

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

28. Januar 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	6
2.1.3	Einordnung Docker	9
2.2	Sicherheitsziele in der IT	9
2.2.1	Vertraulichkeit	10
2.2.2	Integrität	10
2.2.3	Authentizität	10
2.2.4	Verfügbarkeit	10
2.2.5	Verbindlichkeit	11
2.2.6	Privatheit, Anonymität	11
2.2.7	Authorisierung	11
2.3	Einführung in Docker	11
2.3.1	Docker Architektur	13
2.3.2	Dockerfile	14
2.3.3	Containerformate LXC, libcontainer, runC und OCF	16
2.3.4	Images	17
2.3.5	Container	18
2.3.6	Registries	20

3	Fragestellungen / Ziel der Arbeit	22
4	Security aus Linux Kernel-Features	25
4.1	Isolierung durch <code>namespaces</code>	25
4.1.1	Prozessisolierung durch den <code>PID namespace</code>	26
4.1.2	Dateisystemisolierung durch den <code>mount namespace</code> . .	27
4.1.3	Geräteisolierung durch	28
4.1.4	IPC-Isolierung durch den <code>IPC-namespace</code>	28
4.1.5	UTS-Isolierung durch den <code>UTS-namespace</code>	29
4.1.6	Netzwerkisolierung durch den <code>network namespace</code> . .	29
4.1.7	Userisolierung (user namespace)	30
4.2	Ressourcenverwaltung / Limitierung von Ressourcen durch <code>cgroups</code>	31
4.3	Einschränkungen von Zugriffsrechten	34
4.3.1	<code>capabilities</code>	34
4.3.1.1	Beispiele, <code>/proc</code> -Verzeichnis, (Un-)Mounten des Host-Filesystems	34
4.3.2	Linux Security Module (LSM) und Mandatory Access Control (MAC)	34
4.3.2.1	SELinux	34
4.3.2.2	AppArmor	34
4.3.2.3	Seccomp	34
4.4	Docker im Vergleich zu anderen Containerlösungen	34
5	Security im Docker-Ökosystem	35
5.1	Security Policies	36
5.2	Lifecycle- und State-Management von Containern	36
5.3	Docker Images und Registries	36
5.3.1	neues Signierungs-Feature	36
5.4	Docker Daemon	36
5.4.1	REST-API	36
5.4.2	Support von Zertifikaten	36
5.5	Containerprozesse	36

5.6	Docker Cache	36
5.7	privileged Container	36
5.8	Networking	36
5.8.1	bridge Netzwerk	36
5.8.2	overlay Netzwerk	36
5.8.3	DNS	36
5.8.4	Portmapping	36
5.9	Daten-Container	36
5.10	Docker mit VMs	36
5.11	Sicherheitskontrollen für Docker	36
5.12	Tools rund um Docker	36
5.12.1	Docker-Erweiterungen	36
5.12.1.1	Docker Swarm	36
5.12.1.2	Docker Compose	36
5.12.1.3	Nautilus Project	36
5.12.2	Third-Party Tools	36
5.12.3	Vagrant	36
5.12.4	Kubernetes	36
6	Docker in Unternehmen/Cloud-Infrastrukturen	38
7	Fazit	39

Abbildungsverzeichnis

1	Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[25].	2
2	Die Client-Server-Architektur von Docker [10].	14
3	Aufbau eines Docker-Hosts, wenn dieser unter einem Linux-Betriebssystem betrieben wird, das direkt auf der Serverhardware läuft. [58, S.3].	15
4	Dateien im Ordner eines Images (eigene Abbildung).	17
5	Visualisierung eines Vergleichs von Images von <i>Redis</i> , <i>Nginx</i> und <i>CentOS</i> auf Schichtebene [31].	19
6	Screenshot von der Ausführung des Befehls <code>docker pull <image></code> (eigene Abbildung).	19
7	Screenshot von der Ausführung des Befehls <code>docker images</code> (eigene Abbildung).	19
8	Web-UI des Docker Hubs mit den beliebtesten Repositories [15].	21

Tabellenverzeichnis

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 [67, S.1]. Vor allem in Rechenzentren bieten sich Virtualisierungen an, um die Serverressourcen effizienter zu nutzen [57, S.1]. Letztendlich haben es Virtualisierungen ermöglicht, Serverressourcen in der Form von Clouds wie z.B. den *Amazon Web Services*[4] und auf Basis eines Subskriptionsmodells nutzen zu können [57, S.1].

