Umsetzung von Qualifizierung und Sicherheitsmechanismen in Docker-Container Images

Sebastian Schmailzl

***Zusammenfassung*—Dieses Dokument gibt eine Einleitung in das System von Docker und erläutert daraufhin Maßnahmen, wie Sicherheitsmechanismen eingebaut werden können. Zudem wird darauf eingegangen, wie eine Qualifizierung von Docker Images umgesetzt werden kann.**

***Index Terms*—Docker, Container, Sicherheitsmaßnahmen, Qualifizierung**

I. Einleitung

Docker ist ein System zur Isolierung und einfachen Bereitstellung von Anwendungen. Genauer spricht man hierbei von sogenannten Docker Containern. Ein Container bildet die Basis einer Anwendung. In ihm werden alle benötigten Pakete installiert und entsprechende Konfigurationen gesetzt. Vorteil hierbei ist es, dass diese Container auf jeder beliebigen Umgebung ausgeführt werden können. Docker hat das Deployen von Software, also dessen Verteilung komplett revolutioniert. Zu Beginn der Softwareentwicklung, musste sich jeder Entwickler seine Entwicklungsumgebung selbst komplett einrichten. Konkret heißt das, dass er die richtige Version seines Betriebssystems benötigte, alle Pakete mit der richtigen Version installieren und gegebenenfalls aufwändige Konfigurationen erzeugen musste. Dieser mühselige Prozess wurde im Laufe der Zeit bereits mithilfe der virtuellen Maschinen eliminiert. Der Vorteil hierbei war, das einmalig zu Beginn der Softwareentwicklung eine virtuelle Maschine für das Projekt erzeugt wurde, die mit dem benötigten Betriebssystem, den Paketen sowie der Konfiguration ausgestattet wurde. Doch auch diese Lösung war nicht optimal. Die besagte virtuelle Maschine konnte schnell auf eine Größe von mehreren Gigabytes wachsen, da sie nicht nur ein komplettes Betriebssystem enthielt, sondern auch alle Pakete, auch solche, die für den eigentlichen Betrieb der Software nicht benötigt werden. Ein zusätzliches Problem dieser Vorgehensweise war das Verhalten der virtuellen Maschine selbst. Basierend auf der Hardware des Host Computers konnte die Startdauer sowie Responsivität der virtuellen Maschine stark variieren. Zudem konnte es oft der Fall sein, das eine komplette virtuelle Maschine für eine einfache, ressourcenarme Applikation zu überteuert war. Das heißt, dass diese Anwendung kein eigenes Betriebssystem benötigt, sondern es ausreichend wäre in einem abgekapseltem Bereich des Host Systems ausgeführt zu werden. Für ein genau solches Szenario wurde Docker entwickelt. In Abbildung 1 wird der Unterschied zwischen virtuellen Maschinen und Containern graphisch näher dargestellt.[9]

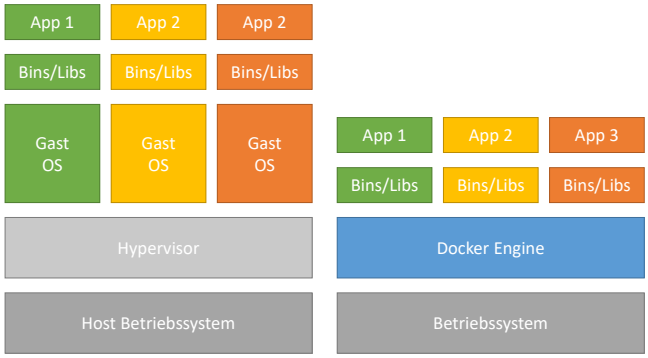

Abbildung 1. Virtuelle Maschine vs. Container

II. Funktionsweise von Docker

In diesem Abschnitt wird die Funktionsweise von Docker Containern und Images näher erläutert, da dies der Grundbaustein zum Verständnis dieser Technologie ist.
Wie bereits erwähnt, ist ein Container in Docker das Bauteil, welches die Anwendung, sowie alle Abhängigkeiten und Konfigurationen zusammenfasst. Die sogenannte Containerisierung geht damit mit einer Art der leichtgewichtigen Virtualisierung einher. Ein Docker Container ist dabei eine Instanz eines Docker Images. Vergleichbar ist dies mit dem Konzept der objektorientierten Programmierung. Hierbei ist eine Klasse die Blaupause und das Objekt eine Instanz. Ebenso ist das Docker Image die Blaupause, welche vorschreibt, wie ein Docker Container aufgebaut werden muss. Dies führt auch dazu, das aus einem einzelnen Docker Image mehrere Container gestartet werden können. Da viele Docker Images an mehreren Stellen wiederverwendet werden können, gibt es dafür die Docker Repositories. Darin können die Images versioniert abgelegt werden. Die Docker, Inc. betreibt zum Beispiel ihr eigenes Repository namens Docker Hub. Um letztendlich die Applikation lauffähig zu machen, sind dann meist nur noch wenige Handgriffe, wie zum Beispiel die Netzwerkkonfiguration nötig. Ein weiterer Vorteil dieser Technologie ist die Skalierbarkeit. Läuft eine Anwendung in einem Docker Container, so können bei Bedarf auf einfachste Weise neue Container generiert werden, die nach dem Stopp vollständig vom Host System verschwinden, ohne weiter Ressourcen zu verbrauchen. Für diese sogenannte Container-Orchestrierung können speziell entwickelte Programme, wie

