



Departement IT en Digitale Innovatie

Container orkestratie beveiliging: identificeren van veiligheidsproblemen en onderzoek naar de impact van beveiligings-tools

Nick Heymans

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Wim De Bruyn
Co-promotor:
Steven Trescinski

Instelling: —

Academiejaar: 2020-2021

Tweede examenperiode

Departement IT en Digitale Innovatie

Container orkestratie beveiliging: identificeren van veiligheidsproblemen en onderzoek naar de impact van beveiligings-tools

Nick Heymans

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Wim De Bruyn
Co-promotor:
Steven Trescinski

Instelling: —

Academiejaar: 2020-2021

Tweede examenperiode

Woord vooraf

Deze bachelorproef luidt het einde van mijn driejarige opleiding Toegepaste Informatica aan de HoGent in. Het onderwerp voor deze bachelorproef werd gekozen vanuit persoonlijke interesses. Hierin probeer ik aan te tonen hoe de beveiliging van een Kubernetes cluster werkt en waarom deze zo belangrijk is. Door de COVID-pandemie zijn de laatste drie semesters van de opleiding op een zeer speciale manier verlopen. Deze pandemie heeft niet enkel effect gehad op de manier van les volgen maar heeft tevens een grote impact gehad op de manier waarop er naar *cybersecurity* gekeken wordt.

Ik apprecieer alle lectoren en docenten die mij in de afgelopen drie jaar enorm geholpen en gesteund hebben. Mijn ouders zou ik willen bedanken om mij de kans te geven om verder te studeren en mij verder te steunen tijdens mijn opleiding. Bij het schrijven van deze bachelorproef heb ik van verschillende personen heel wat hulp gekregen. Als eerste wil ik mijn promotor Wim De Bruyn en mij co-promotor Steven Trescinski bedanken. Vervolgens wil ik mijn vriendin Lisa Jacquemijn bedanken voor de steun en motivatie tijdens het schrijven van deze bachelorproef. Verder wil ik ook Robin Ophalvens bedanken voor het geven van zijn zeer uitgebreide feedback.

Samenvatting

In deze bachelorproef wordt de beveiliging van container applicaties en container orkestratie geanalyseerd. Hierbij zal specifiek gekeken worden naar de grootste veiligheidsrisico's, welke effecten deze kunnen hebben op een productieomgeving en hoe deze vermeden kunnen worden. Tevens zal onderzocht worden welke beveiligings-tools (zoals *Project Calico* en *Kube-hunter*) er bestaan en hoe deze ingezet kunnen worden bij het beveiligen van een container cluster. De laatste jaren is het gebruik van containers in de *Information technology* (IT) infrastructuur fors gestegen en dit zal zo blijven evolueren in de komende jaren. Aan alle technologieën zijn nu eenmaal veiligheidsrisico's verbonden, container orkestratie vormt hier geen uitzondering op de regel. In voorgaand onderzoek werd er reeds gefocust op de grootste veiligheidsrisico's desondanks is er zeer weinig ingegaan op de effecten bij het toepassen van *best practices*. Met deze paper tracht ik het gebruik en de daaraan verbonden risico's van container virtualisatie en container orkestratie te onderzoeken. Daarnaast zal er ook gekeken worden naar hoe de verschillende beveiligings-tools kunnen helpen bij het beveiligen van containers. Ten slotte wordt er onderzocht welk effecten deze hebben op de relevante criteria. Dit laatste zal via een *proof-of-concept* opstelling gebeuren.

Inhoudsopgave

1	Inleiding	17
1.1	Probleemstelling	18
1.2	Onderzoeksvraag	18
1.3	Onderzoeksdoelstelling	18
1.4	Opzet van deze bachelorproef	18
2	Stand van zaken	21
2.1	Containers	21
2.1.1	Container vs. virtuele machine	22
2.1.2	Waarvoor worden containers gebruikt	23
2.2	Docker	23
2.2.1	Hoe werkt Docker	24
2.2.2	Docker componenten en terminologie	24

2.3	Container orkestratie	25
2.3.1	Container orkestratie tools	26
2.4	Kubernetes	27
2.5	Security	30
2.5.1	Meest voorkomende security problemen	30
2.5.2	Hoe een container cluster beveiligen	30
2.6	beveiligings-tools	32
2.6.1	Project Calico	32
2.6.2	Kube-Bench	33
2.6.3	Kube-hunter	34
3	Methodologie	35
3.1	Opzetten van de Kubernetes cluster	36
3.1.1	Linode Kubernetes cluster	36
3.1.2	Resetten van cluster	38
3.1.3	Gegevens- verzameling en verwerking	38
3.2	Scenario 1: Cluster opstelling zonder oog voor security	44
3.3	Scenario 2: Cluster opstelling met gebruik van best practices	49
3.4	Scenario 3: Cluster opstelling met beveiligings-tools	52
3.5	Data analyse	56
3.5.1	Scenario 1	57
3.5.2	Scenario 2	59
3.5.3	Scenario 3	61
3.5.4	Vergelijken van de data	64

4	Conclusie	65
A	Onderzoeksvoorstel	67
A.1	Inleiding en State-of-the-art	67
A.1.1	Wat zijn containers?	67
A.1.2	Waarom container applicaties?	68
A.1.3	Context voor dit onderzoek	68
A.1.4	Verloop van het onderzoek	68
A.2	Methodologie	68
A.3	Verwachte resultaten	69
A.4	Verwachte conclusies	69
A.5	Bijlagen	69
	Bibliografie	71

Lijst van figuren

2.1	Type 1 & Type 2 Hypervisor (VMWare, 2021)	22
2.2	Container vs. virtuele machine (Google, 2016)	23
2.3	Docker componenten (Docker, 2021a)	24
2.4	Kubernetes cluster componenten (Kubernetes, 2021)	28
2.5	Platformen die ondersteunt worden door Calico (Tigera, 2021)	33
3.1	Aanmaken Linode cluster	37
3.2	Toevoegen van nodes aan cluster	38
3.3	kubeconfig.yaml	39
3.4	Klein deel van het gebruikte csv bestand	40
3.5	R code om boxplot van CPU gegevens te bekomen	41
3.6	R code om boxplot van CPU gegevens te bekomen	41
3.7	R code om boxplot van CPU gegevens te bekomen	42
3.8	R code om grafiek van CPU gegevens te bekomen	42
3.9	deployment.yaml	44
3.10	Pods worden klaargemaakt	45
3.11	Output "kubectl edit deployment" commando	46
3.12	Pods worden klaargemaakt	47

3.13	service.yaml	47
3.14	Informatie over de loadbalancer service.	47
3.15	De demo website is nu zichtbaar vanop het internet.	48
3.16	securePol.yaml	50
3.17	role-deployment.yaml	51
3.18	rolebinding-deployment.yaml	51
3.19	Output kube-bench commando	53
3.20	Output kube-hunter remote scanning	54
3.21	Output kube-hunter remote scanning	55
3.22	Output kube-hunter remote scanning	55
3.23	Gemiddeld CPU gebruik (100 - Laatste Kolom).	57
3.24	Gemiddeld RAM gebruik	57
3.25	Boxplot van het CPU gebruik in Scenario 1	57
3.26	Boxplot van het RAM gebruik in Scenario 1	58
3.27	Lijngrafiek van het CPU gebruik in Scenario 1.	58
3.28	Lijngrafiek van het RAM gebruik in Scenario 1.	58
3.29	Opstarttijd van de Node in Scenario 1	58
3.30	Gemiddeld CPU gebruik (100 - Laatste Kolom).	59
3.31	Gemiddeld RAM gebruik in Scenario 2	59
3.32	Boxplot van het CPU gebruik in Scenario 2	59
3.33	Boxplot van het RAM gebruik in Scenario 2	60
3.34	Lijngrafiek van het CPU gebruik in Scenario 2.	60
3.35	Lijngrafiek van het RAM gebruik in Scenario 2.	60
3.36	Opstarttijd van de Node in Scenario 2	60
3.37	Gemiddeld CPU gebruik (100 - Laatste Kolom).	61
3.38	Gemiddeld RAM gebruik	61
3.39	Boxplot van het CPU gebruik in Scenario 3	62
3.40	Boxplot van het RAM gebruik in Scenario 3	62
3.41	Lijngrafiek van het CPU gebruik in Scenario 3.	62
3.42	Lijngrafiek van het RAM gebruik in Scenario 3.	62
3.43	Opstarttijd van de Node in Scenario 3	63

LIJST VAN FIGUREN	13
A.1 Verwachte opstart tijd	70
A.2 Verwacht resource gebruik	70

Lijst van tabellen

3.1	Gemiddeld resource gebruik van de scenario's	64
3.2	Aantal outliers van de scenario's	64
3.3	Tijd die de Node nodig had om op te starten	64

1. Inleiding

Tijdens de ontwikkeling van traditionele applicaties wordt de applicatie uitgewerkt in een specifieke testomgeving. Bij het overzetten van de applicatie van de test- naar de productieomgeving (i.e., van een Linux testomgeving naar een Windows productieomgeving), komen er vaak problemen naar boven). Deze problemen kunnen vermeden worden door gebruik te maken van containers en vergemakkelijken daarbij het uitrollen en schalen.

Een container is een pakket waar één enkele applicatie in zit, samen met alle nodige afhankelijkheden (Education, 2019). Dit zorgt ervoor dat deze gemakkelijk en snel van de ene omgeving naar de andere kan overgezet worden.

Naarmate het gebruik van containers steeg, steeg ook de nood om deze vanuit één centrale locatie te beheren. Om aan deze vraag te voldoen werden container orkestratie tools, zoals Kubernetes¹, ontwikkeld. Deze tools helpen bij het opzetten, uitbreiden en verbinden van een grote hoeveelheid containers.

In deze bachelorproef worden de moderne veiligheidsrisico's geanalyseerd die gepaard gaan met container virtualisatie en container orkestratie. Hierbij zal specifiek gekeken worden naar de grootste veiligheidsrisico's, welke effecten deze kunnen hebben op een productieomgeving en hoe deze vermeden kunnen worden.

Aan alle technologieën zijn nu eenmaal veiligheidsrisico's verbonden, container virtualisatie en container orkestratie vormen hier geen uitzondering op de regel. Een groot onderdeel van container beveiliging zijn de zogenaamde beveiligings-tools (zoals *Project Calico* en *Kube-hunter*).

¹<https://kubernetes.io/>

In voorgaand onderzoek, zoals (Islam Shamim e.a., 2020), werd er reeds gefocust op de grootste veiligheidsrisico's. Echter is er zeer weinig ingegaan op de effecten bij het toepassen van *best practices* en het gebruik van beveiligings-tools. Met deze paper tracht ik het gebruik, en de daaraan verbonden risico's, van container virtualisatie en container orkestratie te onderzoeken. Daarnaast zal er ook gekeken worden naar hoe de verschillende beveiligings-tools kunnen helpen bij het beveiligen van containers.

Ten slotte wordt er onderzocht hoe deze risico's vermeden of opgelost kunnen worden en welk effecten ze hebben op de relevante criteria. Dit laatste zal via een *proof-of-concept* opstelling gebeuren.

1.1 Probleemstelling

De probleemstelling houdt in dat veel bedrijven gebruik maken van containers en container orkestratie zonder hierbij al te veel aandacht te besteden aan de beveiliging hiervan. Daarnaast dient er gekeken te worden naar wat voor effecten de beveiliging van een container omgeving met zich mee brengt en hoe men best te werk gaat.

1.2 Onderzoeksvraag

Wat zijn de belangrijkste beveiligingsrisico's? Welke beveiligings-tools zijn er en hoe werken ze? Welke *best practices* kunnen toegepast worden? Welke impact hebben *best practices* en beveiligings-tools op verschillende criteria?

1.3 Onderzoeksdoelstelling

Het doel van deze bachelorproef is hoofdzakelijk om tot een verslag te komen met daarin aanbevelingen omtrent het beveiligen van een container cluster. Deze aanbevelingen zullen gestaafd worden door enkele scenarios en hun effect op enkele criteria.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

De stand van zaken of *State of the art* geeft een algemeen beeld weer van de technologieën die worden overwogen in dit onderzoek en tevens de verschillende manieren van toepassing.

2.1 Containers

Containers bieden veel voordelen vergeleken met virtuele machines (VM's). Ze kunnen snel opgezet worden en zijn gemakkelijk om te configureren terwijl virtuele machines vaak groot en traag zijn. Containers zijn pakketten waarin een applicatie vervat zit, samen met zijn benodigde afhankelijkheden. Hierdoor kunnen ze vlot van de ene omgeving naar de andere worden overgezet zonder dat er extra configuratie nodig is (Education, 2019). Containers maken gebruik van besturingssysteem-virtualisatie om processen te isoleren van het *host* besturingssysteem. Daarnaast controleert het tevens de CPU gebruik en hoeveelheid RAM geheugen van deze processen (Docker, 2018).

Containers hebben eigenlijk geen eigen besturingssysteem nodig, alle containers delen 1 gezamenlijke *runtime engine*. Een *runtime engine* is de laag die verantwoordelijk is voor de communicatie tussen het besturingssysteem van de host machine en de containers zelf. De meeste gebruikte runtime engine is de *Docker Engine*¹.

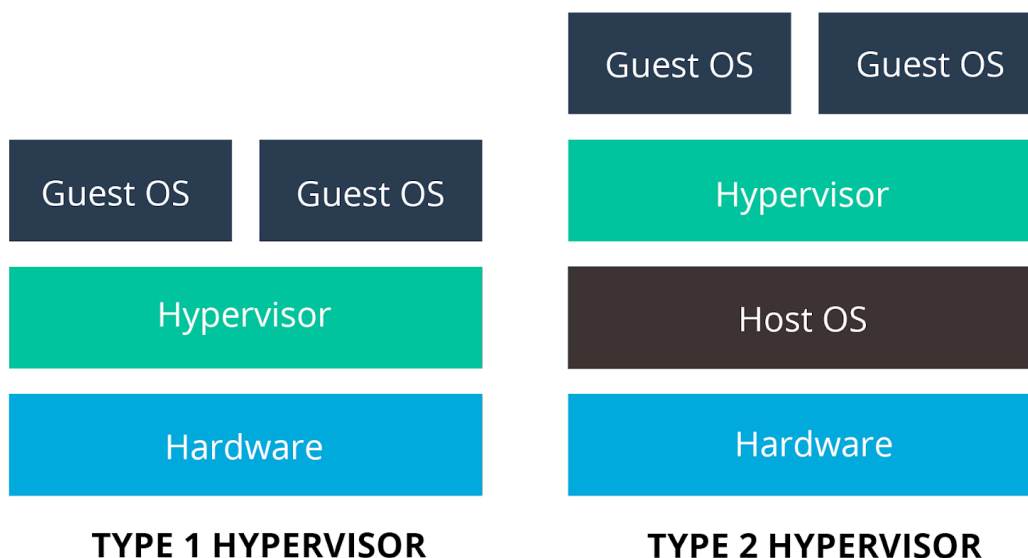
¹docs.docker.com/engine/

2.1.1 Container vs. virtuele machine

Hypervisors

Bij traditionele virtuele machines virtualiseert een *hypervisor* de fysieke hardware. De hypervisor regelt het resource gebruik tussen de verschillende VM's en zorgt ervoor dat de hardware van de host (de fysieke hardware waar de hypervisor op geïnstalleerd is) eerlijk verdeelt wordt. Er zijn 2 types hypervisor (VMWare, 2021), namelijk:

- Type 1: Ook wel *bare metal* hypervisors genoemd. Deze werken rechtstreeks op de fysieke hardware van de host en hebben dus geen onderliggend besturingssysteem nodig. Door het rechtstreekse contact tussen de hypervisor en de hardware wordt een type 1 hypervisor beschouwd als de best presterende en meest efficiënte hypervisor. Een type 1 hypervisor wordt ook beschouwd als het veiligste van de 2, dit omdat de gebreken en kwetsbaarheden die doorgaans in besturingssystemen aanwezig zijn hier onmogelijk zijn.
- Type 2: Ook wel *hosted* hypervisors genoemd. Deze worden doorgaans geïnstalleerd op een bestaand besturingssysteem en steunt daar ook op voor het beheren van de resources. Een groot voordeel van type 2 hypervisors is dat ze een breed gamma aan hardware ondersteunen.

