Master thesis

Cyber Security

Radboud University

Identification and prevention of lateral movement in Kubernetes

Author:
Mike van Haren
s1019337

First supervisor:
dr. ing. P.J.M. Van Aubel
`pol.vanaubel@ru.nl`

Second assessor:
dr. ir. H.P.E. Vranken
`harald.vranken@ru.nl`

External supervisor:
Nathan Keyaerts
`Sue B.V.`
`nathan.keyaerts@sue.nl`

August 21, 2024

Summary

An increasing number of companies make use of cloud computing. Cloud orchestration tools exist to make this easier. Kubernetes is an orchestration tool that helps set up and manage cloud clusters. Using Kubernetes requires correct setup and configuration. If this is not done correctly, the cluster is susceptible to attacks. There are many scenarios where the cluster is misconfigured, leading to an attacker entering the cluster. After entering the cluster, the attacker can laterally move within the cluster to find sensitive information or gain control over some system. This research aims to find a way to identify and prevent lateral movement in the Kubernetes cluster.

We investigated whether `seccomp`, a Linux kernel sandboxing facility that can log and block system calls (syscalls), could be used to achieve this. Two clusters were set up: a cluster using Role-Based Access Control (RBAC) for security and a cluster using both RBAC and `seccomp`. These two clusters are tested with two attacks performed from a compromised `pod` in the cluster. Additionally, we investigated whether the performance of the attacks is affected by the privileged status of the compromised `pod`. The findings indicate that `seccomp` does not work in privileged `pods`. However, `seccomp` does work in unprivileged `pods`.

Although `seccomp` can be used to log and block syscalls, blocking syscalls is not possible in every scenario. If we block a syscall used in either the creation procedure or the regular usage of the `pod`, the `pod` cannot function correctly. While we can block syscalls outside of these, we cannot prevent attacks that only use unblockable syscalls.

While blocking is not possible in every scenario, logging is possible in every scenario. We encountered a problem, however, as the attacks only used syscalls that are also used during the regular usage of the `pod`. In such cases, it is difficult to distinguish between attacks and regular usage of the `pod`. A possible direction for future work might be to combine the logs with additional information to make distinguishing between attacks and regular usage easier.

Overall, we conclude that `seccomp` can be used in specific scenarios. Syscalls not required by the `pod` can be blocked. If lateral movement only consists of unblockable syscalls, the lateral movement cannot be prevented. Lateral movement that consists of blockable syscalls, however, can be prevented. Logging can be done in a broader range of situations, but combining it with additional information is necessary to distinguish between attacks and regular usage of the `pod`.

Contents

Listings

Chapter 1

Introduction

Currently, an increasing number of companies make use of the cloud [1]. One reason for this is that the cloud provides easy scalability of resources. If the demand for resources changes, the number of resources can easily be scaled up or down. The cloud can be seen as a network of virtual machines (VMs) that can perform a task or multiple tasks. This can be the same task, but it can also be a different task. Depending on the demand of a task, it is possible to activate more or fewer computers in the network. This could be arduous to perform manually, but cloud orchestration tools exist to help with this. One such tool is Kubernetes, which can help set up and manage cloud clusters. Kubernetes is used to easily manage, control, and update the network of machines in the cloud.

Using Kubernetes requires correct setup and configuration. Otherwise, the Kubernetes cluster is susceptible to attacks [2, 3]. There are various scenarios in which the cluster has some misconfiguration or exploitable default value that can be used by attackers [4]. An attacker could use a known vulnerability or a misconfigured port to access a `pod` in a Kubernetes cluster [5]. After gaining initial access, attackers can try to access other `pods` and `nodes` in the cluster, which is called lateral movement. This allows them to retrieve valuable data, compromise additional systems, or find a system where privilege escalation can be used [4]. To ensure that the Kubernetes cluster is secure, it is necessary to identify when lateral movement occurs in the cluster. When identifying such movement (or even the presence) of attackers, we must also find a way to prevent lateral movement. Therefore, the goal of this thesis is to answer the following question:

How can lateral movement in Kubernetes be identified and prevented?

We expect that `seccomp` can provide an answer to this question. `Seccomp` is a Linux kernel sandboxing facility that can log and block system calls (syscalls). Syscalls enable the interaction between an application and the underlying Linux kernel [6]. Any action performed by a `pod` in the cluster uses multiple syscalls. With `seccomp`, it is possible to log and block specific syscalls, restricting what the `pods` in the cluster are allowed to do. Therefore, `seccomp` might be useful to identify and prevent lateral movement in Kubernetes.

After describing the related work on cloud security and attack detection and prevention in Section 1.1, Chapter 2 shows essential information to understand cloud computing and Kubernetes. Chapter 3 explains Kubernetes security and the attacker model used for our experiment. Afterwards, Chapter 4 explains the experimental setup and the attacks used to test `seccomp` as our method for identifying and preventing lateral movement. The results of the experiments are shown in Chapter 5. Finally, chapters 6 and 7 discuss and conclude this research.

1.1 Related work

This section describes relevant research in cloud security. Moreover, we describe relevant research regarding detecting and preventing attacks based on logs and network traffic outside the scope of the cloud. We look at three areas of study:

- Studies focusing on decreasing the response time

  Some studies looked at decreasing the response time to attacks in the cloud [7, 8]. They created a prediction model, which looks at past and current events to predict upcoming events. Based on this prediction, the remediation steps could be pre-computed before a potential attack occurs. Correct predictions allow for a decreased response time.

- Studies focusing on syscalls

  Instead of decreasing the response time, it is also possible to look at syscalls and their logs (syslogs). One study trained a prediction model in the context of Kubernetes using the syslogs instead of event logs [9]. They pre-computed the remediation steps of a predicted attack to speed up the remediation process. Another study focused on using syscalls to detect specific attacks on Kubernetes containers [10]. They used machine learning to learn what syscalls are used in cryptomining. After understanding what syscalls were used, the system could be monitored for these syscalls. However, they mentioned that this solution is subject to obfuscation techniques. Instead of machine learning, another study used an existing tool named `Falco` [11]. `Falco` uses rules and checks if the behavior in the Kubernetes cluster adheres to the rules. If this is not the case, `Falco` creates an alert.

  Other studies also looked at syscalls outside the context of the cloud. One study focused on short sequences of syscalls in Unix [12]. Two other studies focused on using a Markov Model for the likelihood of the sequence of syscalls. One of them takes the context into account for the likelihood [13], whilst the other takes the arguments of the syscalls into account for the likelihood [14].

- Studies focusing on network traffic

  Aside from decreasing response time or focusing on syscalls, some studies looked at using network traffic to detect and prevent attacks. One study looked at monitoring the network traffic in the cluster using an additional container in every `pod` [15]. This container records the network traffic of the `pod` and sends it to an external sensor. Another study, however, claims that using such containers adds possible vulnerabilities [16]. Instead of sidecar containers, another study examined Kubernetes' built-in network policies [17]. These network policies can be used to regulate what incoming and outgoing network traffic is allowed.

These studies focused on various solutions for the detection and prevention of attacks. Our research focuses specifically on using syscalls to detect and prevent attacks in the cloud. We investigate whether it is possible to log and block syscalls using `seccomp`. This can help in the detection and prevention of attacks. In summary, the contributions of this work are as follows:

- An investigation of how lateral movement is performed in Kubernetes.
- A method (`seccomp`) for identifying and preventing syscalls.
- Experiments on two clusters, showing the impact of `seccomp`.

Chapter 2

Cloud and Kubernetes

To understand how to identify and prevent lateral movement by attackers in Kubernetes, it is essential first to understand how the cloud and Kubernetes work. Section 2.1 explains some basic information regarding the cloud. Afterwards, Section 2.2 elaborates on some concepts regarding cloud security. Finally, Section 2.3 explains the basics of Kubernetes.

2.1 Cloud basics

Before the cloud became widely used [1], the data was stored locally on the used device. Cloud computing delivers IT resources, both computing power and storage space, over the internet with pay-as-you-go pricing. This means there is no need to invest in data centers or physical servers, as they are owned and managed by the cloud provider. Using cloud computing, scaling the number of resources depending on demand is easier. The cloud enables access to stored data from anywhere, as long as there is an internet connection [18, 19].

This section introduces some advantages of cloud solutions over on-premise solutions, i.e., reliability, scalability, and elasticity. Additionally, some core cloud concepts are explained, i.e., load balancing, containers, and storage. These subsections are based on my interpretation of research papers on cloud computing [18, 19] and the 'AWS cloud practitioner' course [20].

2.1.1 Reliability, scalability, and elasticity

The cloud provides several advantages over on-premise solutions. On-premise solutions require local data centers and physical servers. They could lose the stored data when a physical or digital problem occurs. It is possible to use multiple data centers and servers isolated from each other to store the data reliably. If a problem occurs, another data center still holds the data. However, this could be expensive. When using cloud computing, buying local data centers for backups is unnecessary. Multiple storage locations generally provide reliability without excessive costs, both for storing data and for the availability of the application.

Additionally, on-premise solutions require a fixed number of resources to be chosen at the start. It is possible to decide on the maximum number of resources to achieve maximum availability. This is often unnecessary and results in many idle resources. Instead, it is possible to determine the average number of resources. In this case, the costs for idle resources are minimized. However, this results in a shortage of resources when demand rises above the average. In the cloud, it is possible to easily scale the resources up or down, depending on demand. By doing this, the costs can be minimized, and efficiency can be maximized.

2.1.2 Load balancing

Balancing the workload depends on the architecture. In cloud computing, a decoupled architecture is used. This means that the front end is decoupled from the back end. When requests arrive at the front end, the requests need to be balanced over different instances.[1] Afterwards, the requests need to be handled by the back end. In the decoupled architecture, the requests go through an intermediary, meaning there is no direct contact between the front and back end. The intermediary balances the requests over the available back-end instances. As a result, the intermediary is called a load balancer. As the load balancer distributes the load, the front-end instances do not need to be informed about the specific back-end instances, and vice versa.

2.1.3 Containers

To properly run an application, the application needs the code of the application. The application code also depends on the operating system (OS). Part of these dependencies differ when using a different OS. These elements must be set up correctly to ensure the application runs properly.

Containers are used to make this setup easier. A container takes the code and dependencies together as one self-sufficient package. As the container contains everything needed to run the application, it is possible to run the application on any system with the expected OS. Knowing what OS is used still matters when containers are used, as the foundation and possible dependencies differ. A significant advantage, however, is that the consistency of the underlying environment (aside from the OS) is less of a concern, as anything that is required is contained in the container.

Containers can be used to run applications in the cloud. An orchestration tool can be used when running multiple container replicas. One such tool is Kubernetes, which is explained in more detail in Chapter 3.

[1] Instances are virtual servers that run within a cloud environment.

2.1.4 Storage and databases

There are different options for storing data in the cloud: block storage, object storage, and file storage. Depending on the goal, a different method could be more beneficial.

Block storage

In block storage, data is stored as blocks. Only the necessary blocks are updated when part of the data is changed, while the other blocks stay the same. This kind of storage is useful when files need many small changes.

Object storage

In object storage, data is stored as objects. Every object consists of the data, metadata, and a key. In contrast to block storage, a change in the object requires the entire object to be updated and not only the changed part. In exchange for the less efficient modification of files, the data can be retrieved quicker and scaled (almost) infinitely.

File storage

In file storage, data is stored as blocks. In that regard, it is the same as with block storage. The difference is that block storage only makes the files available on one instance, whilst file storage makes the stored files available on multiple instances.

2.2 Cloud security

This section introduces the core concepts regarding the security of a cloud cluster. This concerns responsibility, user permissions, networking, compliance, and monitoring. These subsections are based on my interpretation of research papers on cloud computing [18, 19] and the 'AWS cloud practitioner' course [20].

2.2.1 Shared responsibility model

When using the cloud to run applications and store data, both the cloud provider and the customer running the applications are responsible for security. The cloud provider has to provide the security *of* the cloud and all the infrastructure it uses. The customer is responsible for the security *in* the cloud. This concerns using the correct settings, encrypting sensitive data, etc.

2.2.2 User permissions and access

In cloud computing, creating users, groups, and roles is possible. When creating such an entity, certain permissions can be granted. Specific users can be granted permissions to perform certain tasks, but granting these permissions to an entire group of users is also possible. Permissions can also be connected to roles. In this case, roles can be assigned to a user or group to provide permissions connected to the role. To ensure security, the principle of least privileges can be used, which means permissions should be minimized to what is strictly necessary.

2.2.3 Networking

Inside the cloud, it is possible to regulate what instances are publicly accessible and what instances are only privately accessible. This can be achieved through a Virtual Private Cloud (VPC). Inside the VPC, different subnets can be created to group instances together. Combined with network rules, these subnets can determine who can access the instances. Instances that should be publicly accessible are placed in a public subnet. Instances that should not be publicly accessible are placed in a private subnet and can only be accessed from the private network. For instance, front-end instances need to be accessed by the public and are placed in the public subnet. At the same time, back-end instances, inaccessible to the public, are placed in the private subnet.

2.2.4 Compliance

When using the cloud, it is crucial to think about compliance. If data is not allowed to leave the country, it is necessary to restrict what data centers are used to run the application or store the data. Customers can specify where the data can(not) be stored, and the cloud provider uses data centers that match the requirements. If this is not done properly, compliance requirements could be violated.

2.2.5 Monitoring and analytics

Monitoring can be used to observe the cluster of instances. This is useful for analyzing the normal usage of the cluster and seeing if things can be improved. Monitoring can also give insight into when an attacker gets into the cluster. Subsection 3.1.5 explains more about how monitoring and logging are important in security and how they can be done in Kubernetes.

