as value, and the whole user data (without the password) back as body payload.

10.2.2 Products endpoints

In these endpoints we have all functionality regarding products, clients can browse them all, browse one specific product, or apply a filter search. All other CRUD operations are only allowed to Admin users. For assigning an image to the product, only files with Joint Photographic Experts Group (JPEG) (or JGP) or Portable Network Graphics (PNG) extensions will be accepted, otherwise it will be rejected.

- **[GET] /products**: Obtain all products information from the database.

- **[GET] /products/:id**: Obtain a the products information with the corresponding id from the database.
- **[POST] /products/filter**: Create a filter resource to get only the products that meet the filter options.
  The data expected is an object with the following properties:
  - **name (optional):** A string.
  - **price (optional):** A number.
  - **stock (optional):** A number.
  - **category (optional):** A string, that must match the name attribute in a row in the Categories table.
  - **premium (optional):** A number between 0 or 1, being 1 considered as premium.
  - **description (optional):** A string.
- **[GET] /products/filter/:category_name**: Obtain all products that have a certain category assigned.
- **[POST] /admin/products**: Add a new product to the database. (Admin only)
  The data expected is an object with the following properties:
  - **name:** A string.
  - **image:** A file with JGEP, JPG or PNG formats.
  - **price:** A number.
  - **stock:** A number.
  - **category:** A string, that must match the name attribute in a row in the Categories table.
  - **premium (optional):** A number between 0 or 1, being 1 considered as premium.
  - **description:** A string.
- **[PUT] /admin/products/:id**: Edit the product with the corresponding id. (Admin only)
  The data expected is an object with the following properties:
  - **name:** A string.
  - **image:** A file with JGEP, JPG or PNG formats.
  - **price:** A number.

  - **stock:** A number.
  - **category:** A string, that must match the name attribute in a row in the Categories table.
  - **premium (optional):** A number between 0 or 1, being 1 considered as premium.
  - **description:** A string.

- **[DELETE] /admin/products/:id**: Delete the product with the corresponding id from the database. (Admin only)

Since every product needs a category, which has to exist in the categories database, if the category we want to assign does not yet exists, we need to create it first.

10.2.3 Categories endpoints

In these endpoints we have all functionality regarding categories, clients can browse them all or browse one specific category. All other CRUD operations are only allowed to Admin users.

- **[GET] /categories**: Obtain all categories information from the database.
- **[GET] /categories/:id**: Obtain a the categories information with the corresponding id from the database.
- **[POST] /admin/categories**: Add a new category to the database. (Admin only)
- **[PUT] /admin/categories/:id**: Edit the category with the corresponding id. (Admin only)
- **[DELETE] /admin/categories/:id**: Delete the category with the corresponding id from the database. (Admin only)

The only data needed to create or edit a category is to provide a name with an alphanumeric string as a value.

10.2.4 Orders endpoints

In the orders endpoints we have the available functionalities for the client user, which involves browsing and editing all orders that are considered owned by that client (being the client_id property equal to the current users ID obtain from the current JWT). The date of the order is always automatically added in the server, it is not provided. This date will be a string in a ISO Date-Time format. This format refers to the ISO 8601 format, that represents date and time by starting with the year, followed by the month, the day, the hour, the minutes, seconds and milliseconds.

- **[GET] /orders/own**: Obtain all owned orders information from the database.
- **[GET] /orders/own/:id**: Obtain the information of owned order with the corre-

sponding id.

- **[PUT] /orders/own/:id**: Edit owned order with the corresponding id.
- **[PUT] /orders/own/:id/cancellation**: Change the status of owned order with the corresponding id to cancelled.
- **[POST] /orders/own/place-order**: Add a new owned order.
  The data expected is a list of objects with the following properties:
  - **product_id:** A string with UUID format referencing an existing product in the database.
  - **quantity:** A number above zero.
  - **price:** A number above zero.
- **[GET] /admin/orders**: Obtain all orders information from the database. (Admin only)
- **[GET] /admin/orders/:id**: Obtain a specific orders information from the database. (Admin only)
- **[PUT] /admin/orders/:id**: Edit the order with the corresponding id. (Admin only)
  The data expected is an object with the following properties:
  - **date (optional):** A string with a date in ISO Date-Time format.
  - **status (optional):** A string that can only take the following values: pending, payed, shipped, delivered or cancelled.
  - **orderItems (optional):** A list of objects with the following properties:
    - **product_id:** A string with UUID format referencing an existing product in the database.
    - **quantity:** A number above zero.
    - **price:** A number above zero.
- **[DELETE] /admin/orders/:id:** Delete the user with the corresponding id from the database. (Admin only)

10.2.5 Users endpoints

Finally, we have the user endpoints, a user that is not a Super Admin, will only have access to the profile endpoints, otherwise access will be forbidden. Only Super Admins will be allowed to edit, delete, or create other users besides itself.

- **[GET] /profile**: Obtain own user information from the database.

