



Fakultät Informatik

Softwaretechnik und Medieninformatik

Bachelorthesis

Evaluierung verschiedener Container Technologien

Corvin Schapöhler

751301

Semester 2018

Firma: NovaTec GmbH

Betreuer: Dipl.-Ing. Matthias Haeussler

Erstprüfer: Prof. Dr.-Ing. Dipl.-Inform. Kai Warendorf

Zweitprüfer: Prof. Dr. Dipl.-Inform. Dominik Schoop

Ehrenwörtliche Erklärung

Hiermit versichere ich, Corvin Schapöhler, dass ich die vorliegende Bachelorarbeit mit dem Titel Evaluierung verschiedener Container Technologien selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht in gleicher oder anderer Form als Prüfungsleistung vorgelegt worden.

Stuttgart, 27. März 2018

Ort, Datum

Corvin Schapöhler

Kurzfassung

Stichwörter: *Container, Docker, Cloud Native, OCI, Linux, rkt, Evaluation*

Abstract


Keywords: *Container, Docker, Cloud Native, OCI, Linux, rkt, Evaluation*


Inhaltsverzeichnis

Ehrenwörtliche Erklärung	I
Kurzfassung	II
Abstract	II
1 Einleitung	1
1.1 Motivation	1
1.2 Aufbau der Arbeit	1
2 Grundlagen	3
2.1 Standards	4
2.2 Funktionsweise	6
2.3 Eigene Implementierung	10
Abbildungsverzeichnis	A
Tabellenverzeichnis	B
Listings	C
Glossar	D
Akronyme	E
Literatur	F

1 Einleitung

1.1 Motivation

Die Welt wird immer stärker vernetzt. Durch den Drang, Anwendungen für viele Nutzer zugänglich zu machen besteht der Bedarf an Cloud-Diensten wie Amazon Web Services (AWS). Eine dabei immer wieder auftretende Schwierigkeit ist es, die Skalierbarkeit des Services  gewährleisten. Selbst wenn viele Nutzer gleichzeitig auf einen Service zugreifen, darf dieser nicht unter der Last zusammenbrechen.

Bis vor einigen Jahren wurde diese Skalierbarkeit durch Virtuelle Maschinen (VMs) gewährleistet. Doch neben großem Konfigurationsaufwand haben VMs auch einen großen Footprint und sind für viele Anwendungen zu ineffizient. Eine Lösung für dieses Problem stellen Container. 


Diese Arbeit beschäftigt sich mit dem Thema Containering und zeigt auf, wie diese den Entwicklungszyklus für Entwickler und DevOps erleichtern.



1.2 Aufbau der Arbeit

Zu Beginn dieser Arbeit werden die Grundlagen der Containertechnologie erklärt. Dabei wird darauf eingegangen, wie Container eine vollständige Isolation des Kernels schaffen. Um die technischen Grundlagen zu verstehen, wird eine eigene Abstraktion eines Container-Prozesses geschaffen.

Dabei wird ein Prozess auf einem Linux Hostsystem vollständig isoliert und die Kapselung dieses gezeigt. Es wird erkenntlich, dass die Isolation einzelner Prozesse auf Linux durch Kernelfeatures ermöglicht wird.

 Folgend werden Standards für Container-Technologien aufgezählt. Diese sind in den letzten Jahren durch den Boom der Technologie unerlässlich geworden. Dabei wird vor allem der Open Container Initiative (OCI) Standard näher beleuchtet, der sich durch die Vielzahl der kooperierenden Firmen durchsetzt.

Im Folgenden wird ein Blick auf die Geschichte der Technologie geworfen und anhand dieser erklärt, wie Docker die am weitesten verbreitete Technologie wurde. Zudem werden alternative Softwarelösungen zu Docker vorgestellt und miteinander verglichen. Dabei soll ein Fokus auf die Aspekte Konfiguration, Sicherheit und Orchestrierung der Container gelegt werden.

 Am Abschluss der Arbeit wird ein Blick in die Zukunft gewagt und Container im Zusammenhang mit Serverless Technologien gebracht.

