

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

10. 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

Inhaltsverzeichnis

1	Überblick	1
1.1	Struktur der Arbeit	3
2	Grundlagen	5
2.1	Virtualisierung	5
2.1.1	Hypervisor-basierte Virtualisierung	6
2.1.2	Container-basierte Virtualisierung	8
2.1.3	Einordnung Docker	11
2.2	Sicherheitsziele in der IT	11
2.2.1	Vertraulichkeit	12
2.2.2	Integrität	12
2.2.3	Authentizität	12
2.2.4	Verfügbarkeit	12
2.2.5	Verbindlichkeit	13
2.2.6	Privatheit, Anonymität	13
2.2.7	Authorisierung	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	20
2.3.6	Registries	22

3	Fragestellungen / Ziel der Arbeit	24
4	Security aus Linux Kernel-Features	27
4.1	Isolierung durch namespaces	27
4.1.1	Prozessisolierung durch den PID namespace	28
4.1.2	Dateisystemisolierung durch den mount namespace	29
4.1.3	Geräteisolierung durch	30
4.1.4	IPC-Isolierung durch den IPC-namespace	30
4.1.5	UTS-Isolierung durch den UTS-namespace	31
4.1.6	Netzwerkisolierung durch den network namespace	31
4.1.7	Userisolierung (user namespace)	32
4.2	Ressourcenverwaltung / Limitierung von Ressourcen durch cgroups	33
4.3	Einschränkungen von Zugriffsrechten	36
4.3.1	capabilities	37
4.3.2	Linux Security Modules (LSMs) und Mandatory Ac- cess Control (MAC)	37
4.3.2.1	SELinux	40
	Type Enforcement (TE)	41
	Multi-Category Security (MCS)	42
	Multi-Level Security (MLS)	43
4.3.2.2	AppArmor	43
4.3.3	Seccomp	45
4.4	Docker im Vergleich zu anderen Containerlösungen	47
5	Security im Docker-Ökosystem	48
5.1	Docker Plugins	49
5.2	Security Policies	49
5.3	Lifecycle- und State-Management von Containern	49
5.4	Docker Images und Registries	49
5.4.1	neues Signierungs-Feature	49
5.5	Docker Daemon	49
5.5.1	REST-API	49

5.5.2	Support von Zertifikaten	49
5.6	Containerprozesse	49
5.7	Docker Cache	49
5.8	privileged Container	49
5.9	Networking	49
5.9.1	bridge Netzwerk	49
5.9.2	overlay Netzwerk	49
5.9.3	DNS	49
5.9.4	Portmapping	49
5.10	Daten-Container	49
5.11	Docker mit VMs	49
5.12	Sicherheitskontrollen für Docker	49
5.13	Tools rund um Docker	49
5.13.1	Docker-Erweiterungen	49
5.13.1.1	Docker Swarm	49
5.13.1.2	Docker Compose	49
5.13.1.3	Nautilus Project	49
5.13.2	Third-Party Tools	49
5.13.3	Vagrant	49
5.13.4	Kubernetes	49
6	Docker in Unternehmen/Cloud-Infrastrukturen	51
7	Fazit	52

Abbildungsverzeichnis

1	Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[37].	2
2	Die Client-Server-Architektur von Docker [19].	16
3	Aufbau eines Docker-Hosts, wenn dieser unter einem Linux-Betriebssystem betrieben wird, das direkt auf der Serverhardware läuft. [82, 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 [43].	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 [25].	23
9	Funktionsweise von <i>System Call</i> -Hooks eines LSMs [99, S.3]. .	39

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 [100, S.1]. Vor allem in Rechenzentren bieten sich Virtualisierungen an, um die Serverressourcen effizienter zu nutzen [81, 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 [81, S.1].

Heutzutage existieren mehrere serverseitige Virtualisierungstechniken, wovon die Hypervisor-gestützten Methoden mit den etablierten Vertretern *Xen*[40], *KVM*[38], *VMware ESXi*[39] und *Hyper-V*[71] die meistverbreitesten sind [100, 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 [31]. 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 [37].

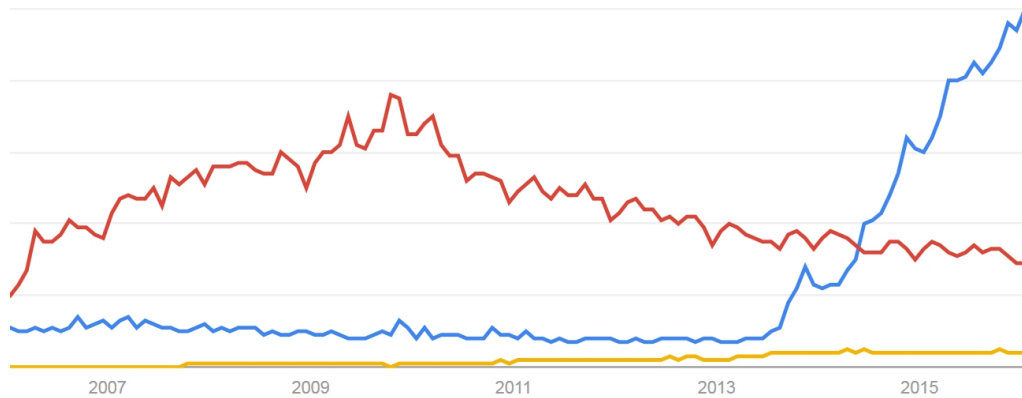


Abbildung 1: Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[37].

Obwohl das Konzept von Containern bereits im Jahr 2000 als *Jails* in dem Betriebssystem *FreeBSD* und seit 2004 als *Zones* unter *Solaris* verwendet wurde [58][57], gelang keiner dieser Technologien vor Docker der Durchbruch. Wie Docker den bis 2013 vorherrschenden Ruf von Containerertechnologien, dass Container noch nicht ausgereift seien [100, 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 Sicherheitsschwächen gegenüber Hypervisor-gestützten Virtualisierungsarten beliebt. Vor allem Multi-Tenant-Services werden gerne mit Docker umgesetzt [97, S.6][19].

1.1 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 in der IT in Kapitel 2.2 erklärt, auf die in der Untersuchung Bezug genommen wird. Abgeschlossen wird das Grundlagenkapitel mit einer Einführung in Docker (Kapitel 2.3), in der die Terminologie sowie Funktionsweise dieser Plattform erläutert wird.

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. **Sicherheitsfunktionen, die der Linux-Kernel anbietet** und teils obligatorisch von Docker eingesetzt werden. Darunter fallen Techniken zur Isolierung, Ressourcen- und Rechteverwaltung von Containern sowie Methoden, um das Hostsystem mit zusätzlichen Linux Sicherheitsfeatures abzusichern.
2. **Sicherheit im Docker-Ökosystem**, also 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

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

In der Arbeit vorkommende Produkt-, Technologie-, Bibliotheken- und Unternehmensnamen sind durchgehend *kursiv* gedruckt. Im Gegensatz dazu sind technische Identifikationsmerkmale und Variablennamen **mono-type** geschrieben.

Eine Ausnahme bildet Docker, in der die reguläre Schreibweise für die Plattform Docker vorgesehen ist, während die kursive Variante das Unternehmen *Docker* meint.

Kapitel 2

Grundlagen

2.1 Virtualisierung

Bei der Virtualisierung, die bereits über 40 Jahre alt ist, werden ein oder mehrere virtuelle IT-Systeme auf einem physischen Computer betrieben. Mehrere solcher Computer können eine virtuelle Infrastruktur bilden, in der physische und virtuelle Maschinen gemeinsam verwaltet werden können [96, S.661].

