Instituto Superior de Engenharia do Porto

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Systematization of Knowledge on Hardening Kubernetes through Native Configurations

LABCIB

Master in Informatics Engineering - 2024/2025

Porto, December 8, 2024

1190830 - Luís Correia
1190974 - Pedro Lemos
1191526 - Pedro Moreira

Version 2, 2024-10-17

Revision History

Revision	Date	Author(s)	Description
1	2024-11-15	Pedro Moreira	Initial version and structure
2	2024-10-17	...	Extended description

Contents

List of Figures **vi**

1 Introduction **1**

2 Objectives and Scope **3**

3 Methodology **5**

4 Background **7**

4.1 Kubernetes Components 7

4.2 What is etcd? 8

4.3 What are Kubernetes Secrets? 8

4.3.1 Key Features of Secrets 9

5 Kubernetes Native Configurations for Hardening **11**

5.1 Data Encryption 11

5.2 API Server Configurations 13

5.2.1 Authentication 14

5.2.2 Admission Controllers 16

5.2.3 API Server Security Flags 19

5.3 Network Policies 19

5.4 RBAC and Authentication 19

6 Complementary Tooling **21**

6.1 Network and Security 21

6.2 RBAC 21

6.3 Monitoring and Logging 21

7 Analysis and Best Practices **23**

8 Conclusions **25**

Bibliography **27**

List of Figures

5.1 Access Control Overview [10] 16
5.2 Enabled admission controllers in testing instance 17

1 Introduction

[Description, considering the project, of what under individual responsibility]

2 Objectives and Scope

[Description, for what under individual responsibility, that inform the goal, the questions related to functional correctness, metrics used and their values, with partial response to questions, and goal achievement analysis. Explicitly mention the tool report(s).]

3 Methodology

[Description, for what under individual responsibility, that inform the goal, the questions related to maintainability, metrics used and their values, with partial response to questions, and goal achievement analysis. Explicitly mention the tool report(s).]

4 Background

Kubernetes is a container orchestration platform that automates the deployment, scaling, and management of containerized applications. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes enables IT teams to deploy and manage applications at scale efficiently using containers.

4.1 Kubernetes Components

Kubernetes architecture is based on a distributed system that consists of two primary types of nodes: master nodes and worker nodes. Each type plays a specific role in ensuring the cluster operates effectively [1].

Master Nodes

Master nodes are the control center of a Kubernetes cluster. They manage the cluster's state, handle scheduling, and coordinate operations. The key components running on master nodes are:

- **API Server** (`kube-apiserver`): Exposes the Kubernetes API and serves as the entry point for all cluster interactions.

- **Controller Manager** (`kube-controller-manager`): Manages controllers that monitor the clusters state and make adjustments to achieve the desired state.

- **Scheduler** (`kube-scheduler`): Component that watches for newly Pods without assigned nodes, and selects one for them to run on. Some factors are taken into the decision, like hardware/software/policy constraints, data location, deadlines, among others.

- **etcd**: Stores all configuration data and the state of the cluster in a consistent and reliable way.

- **Cloud Controller Manager** (`cloud-controller-manager`): This component runs cloud-provider-specific controllers that manage resources outside the Kubernetes cluster. It is only active when the cluster is hosted on a cloud platform.

Worker Nodes

The main responsibilities of a worker node is to process data stored within the cluster and managing networking to ensure seamless communication both between applications within the cluster and with external systems. The control plane assigns tasks to the worker node to carry out these functions. The key components are:

- **Kubelet**: An agent that ensures containers are running and in healthy state according to specifications.
- **Kube Proxy**: Manages network routing and load balancing at the network level.
- **Container Runtime**: The software responsible for running containers.

4.2 What is etcd?

etcd is a distributed key-value database essential for storing the cluster's state and configuration. It ensures data consistency across all nodes using the Raft consensus algorithm, which keeps all nodes in sync even during failures, providing a reliable foundation for Kubernetes operations [2].

A leader is elected among the etcd nodes to manage all write operations, while follower nodes stay synchronized with the leader. If the leader fails, the Raft algorithm ensures a new one is elected quickly, maintaining high availability and preventing service disruption [2].

Additionally, etcd persistently stores the cluster's state, allowing Kubernetes to retrieve and update this information in real-time. It holds metadata and operational data for Kubernetes objects like pods, deployments, and services, enabling the Kubernetes API server to query etcd and consistently enforce the cluster's desired state [2].

