

Final Report

|  |
| --- |
| School of Computing  Faculty of Engineering AND PHYSICAL SCIENCES |

Securing microservice-based healthcare applications in Kubernetes

Danielle Kushnir

Submitted in accordance with the requirements for the degree of  
MSc Computer Science

**2022/23**

**COMP3931 Individual Project**

The candidate confirms that the following have been submitted*:*

|  |  |  |
| --- | --- | --- |
| **Items** | **Format** | **Recipient(s) and Date** |
| *Final Report* | *PDF file* | *Uploaded to Minerva*  *(02/05/23)* |
| *Link to online code repository* | *URL* | *Sent to supervisor and assessor (02/05/23)* |

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) Danielle Kushnir

© 2023 The University of Leeds and Danielle Kushnir

# Summary

This project aims to deploy an existing application, comprised of a web service and a database, as a secure microservice-based application. The security features implemented are to follow healthcare regulations. This aims to investigate if Kubernetes would comply with different healthcare security standards. The project focuses on UK and US regulations, as they represent different strictness, where the US healthcare technical requirements are more specific than UK's.

Objectives:

1. The Kubernetes application complies with UK healthcare security standards.
2. The Kubernetes application complies with US healthcare security standards.
3. System security is improved (measured by scoring better in security tests).

The security requirements for healthcare in both countries and Kubernetes security recommendations were gathered and used to create a security feature checklist. The purpose of this checklist is to keep track of the progress and to provide a concise list of what has been implemented.

System penetration tests ran with the initial state of a Kubernetes cluster running the application and another after the security adjustments. It was found that its pass rate increased by 22% and the fail rate reduced by 50%.

In conclusion, it was found that a microservice-based application using Kubernetes can follow UK and US healthcare security standards and these requirements improved the overall security of the cluster. However, it was not made 100% secure due to some existing vulnerabilities.

# Table of Contents

Summary iii

Table of Contents iiv

Chapter 1 Introduction and Background Research 1

1.1 Introduction 1

1.2 Background Research 1

1.2.1 Healthcare Systems 1

1.2.2 Kubernetes in Medicine 3

1.2.3 Kubernetes Security 3

1.2.4 Similar Work 4

Chapter 2 Methods 6

2.1 Architecture 6

2.1.1 Cluster 6

2.1.2 Nodes 7

2.1.3 Application Overview 7

2.2 Management 8

2.3 Development 9

2.3.1 Sprint 1 9

2.3.2 Sprint 2 9

2.3.3 Sprint 3 9

2.3.4 Sprint 4 10

2.3.5 Sprint 5 10

2.3.6 Sprint 6 10

2.3.7 Sprint 7 10

2.3.8 Sprint 8 12

2.3.9 Sprint 9 12

2.3.10 Sprint 10 12

2.3.11 Sprint 11 13

2.3.12 Sprint 12 13

2.3.13 Sprint 13 13

Chapter 3 Results 14

Chapter 4 Discussion 18

4.1 Sprint 1 18

4.2 Sprint 2 18

4.3 Sprint 3 18

4.4 Sprint 4 18

4.5 Sprint 5 19

4.6 Sprint 6 19

4.7 Sprint 7 19

4.7.1 Limit Resource Usage 19

4.7.2 Container Privileges 20

4.7.3 Audit Logging 20

4.8 Sprint 8 21

4.9 Sprint 9 22

4.10 Sprint 10 22

4.11 Sprint 11 23

4.11.1 TLS Communication 23

4.11.2 Backups 24

4.12 Sprint 12 25

4.12.1 Role-Based Access Control (RBAC) 25

4.12.2 Attack Simulation 27

4.12.3 Vulnerability Scanning 27

4.13 Sprint 13 28

Chapter 5 Conclusions 30

5.1 Conclusions 30

5.2 Ideas for future work 30

List of References 181

Appendix A Self-Appraisal 37

A.1 Critical self-evaluation 37

A.2 Personal reflections and lessons learned 37

A.3 Legal, social, ethical and professional issues 38

A.3.1 Legal Issues 38

A.3.2 Social Issues 39

A.3.3 Ethical Issues 39

A.3.4 Professional Issues 39

Appendix B External Materials 370

Appendix C Security Features Materials 41

# Chapter 1 Introduction and Background Research

## 1.1 Introduction

Security is essential to healthcare due to dealing with sensitive and identifiable patient data. If this data leaks out, it could lead to numerous types of attacks, including identity theft. Strict security regulations were introduced to healthcare systems to prevent attacks involving this data and to ensure that all healthcare systems follow a specific minimum standard. These standards are usually applied to a specific country. However, they can also apply to a few countries, such as with the European GDPR for countries in the European Union.

Deploying a system as a group of individual services working together rather than the traditional single application leads to different ways of viewing security. The common types of applications are monoliths, where a service is deployed as one single application. In contrast, a microservice-based application has many applications working together to provide a service. Naturally, microservices have more attack surfaces than a monolith application (which has just one attack surface). However, the attack surface in a monolith application is much larger since the entire architecture is comprised of that single application. The larger the attack surface, the more likely an attacker can exploit a vulnerability (Souppaya et al., 2017). On the other hand, a microservice architecture is made up of potentially smaller independent services making it have many smaller attack surfaces. Independent services working together to produce an application make it easier to isolate which service has a security issue since the services are separated. In a use case such as healthcare, micro-service-based applications potentially offering better security have led to this investigation.

## 1.2 Background Research

The initial problems with the project are understanding whether the security functionalities of Kubernetes provide the requirements demanded by the regulations of a healthcare system and discovering what those requirements entail for the technical implementations. This chapter focuses on solving both those problems in separate subchapters.

### 1.2.1 Healthcare Systems

What is considered a “secure system” is vague, primarily when used in the context of a healthcare system. Healthcare systems tend to have strict regulations as they must collect personal information about many people.

UK regulations require a law enforcement system to comply with the Data Protection Act (2018), also known as the UK’s General Data Protection Regulation (GDPR). The relevant part of this legislation is the sixth data protection principle which asks for “appropriate security including protection against unauthorised processing and against accidental loss”. According to an official NHS publication, NHS organisations must comply with the Data Protection Act (Corporate Information Governance, 2019). Although it can be vague on what is quantified as “secure” for a healthcare system, the legislation does provide some requirements for the system: authorisation (protection against unauthorised processing) and backup and restoration of data (protection against accidental loss). It also advises that encryption should be used where appropriate. These standards are more relaxed as they aim to be flexible for many systems.

The regulations differ regarding the US, as medical systems must comply with the Health Insurance Portability and Accountability Act (US House, 1996) which provides “national standards” using the specified security rules. HIPAA is a very long document which covers many points. The relevant point for this project is the coverage of electronically protected health information by technical safeguards. According to an article by Maayan (2023), a big reason healthcare systems are not using microservice-based applications is compliance concerns.

HIPAA having strict security regulations also comes at an advantage because it lists precisely what safeguarding measures are required and suggests further addressable ones. Although Figure 1.1 states that encryption is not required, Apricorn (c2023) argues that it is. In the healthcare context, where compromised data can lead to severe security problems due to patient data easily identifying people which can lead to easy identity thefts and other crimes, encrypting patient data should be considered a required measure. The following quote from HIPAA about electronically protected health information states that it should be “rendered unusable, unreadable or indecipherable to unauthorised individuals”. It does not say directly that the data must be encrypted, but making data unreadable to unauthorised individuals is the purpose of encryption. (Lawton, c2023) (Arampatzis, 2021)Table

Description automatically generated

**Figure 1.1** Technical security requirements for systems covered by HIPAA, full document can be found in Department of Health & Human Services USA, 2007.

However, there is also a disadvantage to using HIPAA. Due to the nature of the law, it is subject to change in the future, which means that the end solution in this project may no longer be sufficiently secure for a future version of HIPAA.

Steger (2020) states that healthcare systems’ most common security problems are malware attacks and operating system patching. These are problems that Kubernetes can help with because the application is kept in a container as a cluster. Destroying and recreating clusters helps alleviate the pressure of constantly patching securities, updating operating systems and searching for hidden malware (Miller, 2021).

Even though the NHS does not provide strict regulations for securing patient data, there is an online toolkit called Data Security and Protection Toolkit which assesses a system to “provide assurance that they are practising good data security”. This could be a good tool for risk-assessing a system and investigating its compliance with NHS regulations. However, this project will not investigate this because it is not publicly accessible. (NHS Digital, c2023)

### 1.2.2 Kubernetes in Medicine

Utah-based CHG Healthcare (CHG Healthcare, c2023) uses containers to run its systems. The system works by assigning temporary medical staff to healthcare institutions. It has not yet had a security breach or a public scandal due to data leakage, which implies that the security is adequate. Furthermore, since it is a US healthcare company, it must comply with HIPAA’s strict regulations. It has managed to implement a secure Kubernetes system in healthcare that follows the necessary compliances, meaning that this is an example of a successful microservice-based application in healthcare.

IBM’s Watson Health Cloud is another example of a healthcare system that deploys applications with Kubernetes. This system provides analytics and AI solutions for working with health data. They ensure security with many methods, such as using Namespaces and special containers called sidecars. The Kubernetes master is configured to follow requirements which include limiting resource usage. They use a logging service called Logmet for audit logging in the system. The system has good approaches to security, implying that it is a successful case of Kubernetes and has also been functional for many years (Kapoor, 2017). However, there are some reports of this system retiring “without a direct replacement”. One of the problems it faced was failing to acquire the necessary expertise in the healthcare sector (Clark, 2022).

### 1.2.3 Kubernetes Security

According to an article by Gain (2018), the isolated nature of microservices makes them easier to defend and can prevent an attack from spreading sooner. Another advantage of microservices is that they have less chance of a cascading attack because each service in a microservice-based application is independent, therefore a vulnerability in one cannot be passed since it will only affect this individual service. A monolith is inherently coupled, so one vulnerability can affect the entire system (Bush, 2020) (Brodie, 2016).

