

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

17. Januar 2016

Erstbetreuer:

Prof. Dr. Joachim Charzinski
Hochschule der Medien

Zweitbetreuer:

Patrick Fröger
ITI/GN, Daimler AG

Eidesstattliche Erklärung

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

Unterschrift

Datum

Inhaltsverzeichnis

1	Überblick der Arbeit / Struktur der Arbeit	1
2	Einführung	2
2.1	Virtualisierung	3
2.1.1	Hypervisor-basierte Virtualisierung	4
2.1.2	Container-basierte Virtualisierung	4
2.1.3	Einordnung Docker	7
2.2	Sicherheitsziele in der IT	8
2.2.1	Vertraulichkeit	8
2.2.2	Integrität	8
2.2.3	Verfügbarkeit	8
2.2.4	Authentizität	8
2.2.5	Authorisierung	8
2.2.6	Privatheit, Anonymität	8
2.2.7	Verbindlichkeit	8
2.3	Einführung in Docker	8
2.3.1	Container	10
2.3.2	Images	11
2.3.3	Registries	12
2.3.4	Dockerfile	13
2.3.5	Docker Architektur	13
2.3.6	Containerformat libcontainer	14
3	Ziel der Arbeit/Forschungsfrage	15

4	Security aus Linux Kernel-Features	16
4.1	Isolierung durch namespaces	17
4.1.1	Prozessisolierung (process namespace)	17
4.1.2	Dateisystemisolierung (filesystem namespace)	17
4.1.3	Geräteisolierung (device namespace)	17
4.1.4	IPC-Isolierung (ipc namespace)	17
4.1.5	UTS-Isolierung (uts namespace)	17
4.1.6	Netzwerkisolierung (network namespace)	17
4.1.7	Userisolierung (user namespace)	17
4.2	Ressourcenverwaltung / Limitierung von Ressourcen durch control groups	17
4.3	Einschränkungen von Zugriffsrechten	17
4.3.1	capabilities	17
4.3.1.1	Beispiele, /proc-Verzeichnis, (Un-)Mounten des Host-Filesystems	17
4.3.2	Linux Security Module (LSM) und Mandatory Access Control (MAC)	17
4.3.2.1	SELinux	17
4.3.2.2	AppArmor	17
4.3.2.3	Seccomp	17
4.4	Docker im Vergleich zu anderen Containerlösungen	17
5	Security im Docker-Ökosystem	18
5.1	Docker Images und Registries	19
5.1.1	neues Signierungs-Feature	19
5.2	Docker Daemon	19
5.2.1	REST-API	19
5.2.2	Support von Zertifikaten	19
5.3	Containerprozesse	19
5.4	Docker Cache	19
5.5	privileged Container	19
5.6	Networking	19
5.6.1	bridge Netzwerk	19

5.6.2	overlay Netzwerk	19
5.6.3	DNS	19
5.6.4	Portmapping	19
5.7	Daten-Container	19
5.8	Docker mit VMs	19
5.9	Sicherheitskontrollen für Docker	19
5.10	Tools rund um Docker	19
5.10.1	Docker-Erweiterungen	19
5.10.1.1	Docker Swarm	19
5.10.1.2	Docker Compose	19
5.10.1.3	Nautilus Project	19
5.10.2	Third-Party Tools	19
5.10.3	Vagrant	19
5.10.4	Kubernetes	19
6	Docker in Unternehmen/Cloud-Infrastrukturen	21
7	Fazit/Ausblick	22

Abbildungsverzeichnis

1	Die Client-Server-Architektur von Docker [12, S.10].	14
---	--	----

Tabellenverzeichnis

Kapitel 1

Überblick der Arbeit / Struktur der Arbeit

Kapitel 2

Einführung

Virtualisierte Komponenten nutzen im Vergleich zu nativ (physisch) eingesetzten Komponenten eine zusätzliche Softwareschicht, die den virtualisierten Komponenten, üblicherweise als virtuelle Maschinen (VMs) bezeichnet, mehrere Abstraktionen anbietet, um Funktionen des Gastsystems zu nutzen [13, S.2].