2 Grundlagen

Container werden häufig als leichtgewichtige VMs beschrieben. Dies ist allerdings nicht ganz richtig. Wie in Abbildung 1 zu erkennen, virtualisieren Container kein vollständiges Operating System (OS), sondern lediglich das benötigte Dateisystem. Dabei wird der Kernel des Hosts nicht virtualisiert, sondern mitverwendet. Dies macht Container deutlich leichtgewichtiger als VMs, isoliert allerdings weniger umfangreich als diese. So werden bei Containern Kernel-funktionen mit dem Host-OS geteilt. VMs isolieren diese Funktionen in eigenen virtuellen Betriebssystemen.

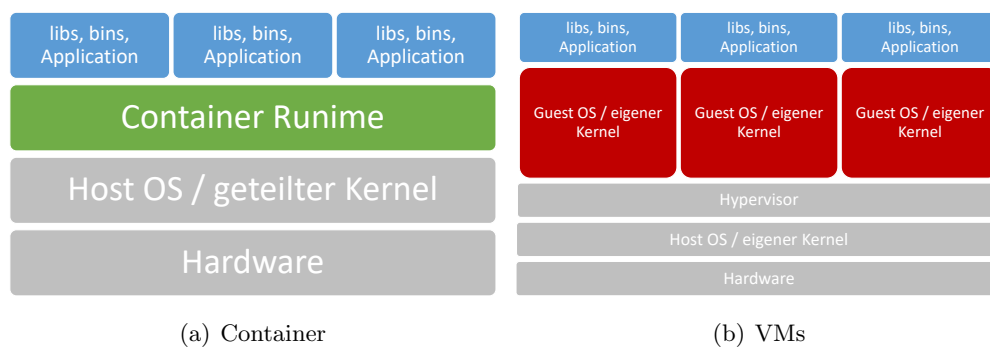



Abbildung 1: Container Isolation im Vergleich zu VMs

Dieses Kapitel behandelt alle benötigten Grundlagen, die zur Isolation eines Prozesses benötigt werden. Es werden vorhandene Standards wie die OCI und benötigte Systemcalls wie Change Root (chroot) näher erläutert. Zudem wird beschrieben, wie die Isolation, die Container bieten, durch Systemmittel des  LinuxKernels selber erreicht werden kann.

2.1 Standards

Durch die immer größere Verwendung von Containern und die Verbreitung verschiedener Container-Runtimes ist die Standardisierung eine wichtige Aufgabe. Folgend werden Standardisierungsprojekte aufgezählt, die bestehenden Spezifikationen erläutert und aktuelle Aufgaben der Projekte näher betrachtet.

2.1.1 App Container



App Container (appc) ist ein Standard, der viele Aspekte innerhalb der Container-Landschaft behandelt. Dabei liegt die Hauptaufgabe darin, eine Laufzeitumgebung wie auch das Imageformat und die Verbreitung von Images zu spezifizieren. Seit 2016 wird das Projekt nicht mehr aktiv weiterentwickelt. Bestandteile der appc wurden von der OCI übernommen und dienen als Vorlage für die Spezifikation dieser.

2.1.2 Open Container Initiative



Die OCI ist eine Initiative, die seit 2015 unter der Linux Foundation agiert. Das Ziel der OCI ist es, einen offenen Standard für Container zu schaffen, so dass die Wahl der Container-Laufzeitumgebung nicht mehr zu Inkompatibilität führt. Dabei liegt der Fokus auf eine einfache, schlanke Implementierung (Open Container Initiative, 2018).

Die OCI arbeitet aktuell an zwei Spezifikationen. Die runtime-spec standardisiert die Laufzeitumgebung von Containern. Dabei wird festgelegt, welche Konfiguration, Prinzipien und Schnittstellen Laufzeitumgebungen stellen müssen. Um die Umsetzung der runtime-spec zu fördern, stellt die OCI eine beispielhafte Implementierung durch runC.

Das zweite Projekt der OCI ist die image-spec. Dieses versucht einen Standard für Images zu definieren. Dabei plant die OCI nicht, vorhandene Image-Formate zu ersetzen, sondern auf diesen aufzubauen und sie zu erweitern (Open Container Initiative, 2018).

