

HOCHSCHULE DER MEDIEN

BACHELORTHESES

**Sicherheitsbetrachtungen von
Applikations-Containersystemen in
Cloud-Infrastrukturen am Beispiel
Docker**

Moritz Hoffmann

Studiengang: Mobile Medien

Matrikelnummer: 26135

E-Mail: mh203@hdm-stuttgart.de

26. Februar 2016

Erstbetreuer:

Prof. Dr. Joachim Charzinski

Hochschule der Medien

Zweitbetreuer:

Patrick Fröger

ITI/GN, Daimler AG

Eidesstattliche Erklärung

„Hiermit versichere ich, Moritz Hoffmann, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Sicherheitsbetrachtungen von Applikations-Containersystemen in Cloud-Infrastrukturen am Beispiel Docker“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

Unterschrift

Datum

Abstract

English version:

....
..

Deutsche Version:

....
...
.
..

Inhaltsverzeichnis

1	Überblick	1
1.1	Ziel der Arbeit	3
1.2	Struktur der Arbeit	3
2	Grundlagen	5
2.1	Virtualisierung	5
2.1.1	Hypervisor-basierte Virtualisierung	6
2.1.2	Container-basierte Virtualisierung	9
2.1.3	Einordnung Docker	11
2.2	Sicherheitsziele in der IT	12
2.2.1	Vertraulichkeit	12
2.2.2	Integrität	13
2.2.3	Verfügbarkeit	13
2.2.4	Datenschutz	13
2.3	Einführung in Docker	13
2.3.1	Docker Architektur	15
2.3.2	Dockerfile	16
2.3.3	Containerformate LXC, libcontainer, runC und OCF	18
2.3.4	Images	19
2.3.5	Container	22
2.3.6	Registries	22
3	Fragestellungen / Problemformulierung	24
4	Sicherheit durch Linux-Funktionen	29

4.1	Isolierung durch namespaces	29
4.1.1	Prozessisolierung durch den PID namespace	30
4.1.2	Dateisystemisolierung durch den mount namespace	32
4.1.3	Geräteisolierung durch	33
4.1.4	IPC-Isolierung durch den IPC-namespace	33
4.1.5	UTS-Isolierung durch den UTS-namespace	34
4.1.6	Netzwerkisolierung durch den network namespace	34
4.1.7	Userisolierung (user namespace)	35
4.2	Ressourcenverwaltung / Limitierung von Ressourcen durch cgroups	37
4.3	Einschränkungen von Zugriffsrechten	39
4.3.1	capabilities	41
4.3.2	Mandatory Access Control (MAC) und Linux Security Modules (LSMs)	42
4.3.2.1	AppArmor	44
4.3.2.2	SELinux	48
	Type Enforcement (TE)	51
	Multi-Category Security (MCS)	51
	Multi-Level Security (MLS)	52
4.3.3	Seccomp	52
4.4	Docker im Vergleich zu anderen Containerlösungen	54
5	Security im Docker-Ökosystem	55
5.1	Private Registries	56
5.2	Verifikation und Verteilung von Images	57
5.2.1	Verifikation von Images	57
5.2.2	Integration von <i>The Update Framework</i>	58
5.3	Verbindung zwischen Daemon und Clients	59
5.4	Docker Plugins	60
5.5	Security Policies und Open Source	63
5.6	Security Best-Practises	63
5.6.1	Datencontainer	63
5.6.2	--privileged Container	63

5.7	Tools	63
5.7.1	Docker Swarm und Docker Compose	63
5.7.2	Kubernetes	63
5.7.3	Vagrant, Puppet, Chef	64
5.7.4	Nautilus Project ?	64
5.7.5	github.com/cloudimmunity/docker-slim	64
5.7.6	Vagrant	64
5.8	Networking	64
5.8.1	DNS	64
6	Docker in Unternehmen/Cloud-Infrastrukturen	65
7	Fazit	66

Abbildungsverzeichnis

1	Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[42].	2
2	Die Client-Server-Architektur von Docker [21].	16
3	Aufbau eines Docker-Hosts, wenn dieser unter einem Linux-Betriebssystem betrieben wird, das direkt auf der Serverhardware läuft. [105, S.3].	17
4	Dateien im Ordner eines Images (eigene Abbildung).	19
5	Visualisierung eines Vergleichs von Images von <i>Redis</i> , <i>Nginx</i> und <i>CentOS</i> auf Schichtebene [51].	21
6	Screenshot von der Ausführung des Befehls <code>docker pull <image></code> (eigene Abbildung).	21
7	Screenshot von der Ausführung des Befehls <code>docker images</code> (eigene Abbildung).	21
8	Web-UI des Docker Hubs mit den beliebtesten Repositories [28].	23
9	Ausschnitt der Ausgabe des Befehls <code>ps -eafxZ</code> auf einem Docker-Host (eigene Abbildung).	31
10	Ausgabe des Befehls <code>ps -eafxZ</code> in einem Docker-Container (eigene Abbildung).	31
11	Funktionsweise von <i>System Call</i> -Hooks eines LSMs [135, S.3].	44
12	Trennung von Regelwerk und Enforcement-Modul. Zuweisung von Security-Contexts (SC) an Objekte und Subjekte [108, S.63].	50

13	Ablaufdiagramm einer Befehlsausführung mit dem Authorisierungs- Plugins <i>AuthZ</i> von Docker	62
----	--	----

Tabellenverzeichnis

2	Mehrere Herausforderungen im Betrieb von IT-Infrastrukturen und deren Lösungsansatz mithilfe von Virtualisierung	7
---	---	---

Kapitel 1

Überblick

Virtualisierung entwickelte sich in den letzten Jahren zu einem allgegenwärtigen Thema in der Informatik. Mehrere Virtualisierungstypen entstanden, als von akademische und industrielle Forschungsgruppen vielseitige Einsatzmöglichkeiten der Virtualisierung aufgedeckt wurden.

Allgemein versteht man unter ihr die Nachahmung und Abstraktion von physischen Ressourcen, z.B. der CPU oder des Speichers, die in einem virtuellen Kontext von Softwareprogrammen genutzt wird.

Die Vorteile von Virtualisierung umfassen Hardwareunabhängigkeit, Verfügbarkeit, Isolierung und Sicherheit, welche die Erfolgsgrundlage der Virtualisierung in heutigen Cloud-Infrastrukturen bilden [136, S.1]. Vor allem in Rechenzentren bieten sich Virtualisierungen an, um die Serverressourcen effizienter zu nutzen [103, S.1]. Letztendlich haben es Virtualisierungen ermöglicht, Serverressourcen in der Form von Clouds, wie z.B. den *Amazon Web Services*[5], und auf Basis eines Subskriptionsmodells nutzen zu können [103, S.1].

Heutzutage existieren mehrere serverseitige Virtualisierungstechniken, wovon die Hypervisor-gestützten Methoden mit den etablierten Vertretern *Xen*[45], *KVM*[43], *VMware ESXi*[44] und *Hyper-V*[91] die meistverbreitesten sind [136, S.2]. Die alternative containerbasierte Virtualisierung, auch Virtualisierung auf Betriebssystemebene (*Operating System-Level Virtualization*) ge-

nannt, wurde in den letzten Jahren durch ihre leichtgewichtige Natur zunehmend beliebt und erlebte mit dem Erfolg von Docker, seit dessen Release im März 2013, einen medienwirksamen Aufschwung [36]. Wie die *Google Trends* in Abb.1 zeigen, stieg das Interesse an Docker seit dessen Release kontinuierlich an, während das Suchwort „virtualization“ im Jahr 2010 seinen Höhepunkt hatte und seitdem an Popularität verlor. Auch das Interesse an der Containerertechnologie *LXC*, aus der Docker entstand, bleibt weit hinter der von Docker zurück [42].

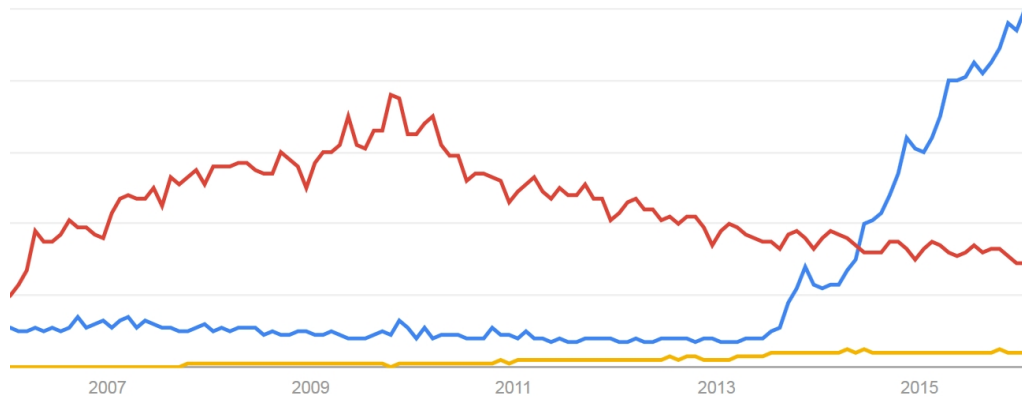


Abbildung 1: Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[42].

Obwohl das Konzept von Containern bereits im Jahr 2000 als *Jails* in dem Betriebssystem *FreeBSD* und seit 2004 als *Zones* unter *Solaris* verwendet wurde [73][72], gelang keiner dieser Technologien vor Docker der Durchbruch. Wie Docker den bis 2013 vorherrschenden Ruf von Containerertechnologien, dass Container noch nicht ausgereift seien [136, S.8], nachhaltig verändern konnte, ist in der Einführung zu Docker in Kapitel 2.3 beschreiben.

Heute sind Container in vielen Szenarien, v.a. skalierbaren Infrastrukturen, trotz intrinsischer Sicherheitsdefizite gegenüber Hypervisor-gestützten Virtualisierungsarten beliebt. Vor allem Multi-Tenant-Services werden gerne mit Docker umgesetzt [132, S.6][21].

1.1 Ziel der Arbeit

Ziel der Arbeit ist es, die von Containersystemen genutzten Sicherheitsmechanismen vorzustellen und zu untersuchen, inwiefern diese zu im Fall von Docker zu einer höheren Sicherheit des Systems beitragen.

Eine ausführliche Konstruktion der Fragestellung erfolgt mithilfe einiger Annahmen in Kapitel 3.

1.2 Struktur der Arbeit

Zu Beginn wird in den Grundlagen ab Kapitel 2.1 die Virtualisierung beschrieben. Dabei werden die zwei prominentesten Virtualisierungstechniken, Hypervisor-basierte (Sektion 2.1.1) und Container-basierte (Sektion 2.1.2) Virtualisierung, gegenübergestellt. In diesem Kapitel werden nur die für diese Arbeit relevante Techniken der Systemvirtualisierung beschrieben, also solche, in denen Funktionen von kompletten Betriebssystemen abstrahiert werden. Andere Arten, beispielsweise die Anwendungs-, Storage- oder Netzwerkvirtualisierung, werden nicht behandelt, da sie isoliert keinen Bezug zu Docker haben. Anschließend werden die allgemeinen Sicherheitsziele von IT-Systemen in Kapitel 2.2 erklärt, auf die im Hauptteil Bezug genommen wird. Abgeschlossen wird das Grundlagenkapitel mit einer Einführung in Docker (Kapitel 2.3), in dem Begriffe sowie Funktionsweisen innerhalb dieser Technologie erläutert werden.

Die genannten Grundlagen sind sehr weitreichende Themengebiete. Um in den einleitenden Kapiteln nicht ausführlich zu werden, sind Eckdaten einiger am Rande auftretender Begriffe im angehängten Glossar zusammengefasst.

Der Hauptteil ab Kapitel 4 untergliedert sich in mehrere Sicherheitsgebiete, in die die Arbeit eingeteilt ist:

1. **Sicherheitsmechanismen, die Linux ermöglicht** und teils obliga-

torisch von Docker eingesetzt werden. Darunter fallen Techniken zur Isolierung, Ressourcen- und Rechteverwaltung von Containern sowie Methoden, um das Hostsystem mit zusätzlichen Sicherheitsfeatures von Linux abzusichern.

2. **Sicherheit im Docker-Ökosystem.** Darunter fallen z.B.

- Integrität von Images
- Absicherung der Kommunikation zwischen dem Docker-Client und dem Docker-Host
- Best-Practices im Umgang mit Docker-Komponenten sowie Sicherheitsrichtlinien.
- Verwendung von Third-Party Tools, wie *Kubernetes*

3. **Sicherheit von Docker in Cloud-Infrastrukturen.** Gegenstand der Untersuchung ist, ob und wie Docker in der Cloud eingesetzt werden kann, sodass Sicherheitsanforderungen von Unternehmen erfüllt werden.

Abgeschlossen wird die Arbeit im letzten Kapitel 7 mit einer Zusammenfassung der Erkenntnisse und einem Ausblick auf die Zukunft von Docker und der containerbasierten Virtualisierung.

In der Arbeit vorkommende Produkt-, Technologie-, Bibliotheken- und Unternehmensnamen sind durchgehend *kursiv* gedruckt. Eine Ausnahme bildet Docker, in der die reguläre Schreibweise für die Plattform Docker vorgesehen ist, während die kursive Variante das Unternehmen *Docker* meint.

Im Gegensatz dazu sind technische Identifikationsmerkmale, Befehle und Variablennamen **mono-type** geschrieben. Platzhalter für aufgeführte Befehlsparameter sind in Großbuchstaben abgedruckt. Ein Befehl `cmd` beispielsweise, der einen Parameter erwartet, ist dementsprechend als `cmd PARAMETER` generisch formuliert.

Kapitel 2

Grundlagen

2.1 Virtualisierung

Bei der Virtualisierung, deren Anfänge sich auf IBM-Forschungsprojekte von 1964 zurückführen lassen, werden ein oder mehrere virtuelle Systeme auf einem physischen System betrieben. Mehrere solcher Systeme können eine virtuelle Infrastruktur bilden, in der physische und virtuelle Systeme gemeinsam verwaltet werden können [131, S.661].

Das Betriebssystem, das in der Rolle eines Wirts mehrere virtuelle Systeme betreibt, wird als Host oder Hostsystem bezeichnet. Systeme, die einzeln oder parallel virtualisiert auf einem solchen Host laufen, werden als Gastsysteme bezeichnet.

Virtualisierte Systeme nutzen im Vergleich zu nativen (physischen) Systemen eine zusätzliche Softwareschicht, die den virtualisierten Systemen in der Ausprägung von virtuellen Maschinen, sogenannte VMs, und Containern, mehrere Abstraktionen anbietet, um Funktionen des Hostsystems abzubilden [136, S.2]. Beide Ausprägungen erwecken aus Sicht des Gastsystems den Eindruck, dass ein alleinstehendes Betriebssystem ausgeführt wird. Der Einsatz von Virtualisierung bietet vielfältige Vorteile für IT-Unternehmen. In der folgenden Auflistung sind einige Problemfaktoren und Anforderungen

an Rechenzentren und darin laufender Software zusammengefasst, die alle durch Virtualisierungslösungen adressiert werden können [111, S.1][131, S.662,672f.][117, S.299].

Wie zu sehen ist, bietet der Einsatz von serverseitiger Virtualisierung eine Reihe von Vorteilen, die ein rein physischer Betrieb von Servern nicht in dem Maß bieten kann. Diese Vorteile eröffneten IT-Unternehmen auch neue Geschäftsmodelle, auf denen der Erfolg von heutigen Cloud-Anbietern wie *Amazon*, *Google* und *Microsoft* beruht. Selbst einhergehende Nachteile der Virtualisierung, wie z.B. Leistungsverluste und in dieser Arbeit untersuchte Sicherheitsrisiken, scheinen, den Vorteilen gegenübergestellt, nicht ins Gewicht zu fallen.

Im Folgenden sind die Konzepte der Virtualisierung auf Hypervisor- und Containerbasis erklärt und in ihren ausschlaggebenden Eigenschaften gegenübergestellt.

2.1.1 Hypervisor-basierte Virtualisierung

Im Kontext einer Hypervisor-basierten Virtualisierung wird das virtuelle System eine VM genannt. VMs enthalten jeweils eine Umgebung, die Abstraktionen eines sogenannten Hypervisors nutzt, um Hardwareressourcen des Hosts zu verwenden. Der Hypervisor, auch seltener *Virtual Machine Monitor* (*VMM*) genannt, ist ein Stück Software, das zwischen einem Host- und einem Gast-Betriebssystem (der VM) vermittelt und Hardwareabstraktionen des ersteren bereitstellt [132, S.6][136, S.2][103, S.2]. Eine weitere wichtige Eigenschaft eines Hypervisors ist es, dem Gast jede für den Betrieb eines Betriebssystems nötige Funktion anzubieten [131, S.106]

Durch diese Technik läuft in jeder VM ein eigenes (Gast-)Betriebssystem, das von solchen anderer VMs isoliert läuft. Durch die Abstraktion des zwischenliegenden Hypervisors ist es möglich, mehrere unterschiedliche Gastbetriebssysteme auf einem physikalischen Host auszuführen [136, S.2][131, S.106].

Herausforderung	Lösungsansatz mithilfe von Virtualisierung
Rechenzentren sollen möglichst effizient betrieben werden, da anfallende Energiekosten für den Betrieb und die Kühlung von Servern auch bei geringer Auslastung anfallen.	Kosteneinsparungen bei der Anschaffung von Hardware und Lizenzen sowie dem Energieverbrauch möglich, da virtuelle Instanzen die physischen Ressourcen effizienter ausnutzen können und demnach in Summe auch weniger physische Maschinen benötigt werden.
Softwarelösungen sollen skalierbar sein. Je nach Bedarf sollen Kapazitäten in der IT-Infrastruktur freigeschaltet oder reduziert werden können.	Die Virtualisierung ermöglicht es, hardwareunabhängige, portierbare und reproduzierbare Softwarekomponenten zu realisieren, die flexibel aktiviert und deaktiviert werden können.
In einem Rechenzentrum wird Software unterschiedlicher Kunden ausgeführt (Multi-Tenancy-Umgebung). Eine Trennung dieser Kundeninstanzen muss gewährleistet sein, sodass diese nicht miteinander interferieren können.	Virtuelle Lösungen bieten verschiedene native Möglichkeiten zur Isolierung von Systemen. Eine Gefahr von Interferenz zwischen Kunden kann je nach eingesetzter Technologie auf unterschiedliche Art und Weise reduziert werden.
Redundanz, um Ausfällen entgegenzuwirken, soll gewährleistet sein.	VMs und Container sind flexibel zu starten und stoppen. Ausfallsicherheit wird in Form mehrerer redundanter virtueller Instanzen und der Unterstützung von Tools ermöglicht.
Bestimmte Architekturen, wie z.B. 3-Tier-Architektur, sollen komfortabel umgesetzt werden.	Einzelne Bausteine beliebiger Architekturen lassen sich mit virtuellen Instanzen unabhängig voneinander betreiben und vernetzen.
Über die Zeit hinweg entstehen unterschiedliche Softwareversionen, die ggf. von teils alten Anwendungen und Bibliotheken abhängen. Die verschiedenen Versionen sollen zuverlässig und unabhängig voneinander betrieben werden. Nicht nur Anwendungen, sondern auch Betriebssysteme unterschiedlicher Art sollen ggf. möglichst hardwareunabhängig einsetzbar sein.	Virtualisierungen bieten gute Migrationseigenschaften und sind mit einer Vielzahl von Betriebssystemen kompatibel. Vor allem die containerbasierte Virtualisierung erlaubt einen hohen Grad an Modularisierung und Kompatibilität. Produkte werden können nicht nur als Programme vertrieben, sondern als plattformunabhängige <i>Virtual Appliances</i> , also in virtuelle Instanzen eingebettete Anwendungen [131, S.672f.].
Neue Services und Kunden sollen	Virtuelle Instanzen können als so-

Trotz der in Tabelle ?? aufgeführten effizienteren Ressourcennutzung im Rahmen von Virtualisierungstechniken, stehen Hypervisor heute unter dem Ruf ineffizient zu arbeiten. Dieser größte Kritikpunkt der genannten Virtualisierungsmethode, lässt sich auf den Erfolg der containerbasierten Virtualisierung zurückführen, die, wie in Kapitel 2.1.2 weiter ausgeführt, weniger Overhead erzeugen.