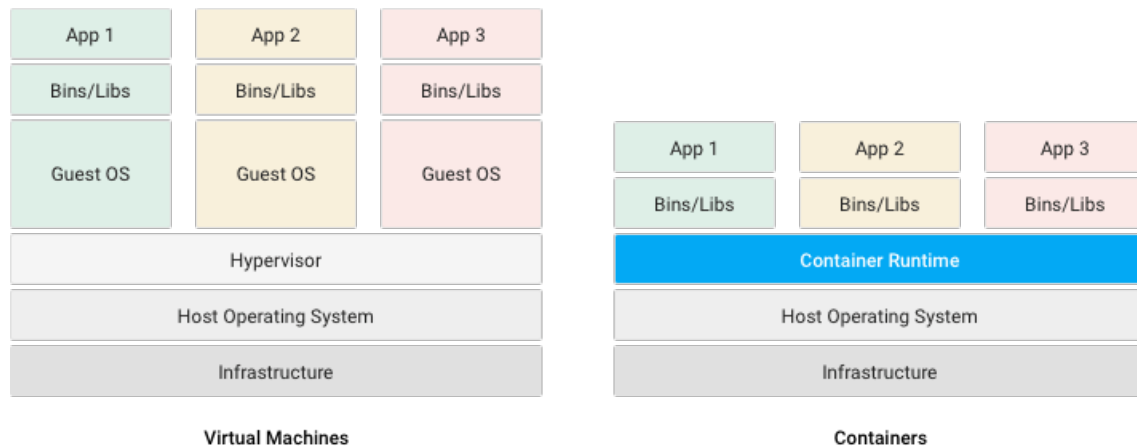


Figuur 2.1: Type 1 & Type 2 Hypervisor (VMWare, 2021)

Tegenwoordig worden type 1 hypervisors in productieomgevingen het meest gebruikt, dit vanwege hun efficiënte *resource* gebruik en veiligheid. Type 2 hypervisors worden meer gebruikt in testomgevingen omdat deze gemakkelijker op te zetten zijn.

Elke VM heeft zijn eigen volwaardig besturingssysteem, met gevolg dat deze veel resources gebruiken en hierbij, in vergelijking met containers, vaak trager zijn. In plaats

van de onderliggende hardware te visualiseren gebruiken containers de kernel van het besturingssysteem (meestal is dit Linux) zelf, zo bevat elke individuele container enkel de applicatie en de bijhorende afhankelijkheden bevat (Education, 2020).



Figuur 2.2: Container vs. virtuele machine (Google, 2016)

2.1.2 Waarvoor worden containers gebruikt

Containers zijn zeer veelzijdig en kunnen dus in veel verschillende omstandigheden gebruikt worden. Enkele *use cases* waarvoor containers zeer geschikt zijn (Docker, 2021b):

- **Microservices:** Containers zijn klein en licht, waardoor ze goed passen bij microservice-architecturen waarin applicaties zijn opgebouwd uit vele, losjes gekoppelde en onafhankelijk services.
- **Modernisering en migratie van applicaties:** Een van de meest voorkomende benaderingen voor het moderniseren van applicaties begint met het containeriseren ervan, zodat ze naar de *cloud* kunnen worden gemigreerd.
- **Nieuwe ontwikkelaars snel inwerken:** Door gebruik te maken van containers verloopt het opzetten van een nieuwe lokale ontwikkelingsomgeving snel en vlot, hierdoor kunnen de ontwikkelaars direct aan de slag.

2.2 Docker

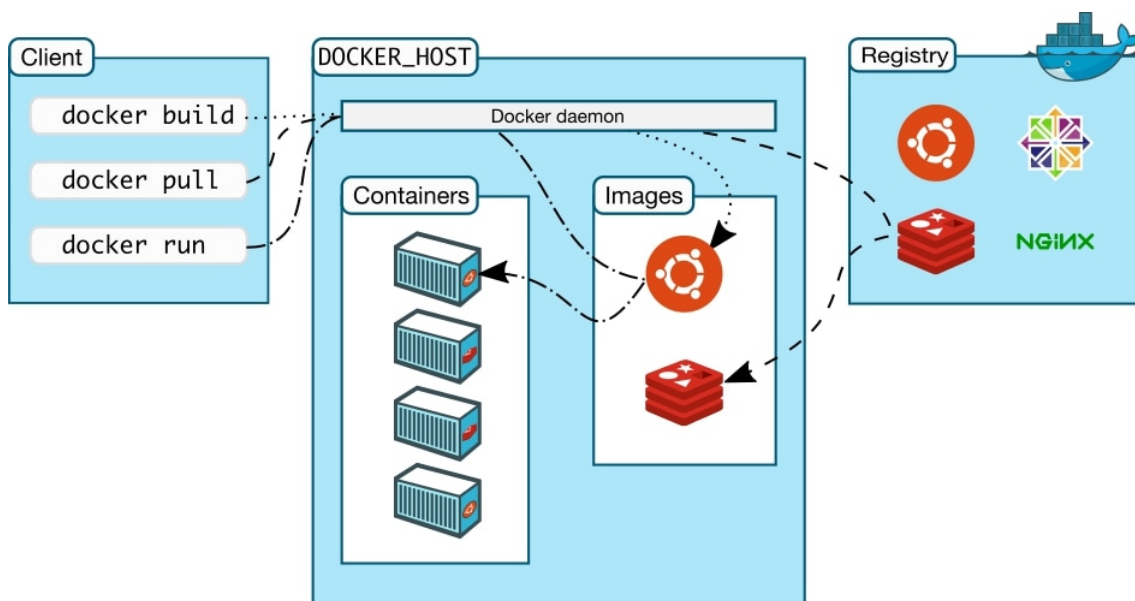
*Docker*² is een open source container platform dat sinds de *release* in 2013 ongeloofelijk populair is geworden. In november 2019 stond de teller van aantal *pulls* op de *Docker hub* op 130 miljard, in juli 2020 stond deze al op 242 miljard. Dat is bijna een verdubbeling op minder dan 8 maand tijd (Kreisa, 2020). Docker-containers kunnen overal draaien, in het datacenter, bij een externe serviceprovider of in de cloud. Docker containers kunnen zowel op Linux als op Windows draaien. Containers die op Windows gebaseerd zijn kunnen

²docker.com/

echter alleen op Windows systemen draaien, maar Linux containers kunnen op zowel Linux systemen en Windows systemen draaien (met behulp van een Linux VM). Dit komt omdat containers ontworpen zijn om het besturingssysteem van de host te gebruiken (Anil e.a., 2018).

2.2.1 Hoe werkt Docker

Docker gebruikt een *Client-Server* architectuur. Deze werkt als volgt: De Docker *Client* communiceert met de Docker *Daemon* (een proces dat op de achtergrond draait en bepaalde (onderhouds-)taken uitvoert). Deze is verantwoordelijk voor het bouwen, runnen en verspreiden van containers (Docker, 2021a).



Figuur 2.3: Docker componenten (Docker, 2021a)

2.2.2 Docker componenten en terminologie

Docker Daemon

De Docker Daemon (dockerd) luistert naar Docker Application programming interface (API) verzoeken en beheert Docker objecten waaronder *images*, *containers*, netwerken, en *volumes*.

Docker Client

De Docker Client is de meest gebruikte manier voor gebruikers om te communiceren met Docker. De client stuurt alle ingevoerde commando's (zoals `docker pull` en `docker run`) door naar de Docker Daemon die deze uitvoert.

Docker registries

Een *Docker register* is een bibliotheek die *Docker images* opslaat. Het standaard register voor Docker is de *Docker Hub*³. Als de Docker daemon geen lokale Docker image vindt gaat deze standaard in de *Docker Hub* zoeken. Wanneer gebruik gemaakt wordt van de `docker pull` of `docker run` commando's, worden de benodigde images uit het register gehaald.

Docker objecten

Docker maakt gebruik van images, volumes en netwerken, al deze onderdelen worden objecten genoemd. Volgens AquaSecurity (2021) en Docker (2021a) zijn dit de belangrijkste objecten:

- **Images:** Docker images zijn *read-only* sjablonen met instructies om een Docker container op te zetten. Docker images vanuit de Docker Hub zijn typisch onmiddellijk klaar voor gebruik, zonder verdere configuratie. Verder kan je tevens bijkomende instructies toevoegen aan de *base image* en deze opslaan als een nieuwe en aangepaste Docker image. Een Docker image is vaak gebaseerd op een andere images (i.e., Een nieuwe image kan gebaseerd zijn op een bestaande *Ubuntu* image maar installeert en configureert daarbij een *Apache* webserver). Alsook is het mogelijk om zelf een compleet nieuwe image te maken met behulp van een *dockerfile*.
- **Containers:** Een container is een uitvoerbare instantie van een image die wordt gecontroleerd via de Docker API. Een container heeft de mogelijkheid om te verbinden met andere containers, aan externe opslag of kan als basis dienen voor een nieuwe image.
- **Volumes:** De persistente gegevens die Docker containers kunnen gebruiken wordt opgeslagen in zogenaamde volumes. Deze volumes worden volledig gecontroleerd via de Docker API en bevinden zich buiten de container zelf. Hierdoor blijft het gewicht van de containers laag en kan de data blijven bestaan ook al wordt de container gestopt of verwijderd.

2.3 Container orkestratie

Container orkestratie helpt bij het opzetten, beheren, schalen en verbinden van een grote hoeveelheid containers. Container orkestratie helpt dus om complexe procedures te vereenvoudigen. Dit door veelvoorkomende processen en werkstromen te stroomlijnen en te optimaliseren. Een ander belangrijk onderdeel van orkestratie is het geautomatiseerd onderhoud van de applicaties die in de containers draaien (RedHat, 2021b).

³hub.docker.com/

Waarvoor wordt container orkestratie gebruikt?

Container orkestratie wordt vooral gebruikt voor het automatiseren en beheren van de configuratie en uitrol, het toewijzen van resources, de *Load balancing* en het monitoren van containers.

2.3.1 Container orkestratie tools

Om aan container orkestratie te gaan doen zijn er natuurlijk tools nodig die ons alle nodig functionaliteiten kunnen aanbieden. DevopsCube (2021) geeft een overzicht van de meest prominente orkestratie tools.

Kubernetes

Kubernetes(K8s)⁴ is een open source, container cluster manager en orkestratie tool. Het is gebouwd met een uitstekende resource manager voor het inzetten van containers op een efficiëntere manier. Kubernetes is voor vele organisaties de “de facto” container orkestratie tool geworden. Volgens CNCF (2021) zijn er meer dan 109 tools om containers te beheren, maar 89% is gebouwd met K8s aan de basis.

Openshift

Openshift behoort tot de 89% tools die gebouwd zijn bovenop Kubernetes. Het Openshift-project⁵ word onderhouden door RedHat⁶. Het heeft zowel een open source versie (*openshift origin*⁷) als een enterprise versie (*openshift container platform*⁸).

Hasicorp Nomad

Nomad⁹ is een orkestratieplatform van Hashicorp¹⁰ dat containers op schaal kan ondersteunen. Op het vlak van applicatiemanagement is het zeer sterk vergelijkbaar met Kubernetes. Echter kan Nomad ook niet-containerapplicaties kan beheren, met gevolg dat deze zich kan distantiëren van de andere orkestratie tools. Daarnaast kan Nomad feilloos geïntegreerd worden met andere tools van Hashicorp.

⁴kubernetes.io/

⁵openshift.com/

⁶redhat.com/

⁷github.com/openshift/origin

⁸openshift.com/products/container-platform

⁹nomadproject.io/

¹⁰hashicorp.com/

2.4 Kubernetes

Hier wordt ingegaan op wat Kubernetes(K8s) doet en hoe het werkt. Daarnaast worden de verschillende technische termen die eigen zijn aan K8s uitgelegd.

Kubernetes is een *open source*, container cluster manager en orkestratie tool. Het werd ontwikkeld door Google om hun container applicaties op grote schaal te kunnen orkestreren. Het project werd in 2014 *open source* gemaakt, dat wil zeggen dat Google heeft besloten om het Kubernetes project samen met de *community* to omwikkelen en onderhouden. Google heeft dan ook, in samenwerking met de Linux Foundation, de Cloud Native Computing Foundation(CNCF)¹¹ opgericht. Zij zijn nu de *maintainers* van het project en kiezen welke veranderingen er worden doorgevoerd. Hierdoor konden ook andere organisaties niet alleen gebruik maken van deze krachtige tool, maar tevens meewerken aan de ontwikkeling ervan. De term *Kubernetes* is Grieks voor “stuurman van een schip” (Kubernetes, 2021). Enkele diensten die door K8s aangeboden worden:

- Automatisch load-balancing op basis van de hoeveelheid verkeer.
- Opslag orkestratie: Het automatisch *mounten* van verschillende types opslag (lokale- of cloud opslag).
- Resource controle: K8s zorgt ervoor dat de beschikbare resources correct en efficiënt verdeeld worden.
- *Self-healing*: K8s kan containers heropstarten, vervangen of stopzetten als deze niet meer voldoende correct werken.

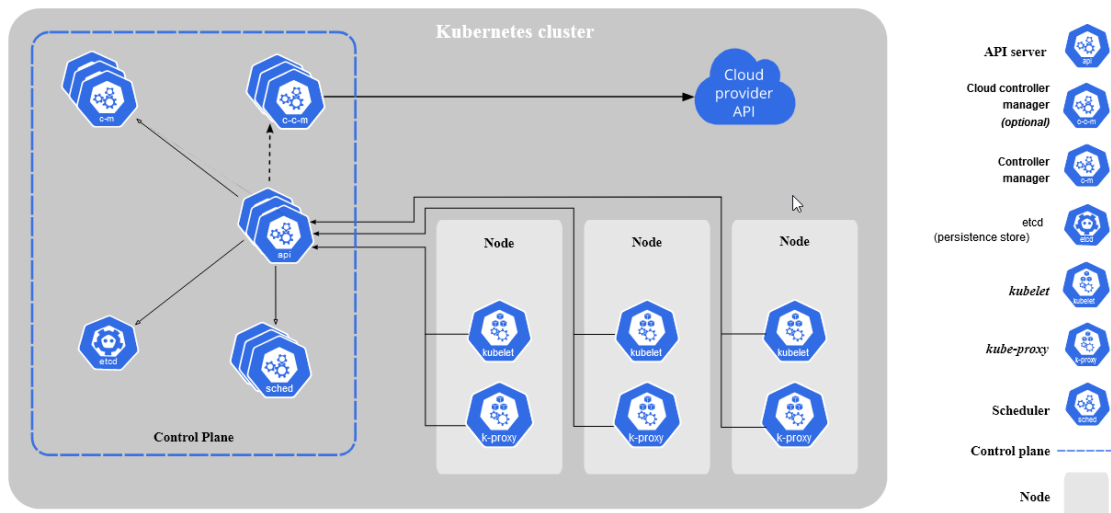
Kubernetes componenten en terminologie

Zoals beschreven in de documentatie van Pedersen e.a. (2021) en RedHat (2021a) worden Kubernetes en de bijhorende componenten als volgt gedefinieerd: Kubernetes is een *cluster* bestaande uit 2 grote delen namelijk het *control plane* en de *nodes*. Deze nodes zijn de door K8s georkestreerde containers. Elke cluster bestaat uit uit minstens één, maar meestal meerdere, nodes. De nodes worden gebruikt om *pods* te hosten. Pods zijn de kleinste mogelijke eenheid in een K8s systeem. Een pod bestaat uit één of meerdere containers die opslag- en netwerk *resources* delen (RedHat, 2021a). In Figuur 2.4 worden de verschillende componenten van een K8s cluster gevisualiseerd.

