CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Automatic enforcement of container security guidelines through policy as code

Master's thesis in Computer science and engineering

Marcus Rönnbäck
Fredrik Åberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Automatic enforcement of container security guidelines through policy as code

Marcus Rönnbäck
Fredrik Åberg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Automatic enforcement of container security guidelines through policy as code

Marcus Rönnbäck
Fredrik Åberg

Supervisor: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering.
Advisor: Johan Tordsson, Elastisys
Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering.

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Marcus Rönnbäck, Fredrik Åberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The increase in Kubernetes usage and container usage in general brings new challenges regarding security. Recent surveys show that container system misconfigurations are the most common cause of concern faults and error handled by system administrators. Common security guidelines exist that can help with ensuring that configurations are correct, but they typically involve manual policy enforcement which can be tedious and time consuming. This process can be automated by employing a "policy-as-code" system which checks and evaluates the validity of given configurations. It is not clear as to what extent it is possible to enforce common security guidelines through policy-as-code. In this thesis, the questions we aim to answer are: To what extent are common security guidelines enforceable through policy-as-code? Does it have any limitations or cases that cannot be covered? Does the implementation of these policies affect performance? Are there any concrete known vulnerabilities that are mitigated by these policies? This is done through empirical studies and evaluations of security guidelines and investigations as to what extent they are enforceable. Our findings using open-source Kubernetes security software is that the overall number of common security guidelines that are enforceable through policy-as-code systems are 33 out of 55, which is 60%. The non-enforceable guidelines depend on external factors such as organizational structure and user permissions, which are hard to implement in a policy-as-code system with current technologies.

Keywords: kubernetes, policy-as-code, policy, open policy agent, rego, opa.

Acknowledgements

We would like to thank our supervisors Ahmed and Johan for supporting us during our thesis work. We would also like to thank our families for supporting us through our education.

Marcus Rönnbäck & Fredrik Åberg, Gothenburg, December 2022

Contents

List of Figures

List of Tables

1 Introduction

In recent years Kubernetes and container technologies in general have gained great popularity in production server usage. A survey from the Cloud Native Computing Foundation (CNCF) [1] answered by engineers at software and technology organizations shows that Kubernetes usage among them has increased to 91% from just 26% in their first survey from 2016. In addition, 32% of respondents answered that one of the primary challenges running container technology in production has been security, the third most common challenge after complexity and culture changes. On top of that, private cloud usage has increased from 45% in 2019 to 52% in the global cloud community. This increase in popularity motivates development in providing more streamlined security solutions for container setups.

A survey performed by RedHat was released on May 5th, 2022 highlighting the current state of security in Kubernetes [2]. More than 300 DevOps and security professionals were queried about their usage of kubernetes and their strategies surrounding containerized software. The survey revealed that 93% of the respondents had experienced at least one security incident in the period January to December 2021. The reasons behind the incidents reported ranged from lack of knowledge to being unable to keep up with development teams. The majority of incidents, 53% were due to misconfigurations. 46% of the respondents were most worried about misconfigurations and accidental exposures, whilst only 16% were most worried about being attacked. World Economic Forum reports that 95% of cybersecurity issues can be traced to human error, and 43% of breaches originate from inside threats, both intentional and unintentional such as misconfigurations [3].

Policies can be employed to ensure the correctness of configurations, for example related to security. However, manual policy enforcement can be tedious and time-consuming, to solve this, some organizations have chosen to employ what is known as *policy-as-code*, namely the concept of using code to check the correctness of configurations. These policies do however need to be designed, and to aid in this, security agencies have released best practices regarding securing kubernetes that can be followed. Some of these guidelines deal with Kubernetes directly and some contain more general security and organizational guidelines. A few examples of these are:

- Center for Internet Security - CIS Benchmarks - Securing Kubernetes [4]
- NSA/CISA Kubernetes hardening guidance [5]
- EU - GDPR Section 2, Article 32 - Security of processing personal data[6]
- International Organization for Standardization (ISO) 27001:2013 [7]

1.1 Problem description

With the survey from CNCF showing an increase in Kubernetes usage to 92% in 2020 from 26% in 2016 [1], and the majority of incidents (53%) in Kubernetes environments being due to misconfigurations [3] that could have been minimized if common security guidelines were followed, there is a need to investigate to what extent these common security guidelines can be automatically enforced through policy-as-code. Manual policy enforcement can be tedious and time-consuming. This can be alleviated by automating the policy enforcement using policy-as-code with policies from common security guidelines. The problem we tackle in this thesis is to identify to what extent common security guidelines [4, 5] are enforceable through policy-as-code. To do this, we empirically study and evaluate the security guidelines that we briefly mentioned above, and investigate to what extent it is possible to enforce them using policy-as-code by developing a prototype. For each particular control in some guideline we need to determine if policy-as-code can be utilized to enforce it, if a subset of the policy can be enforced, or if it is inapplicable.

1.1.1 Specification of issue under investigation

There are three different overarching levels for enforcing these policies that we have identified and that we need to consider when tackling this question. These three categories correspond to where in the lifecycle a policy is applied. Each of these levels require different tools to implement the guidelines.

1. Node configuration covers host system configuration and the configurations for worker nodes and the control plane components, (See Section 3.2). Policies in this category are applied before Kubernetes is allowed to start.
2. Resource deployment covers checks for when a kubernetes resource is deployed. (See Section 3.2.2).
3. The runtime is a broad category and includes for example network traffic and system calls. This section covers policies that intercept system calls and events.

While some of these tasks require kernel access to enforce there are some, for example enforcement of internal network traffic rules that can be implemented without kernel access. The key functionality is to provide a checker that enforces the required security policies for Kubernetes clusters. The produced tools are tested on a Kubernetes cluster that is deployed on a setup according to best practices.

1.2 Limitations

We have decided to focus our work on the *NSA/CISA Kubernetes hardening guidance* and the *CIS Benchmarks* as these are focused on kubernetes. We decide to not include *ISO27001* and the *EU-GDPR*. The *ISO27001* is an organizational standard and is from investigation hard to represent in code as large parts of it has to do with the structure of the organization[7]. The same applies for *EU - GDPR Section 2, Article 32*[6]. This involves a lot of legal terms and issues that we consider are out of scope for an investigation regarding policy-as-code. Following is the full list

of limitations:

1. Some guidelines are quite broad and touch aspects beyond the scope of kubernetes, where a solution can not be made in a generic way. Not all environments use the same methods of specifying configuration settings and there might be different mechanisms used to configure resources. We aim to develop policies towards the standard kubernetes resources, but also make note of situations like the one described above whenever one is identified.
2. We are targeting self-deployed Kubernetes, and not the services managed by for example Amazon or Google, as these may have different tools and standard configurations.
3. We are only targeting Kubernetes clusters running on GNU/Linux operating systems.
4. Policy enforcement will not be implemented in such a way that it can be guaranteed to work retroactively on an existing cluster. Any policy is enforced under the assumption that it will not necessarily cover already deployed configurations.

1.3 Our contributions

Our investigation is to the best of our knowledge the first comprehensive policy-as-code study and implementation of the NSA/CISA guidelines through policy-as-code. It is also the first publicly available study of to what extent the CIS Benchmarks can be enforced through policy-as-code. Each guideline is qualitatively investigated and the results such as limitations and effect on performance is discussed.

The experiments were conducted on a Kubernetes cluster deployed by us. Each guideline is investigated as to if it can be implemented through policy-as-code or not. Each policy's execution time is measured as to see if there is any overhead in the form of latency created by the implementation. The implementation of the proof of concept policies were further experimented on by attempting to deploy resources that violate the policies and by running unit-tests that check the correctness of the policies.

The question we aim to answer is, as presented in Section 1.1: To what extent common security guidelines are enforceable through policy-as-code? The results show that 33 out of 55 controls are implementable. A total of 14 out of 25 NSA controls are enforceable and 19 out of 30 CIS controls are enforceable. This results in an overall enforcement percentage of 60%.

The results show that most implementable policies incur no major latency overhead for either resource deployments or node deployment policies. The one outlier is NSA-1.3 (see Result 6.4.4) which can introduce major latency of varying length based on the input. All resource deployment policies combined had a benchmark result of about 6 milliseconds. The node configuration benchmark result was about 20 milliseconds. These are acceptable numbers for the intended workload.

This investigation will bring knowledge on to what extent common security guidelines are enforceable though policy-as-code systems, what the limitations are, and what the effect on performance is. This in turn will allow the security community

to make well-founded decisions as to what can be further investigated in the future in order to advance in the field of automatic policy enforcement.

1.4 Risk analysis and ethical considerations

Due to the nature of this project only targeting software, there are minimal considerations that can be taken within this context. Any potential penetration testing will only be performed on systems that we own.

2
Related works

In this chapter we present some previous research into the field of container security. As explained in Section 1, 46% of respondents to the survey performed by RedHat answered that misconfigurations and accidental exposures were their top source of worry. Research into the security of Kubernetes has been conducted by numerous people and several vulnerabilities have been found. These vulnerabilities can originate from bugs and/or oversights in the design phase of the underlying software, but some of them exist only due to misconfigurations of the kubernetes installation and the deployment of the software. There exists general security recommendations that can help administrators secure their clusters against these common misconfigurations.

2.1 Threats

The authors of [8] describe ways of attacking targets in third party cloud environments such as Amazon Web Services by placing a virtual machine (VM) on the same instance. They further explain that since the two VMs share the hardware they are able to leak data through the hardware cache. When sharing the physical hardware it is also possible to perform a denial of service attack on the other machine for instance by keeping the disk or CPU busy. The authors of [9] experiment in the same direction by having many hosts in the same physical location consume more power than available and therefore forcing the shutdown of certain machines. The authors of [10] describe a scenario where a user with too many privileges inside a kubernetes environment is able to starve the other containers of computational resources. This is due to containers not having a limit on CPU or memory by default. Co-residency attacks have been utilized to access data from companies such as NordVPN and Tesla, as recently as 2018 [11]. In both cases the cause was a misconfiguration, and the solution was mentioned to be the implementation of configuration monitoring:

> *"This involves deploying tools that can automatically discover resources as soon as they are created, determining the applications running on the resource, and applying appropriate policies based on the resource or application type. Configuration monitoring could have helped Tesla immediately identify that there was an unprotected Kubernetes console exposing their environment." – RedLock CSI Team* [12]

NordVPN was also the victim of a misconfiguration where the third party hosting provider added an insecure remote management account without the knowledge of NordVPN. This allowed for an attacker to gain access to an already expired

Transport Layer Security (TLS) key that could be used in a targeted Man In The Middle (MITM) attack[11].

2.2 Networking in Kubernetes

Network security in Kubernetes has been previously investigated by the authors of *Network policies in kubernetes: Performance evaluation and security analysis* [13]. The authors list challenges that comes with network security in Kubernetes as well as evaluate the performance impact that network policies has when using network policies together with different Container Network Interfaces. They conclude that there is no significant performance overhead when employing network policies as a network isolation security solution.

The authors of [14] describe what network-security related issues exist in Kubernetes. The Container Network Interface plugin in Kubernetes runs with elevated privileges, and as such can be a target for attackers. Plugins that make use of the Linux bridge may be susceptible to MITM attacks such as ARP and DNS spoofing, where a compromised pod may pose as another pod and intercept network traffic.

2.3 Virtualization

The authors of [15] claim that security is a major challenge when running services in virtual environments. Container virtualization as opposed to hypervisor-based virtualization is considered to be less secure due to containers sharing the host kernel. Even though it is considered less secure, it has gained popularity in recent years due to its ease of use and minimal performance overhead.

[15] lists several of these security challenges that comes with container virtualization, such as Linux kernel exploits, container leaks, denial of service, and fake system calls. They also list some internal security features that can help secure against certain vulnerabilities such as Linux Namespaces[1] and Linux Security Modules such as SELinux and AppArmor[16].

To help combat the security challenges that comes with container virtualization the authors of [15] propose a new security layer for containers where all containers must request access to resources, and where the access is controlled by access control policies.

2.4 Policies

There are existing policy libraries containing the CIS Benchmark, notably the Styra DAS[2] Compliance Pack which contains policies for CIS Benchmark. Which guidelines in the CIS Benchmark that this compliance pack covers is not publicly available. There is also no study published by Styra as to what guidelines are enforceable or not. Our investigation results in a complete study of the NSA/CISA hardening

[1] https://man7.org/linux/man-pages/man7/namespaces.7.html
[2] https://www.styra.com/styra-das/

guidance and Section 5 of the CIS Benchmark, with a publicly available investigation as to what guidelines are enforceable or not. There are no existing policy packs for the NSA/CISA hardening guidance, resulting in this one being the first.

Our investigation is to the best of our knowledge the first comprehensive policy-as-code study and implementation of the NSA/CISA guidelines through policy-as-code. It is also the first publicly available study of to what extent the CIS Benchmarks can be enforced through policy-as-code.

Policy enforcement in container environments has previously been investigated by the authors of [17], who show a method of using a policy agent in a container environment to construct a system adhering to zero-trust(cite) principles. The secure architecture implemented by [17] prevents data-leaks and protects data at rest and in transit between non-trusted agents. This architecture is enforced using policy sidecars on the components. The authors present a proof-of-concept that during experimentation showed promising results when scaling with increased application complexity.

3
Background

This chapter introduces how policy-as-code can be used to secure a cluster. The architecture of Kubernetes is explained as well as what a container is. Next, the two sets of policy guidelines are presented and the controls are explained briefly.

3.1 Containers

Containers are a virtualization method where applications are run in (their own) userspace instance, making use of the host operating system's kernel. This means that services that require the same dependency but a different version can run on the same system as the dependencies are packaged together with the application code. The Open Container Initiative was created by Docker and CoreOS in 2015[18]. Together with other leading figures in the container industry and the Linux Foundation this self-governing structure set forth to create open standards for the future of container technology. Google Cloud explains containers in the following way:

> *"Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services."* – Google Cloud[19]

The authors of [20] argue that the principle of container isolation in containerization is less secure than traditional VMs due to all the containers sharing the host-kernel. Containerization is however more resource efficient as each container only contains what is needed, no extra overhead in the form of a OS for each application (which could be the case for a VM).

