Holger Gantikow

Containing Containers?

Wie lässt sich der Wahl bändigen?

Status Quo Container Security

TUEBIX, JUNI 2017

Trusted partner for your Digital Journey

© Atos

Atos

LinkedIn

Holger Gantikow

Senior Systems Engineer at science + computing ag

Stuttgart und Umgebung, Deutschland | IT und Services

133 Kontakte

Aktuell	science + computing ag, science + computing ag, a bull group company
Früher	science + computing ag, Karlsruhe Institute of Technology (KIT) / University of Karlsruhe (TH)
Ausbildung	Hochschule Furtwangen University

Zusammenfassung

Diploma Thesis "Virtualisierung im Kontext von Hocherfügbarkeit" / "Virtualization in the context of High Availability , IT-Know-How, Experience with Linux, especially Debian&Red Hat, Windows, Mac OS X, Solaris, *BSD, HP-UX, AIX, Computer Networking, Network Administration, Hardware, Asterisk, VoIP, Server Administration, Cluster Computing, High Availability, Virtualization, Python Programming, Red Hat Certified System Administrator in Red Hat OpenStack

Current fields of interest:
Virtualization (Xen, ESX, ESXi, KVM), Cluster Computing (HPC, HA), OpenSolaris, ZFS, MacOS X, SunRay ThinClients, virtualized HPC clusters, Monitoring with Check_MK, Admin tools for Android and iOS, Docker / Container in general, Linux 3D VDI (HP RGS, NiceDCV, VMware Horizon, Citrix HDX 3D Pro)

Specialties: Virtualization: Docker, KVM, Xen, VMware products, Citrix XenServer, HPC, SGE, author for Linux Magazin (DE and EN), talks on HPC, virtualization, admin tools for Android and iOS, Remote Visualization

Senior Systems Engineer
science + computing ag
April 2009 – Heute (8 Jahre 3 Monate)

System Engineer
Übersetzung anzeigen

science + computing ag, a bull group company
2009 – Heute (8 Jahre)

Graduand
science + computing ag
Oktober 2008 – März 2009 (6 Monate)

Diploma Thesis: "Virtualisierung im Kontext von Hochverfügbarkeit" - "Virtualization in the context of High Availability"

Intern
Übersetzung anzeigen

Karlsruhe Institute of Technology (KIT) / University of Karlsruhe (TH)
August 2008 – September 2008 (2 Monate)

Research on optimization of computing workflow using Sun Grid Engine (SGE) for MCNPX calculations.

Hochschule Furtwangen University
Dipl. Inform. (FH), Coding, HPC, Clustering, Unix stuff :-)
2003 – 2009

Auf Linkedin & Xing & Twitter zu finden

Institut für Cloud Computing und IT-Sicherheit

IfCCITS

Fakten:

- seit 2009 Forschung im Bereich Cloud Computing und IT-Sicherheit
- Leiter: Prof. Dr. Christoph Reich
- Fakultät: Informatik
- Momentan: 5 PhDs, 4 Masters, 6 Bachelors
- Informationen: www.wolke.hs-furtwangen.de

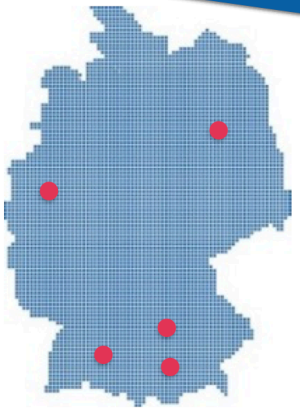
Link: http://wolke.hs-furtwangen.de

Jetzt AtoS!

science + computing ag

Unser Fokus:
IT-Dienstleistungen und Software für technische Berechnungsumgebungen

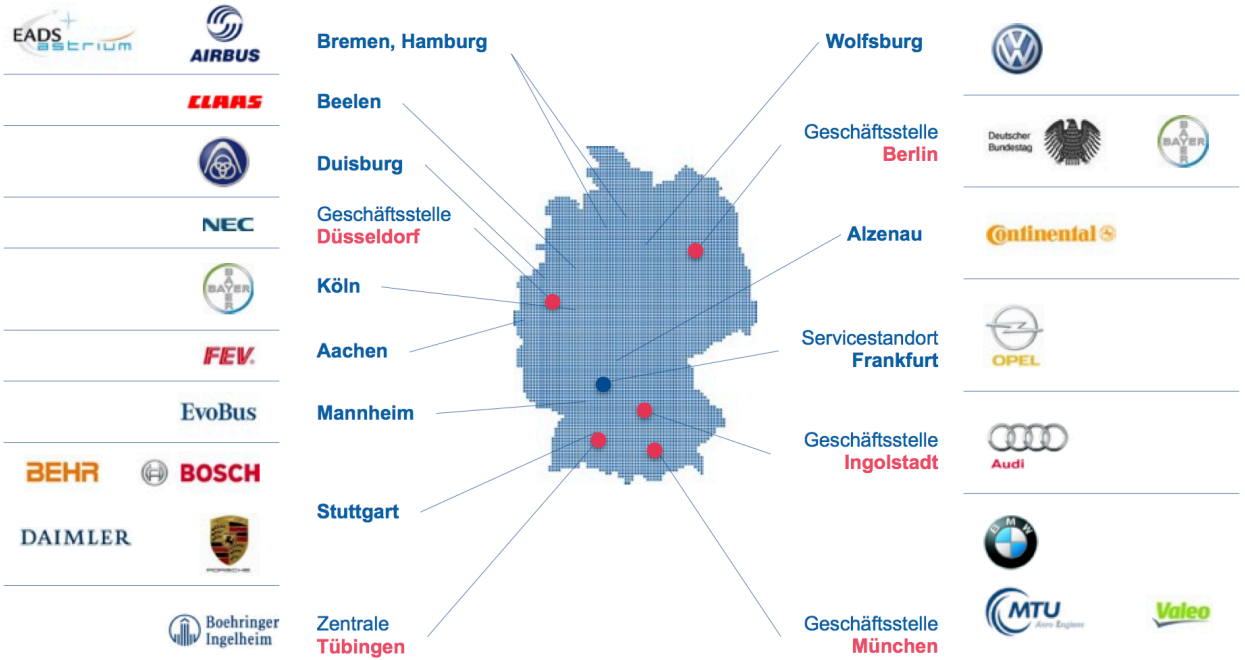
Gründungsjahr	**1989**
Standorte	**Tübingen** **München** **Berlin** **Düsseldorf** **Ingolstadt**
Mitarbeiter	**287**
Hauptaktionär	**Atos SE (100%)** **davor Bull**
Umsatz 2013	**30,70 Mio. Euro**



© 2016 science + computing ag

Kunden der science + computing ag

science + computing
an atos company



© 2016 science + computing ag

Inhalt

Einleitung

Abgrenzung

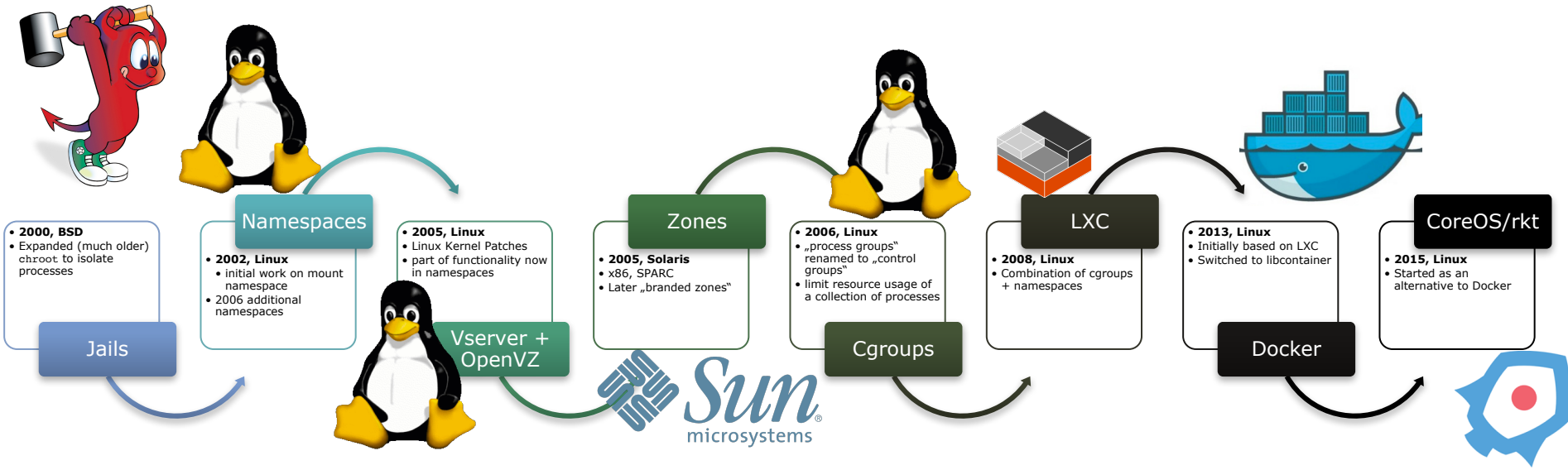
Kein HPC, keine Docker-Alternativen

Kein Singularity+Shifter, kein LXC+Rocket, ...

und auch keine Orchestrierung :(

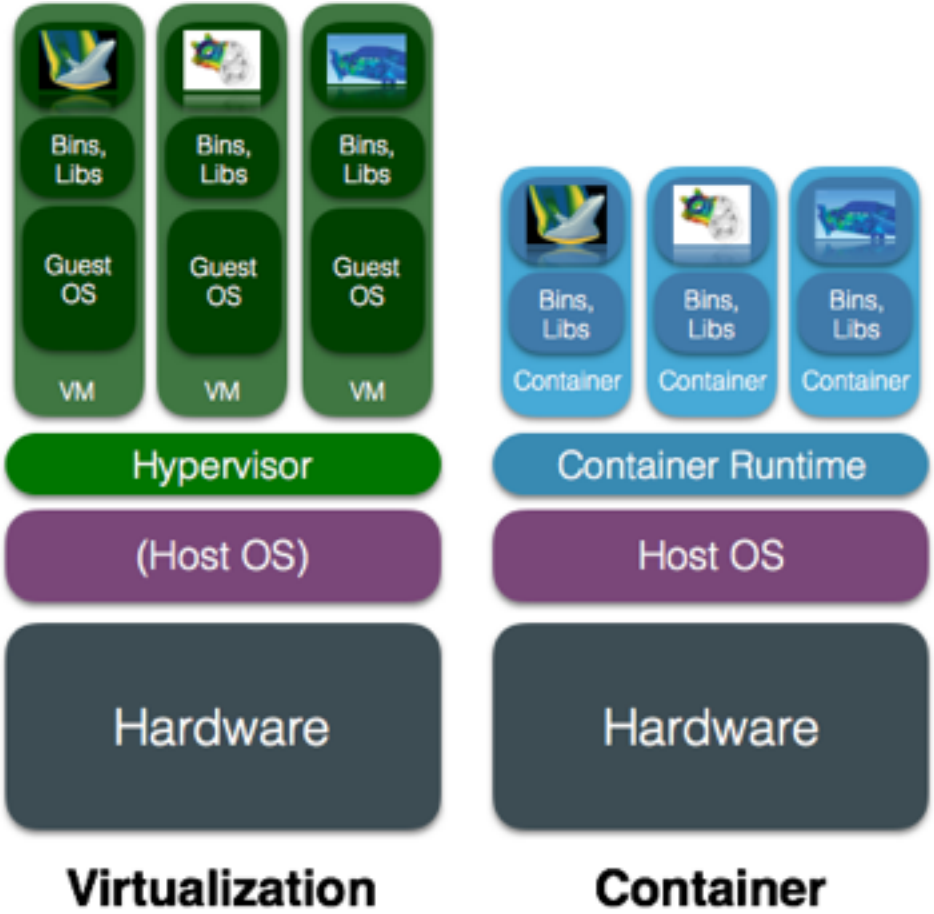
Container Intro

Evolution of *OS-level virtualization*

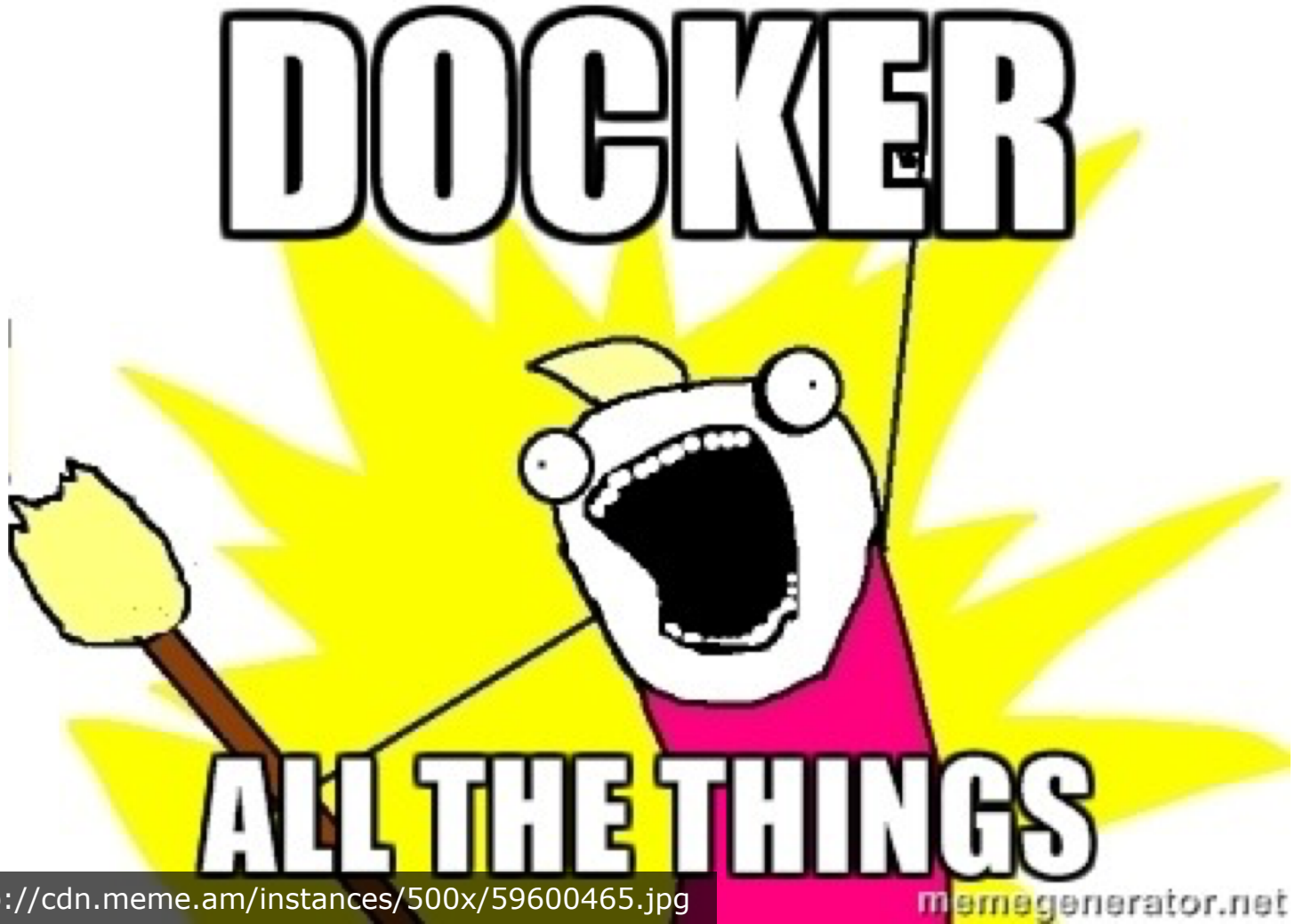


Hypervisor-based virtualization

1999 VMware Workstation 1.0
2001 ESX 1.0 & GSX 1.0
2003 Xen 1st public release
2006 KVM (2.6.10)


Bins, Libs
Guest OS
VM
Bins, Libs
Guest OS
VM
Bins, Libs
Guest OS
VM
Hypervisor
(Host OS)
Hardware
Virtualization
Bins, Libs
Container
Bins, Libs
Container
Bins, Libs
Container
Container Runtime
Host OS
Hardware
Container

Docker Intro



Source: http://cdn.meme.am/instances/500x/59600465.jpg

„Docker is an **open platform** for **developers** and **sysadmins** to build, ship, and run distributed applications, whether on **laptops, data center VMs, or the cloud.**“

https://www.docker.com/whatisdocker

What is Docker

Docker is the world’s leading software container platform.

