Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Honeyclusters: How to secure untrusted Honeypot workloads in Kubernetes

Saul O'Driscoll

August 7, 2021

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
B.A. (Computer Science and Business)

TABLE OF CONTENTS

1. Abstract and Introduction 2

1.1. Relevant Terms Defined 2

2. Rise of Microservices and Containers 4

3. History of Hypervisors and Containers 5

4. Contemporary Security Principles, Paradigms & Mental Models 7

5. Traditional Security vs. Container Security 10

6. What is Kubernetes? 12

7. Kubernetes under the Hood: Node Components 16

8. Kubernetes and Container Security 22

9. Container Threat Modelling 24

9.1 Container Hardening - Tier 1: (Processes, Namespaces and Cgroups) 29

9.2 Container Hardening - Tier 2: (Seccomp-bpf, AppArmor and SELinux) 32

9.3 Container Hardening - Tier 3: (gVisor, Micro VMs, Google SLSA, Trivy) 36

10. Kubernetes Threat Modelling 50

11. Kubernetes Hardening 53

12. FYP Implementation: Honeyclusters 58

12.1 Honeycluster Components 59

12.2 Honeycluster Hardware 60

12.3 Honeycluster Design Choices (K8s Distro, Runtime, Cloud Provider) 61

13. Local Development Flow 67

Conclusion 68

Appendix 68

Honeytokens 68

The age of Serverless 69

Knative 71

Bibliography 72

1. Abstract and Introduction

Developer Operations (DevOps) teams have started to break up their monolithic applications into containerised microservices because it allows for better scaling, increased uptime and more security due to properties such as reduced attack surfaces. Due to how standard container runtimes work, vulnerabilities can now occur in a paradigm previously unseen in virtual machines (VM) based monolithic software stacks. We explore techniques used to approach a VM-esque level of isolation while retaining the benefits that come from microservice based architecture. This extra level of isolation is useful in sites with remote execution, untrusted server workloads, highly secure information storage and air gapped systems. We use honeypots as an example of an untrusted workload. Honeypots are intentionally exposed applications, systems or containers used to get real time data about what exploits are being used and how to defend against them. Kubernetes is used to orchestrate these containerised workloads. We discuss, evaluate and implement techniques used to increase the isolation and sandboxing capabilities of containers.

1.1. Relevant Terms Defined

DevOps

Devops is the combination of the development and the operation of software. Previously (and still in many companies to this day), the development of software and the later deployment and management of the software is done by two separate teams. Over the years, in the search for continuous delivery, short development cycles, rapid feature releases and high availability, these two teams have slowly merged. This has led to a shift in thinking about how software is deployed but also how it is written to be more DevOps friendly. A large amount of tooling and software surrounding logging, orchestration and deployment has sprung up directly from the DevOps movement.

DevSecOps

DevSecOps is a less widely used term. The “Sec” part stands for security and the overarching mindset in DevSecOps is that everyone from devs to ops people are in charge of security and should go about their work with security in mind.

Honeypot

A honeypot is an intentionally vulnerable system, network or account. This could be by leaving default or simple passwords in place, leaking credentials, not patching known exploits or doing anything that intentionally compromises a system. These honeypots can be littered with logging tools so as to collect as much information as possible on attackers in order to study the attacks on the honeypot and gain real time information. There are low and high interaction honeypots. Low interaction honeypots are simplistic honeypots that act as low hanging fruit and tend to be quite unrealistic and easily spotted. High interaction honeypots are the opposite. They are usually fitted with lots of monitoring tools as well as a fully functional filesystem. They can be modelled to reflect a real system with fake sensitive information to get strong data about what kind of attackers are out there.

Container

A container is an application that is bundled with everything it requires to operate including its dependencies and underlying libraries apart from a kernel. Because of their design, containers are somewhat of a silver bullet for the common issue of “it works on my machine, so it should work on yours”. Containers are distributed as images which can be pulled from public or private repositories and run on any machine with a container runtime installed. Containers share the kernel of the underlying OS that they run on and are not bundled with it. This allows for a small image footprint (as small as a few MBs) and fast startup times (a few seconds or less) compared to VMs. Containers are a powerful tool for distributing and managing software, they are often stateless and therefore can be destroyed and relaunched at will without much thought. In many cases containers can be thought of as fungible meaning a container can be swapped for a copy of the same container with very few headaches. A comparison between a herd of cattle (containers) and individual, carefully cared for pets (VMs) is often made. It is important to note that containers do not offer the same base level of isolation as VMs from a security standpoint.

Kubernetes

Kubernetes is a container orchestration tool. Containers have been around for a long time in software and while there was a growing appreciation for their compartmentalisation, they were difficult to manage at scale for a long time. Eventually Google open-sourced their internal container orchestration tool known as Kubernetes. After this, the DevOps community finally had a semi intuitive way of managing containers at scale with access to container specific tools for rollbacks and CI/CD as well as monitoring and logging. Kubernetes itself has a steep learning curve especially when one wants to move into production systems and away from the unsafe defaults that Kubernetes comes with. Oldschool sysadmins and new graduates have to learn an entire new mental framework for how to think about the development and management of software. Kubernetes is and will continue to be a growing staple of how software is run in the future.

2. Rise of Microservices and Containers

As the internet grew there was a growing expectation of low latency, highly and globally available services. Solutions architects faced a few key issues with their VM based monolithic applications:

1. Scaling: Monolith applications are difficult to scale. To solve the scalability problems one can break up a monolithic application into smaller logical components. These components are called microservices.

2. Operational cost: As for any application, the cost of operation increases with the complexity of the workload. Furthermore, updates need to be studied and tested carefully before being deployed.

3. Longer release cycles: Development of monolithic applications becomes significantly harder and more complex as the application grows, so does the maintenance of these applications. Bugs can often come from a totally unrelated corner of the codebase. Testing in large monolithic applications also tends to be more involved because it's harder to test individual components in a void. Same goes for the iteration and integration of new features.

4. Security: Application monoliths have a larger attack surface than microservices and tighter coupling among components making them hard to secure and vulnerable to attackers. Microservices make this easier
only granting files to what something actually needs or only granting network access to what something actually needs

All of these factors extend the release cycles which is the opposite of what contemporary agile software development is all about, namely speedy and secure iteration toward timely feature delivery. There are clear incentives to break up monolithic applications into many smaller microservices.

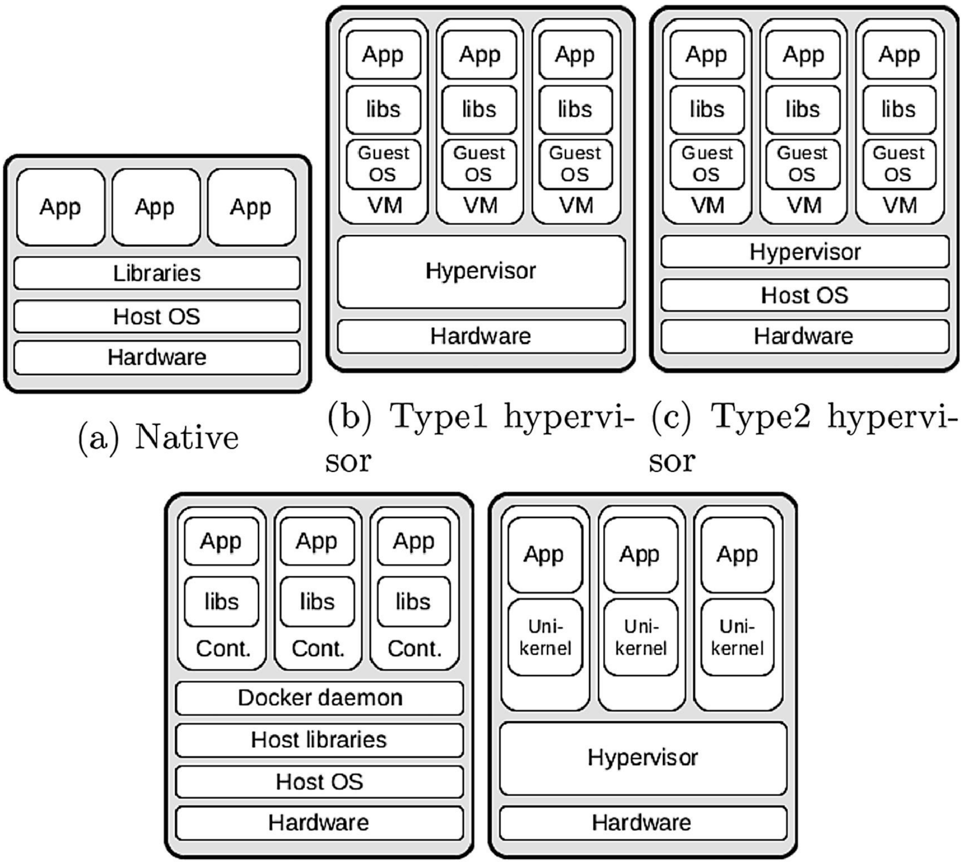
If each microservice of the application has a well defined API, developers can primarily shift their focus on the development of functionality and features of their components. The microservice talks to the other microservices using previously well defined standards. Decoupling the different parts of applications into smaller bite sized chunks leads to less complex logic in each. This makes the individual application easier to debug as well as making it easier to iterate through new versions especially because rolling back becomes cheap/trivial and the modularity of microserves allows for simple dropping in and swapping.

3. History of Hypervisors and Containers

In the early days of computing, each computer was generally used for running a specific application. This meant that if you wanted to host a website you had to run a dedicated server with software like Apache Webserver installed on it. Hosting another website would mean adding an additional machine. This configuration is illustrated in the diagram below under “(a) Native”.

Throughout the 60s and 70s we saw the development of hypervisors. Hypervisors essentially enable you to run multiple instances of operating systems on one host machine. For example system administrators could now run 3 different Linux operating systems on that one machine simultaneously. This enabled system administrators to not have to manage many different machines (each running their own application) and instead manage a smaller amount of ‘beefier’ (more powerful) machines. An added benefit of hypervisors is that they provide a similar amount of isolation to dedicated machines.

There are 2 types of hypervisors. Type 1 Hypervisors are installed directly onto the hardware which is why they are also known as bare-metal hypervisors. Examples of such hypervisors are XenServer, Proxmox, Linux KVM, VMware ESXI and many more. Once installed, one can then spin up virtual instances of any desired OS image like Ubuntu Server 20.04 LTS or Windows Server 2008 on the same machine. Type 2 hypervisors are slightly different because they sit on top of a host OS like Ubuntu, MacOS or Windows. The host OS is installed before the Type-2 Hypervisor. There are tradeoffs to each. Type 2 hypervisors like Oracle’s VirtualBox have the benefit of being a bit more user friendly as they can be installed on existing machines without requiring any reinstallation. Furthermore you get to use the host OS alongside one or more virtual instances.



(a) Native (b) Type1 hypervisor (c) Type2 hypervisor

(d) Container (e) Unikernel

Source: (Martin et al., 2018)

In 1979 moves were made toward creating a mixture of shared, yet isolated environments with the release of the **chroot** (change root) command. With the **chroot** command it was possible to change the apparent root directory for a running process, along with all of its child processes. This was an initial jumping-off point into insulating system processes into their own filesystems without impacting the overall filesystem. In 2000 we got FreeBSD **jails** which offered more powerful sandboxing tooling for users and networks. In 2006 Google launched “process containers” with similar functionality which were renamed to **control groups** (**cgroups**). Socalled **cgroups** are still an integral part of Linux today. In 2008 **LXC** was released which stands for **L**inu**x** **C**ontainers. These were the first true containers for Linux which offered virtualisation at the operating system level as illustrated in figure “(d) Container”.

Most container runtimes (LXC, Docker etc) differ from virtual machines in one fundamental way. They offer multiple isolated Linux environments but ***share the kernel of the underlying host OS***. Each Virtual machine (VMs) running on a hypervisor presides over its own kernel. So why share a kernel?

With a shared kernel, container images can be incredibly small because the kernel does not have to be packaged with the image. For example, Alpine Linux is a Linux distribution that is based on BusyBox and is only 5MB large. Even some of the small OS image ISOs like the DietPi OS for Raspberry Pi's are still larger than 1GB. Additionally the startup times of containers are considerably lower compared to those of VMs, with some having startup times under 0.5 seconds. On the other hand, and one of the largest trade offs, is that containers do not offer nearly as much isolation as VMs. We will explore the security implications of this and other container properties extensively in the sections "Container Threat Model", "Kubernetes Threat Model" and solutions in "Container Hardening" and "Kubernetes Hardending".

Over the last 5 years Linux has continually added to their tooling around process isolation. Features such as "control groups", "namespaces", "seccomp" and "apparmor" were introduced and will be explained in following sections. Collectively these are grouped into tools that enable containers and their security.

The question this paper seeks to answer is, how does one leverage current state of the art container isolation tools and techniques in order to isolate these instances?

Project Implementation Notes / Relevance:

- We use gVisor containers which use **runsc** container runtime, a derivative of **runc** used in **Docker** and **containerd**
- Seccomp is enabled by default as of Kubernetes 1.22 (4th August 2021)
- The honeypot image (Cowrie) is prebuilt and pulled from Docker Hub

4. Contemporary Security Principles, Paradigms & Mental Models

Breaking down everything there is to know about information security practices is a monumental task. This section intends to distill and serve up existing research acutely relevant to this project. The information here is an amalgamation of research in books, academic papers and documentation.

Intuition is one of the most valuable tools to have but hardest things to teach. If you ask an expert in their field why they do a certain thing a certain way, they may have a hard time articulating why or give an answer like "that's just the way it's done". Good teaching will usually lead with why things are done a certain way and give reasons.

Here we will outline general cybersecurity concepts so as to offer a robust first principle framework that one can apply when thinking about securing any sort of system.

4.1 Principle of Least Privilege

The principle of least privilege states that one should limit each user or component to the bare minimum of privileges which they need to perform their function. For example a microservice in charge of sending out the weekly newsletter does not need to have permissions to modify payment information.

4.2 Defence in Depth

The principle of defense in depth states that any system should have multiple defence boundaries rather than a single one, similar to the layers of an onion. This is to protect against things like zero-day exploits (exploits on the day were discovered before the vendor makes a patch) are inherently unpatchable meaning that if an attack breaks past your “one big wall” they are completely free to take over the system. It is better to have multiple checkpoints and layers to your cybersecurity plan this way an attacker has to jump through many hoops while simultaneously trying to avoid detection.

4.3 Reducing Attack Surfaces & Shrinking Trust Boundaries

Reducing the attack surface ties into what was discussed about the downsides of large monolithic applications. Large and complex systems are harder to secure because many parts are tightly coupled leading to higher risk or blanket compromisation.

DevOps teams will generally have an easier time securing 3 smaller application containers rather than one massive monolithic application with multiple APIs and public ports. Reducing this attack surface and hardening exposed components means that there are less vulnerable access points. Reducing the attack surface also implies reducing the amount of code, users and credentials given out. Microservices and containers fit nicely into the information security ethos because they are a great tool in the war against excessive complexity.

4.4 Honeypots & Intrusion Detection

Intrusion detection are methods and applications applied in order to detect attackers within the system as they roam. The “Mandiant Security Effectiveness Report 2020” studied 100+ enterprise level production environments and found that 53% of infiltrations go unnoticed and 91% of attacks did not generate any sort of alert (“Mandiant Security Effectiveness Report,” 2020).

This is a major issue because the longer an attacker is in your system the more havok they are able to wreak. There is a wide array of tooling surrounding intrusion detection such as antivirus software, log parsing and network traffic monitoring.

As previously mentioned, honeypots are intentionally vulnerable and purposely exposed networks or systems used to attract attackers and study their exploits and behaviours in order to defend against them. There are different types of honeypots but generally they can be divided into highly interactive and low interaction. Low interaction honeypots are quite simplistic decoys that might have something resembling a filesystem and are supposed to act as low hanging fruit. Usually the information gained from low interaction honeypots is not as comprehensive compared to the high interaction honeypots however they are easier to develop and cheaper to deploy. High interaction honeypots are often modeled closely after actual production systems with fake sensitive information. They are sometimes placed right alongside production systems but with easily bruteforcable credentials or vulnerable backdoors in order to gain strong data about the exact types of attackers targeting your kind of system. Attackers will usually spend more time in these systems, sometimes installing extra applications or upload/download files from the system. Cowrie, the honeypot used in this project, is highly interactive and keeps track of files uploaded and downloaded to the instance.

In order to be maximally effective, high and low interaction honeypots should preside over extensive logging functionality in order to extract the maximum amount of information from attacks. Logs can be kept within hidden folders in the container or forwarded to a remote log aggregator.

It is viable to build your own honeypot however this project uses an existing open source honeypot due to the vast amount of open-source honeypot projects online (paralax, 2021).

There is an actively maintained repository for the Cowrie docker container on Github (“cowrie/docker-cowrie,” 2021). There is also a Docker Hub Image (“cowrie/cowrie - Docker Image | Docker Hub,” 2021).

Cowrie comes with a rich featureset (Cowrie, 2021):

- Fake filesystem with the ability to add/remove files.
- A full fake filesystem resembling a Debian 5.0 installation is included.
- Cowrie saves files downloaded with wget/curl or uploaded with SFTP and SCP for later inspection
- Session logs are stored in an UML Compatible format for easy replay with the bin/playlog utility.
- SFTP and SCP support for file upload
- Logging of direct-tcp connection attempts (ssh proxying)
- Forward SMTP connections to SMTP Honeypot (e.g. mailoney)
- JSON logging for easy processing in log management solutions

Project Implementation Notes / Relevance:

- Throughout the design of the Honeycluster we keep these security principles in mind
- Cowrie, the honeypot we use, is considered a highly interactive honeypot
- There is not intrusion detection or logging implemented outside of Cowrie.

5. Traditional Security vs. Container Security

In section “1.0 Rise of Microservices and Containers”, it was described how containers work and how they fundamentally differ from hypervisors by sharing the kernel in a multi-tenant environment. By sharing the kernel with the host machine, containers can shorten the syscall execution path and remove the virtual hardware layer. This means containers can also share software resources like libraries to avoid code duplication. This fundamental difference enables container images to be as small as a few megabytes, for example Alpine Linux - a fully fledged unix distribution - and offer faster startup times as well as more nimble code execution.

The container ecosystem has uprooted the entire CI/CD (continuous integration/continuous deployment) pipeline. From the container building, container orchestration and the various repositories from which containers are pulled like Docker Hub (“Docker Hub Container Image Library | App Containerization,” 2021) or GCR Google’s Container Repository (“Google Container Registry,” 2021). Herein lie some of the key differences between traditional and container vulnerabilities.

From a security standpoint the threat model and vulnerability analysis framework has also completely morphed compared to a typical on-prem server architecture. These deployment chains are “often composed of third party elements, running on different platforms from different providers, raising concerns about code integrity” (Martin et al., 2018). While these deployment chains are vital to the container ecosystem they also introduce new attack vectors. We will begin by taking a broad information security view from an operating system standpoint and then dive deeper into each part of the development pipeline.

5.1 Operating System Vulnerabilities

This paper will not go in depth into operating system vulnerabilities which is why this section comes before the others and is kept brief. In most cases (apart from serverless deployments) containers and container orchestration software has to run on some sort of operating system. If an attacker has access to the virtual machine that the containers are running on then any container hardening efforts are futile. Before thinking about how containers may be attacked, one has to consider the security of the underlying virtual machine and operating system. In production you will usually have multiple containers running on a virtual machine or on bare-metal. If you're using container orchestration software like Kubernetes then usually the developer rarely needs to touch the operating system because Kubernetes will handle all the containers running. Nowadays, most servers are not provisioned manually i.e. no one is manually going around and installing the necessary libraries via the command line using a package manager. This is usually automated by one of many provisioning tools such as:

- Chef https://docs.chef.io/
- Ansible https://www.ansible.com/
- Salt https://saltproject.io/
- Puppet https://puppet.com/
- Terraform https://www.terraform.io/

How they are used and their pros and cons fall outside the scope of this project. However it is vital that the underlying operating system is correctly configured and regularly patched. Unintentionally exposed ports, unpatched packages and misconfigured SSH access or permissions have been the achilles heel and root of countless security breaches.

In this project we are running our containers on Minikube for local development and in Google GKE which is used to provision and manage the Kubernetes nodes. Both of these come with a robust set of safe defaults, especially Google's GKE. More details on this in the Project Architecture chapters.

5.2 Mitre's CVE (Common Vulnerabilities and Exposures) & Vulnerability Scanning

The Common Vulnerabilities and Exposures (CVE) system acts as a reference for vulnerabilities discovered. It is maintained by the MITRE Corporation and funded by the United States Department of Homeland Security. It was launched in September of 1999. The CVE database adheres to a certain set of standards, all vulnerabilities have a similar looking code and are therefore easily queryable. An example of this would be the famous OpenSSL heartbleed vulnerability with the CVE code: (**"**CVE - CVE-2014-0160," 2014).

Docker has its own built in docker image vulnerability scanning tool ("Vulnerability scanning for Docker local images," 2021) and there are also 3rd party services that handle vulnerability scanning as or as part of their service offering for example Aqua Security's vulnerability scanning ("Container Vulnerability Scanning for Cloud Native Applications," 2021).

It is important to note that if an attacker can list all images running on a host or within your Kubernetes cluster then they can apply their own vulnerability scanning tools in order to scan for possible exploits. It is a very useful tool and good practice to scan your images for known vulnerabilities before you deploy them as part of your deployment pipeline.

A few interesting and notable papers on surrounding container vulnerabilities:

- Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities | Banyan Security (Desikan, 2015)
- A Defense Method against Docker Escape Attack (Jian and Chen, 2017)
- DDos Attacks (Chelladhurai et al., 2016)

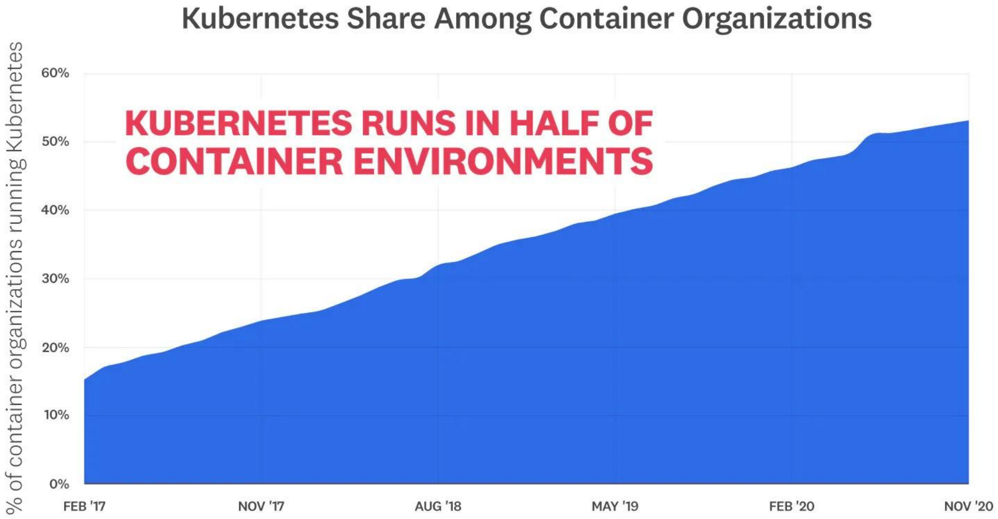
Project Implementation Notes / Relevance:

- Cluster is provisioned manually using the Google Cloud SDK and not using any provisioning tools.
- The underlying OS is the latest secure Google Container Optimised OS.
- Latest Kubernetes version is used
- Images deployed to the cluster have been scanned for vulnerabilities using Trivy which uses the CVE database

6. What is Kubernetes?

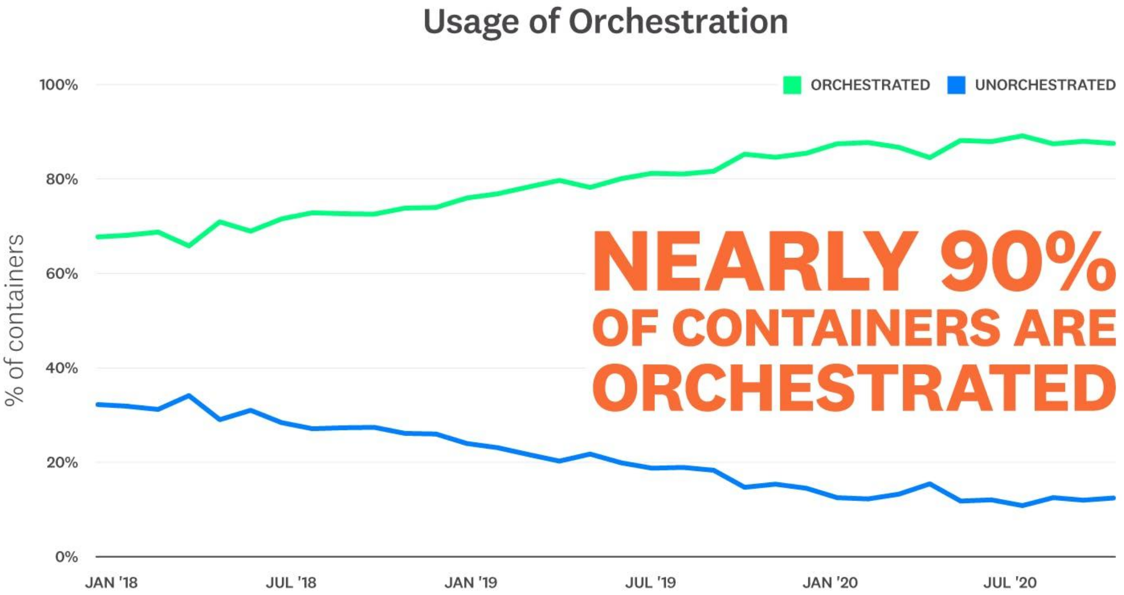
Kubernetes (https://kubernetes.io/) is a container orchestration framework originally developed by Google in-house as their third container-management system. It was eventually open-sourced (https://github.com/kubernetes/kubernetes) in mid-2014 and now leads the charge among container orchestration frameworks. As previously mentioned, the move into the cloud and off-prem has meant that DevOps teams have the ability to distribute their infrastructure across multiple zones or regions (“Regions and zones | Compute Engine Documentation,” 2021) in pursuit of high availability (HA) and low latency operations. Kubernetes has managed to abstract many key components native to large scale deployments and designed comprehensive tooling around them. This of course means that Kubernetes’ extensive toolkit comes with a steep learning curve and is often not recommended for smaller teams because it may introduce unneeded complexity not required for simple systems as well as room for error. A mismanaged or misconfigured kubernetes cluster can be highly vulnerable to attacks, many of which are unique to containerised deployments.