3.2 Kubernetes

Google released Kubernetes 1.0[1] in July 2015 under the Apache License 2.0 together with the Linux Foundation which also coincided with the creation of the CNCF[2]. The roots of Kubernetes can be traced to the Borg system used internally by Google since the early 2000s. This system was kept secret for a long time but in 2015 Google released their paper *Large-scale cluster management at Google with Borg* describing their cluster manager *Borg*[21]. Kubernetes takes its core components from Borg,

[1] https://github.com/kubernetes/kubernetes/releases/tag/v1.0.0

[2] https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/

with many developers from Google still contributing to the open source project. The cloud native community work together to bring new features to the project.

Kubernetes is in essence a set of tools, most of which are userspace components that work in tandem to orchestrate containers and keep track of cluster state. Kubernetes can perform rolling updates of containerized software, restart crashed containers, and create replicas for load balancing purposes among many other features.

The control plane components is comprised of the components that involve cluster-wide state and configuration:

- *kube-apiserver* - The frontend to the control plane, exposes the kubernetes API-server.
- *etcd*[22] - Key value store, the state of all cluster resources is stored in this distributed key-value store. It acts as the single point of truth for resource information across the cluster.
- *kube-scheduler* - Assigns newly created pods and assigns them to a node if none is specified.
- *kube-controller-manager* - Overarching name for all controller processes such as Node controller or Job controller.

The *kubelet* and *kube-proxy* are the core node components, the former managing local containers and the latter handling the network traffic including the load-balancing to and from services running on the cluster. In the following sections the default installation refers to the one obtained by running `kubeadm init`.

3.2.1 Kubernetes Architecture

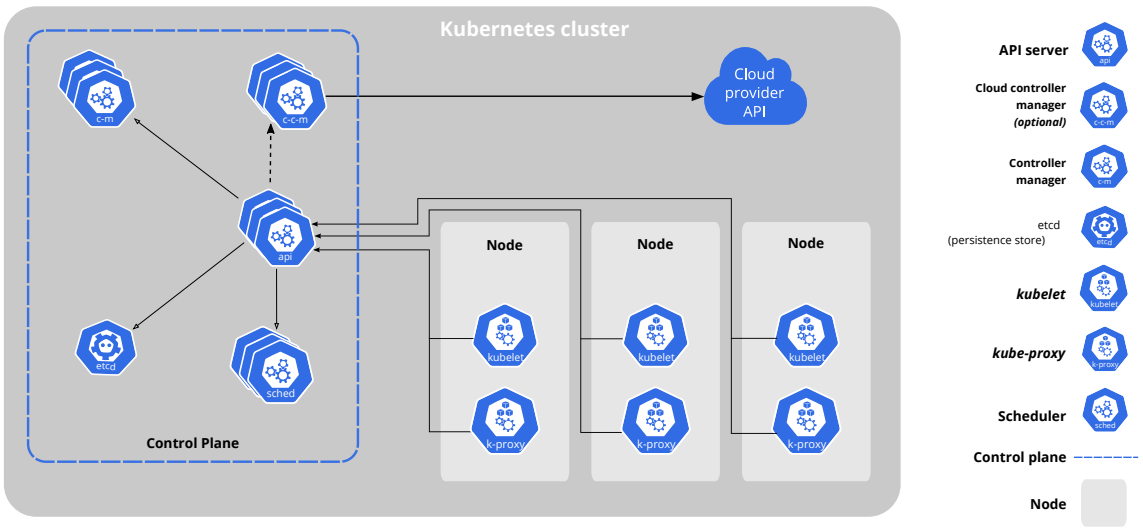

Figure 3.1: Kubernetes Cluster Architecture. Image from Kubernetes Documentation. Licensed under CC BY 4.0

The control plane components on the left side of Figure 3.1 make up the *control plane node*. The three nodes on the right side are the clusters *worker nodes* and

they communicate with the control plane node via the *kube-apiserver*.

Node

There are two kinds of nodes in a kubernetes cluster, control plane nodes and worker nodes. The distinction lies in that the control plane node controls a set of worker nodes. . The control plane node handles all communication with the cluster, and schedules workloads on worker nodes present in the same cluster. A control plane node can also be a worker node, for example when running a single node cluster.

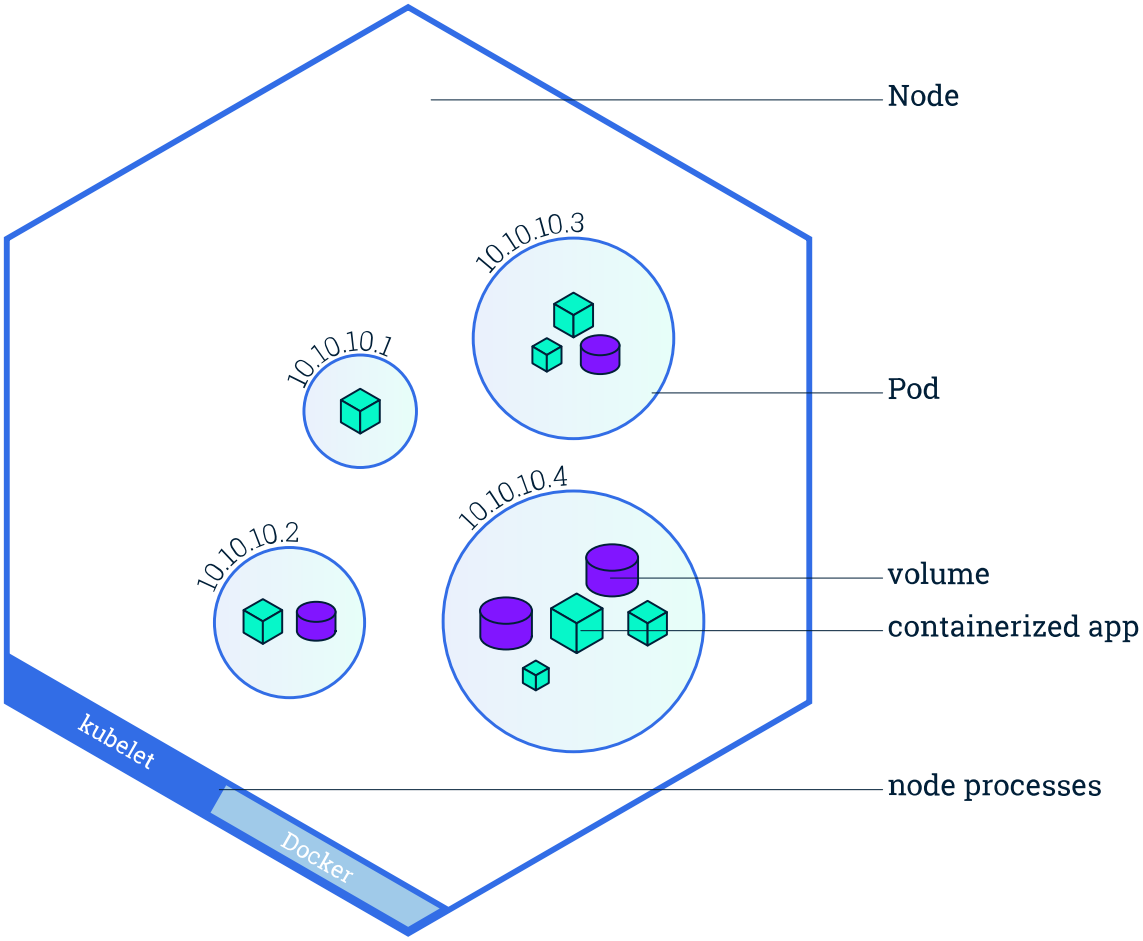

Figure 3.2: Kubernetes Node Overview. Image from Kubernetes Documentation. Licensed under CC BY 4.0

The worker nodes run workloads scheduled to run on them by the control plane. These workloads are managed by the *kubelet* application present on all worker nodes. In order to run containers, a container runtime is necessary. Kubernetes supports a variety of runtimes, as long as they conform to the Kubernetes Container Runtime Interface (CRI). Most commonly used is Docker (87%), containerd(36%), or cri-o(17%) [2].

Pod

A pod is a collection of containers that share resources such as storage and network, see Figure 3.3 for a visualisation. All containers in a single pod are scheduled on the same node as seen in Figure 3.2. A Pod can be seen as a way of encapsulating all services that compose a single application, for example a pod with one container running the application code and one container handling the database together with a volume to store some persistent data.

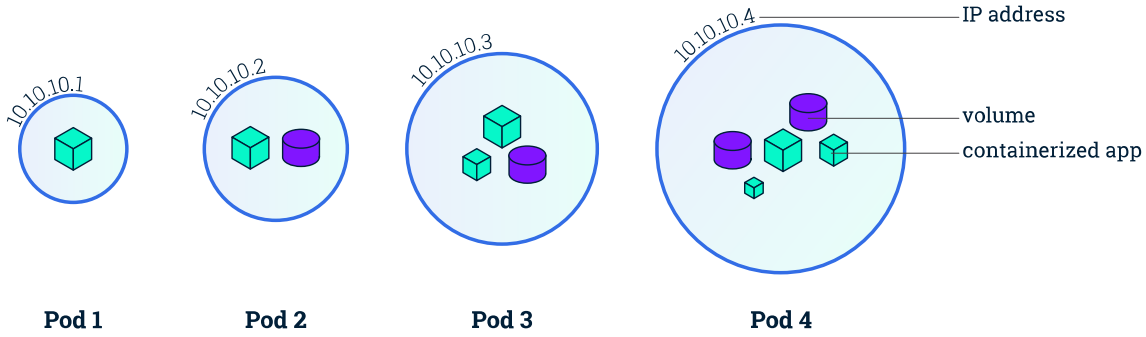

Figure 3.3: Kubernetes Pod Overview. Image from Kubernetes Documentation. Licensed under CC BY 4.0

3.2.2 Kubernetes Resources

Every managed entity in kubernetes is a resource. There are many core resources, but two types of resources that are most instrumental in the architecture - Nodes and Pods. Nodes are the individual running instances of kubernetes, and pods hold one or more containers and describe an individual instance of an application. Every instance of a Resource is stored and managed in the etcd instance as json-formatted manifests. Resources can be created in many different ways, one of the more common ones being to write a YAML-formatted Resource specification.

Workloads in the sense of kubernetes can be one of the following:

- *Jobs* are workloads that just specify a number of executions. Workloads that run at a specified interval are called *CronJobs*.
- *DaemonSets* are used to run services that should be present on each node in the cluster. Nodes that match the description of the *DaemonSet* automatically get the resources specified scheduled.
- *Deployments*, *ReplicaSets*, or *StatefulSets* can be used for regular deployments.
  - *Deployments* are high-level descriptions of a desired state of a group of pods.
  - *ReplicaSets* maintain a specified number of replicas of a Pod and are typically only created by Deployments.
  - *StatefulSets* are used for pods that run stateful applications. One example of this is databases. pods in *StatefulSets* receive a unique identifier (mysql-0, mysql-1, mysql-n...) and in the event of a pod-failure the new pod takes over the old pods name instead of becoming a "new" pod.

3.2.3 Namespace

Namespaces are used to control groups of resources. Namespaces on their own do not add any security, but they allow for easier administration. Namespaces in conjunction with Resource Policies and Role Based Access Control (RBAC) is enough to guarantee isolation between tenants in a cluster. By default there are only three namespaces, namely *kube-system*, *kube-public*, and *default*. The kubernetes control plane is isolated to *kube-system*, but it is possible to create new deployments inside this namespace. *kube-public* is readable by all users, but is also able to accept new deployments. The *default* namespace is used for resources that do not specify a namespace.

3.2.4 Resource Policies

By default, there is no limit to how many resources one pod is allowed to consume on the node. This leads to the possibility of a pod starving all other pods present on the node of processing resources, CPU and memory. To counter this issue there are network and resource policies.

Network policies

By default, all pods in a kubernetes cluster are able to communicate with all other pods on the node due to there being no default limitations with respect to network traffic. In the case of a compromised pod the "attacker" is able to communicate with services not meant for external access. In order to use network policies the clusters Container Network Interface (CNI) needs to support the NetworkPolicy API. The network policies are applied on a namespace level but can be set up so that it only applies to certain pods in the namespace. The network policy handles both *ingress* and *egress*. *Ingress* is for incoming connections and *egress* is for outgoing connections. The network policies specify which traffic is allowed or disallowed based primarily on references to Resource selectors, IP-addresses and ports.

Resource quotas and limit ranges

Resource Quotas limit the total computing resources of all combined workloads in a namespace. The resources in question are memory, CPU-cores, and any extended resources added to the system. The usage of resource quotas requires all new deployments to specify a resource request and resource limit in the workload deployment. To automate this the administrator can create a LimitRange that will have default values that apply to all workloads that have none specified

3.3 NSA/CISA Kubernetes hardening guidance

Kubernetes, when set up using default methods such as *kubeadm* on default OS installations is configured with many security settings disabled. All containers are for example able to run as the root user by default. If a pod running as root is compromised the adversary will have root privileges on the host system. There are

many of these default settings that can and should be changed before deploying an application on a kubernetes cluster. The National Security Agency (NSA) released a kubernetes hardening guidance on August 3rd, 2021, and further updated it on March 15th, 2022 [5]. This hardening guidance written by NSA and the Cybersecurity and Infrastructure Security Agency (CISA) focuses on only kubernetes. The guidance was released to bring forward the challenges that comes with setting up a kubernetes cluster, and most importantly how to secure it and avoid misconfigurations.

The following list contains the guidelines. They are taken directly from the NSA hardening guidance[5]. A small explanation is added to each guideline.