2.3 Kubernetes basics

Kubernetes is a container orchestration tool that helps deploy, manage, and update complex distributed systems in the cloud. It can provide high availability of services through automatic recovery in case of failures. Additionally, it enables the user to scale up or down quickly, depending on the situation. A Kubernetes cluster consists of different `nodes` that belong to one of two categories: head `nodes` on the control plane and worker `nodes` on the data plane. Head `nodes` allow the user to control the cluster and manage what happens, whilst the worker `nodes` run the applications for which the cluster is created. A basic architecture of a Kubernetes cluster can be seen in Figure 2.1. The architecture shows one head `node` and two worker `nodes`. Inside the `nodes`, the components of the specific `nodes` can be seen. The main objects and node components are explained in the subsections below. These subsections are based on my interpretation of various sources: research papers on Kubernetes [3, 5], books on Kubernetes operations [21, 22], the 'A Cloud Guru Certified Kubernetes Administrator' course [23], and some websites about Kubernetes and Docker [24, 25].

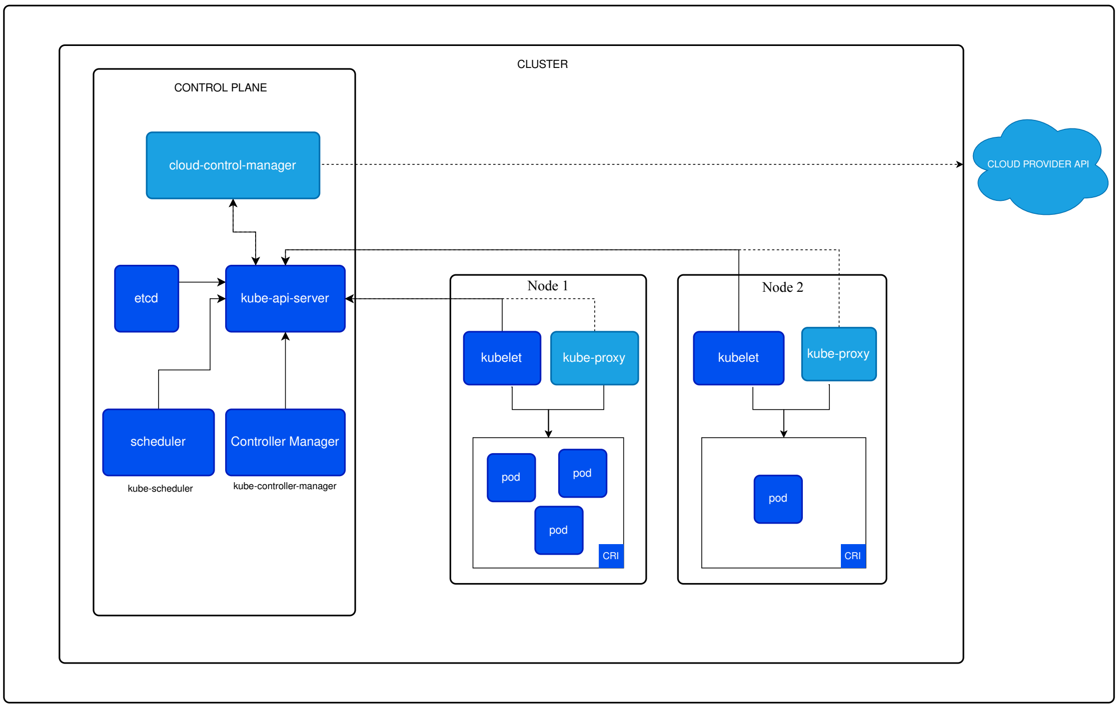

Figure 2.1: The architecture of a Kubernetes cluster with one head `node` *and two worker* `nodes`*.*[2]

2.3.1 Kubernetes objects

In Kubernetes, a variety of objects is used. These objects are used to set the desired state of the cluster. Some important objects are `node`, `pod`, `namespace`, `replicaSet`, `daemonSet`, `deployment`, and `service`. For the sake of clarity, we cover all of these objects. For our experiment, it suffices to understand `nodes`, `pods`, and `namespaces`.

Node

A `node` is a VM with the basic necessities to run containers. There are head `nodes` and worker `nodes`, which will be explained in more detail in subsections 2.3.2 and 2.3.3. In short, a head `node` has a manager role, whilst the worker `nodes` perform the tasks that need to be done. The worker `nodes` have `pods` running on them to run these tasks. It is possible to add restrictions regarding which `pods` can run on a `node`. This can be done

[2]This image comes from `https://kubernetes.io/docs/concepts/architecture/` under the license: Creative Commons Attribution 4.0 International license. It is created by the Linux Foundation®.

by adding a taint to the `node`, which means no `pod` will be scheduled on the tainted `node`.

Pod

A `pod` is a collection of one or more containers (which are explained in Subsection 2.1.3). An advantage of multiple containers in a `pod` is that the containers inside the same `pod` share storage, specifications, and IP address. Having the same IP address means the containers are considered the same system from a networking aspect, leading to easy communication between the containers.

Even though having multiple containers in the same `pod` provides advantages, it results in disadvantages regarding controlled scaling. Generally, scaling up or down is done by changing the number of `pods`. If one `pod` contains both a web server container (front end) and a database container (back end), they cannot be scaled separately. When more front-end instances are needed because of an increased number of simultaneous users, the `pod` is duplicated. Assuming the database is still sufficient, increasing the number of back-end instances is unnecessary. In this case, however, the number of front-end and back-end instances are both increased, whilst the goal was only to increase the number of front-end instances. This leads to less control and higher costs.

To have more fine-grained control, it is generally advised to only have one container in a `pod`. This way, it is possible to scale resources more precisely. For example, in the situation above, duplicating the `pod` containing only the web server results in more front-end instances, without changing the number of back-end instances. This achieves the desired result without the additional costs of unnecessarily increasing the number of back-end instances.

Namespace

There is always a default `namespace` present in a cluster. It is, however, possible to create additional `namespaces`. A `namespace` is a cluster within a cluster. The objects within a `namespace` are isolated from objects outside of the `namespace`. This enables proper organization and grouping within a cluster. A `namespace` can be used to have separate environments (e.g. development environment and production environment).

ReplicaSet

Kubernetes is generally used to run multiple container replicas, as there is no need to use Kubernetes when only one container is necessary. As mentioned above, these containers run on `pods`. A `replicaSet` specifies the desired number of `pod` replicas. The `pods` that are created and maintained by the `replicaSet` follow the `pod` template specified in the `replicaSet spec` field. This template includes the label that specifies what `pods` are part of the `replicaSet`. This means if the desired number of replicas is set to three, there will always be three replicas. If one gets terminated or fails, a new one is created to return to three. Similarly, if too many replicas are present with the specified label, the surplus is deleted to get down to three.

DaemonSet

A special version of `replicaSet` is the `daemonSet`. A `daemonSet` creates a replica of the specified `pod` on every `node`. This takes into account the special scheduling rules; if a `pod` would typically not be scheduled on a `node` (because of taints and lack of tolerations), this will still not happen when using a `daemonSet`.

Deployment

A `deployment` provides an abstraction over `replicaSets`. On top of the functionality that `replicaSets` provide, `deployments` provide functionality such as scaling, rolling updates, and rolling back to previous versions. Because of the added functionality and level of abstraction, `deployments` are typically used instead of `replicaSets`.

Service

A `service` adds an abstraction layer on top of `pods`. A `service` is used to expose an application without the need to understand exactly which `pods` are used for this application. This is useful as `pods` are replaced with other `pods` in case of failure. These new `pods` have different IP addresses, which can be problematic to keep track of. Using a `service` solves this problem. By providing all the `pods` with a proper label, the `service` can reroute any traffic for the application to any of the `pods` with the corresponding label. The `service` automatically tries to balance the load over the available `pods`. This shows that using `services` removes some complexity. There are four types of `services`:

1. `ClusterIP services` expose applications inside the cluster network. Only other `pods` inside the cluster can access the application.

2. `NodePort services` expose applications outside the cluster network. Users and applications can access the application from outside the cluster.

3. `LoadBalancer services` expose applications outside the cluster network. The difference with `NodePort services` is that `LoadBalancer services` use an external cloud load balancer, which only works when the cloud platform supports this functionality.

4. `ExternalName services` use the `externalName` field. The `externalName` field contains the Domain Name System (DNS) name on which the application can be accessed. The application runs outside the cluster and can normally be accessed through this (lengthy) DNS name. By using this `service`, the application can be accessed without the need for the entire DNS name.

2.3.2 Head nodes

Head `nodes` (also known as control plane `nodes`) allow the user to control the cluster. Head `nodes` manage the worker `nodes` in the cluster and act as the brain of the cluster. Depending on the size of the cluster, the number of head `nodes` might vary. Generally, the number is equal to three or five. On the head `nodes`, there are five components: `API server`, `etcd`, `scheduler`, `controller manager`, and `cloud controller manager`.

API server

The API server is the front end of the Kubernetes control plane. It receives requests such as YAML[3] [26] configuration files to state the desired state of the application. These requests are authenticated, authorized, and processed. Afterwards, they are stored in etcd to be processed and used. All requests to the control plane come through here.

etcd

The entire cluster's configuration and state are stored in etcd (also known as the cluster store). The etcd system contains the key-value store for the entire Kubernetes cluster. Additionally, it provides optimistic concurrency, ensuring that race conditions and overwriting changes made by other `nodes` cannot occur.

Scheduler

After creating Kubernetes objects such as `pods` and `deployments`, they still need to run on `nodes` in the cluster. The scheduler is responsible for finding a proper `node` to run the task. This process considers multiple aspects such as `node` taints, (anti-)affinity rules, available resources, etc. The scheduler looks for unscheduled objects and finds the best `nodes` to run them.

Controller manager

The controller manager contains Kubernetes-specific logic. As mentioned before, Kubernetes has automatic recovery. This is done through reconciliation control loops. The controller manager executes these loops. This is necessary for some objects to function correctly, such as `replicaSets`, `deployments`, and `services`. When more replicas are necessary, the controller manager enforces the creation of additional replicas. All actions performed by the controller manager are to achieve the cluster's desired state.

Cloud controller manager

The cloud controller manager contains cloud-specific logic. The exact logic depends on the underlying cloud used for the cluster. They are connected to the cloud provider's API. This enables Kubernetes to be cloud-agnostic and function correctly with multiple

[3] Some examples of YAML files can be seen later in this research, starting from Listing 3.2.

cloud providers. For instance, when creating a `node`, the information about the `node` needs cloud-specific information. In such a case, the cloud controller manager interacts with the cloud provider's API [27].

2.3.3 Worker nodes

Aside from head `nodes`, there are also worker `nodes` (also known as data plane `nodes`). The head `node` manages the worker `nodes`, whilst the worker `nodes` perform the work required in the cluster. Every `node` can run one or multiple `pods`. On all `nodes`, there are some essential components: `kubelet`, `kube-proxy`, and `container runtime`.

Kubelet

Kubelet is an agent that runs on every `node`. It communicates with the API server on the head `node` about the `node`'s status. Additionally, it updates the head `node` regarding the containers currently running on the `node` and what containers should be running there. If this does not match, kubelet runs the reconciliation loop and informs the head `node` about it.

Kube-proxy

Kube-proxy is an agent that runs on every `node`. It enables all containers, `pods`, and `nodes` to communicate without problems. Additionally, it handles the routing and load-balancing of tasks that need to be performed by `pods`.

Container runtime

The container runtime allows the direct execution of containers in a cluster. There are various container runtimes such as Docker or containerd. The container runtime must comply with the Container Runtime Interface (CRI).

Chapter 3

Kubernetes security and the attacker model

This chapter details Kubernetes' security, syscalls, and the attacker model. Section 3.1 explains Kubernetes' security and the required components to achieve it. Subsequently, Section 3.2 provides information on the syscalls and how monitoring these can help identify and prevent attacks. Finally, Section 3.3 explains the attacker model, which is necessary to understand our experimental setup in Chapter 4.

3.1 Kubernetes security

This section introduces the core concepts regarding the security of Kubernetes, i.e., cluster setup, hardening, supply chain security, runtime security, and monitoring and logging. These subsections are based on my interpretation of various sources: research papers on Kubernetes security [2, 15], books on Kubernetes operations [21, 22], and the 'A Cloud Guru Certified Kubernetes Security Specialist' course [28].

3.1.1 Cluster setup

In a Kubernetes cluster, various components must be installed and work together. Kubernetes makes it easier to manage all this, but there is still room for errors and wrong configuration. This subsection goes into detail about `networkPolicy`, `CIS benchmark`, and `binary verification`.

NetworkPolicy

When setting up a cluster, communication between different components is vital. Kubernetes has the `NetworkPolicy` object to ensure proper communication whilst considering security. A `NetworkPolicy` is an object that controls the flow of network communication inside the cluster. The `NetworkPolicy` can be used to define what traffic is or is not allowed for certain `pods`. This concerns both incoming and outgoing traffic. This shows that a `NetworkPolicy` can isolate `pods` from unneeded traffic.

CIS benchmark

The Center for Internet Security (CIS) benchmark is a set of standards and best practices regarding cluster setup. Kube-bench is a tool that checks the Kubernetes cluster against the CIS benchmark. This makes clear how well the cluster is set up. The output of kube-bench provides possible steps that can be taken to improve the cluster.

Binary verification

When installing Kubernetes binaries[1] manually, checking if they are tampered with is generally advised. Not all binaries found online are secure. The official Kubernetes website provides checksum files to check if the used binary was secure or tampered with. By performing this check, it is possible to ensure that the installed binaries are secure.

3.1.2 Hardening

The components in a Kubernetes cluster have specific permissions. These permissions enable the components to perform the tasks they need to perform. There is a risk, however, that components have more permissions than they need. Hardening the cluster focuses on minimizing the permissions to only the required ones. This subsection goes into detail about `seccomp`, `serviceAccount`, `Role-Based Access Control`, `standard ports`, `Kubernetes updates`, `host OS security`, and `AppArmor`.

Seccomp

`Seccomp`, a Linux kernel sandboxing facility that can log and block syscalls, is easily integrated with Kubernetes. There are two versions of `seccomp`: original `seccomp` and the enhanced version `seccomp-BPF`. Kubernetes supports the latter, which provides more freedom in specifying what syscalls are (not) allowed. For conciseness, in this paper, we refer to `seccomp-BPF` with `seccomp`.

