Patrick Deutschmann

Security Aspects of Container Orchestration

Bachelor's Thesis

Graz University of Technology

Institute of Applied Information Processing and Communications
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch

Supervisor: Stefan More

Graz, August 2019

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________ ______________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ______________________ ______________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Containerisation is increasingly gaining traction to run modern applications in distributed environments. To run containers on a large scale and with high availability, container orchestration systems are commonly employed. The most widely used container orchestration system today is Kubernetes, which is highly flexible, but also comes with significant complexity.

In this thesis, we analyse the security of Kubernetes architectures. To do so, we create a layer model to give a holistic view of all relevant aspects. We demonstrate how an example application can securely run in a Kubernetes cluster and which configurations are necessary to strengthen security by employing multiple redundant barriers.

Our research shows that most Kubernetes installers already come with reasonably secure default configurations. However, custom adaptations in consideration of the deployed applications and their requirements to the runtime environment are imperative for secure cluster setup.

Contents

List of Figures

1 Introduction

With the advent of microservices and distributed environments, containerisation has gained a lot of traction. The most common engine today being Docker[1], many modern web applications run in Docker containers, instead of running in manually managed virtual machines environments.

To run distributed applications and services at large scale and with high availability, there is widespread demand for container orchestration systems to take care of administrative tasks such as scaling on-demand, continuous deployment and healing. Tools include Docker Swarm[2] and Apache Mesos[3]. However, the most widely used implementation for container orchestration today is Kubernetes[4], which has its origins in Google's Borg project[5]. It is an open-source system to manage deployment and management of container applications, whereas the container runtime can be Docker, containerd[6], cri-o[7] or similar. The Kubernetes version considered in this thesis is the currently most up-to-date v1.15.

Kubernetes is a very flexible and powerful system that, therefore, also comes with a lot of complexity. This thesis aims to consider it from a security perspective and look into the different aspects relevant for a secure cluster configuration. Background information and preliminaries are explained in Chapter 2.

To structure security aspects related to Kubernetes clusters, we propose a layer model in Chapter 3. The model builds upon the underlying infrastructure and the Kubernetes setup itself. It then factors in security controls provided by Kubernetes to finally regard application and container security. Besides providing a holistic view, it also gives examples on how defence in depth, as

[1] `https://www.docker.com`, accessed 2019-07-25

[2] `https://docs.docker.com/engine/swarm/`, accessed 2019-08-09

[3] `http://mesos.apache.org`, accessed 2019-08-09

[4] `https://kubernetes.io`, accessed 2019-07-25

[5] `https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/`, accessed 2019-07-18

[6] `https://containerd.io`, accessed 2019-07-25

[7] `https://cri-o.io`, accessed 2019-07-25

introduced by Woodside [27], can be applied to perform damage control, if one or more security barriers are breached.

In Chapter 4, we then use this model to examine how an example application can be securely set up to run in a cluster.

Our main contributions, as concluded in Chapter 5:

- We created a structured layer model to give a holistic view of all security aspects of Kubernetes clusters.
- In the context of the model, we showed which configuration setups are necessary and should be taken with respect to real-world applications.
- Using an example application, we demonstrated how defence in depth could be applied to a Kubernetes cluster.

2 Background

In this chapter, we provide preliminary information on the concepts relevant for container orchestration security. These include containers in Section 2.1 and Kubernetes in Section 2.2.

2.1 Containers

The aim of containerisation is to provide applications with their own isolated runtime environments and simplify deployment processes. Containers package all code and dependencies an application needs and can then be independently run in various computing environments. Thereby, containers address many of the use cases previously tackled by virtual machines (VMs).

The main differences between containers and VMs, as explained by Bauer [5] and Zhang et al. [29], are that when using containers, the guest OS is not virtualised on top of a hypervisor, but rather the containers run directly on the host OS using a container daemon. On the one hand, this means that the guests are not fully isolated anymore, as they are using the same host OS. This, on the other hand, comes with significant performance and resource benefits, which is why containers are widely replacing VMs in many use cases, as found by Dawson [9].

In this thesis, we will be primarily looking at Docker[1] as the container engine, yet most principles apply to any container platform.

For containerisation, the application and everything it needs is packaged into an image that is then pushed to a *registry*, such as Docker Hub[2]. Registries can either be private or public and can be running in the cloud or locally. From there, the image can be pulled and executed by, for example, a container orchestration system such as Kubernetes.

[1] https://www.docker.com, accessed 2019-07-18
[2] https://hub.docker.com, accessed 2019-07-18

2.2 Kubernetes

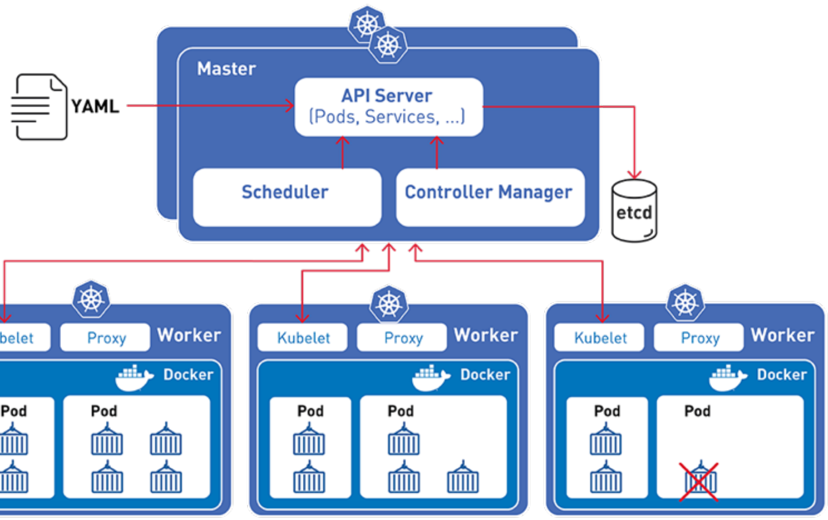

Figure 2.1: This diagram from McDonald [19] shows the structure of a Kubernetes cluster.

Kubernetes is a highly flexible and configurable system for container orchestration that provides automatic deployment, scaling, and administration of container applications, as explained in the *Kubernetes Documentation* [18].

Architecture The Kubernetes architecture is depicted in Figure 2.1. It shows a *cluster*, which is a set of *nodes*[3] that run the container applications and are displayed as blue boxes in the diagram. Nodes can either be physical or virtual machines, the latter being especially prevalent in hosted public cloud scenarios. Every cluster consists of at least one master and one worker node.

The master nodes run so-called *control plane components* that are necessary for the cluster to function. These include the *API server* and `etcd`. The *API server* exposes the *Kubernetes API*[4], which is used to control the cluster state. Administrators typically interact with the API using the CLI tool `kubectl`[5]. The central key-value database `etcd`[6] stores all cluster information including all

[3] https://kubernetes.io/docs/concepts/architecture/nodes/, accessed 2019-07-19
[4] https://kubernetes.io/docs/concepts/overview/kubernetes-api/, accessed 2019-07-19
[5] https://kubernetes.io/docs/reference/kubectl/overview/, accessed 2019-07-19
[6] https://etcd.io, accessed 2019-07-25

Kubernetes objects and is only supposed to be accessed directly by the API server. The nodes are provided with the necessary information via the API server and `etcd`.

Worker nodes run the container daemon and the *kubelet*, which is an agent responsible for managing the node. On the workers, the actual containerised applications run. The containers reside within *pods*[7]. A pod runs one or more containers on the same node. Pods are the smallest units that can be scheduled to run. They, like all other Kubernetes objects, are declaratively specified in the *YAML* format[8]. These specifications include the URL to the container images in the registry, metadata and configuration information. Examples of specification files are given throughout this thesis and in Appendix A.