Kubernetes von Google, genutzt werden. Jedoch bringt wie jede Technologie auch Docker Nachteile mit sich. Der wohl größte Nachteil ist, dass man bei der Virtualisierung mittels Containern eine reduziertere Abschottung als bei vollumfänglichen virtuellen Maschinen oder gar dedizierten Servern hat. Doch auch dieser Nachteil kann sich insgeheim als Vorteil entpuppen. So sind in einem Container nur Pakete installiert, die zur Ausführung der Anwendung benötigt werden und nicht auch noch jene zum Unterhalt des Betriebssystems. Durch diese Minimal-Anzahl an Paketen verringern sich auch die potentiellen Schwachstellen, die ein Angreifer ausnützen könnte, um das System zu kompromittieren. Fälschlicherweise wird oft angenommen, dass Docker Container aufgrund dieser Tatsache sicher seien. Doch auch benötigte Software kann oft Schwachstellen beinhalten. Wie genau Docker Container qualifiziert und Sicherheitsmechanismen eingesetzt werden können wird in den nächsten Kapiteln näher erläutert.[12]

III. Docker Pipeline

Die Docker Pipeline startet zumeist mit dem Erstellen eines eigenen Images. Grundsätzlich werden Docker Images, wie in Abbildung 2 dargestellt, schichtenweise aufgebaut. Dieses Verfahren sorgt dafür, dass die einzelnen Images flexibel und beliebig zusammengestellt werden können. Ein simpler Webserver startet hierbei zum Beispiel mit einem Debian-Basis-Image und erhält in zweiter Schicht den Apache Webserver. Dieser kann bei Bedarf auch einfach durch den nginx Webserver ersetzt werden. Ist das Image kreiert, so wird zur Laufzeit vom Container eine weitere beschreibbare Schicht hinzugefügt, was zur Folge hat, das die darunterliegenden Schichten nicht manipuliert werden und dasselbe Image in mehreren Containern mit jeweils anderer beschreibbarer Schicht ausgeführt werden kann. Um den Bau, also das Zusammenstellen und Konfigurieren der Images zu erleichtern, wurde das sogenannte Dockerfile eingeführt. Das Dockerfile selbst stellt eine Bauanleitung dar, die beschreibt, wie das Image aufgebaut werden muss.[12]

A. Dockerfile

Folgendes Beispiel demonstriert den Aufbau eines Python Webserves mittels eines Dockerfiles. Genauer gesagt werden hier zwei separate Dockerfiles und damit zwei Images genutzt. Der Grund hierbei ist zum einen den schichtenweisen Aufbau der Images zu demonstrieren, zum anderen jedoch auch um bei eventuellen Änderungen am Webserver nicht jedes mal alle Python-Abhängigkeiten neu installieren zu müssen. Im ersten Dockerfile (Listing 1) wird mittels des Befehls *FROM* angegeben, mit welchem Basis Image gestartet wird. Dieser Befehl muss zwingend an erster Stelle im Dockerfile stehen. Die zweite Instruction *MAINTAINER* gibt den Ersteller bzw. Maintainer des Images an. Mittels des Befehls *RUN* wird ein Shell Befehl initiiert. In diesem Fall wird die Installation von Python gestartet. Damit ist die Bauanleitung für das erste Image abgeschlossen.

Listing 1. Beispiel: Ubuntu Image mit Python Dependencies

```
FROM ubuntu:14.04
MAINTAINER Sebastian Schmailzl <
    sebastian.schmailzl@t-online.
    de>
RUN apt-get install -y python
```

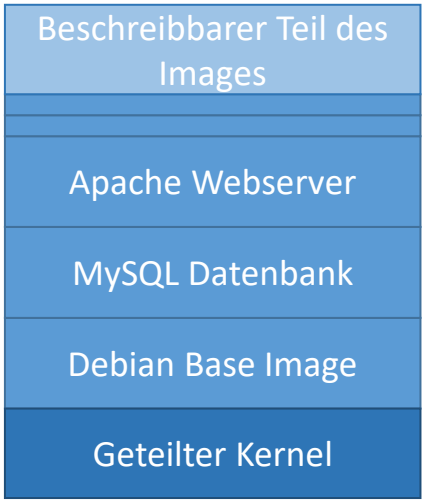

Abbildung 2. Schichtenweiser Aufbau der Docker Images

Das zweite Image (Listing 2) wird auf dem zuvor erstellten aufgebaut. Auch hier steht der Befehl *FROM* wieder an erster Stelle und spezifiziert diesmal das vorherige Image als Grundlage. Auch hier wird wieder ein Maintainer angeben. An dieser Stelle neu ist der Befehl *ADD*. Dieser sorgt dafür, dass aus dem aktuellen Dateisystem die entsprechenden Dateien in das Image eingefügt werden. Da Container nur auf Netzwerkanfragen, die vom Host kommen antworten, dies aber für einen Webserver kontraproduktiv ist, wird mittels *EXPOSE* der Port 8080 für externe Verbindungen freigegeben. Da Aufgrund der Abkapselung der Container vom eigentlichen Hostsystem keine Einsicht in das Dateisystem des Containers während der Laufzeit besteht, wird mit dem Befehl *VOLUME* explizit eine Art Mount Point für den Zugriff vom Host freigegeben. Beim Start des Containers muss dann spezifiziert werden, wohin das freigegebene Verzeichnis im Host Dateisystem gemountet werden soll.