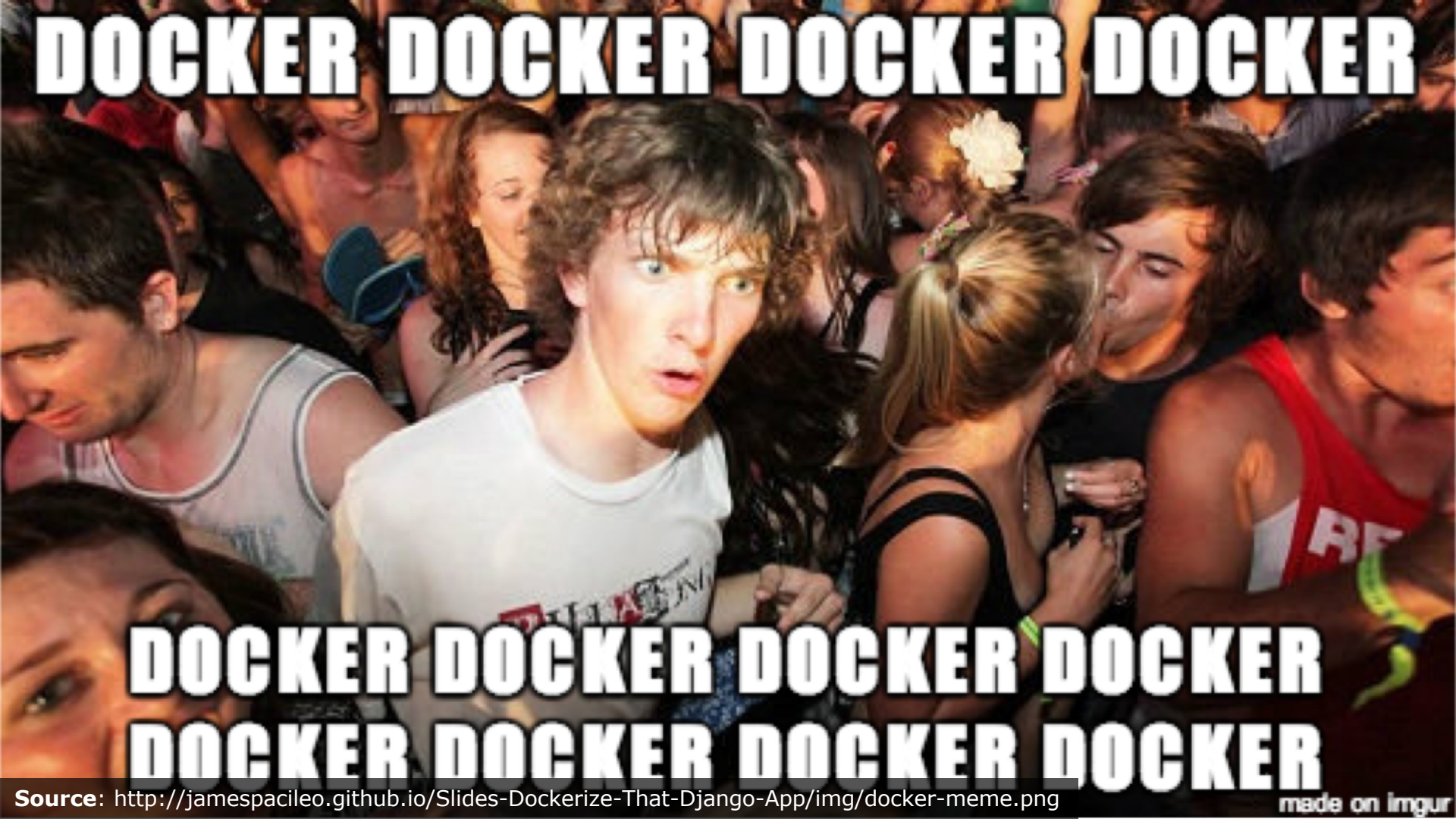
Source: https://www.docker.com/what-docker

From engine to platform

Docker Hub
Docker Toolbox
Docker Compose
Docker Swarm
Docker Machine
Docker Universal Control Plane
Docker Trusted Registry
Docker Cloud
Docker Enterprise Edition
Docker „XYZ“ ;)

The new old Microsoft?

Swarm, Platform lock in (Enterprise Edition), ...
vs *Decoupling the plumbing*



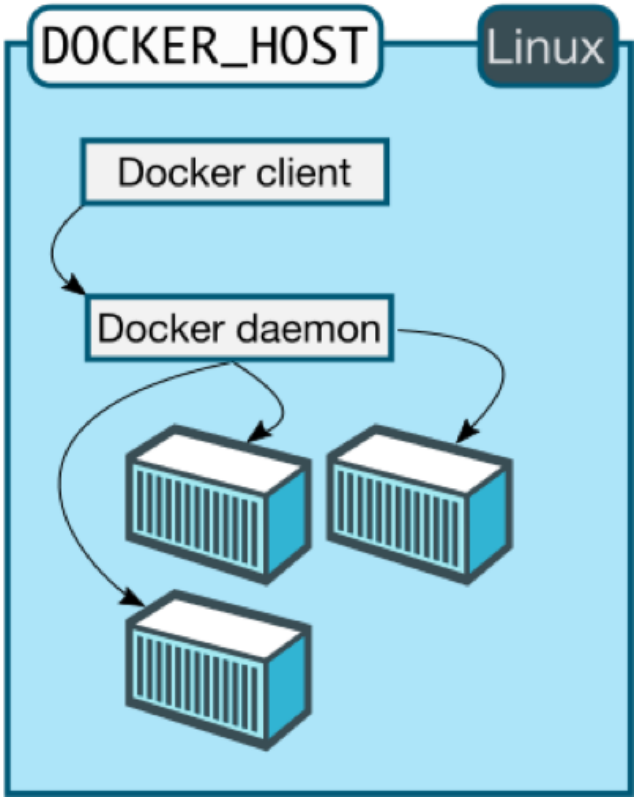
Source: http://jamespacileo.github.io/Slides-Dockerize-That-Django-App/img/docker-meme.png

Docker 101

Source: https://upload.wikimedia.org/wikipedia/commons/7/79/Docker_(container_engine)_logo.png

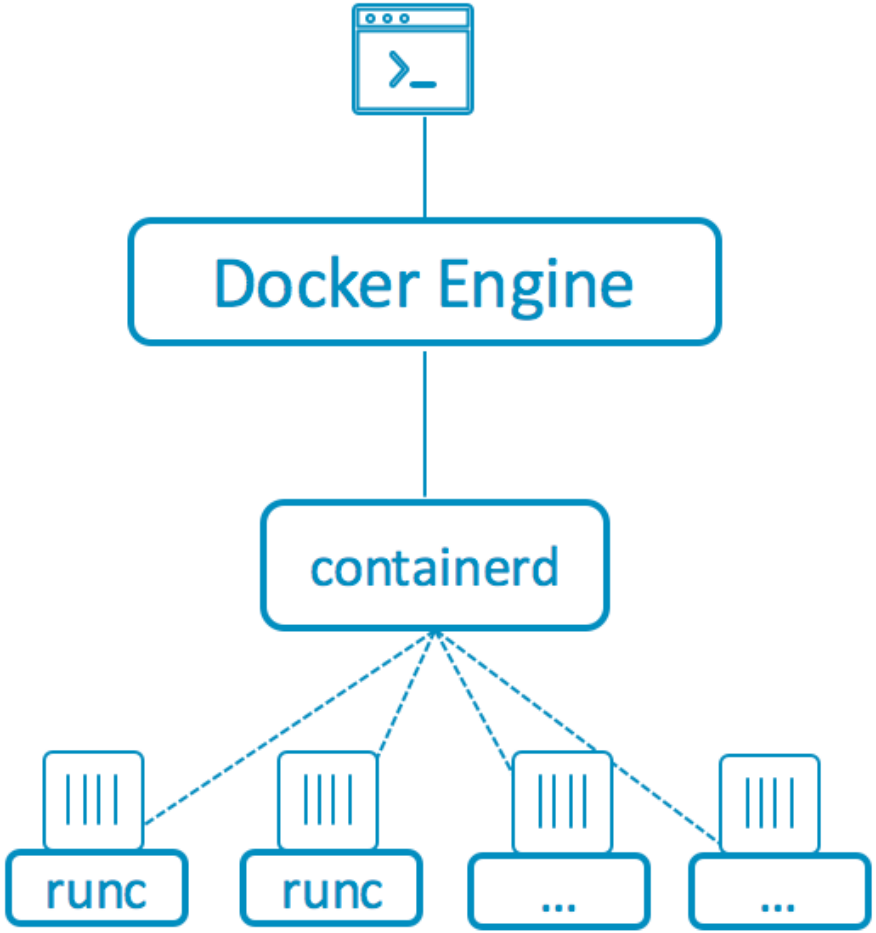
Terminologie
+ Kernkomponenten

Begrifflichkeiten – Core + Workflow Components



Component	Description
Host	(Linux) System with Docker Daemon
Daemon	The engine, running on the host
Client	CLI for interacting with Daemon

Component	Description
Image	contains application + environment
Container	created from image - start, stop, …
Registry	„App Store“ for images Public + private repository possible
Dockerfile	used for automating image build



Same Docker UI and commands

User interacts with the Docker Engine

Engine communicates with containerd

containerd spins up runc or other OCI compliant runtime to run containers

Source: https://blog.docker.com/2016/04/docker-engine-1-11-runc/

Built on existing technology

already included in the Linux Kernel

How are they implemented? Let's look in the kernel source!

- Go to LXR
- Look for "LXC" → zero result
- Look for "container" → 1000+ results
- Almost all of them are about data structures
  or other unrelated concepts like “ACPI containers”
- There are some references to “our” containers
  but only in the documentation

Beweis

Source: https://www.youtube.com/watch?v=sK5i-N34im8 &&
https://de.slideshare.net/jpetazzo/cgroups-namespaces-and-beyond-what-are-containers-made-from-dockercon-europe-2015

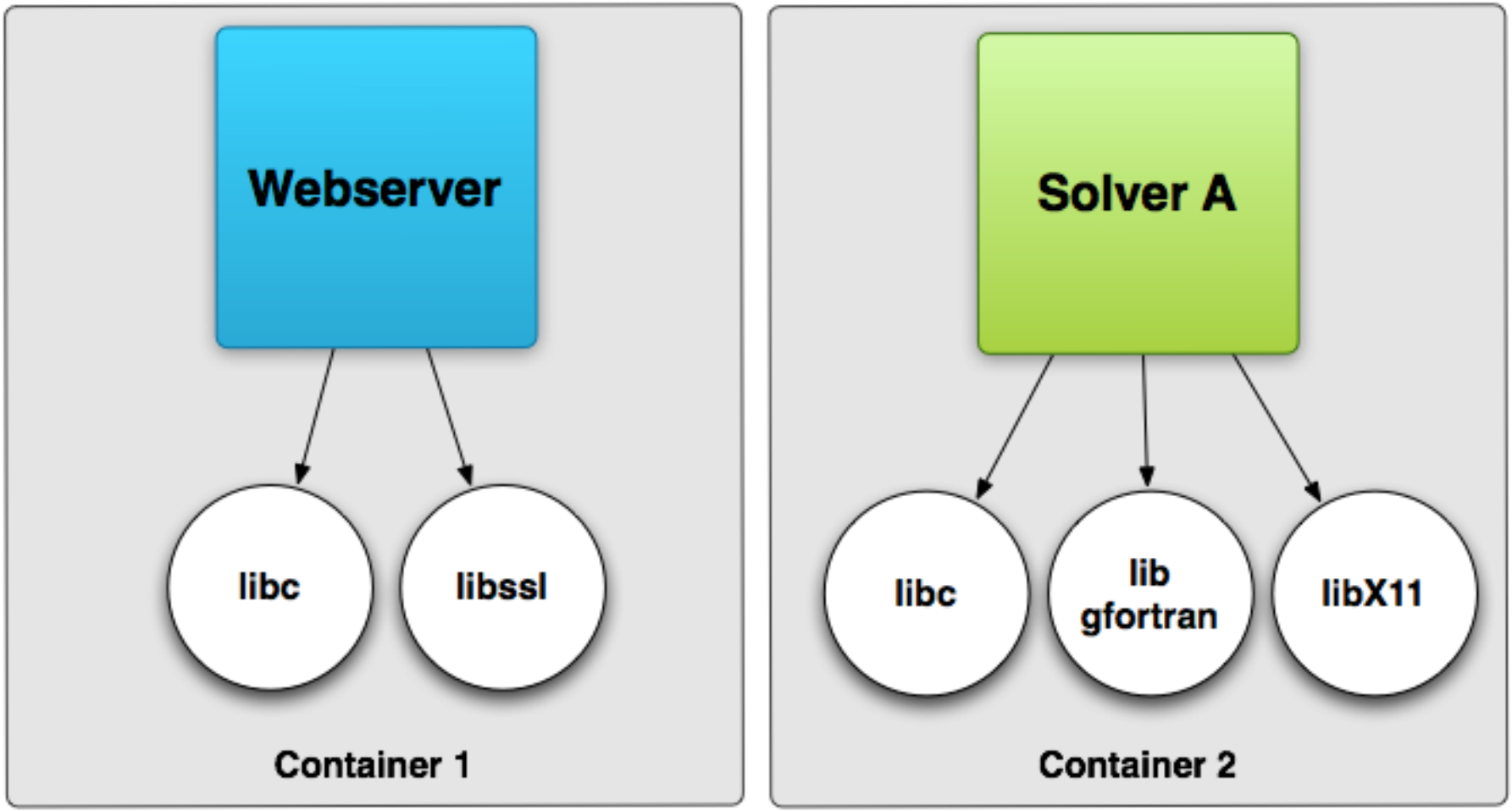
Container = *Namespaces + cgroups*

- Beides Kernelfeatures
  - **Namespaces** : einige Subsysteme *ns-aware* – Illusion *isolierter Betrieb*
  - **Cgroups**: einige Ressourcen kontrollierbar – Limitierung Ressourcenverbrauch

Namespace	Description	Controller	Description
pid	Process ID	**blkio**	Access to block devices
net	Network Interfaces, Routing Tables, …	**cpu**	CPU time
ipc	Semaphores, Shared Memory, Message Queues	**devices**	Device access
mnt	Root and Filesystem Mounts	**memory**	Memory usage
uts	Hostname, Domainname	**net_cls**	Packet classification
user	UserID and GroupID	**net_prio**	Packet priority

Was kann Docker für Dich tun?

Abhängigkeiten isolieren
+ Legacy Code
Conflicting Requirements + Dependencies
+ Code ausliefern


Webserver
libc
libssl
Container 1
Solver A
libc
lib
gfortran
libX11
Container 2

Workflow*workflow*

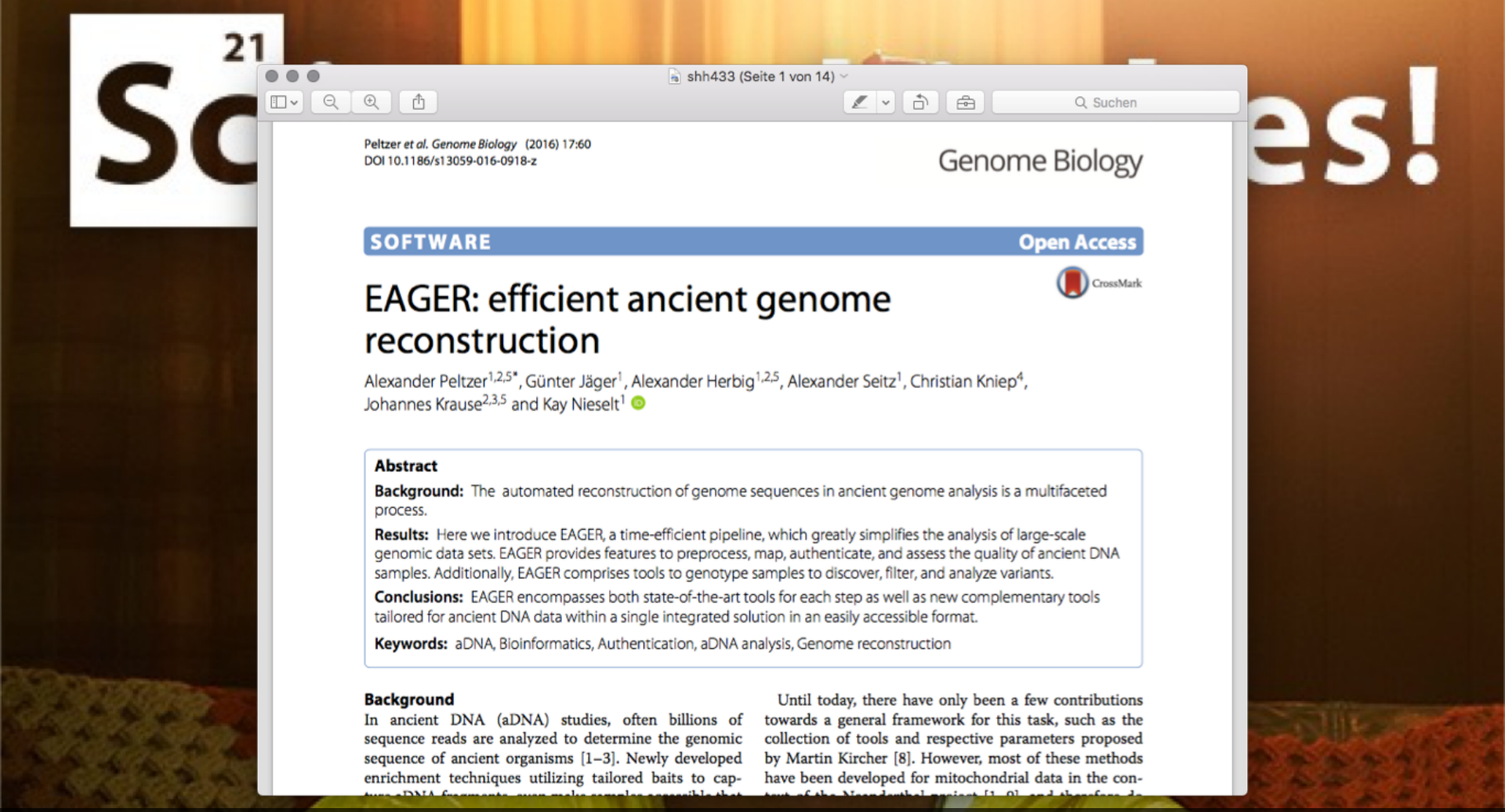
+ Reproduzierbarkeit

„Frozen Environment"

+Flexibilität @HPC

Minimal Dockerfile for Image with $TOOL

```
FROM ubuntu
RUN apt-get update
RUN apt-get install $TOOL
```



Peltzer *et al. Genome Biology* (2016) 17:60
DOI 10.1186/s13059-016-0918-z

Genome Biology

SOFTWARE Open Access

EAGER: efficient ancient genome reconstruction

Alexander Peltzer[1,2,5*], Günter Jäger[1], Alexander Herbig[1,2,5], Alexander Seitz[1], Christian Kniep[4], Johannes Krause[2,3,5] and Kay Nieselt[1]

Abstract

Background: The automated reconstruction of genome sequences in ancient genome analysis is a multifaceted process.