Hypervisor-Technologien werden in solche von Typ 1 und Typ 2 unterschieden:

Der Typ 1 Hypervisor operiert direkt auf der Hardware des Hosts und stellt ein minimales, speziell für den Betrieb von VMs ausgelegtes Betriebssystem dar. Dessen Aufgabe ist es, Kopien der realen Hardware bereitzustellen, die von Gastsystemen, den VMS, genutzt werden können. Wenn in der VM ein Befehl ausgeführt wird, wird dieser an den Hypervisor weitergeleitet, der den Befehl untersucht. Handelt es sich um einen Befehl des Gastbetriebssystems, wird dieser auf dem Host ausgeführt. Im Fall eines Aufrufs einer Anwendung innerhalb des Gasts, emuliert der Hypervisor für diesen Aufruf die Aktion der realen Hardware [131, S.663ff.].

Hypervisor des Typs 2 arbeiten nicht direkt auf der Hardware, sondern einem Host-Betriebssystem, das wiederum selbst direkt auf die Hardware zugreift. In dieser Variante ist der Hypervisor eine gewöhnliche Anwendung, die auf dem Hostbetriebssystem ausgeführt wird. Die Aufrufe einer in dieser Anwendung installierten Anwendung, werden mithilfe einer sogenannten Binärübersetzung in Hypervisor-Prozeduren übersetzt, sofern der initiierte Befehl einen Hardwarezugriff verlangt. Hypervisor-Prozeduren dienen auch hier wieder zur Hardwareemulation, die auf dem Hostbetriebssystem ausgeführt werden. Erfordern Teile der Gastanwendung keinen Hardwarezugriff, werden diese im Gast selbst verarbeitet und verlassen diesen nicht.

Unter beiden Typen wird jedoch eine vollständige Abstraktion von Hostressourcen erzielt. VMs werden in beiden Fällen wie reale Systeme gestartet und unterliegen der Illusion als alleiniges, natives System zu operieren [131, S.665f.]. Es existieren dennoch Anhaltspunkte für Gastsysteme und dessen

Anwendungen, um zu bestimmen, ob sie sich in einer physischen oder virtuellen Umgebung befinden.

Im Durchschnitt führt die zusätzliche Softwareschicht des Typs 2 trotz verschiedener Optimierungsmöglichkeiten, z.B. der sogenannten Paravirtualisierung, zu höheren Performanceeinbußen wie jenen unter Typ 1 [131, S.666f.][103, S.2].

Bekannte Hypervisor-basierte Technologien sind die kommerziellen Vertreter *ESXi* der Firma *VMware* und *Hyper-V* von *Mircosoft*, sowie die Open-Source-Vertreter *Xen* und *KVM* [101, S.1].

2.1.2 Container-basierte Virtualisierung

Container-basierte Virtualisierung wird vorrangig als leichtgewichtige Alternative zu der Hypervisor-basierten Virtualisierung gesehen[136, S.2]. Erstere nutzt direkt den Hostkernel, um virtuelle Umgebungen zu schaffen. Ein Hypervisor wird in diesem Ansatz nicht benötigt [132, S.6+7]. Vielmehr wird das native System und dessen Ressourcen partitioniert, sodass mehrere virtuelle, voneinander isolierte Instanzen, sogenannte *user space* Instanzen, betrieben werden können, die als Container bezeichnet werden [136, S.2][105, S.3][124, S.1]. Die Isolation basiert auf dem Konzept von Kontexten, die unter Linux *Namespaces* genannt werden. Diese, sowie *Control Groups*, die für das Ressourcenmanagement verantwortlich sind, werden in den Kapiteln 4.1 und 4.2 genauer betrachtet [105, S.4].

Container sind durch den Unix-Befehl *chroot*[54] inspriert, der schon seit 1979 im Linux-Kernel integriert ist. In *FreeBSD* wurde eine erweiterte Variante von *chroot* verwendet, um sogenannte *Jails* (FreeBSD-spezifischer Begriff) umzusetzen [31]. In *Solaris*, ein von der Firma *Oracle* entwickeltes Betriebssystem für Servervirtualisierungen[46], wurde dieser Mechanismus in Form von *Zones* (Solaris-spezifischer Begriff) [92] weiter verbessert und es etablierte sich der Name *Container* als Überbegriff, als weitere proprietäre Lösungen von *HP* und *IBM* zur selben Zeit auf dem Markt erschienen [101,

S.2]. Durch die kontinuierliche Weiterentwicklung von Containern in den letzten Jahren, können diese heutzutage laut [132, S.7] als vollwertige Systeme betrachtet werden, nicht mehr als - wie ursprünglich vorgesehen - reine Ausführungsumgebungen.

Während ein Hypervisor für jede VM das komplette Gast-OS abstrahiert, werden für Container direkt Funktionen des Hosts über *System Calls* zur Verfügung gestellt. Im Betrieb von Containern kommunizieren diese direkt mit dem Host und teilen sich den Kernel dessen. Deswegen werden Containerlösungen auch als Virtualisierungen auf Betriebssystemebene (des Hosts) bezeichnet [132, S.6+7][136, S.2][101, S.3].

Dieses Design hat einen entscheidenden Nachteil gegenüber einem Hypervisormodell, der auch Docker betrifft: Das Container-Betriebssystem muss wie das Host-Betriebssystem linuxbasiert sein. In einem Host auf dem *Ubuntu Server* installiert ist, können nur weitere Linux-Distributionen als Container laufen. Ein *Microsoft Windows* kann also nicht als Container auf genannten Host gestartet werden, da die Kernel miteinander nicht kompatibel sind [132, S.6]. Diese Inflexibilität im Spektrum der einsetzbaren Betriebssysteme liegt den linuxoiden Containerlösungen zugrunde. Jedoch gibt es Bemühungen seitens *Docker* und *Microsoft* eine Docker-Lösung für *Microsoft Windows Server 2016* zu implementieren. Durch das *Open Container Project* (siehe Kapitel 2.3.3) ist es dem Unterstützer *Microsoft* nun möglich, den *Windows*-Kernel für das neue standardisierte Containerformat vorzubereiten [121].

Ein großer Vorteil jedoch, der sich durch das schlankere Design ergibt, ist eine fast native Performance der Container [136, S.1], da der Virtualisierungs-Overhead des Hypervisors wegfällt. Unter dem Gesichtspunkt der Rechenleistung beispielsweise, kommt es bei Containerlösungen im Durchschnitt zu einem Overhead von ca. 4%, wenn diese mit der nativen Leistung derselben Hardwarekonfiguration verglichen wird [136, S.4][109, S.5]. In traditionellen Virtualisierungen beansprucht der Hypervisor allein etwa 10-20% der Hostkapazität [100, S.2][109, S.5]. Ein Benchmarktest, der den Durchsatz (Operationen pro Sekunde) eines *VoltDB*-Setups[89] von Hypervisor-basierte Cloudlösungen mit containerbasierten Cloudlösungen verglich, kam zu dem

Ergebnis, dass die Containerlösung unter genanntem Gesichtspunkt sogar eine fünffache Leistung aufweist [82, S.2+3]. In der Praxis machen sich diese Verhältnisse an einer hohen Dichte an Containern bemerkbar und führen zu einer besseren Ressourcenausnutzung [132, S.7+8]. Der resultierende Performancegewinn ist v.a. wichtig für *High Performance Computing*-Umgebungen (HPC), sowie ressourcenbeschränkte Umgebungen wie mobile Geräte und *Embedded Systems* [124, S.1].

Aus der Sicht der Sicherheit kann das Fehlen eines Hypervisors doppeldeutig interpretiert werden: Zum einen schrumpft die Angriffsfläche des Hosts, da nicht das gesamte Betriebssystem virtualisiert wird [132, S.6]. Je weniger Hostfunktionen virtualisiert werden, desto geringer wird auch das Sicherheitsrisiko, dass eine Hostfunktion von einem Angreifer missbraucht werden kann. Zum anderen ist es aus Sicht der Architektur unsicherer, die virtuellen Umgebungen direkt auf einem Host laufen zu lassen. Angriffe, die von einem Gast-OS über die zusätzliche Softwareschicht eines Hypervisors den Host als Ziel haben, sind, wie der Erfolg von Hypervisoren der letzten Jahre bestätigt, sehr schwierig durchzuführen. Deswegen werden Container als weniger sicher im Vergleich zur Hypervisor-gestützten Virtualisierung gesehen [132, S.6]. Mit welchen Sicherheitsmechanismen Container ausgerüstet sind, ist Gegenstand von Kapitel 4.

Auch im Lifecycle von virtuellen Instanzen bieten Container Vorteile: Während in traditionellen VMs ein Neustart dieser Sekunden bis Minuten beansprucht, da das komplette Gast-OS neu gestartet werden muss, entspricht ein Containerneustart nur einem Prozessneustart auf dem Host, der im Millisekundenbereich abgeschlossen ist [101, S.2].

2.1.3 Einordnung Docker

Docker gehört zu den Technologien der Container-basierten Virtualisierung und hat seinen Ursprung in *Linux Container (LXC)*, das mit Docker auf Kernelebene und v.a. Anwendungsebene erweitert wurde [132, S.7][136, S.1][101, S.2].

Docker ist wie in Kapitel 2.1.2 zuvor erwähnt, nicht die erste containerbasierte Virtualisierungslösung. Einige ältere Containersysteme, wie z.B. *Solaris Zones*, existieren bereits länger als Docker, erlangten allerdings nie die Popularität von Docker. Der anhaltende Erfolg von Docker beruht jedoch überwiegend nicht auf den technischen Eigenschaften, die sich von jener der Konkurrenz wie *LXC* und *rkt* abheben, sondern vielmehr auf den Tools und einem effizienten Workflow, den *Docker* seinen Kunden anbietet.

2.2 Sicherheitsziele in der IT

In der IT existieren mehrere Sicherheitsziele, die auf unterschiedliche Art und Weise erreicht werden. Je nach Anwendungsgebiet und Anforderungen werden die einzelnen Ziele unterschiedlich stark priorisiert bzw. nicht angewandt. Auch im Zusammenhang mit der Virtualisierung lassen sich die Sicherheitsziele eingrenzen, sodass in den folgenden Abschnitten nur die in dieser Arbeit relevanten Ziele aufgeführt sind. Grundsätzlich wird zwischen der Sicherheit von Computersystemen und Netzwerken unterschieden. Die Form der Virtualisierung, die mit Docker realisiert wird, ist Element von Computersystemen und ist von den Netzwerken, an die physische Hosts angeschlossen sind, unabhängig. Aus diesem Grund sind in diesem Kapitel nur die vier in [131, S.712f.] von Tanenbaum definierten Sicherheitsziele definiert.

2.2.1 Vertraulichkeit

Die Vertraulichkeit steht für das Konzept von Geheimhaltung. In Computersystemen können z.B. in Kapitel 4 beschriebene Mechanismen des Kernels aktiviert werden, um eine bestimmte Information vor unautorisierten Zugriffen zu schützen. Ausschließlich Befugte haben Zugang zu der Information. In einem Multi-Tenancy-System ist es beispielsweise notwendig, Kundeninformationen von unberechtigten Nutzern geheimzuhalten.

2.2.2 Integrität

Unter Integrität versteht man die Zusicherung, dass bestimmte Daten original und vollständig vorliegen, sowie nachweisbar nicht manipuliert wurden. Ein Sicherheitsmechanismus sieht vor, dass es dem Besitzer einer Datei erlaubt sein soll, eine bestimmte Datei zu modifizieren. In Multi-Tenancy-Systemen muss beispielsweise gewährleistet werden, dass die Daten der einzelnen Kunden geschützt werden und kein Kunde die Integrität von Daten eines anderen Kunden verletzen kann.

2.2.3 Verfügbarkeit

Die Verfügbarkeit bezeichnet die Eigenschaft eines Systems, Anfragen jederzeit zu verarbeiten und andere Systeme nicht negativ zu beeinflussen. Ein prominentes Beispiel eines Angriffs auf die Verfügbarkeit ist die Denial of Service-Attacke, kurz DoS-Attacke, die ein System mit einer Anfragenflut stört bis es unbenutzbar wird und im System gehaltene Ressourcen nicht zugreifbar sind.

2.2.4 Datenschutz

Der Datenschutz stellt sicher, dass Personen vor dem Missbrauch ihrer persönlicher Daten geschützt sind. Aspekte des Datenschutzes haben keine Bedeutung für die Sicherheitsbetrachtung von Docker und sind deswegen nur aus Gründen der Vollständigkeit an dieser Stelle aufgeführt.

2.3 Einführung in Docker

Docker ist eine unter der Apache 2.0 Lizenz veröffentlichte, quelloffene Engine, die den Einsatz von Anwendungen in Containern automatisiert. Sie

ist überwiegend in der Programmiersprache *Golang* implementiert und wurde seit ihrem ersten Release im März 2013 von dem von Solomon Hykes gegründeten Unternehmen *Docker, Inc.*[62], vormals *dotCloud Inc.*, sowie mehr als 1.600 freiwillig mitwirkenden Entwicklern ständig weiterentwickelt. [38][132, S.7][36][1].

Der große Vorteil von Docker gegenüber älteren Containerlösungen, also auch dem Docker-Vorgänger *LXC*, ist das Level an Abstraktion und die Bedienungsfreundlichkeit, die Nutzern ermöglicht wird. Während sich Lösungen vor Docker auf dem Markt durch deren schwierige Installation und Management sowie schwachen Automatisierungsfunktionen nicht etablieren konnten, adressiert Docker genau diese Schwachpunkte [132, S.7] und bietet neben Containern viele Tools und einen Workflow für Entwickler, die beide die Arbeit mit Containern erleichtern sollen [100, S.1].

Wenn wie von Docker empfohlen in jedem Container nur eine Anwendung läuft, begünstigt das eine moderne Service-orientierte Architektur mit *Microservices*. Nach dieser Architektur werden Anwendungen oder Services verteilt zur Verfügung gestellt und durch eine Serie an miteinander kommunizierenden Containern umgesetzt. Der Grad an Modularisierung der dadurch entsteht, kann für die Verteilung, die Skalierung und das Debugging von Service- oder Anwendungskomponenten (Containern) eingesetzt werden [132, S.9]. Je nach Usecase können Container Testumgebungen, Anwendungen bzw. Teile davon, oder Replikate komplexer Anwendungen für Entwicklungs- und Produktionszwecke abbilden. Container also nehmen die Rolle austauschbarer, kombinierbarer und portierbarer Module eines Systems ein [132, S.12].

Eine bekannte Herausforderung in der Softwareentwicklung ist Code, der in der Umgebung eines Entwicklers fehlerfrei ausgeführt wird, jedoch in Produktionsumgebungen Fehler verursacht. In der Regel fallen beide Umgebungen in unterschiedliche personelle Zuständigkeitsbereiche, was vereinfacht eine fehleranfällige Übergabe von Entwicklungs- nach Produktionsumgebung mit sich zieht. Diesem Umstand wurde in der Industrie teilweise mit der Einführung von *DevOps*-Teams entgegengewirkt.

Das Kernproblem im genannten Szenario sind die Entwicklungs- und Produktionsumgebung, zwischen denen Code ausgetauscht wird, da diese sich in Größe, Form und Administration unterscheiden können. Einen anderen Ansatz diese Problem auf eine technische Art und Weise zu lösen, bieten Container. Quellcode - bzw. dessen ausführbarer Build - wird inklusive Ausführungsumgebung flexibel von einem Laptop, auf dem er entwickelt wurde, auf physische oder virtuelle Test- und später Produktionsserver übertragen. Letztere liegen in der Praxis häufig in einer externen Cloud-Infrastruktur, wie z.B. *Azure* des Dienstleisters *Microsoft*. Mit hoher Wahrscheinlichkeit sind die Anwendungscontainer unabhängig von der Infrastruktur sofort startfähig. Dieser kurzlebige Zyklus zwischen Entwicklung, Testen und produktivem Deployment, erlaubt einen effizienten und konsistenten Workflow [132, S.8+12], der mit den Konzepten der *Continuous Integration* und *Continuous Delivery* unterstützt werden kann.

Da Quellcode das wertvollste Asset der meisten IT-Firmen ist und dieser erst dann Wert hat, wenn er bei einem Kunden den produktiven Betrieb aufnimmt, ist der beschriebene Workflow ein wichtiges Entscheidungskriterium bei der Wahl der Virtualisierungslösung [100, S.1]. Das Tooling und die Unterstützung des Workflows ist Dockers große Stärke.

Die folgenden Unterkapitel gehen auf die einzelnen nativen Komponenten im Docker-Ökosystem ein. Nachdem zuerst die Architektur einer Docker-Umgebung sowie zum Betrieb von Containern benötigte Dockerfiles und Formate definiert werden, rückt der Fokus auf praxisnähere Aspekte wie Images, Container und Registries.

2.3.1 Docker Architektur

Docker selbst ist nach einem Client-Server-Modell aufgebaut: Ein Docker-Client kommuniziert mit einem Docker-Daemon, also ein Prozess der den Server abbildet [21]. Beide Teile können auf einer Maschine oder einzeln auf unterschiedlichen Hosts laufen. Die Kommunikation zwischen Client und Daemon geschieht über eine RESTful API. Wie Abb.2 zeigt, ist es dadurch

auch möglich Befehle entfernter Clients über ein Netzwerk an den Daemon zu senden [105, S.3].

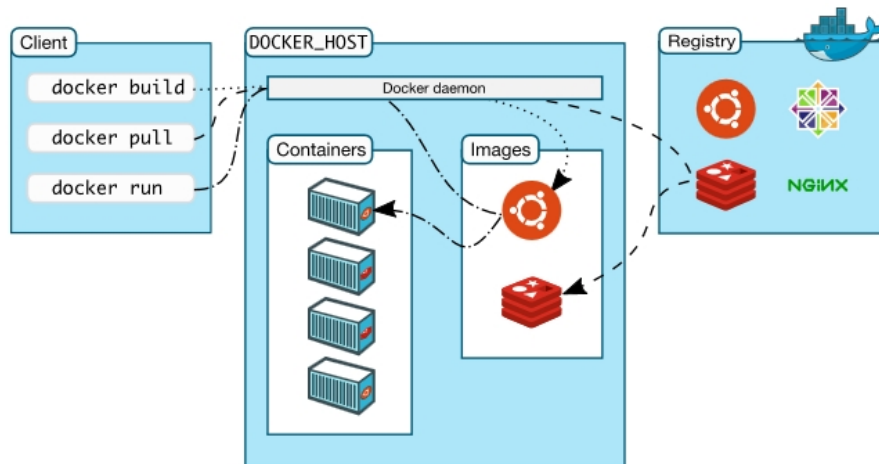


Abbildung 2: Die Client-Server-Architektur von Docker [21].

Der Daemon kann von einer Registry Images (siehe Kapitel 2.3.4 und 2.3.6) beziehen, z.B. dem öffentlichen Docker Hub.

Der Docker-Host selbst ist wie in Abb.3 dargestellt, aufgebaut. Im Idealfall läuft auf der Hardware ein minimales Linux-Betriebssystem, auf dem die Docker-Engine installiert ist. Die Engine verwaltet im Betrieb die Container (siehe Kapitel 2.3.5), in denen in Abb.3 die Apps A-E laufen. Wie auch in der Grafik zu sehen ist, teilen sich die Container gemeinsam verwendete Bibliotheken.

2.3.2 Dockerfile

Ein Dockerfile ist eine Datei mit selbigem Namen, die ein oder mehrere Anweisungen enthält. Letztere werden konsekutiv ausgeführt und führen jeweils zu einer neuen Schicht, die in das später generierte Image einfließt. Damit stellen Dockerfiles eine einfache Möglichkeit dar, Images automatisiert zu generieren.