Listing 2. Beispiel: Ubuntu Image mit Python Dependencies

```
FROM sebastianschmailzl/
    docker_demo_python
MAINTAINER Sebastian Schmailzl <
    sebastian.schmailzl@t-online.
    de>
ADD webserver.py /opt/webserver/
    webserver.py
ADD run.sh /opt/webserver/run.sh
EXPOSE 8080
```

```
VOLUME ["/logs"]
```

Damit ist auch der Bauplan für das zweite Image abgeschlossen und die Images können erzeugt werden.[12]

B. docker build

Wurde das Image bzw. die Images mittels der Dockerfiles spezifiziert, müssen sie nun noch erzeugt werden. Mittels des Kommandozeilenbefehls *docker build* können nun die Images aus den jeweiligen Dockerfiles erzeugt werden. Zum Erzeugen des Images aus Listing 1 wird folgender Befehl verwendet: `docker build --force-rm -t sebastianschmailzl/docker_demo_python python`. Mit diesem Befehl wird das Image im Unterverzeichnis *python* erstellt. Der Parameter *–force-rm* sorgt dafür, dass alle Zwischenimages gelöscht werden. Dies wird auch bei eventuell auftretenden Fehlern durchgeführt. Mit dem Parameter *-t* kann ähnlich wie bei Git Repositories ein Tag hinzugefügt werden. Daraufhin kann das zweite Image mit einem leicht abgewandelten Befehl `docker build --force-rm -t sebastianschmailzl/docker_demo_webserver webserver` gebaut werden. Der Befehl *docker images* kann dazu genutzt werden, um alle im lokalen Repository vorhandenen Images aufzulisten und zu kontrollieren. Die Ausgabe dieses Befehls sollte nach erfolgreichem Ausführen der vorherigen Befehle die zwei erstellten Images auflisten.[12]

C. docker run

Mittels des Befehls `docker run -d --cidfile=webserver.cid --name=webserver -v 'pwd'/logs:/logs sebastianschmailzl/docker_demo_webserver:latest /opt/webserver/run.sh` kann nun aus dem Image ein neuer lauffähiger Container instanziiert und ausgeführt werden. Der Parameter *-d* gibt an, dass der Container abgekapselt starten soll. *–cidfile* spezifiziert eine eindeutige Container-ID an. Mit *–name* kann dem Container ein eindeutiger Name zugewiesen werden. Der Parameter *-v* mountet das zuvor im Dockerfile unter *VOLUME* freigegebene Verzeichnis im log Unterverzeichnis des aktuellen Ordners. Im Anschluss daran wird noch spezifiziert welches Image gestartet und welcher Shell-Befehl ausgeführt werden soll. Dabei ist zu beachten, dass aufgrund der Abkapselung der Container vom eigentlichen Hostsystem nur ein einziger Prozess gestartet werden kann. Mittels des Befehls *docker ps* kann das Erzeugen des Containers überprüft werden. Hier sollte nun ein laufender Container mit dem Namen *webserver* in der Ausgabe aufgelistet sein.[12]

D. Weiter nützliche Docker Befehle

Darüberhinaus existieren noch Befehle wie *docker inspect*. Damit können Informationen über laufende Container abgefragt werden. Die Kommandos *docker pause* und *docker unpause* sorgen dafür, dass die Ausführung des Containers zeitweise pausiert und dann auch wieder begonnen werden kann. *docker stop* und *docker kill* beenden die Containerprozesse. Vergleichbar mit einer gestoppten virtuellen Maschine existieren die Container jedoch weiter und können mit *docker rm* gelöscht werden. Ein gestoppter Container kann mit *docker start* wieder hochgefahren werden.[12]

IV. Sicherheit in der Container Virtualisierung

Da im vorangegangenen Kapitel die Build Pipeline zum Erstellen neuer Docker Images und dem Erzeugen entsprechender Container näher erläutert wurde, kann nun damit begonnen werden die Sicherheit in der Pipeline und den Images näher zu beleuchten. Der Grund warum Docker in den letzten Jahren so an Beliebtheit zunahm, ist die Isolierung der einzelnen Container untereinander. Fälschlicherweise wird angenommen, dass die Container-Technologie in den Aspekten der Sicherheit auf dem höchsten Niveau ist. Jedoch müssen einige Vorkehrungen getroffen werden, um Docker wirklich sicher betreiben zu können.