Het hart van een K8s cluster is de ***Control plane***, hierin bevinden zich alle componenten die de cluster controleren. Tevens zitten alle gegevens omtrent de staat van de cluster en de configuratie hierin verbonden. Alle componenten die deel uitmaken van de *Control plane* kunnen op verschillende machines draaien. Hierbij wordt er wel aangeraden om deze binnen eenzelfde machine te houden. De verschillende onderdelen en hun plaats binnen een cluster worden hieronder besproken (Pedersen e.a., 2021).

De ***kube-apiserver*** wordt gezien als de *Front-end* van een Kubernetes cluster. Deze is de link tussen *nodes* en *pods* van de cluster en de *Control plane*. Het doel van deze apiserver is om de communicatie van de nodes en de Kubernetes API te bolwerken. De apiserver is

¹¹ cncf.io/



Figuur 2.4: Kubernetes cluster componenten (Kubernetes, 2021)

ontworpen om horizontaal te schalen als deze te zwaar belast wordt. Hierbij creëert die nieuwe instanties van zichzelf.

Alle data en informatie met betrekking tot de status van de cluster wordt opgeslagen in *etcd*, een *key-value store database*.

De **kube-scheduler** is verantwoordelijk voor het toekennen van pods aan nodes en voor het verdelen van de resources tussen de verschillende Nodes. Het toekennen van een Node aan een Pod gebeurt aan de hand van verschillende factoren. De mogelijke factoren zijn onder andere de benodigde resources van een Pod en eventuele hardware- en software beperkingen.

Een **kube-controller-manager** bestaat uit verschillende *controllers* die allemaal een verschillende taak op zich nemen. Enkele van deze *controllers* zijn:

- **Node controller:** Het hoofddoel van deze controller is om op te merken en correct te reageren als er Nodes zouden wegvallen.
- **Job controller:** Deze zoekt naar zogenaamde *job-objecten*, die eenmalige taken voorstellen, en creëert bijgevolg de nodige Pods om deze taken uit te voeren.
- **Service Account & Token controllers:** Deze creëert standaard accounts en *API access tokens* voor nieuwe Pods.

Als het *Control plane* het hart van een K8s cluster is dan kunnen we de *Nodes* het lichaam noemen. Deze doen namelijk al het zware werk en worden bestuurt door de *Control plane*.

Het eerste onderdeel van een Node is de **kubelet**, een kleine applicatie die op elke Node aanwezig is. De *kubelet* houdt de gezondheid en status van de Pods, die binnen zijn Node draaien, in het oog. Wanneer de *Control plane* wil dat er iets gebeurt met de Node zorgt de *kubelet* ervoor dat deze acties correct uitgevoerd worden.

De **kube-proxy** is een netwerk *proxy* die de verantwoordelijk is voor de K8s *networkservices*. De *kube-proxy* verzorgt netwerkcommunicatie zowel binnen- als buiten de cluster.

Het volgende component, namelijk de **Container runtime**, werd al reeds besproken in sectie 2.1.

Ten slotte zijn er nog enkele extra *addons* die gebruikt kunnen worden om de functionaliteit van een K8s cluster uit te breiden. Een eerste voorbeeld van een *addon* is de mogelijkheid om persistent geheugen (ook wel *volumes* genoemd) toe te voegen zoals uitgelegd in sectie 2.2.2. Andere voorbeelden zijn het toevoegen van een **cluster DNS**, **Web UI** of **Dashboard** en het opslaan van de log bestanden van de cluster met **cluster-level logging**.

2.5 Security

Hier zal het *security* aspect van containers en container orkestratie besproken worden. In dit hoofdstuk wordt er geantwoord op de onderzoeksvragen “Wat zijn de belangrijkste security risico’s?” en “Welke beveiligings-tools zijn er en hoe werken ze?”.

Uit een rapport van Tripwire (2019) blijkt dat 94% van bevroagden bezorgd zijn over de veiligheid van hun containers. Uit hetzelfde rapport blijkt ook dat 47% zich bewust is van feit dat ze mogelijks kwetsbare containers gebruiken in hun productieomgeving. Het beveiligen van een K8s cluster is dan ook een zeer grote en complexe taak. Dit is omdat er veel verschillende onderdelen zijn die allemaal andere veiligheidsproblemen met zich mee kunnen meebrengen. Vanwege de hoeveelheid tijd en middelen die er nodig zijn voor het implementeren van goede veiligheidsmaatregelen wordt het door veel bedrijven gezien als een *nice to have* in plaats van een *need to have*.

2.5.1 Meest voorkomende security problemen

Een van de meest voorkomende veiligheidsproblemen bij het opzetten van een K8s cluster is het fout, of zelf helemaal niet, definiëren van parameters. Dit kan er mogelijks voor zorgen dat een aanvaller kan ‘ontsnappen’ uit een container (lees: toegang krijgen tot het host systeem). Ook het gebruik van ongecontroleerde Docker images kan ervoor zorgen dat een cluster van binnenuit compromitteren kan worden.

Een ander probleem zijn de onvermijdelijke *bugs* in K8s zelf. Deze is tevens wel een van de gemakkelijkste veiligheidsproblemen om op te lossen, namelijk door altijd de laatste versie van Kubernetes te installeren. Als er een *bug* of kwetsbaarheid wordt ontdekt, wordt deze in de meeste gevallen binnen enkele dagen weggewerkt door het K8s security team.

2.5.2 Hoe een container cluster beveiligen

Hieronder zullen er enkele mogelijke manieren om een container cluster te beveiligen besproken worden.

Lewis (2019) en Rice (2019) geven enkele tips voor het veilig opzetten van een K8s cluster. Ten eerste wordt het gebruik van *Role Based Access Control* (RBAC) stevig aangemoedigd. Door hiervan gebruik te maken, is het mogelijk om rollen toe te kennen aan bepaalde gebruikers. Deze rollen dicteren welke applicaties de gebruiker kan gebruiken en wat die ermee kan doen.

Vervolgens wordt er aangeraden om nooit de administrator (ook wel *root* genoemd) rechten te gebruiken, enkel als het nodig is voor de specifieke applicatie (i.e, *KubeDNS*). Dit kan ervoor zorgen dat wanneer een aanvaller controle krijgt over de cluster, hij nog steeds gelimiteerd is door de restricties die aan gewone accounts worden toegekend met RBAC. Men kan tevens beter de anonieme toegang tot de API ontfeggen en vervangen door een gecontroleerde toegang met RBAC.

Via de Security Policies parameter, namelijk RunAsUser, kan men verplichten om gebruik te maken van een normaal account. Dit wordt geïllustreerd in voorbeeld 2.5.2. Een *Security Policy* controleert de meer veiligheidsgevoelige aspecten van een cluster.

Een voorbeeld van zo een policy is de *seLinux* policy die, zoals zijn naam al doet vermoeden, beroep kan doen op de beveiligingsfunctionaliteiten van Security-Enhanced Linux. De *volumes* policy geeft dan weer controle over de soorten volumes die gebruikt kunnen worden.

```
apiVersion: v1
kind: Pod
metadata:
  name: Example
spec:
  securityContext:
    runAsUser: 1000
```

De communicatie tussen de Pods kan gecontroleerd worden aan de hand van zogenaamde *Network policies*. Dit zijn regels die aanduiden welke pods met elkaar kunnen communiceren en welke soort data er uitgewisseld kan worden (Bannister e.a., 2021; Venugopal, 2019). In voorbeeld 2.5.2 wordt inkomend verkeer naar pods met label “color: blue” alleen toegestaan als het afkomstig is van een pod met label “color: red” en dit enkel op poort 80. De *Network policies* kunnen handmatig worden geconfigureerd of er kan gebruik gemaakt worden van een zogenaamde *Container Network Interface* (CNI). Een CNI helpt met het configureren en onderhouden van de *Network policies*. In sectie 2.6.1 wordt één van de meest prominente CNI’s besproken, namelijk *Project Calico*.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
  namespace: default
spec:
  podSelector:
    matchLabels:
      color: blue
  ingress:
    - from:
      - podSelector:
          matchLabels:
            color: red
      ports:
        - port: 80
```

Fouten maken is menselijk, dus het is perfect mogelijk dat er tijdens het opstellen van de configuratie files enkele fouten insluipen. Daarom wordt het aangeraden om automatische

configuration checks uit te voeren zodat deze fouten vroegtijdig opgemerkt kunnen worden. Een tool die hiervoor vaak gebruikt wordt, namelijk **kube-bench**, wordt besproken in sectie 2.6.2.

Het spreekwoord ‘De ketting is maar zo sterk als zijn zwakste schakel’ is ook hier van toepassing. Hoe beveiligd de cluster ook mag zijn, één onveilige container is voldoende om alle geherinvesteerde moeite teniet te doen.

Volgens Lewis (2019) zijn er verschillende manieren om ervoor te zorgen dat er enkel veilige containers gebruikt worden. De eerste is om enkel *trusted base images* binnen te halen. Dit zijn containers die volledig *from scratch* ontwikkeld zijn door het bedrijf, of de persoon, die container zal draaien. Dit zijn containers die volledig ontwikkeld zijn door het bedrijf dat ze in gebruik neemt. Het *Base* gedeelte van de naam komt van het feit dat deze containers meestal zeer simpel zijn. Zo kan elke container applicatie gebaseerd worden op dezelfde veilige container *image*. Een extra veiligheidsmaatregel die vaak aangeraden wordt is het opzetten van een *private registry*. Wat een *registry* precies is werd al reeds verduidelijkt in sectie 2.2.2. Een *private registry* is niets anders dan een *registry* die volledig door het bedrijf wordt gecontroleerd en enkel door hen wordt gebruikt. Zij kiezen zelf welke *images* er wel of niet worden toegevoegd. Hierdoor vermijdt men dat er per ongeluk een onveilige container wordt gebouwd binnen een cluster.

Spijtig genoeg is niet iedereen die met container clusters werkt even bezorgd over de veiligheid ervan. Om het gebruik van onveilige Pods te vermijden kan men een *policy agent* gebruiken. Hiermee kan er automatisch gecontroleerd worden of de Pods aan alle veiligheidsvoorschriften voldoen, alvorens ze ingeschakeld worden. Een voorbeeld van een *policy agent* is de **Open Policy Agent**¹², ontwikkeld door de *text Cloud Native Computing Foundation (CNCF)*.

2.6 beveiligings-tools

In dit hoofdstuk zal er geantwoord worden op de onderzoeksvraag “Welke beveiligings-tools zijn er?”

Een andere manier om een K8S cluster verder te beveiligen is om bepaalde beveiligings-tools te overwegen. Taylor (2019) geeft een overzicht met de meest prominente tools.

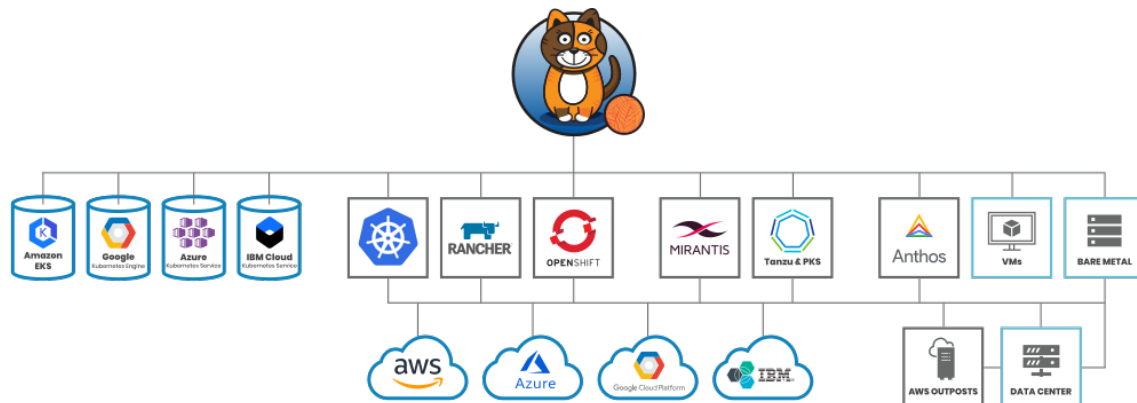
2.6.1 Project Calico

Project Calico¹³ is volledig *open source* en volgens Armstrong (2021) de meeste gebruikte beveiligings-tools voor K8s. Calico is niet enkel een K8s beveiligings-tools maar is tevens volledig geïntegreerd met vele andere platformen. Figuur 2.5 geeft een overzicht van de door Calico ondersteunde platformen. Project Calico creëert een soort van *microfirewall*

¹²openpolicyagent.org/

¹³projectcalico.org/

rond elke service. De policies die ingesteld kunnen worden in Calico worden automatisch omgezet naar firewall regels. Deze worden vervolgens toegepast op elke service. Calico heeft zich onlangs aangesloten bij de CNCF, waardoor het onder deskundig toezicht staat en nog beter geïntegreerd is in het Kubernetes ecosysteem.



Figuur 2.5: Platformen die ondersteunt worden door Calico (Tigra, 2021)

2.6.2 Kube-Bench

Kube-Bench¹⁴ is een *open source* applicatie geschreven in *Go*¹⁵ die controleert of k8s veilig is geïmplementeerd door middel van controles uit te voeren op de cluster. De controles zijn gebaseerd op enkele guidelines die het *Center for Internet Security*¹⁶ (CIS), een organisatie die richtlijnen opmaakt voor het schrijven van veilige code, heeft opgesteld. De zogenaamde *CIS Kubernetes Benchmarks* worden geschreven door de Kubernetes *community* en worden door de CIS gecontroleerd en gebundeld. *Kube-Bench* controleert niet alleen of er fouten in de beveiliging van een cluster zitten, het geeft ook mogelijke oplossingen. Mogelijke controles zijn het controleren van gebruikersautorisatie- en -authenticatie, de versleuteling van gegevens en kijken of de cluster het principe van *least privilege* volgt (een gebruiker krijgt enkel toegang tot gegevens die hij echt nodig heeft). De testen worden gedefinieerd in een 'YAML Ain't Markup Language' (YAML) bestand zodat ze gemakkelijk aangepast en uitgebreid kunnen worden (Rice e.a., 2021). De testen worden uitgevoerd op elke individuele node in de cluster waardoor het vooral geschikt is voor kleinere opstellingen.

¹⁴github.com/aquasecurity/kube-bench

¹⁵golang.org/

¹⁶cisecurity.org/benchmark/kubernetes/

2.6.3 Kube-hunter

Kube-hunter¹⁷ is, net zoals *Kube-Bench*, een open source applicatie gemaakt door *Aqua Security*¹⁸. *Kube-hunter* breidt de functionaliteiten van *Kube-Bench* uit door penetratie testen uit te voeren op de cluster. Dit zorgt ervoor dat administrators problemen met een cluster gemakkelijk kunnen opsporen en verhelpen.

Er zijn drie verschillende manieren om *kube-hunter* te gebruiken. De eerste maakt gebruik van het IP-adres van de cluster om vanop afstand de testen uit te voeren. Bij de tweede manier wordt *kube-hunter* lokaal op één van de machines in de cluster geïnstalleerd om die netwerkinterfaces van die specifieke machine te testen. De derde en laatste manier om *kube-hunter* te gebruiken is om het te installeren in een Pod binnen de cluster. Deze manier kan aantonen wat voor potentiële schade er kan aangericht worden als één van de Pods gecompromitteerd zou worden.

¹⁷github.com/aquasecurity/kube-hunter

¹⁸aquasec.com/

3. Methodologie

Om het effect van zowel K8s *best-practices* als K8s beveiligings-tools op een cluster te onderzoeken werd er gekozen om het onderzoek in 3 scenario's onder te verdelen. Met als doel gegevens te verzamelen over enkele criteria, namelijk het resource gebruik, de stabiliteit en de opstarttijd van de cluster. Voordat we aan de scenario's kunnen beginnen, worden eerst nog alle stappen doorlopen voor het opzetten van de cluster. Nadat de cluster opgezet is zullen we beginnen met het eerste scenario, namelijk het opzetten van een basis cluster om de functionaliteit van K8s aan te tonen en om baseline gegevens te verzamelen voor het onderzoek. Deze cluster zal dienen als basis om verdere scenario's verder uit te werken. Als tweede worden er enkele "best-practice" gehanteerd bij het opzetten en de configuratie van de cluster, dit om te onderzoeken wat voor effect deze hebben op de cluster. Ten slotte zal er bij het opzetten en configureren van de cluster gebruik gemaakt worden van enkele K8s beveiligings-tools, dit om te testen wat deze precies doen en wat voor effect deze kunnen hebben op de criteria.

Gedurende dit onderzoek werd er gebruik gemaakt van de volgende hardware en software:

- Besturingssysteem: Pop!_OS 20.10 x86_64
- Host systeem: MSI GL62 7REX
- Kernel: 5.11.0-7614-generic (Linux)
- Cloud provider: Linode¹
- Nodes: 3 X Linode 2GB (1CPU core, 2GB RAM, 50GB opslag, Debian 5.10.24-1)

¹cloud.linode.com/

3.1 Opzetten van de Kubernetes cluster

In dit hoofdstuk zal de basis cluster opgezet worden. Deze manier van werken zal ook gebruikt worden om de andere scenario's uit te voeren.

Voor het opzetten van de cluster zijn er maar drie onderdelen nodig.

- Account bij een cloud provider, in dit geval Linode.
- Kubectl om de cluster te besturen.
- De Docker image om een applicatie te deployen op de cluster.

Aangezien Kubectl op zowel Linux, Windows als MacOS kan draaien, en de cluster zelf bij de cloud provider wordt gehost kunnen deze scenario's op bijna elke computer worden nagebootst. De stappen in volgende hoofdstukken worden allemaal uitgevoerd op een Linux systeem en kunnen dus verschillen op Windows of MacOS. Het is ook aangeraden om een nieuwe directory aan te maken zodat alle bestanden op eenzelfde plaats te vinden zijn.

Voor dit onderzoek zal gebruik gemaakt worden van de ingebouwde “Linode Kubernetes Engine”(LKE). Deze installeert automatisch de correcte onderdelen op de verschillende worker nodes en creëert ook een gratis master node. Hiervoor werd gekozen omdat het volledig opzetten van een K8s cluster niet binnen de scope van dit onderzoek ligt. Door gebruik te maken van de LKE kunnen we ons dus focussen op de beveiliging van de cluster.

3.1.1 Linode Kubernetes cluster

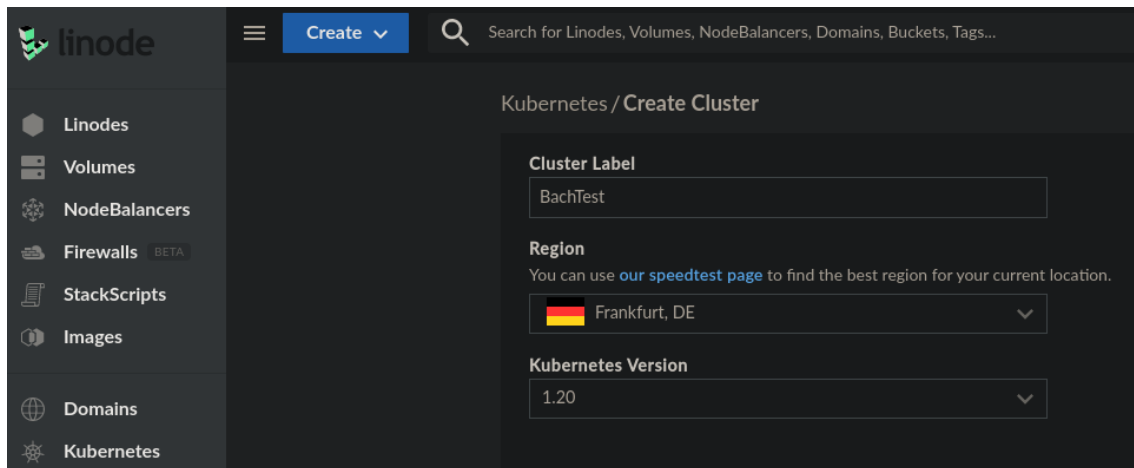
Hieronder worden de verschillende stappen die doorlopen zijn bij het opzetten van een Linode K8s cluster beschreven.

Voor het creëren van een cluster in Linode hebben we enkel een Linode account nodig. Het aanmaken van de cluster zelf is zeer gemakkelijk en gebeurt via `cloud.linode.com > Kubernetes > Create Cluster`. Vul dan de naam, regio en versie van de cluster in zoals in figuur 3.1.

Vervolgens kunnen we nodes toevoegen aan de cluster. Voor dit onderzoek gaan we een kleinschalige cluster opzetten met drie worker nodes en één master node. In figuur 3.2 zijn de verschillende soorten nodes die we kunnen toevoegen opgelijst. Tijdens deze scenario's zal gebruik gemaakt worden van drie “Linode 2GB” nodes. Daarnaast is het ook mogelijk om nodes van verschillende groottes in dezelfde cluster toe te voegen. De master node is niet meegerekend in deze drie nodes maar wordt door Linode gratis aangeboden bij het opzetten van een K8s cluster.

Als alle nodes zijn toegevoegd wordt de cluster aangemaakt. Linode zal hierbij in de achtergrond de gekozen nodes aanmaken alsook de master node klaarmaken voor gebruik.

Eenmaal de cluster is aangemaakt en opgestart is moeten we deze nog configureren. Deze configuratie gebeurt via Kubectl in volgende stappen.



Figuur 3.1: Aanmaken Linode cluster

Als eerste moeten we het kubeconfig.yaml bestand downloaden van Linode. In dit bestand staan alle nodig details om te Kubectl te connecteren met onze cluster. De kubeconfig van de net aangemaakte cluster is te zien in figuur 3.3 (enkele waarden werden ingekort om de leesbaarheid te vergroten).

Vervolgens moet het pad naar het bestand geëxporteerd worden naar de omgevingsvariabele “KUBECONFIG” met het volgende commando:

```
$ export KUBECONFIG=<pad naar file>/kubeconfig.yaml
```

We kunnen testen of dit gelukt is door het volgende commando uit te voeren. Dit zou de drie aangemaakte nodes in onze cluster moeten teruggeven.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
lke25332-32960-608682818f2e	Ready	<none>	6d4h	v1.20.5
lke25332-32960-60868281f030	Ready	<none>	6d4h	v1.20.5
lke25332-32960-608682824efd	Ready	<none>	6d4h	v1.20.5

Nu Kubectl in contact staat met de “kube-apiserver” dewelke op de master node draait, kunnen we beginnen met het configureren van onze cluster. Dit kan zowel met “ad-hoc” commando’s als met zogenaamde deployments die werken via het *principle of desired state*. Met andere woorden specificeren de deployments de verschillende aspecten van de cluster waarbij K8s ervoor zorgt dat aan deze specificaties voldaan worden.

Een deployment is eigenlijk niets anders dan een YAML bestand waarin beschreven wordt hoe de cluster er moet gaan uitzien. Via het volgende commando wordt een deployment op een cluster gezet.

```
$ kubectl apply -f deployment.yaml
```

Add Node Pools
Add groups of Linodes to your cluster with a chosen size.

Shared CPU **Dedicated CPU** **High Memory**

Shared CPU instances are good for medium-duty workloads and are a good mix of performance, resources, and price.

Plan	Monthly	Hourly	RAM	CPUs	Storage				
Linode 2GB	\$10	\$0.015	2 GB	1	50 GB	–	3	+	Add
Linode 4GB	\$20	\$0.03	4 GB	2	80 GB	–	0	+	Add
Linode 8GB	\$40	\$0.06	8 GB	4	160 GB	–	0	+	Add
Linode 16GB	\$80	\$0.12	16 GB	6	320 GB	–	0	+	Add
Linode 32GB	\$160	\$0.24	32 GB	8	640 GB	–	0	+	Add
Linode 64GB	\$320	\$0.48	64 GB	16	1280 GB	–	0	+	Add
Linode 96GB	\$480	\$0.72	96 GB	20	1920 GB	–	0	+	Add
Linode 128GB	\$640	\$0.96	128 GB	24	2560 GB	–	0	+	Add
Linode 192GB	\$960	\$1.44	192 GB	32	3840 GB	–	0	+	Add

Figuur 3.2: Toevoegen van nodes aan cluster

3.1.2 Resetten van cluster

Na het uitvoeren van elk scenario zal de cluster weer worden gereset zodat het vorige scenario geen impact kan hebben op de rest van het onderzoek. Het resetten van de cluster bestaat vooral uit het verwijderen van deployments en services. Het verwijderen van een deployment wordt gedaan aan de hand van volgend commando.

```
$ kubectl delete deployment <Deployment naam>
```

Het verwijderen van een service verloopt op dezelfde manier.

```
$ kubectl delete service <Service naam>
```

3.1.3 Gegevens- verzameling en verwerking

Om conclusies te kunnen trekken uit dit onderzoek hebben we gegevens nodig. Meer bepaald gegevens over de bovengenoemde criteria, namelijk het *resource* gebruik, de stabiliteit en de opstarttijd van de cluster. Deze gegevens zullen gebruikt worden om de


```

1  apiVersion: v1
2  kind: Config
3  preferences: {}
4
5  clusters:
6  - cluster:
7      certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSOtLS0txvd0ZURVRNQkVHQT
8      FVRQpBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQ1FBRGdnRVBBERENDQVFvQ2dn
9      server: https://e08328fb-02d7-49a6-9f12-b4bdeb98510c.eu-central-2.linode.lke.net:443
10     name: lke25796
11
12  users:
13  - name: lke25796-admin
14     user:
15         as-user-extra: {}
16         token: eyJhbGciOiJSUzI1NiIsImtpZCI6IiZ1a29MSDNqVWhXS1YyMjFNNVhJeHFjTFN
17         XaXhwYVhNT3FWb2NjTWFOV2MifQ.eyJpc3MiOiJrdWJ1cm5ldGVzL3N1cnZpY2VhY2NvdW50Iiwia3ViZ
18
19  contexts:
20  - context:
21      cluster: lke25796
22      namespace: default
23      user: lke25796-admin
24      name: lke25796-ctx
25
26  current-context: lke25796-ctx

```

Figuur 3.3: kubeconfig.yaml

onderzoeksvraag “Welke impact hebben *best practices* en beveiligings-tools op de criteria?” te beantwoorden. De verzameling van deze gegevens zal gebeuren via *sysstat*². Dit is open source tool die ons in staat stelt om een hele hoop verschillende soorten data van ons systeem te exporteren naar bruikbare formaten (zoals JSON, CSV en XML). Deze bestanden zullen later in RStudio als bron gebruikt worden om statische modellen mee te maken. Sysstat zal op een van de Nodes in de cluster worden geïnstalleerd. Dit door middel van een ssh verbinding op te zetten met één van de Nodes en dan volgend commando uit te voeren.

```
$ sudo apt-get install sysstat
```

Vervolgens moeten we de data collectie aanzetten door in het bestand `/etc/default/sysstat` de variabele `ENABLED` van “false” naar “true” om te zetten. Vanaf nu zal een van de meegeleverde tools, namelijk `sar`, elke 10 minuten een nieuwe entry maken in het bestand `/var/log/sysstat/sa<dag van de maand>`. We kunnen deze bestanden uitlezen met behulp van het `sar` commando. Om het CPU gebruik van het systeem om te zetten naar een csv bestand, het later in RStudio te verwerken, kunnen we het volgende commando gebruiken. In figuur 3.4 is het net gecreëerde csv bestand te zien.

²github.com/sysstat/sysstat

```
$ sar -f /var/log/sysstat/sa08 >> /root/CpuGebruikDag8.csv
```

	Linux	5.10.0-5-cloud-amd64	(lke25332-32960-608682818f2e)	05/06/21	_x86_64_	(1	CPU)
1	00:00:02	CPU	%user	%nice	%system	%iowait	%steal
2	00:05:01	all	6.36	0.00	3.07	0.01	0.45
3	00:15:01	all	6.31	0.00	3.05	0.01	0.20
4	00:25:01	all	6.42	0.00	3.08	0.01	0.34
5	00:35:01	all	6.26	0.00	2.98	0.01	0.16
6	00:45:01	all	6.16	0.00	2.88	0.01	0.16
7	00:55:01	all	6.03	0.00	2.96	0.01	0.12
8	01:05:01	all	6.30	0.00	2.93	0.01	0.18
9	01:15:01	all	6.35	0.00	3.01	0.00	0.19
10	01:25:01	all	6.26	0.00	3.01	0.01	0.25
11	23:55:01	all	6.22	0.00	3.16	0.01	0.22
12	23:59:01	all	6.02	0.00	2.94	0.00	0.24
13	00:00:01	all	5.96	0.00	3.32	0.02	0.30
14	Average:	all	6.30	0.00	3.08	0.01	0.40

Figuur 3.4: Klein deel van het gebruikte csv bestand

Vervolgens kunnen we door andere *flags* mee te geven aan het sar commando verschillende soorten data uit de logbestanden halen. Het RAM gebruik kan bijvoorbeeld met volgend commando in een csv bestand worden opgeslagen.

```
$ sar -f /var/log/sysstat/sa08 -r >> /root/RAMGebruikDag8.csv
```

Als laatste moeten de gegevens lokaal op de computer beschikbaar zijn. Dit wordt gedaan door een SFTP verbinding te maken met de node waar sysstat op geïnstalleerd is en de volgende commando's uit te voeren.

```
$ sftp root@172.105.86.96
root@172.105.86.96 s password:
Connected to 172.105.86.96.
sftp> lcd /home/nick/Desktop
sftp> get CpuGebruikDag8.csv
Fetching /root/CpuGebruikDag8.csv to CpuGebruikDag8.csv
```

Nu alle nodige gegevens zijn verzamelt kan er begonnen worden aan het verwerken van deze gegevens. Dit wordt gedaan door gebruik te maken van R en Rstudio.

Het eerste criteria is het vergelijken van het gemiddeld resource gebruik van zowel de CPU als het RAM geheugen. Voor het CPU gebruik kan dit simpelweg door de laatste rij van het csv bestand af te lezen. Het gemiddeld gebruik van het RAM geheugen wordt niet automatisch gegenereerd door sar en moet nog berekent worden. De R code om dit te doen wordt beschreven in figuur 3.5.

```

1 library(readr)
2 library(ggplot2)
3 TestRam <- read_table2("SC1_RAMGebruikDag9.csv", skip = 1)
4
5 names(TestRam)[names(TestRam) == "%memused"] <- "RAM"
6 ram <- TestRam[4]
7
8 summary(ram)

```

Figuur 3.5: R code om boxplot van CPU gegevens te bekomen

Het tweede criteria is de stabiliteit van het systeem. Dit zal onderzocht worden door een Boxplot te maken op basis van de verzamelde gegevens om de hoeveelheid outliers visueel voor te stellen. De R code om dit te doen voor het CPU gebruik staat in figuur 3.6. In dit R script wordt het csv bestand eerst ingelezen zonder de eerste rij omdat deze niet nuttig is voor het verdere onderzoek. Vervolgens nemen we enkel de achtste kolom over omdat hier de hoeveelheid ongebruikte CPU kracht staat. Deze wordt dan afgetrokken van honderd om zo tot de gebruikte hoeveelheid CPU te komen. Als laatste wordt de naam van kolom veranderd zodat het gemakkelijker is om deze te gebruiken bij het maken van de boxplot. De laatste lijnen van het script zijn verantwoordelijk voor het creëren en opmaken van de boxplot.

