Security Best-Practices in Kubernetes und Evaluation eines geeigneten Security-Scanners

BACHELORARBEIT

für die Prüfung zum

Bachelor of Science

des Studienganges Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Sebastian Schwegler

19.08.2019

Bearbeitungszeitraum:	10. Juni 2019 bis 19. August 2019
Matrikelnummer:	7311201
Kurs:	TINF16B5
Ausbildungsfirma:	FIRMA
Betreuer der Ausbildungsfirma:	
Gutachter der Studienakademie:	

Erklärung

(gemäß §5(3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. 9. 2015)

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema:

Security Best-Practices in Kubernetes und Evaluation eines geeigneten Security-Scanners

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

, 19.08.2019

_____ _____

Ort, Datum Sebastian Schwegler

Sperrvermerk

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anders lautende Genehmigung der Ausbildungsstätte vorliegt.

, 19.08.2019

_____ _____

Ort, Datum Sebastian Schwegler

, 19.08.2019

_____ _____

Ort, Datum

Vorwort

Diese Bachelorarbeit entstand im Rahmen meines dualen Studiums im Bereich der angewandten Informatik bei der FIRMA Services International GmbH.
Ich möchte allen Personen danken, die mich beim Verfassen dieser Arbeit unterstützt haben.

Als erstes möchte ich mich bei Herrn X und Herrn X für die hervorragende Betreuung bedanken, sie fanden immer Zeit sich ausführlich mit der Bachelorarbeit zu befassen.

Mein Dank gilt aber auch dem Team X, allen voran X, der mir bei fachlichen Fragen immer mit Rat und Tat zur Seite stand.

Schlussendlich möchte ich mich bei allen anderen bedanken, die zum Gelingen dieser Arbeit beigetragen haben.

Abstract

Using Kubernetes is almost mandatory in 2019, leading to FIRMA using Kubernetes. Since there is no knowledge about security in Kubernetes components in FIRMAs IT-Security team, the task of this bachelor thesis was to create a guide for teams to secure their Kubernetes clusters and to provide a detailed knowledge of how Kubernetes components work and interact with each other.

It was also asked to evaluate a suitable security scanner for container images, as well as for Docker and Kubernetes infrastructure components.

Finally, the costs of a local provision of Kubernetes were compared to the costs of using Google Kubernetes Engine and Azure Kubernetes Service, resulting in a local provision being significantly cheaper, even if compared to discount up to 71% of cloud providers.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ABAC *Attribute Based Access Control*
API *Application Progamming Interface*
BSI *Bundesamt für Sicherheit in der Informationstechnik*
CA *Certificate Authority*
CI/CD *Continous Integration/Continous Delivery*
CIDR *Classless inter-domain-routing*
CLI *Command Line Interface*
CN *Common Name*
CNI *Container Network Interface*
CRD *Custom ressource definiton*
CSR *Certificate Signing Request*
DNS *Domain Name System*
HTTP *Hypertext Transfer Protocol*
IP *Internet Protocol*
JWT *JSON Web Token*
LDAP *Lightweight Directory Access Protocol*
mTLS *mutual TLS*
NAT *Network Address Translation*
PSP *Pod Security Policies*
RBAC *Role Based Access Control*
SAN *Subject Alternative Name*
SSO *Single-Sign-On*
TLS *Transport Layer Security*
VM *virtuelle Maschine*
WAF *Web Application Firewall*

Glossar

B

Bridge
: Netzgerät, welches Geräte auf Layer zwei des OSI-Modells verbindet

C

CI/CD
: Sammlung von Techniken, der das Zusammenführen von Teilen einer Anwendung (CI) sowie deren Auslieferung (CD) beschreibt

D

DevOps
: Ansatz aus Softwareentwicklung und Systemadministration zur besseren Zusammenarbeit und Prozessverbesserung

E

Eureka
: Eine Micorservice-Registry, die das Auffinden von Services in Mircoservice-Umgebungen ermöglicht

G

GitHub
: Website zu Quellcodeverwaltung

H

HTTP-Basic-Authentication
: Authentifizierungsmethode im HTTP-Protokoll nach RFC 2617

I

Ingress-Controller
: Pods, die in Kubernetes gestartet und genutzt werden können, um Routingentscheidungen auf Layer sieben treffen zu können

IP-Tables
: Regeln für die Behandlung von IP-Paketen

L

Loadbalancer
: System, das Anfragen mit dem Ziel der Lastverteilung weiterleitet

N

NAT
: Network Address Translation, bezeichnet den Prozess der Veränderung der IP-Adressen im Header von IP-Paketen, sodass z.B. eine öffentliche IP-Adresse durch mehrere Clients genutzt werden kann

P

Path Traversal
: Ausbrechen aus Verzeichnisstrukturen durch manipulierte URLs mit ../ etc.

R

Rolling Update
: *Update von Komponenten ohne Ausfallzeit*

RSA
: Asymetrischer Kryptografie-Algorithmus

S

Scheduler
: Steuerprogramm, welches die Ausführung von Prozessen (im Kubernetes-Umfeld Pods) plant und durchführt

SSO
: Single-Sign-On, ein Nutzer kann nach einmaliger Anmeldung auf alle Dienste zugreifen, für die er berechtigt ist, ohne sich an jedem Dienst einzeln anmelden zu müssen

T

TLS Termination
: Beenden der TLS-Verschlüsselung, um Datenpakete lesen zu können

W

Web Application-Firewall
: Soll Anwendungen vor Angriffen über das Hypertext Transfer Protocol (HTTP) schützen

1. Einleitung

Die FIRMA Services International GmbH betreibt für die Handelspartner der FIRMA Gruppe sowie deren Mitarbeiter zahlreiche Anwendungen, welche über eine zentrale Plattform bereitgestellt werden. Diese Form der Bereitstellung erfordert die Beteiligung eines Plattformteams, welches diese Plattform betreut und neue Softwareversionen der einzelnen Teams auf diese Plattform ausrollt. Diese Art der Bereitstellung ist zum einen sehr kostenintensiv, da sowohl Kosten durch erhöhten Personalaufwand, als auch durch die längere Änderungszeit von Software (durch höhere Durchlaufzeit) entstehen, zum anderen ist diese Art der Bereitstellung nicht mehr zeitgemäß und soll daher in naher Zukunft durch modernere Formen abgelöst werden. Diese neuen Formen sollen sich in die Agilität der einzelnen Teams einfügen, sodass Software in Zukunft auf Kubernetes-Clustern bereitgestellt werden soll.

Kubernetes hat in den letzten Jahren viele Funktionalitäten dazugewonnen, was sich auch im Octoverse-Bericht aus dem Jahr 2018 von GitHub zeigt:

Dort belegt Kubernetes mit circa 6500 Beitragenden den achten Platz und muss sich nur Programmiersprachen und Software wie Visual Studio Code, Angular CLI, Tensorflow sowie Angular und React geschlagen geben. (1)

Um in der Welt der Containertechnologien auf dem aktuellen Stand zu sein, ist es also unerlässlich, sich mit Kubernetes zu beschäftigen und dieses auch zu nutzen.

Um die Schutzziele der IT-Sicherheit zu wahren, ist es notwendig, einen Leitfaden zur Absicherung dieser Cluster zu erstellen, sowie Möglichkeiten aufzuzeigen, wie diese in den DevOps-Prozess integriert werden können. Anhand dieses Leitfadens können einzelne Teams ihre Cluster sicher halten.

2. Grundlagen

Dieses Kapitel erläutert alle Grundlagen, um die Inhalte in den folgenden Kapiteln nachvollziehen zu können.

2.1 Kubernetes

Zunächst sollen die Grundlagen von Kubernetes näher erläutert werden. Hierfür gilt es zunächst zu verstehen, was Kubernetes ist:

"Kubernetes ist eine portable, erweiterbare Open-Source-Plattform zur Verwaltung von containerisierten Arbeitslasten und Services, die sowohl die deklarative Konfiguration als auch die Automatisierung erleichtert. Es hat ein großes, schnell wachsendes Ökosystem. Kubernetes Dienstleistungen, Support und Tools sind weit verbreitet." (2)

Das bedeutet aber auch, dass Kubernetes kein klassisches Plattform-as-a-Service-System ist, da Kubernetes nicht auf der Hardwareebene arbeitet. Vielmehr stellt Kubernetes eine Software-as-a-Service Lösung zur Orchestrierung von Containern bereit. Zusätzlich ist es erweiterbar und zielt auf die Unterstützung von möglichst vielen Anwendungstypen ab. (2)

Kubernetes nutzt verschiedene Komponenten, welche für den Betrieb notwendig sind. Diese sind in der folgenden Abbildung 1 für ein klassisches Kubernetes-Cluster schematisch dargestellt und werden nun näher erläutert.

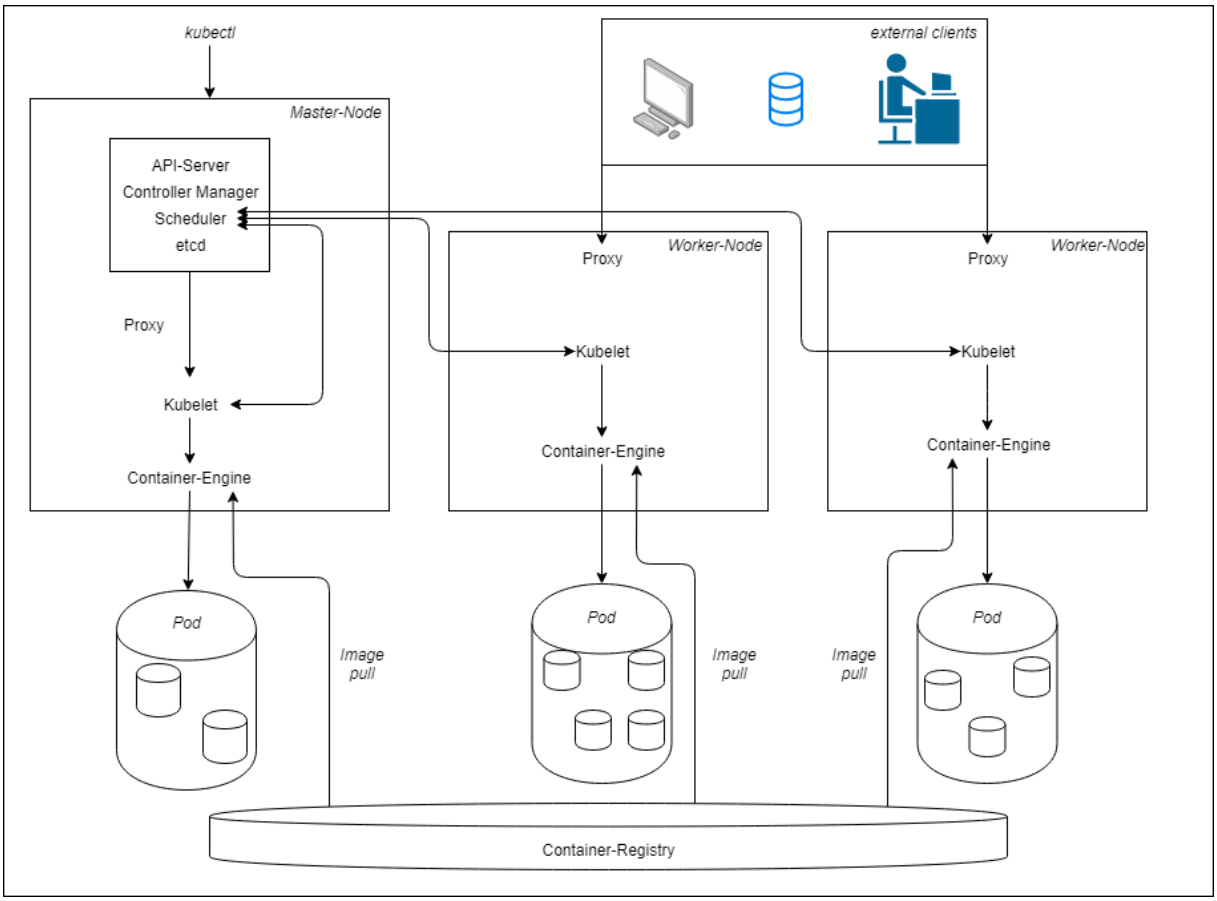


Abbildung 1 Schematischer Überblick über die Kubernetes-Komponenten in einem klassischen Kubernetes-Cluster

In Cloud-Umgebungen sieht diese Architektur in der Regel etwas verändert aus. Hier kommen im Wesentlichen zwei Komponenten hinzu (siehe Abbildung 2): Ein externer Loadbalancer sowie ein Ingress-Controller im Cluster:

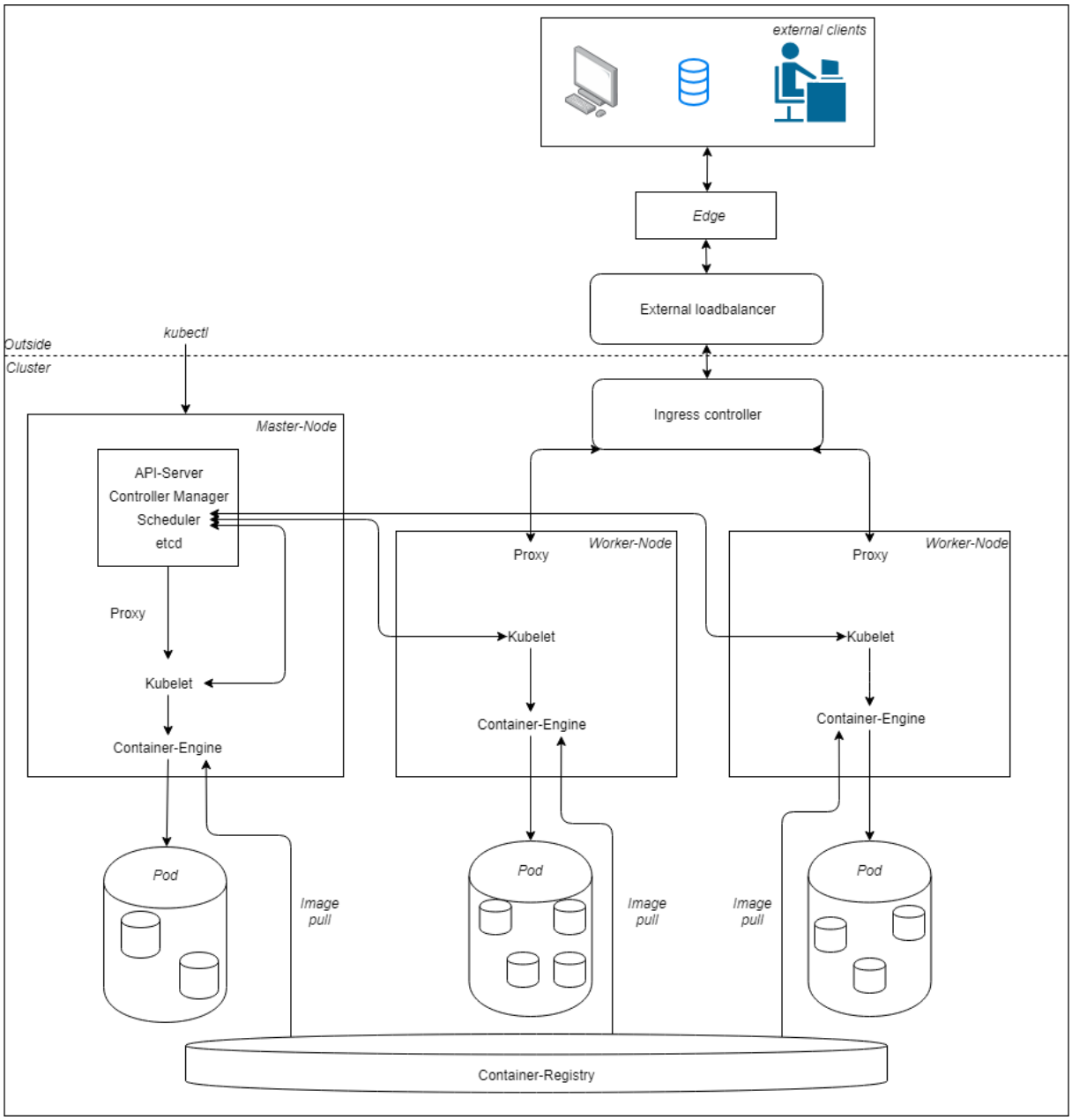


Abbildung 2 Schematischer Überblick über die Kubernetes-Komponenten in der Azure-Cloud

Hierbei ist es wichtig zu erwähnen, dass der externe Loadbalancer keine Funktionalität des Kubernetes-Clusters ist. Dieser muss vom Cloud-Provider bereitgestellt werden. Ein weiteres Feature in der FIRMA-Cloud-Umgebung ist der bereits erwähnte *Edge*. Dieser beinhaltet folgende Komponenten:

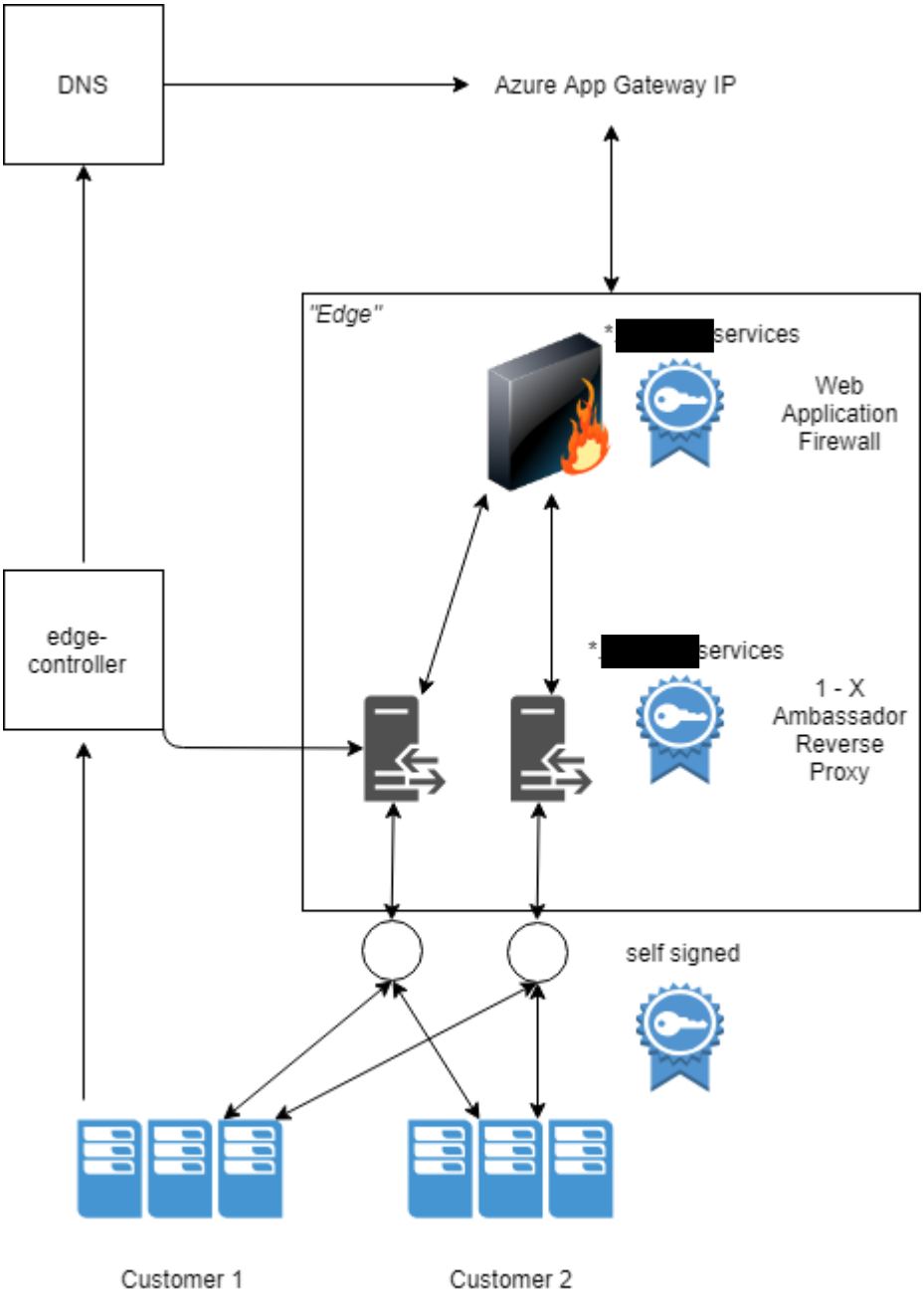


Abbildung 3 Aufbau und Komponenten des Edge

Der Edge besteht, wie in Abbildung 3 zu sehen, aus einer Web Application Firewall, welche eine TLS Termination durchführt und ein Zertifikat für **.Firma.services* nutzt. Darauf folgen mehrere Ambassador-Reverse-Proxies, welche das Routing der Anfragen zu den korrekten Clustern übernehmen. Dort übernehmen dann die Ingress-Controller der jeweiligen Cluster. Der Verkehr zwischen Ambassador und WAF ist ebenfalls mit TLS und dem FIRMA-Zertifikat verschlüsselt. Zusätzlich gibt es die Komponente *edge-controller*, welche zwei Aufgaben hat: DNS-Einträge aus den Clustern extrahieren und entsprechende Einträge im Azure-DNS erzeugen, sowie die

Konfiguration der Ambassador-Proxies, damit diese immer auf die korrekten Cluster verweisen. Das Extrahieren der DNS-Einträge geschieht durch das Auslesen der Host-Parameter aller Ingress-Ressourcen im Cluster. Die DNS-Einträge werden im Azure-DNS erzeugt und verweisen auf die Azure-App-Gateway-IP-Adresse, die Domain *Firma.services* wird in Azure verwaltet.
Dieses Konstrukt wird im weiteren als *Edge* bezeichnet, um die Komplexität zu reduzieren.

2.1.1 Master-Nodes

Master-Nodes bilden die Steuerungsebene des Clusters und treffen globale Entscheidungen über dieses. Ein Master-Node muss in jedem Fall hochverfügbar (3 S. 563) sein und stellt in der Regel folgende Dienste bereit: API-Server, Scheduler, Controller Manager sowie eine Instanz der *etcd*. (3 S. 564)
Der API-Server (manchmal auch *kube-apiserver* genannt) ist die zentrale Stelle zur Steuerung und Orchestrierung des Clusters, mit ihm können alle Ressourcen des Clusters bearbeitet werden. Der API-Server steht als Service bereit (3 S. 564), kann über eine REST-Schnittstelle angesprochen (4) werden und validiert Anfragen, die an ihn gestellt werden. (3 S. 565)
Es ist möglich, eigene beziehungsweise Inhalte von Dritten in diese Validierung einzubringen, dies geschieht über *Custom Ressources Definitions* (*CRD*) (3 S. 565)
Der API-Server kann Cluster-Ressourcen anlegen, modifizieren, lesen und entfernen. Zu Cluster-Ressourcen zählen unter anderem Pods, Jobs, Services, Deployments, Replication Controller und Replication Sets. (3 S. 565)
Der API-Server überwacht aber auch den Zustand des Clusters und startet entsprechende Aktionen. Hierfür überwacht er den gewünschten Zustand des Clusters und speichert diesen Zustand im *etcd*-Store ab. Unter Zuhilfenahme des Controller-Managers und des Schedulers sorgt der API-Server dafür, dass der Wunschzustand im Cluster immer eingehalten wird. (3 S. 565)
Der Controller-Manager bündelt verschiedene Controller:

- Node Controller: Überwacht Nodes auf Ausfall und leitet entsprechende Aktionen ein.
- Replication Controller: Überwacht die gewünschte Anzahl an Pods und leitet entsprechende Maßnahmen ein.

- Endpoints Controller: Befüllt die Endpunkte (Services und Pods).
- Service-Account- und Token-Account-Controller: Erstellt Standardaccounts und API-Token für neue Namespaces.

Es sei angemerkt, dass diese vier Controller-Typen technisch als eigene Prozesse realisiert sind, der Einfachheit wegen aber als eine Binärdatei ausgeliefert werden. (2 S. https://kubernetes.io/docs/concepts/overview/components/)

2.1.2 Worker-Nodes und Pods

Worker-Nodes sind die Bestandteile eines Kubernetes-Cluster, auf welchen die konkrete „Arbeit“ verrichtet wird. Das bedeutet, dass dort die Container laufen. Diese laufen in sogenannten **Pods**. (3 S. 564)

Jeder Node, der als Worker-Node agiert, stellt üblicherweise drei Dienste bereit:

1. Die Container-Engine, in der die konkreten Container ausgeführt werden.
2. Die Kubelet-Instanz, welche die Container-Engine steuert.
3. Eine kube-proxy-Instanz.

Pods stellen die kleinste ausrollbare Einheit innerhalb eines Kubernetes-Clusters dar. Diese können aus mehreren Containern bestehen, es lässt sich somit als eine Art logische Zusammenfassung für Container-Instanzen beschreiben. Kubernetes injiziert in jeden Pod einen sogenannten Pause-Container, der für das Management des Netzinterfaces zuständig ist. Somit besteht ein Pod aus einem oder mehreren explizit angegebenen Containern, implizit aber aus zwei oder mehr Containern.

Ein Pod arbeitet nie Node-übergreifend, sondern ist immer auf einem Node als Entität fixiert. (3 S. 788)

Sofern innerhalb eines Pods mehrere Container laufen, teilen diese sich die gleichen Kernel-Namespaces wie IPC (Interprocess Communication), NET (Network namespaces), MNT (Gemountete Laufwerke sowie PID (Process ID). Dadurch ist eine effiziente Kommunikation zwischen den Containern in einem Pod möglich. (3 S. 788)

So sieht beispielsweise ein Container die Prozesse eines anderen Containers im selben Pod, Container im Pod können durch IPC direkt über systemeigene Funktionen miteinander kommunizieren (SystemV-IPC oder -POSIX-Message-Queues) und alle Container innerhalb eines Pods sehen die gleichen eingebundenen Laufwerke. (3 S. 790-791)

Vor allem IPC könnte den Netzwerkverkehr auf einem Node im Vergleich zu einfachen Containerlösungen wie Docker deutlich reduzieren, da dort die einzelnen Container über das Netzwerk miteinander kommunizieren müssen, selbst wenn sie auf derselben Maschine ausgeführt werden. (3 S. 791)
Diese Möglichkeiten werden aber in der Praxis selten genutzt, da das Aufbauen einer Netzverbindung in Programmiersprachen deutlich einfacher (In Java müsste man über ein Native Interface auf die genannten Ressourcen zugreifen oder eine API einbinden, während das Aufbauen von Netzverbindungen nativ möglich ist) ist, als das Nutzen der oben beschriebenen Komponenten.
Pods ermöglichen es in Kubernetes, funktional zusammenhängende Container auf einem Node als Microservice zu betreiben, aber sie dennoch zu gruppieren.

2.1.3 Replica Sets

Pods sind nicht dazu angedacht, alleine zu laufen, sondern immer in mehreren replizierten Instanzen bereitgestellt zu werden. Defekte Pods müssen folglich automatisch durch intakte ersetzt werden. Dies Aufgabe übernehmen *ReplicaSets*. Diese können, bei Bedarf, auch Pods entfernen, falls dies durch eine Änderung gewünscht ist.
(2 S. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/)
ReplicaSets bieten auch die Möglichkeit, sogenannte *rolling updates* zu ermöglichen. (3 S. 868)
Ein ReplicaSet identifiziert Pods anhand ihres Selektors.
(2 S. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/)

2.1.4 Deployments

Eine komfortablere Möglichkeit, um Pods zu verwalten, bieten sogenannte Deployments. Ein Deployment beschreibt einen gewünschten Zustand. Der Deployment Controller ändert den Zustand des Clusters immer auf den erwünschten Zustand, der im Deployment beschrieben ist.
(2 S. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
Deployments beinhalten ebenfalls eine gewünschte Anzahl von gleichzeitig laufenden Pods, allerdings wird das zugehörige *ReplicaSet* automatisch erzeugt und sollte nicht

vom Nutzer verändert werden.
(2 S. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)

2.1.5 Labels und Selektoren

Labels sind in Kubernetes in beinahe jeder Ressource vorzufinden und ermöglichen es dem Nutzer, eigene Strukturen mit loser Kopplung auf Kubernetes-Ressourcen zu übertragen. (2 S. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)
Labels sind als *Key-Value-Pair* gestaltet und werden durch einen Doppelpunkt getrennt. Labels dürfen nicht mehr als 63 Zeichen lang sein und müssen aus alphanumerischen Zeichen bestehen. Bindestriche, Unterstriche und Punkte dürfen nur zwischen dem Anfangs- und Endzeichen vorkommen.
(2 S. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)
Es sind mehrere Key-Value-Pairs möglich, ein Beispiel für mehrere Labels wäre:

environment: development
tier: backend
partition: database

Über Selektoren kann eine Menge von Ressourcen ausgewählt werden, die bestimmte Label enthält. Achtung: Ein leerer Selektor (*{ }*) selektiert **alle** Ressourcen! Regeln in Policies sind **keine** Selektoren. Dort bedeutet die leere Menge (diese wird dann durch [ ] repräsentiert) auch wirklich leere Menge, also nichts ist erlaubt.
Aus sicherheitstechnischer Sicht können Labels als Grundstein der Sicherheit in Kubernetes bezeichnet werden, denn in den meisten sicherheitstechnischen Einrichtungen in Kubernetes dienen Label zur Identifikation von Ressourcen.

2.1.6 Services und Ingress

Um in Kubernetes Dienste verfügbar zu machen, sind Services notwendig. Dies folgt aus der Tatsache, dass die IP-Adressen von Pods volatil sind und somit nicht für eine Kommunikation genutzt werden können, weiterhin wäre so kein Loadbalancing möglich, da immer derselbe Pod kontaktiert werden würde. Die Verantwortung für das Verteilen der Last würde ausschließlich beim Aufrufer liegen. Kubernetes kennt drei Service-Typen:

1. ClusterIP
2. NodePort
3. Loadbalancer

Diese sollen nun näher erläutert werden. Die einfachste Form der Bereitstellung ist ClusterIP. So sind Services nur innerhalb des Kubernetes-Clusters erreichbar, wie Abbildung 4 veranschaulichen soll:

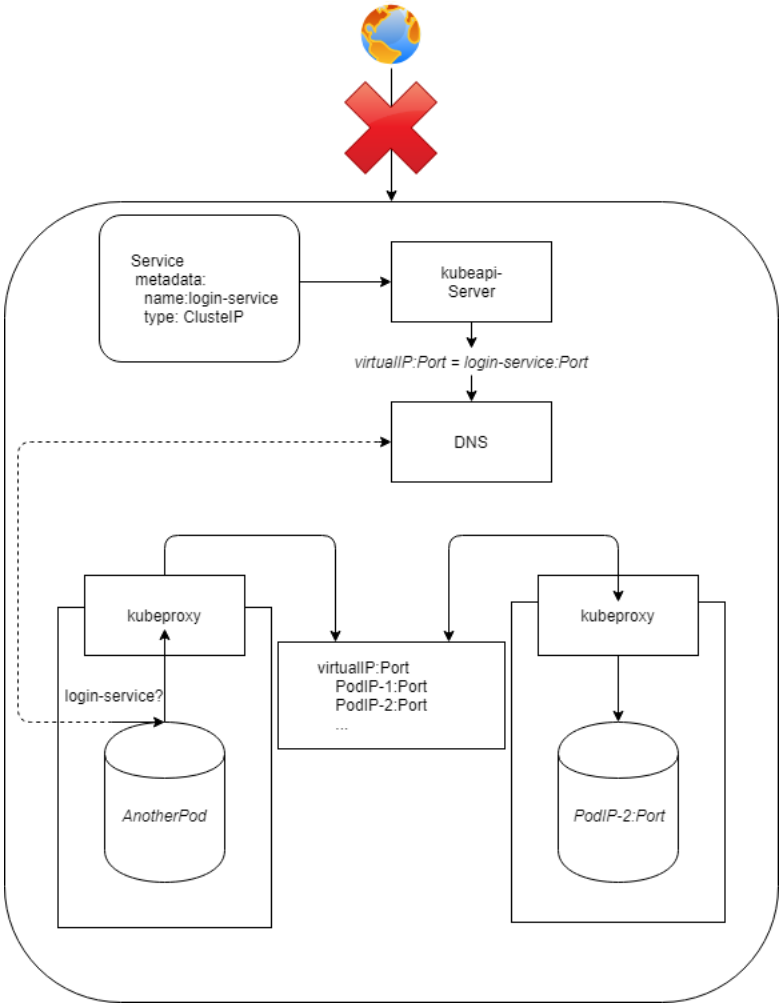


Abbildung 4 Schematische Darstellung des Service-Type ClusterIP

Für den Service wird ein DNS-Eintrag erzeugt, sodass dieser Service über einen DNS-Namen erreichbar ist. Um die in Kubernetes volatil gestalteten Pod-IP-Adressen zu abstrahieren, erhält der Service eine virtuelle IP-Adresse. Das bedeutet, dass diese IP-Adresse keinem Netzinterface zugeordnet ist. Der kube-proxy bildet für diese IP-Adresse IP-Tables, in denen die konkreten IP-Adressen der Pods des Services der virtuellen IP-Adresse des Services zugeordnet sind. Der kube-proxy kann als **eine Art** Router (technisch gesehen ist der kube-proxy am ehesten ein Reverse-Proxy) angesehen werden, der am *eth0*-Interface jedes Pods zwischengeschaltet ist, sodass

alle Datenpakete den kube-proxy passieren. Ein Pod kontaktiert immer den kube-proxy seines Nodes, die Anfragen an den kube-proxy unterliegen somit keinem Loadbalancing. Die virtuellen IP-Adressen der Services und die damit einhergehenden IP-Tables sind in jeder kube-proxy-Instanz verfügbar, also auf jedem Node. Es sind verschiedene Modi für die Bildung der IP-Tables möglich, welche in der Kubernetes-Dokumentation erläutert sind.
(2 S. https://kubernetes.io/docs/concepts/services-networking/#proxy-mode-iptables)
Der gewählte Modus hat aber keinen Einfluss auf die Funktionsweise von Services im Cluster.
Möchte nun ein Pod im Cluster den Service aufrufen, so geschieht dies über eine normale DNS-Anfrage, welche die virtuelle IP-Adresse des Services zurückgibt. An diese kann nun ein Datenpaket gesendet werden. Der kube-proxy nutzt nun seine IP-Tables, um das Datenpaket einem konkreten Pod zuzuordnen und an diesen weiterzuleiten. Technisch gesehen kommt hier Destination-NAT (DNAT) zur Anwendung, die Ziel-Adresse wird von *Service-IP* zu *Pod-IP* geändert. Für das Loadbalancing kommt ein Round-Robin-Verfahren zum Einsatz. **Ein Service des Typs *ClusterIP* ist nicht von außen erreichbar**. Mögliche Einsatzzwecke für diesen Service-Typ sind Services, die nicht von außerhalb des Clusters erreichbar sein sollen oder dürfen, wie zum Beispiel Datenbanken, Services, die Geschäftslogik abbilden oder ähnliche.

Um Services nach außen zu exponieren, ist ein Service vom Typ *NodePort* notwendig. Dieser sorgt dafür, dass auf jedem Node des Clusters ein Port geöffnet wird und der Service so von außen erreichbar ist, wie in Abbildung 5 dargestellt. Dieser Port kann angegeben werden (über den Parameter *NodePort*), falls er nicht angegeben wird, wählt Kubernetes einen zufälligen Port aus. Die Ports liegen zwischen 30000 und 32767. Dies ist der Standardwert für Ports, welcher sich über den Parameter *--service-node-port-range* des *kube-proxy* ändern lässt. Wenn der Port manuell angegeben wird, muss selbst Sorge dafür getragen werden, dass es nicht zu Kollisionen kommt. Daher empfiehlt es sich, die Ports automatisch durch Kubernetes zuteilen zu lassen. Über das Kommando *kubectl describe svc service-name* lässt sich dann der zugewiesene Port herausfinden.

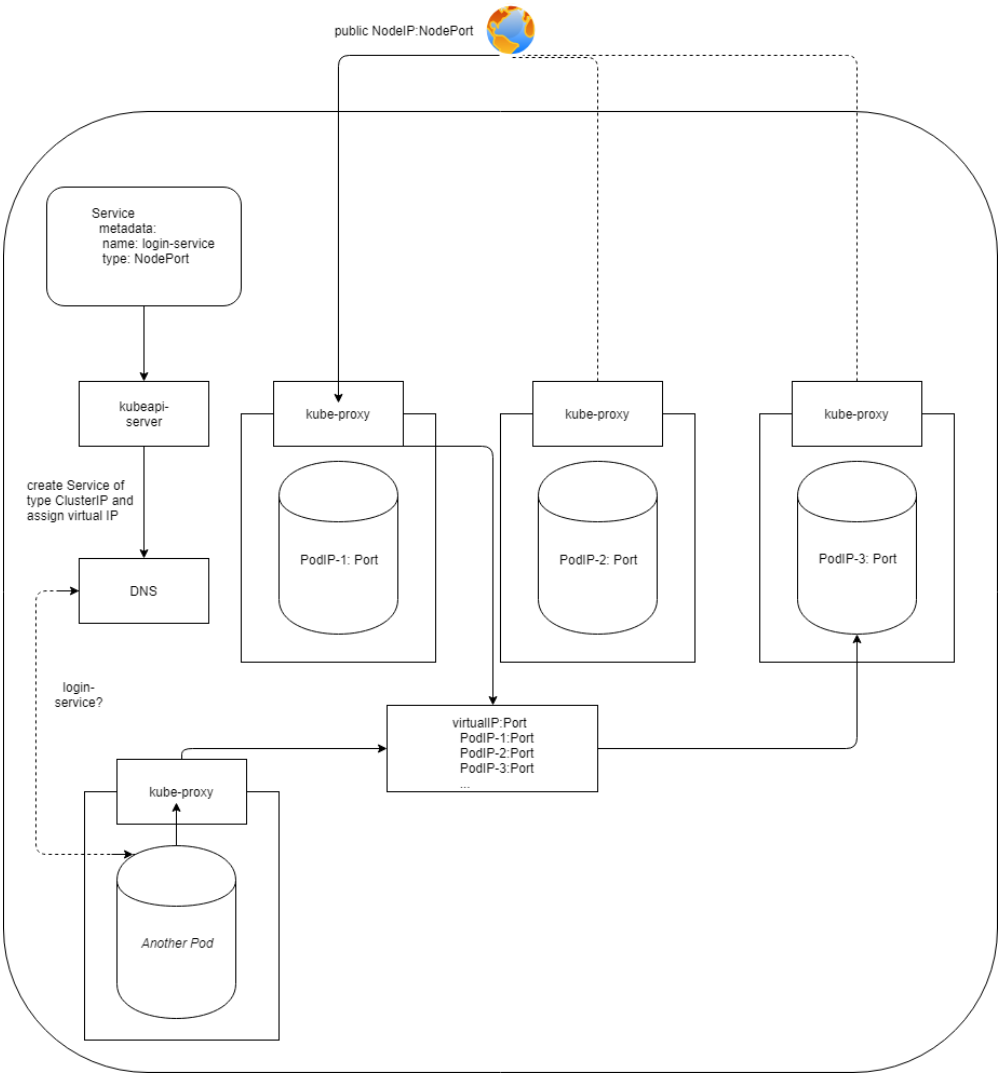


Abbildung 5 Schematische Darstellung des Service-Type NodePort

Der entscheidende Punkt bei Services vom Typ *NodePort* ist, dass alle Nodes des Clusters diesen Port öffnen werden. Die Logik, wie der Service nach außen verfügbar gemacht wird, gestaltet sich wie folgt: Der Nutzer definiert eine Service-Ressource mit dem Typ *NodePort* und spezifiziert alle notwendigen Parameter. Der API-Server erkennt beim Erstellen der Ressource, dass es sich um einen Service mit dem Typ *NodePort* handelt und erzeugt **automatisch** einen gleichen Service vom Typ *ClusterIP*. Dieser trägt **denselben** Namen, sodass der Service intern unter dem Namen zugänglich ist, den der Nutzer in der Service-Ressource vom Typ *NodePort* angegeben hat. Das bedeutet, dass interne Pods den Service durch eine DNS-Abfrage und die daraus resultierende virtuelle IP-Adresse erreichen können. Für Anfragen von internen Pods stellt sich der Ablauf wie in *ClusterIP* beschrieben dar, Zugriffe von außen laufen wie folgt ab:

Der Nutzer kann von außen über das Schema *NodeIP:NodePort* auf den Service zugreifen. Der kube-proxy wird automatisch so konfiguriert, dass er alle Anfragen, die unter diesem Port entgegengenommen werden, an die virtuelle IP-Adresse des automatisch generierten Service vom Typ *ClusterIP* weiterleitet.
Es ist möglich, diesem Konstrukt einen eigenen Loadbalancer vorzuschalten. Dieser arbeitet dann auf Layer vier und kann die Anfragen auf die verschiedenen Nodes verteilen. Falls das Umfeld, in dem das Kubernetes-Cluster arbeitet, keine Unterstützung für den Kubernetes-Service-Typ *LoadBalancer* besitzt, ist ein Service vom Typ *NodePort* die einzige Möglichkeit, Services zu exponieren. Hierbei gilt es folgende Sachverhalte zu beachten:
Falls kein eigener Loadbalancer vorgeschaltet werden kann, so muss dafür gesorgt werden, dass die Aufrufer der Services die öffentlichen IP-Adressen **aller** Nodes des Clusters kennen. Würde der Aufrufer nur eine spezifische IP-Adresse kennen, könnte zwar die Anfrage innerhalb des Clusters auf andere Nodes verteilt werden. Fällt aber der Node, dessen IP-Adresse der Aufrufer besitzt, aus, so ist der Service für den Aufrufer ebenfalls nicht mehr erreichbar. Daher sollte ein Aufrufer immer die IP-Adressen aller Nodes kennen.
Falls ein eigener Loadbalancer vorgeschaltet wird, so muss darauf geachtet werden, dass dieser zum einen die IP-Adressen aller Nodes kennt (damit der Loadbalancer überhaupt funktionieren kann), zum anderen muss dafür gesorgt werden, dass der Loadbalancer regelmäßig die Zustände der Nodes überprüft, um auszuschließen, dass Anfragen an nicht (mehr) verfügbare Nodes verteilt werden. Bei einem vorgeschalteten Loadbalancer muss der Zugriff von außen dann über die IP-Adresse des Loadbalancers erfolgen, idealerweise in Kombination mit einem DNS-Eintrag.