`Seccomp` uses profiles with rules regarding what syscalls are (not) allowed in a `pod`. When creating a `pod`, the `seccompProfile` needs to be specified. Depending on the profile, it is possible to log or block the specified syscalls made by the `pod` [29, 30]. When blocking syscalls, it is essential to consider what syscalls are necessary for the `pod` to function correctly. If they are needed by the `pod`, blocking these syscalls is not possible, but it is possible to log them.

Different approaches are possible. Listing 3.1 shows an allowlist approach, where the default action is to return an error, and only the specified syscalls are allowed. Some syscalls are replaced with dots to shorten the example. Only the specified syscalls are allowed in the allowlist approach, whilst any other syscall is blocked using the default

[1] Binaries are machine-readable files that a computer needs to execute a program.

action. Attackers cannot exploit any blocked syscalls. This approach requires figuring out what syscalls should be allowed [31].

Aside from an allowlist approach, a blocklist approach allows the action by default, and only the specified syscalls are blocked. In this approach, only the syscalls that are known to be exploitable would be blocked. A disadvantage, however, is that attackers might find a way to utilize other syscalls for their attacks. It is possible to forget about exploitable syscalls or not know about specific exploits yet.

Listing 3.1: Allowlist `seccomp` profile where only the specified syscalls are allowed.

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "syscalls": [
        {
            "names": [
                "accept4",
                "epoll_wait",
                "pselect6",
                "futex",
                "madvise",
                "epoll_ctl",
                "getsockname",
                ...
                "setitimer",
                "writev",
                "fstatfs",
                "getdents64",
                "pipe2",
                "getrlimit"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

ServiceAccount

To give `pods` access to the Kubernetes API, Kubernetes uses the `serviceAccount` object. If an attacker gains access to such a `serviceAccount`, depending on the permissions of the `serviceAccount`, the attacker can access the Kubernetes API. To ensure this cannot easily happen, it is crucial to adequately define the permissions of the `serviceAccount`. The first step to achieve this is to keep the permissions of `serviceAccounts` minimal. This includes splitting up permissions over multiple `serviceAccounts` and not giving multiple permissions to a single `serviceAccount`. Role-Based Access Control (RBAC) is used to define a `serviceAccount`'s permissions.

Role-Based Access Control

Kubernetes makes use of RBAC. This means that permissions can be assigned to specific (Cluster)Roles. These (Cluster)Roles can be bound to users, groups, or `serviceAccounts` through a (Cluster)RoleBinding. A Role specifies permissions only inside the specified `namespace`. A ClusterRole specifies permissions that work in the entire cluster, regardless of the `namespace`.

By using RBAC properly, it can be ensured that every user, group, and `serviceAccount` has the proper permissions without having unneeded permissions. As it is possible to bind multiple (Cluster)Roles to a single entity, the permissions can be assigned to separate (Cluster)Roles. This provides fine-grained control over providing only the required permissions. This is coherent with the principle of least privilege mentioned in Subsection 2.2.2.

Standard ports

Kubernetes uses some standard ports. An attacker could try to see if such standard ports are open and if they can be exploited. The impact of such attacks can be diminished by using network segmentation and firewalls. According to the official Kubernetes website, some standard ports are [32]:

- 6443: Kubernetes API server
- 2379-2380: etcd
- 10250: kubelet API
- 10259: kube-scheduler
- 10257: kube-controller-manager
- 30000-32767: NodePort services

Kubernetes updates

One of the basic dangers of using software is unpatched vulnerabilities because of outdated software. Older versions could contain vulnerabilities that have not been patched yet. Therefore, it is important to keep Kubernetes up to date.

Host OS security

When creating a `pod`, the containers are specified in the `spec` field. The containers run in the container environment by default. This isolates the container from the host[2] with no access to the host's resources. It is possible, however, to run containers in the host environment. It is possible to activate privileged mode on the `pod` level. Additionally, it is possible to allow access to the host's resources on the container level through the `hostIPC`, `hostNetwork`, and `hostPID` fields. These options provide access to the host's resources, which can be useful for actions such as monitoring. Doing so, however, entails security issues and should only be done when absolutely necessary.

[2] In case of Kubernetes, the host is the `node`

AppArmor

AppArmor is a Linux security kernel module. It provides granular access control for programs running on Linux systems and can be used to control and limit what a program can do in the host OS. AppArmor uses profiles, which are sets of rules defining what a program can or cannot do. There are two modes:

1. Complain mode:
   Generate a report of what the program is doing without actually preventing the program from doing these things.

2. Enforce mode:
   Prevent the program from doing anything the profile does not allow.

To use AppArmor, profiles must be enabled on all `nodes`. A `pod` that uses AppArmor cannot be started if this is not enabled.

3.1.3 Supply Chain Security

Aside from cluster setup and hardening, it is also important to consider the security of third-party software. Various aspects must be checked and validated. This subsection goes into detail about `images`, `allowlisting registries`, `static analysis`, and `vulnerability scanning`.

Images

Images are the blueprints of containers. Images contain many software components. Any piece of software can contain vulnerabilities that an attacker could exploit. To reduce the odds of vulnerabilities, removing any unnecessary software and having up-to-date versions is vital. Aside from software vulnerabilities, it is also essential to know where the images come from. Attackers could create images with malicious software, which should be avoided.

Additionally, it is possible to validate that the actual images are not tampered with. Kubernetes allows appending the hash to the image inside the container `spec` field. If this hash is correct, there is no problem. If this hash is incorrect, the image has been tampered with. In this case of the latter, the `pod` is created, but the image is not downloaded.

Allowlisting registries

An image registry is a service that stores container images. It can be used to download container images to a cluster quickly. When running a container, the `node` automatically downloads the image from the registry. However, if an attacker controls such a registry, the images cannot be trusted to be secure. To prevent this, it is possible to make a list of what registries are allowed to download images from. This can be done by using OPA Gatekeeper, which is explained in Subsection 3.1.4.

Static analysis

Static analysis means analyzing the source code and Dockerfiles used to create images. Through the analysis, potential security issues can be found. There are some factors to watch out for in the Dockerfile:

1. Ensure that the last USER created in the Dockerfile is not root. If it is root, the entire container process runs as root.
2. Specify the specific version instead of using the `:latest` tag in the FROM directive. Its version is unclear if the `:latest` tag is used.
3. Make sure that the Dockerfile does not install unnecessary software or tools. These only provide additional software with possible vulnerabilities without adding actual helpful functionality.
4. Use Kubernetes secrets to pass sensitive data (e.g. passwords, API keys) to the container at runtime. If sensitive data is stored in the image, it is easier for an attacker to access it.

Aside from checking the Dockerfile, it is also possible to perform static analysis on Kubernetes resources. Files such as the YAML manifests are used to create resources. When checking such files, there are some factors to watch out for:

1. Avoid running as root.
2. Specify the specific version instead of using the `:latest` when specifying the image version. If the `:latest` tag is used, its version is unclear, and newer (unchecked) versions may be automatically downloaded.
3. Ensure containers do not use host `namespaces` or privileged mode unless absolutely necessary. If an attacker compromises the container, there is no direct possibility for him to attack the host.

Vulnerability scanning

A Kubernetes cluster contains many containers and images, all containing software. When a lot of software is used, the odds of vulnerabilities being present increase [33]. Vulnerability scanning is used to scan for known vulnerabilities. In Kubernetes, tools such as Trivy can be used. Trivy scans the cluster for vulnerabilities and creates a report listing the vulnerabilities, where they are, and their risk.

It is possible to automate vulnerability scanning by using an admission controller. Admission controllers intercept requests to the Kubernetes API and approve, deny, or modify them. One such controller is the `ImagePolicyWebhook` controller. This controller intercepts the request of creating a `pod` and scans the used image on vulnerabilities.

3.1.4 Runtime security

Another important aspect of improving Kubernetes' security is to secure the runtime. This subsection goes into detail about `securityContext`, `OPA gatekeeper`, `secret`, and `runtime sandbox`.

SecurityContext

In the specification of a `pod`, there is a field called `securityContext`. This field allows special security and access control settings for the `pods`. There is also a `securityContext` field in the specification of containers. This field allows special security and access control settings for the specific container. If there are multiple containers in a single `pod`, other containers are not influenced by this. The different levels provide a separate list of settings that can be set, where some settings are present in both lists.

OPA Gatekeeper

The Open Policy Agent (OPA) Gatekeeper allows for enforcing highly customizable policies on any Kubernetes object. These policies are defined using the OPA constraint framework. For example, it is possible to force all `pods` to specify resource limits. If a `pod` is created without the resource limits, it is denied until it specifies the resource limits. Similarly, it can be used to list what image registries are allowed or not, as mentioned in Subsection 3.1.3.

Secret

`Secrets` can store sensitive data in a key-value map format where the value is base64 encoded. This data can be passed to containers at runtime by using an environment variable or mounted volume. The sensitive data is secured better by using `Secrets`.

Runtime sandbox

A runtime sandbox provides a specialized runtime. This runtime has additional layers of isolation and greater security but (usually) has reduced performance. Any workload that is not trusted can be run on the runtime sandbox to ensure it does not impact any other part of the system. Some examples are `gVisor` or `Kata containers`. Both of these create a sandbox to run the application.

3.1.5 Monitoring and logging

Aside from securing and hardening the cluster, it is essential to have measures in place for when an attacker still gets in, as was mentioned in Subsection 2.2.5. By using monitoring and logging, it is possible to visualize what happens inside the cluster. This can help recognize when an attacker is attacking the cluster. This subsection goes into detail about `behavioral analytics` and `audit logging`.

Behavioral analytics

Observing what is going on in the cluster and identifying abnormal events is important. This can be done manually, but some tools can help with this. One tool that can help is `Falco` [34]. `Falco` monitors Linux syscalls and generates alerts about suspicious activity. Rules are used to decide what activity is considered suspicious. Additionally, it is possible to specify what information should be included in the alert, meaning it is possible to create a tailor-made alert with the knowledge required for remediation. `Falco` is explained in more detail in Subsection 3.2.2.

Audit logging

Audit logs are chronological records of events in the Kubernetes cluster. This can be used for both real-time threat detection and post-incident analysis. In Kubernetes, the audit policy can be set up as required and every rule can be defined as desired. It is possible to specify the level of detail of the logs. Additionally, it is possible to define what Kubernetes objects the rules apply to and in what `namespace`. A subset of audit logging is syscall logging.

3.2 Kubernetes syscall monitoring

Syscalls enable the interaction between an application in the Kubernetes cluster and the underlying Linux kernel [6]. A syscall is performed whenever an application performs an action. This holds for all actions and is a good measure for identifying attacks. For example, some attacks spawn a new process from inside the container. This makes use of the `execve` syscall [35, 36]. Similarly, the `openat` syscall is used to open a file in a certain location through a specified path [37]. Besides, there are also attacks where the attacker escapes the container and changes the root directory using the `chroot` syscall [16, 38].

Monitoring the Kubernetes cluster is necessary to ensure proper security [39, 40]. This can be done through various methods, such as network traffic monitoring or syscall monitoring. Our research focuses on syscall monitoring, for which the reason is discussed in more detail in Section 4.1. Monitoring the syscalls needs to be done on the Linux kernel level. This can be done through `eBPF`, which is explained in Subsection 3.2.1. Afterwards, Subsection 3.2.2 mentions `eBPF`-based tools that can be used for syscall monitoring in Kubernetes.

3.2.1 Extended Berkeley Packet Filter

Extended Berkeley Packet Filter (`eBPF`) can extend the kernel's capabilities without tinkering with the kernel source code itself. Instead, `eBPF` programs are written in bytecode. These programs are loaded into the kernel when an event triggers a hook, without changing the kernel source code. A hook is a kind of sensor. Aside from using pre-defined hooks, such as hooks for syscalls or function entry/exit, custom hooks are also possible. When a process triggers an event where a hook is placed, the eBPF program is run [41, 42].

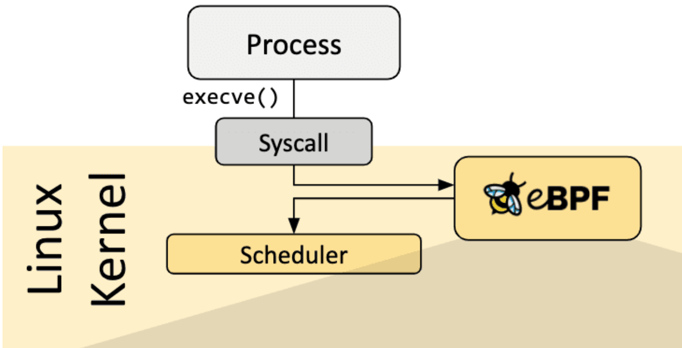

Figure 3.1: An eBPF hook on the execve syscall.[3]

As can be seen in Figure 3.1, it is possible to put a hook specifically on the execve syscall. Before this syscall is executed, the hook is triggered. This causes the eBPF program to be executed first. Afterwards, the actual execve syscall is executed.

By using eBPF, there are a lot of possibilities that could be explored. For our research, monitoring and filtering network traffic could be interesting. Moreover, eBPF can help observe and visualize what is happening in the cluster. The information gained from this makes it possible to kill malicious processes or restrict actions.

3.2.2 Syscall tools

Various tools based on eBPF are available to monitor the syscalls. Every tool has its pros and cons. Some tools are easier to integrate with Kubernetes than others. One tool that is integrated with Kubernetes is seccomp [29], which was explained in Subsection 3.1.2. Aside from seccomp, an extensive list of tools that can be integrated with Kubernetes is found on [43]. These tools could be helpful, but they focus on monitoring the cluster at the application level. Instead, we want to focus on the underlying syscalls on the kernel level.

Another list containing eBPF-based tools that focus on the underlying syscalls is found on [44]. This list contains tools not made with Kubernetes in mind, making the integration harder. Other tools on the list use eBPF to look at network traffic. The list contains various options that can work properly in Kubernetes and can be used for syscall monitoring, but it is infeasible to investigate all of them in-depth. Therefore, aside from seccomp, we only looked at Falco [34].