Virtualisierte Komponenten nutzen im Vergleich zu rein nativen (physischen) Systemen eine zusätzliche Softwareschicht, die den virtualisierten IT-Systemen in der Ausprägung von virtuellen Maschinen, sogenannte (VMs), und Containern, mehrere Abstraktionen anbietet, um Funktionen des Hosts abzubilden [100, S.2]. Beide Ausprägungen erwecken aus Sicht des Gasts den Eindruck, dass ein alleinstehendes Betriebssystem ausgeführt wird. Das Betriebssystem, das direkt auf der Hardware läuft, wird als Host bezeichnet. Systeme, die virtualisiert auf einem Host laufen, werden als Gäste oder Gastssysteme bezeichnet.

Der Einsatz von Virtualisierung bietet vielfältige Vorteile für IT-Unternehmen. In der folgenden Auflistung sind einige Problemfaktoren und Anforderungen an IT-Lösungen zusammengefasst, die alle durch Virtualisierungslösungen

adressiert werden können [88, S.1][96, S.662].

Wie zu sehen ist, bietet der Einsatz von serverseitiger Virtualisierung eine Reihe von Vorteilen, die ein physischer Betrieb nicht bieten kann.

2.1.1 Hypervisor-basierte Virtualisierung

Im Kontext von einer Hypervisor-basierten Virtualisierung, wird die virtuelle Umgebung 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 [97, S.6][100, S.2][81, S.2]. Ein weitere wichtige Eigenschaft eines Hypervisors ist es, dem Gast jede für den Betrieb eines Betriebssystems nötige Funktion anzubieten [96, S.106]

Durch diese Technik läuft in jeder VM ein eigenes (Gast-)Betriebssystem, das von solchen anderer VMs komplett isoliert läuft. Durch die Abstraktion des zwischenliegenden Hypervisors ist es möglich, mehrere unterschiedliche Gastbetriebssysteme auf einem physikalischen Host auszuführen [100, S.2][96, S.106].

Der größte Kritikpunkt dieser Art von Virtualisierung ist deren hoher Bedarf an Hostressourcen, da diese für jede gestartete VM komplett virtualisiert werden müssen, sodass innerhalb der VM ein Gast-OS ausgeführt werden kann [78, S.1][79, S.3].

Hypervisorstechnologien werden unter sich in solche von Typ 1 und Typ 2 unterschieden.

Typ 1 Hypervisor operieren direkt auf der Hardware des Hosts, stellen sozusagen 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

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 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 zusätzliche Kapazitäten in der IT-Infrastruktur bereitgestellt werden können.	loesung 2
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 native Möglichkeiten zur Isolierung von Systemen. Eine Gefahr von Interferenz zwischen Kunden entsteht dadurch nicht.
Redundanz, um Ausfällen entgegenzuwirken, soll gewährleistet sein.	VMs und Container sind flexibel zu starten und stoppen. Ausfallsicherheit wird in Form mehrerer redundanter virtueller Instanzen einfach 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 anderen 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 können mit individuellen Betriebssystemen gestartet werden. 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 „eingeschweißte“ Anwendungen [96, S.672f.].
Neue Services und Kunden sollen einfach in bestehende Infrastrukturen integrierbar sein. Systeme sollen einfach hinzugefügt werden können.	virtuelle Instanzen können als sogenannte Snapshots „eingefroren“ werden und in anderen Infrastrukturen eingesetzt werden.

ein Befehl ausgeführt, 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 [96, 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 eines in dieser Anwendung installierten Betriebssystems, 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 [96, S.665f.].

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 [96, S.666f.][81, S.2].

Bekannte Vertreter von Hypervisoren sind die kommerziellen *ESXi* der Firma *VMware* und *Hyper-V* von *Microsoft*, sowie die ebenfalls namhaften Open-Source-Hypervisor *Xen* und *KVM* [79, S.1].

2.1.2 Container-basierte Virtualisierung

Container-basierte Virtualisierung wird vorrangig als leichtgewichtige Alternative zu der Hypervisor-basierten Virtualisierung gesehen[100, S.2]. Erstere

nutzt direkt den Hostkernel, um virtuelle Umgebungen zu schaffen. Ein Hypervisor wird in diesem Ansatz nicht benötigt [97, 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 [100, S.2][82, S.3][94, S.1]. Die Isolation basiert auf dem Konzept von Kontexten, die unter Linux *namespaces* genannt werden. Diese, sowie *cgroups*, die für das Ressourcenmanagement verantwortlich sind, werden in den Kapiteln 4.1 und 4.2 genauer betrachtet [82, S.4].

Container sind durch den Unix-Befehl *chroot*[46] inspiriert, 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 [27]. In *Solaris*, ein von der Firma *Oracle* entwickeltes Betriebssystem für Servervirtualisierungen[41], wurde dieser Mechanismus in Form von *Zones* (Solaris-spezifischer Begriff) [72] 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 [79, S.2]. Durch die kontinuierliche Weiterentwicklung von Containern in den letzten Jahren, können diese heutzutage als vollwertige Systeme betrachtet werden, nicht mehr als - wie ursprünglich vorgesehen - reine Ausführungsumgebungen [97, S.7].

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 [97, S.6+7][100, S.2][79, 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 [97, 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 [91].

Ein großer Vorteil jedoch, der sich durch das schlankere Design ergibt, ist eine fast nativen Performance [100, S.1] der Container, 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 [100, S.4][86, S.5]. In traditionellen Virtualisierungen beansprucht der Hypervisor allein etwa 10-20% der Hostkapazität [78, S.2][86, S.5]. Ein Benchmarktest, der den Durchsatz (Operationen pro Sekunde) eines *VoltDB*-Setups[69] von Hypervisor-basierte Cloudlösungen mit containerbasierten Cloudlösungen verglich, kam zu dem Ergebnis, dass die Containerlösung unter genanntem Gesichtspunkt sogar eine fünffache Leistung aufweist [64, S.2+3]. In der Praxis machen sich diese Verhältnisse an einer hohen Dichte an Containern bemerkbar und führen zu einer besseren Ressourcenausnutzung [97, 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* [94, 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 [97, 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 designtechnischer Sicht unsicherer, die virtuellen Umgebungen direkt auf einem Host laufen zu lassen. Angriffe, die von einem Gast-OS über die zusätzliche Softwareschicht eines Hypervisors an den

Host gerichtet sind, 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 [97, 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 Host, der im Millisekundenbereich abgeschlossen ist [79, 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 [97, S.7][100, S.1][79, S.2].

Docker ist wie in Kapitel 2.1.2 zuvor angedeutet, nicht die erste containerbasierte Virtualisierungslösung. Einige ältere Containersysteme, wie z.B. *Solaris Zones*, existieren schon länger als Docker, etablierten sich allerdings nie in der Praxis. Der anhaltende Erfolg von Docker beruht überwiegend nicht auf der überlegenden technischen Eigenschaften, sondern vielmehr auf den Tools und dem Workflow, den *Docker* seinen Kunden anbietet.

2.2 Sicherheitsziele in der IT

Folgende Sicherheitsziele können für IT-Systeme definiert werden.

2.2.1 Vertraulichkeit

Die Vertraulichkeit steht für das Konzept von Geheimhaltung. Durch verschiedene kryptographische Verschlüsselungsverfahren kann Klartext in einen unleserlichen Geheimtext transformiert werden, der keine Information über den ursprünglichen Klartext enthält und somit sicher gegenüber Abhörern ist.

2.2.2 Integrität

Unter Integrität versteht man die Zusicherung, dass bestimmte Daten original sind und nachweisbar nicht manipuliert wurden. Integrität kann für Daten z.B. mit kryptographisch sicheren MACs hergestellt werden.