Heutzutage existieren mehrere serverseitige Virtualisierungstechniken, wovon die Hypervisor-gestützten Methoden mit den etablierten Vertretern *Xen*[28], *KVM*[26], *VMware ESXi*[27] und *Hyper-V*[51] die meistverbreitesten sind [67, 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 [19]. 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 [25].

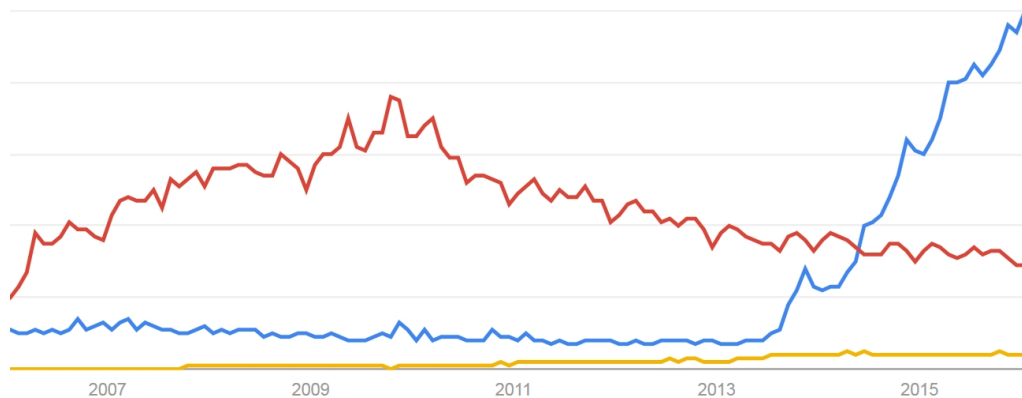


Abbildung 1: Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[25].

Obwohl das Konzept von Containern bereits im Jahr 2000 als *Jails* in dem Betriebssystem *FreeBSD* und seit 2004 als *Zones* unter *Solaris* verwendet wurde [43][42], gelang keiner dieser Technologien vor Docker der Durchbruch. Wie Docker den bis 2013 vorherrschenden Ruf von Containerertechnologien, dass Container noch nicht ausgereift seien [67, 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 [66, S.6][10].

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

Kapitel 2

Grundlagen

2.1 Virtualisierung

Bei der Virtualisierung 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.

Virtualisierte Komponenten nutzen im Vergleich zu nativen (physischen) Systemen eine zusätzliche Softwareschicht, die den virtualisierten Komponenten in der Ausprägung von virtuellen Maschinen (VMs) und Containern, mehrere Abstraktionen anbietet, um Funktionen des Hosts zu nutzen [67, 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 Gastssysteme bezeichnet.

Der Einsatz von Virtualisierung bietet vielfältige Vorteile für IT-Unternehmen. Sie können Kosten für Hardwarebeschaffung, Strom und Klimatisierung einsparen, wenn die Computerressourcen effizienter genutzt werden. Durch die damit verbundene Zentralisierung und Konsolidierung können auch in der Administration Ausgaben reduziert werden [62, S.1].

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 (der VM) vermittelt und Hardwareabstraktionen des ersteren bereitstellt [66, S.6][67, S.2][57, S.2].

Durch diese Technik läuft in jeder VM ein eigenes 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 [67, S.2].

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 [54, S.1][55, 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, während Typ 2 auf einem Host-OS agiert, welches selbst direkt auf die Hardware zugreift. Durch die Trennung von Hypervisor und Host-OS in der Architektur des Typs 2, ist dieser aus Sicht der Performance dem Typ 1 unterlegen [57, 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* [55, S.1].