Um Services automatisiert nach außen bereitzustellen, ist noch ein anderer Service-Typ vorhanden: ein Service des Typs *LoadBalancer*. Dieser sorgt dafür, dass der Cloud-Controller-Manager (der zusammen mit einem Cloud-Provider-Integration-Plugin in der Control-Plane ausgeführt wird, wobei der Cloud-Controller-Manager eine native Kubernetes-Komponente ist, während das Cloud-Provider-Integration-Plugin spezifisch vom Cloud-Provider bereitgestellt wird) beim Cloud Provider einen Loadbalancer beantragt, welcher auf einen automatisch erzeugten Dummy-Service verweist. Dieser Dummy-Service ist vom Typ *NodePort* und hat die Aufgabe, eingehende Anfragen auf die virtuelle IP des Services zu leiten. Es findet eine Art

Mapping statt. (5) Die Erzeugung und Konfiguration des Dummy-Service sowie des Loadbalancers erfolgen automatisiert. Wie der Dummy-Service konfiguriert wird, ist nicht bekannt. Es stehen zwei Verfahren zur Verfügung, zum einen das direkte Lesen aus der etcd-Datenbank, zum anderen die Nutzung des DNS. Auch in diesem Fall wird automatisiert ein Service vom Typ ClusterIP erzeugt (welcher aus Gründen der Übersicht nicht in Abbildung 6 dargestellt ist), sodass sich der Zugriff im Cluster wie in *ClusterIP* darstellt. Der Ablauf einer Anfrage von außen an diesen Service gestaltet sich im FIRMA-Umfeld (beziehungsweise allgemein in Azure) dann wie in Abbildung 6:

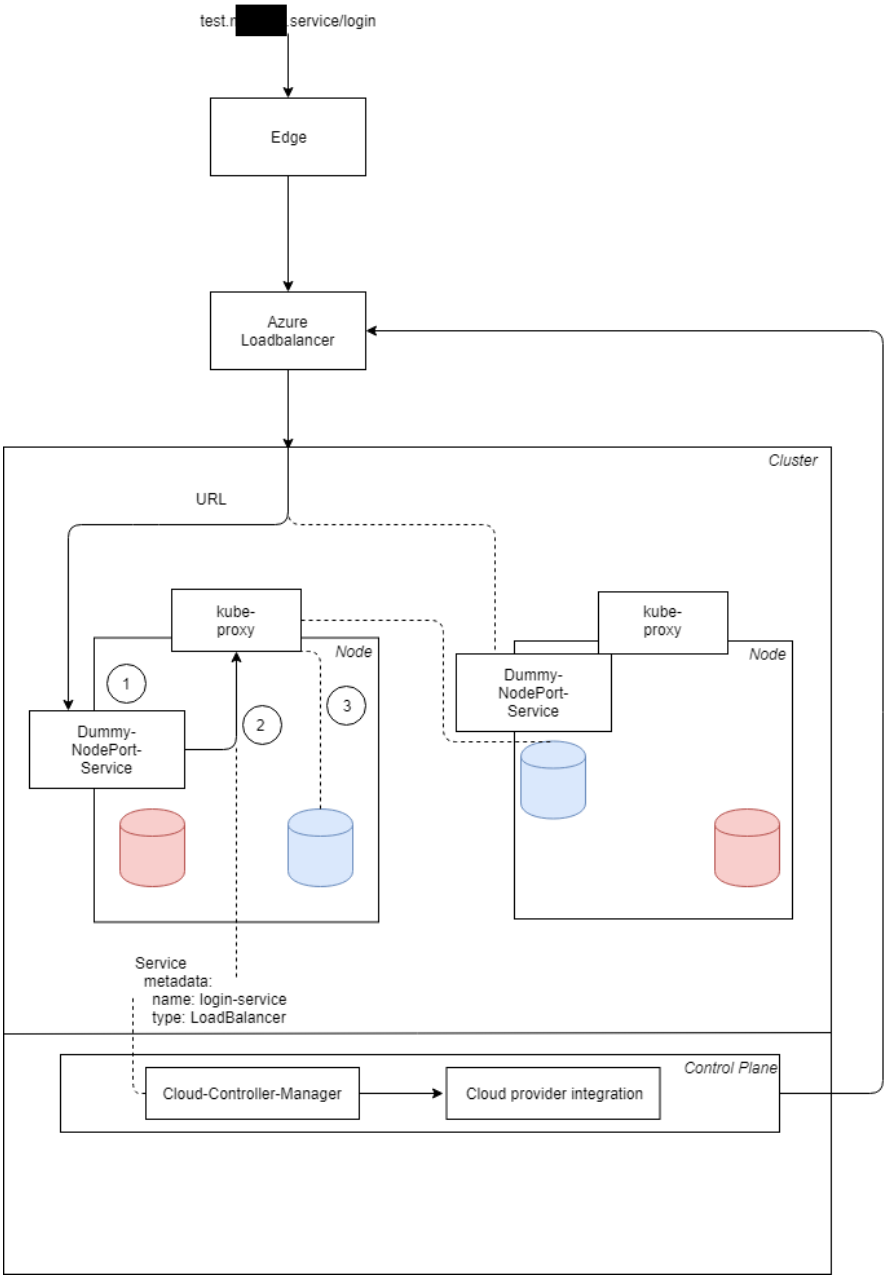


Abbildung 6 Schematische Darstellung des Service-Type LoadBalancer

Der Aufrufer nutzt eine URL, die Anfrage wird über den *Edge* zum erzeugten Loadbalancer geleitet. Dieser leitet die Anfrage an den Dummy-Service weiter (Schritt eins). Da dieser vom Typ *NodePort* ist, kann diese Anfrage an jeden Node des Clusters weitergeleitet werden, in der Praxis wird in der Regel ein Round-Robin-Verfahren eingesetzt, es liegt allerdings in der Hand des Cloud-Providers, wie dies implementiert ist. (2 S. https://kubernetes.io/docs/concepts/services-networking/#loadbalancer) Dieser DummyService leitet nun die Anfrage an die virtuelle IP-Adresse des angefragten Service weiter und der kube-proxy nutzt seine IP-Tables, um die Anfrage an einen konkreten Pod des Services weiterzuleiten (Schritte zwei und drei). Dieser Loadbalancer arbeitet allerdings auf Schicht vier des OSI-Modells, sodass keinerlei anwendungsspezifisches Routing umgesetzt werden kann.
Um Routingentscheidungen auf Layer sieben (Anwendungsschicht) des OSI-Modells treffen zu können, ist ein Ingress-Controller notwendig. Dieser setzt Ingress-Ressourcen (2 S. https://kubernetes.io/docs/concepts/services-networking/ingress/) um und ermöglicht so, Daten der Anwendungsschicht zu nutzen. Diese Möglichkeit, Services zu exponieren, erfordert ebenfalls einen Service des Typs *LoadBalancer*.
Der Ablauf einer Anfrage stellt sich dann folgendermaßen dar:

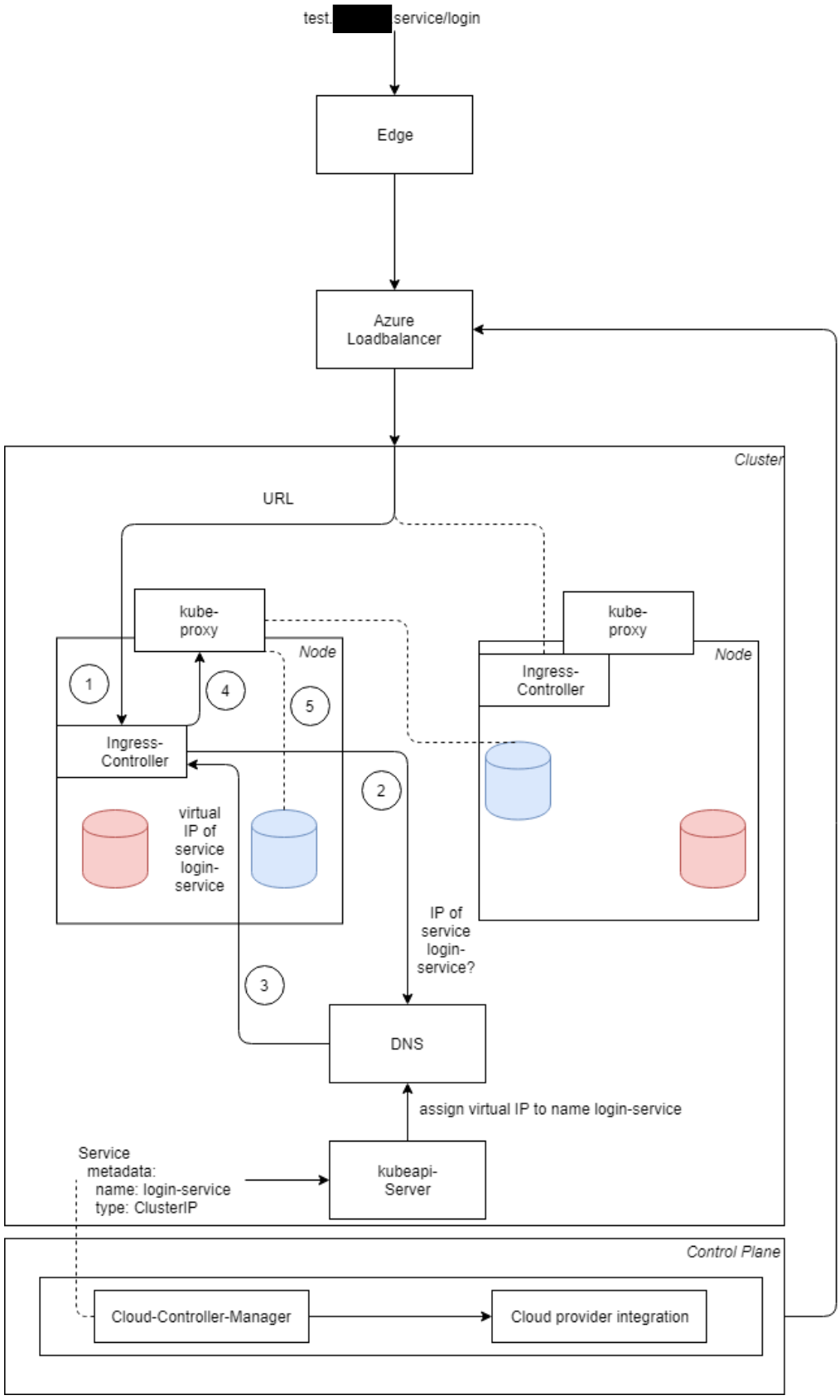


Abbildung 7 Schematische Darstellung des Service-Type LoadBalancer mit Ingress-Controller

Der Nutzer ruft ebenfalls eine URL auf, diese wird, wie bereits bekannt, durch den Edge zum erzeugten Loadbalancer geleitet. Dieser leitet die Anfrage dann ebenfalls

im Round-Robin-Verfahren an einen Node weiter, wo die Anfrage von einem Ingress-Controller entgegengenommen wird (Schritt eins). Dieser kann nun an Hand der Daten der Anwendungsschicht detaillierte Routingentscheidungen treffen und leitet die Anfrage gemäß seinen Regeln weiter. Hierfür benötigt er die virtuelle IP-Adresse des Services, an den die Anfrage weitergeleitet werden soll. Diese Information kann er über eine DNS-Abfrage erhalten (Schritt zwei und drei). Anschließend kann der Ingress-Controller die Anfrage weiterleiten (Schritt vier), der kube-proxy löst die Anfrage gemäß seinen IP-Tables auf und leitet diese an einen konkreten Pod des Services weiter (Schritt fünf). Dieser Pod kann sich auch auf einem anderen Node befinden.

Ein Hinweis zu den Typen *NodePort* und *LoadBalancer* sei hier erwähnt: Es ist in Kubernetes möglich, festzulegen, wie eingehende Verbindungen dieser beiden Typen hinsichtlich Ihrer Quell-IP-Adresse behandelt werden sollen. Diese Eigenschaft nennt sich *ExternalTrafficPolicy* und bestimmt im Wesentlichen, ob eigehende Verbindungen nur an lokale (auf dem Node, der die Anfrage entgegengenommen hat) oder clusterweite Endpunkte (Endpunkte sind in diesem Fall die konkreten Pods des Services) gesendet werden sollen[1]. (6) (2 S. https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/) Diese Einstellungsmöglichkeit ergibt sich aus der Tatsache, dass das Loadbalancing des Loadbalancers und des kube-proxy zwei getrennte Verfahren sind. Somit wäre es möglich, einzustellen, dass die Anfrage auf dem Node „verbleibt“, den der externe Loadbalancer ausgewählt hat, oder die Anfrage unabhängig davon auf einen anderen Nodes im Cluster weiterzuleiten.
Eine tiefergehende Erklärung der Abläufe ist nicht sinnvoll, da Kubernetes intern aus Microservices besteht und sich das konkrete Verhalten beziehungsweise die konkreten Aufgaben der abgebildeten Komponenten mit jeder Release-Version ändern könnten. Weiterhin kommt hinzu, dass verschiedene CNI-Plugins auch zu anderem Verhalten führen: So versucht das Plugin *Cilium*, den kube-proxy komplett durch eigenes Verhalten zu ersetzen.

[1] Für eine detaillierte Erklärung empfiehlt sich dieses Video, welches die Funktionsweise in der Google Kubernetes Engine erläutert: https://www.youtube.com/watch?v=y2bhV81MfKQ&feature=youtu.be&t=1823

2.1.7 Netzarchitektur in Kubernetes

Um die Abläufe in einem Kubernetes-Cluster besser zu verstehen, ist es hilfreich, die zu Grunde liegende Architektur im Hinblick auf das Netz zu verstehen. Hierfür müssen zwei grundlegende Fälle unterschieden werden: inter- und intra-Pod-Kommunikation. Erstere unterscheidet sich maßgeblich von der klassischen Docker-Netzarchitektur, welche in Abbildung 8 dargestellt ist:

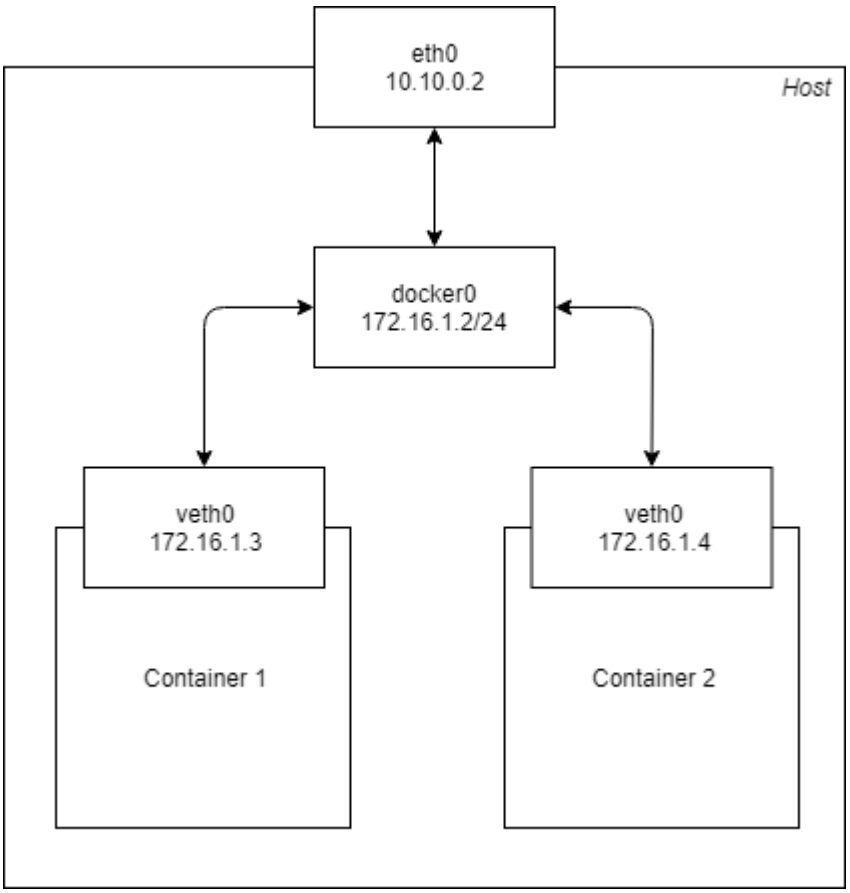


Abbildung 8 Netzarchitektur in Docker

Docker weist jedem Container eine eigene, nicht-öffentliche IP-Adresse zu und verbindet alle Container über eine virtuelle Bridge (*docker0*) mit dem Netzinterface des Hosts. (3 S. 257,259)

Es ist möglich, einem Container keinen Network-Namespace zuzuweisen, er ist dann netztechnisch vollständig isoliert, dies geschieht in Docker über den Parameter *--net=none*.

Dies ist aber nicht mit dem Kubernetes Netzmodell kompatibel, da dort folgende Grundsätze gelten:

- Jeder Pod erhält eine eindeutige IP-Adresse.

- Container sehen dieselbe IP-Adresse, die andere Komponenten verwenden, um sie zu erreichen: → Container innerhalb eines Pods haben dieselbe IP-Adresse. Die Adressierung eines Containers in einem Pod erfolgt über das Schema *PodIP:ContainerPort* (Dies wird über den Parameter *--net=container* erreicht).
- Alle Komponenten können direkt, ohne NAT, miteinander kommunizieren.

Nun ist die Netzarchitektur von Docker nicht mit dem zweiten Grundsatz kompatibel, weshalb Kubernetes hier einen anderen Ansatz wählt, den Host stellt in diesem Fall der Pod dar:

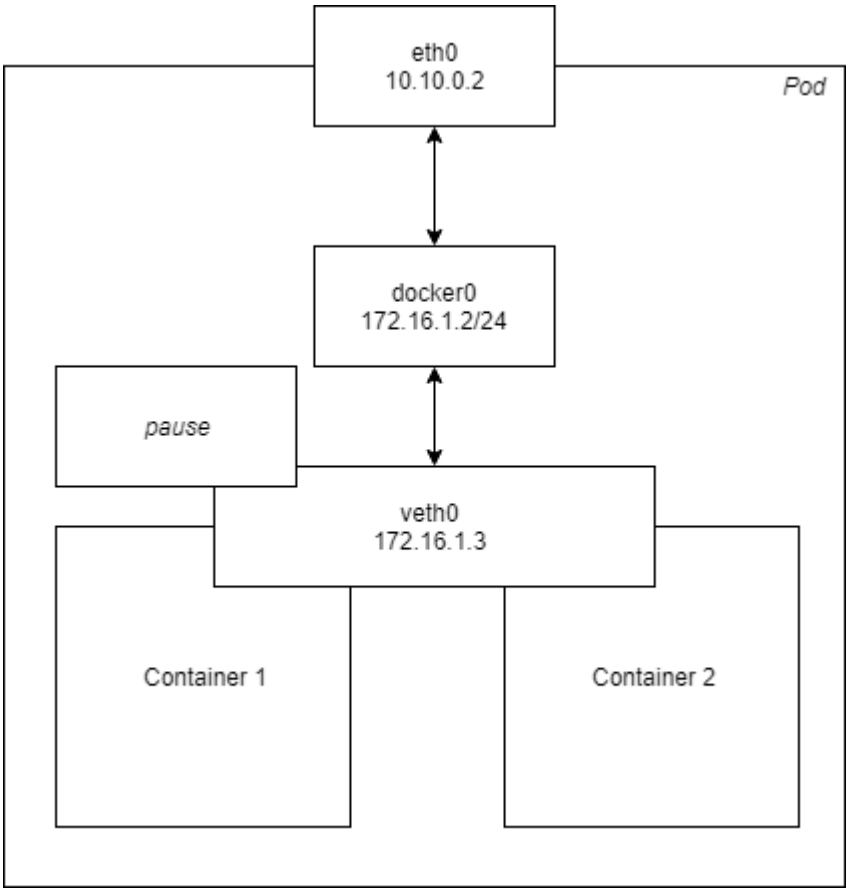


Abbildung 9 Netzarchitektur in Kubernetes

Die Container „teilen“ sich ein Netzinterface (wie in Abbildung 9 zu erkennen), somit ist der zweite Grundsatz erfüllt. Allerdings ist hierfür ein sogenannter *pause*-Container nötig. Dieser fungiert als „Elterncontainer“ und besitzt somit den Network-Namespace, sodass sich alle Container ein Netzinterface teilen können. (7) (8) Daraus folgt aber auch, dass jeder Port nur einmal pro Pod belegt werden kann! Die einzelnen Container müssen selbst dafür Sorge tragen, dass nicht versucht wird, einen bereits genutzten Port zu reservieren. Es empfiehlt sich daher, wie bereits erwähnt, nur einen Container

pro Pod zu nutzen. Falls dennoch mehrere Container in einem Pod betrieben werden sollen, empfiehlt es sich, automatische Portzuweisungen zu nutzen.
Nun muss diese Architektur in das Gesamtsystem, bestehend aus mehreren Nodes, eingebracht werden. Hierfür muss zunächst erläutert werden, wie zwei Pods miteinander kommunizieren können. Dazu werden einfach verschiedene Pods unter einer Bridge mit dem Namen *cbr0* (Abkürzung für custom bridge) wie in Abbildung 10 zusammengeschaltet:

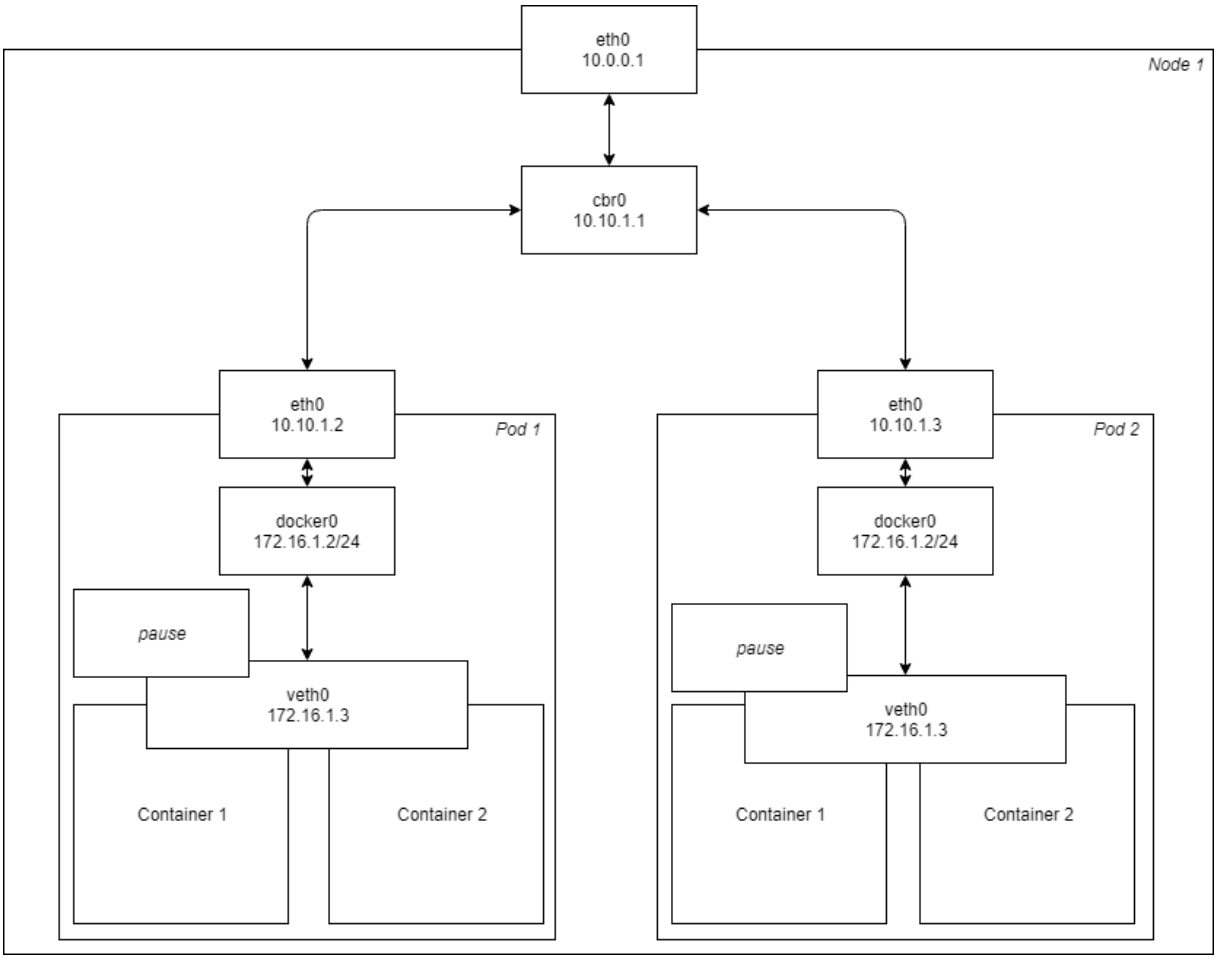


Abbildung 10 Netzarchitektur auf Pod-Ebene

Man sieht deutlich, dass beide Pods eine eigene IP-Adresse haben, somit sind sie eindeutig erreichbar. Welche IP-Adressbereiche „hinter“ dem Netzinterface eines Pods verwendet werden, ist in Kubernetes egal, da die kleinste Einheit ein Pod darstellt.
Wie bereits erwähnt, muss in Kubernetes jeder Pod eine eindeutige IP-Adresse erhalten, hier wurde das Netz 10.0.0.0/24 als privates Netz für das Cluster gewählt, die Pods erhalten das Netz 10.10.0.0/24. Pods können so direkt über ihre IP-Adresse miteinander kommunizieren. Nun sollte es aber auch möglich sein, dass Pods Node-

übergreifend miteinander kommunizieren können. Dies lässt sich durch simple Routen erreichen:

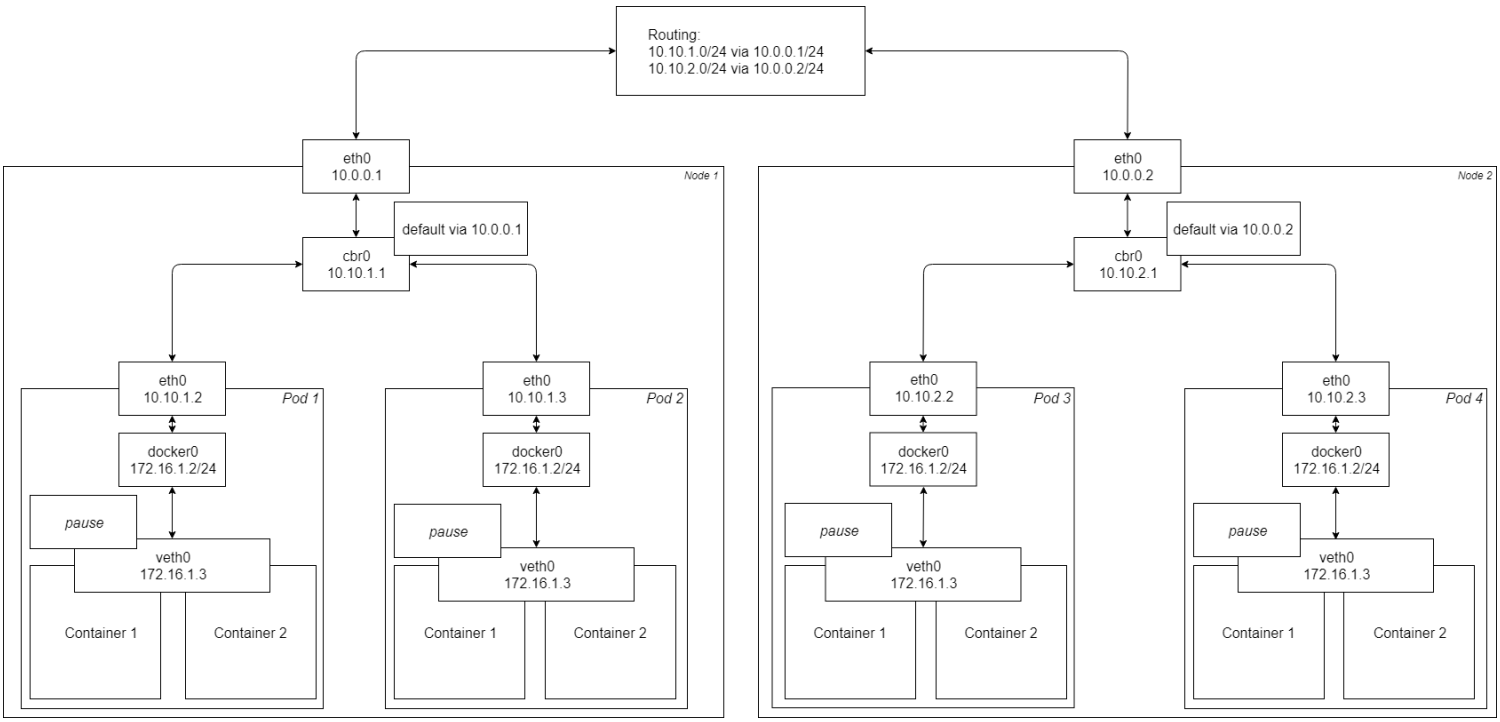


Abbildung 11 Schematische Darstellung der Netzarchitektur mit zwei Nodes

Über die in Abbildung 11 beschriebenen Routen und Wege ist es möglich, dass Pods verschiedener Nodes ohne NAT miteinander kommunizieren. Generell empfiehlt es sich, Nodes und Pods in getrennten IP-Adressbereichen zu halten.
Dieses Darstellungen sollen die Funktionsweise des Kubernetes-Netz erläutern, vor allem in Cloud-Umgebungen muss aber beachtet werden, dass wahrscheinlich kein Einfluss auf die IP-Adressbereiche der Nodes und Pods besteht.

2.1.8 Zertifikate

Kubernetes nutzt standardmäßig Zertifikate, um die Control Plane abzusichern. Alle Komponenten nutzen mTLS, die Zertifikate werden von der Cluster-CA ausgestellt. Bei hochverfügbaren kubeapi-Servern ist es wichtig, dass die SAN (Subject Alternative Names) des verwendeten Loadbalancer korrekt sind beziehungsweise dass die Zertifikate den DNS-Namen sowie die IP-Adresse des Loadbalancers als SAN enthalten, sonst werden Verbindungsversuche der Clients (korrekterweise) scheitern, da die Common-Names (CN) nicht übereinstimmen. Erwähnenswert ist ebenfalls, dass die kubelet-Instanzen ihren Hostname als Teil des CN tragen. Somit ist sichergestellt,

dass jedes Kubelet eine eigene „Identität“ hat. Die exakten Belegungen der CN-, O- und SAN-Felder finden sich in der Kubernetes-Doku.
(2 S. https://kubernetes.io/docs/setup/best-practices/certificates/)

2.2 Container

Container werden umgangssprachlich öfters als Ersatz für virtuelle Maschinen bezeichnet. Dies ist technisch gesehen aber falsch, weshalb in diesem Kapitel die Unterschiede zwischen klassischen virtuellen Maschinen (im Folgenden VM genannt) erläutert werden.
Zunächst soll die klassische Architektur von VMs beleuchtet werden, hier gilt es zwei Typen zu unterscheiden (9):

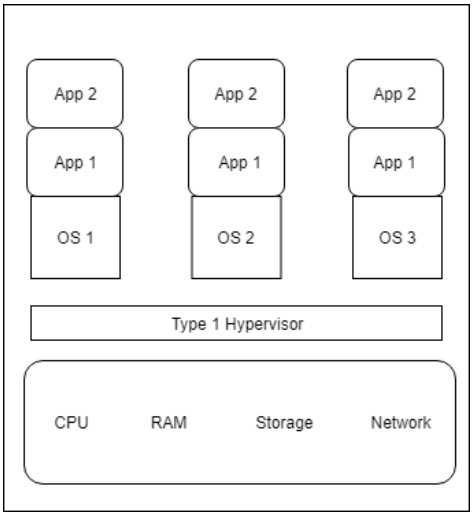


Abbildung 12 Hypervisor vom Typ 1

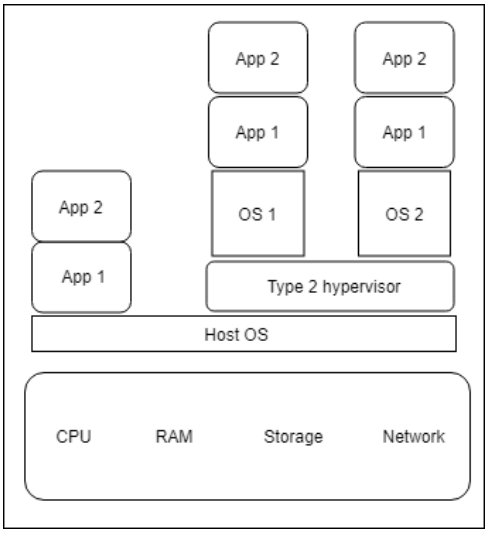


Abbildung 13 Hypervisor vom Typ 2

Ein Hypervisor vom Typ eins (Abbildung 12) setzt direkt auf der Hardware auf und benötigt kein zu Grunde liegendes Betriebssystem. Auf dem Hypervisor können dann verschiedene Betriebssysteme (sogenannte Gastsysteme) installiert werden, die unabhängig voneinander arbeiten.
Ein Hypervisor vom Typ zwei (Abbildung 13) hingegen benötigt ein zu Grunde liegendes Betriebssystem (Hostsystem). Danach können auch mit ihm beliebige Betriebssysteme (sogenannte Gastsysteme) installiert werden. Ein Typ zwei Hypervisor besitzt vor allem den Nachteil, dass jede Aktion eines Gastsystems den Hypervisor passieren muss (dieser muss die Systemaufrufe des Gastsytems in die entsprechenden Äquivalente des Hostsystems übersetzen), was zu einer gewissen Verzögerung führt. Daher werden Hypervisor vom Typ zwei in der Regel auch nicht in

Rechenzentren genutzt, sondern sind Endnutzern in Software wie VirtualBox oder VMWare vorbehalten. Hypervisoren vom Typ eins können die Übersetzung der Systemaufrufe deutlich performanter durchführen. Bei beiden Varianten denkt jedes Betriebssystem, dass es die Hardware alleine nutzt. Die schematische Darstellung aus Abbildung 12 und 13 würde sich für Container folgendermaßen darstellen, die Container-Engine stellt im Fall Docker der *Docker-Daemon* dar:

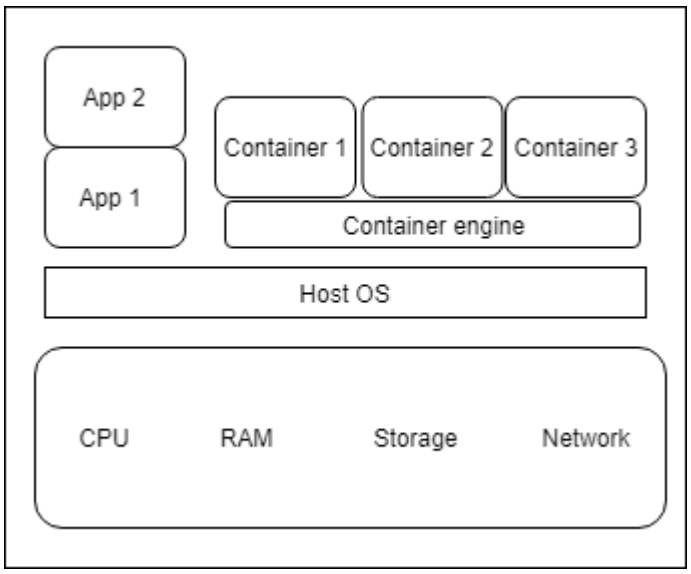


Abbildung 14 Container-Architektur

Der wesentliche Unterschied ist die Stelle, an der die Hardware-Abstraktion erfolgt: Bei klassischen VMs wird die Hardware auf physischer Ebene virtualisiert beziehungsweise abstrahiert, während bei Containern die Abstraktion auf Ebene des Betriebssystems geschieht. Auf dieser Ebene kommen *cgroups* und *namespaces* zum Einsatz, um die Abstraktion zu bilden. Da dies eine Ebene „höher“ geschieht, sind Container flexibler als klassische VMs. Das bedeutet, dass Container auf einer Maschine alle denselben Kernel nutzen, während bei einem Hypervisor jedes Gastsystem seinen eigenen Kernel besitzt.

Daraus folgt aber auch, dass die Angriffsfläche eines Containers deutlich größer ist als die einer VM. Eine VM bietet nur die Möglichkeit, die virtualisierte Hardware wie Speicher, Prozessor (und dessen virtualisierten Interrupt Controller) sowie die Gastadressen des RAM anzugreifen. Bei einem Container stehen jedoch circa 400 Systemaufrufe des Kernels zur Verfügung. Die Sicherheitstrennung erfolgt also bei einer VM durch den Hypervisor, bei einem Container jedoch durch das Betriebssystem.

Daher ist ein Container letztendlich „nur“ ein Prozess eines Betriebssystems, der (im Gegensatz zu einer virtuellen Maschine) direkt auf das Betriebssystem „zugreifen“ kann. Der Prozess wird durch die Container Engine erzeugt (im Fall Docker entspricht das dem Docker-Daemon), welche als eigener Prozess realisiert ist.