To create a pod, its specification is submitted to the Kubernetes API. The control plane takes care of the deployment and scheduling of the specified pod. To do so, first, the *Controller manager* determines the demand for pods, then, the *Scheduler* finds a node on which they should run. Finally, the `kubelet` agent on the node retrieves the configurations from the API server and starts the pod.

Scaling The power of Kubernetes lies in its more advanced concepts, including *Deployments*[9] and *ReplicaSets*[10]. These can be used to define how multiple instances of pods should be run. Kubernetes then takes care that the cluster is always in the desired state, updates are rolled out, and new replicas are spun up if required.

Configuration and Secrets Management For configuration and credentials management, Kubernetes provides the objects *ConfigMap* and *Secret*[11], which can be configured using YAML specifications. They are then mounted into the containers' file systems or passed along using environment variables so that applications can retrieve them.

[7] https://kubernetes.io/docs/concepts/workloads/pods/pod/, accessed 2019-07-19

[8] https://yaml.org, accessed 2019-07-19

[9] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, accessed 2019-07-19

[10] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/, accessed 2019-07-19

[11] https://kubectl.docs.kubernetes.io/pages/app_management/secrets_and_configmaps.html, accessed 2019-07-18

Exposing applications To expose applications running on a set of pods, *Services*[12] can be used. They can either be employed for internal communication within the cluster or can be bound to an *Ingress*[13] to expose them to the public internet.

[12] https://kubernetes.io/docs/concepts/services-networking/service/, accessed 2019-07-18

[13] https://kubernetes.io/docs/concepts/services-networking/ingress/, accessed 2019-07-19

3 Kubernetes Cluster Security

In Kubernetes there are many aspects to consider when looking at it from a security perspective. With its flexibility also comes considerable complexity. In order to come by it and discuss it in a structured manner, we have created the model depicted in Figure 3.1. It is aimed at giving a holistic view over all aspects that might influence the security of a Kubernetes cluster and uses structure ideas from Abbassi [3] and Rice and Hausenblas [22].

The model is composed of the following four layers:

1. The lowest layer we call **Base Infrastructure Security** and it concerns all components on which Kubernetes itself builds. These include the Operating System, the container engine (most likely Docker), and the infrastructure of the public or private cloud provider when for example using Google Cloud Platform or Amazon Web Services as an *Infrastructure as a Service (IaaS)* provider. This layer is described in Section 3.1.
2. Upon that builds **Kubernetes Infrastructure Security** which is described in detail in Section 3.2. It concerns everything related to Kubernetes' control plane components and their configuration in terms of security. Potential issues there include abuse of the internal Kubernetes APIs, such as the Kubelet API, or intercepting internal traffic between control plane components.
3. Additionally to the configuration of the Kubernetes control plane components themselves, there are also security components provided by Kubernetes to secure clusters. These components include Kubernetes' authentication and authorisation mechanisms, pod security policies, secrets management and many more. We group them together in the layer **Kubernetes Security Controls** as described in Section 3.3.
4. The top-most layer is called **App and Container Security** and makes use of all layers underneath it. On this layer, the actual applications run in containers, which in turn are located in pods. Exploiting of vulnerabilities in the application code might breach this layer. These topics are described in detail in Section 3.4.

4	App and Container Security	• Application vulnerabilities • Pod security • Container security
3	Kubernetes Security Controls	• Custers and Namespaces • Authentication, Authorisation and Admission Control • Networking
2	Kubernetes Infrastructure	• Kubernetes Control-Plane • Kubelets • Control traffic
1	Base Infrastructure	• Operating System • Container Engine • Hardware Infrastructure (IaaS)

Figure 3.1: This security model shows the different layers that have to be considered when holistically regarding Kubernetes security.

The layered architecture also makes sense when considering damage control aspects of Kubernetes security: As one layer on the top breaches, the lower layers can still prevent further damage. For example, when there is a vulnerability in the application code on layer 4, a properly configured set of permissions on layer 3 for the pod might still prevent further damage. Inversely, an insecure port left open in a control plane component, such as the API server, on layer 2 might allow an attacker to infiltrate a whole cluster, taking over all pods, containers and applications on the upper layers.

3.1 Base Infrastructure Security

The base infrastructure, in which a Kubernetes cluster is run, sets the foundation for the whole cluster's security, as explained in Abbassi [1]. However well-tuned the Kubernetes cluster configuration is, a breach in the underlying infrastructure of a Kubernetes node can compromise the whole cluster. Depending on whether a cluster runs in a public or private cloud environment, some of the following guidelines may apply. When a cluster runs in a managed environment, such as on Google Kubernetes Engine (GKE) or Amazon Elastic Container Service for Kubernetes (Amazon EKS), a lot of this configuration may already be taken care of by the provider.

Operating System Every component in Kubernetes is run by an underlying operating system, mostly a flavour of Linux. While vulnerabilities in the OS itself are outside the scope of this thesis, some general guidelines should be followed to minimise the risk of a breach on OS level:

- An operating system with container support that has a minimal attack surface should be used, such as *CoreOS Container Linux*[1] or *k3OS*[2] that has been specifically designed for Kubernetes.
- Just like Kubernetes should be regularly patched, the operating system kernel should also be running at the most recent version at all times.
- For many Linux distributions there are hardening guides that can be used as a reference for the system configuration, such as the *CoreOS Container Linux hardening guide*[3].

Container Runtime Upon the OS builds the container runtime in which all containers are run in the cluster. At the time of this writing, mostly the runtime is *Docker*[4], but also alternatives that comply with the *Kubernetes Container Runtime Interface*[5] such as *containerd*[6] gain traction.

Besides the OS, also the container runtime should always be up-to-date and configured securely. It is essential not to forget about it in the stack and acknowledge that containers are not virtual machines and do not provide the same kind of isolation. A breach out of a container can, therefore, mean that not only a pod, but an entire node can be compromised.

Network environment Outside access to cluster resources should be as restricted as possible. In most configurations, access to functionality provided by the cluster is exposed only via Kubernetes Services. Hence it is neither necessary nor recommended for every node to be exposed to the public internet. The attack surface of the whole cluster can, therefore, be reduced by putting all nodes in a private network and exposing only the services via a load balancer.

[1] `https://coreos.com/os/docs/latest/`, accessed 2019-06-10
[2] `https://k3os.io`, accessed 2019-06-10
[3] `https://coreos.com/os/docs/latest/hardening-guide.html`, accessed 2019-06-10
[4] `https://www.docker.com`, accessed 2019-06-10
[5] `https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/`, accessed 2019-06-10
[6] `https://containerd.io`, accessed 2019-06-10

Trusted Platform Modules Trusted Platform Modules (TPMs), as standardised in *Information technology – Trusted platform module library – Part 1: Architecture* [15], can be a powerful addition to the base infrastructure to increase its security. As demonstrated by Tcherniakhovski and Lytvynov [25], TPMs can be used to store cryptographic keys that stay secure from an attacker even if an entire node is compromised.

3.2 Kubernetes Infrastructure Security

The security of Kubernetes infrastructure concerns the configuration of all components of the Kubernetes control plane and their internal communications. When setting up and administering a cluster, there are many configurations that, at first sight, do not appear to be critical to security. Still, the following aspects must not be neglected to ensure a secure cluster setup. Many best practice guidelines are listed in Rice and Hausenblas [22].