	Standard		Container Runtime	
	OCI	appc	Docker	rkt
Container Image	✗	✓	OCI image-spec	appc Image Format
Image Verbreitung	✗	✓	Docker Registry	appc Discovery Spec
Lokales Speicherformat	✓	✗	keine Spezifikation	keine Spezifikation
Runtime	✓	✓	runC	appc runtime Spec

Tabelle 1: Standards OCI und AppC im Vergleich (Polvi, 2015)

Wie in Tabelle 1 zu sehen, wurden einige Konzepte des appc-Projekts in die OCI übernommen. Vor allem die Image-Spezifikation wurde durch die Mitarbeit ehemaliger appc-Maintainer gefördert. Allerdings sind einige Projekte noch nicht übernommen worden. So gibt es keine OCI Spezifikation für die Verbreitung von Images, eines der meistgenutzten Features verschiedener Container-Runtimes. Um die Weiterentwicklung an solchen Projekten zu fördern wurden einige in die Cloud Native Computing Foundation (CNCF) übernommen (Polvi, 2015).

2.1.3 Cloud Native Computing Foundation

Die CNCF beschäftigt sich im Gegensatz zur OCI nicht nur mit Containern, sondern der kompletten Cloud-Native-Landschaft. Projekte wie Kubernetes (K8) und Prometheus werden durch die CNCF weiterentwickelt und publiziert. Da der Cloud-Native Entwicklungsprozess von Containern getragen wird, spielen Technologien wie containerd und rkt eine entscheidende Rolle für die CNCF.

Neben Container-Runtimes beinhaltet die CNCF auch Projekte zur Orchestrierung wie K8, Logging und Monitoring wie auch Spezifikationen, zum Beispiel die TUF, eine Spezifikation die standardisiert, wie Softwarepakete upgedatet werden sollen (CNCF, 2018).

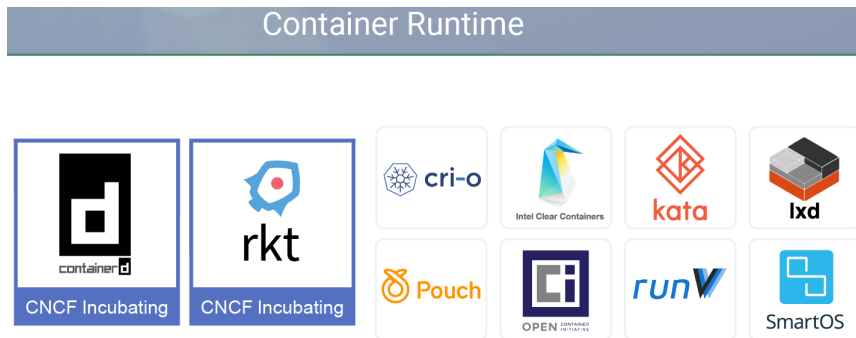


Abbildung 2: CNCF Container Runtime Landschaft (CNCF, 2018)

2.2 Funktionsweise

Container isolieren einzelne Prozesse durch verschiedene Kernel-Technologien, die im Folgenden erklärt werden sollen.

2.2.1 Change Root

Chroot ist ein Unix Systemaufruf, der es erlaubt einen Prozess in einem anderen Wurzelverzeichnis auszuführen (McGrath, 2017). Daraus folgt, dass der Prozess in einer eigenen Verzeichnisstruktur arbeitet und keine Dateien des Host-OS ändern kann. Chroot erlaubt somit die Isolierung des Dateisystems, die Container nutzen.

2.2.2 Control Groups

Control groups (cgroups) dienen dazu, Systemressourcen für einzelne Prozesse zu limitieren. Cgroups sind anders als chroot kein Unix-Feature sondern nur Teil des Linux-Kernels. Im OS sind cgroups als Dateihierarchie repräsentiert. Das gesamte cgroup-Dateisystem ist unter `/sys/fs/cgroup/` zu finden.

Cgroups stellen zur Steuerung verschiedene Controller zur Verfügung, die in Abschnitt 2.3 genauer betrachtet werden.

2.2.3 Namespaces

Namespaces abstrahieren einzelne Bereiche des OS. Sie werden genutzt um diese globalen Ressourcen zu isolieren. Ein Namespace kapselt dabei einzelne Ressourcen. Veränderungen an diesen sind für alle Prozesse innerhalb desselben Namespaces sichtbar, allerdings außerhalb dieses unsichtbar (Biederman, 2017).