4.3 What are Kubernetes Secrets?

Secrets are objects used to store sensitive data, such as passwords, API tokens, or SSH keys. Instead of embedding this information directly into code or configuration files, Secrets allow secure storage and controlled access to sensitive information [3].

4.3.1 Key Features of Secrets

- **Storage**: Secrets are base64 encoded before being stored in `etcd`, but not encrypted by default. Kubernetes can be configured to encrypt them at rest using tools like Key Management Service (KMS).
- **Usage**: Secrets can be mounted as volumes in containers or passed as environment variables, which allows secure handling of credentials.

5 Kubernetes Native Configurations for Hardening

Having introduced what Kubernetes is, we are now able to delve into the main purpose of this report, which is to systematize what built-in features and configurations of this technology we can use to improve a cluster's security posture and resistance to cyber-attacks. This chapter will be split into various sections, each concerning an area of relevance when it comes to Kubernetes Hardening. In each section, a context regarding the security improvements that are available will be presented, and consequently the measures through which one can implement these improvements will also be showcased.

5.1 Data Encryption

As we've seen in the context section of this report, Kubernetes stores Secrets and their data on the etcd backend database without any form of encryption [3] - despite encoding them with Base64, which does not constitute encryption as the plain-text contents are directly obtainable. If an attacker is able to access either the API server or the etcd database directly, he will be able to obtain the plain-text contents of the secrets that are managed by that cluster. One can easily picture the compromise that this can cause, as secrets usually contain business-critical credentials that are used by services running in Kubernetes. For instance, an application Pod might need to access a database, and thus read the connection string and database user credentials from a secret. If an attacker could obtain the content of this secret, he too would at least have the credentials to access the database.

Fortunately, Kubernetes provides a series of configurations that can be applied to encrypt Secrets - and other resources - at rest. This is done through the configuration of some resources on the cluster's API server. Notably, Kubernetes defines the `EncryptionConfiguration` Custom Defined Resource (CRD), which allows practitioners to customize what resources shall be encrypted and through what mechanism (also known as provider) [4].

The flow of actions is as follows: a user or application creates a Secret by sending an API call to the Kubernetes API; the API checks its `EncryptionConfiguration` file to evaluate what encryption provider to use and which key to use for that provider; if a provider is specified, as well as the key to use, the API server uses both to encrypt the provided data; then, this encrypted data is stored in etcd. For the decryption, the API retrieves the secret from etcd and uses the aforementioned `EncryptionConfiguration` to decrypt the contents.

In 5.1, we present a simple example of an `EncryptionConfiguration`. What this does is instruct the Kubernetes API that all Secrets must be encrypted using the `aescbc` provider, using the encryption key specified in `keys.secret` field.