3.3.1 NSA Guidelines

1. **Kubernetes Pod security**
   - 1.1. ***Non-root* containers and *rootless* container engines**
     Make sure containers are run as non-root users by either specifying a user or by using a rootless container engine.
   - 1.2. **Immutable container file systems**
     For containers that do not need access to the root filesystem, set the filesystem as read-only.
   - 1.3. **Building secure container images**
     Only allow containers from trusted container registries and scan all container images for potential vulnerabilities.
   - 1.4. **Pod security enforcement**
     Use Pod Security Admission or make sure pods are *safe* before deployment.
   - 1.5. **Protecting Pod service account tokens**
     Disable the automatic mounting of the default service account token.
   - 1.6. **Hardening container environments**
     - Hypervisor-backed containerization
     - Kernel-based solutions
     - Application sandboxes
2. **Network separation and hardening**
   - 2.1. **Namespaces**
     Use namespaces to further isolate different applications in conjunction with resource policies.
   - 2.2. **Network policies**
     Deny all ingress and egress traffic by default, only allow legitimate traffic.
   - 2.3. **Resource policies**
     Set resource limits to namespaces and pods to ensure that no single pod can starve the system of resources.
   - 2.4. **Control plane hardening**
     Harden the control plane components by using the following:
     - 2.4.1. TLS encryption.
     - 2.4.2. Strong authentication.
     - 2.4.3. Disable access to unnecessary and untrusted networks.

2.4.4. Use RBAC to restrict access to resources.
2.4.5. Secure the etcd datastore.
2.4.6. Kubeconfig File permissions.
2.5. **Worker node segmentation**
Separate the worker nodes from other network segments.
2.6. **Encryption**
Encrypt all internal traffic using TLS.
2.7. **Secrets**
Encrypt all secrets. Stored as unencrypted base64-string by default.
2.8. **Protecting sensitive cloud infrastructure**
Out of scope.
3. **Authentication and authorization**
3.1. **Authentication**
Disable anonymous authentication and use a strong authentication.
3.2. **Role-based access control**
Use RBAC to control access to cluster resources.
4. **Audit Logging and Threat Detection**
4.1. **Logging**
4.1.1. Kubernetes native audit logging configuration
Enable auditing by creating an audit policy
4.1.2. Worker node and container logging
Make use of remote logging
4.1.3. Seccomp: audit mode
Log system calls using seccomp.
4.1.4. Syslog
Forward logs to syslog server.
4.1.5. SIEM platforms
Inregrate kubernetes logs into SIEM platforms.
4.1.6. Service meshes
Consider using service meshes build in logging capabilities.
4.1.7. Fault tolerance
Have fault tolerant setups for logging. For example, overwrite oldest log file if storage is low.
4.2. **Threat Detection**
4.2.1. Alerting
Consider using a monitoring tool to alert on eventual problems.
4.3. **Tools**
Consider employing different tools such as Intrusion Detection Systems.

3.4 CIS Benchmark

The Center for Internet Security (CIS) is a nonprofit organization that publishes best practice guidelines against cyber threats[4, 23]. Similarly to the NSA Guidelines, the guidelines published by CIS aim to help Kubernetes administrators harden their clusters. The CIS Benchmark is written by the *Consensus Community* that consists

of experts in IT security who use their aggregate knowledge to help others. The recommendations in CIS are tailored to kubernetes, and have sections targeting different aspects of Kubernetes [4]. These *best practices* target Kubernetes clusters that are self-managed, and not those managed by Amazon, Google, or other cloud providers. The ones managed by cloud providers may have other tools and standard configurations.

The CIS Benchmark is split into several sections. Section 1 targets the control plane components such as the Master node configuration files, api-server, controller manager, and scheduler. Section 2 describes etcd, Section 3 focuses on the control plane configurations for example authentication, authorization, and logging. Sections 4 describes the worker nodes and their configuration files, and finally Section 5 targets policies. For our investigation we only look at the policies in Section 5, and if necessary refer to the other sections for information. The following list is from Section 5 of Version 1.23 of the CIS Benchmark for Kubernetes[4].

3.4.1 CIS Benchmark guidelines

5. **Policies**
   - 5.1. **RBAC and Service Accounts**
     - 5.1.1. Ensure that the cluster-admin role is only used where required.
     - 5.1.2. Minimize access to secrets.
     - 5.1.3. Minimize wildcard use in Roles and ClusterRoles.
     - 5.1.4. Minimize access to create pods.
     - 5.1.5. Ensure that the default service accounts are not actively used.
     - 5.1.6. Ensure that Service Account Tokens are only mounted when necessary.
     - 5.1.7. Avoid usage of system:masters group.
     - 5.1.8. Limit use of the Bind, Impersonate, and Escalate permissions in the Kubernetes cluster.
   - 5.2. **Pod Security Policies**
     - 5.2.1. Ensure that the cluster has at least one active policy control mechanism in place.
     - 5.2.2. Minimize the admission of privileged containers.
     - 5.2.3. Minimize the admission of containers wishing to share the host process ID namespace.
     - 5.2.4. Minimize the admission of containers wishing to share the host IPC namespace.
     - 5.2.5. Minimize the admission of containers wishing to share the host network namespace.
     - 5.2.6. Minimize the admission of containers with `allowPrivilegeEscalation`.
     - 5.2.7. Minimize the admission of root containers.
     - 5.2.8. Minimize the admission of containers with `NET_RAW` capability.
     - 5.2.9. Minimize the admission of containers with added capabilities.
     - 5.2.10. Minimize the admission of containers with capabilities assigned.
     - 5.2.11. Minimize the admission of Windows HostProcess Containers.

5.2.12. Minimize the admission of HostPath volumes.
5.2.13. Minimise the admission of containers which use HostPorts.
5.3. **Network Policies and CNI**
5.3.1. Ensure that the CNI in use supports Network Policies.
5.3.2. Ensure that all Namespaces have Network Policies defined.
5.4. **Secrets Management**
5.4.1. Prefer using secrets as files over secrets as environment variables.
5.4.2. Consider external secret storage.
5.5. **Extensible Admission Control**
5.5.1. Configure Image Provenance using ImagePolicyWebhook admission controller.
5.6. **General Policies**
5.6.1. Create administrative boundaries between resources using namespaces.
5.6.2. Ensure that the seccomp profile is set to docker/default in your pod definitions.
5.6.3. Apply Security Context to your Pods and Containers.
5.6.4. The default namespace should not be used.

3.4.2 CIS/NSA overlap

There are guidelines in CIS that overlap with the ones in NSA presented earlier in Section 3.3.1. The policies that we have identified that are in CIS but not in NSA are:

- 5.1.1 Ensure that the cluster-admin role is only used where required.
- 5.1.2 Minimize access to secrets.
- 5.1.3 Minimize wildcard use in Roles and ClusterRoles.
- 5.1.4 Minimize access to create pods.
- 5.1.7 Avoid usage of system:masters group.
- 5.1.8 Limit use of the Bind, Impersonate, and Escalate permissions in the Kubernetes cluster.
- 5.2.1 Ensure that the cluster has at least one active policy control mechanism in place.
- 5.2.8 Minimize the admission of containers with NET_RAW capability.
- 5.2.9 Minimize the admission of containers with added capabilities.
- 5.2.10 Minimize the admission of containers with capabilities assigned.
- 5.2.11 Minimize the admission of Windows HostProcess Containers.
- 5.3.1 Ensure that the CNI in use supports Network Policies.
- 5.4.1 Prefer using secrets as files over secrets as environment variables.
- 5.4.2 Consider external secret storage.
- 5.5.1 Configure Image Provenance using ImagePolicyWebhook admission controller.

3.5 Policy-as-code

The term policy-as-code refers to the implementation of automatic enforcement of policies through code. A policy is a rule or condition that needs to meet a specific criteria in order to pass for example a security check.

The benefits of using policy-as-code is the ability to automate policy checks, and minimize potential mistakes that a human could make. By having the the policies automated from a central location and any updates made to them instantly go into effect for the entire cluster.

This implementation of policy-as-code allows for the usage of for example *git* as version control. Any changes to policies can be traced and new functions can be rolled back and traced in case of misconfigurations.

3.6 Open Policy Agent

Some of the controls presented in Section 3.3-3.4 contain policies that can be accurately expressed in policy-as-code systems. The only *graduated*[3] (meaning that it is considered stable and usable in production) CNCF policy agent project is Open Policy Agent (OPA) [24] and the company behind it, Styra, has developed libraries for implementing a subset of Payment Card Industry (PCI) [25] policies. In addition to the PCI-OPA link library Styra claims to have similar libraries for MITRE ATT&CK[4], although the coverage of these are, as far as we can tell, not publicly available.

[3] https://github.com/cncf/toc/blob/bc5df76a05b86152b55cbcf9e577d50e3491e365/process/graduation_criteria.md

[4] https://attack.mitre.org/

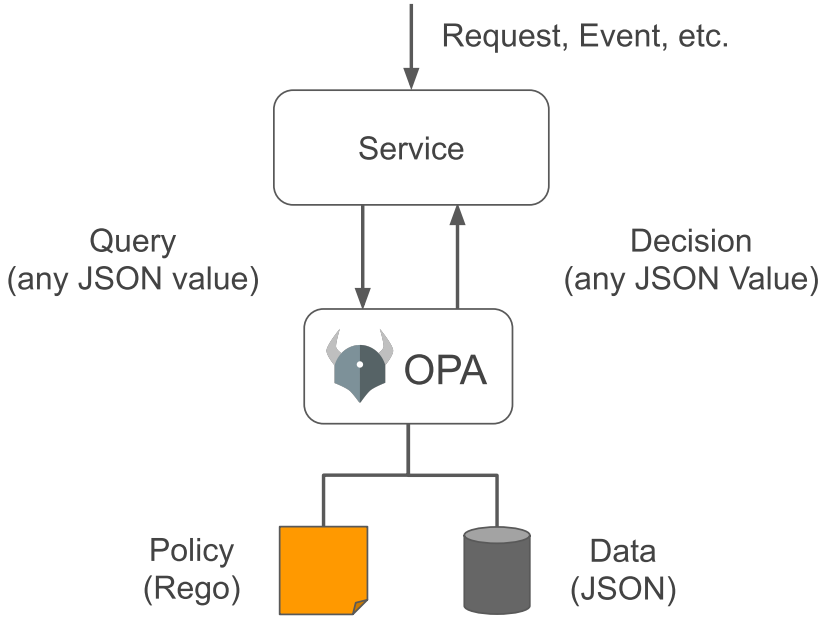

Figure 3.4: Open Policy Agent. Image from OPA Documentation. Licensed under Apache 2.0

Open Policy Agent [26] works by responding to events from services, and then evaluating these queries according to the policies. An example of a request to OPA can be seen in Figure 3.4. In the context of kubernetes, an example of this would be the kubernetes service requesting to deploy a new pod. OPA would then receive a request from kubernetes, which is evaluated against the current applied rules. OPA responds accordingly with an allowed or denied admission which kubernetes adheres to in its decision to execute the deployment.

4
Methods

This section covers the procedure of how the guidelines were investigated and how the testing and evaluation was performed.

4.1 Procedure

4.1.1 Investigation

In order to implement policy-as-code, a kubernetes cluster is needed in order to develop, test, and validate the policies. The only officially supported tool to install kubernetes is *kubeadm*, so that was chosen.

The hardening guidance from NSA is divided into several part, each category targeting different aspects of the kubernetes environment. Each item in a category can be seen as a single control. From the list of guidelines seen in Section 3.3.1 one can see that focusing on one guideline at a time is a suitable way forward.

The first step is to look at each guideline on its own, and classify it as either *cluster setup*, *resource deployment*, or *runtime*. By doing this we are able to more efficiently investigate what tools are needed. In this stage, guidelines that were out of scope would be found, and marked as such. After all guidelines has been classified, a subset of the guidelines is further investigated and development is started.

Guideline overlap

Due to the big overlap between NSA guidelines and CIS Benchmark, the CIS Benchmark is used to provide additional context in situations where the NSA guidelines lack details. The policies in CIS that does not have a counterpart in NSA is investigated in the same way as the other policies.

4.1.2 Development

Implementation is done using OPA[26] with *rego*[1] and *python*. Part of the implementation is to design both positive and negative tests to be used in the next stage. This process is repeated multiple times throughout the project until all or most of the policies have been covered. For policies that end up being considered unimplementable due to many different possible factors, are clearly marked as such with some documentation detailing that status.

[1] https://www.openpolicyagent.org/docs/v0.36.1/policy-language/#what-is-rego

For policies involving configurations and resource deployment OPA is always used. OPA is set up with a `ValidatingWebhook`[2] meaning all new deployments must be validated by OPA. An admission review is received by OPA when the deployment matches the criterias for the webhook to require an admission from OPA. This admission review contains the details about the deployment, and admission rules can be written as to follow the NSA and CIS guidelines.

Enforcement is obtained by having rules that either `allow` or `deny` deployment based on values in the resource deployment manifest. A deny is achieved if the body of a rule evaluates to true. If there are multiple lines, all lines are ANDed together to form the value, this means that for a rule to evaluate to true all lines inside must evaluate to true. An example rule that denies deployment if the pod name is "deny-me" can be seen in Figure 4.1.

```
1  deny[msg] {
2    input.request.object.metadata.name == "deny-me"
3    msg = sprintf("Pod (\%v) name matches "deny-me".",
     ↪  [input.request.object.metadata.name])
4  }
```

Figure 4.1: Example rego rule to deny if the name of the pod is "deny-me".

For policies that target the configuration of Kubernetes on each node, a small script is attached to the systemd service that typically executes kubelet. By specifying that kubelet is not allowed to run unless this script exits with error code 0, i.e "OK", enforcement is achieved.