2.2.1 Wie funktioniert Isolierung mit Docker?

Dieses Kapitel soll die Funktionsweise von Docker erläutern, der Fokus soll aber nicht auf Container-Images oder ähnlichem liegen, sondern auf den zugrunde liegenden Sicherheitsfeatures, die der Isolierung dienen, nämlich *namespaces* und *control groups*. Stark vereinfacht lässt sich sagen: *cgroups*[2] regeln, wieviel der Hardware-Ressourcen ein Container nutzen darf, während *namespaces* regeln, „was“ ein Container sehen darf beziehungsweise sieht. Ein Container beziehungsweise ein Prozess denkt also, dass er alleine läuft, obwohl er sich die Ressourcen des Systems mit vielen anderen Prozessen teilt.

Namespaces werden in Docker genutzt, um Isolierung zu gewährleisten. Der Linux-Kernel stellt verschiedene Namespaces bereit, für Docker sind *PID*, *MNT* sowie *NET* von größter Bedeutung. Weiterhin nutzt Docker auch noch den UID-Namespace (zum Mappen von User-ID im Container und auf dem Host) sowie den UTS-Namespace (damit jeder Container einen eigenen Hostnamen erhält) sowie den IPC-Namespace. PID (Process ID) isoliert die Sicht auf Prozesse, das bedeutet, dass ein Container (beziehungsweise allgemein ein Prozess) nur die Prozesse sieht, die in ihm laufen. Das lässt sich an einem einfachen Beispiel zeigen: wird ein neuer Prozess in einem eigenen PID-Namespace mit einer Bash gestartet, so sieht er nur die Prozesse, welche in ihm laufen:

```
root@ubuntu:~# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.1  0.2  29680  4876 pts/0    S    04:03   0:00 bash
root          9  0.0  0.1  44472  3172 pts/0    R+   04:03   0:00 ps aux
```

Abbildung 15 Auflistung der ausgeführten Prozesse in einem Container

Führt man denselben Befehl auf dem Host aus, so sieht man alle Prozesse des Hosts und stellt fest, dass der gestartete Prozess auf dem Host die Prozess-ID 1874 erhalten hat und unter dem normalen Benutzer läuft, unter dem er gestartet wurde.

[2] Eine sehr ausführliche Beschreibung, wie cgroups funktionieren, findet sich hier: http://docker-saigon.github.io/post/Docker-Internals/#how:cb6baf67dddd3a71c07abfd705dc7d4b

Dasselbe Prinzip gilt auch für die Namespaces *MNT* und *NET*: ein Prozess sieht nur die Dateien (unter Linux sind auch Verzeichnisse Dateien), die in seinem MNT-Namespace liegen, ebenso verhält es sich mit Netzinterfacen und dem NET-Namespace. So erklärt sich zum Beispiel die Netzarchitektur von Docker, dass jeder Container ein eigenes *eth0*-Interface erhält und diese zentral über eine Bridge (*docker0*) zusammengeschaltet werden. Prinzipiell dienen Namespaces der Isolierung der Container vom Host-System.

Weiterhin können in Docker aber auch *control groups* (im Folgenden *cgroups*) genutzt werden, um die Ressourcen des Host-Systems zu überwachen und Beschränkungen durchzusetzen. Es soll nur erläutert werden, welche *cgroups* vorhanden sind, **eine manuelle Beschränkung für Docker-Container vorzunehmen ist sinnlos**, da *cgroups* die Angabe einer Prozess-ID erfordern und diese nach jedem Container-Neustart anders ist. Docker verwaltet die *cgroups* automatisch, wenn entsprechende Ressourcen-Limits für einen Container gesetzt sind. Dies geschieht durch das Anhängen der entsprechenden Parameter an einen *docker run*-Befehl oder das Eintragen der entsprechenden Felder in einer Docker-Compose-Datei. (10 S. https://docs.docker.com/compose/compose-file/#resources)
(10 S. https://docs.docker.com/config/containers/resource_constraints/)

Zunächst lässt sich mit *cgroups* der RAM beschränken. Dies betrifft sowohl Speicherseiten, die auf der Festplatte gespeichert sind, als auch Speicherseiten, die noch nicht auf der Festplatte gespeichert sind. Überschreitet ein Container sein Limit, so wird er (je nach Ressourcenauslastung des Host-Systems) beendet.
Weiterhin ist es noch möglich, CPU und wie viel Speicher auf die Festplatte ausgelagert werden darf, zu beschränken.
(10 S. https://docs.docker.com/config/containers/resource_constraints/)

2.2.2 Was sind Docker-Images?

Um dem Docker-Ansatz *build, ship, run any app, anywhere*, beziehungsweise neuerdings *securely build, share and run any application anywhere* (11) gerecht zu werden, wird ein spezielles Image-Format verwendet, das sich maßgeblich von einem klassischen Dateisystem unterscheidet und nun näher erläutert werden soll.

Docker-Images setzen sich aus verschiedenen Schichten zusammen, sodass das gesamte Image aus vielen Schichten besteht. Jede Schicht besitzt ein Read-only-Dateisystem, welches in der Docker-Dokumentation näher erläutert wird. Das Ziel ist es, dass diese vielen Dateisysteme schlussendlich zu einem einzigen Dateisystem aggregiert werden. (10 S. https://docs.docker.com/storage/storagedriver/aufs-driver/) Daraus folgt zunächst, dass Images auf dem Host gespeichert werden können und von verschiedenen Containern genutzt werden können. Eine Veränderung der Daten im Image ist durch das Read-only-Dateisystem ausgeschlossen. Falls innerhalb eines Images bestehende Dateien verändert werden sollen, stellt Docker den Ansatz *Copy on write* bereit: zur Laufzeit (mit dem Befehl *docker run*) wird ein schreibbares Dateisystem als oberste Schicht des Images erzeugt. Soll nun eine Datei aus den unteren Schichten modifiziert werden, wird die Datei in diese oberste Schicht kopiert und dann modifiziert (copy on write). Anschließend ersetzt die neue Version die alte Version aus der unteren Schicht. Solange nur lesende Zugriffe auf die Dateien der unteren Schichten erfolgen, verbleiben diese in ihren Schichten. In Abbildung 16 und 17 sind diese Schichten beispielhaft im Dockerfile und dem daraus resultierenden Docker-Image dargestellt:

```
FROM node:8.16-jessie-slim
RUN mkdir -p  /usr/nodeapp
WORKDIR /usr/nodeapp
COPY package.json /usr/nodeapp
RUN npm install
COPY /nodeapp /usr/nodeapp/
EXPOSE 8080
CMD["npm",  "start"]
```

Abbildung 16 Beispielhaftes Dockerfile

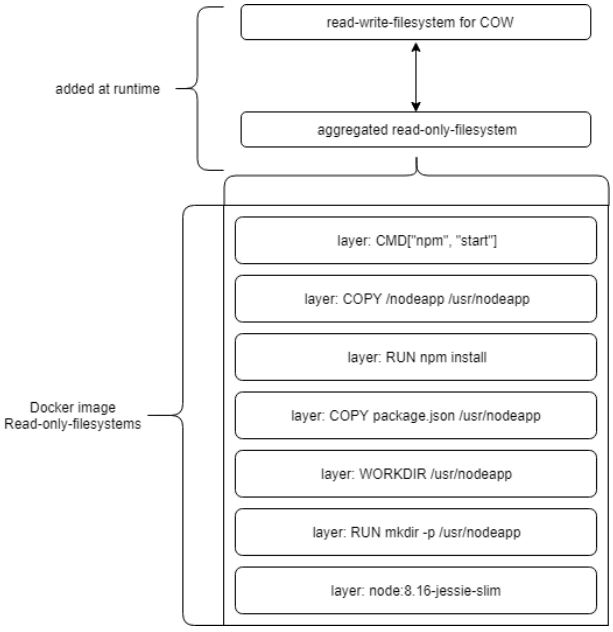


Abbildung 17 Aus Abbildung 16 resultierende Schichten

Die oberste Schicht sowie das aggregierte Read-Only-Dateisystem werden mit dem Beenden des Containers wieder entfernt, sodass die Schichten des Images unverändert bleiben. Daraus folgt aber auch, dass Docker-Images ohne gemountetes

Dateisystem beziehungsweise Verzeichnisse des Hosts oder das Speichern in einer externen Datenbank keine Persistenz bieten!

3. Security Best-Practices in Docker und Kubernetes

Dieses Kapitel beschäftigt sich mit den grundlegenden Best-Practices, um Kubernetes und Docker abzusichern. Da bei FIRMA vermehrt Docker als Container-Engine eingesetzt wird, widmet sich der zweite Teil dieses Kapitels auch diesem Thema.

3.1 Kubernetes

Wie jedes System muss auch Kubernetes gegen Angriffe abgesichert werden. Da in Kubernetes jede Art von Anwendung bereitgestellt werden kann, sind Maßnahmen notwendig, um die Schutzziele der IT-Sicherheit zu gewährleisten. Dieses Kapitel soll gängige Sicherheitsfeatures in Kubernetes erläutern.

3.1.1 Kubernetes-Dashboard

Das Kubernetes-Dashboard ist eine webbasierte Benutzeroberfläche, um das Cluster zu verwalten. Das bedeutet im Konkreten: Ressourcen ändern, löschen und hinzufügen, Logdateien einsehen sowie einen schnellen Überblick über das Cluster zu erhalten. Alle Funktionen, die das Kommandozeilenprogramm *kubectl* beherrscht, sind ebenfalls im Dashboard vorhanden. (2 S. https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard)

Das Dashboard enthält bis zur Version **1.10.0** eine Sicherheitslücke, welche die Umgehung einiger Sicherheitsmechanismen ermöglicht. Um das Dashboard zu nutzen, gibt es seit Version 1.7.0 eine Anmeldefunktion, welche ein Token oder eine *kubeconfig*-Datei akzeptiert, es ist allerdings auch eine Schaltfläche zum Überspringen vorhanden. Bis zur Version **1.10.0** ist es möglich, nach einem Klick auf diese Schaltfläche über das Anhängen einer API-Anfrage in die Adresszeile des Browsers, diese Anfrage durchzuführen und eine Antwort vom Server zu erhalten. So ist es zum Beispiel möglich, die Zertifikate des Dashboards oder anderer Services zu stehlen. (12) Die Anfrage wird aus technischer Sicht über den *ServiceAccount* des Dashboards durchgeführt, sodass auch in etwaigen Logdateien keine expliziten Anzeichen für ein Ausnutzen der Sicherheitslücke gefunden werden können. (12) Die Sicherheitslücke hat keine direkten Auswirkungen auf die Schutzziele Integrity und Availability, allerdings wird das Ziel der Confidentiality in hohem Maße verletzt. Die Sicherheitslücke ist unter der Nummer *CVE-2018-18264* registriert. Die Lücke ist noch

um einiges gefährlicher, wenn das Dashboard nach außen verfügbar ist. Zusätzlich ist die Lücke vergleichsweise frisch, sie stammt aus Dezember 2018, sodass davon auszugehen ist, dass noch viele Versionen des Dashboards mit den betroffenen Versionen im Einsatz sind. Die Lücke wurde mit der Version **1.10.1** geschlossen. (13) Um die Lücke auszunutzen, muss zunächst das Dashboard aufgerufen werden:

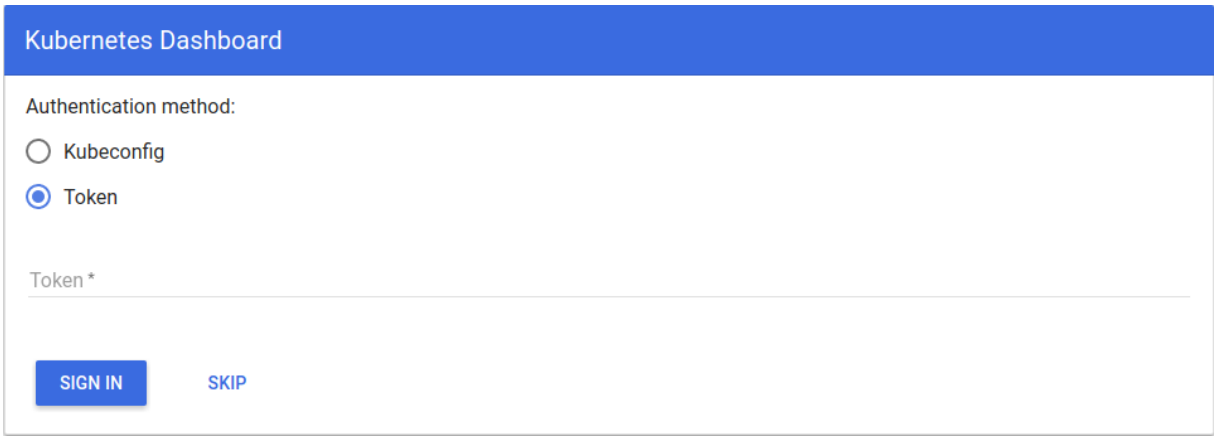


Abbildung 18 Button zum Überspringen der Anmeldung

Wie in Abbildung 18 gezeigt, ist es möglich, die Anmeldung zu überspringen. Nach einem Klick auf den Button „Skip“ wird das Dashboard aus Sicht des standardmäßigen Service-Accounts dargestellt und es ist möglich, Anfragen in der Adresszeile des Browsers auszuführen oder durch die verschiedenen Kategorien des Dashboards zu navigieren und Daten einzusehen. So erhält man für die Anfrage nach den Dashboard-Secrets folgendes Ergebnis:

Abbildung 19 Antwort der Abfrage der Dashboard-Secrets

Wie man in Abbildung 19 sehen kann, liefert der Server für die Anfrage */api/v1/namespaces/kube-system/secrets/kubernetes-dashboard-certs* sowohl das Zertifikat als auch den privaten RSA-Schlüssel. Es war ebenfalls möglich, eine Liste aller Pods zu erhalten. Daher ist es nachvollziehbar, dass die Einstufung in der CVE-Datenbank 7.5 beträgt. (12)

Der Angriffsvektor ist allerdings beschränkt, da das Dashboard standardmäßig nicht direkt über die IP-Adresse eines der Worker-Nodes zugänglich ist; es sind lediglich direkt von außerhalb des Clusters erreichbare Dashboards in den betroffenen Versionen verwundbar. Bei diesen kommt aber erschwerend hinzu, dass es keine automatische Aktualisierung des Dashboards gibt. (14) Somit sind wahrscheinlich noch viele Kubernetes-Cluster mit verwundbaren Dashboard-Versionen in Betrieb. Außerdem ist darauf zu achten, nach der Aktualisierung neue Zertifikate zu erstellen und diese dem Dashboard zur Verfügung zu stellen, da auch dies nicht automatisiert geschieht.

Um den Zugriff auf das Dashboard zu reglementieren, stehen mehrere Möglichkeiten bereit. Seit der Version 1.7 nutzt das Dashboard einen eigenen Service-Account, der keinerlei Rechte besitzt. Es ist zwar möglich, sich mit dessen Token anzumelden, allerdings werden aufgrund der minimalen Rechte keine Daten angezeigt. Möchte man die Token-basierte Anmeldung nutzen, hat man entweder die Möglichkeit, einen eigenen Service-Account mit entsprechenden Rechten zu erzeugen, oder dem Service-Account des Dashboards die entsprechenden Rechte zu geben. Da ein Cluster in der Regel von mehreren Personen genutzt wird, ist eine Verwendung der zweiten Variante nicht sinnvoll, jeder Nutzer sollte seinen eigenen Service-Account mit individuellen Regeln und eigenem Token haben.

Damit lassen sich verschiedene Rollen und eine feingranulare Struktur erreichen.

So lässt sich zum Beispiel ein Service-Account mit ausschließlichem Lesezugriff erstellen, welcher dann für Visualisierungszwecke genutzt werden könnte.

Hierfür sind die Verben *get, list* und *watch* nötig, als Ressourcen werden *configmaps, endpoints, persistentvolumeclaims, pods, replicationcontrollers, replicationcontrollers/scale, serviceaccounts, services, deployments, daemonsets, replicasets, statefulsets, nodes* und *persistencevolumes* benötigt. Diese *ClusterRole* lässt sich dann einem Service-Account zuweisen und man erhält einen ausschließlich lesenden Nutzer. Weitere Möglichkeiten würden in der weiteren Einschränkung der Ressourcen (zum Beispiel eine Rolle, mit der man keine Service-Accounts lesen darf) oder der Namespaces, (zum Beispiel eine Rolle, die nur den Namespace *development* sehen darf) bestehen. Mit solch detaillierten Einschränkungsmöglichkeiten kann das Least-Privilege-Prinzip sehr gut umgesetzt werden. Leider gehen viele Anleitungen zu Kubernetes dazu über, dem standardmäßigen Service-Account des Dashboards entweder die Rolle *cluster-admin* oder eine eigene Rolle mit Administratorrechten

zuzuweisen. Lediglich die offiziellen Dokumente von Google weisen darauf hin, dass man sich darüber im Klaren sein muss, was man damit tut. Da dies in Kubernetes zusammen mit dem eigentlichen Deployment alles in einer einzigen Datei stehen kann, sollte unbedingt der Inhalt von fremden Deployments beziehungsweise Dateien, die in das Kubernetes-Cluster importiert werden, geprüft werden.
Für fortgeschrittene Authentifizierungsmethoden stehen externe Plugins bereit, die die Einbindung von zum Beispiel OAUTH oder LDAP-Diensten ermöglichen. (14 S. https://github.com/kubernetes/ingress-nginx/tree/master/docs/examples/auth/oauth-external-auth)

Da das Dashboard ebenso mit der Kubernetes-API kommuniziert, wie Kommandozeilenprogramme, sollte darauf geachtet werden, dieses entsprechend den vorgestellten Möglichkeiten abzusichern, das bedeutet, die Einhaltung des Least-Privilege-Prinzips zu forcieren sowie regelmäßig auf Sicherheitslücken und Updates zu prüfen. Zusätzlich sollte es grundsätzlich vermieden werden, das Dashboard nach außen zu exponieren. Ein Zugriff ist dann über den Proxy möglich, wie es in diversen Anleitungen beschrieben wird. (2 S. https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard)

3.1.2 Role Based Access Control (RBAC)

RBAC ist aktuell das standardmäßige Autorisierungsverfahren in Kubernetes. Abbildung 20 beschreibt den schematischen Ablauf der Validierung einer Anfrage an den API-Server:

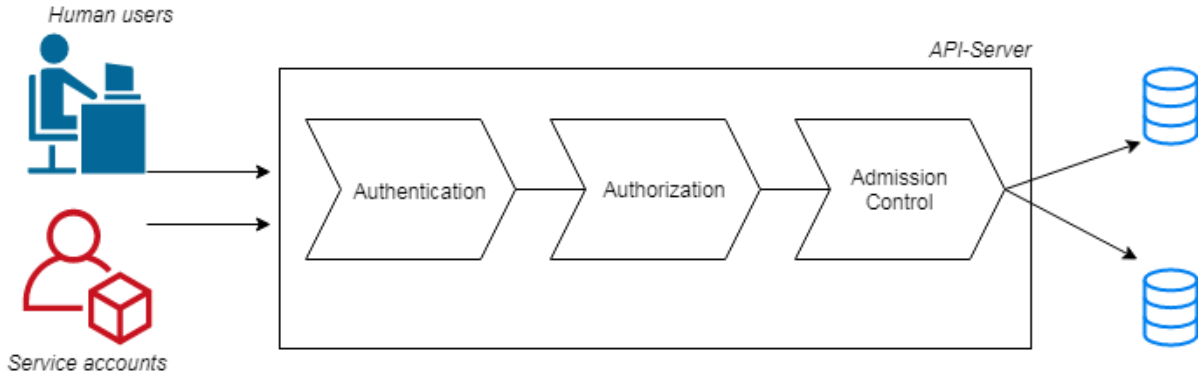


Abbildung 20 Schematischer Kontrollfluss einer Anfrage

Zunächst wird der Anfragende in Schritt eins authentifiziert, anschließend folgt die Autorisierungsprüfung. In diesem Schritt kommt RBAC zur Anwendung. Im letzten

Schritt wird die Anfrage durch verschiedene Admission Controller geprüft. Diese Stufe gliedert sich in zwei Phasen: Zunächst werden alle verändernden Admission Controller angewendet, anschließend alle validierenden. (15 S. 293)

Kubernetes unterscheidet bei RBAC zwei grundlegende Objekte: *Roles* und *RoleBindings*. *Roles* verbinden API-Objekte und die erlaubten Verben, während *RoleBindings* die Verbindung von Rollen mit Subjekten (Nutzer, Gruppen oder ServiceAccounts) herstellen, die diese Rollen dann ausüben beziehungsweise nutzen können. Seit Kubernetes in der Version 1.8 (ab dieser Version ist RBAC in Kuberentes als *stable* markiert) ist dieses Feature standardmäßig aktiviert und auch das standardmäßige Authorisierungsverfahren. In darunterliegenden Versionen ab Version 1.6 (ab dieser Version ist RBAC in Kubernetes in der Beta-Version verfügbar) kann RBAC mit dem Kommando *--authorization-mode=RBAC* beim Starten des API-Servers aktiviert werden. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/rbac/)

Anzumerken ist hierbei, dass sich *Roles* immer nur auf einen Namespace beziehen (sie können jedoch in mehrere Namespaces gleichzeitig verwendet werden, das bedeutet, dass die *Role* „*developer*“ sowohl in den Namespaces *dev*, *prod* und *qm* exisitieren könnte); um clusterweite Rollen zu implementieren existieren *ClusterRole* und das dazugehörige *ClusterRoleBinding*.

Aus den Unterschieden zwischen *Role* und *ClusterRole* folgt auch, dass mit *Roles* nur auf Objekte in einem Namespace zugegriffen werden kann (zum Beispiel Pods im definierten Namespace), mit einer *ClusterRole* jedoch auf Objekte des gesamten Clusters (zum Beispiel Nodes des Clusters) zugegriffen werden kann. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/rbac/)

Bei der Definition von *Roles* kommen sogenannte Verben zum Einsatz, welche bestimmen, was mit API-Ressourcen getan werden darf. Die gängigsten Beispiele für diese Verben sind *get, create, list* und *update*.

So könnte eine einfache *Role*, die es erlaubt, alle Pods und Services im *default*-Namespace abzufragen, wie in Abbildung 21 aussehen:

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1alpha
metadata:
  namespace: default
  name: read-pods-default-ns
rules:
  apiGroups: [""]
  ressources: ["pods", "servcices"]
  verbs: ["get", "list"]
```

Abbildung 21 Einfache Regel, die das Lesen von Pods und Services erlaubt

Diese Datei beschreibt eine *Role*, was durch das Attribut „kind“ angegeben wird; hier wäre auch *ClusterRole* denkbar. Die Liste „metadata“ enthält alle Informationen über die *Role*, die nichts mit den eigentlichen Berechtigungen zu tun haben. Das sind bei RBAC immer der Namespace, in welchem die *Role* gelten soll sowie der Name, welcher später für ein *RoleBinding* benötigt wird. Bei einer *ClusterRole* wird kein Namespace-Attribut benötigt. Die Liste „rules“ spezifiziert nun die eigentlichen Berechtigungen, hierfür sind im Wesentlichen die „ressources“ und die dazugehörigen „verbs“ relevant, die festlegen, was die Rolle mit welchen Ressourcen tun darf. Bei diesem Attribut ist der Platzhalter „ * “ erlaubt, um alle Verben zu ermöglichen. Das Attribut „apiGroups“ beschreibt, welche API-Groups diese Ressource spezifiziert. Da Pods und Services der Default-Gruppe angehören, muss dieses Feld leer sein, da diese Gruppe in Kubernetes tatsächlich durch ein leeres Feld repräsentiert wird. Mit dem Befehl *kubectl api-resources -o wide* lassen sich alle API-Ressourcen und deren dazugehörige API-Group anzeigen. In Abbildung 22 sind einige davon zu sehen:

```
dev@ubuntu:~$ kubectl api-resources -o wide
NAME                              SHORTNAMES   APIGROUP                       NAMESPACED   KIND                             VERBS
bindings                                                                      true         Binding                          [create]
componentstatuses                 cs                                          false        ComponentStatus                  [get list]
configmaps                        cm                                          true         ConfigMap                        [create delete deletecollection get list patch update watch]
endpoints                         ep                                          true         Endpoints                        [create delete deletecollection get list patch update watch]
events                            ev                                          true         Event                            [create delete deletecollection get list patch update watch]
limitranges                       limits                                      true         LimitRange                       [create delete deletecollection get list patch update watch]
namespaces                        ns                                          false        Namespace                        [create delete get list patch update watch]
nodes                             no                                          false        Node                             [create delete deletecollection get list patch update watch]
persistentvolumeclaims            pvc                                         true         PersistentVolumeClaim            [create delete deletecollection get list patch update watch]
persistentvolumes                 pv                                          false        PersistentVolume                 [create delete deletecollection get list patch update watch]
pods                              po                                          true         Pod                              [create delete deletecollection get list patch update watch]
podtemplates                                                                  true         PodTemplate                      [create delete deletecollection get list patch update watch]
replicationcontrollers            rc                                          true         ReplicationController            [create delete deletecollection get list patch update watch]
resourcequotas                    quota                                       true         ResourceQuota                    [create delete deletecollection get list patch update watch]
secrets                                                                       true         Secret                           [create delete deletecollection get list patch update watch]
serviceaccounts                   sa                                          true         ServiceAccount                   [create delete deletecollection get list patch update watch]
services                          svc                                         true         Service                          [create delete get list patch update watch]
initializerconfigurations                      admissionregistration.k8s.io   false        InitializerConfiguration         [create delete deletecollection get list patch update watch]
mutatingwebhookconfigurations                  admissionregistration.k8s.io   false        MutatingWebhookConfiguration     [create delete deletecollection get list patch update watch]
validatingwebhookconfigurations                admissionregistration.k8s.io   false        ValidatingWebhookConfiguration   [create delete deletecollection get list patch update watch]
customresourcedefinitions         crd,crds     apiextensions.k8s.io           false        CustomResourceDefinition         [create delete deletecollection get list patch update watch]
apiservices                                    apiregistration.k8s.io         false        APIService                       [create delete deletecollection get list patch update watch]
controllerrevisions                            apps                           true         ControllerRevision               [create delete deletecollection get list patch update watch]
daemonsets                        ds           apps                           true         DaemonSet                        [create delete deletecollection get list patch update watch]
deployments                       deploy       apps                           true         Deployment                       [create delete deletecollection get list patch update watch]
replicasets                       rs           apps                           true         ReplicaSet                       [create delete deletecollection get list patch update watch]
statefulsets                      sts          apps                           true         StatefulSet                      [create delete deletecollection get list patch update watch]
tokenreviews                                   authentication.k8s.io          false        TokenReview                      [create]
localsubjectaccessreviews                      authorization.k8s.io           true         LocalSubjectAccessReview         [create]
selfsubjectaccessreviews                       authorization.k8s.io           false        SelfSubjectAccessReview          [create]
selfsubjectrulesreviews                        authorization.k8s.io           false        SelfSubjectRulesReview           [create]
subjectaccessreviews                           authorization.k8s.io           false        SubjectAccessReview              [create]
horizontalpodautoscalers          hpa          autoscaling                    true         HorizontalPodAutoscaler          [create delete deletecollection get list patch update watch]
cronjobs                          cj           batch                          true         CronJob                          [create delete deletecollection get list patch update watch]
jobs                                           batch                          true         Job                              [create delete deletecollection get list patch update watch]
certificatesigningrequests        csr          certificates.k8s.io            false        CertificateSigningRequest        [create delete deletecollection get list patch update watch]
leases                                         coordination.k8s.io            true         Lease                            [create delete deletecollection get list patch update watch]
events                            ev           events.k8s.io                  true         Event                            [create delete deletecollection get list patch update watch]
daemonsets                        ds           extensions                     true         DaemonSet                        [create delete deletecollection get list patch update watch]
deployments                       deploy       extensions                     true         Deployment                       [create delete deletecollection get list patch update watch]
```

Abbildung 22 Ausschnitt der API-Ressourcen eines Kubernetes-Cluster

Um die erstellte *Role* nun an einen bestimmten Account zu binden, muss ein *RoleBinding*, wie in Abbildung 23, erstellt werden:

```
1. kind: RoleBinding
2. apiVersion: rbac.authorization.k8s.io/v1alpha
3. metadata:
4.   name: read-pods-default-ns-rolebinding
5.   namespace: default
6. subjects:
7.   - kind: Group
8.     name: testusers
9.     apiGroup: rbac.authorization.k8s.io
10. roleRef:
11.   kind: Role
12.   name: read-pods-default-ns
13.   apiGroup: rbac.authorization.k8s.io
```

Abbildung 23 Rolebinding zur vorhergehenden Role

Der Aufbau ähnelt dem der *Role*, mit dem Unterschied, dass ein *RoleBinding* zusätzlich zu den Subjekten, auf die das *RoleBinding* angewendet werden soll, auch noch ein Attribut „roleRef“ besitzt, welches auf die zu bindende Rolle verweist. Dieser Name ist case-sensitive, das bedeutet, dass Groß- und Kleinschreibung beachtet werden müssen. Ein *RoleBinding* besitzt ebenfalls ein Namespace-Attribut.
Wird eine *ClusterRole* an ein *RoleBinding* gebunden, so wird die Rolle automatisch auf den Namespace des *RoleBinding* eingeschränkt. Somit ist ausgeschlossen, dass irrtümlicherweise Rollen (*ClusterRole*) clusterweit verfügbar sind, obwohl sie über ein „normales“ *RoleBinding* an ein Subjekt gebunden wurden. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/rbac/)

3.1.2.1 Verschiedene Authentifizierungsmöglichkeiten für Benutzer

Kubernetes bietet verschiedene Möglichkeiten, Benutzer zu authentifizieren.

3.1.2.1.1 Mutual-TLS

Nun besteht aber das Problem, dass Kubernetes keine User im technischen Sinne kennt; das bedeutet, dass Kubernetes keine interne Nutzerverwaltung bereitstellt. User im Kubernetes-Umfeld sind nur Zeichenketten, die mit einer Anfrage assoziiert sind. Es kennt zwar die Typen *User*, *Group* und *ServiceAccount*, allerdings müssen die Nutzer durch externe Methoden verwaltet werden. (3 S. 1083)

Authentifizierung an einem Kubernetes-Cluster ist über eine „Basic-Authentication“, über ein Token, über Zertifikate (mit Mutual-TLS) sowie OAuth/OpenID möglich. (3 S. 1092)
Dass eine Authentifizierung an einem Cluster dringend notwendig ist, ist trivial.
Allerdings sind weder die Token-basierte noch die Basic-Authentifizierung für einen produktiven Betrieb von Clustern zu empfehlen, da sie zum einen statisch sind (und somit nur bei einem Neustart neu eingelesen werden können) und zum anderen geheime Authentifizierungsinformation im Klartext enthalten. Dasselbe trifft auf die Verwendung von statischen Passwortdateien zu.
Somit eignen sich für einen sicheren Produktivbetrieb nur die Nutzung von Zertifikaten oder OAuth beziehungsweise OpenID.

Bei der Nutzung von Mutual-TLS wird, anders als im normalen „Internetverkehr“, nicht nur das Serverzertifikat beim Client geprüft, sondern der Server überprüft auch das Clientzertifikat, um dessen Identität zu bestätigen. (16 S. 8-9)
Nun wird Kubernetes so konfiguriert, dass Nutzer sich mit einem Zertifikat am kubeapi-Server authentifizieren können.
Hierfür muss zunächst ein Zertifikat für den Nutzer ausgestellt werden. Dafür wird ein privater Schlüssel für den RSA-Algorithmus benötigt. Das BSI empfiehlt für RSA eine Mindestschlüssellänge von 2000 Bit. (17) Das NIST empfiehlt Schlüssellängen von 2048 Bit und vertritt die Auffassung, dass diese bis 2030 ausreichende Sicherheit bieten. (18) Allerdings gehen die Meinungen über benötigte Schlüssellängen und vor allem über deren Sicherheit in der Zukunft stark auseinander, so ist das ANSSI der Meinung, dass 2048 Bit nur bis zum Jahr 2020 ausreichende Sicherheit bietet (19), ECRYPT vertritt die Position, dass selbst 3072 Bit Schlüssellänge nur bis zum Jahre 2028 Sicherheit bieten wird. (20) Somit muss abgewogen werden, wie lange die erzeugten Zertifikate gültig sein sollen und eine entsprechende Schlüssellänge gewählt werden. Bei kurzlebigen Zertifikaten (bis drei Jahre) ergeben sich durch die Verwendung von 2048 Bit Schlüssellänge nach aktuellem Stand der Rechenleistung und Auswertungen der oben genannten Untersuchungen keine sicherheitstechnischen Bedenken.
Zur Erzeugung der Schlüssel und Zertifikate wird in diesem Beispiel *openssl* verwendet. Andere Bibliotheken sind ebenfalls nutzbar, möglicherweise benötigen diese aber andere Befehle beziehungsweise Parameter.

Beim Erzeugen der Zertifikate empfiehlt es sich, ein Verzeichnis zu erstellen, in welchem alle erzeugten Zertifikate und Schlüssel gespeichert werden, damit die Befehle übersichtlich bleiben.

Im hier gezeigten Beispiel befindet sich die Konsole im Verzeichnis *Desktop/certificates*.

Als erstes wird der bereits erwähnte RSA-Schlüssel für den Testbenutzer *mueller* erzeugt:

openssl genrsa -out mueller.key 2048

Der nun erzeugte Schlüssel wird in einen Certificate-Signing-Request (CSR) eingebunden:

openssl req -new -key mueller.key -out mueller.csr -subj "/CN=mueller/O=developer"

Für Kubernetes ist das Setzen der Attribute CN (Common Name) und O (Organisation) sehr wichtig, da der CN als Benutzername und O als Gruppen interpretiert werden.

Nun kann der CSR mit Hilfe der CA des Kubernetes-Clusters signiert werden:

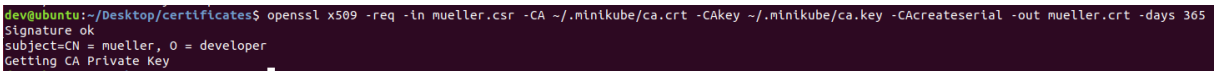


Abbildung 24 Signieren des CSR mit Hilfe der Certificate Authority

Der Vorgang wird für einen zweiten Testnutzer *maier* wiederholt, dieser besitzt jedoch als O-Attribut seines Zertifikats den Wert *administrator*. So kann die Separation beider Berechtigungen getestet werden.

Anschließend werden beide Nutzer in die kubeconfig-Datei exportiert:

kubectl config set-credentials username --client-certificate=certificate_file.crt --client-key=key_file.key

Die Variable *username* beschreibt den Namen, welcher später in der kubeconfig-Datei angezeigt werden soll.

Falls eine Person mehrere Nutzer benötigt, lässt sich jeder Nutzer einem eigenen Kontext zuweisen:

kubectl config set-context user-context-name --cluster=cluster --user=createduser

So kann später einfach mit *kubectl config use-context* zwischen verschiedenen Kontexten gewechselt werden.

Nun wird mit Testnutzer *mueller* versucht, die Liste der Pods im default-Namespace abzufragen: *kubectl get pods*. Da in der Rolle *development* nur der Namespace *development* erlaubt wurde, erhält man hier eine Fehlermeldung. Ergänzt man einen

erlaubten Namespace (in diesem Fall durch Anhängen von *-n development*), erhält man keine Fehlermeldung, sondern die Ausgabe der gewünschten Ressourcen.
Dies gilt für den Testnutzer *maier* analog zum Namespace administration.

Zertifikatsbasierte Anmeldung ist ein sehr guter Ansatz zur Authentifizierung, der in den meisten Teilen des Datenverkehrs im Internet genutzt wird. Im Kubernetes-Umfeld enthält diese Methode einen großen Nachteil: Kubernetes implementiert weder eine Certiciate Revocation List (CRL) noch das Online Certificate Status Protocoll (OCSP). Somit ist es zum jetzigen Stand nicht möglich, einen einmal bestätigten Certificate Signing Request (CSR) zu widerrufen. Dieser Sachverhalt hat maßgebliche Auswirkungen auf die Zugriffssicherung des Kubernetes-Clusters und muss bei der Auswahl einer geeigneten Authentifizierungsmethode unbedingt beachtet werden! Es gibt zwar Lösungen, um diese Funktionen behelfsweise selbst in Kubernetes einzubringen (21), diese basieren aber alle entweder auf der Verwendung der als veraltet deklarierten ABAC (Attribute Based Access Control) oder dem Erzeugen einer neuen Certificate Authority (CA). Dies hätte zur Folge, dass alle bereits bestehenden Zertifikate, die weiterhin gültig sein sollen, neu signiert werden müssten. Eine weitere Möglichkeit besteht darin, die Certificate Authority weiter zu nutzen und ein alleiniges Vertrauen in die Autorisierung über die RBAC zu haben und dem Nutzer dort alle Zugriffsrechte zu entziehen. Auch aus theoretischer Sicht haben Zertifikate einen Nachteil: Ihre Sicherheit beruht darauf, dass der private Schlüssel des Clients nie bekannt wird. Da dieser aber auf der lokalen Festplatte des Clients gespeichert ist, ist dies unter keinen Umständen anzunehmen. In Verbindung mit der fehlenden Unterstützung von CRL ist dies ein sehr gewichtiger Aspekt. Noch größer werden die Probleme bei der Nutzung von Zertifikaten, wenn eine größere Gruppe an Benutzern einzubinden ist, zum Beispiel alle Softwareentwickler eines Unternehmens, die auf ein Kubernetes-Cluster zugreifen dürfen sollen. In solch einem Fall ist der Verwaltungsaufwand für Zertifikate sehr hoch. Trotzdem sollte zumindest der Clusteradministrator ein gültiges Zertifikat besitzen, um sich im Falle eines Ausfalls des ID-Providers von OpenID Connect am Cluster authentifizieren zu können.

3.1.2.1.2 Token- und passwortbasierte Anmeldung

Es ist ebenso möglich, die Benutzer über eine Liste von Token zu authentifizieren. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/authentication/)

Hierfür muss dem kubeapi-Server ein Übergabeparameter bereitgestellt werden, der auf eine Datei mit folgender Struktur verweist:

token,user,uid,"group1,group2,group3"

Die Angabe der Gruppen ist optional, wenn mehrere Gruppen angegeben werden, müssen diese zwingend, wie hier dargestellt, in Hochkommata stehen. Eine einzelne Gruppe benötigt diese nicht. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/authentication/)
Um diese Datei zu nutzen, muss diese dem kubeapi-Server beim Start über den Parameter *--token-auth-file=AUTHFILE* der Speicherort der Datei mitgeteilt werden.
Die Datei im folgenden Beispiel sieht folgendermaßen aus:

sdfksjflöjsdkl2531,mueller,user1,developer
fjfdglädfkjhas9874,maier,user2,administrator

Anschließend wird der kubeapi-Server mit dieser Datei neu gestartet. Danach können unter Verwendung der Token Anfragen an die API gestellt werden.
Dieses Verfahren bietet allerdings einige Nachteile, hier ist vor allem die statische Funktionsweise anzuführen. Möchte man Gruppen von Nutzern ändern, ist immer ein Neustart des kubeapi-Servers nötig (das setzt einen Zugriff auf diesen voraus), was in Produktivsystemen nicht zur Debatte steht. Des Weiteren sind in dieser Datei Anmeldeinformationen im Klartext enthalten. Außerdem müssen die Token manuell erzeugt und an die Nutzer verteilt werden, sodass auch hier nicht von einer geeigneten Lösung gesprochen werden kann. Ein weiteres Argument gegen die Verwendung der beiden statischen Methoden liegt in der häufigen Nutzung einer Versionskontrolle. Somit wäre es naheliegend, die Token- oder Passwortdatei über diese zu verwalten, was zur Folge hat, dass mehrere Personen diese einsehen könnten.

Der Unterschied zwischen der statischen Passwortdatei und der Tokendatei liegt lediglich in der Authentifizierungsmethode im HTTP-Header: Die Passwortdatei nutzt eine Base64-Codierung und eine simple HTTP-Basic-Authentication (Base64-Username:Password), die Tokendatei nutzt ein eigenes Header-Feld (Authorization:

Bearer <Token>). (2 S. https://kubernetes.io/docs/reference/access-authn-authz/authentication/#authentication-strategies)

Für kleine Testsysteme kann diese Art der Authentifizierung jedoch geeignet sein, wenn wenige User und möglicherweise eine grobe Rechtestruktur verwendet werden und wenn die Einrichtung von Zertifikaten beziehungsweise externer Authentifizierung zu aufwendig ist. Sollten für Testsysteme eine Token- beziehungsweise Passwortauthentifizierung eingerichtet werden, ist darauf zu achten, dass für Test- und Produktivsysteme verschiedene Anmeldedaten verwendet werden.

3.1.2.1.3 Authentifizierung über OpenID

Eine weitere Möglichkeit der Authentifizierung bietet OpenID. Dies ist ein dezentrales Authentifizierungssystem, welches es dem Nutzer ermöglicht, sich mit einmaliger Anmeldung an einem Portal an verschiedenen anderen Portalen automatisiert anzumelden. Somit wird Single-Sign-On (SSO) unterstützt. Das aktuellste Protokoll nennt sich formell OpenID Connect, als Synonym wird aber häufig OpenID genutzt. OpenID Connect setzt auf OAuth2 auf, sodass auch diese Begriffe häufiger stellvertretend füreinander verwendet werden.

Um OpenID Connect konkret zu nutzen, ist ein ID-Provider nötig, der den Nutzern ihre IDs bereitstellt. Da OpenID dezentralisiert aufgebaut ist, gibt es viele verschiedene ID-Provider. Dazu zählen unter anderem öffentliche Provider wie GitHub, Amazon sowie Google und Microsoft. Es gibt außerdem private Dienste, die als ID-Provider dienen können, diese sind in nahezu jeder Programmiersprache verfügbar. (22) Der wohl populärste ID-Provider, der lokal gestartet werden kann, ist Keycloak. (23) Dieser eignet sich immer dann, wenn man nicht auf bestehenden, öffentlichen ID-Providern aufbauen möchte. Er kann auch in Verbindung mit bestehenden LDAP-Diensten genutzt werden. (23)

Die folgende Abbildung 25 veranschaulicht den Prozess der Nutzung von OpenID Connect mit Kubernetes:

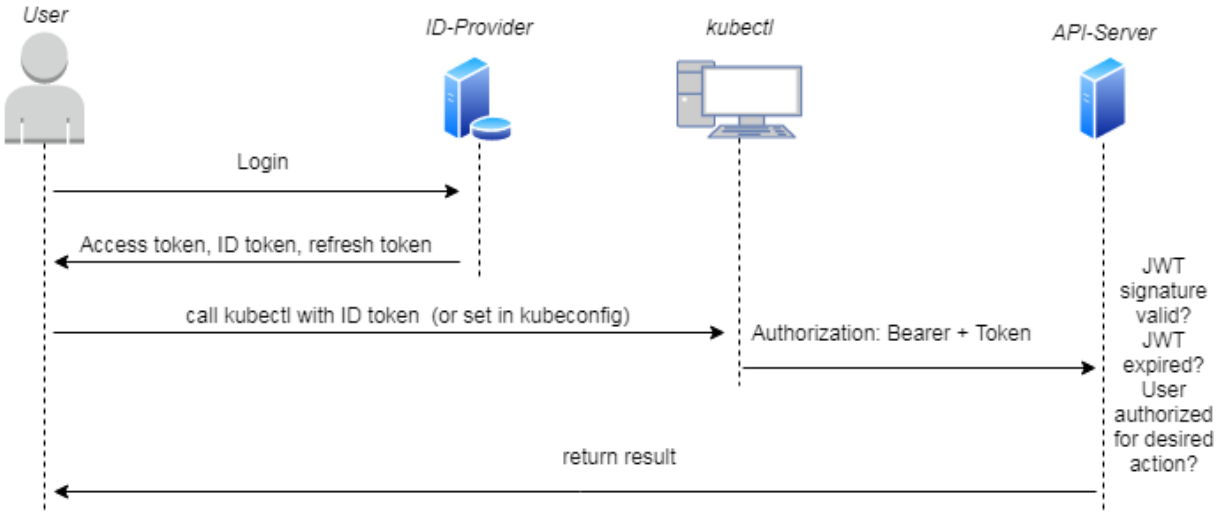


Abbildung 25 Ablauf der OpenID Connect Authentifizierung

Zunächst frägt der Nutzer seine OpenID Connect Token bei seinem ID-Provider ab. In der Praxis können dies wie bereits erwähnt verschiedene Systeme sein. Der ID-Provider gibt dem Nutzer ein JSON Web Token (JWT) mit verschiedenen Informationen zurück, von denen für den Nutzer nur das ID Token relevant ist. Dieses muss dann von *kubectl* zur Authentifizierung am API-Server genutzt werden.

Es sind zwar Angriffe gegen OpenID Connect bekannt (24), unter der Voraussetzung, dass zum Abruf und Transport des JSON Web Token TLS genutzt wird, kann OpenID Connect jedoch als sicher betrachtet werden.

Nachdem der API-Server das Token erhalten hat, prüft er dieses auf Gültigkeit. Falls das Token gültig ist, gilt der Nutzer als authentifiziert und es wird eine Prüfung der Autorisierung über die bereits erläuterten RBAC-Mechanismen durchgeführt. Anschließend wird die entsprechende Aktion ausgeführt (Operation durchführen oder nicht durchführen) und das Ergebnis dem Nutzer mitgeteilt. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/authentication/#users-in-kubernetes)

Um ein Kubernetes-Cluster mit OpenID Connect nutzen zu können, muss dem API-Server mitgeteilt werden, welcher der ID-Provider ist. Dies geschieht mit dem Parameter *oidc-issuer-url*. Zusätzlich werden die Parameter *--oidc-username-claim* und *--oidc-client-id* benötigt. Ersterer wird benötigt, um festzulegen, welches Attribut als Benutzername fungiert. Letzterer wird als eine Art Certificate Authority für Client-IDs benötigt: Alle ID-Tokens müssen für diese ID erzeugt werden. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/authentication/#openid-connect-tokens) (25) In Googles OAuth-Service entspricht der Parameter der Client-

ID des Projektes. Anschließend können Token, die vom ID-Provider ausgestellt wurden, genutzt werden, um sich am Kubernetes-Cluster zu authentifizieren.
Eine beispielhafte RBAC-Regel könnte zum Beispiel folgendermaßen aussehen (ausgehend davon, dass die Rolle *developer* bereits existiert):