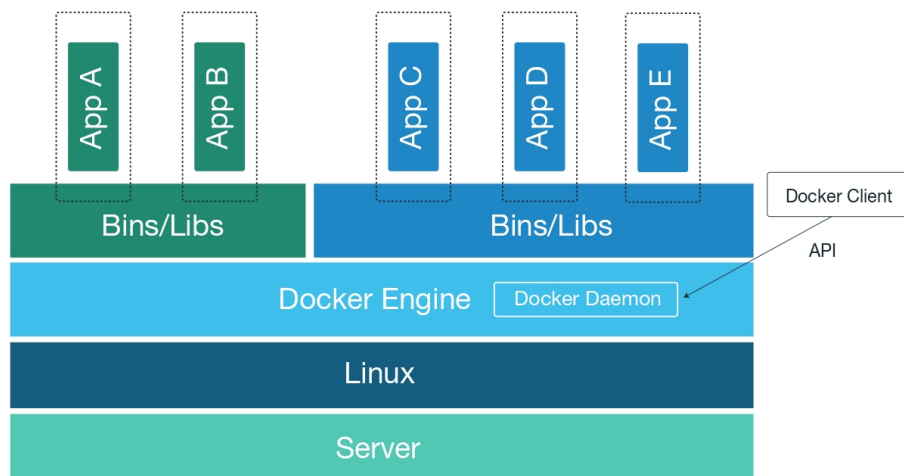


Abbildung 3: Aufbau eines Docker-Hosts, wenn dieser unter einem Linux-Betriebssystem betrieben wird, das direkt auf der Serverhardware läuft. [105, S.3].

Eine Anweisung kann z.B. sein, ein Tool zu installieren oder zu starten, eine Umgebungsvariable festlegen oder einen Port zu öffnen. Ein funktionstüchtiges, minimalistisches Dockerfile ist im Folgenden dargestellt und erklärt.

```
FROM ubuntu
MAINTAINER Moritz Hoffmann <mh203@hdm-stuttgart.de>

RUN \
    apt-get update && \
    apt-get install -y nginx

WORKDIR /etc/nginx
CMD ["nginx"]

EXPOSE 80
EXPOSE 443
```

Die Erklärung der einzelnen Anweisungen [61]:

- **FROM:** Setzt das Basisimage für alle folgenden Anweisungen. Jedes Dockerfile muss diese Anweisung am Anfang enthalten.
- **MAINTAINER:** Hiermit kann ein Autor des Images festgelegt werden.
- **RUN:** Führt angehängten Befehl während des Buildvorgangs aus und erzeugt damit eine neue Schicht.
- **WORKDIR:** Setzt das Arbeitsverzeichnis, von dem aus z.B. alle folgenden **RUN-** und **CMD-**Anweisungen ausgeführt werden. Kann mehrmals pro Dockerfile vorkommen.
- **CMD:** Führt angehängten Befehl aus, wenn der Container gestartet wird. Pro Dockerfile kann es nur eine **CMD-**Anweisung geben.
- **EXPOSE:** Öffnet angegebenen Port des Containers zur Laufzeit, in obigem Beispiel Port 80 und 443 für HTTP und HTTPS. Gebunden wird dieser standardmäßig auf dem Host auf einen „registered“ Port (1024-49151).

2.3.3 Containerformate LXC, libcontainer, runC und OCF

Containerformate bilden das Herzstück der containerbasierten Virtualisierung. In ihnen ist in Form einer API definiert, auf welche Art und Weise Container mit dem Host kommunizieren. Es wird z.B. festgelegt, wie das Dateisystem des Hosts verwendet wird, welche Hostfeatures genutzt werden dürfen und wie die allgemeine Laufzeitumgebung von Containern spezifiziert ist.

Dockers Containerformat hat sich in den letzten Monaten oft verändert, daher soll an dieser Stelle auf die neusten Entwicklungen eingegangen werden.

Im ersten Release von Docker wurde die Ausführungsumgebung *LXC* verwendet, die im März 2014 von der *Docker*-eigenen Entwicklung *libcontainer* abgelöst wurde. *libcontainer* ist komplett in der Programmiersprache *Golang*

implementiert und kann ohne Dependencies mit dem Kernel kommunizieren [17].

Ende Juni 2015 hat Docker angekündigt, zusammen mit mehr als 20 Vertretern aus der Industrie, u.a. *Google*, *IBM* und *VMware*, einen neuen Standard *Open Container Format (OCF)* zu schaffen, welcher im Rahmen des *Open Container Projects (OCP)* entstehen soll [18]. Am gleichen Tag hat Docker *runC* angekündigt, eine Implementierung des *OCF*, die maßgeblich auf dem alten Format *libcontainer* beruht, aber die Spezifikationen von *OCF* umsetzt [52][41][50].

2.3.4 Images

Images bilden als unveränderbare Files die Basis von Containern. Sie sind einfach portierbar und können geteilt, gespeichert und aktualisiert werden. Images sind durch ein *Union-Dateisystem* in Schichten gegliedert, die überlagert ein Image ergeben, das als Container gestartet werden kann [132, S.11]. *Union-Dateisysteme* wie *AuFS* und *Device Mapper* haben gemeinsam, dass sie alle auf dem *Copy-on-write*-Modell basieren [132, S.8][100, S.3][105, S.4].

Genauer gesagt besteht ein Image aus einem Manifest, das auf Datenebene ein oder mehrere Schichten (Layers) referenziert. Images und Schichten sind jeweils über Hashwerte eindeutig referenzierbar und liegen auf dem Docker-Host im Verzeichnis `/var/lib/docker/graph/`. Im Unterordner eines Images liegen mehrere Image-spezifische Dateien (vgl. Abb.4), u.a. das Manifest in der Datei `json`, das in einer JSON-Struktur vorliegt und neben Metainformationen auch Details des Dockerfiles, aus dem das Image generiert wurde, beinhaltet [39].

```
root@moritz-VirtualBox: /var/lib/docker/graph/8d74077f3b19b8a2e663f106aafc2569fea0be6ba79de76988d2da00e87f0201# ll
total 44
drwx----- 2 root root  4096 Jan 21 12:44 ./
drwx----- 8 root root 20480 Jan 21 13:14 ../
-rw----- 1 root root    71 Jan 21 12:44 checksum
-rw----- 1 root root  1294 Jan 21 12:44 json
-rw----- 1 root root     1 Jan 21 12:44 layersize
-rw----- 1 root root    82 Jan 21 12:44 tar-data.json.gz
-rw----- 1 root root  1271 Jan 21 12:44 v1Compatibility
```

Abbildung 4: Dateien im Ordner eines Images (eigene Abbildung).

Images werden Schritt für Schritt erstellt, z.B. mit den folgenden Aktionen [132, S.11].

- Eine Datei hinzufügen
- Ein Kommando ausführen, z.B. ein Tool mittels des Paketmanagers `apt` installieren
- Einen Port öffnen, z.B. den Port 80 für einen Webserver

Die Schichten eines Images umfassen in der Regel jeweils eine minimale Ausführungsumgebung mit Bibliotheken, Binaries und Hilfspaketen sowie den Quellcode der Anwendung, die im Container ausgeführt werden soll. Die Schichtenstruktur erlaubt es, Images modularisiert aufzubauen, sodass sich Änderungen eines Images nur auf eine Schicht auswirken. Soll z.B. in ein bestehendes Image der Webserver *Nginx* integriert werden, kann dieser mit dem Kommando `apt-get install nginx` installiert werden, was eine neue Schicht im Image erzeugt. Eine Auswahl an möglichen Befehlen, die jeweils eine Schicht generieren, ist im Dockerfile-Kapitel 2.3.2 gegeben.

Mit mehreren ähnlichen Images ist gewährleistet, dass nur die konkreten Unterschiede zwischen diesen als eigene Schichten hinterlegt sind. Eine gemeinsame Codebasis, die von mehreren Images genutzt wird, liegt in wenigen Schichten, die sich die Images teilen [100, S.3]. Wie in Abb.5 beispielhaft zu sehen ist, basieren die beiden Images `redis:3.0.6` und `nginx:1.9.9` auf zwei gleichen Schichten, die durch die Anweisungen `ADD` und `CMD` erzeugt werden. In dieser Abbildung sind die Informationen zu dem Image in der ersten Zeile zu sehen und die Schichten der Images sind in den jeweiligen Spalten vertikal gelistet.

Über die Kommandozeile kann z.B. das Image eines *CentOS*-Betriebssystems von der öffentlichen Docker-Registry (siehe Kapitel 2.3.6) wie in Abb.6 mit dem Befehl `docker pull nginx` auf die lokale Maschine gespeichert werden [63][26]. Wie in Abb.6 und Abb.5 zu sehen ist, werden sechs Schichten heruntergeladen, die jeweils über einen Hashwert identifiziert werden und zusammengefügt das angefragte Image `centos:7.2.1511` ergeben.

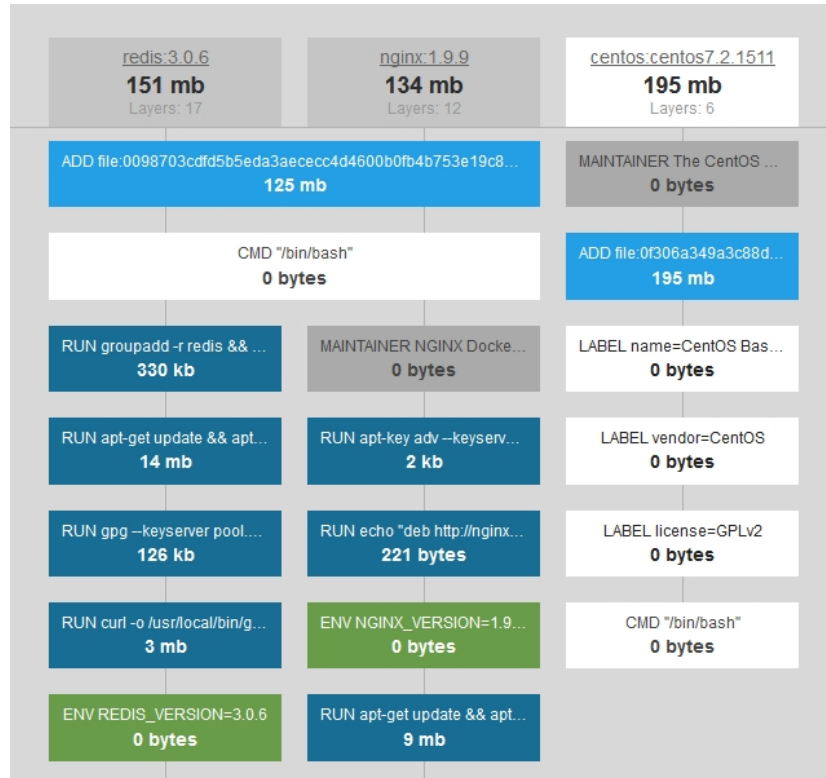


Abbildung 5: Visualisierung eines Vergleichs von Images von *Redis*, *Nginx* und *CentOS* auf Schichtebene [51].

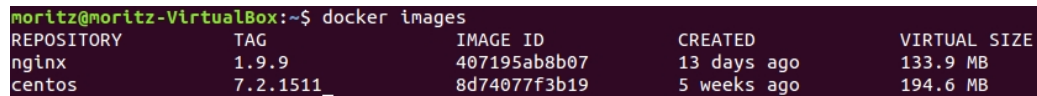
```

moritz@moritz-VirtualBox:~$ docker pull centos:7.2.1511
7.2.1511: Pulling from library/centos
fa5be2806d4c: Pull complete
fd95e76c4fb2: Pull complete
3eeaf11e482e: Pull complete
c022c5af2ce4: Pull complete
aef507094d93: Pull complete
8d74077f3b19: Pull complete
Digest: sha256:9e234be1c6be5de7dd1dae8ed1e1d089e16169df841e9080dfdbdb7e6ad83e5e
Status: Downloaded newer image for centos:7.2.1511

```

Abbildung 6: Screenshot von der Ausführung des Befehls `docker pull <image>` (eigene Abbildung).

Eine Liste aller lokal vorliegenden Images, wie in Abb.7, kann mit dem Befehl `docker images` in der Shell generiert werden [25].



REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
nginx	1.9.9	407195ab8b07	13 days ago	133.9 MB
centos	7.2.1511	8d74077f3b19	5 weeks ago	194.6 MB

Abbildung 7: Screenshot von der Ausführung des Befehls `docker images` (eigene Abbildung).

2.3.5 Container

Ein Container ist die laufende Instanz eines Images, die in Sekundenbruchteilen gestartet werden kann [100, S.1]. Sie beinhalten eine idealerweise minimale Laufzeitumgebung, in der eine oder mehrere Anwendungen laufen.

In Bezug zu anderen Docker-Begriffen, enthält ein Container ein Image und erlaubt eine Reihe von Operationen, die auf ihn angewandt werden können. Darunter fallen z.B. das Erstellen, Starten, Stoppen, Neustarten und Beenden eines Containers. Welchen Inhalt ein Container hat, also ob ein Container z.B. auf einem Datenbank- oder Webserver-Image beruht, ist dafür unerheblich [132, S.12][101, S.2].

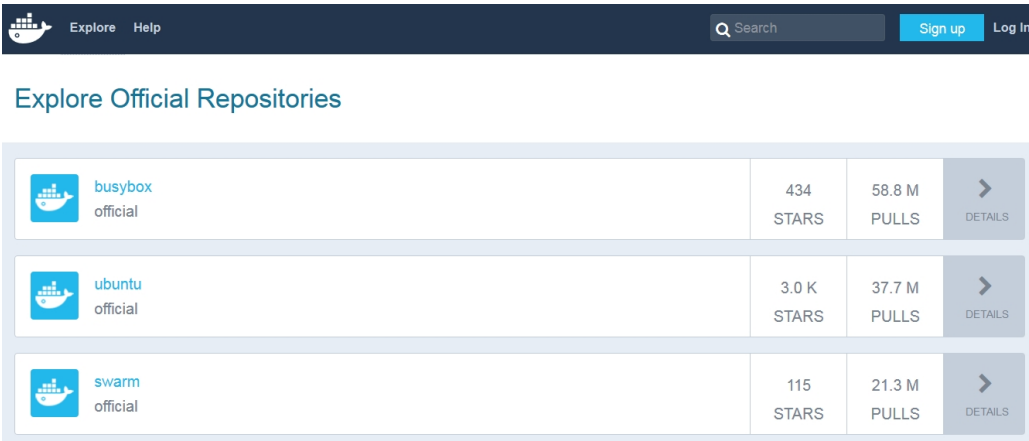
Ein Container wird als privilegiert bezeichnet, wenn er mit Root-Rechten gestartet wurde. Standardmäßig startet ein Container unprivilegiert ohne Root-Rechte. Mit einem reduzierten Set an ausführbaren Aktionen, die über verschiedene, in Kapitel 4 vorgestellte Mechanismen definiert werden, lassen sich Container unabhängig von Root-Rechten einschränken.

2.3.6 Registries

Eine Registry ist eine Webanwendung, die als Speicher- und Verteilerplattform für Images dient. Über eine wohldefinierte API, die Registry-API, sind Docker-Komponenten in der Lage mit Registries zu kommunizieren. Images

sind mit Tags versehen in Repositories gegliedert, die wiederum in der Registry liegen [20]. Ein Repository besteht aus mindestens einem Image.

Docker stellt eine Vielzahl an Images öffentlich und frei verwendbar in einem Service, dem Docker Hub, zur Verfügung [132, S.11][103, S.3][20]. Für dieses System können Personen und Organisationen Accounts anlegen und eigenständig Images in öffentliche und private Repositories hochladen. Das Docker-Hub bietet bereits mehr als 150.000 Repositories, die etwa 240.000 Nutzer zusammenstellten und hochluden, zur freien Verwendung an (Stand Juni 2015) [80, S.16]. Wie in Abb.8 zu sehen ist, werden auch Nutzungsstatistiken pro Image gesammelt und angezeigt. Durch diese erweiternden Features ist das Docker Hub per Definition keine Registry, sondern enthält eine Registry als Teil des Angebotspektrums.






Repository	Stars	Pulls	Details
 busybox official	434 STARS	58.8 M PULLS	DETAILS
 ubuntu official	3.0 K STARS	37.7 M PULLS	DETAILS
 swarm official	115 STARS	21.3 M PULLS	DETAILS

Abbildung 8: Web-UI des Docker Hubs mit den beliebtesten Repositories [28].

Um Images in einem Repository voneinander zu unterscheiden, werden Images Tags zugewiesen, um beispielweise mehrere Versionen eines Images in einem Repository zu kennzeichnen. Die Images werden nach dem Schema `<repository>:<tag>` identifiziert. So gibt es z.B. im offiziellen Repository des Webserver *Nginx* Images mit den Tags `latest`, `1`, `1.9` und `1.9.9` [63]. Wenn bei dem Download kein Tag angegeben ist, wie in Kapitel wird automatisch das aktuellste Image mit dem Tag `latest` bezogen.

Kapitel 3

Fragestellungen / Problemformulierung

Die Wertschöpfung moderner IT-Unternehmen beruht auf dem Angebot von Diensten, auch Services genannt, die über das Internet den Nutzern zur Verfügung gestellt werden. Die Services werden von Anwendungen angeboten, die selbst in Rechenzentren betrieben werden. Der überwiegende Vermögensgegenstand in diesem Modell ist die Software, die in den Rechenzentren produktiv läuft. Der Wert dieser ist direkt abhängig von der Funktionstüchtigkeit eines Rechenzentrums. Je nach Anwendungsfall kommt den Sicherheitszielen aus Kapitel 2.2 unterschiedliche Wichtigkeit zu.

Kunden, die ihr Produkt über Rechenzentren anbieten sind an Sicherheitsfeatures interessiert, die die drei Sicherheitsziele sicherstellen. Die Betreiber von Rechenzentren wiederum müssen diese Nachfrage befriedigen, wollen jedoch gleichzeitig auch selbst Gewinn machen. Durch den Betrieb von Containern in Rechenzentren kann im Vergleich zu Hypervisorlösungen theoretisch mehr Gewinn geschöpft werden, da sich mit ihnen - bei gleicher Hardware - mehr virtuelle Kapazität realisieren lässt. Die Sicherheit virtueller Instanzen darf im Betrieb von Containern jedoch nicht leiden, um weiterhin den Kundenanforderungen zu entsprechen.

Die erste zentrale Frage ist demnach, in wie weit Container für Kundenservices in Rechenzentren Sicherheit bieten. Diese stark verallgemeinerte Fragestellung kann anhand einer Risikoanalyse, wie sie z.B. von Mandl in [119, S.36] vorgeschlagen ist, genauer formuliert werden. Die Risikoanalyse dient gleichzeitig dazu, Annahmen vorzustellen und Schlussfolgerungen zu ziehen, auf deren Basis gegen Ende des Kapitels der Untersuchungsgegenstand der vorliegenden Arbeit definiert und abgegrenzt ist.

Identifikation der Vermögensgegenstände / Wertschöpfungsmerkmale:

Für Betreiber von Rechenzentren der sichere und zuverlässige Betrieb von Kundensoftware. Gewährleistung von Vertraulichkeit, Integrität und Verfügbarkeit der Daten und Anwendungen allen Kunden haben für den Betreiber höchste Priorität.

Identifikation der Bedrohungen: Die verschiedenen Bedrohungsarten können auf Basis eines Systemmodells von containerbasierter Virtualisierung identifiziert werden.

Das Systemmodell von Hypervisorsystemen kann nicht verwendet werden, da das Design der Containersysteme stark von Erstgenannten abweicht. Während nach [119, S.125] virtuelle Maschinen als eigener Sicherheitsmechanismus des Betriebssystems aufgelistet ist, stimmt das für die Containersysteme und deren Konzeption nicht mehr. Andere Sicherheitsfeatures des Hosts, die ab Kapitel 4 vorgestellt werden, müssen aktiviert werden, um die Containersicherheit zu erhöhen.

Bild zum Systemmodell

Bei der ersten Betrachtung kann von zwei unterschiedlichen Gefahrenquellen gesprochen werden.

A: Ausführung von unkontrollierbaren Images: In Containern werden Anwendungen ausgeführt, die nicht zwangsweise vertrauenswürdig sind. Wenn beispielsweise Containerimages von einem öffentlichen Hub bezogen werden, existiert keine Garantie, dass aus diesen Images gestartete Container gegen keines der drei zuvor definierten Sicherheitsziele verstößt. Durch

die Komplexität moderner Anwendungen und deren Abhängigkeiten zu Bibliotheken, ist es selbst bei quelloffenen Anwendungen schwierig, diese als vertrauenswürdig einzustufen. Deswegen muss davon ausgegangen werden, dass in Containern willkürliche Programme ablaufen (*interne Gefahrenquellen*), die - versehentlich oder beabsichtigt - den Host schädigen können.