- **[PUT] /profile/edit**: Edit own user information.
  The data expected is an object with the following properties:
  - **email (optional:)** A string with email format.
  - **firstName (optional):** A string.
  - **secondName (optional):** A string.
- **[PUT] /profile/edit/password**: Edit own user password.
  The data expected is an object with the following properties:
  - **password:** A string longer than 20 characters.
- **[POST] /admin/users**: Add a new user to the database. (Super Admin only)
  The data expected is an object with the following properties:
  - **email:** A string with email format.
  - **firstName:** A string.
  - **secondName:** A string.
  - **privileges (optional):** A number between 0 and 2, being 0 the basic role, and, 1 and 2 the Admin and Super Admin roles respectively.
  - **password:** A string longer than 20 characters.
- **[PUT] /admin/users/:id**: Edit the user with the corresponding id. (Super Admin only)
  The data expected is an object with the following properties:
  - **email (optional):** A string with email format.
  - **firstName (optional):** A string.
  - **secondName (optional):** A string.
  - **privileges (optional):** A number between 0 and 2, being 0 the basic role, and, 1 and 2 the Admin and Super Admin roles respectively.
  - **password (optional):** A string longer than 20 characters.
- **[DELETE] /admin/users/:id:** Delete the user with the corresponding id from the database. (Super Admin only)

11 Security Measures applied

The first measure taken was regarding the NGINX configuration, to allow and ensure that all connections are made using TLS. Therefore, we needed to install SSL certificates in our machine and load them in the configuration file of the web server, and also ensure

that we only allow TLS versions 1.2 and 1.3, since versions 1.0. and 1.1 are deprecated due to being vulnerable to attacks.

```
1    ## SSL LetsEncrypt
2   ssl_certificate /etc/letsencrypt/live/tfm.jediupc.com/
       fullchain.pem;
3   ssl_certificate_key /etc/letsencrypt/live/tfm.jediupc.com/
       privkey.pem;
4
5
6   ssl_session_timeout 1d;
7   ssl_session_cache shared:MozSSL:1m;
8   ssl_session_tickets off;
9
10  # Intermediate configuration
11
12  ssl_protocols TLSv1.2 TLSv1.3;
13  ssl_ciphers TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:
       TLS_CHACHA20_POLY1305_SHA256:ECDHE-ECDSA-AES128-GCM-
       SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-
       GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-
       CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-
       AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384;
```

Listing 4: SSl/TLS configuration

The next security measures we tackled are the ones described by the OWASP Top 10 article, summarised in section 7.2. Since they are the most common present vulnerabilities, and many of them are critical, meaning that a successful attack could compromise the entire business, we consider that it is the best place to start.

11.1 Fixing A01:2021 – Broken Access Control

Therefore, for the first vulnerability, the Broken Access Control, the first thing we implemented was to add and set the Access-Control-Allow-Origin to our domain in the configuration file of the NGINX web server. Doing so we make the web server to only accept requests that come from our domain, which are devices hosted in our Azure VM. Therefore, our frontend will be the only one allowed to do so.

```
1     add_header Access-Control-Allow-Origin "tfm.jediupc.com";
```

In addition to that, we applied an authorisation mechanism using JWT tokens, and roles, as well as two middleware functions (isAdmin and isSuperAdmin) to check the permissions in every endpoint, to make sure that the user is allowed to that resource. Since we store the database ID of the authenticated user in the signed JWT, we can check the privileges of the user in the database, in each middleware.

If the privilege checks fails, we return a 403 HTTP status code in the HTTP response, which is the standard status when a user is forbidden to access that endpoint. Furthermore, in order to implement a logout system in our application. We have a table

to store blacklisted JWTs, of the users that log out. We need this table, because every time a user logs in our application, the resulting token will be different every time, no matter what information we store inside. Hence, in order to simulate the log out, and complete a proper Authentication/Authorisation system, is to store this token when the user logs out, and for every request, add another middleware, that will check if the token used is not allowed (blacklisted) and the user requires to re-authenticate. If the token is no longer allowed, we return a 401 HTTP status code, which is the standard for an Unauthorised request.

```
1  // check if token is blacklisted - middleware
2  export const isTokenBlacklisted = async (req: Request, res:
       Response, next: NextFunction): Promise<Response | void> =>
        {
3    try {
4      if (!req.signedCookies['access_token'] || !req.user)
         return res.sendStatus(401);
5
6      const token: string = req.signedCookies['access_token'];
7      const blacklisted = await Token.findOne({ where: { token
         }, raw: true });
8      if (blacklisted) return res.sendStatus(401);
9
10     return next();
11   } catch (error: unknown) {
12     res.sendStatus(500);
13     throw error;
14   }
15 };
16
17 // check if token is blacklisted - middleware
18 export const isAdminUser = (req: Request, res: Response, next
       : NextFunction): Response | void => {
19   if (!(req.user as User).privileges) return res.sendStatus
       (403);
20
21   return next();
22 };
23
24 export const isSuperAdminUser = (req: Request, res: Response,
       next: NextFunction): Response | void => {
25   if ((req.user as User).privileges !== 2) return res.
       sendStatus(403);
26
27   return next();
28 };
```