```
1. kind: ClusterRoleBinding
2. apiVersion: rbac.authorization.k8s.io/v1alpha
3. metadata:
4.   name: developer-rolebinding-user-name_surename
5. subjects:
6.   - kind: User
7.     name: name.surename@example.com
8. roleRef:
9.   kind: ClusterRole
10.  name: developer
11.  apiGroup: rbac.authorization.k8s.io
```

Abbildung 26 ClusterRoleBinding an die Emailadresse des Nutzers

So wird der Nutzer, wie in Abbildung 26 abgebildet, an die Rolle gebunden und erhält die entsprechenden Privilegien.
Eine Anmerkung muss an dieser Stelle gemacht werden: Es ist im OpenID-Connect Standard vorgesehen, dass ein Feld für mögliche Nutzergruppen (*group claims*) bereitgestellt wird. (25) Es ist möglich, dieses Feld auf entsprechende Kubernetes-Gruppen zu mappen (über den Parameter *--oidc-groups-claim*) (2 S. https://kubernetes.io/docs/reference/access-authn-authz/authentication/), allerdings ist dafür ein ID-Provider notwendig, der dieses Feature unterstützt. Leider unterstützen nicht alle ID-Provider dieses Feature, daher wurde in der oben genannten Vorgehensweise auch das manuelle Mappen auf eine *RoleBinding* beschrieben. Rein technisch muss das automatisierte Mappen der Gruppen von der Authentifizierung des Nutzers abgegrenzt werden: Die Authentifizierung wird durch OpenID Connect übernommen, die Autorisierung durch OAuth2.
Es gibt diverse Tools, die die Nutzung von Open ID Connect mit Kubernetes vereinfachen, indem sie das Anpassen der *kubeconfig*-Datei automatisieren.[3]
OpenID Connect bietet zahlreiche Vorteile gegenüber anderen Verfahren: der größte Vorteil liegt in der Möglichkeit, Zugriffe zu entziehen. Dies wird zwar erst nach Ablauf

[3] https://github.com/gini/dexter
https://github.com/micahhausler/k8s-oidc-helper
https://github.com/negz/kuberos
https://github.com/frodenas/uaa-k8s-oidc-helper

der Time-to-live des Tokens wirksam, dieses lässt sich aber frei einstellen (Parameter *expires*, Zeitdauer in Sekunden) (26), sodass es möglich wäre, dass ein Token beispielsweise nur 24 Stunden lang gültig ist. Somit wäre eine komfortable Nutzerverwaltung mit ausreichender Sicherheit möglich. Ein zweiter Vorteil entsteht durch die sehr kleine Wahrscheinlichkeit von Impersonisation (*dt.: Nachahmung*). Schlussendlich trägt auch zur Sicherheit bei, dass keine Passwörter mehr verwaltet werden müssen.
Allerdings ergeben sich durch die Nutzung von OpenID Connect auch Nachteile. Der größte Nachteil besteht in der Bindung an eine externe Komponente zur Nutzerauthentifizierung. Bei einem Ausfall könnten die Token nur noch bis zum Ablauf ihrer Time-to-live genutzt werden. OpenID Connect hat in manchen Fällen den gleichen Nachteil wie alle anderen Authentifizierungsverfahren: Es ist nach wie vor eine Bindung der einzelnen Nutzer (mit ihrer Emailadresse, mit der sie sich am ID-Provider anmelden) an (*Cluster*)*RoleBinding* notwendig, um die Autorisierung zu prüfen, falls der ID-Provider keine Benutzergruppen unterstützt. Da aber alle gängigen Provider (außer Google) dieses Feature unterstützen, wiegt dieser Nachteil nicht sonderlich schwer.
Schlussendlich stellt OpenID Connect für ein produktives System in den meisten Fällen die komfortabelste und sicherste Möglichkeit bereit, um Authentifizierung, gepaart mit Autorisierung, durchzuführen.

3.1.2.2 RBAC für Service-Accounts

Wie bereits beschrieben, lassen sich mit RBAC Benutzerrechte durchsetzen. Ein simples Beispiel zeigt, dass dies auch für Serviceaccounts nötig sein kann.
In diesem Beispiel wird eine einfache Anwendung durch Remote-Code-Ausführung kompromittiert, wodurch möglicherweise Zugriff auf verschiedene Komponenten des gesamten Kubernetes-Cluster erlangt werden kann.
Als Beispielanwendung kommt ein von Google bereitgestelltes Gästebuch zum Einsatz.[4]
Diese Anwendung besteht aus drei Komponenten, einem Redis Master Slave, den verschiedenen Redis Slaves, sowie dem Frontend, welches die Sicherheitslücke

[4] https://kubernetes.io/docs/tutorials/stateless-application/guestbook/

beinhaltet. Das Javascript des Frontends enthält eine Sicherheitslücke, die Remote-Code-Ausführung erlaubt. (27) Es sendet drei Variablen an den PHP-Controller:

- cmd: Das Kommando, es ist auf *set* kodiert
- key: Der Redis-Key, er ist auf *messages* kodiert
- value: Der Wert beziehungsweise Text, der in das Textfeld der Website eingegeben wird.

Durch das OWASP-Tool *DirBuster* lassen sich Pfade und Dateien auffinden, die nicht durch sichtbare Links erreicht werden können. (28)
DirBuster wird heruntergeladen, lokal ausgeführt und findet die Datei *guestbook.php* sowie die Datei *status.php*. Ruft man die Datei *status.php* auf, sieht man eine Fehlermeldung mit dem Hinweis, dass der Redis-Command-Key gesetzt wird und die Seite neu geladen werden soll. Folgt man dieser Anweisung, erhält man den aktuellen Status: *All services operational*.
Dieses Verhalten lässt die Vermutung zu, dass die Schwachstelle Remote-Code-Ausführung erlaubt. Um dies zu testen, wird der PHP-Controller des Gästebuchs mit einer manipulierten Anfrage aufgerufen:

*guestbook:32500/guestbook.php?cmd=set&key=**command**&value=**ifconfig | grep -Eo 'inet (addr:)?([0-9]*\.){3}[0-9]*' | grep -Eo '([0-9]*\.){3}[0-9]*' | grep -v '127.0.0.1'***

Der Redis-Key wurde nun durch einen anderen Typ ersetzt, um zu testen, ob die Schwachstelle tatsächlich eine Remote-Code-Ausführung erlaubt. Der Befehl gibt die IP-Adresse der lokalen Maschine aus. Sollte Remote-Code-Ausführung möglich sein, wird die Ausgabe dieses Befehls auf der Seite *status.php* zu sehen sein. Wird diese Seite nun aufgerufen, findet sich dort als Ausgabe *192.168.9.131*, was in diesem Fall der lokalen IP-Adresse des Nodes entspricht, der den Pod bereitstellt. Nun wäre es im weiteren Verlauf möglich, zum Beispiel eine Reverse-Shell des Metasploit-Frameworks zu injizieren, indem ein lokaler Webserver gestartet wird, der diese bereitstellt, und der Controller des Gästebuchs mit dem entsprechenden CURL-Befehl aufgerufen wird. (29)
Somit könnten unter anderem Token und Zertifikate ausgelesen werden, was zur Vorbereitung von weiterführenden Angriffen dienen kann, zum Beispiel kompromittierte Docker-Container auf dem Cluster zu starten und dadurch weitere Teile zu übernehmen.

Role Based Access Control kann das Ausnutzen von Sicherheitslücken zwar nicht verhindern, hilft aber enorm, die möglichen Folgen zu minimieren sowie die Arbeit mit dem Kubernetes-Cluster im täglichen Betrieb zu vereinfachen.

*RoleBinding*s für Service-Accounts sehen genauso aus wie für User, mit dem Unterschied, dass das Attribut *kind* nun nicht mehr den Wert *User*, sondern den Wert *ServiceAccount* enthält. Wenn ein Pod keinen eigenen Service-Account hat, so verwendet Kubernetes den *default* Service-Account. (2 S. https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/)

Im gezeigten Beispiel wäre es zum Beispiel möglich, dem Service-Account zu verbieten, Pods zu erstellen. Somit wäre zwar immer noch ein Eindringen möglich, der Schaden würde sich aber auf das Auslesen von Informationen begrenzen.

Ein beispielhafter Service-Account für die verwendeten Redis-Komponenten könnte folgendermaßen aussehen:

Zunächst muss der Service-Account erstellt werden.

Hier wird davon ausgegangen, dass die Redis-Komponenten bereits in ihrem eigenen Namespace laufen, dazu später mehr. Diese Rolle kann dann über eine *RoleBinding* an den Service-Account gebunden werden (Abbildung 27):

```yaml
kind: ServiceAccount
apiVersion: v1
metadata:
  name: redis-restricted-serviceaccount
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1alpha
metadata:
  namespace: redis
  name: redis-read-pods
rules:
  apiGroups: [""]
  ressources: ["pods"]
  verbs: ["get", "list"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: redis-restricted-read-pods-rolebinding
  namespace: redis
subjects:
- kind: ServiceAccount
  name: redis-restricted-serviceaccount
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: redis-read-pods
  apiGroup: rbac.authorization.k8s.io
```

Abbildung 27 beschränkter ServiceAccount

Nun ist der ServiceAccount an die Rolle gebunden und erhält die entsprechenden Privilegien, allerdings muss dieser auch im dazugehörigen Deployment hinterlegt werden, wie in Abbildung 28 gezeigt:

```yaml
kind: Deployment
apiVersion: apps/v1 #for k8s before 1.9.0 use apps/v1beta2
metadata:
  name: redis-master
[...]
spec:
  serviceAccountName: redis-restricted-serviceaccount
  autoMountServcieAccountToken: true
```

Abbildung 28 Deployment mit angegebenem ServiceAccount

Anschließend kann dieses Deployment erstellt werden und alle Pods dieses Deployments nutzen den vermerkten Service-Account mit stark eingeschränkten Rechten.

Die Verwendung von RBAC erhöht die Sicherheit eines Kubernetes-Clusters immens. Es ist zum einen darauf zu achten, Benutzerrechte einzuschränken, allerdings ist es auch wichtig, die Rechte von Service-Accounts einzuschränken.
Es empfiehlt sich, bei RBAC das Least-Privilege-Prinzip anzuwenden, das bedeutet im Konkreten, dass ein Benutzer oder ein Service-Account alle Rechte, die er benötigt, aber so wenig Rechte wie möglich, erhält.

3.1.3 Namespaces

Namespaces bilden in Kubernetes eine Art logische Partitionierung, die aber nichts mit *Kernel-Namespaces* aus dem Docker-Umfeld gemein haben. (3 S. 904-905)
Die logische Partitionierung hat allerdings Grenzen, die im Nachfolgenden erläutert werden. Zunächst sind Namespaces über alle Nodes eines Clusters vorhanden und in jedem Namespace können eigene Regeln, Policies, Autorisierungsverfahren, Quotas und Ressourcenbeschränkungen gelten. Namespaces lassen sich ebenfalls mit Labels versehen. Dies ist zur Selektion in Network-Policies nötig.
Ohne zusätzliche Beschränkungen kann ein Service aus Namespace 1 jederzeit mit Services aus Namespace 2 kommunizieren. Dies ist sowohl direkt über IP-Adressen als auch unter der Notation *Service.Namespace* möglich. Jeder Namespace besitzt einen dazugehörigen DNS-Eintrag.
(2 S. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/)
Daraus resultiert, dass Namespaces keine sicherheitstechnische Funktion haben (und auch nicht sein sollen), sondern lediglich eine logische Trennung zur Unterstützung der Clusterbetreiber ermöglichen. Sollen in Namespaces netzwerktechnische Beschränkungen vorgenommen werden, müssen *NetworkPolicies* verwendet werden.
Es ist außerdem möglich, dass Pods aus verschiedenen Namespaces auf denselben Nodes landen, hierbei sieht man deutlich, dass die Partitionierung durch Namespaces rein logischer Natur und keinesfalls physisch ist.
Der wichtigste Punkt bei der Verwendung von Namespaces stellt deren Schutz dar: Jeder Namespace muss durch RBAC oder andere Autorisierungsverfahren vor versehentlicher Löschung geschützt sein. **Das Löschen eines Namespaces entfernt alle Objekte, die dieser enthält**! (3 S. 905)

Google empfiehlt die eigene Verwendung von Namespaces nur dann, wenn viele Nutzer das Cluster nutzen beziehungsweise ein Cluster für mehrere Teams oder Projekte zur Verfügung gestellt wird.
(2 S. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/)
Diese Empfehlung wird jedoch nicht wirklich beachtet, viele Nutzer wenden Namespaces auch zur logischen Trennung von Teilen ihrer Anwendungen in Kubernetes an.
Standardmäßig sind in Kubernetes drei Namespaces enthalten: *default*, *kube-public* und *kube-system*.
Der *default*-Namespace bildet einen Container für alle Objekte (Pods, Deployments, ReplicaSets etc.), die erzeugt werden und bei deren Erzeugung kein spezifischer Namespace deklariert wird.
In neueren Versionen existiert auch ein Namespace *kube-public*. Objekte in diesem Namespace können von **jedem** gelesen werden. Hierzu zählen auch nicht-authentifizierte Nutzer! Daher kann von einer Nutzung dieses Namespaces beziehungsweise von der Erzeugung von Objekten in diesem aus sicherheitstechnischer Sicht nur abgeraten werden.
Der Namespace *kube-system* dient zur Kapselung der Systemdienste in Kubernetes. Er separiert den System-Pod vom Rest des Kubernetes-Clusters und ist durch einen Admission Controller, zumindest rudimentär, vor versehentlicher Löschung geschützt. (3 S. 906)
Wenn Namespaces genutzt werden, um ein Kubernetes-Cluster *multi-tenant-fähig* (mehrere Mandanten nutzen dasselbe Cluster) zu machen, müssen unbedingt weitere Sicherheitsmaßnahmen ergriffen werden. (3 S. 907) Zusätzlich kommt hier ein erhöhter Verwaltungsaufwand durch viele Regeln und Policies zum Tragen. Daraus folgen unweigerlich hohe Bedenken bezüglich der Sicherheit des Clusters.
Interessant anzumerken ist, dass mache Ressourcen keinem Namespace zugeordnet sind. Über den API-Server lassen sich alle Ressourcen abfragen, die sich in einem oder keinem Namespace befinden:
kubectl api-resources –namespaced=true/false
Bei Betrachtung der Antwort stellt man fest, dass Namespaces, ClusterRoles und die dazugehörigen ClusterRoleBindings, Nodes, Speichermedien sowie andere Policies und Konfigurationsobjekte keinen Namespace besitzen. Dies begründet sich in der

Tatsache, dass diese Objekte alle global verfügbar sein müssen und somit auch keinem Namespace zugehörig sein dürfen.

Beim praktischen Einsatz von Namespaces stellen sich vor allem Fragen nach der Benennung, hier empfiehlt es sich, die Regelungen der eigenen Organisation anzuwenden. Allgemeine Richtlinien nutzen in der Regel die Funktion und den Standort als Namen für einen Namespace, ähnlich wie LDAP-Strukturen. Es sollte außerdem zur Reduzierung der Komplexität und möglicherweise Verwirrung darauf geachtet werden, dass keine nicht benötigten Namespaces angelegt werden und bei Bedarf nicht mehr benötigte Namespaces gelöscht werden. (3 S. 909)

Um eigene Namespaces nutzen zu können, müssen diese zunächst angelegt werden. Dies ist entweder über eine Datei möglich, dort können auch direkt mehrere Labels spezifiziert werden, oder direkt über den Befehl *kubectl create namespace namespace_name*. In der zweiten Variante können aber keine Labels direkt beim Erzeugen hinzugefügt werden, dies ist nur nachträglich mit *kubectl edit namespace* möglich. (3 S. 910) Nun kann der erstellte Namespace in Deployments, Roles etc. genutzt werden. Anschließend kann der Namespace in anderen Ressourcen verwendet werden.

Im Laufe der Zeit hat die schon erwähnte Multi-Tenancy (*dt.: Mehrmandantenfähigkeit*) in das Kubernetes-Umfeld Einzug gehalten. Die Nachteile der herkömmlichen Bereitstellungsart von Kubernetes-Clustern liegen auf der Hand: Jedes Team beziehungsweise jede Applikation benötigt in diesem Ansatz ein eigenes Cluster. Dementsprechend steigt der Verwaltungsaufwand mit jedem dieser Cluster, da jeder neue Mandant ein neues beziehungsweise eigenes Cluster benötigt. Zusätzlich erhöhen sich die Kosten (entweder physische Hardware oder Lizenzkosten für virtuelle Maschinen), denn jedes Cluster muss seine eigenen Kubernetes-Basiskomponenten besitzen. Allerdings liefert dieser Ansatz „kostenlose“ Sicherheit, da verschiedene Teams oder Anwendungen sicher voneinander getrennt (entweder echt physikalisch oder durch virtualisierte Maschinen) sind. Eine detaillierte Analyse dieses Sachverhalts findet sich in Kapitel 5.

Unter Multi-Tenancy wird im Kubernetes-Umfeld die konkrete Trennung zwischen einzelnen Mandanten im selben Kubernetes-Cluster verstanden. Dieses Feature ist in der GKE (Google Kubernetes Engine) implementiert. (30)

Bei einem Multi-Tenant-Ansatz werden ein oder mehrere Namespaces pro Mandant genutzt, jedoch wird ein Namespace nie mehr als einen Mandanten haben. (31) Im Vortrag von David Oppenheimer werden drei Modelle erläutert, von welchen für FIRMA nur das Erste (*Enterprise-Cluster*) (31) relevant ist: mehrere Teams teilen sich dasselbe Cluster. Dieses lässt sich wie in Abbildung 29 schematisch darstellen:

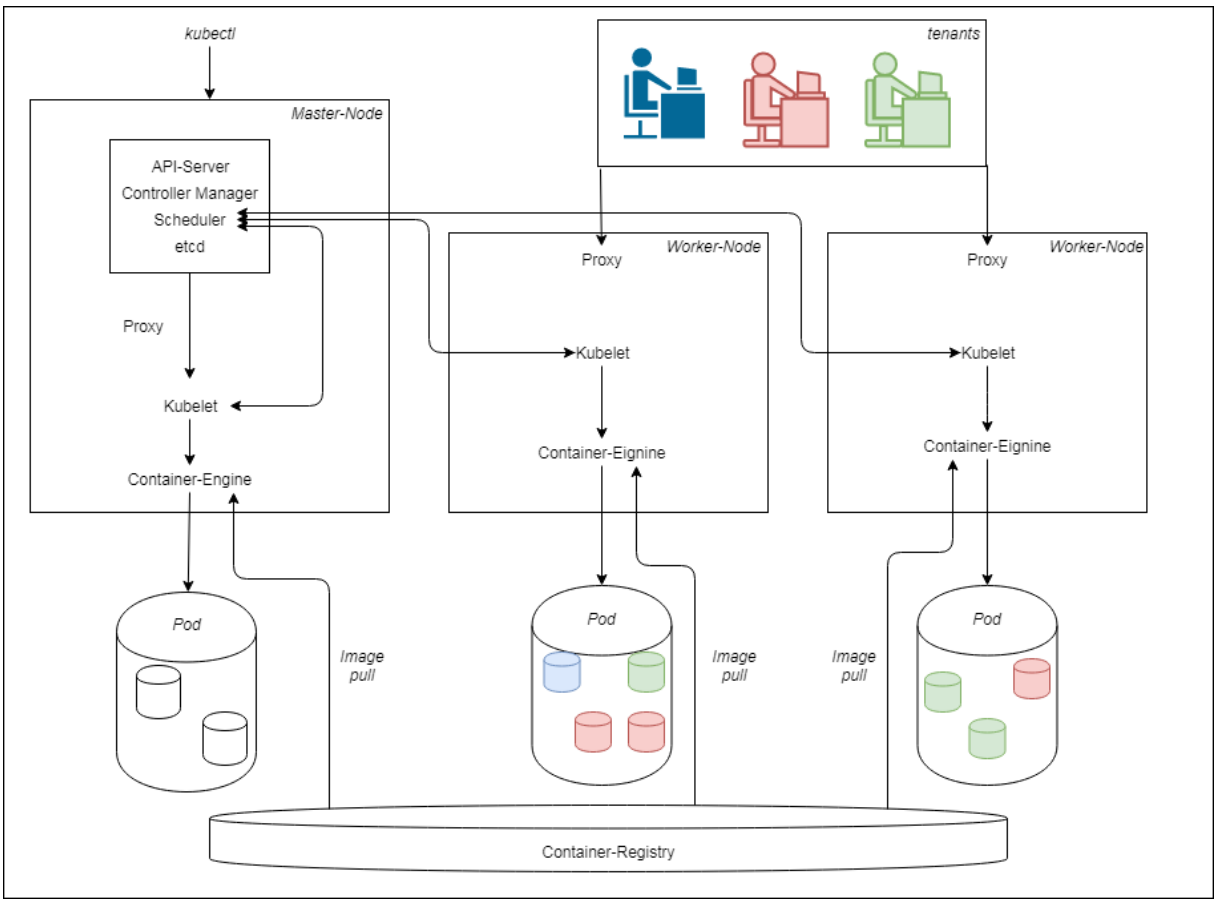
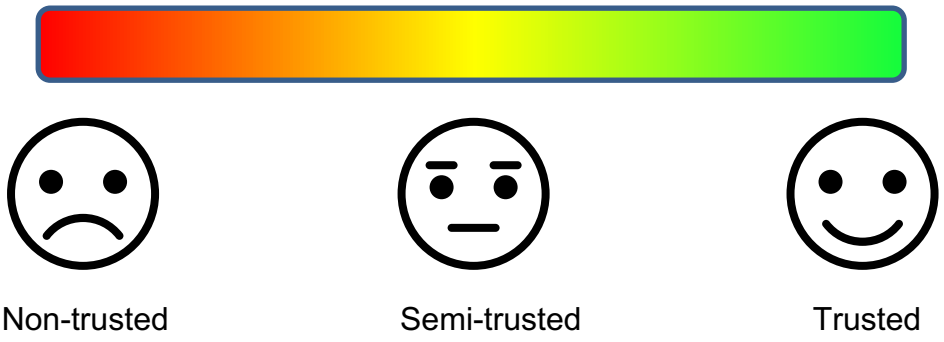


Abbildung 29 Schematische Darstellung eines Multi-Tenancy-Clusters

Die Tenants (*dt.: Mandanten*) wären im Falle der FIRMA Gruppe die einzelnen Teams. Diese teilen sich die Kubernetes-Basiskomponenten, können aber auf dem Cluster ihre eigenen Ressourcen erstellen. In Abbildung 29 wird dies durch die farbig markierten Container in den einzelnen Pods angedeutet. Die wichtigste Anforderung an solch ein Cluster besteht darin, zu gewährleisten, dass unterschiedliche Mandanten nur auf ihre Ressourcen zugreifen dürfen; es sei denn, es ist ausdrücklich gewünscht, dass ein gegenseitiger Zugriff möglich sein soll.

Um die Sicherheit von Komponenten in einem von mehreren Mandanten genutzten Cluster zu bewerten, ist eine Skala mit drei Stufen gängig:



Die Einstufung auf der Skala ist folgendermaßen zu verstehen: Trusted bedeutet, dass das Objekt (Code, Plugin etc.) aus einer geprüften Quelle kommt und mit vertrauenswürdigen Komponenten erstellt beziehungsweise betrieben wird. Sowohl das Produkt als auch alle zur Erstellung des Produkts verwendeten Produkte müssen vertrauenswürdig und geprüft sein. Bei der Einstufung Semi-trusted muss nur das Produkt selbst, nicht jedoch die zur Erstellung benötigten Produkte geprüft und vertrauenswürdig sein. Dies wird wohl auf einen Großteil der in der Praxis eingesetzten Ressourcen zutreffen. Die Einstufung Non-trusted steht für bösartige Software oder Software mit bösartigem oder unbekanntem Inhalt.

Nun muss jede Komponente an Hand dieser Skala bewertet und aus der resultierenden Bewertung abgeleitet werden, wie stark die Komponente eingeschränkt werden muss. So könnte es zum Beispiel vorkommen, dass ein selbst implementiertes Logging-Framework den Status *trusted* erhält und keine starke Einschränkung erhält, da schädliche Funktionen ausgeschlossen werden können.

Die erste Einstufung in einem Kubernetes-Cluster mit mehreren Mandanten muss allerdings für die Mandanten selbst vorgenommen werden. In einem Unternehmen wird dies in den meisten Fällen in der Kategorie *Semi-trusted* erfolgen, ein geregeltes Arbeitsverhältnis besteht. Bei „nicht kontrollierbaren“ Mandanten, zum Beispiel Nutzern, wenn Kubernetes zur Bereitstellung von SaaS genutzt werden soll, müssen mehr beziehungsweise größere Schutzmaßnahmen ergriffen werden. Fremde Nutzer fallen gemäß der Bewertungsskala in die Kategorie *not-trusted*.

In einem Modell, in welchem sich mehrere Mandanten ein Kubernetes-Cluster teilen, empfiehlt es sich, einen zentralen Cluster-Administrator und für jeden Namespace

einen eigenen Namespace-Administrator zu erstellen. So kann jeder Namespace-Administrator seinen Namespace verwalten und Zugriff darauf gewähren (dies entspricht der Abbildung eines sich selbst verwaltenden Teams), während der Cluster-Administrator die Namespace-Administratoren ernennt.
Des Weiteren ist eine Trennung des Netzwerkverkehrs nötig. So soll innerhalb eines Namespaces jeder Datenverkehr zulässig, zwischen Namespaces jedoch nur explizit erlaubter Datenverkehr möglich sein.
Diese Anforderungen müssen mit den in dieser Arbeit aufgezeigten Mechanismen umgesetzt werden. Kubernetes an sich stellt keine Funktionen zur Mehrmandantenfähigkeit bereit, es gibt aber Top-Level-Services wie die Google Kubernetes Engine, welche solche Features versuchen bereitzustellen. (30)
Zusammenfassend lässt sich festhalten, dass Namespaces ein gutes Hilfswerkzeug zur logischen Trennung von Ressourcen sind. Sie bieten allerdings keine Sicherheit, diese kann nur unter der Zuhilfenahme von anderen Sicherheitsmechanismen erreicht werden. Von der Nutzung eines mehrmandantenfähigen Kubernetes-Clusters kann aus aktueller sicherheitstechnischer Sicht nur abgeraten werden, da die Mehrmandantenfähigkeit immer auf der Trennung durch Namespaces aufbaut. Diese wird natürlich durch andere Sicherheitsmechanismen ergänzt, allerdings ist diese Sicherheit nicht mit der einer echten physikalischen Trennung zu vergleichen. Weiterhin ist es zwingend notwendig, dass alle Mandanten die gleichen Kubernetes-Basiskomponenten benutzen, dazu zählen insbesondere der API-Server, der Controller Manager sowie das gesamte DNS. Ein weiteres Problem bei solch einem Ansatz stellt die Kubernetes-Version dar: Alle Mandanten würden die gleiche Kubernetes-Version nutzen, dies ist aber möglicherweise nicht möglich, da es bei sehr kurzen Release-Zyklen (wie im Kubernetes-Umfeld) des Öfteren zu Kompatibilitätsproblemen bei Plugins führen kann. Wird also ein hohes Maß an Sicherheit benötig, sollte die Wahl auf ein eigenes Kubernetes-Cluster für jeden Mandanten fallen. Dies wird auch in gängiger Fachliteratur so empfohlen. (3)

3.1.4 Network-Policies

Network-Policies sind seit Kubernetes in der Version 1.7 als *stable* markiert und bieten Sicherheit auf Netzwerkebene.

(3 S. 1071) (2 S. https://kubernetes.io/docs/concepts/services-networking/network-policies/)

Sie regulieren, wie Pods miteinander kommunizieren dürfen. Standardmäßig akzeptiert jeder Pod alle Verbindungen (Pods sind standardmäßig *non-isolated*) (2 S. https://kubernetes.io/docs/concepts/services-networking/network-policies/) und sind im Wesentlichen notwendig, da Namespaces keine Separation auf Netzwerkebene darstellen.

Network-Policies benötigen ein kompatibles Plugin, um zu funktionieren. Sie sind keine in Kubernetes enthaltene Funktionalität! (3 S. 1072) Leider ist es möglich, Network-Policies zu erstellen und zum API-Server zu senden, auch wenn kein kompatibles Plugin vorhanden ist. Als Antwort auf das Erstellen einer Network-Policy ohne entsprechendes Plugin erhält man als Antwort:

networkpolicy.networking.k8s.io/networkpolicy-name created.

Es ist also möglich, Network-Policies zu erstellen, ohne dass das Kubernetes-Cluster diese verwerten kann. Die Funktionsweise sowie der Einsatzzweck von Network-Policies ist in Abbildung 30 zu sehen:

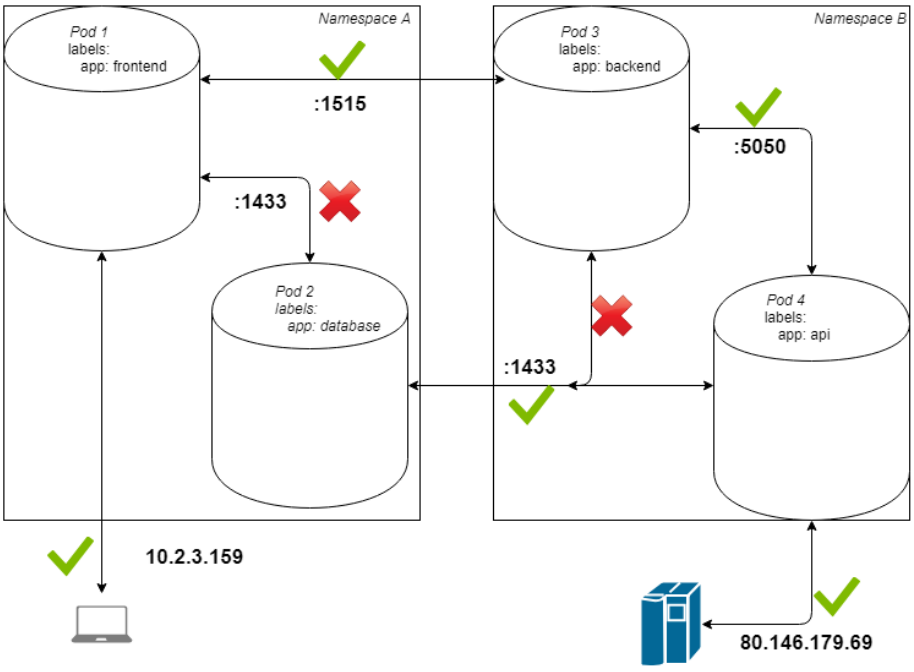


Abbildung 30 Funktionsweise von Network-Policies im Beispielszenario

Sie regeln die Kommunikation zwischen Pods und zwischen externen Clients und Pods. Es ist sowohl die Eingangs- (*Ingress*) als auch die Ausgangsrichtung (*Egress*) regulierbar. Wie aus dem Schaubild hervorgeht, arbeiten Network-Policies mit

sogenannten Labeln, genauer gesagt, sie nutzen Selektoren, um Ressourcen mit bestimmten Labeln auszuwählen.
(2 S. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)
Es sind aber auch Beschränkungen hinsichtlich IP-Adressen und Ports möglich.

Das Szenario in Abbildung 30 beschreibt eine typische Microservice-Architektur, bestehend aus vier Komponenten:

- Ein Frontend, das von allen IP-Adressen zugänglich sein soll.
- Eine Datenbank, die nur aus dem internen IP-Adressbereich, nur auf einem spezifischen Port und nur von spezifischen Labeln zugänglich sein soll. Es soll nicht möglich sein, dass Backend und Frontend direkt mit der Datenbank kommunizieren, der Zugriff soll ausschließlich über den Service API möglich sein.
- Ein Backend, das Anfragen des Frontends entgegennimmt und zu API-Anfragen zusammenfasst, die Ergebnisse konvertiert und zurückgibt.
- Eine API, die die fest definierten Datenbankabfragen bereitstellt und diese abarbeitet und die Ergebnisse zurückgibt. Diese soll sowohl aus dem internen IP-Adressbereich als auch aus dem externen IP-Adressbereich zugänglich sein. Es soll aber einen Adressbereich geben, von dem aus kein eingehender Datenverkehr und eine explizite IP-Adresse, zu der kein ausgehender Datenverkehr erlaubt sein soll.
- In den Namespaces A und B soll keine Kommunikation erlaubt sein, die nicht ausdrücklich zugelassen ist.

Vor der konkreten Anwendung von Network-Policies auf dieses Szenario sollen zunächst die Grundlagen erläutert werden. Hierfür wird noch einmal daran erinnert, dass ein leerer Selektor (*{}*) alle Ressourcen auswählt! (32)
Prinzipiell besteht eine Network-Policy aus drei Teilen (siehe Abbildung 31):

- Welche Ressourcen sollen ausgewählt werden?
- Für welche Richtungen soll die Regel gelten, also eingehender oder ausgehender Datenverkehr)?
- Regeln für die Richtungen: Wer kann sich mit der Ressource verbinden und wohin darf sich die Ressource verbinden?

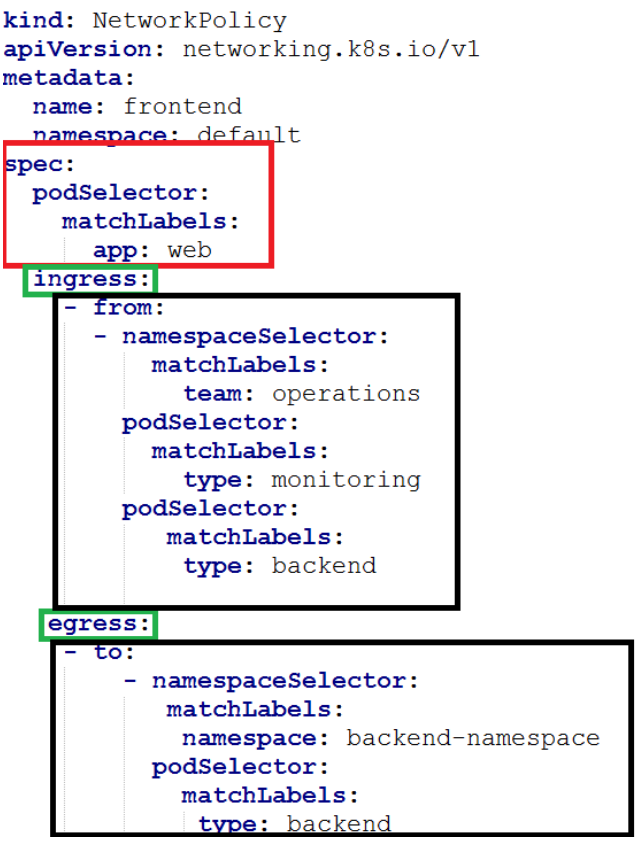


Abbildung 31 Struktur einer Network-Policy

Diese Network-Policy erlaubt eingehenden Datenverkehr aus allen Ressourcen, die den Namespace mit dem Label *team: operations*, Pods die das Label *type: monitoring* haben sowie allen Pods, die das Label *type: backend* haben. Gleichzeitig wird ausgehender Datenverkehr nur in die Namespaces mit dem Label *namespace: backend* und die Pods mit dem Label *type: backend* erlaubt.

Es ist möglich, ein zusätzliches Attribut *policyTypes* anzugeben, welches beschreibt, welche Regeln vorhanden sind. Falls dieses nicht angegeben wird (wie in diesem Kapitel), wird automatisch eine Ingress-Regel erstellt (die ein leeres Array beinhaltet, falls keine Ingress-Regeln definiert sind. Ein leeres Array würde dann der Darstellung [ ] entsprechen). Egress-Regeln werden nur dann erstellt, wenn auch tatsächlich Regeln vorhanden sind. Wenn also explizit nur Egress-Regeln erstellt werden sollen, muss dieses Attribut gesetzt werden.

(2 S. https://kubernetes.io/docs/concepts/services-networking/network-policies/)

Generell gilt, dass das Weglassen von *ingress* oder *egress* mit *ingress/egress: [ ]* gleichzusetzen ist, also keine erlaubten Verbindungen.

Zur Verfeinerung der Regeln sind auch IP-Adressbereiche (unter Verwendung von CIDR-konformen Angaben: IP-Adresse/Subnetzmaske) sowie Protokolle und Ports möglich. (2 S. https://kubernetes.io/docs/concepts/services-networking/network-policies/)
Eine einfache Möglichkeit, um Netzwerkverkehr nur innerhalb eines Namespaces zuzulassen, wäre folgende:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  namespace: secondary
  name: deny-from-other-namespaces
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
  egress:
  - to:
    - podSelector: {}
```