Es existieren heutzutage mehrere Virtualisierungstechniken, wovon die Hypervisor-gestützten Methoden mit den Vertretern Xen, VMware und KVM die meistverbreitesten sind [13, S.2]. Die zwei prominentesten Virtualisierungstechniken, Hypervisor-basierte und Container-basierte Virtualisierung, werden in dieser Arbeit gegenübergestellt (INTERNE-REFERENZ hier?).

Mit dem Release von Docker im Jahr 2013 erlebte die containerbasierte Virtualisierung einen Aufschwung, obwohl sie schon einige Jahre zuvor in der Form von XXXX und XXXX existierte (QUELLEN, mehrere da mehrere Bsp). Wie Docker den bis 2013 vorherrschenden Ruf von Containersystemen nachhaltig verändern konnte, nämlich dass Container noch nicht ausgereift seien [13, S.8], ist im Kapitel „Einführung in Docker,“ beschreiben.

Virtualisierungstechnologien sind in den letzten Jahren sehr beliebt geworden. Einige Softwarelösungen wie Xen, VMware und KVM (QUELLEN) sowie Hardwaresupport von handelsüblichen Prozessoren (QUELLEN, siehe

containerVirtPerformance) haben sich auf dem Markt etabliert. Die Vorteile von Virtualisierung sind Hardwareunabhängigkeit, Verfügbarkeit, Isolierung und Sicherheit, welche die Erfolgsgrundlage der Virtualisierung in Cloud-Infrastrukturen bilden [13, S.1]. Vor allem in Rechenzentren bieten sich Virtualisierungen an, um die Serverressourcen effizienter zu nutzen [9, S.1].

Letztendlich haben es Virtualisierungen ermöglicht, Serverressourcen in der Form von Clouds wie z.B. den *Amazon Web Services* auf Basis eines Subskriptionsmodells nutzen zu können [1][9, S.1].

Trotz ein paar intrinsischen Schwächen von Containerlösungen, werden Container bereits in einer Vielzahl von Szenarien eingesetzt [12, S.6].

Container sind in Infrastrukturen, in denen es auf Skalierbarkeit ankommt, trotz Sicherheitsbedenken beliebt. Vor allem "Multi-TenantSServices werden gerne mit Docker eingesetzt.

2.1 Virtualisierung

Bei der Virtualisierung werden ein oder mehrere virtuelle IT-Systeme auf einem physischen Computer betrieben. Mehrere solcher Computer können eine virtuelle Infrastruktur bilden, in der physische und virtuelle Maschinen gemeinsam verwaltet werden können.

Der Einsatz von Virtualisierung bietet vielfältige Vorteile für IT-Unternehmen. Sie können Kosten für Hardwarebeschaffung, Strom und Klimatisierung einsparen, wenn die Computerressourcen effizienter genutzt werden. Durch die damit verbundene Zentralisierung und Konsolidierung können auch in der Administration Ausgaben reduziert werden [11, S.1].

In diesem Kapitel wird nur die für diese Arbeit relevante Techniken der systembasierten Virtualisierung beschrieben, also solche, in denen Betriebssysteme ablaufen. Die Anwendungs-, Storage- oder Netzwerkvirtualisierung wird nicht behandelt.

2.1.1 Hypervisor-basierte Virtualisierung

Im Kontext von einer Hypervisor-basierten Virtualisierung, wird die virtuelle Umgebung eine virtuelle Maschine (VM) genannt. Die VMs enthalten jeweils eine Umgebung, die Abstraktionen eines sogenannten Hypervisors nutzt, um Hardwareressourcen des Hosts zu verwenden. Der Hypervisor, auch seltener *Virtual Machine Monitor* (VMM) genannt, ist Software, die zwischen einem Host und einem Gast (der VM) vermittelt [12, S.6][13, S.2][9, S.2].