B: Container als Server im Internet: Viele Container stellen einen Dienst über das Internet zur Verfügung und stehen dadurch mit der Außenwelt in Kontakt. Ein Webserver beispielsweise, kann als Containerapplikation betrieben werden, indem er über einen Port Anfragen von Clients entgegennimmt und diese nach abgeschlossener Verarbeitung beantwortet. Die Notwendigkeit, Containerschnittstellen über das Internet anzubieten, kann von Angreifern (*externe Gefahrenquellen*) ausgenutzt werden, um Sicherheitsziele zu verletzen. Der Schutz des Netzwerks und der Verbindung des Hosts an das Internet muss unabhängig von der eingesetzten Containertechnologie realisiert werden.

Auswirkungen potenzieller Risiken: Bei einer näheren Betrachtung jedoch, sind aus Sicht des Hostsystems die Folgen beider Gefahrenquellen identisch. In beiden Fällen muss davon ausgegangen werden, dass ein Container schadhafte Code ausführt. Ob das zugrundeliegende Image fehlerhaft bzw. manipuliert ist (Gefahr A), oder ein Container aktiv von einem Angreifer kontrolliert wird (Gefahr B), spielt für den Host keine Rolle. Der Host muss in der Lage sein, die Systemsicherheit aufrecht zu erhalten. Die Systemsicherheit umfasst hierbei den Schutz des Hosts sowie anderer Container.

Per Definition wird ein Container, der schadhafte Code ausführen kann, also böse bezeichnet. Ein korrekt funktionierender Container, dessen Sicherheitsziele aufrecht gehalten werden sollen, wird abgekürzt als legitim aufgeführt.

TODO: Formale Definition: Set an Containern C auf einem Host. Annahme ist, dass nicht-leeres Subset C' auf dem Host existiert, das böse ist. C' kann 1 bis $\#C$ groß sein. Wenn C' maximal $\#C-1$ groß ist, führt das zu einer stärkeren Behauptung, da es hierbei auch min. 1 legitimen Container zu

schützen gibt

Container c' aus dem Set C' ist in der Lage alle drei Sicherheitsziele zu verletzen. Man-in-the-Middle kann Vertraulichkeit verletzt werden, indem geheime Informationen abgefangen werden. Mit geheimen Informationen können unter Umständen Daten unrechtmäßig manipuliert werden, was die Integrität beeinflusst. Normale Programmflüsse können unterbrochen werden, was Beeinträchtigungen für die Verfügbarkeit mit sich zieht. Auch DoS-Attacken sind von c' aus möglich.

Einige der von c' geführten Angriffe sind nur durchführbar, wenn der Container im Besitz bestimmter Rechte ist. Die Privilegien, die ein Container standardmäßig besitzt, können fest definiert werden.

Privilege Escalation als extra Punkt aufführen? Ist eigtl kein direktes Sicherheitsziel. Eher im Punkt Gegenmaßnahmen aufführen....

Schutzmechanismen / Gegenmaßnahmen: Angewandt auf die Praxis: es muss universeller Ansatz gewählt werden, da jeder Kunde andere Sicherheitsanforderungen hat. Demnach können Schwachstellen, die die Vertraulichkeit, die Integrität oder Verfügbarkeit der Kundensoftware bedrohen, fatale Folgen für den Umsatz und die Reputation der Kunden und Betreiber von Rechenzentren ergeben.

Welche Sicherheitsmodelle und -mechanismen können eingesetzt werden, um Bedrohungspotential von aufgeführten Gefahrenquellen zu minimieren.

Darunter fallen mit Software realisierte Mechanismen zur Isolation, Ressourcenverwaltung und Zugriffskontrollen Einteilung der Kontrollmechanismen in administrative, technische und physische Kontrollen [119, S.40]:

- **Administrative Kontrollen:** Enthält Management-Kontrollen, die z.B. durch Konfigurationen, Entwicklung einer Sicherheitspolitik, Best Practices, Sicherheitsschulungen des Personals, umgesetzt werden.
- **Technische Kontrollen:** Umfasst alle hardware- und softwarebasierten Mechanismen, z.B. ein Zugriffsschutz unter Verwendung einer DAC oder MAC. Docker verfolgt eine softwarebasierte *Defense in depth*, bei

der verschiedenartige Sicherheitsschichten realisiert werden, um einen bestmöglichen Schutz zu ermöglichen. Eine Geheimhaltung von technischen Kontrollen, auch *Security through obscurity* genannt, kann nicht praktiziert werden, da Docker und Linux selbst quelloffene Projekte sind.

- **Physische Kontrollen:** Beinhalten Mechanismen wie Sicherheitsschleusen, Schlösser und Wachpersonal. Obwohl ein Bezug zum Betrieb von Rechenzentren hergestellt werden kann, haben physische Kontrollen keine spezifische Relevanz für die containerbasierte Virtualisierung und sind aus diesem Grund an dieser Stelle nur zum Zweck der Vollständigkeit aufgeführt.

Die vorliegende Arbeit konzentriert sich auf die technischen, softwarebasierten Kontrollen, die von Docker eingesetzt werden. Aber auch administrative Methoden werden in Kapitel 5 vorgestellt.

Kapitel 4

Sicherheit durch Linux-Funktionen

Text hier als Einleitung dazu. Dieses Kapitel stellt die von dem Betriebssystem Linux ermöglichten Sicherheitsmodelle und -mechanismen vor, von denen Docker Gebrauch macht bzw. die Docker unterstützt.

Wie In Kapitel 2.1.2 geschildert, wird Docker zur Zeit nur für Linux angeboten. Aus diesem Grund ist eine Untersuchung von Sicherheitsfeatures anderer Betriebssysteme nicht nötig.

4.1 Isolierung durch namespaces

Namespaces stellen den grundlegenden Mechanismus dar, um eine Isolation für Container zu ermöglichen.

Wenn unter Linux ein neuer Prozess gestartet werden soll, wird über *System Calls* dem Kernel mitgeteilt, einen neuen *namespace* bereitzustellen. Je nach Anforderung gibt es verschiedene *namespaces*, z.B. ein *network namespace*, der dem neuen Prozess ein Netzwerkinterface zuweist. Um Container als isolierte Arbeitsbereiche auf einem Host zu erstellen, werden die *namespaces*

des Kernels verwendet. Da Container selbst eine eigene komplette Laufzeitumgebung darstellen sollen, müssen Bereiche des Hosts durch *namespaces* abgedeckt sein, sodass neben dem Netzwerk auch z.B. ein beschränkter Zugriff auf den Arbeitsspeicher und die CPU gewährleistet ist [100, S.3].

Technisch betrachtet beinhaltet ein *namespace* eine Lookup-Tabelle, die global verfügbare Ressourcen abstrahiert und dem *namespace* bereitstellt. Änderungen globaler Ressourcen sind sichtbar für Prozesse im relevanten *namespace*, jedoch unsichtbar für solche außerhalb [102, S.1+2][55]. Dadurch können *namespaces* als Lösungsansatz des Sicherheitsziels Vertraulichkeit betrachtet werden. Sie sind damit der wesentliche Baustein, um eine Containerisolierung zu realisieren.

4.1.1 Prozessisolierung durch den PID namespace

Jeder Container entspricht auf dem Host zunächst einem Prozess. Da die Container untereinander isoliert sein sollen, dürfen auch die zugrundeliegenden Containerprozesse nicht miteinander interferieren.

Docker erreicht diese Isolierung auf Prozessebene durch die Nutzung des *PID namespace*, in denen Container eingebettet werden. Nach diesem hierarchischen Konzept ist es einem Prozess X nur möglich, selbsterzeugte Kindprozesse zu beobachten und mit ihnen zu interagieren. Elternprozesse, also Prozesse die in der Prozesshierarchie über X stehen, sind für X unsichtbar. Der Elternprozesse haben jedoch die volle Kontrolle über X und können diesen z.B. jederzeit mit dem Befehl `kill` beenden. Darüber hinaus haben Elternprozesse die Möglichkeit mit z.B. einem Aufruf von `ps` alle Kindprozesse überwachen.

Übertragen auf die containerbasierte Virtualisierung bedeutet das, dass der Host vollen Zugriff auf die laufenden Container hat, Containerprozesse jedoch weder Kenntnis von Hostprozessen noch von Prozessen anderer Container besitzen (vgl. Abb.9 und Abb.10). Diese Eigenschaft macht es Angreifern schwieriger Schaden anzurichten, da sie ausgehend von kompromitierten Con-

tainern keine Informationen über Prozesse außerhalb des Containers beziehen können.

Ein weiterer Mechanismus des *PID-namespaces* ist eine Besonderheit des Prozesses mit *PID=1*. Der initiale Containerprozess kann mit der *PID=1* gestartet werden, dem es als *init*-ähnlicher Prozess möglich ist, alle Kindprozesse zu terminieren sobald er selbst beendet wird. Somit können komplette Container durch einen Hostzugriff auf den Containerprozess mit *PID=1* umgehend vollständig heruntergefahren werden.

Wie sich der *PID-namespaces* auf einen gestarteten Docker-Container auswirkt, ist in Abb.9 und Abb.10 zu sehen.

```
unconfined          627 ?      Ssl    0:15 /usr/bin/docker daemon -H fd://
docker-default      3698 pts/17  Ss+    0:00 \_ /bin/bash
docker-default      3748 pts/17  S      0:00 \_ sleep 1000
docker-default      3749 pts/17  S      0:00 \_ sleep 1000
```

Abbildung 9: Ausschnitt der Ausgabe des Befehls `ps -eafxZ` auf einem Docker-Host (eigene Abbildung).

Abb.9 stellt einen Ausschnitt der Ausgabe des Befehls `ps -eafxZ` auf dem Host dar. Er enthält die laufenden Docker-Prozesse des Daemons mit *PID=627* (Zeile 1) und einem Container, in dem eine *Bash* gestartet wurde (Zeile 2). Innerhalb der *Bash* wurde der Befehl `sleep 1000` zweifach ausgeführt (Zeile 3 und 4), der laufende Hostprozesse zurückgibt. Wie zu erkennen ist, sind die Containerprozesse aus Sicht des Hosts mit einer eigenen *PID* vollkommen transparent.

```
[root@c8eb0f37ac70 /]# ps -eafxZ
LABEL                                PID TTY      STAT   TIME COMMAND
docker-default (enforce)             1 ?        Ss      0:00 /bin/bash PATH=/usr/lo
docker-default (enforce)            16 ?        S        0:00 sleep 1000 HOSTNAME=c8
docker-default (enforce)            17 ?        S        0:00 sleep 1000 HOSTNAME=c8
docker-default (enforce)            21 ?        R+       0:00 ps -eafxZ HOSTNAME=c8e
```

Abbildung 10: Ausgabe des Befehls `ps -eafxZ` in einem Docker-Container (eigene Abbildung).

Abb.10 zeigt die vollständige Ausgabe des gleichen Befehls innerhalb eines Containers. Prozesse außerhalb des Containers sind unsichtbar. Außerdem

weisen die Containerprozesse im Vergleich zu Abb.9 nun eine andere PID auf. Da die *Bash* der initiale Containerprozess ist, erhält sie die PID=1.

4.1.2 Dateisystemisolierung durch den `mount namespace`

Auch das Hostdateisystem muss von unrechtmäßigen Zugriffen aus Containern geschützt werden.

Dateisysteme sind allgemein wie Prozesse in Kapitel 4.1.1 hierarchisch aufgebaut. Diese können mithilfe von *mount namespace* unterteilt werden, sodass unter Docker jeder Container eine andere Sicht auf die Verzeichnisstruktur des Hosts hat. Nur ein bestimmtes Unterverzeichnis ist für einen Container sichtbar, wenn er dieses als Mountpoint einbindet.

Eine Hostverzeichnis werden jedoch nicht in den *mount namespace* eingezogen, weil sie von den Docker-Containern benötigt werden, um zu operieren.

Dazu gehören die Verzeichnisse:

- `/sys`:
- `/proc/sys`:
- `/proc/sysrq - trigger`:
- `/proc/irq`:
- `/proc/bus`:

Als Konsequenz erben Container diese notwendigen Verzeichnisse direkt von ihrem Host, was ein Sicherheitsrisiko darstellt. Docker dämmt dieses ein, indem es nur einen reinen Lesezugriff ohne Schreibrechte auf diese Verzeichnisse erlaubt [105, S.4]. Außerdem ist es Containern unter Docker nicht erlaubt, Hostverzeichnisse erneut einzubinden, um Schreibrechte sicher auszuschließen. Dieses Verbot wird durch die Verweigerung der *capability* `bla CAP_SYS_ADMIN` für Container erreicht.

Durch das von Docker genutzte und bereits in Kapitel 2.3.4 beschriebene *COW*-basierte Dateisystem, ist es jedem Container möglich, Änderungen in seinem durch den *mount namespace* zugewiesenen Verzeichnis zu speichern. Containerdaten interferieren dadurch nicht und sind containerübergreifend nicht sichtbar, auch beim Betrieb von Containern, die auf einem gleichen Basisimage beruhen [105, S.4].

[103, S.4]

4.1.3 Geräteisolierung durch

In Unix-basierenden Betriebssystemen wie Linux erfolgt der Zugriff auf Hardware über sogenannte *Device Nodes*, die in dem Dateisystem von speziellen Dateien repräsentiert sind.

Ein paar wichtige *Device Nodes* und deren Zuständigkeiten sind im Folgenden aufgeführt.

- */dev/mem*: Arbeitsspeicher
- */dev/sd**: Files für den Zugriff auf Speichermedien
- */dev/tty*: Terminal

Wie zu sehen ist, handelt sich dabei um teils äußerst kritische Komponenten einer Maschine, über die Container unter keinen Umständen verfügen dürfen. Deswegen ist es notwendig den Zugriff auf *Device Nodes* stark einzuschränken, um den Host vor Missbrauch zu schützen.

[103, S.4]

4.1.4 IPC-Isolierung durch den IPC-namespace

Unter IPC versteht man eine Sammlung an Tools, die für den Datenaustausch zwischen Prozessen genutzt werden. Dazu gehören z.B. *Semaphoren*, *Message Queues* und *Shared Memory Segments*.

Ergänzend zu dem *PID namespace*, der die Sichtbarkeit sowie Kontrolle über Prozesse in der Prozesshierarchie einschränkt, kann auch die Kommunikation zwischen Prozessen limitiert werden.

Docker gewährleistet dies durch den Zuweisung eines *IPC-namespaces* pro Container, in dem ein Prozesse nur mit anderen Prozessen in Kontakt treten kann, wenn sich diese in einem gleichen *IPC-namespace* befinden. Eine versehentliche oder beabsichtigte Interferenz mit Prozessen des Hosts oder anderer Container wird damit ausgeschlossen.

[103, S.4]

4.1.5 UTS-Isolierung durch den UTS-namespace

Nur der Vollständigkeit halber aufgelistet? Oder hat der Relevanz für Container? weniger sicherheitsrelevant oder... Mit einem *UTS-namespace* ist es möglich jedem Container einen eigenen Hostnamen zuzuweisen. Der Container kann diesen Namen abfragen und ändern [106, S.3].

4.1.6 Netzwerkisolierung durch den network namespace

Um einen sicheren Betrieb von Docker zu gewährleisten, müssen Container so konfiguriert sein, dass sie weder den Netzwerkverkehr des Hosts noch anderer Container abhören oder manipulieren können.

Dazu stellt Docker jedem Container einen eigenen unabhängigen Netzwerk-Stack zur Verfügung, der durch *network namespaces* realisiert wird. Jeder Namespace hat seine eigene private IP-Adresse, IP-Routingtabelle, Loopback-Interface und Netzwerkgeräte [106, S.2+3]. Eine Kommunikation zu anderen Containern auf dem gleichen oder entfernten Hosts geschieht dann über diese dafür vorgesehenen Schnittstellen.

Um die oben genannten Netzwerkressourcen anzubieten, wird jedem *network namespace* ein eigenes `/proc/net`-Verzeichnis zugewiesen. Die Nutzung von

Befehlen wie `netstat` und `ifconfig` wird damit, aus einem *network namespace* heraus, auch ermöglicht [102, S.7].

Standardmäßig wird von Containern eine *Virtual Ethernet Bridge* namens `docker0` genutzt, um mit dem Host oder anderen Containern zu kommunizieren. Neu gestartete Container werden dieser Bridge hinzugefügt, indem deren Netzwerkinterface `eth0` mit der Bridge verbunden wird. Aus Sicht des Hosts ist das Interface `eth0` ein virtuelles `veth`-Interface [106, S.3].

Die Bridge leitet ohne Filter alle eingehenden Pakete weiter, welchen Umstand dieses Verbindungsdesign anfällig gegenüber ARP-Spoofing und MAC-Flooding macht. Diesem Nachteil kann Abhilfe geschaffen werden, indem manuelle Filtermethoden mittels beispielsweise *ebtables* in die Bridge integriert werden, oder ein anderes Verbindungsdesign auf basis virtueller Netzwerke gewählt wird.

[103, S.4]

4.1.7 Userisolierung (user namespace)

Bislang werden Container unter Docker und anderen linuxbasierten Containerlösungen mit Root-Rechten gestartet. Falls es einem Angreifer in diesem Szenario gelingt, aus der Containerisolation auszubrechen, ist er automatisch Root-User auf dem Host, was ein hohes Sicherheitsrisiko ist. Durch die potentielle Gefahr dieser Vorgehensweise, wird die Einführung von *user namespaces* als Meilenstein der Containersicherheit gewertet.

Dieser Kernel-*namespace* führt einen Mechanismus ein, unter dem Rootrechte in Containern nicht Rootrechten auf dem Host entsprechen, in anderen Worten ein Root-User im Container auf einen Nicht-Root-User auf dem Host aufgelöst wird. Durch das potentielle Sicherheitsrisiko der bisherigen Vorgehensweise,

Linux verwendet *User IDs* (`uids`) und *Group IDs* (`gids`), um Verzeichnisse und Dateien eines Dateisystems sowie Prozesse mit Eignerinformationen

zu verstehen. *user namespaces* erlauben unterschiedliche `uids` und `gids` innerhalb und außerhalb des *namespace*. Im Kontext des Hosts kann dadurch ein unprivilegierter User (ohne Root-Rechte) existieren, während der gleiche User innerhalb von Containern mit *user namespace* privilegiert ist, also im Besitz von Root-Rechten ist [56].

In der Praxis lassen sich mit diesem Konzept jeweils Root-User mit `uid=0` in Container X und Y auf nicht-privilegierte User mit `uid=1000` und `uid=2000` des Hosts abbilden.

Die Unterstützung von *user namespaces* ist schon seit Version 1.6 geplant, wurde aber erst im Februar 2016 mit Version 1.10 in den Master-Branch von Docker integriert [36][68]. Verzögerungen entstanden durch einen Bug der Programmiersprache *Golang* [94], und Integrationschwierigkeiten in die bestehende Docker-Codebasis [87].

In der aktuell neusten Docker-Version 1.10 werden *user namespaces* nicht automatisch verwendet. Sie müssen manuell, wie z.B. in [118] erklärt, aktiviert werden.

Beide Probleme sind jedoch mittlerweile gelöst, wie die erfolgreiche Integration von *user namespaces* in Docker im Oktober 2015 bestätigt [66]. Dadurch, dass *user namespaces* in der Docker-Roadmap als wichtiges Sicherheitsfeature gesehen werden, ist ein Release dessen bald zu erwarten [40].

Auch für Cloudanbieter sind *user namespaces* von Vorteil: Mit einer Auflösung der Container auf Userebene ist es einerseits möglich Servicenutzung auf Userbasis einzugrenzen und andererseits diese auf Userbasis abzurechnen. Wenn ohne *user namespaces* jede gestartete Containerinstanz einem Hostuser mit `uid=0` zugehörig ist, gestaltet sich die Zuordnung schwieriger [107, S.3].

4.2 Ressourcenverwaltung / Limitierung von Ressourcen durch cgroups

DoS-Attacken mit der Absicht das Sicherheitsziel der Verfügbarkeit zu verletzen, gehören in Multi-Tenant-Service-Systemen zu einem gängigen Angriffsmuster [103, S.5]. Um die Verfügbarkeit von Containern sicherzustellen, bietet der Linux-Kernel sogenannte *Control Groups* (kurz **cgroups**) an, die auch von Docker genutzte Möglichkeiten zum Ressourcenmanagement bereitstellen.

cgroups sind historisch aus dem Konzept von sogenannten *Resource Limits*, auch **rlimits** genannt, gewachsen. Mit *rlimits* werden weiche und harte Limits definiert, die pro Prozess angewandt werden. Der Betrieb von Containern verlangen jedoch eine Ressourcenverteilung auf Containerbasis, sodass Limits pro Container, aus technischer Sicht einem Set an Prozessen, vergeben werden.