Abbildung 32 Network-Policy erlaubt nur Datenverkehr innerhalb des Namespaces

Hier werden über den leeren Selektor alle Pods des Namespaces ausgewählt und damit der Datenverkehr sowohl eingehend als auch ausgehend ermöglicht. Zugriffe aus und in andere Namespaces sind nicht möglich, versucht man zum Beispiel einen HTTP-Server in Namespace *secondary* aus dem *default*-Namespace anzufragen, erhält man als Antwort ein Timeout der Verbindung, da das Netzplugin die Verbindung blockiert. Network-Policies terminieren keine Verbindungen, das bedeutet im Konkreten: wird eine Network-Policy auf dem Cluster erstellt, wirkt diese nur auf neue Verbindungen und beendet keine schon bestehenden Verbindungen. (32)
Es gilt das Prinzip, dass nur Regeln erstellt werden können, die etwas erlauben. Explizite Verbote sind (mit Ausnahme des *except*-Attributes) nicht möglich und müssen durch „nicht-erlauben“ umgesetzt werden, wie in Abbildung 32 dargestellt. Kubernetes verfolgt hier sozusagen das Whitelisting-Prinzip. Dieses Prinzip gilt auch für untergeordnete Attribute, wie zum Beispiel Ports und IP-Bereiche. Wird beispielsweise kein Port spezifiziert, werden alle Ports zugelassen. Sobald jedoch mindestens ein

Port angegeben wird, werden nur noch diese angegebenen zugelassen. Alle anderen werden blockiert. Es ist aktuell nur möglich, *ContainerPorts* (also Ports von Pods) zu reglementieren, die Beschränkung von ServicePorts ist aktuell noch nicht in allen Plugins in Kubernetes möglich. (32) (33) Im Zweifel muss auf diese Funktionalität verzichtet werden, daher wird auch im nachfolgenden Beispiel darauf verzichtet.
Bei der Erstellung und Anwendung von Network-Policies ist zu beachten, dass diese verodert werden: Regeln (das sind sowohl verschiedene Dateien **als** auch einzelne Listenelemente innerhalb einer Datei) werden miteinander verodert und auf Basis dieses Ergebnisses dann bestimmt, ob Datenverkehr erlaubt wird oder nicht. (32)
Listenelemente werden verundet, das bedeutet: stehen in einer „Ebene“ mehrere Elemente ohne Spiegelstrich (also nicht als neues Listenelement), werden diese verundet. Dies ist später in Abbildung 35 zu sehen.
Entgegen der Intuition bedeutet keine Network-Policy jedoch leider nicht, dass jeder Datenverkehr blockiert wird; im Gegenteil: **ohne mindestens eine Network-Policy wird standardmäßig jeder Datenverkehr zugelassen**. Falls eine Network-Policy existiert, werden alle nicht erlaubten Verbindungen im entsprechenden Namespace blockiert. **Es findet ein Wandel von „implizit erlaubt“ hin zu „implizit alles verboten“ statt.**
Außerdem ist zu beachten, dass Network-Policies immer einem Namespace zugehörig sind. Dadurch ist es notwendig, alle Policies in alle benötigten Namespaces zu bringen, sonst wird die Policy nur auf den aktuellen Namespace angewendet. Daraus folgt auch, dass ein *PodSelector* immer nur Pods aus dem aktuellen Namespace selektieren kann. Hierfür können *NamespaceSelectors* verwendet werden, diese selektieren Namespaces anhand deren Labels, wie in Abbildung 31 gezeigt.
Da Kubernetes standardmäßig keine Network-Policies verarbeiten kann und dafür wie bereits erwähnt Plugins nötig sind, lohnt es sich, einen Blick auf deren Funktionsweise mit Network-Policies zu richten. Die Firma *Innovex* hat die Network-Plugins *Weave Net* und *Calico* an Hand des GitHub-Repositories von Ahmet Balkan (33) (mit vielen Beispielen für Network-Policies) getestet. (34)
Innovex hat diese Beispiele automatisiert getestet und kommt zu dem Schluss, dass es Unterschiede zwischen der theoretischen Network-Policy und der tatsächlichen Umsetzung der verschiedenen Plugins gibt. Es ist also sehr empfehlenswert, Network-Policies auf Funktion zu testen. (34) Bei FIRMA (und auch anderen Unternehmen) würde es sich in diesem Fall anbieten, sich auf ein Plugin festzulegen und ein

gemeinsames, teamübergreifendes Knowhow bezüglich dieses Plugins zu erlangen und gemeinsame Testmöglichkeiten zu erarbeiten.

Das in Abbildung 30 gezeigte Setup soll nun in einem Kubernetes-Cluster praktisch umgesetzt werden, um ein Verständnis für die praktische Funktionsweise von Network-Policies zu erlangen.
Hierfür ist zunächst ein Netzwerk-Plugin nötig, welches diese Funktionalität unterstützt. Hierfür fällt die Wahl auf *Calico.* Dies begründet sich darin, dass Calico kein Overlay-Netzwerk wie zum Beispiel Flannel bildet, sondern auf Layer drei des OSI-Modells arbeitet und ein Border-Gateway-Protocol nutzt. Dadurch ist es sehr performant. Weiterhin harmoniert es sehr gut mit Istio, welches in Kaptiel 3.1.6 zum Einsatz kommt.
Zunächst wird die *default*-Regel aus Abbildung 32 auf beide Namespaces angewandt:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace-a
  name: deny-all-in-ns-a
spec:
  podSelector:{}
  ingress: []
```

Abbildung 33 Verbieten des Datenverkehrs in Namespace-A

Diese Formulierung der Regel stellt eine Kurzform dar. Dies wird analog auf Namespace-B angewendet.
Der Datenverkehr zwischen Frontend und Datenbank ist somit implizit unterbunden. Nun kann der Zugriff von außen auf das Dashboard erlaubt werden:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace-a
  name: allow-frontend-externalips
spec:
  podSelector:
   app: frontend
  ingress:
    - from:
      - ipBlock:
          cidr: 0.0.0.0/0
      ports:
       - protocol: TCP
         port: 443
  egress:
    - to:
      - ipBlock:
          cidr: 0.0.0.0/0
      ports:
       - protocol: TCP
         port: 443
```

Abbildung 34 Anwenden von CIDR-Bereichen in Network-Policies

Hier wird ein CIDR-Bereich mit einem Port und Protokoll kombiniert. Der Zugriff ist von allen IP-Adressen (0.0.0.0/0 umfasst in CIDR-Notation alle IP-Adressen von 0.0.0.0 bis 255.255.255.255) unter Port 443 möglich. Über das *except*-Attribut wäre es möglich, einzelne Bereiche auszuschließen.

Nun kann der Namespace-übergreifende Datenverkehr zwischen Frontend und Backend erfolgen:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace-a
  name: allow-frontend-to-backend
spec:
  podSelector:
   app: frontend
  ingress:
    - from:
       - namespaceSelector:
          matchLabels:
            name: namespace-b
        podSelector:
          matchLabels:
            app: backend
   egress:
    - to:
       - namespaceSelector:
          matchLabels:
            name: namespace-b
        podSelector:
          matchLabels:
            app: backend
```

Abbildung 35 Kombination zweier Selektoren

Hier werden zwei Selektoren kombiniert: Es wird jeder Datenverkehr erlaubt, der folgende Bedingungen erfüllt:

- Der Pod hat das Label *app: backend,*
- Der Pod befindet sich in einem Namespace mit dem Label *name: namespace-b.*

Bei Kombinationen von Selektoren muss noch stärker auf die korrekte Formatierung von YAML-Dateien geachtet werden! Ein Spiegelstrich vor dem *PodSelector*-Feld würde dazu führen, dass diese Regel nach den Spezifikationen folgendes erlaubt:

- Datenverkehr mit jedem Namespace mit dem Label *name: namespace-b,*
- Datenverkehr mit jedem Pod, der das Label *app: backend* besitzt, **unabhängig davon, in welchem Namespace er sich befindet**.

Auch bei dieser Regel wäre eine Angabe von Ports und Protokollen möglich.

Jetzt kann in Namespace die Datenbank darauf vorbereitet werden, Datenverkehr aus dem Namespace B des Backends entgegenzunehmen und Antworten an diesen zu versenden:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace-a
  name: allow-database-to-api
spec:
  podSelector:
   app: database
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            name: namespace-b
        podSelector:
          matchLabels:
            app: api
   egress:
    - to:
      - namespaceSelector:
          matchLabels:
            name: namespace-b
        podSelector:
          matchLabels:
            app: api
```

Abbildung 36 Verunden innerhalb einer Regel einer Network-Policy

Hier wurde die Regel zusätzlich noch mit einem Port beschränkt. Der restliche Teil folgt analog der vorhergehenden Regel aus Abbildung 35.

Nun muss noch das Backend darauf vorbereitet werden, Anfragen des Frontends und der API entgegen zu nehmen und zu beantworten:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace-b
  name: allow-backend-to-frontend-and-api
spec:
  podSelector:
   app: backend
  ingress:
    - from:
      - namespaceSelector:
        matchLabels:
          name: namespace-a
       podSelector:
        matchLabels:
          app: frontend
       ports:
        - port: 1515
          protocol: TCP
    - podSelector:
        matchLabels:
          app: api
       ports:
         - port: 5050
           protocol: TCP


   egress:
    - to:
      - namespaceSelector:
        matchLabels:
          name: namespace-a
       podSelector:
        matchLabels:
          app: frontend
       ports:
        - port: 1515
          protocol: TCP
      - podSelector:
        matchLabels:
          app: api
       ports:
        - port: 5050
          protocol: TCP
```

Abbildung 37 Network-Policy für das Frontend

Hier wurden zunächst zwei Regeln spezifiziert und beide mit den entsprechenden Ports spezifiziert.

Schlussendlich muss noch der Zugriff von außen auf die API ermöglicht werden, hier gilt es noch die IP-Adresse 192.36.45.3 vom Datenverkehr auszuschließen:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace-b
  name: allow-external-api
spec:
  podSelector:
   app: api
  ingress:
    - from:
      - ipBlock:
         cidr:0.0.0.0/0
         except:
          - 192.36.45.3/32
        ports:
          - port: 443
            protocol: TCP
   egress:
    - to:
      - ipBlock:
         cidr:0.0.0.0/0
         except:
          - 192.36.45.3/32
        ports:
          - port: 443
            protocol: TCP
```

Abbildung 38 Network-Policy für die API

In produktiven Systemen sollte in jedem Namespace der DNS-Service freigegeben werden.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: namespace
  name: allow-dns
spec:
  podSelector:
    matchLabels:
      app: foo
  ingress:
  - ports:
    - port: 53
      protocol: UDP
    - port: 53
      protocol: TCP
  egress:
  - ports:
    - port: 53
      protocol: UDP
    - port: 53
      protocol: TCP
```

Abbildung 39 Erlauben von DNS

Es ist zu empfehlen, zu Beginn eine clusterweite Default-Policy zu erstellen, die alle Pods isoliert. So wird die Gefahr eines ungewollten Exponierens von Pods minimiert. (2 S. https://kubernetes.io/docs/concepts/services-networking/network-policies/)
Zudem ist es hilfreich, seine Network-Policies regelmäßig zu testen. Dies ist besonders wichtig, da Network-Policies durch externe Komponenten umgesetzt werden und es möglich wäre, dass die konkrete Implementierung vom theoretischen Standard abweicht. Dies führt dazu, dass es hilfreich ist, sich mit den Bedeutungen von *{ }*, *[ ]* und keinem Wert (*empty* vs. *null fields*) im konkret verwendeten Plugin vertraut zu machen.
Der größte Vorteil von NetworkPolicies ist die Möglichkeit der Durchsetzung von Regeln auf Schicht drei des OSI-Modells sowie die Erweiterbarkeit durch *Custom Resource Definitions* (CRD). Diese ermöglichen erweiterte Funktionalitäten in NetworkPolicies, wie zum Beispiel eine eigene Auswertungsreihenfolge von Regeln, clusterweite Regeln oder erweiterte Selektoren. Calico bietet so zum Beispiel auch die Möglichkeit, nur einzelne HTTP-Verben zu erlauben. Zur Wahrung der Kompatibilität zu anderen Network-Policy-Plugins sollte man aber so viele Regeln wie möglich durch Kubernetes-Standardregeln abbilden, um später einfach wechseln zu können.
Leider haben Network-Policies auch einige Nachteile, solange die Möglichkeiten der Kubernetes-Spezifikation betrachtet und CRD der Plugins nicht verwendet werden. Der erste Nachteil ergibt sich aus dem Fehlen von Regulierungsmöglichkeiten auf Schicht sieben des OSI-Modells, solange keine CRDs genutzt werden. Dadurch ist es beispielsweise nicht möglich, einzelne HTTP-Verben zu erlauben oder zu sperren. Weiterhin kann das strukturelle Prinzip von Network-Policies zu erhöhtem Aufwand führen, wenn zum Beispiel nur wenige Verbindungen blockiert werden müssen. Aktuell ist es nicht möglich, explizit Verbindungen zu verbieten und implizit alle anderen zuzulassen. In diesem Zuge sei auch das Fehlen von erweiterten Selektoren erwähnt, es ist nicht möglich, „bekannte“ Funktionen wie *in()* oder *contains()* auf *PodSelectors* und *NamespaceSelectors* anzuwenden.
Der größte Nachteil besteht jedoch darin, dass Network-Policies immer nur in genau einem spezifizierten Namespace (mit Ausnahme der CRD der Plugins) gültig sind. Somit ist es zum einen nicht möglich, eine „echte“ clusterweite Network-Policy durchzusetzen, zum anderen kann das Regelwerk schnell unübersichtlich werden, wenn viele allgemeine Regeln in verschiedenen Namespaces vorhanden sind.

Schlussendlich bleibt anzumerken, dass alle Plugins für NetworkPolicies einem ständigen Wandel unterliegen und der Funktionsumfang schwanken kann. Gleichwohl sind NetworkPolicies unerlässlich für die Sicherheit eines Kubernetes-Clusters und es gibt viele Plugins, sodass jeder das geeignete Plugin für seinen spezifischen Use-Case finden kann. Durch das CNI (Container Network Interface) von Kubernetes ist zwar sichergestellt, dass jedes Plugin Networking und NetworkPolicies unterstützt, allerdings nutzen verschiedene Plugins verschiedene Ansätze, um das Networking abzubilden: während Calico wie bereits erwähnt BGP mit IPv4 nutzt, setzen Flannel oder WeaveNet auf ein Overlay-Netzwerk mit der VXLAN-Technik (da VLAN nur 12 Bit für das VLAN-ID-Feld bereitstellt und dadurch nur 4096 VLAN-IDs möglich wären).[5]

3.1.5 Pod Security Policies (PSP)

Pod Security Policies sind clusterweite Ressourcen, die Regeln festlegen, die Pods erfüllen müssen, um auf dem Kubernetes-Cluster laufen zu dürfen. Das Wichtigste ist zu prüfen, ob PSP im Kubernetes-Cluster aktiviert sind, da nicht alle Kubernetes-Distributionen diese standardmäßig aktivieren. Mit dem Befehl *kubectl get psp* lässt sich überprüfen, ob dieses Feature aktiviert ist. Bei einer Ausgabe der Art *No resources found* oder einer Auflistung von PSP sind diese aktiviert, bei einer Ausgabe der Art *the server does not have a resource type „Pod Security Policies“* sind PSP nicht aktiviert. Dann müssen diese aktiviert werden, dieser Vorgang muss der Dokumentation der konkreten Kubernetes-Distribution entnommen werden.

Außerdem sind standardmäßig keine PSP enthalten, sodass diese erst selbst definiert und auf dem Kubernetes-Cluster erstellt werden müssen.

Ein weiterer wichtiger Aspekt ist der Begriff Pod Security Policy selbst: diese beziehen sich keineswegs auf einen spezifischen Pod, sondern sind clusterweite Ressourcen, die das gesamte Kubernetes-Cluster betreffen.

Die exakten Parameter, welche in PSP genutzt werden können sowie deren Funktion und Standardbelegung können der Kubernetes-Dokumentation entnommen werden. (2 S. https://kubernetes.io/docs/concepts/policy/pod-security-policy/)

[5] Eine gute Einführung in verschiedene CNI-Plugins und deren Funktionsweise sowie Performance findet sich hier: https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/

PSP werden, wie alle anderen Ressourcen auch, durch YAML-Dateien erzeugt. Es können viele einzelne Dateien erstellt und angewendet werden, in einem produktiven System ist es allerdings von Vorteil, wenn alle Regeln in einer Datei stehen.
PSP sind mächtige Werkzeuge, um Sicherheitsrichtlinien durchzusetzen. Das erste, was man verhindern sollte, sind privilegierte Container. Privilegierte Container können auf Geräte des Host-Systems zugreifen. Diese Option ist standardmäßig deaktiviert. In einigen Fällen kann es hilfreich sein, Container zu einem Read-only Dateisystem zu zwingen. Solche Container haben dann keinen Schreibzugriff, womit sich eine Art *immutable infrastructure* erzeugen lässt. Um Änderungen vorzunehmen, muss der Container dann explizit ausgetauscht werden. Ein weiterer wichtiger Aspekt, der mit PSP verhindert werden kann, ist *privilege escalation*. Dieses Feature ist in Kubernetes standardmäßig **nicht** aktiviert, sondern muss explizit eingeschaltet werden. Dies begründet sich darin, dass die Funktionsweise dieser Einstellung das Ändern der effektiven *uid* über den Befehl *setuid* verbietet und um bestehende Binaries dadurch nicht handlungsunfähig zu machen, ist diese Option standardmäßig nicht aktiviert. Wenn dies erlaubt wird, sollte man sich reichlich Gedanken über folgenden Sachverhalt machen: Warum benötigt meine Anwendung mehr Rechte, als sie zu ihrem Start zugesprochen bekommt?
In diesem Zuge sollte man sich auch Gedanken darüber machen, ob ein Container als *root* laufen soll beziehungsweise darf. Auch dies lässt sich durch PSP verhindern. Es ist auch möglich, Gruppen und Nutzer einzustellen. Beide Optionen sollten immer zusammen verwendet werden, da sie in unmittelbarem Zusammenhang stehen und nur zusammen ein optimaler Schutz erreicht werden kann: Privilege Escalation unterbinden ist nutzlos, wenn der Container als *root* laufen darf beziehungsweise, wenn der Container nicht als *root* laufen darf, aber Privilege esacalation nicht verboten ist. Um diese wichtigen Parameter einzustellen, ist folgende Policy nötig:

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example-policy/v1beta1
spec:
  privileged: false
  rule: 'MustRunAsNonRoot'
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: #as required
  seLinux:
    rule: 'runAsAny'
  supplementalGroup:
    rule: 'MustRunAs'
    ranges:
     - min: 1
       max: 65535
  fsGroups:
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
```

Abbildung 40 Pod-Security-Policy mit einfachen Beschränkungen

Die Regel aus Abbildung 40 setzt die erläuterten Regeln um und verbietet die Nutzung der Root-Gruppe durch das Setzen der *range*, da Root-Nutzer und -Gruppen immer die *uid* beziehungsweise *gid* 0 haben.

PSP bieten noch feinere Regulierungsmöglichkeiten, hierzu zählen unter anderem die Freigabe von expliziten Laufwerken für Container sowie das Festlegen von Ports, auf welche der Container zugreifen darf. Prinzipiell gilt: sind PSP aktiviert, muss jeder Pod, der erzeugt werden soll, von mindestens einer PSP validiert werden. Stehen mehrere PSP zur Auswahl, wird die alphabetisch erste PSP verwendet. (2 S. https://kubernetes.io/docs/concepts/policy/pod-security-policy/)

Um PSP nun anwenden zu können, müssen diese noch über RBAC verfügbar gemacht werden, denn PSP sind im Kubernetes-Kontext normale API-Ressourcen.

Die folgende *ClusterRole* und *ClusterRoleBinding* machen die PSP für alle authentifizierten Benutzer beziehungsweise ServiceAccounts verfügbar. Dies geschieht über die Zuweisung des PSP-Namens (oder der PSP-Namen) an das Array *resourceNames*:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: name-of-clusterrole
rules:
  apiGroups: ["extensions"]
  resources: ["podSecurityPolicies"]
  resourceNames: ["name_of_psp", "..."]
  verbs: ["use"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: name_of_rolebinding
  namespace: kube-system
subjects:
  - Kind: Group
    name: system:authenticated
roleRef:
  apiGroup:rbac.authorization.k8s.io
  kind: ClusterRole
  name: name_of_clusterrole
```

Abbildung 41 Anwenden einer Pod-Security-Policy

Möglicherweise ist es nötig, die Gruppe *system:nodes* ebenfalls zu berechtigen, da sonst einige Teile der Kubernetes-Basiskomponenten nicht funktionieren beziehungsweise berechtigt werden.[6]

Je nach verwendeten Netzwerkkomponenten in Kubernetes kann es außerdem möglich sein, dass diese folgende Berechtigungen benötigen: *privileged: yes*, *allowPrivilegeEscalation: yes* sowie *hostNetwork: true*. Es empfiehlt sich, eine zweite Policy zu erstellen und diese über spezifische RBAC an die konkreten Serviceaccounts zu binden. So entsteht eine Art „Ausnahmeregelung“ für diese Serviceaccounts:

[6] https://github.com/kubernetes/kubernetes/issues/70952, abgerufen am 25.06.2019

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: name-of-clusterrole
rules:
  apiGroups: ["extensions"]
  resources: ["podSecurityPolicies"]
  resourceNames: ["name_of_psp", "..."]
  verbs: ["use"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: name_of_rolebinding
  namespace: kube-system
subjects:
  - Kind: ServiceAccount
    name: calico-node
  - Kind: ServiceAccount
    name: kube-proxy
roleRef:
  apiGroup:rbac.authorization.k8s.io
  kind: ClusterRole
  name: name_of_clusterrole
```

Abbildung 42 Anwenden einer Pod-Security-Policy auf explizite Service-Accounts

Man sollte bei RBAC für PSP die Unterschiede zwischen RoleBinding und Role sowie ClusterRole und ClusterRoleBinding beachten, um auszuschließen, dass Pods in anderen Namespaces nicht erzeugt werden, weil deren Service-Accounts nicht auf die entsprechende PSP zugreifen dürfen. Aus diesem Grund empfiehlt es sich, eine PSP zu erstellen, die immer und von allen Ressourcen referenziert werden kann:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: default-psp
rules:
  apiGroups: ["extensions"]
  resources: ["podSecurityPolicies"]
  resourceNames: ["default-psp"]
  verbs: ["use"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: default-psp
  namespace: kube-system
subjects:
  - Kind: Group
    name: system:authenticated
roleRef:
  apiGroup:rbac.authorization.k8s.io
  kind: ClusterRole
  name: default-psp
```

Abbildung 43 Anwenden der default-PSP auf alle Nutzer

Dies lässt sich über eine ClusterRoleBinding zusammen mit der Gruppe *system: authenticated* realisieren.

Die Notwendigkeit von PSP in produktiven Systemen lässt sich an einem simplen Beispiel zeigen: Ein *nginx*-Deployment wird erzeugt und in dieses Deployment ein Dateipfad gemountet:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: team-b
  labels:
    run: nginx
    app: team-b
spec:
 containers:
   - image: joshrosso/nginx-curl:v2
     imagePullPolicy: IfNotPresent
     name: nginx
     volumeMounts:
       - mountPath: /bad
         name: bad
 volumes:
  - name: bad
    hostPath:
      path: /etc/kubernetes/manifests
      type: Directory
```

Abbildung 44 Einfaches Deployment, dass einen simplen Webserver bereitstellt

Dieses mountet ein Verzeichnis in den Container, in welchem manche Kubernetes-Distributionen statische Pods lagern. Diese werden direkt vom *kubelet* erzeugt. Gelingt es also, dort Deployments zu erzeugen, werden diese instanziiert, ohne den API-Server zu passieren. Zunächst wird das Deployment erstellt und geprüft, ob der Container erzeugt wird:

```
Every 2.0s: kubectl get pods

NAME     READY   STATUS    RESTARTS   AGE
team-b   1/1     Running   0          101s
```

Abbildung 45 Verifizieren, dass der Pod erzeugt wurde

Da noch keine PSP aktiv sind, geschieht dies erwartungsgemäß. Nun wechselt man in eine Shell des Pods *team-b* und wechselt dort in das Verzeichnis /bad, welches wie gewünscht gemountet wurde. Anschließend wird *wget* installiert, um YAML-Dateien direkt herunterladen zu können:

```
root@team-b:/# apt install wget
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  wget
0 upgraded, 1 newly installed, 0 to remove and 3 not upgraded.
Need to get 800 kB of archives.
After this operation, 2813 kB of additional disk space will be used.
Get:1 http://security-cdn.debian.org/debian-security stretch/updates/main amd64 wget amd64 1.18-5+deb9u3 [800 kB]
Fetched 800 kB in 0s (3335 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package wget.
(Reading database ... 7729 files and directories currently installed.)
Preparing to unpack .../wget_1.18-5+deb9u3_amd64.deb ...
Unpacking wget (1.18-5+deb9u3) ...
Setting up wget (1.18-5+deb9u3) ...
root@team-b:/#
```

Abbildung 46 Installieren von Software im Pod

Nun kann eine beliebige YAML-Datei heruntergeladen werden, die einen Pod enthält:

```
root@team-b:/bad# wget https://raw.githubusercontent.com/lachie83/k8s-manifest/master/pod-example.yaml
--2019-06-24 11:34:44--  https://raw.githubusercontent.com/lachie83/k8s-manifest/master/pod-example.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.36.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.36.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 214 [text/plain]
Saving to: 'pod-example.yaml'

pod-example.yaml                    100%[================================================================================>]     214  --.-KB/s    in 0s

2019-06-24 11:34:44 (3.91 MB/s) - 'pod-example.yaml' saved [214/214]

root@team-b:/bad# ls -la
total 12
drwxr-xr-x 2 root root 4096 Jun 24 11:34 .
drwxr-xr-x 1 root root 4096 Jun 24 11:34 ..
-rw-r--r-- 1 root root  214 Jun 24 11:34 pod-example.yaml
```

Abbildung 47 Download externer Dateien

Da diese YAML-Datei vom *kubelet* direkt instanziiert wird, passiert die Anfrage nicht den API-Server und der „kompromittierte“ Pod wird als statischer Pod gestartet:

```
dev@ubuntu:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
croc-hunter-aks-rancher-11052310-0  1/1     Running   0          5m51s
team-b                              1/1     Running   0          30m
```

Abbildung 48 Aus heruntergeladener YAML-Datei erzeugter Pod

Es ist also möglich, Container in das System einzubringen, ohne dass hierfür eine Anfrage den API-Server passiert hat.

Dieses Problem würde sich in dieser Form vermeintlich mit einer Network-Policy eingrenzen lassen, allerdings benötigt es nur eine erlaubende Regel für den ein- beziehungsweise ausgehenden Datenverkehr, um diesen Angriff auch in vermeintlich gesicherten Clustern durchzuführen. Der Angriffsvektor wäre dann keine reine technische Schwachstelle mehr, sondern eine Kombination aus mehreren Network-Policies und menschlichem Versagen.

Aber auch die Sicherheitslösung Network-Policy lässt sich aushebeln. Wird in einem Pod oder Deployment der Parameter *hostNetwork: true* gesetzt, hat die NetworkPolicy keine Wirkung, da der Pod dann nicht mehr sein eigenes Netzwerk benutzt, sondern das Host-Netzwerk, was dem Netzwerk des Nodes entspricht.[7] Dadurch wäre es dann möglich, auch auf IP-Adressen innerhalb des gesamten Clusters, wie zum Beispiel Services, zuzugreifen. Durch eine PSP lässt sich durchsetzen, dass Container diesen Parameter nicht setzen dürfen.

3.1.6 TLS everywhere

Transport Layer Security, häufig auch unter der Bezeichnung des Vorgängerprotokolls SSL beziehungsweise Secure Socket Layer bekannt, ist ein hybrides Protokoll zur sicheren Datenübertragung im Internet. Gegenwärtig wird TLS hauptsächlich in Verbindung mit HTTP (dann als HTTPS) eingesetzt. Auch in Kubernetes sollte diese Funktionalität umgesetzt werden, um die Sicherheit der Daten während des Transports zu gewährleisten. In Zeiten, in denen Server virtualisiert werden und in denen keine physische Kontrolle über das Netz besteht, kommt dem Punkt der Verschlüsselung eine immer größere Bedeutung zu. Leider unterstützt Kubernetes eine TLS-Kommunikation nur zwischen den Komponenten der Control Plane. Um TLS für ausgehende Services oder zwischen Pods nutzen zu können, müssen eigene Maßnahmen umgesetzt werden, welche in diesem Kapitel erläutert werden.

[7] “NetworkPolicy does not apply to HostPort: true”: https://github.com/projectcalico/felix/issues/1361

Zunächst gilt es, die Verbindung zwischen Client und dem Kubernetes Loadbalancer beziehungsweise Service abzusichern. Hierfür spezifiziert das Ingress-Objekt die Möglichkeit, TLS-Zertifikate zu hinterlegen. Auch hier gilt, dass verschiedene Ingress-Plugins durch CRD möglicherweise unterschiedliche Funktionalitäten bieten. (2 S. https://kubernetes.io/docs/concepts/services-networking/ingress/)

Achtung: Seit Mitte Juni 2019 gehören Ingress-Objekte der Gruppe ***networking.k8s.io/v1beta1*** an und sind nicht mehr Teil der Gruppe *extensions/v1beta1*.

(2 S. https://kubernetes.io/docs/concepts/services-networking/ingress/)

Bei der Verwendung von TLS ist anzumerken, dass in Rancher-Clustern (automatisiert erzeugte Azure-Cluster, die über *Rancher* verwaltet werden können) innerhalb von FIRMA die Strecke zwischen Client und *Edge* mit einem Wildcard-Zertifikat (von der deutschen Telekom für **.Firma.services* ausgestellt) verschlüsselt ist. Zusätzlich ist HSTS aktiviert, sodass HTTP-Anfragen automatisch auf HTTPS umgeleitet werden. Der genaue Ablauf ist in Abbildung 49 zu sehen:

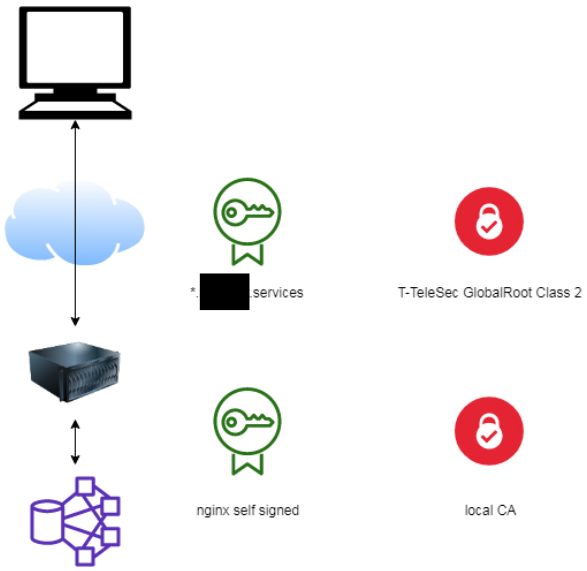


Abbildung 49 TLS-Zertifikate und CAs entlang der Strecke zwischen Ingress und Client

Der Benutzer kommuniziert über das Internet mit dem *Edge*, welcher das bereits erwähnte Wildcard-Zertifikat nutzt. Der *nginx-ingress* nutzt für die Kommunikation mit dem *Edge* ein selbst erstelltes Zertifikat, diesem vertraut der *Edge*. Hier bestehen keine sicherheitstechnischen Bedenken, Verwaltungsaufwand entsteht im Rancher-Umfeld ebenfalls nicht, da dieser Prozess dort automatisiert ist. Somit ist die gesamte Strecke zwischen Ingress-Controller und Client in FIRMA Rancher-Clustern standardmäßig mit TLS verschlüsselt. Diese *TLS Termination (häufiger auch SSL*

Termination) wird benötigt, da zusätzlich noch eine Web Application Firewall zwischengeschaltet ist. Da diese auf Layer sieben des OSI-Modells arbeitet, muss sie den Inhalt der Applikationsschicht sehen können, um zum Beispiel SQL-Injection oder Cross-Site-Scripting zu verhindern.

Im Folgenden soll nun gezeigt werden, wie für ein beliebiges Kubernetes-Cluster eigene Zertifikate genutzt werden können. Hier stehen sowohl eine selbst erzeugte Certificate Authority, als auch LetsEncrypt oder jede andere fremde Certificate Authority (zum Beispiel die FIRMA-eigene) zur Auswahl. Dieser Prozess lässt sich mit *cert-manager* vollständig automatisieren. Um dem Problem der zwischengeschalteten Komponenten in FIRMA Rancher-Clustern zu entgehen, wird der Prozess auf einem lokalen *minikube*-Cluster durchgeführt. Als erstes wird der Prozess mit einer lokalen CA automatisiert. Hierfür wird ein Key und Zertifikat erzeugt und diese mittels *kubectl create secret tls ca-key-pair --key=ca.key --cert=ca.crt* als Secret in das Kubernetes-Cluster eingebracht. Dieses Secret kann nun in einer CRD des *cert-mananager* als Issuer verwendet werden. Es wird ein Issuer für den default-Namespace erzeugt:

```yaml
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: ca-issuer
  namespace: default
spec:
  ca:
    secretName: ca-key-pair
```

Abbildung 50 Issuer des cert-manager

Falls man diese Funktion mit LetsEncrypt nutzen will, muss der Issuer wie folgt aussehen:

```yaml
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
  namespace: ingress-basic
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: x@y.de
    privateKeySecretRef:
      name: letsencrypt-staging
    http01: {}
```

Abbildung 51 ClusterIssuer mit LetsEncyrpt-Konfiguration

Die Emailadresse sollte in der Praxis durch eine echte ersetzt werden. Weiterhin wurde hier zu Testzwecken die Staging-Umgebung von LetsEncrypt genutzt. In der Praxis muss *https://acme-v02.api.letsencrypt.org/directory* verwendet werden, um gültige Zertifikate zu erhalten. Nun kann das Zertifikat für den Service erstellt werden.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: tls-secret
  namespace: ingress-nginx
spec:
  secretName: tls-secret-staging
  dnsNames:
  - test-exam.[redacted].services
  acme:
    config:
    - http01:
        ingressClass: nginx
      domains:
      - demo-aks-ingress.eastus.cloudapp.azure.com
  issuerRef:
    name: letsencrypt-staging
    kind: ClusterIssuer
```

Abbildung 52 Zertifikat, welches mit LetsEncrypt signiert werden soll

In diesem Zertifikat wird nun auf den *Issuer* verwiesen (bei Verwendung einer eigenen CA müssen hier entsprechend die Attribute *name* sowie *kind* angepasst werden). Nun kann in der Ingress-Definition des Services das Secret angegeben werden:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name:ingress-tls-example
  namespace: ingress-nginx
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer: letsencrypt-staging
spec:
  tls:
  - hosts:
    - test-example.[redacted].services
    secretName: tls-secret
  rules:
   [...]
```

Abbildung 53 Ingress-Definition mit Angabe des Issuers, welcher verwendet werden soll

Die Annotation teilt dem *cert-manager* mit, welcher *Issuer* verwendet werden soll, um das Zertifikat für diesen Service zu erzeugen. Damit ist die Einrichtung abgeschlossen. *Cert-manager* wird sich jetzt um die Aktualisierung der Zertifikate kümmern, wenn diese -nach 90 Tagen bei der Verwendung von LetsEncrypt oder der angegeben

Gültigkeit bei selbst erstellten Zertifikaten- abgelaufen sind. In neueren Versionen erstellt *cert-manager* das Zertifikat automatisch, das manuelle Erstellen des Zertifikats fällt dann weg, es sei der Vollständigkeit wegen aber trotzdem gezeigt. Anschließend sollte das Zertifikat geprüft werden:

```
dev@ubuntu:~$ kubectl describe certificate tls-secret
Name:         tls-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  certmanager.k8s.io/v1alpha1
Kind:         Certificate
Metadata:
  Creation Timestamp:  2019-06-26T11:13:53Z
  Generation:          4
  Owner References:
    API Version:           extensions/v1beta1
    Block Owner Deletion:  true
    Controller:            true
    Kind:                  Ingress
    Name:                  test-ingress
    UID:                   2212b575-97e3-11e9-b7c5-2eab01768b9f
  Resource Version:        878664
  Self Link:               /apis/certmanager.k8s.io/v1alpha1/namespaces/default/certificates/tls-secret
  UID:                     7b5fb975-9803-11e9-b7c5-2eab01768b9f
Spec:
  Acme:
    Config:
      Domains:
        test.exam.[redacted].services
      Http 01:
  Dns Names:
    test.exam.markant.services
  Issuer Ref:
    Kind:       ClusterIssuer
    Name:       letsencrypt-staging
  Secret Name:  tls-secret
Status:
  Conditions:
    Last Transition Time:  2019-06-26T11:27:04Z
    Message:               Certificate is up to date and has not expired
    Reason:                Ready
    Status:                True
    Type:                  Ready
  Not After:               2019-09-24T10:27:03Z
Events:
  Type    Reason              Age   From          Message
  ----    ------              ----  ----          -------
  Normal  Generated           135m  cert-manager  Generated new private key
  Normal  GenerateSelfSigned  135m  cert-manager  Generated temporary self signed certificate
  Normal  OrderCreated        135m  cert-manager  Created Order resource "tls-secret-1252087812"
  Normal  OrderComplete       122m  cert-manager  Order "tls-secret-1252087812" completed successfully
  Normal  CertIssued          122m  cert-manager  Certificate issued successfully
```

Abbildung 54 Beschreibung des ausgestellten Zertifikats

In der Praxis dauert die Verifizierung der Anfrage circa 15 Minuten, sodass das Zertifikat anfangs nicht als *ready* markiert wird. Dies geschieht, sobald das Zertifikat erfolgreich ausgestellt wurde.