[3] This image comes from https://ebpf.io/what-is-ebpf/ under the license: Creative Commons Attribution 4.0 International License.

Falco

Similarly to `seccomp`, `Falco` is a tool that also uses rules. `Falco` only provides possibilities for logging when rules are violated, without the possibility to block them. A `rules.yml` file can be created. When running `Falco` with the specified rules, the behavior in the cluster is checked on whether these rules are violated. By logging any action that violates the rules, `Falco` is a useful tool for detecting attackers in the cluster. This allows abnormal behavior to be visualized and the generation of alerts in real time [34].

It is possible to use `Falco` rules for Kubernetes audit logs or syscall logs [45]. Whenever `Falco` observes an event, the rules are checked to see if this event is allowed or not. The desired output format can differ depending on the implementation and how the following process needs to be done.

An example rule can be seen in Listing 3.2 [46]. This rule checks if a non-shell program spawned a shell.[4] Macros can be used to make the rule more readable. A macro functions like a variable that can be used in the rule.

Listing 3.2: A `Falco` rule in YAML that checks if a shell was spawned.

```yaml
- macro: container
  condition: container.id != host

- macro: spawned_process
  condition: evt.type = execve and evt.dir=<

- rule: run_shell_in_container
  desc: a shell was spawned by a non-shell program in a container.
    Container entrypoints are excluded.
  condition: container and proc.name = bash and spawned_process and proc.
    pname exists and not proc.pname in (bash, docker)
  output: "Shell spawned in a container other than entrypoint (user=%user.
    name container_id=%container.id container_name=%container.name shell=%
    proc.name parent=%proc.pname cmdline=%proc.cmdline)"
  priority: WARNING
```

3.3 Attacker model

Our research aims to find a way to detect and prevent lateral movement in a Kubernetes cluster. Lateral movement occurs when an attacker is already inside the cluster. Because of this, the original point of entry into the cluster is out of the scope of our research. All attack attempts in the scope of our research are based on the assumption that the attacker has compromised a `pod` in the cluster. This means the attacker has root-level filesystem access in the compromised `pod`, providing the attacker complete control over the filesystem in the compromised `pod`, as well as the capability to utilize all the permissions of the compromised `pod` to interact with the Kubernetes cluster.

Lateral movement can be done on `pod` and `node` level. On `pod` level, this means that the attacker can gain access to another `pod` from the compromised `pod`. On `node` level,

[4] A shell enables interaction with the OS by entering and executing text commands.

the attacker can gain access to another `node` from the compromised `pod`. To ensure that the `node` level lateral movement is consistent, the compromised `pod` is on the first worker `node` of the cluster for every attack.

We assume the attacker compromises a `pod` with the permissions to perform every possible action on `pods`, `pods/exec`, and `pods/log` in the `developer namespace`. This choice is motivated by the scenario from [38], where this is claimed to be a common setup. The possible actions for these three resources are as follows [47]:

- `pods`: `create`, `delete`, `deletecollection`, `get`, `list`, `patch`, and `update`. These actions enable the compromised `pod` to create, delete, change, and inspect `pods`.
- `pods/exec`: `create` and `get`. These actions enable the compromised `pod` to use the `kubectl exec POD -- COMMAND` command. This executes the `COMMAND` in the `POD` [48].
- `pods/log`: `get`. This action enables the compromised `pod` to use the `kubectl logs POD` command. This outputs the logs of the `POD`.

Aside from the permissions to perform these actions, the compromised `pod` also has the permissions to `get` and `list pods` and `namespaces` cluster-wide. These permissions enable the compromised `pod` to inspect all `pods` and `namespaces` in the cluster.

Two different scenarios are considered concerning the compromised `pod`:

1. The attacker has compromised a privileged `pod`.
2. The attacker has compromised an unprivileged `pod`.

A privileged `pod` can access the host's resources and kernel capabilities. This is not the case in an unprivileged `pod`. This could potentially impact the attack and defense capabilities.

In conclusion, for our research, we assume that the attacker has root-level filesystem access within the compromised `pod`. The attacker can utilize all the permissions of the compromised `pod`. The attacks are tested from a privileged and unprivileged compromised `pod`.

Chapter 4

Methods

This chapter explains the methods used in our research to investigate the identification and prevention of lateral movement in Kubernetes. Section 4.1 explains the pros and cons of two different monitoring methods and which one is used in the proposed solution. Afterwards, Section 4.2 describes the experiment's setup. Finally, Section 4.3 explains the two attacks performed during the experiment.

4.1 Network traffic monitoring versus syscall monitoring

Two methods that could be used for monitoring were considered. Subsection 4.1.1 describes how network traffic monitoring in the Kubernetes cluster could be done, including an explanation of why this method is not chosen. Subsection 4.1.2 describes how syscall monitoring in the Kubernetes cluster could be done, explaining why this method is chosen over the alternative.

4.1.1 Network traffic monitoring

This method monitors the network traffic between `pods`. The idea is to monitor the cluster's regular traffic between `pods` for a while to create a baseline. Afterwards, the traffic is continuously monitored to check whether the traffic differs from the baseline. If the monitored traffic differs too much from the baseline, it is seen as an anomaly, and an alert is created.

Our research has not pursued the use of network traffic monitoring. This is because using network traffic monitoring to identify and prevent lateral movement entails a few challenges:

1. It is time-consuming to set up a proper baseline. An extensive cluster is needed to set up an interesting baseline, which must be monitored for some time.

2. As the created cluster would be run inside the context of a specific company, the baseline contains a certain bias and could make it difficult to generalize to other companies or situations.

3. It may be difficult to decide when the monitored traffic differs too much from the baseline and what margin of difference is allowed.

4. Network traffic must be interpreted to understand whether it is dangerous. This requires extensive knowledge of network traffic to understand whether the abnormal traffic is dangerous.

4.1.2 Syscall monitoring

This method monitors the syscalls used in the cluster. Every action done in the Kubernetes cluster uses syscalls, which cannot be circumvented. Extensive explanations of what every syscall does and how they work can be found on [49].

This method could be achieved by creating a baseline and performing anomaly detection, which results in similar challenges as mentioned in Subsection 4.1.1. Our research does not investigate using machine learning and establishing a baseline to perform syscall monitoring. Instead, we choose to monitor the syscalls based on `seccomp` profiles. If any `pod` tries to perform a syscall, the profile is checked on whether this syscall is allowed. If it is allowed, there is no problem. If it is not allowed, the action is blocked or logged for later inspection. Our research focuses on a manually created `seccomp` profile. We do not focus extensively on what syscalls should or should not be blocked, as the focus is more on the effectiveness of `seccomp`.

4.2 Experimental setup

To test the effectiveness of `seccomp`, we looked at small-scale Kubernetes-like environments like Minikube or Kind. These small-scale alternatives are good for testing the solution without making it too complex. Using full Kubernetes is unnecessarily complex for these tests. Kind is installed on an Ubuntu VM. The official Kubernetes tutorial of `seccomp` uses Kind as the environment [29]. As it is clear that `seccomp` works in Kind, we used Kind for the experiment.[1]

The experimental setup consists of several levels. The MacBook provided during the project is the foundation. On this MacBook, we run VMWare Fusion, which allows us to spin up VMs [50]. Inside VMWare Fusion, we run an Ubuntu VM using an Ubuntu 22.04.4 Desktop iso [51]. When setting up the Ubuntu VM, the default settings were mostly used. We only changed the memory size to ensure enough memory to test the clusters. We run Docker and Kind on the Ubuntu VM to create the clusters. Finally, `kubectl` is required to interact with the created clusters. The exact steps for installing Docker, Kind, and `kubectl` can be found in Appendix A.

[1] Another option is to use Minikube, but it is unclear if `seccomp` works properly in Minikube. As it was clear that `seccomp` works in Kind, no time was spent investigating this alternative.

4.2.1 Clusters

The clusters are created with a configuration file and name: `kind create cluster --config=config.yml --name=cluster-name`. The `name` flag is used to specify the name of the cluster to differentiate between the different clusters. The `config` flag is used to specify what configuration file to use when setting up the cluster. For this experiment, two clusters are created:[2]

1. `no-sec` cluster: This cluster uses RBAC (introduced in Section 3.1.2) to specify who has what permissions. These permissions are not always configured correctly, leading to exploitable flaws in the cluster. It does **not** make use of `seccomp` but rather depends solely on RBAC.

2. `seccomp` cluster: This cluster makes use of `seccomp` (introduced in Section 3.1.2) to secure the cluster. The cluster uses RBAC to provide the exact same permissions as the `no-sec` cluster. `Seccomp` is used to prevent flaws from being exploited.

As shown in Figure 4.1, the clusters consist of a single control plane `node` and two worker `nodes`. The only difference between the clusters is that the `seccomp` cluster uses `seccomp` profiles. This does not influence the layout of the cluster itself, resulting in both clusters having the same layout.

The manifest file of the no-sec cluster only specifies the number of `nodes` without any additional information, as seen in Listing 4.1. This file (named `no-sec.yml`) is used to create the `no-sec` cluster: `kind create cluster --config=no-sec.yml --name=no-sec`.

The manifest file of the `seccomp` cluster specifies the number of `nodes` as well. Additionally, the file shows an extra mount to the `seccomp` profiles for every `node`, as seen in Listing 4.2. Using this mount, we only need one place to change the profiles instead of making changes on every `node`. The manifest file (named `seccomp.yml`) is used to create the `seccomp` cluster: `kind create cluster --config=seccomp.yml --name=seccomp`.

Listing 4.1: Manifest file in YAML for the `no-sec` cluster.

```
apiVersion: kind.x-k8s.io/v1alpha4
kind: Cluster
nodes:
- role: control-plane
- role: worker
- role: worker
```

[2]When multiple clusters are created, `kubectl` works on the cluster that was created last. To change between different clusters, `kubectl`'s context needs to be changed: `kubectl config use-context kind-CLUSTER-NAME`.

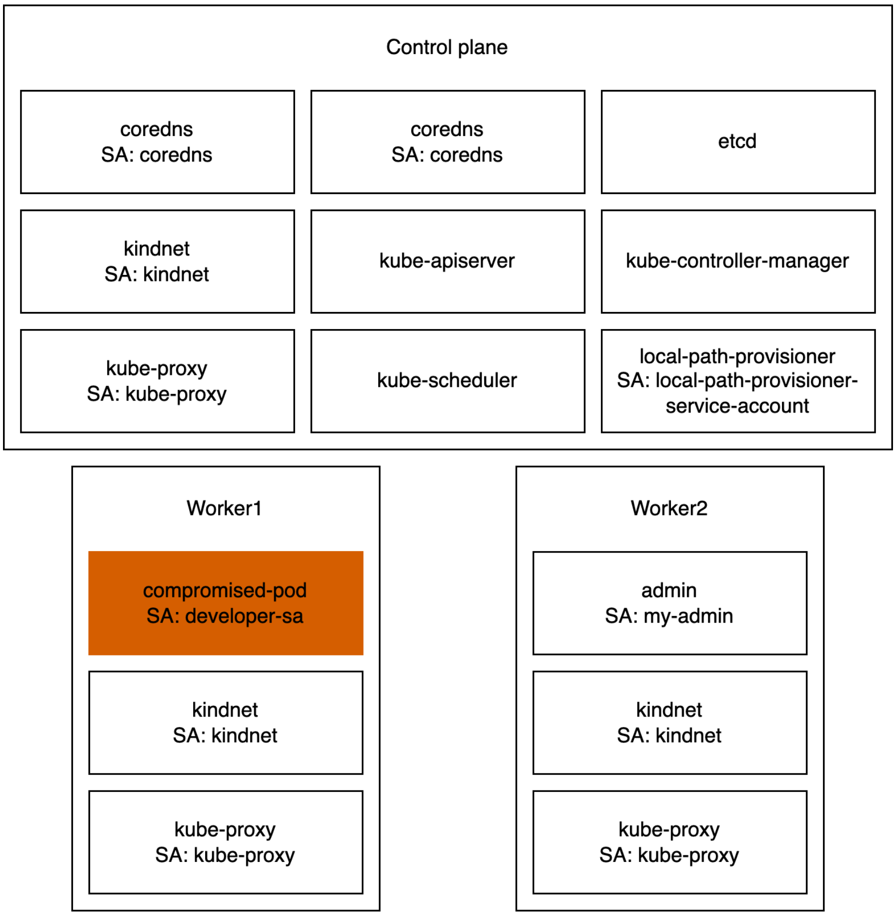

Figure 4.1: Layout of the `no-sec` and `seccomp` clusters with one head `node` and two worker `nodes`. The `pods` can be seen inside the `nodes`.

Listing 4.2: Manifest file in YAML for the `seccomp` cluster.

```
apiVersion: kind.x-k8s.io/v1alpha4
kind: Cluster
nodes:
- role: control-plane
  extraMounts:
  - hostPath: "./profiles"
    containerPath: "/var/lib/kubelet/seccomp/profiles"
- role: worker
  extraMounts:
  - hostPath: "./profiles"
    containerPath: "/var/lib/kubelet/seccomp/profiles"
- role: worker
  extraMounts:
  - hostPath: "./profiles"
    containerPath: "/var/lib/kubelet/seccomp/profiles"
```

To finish creating the clusters, additional Kubernetes components are necessary. We use `serviceAccounts`, (Cluster)Roles, and (Cluster)RoleBindings to enable Role-Based Access Control, for which the same manifest files are used for both clusters. We also need `pod` manifest files to create the `admin` and `compromised pods`. As the `pods` in the `seccomp` cluster use `seccomp` profiles, there are some differences regarding the `pod` manifest files between the two clusters.

ServiceAccounts, (Cluster)Roles, and (Cluster)RoleBindings

After creating the clusters, all the `pods` shown in Figure 4.1 are created except the `compromised-pod` and `admin pods`. To create these `pods`, it is first necessary to create the required `serviceAccounts`, (Cluster)Roles, and (Cluster)RoleBindings. This is done using `kubectl create -f FILE_NAME` with the manifest files in listings 4.3 and 4.4.