Durch diese Technik läuft in jeder VM ein eigenes Betriebssystem, das von solchen anderer VMs komplett isoliert läuft. Durch die Abstraktion des zwischenliegenden Hypervisors ist es möglich, mehrere unterschiedliche Gastbetriebssysteme auf einem physikalischen Host auszuführen [13, S.2].

Der größte Kritikpunkt dieser Art von Virtualisierung ist deren hoher Bedarf an Hostressourcen, da diese für jede gestartete VM komplett virtualisiert werden müssen, sodass innerhalb der VM ein Gast-OS ausgeführt werden kann [7, S.1][8, S.3].

Hypervisor-technologien werden unter sich in solche von Typ 1 und Typ 2 unterschieden. Typ 1 Hypervisor operieren direkt auf der Hardware des Hosts, während Typ 2 auf einem Host-OS agiert, welches selbst direkt Hardware nutzt. Durch die Trennung von Hypervisor und Host-OS in der Architektur des Typs 2, ist dieser aus Sicht der Performance dem Typ 1 unterlegen [9, S.2].

Bekannte Vertreter von Hypervisoren sind die kommerziellen *ESXI* der Firma VMware und *Hyper-V* von Microsoft, sowie die Open-Source Hypervisor *Xen* und *KVM* [8, S.1].

2.1.2 Container-basierte Virtualisierung

(Benötigt keine Emulations- oder Hypervisorschicht [12, S.7].)

Container-basierte Virtualisierung ist eine leichtgewichtige Alternative zu Hypervisor-basierten Virtualisierungen [13, S.2], die den Hostkernel nutzt,

um virtuelle Umgebungen zu schaffen. Die virtuellen Umgebungen werden als Container bezeichnet [9, S.1].

Container sind durch den Unix-Befehl *chroot* inspiziert, der schon seit 1979 im Linux-Kernel integriert ist. In *FreeBSD* wurde eine erweiterte Variante von *chroot* verwendet, um *Jails* (FreeBSD-spezifischer Begriff) umzusetzen. In Solaris wurde dieser Mechanismus in Form von *Zones* (Solaris-spezifischer Begriff) weiter verbessert und es etablierte sich der Name *Container*, als weitere proprietäre Lösungen von HP und IBM zur selben Zeit auf dem Markt erschienen [8, S.2].

Einen Hypervisor wird in diesem Ansatz nicht eingesetzt. Vielmehr wird das native System und dessen Ressourcen partitioniert, sodass mehrere isolierte User-Space Instanzen betrieben werden können, die Container genannt werden [13, S.2] .

Während ein Hypervisor für jede VM das komplette Gast-OS abstrahiert, werden für Container direkt Funktionen des Hosts zur Verfügung gestellt. Deswegen werden Containerlösungen auch als Virtualisierungen auf Betriebssystemebene (des Hosts) bezeichnet [12, S.6][13, S.2]. Aus technischer Sicht ist der Hypervisor ein Stück Software, das eine Abstraktion der Hardware bereitstellt, während Container direkt via *System Calls* mit der Hostmaschine kommunizieren. Das hat zur Folge, dass alle Container direkt mit einem Host kommunizieren und sich damit den Kernel dessen teilen [13, S.2][8, S.3].

Sowohl Hypervisor-gestützte VMs als auch Container erwecken aus Sicht des Gasts den Eindruck, dass ein alleinstehendes Betriebssystem ausgeführt wird (QUELLE: “OpenVZ,” 2012. [Online]. Available: <http://www.openvz.org>). Um diese Illusion zu schaffen, wird jedoch wie beschrieben jeweils ein anderer Ansatz eingesetzt.

Das Containern zugrunde liegende Feature der Isolation wird in der Regel mit *namespaces*, einem Feature des Kernels, realisiert. Die Verteilung und das Management der Hostressourcen wird mit *control groups* umgesetzt (QUELLE?). Beide Techniken werden den Kapiteln „Namespaces“ und „Control Groups“ genauer betrachtet.