3.2.1 Cluster installers

While it is possible to set up a cluster completely manually[7], most are set up using cluster installers such as `kubeadm`[8] or `kops`[9]. They can be used to bootstrap all control plane components, set up the Public Key Infrastructure (PKI) and network configuration, and join worker nodes to the cluster.

Even though most installers come with sane default configurations, it is crucial to verify them and check what is done under the hood, as emphasised by Abbassi [2].

3.2.2 API server

The API server is an essential control plane component, as an attacker that gains control over it has the equivalent of root access to the whole cluster, as described by Rice and Hausenblas [22]. The critical aspect of how the API itself is secured and how the API server fulfils its role as the cluster's Certificate Authority (CA)

[7] https://github.com/kelseyhightower/kubernetes-the-hard-way, accessed 2019-06-11

[8] https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/, accessed 2019-06-11

[9] https://github.com/kubernetes/kops, accessed 2019-06-11

is described in Section 3.3.2, as this resembles a Kubernetes security control and is therefore grouped in the corresponding layer of our model.

Still, any misconfiguration of the API server might mean leaving the front door to the cluster open, which is why its configuration should always be reviewed from a security perspective. This includes the fact that the Kubernetes API should generally not be exposed to the public internet if there is no pressing need.

3.2.3 Kubelets

A `kubelet` is an agent running on every node that is responsible for running its containers, reporting metrics and similar. It receives its commands from the control plane. For that, it provides the so-called *Kubelet API*, which is undocumented, as it is not supposed to be used by anyone other than the control plane.

This API needs to be adequately protected, as otherwise it can be misused as shown by Urcioli [26]. This misconfiguration gave any attacker with network access to nodes an unauthenticated API backdoor to the cluster. In the Kubernetes version 1.15, which is the most current one at the time of this writing, the API is protected by default. The insecure port is closed and the secure ones calls back to the API server to verify whether a request should be allowed (see *Node Authorisation* in Section 3.3.2).

It is worth noting that disallowing node network access, as recommended in Section 3.1, could have prevented this exploit in the first place.

3.2.4 etcd

`etcd` is the central storage location in Kubernetes that stores all cluster data in a key-value data structure. The API server is the only component that is supposed to access it directly.

Access restriction In order to enforce that only the API server can communicate with `etcd`, network policies, as later described in Section 3.3.3, should be set up and an adequate certificate configuration[10] should be put in place.

[10] `https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/security.md`, accessed 2019-06-11

Encryption By default, all data in `etcd` is stored in plain text. As it also stores all secrets, sensitive information might be exposed, if the data on the volume used by `etcd` leaks. In order to mitigate that, encryption can and should be enabled by providing the API server with an encryption configuration using the startup parameter `--encryption-provider-config` as described in the *Kubernetes Documentation* [18][11].

3.2.5 Kubernetes Dashboard

The Dashboard[12] is a Web UI for administrating a Kubernetes cluster and gives a convenient overview over the pods, deployments, services and other objects currently deployed. However, in a recent incident at Tesla, described in Goodin [11], a cluster was compromised due to an insufficiently secured Dashboard and used for crypto mining. This illustrates how leaving it exposed to the public internet or leaving it configured insecurely can compromise the security of an entire cluster. Therefore Rice and Hausenblas [22] suggest to follow these guidelines to secure it:

- Only authenticated users should be allowed access.
- The service account the Dashboard is using should have limited privileges so that users cannot misuse its permissions and rather log in with their own users.
- The dashboard should not be exposed to the public internet.

3.3 Kubernetes Security Controls

Kubernetes provides cluster operators with an extensive set of options and tools to tweak the security of a cluster. These build upon the Kubernetes and base infrastructure and are very important to configure correctly.

[11] https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/, accessed 2019-06-11

[12] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/, accessed 2019-07-19

3.3.1 Namespaces

As described in the *Kubernetes Documentation* [18][13], almost every resource in a Kubernetes cluster belongs to a namespace. Exceptions include only namespaces themselves and some low-level resources such as nodes.

All resources belonging to the control plane components reside within the `kube-system` namespace, while by default all other objects are located in the `default` namespace.

Generally, namespaces can be used to avoid naming conflicts, but also to allow for finer grained access control. For example, using API access control, a role can be created so that a user can list all pods from a particular namespace, but not for any other namespace, as further described in Section 3.3.2. Using this feature makes sense primarily when the cluster is used by a large number of users from, for example, different projects or departments.

Namespaces can sometimes alleviate the need for creating separate clusters to isolate components from each other. Some use cases might include different namespaces per team, application or environment (such as different namespaces for development, testing and production).

However, as pointed out in Altarace and Wilkin [4], while namespaces are suited well for logical partitioning and assigning permission sets on API access level, they do not enforce the partitioning by setting firewall rules or such. Any cluster user or resource can access any other resource in the cluster, even when they are in different namespaces. To achieve partitioning on the network level, see Section 3.3.3.

3.3.2 API Access Control

As explained in Chapter 2, the only component in a Kubernetes cluster that is allowed to modify the cluster state in `etcd` directly is the API server. Therefore all requests that involve reading or modifying the cluster state must be performed via the Kubernetes API. It is well-documented in *Kubernetes API Documentation* [17] and is very powerful, as it is also used by all control plane components to communicate with each other. Hence it needs to be carefully protected from unauthorised access. The API server achieves that using three steps to very if and how a request should be performed, as depicted in Figure 3.2.

[13] `https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/`, accessed 2019-05-31

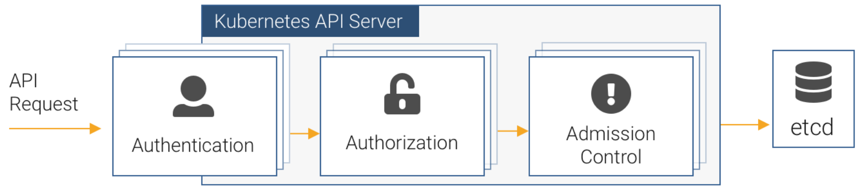

Figure 3.2: This image from Abbassi [2] displays the steps performed by the API server before it persists a request made to it.

First, **Authentication** is performed to determine the identity of the user or system that is trying to access the API. After the identity of the accessing party has been determined, **Authorisation** is used to decide whether they are allowed to access or modify the requested resource. Finally, **Admission Controllers** are applied to the request to validate or mutate it before it is ultimately persisted.

Authentication

Users in Kubernetes can be either Service Accounts or normal (human) users.

Service Accounts are managed directly by Kubernetes and give identities to apps running inside pods as described in *Kubernetes Documentation* [18][14]. With these identities, the applications can access the API server. Service Accounts are Kubernetes objects and are stored in `etcd`. A pod can be configured to use a particular service account, which results in the corresponding certificates being mounted into the pod's file system. With these, the application running in it can authenticate with the API server.