The company Datadog (https://www.datadoghq.com/) which offers “Cloud Monitoring as a Service” does yearly surveys on the state of containers, serverless and container orchestration. Datadog’s 2020 container report showed that despite its steep learning curve, Kubernetes has seen “robust adoption” and spread like wildfire among container organisations with over 50% using Kubernetes as their container orchestration platform (Datadog, 2020).



Source: Datadog

Additionally, the survey showed that due to the size and complexity of container environments “nearly 90% of containers are orchestrated” in one way or another. It is generally unnecessary and prohibitively complex to “roll-your-own” container orchestration and of these 90% most use either Kubernetes, Amazon’s Elastic Container Service (“Amazon ECS | Container Orchestration Service | Amazon Web Services,” 2021), Apache Mesos (“Apache Mesos,” 2021) or Nomad by HashiCorp (“Nomad by HashiCorp,” 2021).



Source: Datadog

Kubernetes can be described as a sort of operating system for containers. One of the key features of Kubernetes is that it abstracts the underlying physical infrastructure (servers and devices at the edge) allowing users of Kubernetes to not have to imperatively think about which machines to run which applications on. At its core, the philosophy of Kubernetes states that: “Everything in Kubernetes is a declarative configuration object that represents the desired state of the system. It is the job of Kubernetes to ensure that the actual state of the world matches this desired state” (Kelsey et al., 2017). Before taking a closer look at how Kubernetes does this, let's use an example to show how Kubernetes’ declarative configurations work.

6.2 Example: Declarative vs Explicit

Kubernetes defines all objects using an Infrastructure as Code framework. Kubernetes objects are generally defined in YAML files which are then applied to a cluster.

Let's say you have an application and you want to be running 3 instances of it at all times no matter what. In a virtual machine environment one would have to decide what machines to run those applications on, deploy them (either manually or using an automated provisioning framework like Ansible or Puppet) and then continually monitor those applications using health checks. If one of them goes down or stops working then the administrator will manually (or via a script) restart that application or server and make sure it is running again. This requires thought to be put into not only the architecture but the developer has to include a way to monitor the status of the application during deployment and development.

Kubernetes takes a different approach. As an administrator - when you’re first configuring your cluster - you define what nodes (in a kubeconfig file) i.e machines will be in your cluster. This is so Kubernetes knows what resources it has access to. Usually, once this part is done, the administrator will have to devote a minimal amount of thought to managing the physical infrastructure unless there is an expansion, package update, or OS update.

In our example scenario the administrator has received a container image from the developer team (e.g. uploaded to DockerHub) and has been asked to run 3 independent instances of it at all times.

Now the administrator has 2 opinions:

1. Manually tell the cluster to:
    a. Download and run an image
    b. Specify the namespace

    c. Scale the image to 3 replicants

2. Simply **apply** a single YAML file to the cluster with all the instructions in it.

Option 1: Explicit method, would look a little something like this:

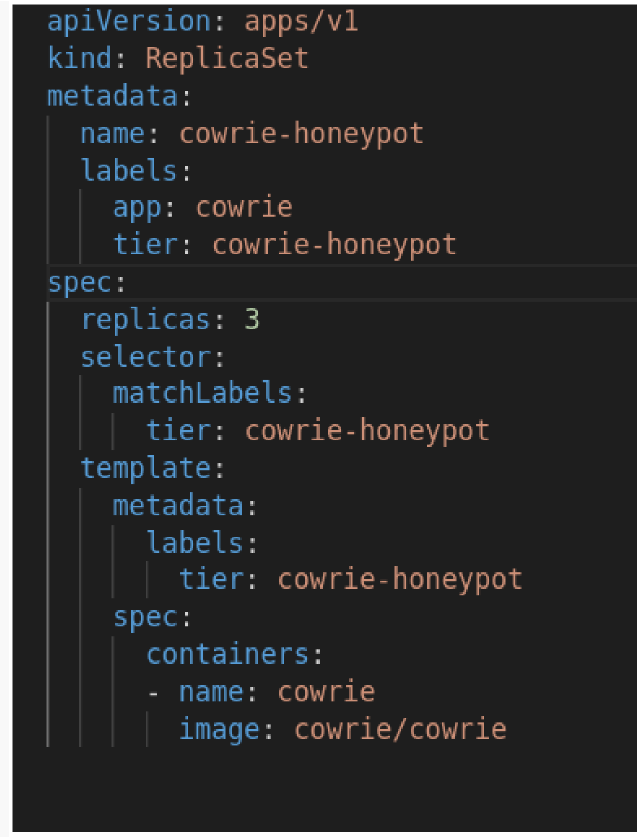
Using **kubectl** and explicitly start a pod and then, in a separate command, tell Kubernetes to run 3 “replicas” of the application using the following commands

1. **kubectl run cowrie --image=cowrie/cowrie --namespace=honeypot**
2. **kubectl scale --replicas=3 pod/cowrie**

Option 2: Declarative, would look like this:

Alternatively, the administrator can define what they want their cluster to look like in a YAML file and then apply that YAML file to the cluster.

cowrie_replicaset.yaml



```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: cowrie-honeypot
  labels:
    app: cowrie
    tier: cowrie-honeypot
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: cowrie-honeypot
  template:
    metadata:
      labels:
        tier: cowrie-honeypot
    spec:
      containers:
      - name: cowrie
        image: cowrie/cowrie
```

CLI Commands:

1. **kubectl apply -f cowrie_replicaset.yaml**

The ‘magic’ of kubernetes is that, once you have your ‘ReplicaSet’ of 3 pods running, if one of these pods dies or is destroyed, Kubernetes will realise this and start a new one instantly without the intervention of an administrator. It will try to constantly maintain the state of the system as is defined in commands given to it or submitted in the YAML file.

The results of the two examples above will be the same however it is highly recommended that one does things as defined in the second example. It is important to understand why.

First of all, if we are applying specifications (YAML files) then we can go back and easily audit what was done to the cluster and point to a specification that was applied.

As we saw in the section on containers, it is generally not advised to SSH into a container and change config files or settings, one should instead tear down the container and change the **Dockercompose** file that leads to that container being created, change it and then destroy and relaunch the container. This way the change is replicable, reusable and documented. In Kubernetes it is similarly a good idea to change the Kubernetes YAML file that is applied to the container rather than issue one time commands to the container and re-apply it.

Furthermore, system administrators often use bash or shell commands to automate DevOps tasks and usually, if a bash script fails half way through then there is no easy way to roll back the changes that were made up to that point. Often this can lead to a "half-configured" system which can be a mess to clean up. Kubernetes offers built-in tooling around rollbacks to previous states.

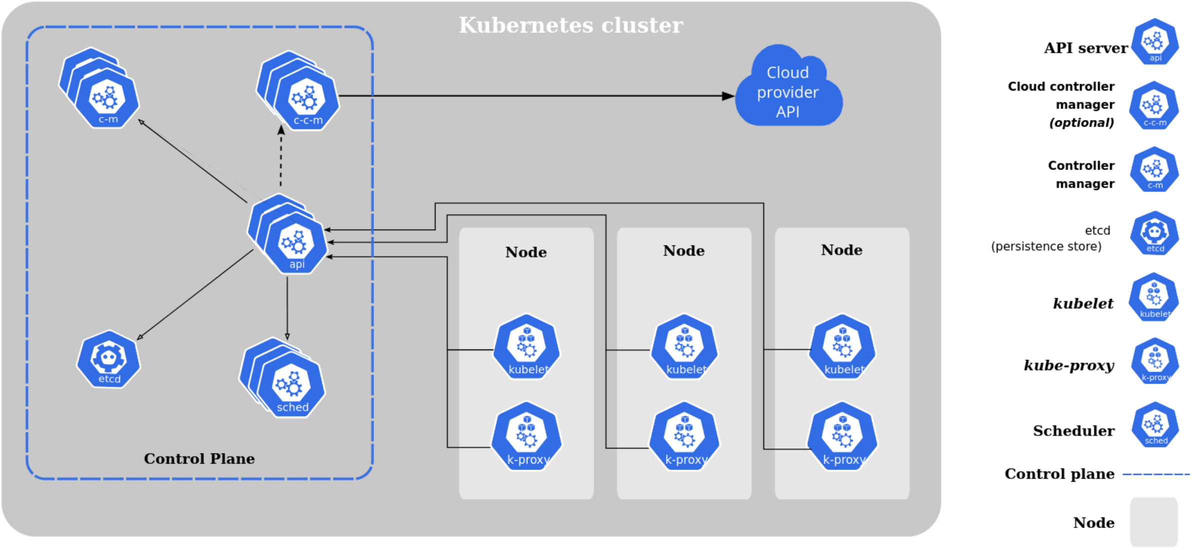
Lastly, by defining the state of a Kubernetes object in a YAML file, one can be sure nothing has changed between command one (running a pod) and command two (scaling a pod) and from a security standpoint we can be sure that we are not "scaling" a vulnerable container. Rather, we are starting a fresh set of containers, pulled from our defined repository and instantly scaling them.

Project Implementation Notes / Relevance:

- In this project all Kubernetes objects are defined using declarative YAML
- These files are stored on the Github (link in Honeycluster section)
- Cluster can be managed using the default Kubernetes commands
- Google Cloud SDK specific commands for launching the cluster are contained in the Github repo.

7. Kubernetes under the Hood: Node Components

This section will provide a high level overview of the Kubernetes components, how they work together and what this enables. By describing and explaining what components run on different Kubernetes nodes the author hopes to allow the reader to “grok” (gain an intuitive understanding) of how Kubernetes works. There is an abundance of great ‘deep dive’ Kubernetes documentation and technical analysis online however this section is not that and is there to serve as a crash course so as to build a foundation with which to understand the project. If you’re looking for a more extensive resource on how to get up and running with Kubernetes, the “Kubernetes: Up and Running: Dive Into the Future of Infrastructure” book by Brendan Burns, Joe Beda, and Kelsey Hightower is an excellent resource.



Source: (Kubernetes.io, 2021)

Kubernetes comes with a wide array of tools that one would need to manage containerised deployments. It has replaced the traditional VM-based way of doing things in thousands of organisations as well as changing the way they handle continuous integration and continuous deployment (CI/CD). While usually this means there is a full migration of workloads and services into container workloads, Kubernetes can also handle hybrid setup with containers and VM workloads . Kubevirt (https://kubevirt.io/) is one such API that enables VMs to be integrated into Kubernetes semi-seamlessly. Now onto different basic Kubernetes terminology

Cluster

A Cluster is the name we will use to describe the sum of all nodes, pods and services. It is essentially our entire application. The word deployment will be used interchangeably.

Pod

Pods can be seen as containers. Generally, in Kubernetes, one pod holds one container however you can also have more than one container in a pod. This can be useful in cases when for instance you want to run a ‘side-car’ daemon in each pod that collects and forwards logs from the main container in that pod to another log aggregator pod.

Node

A node is a server or computer. A node can be a virtual or a physical machine. Nodes host pods. There are two types of nodes: Control Planes (a.k.a “Master Nodes”) and Worker Nodes (sometimes just “Nodes”).

Sidenote: Similar to the nomenclature change that Github made when they changed their “master branches” to “main branches” due to the possible association with the master/slave relationship, Kubernetes has also renamed “Master Nodes” to “Control Planes” in their documentation. Older tutorials and resources may still refer to Control Plane nodes as Master nodes. Going forward the author will try to keep terms uniform and use the terms Control Plane and Worker node however there may be some discrepancies in citations. The terms are interchangeable and both are still widely used.

Each worker node is managed by one or more Control Planes and contains the services necessary to run pods. More information on node components can be found in the **node components** section.

Generally the smallest viable Kubernetes clusters have at least one Worker Node and one Control Plane however there are exceptions to this which are described in the **Local Development Flow** section.

Control Planes (Master Nodes)

The Master Nodes a.k.a the Control Planes are the nodes that hold the Kubernetes API server, Controller Manager, etcd (persistence store) and the Kubernetes scheduler. Optionally it also holds a Cloud Controller Manager which can talk to the Cloud Provider API if Kubernetes is hosted in the cloud. We will take a closer look at these components in the subsequent **“Control Plane Components”** section.

Clusters with two or more Control Planes are called **highly available (HA) clusters** where one Control Plane acts as a fail-safe if one of the others goes down.

Worker Nodes

A worker node is a node which the scheduler can assign pods to run on. There can be one or more Worker Nodes. The Worker Nodes are made up of different components which are outlined in the **Worker Node Components** section. Worker nodes are the nodes that run the actual workloads.

In order for organisations to trust Kubernetes to replace their entire infrastructure is has to be capable of a variety of things:

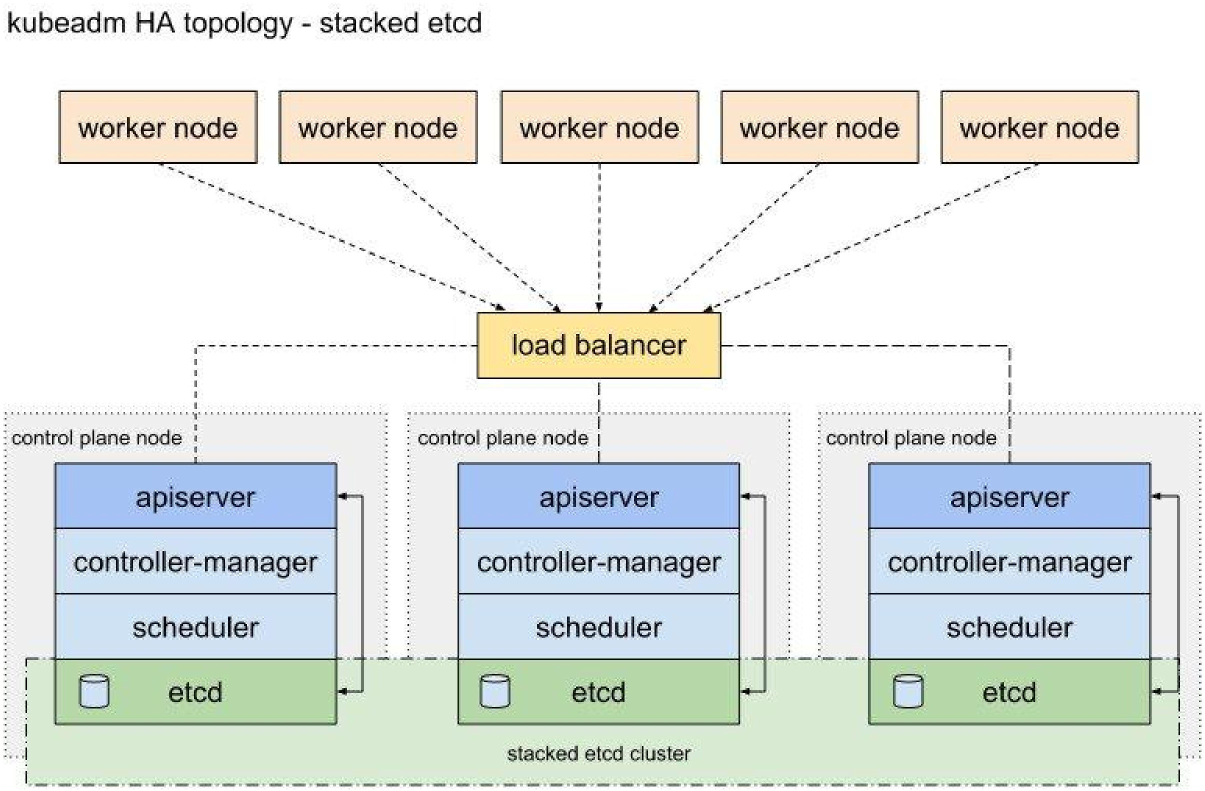
- Load Balancing
- Security
- Authentication
- Logging
- Backups
- Internal and External Networking
- Redundancy
- Storage
- Resource management
- Scheduling
- Secrets management

As shown by example in section 2.1, Kubernetes works by enforcing the desired state principle. A user defines a desired state and Kubernetes makes it so, a very powerful concept which was only a dream for a long time. We saw in the example that this can be done either using imperative commands to the Kubernetes API or via declarative configuration files (YAML files). Almost everything related to Kubernetes revolves around YAML files however one should note that there are “early-stage, experimental projects” like one by AWS called ‘cdk8s’ that allow users to “define Kubernetes native apps and abstractions using object-oriented programming” (https://cdk8s.io/). Cdk8s is not used in this project.

Each Kubernetes cluster has at least one control plane. Within that control plane we have multiple smaller components that enable the features mentioned above. As these components are explained in the following section this should hopefully yield a high-level understanding of how Kubernetes does what it does.

Control Plane Components

The components within the control plane are the brains of the cluster, they handle global tasks like scheduling and are in charge of things like starting a new pod if the “replicas” field of a deployment is unsatisfied e.g. if there are 3 desired replicas of a pods but only 2 running, Kubernetes starts a 3rd one. Generally all control plane components run on the same machine and no application specific containers are run on this machine. There are exceptions to this mantra which are outlined in the Kubernetes development section. (KiND and Minikube and K3s etc). In highly available cluster configurations one can have multiple control planes with a load balancer in front of them so if a control plane were to fail then there will be back ups to avoid the worker nodes being orphaned.



Source: (Kubernetes.io, 2021)

kube-apiserver

The kube-apiserver (Kubernetes.io, 2021) is the component of the control plane that allows the control plane to communicate with the worker nodes and acts as a front end for the Kubernetes control plane. The kube-api server "validates and configures data for the api objects which include pods, services, replicationcontrollers, and others".

kube-controller-manager

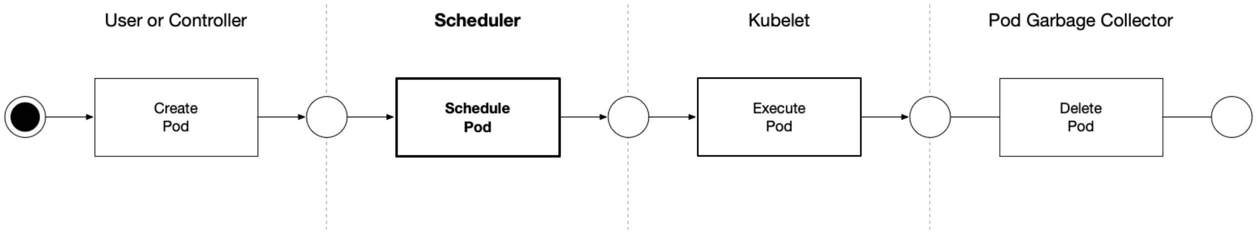
The kube-controller-manager, as the name implies, manages multiple controller processes that run on the control plane. These controllers are all separate processes however "to reduce complexity, they are all combined into a single binary and run in a single process". (Kubernetes.io, 2021)

The most common controller types are:

- Node controller: "Responsible for noticing and responding when nodes go down." (Kubernetes.io, 2021)
- Job Controller: "Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion." (Kubernetes.io, 2021)
- Endpoints controller: "Populates the Endpoints object (that is, joins Services & Pods)." (Kubernetes.io, 2021)
- Service Account & Token controllers: "Create default accounts and API access tokens for new namespaces." (Kubernetes.io, 2021)

kube-scheduler

Newly created pods (created either by the user or a controller) are placed into the Object Store. These pods are called unassigned pods. The Kubernetes Scheduler monitors the Object Store for unassigned pods and assigns it to a node. These assigned pods are then picked up by the “kubelet” processes running on each worker node and executed.



Source: (Tornow, 2018)

Etcd

Etcd is “a distributed, reliable key-value store for the most critical data of a distributed system”. Etcd is not unique to Kubernetes as it is open source software used in other distributed systems however Kubernetes extensively leverages it for storing secrets, the state of the cluster, its configuration, specification and statuses of the running workloads.

Node / Worker Node Components

The control plane usually does not run application workloads and instead schedules pods to be run on worker nodes. These worker nodes have different components compared to the control planes. Node components run on every node and maintain the running pods as well as providing the Kubernetes container runtime environment.

kubelet

The kubelet is “an agent that runs on each [worker] node in the cluster”. Once there are assigned pods the kubelet picks them up and makes sure they are healthy and running according to their PodSpecs which are defined in **YAML** or via other means. Note: the kubelet only manages containers managed by Kubernetes.

kube-proxy

The nodes need a way to handle the networking across the cluster, this is what the kube-proxy is for. Each node has its own internal Cluster IP (INSERT IMAGE HERE OF POD -o wide) and kube-proxy maintains network rules on nodes which allows network communication to your Pods from inside and outside the cluster. The kube-proxy “can do simple TCP, UDP, and SCTP stream forwarding or round robin TCP, UDP, and SCTP forwarding across a set of backends”.

Container runtimes

Container runtimes handle the actual running of the containers and relaying the system calls made by the container to the host machine.

Kubernetes has its own Kubernetes CRI (Container Runtime Interface) which is implemented by other container runtimes to make them compatible with Kubernetes. Currently Kubernetes supports Docker, containerd and CRI-O (an implementation of Open Container Initiative runtime specifically for Kubernetes container runtimes. Initially Kubernetes was using Docker's underlying container runtime however this is being deprecated in v1.20 of Kubernetes in favour of the Kubernetes CRI.

This does not mean Docker images do not work on Kubernetes anymore as "Docker-produced images will continue to work in your cluster with all runtimes, as they always have" (Kubernetes.io, 2020). It is just important that if you're running Kubernetes you have to make sure that your worker nodes are using a supported container runtime before upgrading to a newer version of Kubernetes. As of v1.20 users will get a deprecation warning and in late 2021 and v1.22 of Kubernetes, Docker runtime support will fully be removed and no longer supported by Kubernetes. More insight on the choices made for this project regarding Kubernetes distributions and container runtimes can be found in the section "Honeycluster".