Results: Here we introduce EAGER, a time-efficient pipeline, which greatly simplifies the analysis of large-scale genomic data sets. EAGER provides features to preprocess, map, authenticate, and assess the quality of ancient DNA samples. Additionally, EAGER comprises tools to genotype samples to discover, filter, and analyze variants.

Conclusions: EAGER encompasses both state-of-the-art tools for each step as well as new complementary tools tailored for ancient DNA data within a single integrated solution in an easily accessible format.

Keywords: aDNA, Bioinformatics, Authentication, aDNA analysis, Genome reconstruction

Background

In ancient DNA (aDNA) studies, often billions of sequence reads are analyzed to determine the genomic sequence of ancient organisms [1–3]. Newly developed enrichment techniques utilizing tailored baits to cap-

Until today, there have only been a few contributions towards a general framework for this task, such as the collection of tools and respective parameters proposed by Martin Kircher [8]. However, most of these methods have been developed for mitochondrial data in the con-

Source: www.critic.co.nz/files/article-3423.jpg + Peltzer et al. (2016). EAGER: efficient ancient genome reconstruction.

Performance

„nah am Blech“

An Updated Performance Comparison of Virtual Machines and Linux Containers

Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio
IBM Research, Austin, TX
{wmf, apferrei, rajamony, rubioj}@us.ibm.com

***Abstract*—Cloud computing makes extensive use of virtual machines (VMs) because they permit workloads to be isolated from one another and for the resource usage to be somewhat controlled. However, the extra levels of abstraction involved in virtualization reduce workload performance, which is passed on to customers as worse price/performance. Newer advances in container-based virtualization simplifies the deployment of applications while continuing to permit control of the resources allocated to different applications.**

In this paper, we explore the performance of traditional virtual machine deployments, and contrast them with the use of Linux containers. We use a suite of workloads that stress CPU, memory, storage, and networking resources. We use KVM as a representative hypervisor and Docker as a container manager. Our results show that containers result in equal or better performance than VMs in almost all cases. Both VMs and containers require tuning to support I/O-intensive applications. We also discuss the implications of our performance results for future cloud architectures.

I. Introduction

Virtual machines are used extensively in cloud computing. In particular, the state-of-the-art in Infrastructure as a Service (IaaS) is largely synonymous with virtual machines. Cloud platforms like Amazon EC2 make VMs available to customers and also run services like databases inside VMs. Many Platform as a Servive (PaaS) and Software as a Service (SaaS) providers are built on IaaS with all their workloads running inside VMs. Since virtually all cloud workloads are currently running in VMs, VM performance is a crucial component of overall cloud performance. Once a hypervisor has added overhead, no higher layer can remove it. Such overheads then become a pervasive tax on cloud workload performance. There have been many studies showing how VM execution compares to native execution [30, 33] and such studies have been a motivating factor in generally improving the quality of VM technology [25, 31].

Container-based virtualization presents an interesting alternative to virtual machines in the cloud [46]. Virtual Private Server providers, which may be viewed as a precursor to cloud computing, have used containers for over a decade but many of them switched to VMs to provide more consistent performance. Although the concepts underlying containers such as namespaces are well understood [34], container technology languished until the desire for rapid deployment led PaaS providers to adopt and standardize it, leading to a renaissance in the use of containers to provide isolation and resource control. Linux is the preferred operating system for the cloud due to its zero price, large ecosystem, good hardware support, good performance, and reliability. The kernel namespaces feature needed to implement containers in Linux has only become mature in the last few years since it was first discussed [17]. Within the last two years, Docker [45] has emerged as a standard runtime, image format, and build system for Linux containers.

This paper looks at two different ways of achieving resource control today, viz., containers and virtual machines and compares the performance of a set of workloads in both environments to that of natively executing the workload on hardware. In addition to a set of benchmarks that stress different aspects such as compute, memory bandwidth, memory latency, network bandwidth, and I/O bandwidth, we also explore the performance of two real applications, viz., Redis and MySQL on the different environments.

Our goal is to isolate and understand the overhead introduced by virtual machines (specifically KVM) and containers (specifically Docker) relative to non-virtualized Linux. We expect other hypervisors such as Xen, VMware ESX, and Microsoft Hyper-V to provide similar performance to KVM given that they use the same hardware acceleration features. Likewise, other container tools should have equal performance to Docker when they use the same mechanisms. We do not evaluate the case of containers running inside VMs or VMs running inside containers because we consider such double virtualization to be redundant (at least from a performance perspective). The fact that Linux can host both VMs and containers creates the opportunity for an apples-to-apples comparison between the two technologies with fewer confounding variables than many previous comparisons.

We make the following contributions:

- We provide an up-to-date comparison of native, container, and virtual machine environments using recent hardware and software across a cross-section of interesting benchmarks and workloads that are relevant to the cloud.
- We identify the primary performance impact of current virtualization options for HPC and server workloads.
- We elaborate on a number of non-obvious practical issues that affect virtualization performance.
- We show that containers are viable even at the scale of an entire server with minimal performance impact.

The rest of the paper is organized as follows. Section II describes Docker and KVM, providing necessary background to understanding the remainder of the paper. Section III describes and evaluates different workloads on the three environments. We review related work in Section IV, and finally, Section V concludes the paper.

Zusammenfas

| an ato

"In general, Docker equals or exceeds KVM performance in every case we tested. [...]

Even using the fastest available forms of paravirtualization, KVM still adds some overhead to every I/O operation [...].

Thus, KVM is less suitable for workloads that are latency-sensitive or have high I/O rates.

Container vs. bare-metal:
Although **containers themselves have almost no overhead**, **Docker is not without performance gotchas**. **Docker volumes** have noticeably better performance than files stored in AUFS. **Docker's NAT also introduces overhead** for work- loads with high packet rates.
These features represent a **tradeoff between ease of management and performance** and should be considered on a case-by-case basis.

Source: Felter et al. (2014). An updated performance comparison of virtual machines and linux containers.

```
eduroam0
Hello Wo

real
user
sys
eduroam0
Hello Wo

real
user
sys
eduroam0
```

Source: http://cdn.meme.am/instances/53646903.jpg

„Containers do not contain“

Zitate
aka „Meinungen“

"Some people make the mistake of thinking of containers as a better and faster way of running virtual machines.
From a security point of view, containers are much weaker."

Dan Walsh,
SELinux architect

"Virtual Machines might be more secure today, but containers are definitely catching up."

Jerome Petazzoni,
Senior Software Engineer at Docker

"You are absolutely deluded, if not stupid, if you think that a worldwide collection of software engineers who can't write operating systems or applications without security holes, can then turn around and suddenly write virtualization layers without security holes."

Theo de Raadt,
OpenBSD project lead

"Docker's security status is best described as "it's complicated"."

Jerome Petazzoni,
Senior Software Engineer at Docker

Erbsenzählerei

Virtualization CVEs

Some Free Software VM hosting technologies
Vulnerabilities published in 2014

	Xen PV	KVM+ QEMU	Linux as general container	Linux app container (non–root)
Privilege escalation (guest–to–host)	0	3–5	7–9	4
Denial of service (by guest of host)	3	5–7	12	3
Information leak (from host to guest)	1	0	1	1

Hosts only application, not guest OS

Source: Surviving the Zombie Apocalyse - Ian Jackson
http://xenbits.xen.org/people/iwj/2015/fosdem-security/

Schlechter Ruf?


2
Schlechter Ruf?

Docker Shocker

Vulnerability Matrix

Simple table outlining vulnerability to **this particular** exploit. PRs welcome!

Docker Version	Docker Host OS	Vulnerable?
0.8.1	Ubuntu 12.04 LTS	Yes
0.10.0	Ubuntu 12.04 LTS	Yes
0.11.0	Ubuntu 12.04 LTS	Yes
0.11.1	CoreOS v324.2.0	Yes
0.11.1	Ubuntu 12.04 LTS	Yes
0.12.0	Ubuntu 12.04 LTS	No
1.0	Boot2Docker	No
1.0	CoreOS v343.0.0+	No
1.0	Ubuntu 12.04 LTS	No

Examples

Confirmed vulnerable: Docker 0.11.1 running Ubuntu

```
root@precise64:~# docker version
Client version: 0.11.1
Client API version: 1.11
Go version (client): go1.2.1
```

GitHub, Inc.

```
root@precise64:~# docker run gabrtv/shocker
[***] docker VMM-container breakout Po(C) 2014             [***]
[***] The tea from the 90's kicks your sekurity again.     [***]
[***] If you have pending sec consulting, I'll happily     [***]
[***] forward to my friends who drink secury-tea too!      [***]
[*] Resolving 'etc/shadow'
[*] Found vmlinuz
[*] Found vagrant
[*] Found lib64
[*] Found usr
[*] Found ...
[*] Found etc
[+] Match: etc ino=3932161
[*] Brute forcing remaining 32bit. This can take a while...
[*] (etc) Trying: 0x00000000
[*] #=8, 1, char nh[] = {0x01, 0x00, 0x3c, 0x00, 0x00, 0x00, 0x00, 0x00};
[*] Resolving 'shadow'
[*] Found timezone
[*] Found cron.hourly
...
[*] Found skel
[*] Found shadow
[+] Match: shadow ino=3935729
[*] Brute forcing remaining 32bit. This can take a while...
[*] (shadow) Trying: 0x00000000
[*] #=8, 1, char nh[] = {0xf1, 0x0d, 0x3c, 0x00, 0x00, 0x00, 0x00, 0x00};
[!] Got a final handle!
[*] #=8, 1, char nh[] = {0xf1, 0x0d, 0x3c, 0x00, 0x00, 0x00, 0x00, 0x00};
[!] Win! /etc/shadow output follows:
root:!:15597:0:99999:7:::
daemon:*:15597:0:99999:7:::
bin:*:15597:0:99999:7:::
sys:*:15597:0:99999:7:::
sync:*:15597:0:99999:7:::
games:*:15597:0:99999:7:::
[illegible]15597:0:99999:7:::
```

shocker.c

~/Documents/WORK/2017-talks/03_SC-ContainerSecurity/MA.../Literatur/shocker.c (no function selected)

```
/* shocker: docker PoC VMM-container breakout (C) 2014 Sebastian Krahmer
 *
 * Demonstrates that any given docker image someone is asking
 * you to run in your docker setup can access ANY file on your host,
 * e.g. dumping hosts /etc/shadow or other sensitive info, compromising
 * security of the host and any other docker VM's on it.
 *
 * docker using container based VMM: Sebarate pid and net namespace,
 * stripped caps and RO bind mounts into container's /. However
 * as its only a bind-mount the fs struct from the task is shared
 * with the host which allows to open files by file handles
 * (open_by_handle_at()). As we thankfully have dac_override and
 * dac_read_search we can do this. The handle is usually a 64bit
 * string with 32bit inodenumber inside (tested with ext4).
 * Inode of / is always 2, so we have a starting point to walk
 * the FS path and brute force the remaining 32bit until we find the
 * desired file (It's probably easier, depending on the fhandle export
 * function used for the FS in question: it could be a parent inode# or
 * the inode generation which can be obtained via an ioctl).
 * [In practise the remaining 32bit are all 0 :]
 *
 * tested with docker 0.11 busybox demo image on a 3.11 kernel:
 *
 * docker run -i busybox sh
 *
 * seems to run any program inside VMM with UID 0 (some caps stripped); if
 * user argument is given, the provided docker image still
 * could contain +s binaries, just as demo busybox image does.
 *
 * PS: You should also seccomp kexec() syscall :)
 * PPS: Might affect other container based compartments too
 *
 * $ cc -Wall -std=c99 -O2 shocker.c -static
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
#include <stdint.h>


struct my_file_handle {
    unsigned int handle_bytes;
    int handle_type;
    unsigned char f_handle[8];
};
```

L: 1 C: 1 ANSI C Unicode (UTF-8) Unix (LF) Saved: 08.03.17, 19:12:55 5.115 / 732 /

Source: https://github.com/gabrtv/shocker

Missverständnisse...

Jessie Frazelle's Blog

Docker Containers on the Desktop

Sources: https://blog.jessfraz.com/posts/docker-containers-on-the-desktop.html
https://news.ycombinator.com/item?id=9086751

news.ycombinator.com

Hacker News new | comments | show | ask | jobs | submit login

▲ Docker containers on the desktop (jessfraz.com)

267 points by julien421 744 days ago | hide | past | web | 74 comments | favorite

▲ alexlarsson 743 days ago [-]

This is not sandboxing. Quite the opposite, this gives the apps root access:

First of all, X11 is completely unsecure, the "sandboxed" app has full access to every other X11 client. Thus, its very easy to write a simple X app that looks for say a terminal window and injects key events (say using Xtest extension) in it to type whatever it wants. Here is another example that sniffs the key events, including when you unlock the lock screen: https://github.com/magcius/keylog

Secondly, if you have docker access you have root access. You can easily run something like:

```
docker run -v /:/tmp ubuntu rm -rf /tmp/*
```

Which will remove all the files on your system.

▲ jdub 743 days ago [-]

Just so everyone knows, this is Alex "I have a weird interest in application bundling systems" Larsson, who is doing some badass bleeding edge work on full on sandboxed desktop applications on Linux. :-)

http://blogs.gnome.org/alexl/2015/02/17/first-fully-sandboxe...

http://www.youtube.com/watch?v=t-2a_XYJPEY

Like Ron Burgundy, he's... "kind of a big deal".

(Suffer the compliments, Alex.)

▲ Iv 743 days ago [-]

Yes, I think that it is important to make this point around as docker gains popularity: security is not part of their original design. The problem they apparently wanted to solve initially is the ability for a linux binary to run, whatever its dependencies are, on any system.

It does try to keep containers separated but it does not enforce thatthrough a particularly strong mechanism.

blog.jessfraz.com

ssie Frazelle's Blog

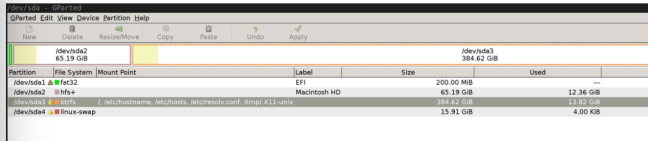
7. Gparted

Dockerfile

Partition your device in a container.

MIND BLOWN.

```
$ docker run -it \
    -v /tmp/.X11-unix:/tmp/.X11-unix \ # mount the X11 socket
    -e DISPLAY=unix$DISPLAY \ # pass the display
    --device /dev/sda:/dev/sda \ # mount the device to partition
    --name gparted \
    jess/gparted
```



Bisschen weiter gespielt...

Das ganze dann noch in einer NFS-Umgebung:

```
[badguy@docker ~]# cd /home/goodguy
bash: cd: /home/goodguy: Permission denied


[badguy@docker ~]# id badguy && id goodguy
uid=1234(badguy) gid=1234(badguy) groups=1234(badguy),1337(docker)
uid=1000(goodguy) gid=1000(goodguy) groups=1000(goodguy)


[badguy@docker ~]# docker run -it -v /home:/nfs3home -u 1000 busybox sh
/ $ id
uid=1000 gid=0(root)
/ $ touch /nfs3home/goodguy/badguy_WAS_HERE && exit
```

Source: http://cdn.meme.am/instances/53646903.jpg

Unpatched Vulnerabilities

Why so serious?

Source: https://docs.google.com/presentation/d/1toUKgqLyy1b-pZlDgxONLduiLmt2yaLR0GliBB7b3L0/

CVE-2015-0235
aka
GHOST

"GHOST is a buffer overflow bug affecting the gethostbyname() and gethostbyname2() function calls in the glibc library. This vulnerability allows a remote attacker that is able to make an application call to either of these functions to execute arbitrary code."

66.6 %

of analyzed images on Quay.io

Coincidence? I think not !

ORIGINAL PHOTO (CC) BY IAN KLUFT

Source: https://docs.google.com/presentation/d/1toUKgqLyy1b-pZlDgxONLduiLmt2yaLR0GliBB7b3L0/

CVE-2014-0160
aka
Heartbleed

"The TLS and DTLS implementations in OpenSSL do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read."