Listing 5.1: EncryptionConfig example

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- secrets
providers:
- aescbc:
  keys:
    - name: key1
      secret: <BASE 64 ENCODED SECRET>
- identity: {}
```

There are several providers which we can use when encrypting secrets, which are specified in the Kubernetes documentation itself [1]. Some facts are worth noting:

- The `identity` provider does not encrypt data. It's the default used by Kubernetes, as we've covered before. Despite this, it has practical utilities, such as serving as one of the encryption providers for instance when migrating clear-text secrets to encrypted ones. So, despite not encrypting data, it would be improper to discard its existence;

- The `aescbc`, `aesgcm` and `secretbox` providers require the encryption key to be specified in the encryption configuration file. Using them protects a cluster in the event of a compromise, but if the hosts in which the Kubernetes API is running are themselves compromised, this does not provide additional security, as the attacker will be able to obtain the encryption key from the local filesystem [5]. Nonetheless, it's an improvement from the default configuration.

---

[1] This comparison can be seen in this documentation link

Besides the providers mentioned above, there exists the `kms` provider, which is currently in version 2 - and version 1 is deprecated. This provider offers a more complex - but more secure - encryption solution. It's purpose is to solve the issue of having the encryption key defined in a local file, as we've covered previously. To address this matter, the kms provider calls upon two new components [5]:

- An external Key Management Service (KMS);
- A Kubernetes KMS provider plugin, which is responsible for connecting with the external KMS for the encryption and decryption processes.

This mechanism adopts an "envelope encryption scheme using a two-key approach" [5], with those keys being: the Data Encryption Key (DEK), which is used to encrypt the resources referenced in the `EncryptionConfig`; and the Key Encryption Key (KEK), which is used to encrypt the DEK and is hosted remotely on the KMS. This way, having the encryption key on a separate system, the main downside of the previous solutions is addressed, since a filesystem compromise of the Kubernetes master nodes wouldn't directly lead to a compromise of the secrets stored in the cluster.

However, we must note that the usage of the `kms` provider requires the usage of third-party tooling, not native to Kubernetes - since it requires the usage of an external KMS.

5.2 API Server Configurations

The API server is an essential component of Kubernetes, as we've explained in 4.1. It is responsible for handling all administrative operations and communications within the cluster, and thus acts as the primary interface for practitioners to interact with the Kubernetes environment. Given its critical role, the API Server is a frequent target for attackers aiming to exploit vulnerabilities, gain unauthorized access, or disrupt cluster operations.

Securing the API server is, thus, a crucial task that cluster administrators must undergo to ensure the integrity, confidentiality and availability of this environment. Misconfigurations or weak security settings can expose the cluster to numerous risks, such as unauthorized access to data, or disruption of operations.

Therefore, to assist practitioners in undergoing this process, we will look into relevant API server configurations which will improve its security posture.

Before proceeding, it's important to note that the API server can be interacted with through many means, such as the `kubectl` utility, as well as REST requests. This highlights the relevance of appropriate firewall practices. Access to the Kubernetes API must be properly limited to business-related needs. A good approach for this is to allow only an organizational VPN's IP or set of IP addresses to make requests to the API, thus limiting the attack vectors. However, were this firewall or VPN to be compromised, further security measures are necessary.

5.2.1 Authentication

When a user attempts to interact with the API server, he must first undergo a process of authentication. This verifies the identity of a user or service before allowing them to communicate with the API server. Kubernetes supports multiple authentication methods which can be employed [6].

X.509 Client Certificates

Kubernetes is able to use certificates for authentication purposes. For this effect, the API server must be started with the `-client-ca-file` flag which should point towards one or more valid Certificate Authority (CA) files, which will be used to validate the certificates used by clients when issuing requests to the API [6]. These certificates can be generated manually using tools such as `easyrsa` and `openssl`.

With this method "any user that presents a valid certificate signed by the cluster's certificate authority (CA) is considered authenticated" [6]. A more proper way to use this is to have users generate their own private key, as well as a Certificate Signing Request (CSR) which specifies necessary attributes that a user certificate must have. Notably, the subject of the certificate will be used by Kubernetes to identify the username of a client, and thus using it for authorization purposes. Then, this CSR is validated and signed by the administrator, using the CA of the cluster, and the certificate is distributed to the client.

This ensures that each user has an account, and further ensures traceability and accountability of any user's actions. However, the manual distribution of certificates might be a hindrance on organizations which are larger and more complex. Nonetheless, it's a native setting of Kubernetes that ensures the hardening of the authentication process.

Service Account Token

These are signed JSON Web Tokens (JWT) which Kubernetes generates so that entities can communicate with the API server [6]. These tokens are associated with ServiceAccount objects, which by default represent the identity of an application in Kubernetes

[7] - their function will be expanded upon in the RBAC section. It's unusual that these tokens are used for the authentication of users, as they are more frequently used by Pods which require to communicate with the Kubernetes API, or by processes that run outside of the cluster but frequently need to talk with its API.

OpenID Connect (OIDC)

This authentication mechanism relies on a third-party identity provider, such as Google or Keycloak, for the generation of a JWT which the user can then present to the API server in order to issue requests [6]. It's still relevant in the scope of this Systematization of Knowledge, as even though it resorts to an external mechanism, it uses a configuration that's available in Kubernetes itself.

It's worth noting that, unlike workloads - which are represented in Kubernetes via Service Accounts - human users are not represented by Kubernetes objects [6, 7]. Therefore, integrating with an external source-of-truth for users such as the ones mentioned above is optimal for handling user interaction with the API Server. The applicability of this can be easier when using a cloud instance, as practitioners can check what options their cloud provider offers [7], but technologies like Keycloak or Dex enable organizations to deploy an identity management system on their own infrastructure.

Webhook Token Authentication

Users are able to specify authentication tokens on their requests to the API, which must contain the value of this token on the Authorization header of the request. What this authentication method relates to is the assessment of the validity of the provided token.

It works by using a remote webhook service when determining user privileges [8]. This external service responds to REST requests and allows or denies access to the requested resources. It's essentially a way to delegate the authorization process to a third-party, thus allowing administrators to inject their custom logic as part of an authorization policy [9].

Anonymous Requests

Besides these login mechanisms, there is the concept of anonymous requests. When enabled, "requests that are not rejected by other configured authentication methods are treated as anonymous requests" [6]. As such, a user would be able to query the API without identifying himself which, if proper RBAC configurations are not employed, could be quite damming to an organization. As such, it is advised that anonymous requests are disabled by using the `-anonymous-auth=false` on the API server. If it needs to be enabled due to organizational needs, it's paramount to ensure that the permissions of unauthenticated users are as restrictive as possible.

5.2.2 Admission Controllers

When considering interactions with the Kubernetes API, there are three actions that take place: authentication, authorization, and admission. This is clearly illustrated in Figure 5.2.2. It's worth highlighting that both human users, as well as Pods, go through the same access control flow, further showcasing the versatility of Kubernetes authentication and authorization methods.

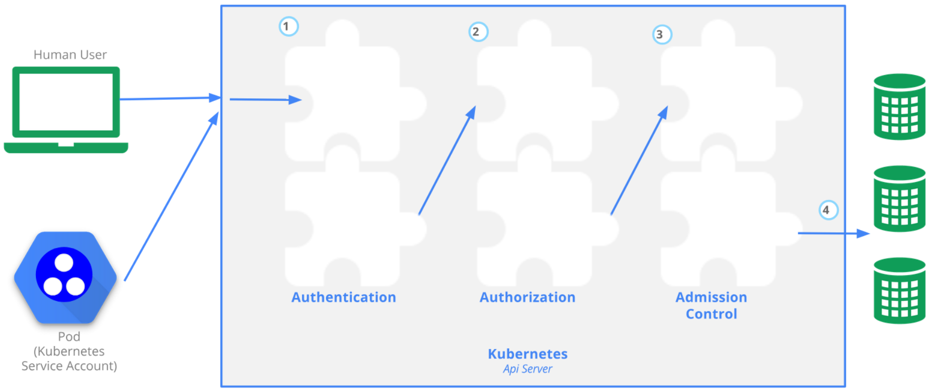


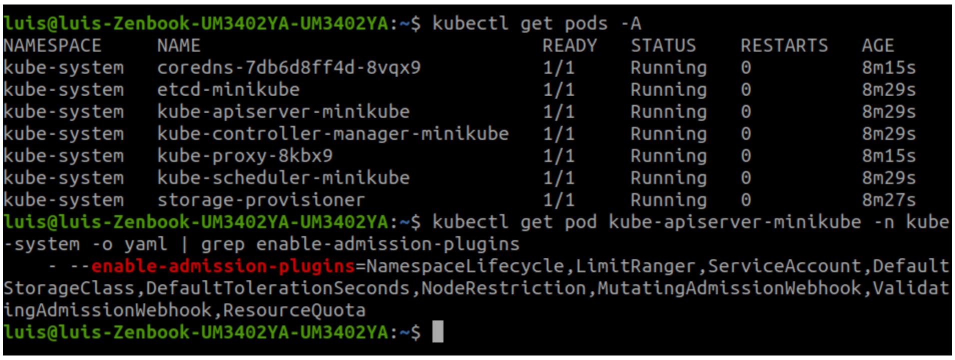
Figure 5.1: Access Control Overview [10]

The first one has been explored in 5.2.1, covering the different methods supported by Kubernetes, as well as recommendations. Considering authorization, this will be covered in 5.4. So, we must now cover the admission phase, as it's the last in the process, as well as a crucial part of it.

In Kubernetes, the admission process is handled by Admission Controllers. These are "software modules that can modify or reject requests" [10]. They do so by intercepting requests to the API before the persistence of resources [11]. Note that the persistence of resources is being specifically mentioned, excluding other types of operations. This is because admission controllers only operate on "requests that create, delete, or modify objects" [11]. As such, read operations (the `get`, `watch` or `list` verbs) must be strictly covered by RBAC policies.

Admission controllers may be validating, mutating, or both. With validating admission controllers, the resources are validated against the set of modules to ensure that they comply with the rules imposed by them, and are only persisted in the cluster if they pass all controllers. In the case of mutating controllers, however, they may modify the data for the resource that's being modified [11].

These modules are enabled at the API level, as we've seen. This is done by using the `-enable-admission-plugins` flag with the controllers that we want to enable. As such, to see what admission controllers are currently enabled in the cluster, we can investigate the manifest of the Pod that runs the API server in the cluster. An example of this is showcased in Figure 5.2.2.