Project Implementation Notes / Relevance:

- In this project we run:
    - 1 managed GKE Control Plane'
    - 2/3 Worker Nodes
        - 1 Secure and hardened gVisor Node
        - 1-2 Price API Nodes
- We use either the gVisor runsc container runtime or the containerd_cos runtime
- Both runtimes are CRI-O compatible

8. Kubernetes and Container Security

This section is broken down into main 2 parts with 2 main subsections:

1. Threat Modeling
    - Container threat modeling
    - Kubernetes threat modeling
2. Hardening
    - Container hardening
    - Kubernetes hardening

Due to the untrusted nature of honeypot workloads, this project revolves around the central theme of security and how it applies to Kubernetes and containerised environments. The following sections will be a detailed examination of:

Kubernetes has made deploying containers at scale a relatively trivial matter. Using a few commands you can have your own cluster and by running:

```
kubectl run honeypot-name --image=<insert-honeypot-image-here> -n mynamespace
```

You can deploy any container to your cluster.

Complexity and creativity is introduced when organisations want to run fast, secure and highly available Kubernetes clusters in production. There is an ‘art’ to running these kinds of secure high performance clusters due to the vast amount of domain knowledge required. A 2019 report by IBM Security showed that “the cost of a data breach has risen 12% over the past 5 years and now costs $3.92 million on average”. In the 3 years leading up to the study, organisations faced a loss of over 11.7 billion records and the study states that “companies need to be aware of the full financial impact that a data breach can have on their bottom line” (IBM, 2019).

Companies are increasingly moving their workloads to the cloud and the need for securely configured infrastructure as well as smooth deployment is the reason for incredibly high salaries for DevOps and Information Security professionals. During the course of the project it became glaringly obvious how wide and deep this field is and that constant vigilance must be practiced if you aspire to work in cybersecurity. The space is constantly evolving and exploit paranoia must be rampant in order to stay safe.

This paper was initially centered around architecting a honeypot cluster in Kubernetes and while this goal was achieved, the main focus became something else. Instead it became a study on how to correctly and securely deploy non-trusted workloads within your cluster and what to pay attention to. Many services allow users to execute unknown or untrusted code on their servers as well as upload untrusted files. These systems need to be hardened and their workloads need to be isolated and sandboxed to avoid any sort of compromise of the system.

Project Implementation Notes / Relevance:

- This section and the ones following it are most relevant to the project
- Untrusted workloads (like honeypots) need very specific kinds of hardening and therefore these threat models are vital
- Not all hardening steps are implemented and not all threats are considered as doing it all would be beyond the scope of this project

- Version 2.0 of the Honeycluster is being designed with significant improvements and will be released down the road.

9. Container Threat Modelling

Project Relevance:

- We have to think differently about hardening OSs versus hardening containers
- Containers are vulnerable in new ways and not vulnerable in others
- Honeypot containers are hardened specifically for the threats they will be facing and therefore not against all conceivable threats to containers

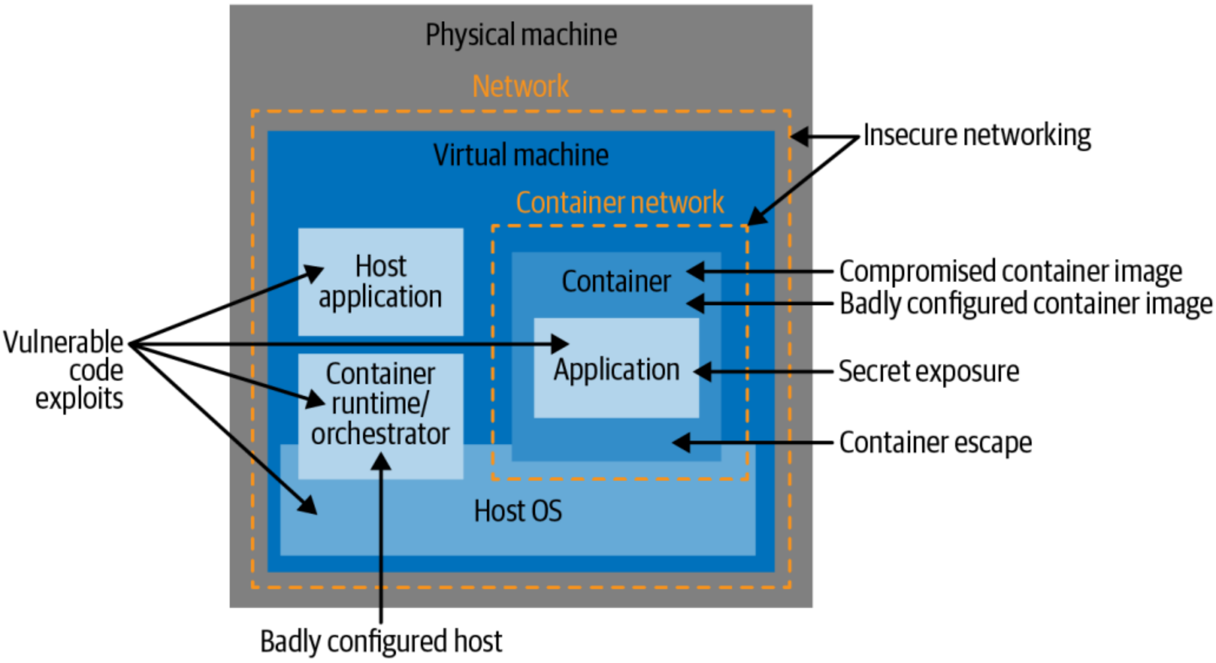
In this section we will take a closer look at how attackers attack containers and clusters of containers. On a macro level one also should consider and model threats surrounding VMs and physical machines as Kubernetes nodes are just that and vulnerabilities here will have a direct effect on the security of the containers and Kubernetes deployments running on them. It falls outside the scope of this project to explore server and VM models. We will begin by examining attack vectors on individual containers and zoom out and look at what additional vectors are introduced when orchestrating a cluster of containers in Kubernetes. This section does not look into what can be done to mitigate these attacks, this is discussed in the “Container Hardening” and “Kubernetes Hardening” section.

There are a few ways to think about threat models. We will take the same approach as the book “Container Security" by Liz Rice. In her book she considers what malicious actors may want to attack our containers

- **External attackers**
- **Internal attackers**
- **Malicious internal actors** such as developers and administrators who have some level of privilege to access the deployment
- **Inadvertent internal actors** (accidental attackers) who may accidentally cause problems
- **Application processes** that, while not sentient beings intending to compromise your system, might have programmatic access to the system

For each actor one must consider a set of permissions:

What access do they have through credentials? For example, do they have access to user accounts on the host machines your deployment is running on?
What permissions do they have on the system? In Kubernetes, this could refer to the role-based access control settings for each user, as well as anonymous users.
What network access do they have? For example, which parts of the system are included within a Virtual Private Cloud (VPC)?



Source: (Rice, 2020)

Vulnerable application code

Every application written relies on a certain codebase that its developers write. This application code may have vulnerabilities introduced by the developer or code that may become vulnerable over time as exploits for certain libraries are discovered.

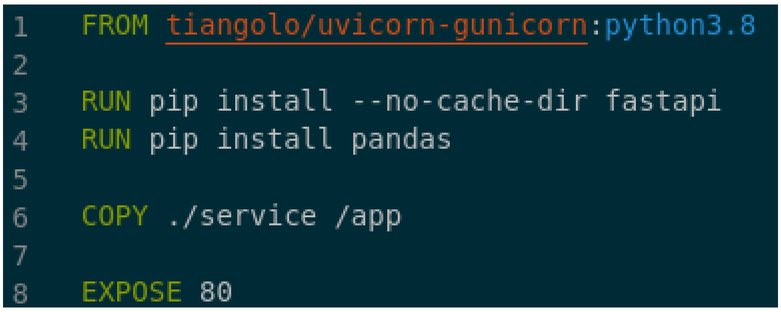
Badly configured container images

Often users believe that because they paid for their software or because millions of other users are using it that it is safely configured out of the box. Configurability is the solution to complexity however may also introduce a lot of configuration overhead. This overhead is often neglected and can be the biggest blindspot of deployments in many cases. Even if everything runs and the code is bulletproof, if the containers or the clusters are badly configured e.g. a port is open that exposes what is running inside the container then no amount of error handling will save you. When you are configuring how a container image is going to be built, there are plenty of opportunities to introduce weaknesses that can later be used to attack the running container. These include configuring the container to run as the root user, giving it more privilege on the host than it really needs.

Supply chain and Container Repository vulnerabilities

If a developer wants to run their application in a container they need to write a Dockerfile which contains instructions on how to build a docker image. Once containers are built/composed they are usually stored in a registry like Docker Hub for later use by other users or clusters that need to re-pull the image when deploying new Pods. DevOps teams need to ask themselves how they can verify that they are pulling the same image that was uploaded by them in the first place and/or how it may have been tampered with. Supply chain attacks often involve malicious container images deployed to container registries in the hope that they are then pulled by unaware users. An added bonus is that your images are not accessible from any machine should you not have access to your own. Some companies host their own container repository in-house so they are not accessible to everyone. Public container registries like Docker Hub can offer a false sense of security to users.

Sample Dockerfile



```
FROM tiangolo/uvicorn-gunicorn:python3.8

RUN pip install --no-cache-dir fastapi
RUN pip install pandas

COPY ./service /app

EXPOSE 80
```

The first line defines what container image will be the base for this container. Docker Hub has over 1,000,000 docker images to choose from. If a user would like to run a content management system like Wordpress in a docker container then it is often easier to pull from an existing library of official images rather than figure out how to take an existing application like Wordpress and correctly configure and package it. https://hub.docker.com/_/wordpress. There are docker images for pretty much anything and if a developer wants to make their software more accessible then it is good practice to maintain a docker image so people don’t have to build their own. This is where problems can be introduced.

A 2015 study of Docker Hub by Banyan Security showed that “ Over 30% of official images in docker hub contain high priority security vulnerabilities” (Desikan, 2015). This figure is arguably higher 6 years down the road in 2021 because most docker images on Docker Hub are not actively maintained and as new vulnerabilities are found these old images remain unpatched.

Another 2017 Study called “A Study of Security Vulnerabilities on Docker Hub” analyzed 356,218 images by using the framework DIVA (Docker Image Vulnerability

Analysis) and showed “both official and unofficial images have more than 180 vulnerabilities on average” with many images not having “been updated for hundreds of days”. Furthermore, with the way docker images work they are often built from parent images which are repurposed and then built upon. For instance an image may use an Alpine Linux image on which the python interpreter is installed which is then used further down the line to build a cowrie honeypot image. This means that if an upstream image has a vulnerability then all child images are also vulnerable. (Shu et al., 2017)

In 2018 security companies Fortinet (Maciejak, 2018) and MacKeeper (Lishchuk, 2021) both discovered malicious docker images on Docker Hub. These images had been up for the whole year and were tainted with ‘crytojacking’ software. They had been pulled over 5 million times.

Cryptojacking is a term for the practice of installing hidden cryptocurrency mining software on a victim's machine. Cryptojacking becomes more effective the more access to compute the mining software has. Even if each victim doesn’t have powerful mining hardware the sheer scale of exploited consumer hardware like this can quickly add up. This software secretly mines a cryptocurrency like Monero (XMR) which is sent to the attacker's wallet. A privacy coin (cryptocurrency designed to be anonymous) like Monero implements ring signatures to obfuscate the signing process of transactions to hide the identity of the attackers. The attackers managed to mine 544 XMR which is over 115,000 Euro at the time of writing.

Another interesting example of a possible supply chain attack was demonstrated in Google’s 2020 Vulnerability Reward Program Contest where contetestants are rewarded with tens and hundreds of thousands of dollars to discover vulnerabilities (Sharma, 2021). The fifth prize went to Brad Geesaman who discovered CVE-2020-15157 "ContainerDrip" (Mitre CVE, 2020). In his writeup Geesaman from Darkbit explains how a “bug could allow an attacker to trick containerd [a container runtime like docker] into leaking instance metadata by [injecting] a malicious container image manifest.” (Geesaman, 2020)

DevOps teams need also to be vigilant about what happens when attackers replace or modify an image between build and deployment.

Badly configured containers

Similar to how container images can be badly configured, the containers themselves can also be configured in such a way that makes them exploitable. Container images that may seem unproblematic and widely used can still be exploited if they are configured in a harmful way whether this is intended or not. A study by Paloalto Networks Unit42 titled “Misconfigured and Exposed: Container Services'' (Palo Alto Networks, 2019) found 40,000+ unique container hosting devices with “default container configurations that allow for quick identification”. While they state that this doesn’t necessarily mean that they are all vulnerable to exploits, it does show how users often take the default settings as secure. In their study Unit42 was able to use a wide array of tools (such as vulnerability scanners) to gain large amounts of information about these exposed services such as what they were running on which ports, what hosting providers were used, whether they were running services like Kubernetes or Docker, which images and more (Palo Alto Networks, 2019).

Vulnerable hosts

No matter how secure your container is, it is vital that the underlying host machine and operating system are secure too. If the underlying hosts are not regularly updated and patched with the latest software then the attack surface is broadened to include the host machines.

It is good practice to reduce the amount of software installed in order to reduce attack surface and maintenance needed. The 2020 datadog container report showed that the most popular Kubernetes version is 17 months old (Datadog, 2020). With new minor versions of Kubernetes every quarter it can be hard for developer operations teams to keep up with the rolling releases especially if this release cycle is different for all software on the host. This is why adopting a strong CI/CD pipeline that allows for minimal downtime is vital.

Exposed secrets

Secrets are a broad term used for sensitive information such as credentials, keys, tokens and passwords stored in containers or deployments. Both Docker and Kubernetes have a way of handling secrets but there is a whole ecosystem and branch of system architecture that explores how secrets should be stored and backed up. There are different levels depending on the needs of the system and how these secrets need to be shared across containers. We discuss secret management more in-depth in 11. Kubernetes Hardening.

Insecure networking

Container networking can be broken down into internal networking (communication between other containers or services in the cluster) and external networking (communications made with users and services outside the cluster). Both of these required stringent monitoring and configuration in order to limit the attack surface. There has been a lot of research and development into how best to handle incoming requests from outside containers as well as across containers. More on this in the container networking section.

Container escape vulnerabilities

While container runtimes like containerd and CRI-O are quite tried and tested over the years, there is still the possibility of the introduction or discovery of zero-day exploits. This is particularly relevant in our case because attackers entering into the honeypot will have free reign over that container. In this case we want to make sure that the containers are hardened against such container escape vulnerabilities. As recently as 2019 an exploit now named runc-escape was found which allows the user to escape a container running runc. Because of this possibility we have to harden against such exploits using kernel isolation tools such as gVisor and microVMs managers which we will discuss later on.

There is a never ending supply of discovered and undiscovered vulnerabilities at every step of the development and deployment process. It is the job of cybersecurity teams to manage risk while accepting and admitting that no system will ever be completely air gapped. Even if the tech is flawless there can always be social engineering attack vectors.

9.1 Container Hardening - Tier 1: (Processes, Namespaces and Cgroups)

The isolation and hardening of containers is a complex topic with multiple books and countless papers written about it. This section will attempt to distill and amalgamate many of the core ideas. Before we begin there are a few key concepts related to how the Linux Kernel works that need to be explained in order to then fully explain container hardening techniques. I call these key concepts “Tier 1 Container Isolation” tools.

Linux Processes

A quick way to think of **processes** in Linux are that they are operations that are actively running on your Linux machine. A **program** as opposed to a **process** is a collection of lines of code that is stored on the HDD or SSD of your machine. These **programs** can invoke **processes**. Every **process** has an **owner** which is identified by the **process ID (also known as PID)**.

When Linux boots, the first process that gets started is called **systemd**, historically it was known as the **init** process. When a new process is launched, 2 things happen. First there is a **fork operation** which takes the current process (a.k.a. parent process), duplicates it and assigns it a new **PID**. The second thing that happens is known as the **execute operation** whereby the new process image is replaced with the image of another process which now runs as a **child process** to the **parent process** that launched it.

One quick note on unix **user IDs (UIDs)**. Each process also has a user ID based on which user started the process. The standard user ID given to the first non root user is 1000. The next user created will have 1001. The root user has UID 0. Generally a **child process** inherits the user ID of its **parent process** that spawned it. Lets illustrate this using an example.

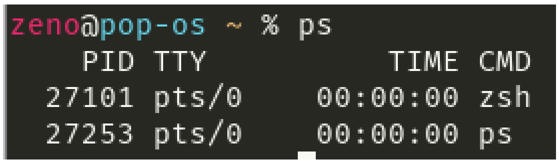
Using the Linux command **ps** we get a snapshot of the current processes we can:

- see that we are running **zsh** which is a shell alternative to bash
- In this case see **ps** which means the process also lists itself as a running process

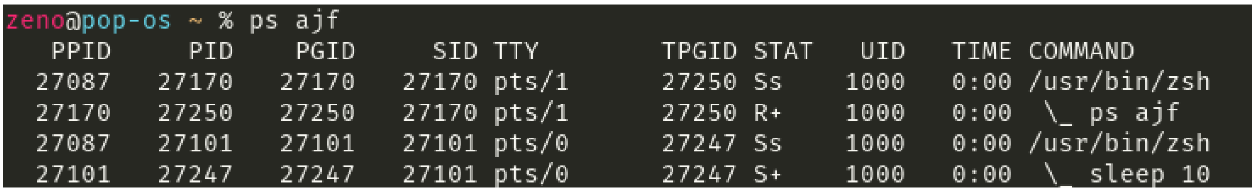
We now run a command called **sleep 10** which just creates a process that returns in 10 seconds. This is useful for example when you are writing shell scripts and need to wait for a certain amount of time. In a new terminal we type **ps ajf** to list all processes and their child processes we can see that:

- see that both **ps afj** and **sleep 10** are running as child processes which have branched off from the original **zsh** shell that created them
- they both have their own unique **PID** and share the **UID** of the user that created it, in this case user 1000.

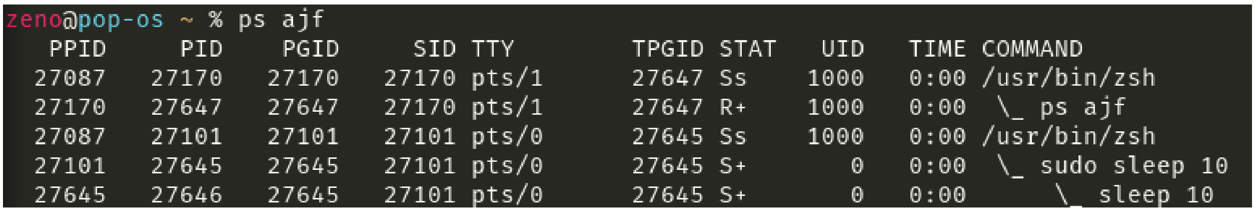
In this last screenshot, we run **sudo sleep 10** which runs the command as super-user a.k.a root. We can see that there are not multiple **child processes** which inherit the properties of the **parent process**. We can see that the **UID** of the **child process** has now changed and that the **child process** has inherited the root **UID** of 0.



```
zeno@pop-os ~ % ps
    PID TTY          TIME CMD
  27101 pts/0    00:00:00 zsh
  27253 pts/0    00:00:00 ps
```



```
zeno@pop-os ~ % ps ajf
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
  27087   27170   27170   27170 pts/1      27250 Ss    1000   0:00 /usr/bin/zsh
  27170   27250   27250   27170 pts/1      27250 R+    1000   0:00  \_ ps ajf
  27087   27101   27101   27101 pts/0      27247 Ss    1000   0:00 /usr/bin/zsh
  27101   27247   27247   27101 pts/0      27247 S+    1000   0:00  \_ sleep 10
```



```
zeno@pop-os ~ % ps ajf
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
  27087   27170   27170   27170 pts/1      27647 Ss    1000   0:00 /usr/bin/zsh
  27170   27647   27647   27170 pts/1      27647 R+    1000   0:00  \_ ps ajf
  27087   27101   27101   27101 pts/0      27645 Ss    1000   0:00 /usr/bin/zsh
  27101   27645   27645   27101 pts/0      27645 S+       0   0:00  \_ sudo sleep 10
  27645   27646   27645   27101 pts/0      27645 S+       0   0:00      \_ sleep 10
```

This high level illustration of the behaviour of PID, UID and process branching will be important moving forward.

Kernel Namespaces

Kernel **namespaces** in Linux are used to control what a process can see. Using namespaces one can restrict what virtual resources the process sees and works with. If one wants to restrict what physical resources the process can use then this can be done via Control Groups (cgroups) which are explained directly after this section on namespaces.

After namespaces were introduced to the Linux Kernel 2.4.19 in 2002, many different namespaces were defined for all kinds of purposes. Here is a list of them, as of May 2021 from the linux manpages (Linux Manpages, 2021):

- **Unix Timesharing System (UTS)**
  - Hostname and NIS domain name isolation
- **Network**
  - Network device, stack and port isolation + more
- **User and Group IDs**
  - User and group ID isolation
- **Inter-process communications (IPS)**
  - System V IPC, POSIX, message queue isolation
- **Control groups (cgroups)**
  - Cgroup root directory isolation
- **Process IDs**
- **Mount points**

A process will always occupy exactly one namespace of each of the ones listed.

The first paragraph of the Linux Manual for Namespaces is as follows:

“A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are of the namespace, but are invisible to other processes. **One use of namespaces is to implement containers**” (Linux Manpages, 2021)

Evidently there is a strong correlation between namespaces and containers which we will explore in further detail later on.

Control Groups (cgroups)

Control groups or cgroups are used to limit the physical resources that a group of processes can use. This includes RAM (memory), CPU and network input/output usage. This is very valuable from a security standpoint because it can avoid resource hogging by any one or group of processes. An example here would be if a container has been taken over by a malicious actor and is now mining cryptocurrency. If the container has no limit to its access to resources then the container will use all available CPU to generate cryptocurrency via transaction validation thereby slowing down all other processes on that machine as well as increasing the electricity bill of whoever is running the server. Cgroups can also prevent outage from a fork bomb which is an attack whereby child processes are spawned ad infinitum and recursively until the machine crashes.

Project Implementation Notes / Relevance:

- While we do not directly implement any of these including cgroups they are used by other tools that we implement.

9.2 Container Hardening - Tier 2: (Seccomp-bpf, AppArmor and SELinux)

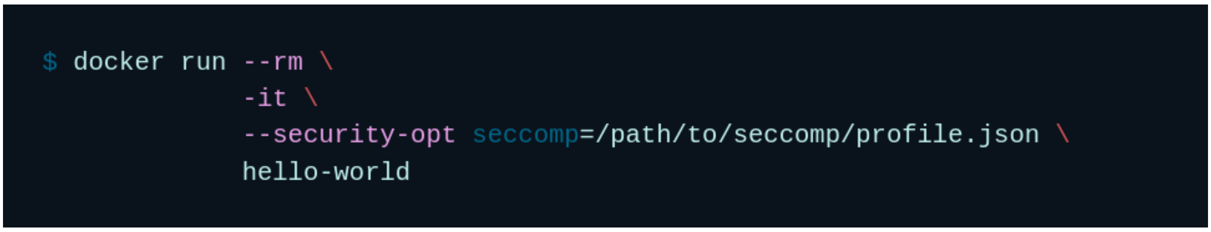
Havin Linux Namespaces, Control Groups and a few other capabilities are the building blocks that enable containers. They all further isolate their own way. In her book Container Security the author Liz Rice states that “Containers are simply Linux processes with a restricted view.” (Rice, 2020) We remember the key difference between VMs and containers is that VMs do not share kernels between Tenants while containers do. This implies a higher level of base isolation in VMs compared to containers.

Seccomp-bpf

Linux “system calls” are a way for applications to ask the kernel to perform operations for the application. Seccomp (“secure computing mode”) is a tool for setting restrictions on an application’s ability to make certain system calls. This is a very powerful concept because if a malicious application is run in seccomp mode it will not be able to execute the malicious system calls it was designed to execute. When seccomp entered the Kernel in 2005 seccomped application could only return signals (*sigreturn*), terminate (*exit*) and read and write from files that were already open before seccomp mode was enabled for that application. While this heavily restricted applications it also meant that not much could be achieved by any application in seccomp mode.

