

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

20. März 2016

Erstbetreuer:

Prof. Dr. Joachim Charzinski

Hochschule der Medien

Zweitbetreuer:

Patrick Fröger

ITI/GN, Daimler AG

Eidesstattliche Erklärung

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

Unterschrift

Datum

Abstract

In recent years, virtualization technologies have established themselves in server infrastructures. Not including hypervisor-based virtualization approaches, Docker, as a popular representative of container-based virtualization, has proved to be very successful in the market. Despite many attractive properties, container systems are considered questionable in terms of security due to their architecture. This paper analyzes built-in security models and mechanisms as well as cloud-centric integration capabilities of container technologies using the example of Docker.

Kurzfassung

Virtualisierungstechnologien haben sich über die letzten Jahre in Serverinfrastrukturen etabliert. Neben hypervisorbasierten Techniken hat sich mit dem Erfolg von Docker die containerbasierte Virtualisierung auf dem Markt behauptet. Abgesehen von vielen attraktiven Eigenschaften von Containersystemen, ist die Sicherheit dieser aufgrund ihrer Architektur fragwürdig. Diese Arbeit untersucht die verwendeten Sicherheitsmodelle und -mechanismen sowie Integrationsmöglichkeiten in Cloud-Infrastrukturen von Containertechnologien am Beispiel von Docker.

Inhaltsverzeichnis

1	Überblick	1
1.1	Ziel der Arbeit	3
1.2	Struktur der Arbeit	3
1.3	Stil der Arbeit	4
2	Grundlagen	6
2.1	Virtualisierung	6
2.1.1	Hypervisorbasierte Virtualisierung	9
2.1.1.1	Hypervisor Typ 1	10
2.1.1.2	Hypervisor Typ 2	10
2.1.2	Containerbasierte Virtualisierung	11
2.1.3	Einordnung Docker	13
2.2	Sicherheitsziele in der IT	13
2.2.1	Vertraulichkeit	14
2.2.2	Integrität	14
2.2.3	Verfügbarkeit	14
2.2.4	Authentizität	14
2.3	Einführung in Docker	15
2.3.1	Architektur	16
2.3.2	Dockerfile	17
2.3.3	Containerformate <i>LXC</i> , <i>libcontainer</i> und <i>OCF</i>	19
2.3.4	Image	19
2.3.5	Container	22
2.3.6	Registry	23

3	Fragestellung	25
3.1	Identifikation des Systems	26
3.2	Identifikation der Bedrohungen	27
3.3	Auswirkungen der identifizierten Risiken	28
3.4	Anforderungen an die Sicherheit von Containern	29
3.5	Theoretischer Lösungsansatz	30
3.6	Umsetzung des Lösungsansatzes	30
3.7	Ziel der Arbeit	31
4	Sicherheit durch Linux-Funktionen	32
4.1	Isolierung	33
4.1.1	Prozessisolierung durch den PID-Namespace	34
4.1.2	Dateisystemisolierung durch den Mount-Namespace	37
4.1.3	IPC-Isolierung durch den IPC-Namespace	38
4.1.4	Netzwerkisolierung durch den Network-Namespace	38
4.1.5	Userisolierung durch den User-Namespace	39
4.1.6	UTS-Isolierung durch den UTS-namespace	40
4.1.7	Geräteisolierung	41
4.2	Ressourcenverwaltung durch Control Groups	41
4.3	Einschränkung von Zugriffsrechten	44
4.3.1	Capabilities	46
4.3.2	Mandatory Access Control (MAC) und Linux Security Modules (LSMs)	47
4.3.2.1	AppArmor	49
4.3.2.2	SELinux	53
4.3.3	Seccomp	57
5	Sicherheit im Docker-Ökosystem	60
5.1	Private Registries	61
5.2	Verifikation und Verteilung von Images	61
5.2.1	Verifikation von Images	62
5.2.2	Integration von <i>The Update Framework</i>	63
5.3	Verbindung zwischen Daemon und Client	63

5.4	Docker Plugins	65
5.5	Open-Source-Charakter und Sicherheitspolitik von Docker . .	67
5.6	Best-Practices für die Sicherheit	69
5.6.1	Skripte	69
5.6.2	Datencontainer	70
5.6.3	Verwaltung von Credentials	71
5.7	Tools	71
5.7.1	Kubernetes	72
5.7.2	docker-slim	72
5.7.3	Bane	73
6	Docker in Cloud-Infrastrukturen	75
6.1	Public Cloud	76
6.1.1	Beispiel: Microsoft Azure	77
6.1.2	Beispiel: IBM SoftLayer	78
6.2	Private Cloud	79
6.2.1	Beispiel: OpenStack	80
6.3	Hybrid Cloud	82
7	Fazit	83

Abbildungsverzeichnis

1	Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[54].	2
2	Die Client-Server-Architektur von Docker [27].	16
3	Aufbau eines Docker-Hostsystems [141, S.3].	17
4	Dateien im Ordner eines Images (eigene Abbildung).	20
5	Vergleich von drei Images auf Schichtebene [68].	21
6	Screenshot von der Ausführung des Befehls <code>docker pull IMAGE</code> (eigene Abbildung).	22
7	Screenshot von der Ausführung des Befehls <code>docker images</code> (eigene Abbildung).	22
8	Schichtenaufbau eines Containers [26].	23
9	Web-UI des Docker Hubs mit den beliebtesten Repositories [35].	24
10	Prozesshierarchie unter Verwendung von PID-Namespaces (eigene Abbildung auf Basis von [180]).	35
11	Ausschnitt der Ausgabe des Befehls <code>ps -eafxZ</code> auf einem Docker-Hostsystem (eigene Abbildung).	36
12	Ausgabe des Befehls <code>ps -eafxZ</code> in einem Docker-Container (eigene Abbildung).	36
13	Funktionsweise von <i>System Call</i> -Hooks im LSM-Framework [199, S.3].	48
14	Trennung von Regelwerk und Enforcement-Modul. Zuweisung von Security-Contexts (SC) an Objekte und Subjekte [151, S.63].	55

Tabellenverzeichnis

1	Herausforderungen und Lösungsansätze durch Virtualisierung in Rechenzentren.	7
2	Anweisungen eines Dockerfiles und deren Bedeutung.	18

Kapitel 1

Überblick

Die 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 Programmen genutzt wird.

Die Vorteile von Virtualisierung umfassen Hardwareunabhängigkeit, Verfügbarkeit, Isolierung und Sicherheit, welche die Erfolgsgrundlage von heutigen Cloud-Infrastrukturen in Rechenzentren bilden [200, S.1]. Letztendlich haben es Virtualisierungslösungen ermöglicht, Dienste von Clouds, wie z.B. den *Amazon Web Services* [5], auf Basis eines Subskriptionsmodells nutzen zu können [132, S.1].

Heutzutage existieren mehrere serverseitige Virtualisierungstechniken, wovon die hypervisor-gestützten Methoden mit den etablierten Vertretern *Xen* [57], *KVM* [55], *VMware ESXi* [56] und *Hyper-V* [118] die meistverbreitetsten sind [200, S.2]. Die alternative containerbasierte Virtualisierung, auch Virtualisierung auf Betriebssystemebene genannt, 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 [48]. Wie die *Google Trends* in Abb.1 zeigen, stieg das Interesse an Docker seit dessen Release kontinuierlich an. Das Interesse an der Containertechnologie *LXC*, aus der Docker entstand, bleibt weit hinter der von Docker zurück. Das Suchwort „Virtualization“ erlebte im Jahr 2010 seinen Höhepunkt hat seitdem an Popularität verloren [54]. Obwohl Docker eine Art von Virtualisierungstechnologie ist, wird der Begriff der Virtualisierung meist mit traditioneller hypervisorbasierter Virtualisierung verbunden. Der Virtualisierungsaspekt von Docker wird dagegen meist mit Containern beschrieben.

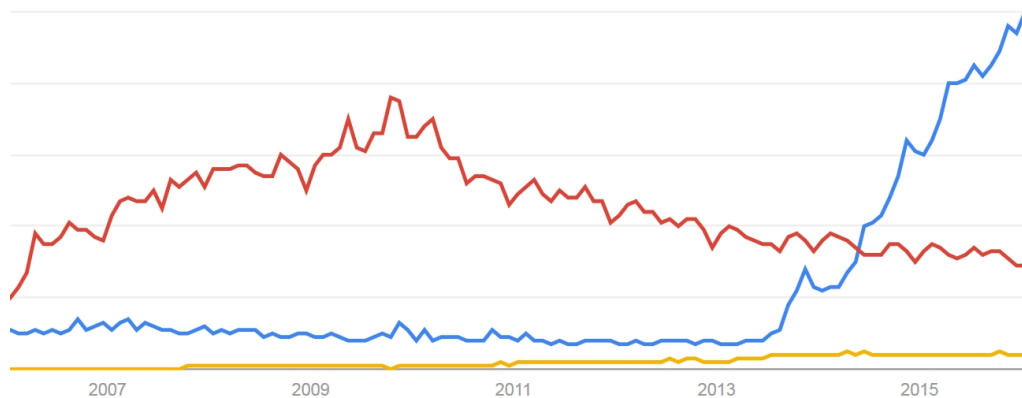


Abbildung 1: Google Trends der Suchbegriffe „Virtualization“ (rot), „Docker“ (blau) und „LXC“ (gelb) von Januar 2006 bis Januar 2016[54].

Obwohl das Konzept von Containern bereits im Jahr 2000 als *Jails* in dem Betriebssystem *FreeBSD* und seit 2004 als *Zones* unter *Solaris* verwendet wurde [97][96], erreichte keiner dieser Technologien vor Docker den Durchbruch. Wie Docker den bis 2013 vorherrschenden Ruf von Containertechnologien, dass Container noch nicht ausgereift seien [200, S.8], nachhaltig verändern konnte, ist in der Einführung zu Docker in Kapitel 2.3 beschreiben.

Heute sind Container in vielen Szenarien, v.a. skalierbaren Infrastrukturen, trotz intrinsischer Sicherheitsdefizite gegenüber hypervisorgestützten Virtualisierungsarten beliebt. Docker eignet sich z.B. bei der Realisierung von Multi-Tenant-Services [196, S.6][27].

1.1 Ziel der Arbeit

Ziel der Arbeit ist es, die von der Container-Technologie Docker genutzten Sicherheitsmechanismen zu untersuchen. Die sicherheitsrelevanten Modelle und Mechanismen werden im Rahmen der Arbeit vorgestellt und es wird erörtert, wie diese zu einer höheren Sicherheit in Docker-Systemen beitragen. Außerdem werden Integrationsmöglichkeiten von Docker sowie sicherheitsrelevante Merkmale im Kontext von Cloud-Infrastrukturen dargestellt.

Eine ausführliche Konstruktion der Fragestellung erfolgt mithilfe einer Risikoanalyse in Kapitel 3.

1.2 Struktur der Arbeit

Zu Beginn wird im Grundlagenkapitel 2 ein Wortschatz eingeführt, der grundlegende Merkmale der Virtualisierung, der IT-Sicherheit sowie von Docker definiert.

Genauer werden die zwei prominentesten Virtualisierungstechniken, die hypervisorbasierte (Abschnitt 2.1.1) und containerbasierte (Abschnitt 2.1.2) Virtualisierung, gegenübergestellt. Andere Arten, z.B. die Anwendungs-, Speicher- oder Netzwerkvirtualisierung, werden nicht behandelt, da sie isoliert keinen Bezug zu Docker haben. Anschließend werden die allgemeinen Sicherheitsziele von IT-Systemen in Kapitel 2.2 erklärt, auf die im Hauptteil der Arbeit Bezug genommen wird. Abgeschlossen wird das Grundlagenkapitel mit einer Einführung in Docker (Kapitel 2.3), in dem Begriffe sowie Funktionsweisen innerhalb dieser Technologie erläutert werden.

Die genannten Grundlagenthemen sind sehr weitreichende Themengebiete. Um in den einleitenden Kapiteln nicht ausführlich zu werden, sind Eckdaten einiger, nicht im Fokus der Arbeit stehende Begriffe im angehängten Glossar geschildert.

In Kapitel 3 wird die Fragestellung auf Basis einiger Annahmen hergeleitet.

Es wird ein Systemmodell eingeführt, das als Referenz für die Untersuchungen in Kapitel 4 herangezogen wird.

Der Hauptteil von Kapitel 4 bis 6 untergliedert sich in mehrere Gebiete, in die die Arbeit eingeteilt ist:

1. **Kapitel 4 - Sicherheit durch Linux-Funktionen:** Vorstellung von Linux-Funktionen, die die Isolierung, Ressourcenverwaltung und Zugriffskontrolle ermöglichen. Darstellung der Integration dieser Mechanismen unter Docker.
2. **Kapitel 5 - Sicherheit im Docker-Ökosystem:** Erörterung sicherheitsrelevanter Merkmale im Docker-Ökosystem. Umfasst u.a. die Verifikation von Docker-Images, Plugins und die Unterstützung von Tools.
3. **Kapitel 6 - Docker in Cloud-Infrastrukturen:** Beschreibung der Integrationsmöglichkeiten von Docker in verschiedene Cloud-Infrastrukturen. Erklärung sicherheitsrelevanter Merkmale dieser Infrastrukturen.

Abgeschlossen wird die Arbeit im letzten Kapitel 7 mit einer Zusammenfassung der Erkenntnisse. Außerdem werden aktuelle Trends sowie mögliche zukünftige Entwicklungen in der containerbasierten Virtualisierung, insbesondere von Docker, vorgestellt.

1.3 Stil der Arbeit

Da die meisten Quellen in der englischen Sprache verfasst sind, hat sich auch im deutschen Sprachgebrauch eine englische Terminologie von Gegenständen der IT etabliert. Zusammen mit der Tatsache, dass die Übersetzung englischer Begriffe häufig nicht das Verständnis in deutschen Texten fördert, sind einige Begriffe original in englisch gehalten.

In der Arbeit vorkommende Produkt-, Technologie-, Bibliothek- und Unternehmensnamen sowie Programmiersprachen und Konzepte der IT sind durchgehend *kursiv* gedruckt. Eine Ausnahme bildet Docker, in der die re-

gulgäre Schreibweise für die Technologie Docker vorgesehen ist, während die kursive Variante das Unternehmen *Docker* meint.

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

Kapitel 2

Grundlagen

2.1 Virtualisierung

Der Einsatz von Virtualisierung bietet vielfältige, attraktive Eigenschaften für IT-Unternehmen. In der folgenden Auflistung sind einige Problemfaktoren und Anforderungen an Rechenzentren und darin laufender Software zusammengefasst, die alle durch Virtualisierungslösungen adressiert werden können [156, S.1][195, S.662,672f.][169, S.299].

Die Virtualisierungsmerkmale aus Tabelle 1 eröffneten IT-Unternehmen neue Geschäftsmodelle, auf denen der Erfolg von heutigen Cloud-Anbietern wie *Amazon*, *Google* und *Microsoft* beruht. Selbst einhergehende Nachteile der Virtualisierung, wie z.B. Leistungsverluste und in dieser Arbeit untersuchte Sicherheitsrisiken, scheinen, basierend auf dem Erfolg genannter Anbieter, nicht ins Gewicht zu fallen.

Die Absicht der in dieser Arbeit relevanten Virtualisierung ist es, die in Tabelle 1 genannten Vorteile zu erreichen, indem ein System in ein oder mehrere Systeme aufgeteilt wird. Per Definition besteht ein System aus allen Ressourcen, die eine Anwendung benötigt, um fehlerfrei laufen zu können [195, S.106]. Technisch gesehen beinhaltet es ein Betriebssystem und zugehörige

Herausforderung	Lösungsansatz mithilfe von Virtualisierung
Effizienter Betrieb von Rechenzentren	Einsparungen bei Anschaffung von Hardware und Lizenzen. Bessere Auslastung physischer Ressourcen.
Skalierbarkeit und Redundanz von Diensten	Virtuelle Systeme sind i.d.R. hardwareunabhängig, portierbar und reproduzierbar. Damit sind Skalierbarkeit und Redundanz realisierbar.
Realisierung von Multi-Tenancy-Umgebungen	Abgrenzung von Kundeninstanzen durch Isolierung.
Realisierung bestimmter Architekturen, wie z.B. einer <i>3-Tier-Architektur</i>	Reproduzierbarkeit, flexible Kopplung mit unabhängigen Netzwerkfunktionen von virtuellen Systemen.
Unterstützung von Legacy-Code und mehrerer Software-Versionen	Gute Migrationseigenschaften, hoher Grad an Kompatibilität, Unabhängigkeit von Hardware und Betriebssystemen z.B. mit <i>Virtual Appliances</i> möglich.
Administration und Wartbarkeit	Zentralisiert für physische und virtuelle Systeme möglich, auch mit Unterstützung von Tools. Senkung von Administrationskosten [156, S.1][195, S.661].

Tabelle 1: Herausforderungen und Lösungsansätze durch Virtualisierung in Rechenzentren.

Peripheriegeräte, wie z.B. eine Netzwerkkarte.

Systeme können auf mehrere Arten unterschieden werden, z.B. in physische und virtuelle Systeme:

- **Physisches System:** System, das direkt auf der Hardware läuft.
- **Virtuelles System:** System, das virtuell innerhalb eines anderen physischen oder virtuellen Systems läuft.

Eine weitere Unterscheidung trennt Systeme nach ihrer Rolle:

- **Hostsystem:** Physisches oder virtuelles System, das virtuelle Systeme (Gastsysteme) betreibt. In einem Hostsystem läuft eine Virtualisierungssoftware wie z.B. Docker, die die Instanziierung von Gastsystemen ermöglicht.

Beispiel: Linux-Distribution mit Docker-Installation

- **Gastsystem:** Virtuelles System, das im Kontext eines übergeordneten Hostsystems läuft.

Beispiel: Docker-Container

Das Hostsystem ist nicht mit einem Host im Sinne der Netzwerktechnik zu verwechseln. Der Host eines Netzwerks meint einen kommunikationsfähiger Netzwerkknoten. Sowohl Host- als auch Gastsysteme können Hosts eines Netzwerks darstellen.

Wie diese Definition andeuten, sind virtuelle Verschachtelungen innerhalb eines physischen Hostsystems möglich und auch in der Praxis begrenzt sinnvoll. Beispielsweise kann in einem Rechenzentrum, jeder physische Host mehreren Kunden dienen, indem jedem Kunden ein Gastsystem auf diesem physischen Host zugewiesen wird. Ein Kunde kann dieses Gastsystem wiederum als virtuelles Hostsystem betreiben, das jede Anwendung des Kunden in einem eigenen, neuen Gastsystem virtualisiert. Wie diese Trennung von Kunden und Kundenanwendungen zeigt, eignet sich eine Verschachtelung der virtuellen Systeme zur Abbildung von organisatorischen und operationalen Strukturen.

Gastssysteme existieren in zwei Ausprägungen:

- **Hypervisorbasierte Gastssysteme:** Ein Gastsystem, das durch einen Hypervisor des übergeordneten Hostsystems instanziiert wird. Ein hypervisorbasiertes Gastssystem wird als virtuelle Maschine, kurz VM, bezeichnet.
- **Containerbasierte Gastssysteme:** Ein Gastsystem, das von einer auf dem Hostsystem installierten Containertechnologie erstellt wurde. Ein containerbasiertes Gastsystem wird abgekürzt als Container bezeichnet.

Beide Ausprägungen von erwecken aus Sicht des Gastsystems den Eindruck, dass ein alleinstehendes, direkt auf der Hardware laufendes System ausgeführt wird. Die Art und Weise, wie beide Arten Funktionen eines übergeordneten Hostsystems abstrahieren, weist Unterschiede auf, die in Abschnitt 2.1.1 und 2.1.2 näher beschrieben sind.

2.1.1 Hypervisorbasierte Virtualisierung

Im Kontext einer hypervisorbasierten Virtualisierung wird das Gastsystem eine VM genannt. VMs enthalten jeweils eine Umgebung, die Abstraktionen eines sogenannten Hypervisors nutzt, um Hardwareressourcen des Hostsystems zu verwenden. Der Hypervisor, auch seltener *Virtual Machine Monitor* (*VMM*) genannt, ist eine Software, die zwischen dem Host- und Gastsystem vermittelt und Abstraktionen des ersteren bereitstellt [196, S.6][200, S.2][195, S.105].

Durch diese Technik läuft in jeder VM ein eigenes Betriebssystem, das von solchen anderer VMs isoliert läuft. Durch die Abstraktion des zwischenliegenden Hypervisors ist es möglich, mehrere unterschiedliche Gastssysteme auf einem Hostsystem auszuführen [200, S.2][195, S.106].

Trotz der in Tabelle 1 aufgeführten effizienteren Ressourcennutzung im Rahmen von Virtualisierungstechniken, stehen Hypervisor heute unter dem Ruf ineffizient zu arbeiten. Dieser größte Kritikpunkt der genannten Virtuali-

sierungsmethode, lässt sich auf den Erfolg der containerbasierten Virtualisierung zurückführen, die, wie in Kapitel 2.1.2 weiter ausgeführt, weniger Leistungsverluste erzeugt.

Bekannte hypervisorbasierte Technologien sind die kommerziellen Vertreter *ESXi* der Firma *VMware* und *Hyper-V* von *Mircosoft*, sowie die Open-Source-Vertreter *Xen* und *KVM* [130, S.1].

Generell werden Hypervisor in solche von Typ 1 und Typ 2 unterschieden.

2.1.1.1 Hypervisor Typ 1

Der Typ 1 Hypervisor operiert direkt auf der Hardware des Hosts und stellt ein minimales, speziell für den Betrieb von VMs ausgelegtes Betriebssystem dar. Dessen Aufgabe ist es, Kopien der realen Hardware bereitzustellen, die von VMs genutzt werden können. Wenn in der VM ein Befehl ausgeführt, wird dieser an den Hypervisor weitergeleitet, der den Befehl untersucht. Handelt es sich um einen Befehl des Betriebssystems der VM, wird dieser auf dem Hostsystem ausgeführt. Im Fall eines Aufrufs einer Anwendung innerhalb der VM, emuliert der Hypervisor für diesen Aufruf die Aktion der realen Hardware [195, S.663ff.].

2.1.1.2 Hypervisor Typ 2

Hypervisor des Typs 2 arbeiten nicht direkt auf der Hardware, sondern innerhalb eines Hostsystem, das wiederum selbst direkt auf die Hardware zugreift. In dieser Variante ist der Hypervisor eine gewöhnliche Anwendung, die auf dem Hostsystem ausgeführt wird.

Falls ein Aufruf aus dem Gastsystem einen Hardwarezugriff benötigt, wird die Hardware - wie auch bei Typ 1 - emuliert. Erfordert ein Aufruf keinen Hardwarezugriff, wird dieser innerhalb des Gastsystems verarbeitet und verlässt dieses nicht.

Im Durchschnitt führt die zusätzliche Softwareschicht des Hypervisors Typ 2 trotz verschiedener Optimierungsmöglichkeiten, z.B. der sogenannten Paravirtualisierung, zu höheren Leistungseinbußen wie jenen unter Typ 1 [195, S.666f.].

2.1.2 Containerbasierte Virtualisierung

Die containerbasierte Virtualisierung wird vorrangig als leichtgewichtige Alternative zu der hypervisor-gestützten Virtualisierung gesehen [200, S.2].

Während ein Hypervisor für jede VM ein vollständiges System abstrahiert, werden für Container direkt Funktionen des Hostsystems über *System Calls* zur Verfügung gestellt. Containern kommunizieren über diese direkt mit dem Hostsystem und teilen sich den Kernel dessen [196, S.6f.][200, S.2]. Ein Hypervisor wird in diesem Ansatz nicht benötigt. Vielmehr wird das Hostsystem und dessen Ressourcen partitioniert, sodass mehrere virtuelle, voneinander isolierte Container betrieben werden können [141, S.3][179, S.1].

Die Isolation basiert auf dem Konzept von Kontexten, die unter Linux *Namespace*s genannt werden. Diese, sowie *Control Groups*, die für das Ressourcenmanagement verantwortlich sind, werden in den Kapiteln 4.1 und 4.2 genauer betrachtet [141, S.4].