2.1.2 Container-basierte Virtualisierung

Container-basierte Virtualisierung wird vorrangig als leichtgewichtige Alternative zu der Hypervisor-basierten Virtualisierung gesehen[67, S.2]. Erstere

nutzt direkt den Hostkernel, um virtuelle Umgebungen zu schaffen. Ein Hypervisor wird in diesem Ansatz nicht benötigt [66, 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 [67, S.2][58, S.3][64, 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 [58, S.4].

Container sind durch den Unix-Befehl *chroot*[33] 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 [16]. In *Solaris*, ein von der Firma *Oracle* entwickeltes Betriebssystem für Servervirtualisierungen[29], wurde dieser Mechanismus in Form von *Zones* (Solaris-spezifischer Begriff) [52] 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 [55, 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 [66, 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 [66, S.6+7][67, S.2][55, 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 [66, 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 [63].

Ein großer Vorteil jedoch, der sich durch das schlankere Design ergibt, ist eine fast nativen Performance [67, 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 [67, S.4][61, S.5]. In traditionellen Virtualisierungen beansprucht der Hypervisor allein etwa 10-20% der Hostkapazität [54, S.2][61, S.5]. Ein Benchmarktest, der den Durchsatz (Operationen pro Sekunde) eines *VoltDB*-Setups[49] von Hypervisor-basierte Cloudlösungen mit containerbasierten Cloudlösungen verglich, kam zu dem Ergebnis, dass die Containerlösung unter genanntem Gesichtspunkt sogar eine fünffache Leistung aufweist [46, S.2+3]. In der Praxis machen sich diese Verhältnisse an einer hohen Dichte an Containern bemerkbar und führen zu einer besseren Ressourcenausnutzung [66, 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* [64, 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 [66, 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 [66, 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 [55, 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 [66, S.7][67, S.1][55, 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.*[38], vormals *dotCloud Inc.*, sowie mehr als 1.600 freiwillig mitwirkenden Entwicklern ständig weiterentwickelt. [21][66, S.7][19][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 [66, S.7] und bietet neben Containern viele Tools und einen Workflow für Entwickler, die beide die Arbeit mit Containern erleichtern sollen [54, 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 [66, 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 [66, 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 [66, 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 [54, 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 [10]. 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 [58, 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.

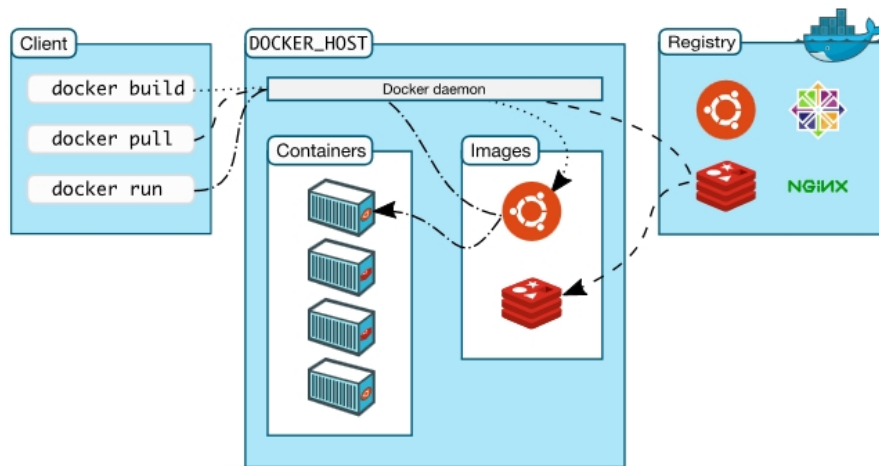


Abbildung 2: Die Client-Server-Architektur von Docker [10].

