Bachelor's Thesis Computing Science

# Discovering the jump

*Extending automated Kubernetes security assessment*

Willem Medendorp
s1040947

March 28, 2023

*First supervisor/assessor:*
dr. ir. E. Poll

*Second assessor:*
prof. dr. F.W. Vaandrager

*Internship supervisor:*
G. Smelt

Radboud University

## Abstract

*Kubernetes* is the de facto standard when in comes to container orchestration in the cloud. Large services are build upon it and many users rely on it for their everyday activities. In the current infrastructure landscape every scalable application is split into containers. A container for the database, a container for the router, a container for the application etc. These containers are spread out over many servers, these servers combined are called a cluster. We can manage and remove complexity from complicated clusters by making use of Kubernetes.

Due to its prevalence scrutinizing the security is all the more important. Can we assess whether a Kubernetes is secure? Creating a cluster becomes easier with Kubernetes, however it remains a challenging task where security oversights may happen. Having the wrong permission in the right place could allow and adversary to escalate its privileges. This fact makes it important to assess all permissions that are being granted assuring that permission are minimized. Manually doing this requires in-dept security knowledge and a lot of time, so automated assessment tools were developed. In this research we looked into one such tool *Kubescape* and try to answer the question: *"Can we extended Kubescape to discover trampoline pods?"*. Trampoline pods are a new privilege escalation path within a cluster. We achieve this by extracting trampoline pod specific rules from an existing tool namely *RBAC Police* and implementing these in a new *Kubescape framework*. We implemented this successfully for 1 new rule and adapted 6 existing rules. Furthermore, we tested our new rule against a vulnerable cluster where it was able to detect the trampoline pod. Concluding, we were able to extend *Kubescape* with a new rule and adapt 6 existing rules with the possibility to add more rules in the future.

# Contents

**1 Introduction** **4**

**2 Kubernetes background** **6**
- 2.1 Containers . . . 6
  - 2.1.1 Docker . . . 7
- 2.2 Kubernetes . . . 8
  - 2.2.1 Control plane . . . 9
  - 2.2.2 Data plane components . . . 10
  - 2.2.3 Objects . . . 11
  - 2.2.4 Role Based Access Control . . . 13
- 2.3 Related technologies . . . 17

**3 Security background** **19**
- 3.1 Steps in a typical attack . . . 19
- 3.2 Initial access to pods . . . 20
- 3.3 Access to the node . . . 20
  - 3.3.1 Container escapes . . . 21
  - 3.3.2 Privileged container . . . 22
  - 3.3.3 Kubelet API . . . 22
- 3.4 Becoming cluster admin . . . 22

**4 Trampoline pods** **24**
- 4.1 Origin . . . 24
- 4.2 Principles of a trampoline pod . . . 25
- 4.3 Powerful permissions and classes of trampoline pods . . . 26
  - 4.3.1 Rule abbreviations . . . 26
  - 4.3.2 Manipulate Authentication/Authorization . . . 26
  - 4.3.3 Acquire Tokens . . . 27
  - 4.3.4 Remote Code Execution . . . 28
  - 4.3.5 Steal Pods . . . 28
  - 4.3.6 Meddler-in-the-Middle . . . 29

**5 Automated assessment of Kubernetes permissions** **30**
5.1 Assessing Kubernetes . . . . . 30
5.2 Kubescape . . . . . 31
5.2.1 What is Kubescape? . . . . . 31
5.2.2 Kubescape architecture . . . . . 32
5.2.3 Structure of the regolibrary . . . . . 33
5.2.4 Running Kubescape . . . . . 33
5.3 RBAC Police . . . . . 34
5.3.1 What is RBAC Police? . . . . . 35
5.3.2 Running RBAC Police . . . . . 35
5.3.3 Structure of RBAC Police . . . . . 36

**6 Kubescape extension: discovering trampoline pods** **37**
6.1 Approach . . . . . 37
6.2 Finding equivalent rules . . . . . 38
6.3 Creating a new Kubescape framework . . . . . 40

**7 Testing the extension** **43**
7.1 Testing of new rule with test runner . . . . . 43
7.2 Testing it with Kubernetes . . . . . 44
7.2.1 Setting up a vulnerable cluster . . . . . 44
7.2.2 Testing with RBAC Police . . . . . 46
7.2.3 Testing trampoline pod framework . . . . . 46

**8 Future Work** **48**
8.1 Finding more powerful permissions . . . . . 48
8.2 Other Kubernetes security assessment tools . . . . . 48
8.3 Extension for all uncovered RBAC police rules . . . . . 49
8.4 Highlight the affected pod in the results . . . . . 49

**9 Conclusions** **50**

**Appendix** **54**
A.1 Kubescape output . . . . . 55
A.2 Example of Kubescape Rego . . . . . 55
A.3 Example of RBAC police REGO . . . . . 57
A.4 RBAC Police eval . . . . . 58
A.5 RBAC Police expand . . . . . 59
A.6 Comparison of rules that are equivalent . . . . . 61
A.7 Comparison of rules that are partially equivalent . . . . . 63
A.8 Token request rule . . . . . 64
A.9 Token request metadata . . . . . 65
A.10 Create pod control . . . . . 67
A.11 Token request control . . . . . 67

A.12 Trampoline pod framework . . . . . . . . . . . . . . . . . . . . . 68
A.13 Test runner for roleBinding . . . . . . . . . . . . . . . . . . . . . 68
A.14 Test runner for roleBinding . . . . . . . . . . . . . . . . . . . . . 70
A.15 RBAC Police evaluate token request . . . . . . . . . . . . . . . 74
A.16 Kubescape trampoline pod framework . . . . . . . . . . . . . . 75

# Chapter 1

# Introduction

*Kubernetes*[1] originally developed by google and open sourced in 2014 and adopted by the *Cloud Native Computing Foundation* (CNCF)[2] which is part of the Linux Foundation. With cloud becoming popular Kubernetes quickly became the preferred container orchestration platform. Kubernetes is responsible for 59% of the global production servers according to a survey by CNCF [6]. The survey also concluded that for the respondents not using Kubernetes in production yet, almost all were evaluating or considering to use Kubernetes in the future. Kubernetes clearly is an influential and important technology making it very interesting for cybersecurity research. If a critical exploit were to be found thousands of servers would be effected. It would then not result in one application being compromised, but the entire production environment of many servers.

Luckily there are many bright minds securing Kubernetes by developing good security practices in work by Shamim [9], Shamim, Bhuiyan [4] and Alawneh, Abbadi [10]. However, it is still up to the admin to set up and deploy Kubernetes correctly. Kubernetes is complex and has many configurations. Misconfigurations can have catastrophic effects by providing full access to a malicious user. Errors are inevitable, but they should be discovered by automated assessments.

Kubernetes uses many terms for all its objects. To clarify the terms we use `mono font` for Kubernetes objects, for example `ServiceAccount`. The terms: cluster, node, pod and trampoline pod will be omitted from the mono fonts as they are used frequently and thus hamper readability. For all objects relating to the tools we use *italic font*, for example *Kubescape*. With these notation conventions we can now discuss Kubernetes in more detail.

*Kubescape*[3] is one such tool that can discover these errors. *Kubescape* is open-source tool assessing security compliance based on different frame-

[1]https://kubernetes.io/

[2]`https://www.cncf.io/`

[3]`https://github.com/kubescape/kubescape`

works. And provides vulnerability scanning of known weaknesses. For our research we will only look into the vulnerability scanning aspects of *Kubescape*. It assesses all configuration files detecting misconfigurations according to three recognized frameworks: NSA-CISA [12], CIS BENCHMARK [13] and MITRE ATT&CK [17]. *Kubescape* can be run during security assessment from the command line. But could also be integrated into the continuous integration/continuous development pipeline. For this research we will only look at the former.

Not all misconfigurations are however equal, some result in an extra exposed port on a server whilst others result in complete compromise. A new class of misconfigurations have been identified that could lead to the entire Kubernetes cluster being compromised. This class of misconfigurations create so-called trampoline pods. In chapter 4 we will go in-dept on trampoline pods. At the time of writing *Kubescape* is not able to detect this class of misconfigurations and also does not have plans to add this is the near future.

We research a possible extension of *Kubescape* to be able to detect trampoline pods. We do this by first discussing the relevant background on containers and Kubernetes in chapter 2. This chapter will also include details about *Role-Based Access Control* (RBAC) the main type of misconfigurations we will be looking for. With those topics introduced we discuss the relevant services vulnerable to attack in chapter 3. In which we also discuss some basic attack strategies. With this basic Kubernetes security understanding we can dive deeper in certain RBAC permission and the resulting trampoline pods in chapter 4. With some basic Kubernetes security understanding in chapter 5 we look into two tools that aid in security assessments: *Kubescape* and *RBAC Police*.

After we have the background on Kubernetes (ch. 2) and its security (ch. 3), trampoline pods (ch. 4) and automated assessment (ch. 5) we can design an extension for *Kubescape* framework in chapter 6. This extension is a new *Kubescape framework* that consist out of *rules* that assess RBAC permissions of all Kubernetes objects in a cluster. These *rules* will fail when a pod has trampoline pod permissions assigned to it. The specification of these trampoline pod *rules* are extracted from *RBAC police*. We extend *Kubescape* instead of just using *RBAC Police* because *Kubescape* is a much more capable tool with a broader audience and better usability. To extend *Kubescape* we bring possibility to discover trampoline pods to even more users.

To verify our implementation we, in chapter 7, test this extension against a Kubernetes clusters with various pods to verify if the extension is able to detect trampoline pods. In chapter 8 we discuss future work and chapter 9 we conclude our research and discuss the results.

# Chapter 2

# Kubernetes background

In this chapter we will discuss the relevant background information to understand the workings of Kubernetes. Not everything mentioned here is needed to understand the principles of this research. To follow this research a grasp of containerization is needed, see section 2.1. To provide context we will highlight some objects inside Kubernetes. However, these are not crucial to understand the later extension in chapter 6. The core of the research relies on understanding of RBAC with Kubernetes, for this see 2.2.4.

This chapter will start by the fundamental building block: the `container`. We will then discuss Kubernetes as from an architecture point of view, such that we can discuss the different Kubernetes components more in-dept afterwards. Then we will highlight different Kubernetes objects which are relevant for exploitation. To finally conclude with Role Based Access Control which is relevant for discovering trampoline pods. For even more in-dept information the online documentation[1] can be viewed.

## 2.1 Containers

What are containers and why do we use and need them? A common problem in development environments is that code works on the machine of Alice, but fails on the machine of Bob due to dependency problems. Bob must patch his system costing him time and effort. There is an easier way to do this, Alice just sends her entire system, then Bob can run it without any configuration. This is the basic essence of containerization. Package everything that an application needs to run: OS environment; libraries; packages; binaries and code in to a container and ship that.

This might look familiar to shipping virtual machines (VM). However, one would soon discover the immense performance cost of running multiple VM's. Containers do not rely on a guest operating system and hypervisor, they operate on a *container runtime* and use cgroups and namespaces, which

[1]`https://kubernetes.io/docs/concepts/overview/`

are not further relevant. The container runtime creates an environment such that from the application view it looks like a full system whilst in reality it is just a small subset. In Figure 2.1 the difference between containers and VM is showcased with Docker as the container runtime.

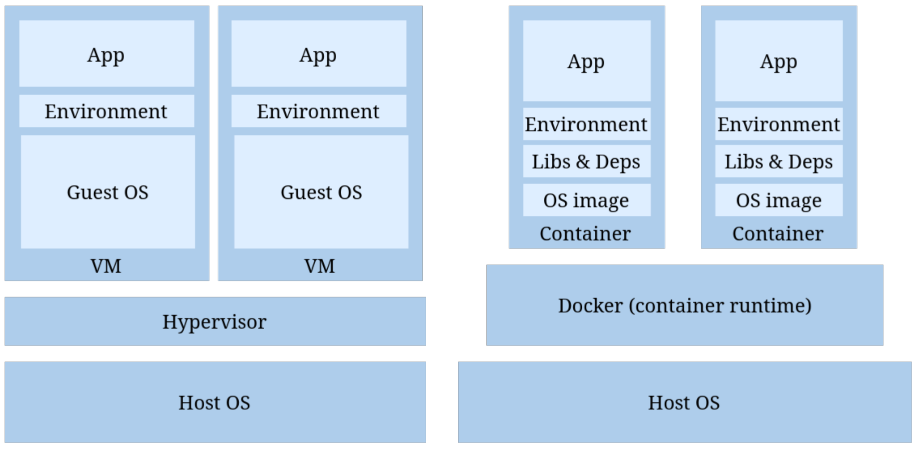

Figure 2.1: Containers compared to VMs

### 2.1.1 Docker

For this research the internal workings of containers are not relevant. However, we will look at how containers are constructed. For this we will use Docker[2] as an example. The contents of a docker container are defined in a *Dockerfile*. This file contains instructions on how to build and run the container. We can take a look at the `Dockerfile` for a simple Apache server in figure 2.2. The `Dockerfile` starts with the `FROM` instruction this defines

```
FROM ubuntu
RUN apt update
RUN apt install {y apache2
RUN apt install {y apache2-utils
RUN apt clean
EXPOSE 80
CMD [\apache2ctl", \-D", \FOREGROUND"]
```

Figure 2.2: Example of Dockerfile

which *image* is used as a basis or `scratch` for no basis. `Images` are read-only instruction to build a container, or can be seen as predefined `Dockerfiles`.

[2]`https://www.docker.com/`

`Images` are stored in a registry, these can be public or private. Most common applications and OS's have `images` available. With `RUN` an arbitrary command inside the container is ran. `EXPOSE` defines what port to open and finally `CMD` sets the default command when the container is being started. The `Dockerfile` can be used to build a container once build they can be run. Such a `Dockerfile` can also be build into an `image` such that it can be added to a registry. Everyone with access to that registry can easily build and deploy the container.

Apart from use in development many production environments also use containers by splitting up one application into many services. This is the so-called microservice architecture [8]. The switch from monolith applications to microservices improved deployment speed and made scaling easier. If for example more front-ends capacity is needed more containers can be deployed quickly. Without also needing to deploy more back-end containers. Managing this all by hand on great scale would be infeasible so container orchestration was born.

## 2.2 Kubernetes

As mentioned before Kubernetes is an open-source container orchestration system for automating the deployment, scaling and management of containers. It has its origins at Google but is now maintained by the Cloud Native Computing Foundation (CNCF)[3]. Kubernetes is not the only container orchestrations system alternative include: Docker Swarm, Apache Mesos, and HashiCorp Nomand. However, Kubernetes remains the de facto enterprise standard. Kubernetes can be run on-premise, in the cloud or in a combination also known as 'cloud agnostic'. For organizations that do not have the knowledge to set up a cluster, cloud providers such as Microsoft, Google and Amazon also provide Kubernetes as a service, referred to as PaaS (Platform as a service). These services are called Microsoft AKS, Google GKE and Amazon AWS EKS. Here providers manage the *control plane* to take away complexity. The control plane is the brains of a cluster and manages all other components.