Listing 4.3: Manifest file in YAML for the `my-admin` `serviceAccount`.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  annotations:
    kubernetes.io/enforce-mountable-secrets: "true"
  name: my-admin

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: my-admin-rights
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["*"]

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-admin-rights
subjects:
- kind: ServiceAccount
  name: my-admin
  apiGroup: ""
  namespace: default
roleRef:
  kind: ClusterRole
  name: my-admin-rights
  apiGroup: rbac.authorization.k8s.io
```

Listing 4.3 shows the manifest file for a `serviceAccount` named `my-admin`. This `serviceAccount` is bound to the ClusterRole `my-admin-rights`. This ClusterRole provides cluster-wide permission to perform any action on any resource in the cluster. This means that `pods` with the `my-admin serviceAccount` can do anything with any resource in the cluster.

Listing 4.4 shows the manifest file for a namespace called `developers`. It also shows a `serviceAccount` in this namespace called `developer-sa`. This `serviceAccount` is bound to the ClusterRole `developer-role-ns` and the Role `developer-role-pod`. The ClusterRole provides cluster-wide permission to `get` and `list` namespaces and `pods` in the cluster. The Role provides permission in the `developers` namespace to perform any action on `pods`, `pods/exec`, and `pods/log`. What these permissions entail was explained in Section 3.3.

Listing 4.4: Manifest file in YAML for the `developer-sa` `serviceAccount`.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: developers

---

apiVersion: v1
kind: ServiceAccount
metadata:
  namespace: developers
  name: developer-sa

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: developer-role-ns
rules:
- apiGroups:
  - ""
  resources:
  - namespaces
  - pods
  verbs:
  - get
  - list

---

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer-role-pod
  namespace: developers
rules:
```

```
- apiGroups:
  - ""
  resources: ["pods", "pods/exec", "pods/log"]
  verbs: ["*"]

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  namespace: developers
  name: developer-role-binding1
roleRef:
  kind: ClusterRole
  name: developer-role-ns
  apiGroup: rbac.authorization.k8s.io
subjects:
 - kind: ServiceAccount
   name: developer-sa
   namespace: developers

---

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: developers
  name: developer-role-binding2
roleRef:
  kind: Role
  name: developer-role-pod
  apiGroup: rbac.authorization.k8s.io
subjects:
 - kind: ServiceAccount
   name: developer-sa
   namespace: developers
```

Pod creation in the no-sec cluster

After creating the required `serviceAccounts`, we created `pods` that define their permissions using one of these `serviceAccounts`. The created `pods` use a simple `nginx` container, often used for web applications, to keep the experiment environment simple for demonstration purposes.

Listing 4.5 shows the manifest file for the `admin pod` on the second worker `node`. Line 6 shows that this `pod` has the `my-admin serviceAccount`, which means it has the permissions connected to this `serviceAccount`. Additionally, it has a volume mount to the `node`. This means that the `/mnt/important-data` directory on the `pod` is the same as the `/important` directory on the `node`. Section 4.3.2 shows how mounted files and directories could be affected during an attack. Finally, this `pod` is placed on the `no-sec-worker2 node`, as specified in line 18.

Listings 4.6 and 4.7 show the manifest files for the unprivileged and privileged `compromised-pod pods`, respectively. The only difference between the two `pods` is whether they are privileged. In both manifest files, line 7 shows that the `pods` have the `developer-sa serviceAccount`, which means they have the permissions connected to this `serviceAccount`. In Listing 4.6, line 11 shows that the `pod` is placed on the `no-sec-worker node`. Similarly, in Listing 4.7, line 13 shows that the `pod` is placed on the `no-sec-worker node` as well. Thus, the `compromised-pod pod` is on another `node` than the `admin pod` in all tested attack scenarios. This is done to show whether the attacks are affected by what `node` the `pods` are on.

Listing 4.5: Manifest file in YAML for the ***`admin pod`****.*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: admin
spec:
  serviceAccountName: my-admin
  volumes:
  - name: important-data
    hostPath:
      path: /important
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: important-data
      mountPath: /mnt/important-data
  hostNetwork: true
  nodeName: no-sec-worker2
```

Listing 4.6: Manifest file in YAML for the unprivileged ***`compromised-pod pod`****.*

```yaml
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: compromised-pod
5   namespace: developers
6 spec:
7   serviceAccountName: developer-sa
8   containers:
9   - name: nginx
10    image: nginx
11  nodeName: no-sec-worker
```

Listing 4.7: Manifest file in YAML for the privileged `compromised-pod-priv` pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: compromised-pod
  namespace: developers
spec:
  serviceAccountName: developer-sa
  containers:
  - name: nginx
    image: nginx
    securityContext:
      privileged: true
  nodeName: no-sec-worker
```

Pod creation in the seccomp cluster

The process for the `seccomp` cluster is similar to that of the `no-sec` cluster. The `pod` manifests require additional lines to use the `seccomp` profiles. The actual profiles need to be created to ensure they are recognized when creating the `pod`. When creating the `seccomp` profiles, it is possible to log, allow, or block syscalls.

Listing 4.8 shows the `seccomp` profile in the `audit.json` file. By setting the default action to `LOG`, this profile creates logs for any syscall that is not explicitly specified. The profile allows the specified syscalls (`read` and `write`) to be made without logging them. This is done because every key press when typing results in one log for both `read` and `write`. As this results in many logs that do not provide any relevant information, we decided not to log them. No syscalls are blocked through this profile.

Many logs were created even without `read` and `write` being logged. Linux suppresses a part of the logging by default to prevent overwhelming amounts of logs. To see all the syscalls used during this experiment, we turned off the suppression: `sudo sysctl -w /kernel/printk_ratelimit=0` [52]. The logs for syscalls are located in a special file on the VM: `/var/log/syslog`. Using the `tail -f /var/log/syslog` command in the VM makes it possible to see these logs in real time [29].

Listing 4.9 shows the `seccomp` profile in the `block.json` file. This profile allows any syscall not explicitly specified by setting the default action to `ALLOW`. The profile blocks the specified syscalls, resulting in an error when one of them is executed.

Listing 4.8: The `seccomp` profile in the `audit.json` file.

```
{
    "defaultAction": "SCMP_ACT_LOG",
    "syscalls": [
        {
            "names": [
                "read",
                "write"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

Listing 4.9: The `seccomp` profile in the `block.json` file.

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "syscalls": [
        {
            "names": [
                "execve",
                "openat",
                "chroot"
            ],
            "action": "SCMP_ACT_ERRNO"
        }
    ]
}
```

Listing 4.10 shows the manifest file for the `admin-seccomp pod`. It is similar to the manifest in Listing 4.5, with the addition of lines 17-20. These lines make sure that this `pod` uses the `seccomp` profile specified in the `audit.json` file.

Listing 4.11 shows the manifest file for the `compromised-pod-seccomp pod`. It is similar to the manifest in Listing 4.6, with the addition of lines 11-14. These lines make sure that this `pod` uses the `seccomp` profile specified in the `audit.json` file.

Listing 4.12 shows the manifest file for the `compromised-pod-priv-seccomp pod`. It is similar to the manifest in Listing 4.7, with the addition of lines 13-16. These lines make sure that this `pod` uses the `seccomp` profile specified in the `audit.json` file.

As can be seen, the manifest files for the `pods` are exactly the same as the manifest files for the `no-sec` cluster, with the same `serviceAccount` defined. The only difference is the added lines specifying the `seccomp` profiles. These three manifest files use the profile in the `audit.json` file, shown in Listing 4.8, to perform logging. We also tested the attacks with the profile in the `block.json` file, shown in Listing 4.9, to perform blocking.

Listing 4.10: Manifest file in YAML for the `admin-seccomp` *pod.*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: admin-seccomp
spec:
  serviceAccountName: my-admin
  volumes:
  - name: important-data
    hostPath:
      path: /important
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: important-data
      mountPath: /mnt/important-data
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  hostNetwork: true
  nodeName: no-sec-worker2
```

Listing 4.11: Manifest file in YAML for the unprivileged `compromised-pod-seccomp` *pod.*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: compromised-pod-seccomp
  namespace: developers
spec:
  serviceAccountName: developer-sa
  containers:
  - name: nginx
    image: nginx
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  nodeName: no-sec-worker
```

Listing 4.12: Manifest file in YAML for the privileged `compromised-pod-priv-seccomp` pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: compromised-pod-priv-seccomp
  namespace: developers
spec:
  serviceAccountName: developer-sa
  containers:
  - name: nginx
    image: nginx
    securityContext:
      privileged: true
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  nodeName: no-sec-worker
```

4.3 Attacks on the clusters

This section explains what attacks are performed to test the clusters introduced in Section 4.2. As mentioned in Section 3.3, the attacker has compromised a `pod` in the cluster. The compromised `pod` has the `developer-sa` `serviceAccount` connected. This means that the compromised `pod` has the permissions to perform any action on `pods`, `pods/exec`, and `pods/log` in the developer namespace. Additionally, the compromised `pod` also has the permissions to get and list `pods` and `namespaces` cluster-wide. As the attacker has compromised this `pod`, the attacker also has these permissions.

Both attacks include a step that requires creating a `pod` on the cluster. To do this, the attacker must install `kubectl` on the compromised `pod`. The commands for this are mentioned in Appendix A.1.2, but they are repeated here (without `sudo`, as the attacker has root access on the compromised `pod`):

- `curl -LO "https://dl.k8s.io/release/$(curl -L -s \`
  `https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"`
- `install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl`

After installing `kubectl`, it is possible to use `kubectl` to interact with the cluster. A manifest file is needed to create a `pod`. This file can be created by using the `cat` command, which is already installed. The following construction can be used to create a `pod` manifest file in YAML named `file.yml`:

```
cat <<EOF >> file.yml
manifest code
EOF
```

4.3.1 Attack 1: Find and retrieve secret information in a pod

This attack aims to find resources like `secrets` or `configMaps` for which the attacker does not have the permissions to access them. With the permissions of the compromised `pod`, it is impossible to see the content if these resources directly. This attack makes use of the fact that volumes mounted on other `pods` can also be mounted on `pods` created by the attacker. As the attacker has the permissions to create `pods` and access their logs, he can use this to retrieve the content of those resources.

The attack uses the compromised `pod`'s permissions to get and list all `pods` in the entire cluster. These permissions are used to inspect the `pods` and see if some `pod` has an interesting volume mount. When inspecting the `admin pod`, it can be seen that a `configMap` named `kube-root-ca.crt` is mounted. Based on the permissions of the `developer-sa serviceAccount`, the attacker *should* not have permission to see the content of this `configMap`. To retrieve the content of the `configMap` anyway, the attacker creates a `pod` that mounts the `kube-root-ca.crt configMap` using the `pod` manifest in Listing 4.13. The created `pod` executes the command in line 14, printing the content of the specified file (the `configMap`) to the standard output [53]. The content of the standard output can be found in the logs of the `pods`, to which the attacker *does* have access. The logs of the `attack1 pod` can be inspected using the following command: `kubectl logs attack1`.

To summarize, this attack makes use of the compromised `pod`'s permission to get and list all the `pods` in the cluster, as well as creating a new `pod` in the `developers` namespace and accessing its logs. The attack does not require the permission to enter the newly created `pod` (which is provided through permissions on `pods/exec`).

Listing 4.13: Manifest file in YAML for the attack1 `pod`.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: attack1
5    namespace: developers
6  spec:
7    volumes:
8    - name: config-volume
9      configMap:
10       name: kube-root-ca.crt
11   containers:
12   - image: busybox
13     name: print-configmap
14     command: ["/bin/sh", "-c", "cat /etc/config/*"]
15     volumeMounts:
16     - name: config-volume
17       mountPath: /etc/config
18   restartPolicy: Never
```

4.3.2 Attack 2: Gain node-level access on another node

This attack aims to gain root-level filesystem access on the `node`. It is based on an existing attack [38]. A `pod` is created using the `pod` manifest in Listing 4.14. To specify on what `node` this `pod` should be created, the `nodeName` field in the `pod` specification is used. Line 22 shows that this `pod` is created on the `no-sec-worker2 node`. Line 23 shows that this `pod` is created on the `seccomp-worker2 node`. Depending on what cluster is being attacked, the corresponding line is used. These are the `nodes` on which the `admin` and `admin-seccomp` pods are, respectively.

The manifest is used to create a `pod`, which can be used to break out to the `node` level. The volume mount in the manifest establishes a connection between the `/host` folder on the `pod` and the root directory of the `node`. Aside from the volume mount, the only requirement is that the `bash` command can be performed on the `pod`. The `bash` command enables the attacker to execute commands in the `pod` [54]. For simplicity, the attack `pod` uses a simple Ubuntu container that sleeps forever without performing other actions.

To access the created `pod`, the attacker uses `kubectl exec -it attack2 -- bash`. This executes the `bash` command in the `attack2 pod` [48]. When inside the `pod`, the root directory can be changed: `chroot /host bash`. This command changes the root directory for the current process to the `/host` folder and enables the attacker to execute commands in this folder. Because of the volume mount, this is equal to changing the root directory of the `pod` to the root directory of the `node`. Therefore, the attacker has now achieved root-level filesystem access on the `node`.

With root-level filesystem access on the `node`, the attacker can interact with the Kubernetes cluster with the `node`'s permissions. This does require installing `kubectl` again. This can be done with the commands mentioned in Section 4.3. To interact with the cluster, it is necessary to state the configuration file:
`kubectl --kubeconfig=/etc/kubernetes/kubelet.conf COMMAND`. If the configuration file is not specifically stated when using `kubectl`, the interaction with the cluster fails.

Aside from interacting with the cluster, accessing any file or folder on the `node` is also possible. This includes files and folders mounted on all the `pods` on the `node`. If a `pod` uses the content of such a mounted file, it is possible to affect that `pod` by changing or removing the file. It also allows for inspecting the content of these files, which could be prohibited and blocked if tried by other means.