Containerlösungen umfassen die Technologien *OpenVZ*, *Solaris Zones*, sowie Linux-Container wie *Linux VServer*, *Linux Container (LXC)* [12, S.7][13, S.1] und *Docker*, welches im Fokus dieser Arbeit steht.

Moderne Container können als vollwertige Systeme betrachtet werden, nicht mehr als ursprünglich vorgesehen, reine Ausführungsumgebungen [12, S.7].

In Container-basierten Systemen hingegen, laufen die Container im „User Space“ direkt auf dem Kernel des Host-OS und nutzen dessen *System Call-Interface* [12, S.6+7]. Dadurch kommt es im Vergleich zu Hypervisor-Virtualisierungen zu einer fast nativen Performance [13, S.1], da der Virtualisierungs-Overhead des Hypervisors wegfällt. Unter dem Gesichtspunkt der Rechenleistung beispielsweise, kommt es bei Containerlösungen im Durchschnitt zu einem Overhead von ca. 4%, wenn diese mit der nativen Leistung einer festen Hardwarekonfiguration verglichen wird [13, S.4][10, S.5]. In traditionellen Virtualisierungen beansprucht der Hypervisor allein etwa 10-20% der Hostkapazität [7, S.2][10, S.5]. In der Praxis machen sich diese beiden Verhältnisse an einer hohen Dichte an Containern auf einem Container-basiertem Host und dadurch einer indirekt besseren Ressourcenausnutzung bemerkbar [12, S.7+8].

Ein Benchmarktest, der den Durchsatz (Operationen pro Sekunde) eines *VoltDB*-Setups von Hypervisor-basierte Cloudlösungen mit Container-basierten Cloudlösungen verglich, kam zu dem Ergebnis, dass die Containerlösung unter genanntem Gesichtspunkt eine fünffache Leistung aufwies [6, S.2+3].

Auch im Lifecycle von virtuellen Instanzen bieten Container Vorteile: Während in traditionellen VMs das komplette Gast-OS neu gestartet werden muss, um Änderungen zu übernehmen, entspricht ein Containerneustart nur einem Neustart eines Prozesses auf Host [8, S.2].

Container-Lösungen erlauben es, mehrere voneinander isolierte „User Space“-Instanzen parallel auf einem einzigen physischen Host zu betreiben [12, S.6]. Dadurch, dass ein Hypervisor in einer solchen Konfiguration nicht existiert und die Container direkt Hostkernel-Features nutzen, gibt es einen entscheidenden Nachteil für Containerlösungen - und damit auch Docker - gegenüber Hypervisor-basierter Virtualisierung: Das Container-Betriebssystem muss wie

das Host-Betriebssystem Linux-basiert sein. In einem Host auf dem Ubuntu Server installiert ist, können nur weitere Linux-Distributionen als Container laufen. Ein Microsoft Windows kann also nicht als Container auf genannten Host gestartet werden, da die Kernel miteinander nicht kompatibel sind [12, S.6]. Diese Inflexibilität im Spektrum der einsetzbaren Betriebssysteme liegt den Containerlösungen zugrunde.

Außerdem werden Container als weniger sicher im Vergleich zur Hypervisor-gestützten Virtualisierung gesehen [12, S.6].

Hingegen muss in containerbasierten Systemen nicht das gesamte Betriebssystem virtualisiert werden, da von den Containern direkt auf den Host-Kernel zugegriffen wird. Zum Einen schrumpft dadurch die Angriffsfläche des Hosts [12, S.6], da, wie später noch zu sehen ist, die Zugriffsrechte der Container auf den Host sehr feingranular festgelegt werden müssen. Zum Anderen entsteht durch diese Tatsache ein Risiko im Design, weil Host-Features ohne Hypervisor direkt genutzt werden.

2.1.3 Einordnung Docker

Docker gehört zu den Technologien der Container-basierten Virtualisierung.