[2] https://v1-23.docs.kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#validatingadmissionwebhook

5
Evaluation

The question we aim to answer is: To what extent are common security guidelines enforceable through policy-as-code. The main evaluations will be on several aspects. Individual policies will be evaluated with the following questions:

1. **Is it implementable? If not, why?**
   Including methods that may require additional dependencies over default kubernetes, but not including system configurations outside the realm of kubernetes (see Section 1.2 on limitations for more info), note if the control is implementable as policy-as-code. If not implementable, note the reasons why.
2. **Does the implementation of this policy affect performance?**
   The metrics we will consider are explained in more detail in Section 5.1
3. **Does it have any limitations or cases that cannot be covered?**
   Assuming a default or generic configuration, note any further requirements on top of default kubernetes, for example when external software is required in order to enforce the policy.
4. **Are there any concrete known vulnerabilities that are mitigated by this policy?**
   In the case of a particular policy, if it is stated in the guidelines or in some other way made apparent which vulnerabilities are being mitigated, this is documented.

5.1 Performance

The authors of [27] showed that security-focused runtimes like `gVisor` and `Kata` were not as performant as the default Kubernetes runtime `runc`. As such, the policies could introduce performance overhead when implemented. In our case the overhead is policy execution time resulting in latency increase a when deploying Kubernetes resources. As such, it is beneficial to introduce some benchmarking to ascertain what kind of performance trade-offs are required when implementing some of the policies.

There is a benchmarking tool built in to OPA, which can be helpful to diagnose performance overhead in the implementation. This is most important for policies that are applied frequently. How performance is measured is impacted by the type of policy. The time it takes to run for example a deployment policy is not as important as it is for a network policy.

For resource deployment policies the metric we measure is the increase in latency that is introduced when deploying a new resource. For node configuration policies

the metric that we measure is the one-time latency that is introduced by the check that is executed before kubelet is allowed to start. Each policy is tested 1000 times, this will show what the average increase in latency is for the resource policies.

5.2 Experimental Environment

The development and testing environment consisted of two virtual machines, see Figure 5.1. The first one (VM1) is allocated with 12 vCPUs and 16 GB of RAM. This is virtualized in Proxmox 7.0-11 (Debian 11). VM2 is hosted inside VM1 using qemu and libvrt. This VM is allocated with 4 vCPUs and 2GB of RAM. This nested virtualization allows for easy creation of additional kubernetes nodes.

The software specifications for the two VMs are as follow:

- VM1:
  - OS: Ubuntu Server 21.10
- VM2:
  - OS: Ubuntu Server 21.10
  - Open Policy Agent v0.36.1
  - Kubernetes v1.23.2 (installed using kubeadm)

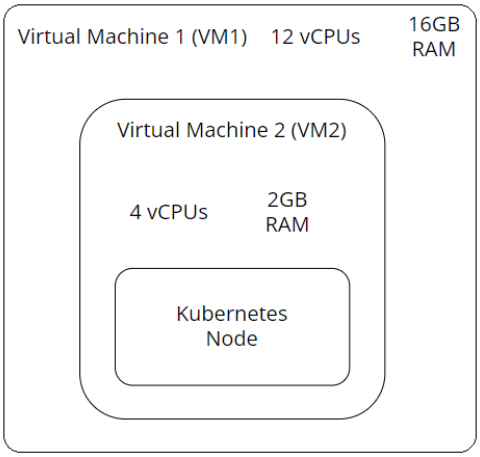

Figure 5.1: Virtual Machine Layout

The Kubernetes node running on VM2 is the control plane node in our configuration, see Section 3.2.1 for more information of what that entails. For internal networking within the cluster, the open-source Container Network Interface (CNI) Calico was chosen due to its popularity in the community.

This control plane node runs OPA alongside a kube-mgmt sidecar which provides the policy engine with kubernetes resource data. OPA fetches its policies from a supplied bundles-url. This is supplied as an argument in the manifest the OPA sidecar, see Figure 5.2 line 7-10. For development purposes this bundle was hosted on a nginx webserver running on the same node. In reality this bundle can be hosted on any server reachable by this pod.

```
- name: opa
      image: openpolicyagent/opa:0.36.1-rootless
      args:
        - run
        - '--server'
        - '--tls-cert-file=/certs/tls.crt'
        - '--tls-private-key-file=/certs/tls.key'
        - '--addr=0.0.0.0:8443'
        - '--addr=http://127.0.0.1:8181'
        - '--set=services.default.url=
              http://nginx-deployment.nginx-app.svc.cluster.local:8080'
        - '--set=bundles.default.resource=bundle.tar.gz'
        - '--log-format=json-pretty'
        - '--set=status.console=true'
        - '--set=decision_logs.console=true'
```

Figure 5.2: Open Policy Agent *opa-server* kubernetes resource manifest

To generate this bundle of policies, the command `opa build -b <SOURCE>` was used, the argument `-b` was added to output the files to a `bundle.tar.gz`-file.

To allow OPA to list namespaces, network policies and resource quotas, line 4-7 in Figure 5.3 were added as arguments to the kube-mgmt sidecar. These are now readable from within OPA under the `data.kubernetes.<resource>` variable.

```
- name: kube-mgmt
      image: openpolicyagent/kube-mgmt:2.0.1
      args:
        - '--replicate-cluster=v1/namespaces'
        - '--replicate=networking.k8s.io/v1/ingresses'
        - '--replicate=networking.k8s.io/v1/networkpolicies'
        - '--replicate=v1/resourcequotas'
```

Figure 5.3: Open Policy Agent *kube-mgmt* kubernetes resource manifest

5.3 Testing and validation

As the policies touch upon different aspects of containers and the orchestration there is no generic way to test the policies. Each test has to be tailored to a specific policy. When testing the implementation of a policy it is important to consider the impact of the decision that is taken to enforce it. A policy should only do the minimal action required to implement the guideline and not be intrusive to the intended functionality of the cluster. An implementation of securing a cluster from outside attacks is to turn of all network traffic. This does however impede on the clusters ability to function correctly.

The node configuration control implementation is tested by configuring the Kubernetes node in such a way that it violates the policies. We anticipate that the effect on performance is negligible and that the check stops the execution of kubelet if the configuration validates the policies, and allows execution if it is valid.

For all of the controls implemented via OPA the built in command `opa test` is used. Each policy has its own tests, both positive and negative tests. The meaning of this is that there is at least one test that checks that the policy allows a valid pod configuration, a positive test. There is also at least one test that checks that the policy denies a invalid configuration, called a negative test. Additional tests were added that tests for inputs where the checked field is missing or empty. OPA checks that the inputs are either `true` or `false`, or that an input field is set to a specific value.

The `opa test` command verifies the correctness and performance of the policies as well as the coverage of the tests. This is done by checking that all lines of the rule is evaluated. As we have both negative and positive tests, all lines should evaluate to a value. The result of the testing includes the total duration to execute the test, this serves as a measure of latency that results in overhead, and whether or not each control passed or failed. As duration to evaluate the rule is measured, this is an indicator of performance. Each policy is tested with mock-data that alters the input on the values that OPA checks, this is done to ensure OPA handles all cases. We anticipate that all policies pass their respective tests, and that the effect on performance is negligible for the intended workload.

6
Results

This chapter presents the results in general, as well as in detail on a per guideline-basis. The questions that these results aim to answer are the ones presented in Chapter 5:

6.1 Node Configuration

Policies that target the configuration of Kubernetes have been classified as *Node Configuration*. The enforcement of these policies needs to happen before Kubernetes, i.e kubelet, is started. The execution of kubelet on each node is typically managed by a systemd service. Before the `kubelet.service` is started, systemd can ensure that certain prerequisites are met. This is done by creating a secondary service that specifies that the `kubelet.service` requires it before starting, see Figure 6.1. This secondary service only executes the script with policy checks, and if all checks pass the script terminates with code 0, meaning success. This allows the `kubelet.service` to start. The script code can be seen in Appendix A.1.

```
1  [Unit]
2  Description=CheckPolices
3  Before=kubelet.service
4
5  [Service]
6  Type=oneshot
7  ExecStart=python3 /home/kubemaster/policy-library/script.py
8
9  [Install]
10 RequiredBy=kubelet.service
```

Figure 6.1: The systemd service created that is required to run before kubelet.

6.2 Resource Deployment

For controls that target resource deployment, OPA is used as a `ValidatingWebhook` as explained in Section 4.1.2. Every resource deployment in Kubernetes that matches the configuration of the `ValidatingWebhook` will generate an AdmissionReview that OPA responds to based on the supplied rules. The `ValidatingWebhook` configuration

can be seen in Figure 6.2. This configuration will generate AdmissionReviews for all operations that either `CREATE` or `UPDATE` resources as seen on line 9.

```
1  kind: ValidatingWebhookConfiguration
2  apiVersion: admissionregistration.k8s.io/v1
3  metadata:
4    name: opa-validating-webhook
5  webhooks:
6    - name: validating-webhook.openpolicyagent.org
7    ...
8      rules:
9        - operations: ["CREATE", "UPDATE"]
10         apiGroups: ["*"]
11         apiVersions: ["*"]
12         resources: ["*"]
```

Figure 6.2: The Open Policy Agent ValidatingWebhook-configuration.

OPA rules use the Rego language which is a high-level declarative language. Each value in the deployment manifest seen in Figure 6.3 can be accessed in OPA from `input.request.object.[<key>]`, for example `input.request.object.metadata.name` would yield the name of the pod.

```
1      apiVersion: v1
2      kind: Pod
3      metadata:
4        name: nginx-test
5      spec:
6        containers:
7          - name: nginx-test-pod
8            image: docker.io/nginx
```

Figure 6.3: Sample Pod deployment manifest.

6.3 Coverage

The legend for the tables in the following two sections is:
Assessment:

- NC = Node Configuration.
- RD = Resource Deployment.

Implementable:

- ✓= Implementable.
- × = Not implementable.
- OOS = Out Of Scope.

6.3.1 NSA

Number	Control	Asses.	Impl.
NSA-1.1	Non-root containers	RD	✓
NSA-1.2	Immutable container file systems	RD	✓
NSA-1.3	Building secure container images	RD	✓
NSA-1.4	Pod security enforcement	RD	✓
NSA-1.5	Protecting Pod service account tokens	RD	✓
NSA-1.6	Hardening container environments	RD	✓
NSA-2.1	Namespaces	RD	✓
NSA-2.2	Network Policies	RD	✓
NSA-2.3	Resource Policies	RD	✓
NSA-2.4	Control plane hardening	NC	Partially
NSA-2.5	Worker node segmentation	NC	×
NSA-2.6	Encryption	NC	✓
NSA-2.7	Secrets	NC	✓
NSA-2.8	Protecting sensitive cloud infrastructure	NC	OOS
NSA-3.1	Authentication	NC	✓
NSA-3.2	Role-based access control	NC	✓
NSA-4.1.1	Logging 1	NC	OOS
NSA-4.1.2	Logging 2	NC	OOS
NSA-4.1.3	Logging 3 - Seccomp: audit mode	RD	✓
NSA-4.1.4	Logging 4	NC	OOS
NSA-4.1.5	Logging 5	NC	OOS
NSA-4.1.6	Logging 6	NC	OOS
NSA-4.1.7	Logging 6	NC	OOS
NSA-4.2.1	Alerting	NC	OOS
NSA-4.3	Tools	NC	OOS
	Total		**14/25**

Table 6.1: Table of NSA guideline coverage.

6.3.2 CIS

Number	Control	Asses.	Impl.
CIS-5.1.1	Ensure that the cluster-admin role is only used where required.	NC	×
CIS-5.1.2	Minimize access to secrets	NC	×
CIS-5.1.3	Minimize wildcard use in Roles and ClusterRoles	NC	×
CIS-5.1.4	Minimize access to create pods.	NC	×

CIS-5.1.5	Ensure that the default service accounts are not actively used.	RD	✓
CIS-5.1.6	Ensure that Service Account Tokens are only mounted when necessary.	RD	✓
CIS-5.1.7	Avoid usage of system:masters group.	NC	×
CIS-5.1.8	Limit use of the Bind, Impersonate, and Escalate permissions in the Kubernetes cluster	NC	×
CIS-5.2.1	Ensure that the cluster has at least one active policy control mechanism in place.	NC	×
CIS-5.2.2	Minimize the admission of privileged containers.	RD	✓
CIS-5.2.3	Minimize the admission of containers wishing to share the host process ID namespace.	RD	✓
CIS-5.2.4	Minimize the admission of containers wishing to share the host IPC namespace.	RD	✓
CIS-5.2.5	Minimize the admission of containers wishing to share the host network namespace.	RD	✓
CIS-5.2.6	Minimize the admission of containers with allowPrivilegeEscalation.	RD	✓
CIS-5.2.7	Minimize the admission of root containers.	RD	✓
CIS-5.2.8	Minimize the admission of containers with NET_RAW capability.	RD	✓
CIS-5.2.9	Minimize the admission of containers with added capabilities.	RD	✓
CIS-5.2.10	Minimize the admission of containers with capabilities assigned.	RD	✓
CIS-5.2.11	Minimize the admission of Windows HostProcess Containers.	RD	OOS
CIS-5.2.12	Minimize the admission of HostPath volumes.	RD	✓
CIS-5.2.13	Minimize the admission of containers which use HostPorts.	RD	✓
CIS-5.3.1	Ensure that the CNI in use supports Network Policies.	NC	×
CIS-5.3.2	Ensure that all Namespaces have Network Policies defined.	RD	✓
CIS-5.4.1	Prefer using secrets as files over secrets as environment variables.	NC	×
CIS-5.4.2	Consider external secret storage.	NC	OOS
CIS-5.5.1	Configure Image Provenance using ImagePolicyWebhook admission controller.	NC	✓
CIS-5.6.1	Create administrative boundaries between resources using namespaces.	RD	✓
CIS-5.6.2	Ensure that the seccomp profile is set to docker/default in your pod definitions.	RD	✓

CIS-5.6.3	Apply Security Context to your Pods and Containers.	RD	✓
CIS-5.6.4	The default namespace should not be used.	RD	✓
	Total		**19/30**

Table 6.2: Table of CIS benchmark guideline coverage.

6.4 NSA Results

The result of implementing the controls in Section 3.3.1 and 3.4.1 is presented in detail in this section.