Viele Containertechnologien erweiterten deswegen **rlimits** mit eigenen Features. Z.b. fügten die Entwickler von *FreeBSD* für den Betrieb von *Jails* sogenannte *Hierarchical Resource Limits* hinzu [33]. *Solaris* bietet die Nutzung von *Resource Pools* an, die eine Partitionierung von Ressourcen implementiert [13]. Auch *OpenVZ* und *Linux-VServer* erweitern **rlimits**, sodass Ressourcenlimits pro Container definiert werden können [124, S.15+16].

Die Nachteile von **rlimits** wurden mit der Implementierung von **cgroups** für den Linux-Kernel behoben. Mit diesem relativ neuen Mechanismus werden Prozesse in hierarchischen Gruppen angeordnet, die individuell verwaltet werden und deren Attribute vererbt werden können. Neben vielseitiger und feingranularer Funktionen zum Management von z.B. CPU- und Speicherressourcen, können unter **cgroups** komplexe Verfahren implementiert werden, die zur Korrektur von limitüberschreitender Prozesse dienen [14]. Die Implementierung von **cgroups** wurde ab 2012 weiter verbessert, sodass eine Update unter dem Namen *Unified Control Group Hierarchy* seit 2014 in den Linux-Kernel integriert ist [32][85]. Von Docker wird die neue *Unified Hierarchy*

noch nicht verwendet [?]

Wichtig zu erwähnen ist, dass die Implementierung von **cgroups**, verglichen mit der von **rlimits**, angeblich noch nicht vollständig ist. Das Feature Dateisysteme als Ressourcen mit **cgroups** zu steuern, fehlt nach Angaben von [124, S.19]. Auch im Quellcode von *runC* ist eine Dateisystem-Interface als „nicht unterstützt“ gekennzeichnet und wird demzufolge auch nicht von Docker genutzt. [37]. Diskussionen im *GitHub*-Repository von Docker verweisen in Bezug zu diesem Feature auf Abhängigkeiten zur Art des Dateisystems. Offenbar lassen sich sogenannte Dateisystem-Quotas nur mit *Device Mapper* und *Brfs* softwaretechnisch lösen, *AuFS* jedoch ermöglicht das nur indirekt über die Zuweisung von Festplattenpartitionen fester Größe. Diese Gegebenheit lässt vermuten, dass eine universelle Lösung aktuell an der Breite unterstützter Dateisysteme scheitert [4]. Die neusten Entwicklungen sehen jedoch eine Quota-Implementierung vor [2].

Dennoch ist diese Erweiterung im Sinne einer einheitlichen Verwaltung von Ressourcen mit **cgroups** vorgesehen [124, S.16+19].

Alle gängigen Linux-basierten Containerlösungen, darunter auch Docker, nutzen aktuell **cgroups**, um Ressourcen für Container zu verwalten [124, S.16]. Der Einsatz von **cgroups** unter Docker umfasst, wie im Quellcode von *runC* zu sehen ist, die Kontrolle über CPU, Arbeitsspeicher, Geräte (*Devices Nodes*, Netzwerkinterfaces und I/O-Operationen auf Speichermedien wie HDD, SSD und USB-Speicher [14][37]. Die Verwaltung von Letzteren wurden mit dem neusten Docker-Release, Version 1.10, erweitert [?].

Über die Kommandozeile lässt sich der **run**-Befehl, der ausgeführt wird, um einen Container zu starten, mit Angaben zur Ressourcennutzung parametrisieren. Z.b. bewirkt die Hintereinanderausführung folgender Befehle, dass dem zuletzt gestarteten Container doppelt so viel CPU-Leistung zur Verfügung gestellt wird, wie dem ersten Container [27].

```
user@machine:$ docker run <IMAGE> --cpu-shares=50
user@machine:$ docker run <IMAGE> --cpu-shares=100
```

Neben dem Ressourcenmanagement bieten `cgroups` auch Nutzungsstatistiken an. Diese können unter Docker mit dem Befehl `docker stats <CONTAINER> [<CONTAINER>]` abgerufen werden [23].

4.3 Einschränkungen von Zugriffsrechten

Kapitel mit nächstem Unterkapitel verschmelzen. Oder eigenes LSM-capability Kapitel machen, was diese Unterteilung rechtfertigt. Zunächst werden zwei grundsätzliche Sicherheitsprobleme von Betriebssystemen, insbesondere Linux, vorgestellt, die in diesem Kapitel mit einigen geeigneten Lösungsansätzen adressiert werden.

- Programmierfehler in Anwendungen oder im Kernel sind, wie die regelmäßige Entdeckung neuer Schwachstellen beweist, nicht auszuschließen. Auch wenn Sicherheitslücken in der Regel schnell behoben werden, schützt diese Maßnahme das Betriebssystem nicht von Zero-Day-Exploits.
- Standardmäßig kontrolliert Linux den Zugriff auf Ressourcen anhand der Identität des anfragenden Users. Diese sogenannte *Discretionary Access Control*, kurz DAC, implementiert eine einfache Form von *Access Control List* (ACL). Wie in Kapitel 3 aufgezeigt, muss davon ausgegangen werden, dass ein Container von einem Angreifer kontrolliert wird. Ein solcher nicht-privilegierter Container kann beispielsweise Sicherheitslücken des Linux-Werkzeugs `ping` ausnutzen, da das Werkzeug selbst mit Root-Rechten agiert, selbst aber von jedem Benutzer im System gestartet werden kann, auch einem kompromittiertem Container [128, S.26]. Sobald der Angreifer über Root-Rechte verfügt, wird der Schutz der DAC umgangen.

Sofern praktikabel macht es demnach Sinn, das Betriebssystem präventiv zu schützen, indem der potentielle Schaden von Angriffen auf Basis des *Principle Of Least Privilege* eingedämmt wird. Diese Prinzip basiert auf der Einschränkung von Zugriffsrechten, der auch für Root-Nutzer unterliegen. Neben

diesem Prinzip, kann das Vorgehen von Docker auf ein zweites Modell, *Defense In Depth*, zurückgeführt werden. Unter diesem Ansatz werden mehrere Mechanismen kombiniert. Das Ziel hierbei ist es, die Gesamtsicherheit eines Systems zu erhöhen, indem mehrere unabhängige Sicherheitsschichten implementiert werden. Der Funktionsumfang der Schichten überschneidet sich im Fall von Docker teilweise.

Linux bietet verschiedene Möglichkeiten Mechanismen auf Basis der beiden vorgestellten Modelle zu realisieren. Diejenigen Möglichkeiten, für die Docker eine individuelle Unterstützung bietet, werden in diesem Kapitel vorgestellt. Es existieren noch weitere Sicherheitsupdates für den Linux-Kernel, wie *grsecurity* und *PAX*, allerdings erfordern jene durch ihre anwendungsunabhängige Natur keine Unterstützung seitens Docker [24]. Aus diesem Grund wird diese Art von Sicherheitserweiterung in dieser Arbeit nicht weiter betrachtet.

Die Docker-Entwickler haben in den letzten Monaten die Integrations- und Anpassungsmöglichkeiten zusätzlicher Sicherheitsmaßnahmen, v.a. MACs, stark verbessert, da die Containersicherheit für *Docker* nach eigenen Aussagen höchste Priorität hat [40][36]. Auch die Tatsache, dass sich zur Zeit die Konkurrenz *CoreOS* mit der Containerlösung *rkt* als sicherheitsfokussierte Alternative zu Docker auf dem Virtualisierungsmarkt etablieren will [98], ist für *Docker* Anreiz die Sicherheit ihrer eigenen Entwicklung nicht zu vernachlässigen. Die Veröffentlichung des großen Sicherheitsupdates für Docker mit der Version 1.10 am vierten Februar 2016 geschah ca. drei Stunden nach der Ankündigung der Version 1.0 von *rkt* seitens *CoreOS* [96][97], was eine starke Konkurrenz zwischen den beiden Parteien vermuten lässt.

Zur Veranschaulichung werden in den folgenden Abschnitten Prozesse und User, die Zugriffsanfragen starten können, als Subjekte oder Akteure bezeichnet. Ressourcen, auf die ein Zugriff erfolgen kann, werden im Gegensatz dazu als Objekte behandelt. Objekte umfassen alle jene Ressourcen, die von einem Kernel in interne Kernelobjekte aufgelöst werden können, also z.B. Dateien, Verzeichnisse, Sockets, Geräte, etc. Der Zugriff eines Subjekts auf ein Objekt wird zugelassen, sofern das Subjekt das dafür notwendige Recht, oder

gleichbedeutend Privileg, besitzt.

4.3.1 capabilities

Capabilities ist ein Feature, dass seit Kernelversion 2.2.11 in Linux integriert ist, um den traditionellen DAC-Mechanismus zu verfeinern [128, S.42]. Es existieren verschiedene Definitionen des Begriffs „Capability“. In dieser Arbeit sind unter diesem Begriff die POSIX-Capabilities gemeint, die seit Version 2.2 des Kernels in Linux fest integriert sind.

Als natives Feature werden Capabilities nicht nur von Docker-Prozessen direkt verwendet, sondern auch die beiden MACs *AppArmor* und *SELinux* machen indirekt Gebrauch von ihnen.

Capabilities haben die Absicht den traditionellen DAC-Mechanismus zu verfeinern, indem Rechte in kleine Einheiten aufgeteilt sind. Genauer ermöglicht das Feature, dass die dem Root-User mit `UID=0` zustehenden Rechte in individuelle, voneinander unabhängige Einheiten unterteilt werden. Jede privilegierte Aktion ist auf eine **capability** abgebildet. Nicht-privilegierten Subjekten ist es mithilfe von Capabilities möglich, privilegierte Operationen auszuführen, sofern sie die dafür notwendige Capability besitzen [114, S.33][128, S.39].

Insgesamt existieren aktuell 32 Capabilities, die außer ihrer Nummer - einer Zahl zwischen 0 und 31 - einen Namen tragen, der jeweils mit `CAP_` beginnt. Um z.B. einem Subjekt die Modifikation des Kernels zu verbieten, muss diesem die Capability `CAP_SYS_MODULE` entzogen sein [128, S.42].

Im Laufe der Jahre sind weitere unterschiedliche Privilegien entstanden, deren in Summe mehr als 32 Elemente umfasst. Aus diesem Grund wurden v.a. zwei Capabilities, `CAP_NET_ADMIN` und `CAP_SYS_ADMIN`, überladen, um den Privilegien nutzen zu können [128, S.40f.]. Bei diesen beiden Capabilities handelt es sich um die mächtigsten Vertreter, weswegen diese nur mit Bedacht zu gewähren sind.

Container sind unter Docker standardmäßig in der Lage 14 Capabilities zu verwenden [83]. Mit den Parametern `--cap-add` und `--cap-drop` des Docker-Befehls `run` können Capabilities zusätzlich beim Startvorgang von Containern individuell genehmigt oder verweigert werden. In Version 1.10 und neuer ist jedoch zu beachten, dass das *Seccomp*-Standardprofil manuelle Änderungen des verwendeten Capability-Sets überschreibt. Eigene Capabilities werden nur beachtet, wenn das *Seccomp*-Profil deaktiviert wurde [27]. Die Rolle von *Seccomp*, der erst kürzlich von Docker unterstützte Sicherheitsmechanismus, ist in Kapitel 4.3.3 erklärt.

4.3.2 Mandatory Access Control (MAC) und Linux Security Modules (LSMs)

Wie bereits erwähnt, erfüllt der DAC-Mechanismus nicht moderne Anforderungen an die Sicherheit in Containersystemen, da er z.B. von Superuser-Tools umgangen werden kann. Aus diesem Grund integriert Docker die beiden MAC-Mechanismen *Security-Enhanced Linux (SELinux)* und *AppArmor*, die anhand eines identitätsunabhängigen Regelwerks zusätzliche Sicherheit bieten. Unter Einbeziehung solcher Kontrollmodule kann nun z.B. auch der Zugriff eines Angreifers eingeschränkt werden, selbst wenn dieser über Sicherheitslücken in den Besitz von Root-Rechten gelangt.

Die Integration der beiden MACs geschieht modular über das *Linux Security Modules*-Framework, auch abgekürzt LSM genannt, das inzwischen standardmäßig in den Kernel eingebaut ist. Module, die in das Framework eingebettet werden, sind kombinierbar. Sicherheitsmodelle können so konsekutiv umgesetzt werden, um die bereits angesprochene *Defense In Depth* umzusetzen [135, S.3].

Die Kontrolle unter LSM geschieht, wie in Abb.11 dargestellt, in Form eines LSM-Hooks. Unter einem Hook ist ein zwischengeschaltener Aufruf gemeint, der einen *System Call* unterbricht und eine Weiterverarbeitung in ein Sicherheitsmodul anstößt. Erst nach einer Antwort eines oder mehrerer

hintereinander geschaltener Module, wird die normale Weiterausführung des *System Calls* fortgesetzt bzw. für den Fall, dass die Entscheidung des Moduls restriktiver Natur war, verweigert.

An dieser Stelle ist wichtig zu erwähnen, dass Module des LSM-Frameworks den nativen DAC-Mechanismus nicht überschreiben, sondern ergänzen. Wie in Abb.11 dargestellt, greift der LSM-Hook erst nachdem ein User einen sicherheitskritischen *System Call* ausführt hat, das hierbei angefragte Objekt aufgelöst wurde, ein Fehler-Check abgeschlossen und der Zugriff über den klassischen DAC-Mechanismus genehmigt wurde. Erst wenn der Kernel versucht auf das aufgelöste Kernelobjekt zuzugreifen, wird der Hook ausgeführt, der den Zugriff in das zugehörige LSM-Modul weiterleitet. Das Modul genehmigt oder verweigert den Zugriff anhand den ihm vorliegenden Attribute im Sicherheitskontext.

Durch diese Reihenfolge ist sichergestellt, dass die Nutzung einer LSM-Schnittstelle optional ist und unabhängig von der DAC funktioniert [34]. Deswegen können auch Anwendungen, die das LSM-Framework nicht unterstützen, weiterhin funktionieren. Außerdem ist damit die Gefahr, durch MACs neue Sicherheitslücken in das System einzuführen, ausgeschlossen.

Die Vorgehensweise unterscheidet sich damit grundlegend von der regulären Implementierung einer MAC, da letztere durch ihre obligatorische Natur normalerweise zu Beginn einer Zugriffskontrolle ausgeführt wird [135, S.3].

Der Ausführungszeitpunkt des Hooks bietet den Vorteil, dass der komplette Kontext der Zugriffsanfrage an dieser Stelle vorliegt und vollständig von einem LSM-Modul ausgewertet werden kann [135, S.2]. Neben der Kompatibilität zum DAC-Mechanismus, wird dadurch zusätzlich die Granularität der Zugriffskontrolle verbessert [134].

Bei der Verwendung von MACs ist auf den Overhead zu achten, den die einzelnen Module verursachen. Nach den durchgeführten Benchmarks in [128, S.51ff.] kommt es unter *AppArmor* und *SELinux* bei der Ausführungsgeschwindigkeit hauptsächlich zu Verlusten zwischen 0% und 11%. Spezielle Aktionen, wie z.B. das Öffnen und Schließen einer Datei unter *AppArmor*, können die

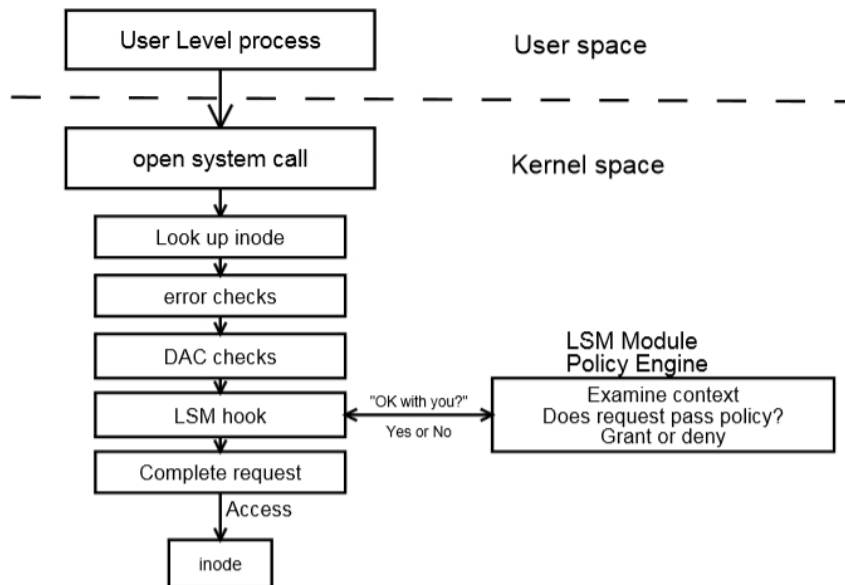


Abbildung 11: Funktionsweise von *System Call*-Hooks eines LSMs [135, S.3].

Ausführungszeit verdoppeln.

Der Sicherheitskontext kann unter dem Betriebssystem Linux für Subjekte mit dem Befehl `ps` und für Objekte mit dem Befehl `ls`, jeweils mit dem Parameter `-Z` ausgegeben werden.

In den folgenden Unterkapiteln sind die beiden MACs *AppArmor* und *SELinux* vorgestellt. Exemplarisch wird das Standardprofil von *AppArmor*, das Docker verwendet, im Detail analysiert. Ein Ausblick in die Zukunft von MACs unter Docker ist in Kapitel 7 gegeben.

4.3.2.1 AppArmor

AppArmor implementiert eine MAC und wurde als leicht konfigurierbare Alternative zu *SELinux* entwickelt. Es kommt in Linuxdistributionen wie *Debian*, *Ubuntu* und *OpenSUSE* standardmäßig zum Einsatz [6].

Auf Basis von textbasierten Profilen werden anwendungsspezifische Regeln definiert, die den Zugriff auf Objekten über Pfade im Dateisystem und sieben

kombinierbare Berechtigungstypen festlegen [93][12]. Durch eine Inkludieranweisung lassen sich mehrere Profile modular kombinieren [12].

Im Wesentlichen werden die Schutzziele über die Berechtigungstypen realisiert. Die Integrität kann direkt mit der Verwehrung des Schreibrechts (**w**) hergestellt werden, während die Vertraulichkeit auf einem Leseverbot (**r**) basiert. Dadurch, dass sich z.B. mit einem Schreibvorgang in die Datei `/proc/sysrq-trigger` das Hostsystem neustarten lässt, haben die zugewiesenen Berechtigungen auch Einfluss auf die Verfügbarkeit.

AppArmor unterstützt drei Betriebsmodi, kombinierbar sind und folgende Zwecke erfüllen [128, S.82]:

- *Audit-Modus*: Alle erlaubten Zugriffe werden protokolliert.
- *Complain-Modus*: Ein Lernmodus, bei dem das Verhalten einer Anwendungen beobachtet wird und aus diesem ein automatisch generiertes Sicherheitsprofil erstellt wird [93]. In diesem Modus werden Zugriffe, die gegen die Profilregeln verstoßen, nur aufgezeichnet und nicht unterbunden.
- *Enforce-Modus*: Die Regeln eines Profils werden erzwungen. Verstöße werden protokolliert.

Das Standardprofil `docker-default` [9], das im **Enforce-Modus** in nicht-privilegierten Containern zum Einsatz kommt [110], wird bei der Installation von Docker in die Datei `/etc/apparmor.d/docker` geschrieben. Administratoren können sich mit dem Befehl `sudo aa-status` vergewissern, ob das Standardprofil aktuell aktiv ist. Auch die zurückgegebenen Sicherheitskontexte in der ersten Spalte von Abb.... und Abb.... belegen die standardmäßige Verwendung von `docker-default`. Es existiert auch ein Profil für den Docker-Daemon [10], allerdings muss dieses manuell aktiviert werden [11].

Das *AppArmor*-Profil für Container kann mit dem `run`-Parameter `--security-opt=apparmor:PR` manuell überschrieben werden, sofern dieses in *AppArmor*, z.B. mit dem CLI-Tool `apparmor_parser` [84], zuvor importiert wurde [27].