Docker nutzt moderne Linux-Kernel-features, wie z.B. *control groups* und *namespaces*, um ein Ressourcenmanagement zwischen Containern und eine effektive Isolierung der Container vom Hostsystem zu realisieren [12, S.7].

Docker ist wie in KAPITEL-ZUVOR angedeutet, nicht die erste container-basierte Virtualisierungslösung. Einige ältere Containersysteme, wie z.B. *Solaris Zones*, gibt es schon viel länger als Docker, aber wurden allerdings nie von der Industrie als Lösung akzeptiert. Worauf beruht also der Erfolg von Docker in den letzten Jahren? Dieser Frage werde ich im folgenden Kapitel nachgehen.

2.2 Sicherheitsziele in der IT

2.2.1 Vertraulichkeit

2.2.2 Integrität

2.2.3 Verfügbarkeit

2.2.4 Authentizität

2.2.5 Authorisierung

Ist das eigenes Sicherheitsziel? Quellen widersprechen sich.

2.2.6 Privatheit, Anonymität

2.2.7 Verbindlichkeit

2.3 Einführung in Docker

Docker ist eine unter der Apache 2.0 Lizenz veröffentlichte Open-Source Engine, die den Einsatz von Anwendungen in Containern automatisiert. Es ist überwiegend in der Programmiersprache *Golang* von dem Unternehmen *Docker, Inc.* (vormals *dotCloud Inc.*), implementiert [3][12, S.7].

Docker erweitert *LXC* um eine Schnittstelle auf Kernel- und Applikationslevel [8, S.2].

Der große Vorteil von Docker gegenüber älteren Containerlösungen ist das Level an Abstraktion und die Bedienungsfreundlichkeit, die Nutzern ermöglicht wird. Während sich Lösungen vor Docker auf dem Markt durch deren schwierige Installation und Management sowie schwachen Automatisierungsfunktio-

nen nicht etablieren konnten, adressiert Docker genau diese Schwachpunkte [12, S.7] und bietet neben Containern viele Tools und einen Workflow für Entwickler, die beide die Arbeit mit Containern erleichtern soll [7, S.1].

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

Ein bekanntes Problem bei der Softwareentwicklung ist, dass Code in der Umgebung eines Entwicklers fehlerfrei ausgeführt wird, jedoch in Produktionsumgebungen Fehler verursacht. In der Regel fallen beide Umgebungen in unterschiedliche personelle Zuständigkeitsbereiche, was vereinfacht eine Übergabe von Entwicklungs- nach Produktionsumgebung mit sich zieht. Diesem Umstand wurde mit der Einführung von *DevOps*-Teams entgegengewirkt. Diese Teams sind sowohl für die Entwicklung (*Dev* = Development) eines Produkts als auch den Betrieb (*Ops* = Operations) dessen verantwortlich. Der Overhead einer Übergabe fällt dadurch weg. (DAZU QUELLEN FINDBAR?)

Einen anderen Ansatz dieses Problem zu lösen, liefern Container: Das Kernproblem im genannten Szenario sind die Entwicklungs- und Produktionsumgebung, zwischen denen Code ausgetauscht wird, da diese nicht identisch sind. Mithilfe von Containern können in der ansonsten gleichen Konstellation nun ganze Container, die den Code beinhalten, zwischen den Umgebungen ausgetauscht werden. Der große Vorteil der Container ist, dass die Ausführungsumgebung in diesem bereits enthalten ist, also mit sehr hoher Wahrscheinlichkeit in einer Entwicklerinfrastruktur als auch auf ei-

nem Produktionsinfrastruktur startfähig ist (Anstelle von „infrastruktur“ kann auch von „Umgebung“ gesprochen werden)(BEGRIFFLICHKEIT ERKLÄREN: es sind alle Umgebungen“ (Entwicklerumgebung, Produktionsumgebung, Containerumgebung) – KLARER FORMULIEREN).