Seccomp was improved using the Berkeley Packet Filter (BPF) in 2012 and became *seccomp-bpf*. The BPF is a very fast network packet analysis and filtering tool. Combining this with seccomp allows for each process to have its own seccomp profile which bpf then uses to determine if a system call is permitted or not. BPF’s speed is achieved by “in place [filtering] (e.g. where the network interface Direct memory access (DMA) engine put it)” (McCanne and Jacobson, 1993) whereas the previous filters (e.g. SunOS's STEAMS NIT) took the approach of copying packets to the kernel buffer before filtering them.

Seccomp-bpf profiles for Docker



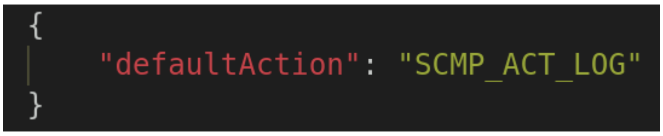
```
$ docker run --rm \
             -it \
             --security-opt seccomp=/path/to/seccomp/profile.json \
             hello-world
```

Here you can see how simple it is to apply a seccomp profile to a container in Docker. It is as simple passing a seccomp profile in JSON format using the command:

docker --security-opt seccomp=<path_to_file> flag.

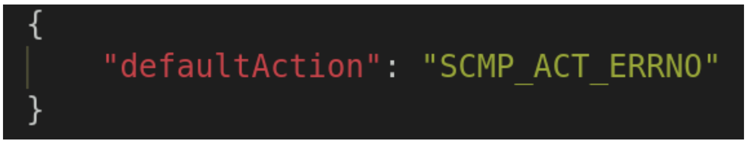
Seccomp can be configured to either only audit system calls, raise and log violations, or do very fine grained allowing and disallowing of system calls. Here JSON examples of each.

Audit / Log Only:



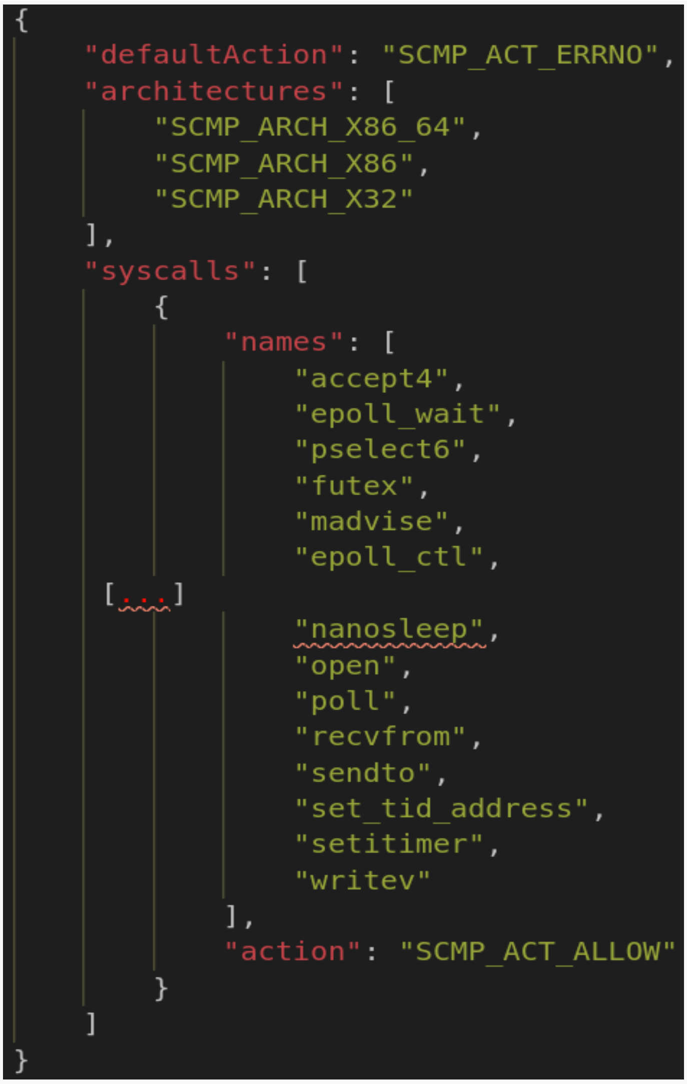
```
{
    "defaultAction": "SCMP_ACT_LOG"
}
```

Error out on any system call (no real application, just example):



```
{
    "defaultAction": "SCMP_ACT_ERRNO"
}
```

Fine grained Seccomp profile (shortened):



```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "names": [
                "accept4",
                "epoll_wait",
                "pselect6",
                "futex",
                "madvise",
                "epoll_ctl",
     [...]
                "nanosleep",
                "open",
                "poll",
                "recvfrom",
                "sendto",
                "set_tid_address",
                "setitimer",
                "writev"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

This way of defining limitations is invaluable in the container space because users of seccomp can control and audit containerised applications to make sure they don't access anything outside of their purview.

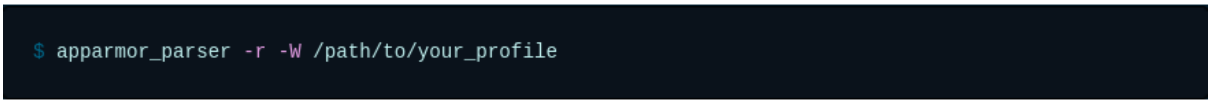
The default seccomp applied to Docker at runtime blocks over 40 of the 300+ syscalls without negatively affecting most containerised applications. It should not be disabled unless there is a good reason to do so.

Seccomp is exceptionally useful for blanket limiting certain system calls to the kernel but has also been critiqued for not being able to "dereference syscall kernel pointers [meaning] this filtering remains too coarse grained to significantly improve container's security" (McCanne and Jacobson, 1993).

AppArmor

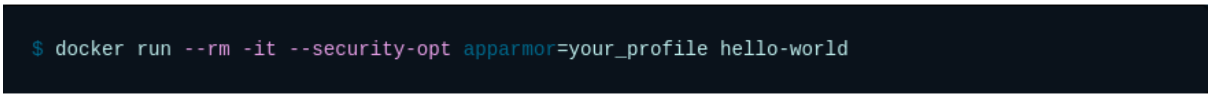
AppArmor (US spelling) is one of the simplest Linux kernel security tools available to system administrators. Linux file permissions generally work using *discretionary access controls* meaning that file owners can then impart permission to other users for that file or overwrite their own permissions. AppArmor implements *mandatory access controls* which offers a whole new level granularity for not only file access permissions but also system calls and access.

AppArmor works by implementing **application profiles**. Apparmor profiles either already come installed or can be loaded using the *apparmor_parser* command-line tool like this:



```
$ apparmor_parser -r -W /path/to/your_profile
```

You can then run a container with that custom profile applied to it by using the same flag as we did for seccomp:



```
$ docker run --rm -it --security-opt apparmor=your_profile hello-world
```

These screenshots are from the docker documentation on apparmor which can be found here along with more instructions on writing apparmor profiles, loading and unloading. https://docs.docker.com/engine/security/apparmor/

Linux does come with many default apparmor profiles.The *docker-default* apparmor profile is the default apparmor profile applied to docker containers (not docker-engine) when they are launched. This profile is loaded unless another profile is specified. In Kubernetes however, no apparmor profile is applied by default. Instead, the desired profile needs to be loaded onto every node manually using a provisioning tool using *apparmor-parser* before it can be included in the Kubernetes object YAML files.

Each application can have its own profile or group profiles can be defined when containers are launched.

SELinux

Like all the other tools described previously SELinux manages access and control of applications, processes and files. SELinux was first released on the 22nd of December 2000 (NSA, 2016). It was then merged into the Linux Kernel 2.6.0-test3 on 8th of August 2003. SELinux was a project chartered internally by the NSA. Red Hat made significant contributions to the project as did an array of other organisations. SELinux is seen as a more complex solution compared to AppArmor even though both are their own Linux Security Modules (LSM) implementations. More information on the LSM can be found in this paper titled “Linux Security Module Framework” by Chris Wright et. al. (Wright et al., 2002)

The granularity when it comes to profiles and policies with SELinux is unmatched by any LSM implementation but with this great granularity comes a much steeper learning curve as well as more room for misconfiguration. SELinux’s profile system works like a key value store with security labels being applied to every object and then rules are applied to those labels. The same goes for file paths which are first labeled and then managed. AppArmor on the other hand doesn’t do this and profiles as well as paths are directly managed.

Project implementation notes / relevance:

- As of Kubernetes 1.22 (August 5th 2021) Seccomp is enabled by default and therefore also used in this project to harden the running containers.

9.3 Container Hardening - Tier 3: (gVisor, Micro VMs, Google SLSA, Trivy)

Tier 3 is reserved for extra measures as well as ideas, tools and resources that may assist in further hardening a container. Previous tiers have directly concerned themselves with the specific hardening of the container as a process. As we saw in the previous section on container threat models, there are a few more things to think about when it comes to how containers are vulnerable. We will look at:

- **Kernel Sandboxing** via gVsior
- **MicroVMs** via AWS Firecracker
- **Image Vulnerability Scanning via** Trivy
- **Supply Chain Integrity** via Google SLSA
- **Container Image Integrity** via Sigstore

Kernel Sandboxing: Google gVisor

When it comes to kernel isolation specifically there are many options, the big three are:

- gVisor (Google, OpenSource)
- Kata Containers (OpenSource)
- Firecracker (AWS, OpenSource) -> MicroVM Solution

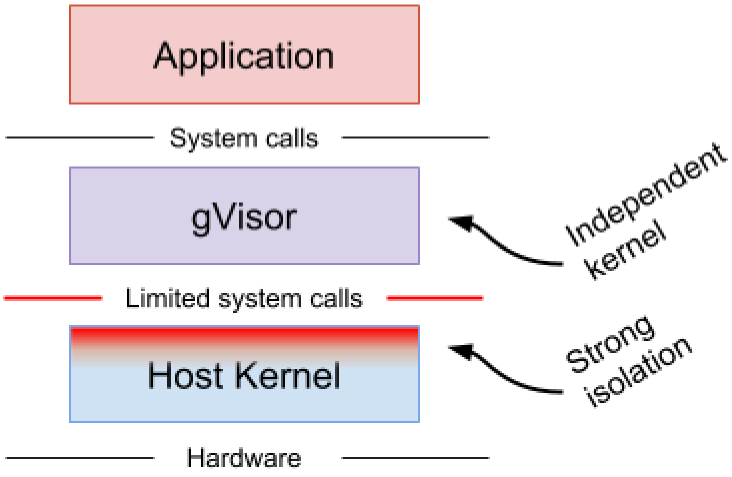
A great overview of container runtimes with a focus on these is a video by @joerg_schad of ArangoDB which can be found on YouTube. It is titled “gVisor, Kata

Containers, Firecracker, Docker: Who is Who in the Container Space?” (Schad, 2020)

Every one of the above-mentioned container runtimes / kernel isolation tools focuses on offering something close to the security of a VM while also retaining the benefits of containers, namely their speed. While all three yield similar levels of isolation, they achieve this via different means. We decided to use gVisor because our cloud provider Google has the best integration with gVisor.

Google gVisor

GVisor presents itself as a “merged guest kernel and Virtual Machine Monitor(VMM), or as seccomp on steroids” (Gvisor Team, 2021).



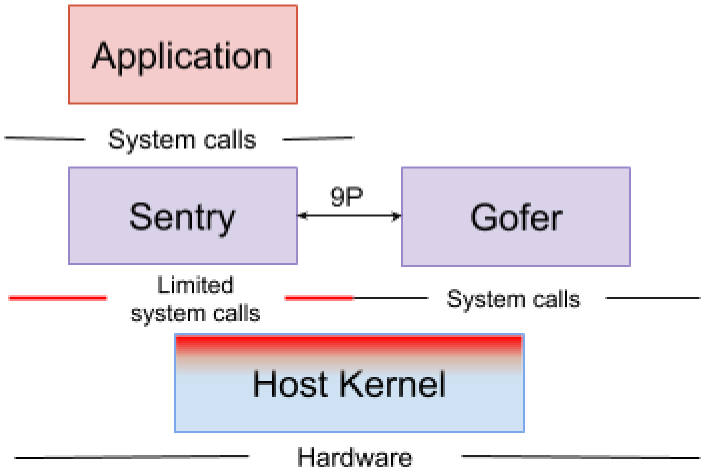
Source: (Gvisor Team, 2021)

GVisor isolates workloads by filtering “system calls, signal delivery, memory management and page faulting logic, the threading model, and more” (Gvisor Team, 2021) through a Linux Kernel written in Go. This part of gVisor is called the Sentry and runs in userspace.

GVisor is built with the principle of defence in depth in mind:

- Sentry will only make system calls to maintain itself
- Sentry never executes container system calls directly on the host kernel
- File handling is not done by Sentry, instead it is done by a gVisor component called Gofer
- The userspace linux kernel written in Go only implements universal and common functionality of the host linux kernel and does not exposed specialised functionality
- The Sentry code is carefully audited and validated and cost is broken up to control potentially unsafe code and integrity of the rules

Sentry communicates with Gofer for file handling using the 9P (**Plan 9 File System Protocol**)



Source: (Gvisor Team, 2021)

GVisor (runsc) vs. Standard runtime (runc): a Performance Comparison

Since gVisor implements an additional layer between the host kernel and the application, there are naturally going to be some performance overheads in some areas. These costs are the price one pays to harden containers in such a way.

It is important to understand that this performance hit will continually shrink as developers work on reducing the implementation costs. However there will always be some structural costs imposed due to the design choices.

These kernel hardened sandboxes are definitely not appropriate for every type of workload and often not even required because for some workloads the breaking into containers is what should be worried about.

gVisor has been tested using the benchmarking tools found here:

https://github.com/google/gvisor/tree/master/test/benchmarks

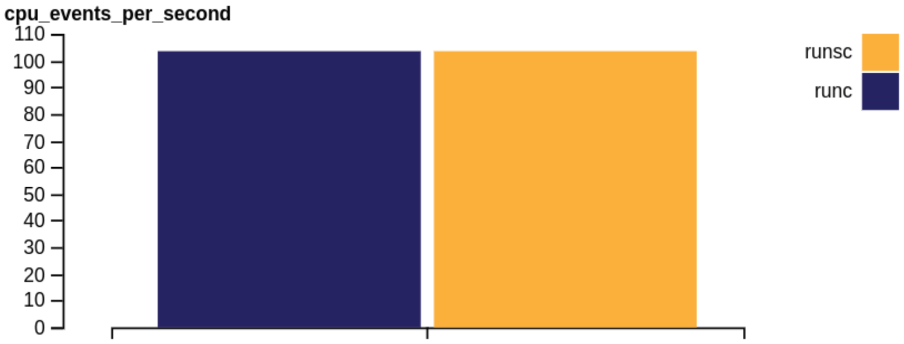
Runc and runsc (highly secure runc version used in gVisor) were tested for:

- Memory access
- Memory usage
- CPU performance
- System calls

- Start-up time
- Network
- File System

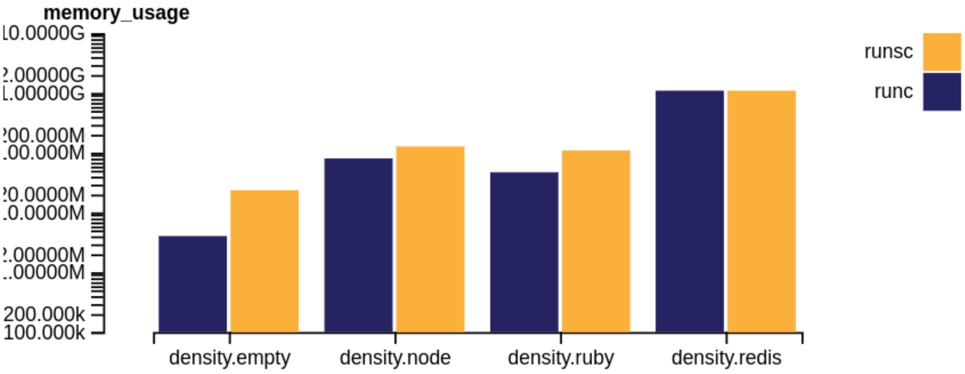
Memory performance

Memory performance overall was not heavily degraded by gVisor. As we can see, the memory bandwidth is comparable and overhead is minimal.



Source: (Gvisor Team, 2021)

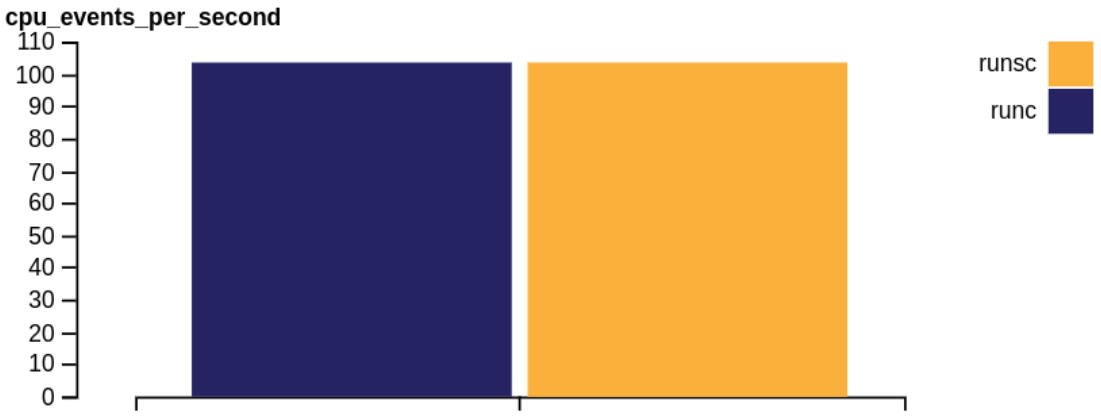
Memory usage however did always run slightly higher on runsc vs standard runc as we can see in the graph below.



Source: (Gvisor Team, 2021)

CPU performance

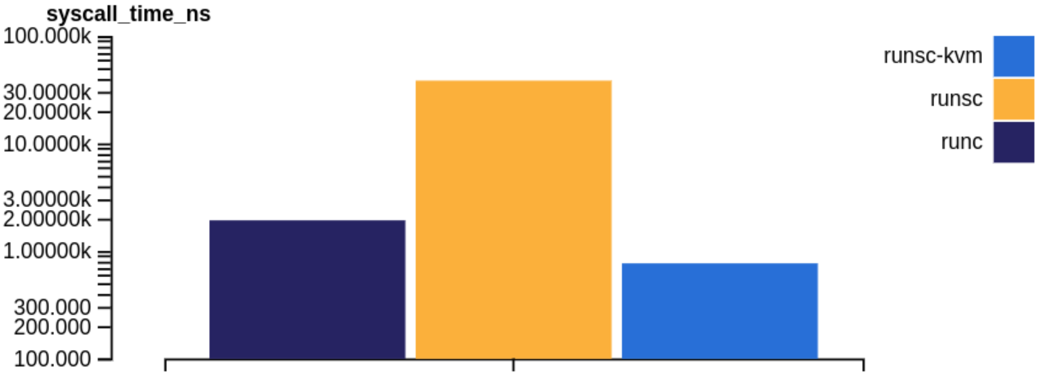
The story is the same for CPU performance. GVisor does not interfere with the real execution of CPU instructions and therefore no overhead can be found here.



Source: (Gvisor Team, 2021)

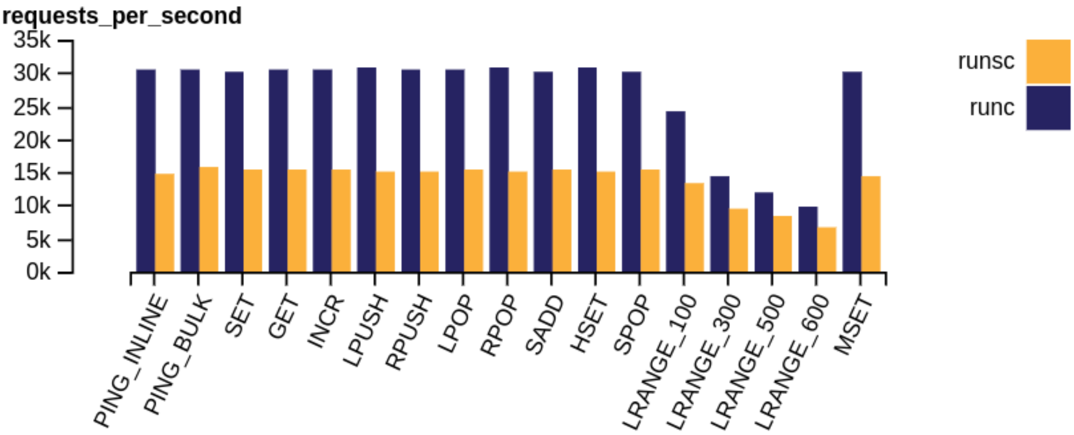
System calls

System calls are where one evidently finds the most drastic slowdowns because they are being fed through a different layer. In this graph is the runsc-kvm Sentry platform which performs well on paper but it should be noted that runsc-kvm performs very poorly in nested-virtual environments.



Source: (Gvisor Team, 2021)

When running comprehensive benchmarks on different syscall variations gVisor is shown to take a decisive performance hit when it comes to a large amount of small operations but handles large operations well due to most of the computation happening within the container.

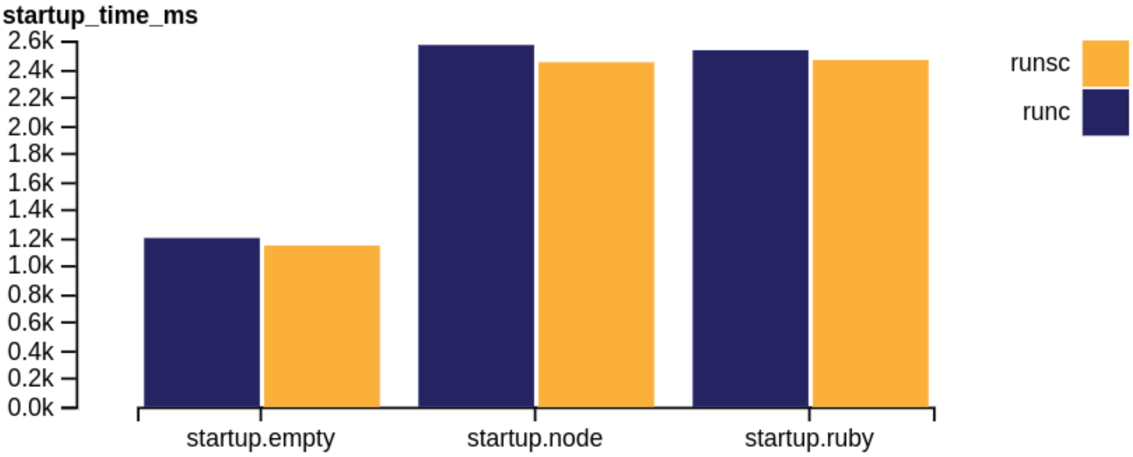


Source: (Gvisor Team, 2021)

Start-up time

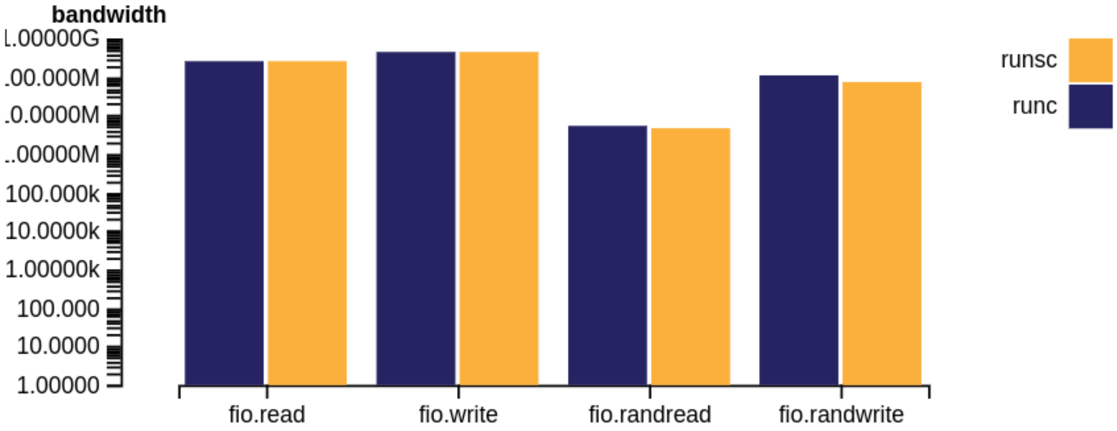
Start-up times are vital to a lot of workloads and one of the big advantages of using containers over virtual machines. The big concern with kernel isolation tools is often their slow startup time due to their bulker implementation and extra security features.

In this benchmark the start-up times of: an empty Alpine Linux container, a node application, and a ruby application were compared. Surprisingly no glaring start-up time discrepancies were found. Even the slight differences are supposedly due to Docker itself. (Gvisor Team, 2021)



Source: (Gvisor Team, 2021)

Read-Write



Source: (Gvisor Team, 2021)

MicroVMs via AWS Firecracker

AWS now offers hundreds of services. The two of particular interest here are AWS Lambda and AWS Fargate. The former being AWS' serverless a.k.a. Function as a service solution (FaaS) and the latter being something similar to GKE Autopilot. What FaaS allows users to do is run functions or code snippets in the cloud without having to configure servers or containers. This can allow someone for instance to write an entire API in the AWS console and deploy it without managing any hardware.

AWS Fargate and GKE Autopilot are similarly trying to simplify the user experience by allowing one to leverage the power of Kubernetes and containers without needing to wrangle the steep learning curve. Instead they take your container images and deploy them on your behalf. Now what does this have to do with AWS Firecracker and VMs?

In both these services, the underlying infrastructure is abstracted for the user, complexity is removed. The customer of course pays extra for this inconvenience but also receives Google's or Amazon's wealth of expertise and best practices.

In order to do this, they have to offer their customers are certain security guarantee and this is where AWS Firecracker comes in. AWS Firecracker says their mission is to "Our mission is to enable secure, multi-tenant, minimal-overhead execution of container and function workloads" (*firecracker-microvm/firecracker*, 2021)

The idea here is to merge VMs and containers in order to get the best of worlds i.e. the isolation of VMs and the agility of containers. This is required so they can offer their clients are certain peace of mind and so that even if AWS Fargate were to schedule 2 workloads, coming from different customers onto the same machine, that there is no way there would be a risk of someone breaking out of a container and move laterally into the other users workload.

AWS Firecracker uses several technologies under the hood such as KVM, Linux's tried and tested hypervisor, OpenNebula. Kata containers and many more found here https://firecracker-microvm.github.io/.

AWS Firecracker uses KVM to spin up mini-kernels for each container so that they do not share them and are therefore more secure but without having to spin up an entire operating system and all the bloat that comes with that.

Project implementation notes / relevance:

- We do not use MicroVMs in this project
- gVisor offers a similar level of isolation and is native to GKE so it is used instead

Supply Chain Attack mitigation

As previously described, when a supply chain attack occurs we can infer that there was malicious tampering of, or injection into containers before or during deployment. Attacks are mitigated by controlling the chain of command and verifying container images before they are spun up. The most rigorous, up-to-date and precise documentation about defending against supply chain attacks is contained in the unclassified "Container Hardening Guide, Version 1, Release 1, 15 October 2020" by Defense Information Systems Agency (DISA) for the United States Department of Defense (DoD).

In this document the DISA defines a "DoD Hardened Containers (DHC)" as a "Open Container Image (OCI)-compliant image that is secured and made compliant with the DoD Hardened Containers Cybersecurity Requirements". (Department Of Defence and Defence Information Systems Agency, 2020)

Common ways for insuring container image validity include:

- Maintaining a private container image repository
- Inspecting changes to a container's filesystem
- Scanning containers for known vulnerabilities
- Supply Chain Validation
- Secure CI/CD pipelines

Private Container Image Repositories (PCIR)

This DoD has its own repository for their DHCs. This private repo is called 'Iron Bank', also known as the "DoD Centralized Artifacts Repository (DCAR)" which stores DHCs and their documentation. Iron Bank can be found here and images can be downloaded from it https://repo1.dso.mil/dsop. It is read-only and only members of the Container Hardening Team (CHT) are allowed to make changes.

It is common that corporate and governmental DevOps teams tend to keep their container images on private repositories. The DoD Container Hardening Guide has a list of approved container repositories that it lists in order of priority:

- Iron Bank/DCAR (approved DoD-wide; trusted)
- Product vendor proprietary repository (untrusted)
- Docker Hub (untrusted)
- Red Hat Container Repository (untrusted)

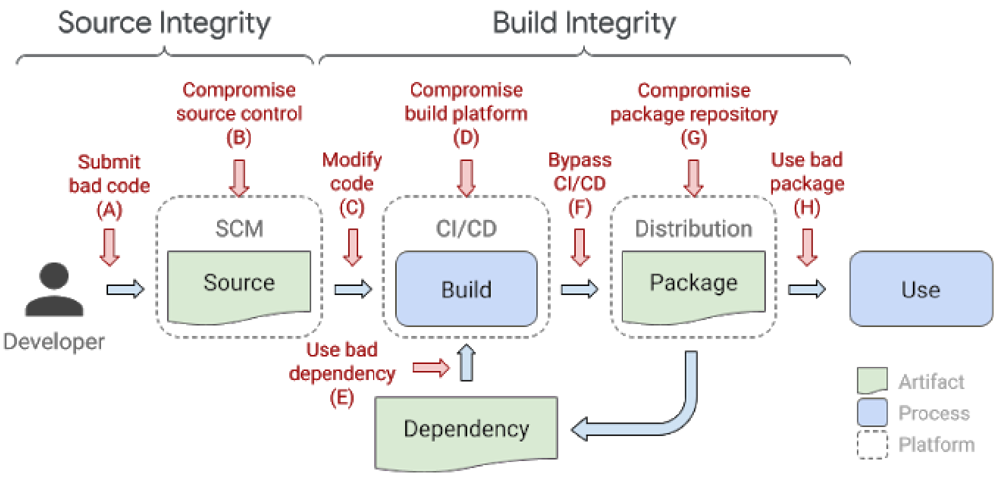
Red Hat has their own Kubernetes distribution called OpenShift which comes with its own integrated image registry solution.

Project Implementation Notes / Relevance:

- Docker Hub was used for the pulling of container images in this project

Supply Chain Integrity via Google SLSA ("Salsa")

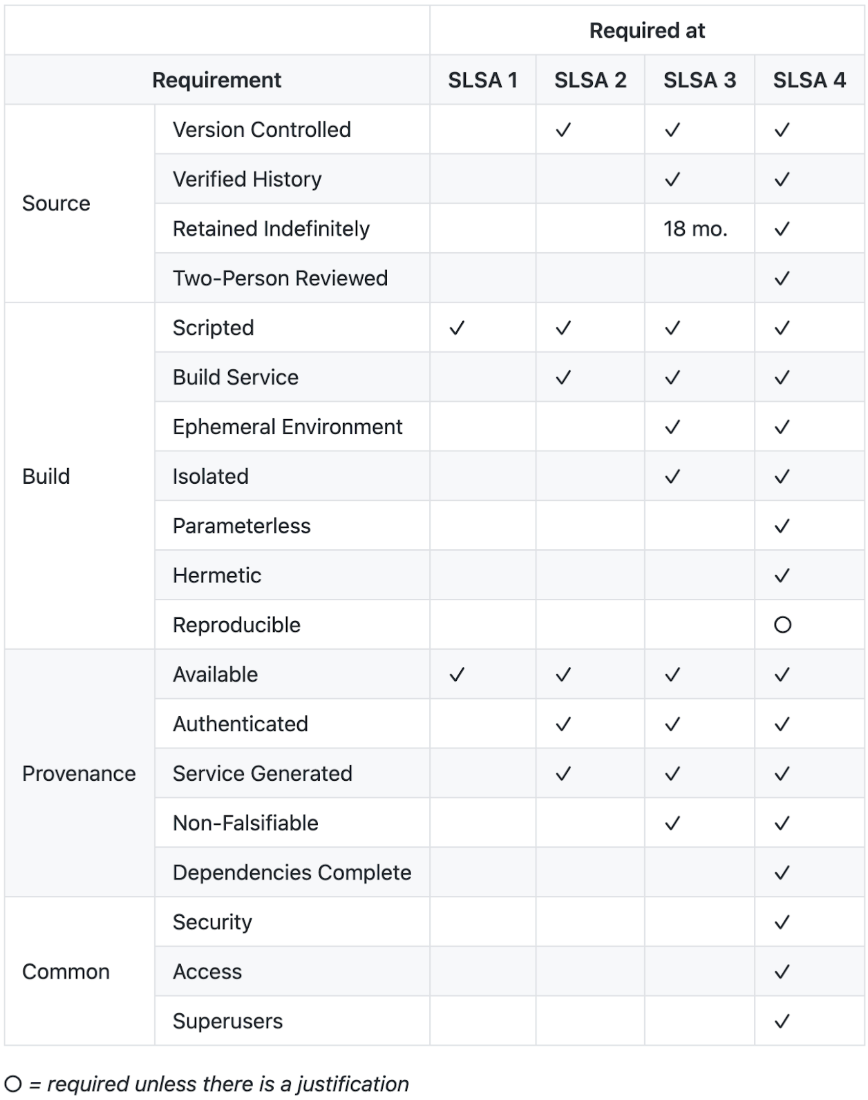
According to a 2020 report titled "State of the Software Supply Chain" (Sonatype, 2020) there is a "430% [year on year] growth in cyber attacks targeting open source software projects". With many open source projects incorporating "hundreds - if not thousands - of dependencies from other open source projects" (Sonatype, 2020). Shockingly they found that "40% of all npm packages rely on code with known vulnerabilities" (Sonatype, 2020). NPM is the Node package manager and main package manager for most javascript developers. This shows how much integrity many supply chains lack and the weak links in them that contribute to this. In the figure below is a flowchart that is meant to illustrate at how many points all these vulnerabilities can occur. All graphs



Source: (Google, 2021)

Google has their own internal framework for classifying how secure a supply chain is for their production workloads. There are 4 SLSA levels and each gets progressively more secure. All of Google's production workloads have to adhere to these standards. Directly below is a table from their blog that outlines what is contained in the four levels. As you can clearly see the table is broken up in the subsections according to where vulnerabilities can occur. This table also neatly illustrates which things are of most importance when designing around supply chain integrity and what Google priorities and what is considered an absolute minimum by them. In the next subsections we will look at Docker diff and Trivy, two tools that can help with validating integrity of images as well as notify the user of the tools to any possible malicious packages present in an image.

SLSA Integrity Levels Table



		Required at			
Requirement		SLSA 1	SLSA 2	SLSA 3	SLSA 4
Source	Version Controlled		✓	✓	✓
	Verified History			✓	✓
	Retained Indefinitely			18 mo.	✓
	Two-Person Reviewed				✓
Build	Scripted	✓	✓	✓	✓
	Build Service		✓	✓	✓
	Ephemeral Environment			✓	✓
	Isolated			✓	✓
	Parameterless				✓
	Hermetic				✓
	Reproducible				○
Provenance	Available	✓	✓	✓	✓
	Authenticated		✓	✓	✓
	Service Generated		✓	✓	✓
	Non-Falsifiable			✓	✓
	Dependencies Complete				✓
Common	Security				✓
	Access				✓
	Superusers				✓

○ = required unless there is a justification

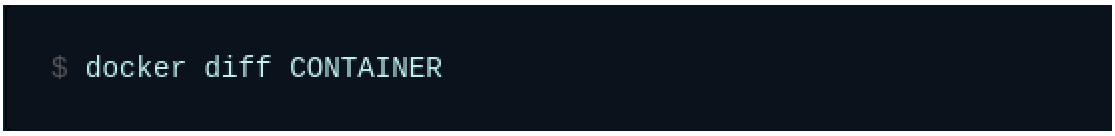
Source: (Google, 2021)

Inspect changes to a container’s filesystem

A simple straightforward way of making sure your docker container does not vary from what you expect it to be or what you initially launched it as is to check the difference between a running container and the initial container image.

Using the command *docker diff* you can “Inspect changes to files or directories on a container’s filesystem” (Docker, 2021)

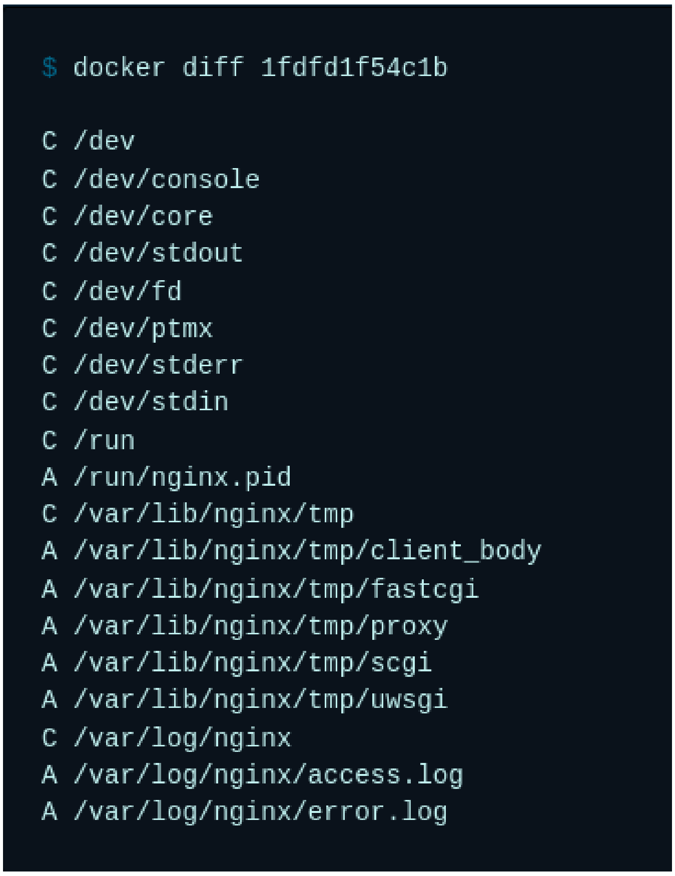
Usage Example