2.2.3 Authentizität

Authentizität beschreibt die Identifikation eines Objekts gegenüber einem System. Maßnahmen der Authentifikation sind z.B. Passwortabfragen, digitale Zertifikate oder biometrische Merkmale einer Person. Ist eine Authentifikation erfolgreich, ist die Echtheit des Objekts bestätigt.

2.2.4 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.

2.2.5 Verbindlichkeit

Die Verbindlichkeit eines Systems sagt aus, dass jede Aktion eindeutig auf eine Ursache, also z.B. einen User, der die Aktion ausgeführt hat, zurückzuführen ist.

2.2.6 Privatheit, Anonymität

Die Anonymität als Schutzziel erfüllt i.d.R. Datenschutzbestimmungen, nach denen Nutzer nicht als Individuen identifiziert werden dürfen. Dieses Ziel hat keinen Bezug zur vorliegenden Arbeit, soll aber zur Vollständigkeit an dieser Stelle aufgeführt sein.

2.2.7 Authorisierung

Ist das eigenes Sicherheitsziel? Quellen widersprechen sich.

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.*[51], vormals *dotCloud Inc.*, sowie mehr als 1.600 freiwillig mitwirkenden Entwicklern ständig weiterentwickelt. [33][97, S.7][31][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 [97, S.7] und bietet neben Containern viele Tools und einen Workflow für Entwickler, die beide die Arbeit mit Containern erleichtern sollen [78, 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 (Container) eingesetzt werden [97, 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 [97, S.12].

Ein bekanntes Problem bei der Softwareentwicklung ist, dass Code 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 Übergabe von Entwicklungs- nach Produktionsumgebung mit sich zieht. Diesem Umstand wurde in der Industrie mit der Einführung von *DevOps*-Teams entgegengewirkt.

Das Kernproblem im genannten Szenario sind die Entwicklungs- und Produktionsumgebung, zwischen denen Code ausgetauscht wird, da diese unterschiedlicher Natur sind. Einen anderen Ansatz diese Problem auf eine technische Art und Weise zu lösen, bieten Container. Quellcode wird inklusive Ausführungsumgebung flexibel von einem Laptop auf einen Testserver und später auf einen physischen oder virtualisierten Produktionsserver oder Cloud-Infrastruktur, wie z.B. *Microsoft Azure*, geschoben (und umgekehrt). Mit hoher Wahrscheinlichkeit sind die Anwendungscontainer unabhängig von der Infrastruktur sofort startfähig. Dieser kurzlebige Zyklus zwischen Entwicklung, Testen und Deployment erlaubt einen effizienten und konsistenten Workflow [97, S.8+12].

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 [78, 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 [19]. 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 [82, S.3].

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.

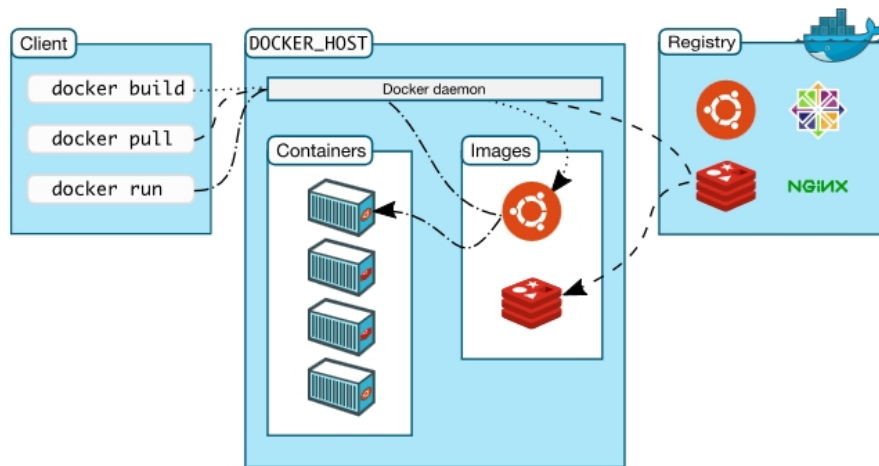


Abbildung 2: Die Client-Server-Architektur von Docker [19].

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.

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
```

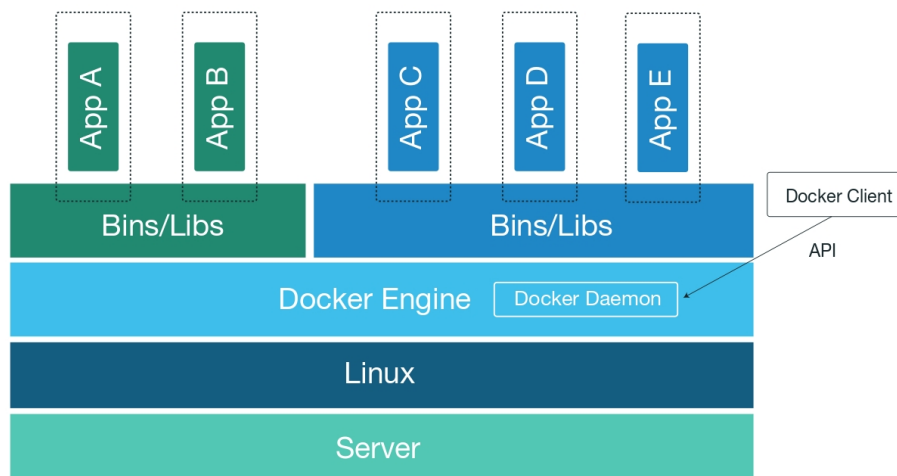


Abbildung 3: Aufbau eines Docker-Hosts, wenn dieser unter einem Linux-Betriebssystem betrieben wird, das direkt auf der Serverhardware läuft. [82, S.3].

```
CMD ["nginx"]
```

```
EXPOSE 80
```

```
EXPOSE 443
```

Die Erklärung der einzelnen Anweisungen [50]:

- **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.

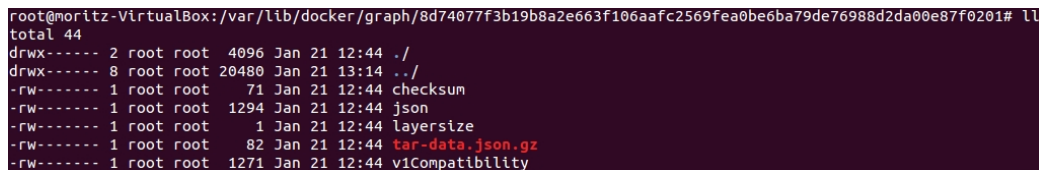
Im ersten Release von Docker wurde die Ausführungsumgebung *LXC* verwendet, die im März 2014 von der *Docker*-eigenen Entwicklung *libcontainer* abgelöst wurde. *libcontainer* ist komplett in der Programmiersprache *Golang* implementiert und kann ohne Dependencies mit dem Kernel kommunizieren [15].

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 [16]. 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 [44][36][42].

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 [97, S.11]. *Union*-Dateisysteme wie *AuFS* und *Device Mapper* haben gemeinsam, dass sie alle auf dem *Copy-on-write*-Modell basieren [97, S.8][78, S.3][82, 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 [34].



```
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 viCompatibility
```

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

Images werden Schritt für Schritt erstellt, z.B. mit den folgenden Aktionen [97, 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 auswirkt. 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 [78, 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 [52][23]. 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.

Eine Liste aller lokal vorliegenden Images, wie in Abb.7, kann mit dem Befehl `docker images` in der Shell generiert werden [22].

2.3.5 Container

Ein Container ist die laufende Instanz eines Images, die in Sekundenbruchteilen gestartet werden kann [78, 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.



Abbildung 5: Visualisierung eines Vergleichs von Images von *Redis*, *Nginx* und *CentOS* auf Schichtebene [43].

