Securing Flowcontrol

Internship Maarten Hormes

Limax

Horst

Date	**:**	**21/12/2022**
Version	**:**	**1.2**
Status	**:**	**Final version**
Author	**:**	**Maarten Hormes**

FONTYS UNIVERSITY OF APPLIED SCIENCES **HBO-ICT: English Stream**

Data student:	
Family name, initials:	**Hormes, M.**
Student number:	**456806**
project period: (from – till)	**29/08/2022 – 27/01/2023**
Data company:	
Name company/institution:	**Limax**
Department:	**ICT**
Address:	**Venrayseweg 126b**
Company tutor:	
Family name, initials:	**Peeters, R**
Position:	**Software engineer**
University tutor:	
Family name, initials:	**Dam van, A.**
Final report:	
Title:	**Securing Flowcontrol**
Date:	**10/01/2022**

Approved and signed by the company tutor:

Date: 10-01-2023

Signature: Rik

Version

Version	Date	Author(s)	Amendments	Status
0.1	05/09/2022	Maarten	Setup internship report	Initial draft
0.2	12/09/2022	Maarten	Recorded first research conclusions	
0.3	07/11/2022	Maarten	Added summary and conclusion from research document questions 1, 2, 3	
0.4	14/11/2022	Maarten	Added summary and conclusion from question 4 & 5	
0.5	17/11/2022	Maarten	Revised summary and conclusion -> updated structure after tips from Marielle	
0.6	24/11/2022	Maarten	Further refined research question results	First version
0.7	28/11/2022	Maarten	Updated report after feedback Marco. Added information in chapter 1, 2 and 3 from project plan.	
0.8	30/11/2022	Maarten	Updated glossary and moved preface after feedback Marielle.	
0.9	06/12/2022	Maarten	Processed chapter 1 & 4 feedback from Marielle.	
1.0	15/12/2022	Maarten	Finished processing last feedback. V1.0 achieved.	V1.0
1.1	19/12/2022	Maarten	Moved preface into chapter 1 and completed introduction. Finished chapter 5 and added personal evaluation. Finalized version with summary.	Pre-holiday final version
1.2	21/12/2022	Maarten	Added research methods to reflection, included reference to risk analysis for requirements, gave context to splitting of question 2, clarified some of my process steps (in chapters; 4.2, 4.4, 5), reformulated recommendation and supplemented evaluation.	

Distribution

Version	Date	To
0.6	24/11/2022	Marielle Fransen, gain first feedback
0.6	25/11/2022	Marco Hormes, gain feedback on report structure
0.7	30/11/2022	Marielle Fransen, gain feedback on updated chapters
1.0	15/12/2022	Marielle Fransen, gain feedback on chapter 4 after processing V0.7 feedback.
1.1	19/12/2022	Marielle Fransen, gain feedback on final version sent before the holiday starts (last version to gain feedback on before deadline)
1.2	21/12/2022	Alexander van Dam (replaced Marielle), show final version, last feedback chance

Table of content

Summary

This document was created during my internship period at Limax B.V. in Horst.
Here I worked with the development team on a system called Flowcontrol, created with the aim of digitalizing Limax' business process. My assignment during this internship was to research and improve the existing security implementation to a more secure option according to industry standards. Since the development of the system was still at an early stage, the security implementation was very limited and simple. Before the system could go live and have actual users, there is the need of improving this security implementation. This led to the main research question of my internship:

How to improve the security of the Flowcontrol system to secure the communication between microservices and the client?

To improve the security implementation, I first would need to understand what this entails. With this in mind, I created an analysis of the OWASP top 10 on the previous security implementation. This analysis helped me identify risks that need to be tackled with the new implementation. By using this analysis as a SWOT analysis, it became clear what parts of the previous implementation need to be updated. This analysis, combined with an interview with my conceptual tutor helped me and Rik (my technical guide) to formulate requirements for this new security implementation.

After I knew what needed to happen, the questions of how and with what technologies came to be. During my research I managed to answer these questions during sub-question 2: What is the best fitting framework/strategy for securing the Flowcontrol system?
Since at this point I knew what requirements are stated by Limax and the risks that a system like Flowcontrol carries, I could create an available product analysis. This analysis, combined with research into best, good, and bad practices led me to the conclusions that using a combination of Spring security and an implementation of the OAuth2.0 protocol would be the best fitting framework/strategy. This OAuth2.0 implementation became Keycloak after another available product analysis.

At this point it was clear what needed to happen to call the Flowcontrol system secure and what technologies I would be using. This led me into the continues process of trying to implement these new technologies. This process was a combination of literature studies, where I dove into documentation of the used technologies (like keycloak, its JavaScript adapter and Spring security), and code reviews. These code reviews were achieved by me creating test implementations of the new technologies and going over them with my internship guide. After we agreed on the test implementation I could implement it similarly in Flowcontrol.

A similar process was used while implementing the new security technologies in the front end. Since I never worked with Vue.js before this internship, it was hard to create a test implementation of a similar front end. This led to the fact that the code reviews that were done during the back end implementation were replaced by peer programming. This peer programming happened in the latest version of the front end and allowed me to implement the new security technologies with a guiding eye and helping hand from Rik. After these peer programming sessions, I was capable of working on and improving the front end independently.

Once the implementation of the authentication server, its connection to the back end and the communication to the front end were done, it was time for the final part of my deliverables. Being the centralized configuration server. After working with Flowcontrol for a few weeks, it became clear that a config server did not have the highest priority, nor would it be necessary. This conclusion was reached after discovering the Spring cloud gateway configuration options. The configuration on the gateway, combined with a centralized file configuration substituted the config server very nicely and helped me deliver on the agreed centralized security configurations.

Glossary

Possible abbreviation	Term	Meaning
API	Application programming interface	A collection of endpoints that form a secure opening in a back end to allow users/machines to interact with the code behind the API
CORS	Cross Origin Resource Sharing	The malicious act of trying to access protected resources from other origins (site)
CRUD	Create, Update, Read, Delete	An acronym used to describe the default functions of an entity in object-oriented development
SWOT analysis	Strengths, Weakness, Opportunity, and Threat analysis	An analysis defining the strengths, weaknesses and opportunities of a system or product.
CSRF	Cross Site Request Forgery	The malicious act of trying to edit protected resources from other domains (site), by inserting code
JWT	Json Web Token	A web token often used to store information about authenticated users in web applications.
KC	Keycloak	Provider of authorization server software. In this document used for both naming the company as the software they developed
OAuth2.0	Open Authorization 2.0	A protocol that defines the de facto authorization and authentication flows
Open-source web-application security project	OWASP	An open-source project created to inform developers about the most common and dangerous security risks.
RBAC	Role Based Access Control	Defining access rules for users based on their role.
N/A	Spring	A Java framework. Often used as default Java framework.

1: Introduction

This report concludes the internship of my 3rd year at Fontys HBO-ICT. The internship was held at Limax B.V. in Horst where I worked with the software development team. The internship was held over a period of 5 months.

Limax is part of a group (Limax Group) of cultivation and trading companies in the agribusiness in the Netherlands and Poland. They mainly work with mushrooms. The location in Horst where I am performing my internship is the middleman between Champignonland (the mushroom-producing company which is part of the Limax group) and the stores selling their products.

The overall goal of the internship was to support the development of a microservices based system called Flowcontrol. This Flowcontrol system was created with the intention to improve and digitalize the business process of Limax. My task was to research and improve the existing security implementation to a more secure option according to industry standards.

The following chapters of the report discuss the company, assignment, research, and conclusions in more detail.
Chapter 2 gives information about Limax BV, the company where I performed my internship, and my part in the company.
Chapter 3 goes over the assignment tied to this internship, the research involved and further related information.
Chapter 4 holds the research process & results, going over the how and why of my research findings and conclusions.
Chapter 5 concludes the research related section of the report, formulating a conclusion and further recommendations to Limax.
Lastly, you can find my personal process evaluation, followed by the appendices.