Namespace	Ressource
Cgroup	Cgroup-Dateisystem
IPC	System V IPC, POSIX Nachrichten
Network	Netzwerk Geräte, Stacks, Ports, ...
Mount	Mount Punkte
PID	Prozess IDs
User	Nutzer und Gruppen IDs
UTS	Hostnamen und Domännennamen


Tabelle 2: Linux Namespaces und verbundene Ressourcen (Biederman, 2017)

2.2.4 Mounting

Durch die Isolation eines Prozesses und die Bedingung, das Container unveränderlich sein sollen stellt sich die Frage, wie man Containern Dateien aus dem Host-System zur Verfügung stellt. Dies ist vor allem wichtig, wenn bei Veränderung der Umgebung nicht den Container neu gestartet werden soll. Sollte zum Beispiel eine neue Datei durch einen Webserver zur Verfügung gestellt werden, möchte man nicht den Container neu starten. Die Lösung dieses Problems ist der Unix-Systembefehl `mount`.

Mit diesem Befehl wird eine beliebige Dateihierarchie an eine andere Stelle des Dateibaums angeheftet. Durch dieses vorgehen kann man Ordner vom Host-System für das mit `chroot` isolierte Dateisystem des Containers zugänglich machen.

Dabei ist zu beachten, dass es sich bei dem gemounteten Ordner nicht um einen symbolischen Link handelt. Dieser hätte durch den Aufruf von `chroot` keine

 Wirkung für den Container, da dieser keine Verzeichnisse über `/home/rootfs/` kennt und annimmt, dieses wäre das Wurzelverzeichnis.

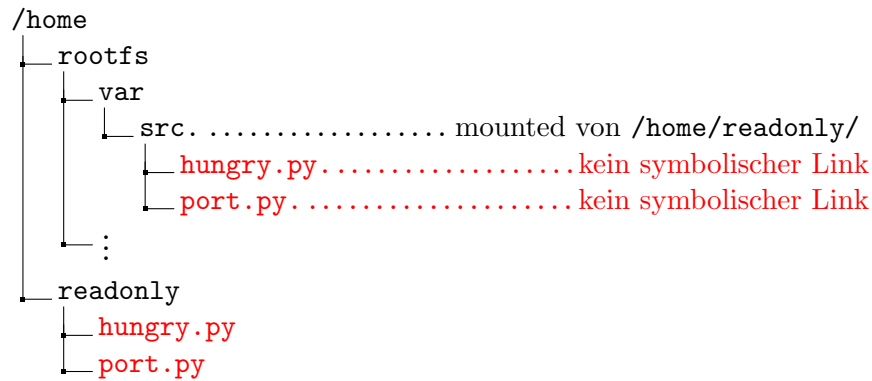


Abbildung 3: Auszug aus Dateisystem mit gemounteten Dateien

2.2.5 Netzwerk

Einen weiteren Aspekt, den Container vom Host-OS isolieren ist das Netzwerk. Dabei kommen virtuelle Ethernet-Adapter zum Einsatz. Diese erlauben es, ein unabhängiges Netzwerk zu erzeugen. Ein Ende des virtual Ethernet (VEth) wird dabei der PID des Containers zugewiesen, das andere dem Host. Zusätzlich wird der Network-Namespace genutzt um eine vollständige Isolation des Netzwerks zu erhalten.





Abbildung 4: Netzwerkseparation zwischen Host und Container

2.2.6 Sicherheit


"Docker is about running random code downloaded from the Internet and running it as root"

—Dan Walsh (Red Hat)

Container haben ein großes Problem. Alle genannten Kernel-Features müssen als Nutzer `root` ausgeführt werden. Dadurch hat der isolierte Prozess administrative Rechte und kann sich von der Isolation durch entsprechende Aufrufe befreien. Um dies zu verhindern, können verschiedene Sicherheitskonzepte der Linuxwelt verwendet werden. 

Das leichteste dieser Konzepte sind Capabilities. Jeder Benutzer  über auch jeder Datei kann eine Liste an Capabilities zugeordnet oder genommen werden. Dabei können einzelnen Dateien beispielsweise die Rechte genommen werden, auf Port 80 zu hören. Auch viele Systemaufrufe können über Capabilities gewährt oder verwehrt werden. Ein anderes Konzept ist die Implementation eines *Mandatory Access Control*-Systems wie SELinux oder AppArmor. Diese Implementationen sind granularer als Capabilities, allerdings mit einem höheren Konfigurationsaufwand verbunden.

2.2.7 Container unter Windows

Bislang wurden Container nur unter Linux verwendet. Viele Kernel-Features des Linux-Kernels erlauben  eine Isolation und wurden sogar teilweise spezifisch für diese entwickelt. Seit 2016 können spezifisch Docker-Container **aber** auch unter Windows genutzt werden. Dabei trennt Microsoft Container in zwei verschiedene Isolationen auf.