Container sind durch den Unix-Befehl *chroot* 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 [43]. In *Solaris*, ein von dem Unternehmen *Oracle* entwickeltes Betriebssystem für Servervirtualisierungen [58], wurde dieser Mechanismus in Form von *Zones* (*Solaris*-spezifischer Begriff) [119] weiter verbessert. Auch proprietäre Lösungen von *HP* und *IBM* erschienen auf dem Markt [130, S.2].

Durch die kontinuierliche Weiterentwicklung von Containern in den letzten Jahren, können diese heutzutage laut [196, S.7] als vollwertige Systeme betrachtet werden, nicht mehr als - wie ursprünglich vorgesehen - reine Ausführungsumgebungen auf Basis von *chroot*.

Dieses Design hat einen entscheidenden Nachteil gegenüber dem Hypervisor-Modell, der auch Docker betrifft: Das Betriebssystem im Hostsystem muss Linux-basiert sein [196, S.6]. Allerdings streben *Docker* und *Microsoft* eine native Docker-Lösung für *Microsoft Windows Server 2016* an [177].

Ein großer Vorteil jedoch, der sich durch das schlanke Design von Containern ergibt, ist eine fast native Performance dieser [200, S.1], da der Hypervisor in diesem Modell nicht existiert. Unter dem Gesichtspunkt der Rechenleistung z.B., kommt es bei Containerlösungen im Durchschnitt zu einem Verlust von ca. 4%, wenn dieser mit der nativen Leistung derselben Hardwarekonfiguration verglichen wird [200, S.4][153, S.5]. In traditionellen Virtualisierungen beansprucht der Hypervisor allein etwa 10-20% der Hostkapazität [129, S.2][153, S.5]. Ein Benchmarktest, der den Durchsatz (Operationen pro Sekunde) einer *VoltDB*-Konfiguration [116] auf hypervisor- und containerbasierten Systemen gegenüberstellt, kommt zu dem Ergebnis, dass Container eine fünffache Leistung aufweisen [106, S.2f.].

In der Praxis ermöglichen diese Verhältnisse eine hohe Dichte an Containern und führen zu einer besseren Ressourcenausnutzung [196, S.7f.]. Der resultierende Leistungsgewinn ist v.a. für *High Performance Computing*, sowie ressourcenbeschränkte Umgebungen wie mobile Geräte und *Embedded Systems*, wichtig [179, S.1].

Aus der Sicht der Sicherheit kann das Fehlen eines Hypervisors kontrovers interpretiert werden: Zum einen schrumpft die Angriffsfläche des Hostsystems, da nicht das gesamte Betriebssystem virtualisiert wird [196, S.6]. Je weniger Hostfunktionen virtualisiert werden, desto geringer wird auch das Sicherheitsrisiko, dass eine Hostfunktion von einem Angreifer missbraucht wird. Zum anderen ist das Konzept eines geteilten Kernels generell unsicherer, da die Sicherheitsziele hierbei durch zusätzliche Mechanismen gewahrt werden müssen.

Die angewandten Sicherheitsmodelle und -mechanismen werden ausführlich in Kapitel 4 behandelt.

Auch im Lifecycle von virtuellen Instanzen bieten Container Vorteile: Während

in traditionellen VMs ein Neustart Sekunden bis Minuten beansprucht, entspricht der Neustart von Containern nur einem Prozessneustart auf dem Hostsystem, der im Millisekundenbereich abgeschlossen ist [130, S.2]. Im Fall von VMs muss das komplette Gastbetriebssystem neu gestartet werden.

2.1.3 Einordnung Docker

Docker zählt zu den Technologien der containerbasierten Virtualisierung und hat seinen Ursprung in *Linux Container (LXC)*, das mit Docker rundum erweitert wurde. [196, S.7][200, S.1][130, S.2].

Docker ist wie in Kapitel 2.1.2 zuvor erwähnt, nicht die erste containerbasierte Virtualisierungslösung. Einige ältere Containersysteme, wie z.B. *Solaris Zones*, existieren bereits länger als Docker, erlangten allerdings nie die Popularität von Docker.

2.2 Sicherheitsziele in der IT

In der IT existieren mehrere Sicherheitsziele, die auf unterschiedliche Art und Weise erreicht werden. Je nach Anwendungsgebiet und Anforderungen werden die einzelnen Ziele unterschiedlich stark priorisiert bzw. nicht angewandt. Auch im Zusammenhang mit der Virtualisierung lassen sich die Sicherheitsziele eingrenzen, sodass in den folgenden Abschnitten nur die in dieser Arbeit relevanten Ziele aufgeführt sind.

Grundsätzlich wird zwischen der Sicherheit von Computersystemen und Netzwerken unterschieden. Die Virtualisierung auf Containerbasis ist vorrangig ein Element von Computersystemen und ist von den Netzwerken, an die Systeme angeschlossen sind, zunächst unabhängig. Aus diesem Grund sind in diesem Kapitel nur die von Tanenbaum definierten Sicherheitsziele definiert [195, S.712f.].

2.2.1 Vertraulichkeit

Die Vertraulichkeit steht für das Konzept von Geheimhaltung. In Computersystemen können die z.B. in Kapitel 4 beschriebenen Mechanismen von Linux aktiviert werden, um eine bestimmte Information vor unauthorisierten Zugriffen zu schützen. Somit haben ausschließlich Befugte Zugang zu kritischen Information. In einem Multi-Tenancy-System ist es beispielsweise notwendig, Kundeninformationen von unberechtigten Nutzern geheimzuhalten.

2.2.2 Integrität

Unter Integrität versteht man die Zusicherung, dass bestimmte Daten original und vollständig vorliegen, sowie nachweisbar nicht manipuliert wurden. Ein Sicherheitsmechanismus, der die Integrität sicherstellen soll, bewirkt z.B., dass es nur dem Besitzer einer Datei möglich ist, diese zu modifizieren. In Multi-Tenancy-Systemen muss beispielsweise gewährleistet werden, dass die Daten der einzelnen Kunden geschützt werden und kein Kunde die Integrität von Daten anderer Kunden verletzen kann.

2.2.3 Verfügbarkeit

Die Verfügbarkeit bezeichnet die Eigenschaft eines Systems, Anfragen jederzeit zu verarbeiten und andere Systeme nicht negativ zu beeinflussen. Angriffe gegen die Verfügbarkeit werden als Denial of Service-Angriffe bezeichnet. Z.B. muss in Multi-Tenancy-Systemen sichergestellt sein, dass ein Kunde nicht die Ressourcen anderer Kunden erschöpft.

2.2.4 Authentizität

Authentizität bezeichnet die Sicherstellung der Echtheit und Überprüfbarkeit. I.d.R. wird die Identifikation eines Nutzers im Rahmen einer Authentifizierung auf Echtheit geprüft. In einem System muss z.B. sichergestellt werden,

dass nur bestimmte Nutzer Zugriff auf eine sicherheitskritische Ressource erlangen.

2.3 Einführung in Docker

Docker ist eine unter der Apache 2.0-Lizenz veröffentlichte, quelloffene Technologie, die neben dem Betrieb von Containern auch Möglichkeiten für die Konfiguration und Automatisierung dieser anbietet. Sie ist überwiegend in der Programmiersprache *Golang* implementiert und wird seit ihrem ersten Release im März 2013 von dem von Solomon Hykes gegründeten Unternehmen *Docker, Inc.*[80] sowie mehr als 1.600 freiwillig mitwirkenden Entwicklern erweitert. [50][196, S.7][48][1].

Der große Vorteil von Docker gegenüber älteren Containerlösungen, wie z.B. dem Docker-Vorgänger *LXC*, ist der Grad 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 [196, S.7] und bietet parallel viele Tools für Entwickler an. Auch die IT-Konzepte *Continuous Delivery*, *Continuous Integration*, *Immutable Infrastructure*, *Microservices* und *DevOps*-Teams werden durch die Eigenschaften von Docker begünstigt.

Je nach Anwendungsfall können Container Testumgebungen, Anwendungen bzw. Teile davon, oder Replikate komplexer Anwendungen für Entwicklungs- und Produktionszwecke abbilden. Container nehmen dadurch die Rolle austauschbarer, kombinierbarer und portierbarer Module einer Architektur ein, die mithilfe besagter IT-Konzepte automatisierbar ist [196, S.12].

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 von Docker: Images, Container und Registries.

2.3.1 Architektur

Die Architektur von Docker ist nach einem Client-Server-Modell aufgebaut: Ein Docker-Client kommuniziert mit einem Docker-Daemon, der den Server abbildet [27]. Beide Teile können auf einer Maschine oder einzeln auf unterschiedlichen Hosts laufen. Die Kommunikation zwischen Client und Daemon geschieht auf Basis von HTTP über eine RESTful API. Wie Abb.2 zeigt, ist es dadurch auch möglich Befehle entfernter Clients über ein Netzwerk an den Daemon zu senden [141, S.3].

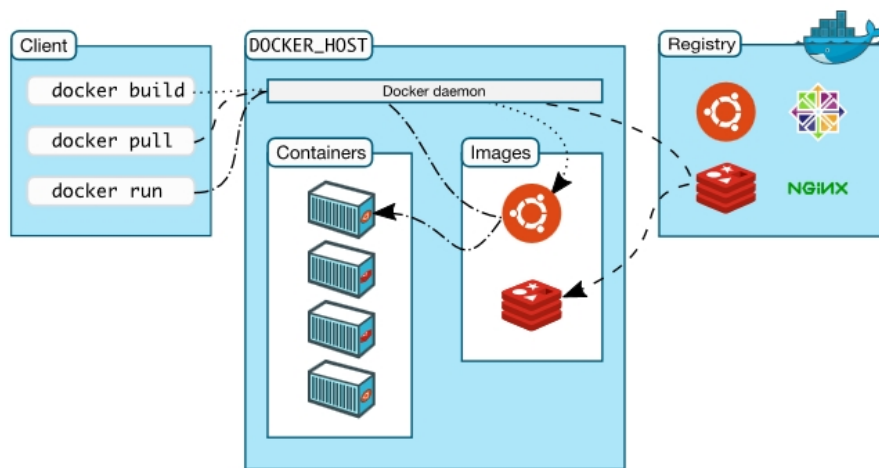


Abbildung 2: Die Client-Server-Architektur von Docker [27].

Der Daemon kann von einer Registry Images (siehe Abschnitt 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. In einem Hostsystem läuft ein Linux-Betriebssystem, auf dem die Docker-Engine installiert ist. Die Engine verwaltet den Betrieb von Containern (siehe Abschnitt 2.3.5), in denen in Abb.3 die Applikationen A-E laufen. Wie auch in der Grafik zu sehen ist, teilen sich die Container gemeinsam verwendete Bibliotheken. Diese Eigenschaft wird in Abschnitt 2.3.4 näher untersucht.

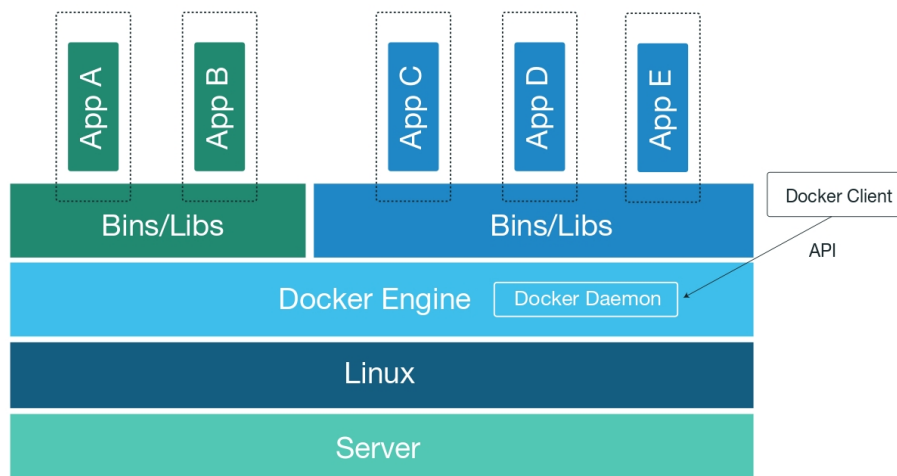


Abbildung 3: Aufbau eines Docker-Hostsystems [141, S.3].

2.3.2 Dockerfile

Ein Dockerfile ist eine Datei mit selbigem Namen, die die Erstellung eines Images mithilfe einer oder mehrerer Anweisungen spezifiziert. Anweisungen werden konsequentiv ausgeführt und führen jeweils zu einer neuen Schicht eines Images, die später in Summe das Image ergeben. Damit stellen Dockerfiles eine einfache Möglichkeit dar, Images auf Basis einer Vorlage automatisiert zu generieren.

Eine Anweisung kann z.B. ein Tool installieren oder starten, eine Umgebungsvariable festlegen oder einen Port öffnen. Ein exemplarisches Dockerfile ist im Folgenden dargestellt [149].

```
FROM ubuntu

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

WORKDIR /etc/nginx
CMD ["nginx"]
```

EXPOSE 80
EXPOSE 443

Wie in diesem Dockerfile zu erkennen ist, bestehen Anweisungen aus einem Schlüsselwort in Großbuchstaben und mindestens einem Parameter. Die Erklärungen zu den einzelnen Anweisungen ist in Tabelle 2 gegeben [79]:

Schlüsselwort einer Anweisung	Bedeutung
FROM	Setzt das Basisimage für alle folgenden Anweisungen. Jedes Dockerfile muss diese Anweisung am Anfang enthalten. In obigem Beispiel wird die neuste <i>Ubuntu</i> -Version verwendet.
RUN	Führt den angehängten Befehl während des Buildvorgangs aus und erzeugt damit eine neue Schicht.
WORKDIR	Setzt das Arbeitsverzeichnis als Bezugspunkt für alle folgenden RUN- und CMD-Anweisungen. Kann mehrmals pro Dockerfile vorkommen.
CMD	Führt den angehängten Befehl aus, wenn der Container gestartet wird. Pro Dockerfile kann nur eine CMD-Anweisung existieren.
EXPOSE	Öffnet den angegebenen Port des Containers zur Laufzeit. In obigem Beispiel sind das Port 80 und 443 für HTTP und HTTPS. Auf dem Hostsystem werden diese standardmäßig auf einen Port zwischen 1024 und 49151 abgebildet.

Tabelle 2: Anweisungen eines Dockerfiles und deren Bedeutung.

2.3.3 Containerformate *LXC*, *libcontainer* 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 Hostsystem kommunizieren. Es wird z.B. festgelegt, wie das Dateisystem des Hostsystems verwendet wird, welche Features 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 weitere Abhängigkeiten mit dem Linux-Kernel kommunizieren [21].

Ende Juni 2015 kündigte Docker an, 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 [22]. Gleichzeitig stellte *runC* vor, das eine Implementierung des *OCF* darstellt und maßgeblich auf dem alten Format *libcontainer* beruht [69][53][64].

2.3.4 Image

Images bilden als unveränderbare Dateien die Basis von Containern. Sie sind einfach portierbar und können geteilt, gespeichert und aktualisiert werden. Images sind durch ein *Union*-Dateisystem in Schichten unterteilt, die überlagert ein Image ergeben, das als Container gestartet werden kann [196, S.11]. Die von Docker unterstützten *Union*-Dateisysteme, wie z.B. *AuFS*, *Brfs* und *Device Mapper* basieren auf dem *Copy-On-Write*-Modell [196,

S.8][129, S.3][141, S.4].