```

1 # Boxplot van het cpu gebruik van de laatste 24h
2
3 library(readr)
4 library(ggplot2)
5 CpuData <- read_table2("SC2_CPUGebruikDag12.csv", skip = 1)
6
7 CpuData <- CpuData[8]
8 CpuUsage <- 100-CpuData
9
10 names(CpuUsage)[names(CpuUsage) == "%idle"] <- "Cpu"
11
12 cpuBoxPlot <- ggplot(CpuUsage, aes(x=Cpu)) + geom_boxplot()
13
14 cpuBoxPlot <- cpuBoxPlot+ theme(axis.text.y = element_blank(), axis.ticks.y = element_blank())
15
16 cpuBoxPlot <- cpuBoxPlot + labs(x = "CPU gebruik in %")
17
18 cpuBoxPlot

```

Figuur 3.6: R code om boxplot van CPU gegevens te bekomen

Voor de gegevens van het RAM gebruik wordt op dezelfde manier te werk gegaan. De R code om de gegevens met betrekking tot het RAM geheugen wordt beschreven in figuur 3.7.

Om ervoor te zorgen dat we de stabiliteit van de verschillende scenario's met elkaar kunnen vergelijken is het ook nodig om te weten hoeveel *outliers* er precies aanwezig zijn. Dit is niet altijd zo gemakkelijk te tellen als deze kort bij elkaar liggen. De *outliers* kunnen

```

1 library(readr)
2 library(ggplot2)
3 TestRam <- read_table2("SC2_RAMGebruikDag12.csv", skip = 1)
4
5 names[TestRam][names[TestRam] == "%memused"] <- "RAM"
6 ram <- TestRam[4]
7
8 RAMBoxPlot <- ggplot(ram,aes(x=RAM)) + geom_boxplot()
9
10 RAMBoxPlot <- RAMBoxPlot + theme(axis.text.y = element_blank(), axis.ticks.y = element_blank())
11
12 RAMBoxPlot <- RAMBoxPlot + labs(x = "Ram gebruik in %")
13
14 RAMBoxPlot

```

Figuur 3.7: R code om boxplot van CPU gegevens te bekomen

allemaal opgelijst worden met het volgend commando.

```

> boxplot(ram)$out
[1] 60.90 60.96 61.01 61.07 61.06 61.16 61.20 61.21 61.31 61.29 61.35
[12] 61.49 61.44 61.49 61.55 61.59 61.68 61.72 61.78 61.76 61.80 61.84
[23] 61.95 61.95 61.99 62.04 62.13 62.17 62.30 62.43 62.39 62.43 62.47

```

Als derde criteria zal er gekeken worden naar het algemene resource gebruik van de cluster. Deze zal voorgesteld worden als een lijngrafiek waarin het resource gebruik van zowel de CPU als het RAM geheugen worden voorgesteld ten opzichte van de tijd. Het R script om dit te bekomen staat in figuur 3.8.

```

1 library(readr)
2 library(ggplot2)
3 CPUOverTime <- read_table2("SC1_CpuGebruikDag9.csv", skip = 1)
4 CPUOverTime <- head(CPUOverTime, - 2)
5
6 #CPUOverTime <- CPUOverTime[1:8]
7 names(CPUOverTime)[names(CPUOverTime) == "00:00:01"] <- "Tijd"
8 names(CPUOverTime)[names(CPUOverTime) == "%idle"] <- "Cpu"
9
10 CPUOverTime[8] <- 100 - CPUOverTime[8]
11
12 CPUGraph <- ggplot(CPUOverTime, aes(Tijd,Cpu,group = 1)) + geom_line()
13
14 CPUGraph <- CPUGraph + theme(axis.ticks.x = element_blank(), axis.text.x = element_blank())
15
16 CPUGraph <- CPUGraph + coord_cartesian(ylim = c(5, 15))
17
18 CPUGraph

```

Figuur 3.8: R code om grafiek van CPU gegevens te bekomen

Als laatste criteria zal de tijd die de node nodig heeft om op te starten vergeleken worden. Dit kan simpelweg door het volgende commando uit te voeren op de node.

```
$ systemd-analyze
```

3.2 Scenario 1: Cluster opstelling zonder oog voor security

Het eerste scenario bestaat eruit om een simpele cluster op te zetten zonder specifiek oog te hebben voor de beveiliging. De deployment die zal gebruikt worden wordt in figuur 3.9 weergegeven. Enkele van de variabelen worden hieronder uitgelegt:

- `image: thenetworkchuck/nccoffee:pourover`: Dit is de Docker image met de statische demo site.
- `replicas:3`: Het aantal pods die moeten worden gecreëerd wordt door deze variabele ingesteld. In dit geval zijn dat er 3.
- `imagePullPolicy: Always`: Deze variabele zorgt er in dit scenario voor dat de Docker image steeds word gedownload uit de registry als de deployment wordt gebruikt.
- `containerPort: 80`: Dit zorgt ervoor dat de site via poort 80 beschikbaar zal zijn.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: networkchuckcoffee-deployment
5   labels:
6     app: nccoffee
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: nccoffee
12   template:
13     metadata:
14       labels:
15         app: nccoffee
16     spec:
17       containers:
18         - name: static-site
19           image: thenetworkchuck/nccoffee:pourover
20           imagePullPolicy: Always
21           ports:
22             - containerPort: 80
```

Figuur 3.9: deployment.yaml

Als de deployment YAML bestand klaar is kan deze met volgend commando op de cluster zetten.

```
$ kubectl apply -f deployment.yaml
```

Het opzetten van de pods kan gevolgt worden met het volgende commando. In figuur 3.10 is te zien hoe de pods een voor een klaargemaakt worden.

```
$ kubectl get pods
```

```
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
networkchuckcoffee-deployment-9656545f5-9lcmf	1/1	Running	0	21s
networkchuckcoffee-deployment-9656545f5-rz669	1/1	Running	0	21s
networkchuckcoffee-deployment-9656545f5-shrc5	1/1	Running	0	21s

Figuur 3.10: Pods worden klaargemaakt

Als we nu de deployment willen aanpassen zonder de cluster volledig af te breken kunnen we het commando

```
$ kubectl edit deployments networkchuckcoffee-deployment
```

gebruiken. In figuur 3.11 is de output van dit commando te zien. Als we in dit bestand bijvoorbeeld de replicas variabele zouden aanpassen en het bestand opstaan, zal K8s merken dat er iets is veranderd. Omdat er gebruik wordt gemaakt van het “Principle of desired state” zal er automatisch aan de nieuwe specificaties worden voldaan. Om dit aan te tonen zal de variabele veranderd worden van drie naar vijf.

Als we het aantal replicas verhoogd hebben van drie naar 5 en het `kubectl get pods` commando uitvoeren krijgen we, zoals in figuur 3.12 te zien, dat er twee extra pods worden bijgemaakt. Deze pods worden via de *scheduler* op de master node verdeeld over de drie nodes in onze cluster.

Nu we klaar zijn met het opzetten van de deployment is het mogelijk om de cluster bloot te stellen aan het internet. Momenteel is het nog niet mogelijk om de site van buiten de cluster te bekijken. Dit komt omdat er nog geen “service” draait die de cluster opent naar het internet. De service die hier zal opgezet worden zal een “loadbalancer” creëren op Linode die automatisch al het verkeer verdeelt over alle pods in de cluster. Het YAML bestand om de service aan te maken is te zien in figuur 3.13. De code selector: app: nccoffee zorgt ervoor dat de loadbalancer het verkeer tussen alle pods waar de applicatie “nccoffee” op draait verdeelt. Wanneer er pods bijkomen of verdwijnen zal de loadbalancer zich automatisch aanpassen.

Het volgende commando kan gebruikt worden om het publieke IP-adres van de loadbalancer te vinden.

```
$ kubectl get services
```

Er is ook een manier om een meer gedetailleerde beschrijving van de service te krijgen. Namelijk door middel van het volgende commando. In figuur is de output van dit commando en alle informatie over de loadbalancer service te zien. De IP-adressen die worden getoond bij Endpoints: zijn de IP-adressen van alle achterliggende pods waar de “nccoffee” applicatie op draait. Als er nu naar het publieke IP-adres van de loadbalancer wordt gesurft is de demo website te zien zoals in figuur 3.15.

```

1  # Please edit the object below. Lines beginning with a '#' will be ignored,
2  # and an empty file will abort the edit. If an error occurs while saving this file will be
3  # reopened with the relevant failures.
4  #
5  apiVersion: apps/v1
6  kind: Deployment
7  metadata:
8    annotations:
9      deployment.kubernetes.io/revision: "1"
10     kubectl.kubernetes.io/last-applied-configuration: |
11       {"apiVersion":"apps/v1","kind":"Deployment",metadata":{"annotations":{}
12       ,"labels":{"app":"nccoffee"},"name":"networkchuckcoffee-deployment"
13       ,"namespace":"default"},"spec":{"replicas":3,"selector":{"matchLabels":{"app":"nccoffee"}}
14       ,"template":{"metadata":{"labels":{"app":"nccoffee"}}
15       ,"spec":{"containers":[{"image":"thenetworkchuck/nccoffee:pourover"
16       ,"imagePullPolicy":"Always","name":"static-site"
17       ,"ports":[{"containerPort":80}]}]}}}}
18     creationTimestamp: "2021-05-03T16:07:03Z"
19     generation: 1
20     labels:
21       app: nccoffee
22     name: networkchuckcoffee-deployment
23     namespace: default
24     resourceVersion: "75837"
25     uid: c3ca052e-01d6-4083-a9ac-de1f8ff48170
26  spec:
27     progressDeadlineSeconds: 600
28     replicas: 3
29     revisionHistoryLimit: 10
30     selector:
31       matchLabels:
32         app: nccoffee

```

Figuur 3.11: Output “kubectl edit deployment” commando

```
$ kubectl describe services coffee-service
```



```

> kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
networkchuckcoffee-deployment-9656545f5-8dmrh  0/1     ContainerCreating   0           2s
networkchuckcoffee-deployment-9656545f5-9lcmf  1/1     Running             0           77m
networkchuckcoffee-deployment-9656545f5-c2f5l  0/1     ContainerCreating   0           2s
networkchuckcoffee-deployment-9656545f5-rz669  1/1     Running             0           77m
networkchuckcoffee-deployment-9656545f5-shrc5  1/1     Running             0           77m

```

Figuur 3.12: Pods worden klaargemaakt

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: coffee-service
5    annotations:
6      service.beta.kubernetes.io/linode-loadbalancer-throttle: "4"
7    labels:
8      app: coffee-service
9  spec:
10   type: LoadBalancer
11   ports:
12   - name: http
13     port: 80
14     protocol: TCP
15     targetPort: 80
16   selector:
17     app: nccoffee
18   sessionAffinity: None

```

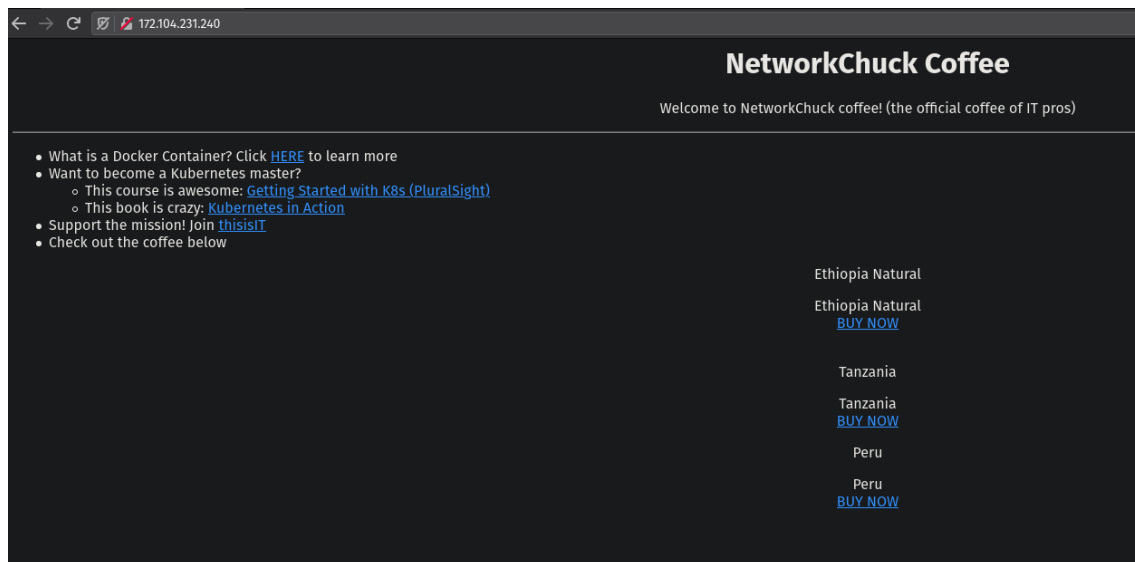
Figuur 3.13: service.yaml

```

> kubectl describe services coffee-service
Name:                coffee-service
Namespace:            default
Labels:               app=coffee-service
Annotations:          service.beta.kubernetes.io/linode-loadbalancer-throttle: 4
Selector:             app=nccoffee
Type:                 LoadBalancer
IP Families:          <none>
IP:                  10.128.236.99
IPs:                  10.128.236.99
LoadBalancer Ingress: 172.104.231.240
Port:                 http 80/TCP
TargetPort:           80/TCP
NodePort:             http 30852/TCP
Endpoints:            10.2.1.2:80,10.2.1.3:80,10.2.1.4:80 + 2 more...
Session Affinity:     None
External Traffic Policy: Cluster
Events:
  Type    Reason              Age   From                Message
  ----    -
  Normal  EnsuringLoadBalancer  24s   service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer  24s   service-controller  Ensured load balancer

```

Figuur 3.14: Informatie over de loadbalancer service.



Figuur 3.15: De demo website is nu zichtbaar vanop het internet.

3.3 Scenario 2: Cluster opstelling met gebruik van best practices

Het tweede scenario bestaat eruit om een simpele cluster op te zetten en gebruik te maken van enkele *security best practices*. *Best-practices* zoals het gebruik maken van een *trusted base image* en een *private registry* zullen niet gebruikt worden omdat deze vooral dienen om de ontwikkeling van een container applicatie te beveiligen. *Pod Security Policies* en *RBAC* zullen wel toegepast worden op de cluster.

Als eerste zullen er enkele Security Policies geconfigureerd worden. Het “PodSecurityPolicy.YAML” bestand is te zien in figuur 3.16, enkele van de variabelen worden hieronder uitgelegd:

seccomp.security.alpha.kubernetes.io/defaultProfileName: 'runtime/default': Op Linode worden nodes automatisch vooraf geïnstalleerd en geconfigureerd met AppArmor (gelijkaardig aan SELinux). Deze variabele zorgt ervoor dat de standaard “container runtime” profiel wordt gebruikt. Nog een mogelijke optie is *Docker runtime engine* (in dit geval werd deze al standaard ingesteld).

privileged: false beslist of er geprivilegieerde pods aangemaakt kunnen worden. Dit staat standaard op “false” en moet enkel op “true” gezet worden wanneer een pod andere onderdelen van host nodig heeft om te werken.

allowPrivilegeEscalation: false: Dit zorgt ervoor dat de user ID niet kan veranderd worden. Een “child-process” kan hierdoor dus ook nooit meer privileges krijgen dan de “Parent-process”. Door deze boolean op “false” te zetten wordt het moeilijker voor een aanvaller om aan “privilege escalation” te doen.

hostNetwork: false: Deze variabele geeft een pod de mogelijkheid om het netwerk van de host te gebruiken. Zo kan deze ook applicaties, die de via de *localhost* werken, zien en deze aanpassen. Het wordt hier op “false” gezet omdat deze functie niet nodig is.

runAsUser: MustRunAsNonRoot: Hier wordt de user ID gecontroleerd waarmee de Pods worden aangemaakt. De waarde “MustRunAsNonRoot” zorgt ervoor dat de containers niet door de root gebruiker kunnen worden aangemaakt. Dit wordt vooral gebruikt om te voorkomen dat iemand die (ongewenst) root privileges heeft verkregen nieuwe (onveilige) containers kan creëren.

SupplementalGroups: 'MustRunAs' verplicht hier het gebruik van een “non-root” groep bij het aanmaken van nieuwe pods.

ReadOnlyRootFilesystem: false laat het aanmaken van containers met een aanpasbare “root filesystem” toe. Door deze variabele op “true” te zetten zou elke nieuwe container aangemaakt worden met een “Read-only root filesystem”.