```

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).

```

moritz@moritz-VirtualBox:~$ docker images
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).

Darunter fallen z.B. das Erstellen, Starten, Stoppen, Neustarten und Beenden eines Containers. Welchen Inhalt einen Container hat, also ob ein Container z.B. auf einem Datenbank- oder Webserver-Image beruht, ist dafür unerheblich [97, S.12][79, S.2].

Container werden als privilegiert bezeichnet, wenn sie mit Root-Rechten gestartet werden. Standardmäßig startet ein Container mit einem reduzierten Set an sog. **capabilities**, welches keine vollen Root-Rechte umfasst.


2.3.6 Registries

Eine Registry ist ein gehosteter Service, der als Speicher- und Verteilerplattform für Images dient. Mit Tags versehen werden die Images in Repositories gegliedert, die wiederum in der Registry liegen [18]. Ein Repository besteht aus mindestens einem Image.

Docker stellt eine Vielzahl an Images öffentlich und frei verwendbar in einer eigenen zentralen Registry, dem Docker Hub, zur Verfügung [97, S.11][81, S.3][18]. 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) [62, S.16]. Wie in Abb.8 zu sehen ist, werden auch Nutzungsstatistiken pro Image gesammelt und angezeigt.

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** [52]. Wenn bei dem Download kein Tag angegeben ist, wie in Kapitel wird automatisch das aktuellste Image mit dem Tag **latest** bezogen.

Docker bietet außerdem an, private Registries zu erstellen. Diese können dann, z.B. gesichert von einer unternehmenseigenen Firewall, betrieben wer-


[Explore](#)
[Help](#)

[Sign up](#)
[Log In](#)

Explore Official Repositories




 <div> <div>busybox</div> <div>official</div> </div>	<div>434</div> <div>STARS</div>	<div>58.8 M</div> <div>PULLS</div>	<div>></div> <div>DETAILS</div>
 <div> <div>ubuntu</div> <div>official</div> </div>	<div>3.0 K</div> <div>STARS</div>	<div>37.7 M</div> <div>PULLS</div>	<div>></div> <div>DETAILS</div>
 <div> <div>swarm</div> <div>official</div> </div>	<div>115</div> <div>STARS</div>	<div>21.3 M</div> <div>PULLS</div>	<div>></div> <div>DETAILS</div>

Abbildung 8: Web-UI des Docker Hubs mit den beliebtesten Repositories [25].

den. Neben der Vertraulichkeit, bieten private Registries den Vorteil, dass sich die Speicherung und Verteilung von Images an den internen Softwareentwicklungsprozess anpassen lassen. Registries selbst können als Container betrieben werden [18].

Der Zugriff auf eine Registry kann über TLS und der Verwendung eines Zertifikats, sowie *basic authentication* abgesichert werden [18].

Kapitel 3

Fragestellungen / Ziel der Arbeit

Das zentrale Konzept, auf dem alle Containertechnologien beruhen, ist das der Isolierung. Im Kontext von Containern kann die Isolierung definiert werden als Trennung zwischen Containern und einem Host, sowie die Trennung zwischen Containern [94, S.1].

Auf einem System mit Host und einem oder mehreren Containern, stellt sich zunächst die Frage welche Art und Richtung von Kommunikation zwischen diesen beiden Komponenten erlaubt und nicht erlaubt sein soll. Dadurch, dass der Docker-Daemon auf dem Host läuft und es dessen Aufgabe ist u.a. den Container-Lifecycle zu kontrollieren, braucht dieser Zugriff auf die Container. Verallgemeinert ist also die Kommunikation von Host zu Container erforderlich und damit erlaubt.

Was in einem Container passiert, ist zweitrangig, da der Container bei Fehlfunktionen jederzeit seitens des Hosts neu gestartet werden kann. Wichtig ist aber, dass der Container selbst von der Außenwelt, also dem Host und anderen Containern, isoliert ist und seine Aufrufe gegen den Hostkernel streng limitiert sind und diese den Host nicht beeinträchtigen können.

Mehrere Sicherheitsfragen für Container-basierte Systeme sind in den folgen-

den Punkten formuliert. Sie beruhen auf der Annahme, dass ein Angreifer die Kontrolle über einen Container X übernommen hat und versucht, über diesen Schaden zu verursachen.

Situationen, in denen ein Angreifer bereits zu Beginn die Kontrolle über den Host hat, werden nicht betrachtet, da der Angreifer in dieser Lage bereits gewonnen hat und Container nach belieben manipulieren kann.

- (1) Ist es dem Angreifer möglich, seine in X erworbenen Rechte auf den Hosts zu erweitern, sodass er auf letzteren Root-Rechte erwirken kann? (Verletzte Sicherheitsziele: Vertraulichkeit, Authentizität, Integrität)
- (2) Ist es dem Angreifer möglich, auf einen anderen Container Y des gleichen Hosts zuzugreifen? (Verletzte Sicherheitsziele: Vertraulichkeit, Authentizität, Integrität)
- (3) Ist es dem Angreifer möglich, den Container oder Host auf eine Art und Weise zu beeinflussen, die den Betrieb anderer Container auf diesem oder entfernten Hosts beeinträchtigt? (Verletzte Schutzziele: Verfügbarkeit, Integrität) (Ressourcenverwaltung)
- (4) Ist es dem Angreifer möglich, den Container X negativ zu beeinflussen oder ihn zum Absturz zu bringen? (Lifecyclemanagement des Docker-Hosts)
- (5) Wie wird natürlichen Fehlfunktionen von Containern entgegengewirkt? (Lifecyclemanagement des Docker-Hosts)
- (6) *weitere Punkte?*

Frage (1.) und (2.) zielen auf technischer Ebene auf die Isolation der Container ab. Eine Umformulierung in „Sind Container ausreichend isoliert, um den Host zu schützen?“ ist möglich.

Wenn von der Netzwerkseite abgesehen wird, lässt sich das Szenario der Fragestellung (2.) auf das der Frage (1.) reduzieren, da der Zugriff auf andere Container nur über den lokalen Host möglich ist. Genauer gesagt ist der Zugriff auf andere Prozesse nur dann möglich, wenn Root-Rechte auf dem

Host vorhanden sind. Die bereits generalisierten Sicherheitsfrage ist in (A.) unter Berücksichtigung dieses Punkts, erweitert

Die Fragen (3.), (4.) und (5.) teilen sich den Aspekt der Verfügbarkeit, der in Formulierung (B.) aufgegriffen wird.

—

Finale Umformulierungen und Generalisierungen:

- (A) Sind Container ausreichend isoliert, sodass ausgehend von Containern keine Root-Rechte auf dem Hostsystem erwirkt werden können?
- (B) Kann der Betrieb von Containern negativ beeinflusst werden, sodass die Verfügbarkeit von Anwendungen darunter leidet?
- (C) *weitere Punkte?*

ALT:

Um Frage (1.) zu beantworten, wird im ersten Hauptkapitel die intrinsische Sicherheit von Docker untersucht. Damit ist eine Reihe von Sicherheitsfeatures des Linux Kernels gemeint, die u.a. Docker nutzt, um nach Aussage des Unternehmens *Docker* sichere Container zu ermöglichen. V.a. Mechanismen zur Isolation und Ressourcenverwaltung werden betrachtet, da sie direkt mit den erwünschten Sicherheitszielen aus Kapitel 2.2 in Bezug stehen.

Des Weiteren stellt sich die Frage, ob die Arbeit mit Docker und seinen Containern sicher ist. Wie in der Einführung zu Docker beschrieben, stellt Docker zusammen mit anderen Anbietern einen Workflow und eine Palette an Tools zur Verfügung, die die Arbeit mit Containern erleichtern sollen. Wie diese Tools zur Sicherheit bzw. Angreifbarkeit von Docker-Systemen beitragen, wird im Kontext von den Sicherheitszielen betrachtet.

Nicht betrachtet werden die Sicherheitsrisiken, die sich durch den Betrieb eines Rechnernetzwerks ergeben, in dem Docker-Knoten existieren. Sicherheit aus Sicht der Netzwerktechnik und den verschiedenen OSI-Schichten ist nicht Gegenstand der Untersuchung.

Kapitel 4

Security aus Linux Kernel-Features

4.1 Isolierung durch namespaces

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 [78, 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 [80, S.1+2][47]. 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. Diese Eigenschaft macht es Angreifern schwieriger Schaden anzurichten, da sie ausgehend von kompromitierten Containern keine Informationen über Prozesse außerhalb des Containers beziehen können.

Ein weiterer Mechanismus des *PID-namespace* 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.

[81, S.4]

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 [82, 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 [82, S.4].

[81, 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.

[81, 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 beabsichtige Interferenz mit Prozessen des Hosts oder anderer Container wird damit ausgeschlossen.

[81, 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 [83, 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 [83, 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 [80, 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 [83, 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.

[81, 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 Eigenerinformationen zu versehen. *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 [48].

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 [31][56]. Verzögerungen entstanden durch einen Bug der Programmiersprache *Golang* [74], und Integrationschwierigkeiten in die bestehende Docker-Codebasis [67].

In der aktuell neusten Docker-Version 1.10 werden *user namespaces* nicht automatisch verwendet. Sie müssen manuell, wie z.B. in [90] erklärt, aktiviert werden.

Beide Probleme sind jedoch mittlerweile gelöst, wie die erfolgreiche Integration von *user namespaces* in Docker im Oktober 2015 bestätigt [54]. Dadurch, dass *user namespaces* in der Docker-Roadmap als wichtiges Sicherheitsfeature gesehen werden, ist ein Release dessen bald zu erwarten [35].

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 [84, 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 [81, 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 [29]. *Solaris* bietet die Nutzung von *Resource Pools* an, die eine Partitionierung von Ressourcen implementiert [11]. Auch *OpenVZ* und *Linux-VServer* erweitern **rlimits**, sodass Ressourcenlimits pro Container definiert werden können [94, 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 [12]. 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 [28][66]. 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 [94, S.19]. Auch im Quellcode von *runC* ist eine Dateisystem-Interface als „nicht unterstützt“ gekennzeichnet und wird demzufolge auch nicht von Do-

cker genutzt. [32]. 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 *Brdfs* 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 [94, S.16+19].

Alle gängigen Linux-basierten Containerlösungen, darunter auch Docker, nutzen aktuell **cgroups**, um Ressourcen für Container zu verwalten [94, 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 [12][32].

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 [24].