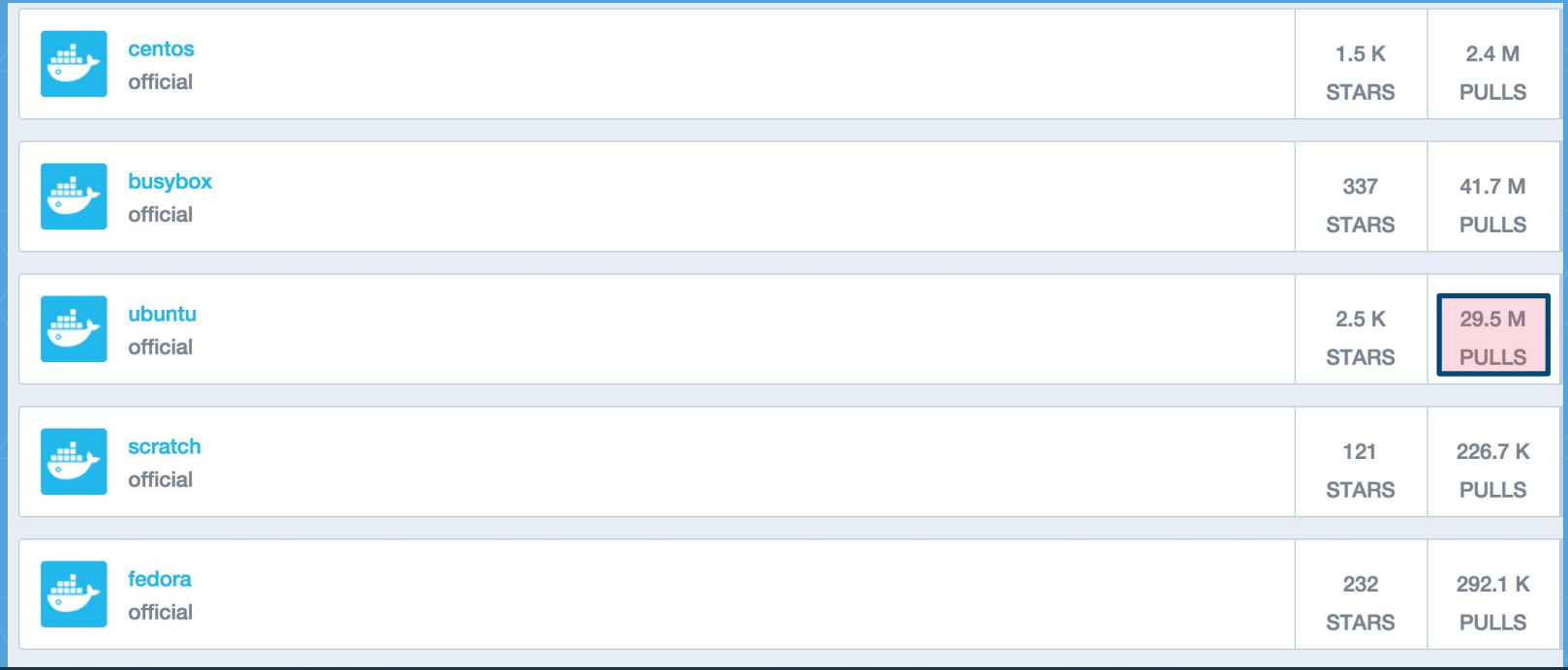
80 %

of analyzed images on Quay.io

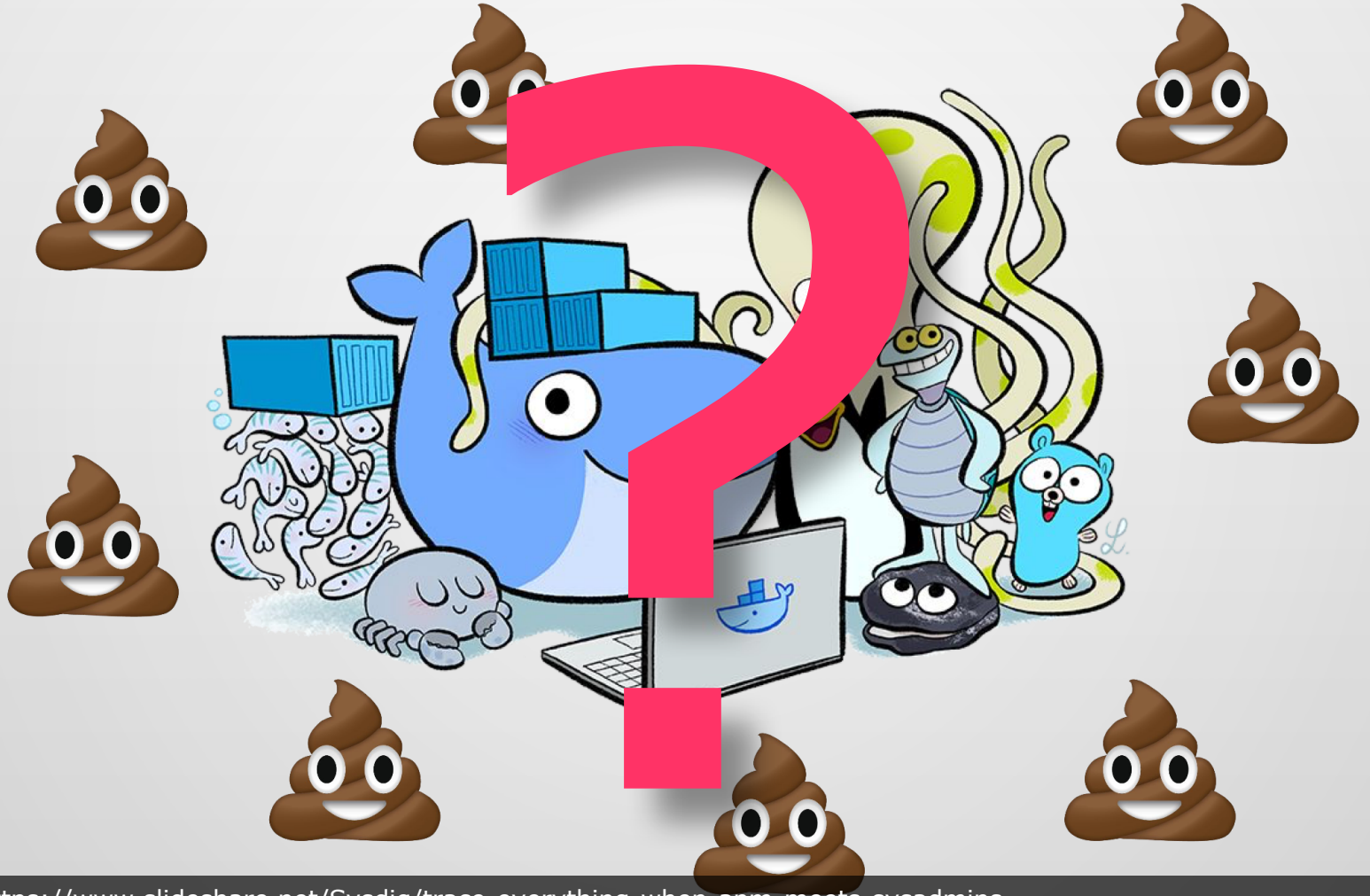
ORIGINAL PHOTO (CC) BY IAN KLUFT

Source: https://docs.google.com/presentation/d/1toUKgqLyy1b-pZlDgxONLduiLmt2yaLR0GliBB7b3L0/

Most containers built on same base layers



Source: https://docs.google.com/presentation/d/1toUKgqLyy1b-pZlDgxONLduiLmt2yaLR0GliBB7b3L0/



Source: https://www.slideshare.net/Sysdig/trace-everything-when-apm-meets-sysadmins

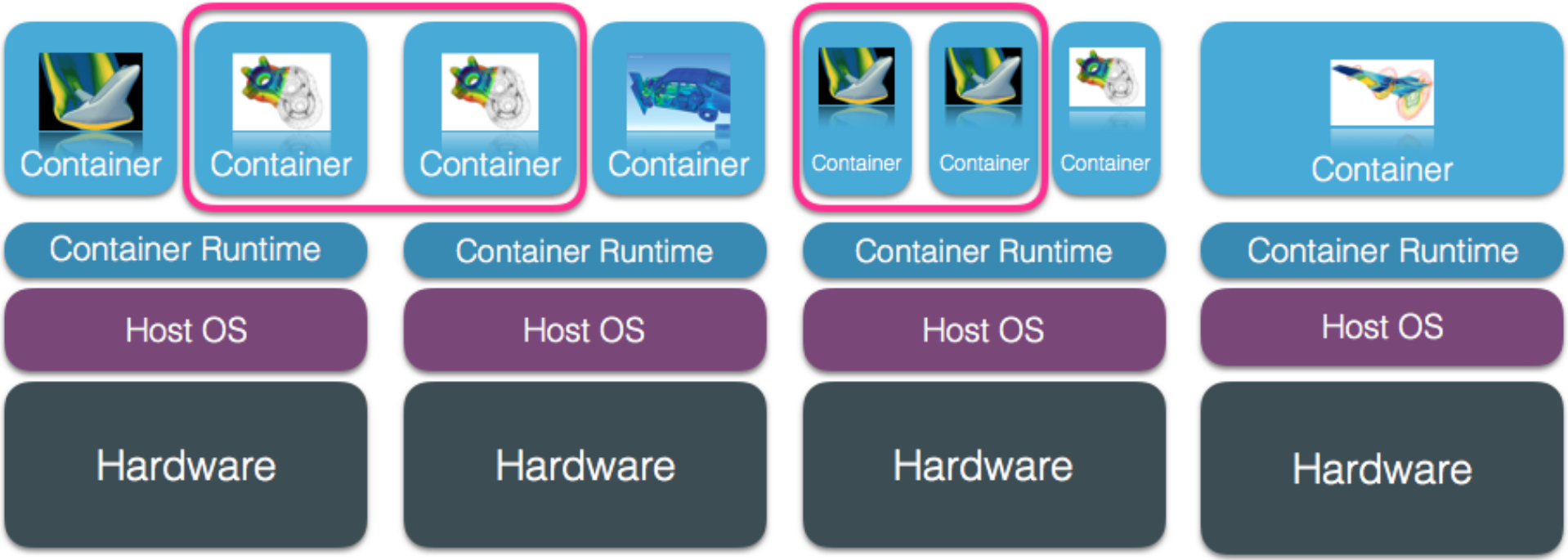

3
A Year in Docker Security...

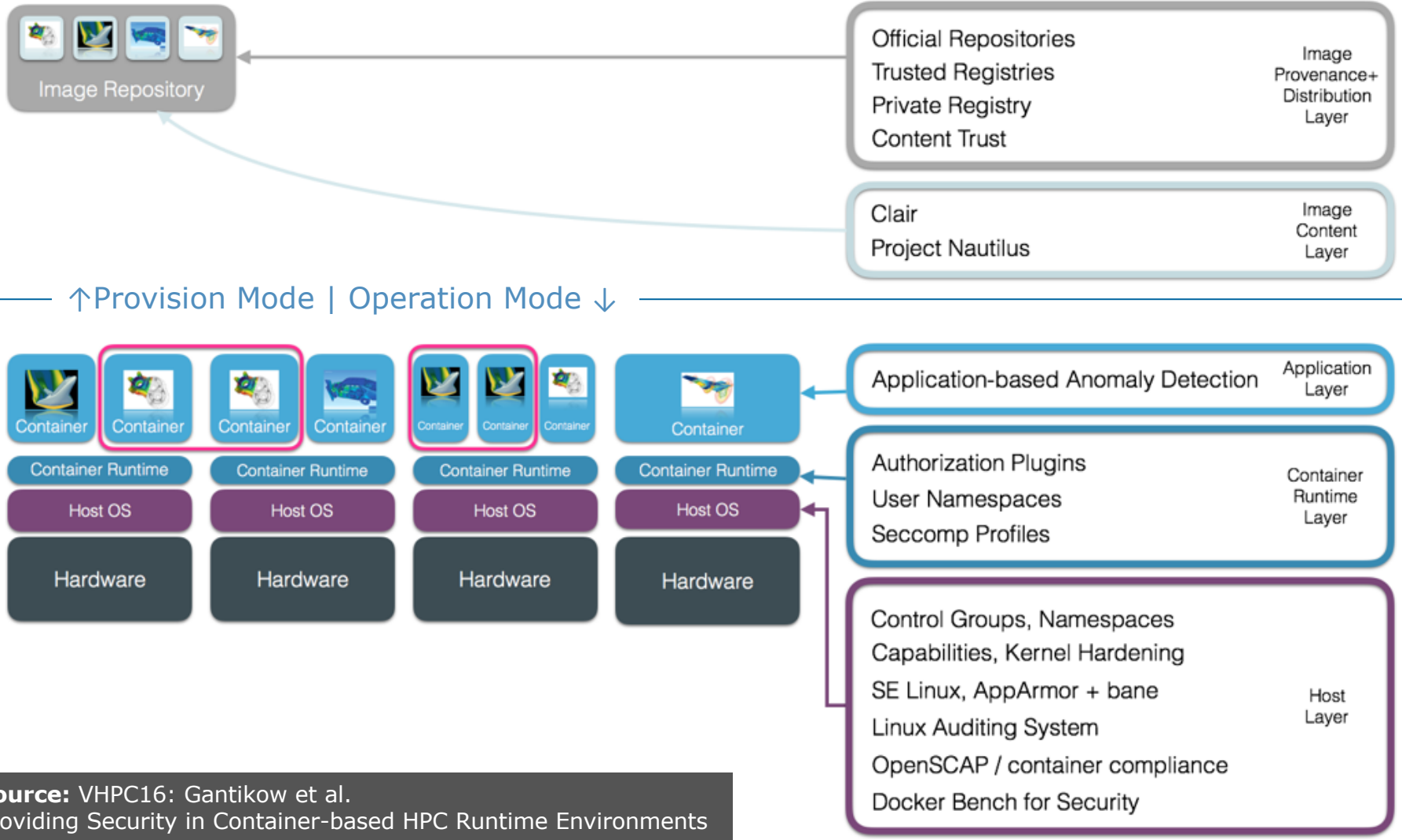
Defense in depth

multiple layers of security controls

„secure platform, secure access, secure content"

Ausgangspunkt





Source: VHPC16: Gantikow et al.
Providing Security in Container-based HPC Runtime Environments

Provision Mode

Image Provenance + Distribution

Image Provenance + Distribution

- **Tools and Technologies**
  - Official Repositories (-> *)
  - Trusted Registries (on premises)
  - Content Trust (image signing + verification)
  - Docker Store (new, fully „compliant, commercially supported software“)
  - Private Registry

- **Recommendations**
  - Build, sign and maintain your own (base) images
  - Use a private repository with „curated“ images
  - When relying on DockerHub: limit to official repositories

Image Content

Clair

Clair

- Aus dem **CoreOS-Projekt**, OpenSource – Apache 2.0 Lizenz

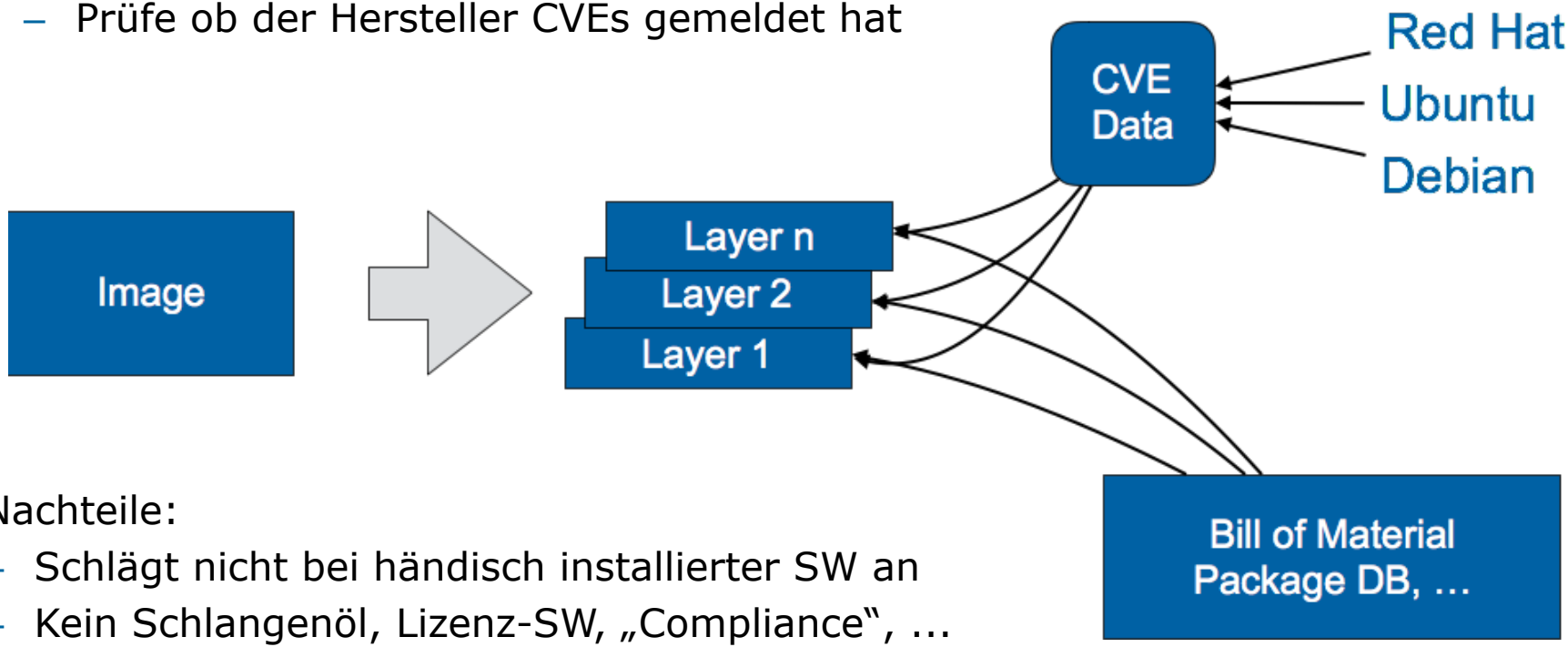
- Integriert in die **Registry Quay.io**
  - Prüft dort jedes neue Image
  - Prüft bestehende Images regelmäßig auf neu gemeldete Schwachstellen

- **Alternativen** (kommerziell):
  - Project Nautilus aka „Docker Security Scanning“
  - OpenShift: Red Hat CloudForms mit OpenSCAP Image Scans
  - IBM Bluemix (Vulnerability Advisor?)