This attack makes use of the compromised `pod`'s permission to create a new `pod` and enter it with `kubectl exec`. The attack does not require the permission to access the logs of the `pod` (which is provided through permissions on `pods/log`). It could make use of the permission to get and list all the `pods` in the cluster, as this could provide information about what `node` contains interesting information.

Listing 4.14: Manifest file in YAML for the attack2 `pod`.

```
apiVersion: v1
kind: Pod
metadata:
  name: attack2
  namespace: developers
spec:
  volumes:
  - name: host-fs
    hostPath:
      path: /
  containers:
  - image: ubuntu
    name: attacker-pod
    command: ["/bin/sh", "-c", "sleep infinity"]
    securityContext:
      privileged: true
      allowPrivilegeEscalation: true
    volumeMounts:
    - name: host-fs
      mountPath: /host
  restartPolicy: Never
  nodeName: no-sec-worker2
  nodeName: seccomp-worker2
```

Chapter 5

Results

As seen in Table 5.1, the attacks are successful in every case. `Seccomp` did not prevent the attack in any scenario. One reason for these results is that `seccomp` does not work in privileged `pods`. This means that `seccomp` cannot be used when the attacks are performed from a privileged compromised `pod`. When the attacks are performed from an unprivileged compromised `pod`, `seccomp` can be used. However, the problem is that blocking certain syscalls, with the `seccomp` profile in the `block.json` file shown in Listing 4.9, causes the compromised `pod` to fail during the creation phase. Section 5.2 explains this in more detail.

Cluster	Compromised `pod`	Attack 1	Attack 2
no-sec	unprivileged	✓	✓
	privileged	✓	✓
seccomp	unprivileged	✓	✓
	privileged	✓	✓

Table 5.1: Attack outcomes across all test scenarios of this experiment, where a ✓ implies a successful attack and a × would have implied a failed attack.

In contrast, it is possible to use the `seccomp` profile in the `audit.json` file shown in Listing 4.8. This does not block any syscall but provides logs of what syscalls are used at what time, excluding the `read` and `write` syscalls. As no syscalls are blocked, the compromised `pod` starts up properly, and it is possible to log the syscalls. However, not blocking the syscalls means that the attacks work the same as they did on the `no-sec` cluster. Therefore, the attacks that work on the `no-sec` cluster also work on the `seccomp` cluster. The only difference is that it is possible to see the logs of what happens during the attack.

For both clusters, the attacks are performed from an unprivileged and a privileged compromised `pod` to see if this impacts the attack or defense. The results of the `no-sec` cluster are elaborated on in Section 5.1, whilst the results of the `seccomp` cluster are elaborated on in Section 5.2.

5.1 Results no-sec cluster

5.1.1 Results no-sec cluster: Attack 1

The attack works similarly for the unprivileged and privileged compromised `pods`, leading to the same results. The steps are shown below, with step 7 showing that it is possible to retrieve the content of the `configMap` without having the permission to see its content. This attack shows that the attacker can access information on the cluster for which the attacker does not have permission.

1. Install `kubectl` with the commands mentioned in Section 4.3 to interact with the cluster.
2. Retrieve all `pods`: `kubectl get pods -A`.
3. Inspect the `pods` to find interesting mounted information: `kubectl get pods admin -n default -o yaml`. This shows that the `admin pod` has the `kube-root-ca.crt configMap` mounted.
4. Try to see the content of the `configMap` directly: `kubectl get configmaps`. This results in an error because of a lack of permissions.
5. Create the manifest file `attack1.yml` in Listing 4.13 using the commands mentioned in Section 4.3.
6. Create the `pod` with the `pod` manifest: `kubectl create -f attack1.yml`. The created `pod` executes the "`/bin/sh -c cat /etc/config/*`" command to print the content of the `configMap` to the standard output.
7. Use `kubectl logs attack1` to see the output of the previous step, i.e., the content of the mounted `configMap`:


```
-----BEGIN CERTIFICATE-----
MIIDBTCCAe2gAwIBAgIIV4WkQYPnbjEwDQYJKoZIhvcNAQELBQAwFTETMBEGA1UE
AxMKa3ViZXJuZXRlczAeFw0yNDA2MTAwOTU1MjlaFw0zNDA2MDgxMDAwMjlaMBUx
...
UxZ8cLngM4xSglfh+695M8/R5YwBd4myx334LS7Cy7c4im+tYvVAOvY5hEnMvVFa
wgHvFdhsCdO2
-----END CERTIFICATE——
```


Figure 5.1: The content of the `configMap` as output of the `kubectl logs` command.

By inspecting the existing `pods` in step 3, it is possible to determine what resources there are in the cluster. The compromised `pod` does not have the required permissions to see all resources. As mentioned in step 4, trying to see the content of the `configMap` resulted in an error. Using the `attack1.yml` file to create the `attack1 pod` makes it possible to work around this lack of permissions. This attack shows that it is possible to see the content of the `configMap`, whilst it should not be possible according to the permissions.

5.1.2 Results no-sec cluster: Attack 2

The attack works similarly for the unprivileged and privileged compromised `pods`, leading to the same results. The steps are shown below, with step 7 showing that it is possible to gain `node`-level access on the second worker `node`. This indicates that lateral movement on the `node` level is possible. The attacker can do various things from the `node`, as shown in steps 8 and 9.

1. Install `kubectl` with the commands mentioned in Section 4.3 to interact with the cluster.

2. Retrieve all `pods`: `kubectl get pods -A`.

3. Inspect the `pods` to find interesting mounted information: `kubectl get pods admin -n default -o yaml`. This shows that the `admin pod` has the `/important` folder mounted.

4. Create the manifest file `attack2.yml` in Listing 4.14 using the commands mentioned in Section 4.3. Line 23 needs to be removed to ensure the `nodeName` corresponds to the `no-sec` cluster.

5. Create the `pod` with the `pod` manifest: `kubectl create -f attack2.yml`. The created `pod` only sleeps and performs no other action. There is a connection between the `/host` folder on the `pod` and the root directory on the `no-sec-worker2 node`. This means that the `/host` folder on the `pod` is the same as the root directory of the `node`. All files and folders are the same, so changing a file in the `/host` folder does the same with that file in the root directory of the `node`.

6. Use `kubectl exec -it attack2 -n developers -- bash` to enter the `pod`. The attacker can now execute commands in the `attack2 pod`.

7. Inside the `attack2 pod`, execute the `chroot /host bash` command. This changes the current process's root directory to the `/host` folder. Because of the connection between this folder on the `pod` and the root directory of the `node`, the current process's root directory is changed to the root directory of the `node`. This causes the attacker to gain `node`-level access to the second worker `node`.

8. The attacker can interact with the cluster using the permissions of the `no-sec-worker2 node`: `kubectl --kubeconfig=/etc/kubernetes/kubelet.conf COMMAND`. By specifying the configuration file, the attacker can use `kubectl` as regular to perform any command, as long as the permissions allow it.

9. Instead of interacting with the cluster, it is possible to access the files and folders. From step 3, it can be inferred that the `/important` folder on the `node` is used in the `admin pod`. Various things can be done with the files in this folder. Below are some examples how the attacker can interact with the files:

(a) The attacker can use `ls` to see the files in the current (`/important`) folder, showing two files. With `cat`, the content of these files can be seen, as shown in Figure 5.2.

```
root@attack2:/important# ls
file.txt  file2.txt
root@attack2:/important# cat file.txt
This is the normal content
of this file. There is
nothing wrong with it, it
is all normal text.
root@attack2:/important# cat file2.txt
This is another file
with normal text
as well.
```

Figure 5.2: Using the `cat` command to see the content of the files.

(b) The attacker can use `vim` to change the content of a file. Here, `cat` is used to show the changed content before and after changing the file, as shown in Figure 5.3.

```
root@attack2:/important# cat file2.txt
This is another file
with normal text
as well.
root@attack2:/important# vim file2.txt
root@attack2:/important# cat file2.txt
This is another file
I can remove a part
of the original and
add malicious things.
```

Figure 5.3: Using the `vim` command to change the content of the files.

(c) The attacker can use `vim` to create an entirely new file. Afterwards, `ls` shows that a new file is created. Here, `cat` is used to show the content of this new file, as shown in Figure 5.4.

```
root@attack2:/important# vim file3.txt
root@attack2:/important# ls
file.txt  file2.txt  file3.txt
root@attack2:/important# cat file3.txt
This is an entirely
new file created
by the attacker
```

Figure 5.4: Using the `vim` command to create new files.

The attacker can inspect the existing `pods` to determine what files and folders are used and on what `node` they can be found. Using the `attack2.yml` file to create the `attack2 pod` enables the attacker to gain root-level filesystem access to the no-sec-worker2 node. With root-level filesystem access, the attacker can create new files. Additionally, the attacker can inspect, change, or delete existing files on the node. This concerns both files mounted on pods, as well as files necessary for the proper functioning of the node. For instance, it is possible to make the node crash by removing every file: rm -rf / --no-preserve-root [55]. We experimentally verified that these actions can be performed.

5.2 Results seccomp cluster

At the start of this chapter, we already mentioned that seccomp did not prevent any of the attacks. For the privileged compromised pod, this is because seccomp does not work in privileged pods. For the unprivileged pod, this is because the relevant syscalls cannot be blocked. When we tried creating a pod that blocks syscalls such as execve, openat, or chroot, we got an error. Therefore, syscalls needed when creating the pod cannot be blocked. Besides, when the pod is used by a developer to perform specific actions, the pod requires certain syscalls. Any syscalls required by the pod during normal usage cannot be blocked.

The main steps of attack 1 can be summarized as creating a pod and checking the logs of a pod. The main steps in attack 2 can be summarized as creating a pod and entering a pod. These steps are also performed during normal usage of the pod when a developer has to do their work. Because of this, it is rather difficult to block syscalls required by these actions. However, logging is still possible, as shown in sections 5.2.1 and 5.2.2.

5.2.1 Results seccomp cluster: Attack 1

This section explains both the attack and its results, as well as the logs collected through seccomp.

Attack

The attack works precisely as the attack on the no-sec cluster shown in Section 5.1.1. The only difference is in step 3: instead of the admin pod, we use the admin-seccomp pod.

Logging

It is possible to monitor the syslogs to see what is going on in the unprivileged compromised pod. However, seccomp does not work with privileged pods. For the unprivileged compromised pod, the syslogs can be accessed on the VM in the /var/log/syslog file. Using the tail -f /var/log/syslog command makes it possible to see these logs in real time.

Checking these logs results in a long list of logs. A small sample of logs for creating a pod is shown in Figure 5.5. The main focus of every log entry is the used syscall at the beginning of every third line. As can be seen in the figure, syscalls 13 (`rt_sigaction`), 59 (`execve`), 158 (`arch_prctl`), 204 (`sched_getaffinity`), and 257 (`openat`) are used when creating a `pod`. More syscalls are used, but not shown here, adding up to 46 unique syscalls.

```
Jun 20 14:42:32 mike-virtual-machine kernel: [12093.532500] audit: type=1326 audit(1718887352.032:182205): auid=4294967295
uid=0 gid=0 ses=4294967295 subj=unconfined pid=32338 comm="bash" exe="/usr/bin/bash" sig=0 arch=c000003e
syscall=13 compat=0 ip=0x748c20dbd11f code=0x7ffc0000

Jun 20 14:42:32 mike-virtual-machine kernel: [12093.532539] audit: type=1326 audit(1718887352.032:182206): auid=4294967295
uid=0 gid=0 ses=4294967295 subj=unconfined pid=32338 comm="bash" exe="/usr/bin/bash" sig=0 arch=c000003e
syscall=59 compat=0 ip=0x748c20e55a17 code=0x7ffc0000

Jun 20 14:42:32 mike-virtual-machine kernel: [12093.533096] audit: type=1326 audit(1718887352.032:182207): auid=4294967295
uid=0 gid=0 ses=4294967295 subj=unconfined pid=32338 comm="kubectl" exe="/usr/local/bin/kubectl" sig=0 arch=c000003e
syscall=158 compat=0 ip=0x47693f code=0x7ffc0000

Jun 20 14:42:32 mike-virtual-machine kernel: [12093.533170] audit: type=1326 audit(1718887352.032:182208): auid=4294967295
uid=0 gid=0 ses=4294967295 subj=unconfined pid=32338 comm="kubectl" exe="/usr/local/bin/kubectl" sig=0 arch=c000003e
syscall=204 compat=0 ip=0x476996 code=0x7ffc0000

Jun 20 14:42:32 mike-virtual-machine kernel: [12093.533173] audit: type=1326 audit(1718887352.032:182209): auid=4294967295
uid=0 gid=0 ses=4294967295 subj=unconfined pid=32338 comm="kubectl" exe="/usr/local/bin/kubectl" sig=0 arch=c000003e
syscall=257 compat=0 ip=0x47615a code=0x7ffc0000
```

Figure 5.5: Sample of the syslogs when creating a `pod`*.*

When the attacker reads the logs of a `pod`, the syslogs can also be monitored. When reading the logs of a `pod`, the syscalls match those we saw in the syslogs of creating a `pod`, excluding two. In every step of the attack, there are various syscalls. This includes syscalls `execve` and `openat`. As mentioned in Section 3.2, these syscalls are also used in other attacks.

These logs show that, even though the attack cannot be blocked, logging can be used to see what syscalls are used in every step of the attack. One problem, however, is that the steps of this attack consist of regular usage of the `pod`. The same steps can be performed when a developer has to do their work. Because of this, it is difficult to distinguish between a developer performing an action and an attacker performing it.

5.2.2 Results seccomp cluster: Attack 2

This section explains both the attack and its results, as well as the logs collected through `seccomp`.

Attack

The attack works precisely as the attack on the `no-sec` cluster shown in Section 5.1.2. The only difference is in the `pod` and `node` names. Instead of the `admin pod`, we use the `admin-seccomp pod`. Instead of the `no-sec-worker2` `node`, we use the `seccomp-worker2` `node`.