```
$ docker diff CONTAINER
```

Symbol	Description
A	A file or directory was added
D	A file or directory was deleted
C	A file or directory was changed

Output Example



```
$ docker diff 1fdfd1f54c1b

C /dev
C /dev/console
C /dev/core
C /dev/stdout
C /dev/fd
C /dev/ptmx
C /dev/stderr
C /dev/stdin
C /run
A /run/nginx.pid
C /var/lib/nginx/tmp
A /var/lib/nginx/tmp/client_body
A /var/lib/nginx/tmp/fastcgi
A /var/lib/nginx/tmp/proxy
A /var/lib/nginx/tmp/scgi
A /var/lib/nginx/tmp/uwsgi
C /var/log/nginx
A /var/log/nginx/access.log
A /var/log/nginx/error.log
```

Container Scanning

Container scanning is a fascinating part of the DevOps pipeline. Tools like Trivy, Anchore, Prisma/StackRox and OpenSCAP are known as scanners. As the name implies they can scan for vulnerabilities and outdated packages but can also check for misconfigurations and adherence to standards. Scanners use references like the CVE database to cross check the version numbers of installed packages against **known** vulnerabilities that were discovered for or after that release.

Vulnerability scanning shouldn’t only happen when container images are being deployed to the cluster, it is not a one time thing. The scanning process should happen regularly on deployed containers and ideally also be integrated as part of one's deployment pipeline. Vulnerability scanning can also detect malware injected into the container.

Trivy

Trivy is a container vulnerability scanner open sourced by Aqua Security (https://github.com/aquasecurity/trivy)

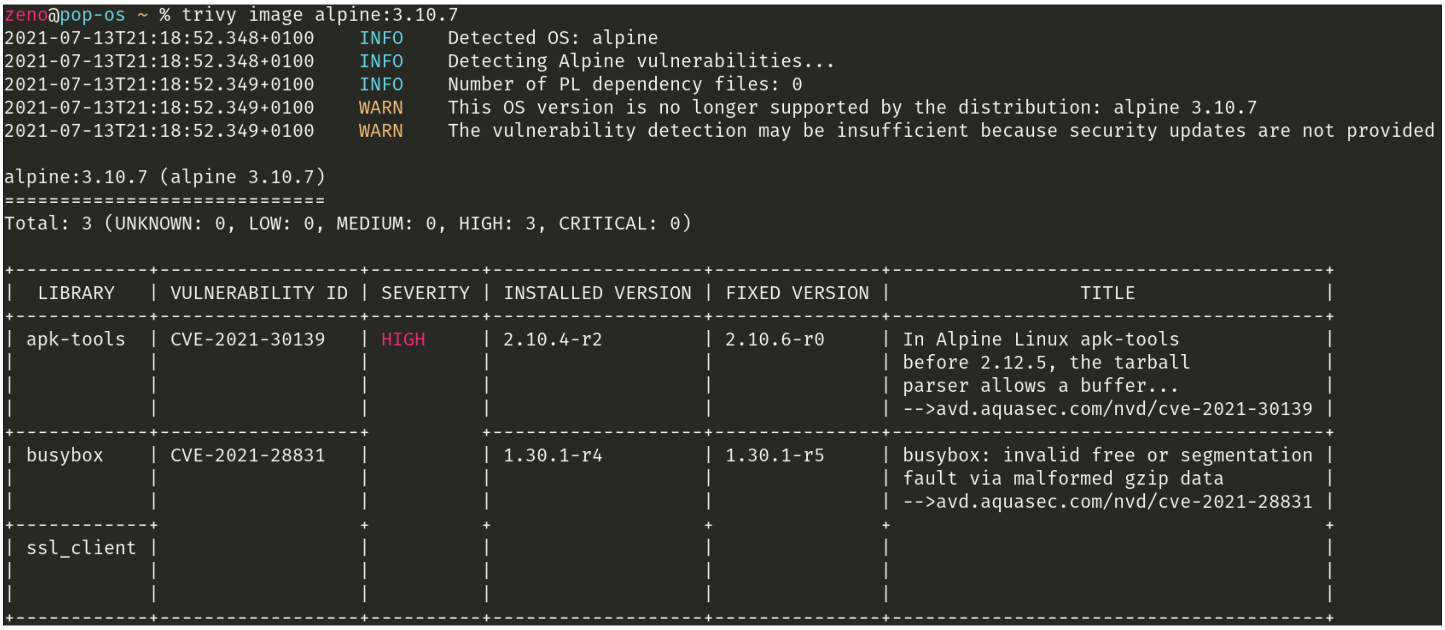
Trivy can:

- Scan a file or filesystem (dedicated machine, virtual machine and unpacked container image filesystem)
- Scan a Git Repository
- Scan a container image (+ from private docker registries)
- Be embedded in a Dockerfile so it scans as the image builds
- Scan lock files (application dependencies) for vulnerabilities (package-lock.json(Node.js), composer.lock,(PHP), Gemfile.lock(Ruby))
- Containers can scan themselves with Trivy

Running Trivy is, as the name suggests, trivial. You simply pass the name of a container on your machine or image followed by the image tag and Trivy will either begin a scan or download the image first and then begin a scan.

Below in the image you can see the completed scan of the alpine:3.10.7 image. It returned 3 critical vulnerabilities during the scan for **apk-tools**, **busybox** and **ssl_client**. To learn more about the vulnerabilities you can look up the vulnerability ID in a vulnerability database like CVE. Trivy also lists the package version in which this vulnerability is fixed (if it has been fixed) so you know what to upgrade to.

Example Trivy Scan of the alpine:3.10.7 Linx Image



```
zeno@pop-os ~ % trivy image alpine:3.10.7
2021-07-13T21:18:52.348+0100    INFO    Detected OS: alpine
2021-07-13T21:18:52.348+0100    INFO    Detecting Alpine vulnerabilities...
2021-07-13T21:18:52.349+0100    INFO    Number of PL dependency files: 0
2021-07-13T21:18:52.349+0100    WARN    This OS version is no longer supported by the distribution: alpine 3.10.7
2021-07-13T21:18:52.349+0100    WARN    The vulnerability detection may be insufficient because security updates are not provided

alpine:3.10.7 (alpine 3.10.7)
============================
Total: 3 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 3, CRITICAL: 0)