  - Konzept ähnlich – oft mehr Features

Funktionsweise

- Vorgehen – für alle Schichten eines Images:
  - Prüfe ob der Hersteller CVEs gemeldet hat



- Nachteile:
  - Schlägt nicht bei händisch installierter SW an
  - Kein Schlangenöl, Lizenz-SW, „Compliance“, ...

Operation Mode

Host

Control Groups + Namespaces

Reminder: *Namespaces + cgroups*

- Essentiellste Features
  - **Namespaces** : *isolierter Betrieb*
  - **Cgroups**: Limitierung Ressourcenverbrauch

Namespace	Description	Controller	Description
pid	Process ID	**blkio**	Access to block devices
net	Network Interfaces, Routing Tables, …	**cpu**	CPU time
ipc	Semaphores, Shared Memory, Message Queues	**devices**	Device access
mnt	Root and Filesystem Mounts	**memory**	Memory usage
uts	Hostname, Domainname	**net_cls**	Packet classification
user	UserID and GroupID	**net_prio**	Packet priority

Capabilities + Kernel Hardening

Capabilities + Kernel Hardening

- **Capabilities** : sehr grob, Add + Drop möglich – genau wissen was man tut ;)
- Übersicht über Capabilities
  - http://man7.org/linux/man-pages/man7/capabilities.7.html

Capatibilies auch aktivierbar – hier Hostname setzen

```
$ docker run –rm –ti busybox sh
/ # hostname foo
hostname: sethostname: Operation not permitted
$ docker run –rm –ti —cap-add=SYS_ADMIN busybox sh
/ # hostname foo<hostname changed>
```

- **Kernel Hardening**: möglich hier zu Patchen, ggf Support-Konflikt
  - Grsecurity, PaX – Ausnutzung von Buffer Overflows reduzieren, ...
  - Im HPC Bereich nicht so gerne gesehen, da ggf Code zur Laufzeit generiert

Mandatory Access Control Systems
SELinux, AppArmor + bane

PROJECT ATOMIC

Get Started Documentati

Google+ RSS

Docker and SELinu

The interaction between SE
concerns: protection of the
another.

SELinux Labels for Doc

SELinux labels consist of 4 p

```
User:Role:Type:level.
```

SELinux controls access to p
forms of SELinux protection
(MCS) separation.

Type Enforcement

Type enforcement is a kind
process type. It works in the
container process is svirt_lx
all files types under /usr and
permitted to use the networ

„www.projectatomic.io/blog/feed.xml" in neuem Tab

docs.docker.com/engine/security/apparmor/

- Use trusted images
- AppArmor security profiles for Docker
- Seccomp security profiles for Docker
- Extend Engine
- Dockerize an application
- Engine reference
- Migrate to Engine 1.10
- Breaking changes
- Deprecated Engine Features
- FAQ
- Docker Swarm
- Docker Compose
- Docker Hub
- CS Docker Engine
- Universal Control Plane
- Docker Trusted Registry
- Docker Cloud

Understand the p

The `docker-default` profile is the defa
is moderately protective while providing
compatibility. The profile is the following:

```
#include <tunables/global>


profile docker-default flags=(attac

  #include <abstractions/base>


  network,
  capability,
  file,
  umount,

  deny @{PROC}/{*,**^[0-9*],sys/ker
  deny @{PROC}/sysrq-trigger rwklx,
  deny @{PROC}/mem rwklx,
  deny @{PROC}/kmem rwklx,
  deny @{PROC}/kcore rwklx,

  deny mount,

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

GitHub - jfrazelle/bane: Cu

GitHub, Inc. [US] https://github.com/jfrazelle/bane

> "Reviewing AppArmor profile pull requests is the *bane* of my existence"
>
> • Jess Frazelle

AppArmor Policy auswählen

```
$ docker run --rm -it --security-opt
apparmor=docker-default/or-my-policy
hello-world
```

```
-d    run in debug mode
-profile-dir string
      directory for saving the profiles (default "/etc/apparmor.d/containers")
-v    print version and exit (shorthand)
-version
      print version and exit
```

Sources: http://www.projectatomic.io/docs/docker-and-selinux/
https://docs.docker.com/engine/security/apparmor/#understand-the-policies
https://github.com/jessfraz/bane

Linux Auditing System

Linux Auditing System (auditd)

- **Zugriffsüberwachungssystem (!Enforcement)**
- Logger für SELinux
- Regeln auf Basis von Dateien und Syscalls

```
# monitor unlink() and rmdir() system calls.
-a exit,always -S unlink -S rmdir
# monitor open() system call by UID 4711.
-a exit,always -S open -F loginuid=4711
# monitor write-access and change in file properties (r/w/x) of the these files.
-w /etc/passwd -p wa
```

- **Einsatz**: Missbrauch und unauthorisierte Aktivitäten, Loggen von:
  - Docker related activities (Containerstart, Änderung Config, Zertifikate, Keys,...)
  - Erweiterung bestehender Audits + Integration in Monitoring

OBSOLET!
Jetzt Teil von OpenSCAP „oscap-docker“

OpenSCAP/Container Compliance

GitHub, Inc.

Scanning Docker image using OpenSCAP

Run any OpenSCAP command within chroot of mounted docker image.

```
# oscap-docker image IMAGE_NAME [OSCAP_ARGUMENTS]
```

Learn more about OSCAP_ARGUMENTS in `man oscap`.

Exemplary usage

Tested on Fedora host.

```
# yum install scap-security-guide openscap-scanner docker-io
# sed -i 's/<platform idref=.*$//g' /usr/share/xml/scap/ssg/fedora/ssg-fedora-ds.xml
# service docker start
# docker pull fedora
# oscap-docker image fedora xccdf eval \
    --profile xccdf_org.ssgproject.content_profile_common \
    /usr/share/xml/scap/ssg/fedora/ssg-fedora-ds.xml
```

Scanning Docker container

Run OpenSCAP scan within chroot of running docker container. This may differ from scanning docker image due to defined mount points.

```
# oscap-docker container CONTAINER_NAME [OSCAP_ARGUMENTS]
```

Vulnerability scan of Docker container

Link: https://github.com/OpenSCAP/container-compliance

Docker Bench for Security

Start des Benchmarks

```
docker run -it --net host --pid host --cap-add
audit_control -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
```

```
# ------------------------------------
# Docker Bench for Security v1.0.0
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-pra
# Inspired by the CIS Docker 1.11 Benc
# https://benchmarks.cisecurity.org/do
# ------------------------------------

Initializing Sat Apr 30 23:04:50 CEST


[INFO] 1 - Host Configuration
[WARN] 1.1  - Create a separate partit
[PASS] 1.2  - Use an updated Linux Ker
[PASS] 1.4  - Remove all non-essential services from the host - Network
[PASS] 1.5  - Keep Docker up to date
[INFO]       * Using 1.12.0 which is current as of 2016-04-27
[INFO]       * Check with your operating system vendor for support and security maintenance for docker
[INFO] 1.6  - Only allow trusted users to control Docker daemon
[INFO]       * docker:x:999:tsj
[PASS] 1.7  - Audit docker daemon - /usr/bin/docker
[PASS] 1.8  - Audit Docker files and directories - /var/lib/docker
[PASS] 1.9  - Audit Docker files and directories - /etc/docker
[PASS] 1.10 - Audit Docker files and directories - docker.service
[PASS] 1.11 - Audit Docker files and directories - docker.socket
[PASS] 1.12 - Audit Docker files and directories - /etc/default/docker
[INFO] 1.13 - Audit Docker files and directories - /etc/docker/daemon.json
[INFO]       * File not found
[PASS] 1.14 - Audit Docker files and directories - /usr/bin/docker-containerd
[PASS] 1.15 - Audit Docker files and directories - /usr/bin/docker-runc


[INFO] 2 - Docker Daemon Configuration
[PASS] 2.1  - Restrict network traffic between containers
[PASS] 2.2  - Set the logging level
[PASS] 2.3  - Allow Docker to make changes to iptables
[PASS] 2.4  - Do not use insecure registries
[PASS] 2.5  - Do not use the aufs storage driver
[INFO] 2.6  - Configure TLS authentication for Docker daemon
[INFO]       * Docker daemon not listening on TCP
[INFO] 2.7 - Set default ulimit as appropriate
[INFO]       * Default ulimit doesn't appear to be set
```

Link: https://github.com/docker/docker-bench-security

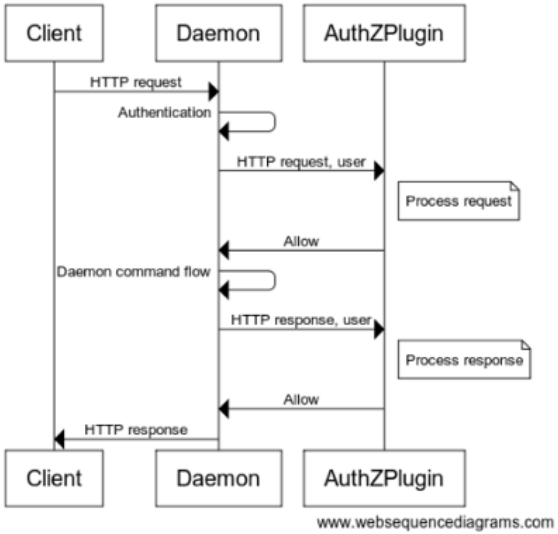
Container Runtime

Ab Docker 1.10

Authorization Plugins

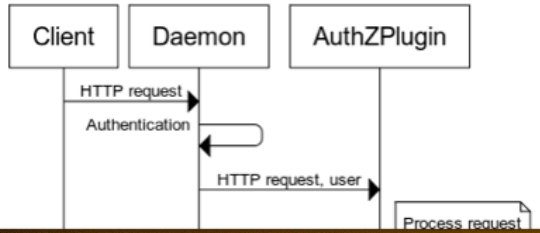
docs.docker.com/engine/extend/plugins_authc

Authorization allow scenario



www.websequencediagrams.com

Authorization deny scenario



GitHub, Inc. github.com/twistlock/authz

Examples

Below are some examples for basic policy scenarios:

1. Alice can run all Docker commands: `{"name":"policy_1","users":["alice"],"actions":[""]}`
2. All users can run all Docker commands: `{"name":"policy_2","users":[""],"actions":[""]}`
3. Alice and Bob can create new containers: `{"name":"policy_3","users":["alice","bob"],"actions":["container_create"]}`
4. Service account can read logs and run container top: `{"name":"policy_4","users":["service_account"],"actions":["container_logs","container_top"]}`
5. Alice can perform anything on containers: `{"name":"policy_5","users":["alice"],"actions":["container"]}`
6. Alice can only perform get operations on containers: `{"name":"policy_5","users":["alice"],"actions":["container"],"readonly":true }`

Installing the plugin

The authorization plugin can run as a container application or as a host service.

Running inside a container

1. Install the containerized version of the Twistlock authorization plugin:

```
$ docker run -d  --restart=always -v /var/lib/authz-broker/policy.json:/var/lib/authz-broker/policy.json -v /run,
```

```
For auditing using syslog hook add the following settings to the docker command:<code>-e AUDITOR-HOOK:syslog -v /
For auditing using file add the following settings to the docker command:<code>-e AUDITOR-HOOK:file -v PATH_TO_LO
```

Link: https://github.com/twistlock/authz

Ab Docker 1.10

User Namespaces

“Phase 1” Usage Overview

```
# docker daemon --root=2000:2000 ...
drwxr-xr-x root:root  /var/lib/docker
drwx------ 2000:2000  /var/lib/docker/2000.2000
```

Start the daemon with a remapped root setting (in this case uid/gid = 2000/2000)