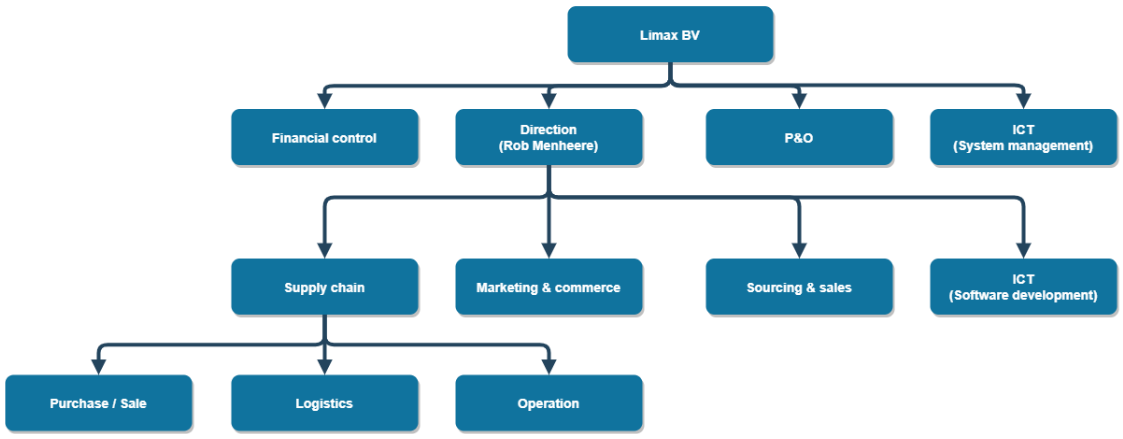
This report is written with my research documentation as main source. This research documentation, which can be found in Appendix A, is the place where I stored all the research during my internship. Referenced articles, sites and people are documented in the references list of the research documentation.
In this document, there will be little references since all information comes from my research document. Interested parties can use this research documentation to find the references used during my research.

2: About the company

Limax is part of a group (Limax Group) of cultivation and trading companies in the agribusiness in the Netherlands and Poland. They mainly work with mushrooms. The location in Horst where I am performing my internship is the middleman between Champignonland (the mushroom-producing company which is part of the Limax group) and the stores selling their products. The semi-finished products arrive at Limax in horst. Here the product is packed into the right sized container containing a certain amount of product (mainly different types of mushrooms). Each container gets a label containing product information and all information the store needs to sell the product (barcode, store name, etc.).

The finished product gets shipped to the stores. Some companies Limax works with are Lidl, Jumbo Edeka and Hello Fresh.

Limax is no IT company and there is no dedicated IT-department. Here at Limax there are currently 2 ICT employees and a third conceptual 'guide' that acts as the product owner of the software they are developing. Once of these IT employees covers the system management for the office and production halls. The other works as software engineer and is my technical guide during this internship.



Software…? You may ask. Yes, Limax is working on their own system. This system, Flowcontrol, is intended to digitalize their business process. This development is still in an early stage, especially since there is only 1 developer currently working on it. During my internship I will be joining the software development "department".

More information about the system and my assignment can be found in chapter 3: Assignment overview.

3: Assignment overview

Current situation

The last few years, Limax has the need to digitalize the business process. This goal was set to increase profit and simplify work. The system they are developing is called Flowcontrol. A few years ago, Limax started working with some interns and freelancers. These people created parts of this system, all using different technologies, design, standards, etc.

This resulted in a combination of programs that would cause more chaos and issues than it prevented. This made Limax choose to integrate the current software implementations to a centralized system. They chose to go for a Java back end (Spring boot) and Vue.js based front end. The back end was designed as a microservice based application.

The first part of this platform is currently under development since the first microservices have been created. Some basics of a few microservices have been completed, but the development process is still fully going on.

Opportunities

Since the team is very small, the priority of making a solid security implementation was very low. This creates the opportunity for me to research what is needed in order to create such a security implementation and to implement it.

The security of the Flowcontrol system at this point is very basic. This gives me the opportunity to research the most fitting way of securing the system and each separate microservice. This most fitting strategy can be implemented on a general level (configuration that covers each individual microservice) and if time allows it on one or multiple microservices.

Project goal

The goal of this project is to secure communication between microservices and the client of the Flowcontrol system.
To secure this communication, I am going to improve the current security implementation by implementing a centralized configuration (prevents e.g., code duplication, the need to make the same changes in different files, exposing sensitive information on multiple locations), change the current HTTP basic authorization flow to a more secure option (Limax prefers the usage of industry standards) and configure some (too little time to do all) endpoints to ensure only authorized users have access to them.

The security of this system will play a huge part in keeping all sensitive information secure. This means that no one can use any of the exposed endpoints without being authorized to use them.
If unauthorized users could access these points, that could result in a leak of sensitive company information, fake or incorrect data inserted by this unauthorized user, missing or corrupted data, etc.

To assure I can finish my assignment within the given time frame, requirements, constraints and deliverables have been drafted. These are shortly discussed on the next page. The project plan (appendix B) holds the complete discussion related to these sections.

Requirements
After discussing the possible research and assignment for my internship with Limax, they have constructed some requirements to follow. These are listed below:

- Ensure centralized security configuration and secret keeping.
- Use centralized configuration to provide general security for the article module.
- Configure roles and scopes for each of the article module endpoints.
- Springdoc-openapi documentation needs to be configured to use the new authorization flow.
- Front end (Vue.js) needs to support the new authorization flow.
- Identify the current risks in the Flowcontrol system.
- Configure security using a scopes strategy, meaning each endpoint will have their own authorization.
- The exclusive usage of widely used software solutions, e.g. frameworks, plugins, etc.

Constraints
For this project, there are some technologies that I will be using since Limax already decided to use them as the basis of their system.
This means that there are the following constrains:
- The back end is written in Java (Maven) using Spring Boot
- The front end of the application is an SPA written in Vue.js
- The back end is designed as microservices using Spring Cloud

Deliverables

This diagram holds an overview of what will be delivered during my internship. This can be categorized as a back-end security implementation, front-end security (RBAC) implementation and updated documentation according to the implementations coming from my research.