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
```

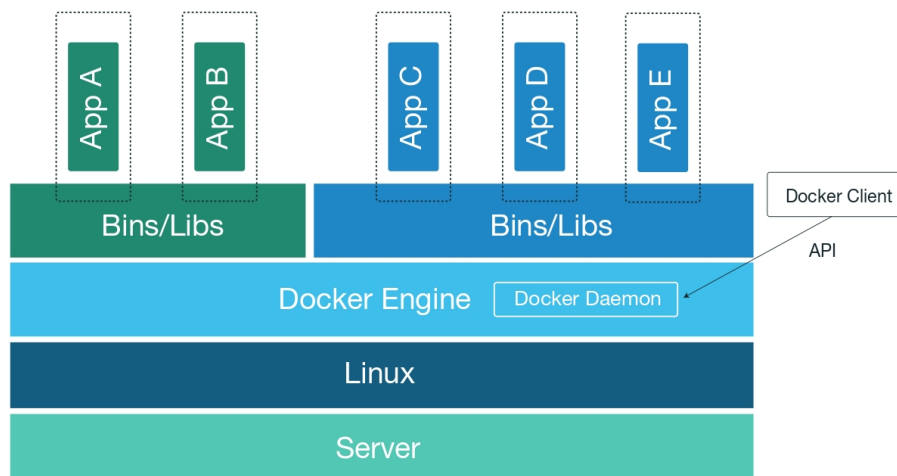


Abbildung 3: Aufbau eines Docker-Hosts, wenn dieser unter einem Linux-Betriebssystem betrieben wird, das direkt auf der Serverhardware läuft. [58, S.3].

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

```
EXPOSE 80
```

```
EXPOSE 443
```

Die Erklärung der einzelnen Anweisungen [37]:

- **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.

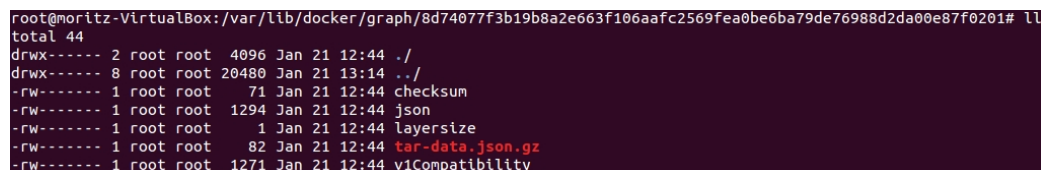
Im ersten Release von Docker wurde die Ausführungsumgebung *LXC* verwendet, die im März 2014 von der *Docker*-eigenen Entwicklung *libcontainer* abgelöst wurde. *libcontainer* ist komplett in der Programmiersprache *Golang* implementiert und kann ohne Dependencies mit dem Kernel kommunizieren [7].

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 [8]. 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 [32][24][30].

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 [66, S.11]. *Union*-Dateisysteme wie *AuFS* und *Device Mapper* haben gemeinsam, dass sie alle auf dem *Copy-on-write*-Modell basieren [66, S.8][54, S.3][58, 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 [22].



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

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

Images werden Schritt für Schritt erstellt, z.B. mit den folgenden Aktionen [66, 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 [54, 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 [39][13]. 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 [12].

2.3.5 Container

Ein Container ist die laufende Instanz eines Images, die in Sekundenbruchteilen gestartet werden kann [54, 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 [31].

```

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 [66, S.12][55, 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 [9]. 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 [66, S.11][57, S.3][9]. 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) [44, 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** [39]. 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

 Search

Sign up

Log In

Explore Official Repositories




	busybox official	434 STARS	58.8 M PULLS	➤ DETAILS
	ubuntu official	3.0 K STARS	37.7 M PULLS	➤ DETAILS
	swarm official	115 STARS	21.3 M PULLS	➤ DETAILS

Abbildung 8: Web-UI des Docker Hubs mit den beliebtesten Repositories [15].

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 [9].

Der Zugriff auf eine Registry kann über TLS und der Verwendung eines Zertifikats, sowie *basic authentication* abgesichert werden [9].

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 [64, 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 [54, 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 [56, S.1+2][34]. 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.

[57, 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 [58, 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 [58, S.4].

[57, 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.

[57, 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.

[57, 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 [59, 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 [59, 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 [56, 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 [59, 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.