A. Sicherheit des Docker Daemons und Host Systems

Der Docker Daemon ist der Service im Hintergrund, welcher dafür sorgt, dass die einzelnen Container separat laufen können. Das die Schwachstellen nicht nur in einem Container liegen, sondern auch das System der Containerisierung, in diesem Fall der Docker Daemon, betreffen können, zeigt ein erfolgreicher Angriff auf die Plattform *Play with Docker*. Diese Online-Platform bezeichnet sich als Docker Spielwiese und stellt Nutzern kostenlose Container zur Verfügung, mit denen Docker Experimente durchgeführt werden können. Konkret bekommt jeder User einen Docker Container mit Alpine-Linux. In diesem persönlichen Container können wiederum neue Container gestartet werden. Doch wie sich herausstellte, gelang es Sicherheitsforschern des Unternehmens CyberArk Ende 2018 aus diesem persönlichen Container auszubrechen und Code auf dem Host-System auszuführen. Dabei versuchten die Angreifer dem Kernel, der bei der Container Technologie von allen Containern geteilt wird, ein eigenes Modul unterzujubeln, mit dem sie Zugriff auf eine Shell und damit das System bekommen können. Das Problem im Falle *Play with Docker* war, dass die Container mit erhöhten Rechten gestartet wurden. Erst dies ermöglichte das Laden eines eigenen Kernel Moduls. Dieser Angriff sollte dazu dienen, den Cloud-Infrastruktur Betreibern zu zeigen, dass keine hundertprozentige Isolierung ohne weitere Maßnahmen möglich ist.[10]
Um zu verstehen wie genau der Angriff von statten ging und wie er verhindert werden kann, ist wichtig zu betrachten, wie die Container Technologie auf Kernel Ebene funktioniert. Wie bereits mehrfach erwähnt, wird bei der Containerisierung im Gegensatz zu herkömmlichen Maschinen derselbe Kernel für alle Container verwendet. Dabei wird mittels des Linux Kernel der Container in einem abgetrennten Unterbereich des Kernels ausgeführt. Um dies zu realisieren werden unter anderem Namespaces und Cgroups benutzt. Namespaces wird hauptsächlich dazu genutzt um die eigenen Ressourcen unabhängig vom Hauptsystem zu verwalten. Zum Beispiel kann

damit ein root User im Container über gewohnte User ID (UID) 0 angesprochen werden. Selbiger User bekommt jedoch außerhalb des Containers eine unprivilegierte UID. Cgroups funktioniert ähnlich wie Namespaces und schränkt verfügbare Ressourcen auf bestimmte Prozesse ein. Cgroups wirkt meist auf Gruppen von Prozessen und kann diese auch Einfrieren und Aufwecken. Mit der Kombination dieser beiden Systeme können die Container emuliert werden. Jedoch ist zu beachten, dass auch hier bei unachtsamer Rechteverteilung im Host-System Ausbrüche aus den Containern möglich sind.[2]

1) Kernel absichern: Um ein Ausbrechen aus dem Container und daraus folglich eine Übernahme des Host Systems mit den darauf laufenden Prozessen und Containern zu unterbinden kann der Kernel weiter abgeriegelt werden. Dabei gibt es wie in vielen Bereichen der Informatik keine allgemein anwendbare Lösung, sondern nur Empfehlungen, welche Einstellungen getroffen werden können oder sollen. Je nach Anwendungsgebiet der Container könnten gewisse Einstellungen dass System zu stark reglementieren, wodurch je nach Anwendungsfall andere Einstellungen getroffen werden müssen. Grundsätzlich empfohlen wird das Verwenden des aktuellsten Kernels, da dieser nach wie vor das Herzstück des Systems ist. Ebenso werden regelmäßig Schwachstellen im Kernel in Bezug auf die Container-Technologie gefunden und behoben.Um herauszufinden, welche Version des Kernels verwendet wird, kann folgender Befehl verwendet werden: `uname -a`. Die Ausgabe zeigt dann die Version des Kernels an. Mithilfe von `sudo apt-get dist-upgrade` kann der Kernel auf die neueste Version gebracht werden. Ebenso sollte der Root User deaktiviert werden. Falls es einem Container gelingt, aus der Isolation auszubrechen, sollte er nicht mit einem root Zugang im Host System begrüßt werden. Desweiteren bringt die Deaktivierung der Passwort Authentifizierung eine weitere Sicherheitsvorkehrung. Statt der Anmeldung mittels Passwort findet die Anmeldung nun mit SSH Keys statt. Ein weiterer Punkt das System sicher zu machen, ist darauf zu verzichten, Container im privilegierten Modus, also mit erhöhten Rechten auszuführen. Ein Container, der aus der Isolation ausbricht, hat damit alle Rechte und kann das Hostsystem in jeder beliebigen Weise bearbeiten und manipulieren. Zudem kann ein Container, welcher mit erhöhten Rechten ausgeführt wird neue Module in den Kernel laden. Dies hat zufolge, dass das Modul in allen Containern zur Verfügung steht. Da der Kernel unter allen Containern geteilt wird, stehen auch alle Kernel Module in allen Containern zur Verfügung. Deshalb ist es wichtig den Kernel nur auf die nötigsten Module zu beschränken und zu verhindern, dass einzelne Container im Falle eines Ausbruchs weitere Module laden können.[5]

2) Docker Installation absichern: Ebenso stehen Tools zur Verfügung, die die Docker Installation auf ihre Sicherheit überprüfen. So zum Beispiel das Tool *docker-bench-security*. Hierbei handelt es sich um ein offizielles Docker Image mit einem Skript, welches viele Best-Practices im Bezug auf das Deployen von Docker Containern überprüft und bewertet. In einer Docker Installation kann es einfach mit dem Befehl `docker run -it --net host --pid host --userns host --cap-add audit_control -e DOCKER_CONTENT_TRUST=\$DOCKER_CONTENT_TRUST -v /etc:/etc -v /usr/bin/docker-containerd:/usr/bin/docker-containerd -v /usr/bin/docker-runc:/usr/bin/docker-runc -v /usr/lib/systemd:/usr/lib/systemd -v /var/lib:/var/lib -v /var/run/docker.sock:/var/run/docker.sock --label docker_bench_security docker/docker-bench-security` ausgeführt werden. Dies hat zur Folge, dass das entsprechende Image vom Docker Hub heruntergeladen und ausgeführt wird. In der resultierenden Ausgabe finden sich alle überprüften Punkte mit einer dem Ergebnis entsprechenden farblichen Markierung. So können einzelne Punkte entweder mit *WARN*, *NOTE*, *PASS* oder *INFO* versehen werden. So wird darauf hingewiesen, welche Einstellungen eventuell als kritisch angesehen werden und kontrolliert, bzw. geändert werden sollten.[5]