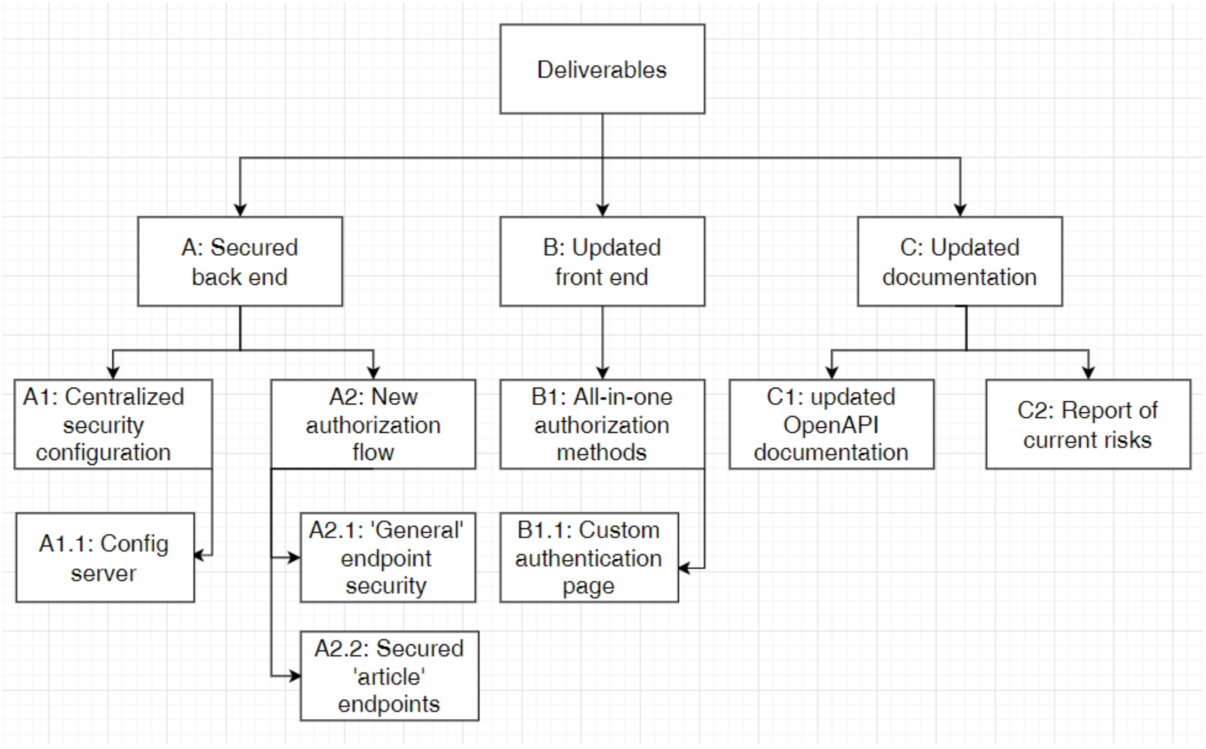

Deliverables
A: Secured back end
B: Updated front end
C: Updated documentation
A1: Centralized security configuration
A2: New authorization flow
B1: All-in-one authorization methods
C1: updated OpenAPI documentation
C2: Report of current risks
A1.1: Config server
A2.1: 'General' endpoint security
B1.1: Custom authentication page
A2.2: Secured 'article' endpoints

Figure 1: Deliverables

4: Process and results

This chapter of the report is used to describe the process and results of my research. It is structured around the research questions and explains how and why the conclusions and its supporting implementations of each sub-question came to be. Most of these questions have some technical details. All technical detail mentioned is used to describe how and why certain implementation came to be and will not be used to dive into each little bit of the implementation.
Each of these sub-questions contribute to answer the main research question:

How to improve the security of the Flowcontrol system to secure the communication between microservices and the client?

The sub-questions used to formulate an answer to the main question:

1. What are the requirements stated by Limax for securing Flowcontrol?
2. What is the best fitting framework/strategy for securing the Flowcontrol system?
    - 2.1 What are the current security risks (OWASP) that need to be prevented with this framework/strategy?
    - 2.2 What are the current industry standards that could be used to secure the Flowcontrol system?
    - 2.3 Which OAuth2.0 implementation is most suitable for securing the Flowcontrol system?
3. How to implement this (sub-question 2.2) fitting security industry standard in a Spring RESTful API?
4. What is an efficient way of centralizing security configurations between microservices?
5. How to ensure the front end is capable of using the new authorization flow?
    - 5.1 Which OAuth2.0 grant is best fitting for securing the communication between the vue.js front end and Spring back end?

Research approach

At the beginning of my internship, I defined the research approach per sub-question. This expected approach can be found as the first part of each of the sub-questions. This expected approach is supplemented by information on how I actually came to an answer and a conclusion to the sub-question in question.

Adjustments

As to be expected, there are adjustments made during the research. These adjustments came to be after discovering something previously unknown, having a better understanding of the client's needs or after simply revisiting a subject. After performing some research, it became clear that question 2 and 5 needed to be extended. This led to the addition of questions:
2.3: Which OAuth2.0 implementation is most suitable for securing the Flowcontrol system?
5.1: Which OAuth2.0 grant is best fitting for securing the communication between the vue.js front end and Spring back end?

4.1: What are the requirements stated by Limax for securing Flowcontrol?

Expected method: ***Library:*** Expert interview. ***Library:*** SWOT analysis.
Why these methods?: The requirements for Flowcontrol will be very specific towards Limax's needs. Talking with someone inside the company that understands these needs makes it easier to map them out. By making a SWOT analysis on the current security implementation I can map out the opportunities and threats with the current implementation to support the expert in formulating the requirements.

Before I can start to improve the security implementation of the Flowcontrol system, I would need to know what 'improving the security of the Flowcontrol system' means. Discovering what needs to happened can be done in multiple ways. Here I have conducted an expert interview. Since I was looking into a Limax and Flowcontrol specific situation, finding an expert wasn't hard. Both my internship coordinators here at Limax are the creators of the system. Since they have been working with Flowcontrol since practically day one, they have the required knowledge to express the security needs both short as long therm. From this interview and their knowledge within the Flowcontrol system, we drafted the following requirements for my internship assignment.

- Identify the current risks in the Flowcontrol system.
- Ensure centralized security configuration and secret keeping.
- Use this centralized configuration to provide general security (for the article module).
- Configure security using a scopes strategy, meaning each endpoint will have their own authorization.
- Configure roles and scopes for each of the article module endpoints.
- Springdoc-openapi documentation needs to be configured to use the new authorization flow.
- Front end (Vue.js) needs to support the new authorization flow.
- The exclusive usage of widely used software solutions. E.g., frameworks, plugins, etc.

Next to gaining insights from the expert interview, I have conducted a SWOT analysis on the previous security implementation using the OWASP top 10. The OWASP top 10 holds the 10 most common and dangerous security risks of the year. The analysis document itself can be found in appendix A of the research document. The following requirements came to existence after discussing these this OWASP analysis document with my coordinators.

- Use encryption for saving secrets and configuration values
- Use up-to-date and strong encryption/hash algorithms
- The usage of up-to-date and trusted libraries, plugins and modules
- Log logs in such a way they can be monitored (manual & automated message detection)

Sub-question 4.2.1 discusses the security risks of a system like Flowcontrol. The OWASP analysis was done to identify these risks. Some contexts on how these above named requirements came to be, can be found in the discussion of sub-question 4.2.1

4.2: What is the best fitting framework/strategy for securing the Flowcontrol system?

Expected method: ***Library:*** Best, good, and bad practices. ***Library:*** Available product analysis. ***Lab:*** A/B testing.
Why: By looking into the best, good, and bad practices of securing a system like Flowcontrol, I can find a fitting framework/strategy that is up to industry standards. In case there are multiple solutions, creating a small test implementation of both can help see which one suits best to the Flowcontrol system and Limax's requirements.

By answering this question, I can ensure that the security implementation that I will be creating is the best fitting option. This ensures the basis of my research and implementation during the remainder of my internship and prevents a necessary revisit of the subject.
Since this question is quite broad, there was the need of splitting it up in smaller sub-questions:

- **4.2.1 What are the current security risks (OWASP) that need to be prevented with this framework/strategy?**
- **4.2.2 What are the current industry standards that could be used to secure the Flowcontrol system?**
- **4.2.3 Which OAuth2.0 implementation is most suitable for securing the Flowcontrol system?**

This division in questions was created to ensure that the research results I would be presenting will fit the previous mentioned requirements. By knowing the security risks that come to play in a system like Flowcontrol I can ensure that the eventual framework/strategy is capable of securing the system against these risks. Because this OWASP list is an industry standard that is updated regularly, I know that the eventual solution will be up to date since it needs to secure the system against the latest, most recent, risks.
Another requirement coming from Limax is to use widely used software solutions, or industry standards, to secure the Flowcontrol system. This requirement led me to formulate a question to investigate these possible industry standards.

Question 4.2.1 discusses multiple security risks that must be prevented with implementing this new framework/strategy. These risks come from the OWASP top 10 and compare the current state of the system to the desired state of a fully secured REST API. The four biggest risks coming from this analysis are:

- Cryptographic failures
- Security misconfiguration
- Security logging and monitoring failures
- Broken Access control

When it comes to securing a back end, question 4.2.2 goes over 2 widely used security solutions. The following conclusion can be drawn from question 4.2.2: Using a combination of an OAuth2.0 implementation & Spring security would be the most suitable option for securing the Flowcontrol system. This OAuth2.0 implementation will manage the creation of the JWT tokens and storing user data. This no longer needs to be implemented in the Flowcontrol back end.
Even when using an OAuth2.0 implementation, we still need to possibility to define certain required roles or permissions per module/endpoint and configure CORS-protection. This will be done using Spring security. Spring security is a must here since Spring web does not have any security configuration options.
This combination offers the opportunity of covering most of the security risks present in the Flowcontrol system. Considering the time available for this project and the fact that both Spring

security and the usage of OAuth2.0 belong to the de facto standard of securing Spring applications, this seems like the best fitting strategy of securing the Flowcontrol system.