Listing 5: Authorisation Middlewares

Furthermore, since only our server needs to establish a connection to the database

and operate it, in our database Docker configuration we create a specific database for the application, and prepare a user with only access to connect to that database, and perform SELECT, UPDATE, INSERT and DELETE queries, over the tables in that database. However, this user does not have access to other databases, and can the only operation allowed at database level is to connect to it and nothing else. In addition, we define in our PostgreSQL *pg_hba.conf* file, that will only allow two type of connections. The first one being a localhost connection for the root user, with a SHA-256 hashed password, and the second one for the user server, again with a SHA-256 hashed password but only that comes from our Docker network and to the database of our application specifically.

```
1 local     all          root                          scram-sha-256
2 host   tfm-database   server   192.168.80.0/28    scram-sha-256
```

Listing 6: PostgreSQL ph_hba configuration file

11.2 Fixing A02:2021 – Cryptographic Failures

In order to fix this vulnerability we just need to keep up to date with the most secure ciphers nowadays, and check which ones are insecure. Regarding TLS ciphers, in the code listing 4 we already allowed only TLS versions 1.2 and 1.3, and the allowed ciphers specified below. Furthermore, regarding password hashing, the general scenario is to use a package called Bcrypt [31] that uses key derivation functions to hash the passwords, in order to store the resulting hash in the database. However, we used Argon2 [32], a package to hash content as well, but uses a stronger key derivation function compared to Bcrypt to achieve it. This results in a more robust and brute force resistant hashes, and provides a key derivation function to create hashes resistant to Graphics Processing Units (GPU) brute force attacks as well.

11.3 Fixing A03:2021 – Injection

This vulnerability refers to the ability of an attacker to inject malicious code to our application, via input fields or via a payload sent to our server. The most typical injections are SQL Injection, since a successful SQL injection attack may lead to a complete leakage of data from the database, or absolute control over the database, database tables or all databases. This attack consists in injecting SQL queries in inputs of our frontend, than will be sent afterwards as payload to our server, to try to overwrite our own predefined queries to perform another operation. This happens only if we concatenate the query values (malicious query) of our payload directly to our SQL query.

For instance, given the following insecure query:

```
1 "SELECT * FROM products WHERE category = '";
```

Listing 7: SQL Injection vulnerable query

If we send the following payload:

```
https://insecure.com/products?name=Computer'+OR+1=1--
```

When we concatenate that query to our query string, the first character (´) will break the string and the last two characters (**--**) will comment the rest of the query. Resulting in the following computed query:

```
SELECT * FROM products WHERE name = 'Computer' OR 1=1--';
```

Listing 8: SQL Injection Attack performed on vulnerable query

This is solved by using parameterised queries, instead of plain strings, and sanitising our inputs. Since we are using Sequelize, a NodeJS package to define our database Models, which also provides functions that already have parameterised queries, so we only need to sanitise our input, in order to be protected against this attacks. We have to many functions to validate the payloads in the HTTP requests, and check the payload format corresponds to the expected inputs described in section 10.2.2.

Injection can be applied to headers as well. An example of that is the Host Header Injection [33] attack. This attack consists of exploiting vulnerable websites that handle the value of the Host header in an unsafe way. If the server implicitly trusts the Host header, and fails to validate or escape it properly, an attacker may be able to use this input to inject harmful payloads that manipulate server-side behaviour. In order to protect against this attack, what is needed is to check if the value of the Host header of a request is valid, if not so, a 444 HTTP error code is returned, otherwise, we proceed with the request. This 444 HTTP code is special in NGINX, since it is done to close the connection without sending a response. This is commonly used in NGINX when an attack or bad behaviour is detected.

```
1  if ( $host !~* ^(tfm.jediupc.com)$ ) {
2    return 444;
3  }
4
5  if ( $http_host !~* ^(tfm.jediupc.com)$ ) {
6    return 444;
7  }
```

Listing 9: Protection against Host Header Injection

In our case, we just need to check if the value is different from our domain, if it is different in any way, we return the 444 code. We check both the **$host** and **$http_host** variables of NGINX. Usually with just the first one is enough, however, some plugins or modules of NGINX may use the second one as well, so we just check both of them.

11.4 Fixing A04:2021, A05:2021 and A06:2021 vulnerabilities

Here will be summarised the measures applied for this group of vulnerabilities since they share many among them. This refers to Insecure Design, Security Misconfiguration and Vulnerable and Outdated Components vulnerabilities respectively. Since we are applying a security by design pattern, meaning that we are focusing always in the most secure version of any implementation we want to make, we are already covering

the first vulnerability. For instance, a good example is all the security mechanism we implemented from the beginning, including in the architecture of the project, with all the docker containers isolated in a network for their own, watching permissions in the app itself regarding authentication and authorisation, and so on. Furthermore, we also have a firewall provided by Azure, and we only have HTTP and HTTPS ports to the outside traffic, nothing else. This links with the second and third vulnerabilities, since we are handling all our file permissions and configurations with Docker compose, allowing only what each container absolutely needs, and with the Github CI/CD pipeline, which in order to connect to our VM and deploy the needed files, it uses an SSH key to make the connection. Moreover, all variables needed to create the environment files needed for our application are injected via Github secrets, which are encrypted and private. Finally, one thing we did with dockers is specify the version for each of all the technologies used, so we can ensure that always will be used that version, and manually install updates if needed, to avoid deprecated packages.