```
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 [20].

4.3 Einschränkungen von Zugriffsrechten

Kapitel mit nächstem Unterkapitel verschmelzen. Oder eigenes LSM-capability Kapitel machen, was diese Unterteilung rechtfertigt.

Dieses Kapitel stellt weitere Sicherheitsmechanismen vor, die Docker nutzt.

Oben behandelte Namespaces hatten schon einige Bugs. Alle zwar bisher behoben, trotzdem Beweis dafür, dass Namespace- und Cgroups-Sicherheit allein nicht ausreichen, um Container zu isolieren.

Im Normalfall kontrolliert Linux den Zugriff auf Ressourcen anhand der Identität des anfragenden Users. Diese sogenannte *Discretionary Access Control*, kurz DAC, implementiert eine einfache Form von ACL. Identifikationsmerkmale dieser umfassen den Besitzer (owner) einer Datei, die Gruppenzugehörigkeit des Besitzers (group) und sogenannte Permission-Flags aus der Menge **r**, **w**, **x**. Letztere legen z.B. für Dateien fest, ob Lese- (r=read), Schreib- (w=write) und Ausführrechte (x=execute) gewährt werden.

LSMs bieten die Möglichkeit, Ressourcenzugriffe userunabhängig zu überprüfen. Sie realisieren eine weitere Sicherheitsschicht, die Angreifern festgelegte Ressourcen verweigert, selbst wenn sie Root-Rechte in einem Container haben.

Die Docker-Entwickler haben in den letzten Monaten die Integrations- und Anpassungsmöglichkeiten von zusätzlichen Sicherheitsmaßnahmen, v.a. MACs, stark verbessert, da die Containersicherheit für *Docker* höchste Priorität hat [35][31]. Auch die Tatsache, dass sich zur Zeit die Konkurrenz *CoreOS* mit der Containerlösung *rkt* als sicherheitsfokussierte Alternative zu Docker auf dem Virtualisierungsmarkt etablieren will [77], 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 04. Februar 2016 geschah in der Tat ca. drei Stunden nach der Ankündigung der Version 1.0 von *rkt* seitens *CoreOS* [75][76], was eine starke Konkurrenz zwischen den beiden Parteien erkenntlich macht.

4.3.1 capabilities

Capabilities ist ein Feature, dass seit Kernelversion 2.2 in Linux integriert ist, um die traditionellen UNIX-Privilegien zu verfeinern. Es erlaubt die dem Root-User mit `UID=0` zustehenden Rechte in individuelle, voneinander unabhängige Einheiten zu unterteilen, die **capabilities** genannt werden. Jede privilegierte Operation ist auf eine **capability** abgebildet. Zum Einen ist es Prozessen nicht-privilegierter User damit möglich, Operationen auszuführen, die ohne Einsatz dieses Features nur Root-User ausführen dürfen (Rechteauserweiterung). Zum Anderen lassen sich Prozesse auch mit **capabilities** einschränken, sodass sie z.B. nur von einem stark reduzierten Rechteset Gebrauch machen können (Rechteeinschränkung)[89, S.33].

Der DAC-Mechanismus ist nur in der Lage Root-Rechte an Prozesse zu vergeben, die eine privilegierte Operation ausführen wollen. Mit Root-Rechten werden alle normalen Rechtekontrollen des DACs umgangen. Aus sicherheitstechnischer Sicht ist das kritisch, da der anfragende Prozess mit Root-Rechten auf dem System jede denkbare Operation ausführen kann und nicht, wie ursprünglich vorgesehen, nur die ihm privilegierte Operation [89, S.797]. Mithilfe von **capabilities** wird im Gegensatz zu dem „Alles-oder-nichts“-Ansatz des DACs eine feinere und sichere Rechtevergabe ermöglicht.

Capability-Namen beginnen mit dem Prefix `CAP_` [89, S.33,S.797].

Evtl. hieraus nur capability-Eintrag im Glossar machen. Oder in Grundlagen Kapitel.

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

Der DAC-Mechanismus erfüllt nicht moderne Anforderungen an die Sicherheit in Containersystemen, da sie z.B. von Superuser-Tools wie *sudo* umgangen werden kann [99, S.1]. Aus diesem Grund integriert Docker MACs wie *SELinux* und *AppArmor*, die anhand eines identitätsunabhängigen Regel-

werks zusätzliche Sicherheit bieten. Unter Einbeziehung solcher Kontrollmodule kann nun z.B. auch der Ressourcenzugriff eines Angreifers eingegrenzt werden, selbst wenn dieser in den Besitz von Root-Rechten gelangen konnte.