In an article written by Golden (c2023), he says that a common way of attacking a system is by exploiting its vulnerability when software is not up to date on its security patches. Updating a monolith architecture is much slower, but patching microservices is easier and quicker. Furthermore, monolithic applications are a big mix of code, so it can be challenging to know how it works, which means that a security error in one may result in the same error in the other. This error might not be discovered because the team will likely not know another team’s code. Microservices make it easier to split the work and allow collaboration since work is done on individual services instead of contributing to the same complex one.

Kubernetes is self-monitoring and self-healing which helps minimise operational vulnerabilities by adding more automation (thus also minimising human error). According to a blog written by Miller (2021), Kubernetes “provide a secure framework for data protection” and may be able to give healthcare systems “the security and compliance they need”.

Images contain all components used to run an application, but the image may have vulnerabilities. A common reason for images to be insecure is when they are no longer up to date, so some vulnerabilities that would have otherwise been fixed appear. A containerised environment deploys an image, so if the image is insecure, this poses a risk to the whole environment.

Furthermore, images could contain malware, commonly found when using an image from an untrusted third party. These programs can be used to attack the cluster, leak data or provide vulnerabilities that can be exploited later.

Another interesting security concern in Kubernetes clusters introduced by Souppaya et al. (2017) is rogue containers. These containers are not correctly configured or put through vulnerability testing, making them more likely to contain vulnerabilities. A common reason for having rogue containers is testing during development. Temporary containers are created to test features during development. If these containers are not removed after use, they become rogue containers.

Using the Kubernetes documentation and contributions from other sources (TechWorld with Nana, 2022) (Jackson, 2020) (Souppaya et al., 2017), a security feature checklist was created, which can be seen in Table C.2. This checklist provides a way to keep track of what features the project has implemented. It combines the technical requirements from The Data Protection Act, HIPAA and good Kubernetes practices.

### 1.2.4 Similar Work

In a paper written by Esposito et al. in 2017, they propose some security requirements for healthcare-related data using a microservices approach. This paper comes to a similar conclusion on the benefits of using microservice-based applications for healthcare. It expands by going into how microservices could be used within the healthcare domain, for example, by looking at communication flows throughout the information system. They could have used this communication flow to identify attack surfaces in the system further. Three issues were discovered in their solution, one being preserving privacy during data breaches, in which required emergency access to the system could potentially contribute. For the work done in this project, this problem would breach one of the HIPAA technical safeguarding requirements. Moreover, this paper only discusses a select number of security requirements, whereas other similar works went into greater depth.

A study was conducted by Ianculescu et al. in 2020 to investigate using microservices as a better way of managing healthcare data. Similarly to this project, they use EU GDPR and USA’s HIPAA recommendations to create a list of security requirements and conclude that microservices have become more appealing for healthcare than monolith applications. As a research project, it did not practically investigate security features as done in this project and only provided requirements.

The most similar work was an investigation by Dissanayaka et al. in 2017, where they looked at whether Linux Singularity Containers would be a good choice for securing protected medical data. They also examined whether medical data stored in MongoDB will comply with HIPAA regulations, concluding that it fulfils some but not most. This paper detailed the possible attack surfaces in the application. They look at very similar security requirements covered in this project, such as authorisation, auditing and encryption. A problem with this solution is that it does not comply with all HIPAA requirements. Many of the HIPAA technical safeguards were not fulfilled.

None of the works looked at Kubernetes as the technology to provide a microservice-based healthcare application.

# Chapter 2 Methods

## 2.1 Architecture

The main design choice was whether to implement an image for the container for the web application or to use an existing one. The decision was to use an existing basic image that displays data from a database and allows a user to alter the database using the website, simulating the required communications in a healthcare system in a rudimentary form. This was chosen because the project should focus on the security aspects of Kubernetes and healthcare rather than the implementation of a testing image which would affect the depth of the security investigation. This section covers the architecture of the container working with a pre-existing image. (TechWorld with Nana, 2022)

### 2.1.1 Cluster

The entire application runs as one Kubernetes cluster with two nodes: Control Plane and Minikube. Every Kubernetes cluster is required to have at least one Control Plane node, as it is the master node which manages the Kubernetes cluster. The Minikube node is a worker node and has two deployments of the application (one for the web application and the other for the database) and will contain any other deployments for third-party applications. This cluster runs as a Docker container.

Initially, the plan for the architecture was to use only one node and have all the applications on there. However, this poses two issues. Firstly, the node may need more resources if it distributes to many Pods and these resource requirements may not be fulfilled by a single node. Secondly, having the applications work in the same node as critical areas of the cluster could lead to security issues if the attacker can enter the node from the application.

Diagram

Description automatically generated

**Figure 2.1** Diagram of the Kubernetes cluster.

### 2.1.2 Nodes

The Minikube node contains many other sub-units. Pods represent the worker processes and there are two main Pods, one for the web application and another for the MongoDB database. There will also be some temporary Pods for external services, such as for vulnerability testing, where a pod running that service will be created and destroyed when finished. Pods for third-party applications will also be stored on this node, for example Linkerd Pods. Each Pod has its Service which it uses for communication. The web application Pod has an external Service to access it through a browser. The node also contains a ConfigMap and a Secret, both storing configurations related to the database to help the applications operate.

The control plane node contains many important aspects of the cluster, including the etcd and API server. The API server enables interaction with the cluster and represents the master processes. The Command Line Tool provides a way of communicating with the API server. A user interface or API can be used instead of Kubectl, but Kubectl is the most powerful because it lets anything be accessed or configured in the cluster. Due to this reason, it was picked over other Command Line Tool options.

Diagram

Description automatically generated

**Figure 2.2** The main architecture of the Minikube node. Both Pods are in the ‘default’ Namespace.

### 2.1.3 Application Overview

The architecture has two deployed Services: the MongoDB app with internal Service and the example website with external Service. The MongoDB Service forwards requests to the MongoDB Pod, likewise with the web application Service and Pod. The ConfigMap stores MongoDB’s endpoint. The Secret stores MongoDB credentials (username and password which should not be stored as plaintext). The web application uses the Secret and ConfigMap to utilise the database when needed.

**Diagram

Description automatically generated**

**Figure 2.3** Application overview.

Shape

Description automatically generated with low confidence

**Figure 2.4** Request forwarding, using the MongoDB Pod as an example.

## 2.2 Management

A screenshot of a computer

Description automatically generated with medium confidenceThe configuration files and final report were properly managed using GitHub, a Git version control system. Commits were pushed after each sprint and each commit was as atomic and complete as possible. Not all changes to the project could be committed to GitHub, as many were changes to the configuration of the control node in the cluster itself rather than extra manifests that were applied. An example is editing the API server manifest file to add a new line containing a flag.

**Figure 2.5** Example of file version control.

This project's files contain comments explaining the configuration to make it more readable. They also contain references to any sources that contributed information or part of the configuration file.

The project was developed in sprints, where each sprint was given an amount of time according to an estimation of the task's difficulty. This way was done to avoid assigning too much time to small tasks. Each sprint had a specific goal for what to achieve, for example to implement a specific security feature within that timeframe. The report was edited after each sprint to detail what happened in that sprint, which meant the report was technically complete after the final sprint.

Progress was tracked using the security feature checklist. Features were colour-coded based on the current progress: green marking the feature as implemented, yellow as in progress and no colour for incomplete.

Graphical user interface, text, application

Description automatically generated

**Figure 2.6** Example of progress tracking.

Finally, throughout the sprints there would be a regular check where the web application service would be started and tested to see if the functionality remained the same. This was done to discover if additions from breaking the application.

## 2.3 Development

### 2.3.1 Sprint 1

The first sprint investigates general Kubernetes knowledge, such as what Kubernetes is and its components in a cluster, which helped clarify the application's design. Finding the example image to use for the project was done in this phase.

### 2.3.2 Sprint 2

Sprint 2 was a second research sprint to investigate security measures in healthcare systems and what sort of standards/compliances a system must have. UK and US compliances are in the objectives of this project, so these were the main compliances investigated in Sprint 2.

### 2.3.3 Sprint 3

Sprint 3 was the third research sprint to investigate how Kubernetes is currently used in healthcare. The main discovery was that although different from the standard, there are live systems deployed using Kubernetes. Though numerous, this sprint only looked at two examples of these live systems discussed in section 1.2.2.

### 2.3.4 Sprint 4

The final research sprint focused on security features offered by Kubernetes. The main goal was to assign Kubernetes features to each requirement/good practice composed in the previous sprints. The result of this sprint was the list of security recommendations in C.2.

### 2.3.5 Sprint 5

The aim of Sprint 5 was to deploy an existing image of a web application into a Kubernetes cluster. The first half of the sprint was dedicated to creating the architectural design following the structure of the image and the second half focused on configuring the application following the application’s architectural design.

Although the plan is to implement all the security features, test and discuss them, realistically it may pose a challenge due to time. Thus, the primary purpose of the checklist is to track what features have been investigated. Since the checklist is concise, it also lists what has been implemented in the project which can be used to determine if project objectives were met. The checklist created in this sprint can be seen in Table C.1. The filled checklist can be seen in Table 3.2.

### 2.3.6 Sprint 6

In this sprint, two vulnerability scanners scanned the cluster to provide an initial project security scan. This scan can be used at the end of the project to analyse how well the cluster has been secured by comparing it to the initial results and to fulfil the project's third objective. These scanners worked differently from each other and produced different types of results to give a more diverse set of results.

The first security application is called kube-bench, developed by Aqua Security. This tool checks if Pods in a Kubernetes container comply with CIS Kubernetes Benchmark.