We will now have a look at the architecture of a Kubernetes cluster for this Figure 2.3 provides a practical overview. The largest unit is the cluster itself which is comprised out of *nodes*. There are two types of nodes: master and worker. Master nodes are part of the control plane in a small cluster this is just one node. In a large cluster this can be many nodes for redundancy and speed. The components of the control plane will be covered in section 2.2.1. Worker nodes are part of the data plane there usually many of these in a cluster. More about the worker node components in section 2.2.2. Going one step down we get *pods* they run inside nodes see Figure

[3]`https://www.cncf.io/`

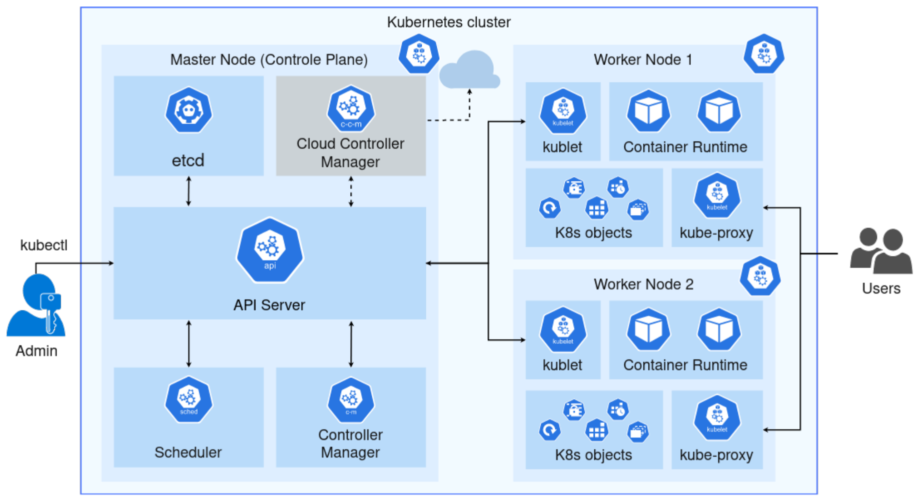

Figure 2.3: Kubernetes Architecture

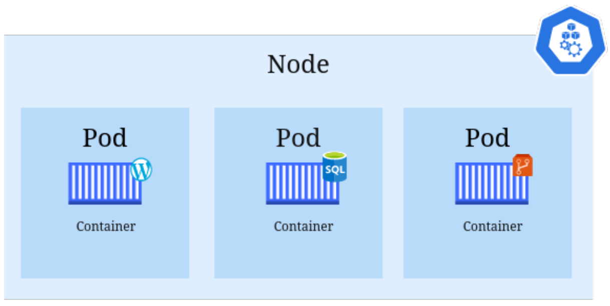

Figure 2.4: Node, Pods, Container build up

2.4. Pods usually contain one container and thus one service/application. It is however possible to have multiple containers in one pod, although this is not recommended. In Figure 2.3 we can see an overview of all the components and `objects` in a cluster. In section 2.2.3 we will discuss the relevant `objects` in more detail.

### 2.2.1 Control plane

First all components inside the control plane

- **kube-apiserver**
  The front-end of the Kubernetes control plane. It provides access to the `Kubernetes API` to the outside and communicates internally with all the nodes. `Kube-apiserver` can only be access via correct client certificates or JWTs (JSON Web Tokens)

- **etcd**

A key value store for all cluster related data. This includes the configuration of the cluster and permissions inside the cluster. Typically this is located on master nodes but sometimes also hosted externally for redundancy. Configuration files are written in `YAML` which is a human-readable data-serialization language.

- **kube-controller-manager**
  Runs the control loop processes which is a non-terminating loop that regulates the state of the cluster. It tries to match the desired state by making changes to the current one. A part of the kube-controller-manager is the *node controller* which is responsible for noticing and responding when nodes go down.

- **kube-scheduler**
  Watches for newly created pods with no assigned node and selects a node for them to run on. The scheduler can take many factors into account for example: resource requirements, hardware/software/policy constraints, affinity, data locality and inter-workload interference. Such that a pod can run optimally and that the cluster runs as efficiently as possible.

- **cloud-controller-manager**
  Embeds cloud-specific control logic by linking it to the cloud provider's API for control and load balancers. For on-premise clusters these are not needed.

### 2.2.2 Data plane components

Every worker node also contains some fixed components.

- **Container Runtime**
  Responsible for running the containers. This is done via application that implement the Kubernetes Container Runtime Interface (CRI) for example: `Docker`, `containerd` and `CRI-O`

- **kubelet**
  Agent that facilitates communication from outside the node. It does this by receiving instructions from API server after being selected by the Scheduler. It interacts with the `Container Runtime` and ensures that the desired containers are running on that node.

- **kube-proxy**
  Network proxy that takes care of (virtual) routing. The routing is based on iptable rules. It is also responsible for enforcing *NetworkPolicy*, later more about this.

- **CoreDNS**
  Serves DNS records to other components. This is needed when Kubernetes is provided via PaaS (Platform as a Service). It can kook up domains such as: `wordpress.app.svc.cluster.local`.
- **Pods**
  As mentioned before the smallest unit in which `containers` run. A pod is a group of one or more `containers` and *volumes*. A `volume` is a specified and assigned storage object.

### 2.2.3 Objects

Now that we have all components highlighted we can cover different `objects` within Kubernetes. These `objects` are passed between components and facilitate the workings and structure of a Kubernetes cluster

- **ReplicaSet**
  A collection of identical pods these are created based on the template. They are often managed by `deployments` and are commonly used for stateless data such as nginx web root. An example of a `ReplicaSet` would look like this:

```
apiVersion: apps/v1
kind: ReplicaSet
metdadata:
    name: frontend
    labels:
        app: guestbook
        tier: frontends
spec:
    replicas:3
    selector:
        matchLabels:
            tier: frontend
        template:
            metadata:
                labels:
                    tier:   frontend
    spec:
        container:
        - name: nginx
            image: gcr.o/google_samples/nginx-frontend:v3
```

- **Deployment**
  Is a template for pods to be deployed by the scheduler. It also contains

a `ReplicaSet` that scales the pods. Deployment's can be seen as the actual applications in the cluster. A `deployment` would have the following structure.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels: nginx
spec:
    replicas: 3
    selector:
        matchLabels:
            app: nginx
    template:
        metadata:
            lables:
                app: nginx
        spec:
            containers:
            - name: nginx
                image: nginx:1.14.2
                ports:
                - containerPort: 80
```

- **Service**
  Is an abstract way to expose a set of pods. It creates a DNS name, IP and port for the pods that are behind it. The pods behind a service can be dynamically be created and removed without external applications having to rediscover the IP addresses etc. There are four types of Services:
  - *ClusterIP*: It exposes the Service on the cluster-internal IP, such that it is only reachable from within the cluster
  - *NodePort*: Exposes the Service op the Node IP at a static port
  - *LoadBalancer* Exposed the services externally using the load balancer of the cloud provider.
  - *ExternalName* Maps a Service to contents of a name field such as `foo.bar.example.com`
- **Namespace**
  Is a logical grouping of resources a cluster it however does not provide separation. Separation must be achieved by `NetworkPolicy` to

restrict network traffic to resources within a cluster. Namespace primary usage is to make managing resources easier, for example the same app one in production and one in development can be referenced by `app.prod.svc.cluster.local` and `app.dev.svc.cluster.local` respectively.

- **DaemonSet**
  Similar to a ReplicaSet, however this set runs at least one pod on every node in the cluster. To can be used for ingress controllers and log aggregator services you want to run on every node.

- **Secret**
  Kubernetes uses secrets to authenticate, these are base64 encoded key-value pairs. Secrets are read-only objects and can be considered as static content. They are mounted within pods as files or set as environment variables. A ServiceAccount token looks like this:

```
apiVersion: v1
data:
 ca.crt: LS0tLS1CRUdJTiBDRVJUSU.......
 namespace: ZGVmYXVsdA==
 token: ZXlKaGJHY2lPaUpTVXpJMU5pSXNJblI1Y.......
kind: Secret
metadata:
 annotations:
   kubernetes.io/service-account.name: default
   kubernetes.io/service-account.uid: df441c69-f4ba-11e6-8157-525400225
 creationTimestamp: 2017-02-17T02:43:33Z
 name: default-token-2mfqv
 namespace: default
 resourceVersion: "37"
 selfLink: /api/v1/namespaces/default/secrets/default-token-2mfqv
type: kubernetes.io/service-account-token
```

### 2.2.4 Role Based Access Control

In this section we will look into RBAC. By first covering some general literature about RBAC. Then we look at what RBAC applies to in Kubernetes: `resources` and `verbs`. Then how these permissions are assigned with the `Role` and `RoleBindings`.

**General literature about RBAC**

RBAC dates at least back to the 1970's where it was implemented in limited forms of access constrains based on the user's role. There was not any standard model nor standard implementation. A general-purpose Role Based Access Control model was proposed in 1992 [1]. Here it was created out of necessity to grant more fine-grained access to resources than the at the time used access control models developed by the Department of Defense.

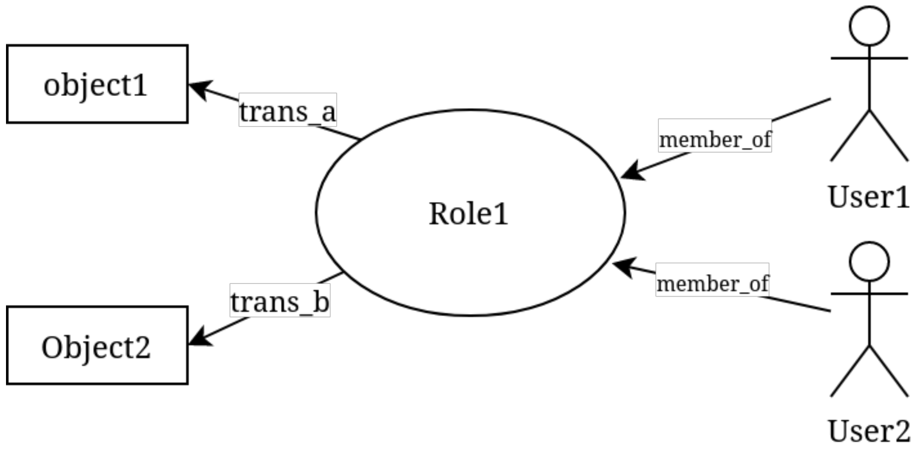

Figure 2.5: Role relationships

From [1] we can give a simple formal description, in terms of sets and relation, of RBAC.

For each subject the set of roles they are actively using:

$$AR(s : subject) = \{\text{active roles for } s\}$$

Each subject may be authorized to use multiple roles

$$RA(s : subject) = \{\text{authorized roles for } s\}$$

Each role may be authorized for multiple transaction:

$$TA(r : role) = \{\text{authorized transactions for } r\}$$

Subjects may execute transactions. The predicate $exec(s, t)$ is true if and only if subject $s$ can execute transaction $t$ at the current time.

$$exec(s : subject, t : transaction) = true \text{ iff } s \text{ can execute } t$$

With these basic sets we can define three rules:

1. Role assignment: A subject s can execute a transaction iff s has a role

$$\forall s : subject, t : transaction, (exec(s,t) \Rightarrow AR(s) \neq \emptyset)$$

2. Role authorization: A subject s active role must be authorized for s

$$\forall s : subject, (AR(s) \subseteq RA(s))$$

3. Transaction authorization: A subject s can execute transaction t iff t is authorized by s's active role

$$\forall s : subject, t : transaction, (exec(s,t) \Rightarrow t \in TA(AR(s)))$$

Rule (1) states that all subject must have a role to execute a transaction. Rule (2) states that a subject can use roles it that they are authorized for. Rule (3) states that a subject can only execute transactions that are authorized by their active role.

### RBAC in Kubernetes

Within Kubernetes the same structure and rules apply as in the general literature. However, there are some differences. Within Kubernetes transactions are split into the action: `verb` and the object: `resource`. The `verb` is the action that is performed on the `resource`. For example, the `verb` `create` is used to create a `resource`. The `resource` is the object that is acted upon. For example, the `resource` of `ServiceAccount` can be created. In Kubernetes RBAC is handled by the `rbac.autorization.k8s.io` API group. This makes it possible to dynamically configure policies through the Kubernetes API. We will cover 7 objects that are used to configure RBAC with in Kubernetes: `ServiceAccounts`, `Users`, `Groups`, `Resources`, `Verbs`, `Roles`, `RoleBindings`.

### Subjects

Within Kubernetes there are three types of subjects: `ServiceAccounts`, `Users` and `Groups`. `ServiceAccounts` are used to grand permissions to pods and `deployments`, `users` are used to represent a person and `groups` are used to represent a group of people. These are used to grant permissions to multiple people at once. In this thesis we will only use `ServiceAccounts`.

### Resources and verbs

The definition of a `resource` from the official documentation: *"A resource is an endpoint in the Kubernetes API that stores a collection of API objects of* *a certain kind; for example, the built-in pods resource contains a collection of Pod objects."*[4]. This description is still relatively vague and encompasses a lot of objects. `Resources` can be seen as everything that can be interacted with. Via the API these interactions can be to create, remove, update, read an object. These interactions are also codified in `verbs` these are: `create`, `get`, `list`, `watch`, `update`, `patch`, `delete` and `delete collection`. A RBAC rule on the `resource "pods"` would look as following:

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

The `resources` and `verbs` can also be defined by wildcards. Instead of listing them all out we can specify `verbs:[*]`. With `""` we specify the core API group.

### Role

A `Role` or `CulsterRole` contains a set of rules that defines that `roles` permissions. All permissions are purely additive, so there are no deny rules only allow rules. This means by default a new role can not do anything. A `Role` is bound to a certain `namespace`, this must be specified upon creation. In contrast a `ClusterRole` is not namespaced. This is useful to define permissions over multiple namespaces or to define permissions on cluster-scoped resources. Here is an example of a `role` in the `"default" namespace` that can be used to grant read access to pods.

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

A `ClusterRole` would be similar only the namespace would be omitted.

### RoleBinding

To bind roles to `users`, `groups` or `ServicesAccounts` we need `RoleBindings`. A `RoleBinding` grants the permissions of a `Role` the subject it is bound to.

[4]https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

A `RoleBinding` contains a list of subjects for which the `Role` applies. They are specific to a `namespace` whereas a `ClusterRoleBinding` grants that access cluster-wide. A `RoleBinding` can reference any `Role` in its `namespace`. However, it can also be bound to a `ClusterRole` which then gets bound to the `namespace` of the `RoleBinding`.

With `Roles` and `RoleBindings` we only need subjects to bind them to. For user's we have `UserAccounts` these authenticate to the Kubernetes API via a valid token. In most cases the identity of the user is managed externally for example by Azure AD. For pods to authenticate to the Kubernetes API we don't want to use `UserAccounts`, for this we have `ServicesAccounts`. The default `ServiceAccount` is assigned to a pod when no other is specified. The default `ServiceAccount` has no permissions. If we want to grant certain rights to a pod we create a special `ServiceAccount` for it. We then bind a `Role` to that account via a `RoleBinding`. This `ServiceAccount` can now be specified in the `PodSpec`, upon creation of the pod Kubernetes mounts the `ServiceAccount` configuration inside the pod. When this pod communicates to the Kubernetes API it uses the token in the `ServiceAccount` to authenticate. This token is readable for every program running in that pod. The node on which the pod is running also has access to the `ServiceAccounts` of the pods inside it. Grating someone one with access to the node access to all `ServiceAccounts` contained in it. This creates security problems when adversary has gained access to a node, in chapter 4 on trampoline pods we will delve more into this.

## 2.3 Related technologies

There a few technologies that are not directly part of Kubernetes itself but are commonly used. These technologies are mentioned to provide a better understanding of the context in which Kubernetes is used.

### kubectl

Is the command line interface to deploy application, inspect and manage cluster resource and view logs. `Kubectl` check the environment variables for to correct cluster and certificate to use. It uses the locally stored `Kubeconfig` to connect and authenticate to a cluster. `kubectl` is the primary way to interact with a cluster via the command line.

### Cloud native landscape

Kubernetes is not a stand-alone technology, it is part of a bigger cloud native landscape. The CNCF also provides a formal definition on their map[5]. For the size of the landscape a copy can be seen in 2.6, for details visit the source. The landscape consist out of many projects spread over different categories. These categories include: Scheduling & Orchestration, Databases, Cloud native storage, Logging, CI&CD and many more. Some of the projects that part of the cloud native landscape are:`Helm` is a package manager for Kubernetes; `Prometheus` is a monitoring system; `Argo` is a CI&CD system. These other project are commonly included in Kubernetes as add-ons. However, relying on external projects and not doing proper security assessment on those projects can pose risk to a cluster. An external project might gather too strong permission which could be used for privilege escalation, this forms the basis for trampoline pods, more about this is chapter 4.

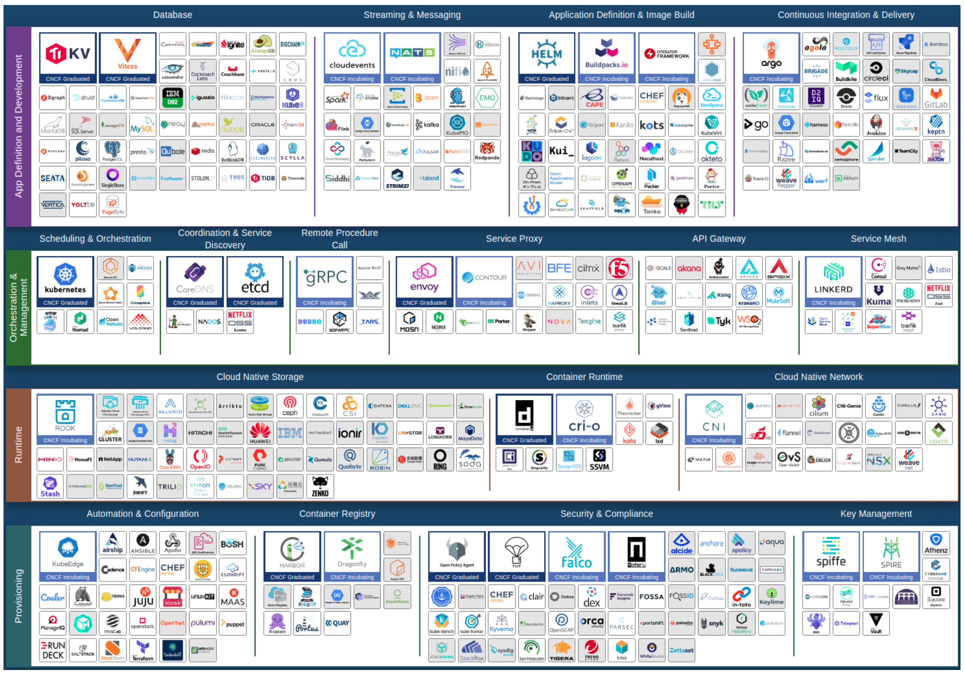

Figure 2.6: Cloud native landscape Source: `https://landscape.cncf.io/`

[5]`https://landscape.cncf.io/`

# Chapter 3

# Security background

Now that we have established a basic understanding of Kubernetes we can look at its security aspects. This chapter will cover a general privilege escalation path whilst showing the relevant security concepts. Not all security concepts are needed to understand this research, they however provide context for the uninitiated. The steps in a typical attack are highlight in section 3.1.

The remaining of the chapter follows a bottom up approach starting at the initial access and ending at the cluster admin. For this path we will use the MITRE attack framework [17] to provide a generalized naming convention. The level from the framework that we will include are: *Initial Access*, *Privilege Escalation to host* and *Privilege Escalation to admin*. These levels translated to Kubernetes terms, initial access starts at the pod see section 3.2. One up we go to escape into the node see section 3.3 and finally elevate to cluster admin see section 3.4 and chapter 4.

## 3.1 Steps in a typical attack

1) An adversary can gain access to a `container` by compromising the application it is containing. 2) It can then use a *container escape* to leave the `container` and pod and get access to the node. 3) With access to the node it also has access to the files of all other pods on that node. 4) As such it has access to the `ServiceAccounts` of the other pods which it could use to escalate its privileges. Pods that have such powerful permissions that they allow for privilege escalation we call trampoline pods. In figure 3.1 these steps are numbered and visualized. We will use these numbers to indicate the step in the attack path.

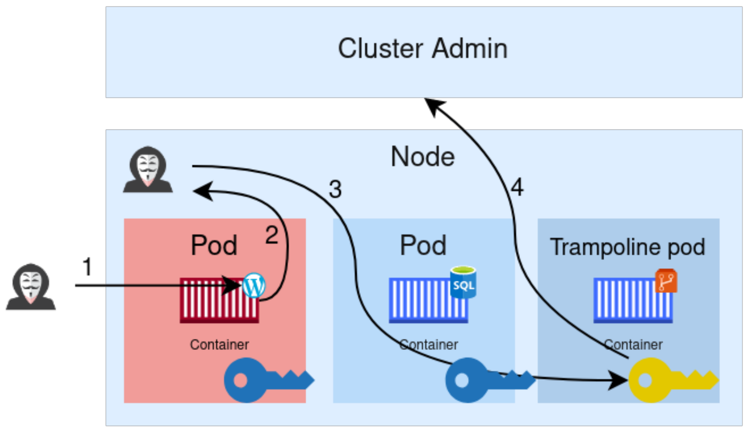

Figure 3.1: Escalation path

## 3.2 Initial access to pods

Step 1 of most attacks is getting accessing to a pod. This is commonly achieved by compromising the application running inside it. Applications can differ a lot so exploitation of it also differs. An application could be vulnerable allowing an adversary to exploit it to gain RCE. This is *Exploit Public-Facing Application* from the MITRE framework [17].

A more advanced way to get access to a pod is by using a *compromised image*. Here an adversary has deployed a malicious pod via compromising an `image` in a registry. This can be either in a Kubernetes registry and Docker registry. Such images can also be introduced during the CI/CD pipeline[14]. The compromised image would include a backdoor or exfiltrate secrets. When a new pod is deployed it pulls this compromised `images` and uses it as a base. With this the adversary has directly access to that pod. This would be a type of supply chain attack. This can either be done by infiltrating a private registry or by mimic a popular `image` in a public registry [3]. In the MITRE framework [17] this would be the *External Remote Services* technique. The resulting state is visualized in figure 3.2.

## 3.3 Access to the node

Step 2 is expanding the position because for most adversary one pod isn't that interesting. So the goal is to escape the pod to get access to `Node`. Such that has access to all pods inside it and possibly has a path to escalate to cluster admin. The `containers` that are inside pods run on the `Container Runtime` of the node. So if we escape the `container` by exploiting the runtime we also escape the pod. So a `container` escape is also a pod escape we will cover this in section 3.3.1. `Container` escapes are part of the *Escape*

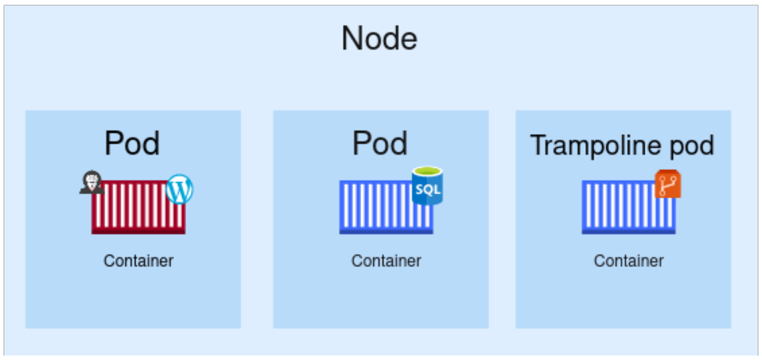

Figure 3.2: Initial access

*to Host* technique in the MITRE framework [17]. Apart from privilege escalation it is also possible to directly access a node via exposed services like the `kubelet API`, see 3.3.3. In the MITRE framework this covered in the *External Remote Services* technique.

The final way to access a node is via valid credentials. In the MITRE framework this is technique is called *Valid accounts*. We will not cover this since its impact is trivial and execution mostly happens outside the domain of Kubernetes.

### 3.3.1 Container escapes

Although for this research we do not need to go in dept to `container` security it is good to have a basic understanding to be able to form the bigger picture. `Container` escapes are as the name suggests escapes out of a `container` to get access to the host. `Container` escapes are more easily realized then VM escapes. Because `containers` don't actually have their own OS but just use a namespaced and partitioned part of the hosts. `Containers` make heavy use of Linux kernel features, thus making hosts vulnerable to kernel exploits. An example of this is CVE-2021-22555 [7] which exploits a bug in the network interface. However, for most exploits the container must be granted some *capabilities*. Capabilities are privileges that a `container` has on the host. For example CVE-2021-22555 needs the capability `CAP_NET_ADMIN` to function. This capability provides a `container` rights to manage the network of the host, which might be needed for an application that provides routing. An different type of `container` escapes rely on misconfigurations and overzealous privileges. For example the case that a `container` is set up as a *privileged container* then it has all capabilities. From which an attacker could easily mount the hosts file system or spawn new malicious `containers`. For more examples and specifics Carlo Polop provides a great online resource [16]. Figure 3.3 visualizes the state after a container escape.

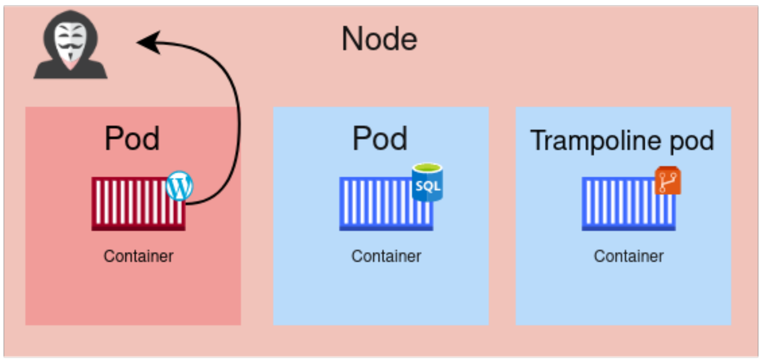

Figure 3.3: Access to node

### 3.3.2 Privileged container

Having code execution in a `container` does not directly mean that an adversary can escape the `container`. However if the adversary is in a privileged `container` escaping it becomes trivial. Privileged `containers` have all `capabilities` assigned to it increasing the attack surface drastically. In deployment privileged `containers` should never exist.

### 3.3.3 Kubelet API

Step 2 can also be achieved without step 1 as a prerequisite. A different way to get access to a node is via the `Kubelet API`. This can be done directly if for example `Kubelet` is exposed to the internet. Or indirectly if an adversary is able to use Cross Site Request Forgery to access the `Kubelet API`. The `Kubelet API` uses by default no authentication. Because no authentication is required a simple exposed port can lead to easy code execution. `Kubelet` takes instruction from the `Kubernetes API`. These instructions include which pods to `deploy` or `delete`. `Kubelet` also keeps track of the health of the pods and does logging. This can all be accessed by an adversary if it has access to the `Kubelet API`. It could thus spawn its own rouge `Deployment` with a RCE or execute command directly in a pod. With the tool *Kubeletctl*[1] penetration testers can attach to a container to extract configs, list pods, read logs or gain a shell. See figure 3.4 for a visual representation of this path.

## 3.4 Becoming cluster admin

Finally at the top of the hierarchy is cluster admin, step 4 in the core concepts. Just like access to pods and nodes there are various ways to

[1]`https://github.com/cyberark/kubeletctl`

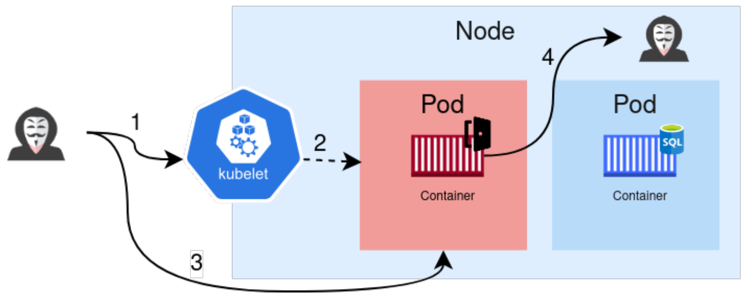

Figure 3.4: Access to node via kubelet

achieve this. Just like accessing nodes *External Remote Services* technique can pose an issue to cluster security. For example via an exposed dashboard. The dashboard allows for visual interaction with the Kubernetes cluster. It could be setup unauthenticated or be used with stolen credentials. The *Valid accounts* technique also applies for cluster admin, if an adversary for example hijacks the computer of a developer with cluster admin config installed. All these attacks rely on aspects outside the scope of Kubernetes.

Within Kubernetes there are also ways to becomes cluster admin. Within Kubernetes the aim is to get an admin token. In the MITRE attack framework this is the *Steal Application Access Token* technique. To achieve this an adversary must be able to misuse strong permissions that are being granted in its environment. Examples of this include: ability to list all `secrets` or to create a pod with the admin `secret`. A summary of the 10 most prevalent attacks can be found online [5]. All these attack come down to finding and misusing certain RBAC permissions. However, the chance of this is quite unlikely since most admins understand not the give powerful permissions to externally facing pods. When an adversary has escaped to pod to the node it does get more probable that an adversary has access to strong permissions. Because then the adversary does not only have the permissions of it original pod but the permissions of all pods on the node. Thus, these pods allow an adversary to jump to cluster admin and for that reason are coined trampoline pods. Since programmatically discovering trampoline pods is the main focus of this thesis we will go more in depth in chapter 4.

# Chapter 4

# Trampoline pods

In this chapter we go more in dept into attack paths that use trampoline pods. Trampoline pods are a new relatively unknown Kubernetes security concept. As discussed in section 3.4 Trampoline pods describe pods that have a collection of powerful permissions such that the pod can be exploited to escalate privileges. This makes these pods interesting because if an adversary has control over a node it could use them to gain cluster admin rights. In this chapter we will first go in to their origin in section 4.1. Then describe what the basic principles are in section 4.2 to finally go into the classes of trampoline pods that exist in section 4.3.

## 4.1 Origin

The term trampoline pod was first coined by Avrahami and Ben Hai at Palo Alto networks in their whitepaper on how many `DaemonSets` contain trampoline pods [11]. In this section we will go over their findings.

Their hypothesis was *"Does container escape equal Cluster admin?"*. Container escapes are very probable because containers providing imperfect security. It is then the question if the rest of cluster is secure enough to contain the adversary. Their research found that when an adversary had code execution in a node it was possible to exploit other pods to gain cluster admin. These other pods where thus coined trampoline pods, because they allowed and adversary to jump to cluster admin. These pods had such powerful permissions that an adversary could misuse them.

Most admins have a good grip on what permissions pods have and makes it thus unlikely that pods have excessively strong permissions. However, the permissions that external add-ons take with them usually are not reviewed, since these are installed using package managers. These add-ons provide a large permission's oversight. It might be unlikely that an adversary is on the same node as such a highly privileged add-on pod. However, when these are part of the `DaemonSet` it is no longer a game of chance.

With their research they identified five classes of trampoline pods which can be used in different attack paths, more detail in section 4.3. These classes exist out of a set of powerful permissions that single pod could have. With all these permissions defined they looked at popular add-ons to see how many included trampoline pods. In the beginning of the research period 75% of `Daemonsets` included trampoline pods. After their research and disclosure 25% still had powerful trampoline pods. With their research they achieved 2/3 reduction of trampoline pods in popular add-on `DeamonSets`. This was achieved by contacting all developers over their overzealous permission allocations. They also released a tool called *RBAC Police*, which is able to assess whether trampoline pods exists in a cluster in section 5.3 more about *RBAC Police*.

## 4.2 Principles of a trampoline pod

Trampoline pods as described in section 4.1 are pods with powerful permissions. These permissions are granted to the `ServiceAccount` mounted in the pod. The `ServiceAccounts` are bound to certain `roles` that have permission defined in RBAC rules. In essence these `ServiceAccounts` form keys that a pod can use to communication with `Kubernetes API`. However not all keys are created equal, some keys are cluster scoped or have admin like permissions. If an adversary where to get access to a node it would be able to read all mounted `ServiceAccounts` in that node. So to generalize, access to the node means access to all keys inside it. This becomes problematic if pods in the node have powerful permissions. Because these could be misused by an adversary on the node to gain higher privileges. Such a pod allows jumping to cluster admin and are thus called trampoline pods. An adversary on a node can use all `ServiceAccounts`, so in practice an

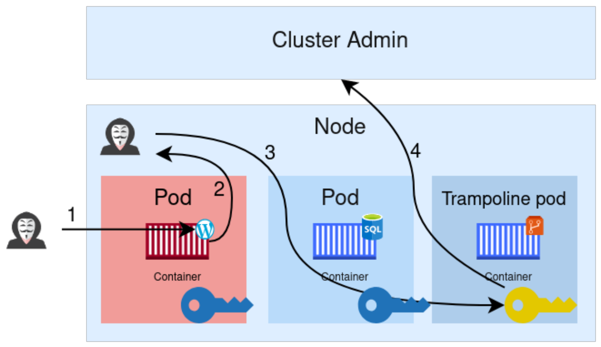

Figure 4.1: Escalation path with trampoline pod

adversary has all the permission of the pods combined. If we want to detect such powerful pods we must define what permissions are powerful. For this we will follow the classes in the work of Avrahami and Ben Hai [11]

## 4.3 Powerful permissions and classes of trampoline pods

In their whitepaper [11] Avrahami and Ben Hai have created classes that contain powerful permissions. In this section we will look into these classes. Every class forms a type of attack that could result in cluster admin. The classes we have are: *Manipulate Authentication and Authorization*, *Acquire Tokens*, *Remote Code Execution*, *Steal Pods* and *Meddler in the Middle.* For all these classes we define sets $C_i$ to $C_v$.
It is important to understand that all permissions in these classes are considered only admin equivalent if they are scoped on the cluster or a privileged `namespace` such as `kube-system`. If they are on an ordinary `namespace` they might still provide option for privilege elevation. We however won't consider this since this would broaden the scope too much. The permissions in these classes are defined by RBAC rules. This means that it is always a `verb` on a `resource`. For example the permission to list secrets is in RBAC described with the `verb:"list"` on the `resource:"secrets"`. In addition, a RBAC rule can also be scoped to a `apiGroup`. For this research we will only consider the `verb` and `resource` part of the RBAC rule. Because the `apiGroup` is usually not specified and thus defaults to `""` which is the core `apiGroup`.

### 4.3.1 Rule abbreviations

To prevent the lists of permissions from becoming too verbose we will use some abbreviations. The `verbs: control` and `modify` do not exist in Kubernetes but are introduced to make the lists below less verbose. The `verbs: update` and `patch` are combined in `modify`. Where `modify` and `create` are combined in `control`. Also some resources are combined: `DaemonSets`, `Deployments`, `CronJobs` and other pod controllers are combined into `pod controllers`. Saving 21 additional permission definitions.

Resources such as `node/status` do not imply node or `status`. But `status` as the sub-`resource` of node. This is the same notation as used for the Kubernetes API.

The `verbs: impersonate`, `escalate` and `bind` are all on `users`, `serviceaccount`, `groups` or `uid`

### 4.3.2 Manipulate Authentication/Authorization

The permissions in this class are powerful by design. They are meant to escalate privileges in certain scenarios. A pod that creates new roles might to need impersonate that cluster admin for certain roles. But should not execute everything as cluster admin. Cluster operators should be really careful when granting these permissions. This class is defined as:

$$\begin{aligned}C_i = \{&\texttt{impersonate},\\ &\texttt{escalate},\\ &\texttt{bind},\\ &\texttt{control mutating webhooks},\\ &\{\texttt{approve signers} \wedge\\ &\ \texttt{update certificatesigning-requests/approval}\}\}\end{aligned}$$

With the conjunction $\wedge$ meaning that both permissions are required.
**Attack Example**
An adversary that can bind `ClusterRoleBinding` can bind an admin `role` itself. This way it can grant the pre-installed cluster-admin (`ClusterRole`) to its compromised identity.

### 4.3.3 Acquire Tokens

In this class an attacker is able to acquire or create powerful tokens. This can be achieved directly or indirectly. The impact of the permissions in the class rely on the scope of the tokens, whether they are scoped over a privileged namespace or not.
This class is defined as:

$$\begin{aligned}C_{ii} = \{&\texttt{list secrets},\\ &\texttt{create secrets},\\ &\texttt{create serviceaccount/token},\\ &\texttt{create pods},\\ &\texttt{control pod controllers},\\ &\texttt{control validating webhooks},\\ &\texttt{control mutating webhooks}\}\end{aligned}$$

**Attack Example**
An adversary with `create serviceaccounts/token` permissions in `kube-system` namespace, can issue new powerful tokens for a `ServiceAccounts` it has control of. It could for example generate a new admin token for itself and get direct cluster admin privileges.

### 4.3.4 Remote Code Execution

Permissions in this class allow for code execution on pods and possible also nodes. If it can execute on a pod in the `control plane` it could trivially extract admin tokens. In other cases it does not directly provide privilege escalation, however it does allow for privilege elevation by moving lateral through the cluster. Other pods might then have the ability to privilege escalation.
This class is defined as:

$$\begin{aligned} C_{iii} = \{&\texttt{create pods/exec},\\ &\texttt{update pods/ephemeralcontainers},\\ &\texttt{create nodes/proxy},\\ &\texttt{control pods},\\ &\texttt{control pod controllers},\\ &\texttt{control mutating webhooks}\} \end{aligned}$$

**Attack Example**
An adversary has code execution on a different node with a powerful pod. It could then use that pod to list all secrets.

### 4.3.5 Steal Pods

A combination of permissions in this class can be used to steal pods. Stealing pods is done to get access to stronger `ServiceAccounts` from different nodes. This works by ensuring that a new pod lands inside the compromised node. This is achieved by evicting a pod and limiting the capacity of other nodes. This class is defined as:

$$\begin{aligned} C_{iv} = \{&\texttt{control pods},\\ &\{\texttt{control pod controllers} \wedge\\ &\texttt{control mutating webhooks} \wedge\\ &\{\texttt{control nodes} \vee \texttt{control nodes/status}\}\wedge\\ &\texttt{create pods/eviction} \wedge\\ &\texttt{delete pods} \wedge\\ &\texttt{delete nodes} \wedge\\ &\{\texttt{modify pods/status} \vee \texttt{modify pods}\}\}\} \end{aligned}$$

With the injunction $\vee$ meaning that either of the permissions are required.
**Attack Example**
`Node` B contains a pod that can list admin secrets, by for example having `list secrets` on `kube-system`. We want to steal it onto our node A. We

taint all other nodes with `NoExecute` such that it is evicted and rescheduled on our node.

### 4.3.6 Meddler-in-the-Middle

Permissions in this class allows an adversary to mount a Meddle-in-the-middle attack. These attacks are however generally low impact because they require an adversary to already have access to different parts of a cluster. Securing communication with TLS can also defeat most MitM attacks.

$$\begin{aligned}C_v = \{&\texttt{control endpointslices},\\ &\texttt{modify endpoints},\\ &\texttt{modify services status},\\ &\texttt{modify pods},\\ &\texttt{create services},\\ &\texttt{control mutating webhooks}\}\end{aligned}$$

# Chapter 5

# Automated assessment of Kubernetes permissions

Due to the complexity of Kubernetes and the many configurations it is prone to user error. Assessing Kubernetes manually is a tedious task and prone to human error. Therefore, automated assessment tools are needed to help administrators and security auditors assess Kubernetes clusters. Automated assessment tools are also important to establish trust in the security of a Kubernetes cluster see the work of Ullah, Ahmed and Ylitalo on this [2]. In the chapter we will first discuss how a Kubernetes cluster can be assessed, what resources are interesting and how to gather this data in section 5.1. We will then discuss two out of many automated assessment solutions that exist for Kubernetes. By first discussing*Kubescape* in section 5.2 and secondly discuss *RBAC-police* in section 5.3. There do exist other tools for security assessment, we will highlight them in section 8.2.

## 5.1 Assessing Kubernetes

In this section we will discuss techniques to assess a Kubernetes cluster. First we discuss how to collect information about the cluster then how to access the collected information. Kubernetes consists out of: `containers`, configuration and a database. The `containers` contain the various applications. For a Kubernetes assessment they are out of scope, because from the cluster perspective it is just a black box. The configuration are created by a developer or cloud engineer and are prone to error. Thus, the most meaningful assessment of a cluster must be done on the configurations. Configuration of a cluster exist out of many `YAML` files each describing a `resource`. Configuration assessment can be either done before or after deployment.

- **Before deployment**

Since Kubernetes is build up from configuration we can preemptively look for vulnerabilities. This can be integrated into CI/CD pipelines where the automated assessment would fail the pipeline if a vulnerability was introduced. This way new builds are always assessed before deployment.

- **After deployment**
  Assessing after deployment has two perspectives. An external approach where the tool only has access to publicly exposed end-points. As if it were an adversary. Or an internal approach where the tool can either be deployed in a pod with cluster wide access or collect full cluster information via the Kubernetes API. With access to the cluster it can list `resources` via the Kubernetes API and iterate over them. A deployed pod is best used to continuously monitor a cluster. Assessment via the Kubernetes API are well suited for security audit on an existing cluster by a third party. Since this thesis is written at Secura which does the latter, we will only consider this option.

To access whether a resource is vulnerable can be done in multiple ways. One approach is to have a collection of known vulnerabilities which then can be assessed against the `resources` if they still apply. This can be done by looking at the version or more aggressively testing if the accompanying exploit still works. Another approach to access the security is by checking whether the software adheres to set policies, these policies are seen as secure standards. To limit the scope of the thesis we will only look into security assessment based on policies. Because this is the approach that *Kubescape* takes and the approach we want to extend.

## 5.2 Kubescape

In this section we will discuss what *Kubescape* is and how it works. In section 5.2.1 we will have a general overview. After that in section 5.2.2 we will look at how *Kubescape* is designed and structured. In section 5.2.3 we look at the structure of rules library. In this chapter we will not go in into rules it is lacking to discover trampoline pods. This will be covered in chapter 6.

### 5.2.1 What is Kubescape?

*Kubescape* is an open-source Kubernetes tool developed by ARMO that does risk analysis and assesses security compliance. *Kubescape* scans Kubernetes clusters by collecting `YAML` files and `HELM` charts, on which it tries to detect misconfigurations based on multiple security frameworks. The formally recognized security frameworks are: NSA-CISA [12], MITRE ATTACK [17], CIS [13]. It also includes two of its on security frameworks: ARMOBest and DevOpsBest, the specifics of these are not clearly defined. *Kubescape* is during assessment used to quickly check for security framework compliance and to highlight glaring issues. Security framework compliance is an easy management pleaser because of its appeal to authority. Thus essential for reports and customer satisfaction. For this thesis we look in to *Kubescape* mainly because it is one of the tools Secura uses and the tool for which I wrote a parser as an internship project.

*Kubescape* can be used in two ways, as a command line tool or as a deployed `container`. CLI tool generates a table with the discoveries, which include controls that failed on certain `resources`. The CLI tool can also output to other formats such as `JSON` for further processing. *Kubescape* deployed as `container` can be used for continuous assessment. *Kubescape* is accompanied by the ARMO Kubescape Cloud interface which shows detailed results of scans and contains a RBAC visualizer. For continuous assessment this provides a clear monitor for the security status of the cluster. The CLI tool can also make use of the AMRO Kubescape cloud interface, for this data needs to be uploaded to their servers. In most cases this is not allowed for companies that provide security assessments, such as at Secura. Because this thesis is written accompanying an internship at Secura we will only look into functionalities of Kubescape that can be used during a security assessment. These functionalities are the CLI application and the `JSON` output.

### 5.2.2 Kubescape architecture

The team behind `Kubescape` has translated a set of security frameworks into their own *Kubescape frameworks*. Here a *framework* is just a hierarchical structure which contains *controls*. A `control` is collection of *rules* all assessing a single vulnerability. For example the *control* "*Ensure no privileged containers*" assesses all deployed `containers` and `pod controllers` for privileged `containers`. This assessment is done by the Open Policy Agent[1] (OPA) engine which gets its policies from Kubernetes regolibrary[2]. For a visual representation of this we can look at figure 5.1.

OPA is the de facto standard for assessing policies inside a Kubernetes. OPA uses its own language to define these policies this language is called Rego[3]. Rego is an extension of the decades old query language Datalog[4]. The extensions add support for structured document models such as JSON and YAML. How OPA works internally is not relevant for this thesis. It is only relevant to know that it can collect all `resources` and provides an interface to test against them with *rules*.

[1]`https://www.openpolicyagent.org/`

[2]`https://github.com/kubescape/regolibrary`

[3]`https://www.openpolicyagent.org/docs/latest/policy-language/`

[4]`https://en.wikipedia.org/wiki/Datalog`

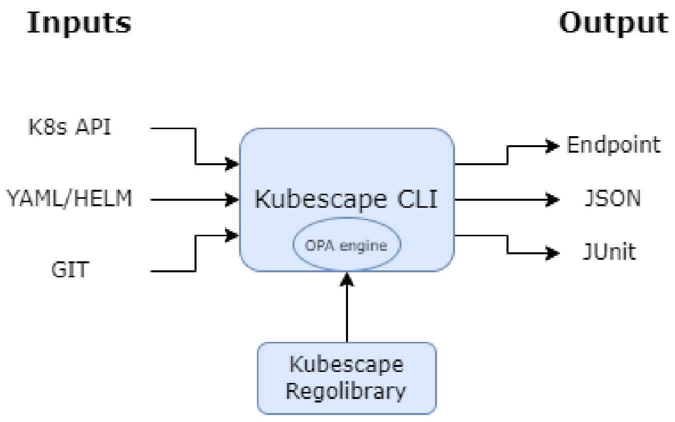

Figure 5.1: Architecture of the Kubescape
[15]

### 5.2.3 Structure of the regolibrary

How *Kubescape* must assess `resources` is not defined in its own source code. This information gets pulled from on different repository: the *regolibrary*. The *regolibrary* contains all *Kubescape frameworks*, *controls*, and *rules*. All *frameworks* are contained in the folder `frameworks`. A *framework* is just a single JSON definition containing the name, description and an array of all *controls* for an example look at appendix A.12. *Controls* are also only a single JSON definition in the folder `controls`. They contain metadata and most importantly a list of all used *rules*, for an example look at appendix A.11. *Rules* are stored in the folder `rules`, here every *rule* is in its own folder with the name of the *rule*. Inside a rule folder we have the `rule.metadata.json` and `raw.rego`. The metadata contains data such as its name and the `resources` it applies to. The `raw.rego` contains the actual policy-logic, for an example look at appendix A.2. To make our extension we must add our own framework and possibly also new *controls* and *rules*.

### 5.2.4 Running Kubescape

*Kubescape* can be easily setup locally by running the install script from its repository [15]. This installs the latest version into the path. If we however want to make modifications to *Kubescape* we must build it locally. For this we will clone the repository:

