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, November 28, 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

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 [**the_linux_foundation_kubernetes_2024**].

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 [**etcd_faq_2024**].

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 [**etcd_faq_2024**].

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 [**etcd_faq_2024**].

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 [**the_linux_foundation_secrets_202**

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 [**the_linux_foundation_secrets_2024**] - 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) [**the_linux_foundation_encrypting_2024**].

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 [**gkatziouras_kubernetes_2024**]. 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 [**gkatziouras_kubernetes_2024**]:

- 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" [**gkatziouras_kubernetes_2024**], 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 [**the_linux_foundation_authenticating_2024**]:

- **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 [**the_linux_foundation_authenticating_2024**]. These certificates can be generated manually using tools such as `easyrsa` and `openssl`.
- **Service Account Tokens**: These are signed JSON Web Tokens (JWT) which Kubernetes generates so that users, as well as pods, can communicate with the API server [**the_linux_foundation_authenticating_2024**]. These tokens are associated with ServiceAccount objects, whose 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):**
- **Webhook Token Authentication:**

NOTAS:

- dizer que `--anonymous-auth=false` deve ser usado para nao permitir auth sem identidade

- dizer que static tokens nao expiram portanto nao devem ser extensivamente usados

5.2.2 Admission Controllers

https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ https://kubernetes.io/blog/2022/your-admission-controllers-and-webhooks/ https://kubernetes.io/docs/concepts/security/controlling-access/#admission-control

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

falar de adicionar NamespaceLifecycle. falar de configurar as ResourceQuotas em cada deployment talvez?

5.2.3 API Server Security Flags

(https://chatgpt.com/share/674613c5-f8c8-8013-80ee-268d0270a2e1)

List and explain important API Server flags for hardening the cluster:

a. Network Security

–secure-port: Ensure the API Server listens on a secure port (default: 6443) and disables non-secure ports (–insecure-port=0).

–bind-address: Limit the API Server to listen only on the internal network (e.g., 127.0.0.1 or cluster-internal IP).

–tls-cert-file and –tls-private-key-file: Configure TLS to encrypt communication.

b. Authentication and Authorization Flags

–anonymous-auth=false: Disable anonymous access to the API Server.

–enable-bootstrap-token-auth=true: Use temporary tokens for secure cluster bootstrapping.

c. Audit Logging

–audit-log-path and –audit-log-maxage: Enable and configure audit logs to track all API requests and detect malicious activity.

d. etcd Encryption

–encryption-provider-config: Ensure sensitive data like Secrets is encrypted before being stored in etcd.

5.3 Network Policies

5.4 RBAC and Authentication

6 Complementary Tooling

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

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

...