The same company, Aqua Security, developed the second security application, called kube-hunter. This application checks for container vulnerabilities from known vulnerabilities.

### Sprint 7

This sprint worked on adding three security enhancements from the checklist to the cluster: limiting resource usage on the cluster, container privileges and audit logging.

These three security enhancements were all added in the same sprints because they were much smaller tasks than the other security features on the checklist, so they could be completed within the same timeframe.

The first enhancement was managing resource usage. Pods can be configured only to use a specific amount of a resource. The first step was to find the minimum amount of memory and CPU the Pods need. This was found by getting a detailed node description using the "kubectl describe node {name}" command.

Text

Description automatically generated

**Figure 2.7** The capacity of resources available to the “minikube” node.

Since there are no limitations, the node can use up to the capacity. Then we can check how much CPU and memory (RAM) Pods use from requests by looking further down the description.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 2.8** Resource usage for non-terminated Pods (thus kube-bench and kube-hunter are not included).

The second enhancement was restricting container privileges using Pod security contexts. As part of the pre-testing to check if Pods have too many privileges, we will use root in a Bash terminal inside a Pod.

Text

Description automatically generated

**Figure 2.9** Running a Bash terminal in the web application Pod while having full privileges and using the “id” command to get a list of the privileges.

The user ID (uid), group ID (gid) and groups are all root, which means access to all filesystems. However, it does not need these privileges. The Pods have been reconfigured so they run with a uid of 1000, gid of 3000 and are in supplementary group 2000. The shell in both Pods will no longer run as root.

For extra security, the following features have been added: configurations to ensure Pod runs as non-root, security context for containers which only allows it to run with uid of 10000 and access the filesystem with read-only mode. (Badawy, 2023) (Singh, 2021)

The resource limit and security context features can be found in the “webapp.yaml” and “mongo.yaml” configurations.

Audit logging was added to the cluster by creating two new files inside the control plane and editing the config to use those files (Olaogun, 2023). Both these files are stored in the /etc/kubernetes directory. The first file was a Pod policy configuration, which tells which actions should be logged and which should be ignored. A copy of this file can be seen in /External/audit-policy.yaml in the project repository. The second file is an “audit.log”, where the audit log data will be stored. To test the audit logging, a new Pod called “nginx” was created and it produced a log that can be found in /External/Example Audit Log.txt in the repository.

### 2.3.8 Sprint 8

Sprint 8 focused on only allowing the control plane to access etcd (by disallowing anything else in the cluster) and admission controlling.

There are several methods of securing etcd, but we will focus on securing it by preventing unnecessary access to etcd and encrypting it in transition (TLS communication) and at rest. Adding TLS communication here contributes to the general TLS communication goal in the checklist. The use of certificates for TLS in etcd communication can be configured in the etcd manifest.

Text

Description automatically generated

**Figure 2.10** Certification and key file for TLS configured in the etcd.yaml file in the control plane, both highlighted in yellow.

A new file containing an encryption configuration was created on the control plane called “enc.yaml” and can also be found in /External/encryption-config.yaml in the repository. The configuration uses the “secretbox” algorithm for encryption.

Kubernetes, by default, has numerous admission control plugins enabled for the cluster in the API server manifest.

### 2.3.9 Sprint 9

This sprint focuses on adding a network policy to prevent Pods from freely communicating with each other, as done by default. A Pod network security policy called “default-policy.yaml” was configured, which denies communication between Pods.

### 2.3.10 Sprint 10

Sprint 10 focused on Pod security admission (or Pod access control) implemented by adding labels to Pods to enforce a specific Kubernetes standard, seen in Table 2.1.

|  |  |
| --- | --- |
| **Pod** | **Standards** |
| MongoDB | Enforce – baseline  Audit – baseline  Warn - privileged |
| Web application | Enforce – baseline  Audit – privileged  Warn - privileged |

**Table 2.1** Pod security admission standards in the cluster.

### 2.3.11 Sprint 11

Sprint 11 first focused on securing communication between Pods to satisfy the TLS communication requirement and then added a backup and restore method for etcd.

Secure communication between Pods is handled by a third-party application called Linkerd and backups and restorations are done manually using etcdctl.

### 2.3.12 Sprint 12

Sprint 12 covered the last security additions left in the checklist: Role-Based Access Control, simulation of attacks and a cluster vulnerability scan.

RBAC was added to enforce authorisation and access control on a system by specifying user permissions. The permissions for user actions can be defined within a cluster or a Namespace using RBAC roles. The configurations for the role can be found in the /Roles path in the GitHub repository.

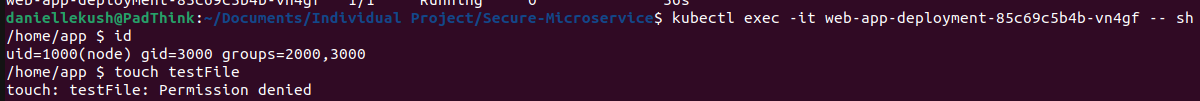
Manual tests were also done to simulate different types of attacks on the cluster. The results can be found in Table 3.1.

With all the security features from the checklist implemented, the cluster was scanned again to check for vulnerabilities.

### 2.3.13 Sprint 13

The final sprint of the project aims to improve the results of the vulnerability scanners. The feedback from the Sprint 12 scans provided a lot of possible improvements. The final results of the vulnerability scanner can be found in the paths /External/post-kube-bench-log.txt and post-kube-hunter-log.txt in the GitHub repository.

# Chapter 3 Results



**Figure 3.1** Test for creating a file on a Pod with a security context no longer allows to run as root.

## Text Description automatically generated

**Figure 3.2** Test for default network policy is configured to isolate Pods by preventing ingress and egress connections.

Text

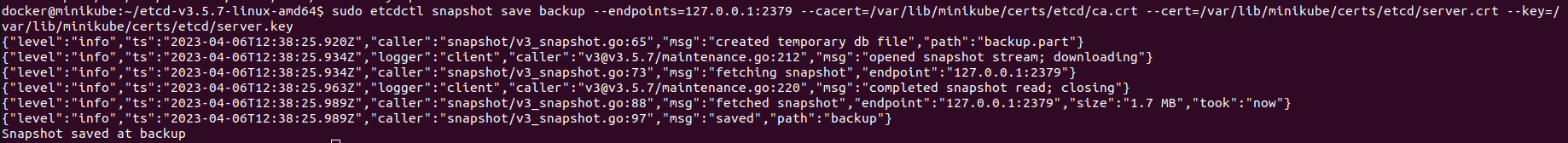
Description automatically generated

**Figure 3.3** Test to check if the Linkerd download has been successful.

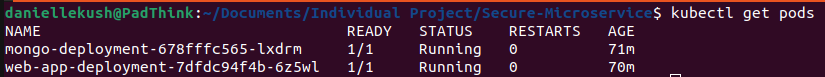
Text

Description automatically generated

**Figure 3.4** Results of running “linkerd check” show that the Linkerd control plane has been configured correctly.



**Figure 3.5** Testing saving a snapshot of etcd’s current state in the control node called “backup.db”.



**Figure 3.6** Results of restoring a backup after all Pods were deleted.

Text

Description automatically generated

**Figure 3.7** Testing the read-write role using a client dry run which successfully made the role and applied it to a user called Jane.

|  |  |  |
| --- | --- | --- |
| **Test name** | **Description** | **Result** |
| Public repository search for credentials | Public code repositories like GitHub may accidentally contain hardcoded credentials. Third-Party tools such as TruffleHog (TruffleHog, c2023) can be used or these can be manually searched. | At the time of the testing, the GitHub repository with the code is private, so no credentials are leaked (Pass). When it is opened for submission, it will technically be leaking the MongoDB Secret in Base64. |
| API server access | Kubernetes API endpoints should not allow anonymous access. The test runs a curl command against a known API endpoint (found by checking the IP and port used by the API server node in the “kube-system” Namespace). | Error message, stating the user is forbidden as they are anonymous. (Pass) |
| etcd server access | etcd stores confidential cluster information, so check to see if the endpoint allows anonymous access. | Error message about the legitimacy of the requester because they could not be verified. (Pass) |
| etcd Secret access | Check if an anonymous user can retrieve Secrets in etcd. | Error message giving a warning of the attempt. (Pass) |
| etcd with TLS | Check if etcd is correctly configured with a TLS certificate. | Permission denied message. (Pass) |
| Writeable hostPath Mount | An attacker may try to access underlying operating system by adding a writeable hostPath volume. The test will try to write one as an authorised user. | Permission denied message stating that only root can do that. (Pass) |

**Table 3.1** Results of manual testing with the description of the test cases (Min, 2021).

|  |  |
| --- | --- |
| **Security feature** | **Covered? (Yes/No)** |
| TLS for communication traffic | Yes (Sprint 8 & 11) |
| Limit resource usage on a cluster | Yes (Sprint 7) |
| Container privileges | Yes (Sprint 7) |
| Network policies | Yes (Sprint 9) |
| Secure etcd | Yes (Sprint 8) |
| Audit logging | Yes (Sprint 7) |
| Secrets instead of ConfigMap for private configuration data | Yes (Sprint 5) |
| Scan containers for known vulnerabilities | Yes (Sprint 6, 12 & 13) |
| Automate popular attacks | Yes (Sprint 6, 12 & 13) |
| Restoration of data using backups | Yes (Sprint 11) |
| Role-Based Access Control (RBAC) | Yes (Sprint 12) |
| Admission control | Yes (Sprint 10) |

**Table 3.2** Completed security feature checklist.

The requirements for The Data Protection Act were to provide authorisation (covered by RBAC), backup and restoration of data and "appropriate technical measures" (covered by the rest of the security measures). This objective has been fulfilled.