For human users, no Kubernetes objects in `etcd` exist. The administration of those is outsourced to an external entity. The API server offers multiple strategies to authenticate such users, a subset of which are the following:

- The default way of authentication between control plane components is using **X.509 certificates** within the PKI of the cluster. For that to function, the API server is handed the CA file using the start-up parameter `--client-ca-file=/path/to/ca.crt`. Users can then authenticate using a certificate signed by the provided CA.
  At the time of this writing, however, Kubernetes does not have support for Certificate Revocation Lists (CRL) or Online Certificate Status

[14] `https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/`, accessed 2019-06-10

Protocol (OCSP), with which certificates could be revoked. The issue is currently under debate within the Kubernetes developer community[15]. In the meantime, authorisation has to be used to strip users from permissions retroactively.
- An **Authentication Proxy** can be used to outsource the task of authentication to an external entity that might run within our outside of the cluster.
- In order to enable support for Single Sign-On (SSO), the API server can be configured to use **OpenID Connect (OIDC)**[16] for authentication. It builds upon the OAuth 2.0 flow and allows the use of external identity providers that implement the standard, such as the common *"Sign in with Facebook/Google/Microsoft"* known from various places on the web.
- For testing and demo purposes it is also possible to use a **static password file** as per the definition in the RFC 7617 "The 'Basic' HTTP Authentication Scheme" by Reschke [21]. However, it is rarely practical in real-world use cases as the password file needs to be maintained manually.

Authentication considers no actual permissions. It only determines the identity of the authenticating entity and several attributes about it, such as the username and group memberships. This information is passed on to the next step, authorisation, in which the actual mapping step of who is allowed to do what happens.

Authorisation

As is the case for authentication, Kubernetes also provides several options for how authorisation can be performed as defined in the documentation *Kubernetes Documentation* [18][17]:

- **Node Authorisation** is used only by `kubelets`, so that nodes are equipped with the minimum set of permissions they need to operate within the cluster.
- **Attribute-based Access Control (ABAC)** grants permissions by explicit policies based on the users' attributes, such as name, location or department. ABAC, however, has been deprecated since Kubernetes version 1.6 and is not recommended to be used anymore.

[15] https://github.com/kubernetes/kubernetes/issues/18982, accessed 2019-05-28
[16] https://openid.net/connect/, accessed 2019-05-28
[17] https://kubernetes.io/docs/reference/access-authn-authz/authorization/, accessed 2019-05-31

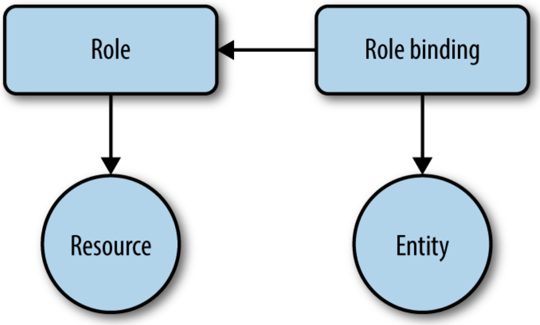

Figure 3.3: This graphic from Hausenblas and Goasguen [13] depicts the basic building blocks of RBAC.

- **Role-based Access Control (RBAC)** defines roles that come with certain permissions. These roles can then be bound to users to grant them access.
- **Webhook Mode** queries an external REST service to outsource the authorisation to it.

As Yuan and Tong [28] point out, conceptually there are advantages and disadvantages to both ABAC and RBAC. While ABAC provides more flexibility, RBAC lends itself better for analysis and risk assessment. The main advantage of ABAC is that it can perform restrictions directly based on users' attributes. In the context of Kubernetes, however, its developers Jacob Simpson and Cullen [16] argue that only RBAC should be used anymore. That is acceptable primarily as the third step of API access control in Kubernetes, *Admission Control*, can provide filters to accommodate for such finer-grained control.

RBAC is the recommended way of performing authorisation at the time of this writing, as it is easier to understand and configure than ABAC while retaining most of its configuration power. Therefore, we will only consider it in this work and disregard the other options.

RBAC The basic building blocks of RBAC are depicted in Figure 3.3. *Roles* are used to define access to certain *Resources*. Rules can either be defined cluster-wide or namespace wide, depending on the objects they control access to.

A role that allows reading pods is created by applying a YAML file like this:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
```

```
  name: pod-reader
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Besides some metadata, a role defines certain API groups (in the above example the `""` indicate the core API), resources and certain verbs that correspond to the actions that are allowed on the resource.

In order to assign the role to an *Entity*, in this example case a `ServiceAccount` called `example-app-sa`, a *RoleBinding* for the namespace `default` can be created by applying a YAML file like this:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: example-app-sa
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Besides *Roles* and *RoleBindings*, there are also *ClusterRoles* and *ClusterRoleBindings* that work similarly, just that they manage permissions on the scope of the whole cluster instead of being bound to a specific namespace.

As for all security configurations, the principle of Least Privilege applies especially when it comes to the configuration of RBAC. It has been initially defined by Saltzer [23] as "Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.".

Applied to Kubernetes API access control, this means that an application which does not need to access the Kubernetes API to function, as is likely the case for the vast majority of applications, should not be allowed to do so.

As pointed out by Rice and Hausenblas [22], it is therefore recommended to disable the auto-mounting of the default Service Account into pods by setting `automountServiceAccountToken: false` in the pod specification.

Section 4.2.3. shows another practical example of an RBAC configuration that adheres to this principle.

Admission Control

As described in Isberner [14] and *Kubernetes Documentation* [18][18], admission control allows for a set of plug-ins to enforce rules about how a cluster is used, by being applied after a request to the API server has passed authentication and authorisation. These plug-ins can be either *mutating*, *validating* or both. First, all mutating admission controllers are executed, then the validating controllers. If any of them rejects the request, the processing is aborted and the request is rejected.

An especially important admission controller with respect to security is the built-in `PodSecurityPolicy`. It can be used to forbid containers running with privileged users or set containers' file systems to read-only. Both of these are sensible best practices, as further explained in the context of container security in Section 4.2.4.

Besides standard admission controllers that ship with Kubernetes and should be disabled or altered only under certain circumstances, there is also support for custom controllers that build upon the standard `MutatingAdmissionWebhook` or `ValidatingAdmissionWebhook`. These allow for high flexibility as their admission is based on a REST call to a service running within the cluster. This feature can be used to develop custom restrictions, such as this example[19] that causes all containers to be run with a non-root user, except they are explicitly marked to do otherwise.

3.3.3 Networking

Communication between Kubernetes components and the outside world is a central aspect for Kubernetes to function. As described in detail in the article

[18] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/, accessed 2019-05-25

[19] https://github.com/stackrox/admission-controller-webhook-demo, accessed 2019-05-28

series Betz [6] and in the documentation *Kubernetes Documentation* [18][20], it can be broken down in to several aspects:

- Containers within the same pod communicate with each other via localhost.
- Pods communicate with other pods and with the control plane adhering to the Kubernetes Networking model[21] that defines an overlay network for that purpose. The model is aimed to be very flexible and should allow for easy porting from virtual machines. It mainly builds upon the premise that pods can be addressed with their own IP addresses, despite multiple pods running on the same node. The network model is not directly implemented in the Kubernetes core, but rather different plug-ins can be used to meet its requirements. These plug-ins integrate with nodes and their kernels to establish the layer 3 (network layer) communication as defined in Day and Zimmermann [10].
- Communication with the outside world is handled by services that can be bound to Ingress[22] objects.

From a security perspective, an especially important topic in networking are Network Policies[23], which allow configuration on which pods are allowed to communicate with each other and with other network endpoints. Network policies are realised in the Kubernetes object `NetworkPolicy`. Still, not all networking plug-ins support them.

An example for a `NetworkPolicy` could look like this:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: example-policy
spec:
  podSelector:
    matchLabels:
      app: example-app
```

[20] https://kubernetes.io/docs/concepts/cluster-administration/networking/, accessed 2019-06-10

[21] https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model, accessed 2019-06-10

[22] https://kubernetes.io/docs/concepts/services-networking/ingress/, accessed 2019-06-10

[23] https://kubernetes.io/docs/concepts/services-networking/network-policies/, accessed 2019-05-28

```
  policyTypes:
  - Egress
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

The `policyTypes` specify whether egress traffic, ingress traffic or both are affected by this policy. The white-list principle applies so that only the specified traffic is allowed. This example applies to all pods with the label `app` set to `example-app`. It allows incoming traffic only from pods with the label `access` set to `true` and prohibits all outgoing traffic.