```
luis@luis-Zenbook-UM3402YA-UM3402YA:~$ kubectl get pods -A
NAMESPACE     NAME                               READY   STATUS    RESTARTS   AGE
kube-system   coredns-7db6d8ff4d-8vqx9           1/1     Running   0          8m15s
kube-system   etcd-minikube                      1/1     Running   0          8m29s
kube-system   kube-apiserver-minikube            1/1     Running   0          8m29s
kube-system   kube-controller-manager-minikube   1/1     Running   0          8m29s
kube-system   kube-proxy-8kbx9                   1/1     Running   0          8m15s
kube-system   kube-scheduler-minikube            1/1     Running   0          8m29s
kube-system   storage-provisioner                1/1     Running   0          8m27s
luis@luis-Zenbook-UM3402YA-UM3402YA:~$ kubectl get pod kube-apiserver-minikube -n kube
-system -o yaml | grep enable-admission-plugins
    - --enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,Default
StorageClass,DefaultTolerationSeconds,NodeRestriction,MutatingAdmissionWebhook,Validat
ingAdmissionWebhook,ResourceQuota
luis@luis-Zenbook-UM3402YA-UM3402YA:~$ 
```

Figure 5.2: Enabled admission controllers in testing instance

By consulting the Kubernetes documentation in [11], we're able to confirm that the default admission controllers in version 1.31 are the ones shown in 5.2.

Listing 5.2: Default Admission Controllers