Genauer gesagt, besteht ein Image aus einem Manifest, das ein oder mehrere Schichten (Layers) referenziert. Images und Schichten sind jeweils über Hashwerte eindeutig referenzierbar und liegen auf dem Docker-Hostsystem im Verzeichnis `/var/lib/docker/graph/`. Im Unterordner eines Images liegen mehrere, für ein Image spezifische Dateien (s. 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 [51].

```
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 schrittweise aus den Anweisungen eines Dockerfiles erstellt. Die Schichten eines Images stellen i.d.R. eine minimale Ausführungsumgebung mit Bibliotheken, Binaries und Hilfspaketen sowie den Quellcode der Anwendung, die im Container ausgeführt werden soll, dar. Die Schichtenstruktur erlaubt es, Images modular aufzubauen, sodass sich Änderungen eines Images nur auf eine Schicht auswirken.

Soll z.B. in ein bestehendes Image der Webserver *Nginx* integriert werden, kann dieser mit dem Kommando `apt-get install nginx` innerhalb eines Containers zur Laufzeit installiert werden, was eine neue Schicht im Image zur Folge hat.

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 vor, die sich die Images teilen [129, S.3].

Wie in Abb.5 zu sehen ist, basieren die beiden beispielhaften Images `redis:3.0.6` und `nginx:1.9.9` auf zwei gleichen Schichten, die durch die Anweisungen

ADD und CMD jeweils erzeugt werden. In dieser Abbildung sind die einzelnen Schichten eines Images vertikal gelistet. Merkmale der resultierenden Images sind in der ersten Zeile zusammengefasst.

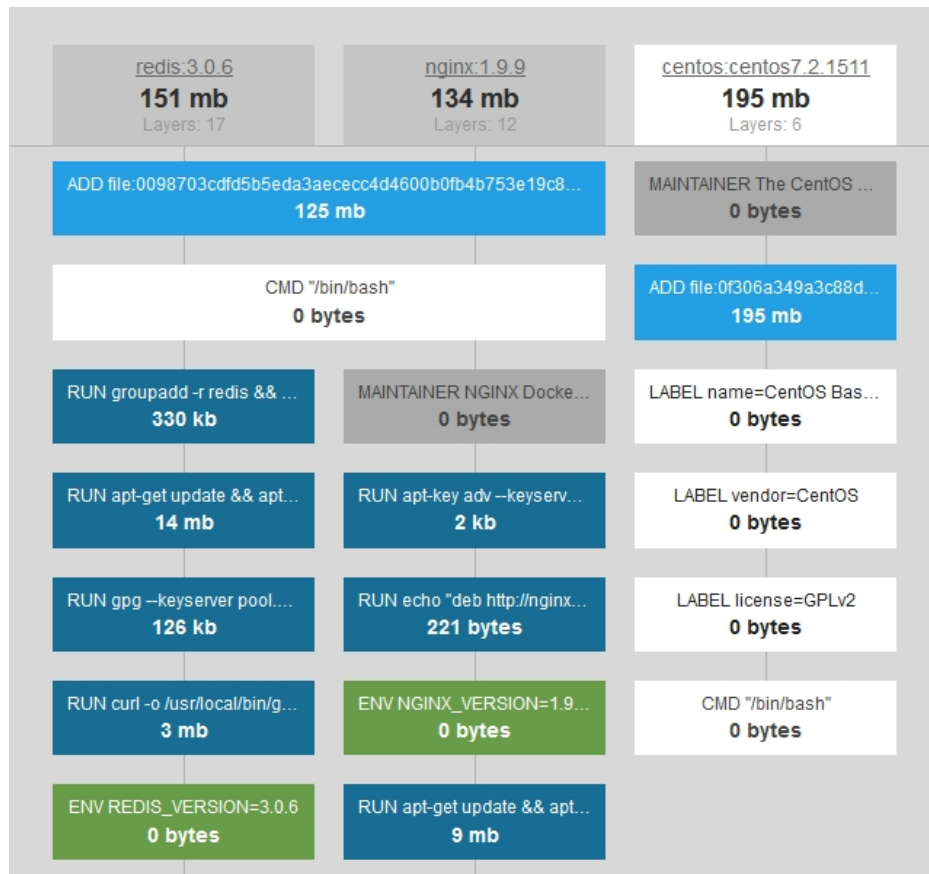


Abbildung 5: Vergleich von drei Images auf Schichtebene [68].

Wie in Abb.6 dargestellt, kann über die Shell z.B. das Image eines *CentOS*-Betriebssystems mit dem Befehl `docker pull nginx` aus dem *Docker Hub* (siehe Abschnitt 2.3.6) bezogen werden [82][32]. Wie in Abb.5 und Abb.6 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 auf dem Hostsystem vorliegenden Images, kann - wie in Abb.7 - mit dem Befehl `docker images` in der Shell generiert werden [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).

2.3.5 Container

Ein Container ist die laufende Instanz eines Images, die in Sekundenbruchteilen startet [129, S.1]. Er existiert als Gastsystem auf einem Docker-Hostsystem und kann mehrere Anwendungen betreiben. Nach dem Konzept von *Micro-services* läuft in einem Container nur eine Anwendung.

Ein Container wird über eine Reihe an Operationen gesteuert. Diese umfassen 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 [196, S.12][130, S.2].

Z.B. wird ein Container über das Kommando `docker run IMAGE` gestartet und über `docker rm CONTAINER` gestoppt.

Wie in Abb.8 illustriert, wird dem Image beim Startvorgang eine weitere Containerschicht hinzugefügt, in die Datenänderungen während dem Containerbetrieb geschrieben werden. Auf die Schichten des unterliegenden Images wird nur lesend zugegriffen.

Ein Container wird als privilegiert bezeichnet, wenn er mit Rootrechten gest-

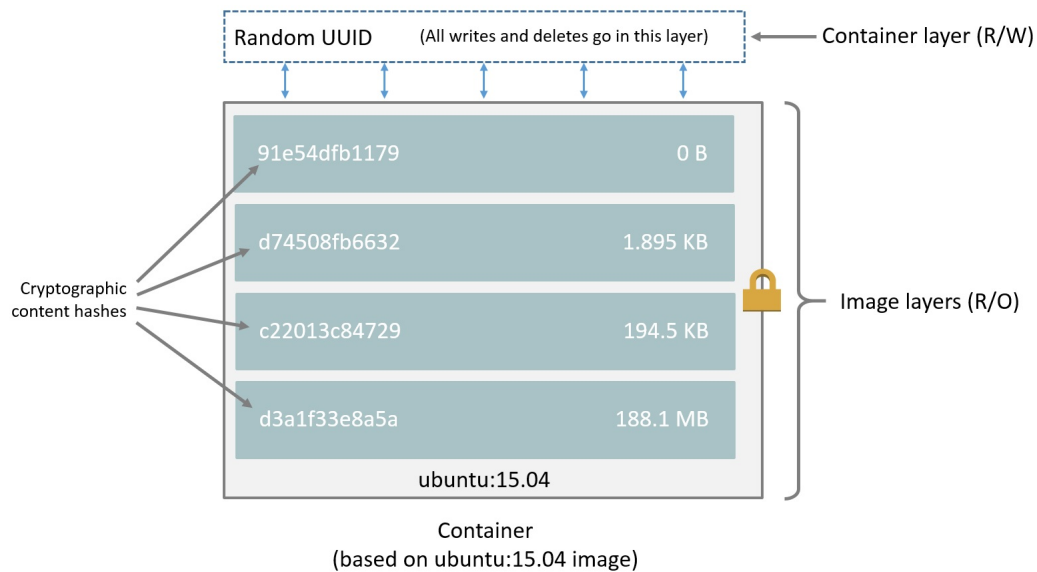


Abbildung 8: Schichtenaufbau eines Containers [26].

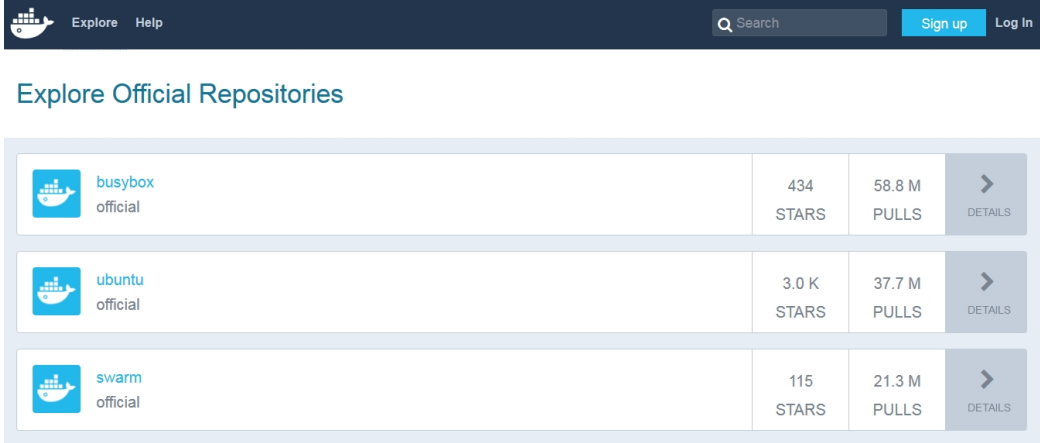
artet wurde. Standardmäßig startet ein Container unprivilegiert ohne Root-rechte. Mit einem reduzierten Set an ausführbaren Aktionen, die über verschiedene, in Kapitel 4 vorgestellte Mechanismen definiert werden, lassen sich Container unabhängig von Rootrechten einschränken.

2.3.6 Registry

Eine Registry ist eine Webanwendung, die als Speicher- und Verteilerplattform für Images dient. Über eine Registry-API, sind Docker-Komponenten in der Lage mit Registries zu kommunizieren. Images sind mit Tags versehen in Repositories gegliedert, die wiederum in der Registry persistiert werden [25].

Docker stellt eine Vielzahl an Images frei verfügbar in einem Dienst, dem *Docker Hub*, zur Verfügung [196, S.11][132, S.3][25]. Für dieses System können Personen und Organisationen Konten anlegen und eigenständig Images in öffentliche und private Repositories hochladen. Das *Docker Hub* bietet bereits mehr als 150.000 Repositories, die von ca. 240.000 Nutzer zusammenstellt und hochgeladen wurden (Stand Juni 2015) [105, S.16]. Wie in Abb.9

zu sehen ist, werden auch Nutzungsstatistiken pro Image gesammelt und angezeigt. Durch diese erweiternden Features ist das *Docker Hub* strenggenommen keine Registry, sondern enthält eine Registry als Teil einer Webanwendung.






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 9: Web-UI des Docker Hubs mit den beliebtesten Repositories [35].

Um Images in einem Repository voneinander zu unterscheiden, werden diese mit sogenannten Tags gekennzeichnet, um mehrere Versionen eines Images in einem Repository zu markieren. Die Images werden nach dem Schema `<repository>:<tag>` identifiziert. Es gibt z.B. im offiziellen Repository des Webserver *Nginx* mehrere Images mit den Tags `latest`, `1`, `1.9` und `1.9.9` [82].

Kapitel 3

Fragestellung

Die Wertschöpfung moderner IT-Unternehmen beruht auf dem Angebot von Diensten, die über das Internet den Nutzern zur Verfügung gestellt werden. Die Dienste werden von Anwendungen realisiert, die i.d.R. in Rechenzentren betrieben werden. Der überwiegende Vermögensgegenstand in diesem Modell ist die Software, die in den Rechenzentren produktiv läuft. Der Wert dieser ist direkt abhängig von Eigenschaften wie der Leistung und Sicherheit eines Rechenzentrums.

Kunden, die ihr Produkt über Rechenzentren anbieten, sind u.a. an Sicherheitsfeatures interessiert, die die Sicherheitsziele aus Abschnitt 2.2 sicherstellen. Je nach Anwendungsfall kommt diesen Zielen unterschiedliche Wichtigkeit zu. Die Erfüllung der Sicherheitsziele muss für einen sicheren Betrieb auf Anwendungs-, Virtualisierung- und Netzwerkebene gegeben sein.

Die Betreiber von Rechenzentren streben gleichzeitig einen möglichst effizienten Betrieb der Kundensysteme in der eigenen Infrastruktur an. Wie in Kapitel 2 gezeigt, bietet der Einsatz von Containertechnologien aus Sicht der Leistung, Effizienz sowie administrativen Faktoren, attraktive Eigenschaften für die Betreiber von Rechenzentren. Mit gegebener Hardware lassen sich z.B. mit Containertechnologien durchschnittlich mehr Kundenanwendungen realisieren, wie wenn letztere mit der konventionellen Hypervisortechnologie

betrieben werden.

Mit den außer Frage stehenden Leistungs- und Effizienzvorteilen, ist demnach die erste zentrale Frage für Betreiber von Cloud-Infrastrukturen, inwiefern Sicherheitsfunktionen von Containertechnologien die erforderlichen Sicherheitsziele erfüllen. Diese Fragestellung kann anhand einer genaueren Untersuchung genauer formuliert werden, die auf der von Mandl in [172, S.36] vorgestellten Risikoanalyse sowie den Empfehlungen von NIST in [185, S.23] basieren. Die Risikoanalyse dient gleichzeitig dazu, Annahmen vorzustellen und Schlussfolgerungen zu ziehen, auf deren Basis die zu betrachtenden Eigenschaften von Docker gegen Ende dieses Kapitels definiert werden.

3.1 Identifikation des Systems

Das Systemmodell von hypervisorbasierten Systemen kann nicht verwendet werden, da das Design der Containersysteme stark von Erstgenannten abweicht. Während nach [172, S.125] virtuelle Maschinen als eigener Sicherheitsmechanismus des Betriebssystems aufgelistet ist, stimmt das für die Containersysteme und deren Architektur nicht. Andere Sicherheitsfeatures des Hostsystems müssen aktiviert werden, um die Containersicherheit zu erhöhen.

Bild zum Systemmodell

Das Systemmodell ist zunächst definiert als ein Hostsystem h und einer Containermenge M , die aus den Containern m_1, m_2, \dots, m_n besteht. Die Container werden von einem Docker-Daemon verwaltet, der in h läuft.

Verteilte Docker-Systeme, in denen mehrere Hosts miteinander kommunizieren, werden in dieser Arbeit nicht betrachtet. Aus diesem Grund existiert in diesem Systemmodell nur ein Hostsystem h mit Docker-Installation, das alle Container in M verwaltet.

3.2 Identifikation der Bedrohungen

In diesem Schritt werden die verschiedenen Bedrohungsarten auf Basis des zuvor definierten Systemmodells erörtert.

Aus Abb.?? und der Funktionsweise von Docker (s. Abschnitt 2.3) können zwei unterschiedliche Gefahrenquellen (A) und (B) identifiziert werden:

- (A) **Docker-Spezifika:** In Containern werden Anwendungen ausgeführt, die nicht zwangsweise vertrauenswürdig sind. Wenn beispielsweise Containerimages von einer öffentlichen Registry, wie dem *Docker Hub*, bezogen werden, existiert keine Garantie, dass aus diesen Images gestartete Container nicht gegen die Sicherheitsziele der Vertraulichkeit, Integrität und Verfügbarkeit verstoßen. Durch die Komplexität moderner Anwendungen und deren Abhängigkeiten zu Bibliotheken, ist es selbst bei quelloffenen Anwendungen schwierig, diese als vertrauenswürdig einzustufen. Deswegen muss davon ausgegangen werden, dass in Containern willkürliche Programme ablaufen (*interne Gefahrenquellen*), die - versehentlich oder beabsichtigt - den Host schädigen können.

Außerdem kann jederzeit ein Fehler in der Codebasis von Docker entdeckt werden, der die Sicherheit von Docker-Hostsystemen und Containern beeinträchtigt.

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

3.3 Auswirkungen der identifizierten Risiken

Bei einer näheren Betrachtung, sind aus Sicht des Hostsystems die Folgen beider Gefahrenquellen sehr ähnlich. Beide Gefahrenquellen (A) und (B) führen dazu, dass ein oder mehrere Container eines Hostsystems schadhaften Code ausführen. Eine Unterscheidung der Risiken, die bei der Untersuchung der Eintrittswahrscheinlichkeiten von Bedeutung ist, wird durch folgende Schlussfolgerung irrelevant.

Schlussfolgerung: Ein Container ist nicht vertrauenswürdig. Bei der Konstruktion einer Sicherheitsarchitektur muss davon ausgegangen werden, dass ein Container von einem Angreifer kontrolliert wird.

Mithilfe dieser Erkenntnis kann das Systemmodell unter Einbeziehung eines Angreifermodells erweitert werden. M wird in zwei echte, nicht-leere Teilmengen C und \overline{C} unterteilt. C enthält korrekt funktionierende, legitime Container und \overline{C} kompromittierte Container.

$$\{\} \neq C \subset M \quad , \quad \{\} \neq \overline{C} \subset M \quad (3.1)$$

$$M = C \cup \overline{C} = \{m \mid m \in C \text{ oder } m \in \overline{C}\} \quad (3.2)$$

Per Konvention wird fortan ein beliebiges Element der Menge C als c und ein beliebiges Element der Menge \overline{C} als \bar{c} notiert.

Mit den Definitionen 3.1 und 3.2 wird ein Systemmodell erzeugt, das von mindestens zwei Container in einem Hostsystem ausgeht, wovon mindestens ein Container c legitim und mindestens ein Container \bar{c} kompromittiert ist.

Damit bildet das Modell den Betrieb von Docker-Containern realitätsnah ab, da diese so konstruiert wurden, dass sie i.d.R. in einer Vielzahl parallel auf h laufen. Wie aus obiger Schlussfolgerung hervorgeht, muss angenommen werden, dass \overline{C} eine nicht-leere Menge ist, also mindestens ein Container \bar{c} existiert, der Schadcode ausführt und die Sicherheitsziele von h und zu

schützenden Containern in C verletzen kann. Beide Eigenschaften werden von diesem Systemmodell abgedeckt.

Durch die Existenz von \bar{c} , können die Sicherheitsziele der Vertraulichkeit, Integrität und Verfügbarkeit verletzt werden:

- **Vertraulichkeit:** \bar{c} kann Aktionen ausführen, die unbefugten Zugriff auf Informationen von h und Container aus C ermöglichen.
Beispiele: MITM-Angriff, Lesen von Daten anderer Container
- **Integrität:** \bar{c} kann Aktionen ausführen, die Daten von h und C manipulieren.
Beispiele: Überschreiben oder Löschen von Daten anderer Container
- **Verfügbarkeit:** \bar{c} kann Aktionen ausführen, um die von h vorgesehenen Mechanismen zur Aufrechterhaltung der Verfügbarkeit, zu umgehen. Die Funktionstüchtigkeit von h und C kann dadurch beeinträchtigt werden.
Beispiele: DoS-Angriffe wie Erschöpfung von Host-Ressourcen

3.4 Anforderungen an die Sicherheit von Containern

Aus den Annahmen und der Schlussfolgerung in Abschnitt 3.3, lässt sich eine allgemeine Sicherheitsanforderung formulieren:

Von einem kompromittiertem Container \bar{c} , darf der zu schützende Rest des Systems nicht betroffen sein. Die Sicherheitsziele der Vertraulichkeit, Integrität und Verfügbarkeit sollen für h und C gewahrt werden.

3.5 Theoretischer Lösungsansatz

Die Strategie zur Erfüllung der definierten Sicherheitsanforderungen, beruht unter Docker auf einem zweistufigen Ansatz:

- (1) Angreifen soll es erschwert und bestenfalls unmöglich gemacht werden, aus der virtualisierten Umgebung eines \bar{c} auszubrechen. Angreifer sollen nicht die Möglichkeit besitzen, die ursprünglich für \bar{c} vorgesehenen Rechte zu erweitern.
- (2) Falls es einem Angreifer dennoch gelingen sollte (1) zu verletzen, soll er durch weitere Zugriffskontrollen daran gehindert werden, Schaden in h und den zu schützenden Containern aus C anzurichten.

Mithilfe der im nächsten Abschnitt aufgelisteten Mittel, versucht Docker diese umzusetzen.

3.6 Umsetzung des Lösungsansatzes

Zunächst lassen sich die möglichen Sicherheitsmechanismen in drei Arten unterteilen. Zur Einordnung der folgenden Kapitel, sind diese kurz vorgestellt [172, S.40ff.][185, S.23]:

- **Technische Kontrollen:** Umfasst alle hardware- und softwarebasierten Mechanismen, z.B. ein Zugriffsschutz unter Verwendung einer MAC.
- **Administrative Kontrollen:** Enthält Managementkontrollen, die z.B. durch Konfigurationen, Entwicklung einer Sicherheitspolitik, Reduzierung der Fehleranfälligkeit durch Automatisierung und Sicherheitsschulungen des Personals umgesetzt werden.
- **Physische Kontrollen:** Beinhalten Mechanismen wie Sicherheitsschleusen, Schlösser und Wachpersonal. Obwohl ein Bezug zum Betrieb von Rechenzentren hergestellt werden kann, haben physische Kontrollen keine spezifische Relevanz für die containerbasierte Virtualisierung und

sind aus diesem Grund an dieser Stelle nur zum Zweck der Vollständigkeit aufgeführt.

Docker hat als Softwareprodukt die Möglichkeit, direkt technische Kontrollen in die Codebasis einzubauen bzw. solche, die der Linux-Kernel implementiert, zu unterstützen. Auch administrative Sicherheitsansätze verfolgt Docker, die teilweise durch technische Mittel begünstigt werden.

Die Umsetzung erfolgt mittels zwei Prinzipien, die am Anfang von Kapitel 4 vorgestellt sind.

3.7 Ziel der Arbeit

Gegenstand dieser Arbeit ist, die von Docker integrierten softwarebasierten und administrativen Maßnahmen zu analysieren.

Im Rahmen der Arbeit werden die Sicherheitsmaßnahmen sowie deren Einsatz unter Docker vorgestellt. Die in diesem Kapitel erarbeiteten Annahmen und Schlussfolgerungen bilden dabei die Basis der Untersuchung. Die einzelnen Sicherheitsmerkmale sind dabei unabhängig von der Infrastruktur. Das bedeutet, dass sie bei einem lokalen Betrieb von Docker auf Computern von Entwicklern, bis hinzu Docker-Installationen in Cloud-Infrastrukturen relevant sind.

In dieser Arbeit wird auch betrachtet, welche Integrationsmöglichkeiten für Docker in Cloud-Infrastrukturen existieren und welche speziellen Sicherheitskomponenten einsetzbar sind bzw. in welchem Umfang diese von externen Dienstleistern angeboten werden.

Kapitel 4

Sicherheit durch Linux-Funktionen

Die Docker-Entwickler haben in den letzten Monaten die Integrations- und Anpassungsmöglichkeiten zusätzlicher Sicherheitsmaßnahmen, v.a. die von Mandatory Access Controls (MACs), stark verbessert, da die Containersicherheit für *Docker* nach eigenen Aussagen höchste Priorität hat [52][48].

Auch die Tatsache, dass sich zur Zeit die Konkurrenz *CoreOS* mit der Containerlösung *rkt* als sicherheitsfokussierte Alternative zu Docker auf dem Virtualisierungsmarkt etablieren will [127], ist Anreiz für *Docker* die Sicherheit ihrer eigenen Entwicklung nicht zu vernachlässigen. Die Veröffentlichung des großen Sicherheitsupdates für Docker mit der Version 1.10 am 4. Februar 2016 geschah wenige Stunden nach der Ankündigung der Version 1.0 von *rkt* seitens *CoreOS* [125][126], was eine starke Konkurrenz zwischen den beiden Anbietern vermuten lässt.

Die allgemeine Vorgehensweise von Docker beruht auf den beiden Prinzipien *Defense In Depth* und *Principle Of Least Privilege*, um einen möglichst sicheren Betrieb von Containern zu ermöglichen.

- *Defense In Depth*: Es werden möglichst viele, unabhängige Sicherheitsschichten kombiniert aktiviert, um die Gesamtsicherheit eines Systems

zu erhöhen. Der Funktionsumfang der einzelnen Mechanismen kann sich - wie auch in der vorliegenden Sicherheitsarchitektur von Docker der Fall ist - mit Mechanismen anderer Schichten überschneiden.

- *Principle Of Least Privilege*: Die angewandten Sicherheitsmechanismen bewirken, dass die von Docker initiierte Aktionen eingeschränkt werden. Docker ist nur befugt Operationen auszuführen, die dazu dienen, seinen Zweck als Containertechnologie zu erfüllen. Alle anderen Privilegien werden Docker verwehrt.

Dieses Kapitel stellt die von dem Betriebssystem Linux und dessen Kernel ermöglichten Sicherheitsmechanismen vor, die Docker im Rahmen dieser beiden Prinzipien unterstützt.

Dazu zählt die Isolierung kritischer Bereiche, die auf Basis des `chroot`-Befehls in Form von Namespaces, erfolgt (s. Abschnitt 4.1). Außerdem werden Control Groups, kurz Cgroups, vorgestellt, die eine Ressourcenverwaltung implementieren (s. Abschnitt 4.2). Während beide Mechanismen eine hohe Sicherheitsrelevanz aufweisen, können sie als featureerweiternde Funktionen eines Hostsystems verstanden werden, deren alleiniges Ziel es ist, die Idee von virtuellen Gastsystemen, den Containern, zu realisieren.

Anschließend werden Zugriffskontrollen erklärt, die in mehreren Variationen unter Docker existieren (s. Abschnitt 4.3). Mechanismen dieser Art setzen teilweise Sicherheitsmodelle um, z.B. auf der Basis von sogenannten Capabilities.

Wie in Abschnitt 2.1.2 bereits geschildert wurde, wird Docker zur Zeit nur für Linux angeboten. Aus diesem Grund ist eine Untersuchung von Sicherheitsfeatures anderer Betriebssysteme in dieser Arbeit irrelevant.

4.1 Isolierung

Die Isolierung für Container wird unter Linux für die meisten kritischen Bereiche eines Systems über Namespaces realisiert.

Da Anwendungen innerhalb von Container eine komplette Laufzeitumgebung angeboten werden muss, müssen alle relevanten Bereiche des Hostsystems durch Namespaces abgedeckt sein. Wenn unter Linux z.B. ein neuer Prozess gestartet werden soll, wird dem Kernel über *System Calls* mitgeteilt, hierfür einen neuen Namespace bereitzustellen.

Aus technischer Sicht beinhaltet ein Namespace eine Baumstruktur zu betrachtender Ressourcen. Diese Struktur ist in einer Lookup-Tabelle abgebildet, die global verfügbare Ressourcen abstrahiert und innerhalb eines Namespaces bereitstellt. Änderungen globaler Ressourcen sind unsichtbar für Prozesse in einem Namespace, jedoch sichtbar für solche außerhalb. Namespaces sind in der Theorie beliebig verschachtelbar. In der Realität wird diese Eigenschaft durch die Größe der Lookup-Tabelle begrenzt [131, S.1f.][71].

Dadurch kann das Konzept von Namespaces als Lösungsansatz betrachtet werden, um das Sicherheitsziel Vertraulichkeit zu erreichen. Sie sind der grundlegende Baustein, um eine Containerisolierung unter Linux zu realisieren. Auch adressieren Namespaces den Punkt (1) aus Abschnitt 3.5.

Unter Linux existieren sechs Namespaces, die in die Kategorien IPC, Network, Mount, PID, User und UTS unterteilt sind. Von Docker werden alle verwendet, der User-Namespace jedoch wird erst seit Docker-Version 1.10 unterstützt.

Es besteht die Möglichkeit, dass bald weitere Namespaces dem bestehenden Set hinzugefügt werden. Zum einen wird derzeit ein eigener Namespace für Control Groups implementiert (siehe Abschnitt 4.2). Zum anderen ist in Diskussion, für die bestehenden Sicherheitsmodule des Kernels (siehe Kapitel 4.3.2), diverse Logfunktionen und Geräte unter Linux, eigene Namespaces zur Verfügung zu stellen [157, S.19].

4.1.1 Prozessisolierung durch den PID-Namespace

Jeder Container entspricht auf dem Hostsystem zunächst einem Prozess. Da die Container untereinander isoliert sein sollen, dürfen auch die zugrundelie-

genden Containerprozesse nicht miteinander interferieren.

Docker erreicht diese Isolierung auf Prozessebene durch die Nutzung von PID-Namespaces, in die Container eingebettet werden. Nach diesem hierarchischen Konzept ist es einem Prozess nur möglich, selbsterzeugte Kindprozesse zu beobachten und mit ihnen zu interagieren.

In Abb.10 ist ein Beispiel von PID-Namespaces gegeben. Ein Prozess besitzt die PID 3698, die außerhalb des rot markierten Namespaces gültig ist. Außerdem besitzt er die PID 1, die innerhalb des Namespaces vergeben wurde. Der Elternprozess mit PID 627 ist in der Lage diesen jederzeit mit dem Befehl `kill` zu beenden oder allgemein Kindprozesse mit einem Aufruf von `ps` zu überwachen. Der Prozess mit PID 3698 hingegen befindet sich im roten Namespace und hat keine Kenntnis von der übergeordneten Prozesshierarchie.

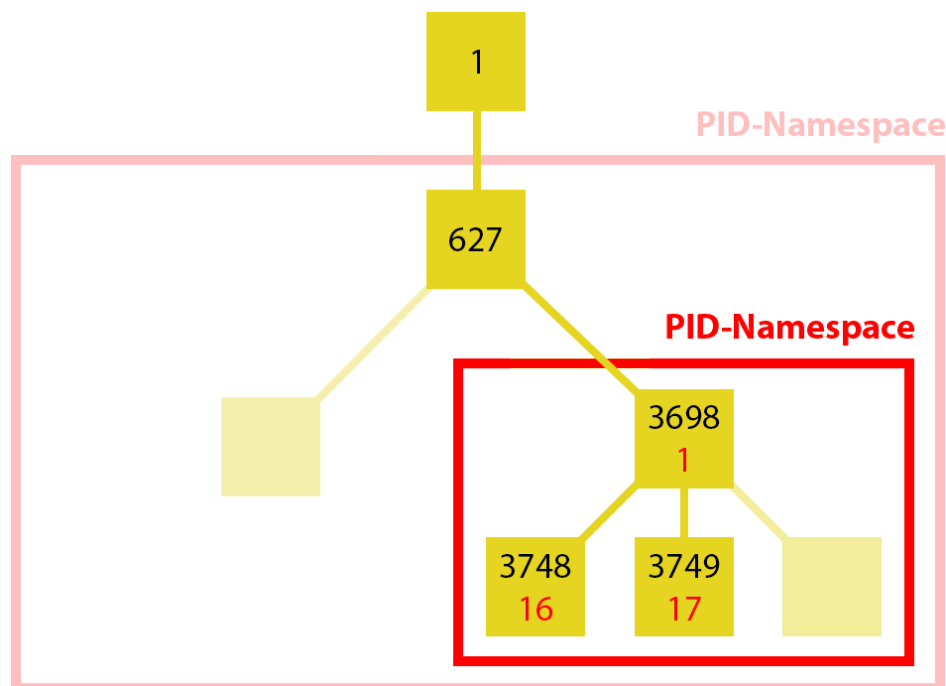


Abbildung 10: Prozesshierarchie unter Verwendung von PID-Namespaces (eigene Abbildung auf Basis von [180]).

Übertragen auf die containerbasierte Virtualisierung bedeutet das, dass das

Hostsystem vollen Zugriff auf die laufenden Container hat, Containerprozesse jedoch keine Kenntnis von Prozessen außerhalb des eigenen Namespaces besitzen. Diese Eigenschaft erschwert es Angreifern, ausgehend von \bar{c} , Schaden anzurichten, da sie keine Informationen über Prozesse außerhalb des eigenen PID-Namespaces erlangen können.

Ein weitere nützliche Eigenschaft ist das Setzen der PID auf 1 für einen Prozess, dem ein PID-Namespace zugewiesen wurde. Ein Prozess mit PID 1 ist es möglich, jeder seiner Kindprozesse zu terminieren sobald er selbst beendet wird. Beim Betrieb von Containern kann dieses Feature genutzt werden, um über einen Befehl des Hostsystems einen Container vollständig herunterzufahren.

Wie sich dieses Konzept der Geheimhaltung in der Praxis auf gestartete Docker-Container auswirkt, ist in Abb.11 und Abb.12 zu sehen. Zum Vergleich ist die gleiche Situation in obiger Abb.10 dargestellt.

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

Abbildung 11: Ausschnitt der Ausgabe des Befehls `ps -eafxZ` auf einem Docker-Hostsystem (eigene Abbildung).

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

Abbildung 12: Ausgabe des Befehls `ps -eafxZ` in einem Docker-Container (eigene Abbildung).

Abb.11 zeigt einen Ausschnitt der Ausgabe des Befehls `ps -eafxZ` auf dem Hostsystem. Er enthält die laufenden Docker-Prozesse des Daemons mit PID 627 (Zeile 1) sowie die eines Containers, in dem eine Shell gestartet wurde (Zeile 2). Innerhalb der Shell wurde der Befehl `sleep 1000` zweifach ausgeführt, was die Erstellung zweier neuer Prozesse zur Folge hat (Zeile 3 und 4). Wie zu erkennen ist, sind die einzelnen Containerprozesse aus Sicht des Hostsystems identifizierbar.

Abb.12 zeigt die vollständige Ausgabe des gleichen Befehls innerhalb eines Containers. Prozesse außerhalb des Containers sind nicht sichtbar. Außerdem weisen die Containerprozesse im Vergleich zu Abb.11 nun eine andere PID auf. Da die Shell der initiale Containerprozess ist, erhält sie die PID 1.

4.1.2 Dateisystemisolierung durch den Mount-Namespace

Auch das Dateisystem des Hostsystems h muss vor unrechtmäßigen Zugriffen aus einem Container \bar{c} geschützt werden.

Allgemein ist ein Dateisystem, wie eine Prozessstruktur in Abschnitt 4.1.1, hierarchisch aufgebaut. Erstgenanntes kann mithilfe des Mount-Namespace unterteilt werden, sodass unter Docker jeder Container eine andere Sicht auf die Verzeichnisstruktur des Hostsystems hat. Nur ein bestimmtes Unterverzeichnis ist für einen Container sichtbar, wenn er dieses als Mountpoint einbindet.