Eine Möglichkeit, Kontrollmodule in Linux zu integrieren bietet das *Linux Security Modules*-Framework, auch kurz LSM genannt, das inzwischen standardmäßig in den Kernel eingebaut ist. Module, die in das Framework eingebettet werden, sind kombinierbar, Sicherheitsmodelle konsekutiv umgesetzt werden können [99, S.3]. Das überwiegend implementierte Sicherheitsmodell ist das der *Access Control List* (ACL), das anhand definierter Parameter entscheidet, ob der Zugriff auf eine Resource genehmigt oder verweigert wird. Ressourcen umfassen in diesem Kontext alle jene, die von einem Kernel in interne Kernelobjekte aufgelöst werden können, also z.B. Dateien, Verzeichnisse, Sockets, Geräte, etc. Im folgenden werden die Begriffe Ressource und Objekt synonym verwendet. Die eigentliche Kontrolle geschieht, wie in Abb.9 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. LSM-Kontrollen geschehen erst, wenn die DAC den Zugriff gestattet hat (vgl. Abb.9). Durch diese Reihenfolge ist sichergestellt, dass die Nutzung einer LSM-Schnittstelle optional ist und unabhängig von der DAC funktioniert [30]. Deswegen können auch Anwendungen, die das LSM-Framework nicht unterstützen, weiterhin funktionieren.

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 [99, S.3].

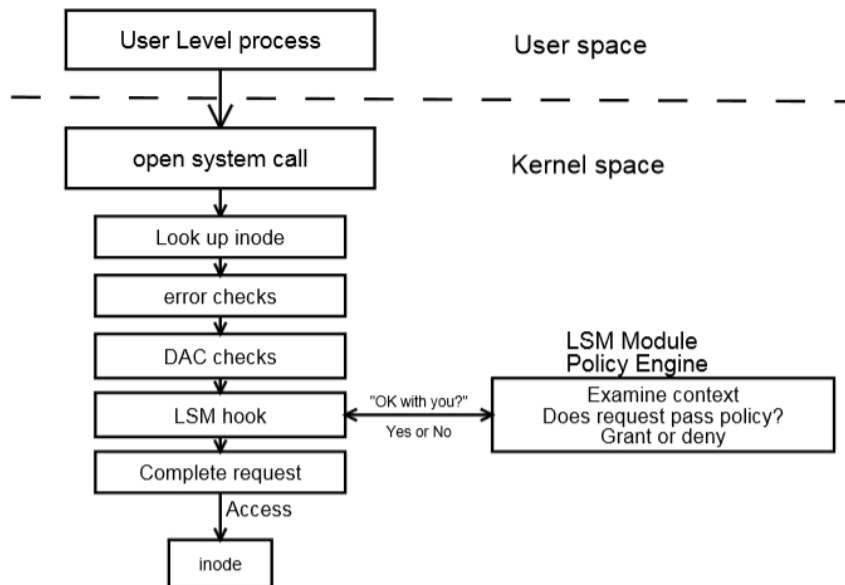


Abbildung 9: Funktionsweise von *System Call*-Hooks eines LSMs [99, S.3].

Wie in Abb.9 dargestellt, greift der LSM-Hook erst, wenn ein User einen sicherheitskritischen *System Call* ausführt, die hierbei angefragte Ressource aufgelöst werden, Fehler-Checks abgeschlossen und der Zugriff über den klassischen DAC-Mechanismus genehmigt wurde. Erst wenn der Kernel versucht auf das 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 Zugriffsparametern.

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 [99, S.2]. Zusätzlich wird dadurch die Granularität der Zugriffskontrolle verbessert werden [98].

In den folgenden Unterkapiteln werden die beiden MACs *SELinux* und *AppArmor*, sowie deren Einsatzzweck unter Docker vorgestellt. Ein Ausblick in die Zukunft von MACs und Docker ist in Kapitel 7 gegeben.

4.3.2.1 SELinux

SELinux steht für *Security Enhanced Linux* und implementiert eine feingranulare MAC, die ursprünglich von der NSA entwickelt wurde. Es wird verwendet, um Anforderungen an die Integrität und Vertraulichkeit von Prozessen und Daten, sowie das *Principle Of Least Privilege* zu realisieren [68].

Die Beschränkungen beruhen auf einer Sicherheits-Policy, die alle Aktionen zwischen Usern und Ressourcen regelt. Die Policy besteht aus Anweisungen, die konkrete Sicherheitslabel, wie sie im nächsten Abschnitt beschrieben sind, abbilden. Durch die strikte Trennung der Regelwerk 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 vereinfachten [85, 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 [21].

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 Prozesse verwaltet [92]. In Zuge dessen wird jedem Prozess zur Laufzeit und jedem Objekt im System ein Label nach dem Schema `User:Role:Type:Level` zugewiesen [17]. Für Objekte wird das Label in den erweiternden Attributen (`xattr`) geschrieben [85, S.65]. Die erste Komponente `User` eines Labels ist von einem Linux-Users, der mit DAC ausgewertet wird, unabhängig.

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 [73]. Die Regeln werden in ihrer Summe auch als Policy bezeichnet.

Seit November 2015 wird Docker mit dem Release 1.9.0 mit einer stan-

dardmäßigen *SELinux*-Policy ausgestattet, die in dem *rpm*-basierten Distributionen wie *CentOS* und *Fedora* verwendet wird [31][3]. Diese Standard-Policy kann über [61] aufgerufen werden.

Unter Docker können die vier Label-Attribute mit dem Parameter `--security-opt="label:LABEL"` für den `run`-Befehl pro Container manuell spezifiziert werden [24].

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 *SELinux*-Bausteine, die im Betrieb von Docker eingesetzt werden, sind *Type Enforcement* und *Multi-Category Security* (MCS). Die Eigenheiten beider Mechanismen und deren Funktionsweise unter Docker werden im Folgenden erklärt. Ein weiteres Element *Multi-Level Security* (MLS) wird in dieser Arbeit nicht behandelt, da es unter Docker keine Verwendung findet.

Type Enforcement (TE) Die wichtigste Komponente von *SELinux* ist das *Type Enforcement*. Nach diesem Modell wird jedem Objekt ein Typ zugewiesen, das bei einer Zugriffsoperation ausgewertet wird. Der Typ eines Objekts ist im Label an dritter Stelle definiert.

Jeder Docker-Prozess hat z.B. im Label den Typ `docker_t` definiert. Im *SELinux*-Regelwerk ist festgelegt, dass Prozesse eines Typs nur auf Verzeichnisse und Dateien vollen Zugriff hat, die mit bestimmten Labeltypen versehen sind. Im konkreten Fall von `docker_t` umfasst diese Konfiguration ein Set, das z.B. die Typen `cgroup_t`, `docker_config_t`, `docker_log_t` und `svirt_sandbox_file_t` enthält. Versucht ein Docker-Prozess auf Objekte anderer Typen zuzugreifen, 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. Prozesse, die mit dem Typ `docker_t` aus der Docker-Domäne stammen, sind streng von Objekten anderer Domänen abgegrenzt und können nicht 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...*

4.3.2.2 AppArmor

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

Der fundamentale Unterschied zu *SELinux* ist, dass Objekten keine Label zugewiesen werden, sondern grundsätzlich Pfadnamen und sieben Berechtigungstypen dazu dienen, ein Sicherheitsprofil zu definieren [73][10].

AppArmor unterstützt einen Lernmodus, indem das Verhalten einer Anwendungen beobachtet wird und daraus automatisch ein Sicherheitsprofil erstellt wird [73].