This leaves one question unanswered: What OAuth2.0 implementation is most suitable for securing the Flowcontrol system. Like sub-question 4.2.3 mentions: Keycloak.
Keycloak is the most suitable option since it is widely used, free and open source, supports a lot of different OAuth2.0 authorization grants, offers the option to customize a lot and has very nice out-of-the-box options. Compared to its competitor (Auth0) it is, for Flowcontrol, the better option.

4.2.1 What are the current security risks (OWASP) that need to be prevented with this framework/strategy?

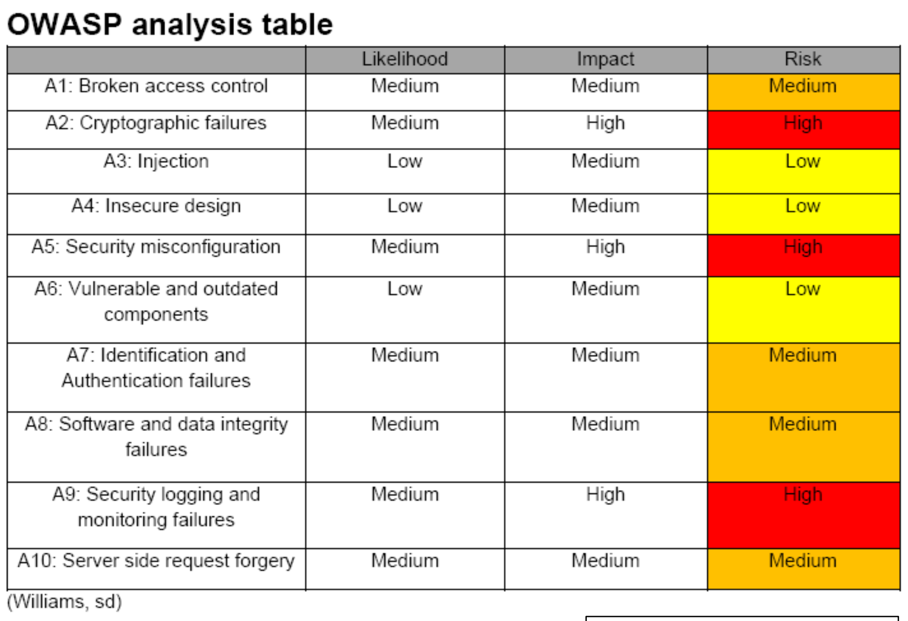
Creating a security implementation without knowing what to securing it from will be a tough challenge. For this reason, it was necessary to investigate the security risks of the previous security implementation of Flowcontrol and to gain knowledge on how to improve it.

For this I have created an OWASP top 10 analysis for the previous security implementation of the Flowcontrol system. As mentioned, the OWASP top 10 holds a list of the top 10 security risks coming into play with web applications. This list of risks gets drafted by analyzing and reviewing hundreds of web applications and projects. They check how well applications are secured against common risks, which risks are barely secured against, how popular and easy an exploit/hack/malicious action is. All this and more data get compiled to create a top 10 list of security risks.

This analysis can be found in appendix A of the research report. Since this report can be read in detail, I won't be going over each of the 10 OWASP items. Instead, I listed the four prioritized risks in the conclusion of sub-question 4.2. This prioritization is needed to ensure that the most important parts are covered before the application goes live.

Next to the goal of going live, there is a time restriction. The 18 weeks that my internship lasts are not enough to secure each and every bit of the system. By creating this prioritization, I can ensure I will be creating the most important parts first, and to finish these parts, instead of starting a lot of implementations without finishing any of it. I created this prioritization by creating an analysis table (figure 2). This table combines the likelihood and impact of a security fault, generating the overall risk of that specific top 10 item.

Looking at the analysis table (figure 2), you can see 3 items from the OWASP top 10 having a high risk. Logically, I concluded that looking into preventing these 3 items first would be the best course of action. Next to the 3 items with the highest risk, there is another risk that needs to be prevented before Flowcontrol can go live: Broken Access control. Broken access control is not only a security risk, but essential for the functionality of the system. Since multiple parts of the production process are integrated in the Flowcontrol system, multiple ''ranks'' of people will be using the system. A RBAC system would be necessary to ensure the expected functionality of Flowcontrol.



OWASP analysis table

	Likelihood	Impact	Risk
A1: Broken access control	Medium	Medium	Medium
A2: Cryptographic failures	Medium	High	High
A3: Injection	Low	Medium	Low
A4: Insecure design	Low	Medium	Low
A5: Security misconfiguration	Medium	High	High
A6: Vulnerable and outdated components	Low	Medium	Low
A7: Identification and Authentication failures	Medium	Medium	Medium
A8: Software and data integrity failures	Medium	Medium	Medium
A9: Security logging and monitoring failures	Medium	High	High
A10: Server side request forgery	Medium	Medium	Medium

(Williams, sd)

Figure 2: OWASP analysis

4.2.2 What are the current industry standards that could be used to secure the Flowcontrol system?

Like question 4.2.1 mentions, there are a few things to consider when securing (especially the back end of) a system like Flowcontrol.
To quickly summarize, there is a big need of filtering user input, to check if all incoming data is as expected, without malicious content. Before the code should even filter the input, it needs to check whether the user is allowed to access the application from that point or not. Next to these biggest items, there is the option to create a better security implementation by having centralized configuration, use encryption for secret keeping, have a proper logging structure, etc.

Securing the front end is, luckily, a bit easier. When it comes to securing the front end, there is a big need to validate all user input in order to prevent issues like injection or cross-site request forgery. Next to validating input, there needs to be some filtering going on, checking if the user trying to access a certain page of the application has the actual right to visit that page.

Since these 2 parts of the system need to be secured in different ways, I will be looking into different possibilities for both the back end and the front end. Since securing the back end is a bigger and more prioritized task than securing the front end, we, me and my coordinator, have decided to investigate the possibilities for that first. Once the back end has found a fitting security solution, I will be researching how to implement a fitting security solution into the front end.

Once you start looking into securing a Spring API, you'll find 2 things quite often. One being an implementation of Spring security, the other being an implementation of the OAuth2.0 framework.

This of course raises the question, what is Spring security? And what is the OAuth2.0 framework?

Spring security is a framework (part of the bigger Spring framework) that secures J2EE-based enterprise applications, by providing powerful, customizable security features like authentication and authorization.
Next to that, Spring security offers encrypted password storage, out of the box CSRF protection, brute force attack protection and Java configuration support.
Spring security is pretty simple to get into. The configuration can be done in a very minimalistic way, since Spring takes care of a lot behind the scenes.

When it comes to access control, Spring offers the possibility to filter access based on roles and authorities. In function practically the same, the difference being a 'ROLE_' prefix.

Currently Flowcontrol is secured using a very basic Spring security configuration. This configuration includes a simple algorithm to create and read JWT tokens, uses the password authorization grant and provides access based on the assigned role of the user. Even though there is some implementation of Spring security, which is more than it offers out of the box, this implementation is very basic. This means that even if using Spring security is the best option, it still needs to be configured to act as a solid security layer.

Another option would be to use/create an implementation of the OAuth2.0 framework.
OAuth2.0 is a framework / protocol, designed to allow a website or application to access resources on behalf of a user. "It replaced OAuth 1.0 in 2012 and is now the de facto industry standard for online authorization".