B. Sicherheit der Docker Images

Da nun die Docker Installation abgesichert ist, kann ein Blick auf eine weitere mögliche Sicherheitslücke geworfen werden. So können auch Fehler und Schwachstellen über die Images erzeugt werden. Demzufolge gibt es an dieser Stelle wieder einige Best Practices die beschreiben, wie Docker Images allgemein sicherer gemacht werden können. Leider findet sich auch an dieser Stelle keine allgemeingültige Anleitung, da die Vielfalt an Docker Images zu groß und die Einsatzbereiche zu unterschiedlich sind. Docker Images sind grundsätzlich Base Images, die durch diverse Pakete erweitert und zu dem benötigten Endresultat zusammengebaut werden. Genau hier kann ein Vorteil von Docker Images zu einem Nachteil werden. Der hier genannte Vorteil ist, dass sich in dem ausgeführten Image nur jene Pakete befinden, die für die Ausführung des eigentlichen Programms benötigt werden. So ist in einem Image für eine Python Anwendung nur das Python Environment installiert, keinesfalls aber die Umgebung für .NET Programme. Somit können sich zu diesem Zeitpunkt schon Schwachstellen in nicht benötigten Paketen ausschließen lassen. Jedoch sind damit nicht alle Sicherheitslücken geschlossen, da sich auch in den noch benötigten Paketen Schwachstellen befinden können. Hiermit setzt sich das Image nur noch aus Bibliotheksfunktionen der jeweiligen Programmiersprache, den zusätzlichen Packages und dem eigens geschriebenen Quellcode zusammen. In allen 3 übrigen Komponenten können sich noch Schwachstellen befinden. Um nun Sicherheitslücken in den Images zu finden sollten diese regelmäßig darauf hin untersucht werden.

1) Docker Image Security Scanning: Mithilfe diverser Programme können Docker Images auf Schwachstellen überprüft werden. Die Funktionsweise dieser Programme ist nahezu identisch. Das Programm durchforstet dabei alle auf dem Image installierten Pakete und Abhängigkeiten und kontrolliert diese auf bekannte Sicherheitslücken. Im folgenden wird beispielhaft mit dem Tool Anchore eine Debian Image

auf Sicherheitslücken durchforstet. Mit dem unter Listing 3 gezeigten Satz an Shell Befehlen kann Anchore in einem Docker Container installiert werden.

Listing 3. Installation von Anchore

```
mkdir ~/aevolume
cd ~/aevolume

docker pull docker.io/anchore/anchore-
    engine:latest
docker create --name ae docker.io/anchore
    /anchore-engine:latest
docker cp ae:/docker-compose.yaml ~/
    aevolume/docker-compose.yaml
docker rm ae

docker-compose pull
docker-compose up -d

export ANCHORE_CLI_USER=admin
export ANCHORE_CLI_PASS=foobar

docker run --net=host -e ANCHORE_CLI_URL=
    http://localhost:8228/v1/ -it anchore/
    engine-cli
```

Befindet man sich nun in dem eben gestarteten Container, kann man das zu testende Image hinzufügen. Dies erfolgt durch den Befehl `anchore-cli image add docker.io/library/debian:latest`. Mithilfe des Befehls `anchore-cli image wait docker.io/library/debian:latest` kann der Status des Schwachstellen Scans abgefragt werden. Das Kommando `anchore-cli image list` listet alle Images auf, die bei Anchore für Security Scans hinterlegt sind. Sobald die Überprüfung des Images abgeschlossen ist, kann mittels `anchore-cli image vuln docker.io/library/debian:latest all` das Ergebnis abgefragt werden. Der Parameter all bewirkt, dass sowohl Schwachstellen aus den Paketen sowie zusätzlich aus dem Betriebssystem angezeigt werden. In der Spalte *Severity* findet sich eine Einstufung inwiefern die Schwachstelle sicherheitskritisch für das komplette System ist. Der Befehl `anchore-cli evaluate check docker.io/library/debian:latest` führt noch eine Überprüfung der Ergebnisse durch und gibt an, ob das Image den grundsätzlichen Richtlinien entspricht. Ist dies der Fall, so ist in der Ausgabe das Resultat *Status: pass* vermerkt. Nun ist hierbei jedoch anzumerken, dass es sich bei dem gescannten Image lediglich um ein offizielles, viel genutztes Debian Image handelte. Daraus folgt auch, dass dieses Image an sich sehr sicher ist. Würden wir jetzt probehalber ein Python Base Image mit einem einfachen Python Hello-World Output scannen, so gingen die Anzahl der Schwachstellen schon in den hohen dreistelligen Bereich. Darunter befinden sich auch einige, die in der Spalte *Severity* mit *Low* oder *Medium* gekennzeichnet werden. Ebenso resultiert der *evaluate check*-Befehl mit dem Ergebnis *Status: fail*. All diese Schwachstellen und Sicherheitslücken wurden lediglich durch das Nutzen eines offiziellen Python Images eingeschleust. Demzufolge ist das Scannen von Docker Container Images ein nötiger aber zu gleich nicht der einzige Weg Images abzusichern.[7]