[57, 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 unprivilegiertes User (ohne Root-Rechte) existieren, während der gleiche User innerhalb von Containern mit *user namespace* privilegiert ist, also im

Besitz von Root-Rechten ist [35].

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.

Das aktuelle Release von Docker unterstützt noch keine *user namespaces*, obwohl dieses Feature schon seit Version 1.6 geplant war [19][41]. Verzögerungen entstanden durch einen Bug der Programmiersprache *Golang* [53], und Integrationschwierigkeiten in die bestehende Docker-Codebasis [48].

Beide Probleme sind jedoch mittlerweile gelöst, wie die erfolgreiche Integration von *user namespaces* in Docker im Oktober 2015 bestätigt [40]. Dadurch, dass *user namespaces* in der Docker-Roadmap als wichtiges Sicherheitsfeature gesehen werden, ist ein Release dessen bald zu erwarten [23].

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 [60, 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 [57, 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 [18]. *Solaris* bietet die Nutzung von *Resource Pools* an, die eine Partitionierung von Ressourcen implementiert [5]. Auch *OpenVZ* und *Linux-VServer* erweitern **rlimits**, sodass Ressourcenlimits pro Container definiert werden können [64, 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 [6]. 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 [17][47].

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 [64, S.19]. Auch im Quellcode von *runC* ist eine Dateisystem-Interface als „nicht unterstützt“ gekennzeichnet und wird demzufolge auch nicht von Docker genutzt. [20]. 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 [3]. Die neusten Entwicklungen sehen jedoch eine Quota-Implementierung vor [2].

Dennoch ist diese Erweiterung im Sinne einer einheitlichen Verwaltung von Ressourcen mit **cgroups** vorgesehen [64, S.16+19].

Alle gängigen Linux-basierten Containerlösungen, darunter auch Docker, nutzen aktuell **cgroups**, um Ressourcen für Container zu verwalten [64, 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 [6][20].