6.4.1 NSA

6.4.2 *Non-root* containers and *rootless* container engines

- Assessment: Resource Deployment
- Status: Implementable

This control was deemed implementable. NSA specifies that administrators should set the `securityContext.runAsNonRoot` value to `true`. For this control an unset value is the same as `false`. An alternative approach is to set the `securityContext.runAsUser` and `securityContext.runAsGroup` to a non-zero integer. The container `UID` =0 then map to this integer on the host filesystem, therefore not having root privilegies on the host. This can however break some containers due to permissions issues when accessing files. The code implementation of this can be seen in Appendix B.1.

The guidelines mention the usage of rootless container engines, it is possible to run a standalone rootless docker instance, but there is not any widespread support for it. Each new deployment is checked with OPA and the relevant part of the deployment is checked against the rules. The error message that is shown when a misconfigured pod is deployed is shown in Figure 6.4. A correct deployment file is seen in Figure 6.5.

```
Error from server: error when creating "pod.yaml": admission webhook
↪  "validating-webhook.openpolicyagent.org" denied the request:
 -- nginx-test-pod must have securityContext.runAsNonRoot set to true
```

Figure 6.4: Error message when using `kubectl` to deploy a pod without `runAsNonRoot` set to `true`.

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: nginx-test
5   spec:
6     containers:
7       - name: nginx-test-pod
8         image: docker.io/nginx
9         securityContext:
10          runAsNonRoot: true
```

Figure 6.5: `runAsNonRoot` set to `true` on line 10 in a deployment file to allow deployment.

6.4.3 Immutable container file systems

- Assessment: Resource Deployment
- Status: Implementable

This control is implementable. NSA specifies that the setting `readOnlyRootFilesystem` should be set to true. This ensures that the default file system of the container is read-only, which can prevent some vulnerabilities that make use of writing to files that does not need to be written to in the first place. Each container deployment is checked by OPA, and the value of its `securityContext.readOnlyRootFilesystem` is checked and if it is not set to `true` the deployment will fail, this means that an unset value is equal to `false`. The implementation of this can be seen in Figure 6.8. When attempting to deploy a pod that does not fulfill this policy the deployment is stopped and the errors as seen in Figure 6.6 are shown. This setting does not allow for the root-filesystem of the container to change. For the container to be able to write data, a volume needs to be mounted to the container. A correct deployment with write access can be seen in Figure 6.7.

```
Error from server: error when creating "pod.yaml": admission webhook
↪  "validating-webhook.openpolicyagent.org" denied the request:
 -- nginx-test-container must have securityContext.readOnlyRootFilesystem set to
 ↪  true
```

Figure 6.6: Error message when using `kubectl` to deploy a pod without `readOnlyRootFilesystem` set to `true`.

```
      containers:
        - name: nginx-test-pod
          image: docker.io/nginx
          securityContext:
            readOnlyRootFilesystem: true
          volumeMounts:
          - mountPath: /writeable/location/here
            name: volName
        volumes:
        - emptyDir: {}
          name: volName
```

Figure 6.7: `readOnlyRootFilesystem` set to `true` on line 10 in a deployment file to allow deployment. Additional volume mounted to allow writing.

```
deny[msg] {
  allowed_operation
  fields := ["readOnlyRootFilesystem"]
  field := fields[_]
  container := input_containers[_]
  not get_field_value(field, container, input.request)
  msg = sprintf("%v must have securityContext.%v set to true", [container.name,
  ↪  field])

}
```

Figure 6.8: The deny rule written in rego to enforce immutable file systems

6.4.4 Building secure container images

- Assessment: Resource Deployment
- Status: Implementable

This control is deemed implementable as long as the assumption that there is an image-scanner present holds. This is a two-part control as described in the background. Part one involves checking whether or not the URL to each container-image is trusted based on a list of trusted-registries. In our implementation these trusted registries need to be specified by the administrator by adding them to the `config.rego` file, see Figure 6.9. This could be implemented to fetch the config in other ways as well, for example using Kubernetes ConfigMap Resources[1]. If a registry not present in this list is used, the error shown in Figure 6.10 is generated. Every container has an image, see line 8 in Figure 6.7 for an example.

[1] https://v1-23.docs.kubernetes.io/docs/concepts/configuration/configmap/

```
trusted_registries_set := {
        "docker.io",
        "k8s.gcr.io",
}
```

Figure 6.9: `config.rego` file containing the trusted registries.

```
Error from server: error when creating "pod.yaml": admission webhook
↪  "validating-webhook.openpolicyagent.org" denied the request:
 -- Container (nginx-test-container) has untrusted registry.
```

Figure 6.10: Error message when using `kubectl` to deploy a pod with an image from an untrusted registry.

```
// From config.rego
trivy_wrapper_url := "http://localhost:5555"
```

Figure 6.11: `config.rego` file containing the image scanning endpoint.

```
...
scan_image(image) = report {
        request := {
                "url": trivy_wrapper_url,
                "method": "POST",
                "body": {"image": image},
                "timeout": "20s",
        }

        response := http.send(request)
        response.status_code == 200
        report := {
                "status": "OK",
                "message": response.body,
        }
}
...
```

Figure 6.12: `image_scanning.rego` file containing the image scanning rule. Code from the Proof of Concept OPA-rule that calls `trivy`.

The second part to this control is to scan each image that is deployed to the cluster with the use of an image scanner. By assuming that there is an image scanner present somewhere and that it is reachable via http, then the control is implementable with the use of OPA. This is done by specifying the image scanning endpoint the `config.rego` file, see Figure 6.11. OPA sends the registry-link of the image to the image to the image scanner, see line 10 in Figure 6.12. OPA then makes a decision based on the values returned by the image scanner. A proof of concept was created by wrapping the open source image scanner *trivy* with an http-wrapper written in python. This allowed for OPA to fetch scan data, albeit in a slow manner as it was a proof of concept.

6.4.5 Pod security enforcement

- Assessment: Resource Deployment
- Status: Implementable

If the cluster is using the new *Pod Security Admission* currently in beta[2], it is possible to use OPA in order to ensure that each namespace has a pod security admission-label that is either *warn*, *audit*, or *enforce*. If the beta feature is not used then OPA can used to check each deployment configuration, in a similar way to how the *Pod Security Admission* does it.

OPA is able to enforce that the values for the fields in Table 6.3 are compliant with the NSA and CIS policies and deny the pods if they are not. This can done in the same way as described in Section 6.4.3.

Field	**Allowed Value**
allowPrivilegeEscalation	false
privileged	false
hostIPC	false
hostPID	false
hostNetwork	false
allowedHostPaths	dummy path with readOnly: true
runAsUser	non-zero integer
runAsGroup	non-zero integer
supplementalGroups	non-zero integer
fsGroup	non-zero integer
seLinux	see details 6.4.5.1
AppArmor annotations	see details 6.4.5.2
seccomp annotation	seccomp auditing profile
readOnlyRootFilesystem	true[3]

Table 6.3: Table of pod fields that OPA should enforce according to NSA and CIS.

[2] https://v1-23.docs.kubernetes.io/docs/concepts/security/pod-security-admission/

[3] See section 6.4.3 for more information

6.4.5.1 seLinux

This control doesn't specify to use seLinux labeling or any specific labels, but only that one should consider it. However, if a specific label was asked for in the control it would be trivial to implement. This does require that the host operating system has enabled seLinux. A check could be added for the node configuration phase to enforce this as well.

6.4.5.2 AppArmor annotations

Similarly to the seLinux security context annotations, the NSA Guidance makes no specific recommendation to use AppArmor annotations, only to consider it if the host node supports it. Most operating systems have supported this for a long time and some operating systems have it enabled by default. Implementing a policy to require AppArmor annotations is trivial and requires a simple check of the deployment manifest. Additionally the specific options of the applied AppArmor configuration can be checked in the node configuration phase.

6.4.6 Protecting Pod service account tokens

- Assessment: Resource Deployment
- Status: Implementable

This control is deemed implementable. By default kubernetes mounts the default service account token to all pods. The control is performed by checking that the automatic mounting of the default service token is disabled. To disable it the `automountServiceAccountToken` needs to be set to `false` in the pods deployment yaml. For the cases where a service account is needed, explicitly create and mount a non-default service account. Most pods do not require direct access, and therefore does not use a service account. If the pod is compromised the token can be used by an adversary to further compromise the cluster.

```
Error from server: error when creating "pod.yaml": admission webhook
↪  "validating-webhook.openpolicyagent.org" denied the request:
 -- Pod nginx-test does not have automountServiceAccountToken set to false.
```

Figure 6.13: Error message when using `kubectl` to deploy a pod that does not have `automountServiceAccountToken` set to `false`.

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: nginx-test
5   spec:
6     automountServiceAccountToken: false
7     containers:
8       - name: nginx-test-pod
9         image: docker.io/nginx
```

Figure 6.14: `automountServiceAccountToken` set to `false` on line 6 in a deployment file to allow deployment.

6.4.7 Hardening container environments

- Assessment: Resource Deployment
- Status: Implementable

This control consists of the following parts:

- **Hypervisor-backed containerization**
  Using OPA it is possible to enforce that a Pod has a specific runtime-class defined in in order for it to be deployed. This does however require that the specified runtime-class is defined on the host system.
- **Kernel-based solutions**
  The seccomp-tool can limit a containers abilities to perform syscalls. This is however disabled by default. This can be enforced, see section 6.4.5
- **Application sandboxes**
  These sandboxes are defined as kubernetes runtime-classes, and are therefore also enforceable in the same manner as the Hypervisor-backed containerization.

6.4.8 Namespaces

- Assessment: Resource Deployment
- Status: Implementable

This control is partially up to interpretation, but both interpretations have been determined by us to be fully implementable. Both interpretations identified specifies that kube-system and kube-public are for cluster services. The difference lies in how strict to be in regards to the *default* namespace and isolation of applications. The *default* namespace is defined by kubernetes as *"The default namespace for objects with no other namespace"* [28]. The different interpretations lead to a similar implementation. Check that the namespace of a deployment is not in the set `kube-system, kube-public`, or the set `kube-system, kube-public, default` depending on how to handle the default namespace. Regardless of how it is interpreted both examples can be written as a policy, see Figure 6.16. Figure 6.17 shows the

error when trying to deploy the pod seen in Figure 6.15 in the protected namespace `kube-public` specified on line 5.

```
1     apiVersion: v1
2     kind: Pod
3     metadata:
4       name: nginx-test
5       namespace: kube-public
6     spec:
7         ...
```

Figure 6.15: Deployment file of a pod in namespace *kube-public*

```
1  deny[msg] {
2    is_kube_ns
3    msg := sprintf("Pod %v can not be deployed in namespace %v",
     ↪  [input.request.object.metadata.name,
     ↪  input.request.object.metadata.namespace])
4  }
5  is_kube_ns {
6    input.request.object.metadata.namespace == "kube-public"
7  }
8  is_kube_ns {
9    input.request.object.metadata.namespace == "kube-system"
10 }
11 is_kube_ns {
12   input.request.object.metadata.namespace == "default"
13 }
```

Figure 6.16: The deny rule written in rego to protect kube namespaces.

```
Error from server: error when creating "pod.yaml": admission webhook
↪  "validating-webhook.openpolicyagent.org" denied the request:
 -- Pod nginx-test can not be deployed in namespace kube-public
```

Figure 6.17: Error message when using `kubectl` to deploy a pod to the protected namespace *kube-public*.

6.4.9 Network Policies

- Assessment: Resource Deployment

- Status: Implementable

The NSA hardening guidance lists three steps to follow to when implementing network policies. These are:

1. Use a CNI plugin that supports the NetworkPolicy API.
2. Create policies that select Pods using podSelector and/or the namespaceSelector.
3. Use a default policy to deny all ingress and egress traffic. Ensures unselected Pods are isolated to all namespaces except kube-system

By default kubernetes does not restrict any traffic and all pods can communicate with any other pod. If there is a route from the external network in to a pod via for example a service then all external clients are also able to communicate with it.

The implementation of this control can be made in two different ways depending on which CNI is used within the cluster. To be able to use the network policies the CNI needs to support it. Calico was used in our implementation, because it supports this NetworkPolicy API. When investigating this control it was also discovered that Calico has an extra resource, namely the `GlobalNetworkPolicy`. This resource allows for administrators to setup non-namespaced Network Policies that could simply select all pods in the cluster and not the ones in a single namespace.

```
1 has_networkpolicy {
2   networkpolicies[input.request.object.metadata.namespace]
3 }
4
5 deny[msg] {
6   input.request.kind.kind == "Pod"
7   not has_networkpolicy
8   msg := sprintf("Pod (%v) is not in a namespace with an existing
    ↪  NetworkPolicy", [input.request.object.metadata.name])
9 }
```

Figure 6.18: The deny rule written in rego to enforce that at least one NetworkPolicy is present in the namespace.

The other way of implementing this is to use standard kubernetes `NetworkPolicy`. These are namespaced and it is therefore necessary for them to be applied to each namespace. OPA is then used to check the deployments, in essence it checks that there exists a network policy that applies to the namespace the deployment targets, see Figure 6.18 and Figure 6.19. It does not necessarily check that the network policy applies to the specific resource in question, but that there is one in the namespace. Combining this control with the argument that all traffic not explicitly allowed should be denied means that all namespaces should have a default-deny policy enabled.

```
1  Error from server: error when creating "pod.yaml": admission webhook
   ↪ "validating-webhook.openpolicyagent.org" denied the request:
2  -- Pod (nginx-test) is not in a namespace with an existing NetworkPolicy--
```

Figure 6.19: Deployment file of a pod in a namespace that does not have a NetworkPolicy.

One way to make sure that the pod is covered by a network policy is to create one before creating the pod. This can be done in the same file by appending the deployments as in Figure 6.20. First a `Namespace` is created, and after that the `NetworkPolicy` is created, with a podSelector (line 16). This podSelector matches the one for the pod (line 32). This specific policy allows ingress only on port 80, and block all egress. The same procedure can be used for a default deny policy. A new policy can later be added to allow the expected ingress-traffic.

```yaml
kind: Namespace
apiVersion: v1
metadata:
  name: nginx-test
  labels:
    app: test
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: nginx-test-networkpolicy
  namespace: nginx-test
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - ports:
    - port: 80
  egress:
  - {}
  policyTypes:
  - Ingress
  - Egress
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-test
  namespace: nginx-test
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: docker.io/nginx
```

Figure 6.20: Deployment file of a namespace, network policy, and pod.