2) Bessere Docker Images finden: Ein wirklich sicheres Image zu bauen, wäre eventuell dadurch gegeben mit einem wirklich leeren Container zu beginnen. Dies würde jedoch wahrscheinlich zu mehr Sicherheitslücken führen, wie wenn ein vorgefertigtes Base Image verwendet werden würde, da wirklich jedes Paket einzeln installiert und konfiguriert werden müsste. Da viele Base Images oftmals mit Shells oder Paket Managern vorinstalliert werden, wurden von Google die sogenannten *Distroless* Images gebaut. Diese beinhalten wirklich nur die minimalsten Anforderungen zur Ausführung der Applikation. Im folgenden wird ein Distroless Image Anchore zur Überprüfung vorgelegt. Mit Listing 4 wird eine Python Distroless Image erzeugt und mit einem einfachen Hello-World Skript ausgestattet.

Listing 4. Beipiel: Distroless Python Image

```
FROM python:3-slim AS build-env
ADD . /app
WORKDIR /app

FROM gcr.io/distroless/python3
COPY --from=build-env /app /app
WORKDIR /app
CMD ["hello.py", "/etc"]
```

Die Überprüfung dieses Images ergibt lediglich eine Anzahl an Schwachstellen im mittleren zweistelligen Bereich und wenige, die mit *Low* gekennzeichnet sind. Ebenso erreicht das Image *Status: pass* im Richtlinientest. Dementsprechend kann daraus gefolgert werden, dass Distroless Images solchen wie zum Beispiel Debian vorgezogen werden sollten.[7]

3) Grenzen des Docker Image Security Scannings: Doch auch das Verwenden der Distroless Images ist nicht das non plus ultra in Sachen Docker Container Sicherheit. Auch Programme wie Anchore können nicht alle Schwachstellen finden. Zwar können die Testrichtlinien angepasst werden, jedoch bringt dies meist auch nicht den gewünschten Mehrerfolg. Diese Art der Programme, die Pakete auf öffentlich bekannte Sicherheitslücken durchsuchen, sind oftmals in ihrer Ausführung beschränkt. So kann es zu einem Problem für Anchore werden, wenn Pakete unter Verwendung sind, deren Schwachstellen nicht dokumentiert sind. Im Endeffekt bedeutet dies, das die Pakete nicht automatisch sicher sind, nur weil keine Sicherheitslücken gefunden wurden. Zudem können Open-Source Pakete Probleme verursachen. Hier kann es passieren, dass diese unter einem anderen Namen verwendet werden, was dazu führt, dass der Security Scanner keine Schwachstellen für das Paket findet. Für diesen Fall gibt es wiederum spezielle Programme, die den Ursprung des Codes zurückverfolgen. Zusammenfassend lässt sich sagen, dass das Untersuchen der Docker Images auf Schwachstellen ein Schritt mehr in Richtung Sicherheit ist, keinesfalls jedoch der einzige. Nur das Zusammenspiel bereits genannter und noch folgender

Sicherheitsmaßnahmen sorgt für erhöhte Sicherheit. Wichtig ist auch, dass das Scannen der Docker Images regelmäßig wiederholt wird, vor allem jedoch dann, wenn Teile des Images oder Quellcodes verändert oder hinzugefügt werden.[4]

C. Sicherheit in Docker Container

Nachdem nun die Docker Installation an sich, sowie die Images auf Schwachstellen geprüft und Wege erläutert wurden, wie diese Sicherheitslücken geschlossen werden können, fehlt noch eine Komponente der Containervirtualisierung. So kann jeder einzelne laufende Container eine Schwachstelle für das gesamte System darstellen. Jeder Container kann je nach Konfiguration unterschiedlich agieren und reagieren. Deshalb muss jeder Container separat abgesichert und überwacht werden. Dazu zählt unter anderem die Begrenzung des Containers auf Systemressourcen wie CPU und Arbeitsspeicher. Zudem muss die Netzwerksicherheit gewährleistet, sowie das Einbinden von externen Volumnes geregelt sein.[3]

1) Ressourcenbegrenzung von Containern: Das Begrenzen von Ressourcen ist ein wichtiger Punkt im Betrieb von Containervirtualisierung. Sollte es zu einem Ausbruch eines Programms in einem Container oder einer Fehlfunktion eines Containers kommen, so kann im schlimmsten Fall das Host-System negativen Einfluss nehmen. Explizit bedeutet das, dass ein Container einen ungewöhnlich hohen Rechenbedarf aufweisen kann, was dazu führt, das anderen Containern nicht mehr genügend Rechenleistung zur Verfügung steht. Aus diesem Grund ist es besonders wichtig beim gleichzeitigen Betrieb von mehreren Containern jeweils Grenzen anzugeben. Dies geschieht einfach per Parameterübergabe bei Start des Containers. Mittels `sudo docker run -it --memory="1g" ubuntu /bin/bash` wird der Container auf die Nutzung von maximal einem Gigabyte Arbeitsspeicher beschränkt. Der Parameter akzeptiert positive, ganzzahlige Werte mit den Suffix *b, k, m, g*. Mit *–memory-reservation* kann ein Mindestwert gesetzt werden. Desweiteren wird über den Parameter *–cpus=".5"* in diesem Fall die CPU Zeit auf 50000 Microsekunden gesetzt. Die Option *–cpus-shares* setzt die Anzahl an CPU Zyklen. Der Standardwert hierbei ist 1024.[11]