`docker-default` wurde erst kürzlich während der Erstellung dieser Arbeit aktualisiert. Aus der Commit-Nachricht geht allerdings nicht hervor, ob damit eine Sicherheitslücke geschlossen wurde oder die Änderungen im Zuge einer Funktionsänderung von Containern entstanden sind [15]. Die aktuelle Implementierung des Profils, die lokal mit dem Konsolenbefehl `cat /etc/apparmor.d/docker` ausgegeben werden kann, sieht einige Einträge vor, die im Anhang ?? zusammengefasst sind. Im Folgenden werden die einzelnen Einträge gruppiert in der Reihenfolge des Vorkommens analysiert.

```
#include <tunables/global>
```

Diese Anweisung inkludiert einige Variablen für die weitere Verwendung. Darunter ist auch die Variable `@PROC` definiert, die in diesem Profil verwendet wird und das virtuelle Verzeichnis `/proc/` auflöst.

```
profile docker-default flags=(attach_disconnected,mediate_deleted)
...
}
```

Die erste Zeile markiert den Start des Profils `docker-default` mit zwei Flags. Das Flag `attach_disconnected` gibt an, dass Verzeichnisse außerhalb eines Namespace direkt in das Rootverzeichnis `/` eingefügt werden [7]. Die Angabe von `mediate_deleted` bewirkt eine versuchte Auflösung eines im Speicher gelöschten Objekts anhand dessen Pfad im Dateisystem [8]. Beide Flags wirken sich nicht direkt auf die Zugriffsverwaltung unter Docker aus und sind nur zur Vollständigkeit erwähnt.

```
#include <abstractions/base>
```

Hiermit werden einige Zugriffsregeln eingebunden, die von fast allen Programmen benötigt werden [128, S.100].

```
network,
```


nels.

```
deny mount,
```

Diese Zeile verbietet das Einbinden jeglicher Mountpoints.

```
deny /sys/[~f]*/** wklx,  
deny /sys/f[~s]*/** wklx,  
deny /sys/fs/[~c]*/** wklx,  
deny /sys/fs/c[~g]*/** wklx,  
deny /sys/fs/cg[~r]*/** wklx,  
deny /sys/firmware/efi/efivars/** rwklx,  
deny /sys/kernel/security/** rwklx,
```

Diese Liste untersagt, abgesehen von einem Lesezugriff, alle anderen Privilegien aus dem bereits geschilderten Rechteset für das Verzeichnis `/sys/`. Die Ausnahme bilden effektiv Dateien und Unterverzeichnisse in `/sys/fs/cgroups/`. Damit ist es Containern möglich *Control Groups*-Informationen des Hostsystems zu beziehen.

Von Dateien und Unterordnern in `/sys/firmware/efi/efivars/` und `/sys/kernel/security` darf auch nicht gelesen werden. Erstes bietet Informationen über einige Bootvariablen, letzteres gewährt Einblick in die aktuelle Konfiguration von Sicherheitsmodulen wie *AppArmor* und *SELinux* [86][57][77].

4.3.2.2 SELinux

SELinux implementiert eine feingranulare MAC, die ursprünglich von der NSA entwickelt wurde. Es wird verwendet, um mithilfe von *Type Enforcement* und *Multi-Level Security* Anforderungen an die Integrität und Vertraulichkeit zu erfüllen [88]. Der wesentliche Unterschied zu *AppArmor* ist, dass die Zugriffsverwaltung über Label erfolgt, die Subjekten und Objekten in einem Sicherheitskontext angehängt sind.

Auch unter *SELinux* beruhen die Regeln auf einem Profil, das in auch Policy genannt wird. Die Policy besteht aus Anweisungen, die konkrete Sicherheitslabel, wie sie im nächsten Abschnitt beschrieben sind, abbilden. Durch die in Abb.12 dargestellte strikte Trennung des Regelwerks und dessen Durchsetzung, lassen sich mit *SELinux* hohe Sicherheitsanforderungen erfüllen. Durch detailreiche Anpassungsmöglichkeiten steht diese MAC unter dem Ruf besonders schwer konfigurierbar zu sein, obwohl GUI-Tools wie *system-config-selinux* und die Bibliothek *libsemanage* den Umgang mit *SELinux*-Regelwerken in den letzten Jahren vereinfacht haben [108, S.62,S.67].

Die *Red Hat*-basierten Linux-Distributionen *Red Hat Enterprise Linux*, *Fedora* und *CentOS* sind in der Lage *SELinux* als zusätzlichen Schutzmechanismus zu verwenden [24].

SELinux kennt das Konzept des DACs von Ownern und Groups nicht. Der komplette Funktionsumfang von *SELinux* beruht auf einem Labeling-System, das Zugriffe individuell für Subjekte verwaltet [122]. In Zuge dessen wird jedem Subjekt zur Laufzeit und jedem Objekt im System ein Label nach dem Schema **User:Role:Type:Level** zugewiesen [19]. Die erste Komponente **User** eines Labels ist von einem Linux-Users, der mit DAC ausgewertet wird, unabhängig.

Das Label ist in die erweiternden Attribute (**xattr**) von Subjekten und Objekts geschrieben [108, S.65]. In Abb.12 ist das Label als *SC* (Security-Context) illustriert.

SELinux wertet bei einem Zugriff das Label des zugreifenden Prozesses und das Label der betroffenen Ressource anhand einem definierten Regelwerk aus und entscheidet, ob die Operation fortgesetzt werden darf [93]. Die Regeln werden in ihrer Summe auch als Policy bezeichnet.

Seit November 2015 wird Docker mit dem Release 1.9.0 mit einer standardmäßigen *SELinux*-Policy ausgestattet, die in dem *rpm*-basierten Distributionen wie *CentOS* und *Fedora* verwendet wird [36][3]. Diese Standard-Policy kann über [78] aufgerufen werden.

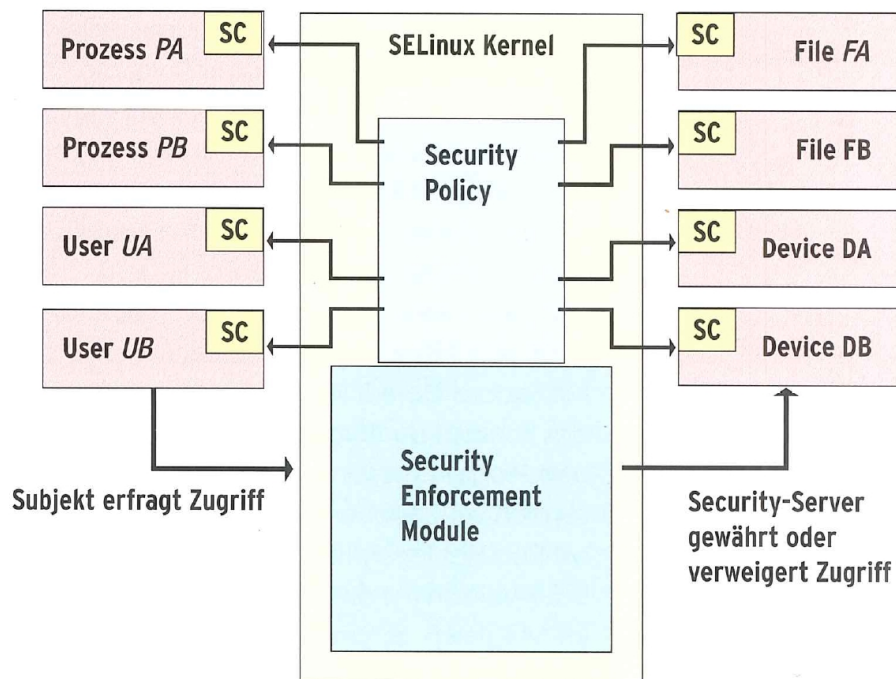


Abbildung 12: Trennung von Regelwerk und Enforcement-Modul. Zuweisung von Security-Contexts (SC) an Objekte und Subjekte [108, S.63].

Unter Docker können die vier Label-Attribute mit dem Parameter `--security-opt="label:LABEL"` für den `run`-Befehl pro Container manuell spezifiziert werden [27].

SELinux selbst stellt keine *namespaces* zur Verfügung. Deswegen kann pro Host nur eine SELinux-Policy aktiv sein, die für alle Hostcontainer angewandt wird.

Auf eine Anwendung abgestimmtes *SELinux*-Regelwerk wird mit der sicherheitskritischen Anwendung zusammen verteilt. Bei der Installation wird dann auch die anwendungsspezifische Policy in das LSM geladen, sodass der Sicherheitsmechanismus nicht manuell eingepflegt werden muss. Außerdem ist die Policy sofort bei der ersten Verwendung der Anwendung aktiv.

Die Funktionsweise zweier *SELinux*-Bausteine, *Type Enforcement* und *Multi-Level Security*, werden im Folgenden erklärt. Ein dritter Baustein, *Multi-Level Security*, wird in dieser Arbeit nicht behandelt, da dieser unter Docker keine Verwendung findet.

Type Enforcement (TE) Die wichtigste Komponente von *SELinux* ist das *Type Enforcement*. Beim *Type Enforcement* werden jedem Subjekt und jedem Objekt ein Typ zugewiesen, deren Auswertung bei jeder Zugriffsoperation zwischen Subjekt und Objekt stattfindet. Der Typ ist im Label an dritter Stelle definiert.

In der Praxis existieren für jede Anwendung eigene Typen, da jede Anwendung eigene Rechte benötigt, um ihre Funktion zu erfüllen. Jedem Docker-Prozess ist z.B. der Typ `docker_t` zugewiesen. Im *SELinux*-Regelwerk ist festgelegt, dass Prozesse eines Typs nur auf Objekte vollen Zugriff hat, die mit bestimmten Labeltypen versehen sind. Im konkreten Fall von `docker_t` umfasst diese Konfiguration ein Objektset, das u.a. aus den Typen `docker_config_t`, `docker_log_t` und `svirt_sandbox_file_t` besteht [79]. Versucht ein Docker-Prozess auf Objekte zuzugreifen, deren Typ nicht in [79] gelistet ist, wird die Operation unterbunden.

Da in *SELinux*-Umgebungen jedem Objekt im System ein Label zugewiesen ist, wird eine zuverlässige Kontrolle von Objektzugriffen über die Auswertung von Typen ermöglicht. Subjekte, die mit dem Typ `docker_t` aus der Docker-Domäne stammen, sind streng von Objekten anderer Domänen abgegrenzt und können nicht mit diesen interagieren.

Multi-Category Security (MCS) Durch das einheitliche Regelwerk für alle Docker-Prozesse, ist mit einem *Type Enforcement* gewährleistet, dass Docker nicht unbefugt auf geschützte oder unrelevante Dateien des Hosts zugreifen darf.

SELinux bietet mit dem MCS-Mechanismus jedoch auch eine Möglichkeit, Docker-Prozesse untereinander zu trennen, auch wenn sie den gleichen Typ, z.B. `docker_t` aufweisen. Dieses Sicherheitsfeature ist unter Docker auch mit *Type Enforcements* realisierbar, wenn jeder Container mit einem eigenen Typ operiert. Diese feinere Typenunterteilung wirkt sich aber auf die Komplexität der Policy aus, weswegen in der Regel der MCS-Mechanismus für dieses Sicherheitsfeature eingesetzt wird.

Der MCS-Mechanismus arbeitet mit dem letzten Teil des Labels, dem `Level`. Das Level unterteilt sich mit der Schreibweise `sensitivity[:category-set]` in eine Sensitivität, oder auch Schutzstufe genannt, und optionalen Kategorien. Für die MCS sind die Kategorien von Bedeutung. Die Schutzstufe wird ignoriert, weil sie nur unter der nicht von Docker verwendeten MLS Anwendung findet. Im Fall von Docker hat die Schutzstufe deswegen einen konstanten Wert von `s0`.

Beim Startvorgang eines Containers wird diesem eine zufällige Kategorie anhand einer Nummer zwischen 0 und 1023 zugewiesen. Diese Kategorie, z.B. `c623`, wird daraufhin auch vom Docker-Daemon auf den Inhalt containerspezifischer Verzeichnisse angewandt. Sobald während des Betriebs ein Container Zugriff auf ein Objekt fordert, wird seine Kategorie mit dem des angefragten Objekts verglichen. Stimmen diese überein, ist der MCS-Check erfolgreich und der Zugriff wird freigegeben.

Die Sicherheit von MCS unter Docker beruht auf der Annahme, dass der Docker-Daemon zuverlässig eindeutige Kategorien an die Container vergibt.

Falls ein Angreifer einen Container unter Kontrolle hat, ist es ihm durch die MCS nicht möglich, außerhalb von dem komprimiertem Container Schaden anzurichten.

Multi-Level Security (MLS) *DELETE – MLS nicht unter Docker genutzt...*

4.3.3 Seccomp

Seccomp steht für *Secure Computing Mode* und setzt einen von *Google* implementierten Mechanismus um, der den Zugriff von Prozessen auf *System Calls* einschränkt. Die Idee von *Seccomp* ist es, die Angriffsfläche des Kernels zu minimieren, indem bestimmte *System Calls* für Useranwendungen gesperrt werden. Die Gefahr, dass fehlerbehaftete oder unsichere *System Calls* genutzt werden, die die Anwendung zum fehlerfreien Betrieb nicht benötigt,

wird dadurch reduziert [93][53][123].

Seccomp ist nicht wie *SELinux* und *AppArmor* als LSM implementiert, sondern arbeitet auf Applikationslevel.

Das hat die positive Auswirkung, dass *Seccomp*-Profile auch von nicht-privilegierten Nutzern geladen werden können. Mit LSMs ist dies nicht möglich. Die Verwendung von *Seccomp* für einen Prozess bewirkt, dass jener in einen „sicheren“ Zustand übergeht, sodass er nur noch fest definierte *System Calls* ausführen kann.

Das originale *Seccomp*, auch als *mode 1* bekannt, stellt nur den Zugriff auf vier *System Calls* zur Verfügung: **read**, **write**, **exit** und **sigreturn**. Diese vier Aufrufe repräsentieren ein minimales Set an Operationen, die eine nicht vertrauenswürdige Anwendung ausführen darf [93].

Ein Update *mode 2* macht das Set an erlaubten *System Calls* mithilfe von Filtern frei konfigurierbar und führt ein *Audit Logging* ein [93][53].

Mithilfe der *Seccompn*-Anweisungen **allow**, **deny**, **trap**, **kill** und **trace** sind neben der Sperrung noch weitere Aktionen, die zur Kontrolle von *System Calls* dienen, möglich [110].

Seit Oktober 2015 ist eine *Seccomp*-Unterstützung in Planung und Entwicklung [30][67]. Diese wurde in Form eines *Seccomp*-Standardprofils sowie der Option eigene Profile einzubinden, der zum Erstellungszeitpunkt dieser Arbeit neusten Docker-Version 1.10 am 04. Februar 2016 hinzugefügt [36][76][75][110]. Das Standardprofil basiert seit Ende 2015 auf einer Whitelist (davor einer Blacklist), sprich es blockiert, abgesehen von den in dieser Liste aufgeführten Operationen, alle *System Calls* [76].

Eine aktuelle Liste der explizit erlaubten und resultierend blockierten Aufrufe ist in [75] und [76] zu finden.

Seit der Umstellung von Blacklisting auf Whitelisting wurde die *System Call*-Auflistung in einem Zeitraum von ca. fünf Wochen 47 Änderungen unterzogen. Davon sind 40 Neueinträge und sieben Löschungen zu registrieren [16]. Während sich es bei den Neueinträgen um bewusste Funktionserweiterun-

gen handeln kann, werfen sieben Löschungen den Verdacht auf, dass das *Seccomp*-Standardprofil weder als vollständig noch ausreichend getestet betrachtet werden kann.

Die offizielle Dokumentation des **run**-Befehls sieht noch keine Anpassungsmöglichkeit von *Seccomp* vor [27]. Jedoch ist an anderer Stelle im *GitHub*-Repository vermerkt, dass sich das Standardprofil mit dem Parameter **--security-opt seccomp:PROFILEPATH** überschreiben lässt [76]. Ist es nötig, Container ohne *Seccomp* zu starten, kann das mit der Option **--security-opt seccomp:unconfined** realisiert werden [110].

4.4 Docker im Vergleich zu anderen Containerlösungen

Kapitel 5

Security im Docker-Ökosystem

In diesem Kapitel werden einige Anwendungsaspekte von Docker unter einem Sicherheitskontext beleuchtet. Maßgeblich sollen weiterhin die drei definierten Sicherheitsziele zur Bewertung von Sicherheitseigenschaften dienen. Während Kapitel 4 native Sicherheitskomponenten von Docker und Linux untersucht hat, wird in den folgenden Abschnitten das Docker-Ökosystem in Betracht gezogen. Unter einem Ökosystem versteht sich hierbei die Gesamtheit aller Komponenten und Interaktionsmöglichkeiten, die im Zusammenhang mit Docker existieren. Der Fokus der Untersuchung liegt auf Anwendungsebene. Das bedeutet, dass hauptsächlich Methoden vorgestellt werden, die Docker Entwicklern und Administratoren zur Verfügung stellt, um die Arbeit mit den Docker-Komponenten aus Kapitel 2.3 sicher zu gestalten. Außerdem wird vorgestellt, wie sich die sicherheitsrelevanten Komponenten und Operationen in den letzten Monaten aus der Sicherheitsperspektive geändert haben.

Die Untersuchung sieht folgende Themen vor:

- Verbindung zwischen Docker-Client und Docker-Daemon
- Verwaltung von Images

- Betrieb von Containern
- Verwendung von Plugins
- Verwendung von 3rd-Party-Tools, wie z.B. Kubernetes

5.1 Private Registries

Docker bietet neben der Nutzung des öffentlichen Docker Hubs an, private Registries zu erstellen. Diese können dann, z.B. von einer Firewall gesichert oder von einem Load-Balancer unterstützt, in der firmeneigenen Infrastruktur oder in Rechenzentren externer Public-Cloud-Anbieter betrieben werden. Für Cloud-Anbieter stellt Docker einige Speichertreiber zur Verfügung, z.B. für Amazons S3 [74], Microsofts Azure [59], und OpenStacks Swift [65]. Bei Bedarf können eigene Speichertreiber mit der *Storage-API* implementiert werden [29]. Neben der Vertraulichkeit von Images, bieten private Registries den Vorteil, dass sich die Speicherung und Verteilung von Images an den internen und häufig durch *Continuous Integration* und *Continuous Delivery* automatisierten Softwareentwicklungsprozess anpassen lassen. Registries selbst können als Container betrieben werden [20].

Außerdem lässt sich das öffentliche Docker Hub in einer privaten Registry spiegeln. Bei dem Herunterladen von Images aus der öffentlichen Registry, kann somit auf eine externe Verbindung verzichtet werden, sofern die Spiegelung in Form einer privaten Registry im eigenen Netz existiert. Docker kann mit der Option `--registry-mirror=ADDRESS` angewiesen werden, anstelle des Docker Hubs eine Spiegelung zu verwenden [71].

Der Zugriff auf eine Registry kann z.B. über HTTPS und der Verwendung von Zertifikaten abgesichert werden [20] (vgl. Kapitel 5.3).

5.2 Verifikation und Verteilung von Images

Der Sicherheitsforscher Jonathan Rudenberg hat im Dezember 2014 drei Sicherheitsrisiken im Zusammenhang mit Dockers damaliger Verifikation und Verteilung von Images aufgedeckt [126][125]. U.a. ist es durch Verwendung des `docker pull`-Befehls möglich manipulierte Images zu beziehen, die bereits beim Entpacken auf dem lokalen System beliebige Dateien im Hostsystem überschreiben können [113]. Sowohl die Integrität von Daten als auch die Verfügbarkeit der Hosts sind durch eine solche Gefahr direkt gefährdet. Auch die Aktualität von Images, sowie die Authentizität von Personen und Organisationen, die Images veröffentlichen, kann darunter leiden, wie in Kapitel 5.2.2 zu sehen ist.

In Docker wurden seit Version 1.8 schrittweise Mechanismen implementiert, die die Verifikation einerseits und das Verteilungsmodell von Images verbessern sollen. Diese umgesetzten, teilweise sich überschneidenden Ansätze, sind im Folgenden in Aspekte der Verifikation und Aspekte der Verteilung aufgeteilt.