6.4.10 Resource Policies

- Assessment: Resource Deployment
- Status: Implementable

This control is implementable. Resource policies can mitigate denial of service attacks where a pod would consume all available computational power and/or memory

bandwidth. By limiting the amount of resources a pod is allowed to consume we can make sure that accidental misconfigurations on one pod does not have negative effects on other pods on the same node. As ResourceQuotas control the total resource resource consumption within a single namespace, the targeted namespace is checked for a `ResourceQuota`. By creating a `LimitRange` one can have kubernetes automatically assign a minimum resource allocation to all new pods. Unless this is done each pod needs to have its limits set. OPA is able to check this as well by checking the deployment's namespace and checking for the LimitRanges and ResourceQuotas applied to it. The checks for ResourceQuotas is seen in Figure 6.21.

```
1  has_resourcequota {
2    resourcequotas[input.request.object.metadata.namespace]
3  }
4
5  deny[msg] {
6    input.request.kind.kind == "Pod"
7    not has_resourcequota
8    msg := sprintf(
9      "The namespace of pod (%v) does not have a ResourceQuota",
10     [input.request.object.metadata.name],
11   )
12 }
```

Figure 6.21: The deny rule written in rego to enforce that at least one ResourceQuota is present in the namespace.

```
Error from server: error when creating "pod.yaml": admission webhook
↪  "validating-webhook.openpolicyagent.org" denied the request:
-- Container nginx in Pod nginx-test has no resource limits set--
-- The namespace of pod (nginx-test) does not have a ResourceQuota
```

Figure 6.22: Deployment file of a pod in a namespace without `ResourceQuota` and `LimitRange`.

To pass the error on line 4 of Figure 6.22 a `ResourceQuota` needs to be created for the namespace, see Figure 6.23. This `ResourceQuota` enforces that the total resource usage of all containers in the namespace stays below 2 GiB and 2 CPU cores. No single container is allowed to request more than 1 CPU core and more than 1 Gibibyte of memory at the time of creation. These requests are added to the containers deployment file, see Figure 6.24.

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: nginx-test-rq
  namespace: nginx-test
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Figure 6.23: Deployment file of a ResourceQuota for a namespace.

To automate this, a LimitRange can be created. It will automatically allocate a default limit to all containers unless another limit is specified. This default limit can be seen in Figure 6.25 on line 10-12.

```yaml
  containers:
  - name: nginx
    image: docker.io/nginx
    resources:
      limits:
        memory: "512Mi"
        cpu: "800m"
      requests:
        memory: "256Mi"
        cpu: "400m"
```

Figure 6.24: Deployment file of a pod with limits.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: nginx-test-limit-range
spec:
  limits:
  - default:
      memory: 256Mi
      cpu: "500m"
    defaultRequest:
      memory: 128Mi
      cpu: "500m"
    type: Container
```

Figure 6.25: Deployment file of a limitrange that automatically allocates resources for new containers.

6.4.11 Control plane hardening

- Assessment: Node configuration
- Status: Partially

NSA specifies that the control plane should be highly protected due to its sensitive capabilities. To do this there are six steps to follow, and these are:

- Set up internal TLS encryption.
  Assessment: Implementable
- Use strong authentication methods.
  Assessment: Implementable
- Disable access to internet and unnecessary, or untrusted networks.
  Assessment: Not implementable
- Use RBAC policies to restrict access.
  Assessment: Implementable
- Secure the etcd-datastore with authentication and RBAC policies.
  Assessment: Not implementable
- Protect `kubeconfig` files from unauthorized modifications.
  Assessment: Implementable

Set up TLS encryption

Scan kube-api manifest for `--insecure-port=0` to make sure the unsecure port is disabled. This can be done by scanning the kube-apiserver manifest before kubelet is allowed to start on the control plane. This flag has been non-operational since Kubernetes v1.20 and will be removed in v1.24 so a control for this has not been written. It can however be done in the same way as `--anonymous-auth=false` seen on line 30 in Appendix A.1.

Strong authentication

Kubernetes assumes that a cluster-independent service manages *normal user accounts*[4]. This is further explained in Section 6.4.16, it is possible to check that the kube-apiserver has the arguments for any strong authentication set and that the weak ones are disabled. The support for basic password authentication was deprecated in kubernetes v1.16 and removed in v1.19[5]. It can therefore be argued that kubernetes currently only supports strong authentication methods. This control can be implemented by checking that at least one authentication method is enabled.

Untrusted networks

NSA specifies that the kubernetes api-server should not be exposed to any untrusted networks such as the internet. This can be done using either an NetworkPolicy or a dedicated firewall on the network. The resources in *kube-system* needs to be able to communicate with other control plane resources as well as other worker nodes. For the control plane this can be done by allowing the following ports:

Protocol	Direction	Port	Service
TCP	In	6443	API-server
TCP	In	2379, 2380	etcd API
TCP	In	10250	kubelet API
TCP	In	10257	kube-controller-manager
TCP	In	10259	kube-scheduler

Table 6.4: Ports that the control plane should allow access for.

In the case where this is done with a `NetworkPolicy` there is no general way to enforce it as the network topologies differ between setups. The same applies when using a firewall.

RBAC Policies

Can only enforce that RBAC is enabled, the usage of it is not enforceable, see Section 6.4.17.

Secure etcd

From Section 6.4.17 we can conclude that we can not enforce the usage of RBAC. NSA specifies that the etcd datastore should only trust certificates assigned to the API-server. As there is no way to determine if the supplied certificate is only available to the Kubernetes API the control is deemed unenforceable.

[4] https://v1-23.docs.kubernetes.io/docs/reference/access-authn-authz/authentication/

[5] https://github.com/kubernetes/kubernetes/commit/df292749c9d063b06861d0f4f1741c37b815a2fa

Protect `.kubeconfig` files

As described in Section 6.1 it is possible to add checks that are executed before kubelet is allowed to start. This control is part of the node configuration check. The ownership and permissions of the files are checked to make sure that the owner of the home folder the files are contained in is the only one to have read permissions. This code can be seen in Appendix A.1 line 69-74.

6.4.12 Worker node segmentation

- Assessment: Node configuration
- Status: Not implementable

The worker node segmentation can be implemented by placing worker nodes on a separate network where there are only worker nodes and no other machines. If a worker node becomes compromised it can only access other worker nodes. As this is done through firewalls or physical network segmentation this can not be enforced through policy-as-code.

6.4.13 Encryption

- Assessment: Node configuration
- Status: Implementable given certain conditions

This control requires that all internal communication should be encrypted using TLS 1.2 or 1.3. This can be set up using service meshes such as Istio[29] or Linkerd[30] and enforcing mTLS for example. In general it is not possible to achieve generic mTLS without the use of a service mesh or some other custom sidecar solution that handles network traffic. This leads to there being different policies needed for different service meshes. As an example, if the cluster makes use of an Istio service mesh it is possible to restrict the usage to only TLS by applying a *PeerAuthentication*[6] resource to the namespace that needs to be restricted. We deem this to be enforceable under the condition that the cluster has to be using a service mesh that supports mTLS.

6.4.14 Secrets

- Assessment: Node configuration
- Status: Implementable

Secrets in kubernetes are by default stored as unencrypted base64-strings that can be read by anyone that has access to the API. There are two ways of encrypting these secrets, one of them being the usage of an external key management service (KMS) and the other is to enable Secret data-at-rest encryption in the API-server. To enable encryption in both cases the kube-apiserver requires the `--encryption-provider-config` to be present in the kube-apiserver manifest. As this manifest is executed before

[6] https://istio.io/v1.15/docs/reference/config/security/peer_authentication/

kubernetes and any policy-engines are running, it is not possible to have the policy engine for resource deployment check that this manifest contains the argument `--encryption-provider-config`. This is done during the node configuration phase described in Section 6.1.

6.4.15 Protecting sensitive cloud infrastructure

From the limitations presented in section 1.2 this guideline was deemed out of scope.

6.4.16 Authentication

- Assessment: Node configuration
- Status: Implementable

As kubernetes has two kinds of users and the usage of service accounts was handled in section 6.4.6 this section will be regarding the normal user accounts and the authentication of those. Kubernetes assumes that the administrator has set up a third party authentication service for normal accounts, such as OpenID Connect or X509 client certificates. The enforcement of authentication is possible as previously described in Section 6.4.11. In order to require authentication for all calls to the api-server it needs to be started with the `--anonymous-auth=false` argument. This denies all api-requests that does not have authentication. This argument is set in the `kube-apiserver` manifest and loaded as kubernetes starts. Therefore it needs to be checked as part of the node configuration checks.

6.4.17 Role-based access control

- Assessment: Node configuration
- Status: Implementable

RBAC is enabled by default, but it can be disabled. In order to enforce that it is enabled the kube-apiserver manifest needs to be checked prior to kubelet starting. The rest of the control only specifies to use RBAC to limit access to resources. Any specific RBAC rules are not specified in the control. The kube-apiserver manifest needs to be checked during the node configuration checks to ensure that `--authorization-mode=<LIST>` contains `RBAC`. This implementation can be seen in Appendix A.1 line 44.

6.4.18 Logging

All controls in this category except *Seccomp: audit mode* relies on an external logging server/service to be present. Because of the first limitation 1 the other controls are out of scope.

- Kubernetes native audit logging configuration.
  - Assessment: Node configuration.
  - Status: Out of scope
- Worker node and container logging.

  - Assessment: Node configuration.
  - Status: Out of scope
- Seccomp: audit mode
  - Assessment: Resource Deployment
  - Status: Implementable

  Using OPA it is possible to ensure that the value `input.request.object.spec.securityContext.seccompProfile.type` is set to Localhost and that a profile is present. This rule can be seen in Figure 6.26
- Syslog
  - Assessment: Node configuration
  - Status: Out of scope
- SIEM platforms
  - Assessment: Node configuration.
  - Status: Out of scope
- Service meshes
  - Assessment: Node configuration.
  - Status: Out of scope
- Fault tolerance
  - Assessment: Node configuration.
  - Status: Out of scope

```
1 correct_seccomp {
2     input.request.object.spec.securityContext.seccompProfile.type == "Localhost"
3     input.request.object.spec.securityContext.seccompProfile.localhostProfile
4 }
```

Figure 6.26: OPA Rule that ensures that a pod has a seccomp profile.

6.4.19 Threat Detection

Both *Alerting* and *Tools* require external tools to be used, such as *Security Information and Event Management* (SIEM) to monitor logs and *Intrusion Detection Systems* (IDS) to monitor the cluster. Due to the first limitation 1 these are out of scope.

Alerting

- Assessment: Node configuration.
- Status: Out of scope

Tools

- Assessment: Node configuration.
- Status: Out of scope

6.5 CIS Results

The following CIS-controls do not overlap with the ones in NSA as described in Section 3.4.2.

Non-generic controls

- CIS-5.1.1 Ensure that the cluster-admin role is only used where required.
- CIS-5.1.2 Minimize access to secrets.
- CIS-5.1.3 Minimize wildcard use in Roles and ClusterRoles.
- CIS-5.1.4 Minimize access to create pods.
- CIS-5.1.8 Limit use of the Bind, Impersonate, and Escalate permissions in the Kubernetes cluster.

These five controls have the same assessment, status, and motivation, and are therefore combined into one.

- Assessment: Node configuration.
- Status: Not implementable.

There is no generic approach to these controls as they need to be reviewed on a case-by-case basis. This depends on organizational structure and is too complicated to perform programmatically. These controls are not out of scope as they target configurations of kubernetes resources, but is not implementable due to relying on external factors or decisions.

CIS-5.1.7 Avoid usage of system:masters group

- Assessment: Node configuration.
- Status: Not implementable.

This can not be enforced as a policy, it can however be implemented as a periodic check in the form of a cronjob or something similar.

CIS-5.2.1 Ensure that the cluster has at least one active policy control mechanism in place

- Assessment: Node configuration.
- Status: Not implementable.

This can not be enforced as a policy, it can however be implemented as a periodic check in the form of a cronjob or something similar.

CIS-5.2.8 Minimize the admission of containers with NET_RAW capability

- Assessment: Resource Deployment.
- Status: Implementable.

This is implementable and can be enforced with OPA with the following rule, see Figure 6.27. This requires each pod to explicitly drop the NET_RAW capability.