The requirements for HIPAA were to provide unique user identification (covered by RBAC), audit controls (covered by audit logging) and a person or entity authentication (covered by RBAC and admission control). The emergency access procedure requirement is fulfilled because permissions in the cluster can be changed very quickly using RBAC to allow emergency access, for example, placing the user in the root group so they become privileged. Other optional technical safeguards, such as encryption during transmission and to data, were added.

|  |  |  |  |
| --- | --- | --- | --- |
| **Vulnerability scanner** | **Passes** | **Fails** | **Warnings** |
| kube-hunter | 0 | 5 | - |
| kube-bench | 58 | 13 | 53 |

**Table 3.3** Initial results from both vulnerability scanners.

|  |  |  |  |
| --- | --- | --- | --- |
| **Vulnerability scanner** | **Passes** | **Fails** | **Warnings** |
| kube-hunter | 2 | 3 | - |
| kube-bench | 72 | 4 | 48 |

**Table 3.4** Final results from both vulnerability scanners.

Table 3.3 and Table 3.4 show that the pass rate is greater than in the initial scan, with fewer fails and warnings. Explanations for why failures or vulnerabilities occurred can be found in the text log files from the repository. The goal of having an improved result from the vulnerability scanners was achieved. Sixteen tests that did not pass now pass, reducing the number of tests that failed or gave warnings.

Timeline

Description automatically generated

**Figure 3.8** Overview of the project’s sprints as a Gantt chart with dates.

This project is complex because the completed application is much more developed than the one seen in Sprint 5. The initial application is vulnerable to many attacks, such as a non-encrypted etcd being vulnerable to a ransomware attack, the data can only be reliably restored in a backup or by paying a considerable sum and trusting the attacker to follow their word. The work in this project is also similar to work in other projects discussed in Chapter 1. Ianculescu et al. focused on gathering security requirements that fit US and European healthcare security regulations, the same type of work done in Sprint 2-4. Dissanayaka et al. investigated if Linux Singularity containers with MongoDB could follow HIPAA regulations, which is similar to checking if Kubernetes would follow healthcare regulations.

# Chapter 4 Discussion

## 4.1 Sprint 1 – Investigating the Problem

The main challenge discovered in this sprint was working with the image. Because healthcare systems have privacy concerns to maintain, finding a publicly available system that uses microservice-based applications was unsuccessful. Making a custom image to fit the project would aid the investigation, as a mock-up of a healthcare system would represent a more realistic use case and thus be more accurate. However, creating a custom image will take further effort because it is essentially programming an additional piece of working software that will also have to follow its own security measures. The work put into a custom image would take work out of investigating Kubernetes security. This challenge was solved by deciding to use a pre-existing image and working with it to make the cluster more secure. Then the project could focus on testing security rather than making a healthcare system.

## 4.2 Sprint 2 – Healthcare Security

A challenge discovered in this sprint was making a system that complies with HIPAA as it has many more requirements that may not all be satisfied in time. This challenge was mitigated by following a checklist of requirements and aiming to fulfil as many as possible successfully before the end of the project.

## 4.3 Sprint 3 – Kubernetes in Healthcare

A problem discovered in this sprint is finding publicly available examples of healthcare systems that use Kubernetes. Due to information disclosure issues, many systems do not disclose that they are using Kubernetes, so it was hard to find existing examples of systems. Even if they do, they do not always disclose how the system functions and what security features are used to prevent aiding security breaches.

## 4.4 Sprint 4 – Kubernetes Security

All the requirements could be fulfilled using Kubernetes or third-party applications, which can be additionally deployed in the cluster.

## 4.5 Sprint 5 – Application Deployment

The cluster's design followed a generic Kubernetes cluster with the addition of the microservice-based application components provided by the image. These diagrams can be seen in the Architecture section in Chapter 2.

## 4.6 Sprint 6 – Initial Container Scan

The CIS Kubernetes Benchmark is a security configuration guideline for Kubernetes created by a community consensus process (Center for Internet Security, c2023). kube-bench checks if clusters follow these guidelines. An advantage of using this scanner is that it can find security problems that have yet to be considered because they were not discovered in the research or are problems from the image itself.

The container uses this scanner by adding another Pod to run it as a Job and then complete it. Once completed, it provides a detailed log. Both the kube-bench Pod configuration file and the result of this log can be found in the “External” folder in the GitHub repository called “pre-kube-bench-log.txt”. kube-bench tests for many vulnerabilities and the cluster had 58 passes, 13 fails and 53 warnings.

Unlike the first application, the configuration file for the Pod running kube-hunter was taken directly from its GitHub page instead of being stored in the “External” folder (Samarasekara, 2020). The results of this application can be found in a detailed log called “pre-kube-hunter-log.txt” in the “External” folder.

Along with comparing the results of vulnerability scanners initially and at the end of the project, the vulnerability scanners also provide additional security tasks to complete as part of the “Scan containers for known vulnerabilities”. For example, one of the vulnerabilities that kube-hunter found in the container is that the container’s Kubernetes version is exposed, which could help an attacker because there might be a vulnerability associated with this version of Kubernetes. Therefore, ideally, a container should not expose information publicly if it is unnecessary.

## 4.7 Sprint 7 – Minor Enhancements

### 4.7.1 Limit Resource Usage

The first security addition is limiting the resource usage on a cluster so that Pods only use the required resources. The background research found this is important because it prevents requesting unreasonable values, such as a large amount of CPU, which could lead to attacks such as denial-of-service.

We need to know what reasonable resource usage is and assign that because if a service cannot access the resources it needs, it may get out-of-memory errors or reduce the system’s performance (Martίnez, 2022) (Daria, 2022).

The highest CPU request used 200m (200 millicpu/millicores), which is the API server. The highest memory request used 100Mi (100 mebibytes), which was etcd. These values make sense because of the functions of the API server and etcd in a cluster. Although the MongoDB or web application Pods made no requests, the values show the size of a reasonable request, which is the size that would be added. These values were rounded up, giving the limit for CPU requests as 300m and 170Mi for memory requests (since one Pod had a memory limit of 170Mi). (Densify, 2021) (Nguigi, 2020)

### 4.7.2 Container Privileges

Pods in a container must not run tasks with escalated privileges when they do not require it. By default, all Pods in a cluster are considered as privileged. Hence they have no limitations in what they may do. Leaving this in the cluster poses a security issue because if any Pod is compromised, an attacker does not need to bother gaining privileges since the Pod they captured already has it, making it easier to continue their attack. This type of security problem can be mitigated using Pod security context policies. Pod security context policies define a Pod's privilege and access control settings (Beiranvand, 2020) (Souppaya et al., 2017).

Originally, it was thought that none of the Pods in the cluster required high privileges to function. However, it was later discovered that the security context policies disrupted the MongoDB and prevented it from running. This problem did not apply to the web application Pod and was rectified later in the sprint after being discovered.

Further restrictions can be applied to Pods for even more security if required. Linux-specific permissions and different types of permissions are listed and described in a GitHub repository by torvalds, c2022.

### 4.7.3 Audit Logging

Audit logging allows a way to see the history of actions in a cluster. Audit logging provides three security advantages to a system:

If an attack occurs, investigating the action history can show culprits and what the attack affected for damage control.

It prevents repudiation because we know who the culprit is and precisely what was done which can help with prosecution.

It can be monitored to discover potential attacks, for example suspicious activity.

Even though the problem of not having a log was solved, the produced log could have been more user-friendly. It is hard to read and extract user data. There are solutions to this online, such as using a third-party tool to handle the logging and producing a much cleaner version. An example of an audit logging tool is ContainIQ (c2023).

## 4.8 Sprint 8 – Secure etcd

etcd stores the cluster’s data, so it is essential to protect it from being compromised. According to an article by Burillo (2018), “Access to etcd is equivalent to root permission in the cluster”. If it is compromised, an attacker can bypass the API server and destroy/update resources or gain access to the data in resources. By default, there is no restricted access to the etcd. Read access to etcd is also dangerous because an attacker can use it to elevate privileges (Janashia, 2022). This is an apparent security concern for any use case, but even more so for a healthcare system, as the data in the resources are highly confidential.

Conveniently, TLS for etcd communication already exists. It has been a Kubernetes default since 2020, meaning that etcd already has encryption in transit using TLS.

The next step was securing etcd at rest. Currently, etcd only stores one Secret: the MongoDB username and password. Since MongoDB is the database storing healthcare data, the credentials must be protected. By default, Secrets are stored in Base64 encoding, which is more secure than plaintext because a human cannot directly read it, but it is still simple to decode. According to Beiranvand (2020), storing data as Base64 is the same as storing it as plaintext to an attacker. However, this new policy only applies to newly created Secrets, not existing ones, so the MongoDB Secret was deleted and reapplied. The encryption algorithm is not the recommended default over plaintext (aescbc) because it is weak and vulnerable to padding oracle attacks. Instead, we use secretbox, a newer and more robust algorithm. The only problem this may face is that Kubernetes considers it as “may not be acceptable in environments that require high levels of review”. However, it is satisfactory for the requirements because neither The Data Protection Act nor HIPAA specifies the degree of encryption as sufficient protection against unauthorised parties. On the other hand, healthcare systems may be subject to high levels of review (Logan, 2020).

The issue discussed in Sprint 7 for MongoDB Pod’s security context was fixed in this sprint.

A weird problem was encountered in this sprint when making changes to the “kube-apiserver.yaml” file in the control plane (in which we must add the audit log and Secret encryption configuration files). When editing the file, it saves the changes and they are applied to the cluster. However, it changes back to default when the node is restarted. It only happens to the manifest file itself. The configuration files that were added remain. This is an obvious problem because the test environment cannot stay up for so long without restarting, but restarting means having to redo the configuration file references in the API server manifest, which makes developing and testing more obnoxious.