Eine weitere wichtige Eigenschaft von Docker ist Konsistenz: Die Umgebungen, in denen Softwareentwickler Code schreiben, sind identisch mit den Umgebungen, die später auf Servern laufen.

Die Wahrscheinlichkeit, dass ein Fehler erst im Betrieb auftritt, nicht aber in der Entwicklung, wird dadurch sehr klein gehalten [12, S.8].

Quellcode kann inklusive virtualisierter Ausführungsumgebung flexibel von einem Laptop auf einen Testserver und später auf einen physischen oder virtualisierten Produktionsserver oder Cloud-Infrastruktur, wie z.B. Microsoft Azure, geschoben werden. Dieser kurzlebige Zyklus zwischen Entwicklung, Testen und Deployment erlaubt einen effizienten Workflow [12, S.8+12]. Da Quellcode das wertvollste Asset der meisten IT-Firmen ist und dieser erst dann Wert hat, wenn er bei einem Kunden ausgeführt wird, macht den beschriebenen Workflow zu einem wichtigen Entscheidungsgrund bei der Wahl der Entwicklerumgebung [7, S.1].

Die Eigenheiten von Docker sowie die gängige Begrifflichkeiten im Docker-Ökosystem, werden in den folgenden Unterkapiteln genauer beleuchtet.

2.3.1 Container

Der Begriff „Container“ ist bisher schon oft gefallen, deswegen will ich auf ihn zuerst eingehen.

Docker-Container beinhalten eine idealerweise minimale Laufzeitumgebung, in der eine oder mehrere Anwendungen laufen.

In Bezug zu anderen Docker-Begriffen, enthält ein Container ein Software-Image und erlaubt eine Reihe von Operationen, die auf ihn angewandt werden können. Darunter fallen z.B. das Erstellen, Starten, Stoppen, Neustarten

und Beenden eines Containers. Welchen Inhalt einen Container hat, also ob ein Container auf einem Datenbank- oder Webserver-Image beruht, ist dafür unerheblich [12, S.12][8, S.2].

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

2.3.2 Images

Ein Image besteht aus ein oder mehreren Schichten (Layers), wobei eine Schicht auch ein Image darstellen kann.

Images liegen Containern als statische Files zugrunde. Container werden auf der Basis von Images gestartet. Images sind durch ein *Union*-Dateisystem in Schichten gegliedert, die überlagert ein Image ergeben, das als Container gestartet werden kann [12, S.11].

Union-Dateisysteme haben gemeinsam, dass sie alle auf dem *Copy-on-write*-Modell basieren [12, S.8]. Konkrete Vertreter sind *AuFS*, *Btrfs* und *Device Mapper* [7, S.3].

Die Schichten eines Images umfassen in der Regel jeweils eine minimale Ausführungsumgebung mit Bibliotheken, Binaries und Hilfspaketen sowie den Quellcode der Anwendung, die im Container ausgeführt werden soll. Die Schichtenstruktur erlaubt es, Images modularisiert aufzubauen, sodass sich Änderungen eines Images nur auf eine Schicht auswirkt. Soll z.B. in ein bestehendes Image der Webserver *Nginx* integriert werden, kann dieser mit dem Kommando `sudo apt-get install nginx` installiert werden, was eine neue Schicht im Image erzeugt. Mit mehreren ähnlichen Images ist gewährleistet, dass nur die konkreten Unterschiede zwischen diesen als eigene Schichten hinterlegt sind. Eine gemeinsame Codebasis, die von mehreren Images genutzt wird, liegt in wenigen Schichten, die sich die Images teilen [7, S.3].

Images werden Schritt für Schritt erstellt, z.B. mit den folgenden Aktionen [12, S.11]:

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

Images sind einfach portierbar und können geteilt, gespeichert und aktualisiert werden [12, S.11].

Auf der Basis von existierenden Images können durch das Hinzufügen neuer Schichten durch oben beschriebene Aktionen, neue Images erstellt werden (EIGENES STATEMENT).