Das virtuelle `/proc`-Verzeichnis des Hostsystems bleibt von Mount-Namespaces unberührt, da es essentielle Informationen für jeden Prozess enthält [138][159]. Außerdem erfordern Anwendungen, die innerhalb von Containern laufen, Zugriff auf besagtes Verzeichnis. Als Konsequenz erben Container diese notwendigen Verzeichnisse direkt von ihrem Host.

Es enthält aber auch Details des Kernels und anderer Prozesse [72]. Diese müssen wegen der Existenz von \bar{c} vor Lese- und Schreibzugriffen geschützt werden. Linux erlaubt es, Verzeichnisse derart freizugeben, dass Prozesse nur lesend mit ihnen in Verbindung treten können. Docker unterstützt das standardmäßige Einbinden ohne Schreibrechte [141, S.4f.].

Einem Containern ist es nicht erlaubt, Verzeichnisse des Hostsystems erneut einzubinden. Dieses Verbot wird technisch über die Verweigerung der Capability `CAP_SYS_ADMIN` (s. Abschnitt ??) erreicht. Ist ein Container im Besitz dieses Rechts, kann dieser Verzeichnisse, deren Zugriff ihm ursprünglich nur lesend gestattet wurde, neu einbinden. Im Rahmen des neuen Mountvorgangs, kann der Container über die Zugriffsart auf einzubindenden Verzeich-

nisse frei verfügen und ist nicht allein auf das Leserecht beschränkt.

Mit dem Konzept von Mount-Namespaces interferieren Daten von Containern nicht. Jede oberste, beschreibbare Schicht eines Containers wird in einem speziell dem Container zugewiesenen Verzeichnis gespeichert. Unter der Annahme, dass es sich bei Mount-Namespaces um ein sicheres Verfahren handelt, kann ein \bar{c} nicht in Verzeichnisse anderer Container aus C schreiben.

4.1.3 IPC-Isolierung durch den IPC-Namespaces

Ergänzend zu den PID-Namespaces, die die Sichtbarkeit sowie Kontrolle über Prozesse in der Prozesshierarchie einschränken, kann auch die Kommunikation zwischen Prozessen, die Inter Process Communication, limitiert werden. Für diesen Zweck erlaubt Linux die Erstellung von IPC-Namespaces.

Docker gewährleistet dies durch die standardmäßige Zuweisung eines IPC-Namespaces pro Container, in dem ein Prozesse nur mit anderen Prozessen in Kontakt treten kann, wenn sich diese im gleichen IPC-Namespaces befinden. Eine versehentliche oder beabsichtige Interferenz zwischen Containern oder zwischen Containern und Docker-unabhängigen Prozessen des Hostsystems wird damit ausgeschlossen.

4.1.4 Netzwerkisolierung durch den Network-Namespaces

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

Dazu stellt Docker jedem Container einen eigenen unabhängigen Netzwerkstack zur Verfügung, der durch Network-Namespaces realisiert wird. Jeder Namespace hat seine eigene private IP-Adresse, IP-Routingtabelle, Loopback-Interface und Netzwerkgeräte [142, S.2f.]. Eine Kommunikation zu anderen

Containern auf dem gleichen oder entfernten Hosts geschieht nur über dafür vorgesehene Schnittstellen.

Um die oben genannten Netzwerkressourcen anzubieten, wird jedem Network-Namespace ein eigenes `/proc/net`-Verzeichnis zugewiesen, das netzwerkspezifische Informationen enthält [72]. Die Nutzung von Befehlen wie `netstat` und `ifconfig` wird damit aus einem Network-Namespace heraus auch ermöglicht [131, S.7].

Standardmäßig wird von Containern eine *Virtual Ethernet Bridge* mit dem Namen `docker0` genutzt, um mit dem Hostsystem und anderen Containern zu kommunizieren. Neu gestartete Container werden dieser Bridge hinzugefügt, indem deren Netzwerkinterface `eth0` mit der Bridge verbunden wird. Aus Sicht des Hostsystems ist das Interface `eth0` ein virtuelles *veth*-Interface. Angepasste Netzwerktreiber können für Bridges und sogenannte Overlay-netzwerke, die eine Kommunikation zwischen mehreren Docker-Systemen ermöglichen, erstellt und genutzt werden [142, S.3][110]. Aus Gründen der Netzwerksicherheit macht es Sinn eigene Treiber zu verwenden, da die standardmäßige Bridge ohne Filter operiert. Filter können z.B. mit dem Programm `ebtables` erstellt werden.

4.1.5 Userisolierung durch den User-Namespaces

Bislang werden Container unter Docker und anderen Containerlösungen mit Rootrechten gestartet. Falls es einem Angreifer in diesem Szenario gelingt, aus der Isolierung von \bar{c} auszubrechen, ist er automatisch Root-User auf dem Host, was ein hohes Sicherheitsrisiko birgt. Durch die potentielle Gefahr dieser Vorgehensweise, ist die Einführung von User-Namespaces mit Docker-Version 1.10 ein Meilenstein für die Sicherheit von Docker.

Dieser Namespace führt einen Mechanismus ein, unter dem Rootrechte in Containern nicht Rootrechten auf dem Hostsystem entsprechen. Ein Root-User innerhalb eines Containers wird auf dem Hostsystem auf einem Nicht-Root-User abgebildet.

Linux verwendet User-IDs (UIDs) und Group-IDs (GIDs), um Verzeichnisse und Dateien eines Dateisystems sowie Prozesse mit Eignerinformationen zu versehen. User-Namespaces erlauben unterschiedliche UIDs bzw. GIDs innerhalb und außerhalb des Namespaces. Im Kontext des Hostsystems kann dadurch ein unprivilegierter User ohne Rootrechte existieren, während der gleiche User innerhalb von Containern mithilfe von User-Namespaces privilegiert ist [74].

Eine Unterstützung von User-Namespaces war seit Docker-Version 1.6 vorgesehen [91]. Durch einen Bug der Programmiersprache *Golang* und Integrationschwierigkeiten in die bestehende Docker-Codebasis, kam es jedoch zu Verzögerungen [121][114][89]. Beide Probleme sind jedoch mittlerweile behoben, wie die erfolgreiche Integration von User-Namespaces in aktuelle Releases von Docker bestätigt [89]. In der aktuell neusten Docker-Version 1.10 werden besagte Namespaces nicht automatisch verwendet. Sie müssen manuell, wie z.B. in [170] erklärt, aktiviert werden [48]. Eine standardmäßige Verwendung des Namespace ist in Zukunft zu erwarten.

Auch für Cloudanbieter sind User-Namespaces von Vorteil: Mit einer Auflösung der Container auf Userebene ist es möglich die Nutzung von Diensten auf Userbasis einzugrenzen und abzurechnen [145, S.3].

4.1.6 UTS-Isolierung durch den UTS-namespace

Mit einem *UTS-namespace* ist es möglich, jedem Container einen eigenen Hostnamen zuzuweisen. Der Container kann diesen Namen abfragen und ändern [142, S.3]. Dieser Namespace trägt damit nicht zur Containersicherheit bei und ist daher nur aus Gründen der Vollständigkeit an dieser Stelle erwähnt.

4.1.7 Geräteisolierung

In Unix-basierenden Betriebssystemen wie Linux erfolgt der Zugriff auf Hardware über sogenannte Device Nodes (Geräte), die im Dateisystem von speziellen Dateien repräsentiert sind.

Ein paar wichtige Geräte und deren Zuständigkeiten sind im Folgenden aufgeführt. Das Netzwerkinterface ist auch ein Gerät, welches bereits mit dem Network-Namespace isolierbar ist [143].

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

Wie zu sehen ist, handelt sich dabei um kritische Systemkomponenten, über die Container unter keinen Umständen verfügen dürfen. Deswegen ist es notwendig den Zugriff auf Geräte stark einzuschränken, um den Host vor Missbrauch zu schützen.

Aktuell existiert im Linux-Kernel kein eigener Namespace für Geräte. Die Implementierung von Device-Namespaces erfolgte bereits von *Cellrox*-Mitarbeitern im Jahr 2013 [165]. Eine Integration in den Linux-Kernel fand jedoch bislang nicht statt.

Docker löst dieses Problem, indem es standardmäßig den Zugriff auf Geräte von unprivilegierten Containern untersagt. Über die Befehlssyntax `docker run --device=DEVICE` können auch unprivilegierten Containern Geräte manuell zur Verwendung freigegeben werden. Privilegierte Container können beliebig auf Geräte zugreifen [33].

4.2 Ressourcenverwaltung durch Control Groups

DoS-Angriffe beabsichtigen das Sicherheitsziel der Verfügbarkeit zu verletzen und gehören in Multi-Tenant-Service-Systemen zu einem verbreiteten An-

griffsmuster [132, S.5]. Um die Verfügbarkeit von Containern sicherzustellen, bietet der Linux-Kernel sogenannte Control Groups (cgroups) an.

Alle gängigen Linux-basierten Containerlösungen, darunter auch Docker, nutzen aktuell Control Groups, um Ressourcen für Container zu verwalten [179, S.16]. Der Einsatz von Cgroups unter Docker umfasst - wie im Quellcode von *runC* zu sehen ist - die Kontrolle über CPU, Arbeitsspeicher, Geräte, Netzwerkinterfaces und I/O-Operationen auf Speichermedien wie HDD, SSD und USB-Speicher [14][49]. Die Verwaltung von Letzteren wurden mit dem neusten Docker-Release, Version 1.10, erweitert [123].

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 [33].

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

Neben dem Ressourcenmanagement bieten Control Groups auch Nutzungsstatistiken an. Diese sind unter Docker mit dem Befehl **docker stats CONTAINER [CONTAINER]** für ein oder mehrere Container abrufbar [29].

Control Groups sind historisch aus dem Konzept von sogenannten Resource Limits, auch Rlimits genannt, gewachsen. Mit Resource Limits werden weiche und harte Limits definiert, die jedem Prozess zugewiesen werden. Der Betrieb von Containern verlangt jedoch eine Ressourcenverteilung auf Containerbasis. Wie in Abschnitt 4.1.1 gezeigt wurde, entspricht ein Container einem Set an Prozessen.

Viele Containertechnologien erweiterten deswegen Resource Limits mit eigenen Features. Z.B. fügten die Entwickler von *FreeBSD* für den Betrieb von *Jails* sogenannte *Hierarchical Resource Limits* hinzu [45]. *Solaris* bietet die Nutzung von *Resource Pools* an, die eine Partitionierung von Ressourcen

implementieren [13]. Auch *OpenVZ* und *Linux-VServer* erweitern die Resource Limits, sodass weiche und harte Limits pro Container definiert werden können [179, S.15f.].

Die Nachteile von Ressource Limits wurden mit der Implementierung von Control Groups 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 Control Groups komplexe Verfahren implementiert werden, die zur Korrektur von limitüberschreitender Prozesse dienen [14]. Die Implementierung von Control Groups wurde ab 2012 weiter verbessert, sodass eine Update unter dem Namen *Unified Control Group Hierarchy* im Jahr 2014 in den Linux-Kernel integriert wurde [44][111]. Von Docker wird dieses Update noch nicht verwendet [34]. In zukünftige Kernelversionen soll ein neuer Control Groups-Namespace implementiert werden, der auf der *Unified Hierarchy* basiert [144].

Wichtig zu erwähnen ist, dass die Implementierung von Control Groups, verglichen mit der von Ressource Limits, angeblich nicht vollständig ist. Das Feature, Dateisysteme als Ressourcen mit Control Groups zu steuern, fehlt nach Aussagen von Reshetova et al. [179, S.19]. Auch im aktuellen Quellcode von *runC* ist eine Schnittstelle zum Dateisystem als „nicht unterstützt“ gekennzeichnet und wird demzufolge auch nicht von Docker genutzt [49]. Diskussionen im *GitHub*-Repository von Docker verweisen in Bezug zu diesem Feature auf Abhängigkeiten zur Art des Dateisystems. Offenbar lassen sich Kontingente für die Nutzung von Dateisystemen nur mit *Device Mapper* und *Brtrfs* softwaretechnisch lösen. *AuFS* ermöglicht dieses Feature hingegen 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 Implementierung von Kontingenten vor [2].

Es ist zu erwarten, dass die fehlenden Funktionen von Control Groups sowie die ausstehende Unterstützung der *Unified Hierarchy* zukünftig im Rahmen

der Integration des Control Groups-Namespace stattfindet.

Für Docker-Version 1.11 wurde außerdem die Unterstützung von PID-Control Groups angekündigt. Sie sollen ergänzen zu den bereits unterstützten Ressourcentypen die Anzahl der erlaubten Prozesse innerhalb eines Containers regulieren [184][155]. Dieses Update verhindert das Angriffsmuster sogenannter Fork Bombs.

4.3 Einschränkung von Zugriffsrechten

Auf Basis der in 3.3 erarbeiteten Annahme, dass Angreifer ausgehend von \bar{c} Angriffe gegen die Sicherheitsziele von h und C ausüben kann, müssen weitere Sicherheitsmechanismen verwendet werden, um letztere zu schützen. Diese zusätzlichen Mechanismen realisieren Zugriffskontrollen, die auch dann die Sicherheit von h und C wahren, wenn es einem Angreifer möglich ist, seine Privilegien in \bar{c} unrechtmäßig auszudehnen.

Die in diesem Kapitel vorgestellten Maßnahmen zur Zugriffskontrolle adressieren Punkt (2) des Lösungsansatzes aus Abschnitt 3.5.

Zunächst werden zwei grundsätzliche Sicherheitsrisiken von Betriebssystemen, insbesondere Linux, vorgestellt, die in diesem Kapitel mit geeigneten Lösungsansätzen adressiert werden.

- Wie die regelmäßige Entdeckung neuer Schwachstellen zeigt, sind Programmierfehler in Anwendungen und im Kernel nicht auszuschließen. Eine schnelle Behebung von Sicherheitslücken ist nur dann möglich, wenn diese den Entwicklern bekannt sind. Vor sogenannten Zero-Day-Exploits existiert kein Schutz.
- Standardmäßig kontrolliert Linux den Zugriff auf Ressourcen anhand der Identität des anfragenden Users. Diese sogenannte *Discretionary Access Control* (DAC) implementiert eine einfache Form des Konzepts *Access Control List* (ACL). Wie folgendes Beispiel zeigt, kann ein nicht-privilegierter \bar{c} kann die DAC einfach umgehen.

Ping und andere Tools werden standardmäßig mit Rootrechten ausgeführt und können von beliebigen Nutzern im System gestartet werden. Die Rootrechte bewirken, dass die DAC Programmen dieser Art jeglichen Zugriff erlaubt. Damit ist auch \bar{c} fähig, Programme wie *Ping* auszuführen und über deren Sicherheitslücken mit Rootrechten auf das Hostsystem zu wirken [189, S.26].

Sicherheitslücken, insbesondere Zero-Day-Exploits in weit verbreiteten Programmen, bildet dadurch eine große Gefahr für den sicheren Betrieb von Linux und Containern. Es macht daher Sinn das Betriebssystem präventiv zu schützen, indem der potentielle Schaden genannter Gefahren auf Basis des *Principle Of Least Privilege* eingedämmt wird.

Linux bietet verschiedene Möglichkeiten an, Mechanismen auf Basis dieses Prinzips zu nutzen. Diejenigen Möglichkeiten, für die Docker eine individuelle Unterstützung bietet, werden in diesem Kapitel vorgestellt. Einige der erklärten Mechanismen bieten anpassbare Sicherheitskonfiguration, z.B. in der Form von Profilen, an. Dadurch ist eine Begrenzung oder Ausdehnung von Rechten systemweit oder individuell pro Container möglich.

Es existieren noch weitere Sicherheitserweiterungen für den Linux-Kernel, wie z.B. *grsecurity* [59]. Durch deren anwendungsunabhängige Natur erfordern diese allerdings keine Unterstützung seitens Docker [30]. Aus diesem Grund wird diese Art von Sicherheitserweiterung in dieser Arbeit nicht weiter betrachtet.

Unter Linux sind Prozesse und Nutzer unter Linux eng gebunden. Jeder Prozess agiert in der Rolle eines Nutzers, von dem er gestartet wurde. Zur Veranschaulichung werden in den folgenden Abschnitten Prozesse, die als Akteure Zugriffe erfragen können, als Subjekte bezeichnet. Ressourcen, auf die ein Zugriff erfolgen kann, sind im Gegensatz dazu als Objekte notiert. Objekte umfassen alle jene Ressourcen, die von einem Kernel in interne Kernelobjekte aufgelöst werden können, also z.B. Dateien, Verzeichnisse, Sockets, Geräte, etc. Der Zugriff eines Subjekts auf ein Objekt wird zugelassen, sofern das Subjekt das dafür notwendige Recht - oder gleichbedeutend: Privileg - be-

sitzt.

4.3.1 Capabilities

Capabilities sind ein Feature, um den traditionellen DAC-Mechanismus zu verfeinern. Es existieren verschiedene Definitionen des Begriffs „Capability“. In dieser Arbeit sind unter diesem Begriff die POSIX-Capabilities gemeint, die seit Version 2.2.11 des Kernels in Linux fest integriert sind [189, S.42].

Als Linux-natives Feature werden Capabilities nicht nur von Docker-Prozessen direkt verwendet, sondern auch die beiden MACs *AppArmor* und *SELinux* machen indirekt Gebrauch von ihnen. Letztere sind in den Abschnitten 4.3.2.1 und 4.3.2.2 beschrieben.

Capabilities verfeinern die DAC, indem alle unter Linux erwirkbaren Rechte in kleine Einheiten unterteilt sind. Genauer gesagt ermöglicht das Feature, dass die dem Rootnutzer zustehenden Rechte in individuelle, voneinander unabhängige Einheiten unterteilt werden. Jede privilegierte Aktion ist auf eine Capability abgebildet. Nicht-privilegierten Subjekten ist es mithilfe von Capabilities möglich, privilegierte Operationen auszuführen, sofern sie die dafür notwendige Capability besitzen [164, S.33][189, S.39].

Insgesamt existieren aktuell 32 Capabilities, die außer ihrer Nummer - einer Zahl zwischen 0 und 31 - einen Namen tragen, der jeweils mit `CAP_` beginnt. Um z.B. einem Subjekt die Modifikation des Kernels zu verbieten, muss diesem die Capability `CAP_SYS_MODULE` entzogen sein [189, S.42].

Im Laufe der Jahre sind durch Linux-Erweiterungen weitere Privilegien entstanden, die in Summe mehr als 32 Elemente umfassen. Aus diesem Grund wurde das Funktionsspektrum von v.a. zwei Capabilities, `CAP_NET_ADMIN` und `CAP_SYS_ADMIN`, überladen, um auch die neuen Privilegien nutzen zu können [189, S.40f.].

Container sind unter Docker standardmäßig in der Lage 14 Capabilities

zu verwenden [107]. Mit den Parametern `--cap-add` und `--cap-drop` des Docker-Befehls `run` können Capabilities zusätzlich beim Startvorgang von Containern individuell genehmigt oder verweigert werden. In Version 1.10 und neuer ist jedoch zu beachten, dass das *Seccomp*-Standardprofil manuelle Änderungen des verwendeten Capability-Sets überschreibt. Mit `--cap-*` definierte Capabilities werden nur beachtet, wenn das *Seccomp*-Profil deaktiviert wurde [33]. *Seccomp* ist in Abschnitt 4.3.3 erklärt.

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

Wie bereits erwähnt, erfüllt der DAC-Mechanismus nicht moderne Anforderungen an die Sicherheit in Containersystemen, da er z.B. von Tools wie *Ping* umgangen werden kann. Aus diesem Grund integriert Docker die beiden MAC-Mechanismen *SELinux* und *AppArmor*, die anhand eines identitätsunabhängigen Regelwerks zusätzliche Sicherheit bieten. Unter Einbeziehung dieser Kontrollen kann der Zugriff von `c` eingeschränkt werden, selbst wenn dieser über Sicherheitslücken in den Besitz von Rootrechten gelangen konnte.

Die Integration der beiden MACs geschieht modular über das *Linux Security Modules*-Framework (LSM), das inzwischen standardmäßig in den Kernel eingebaut ist. Module, die in das Framework eingebettet werden, sind kombinierbar. Mehrere Sicherheitsmechanismen können so konsekutiv umgesetzt werden, um eine *Defense In Depth* umzusetzen [199, S.3].

Durch eine Erweiterung von Namespaces können Module der LSM eventuell zukünftig über Namespaces verwaltet werden.

Die Kontrolle unter LSM geschieht, wie in Abb.13 dargestellt, in Form eines LSM-Hooks. Unter einem Hook ist ein zwischengeschaltener Aufruf gemeint, der einen *System Call* unterbricht und die Weiterverarbeitung dessen in einem Sicherheitsmodul, wie *SELinux*, anstößt. Erst nach einer Antwort eines oder mehrerer hintereinander geschaltener Module, wird die normale Weiter-

ausführung des *System Calls* fortgesetzt bzw. für den Fall, dass das Modul die Weiterausführung blockiert, verweigert.

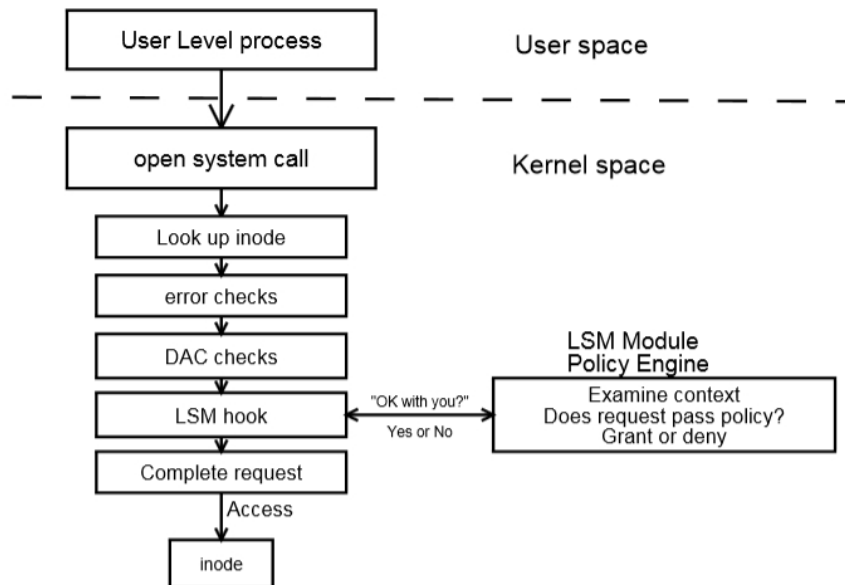


Abbildung 13: Funktionsweise von *System Call*-Hooks im LSM-Framework [199, S.3].

Der Ausführungszeitpunkt des Hooks bietet den Vorteil, dass der komplette Kontext der Zugriffsanfrage an dieser Stelle vorliegt und vollständig von einem LSM-Modul ausgewertet werden kann [199, S.2]. Neben der Kompatibilität zum DAC-Mechanismus, wird dadurch zusätzlich die Granularität der Zugriffskontrolle verbessert [198].

An dieser Stelle ist wichtig zu erwähnen, dass Module des LSM-Frameworks den nativen DAC-Mechanismus nicht überschreiben, sondern ergänzen. Wie in Abb.13 dargestellt, greift der LSM-Hook erst nachdem ein Nutzer einen sicherheitskritischen *System Call* ausführt hat, das angefragte Objekt aufgelöst, ein Fehler-Check abgeschlossen und der Zugriff über den DAC-Mechanismus genehmigt wurde. Erst wenn der Kernel versucht auf das aufgelöste Kernelobjekt zuzugreifen, wird der Hook ausgeführt, der den Zugriff in das zugehörige LSM-Modul weiterleitet. Das Modul genehmigt oder verweigert den Zugriff anhand den ihm vorliegenden Attribute im Sicherheitskontext.

Durch diese Reihenfolge ist sichergestellt, dass die Nutzung einer LSM-Schnittstelle optional ist und unabhängig von der DAC funktioniert [46]. Deswegen können auch Anwendungen, die das LSM-Framework nicht unterstützen, weiterhin funktionieren. Außerdem ist durch den rein erweiternden Charakter die Gefahr ausgeschlossen, mit *SELinux* oder *AppArmor* neue Sicherheitslücken in das System einzuführen.

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 [199, S.3].

Bei der Verwendung von Sicherheitsmodulen ist auf die Leistungsverluste zu achten, die die einzelnen MACs verursachen. Nach den durchgeführten Benchmarks in [189, S.51ff.] kommt es unter *AppArmor* und *SELinux* bei der Ausführungsgeschwindigkeit der meisten Testoperationen zu Verlusten zwischen 0% und 11%. Spezielle Aktionen, wie z.B. das Öffnen und Schließen einer Datei unter *AppArmor*, können die Ausführungszeit verdoppeln.

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

In den folgenden Unterkapiteln sind die beiden MAC-Implementierungen *AppArmor* und *SELinux* vorgestellt. Exemplarisch wird das Standardprofil von *AppArmor*, das Docker verwendet, im Detail analysiert. Wie sich MACs unter Docker in Zukunft entwickeln, ist im Fazit in Kapitel 7 geschildert.

4.3.2.1 AppArmor

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

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

ben kombinierbare Berechtigungstypen festlegen [120][12]. Durch eine Inkludieranweisung lassen sich mehrere Profile modular kombinieren [12].

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

AppArmor unterstützt drei Betriebsmodi, die jeweils folgenden Zweck erfüllen [189, S.82]:

- *Audit*-Modus: Alle erlaubten Zugriffe werden protokolliert.
- *Complain*-Modus: Ein Lernmodus, bei dem das Verhalten einer Anwendungen beobachtet wird und aus diesem ein automatisch generiertes Sicherheitsprofil erstellt wird [120]. In diesem Modus werden Zugriffe, die gegen die Profilregeln verstoßen, nur aufgezeichnet und nicht unterbunden.
- *Enforce*-Modus: Die Regeln eines Profils werden erzwungen. Verstöße werden protokolliert.

Das Docker-Standardprofil `docker-default` [9], das im *Enforce*-Modus in nicht-privilegierten Containern zum Einsatz kommt [155], wird bei der Installation von Docker in die Datei `/etc/apparmor.d/docker` geschrieben. Administratoren können sich mit dem Befehl `sudo aa-status` vergewissern, ob das Standardprofil aktuell aktiv ist. Auch die zurückgegebenen Sicherheitskontexte in der ersten Spalte von Abb.11 und Abb.12 belegen die standardmäßige Verwendung von `docker-default`. Es existiert auch ein Profil für den Docker-Daemon [10], allerdings muss dieses manuell aktiviert werden [11].

`docker-default` kann mit dem `run`-Parameter `--security-opt="apparmor:PROFILE"` manuell überschrieben werden, sofern `PROFILE` zuvor, z.B. mit dem CLI-

Werkzeug `apparmor_parser` [109], importiert wurde [33].

Das Standardprofil `docker-default` wurde während der Erstellung dieser Arbeit aktualisiert. Aus der Commit-Nachricht geht allerdings nicht hervor, ob damit eine Sicherheitslücke geschlossen wurde oder die Änderungen im Zuge einer Funktionsänderung von Containern entstanden sind [15].

Die aktuelle Implementierung des Profils, die auf einem Docker-Hostsystem lokal mit dem Konsolenbefehl `cat /etc/apparmor.d/docker` ausgegeben werden kann, wird im Folgenden gruppiert und chronologisch in der Reihenfolge des Vorkommens analysiert.

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

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

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

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

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

Hiermit werden einige Zugriffsregeln eingebunden, die von fast allen Programmen benötigt werden [189, S.100].