```
        CertificateApproval, CertificateSigning,
            CertificateSubjectRestriction, DefaultIngressClass,
            DefaultStorageClass, DefaultTolerationSeconds, LimitRanger,
            MutatingAdmissionWebhook, NamespaceLifecycle,
            PersistentVolumeClaimResize, PodSecurity, Priority,
            ResourceQuota, RuntimeClass, ServiceAccount,
            StorageObjectInUseProtection, TaintNodesByCondition,
            ValidatingAdmissionPolicy, ValidatingAdmissionWebhook
```

Thus, we're able to confirm that our testing instance is missing some of the defaults, which should be enabled. To enable them, we can simply edit the `kube-apiserver` manifest, adding the necessary controllers to the flag shown in Figure 5.2.2.

The available Admission Controller plugins vary vastly in functionality, and cover the validation and mutation of requests "mostly from the resource management and security points of view" [12]. Our recommendation is for practitioners to:

1. Validate what Admission Controllers are currently enabled in the cluster;
2. Investigate what plugins can be useful in their specific organizational context. This can be done by consulting the documentation of each controller's function [2], and evaluating if it responds to any organizational needs.

Besides these recommendations, we must also consider the ResourceQuota and LimitRanges admission controllers separately, as their implementation is quite helpful in ensuring the proper deployment of applications in a Kubernetes cluster.

ResourceQuotas and LimitRanges

Kubernetes offers the ResourceQuota and LimitRanges admission controllers by default, which can be used to restrict resource usage for namespaces, nodes or Pods [13].

Resource quotas can be used to limit the aggregate resource consumption per namespace [14]. With them, practitioners can manage their computational resources used by their clusters' namespaces more effectively. For instance, they can be used to limit the total amount of CPU and memory requested and/or used by each namespace.

It's worth noting the difference between the requested and the used values, as an administrator must be aware of this to properly define the resource quotas for the cluster. Considering the CPU and memory attributes, the requested value represents the resources the Pod specifies at the time of its initialization. This does not mean that the Pod will automatically use the requested amount of CPU and memory, but that it will require them during the execution. As an example, a Pod can request 1Gi of RAM, but only consume 100Mi while it's executing. However, if the memory usage has any spike during the Pod's operation - for instance, due to a spike in traffic - it will be able to use more of it, since that initial request has been allocated to it.

Further analyzing this example scenario, if this memory spike were to increase further, it could hinder the operation of the cluster as a whole, since it's exhausting the resources. This is where the concept of limits is useful. If a limit of memory is defined in this case, the pod will be able to use more that it has requested - as we've explained prior - but not more than the defined limit.

Having this context established, we assert as a good practice to define resource quotas which impose maximum values not only on the compute power requested for a namespace, but also on the limits which said namespace will not be able to exceed. A practical example of a ResourceQuota manifest matching this description is presented in 5.3.

---

[2] Presented in this documentation link

Listing 5.3: ResourceQuota example

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-cpu-mem-resourcequota
spec:
  hard:
    requests.cpu: "1"
        requests.memory: 1Gi
        limits.cpu: "2"
        limits.memory: 2Gi
```

Whereas a ResourceQuota imposes a limit on the resources used by a namespace, the LimitRange policy imposes limits per Pod [13]. It's objective is to help adminstrators ensure "that a single object cannot monopolize all available resources within a namespace" [15]. As such, it provides constraints to limit resource allocations per Pod or even container within each Pod, as well as storage requests per each `PersistentVolumeClaim`.