Kubernetes, by default, has numerous admission plugins enabled for clusters. The plugins in this cluster which are relevant to security are:

* NamespaceLifecycle – Prevents a Namespace that is terminating from having new objects created and ensures requests in a non-existent Namespace are rejected.
* LimitRanger – Applies a 0.1 CPU requirement to all Pods in the default namespace.
* NodeRestriction – Prevents kubelets from deleting their node API object and restricts some label modifications.
* ResourceQuota – Observes incoming requests and ensures it does not violate resource quota constraints.

## 4.9 Sprint 9 – Pod Network Security

Defining network policies is essential because, by default, all communication between Pods is allowed. If a container is compromised, allowing all network traffic may expose other containers in the cluster (Souppaya et al., 2017). We want to limit the amount of connection between Pods as much as possible to mitigate what a malicious user can do to the system in the case of an attack. A network policy adds an extra layer of security in a cluster as in the case of a security breach. If attackers reach the cluster network, the policy can stop them from sending traffic to applications inside Pods. Another issue that a network policy solves is if an attacker compromises a container and tries to explore the network to move to other containers. Restricting communication in a network will make this type of attack much harder to do successfully (Beiranvand, 2020).

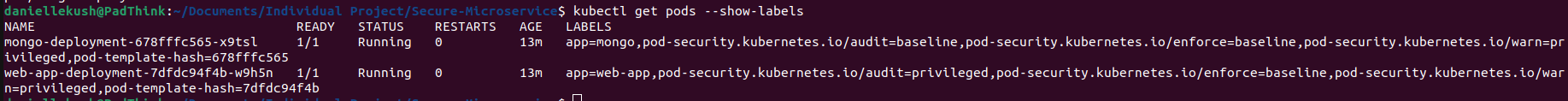
The idea in this sprint is to add a deny-all ingress (incoming communication) and all egress (outgoing communication) between Pods since the image being used does not require any communication between the Pods to function. This policy is configured to be assigned to all Pods in this Namespace by default, meaning that if new Pods are added and required to communicate with other Pods they will need their own network security policy. (De Tender, 2022) (Jain, 2022)

## 4.10 Sprint 10 – Pod Security Admission

Pod security admission is a way of defining security levels for Pods so that it restricts their behaviour. There are three Pod security standards: privileged (no restrictions), baseline (some restrictions) and restricted (very restricted). Along with these standards, three actions can be taken: warn, audit and enforce. For this project, the Pods have been restricted as much as possible without restricting their required behaviour. Audit actions are also used for further audit logging (Learn with GVR, 2022).

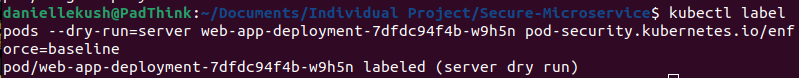


**Figure 4.1** Pods have been given the security admission requirements using labels.



**Figure 4.2** Full list of labels for each Pod following the standards defined in Table 2.1.

One of the problems in this sprint was assigning security standards without breaking the cluster. If a standard is too restrictive, the application will not function properly. This problem was mitigated by using the “—dry-run=server” option when assigning labels, which tests how the label will affect the Pod. It will either label successfully or display an error, providing feedback on whether this level is sufficient. The dry-run option discovered that the restricted mode is too restrictive for the Mongo and Web Pods, so they used baseline (Admin, c2023).



**Figure 4.3** Pod labelled using server dry run was successful.

Another problem discovered was Pods coming back whenever deleted, which would essentially undo the removal because the same Pod replaced it with a different name. This problem made testing features harder than it needed to be. The solution is to delete the deployment instead of Pod (nomad et al.) because the deployment keeps recreating the Pod when it is deleted. When this is done, the configuration can be deployed again and a new Pod will be created with a new configuration (if it was changed).

## 4.11 Sprint 11 – TLS Communication & Backups

### 4.11.1 TLS Communication

In sprint 9, the network policy defined closed communication between the two main Pods (web application and MongoDB) since they do not need to communicate to function. This means the only other place communication needs to be encrypted is when handling etcd, which was done in sprint 8. Although unnecessary, this sprint will still try to secure communication between Pods so that in the case of adding more Pods that need to communicate with another Pod, that communication is secured by TLS.

TLS is a protocol that provides security over a network. It protects exchanged data (encryption), provides authentication using certificates and ensures that the data received is the exact sent (integrity). Encryption provides confidentiality to the communication, authentication prevents an attacker from getting away with being at the other end of the connection and integrity prevents an attacker from modifying any part of the data sent during the communication. These are all qualities necessary in a healthcare system. By default TLS only lets the client authenticate the server and does not allow for the other way around. For better security, it would be ideal to have a symmetric check. Another version of TLS is mutual TLS (mTLS) which allows this. mTLS protects against unauthorised access to the network. (Linkerd Authors, c2023)

Although TLS sounds very good on paper, it also has some disadvantages. The specification for it is complex and lacks detail. Where it does contain detail, they do not always make sense. Because of this, most implementations of TLS do not fully follow the specification.

A third-party application called Linkerd adds this extra security to Pod communication. This can be added to the cluster and configured to allow TLS protocols and mTLS connections. mTLS is very complex, so using a third-party application saves time in implementing the protocol. Furthermore, Linkerd provides mTLS without requiring extra configurations (Morgan, c2023).

Since Linkerd adds many Pods into the cluster, it has been added into its own Namespace called ‘linkerd’, where the MongoDB and web application Pods remain in the default Namespace on the worker node. Separating Pods into Namespaces not only structures the cluster better but also provides an extra layer of security, as a vulnerability in one Namespace may not necessarily affect the other Namespaces.

To use Linkerd, pipe Linkerd into the Kubectl apply command when adding a new Pod.

### 4.11.2 Backups

Data backup and restoration have many advantages, but for securing healthcare systems, the main advantage lies in mitigating ransomware. Ransomware is a type of malware that encrypts data on a victim’s machine/system and only the attacker has the key for decryption. The attacker would try and extort the victim, asking for pay in trade for the data. If there is a backup of the data, the system can be cleaned of the malware and then restored. This type of attack is disruptive to the system and can lead to the loss of highly confidential data, so a healthcare system would be a frequent target (Berry, 2022). According to statistics from an article by Fox (2023), 2021 had 91 annual ransomware attacks and roughly one in five could restore their data using backups. 15.8% of the attacks had some or all the private data posted on the dark web for sale. Clearly, this type of attack is a big issue in the healthcare industry.

The method added to the cluster is by saving a current snapshot of etcd on the control node and restoring it when needed using the saved snapshot (Execute on Command, 2022).

Several problems were discovered during this sprint. The first of which is when Linkerd could not be found, the fix is to run “export PATH=$PATH:/home/daniellekush/.linkerd2/bin” in a terminal. Another problem with Linkerd is that sometimes it gives an error saying “no objects passed to apply” when trying to install it. The way to fix it is to add the “--set proxyInit.runAsRoot=true” flag to the command. If a third-party deployment deployed a Pod, there is no direct access to the manifests. Changing their configuration cannot be done locally, but it can be done using kubectl’s edit deployment command. This command opens the deployment’s YAML file.

Furthermore, a problem was found with the design. The cluster’s architecture initially involved only one node, and everything (including critical regions such as etcd) was in the same node as application Pods. This is a security vulnerability in the architecture because if an attacker entered the node through the application, they would also have access to the control node. Therefore, the cluster was changed to support using two nodes.

The biggest problem discovered in this sprint is the insecurity with the backups. The backups are stored as plaintext in etcd and the encryption configuration introduced in Sprint 8 does not cover the encryption of snapshots. Components of etcd are encrypted and the cluster has been secured to prevent all sorts of vulnerabilities in the control node, but if someone had managed to find the backup data for etcd, they would not need to bother with decrypting the main data. A way to mitigate this issue is by moving the exported database snapshot from the control node to a second location and encrypting the file there. There are also problems with manually backing up and restoring, meaning that snapshots could be missed. Companies use automatic backup and restore applications such as Kasten’s K10 (Kasten, c2023). According to research, this is the best free option available, but it refused to work on the cluster as its Pods kept failing when deployed. A good practice mentioned earlier is storing the backups in a secure secondary location. Many options were paid or provided free, with very limited storage.

## 4.12 Sprint 12 – Final Main Additions

### 4.12.1 Role-Based Access Control (RBAC)

RBAC is a good way of regulating user actions in a cluster, requiring users (and entities) to authenticate themselves. However, there are also some issues. Firstly, Kubernetes only offers the manual configuration of roles, so there is no way of automatically giving new users roles or updating them. With the potential scale of a healthcare system, this can be very complicated and inefficient, which could lead to misconfigurations resulting in vulnerabilities. Another problem is that Kubernetes does not offer a built-in method of easily identifying which level of access a user has within a cluster. This can cause issues if a user is granted a too-privileged role because it can only be easily discovered after the user exploits this. However, security features aim to defend a system before an attacker manages to exploit it. Finally, Kubernetes does not provide the functionality to help manage complex RBAC configurations. A third-party tool can be used to fix these issues. The problem is that a lot of these third-party tools require payment so they could not be applied to this project (Magnusson, c2023).

Since the testing system for this project is relatively small and straightforward, the manual RBAC approach will be used. Three different roles were made that allow different permissions.

|  |  |
| --- | --- |
| **Role** | **Description** |
| read-only | Permission to read all resources in the default Namespace. Applied to authenticated users. |
| read-write | Permission to read and write all resources in the default Namespace. Applied to the dev ServiceAccount. |
| admin | Permission to read and write to any resource across the whole cluster. Applied to the admins group. |

**Table 4.1** The three custom roles created for the cluster along with descriptions of what they are allowed to do.

The read-only role is treated as the most fundamental role in terms of permissions, whereas the admin role is the most privileged. The read-only role is applied only to authenticated users, so unauthenticated users cannot read the “default” Namespace. The read-write role is applied to “dev” (developer) accounts, allowing them to edit files when needed. The developer account should only be given to developers. The admin role is applied to any account in the “admins” group. Accounts should only be added to this group if they are an administrator. Unauthenticated users cannot access or edit parts of the cluster using these roles. The configurations for the web and MongoDB applications were changed so that they use the “dev” ServiceAccount by default (Magnusson, c2023) (Luu, 2022). All the roles are saved in separate manifests in the “Roles” folder in the repository.