OAuth2.0 works as follows:
A middleman is used for the authorization process. This middleman is an implementation of the OAuth2.0 framework / protocol.
This middleman, often called the authorization server, stands in between the client (front end) and resource server (back end). When the client requests access to the resource, the authorization server comes into play, checks roles, permissions, and policies, and generates a token including these values. This token can be used to access the resource on the back end. There are a few different authentication flows (grants) that are supported by the OAuth2.0 protocol. This allows us to chose and use a grant that is best fitting to our situation.
There are multiple companies that offer software that can act as the authorization server in this OAuth2.0 protocol. This software runs as a standalone server (possibly in a docker container).

4.2.3 Which OAuth2.0 implementation is most suitable for securing the Flowcontrol system?

Before we can answer that question, we first would need to know what possible implementations there are. After looking into widely used OAuth2.0 implementations, I found these two coming back often:

- Auth0
- Keycloak

Both implementations can be described as User Management and Authentication tools.

Auth0 describes itself as: a flexible, drop-in solution to add authentication and authorization services to your applications. Your team and organization can avoid cost, time and risk that come with building your own solution to authenticate and authorize users.

On their website they continue describing some use cases where their software could be a suitable fit. I summarized the usage from these use cases to the following:

- Secure your API with the OAuth2.0 framework
- Allow users to login with both username password combinations as social accounts (Facebook, Google, Twitter, etc.)
- Securing communication between a JavaScript front end and an API (using OAuth2.0)
- Implement Single Sign-on
- Allow users to login using a onetime email or SMS
- Include multi-factor authentication
- Monitor users on your site. Use this data to improve flows, measure user retention, etc.

As you can see, Auth0 has a lot to offer when it comes to securing applications. When using Auth0 it is possible to give the complete authentication and authorization out of hands and let them manage it. Auth0 has a lot of very useful out-of-the-box options ready for you.
This does bring us to possibly the biggest issues with Auth0, its payment plan. There is a free trial to use when it comes to testing the usage of Auth0, but if this software will be used to secure the complete Flowcontrol system, there will be a point where not paying is no longer an option.

Since the discovery of this payment plan, I have been looking for an alternative free OAuth2.0 implementation that could offer the same as Auth0 when it comes to authentication and authorization. This search brought me to the previously mentioned Keycloak.
Keycloak is an open-source tool for Identity and Access management. Just like Auth0, it offers authentication and authorization solutions that developers can use to simplify implementations of frameworks such as OAuth2.0.
Some things Keycloak offers:

- Support for different protocols, being: OpenID Connect, OAuth2.0 and SAML 2.0
- Full support for OAuth2.0 grant types
- Full support for Single Sign-on
- A web-based GUI for easy configuration of the authorization server
- The possibility to use keycloak as a standalone user identity and access manager (authentication and authorization).
- Support for social identity providers such as; Google, Twitter, Facebook, etc.

Keycloak offers the possibility to give the complete authentication and authorization out of hands. It supports the usage of OAuth2.0 and its different grant types. The UI is a big plus, since it makes it easy to setup and maintain the server since writing code is not necessary. Another bonus about Keycloak is the fact that its open source and free. This means that there are a lot of libraries, plugins and solutions available that could benefit our situation.

There is one down-side to Keycloak. Being their outdated documentation. The outdated documentation happens since a lot of updates and new versions are released. This does make it harder to find the right solution inside the official documentation when something does not work. Luckily, Keycloak has a helpful community forum that helped me out a few times already. At this moment this is a risk I am willing to take, especially compared to all the benefits of using Keycloak.

As you can see, Auth0 and Keycloak can both be used to manage user identity (authentication) and access (authorization). Looking specifically at Flowcontrol (as this research intends to do) I recommend the usage of Keycloak. This is because it is completely free to use and open source. Meaning there is more possibility for customization, either by me, or developers who came before, that created solutions fitting to our situation. Since the software is free, it is more widely used. This results in more online available information, about the general usage and the issues people have come across.

4.3: How to implement this fitting security industry standard (Keycloak) in a Spring RESTful API?

Expected method: ***Library:*** Literature study. ***Showroom:*** Code review.
Why: Since this new security implementation will be according to an industry standard, I firmly believe and hope to find enough recourses online that will guide me through implementing the new security features into the Spring RESTapi. In case I get stuck and don't manage to find a solution, I can always ask my tutor to look at my implementation in the hopes of finding the reason I got stuck.

The following part will describe how to implement Keycloak in a Spring RESTapi and the reasoning behind the choices made, often coming from these issues. Since this was a continues process I won't be mentioning if I ran into the issues when creating a test or Flowcontrol integration. All implementations shown comes from Flowcontrol and reflects the current situation (unless mentioned otherwise).

Getting started: The first obvious thing to do is to add the Maven dependency to the project. This dependency needs to be added to use Keycloak inside Spring boot.
Adding this dependency introduces a new annotation that can be used to configure Keycloak. This annotation (@KeycloakConfiguration) includes Spring Security annotations, that normally are used to define the security configuration of Spring security. This means that when using this annotation, Spring finds the class and uses it as the security configuration. In this configuration we can configure access control, CORS, CRSF, etc. like we are used to in a security configuration class.

This configuration class was first created in the Article module, since this was the first module I implemented Keycloak in. This class uses the @KeycloakConfiguration annotation, which tells Spring boot that this class functions as the configuration class to handle incoming requests.
Next to that I setup an authentication builder, much like Spring security would normally need. This is needed to handle incoming requests holding tokens and to extract the data the token holds.
Lastly there is the configure method, again very similar to the Spring security configure method. At the beginning there are only a few accesses rules setup. These are needed for the API documentation, which I will come to at a later moment. Better access control is discussed later as well.

One thing to note are that both the CSRF and CORS options are disabled in the module's configuration. This is done since all request coming to the article module are redirects coming from the Spring cloud gateway. Since we are running the microservices in Docker, we can configure the docker compose file to only expose a port for each module inside the Docker network. Resulting in a situation where requests to the separate modules are only possible via the gateway.
This means that CRSF and CORS related configuration will happen on top level and are covered on a later stage (sub-question 4.4). Centralizing configuration prevents having a lot of duplicate code, spending time copying everything between the microservices and brings the risk of mismatching configurations between modules to live.

After adding the dependency and configuration class, there still is the need of adding some Keycloak related values into the application properties of the article module. This is needed since the back end wants to contact Keycloak on a request to ask if the access token supplied to the back end comes from Keycloak. These values can simply be added somewhere in the properties and some real Spring magic happens under the hood to make it all work.
This is basically all that is needed to generally include Keycloak in a back end module. There is still some other configuration that needs to be done. This configuration is specific to each module or even endpoint.
The controllers in the Flowcontrol system rely on a base controller. This base controller holds CRUD operations for all entities. Since all controllers inherit the same methods, we cannot define access

rules for each one of them. This caused an issue since certain CRUD operations need higher permissions (other roles) than others.