Hyper-V Container, die unter Windows 10 genutzt werden sind **nichts weiter als** extrem abgespeckte Hyper-V VMs, die nur noch nötigste Features einer VM behalten. Dabei wird **allerdings**, nicht wie in Abbildung 1 dargestellt auch der Kernel virtualisiert. Die zweite Alternative basiert auf Windows Server 2016 Container, die nur unter Windows Server 2016 und neueren Windows Server Betriebssystemen funktionieren. Diese implementieren eine wirkliche Isolation durch den Windows-Kernel.

2.3 Eigene Implementierung

Um die in Abschnitt 2.2 erläuterten Kernel-Features näher zu beleuchten wird folgend gezeigt, wie ein Prozess isoliert vom Host-System ausführen kann. Dabei wird darauf eingegangen, wie ein Tarball, durch das Tool buildroot erstellt werden kann, was am Beispiel eines Tarballs mit der Python-Runtime gezeigt wird. Dieser lässt sich folgend in Container-Runtimes wie Rocket (rkt) importieren. Im folgenden wird dieser Tarball mithilfe der Kernel-Funktionen aus Abschnitt 2.2 isoliert. Das Ergebnis ist ein Dateisystem innerhalb eines Host-OS, indem eine Instanz der Python-Runtime isoliert und ohne Root-Berechtigungen ausgeführt wird.

2.3.1 Erstellen eines Tarballs

Das Erzeugen eines Tarballs ist durch das Tool buildroot einfach. Buildroot ist ein Unix-Tool, welches zur Erstellung von minimalistischen Linux-Distributionen für Embedded-Systems entworfen wurde. Es erlaubt aber auch, nur ein Dateisystem zu erzeugen, ohne Kernel oder Init-System. Dieses Feature erlaubt es, ein Image für Container zu erzeugen.

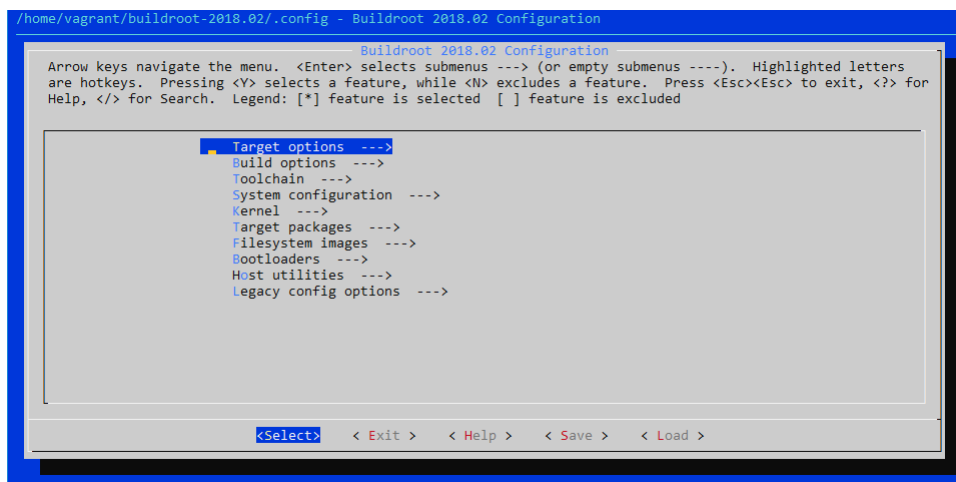



Abbildung 5: Buildroot Menü nach Start des Tools

Zudem erlaubt es buildroot, einzelne Bibliotheken, wie zum Beispiel die Python-Runtime, beim Build-Prozess der Distribution zu integrieren. Nach dem Einstellen der benötigten Bibliotheken und dem deaktivieren der, für Container, unnötigen Features erzeugt Buildroot eine Config-Datei, die all Änderungen beinhaltet. Durch das Starten des Build-Prozesses mit dem Befehl `make` wird die gewünschte Linux-Distribution erstellt. 

Am Ende dieses Prozesses liegt im Ordner `/buildroot/out/images/` das gewünschte Dateisystem `rootfs.tar`.

