Systematization of Knowledge on Hardening Kubernetes through Native Configurations

LABCIB

Master in Informatics Engineering - 2024/2025

Porto, November 23, 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 **v**

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 8

4.4 Securing Secrets 8

5 Kubernetes Native Configurations for Hardening **11**

5.1 Data Encryption 11

5.2 API Server Configurations 13

5.3 Network Policies 13

5.4 RBAC and Authentication 13

6 Complementary Tooling **15**

7 Analysis and Best Practices **17**

8 Conclusions **19**

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 consists of two main types of nodes:

Master Nodes

Master nodes are responsible for managing and controlling the cluster. They run components that manage the cluster's state, such as creating and scaling pods. Master nodes include:

- **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 cluster's state and make adjustments to achieve the desired state.
- **Scheduler** (`kube-scheduler`): Assigns pods to available nodes in the cluster.
- **etcd**: Stores all configuration data and the state of the cluster in a consistent and reliable way.

Worker Nodes

Worker nodes are where the pods (the units of execution in Kubernetes) run. Each worker node includes:

- **Kubelet**: An agent that ensures containers are running 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 store used by Kubernetes to store all critical cluster data, such as configuration settings and the desired state. It ensures that all components (e.g., API Server and Controller Manager) have access to consistent, up-to-date data and can operate in a coordinated manner.

4.3 What are Kubernetes Secrets?

Secrets in Kubernetes 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.

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.

4.4 Securing Secrets

To protect sensitive information stored in Secrets, follow these best practices:

- **Encryption at Rest**: Enable encryption for Secrets using a key management service (KMS).
- **Access Control**: Use **Role-Based Access Control (RBAC)** to restrict access to Secrets to only authorized users and services.
- **Network Security**: Ensure communication between nodes, `etcd`, and the API Server is secured using TLS.

Explicar o que é o Kubernetes e como usa o etcd (isto é importante). Falar também brevemente sobre o que são Secrets (porque depois vamos referir como é que os protegemos melhor). Relativamente ao Kubernetes parece-me interessante falar do básico, ou seja o que são master e worker nodes, quais são os componentes (API server, kubelet, essas coisas que se encontram facilmente na documentação deles). Não precisa de ser muito extensivo mas convém dar algum contexto para depois sabermos o que estamos a dizer

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

```yaml
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

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

...