Nadat de PodSecurityPolicy is opgesteld kan deze met volgend commando op de cluster toegepast worden.

```
$ kubectl create -f securePol.yaml
```

```

1  apiVersion: policy/v1beta1
2  kind: PodSecurityPolicy
3  metadata:
4    name: restricted
5    annotations:
6      seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default,runtime/default'
7      apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
8      seccomp.security.alpha.kubernetes.io/defaultProfileName: 'runtime/default'
9      apparmor.security.beta.kubernetes.io/defaultProfileName: 'runtime/default'
10 spec:
11   privileged: false
12   allowPrivilegeEscalation: false
13   hostNetwork: false
14   hostIPC: false
15   hostPID: false
16   runAsUser:
17     rule: 'MustRunAsNonRoot'
18   seLinux:
19     rule: 'RunAsAny'
20   supplementalGroups:
21     rule: 'MustRunAs'
22     ranges:
23       - min: 1
24         max: 65535
25   readOnlyRootFilesystem: false

```

Figuur 3.16: securePol.yaml

De PodSecurityPolicies die op de cluster actief zijn kunnen met volgend commando opgevraagd worden.

```

$ kubectl get psp
NAME          PRIV  SELINUX  RUNASUSER          SUPGROUP  READONLYROOTFS
restricted    false RunAsAny  MustRunAsNonRoot  MustRunAs false

```

Vervolgens zal er een zeer simpele testimplementatie van RBAC opgesteld worden. In figuur 3.17 is een kleine role gedefinieerd, deze is verantwoordelijk voor **<Uitleg over Roles>**. Het YAML bestand om de roleBinding te implementeren is te zien in figuur 3.18. **<Uitleg over rolebindings>**

De role en rolebinding kunnen op de cluster worden uitgevoerd de volgende “create” commando’s.

```

$ kubectl create -f role-deployment.yaml
role.rbac.authorization.k8s.io/deployment-manager created
$ kubectl create -f rolebinding-deployment.yaml
rolebinding.rbac.authorization.k8s.io/deployment-manager-binding created

```

```
1 kind: Role
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   namespace: default
5   name: deployment-manager
6 rules:
7 - apiGroups: [ "", "extensions", "apps" ]
8   resources: [ "deployments", "pods" ]
9   verbs: [ "*" ]
```

Figuur 3.17: role-deployment.yaml

```
1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1beta1
3 metadata:
4   name: deployment-manager-binding
5   namespace: default
6 subjects:
7 - kind: User
8   name: tester
9   apiGroup: ""
10 roleRef:
11   kind: Role
12   name: deployment-manager
13   apiGroup: ""
```

Figuur 3.18: rolebinding-deployment.yaml

3.4 Scenario 3: Cluster opstelling met beveiligings-tools

In dit laatste scenario zal er gewerkt worden de beveiligings-tools die besproken zijn in sectie 2.6. Project Calico zal niet worden getest aangezien dat deze automatisch wordt gebruik in de LKE en het niet mogelijk is om deze te configureren. Kube-bench en Kube-hunter zullen wel uitgetest worden. De installatie en gebruik van deze tools wordt hieronder gedemonstreerd.

Als eerste is Kube-bench aan de beurt. Zoals beschreven in sectie 2.6.2 zal Kube-bench de node gaan testen tegen de *CIS kubernetes Benchmarks* en een overzicht geven van de conformiteit ervan. Om Kube-bench te installeren is het nodig om een SSH verbinding op te zetten met een node in de cluster. Wanneer de verbinding gemaakt is kunnen de volgende commando's uitgevoerd worden om de installatie te voltooien.

```
$ curl -L https://github.com/aquasecurity/kube-bench/releases/download/v0.3.1/kube-bench_0.3.1_linux_amd64.deb  
-o kube-bench_0.3.1_linux_amd64.deb  
$ sudo apt install ./kube-bench_0.3.1_linux_amd64.deb -f
```

Kube-bench kan nu gebruikt worden om de conformiteit van de cluster te controleren. Dit kan op verschillende manieren door enkele *flags* mee te geven aan de applicatie. Configuratie van kube-bench kan op de volgende manieren gebeuren:

- `$ kube-bench --benchmark cis-1.5` zal er voor zorgen dat de node wordt getoetst op basis van de *CIS benchmark* versie 1.5. Andere opties zijn: `cis-1.6`, `gke-1.0` (Google K8s engine), `eks-1.0` (Elastic K8s engine) en `ack-1.0`. GKE, EKS en ACK staan respectievelijk voor Google Kubernetes Engine, Elastic Kubernetes Engine en AWS Controllers for Kubernetes. Deze drie opties zorgen ervoor dat er checks worden uitgevoerd die specifiek voor die platformen zijn geschreven.
- `$ kube-bench run --targets master,node`: Door dit commando uit te voeren worden enkel de test met betrekking tot de *master-* en *worker nodes* uitgevoerd.

Om het effect van Kube-bench op het resource gebruik aan te tonen zal de cluster getest worden tegen de CIS 1.5 benchmarks. De output van dit commando is te zien in figuur 3.19. Kube-bench geeft hierin een overzicht van welke delen van de cluster wel of niet voldoen aan de benchmarks. Verder geeft het ook nog tips over hoe deze problemen kunnen opgelost worden.

De volgende beveiligings-tools is *Kube-hunter*. Zoals besproken in sectie 2.6.3 kan kube-hunter op drie verschillende manieren worden gebruikt. In dit deel van het scenario zullen alle 3 de delen aan bod komen.

De eerste manier om Kube-hunter te gebruiken is om het node te installeren zodat het alle netwerkinterfaces kan scannen. Ook hier zijn er verschillende manieren om dit te doen, namelijk via het commando `pip3 install kube-hunter` of met behulp van een Docker container. Hier werd voor de tweede optie gekozen. Met volgend commando wordt de Docker container op de node geïnstalleerd en geconfigureerd.

```

1 kube-bench
2 [INFO] 2 Worker Node Security Configuration
3 [INFO] 2.1 Kubelet
4 [FAIL] 2.1.1 Ensure that the --allow-privileged argument is set to false (Scored)
5 [PASS] 2.1.2 Ensure that the --anonymous-auth argument is set to false (Scored)
6 [PASS] 2.1.3 Ensure that the --authorization-mode argument is not set to AlwaysAllow (Scored)
7 [PASS] 2.1.4 Ensure that the --client-ca-file argument is set
8 as appropriate (Scored)
9 [FAIL] 2.1.5 Ensure that the --read-only-port argument is set to 0 (Scored)
10 [PASS] 2.1.6 Ensure that the --streaming-connection-idle-timeout argument is
11 not set to 0 (Scored)
12 [FAIL] 2.1.7 Ensure that the --protect-kernel-defaults argument is set to true (Scored)
13 [PASS] 2.1.8 Ensure that the --make-iptables-util-chains argument is set to true (Scored)

```

Figuur 3.19: Output kube-bench commando

```

$ docker run -it --rm --network host aquasec/kube-hunter
Choose one of the options below:
1. Remote scanning      (scans one or more specific IPs or DNS names)
2. Interface scanning   (scans subnets on all local network interfaces)
3. IP range scanning    (scans a given IP range)
Your choice: 2

```

Dit zal ons een zeer gedetailleerd beeld geven over hoe welke nodes en services er aanwezig zijn in het lokale netwerk van de node. Het geeft ook een overzicht van de, al dan niet, gevonden zwakheden in het netwerk.

De tweede manier om kube-hunter te gebruiken is om het op een machine te installeren die buiten de cluster staat. Deze manier geeft ons een overzicht van hoe de cluster er voor een aanvaller uitziet. Dit kan aan de hand van dezelfde Docker container maar dan met de eerste optie, namelijk het “remote scanning”. De output van dit commando wordt in figuur 3.20 getoond.

De derde, en laatste, manier waarop kube-hunter kan gebruikt worden is door het als pod binnen de cluster te installeren. Dit simuleert wat er zou kunnen gebeuren als er gebruik wordt gemaakt van een gecompromitteerde pod. Dit kan simpelweg door het volgende commando uit te voeren om een “job” op te starten. De job zelf is te zien in figuur 3.22.

```
$ kubectl create -f ./job.yaml
```

Als de job aangemaakt is kunnen we op zoek naar de naam van de pod. Dit kan met volgend commando.

```
$ kubectl describe job kube-hunter
```

De resultaten van de interne scan kunnen opgevraagd worden door naar de logs van de pod te kijken. Figuur 3.21 toont het commando en de bijhorende output.

```

1  docker run -it --rm --network host aquasec/kube-hunter
2  Choose one of the options below:
3  1. Remote scanning      (scans one or more specific IPs or DNS names)
4  2. Interface scanning   (scans subnets on all local network interfaces)
5  3. IP range scanning    (scans a given IP range)
6  Your choice: 1
7  Remotes (separated by a ','): 45.79.249.208
8  2021-05-12 16:15:19,611 INFO kube_hunter.modules.report.collector
9  Started hunting
10 2021-05-12 16:15:19,611 INFO kube_hunter.modules.report.collector
11 Discovering Open Kubernetes Services
12 2021-05-12 16:15:19,688 INFO kube_hunter.modules.report.collector
13 Found open service "Kubelet API" at 45.79.249.208:10250
14
15 Nodes
16 +-----+-----+
17 | TYPE          | LOCATION          |
18 +-----+-----+
19 | Node/Master   | 45.79.249.208    |
20 +-----+-----+
21
22 Detected Services
23 +-----+-----+-----+
24 | SERVICE      | LOCATION          | DESCRIPTION          |
25 +-----+-----+-----+
26 | Kubelet API   | 45.79.249.208:10250 | The Kubelet is the  |
27 |               |                   | main component in   |
28 |               |                   | every Node, all pod |
29 |               |                   | operations goes     |
30 |               |                   | through the kubelet |
31 +-----+-----+-----+
32
33 No vulnerabilities were found

```

Figuur 3.20: Output kube-hunter remote scanning


```

1      $ kubectl logs kube-hunter-zwwsz
2 2021-05-12 16:31:13,364 INFO kube_hunter.modules.report.collector Started hunting
3 2021-05-12 16:31:13,373 INFO kube_hunter.modules.report.collector
4 Discovering Open Kubernetes Services
5 2021-05-12 16:31:13,378 INFO kube_hunter.modules.report.collector
6 Found vulnerability "CAP_NET_RAW Enabled" in Local to Pod (kube-hunter-zwwsz)
7 2021-05-12 16:31:13,378 INFO kube_hunter.modules.report.collector
8 Found vulnerability "Read access to pod's service account token" in Local to Pod (kube-hunter-zwwsz)
9 2021-05-12 16:31:13,378 INFO kube_hunter.modules.report.collector
10 Found vulnerability "Access to pod's secrets" in Local to Pod (kube-hunter-zwwsz)
11 2021-05-12 16:31:36,101 INFO kube_hunter.modules.report.collector
12 Found open service "API Server" at 10.128.0.1:443
13 2021-05-12 16:31:36,157 INFO kube_hunter.modules.report.collector
14 Found vulnerability "K8s Version Disclosure" in 10.128.0.1:443
15 2021-05-12 16:31:36,162 INFO kube_hunter.modules.report.collector
16 Found vulnerability "Access to API using service account token" in 10.128.0.1:443
17
18 Nodes
19 +-----+-----+
20 | TYPE          | LOCATION    |
21 +-----+-----+
22 | Node/Master   | 10.128.0.1  |
23 +-----+-----+

```

Figuur 3.21: Output kube-hunter remote scanning

```

1 ---
2 apiVersion: batch/v1
3 kind: Job
4 metadata:
5   name: kube-hunter
6 spec:
7   template:
8     spec:
9     containers:
10      - name: kube-hunter
11        image: aquasec/kube-hunter
12        command: ["kube-hunter"]
13        args: ["--pod"]
14      restartPolicy: Never
15    backoffLimit: 4

```

Figuur 3.22: Output kube-hunter remote scanning

3.5 Data analyse

In dit hoofdstuk zal de verzamelde data van de drie scenario's verwerkt en geanalyseerd worden. Dit zal gebeuren zoals beschreven in sectie 3.1.3. De drie scenario's zullen vervolgens aan de hand van deze analyse met elkaar vergeleken worden.

3.5.1 Scenario 1

Als eerste zal het gemiddelde gebruik van zowel de CPU als het RAM geheugen verzameld worden. Voor de CPU is dit 10% zoals te zien in figuur 3.23. Het gemiddelde RAM gebruik is af te lezen wanneer we de code uit figuur 3.5 uitvoeren op de dataset. De uitvoer hiervan is 90.13%, zoals in figuur 3.24 te zien is.

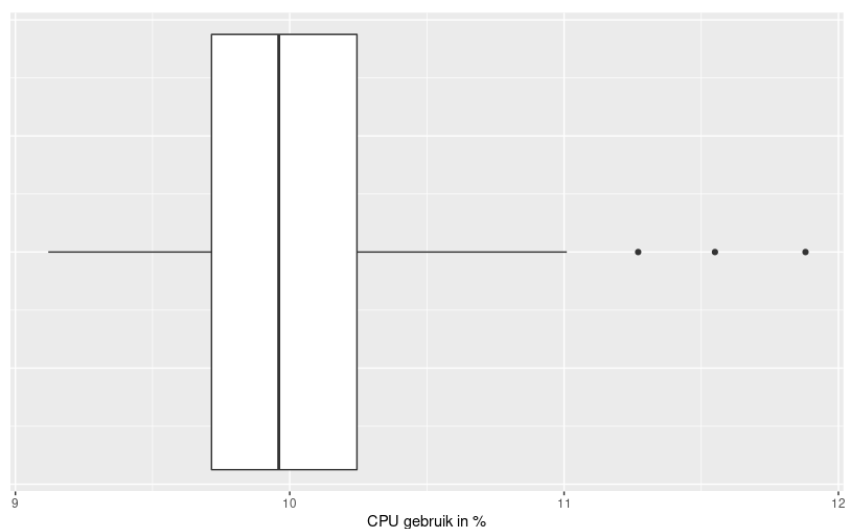
147	Average:	all	6.24	0.00	3.19	0.01	0.56	90.00
-----	----------	-----	------	------	------	------	------	-------

Figuur 3.23: Gemiddeld CPU gebruik (100 - Laatste Kolom).

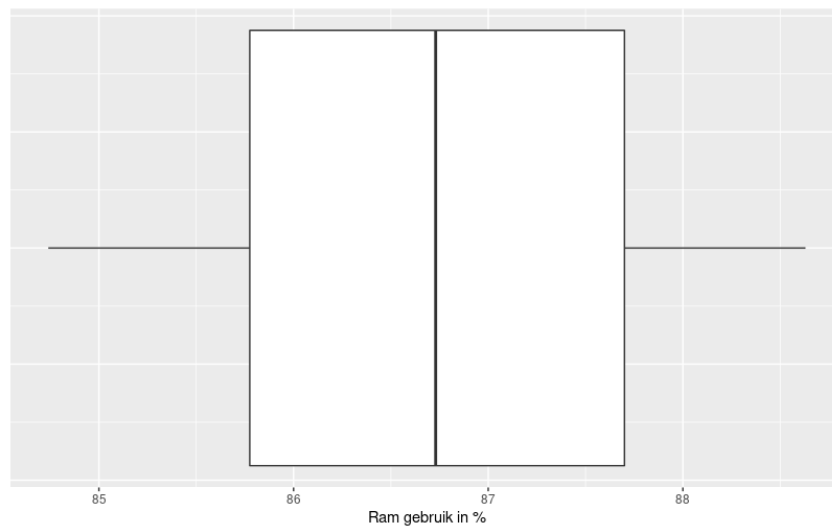
```
> summary(ram)
RAM
Min.    :89.33
1st Qu.:89.75
Median :90.11
Mean    :90.13
3rd Qu.:90.39
Max.    :91.25
```

Figuur 3.24: Gemiddeld RAM gebruik

Ten tweede zullen de gegevens in verband met de stabiliteit van het systeem bekeken worden. In figuur 3.25 en 3.26 zijn de boxplots voor zowel het CPU als RAM gebruik te zien. We kunnen zien dat bij het CPU gebruik er maar 3 *outliers* zijn en bij het RAM gebruik zijn er zelf geen *outliers*. Dit wijst erop dat het RAM gebruik zeer stabiel blijft.