This issue made me switch to the usage of the antMatchers pattern. After conversing with my tutor, I have decided to create enumerations that hold the roles and paths of the Flowcontrol system. Using these enumerations, I can setup the fine-grained permissions that the @PreAuthorize could not supply. A snippet of the path and roles enumerations can be seen in figure 3 and 4 respectively.

```
public enum Paths {
    //region ARTICLE PATHS
    ARTICLE("/v1/articles"), ARTICLE_ID("/{articleId}"), PALLET_TYPE("/pallettype"),
    GET_NAME("/getname"), ALREADY_EXISTS("/doesalreadyexist/{excelCode}"),
    //endregion

    //region CASK PATHS
          ...
    //endregion
```

Figure 3: Paths Enum

```
public enum Roles {
    ADMIN("SUPER_ADMIN"), ARTICLE("ARTICLE_ADMIN"), MAINTAINER("MAINTAINER"),
    FARMER("FARMER"), SUB_FARMER("SUB_FARMER"), LOGISTICS("LOGISTICS"),
    PRODUCTION("PRODUCTION"), PLANNER("PLANNER"), ADMINISTRATION("ADMINISTRATION"),
    SALES("SALES");
```

Figure 4: Roles Enum

Next to creating the configuration, there is the need of making some changes to the current security implementation. One of these, is the swagger-UI API documentation (according to the OpenAPI specification). Swagger had to be re-written to use Keycloak and the new authorization flow. This was done to simulate the back end responses on requests from the front end. If swagger does not use Keycloak, you cannot rely on the sample responses swagger shows, simply since a whole security layer is skipped.
This was eventually achieved by adding a security scheme to the OpenAPI configuration class.

```
@Configuration
@OpenAPIDefinition(info = @Info(title = "Article service", version = "v1"))
@SecurityScheme(
        name = "Keycloak",
        type = SecuritySchemeType.OAUTH2,
        flows = @OAuthFlows(
                authorizationCode = @OAuthFlow(
                        authorizationUrl = "${keycloak.auth-server-
url}/realms/${keycloak.realm}/protocol/openid-connect/auth",
                        tokenUrl = "${keycloak.auth-server-url}/realms/${key-
cloak.realm}/protocol/openid-connect/token"
                )
        )
)
public class OpenApi3Config {

}
```

Figure 5: Swagger-UI config

Here we supply the needed information for Swagger-UI to connect to the keycloak server, request a token and show the API responses as they would be in the front end.
Swagger-UI does want us to add one last annotation. Being:
@SecurityRequirement(name=”Keycloak”)
This annotation is a class level annotation and tells Swagger-UI that all endpoints in the method need to use the Keycloak security scheme shown in figure 5.

4.4: What is an efficient way of centralizing security configurations between microservices?

Expected method: ***Library:*** Best, good and bad practices. **Library:** Community research.
Why: By once again looking into what has worked before in a similar situation can give good insights into the 'best' or most efficient way of centralizing configurations. Multiple 'lab' research methods can fit this question since different solutions can be tested to see which one fits the need of Limax best.

When it comes to centralizing configuration, there are multiple surfaces to touch.
Some configuration can be centralized in the sense that values from an global file (like an .env) are used by all microservices. This centralization simplifies the process of changing these values and reduces the chance of making mistakes like forgetting to update one specific microservice or making a type, breaking some functionality of the application. An example of this such an .env file can be seen in figure 6.

```
#Client
KEYCLOAK_URL=http://keycloak-service:8180
KEYCLOAK_REALM=Flowcontrol
KEYCLOAK_CLIENT_ID=flow-control

#Gateway
GATEWAY=test
GATEWAY_HOST=127.0.0.1
GATEWAY_PORT=8762
EUREKA_PORT=8761

#Services
TRANSPORT_PORT=7072
ARTICLE_PORT=7078
FARMER_PORT=7071

#Database
DATABASE_ROOT_PASSWORD=root

DATABASE_DEV_PORT=3386
DATABASE_DEV_INIT_VOLUME=./init
DATABASE_DEV_VOLUME=./docker-volumes/database/mariadb/dev/
DATABASE_DEV_USERNAME=user
DATABASE_DEV_PASSWORD=user
```

Figure 6: .env file

In this snippet we can identify another security risk I am looking into. Being: Security misconfigurations. Here we can still see that the development database uses a very simple username password combination. It shows another reason why centralizing configurations like these is important. If we every were to change this password, we would only have to update it in this file. Once the env values are updated, all microservices will use the new username and password. With this we tackle both the centralized configuration as the security misconfigurations.

A top-level configuration (a configuration on the Spring cloud gateway) has been mentioned a few times already. This configuration can be seen in the code snippet of figure 7.

```
cloud:
  gateway:
    default-filters:
      - DedupeResponseHeader=Access-Control-Allow-Credentials Access-
Control-Allow-Origin
    globalcors:
      corsConfigurations:
        '[/**]':
          allowedOrigins: "http://localhost:8080"
          allowedMethods:
              - GET
              - POST
              - PUT
              - DELETE
              - OPTIONS
          allowedHeaders:
              - Origin
              - Authorization
              - Content-Type
              - Accept
```

Figure 7: Top-level configuration

We can see that the globalcors configuration is applied to all paths. We only accept requests coming from the same domain or localhost:8080 (which is where the front end resides).
For the allowed methods, we only use CRUD operations and open the OPTIONS method for the preflight requests.
When it comes to the headers, we allow the basics. Origin is needed since we setup CORS configuration. The Authorization is needed for sending JWT tokens and the Content-Type and Accept header are used to ensure we receive data in a format that we expect.

The next configuration to investigate is CSRF and XSS attacks. After some research I discovered that it seems like I need to decide to either protecting the system against CRSF attacks or XSS attacks

This is a choice no developer wants to make. In my further research to decide which of these JWT storing options would be most suitable for securing the Flowcontrol system, I discovered an online discussion going on between developers siding with either of the solutions.
While reading into this discussion I discovered the following:

- Both local storage as Cookie storage are vulnerable to XSS attacks. For local storage this is due to the logical reason of the JWT being accessible through JavaScript. Cookie storage is vulnerable, since if a ‘hacker’ could inject JavaScript into our application and send a request to our back end, the cookie will be included in the request, thus not defend against this XSS attack. The JWT value in the cookie would not directly be accessible, but can still be used in the same way as an XSS attack would look if the system used local storage
- CSRF protection can be supplied in more ways than just using local storage. A key part of a Cross Site Request Forgery attack is the cross site of it. CORS, Cross Origin Resource Sharing, configuration defines how our application will handle requests coming from a different origin. It basically opens a door in the Same Origin Policy (SOP). It tells the browser which domains have access to the response coming from our back end. This does not mean that CSRF is no longer possible but makes it impossible for attackers to steal your credentials (JWT) from a response of the back end. This response can only be read by the same domain (opening the XSS danger).

From my research into protecting the system against CSRF and XSS I can conclude that local store is the most fitting storage for keeping our JWTs. Using local storage will protect the application against

CSRF attacks. When looking into XSS, neither local storage nor cookies will offer protection in case a ‘hacker’ is capable of injecting JavaScript into our application.

Once again is this centralized configuration a good way of preventing security misconfigurations. After a few weeks I already noticed that when working with microservices, it is very easy to lose overview. Especially if something structural changes, and these changes must be made in multiple microservices.
E.g., CORS, when I first created the CORS filter, I forgot to allow OPTION requests to the article module. Since I was sure I updated the CORS filters on each of the microservices, I was left helpless when the article service stopped working properly.
Once I moved the configuration to the gateway I came to the realization the article module missed this allowedMethod. If I centralized this configuration at the beginning, it would have saved me time and a lot of effort.

A final note:
At the beginning of my internship, I drafted my deliverables with my internship coordinator. At that moment it seemed like creating a config server would be the best fitting solution to centralize security configurations. The goal that this config server was supposed to achieve, centralizing security configuration, is already achieved by using the .env file and the gateway configuration. This would mean that adding this config server would only cost a lot of time, add unneeded complexity to the project and eventually replace a properly functioning part of the system. This means that after a conversation with my internship guide at Limax, I have concluded that delivering a config server would no longer be part of my assignment.

4.5: How to ensure the front end is capable of using the new authorization flow?

Expected method: ***Library:*** Literature study. ***Library:*** Usability tests
Why: Since the new authorization flow, coming from the new security framework/strategy, will be part of an industry standard, there should be more than enough resources online to find guidance in adjusting the front end to support this new flow. Once an implementation has been made, it can be tested on usability to ensure the easy use of the front end, meaning that the integration to the new flow went seamless.

To call the entire system secured, an integration in the front end needs to be made as well.
During the previous research I have mentioned OAuth2.0's authorization grants/flows. These flows describe how a client can gain an access token from the authorization server. Sub-question 4.5.1 dives deeper into these flows and explains why the front end of the Flowcontrol system will be using the Authorization code + PKCE grant.