Text

Description automatically generated

**Figure 4.4** The MongoDB Pod lists the developer ServiceAccount in its configuration. Checked using the command “kubectl edit pods/[mongo-deployment pod name] -o yaml.

### 4.12.2 Attack Simulation

Third-party tools exist for simulating attacks on clusters and containers. In this project, three examples were looked at. First is Gremlin (Sengupta, 2020), which simulates attacks on EKS clusters. Gremlin uses chaos testing to help find weaknesses in a system. Unfortunately, even though EKS clusters are common in the industry, the cluster used in this project is not EKS. The second attack simulator is called Simulator (Simulator, c2023). Simulator creates a cluster using AWS accounts to run system penetration tests and provides feedback on mitigating any discovered vulnerabilities. AWS accounts are also commonly used in the industry, but unfortunately, not by this project. The final tool is kube-hunter, which was used in Sprint 6. This tool is an automatic vulnerability scanner which provides a report stating which vulnerabilities were found in the cluster made by attacking the cluster.

### 4.12.3 Vulnerability Scanning

kube-bench was scanned again and produced different results from the initial scan. The test had to be done differently (Wilson, 2023) because the initial structure of the cluster only used one node for everything, but now there are two different nodes. The scan must be done on the master node, so kube-bench was configured to run there. In total, there were 61 passes, 11 fails and 52 warnings. This is a slight improvement over the initial scan.

Similarly, kube-hunter was scanned again but had no change, the same system vulnerabilities remain.

Although there was only a minor improvement from adding the requirements from the security checklist, the scanners provided a lot of detail on what the problem was and what went wrong. This meant that further sprint cycles could be done to fix these problems and improve the security of the containers.

## 4.13 Sprint 13 – Vulnerability Fixes

Although the project this far focused on implementing security features that follow the requirements of HIPAA and The Data Protection Act, it is also vital to ensure that the cluster has as few vulnerabilities as possible. The advantage of using kube-bench and kube-hunter for discovering vulnerabilities in the system is that they provide detail on how to mitigate issues.

Two vulnerabilities that were discovered by kube-hunter were fixed:

1. Vulnerability CVE-2020-10749 – CAP\_NET\_RAW is enabled by default for Pods. If an attacker managed to compromise a Pod, they may be able to attack other Pods on the same node. This is an ARP poisoning and IP spoofing vulnerability (StackRox, 2020). It was mitigated by disabling CAP\_NET\_RAW and NET\_RAW in the security contexts of Pods (KUBESEC.IO, c2023). For automatic filtering, instead of adding it to every security context of new Pods, disable this in the Pod admission control.
2. KHV002 information disclosure – Exposing information about the infrastructure of the cluster. Publicly saying that the cluster is using Kubernetes and the specific version could be used by an attacker to target known vulnerabilities of this version of Kubernetes. Mitigated by adding the “--enable-debugging-handlers=false” flag in the API server Pod specification file (kube-apiserver.yaml in /etc/kubernetes/manifests/ on the master node).

Eleven warnings and errors given by kube-bench in Sprint 12 were fixed. Four more warnings were identified as false negatives. The fixes are summarised below:

* 1.1.9 Ensure that the Container Network Interface file permissions are set to 600 – The Container Network Interface is important for overlay networking and should only be writable by administrators on the system. They were mitigated by changing the file permissions using the “sudo chmod 600 /etc/cni/net.d” command on the master node (Tenable, c2023).
* 1.1.10 Ensure that the Container Network Interface file ownership is set to root:root – For the same reason as 1.1.9, these files should be owned by root:root (user is root and group is root). This was mitigated by changing file owner using “sudo chown root:root /etc/cni/net.d” on the master node.
* 1.1.19 Ensure that the Kubernetes PKI directory and file ownership are set to root:root – The PKI directory stores certificates used in many operations, including TLS communication. This was mitigated by changing the file owner using “sudo chown -R root:root /var/lib/minikube/certs/” on the master node.
* 1.1.20 and 1.1.21 are both false negatives. It checks for the existence of PKI certificate files and fails when it does not find them, but they exist just in a different directory (in /var/lib/minikube/certs)
* 1.2.17 Ensure that the --profiling argument is set to false – The profiling feature is used to identify specific performance bottlenecks and generates program data that could be exploited for information disclosure. This extra attack surface is unnecessary, so it should be set to false in the API server Pod specification (Tenable, c2023).
* 1.2.19 Ensure that the --audit-log-maxage argument is set to 30 – The age of audit logs should be configured to state how long to retain the logs. This is mitigated by adding the flag and setting it to 30 in the API server Pod specification (Tenable, c2023).
* 1.2.20 Ensure that the --audit-log-maxbackup argument is set to 10 – Specifies how many audit logs to keep and when a new log is created, an old one is overwritten. This was mitigated by adding the flag and setting it to 10 in the API server Pod specification (Tenable, c2023).
* 1.2.21 Ensure that the --audit-log-maxsize argument is set to 100 – Specifies how big (in MB) the audit log files should be. The bigger this is, the more log data there is (at the cost of constantly storing files this size) (Tenable, c2023).
* 1.2.22 Ensure that the --request-timeout argument is set as appropriate – This determines the global request timeout limit, which is important to set because if it is too large, the API server will be vulnerable to Denial-of-Service attacks. If set too low, some resources in the cluster may be inaccessible if the connection is slow or if the request exceeds what can be transmitted within the limited time. This was mitigated by adding the flag and setting it to 300 seconds in the API server Pod specification (Tenable, c2023).
* 1.2.29 and 1.2.30 are both false negatives. They check for the existence of an encryption configuration file and its provider. The encryption configuration file can be found in “/etc/kubernetes/enc/enc.yaml” and the provider is configured as “secretbox”. This has been mounted and flagged as the encryption configuration file in the API server Pod specification.
* 1.3.1 Ensure that the --terminated-pod-gc-threshold argument is set as appropriate – The threshold value determines when to activate garbage collection, which is important to ensure enough resource is available. Depending on the system, the default value is 12,500, which could be too high. This was mitigated by adding the flag and setting it to 10 in the API server Pod specification (Tenable, c2023).
* 1.3.2 Ensure that the --profiling argument is set to false – Same reason as 1.2.17 but for the profiling feature on the controller manager. It was mitigated by adding the flag and setting it to false in the “kube-controller-manager.yaml” file.
* 1.4.1 Ensure that the --profiling argument is set to false – Same reasons as 1.2.17 and 1.3.2 but for the scheduler. It was mitigated by adding the flag and setting it to false in the “kube-scheduler.yaml” file.

## 5.1 Conclusions

The project had three main goals, which it has been successful in doing. The first goal was to follow HIPAA guidelines, where it has successfully implemented all the required security and suggested measures. The second goal was to follow The Data Protection Act, which it has also managed to do successfully by implementing the required security measures. The third goal was to improve the security of the microservice-based application, which can be seen in the comparison of scanning results from the beginning of the project and the end. The pass rate increased by 22% and the fail rate reduced by 50%.

However, there were three vulnerabilities, four fails and 48 warnings. Although it achieved the goals, the system could have been secured further.

It was also found that there were ways of making handling the security of a cluster easier. These were paid tools, so they were not used, but they do have the potential to fit healthcare systems better.

## 5.2 Ideas for future work

Due to time limitations, the project did not fix every vulnerability and add all of the CIS Benchmark standards. Future work would be to improve the cluster’s score on the vulnerability scanners.

One of the common attacks in healthcare discovered in the Background Research chapter is malware in the software. A way to solve that is to use a runtime anti-malware deployed in the cluster, such as one proposed by Aqua Security. With the importance of healthcare, reacting to a common attack as fast as possible would be an advantage, so experimenting with similar technologies and what types of malware attacks they can defend themselves from would be interesting (Aqua Security, 2022).

The deployment did not address image risks because it used the assumption that the simple image example was secure. In reality, this is unlikely to be the case. Future work would include testing how to mitigate image vulnerabilities.

# List of References

Data Protection Act 2018. (c.40). [Online]. London: HMSO. [15 February 2023]. Available from: https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted

Corporate Information Governance. 2019. Data Protection Policy. [Online]. Version 5.1. Leeds: NHS. [13 February 2023]. Available from: <https://www.england.nhs.uk/wp-content/uploads/2019/10/data-protection-policy-v5.1.pdf>

US House. Committee on Ways and Means. Health Insurance Portability and Accountability Act. 1996. Hearing, 21 August. Washington: Government Printing Office

Maayan, G.D.M. 2023. Kubernetes in the Health Industry: What You Should Know. HealthTechZone.

Steger, A.S. 2020. How Kubernetes Paves the Way for Secure Innovation in Healthcare. HealthTech.

Apricorn. c2023. Data Encryption in Healthcare. Apricorn.