2.3.2 Isolieren der Python-Runtime

In Abschnitt 2.3.1 wurde ein Dateisystem mit der Python-Runtime erstellt. Dieses muss nun isoliert, ein Pythonprogramm in das Dateisystem gemountet und ausgeführt werden.

Der in [Abschnitt 2.3.1](#) erstellte Tarball wird durch folgende Command Line Interface (CLI)-Befehle entpackt.

```
cd /home
mkdir rootfs
cp rootfs.tar rootfs/
cd rootfs
sudo tar xvf rootfs.tar
sudo rm rootfs.tar
```

Listing 1: Entpacken des buildroot Tarballs nach `/home/rootfs/`

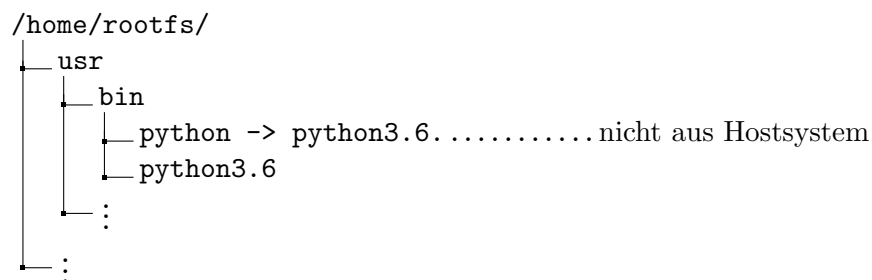


Abbildung 6: Dateibaum nach entpacken der `rootfs.tar`

Um einen Prozess mit dem Wurzelverzeichnis `/home/rootfs/` auszuführen, ist lediglich der folgende Aufruf nötig.

```
sudo chroot rootfs /usr/bin/python3.6 -m http.  
↪ server
```

Listing 2: Shell-Commands um Python Webserver mit definierter Root zu starten

Durch diesen wird ein Webserver auf Adresse `http://0.0.0.0:8000` ausgeführt, der alle ihm zugänglichen Dateien zum Download bereitstellt. Beim Aufrufen dieser Adresse erkennt man **schnell**, dass der Webserver nur Zugriff auf die in `/home/rootfs/` liegenden Dateien hat.

Directory listing for /

- [bin/](#)
- [dev/](#)
- [etc/](#)
- [lib/](#)
- [lib64/](#)
- [linuxrc/](#)
- [media/](#)
- [mnt/](#)
- [opt/](#)
- [proc/](#)
- [root/](#)
- [run/](#)
- [sbin/](#)
- [sys/](#)
- [tmp/](#)
- [usr/](#)
- [var/](#)

Abbildung 7: Python Webserver mit festgesetztem Root-Verzeichnis

Durch diese Dateitrennung tritt allerdings ein großes Problem aus. Der Python-Webserver wird mit erhöhten Rechten ausgeführt, da diese für den Aufruf von `chroot` benötigt werden.


```
vagrant@vagrant:/home$ sudo chroot rootfs/ /bin/sh
/ # whoami
root
/ #
```

Abbildung 8: Root-Eskalation durch Aufruf von `sudo chroot`

Dies führt zu vielen Problemen. Ein Prozess, der nur auf diese Weise isoliert wird, könnte beispielsweise Prozesse auf dem Hostsystem mit `kill <pid>` beenden. Die Lösung dieses Problems sind die in Abschnitt 2.2.3 beschriebenen namespaces.

Um alle Prozesse des Hostsystems vor dem Container zu verstecken, muss der PID-Namespace des Container-Prozesses neu gemounted werden.

```
sudo unshare -p --mount-proc=$PWD/rootfs/proc -f
↪ chroot rootfs /bin/sh
```

Listing 3: Remount des PID-Namespaces und Chroot einer Shell

Beim Aufruf von "`ps aux`" wird nur noch der Prozess `/bin/sh` angezeigt, der die PID 1 bekommen hat. Leider löst dieses Vorgehen nicht das Problem, dass noch immer die gestartete `/bin/sh` mit erhöhten Rechten ausgeführt wird. Ein Prozess, der auf diese Weise isoliert wird, kann zum Beispiel auf Port 80 hören. Um diese Berechtigungen zu entfernen werden die in Abschnitt 2.2.6 angesprochenen Capabilities verwendet.