AppArmor-Profile basieren auf einfach lesbaren Textdateien. Durch eine Inkludieranweisung lassen sich mehrere Profile modular kombinieren [10].

Das Standardprofil `docker-default`[7], das im `enforce`-Modus unter *.deb*-Releases in nicht-privilegierten Containern zum Einsatz kommt [87], wird bei der Installation von Docker in die Datei `/etc/apparmor.d/docker` geschrieben. Dieses Standardprofil kann mit dem `run`-Parameter `--security-opt=apparmor:PROFILE` manuell überschrieben werden, sofern dieses in *AppArmor*, z.B. mit dem CLI-Tool `apparmor_parser` [65], zuvor importiert wurde [24].

Das Profil `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 [13]. Die aktuelle Implementierung des Profils, die mit dem Konsolenbefehl `cat /etc/apparmor.d/docker` ausgegeben werden kann, sieht folgende *AppArmor*-Regeln vor:

```
#include <tunables/global>

profile docker-default flags=(attach_disconnected,mediate_deleted)
#include <abstractions/base>

network,
capability,
file,
umount,

deny @{PROC}/{*,**^[0-9*],sys/kernel/shm*} wx,

deny @{PROC}/* w,
deny @{PROC}/{[^1-9],[^1-9][^0-9],[^1-9s][^0-9y][^0-9s],[^1-9][^0-
deny @{PROC}/sys/[^k]** w,
deny @{PROC}/sys/kernel/{?,?,[^s][^h][^m]**} w,

deny @{PROC}/sysrq-trigger rwklx,
deny @{PROC}/mem rwklx,
deny @{PROC}/kmem rwklx,
deny @{PROC}/kcore rwklx,

deny mount,

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,
}

```

Evtl. in Anhang damit

Es existiert auch ein Profil für den Docker-Daemon [8], allerdings muss dieses manuell aktiviert werden [9].

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 [73][45][93].

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 [73].

Ein Update *mode 2* macht das Set an erlaubten *System Calls* mithilfe von

Filtern frei konfigurierbar und führt ein *Audit Logging* ein [73][45].

Mithilfe der *Seccomp*-Anweisungen **allow**, **deny**, **trap**, **kill** und **trace** sind neben der Sperrung noch weitere Aktionen, die zur Kontrolle von *System Calls* dienen, möglich [87].

Seit Oktober 2015 ist eine *Seccomp*-Unterstützung in Planung und Entwicklung [26][55]. 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 [31][60][59][87]. 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* [60].

Eine aktuelle Liste der explizit erlaubten und resultierend blockierten Aufrufe ist in [59] und [60] 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 [14]. Während sich es bei den Neueinträgen um bewusste Funktionserweiterungen 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 [24]. Jedoch ist an anderer Stelle im *GitHub*-Repository vermerkt, dass sich das Standardprofil mit dem Parameter **--security-opt seccomp:PROFILEPATH** überschreiben lässt [60]. Ist es nötig, Container ohne *Seccomp* zu starten, kann das mit der Option **--security-opt seccomp:unconfined** realisiert werden [87].

4.4 Docker im Vergleich zu anderen Containerlösungen

Kapitel 5

Security im Docker-Ökosystem

5.1 Docker Plugins

5.2 Security Policies

5.3 Lifecycle- und State-Management von Containern

5.4 Docker Images und Registries

5.4.1 neues Signierungs-Feature

5.5 Docker Daemon

5.5.1 REST-API

5.5.2 Support von Zertifikaten

49

5.6 Containerprozesse

5.7 Docker Cache

den.

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.

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 [79, S.4].

Glossary

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. 17

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.

[70] . 1, 10, 14

Denial of Service *gescheite Quelle. Buch hier am besten..* 12

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 [95]. 14

Kernelobjekt Datenstrukturen im Kernel, die verschiedene Ressourcen abbildet und von LSMs ausgewertet werden kann [53]. 38, 39

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 [49]. 2, 33

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 [63]. 34

Abkürzungsverzeichnis

ACL Access Control List. 36, 38

API Application Programming Interface. 15, 18, 49

ARP Address Resolution Protocol. 32

cgroups Control Groups. 33–35

CLI Command-Line Interface. 43

COW Copy-On-Write. 29

CPU Central Processing Unit. 1, 27, 34, 35

DAC Discretionary Access Control. 36

DoS Denial of Service. 12, 33, *Glossary*: Denial of Service

HDD Hard Disk Drive. 35

HPC High Performance Computing. 10

HTTP Hypertext Transfer Protocol. 18

HTTPS Hypertext Transfer Protocol Secure. 18

I/O Input and Output. 35

IPC Inter Process Communication. 30, 31

IT Informationstechnik. 3, 5, 11, 15

JSON JavaScript Object Notation. 19

LSM Linux Security Modules. 37, 38

MAC Mandatory Access Control (nicht Netzwerkkommunikation). 37

MAC Media Access Control (Netzwerkkommunikation). 32

MLS Multi-Level Security. 42

OCF Open Container Format. 18

OCP Open Container Project. 18

OS Operating System. 6, 9–11

OSI Open Systems Interconnection (Modell). 26

PID Process ID, Process Identifier. 28, 30

REST Representational State Transfer. 15, 49

rlimits Resource Limits. 34

SELinux Security Encanced Linux. 40

SSD Solid State Drive. 35

TLS Transport Layer Security. 23

UI User Interface. 23

USB Universal Serial Bus. 35

UTS UNIX Time Sharing. 31

VM Virtual Machine. 5, 6, 9, 11, 49

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 profile template for containers. über Website <https://github.com/docker/docker/tree/master/profiles/apparmor> , aufgerufen am 09.02.2016.
- [8] Apparmor profile template for the daemon. über Website <https://github.com/docker/docker/blob/master/contrib/apparmor/template.go> , aufgerufen am 03.02.2016.

- [9] Apparmor security profiles for docker. über Website <https://github.com/docker/docker/blob/master/docs/security/apparmor.md> , aufgerufen am 05.02.2016.
- [10] Apparmor wiki - quickprofilelanguage. über Website <http://wiki.apparmor.net/index.php/QuickProfileLanguage> , aufgerufen am 05.02.2016.
- [11] Cgroup unified hierarchy - documentation/cgroups/unified-hierarchy.txt. über Website <https://lwn.net/Articles/601923/> , aufgerufen am 27.01.2016.
- [12] 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.
- [13] Commit - fix proc regex. über Website <https://github.com/docker/docker/commit/2b4f64e59018c21aacf311d5c774dd5521b5352> , aufgerufen am 09.02.2016.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] Docker and selinux. über Website <http://www.projectatomic.io/docs/docker-and-selinux/> , aufgerufen am 05.02.2016.

- [18] Docker docs - registry. über Website <https://docs.docker.com/registry/> , aufgerufen am 18.01.2016.
- [19] Docker docs - understanding the architecture. über Website <https://docs.docker.com/engine/introduction/understanding-docker/> , aufgerufen am 14.01.2016.
- [20] Docker documentation - runtime metrics. über Website <https://docs.docker.com/engine/articles/runmetrics/> , aufgerufen am 27.01.2016.
- [21] Docker documentation - security. über Website <https://docs.docker.com/engine/articles/security/> , aufgerufen am 05.02.2016.
- [22] Docker documentation für den befehl `docker images`. über Website <https://docs.docker.com/engine/reference/commandline/images/> , aufgerufen am 21.01.2016.
- [23] Docker documentation für den befehl `docker pull`. über Website <https://docs.docker.com/engine/reference/commandline/pull/> , aufgerufen am 21.01.2016.
- [24] Docker documentation für den befehl `docker run`. über Website <https://docs.docker.com/engine/reference/run/> , aufgerufen am 27.01.2016.
- [25] Docker hub - explore. über Website <https://hub.docker.com/explore/> , aufgerufen am 15.01.2016.
- [26] Docker security profiles (seccomp, apparmor, etc) #17142. über Website <https://github.com/docker/docker/issues/17142#issuecomment-148974642> , aufgerufen am 05.02.2016.
- [27] *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.