2) Netzwerksicherheit in Containern: Da Container meistens nicht abgekapselt arbeiten, sondern mit der Außenwelt kommunizieren müssen, wird eine Netzwerkverbindung benötigt, die Zugriffe von außen auf den Container zulässt. Wie bereits erwähnt wird jedoch grundsätzlich jeder Netzwerktraffic, welcher von außen auf den Container zugreifen will abgeblockt. Laufen jedoch innerhalb der Container Microservices, so müssen diese miteinander kommunizieren. Da diese sich jedoch nicht auf dem gleichen System, sondern getrennten Containern befinden, sind Technologien wie Remote Procedure Calls nicht anwendbar. Stattdessen wird auf netzwerkbasierte Kommunikation zurückgegriffen. Damit nun Datenverkehr mit der Außenwelt ermöglicht werden kann, müssen die benötigten Ports explizit freigegeben werden. Konkret bedeutet dies, das beim Erstellen des Images der Port mittels des Befehls *EXPOSE* und folgend der Portnummer

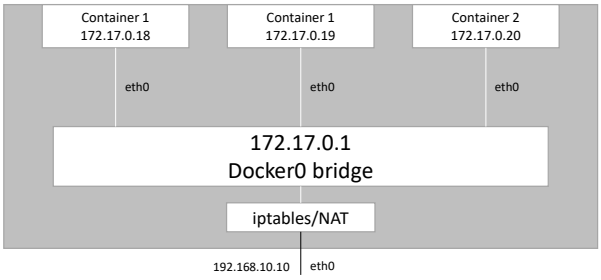

Abbildung 3. Netzwerkverbindung von Docker Containern

spezifiziert wird. Dies hat zufolge, dass der Docker Daemon beim Start des Containers eine IP und Portumleitung einrichtet. Diese Weiterleitung lässt ab sofort jegliche Zugriffe vom internen oder externen Netzwerk auf diesen Port zu. Dies ist in einem produktivem Umfeld jedoch nicht zu empfehlen, da die Öffnung für alle nach außen ein großes Sicherheitsrisiko mit sich bringt.[8]
Sicherer ist die händische Erstellung der Umleitung mit den jeweils gewünschten Einschränkungen. Hier gibt es die Möglichkeit dem Docker Daemon komplett zu verbieten, die iptables Datei im Host System zu verändern. Laut Entwicklerdokumentation erhöht dies jedoch das Risiko, die Netzwerkkommunikation komplett lahmzulegen. Die von den Entwicklern empfohlene Variante hingegen ist die vom Docker Daemon eingerichteten Regeln um weitere negierte zu ersetzen. Konkret heißt das, falls die Kommunikation nicht für externe Netzwerke geöffnet werden sollte, wird eine Regel eingerichtet, die dies verhindert. Dadurch wird der Docker Daemon nicht in der Ausführung seiner Tätigkeit gehindert, eine Sicherheit jedoch dennoch gewährleistet. Der Befehl `iptables -I DOCKER-USER -i ext_if ! -s 192.168.1.1 -j DROP` gibt zum Beispiel an, dass nur noch die IP-Adresse *192.168.1.1* Zugriff hat. Anzumerken ist hierbei, das *ext_if* mit dem Namen des Netzwerkadapters ausgetauscht werden muss. Alles in allem ist die Netzwerksicherheit in Docker ein wichtiger Punkt, um gegen Angriffe von außen geschützt zu sein. Jedoch gibt es auch hier keine allgemeingültige Anleitung, sondern lediglich einige Inspirationen wie dieses komplexe Thema abgesichert werden kann.[1]

3) Docker Volume Sicherheit: Eine weitere Schwachstelle in der Containervirtualisierung können extern eingebunden Volumes darstellen. Werden diese unkontrolliert verwendet, kann entweder der Container Dateien außerhalb unerlaubt bearbeiten oder es können Schaddateien in den Container eindringen. Letzteres ist speziell dann kritisch, wenn das gleiche Volume von mehreren Containern genutzt wird. Die Docker Umgebung bringt kein Sicherheitskonzept im Bezug auf externe Daten mit sich. Konkret bedeutet das, dass Ordner wie /etc in einen Container gemountet werden können. Wird der Container zusätzlich noch mit root-Rechten ausgeführt, so

kann er uneingeschränkt wichtige Dateien des Host-Systems manipulieren. Der bevorzugte Weg hier wäre jeden einzelnen Container unter einem anderen User mit anderer UID laufen zu lassen. Das wiederum hätte zur Folge, das jeder User Zugriff auf das Verzeichnis */var/lib/docker/volumes* bräuchte. Würde man das Verzeichnis auf Vollzugriff umstellen, hätte man eine noch viel größere Sicherheitslücke geschaffen. Die Verzeichnisberechtigungen für jeden Nutzer einzeln zu konfigurieren, wäre bei einer großen Anzahl an Containern eine mühselige, fehleranfällige und händische Arbeit. Wird ein Volume nur von einem einzigen Container verwendet, so können entsprechende Berechtigungen gesetzt werden, die dann auch die benötigte Sicherheit bieten. Wird das Volume jedoch mehrfach benutzt, so ist, wie eben beschrieben, noch kein empfohlener Weg vorhanden. Da jedoch in vielen Anwendungsfällen von Docker Orchestrierungstools wie Kubernetes verwendet werden, muss man an dieser Stelle darauf vertrauen, dass das Tool die Volumes entsprechend abgesichert einbindet. So ist zu sehen, dass auch Volumes eine Schwachstelle in der Containervirtualisierung sein können und deshalb mit Vorsicht angewendet werden müssen.[6]