11.5 Fixing A07:2021 – Identification and Authentication Failures

This topic has been already addressed in section 10.1, however, one of the measures applied regarding authentication, is to ensure all user passwords are twenty characters long. Password length is one of the biggest vulnerabilities in every system, since weak passwords can be guessed via brute force attacks. In order to fix this, we applied two more measures together with our password length requirements. Rate limits have been applied in our NGINX configuration file, in which we set a maximum limit of ten requests per second for our */api* resource. If this limit is reached, the web server returns a 429 HTTP status code (standard response for making too many requests) and block that IP address for twenty minutes. This block means rejecting automatically any requests sent by that IP during the duration of the penalty.

```
1    limit_req_zone $binary_remote_addr zone=serverlimit:20m
       rate=10r/s;
2    limit_req_status 429;
```

Listing 10: Request rate limit configuration.

```
1    location /api {
2     limit_req zone=serverlimit;
3     ...
4   }
```

Listing 11: Request rate limit zone application.

The second additional measure taken consists in, for any authentication failure, always return the same error message, alerting that the credentials provided are wrong to avoid User Enumeration attacks. This attacks consist in introducing random users or known existing users to know if the error message received changes between an existing user an non existing one. This allows an attacker to map all user accounts in the database, and performed a more targeted brute forcing attack, which a higher chance of success.

11.6 Fixing A08:2021 – Software and Data Integrity Failures

In our project we only relay on third-parties regarding our NodeJs packages, and our Github repository, where we store the secrets needed by our application. All NodeJs packages used are well-known among the community and are developed, patched and reviewed constantly not only by the main authors of the packages but also by authorised members of the community, in order to correct bugs, add features, and improve. Regarding Github, we only depend on for our CI/CD pipeline infrastructure. Github is also a very well-known platform, and provide an encrypted storing system for secrets, which ciphertext still appears in the logs of the pipeline while running, instead of plane text, as an additional security measure. Therefore, for this reasons, we can trust those third-parties to maintain secrecy and ensuring security.

11.7 Fixing A09:2021 – Security Logging and Monitoring Failure

In order to implement logging in our application, to be able to trace back through all events after a successful attack or a server shutdown produced by an error, we used the NodeJS packaged Winston. Winston [34] is one of the most popular logging packages used which also receives lots of support from the community. In essence, it is a simple and universal logging library with support for multiple transports. A transport is essentially a storage device for logs, in our case, we used File transports, to store all events in a file in our container file system.

11.8 Fixing A10:2021 – Server-Side Request Forgery (SSRF)

In our application, we can only be affected by this vulnerability, in the products endpoints. Since we accept a URL, pointing to the image of the product that is served by the server, this can be a possible flaw. Hence, to correct that, we do not allow a user to manually change the URL value of that parameter. Instead, it can upload a new image in JPEG (or JPG) or PNG formats, since any other file format will be rejected, and internally a new URL will be computed and updated in database. Since we do not allow a user to input any URL that could make random and uncontrolled HTTP requests to uncontrolled servers or networks, we are protected against this vulnerability.

12 Additional Security Measures

There are more security measures and good practices applied to the project, although not all of them are regarding the server alone. Some of them may be regarding Docker containers and images or regarding HTTP security headers applied in the configuration file of NGINX, for instance.

12.1 Dockers

In order to improve security on our docker environment, we followed another list of OWASP security measures [35] to ensure the services work properly and following good practices. All containers will be running with a non-privileged user, which is provided by all the images we are using, but need to be specified in every one in other to use it. Otherwise, all containers will run with the root user by default, which is a very bad

practice, since it could escalate to delete the container. Furthermore, all permanent volumes created in the Host VM, which are folders maped to container folder inside of the file system of a container, will be set as readonly by default. An exception of this will be the **/var/log** directory, since the containers will need write access to create the log files.

In case of a Denial of Service (DoS) or a Distributed Denial of Service (DDoS) attack would occur, all containers will run with specific minimum and maximum limits of resources. Finally, when running Docker compose, the log level will be set to *info*, to log events of *info* and above, in case we would want to review them later.