Logging

It is possible to monitor the syslogs to see what is going on in the unprivileged compromised `pod`. However, `seccomp` does not work with privileged `pods`. For the unprivileged compromised `pod`, the syslogs can be accessed on the VM in the `/var/log/syslog` file. Using the `tail -f /var/log/syslog` command makes it possible to see these logs in real time.

Checking these logs results in a long list of logs. Steps 1-5 used in attack 2 were similar to steps in attack 1, resulting in similar logs. Steps 6-9 differ from attack 1 and involve entering a self-created `pod`. When the attacker enters the `attack2 pod` and interacts there, the syslogs can also be monitored:

- Step 6: Entering the `pod` using `kubectl exec` results in the usage of 43 syscalls (excluding `read` and `write`).
- Step 7: Changing the root only shows four syscalls: 24 (`sched_yield`), 35 (`nanosleep`), 202 (`futex`), and 281 (`epoll_pwait`).
- Step 8: Interacting with the cluster (`get pods -A`) only shows three syscalls: 35 (`nanosleep`), 202 (`futex`), and 281 (`epoll_pwait`).
- Step 9: Interacting with a folder and file mounted on a `pod` results in seven syscalls: 15 (`rt_sigreturn`), 24 (`sched_yield`), 35 (`nanosleep`), 39 (`getpid`), 202 (`futex`), 234 (`tgkill`), and 281 (`epoll_pwait`).

In step 7, the root is changed in an unmonitored `node`. We experimentally verified that changing the root makes use of the `chroot` (161) syscall. In the logs of step 7, we could only see four syscalls, not including the `chroot` syscall. This shows that the monitoring does not show the exact actions of the attacker when the attacker enters an unmonitored `pod` from the compromised `pod`. Additionally, steps 8 and 9 also show a small number of syscalls. This indicates that the attacker is outside the confines of what logging can see properly after entering a self-created `pod`.

These logs show that, even though the attack cannot be blocked, logging can be used to see what syscalls are used in most steps of the attack. We can see, however, that in steps 6-9, logging does not achieve the desired results. Another problem is that the steps of this attack consist of regular usage of the `pod`. The same steps can be performed when a developer has to do their work. Because of this, it is difficult to distinguish between a developer performing an action and an attacker performing it.

Chapter 6

Discussion

This research focuses on using seccomp to identify and prevent lateral movement in the Kubernetes cluster. Section 6.1 explains the key findings. Afterwards, Section 6.2 describes the limitations encountered during this research. Finally, Section 6.3 elaborates on what could be done in the future to continue and improve research in this field.

6.1 Key findings

The results indicate that seccomp cannot be used in every scenario to identify and prevent lateral movement, which is shown in Table 6.1. Two main categories can be distinguished: unprivileged compromised pod and privileged compromised pod. As can be seen, seccomp does not work in privileged pods. This is because privileged pods run in unconfined mode, disabling seccomp [29]. Seccomp works in most scenarios when used in unprivileged pods. However, logging does not always provide useful information in these cases, as it is hard to distinguish between regular usage and an attack. This information could be combined with other information to provide better insights, which is elaborated on in the section below.

6.1.1 Unprivileged compromised pod

In the unprivileged compromised pod, it is possible to use seccomp. This resulted in various problems regarding the blocking and logging functionalities provided by seccomp.

Use seccomp for blocking

Blocking the syscalls using seccomp proved to be difficult. Certain syscalls are necessary to start up a pod. These syscalls cannot be blocked; otherwise, the pod does not start up and results in an error during the creation phase. If these syscalls are used during attacks, using seccomp to block the attack is impossible.

Aside from the syscalls used during the creation phase, there are also syscalls used during the regular usage of the pod. A pod has specific tasks and exists for a reason.

Action		Unprivileged compromised `pod`	Privileged compromised `pod`
Blocking	Syscalls used in the creation phase	×	×
Blocking	Syscalls used in regular usage of the compromised `pod`	×	×
Blocking	Other syscalls	✓	×
Logging	Syscalls used in the creation phase	✓	×
Logging	Syscalls used in regular usage of the compromised `pod`	✓	×
Logging	Other syscalls	✓	×

Table 6.1: Outcomes of blocking and logging specific syscalls with `seccomp`, where a ✓ implies the action can be performed successfully and a × implies the action cannot be performed successfully.

The syscalls that the `pod` needs to function correctly cannot be blocked. If these syscalls are used during the attacks, using `seccomp` to block the attack is impossible.

If attacks use syscalls that are not needed during the creation of the `pod` nor the regular usage of the `pod`, `seccomp` could provide proper protection against those attacks. Logging can be used to find out what syscalls are used during the creation phase and regular usage. We can create a `seccomp` profile with an allowlist approach. In the profile, we allow the necessary syscalls, and we use `BLOCK` as the default action to block all other unnecessary syscalls. Any attack using any of the blocked syscalls gets blocked by this method.

Blocking did not work for the two attacks tested in this research. Primarily, this is because trying to block the `execve`, `openat`, and `chroot` syscalls caused a problem during the creation phase. Even if this was not the case, the attacks consisted of actions that could also be performed during the regular usage of the `pod`. In scenarios where the attack uses different syscalls, it could be possible to block them properly.

Use seccomp for logging

Aside from blocking the syscalls, it is also possible to log them. Logging can be done for all syscalls. Syscalls that are used during the creation phase do not pose any problems. The main problem with logging lies in the syscalls used during regular usage of the `pod`. Any syscall used during regular usage cannot be blocked, but it is possible to log them instead. However, the logs do not always provide meaningful information. If we use certain syscalls during regular usage of the `pod`, we will often see these syscalls in the logs. If an attacker performs the same actions using the same syscalls, it is hard to distinguish between the regular usage of the pod and an attacker performing an attack. This means that using the logs to identify attacks would result in many false positives.

It would be possible to combine logging with other measures, such as the working schedule. Using different accounts for different employees can be used to check whether the situation is normal or suspicious. For instance, when an account belonging to one of the developers is used while they are not working, this seems rather suspicious. On the contrary, when that same account is used when the developer is supposed to be working, it is just a regular case and poses no problem. This method is an example of RBAC-A, which is a combination of Role-Based Access Control and Attribute-Based Access Control [56]. There might be possibilities to use the logging in combination with additional information to utilize the logs created through `seccomp` properly.

Similarly to blocking, logging would be possible with syscalls not used during the creation of the `pod` nor the regular usage of the `pod`. However, in this case, blocking would also be possible. Depending on the desired result, it is possible to use either logging or blocking for such syscalls.

Logging did work for the two attacks tested in this research. The resulting logs showed what syscalls were used in the attacks. As mentioned before, the two attacks consisted of actions that could also be performed during regular usage of the `pod`. Because of this, it is difficult to say how useful the logs were. If a developer performs these actions, the logs would look the same. Therefore, using `seccomp` to identify lateral movement seems complicated. It is possible to use identify lateral movement, but in cases where logging might provide useful information, it is also possible to use blocking instead of logging.

6.1.2 Privileged compromised pod

As mentioned in sections 5.2 and 6.1, `seccomp` does not work in privileged `pods` [29]. We found no clear specification of why the developers designed it like this. Still, we assume that this is because privileged `pods` have the capabilities to change or remove the restrictions the `seccomp` profile imposes anyway. This makes `seccomp` redundant and possibly misleading on privileged `pods`, as we cannot be sure that the profile is always active on the `pod`. As Kubernetes' documentation shows, privileged `pods` disable not only `seccomp`, they also disable other security measures like `AppArmor` and `SELinux` [57]. This shows that the Kubernetes architecture, as it currently exists, cannot use these security measures at all in privileged `pods`.

Additionally, privileged `pods` provide many permissions. In most cases, not all of these permissions are necessary, and the needed permissions can be provided through other means than making the `pods` privileged. In such cases, it would be best practice to use the other method and minimize the number of privileged `pods`. However, some `pods` require the permissions provided through being privileged. Creating such privileged `pods` needs to be considered carefully. Even though it would be better if `seccomp` could work properly in privileged `pods`, we want to minimize the number of privileged `pods` anyway. Because of this, `seccomp` not working in privileged `pods` is not too big of a problem.

6.2 Limitations

While performing this research, we encountered a few limitations. The most significant limitation of our research is the limited attack scenario coverage. The two attacks demonstrated the opportunities and challenges of `seccomp` in Kubernetes environments. However, this does not encompass all possible scenarios. `Seccomp`'s effectiveness depends on the overlap between syscalls used during lateral movement and syscalls used during the creation phase and regular usage of the `pods`. Depending on the scenario, it might be possible to use `seccomp` in combination with other security mechanisms, such as RBAC-A, to achieve better results.

Another important limitation is that this study mainly focused on `seccomp`. Aside from `seccomp`, other tools could help in the identification and prevention of lateral movement in Kubernetes. `Seccomp` is not sufficient to solve this problem. A combination of techniques might be necessary to achieve the optimal results, which was not investigated during this research.

This research is also limited by the fact that the attacks were performed on self-created clusters in an experimental setting. The findings might not be applicable in real-world settings, where the scale, variability of workloads, and the heterogeneity of the infrastructure could influence the results.

Finally, this research is limited because `pods` created during the attacks do not have a profile. `Seccomp` profiles are not mandatory to implement when creating a `pod`. Results might differ when logging and blocking is possible inside a `pod` created by the attacker. Implementing mandatory `seccomp` profiles could improve the security and effectiveness of logging and blocking.

6.3 Future work

Based on our research, a few directions can be investigated. This section describes these directions in more detail.

Alternative attacker models

As briefly mentioned in the first point of the limitations, the limited attack scenario coverage impedes our judgement regarding `seccomp`'s effectiveness. One possibility to look at in the future is to perform similar research but with a different attacker model. Using a different attacker model, `seccomp` might be usable to achieve the desired result.

Focus on specific attack syscalls

During our research, it became clear that many syscalls are connected to just a single process. It could be interesting to investigate which syscalls or chains of syscalls are used in attacks regularly. Focusing on chains of syscalls could result in different possibilities regarding blocking and logging.

Alternative tools

Instead of using `seccomp`, it is possible to take a different approach. This can be considered on two levels:

1. The field of syscall monitoring has tools aside from `seccomp`. These tools might provide different approaches, resulting in different results. This could improve the results or result in similar challenges and issues connected to syscalls.

2. Another option is to investigate possibilities outside the field of syscall monitoring. One possibility is to look at network traffic monitoring, already introduced in Section 4.1.

This research focused on syscall monitoring and briefly looked at network traffic monitoring. There might be other techniques that can be used to identify or prevent lateral movement in Kubernetes. Exploring and investigating these methods could be interesting for further research.

Attacks through Docker

Running the Kind cluster uses Docker. We Accessed the `pods` and `nodes` using `kubectl`, which is inside the context of the cluster. It would also be possible to enter the `node` using `docker exec -it NODE-NAME COMMAND-NAME`. It uses a user called `kubernetes-admin`. This user has permissions that differ from those of the entered `node`. Investigating the possibilities of attacks through Docker and defenses against such attacks might be interesting in the future.

Machine learning

We manually looked at what syscalls to block, whilst machine learning could potentially help with this. Machine learning might be a good tool to determine what syscalls should be blocked. One study already used machine learning to find patterns in the syscall usage of crypto mining attacks [10]. The study focused on a pattern of syscalls, which means it did not only look at single syscalls. `Seccomp` looks at single syscalls without considering the context or pattern. It might be interesting to investigate whether using the context or patterns of syscalls could attain better results regarding the identification and prevention of lateral movement. Machine learning might also be used in different methods like anomaly detection.

Chapter 7

Conclusions

This research aimed to investigate the identification and prevention of lateral movement in the Kubernetes cluster using `seccomp`. This was done using two clusters and two attack scenarios based on the attacker model. The results of this study show that `seccomp` is not usable in every scenario. `Seccomp` cannot be used in privileged `pods`. In the context of unprivileged `pods`, `seccomp` can be used, but it depends on the scenario whether it is helpful in identifying and preventing lateral movement.

The tested attacks used syscalls that are also used during the creation phase and regular usage of the compromised `pod`. Our research showed that using `seccomp` to block these syscalls is impossible. Blocking syscalls used during container creation makes it impossible to create the compromised `pod` before the attack can even be tested. Blocking syscalls used during the regular usage of the compromised `pod` makes it impossible for `pod` to function correctly. Therefore, these syscalls cannot be blocked. It is possible, however, to block any other syscall, which means that `seccomp` can be used to block specific attacks that use such syscalls. For this to be an option, the attack must use at least one syscall not used in container creation nor during regular usage of the `pod`. Another option would be to focus on chains of syscalls instead of single syscalls. This might provide different results in what we can and cannot block.

Aside from blocking, this research showed that using `seccomp` for logging the syscalls was possible in unprivileged `pods`. The problem with this, however, is that it is questionable how informative the logs are. If the attack uses syscalls that are also used in regular usage of the `pod`, it is not easy to distinguish between attacks and regular usage. Combining logging with other measures could make this easier. As mentioned in Section 6.1.1, combining the information from logging with RBAC-A could improve the effectiveness of using `seccomp`. Using the additional information, it might be possible to distinguish between attacks that resemble regular usage of the compromised `pod` and the actual regular usage of the compromised `pod`. Another option would be to use anomaly detection and machine learning to distinguish them.

Logging in combination with additional information could provide information about lateral movement in the Kubernetes cluster. Another case where `seccomp` could be helpful in logging is when the attacks use syscalls that are not part of the regular usage of the compromised `pod`. However, using `seccomp` to block these syscalls would also be possible under these circumstances. Instead of only identifying the lateral movement, it would be possible to prevent the lateral movement directly.

In our experiment, we created two clusters with Kind, as `seccomp` works in Kind. There was no clear information about whether `seccomp` works in Minikube, indicating that not all Kubernetes ecosystem tools support every possible security mechanism. Because of this, it is crucial to choose the correct tools and security mechanisms when working with Kubernetes.