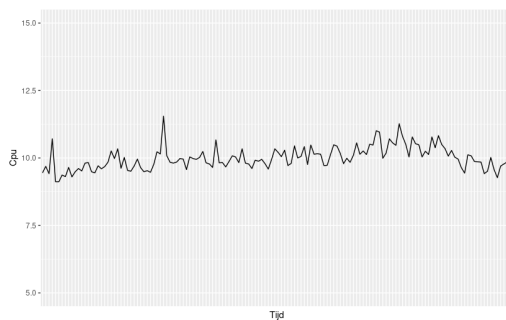


Figuur 3.25: Boxplot van het CPU gebruik in Scenario 1

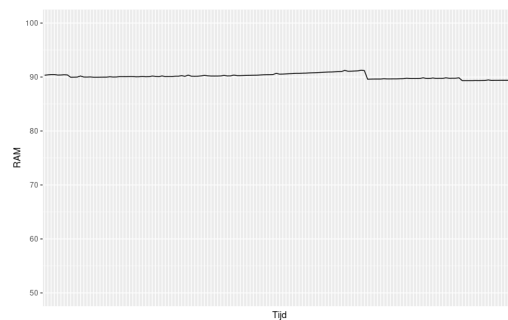


Figuur 3.26: Boxplot van het RAM gebruik in Scenario 1

Als derde criteria worden de gegevens van het algemene CPU en RAM gebruik bekeken. Deze worden gevisualiseerd aan de hand van een lijngrafiek. Het resultaat van scenario 1 is te zien in figuur 3.27 en 3.28.



Figuur 3.27: Lijngrafiek van het CPU gebruik in Scenario 1.



Figuur 3.28: Lijngrafiek van het RAM gebruik in Scenario 1.

Het laatste criteria is de opstarttijd van de node. Deze kan men simpelweg vinden aan de hand van het `systemd-analyze` commando. De uitvoer van dit commando is te zien in figuur 3.29.

```
$ systemd-analyze
Startup finished in 6.121s (kernel) + 3.296s (userspace) = 9.418s
```

Figuur 3.29: Opstarttijd van de Node in Scenario 1

3.5.2 Scenario 2

Als eerste zal het gemiddelde gebruik van zowel de CPU als het RAM geheugen verzameld worden. Voor de CPU is dit 10.15% zoals te zien in figuur 3.30. Het gemiddelde RAM gebruik is af te lezen wanneer we de code uit figuur 3.5 uitvoeren op de dataset. De uitvoer hiervan is 90.75%, zoals in figuur 3.31 te zien is.

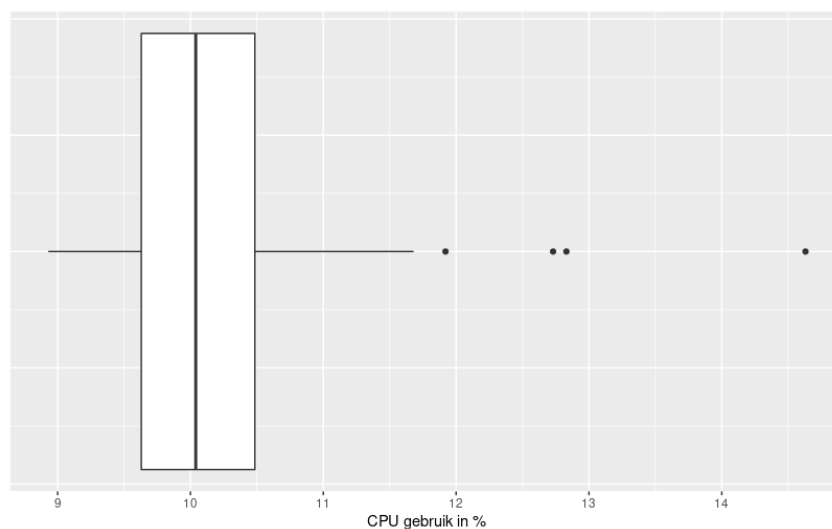
146	Average:	all	6.75	0.00	3.14	0.06	0.20	89.85
-----	----------	-----	------	------	------	------	------	-------

Figuur 3.30: Gemiddeld CPU gebruik (100 - Laatste Kolom).

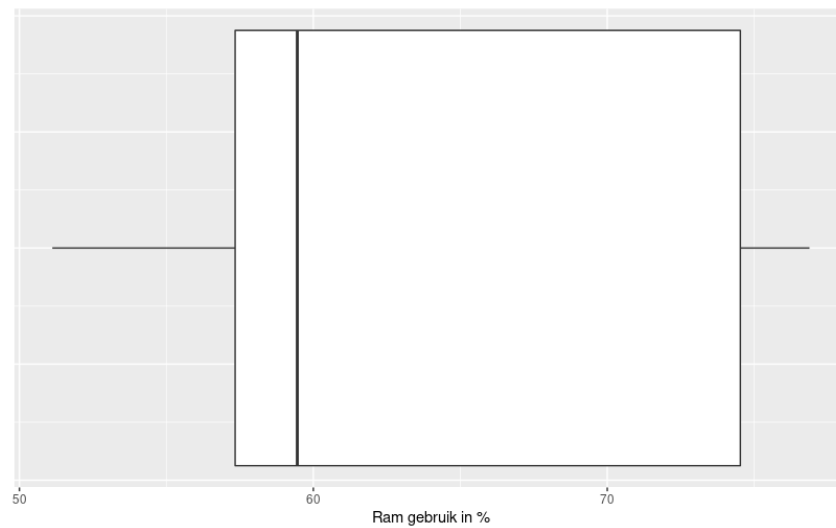
```
> summary(ram)
RAM
Min.    :90.21
1st Qu.:90.60
Median :90.74
Mean    :90.75
3rd Qu.:90.91
Max.    :91.27
```

Figuur 3.31: Gemiddeld RAM gebruik in Scenario 2

Ten tweede zullen de gegevens in verband met de stabiliteit van het systeem bekeken worden. In figuur 3.32 en 3.33 zijn de boxplots voor zowel het CPU als RAM gebruik te zien. We kunnen zien dat bij het CPU gebruik er 4 *outliers* zijn en bij het RAM gebruik zijn er zelf geen *outliers*.

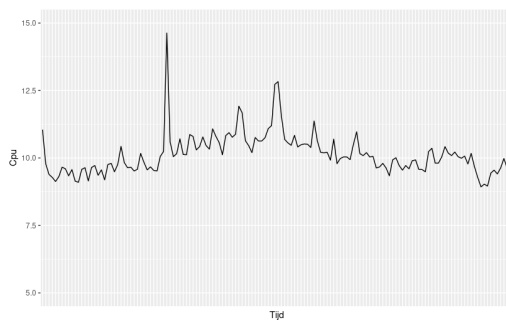


Figuur 3.32: Boxplot van het CPU gebruik in Scenario 2

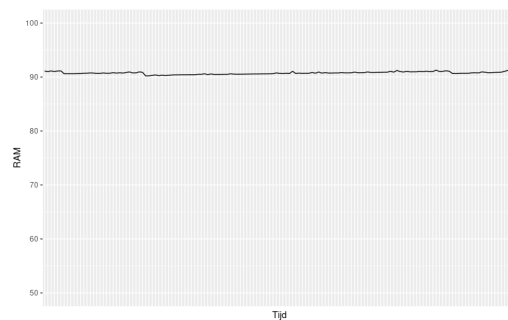


Figuur 3.33: Boxplot van het RAM gebruik in Scenario 2

Als derde criteria worden de gegevens van het algemene CPU en RAM gebruik bekeken. Deze worden gevisualiseerd aan de hand van een lijngrafiek. Het resultaat van scenario 1 is te zien in figuur 3.34 en 3.35.



Figuur 3.34: Lijngrafiek van het CPU gebruik in Scenario 2.



Figuur 3.35: Lijngrafiek van het RAM gebruik in Scenario 2.

Het laatste criteria is de opstarttijd van de node. Deze kan men simpelweg vinden aan de hand van het `systemd-analyze` commando. De uitvoer van dit commando is te zien in figuur 3.36.

```
$ systemd-analyze
Startup finished in 6.246s (kernel) + 2.997s (userspace) = 9.243s
```

Figuur 3.36: Opstarttijd van de Node in Scenario 2

3.5.3 Scenario 3

Als eerste zal het gemiddelde gebruik van zowel de CPU als het RAM geheugen verzameld worden. Voor de CPU is dit 12.3% zoals te zien in figuur 3.37. Het gemiddelde RAM gebruik is af te lezen wanneer we de code uit figuur 3.5 uitvoeren op de dataset. De uitvoer hiervan is 90.13%, zoals in figuur 3.38 te zien is.

146	Average:	all	7.26	0.00	4.08	0.01	0.95	87.70
-----	----------	-----	------	------	------	------	------	-------

Figuur 3.37: Gemiddeld CPU gebruik (100 - Laatste Kolom).

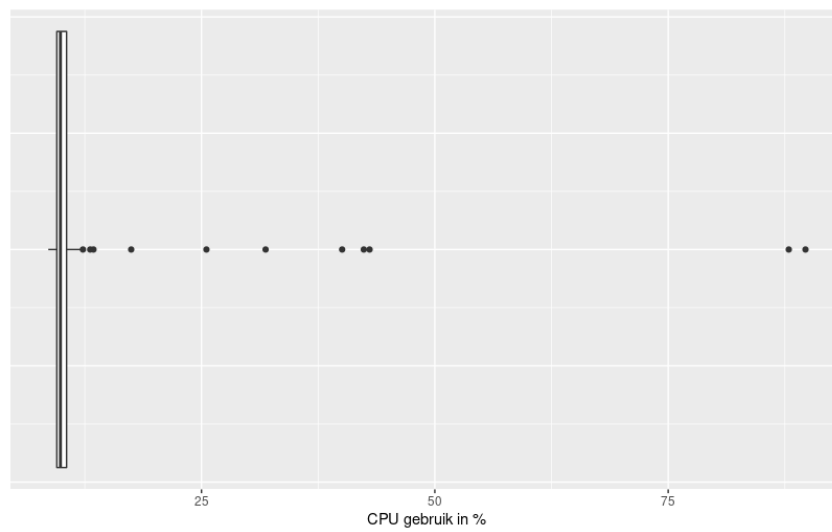
```
> summary(ram)
RAM
Min.    :89.33
1st Qu.:89.75
Median :90.11
Mean    :90.13
3rd Qu.:90.39
Max.    :91.25
```

Figuur 3.38: Gemiddeld RAM gebruik

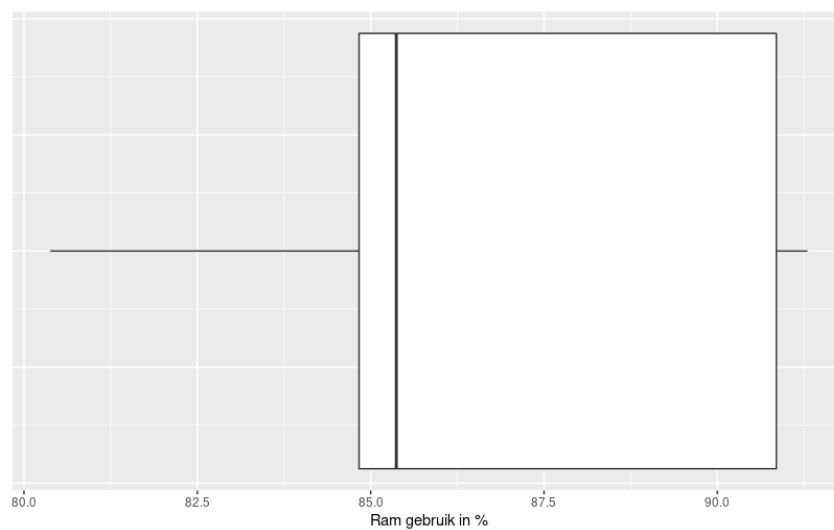
Ten tweede zullen de gegevens in verband met de stabiliteit van het systeem bekeken worden. In figuur 3.39 en 3.40 zijn de boxplots voor zowel het CPU als RAM gebruik te zien. Bij het CPU gebruik zien we 11 outliers en bij het RAM gebruik zijn er geen outliers. Dit wijst er op dat het CPU gebruik minder stabiel blijkt te zijn in vergelijking met de vorige Scenario's.

Als derde criteria worden de gegevens van het algemene CPU en RAM gebruik bekeken. Deze worden gevisualiseerd aan de hand van een lijngrafiek. Het resultaat van scenario 1 is te zien in figuur 3.41 en 3.42. De eerste piek in CPU gebruik is te wijten aan het gebruik van Kube-bench terwijl de tweede en derde piek zijn veroorzaakt door het gebruik van Kube-hunter.

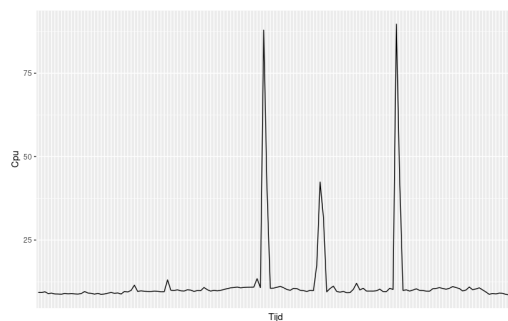
Het laatste criteria is de opstarttijd van de node. Deze kan men simpelweg vinden aan de hand van het `systemd-analyze` commando. De uitvoer van dit commando is te zien in figuur 3.43.



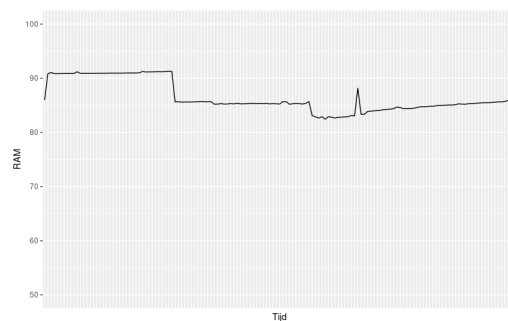
Figuur 3.39: Boxplot van het CPU gebruik in Scenario 3



Figuur 3.40: Boxplot van het RAM gebruik in Scenario 3



Figuur 3.41: Lijngrafiek van het CPU gebruik in Scenario 3.



Figuur 3.42: Lijngrafiek van het RAM gebruik in Scenario 3.


```
$ systemd-analyze  
Startup finished in 6.153s (kernel) + 5.180s (userspace) = 11.334s
```

Figuur 3.43: Opstarttijd van de Node in Scenario 3

3.5.4 Vergelijken van de data

Nu alle gegevens verzameld en verwerkt zijn kunnen we deze met elkaar gaan vergelijken. Als eerste zal het gemiddeld *resource* gebruik van de verschillende scenario's besproken worden. Tabel 3.1 geeft een overzicht van de data. In deze tabel is duidelijk te zien dat zowel de *best practices* als de beveiligings-tools een invloed hebben gehad op het CPU gebruik. De hoeveelheid RAM die gebruikt werd is daarentegen hetzelfde gebleven bij zowel Scenario één als drie. Deze is in scenario twee zelfs gezakt. Ook al werden in scenario drie de tools maar sporadisch gebruikt, toch hebben deze een aanzienlijke impact op het gemiddeld *resource* gebruik van de cluster gehad.

Gemiddeld resource gebruik	Scenario1	Scenario2	Scenario3
CPU	10	10.15	12.3
Ram	90.13	90.75	90.13

Tabel 3.1: Gemiddeld resource gebruik van de scenario's

Het tweede criteria dat onderzocht werd is de stabiliteit van de cluster, dit door het aantal *outliers* te tellen. In tabel 3.2 is te zien hoeveel *outliers* er per scenario werden gedetecteerd. Geen enkel scenario had effect op de stabiliteit van het RAM gebruik maar het CPU gebruik werd wel iets instabieler in scenario drie. Deze grote hoeveelheid *outliers* is te wijten aan het feit dat de beveiligings-tools niet constant in gebruik zijn. De *outliers* komen hier dus overeen met de momenten waarop de tools actief werden gebruikt. Hieruit is te concluderen dat *best practices* een kleine (praktisch verwaarloosbare) impact hebben op de stabiliteit. Alsook dat beveiligings-tools een tijdelijke maar toch aanzienlijk effect hebben op de data.