12.2 NGINX

A part from the settings applied that are already mentioned and explained in previous sections, we added the following HTTP headers as well following the OWASP HTTP recommendations [36]:

```
1  add_header X-XSS-Protection "0";
2  add_header X-Frame-Options "DENY";
3  add_header Access-Control-Allow-Methods "GET";
4  add_header Access-Control-Allow-Credentials "true";
5  add_header Access-Control-Allow-Origin "tfm.jediupc.com";
6    add_header Access-Control-Allow-Headers "Authorization,
       Origin, X-Requested-With, Content-Type, Accept";
7  add_header X-Content-Type-Options nosniff;
8  add_header Referrer-Policy "strict-origin-when-cross-origin
     ";
9  add_header Strict-Transport-Security "max_age=31536000;
     includeSubDomains; preload" always;
10 add_header Content-Security-Policy "default-src 'self';
     font-src 'self';img-src 'self' data:; script-src 'self';
      style-src 'self' 'unsafe-inline'";
```

Listing 12: Security HTTP headers.

```
1  proxy_hide_header X-Powered-By;
```

Listing 13: Removing X-Powered-By header.

The **X-Frame-Options** is used to indicate that a browser should not be allowed to render a page embedded in a site, in order to avoid clickjacking attacks. Clickjacking attacks consists in using multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top level page.

The header **X-XSS-Protection** blocks the browser if detects a Cross Site Scripting (XSS) attack. We disabled it because implementing would only result in undesired behaviours. Instead, there is a way better option which is the **Content-Security-Policy** header. This header helps to detect and mitigate certain types of attacks, including XSS and data injection attacks, but which better results and more control over the resulting behaviour.

In this header we set everything to self, meaning that we will only allow resources (javascript files, images, multi-media files, etc...) only from our domain. In addition to that, the **X-Content-Type-Options** header is set to *nosniff*, to indicate that the types advertised in the Content-Type headers should be followed and not to be changed.

The **Referrer-Policy** header is used to send information about the previous website. The recommended practice is to only send referrer information to other sites, and not within the application. Furthermore, we also add the **Strict-Transport-Security** header, also known as HTTP Strict Transport Security (HSTS) header. This header is used to inform browsers that the site should only be accessed using HTTPS, and that any future attempts to access it using HTTP should automatically be converted to HTTPS. We set the HSTS header to also apply to all subdomains, to be pre-loaded and always present in all responses.

The **Access-Control-Allow** headers define the configuration regarding CORS policy. The **Access-Control-Allow-Credentials** header tells browsers whether to expose the response to the frontend JavaScript code when the requests credentials mode is *include*. We want to allow this, since we will be expecting credentials in the requests for the authentication required endpoints. The **Access-Control-Allow-Origin** is one of the most important security headers, since it indicates whether the response can be shared with requesting code from the given origin. In our configuration is set to only allow sharing the responses with the "*tfm.jediupc.com*" origin, which is our application.This is done in order to ensure that non cross domain requests can see the responses of any requests made against our server. However, this does not stop the request from being made, its sole purpose is to not send the response back to the cross domain client. The **Access-Control-Allow-Methods** and **Access-Control-Allow-Headers** define the methods allowed and to indicate which HTTP headers can be used during the actual request, in CORS pre-flight requests. Since all requests will be made from within the domain, we will only allow GET request in CORS pre-flight requests, since it is the only method that is not a potential thread to our data. The allowed headers specified by the second header refer to some common metadata headers.

Finally, we remove the **X-Powered-By** header from all responses. This header is automatically added by one of NGINX modules, and it shows information about the technology used in the server. This is a clear example of Information Disclosure, since it is revealing to potential attackers the technology used in the server, allowing them to narrow down the research of vulnerabilities.

12.3 CSRF Tokens

An additional security measure implemented, is the use of Cross-Site Request Forgery (CSRF) tokens. A CSRF [37] token is a unique, secret and unpredictable value that is generated by the server-side application and transmitted to the client in such a way that it is included in a subsequent HTTP request made by the client. Since an attacker cannot determine or predict the value of a user's CSRF token, they cannot construct a request with all the parameters that are necessary for the application to fulfill the requirements of the request, therefore it can not construct a fully valid HTTP request suitable for feeding to a victim user.

As seen in the previous subsection 12.2 about the NGINX security headers, if an attacker performs a Cross Domain request, with a DELETE method for instance, to a valid url with valid parameters, the request will succeed. The Access-Control-Allow-Origin will not stop the request, it will just not send a response back to the attacker. In other words, the request will be passed to the server, handled and the requested resource will be deleted, if everything is correct. This of course is a problem, because even if the attacker is not able to know whether the request was successful or not since there is no response, the attack is successful regardless. CSRF tokens allow us to have a mechanism in our endpoints controllers to check whether a request is legit or not.

```
this.app.use(csurf({
    cookie: {
        path: '/',
        httpOnly: true,
        key: 'XSRF-TOKEN',
        domain: 'tfm.jediupc.com',
        secure: process.env.NODE_ENV === 'production',
        signed: process.env.NODE_ENV === 'production',
    }
}));
```

Listing 14: CSRF middleware protection

By default, CSRF tokens will only be checked in requests with methods that are not the HEAD or GET methods. We will leave this as default, since there is absolutely no issue in allowing get requests even from attackers, since it cannot modify the data. Moreover, the CSRF token is configured as a cookie, with the same configuration as the authentication cookies. We also set the cookie name to one of our choosing instead of using the default value, which would reveal information to an attacker about the technology used in the backend. The secret used for signing the cookie containing the CSRF token will be the same as the cookies for the JWT.