Über die Kommandozeile kann z.B. das Image eines *Nginx*-Webserver von der öffentlichen Docker-Registry mit dem Befehl `docker pull nginx` auf die lokale Maschine gespeichert werden.

2.3.3 Registries

Eine Registry ist ein gehosteter Service, der Images in Repositories anbietet.

Docker stellt eine Vielzahl an Images öffentlich in einer eigenen zentralen Registry, dem Docker Hub, zur Verfügung [12, S.11][9, S.3]. Für dieses System kann jeder Nutzer einen Account anlegen und eigenständig Images in öffentliche und private Repositories hochladen. Das Docker-Hub bietet bereits mehr als 150.000 Repositories, die etwa 240.000 Nutzer zusammenstellten und hochluden, zur freien Verwendung an (Stand Juni 2015) [5, S.16]. Die Einträge im Hub können von Nutzern bewertet werden. Außerdem wird angezeigt, wie oft ein Image bereits über das Hub bezogen wurde.

Ein Repository besteht aus mindestens einem Image. Um Images in einem Repository voneinander zu unterscheiden, werden Images Tags zugewiesen,

um beispielweise mehrere Versionen eines Images in einem Repository zu kennzeichnen. Die Images werden nach dem Schema `<repository>:<tag>` identifiziert. So gibt es z.B. im offiziellen Repository des Webserver *Nginx* Images mit den Tags `latest`, `1`, `1.9` und `1.9.9` [4]. Wenn bei dem Download kein Tag angegeben ist, wie in Kapitel wird automatisch das aktuellste Image `latest` bezogen, wie es im letzten Kapitel „Images“ praktiziert wurde.

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

2.3.4 Dockerfile

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

Eine Anweisung kann z.B. sein, ein Tool zu installieren oder zu starten, eine Umgebungsvariable festlegen oder einen Port öffnen.

2.3.5 Docker Architektur

Docker selbst ist nach einem Client-Server-Modell aufgebaut: Ein Docker-Client kommuniziert mit einem Docker-Daemon, also ein Prozess der den Server abbildet [2]. Beide Teile können auf einer Maschine oder einzeln auf unterschiedlichen Hosts laufen. Die Kommunikation zwischen Client und Daemon geschieht über eine RESTful API. Wie Abb.1 zeigt, ist es dadurch auch möglich Befehle eines oder mehrerer entfernter Clients über ein Netzwerk an den Daemon zu senden [9].