Aantal outliers	Scenario1	Scenario2	Scenario3
CPU	3	4	11
Ram	0	0	0

Tabel 3.2: Aantal outliers van de scenario's

Het laatste criteria dat werd onderzocht is de opstart tijd van een node in de cluster. Tabel 3.3 toont de hoeveelheid seconden de node in elk scenario nodig had om op te starten. Deze tabel maakt duidelijk dat *best practices* geen (of in dit geval een positief) effect hebben op de opstarttijd van een cluster. De beveiligings-tools aan de andere kant zorgde er voor dat de node twee seconden trager opstarten.

	Scenario1	Scenario2	Scenario3
Opstarttijd	9.418	9.243	11.334

Tabel 3.3: Tijd die de Node nodig had om op te starten

4. Conclusie

Dit onderzoek heeft aangetoond dat er veel verschillende manieren zijn om een Kubernetes cluster te beveiligen. Gaande van het correct volgen van *best practices* tot het gebruik van groot aantal beveiligings-tools. Er is ook gebleken dat het beveiligen van een K8s cluster heel wat werk met zich meebrengt, waardoor sommige organisaties dit als optioneel zien. Uit het onderzoek bleken er twee zeer belangrijke risico's bij het opzetten van een K8s cluster, namelijk menselijke fouten bij het configureren van de cluster en bugs die onvermijdelijk zijn in een project van deze schaal.

De *best practices*, waaronder het gebruik van *Pod Security Policies* en RBAC, blijken zeer krachtige tools bij het beveiligen van een cluster. Tijdens het onderzoek is gebleken dat er zeer veel *best practices* zijn. Deze hadden voornamelijk betrekking tot het correct gebruiken van ingebouwde opties. Het nadeel aan deze *best practices* is dat het zeer complex kan worden in een grote productieomgeving, waardoor het soms achterwege wordt gelaten.

Omdat K8s zo veel gebruikt wordt, zijn er tevens veel bedrijven die hun eigen tools voor het beveiligen van een cluster ontwikkelen. Deze tools zijn meestal, net zoals Kubernetes zelf, open source waardoor iedereen ze kan gebruiken. De twee meest gebruikte en bekendste beveiligings-tools zijn Kube-bench en Kube-hunter. Beiden worden gebruikt om de beveiliging van de cluster te controleren, maar ze doen dit elk op hun eigen manier. Kube-bench controleert de cluster op basis van vooropgestelde benchmarks, dit terwijl Kube-hunter eerder wordt gebruikt als automatische penetratie test.

Uit het onderzoek is ook gebleken dat zowel het toepassen van *best practices* als het gebruik van beveiligings-tools weinig effect heeft op de vooropgestelde criteria. De stijging in resource gebruik was detecteerbaar en verwacht maar in de meeste gevallen was deze

praktisch verwaarloosbaar. Ondanks zijn er nog veel bedrijven die te weinig inzetten op de beveiliging van hun K8s clusters. In toekomstig onderzoek zou er mogelijks verder gekeken kunnen worden naar de achterliggende reden voor deze nalatigheid.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Inleiding en State-of-the-art

A.1.1 Wat zijn containers?

Het uitrollen en schalen van applicaties wordt steeds vaker gedaan met behulp van containers. Tijdens de ontwikkeling van traditionele applicaties wordt de applicatie ontwikkeld in een specifiek testomgeving. Vervolgens wordt de applicatie overgezet naar de productieomgeving wat vaak voor problemen zorgt (bijvoorbeeld van een linux testomgeving naar een Windows productieomgeving). Een container is een pakket waar één enkele applicatie in zit, samen met alle nodige afhankelijkheden (Education, 2019). Dit zorgt ervoor dat deze gemakkelijk en snel van de ene omgeving naar de andere kan overgezet worden. De containers maken gebruik van een 'runtime engine', dit is een laag die verantwoordelijk is voor de communicatie tussen het operating system van de host machine en de containers zelf. De meeste gebruikte 'runtime engine' is de 'Docker Engine'¹. Deze is al sinds 2013 de industriestandaard als het gaat over container software (McCarty, 2018). Naarmate het gebruik van containers steeg, steeg ook de nood naar op manier om deze vanuit één centrale locatie te beheren. Om aan deze vraag te voldoen werden container orkestratie tools, zoals Kubernetes², ontwikkeld. Deze tools helpen bij het opzetten, uitbreiden en verbinden van een grote hoeveelheid containers.

¹<https://docs.docker.com/engine/>

²<https://kubernetes.io/>

A.1.2 Waarom container applicaties?

Container applicaties hebben enkele voordelen tegenover normale applicaties, ze draaien namelijk geïsoleerd van de rest van het systeem. Ze kunnen dus perfect werken zonder afhankelijk te zijn van andere containers. Dit garandeert dat als er één container aangetast is, de rest zonder interruptie kan verderwerken. De containers delen wel verschillende resources van het host systeem, wat de deur opent voor veiligheidsinbreuken tussen containers.

A.1.3 Context voor dit onderzoek

Gartner (Petty, 2019) voorspeld dat tegen 2022 maar liefst 75% van alle internationale organisaties gecontaineriseerde applicaties zullen gebruiken in hun productieomgeving. Dit zowel in lokale datacenters alsook in online cloud omgevingen. Uit een rapport van Tripwire (2019) blijkt dat 94% van bevroagden bezorgd zijn over de veiligheid van hun containers. Uit hetzelfde rapport blijkt ook dat 47% weet dat ze kwetsbare containers gebruiken in hun productieomgeving. Spijtig genoeg werd bij voorgaande onderzoeken, zoals StackRox (2020), het effect van 'security best practices en tools' op opzetsnelheid, benodigde resources en stabiliteit steeds onderbelicht.

A.1.4 Verloop van het onderzoek

In deze paper zal ik onderzoeken wat de belangrijkste bronnen van veiligheidsinbreuken zijn en hoe deze vermeden kunnen worden. Tegelijkertijd krijg ik via dit onderzoek de opportuniteit om er achter te komen of er vooral technische problemen of menselijke fouten aan de basis liggen van de veiligheidsrisico's. In de volgende paragraaf staat er beschreven hoe ik te werk zal gaan.

A.2 Methodologie

Voor dit onderzoek zullen er drie scenario's opgezet worden. Elk scenario zal verschillende keren worden uitgevoerd en voor elk criteria zal het genomen worden (eventueel rekening houdende met uitschieters). Bij elke scenario zullen er verschillende 'security best practices en tools' gebruikt worden. Deze zullen getest worden op basis van de volgende criteria:

- Deployment snelheid
- Benodigde resources
- Stabiliteit

Voorbeelden van scenario's:

- S(0): Er wordt een container applicatie opgezet in een Kubernetes cluster zonder extra security configuratie.
- S(1): Er wordt een container applicatie opgezet in een Kubernetes cluster en enkele 'best practices' worden toegepast.
- S(2): Er wordt een container applicatie opgezet in een Kubernetes cluster waar er gebruik word gemaakt van enkele 'security tools' zoals 'Project Calico' en 'Kube-hunter'.

Door gebruik te maken van deze scenario's en criteria hopen we vast te stellen dat het toepassen van 'security best practices en tools' een positieve invloed heeft op het gebruik van containers en orkestratie tools.

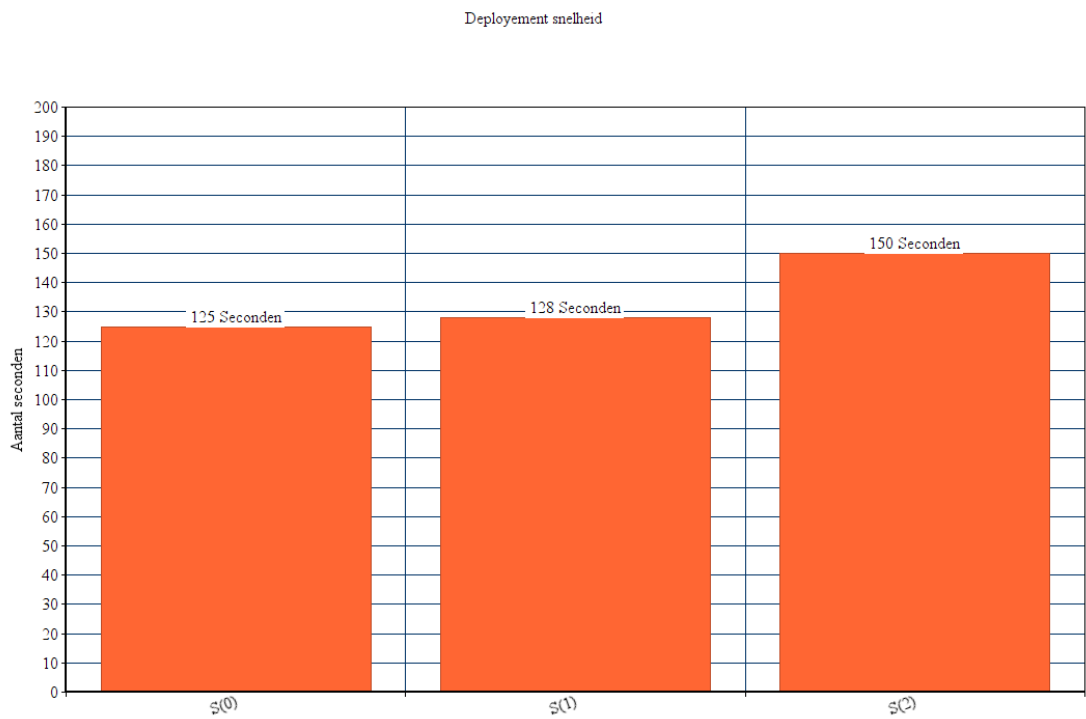
A.3 Verwachte resultaten

Op basis van de criteria wordt er verwacht dat Scenario 0 en 1 even snel op opgezet kunnen worden en evenveel resources gebruiken. Ze zullen beide kwetsbaarder zijn aangezien de 'best practices' vooral bestaan uit het correct gebruik van wachtwoorden en gebruiker privileges. Scenario 2 daarentegen zal iets meer tijd nodig hebben om opgezet te worden (zie Figuur1) en zal daarbij meer resources gebruiken (zie Figuur2). Dit zou te wijten zijn aan de gebruikte 'security tools' die extra tijd en resource nodig hebben.

A.4 Verwachte conclusies

Uit dit onderzoek willen we concluderen dat het toepassen van 'best practices' en het correct gebruik van security tools een positief effect teweeg brengt bij het gebruik van container orkestratie tools. We trachten daarnaast ook aan te duiden dat het omzeilen van security risico's een belangrijk aspect is bij het ontwikkelen van container applicaties. Tot slot kan er geconcludeerd worden dat het beveiligen van container clusters steeds belangrijker wordt. Daarnaast is het tevens van belang dat de persoon die een cluster opzet daarbij de onderliggende werkwijze goed kent en zich bewust is van de mogelijke valkuilen.

A.5 Bijlagen



Figuur A.1: Verwachte opstart tijd

Benodigde resources	CPU %	<u>Memory %</u>
S(0)	45%	30%
S(1)	46%	32%
S(2)	55%	45%

Figuur A.2: Verwacht resource gebruik

Bibliografie

- Anil, N., Coulter, D., Victor, Y., Parente, J., Warren, G. & Wenzel, M. (2018, augustus 31). *What is Docker?* Verkregen 9 maart 2021, van <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined>
- AquaSecurity. (2021). *Docker Architecture*. <https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/>
- Armstrong, J. (2021, januari 6). *A 2020 Review of the World's Most Popular Kubernetes CNI*. <https://www.projectcalico.org/a-2020-review-of-the-worlds-most-popular-kubernetes-cni/>
- Bannister, T., Katz, R., Sychevskiy, I. & Corleissen, Z. (2021, april 1). *Network Policies*. <https://kubernetes.io/docs/concepts/services-networking/network-policies/>
- CNCF. (2021, maart 16). *CNCF Cloud Native Interactive Landscape*. <https://landscape.cncf.io/>
- DevopsCube. (2021, maart 1). *List of Best Docker Container Orchestration Tools/Services*. <https://devopscube.com/docker-container-clustering-tools/>
- Docker. (2021a, maart 5). *Docker overview*. Verkregen 9 maart 2021, van <https://docs.docker.com/get-started/overview/>
- Docker. (2021b, maart 3). *How Docker Helps Development Teams*. <https://www.docker.com/use-cases>
- Education, I. C. (2019, mei 25). *Containerization*. <https://www.ibm.com/cloud/learn/containerization#toc-what-is-co-r25Smlqq>
- Education, I. C. (2020, september 2). *Containers vs. VMs: What's the Difference?* Verkregen 8 maart 2021, van <https://www.ibm.com/cloud/blog/containers-vs-vms>
- Google. (2016). *Containers at Google: A better way to develop and deploy applications*. <https://cloud.google.com/containers>

- Islam Shamim, M. S., Ahamed Bhuiyan, F. & Rahman, A. (2020). XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. *2020 IEEE Secure Development (SecDev)*, 58–64. <https://doi.org/10.1109/SecDev45635.2020.00025>
- Kreisa, J. (2020, juli 30). *Docker Index: Dramatic Growth in Docker Usage Affirms the Continued Rising Power of Developers*. Verkregen 9 maart 2021, van <https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/>
- Kubernetes. (2021, februari 1). *What is Kubernetes?* <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>
- Lewis, I. Kubernetes Security Best Practices. Engles. In: KubeCon + CloudNativeCon (Kubernetes Forum Seoul, 10 december 2019). Seoul: Cloud Native Computing Foundation, 2019, december 10. Verkregen 29 maart 2021, van <https://www.youtube.com/watch?v=wqsUfvRyYpw>
- McCarty, S. (2018, februari 22). *A Practical Introduction to Container Terminology*. <https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/#h.6yt1ex5wfo3l>
- Pedersen, B. E., Tokuda, T., Nakamura, H., Yi, J. & Wang, S. (2021, januari 3). *Kubernetes Components*. <https://kubernetes.io/docs/concepts/overview/components/>
- Pettey, C. (2019, april 23). *6 Best Practices for Creating a Container Platform Strategy*. <https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/>
- RedHat. (2021a, maart 24). *Introduction to Kubernetes architecture*. Verkregen 24 maart 2021, van <https://www.redhat.com/en/topics/containers/kubernetes-architecture>
- RedHat. (2021b, maart 16). *What is container orchestration?* <https://www.redhat.com/en/topics/containers/what-is-container-orchestration>
- Rice, L. The State of Kubernetes Security. Engles. In: Paris Container Day. Paris Container Day. Parijs, 2019, juni 12. Verkregen 29 maart 2021, van https://www.youtube.com/watch?v=_l56oUxHSio
- Rice, L., Rojas, R., Huang, H. & Rotem, Y. (2021, februari 23). *kube-bench*. <https://github.com/aquasecurity/kube-bench>
- StackRox. (2020, februari 19). *State of Container and Kubernetes Security* (onderzoeksrap.). StackRox. <https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/>
- Taylor, T. (2019, maart 25). *Top 7 Kubernetes security tools to harden your container stack*. Verkregen 29 maart 2021, van <https://techgenix.com/kubernetes-security-tools/>
- Tigera. (2021, maart 28). *What is Calico?* <https://www.projectcalico.org/>
- Tripwire. (2019, januari 1). *Tripwire State of Container Security Report* (onderzoeksrap.). Tripwire. Verkregen 30 november 2020, van <https://3b6xlt3iddqmuq5vy2w0s5d3-wpengine.netdna-ssl.com/state-of-security/wp-content/uploads/sites/3/Tripwire-Dimensional-Research-State-of-Container-Security-Report.pdf>
- Venugopal, V. (2019, april 5). *Guide to Kubernetes Ingress Network Policies*. <https://www.openshift.com/blog/guide-to-kubernetes-ingress-network-policies>
- VMWare. (2021, februari 8). *Hypervisor*. <https://www.vmware.com/topics/glossary/content/hypervisor>