Die Verwendung von *cert-manager* ist einfach und intuitiv. Zusätzlich wird das Sicherheitsniveau deutlich erhöht, da so eine Verschlüsselung der Daten möglich ist.

An dieser Stelle sei darauf hingewiesen, dass *cert-manager* der offizielle Nachfolger von *kube-lego* ist. Letzteres wird nur noch gewartet, neue Features werden nicht mehr implementiert. Die Tools stehen also nicht in Konkurrenz zueinander. (35)

Nun besteht immer noch das Problem, dass die zu Grunde liegende Netzinfrastruktur als kompromittiert betrachtet werden muss und somit eine Verschlüsselung auf der Anwendungsschicht bei der Kommunikation von Pods, Ingress und Services notwendig ist. Auch dieses Feature muss über Plugins realisiert werden, das bekannteste ist *Istio*. Dieses beinhaltet sehr viele Funktionalitäten, so ist standardmäßig auch ein *Prometheus* und ein *Grafana*-Dashboard enthalten, um das Cluster zu visualisieren.

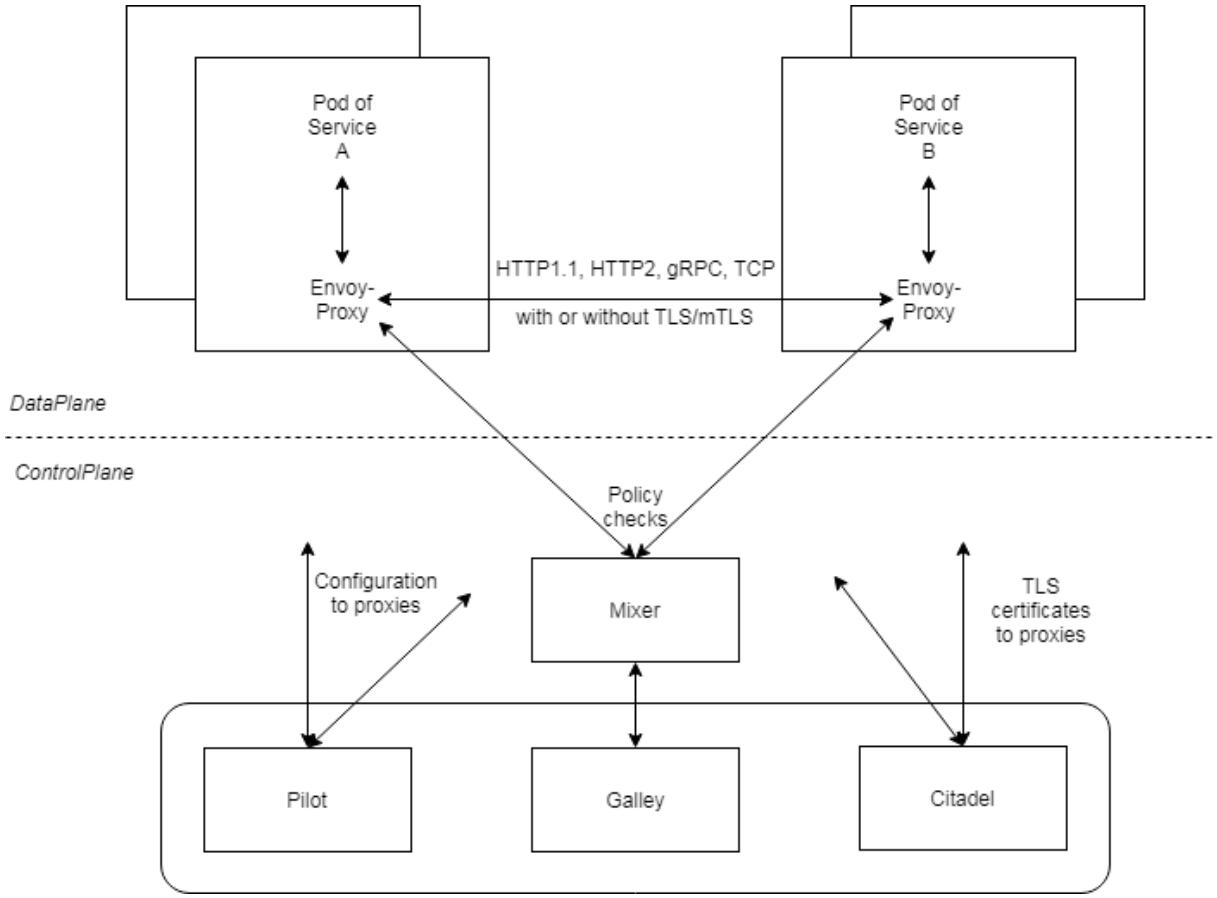


Abbildung 55 Schematische Struktur der Istio-Komponenten

In Abbildung 55 ist die schematische Struktur von Istio dargestellt. Die Funktionalität von Istio wird dadurch gewährleistet, dass in jeden Pod ein Envoy-Proxy injiziert wird. Es ist möglich, Namespaces mit entsprechenden Labels zu versehen, sodass dies automatisiert für jeden erzeugten Pod geschieht. Envoy stellt erweiterte Funktionalität wie TLS Termination, Loadbalancing auf Layer 7 (HTTP), prozentbasierte Rollouts (10% zu Version 1, 90% zu Version 2) sowie hilfreiche Entwicklungs- und Testfunktionen wie Fault Injection und Circuit Breaking bereit. (36 S. https://istio.io/docs/concepts/what-is-istio/)

Die Komponente Mixer fasst Telemetriedaten der Envoy-Proxies zusammen und ermöglicht die Nutzung und Durchsetzung von Beschränkungen bezüglich des Datenverkehrs.
Citadel stellt die Kernkomponente zur Nutzung der Verschlüsselung dar: Citadel automatisiert das Erstellen von Zertifikaten für Pods o.ä. im gesamten Cluster und ermöglicht dadurch eine durchgehende Nutzung von TLS bis zu dem Pod, der die Anfrage bearbeitet (und nicht wie bei Ingress nur bis zum Einstiegspunkt in das Cluster). Die Zertifikate werden durch Citadel als Kubernetes-Secret in die entsprechenden Envoy-Proxies eingebunden
Der Pilot konfiguriert die Envoy-Proxies; um diesen Prozess performant zu gestalten, versucht er, so viele Daten wie möglich im Voraus zu berechnen.

Um Istio zu installieren, müssen seit der Version *1.2.2* (am 24.06.2019 veröffentlicht) mehrere YAML-Dateien installiert werden, die alle benötigten CRDs für Istio enthalten. Vor Version *1.2.2* waren alle CRDs in der Datei *install/kubernetes/helm/istio/templates/crds.yaml* enthalten. Dem ist nun, wie bereits erwähnt, nicht mehr so. Um nicht mehrere Male *kubectl* mit verschiedenen Dateien aufrufen zu müssen, können alle CRDs folgendermaßen installiert werden:
*for file in install/kubernetes/helm/istio-init/files/crd*yaml; do kubectl apply -f $file; done*
Anschließend können die eigentlichen Komponenten von Istio installiert werden:
kubectl apply -f install/kubernetes/istio-demo-auth.yaml
Diese Datei konfiguriert Istio in den sogenannten *strict mutual TLS* Modus, sodass ausschließlich TLS-Datenverkehr zulässig ist. Das bedeutet aber nicht, dass damit der Datenverkehr automatisiert verschlüsselt wird! Istio wird in diesem Modus lediglich alle Datenpakete verwerfen, die nicht verschlüsselt sind. Daneben gibt es auch noch einen *permissive TLS* Modus, der sowohl verschlüsselten als auch unverschlüsselten Datenverkehr erlaubt. Dieser ist zu empfehlen, wenn Istio in eine bestehende Umgebung eingefügt werden soll, um Fehler zu vermeiden. In solch einem Falle sollte aber zügig auf die alleinige Vewendung des *strict mutual TLS* Modus umgestellt werden.
Um die Anwendung von mTLS zwischen Services zu zeigen, kommt die Anwendung *Hipstershop* zum Einsatz. Diese besteht aus vielen Microservices:

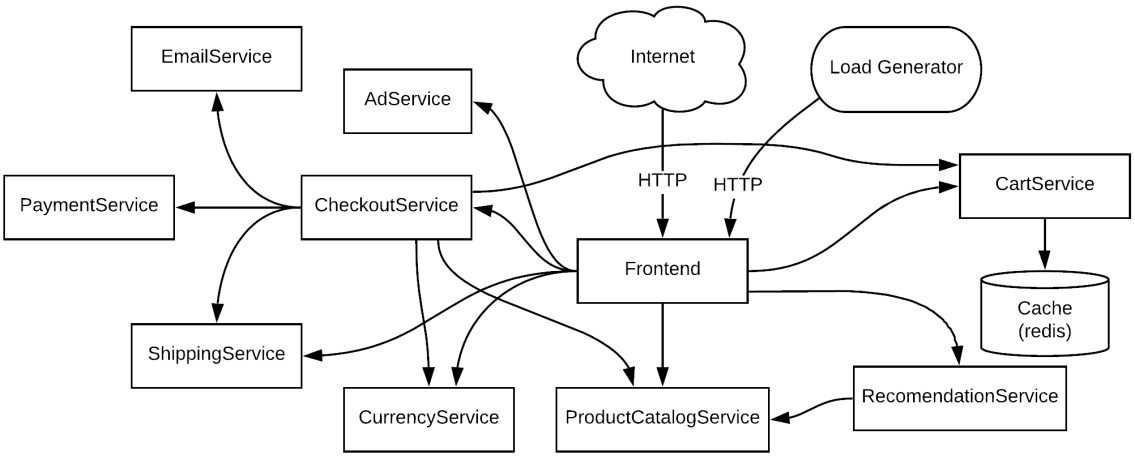


Abbildung 56 Aufbau der Anwendung HipsterShop

Diese Anwendung soll nun mit mTLS gesichert werden, sodass kein unverschlüsselter Datenverkehr möglich ist. Um dies zu erreichen, werden zwei CRD von Istio benötigt: Eine Policy, um den eingehenden Datenverkehr zu validieren sowie eine *DestinationRule*, um ausgehenden Datenverkehr zu reglementieren. Es gibt auch hier (in Anlehnung an andere Policies) Namespace-gebundene (normale Policy) und Mesh-weit gültige Policies (MeshPolicy). (36) Es ist möglich, sowohl einzelne Services als auch mehrere Services abzusichern. Im ersten Schritt soll das Frontend mit TLS abgesichert werden. Hierfür sind, wie bereits erwähnt, eine *DestinationRule* und eine Policy notwendig. Zur Vereinfachung werden diese in dieselbe Datei geschrieben:

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "frontend-authn"
spec:
  targets:
  - name: frontend
  peers:
  - mtls: {}
---
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "frontend-dr"
  namespace: "default"
spec:
  host: "frontend.default.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

Abbildung 57 Absichern mit mTLS

Diese Datei definiert zunächst die eingehenden Verbindungen. Dies geschieht mit dem Typ *Policy*. Es gibt auch den Typ *MeshPolicy*, diese Policy gilt dann im gesamten Cluster. Hierfür wird über das *targets*-Attribut mit einem Label-Selektor angegeben, auf welche Pods die Regel zutreffen soll. Diese können auch durch die Angabe von Ports ergänzt werden. Anschließend wird eine Liste an *peers* angegeben, die aktuell noch einen leeren Eintrag *mtls: { }* enthält. Dies kann sich in Zukunft möglicherweise ändern. Hier ist es auch möglich, nur einzelne Verbindungen zuzulassen beziehungsweise einzelne Verbindungen auszuschließen. (36 S. https://istio.io/docs/reference/config/istio.authentication.v1alpha1/)
Anschließend wird die DestinationRule für ausgehenden Datenverkehr erstellt. Diese benötigt den exakten Host, dessen ausgehende Verbindungen „reglementiert“ werden sollen. Anschließend wird die TrafficPolicy definiert. Hier ist es ebenfalls möglich, SubjectAlternativeNames zu deklarieren. Folgende TLS-Modi sind möglich:

- DISABLE: nicht empfohlen, kein TLS nutzen.
- SIMPLE: „normales TLS“ zwischen Clients, Zertifikat des aufgerufenen wird verifiziert.
- MUTUAL: mTLS, erfordert manuelle Angabe der Zertifikate, des privaten Schlüssels sowie der CA.
- ISTIO_MUTUAL: mTLS mit Istio-automatisierten Zertifikaten. Dies ist der zu empfehlende Modus für produktive Umgebungen.

Versucht man nun, per HTTP aus einem anderen Pod auf das Frontend zuzugreifen:
kubectl exec $(kubectl get pod -l app=paymentservice -o jsonpath={.items..metadata.name}) -c istio-proxy -- curl http://frontend:80/ -o /dev/null -s -w '%{http_code}\n'

erhält man eine Fehlermeldung:
command terminated with exit code 56
Wird der Curl-Befehl mit HTTPS aufgerufen, resultiert dies in einer korrekten HTTP:200-Antwort.

Falls manuelle Zertifikate genutzt werden sollen, kann MUTUAL verwendet werden:

```yaml
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "frontend-authn"
spec:
  targets:
  - name: frontend
  peers:
  - mtls: {}
---
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "frontend-dr"
  namespace: "default"
spec:
  host: "frontend.default.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: MUTUAL
      clientCertificate: /etc/certs/frontend_certificate.pem
      privateKey: /etc/certs/frontend_private_key.pem
      caCertificates: /etc/certs/customCA.pem
```

Abbildung 58 mTLS mit angegebenen, eigenen Zertifikaten

Ein relevanter Vorteil in produktiven Systemen ergibt sich hierdurch in der Regel nicht. Wie bereits erwähnt, ist es auch möglich, ganze Namespaces mit mTLS abzusichern. Da die Anwendung im default-Namespace bereitgestellt wird, soll dieser nun abgesichert werden.

```yaml
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "default-tls-policy"
  namespace: "default"
spec:
  peers:
  - mtls: {}
---
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "default-tls-policy"
  namespace: "default"
spec:
  host: "*.default.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

Abbildung 59 Absichern eines ganzen Namespaces durch Auswählen von { }

Die Regeln folgen der bereits gezeigten, es wird lediglich kein Pod mehr selektiert. Für die DestinationRule wird der gesamte Namespace (durch den Wildcard-Selektor) ausgewählt.

Im FIRMA-Umfeld ergibt sich das Problem, dass in Azure nur automatisierte DNS-Einträge erstellt werden können, wenn eine Ingress-Ressource definiert wird. Da Istio dies nicht tut, sondern eine Gateway-Ressource verwendet, sind Istio-Services nur über die IP-Adresse des Loadbalancers verfügbar. Ebenfalls werden diese Services nicht über TLS mit dem FIRMA-Zertifikat ausgeliefert. Daher sollen hier drei Möglichkeiten aufgezeigt werden:

1. Alleinige Nutzung von nginx-Ingress (nicht empfohlen)
2. Nutzung von Istio-Ingress
3. Anlegen eines DNS-Eintrags und Nutzung von Istio-Gateway

Die alleinige Nutzung von nginx-Ingress ist nicht empfohlen, da so keine gesicherte Kommunikation zwischen Pods möglich ist. Der Vollständigkeit wegen sei es trotzdem gezeigt:
Es wird zunächst ein Deployment und der dazugehörige Service erstellt:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-kubernetes
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: hello-kubernetes
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-kubernetes
  template:
    metadata:
      labels:
        app: hello-kubernetes
    spec:
      containers:
      - name: hello-kubernetes
        image: paulbouwer/hello-kubernetes:1.5
        ports:
        - containerPort: 8080
```

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  namespace: default
spec:
  rules:
  - host: helloworld.[redacted].services
    http:
      paths:
      - backend:
          serviceName: hello-kubernetes
          servicePort: 80
```

Abbildung 60 Deployment und dazugehöriger Service

Mit der alleinigen Nutzung von nginx-Ingress ergibt sich folgendes Schaubild, welches die Verschlüsselung in diesem Szenario darstellt:

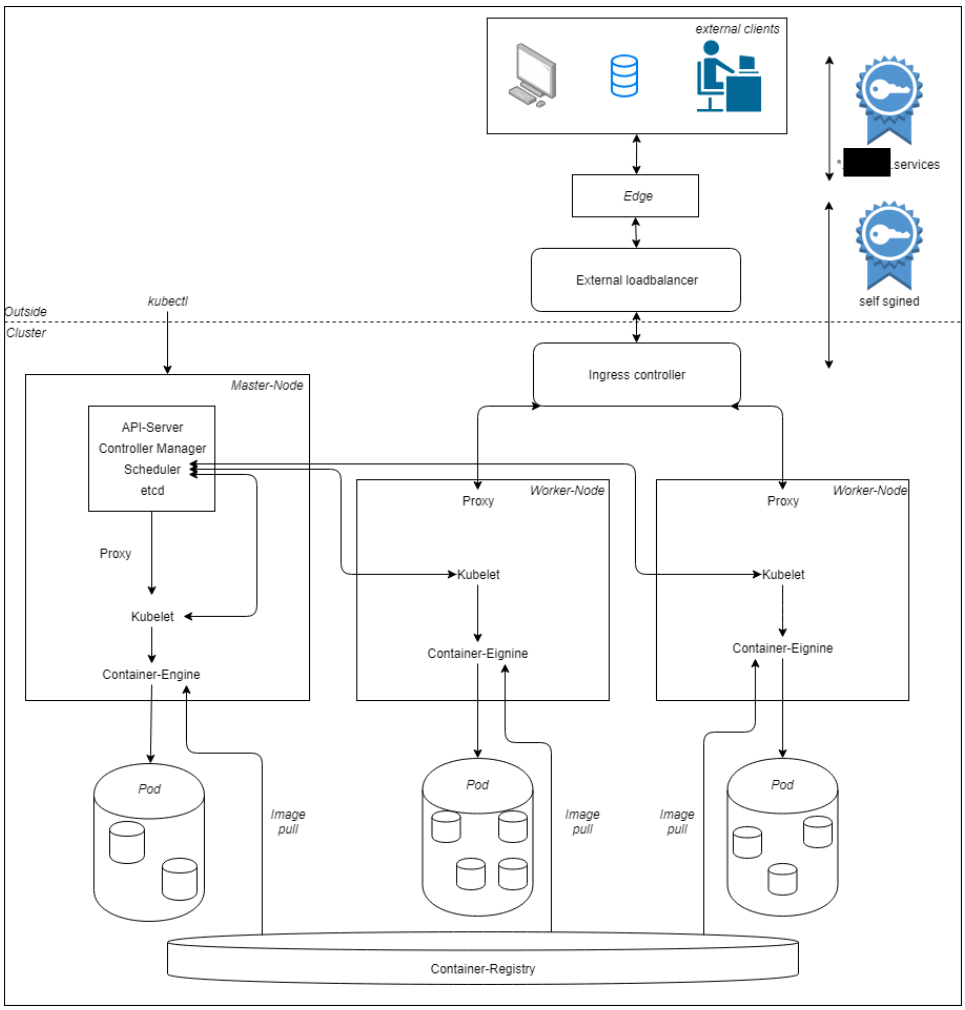


Abbildung 61 Verschlüsselung in Azure bei Nutzung von nginx-Ingress

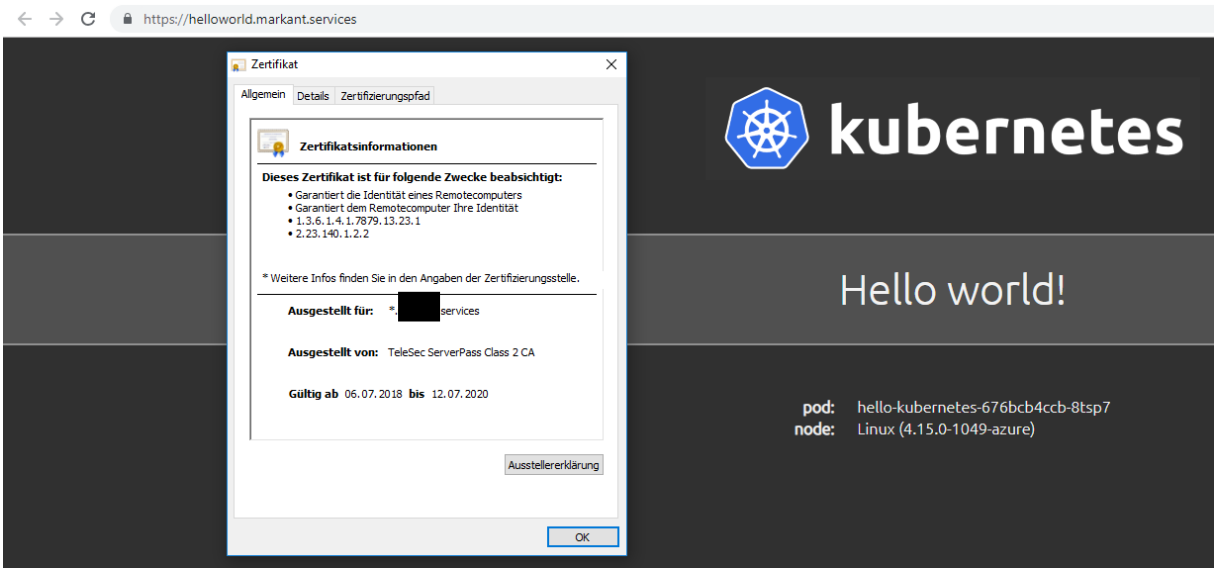


Abbildung 62 Bereitgestellte Seite mit FIRMA-Zertifikat

Hieraus resultiert, dass der Service über die angegebene Domain erreichbar ist und das FIRMA-Zertifikat verwendet wird. Allerdings ist die Kommunikation innerhalb des Clusters nun nicht geschützt, wie aus Abbildung 62 hervorgeht: die Strecke zwischen Client und *Edge* ist mit einem von der deutschen Telekom signierten Zertifikat verschlüsselt, die Strecke zwischen Edge und Ingress-Controller ist mit einem selbst signierten Zertifikat verschlüsselt. Innerhalb des Clusters findet dann keine Verschlüsselung mehr statt.
Istio stellt auch eine eigene Ingress-Ressource bereit, allerdings funktioniert die automatische Bereitstellung eines DNS-Eintrags auch mit dieser nicht.

Möchte man seinen Service über eine Domain verfügbar machen, besteht die Möglichkeit, einen manuellen DNS-Eintrag vornehmen zu lassen und wie gewohnt das Istio-Gateway zu nutzen. Der DNS-Eintrag verweist dann auf die (in der Regel, bei FIRMA immer, statische) IP-Adresse des Istio-Gateways. Dies hat zur Folge, dass die Verbindung nicht mehr über den Edge geführt wird. Die Nutzung von TLS muss dann über Istio abgebildet werden. Somit ist Istio über die ganze Datenverbindung für die Nutzung von TLS zuständig. Hier besteht die Möglichkeit, ein festes Zertifikat per Secret einzubinden, oder den *cert-manager* zu nutzen. Letzteres ist in der Regel zu bevorzugen. Es soll auch möglich sein, nginx-Ingress in Verbindung mit den mächtigen VirtualService von Istio zu nutzen[8], es war aber nicht möglich, diese Konstellation im

[8] https://github.com/istio/istio/issues/7776

FIRMA-Cluster nachzustellen. Das Vorhaben scheitert am Vorhaben, den nginx-Ingress-Controller mit dem VirtualService von Istio zu verbinden. Diese Option würde zwar zwei Plugins kombinieren, wäre aber aus sicherheitstechnischer Sicht optimal: Intern wird mit Istio mTLS angewendet und nach außen integriert sich nginx-Ingress optimal in die Automatisierung von Azure.

Somit bleibt bei der Nutzung von Istio im Azure-Umfeld nur die Möglichkeit, eigene Zertifikate zu nutzen. Da die Loadbalancer-IP statisch ist, können hier FIRMA-eigene Zertifikate verwendet werden. Für interne Services empfiehlt es sich -zur Reduktion von Kosten- ein selbst signiertes Zertifikat zu nutzen, für externe Anwendungen sollte man ein Zertifikat der Telekom-CA verwenden und dann in ein Istio-Gateway einbinden:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: example-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      serverCertificate: /etc/istio/ingressgateway-certs/[redacted]_example_tls.crt
      privateKey: /etc/istio/ingressgateway-certs/[redacted]_example_tls.key
    hosts:
    - "example.[redacted].services"
```

Abbildung 63 Istio-Gateway mit eigenen Zertifikaten

Es muss der Speicherort der Zertifikate beachtet werden, Istio erkennt nur Zertifikate, welche sich im gezeigten Verzeichnis befinden. Auch Unterordner in */ingressgateway-certs* sind nicht zulässig.
(33 S. https://istio.io/docs/tasks/traffic-management/ingress/secure-ingress-mount/)
Somit ergibt sich bezüglich der Verschlüsselung folgendes Schaubild:

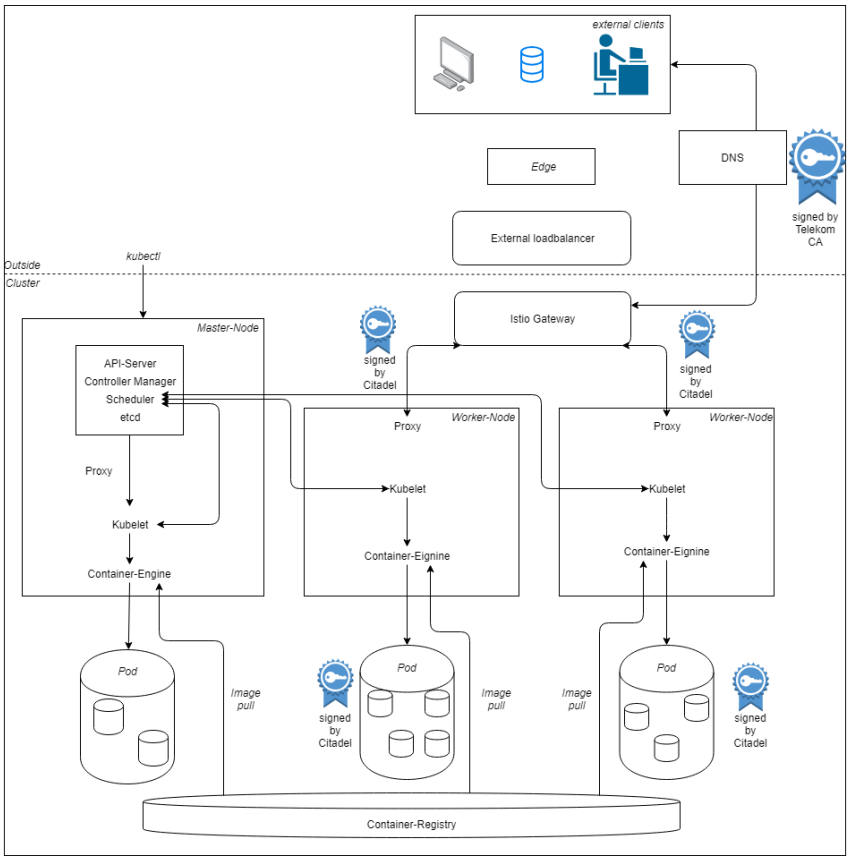


Abbildung 64 Verschlüsselung bei Nutzung von Istio-Gateway

Der Verkehr zwischen Client und Istio ist, wie bereits erwähnt, durch ein Zertifikat der Telekom-CA abgesichert. Im Cluster wird der Datenverkehr dann über mTLS durch Istio mit von Citadel ausgestellten Zertifikaten verschlüsselt.

3.2 Docker

Dieses Kapitel beschäftigt sich mit der Absicherung von Docker-Containern. Da diese immer populärer werden, soll hier auch ein Augenmerk auf das Absichern des gesamten Prozesses des Erstellens von Docker-Images betrachtet werden: Vom Bau der Container-Images bis zum laufenden Container. Dies ist auch ein zentrales Problem bei der Sicherheit von Docker-Containern, nämlich dass nicht ein einzelnes Artefakt, sondern ein ganzer Prozess (Abbildung 65 zeigt einen generellen Prozess der Erstellung von Docker-Images) über verschiedene Personen und technische Einrichtungen hinweg abgesichert werden muss. Hier kommt erschwerend hinzu, dass zwischen den einzelnen Stufen immer häufiger Daten über Netze transportiert werden,

die nicht als sicher eingestuft werden können.

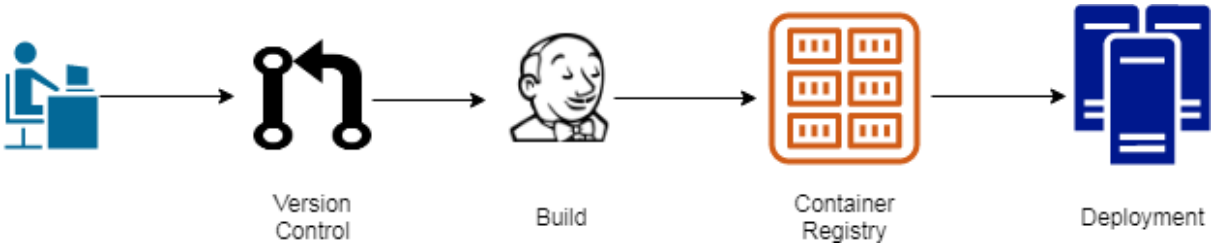


Abbildung 65 Prozesskette eines Docker-Images

Zunächst muss aber der Unterschied zwischen Docker und Kubernetes klargestellt werden: während Docker nur eine Virtualisierungstechnologie darstellt, ist Kubernetes ein Werkzeug, um Container zu verwalten. Docker besitzt ebenfalls ein solches Managementtool, dieses nennt sich Docker Swarm.
(10 S. https://docs.docker.com/engine/swarm/)
Docker lässt sich auch ohne Docker Swarm oder Kubernetes (beziehungsweise allgemein ohne übergeordnetes Managementtool) verwenden, da Docker, wie bereits erwähnt, alleinig die Virtualisierungstechnologien des Linux-Kernel nutzt, allerdings muss dann die Verwaltung manuell vorgenommen werden. Docker lässt sich neuerdings auch unter Windows nutzen, dies war lange Zeit aufgrund der fehlenden Kernel-Features nur über Umwege möglich. (37)

Dieses Kapitel wird sich mit den Bereichen *Build* und *Depoyment* beschäftigen. Die Notwendigkeit von Sicherheitsmaßnahmen lässt sich an einem Vorfall in der öffentlichen Docker-Registry zeigen: Dort waren zwischen 2017 und 2018 mehrere Docker-Images bereitgestellt worden, die bösartige Programme zum „Kryptomining“ nachgeladen haben. (38) Diese Images wurden circa fünf Millionen Mal heruntergeladen. Es ist davon auszugehen, dass diese schädlichen Images auch als Base-Images verwendet wurden, sodass die schädlichen Images auch in zahlreichen privaten Registries vorhanden sein werden. (3 S. 165) Die Dunkelziffer der Downloads wird somit deutlich größer als fünf Millionen sein .

3.2.1 Dockerfiles

Jedes Docker-Image wird über ein sogenanntes Dockerfile erstellt. Ein Dockerfile besteht aus mehreren Zeilen der Form *BEFEHL Parameter*, zu beachten gilt es lediglich, dass der Befehl immer großgeschrieben wird. Kommentare sind mit einer

Raute zu Beginn einer Zeile möglich. Abbildung 66 zeigt ein beispielhaftes Dockerfile, welches die Datenbank *mongoDB* in einem Ubuntu-Linux startet.

```
FROM nexus.[redacted].de/ubuntu:16.04
RUN apt update
RUN apt upgrade
RUN apt install mongodb-10gen
RUN mkdir -p /data/mongodb
EXPOSE 1580
CMD ["/usr/bin/mongod", "--port 1580"]
```

Abbildung 66 Beispielhaftes Dockerfile zum Starten einer MongoDB

Der erste Befehl lautet FROM und stellt immer die erste Anweisung dar. FROM beschreibt, welches Image als Grundlage für das zu erzeugende Image dienen soll. Meistens wird dieses als Base-Image bezeichnet. Hier sind alle Images denkbar, es ist aber ratsam, das Base-Image so klein wie möglich zu halten. Dies begründet sich vor allem darin, dass ein Container keine vollwertige virtuelle Maschine ist, sondern eine spezifische Aufgabe so effizient wie möglich erfüllen soll.

Bei der Verwendung von FROM sollte **immer** die Registry mit angegeben werden, aus der das Image heruntergeladen werden soll. Hierfür kann einfach die URL der Registry vor den Image-Tag geschrieben werden.

Ein weiterer Befehl in einem Dockerfile ist RUN. Dieser ermöglicht das Ausführen von Shell-Befehlen. Dadurch können innerhalb des Containers mehr oder weniger alle Operationen durchgeführt werden. Weiterhin ist es möglich, Ports nach außen freizugeben sowie Verzeichnisse in den Container einzubinden.

Sehr wichtig ist es, in Dockerfiles immer einen USER anzugeben. Wird dies nicht getan, läuft der Container automatisch als Root-Benutzer. Hiervon ist dringend abzuraten und es gibt auch nur wenige Fälle, in denen dieses Verhalten wirklich erforderlich ist. Es sollte vorab ein passender User erzeugt werden, der nur die unbedingt notwendigen Rechte zur Funktion des Containers erhält.

Einer der wichtigsten Punkte in Dockerfiles ist die Verwendung von COPY statt ADD. Diese beiden Befehle tun zwar dasselbe, sie fügen Ordner oder Dateien in das Docker-Image ein, jedoch akzeptiert ADD im Gegensatz zu COPY auch eine URL als Quelldatei. Weiterhin kann ADD automatisiert Archive entpacken, sodass möglicherweise Angriffsvektoren wie Path Traversal entstehen können. (39) (40) Deshalb sollte COPY verwendet werden, solange die Funktionen von ADD nicht explizit benötigt werden.

Ein weiterer Punkt, der bei Dockerfiles nützlich ist, ist Caching. Dies ist zwar kein sicherheitstechnischer Aspekt, er wird aber trotzdem erwähnt, da immer größere Images zu einer längeren Build-Dauer führen, was in modernen DevOps(Sec)-Prozessen zu Problemen führen kann. Es empfiehlt sich, die Befehle eines Dockerfile vom „am wenigsten ändernden“ zum „am meisten ändernden“ Inhalt zu sortieren. Dies begründet sich darin, dass geänderte Inhalte den Cache bei einem Build-Vorgang invalidieren und der Build somit länger dauert. (siehe Abbildung 16 und 17 in Kapitel 2.2.2). Docker wird bei einer geänderten Schicht alle darüberliegenden Schichten neu erstellen, da diese von der Änderung betroffen sein könnten. Daraus folgt auch, dass bei COPY Befehlen darauf geachtet werden sollte, nur die unbedingt notwendigen Dateien in das Image zu kopieren. Nimmt man als Beispiel an, dass ein Verzeichnis A exisitiert und in diesem Verzeichnis zwei Dateien vorhanden sind: java-application.jar und unnecessary-file.txt.
Nun wird folgender Befehl angenommen: COPY . /A.
Wird nun die Datei unnecessary-file.txt geändert, wird der Cache invalidiert, obwohl eigentlich nur die JAR-Datei für die Java-Anwendung benötigt wird. Daher sollten Befehle in Dockerfiles immer **so präzise wie möglich sein**, um eine Invalidierung des Caches zu verhindern und den Build-Vorgang so zu beschleunigen. Die genaue Funktionsweise des Caches ist in der Dokumentation der Docker-Engine dargestellt. (10 S. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#leverage-build-cache)
Weiterhin gilt es bei Dockerfiles zu beachten, keine unnötigen Abhängigkeiten zu installieren, was zu schnelleren Build-Vorgängen führt und die Angriffsfläche des Images erheblich reduziert.
Es empfiehlt sich außerdem, nach offiziellen Images zu suchen, welche für den Einsatz in Container-Images optimiert sind und in der Regel schon viele Standardeinstellungen enthalten, sodass diese nicht beim Image-Build durchgeführt werden müssen. Zusätzlich profitiert man bei diesen Images von regelmäßigen Updates. So sollte man Java nicht über den Paketmanager (in den meisten Fällen *apt*) installieren. Stattdessen sollte man den FROM-Befehl nutzen: FROM: openjdk:13.
Wie man hier sieht, sollten immer spezifische Tags genutzt werden. Es wäre natürlich auch möglich, *openjdk:latest* zu nutzen; da *latest* allerdings immer die aktuellste Version referenziert, könnte dies zu ungewollten Auswirkungen führen. Generell gilt bei Image-Tags: Je spezifischer, desto besser. Dies gilt auch für Dateigrößen, so sollte

man bei der Verwendung von Tags auch die Dateigrößen der Images prüfen, möglicherweise ist ein kleineres Image verfügbar. So hat das openjdk mit *Alpine*[9] als Base-Image (FROM: openjdk:13-alpine) eine Größe von circa 90 Megabyte, der Tag openjdk:13 produziert ein Image von fast 700 Megabyte Größe.
Es sollte zudem davon abgesehen werden, Konfigurationen oder *Secrets*[10] in Dockerfiles durch Umgebungsvariablen zu spezifizieren. Ein solches Vorgehen würde diese Secrets im Image hinterlegen. Dadurch könnte jeder, der Zugriff auf dieses Image hat, diese Secrets auslesen. Zudem würde ein solches Vorgehen dazu führen, dass ein Dockerfile keine Vorlage mehr ist, um eine spezifische Anwendung zu erstellen, denn die Anwendung wäre dann maßgeblich von den hinterlegten Secrets abhängig. Würden sich also die Secrets ändern, müsste das Dockerfile ebenso geändert werden. Daher sollten Secrets der Anwendung auf anderem Wege bereitgestellt werden, zum Beispiel durch die zu Grunde liegende Infrastruktur (Kubernetes-Secrets etc.) oder eine externe Lösung (z.B. Hashicorp Vault), die dann direkt aus der Anwendung heraus kontaktiert wird. Falls das Secret unbedingt im Dockerfile hinterlegt werden muss, sollten die Docker-eigenen Secrets-Befehle (*--mount=type*=secret) verwendet werden. Dies gilt auch für möglicherweise benötigte SSH-Komponenten, um zum Beispiel Git-Repositories zu klonen. (10)

Dieses Feature ist leider nur in *Docker Swarm* verfügbar. (41) Dort werden alle Secrets verschlüsselt in einer Datenbank gespeichert. Wird ein Secret mit *docker secret create* erzeugt, wird dieses automatisch zur Datenbank des *Docker Swarm* gesendet, dort verschlüsselt und anschließend gespeichert. Die Verbindung ist über mTLS mit Zertifikaten der *Docker Swarm*-eigenen CA gesichert. Als Verschlüsselungsalgorithmus kommt *NaCl Salsa 20* mit einem 256 Bit langen Schlüssel zum Einsatz. Der Docker-Container erhält dann über seinen *Swarm-Node* (dieser authentifiziert sich ebenfalls per mTLS an der Swarm-Datenbank) Zugriff auf das Secret und mountet es unverschlüsselt in den Container.