Department of Health & Human Services USA, 2007. HIPAA Security Series. CENTERS for MEDICARE & MEDICAID SERVICES

Miller, K.M. 2021. How Kubernetes is Changing the Face of Medical Technology. 24 November. Fairwinds. [Online]. [8 March 2023]. Available from: <https://www.fairwinds.com/blog/how-kubernetes-is-changing-the-face-of-medical-technology>

Lawton, G.L. c2023. 6 New Requirements for Securing Microservices vs. Monolithic Apps. TRACEABLE. [Online]. [9 February 2023]. Available from: <https://www.traceable.ai/blog-post/6-new-requirements-for-securing-microservices-versus-monolithic-applications>

NHS Digital. c2023. Data Security and Protection Toolkit. [Online]. Available from: <https://www.dsptoolkit.nhs.uk/>

CHG Healthcare. c2023. CHG Healthcare. [Online]. Available from: <https://chghealthcare.com/>

Golden, B.G. c2023. 4 ways to exploit microservices architecture for better app sec. TechBeacon. [Online]. [12 February 2023]. Available from: <https://techbeacon.com/app-dev-testing/4-ways-exploit-microservices-architecture-better-app-sec>

Bush, T.B. 2020. Which Is More Secure: Monolith or Microservices?. 10 September. Nordic APIs. [Online]. [9 February 2023]. Available from: <https://nordicapis.com/which-is-more-secure-monolith-or-microservices/>

Gain, B.C.G. 2018. Microservices Security: Probably Not What You Think It Is. THENEWSTACK.

Brodie, B.B. 2016. Increase Security by Transitioning From Monolith to Microservices Architecture. 8 July. DZone. [Online]. [12 February 2023]. Available from: <https://dzone.com/articles/increase-security-by-transitioning-from-monolith-t>

TechWorld with Nana. 2022. Kubernetes Security Best Practices you need to know | THE Guide for securing your K8s cluster!. [Online]. [12 February 2023]. Available from: <https://www.youtube.com/watch?v=oBf5lrmquYI>

TechWorld with Nana. 2022. Kubernetes Crash Course for Absolute Beginners [NEW]. [Online]. [27 January 2023]. Available from: <https://www.youtube.com/watch?v=s_o8dwzRlu4>

Arampatzis, A.A. 2021. Encryption and etcd: The key to securing Kubernetes. 20 April. INFOSEC. [Online]. [13 March 2023]. Available from: <https://resources.infosecinstitute.com/topic/encryption-and-etcd-the-key-to-securing-kubernetes/>

Samarasekara, H.S. 2020. KUBE-HUNTER installing and testing. 18 June. Medium. [Online]. [1 March 2023]. Available from: <https://medium.com/@heshani.samarasekara/kube-hunter-installing-and-testing-3ea6cfc09738>

Martίnez, J.M. 2022. Understanding Kubernetes Limits and Requests. 18 November. sysdig. [Online]. [26 March 2023]. Available from: <https://sysdig.com/blog/kubernetes-limits-requests/>

Daria, E.D. 2022. Kubernetes Requests and Limits Demystified. THENEWSTACK.

Densify, 2021. Guide to Kubernetes Tools: Managing Kubernetes Resource Limits. Densify.

Nguigi, J.N. 2020. Understanding Resource Requirements — Kubernetes. 10 March. Medium. [Online]. [6 March 2023]. Available from: <https://ngugijoan.medium.com/understanding-resource-requirements-kubernetes-9bd34245f779>

Badawy, M.B. 2023. Harden Kubernetes cluster with pod and container security contexts. 9 January. EARTHLY. [Online]. Available from: <https://earthly.dev/blog/k8s-cluster-security/>

Singh, A.K.S. 2021. Getting started with Kubernetes Pod Security Context and Policy. 9 March. Medium. [Online]. [7 March 2023]. Available from: <https://medium.com/nerd-for-tech/getting-started-with-kubernetes-pod-security-context-and-policy-26619529a64b>

Morgan, A.G.M, Kjeldaas, A.K. c2022. torvalds. [Online]. Available from: <https://github.com/torvalds/linux/blob/master/include/uapi/linux/capability.h>

Olaogun, D.O. 2023. Kubernetes Audit Logs | Use Cases & Best Practices. 13 March. ContainIQ. [Online]. [1 March 2023]. Available from: <https://www.containiq.com/post/kubernetes-audit-logs>

ContainIQ. c2023. ContainIQ. [Online]. Available from: <https://www.containiq.com/>

Burillo, M.B. 2018. Securing Kubernetes components: kubelet, Kubernetes etcd and Docker registry – Kubernetes security guide (part 3). 24 April. sysdig. [Online]. [1 March 2023]. Available from: <https://sysdig.com/blog/kubernetes-limits-requests/>

Janashia, N.J. 2022. Kubernetes Security Best Practices: 10 Steps to Securing K8s. 3 May. Aqua. [Online]. Available from: <https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/>

Logan, M.L. 2020. The Basics of Keeping Kubernetes Clusters Secure. 14 October. TREND Micro. [Online]. [13 March 2023] Available from: <https://www.trendmicro.com/vinfo/my/security/news/virtualization-and-cloud/the-basics-of-keeping-your-kubernetes-cluster-secure-part-1>

De Tender, P.D. 2022. Kubernetes network policy best practices. 21 December. snyk. [Online]. [20 March 2023] Available from: <https://snyk.io/blog/kubernetes-network-policy-best-practices/>

Jain, P.J. 2020. Kubernetes Network Policy. 29 September. K21Academy. [Online]. [1 March 2023] Available from: <https://k21academy.com/docker-kubernetes/network-policies-in-kubernetes/>

Learn with GVR. 2022. Kubernetes v1.25 - Pod Security Admission Control - PodSecurityPolicy Successor. [Online]. [28 March 2023]. Available from: <https://www.youtube.com/watch?v=JYM7mSShfp0>

GoLinuxCloud. c2023. How to add label to running pod in Kubernetes. [no date]. GoLinuxCloud. [Online]. [28 March 2023] Available from: <https://www.golinuxcloud.com/kubernetes-add-label-to-running-pod/>

nomad, [Not Given], Yap, N.Y. 2021. Kubernetes pod gets recreated when deleted. [23 March]. GoLinuxCloud. [Online]. [29 March 2023] Available from: <https://stackoverflow.com/questions/40686151/kubernetes-pod-gets-recreated-when-deleted>

Beiranvand, H.B. 2020. Security Measures inside Kubernetes cluster. [30 March]. GoLinuxCloud. [Online]. [2 April 2023] Available from: <https://itnext.io/security-measures-inside-kubernetes-cluster-e43b3a328473>

Linkerd Authors. c2023. Automatic mTLS. Buoyant. [Online]. Available from: <https://linkerd.io/2.12/features/automatic-mtls/>

Morgan, W.M. c2023. A Kubernetes engineer’s guide to mTLS: Mutual authentication for fun and profit. [no date]. Buoyant. [Online]. [3 April 2023] Available from: <https://buoyant.io/mtls-guide>

Jackson, K.J. 2020. Kubernetes security best practices. bluematador. [Online]. Available from: <https://www.bluematador.com/blog/kubernetes-security-essentials>

Kapoor, S.K. 2017. How Watson Health Cloud Deploys Applications with Kubernetes. [14 July]. kubernetes. [Online]. [4 April 2023] Available from: <https://kubernetes.io/blog/2017/07/how-watson-health-cloud-deploys/>

Clark, L.C. 2022. IBM to fire Watson IoT Platform from its cloud. [15 November]. The Register. [Online]. [4 April 2023] Available from: <https://www.theregister.com/2022/11/15/ibm_set_to_retire_watson/>

Fox, A.F. 2023. Half of ransomware attacks have disrupted healthcare delivery, JAMA report finds. [10 January]. Healthcare IT News. [Online]. [4 April 2023] Available from: <https://www.healthcareitnews.com/news/half-ransomware-attacks-have-disrupted-healthcare-delivery-jama-report-finds>

Berry, M.D.B. 2022. Ransomware attacks against healthcare organizations nearly doubled in 2021, report says. [5 July]. Thomson Reuters. [Online]. [4 April 2023] Available from: <https://www.thomsonreuters.com/en-us/posts/investigation-fraud-and-risk/ransomware-attacks-against-healthcare/>

Aqua Security. 2022. Aqua Security demo of anti-malware detection and protection in running workloads. [Online]. [8 April 2023]. Available from: <https://www.youtube.com/watch?v=YIZSOB6yllA>

Aqua Security. 2022. Aqua Launches the Industry’s First Out-of-the-Box Runtime Security with Advanced Protection Against the Most Sophisticated Threats. [25 July]. Aqua Security. [Online]. [4 April 2023] Available from: <https://www.aquasec.com/news/cloud-native-runtime-security-advanced-protection/>

Kasten. c2023. Kasten. [Online]. Available from: <https://www.kasten.io/product/>

Execute on Command. 2022. Kubernetes Tutorial For Beginners | Kubernetes Backup and Restore Explained Step by Step. [Online]. [7 April 2023]. Available from: <https://www.youtube.com/watch?v=vtSUlcN4Kfg>

Souppaya, M.S, Morello, J.M, Scarfone, K.S. 2017. 800-190. NIST Special Publication: Application Container Security Guide. Place of publication: NIST.