This Authorization code + PKCE grant is supported by Keycloak, making it quite easy to implement it into the front end. The following part will go over this implementation. The assumption is made that the reader is informed about the functioning of the authorization code + PKCE grant (explained in sub-question 4.5.1).

My research led me to the discovery of a Keycloak-JavaScript adapter. This adapter would make it possible to create an instance of the adapter, configure it so it can communicate with the Keycloak server and manage the login/logout redirects, exchanging the authorization code for the access token, storing the token, and making the refresh request when a token is expired.

During the configuration of this adapter, we can use the "InitOptions" to choose the method used for the PKCE part of the grant. Adding this method to the configuration is enough for the adapter to know it should use the authorization code + PKCE grant.
This makes the integration a lot easier, especially since the adapter creates the code verifier and challenge.

Now that we have an instance of the Keycloak adapter, we need to be able to access it from "all'' components and classes. Since the instance gets exported, each class that would need keycloak can simply import it.
Simply importing this instance in all classes seemed like a solution that as just too easy. But it wasn't. After testing the current implementation manually, I have not run into issues that have to do with the reactive state of the Keycloak adapter instance.

Since I have successfully implemented Keycloak in the front end, we can start using if to define RBAC (Role Based Access Control) in the front end.

The front end has an access control class and just like the name says, is this class responsible for the access control.
In here I built a system to manage access token. There is a filter that checks if a user holds an access token, which is not expired before, letting them continue to the router. In case an expired token is found, a request to refresh it will be sent. I created a flow diagram to support the development of this access filter. The final version of the diagram is available for reference in appendix C of the research document.

Next to filtering the token, this class holds the access rules. A snippet of can be found on the next page. When looking at this code snippet, you might notice a few rules set-up before a user can

access a page. All the pages are only available for authenticated users (only /home is available without authentication). This means that unauthenticated users get send back to the home page. Next to being authenticated there is a possibility of requiring a role. Vue router works wonderful since I can get the META data from the route, which is defined in a different class file. In this META data I added roles, like “ROLE_ADMIN” and “ROLE_USER”.
Once a request is made to open a route that holds these required roles, the system will first check if the user holds these roles. If this is not the case, the user gets presented a screen that permissions are missing. The flow I just described can be found in the previously mentioned access control flow diagram (see appendix C of the research documentation)

4.5.1: Which OAuth2.0 grant is best fitting for securing the communication between the vue.js front end and Spring back end?

Keycloak lets you choice between different OAuth2.0 grants to use when it comes to requesting an access token. A grant is another word for a flow, a flow that is executed to request and gain an access token. These flows are created for different situations and can support the safety of you application by choosing the correct one.
OAuth2.0 offers a few different grant types being:

- Implicit
- Authorization code
- Authorization code + PKCE
- Client credentials
- Password
- Device authorization grant

The **authorization code + PKCE grant** is an extension to the authorization code grant.

To understand how this is an extension to the authorization code grant, let's investigate the authorization code grant flow. A sequence diagram of this grant can be found in appendix A: the research document.
With the authorization code grant, the user requests to login like they always would. Instead of sending their username and password and receiving an access token on successful login, there is a step in between. After the user supplies their credentials, and the login is successful, the authorization server returns an authorization code. The client (the JavaScript adapter in our case) send this authorization code, combined with the client id and secret to another endpoint on the authorization server. Here we check if the authorization code is valid and if the id and secret are correct. In case all checks are cleared, the server returns an access token our user can use to access the back end. This last step is also included in the sequence diagram but is not part of the authorization code grant. The grant ends when the user receives an access token.

After understanding how the authorization code grant works, we can look at the extension, being PKCE. Figure 8 is a sequence diagram visualizing the process explained next. Even though the flow of these grants is similar, there are some differences.
The user supplies their credentials. The front end creates a so-called code verifier and code challenge. The code challenge is sent to the authorization server on authorization code request. The authorization server receives the request for the authorization code and stores the code challenge. If the user credentials are correct, an authorization code is returned. This authorization code is the same as we have seen in the authorization code grant. Once the front end receives this authorization code, it sends a request to another endpoint on the authorization server. Instead of sending a client id and secret, the code verifier is sent along with the request.
This code verifier and challenge get compared (the challenge is a hashed and encoded version of the verifier) and if the validation goes successful, we get an access token back from the authorization server.

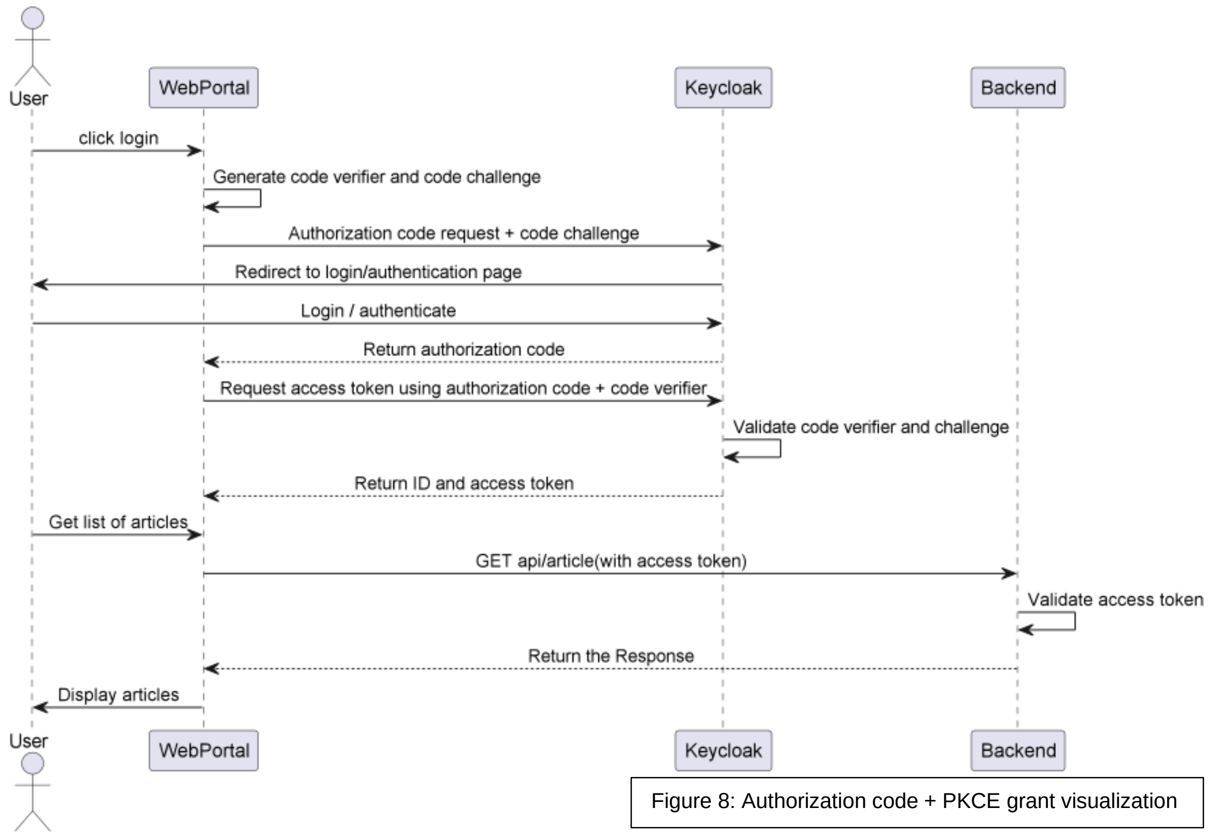

User
WebPortal
Keycloak
Backend
click login
Generate code verifier and code challenge
Authorization code request + code challenge
Redirect to login/authentication page
Login / authenticate
Return authorization code
Request access token using authorization code + code verifier
Validate code verifier and challenge
Return ID and access token
Get list of articles
GET api/article(with access token)
Validate access token
Return the Response
Display articles
User
WebPortal
Keycloak
Backend

Figure 8: Authorization code + PKCE grant visualization

After looking at both flows it should be obvious why we cannot use the authorization code grant with a public client but can use the PKCE one. This is simply since the single page application cannot hold secrets. This makes the whole authorization code exchange useless, since a 'hacker' could simply check the code of the web page and discover the secret.
The PKCE string (code verifier) gets created again on each login request. Even if the request holding the authorization code gets intercepted, without having this code verifier you cannot get an access token.