```
network ,
capability ,
file ,
umount ,
```

Diese Regeln erlauben die grundsätzliche Verwendung von Netzwerkfunktionen, Capabilities und dem Dateisystem. Außerdem können Mounts mithilfe von `umount` ausgeworfen werden.

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

Diese Anweisungen regeln den Schreibzugriff im Verzeichnis `/proc/`. Effektiv wird das Schreibrecht auf alle Dateien und Verzeichnisse untersagt, die nicht in `/proc/<number>/` und `/proc/sys/kernel/shm*` liegen. Erstere, numerischen Verzeichnisse geben Zugriff auf prozessspezifische Daten. Jede Nummer repräsentiert eine Prozess-ID der aktuell laufenden Prozesse. Dateien im Verzeichnis `/proc/sys/kernel/`, deren Name mit `shm` beginnt, beziehen sich auf geteilte Speicherbereiche, den *Shared Memory* von Prozessen [94].

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

Mittels dieser Direktiven wird Docker das Lese-, Schreib-, Dateisperrungs-, Linkerzeugungs- und Ausführrecht (in dieser Reihenfolge) für angegebene Dateien verwehrt. Mithilfe von `sysrq-trigger` können Aktionen programmatisch ausgeführt werden, die auch über die *Magischen S-Abf-Tasten* bewirkt werden können. Darunter fallen u.a. kritische Befehle zur Terminierung von Prozessen oder dem Neustart des Betriebssystems [76][93].

Die Einträge `mem`, `kmem` und `kcore` beinhalten Speicherbereiche des Kernels.