In most cases, it makes sense to prohibit egress by default. In this case, even if a pod is entirely hijacked, it cannot directly communicate with the outside world via a reverse shell (connect-back shell). Of course, however, such a restriction could be bypassed by using a bind shell that listens on a specific port for an incoming connection from the attacker as explained in *Hacking with Netcat part 2: Bind and reverse shells* [7]. Still, such an approach would require more effort from the attacker.

An example of how Network Policies can be used to secure a real-world cluster is given in Section 4.2.3.

3.4 App and Container Security

The top-most layer of the model concerns the applications that run in the Kubernetes cluster within their container environments. Their security relies on the lower layers while also being protected by them.

3.4.1 Application Security

The largest attack surface on this layer naturally are the applications themselves. As the whole class of application vulnerabilities cannot be mitigated entirely, Kubernetes provides many damage control tools on lower levels that limit an attacker's possibilities, even when an application has been compromised. This is a good practice example of defence in depth, as explained in Woodside [27].

Image Scanning Often, the containers that are run in a cluster build upon public images and included application packages. Image scanning tools, such as *Anchore Engine*[24] and *Clair*[25], scan images before they are deployed and analyse whether they contain known vulnerabilities. Using the admission controller `ImagePolicyWebhook`, as explained in Section 3.3.2, API access control can be configured to refuse the deployment of images for which vulnerabilities are reported.

3.4.2 Container Configuration

In this section, several precautions are explained that can be taken on container level to increase cluster security.

Non-privileged containers As pointed out in detail in Hausenblas [12], applications in containers should run with non-privileged users, as most of them should not require root permissions. This restriction can be enforced by adding the following statements to the `PodSecurityPolicy` configuration:

```
allowPrivilegeEscalation: false
runAsUser:
  rule: 'MustRunAsNonRoot'
```

One side effect of not running as root is that applications can not bind to well-known ports (< 1024), but this can be easily mitigated by using the port binding capabilities of the container runtime.

Minimal containers Containers should only run the bare minimum of applications they require to fulfil their aim. It is therefore considered bad practice to run, for example, an SSH server to do maintenance work, as explained in Petazzoni [20]. Most actions that would require having an SSH server running can also be achieved using Kubernetes features.

[24] https://github.com/anchore/anchore-engine, accessed 2019-06-28
[25] https://github.com/coreos/clair, accessed 2019-06-28

Secrets and Configuration Secrets and configuration items should not be baked into the container images, but should instead be managed using the Kubernetes objects `Secret` and `ConfigMap` respectively. These can then either be exposed to the containers using environment variables or by mounting them to a volume accessible to the container. By doing so, they can be more easily adapted and are not prone to leak when a container image leaks.

Rule-based Execution As explained in the *Kubernetes Documentation* [18][26], a `PodSecurityPolicy` can be used to enforce several rule-based execution restrictions. The configuration options include:

- `seccomp`, as documented in *SECCOMP(2) Linux Programmer's Manual* [24], can limit the allowed system calls for user-space applications. At the time of this writing, `seccomp` is disabled by default in Kubernetes so that even the default Docker profile[27] does not apply.
- Policies of SELinux[28] and AppArmor[29] can be configured for finer-grained access control.
- Linux capabilities, as explained in Boelen [8], are a kernel feature that can grant a process running as root some, but not all privileged capabilities. In Kubernetes, the whitelist of capabilities that can be added to a container can be configured.

[26] `https://kubernetes.io/docs/concepts/policy/pod-security-policy/`, accessed 2019-06-25

[27] `https://docs.docker.com/engine/security/seccomp/`, accessed 2019-06-28

[28] `https://selinuxproject.org/page/Main_Page`, accessed 2019-06-25

[29] `https://kubernetes.io/docs/tutorials/clusters/apparmor/`, accessed 2019-06-25

4 Securing an Example Application

In this chapter, the security concepts introduced above are put into practice by applying them to a real-world application. This application has been created solely for demonstration purposes to resemble a use case that makes extensive use of Kubernetes' features while still retaining simplicity to be generally applicable and easy to understand. The full configuration files of the application can be found in Appendix A.

4.1 Example Application

Functionality The example application is a simple dashboard for Kubernetes to view Pods in certain or all namespaces. It can also display and scale ReplicaSets. A screenshot of it is depicted in Figure 4.1. For providing this functionality, it queries the API server for information, communicates between different components using services and exposes its functionality via an ingress.

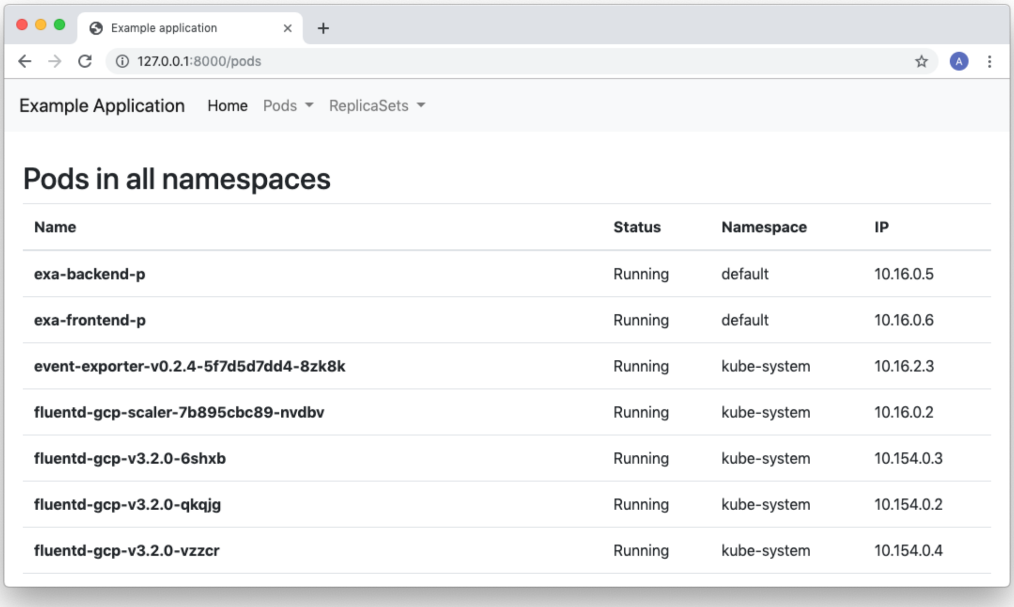

Figure 4.1: This screenshot depicts the front page of the example application.

Setup The application is written in Python and makes use of Flask[1] as well as the Kubernetes Python Client[2].

The setup consists of a backend part, `exa-backend`, and a frontend part, `exa-frontend`. `exa-backend` provides a REST API and directly accesses the Kubernetes API, while `exa-frontend` uses the backend for retrieving the data.