12.4 Type Guards and Type Sanitisers

As an example of sanitisers mentioned in section 11.3, the following couple of functions are used to sanitise primitive type inputs, and are used in all other validators as well:

```
export const sanitizeString = (input: string): string =>
   input.replace(/[<>\n\t]/g, "");

export const sanitizeObject = (input: any): void => {
    Object.keys(input).map(key => {
        if (input[key] && typeof input[key] === 'string') {
            input[key] = sanitizeString(input[key]);
            if (!isNaN(input[key])) input[key] = parseInt(
               input[key]);
        }
    });
```

```
};
```

Listing 15: Sanitiser functions for primitive inputs

The first function ensures that the characters that match the regular expression defined in the first parameter of the function replace, are replaced by an empty string. This way we avoid a string with white spaces, or HTML tags that can perform a XSS attack or a HTML injection attack. These functions are used in all validators to ensure that the values of the payloads not only fulfill the corresponding expected type but also to make sure that no malicious data ends being stored in the database.

Furthermore, we have implemented Type Guards as well, to ensure that all request body payloads are of the expected types, hence acting as validators.

```
export const isUserLogin = (instance: UserLogin): instance is
    UserLogin => {
    if (Object.keys(instance).length === 0) return false;

    const mandatoryTemplate: number = 2;
    const template: UserLogin = {
        email: "example@gmail.com",
        password: "templateString"
    };

    let isTemplate: boolean = true;
    let mandatoryAmount: number = 0;
    Object.keys(instance).find(key => {
        // if property does not exists
        if (template[key as keyof UserLogin] === undefined) {
            isTemplate = false;
            return true; // break loop
        }

        mandatoryAmount++;

        if (typeof instance[key as keyof UserLogin] !==
           typeof template[key as keyof UserLogin]) {
            isTemplate = false;
            return true; // break loop
        }

        if (key === "email" && !validator.isEmail(instance["
           email"])) {
            isTemplate = false;
            return true; // break loop
        }

        return false;
```

```
32      });
33
34      if (mandatoryAmount !== mandatoryTemplate) return false;
35
36      return isTemplate;
37  };
```

Listing 16: Type Guard function for user login type.

Inside all validators, the instance is examined in order to check that has exactly all the mandatory data fields required by the type, no less. In the example shown in Listing 16, we check that the instanced passed as parameter, contains only the mandatory fields defined by the *UserLogin* interface. If the instance is empty or has more fields that the ones required, is also rejected.

12.5 Azure Server

The server where the whole application is hosted is in Azure. Therefore, we have many tools to provide extra security. In our case, we decided to harden the SSH security, and apply a good firewall policy. Allowing SSH connections is always delicate, and the best case scenario, is only enabling ssh connections whenever is absolutely needed, and disable them afterwards. However, since we have a CI/CD pipeline that relies on SSH to be able to connect and deploy the application, that was not a possible solution. Instead, we made changes in the SSH configuration to try to harden it as much as possible to lower the risk.

```
1  # What ports, IPs and protocols we listen for
2  Port 52525
3
4  # Authentication:
5  PermitRootLogin no
6
7  PasswordAuthentication no
8
9  UsePAM no
10
11 ...
```

Listing 17: SSH configuration.

We only changed some attributes inside the configuration, but ones that really make a huge difference regarding connection security. First, the well known port for ssh service is 22, so we changed it to a random Port number out of the well known ports interval. Doing this will make that massive attacks targeted to SSH ports will not work in our machine, since it is no longer using port 22. Of course, an attacker that performs a targeted port scanning to our machine, will be able to see that the SSH service is running on port 52525, but we are already making things more difficult.

Furthermore, we disable root login in order to only allow SSH connections to unprivileged users. In case of a successful connection by an attacker, root will not be available as user to be connected via SSH. In addition to that, only connections with ssh keys are allowed, no passwords at all. This is a very substantial change, the most important one even. Disabling password authentication removes the possibility for an attacker to perform brute force attacks with passwords. In fact, the Github pipeline has two different ssh keys of 4096 bits length (being 2048 bits the minimum considered secure) for the frontend and backend deployments. PAM is disabled in order to run ssh in a non-root user, to strengthen security against privilege escalation.

Regarding the firewall configuration, the ACL is set to only allow packets with ports destination 80, 443 and 52525 via TCP. Since we only want packets to directed to either the application, or the SSH service and rejecting protocols like Internet Control Message Protocol (ICMP) (ping packets) as well.

Conclusions

13 Results and Discussion

In summary, we successfully built a functional e-commerce API application following all the appropriate procedures and good practices previously proposed in order to achieve a good level of security. In addition to that, we added additional security measures to not only the server, but the rest of components involved in the architecture as well.