```
git clone https://github.com/kubescape/kubescape.git &&
cd kubescape
```

We can then build its dependencies:

```
make libgit2
```

And finally build *Kubescape* itself:

```
make build
```

We should now have a working *Kubescape* instance. We can test this by running:

```
make test
```

With that being successful we can try to run it against a cluster. For this you need a cluster and a correct setup `kubeconfig` file. The cluster we will be using in this thesis was provided by Secura. To connect to this cluster we need to sign in to the web portal and collect our Azure CLI commands. We can then run:

```
az account set --subscription xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxx
az aks get-credentials --resource-group kubernetes-internship
   --name Willems_sandbox
```

With this Azure CLI has set up our `kubeconfig` and we are able to interact with our cluster via *kubectl*. We can try for example collecting all deployments with:

```
kubectl get deployments
```

Now that we have everything setup we can try and run *Kubescape*. In this example we want to assess the cluster according to the NSA framework[12]. We want to keep the collected data local and not have it upload the results to the cloud. For this we use the `--keep-local` flag. To run *Kubescape* we use:

```
./kubescape scan framework nsa --keep-local
```

This will give us the results in the terminal. For an example see appendix A.1. If we however want to further process the output of *Kubescape* we can use the arguments `--format json --format-version v2 --output results.json`. This will give us a large JSON file of more than 10000 lines which include all `resources` and *controls*. This output on its own isn't very useful yet. So for the accompanying internship I created parser which can extract the failed `resources` and shows fix paths. This parser may not be shared publicly but can be provided upon request at Secura.

## 5.3 RBAC Police

In this section we will look into the tool *RBAC Police*[5] to understand its origin and look at its functionalities. As for *Kubescape* we will show how to set up and run the tool. We will also look at the structure of the *RBAC Police rules*.

[5]`https://github.com/PaloAltoNetworks/rbac-police`

### 5.3.1 What is RBAC Police?

*RBAC Police* is a tool developed by Avrahami and Ben Hai accompanying their whitepaper about trampoline pods [11]. This is the same research as mentioned in chapter 4. The tool was designed to discover trampoline pods in a cluster by evaluating the RBAC permissions of all pods, nodes, `serviceaccounts`, `users` and `groups`. It does this in a similar manner to *Kubescape* by using the OPA engine and *rules* written in rego. These *rules* are however differently structured, so these can not be used interchangeably. As an example see appendix A.2 and appendix A.3 where both *rules* check for `"create"` permissions. We can see that they use a different structure and also have their own auxiliary functions, but in the core they both iterate over `resources` and assess whether they contain the `verb "create"` on the `resources "serviceaccounts/token"`. Compared to *Kubescape* is *RBAC Police* a relatively simple tool, *RBAC Police* is only able to scan the 20 included *rules* and provides the output in JSON format. The JSON format is unfortunately very verbose making it hard to see at a glance what is vulnerable.

### 5.3.2 Running RBAC Police

Running `RBAC Police` is very similar to *Kubescape*. We first clone the repository:

```
git clone https://github.com/PaloAltoNetworks/rbac-police
cd rbac-police
```

Then we build RBAC Police:

```
go build
```

As with `Kubescape` we must make sure that we have our `kubeconfig` properly configured. For these steps look at section 5.2.4. We can now run *RBAC Police* against our cluster with:

```
./rbac-police eval lib/
```

This will give us all failed rules. If we want to inspect a `Role` and `ClusterRoles` further we can expand with them with:

```
./rbac-police expand
-z sa=kube-system:addon-http-application-routing-nginx-ingress-serviceaccount
```

This will give us a document with all `Roles` and permissions. For the output of these two command look at appendix A.4 and appendix A.5

### 5.3.3 Structure of RBAC Police

The structure of RBAC Police is relatively very straight forward. It has a normal Go package layout and in addition has the `lib` folder that contains all the rego files.

# Chapter 6

# Kubescape extension: discovering trampoline pods

Our goal is to extend Kubescape such that it has the ability to discover trampoline pods. We achieve this by incorporating the *rules* of *RBAC Police* into the *regolibrary*. Since *RBAC Police* is a tool that is specifically designed to find trampoline pods, it is a good basis for our extension. In section 6.1 we will discuss what our extension needs to contain and how we will approach it. We will then put our approach to work and identify the *rules* that are already covered by *regolibrary* in section 6.2. In section 6.3 we will discuss the creation of a new *Kubescape framework* and how it can be used.

## 6.1 Approach

In chapter 5 we went into the structure of *Kubescape* and *RBAC Police*, here we looked into their capabilities and structure. By looking at the structure we also identified the location of the *rules* and the possible location for our extension. Our extension won't be in the actual *Kubescape* code but will be in the *regolibrary* that it uses, since this is the place where are *rules* are located.

We can make our extension by creating a new *framework* with the sole purpose of discovering trampoline pods. By creating a new *framework* we get a clearly contained extension which we are able to test on its own. It also won't interfere we with structure of existing *frameworks*, making it more likely to be added to the *regolibrary* repository in the future.

To work with the *RBAC Police rules* we must have a clear definition of what they are and how we will reference them. As mentioned before in section 5.3 all *rules* are stored in the `lib` folder. We will use these filenames as the basis of the *rule* definitions. We will include the description of a *rule* to make clear for what the *rule* serves. For an overview see table 6.1

| **Severity** | **Name** | **Description** |
|---|---|---|
| Critical | approve_csrs | Create and approve certificatesigningrequests |
| Critical | assign_sa | Create pods or create, update or patch pod controllers |
| Critical | bind_roles | Bind clusterrolebindings or bind rolebindings in privileged namespaces |
| Critical | cluster_admin | Roles with cluster admin privileges |
| High | control_webhooks | Create, update or patch webhooks |
| Critical | eks_modify_aws_auth | Modify configmaps in the kube-system |
| Critical | escalate_roles | Escalate clusterrole or roles in privileged namespaces |
| Critical | impersonate | Impersonate users, groups or other serviceaccounts |
| Critical | issue_token_secrets | create or modify secrets in privileged namespaces |
| Medium | list_secrets | list secrets cluster-wide excl. privileged namespace |
| Low | modify_node_status | modify nodes' status influence nodeAffinity or nodeSelectors |
| Low | modify_pod_status | modify pods' status |
| High | modify_pods | update or patch pods in privileged namespaces with RCE |
| Medium | modify_service_status | Exploit CVE-2020-8554 |
| High | node_proxy | access to the nodes/proxy subresource for RCE |
| Low | obtain_token_weak_ns | retrieve or issue SA tokens in unprivileged namespaces |
| High | pods_ephemeral_ctrs | update or patch pods/ephemeralcontainers for RCE |
| High | pods_exec | create pods/exec permission in privileged namespaces |
| Low | prodiverIAM | Possbile abuse of ServiceAccounts assigned cloud provider IAM |
| Medium | rce_weak_ns | update or patch pods or create pods/exec in unprivileged namespaces |
| Critical | retrieve_token_secrets | retrieve secrets in privileged namespaces |
| High | steal_pods | delete or evict pods in privileged namespaces |
| Critical | token_request | create TokenRequests (serviceaccounts/token) in privileged namespaces |

Table 6.1: Rules in RBAC Police

## 6.2 Finding equivalent rules

Before we create our *framework* it is wise see which *rules* are already contained in the *regolibrary*, such that we don't introduce a *rule* twice. To do this we iterate over all *rules* in *RBAC Police* and search for equivalent *rules* in the *regolibrary*. Finding an equivalent *rules* can be done by identifying a combination of `resources`, `verbs`, `namespaces` and `apiGroups` and searching for the same combination in the *regolibrary*. Only *rules* that cover same combination or a superset of it are considered equivalent. Supersets are equivalents since the terms in the *rules* are in general disjunctive. Only in a few instances are terms conjunctive, in that case will we mention it the comparison. The wildcard (`"*"`) is not considered a superset of all terms in the consideration if *rules* are equivalent, this would make all *rules* equivalent to each other. With some *rules* it might be the case that a *rule* from *RBAC*

*Police* is partially covered because it does not include all the `verbs` or is only on the general `namespace`. We will highlight those *rules* as partially covered. The results of this equivalence comparison can be seen in table 6.2. In appendix A.6 all the comparisons for these *rules* are shown. In appendix A.7 the comparisons of the partially equivalent *rules* are shown.

| **RBAC Police rule** | **regolibrary rule** |
|---|---|
| **Covered** | |
| assign_sa | rule-can-create-pod-kube-system-v1 |
| cluster_admin | cluster-admin-role |
| list_secrets | rule-can-list-get-secrets-v1 |
| modify_pods | rule-can-create-modify-pod-v1 |
| obtain_token_weak_ns | rule-can-list-get-secrets-v1<br>rule-can-create-pod |
| rce_weak_ns | exec-into-container-v1 |
| **Partially covered** | |
| bind_roles | rule-can-bind-escalate |
| escalate_roles | rule-can-bind-escalate |
| impersonate | rule-can-impersonate-users-groups-v1 |
| retrieve_token_secrets | rule-can-list-get-secrets-v1 |
| **Not Covered** | |
| approve_csrs | |
| control_webhooks | |
| eks_modify_aws_auth | |
| issue_token_secrets | |
| modify_node_status | |
| modify_pod_status | |
| modify_service_status | |
| node_proxy | |
| pods_ephemeral_ctrs | |
| pods_exec | |
| prodiverIAM | |
| steal_pods | |
| token_request | |

Table 6.2: Comparison between RBAC Police and Kubescape regolibrary

We can see that quite some *rules* are already covered in the *regolibrary*. For our *framework* we do not need to create a new *rule* for these but can reference the existing *rules*. All other *rules* do have to be created. However, due to time constrains we will only implement one new *rule* to show that it is possible and leave the rest of the *rules* for future work.

## 6.3 Creating a new Kubescape framework

For the creation of a new *framework* we can follow the instructions provided in the README of the *regolibrary*[1]. The instruction contain examples of how the JSON documents need to be formatted we can adapt these to make our own framework. We will be starting at the smallest entities of the *framework* and build up from there. At the bottom are the *rules*.

### Trampoline pod rules

In table 6.2 we have already collected seven *rules* that discover trampoline pods, we will use these *rules* in our framework. Ideally we would implement all*rules* from *RBAC Police* that are not covered in the *regolibrary*. This is however due to time constrains not possible. For this reason we will implement one rule to show it is possible to implement more. The one *rule* we will implement is `token_request`. This *rule* was chosen because it has a critical severity status and thus is impactful to add. It is also easy to test making the verification in the next chapter simpler to achieve.

To create the *rule* we will add the directory `rule-can-create-sa-token-kube-system` to the `rules` directory. This name is chosen to have the same structure as other *rules*. In there we will create `raw.rego` and `rule.metadata.json` such as the instruction prescribe. To write the new *rule* we must extract the combination of terms from the *RBAC Police*, just as we did for equivalence comparisons.

| Rule: | `token_request` |
|---:|---|
| `verbs`: | `"create","*"` |
| `resources`: | `"serviceaccounts/token","*"` |
| `namespace`: | `"kube-system"` |
| `apiGroup`: | `""` |

Table 6.3: Terms of token_request

We must now write a *rule* in the format of the *regolibrary* to assess these terms. We can find a reference of the *rule* format in the README. It is however more efficient to find a similar *rule* and adapt it to our terms. The perfect *rule* for us to adapt is `rule-can-create-pod-kube-system-v1`. This *rule* assess whether a role can access the `kube-system namespace` and has the `create` verb on `pods` resource. All we need to change is resource from

[1]`https://github.com/kubescape/regolibrary/blob/master/README.md`

`"pods","*"` to `"serviceaccounts/token","*"`. The result of this can be found in appendix A.8. For the `rule.metadata.json` we can do the same, here we have to set the name to `rule-can-create-sa-token-kube-system` and provide a suitable description, for `rule.metadata.json` see appendix A.9

### Trampoline pod controls

Now that we have *rules* we need controls. For all existing *rules* we can reference the control they are a part of, their parent control. We can easily find these by filtering on all files that contain the *rule's* name. If there are multiple we can choose the control that is the most specific. The most specific would be the control with the least RBAC *rules*. The list of parent controls can be found in table 6.4.

| **Rule** | **Parent control** |
|---|---|
| rule-can-create-pod-kube-system-v1 | No parent |
| cluster-admin-role | Ensure that the cluster-admin role is only used where required |
| rule-can-list-get-secrets-v1 | Minimize access to secrets |
| rule-can-create-modify-pod-v1 | New container |
| rule-can-create-pod | Minimize access to create pods |
| exec-into-container-v1 | Exec into container |

Table 6.4: Parent controls

We however have discovered that one control has been placed on inactive and that one *rule* is orphaned. The `New container` control can be moved back to the `controls` directory to be used again. For the orphaned *rule* have to create a *control*, we can do that together with our new *rule*. *Controls* consist out of mostly metadata en information for the end user. To create the *control* we followed the structure provided in the README and look at the wording of other *controls*. The *control* for the orphaned *rule* can be found in appendix A.10 and for our new *rule* in appendix A.11. We can now make a list of all *controls* including our two new ones:

- `Minimize access to create serviceaccounts tokens`
- `Minimize access to create pods in kube-system`
- `Ensure that the cluster-role is only used where required`
- `Minimize access to secrets`
- `Exec into container`

### Trampoline pod framework

With a list of *controls* we can now create our *framework*. As with all other steps we get the structure from the README. A *framework* isn't much more than a name, a description and a list of the *control* names. Our *framework* can be found in appendix A.12. With that we have now created our own *framework*. However, creating a *framework* is worth noting if we do not test and verify it. We will do this in chapter 7.

As a final step to be able to use and test our *framework* we should generate a release. This way we can reference it in *Kubescape*. We do this by calling `python scripts/export.py` inside the *regolibrary* directory. This will create the folder `release`. This folder will contain all *frameworks* and *control*. To use our *framework* with *Kubescape* we run it with the argument `--use-from`. So in our case that would result in `kubescape scan --use-from <release/trampoline-pods.json>`

# Chapter 7

# Testing the extension

Now that we have created our framework we must test it and verify that it works. First we have the test whether our rules function properly. This can be done with the build-in *test runner* we will do this in section 7.1. If our *rule* work properly on its own we must test whether *Kubescape* is able to detect trampoline pods in a cluster. For this we must set up a vulnerable cluster that violates our *rule*. To then run *Kubescape* with our *framework* against it to try discover the trampoline pod.

## 7.1 Testing of new rule with test runner

To test if our new *rule* has valid syntax and gives us the expected output we can use the *test runner* . This is included in the `regolibrary`. We can run the *test runner* with `go test -v -tags=static rego_test.go -run TestSingleRule`. The tool is very basic and only uses hard coded values. We must set which *rule* we want to test on line 40 in `rego_test.go`. We also need to specify the test data and expected output. For the test data we must create a folder with the *rule* name in the `rules-tests` directory. As we did for the *rule* we can again adapt the tests from `rule-can-create-pod-kube-system-v1` to match for creation of `serviceAccounts/tokens`. This includes two tests, one for `clusterRole` with `roleBinding` and one for `clusterRole` with `ClusterRoleBinding`. These test can be found in appendix A.13 and appendix A.14 accordingly. We choose for these two role and binding combinations since they could actually lead to trampoline pods. A `role` with `roleBinding` would not create a trampoline pod because it would then only be scoped to the `kube-system namespace` thus never functioning as a trampoline pod. The last configuration of `role` in combination with `clusterRoleBinding` would also not be able to result in a trampoline pod because a `role` can only be bound to its namespace. This would thus never be available outside `kube-system`. When we run the *test runner* both cases test successfully, so we can now test the *framework* as a whole.

## 7.2 Testing it with Kubernetes

For this we have to set up the cluster with a trampoline pod. We create an admin equivalent `Role` bound to a `ServiceAccount` this will be done in section 7.2.1. We will then use *RBAC Police* to see if it is able to discover the introduced vulnerability in section 7.2.2. With *RBAC Police* being able to discover it we will test whether our new *framework* is also able to discover the trampoline pod in section 7.2.3.

### 7.2.1 Setting up a vulnerable cluster

In this section we will go into the steps to deploy a trampoline pod. We will create a trampoline pod with the `create serviceaccounts/token` permissions because this is the one *rule* we added to the framework. To deploy this trampoline pod we need a cluster and have a working connection with `kubectl` for this see the azure steps in section 5.2.4. To create a trampoline pod we need four configurations: `clusterrole.yaml`, `serviceaccount.yaml`, `clusterrolebinding.yaml` and `deployment.yaml`. The `clusterRole` contains the actual permissions, for token creation we need the `verb: "create"` on `resource: "serviceaccounts/token"`. The role configuration will then look as following:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
   name: create-serviceaccount-token
rules:
   - apiGroups: [""]
      resources: ["serviceaccounts/token"]
      verbs: ["create", "list"]
```

Listing 7.1: clusterrole.yaml

The `verb:"list"` was added to verify that the detection is disjunctive and does not require exact matches. To use our `role` we have to bind it to a `ServiceAccount` that a pod in the default namespace could use.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
   name: create-tokens
   namespace: default
```

Listing 7.2: serviceaccount.yaml

The role and `ServiceAccount` have to be bound together for this we define the `ClusterRoleBinding`:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
   name: create-serviceaccount-token
subjects:
- kind: ServiceAccount
   name: create-tokens
   namespace: default
roleRef:
   kind: ClusterRole
   name: create-serviceaccount-token
   apiGroup: rbac.authorization.k8s.io
```

Listing 7.3: clusterrolebinding.yaml

The last configuration we need is the deployment. With the deployment we can specify which `container` needs which `ServiceAccount`. For our example we create a mock web server that has the `ServiceAccount: create-tokens` bound to it.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
   name: nginx-deployment
   labels:
      app: nginx
spec:
   replicas: 2
   selector:
      matchLabels:
      app: nginx
   template:
      metadata:
      labels:
         app: nginx
      spec:
         serviceAccountName: create-tokens
         containers:
         - name: nginx
            image: nginx:1.14.2
            ports:
            - containerPort: 80
```

Listing 7.4: deployment.yaml

With all these configuration files create we can apply them to the cluster. This is done using `kubectl`:

```
kubectl apply -f <filename>
```

We do this for all files in the same order as they were mentioned. With all configurations applied we verify if our deployment rolled out successfully with:

```
kubectl rollout status deployment nginx-deployment
```

### 7.2.2 Testing with RBAC Police

As mentioned in section 5.3.2 we can run *RBAC Police* by executing:

```
./rbac-police eval ./lib/
```

Since we only introduced a trampoline pod with `"create serviceaccounts/token` permissions we will only evaluate the "token-request.rego" which assess this.