Da die isolierte Shell aktuell als Nutzer Root läuft und somit alle Capabilities hat, werden einzelne Capabilities entfernt.

```
capsh --drop=cap_net_bind_service --chroot=
↪ rootfs/ --
```

Listing 4: Entfernen der Capability um auf Port 80 zu hören

Durch diesen Aufruf hat, die in `rootfs` gestartete `/bin/bash` nicht mehr die Möglichkeit, auf niedrigere Ports, wie Port 80 zu hören.

Um vollständige Isolation des Containers zu erreichen, müssen allerdings auch

Systemressourcen, wie Arbeitsspeicher oder CPU-Zyklen limitiert werden. Dazu dienen die in Abschnitt 2.2.2 beschriebenen cgroups.

Dabei sind vor allem die vorhandenen Controller von Bedeutung. Diese erlauben durch Einträge in Dateien eine präzise Steuerung einzelner Systemressourcen.


 Controller	Ressource
io	Zugriff und Nutzung von Block Geräten wie Festplatten
memory	Monitoring und Beschränken des Arbeitsspeichers
pids	Limitierung der Anzahl an Unterprozessen
perf_event	Erlaubt Performance Monitoring der Prozesse
rdma	Zugriffe über RDMA limitieren oder sperren
cpu	CPU-Zyklen und maximale CPU-Bandwidth

Tabelle 3: Cgroups-Controller und deren Verwendung (Kerrisk, 2018)

Um eine cgroup zu erstellen, muss **nur** ein Ordner unterhalb des Wurzelverzeichnisses erstellt werden. Um einen Prozess einer cgroup zuzuordnen, wird die PID des Prozesses in die Datei `/sys/fs/cgroup/<controller>/<cgroupname>/tasks` geschrieben.

```
mkdir /sys/fs/cgroup/memory/container
echo "1111" > /sys/fs/cgroup/memory/container/
↪ tasks
```

Listing 5: Erzeugen einer memory cgroup namens container

Um festzusetzen, wie viel Arbeitsspeicher der isolierte Prozess nutzen darf, kann man innerhalb der cgroup einzelne Limits festlegen.

```
echo "0" > /sys/fs/cgroup/memory/container/
↪ memory.swappiness
echo "100000000" > /sys/fs/cgroup/memory/
↪ container/memory.limit_in_bytes
```

Listing 6: Limitieren maximaler Arbeitsspeicher und Swap deaktivieren

Um zu testen, ob die Zuweisung funktioniert und durch den isolierten Prozess

maximal 100Mb Arbeitsspeicher belegt werden können, kann folgendes Python Programm ausgeführt werden. Abbildung 9 zeigt die Ausgabe des Prozesses.

```
import time
TEN_MEGABYTE = 100000000

f = open(" /dev/urandom" , "rb")
data = bytearray()
i = 0

while True:
    data.extend(f.read(TEN_MEGABYTE))
    i += 1
    print("%dMb belegt" % (i*10,))
    time.sleep(1)
```

Listing 7: Python Programm um Arbeitsspeicher zu verbrauchen

```
/ # python /var/src/hungry.py
10Mb belegt
20Mb belegt
30Mb belegt
40Mb belegt
50Mb belegt
60Mb belegt
70Mb belegt
80Mb belegt
Killed
```

Abbildung 9: Ausgabe des Pythonprogramms

Abbildungsverzeichnis

1	Container Isolation im Vergleich zu VMs	3
2	CNCF Container Runtime Landschaft (CNCF, 2018)	6
3	Auszug aus Dateisystem mit gemounteten Dateien	8
4	Netzwerkseparation zwischen Host und Container	8
5	Buildroot Menü nach Start des Tools	10
6	Dateibaum nach entpacken der rootfs.tar	11
7	Python Webserver mit festgesetztem Root-Verzeichnis	12
8	Root-Eskalation durch Aufruf von sudo chroot	13
9	Ausgabe des Pythonprogramms	15

Tabellenverzeichnis

1	Standards OCI und AppC im Vergleich (Polvi, 2015)	5
2	Linux Namespaces und verbundene Ressourcen (Biederman, 2017)	7
3	Cgroups-Controller und deren Verwendung (Kerrisk, 2018) . . .	14

Listings

1	Entpacken des buildroot Tarballs nach /home/rootfs/	11
2	Shell-Commands um Python Webserver mit definierter Root zu starten	12
3	Remount des PID-Namespaces und Chroot einer Shell	13
4	Entfernen der Capabilty um auf Port 80 zu hören	13
5	Erzeugen einer memory cgroup namens container	14
6	Limitieren maximaler Arbeitsspeicher und Swap deaktivieren . .	14
7	Python Programm um Arbeitsspeicher zu verbrauchen	15

Glossar

Cloud-Native Cloud-Native Anwendungen sind spezifisch für Cloud-Architekturen entwickelt. Sie spalten meistens große Funktionalitäten in kleine Micro-services, die mittels API miteinander kommunizieren und sind somit skalierbar, loose gekoppelt und ausfallsicherer als Full-Client Anwendungen.