[1] `http://flask.pocoo.org/`, accessed 2019-07-02
[2] `https://github.com/kubernetes-client/python`, accessed 2019-07-02

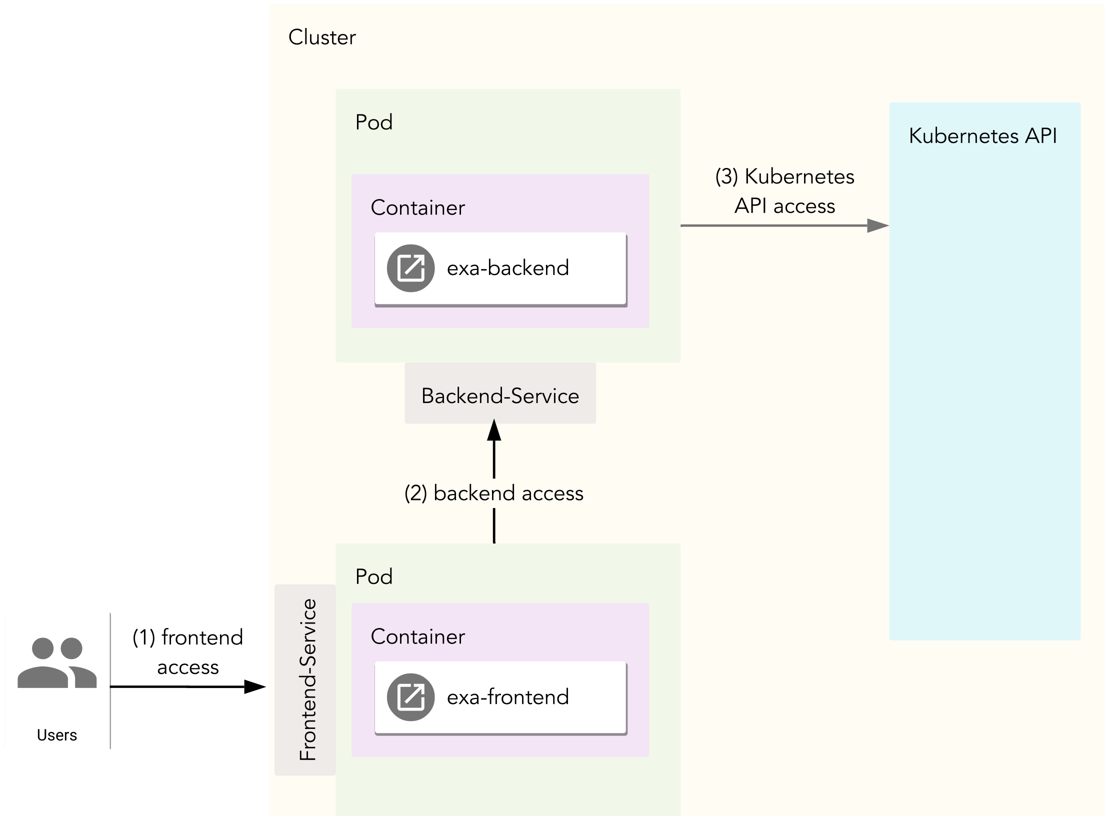

Figure 4.2: This architecture diagram shows how the example application runs within the Kubernetes cluster.

The Kubernetes cluster is run in Google Kubernetes Engine (GKE) on Google Cloud Platform (GCP)[3]. The architecture of the cluster is depicted in Figure 4.2. The frontend and backend components run in separate pods. Users access the frontend via an ingress that maps to the frontend service. The frontend then retrieves all information by calling the backend service, which in turn makes the request to the Kubernetes API to perform the user's query. We are choosing this setup intentionally so that the frontend does not have direct access to the API to resemble a typical frontend-backend architecture. The following sections describe how we use Kubernetes to enforce these environment restrictions.

[3] `https://cloud.google.com/kubernetes-engine/`, accessed 2019-07-02

4.2 Security Measures

In this section, the measures that were taken to secure the setup are explained. They are structured according to the layer model introduced in Chapter 3.

4.2.1 Base Infrastructure Security

As this setup uses GKE as an infrastructure provider, most considerations for base infrastructure security are already taken care of by Google[4]. By default, GKE clusters run the newest version of Google's Container-Optimized OS[5] that comes with a current version of the container runtime as well.

However, as all cluster components are running within the GCP environment, they are also subject to Google's Cloud Identity & Access Management (IAM)[6]. It works alongside Kubernetes RBAC, yet requires some special attention when applications within the Kubernetes cluster access other resources on GCP outside of the cluster[7].

4.2.2 Kubernetes Infrastructure Security

When it comes to Kubernetes' infrastructure components, most relevant security aspects are still closely tied to the underlying base infrastructure, in this case, GKE.

API Server The aforementioned IAM, by default, allows login using username and password. We disable it in the cluster configuration in the GCP web interface and only use certificates, which we regularly rotate[8], to authenticate within the cluster.

[4] `https://cloud.google.com/kubernetes-engine/docs/concepts/security-overview`, accessed 2019-07-09

[5] `https://cloud.google.com/container-optimized-os/docs/concepts/features-and-benefits`, accessed 2019-07-09

[6] `https://cloud.google.com/iam/`, accessed 2019-07-09

[7] `https://cloud.google.com/kubernetes-engine/docs/how-to/iam`, accessed 2019-07-09

[8] `https://cloud.google.com/kubernetes-engine/docs/how-to/credential-rotation`, accessed 2019-07-09

Secret Encryption As explained in Section 3.2.4, it is vital to encrypt all secrets stored in `etcd`. GKE already provides encryption at rest by default[9], in case an attacker gains access to an offline copy of the `etcd` database. However, to protect against the case that an attacker gains online access to the node that runs `etcd`, we also configure application-layer encryption. To set it up, we generate a key, store it in Google's Cloud Key Management Service (KMS)[10] and hand it to the cluster configuration using the command line option `--database-encryption-key`[11].

Dashboard As the standard Kubernetes dashboard is deprecated and disabled by default on GKE, no further configuration was necessary to secure the setup. The GCP Console[12], which acts as a replacement for the standard dashboard on GKE, again uses Google's IAM and therefore needs no additional protection. It is conceptually not accessible by unauthenticated users.

4.2.3 Kubernetes Security Controls

We put the following Kubernetes security controls in place to secure our cluster configuration.

Namespaces The example application is set up to run in its own namespace `admin` to be isolated from other components running in the cluster. The namespace is created by applying a YAML file like this:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: admin
  labels:
    name: admin
```

[9] https://cloud.google.com/security/encryption-at-rest/default-encryption/, accessed 2019-07-09
[10] https://cloud.google.com/kms/docs/, accessed 2019-07-09
[11] https://cloud.google.com/kubernetes-engine/docs/how-to/encrypting-secrets, accessed 2019-07-09
[12] https://console.cloud.google.com/, accessed 2019-07-09

All objects (pods, service accounts, services, etc.) for the example application are created in this namespace by running `kubectl apply -f <yaml-file> --namespace=admin`.

API Access Control As the example application needs to access the Kubernetes API, access control for it needs to be set up.

Authentication In our cluster configuration, authentication is done using manually distributed X.509 certificates, as explained in Section 3.3.2, which is viable as the cluster only has few users that need to access this admin application. We made this choice not out of any security considerations, so any other means of authentication could be used as well.

The frontend application does not directly access the Kubernetes API and therefore does not need an identity in the cluster. It is consequently set up not to use a service account at all:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: exa-frontend-p
  # ...
spec:
  # ...
  automountServiceAccountToken: false
```

As anonymous API access is disabled by default, the pod `exa-frontend-p` now has no access to the API directly, even if an attacker gains control over the whole pod.