Magnusson, A.M. c2023. Kubernetes RBAC Explained: Challenges, Approaches, and More. [13 February]. StrongDM. [Online]. [10 April 2023] Available from: <https://www.strongdm.com/blog/kubernetes-rbac-role-based-access-control>

Luu, K.L. 2022. How to Create and Configure Your Kubernetes Service Account. [14 September]. Release. [Online]. [10 April 2023] Available from: <https://release.com/blog/how-to-create-and-configure-your-kubernetes-service-account>

Sengupta, S.S. 2020. The Secret To Succesully Simulate An Attack With Gremlin To Test Kubernetes. [3 November]. HackerNoon. [Online]. [12 April 2023] Available from: <https://hackernoon.com/the-secret-to-succesully-simulate-an-attack-with-gremlin-to-test-kubernetes-i8113wa9>

Simulator. c2023. ControlPlane. [Online]. Available from: <https://github.com/controlplaneio/simulator>

TruffleHog. c2023. Truffle Security. [Online]. Available from: <https://github.com/trufflesecurity/TruffleHog>

Min, D.M. 2021. Kubernetes Attack Surface. [17 June]. OPTIV. [Online]. [12 April 2023] Available from: <https://www.optiv.com/insights/source-zero/blog/kubernetes-attack-surface>

Wilson, B.W. 2023. Kube-Bench: Kubernetes CIS Benchmarking Tool. [9 February]. Devopscube. [Online]. [12 April 2023] Available from: <https://devopscube.com/kube-bench-guide/>

StackRox. 2020. Mitigating CVE-2020-10749 in Kubernetes Environments. [5 June]. Red Hat. [Online]. [17 April 2023] Available from: <https://www.stackrox.io/blog/mitigating-kubernetes-cve-2020-10749>

KUBESEC.IO. c2023. Reducing kernel capabilities available to a container limits its attack surface. Control Plane. [Online]. Available from: <https://kubesec.io/basics/securitycontext-capabilities/>

Tenable. c2023. Audits. Tenable. [Online]. Available from: <https://www.tenable.com/audits>

Esposito, C.E, Castiglione, A.C, Tudorica, C.A.T, Pop, F.P. 2017. Security and privacy for cloud-based data management in the health network service chain: a microservice approach. IEEE Communications Magazine, vol. 55, no. 9, pp. 102-108.

Ianculescu, M.I, Alexandru, A.A. 2020. Microservices – A Catalyzer for Better Managing Healthcare Data Empowerment. National Institute for Research and Development in Informatics.

Dissanayaka, A.M.D, Shetty, R.R., Kothari, S.K., Mengel, S.M., Gittner, L.G., Vadapalli, R.V. 2017. A Review of MongoDB and Singularity Container Security in regards to HIPAA Regulations. Texas Tech University.

# Appendix A Self-Appraisal

## A.1 Critical self-evaluation

The project was a success as three goals were achieved. However, two goals could have been improved.

Firstly, there were further suggestions made by HIPAA that were not implemented in the project: the integrity standards and automatic logoff. A feature could have been added to authenticate health information to maintain integrity in the database and transit. Automatic logoff could have been looked at for either the image or in containers, as the closest thing to automatic logoff in the project is request timeouts after 300 seconds.

Secondly, although there was an improvement with the security scanners, vulnerabilities were still discovered, which meant that even though the cluster had implemented the required technical safeguards, it was still unsafe from certain types of attacks.

These two issues are problems that could have been worked on if the project had been extended.

The project timeline followed the plan as sufficient time was given to allow work per sprint.

Many other healthcare standards were not investigated since they apply to other countries and give more areas to investigate.

One issue that arose in the sprints where the solution was never discovered is the API server files resetting to default when the control node is restarted. This issue meant that any time control node security features needed to be tested, they had to be re-added, which was a waste of time.

## A.2 Personal reﬂection and lessons learned

The approach to the project was good because the background research helped compose the requirements. Then those requirements were fulfilled based on the possibilities of Kubernetes. Two standards were investigated, one much stricter than the other, to mitigate the risk of the project not complying with any regulations. So at least one could have been completed and the other discussed.

Separating work into sprints helped the workload. Each sprint was separated in half: one for implementing the feature and the other to draft that sprint into the report. Working in this way helped the report stay up to date, along with marking off the checklist as the project progressed. This project also taught me how useful the background research is because the implementation was entirely based on discoveries made in research. It provided a lot of ideas of how to solve the problem. Working on this project taught me project management skills that I can take with me into future projects.

There were some problems during the sprints with correctly configuring security features due to a lack of experience which meant that it took longer to add a particular security feature than needed. Although these problems could have been prevented, having now come across them, I can prevent them in the future.

A way to avoid the manifest reset issue mentioned in A.1 was to keep my computer open and run the cluster “indefinitely”. However, that led to the computer slowing down and not being a feasible solution in the long run. This hacky fix could work for a server, but any time the server is reset, the control node will need to be reconfigured again, which does not scale well.

A similar work by Esposito et al. explored different attack surfaces in a medical system and how to protect them. It was an interesting angle that my investigation could have looked at too because it delves more into security techniques, such as using STRIDE to identify vulnerabilities in a system. I think the project could have been done better if I also looked into security frameworks such as MITRE ATT&CK as my approach for securing the system.

## A.3 Legal, social, ethical and professional issues

### A.3.1 Legal Issues

The project has no legal issues because all third-party applications are publicly available and the system does not interact with other users or data. The data used by the database comes from the publicly available application or contains test data during development, not of which are real user data. The image used as the application is a publicly available example which is free to use. Hence this project had no legal issues.

There are many legal issues for the subject of this project. Hacking, in the sense of unauthorised access to a system, is illegal in many countries, which is true for the US and UK. It is possible to legally hack a system, which is when a commissioned person tests the security systems to find vulnerabilities, such as by running penetration tests. They may obtain private data in their attack, which would breach privacy laws, such as committing fraud using someone’s private data.

Another legal issue is systems following security requirements. Healthcare systems must follow the law on how to secure their service. For example, US healthcare systems must follow the HIPAA guidelines. If they do not, they will be breaking the law.

### A.3.2 Social Issues

The project has no social issues because it does not interact with other users or data. There are, however, social issues for applications in healthcare.

Healthcare uses sensitive and identifiable information that could be used for identity theft. Attackers can use people’s data to carry out damaging activities which could ruin the person’s reputation.

Leaking private information may also enact social consequences on a person. For example, people in the victim’s social circle find out the victim has some disease which would lead to people treating them differently. Other sensitive data could cause social issues such as disabilities, insurance and others.

### A.3.3 Ethical Issues

### The project has no ethical issues because it does not interact with other users or data.

On the other hand, security in healthcare has ethical issues to consider, for example, the privacy of patient data. Anyone who does not require access to the data should not be able to know private information about another person. Letting third parties or other staff see sensitive information about a person if they can be identified is unethical and invades their privacy.

Another issue is with data collection. The system should only store data about a person if required. It should only store the minimal data requirement and not collect further information.

### A.3.4 Professional Issues

There were no professional issues since no external parties were involved in the project and it was worked on individually.

# Appendix B External Materials

Software:

* minikube – local Kubernetes
* kubectl – command line tool for communicating with a Kubernetes cluster
* etcdctl – command line tool for interacting with etcd
* k8s-in-hour – image of an example of a common web application architecture

Tools:

* kube-bench – tool for checking whether Kubernetes is secured by CIS Kubernetes Benchmark standards
* kube-hunter – tool for hunting security weaknesses in Kubernetes clusters
* Linkerd – secures communication between Pods on a cluster

# Appendix C Security Features Materials

|  |  |
| --- | --- |
| **Security feature** | **Covered? (Yes/No)** |
| TLS for communication traffic |  |
| Limit resource usage on a cluster |  |
| Container privileges |  |
| Network policies |  |
| Pods access control |  |
| Audit logging |  |
| Secrets instead of ConfigMap |  |
| Secure etcd |  |
| Scan containers for known vulnerabilities |  |
| Automate popular attacks |  |
| Automated restoration of data using backups |  |
| Role access-based control |  |
| Security admission |  |

**Table C.1** Project security features checklist from Sprint 4.

|  |  |
| --- | --- |
| **Security practice** | **Description** |
| TLS for communication traffic | By default communication between pods is unencrypted. An attacker can easily read the communication between the pods because they are in plaintext. Although TLS can have performance issues, it uses encryption and certificates which protect communication and can help identify insecure traffic. |
| Authorisation using Role-Based Access Control | Only allow certain users/groups to do certain actions. Keep as restrictive as possible so they only do what they absolutely need to. |
| Controlling access to the kubelet | The kubelet (the primary node) allows unauthenticated access by default, so this should be restricted. |
| Limit resource usage on a cluster | Relates to the idea of using policies to limit the number/capacity of resources and the size of resources to prevent users from requesting unreasonable values. |
| Container privileges | Avoid using root and running containers as root. If a container is compromised this provides the attacker with easier privilege escalation. Prevent containers from loading unwanted kernels (clear security issue because it is as if the attacker had access to a computer’s terminal/command prompt). |
| Network policies | Restricting network access using policies. By default all pods can communicate with every other pod in their Namespace. |
| Pods access control | By default all pods can talk to each other, so if an attacker gains access to one they can access any other pod. Remove any unnecessary communication between pods. |
| Restrict access to etcd | Should be limited to control plane only because etcd stores the data of the cluster. An attacker can bypass the API server and destroy/update resource if they gain access to etcd. They can also gain access to the data in resources. |
| Audit logging | For analysing compromise. |
| Secrets | Use secrets for confidential information since data is encoded instead of stored as plaintext. This will need to be encrypted with a third party tool. |
| Network access to control plane | Restrict which addresses can access the cluster instead of allowing it to be public. |
| Network access to nodes | Nodes should only accept connections from control plane on specific ports, NodePort and LoadBalancer. |
| Encrypt etcd | Encrypt all storage at rest. |
| Scan containers for known vulnerabilities |  |
| Sign container images | To create trusted content in containers. Images from untrusted sources may have malware or backdoors or they may be using vulnerable tools/libraries. It is important for images to be secure because an attacker can break out of the container the image is in and gain access to the host and from there access every other container that run on this host. From here they can access any data stored and see kubelet configuration files which they can use for further attacks. |
| Limit port ranges | Less possible entry points in the application. |
| Static code analysis | Discover common vulnerabilities in code. |
| Automate popular attacks | To see if popular attacks penetrate the system. |
| Automated restoration of data using backups | Backup data in the case of losing and leaking of data or even ransomware attacks. Should be a system that automates regular backups and restoration of data. Stores the backup safely (so you cannot lose the backup too). |
| Configuration security policies | Define security policies that enforce specific configurations so you can enforce a desired behaviour all the time rather than relying on the developers to understand how to configure an application securely. |

**Table C.2** Summary of key security features identified from multiple sources as part of the background research to investigate industry recommendations.