[9] Minimalistische Linux-Distribution mit circa 5 Megabyte Größe
[10] Geheimnisse, z.B. Zugangsdaten

3.2.2 Image-Signing / Docker Content Trust (DCT)

Der sicherheitstechnische Kernprozess beim Bau eines Images bildet das Signieren des Images, um zu verhindern, dass dieses während des Transports zur Container-Registry manipuliert wird. Dies wird in Docker über *Notary* beziehungsweise *The Update Framework* (TUF) realisiert. Docker bietet diese Sicherheitsmechanismen in ihren eigenen Lösungen unter dem Namen *Docker Content Trust* an, es ist aber auch möglich, diese Mechanismen über die Implementierung *Notary* (42) lokal zu nutzen. TUF wurde nicht auf Docker oder Container spezialisiert eingeführt beziehungsweise entwickelt, es soll das allgemein bestehende Problem lösen, digitale Inhalte zu signieren und den Ersteller zu verifizieren. Docker Trusted Registry beinhaltet *Notary*, sodass dort Docker Content Trust einfach aktiviert werden kann. Für private Container-Registries muss *Notary* möglicherweise manuell installiert werden, weiterhin muss geprüft werden, ob die Registry Docker Content Trust unterstützt. Im Wesentlichen beschreibt Docker Content Trust das Verfahren des Signierens und Prüfens von Images beim Client, Docker Trusted Registry ist Dockers Enterprise-Image-Registry, welche dieses Feature standardmäßig unterstützt. Eine Anleitung zur Nutzung von Docker Content Trust findet sich in der Dokumentation der Docker-Engine. (10) Durch die konsequente Nutzung von Docker Content Trust lässt sich, in Kombination mit Security-Scannern, eine Registry aufbauen, welche unter praktischen Aspekten als sicher erachtet werden kann. Bei mehrschichtigen Images entsteht eine Kette von signierten Images.

Signaturen schützen vor kompromittierten Images, indem die ankommende Signatur mit einer zu erfüllenden Signatur verglichen wird. Stimmen beide überein, handelt es sich (nach aktuellem Stand der Technik, nach welchem es nicht möglich ist, zu einem gegebenen Hashwert eine entsprechende -sinnvolle- Bitfolge zu erzeugen) um das gewünschte Image, falls nicht, wurde das Image entweder kompromittiert oder auf dem Transportweg manipuliert. Dies ist im Docker-Umfeld extrem wichtig, da Docker-Images auf Tags basieren, welche letztendlich Verweise auf Hashwerte in der Docker-Registry darstellen. Es ist jedoch möglich, dass der Tag *image:xy* auf einen anderen Hashwert verweist als erwartet, da jemand ein neues Image mit diesem Tag hochgeladen hat. Dieses Problem wird mit Docker Content Trust gelöst, indem die Zuordnung Tag → Hashwert durch kryptografische Methoden „fixiert“ wird. Es ist sozusagen möglich, die Docker Content Trust Architektur nach dem aktuellsten, signierten Hashwert für einen spezifischen Tag zu fragen.

Die Signaturen basieren in Docker Content Trust auf verschiedenen Schlüsseln, welche in der Dokumentation erläutert werden. (43 S. https://docs.docker.com/engine/security/trust/content_trust/) Das Verfahren weicht maßgeblich von klassischen Signaturverfahren wie RSA oder Hashfunktionen ab (es sind in Docker Content Trust mehrere Hierarchiestufen abgebildet), allerdings sind zwei erfolgreiche Security-Audits für das zu Grunde liegende Verfahren sowie die Implementierung vorhanden. (44) (45) Sowohl *nccgroup*, mit 35 weltweiten Standorten und circa 2000 Angestellten, als auch *cure53*, können als prüfende Firmen als vertrauenswürdig angesehen werden.

Docker Content Trust liegt ein Framework zu Grunde, welches das Ziel hat, sichere Softwareupdates zu ermöglichen und dabei so flexibel wie möglich zu sein. (46)

Docker Content Trust beziehungsweise das darunterliegende Framework schreibt zunächst das Vorhandensein einer *root role* vor. Diese signiert eine Datei *root.json*, welche die öffentlichen Schlüssel **aller anderen Rollen** beinhaltet und damit den *root of trust* bildet. Clients können über diese Datei die Signaturen von anderen Rollen überprüfen. Weiterhin gibt es drei Rollen und dazugehörige Dateien:

- Eine *snapshot role*, die die dazugehörige Datei *snapshots.json* signiert. Diese enthält die Metadaten zu allen Dateien (das Framework legt sich nicht auf einen Begriff wie Docker-Image oder Softwareversion fest, sondern spricht allgemein von *target files*), die Metainformationen enthalten.
- Weiterhin gibt es die *timestamp role*, welche die Datei *timestamp.json* signiert. Diese Datei enthält die Metadaten der aktuellsten *snapshot.json*-Datei. Daraus folgt, dass diese Datei regelmäßig aktualisiert wird. Diese Datei bildet den Kern für die sogenannte *freshness guarantee*, also die Sicherheit, dass die angeforderte Dateiversion auch wirklich die aktuelle ist.
- Schlussendlich gibt es die *target role*, mit der die eigentlichen Dateien signiert werden. Hier ist anzumerken, dass diese Rolle eine Delegation erlaubt, sodass mit dem *target role key* weitere Schlüsselpaare erzeugt werden können, die dann spezifischere Berechtigungen erhalten, zum Beispiel das Signieren von Dateien in einem bestimmten Repository. Der private Schlüssel der *target role* ist auf dem Client gespeichert, damit dieser seine Images signieren kann.

Als Algorithmen kommen entweder RSASA-PSS oder ed25519 zum Einsatz (46), das Framework bietet aber die Möglichkeit, andere Verfahren zu nutzen. In jedem Fall

müssen die Verfahren aber den Grundsätzen der asymmetrischen Verschlüsselung genügen.

Docker Content Trust benötigt zwei weitere Komponenten, um zu funktionieren: Einen *Notary-Server* sowie einen *Notary-Signer*. Ersterer speichert die Metainformationen und gibt diese an die Clients weiter, letzterer signiert die *timestamp.json* sowie die *snapshot.json* mit den entsprechenden privaten Schlüsseln der dazugehörigen Rollen und enthält alle öffentlichen Schlüssel. Der einzige Schlüssel, der nicht in diesem System gespeichert ist, ist der Schlüssel der *root role*. Dieser ist dauerhaft offline und wird nur genutzt, wenn Schlüsselrotation stattfinden soll. Beide Komponenten nutzen getrennte Datenbanken, sodass Schlüssel und Metainformationen nicht auf dem gleichen System vorhanden sind, woraus folgt, dass die Schlüssel nicht in einem „verwundbaren“ Front-End-Service gespeichert sind, die Anfragen der Nutzer (im Docker-Umfeld ist das ausschließlich die Docker-Engine) werden alle an den *Notary-Server* gestellt:

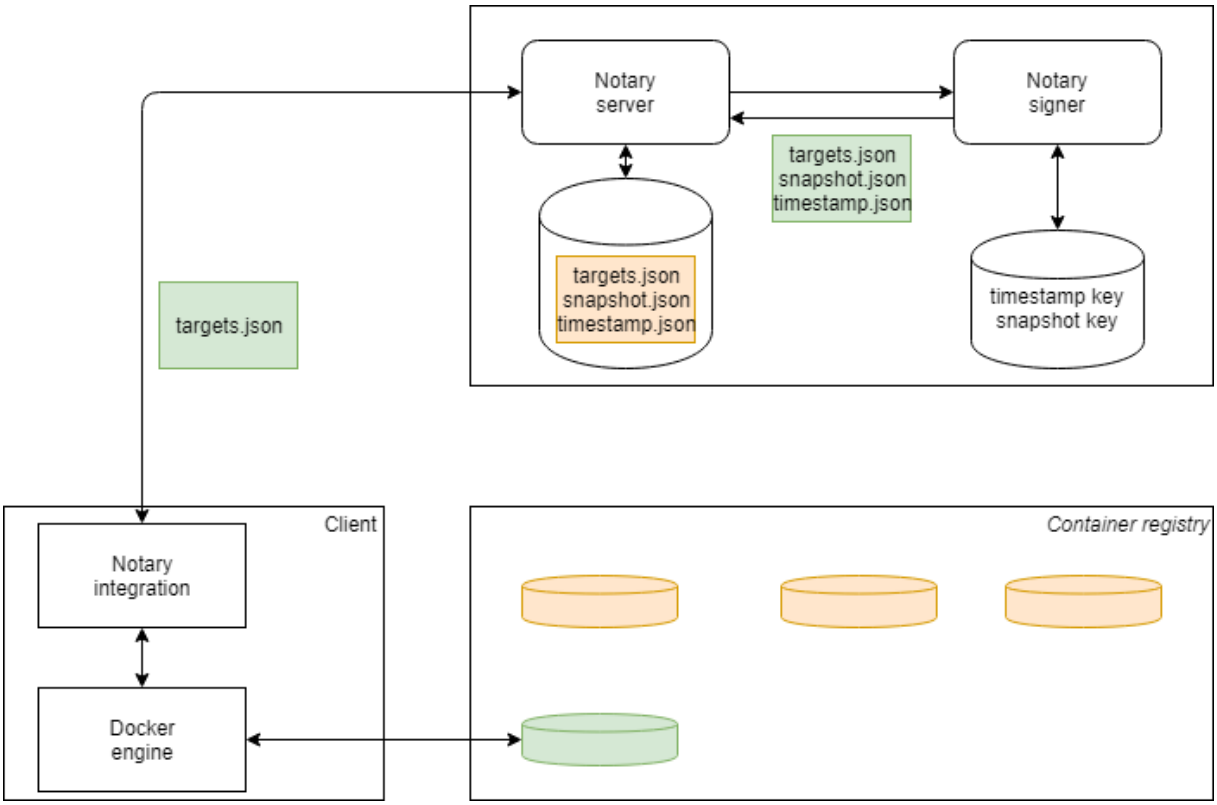


Abbildung 67 Architektur einer Container-Registry bei der Verwendung von Notary

Möchte nun ein Nutzer ein Image in die Container-Registry hochladen, so erstellt er dieses, signiert es mit seinem privaten Schlüssel und erzeugt eine neue *targets.json*-Datei. Diese wird zum Notary-Server gesendet, welcher die Datei prüft und bei erfolgreicher Prüfung neue *snapshot.json*- und *timestamp.json*-Dateien erzeugt. Diese

werden an den Notary-Signer gesendet, dort signiert, zurück an den Notary-Server gesendet und dort gespeichert. Die öffentlichen Schlüssel der Clients liegen ebenfalls in der Datenbank des Notary-Signer, sodass sie zugänglich sind.

Diese Beschreibung bildet nur die prinzipielle Funktionsweise ab, eine weitergehende Betrachtung ist nicht möglich, da das zu Grunde liegende Framework lediglich eine Spezifikation vorgibt. Es gibt zwar eine Open-Source-Implementierung namens *Notary*, allerdings weicht die Docker-interne Implementierung hiervon ab.
Prinzipiell geschieht dieser Prozess in Docker vollautomatisiert, ohne dass der Nutzer eingreifen muss. Die einzige Voraussetzung ist das Vorhandensein einer Docker Trusted Registry sowie das Aktivieren von Docker Content Trust. Dies ist über die Umgebungsvariable *DOCKER_CONTENT_TRUST=1* möglich.
Leider ist es aktuell nicht möglich, Docker Content Trust mit der bei FIRMA eingesetzten Container-Registry *Nexus Repository Manager* zu nutzen, allerdings gibt es bereits ein konkretes JIRA-Ticket, in welchem diese Anforderung bearbeitet wird. (47)
Es sei allerdings erwähnt, dass Docker Content Trust **kein Security-Scanner ist!** Docker Content Trust wird nur prüfen, ob das angeforderte Image tatsächlich das ist, was angefragt wurde und ob der Ersteller validiert ist. Es ist dennoch möglich, dass signierte Images Schadcode enthalten!

3.2.3 Sicherheitsaspekte in Containern

Um Container sicher in produktiven Umgebungen betreiben zu können, ist es wichtig, deren Rechte einzuschränken. Docker ist eine andere Art der Softwareverteilung. Daher sollten Container als Unix-Prozesse angesehen und dementsprechend abgesichert werden.

Weiterhin steht bei Containern vor allem die Absicherung des Host-Systems im Vordergrund, dies sollte nach den Richtlinien der entsprechenden Linux-Distribution geschehen. Dies ergibt sich vor allem aus der Tatsache, dass Docker direkt auf dem Betriebssystem aufsetzt und es somit möglich wäre, aus einem Container heraus auf den Kernel beziehungsweise das Betriebssystem zuzugreifen. Wie bereits in Kapitel 2 erläutert, müssen Betriebssystem des Host und des Containers nicht identisch sein, da beide denselben Kernel nutzen. Daher muss zusätzlich zu Docker auch das

Betriebssystem entsprechend gesichert werden. (48) Zum Absichern des Betriebssystems gehört auch, dafür Sorge zu tragen, dass alle Softwarepakete, **allen voran Docker**, auf dem aktuellsten Stand sind, um Sicherheitslücken zu schließen.

Zusätzlich sollte beachtet werden, dass Docker standardmäßig das Verzeichnis */var/lib/docker* nutzt. Dort wird Docker alle Dateien (außer die Konfiguration des Docker-Daemon), inklusive der Images, ablegen. (10 S. https://docs.docker.com/engine/reference/commandline/dockerd/)
Da dieses Verzeichnis standardmäßig unter / eingespielt wird, kann es vorkommen, dass das Verzeichnis voll wird und dadurch der Host nicht mehr benutzbar ist. Um diesem Sachverhalt vorzubeugen, empfiehlt es sich, eine eigene (logische) Partition anzulegen, die exklusiv von Docker genutzt wird.
Für dieses Vorhaben müssen alle Container sowie der Docker-Service gestoppt werden. Anschließend kann ein Verzeichnis erstellt werden:

```
root@ubuntu:/# mkdir -p /extravolume/docker
root@ubuntu:/# sudo chown root:root /extravolume/docker/ && chmod 701 /extravolume/docker/
```

Abbildung 68 Erzeugen eines neuen Verzeichnisses für Docker

Es muss auf den korrekten Besitzer sowie die korrekten Berechtigungen des Verzeichnisses geachtet werden. Danach kann die Konfiguration des Docker-Daemon entsprechend angepasst werden. Die Datei, welche den Docker-Daemon konfiguriert, findet sich im Verzeichnis */etc/systemd/system/docker.service.d*. Leider existieren diese Datei sowie das dazugehörige Verzeichnis nicht standardmäßig, sodass beides bei erstmaliger Anpassung der Konfiguration des Docker-Daemon erzeugt werden muss. Dafür sind in diesem Verzeichnis natürlich Root-Rechte erforderlich. Nun kann die Konfiguration des Docker-Daemon angepasst werden, unter Linux findet sich diese Datei unter dem Pfad */etc/docker/daemon.json*. Sie kann aber auch an einem anderen Ort vorhanden sein, falls beim Start des Daemon eine andere Konfigurationsdatei angegeben wird. In der Datei kann nun das Argument *data-root* ergänzt werden:

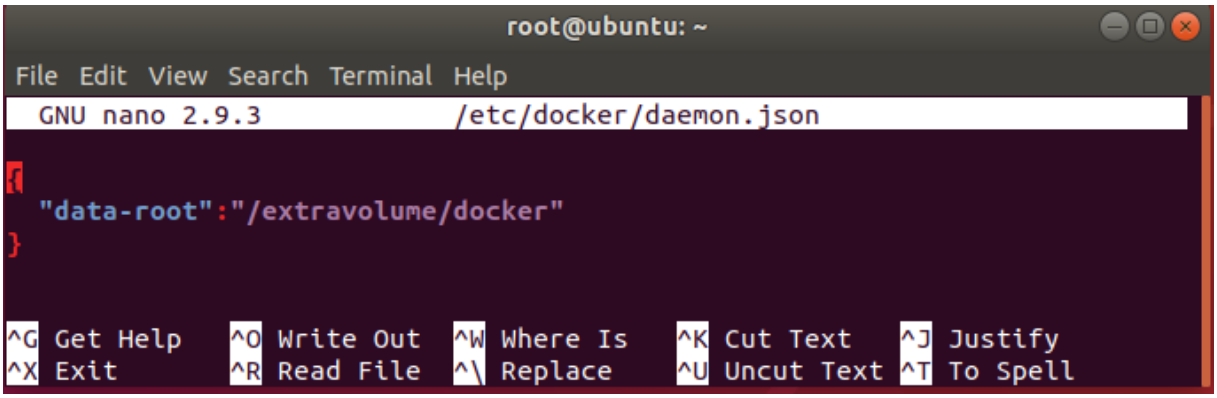


Abbildung 69 Angeben des neuen Pfades des Docker-Verzeichnisses

Nach einem Neustart des Docker-Daemon lässt sich über den Befehl *docker info* herausfinden, dass das Verzeichnis entsprechend geändert wurde:

```
Docker Root Dir: /extravolume/docker
```

Abbildung 70 Docker hat das neue Verzeichnis erkannt

Ein weiterer wichtiger Punkt beim Absichern des Host-Systems ist die Benutzergruppe *docker*. Mitglieder dieser Gruppe werden automatisch *root* und können ohne den *sudo*-Befehl Docker-Befehle ausführen. Die Notwendigkeit, dass der Docker-Daemon als *root* ausgeführt wird, ergibt sich aus Features wie Verzeichnisse in einen Container mounten. Mitglieder der docker-Gruppe können zwar keine anderen Befehle, die eine *root*-Bereichtigung benötigen, ausführen, es ist allerdings ausreichend, einen Docker-Container zu starten und in diesem eine Bash zu starten, um *root*-Zugriff auf den Host zu erhalten. (49)

(10 S. https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface)

Weiterhin ist es dann möglich, das *root*-verzeichnis des Hosts in einen Container einzubinden. (10 S. https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface)

Die Dokumentation von Docker weist darauf hin, **dass nur vertrauenswürdige Benutzer Mitglied der Docker-Gruppe sein sollten**. Standardmäßig enthält diese Gruppe keine Mitglieder. Eine native Möglichkeit in Docker selbst, Docker-Befehle ohne *root*-Rechte auszuführen, gibt es aktuell nur im als experimentell deklarierten *rootless mode*. (50)

(35 S. https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface)

Dieser unterstützt aber aktuell weder Overlay-Netze, noch das Exponieren von Ports größer als 1024. (50) Somit dürfte dieser Modus für produktive Systeme unbrauchbar sein.
Die Problematik der Gruppenzugehörigkeit zur Docker-Gruppe existiert seit dem Bestehen von Docker, aktuelle Workarounds gehen dazu über, einen Eintrag in der Datei */etc/sudoers* zu erstellen, in dem einem spezifischen Nutzer explizit erlaubt wird, auf */usr/bin/docker* zuzugreifen. Diese Möglichkeit erhöht die Sicherheit aber nicht, da sie das gleiche Problem hervorruft: Nicht-privilegierte Nutzer können privilegierte Docker-Container starten, das Root-Dateisystem (*/*) einbinden und im Container verändern. Bei dieser Variante profitiert man lediglich vom zusätzlichen Auditing der sudo-Befehle. Es bleibt abzuwarten, ob sich hier in Zukunft Verbesserungen ergeben. **Aktuell hängt die Sicherheit eines Docker-Hosts von der Vertrauenswürdigkeit seiner Nutzer ab**.

Zuletzt ist bei der Absicherung des Hosts wichtig, dass die entsprechenden Docker-Verzeichnisse überwacht werden. Diese sind:

- Der Docker-Daemon selbst,
- /var/lib/docker (beziehungsweise das *data-root*-Verzeichnis),
- /etc/docker,
- Der Docker-Service und -Socket (docker.service, docker.socket),
- Die Konfigurationsdatei des Docker-Daemons (/etc/docker/daemon.json).

(51)
Wie bereits im vorhergehenden Kapitel gezeigt, ist es möglich, den Docker-Daemon entsprechend eigener Bedürfnisse zu konfigurieren. Hierfür steht eine Reihe von Optionen zur Verfügung.
(10 S. https://docs.docker.com/engine/reference/commandline/dockerd/)
Zunächst sollte das Log-Level (*log-level*) den Anforderungen entsprechend eingestellt werden. Der Standardwert ist *info*, *debug* sollte nur in Ausnahmefällen genutzt werden, da in diesem Modus um einiges mehr Log-Ausgaben produziert werden. Unter Ubuntu ist es zurzeit so, dass das Log-Level ignoriert wird, wenn der standardmäßige Dienst *systemd* zur Aufbereitung der Logs genutzt wird.[11]

[11] https://github.com/docker/for-linux/issues/127

Weiterhin sollte darauf geachtet werden, dass keine unsicheren Registries vorhanden sind, um zu verhindern, dass Docker Images über HTTP heruntergeladen werden oder Man-in-the-Middle-Angriffe (durch fehlerhafte Zertifikate) nicht beachtet werden.
Unsichere Registries werden mit dem Kommando *dockerd --insecure-registry adress:port* oder in der Konfigurationsdatei des Docker-Daemon hinterlegt. Es sollte ebenfalls darauf geachtet werden, dass keine Registries mit der veralteten Version 1 (durch /v1 zu erkennen) genutzt werden. Diese werden in Docker als *legacy registry* bezeichnet und können über das Argument *--disable-legacy-registry* verboten werden.

Der Docker-Daemon lässt sich auch über HTTP(S) nach außen exponieren, um die API-Funktionen auch außerhalb des Hosts bereitzustellen. Die Bereitstellung über HTTPS geschieht implizit über die Angabe zweier Parameter: *--tlsverify* sowie *tlscacert*. Werden diese nicht angegeben, ist der Docker-Daemon über HTTP erreichbar, sodass jeder mit Zugriff auf die IP-Adresse und Port des Daemons diesen steuern kann. (51 S. 45) Das Erzeugen der Zertifikate und Einrichten des Daemon ist sehr ausführlich in der Dokumentation von Docker beschrieben. (10 S. https://docs.docker.com/engine/security/https/)

Da Container zur Laufzeit Systemressourcen benötigen, müssen sie entsprechend beschränkt werden, damit das Host-System nicht überlastet werden kann. So lassen sich CPU, RAM sowie I/O-Operationen für einzelne Container beschränken. Hierfür gibt es sehr gute Anleitungen. (52) (10 S. https://docs.docker.com/config/containers/resource_constraints/)

Falls Verzeichnisse in den Container eingebunden werden müssen, sollte die Angabe so spezifisch wie möglich sein. Falls ein Schreibzugriff nicht unbedingt notwendig ist, sollte das Verzeichnis beim Mounten als Read-only deklariert werden. In docker-compose-Dateien steht hierfür das Feld *read_only* zur Verfügung, für den Befehl *docker run -v* lässt sich das Verzeichnis mit *:ro* Versehen und für das neuere *--mount*-Kommando steht die Option *readonly* bereit. (10 S. https://docs.docker.com/storage/volumes/)

Für tiefergehende Absicherungen ist es ratsam, *AppArmor* oder *SELinux* zu nutzen, hierfür stehen zahlreiche Anleitungen bereit. (10 S.

https://docs.docker.com/engine/security/apparmor/)
Diese Sicherheitsfeatures müssen jedoch sehr spezialisiert auf den Anwendungsfall hin erstellt werden, sodass unter Umständen ein hoher Aufwand erforderlich sein kann, um dies umzusetzen.

4. Evaluation eines geeigneten Security-Scanners

Dieses Kapitel soll sich mit der Auswahl eines geeigneten Security-Scanners für Docker-Images beschäftigen. Zusätzlich soll ausgewertet werden, ob ein ähnliches Tool existiert, um die Absicherung von Kubernetes-Clustern und der Konfiguration von Docker-Hosts zu untersuchen Die Notwendigkeit eines Security-Scanners ergibt sich aus den zahlreichen Schichten, aus denen ein Docker-Image bestehen kann sowie den zahlreichen in dieser Arbeit vorgestellten Sicherheitsmechanismen von Kubernetes. Zusätzlich haben im Jahr 2018 nur knapp 56% der Befragten einer Umfrage von Sonatype angegeben, dass sie ein Security-Tool für ihre Container nutzen. Immerhin hat sich diese Zahl im Vergleich zum Vorjahr 2017 verdoppelt, was ein erfreulicher Trend ist. (53)

4.1 Überblick über verfügbare Angebote

Zunächst müssen kostenlose und kostenpflichtige Angebote unterschieden werden. In diesem Kapitel sollen nur kostenlose Scanner untersucht werden, zusätzlich werden alle kompromittierten Images auch gegen die im FIRMA-Umfeld testweise eingesetzte (kostenpflichtige) Lösung *rapid7 InsightVM* (54) getestet.

Es gibt am Markt unzählige Security-Tools für Docker, allerdings sind nur wenige echte Security-Scanner. Die meisten Tools, die im Kontext „Docker Security“ gelistet werden, sind Tools für Docker und Kubernetes, wie zum Beispiel Netzplugins (Calico, Cilium) oder Hilfsprogramme, um sicherheitsrelevante Daten in Containeranwendungen bereitzustellen, hier sei beispielhaft Hashicorp Vault genannt.

Als kostenlose Security-Scanner sollen in diesem Test *Clair* (entwickelt von coreOS) (55), *Anchore* (56) sowie *docker-bench-security* (ein Docker-eigenes Tool) (43) zum Einsatz kommen. Während Clair und Anchore echte Scanner sind, bietet *docker-bench-security* eine Überprüfung der Konfiguration des Host-Systems sowie des Images und dessen Dockerfile.

Anchore lässt sich direkt an eine vorhandene Container-Registry anbinden (57), *Clair* bietet diese Möglichkeit leider nicht, dieses Tool muss für jedes Image manuell aufgerufen werden.

4.2 Erzeugen von kompromittierten Images

Um ein möglichst aussagekräftiges Testergebnis zu erhalten, soll jeder Scanner mit mehreren Images getestet werden. Diese beinhalten sowohl kompromittierte Base-Images, als auch spezifische Verwundbarkeiten der enthaltenen Softwarepakete. Um ein möglichst neutrales Testergebnis zu erreichen, werden alle Tests in einer lokalen Installation (ausgenommen insightVM) des zu testenden Scanners sowie einer lokalen Container-Registry durchgeführt. Folgende Images werden zum Testen verwendet:

1. openjdk:10-jdk: Ein sehr häufig verwendetes Java-Base-Image
2. elasticsearch:1.4.2: Enthält eine Sicherheitslücke, welche das Auslesen von Information ermöglichen kann
3. ghostscript:9.23-python: Eine Python-basierte Ghostscript-Version, die eine Sicherheitslücke enthält, welche einen Buffer-Overflow ermöglicht
4. jboss:4.0.5: Ein Java-Server, der eine Sicherheitslücke beinhaltet, die es ermöglicht, in einer Shell Befehle zu nutzen, für die möglicherweise keine Berechtigungen vorhanden sind
5. jenkins:2.138: Ein sehr verbreiteteter Buildserver, der eine Sicherheitslücke enthält, über die es möglich ist, Methoden auf Java-Objekten durch manipulierte URLs aufzurufen
6. joomla:3.7.0: Ein Content-Management-System, welches anfällig für SQL-Injection ist
7. mysql:5.5.23: Ein Image der bekannten *MariaDB* (welche zum Beispiel Bestandteil des XAMP/LAMP ist), die in bestimmten Umgebungen eine Umgehung der Autorisierung ermöglicht
8. nginx:heartbleed: Ein Webserver, der die Sicherheitslücke (*Heartbleed-Bug*) enthält
9. openssh:7.7: Ein openSSH-Paket
10. php:5.4.1-cgi: Ermöglicht Code-Ausführung durch manipulierte URLs

Bei der Zusammenstellung der zu testenden Images wurde darauf geachtet, ein möglichst breites Spektrum an Angriffsvektoren sowie eine Mischung aus älteren und neueren Sicherheitslücken zu bilden.

Nun wird jeder Scanner mit diesen Images getestet.

4.2 Definition einer Metrik

Um eine Bewertung vornehmen zu können, ist es nötig, eine Metrik festzulegen. Hierfür stehen zwei mögliche Ansätze zur Verfügung: Man könnte primitiv auswerten, wie viele der vorhandenen Schwachstellen ein Scanner erkennt und daraus eine absolute oder relative Zahl bilden. Bei diesem Ansatz würde aber das Nicht-erkennen einer schwerwiegenden Schwachstelle, wie des Heartbleed-Bugs, nicht stärker ins Gewicht fallen als eine unbedeutendere Sicherheitslücke.
Daher wurde entschieden, die Metrik folgendermaßen festzulegen:

$$Score_{scanner} = \sum_{1}^{i} found_i * CVE_{CVSS-Impact(i)} \ , i < Anzahl\ CVEs\ im\ Image, found\ \in \{0,1\}$$

Der Wert eines spezifischen Scanners berechnet sich aus der Summe der CVSS-Einstufungen aller gefundenen CVEs im Image. Somit ist sichergestellt, dass die Wertigkeit der Schwachstellen ebenfalls in das Gesamtergebnis einfließt. Der Parameter *found* gibt an, ob der Scanner eine bestimmte CVE gefunden hat. Ist dies nicht der Fall, wird der Parameter *0* und das Gesamtergebnis verringert sich entsprechend.

4.3 Testdurchführung

Die Durchführung erfordert zunächst eine Möglichkeit, automatisiert CVSS-Einstufungen zu bekannten CVEs zu erhalten, da alle Images zusammen circa 5500 CVEs enthalten. Hierfür steht nur eine einzige kostenlose API bereit: https://cve.circl.lu/. Diese benötigt allerdings zum Zeitpunkt der Testdurchführung bis zu drei Minuten, um zu einer CVE die passende CVSS-Einstufung zu liefern, da an der Serverinfrastruktur Wartungsarbeiten durchgeführt wurden. Daher wurde dieser Prozess mit *Selenium Webdriver* umgesetzt. Dieses Framework erlaubt das „Fernsteuern“ eines Browsers. So kann die NVD automatisiert durchsucht werden. Die Zeitdauer lässt sich somit auf circa fünf Minuten pro Image pro Scanner reduzieren. Anschließend werden die Ergebnisse automatisiert in eine Tabelle geschrieben und somit grafisch aufbereitet dargestellt. Unit-Tests sichern die Korrektheit.
Zunächst wird der Scanner *Anchore* getestet. Dieser erzielt folgende Resultate, die hier in Kurzform dargestellt werden:

Image name:tag	Found CVEs	Total CVEs	CVE impact sum
openjdk:10-jdk	715	715	4675,2
elasticsearch:1.4.2	163	163	1051,2
ghostscript:9.23-python	202	202	1338,2
jboss:as-4.0.5	540	540	3543,6
jenkins:2.138	555	555	3643,3
joomla:3.7.0	634	634	4115,4
mysql:5.5.23	656	656	4275,7
nginx:heartbleed	527	527	3444,8
openssh:7.7	715	715	4675,2
php:5.4.1-cgi	733	733	4770,6

Tabelle 1 Ergebnisse des Tests

Erfreulicherweise liefern sowohl Anchore, als auch Claire, als auch insightVM das gleiche Resultat und haben alle bekannten CVEs erkannt, welche in den Images vorhanden sind. Leider haben diese Scanner einen gemeinsamen Nachteil: Sie bieten nur statische Analyse gegenüber bekannten Schwachstellen.

Docker-bench-security erkennt zuverlässig, ob gängige Best-Practices in Docker-Images und Dockerfiles eingehalten werden. Weiterhin wird überprüft, ob das Hostsystem gemäß den Best-Practices konfiguriert ist.

Host configuration		
	Eigene Partition für Container	Erkannt
	Neuste Docker-Version	Erkannt
	Auditing für Docker-Daemon eingerichtet	Erkannt
Docker Daemon configuration		
	Keine insecure registries	Erkannt
	Zentrales Logging	Erkannt

	Netzverkehr zwischen Containern beschränkt	Erkannt
Docker image and build file		
	Content Trust aktiviert	Erkannt
	Healtcheck	Erkannt
	COPY statt ADD	Erkannt

Tabelle 2 Resultate für docker-bench-security

Für den Einsatz in produktiven Umgebungen empfiehlt es sich, das Scannen der Container-Images an zentraler Stelle über insightVM oder Anchore durchzuführen. Daraus ergeben sich mehrere Vorteile:

Das Installieren und Integrieren in den CI/CD-Prozess erfordert Aufwand, den einzelne Teams möglicherweise nicht leisten möchten. Dadurch ist die Wahrscheinlichkeit hoch, dass aus zeitlichen Gründen kein Image-Scanner zum Einsatz kommt. Weiterhin ist es möglich, Image-Scanner mit eigenen Regeln zu nutzen, sodass es vorkommen kann, dass verschiedene Teams ihre Images an Hand unterschiedlicher Kriterien überprüfen. Diese beiden Punkte fallen bei der Nutzung von insightVM beziehungsweise Anchore an zentraler Stelle weg. Zusätzlich würde die Nutzung von insightVM dazu führen, dass sich Teams mehr Gedanken über die Sicherheit ihrer Container-Images machen, da alle Images zentral und einheitlich geprüft werden. InsightVM stellt für alle Produkte der Rapid7-Reihe einen universellen WebHook bereit (58), sodass es mit einer eigens entwickelten Software möglich wäre, einzelne Teams automatisiert über die Sicherheitslücken ihrer Container-Images zu informieren.

4.4 Laufzeitbeobachtung mit Falco

Einer der beiden Scanner in Kombination mit *docker-bench-security* ist zwar ein erster Schritt zu einer sicheren Container-Umgebung, hilft aber nicht gegen unbekannte Schwachstellen und ermöglicht keine verhaltensbasierte Erkennung.

Hier schafft das Tool *Falco* (59) Abhilfe. Der Nutzen dieses Tools lässt sich an einem Beispiel zeigen: NodeJS enthält eine „Schwachstelle“, die Remote-Code-Execution zulässt. (60) Der Begriff Schwachstelle ist hier bewusst gewählt. Es ist keine registrierte Schwachstelle, sondern ein durch den Entwickler verursachtes Problem,

indem die *eval*-Funktion falsch benutzt wird. (61) Sie basiert auf einer *immediately invoked function expression*. (62)
Davor wird kein Security-Scanner warnen, denn diese Schwachstelle wird erst zur Laufzeit erkannt werden. Der im Blog von Sysdig beschriebene Angriff wurde nachgestellt und es wurde tatsächlich keine Schwachstelle im Container-Image erkannt. Mit den entsprechenden Falco-Regeln wird eine Warnung ausgegeben und es ist möglich, im laufenden Betrieb des Containers dessen Verhalten nachzuvollziehen.

Falco ermöglicht es, Container zu ihrer Laufzeit zu überwachen und gemäß einem Regelwerk Warnungen auszulösen. Hierbei ist es wichtig klarzustellen, dass *Falco* ein sogenanntes Auditing-Tool ist. Es überwacht und erzeugt Warnungen, im Gegensatz zu Enforcement-Tools wie *AppArmor* oder *SELinux* wird Falco keine Prozesse beenden!
Die Installation von Falco in Kubernetes-Clustern ist gut dokumentiert, es wird empfohlen, Falco als *DaemonSet* auszurollen, sodass auf jedem Node ein Falco-Pod existiert. (63)
Falco lässt sich sehr feingranular konfigurieren und wird mit einem standardmäßigen Regelwerk ausgeliefert, welches eine gute Abdeckung von gängigen Aspekten bietet. Dieses Regelwerk enthält unter anderem Regeln für das Öffnen von sensitiven Dateien, das Öffnen von verschiedenen Shells und Skripten sowie das Kopieren von Dateien. So lässt sich auch in einer standardmäßigen Installation von Falco eine gewisse Überwachung realisieren.
Zusätzlich können, auf den Einsatzbereich angepasste, Regeln angewendet werden. Diese sollten in einer eigenen Datei gesammelt werden, da die Datei mit dem Standardregelwerk sich bei Updates ändern kann. Würde man dort seine eigenen Regeln hinterlegen, würden diese nach einem Update überschrieben werden.
Falco ermöglicht es, drei Typen zu definieren:

1. Macros: Macros bestehen aus einem Namen, welcher später in *Rules* verwendet wird, sowie einer *condition*, die eine Art Bedingung beschreibt, welche erfüllt sein muss.
2. Lists: Falco ermöglicht das Definieren von Listen, um Schreibaufwand zu sparen und die Lesbarkeit von Rules zu erhöhen. Listen bestehen aus einem Namen sowie einem Array von Elementen.

3. Rules: Rules sind die Regeln, die von Falco ausgewertet werden. Sie bestehen aus einem Namen, einer Beschreibung, einer Bedingung (in der dann die bereits erwähnten Makros verwendet werden können), einem Ausgabe-Platzhalter, der festlegt, welcher Text beim Auslösen dieser Regel ausgegeben werden soll sowie eine Priorität der Regel.

Durch diese drei Typen ist es in Falco möglich, sprechende und einfach verständliche Regeln zu erstellen.

So könnte ein Makro, welches die *bin*-Verzeichnisse enthält, folgendermaßen aussehen:

macro: bin_directories

condition: fd.directory in (/bin, /sbin, /usr/bin, /usr/sbin)

Eine Liste, welche alle Shells enthält, könnte folgendermaßen aussehen:

list: known_shells

items: [bash, csh, dash, zsh, ksh, sh, tcsh]

Komplexere Makros, wie das Erkennen von SSH-Verbindungen zu und von nicht erlaubten Hosts, würden sich dann folgendermaßen realisieren lassen:

macro: inbound_outbound_network_connections

condition: >

(((evt.type in (accept, listen, connect) and evt.dir=<)) or (fd.typechar=4 or fd.typechar = 6) and (fd.ip != ”0.0.0.0” and fd.net != “127.0.0.0/8”) and (evt.rawres >= 0 or evt.res = EINPROGRESS)

Die Regel dazu könnte dann folgendermaßen aussehen:

rule: SSH-Connection recognized

desc: Detects any SSH connection

condition: (inbound_outbound_network_connections) and fd.sport=22

output: SSH Connection recognized (command=%proc.cmdline connection=%fd.name user=%user.name container_id=%container.id image=%container.image.repository)

priority: INFO

Der wichtigste Punkt, der bei der Erstellung von Regeln beachtet werden muss, ist,

dass jede Regel mindestens einen Ausdruck *evt.type=?* enthalten muss! Andernfalls wird die Regel ignoriert, da sie nicht ausgewertet werden kann.