```
$ docker run -ti --name fred --rm busybox /bin/sh
/ # id
uid=0(root) gid=0(root) groups=10(wheel)
```

Start a container and verify that inside the container the uid/gid map to root (0/0)

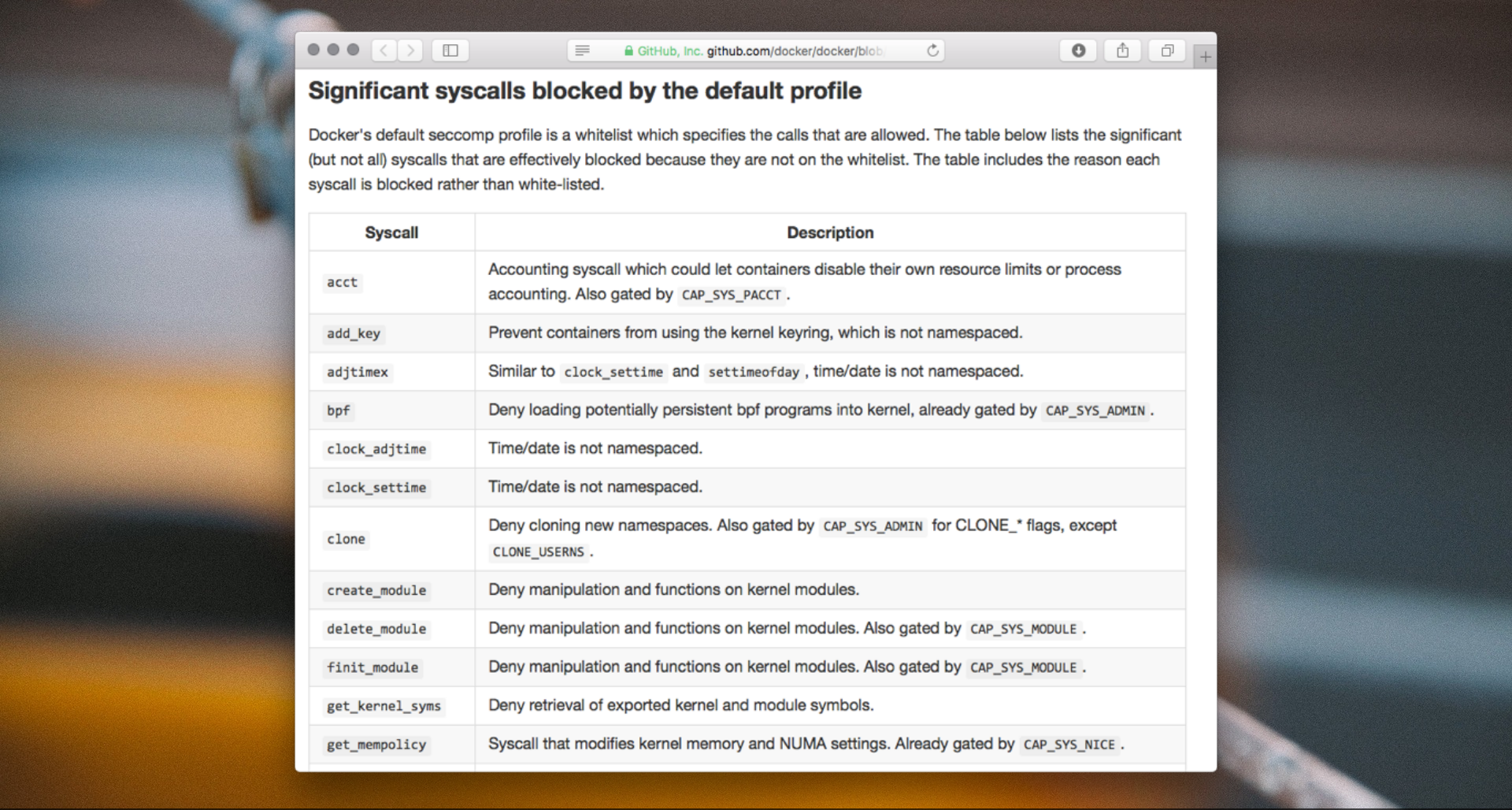
```
$ docker inspect -f ‘{{ .State.Pid }}’ fred
8851
$ ps -u 2000
  PID TTY          TIME CMD
 8851 pts/7    00:00:00 sh
```

You can verify that the container process (PID) is actually running as user 2000

Link: https://events.linuxfoundation.org/sites/events/files/slides/User Namespaces - ContainerCon 2015 - 16-9-final_0.pdf

Ab Docker 1.10

Seccomp Profiles



Significant syscalls blocked by the default profile

Docker's default seccomp profile is a whitelist which specifies the calls that are allowed. The table below lists the significant (but not all) syscalls that are effectively blocked because they are not on the whitelist. The table includes the reason each syscall is blocked rather than white-listed.

Syscall	Description
`acct`	Accounting syscall which could let containers disable their own resource limits or process accounting. Also gated by `CAP_SYS_PACCT`.
`add_key`	Prevent containers from using the kernel keyring, which is not namespaced.
`adjtimex`	Similar to `clock_settime` and `settimeofday`, time/date is not namespaced.
`bpf`	Deny loading potentially persistent bpf programs into kernel, already gated by `CAP_SYS_ADMIN`.
`clock_adjtime`	Time/date is not namespaced.
`clock_settime`	Time/date is not namespaced.
`clone`	Deny cloning new namespaces. Also gated by `CAP_SYS_ADMIN` for CLONE_* flags, except `CLONE_USERNS`.
`create_module`	Deny manipulation and functions on kernel modules.
`delete_module`	Deny manipulation and functions on kernel modules. Also gated by `CAP_SYS_MODULE`.
`finit_module`	Deny manipulation and functions on kernel modules. Also gated by `CAP_SYS_MODULE`.
`get_kernel_syms`	Deny retrieval of exported kernel and module symbols.
`get_mempolicy`	Syscall that modifies kernel memory and NUMA settings. Already gated by `CAP_SYS_NICE`.

Source: https://docs.docker.com/engine/security/seccomp/#significant-syscalls-blocked-by-the-default-profile

Und sonst noch?

docker run …

`--pid-limit:` PID limitations per container, prevent fork-bombs

`--security-opt=no-new-privileges:` prevent privilege escalation

`--readonly:` Container / RO, for immutable container images

Securing the infrastructure and the workloads of linux containers

Massimiliano Mattetti*, Alexandra Shulman-Peleg†, Yair Allouche†, Antonio Corradi*, Shlomi Dolev‡, Luca Foschini*
* CIRI ICT, University of Bologna
† IBM Cyber Security Center of Excellence
‡ Ben-Gurion University

***Abstract*—One of the central building blocks of cloud platforms are linux containers which simplify the deployment and management of applications for scalability. However, they introduce new risks by allowing attacks on shared resources such as the file system, network and kernel. Existing security hardening mechanisms protect specific applications and are not designed to protect entire environments as those inside the containers. To address these, we present a LiCShield framework for securing of linux containers and their workloads via automatic construction of rules describing the expected activities of containers spawned from a given image. Specifically, given an image of interest LiCShield traces its execution and generates profiles of kernel security modules restricting the containers' capabilities. We distinguish between the operations on the linux host and the ones inside the container to provide the following protection mechanisms: (1) Increased host protection, by restricting the operations done by containers and container management daemon only to those observed in a testing environment; (2) Narrow container operations, by tightening the internal dynamic and noisy environments, without paying the high performance overhead of their on-line monitoring. Our experimental results show that this approach is efficient to prevent known attacks, while having almost no overhead on the production environment. We present our methodology and its technological insights and provide recommendations regarding its efficient deployment with intrusion detection tools to achieve both optimized performance and increased protection. The code of the LiCShield framework as well as the presented experimental results are freely available for use at https://github.com/LinuxContainerSecurity/LiCShield.git.**

1 Introduction

Shifting away from traditional on-premises computing, cloud environments allow to reduce costs via efficient utilization of servers hosting multiple customers over the same shared pools of resources. Linux containers are a disruptive technology enabling better server utilization together with simplified deployment and management of applications. Linux containers provide a lightweight operating system level virtualization via grouping resources like processes, files, and devices into isolated spaces that give you the appearance of having your own machine with near native performance and no additional virtualization overheads. When comparing between containers and VMs (in terms of CPU, memory, storage and networking resources), containers exhibited better or equal results than VM in almost all cases [24]. Furthermore, container management software, such as the Docker[1] technology [18], enable an easy packaging and deployment of applications, supporting the DevOps model of speeding up the development life-cycle through rapid change, from prototype to production [29], [34]. As a result, linux containers became widely adopted across all of the cloud layers such as Infrastructure as a service (IaaS), where they allow achieving near-native performance and Platform as a service (PaaS), linux containers are used as deployment packages allowing easy on-boarding of applications (e.g. CloudFoundry [11]).

Container threats and protection mechanisms. While optimizing the speed of deployment, linux containers were not designed as a security mechanism to isolate between untrusted and potentially malicious containers. They lack the extra layer of virtualization and thus, are less secure than VMs [2], [1]. Their vulnerabilities range from kernel exploits and attacks on the shared linux host resources to misconfigurations, side channels and data leakage [20]. Thus, container security is considered an obstacle for an even wider adoption of containerization technologies [4]. There are two main types of protection mechanisms that can be applied to container environments: security hardening mechanisms (e.g., AppArmor [16] and SELinux [8]) and host based intrusion detection systems. However, applying both mechanisms to container environments is not straightforward due to several reasons. First, there are limitations in properly deploying them in container environments where part of the workload is executed on the host and part inside the container, in which case multiple processes and applications should be grouped and protected together. Second, their practical application to the noisy container environments (see Section 5) is not straightforward.

Our approach and contributions. We present the LiCShield framework for protection of **Li**nux **C**ontainers and their workloads. Given a container image of interest, we automatically construct the security profiles protecting its execution both on the linux host and within the container. We provide a tool-set to trace and analyze containers' executions, separating the traces on the host and inside the containers. We automatically construct AppArmor rules for two different

[1]Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries. Docker, Inc. and other parties may also have trademark rights in other terms used herein.

978-1-4673-7676-3/15/$31.00 ©2015 IEEE

LiCShield

Host OS	...tem, volumes, memory ... working		
Container Engine	Vulnerabilities in the container engine (running as root) or the libraries loaded by it	Host and containers	CVEs at [14], ... loaded for compression [13])
Shared Bin/Libs	Loading malicious modules	Containers	Loading a malicious shared object */usr/lib/libnginx.so* [27]
Applications	Cross-container leakage	Containers	One container can access the packets of another container via ARP spoofing [36]

TABLE I
EXAMPLE OF ATTACK ON THE COMPONENTS OF CONTAINER ENVIRONMENTS DEPICTED IN FIGURE 1. ADDITIONAL EXAMPLES CAN BE FOUND AT [12], [14], [20]

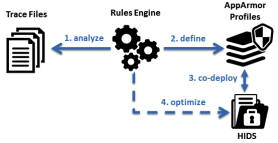


Fig. 2. **Approach Overview.**

[13]. The profiles generated by LiCShield overcome these limits by providing a fine-grained control over the containers and protection against possible vulnerabilities of the container management tools such as Docker daemon.

3 LiCShield Approach

Our main goal is to improve the security of cloud servers executing linux containers, without requiring any significant changes to the code of cloud platforms, linux distributions or the container management software, automating the workflow that can be applied without requiring any other intervention.

Figure 2 provides an overview of the LiCShield architecture consisting of the following stages:

1) *Trace and analyze:* LiCShield traces the container creation and execution in a synthetic testing environment, collecting the information about the performed operations, their resources and required permissions.
2) *Define rules:* The traces are processed to create rules that are used for two purposes: first to generate improved profiles for linux kernel security modules, such as AppArmor, restricting the containers' capabilities; second to generate rules that can be used to improve the intrusion detection systems, by automatically feeding the categories describing normal activities.
3) *Co-deploy:* We advocate that there is a need to differentiate between the protection of the host and the container workloads. For the critical host protection, we suggest to co-deploy LiCShield with HIDS, to achieved higher levels

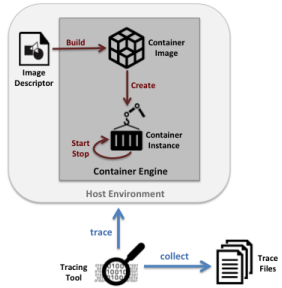


Fig. 3. **Flow Overview.**

of security. At the same time, we suggest that noisy, low risk components can be protected only by LiCShield.
4) *Optimize:* LiCShield rules can be used to optimize the learning phase of intrusion detection systems, by providing the description of the expected activities. This has several benefits: first, reducing the number of false positive alerts; second, optimizing the setup and learning phase. Collecting the information on a per-image basis in pre-production with LiCShield, saves the overhead of learning the execution of each of containers spawned from the same image in the production setup.

4 LiCShield Description

Figure 3 shows the first step of the profile generation process, that we call the tracing phase. In this stage LiCShield takes a Dockerfile as input, starts the Docker daemon, sends to it commands using its REST API, and records their execution. Specifically, it first builds a new container image from the Dockerfile and then runs this image in a new container, while tracing the execution. Below we detail the main mechanisms of LiCShield which include: (1) Tracing the kernel operations;

Source: Mattetti, M., Shulman-Peleg, A., Allouche, Y., Corradi, A., Dolev, S., & Foschini, L. (2015). Securing the infrastructure and the workloads of linux containers.

Application

Viele Möglichkeiten, wenig fertiges

Anomaly Detection

Applying Bag of System Calls for Anomalous Behavior Detection of Applications in Linux Containers

Amr S. Abed
Department of Electrical & Computer Engineering
Virginia Tech, Blacksburg, VA
amrabed@vt.edu

T. Charles Clancy, David S. Levy
Hume Center for National Security & Technology
Virginia Tech, Arlington, VA
{tcc, dslevy}@vt.edu

***Abstract*—In this paper, we present the results of using bags of system calls for learning the behavior of Linux containers for use in anomaly-detection based intrusion detection system. By using system calls of the containers monitored from the host kernel for anomaly detection, the system does not require any prior knowledge of the container nature, neither does it require altering the container or the host kernel.**

I. Introduction

Linux containers are computing environments apportioned and managed by a host kernel. Each container typically runs a single application that is isolated from the rest of the operating system. A Linux container provides a runtime environment for applications and individual collections of binaries and required libraries. Namespaces are used to assign customized views, or permissions, applicable to its needed resource environment. Linux containers typically communicate with the host kernel via system calls.

By monitoring the system calls between the container and the host kernel, one can learn the behavior of the container in order to detect any change of behavior, which may reflect an intrusion attempt against the container.

One of the basic approaches to anomaly detection using system calls is the *Bag of System Calls* (BoSC) technique. The BoSC technique is a frequency-based anomaly detection technique, that was first introduced by Kang et al. in 2005 [1]. Kang et al. define the bag of system call as an ordered list $< c_1, c_2, \ldots, c_n >$, where n is the total number of distinct system calls, and c_i is the number of occurrences of the system call, s_i, in the given input sequence. BoSC has been used for anomaly detection at the process level [1] and at the level of virtual machines (VMs) [2][3][4], and has shown promising results.

The fewer number of processes in a container, as compared to VM, results in reduced complexity. The reduced complexity gives the potential for the BoSC technique to have high detection accuracy with a marginal impact on system performance when applied to anomaly detection in containers.

In this paper, we study the feasibility of applying the BoSC to passively detect attacks against containers. The technique used is similar to the one introduced by [3]. We show that a frequency-based technique is sufficient for detecting abnormality in container behavior.

The rest of this paper is organized as follows. Section II provides an overview of the system. Section III describes the experimental design. Section IV discusses the results of the experiments. Section V gives a brief summary of related work. Section VI concludes with summary and future work.

II. System Overview

In this paper, we use a technique similar to the one described in [3] applied to Linux containers for intrusion detection. The technique combines the sliding window technique [5] with the bag of system calls technique [1] as described below.

The system employs a background service running on the host kernel to monitor system calls between any Docker containers and the host Kernel. Upon start of a container, the service uses the Linux `strace` tool to trace all system calls issued by the container to the host kernel. The `strace` command reports system calls with their originating process ID, arguments, and return values. A table of all distinct system calls in the trace is also reported at the end of the trace along with the total number of occurrences.

The full trace, and the count table, are stored into a log file that is processed offline and used to learn the container behavior after the container terminates. At this point, we are not performing any real-time behavior learning or anomaly detection. Therefore, dealing with the whole trace of the container offline is sufficient for our proof-of-concept purposes. However, for future purposes, where behavior learning and anomaly detection is to be achieved in real time (in which case the full trace would not be available), the learning algorithm applied would slightly differ from the one described here. However, the same underlying concepts will continue to apply.

The generated log file is then processed to create two files, namely syscall-list file and trace file. The syscall-list file holds a list of distinct system calls sorted by the number of occurrences. The trace file holds the full list of system calls as collected by `strace` after trimming off arguments, return values, and process IDs. The count file is used to create an

Anomaly Detection, Alerting, and Incident Response for Containers

GIAC GCIH Gold Certification

Author: Alex Borhani, r.alex.borhani@gmail.com
Advisor: Chris Walker
Accepted: February 19, 2017

Abstract

With the rapid adoption of containerized technologies to support the agile development and operations (DevOps) methodology, the necessity of formulating a comprehensive prevention, detection, and incident response (IR) security strategy in those environments is critical. Though various mechanisms exist to fulfill preventive strategies for containers, such as system hardening and continuously patching images, the need to implement similar levels of detection capabilities is also vital, particularly because many preventative security efforts are eventually neutralized or, worse yet, never implemented properly. By outlining the capabilities of several open source technologies, this paper will demonstrate the viability of detecting an anomaly, alerting on the presence of an anomaly, and facilitating IR to eliminate an anomaly within a containerized and orchestrated environment.

© 2017 The Author, Author retains full rights.

Source: Abed et al., Applying Bag of System Calls for Anomalous Behavior Detection of Applications in Linux Containers.
Borhani, A. (2017). Anomaly Detection, Alerting, and Incident Response for Containers.

Geplante Verbesserung

Blick in die Zukunft

- **Fully unprivileged containers**
  - Starten von Containern durch *non-root* User ohne Rechteerweiterung
  - Hier in letzter Zeit einiges an Bewegung - aber noch weiter Weg

- Verstärkte **Aktivierung von Sicherheitsfeatures** *„by default"*
  - Content Trust, ...

- **Phase 2 der User Namespaces:**
  - custom namespaces per Container
  - Upstream Kernel Support wohl schon da
  - Ziel: Einsatz in multi-tenant Umgebungen für uid/gid mapping pro Kunde

Link: Rootless containers - https://github.com/opencontainers/runc/pull/774 && https://www.cyphar.com/blog/post/rootless-containers-with-runc

Was sonst noch geschah...

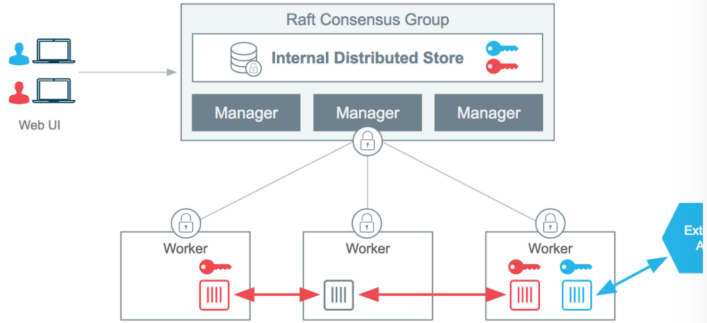
Ab Docker 1.13

Docker Secrets

blog.docker.com

By integrating secrets into Docker orchestration, we are able to deliver a solution for the secrets management problem that follows these exact principles.

The following diagram provides a high-level view of how the Docker swarm mode architecture is applied to securely deliver a new type of object to our containers: a secret object.



In Docker, a secret is any blob of data, such as a password, SSH private key, TLS Certificate, or any other piece of data that is sensitive in nature. When you add a secret to the swarm (by running `docker secret create`), Docker sends the secret over to the swarm manager over a mutually authenticated TLS connection, making use of the built-in Certificate Authority that gets automatically created when bootstrapping a new swarm.