```
deny mount,
```

Diese Zeile verbietet das Einbinden jeglicher Mountpoints.

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

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

Zusätzlich darf von Dateien und Unterordnern in `/sys/firmware/efi/efivars/` und `/sys/kernel/security` nicht gelesen werden. Das erstgenannte Verzeichnis bietet Informationen über einige Bootvariablen. Zweitgenanntes gewährt Einblick in die aktuelle Konfiguration von Sicherheitsmodulen wie *AppArmor* und *SELinux* [112][75][102].

4.3.2.2 SELinux

SELinux implementiert eine feingranulare MAC, die ursprünglich von der NSA entwickelt wurde. Es wird verwendet, um mithilfe der Konzepte *Type Enforcement* und *Multi-Level Security* (MLS) Anforderungen an die Vertraulichkeit und Integrität zu erfüllen [115].

Der wesentliche Unterschied zu *AppArmor* ist, dass die Zugriffsverwaltung unter *SELinux* über Label erfolgt, die Subjekten und Objekten in einem Sicherheitskontext angehängt sind.

Auch unter *SELinux* beruhen die Regeln auf einem Profil, das in diesem Kontext auch Regelwerk genannt wird. Das Regelwerk besteht aus Anweisungen, die konkrete Sicherheitslabel, wie sie im nächsten Abschnitt beschrieben sind, abbilden. Durch die in Abb.14 dargestellte strikte Trennung des Regelwerks und dessen Durchsetzung, lassen sich mit *SELinux* hohe Sicherheitsanforderungen erfüllen. Durch detailreiche Anpassungsmöglichkeiten steht diese MAC unter dem Ruf, schwer konfigurierbar zu sein. GUI-Werkzeuge, wie *system-config-selinux* und die Bibliothek *libsemanage*, verbessern die Bedienung von *SELinux* und dessen Regelwerken [151, S.62,67].

SELinux kennt das Konzept der DAC von Besitzern und Gruppen nicht. Der komplette Funktionsumfang von *SELinux* beruht auf einem Labelingsystem, das Zugriffe individuell für Subjekte verwaltet [158]. In Zuge dessen wird jedem Subjekt zur Laufzeit und jedem Objekt im System ein Label nach dem Schema **User:Role:Type:Level** zugewiesen [23]. Die erste Komponente **User** eines Labels ist von einem Nutzer unter Linux unabhängig.

Das Label ist in die erweiternden Attribute (**xattr**) von Subjekten und Objekten geschrieben [151, S.65]. In Abb.14 ist das Label als *SC* (Security-Context) illustriert.

SELinux wertet bei einem Zugriff das Label des zugreifenden Prozesses und das Label der betroffenen Ressource anhand einem definierten Regelwerk aus und entscheidet, ob die Operation fortgesetzt werden darf [120].

Wie unter *AppArmor*, bietet auch *SELinux* Möglichkeiten zum Logging von Zugriffen an.

Die Linux-Distributionen *Red Hat Enterprise Linux*, *Fedora* und *CentOS* verwenden *SELinux* standardmäßig [30]. Unter anderen Distributionen wie *Debian* und *Ubuntu* kann *SELinux* manuell installiert werden [19][20].

Seit November 2015 Docker unter dem Release 1.9 mit einer standardmäßigen

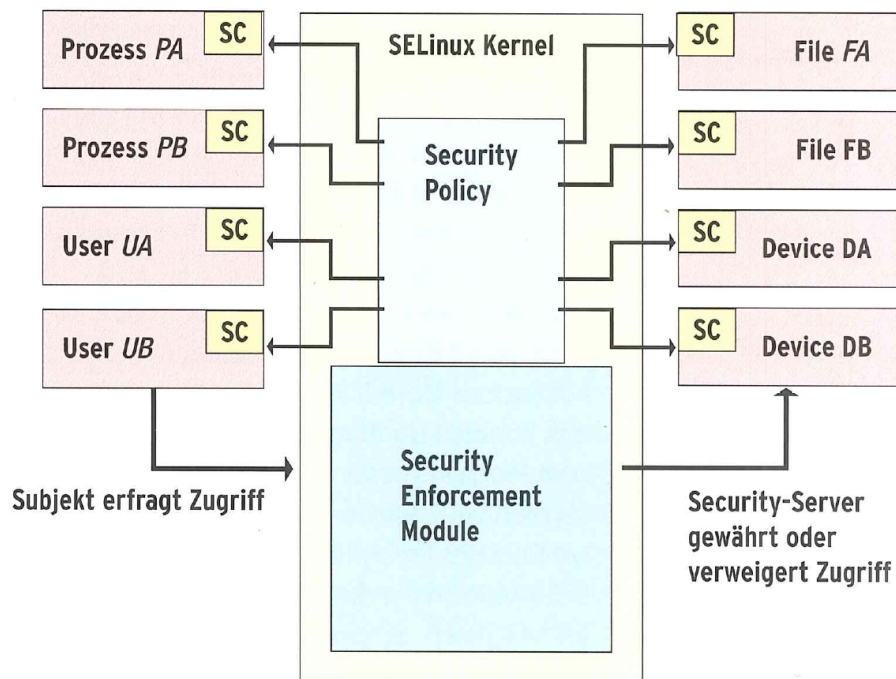


Abbildung 14: Trennung von Regelwerk und Enforcement-Modul. Zuweisung von Security-Contexts (SC) an Objekte und Subjekte [151, S.63].

SELinux-Policy ausgestattet [48][3]. Dieses standardmäßige Regelwerk kann über [103] aufgerufen werden.

Unter Docker können die vier Labelattribute mit dem Parameter `--security-opt="label:LABEL"` für den `run`-Befehl pro Container manuell spezifiziert werden [33].

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

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

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

In der Praxis existieren für jede Anwendung eigene Typen, da jede Anwendung spezielle Rechte benötigt, um ihre Funktion zu erfüllen. Jedem Docker-Prozess ist z.B. der Typ `docker_t` zugewiesen, der gleichzeitig eine gemeinsame Domäne für besagte Prozesse definiert. Im *SELinux*-Regelwerk ist festgelegt, dass Prozesse eines Typs nur auf Objekte vollen Zugriff hat, die mit bestimmten Labeltypen versehen sind. Im konkreten Fall von `docker_t` umfasst diese Konfiguration ein Objektset, das u.a. aus den Typen `docker_config_t`, `docker_log_t` und `svirt_sandbox_file_t` besteht [104]. Versucht ein Docker-Prozess auf Objekte zuzugreifen, deren Typ nicht in [104] gelistet ist, wird die Operation unterbunden.

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

Multi-Category Security (MCS) Durch das einheitliche Regelwerk für alle Docker-Prozesse, ist mit einem *Type Enforcement* gewährleistet, dass ein kompromittierter Container \bar{c} nicht unbefugt auf geschützte oder unrelevante Daten außerhalb der Docker-Domäne zugreifen darf. Docker-unspezifische Subjekte und Objekte werden dadurch im Hostsystem geschützt.

MCS implementiert eine Möglichkeit, \bar{c} den Zugriff auf *czu* verweigern. Mit *Type Enforcement* ist das nicht realisierbar, da \bar{c} und c beide durch den identischen Typ `docker_t` der Docker-Domäne zugehörig sind.

Theoretisch wäre die Funktion der MCS auch mit dem *Type Enforcement* realisierbar, wenn jeder Container mit einem eigenen Typ operiert. Die Domäne

würde dabei nicht für alle Docker-Subjekte gelten, sondern nur für solche eines Containers. Diese feinere Typenunterteilung wirkt sich aber auf die Komplexität des Regelwerks negativ aus. In der Praxis wird deswegen der MCS-Mechanismus für dieses Sicherheitsfeature eingesetzt.

Der MCS-Mechanismus arbeitet mit dem letzten Teil des Labels: dem Level. Das Level unterteilt sich durch den Aufbau 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 von Docker nicht verwendeten MLS Anwendung findet. Im Fall von Docker hat die Schutzstufe deswegen einen konstanten Wert von `s0` [98][98].

Beim Startvorgang eines Containers wird diesem eine zufällige Kategorie anhand einer Nummer zwischen 0 und 1023 zugewiesen. Diese Kategorie, z.B. nach obiger Syntax `c623`, wird daraufhin auch vom Docker-Daemon auf den Inhalt containerspezifischer Verzeichnisse angewandt. Sobald während des Betriebs ein Container den Zugriff auf ein Objekt fordert, wird seine Kategorie mit dem des angefragten Objekts verglichen. Stimmen diese überein, ist die MCS-Kontrolle 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 [178, S.200f.].

Damit erfüllt MCS die zuvor beschriebene Anforderung. Einem \bar{c} ist es ihm Rahmen einer korrekt implementierten MCS unter *SELinux* nicht mehr möglich, Schaden an zu schützenden Containern aus *C* auszuüben.

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 für Anwendungen bestimmte *System Calls* blockiert

werden. Die Gefahr, dass fehlerbehaftete oder unsichere *System Calls* genutzt werden, die die Anwendung zum fehlerfreien Betrieb nicht benötigt, wird dadurch reduziert [120][70][160].

Seccomp ist nicht wie *SELinux* und *AppArmor* als LSM implementiert [136].

Die Verwendung von *Seccomp* für einen Prozess bewirkt, dass dieser in einen „sicheren“ Zustand übergeht, sodass er nur noch zuvor 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 [120].

Ein Update *Mode 2* macht das Set an erlaubten *System Calls* mithilfe von Filtern frei konfigurierbar und führt ein *Audit Logging* ein [120][70].

Mithilfe der *Seccomp*-Anweisungen `allow`, `deny`, `trap`, `kill` und `trace` sind neben der Sperrung noch weitere Aktionen möglich, die zur Kontrolle von *System Calls* dienen [155].

Seit Oktober 2015 ist eine *Seccomp*-Unterstützung für Docker in Planung und Entwicklung [154][90]. Diese wurde in Form eines *Seccomp*-Standardprofils und der Option eigene Profile einzubinden, in aktuell neusten Docker-Version 1.10 im Februar 2016, hinzugefügt [48][101][100][155]. Das Standardprofil basiert seit Ende 2015 auf einer Whitelist (davor einer Blacklist). Dadurch blockiert es alle *System Calls*, die nicht in der Whitelist als erlaubte Operationen aufgeführt sind [101].

Eine aktuelle Liste der explizit erlaubten *System Calls* ist in [100] und [101] zu finden.

Seit der Umstellung von Blacklisting auf Whitelisting wurde die *System Call*-Auflistung in einem Zeitraum von ca. fünf Wochen 47 Änderungen unterzogen. Davon sind 40 Neueinträge und sieben Löschungen zu registrieren [16]. Während es sich 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 [33]. Jedoch ist an anderer Stelle im *GitHub*-Repository vermerkt, dass sich das Standardprofil mit dem Parameter `--security-opt seccomp:PROFILEPATH` überschreiben lässt [101]. Ist es nötig, Container ohne *Seccomp* zu starten, kann das mit der Option `--security-opt seccomp:unconfined` realisiert werden [33][155].

Kapitel 5

Sicherheit im Docker-Ökosystem

In diesem Kapitel werden einige Anwendungsaspekte von Docker aus Sicherheitssicht beleuchtet. Maßgeblich sollen weiterhin die in Abschnitt 2.2 definierten Sicherheitsziele zur Bewertung von Docker-Eigenschaften dienen. Während Kapitel 4 native Sicherheitskomponenten von Docker und Linux untersucht, wird in den folgenden Abschnitten das Docker-Ökosystem in Betracht gezogen. Darunter versteht sich die Gesamtheit aller Komponenten und Interaktionsmöglichkeiten, die im Zusammenhang mit Docker existieren. Der Fokus der Untersuchung liegt auf Anwendungsebene. Das bedeutet, dass hauptsächlich Methoden vorgestellt werden, die Docker Entwicklern und Administratoren zur Verfügung stellt, um die Arbeit mit den Docker-Komponenten aus Kapitel 2.3 sicher zu gestalten. Außerdem wird vorgestellt, wie sich die sicherheitsrelevanten Komponenten und Operationen in den letzten Monaten verändert haben.

5.1 Private Registries

Docker bietet neben der Nutzung des öffentlichen *Docker Hubs* an, private Registries zu erstellen. Diese können dann, z.B. von einer Firewall gesichert, in einer unternehmenseigenen Infrastruktur oder in Rechenzentren externer Public-Cloud-Anbieter betrieben werden. Private Registries können als Container betrieben werden [25].

Für Cloud-Anbieter stellt Docker einige Speichertreiber zur Verfügung, z.B. für Amazons S3 [99], Microsofts Azure [78], und OpenStacks Swift [86]. Bei Bedarf können eigene Speichertreiber mithilfe einer API implementiert werden [41].

Neben der Vertraulichkeit von Images, bieten private Registries den Vorteil, dass sich die Speicherung und Verteilung von Images an den internen und häufig durch *Continuous Integration* und *Continuous Delivery* automatisierten Softwareentwicklungsprozess anpassen lassen. .

Außerdem lässt sich das *Docker Hub* in einer privaten Registry spiegeln. Bei dem Herunterladen von Images aus der öffentlichen Registry, kann somit auf eine externe Verbindung verzichtet werden, sofern die Spiegelung in Form einer privaten Registry im eigenen Netz existiert. Docker kann mit der Option `--registry-mirror=ADDRESS` angewiesen werden, anstelle des *Docker Hubs* eine Spiegelung zu verwenden [95]. Diese Form der Redundanz ist eine Maßnahme, um die Verfügbarkeit von Images sowie Geschwindigkeit des Downloads dieser zu erhöhen.

Der Zugriff auf eine Registry kann über HTTPS und der Verwendung von Zertifikaten abgesichert werden [25] (vgl. Abschnitt 5.3).

5.2 Verifikation und Verteilung von Images

Der Sicherheitsforscher Jonathan Rudenberg hat im Dezember 2014 drei Sicherheitsrisiken im Zusammenhang mit Dockers damaliger Verifikation und

Verteilung von Images aufgedeckt [183][182]. U.a. ist es durch Verwendung des `docker pull`-Befehls möglich manipulierte Images zu beziehen, die bereits beim Entpacken auf dem lokalen Hostsystem beliebige Dateien überschreiben können [162]. Sowohl die Datenintegrität als auch die Verfügbarkeit des Hostsystems sind durch eine solche Sicherheitslücke direkt gefährdet. Wie in Abschnitt 5.2.2 zu sehen ist, kann auch die Aktualität von Images sowie die Authentizität von Personen und Organisationen, die Images veröffentlichen, darunter leiden.

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

5.2.1 Verifikation von Images

Seit Februar 2016 sind mit der Veröffentlichung von Docker-Version 1.10 Images über deren Inhalt adressierbar. Auf Implementierungsebene bedeutet das, dass die Layer nicht mehr über zufällig generierte UUIDs referenziert werden. Es werden SHA256-Hashwerte über die Layerdaten gebildet werden [139, S.16]. Der SHA256-Hashalgorithmus wird derzeit als kryptographisch sicher gesehen. Folglich sind die generierten Layer-IDs kollisionsicher und damit einmalig.

Durch die deterministische Natur von Hashfunktionen wird gleichzeitig eine Methode implementiert, die die Integrität der Schichten sicherstellt. In der Praxis kann die Korrektheit von Daten der Schichten validiert werden, indem ein frisch berechneter Hash einer Schicht mit dem referenzierten Hasheintrag in den Image-Metadaten (Manifest) verglichen wird. Die referenzierten Hashwerte der Schicht sind im Manifest in Form eines Hashbaums strukturiert. Seit der Version 2 des Manifests, kann die Manifestdatei signiert werden, um auch die Integrität der Metadaten zu gewährleisten [122].

Die zuvor verwendeten UUIDs erfüllen die deterministische Eigenschaft nicht, da sie unabhängig vom Inhalt bei jeder Generierung zufällig auf Basis der PRNG-Implementierung in *Golang* entstehen [140].

5.2.2 Integration von *The Update Framework*

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

Im August 2015 wurde mit der Docker-Version 1.8 ein Paket- und Verteilungsmodell umgesetzt, das die von Rudenberg entdeckten Schwächen in der Bereitstellung von Images beheben soll [173]. Unter dem Featurenamen *Docker Content Trust* integriert Docker das Modell *The Update Framework* (TUF) [62], welches Gefahrenquellen wie manipulierte Images, Replay- und MITM-Angriffe ausschließt. Die Sicherheit von TUF basiert auf der Signierung von Images, mit der anhand mehrerer kryptographischer Schlüssel die Integrität, Authentizität sowie Aktualität von Images sichergestellt wird. Die Verwendung dieses Features ist optional und kann mit der Umgebungsvariable `DOCKER_CONTENT_TRUST` gesteuert werden. *Docker Content Trust* wird in Docker als Notary integriert. Der Notary implementiert das TUF und bietet Erstellern von Inhalten die Möglichkeit ihre Daten zu signieren. Die signierten Daten können dann über einen Notary-Server zum Download angeboten werden [47][173].

5.3 Verbindung zwischen Daemon und Client

Wie in der Architektur von Docker in Abschnitt 2.3.1 dargestellt, werden Anweisungen von Docker-Clients an einen Docker-Daemon übertragen, der diese über eine REST-API empfängt. Standardmäßig findet diese Kommunikation über einen nicht netzwerkfähigen Unix-Socket statt [30].

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

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

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

Die entsprechende Konfiguration eines Daemons kann z.B. mit dem Befehl `docker daemon --tlsverify --tlscacert=CA.pem --tlscert=SERVER-CERT.pem --tlskey=SERVER-KEY.pem` vorgenommen werden. Analog dazu erfolgt die clientseitige Einstellung über `docker --tlsverify --tlscacert=CA.pem --tlscert=CERT.pem --tlskey=KEY.pem COMMAND`. Der Parameter `--tlsverify` gibt jeweils an, dass der Kommunikationspartner authentifiziert werden muss. Die Authentifikation geschieht über die Parameterwerte `--tlscert` und `--tlskey` des Kommunikationspartners, die zusammen die Identität dessen bekannt geben. Unter Angabe eines CA-Zertifikats mit Parameter `--tlscacert` hat die Authentifizierung nur dann Erfolg, wenn das Zertifikat des Kommunikationspartners von dieser CA ausgestellt wurde [28]. In einer Unternehmensinfrastruktur kann so die Kommunikation durch eine unternehmenseigene CA weiter eingegrenzt werden. Eine detailreichere Beschreibung der verschiedenen Betriebsmodi ist unter [28] gegeben.

Über die Umgebungsvariable `DOCKER_TLS_VERIFY` sowie der Speicherung der notwendigen Zertifikate und Schlüssel unter `.docker/` im Homeverzeichnis, kann die Konfiguration der Authentifizierung einmalig für die zukünftige Kommunikation vorgenommen werden [28].

5.4 Docker Plugins

Seit Juni 2015 unternahmen die Entwickler von Docker Anstrengungen, um optionale Komponenten von Docker in eine eigene Plugininfrastruktur zu integrieren, in der Plugins modular aktiviert und deaktiviert werden können [48][161]. Plugins werden von einem Docker-Daemon genutzt und erweitern dessen Fähigkeiten. Neben den ersten Plugins für diverse Netzwerkfunktionen, z.B. *Weave*, und der Einbindung von Datenträgern, z.B. *Flocker*, fand im Frühjahr 2016 mit Docker-Version 1.10 auch ein ursprünglich von *Twistlock*[63] entwickeltes Authorisierungsplugin Einzug in Docker. Zur Vereinfachung wird dieses im Folgenden als *AuthZ* bezeichnet [128][161][166].

Ergänzend dazu war auch ein Authentifizierungsplugin *AuthN* geplant, das Nutzer, vor deren Authorisierung durch *AuthZ*, authentifiziert [197]. Am 23. Februar 2016 hat jedoch ein Docker-Mitarbeiter bekannt gegeben, dass die Integration von *AuthN* eingestellt wird. Grund hierfür ist, dass die Authentifizierung - nach der Meinung einiger Docker-Entwickler - leicht außerhalb des Daemons stattfinden kann, z.B. mit dem Authentifizierungsdienst Kerberos [60][193][134].

Das Sicherheitsplugin *AuthZ* hat zum Ziel ein Framework bereitzustellen, über das es Administratoren möglich ist, eine nutzer- und rollenbasierte Sicherheitspolitik (RBAC) umzusetzen. Diese umfasst Regeln, die die Benutzung des Docker-Daemons betreffen. Nach der ursprünglichen Implementierung von *Twistlock* sind die Regeln in eine JSON-Struktur gefasst [167]. Ohne ein solches Plugin ist jedem Nutzer, der den Docker-Daemon ausführen kann, die komplette Kontrolle über das Docker-System gegeben. V.a. in Unternehmen macht es aber Sinn, verschiedenen Nutzer im Rahmen eines RBAC-Sicherheitsmodells eine bestimmte Rolle zuzuweisen, die deren Rechte definiert. Ein einfacher Anwendungsfall könnte folgende Regeln beinhalten [166]:

- User in der Gruppe *Operations* dürfen nur Container starten und stoppen. Sie sollen nur `docker run CONTAINER` und `docker rm CONTAINER`

ausführen können.

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

Aus Sicht der Architektur funktioniert die Authorisierung, wie sie in Abb.?? illustriert ist. Die Anfragen von lokalen oder entfernten Clients werden, wie in Kapitel 2.3.1 beschrieben, an einen Daemon geschickt. Dieser führt nun nicht die Befehle der Clients umgehend aus, sondern leitet die Anfrage an das *AuthZ*-Framework weiter. Genauer erhält *AuthZ* einen Nutzerkontext und einen Befehlskontext, die, anhand der zuvor definierten Regeln, ausgewertet werden. Anhand der dem Plugin vorliegenden Parameter entscheidet es, ob der Nutzer berechtigt ist, den angefragten Befehl auszuführen. Falls das nicht der Fall ist, wird eine Fehlermeldung über den Daemon an den Client gesendet (vgl. Abb.??). Falls die Anfrage genehmigt wurde, führt der Daemon den darin enthaltenen Befehl aus und kontaktiert das Plugin ein zweites Mal mit dem Ergebnis des ausgeführten Befehls (vgl. Abb.??). *AuthZ* hat hierbei die Gelegenheit, die Antwort zu modifizieren bevor sie zum Client gesendet wird [194][166].

Die Struktur von Anfragen und Antworten, die über das HTTP-Protokoll ausgetauscht werden, sind in [124] spezifiziert.

Mehrere Sicherheitsmodule können konsekutiv ausgeführt werden, sodass jeder clientseitigen Interaktion mit dem Daemon, die Ausführung mehrerer *AuthZ*-Implementierungen folgt.

Mit folgender Syntax können Authorisierungs-Plugins für den Daemon aktiviert werden [124]:

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

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

5.5 Open-Source-Charakter und Sicherheitspolitik von Docker

Docker ist wie Linux ein Open-Source-Projekt, das nicht nur von Docker-Mitarbeitern entwickelt wird, sondern die Unterstützung vieler freiwilliger Entwickler und Sicherheitsexperten findet. Der Open-Source-Charakter von Docker und Software allgemein kann jedoch aus Sicherheitssicht kontrovers diskutiert werden:

Vorteile:

- Durch die Vielzahl an Involvierten können Sicherheitslücken schneller entdeckt und behoben werden. Über 1300 Individuen haben sich bislang für die Weiterentwicklung der Codebasis und der Aufdeckung von Fehlern und Sicherheitsrisiken an Docker beteiligt.
- Eigene Sicherheitspraktiken können von Anwendern durch die Kenntnis über den Quellcode hinzugefügt werden.
- Nach dem *Kerckhoffs'schen Prinzip* ist einer Anwendung von *Security Through Obscurity* generell abzuraten. Ein offenes Design von Serversystemen ist von NIST als allgemeines Sicherheitsprinzip vorgeschlagen [185, S.15].

Nachteile:

- Sicherheitslücken sind von Angreifern prinzipiell schneller zu finden, wenn diese im Besitz des Quellcodes sind.
- Durch die lose Zusammenarbeit vieler Experten ist nicht gewährleistet, dass bestimmte Implementierungen nach dem Mehraugenprinzip kontrolliert werden. Die böswillige Absicht nur eines Entwicklers kann in einer Open-Source-Kultur weiterhin starke Beeinträchtigungen der Sicherheit mit sich bringen.

Die Vor- und Nachteile zeigen auf, dass es keinen eindeutigen Gewinner nach Maßstäben der Sicherheit gibt. Sowohl Open-Source- als auch proprietäre Softwarelösungen können erfolgreich sein, wie die Vergangenheit bekannter Betriebssysteme und Anwendungen zeigt. Für *Docker* hat sich der Open-Source-Ansatz bislang bewährt.

Um die Zusammenarbeit im Docker-Projekt zu organisieren, wurden Regeln und Konventionen formuliert, die auch die Sicherheit betreffen [17]. Zusammen mit den Sicherheitsrichtlinien aus der offiziellen Docker-Homepage, ergeben sich folgende weitere, sicherheitsrelevante Eigenschaften von Docker.

- Prinzip von *Responsible Disclosure* wird als Hauptbestandteil der Sicherheitspolitik von Docker umgesetzt. Konform dazu, können entdeckte Schwachstellen jederzeit an eine zentrale E-Mail-Adresse kommuniziert werden [42].
- Wartung einer zentralen CVE-Datenbank, die bekannte Sicherheitsrisiken von Docker enthält. Die darin veröffentlichten Informationen erfüllen das Prinzip *Responsible Disclosure* [24].
- Externe Sicherheitsfirmen werden vierteljährlich beauftragt, Sicherheitsaudits und Penetrationstests zur Kontrolle der Codebasis und Infrastruktur durchzuführen [141, S.5].

5.6 Best-Practices für die Sicherheit

In den folgenden Abschnitten sind verschiedene Best-Practices im Umgang mit Docker aufgezählt.

Während insgesamt weitaus mehr Best-Practices existieren, sind im Folgenden nur jeden ausgewählt, die in den meisten Anwendungsfällen angewandt werden können.

5.6.1 Skripte

Es existieren Skripte, die Docker-Hostsysteme nach eingehaltenen und nicht erfüllten Best-Practices überprüfen.

Der bekannteste Vertreter ist *Docker Bench* [175], welcher in das Docker-Projekt integriert ist und nach Aussagen der Entwickler auf Befunden des Sicherheitsberichts des unabhängigen *Center for Internet Security* beruht [150]. Dieser Bericht ist im Mai 2015 entstanden und enthält sicherheitsrelevante Befunde zu Docker in der Version 1.6. Aktuellere Berichte zu moderneren Docker-Versionen existieren nicht.

Die darin enthaltenen Kontrollen sind in einem eigenen Image als Shellskripte definiert. Zur Ausführung der Tests muss dieses Image als Container gestartet werden. Im Rahmen dieser Tests werden direkt über Shellbefehle abrufbare Parameter untersucht. Darunter sind z.B.:

- Rechte, Besitzer- und Gruppenzugehörigkeiten von Dateien und Verzeichnissen
- Docker- und Kernelversion
- Log- und Netzwerkeinstellungen
- TLS-Unterstützung
- Existenz von (nicht-)privilegierten Containern
- Verwendung von Capabilities, *AppArmor*- und *SELinux*-Profilen

Die konkreten Tests sind unter [174] in ihrer aktuellsten Version im Detail veröffentlicht.

5.6.2 Datencontainer

Die von Werkzeugen wie *Docker Bench* ausführbare Tests beschränken sich auf explizit abrufbare Parameter des Hosts und von Docker. In Serverinfrastrukturen existieren jedoch auch Sicherheitskriterien, deren Kontrolle nicht über Abfragen von Einstellungen des Hostsystems oder von Docker abzudecken ist. V.a. Merkmale der Infrastruktur wirken sich auf die Verfügbarkeit von Diensten sowie der Wartbarkeit einer Infrastruktur aus.

Aus diesem Grund wird von *Docker* empfohlen, eigene Datencontainer zu betreiben, über die Anwendungen anderer Container ihren Zustand auf Datenebene persistieren können. Anwendungen können in diesem Szenario über mehrere Container hinweg auf den Datencontainer zugreifen, sofern letzterer Methoden implementiert, um Race-Conditions zu verarbeiten. Standardmäßig gehen Änderungen, die während dem Betrieb von Containern auftreten, verloren, sobald der Container gestoppt wird. Diese Eigenschaft beruht auf der obersten Schicht eines Containers, die bei Containerstart einem Image hinzugefügt wird. Dieser Layer ist unabhängig von dem zugrundeliegendem Image und existiert nur temporär zur Laufzeit des Containers.

Dieser Eigenschaft unterliegen auch Datencontainer. Deshalb empfiehlt es sich, die Verzeichnisse von Datencontainern aus dem Hostsystem über Mounts einzubinden. Welche Verzeichnisse des Hostsystems über Datencontainer für Anwendungscontainern verfügbar sind, kann in Datencontainern zentral eingestellt werden [77].

5.6.3 Verwaltung von Credentials

Informationen zu Benutzernamen und Passwörtern sollten nicht statisch als Zeichenketten in ein Image integriert sein. Vielmehr sollen Credentials generell als Umgebungsvariablen mit der `ENV`-Direktive der Dockerfiles in Images verfügbar gemacht werden.

Beispielsweise setzen die Entwickler von der Datenbank *Postgres* diese Praxis in ihrem *Postgres*-Image um, indem sie die Umgebungsvariablen `POSTGRES_USER` und `POSTGRES_PASSWORD` definieren, die beim Startvorgang des Containers abgefragt werden [81][152].

5.7 Tools

Im Docker-Ökosystem existieren zahlreiche Tools, die entweder den Funktionsumfang von Docker erweitern oder zusätzliche Möglichkeiten bieten, Docker-Container zu verwalten. Neben Veröffentlichungen von *Docker*, z.B. mit *Docker Swarm*, *Docker Compose* und *Docker Datacenter*, können auch einige Entwicklungen von Drittanbietern verwendet werden. Diese umfassen beispielsweise *Kubernetes* [61], *Shipyards* [61] und *docker-slim* [92]. Die genannten Tools decken teilweise gleiche Features ab oder bauen im Fall von *Docker Swarm* und *Shipyards* aufeinander auf.

Viele erweiternde Features werden von Docker auch mittels der Schnittstelle Docker-Plugins unterstützt. Die sicherheitsrelevante Erweiterung *AuthZ* wurde bereits in Kapitel 5.4 vorgestellt.

Im Folgenden werden ausgewählte Werkzeuge kurz vorgestellt. Außerdem wird erörtert inwiefern sie zur Sicherheit von Docker beitragen oder welche von Docker unabhängigen Sicherheitsfunktionen sie anbieten.

5.7.1 Kubernetes

Als ausgewählter Vertreter der Verwaltungs- und Orchestrierungswerkzeuge, wird in diesem Abschnitt das von *Google* entwickelte Open-Source-Werkzeug *Kubernetes* vorgestellt.

Der Fokus von *Kubernetes* liegt auf der Verwaltung und Orchestrierung von Containern, die über mehrere Hosts verteilt sind. Damit eignet sich das Tool v.a. in Cloud-Infrastrukturen, welche sich i.d.R. aus mehreren, über das Netzwerk miteinander verbundenen Hostsystemen zusammensetzen. Es führt neue Konzepte ein, um eine Vielzahl von Containern auf physische und virtuelle Systemen zu verwalten.

Aus der Sicht von *Kubernetes* stellen z.B. Docker-Container die kleinste zu verwaltende Einheit dar. Welche Anwendung in einem Container läuft und welche Sicherheitsmechanismen aus Kapitel 4 der Container nutzt, um das jeweilige Hostsystem zu schützen, ist für den Betrieb von *Kubernetes* zunächst unerheblich.

Kubernetes erfüllt hauptsächlich administrative Anforderungen an die Sicherheit, die auch hier wieder auf Basis des *Principle of Least Privilege* umgesetzt werden. Damit ist die Implementierung einer rollenbasierten Nutzerkontrolle (RBAC) gemeint, die Nutzer über alle von *Kubernetes* kontrollierten Container hinweg, autorisiert und authentifiziert [147]. Im Vergleich dazu, ermöglicht die Verwendung des Authentifizierungs-Framework *AuthZ* aus Kapitel 5.4 nur die Authentifizierung von Nutzern auf einem Hostsystem.

5.7.2 docker-slim

docker-slim ist ein Werkzeug, das u.a. automatisch Sicherheitsprofile erstellt. Es untersucht dabei Anwendungen, die in einem Container laufen und zeichnet Zugriffe dieser auf, z.B. die verwendeten *System Calls*. Aus diesen Aufzeichnungen und einer Analyse der statischen Daten eines Images werden

Profile für *Seccomp* und *AppArmor* erstellt, die anschließend verwendet werden können. Die Generierung eines *AppArmor*-Profils ist aktuell in der Entwicklungsphase [92].

5.7.3 Bane

Auch das von Docker-Mitarbeiterin Jessie Frazelle entwickelte Werkzeug *Bane* hat zum Ziel, *AppArmor*-Profile automatisch zu generieren. Im Gegensatz zu *docker-slim* analysiert es nicht Images und Container, sondern übersetzt Anweisungen einer gut lesbaren Konfigurationsdatei in ein *AppArmor*-Profil. Der Vorteil der Konfigurationsdatei gegenüber einer Datei, die ein *AppArmor*-Profil enthält, ist, dass sich ausdrückstärkere, kategorisierbare Anweisungen formulieren lassen, sodass zum Erstellen der Datei keine speziellen *AppArmor*-Kenntnisse erforderlich sind [148].

Ein beispielhafter Ausschnitt einer Konfigurationsdatei im TOML-Format könnte folgendermaßen aussehen [188]:

```
...
[Filesystem]
WritablePaths = [
    "/var/run/nginx.pid"
]
AllowExec = [
    "/usr/sbin/nginx"
]
DenyExec = [
    "/bin/sh",
    "/usr/bin/top"
]
...
```

Bei der Übersetzung in ein valides *AppArmor*-Profil, entstehen daraus folgende Zeilen [187]:

```
...  
/var/run/nginx.pid w,  
/usr/sbin/nginx ix,  
deny /bin/sh mrwklx,  
deny /usr/bin/top mrwklx,  
...
```

Wie bei einem Vergleich der beiden Fragmente zu sehen ist, ist die Konfigurationsdatei strukturierter und durch Verwendung von aussagekräftigen Bezeichnern wie `Filesystem`, `WritablePaths`, `AllowExec` und `DenyExec` übersichtlicher als das daraus generierte *AppArmor*-Profil.

Nach Aussage von Jessie Frazelle in [148], [154] und [155] stellt *Bane* den Grundbaustein eines zukünftigen universalen, nativen Sicherheitmoduls mit Profilen für Capabilities, *AppArmor* und *Seccomp* dar. Dieses wird voraussichtlich in eine zukünftige Docker-Version einfließen.

Kapitel 6

Docker in Cloud-Infrastrukturen

Wie in der Abschnitt 2.1 des Grunlagenkapitels gezeigt, bietet der Einsatz von Virtualisierungslösungen, insbesondere Containern, in Rechenzentren einige Vorteile.

Je nach Anforderungen der Anwendung und der verfügbaren unternehmensinternen Ressourcen und externen Anbieter, lässt sich eine Lösung mithilfe Abb..... und Abb.... bilden. Unabhängig von der gewählten Virtualisierungstechnologie, lassen sich die Vorteile von Containern bezüglich Automatisierung, und immer nutzen. Auch wenn sich z.B. ein Unternehmen aus Sicherheitsgründen für eine Private Cloud (Abb.3.1) in Kombination mit einer hypervisorbasierten Virtualisierung (Abb.4.1) entscheidet, können die besagten Vorteile von Containern trotzdem innerhalb einer VM der Private Cloud genutzt werden, indem in die VMs z.B. Docker installiert wird.

Mit dem CLI-Werkzeug *Docker Machine* stellt *Docker* die Option zur Verfügung, von Linux-, Mac- und Windows-Clients aus lokale und entfernte Docker-Hostsysteme zu verwalten. Mit frei konfigurierbaren, sowie auf Cloud-Anbietern angepassten Treibern, eignet sich das Werkzeug auch zur Administration von Docker-Installationen in Private und Public Clouds [40][36][113].

Durch den hohen Bekanntheitsgrad und der Popularität von Docker in Entwickler- und Managementkreisen sind Anbieter von Public-Clouds gezwungen, eine Unterstützung von Docker in ihr eigenes Portfolio aufzunehmen. Tatsächlich reagieren die großen Anbieter wie *Amazon*, *Microsoft*, *IBM* und *Google* mit vielseitigen, häufig Docker-basierten Containerangeboten. Die etablierten Cloud-Ausprägungen SaaS, PaaS und IaaS werden durch explizite Containerangebote ergänzt, sodass zur Zeit der Begriff CaaS (Container as a Service) als weitere Form von Cloud-Angeboten Gestalt annimmt. Die Auswirkungen von Docker auf genannte Anbieter sind in Abschnitt 6.1 beschrieben.

Auch in Private Clouds ist Docker ein Thema. Die in Kapitel 4 und 5 aufgeführten Eigenschaften von Docker eignen sich für eine unternehmensinterne Anwendung der Technologie, wie in Kapitel 6.2 näher erläutert wird.

6.1 Public Cloud

Während der Begriff CaaS von Unternehmen hauptsächlich für Marketingzwecke genutzt wird, sind sich die großen Vertreter auf dem Virtualisierungsmarkt selbst nicht über die Bedeutung des Begriffs einig. *Google* z.B. versteht, nach Aussage von Brendan Burns, darunter eine Mischform aus PaaS und IaaS, die von ihrer eigenen Entwicklung *Kubernetes* ermöglicht wird [133, S.12]. *Docker* hingegen umschreibt bei der eigenen Definition von CaaS das Konzept von DevOps, das einen von Docker ermöglichten, flexiblen Workflow der Zukunft realisiert. „Der Weg zu CaaS ist mit Docker gepflastert“ ist eine Aussage eines Artikels des offiziellen Docker-Blogs [163].

So versuchen Unternehmen dem Begriff CaaS nach eigenen Vorstellungen Gestalt zu verleihen und ihn zum eigenen Vorteil zu nutzen.

Fest steht, dass Docker als disruptive Technologie von allen Anbieter von Public Clouds akzeptiert und unterstützt werden muss. Diese Tatsache beruht weniger auf der technischen Überlegenheit von Docker gegenüber anderen Containerlösungen oder der Hypervisor-basierten Virtualisierung, sondern auf einem einfachen, automatisierbaren Workflow, der im Modell von Con-

tinuous Integration und Continuous Delivery von Docker ermöglicht wird. Die Innovation, mit der sich IT-Unternehmen aktuell konfrontiert sehen, geschieht dadurch vorrangig auf Prozess- und Geschäftsebene.

Wie große Public Cloud-Anbieter Docker in ihr bestehendes Produktportfolio integrieren und welche Sicherheitsfeatures diese anbieten, ist am Beispiel von *Azure* von *Microsoft* und *SoftLayer* von *IBM* in den nächsten Abschnitten aufgezeigt. Beide Parteien werben mit einer engen *Docker*-Partnerschaft und guten Integrationsmöglichkeiten von Docker in die eigene Infrastruktur [87][88]. Für beide Produkte stellt *Docker* eigene Treiber zur Verfügung, die von der *Docker Machine* eingebunden werden können [37][38]. Die Treiber enthalten Daten wie z.B. eine Kundennummer, Anmeldeinformationen und andere plattformspezifische Einstellungen der Public Cloud.

6.1.1 Beispiel: Microsoft Azure

Die Public Cloud-Plattform *Azure* bietet eine Integration von Docker auf Basis von VM-Erweiterungen an. Eine Erweiterung mit dem Namen *Docker Extension* installiert bei der Erstellung einer neuen VM in dieser einen Docker-Daemon. Das Gastbetriebssystem in der VM wird dadurch zu einem Docker-Host. Angeblich unterstützt *Azure* aktuell nur die Linux-Distribution *Ubuntu* in der Version 14.04 LTS. Laut neueren Informationen aus dem Repository der Docker-Erweiterung, wird neben *Ubuntu* auch *CoreOS*, *CentOS* 7.1 und höher, sowie *RHEL* 7.1 und höher unterstützt [146].

Die Docker-Erweiterung lässt sich sowohl über eine Webanwendung als auch über ein Kommandozeilen-Tool installieren. Auch Zertifikate zur Sicherung der Kommunikation (siehe Kapitel 5.3) können bei der Einrichtung ausgewählt werden [191][192].

Unter *Azure* kann Docker ausschließlich innerhalb einer VM betrieben werden. Grund hierfür ist, dass *Microsoft* sein eigenes Betriebssystem *Windows Server* sowie seinen Hypervisor *Hyper-V* nutzt, um VMs zu realisieren. Eine direkte Installation von Docker auf diesem Betriebssystem

ist ausgeschlossen, da Docker aktuell nur in Kombination mit einem Linux-Betriebssystem läuft. Diese Einschränkung soll mit der nächsten Version des Serverbetriebssystems von *Microsoft*, *Windows Server 2016*, aufgehoben werden. *Windows Server 2016*, das aktuell als Technical Preview 4 vorliegt, verspricht Docker nativ, ohne die Hilfe eines Hypervisors, zu unterstützen. Der Betrieb von Containern auf der Plattform *Azure* wird bereits in den Ausprägungen *Windows Server Containers* und *Hyper-V Containers* diversifiziert. Ersteres sieht die zukünftige Möglichkeit vor, Docker direkt auf *Windows Server 2016* auszuführen. Letztere meint den Betrieb von Containern innerhalb einer VM, wie er aktuell von *Azure* angeboten wird [87][176].

Im Vergleich zur Konkurrenz werden *Azure* gute Integrationsmöglichkeiten von Public und Private Clouds auf Basis von *Windows Server* nachgesagt, was *Microsoft* zu einem attraktiven Anbieter von Hybrid Clouds macht [201].

Sicherheitsfeatures von *Azure* können wie Docker als Erweiterungen in VMs integriert werden. Diese beinhalten u.a. Datenverschlüsselung, Firewalls, Intrusion Detection und Anti-Malware Tools. Diese Module können unabhängig von Docker für jede VM aktiviert werden [190]. Da diese Module dadurch keinen Bezug zur Art der Virtualisierung haben und für jede VM unabhängig von Docker aktiviert werden können, sind diese Sicherheitsfeatures an dieser Stelle nur kurz erwähnt.

6.1.2 Beispiel: IBM SoftLayer

Zwischen *Docker* und *IBM* existiert eine Partnerschaft, die eine enge Integration von Docker in das Portfolio von *IBM Cloud* zur Folge hat. Während *IBM Cloud* überwiegend PaaS-Angebote, insbesondere *Bluemix*, umfasst, ist *SoftLayer* ein von *IBM* erworbenes Unternehmen, deren Geschäftsmodell auf IaaS beruht [137]. Nach der Definition von Brendan Burns, die CaaS als Mischform aus PaaS und IaaS beschreibt (siehe Kapitel 6.1), deckt *Bluemix* seit der Einführung von Docker-basierten *IBM Containers* im September 2015 auch CaaS-Angebote ab [181]. Das Portfolio von *IBM Cloud* wird wiederum auf der Infrastruktur von *SoftLayer* betrieben [135].

SoftLayer klassifiziert seine IaaS-Angebote in „Bare Metal“- und virtuelle Server. Erstere beziehen sich auf dedizierte physische Hosts, die Kunden in den Rechenzentren von *SoftLayer* mieten können. Auch wenn dieses Modell als Lösung unter dem Namen „Private Cloud“ verkauft wird [65], ist es - nach der gewählten Definition des Begriffs im Glossar - kein Angebot der Cloud, sondern klassisches Hosting von physischen Servern. Mit dem zweitgenannten Angebot sind virtuelle Instanzen gemeint, die unter Verwendung von vier verschiedenen Hypervisoren und neun unterschiedlichen Gastbetriebssystemen erzeugt werden können [67]. Durch die Natur von IaaS sowie den freien Konfigurationsmöglichkeiten, ist der Betrieb von Docker-Containern unter beiden Varianten realisierbar.

Wie unter *Azure* werden Sicherheitskomponenten zusätzlich angeboten. Auch *SoftLayer* bietet seinen Kunden die Wahl zwischen einigen, bereits in Kapitel 6.1.1 aufgelisteten Mechanismen unterschiedlicher Hersteller an [66].

6.2 Private Cloud

Private Clouds definieren sich dadurch, dass sie von einem Unternehmen in einem eigenen Rechenzentrum betrieben werden.

Durch die volle Kontrolle über die Hardware des eigenen Rechenzentrums, ist die Bereitstellung von Cloud-Diensten in beliebiger Granularität möglich. IaaS, PaaS und SaaS können einzeln oder in Kombination in einer Private Cloud betrieben werden. In einer solchen Cloud, wird die Multi-Tenant-Umgebung von Public Clouds zu einer Single-Tenant-Umgebung, da per Definition nur Cloud-Dienste des Betreibers selbst existieren. Diese wichtige Eigenschaft hat direkte Folgen für die Sicherheit: Mehrere Kunden müssen nicht mehr wie es in Public Clouds der Fall ist, voneinander isoliert werden. Eine Trennung innerhalb der Private Cloud gibt es nur noch nach Anwendungen, die nicht miteinander interferieren sollen. Während Sicherheitsmechanismen, um die Verfügbarkeit dieser Anwendungen sicherzustellen, weiterhin wichtig sind, kommt solchen, die die Vertraulichkeit und Integrität gewährleisten

sollen, weniger Bedeutung in Private Clouds als in Public Clouds zu. Diese Schlussfolgerung beruht auf der Annahme, dass Anwendungen der Private Cloud von Natur aus vertrauenswürdiger sind, da sie vollständig beobachtet und kontrolliert werden können, sowie oftmals von eigenen Unternehmensmitarbeitern entwickelt wurden. Diese Sicherheitsvorteile existieren in diesem Ausmaß nicht in Private Clouds, die in Rechenzentren Dritter betrieben werden. Diese Konstellation wird auch als Managed Private Cloud bezeichnet und widerspricht der in dieser Arbeit gewählten Definition von Private Clouds. Wie die Bezeichnung aussagt, werden Managed Private Clouds von externen Anbietern verwaltet. Jedoch bezieht der Kunde hierbei keine mit anderen Kunden geteilte Instanz, sondern erhält eine im Rechenzentrum physisch getrennte, dedizierte Infrastruktur [65].

Das potentiell geringere Sicherheitsrisiko in Private Clouds kann sich auch auf die Art der gewählten Virtualisierungsarchitektur auswirken. Der reine Betrieb von Containern auf einem Docker-Hostsystem ist ohne die Unterstützung von Hypervisor-Technologien, je nach Sicherheitsanforderungen, möglich.

Private Clouds werden u.a. als kommerziellen Produkte angeboten. Z.B. verkauft *Microsoft* die Lösung *Windows Azure Pack*, welche auf proprietärer Software von *Microsoft* basiert und in Rechenzentren von Kunden installiert wird. Im Open-Source-Bereich dominiert *OpenStack* als Implementierung für Private und Hybrid Clouds (siehe Kapitel 6.3), das im folgenden Kapitel vorgestellt wird.

6.2.1 Beispiel: OpenStack

OpenStack ist eine Open-Source-Software mit Apache-2-Lizenz, die es erlaubt, Cloud-Angebote nach dem IaaS-Modell zu realisieren. Grundlegende Komponenten, die in einem Rechenzentrum existieren, werden über einige Module angeboten. Z.B. wird Rechenkapazität mit Nova, Orchestrierung mit Heat und Speicherkapazität mit Swift und Cinder abgebildet. Die einzelnen Komponenten können über eine Vielzahl an APIs miteinander kommunizie-

ren und erfüllen, meist in Kombination, die jeweiligen Anforderungen an eine IaaS-Lösung.

Durch die Vielseitigkeit und Größe von *OpenStack*, wird der Technologie eine hohe Komplexität und eine lange Einarbeitungszeit nachgesagt. *SUSE* und andere Linux-Anbieter profitieren von dieser Eigenschaft, indem sie z.B. mit *SUSE OpenStack Cloud* eigene *OpenStack*-Lösungen anbieten, die Kunden nach eigenen Aussagen eine einfach realisierbare Private Cloud ermöglicht [108][168, S.2,4]. Das Startup *Platform 9* verspricht *OpenStack*-as-a-Service; ein Angebot, das die Komplexität von *OpenStack* für Kunden verbergen soll [117].

Trotz der Komplexität integrieren viele Anbieter auf dem Cloud-Markt vermehrt *OpenStack* in ihr Portfolio. Unterstützer von *OpenStack* sind u.a. *SUSE*, *Red Hat*, *IBM*, *Oracle* und *Intel* [171][201]. Auch die steigende Nachfrage nach Hybrid Clouds in der Industrie, begünstigt *OpenStack*, wie in Kapitel 6.3 gezeigt wird.

Die Entwicklung von *OpenStack* ist, mit der Verbesserung bestehender und Entwicklung neuer Modulfunktionen, sehr aktiv. Anfang 2015 wurde mit *Projekt Magnum* ein weiteres Modul angekündigt, das Docker und *Kubernetes* in *OpenStack* verfügbar macht [85]. Davor war ein Betrieb von Containern nur über den Treiber *Nova-Docker* für das Modul Nova möglich, dessen Funktionsumfang im Zusammenhang mit Containern sehr begrenzt ist [186][84]. Magnum nutzt die bestehenden Funktionen von Nova und Heat, die virtuelle Instanzen verwalten und fügt Schnittstellen hinzu, um mit Docker- und *Kubernetes*-Installationen innerhalb der virtuellen Instanzen zu kommunizieren [85].

Neben der neuen Integration von Magnum, unterstützt *OpenStack* auch viele Hypervisor-Technologien wie z.B. KVM, XEN, Hyper-V von *Microsoft* und vSphere von *VMware*.

Auch für *OpenStack* bietet Docker einen eigenen Treiber an [39].

6.3 Hybrid Cloud

Hybrid Clouds bezwecken eine Kombination aus Private und Public Cloud. Beide Arten von Clouds bieten Schnittstellen an, die eine Kommunikation der beiden Arten ermöglicht. Für Benutzer von Cloud-Diensten ist die technische und meist geographische Trennung der beiden Infrastrukturen unsichtbar, da diese im Hintergrund von einer Hybrid-Cloud-Lösung verwaltet wird.

Durch die Verknüpfung von eigenen und fremden Rechenzentren, ist die Kompatibilität zwischen beiden Infrastrukturen von großer Bedeutung. Ohne sich in die Abhängigkeit eines Hybrid-Cloud-Produkts mit kommerziellen Technologien zu begeben, ist *OpenStack* auch für dieses Szenario eine attraktive Alternative. Durch die zunehmende Akzeptanz von *OpenStack* und dessen Integrationsmöglichkeiten in kommerzielle Produkte, wird sowohl die bereits genannte, notwendige Kompatibilität gewährleistet, als auch ein sogenannter Vendor-Lock-In vermieden. Durch die Verwendung von *OpenStack* kann einfach zwischen Anbietern gewechselt werden. Außerdem können die Angebote von Hybrid-Cloud-Lösungen auf der Basis der zugrundeliegenden Open-Source-Technologie transparent miteinander verglichen werden.

Mit dem Freiheitsgrad einer von *OpenStack* realisierten IaaS, lassen sich eine Vielzahl von Betriebssystem und Virtualisierungstechnologien nutzen. Dadurch ist auch der Betrieb von Containern mit Docker und Kubernetes, insbesondere durch das Modul Magnum, jederzeit möglich.

Kapitel 7

Fazit

Wie die Vielzahl an vorgestellten Sicherheitsmechanismen aus Kapitel 4 zeigt, setzt Docker mit diesen nach den Prinzipien *Defense In Depth* und *Principle Of Least Privilege* eine komplexe, mehrschichtige Sicherheitsarchitektur auf Softwarebasis um. Die verschiedenen Mechanismen werden wahrscheinlich in einem zukünftigen Docker-Release im Rahmen eines benutzerfreundlichen Werkzeugs zusammengefasst. Dieses soll auf der Basis eines universellen Sicherheitsprofils, die einzelnen Sicherheitstechnologien entsprechend konfigurieren [154]. Auch das sicherheitsrelevante Plugin-Framework soll in Docker-Version 1.11 überarbeitet werden [52][?].

Diese, in Kapitel 3.6 als technische Kontrollen definierten Maßnahmen, werden durch weitere technische und administrative Konzepte aus Kapitel 5 ergänzt. Eine Geheimhaltung von technischen Sicherheitsmaßnahmen kann, wie in Kapitel 5.5 erörtert, nicht praktiziert werden.

Wie die Ergebnisse aus Kapitel 6 zeigen, bieten alle bekannten Anbieter von Public Clouds, sowie *OpenStack* als Lösungsansatz von Private Clouds dokumentierte Integrationsmöglichkeiten für Docker an. Spezielle Sicherheitsmerkmale in Cloud-Infrastrukturen existieren innerhalb eines Docker-Hostsystems nicht. Allgemeine, sicherheitsrelevante Merkmale innerhalb eines Docker-Systems wurden in Kapitel 4 und 5 behandelt. Vielmehr rückt in Cloud-Infrastrukturen

die Netzwerksicherheit sowie Verwaltungsaspekte in den Vordergrund, die im Fall einiger Public Cloud-Anbietern mit separaten Komponenten realisiert werden können.

Ein weiterer wichtiger Punkt in der Sicherheit von Cloud-Infrastrukturen ist die Art des Cloud-Angebots: Die am meisten verbreiteten Typen SaaS, PaaS und IaaS weisen in ihrer Natur unterschiedliche Sicherheitsmerkmale auf. Da man als Nutzer von IaaS einen hohen Freiheitsgrad bei der Architektur eigener Dienste erhält, sind auch Sicherheitseigenschaften der genutzten Infrastruktur eigenverantwortlich umzusetzen. Diese Verantwortung wird durch den engeren Rahmen von SaaS und PaaS generell auf den Betreiber der Infrastruktur übertragen.

Das Docker, das unter PaaS und IaaS verwendet werden kann, unterscheidet sich unter beiden Angebotstypen nicht. Damit sind auch die Sicherheitseigenschaften von Docker für PaaS und IaaS identisch. Spezielle Anforderungen an die Sicherheit für Anwendungen in können umgesetzt werden, indem die in Kapitel 4 und 5 vorgestellten Methoden und Mechanismen aktiviert werden oder weitere Sicherheitsinstanzen außerhalb des Docker-Ökosystems hinzugezogen werden. Letztere können z.B. Firewalls oder Authentifizierung über *Kerberos* sein.

Bereits im Docker-Projekt existiert eine firmenübergreifende, interdisziplinäre Zusammenarbeit von Entwicklern, um die Technologie zu erweitern und abzusichern. Durch die Kooperation von Organisationen und Unternehmen der letzten Monate im Rahmen der OCI, weitete sich die Zusammenarbeit aus, um ein standardisiertes und universales Containerformat zu erstellen und etablieren.

Docker ist dadurch gezwungen sein aktuelles Geschäftsmodell zu überdenken. Mit dem quelloffenen OCF wird ein Industriestandard geschaffen, mit dem sich Docker nur noch am Rande profilieren kann. Durch die Standardisierung von Containern ist zu erwarten, dass sich die ursprüngliche Innovationspotential von Containern, das allein *Docker* ausnutzen konnte, auf die Orchestrierung und Verwaltung von Containern verschiebt. *Docker* wird es

in diesem Bereich schwieriger haben, eine Marktdominanz zu erreichen, da die Konkurrenz zu *Docker Swarm* mit *Kubernetes* von *Google* und *Marathon* von *Mesosphere* bereits heute sehr beliebt ist.

Wie die jüngsten Aktivitäten von *Docker* zeigen, setzt das Unternehmen den öffentlichen Fokus seiner Arbeit zunehmend auf Unternehmenslösungen und die Bereitstellung von Images und Werkzeugen.

Die für die Sicherheit interesstanten Aktivitäten im Docker-Blog von Oktober 2015 bis März 2016 sowie Beiträge auf der *GitHub*-Plattform im gleichen Zeitraum bestätigen diesen Trend. Diese öffentlichen Aktivitäten sind in folgender Auflistung stichpunktartig genannt:

- Verbesserung der bestehenden Plattform:
 - Erweiterbarkeit von Docker durch Plugins: ein generisches Plugin-Framework wird für Docker-Version 1.11 erwartet [52][?].
 - Universelles, einfach zu bedienendes Werkzeug zur Erstellung von Sicherheitsprofilen für *SELinux*, *AppArmor* und *Seccomp* angekündigt [154].
- Werkzeuge zur Orchestrierung, Verwaltung und Skalierung von Containern. Akquirierung von *Conductant, Inc.*. [?].
- Akquirierung von *Unikernel Systems* und Entwicklung von sogenannten Unikernels. [?]

V.a. der Kauf von *Unikernel Systems* im Januar 2016 ist aus Sicht der Sicherheit sehr interessant. Die Entwickler von *Unikernel Systems* entwerfen Unikernels, die in Docker-Container langfristig eingebaut werden sollen. Unter der Verwendung von Unikernels wird das klassische Konzept eines geteilten Kernels mit einem Konzept ersetzt, das pro Container einen minimalen Unikernel vorsieht. Neben flexiblen Einsatzmöglichkeiten, wirkt sich Unikernels positiv auf die Sicherheit in Containern aus. Gelingt es *Docker* Unikernels in die eigene Technologie standardmäßig zu integrieren, wäre das ein Meilenstein für die Sicherheit von Containern.

Nicht nur Docker, sondern auch Linux verändert sich. Wie in Kapitel 4.1 und 4.2 erwähnt, sind Erweiterungen des Linux-Kernels, in Form von neuen Namespaces und einer Überarbeitung der Control Groups, absehbar. Durch die allgemeine Unzufriedenheit über die aktuelle Implementierung von LSMs, ist es möglich, dass diese Architektur zukünftig Änderungen unterzogen wird [?]. Durch die Mitarbeit vieler Experten, auch z.B. von *Red Hat*, sollten zukünftige Linux-Neuerungen schnell in Docker integrierbar sein.

Docker mag für viele eine zu junge und nicht ausreichend getestete Technologie sein, um in Produktion laufende Anwendungen zu betreiben. Es muss allerdings beachtet werden, dass sich bei der Konzeption von Infrastrukturen niemand zwischen Containern und der Virtualisierung mit Hypervisoren entscheiden muss. Vielmehr macht in einem sicherheitskritischen Umfeld eine Kombination beider Virtualisierungstechniken Sinn. Die Sicherheit eines Hypervisors und der von Containern ermöglichte, automatisierbare Workflow lassen sich in einer kombinierten Lösungen gewinnbringend einsetzen. Beide Technologien können gleichzeitig genutzt werden, um neben der Sicherheit auch logische Strukturen in der Infrastruktur abzubilden.

Glossary

3-Tier-Architektur Serverarchitektur, die logische Trennung eines Dienstes in drei Schichten vorsieht: Anwendungs-, Logik- und Datenschicht.

7

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

chroot Steht für **change root** und ist eine Funktion von UNIX-Systemen, um das Root-Verzeichnis des aktuellen Prozesses zu wechseln [18]. Grundlegendes Konzept für Linux-Namespaces. 11

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

- Private Cloud: Dienste werden in einem eigenen Rechenzentrum betrieben.
- Public Cloud: Dienste werden in Rechenzentren externer Anbieter betrieben. Virtuelle Instanzen werden mit anderen Kunden des Betreibers geteilt. Beispiele: *Amazon Web Services*, *Microsoft Azure*
- Hybrid Cloud: Mischform aus Private und Public Cloud. Nahtlose, für den Nutzer unsichtbare Integration der beiden Grundformen.

. 1

Continuous Delivery Eine Praxis in der Softwareentwicklung, bei der Software zu jeder Zeit in Produktionsumgebungen einsetzbar ist. Basiert auf dem Prinzip der Continuous Integration [?]. 75

Continuous Integration Eine Praxis in der Softwareentwicklung, bei der jeder Entwickler eines Teams seine Arbeit regelmäßig mit anderen Projektkomponenten integriert. Jede Integration wird mit Builds und Tests verifiziert [?]. 75

Copy-On-Write Verfahren, bei dem mehrere Prozesse auf die gleiche, geteilte Ressource zugreifen. Erst wenn ein Prozess die Ressource manipulieren will, erhält dieser eine eigene Kopie der Ressource. Somit ist die Integrität der Ressource für die anderen Prozesse weiterhin gewährleistet [26]. 19

Denial of Service Allgemeine Bezeichnung von Angriffen, die i.d.R. durch die Überlastung von Ressourcen das Sicherheitsziel der Verfügbarkeit verletzen. Beispiel: Fork-Bomb. 14, *siehe* Fork Bomb

Fork Bomb Angriff, der die Ressourcen eines Systems erschöpft, indem ein das schadhafte Programm rekursiv Kopien von sich selbst in einer Endlosschleife erzeugt. Durch den damit verbundenen Speicherbedarf, verletzt der Angriff das Sicherheitsziel der Verfügbarkeit. 44

Inter Process Communication Die Kommunikation zwischen Prozessen. Unterstützt durch eine Sammlung von Tools, die z.B. aus Semaphoren, Message Queues und geteilte Speichersegmente bestehen.. 38

Kernelobjekt Datenstrukturen im Kernel, die verschiedene Ressourcen (z.B. Datei, Verzeichnis, etc.) abbildet und von LSMs ausgewertet werden kann [83]. 45, 48

Multi-Tenant-Service Serveranwendungen bzw. -umgebung, die mehreren Nutzern (ggf. Kunden) gleichzeitig dient. 2, 41

Shell Benutzerschnittstelle, die interaktive und kommandobasierte Nutzung von Betriebssystemen und Anwendungen ermöglicht. Unter Linux meist als Bash verfügbar, eine Implementierung der Shell. 21

System Call Fundamentale Schnittstelle unter Linux, die Anwendungen den Zugriff auf Funktionen des Kernels ermöglicht [73]. 11, 48

Virtual Appliance VM, die Paket aus Anwendung, Bibliothek und Betriebssystem enthält. Kunden erwerben funktionierende VM, anstelle einer alleinigen Anwendung [195, S.672f.]. 7

weiche und harte Limits Numerische Angabe zur maximalen Nutzung einer Ressource. Eine Methode, um Verfügbarkeit zu ermöglichen. Das weiche Limit dient als Richtwert. Das harte Limit stellt den Maximalwert dar. Angestrebeter Zustand: weiches Limit \leq hartes Limit. 42

Zero-Day-Exploit Schwachstelle in einer Software, die Angreifern bekannt ist und von diesen ausgenutzt wird. Ersteller der Software ist die Schwachstelle (noch) nicht bekannt. Aus diesem Grund kann die Sicherheitslücke nicht behoben werden.. 44

Abkürzungsverzeichnis

ACL Access Control List. 44

API Application Programming Interface. 16, 19, 63

AppArmor Application Armor. 47, 49

CA Certificate Authority. 64

cgroups Control Groups. 42

CLI Command-Line Interface. 50

CPU Central Processing Unit. 1, 42

CVE Common Vulnerabilities and Exposures. 68

DAC Discretionary Access Control. 44

DoS Denial of Service. 41, *Glossary*: Denial of Service

GUI Graphical User Interface. 54

HDD Hard Disk Drive. 42

HTTP Hypertext Transfer Protocol. 16, 18

HTTPS Hypertext Transfer Protocol Secure. 18, 61

I/O Input and Output. 42

IP Internet Protocol. 38

IPC Inter Process Communication. 38, *Glossary: Inter Process Communication*

IT Informationstechnik. 3, 6

JSON JavaScript Object Notation. 20

LSM Linux Security Modules. 47

LTS Long Term Support. 77

LXC Linux Containers. 19

MAC Mandatory Access Control (Zugriffskontrolle). 30, 46, 47

MLS Multi-Level Security. 53, 57

NSA National Security Agency. 53

OCF Open Container Format. 19