- [28] Fixing control groups. über Website <https://lwn.net/Articles/484251/> , aufgerufen am 27.01.2016.
- [29] Freebsd - hierarchical resource limits. über Website https://wiki.freebsd.org/Hierarchical_Resource_Limits , aufgerufen am 27.01.2016.
- [30] 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.
- [31] Github repository changelog von docker. über Website <https://github.com/docker/docker/blob/master/CHANGELOG.md> , aufgerufen am 05.02.2016.
- [32] Github repository der cgroups-implementierung von runc. über Website <https://github.com/opencontainers/runc/tree/master/libcontainer/cgroups/fs> , aufgerufen am 27.01.2016.
- [33] Github repository der docker engine. über Website <https://github.com/docker/docker> , aufgerufen am 11.01.2016.
- [34] Github repository glossar von docker. über Website <https://github.com/docker/distribution/blob/master/docs/glossary.md> , aufgerufen am 21.01.2016.
- [35] Github repository roadmap von docker. über Website <https://github.com/docker/docker/blob/master/ROADMAP.md> , aufgerufen am 05.02.2016.
- [36] Github repository von *runC*. über Website <https://github.com/opencontainers/runc> , aufgerufen am 21.01.2016.
- [37] 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.

- [38] Homepage des kvm hypervisors und virtualisierungslösung. über Website http://www.linux-kvm.org/page/Main_Page , aufgerufen am 18.01.2016.
- [39] Homepage des vmware esxi hypervisors. über Website <https://www.vmware.com/de/products/esxi-and-esx/overview> , aufgerufen am 18.01.2016.
- [40] Homepage des xen hypervisors. über Website <http://www.xenproject.org/> , aufgerufen am 18.01.2016.
- [41] Homepage *Solaris* betriebssystem. über Website <http://www.oracle.com/de/products/servers-storage/solaris/solaris11/overview/index.html> , aufgerufen am 18.01.2016.
- [42] Homepage von *runC*. über Website <https://runc.io/> , aufgerufen am 21.01.2016.
- [43] 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.
- [44] Introducing runc: a lightweight universal container runtime. über Website <http://blog.docker.com/2015/06/runc/> , aufgerufen am 21.01.2016.
- [45] 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.
- [46] 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.
- [47] Linux programmer's manual - namespaces(7). über Website <http://man7.org/linux/man-pages/man7/namespaces.7.html> , aufgerufen am 28.01.2016.

- [48] 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.
- [49] Multi-tenant data architecture. über Website <https://msdn.microsoft.com/en-us/library/aa479086.aspx> , aufgerufen am 19.01.2016.
- [50] Offizielle dockerfile dokumentation. über Website <https://docs.docker.com/engine/reference/builder/#expose> , aufgerufen am 22.01.2016.
- [51] Offizieller twitter-account des docker-gründers, solomon hykes. über Website <https://twitter.com/solomonstre> , aufgerufen am 18.01.2016.
- [52] Offizielles repository des webserver nginx. über Website https://hub.docker.com/_/nginx/ , aufgerufen am 11.01.2016.
- [53] 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.
- [54] 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.
- [55] Phase 1: Initial seccomp support #17989. über Website <https://github.com/docker/docker/pull/17989> , aufgerufen am 05.02.2016.
- [56] Proposal: Support for user namespaces #7906. über Website <https://github.com/docker/docker/issues/7906> , aufgerufen am 28.01.2016.
- [57] Release notes von *FreeBSD V.4* und *Jails*. über Website <https://www.freebsd.org/releases/4.0R/notes.html> , aufgerufen am 19.01.2016.

- [58] Release notes von *Solaris 10*. über Website <https://docs.oracle.com/cd/E19253-01/pdf/817-0552.pdf> , aufgerufen am 19.01.2016.
- [59] Seccomp default profile. über Website <https://github.com/docker/docker/tree/master/profiles/seccomp> , aufgerufen am 09.02.2016.
- [60] Seccomp security profiles for docker. über Website <https://github.com/docker/docker/blob/master/docs/security/seccomp.md> , aufgerufen am 05.02.2016.
- [61] Selinux default policy profile. über Website <https://github.com/docker/docker/tree/master/contrib/docker-engine-selinux> , aufgerufen am 03.02.2016.
- [62] Slides of keynote at dockercon in san francisco - day 2. über Website de.slideshare.net/Docker/dockercon-15-keynote-day-2/16 , aufgerufen am 11.01.2016.
- [63] 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.
- [64] Softlayer benchmark, data sheet. über Website https://voltdb.com/sites/default/files/voltdb_softlayer_benchmark_0.pdf , aufgerufen am 14.01.2016.
- [65] 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.
- [66] The unified control group hierarchy in 3.16. über Website <https://lwn.net/Articles/601840/> , aufgerufen am 27.01.2016.
- [67] User namespaces - phase 1 #15187. über Website <https://github.com/docker/docker/issues/15187> , aufgerufen am 28.01.2016.
- [68] 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.
- [69] Voltdb homepage. über Website <https://voltdb.com/> , aufgerufen am 18.01.2016.
- [70] Was bedeutet public, private und hybrid cloud? über Website <http://www.cloud.fraunhofer.de/de/faq/publicprivatehybrid.html> , aufgerufen am 19.01.2016.
- [71] Überblick hyper-v hypervisor von microsoft. über Website <https://technet.microsoft.com/library/hh831531.aspx> , aufgerufen am 18.01.2016.
- [72] Übersicht zu *Solaris Zones*. über Website https://docs.oracle.com/cd/E24841_01/html/E24034/gavhc.html , aufgerufen am 18.01.2016.
- [73] 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.
- [74] Google code archive - go issue #8447. über Website <https://code.google.com/archive/p/go/issues/8447> , aufgerufen am 28.01.2016, 2014.
- [75] 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'.
- [76] 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'.

- [77] 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.
- [78] Charles Anderson. Docker. *IEEE Software*, 2015.
- [79] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, September 2014.
- [80] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. Technical report, IBM and Arastra, July 2008.
- [81] Thanh Bui. Analysis of docker security. Technical report, Aalto University School of Science, January 2015.
- [82] 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.
- [83] Rajdeep Duo, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. *IEEE International Conference on Cloud Engineering*, 2014.
- [84] 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.
- [85] Dr. Stefan Fischer et al, editor. *Security - IT-Sicherheit unter Linux von A bis Z*. Linux Magazine, 2008.
- [86] 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.

- [87] 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.
- [88] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Katalog B 3.304 Virtualisierung*, 2011.
- [89] Michael Kerrisk. *The Linux Programming Interface - A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.
- [90] 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.
- [91] 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.
- [92] Daniel J. Walsh (RedHat). Your visual how-to guide for selinux policy enforcement. über Website <https://opensource.com/business/13/11/selinux-policy-guide> , aufgerufen am 01.02.2016, November 2013.
- [93] 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.
- [94] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of os-level virtualization technologies. Technical report, Intel OTC Finland, Ericsson Finland, University of Helsinki, Aalto University Finland, July 2014.
- [95] 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.
- [96] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 3 edition, 2009.
- [97] James Turnbull. *The Docker Book*. 1.2.0 edition, September 2014.
- [98] 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.
- [99] 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.
- [100] 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.