```
$ echo "This is a secret" | docker secret create my_secret_data -
```

windsock.io

How are secrets updated?

By design, secrets in Docker swarm mode are immutable. If a secret needs to be rotated, it must first be removed from the service, before being replaced with a new secret. The replacement secret can be mounted in the same location. Let's take a look at an example. First we'll create a secret, and use a version number in the secret name, before adding it to a service as `password`:

```
$ < /dev/urandom tr -dc 'a-z0-9' | head -c 32 | docker secret create my_secret_v1.0 -
o8ozmi3sc0clf55p90oo7unaj
$ docker service create --name nginx --secret source=my_secret_v1.0,target=password nginx
t19vuui8u7le66ct0z9cwshlx
```

Once the task is running, the secret will be available in the container at `/var/run/secrets/password`. If the secret is changed, the service can be updated to reflect this:

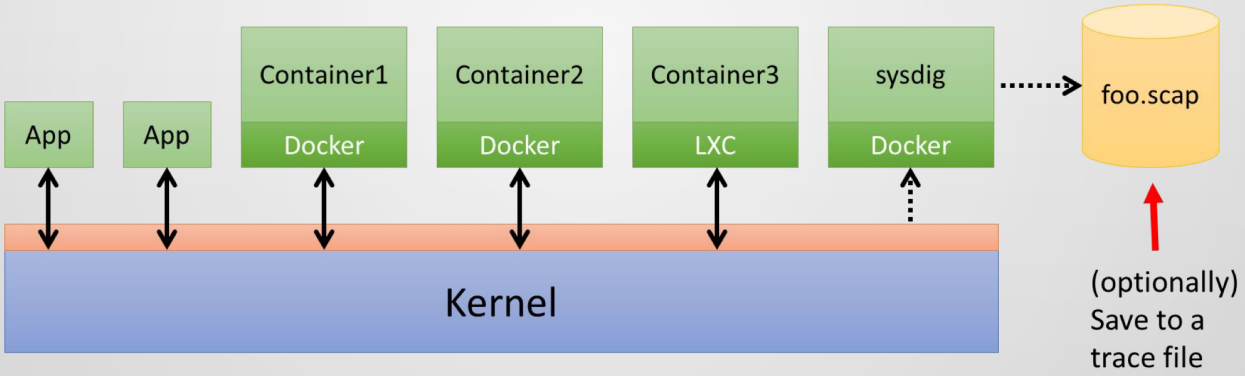
```
$ < /dev/urandom tr -dc 'a-z0-9' | head -c 32 | docker secret create my_secret_v1.1 -
p4zugztwx00jf48zz9drv2ov0
$ docker service update --secret-rm my_secret_v1.0 --secret-add source=my_secret_v1.1,target
=password nginx
nginx
```

Source: https://blog.docker.com/2017/02/docker-secrets-management/ && http://windsock.io/secrets-come-to-docker/

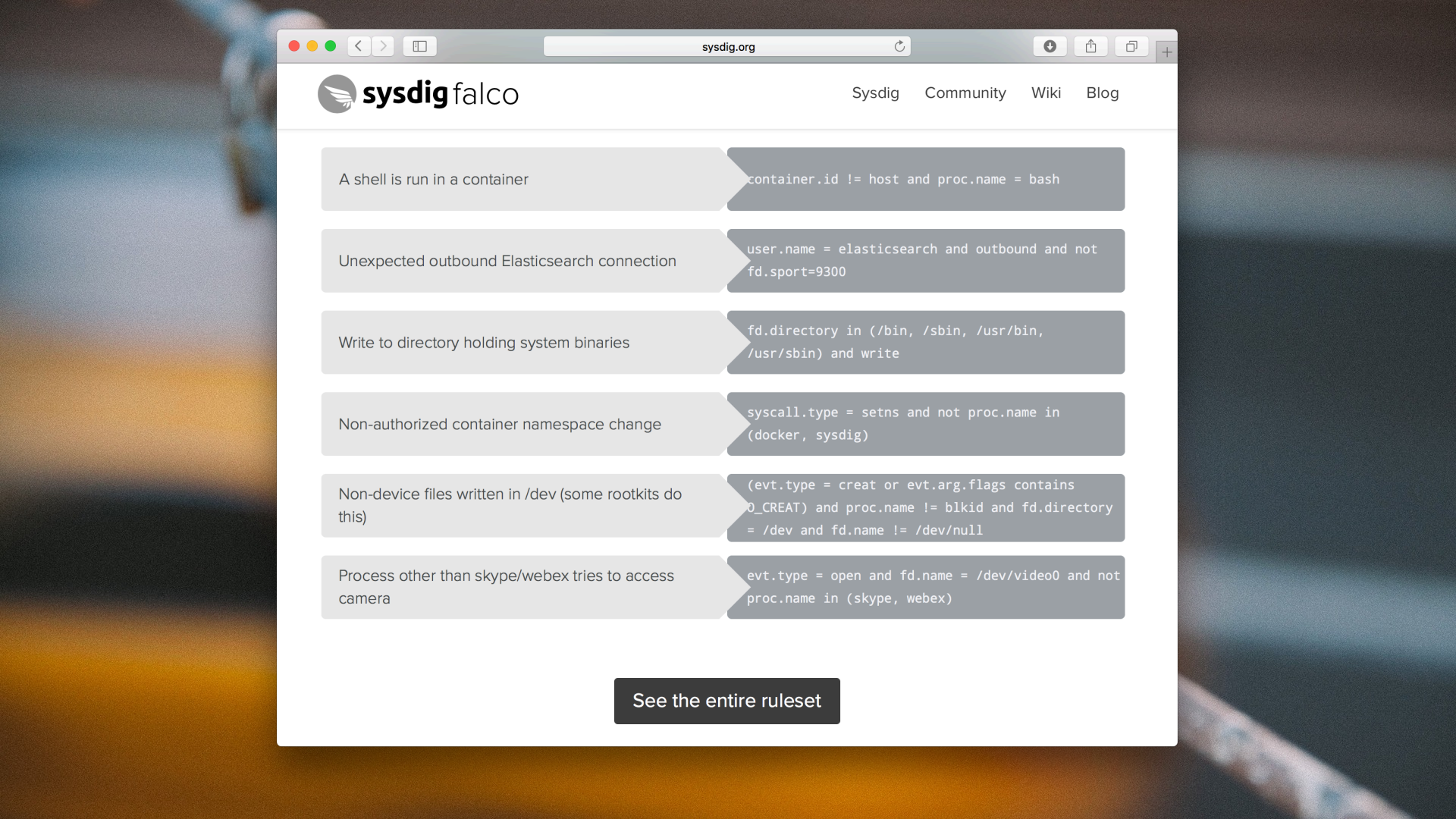
Sysdig + Sysdig/Falco

Think of sysdig as **strace + tcpdump + htop + iftop + lsof + transaction tracing +** awesome sauce.

Sysdig



Source: https://www.slideshare.net/Sysdig/trace-everything-when-apm-meets-sysadmins


sysdig.org
sysdig falco
Sysdig
Community
Wiki
Blog
A shell is run in a container
container.id != host and proc.name = bash
Unexpected outbound Elasticsearch connection
user.name = elasticsearch and outbound and not fd.sport=9300
Write to directory holding system binaries
fd.directory in (/bin, /sbin, /usr/bin, /usr/sbin) and write
Non-authorized container namespace change
syscall.type = setns and not proc.name in (docker, sysdig)
Non-device files written in /dev (some rootkits do this)
(evt.type = creat or evt.arg.flags contains O_CREAT) and proc.name != blkid and fd.directory = /dev and fd.name != /dev/null
Process other than skype/webex tries to access camera
evt.type = open and fd.name = /dev/video0 and not proc.name in (skype, webex)
See the entire ruleset

Trotzdem!

Prozesse

Nicht das Rad neu erfinden!

Bestehende Best Practices übertragen

Images

- als erstes „Einfallstor“ – viele offene Fragen – Prozesse übertragen!

  - **Verwaltung** – Auth Integration, rollenbasiert, Integration dritter Quellen
  - **Scannen** – Statische Analyse: Shellshock, SSL, ... – Umgang damit?
  - **Bauen** – Prozess übertragbar? Integration in Configuration Management?
  - **Integrität** – Build -> Run „untampered“? Signing?
  - **Umgang** mit Third Party Images – ggf analog zu weiterer Software
  - **Lifecycle** – Patchprozess, was wie updaten? **Kehrwoche!**
  - **Basisimage** - unternehmensweit?

- Passwortverwaltung – wie Passwörter in Applikation einbringen

Sicherheit, Audits

- **root, docker-Gruppe** – Default: Zugriff auf *docker* = *erweitere Rechte*
  - **Wer** darf zugreifen – **wie** kontrollieren wer **was** laufen lässte? (RBAC)
- **Monitoring**
  - **Was** läuft alles – und **woher** ist das?
  - Ist das "**sicher**" (Patchlevel)? *Kritische* **Modifikationen** durch Container?
- **„Forensic"** – wenn was schief ging
  - **Wer** hat den bösen Container gestartet? Wer gebaut?
  - **Was** hat ein bereits beendeter Container angerichtet? ... anrichten können?
  - **Logging** – was, wohin, Standards?

- Bietet meine Enterprise Distribution die neusten Sicherheitsfeatures?
  - Welche Docker Version ist so ggf verfügbar?

Containers - Vulnerability Analysis

Theo Combe
Nokia
Bell Labs France
Nozay, France
Email: theo-nokia@sutell.fr

Antony Martin
Nokia
Bell Labs France
Nozay, France
Email: antony.martin@nokia.com

Roberto Di Pietro
Nokia
Bell Labs France
Nozay, France
Email: roberto.di-pietro@nokia.com

***Abstract*—Cloud based infrastructures have typically leveraged virtualization. However, the need for always shorter development cycles, continuous delivery and cost savings in infrastructures, led to the rise of containers. Indeed, containers provide faster deployment than virtual machines and near-native performance. In this work, we study the security implications of the use of containers in typical use-cases, through a vulnerability-oriented analysis of the Docker ecosystem. Indeed, among all container solutions, Docker is currently leading the market. More than a container solution, it is a complete packaging and software delivery tool. In particular, we provide several contributions to the analysis of the containers security ecosystem: using a top-down approach, we point out vulnerabilities —present by design or driven by some realistic use-cases— in the different components of the Docker environment. Moreover, we detail real world scenarios where these vulnerabilities could be exploited, propose possible fixes, and, finally discuss the adoption of Docker by PaaS providers.**

KEYWORDS

Security, Containers, Docker, Virtual Machines, DevOps, Orchestration.

I. Introduction

Virtualization-rooted cloud computing is a mature market. There are both commercial and Open Source driven solutions. For the former ones, one may mention Amazon's Elastic Compute Cloud (EC2) [1], Google Compute Engine [2] [3], VMware's vCloud Air, Microsoft's Azure, while for the latter ones examples include OpenStack combined with virtualization technologies such as KVM or Xen.

Recent developments have set the focus on two main directions. First, the acceleration of the development cycle (agile methods and *devops*) and the increase in complexity of the application stack (mostly web services and their frameworks) trigger the need for a fast, easy-to-use way of pushing code into production. Further, market pressure leads to the densification of applications on servers. This means running more applications per physical machine, which can only be achieved by reducing the infrastructure overhead.

In this context, new lightweight approaches such as containers or unikernels [4] become increasingly popular, being more flexible and more resource-efficient. Containers achieve their goal of efficiency by reducing the software overhead imposed by virtual machines (VM) [5] [6] [7], thanks to a tighter integration of guest applications into the host operating system (OS). However, this tighter integration also increases the attack surface, raising security concerns.

The existing work on container security [8] [9] [10] [11] focuses mainly on the relationship between the host and the container. This is absolutely necessary because, while virtualization exposes well-defined resources to the guest system (virtual hardware resources), containers expose (with restrictions) the host's resources (e.g. IPC / filesystem) to the applications. However, the latter feature represents a threat for confidentiality and availability of applications running on the same host.

Containers are now part of a complex ecosystem - from container to various repositories and orchestrators - with a high level of automation. In particular, container solutions embed automated deployment chains [12] meant to speed up code deployment processes. These deployment chains are often composed of third parties elements, running on different platforms from different providers, raising concerns about code integrity. This can introduce multiple vulnerabilities that an adversary can exploit to penetrate the system. To the best of our knowledge, while deployment chains are fundamental for the adoption of containers, the security of their ecosystem has not been fully investigated yet.

The vulnerabilities we consider are classified, relatively to a hosting production system, from the most remote ones to the most local ones, using Docker as a case study. We actually focus on Docker's ecosystem for three reasons. First, Docker successfully became the reference on the market of container and associated *DevOps* ecosystem. In particular, 92% of surveyed people by ClusterHQ and DevOps.com [13] are using or planning to use Docker in a container solution. Second, security is the first barrier to container adoption in production environment [13]. Finally, Docker is already running in some environments which enable experiments and exploring the practicality of some attacks.

In this paper, we provide several contributions. First, we make a thorough list of security issues related to the Docker ecosystem, and run some experiments on both local (host-related) and remote (deployment-related) aspects of this ecosystem. Second, we show that the design of this ecosystem triggers behaviours (captured in three use-cases) that lower security when compared to the adoption of a VM based solution, such as automated deployment of untrusted code. This is the consequence of both the close integration of containers into the host system and of the incentive to scatter the deployment pipeline at multiple cloud providers. Finally, we argument on the fact that these use-cases trigger and

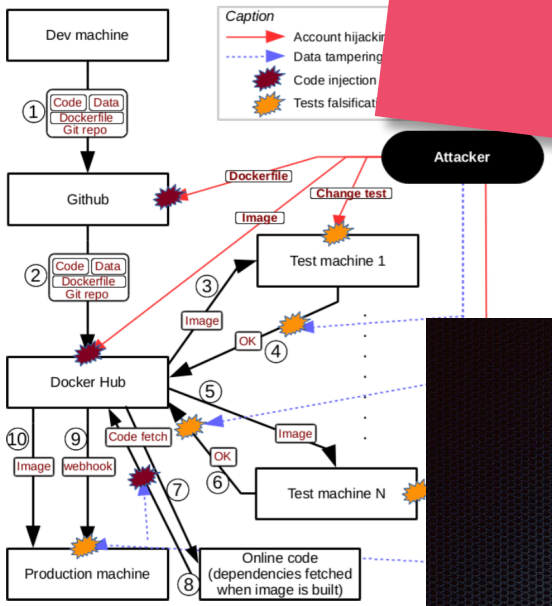


Fig. 4: Automated deployment setup in
using github, the Docker Hub, external
repositories from where code is downlo
process.

What could possibly go wrong?

Attacking a Big Data Developer

Dr. Olaf Flebbe
of ät oflebbe.de

ApacheCon Bigdata Europe
16.Nov.2016 Seville

Source: Combe et al., Containers - Vulnerability Analysis. +
http://events.linuxfoundation.org/sites/events/files/slides/AttackingBigDataDeveloper_0.pdf

effectivemachines.com

Docker Security in Framework Managed, Multi-user Environments

By Allen Wittenauer | 2017-06-02 0 Comment

US Coast Guard Rescue Demo

A while back, Jessie Frazelle wrote and published an informative blog post on the differences between containers, zones, and jails. Since it touched on security, the blog post reminded me of a conversation that was had last year when a contributor to the Apache Yetus project asked about this blog post about one of the bigger security issues with regards to using Docker.

The TL;DR of the issue is that anything permitted usage of dockerd's socket is allowed to mount root or any other file system that the daemon can also access. The solution provided in the Project Atomic (aka Red Hat) blog was mainly about the idea that one should just use sudo to limit the damage. In other words, wrap the docker command such that it specifies all the parameters. Additionally, they proposed the idea that dockerd should do some authorization to limit who can access dockerd.

For many deployments, this proposed solution of using sudo probably works well. But for others, the

Search

Recent Posts

- Docker Security in Framework Managed, Multi-user Environments
- Unofficial History of the HDFS Audit Log
- Building Your Own Apache Hadoop Distribution
- Taking Control of Daemons in Apache Hadoop
- Adding to Apache Hadoop's Classpath

Recent Comments

Allen Wittenauer on Building Your Own Apache Hadoop Distribution

david serafini on Building Your Own Apache Hadoop Distribution

Categories

- Apache Hadoop
- Apache Yetus

Meta

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.org

Source: https://effectivemachines.com/2017/06/02/docker-security-in-framework-managed-multi-user-environments/

Weiterführendes

Introduction to Container Security

create resource restrictions around deployed applications. The Docker container model supports and enforces these restrictions by running applications in their own root filesystem, allows the use of separate user accounts, and goes a step further to provide application sandboxing using Linux *namespaces* and *cgroups* to mandate resource constraints. While these powerful isolation mechanisms have been available in the Linux kernel for years, Docker brings forward and greatly simplifies the capabilities to create and manage the constraints around distributed applications containers as independent and isolated units.

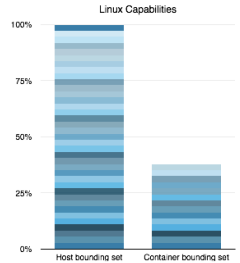
Docker takes advantage of a Linux technology called *namespaces*[2], to provide the isolated workspace we call *container*. When a container is deployed, Docker creates a set of namespaces for that specific container, isolating it from all the other running applications.

Docker also leverages Linux control groups. Control groups[3] (or *cgroups* for short), are the kernel level functionality that allows Docker to control what resources each container has access to, ensuring good container multi-tenancy. Control groups allow Docker to share available hardware resources and, if required, set up limits and constraints for containers. A good example is limiting the amount of memory available to a specific container, so it doesn't completely exhaust the resources of the host.

Process Restrictions

Restricting access and capabilities reduces the amount of surface area potentially vulnerable to attack. Docker's default settings are designed to limit Linux capabilities. While the traditional view of Linux considers OS security in terms of root privileges versus user privileges, modern Linux has evolved to support a more nuanced privilege model: capabilities.

Linux capabilities allow granular specification of user capabilities and traditionally, the root user has access to every capability. Typical non-root users have a more restricted capability set, but are usually given the option to elevate their access to root level through the use of *sudo* or *setuid* binaries. This may constitute a security risk.



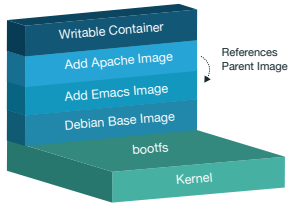
[2] http://man7.org/linux/man-pages/man7/namespaces.7.html

The default bounding set of capabilities inside a Docker container is less than half the total capabilities assigned to a Linux process (see Linux Capabilities figure). This reduces the possibility of escalation to a fully privileged root user through application-level vulnerabilities. Docker employs an extra degree of granularity, which dramatically expands on the traditional root/non-root dichotomy. In most cases, the application containers do not need all the capabilities attributed to the root user, since the large majority of the tasks requiring this level of privilege are handled by the OS environment external to the container. Containers can run with a reduced capability set that does not negatively impact the application and yet improves the overall security system levels and makes running applications more secure by default. This makes it difficult to provoke system level damages during intrusion, even if the intruder manages to escalate to root within a container because the container capabilities are fundamentally restricted.

Device & File Restrictions

Docker further reduces the attack surface by restricting access by containerized applications to the physical devices on a host, through the use of the device resource control groups (*cgroups*) mechanism. Containers have no default device access and have to be explicitly granted device access. These restrictions protect a container host kernel and its hardware, whether physical or virtual, from the running applications.

Docker containers use copy-on-write file systems, which allow use of the same file system image as a base layer for multiple containers. Even when writing to the same file system image, containers do not notice the changes made by another container, thus effectively isolating the processes running in independent containers.



Any changes made to containers are lost if you destroy the container, unless you commit your changes. Committing changes tracks and audits changes made to base images as a new layer which can then be pushed as a new image for storage in Docker Hub and run in a container. This audit trail is important in providing information to maintain compliance. It also allows for fast and easy rollback to previous versions, if a container has been compromised or a vulnerability introduced. There are a few core Linux kernel file systems that have to be in the container environment

Source: https://www.docker.com/sites/default/files/WP_Intro%20to%20container%20security_03.20.2015%20(1).pdf

VULNERABILITY EXPLOITATION IN DOCKER CONTAINER ENVIRONMENTS

ANTHONY BETTINI, FOUNDER & CEO, FLAWCHECK
ABETTINI@FLAWCHECK.COM

Presented at Black Hat Europe 2015

INTRODUCTION

Containers have been around for a long time. But only recently, have container-based virtualization solutions become commonplace within the enterprise. Docker in particular is everywhere. But why? And what does it mean for enterprise security? Is vulnerability exploitation of Docker containers any different from vulnerability exploitation of application vulnerabilities on virtual machines? Are the ways to secure them any different?

Before we jump into the vulnerability exploitation piece of the equation, it makes sense to review the security provided by the container solutions themselves – namely the workload isolation security. As with any new topic, it makes sense to start with a bit of history. How did we even get here?

MODERN HISTORY OF LINUX CONTAINERS

Today, Docker is the most widely used container-based virtualization technology. But Docker itself is an application (technically, a daemon), built on the container technology provided by the Linux kernel. The container technology provided by the Linux kernel isn't new though, it has been evolving over time, for a very long time.

Linux container technology is generally accepted to trace back to the days of chroot. Chroot was introduced way back in 1979 and started to address the isolation problem. Chroot, or "*ch*ange *root*" changes the view of the file system for the process and its children. This was particularly useful for applications such as ftpd, to restrict the view of the ftp client to subfolders of the chroot'd parent. But chroot itself wasn't built for security

FlawCheck Inc. © Copyright 2015 All Rights Reserved
FlawCheck Confidential. This document is issued by FlawCheck and is Company Proprietary. Please do not re-distribute without permission. Questions or issues may be directed to FlawCheck

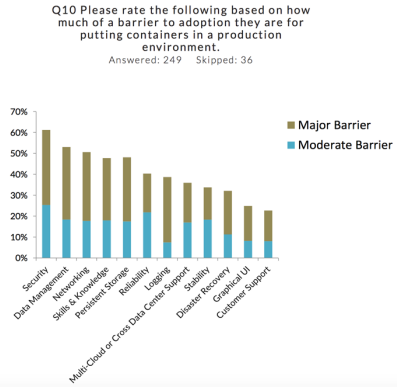
In June 2015, ClusterHQ asked enterprises "What are the biggest barriers to putting containers in a production environment?" This time, an even higher percentage of enterprises (>60%) said that security was the #1 barrier to putting containers in a production environment.

WHAT ARE THE BIGGEST BARRIERS TO PUTTING CONTAINERS IN A PRODUCTION ENVIRONMENT?

In this question respondents had the option of rating certain categories as a major barrier, moderate barrier, minor barrier or no barrier at all.

Security was the highest rated barrier to increased adoption. The second biggest barrier was data management.

Note: we combined the major and moderate barrier responses and grouped them to weigh biggest barriers.

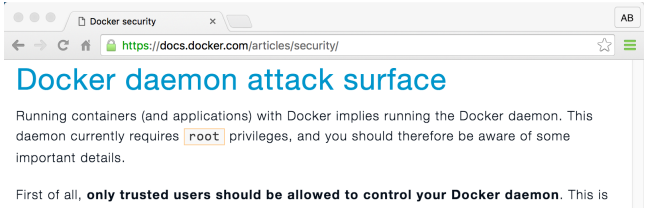


In August 2015, FlawCheck and one of our partners, surveyed enterprises asking which piece of the security equation was their top concern about running containers in production environments.

FlawCheck Inc. © Copyright 2015 All Rights Reserved
FlawCheck Confidential. This document is issued by FlawCheck and is Company Proprietary. Please do not re-distribute without permission. Questions or issues may be directed to FlawCheck

Downloading code without looking at it, that changes often and lacks any integrity check, then piping it to an interpreter and executing it as root (via putting your password into sudo, hopefully), is a fail. At least curl validates the certificate and get.docker.com doesn't support DHE or export ciphers, but still it's an awful workflow from a security perspective.

After Docker is installed, you'll realize that it's actually a daemon that runs as root:



In the event your PaaS is starting Docker with the incorrect parameters, such as host networking, users can actually shutdown the container host!