+------------+------------------+----------+-------------------+---------------+---------------------------------------+
|  LIBRARY   | VULNERABILITY ID | SEVERITY | INSTALLED VERSION | FIXED VERSION |                 TITLE                 |
+------------+------------------+----------+-------------------+---------------+---------------------------------------+
| apk-tools  | CVE-2021-30139   | HIGH     | 2.10.4-r2         | 2.10.6-r0     | In Alpine Linux apk-tools             |
|            |                  |          |                   |               | before 2.12.5, the tarball            |
|            |                  |          |                   |               | parser allows a buffer...             |
|            |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2021-30139 |
+------------+------------------+          +-------------------+---------------+---------------------------------------+
| busybox    | CVE-2021-28831   |          | 1.30.1-r4         | 1.30.1-r5     | busybox: invalid free or segmentation |
|            |                  |          |                   |               | fault via malformed gzip data         |
|            |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2021-28831 |
+------------+                  +          +                   +               +                                       +
| ssl_client |                  |          |                   |               |                                       |
|            |                  |          |                   |               |                                       |
|            |                  |          |                   |               |                                       |
+------------+------------------+----------+-------------------+---------------+---------------------------------------+
```

Project Implementation Notes / Relevance:

- All images run on Trivy were scanned for vulnerabilities before deployment
- Supply chain integrity was kept in mind with guidance from the Google SLSA framework
- GVisor is used for all honeypot containers
- Cons of gVisor were evaluated
- Images are pulled from a public repository not a private one

10. Kubernetes Threat Modelling

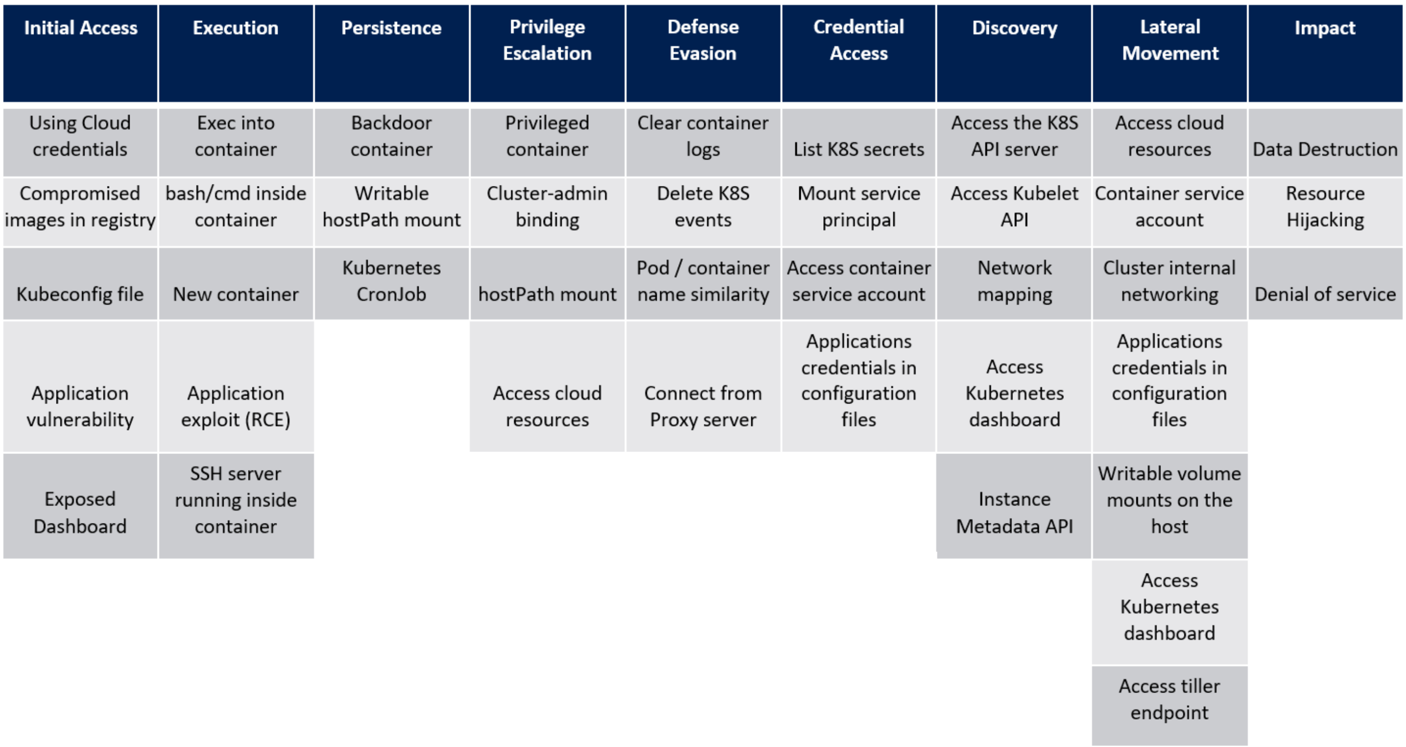
There is a natural overlap between the Kubernetes threat model and threat model for containerised environments. This section draws upon Microsoft's Kubernetes Threat Matrix (Weizman, 2020). This threat matrix in turn is heavily inspired by MITRE ATT&CK framework which key components are:

- Initial access
- Execution
- Persistence
- Privilege escalation
- Defense evasion
- Credential access
- Discovery
- Lateral movement
- Collection and exfiltration
- Command and control

The Mitre Corporation also maintains the CVE (Common Vulnerabilities and Exposures) which was described earlier.

When we think about Kubernetes and ways to model its threats we have to zoom out from the container standpoint and apply a more DevSecOps centric mindset. There is, of course, a lot of overlap from the container threat model we already discussed however Kubernetes absolutely adds more complexity to the threat model due to its configurability and extensibility.

Microsoft Kubernetes Threat Matrix



Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API server	Access cloud resources	Data Destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account	Resource Hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Access container service account	Network mapping	Cluster internal networking	Denial of service
Application vulnerability	Application exploit (RCE)		Access cloud resources	Connect from Proxy server	Applications credentials in configuration files	Access Kubernetes dashboard	Applications credentials in configuration files	
Exposed Dashboard	SSH server running inside container					Instance Metadata API	Writable volume mounts on the host	
							Access Kubernetes dashboard	
							Access tiller endpoint	

Source: (Weizman, 2020)

In order to align ourselves with contemporary thinking and the state of the art we will use the latest NSA “Kubernetes Hardening Guidance” (NSA, 2021) released August 3rd 2021. In this document the NSA cybersecurity team outlines, similarly to this document, a threat model and hardening steps with some examples.

Their threat model, combined with Microsoft’s threat models offers a strong foundation on which to build secure Kubernetes deployments.

Supply Chain Attacks and Risks

With the recent Solarwinds attack (Kisielius, 2021) we saw how attackers managed to infiltrate one of the biggest Network Management Systems (NSM) company with over 300,000 customers including the US government and inject their malware into the supply chain. Attackers were able to insert their malware into an update of Solarwinds’ Orion product and *sign* this with a correct certificate. This is a prime example of a surgically executed malware attack on a huge organisation. This update was then pushed to all customers who all had their systems compromised.

With Kubernetes one has to think beyond the application level and consider the infrastructure too because a secure cluster will inherit the vulnerable state of the insecure it is running upon and nullify many of the hardening efforts made. This means that if cyber actors manage to compromise a worker node it will likely not matter how secure those pods running on the node are if they can be directly accessed from the outside.

Kubernetes and Container APIs

Kubernetes components like the Control Plane and the worker nodes both expose a significant amount of APIs that are used to acquire information about the cluster as well as manage it. One also has to think about the workloads within the cluster and how they are networked and how a malicious actor could possibly access these

In 2018, attackers managed to compromise a Kubernetes cluster run by Tesla (RedLock CSI Team, 2018). Attackers were able to access a pod that was not password protected and which contained credentials to Tesla's AWS account. From there they were able to access sensitive telemetry data stored in an AWS S3 bucket. They also installed crypto mining software within that pod (a practice called "cryptojacking") and had the mining software listen on a "non-standard port [making it] hard to detect" (RedLock CSI Team, 2018).

This was not the first attack of such kind and according to the "State of the Software Supply Chain" (Sonatype, 2020) paper we can only anticipate more of these attacks. Companies like Tesla that run large clusters of machine learning workloads in order to train their autonomous driving models are particularly juicy targets for attackers due to the large amount of compute they have access to. These resources can then be leveraged to mine crypto at scale.

.

As seen in the Tesla hack and Microsoft's Kubernetes Threat Matrix, lateral movement into cloud service accounts or other parts of the cluster are a very real issue that need be thought about before, during and after deployment by monitoring system resources and detecting anomalies

Social Engineering and Insider Threats

A commonly touted phrase in cybersecurity is that the humans designing and using the systems are often the largest security threat. While you cannot bribe, blackmail, trick/mislead (in most cases) or manipulate a computer, you most definitely can do this with humans. Many attacks start with some employee being tricked into sharing their credentials like in the 2020 Twitter account hijacking where 130 high profile accounts were compromised. Another example of this is the Stuxnet hack on the Iranian nuclear power plant where USB sticks with the Stuxnet virus were littered around the parking lot until one employee finally picked one up and plugged it into the network.

These attacks can be the hardest to secure against because of the unpredictability of humans. A Kubernetes Admin who has access to the whole cluster may be the weakest link if he himself is compromised or becomes disgruntled. Kubernetes “two-person integrity controls” (NSA, 2021) and therefore there will always exist an account with super user privileges. Using a tool like Gatekeeper for Kubernetes, one can define very strict rules about what types of pods can be scheduled within the cluster and what images can be used. This can be useful if someone has access to scheduling privileges because unauthorised pods will be rejected by the Gatekeeper policies. One can also use things like Kubernetes’ built in RBAC (Rule based access controls) authorization to restrict access to sensitive information, more on that in section 11.

11. Kubernetes Hardening

The author of this project believes that by studying existing open-source solutions one can gain a strong understanding of where Kubernetes is headed in the future.

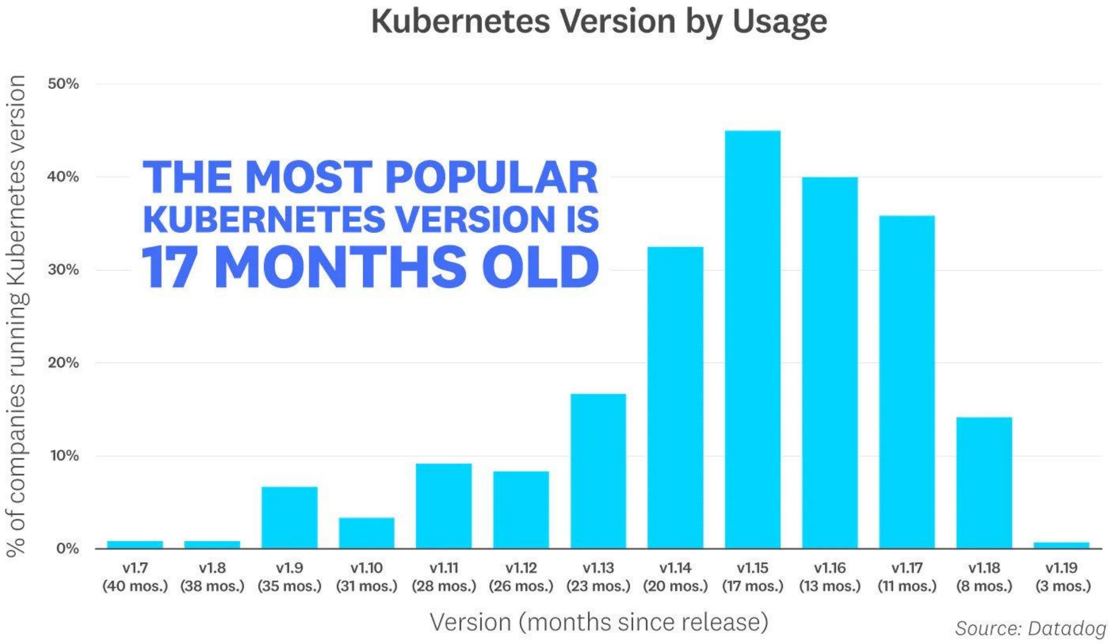
Kubernetes hardening is a discipline that incorporates all previous DevSecOps, Information Security and container hardening principles we have described. Since we are working with a GKE cluster, the main resource accessed on this topic is the Google GKE “Hardening your cluster’s security” (Google Cloud, 2021) guide which offers a wealth of detailed documentation on steps that can be taken to harden Cloud Clusters.

This section summarizes and merges includes research from the Google hardening guide, other Kubernetes hardening resources online as well as academic papers.

Stay up-to-date

One of the best things you can do to improve the security of any system is to update it. The same goes for updating Kubernetes in a timely fashion. New patches and security related updates are added all the time to keep systems safe.

The 2020 Datadog container report showed that the most popular Kubernetes version is 17 months old (Datadog, 2020) which may point to how difficult and time consuming it is to tear down your whole cluster and relaunch it with a new Kubernetes version without the guarantee that everything will work as it did previously.



Source: Datadog

Nevertheless it is vital for any serious organisation to have a process for tearing down their entire infrastructure and rebuilding it. More on this in the section on chaos engineering.

A great tool for keeping your cluster dependencies updated is WhiteSource's 'Renovate' (WhiteSource, 2021). Renovate works with Helm Charts. Renovate will monitor a Helmfile with all your installed dependencies and make pull requests to that Helmfile when new versions of packages are available.

Restricting access

Since the control planes are the main managers of a cluster throughout its lifetime, it is vital that they cannot be compromised for fear of false instructions being issued from the control plane(s) leading to the compromise of an entire cluster. Similarly important is the hardening and restricting of network access to the worker nodes since they can individually also be exploited.

When running virtual machines of any kind it is common to disable any and all password logins through SSH. Private keys of authorised systems can be added in advance or authorised if they themselves provide the SSH key of the server they are trying to log into.

This should be done with Kubernetes nodes to avoid brute force password accessing of the server. If you do want to enable password login via SSH to the nodes then make sure to install a package like fail2ban which times out SSH users who attempt to login with the wrong password for a period of time.

This sort of private key authentication is also useful because one has tight controls over who has access to the cluster which can be revoked at any time. With password this would mean resetting a password for everyone and having to redistribute it.

Isolate your workloads (gVisor)

A key discovery during the research for this project was the idea of kernel sandboxing. Most exploits and vulnerabilities already had their mitigation practices against existing attacks and thought out exploits. During the literature review however, there was a consistent mention of kernel exploits and container breakout vulnerabilities. In my mind these were the most catastrophic to a system and highly vulnerable to zero day exploits. Lateral as well as horizontal movement within a system using a single vulnerability that can't be protected against using basic container runtimes. This led to research into UniKernels, MicroVMs and modified execution paths for system calls in order to mitigate kernel exploits.`

Configure Cloud Admin Accounts

For any cluster running on the cloud, DevSecOps teams need to think about their cloud provider account as one of the main achilles heels of their infrastructure. This is where resources can be deleted, added or modified meaning a lot of damage can be done from the service accounts. It is for this reason that administrators need to carefully configure who has access to what accounts and make sure that each account has only the permissions that it needs to complete its tasks.

By creating accounts for separate teams and team members with permission that fit their exact needs but do not exceed what is necessary in terms of privileges. Labeling and tracking these members will allow for a lot of information to be gathered and quick diagnosis of who was in charge of what change.

Restrict access to cluster

RBAC, short for Role-Based Access Control, is integrated into Kubernetes and allows for access management with regards to cluster resources and namespaces. Administrators can divide up the cluster and then apply RBAC rules to different namespaces. This will allow for controlled access by the developers with regards to production clusters.

Developers would only have access to their necessary applications and resources so they can manage and deploy solely to their relevant namespace. In order to effectively perform segregation of the cluster we have to refer back to the principle of least access and model the tasks that developers will be performing on the cluster.

Kubernetes RBAC also works with Google's Identity and Access Management (IA) suite.

Restricting unauthorized access to cluster API

Managing Kubernetes networking can be quite tricky especially from a security standpoint. You still want all your internal networking to work between nodes while only allowing certain nodes to be accessed and have access to the internet. The two main ways that Google suggests for this is to “Configure Authorized networks to restrict access to set IP ranges” or “Set up a private cluster to restrict access to a virtual private cloud (VPC)”.

If neither of these options is set up then “the schema of CustomResources, APIService definitions, and discovery information hosted by extension API servers” should be treated as public. There have been occurrences of attackers gaining valuable information about a server's components using these open misconfigured APIs. This information can then be used to form and plan attacks.

The CVE-2019-11253 attack was coined a “Billion Laughs” attack. If the kube-apiserver was exposed then unauthorized users were able to send malicious payloads containing YAML or JSON and overload the server by making it use CPU cycles. This is a type of denial of service attack and potentially led to the server being unavailable or even crashing.

Versions before Kubernetes v1.14.0 had a default RBAC policy that allowed anonymous users to submit requests that could trigger this vulnerability. Anonymous utilisation of the APIs can be useful in some situations however by exploiting a bug in the way Kubernetes parses self referential and recursive YAML certain requests can make the CPU hang -> https://github.com/kubernetes/kubernetes/issues/83253

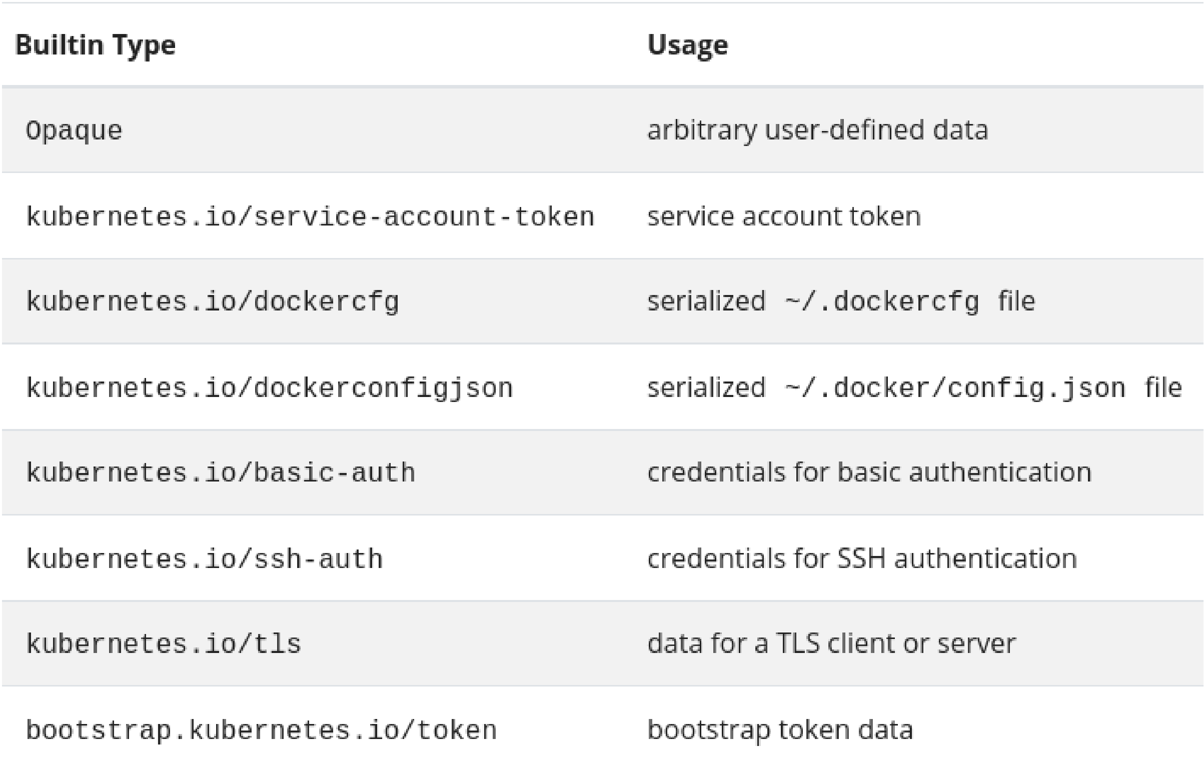
Secret Management

Secrets are a broad term used for sensitive information such as credentials, keys, tokens and passwords stored in containers or deployments. Secret management is a astronomically large topic and the way a DevSecOps team wants to handle secrets is very much left up to them in Kubernetes. Secrets in Kubernetes are their own objects and rather than being include explicitly in every Pod definition or container image they can be added later and take on many different forms such as:

- As a file in a mounted volume to one or more containers
- As an environment variable
- As a parameter for the kubelet to authenticate itself to private image repositories

Source: (Kubernetes.io, 2021)

There are an array of different types of secrets in Kubernetes as seen in this table from the Kubernetes documentation:



Builtin Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	service account token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

Source: (Kubernetes.io, 2021)

Kubernetes secrets are incredibly powerful however Kubernetes' defaults around secret storage and communication aren't safe, a common theme in Kubernetes because the intended users of it are assumed to be experienced in all aspects of running a production system.. By default secrets are stored in plaintext in etcd however etcd can be enabled to do encryption at rest. This combined with the fact that etcd is not configured by default to be using encrypted communication like TLS makes it a poor out of the box experience. There are a lot of open source and third party solutions that have been developed to manage secrets in Kubernetes because it is such a large and critical part of the infrastructure.

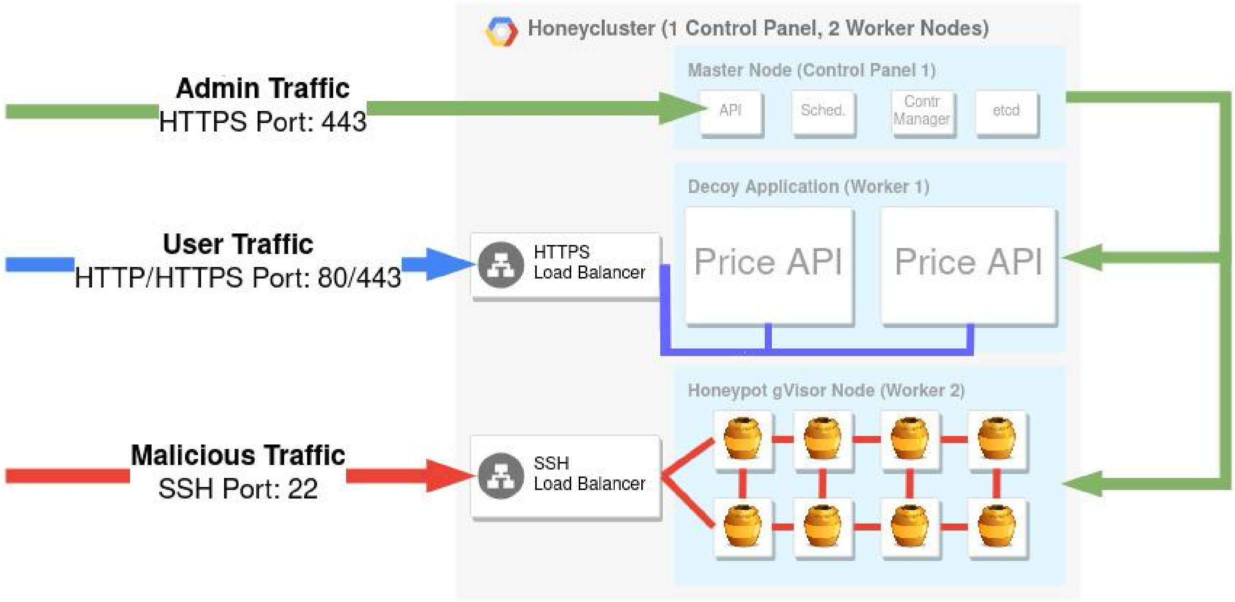
Hashicorp's Vault is one such solution for handling Kubernetes secrets. Vault can be installed via. Helm charts and when started, launches two pods. The main Vault pod is in charge of storing and managing secrets. A second pod called vault-agent-injector does the injecting of secrets into pods. Secrets (e.g. MySQL DB credentials) are created in the main Vault pod and then consequently injected into containers during or after deployment. Hashicorp's Vault documentation is fantastic and I recommend taking a look at it if you want to learn more. https://www.vaultproject.io/docs

In GKE, you can combine Kubernetes secrets with Google's built in key management systems (KMS) to enable envelope encryption. Normally GKE handles all Secret encryption which happens at rest using a *data encryption key* (DEK).

When using Google’s KMS, this DEK is then wrapped and encrypted by a *key encryption key* (KEK) at the application layer which is stored in Google’s KMS. This allows for tighter access control over the keys, easier and fewer KEK rotations and an easier time not reusing DEKs (which should be never done across users).

12. FYP Implementation: Honeyclusters

This section defines the Honeycluster framework and its building blocks. For documentation on this project, files used and a quickstart guide please view the open-source Github repository here: https://github.com/s04/Honeynetes



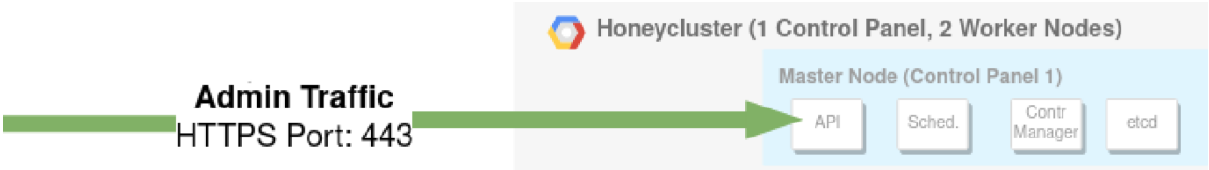
Honeycluster v1.0

Honeyclusters are a type of orchestrated container cluster composed of one part honeypot containers and one part decoy application which acts as bait. A simple Honeycluster is composed of 3 parts (1 control plane, 2 worker pods).

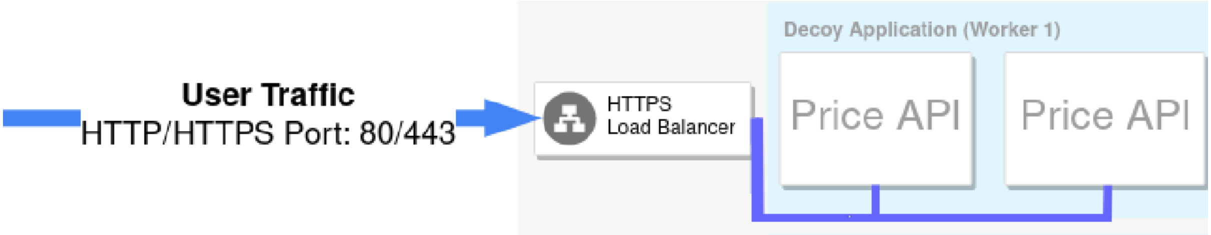
12.1 Honeycluster Components

In our 1 control plane and 2 worker node configuration the Honeycluster is not highly available due to the singular control plane. One worker node houses a cluster of honeypot containers, the other one runs the decoy application that acts as bait to attackers. We intentionally spread the two applications across two worker nodes for further isolation.

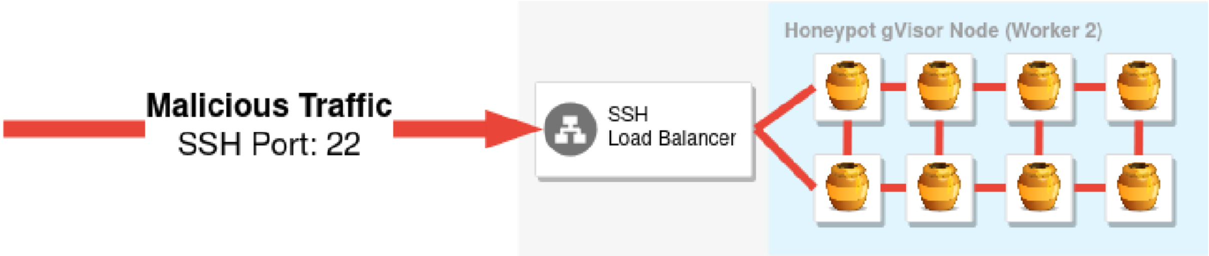
There are 3 main types of traffic that the cluster receives. Traffic labeled in green is authorised admin traffic, blue traffic is **authorised non-malicious traffic** and traffic in red is **authorised malicious traffic**.


Honeycluster (1 Control Panel, 2 Worker Nodes)
Master Node (Control Panel 1)
API
Sched.
Contr Manager
etcd
Admin Traffic
HTTPS Port: 443

Admin traffic via the Kubernetes API is used to control and manage the cluster



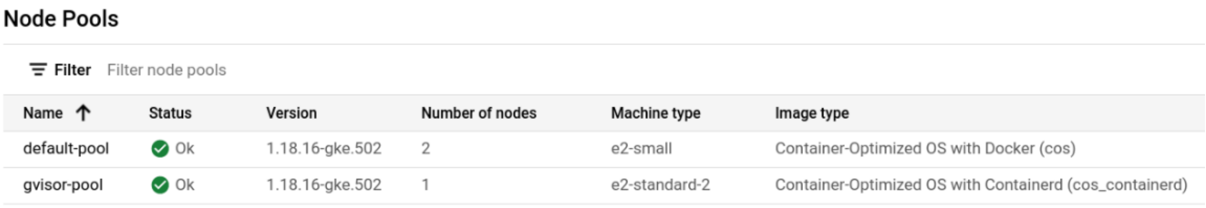
The decoy application is a pair of bitcoin price APIs (fronted by HTTPS LB)



The gVisor honeypot node runs 8 Cowrie containers (fronted by SSH LB)

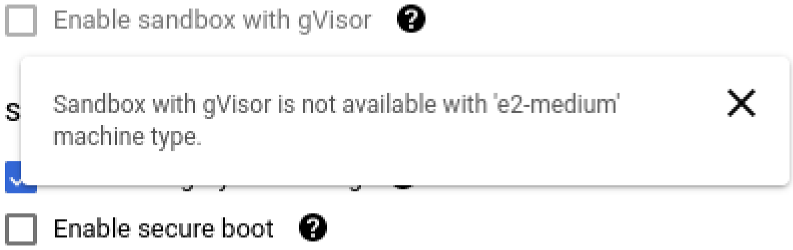
12.2 Honeycluster Hardware

Our price API node is an **e2-medium** (2 vCPU + 4GB RAM) GCP instance while the honeypots are run on an **e2-standard-2** instance (2 vCPU + 8GB RAM). This design choice is dictated by GKE. In order to use gVisor on a Kubernetes node in GKE the node needs to be an e2-standard-2 instance or greater.


Node Pools

Name ↑	Status	Version	Number of nodes	Machine type	Image type
default-pool	✓ Ok	1.18.16-gke.502	2	e2-small	Container-Optimized OS with Docker (cos)
gvisor-pool	✓ Ok	1.18.16-gke.502	1	e2-standard-2	Container-Optimized OS with Containerd (cos_containerd)

GKE Node Pool Console



☐ Enable sandbox with gVisor

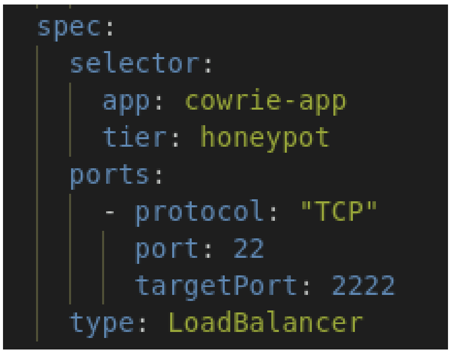
Sandbox with gVisor is not available with 'e2-medium' machine type.

☐ Enable secure boot

Minimum requirement: e2-standard-2 VPS

The decoy application is a simple cryptocurrency price API. This is used to bait attackers in and make the application look like a real target. It serves crypto prices from a csv file using the python FastApi library. As the diagrams show, traffic to this API comes via HTTP/HTTPS on port(s): 80/443. We run 2 replicas of the price API application so one can act as a failover as well as handle overflow traffic. The HTTPS load balancer routes incoming calls to the APIs based on default kubernetes scheduling rules.

Because this is a managed Kubernetes cluster, we will likely not access individual worker nodes with SSH in this setup. This means that we know that incoming SSH traffic will likely be malicious traffic trying to gain access to the nodes. Knowing this we can see that there is an SSH load balancer accepting requests on port 22 and routing these to port 2222 on one of the (8) cowrie containers.