In contrast, the backend needs to access the Kubernetes API, which is why we create an identity, a service account called `exa-backend-sa`, for it using this configuration file:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: exa-backend-sa
```

We then assign the service account to its pod by adapting the pod's configuration accordingly:

```
apiVersion: v1
kind: Pod
metadata:
  name: exa-backend-p
  # ...
spec:
  # ...
  serviceAccountName: exa-backend-sa
```

This configuration causes the access tokens to be mounted into the container's file system in `/var/run/secrets/kubernetes.io/serviceaccount`[13]. The back-end application fetches them and uses them when performing requests to the API server.

Authorisation For authorisation, we use RBAC for all the reasons explained in 3.3.2. To equip the pod with the privileges it needs, we define a role that is bound to the pod resource:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: admin
  name: pod-reader
rules:
- apiGroups: [""] # "" = core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

To create the link between the role and the service account of the backend pod, a `RoleBinding` is set up:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
```

[13] https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/, accessed 2019-07-16

```
  name: exa-backend-read-pods
  namespace: admin
subjects:
- kind: ServiceAccount
  name: exa-backend-sa
  namespace: admin
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

With this configuration, the backend pod has the privileges to `get`, `watch` and `list` all pods in the namespace `admin`. To grant these privileges for all namespaces, instead of a `Role`, a `ClusterRole` and instead of a `RoleBinding`, a `ClusterRoleBinding` needs to be created. The full RBAC configuration for our cluster can be found in Appendix A.

Admission Control In our cluster, we use the standard set of enabled admission controllers with the addition of the controller responsible for handling pod security policies, as further explained in the context of container security in Section 4.2.4.

Networking In the example application's setup, the pod `exa-backend-p` never needs to be accessed directly from outside the cluster. Specifically, `exa-frontend-p` is the only pod that ever needs to access it. To lock down network traffic to these requirements, we make use of network policies, as explained in Section 3.3.3.

To do so, we enable network policies in the GKE cluster[14] and apply the following policy:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: exa-backend-policy
spec:
  podSelector:
    matchLabels:
```

[14] `https://cloud.google.com/kubernetes-engine/docs/how-to/network-policy`, accessed 2019-07-14

```
      app: exa-backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access-exa: "true"
```

As the pod configuration for the frontend contains the label `access-exa` set to `true` (see Appendix A, Listing 1), it is allowed to the access the backend, but no other pods or external resources are.

4.2.4 App and Container Security

Both `exa-frontend` and `exa-backend` run as non-root users and have disallowed privilege escalation in the security contexts of their pod configurations:

```
securityContext:
    allowPrivilegeEscalation: false
    runAsUser: 1000
```

To enforce non-privileged containers as a general policy in the cluster, we apply a `PodSecurityPolicy` (see Appendix A, Listing 11), which also enables seccomp and AppArmor configurations, volume restrictions and a read-only file system. Our policy mainly builds upon the restricted example in the *Kubernetes Documentation* [18][15].

To activate it, we bind it to a role, connect the role to all users and service accounts using a role binding and activate the responsible admission controller in GKE[16].

The `PodSecurityPolicy` applies whenever a pod is attempted to be created or updated. Before the change is allowed to go through, it is checked whether it fulfils the criteria in the policy. If it does not, either container creation or container configuration will fail with an error and the insecure container is not started.

[15] https://kubernetes.io/docs/concepts/policy/pod-security-policy/, accessed 2019-07-16

[16] https://cloud.google.com/kubernetes-engine/docs/how-to/pod-security-policies, accessed 2019-07-16

5 Conclusion

Kubernetes is a flexible, state-of-the-art system for container orchestration in the modern age, that also brings a lot of complexity. In this thesis, we have taken a holistic view of the relevant security aspects of container orchestration in Kubernetes and categorised them into a layer model. We demonstrated, how an example application can be run securely in a Kubernetes cluster on Google Kubernetes Engine (GKE) and which configurations are necessary to ensure that.

Our research demonstrated that Kubernetes and its installers mostly already come with secure default setups. However, many configurations cannot be given by default, as cluster installers cannot anticipate the logic and premises of the applications run in real-world clusters. Consequently, aspects such as custom RBAC configurations, network policies and pod security policies always need to be manually configured by a cluster administrator.

While the technical details of how container orchestration security is approached will likely change in future, the underlying concepts will remain. Even ever-improving default setups cannot entirely replace a thorough review of the cluster paired with necessary custom security configurations.

Appendix

Appendix A: Configuration for Example Application Cluster

This is a collection of all relevant Kubernetes configuration files applied to the cluster in which the example application runs.

Frontend Components

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: exa-frontend-p
  labels:
    # to allow network access to exa-backend
    # (enforced by network policy)
    access-exa: "true"
spec:
  containers:
    - image: docker.io/deutschmann/exa-frontend
      name: exa-frontend
      ports:
        - containerPort: 8000
          name: http
          protocol: TCP
      env:
        # read by the application at runtime
        - name: FLASK_DEBUG
          value: "1"
        - name: BACKEND_NAME
          value: "exa-backend" # backend service name
        - name: BACKEND_PORT
```

```
          value: "80" # backend service port
      securityContext:
        allowPrivilegeEscalation: false
        runAsUser: 1000
  automountServiceAccountToken: false # SA disabled
```

Listing 1: Configuration file for exa-frontend pod

```
kind: Service
apiVersion: v1
metadata:
  name: exa-frontend
spec:
  type: NodePort
  selector:
    app: exa-frontend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8000
```

Listing 2: Configuration file for exa-frontend service that routes to the exa-frontend pod

Backend Components

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: exa-backend-sa
```

Listing 3: Configuration file for the service account used by the backend application to access the Kubernetes API

```
apiVersion: v1
kind: Pod
metadata:
  name: exa-backend-p
  labels:
```

```yaml
    app: exa-backend
spec:
  containers:
    - image: docker.io/deutschmann/exa-backend
      name: exa-backend
      ports:
        - containerPort: 5000
          name: http
          protocol: TCP
      env:
        - name: FLASK_DEBUG
          value: "1"
        - name: LOCAL_CLUSTER
          value: "0"
      securityContext:
        allowPrivilegeEscalation: false
        runAsUser: 1000
  serviceAccountName: exa-backend-sa
```

Listing 4: Configuration file for exa-backend pod

```yaml
kind: Service
apiVersion: v1
metadata:
  name: exa-backend
spec:
  selector:
    app: exa-backend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

Listing 5: Configuration file for exa-backend service that routes to the exa-backend pod

RBAC

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
```

```
metadata:
  name: pod-reader
rules:
- apiGroups: [""] # "" = core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Listing 6: Configuration file to define a cluster role that allows reading access to pods

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rs-admin
rules:
- apiGroups: ["extensions", "apps"] # "" = core API group
  resources: ["replicasets"]
  verbs: ["get", "watch", "list", "scale"]
```

Listing 7: Configuration file to define a cluster role that allows reading and scaling replica sets

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: exa-backend-read-pods
subjects:
- kind: ServiceAccount
  name: exa-backend-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Listing 8: Configuration file that binds the cluster role `pod-reader` defined in Listing 6 to the service account defined in Listing 3

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: exa-backend-admin-replicasets
```

```yaml
subjects:
- kind: ServiceAccount
  name: exa-backend-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: rs-admin
  apiGroup: rbac.authorization.k8s.io
```

Listing 9: Configuration file that binds the cluster role `rs-admin` defined in Listing 7 to the service account defined in Listing 3

Networking

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: exa-backend-policy
spec:
  podSelector:
    matchLabels:
      app: exa-backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access-exa: "true"
```

Listing 10: Configuration file that restricts network communication as explained in Section 3.3.3