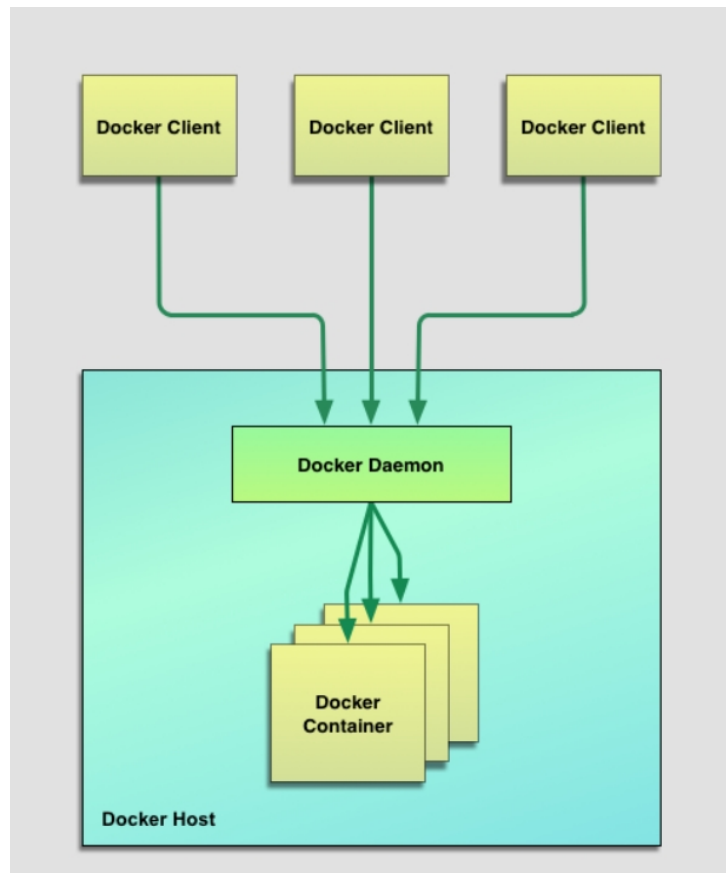


Abbildung 1: Die Client-Server-Architektur von Docker [12, S.10].

2.3.6 Containerformat libcontainer

`libcontainer` ist ein natives Linux Containerformat, das von dem Docker-team entwickelt wurde und seit Version 0.9 das Format *Linux Containers* (*LXC*) ablöst [9].

Kapitel 3

Ziel der
Arbeit/Forschungsfrage

Kapitel 4

Security aus Linux Kernel-Features

4.1 Isolierung durch namespaces

4.1.1 Prozessisolierung (process namespace)

4.1.2 Dateisystemisolierung (filesystem namespace)

4.1.3 Geräteisolierung (device namespace)

4.1.4 IPC-Isolierung (ipc namespace)

4.1.5 UTS-Isolierung (uts namespace)

4.1.6 Netzwerkisolierung (network namespace)

4.1.7 Userisolierung (user namespace)

4.2 Ressourcenverwaltung / Limitierung von Ressourcen durch `cgroups`

4.3 Einschränkungen von Zugriffsrechten

4.3.1 capabilities

Kapitel 5

Security im Docker-Ökosystem

5.1 Docker Images und Registries

5.1.1 neues Signierungs-Feature

5.2 Docker Daemon

5.2.1 REST-API

5.2.2 Support von Zertifikaten

5.3 Containerprozesse

5.4 Docker Cache

5.5 privileged Container

5.6 Networking 19

5.6.1 bridge Netzwerk

5.6.2 overlay Netzwerk

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

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

Kapitel 6

Docker in Unternehmen/Cloud- Infrastrukturen

Kapitel 7

Fazit/Ausblick

Spekulation in der Industrie ist, dass sich Organisationen und Unternehmen zusammenschließen und sich auf eine neue, universale Lösung einigen, die die heutigen Fähigkeiten der sich ergänzenden Technologien Docker und Kubernetes, abdeckt [8, S.4].

Glossary

Cloud blablabla. 3

Literaturverzeichnis

- [1] Amazon web services. über Website <https://aws.amazon.com/de/> , aufgerufen am 14.01.2016.
- [2] Docker docs - understanding the architecture. über Website <https://docs.docker.com/engine/introduction/understanding-docker/> , aufgerufen am 14.01.2016.
- [3] Github repository der docker engine. über Website <https://github.com/docker/docker> , aufgerufen am 11.01.2016.
- [4] Offizielles repository des webservers nginx. über Website https://hub.docker.com/_/nginx/ , aufgerufen am 11.01.2016.
- [5] Slides of keynote at dockercon in san francisco - day 2. über Website de.slideshare.net/Docker/dockercon-15-keynote-day-2/16 , aufgerufen am 11.01.2016.
- [6] Softlayer benchmark, data sheet. über Website https://voltdb.com/sites/default/files/voltdb_softlayer_benchmark_0.pdf , aufgerufen am 14.01.2016.
- [7] Charles Anderson. Docker. *IEEE Software*, 2015.
- [8] David Bernstein. Containers and cloud: From lxc to docker to kubnetes. *IEEE Cloud Computing*, September 2014.
- [9] Thanh Bui. Analysis of docker security. Technical report, Aalto University School of Science, January 2015.

- [10] 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.
- [11] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Katalog B 3.304 Virtualisierung*, 2011.
- [12] James Turnbull. *The Docker Book*. 1.2.0 edition, September 2014.
- [13] 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.