5.2.1 Verifikation von Images

Seit Februar 2016 mit der Veröffentlichung von Docker-Version 1.10 sind Images über deren Inhalt adressierbar. Auf Implementierungsebene bedeutet das, dass die Layer-IDs nicht wie zuvor zufällig generierte UUIDs repräsentieren, sondern als SHA256-Hashwerte, die über die Layerdaten gebildet werden, vorliegen [?, S.16]. Der SHA256-Hashalgorithmus wird derzeit als kryptographisch sicher gesehen, was zur Folge hat, dass die generierten Layer-IDs kollisions sicher sind und damit als einmalig gelten. Durch die deterministische Natur von Hashfunktionen wird gleichzeitig eine Methode implementiert, die die Integrität von Layern sicherstellt. In der Praxis kann die Korrektheit von Layerdaten validiert werden, indem ein frisch berechneter Hash eines Layers mit dem referenzierten Hasheintrag in den Image-Metadaten, dem Manifest, verglichen wird. Die referenzierten Hashwerte der

Layer sind im Manifest in Form eines Hashbaums strukturiert. Seit der Version 2 des Manifests, kann die Manifestdatei signiert werden, um auch die Integrität der Metadaten zu gewährleisten [?].

Die zuvor verwendeten UUIDs erfüllen die deterministische Eigenschaft nicht, da sie unabhängig von den Daten bei jeder Generierung auf Basis der PRNG-Implementierung in *Golang* entstehen [?][?].

5.2.2 Integration von *The Update Framework*

Die Integrität von Images spielt auch bei der Verteilung von Images über Docker-Registries eine große Rolle.

Im August 2015 wurde mit der Docker-Version 1.8 ein Paket- und Verteilungsmodell umgesetzt, das die von Rudenberg entdeckten Schwächen in der Bereitstellung von Images beheben soll [120]. Unter dem Featurenamen *Docker Content Trust* integriert Docker das Model *The Update Framework* (TUF) [48], welches Gefahrenquellen wie manipulierte Images, Replay- und MITM-Angriffe ausschließt. Die Sicherheit von TUF basiert auf der Signierung von Images, mit der anhand mehrerer kryptographischer Schlüssel die Integrität, Authentizität sowie Aktualität von Images sichergestellt wird. Die Verwendung dieses Features ist optional und kann mit der Umgebungsvariable `DOCKER_CONTENT_TRUST` gesteuert werden.

Docker Content Trust wird in Docker als Notary integriert. Der Notary implementiert das TUF in *Golang* und bietet Erstellern von Inhalten die Möglichkeit ihre Daten zu signieren. Die signierten Daten können dann über einen Notary-Server zum Download angeboten werden [35][120].

5.3 Verbindung zwischen Daemon und Clients

Wie in Kapitel 2.3.1 dargestellt, werden Anweisungen von Docker-Clients an einen Docker-Daemon übertragen, der diese über eine REST-API empfängt. Standardmäßig findet diese Kommunikation seit Version 0.5.2 über einen nicht netzwerkfähigen UNIX-Socket statt [24].

Eine Umgebung, die vorsieht Client und Daemon voneinander getrennt über ein Netzwerk zu betreiben, benötigt jedoch einen HTTP-Socket, um die Konnektivität der beiden Komponenten über das Netzwerk zu gewährleisten.

Obwohl die Netzwerksicherheit nicht Bestandteil dieser Arbeit ist, werden die Mechanismen, die Docker zur Absicherung der Kommunikation zwischen Client und Daemon unterstützt, kurz vorgestellt.

Mittels eigener Zertifikate können sich Daemon und Clients gegenseitig sicher über HTTPS authentifizieren. Unbefugte, fremde Daemons oder Clients können dadurch nicht mit einem vertrauenswürdigen Komplementär interagieren. Die Authentifizierung kann demnach uni- oder bidirektional erfolgen. Durch die sichere Kommunikation mittels HTTPS, das auf dem Protokoll TLS basiert, erfüllen die zu übermittelnden Daten die Sicherheitsziele Vertraulichkeit und Integrität.

Die entsprechende Konfiguration eines Daemons kann z.B. mit dem Befehl `docker daemon --tlsverify --tlscacert=CA.pem --tlscert=SERVER-CERT.pem --tlskey=SERVER-KEY.pem` vorgenommen werden. Analog dazu erfolgt die clientseitige Einstellung über `docker --tlsverify --tlscacert=CA.pem --tlscert=CERT.pem --tlskey=KEY.pem COMMAND`. Der Parameter `--tlsverify` gibt jeweils an, dass der Kommunikationspartner authentifiziert werden muss. Die Authentifikation geschieht über die Parameterwerte `--tlscert` und `--tlskey` des Kommunikationspartners, die zusammen die Identität dessen bekannt geben. Unter Angabe eines CA-Zertifikats mit Parameter `--tlscacert` hat die Authentifizierung nur dann Erfolg, wenn das Zertifikat des Kommunikationspartners von dieser CA ausgestellt wurde [22]. In einer Unternehmensin-

frastruktur kann so die Kommunikation durch eine unternehmenseigene CA weiter restriktiviert werden. Eine detailreichere Beschreibung der verschiedenen Betriebsmodi ist unter [22] gegeben.

Über die Umgebungsvariable `DOCKER_TLS_VERIFY` sowie der Speicherung der notwendigen Zertifikate und Schlüssel unter `.docker/` im Homeverzeichnis, kann die Konfiguration der Authentifizierung einmalig für die zukünftige Kommunikation vorgenommen werden [22].

5.4 Docker Plugins

Seit Juni 2015 unternahmen die Entwickler von Docker Anstrengungen, um optionale Komponenten von Docker in eine eigene Plugin-Infrastruktur zu integrieren, in der Plugins modular aktiviert und deaktiviert werden können [36][112]. Plugins werden von einem Docker-Daemon genutzt und erweitern dessen Fähigkeiten. Neben den ersten Plugins für diverse Netzwerkfunktionen, z.B. *Weave*, und der Einbindung von Datenträgern, z.B. *Flocker*, fand im Frühjahr 2016 mit Docker-Version 1.10 auch ein ursprünglich von *Twistlock*[49] entwickeltes Authorisierungs-Plugin Einzug in Docker, das in diesem Kapitel zur Vereinfachung auch als *AuthZ* bezeichnet wird [99][112][115].

Ergänzend dazu war auch ein Authentifizierungs-Plugin *AuthN* geplant, das Nutzer vor deren Authorisierung durch *AuthZ*, authentifiziert [133]. Am 23. Februar 2016 hat jedoch ein Docker-Mitarbeiter bekannt gegeben, dass die Integration von *AuthN* eingestellt wird. Grund hierfür ist, dass die Authentifizierung - nach der Meinung einiger Docker-Entwickler - leicht außerhalb des Daemons stattfinden kann, z.B. mit dem Authentifizierungsdienst Kerberos[47] [129][104].

Das Sicherheitsplugin *AuthZ* hat zum Ziel ein Framework bereitzustellen, über das es Administratoren möglich ist, eine nutzer- und rollenbasierte Sicherheitspolitik umzusetzen. Diese umfasst Regeln, die die Benutzung des Docker-Daemons betreffen. Nach der ursprünglichen Implementierung von *Twistlock* sind die Regeln in eine JSON-Struktur gefasst [116]. Ohne ein

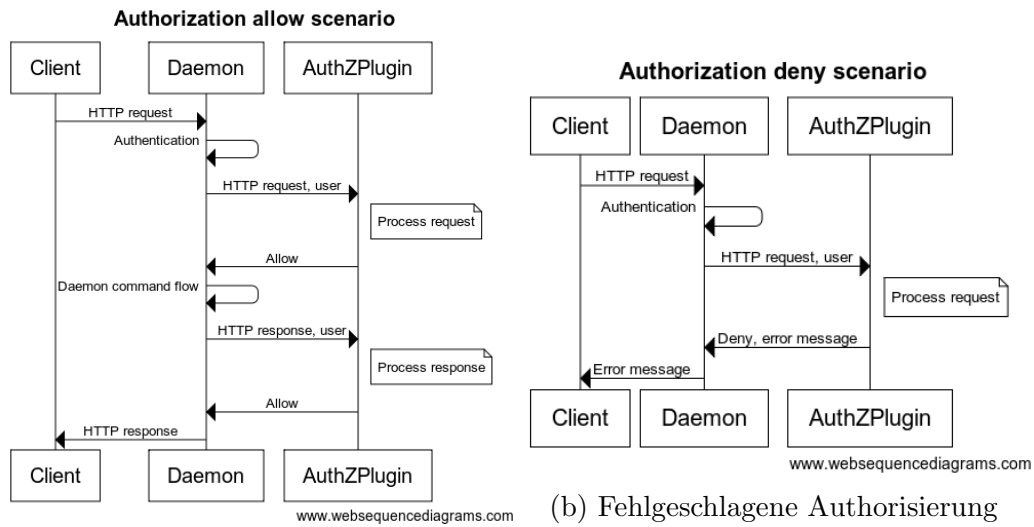
solches Plugin ist jedem Nutzer, der den Docker-Daemon ausführen kann, die komplette Kontrolle über das Docker-System gegeben. V.a. in Unternehmen macht es aber Sinn, verschiedenen Nutzer im Rahmen eines RBAC-Sicherheitsmodells eine bestimmte Rolle zuzuweisen, die deren Rechte definiert. Ein einfacher Anwendungsfall könnte folgende Regeln beinhalten [115]:

- User in der Gruppe *Operations* dürfen nur Container starten und stoppen. Sie sollen nur `docker run CONTAINER` und `docker rm CONTAINER` ausführen können.
- User in der Gruppe *Audit* dürfen nur Informationen von Images und Containern abfragen. Sie sollen nur `docker inspect IMAGE|CONTAINER` ausführen können.
- User *Admin* darf jede Operation `docker . . .` über den Daemon ausführen.

Aus Sicht der Architektur funktioniert die Authorisierung, wie sie in Abb.13 illustriert ist. Die Anfragen von lokalen oder entfernten Clients werden, wie in Kapitel 2.3.1 beschrieben, an einen Daemon geschickt. Dieser führt nun nicht die Befehle der Clients umgehend aus, sondern leitet die Anfrage an das *AuthZ*-Framework weiter. Genauer erhält *AuthZ* einen Nutzerkontext und einen Befehlskontext, die, z.B. anhand der zuvor definierten Regeln, ausgewertet werden. Anhand der dem Plugin vorliegenden Parameter entscheidet es, ob der Nutzer berechtigt ist, den angefragten Befehl auszuführen. Falls das nicht der Fall ist, wird eine Fehlermeldung über den Daemon an den Client gesendet (vgl. Abb.13b). Falls die Anfrage genehmigt wurde, führt der Daemon den darin enthaltenen Befehl aus, und kontaktiert das Plugin ein zweites Mal mit dem Ergebnis des ausgeführten Befehls (vgl. Abb.13a). *AuthZ* hat hierbei die Gelegenheit, die Antwort, bevor sie zum Client gesendet wird, zu modifizieren [130][115].

Die Formate von Anfragen und Antworten, die über das HTTP-Protokoll ausgetauscht werden, sind in [95] spezifiziert.

Mehrere Sicherheitsmodule können konsekutiv ausgeführt werden, sodass jede clientseitige Interaktion mit dem Daemon, die Ausführung mehrerer *AuthZ*



(a) Erfolgreiche Autorisierung

(b) Fehlgeschlagene Autorisierung

Abbildung 13: Ablaufdiagramm einer Befehlsausführung mit dem Autorisierungs-Plugins *AuthZ* von Docker

thZ-Implementierungen zur Folge hat.

Mit folgender Syntax können Autorisierungs-Plugins für den Daemon aktiviert werden [95]:

```
docker daemon --authorization-plugin=PLUGIN1 \
               [--authorization-plugin=PLUGIN2] [...]
```

Da es sich bei diesen sicherheitsrelevanten Plugins um ein sehr neues Feature von Docker handelt, war die offizielle Dokumentation zum Zeitpunkt der Erstellung dieser Arbeit nicht vollständig. V.a. die fehlende Spezifikation des Formats eines *AuthZ*-Regelwerks im Docker-Repository lässt vermuten, dass eine vollständige Einführung von *AuthZ* erst im Rahmen zukünftiger Docker-Releases stattfindet.

5.5 Security Policies und Open Source

5.6 Security Best-Practises

5.6.1 Datencontainer

5.6.2 --priviledged Container

5.7 Tools

5.7.1 Docker Swarm und Docker Compose

orchestration, management fokus, sicherheitsrelevant?

5.7.2 Kubernetes

orchestration, management fokus, sicherheitsrelevant?

Ein Hauptfeature von Containern ist deren flexibler Einsatz in Anwendungsclustern, die eine Multi-Tier-Anwendung / Multi-Tenant-Architektur abbilden.

Im Juni 2014 hat Google das Open-Source Tool *Kubernetes* angekündigt, das Cluster mit Docker-Containern verwalten soll. Laut Google ist Kubernetes die Entkopplung von Anwendungscontainern von Details des Hosts. Soll in Datencentern die Arbeit mit Containern vereinfachen.

Neben einigen Startups, haben sich *Google, Microsoft, VMware, IBM* und *Red Hat* als *Kubernetes*-Unterstützer geäußert.

5.7.3 Vagrant, Puppet, Chef

orchestration, management fokus, sicherheitsrelevant?, relevant?

5.7.4 Nautilus Project ?

5.7.5 github.com/cloudimmunity/docker-slim

security focus. Noch relevant?

5.7.6 Vagrant

5.8 Networking

Portmapping

bridge-Netzwerk

overlay-Netzwerk

(

5.8.1 DNS

)

Kapitel 6

Docker in Unternehmen/Cloud- Infrastrukturen

Kapitel 7

Fazit

Spekulation in der Industrie ist, dass sich Organisationen und Unternehmen zusammenschließen und sich auf eine neue, universale Lösung einigen, die die heutigen Fähigkeiten der sich ergänzenden Technologien Docker und Kubernetes, abdeckt [101, S.4].

Glossar

Best-Practice Eine bestimmte, ideale Vorgehensweise für den Umgang mit einer Sache, die zu einem erwünschten Zustand, z.B. der Erfüllung eines Standards, beiträgt. Im Fall von Docker kann es eine Best-Practice sein, Images zu signieren um deren Integrität zu gewährleisten. 4

Build Ein Erstellungsprozess, bei dem Quellcode in ein Objektcode bzw. direkt in ein fertiges Programm automatisch konvertiert wird. 18

Cloud Eine entfernte Rechnerinfrastruktur, die Dienste (Anwendungen, Plattformen, etc.) zur Nutzung bereitstellt.

- Private Cloud: Dienste werden aus Gründen der Sicherheit oder des Datenschutzes nur firmenintern für eigenen Mitarbeiter angeboten.
- Public Cloud: Dienste sind öffentlich nutzbar.
- Hybrid Cloud: Mischform aus einer privaten und öffentlichen Cloud. Manche Dienste werden nur firmenintern verwendet, andere auch von außerhalb des Firmennetzes.

[90] . 1, 10

Denial of Service *gescheite Quelle. Buch hier am besten..* 13

DevOps DevOps-Teams sind sowohl für die Entwicklung (*Dev* = Development) eines Produkts als auch den Betrieb (*Ops* = Operations) dessen verantwortlich. Durch die gemeinsame Ergebnisverantwortung fällt der

Overhead einer Übergabe, zwischen ansonsten getrennten Teams, weg [127]. 14

Kernelobjekt Datenstrukturen im Kernel, die verschiedene Ressourcen abbildet und von LSMs ausgewertet werden kann [64]. 40, 43

Multi-Tenant-Service Eine Serveranwendungen, die mehrere Nutzer gleichzeitig verwenden. Jeder Nutzer kann nur auf seine eigenen Daten zugreifen und interferiert nicht mit anderen Nutzern. Auf dem Server kann die Anwendung, die dieses Prinzip umsetzt, in einer Instanz (ohne Redundanz) laufen [60]. 2, 37

weiche und harte Limits Das weiche Limit dient als Richtwert zum Ausmaß einer Ressourcennutzung. Das harte Limit stellt den Maximalwert dar. Das weiche Limit ist immer kleiner als das harte Limit. In der Implementierung in Solaris, startet bei Überschreitung des weichen Limits ein Timer. Wenn Timer eine bestimmte Zeit überschreitet, wird weiches Limit kurzzeitig wie das harte Limit erzwungen [81]. 37

Abkürzungsverzeichnis

- ACL** Access Control List. 39
- API** Application Programming Interface. 15, 18
- ARP** Address Resolution Protocol. 35
- cgroups** Control Groups. 37–39
- CLI** Command-Line Interface. 45
- COW** Copy-On-Write. 33
- CPU** Central Processing Unit. 1, 30, 37, 38
- DAC** Discretionary Access Control. 39, 49
- DoS** Denial of Service. 13, 37, *Glossary*: Denial of Service
- HDD** Hard Disk Drive. 38
- HPC** High Performance Computing. 11
- HTTP** Hypertext Transfer Protocol. 18
- HTTPS** Hypertext Transfer Protocol Secure. 18, 56
- I/O** Input and Output. 38
- IPC** Inter Process Communication. 33, 34
- IT** Informationstechnik. 3, 5, 15

JSON JavaScript Object Notation. 19

LSM Linux Security Modules. 42

MAC Media Access Control (Netzwerkkommunikation). 35

MAC Mandatory Access Control (nicht Netzwerkkommunikation). 42

MLS Multi-Level Security. 52

OCF Open Container Format. 19

OCP Open Container Project. 19

OS Operating System. 10, 11

PID Process ID, Process Identifier. 30, 31, 34

REST Representational State Transfer. 15

rlimits Resource Limits. 37, 38

SELinux Security-Enhanced Linux. 48

SSD Solid State Drive. 38

UI User Interface. 23

USB Universal Serial Bus. 38

UTS UNIX Time Sharing. 34

VM Virtual Machine. 5, 6, 10, 11

Literaturverzeichnis

- [1] About docker. über Website <https://www.docker.com/company> , aufgerufen am 18.01.2016.
- [2] Add disk quota support for btrfs #19651. über Website <https://github.com/docker/docker/pull/19651> , aufgerufen am 28.01.2016.
- [3] Add docker selinux policy for rpm #15832. über Website <https://github.com/docker/docker/pull/15832> , aufgerufen am 03.02.2016.
- [4] Add quota support for storage backends #3804. über Website <https://github.com/docker/docker/issues/3804> , aufgerufen am 28.01.2016.
- [5] Amazon web services. über Website <https://aws.amazon.com/de/> , aufgerufen am 14.01.2016.
- [6] Apparmor. über Website <https://wiki.ubuntuusers.de/AppArmor/> , aufgerufen am 05.02.2016.
- [7] Apparmor core policy reference. über Website http://wiki.apparmor.net/index.php/AppArmor_Core_Policy_Reference , aufgerufen am 18.02.2016.
- [8] Apparmor faq. über Website http://wiki.apparmor.net/index.php/FAQ#Failed_name_lookup_-_deleted_entry , aufgerufen am 18.02.2016.