In summary, in our attacker model, our research shows that `seccomp` cannot be used to prevent lateral movement in every scenario. In contrast, `seccomp` can be used to identify lateral movement in the context of unprivileged `pods`. However, it is important to consider the limitations of this study, such as the limited attack scenario coverage, the experimental setting, and the focus on `seccomp`. It might be necessary to combine `seccomp` logging with additional information to minimize the number of false positives when identifying lateral movement. Other techniques to identify or prevent lateral movement in Kubernetes may exist but were not found in our research. Future research is needed to understand the effects of these techniques better.

Acknowledgments

First of all, I would like to express my deepest appreciation to my external supervisor at Sue, Nathan Keyaerts, for his valuable guidance, support, and provided knowledge throughout my research. We had regular meetings to discuss progress and ideas, which contributed to the success of this research. I would like to extend my sincere thanks to my internal supervisor at Radboud University, Pol van Aubel. The regular meetings in which we discussed progress, ideas, and feedback were extremely helpful for the progress of this research. Thanks should also go to Reinier Goeman within Sue for his support, open communication, and encouragement during the process. I am also grateful to all my fellow interns and co-workers at Sue for their valuable contributions. Their help, knowledge, and support proved very helpful to this research. Finally, a special thanks to Harald Vranken, my second assessor, for his willingness and expertise to assess my research.

References

[1] N. Sfondrini, G. Motta, and A. Longo, "Public cloud adoption in multinational companies: A survey," in *2018 IEEE International Conference on Services Computing (SCC)*, IEEE, 2018, pp. 177–184.

[2] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *2020 IEEE Secure Development (SecDev)*, pp. 58–64, 2020.

[3] S. I. 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*, 2021, pp. 1689–1690.

[4] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," *IEEE Security & Privacy*, vol. 19, no. 5, pp. 46–56, 2021.

[5] P. Mytilinakis, "Attack methods and defenses on Kubernetes," M.S. thesis, Πανεπιστήμιο Πειραιώς, 2020.

[6] M. Bagherzadeh, N. Kahani, C.-P. Bezemer, A. E. Hassan, J. Dingel, and J. R. Cordy, "Analyzing a decade of Linux system calls," *Empirical Software Engineering*, vol. 23, pp. 1519–1551, 2018.

[7] S. Majumdar *et al.*, "Leaps: Learning-based proactive security auditing for clouds," in *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*, Springer, 2017, pp. 265–285.

[8] H. Kermabon-Bobinnec *et al.*, "Prospec: Proactive security policy enforcement for containers," in *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*, 2022, pp. 155–166.

[9] S. Bagheri, H. Kermabon-Bobinnec, S. Majumdar, Y. Jarraya, L. Wang, and M. Pourzandi, "Warping the defence timeline: Non-disruptive proactive attack mitigation for kubernetes clusters," in *ICC 2023-IEEE International Conference on Communications*, IEEE, 2023, pp. 777–782.

[10] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE transactions on parallel and distributed systems*, vol. 32, no. 3, pp. 674–691, 2020.

[11] H. Gantikow, C. Reich, M. Knahl, and N. Clarke, "Rule-based security monitoring of containerized workloads," SCITEPRESS-Science and Technology Publications, 2019.

[12] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings 1996 IEEE symposium on security and privacy*, IEEE, 1996, pp. 120–128.

[13] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2016, pp. 467–478.

[14] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, 2008.

[15] S. S. Ledaal, "Tapping network traffic in Kubernetes," M.S. thesis, University of Agder, 2023.

[16] A. Halinen, "Security risks for sidecar containers in kubernetes," 2024. [Online]. Available: `https://aaltodoc.aalto.fi/server/api/core/bitstreams/afda5fe4-edbe-43bb-996f-92ff0a663977/content`.

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

[18] A. Huth and J. Cebula, "The basics of cloud computing," *United States Computer*, pp. 1–4, 2011. [Online]. Available: `https://findnerd.s3.amazonaws.com/data/152759075583.pdf`.

[19] S. P. Mirashe and N. V. Kalyankar, "Cloud computing," *arXiv preprint arXiv:1003.4074*, 2010.

[20] *AWS Cloud Practitioner Essentials*, `https://explore.skillbuilder.aws/learn/course/134/aws-cloud-practitioner-essentials`, Accessed: 23 Apr, 2024.

[21] B. Burns and C. Tracey, *Managing Kubernetes: operating Kubernetes clusters in the real world*. O'Reilly Media, 2018.

[22] N. Poulton and P. Joglekar, *The Kubernetes book*. Independent, 2020.

[23] A Cloud Guru, *Certified Kubernetes Administrator Course*, `https://learn.acloud.guru/course/certified-kubernetes-administrator/dashboard`, Accessed: 21 Feb, 2024.

[24] Atlassian, *Kubernetes vs. Docker*, Accessed: 14 Feb, 2024. [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker.

[25] Docker, *What is a Container?* Accessed: 14 Feb, 2024. [Online]. Available: https://www.docker.com/resources/what-container/.

[26] O. Ben-Kiki, C. Evans, and I. döt Net, *YAML Ain't Markup Language (YAML) Version 1.2.2*, https://yaml.org/spec/1.2.2/, Accessed: 31 Jul, 2024, 2021.

[27] GeeksforGeeks, *Kubernetes Cloud Controller Manager*, Accessed: 5 Aug, 2024, 2023. [Online]. Available: https://www.geeksforgeeks.org/kubernetes-cloud-controller-manager/.

[28] A Cloud Guru, *Certified Kubernetes Security Specialist Course*, https://learn.acloud.guru/course/certified-kubernetes-security-specialist/dashboard, Accessed: 29 Mar, 2024.

[29] Kubernetes Authors. "Seccomp Security Profiles for Pods." Accessed: 1 Mar, 2024. (2023), [Online]. Available: https://kubernetes.io/docs/tutorials/security/seccomp/.

[30] D. Carr. "Seccomp Notifier: Bring Seccomp Information Directly to Userspace." Accessed: 1 Mar, 2024. (2022), [Online]. Available: https://kubernetes.io/blog/2022/12/02/seccomp-notifier/.

[31] 4ARMED, *Kubernetes Seccomp*, https://www.4armed.com/blog/kubernetes-seccomp/, Accessed: 9 Apr, 2024.

[32] Kubernetes Authors, *Kubernetes Documentation*, https://kubernetes.io/docs/reference/networking/ports-and-protocols/, Accessed: 21 Mar, 2024.

[33] M. C. Sánchez, J. M. C. de Gea, J. L. Fernández-Alemán, J. Garcerán, and A. Toval, "Software vulnerabilities overview: A descriptive study," *Tsinghua Science and Technology*, vol. 25, no. 2, pp. 270–280, 2019.

[34] Sysdig, *Falco*, https://falco.org/, Accessed: 1 Mar, 2024.

[35] Palo Alto Networks. "Breaking Docker via RunC: Explaining CVE-2019-5736." Accessed: 2 Apr, 2024. (2019), [Online]. Available: https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/.

[36] Palo Alto Networks Unit 42, *Unsecured kubernetes instances*, https://unit42.paloaltonetworks.com/unsecured-kubernetes-instances/, Accessed: 3 Apr, 2024, 2021.

[37] Openwall. "Open source software security (oss-security) mailing list archives." Accessed: 2 Apr, 2024. (2016), [Online]. Available: https://www.openwall.com/lists/oss-security/2016/11/23/6.

[38] A. Datta, *Kubernetes namespace breakout using insecure host path volume part 1*, https://blog.appsecco.com/kubernetes-namespace-breakout-using-insecure-host-path-volume-part-1-b382f2a6e216, Accessed: 3 Apr, 2024, 2020.

[39] R. Glushach. "Monitoring and logging in kubernetes: Unlocking efficiency and insights." Accessed: 2 Apr, 2024. (2023), [Online]. Available: https://romanglushach.medium.com/monitoring-and-logging-in-kubernetes-unlocking-efficiency-and-insights-7b84c1d7b3ed.

[40] A. Pratap. "Logging and monitoring in kubernetes." Accessed: 2 Apr, 2024. (2023), [Online]. Available: https://dev.to/adityapratapbh1/logging-and-monitoring-in-kubernetes-53o2.

[41] eBPF.io authors, *eBPF - Extended Berkeley Packet Filter*, https://ebpf.io/what-is-ebpf/, Accessed: 5 Apr, 2024.

[42] Tigera. "eBPF Overview - Tigera." Accessed: 17 Apr, 2024. (2024), [Online]. Available: https://www.tigera.io/learn/guides/ebpf/.

[43] C. N. C. F. (CNCF), *Projects*, https://www.cncf.io/projects, Accessed: 10 Apr, 2024.

[44] eBPF.io, *eBPF Applications*, https://ebpf.io/applications/, Accessed: 10 Apr, 2024.

[45] Sysdig, *Getting started with runtime security and falco*, https://sysdig.com/blog/intro-runtime-security-falco/, Accessed: 1 Mar, 2024.

[46] Sysdig, *Rules examples*, https://falco.org/docs/reference/rules/examples/, Accessed: 1 Mar, 2024.

[47] a. Bala, *List of kubernetes rbac rule verbs*, https://stackoverflow.com/questions/57661494/list-of-kubernetes-rbac-rule-verbs, Accessed: 27 Jun, 2024.

[48] K. Documentation, *Kubectl exec*, https://kubernetes.io/docs/reference/kubectl/generated/kubectl_exec/, Accessed: 27 Jun, 2024.

[49] F. Valsorda, *Linux syscall table*, https://filippo.io/linux-syscall-table/, Accessed: 4 Apr, 2024.

[50] *Vmware fusion evaluation*, https://www.vmware.com/products/fusion/fusion-evaluation.html, Accessed: 18 Apr, 2024.

[51] Ubuntu, *Download ubuntu*, https://ubuntu.com/download, Accessed: 18 Apr, 2024.

[52] Marki555, *Role of kauditd_printk_skb in linux kernel*, https://stackoverflow.com/questions/54955662/role-of-kauditd-printk-skb-in-linux-kernel, Accessed: 12 Jun, 2024.

[53] Linux man-pages project, *Cat(1) manual page*, https://www.man7.org/linux/man-pages/man1/cat.1.html, Accessed: 27 Jun, 2024.

[54] L. man-pages project, *Bash(1) manual page*, https://www.man7.org/linux/man-pages/man1/bash.1.html, Accessed: 27 Jun, 2024.

[55] LinuxConfig.org, *How to crash linux*, https://linuxconfig.org/how-to-crash-linux, Accessed: 1 Jul, 2024.

[56] R. Kuhn, E. Coyne, and T. Weil, "Adding attributes to role-based access control," 2010.

[57] Kubernetes, *Linux kernel security constraints for pods and containers*, https://kubernetes.io/docs/concepts/security/linux-kernel-security-constraints/, Accessed: 15 Jun, 2024.

[58] Docker, *Install Docker Engine on Ubuntu*, https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository, Accessed: 18 Apr, 2024.

[59] Blaq_2022, *Error when running the command minikube start*, https://stackoverflow.com/questions/72568590/error-when-running-the-command-minikube-start, Accessed: 22 Apr, 2024.

[60] K. SIGs, *Kind - quick start*, https://kind.sigs.k8s.io/docs/user/quick-start/, Accessed: 18 Apr, 2024.

[61] Kubernetes Authors, *Install kubectl on Linux*, https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/, Accessed: 18 Apr, 2024.

[62] Kubernetes SIGs, *Known Issues - Pod Errors Due to Too Many Open Files*, https://kind.sigs.k8s.io/docs/user/known-issues/#pod-errors-due-to-too-many-open-files, Accessed: 29 Apr, 2024.

Appendix A

Appendix

A.1 Installation steps of Docker, Kind, and Kubectl

This section explains the exact steps required to install Docker, Kind, and `kubectl`, as we have done in our research.

A.1.1 Installing Docker

To install Docker, the instructions on the official Docker website were followed [58]:

- `sudo apt-get update`
- `sudo apt-get install ca-certificates curl`
- `sudo install -m 0755 -d /etc/apt/keyrings`
- ```
  sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
  -o /etc/apt/keyrings/docker.asc
  ```
- `sudo chmod a+r /etc/apt/keyrings/docker.asc`
- ```
  echo  "deb [arch=$(dpkg --print-architecture) \
  signed-by=/etc/apt/keyrings/docker.asc] \
  https://download.docker.com/linux/ubuntu \
  $(.  /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
  ```
- `sudo apt-get update`
- ```
  sudo apt-get install docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin
  ```

When trying to use Docker, Docker complained about not having permission to do something. Running Docker as root is not allowed, so it did not fix the problem. To fix the problem, the following commands were used [59]:
  ```

- `sudo chmod 660 /var/run/docker.sock`
- `sudo addgroup --system docker`
- `sudo adduser #USERNAME docker`
- `newgrp docker`

Originally, `chmod 666` was used. This allows the current user, group, and everyone else to read and write the specified file. As this is not a good practice, we set the last digit to 0. This fixed the issue without giving every other user the same permissions.

A.1.2 Installing Kind and Kubectl

After installing Docker, Kind was installed according to the instructions on the Kind GitHub [60]:

- `[ $(uname -m) = x86_64 ] && curl -Lo ./kind \`
  `https://kind.sigs.k8s.io/dl/v0.22.0/kind-$(uname)-amd64`
- `chmod +x ./kind`
- `sudo mv ./kind /usr/local/bin/kind`

To use Kind, `kubectl` is also required. To install `kubectl`, the following commands were used [61]:

- `curl -LO "https://dl.k8s.io/release/$(curl -L -s \`
  `https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"`
- `sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl`

After these steps, Kind can be used to create small clusters. However, there were not enough resources when creating a cluster with more than two nodes. This is a known issue and can be solved by adding two lines to the `/etc/sysctl.conf` file [62]:

- `fs.inotify.max_user_watches = 524288`
- `fs.inotify.max_user_instances = 512`

To enforce the changed settings, the VM needs to be restarted. After restarting the VM, there were no problems when creating a cluster with multiple nodes.