```
package kubernetes.admission.cis_drop_net_raw

deny[msg] {
        input.request.kind.kind == "Pod"
        container := input.request.object.spec.containers[_]
        not in_list("NET_RAW")
        msg := sprintf("Pod \%v does not have
        ↪  securityContext.capabilities.drop set to \%v",
        ↪  [input.request.object.metadata.name, "NET_RAW"])
}
in_list(value) {
        container := input.request.object.spec.containers[_]
        container.securityContext.capabilities.drop[_] == value
} else = false {
        true
}
```

Figure 6.27: Rego rule that denies the admission of pods that does not drop the NET_RAW capability.

CIS-5.2.9 Minimize the admission of containers with added capabilities

- Assessment: Resource Deployment.
- Status: Implementable.

This can be implemented in the same fashion as the previous control, see Section 6.5

CIS-5.2.10 Minimize the admission of containers with capabilities assigned

- Assessment: Resource Deployment.
- Status: Implementable.

This can be implemented in the same fashion as the previous control, see Section 6.5

CIS-5.2.11 Minimize the admission of Windows HostProcess Containers

From our limitations presented in section 1.2 this guideline was deemed out of scope.

CIS-5.3.1 Ensure that the CNI in use supports Network Policies

- Assessment: Node configuration.
- Status: Not implementable.

The process of figuring out if the CNI in use supports Network Policies requires reading the documentation for the CNI currently in use. This can not be done in a generic way. A list of common CNIs can be compiled, but it can never be up to date.

CIS-5.4.1 Prefer using secrets as files over secrets as environment variables

- Assessment: Node configuration.
- Status: Not implementable.

This is not a Kubernetes problem, but rather a problem with the code of the application running inside the container. Applications that require secrets as environment variables needs to be modified, which is not in our scope. The problem is relevant, but the remediation is not.

CIS-5.4.2 Consider external secret storage

- Assessment: Node configuration.
- Status: Out of scope.

As this guideline targets external secrets via a cloud provider or other third party secrets management solution, this is deemed out of scope per limitation 1 and 2.

CIS-5.5.1 Configure Image Provenance using ImagePolicyWebhook admission controller

- Assessment: Node configuration.
- Status: Implementable.

In order to use an ImagePolicyWebhook controller the API-servers requires two arguments. The arguments are `--admission-control-config-file=<file>` and `--enable-admission-plugins=ImagePolicyWebhook`. The script in Appendix A.1 includes a check for this on lines 34 and 48. The file supplied to the first argument needs to contain the `configuration.imagePolicy` object.

6.6 Performance

In the following section the latencies are presented in boxplots where 50% of the n values are inside the box. Outliers are marked in red, and represent 0.35% of values on either end of the distribution. This means that 99.3% of the n values are inside the whiskers.

6.6.1 Node configuration

To measure the performance impact of the node configuration script the python module `perf_counter` was used. The result of measuring the evaluation time of $n = 1000$ evaluations of the node configuration script can be seen in Figure 6.28. This module measures the time it takes to execute the script, which will result in overhead in the form of latency. This shows that the check has an insignificant impact on the performance (i.e the time it takes to start kubelet) which is the expected outcome. This script can be seen in Appendix A.1.

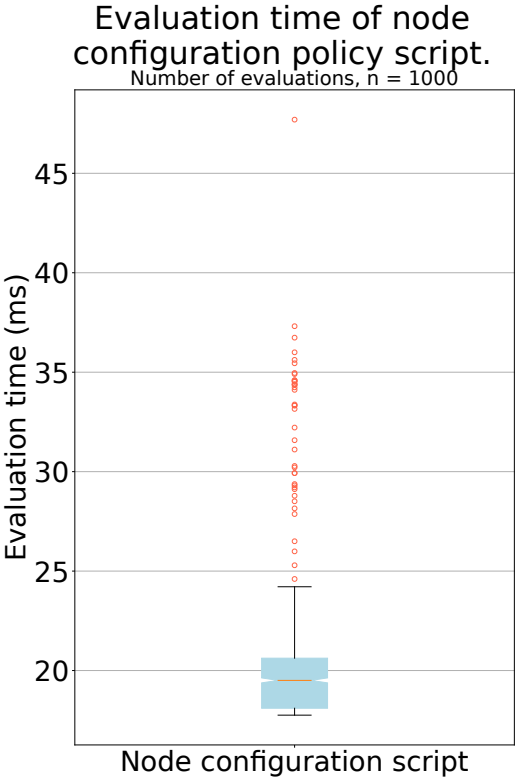

Figure 6.28: The evaluation time of all node configuration policies combined in the script.

6.6.2 Resource Deployment

To measure the performance of the resource deployment policies we measure the time it takes to execute the policies. This gives us the latency overhead of each individual policy. We expect these times to be very small, and the results show that the median for all single policies is below 5ms. The absolute majority of all values are below 10ms. There are however a few outliers marked in red. This does not mean that these policies in particular have a large overhead but that there was likely due to CPU scheduling, caching or other normal factors due to the low execution time in general. Each policy evaluation during resource deployment takes a few milliseconds which is much lower in comparison to the time it takes to download container images, which depending on network speed can take several seconds. All policy checks during the this phase have no large impact on the performance, with the exception of image scanning which has a large timeout due to our proof of concept design, explained in Section 7.3.1.A graph of the all the latencies when the policies are executed on their own can be seen in Figure 6.30. When all policies are combined in to a single rule, the time it takes for the rule to evaluate is smaller than when executed on their own. The result of a rule that does this can be seen in Figure 6.29

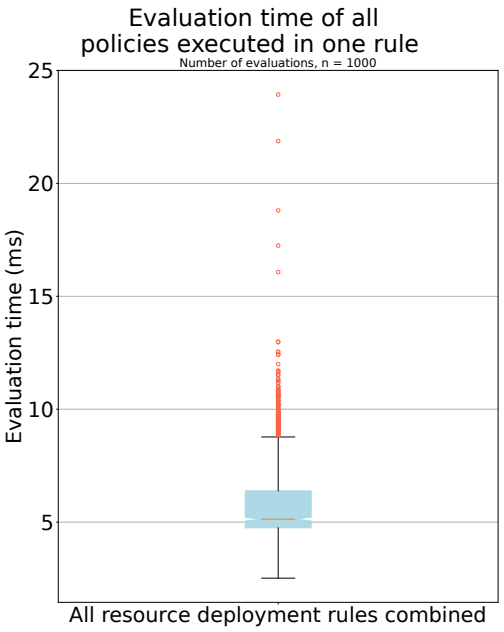

Figure 6.29: The execution time of all resource deployment policies executed as a single rule.

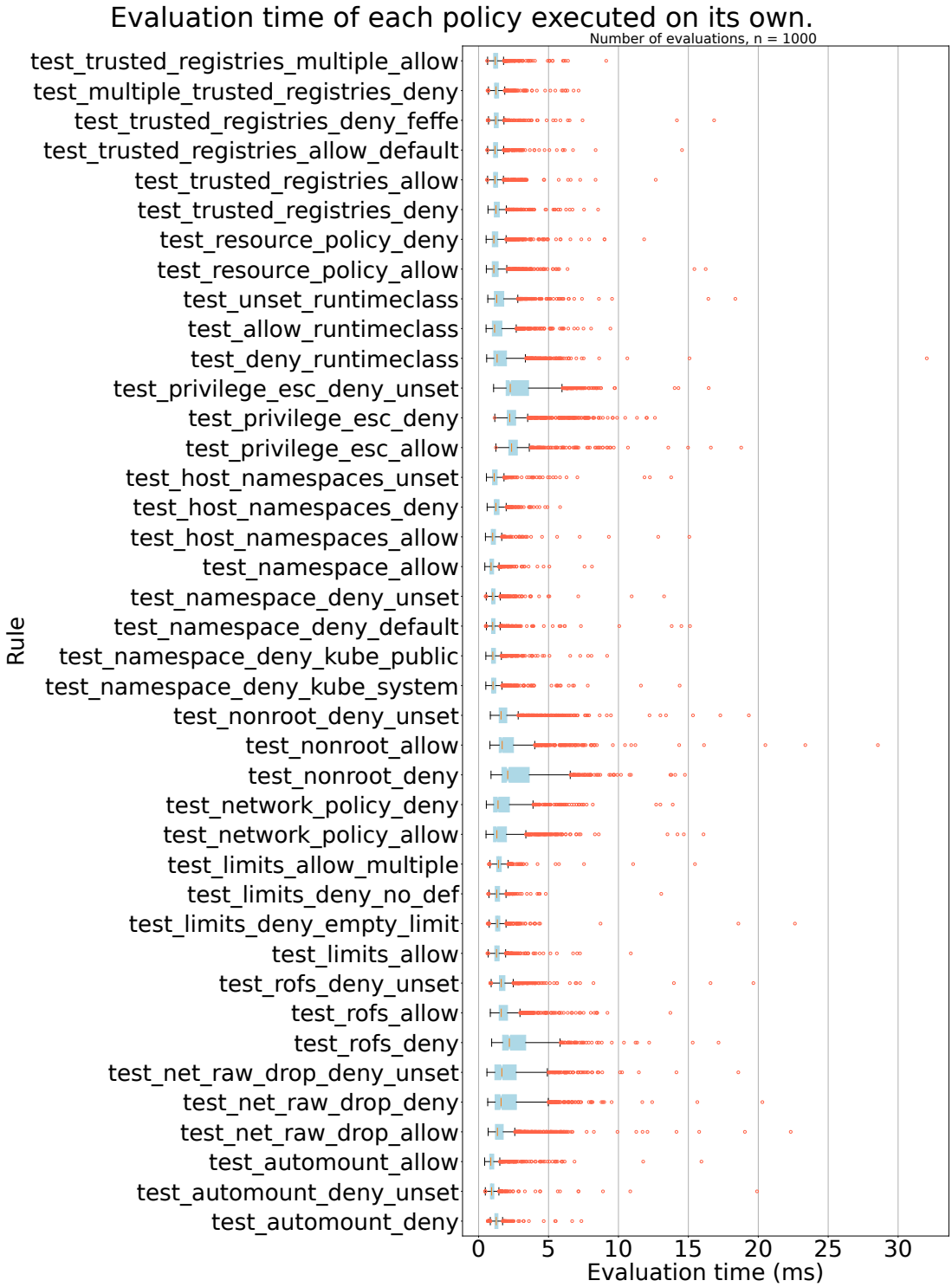

Figure 6.30: The execution time of all resource deployment policies executed on their own.

7
Discussion

7.1 Changes in Kubernetes

During the time period that this work was conducted, Kubernetes and OPA received several updates. In addition, the NSA/CISA hardening guidance also received an update which altered some parts of the guidance. We have aimed to fulfill the latest version, but in the cases where an implementation had already been created we kept it in the report. One of the bigger changes is NSA-1.4 that in the first version detailed the use of Pod Security Policies (PSP), now deprecated and set to be removed in Kubernetes v1.25. The replacement for PSP is Pod Security Admissions that is currently in beta. As explained in Section 5.2 detailing the experimental environment, the Kubernetes version used is v1.23.2 and the OPA version is v0.36.1. Kubernetes and OPA were not kept up to date as this could have unforeseen negative effects on our system which would take time away from the investigation.

7.2 Enforcement on different levels

The results show that all 29 controls targeting resource deployment are implementable, whilst only five out of 16 controls for node configuration are implementable. The common factor between the five that are implementable are that they are configuration settings in files present on the node file system. These files can be read without kubernetes running and this allows for enforcement by blocking execution of kubernetes if the configuration breaks the policies. The unimplementable controls in the node configuration category have several common factors, one of these being that they are configuration settings that need to be reviewed on a case-by-case basis, such as user permissions. Another factor is that they target kubernetes configurations that depend on, for example, network or organizational structure.

The node configuration controls in our implementation are checked through the use of a python script, but it can be checked in various ways. There is room for a policy engine that can check the kubernetes installation before it is allowed to start. While it it possible to use a standalone version of OPA that runs from the command line by executing `opa eval --data <data> --input input.json <rule>`, scripting a simple proof-of-concept with python was the path of least resistance since we had used it before, but this could also be done in other scripting languages such as bash.

7.3 Guidelines discussion

For all resource deployments it is possible to alter the rules a little bit and be able to ignore specific rules by assigning labels. Containers that require root privileges to run are completely blocked from running with these controls in place. Since these are *guidelines* it is allowed to alter them and still be considered safe as long as you are aware of the consequences. The majority of CIS-controls use the wording *"Minimize..."* which implies that containers which for example requires root privileges should only be minimized and not denied. One might argue that instead of stopping a container violating a policy from running, there could simply be a warning instead. The drawback of this is that a container might be executed with too many privileges. Denying allows this to be solved by first blocking the execution and then adding as few privileges as possible to make the container run.

7.3.1 Image Scanning

The 6.4.4 control is an odd case as it requires the use of an external program to perform the actual scan. The policy itself however, can be agnostic and may rely on any image scanner that provides the necessary information to make a decision. Practically, the policy has bad performance, at least with our proof-of-concept. It can take more than 10 seconds in some cases to perform the scan and that time can be increased with image size and the number of layers in the image, or if the scanner database is out of date and needs updating before performing the scan. However, it was included as a proof-of-concept, and it technically achieves the intended purpose.

7.3.2 Encryption

Enforcing TLS sounds like it might be something that's fairly trivial to achieve, but no, it can in fact often be relatively difficult to enforce that no non-TLS communication is allowed to happen. The solution to this that was presented in results makes use of mTLS, which in the Istio[1] implementation works by placing network-managing sidecars on every pod and that network-managing sidecar is written to communicate using TLS only.

Another solution to achieving encrypted networking across the cluster is to use something like Calico's WireGuard[2] setting. This achieves encrypted mutual pod communication, but not using TLS 1.2 or 1.3 which the control specifies. In a scenario where these guidelines are followed as, well, guidelines, we recommend investigating the possibility of implementing cluster wide encryption using something like WireGuard[31], as its configuration can often be simpler.

[1] https://istio.io/v1.15/about/service-mesh/

[2] https://projectcalico.docs.tigera.io/archive/v3.24/security/encrypt-cluster-pod-traffic

7.3.3 Performance

This investigation quantitatively shows that the use of these policy-as-code implementations does not have significant overheads, with the latency added being below 5ms for the absolute majority of all single policies, with a median of around 5ms as shown in Figure 6.30 in Section 6.6.2. All policies executed in one rule on the same input shows a similar overhead, with a median of around 5ms. This does not account for the image scanning policy in Section 7.3.1 as the execution of this is partially based on which image is used for the scan and the state of the scanner database.

7.3.4 Controls for node configuration

Some of the controls outlined in the NSA hardening guidance are meant to be applied during cluster setup or bootstrapped into some core cluster configuration at a later point. These kinds of controls are hard to conceptually deal with in traditional policies as the methods of achieving the desired cluster configuration are varied and there is no one standard method of accomplishing it. Some of these controls could theoretically be implemented in the setup process, by applying policies to whatever runs the cluster setup. One of the more popular semi-automated methods of deploying a cluster is to use kubespray[3] which in essence is just a set of ansible roles. At this moment there is no publicly available method to apply policy code to an ansible playbook but in theory it should be possible. All that would be required is that some program wraps the roles and configurations for the ansible playbook, and then either denies or allows the request to run the playbook. Alternatively, it could also be possible to extend ansible itself to provide the actions to a policy engine and follow the policy engine's decision.

7.4 Lack of runtime policies

At the start of this investigation we thought that at least a few policies would target runtime, in the end that was not the case. The guidelines appear to be targeted towards the configuration of the cluster and not the runtime. NSA/CISA has two sections on Alerting and Threat Detection, and to consider using external tools. Although this is out of scope, these two are good examples of runtime policies as these monitor the cluster in its running state and would be able to perform actions depending on what is going on. Kubernetes lack of a native auditing solution is the reason as to why these require external tools and are out of scope to our investigation.

Monitoring the cluster in its running state could help secure it by monitoring the running processes and acting upon any anomalies. From Falco's[4] documentation we can see that it is possible to notify when for example a shell is spawned in a container. This could be an indicator that a container is compromised and that an adversary has created a shell process. Using Falco in conjunction with some other tool to close down this container would be a perfect example of a policy that targets the

[3] https://github.com/kubernetes-sigs/kubespray
[4] https://v0-33.falco.org/docs/examples/

runtime. Although external tools are out of scope for us, it can be an important part of hardening a cluster as it is impossible to cover all angles of attack, and by monitoring the runtime it is possible to see when anomalies occur.

7.5 Working with OPA and Rego

This section will discuss the challenges of using OPA and Rego as we had not used them before.

7.5.1 Benefits and drawbacks of using OPA

OPA and KubeLinter[5] are the two most common open source Kubernetes security softwares used by the Kubernetes community[2]. OPA can run automatically whereas KubeLinter requires manual usage. Using OPA meant that we had to learn a new language namely Rego. Using other solutions such as Kubewarden[6] would have allowed us to use languages such as `Rust` or `Go` among others. As OPA is the most common choice for automatically enforcing policies, there is a large community and great documentation which helped us during development.

In Section 6.4.4 about image scanning, the execution of the image scanner is done by a wrapper written in python that is called by OPA via http. This was a workaround to what we found to be a drawback of OPA/Rego, that it is unable to execute shell commands to make use of external software.

7.5.2 Challenges

- The debugging phase of writing rego rules can be quite tedious as the rule stops evaluation once one line in the rule-body evaluates to false. This means that error messages placed further down in the rule will simply not show.
- In order to debug the rules as they were evaluated, we needed to access the logs for the OPA container and refresh them periodically.
- The result when using `input.request.metadata.<key>` for a key that is missing in the deployment manifest is `undefined`. Comparing `undefined` to either `true` or `false` will in turn also evaluate to `undefined`. For settings where a omitted key equals an unsafe default value, a check needs to be done to ensure the key exists before comparing it to the values specified by the policy.
- Describing some common relations like `every` required superfluous wrapping rules. For the `every` example this has been addressed in OPA v0.38.1 as a new keyword[7].

[5] https://github.com/stackrox/kube-linter
[6] https://www.kubewarden.io/
[7] https://www.openpolicyagent.org/docs/v0.38.1/policy-language/#every-keyword

8

Conclusion

From the results we conclude that a majority of controls were enforceable through policy-as-code. In the category resource deployment this number was 100%. Controls that target the node configuration were not enforceable to the same extent, with only $\approx$ 31% being fully enforceable. In total 60% (33 out of 55) of controls where enforceable. The common factor for the set of non enforceable controls are that they are either too complicated to implement programmatically or that they are unable to be written in a generic way due to factors outside the scope of Kubernetes. The overhead for policy execution during resource deployment was around 5 milliseconds, which is a negligible amount of time for the intended workload. For node configuration policies the total execution time was around 20 milliseconds on our experimental setup which is also a negligible amount.

In conclusion, policy-as-code appears to be a promising approach to automate implementation and enforcement of common security measures which can simplify the workflow for many, but that it does not necessarily guarantee full coverage of common guidelines at this point in time. The implementation of the guidelines as policies has proven to be time consuming, which further strengthens the motivation of automatic policy enforcement. The iterative approach to implementing the guidelines, i.e focusing on one control at a time proved to be a successful approach.

This investigation brings knowledge on to what extent common security guidelines are enforceable though policy-as-code systems, what the limitations are, and what the effect on performance is. We hope this in turn allows the security community to make well-founded decisions as to what can be further investigated in the future in order to advance in the field of automatic policy enforcement and the increase in security it can provide if implemented correctly.

Bibliography

[1] Cloud Native Computing Foundation, *CNCF SURVEY*, Accessed: 2022-12-11, 2020. [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[2] RedHat, *State of Kubernetes security - 2022*, Accessed: 2022-12-11. [Online]. Available: https://www.redhat.com/en/resources/state-kubernetes-security-report.

[3] World Economic Forum, *The Global Risks Report 2022*, Accessed: 2022-12-11. [Online]. Available: https://www3.weforum.org/docs/WEF_The_Global_Risks_Report_2022.pdf.

[4] Center for Internet Security, *CIS Benchmarks - Securing Kubernetes*, Accessed: 2022-12-11. [Online]. Available: https://www.cisecurity.org/benchmark/kubernetes/.

[5] National Security Agency, Cybersecurity and Infrastructure Security Agency, *Kubernetes hardening guidance v 1.2*, Accessed: 2022-12-11. [Online]. Available: https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF.

[6] European Parliament, Council of the European Union, "Council regulation (EU) no 2016/679," *OJ*, vol. L 119, pp. 51–52, May 2016, Accessed: 2022-12-11. [Online]. Available: http://data.europa.eu/eli/reg/2016/679/oj.

[7] ISO, "Information technology — security techniques — information security management systems — requirements," International Organization for Standardization, Tech. Rep. ISO27001:2013, Oct. 2013.

[8] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 199–212, ISBN: 9781605588940. DOI: 10.1145/1653662.1653687. [Online]. Available: https://doi.org/10.1145/1653662.1653687.

[9] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "Containerleaks: Emerging security threats of information leakages in container clouds," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 237–248. DOI: 10.1109/DSN.2017.49.

[10] S. I. Shamim, "Mitigating security attacks in kubernetes manifests for security best practices violation," ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 1689–1690, ISBN: 9781450385626. DOI: 10.1145/3468264.3473495. [Online]. Available: https://doi.org/10.1145/3468264.3473495.

[11] S. Shringarputale, P. McDaniel, K. Butler, and T. La Porta, "Co-residency attacks on containers are real," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW'20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 53–66, ISBN: 9781450380843. DOI: 10.1145/3411495.3421357. [Online]. Available: https://doi.org/10.1145/3411495.3421357.

[12] RedLock CSI Team, *Lessons from the Cryptojacking Attack at Tesla*, Accessed: 2022-12-11. [Online]. Available: https://web.archive.org/web/20180617010540/https://blog.redlock.io/cryptojacking-tesla.

[13] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," in *2021 Joint European Conference on Networks and Communications 6G Summit (EuCNC/6G Summit)*, 2021, pp. 407–412. DOI: 10.1109/EuCNC/6GSummit51104.2021.9482526.

[14] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," *IEEE Security & Privacy*, vol. 19, no. 05, pp. 46–56, Sep. 2021, ISSN: 1558-4046. DOI: 10.1109/MSEC.2021.3094726.

[15] V. V. Sarkale, P. Rad, and W. Lee, "Secure cloud container: Runtime behavior monitoring using most privileged container (mpc)," in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, 2017, pp. 351–356. DOI: 10.1109/CSCloud.2017.68.

[16] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *11th USENIX Security Symposium (USENIX Security 02)*, 2002.

[17] L. Miller, P. Mérindol, A. Gallais, and C. Pelsser, "Towards secure and leak-free workflows using microservice isolation," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, 2021, pp. 1–5. DOI: 10.1109/HPSR52026.2021.9481820.

[18] The Linux Foundation, *About the Open Container Initiative*, Accessed: 2022-12-11. [Online]. Available: https://opencontainers.org/about/overview/.

[19] Google Cloud, *What are containers?* Accessed: 2022-12-11. [Online]. Available: https://cloud.google.com/learn/what-are-containers.

[20] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019. DOI: 10.1109/ACCESS.2019.2945930.

[21] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, Bordeaux, France: Association for Computing Machinery, 2015, ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. [Online]. Available: https://doi.org/10.1145/2741948.2741964.

[22] RedHat, *etcd*, Accessed: 2022-12-11. [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-etcd#kubernetes-and-etcd.

[23] The Center for Internet Security, *The CIS Benchmarks Community Consensus Process*, Accessed: 2022-12-11. [Online]. Available: https://www.

cisecurity.org/insights/blog/the-cis-benchmarks-community-consensus-process.

[24] Cloud Native Computing Foundation, *CNCF - Graduated and incubating projects*, Accessed: 2022-12-11. [Online]. Available: https://www.cncf.io/projects/.

[25] Styra, *How Styra Maps to PCI Data Security Standard v3.2*, Accessed: 2022-12-11. [Online]. Available: https://registration.styra.com/pci_dss_v3.2.

[26] Open Policy Agent, *Documentation*, Accessed: 2022-12-11. [Online]. Available: https://www.openpolicyagent.org/docs/latest/.

[27] W. Viktorsson, C. Klein, and J. Tordsson, "Security-performance trade-offs of kubernetes container runtimes," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–4. DOI: 10.1109/MASCOTS50786.2020.9285946.

[28] Kubernetes, *Namespaces*, Accessed: 2022-12-11. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[29] Istio, *The Istio service mesh*, Accessed: 2022-12-11. [Online]. Available: https://istio.io/latest/about/service-mesh/.

[30] Buoyant, Inc, *Linkerd*, Accessed: 2022-12-11. [Online]. Available: https://linkerd.io/2.11/features/automatic-mtls/.

[31] Jason A. Donenfeld, *WireGuard: Next Generation Kernel Network Tunnel*, Accessed: 2022-12-11. [Online]. Available: https://www.wireguard.com/papers/wireguard.pdf.

A

Appendix 1

A.1 startup_script.py