```
./rbac-police eval lib/token_request.rego
```

The output can be seen in appendix A.15. Here we see that it detected the `ServiceAccount` and shows pods it is bound. In this case `nginx-deployment-6544fdf46f-lmjtj` and `nginx-deployment-6544fdf46f-8vc2n`

### 7.2.3 Testing trampoline pod framework

In the previous section we have shown that a vulnerability was introduced into the cluster. Now we need to test whether our new *framework* is also able to detect this vulnerability. As mentioned in section 6.3 we generate a new release of our Kubescape framework inside the regolibrary directory with `python scripts/export.py`. In `./release` we now have our `trampoline-pods.json`. To assess the cluster we can run:

```
kubescape scan --use-from ./release/trampoline-pods.json
```

This will provide the table as mentioned in section 5.2.4 which can be seen in appendix . This table only globally provides which *controls* failed and how many `resources` did. To determine whether our new *framework* detects the introduced vulnerability we have to add the verbose flag `-v`. This will provide us with a more in dept result of all the `resources` that failed. Running it again with the verbose flag:

```
kubescape scan --use-from ./release/trampoline-pods.json -v
```

```
   ############################################################################
ApiVersion:
Kind: ServiceAccount
Name: create-tokens
Namespace: default

Controls: 5 (Failed: 1, Excluded: 0)

+----------+-------------------------------+-------------------------------------+-----------------------------------------+
| SEVERITY |          CONTROL NAME         |                DOCS                 |           ASSISTANT REMEDIATION         |
+----------+-------------------------------+-------------------------------------+-----------------------------------------+
| Medium   | Minimize access to create     | https://hub.armosec.io/docs/tp-0002 | relatedObjects[1].rules[0].resources[0] |
|          | serviceaccounts tokens        |                                     | relatedObjects[1].rules[0].verbs[0]     |
|          |                               |                                     | relatedObjects[1].rules[0].apiGroups[0] |
|          |                               |                                     | relatedObjects[0].subjects[0]           |
|          |                               |                                     | relatedObjects[0].roleRef.name          |
+----------+-------------------------------+-------------------------------------+-----------------------------------------+
```

The full output can be seen in appendix A.16. With the verbose flag every failed `resource` gets highlighted. The `resource` we introduced 'create-token' failed our *control*. So our *framework* was able to detect this type of trampoline pod. It however does not mark the pod as vulnerable. This is because the pod is not seen as vulnerable. It is only vulnerable if the `ServiceAccount` is bound to the pod that has the `create serviceaccount/token` permission. For future work we could add a check to see if the `ServiceAccount` that has the trampoline pod permissions is bound to a pod. This could then mark the pod as vulnerable.

# Chapter 8

# Future Work

In this thesis we covered many subjects and ideas. There were however ideas we could not cover or go in to more dept. Because these subjects are still interesting for further discovery we will highlight them here.

## 8.1 Finding more powerful permissions

Avrahami and Ben Hai[11] already provided many permissions that could lead to privilege elevation, see table 6.1. It is however possible that there are more permission combinations that could lead to privilege escalation. For this we would need to do more research into the Kubernetes API and the RBAC system. This would require much diligence and in-dept knowledge of the Kubernetes API. This is however remains an interesting subject for further research.

## 8.2 Other Kubernetes security assessment tools

In this thesis we have only looked at *Kubescape* and *RBAC police*. They are however two only two tools that provide Kubernetes security assessment. Due to time constrains other tools where only shortly inspected. Here we found that neither *Kube Hunter*[1], *kube-bench*[2] nor *MKIT*[3] was able to do trampoline pod discovery. These tools were found by a quick google search, there are however many more tools available. For a more wholistic approach all popular tools needs to be evaluated to determine which tools have to capabilities to do RBAC evaluation. Future research should evaluate which tools are also capable of doing RBAC evaluation and which tools are able to represent this in a clear manner.

[1]`https://github.com/aquasecurity/kube-hunter`
[2]`https://github.com/aquasecurity/kube-bench`
[3]`https://github.com/darkbitio/mkit`

## 8.3 Extension for all uncovered RBAC police rules

In this thesis we only created an extension for permission combination of `create tokens`. To make the *framework* practical and useful it needs to be extended to cover all rules which noted as not covered in table 6.2. This would require similar implementations as the one we did for the `create token` permissions. This would take considerable time and effort which was not possible in the scope of this thesis. A completed Kubescape trampoline pod framework would be a useful tool for Kubernetes security auditors and the Kubernetes community. Making this an interesting subject for future research.

## 8.4 Highlight the affected pod in the results

Our *framework* now discovers trampoline pods on the basis of `ServiceAccounts` in use. In the results *Kubescape* now only shows the `ServiceAccounts`. Ideally it would be better to show the pod that is affected by the `ServiceAccounts`. To achieve this we would have to change functionalities stretching being discovering trampoline pods. Instead of extending the *regolibrary* this would require change in the *Kubescape* source code itself. Since this would change the functionality of *Kubescape* it would require quite an effort to get this change implemented and ideally merged into the repository. This however would still remain an interesting subject for future research.

# Chapter 9

# Conclusions

Configuring Kubernetes securely is difficult. This problem stems from the complexity of Kubernetes itself, creating a cluster requires knowledge of the many objects inside Kubernetes. Each `object` and `resource` has a configuration defining what it: is, can and must do. This makes in essence that Kubernetes uses just one large collection of configurations. To assess whether these configurations are secure by hand is a labor-intensive task. So for this automated assessments tools have been developed which are highlighted in chapter 5.

*Kubescape* is popular and by the CNFC recognized automated assessment tool that assesses whether a cluster adheres to different security frameworks. It does this by collecting all resource and comparing these to specified policies in its rule library. Every *rule* that fails gets collected and is shown to the user. A second tool that does automated assessments is *RBAC Police*, this tool however contains only a very specific set of *rules* that detects trampoline pods. Trampoline pods are a class of pods that when exploited allow for privilege escalation. Since *RBAC Police* was released in tandem with the research on trampoline pods we can assume that it is able to detect all trampoline pods. We wanted to extend *Kubescape* with the *rules* from *RBAC Police* to make it able to detect trampoline pods as well and thus providing this capability to more security researchers. This would make *Kubescape* an even more complete tool for Kubernetes security assessments.

Our goal for this thesis was to create a *Kubescape* extension to detect trampoline pods. We achieved this by creating a new *Kubescape framework* inside the *regolibrary.* In our new *framework* we added *rules* to detect permission that enable pods to become trampoline pods. For the specification of our *rules* we looked at *RBAC Police* since it is the authority on trampoline pods. From *RBAC Police* we extracted the `verbs`, `resources` and `subjects` that form a *rule* for a specific type of trampoline pod. Such that we could implement them in a manner that *Kubescape* understands. We successfully created a new *Kubescape rule* for 1 of the 17 uncovered *RBAC* *police rules*. The implementation method for this one *rule* can also be applied to the 16 remaining *rules* without much effort. Apart from new *rules* we were able to add 4 existing *Kubescape rules* to our own *framework* that aided in trampoline pod discovery.

We were able to test our new *rule* by writing units tests for the test runner in the *regolibrary*. And we were able to use our extension on a vulnerable cluster with trampoline pods. Here our extensions provided equivalent results to *RBAC Police* by highlighting the trampoline pod permission. From that we can conclude that our framework is able to detect this type of trampoline pod.

Another conclusion we can make after this research is that Kubernetes is a very complex and broad system. This makes it too broad for a bachelor thesis to cover every aspect equally. We were able to cover a small part of the Kubernetes security landscape, but there is still a lot more to be researched. With the absence of containerization in the bachelor curriculum we would not recommend a bachelor student to do a thesis on Kubernetes security without a lot of prior knowledge and the willingness to write a very broad background. It is unlikely that containerization will disappear in the near feature making it an interesting subject to be added to the curriculum such that research on Kubernetes security can be done by bachelor students.

# Bibliography

[1] David Ferraiolo and Richard Kuhn. "Role-Based Access Controls". In: *Proceedings of the 15th National Computer Security Conference.* National Institute of Standards and Technology. 1992, pp. 554–563. URL: `https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1992/10/13/role-based-access-controls/documents/ferraiolo-kuhn-92.pdf`.

[2] Kazi Wali Ullah, Abu Shohel Ahmed, and Jukka Ylitalo. "Towards Building an Automated Security Compliance Tool for the Cloud". In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications.* 2013, pp. 1587–1593. DOI: `10.1109/TrustCom.2013.195`.

[3] Augusto Remillano II. *Malicious Docker Hub Container Images Used for Cryptocurrency Mining.* News arcticle from Trend Micro. Aug. 2020. URL: `https://www.trendmicro.com/vinfo/us/security/news/virtualization-and-cloud/malicious-docker-hub-container-images-cryptocurrency-mining`.

[4] Md. Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices". In: *2020 IEEE Secure Development (SecDev).* 2020, pp. 58–64. DOI: `10.1109/SecDev45635.2020.00025`.

[5] Or Azarzar. *10 ways to Escalate Privileges in Kubernetes.* Lightspin. 2021. URL: `https://blog.lightspin.io/kubernetes-pod-privilege-escalation`.

[6] Cloud Native Computing Foundation. *CNCF Annual Survey 2021.* 2021. URL: `https://www.cncf.io/reports/cncf-annual-survey-2021/`.

[7] Andy Nguyen. *CVE-2021-22555: Turning x00x00 into 10000$.* July 2021. URL: `https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html`.

[8] Chris Richardson. *What are microservices?* 2021. URL: `https://microservices.io`.

[9] Shazibul Islam Shamim. “Mitigating Security Attacks in Kubernetes Manifests for Security Best Practices Violation”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1689–1690. DOI: 10.1145/3468264.3473495.

[10] Muntaha Alawneh and Imad Abbadi. “Expanding DevSecOps Practices and Clarifying the Concepts within Kubernetes Ecosystem”. In: *2022 Ninth International Conference on Software Defined Systems (SDS).* 2022, pp. 1–7. DOI: 10.1109/SDS57574.2022.10062874.

[11] Yuval Avrahami and Shaul Ben Hai. *Kubernetes Privilege Escalation: Excessive Permissions in Popular Platforms.* Tech. rep. MSU-CSE-06-2. Paloalto Networks, 2022. URL: https://www.paloaltonetworks.com/apps/pan/public/downloadResource?pagePath=/content/pan/en%5C_US/resources/whitepapers/kubernetes-privilege-escalation-excessive-permissions-in-popular-platforms.

[12] National Security Agency Cybersecurity and Infrastructure Security Agency. *NSA Kubernetes Hardening Guide.* 2022. URL: https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF.

[13] Center for Internet Security. *CIS Kubernetes Benchmark.* 2022. URL: https://workbench.cisecurity.org/benchmarks/8973.

[14] Nicholas Pecka Lotfi Ben Othmane and Altaz Valani. “Privilege Escalation Attack Scenarios on the DevOps Pipeline Within a Kubernetes Environment”. In: *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering.* ICSSP’22. Pittsburgh, PA, USA: Association for Computing Machinery, 2022, pp. 45–49. ISBN: 9781450396745. DOI: 10.1145/3529320.3529325.

[15] ARMO. *Kubescape.* webpage. 2023. URL: https://github.com/kubescape/kubescape.

[16] Carlos Polop. *Docker Breakout / Privilege Escalation.* 2023. URL: https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-breakout/docker-breakout-privilege-escalation.

[17] *MITRE ATT&CK Matrix for containers.* URL: https://attack.mitre.org/matrices/enterprise/containers/.

# Appendix

In this appendix we have outputs from *Kubescape* and *RBAC Police* that were too large to include in the text.The appendix also includes an overview of the rule comparisons. Aswell as the rule and metadata for the Kubescape extension.

## A.1 Kubescape output

Output table of *Kubescape* whill running in the terminal. Used in section 5.2.4

```
-> kubescape git:(master) ./kubescape scan framework nsa --keep-local
[info] Kubescape scanner starting
[warning] unknown build number, this might affect your scan results. Please make sure you are updated to latest version
[warning] Kubernetes cluster nodes scanning is disabled. This is required to collect valuable data for certain controls.
          You can enable it using  the --enable-host-scan flag
[info] Downloading/Loading policy definitions
[success] Downloaded/Loaded policy
[info] Accessing Kubernetes objects
[success] Accessed to Kubernetes objects
[info] Requesting Host scanner data
[info] Scanning. Cluster: Willems_sandbox
[success] Done scanning. Cluster: Willems_sandbox

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Controls: 19 (Failed: 14, Excluded: 0, Skipped: 2)
Failed Resources by Severity: Critical | 0, High | 14, Medium | 54, Low | 8

+----------+--------------------------------------------------+------------------+--------------------+---------------+--------------+
| SEVERITY |                   CONTROL NAME                   | FAILED RESOURCES | EXCLUDED RESOURCES | ALL RESOURCES | % RISK-SCORE |
+----------+--------------------------------------------------+------------------+--------------------+---------------+--------------+
| Critical | Disable anonymous access to Kubelet service      |        0         |         0          |       0       |   skipped*   |
| Critical | Enforce Kubelet client TLS authentication        |        0         |         0          |       0       |   skipped*   |
| High     | Applications credentials in configuration files |        2         |         4          |      40       |      4%      |
| High     | HostNetwork access                               |        3         |         4          |      19       |     13%      |
| High     | Privileged container                             |        1         |         3          |      19       |      4%      |
| High     | Resource limits                                  |        8         |         7          |      19       |     39%      |
| Medium   | Allow privilege escalation                       |        8         |         10         |      23       |     33%      |
| Medium   | Automatic mapping of service account             |        17        |         54         |      72       |     24%      |
| Medium   | Cluster internal networking                      |        2         |         3          |       6       |     33%      |
| Medium   | Cluster-admin binding                            |        1         |         2          |      85       |      1%      |
| Medium   | Container hostPort                               |        1         |         2          |      19       |      4%      |
| Medium   | Exec into container                              |        2         |         2          |      85       |      2%      |
| Medium   | Ingress and Egress blocked                       |        8         |         11         |      20       |     37%      |
| Medium   | Linux hardening                                  |        7         |         10         |      19       |     34%      |
| Medium   | Non-root containers                              |        8         |         11         |      19       |     39%      |
| Low      | Immutable container filesystem                   |        8         |         10         |      19       |     39%      |
+----------+--------------------------------------------------+------------------+--------------------+---------------+--------------+
|          |                 RESOURCE SUMMARY                 |        21        |         60         |      189      |    14.05%    |
+----------+--------------------------------------------------+------------------+--------------------+---------------+--------------+
FRAMEWORK NSA

* enable-host-scan flag not used. For more information: https://hub.armosec.io/docs/host-sensor

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Scan results have not been submitted:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Run with '--verbose'/'-v' flag for detailed resources view
```

## A.2 Example of Kubescape Rego

Example of Rego used by *Kubescape* for in section 5.2.3.