TODO: ACABAR PARTE DE LIMITRANGE

5.2.3 API Server Security Flags

Throughout this chapter, we've already explored some API configurations that can be enabled to harden the security posture of a Kubernetes cluster. Besides those, there are several others which we find relevant to systematize in the present document. That is, therefore, the goal that this subsection aims to achieve.

5.3 Network Policies

5.4 RBAC and Authentication

6 Complementary Tooling

While Kubernetes provides robust built-in capabilities for container orchestration, there is a broad list of complementary tools that allow to extend these capabilities. These tools enhance areas such as advanced security features and comprehensive monitoring to strengthen cluster security. This section highlights several tools that offer security-enhancing features and explains how each contributes to improving cluster security.

6.1 Network and Security

Text, text, text...

6.2 RBAC

Text, text, text...

6.3 Monitoring and Logging

Text, text, text...

7 Analysis and Best Practices

[Description, for what under individual responsibility, that inform the goal, the questions related to architectural compliance, metrics used and their values, with partial response to questions, and goal achievement analysis. Explicitly mention the tool report(s).]

8 Conclusions

...

Bibliography

[1] The Linux Foundation. *Kubernetes Components*. Kubernetes Components. URL: `https://kubernetes.io/docs/concepts/overview/components/`.

[2] etcd. *FAQ*. etcd. Section: docs. URL: `https://etcd.io/docs/v3.6/faq/` (visited on 11/28/2024).

[3] The Linux Foundation. *Secrets*. en. Section: docs. 2024. URL: `https://kubernetes.io/docs/concepts/configuration/secret/` (visited on 11/17/2024).

[4] The Linux Foundation. *Encrypting Confidential Data at Rest*. en. Section: docs. 2024. URL: `https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/` (visited on 11/17/2024).

[5] Emmanouil Gkatziouras et al. *Kubernetes secrets handbook: design, implement, and maintain production-grade Kubernetes Secrets management solutions*. en. Place of publication not identified: Packt Publishing, 2024. ISBN: 978-1-80512-322-4 978-1-80512-715-4.

[6] The Linux Foundation. *Authenticating*. en. Section: docs. 2024. URL: `https://kubernetes.io/docs/reference/access-authn-authz/authentication/` (visited on 11/17/2024).

[7] Andrew Martin and Michael Hausenblas. *Hacking kubernetes: threat-driven analysis and defense*. eng. Sebastopol: O'Reilly Media, Incorporated, 2021. ISBN: 978-1-4920-8173-9 978-1-4920-8170-8.

[8] The Linux Foundation. *Webhook Mode*. en. Section: docs. 2024. URL: `https://kubernetes.io/docs/reference/access-authn-authz/webhook/` (visited on 12/08/2024).

[9] Anders Eknert. *The Kubernetes Authorization Webhook*. en-US. May 2022. URL: `https://www.styra.com/blog/kubernetes-authorization-webhook/` (visited on 12/08/2024).

[10] The Linux Foundation. *Controlling Access to the Kubernetes API.* en. Section: docs. 2023. URL: `https://kubernetes.io/docs/concepts/security/controlling-access/` (visited on 12/07/2024).

[11] The Linux Foundation. *Admission Control in Kubernetes.* en. Section: docs. 2024. URL: `https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/` (visited on 12/08/2024).

[12] Gigi Sayfan. *Mastering Kubernetes: level up your container orchestration skills with Kubernetes to build, run, secure, and observe largescale distributed apps, third edition.* en. 3rd ed. Place of publication not identified: Packt Publishing, 2020. ISBN: 978-1-83921-125-6 978-1-83921-308-3.

[13] National Security Agency and Cybersecurity and Infrastructure Security Agency. “Kubernetes Hardening Guide”. In: *Cybersecurity Technical Report* (2022). URL: `https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF`.

[14] The Linux Foundation. *Resource Quotas.* en. Section: docs. 2024. URL: `https://kubernetes.io/docs/concepts/policy/resource-quotas/` (visited on 12/08/2024).

[15] The Linux Foundation. *Limit Ranges.* en. Section: docs. 2024. URL: `https://kubernetes.io/docs/concepts/policy/limit-range/` (visited on 12/08/2024).