V. Ausblick

Da nun alle Komponenten der Containervirtualisierung mit Docker betrachtet und deren potentiellen Sicherheitslücken erläutert wurden, kann ein Ausblick darauf gegeben werden, was noch an Neuerungen und Erweiterungen kommen sollte, um den Betrieb noch sicherer zu gestalten. Wie im vorherigen Kapitel bereits erläutert, ist das Einbinden externer Dateien und Verzeichnisse in Docker noch nicht wirklich mit einem Sicherheitssystem ausgestattet. Somit ist dies definitiv ein Punkt, der von Seiten der Entwickler ausgebaut werden muss. Da der Trend jedoch nach wie vor in Richtung Virtualisierung per Container geht, wird sich in den nächsten Jahren in diesem Bereich sicherlich noch einiges weiterentwickeln.

VI. Fazit

Alles in allem gibt die durchgeführte Analyse der Schwachstellen einen groben Überblick über die möglichen Angriffspunkte. Dies stellt jedoch auch heraus, dass in dieser Technologie viele einzelne Komponenten betrachtet werden müssen, um das ganze System sicher betreiben zu können. Auf jeden Fall muss der Betreiber des Systems proaktiv handeln und von Beginn an die Installation so absichern, dass es gar nicht erst zu gröberen Zwischenfällen mit einzelnen Containern kommen kann. Auch reicht es nicht aus, nur einzelne Bestandteile des System, wie zum Beispiel den Docker Daemon abzusichern. Ein gutes Sicherheitskonzept kann nur dann erfolgreich umgesetzt werden, wenn alle drei Komponenten entsprechen sicher operieren. Im Gegensatz zur Virtualisierung mit virtuellen Maschinen wird in diesem Fall viel unter den Systemen geteilt, was dazu führt das viele Komponenten miteinander agieren müssen. Fällt nur eines dieser Systeme aufgrund einer Sicherheitslücke aus, ist die Stabilität des Gesamtsystems gefährdet. Somit ist die Containervirtualisierung ein mächtiges System, das auch an geeigneter Stelle Ressourcen sparen kann. Teilweise ist es jedoch trotz ihrer hohen Grundsicherheit den Schwachstellen einiger Komponenten ausgesetzt.

Literatur

[1] URL: https://docs.docker.com/network/iptables/ (besucht am 02.07.2020).

[2] Stephan Augsten. *Was sind Docker Container?* 2017. URL: https://www.dev-insider.de/was-sind-docker-container-a-597762/ (besucht am 29.06.2020).

[3] Gabriel Avner. *Docker Container Security: Challenges and Best Practices*. 2019. URL: https://resources.whitesourcesoftware.com/blog-whitesource/docker-container-security-challenges-and-best-practices (besucht am 01.07.2020).

[4] Gabriel Avner. *Docker Image Security Scanning: What It Can and Can't Do*. 2019. URL: https://resources.whitesourcesoftware.com/blog-whitesource/docker-image-security-scanning (besucht am 01.07.2020).

[5] Theo Despoudis. *How to Lock Down the Kernel to Secure the Container*. 2019. URL: https://thenewstack.io/how-to-lock-down-the-kernel-to-secure-the-container/ (besucht am 29.06.2020).

[6] Chris M. Evans. *Docker Data Security Complications*. 2017. URL: https://www.networkcomputing.com/cloud-infrastructure/docker-data-security-complications (besucht am 03.07.2020).

[7] Martin Heinz. *Analyzing Docker Image Security*. 2020. URL: https://martinheinz.dev/blog/19 (besucht am 01.07.2020).

[8] Fei Huang. *Improving Docker Security: A Better Way to Secure Your Container Network*. URL: https://neuvector.com/docker-security/improving-docker-security-better-way-to-secure-containers/ (besucht am 02.07.2020).

[9] Thomas Claudius Huber. *Docker: Einstieg in die Welt der Container*. 2019. URL: https://entwickler.de/online/windowsdeveloper/docker-grundlagen-dotnet-container-579859289.html (besucht am 03.07.2020).

[10] Jan Mahn. *Sicherheitsforscher brechen aus Docker-Container aus*. 2019. URL: https://www.heise.de/security/meldung/Sicherheitsforscher-brechen-aus-Docker-Container-aus-4276108.html (besucht am 29.06.2020).

[11] Max Ostryzhko. *How to limit a docker container's resources on ubuntu 18.04*. 2018. URL: https://hostadvice.com/how-to/how-to-limit-a-docker-containers-resources-on-ubuntu-18-04/ (besucht am 01.07.2020).

[12] Lukas Pustina. *Docker Basics: Einführung in die System-Level-Virtualisierung*. 2015. URL: https://entwickler.de/online/development/docker-basics-system-level-virtualisierung-125514.html (besucht am 29.06.2020).