```
import os
import os.path
import pwd
import sys
import logging
import argparse
import yaml
import errno

path = os.path.abspath("/home/kubemaster/.kube")

files = os.listdir(path)

def get_users():
    return pwd.getpwall()

def get_file_permissions(f: str) -> str:
    return bin(os.stat(f).st_mode)[-9:]

def get_file_uid(f: str) -> str:
    return os.stat(f).st_uid

def check_api() -> bool:
    is_valid = True
    with open("/etc/kubernetes/manifests/kube-apiserver.yaml", "r") as f:
        logging.debug("Checking API-Server manifest")
        config = yaml.full_load(f)
        container_arguments = config["spec"]["containers"][0]["command"]
        logging.debug("Checking for anonymous authentication")
        if "--anonymous-auth=false" not in container_arguments:
            logging.error("Anonymous requests are not disabled")
            is_valid = False
        logging.debug("Checking for image provenance file")
        if "--admission-control-config-file=" not in container_arguments:
```

```
35              logging.error("Image provenance file not set")
36              is_valid = False
37          logging.debug("Checking for Secrets Encryption")
38          if not any("--encryption-provider-config=" in string for string in
            ↪  container_arguments):
39              logging.error("Secrets Encryption not enabled")
40              is_valid = False
41          logging.debug("Checking for RBAC authorization")
42          logging.debug("Checking for image provenance file")
43          for item in container_arguments:
44              if item.find("--authorization-mode=") != -1:
45                  if item.find("RBAC") == -1:
46                      logging.error("RBAC is not enabled")
47                      is_valid = False
48              if item.find("--enable-admission-plugins") != -1:
49                  if item.find("ImagePolicyWebhook") == -1:
50                      logging.error("ImagePolicyWebhook admission plugin not
                        ↪  enabled")
51                      is_valid = False
52      f.close()
53      return is_valid
54
55  def kubectl_config() -> bool:
56      users = get_users()
57
58      logging.debug("Checking for permissions on kubectl config files")
59      users_ok = True
60      for user in users:
61          path = user.pw_dir
62          filepath = os.path.join(path, ".kube/config")
63
64          if not os.path.exists(filepath):
65              logging.debug(f"Skipping User {user.pw_name}")
66              continue
67          else:
68              logging.debug(f"Checking user {user.pw_name}")
69              if user.pw_uid != get_file_uid(filepath):
70                  users_ok = False
71                  logging.error(f"File {filepath} is not owned by {user.pw_name}")
72              if get_file_permissions(filepath) != "110000000":
73                  users_ok = False
74                  logging.error(f"File {filepath} has wrong permissions. Only
                    ↪  owner should have access. actual:
                    ↪  {get_file_permissions(filepath)}, expected: 110000000")
75              logging.debug(f"User {user.pw_name} is OK")
76      return users_ok
```

```
77
78      for f in files:
79          abspath = os.path.join(path, f)
80          print(abspath)
81          perm = get_file_permissions(abspath)
82          print(perm)
83
84  if __name__ == "__main__":
85      parser = argparse.ArgumentParser()
86      parser.add_argument("--log")
87      args = parser.parse_args()
88      if len(sys.argv) > 1 and args.log.upper() in ["DEBUG"]:
89          loglevel = logging.DEBUG
90      else:
91          loglevel = logging.INFO
92      if not isinstance(loglevel, int):
93          raise ValueError('Invalid log level: %s' % loglevel)
94      logging.basicConfig(level=loglevel)
95      if not kubectl_config():
96          sys.exit(1)
97      if not check_api():
98          sys.exit(1)
99      sys.exit(0)
```

B

Appendix 2

B.1 nonroot.rego

```
package kubernetes.admission.nonroot

import data.kubernetes.library.get_field_value
import data.kubernetes.library.input_containers

deny[msg] {
  allowed_operation

  # Futureproofing, just add more fields
  fields := ["runAsNonRoot"]
  field := fields[_]
  container := input_containers[_]

  not get_field_value(field, container, input.request)

  msg = sprintf("%v must have securityContext.%v set to true", [container.name,
  ↪  field])
}

allowed_operation {
  input.request.operation == "CREATE"
}

allowed_operation {
  input.request.operation == "UPDATE"
}
```