OCP Open Container Project. 19

PID Process ID, Process Identifier. 34, 44

RBAC Role-Based Access Control. 65, 72

REST Representational State Transfer. 16, 63

RHEL Red Hat Enterprise Linux. 77

runC runContainer. 19

SELinux Security Encanced Linux. 47, 53

SSD Solid State Drive. 42

TLS Transport Layer Security. 64

TOML Tom's Obvious, Minimal Language. 73

TUF The Update Framework. 63

UI User Interface. 24

USB Universal Serial Bus. 42

UTS UNIX Time Sharing. 40

VM Virtual Machine. 9, 11, 13

Literaturverzeichnis

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

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

64e8fa9199fb345e017c8ef5299ca69cee01ab4c/CONTRIBUTING.md ,
aufgerufen am 28.02.2016.

- [18] Debian handbuch - befehl chroot2. über Website <http://manpages.debian.org/cgi-bin/man.cgi?apropos=0&format=html&query=chroot&sektion=2&locale=de> , aufgerufen am 17.03.2016.
- [19] Debian wiki - selinux setup. über Website <https://wiki.debian.org/SELinux/Setup> , aufgerufen am 05.02.2016.
- [20] Debian wiki - selinux setup. über Website <https://wiki.ubuntu.com/SELinux> , aufgerufen am 05.02.2016.
- [21] 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.
- [22] 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.
- [23] Docker and selinux. über Website <http://www.projectatomic.io/docs/docker-and-selinux/> , aufgerufen am 05.02.2016.
- [24] Docker cve database. über Website <https://www.docker.com/docker-cve-database> , aufgerufen am 28.02.2016.
- [25] Docker docs - registry. über Website <https://docs.docker.com/registry/> , aufgerufen am 18.01.2016.
- [26] Docker docs - understand images, containers, and storage drivers. über Website <https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/> , aufgerufen am 17.03.2016.
- [27] Docker docs - understanding the architecture. über Website <https://docs.docker.com/engine/introduction/understanding-docker/> , aufgerufen am 14.01.2016.

- [28] Docker documentation - protect the docker daemon socket. über Website <https://docs.docker.com/engine/security/https/> , aufgerufen am 24.02.2016.
- [29] Docker documentation - runtime metrics. über Website <https://docs.docker.com/engine/articles/runmetrics/> , aufgerufen am 27.01.2016.
- [30] Docker documentation - security. über Website <https://docs.docker.com/engine/security/security/> , aufgerufen am 24.02.2016.
- [31] Docker documentation für den befehl `docker images`. über Website <https://docs.docker.com/engine/reference/commandline/images/> , aufgerufen am 21.01.2016.
- [32] Docker documentation für den befehl `docker pull`. über Website <https://docs.docker.com/engine/reference/commandline/pull/> , aufgerufen am 21.01.2016.
- [33] Docker documentation für den befehl `docker run`. über Website <https://docs.docker.com/engine/reference/run/> , aufgerufen am 18.03.2016.
- [34] Docker does not run with unified cgroup hierarchy #16238. über Website <https://github.com/docker/docker/issues/16238> , aufgerufen am 10.03.2016.
- [35] Docker hub - explore. über Website <https://hub.docker.com/explore/> , aufgerufen am 15.01.2016.
- [36] Docker machine driver - generic. über Website <https://docs.docker.com/machine/drivers/generic/> , aufgerufen am 08.03.2016.
- [37] Docker machine driver - microsoft azure. über Website <https://docs.docker.com/machine/drivers/azure/> , aufgerufen am 08.03.2016.

- [38] Docker machine driver - microsoft azure. über Website <https://docs.docker.com/machine/drivers/soft-layer/> , aufgerufen am 08.03.2016.
- [39] Docker machine driver - openstack. über Website <https://docs.docker.com/machine/drivers/openstack/> , aufgerufen am 08.03.2016.
- [40] Docker machine overview. über Website <https://docs.docker.com/machine/overview/> , aufgerufen am 08.03.2016.
- [41] Docker registry storage driver. über Website <https://docs.docker.com/registry/storagedrivers/> , aufgerufen am 24.02.2016.
- [42] Docker security portal. über Website <https://www.docker.com/docker-security> , aufgerufen am 28.02.2016.
- [43] *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.
- [44] Fixing control groups. über Website <https://lwn.net/Articles/484251/> , aufgerufen am 27.01.2016.
- [45] FreeBSD - hierarchical resource limits. über Website https://wiki.freebsd.org/Hierarchical_Resource_Limits , aufgerufen am 27.01.2016.
- [46] 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.
- [47] Github repository - docker notary. über Website <https://github.com/docker/notary> , aufgerufen am 24.02.2016.
- [48] Github repository changelog von docker. über Website <https://github.com/docker/docker/blob/master/CHANGELOG.md> , aufgerufen am 05.02.2016.

- [49] Github repository der cgroups-implementierung von runc. über Website <https://github.com/opencontainers/runc/tree/master/libcontainer/cgroups/fs> , aufgerufen am 27.01.2016.
- [50] Github repository der docker engine. über Website <https://github.com/docker/docker> , aufgerufen am 11.01.2016.
- [51] Github repository glossar von docker. über Website <https://github.com/docker/distribution/blob/master/docs/glossary.md> , aufgerufen am 21.01.2016.
- [52] Github repository roadmap von docker. über Website <https://github.com/docker/docker/blob/master/ROADMAP.md> , aufgerufen am 15.03.2016.
- [53] Github repository von *runC*. über Website <https://github.com/opencontainers/runc> , aufgerufen am 21.01.2016.
- [54] 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.
- [55] Homepage des kvm hypervisors und virtualisierungslösung. über Website http://www.linux-kvm.org/page/Main_Page , aufgerufen am 18.01.2016.
- [56] Homepage des vmware esxi hypervisors. über Website <https://www.vmware.com/de/products/esxi-and-esx/overview> , aufgerufen am 18.01.2016.
- [57] Homepage des xen hypervisors. über Website <http://www.xenproject.org/> , aufgerufen am 18.01.2016.
- [58] Homepage *Solaris* betriebssystem. über Website <http://www.oracle.com/de/products/servers-storage/solaris/solaris11/overview/index.html> , aufgerufen am 18.01.2016.
- [59] Homepage grsecurity. über Website <https://grsecurity.net/> , aufgerufen am 25.02.2016.

- [60] Homepage kerberos. über Website <http://web.mit.edu/kerberos/> , aufgerufen am 26.02.2016.
- [61] Homepage kubernetes. über Website <http://kubernetes.io/> , aufgerufen am 20.03.2016.
- [62] Homepage the update framework. über Website <https://theupdateframework.github.io/> , aufgerufen am 24.02.2016.
- [63] Homepage twistlock. über Website <https://www.twistlock.com/> , aufgerufen am 26.02.2016.
- [64] Homepage von *runc*. über Website <https://runc.io/> , aufgerufen am 21.01.2016.
- [65] Ibm softlayer - private cloud solutions. über Website <http://www.softlayer.com/PRIVATE-CLOUDS> , aufgerufen am 08.03.2016.
- [66] Ibm softlayer - security software. über Website <http://www.softlayer.com/SECURITY-SOFTWARE> , aufgerufen am 08.03.2016.
- [67] Ibm softlayer - server software. über Website <http://www.softlayer.com/software> , aufgerufen am 08.03.2016.
- [68] 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.
- [69] Introducing runc: a lightweight universal container runtime. über Website <http://blog.docker.com/2015/06/runc/> , aufgerufen am 21.01.2016.
- [70] 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.

- [71] Linux programmer's manual - namespaces(7). über Website <http://man7.org/linux/man-pages/man7/namespaces.7.html> , aufgerufen am 28.01.2016.
- [72] Linux programmers manual - proc(5).
- [73] Linux programmers manual - system calls syscalls2. über Website <http://man7.org/linux/man-pages/man2/syscalls.2.html> , aufgerufen am 17.03.2016.
- [74] 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.
- [75] Low level interfaces to the apparmor kernel module - securityfs. über Website http://wiki.apparmor.net/index.php/Kernel_interfaces#securityfs_-_2Fsys.2Fkernel.2Fsecurity.2Fapparmor , aufgerufen am 18.02.2016.
- [76] Magic sysrq. über Website https://wiki.ubuntuusers.de/Magic_SysRQ/ , aufgerufen am 18.02.2016.
- [77] Manage data in containers. über Website <https://docs.docker.com/engine/userguide/containers/dockervolumes/> , aufgerufen am 10.03.2016.
- [78] Microsoft azure storage driver. über Website <https://docs.docker.com/registry/storage-drivers/azure/> , aufgerufen am 24.02.2016.
- [79] Offizielle dockerfile dokumentation. über Website <https://docs.docker.com/engine/reference/builder/#expose> , aufgerufen am 22.01.2016.
- [80] Offizieller twitter-account des docker-gründers, solomon hykes. über Website <https://twitter.com/solomonstre> , aufgerufen am 18.01.2016.
- [81] Offizielles repository der datenbank postgres. über Website https://hub.docker.com/_/postgres/ , aufgerufen am 02.03.2016.

- [82] Offizielles repository des webservers nginx. über Website https://hub.docker.com/_/nginx/ , aufgerufen am 11.01.2016.
- [83] 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.
- [84] Openstack - docker. über Website <https://wiki.openstack.org/wiki/Docker> , aufgerufen am 08.03.2016.
- [85] Openstack - magnum. über Website <https://wiki.openstack.org/wiki/Magnum> , aufgerufen am 08.03.2016.
- [86] Openstack swift storage driver. über Website <https://docs.docker.com/registry/storage-drivers/swift/> , aufgerufen am 24.02.2016.
- [87] Partner portal - docker and microsoft. über Website <https://www.docker.com/microsoft> , aufgerufen am 08.03.2016.
- [88] Partner portal - docker subscription with ibm. über Website <https://www.docker.com/IBM> , aufgerufen am 08.03.2016.
- [89] 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.
- [90] Phase 1: Initial seccomp support #17989. über Website <https://github.com/docker/docker/pull/17989> , aufgerufen am 05.02.2016.
- [91] Proposal: Support for user namespaces #7906. über Website <https://github.com/docker/docker/issues/7906> , aufgerufen am 28.01.2016.
- [92] Readme-datei von docker-slim. über Website <https://github.com/cloudimmunity/docker-slim/blob/bfe1202c60ffad5b1ac9abb46e0d557eadfe15e1/README.md> , aufgerufen am 20.03.2016.

- [93] Red hat enterprise linux reference guide - /proc/sysrq-trigger. über Website https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Reference_Guide/s2-proc-sysrq-trigger.html , aufgerufen am 18.02.2016.
- [94] Red hat enterprise linux documentation - chapter 7. setting shared memory. über Website https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Tuning_and_Optimizing_Red_Hat_Enterprise_Linux_for_Oracle_9i_and_10g_Databases/chap-Oracle_9i_and_10g_Tuning_Guide-Setting_Shared_Memory.html , aufgerufen am 18.02.2016.
- [95] Registry as a pull through cache. über Website <https://docs.docker.com/registry/mirror/> , aufgerufen am 23.02.2016.
- [96] Release notes von *FreeBSD V.4* und *Jails*. über Website <https://www.freebsd.org/releases/4.0R/notes.html> , aufgerufen am 19.01.2016.
- [97] Release notes von *Solaris 10*. über Website <https://docs.oracle.com/cd/E19253-01/pdf/817-0552.pdf> , aufgerufen am 19.01.2016.
- [98] Rhel dokumentation - chapter 3. selinux contexts. über Website https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/chap-Security-Enhanced_Linux-SELinux_Contexts.html , aufgerufen am 06.02.2016.
- [99] S3 storage driver. über Website <https://docs.docker.com/registry/storage-drivers/s3/> , aufgerufen am 24.02.2016.
- [100] Seccomp default profile. über Website <https://github.com/docker/docker/blob/master/profiles/seccomp/default.json> , aufgerufen am 18.02.2016.
- [101] Seccomp security profiles for docker. über Website <https://github.com/docker/docker/blob/master/docs/security/seccomp.md> , aufgerufen am 05.02.2016.

- [102] securityfs. über Website <https://lwn.net/Articles/153366/> , aufgerufen am 18.02.2016.
- [103] Selinux default policy profile. über Website <https://github.com/docker/docker/tree/master/contrib/docker-engine-selinux> , aufgerufen am 03.02.2016.
- [104] Selinux default policy profile - docker.te. über Website <https://github.com/docker/docker/blob/master/contrib/docker-engine-selinux/docker.te> , aufgerufen am 17.02.2016.
- [105] Slides of keynote at dockercon in san francisco - day 2. über Website <http://de.slideshare.net/Docker/dockercon-15-keynote-day-2/16> , aufgerufen am 11.01.2016.
- [106] Softlayer benchmark, data sheet. über Website https://voltdb.com/sites/default/files/voltdb_softlayer_benchmark_0.pdf , aufgerufen am 14.01.2016.
- [107] Standard-capabilities von docker. über Website https://github.com/docker/docker/blob/master/daemon/execdriver/native/template/default_template_linux.go , aufgerufen am 17.02.2016.
- [108] Suse pain-free private cloud. über Website <https://www.suse.com/promo/cloud/> , aufgerufen am 08.03.2016.
- [109] 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.
- [110] Understand docker container networks. über Website <https://docs.docker.com/engine/userguide/networking/dockernetworks/> , aufgerufen am 10.03.2016.
- [111] The unified control group hierarchy in 3.16. über Website <https://lwn.net/Articles/601840/> , aufgerufen am 27.01.2016.

- [112] Unified extensible firmware interface. über Website https://wiki.archlinux.org/index.php/Unified_Extensible_Firmware_Interface , aufgerufen am 18.02.2016.
- [113] Use docker machine to provision hosts on cloud providers. über Website <https://docs.docker.com/machine/get-started-cloud/> , aufgerufen am 08.03.2016.
- [114] User namespaces - phase 1 #15187. über Website <https://github.com/docker/docker/issues/15187> , aufgerufen am 28.01.2016.
- [115] 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.
- [116] Voltdb homepage. über Website <https://voltdb.com/> , aufgerufen am 18.01.2016.
- [117] Why platform9 managed private cloud? über Website <http://platform9.com/why-platform9/> , aufgerufen am 08.03.2016.
- [118] Überblick hyper-v hypervisor von microsoft. über Website <https://technet.microsoft.com/library/hh831531.aspx> , aufgerufen am 18.01.2016.
- [119] Übersicht zu *Solaris Zones*. über Website https://docs.oracle.com/cd/E24841_01/html/E24034/gavhc.html , aufgerufen am 18.01.2016.
- [120] 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.

- [121] Google code archive - go issue #8447. über Website <https://code.google.com/archive/p/go/issues/8447> , aufgerufen am 28.01.2016, 2014.
- [122] Image manifest version 2, schema 1. über Website <https://github.com/docker/distribution/blob/4c850e7165f4d8d7ea3df2454a2e2a890829d5ce/docs/spec/manifest-v2-1.md> , aufgerufen am 28.02.2016, December 2015.
- [123] Docker 1.10: New compose file, improved security, networking and much more! über Website <https://blog.docker.com/2016/02/docker-1-10/> , aufgerufen am 05.02.2016, Februar 2016.
- [124] Extended documentation: Create an authorization plugin. über Website <https://github.com/docker/docker/blob/e310d070f498a2ac494c6d3fde0ec5d6e4479e14/docs/extend/authorization.md> , aufgerufen am 26.02.2016, January 2016.
- [125] 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'.
- [126] 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'.
- [127] 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.
- [128] Understanding engine plugins. über Website <https://github.com/docker/docker/blob/cc085be7cc19d2d1aed39c243b6990a7d04ee639/docs/extend/plugins.md> , aufgerufen am 26.02.2016, 2016.
- [129] Charles Anderson. Docker. *IEEE Software*, 2015.

- [130] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, September 2014.
- [131] 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.
- [132] Thanh Bui. Analysis of docker security. Technical report, Aalto University School of Science, January 2015.
- [133] Brendan Burns. Containers, clusters and kubernetes. über Website <http://www.slideshare.net/Docker/docker-48351569> , aufgerufen am 08.03.2016, November 2014.
- [134] David Calavera. Comment on won't-fix status of *AuthN* in issue: Add authentication to the docker daemon #18514. über Website <https://github.com/docker/docker/pull/18514#issuecomment-187942565> , aufgerufen am 26.02.2016, February 2016.
- [135] Scott Cook and Heather Fitzsimmons. Ibm and docker announce strategic partnership to deliver enterprise applications in the cloud and on prem. über Website <https://www-03.ibm.com/press/us/en/pressrelease/45597.wss> , aufgerufen am 08.03.2016, December 2014.
- [136] Jonathan Corbert. Seccomp: replacing security modules? über Website <https://lwn.net/Articles/443099/> , aufgerufen am 01.02.2016, Mai 2011.
- [137] Kimi Cousins. Ibm delivers docker based container services. über Website <https://developer.ibm.com/bluemix/2015/06/22/ibm-containers-on-bluemix/> , aufgerufen am 08.03.2016, June 2015.
- [138] Michael Crosby. Creating containers - part 1. über Website <http://crosbymichael.com/creating-containers-part-1.html> , aufgerufen am 01.02.2016, November 2014.

- [139] Stephen Day. A new model for image distribution. über Website <http://www.slideshare.net/Docker/docker-48351569> , aufgerufen am 28.02.2016, May 2015.
- [140] Stephen Day and Arnaud Porterie. Github repository - image uuid implementation. über Website <https://github.com/docker/distribution/blob/2c9ab4f441b3e5218e10ef99923d7f86e8494f2c/uuid/uuid.go> , aufgerufen am 28.02.2016, July 2015.
- [141] 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.
- [142] Rajdeep Duo, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. *IEEE International Conference on Cloud Engineering*, 2014.
- [143] Jake Edge. Device namespaces. über Website <https://lwn.net/Articles/564854/> , aufgerufen am 17.03.2016, August 2013.
- [144] Jake Edge. Control group namespace. über Website <https://lwn.net/Articles/621006/> , aufgerufen am 17.03.2016, November 2014.
- [145] 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.
- [146] Ahmet Alp Balkan et al. Readme.md of github repository - azure virtual machine extension for docker. über Website <https://github.com/Azure/azure-docker-extension/blob/032b0086397155d22f27639aca3b6016b0dfbef4/README.md> , aufgerufen am 08.03.2016, March 2016.
- [147] Brendan Burns et al. Security in kubernetes. über Website <https://github.com/kubernetes/kubernetes/blob/>

- 2b2f2857771c748f0f0e261f3b8e2ad1627325ce/docs/design/security.md , aufgerufen am 01.03.2016, December 2015.
- [148] Jessie Frazelle et al. Readme.md of github repository - bane. über Website <https://github.com/jfrazelle/bane/blob/bbc594c9dce4351b1175737cb3b4d9ee65803648/README.md> , aufgerufen am 01.03.2016, March 2016.
 - [149] Pilwon Huh et al. Offizielles dockerfile für nginx webserver. über Website <https://github.com/dockerfile/nginx/blob/d5d34a100d01b2d27d0fa1662854fbe3b05325aa/Dockerfile> , aufgerufen am 27.01.2016.
 - [150] Pravin Goyal et al. Cis docker 1.6 benchmark. Technical report, Center for Internet Security, April 2015.
 - [151] Stefan Fischer et al, editor. *Security - IT-Sicherheit unter Linux von A bis Z*. Linux Magazine, 2008.
 - [152] Tianon Gravi et al. Postgres docker image - run script. über Website <https://github.com/docker-library/postgres/blob/443c7947d548b1c607e06f7a75ca475de7ff3284/9.5/docker-entrypoint.sh> , aufgerufen am 02.03.2016, January 2016.
 - [153] 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.
 - [154] Jessie Frazelle. Docker security profiles (seccomp, apparmor, etc) #17142. über Website <https://github.com/docker/docker/issues/17142#issuecomment-148974642> , aufgerufen am 15.03.2016, November 2015.
 - [155] 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.

- [156] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Katalog B 3.304 Virtualisierung*, 2011.
- [157] Serge Hallyn. Container security - past, present and future. über Website <http://events.linuxfoundation.org/sites/events/files/slides/security.pdf> , aufgerufen am 10.03.2016, August 2015.
- [158] Daniel J. Walsh (Red Hat). 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.
- [159] Daniel J. Walsh (Red Hat). Are docker containers really secure? über Website <https://opensource.com/business/14/7/docker-security-selinux> , aufgerufen am 10.03.2016, Juli 2014.
- [160] Daniel J. Walsh (Red Hat). Docker security in the future. über Website <https://opensource.com/business/15/3/docker-security-future> , aufgerufen am 05.02.2016, March 2015.
- [161] Adam Herzog. Extending docker with plugins. über Website <https://blog.docker.com/2015/06/extending-docker-with-plugins/> , aufgerufen am 26.02.2016, June 2015.
- [162] Trevor Jay. Before you initiate a docker pull: über Website <https://securityblog.redhat.com/2014/12/18/before-you-initiate-a-docker-pull/> , aufgerufen am 24.02.2016, December 2014.
- [163] Betty Junod. Containers as a service (caas) as your new platform for application development and operations. über Website <https://blog.docker.com/2016/02/containers-as-a-service-caas/> , aufgerufen am 08.03.2016, February 2016.
- [164] Michael Kerrisk. *The Linux Programming Interface - A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

- [165] Oren Laadan. Rfc: Device namespaces. über Website <https://lwn.net/Articles/564977/> , aufgerufen am 17.03.2016, August 2013.
- [166] Liron Levin. Docker authz plugins: Twistlock's contribution to the docker community. über Website <https://www.twistlock.com/2016/02/18/docker-authz-plugins-twistlocks-contribution-to-the-docker-community/> , aufgerufen am 26.02.2016, February 2016.
- [167] Liron Levin. Json-struktur eines *AuthZ*-regelwerks. über Website <https://github.com/twistlock/authz/blob/2b67bbbfb9bbc1579ade722dc84c1e3c3440fbb/authz/policy.json> , aufgerufen am 26.02.2016, January 2016.
- [168] Martin Gerhard Loschwitz. Openstack - viele brauchen es, keiner versteht es - wir erklären es. über Website <http://www.golem.de/news/openstack-viele-brauchen-es-keiner-versteht-es-wir-erklaeren-es-1503-112814.html> , aufgerufen am 08.03.2016, March 2015.
- [169] Peter Mandl. *Grundkurs Betriebssysteme*. Springer, 4 edition, 2014.
- [170] 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.
- [171] Rainald Menge-Sonnentag. Openstack liberty: Cloud-framework macht das dutzend voll. über Website <http://www.heise.de/developer/meldung/OpenStack-Liberty-Cloud-Framework-macht-das-Dutzend-voll-2849083.html> , aufgerufen am 08.03.2016, October 2015.
- [172] Mike Meyers and Shon Harris. *CISSP - Certified Information Systems Security Professional*. Springer, 3 edition, 2009.
- [173] Diogo Mónica. Introducing docker content trust. über Website <https://blog.docker.com/2015/08/content-trust-docker-1-8/> , aufgerufen am 24.02.2016, August 2015.

- [174] Diogo Mónica and Thomas Sjögren et al. Docker bench tests. über Website <https://github.com/docker/docker-bench-security/tree/master/tests> , aufgerufen am 02.03.2016.
- [175] Diogo Mónica and Thomas Sjögren et al. Github repository - docker bench. über Website <https://github.com/docker/docker-bench-security> , aufgerufen am 02.03.2016, May 2015.
- [176] Neil Peterson. Windows containers. über Website https://msdn.microsoft.com/en-us/virtualization/windowscontainers/about/about_overview , aufgerufen am 08.03.2016, February 2016.
- [177] 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.
- [178] Pethuru Raj, Jeeva S. Chelladurai, and Vinod Singh. *Learning Docker*. Packt Publishing, Juni 2015.
- [179] 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.
- [180] Mahmud Ridwan. Separation anxiety: A tutorial for isolating your system with linux namespaces. über Website <https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces> , aufgerufen am 19.03.2016.
- [181] Chris Rosen. Ibm containers launch in london. über Website <https://developer.ibm.com/bluemix/2015/09/30/ibm-containers-launch-london/> , aufgerufen am 08.03.2016, September 2015.

- [182] Jonathan Rudenberg. Docker image insecurity. über Website <https://titanous.com/posts/docker-insecurity> , aufgerufen am 24.02.2016, December 2014.
- [183] Jonathan Rudenberg. Tarsum insecurity #9719. über Website <https://github.com/docker/docker/issues/9719> , aufgerufen am 24.02.2016, December 2014.
- [184] Aleksa Sarai. Add pids cgroup support to docker #18697. über Website <https://github.com/docker/docker/pull/18697> , aufgerufen am 17.03.2016, Dezember 2015.
- [185] Karen Scarfone, Wayne Jansen, and Miles Tracy. *Guide to General Server Security*. National Institute of Standards and Technology, special publication 800-123 edition, July 2008.
- [186] Udo Seidel. Openstack goes container. über Website <http://www.heise.de/newsticker/meldung/OpenStack-goes-Container-2660052.html> , aufgerufen am 08.03.2016, May 2015.
- [187] Thomas Sjögren. Bane - apparmor sample file. über Website <https://github.com/jfrazelle/bane/blob/0ceb725f531cfb95a82a3586f6318343f3f8d17c/docker-nginx-sample> , aufgerufen am 01.03.2016, November 2015.
- [188] Thomas Sjögren. Bane - toml configuration file. über Website <https://github.com/jfrazelle/bane/blob/0ceb725f531cfb95a82a3586f6318343f3f8d17c/sample.toml> , aufgerufen am 01.03.2016, November 2015.
- [189] Ralf Spenneberg. *SELinux & AppArmor*. Addison-Wesley, 1 edition, 2008.
- [190] Ralph Squillace. About virtual machine extensions and features. über Website <https://azure.microsoft.com/en-us/documentation/>

- articles/virtual-machines-extensions-features/ , aufgerufen am 08.03.2016, August 2015.
- [191] Ralph Squillace. Die docker-erweiterung für virtuelle linux-computer auf azure. über Website <https://azure.microsoft.com/de-de/documentation/articles/virtual-machines-docker-vm-extension/> , aufgerufen am 08.03.2016, October 2015.
- [192] Ralph Squillace. Using the docker vm extension from the azure command-line interface (azure cli). über Website <https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-docker-with-xplat-cli/> , aufgerufen am 08.03.2016, January 2016.
- [193] Dima Stopel. Adding kerberos support to docker #13697. über Website <https://github.com/docker/docker/issues/13697#issue-84531039> , aufgerufen am 26.02.2016, June 2015.
- [194] Dima Stopel. User access control in docker daemon #14674. über Website <https://github.com/docker/docker/issues/14674#issue-95409105> , aufgerufen am 26.02.2016, July 2015.
- [195] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 3 edition, 2009.
- [196] James Turnbull. *The Docker Book*. 1.2.0 edition, September 2014.
- [197] Daniel J. Walsh and Matthew Heon. Docker access control. über Website <https://github.com/rhatdan/docker-rbac> , aufgerufen am 26.02.2016.
- [198] 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.

- [199] 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.
- [200] 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.
- [201] Serdar Yegulalp. Ibm embraces docker, openstack in bluemix hybrid cloud plans. über Website <http://www.infoworld.com/article/2887579/hybrid-cloud/ibm-embraces-docker-openstack-in-bluemix-hybrid-cloud-plans.html> , aufgerufen am 08.03.2016, February 2015.