```
package armo_builtins

import future.keywords.in

```

```
# fails if user has create access to pods within kube-system namespace
deny[msga] {
  subjectVector := input[_]
  role := subjectVector.relatedObjects[i]
  rolebinding := subjectVector.relatedObjects[j]
  endswith(role.kind, "Role")
  endswith(rolebinding.kind, "Binding")

  can_create_to_pod_namespace(rolebinding)
  rule := role.rules[p]

  subject := rolebinding.subjects[k]
  is_same_subjects(subjectVector, subject)

is_same_subjects(subjectVector, subject)
  rule_path := sprintf("relatedObjects[%d].rules[%d]", [i, p])

  verbs := ["create", "*"]
  verb_path := [sprintf("%s.verbs[%d]", [rule_path, l])
      | verb = rule.verbs[l]; verb in verbs]
  count(verb_path) > 0

  api_groups := ["", "*"]
  api_groups_path := [sprintf("%s.apiGroups[%d]", [rule_path, a])
    | apiGroup = rule.apiGroups[a]; apiGroup in api_groups]
  count(api_groups_path) > 0

  resources := ["serviceaccounts/token", "*"]
  resources_path := [sprintf("%s.resources[%d]", [rule_path, l])
    | resource = rule.resources[l]; resource in resources]
  count(resources_path) > 0

  path := array.concat(resources_path, verb_path)
  path2 := array.concat(path, api_groups_path)
  finalpath := array.concat(path2, [
     sprintf("relatedObjects[%d].subjects[%d]", [j, k]),
     sprintf("relatedObjects[%d].roleRef.name", [j]),
  ])

  msga := {
     "alertMessage": sprintf("Subject: %s-%s can create serviceaccounts/tokens
         in kube-system", [subjectVector.kind, subjectVector.name]),
     "alertScore": 3,
     "failedPaths": finalpath,
     "fixPaths": [],
     "packagename": "armo_builtins",
     "alertObject": {
        "k8sApiObjects": [],
        "externalObjects": subjectVector,
    },
  }
}

# 1. rolebinding in kubesystem ns + role in kubesystem ns
# 2. rolebinding in kubesystem ns + clusterrole
can_create_to_pod_namespace(rolebinding) {
   rolebinding.metadata.namespace == "kube-system"
}

# 3. clusterrolebinding + clusterrole
can_create_to_pod_namespace(rolebinding) {
   rolebinding.kind == "ClusterRoleBinding"
}

# for service accounts
is_same_subjects(subjectVector, subject) {
   subjectVector.kind == subject.kind
   subjectVector.name == subject.name
   subjectVector.namespace == subject.namespace
}

# for users/ groups
is_same_subjects(subjectVector, subject) {
   subjectVector.kind == subject.kind
   subjectVector.name == subject.name
   subjectVector.apiGroup == subject.apiGroup
}
```

## A.3 Example of RBAC police REGO

Example of Rego used by *RBAC Police* for in section 5.2.3.

```
package policy
import data.police_builtins as pb
import future.keywords.in

describe[{"desc": desc, "severity": severity}] {
  desc := sprintf("Identities that can create TokenRequests (
    serviceaccounts/token) in privileged namespaces (%v) can
    issue tokens for admin-equivalent SAs", [concat(", ", pb.
    privileged_namespaces)])
  severity := "Critical"
}
targets := {"serviceAccounts", "nodes", "users", "groups"}

evaluateRoles(roles, owner) {
  not pb.nodeRestrictionEnabledAndIsNode(owner)
  some role in roles
  pb.affectsPrivNS(role)
  some rule in role.rules
  pb.subresourceOrWildcard(rule.resources, "serviceaccounts/
    token")
```

```
 pb.valueOrWildcard(rule.verbs, "create")
 pb.valueOrWildcard(rule.apiGroups, "")
}
```

## A.4 RBAC Police eval

Example of the output RBAC Police eval command for section 5.3.2.

```
->  rbac-police git:(main) ./rbac-police eval ./lib/retrieve_token_secrets.rego
{
 "policyResults": [{
  "policy": "./lib/retrieve_token_secrets.rego",
  "severity": "Critical",
  "description": "Identities that can retrieve secrets in privileged namespaces (kube-system) can obtain tokens of admin-equivalent SAs",
  "violations": {
   "serviceAccounts": [{
     "name": "addon-http-application-routing-nginx-ingress-serviceaccount",
     "namespace": "kube-system",
     "nodes": [{
      "aks-agentpool-24176822-vmss00000h": [
       "addon-http-application-routing-nginx-ingress-controller-c64771d"
      ]
     }]
    },
    {
     "name": "csi-azurefile-node-sa",
     "namespace": "kube-system",
     "nodes": [{
       "aks-agentpool-24176822-vmss00000g": [
        "csi-azurefile-node-7jrph"
       ]
      },
      {
       "aks-agentpool-24176822-vmss00000h": [
        "csi-azurefile-node-d9jxn"
       ]
      }
     ]
    },
    {
     "name": "tigera-operator",
     "namespace": "tigera-operator",
     "nodes": [{
      "aks-agentpool-24176822-vmss00000h": [
       "tigera-operator-74fc475fbb-bggst"
      ]
     }]
```

```
        }
      ],
      "nodes": [
        "aks-agentpool-24176822-vmss00000g",
        "aks-agentpool-24176822-vmss00000h"
      ]
    }
  }],
  "summary": {
    "failed": 1,
    "passed": 0,
    "errors": 0,
    "evaluated": 1
  }
}
```

## A.5 RBAC Police expand

Example of the output RBAC Police expand command for section 5.3.2.

```
-> rbac-police git:(main) ./rbac-police expand -z sa=kube-
    system:addon-http-application-routing-nginx-ingress-
    serviceaccount
{
  "name": "addon-http-application-routing-nginx-ingress-
    serviceaccount",
  "namespace": "kube-system",
  "nodes": [{
    "name": "aks-agentpool-24176822-vmss00000h",
    "pods": [
      "addon-http-application-routing-nginx-ingress-controller-
    c64771d"
    ]
  }],
  "roles": [{
      "name": "addon-http-application-routing-nginx-ingress-
    role",
      "effectiveNamespace": "kube-system",
      "rules": [{
          "verbs": [
            "get"
          ],
          "apiGroups": [
            ""
          ],
          "resources": [
            "configmaps",
            "pods",
            "secrets",
            "namespaces"
          ]
        },
        {
          "verbs": [
            "get",
            "update"
          ],
          "apiGroups": [
            ""
          ],
          "resources": [
            "configmaps"
          ],
          "resourceNames": [
            "ingress-controller-leader-addon-http-application-routing",
            "ingress-controller-leader"
          ]
        },
        {
          "verbs": [
            "create"
          ],
          "apiGroups": [
            ""
          ],
          "resources": [
            "configmaps"
          ]
        },
        {
          "verbs": [
            "get"
          ],
          "apiGroups": [
            ""
          ],
          "resources": [
            "endpoints"
          ]
        }
      ]
    },
    {
      "name": "addon-http-application-routing-nginx-ingress-clusterrole",
      "rules": [{
          "verbs": [
            "list",
            "watch"
          ],
          "apiGroups": [
            ""
          ],
          "resources": [
            "configmaps",
            "endpoints",
            "nodes",
            "pods",
            "secrets"
          ]
        },
        {
          "verbs": [
            "get"
          ],
          "apiGroups": [
            ""
          ],
          "resources": [
            "nodes"
          ]
        },
        {
          "verbs": [
            "get",
            "list",
            "watch"
          ],
          "apiGroups": [
            ""
          ],
          "resources": [
            "services"
          ]
        }
    }
  ]
}
```

## A.6 Comparison of rules that are equivalent

The following tables show the rules that are equivalent between *RBAC police* and *Kubescape* for section 6.2 .

| | *RBAC police* | *Kubescape* |
|---|---|---|
| Rule: | modify_pods | rule-can-create-modify-pod-v1 |
| `verbs`: | `"patch", "update", "*"` | `"create", "patch", "update", "*"` |
| `resources`: | `"pods"` | `"pods", "deployments", "daemonsets"` `"replicasets", "statefulsets",` `"jobs", "cronjobs", "*"` |

| | *RBAC police* | *Kubescape* |
|---:|---|---|
| Rule: | cluster_admin | cluster-admin-role |
| `verbs`: | `"*"` | `"*"` |
| `resources`: | `"*"` | `"*"` |

| | *RBAC police* | *Kubescape* |
|---:|---|---|
| Rule: | list_secrets | rule-can-list-get-secrets-v1 |
| `verbs`: | `"list","*"` | `"list", "get","watch", "*"` |
| `resources`: | `"secrets","*"` | `"secrets","*"` |

| | *RBAC police* | *Kubescape* |
|---:|---|---|
| Rule: | assign_sa | rule-can-create-pod-kube-system-v1 |
| `verbs`: | `"create","*"` | `"create", "*"` |
| `resources`: | `"pods","*"` | `"pods","*"` |
| `namespace`: | `"kube-system"` | `"kube-system","*"` |

| | *RBAC police* | *Kubescape* |
|---:|---|---|
| Rule: | obtain_token_weak_ns | rule-can-list-get-secrets-v1 |
| `verbs`: | `"list","get","*"` | `"get","list","watch","*"` |
| `resources`: | `"secrets","serviceaccounts/token","*"` | `"secrets","*"` |
| OR | | |
| Rule: | obtain_token_weak_ns | rule-can-create-pod |
| `verbs`: | `"create","*"` | `"create", "*"` |
| `resources`: | `"pods","*"` | `"pods","*"` |

Table 1: Contains two cases covered in two rules:

| | *RBAC police* | *Kubescape* |
|---:|---|---|
| Rule: | bind_roles | rule-can-bind-escalate |
| `verbs`: | `"bind","*"` | `"bind", "*"` |
| `resources`: | `"clusterrolebindings","rolebindings","*"` | `"clusterrolebindings","rolebindings",` `"clusterroles", "roles","*"` |
| `namespace`: | `"kube-system"` | |
| `apiGroups`: | `"rbac.authorization.k8s.io","*"` | `"rbac.authorization.k8s.io","*"` |

Table 2: `rule-can-bind-escalate` is not specific to privilege namespaces however reports on all. We however consider this equivalent since the privileged namespace is a subset of it.

| | *RBAC police* | *Kubescape* |
|---|---|---|
| Rule: | escalate_roles | rule-can-bind-escalate |
| `verbs`: | `"escalate","*"` | `"escalate", "*"` |
| `resources`: | `"clusterrolebindings","rolebindings","*"` | `"clusterrolebindings","rolebindings"` `"clusterroles", "roles","*"` |
| `namespace`: | `"kube-system"` | |
| `apiGroups`: | `"rbac.authorization.k8s.io","*"` | `"rbac.authorization.k8s.io","*"` |

| | *RBAC police* | *Kubescape* |
|---|---|---|
| Rule: | rce_weak_ns | exec-into-container-v1 |
| `verbs`: | `"create","*"` | `"create", "*"` |
| `resources`: | `"pods/exec","*"` | `"pods/exec","pods/*","*"` |
| OR | | |
| `verbs`: | `"update","patch","*"` | - |
| `resources`: | `"pods","*"` | - |

Table 3: Update and patch are not covered

## A.7 Comparison of rules that are partially equivalent

The following tables show the rules that are partially equivalent between *RBAC police* and *Kubescape* for section 6.2 .

| | *RBAC police* | *Kubescape* |
|---|---|---|
| Rule: | impersonate | rule-can-impersonate-users-groups-v1 |
| `verbs`: | `"impersonate","*"` | `"impersonate", "*"` |
| `resources`: | `"users","groups","serviceaccounts","*"` | `"users", "serviceaccounts",` `"groups", "uids", "*"` |
| OR | | |
| `verbs`: | `"impersonate","*"` | - |
| `resources`: | `"userextras","*"` | - |
| `apiGroups`: | `"rbac.authorization.k8s.io","*"` | - |

Table 4: Impersonate on userextras is not covered

| | *RBAC police* | *Kubescape* |
|---|---|---|
| Rule: | retrieve_token_secrets | rule-can-list-get-secrets-v1 |
| `verbs`: | `'list","get","*"` | `"list", "get", "*"` |
| `resources`: | `"secrets","*"` | `"secrets","*"` |
| `namespace`: | `"kube-system"` | `"*"` |

Table 5: The regolibrary only assess the wildcard not privileged namespaces

## A.8 Token request rule

The new rego rule that is used to assess the token request inside the privileged namespace for section 6.3
`rules/rule-can-create-sa-token-kube-system/raw.rego`

```
package armo_builtins

import future.keywords.in

# fails if user has create access to pods within kube-system
    namespace
deny[msga] {
  subjectVector := input[_]
  role := subjectVector.relatedObjects[i]
  rolebinding := subjectVector.relatedObjects[j]
  endswith(role.kind, "Role")
  endswith(rolebinding.kind, "Binding")

  can_create_to_pod_namespace(rolebinding)
  rule := role.rules[p]

  subject := rolebinding.subjects[k]
  is_same_subjects(subjectVector, subject)

is_same_subjects(subjectVector, subject)
  rule_path := sprintf("relatedObjects[%d].rules[%d]", [i, p])

  verbs := ["create", "*"]
  verb_path := [sprintf("%s.verbs[%d]", [rule_path, l]) | verb
    = rule.verbs[l]; verb in verbs]
  count(verb_path) > 0

  api_groups := ["", "*"]
  api_groups_path := [sprintf("%s.apiGroups[%d]", [rule_path, a
    ]) | apiGroup = rule.apiGroups[a]; apiGroup in api_groups]
  count(api_groups_path) > 0

  resources := ["serviceaccounts/token", "*"]
  resources_path := [sprintf("%s.resources[%d]", [rule_path, l
    ]) | resource = rule.resources[l]; resource in resources]
  count(resources_path) > 0

  path := array.concat(resources_path, verb_path)
  path2 := array.concat(path, api_groups_path)
  finalpath := array.concat(path2, [
    sprintf("relatedObjects[%d].subjects[%d]", [j, k]),
    sprintf("relatedObjects[%d].roleRef.name", [j]),
  ])

  msga := {
    "alertMessage": sprintf("Subject: %s-%s can create
   serviceaccounts/tokens in kube-system", [subjectVector.kind
   , subjectVector.name]),
    "alertScore": 3,
    "failedPaths": finalpath,
    "fixPaths": [],
    "packagename": "armo_builtins",
    "alertObject": {
      "k8sApiObjects": [],
      "externalObjects": subjectVector,
    },
  }
}

# 1. rolebinding in kubesystem ns + role in kubesystem ns
# 2. rolebinding in kubesystem ns + clusterrole
can_create_to_pod_namespace(rolebinding) {
  rolebinding.metadata.namespace == "kube-system"
}

# 3. clusterrolebinding + clusterrole
can_create_to_pod_namespace(rolebinding) {
  rolebinding.kind == "ClusterRoleBinding"
}

# for service accounts
is_same_subjects(subjectVector, subject) {
  subjectVector.kind == subject.kind
  subjectVector.name == subject.name
  subjectVector.namespace == subject.namespace
}

# for users/ groups
is_same_subjects(subjectVector, subject) {
  subjectVector.kind == subject.kind
  subjectVector.name == subject.name
  subjectVector.apiGroup == subject.apiGroup
}
```

## A.9 Token request metadata

The metadata of the new rego rule that is used to assess the token request inside the privileged namespace for section 6.3

rules/rule-can-create-sa-token-kube-system/rule.metadata.json

```json
{
    "name": "rule-can-create-sa-token-kube-system",
    "attributes": {
      "microsoftK8sThreatMatrix": "Privilege Escalation::Cluster-admin binding",
      "armoBuiltin": false,
      "resourcesAggregator": "subject-role-rolebinding",
      "useFromKubescapeVersion": "v1.0.133"
    },
    "ruleLanguage": "Rego",
    "match": [
      {
        "apiGroups": [
          "*"
        ],
        "apiVersions": [
          "*"
        ],
        "resources": [
            "Role",
            "ClusterRole",
            "ClusterRoleBinding",
            "RoleBinding"
          ]
      }
    ],
    "ruleDependencies": [],
    "description": "determines which users can create serviceaccount/tokens in kube-system namespace",
    "remediation": "",
    "ruleQuery": "armo_builtins"
  }
```

## A.10 Create pod control

The new control that uses an existing rego rule for section 6.3
`controls/Minimize access to create pods in kube-system.json`

```
{
    "name": "Minimize access to create pods in kube-system",
    "id":"tp-0001",
    "controlID":"tp-0001",
    "description": "The ability to create pods in a privileged
   namespace provides a number of opportunities for privilege
   escalation. As such, access to create new pods should be
   restricted to the smallest possible group of users.",
    "long_description": "The ability to create pods in a
   cluster opens up possibilities for privilege escalation and
    should be restricted, where possible.",
    "remediation": "Where possible, remove ‘create‘ access to ‘
   pod‘ objects in the cluster.",
    "manual_test": "Review the users who have create access to
   pod objects in the Kubernetes API.",
    "test": "Check which subjects have RBAC permissions to
   create pods.",
    "attributes": {
        "armoBuiltin": false
    },
    "rulesNames": [
        "rule-can-create-pod-kube-system-v1"
    ],
    "baseScore": 6,
    "impact_statement": "Care should be taken not to remove
   access to pods to system components which require this for
   their operation"
}
```

## A.11 Token request control

The new control that uses the new rego rule for section 6.3
`controls/Minimize access to create serviceaccounts tokens.json`