- [9] Apparmor profile template for containers. über Website <https://github.com/docker/docker/tree/master/profiles/apparmor> , aufgerufen am 09.02.2016.
- [10] Apparmor profile template for the daemon. über Website <https://github.com/docker/docker/blob/master/contrib/apparmor/template.go> , aufgerufen am 03.02.2016.
- [11] Apparmor security profiles for docker. über Website <https://github.com/docker/docker/blob/master/docs/security/apparmor.md> , aufgerufen am 05.02.2016.
- [12] Apparmor wiki - quickprofilelanguage. über Website <http://wiki.apparmor.net/index.php/QuickProfileLanguage> , aufgerufen am 05.02.2016.
- [13] Cgroup unified hierarchy - documentation/cgroups/unified-hierarchy.txt. über Website <https://lwn.net/Articles/601923/> , aufgerufen am 27.01.2016.
- [14] Chapter 1. introduction to control groups (cgroups). über Website https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html , aufgerufen am 27.01.2016.
- [15] Commit - fix proc regex. über Website <https://github.com/docker/docker/commit/2b4f64e59018c21aacbf311d5c774dd5521b5352> , aufgerufen am 09.02.2016.
- [16] Commit history - seccomp default profile. über Website https://github.com/docker/docker/commits/37d35f3c280dc27a00f2baa16431d807b24f8b92/daemon/execdriver/native/seccomp_default.go , aufgerufen am 09.02.2016.
- [17] Docker 0.9: Introducing execution drivers and libcontainer. über Website <https://blog.docker.com/2014/03/>

- `docker-0-9-introducing-execution-drivers-and-libcontainer/`
, aufgerufen am 21.01.2016.
- [18] Docker and broad industry coalition unite to create open container project. über Website <http://blog.docker.com/2015/06/open-container-project-foundation/> , aufgerufen am 21.01.2016.
- [19] Docker and selinux. über Website <http://www.projectatomic.io/docs/docker-and-selinux/> , aufgerufen am 05.02.2016.
- [20] Docker docs - registry. über Website <https://docs.docker.com/registry/> , aufgerufen am 18.01.2016.
- [21] Docker docs - understanding the architecture. über Website <https://docs.docker.com/engine/introduction/understanding-docker/> , aufgerufen am 14.01.2016.
- [22] Docker documentation - protect the docker daemon socket. über Website <https://docs.docker.com/engine/security/https/> , aufgerufen am 24.02.2016.
- [23] Docker documentation - runtime metrics. über Website <https://docs.docker.com/engine/articles/runmetrics/> , aufgerufen am 27.01.2016.
- [24] Docker documentation - security. über Website <https://docs.docker.com/engine/security/security/> , aufgerufen am 24.02.2016.
- [25] Docker documentation für den befehl `docker images`. über Website <https://docs.docker.com/engine/reference/commandline/images/> , aufgerufen am 21.01.2016.
- [26] Docker documentation für den befehl `docker pull`. über Website <https://docs.docker.com/engine/reference/commandline/pull/> , aufgerufen am 21.01.2016.

- [27] Docker documentation für den befehl `docker run`. über Website <https://docs.docker.com/engine/reference/run/> , aufgerufen am 18.02.2016.
- [28] Docker hub - explore. über Website <https://hub.docker.com/explore/> , aufgerufen am 15.01.2016.
- [29] Docker registry storage driver. über Website <https://docs.docker.com/registry/storagedrivers/> , aufgerufen am 24.02.2016.
- [30] Docker security profiles (seccomp, apparmor, etc) #17142. über Website <https://github.com/docker/docker/issues/17142#issuecomment-148974642> , aufgerufen am 05.02.2016.
- [31] *FreeBSD* einföhrung in *Jails*. über Website https://www.freebsd.org/doc/de_DE.ISO8859-1/books/handbook/jails-intro.html , aufgerufen am 18.01.2016.
- [32] Fixing control groups. über Website <https://lwn.net/Articles/484251/> , aufgerufen am 27.01.2016.
- [33] FreeBSD - hierarchical resource limits. über Website https://wiki.freebsd.org/Hierarchical_Resource_Limits , aufgerufen am 27.01.2016.
- [34] Getting started with multi-category security (mcs). über Website https://www.centos.org/docs/5/html/Deployment_Guide-en-US/sec-mcs-getstarted.html , aufgerufen am 02.02.2016.
- [35] Github repository - docker notary. über Website <https://github.com/docker/notary> , aufgerufen am 24.02.2016.
- [36] Github repository changelog von docker. über Website <https://github.com/docker/docker/blob/master/CHANGELOG.md> , aufgerufen am 05.02.2016.
- [37] Github repository der cgroups-implementierung von runc. über Website <https://github.com/opencontainers/runc/tree/master/libcontainer/cgroups/fs> , aufgerufen am 27.01.2016.

- [38] Github repository der docker engine. über Website <https://github.com/docker/docker> , aufgerufen am 11.01.2016.
- [39] Github repository glossar von docker. über Website <https://github.com/docker/distribution/blob/master/docs/glossary.md> , aufgerufen am 21.01.2016.
- [40] Github repository roadmap von docker. über Website <https://github.com/docker/docker/blob/master/ROADMAP.md> , aufgerufen am 05.02.2016.
- [41] Github repository von *runC*. über Website <https://github.com/opencontainers/runc> , aufgerufen am 21.01.2016.
- [42] Google trends der suchbegriffe *Docker*, *Virtualization* und *LXC*. über Website <https://www.google.de/trends/explore#q=docker%2Cvirtualization%2Clxc> , aufgerufen am 19.01.2016.
- [43] Homepage des kvm hypervisors und virtualisierungslösung. über Website http://www.linux-kvm.org/page/Main_Page , aufgerufen am 18.01.2016.
- [44] Homepage des vmware esxi hypervisors. über Website <https://www.vmware.com/de/products/esxi-and-esx/overview> , aufgerufen am 18.01.2016.
- [45] Homepage des xen hypervisors. über Website <http://www.xenproject.org/> , aufgerufen am 18.01.2016.
- [46] Homepage *Solaris* betriebssystem. über Website <http://www.oracle.com/de/products/servers-storage/solaris/solaris11/overview/index.html> , aufgerufen am 18.01.2016.
- [47] Homepage kerberos. über Website <http://web.mit.edu/kerberos/> , aufgerufen am 26.02.2016.
- [48] Homepage the update framework. über Website <https://theupdateframework.github.io/> , aufgerufen am 24.02.2016.

- [49] Homepage twistlock. über Website <https://www.twistlock.com/> , aufgerufen am 26.02.2016.
- [50] Homepage von *runC*. über Website <https://runc.io/> , aufgerufen am 21.01.2016.
- [51] Imagelayers of three different docker images. über Website <https://imagelayers.io/?images=redis:3.0.6,nginx:1.9.9,centos:centos7.2.1511> , aufgerufen am 21.01.2016.
- [52] Introducing runc: a lightweight universal container runtime. über Website <http://blog.docker.com/2015/06/runc/> , aufgerufen am 21.01.2016.
- [53] Kernel documentation: Secure computing with filters. über Website http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/prctl/seccomp_filter.txt , aufgerufen am 01.02.2016.
- [54] Linux manual page chroot. über Website https://www.freebsd.org/doc/de_DE.ISO8859-1/books/handbook/jails-intro.html , aufgerufen am 18.01.2016.
- [55] Linux programmer's manual - namespaces(7). über Website <http://man7.org/linux/man-pages/man7/namespaces.7.html> , aufgerufen am 28.01.2016.
- [56] Linux programmer's manual - user_namespaces(7). über Website http://man7.org/linux/man-pages/man7/user_namespaces.7.html , aufgerufen am 28.01.2016.
- [57] Low level interfaces to the apparmor kernel module - securityfs. über Website http://wiki.apparmor.net/index.php/Kernel_interfaces#securityfs_-_2Fsys.2Fkernel.2Fsecurity.2Fapparmor , aufgerufen am 18.02.2016.
- [58] Magic sysrq. über Website https://wiki.ubuntuusers.de/Magic_SysRQ/ , aufgerufen am 18.02.2016.

- [59] Microsoft azure storage driver. über Website <https://docs.docker.com/registry/storage-drivers/azure/> , aufgerufen am 24.02.2016.
- [60] Multi-tenant data architecture. über Website <https://msdn.microsoft.com/en-us/library/aa479086.aspx> , aufgerufen am 19.01.2016.
- [61] Offizielle dockerfile dokumentation. über Website <https://docs.docker.com/engine/reference/builder/#expose> , aufgerufen am 22.01.2016.
- [62] Offizieller twitter-account des docker-gründers, solomon hykes. über Website <https://twitter.com/solomonstre> , aufgerufen am 18.01.2016.
- [63] Offizielles repository des webserverns nginx. über Website https://hub.docker.com/_/nginx/ , aufgerufen am 11.01.2016.
- [64] Opaque security fields. über Website https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright_html/node6.html#subsec:opaque , aufgerufen am 05.02.2016.
- [65] Openstack swift storage driver. über Website <https://docs.docker.com/registry/storage-drivers/swift/> , aufgerufen am 24.02.2016.
- [66] Phase 1 implementation of user namespaces as a remapped container root #12648. über Website <https://github.com/docker/docker/pull/12648> , aufgerufen am 28.01.2016.
- [67] Phase 1: Initial seccomp support #17989. über Website <https://github.com/docker/docker/pull/17989> , aufgerufen am 05.02.2016.
- [68] Proposal: Support for user namespaces #7906. über Website <https://github.com/docker/docker/issues/7906> , aufgerufen am 28.01.2016.
- [69] Redhat enterprise linux reference guide - /proc/sysrq-trigger. über Website <https://access.redhat.com/documentation/>

- en-US/Red_Hat_Enterprise_Linux/4/html/Reference_Guide/s2-proc-sysrq-trigger.html , aufgerufen am 18.02.2016.
- [70] Redhat enterprise linux documentation - chapter 7. setting shared memory. über Website https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Tuning_and_Optimizing_Red_Hat_Enterprise_Linux_for_Oracle_9i_and_10g_Databases/chap-Oracle_9i_and_10g_Tuning_Guide-Setting_Shared_Memory.html , aufgerufen am 18.02.2016.
 - [71] Registry as a pull through cache. über Website <https://docs.docker.com/registry/mirror/> , aufgerufen am 23.02.2016.
 - [72] Release notes von *FreeBSD V.4* und *Jails*. über Website <https://www.freebsd.org/releases/4.0R/notes.html> , aufgerufen am 19.01.2016.
 - [73] Release notes von *Solaris 10*. über Website <https://docs.oracle.com/cd/E19253-01/pdf/817-0552.pdf> , aufgerufen am 19.01.2016.
 - [74] S3 storage driver. über Website <https://docs.docker.com/registry/storage-drivers/s3/> , aufgerufen am 24.02.2016.
 - [75] Seccomp default profile. über Website <https://github.com/docker/docker/blob/master/profiles/seccomp/default.json> , aufgerufen am 18.02.2016.
 - [76] Seccomp security profiles for docker. über Website <https://github.com/docker/docker/blob/master/docs/security/seccomp.md> , aufgerufen am 05.02.2016.
 - [77] securityfs. über Website <https://lwn.net/Articles/153366/> , aufgerufen am 18.02.2016.
 - [78] Selinux default policy profile. über Website <https://github.com/docker/docker/tree/master/contrib/docker-engine-selinux> , aufgerufen am 03.02.2016.

- [79] Selinux default policy profile - docker.te. über Website <https://github.com/docker/docker/blob/master/contrib/docker-engine-selinux/docker.te> , aufgerufen am 17.02.2016.
- [80] Slides of keynote at dockercon in san francisco - day 2. über Website [de.slideshare.net/Docker/dockercon-15-keynote-day-2/16](https://www.slideshare.net/Docker/dockercon-15-keynote-day-2/16) , aufgerufen am 11.01.2016.
- [81] Soft limits and hard limits. über Website <https://docs.oracle.com/cd/E19455-01/805-7229/sysresquotas-1/index.html> , aufgerufen am 28.01.2016.
- [82] Softlayer benchmark, data sheet. über Website https://voltdb.com/sites/default/files/voltdb_softlayer_benchmark_0.pdf , aufgerufen am 14.01.2016.
- [83] Standard-capabilities von docker. über Website https://github.com/docker/docker/blob/master/daemon/execdriver/native/template/default_template_linux.go , aufgerufen am 17.02.2016.
- [84] Ubuntu manpage: apparmor_parser - loads apparmor profiles into the kernel. über Website http://manpages.ubuntu.com/manpages/raring/man8/apparmor_parser.8.html , aufgerufen am 05.02.2016.
- [85] The unified control group hierarchy in 3.16. über Website <https://lwn.net/Articles/601840/> , aufgerufen am 27.01.2016.
- [86] Unified extensible firmware interface. über Website https://wiki.archlinux.org/index.php/Unified_Extensible_Firmware_Interface , aufgerufen am 18.02.2016.
- [87] User namespaces - phase 1 #15187. über Website <https://github.com/docker/docker/issues/15187> , aufgerufen am 28.01.2016.
- [88] Virtualization security guide - chapter 4. svirt. über Website https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Virtualization_Security_

Guide/chap-Virtualization_Security_Guide-sVirt.html#sect-Virtualization_Security_Guide-sVirt-Introduction ,
aufgerufen am 01.02.2016.

- [89] Voltdb homepage. über Website <https://voltdb.com/> , aufgerufen am 18.01.2016.
- [90] Was bedeutet public, private und hybrid cloud? über Website <http://www.cloud.fraunhofer.de/de/faq/publicprivatehybrid.html> , aufgerufen am 19.01.2016.
- [91] Überblick hyper-v hypervisor von microsoft. über Website <https://technet.microsoft.com/library/hh831531.aspx> , aufgerufen am 18.01.2016.
- [92] Übersicht zu *Solaris Zones*. über Website https://docs.oracle.com/cd/E24841_01/html/E24034/gavhc.html , aufgerufen am 18.01.2016.
- [93] Overview of linux kernel security features. über Website <https://www.linux.com/learn/docs/727873-overview-of-linux-kernel-security-features/> , aufgerufen am 01.02.2016, July 2013.
- [94] Google code archive - go issue #8447. über Website <https://code.google.com/archive/p/go/issues/8447> , aufgerufen am 28.01.2016, 2014.
- [95] Extended documentation: Create an authorization plugin. über Website <https://github.com/docker/docker/blob/e310d070f498a2ac494c6d3fde0ec5d6e4479e14/docs/extend/authorization.md> , aufgerufen am 26.02.2016, January 2016.
- [96] Hacker news - docker 1.10.0 is out. über Website <https://news.ycombinator.com/item?id=11037543> , aufgerufen am 05.02.2016, February 2016. Aufruf am 05.02.2016 um 15:04 Uhr. Eintrag erstellt '16 hours ago'.

- [97] Hacker news - the security-minded container engine by coreos: rkt hits 1.0. über Website <https://news.ycombinator.com/item?id=11035955> , aufgerufen am 05.02.2016, February 2016. Aufruf am 05.02.2016 um 15:04 Uhr. Eintrag erstellt '19 hours ago'.
- [98] The security-minded container engine by coreos: rkt hits 1.0. über Website <https://coreos.com/blog/rkt-hits-1.0.html> , aufgerufen am 05.02.2016, February 2016.
- [99] Understanding engine plugins. über Website <https://github.com/docker/docker/blob/cc085be7cc19d2d1aed39c243b6990a7d04ee639/docs/extend/plugins.md> , aufgerufen am 26.02.2016, 2016.
- [100] Charles Anderson. Docker. *IEEE Software*, 2015.
- [101] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, September 2014.
- [102] Sukadev Bhattiprolu, Eric W. Biederman, Serge Halryn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. Technical report, IBM and Arastra, July 2008.
- [103] Thanh Bui. Analysis of docker security. Technical report, Aalto University School of Science, January 2015.
- [104] David Calavera. Comment on wont-fix status of *AuthN* in issue: Add authentication to the docker daemon #18514. über Website <https://github.com/docker/docker/pull/18514#issuecomment-187942565> , aufgerufen am 26.02.2016, February 2016.
- [105] Docker. Introduction to docker security. über Website https://www.docker.com/sites/default/files/WP_Intro%20to%20container%20security_03.20.2015%20%281%29.pdf , aufgerufen am 18.01.2016, March 2015.

- [106] Rajdeep Duo, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. *IEEE International Conference on Cloud Engineering*, 2014.
- [107] Phil Estes. Rooting out root: User namespaces in docker. über Website http://events.linuxfoundation.org/sites/events/files/slides/User%20Namespaces%20-%20ContainerCon%202015%20-%202016-9-final_0.pdf , aufgerufen am 28.01.2016, 2015.
- [108] Stefan Fischer et al, editor. *Security - IT-Sicherheit unter Linux von A bis Z*. Linux Magazine, 2008.
- [109] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. Ibm research report - an updated performance comparison of virtual machines and linux containers. Technical report, IBM Research Divison - Austin Research Laboratory, July 2014.
- [110] Jessie Frazelle. Docker engine 1.10 security improvements. über Website <http://blog.docker.com/2016/02/docker-engine-1-10-security/> , aufgerufen am 05.02.2016, February 2016.
- [111] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Katalog B 3.304 Virtualisierung*, 2011.
- [112] Adam Herzog. Extending docker with plugins. über Website <https://blog.docker.com/2015/06/extending-docker-with-plugins/> , aufgerufen am 26.02.2016, June 2015.
- [113] Trevor Jay. Before you initiate a docker pull: über Website <https://securityblog.redhat.com/2014/12/18/before-you-initiate-a-docker-pull/> , aufgerufen am 24.02.2016, December 2014.
- [114] Michael Kerrisk. *The Linux Programming Interface - A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

- [115] Liron Levin. Docker authz plugins: Twistlock's contribution to the docker community. über Website <https://www.twistlock.com/2016/02/18/docker-authz-plugins-twistlocks-contribution-to-the-docker-community/>, aufgerufen am 26.02.2016, February 2016.
- [116] Liron Levin. Json-struktur eines *AuthZ*-regelwerks. über Website <https://github.com/twistlock/authz/blob/2b67bbbfab9bbc1579ade722dc84c1e3c3440fbb/authz/policy.json>, aufgerufen am 26.02.2016, January 2016.
- [117] Peter Mandl. *Grundkurs Betriebssysteme*. Springer, 4 edition, 2014.
- [118] Rory McCune. Docker 1.10 notes - user namespaces. über Website <https://raesene.github.io/blog/2016/02/04/Docker-User-Namespaces/>, aufgerufen am 05.02.2016, January 2016.
- [119] Mike Meyers and Shon Harris. *CISSP - Certified Information Systems Security Professional*. Springer, 3 edition, 2009.
- [120] Diogo Mónica. Introducing docker content trust. über Website <https://blog.docker.com/2015/08/content-trust-docker-1-8/>, aufgerufen am 24.02.2016, August 2015.
- [121] Arnaud Porterie. Introducing the technical preview of docker engine for windows server 2016. über Website <https://blog.docker.com/2015/08/tp-docker-engine-windows-server-2016/>, aufgerufen am 22.01.2016, 2015.
- [122] Daniel J. Walsh (RedHat). Your visual how-to guild for selinux policy enforcement. über Website <https://opensource.com/business/13/11/selinux-policy-guide>, aufgerufen am 01.02.2016, November 2013.
- [123] Daniel J. Walsh (RedHat). Docker security in the future. über Website <https://opensource.com/business/15/3/docker-security-future>, aufgerufen am 05.02.2016, March 2015.

- [124] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of os-level virtualization technologies. Technical report, Intel OTC Finland, Ericsson Finland, Univerisity of Helsinki, Aalto Univeristy Finland, July 2014.
- [125] Jonathan Rudenberg. Docker image insecurity. über Website <https://titanous.com/posts/docker-insecurity> , aufgerufen am 24.02.2016, December 2014.
- [126] Jonathan Rudenberg. Tarsum insecurity #9719. über Website <https://github.com/docker/docker/issues/9719> , aufgerufen am 24.02.2016, December 2014.
- [127] Jürgen Rühling. Devops in unternehmen etablieren - ein ziel, ein team, gemeinsamer erfolg. über Website <http://www.heise.de/developer/artikel/DevOps-in-Unternehmen-etablieren-2061738.html> , aufgerufen am 18.01.2016, December 2013.
- [128] Ralf Spenneberg. *SELinux & AppArmor*. Addison-Wesley, 1 edition, 2008.
- [129] Dima Stopel. Adding kerberos support to docker #13697. über Website <https://github.com/docker/docker/issues/13697#issue-84531039> , aufgerufen am 26.02.2016, June 2015.
- [130] Dima Stopel. User access control in docker daemon #14674. über Website <https://github.com/docker/docker/issues/14674#issue-95409105> , aufgerufen am 26.02.2016, July 2015.
- [131] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 3 edition, 2009.
- [132] James Turnbull. *The Docker Book*. 1.2.0 edition, September 2014.
- [133] Daniel J. Walsh and Matthew Heon. Docker access control. über Website <https://github.com/rhatdan/docker-rbac> , aufgerufen am 26.02.2016.

- [134] Chris Wright. Lsm design: Mediate access to kernel objects. über Website https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright_html/node3.html , aufgerufen am 01.02.2016, May 2002.
- [135] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. Technical report, WireX Communications, Inc. and Intercode Pty Ltd and NAI Labs and IBM Linux Technology Center, June 2002.
- [136] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. *IEEE PDP 2013*, 2012.