Um die Aktionen, welche auf dem System ausgeführt werden, verarbeiten zu können, stellt Falco für jedes Event ein Objekt bereit, dessen Informationen zugänglich sind. Diese Informationen kommen zum einen aus einem speziellen Kernel-Modul, das Systemaufrufe abfängt und auswertet, zum anderen aus den Audit-Logs des kubeapi-Servers. Da Falco zusätzlich noch Pods im User-Space platziert, können so die verschiedenen Event-Quellen mit höheren Informationen angereichert werden. Dadurch kann in Falco-Regeln auf verschiedene Objekte zugegriffen werden (63 S. https://falco.org/docs/rules/supported-fields/):

- fd: FileDescriptor, enthält alle Informationen zu Dateien. Netzverbindungen werden auch durch diese Klasse abgebildet
- proc und thread: Informationen zu Threads und Prozessen
- evt: Das Objekt für Informationen zum zu Grunde liegenden Systemcall
- user und group: Enthält die Informationen zum Benutzer und der Gruppe, mit der eine Aktion ausgeführt wurde
- container: Enthält alle Informationen zum Container, falls eine Aktion aus einem Container heraus ausgeführt wurde
- k8s: Erlaubt Zugriff auf die Daten der entsprechenden Kubernetes-Ressourcen, wie Labels und Namen von Deployments und Pods
- ka: Objekt für die Kubernetes Audit-Logs

So ist es möglich, wie in den oben beschriebenen Regeln auf tiefster Ebene zu filtern und Warnmeldungen zu erzeugen. Diese Meldungen können auf verschiedene Weisen ausgegeben werden, hierfür stehen zum Beispiel das Schreiben auf die Konsole (*stdout*), das Schreiben in eine Datei oder das Senden an ein Programm oder eine URL zur Verfügung. (59 S. https://falco.org/docs/configuration/)
Es ist in jedem Fall darauf zu achten, die Regeln möglichst genau an den Use-Case der Umgebung anzupassen, um unnötige Warnmeldungen und damit einen Verlust des Bewusstseins für die Überprüfung von Containern zu vermeiden. Falco in Kombination mit einem Security-Scanner erhöht die Sicherheit von Kubernetes-Clustern erheblich, da zum einen die Container-Images überprüft werden, zum anderen aber auch zur Laufzeit des Containers Einblicke in dessen Verhalten erlangt

werden können. Allerdings sei angemerkt, dass Falco erst seit Mitte 2018 existiert. Zwar steht mit Sysdig eine Firma hinter dem Open-Source-Projekt, die Zukunft ist aber aufgrund des jungen Alters noch nicht gesichert.

4.5 Statische Kubernetes-Analyse

Ähnlich wie *docker-bench-security* gibt es auch für Kubernetes die Möglichkeit, die Dateien, die zur Erstellung von Ressourcen benötigt werden, zu analysieren. Das Tool *Kubesec* (64) bietet eine Risikoanalyse für YAML-Dateien, welche für Kubernetes bestimmt sind. Dieses Tool ist noch sehr neu und unterstützt zum Stand dieser Arbeit leider noch keine (*Cluster*)*Roles* und (*Cluster*)*RoleBindings*.
Allerdings können Pods, Deployments und DaemonSets überprüft werden. Diese werden von *kubesec* ausgewertet und anschließend mit einer entsprechenden Punktzahl versehen, wobei alle Werte kleiner als null bedeuten, dass die Datei überarbeitet werden sollte. Bei fehlgeschlagenen Prüfungen enthält die Antwort ein JSON-Array mit kritischen Fehlern, die in der Datei gefunden wurden, zum Beispiel wenn ein Container das Privileg *CAP_SYS_ADMIN* erhält.

Weiterhin gibt es noch das Tool *kube-bench*, welches überprüft, ob Kubernetes „sicher“ konfiguriert ist. Dabei wird nach dem CIS-Kubernetes-Benchmark geprüft. (65) Das in Azure bereitgestellte Cluster erreichte eine sehr gute Sicherheitseinstufung.
Die Nutzung dieser Tools ist in jedem Fall empfehlenswert, da die YAML-Dateien in Kubernetes immer größer und -durch die Formatierungsregeln des YAML-Formats bedingt- dadurch auch immer unübersichtlicher werden.

5. Wirtschaftliche Betrachtungen

In diesem Kapitel sollen nun die Betriebskosten für Kubernetes-Cluster analysiert werden. Dies zielt darauf ab, zwei Fragestellungen zu klären:

1. Ist eine lokale Bereitstellung („on-premise“) signifikant teurer als die Bereitstellung in der Cloud?
2. Ist die Strategie „ein Cluster pro Mandant“ signifikant teurer als die Strategie „mehrere Mandanten pro Cluster“?

Um auf FIRMA übertragbare Ergebnisse zu erhalten, soll die Analyse an Hand einer durchschnittlichen Hardwareausstattung stattfinden. Weiterhin richten sich die Hardwareanforderungen auch nach den installierten Plugins, in dieser Analyse werden dies Istio und Calico sein:

Es werden für die lokale Bereitstellung acht Nodes vorgesehen (drei Master-Nodes) und fünf Worker-Nodes) welche folgende Anforderungen erfüllen sollen (alle Angaben beziehen sich auf einen Node):

RAM	CPU	Speicher (HDD)
16 GB	16 Cores	500 GB

Tabelle 3 Hardwareanforderungen an einen Node

Zur Betrachtung der Kosten bei einer Bereitstellung in der Cloud sollen die Google Kubernetes Engine (GKE) und Azure Kubernetes Service (AKS) dienen, zur Analyse der lokalen Kosten werden die benötigte Hardware, die anteiligen Kosten für den Betrieb eines Rechenzentrums, sowie die benötigten Betriebssysteme für die Nodes betrachtet. Die Analyse zieht zunächst keinen Rabatt durch eine fest vereinbarte Laufzeit in der Cloud in Betracht, dies geschieht erst später.

Die Hardwareanforderungen ergeben sich aus der Tatsache, dass auf Worker-Nodes mehrere Pods laufen (dadurch erhöht sich der Speicherbedarf) sowie den vergleichsweise hohen Systemanforderungen durch Istio, da die Mixer-Komponente dort durchgehend Metriken berechnet und an die Envoy-Proxies meldet.

Für Cloud-Anbieter ergeben sich folgende Hardware-Spezifikationen, alle Details der Tabelle 3 beziehen sich ebenfalls auf einen Node. Um Vergleichbarkeit mit lokaler Hardware zu erhalten, werden Kubernetes-Cluster mit einer 100-prozentigen Auslastung über 24 Stunden und sieben Tage pro Woche berechnet. Hochverfügbare Master-Nodes sind in diesen Preisen bereits inbegriffen und müssen nicht extra gemietet werden. Auch hier sind alle Angaben pro Node dargestellt:

Anbieter	Google	Microsoft Azure
Instance type	N1-standard-16 (60 GB RAM, 16 vCPUs)	B16MS (64GB RAM, 16vCPUs)
SSD	1 mal 375GB	1 mal 128 GB
Location	Frankfurt (europe-west3)	Europa, Westen
Laufzeit	24x7/356	24x7/365
Persistance Disk	4 TB (8*500 GB)	4 TB (8*512 GB)
Persistence Disk location	Frankfurt (europe-west3)	Europa, West
Snapshot storage (Backup)	0 GB	0 GB

Tabelle 4 Systeminformationen pro Node in GKE und AKS

Hier müssen noch Kosten für einen clusterweiten Loadbalancer addiert werden:

Loadbalancer Region	Frankfurt (europe-west3)	Europa, West
Loadbalancer forwarding rules	30	30
Loadbalancer Network Traffic pro Monat	50 GB	50 GB

Tabelle 5 Kosten für einen Loadbalancer

Daraus ergeben sich folgende Teilkosten für fünf Nodes:

Anbieter	Google	Microsoft Azure
Kubernetes-Engine	2.410,19€/Monat	2.409,27€/Monat
Load-Balancing	216,96€/Monat	169,50€/Monat
Storage	176,70€/Monat	146,80€/Monat
Monatliche Gesamtkosten	2.803,60€/Monat	2.725,27€
Jährliche Gesamtkosten	33.643,20€	32.706,84€

Tabelle 6 Gesamtkosten für GKE beziehungsweise AKS

Google ist bei vergleichbarer Leistung der Nodes circa 1000€ pro Jahr teurer, allerdings ergibt sich dieser Wert aus den deutlich höheren Kosten für Speicher und Loadbalancer. Berechnungen anderer Onlinemagazine ergaben ähnliche Preisstrukturen. (66)

Potenzial zur Einsparung bieten die Rabattprogramme der Anbieter, Azure gewährt zum Stand der Erstellung der Arbeit bei einer Reservierung von einem Jahr einen

Rabatt von 41%, bei drei Jahren sogar 71% auf die Kubernetes-Instanzen. Bei Googles GKE stellt sich die Rabattstruktur ähnlich dar.

Ein lokales Setup erfordert pro Cluster eine RedHat Enterprise Linux-Lizenz pro Node. Die Pods, welche auf den Nodes laufen, werden durch ihre Container-Images mit einem Betriebssystem ausgestattet, in der Regel ist dies ein Ubuntu. Dadurch fallen hier nur in Ausnahmefällen Lizenzgebühren an. Für virtualisierte Nodes fallen eventuell noch weitere Kosten für einen Hypervisor an, bei FIRMA wäre dies VMWare vSphere ESXi.
Die Kosten für eine RedHat Enterprise-Lizenz belaufen sich auf 99USD pro Stück (67) Somit ergeben sich Lizenzkosten von 706€. Diese Lizenzen sind für ein Jahr gültig.
Zu diesen Kosten müssen noch Kosten für die Server addiert werden. Hierfür soll ein kompletter Sever betrachtet werden. Es soll mit einem Austausch der Server-Hardware nach fünf Jahren geplant werden, was ein gängiger Abschreibungszeitraum ist. Ein fertig konfigurierter Server von Dell der Serie Smart Value PowerEdgeR630 mit zwei Intel Xeon E5-2698-Prozessoren (zweimal 20 Prozessorkerne), 144GB RAM sowie 2TB HDD kostet regulär 15.801€ pro Stück. Allerdings müssen von diesen Servern zwei Stück erworben werden, um Redundanz zu erhalten. Die Nodes und Master des Clusters werden dann über Virtualisierung realisiert, um eine optimale Auslastung der Hardware zu erzielen. Insgesamt fallen für beide Server 30.962€ an Kosten an. Dadurch ergeben sich folgende Hardwarekosten pro Jahr:

Posten	Kosten
Lizenzen	706 €
Server (pro Jahr bei fünf Jahren Abschreibung)	6.192,40€
Jährliche Gesamtkosten	**6.898,40€**

Tabelle 7 Jährliche Gesamtkosten für lokale Bereitstellung

Im Monat fallen so circa 574,86€ an Kosten an. Dieses Beispielszenario dient nur zur Abschätzung, ob Cloud-Dienste teurer als eine lokale Bereitstellung sind. Um die Betriebskosten für die Rechenzentrumsinfrastruktur abschätzen zu können, wird die Bereitstellung dieses Setups in einem regionalen Rechenzentrum betrachtet. Dort kosten 47 Höheneinheiten (HE) Rackspace 550€ pro Monat, während der Strom mit 26 Cent pro Kilowattstunde berechnet wird. Mit diesen Preisen sind alle Kosten des Rechenzentrums abgedeckt, allerdings wird dort kein 24/7-Operating geboten. Der

ausgewählte Server verbraucht unter Volllast 750 Watt; um eine durchschnittliche Nutzung abzudecken, wird mit einer Auslastung von zwei Drittel der Maximalleistung gerechnet. Damit ergeben sich 500 Watt Leistung pro Server, die im Monat 187,20€ an zusätzlichen Kosten verursachen. Hinzu kommen anteilige Kosten für zwei Höheneinheiten die somit 23,40€ im Monat kosten. Dadurch addieren sich die Gesamtkosten auf **circa 785,46€ im Monat**. Zum Vergleich: Googles „Topmodell“ mit 96vCPUs und 86 GB RAM würde alleine im Monat 18.200,20 USD (circa 16.200€) kosten. (68) Allerdings geben weder Google noch Microsoft Auskunft darüber, welche Hardwarekomponenten verbaut sind, sodass diese Preise **nur zum Abschätzen in Relation zueinander** gesetzt werden können. Es ist aber davon auszugehen, dass eine lokale Bereitstellung zum jetzigen Zeitpunkt günstiger als eine Bereitstellung in der Cloud ist. Selbst mit einer zugesagten Nutzung von drei Jahren würden sich die Kosten für das gezeigte Cluster in der Google Kubernetes Engine nur auf circa 1.800€ pro Monat senken lassen, es wäre also immer noch circa 1000€ teurer als eine lokale Bereitstellung. Allerdings ist hier entscheidend, wie sich die Rabattstruktur und das allgemeine Preisniveau der Cloud-Anbieter in Zukunft entwickeln wird, eine Bereitstellung in der Cloud könnte aber vor allem für Anwendungen mit niedrigen oder mittleren Hardwareanforderungen eine Alternative darstellen. Bei einer lokalen Bereitstellung muss allerdings beachtet werden, dass das Kubernetes-Setup **manuell** durchgeführt werden muss, weiterhin muss die Instandhaltung der Server selbst durchgeführt werden oder ein Wartungsvertrag abgeschlossen werden. Diese sind im FIRMA-Umfeld üblich und kosten pro Jahr erfahrungsgemäß ungefähr 15% des Anschaffungspreises des Servers, in diesem Beispiel würden die Kosten 197,51€ pro Server und Monat betragen. Wird ein solcher Wartungsvertrag mit in die Berechnung einbezogen, erhöhen sich die Kosten auf circa 1.180,49€, was immer noch weit unter dem aktuell erreichbaren Preis für eine Bereitstellung in der Cloud liegt.

Um die Frage zu beantworten, ob ein Mehrmandantencluster in der Cloud signifikant günstiger als ein Cluster pro Mandant ist, muss zunächst klargestellt werden, dass sich diese Kosten nur sehr grob und nicht an Hand konkreter Zahlen abschätzen lassen. Es gibt zum Stand der Erstellung dieser Arbeit keine zuverlässigen Quellen, aus denen hervorgeht, wie eine Abrechnung in der Cloud in einem solchen Mehrmandantencluster durchgeführt wird. Die Kosten lassen sich aber anhand einer Beispielrechnung abschätzen: Es werden sowohl bei Google als auch bei Microsoft

zwei Cluster konfiguriert: ein Cluster mit fünf Nodes sowie ein Cluster mit 25 Nodes. Es werden dieselben Modelle beziehungsweise Node-Typen betrachtet, wie in Tabelle vier. Vom Cluster mit fünf Nodes werden fünf Stück gekauft, vom Cluster mit 25 Nodes nur eines. Für dieses Szenario erhält man folgende Kosten:

Anbieter/Cluster-Typ	5*5 Nodes	1*25 Nodes	Faktor
Google	12.242,25€	11.242,26€	1,08895
Microsoft	12.046,35€	12.046,37€	0,99999

Tabelle 8 Vergleich zwischen ein Cluster pro Mandant und Mehrmandantencluster

Berechnungen für noch größere Cluster mit bis zu 100 Nodes ergeben ähnliche Faktoren bezüglich der Berechnung 20 Cluster mit fünf Nodes gegenüber ein Cluster mit 100 Nodes, sodass davon ausgegangen werden kann, dass der Preis für die Komponenten der Control Plane auf jeden Node umgelegt wird. Damit ist die Strategie „ein Mandant pro Cluster“ nicht teurer als ein Mehrmandantencluster.

6. Ausblick

Wie alle Technologien unterliegen auch Docker und Kubernetes dem Wandel der Zeit. Kubernetes hat Docker-Containern zu einem weiteren Aufschwung verholfen und insbesondere das Entwickeln von Microservices deutlich vereinfacht, indem ein gut abstrahiertes Netzmodel eingesetzt wird, welches Service Discovery über DNS ermöglicht. Tools wie *Eureka* gehören damit der Vergangenheit an, weiterhin haben Container die Wiederverwendbarkeit von Software deutlich erhöht.

Allerdings bringen Container immer noch einige Fallstricke mit sich, hier seien beispielsweise die heikle Rechtesituation des Docker-Daemon oder die Tatsache, dass bei der Verwendung von IPv6 in Docker-Containern alle Ports nach außen freigegeben werden, sowie eine schwierige Situation der Mehrmandantenfähigkeit genannt. Weiterhin bleibt es spannend abzuwarten, ob Container in Zukunft wirklich als „leichtgewichtige VMs“ dienen können, das bedeutet, dass ein Docker-Image einen eigenen Kernel besitzt. Damit wäre auch implizit eine echte Mehrmandantenfähigkeit umgesetzt.

Allerdings wird die Konkurrenz für Docker zunehmend größer, dies wurde auch durch die Einführung des Container Runtime Interfaces in Kubernetes im Jahr 2016 begünstigt. Alternativen wie *rkt* von CoreOS oder *cri-o* wachsen ständig. Während Docker im Jahr 2017 mit über 99% Marktanteil noch ein riesiges Monopol hatte (69), entfallen im Jahr 2018 nur noch 83% auf Docker, der größte Konkurrent ist *rkt* mit 13%. (70)

Auch Kubernetes konnte seinen Marktanteil ausbauen, er beträgt nun 51%, gefolgt von Docker Swarm mit 11%. Die Zukunft von Kubernetes hängt maßgeblich von der Entwicklung der Mehrmandantenfähigkeit ab, da dieses Feature das Einzige ist, was Kubernetes nicht abdecken kann. Dies begründet sich darin, dass Kubernetes auf Containerarchitekturen aufbaut und dadurch die damit verbundenen Probleme in Bezug auf die Mehrmandantenfähigkeit übernommen hat.[12] Diese ist aktuell nur mit enormem Verwaltungsaufwand und fraglicher Sicherheit zu erreichen. Weiterhin sind an Kubernetes Verbesserungen im Bereich der Kryptografie zu erwarten. Hier seien die fehlende Certificate Revocation List sowie das standardmäßige Absichern aller Verbindungen über HTTPS genannt. Zu diesem Schluss kommt auch ein kürzlich

[12] Ein sehr interessanter Ansatz, um das Problem Mehrmandantenfähigkeit in Kubernetes zu lösen, findet sich hier: https://blog.jessfraz.com/post/hard-multi-tenancy-in-kubernetes/

veröffentlichter Security Audit zu Kubernetes, welcher acht Wochen dauerte. (71) Weiterhin werden dort auch erhebliche Verbesserungen am Quellcode vorgeschlagen, sowie einige teils höchstkritische Sicherheitslücken enthüllt. (71) So nutzt der API-Server zwar mTLS, um mit dem *kubelet* zu kommunizieren, allerdings wird in der entsprechenden Implementierung der Parameter *InsecureSkipVerify: true* gesetzt, sodass Man-in-the-Middle-Angriffe problemlos möglich sind und nicht erkannt werden können. (71 S. 38) Dazu kommen kleinere, weniger kritische, dafür aber sinnbildliche Probleme, wie das Auslesen von Nutzereingaben ohne Prüfung: so wird der Port für Pods bei Nutzung eines imperativen Kommandos in *kubectl* nicht geprüft, sondern die Nutzereingabe einfach entgegengenommen und in einen Integer-Wert überführt. Der Wert 2147483647 würde somit zu Port 80 führen. (71 S. 43)

Durch den Security-Audit und die dadurch bekannt gewordenen Probleme sind in Kubernetes in Zukunft Änderungen zu erwarten, es bleibt abzuwarten, ob auch das Kubernetes-Ökosystem und die dazugehörigen Abläufe überdacht und möglicherweise vereinfacht werden.

7. Checkliste Kubernetes und Docker

☐ Dashboard nicht nach außen bereitgestellt, eigenen Serviceaccount gemäß Least-Privilege-Prinzip angelegt, aktuellste Dashboardversion

☐ Alle Kubernetes-Dateien prüfen! Keine heruntergeladenen Dateien ohne vorherige Prüfung auf das Cluster anwenden

☐ Kubernetes-Dateien möglichst spezifisch, keine große Datei für „alles“ (hilft enorm, wenn einzelne Dinge geändert werden müssen)

RBAC *Kapitel 3.1.2*

☐ RBAC korrekt konfiguriert: Alle Nutzer gemäß Least-Privilege-Prinzip berechtigt, keine „all-in-one“-Benutzer

☐ Geeignete Authentifizierungsmethode eingerichtet (inklusive Backup-Zertifikat für mindestens einen Cluster-Administrator)

☐ Serviceaccounts für Deployments bzw. Pods erstellt und eingebunden

Namespaces *Kapitel 3.1.3*

☐ Keine Deployments bzw. Pods im default- oder kube-public-Namespace, falls nicht ausdrücklich erwünscht

☐ Nicht alleinig auf Namespaces zur Trennung vertraut

☐ Keine leeren oder ungenutzten Namespaces

Network-Policies *Kapitel 3.1.4*

☐ CNI-Plugin installiert (ohne dieses funktionieren Network-Policies nicht!)

☐ Default *deny-all*-Regel für **jeden** Namespace erstellt (Network-Policies sind *namspespaced objects*, mit Außnahme von CRDs)

☐ DNS in jedem Namespace freigegeben, mindestens Port 53 UDP, bei vermehrt auftretenden „großen“ Antworten (viele IPv6-Adressen, DNS over TLS, allgemein alle Antworten, die größer als 512 Bytes sind) auch über Port 53 TCP

☐ Wandel von alles erlaubt (ohne mindestens eine aktive Network-Policy) zu alles verboten, was nicht explizit erlaubt ist (mit mindestens einer expliziten Network-Policy **und** einem CNI-Plugin, welches diese durchsetzt)

☐ Regelmäßige Überprüfung der Network-Policies auf Funktion und nach jedem Update des CNI-Plugin sowie nach jedem Anwenden einer neuen Network-Policy oder einer Änderung an einer bestehenden Network-Policy

Pod-Security-Policies *Kapitel 3.1.5*

☐ Default-Policy erstellt

☐ Serviceaccounts für passende Policies autorisiert

☐ Möglichste keine privilegierten Container zugelassen

☐ Plausible Policies: Kein *runAsUser: MustRunAsNonRoot*, aber *userid: 0* erlaubt, kein *allowPrivilegeEscalation: true* mit *privileged: false*

☐ Wirksamkeit von Pod-Security-Policies regelmäßig prüfen

TLS everywhere *Kapitel 3.1.6*

☐ Ingress-Ressourcen oder CNI-Ressourcen mit korrekten Zertifikaten konfiguriert

☐ Verschlüsselung bis auf Pod-Ebene

Docker *Kapitel 3.2*

☐ Best-Practices in Docker-Files beachtet

☐ Docker-Host und Docker-Image mit *docker-bench-security* auf Best-Practices prüfen

Security-Scanner *Kapitel 4*

☐ Security-Scanner mit automatischem Scan für jedes neue Image in der Container-Registry nutzen

☐ Auswerten der Scans und entsprechende Aktionen einleiten

☐ Monitoring-Tool für Kubernetes eingerichtet (zum Beispiel Falco) und regelmäßiges Auswerten der Logdateien

☐ Kubernetes-Dateien mit entsprechendem Tool geprüft

I. Abbildungsverzeichnis

II. Tabellenverzeichnis

III. Literaturverzeichnis

1. **GitHub Inc.** [Online] 2018. [Zitat vom: 23. 05 2019.] https://octoverse.github.com/projects.

2. **The Authors of Kubernetes.** [Online] 26. 03 2019. [Zitat vom: 23. 05 2019.] https://kubernetes.io/de/docs.

3. **Liebel, Oliver.** *Skalierbare Container-Infrastrukturen: Das Handbuch für Administratoren.* Bonn : Rheinwerk Computing, 2019. 978-3-8362-6385-6.

4. **The Kubernetes Authors.** [Online] 2019. [Zitat vom: 06. 05 2019.] https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/.

5. **neokyle.** Stackoverflow. *What's the difference between ClusterIP, NodePort and LoadBalancer service types in Kubernetes?* [Online] 09. 09 2018. [Zitat vom: 12. 07 2019.] https://stackoverflow.com/questions/41509439/whats-the-difference-between-clusterip-nodeport-and-loadbalancer-service-types.

6. **Kim, Andrew Sy.** asykim. [Online] 20. 08 2018. [Zitat vom: 15. 07 2019.] https://www.asykim.com/blog/deep-dive-into-kubernetes-external-traffic-policies.

7. **Lewis, Ian.** [Online] 10 2017. [Zitat vom: 15. 07 2019.] https://www.ianlewis.org/en/almighty-pause-container.

8. **Paris, Eric.** Google Groups. *What is the role of 'pause' container?* [Online] 25. 09 2014. [Zitat vom: 15. 07 2019.] https://groups.google.com/forum/#!topic/kubernetes-users/jVjv0QK4b_o.

9. **Goldberg, Robert P.** *Architecture of virtual machines.* Harvard University Cambridge : s.n., 1973.

10. **Docker.** Docker Engine. *Reference.* [Online] 2019. [Zitat vom: 09. 07 2019.] https://docs.docker.com/.

11. **Docker Inc.** Docker. [Online] 2019. [Zitat vom: 01. 08 2019.] https://www.docker.com/.

12. **National Institute of Standards and Technology.** National Vulnerability Database. *CVE-2018-18624 Detail.* [Online] 01. 31 2019. [Zitat vom: 11. 06 2019.] https://nvd.nist.gov/vuln/detail/CVE-2018-18264.

13. **The Authors of Kubernetees.** [Online] 12. 12 2018. [Zitat vom: 12. 06 2019.] https://github.com/kubernetes/dashboard/releases.

14. **The Kubernetes Authors.** [Online] 29. 01 2019. [Zitat vom: 11. 06 2019.] https://github.com/kubernetes/dashboard/wiki/Installation.

15. **Linux Foundation.** LFS458: Kubernetes Administration. *Version 1.13.1.* 2019.

16. **Oiwa, Y, et al.** IETF. *RFC 8120: Mutual Authentication Protocol for HTTP.* [Online] 04 2017. [Zitat vom: 13. 06 2019.] https://tools.ietf.org/html/rfc8120.

17. **Bundesamt für Sicherheit in der Informationstechnik.** BSI. *Technische Richtlinie.* [Online] 22. 02 2019. [Zitat vom: 13. 06 2019.] https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=10.

18. **Kaliski, Burt.** RSA Laboratories. [Online] 06. 05 2003. [Zitat vom: 14. 06 2019.] https://web.archive.org/web/20170417095741/https://www.emc.com/emc-plus/rsa-labs/historical/twirl-and-rsa-key-size.htm.

19. **Agence nationale de la sécurité.** Référentiel Général de Sécurité. *Version 2.0.* [Online] 21. 02 2014. [Zitat vom: 14. 06 2019.] https://www.ssi.gouv.fr/uploads/2014/11/RGS_v-2-0_B1.pdf.

20. **ECRYPT – Coordination & Support Action.** Algorithms, Key Size and Protocols Report (2018). [Online] 28. 02 2018. [Zitat vom: 14. 06 2019.] http://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf.

21. **Rockoff, Bryant.** Stackoverflow.com. [Online] 19. 05 2016. [Zitat vom: 15. 06 2019.] https://stackoverflow.com/questions/36919323/how-to-revoke-signed-certificate-in-kubernetes-cluster.
22. **OpenID.** OpenID. *Wiki.* [Online] 2011. [Zitat vom: 15. 06 2019.] http://wiki.openid.net/w/page/12995226/Run%20your%20own%20identity%20server.
23. **Keycloak, sponsored by RedHat.** [Online] [Zitat vom: 17. 06 2019.] https://www.keycloak.org/.
24. **Mladenov, Vladislav und Mainka, Christian.** [Online] 13. 01 2017. [Zitat vom: 2019. 06 2019.] https://www.nds.ruhr-uni-bochum.de/media/ei/veroeffentlichungen/2017/01/13/OIDCSecurity_1.pdf.
25. **OpenID Connect.** Core 1.0 incorporating errata set 1. [Online] 08. 11 2014. [Zitat vom: 17. 06 2019.] https://openid.net/specs/openid-connect-core-1_0.html#IDTokenValidation.
26. **OAuth.** OAuth Dokumentation. [Online] [Zitat vom: 15. 06 2019.] https://www.oauth.com/oauth2-servers/making-authenticated-requests/refreshing-an-access-token/.
27. **The Authors of Kubernetes.** [Online] 24. 05 2017. [Zitat vom: 13. 06 2019.] https://github.com/kubernetes/examples/blob/master/guestbook/php-redis/controllers.js.
28. **OWASP.** KALI-Tools. [Online] [Zitat vom: 13. 06 2019.] https://tools.kali.org/web-applications/dirbuster.
29. **NETSEC.** [Online] [Zitat vom: 13. 06 2019.] https://netsec.ws/?p=331.
30. **Google.** [Online] 19. 03 2019. [Zitat vom: 17. 06 2019.] https://cloud.google.com/kubernetes-engine/docs/concepts/multitenancy-overview?hl=de.
31. **Oppenheimer, David.** Youtube. *Multi-Tenancy in Kubernetes: Best Practices Today, and Future Directions - David Oppenheimer.* [Online] Cloud Native Computing Foundation, 06. 05 2018. [Zitat vom: 17. 06 2019.] https://www.youtube.com/watch?v=xygE8DbwJ7c.
32. **Balkan, Ahmet.** Securing Cluster Networking with Network Policies - Ahmet Balkan, Google. *Cloud Native Computing Foundation.* [Online] 15. 12 2017. [Zitat vom: 17. 06 2019.] https://www.youtube.com/watch?v=3gGpMmYeEO8.
33. —. Github. *kubernetes-network-policy-recipes.* [Online] 25. 10 2018. [Zitat vom: 18. 06 2019.] https://github.com/ahmetb/kubernetes-network-policy-recipes.
34. **Bischoff, Maximilian.** Innovex Blog. [Online] 10. 01 2019. [Zitat vom: 18. 06 2019.] https://www.inovex.de/blog/test-kubernetes-network-policies/.
35. **Jetstack.** Github. [Online] 06. 09 2018. [Zitat vom: 26. 06 2019.] https://github.com/jetstack/kube-lego.
36. **Istio Authors.** Istio Documentation. *Version 1.2.2.* [Online] 24. 06 2019. [Zitat vom: 30. 06 2019.] https://istio.io/docs/tasks/security/authn-policy/#globally-enabling-istio-mutual-tls.
37. **Fullton, Scott M.** thenewstack.io. *Dockercon 2017: How Docker Changed Windows So Windows Could Change Docker.* [Online] 24. 04 2017. [Zitat vom: 01. 08 2019.] https://thenewstack.io/docker-changed-windows-windows-change-docker/.
38. **Cimpanu, Catalin.** bleepingcomputer.com. [Online] 13. 06 2018. [Zitat vom: 02. 07 2019.] https://www.bleepingcomputer.com/news/security/17-backdoored-docker-images-removed-from-docker-hub/.
39. **Synk Ltd.** Blog. *10 Docker Image Security Best Practices.* [Online] 06. 03 2019. [Zitat vom: 01. 08 2019.] https://snyk.io/blog/10-docker-image-security-best-practices/.
40. —. Zip Slip Vulnerability. [Online] 15. 04 2018. [Zitat vom: 01. 08 2019.] https://snyk.io/research/zip-slip-vulnerability.

41. **Li, Ying.** Docker Blog. *Introducing Docker Secrets Management.* [Online] 09. 02 2017. [Zitat vom: 01. 08 2019.] https://blog.docker.com/2017/02/docker-secrets-management/.
42. **The Update Framework (TUF).** Notary. [Online] [Zitat vom: 11. 07 2019.] https://github.com/theupdateframework/notary.
43. **Docker.** Github. *docker/docker-bench-security.* [Online] 11. 07 2019. [Zitat vom: 18. 07 2019.] https://github.com/docker/docker-bench-security.
44. **Heiderich, M Dr.-Ing., et al.** GitHub. *theupdateframework/notary.* [Online] 07. 08 2018. [Zitat vom: 01. 08 2019.] https://github.com/theupdateframework/notary/blob/master/docs/resources/cure53_tuf_notary_audit_2018_08_07.pdf.
45. **Ritter, Tom, Kircanski, Aleks und Curtis, Tyler.** GitHub. *theupdateframework/notary.* [Online] 31. 07 2015. [Zitat vom: 01. 08 2019.] https://github.com/theupdateframework/notary/blob/master/docs/resources/ncc_docker_notary_audit_2015_07_31.pdf.
46. **Framework, The Update.** GitHub. *theupdateframework/tuf/docs/spec.txt.* [Online] 25. 02 2016. [Zitat vom: 05. 08 2019.] https://github.com/theupdateframework/tuf/blob/1bed3e09a478c2c918ffbff10b9118f6e52ee129/docs/tuf-spec.txt.
47. **Sonatype.** JIRA. *NEXUS-1136: Docker Content Trust.* [Online] 16. 07 2019. [Zitat vom: 05. 08 2019.] https://issues.sonatype.org/browse/NEXUS-11336.
48. **Bundesamt für Sicherheit in der Informationstechnik.** [Online] 15. 05 2018. [Zitat vom: 30. 07 2019.] https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/IT-Grundschutz-Modernisierung/BS_Container.pdf?__blob=publicationFile&v=4.
49. **Foster, Chris.** Privilege escalation via Docker. [Online] 22. 04 2015. [Zitat vom: 01. 08 2019.] https://fosterelli.co/privilege-escalation-via-docker.html.
50. **Docker.** Github. *engine/docs/rootless.md.* [Online] 03. 06 2019. [Zitat vom: 30. 07 2019.] https://github.com/docker/engine/blob/v19.03.0-rc3/docs/rootless.md.
51. **Center for Internet Security.** *CIS Docker 1.13.0 Benchmark.* [Hrsg.] Center for Internet Security. 1.0.0. 2017.
52. **Goldmann, Marek.** Blog. *Resource management in Docker.* [Online] 11. 09 2014. [Zitat vom: 30. 07 2019.] https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/.
53. **Sonatype.** Blog. *DevSecOps and Containers: The Numbers Don't Lie.* [Online] 17. 04 2018. [Zitat vom: 18. 07 2019.] https://blog.sonatype.com/numbersdontlie.
54. **Rapid7.** InsightVM. [Online] [Zitat vom: 18. 07 2019.] https://www.rapid7.de/products/insightvm/.
55. **coreos.** Github. *coreos/clair.* [Online] 18. 07 2019. [Zitat vom: 18. 07 2019.] https://github.com/coreos/clair/.
56. **Anchore Inc.** Anchore. *Documentation.* [Online] 11 2018. [Zitat vom: 18. 07 2019.] https://docs.anchore.com/current/docs/.
57. **Anchore Admin.** [Online] 06. 11 2018. [Zitat vom: 06. 08 2019.] https://anchore.freshdesk.com/support/solutions/articles/36000003888-configuring-access-to-registries.

58. **Rapid7.** Documentation. *Universal Webhook.* [Online] [Zitat vom: 29. 07 2019.] https://insightidr.help.rapid7.com/docs/webhook.
59. **Sysdig Inc.** sysdig. *open source: Falco.* [Online] 2019. [Zitat vom: 23. 07 2019.] https://sysdig.com/opensource/falco/.
60. **Sysdig Inc.** Blog. *Detecting Cryptojacking with Sysdig's Falco.* [Online] 13. 03 2018. [Zitat vom: 25. 07 2019.] https://sysdig.com/blog/detecting-cryptojacking-with-sysdigs-falco/.
61. **OpSexX.** opsecx. *Blog.* [Online] 08. 02 2017. [Zitat vom: 23. 07 2019.] https://opsecx.com/index.php/2017/02/08/exploiting-node-js-deserialization-bug-for-remote-code-execution/.
62. **Alman, Ben.** benalman.com. *Immediately-Invoked Function Expression (IIFE).* [Online] 15. 11 2010. [Zitat vom: 23. 07 2019.] http://benalman.com/news/2010/11/immediately-invoked-function-expression/.
63. **Sysdig Inc.** Falco. *documentation.* [Online] 2019. [Zitat vom: 23. 07 2019.] https://falco.org/docs/installation/.
64. **kubesec.io.** KUBESEC.IO v2. [Online] [Zitat vom: 29. 07 2019.] https://kubesec.io/.
65. **Center for internet security.** CIS Benchmarks. *Securing Kubernetes.* [Online] 17. 07 2019. [Zitat vom: 29. 07 2019.] https://www.cisecurity.org/benchmark/kubernetes/.
66. **Haider, Hasham.** replex.io. *The Ultimate Kubernetes Cost Guide: AWS vs GCP vs Azure vs Digital Ocean.* [Online] 18. 09 2018. [Zitat vom: 06. 08 2019.] https://www.replex.io/blog/the-ultimate-kubernetes-cost-guide-aws-vs-gce-vs-azure-vs-digital-ocean.
67. **RedHat.** Store. [Online] [Zitat vom: 09. 07 2019.] https://www.redhat.com/en/store/red-hat-enterprise-linux-developer-suite#?sku=RH2262474.
68. **Google.** Google Cloud Platform Pricing Calculator. [Online] 08. 07 2019. [Zitat vom: 08. 07 2019.] https://cloud.google.com/products/calculator/#id=a3dcc642-6419-423a-8f55-167cfa17cebf.
69. **Dave, Apurva.** Sysdig Blog. *The 2017 Docker Usage Report.* [Online] 12. 04 2017. [Zitat vom: 06. 08 2019.] https://sysdig.com/blog/sysdig-docker-usage-report-2017/.
70. **Carter, Eric.** Sysdig Blog. *2018 Docker usage report.* [Online] 29. 05 2018. [Zitat vom: 06. 08 2019.] https://sysdig.com/blog/2018-docker-usage-report/.
71. **Edwards, Stefan, et al.** GitHub. *Kubernetes Final Report.* [Online] 31. 05 2019. [Zitat vom: 08. 08 2019.] https://github.com/kubernetes/community/blob/master/wg-security-audit/findings/Kubernetes%20Final%20Report.pdf.
72. **mccormd.** Github. *kubernetes.* [Online] 10. 08 2017. [Zitat vom: 18. 06 2019.] https://github.com/kubernetes/kubernetes/issues/50451.
73. **Romana project.** Kubernetes Blog. *High performance network policies in Kubernetes clusters.* [Online] 21. 09 2016. [Zitat vom: 18. 06 2019.] https://kubernetes.io/blog/2016/09/high-performance-network-policies-kubernetes/.
74. **Amazon.** AWS. *Elastic Load Balancing: Pricing.* [Online] [Zitat vom: 08. 07 2019.] https://aws.amazon.com/de/elasticloadbalancing/pricing/.
75. **Microsoft.** Azure. *Load Balancer: Pricing.* [Online] [Zitat vom: 08. 07 2019.] https://azure.microsoft.com/de-de/pricing/calculator/?service=load-balancer.
76. —. Microsoft Azure. *Preisrechner.* [Online] 08. 07 2019. [Zitat vom: 08. 07 2019.] https://azure.microsoft.com/de-

de/pricing/calculator/?&OCID=AID2000076_SEM_PNv6MpWy&lnkd=Google_Azure_Brand&dclid=CK63uuq0peMCFSIUiwodFNwNOg.
77. **VMWare.** vSphere. *Pricing.* [Online] 2019. [Zitat vom: 09. 07 2019.] https://www.vmware.com/de/reusable_content/vsphere_pricing.html.
78. **heise.** [Online] 06 2017. [Zitat vom: 09. 07 2019.] https://www.heise.de/newsticker/meldung/SSD-Langzeittest-beendet-Exitus-bei-9-1-Petabyte-3755009.html.
79. **klarsicht-it.** CPUs. [Online] [Zitat vom: 09. 07 2019.] https://www.klarsicht-it.de/pc-server/komponenten/cpu-intel/socket-lga3647/257481/intel-xeon-gold-6143-2.8-ghz-16-kerne-lga3647-socket-fuer-proliant-dl380-gen10.
80. —. RAM. [Online] [Zitat vom: 09. 07 2019.] https://www.klarsicht-it.de/pc-server/komponenten/arbeitsspeicher/ddr4-so-dimm/?p=1.
81. **Wikipedia.** Wikipedia. *Probabilistic Siganture Scheme.* [Online] 20. 05 2019. [Zitat vom: 05. 08 2019.] https://de.wikipedia.org/wiki/Probabilistic_Signature_Scheme.
82. **Langley, A, et al.** IETF. *RFC 7748: Elliptic Curves for Security.* [Online] 01 2016. [Zitat vom: 08. 05 2019.] https://tools.ietf.org/html/rfc7748.