Pod Security Policy

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
```

```yaml
  name: exa-psp-restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames:
      'docker/default,runtime/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames:
      'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:
      'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName:
      'runtime/default'
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  # Allow core volume types.
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    # Require the container to run without root privileges.
    rule: 'MustRunAsNonRoot'
  seLinux:
    # This policy assumes the nodes are using AppArmor
    # rather than SELinux.
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
```

```
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  readOnlyRootFilesystem: true
```

Listing 11: Configuration file that enforces a `PodSecurityPolicy` admission controller as explained in Section 3.3.2.

Appendix B: Acronyms

ABAC Attribute-based Access Control
CA Certificate Authority
CRL Certificate Revocation Lists
GCP Google Cloud Platform
GKE Google Kubernetes Engine
IaaS Infrastructure as a Service
IAM Identity & Access Management
KMS Key Management Service
OCSP Online Certificate Status Protocol
OIDC OpenID Connect
PKI Public Key Infrastructure
RBAC Role-based Access Control
SSO Single Sign-On
TPM Trusted Platform Module
VM virtual machine

Bibliography

[1] Puja Abbassi. *Securing the Base Infrastructure of a Kubernetes Cluster*. Nov. 2018. URL: https://blog.giantswarm.io/securing-the-base-infrastructure-of-a-kubernetes-cluster/ (accessed on 06/10/2019) (cit. on p. 8).

[2] Puja Abbassi. *Securing the Configuration of Kubernetes Cluster Components*. Dec. 2018. URL: https://itnext.io/securing-the-configuration-of-kubernetes-cluster-components-c9004a1a32b3 (accessed on 04/24/2019) (cit. on pp. 10, 14).

[3] Puja Abbassi. *Why Is Securing Kubernetes so Difficult?* Oct. 2018. URL: https://blog.giantswarm.io/why-is-securing-kubernetes-so-difficult/ (accessed on 04/24/2019) (cit. on p. 7).

[4] Mike Altarace and Daz Wilkin. *Kubernetes Namespaces: use cases and insights*. Aug. 2016. URL: https://kubernetes.io/blog/2016/08/kubernetes-namespaces-use-cases-insights/ (accessed on 06/07/2019) (cit. on p. 13).

[5] Roderick Bauer. *What's the Diff: VMs vs Containers*. June 2018. URL: https://www.backblaze.com/blog/vm-vs-containers/ (accessed on 07/18/2019) (cit. on p. 3).

[6] Mark Betz. *Understanding kubernetes networking*. Oct. 2017. URL: https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727 (accessed on 05/02/2019) (cit. on p. 19).

[7] *Hacking with Netcat part 2: Bind and reverse shells*. https://www.hackingtutorials.org/networking/hacking-netcat-part-2-bind-reverse-shells/. 2016. (Accessed on 05/31/2019) (cit. on p. 20).

[8] Michael Boelen. *Linux capabilities 101*. Nov. 2014. URL: https://linux-audit.com/linux-capabilities-101/ (accessed on 06/28/2019) (cit. on p. 22).

[9] Margaret Dawson. *Red Hat Global Customer Tech Outlook 2019*. Dec. 2018. URL: https://www.redhat.com/en/blog/red-hat-global-customer-tech-outlook-2019-automation-cloud-security-lead-funding-priorities?source=bloglisting (accessed on 08/11/2019) (cit. on p. 3).

[10] John D Day and Hubert Zimmermann. "The OSI reference model". In: *Proceedings of the IEEE* 71.12 (1983), pp. 1334–1340 (cit. on p. 19).

[11] Dan Goodin. *Tesla cloud resources are hacked to run cryptocurrency-mining malware*. Feb. 2018. URL: https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/ (accessed on 06/11/2019) (cit. on p. 12).

[12] Michael Hausenblas. *Non-privileged containers FTW!* URL: http://canihaznonprivilegedcontainers.info (accessed on 06/25/2019) (cit. on p. 21).

[13] Michael Hausenblas and Sébastien Goasguen. *Kubernetes Cookbook. Building Cloud Native Applications*. O'Reilly Media, Inc., 2018 (cit. on p. 16).

[14] Malte Isberner. *A Guide to Kubernetes Admission Controllers*. Mar. 2019. URL: https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/ (accessed on 05/24/2019) (cit. on p. 18).

[15] *Information technology – Trusted platform module library – Part 1: Architecture*. Standard. Geneva, CH: International Organization for Standardization, Aug. 2015 (cit. on p. 10).

[16] Greg Castle Jacob Simpson and CJ Cullen. *RBAC Support in Kubernetes*. Apr. 2017. URL: https://kubernetes.io/blog/2017/04/rbac-support-in-kubernetes/ (accessed on 04/26/2019) (cit. on p. 16).

[17] *Kubernetes API Documentation*. https://kubernetes.io/docs/concepts/overview/kubernetes-api/. 2019. (Accessed on 04/25/2019) (cit. on p. 13).

[18] *Kubernetes Documentation*. https://kubernetes.io/docs/. 2019. (Accessed on 04/25/2019) (cit. on pp. 4, 12–15, 18, 19, 22, 31).

[19] Carol McDonald.
Kubernetes, Kafka Event Sourcing Architecture Patterns and Use Case Examples.
May 2018. URL: https://mapr.com/blog/kubernetes-kafka-event-sourcing-architecture-patterns-and-use-case-examples/ (accessed on 07/18/2019) (cit. on p. 4).

[20] Jérôme Petazzoni.
If you run SSHD in your Docker containers, you're doing it wrong! June 2014.
URL: https://jpetazzo.github.io/2014/06/23/docker-ssh-considered-evil/ (accessed on 06/25/2019) (cit. on p. 21).

[21] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617.
RFC Editor, Sept. 2015 (cit. on p. 15).

[22] Liz Rice and Michael Hausenblas. *Kubernetes Security*. First Edition.
O'Reilly Media, Inc., 2018. ISBN: 978-1-492-04600-4
(cit. on pp. 7, 10, 12, 18).

[23] Jerome H. Saltzer.
"Protection and the Control of Information Sharing in Multics".
In: *Commun. ACM* 17.7 (July 1974), pp. 388–402. ISSN: 0001-0782.
DOI: 10.1145/361011.361067.
URL: http://doi.acm.org/10.1145/361011.361067 (cit. on p. 17).

[24] *SECCOMP(2) Linux Programmer's Manual*. Mar. 2019 (cit. on p. 22).

[25] Alex Tcherniakhovski and Andrew Lytvynov.
Securing Kubernetes with Trusted Platform Module (TPM). May 2019.
URL: https://www.youtube.com/watch?v=_kxmkI8Kc8Y (accessed on 06/10/2019) (cit. on p. 10).

[26] Alexander Urcioli.
Analysis of a Kubernetes hack - Backdooring through kubelet. Mar. 2018. URL: https://medium.com/handy-tech/analysis-of-a-kubernetes-hack-backdooring-through-kubelet-823be5c3d67c (accessed on 06/11/2019) (cit. on p. 11).

[27] Simon Woodside.
Defence in Depth. The medieval castle approach to internet security. June 2016.
URL: https://medium.com/@sbwoodside/defence-in-depth-the-medieval-castle-approach-to-internet-security-6c8225dec294 (accessed on 07/18/2019) (cit. on pp. 2, 20).

[28] Eric Yuan and Jin Tong.
"Attributed based access control (ABAC) for web services".
In: *IEEE International Conference on Web Services (ICWS'05)*. IEEE. 2005
(cit. on p. 16).

[29] Qi Zhang et al. "A comparative study of containers and virtual machines in big data environment". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 178–185 (cit. on p. 3).