```
{
   "name": "Minimize access to create serviceaccounts tokens",
   "id": "tp-0002",
   "controlID":"tp-0002",
   "description": "The Kubernetes API stores secrets, which may
    be service account tokens for the Kubernetes API or
   credentials used by workloads in the cluster. Access to
   these secrets should be restricted to the smallest possible
    group of users to reduce the risk of privilege escalation
   .",
   "remediation": "Where possible, remove ‘create‘ access to ‘
   serviceaccounts/token‘ objects in the cluster.",
   "manual_test": "Review the users who have `create` access to
    `serviceaccounts/token` objects in the Kubernetes API.",
   "test": "Check which subjects have RBAC permissions to
   create Kubernetes tokens.",
   "references": [
       "https://www.paloaltonetworks.com/apps/pan/public/
   downloadResource?pagePath=/content/pan/en_US/resources/
   whitepapers/kubernetes-privilege-escalation-excessive-
   permissions-in-popular-platforms"
   ],
   "attributes": {
       "armoBuiltin": false
   },
   "rulesNames": [
       "rule-can-create-sa-token-kube-system"
   ],
   "baseScore": 6,
   "impact_statement": "Care should be taken not to remove
   access to token creation of system components which require
    this for their operation"
}
```

## A.12 Trampoline pod framework

Our new *Kubescape framework* for section 6.3
`frameworks/trampoline_pods.json`

```
{
    "name": "Trampoline-pods",
    "description": "This framework highlights overzealous pod
    permisions which could lead to privilege escalation",
    "attributes": {
      "armoBuiltin": false
    },
    "controlsNames": [
        "Minimize access to create serviceaccounts tokens",
        "Minimize access to create pods in kube-system",
        "Ensure that the cluster-admin role is only used where
    required",
        "Minimize access to secrets",
        "Exec into container"
     ]
}
```

## A.13 Test runner for roleBinding

This section includes the Testrunner test for `rolebinding`. Description of the test running can be found in 7.1
`input/clusterrole.yaml`

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: test
rules:
- apiGroups: [""]
  resources: ["serviceaccounts/token"]
  verbs: ["create", "list"]
```

input/rolebinding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name:  serviceaccounts/token
  namespace: kube-system
subjects:
- kind: ServiceAccount
  name: tokener
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: test
  apiGroup: rbac.authorization.k8s.io
```

expected.json

```json
[
    {
        "alertMessage": "Subject: ServiceAccount-tokener can create serviceaccounts/tokens in kube-system",
        "failedPaths": [
            "relatedObjects[1].rules[0].resources[0]",
            "relatedObjects[1].rules[0].verbs[0]",
            "relatedObjects[1].rules[0].apiGroups[0]",
            "relatedObjects[0].subjects[0]",
            "relatedObjects[0].roleRef.name"
        ],
        "fixPaths": [],
        "ruleStatus": "",
        "packagename": "armo_builtins",
        "alertScore": 3,
        "alertObject": {
            "externalObjects": {
                "apiGroup": "rbac.authorization.k8s.io",
                "kind": "ServiceAccount",
                "name": "tokener",
                "relatedObjects": [
                    {
                        "apiVersion": "rbac.authorization.k8s.io/v1",
                        "kind": "RoleBinding",
                        "metadata": {
                            "name": "serviceaccounts/token",
                            "namespace": "kube-system"
                        },
                        "roleRef": {
                            "apiGroup": "rbac.authorization.k8s.io",
                            "kind": "ClusterRole",
                            "name": "test"
                        },
                        "subjects": [
                            {
                                "apiGroup": "rbac.authorization.k8s.io",
                                "kind": "ServiceAccount",
                                "name": "tokener"
                            }
                        ]
                    },
                    {
                        "apiVersion": "rbac.authorization.k8s.io/v1",
                        "kind": "ClusterRole",
                        "metadata": {
                            "name": "test"
                        },
                        "rules": [
                            {
                                "apiGroups": [
                                    ""
                                ],
                                "resources": [
                                    "serviceaccounts/token"
                                ],
                                "verbs": [
                                    "create",
                                    "list"
                                ]
                            }
                        ]
                    }
                ]
            }
        }
    }
]
```

## A.14 Test runner for roleBinding

This section includes the Testrunner test for clusterrolebinding. Description of the testrunner can be found in 7.1
`input/clusterrole.yaml`

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

```yaml
metadata:
  name: test
rules:
- apiGroups: [""]
  resources: ["serviceaccounts/token"]
  verbs: ["create", "list"]
```

input/clusterrolebinding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: create-tokens-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
- kind: Group
  name: dev
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: test
  apiGroup: rbac.authorization.k8s.io
```

expected.json

```json
[
    {
        "alertMessage": "Subject: Group-dev can create
    serviceaccounts/tokens in kube-system",
        "failedPaths": [
            "relatedObjects[1].rules[0].resources[0]",
            "relatedObjects[1].rules[0].verbs[0]",
            "relatedObjects[1].rules[0].apiGroups[0]",
            "relatedObjects[0].subjects[1]",
            "relatedObjects[0].roleRef.name"
        ],
        "fixPaths": [],
        "ruleStatus": "",
        "packagename": "armo_builtins",
        "alertScore": 3,
        "alertObject": {
            "externalObjects": {
                "apiGroup": "rbac.authorization.k8s.io",
                "kind": "Group",
                "name": "dev",
                "relatedObjects": [
                    {
                        "apiVersion": "rbac.authorization.k8s.
    io/v1",
                        "kind": "ClusterRoleBinding",
                        "metadata": {
                            "name": "create-tokens-global"
                        },
                    "roleRef": {
                        "apiGroup": "rbac.authorization.k8s.io",
                        "kind": "ClusterRole",
                        "name": "test"
                    },
                    "subjects": [
                        {
                            "apiGroup": "rbac.authorization.k8s.io",
                            "kind": "Group",
                            "name": "manager"
                        },
                        {
                            "apiGroup": "rbac.authorization.k8s.io",
                            "kind": "Group",
                            "name": "dev"
                        }
                    ]
                },
                {
                    "apiVersion": "rbac.authorization.k8s.io/v1",
                    "kind": "ClusterRole",
                    "metadata": {
                        "name": "test"
                    },
                    "rules": [
                        {
                            "apiGroups": [
                                ""
                            ],
                            "resources": [
                                "serviceaccounts/token"
                            ],
                            "verbs": [
                                "create",
                                "list"
                            ]
                        }
                    ]
                }
            ]
        }
    }
},
{
    "alertMessage": "Subject: Group-manager can create serviceaccounts/tokens in kube-system",
    "failedPaths": [
        "relatedObjects[1].rules[0].resources[0]",
        "relatedObjects[1].rules[0].verbs[0]",
        "relatedObjects[1].rules[0].apiGroups[0]",
        "relatedObjects[0].subjects[0]",
        "relatedObjects[0].roleRef.name"
    ],
    "fixPaths": [],
    "ruleStatus": "",
    "packagename": "armo_builtins",
    "alertScore": 3,
    "alertObject": {
        "externalObjects": {
            "apiGroup": "rbac.authorization.k8s.io",
            "kind": "Group",
            "name": "manager",
            "relatedObjects": [
                {
                    "apiVersion": "rbac.authorization.k8s.io/v1",
                    "kind": "ClusterRoleBinding",
                    "metadata": {
                        "name": "create-tokens-global"
                    },
                    "roleRef": {
                        "apiGroup": "rbac.authorization.k8s.io",
                        "kind": "ClusterRole",
                        "name": "test"
                    },
                    "subjects": [
                        {
                            "apiGroup": "rbac.authorization.k8s.io",
                            "kind": "Group",
                            "name": "manager"
                        },
                        {
                            "apiGroup": "rbac.authorization.k8s.io",
                            "kind": "Group",
                            "name": "dev"
                        }
                    ]
                },
                {
                    "apiVersion": "rbac.authorization.k8s.io/v1",
                    "kind": "ClusterRole",
                    "metadata": {
                        "name": "test"
                    },
                    "rules": [
                        {
                            "apiGroups": [
                                ""
                            ],
                            "resources": [
                                    "serviceaccounts/token"
                                ],
                                "verbs": [
                                    "create",
                                    "list"
                                ]
                            }
                        ]
                    }
                ]
            }
        }
    }
]
```

## A.15 RBAC Police evaluate token request

Output provided by RBAC Police when discovering the introduced vulnerability. This output is part of section 7.2.2.

```
-> ./rbac-police eval lib/token_request.rego
{
    "policyResults": [
        {
            "policy": "lib/token_request.rego",
            "severity": "Critical",
            "description": "Identities that can create TokenRequests (serviceaccounts/token)
    in privileged namespaces (kube-system) can issue tokens for admin-equivalent SAs",
            "violations": {
                "serviceAccounts": [
                    {
                        "name": "create-tokens",
                        "namespace": "default",
                        "nodes": [
                            {
                                "aks-agentpool-38837069-vmss000000": [
                                    "nginx-deployment-6544fdf46f-lmjtj"
                                ]
                            },
                            {
                                "aks-agentpool-38837069-vmss000001": [
                                    "nginx-deployment-6544fdf46f-8vc2n"
                                ]
                            }
                        ]
                    }
                ],
                "nodes": [
                    "aks-agentpool-38837069-vmss000000",
                    "aks-agentpool-38837069-vmss000001"
                ]
            }
        }
    ],
    "summary": {
        "failed": 1,
        "passed": 0,
        "errors": 0,
        "evaluated": 1
    }
}
```

## A.16 Kubescape trampoline pod framework

The output provided by Kubescape when using the new framework. This output is part of section 7.2.3.

```
  ./kubescape scan --use-from ../regolibrary/release/trampoline-pods.json -v
[info] Kubescape scanner starting
[warning] unknown build number, this might affect your scan results. Please make sure you are updated to latest version
[warning] Kubernetes cluster nodes scanning is disabled. This is required to collect valuable data for certain controls.
You can enable it using  the --enable-host-scan flag
[info] Downloading/Loading policy definitions
[success] Downloaded/Loaded policy
[info] Accessing Kubernetes objects
[success] Accessed to Kubernetes objects
[info] Scanning. Cluster: do-ams3-k8s-test-cluster
[success] Done scanning. Cluster: do-ams3-k8s-test-cluster

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

################################################################################
ApiVersion:
Kind: ServiceAccount
Name: create-tokens
Namespace: default

Controls: 5 (Failed: 1, Excluded: 0)

+----------+--------------------------------+-------------------------------------+-----------------------------------------+
| SEVERITY |          CONTROL NAME          |                DOCS                 |          ASSISTANT REMEDIATION          |
+----------+--------------------------------+-------------------------------------+-----------------------------------------+
| Medium   | Minimize access to create      | https://hub.armosec.io/docs/tp-0002 | relatedObjects[1].rules[0].resources[0] |
|          | serviceaccounts tokens         |                                     | relatedObjects[1].rules[0].verbs[0]     |
|          |                                |                                     | relatedObjects[1].rules[0].apiGroups[0] |
|          |                                |                                     | relatedObjects[0].subjects[0]           |
|          |                                |                                     | relatedObjects[0].roleRef.name          |
+----------+--------------------------------+-------------------------------------+-----------------------------------------+

################################################################################
ApiVersion:
Kind: ServiceAccount
Name: dosecret-operator
Namespace: kube-system

Controls: 5 (Failed: 1, Excluded: 0)

+----------+----------------------------+---------------------------------------+-----------------------------------------+
| SEVERITY |        CONTROL NAME        |                 DOCS                  |          ASSISTANT REMEDIATION          |
+----------+----------------------------+---------------------------------------+-----------------------------------------+
| Medium   | Minimize access to secrets | https://hub.armosec.io/docs/cis-5-1-2 | relatedObjects[1].rules[0].resources[1] |
|          |                            |                                       | relatedObjects[1].rules[0].verbs[2]     |
|          |                            |                                       | relatedObjects[1].rules[0].verbs[3]     |
|          |                            |                                       | relatedObjects[1].rules[0].verbs[6]     |
|          |                            |                                       | relatedObjects[1].rules[0].apiGroups[0] |
|          |                            |                                       | relatedObjects[0].subjects[0]           |
|          |                            |                                       | relatedObjects[0].roleRef.name          |
+----------+----------------------------+---------------------------------------+-----------------------------------------+

################################################################################
ApiVersion: rbac.authorization.k8s.io
Kind: Group
Name: k8saas:authenticated

Controls: 5 (Failed: 5, Excluded: 0)

+----------+--------------------------------+---------------------------------------+-----------------------------------------+
| SEVERITY |          CONTROL NAME          |                 DOCS                  |          ASSISTANT REMEDIATION          |
+----------+--------------------------------+---------------------------------------+-----------------------------------------+
| High     | Ensure that the cluster-admin  | https://hub.armosec.io/docs/cis-5-1-1 | relatedObjects[1].rules[0].resources[0] |
|          | role is only used where        |                                       | relatedObjects[1].rules[0].verbs[0]     |
|          | required                       |                                       | relatedObjects[1].rules[0].apiGroups[0] |
|          |                                |                                       | relatedObjects[0].subjects[0]           |
|          |                                |                                       | relatedObjects[0].roleRef.name          |
+----------+--------------------------------+---------------------------------------+                                         +
| Medium   | Exec into container            | https://hub.armosec.io/docs/c-0002    |                                         |
|          |                                |                                       |                                         |
|          |                                |                                       |                                         |
|          |                                |                                       |                                         |
|          |                                |                                       |                                         |
+          +--------------------------------+---------------------------------------+                                         +
|          | Minimize access to create pods | https://hub.armosec.io/docs/tp-0001   |                                         |
|          | in kube-system                 |                                       |                                         |
|          |                                |                                       |                                         |
|          |                               |                                      |                                           |
|          |                               |                                      |                                           |
+          +-------------------------------+--------------------------------------+                                           +
|          | Minimize access to create     | https://hub.armosec.io/docs/tp-0002  |                                           |
|          | serviceaccounts tokens        |                                      |                                           |
|          |                               |                                      |                                           |
|          |                               |                                      |                                           |
|          |                               |                                      |                                           |
+          +-------------------------------+--------------------------------------+                                           +
|          | Minimize access to secrets    | https://hub.armosec.io/docs/cis-5-1-2 |                                          |
|          |                               |                                      |                                           |
|          |                               |                                      |                                           |
|          |                               |                                      |                                           |
|          |                               |                                      |                                           |
+----------+-------------------------------+--------------------------------------+-------------------------------------------+

Controls: 5 (Failed: 5, Excluded: 0, Skipped: 0)
Failed Resources by Severity: Critical - 0, High - 1, Medium - 6, Low - 0

+----------+-----------------------------------------------------------------+------------------+--------------------+---------------+--------------+
| SEVERITY |                          CONTROL NAME                           | FAILED RESOURCES | EXCLUDED RESOURCES | ALL RESOURCES | % RISK-SCORE |
+----------+-----------------------------------------------------------------+------------------+--------------------+---------------+--------------+
| High     | Ensure that the cluster-admin role is only used where required |        1         |         1          |      66       |      2%      |
| Medium   | Exec into container                                             |        1         |         1          |      80       |      1%      |
| Medium   | Minimize access to create pods in kube-system                   |        1         |         7          |      80       |      1%      |
| Medium   | Minimize access to create serviceaccounts tokens                |        2         |         2          |      80       |      3%      |
| Medium   | Minimize access to secrets                                      |        2         |         10         |      80       |      3%      |
+----------+-----------------------------------------------------------------+------------------+--------------------+---------------+--------------+
|          |                        RESOURCE SUMMARY                         |        3         |         15         |      80       |    1.82%     |
+----------+-----------------------------------------------------------------+------------------+--------------------+---------------+--------------+
FRAMEWORK Trampoline-pods
```