This makes the authorization code + PKCE grant the most suitable OAuth2.0 flow for the Flowcontrol system.

5: Conclusion and recommendations

This report, my research documentation and complete internship were al created and performed to answer the question:

How to improve the security of the Flowcontrol system to secure the communication between microservices and the client?

Flowcontrol is a Spring boot REST API back end combined with a Vue.js front end. To call the system secure, both the back end and front end need to be secured.

The Spring boot back end can make use of Spring security. The usage of Spring security allows me to create configurations when it comes down to CORS, CRSF, and RBAC. This configuration is, wherever possible, centralized in either the Spring cloud gateway or the centralized .env file. This centralization solves the issue of security misconfiguration over multiple microservices and increases the maintainability of these centralized configurations.
By using a OAuth2.0 authentication server from keycloak, I can give the whole authentication and authorization flow out of hands, including the creation of the code verifier & challenge. This means that the keycloak server will manage all users, user information and user authentication. Once we acquire a JWT and use it to make requests, the back end will use the authentication server to check the legitimacy of the JWT.

Compared to the previous security implementation, we can now easily configure the back end RBAC rules since all roles and users are easily managed via the Keycloak portal. The back end no longer uses a basic password authentication grant and is better secured against token forgery by using the authentication code + PKCE grant. By centralizing some of the configuration, I prevented future misconfiguration across multiple microservices (since they all use the same) and increased the application’s maintainability.

The Vue.js front end is secured using the keycloak JavaScript adapter. This adapter helps me implement keycloak in the front end immensely. It takes care of creating and communicating the code verifier and code challenge to the authentication server and manages the JWT’s coming from it, as the refreshing of the tokens on expiration.
This keycloak adapter gets used by the access control and API service in the front end. These use the stored values to check authorization, access tokens, refresh token, the authentication server address, and a lot more.

The communication from the front end to the back end is better secured by the usage of the authorization code + PKCE grant and is now easily configurable by the adapter. RBAC in the front end is now easy to configure and align with the rules of the back end.

Currently, the RBAC configuration is only performed on the article module. These configuration steps need to be repeated for the other microservices. I recommend Limax and Rik to invest time in creating this RBAC configuration in order to complete the securing of the Flowcontrol system.
Keycloak is already implemented over the whole application but won’t be of any use if the back end does not care about authorization.
This update to the overall RBAC needs to happen in the front end for the same reason.
Changes in security implementations are needed to keep up with the latest progress in the cybersecurity field. The current implementation is easy to maintain and should be checked against risks and issues at least once a year (upon new OWASP top 10 release).

6: Evaluation

Looking back at 5 months of internship, I feel proud and happy. After a very disappointing 6 months of creative technology, I feel way better during this semester. Not only do I feel better, but I learned bit by bit to talk about my mental state and take rest whenever needed (physically or mentally). This is for me one of the biggest personal growths I was able to experience during this period. This time last year I fell into a hole which was hard to climb out. I am proud of myself to stand here now and be capable of finishing this semester and internship even though it has not been easy, mentally, and physically. I have grown as a human being!

On a technical level I learned a ton. Not only did I have to dive into cybersecurity, which is normally something outside of my interest, I also helped Rik with the development of other Flowcontrol features. I discovered an interest in security I did not have before I started my research.
I learned a lot about general API and front end security, how to secure Spring boot applications, how to work with microservices, got a better grasp on docker's functionality and usage, learned a new front end language and much more.

When it comes to my research, I can conclude that my expected way of working (expected research methods) was correct. As can be seen in each sub-question, and the summary as well, I ended up performing my research with the DOT-methods I first predicted to be using. This is, for me, a sign that I can properly estimate what research (methods) would fit different type of questions. Next to that it shows that I had a proper grasp of the assignment I would be performing when drafting these expectations. Looking at improvements for next projects, I would advice myself to evaluate the expected methods before starting my research. My expectations could always be off, and even though I ended up with fitting research methods this time, I did not question my own predictions before I started performing my research. In case of a bad prediction, this could cost me a lot of time and 'wasted' research to some extent.

I am glad to say I have felt part of the Limax team since the first weeks of the internship. This was, next to my personal and technical growth, one of the goals of my internship. By working with other developers, I feel like I have learned to work and research more efficiently. This was achieved by seeing how my guidance at Limax sometimes ran into unknown issues or breaking parts of the application. By seeing how others handle a situation where they feel stuck I now know better how to formulate internet searches, how and when to debug and how to write easier to debug code.

Personally, I feel like this is best represented in my keycloak implementation. Creating this implementation, ensuring the configuration works, integrating it in docker and eventually on the production machine was a challenge. Here I got 'stuck' with this implementation a lot of times during my internship. By asking for help online (stack overflow and keycloak community) I managed to better understand the issue at hand and how to resolve it. Next to that I used my new technical skills to easily debug the keycloak server when I ran into bugs and issues.

All in all, I am very satisfied with my performance, the results achieved, the guidance from Limax, and the confirmation that I would enjoy working full-time in a small software development team!

Appendices:

A: Research report

Research report

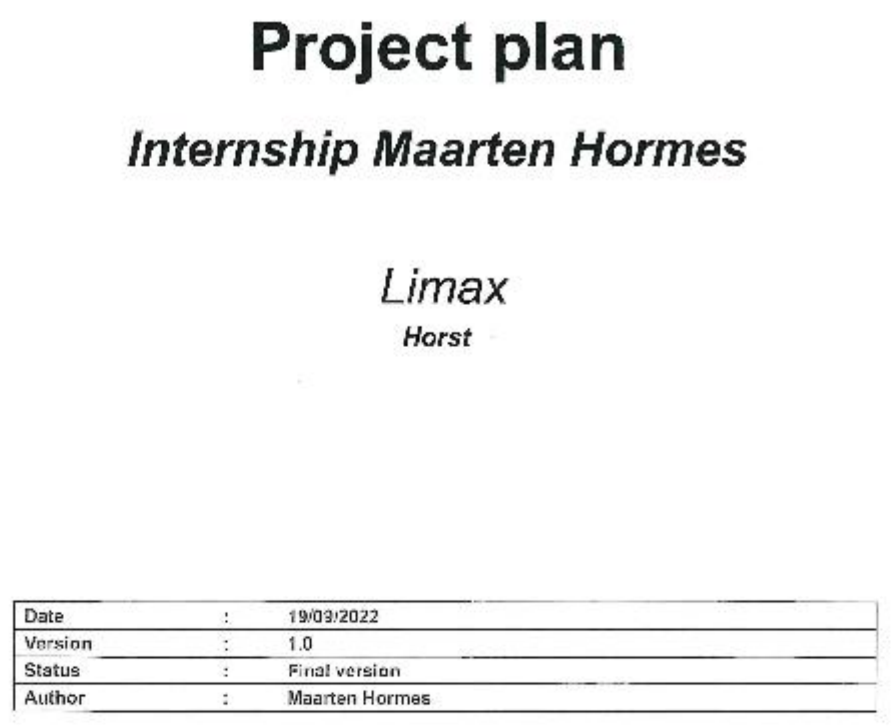
Maarten Hormes

Limax

Horst

Date	:	14/11/2022
Version	:	1.1
Status	:	First version
Author	:	Maarten Hormes

B: Project plan



Project plan

Internship Maarten Hormes

Limax

Horst

Date	:	19/03/2022
Version	:	1.0
Status	:	Final version
Author	:	Maarten Hormes