```yaml
spec:
  selector:
    app: cowrie-app
    tier: honeypot
  ports:
    - protocol: "TCP"
      port: 22
      targetPort: 2222
  type: LoadBalancer
```

12.3 Honeycluster Design Choices (K8s Distro, Runtime, Cloud Provider)

This section evaluates the design choices made during the course of this project and acts as a justification for the final design.

Which Kubernetes distribution?

In terms of Kubernetes platforms a.k.a. Kubernetes distributions, a few different variations that can be found. All of these are based on the Kubernetes platform but differ in their use cases. The mission statement of this thesis dictated some of the tooling and using the official Kubernetes release was definitely one of these requirements. Within Kubernetes one can find multiple implementations of it. Some are more development centric (Minikube, KIND) and some are production ready (K3s and OpenShift). We will be looking at K3s and OpenShift in this section and the other two will be discussed in the “Local Development Flow” section.

K3s

K3s which can be found at https://k3s.io/ is a Kubernetes distribution developed by Rancher Labs. It is meant to be a lightweight Kubernetes platform for the Edge, ARM devices and the Internet of Things (IoT). K3s takes up less than 40MB which makes it ideal for smaller more nimble deployments. K3s is fully compliant to the Kubernetes standards and packaged as a single binary. It markets itself as having "batteries-included" features such as service load balancers, a Helm controller and the Traefik ingress controller. K3s also has the advantage of being a single binary with no distinction between control planes and worker nodes. It is quite easy to get up and running with K3s because you can easily add worker nodes to an existing control plane by passing the URL of a server which then automatically adds itself to the cluster.

K3s is able to keep its memory footprint so low by removing a lot of the components that are not vital to a default Kubernetes cluster. They also added things like a container runtime (containerd), CoreDNS (https://coredns.io/) which handles DNS and Service Discovery as well as support for Helm Charts (https://helm.sh/) out of the box which is essentially a Kubernetes Package Manager. Rancher Labs was able to make the K3s binary so small by removing bulky parts of the official Kubernetes distribution such as cloud provider plugins and storage volume plugins. Etcd, which is the distributed storage for Kubernetes, was also removed in favour of SQLite, a popular lightweight SQL implementation often found in mobile applications. SQLite is not a distributed database and therefore often touted as the weakest part of K3s but this can be mitigated by pointing the cluster state storage to an supported external database like MySQL, PostgreSQL or etcd. This theme of "batteries included but replaceable" can be found for pretty much every included component of K3s making it highly modular yet beginner friendly.

As mentioned, K3s lends itself nicely to ARM devices and is beginner friendly making it great for development and testing of clusters on devices like Raspberry Pis. Initially this was the route this project was going to take until other options which will be discussed in the 'Local Development Flow' section were favoured.

Red Hat OpenShift

OpenShift is Red Hat's implementation of Kubernetes. It markets itself as an enterprise open source container orchestration platform. They have a free starter tier for experimentation, testing or development. More advanced tiers are paid. It is powered by the community distribution of Kubernetes called 'okd' (https://www.okd.io/). There is a $263/year per cluster fee (equivalent to $0.03/hour). AWS has their own Red Hat OpenShift service on AWS for pricing examples can be found here (https://aws.amazon.com/rosa/pricing/).

OpenShift is more of a commercial product rather than a project meaning users get paid support and help with managing their infrastructure. OpenShift also comes with a more stringent approach to security with specific policies surrounding what images can and can't be run. The additional integrated Web-UI supported by OpenShift is a nice addition, something that offers a dashboard for visualising workloads, servers and projects. The standard Kubernetes distribution also has access to a dashboard which requires separate installation. Kubernetes can pull from standard registries like Docker Hub or your own Docker registry but does not come with an integrated image registry solution, OpenShift does.

OpenShift was overkill for this project and had an even steeper learning curve compared to Kuberentes. Rancher Labs, the company behind K3s also has another flavour of Kubernetes called RKE2 also known as RKE Government which "focuses on security and compliance within the U.S. Federal Government sector" https://docs.rke2.io/.

Which cloud provider?

For anyone starting their own Kubernetes project with no experience, it is rarely recommended to roll your own cluster from the ground up using Kubernetes' kubectl tool. Many things can go wrong, best practices are not in place and cloud platforms have put decades of expertise behind their respective cloud platforms to make them user friendly.

In terms of cloud platforms there are 3 big names in the space. Amazon, Google and Microsoft each have their own Kubernetes Managed Services called:

- Amazon Elastic Kubernetes Service (Amazon EKS)
- Google Kubernetes Engine (GKS)
- Azure Kubernetes Service (AKS)

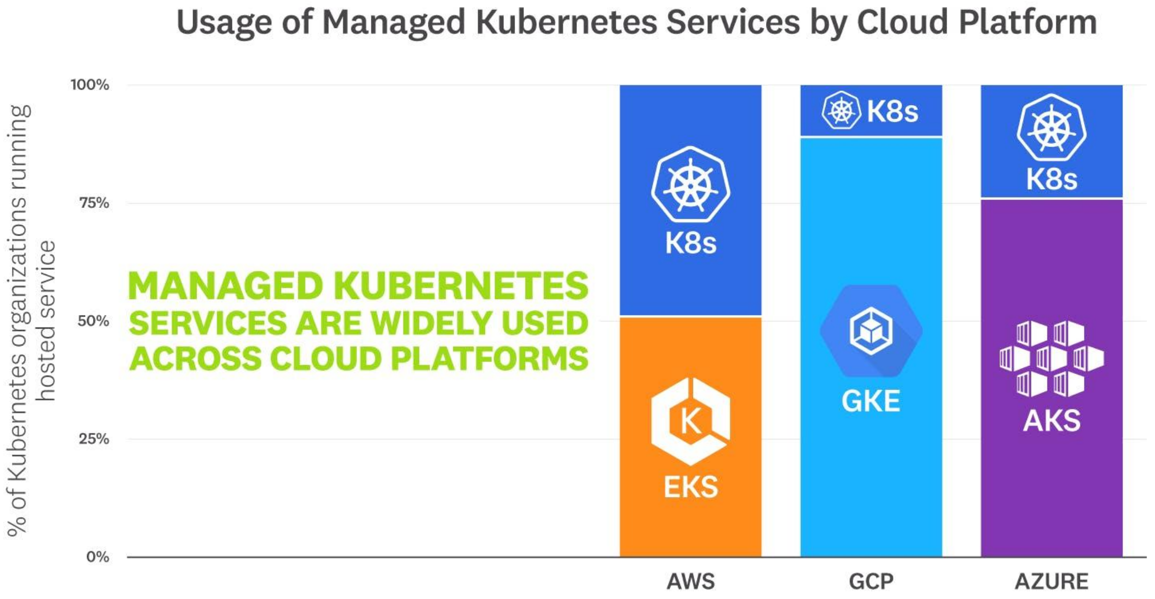
Managed Kubernetes Services are extremely useful because as the name implies, they are managed. This means that whichever service provider you choose will manage some of the arduous, time consuming, easily misconfigurable, and costly parts of the Kubernetes experience for you. While Kubernetes itself allows you to worry less about managing underlying server infrastructure, managed Kubernetes allows you to further abstract this by having large parts of Kubernetes managed for you which allows users to focus on development and deployment. Managed features include:

- Managed upgrades of cluster nodes
- Dedicated Monitoring
- Multiple Availability Zones

- Command Line Interface support
- High Availability Cluster support
- On-Premise (run managed nodes on your own hardware)
- Security

The deciding factor here was that Google's cloud platform offers $300 worth of free credits to new users as well as the fact that Kubernetes was developed by Google giving you confidence in the fact that your Kubernetes is managed with best-practices in mind.

Lastly, "GKE, AKS, and EKS dominate on their respective cloud platforms" according to the datadog container survey. They found that "~90 percent of organizations running Kubernetes on Google Cloud rely on GKE to manage their increasingly dynamic environments" (Datadog, 2020)



Source: Datadog

Which container runtime?

Container runtimes, as the name implies, are the pieces of software that enable containers to run on top of operating systems. The container runtime is in charge of running containers but also does things like managing container images and depending on the container runtime it may have additional capabilities. The paper "Performance Evaluation of Container Runtimes" by Lennart Espe et al. is used as reference for this section and explores this topic much more in depth.

Before we go more in depth, here is a brief mental map of how different container runtimes and standards are connected:

- **Docker Engine** is built on a container runtime called **containerd**, Docker Engine itself adds tooling for a better developer experience around containers. Docker is just a single tool in a world of containers and is not synonymous with containers.
- **Open Container Initiative (OCI)** “is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes” (https://opencontainers.org/)
- **Kubernetes Container Runtime Interface (CRI)** defines an API between Kubernetes and the container runtimes.
- **CRI-O** “is an implementation of the Kubernetes **CRI** (Container Runtime Interface) to enable using **OCI** (Open Container Initiative) compatible runtimes” (https://cri-o.io/)
- **GVisor (Runsc)** is a highly secure variant of the popular container runtime runC.

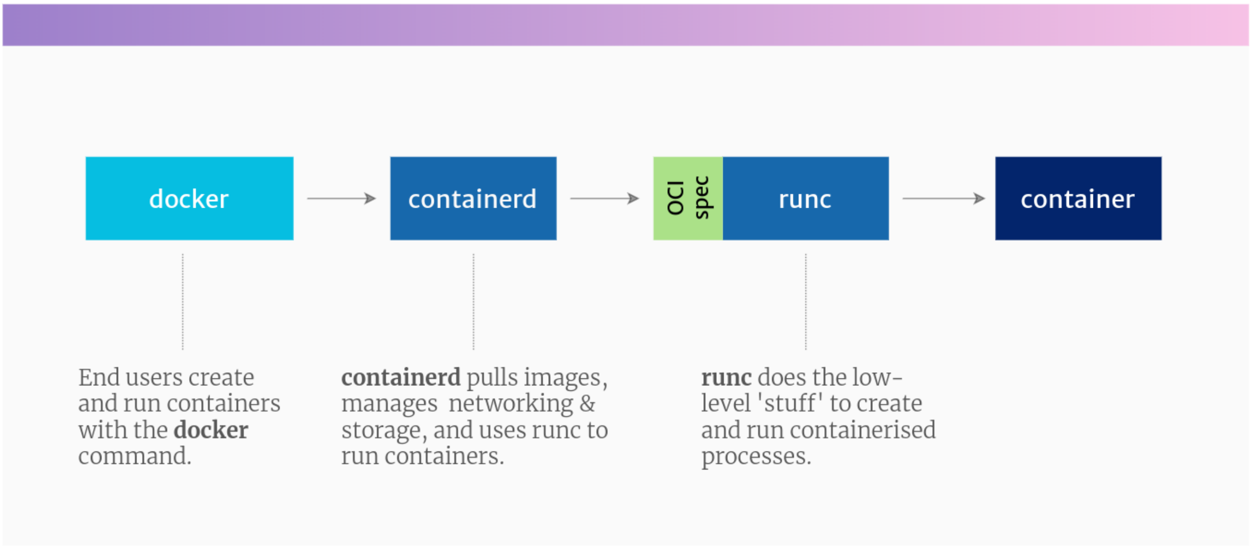
Runc

Linux OS containers is a blanket term for a large collection of isolation tools introduced to Linux such as control groups, namespaces, seccomp, apparmor which we have explored already. These were all integrated together to form runC which is a “unified low-level component” (Hykes, 2015). RunC has matured into a lightweight, portable container runtime which can be applied to manage and interact with system features surrounding containers. RunC’s features include but are not limited to:

- Support for Linux namespaces
- Native support for Selinux, Apparmor, seccomp, control groups and more.
- Native support of Windows 10 containers by Microsoft
- Planned native support for Arm, Power, Sparc with direct support from chip manufacturers.
- Planned native support for bleeding edge hardware features – DPDK, sr-iov, tpm, secure enclave, etc.
- Portable performance profiles
- A formally specified configuration format by the Open Container Project with backing from the Linux Foundation making it a real standard.

Containerd

Containerd is the industry standard container runtime due to its wide adoption. Underlying it is ‘runC’ which in turn uses libcontainer under the hood to interact with the Linux kernel. Containerd came from the Docker project and was spun out into its own project. Containerd is still used by Docker today as its underlying container runtime which means that it is installed simultaneously when Docker is installed. It can pull images from registries, start, stop and pause containers along with a list of other actions.


docker
containerd
OCI spec
runc
container
End users create and run containers with the **docker** command.
containerd pulls images, manages networking & storage, and uses runc to run containers.
runc does the low-level 'stuff' to create and run containerised processes.

Source: (Donohue, 2020)

CRI

CRI can be thought of as an API that enables Kubernetes to make use of different container runtimes whether that be containerd or CRI-O. This is great for developers because they do not need to manually implement a Kubernetes API from the ground up if they want to add compatibility with their container runtime.

CRI-O

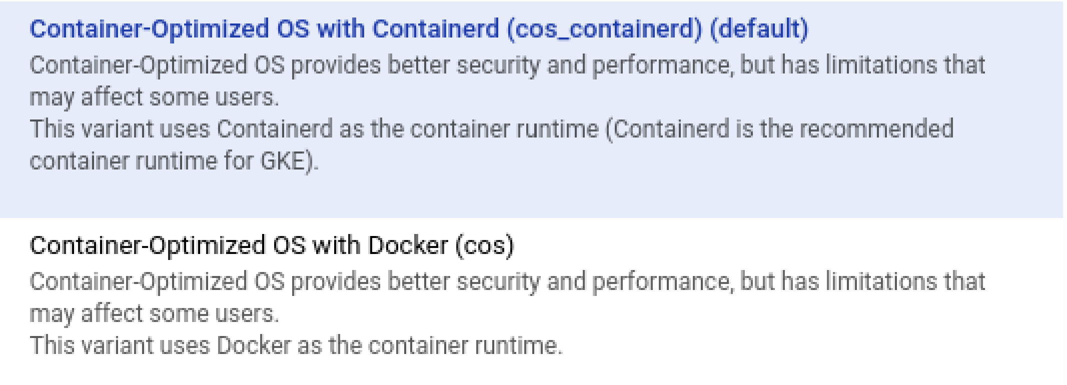
CRI-O is a container runtime specifically built from the ground up for Kubernetes. It implements the Container Runtime Interface (CRI) It provides similar functionality to containerd.

GVisor (runsc)

This project involves running untrusted workloads and therefore the bar for security and isolation is set higher than your average application. While a container runtime like runC can use tools like AppArmor, SELinux and Seccomp to limit system calls and access, gVisor takes isolation to a whole new level.

Decision (Container Runtime)

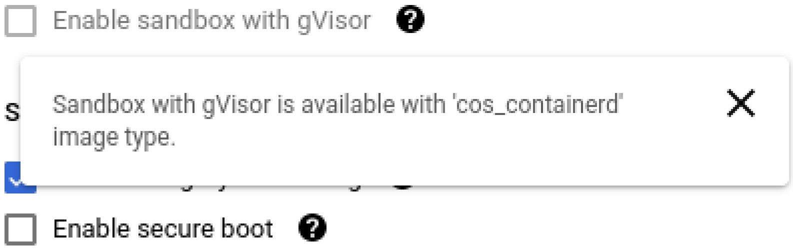
For this project, containerd was used as a runtime. Google GKE offers access to many different runtimes that are selected when defining the image type for the cluster nodes. The default option as shown below is “Container-Optimized OS with Containerd” known as cos_containerd. There are other options that include the “Container-Optimized OS with Docker” as well as “Ubuntu with Docker” and more.



Container-Optimized OS with Containerd (cos_containerd) (default)
Container-Optimized OS provides better security and performance, but has limitations that may affect some users.
This variant uses Containerd as the container runtime (Containerd is the recommended container runtime for GKE).

Container-Optimized OS with Docker (cos)
Container-Optimized OS provides better security and performance, but has limitations that may affect some users.
This variant uses Docker as the container runtime.

For this project gVisor was an essential part of the architecture. If you want to use gVisor within Google's GKE have to select the cos_containerd image otherwise the option will not be available to you.



□ Enable sandbox with gVisor

S

Sandbox with gVisor is available with 'cos_containerd' image type.

☑

□ Enable secure boot

13. Local Development Flow

When developing applications for Kubernetes and testing out cluster configurations it is often cumbersome to be constantly deploying to the cloud or a fully fledged cluster. There have been several tools developed to meet the demand for local Kubernetes development.

K3s

K3s, as has been covered, is a trimmed down but fully functional and compliant Kubernetes distribution for the Edge, IoT and lower powered devices.

Minikube

Minikube lets you run a single node cluster on your own machine. This single node cluster acts as the control plane as well as the worker node. You can schedule workloads, deploy pods and services, do rolling updates and rollbacks and control your cluster as you would a normal one using kubectl.

Minikube was used for local development in this project because it offered something that is as close as possible to a fully fledged cluster while not requiring any billing whatsoever. After the fine details were worked out then I was able to move the cluster to the cloud when my confidence was high enough and I felt like I had a good grasp of Kubernetes.

KIND (Kubernetes in Docker)

KIND is a very interesting Kubernetes tool. Like Minikube it is used for running local Kubernetes clusters. KIND, as the name suggests, uses Docker containers as nodes for the cluster. This means that instead of running Kubernetes on nodes which are virtual or physical hardware, KIND builds containers off a base image which has Docker and Kubernetes dependencies installed and schedules workloads onto these.

The Kubernetes documentation and some of the examples use Minikube so I decided to go with Minikube because I wanted to use something as close as possible to a real cluster. KIND has recently gained popularity but anecdotally, Minikube seems to still be the preferred option for local Kubernetes development.

Conclusion

The aim of this Final Year Project was initially to run low cost honeypots in Kubernetes. It began without any knowledge of containers or Kubernetes and eventually developed into this writeup on security surrounding containers and Kubernetes specifically with regards to running untrusted workloads. Much was learned during the process and this project taught me, among many other things, that Kubernetes will be a staple of DevOps and a growing part of how systems are deployed for the foreseeable future. What this also means is that I aim to personally pursue a career in DevOps and Kubernetes development in order to be part of and further the shift from legacy monoliths toward a containerised future. Hopefully this project has taught the reader something, it definitely taught me a lot. Due to how much information and complexity there is surrounding the topics covered in this essay it was hard to constrain certain sections as well as simply absorb all the knowledge before putting it down in structured writing.

Appendix

This appendix is meant to contain subsections that did not have a fitting parent section but are still relevant to the project and too valuable to scrap.

Honeytokens

Honeytokens are an exciting tool that allows you to place so-called honeytokens around your system, usually behind your firewall, that trigger an alert if they are accessed. These can be things like fake AWS credentials left somewhere around the system that no legitimate user of your organisation would use and would therefore raise an alert if used.

They come in a wide variety of appearances such as (taken from https://canarytokens.org):
Web hook / URL token: Alert when a URL is visited
DNS token: Alert when a hostname is requested
Unique email address: Alert when an email is sent to ; unique address
Customer Image Web Bug: Alert when an image you uploaded is viewed
Microsoft Word Document: Get alerted when a document is opened in Microsoft Word
PDF Document: Get alerted when a PDF document is opened in Acrobat Reader
Windows Folder: Be notified when a Windows Folder is browsed in Windows Explorer
Custom exe / binary: Fire an alert when an EXE or DLL is executed
Cloned Website: Trigger an alert when your website is cloned
SQL Server: Get alerted when MS SQL Server databases are accessed
QR Code: Generate a QR code for physical tokens
SVN: Alert when someone checks out an SVN repository

Canary Tokens (https://canarytokens.org/generate) are an accessible implementation of honeytokens which offer the list of tokens mentioned above.

Documentation can be found here: https://docs.canarytokens.org/.

Honeytokens are part of the planned improvements of this project see future improvements section but have not been implemented in V1.0.

The age of Serverless

Serverless workloads can seem a bit paradoxical at first glance, isn't the whole point of workloads that they run on servers? The word serverless is itself a bit of a misnomer. Servers will always be involved and eventually all code will be executed on a server somewhere. What is meant by serverless is that developers and DevOps teams no longer worry about the underlying infrastructure. This doesn't mean "set and forget" like in Kubernetes where teams still have to be conscious of the underlying nodes. This means they never even have to think about it because running the infrastructure has been fully abstracted into a service.

Serverless is sometimes also called functions as a service (Faas) or event driven compute. The idea is that one only has to worry about the code. Once the code has been finalised it can be deployed instantly. No worrying about dependencies, compute power, storage etc.

AWS Lambda was a revolutionary product in the serverless paradigm. With AWS Lambda you can do what was described above and build entire APIs without once leaving the online AWS interface. Serverless is extremely powerful but with this convenience comes a cost. AWS now not only manages your infrastructure but also the running of your code, libraries and everything else. For some businesses, this is not a viable option because they need more granular access to the hardware and possibly have projects that don't fit into AWS Lambda.

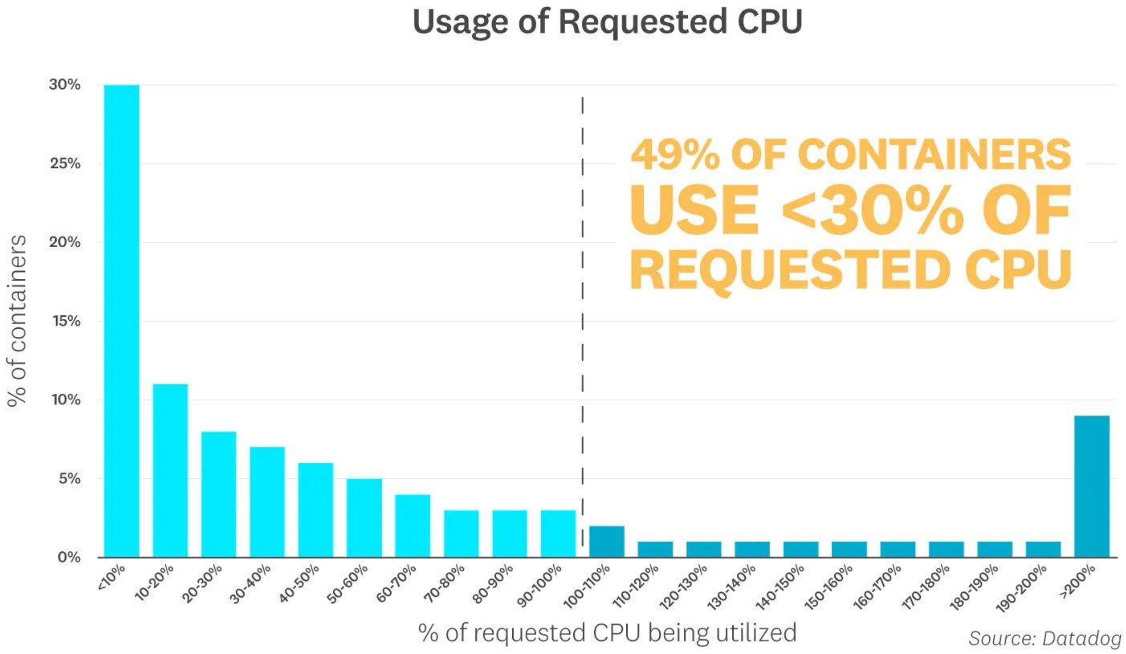