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 [14].

```
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 [11].

4.3 Einschränkungen von Zugriffsrechten

4.3.1 capabilities

4.3.1.1 Beispiele, /proc-Verzeichnis, (Un-)Mounten des Host-Filesystems

4.3.2 Linux Security Module (LSM) und Mandatory Access Control (MAC)

4.3.2.1 SELinux

4.3.2.2 AppArmor

4.3.2.3 Seccomp

4.4 Docker im Vergleich zu anderen Containerlösungen

Kapitel 5

Security im Docker-Ökosystem

5.1 Security Policies

5.2 Lifecycle- und State-Management von Containern

5.3 Docker Images und Registries

5.3.1 neues Signierungs-Feature

5.4 Docker Daemon

5.4.1 REST-API

5.4.2 Support von Zertifikaten

5.5 Containerprozesse

36

5.6 Docker Cache

5.7 privileged Container

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 [55, 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. 15

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.

[50] . 1, 8, 12

Denial of Service *gescheite Quelle. Buch hier am besten..* 10

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 [65]. 12

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 [36]. 2, 31

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 [45]. 32

Abkürzungsverzeichnis

API Application Programming Interface. 13, 16, 36

ARP Address Resolution Protocol. 30

cgroups Control Groups. 31–33

COW Copy-On-Write. 27

CPU Central Processing Unit. 1, 25, 32, 33

DoS Denial of Service. 10, 31, *Glossary*: Denial of Service

HDD Hard Disk Drive. 33

HPC High Performance Computing. 8

HTTP Hypertext Transfer Protocol. 16

HTTPS Hypertext Transfer Protocol Secure. 16

I/O Input and Output. 33

IPC Inter Process Communication. 28, 29

IT Informationstechnik. 3, 5, 9, 13

JSON JavaScript Object Notation. 17

LSM Linux Security Model. 34

MAC Mandatory Access Control (nicht Netzwerkkommunikation). 34

MAC Media Access Control (Netzwerkkommunikation). 30

OCF Open Container Format. 16

OCP Open Container Project. 16

OS Operating System. 6–9

OSI Open Systems Interconnection (Modell). 24

PID Process ID, Process Identifier. 26, 28

REST Representational State Transfer. 13, 36

rlimits Resource Limits. 31, 32

SELinux Security Encanced Linux. 34

SSD Solid State Drive. 33

TLS Transport Layer Security. 21

UI User Interface. 21

USB Universal Serial Bus. 33

UTS UNIX Time Sharing. 29

VM Virtual Machine. 5–7, 9, 36

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 quota support for storage backends #3804. über Website <https://github.com/docker/docker/issues/3804> , aufgerufen am 28.01.2016.
- [4] Amazon web services. über Website <https://aws.amazon.com/de/> , aufgerufen am 14.01.2016.
- [5] Cgroup unified hierarchy - documentation/cgroups/unified-hierarchy.txt. über Website <https://lwn.net/Articles/601923/> , aufgerufen am 27.01.2016.
- [6] 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.
- [7] 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.

- [8] 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.
- [9] Docker docs - registry. über Website <https://docs.docker.com/registry/> , aufgerufen am 18.01.2016.
- [10] Docker docs - understanding the architecture. über Website <https://docs.docker.com/engine/introduction/understanding-docker/> , aufgerufen am 14.01.2016.
- [11] Docker documentation - runtime metrics. über Website <https://docs.docker.com/engine/articles/runmetrics/> , aufgerufen am 27.01.2016.
- [12] Docker documentation für den befehl `docker images`. über Website <https://docs.docker.com/engine/reference/commandline/images/> , aufgerufen am 21.01.2016.
- [13] Docker documentation für den befehl `docker pull`. über Website <https://docs.docker.com/engine/reference/commandline/pull/> , aufgerufen am 21.01.2016.
- [14] Docker documentation für den befehl `docker run`. über Website <https://docs.docker.com/engine/reference/run/> , aufgerufen am 27.01.2016.
- [15] Docker hub - explore. über Website <https://hub.docker.com/explore/> , aufgerufen am 15.01.2016.
- [16] *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.
- [17] Fixing control groups. über Website <https://lwn.net/Articles/484251/> , aufgerufen am 27.01.2016.

- [18] FreeBSD - hierarchical resource limits. über Website https://wiki.freebsd.org/Hierarchical_Resource_Limits , aufgerufen am 27.01.2016.
- [19] Github repository changelog von docker. über Website <https://github.com/docker/docker/blob/master/CHANGELOG.md> , aufgerufen am 28.01.2016.
- [20] Github repository der cgroups-implementierung von runc. über Website <https://github.com/opencontainers/runc/tree/master/libcontainer/cgroups/fs> , aufgerufen am 27.01.2016.
- [21] Github repository der docker engine. über Website <https://github.com/docker/docker> , aufgerufen am 11.01.2016.
- [22] Github repository glossar von docker. über Website <https://github.com/docker/distribution/blob/master/docs/glossary.md> , aufgerufen am 21.01.2016.
- [23] Github repository roadmap von docker. über Website <https://github.com/docker/docker/blob/master/ROADMAP.md> , aufgerufen am 28.01.2016.