Specifically, we applied functionalities, security mechanisms and other technologies to protect against all the vulnerabilities described in the OWASP Top 10. We researched about the best encryption algorithms and password hashing functions, as well as adding many security headers and numerous security configurations regarding logging or access controls to strengthen even more our application security and reliability. Moreover, we applied additional measures like the CSRF tokens that prevent malicious actions over our data, as well as type guards and type sanitisers to avoid potential malicious payloads, and have control about what data we are getting and whether is acceptable or not.

On the other hand, regarding infrastructure, we applied different security measures to isolate the different components of our app running on different docker containers, and a very specific control policy for database connections. Not only that, but also was added a strong firewall configuration and the SSH connection configuration was hardened significantly. The hardening of SSH connections was crucial, since we rely on that service for our CI/CD infrastructure and service availability.

However, in cybersecurity new threats arise every day, and what used to be secure eventually might not be it anymore. That is why we need to raise the minimum number of security mechanisms in every application, to make the Internet safer. This thesis provided a practical example and reference of the many different organisations involved in cybersecurity and standardisation, and all the information regarding security measures and good practices that is available about this topic. the world of Cybersecurity grows larger every day, and so do the threats and complexity of the attacks, in addition to the growing market for malicious techniques and malware.

14 Future improvements

As a future improvement, an authentication and authorisation system performed using Zero Knowledge Proof mechanisms might be very interesting. Zero Knowledge Proof is a protocol that allows to proof an statement to an entity, without revealing anything about the statement apart from the veracity of the statement. This requires a more complex setup, since requires managing certificates and perform client encryption in order to do so. However, many technologies are trying to implement this kind of protocols, to protect the users and protect their data as well, and can be a very good way to increase not only security but data privacy as well.

Acronyms

ACL Access Control List.

API Application Program Interface.

CDNs Content Delivery Networks.

CI/CD Continuous Integration / Continuous Deployment.

CORS Cross-origin resource sharing.

CRUD Create Read Update Delete.

CSRF Cross-Site Request Forgery.

CVE Common Vulnerabilities and Exposures.

CVSS Common Vulnerability Scoring System.

DDoS Distributed Denial of Service.

DoS Denial of Service.

FDD Feature-Driven Development.

FTP File Transfer Protocol.

GPU Graphics Processing Units.

HSTS HTTP Strict Transport Security.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

ICMP Internet Control Message Protocol.

IETF Internet Engineering Task Force.

ISO International Organisation for Standardisation.

IT Information Technology.

JPEG Joint Photographic Experts Group.

JWT JSON Web Token.

MVC Model View Controller.

NIST National Institute of Standards and Technology.

OAuth Open Authorization.

OSINT Open Source INTelligence.

OWASP Open Web Application Security Project.

PNG Portable Network Graphics.

REST REpresentational State Transfer.

RP Relaying Party.

SEO Search Engine Optimization.

SMTP Simple Mail Transfer Protocol.

SQL Structured Query Language.

SSL Secure Sockets Layer.

SSRF Server-Side Request Forgery.

TLS Transport Layer Security.

UI User Interface.

URL Uniform Resource Locator.

UUID Universally Unique Identifier.

VM Virtual Machine.

VPN Virtual Private Network.

W3C World Wide Web Consortium.

XP Extreme Programming.

XSS Cross Site Scripting.

Bibliography

[1] © 2022 Red Hat, Inc. *What is an API?* [Online; accessed March 2, 2022]. URL: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

[2] ©1998–2022 by individual mozilla.org contributors. *MVC - MDN Web Docs Glossary: Definitions of Web-related terms: MDN*. [Online; accessed March 2, 2022]. URL: https://developer.mozilla.org/en-US/docs/Glossary/MVC.

[3] Copyright ©2022 Educative, Inc. *What is rest?* [Online; accessed March 2, 2022]. URL: https://www.educative.io/answers/what-is-rest.

[4] © 2012-2022 Microsoft. *Documentation - typescript for the new programmer*. [Online; accessed June 4, 2022]. URL: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html.

[5] Copyright © 2022 Fortinet, Inc. *What is Black Hat Security? definition*. [Online; accessed March 2, 2022]. URL: https://www.fortinet.com/resources/cyberglossary/black-hat-security.

[6] The Daily Swig. *OSINT: What is open source intelligence and how is it used?* [Online; accessed March 7, 2022]. URL: https://portswigger.net/daily-swig/osint-what-is-open-source-intelligence-and-how-is-it-used.

[7] ©2022 Synopsys, Inc. *Application Security Blog*. [Online; accessed March 18, 2022]. URL: https://www.synopsys.com/blogs/software-security/top-4-software-development-methodologies/.

[8] © 2013-2021 Docker Inc. *Docker overview*. [Online; accessed June 4, 2022]. URL: https://docs.docker.com/get-started/overview/.

[9] © 2022 GitHub, Inc. *Understanding github actions*. [Online; accessed June 4, 2022]. URL: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

[10] © W3C. *About W3C*. [Online; accessed April 2, 2022]. URL: https://www.w3.org/Consortium/.