Furthermore, AWS Lambda can be more expensive if we assume that a workload is making use of 100% of a virtual instance's resources. If this workload is translated into AWS Lambda functions it likely will be more costly than running that virtual instance. As we learned however, it is extremely rare that workloads are optimised and make perfect usage of all resources. Virtual resources are often underutilised.

These are the cases where AWS Lambda shines because you get billed by the millisecond and only as long as your function calls run. Not while they are on standby. It's really great for workloads that are rarely run or called upon.

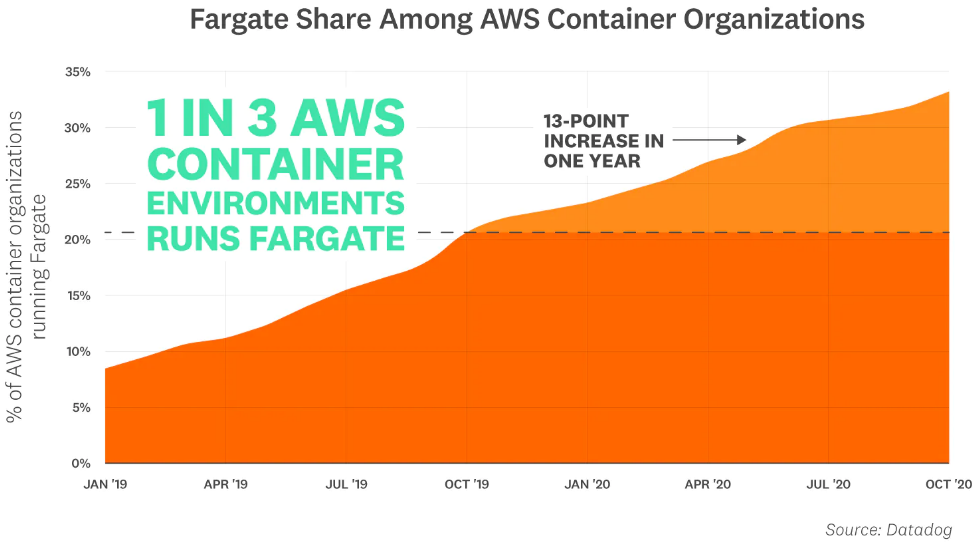
GKE Autopilot & AWS Fargate (Serverless Containers)

GKE Autopilot (Google Autopilot, 2021) is Google's attempt at taking the managed Kubernetes approach to an extreme. In this new mode of operation (compared to the standard GKE experience where one can still manage the cluster's underlying infrastructure) Google takes care of the underlying nodes and node pools completely. In GKE Autopilot mode, Google will optimize and manage clusters to reduce costs and increase workload availability. DevOps teams using GKE Autopilot simply choose container images and pay for the CPU, memory, and storage that the Pods in their cluster require while they run. Google are the ones who initially developed Kubernetes and therefore have an incredible amount of knowledge and GKE best-practices which they apply to your cluster(s) through GKE Autopilot. GKE Autopilot was not used for this project because it had not yet existed when development started.

The advent of GKE Autopilot should not be taken lightly and seriously considered for larger deployments where costs can quickly balloon. The 2020 Datadog Container Report found that “a majority of Kubernetes workloads are underutilizing CPU and memory”(Datadog, 2020). When defining pod specs, developers can state a minimum requirement of memory and CPU that this pod will require. If this is set too high then Kubernetes will not allocate workloads efficiently. This is a widespread problem because allocating too little is often more of a headache than allocating too many resources but has an adverse effect on costs. Stress testing is an invaluable approach to seeing how many resources your workloads require. Below is a graph taken from the Datadog Report and as one can see, 49% of containers use <30% of requested CPU.



AWS Fargate is AWS' answer to GKE Autopilot and serves a similar purpose. The datadog survey found that a staggering "1 in 3 AWS container environments runs Fargate" https://www.datadoghq.com/container-report/#5 proving it to be a very solid and popular option among customers. Two years ago this number hovered around 6% and now represents 32% pointing to explosive growth in the serverless container paradigm.



Knative

Knative https://knative.dev/docs/ deserves an honorable mention here. We know that most Kubernetes workloads run in the cloud and that of these workloads we know that in the AWS ecosystem, 1 in 3 of the workloads run in AWS Fargate which is AWS' serverless container offering.

Knative is run on top of a Kubernetes cluster in order for developers or users to schedule serverless workloads on that cluster. It allows organisations who don't want to run in the cloud but still leverage the serverless paradigm, to run their own serverless workloads on Kubernetes. Knative can of course be installed on a normal virtual instance in the cloud or on a managed Kubernetes service in the cloud. Knative has the added bonus of being open-source like Kubernetes.

Kubeless.io aims to serve a similar purpose of building "advanced applications with FaaS on top of Kubernetes" according to their site. https://kubeless.io/,

Fission by Platform 9 https://fission.io/, OpenWhisk by Apache https://openwhisk.apache.org/ and OpenFaas https://www.openfaas.com/ are all cornering a similar market to Knative and Kubeless.

Amazon ECS | Container Orchestration Service | Amazon Web Services [WWW Document], 2021. URL https://aws.amazon.com/ecs/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc&ecs-blogs.sort-by=item.additionalFields.createdDate&ecs-blogs.sort-order=desc (accessed 7.23.21).

Apache Mesos [WWW Document], 2021. . Apache Mesos. URL http://mesos.apache.org/ (accessed 7.23.21).

Apache OpenWhisk is a serverless, open source cloud platform [WWW Document], n.d. URL https://openwhisk.apache.org/ (accessed 7.23.21).

AppArmor security profiles for Docker [WWW Document], 2021. . Docker Documentation. URL https://docs.docker.com/engine/security/apparmor/ (accessed 7.23.21).

aquasecurity/trivy: Scanner for vulnerabilities in container images, file systems, and Git repositories, as well as for configuration issues [WWW Document], 2021. . GitHub. URL https://github.com/aquasecurity/trivy (accessed 7.23.21).

Canarytokens [WWW Document], n.d. URL https://docs.canarytokens.org/ (accessed 7.23.21).

Canarytokens.org - Quick, Free, Detection for the Masses, n.d. URL https://blog.thinkst.com/p/canarytokensorg-quick-free-detection.html (accessed 7.23.21).

CDK for Kubernetes [WWW Document], n.d. URL https://cdk8s.io (accessed 7.23.21).

Chef Documentation [WWW Document], n.d. URL https://docs.chef.io/ (accessed 7.23.21).

Chelladhurai, J., Chelliah, P.R., Kumar, S.A., 2016. Securing Docker Containers from Denial of Service (DoS) Attacks, in: 2016 IEEE International Conference on Services Computing (SCC). Presented at the 2016 IEEE International Conference on Services Computing (SCC), pp. 856–859. https://doi.org/10.1109/SCC.2016.123

Container Registry [WWW Document], 2021. . Google Cloud. URL https://cloud.google.com/container-registry (accessed 7.23.21).

Container Vulnerability Scanning for Cloud Native Applications [WWW Document], 2021. . Aqua. URL https://www.aquasec.com/products/container-vulnerability-scanning/ (accessed 7.23.21).

containerd &ndash; containerd overview [WWW Document], 2021. . containerd. URL https://containerd.io/docs/ (accessed 7.23.21).

CoreDNS: DNS and Service Discovery [WWW Document], n.d. URL https://coredns.io/ (accessed 7.23.21).

Cowrie, 2021. . Cowrie.

cowrie/cowrie - Docker Image | Docker Hub [WWW Document], 2021. URL https://hub.docker.com/r/cowrie/cowrie (accessed 7.23.21).

cowrie/docker-cowrie: Cowrie Docker GitHub repository [WWW Document], 2021. . GitHub. URL https://github.com/cowrie/docker-cowrie (accessed 7.23.21).

CVE - CVE-2014-0160 [WWW Document], 2014. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160 (accessed 7.23.21).

CVE-2019-11253: Kubernetes API Server JSON/YAML parsing vulnerable to resource exhaustion attack · Issue #83253 · kubernetes/kubernetes [WWW Document], n.d. . GitHub. URL https://github.com/kubernetes/kubernetes/issues/83253 (accessed 7.23.21).

Datadog, 2020. 11 facts about real world container use [WWW Document]. 11 facts about real world container use. URL https://www.datadoghq.com/container-report/ (accessed 7.23.21).

Datadog, 400AD. Cloud Monitoring as a Service | Datadog [WWW Document]. Cloud Monitoring as a Service. URL https://www.datadoghq.com/ (accessed 7.23.21).

Department Of Defence, Defence Information Systems Agency, 2020. (DoD) Container Hardening Process Guide.

Desikan, T., 2015. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities | Banyan Security [WWW Document]. URL https://www.banyansecurity.io/blog/over-30-of-official-images-in-docker-hub-contain-high-priority-security-vulnerabilities/ (accessed 7.23.21).

Docker, 2021. docker diff [WWW Document]. Docker Documentation. URL https://docs.docker.com/engine/reference/commandline/diff/ (accessed 7.23.21).

Docker Hub Container Image Library | App Containerization [WWW Document], 2021. URL https://hub.docker.com/ (accessed 7.23.21).

Documentation [WWW Document], n.d. . Vault by HashiCorp. URL https://www.vaultproject.io/docs (accessed 7.23.21).

Donohue, T., 2020. The differences between Docker, containerd, CRI-O and runc [WWW Document]. Tutorial Works. URL https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/ (accessed 7.23.21).

etcd [WWW Document], n.d. . etcd. URL https://etcd.io/ (accessed 7.23.21).

firecracker-microvm/firecracker, 2021. . firecracker-microvm.

Fully Managed Private and Edge Cloud Platform [WWW Document], n.d. . Platform9. URL https://platform9.com/ (accessed 7.23.21).

Geesaman, B., 2020. CVE-2020-15157 “ContainerDrip” Write-up [WWW Document]. URL https://darkbit.io/blog/cve-2020-15157-containerdrip (accessed 8.7.21).

GitHub brings supply chain security features to the Go community, 2021. . The GitHub Blog. URL https://github.blog/2021-07-22-github-supply-chain-security-features-go-community/ (accessed 7.25.21).

Google Autopilot, 2021. Autopilot overview | Kubernetes Engine Documentation [WWW Document]. Google Cloud. URL https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview (accessed 7.23.21).

Google Cloud, 2021. Hardening your cluster's security | Kubernetes Engine Documentation [WWW Document]. Google Cloud. URL https://cloud.google.com/kubernetes-engine/docs/how-to/hardening-your-cluster (accessed 7.23.21).

Gummaraju: Over 30% of official images in docker... - Google Scholar [WWW Document], n.d. URL https://scholar.google.com/scholar_lookup?title=Over%2030%25%20of%20Official%20Images%20in%20Docker%20Hub%20Contain%20High%20Priority%20Security%20Vulnerabilities&author=J.%20Gummaraju&publication_year=2015 (accessed 7.23.21).

Gvisor Team, 2021a. What is gVisor? [WWW Document]. URL https://gvisor.dev/docs/ (accessed 7.23.21).

Gvisor Team, 2021b. gvisor/test/benchmarks at master · google/gvisor [WWW Document]. GitHub. URL https://github.com/google/gvisor (accessed 7.23.21).

Hat, A., Red, n.d. Ansible is Simple IT Automation [WWW Document]. URL https://www.ansible.com (accessed 7.23.21).

Helm [WWW Document], n.d. URL https://helm.sh/ (accessed 7.23.21).

Home [WWW Document], n.d. . Salt Project. URL https://saltproject.io/ (accessed 7.23.21).

Hykes, S., 2015. Introducing runC: a lightweight universal container runtime. Docker Blog. URL https://www.docker.com/blog/runc/ (accessed 7.23.21).

IBM, 2019. IBM Study Shows Data Breach Costs on the Rise; Financial Impact Felt for Years [WWW Document]. IBM News Room. URL https://newsroom.ibm.com/2019-07-23-IBM-Study-Shows-Data-Breach-Costs-on-the-Rise-Financial-Impact-Felt-for-Years (accessed 7.23.21).

Ironbank Containers [WWW Document], n.d. . GitLab. URL https://repo1.dso.mil/dsop (accessed 7.23.21).

Jian, Z., Chen, L., 2017. A Defense Method against Docker Escape Attack, in: Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, ICCSP '17. Association for Computing Machinery, New York, NY, USA, pp. 142–146. https://doi.org/10.1145/3058060.3058085

K3s: Lightweight Kubernetes [WWW Document], n.d. URL https://k3s.io/ (accessed 7.23.21).

Kelsey, H., Joe, B., Brendan, B., 2017. Kubernetes: Up and Running [Book] [WWW Document]. URL https://www.oreilly.com/library/view/kubernetes-up-and/9781491935668/ (accessed 7.23.21).

Kisielius, J., 2021. Breaking Down the SolarWinds Supply Chain Attack. SpyCloud. URL https://spycloud.com/solarwinds-attack-breakdown/ (accessed 8.6.21).

Knative - Knative [WWW Document], n.d. URL https://knative.dev/docs/ (accessed 7.23.21).

Kubeless [WWW Document], n.d. URL https://kubeless.io/ (accessed 7.23.21).

Kubernetes, n.d. Kubernetes: zero downtime update at 10 million QPS.

Kubernetes.io, 2021a. Options for Highly Available topology [WWW Document]. Kubernetes. URL https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/ (accessed 7.23.21).

Kubernetes.io, 2021b. Kubernetes Components [WWW Document]. Kubernetes. URL https://kubernetes.io/docs/concepts/overview/components/ (accessed 7.23.21).

Kubernetes.io, 2021c. kube-apiserver [WWW Document]. Kubernetes. URL https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ (accessed 7.23.21).

Kubernetes.io, 2020. Don’t Panic: Kubernetes and Docker [WWW Document]. Kubernetes. URL https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/ (accessed 7.23.21).

kubernetes/kubernetes: Production-Grade Container Scheduling and Management [WWW Document], 2021. . GitHub. URL https://github.com/kubernetes/kubernetes (accessed 7.23.21).

Linux Manpages, 2021. namespaces(7) - Linux manual page [WWW Document]. URL https://man7.org/linux/man-pages/man7/namespaces.7.html (accessed 7.23.21).

Lishchuk, R., 2021. 6 Major Data Breaches We Found and Reported in 2018 [WWW Document]. URL https://mackeeper.com/blog/data-breach-reports-2018/ (accessed 7.23.21).

Ltd, O., n.d. OpenFaaS - Serverless Functions Made Simple [WWW Document]. OpenFaaS - Serverless Functions Made Simple. URL https://www.openfaas.com/ (accessed 7.23.21).

Maciejak, D., 2018. Yet Another Crypto Mining Botnet? [WWW Document]. Fortinet Blog. URL https://www.fortinet.com/blog/threat-research/yet-another-crypto-mining-botnet.html (accessed 7.23.21).

Mandiant Security Effectiveness Report [WWW Document], 2020. . FireEye. URL https://www.fireeye.com/current-threats/annual-threat-report/security-effectiveness-report.html (accessed 7.23.21).

Martin, A., Raponi, S., Combe, T., Di Pietro, R., 2018. Docker ecosystem – Vulnerability Analysis. Computer Communications 122, 30–43. https://doi.org/10.1016/j.comcom.2018.03.011

McCanne, S., Jacobson, V., 1993a. The BSD Packet Filter: A New Architecture for User-level Packet Capture, in: USENIX Winter 1993 Conference (USENIX Winter 1993 Conference). USENIX Association, San Diego, CA.

McCanne, S., Jacobson, V., 1993b. The BSD Packet Filter: A New Architecture for User-Level Packet Capture, in: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings, USENIX'93. USENIX Association, USA, p. 2.

Mitre CVE, 2020. CVE - CVE-2020-15157 [WWW Document]. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15157 (accessed 7.23.21).

Nomad by HashiCorp [WWW Document], 2021. . Nomad by HashiCorp. URL https://www.nomadproject.io/ (accessed 7.23.21).

NSA, 2021. NSA, CISA release Kubernetes Hardening Guidance [WWW Document]. National Security Agency Central Security Service. URL https://www.nsa.gov/News-Features/Feature-Stories/Article-View/Article/2716980/nsa-cisa-release-kubernetes-hardening-guidance/ (accessed 8.6.21).

NSA, 2016. National Security Agency Shares Security Enhancements to LINUX [WWW Document]. National Security Agency Central Security Service. URL https://www.nsa.gov/news-features/press-room/Article/1637367/national-security-agency-shares-security-enhancements-to-linux/ (accessed 7.23.21).

OKD - The Community Distribution of Kubernetes that powers Red Hat OpenShift. [WWW Document], n.d. URL https://www.okd.io/ (accessed 7.23.21).

Open Container Initiative - Open Container Initiative [WWW Document], n.d. URL https://opencontainers.org/ (accessed 7.23.21).

Palo Alto Networks, 2019. Misconfigured and Exposed: Container Services. Unit42. URL https://unit42.paloaltonetworks.com/misconfigured-and-exposed-container-services/ (accessed 7.23.21).

paralax/awesome-honeypots: an awesome list of honeypot resources [WWW Document], 2021. . GitHub. URL https://github.com/paralax/awesome-honeypots (accessed 7.23.21).

Production-Grade Container Orchestration [WWW Document], 2021. . Kubernetes. URL https://kubernetes.io/ (accessed 7.30.21).

Red Hat OpenShift Service on AWS Pricing | Amazon Web Services [WWW Document], n.d. . Amazon Web Services, Inc. URL https://aws.amazon.com/rosa/pricing/ (accessed 7.23.21).

RedLock CSI Team, P.A., 2018. Lessons from the Cryptojacking Attack at Tesla [WWW Document]. RedLock. URL https://redlock.io/blog/cryptojacking-tesla (accessed 8.6.21).

Regions and zones | Compute Engine Documentation [WWW Document], 2021. . Google Cloud. URL https://cloud.google.com/compute/docs/regions-zones (accessed 7.23.21).

Research, T.A., n.d. Know. Before it matters [WWW Document]. Canarytokens. URL https://canarytokens.org (accessed 7.23.21a).

Research, T.A., n.d. Know. Before it matters [WWW Document]. Canarytokens. URL https://canarytokens.org (accessed 7.23.21b).

Rice, L., 2020. Container security: fundamental technology concepts that protect containerized applications, First edition. ed. O'Reilly, Beijing Boston Farnham Sebastopol Tokyo.

s04/Honeynetes: Kubernetes honeypots and intrusion detection system using gVisor [WWW Document], n.d. . GitHub. URL https://github.com/s04/Honeynetes (accessed 7.23.21).

Schad, S., 2020. gVisor, Kata Containers, Firecracker, Docker: Who is Who in the Container Space?

Secrets [WWW Document], n.d. . Kubernetes. URL https://kubernetes.io/docs/concepts/configuration/secret/ (accessed 7.23.21).

Serverless Functions for Kubernetes [WWW Document], n.d. . Fission. URL https://fission.io/ (accessed 7.23.21).

Sharma, H., 2021. Announcing the winners of the 2020 GCP VRP Prize. Google Online Security Blog. URL https://security.googleblog.com/2021/03/announcing-winners-of-2020-gcp-vrp-prize.html (accessed 7.23.21).

Shu, R., Gu, X., Enck, W., 2017. A Study of Security Vulnerabilities on Docker Hub, in: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY '17. Association for Computing Machinery, New York, NY, USA, pp. 269–280. https://doi.org/10.1145/3029806.3029832

Sonatype, 2020. 2020 State of the Software Supply Chain - The 6th Annual Report on Global Open Source Software Development.

team, T.K. io website,. KubeVirt.io [WWW Document]. KubeVirt.io. URL https://kubevirt.io// (accessed 7.23.21).

Terraform by HashiCorp [WWW Document], n.d. . Terraform by HashiCorp. URL https://www.terraform.io/ (accessed 7.23.21).

Tornow, D., 2018. The Kubernetes Scheduler [WWW Document]. Medium. URL https://dominik-tornow.medium.com/the-kubernetes-scheduler-cd429abac02f (accessed 7.23.21).

Vulnerability scanning for Docker local images [WWW Document], 2021. . Docker Documentation. URL https://docs.docker.com/engine/scan/ (accessed 7.23.21).

Webteam, P., n.d. Powerful infrastructure automation and delivery | Puppet [WWW Document]. URL https://puppet.com/ (accessed 7.23.21).

Weizman, Y., 2020. Threat matrix for Kubernetes. Microsoft Security Blog. URL http://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/ (accessed 7.23.21).

WhiteSource, 2021. Whitesource Renovate: Automated Dependency Updates. WhiteSource. URL https://www.whitesourcesoftware.com/free-developer-tools/renovate/ (accessed 7.23.21).

Wordpress - Official Image | Docker Hub [WWW Document], n.d. URL https://hub.docker.com/_/wordpress (accessed 7.23.21).

Wright, C., Cowan, C., Morris, J., Smalley, S., Kroah-Hartman, G., 2002. Linux Security Module Framework.