- [24] Github repository von *runC*. über Website <https://github.com/opencontainers/runc> , aufgerufen am 21.01.2016.
- [25] 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.
- [26] Homepage des kvm hypervisors und virtualisierungslösung. über Website http://www.linux-kvm.org/page/Main_Page , aufgerufen am 18.01.2016.
- [27] Homepage des vmware esxi hypervisors. über Website <https://www.vmware.com/de/products/esxi-and-esx/overview> , aufgerufen am 18.01.2016.

- [28] Homepage des xen hypervisors. über Website <http://www.xenproject.org/> , aufgerufen am 18.01.2016.
- [29] Homepage *Solaris* betriebssystem. über Website <http://www.oracle.com/de/products/servers-storage/solaris/solaris11/overview/index.html> , aufgerufen am 18.01.2016.
- [30] Homepage von *runC*. über Website <https://runc.io/> , aufgerufen am 21.01.2016.
- [31] 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.
- [32] Introducing runc: a lightweight universal container runtime. über Website <http://blog.docker.com/2015/06/runc/> , aufgerufen am 21.01.2016.
- [33] 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.
- [34] Linux programmer's manual - namespaces(7). über Website <http://man7.org/linux/man-pages/man7/namespaces.7.html> , aufgerufen am 28.01.2016.
- [35] 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.
- [36] Multi-tenant data architecture. über Website <https://msdn.microsoft.com/en-us/library/aa479086.aspx> , aufgerufen am 19.01.2016.
- [37] Offizielle dockerfile dokumentation. über Website <https://docs.docker.com/engine/reference/builder/#expose> , aufgerufen am 22.01.2016.

- [38] Offizieller twitter-account des docker-gründers, solomon hykes. über Website <https://twitter.com/solomonstre> , aufgerufen am 18.01.2016.
- [39] Offizielles repository des webserver nginx. über Website https://hub.docker.com/_/nginx/ , aufgerufen am 11.01.2016.
- [40] 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.
- [41] Proposal: Support for user namespaces #7906. über Website <https://github.com/docker/docker/issues/7906> , aufgerufen am 28.01.2016.
- [42] Release notes von *FreeBSD V.4* und *Jails*. über Website <https://www.freebsd.org/releases/4.0R/notes.html> , aufgerufen am 19.01.2016.
- [43] Release notes von *Solaris 10*. über Website <https://docs.oracle.com/cd/E19253-01/pdf/817-0552.pdf> , aufgerufen am 19.01.2016.
- [44] Slides of keynote at dockercon in san francisco - day 2. über Website [de.slideshare.net/Docker/dockercon-15-keynote-day-2/16](https://www.slideshare.net/Docker/dockercon-15-keynote-day-2/16) , aufgerufen am 11.01.2016.
- [45] 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.
- [46] Softlayer benchmark, data sheet. über Website https://voltdb.com/sites/default/files/voltdb_softlayer_benchmark_0.pdf , aufgerufen am 14.01.2016.
- [47] The unified control group hierarchy in 3.16. über Website <https://lwn.net/Articles/601840/> , aufgerufen am 27.01.2016.
- [48] User namespaces - phase 1 #15187. über Website <https://github.com/docker/docker/issues/15187> , aufgerufen am 28.01.2016.

- [49] Voltdb homepage. über Website <https://voltdb.com/> , aufgerufen am 18.01.2016.
- [50] Was bedeutet public, private und hybrid cloud? über Website <http://www.cloud.fraunhofer.de/de/faq/publicprivatehybrid.html> , aufgerufen am 19.01.2016.
- [51] Überblick hyper-v hypervisor von microsoft. über Website <https://technet.microsoft.com/library/hh831531.aspx> , aufgerufen am 18.01.2016.
- [52] Übersicht zu *Solaris Zones*. über Website https://docs.oracle.com/cd/E24841_01/html/E24034/gavhc.html , aufgerufen am 18.01.2016.
- [53] Google code archive - go issue #8447. über Website <https://code.google.com/archive/p/go/issues/8447> , aufgerufen am 28.01.2016, 2014.
- [54] Charles Anderson. Docker. *IEEE Software*, 2015.
- [55] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, September 2014.
- [56] Sukadev Bhattiprolu, Eric W. Biederman, Serge Halryn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. Technical report, IBM and Arastra, July 2008.
- [57] Thanh Bui. Analysis of docker security. Technical report, Aalto University School of Science, January 2015.
- [58] 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.
- [59] Rajdeep Duo, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. *IEEE International Conference on Cloud Engineering*, 2014.

- [60] 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.
- [61] 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.
- [62] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Katalog B 3.304 Virtualisierung*, 2011.
- [63] 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.
- [64] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of os-level virtualization technologies. Technical report, Intel OTC Finland, Ericsson Finland, Univerisity of Helsinki, Aalto Univeristy Finland, July 2014.
- [65] 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.
- [66] James Turnbull. *The Docker Book*. 1.2.0 edition, September 2014.
- [67] 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.