[11] © NIST. *About NIST*. [Online; accessed April 2, 2022]. URL: https://www.nist.gov/about-nist.

[12] © NIST. *Cybersecurity*. [Online; accessed April 4, 2022]. URL: https://www.nist.gov/cybersecurity.

[13] © 2022, OWASP Foundation, Inc. *About the OWASP Foundation*. [Online; accessed March 1, 2022]. URL: https://owasp.org/about/.

[14] © 2022, OWASP Foundation, Inc. *OWASP Top 10:2021*. [Online; accessed March 22, 2022]. URL: https://owasp.org/Top10/.

[15] © ISO All Rights Reserved. *About Us*.

[16] © IETF. *About*. [Online; accessed April 4, 2022]. URL: https://www.ietf.org/about/.
[17] © 1996-2022 The PostgreSQL Global Development Group. *PostgreSQL*. [Online; accessed June 9, 2022]. URL: https://www.postgresql.org/.
[18] ©2022 Educative, Inc. *What is SQL?* [Online; accessed June 9, 2022]. URL: https://www.educative.io/answers/what-is-sql.
[19] © 2022 Meta Platforms, Inc. *React – a JavaScript library for building user interfaces*. [Online; accessed June 9, 2022]. URL: https://reactjs.org/.
[20] © Copyright IBM Corporation 2012, 2022. *Openid Connect*. [Online; accessed June 4, 2022]. URL: https://www.ibm.com/docs/en/was-liberty/base?topic=liberty-openid-connect.
[21] Dick Hardt - Microsoft. *The oauth 2.0 authorization framework*. [Online; accessed June 10, 2022]. URL: https://www.rfc-editor.org/rfc/rfc6749.
[22] OktaDev. *OAuth 2.0 and OpenID Connect (in plain English)*. [Online; accessed June 10, 2022]. URL: https://www.youtube.com/watch?v=996OiexHze0.
[23] © 2013 - 2022 Auth0® Inc. *JSON web tokens introduction*. [Online; accessed June 10, 2022]. URL: https://jwt.io/introduction.
[24] Michael B. Jones, John Bradley and Nat Sakimura. *RFC 7519 - JSON web token (JWT)*. [Online; accessed June 10, 2022]. URL: https://datatracker.ietf.org/doc/html/rfc7519.
[25] © 2022 Okta. *HMAC (hash-based message authentication codes) definition*. [Online; accessed June 10, 2022]. URL: https://www.encryptionconsulting.com/education-center/what-is-rsa/.
[26] © 2018 – 2022 All Rights Reserved - Encryption Consulting LLC. *RSA: What is RSA?: Encryption consulting*. [Online; accessed June 10, 2022]. URL: https://www.okta.com/identity-101/hmac/.
[27] © 2018 – 2022 All Rights Reserved - Encryption Consulting LLC. *Elliptic Curve Digital Signature Algorithm (ECDSA) | Encryption Consulting*. [Online; accessed June 10, 2022]. URL: https://www.encryptionconsulting.com/education-center/what-is-ecdsa/.
[28] © 1999-2021 The OpenSSL Project Authors. *OpenSSL*. [Online; accessed June 10, 2022]. URL: https://www.openssl.org/.
[29] ©1998–2022 by individual mozilla.org contributo. *HTTP cookies - http: MDN*. [Online; accessed June 10, 2022]. URL: https://developer.mozilla.org/es/docs/Web/HTTP/Cookies.
[30] © ITU 2022 All Rights Reserved. *Universally unique identifiers (uuids)*. [Online; accessed June 15, 2022]. URL: https://www.itu.int/en/ITU-T/asn1/Pages/UUID/uuids.aspx.
[31] © 2013-2022 Auth0 Inc. *Hashing in action: Understanding bcrypt*. [Online; accessed June 22, 2022]. URL: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/.
[32] Alex Biryukov, Daniel Dinu, DmitryKhovratovich. *P-H-C/PHC-winner-argon2: The password hash argon2, winner of PHC*. [Online; accessed June 22, 2022]. URL: https://github.com/P-H-C/phc-winner-argon2.
[33] © 2022 PortSwigger Ltd. *Web Security Academy*. [Online; accessed June 22, 2022]. URL: https://portswigger.net/web-security/host-header.
[34] Charlie Robbins. *Winstonjs/Winston: A logger for just about everything*. [Online; accessed June 22, 2022]. URL: https://github.com/winstonjs/winston#readme.

[35] ©Copyright 2021 - CheatSheets Series Team. *Docker security cheat sheet*. [Online; accessed June 22, 2022]. URL: https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html.

[36] ©Copyright 2021 - CheatSheets Series Team. *HTTP security response headers cheat sheet*. [Online; accessed June 22, 2022]. URL: https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Headers_Cheat_Sheet.html.

[37] © 2022 PortSwigger Ltd. *CSRF tokens*. [Online; accessed June 22, 2022]. URL: https://portswigger.net/web-security/csrf/tokens.