DevOps DevOps (von *Development* und *Operations*) dienen der einfacheren Auslieferung von Software an Entwickler wie an den Kunden. Dabei verwenden sie Continuous Integration (CI) Tools wie Jenkins um eine automatisierte Bereitstellung zu gewährleisten.

Image Ein Container Image ist eine Datei, die spezifiziert, wie ein Container von der Laufzeitumgebung ausgeführt werden soll.

Kernel Der Kernel ist der Kern eines Betriebssystems. In ihm werden die elementaren Bestandteile des Betriebssystems implementiert.

Tarball Ein Tarball ist ein Dateisystem, das mit dem Tool tar komprimiert wurde. In den meisten Fällen befindet sich ein komplettes Root-Filesystem in diesem.

Akronyme

appc App Container.

AWS Amazon Web Services.

cgroup control group.

chroot Change Root.

CI Continous Integration.

CLI Command Line Interface.

CNCF Cloud Native Computing Foundation.

K8 Kubernetes.

OCI Open Container Initiative.

OS Operating System.

rkt Rocket.

VEth virtual Ethernet.

VM Virtuelle Machine.

Literatur

- BIEDERMAN, Michael Kerrisk; Eric W., 2017. *namespaces(7) - Linux Manual Page*.
- CHIANG, Eric, 2017. *Containers from Scratch*. Auch verfügbar unter: <https://ericchiang.github.io/post/containers-from-scratch/>.
- CNCF, 2017. *Cloud Native Computing Foundation*. Auch verfügbar unter: <https://www.cncf.io/>.
- CNCF, 2018. *CNCF Cloud Native Interactive Landscape*. Auch verfügbar unter: <https://landscape.cncf.io/>.
- CREQUY, Simone Gotti; Luca Bruno; Iago López Galeiras; Derek Gonyeo; Alban, 2018. *App Container*. Auch verfügbar unter: <https://github.com/appc>.
- DOCKER INC, 2018. *Docker security*. Auch verfügbar unter: <https://docs.docker.com/engine/security/security/>.
- HARRINGTON, Brian "Readbeard", 2015. *Building minimal Containers: Getting Weird with Containers*. Auch verfügbar unter: https://github.com/brianredbeard/minimal_containers.
- HEO, Tejun, 2015. *Control Group v2*.
- KERRISK, Serge Hallyn; Michael, 2018. *cgroups(7) - Linux Manual Page*.
- MAS, Raphaël Hertzog; Roland, 2015. *The Debian Administrator's Handbook, Debian Jessie from Discovery to Mastery*. Freexian. ISBN 9791091414043.
- MATTHIAS, Karl; KANE, Sean P., 2015. *Docker: Up & Running: Shipping Reliable Containers in Production*. O'Reilly Media. ISBN 978-1-491-91757-2.
- MCGRATH, Roland, 2017. *chroot(1) - Linux Manual Page*.
- OPEN CONTAINER INITIATIVE, 2018. *Open Container Initiative*. Auch verfügbar unter: <https://www.opencontainers.org/>.

- PETAZZONI, Jérôme, 2013. *Create Lightweight Containers with Buildroot*. Auch verfügbar unter: <https://blog.docker.com/2013/06/create-light-weight-docker-containers-buildroot/>.
- POLVI, Alex, 2015. *Making Sense of Container Standards and Foundations: OCI, CNCF, appc and rkt*. Auch verfügbar unter: <https://coreos.com/blog/making-sense-of-standards.html>.
- ROUMELIOTIS, Rachel, 2016. The Open Container Essentials Video Collection. In: *The Open Container Essentials Video Collection*. O'Reilly Open-Source Conference. ISBN 9781491968253.
- ZAK, Karel, 2015. *mount(8) - Linux Manual Page*.

Todo list

Figure: Netzwerk Graph für Container und Host?	8
--	---