```
[ubuntu:pts/7:21:20:~% sudo docker run -it ubuntu bash
[root@08c9aab15aa5:/# shutdown now
 shutdown: Unable to shutdown system
[root@08c9aab15aa5:/# exit
 exit
[ubuntu:pts/7:21:20:~% sudo docker run --net=host -it ubuntu bash
[root@ubuntu:/# shutdown now
 root@ubuntu:/# exit
 ubuntu:pts/7:21:20:~% Connection to 172.16.135.157 closed by remote host.
 Connection to 172.16.135.157 closed.
```

Docker does actually provide a warning message against this and in practice, it's easy to avoid, but enabling host networking has surprising consequences.

In an older release of Docker, Docker actually blacklisted kernel calls (remember Docker is basically acting as a man-in-the-middle between the container and the kernel). By blacklisting kernel calls, Docker missed an

FlawCheck Inc. © Copyright 2015 All Rights Reserved
FlawCheck Confidential. This document is issued by FlawCheck and is Company Proprietary. Please do not re-distribute without permission. Questions or issues may be directed to FlawCheck

Source: https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf

O'REILLY®

Compliments of OPENSHIFT by Red Hat

Docker Security

Using Containers Safely in Production

Adrian Mouat

ing hosts between users and will result in a higher number of VMs and/or machines than reusing hosts, but is important for security. The main reason is to prevent container breakouts resulting in a user gaining access to another user's containers or data. If a container breakout occurs, the attacker will still be on a separate VM or machine and unable to easily access containers belonging to other users.

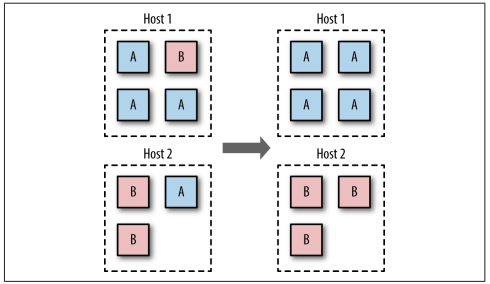


Figure 1-1. Segregating containers by host

Similarly, if you have containers that process or store sensitive information, keep them on a host separate from containers handling less-sensitive information and, in particular, away from containers running applications directly exposed to end users. For example, containers processing credit-card details should be kept separate from containers running the Node.js frontend.

Segregation and use of VMs can also provide added protection against DoS attacks; users won't be able to monopolize all the memory on the host and starve out other users if they are contained within their own VM.

In the short to medium term, the vast majority of container deployments will involve VMs. Although this isn't an ideal situation, it does mean you can combine the efficiency of containers with the security of VMs.

Set a USER

Never run production applications as `root` inside the container. That's worth saying again: *never run production applications as root inside the container*. An attacker who breaks the application will have full access to the container, including its data and programs. Worse, an attacker who manages to break out of the container will have `root` access on the host. You wouldn't run an application as root in a VM or on bare metal, so don't do it in a container.

To avoid running as `root`, your Dockerfiles should always create a nonprivileged user and switch to it with a `USER` statement or from an entrypoint script. For example:

```
RUN groupadd -r user_grp && useradd -r -g user_grp user
USER user
```

This creates a group called `user_grp` and a new user called `user` who belongs to that group. The `USER` statement will take effect for all following instructions and when a container is started from the image. You may need to delay the `USER` instruction until later in the Dockerfile if you need to first perform actions that need root privileges such as installing software.

Many of the official images create an unprivileged user in the same way, but do not contain a `USER` instruction. Instead, they switch users in an entrypoint script, using the *gosu* utility. For example, the entry-point script for the official Redis image looks like this:

```
#!/bin/bash
set -e
if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

This script includes the line `chown -R redis .`, which sets the ownership of all files under the images data directory to the `redis` user. If the Dockerfile had declared a `USER`, this line wouldn't work. The next line, `exec gosu redis "$@"`, executes the given `redis` command as the redis user. The use of `exec` means the current shell is replaced with `redis`, which becomes PID 1 and has any signals forwarded appropriately.

Source: https://www.openshift.com/promotions/docker-security.html

nccgroup

NCC Group Whitepaper

Understanding and Hardening Linux Containers

April 20, 2016 – Version 1.0

Prepared by
Aaron Grattafiori – Technical Director

Abstract

Operating System virtualization is an attractive feature for efficiency, speed and modern application deployment, amid questionable security. Recent advancements of the Linux kernel have coalesced for simple yet powerful OS virtualization via Linux Containers, as implemented by LXC, Docker, and CoreOS Rkt among others. Recent container focused start-ups such as Docker have helped push containers into the limelight. Linux containers offer native OS virtualization, segmented by kernel namespaces, limited through process cgroups and restricted through reduced root capabilities, Mandatory Access Control and user namespaces. This paper discusses these container features, as well as exploring various security mechanisms. Also included is an examination of attack surfaces, threats, and related hardening features in order to properly evaluate container security. Finally, this paper contrasts different container defaults and enumerates strong security recommendations to counter deployment weaknesses– helping support and explain methods for building high-security Linux containers. Are Linux containers the future or merely a fad or fantasy? This paper attempts to answer that question.

Weak or missing procfs and sysfs limits by default. Rkt is effectively missing a number of limits for procfs (`/proc`) and sysfs (`/sys`), allowing information to leak from the container host or easily allowing attacks from the guest container. This includes but is not limited to the following exploits discussed within 7.2.1 on page 52: uevent_helper, sysrq-trigger, core_pattern, and modprobe. While some protections are enabled by default via read-only bind mounts, these can be easily subverted by using `CAP_SYS_ADMIN` to remount the mounts as read-write.

9.13 Container Defaults

Listed below are the relevant security features for the three major container platforms explored within this paper. Each security feature is covered directly or indirectly within this paper and the title can be clicked, for those which are covered in detail, in order to jump to the relevant section. To avoid any misconceptions, the following parameters are defined as to their use in the table below:

- Default: The security feature is enabled by default.
- Strong Default: The most secure configuration is enabled by default.
- Weak Default: A less secure configuration is enabled by default.
- Optional: The security feature can be optionally configured. This is not a given weakness unless no other equivilant feature can be configured or enabled.
- Not Possible: The security feature cannot be configured in any way, no documentation exists, the feature is still under development, or the feature is not planned to be implemented.

Available Container Security Features, Requirements and Defaults			
Security Feature	**LXC 2.0**	**Docker 1.11**	**CoreOS Rkt 1.3**
User Namespaces	Default	Optional	Experimental
Root Capability Dropping	Weak Defaults	Strong Defaults	Weak Defaults
Procfs and Sysfs Limits	Default	Default	Weak Defaults
Cgroup Defaults	Default	Default	Weak Defaults
Seccomp Filtering	Weak Defaults	Strong Defaults	Optional
Custom Seccomp Filters	Optional	Optional	Optional
Bridge Networking	Default	Default	Default
Hypervisor Isolation	Coming Soon	Coming Soon	Optional
MAC: AppArmor	Strong Defaults	Strong Defaults	Not Possible
MAC: SELinux	Optional	Optional	Optional
No New Privileges	Not Possible	Optional	Not Possible
Container Image Signing	Default	Strong Defaults	Default
Root Interation Optional	True	False	Mostly False

Source: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-10pdf/

CIS Docker 1.12.0 Benchmark

v1.0.0 - 08-15-2016

1.8 Audit Docker files and directories - /var/lib/docker (Scored,

Profile Applicability:

- Level 1 - Linux Host OS

Description:

Audit `/var/lib/docker`.

Rationale:

Apart from auditing your regular Linux file system and system calls, audit all D related files and directories. Docker daemon runs with '`root`' privileges. Its beh depends on some key files and directories. `/var/lib/docker` is one such direct all the information about containers. It must be audited.

Audit:

Verify that there is an audit rule corresponding to `/var/lib/docker` directory.

For example, execute below command:

```
auditctl -l | grep /var/lib/docker
```

This should list a rule for `/var/lib/docker` directory.

Remediation:

Add a rule for `/var/lib/docker` directory.

For example,
Add the line as below in `/etc/audit/audit.rules` file:

```
-w /var/lib/docker -k docker
```

Then, restart the audit daemon. For example,

```
service auditd restart
```

Impact:
Auditing generates quite big log files. Ensure to rotate and archive them period create a separate partition of audit to avoid filling root file system.

2.11 Use authorization plugin (Scored)

Profile Applicability:

- Level 2 - Docker

Description:

Use authorization plugin to manage access to Docker daemon.

Rationale:

Docker's out-of-the-box authorization model is all or nothing. Any user with permission to access the Docker daemon can run any Docker client command. The same is true for callers using Docker's remote API to contact the daemon. If you require greater access control, you can create authorization plugins and add them to your Docker daemon configuration. Using an authorization plugin, a Docker administrator can configure granular access policies for managing access to Docker daemon.

Audit:

```
ps -ef | grep dockerd
```

Ensure that the '`--authorization-plugin`' parameter is set as appropriate.

Remediation:

Step 1: Install/Create an authorization plugin.

Step 2: Configure the authorization policy as desired.

Step 3: Start the docker daemon as below:

```
dockerd --authorization-plugin=<PLUGIN_ID>
```

Impact:

Each docker command specifically passes through authorization plugin mechanism. This might introduce a slight performance drop.

Default Value:

By default, authorization plugins are not set up.

Source: https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.12.0_Benchmark_v1.0.0.pdf

Docker Reference Architecture: Securing Docker Datacenter and Security Best Practices

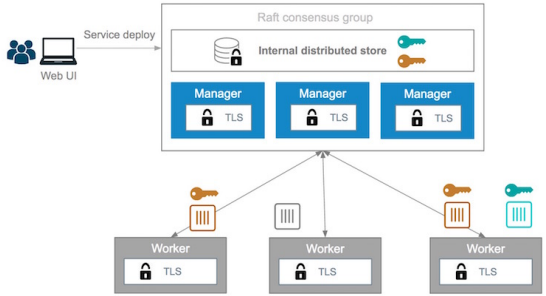
Secrets requires a Swarm mode cluster. You can use secrets to manage any sensitive data which a container needs at runtime but you don't want to store in the image or in source control such as:

- Usernames and passwords
- TLS certificates and keys
- SSH keys
- Other important data such as the name of a database or internal server
- Generic strings or binary content (up to 500 kb in size)

> **Note**: Docker secrets are only available to Swarm services, not to standalone containers. To use this feature, consider adapting your container to run as a service with a scale of 1.

Another use case for using secrets is to provide a layer of abstraction between the container and a set of credentials. Consider a scenario where you have separate development, test, and production environments for your application. Each of these environments can have different credentials, stored in the development, test, and production swarms with the same secret name. Your containers only need to know the name of the secret to function in all three environments.

When you add a secret to the swarm, Docker sends the secret to the Swarm manager over a mutual TLS connection. The secret is stored in the Raft log, which is encrypted. The entire Raft log is replicated across the other managers, ensuring the same high availability guarantees for secrets as for the rest of the swarm management data.



When you grant a newly-created or running service access to a secret, the decrypted secret is mounted into the container in an in-memory filesystem at `/run/secrets/<secret_name>`. You can update a service to grant it access to additional secrets or revoke its access to a given secret at any time.

Source:
https://success.docker.com/KBase/Docker_Reference_Architecture%3A_Securing_Docker_Datacenter_and_Security_Best_Practices

7 Zusammenfassung & Fazit

Containers do not contain?

Threads & Mitigation I – *Container* <-> ...

Threat	Mitigation
DoS against Host +other containers (*noisy neighbours)* | Forkbomb	Cgroups, Quotas, Kernel PID limits
Access to host and private information + other containers	Namespaces, seccomp, LSM*: AppArmor, SELinux
Kernel modification + module load	Capabilities (dropped by default), seccomp, LSM, no –privileged
API socket access (for Docker administration + full control over other container)	Only share socket with AuthZ limitations, TLS for TCP endpoints

* LSM = Linux Security Modules

Threads & Mitigation II – *External -> Container*

Threat	Mitigation
DDoS attacks	Monitoring infrastructure + Cgroups, Quotas, Kernel PID limits
(Malicious) remote access	Application security model * Secure passwords * --readonly FS *
Exploits	Static vul. scanning (Clair, ...) Dynamic scanning
Application Security	Not container specific * Reduced impact by container *walls*

* same procedure as non-containerized

** cgroups usage more likely in containerized environments

root exploit

twitter.com/Ben_Hall/status/728596633978572801

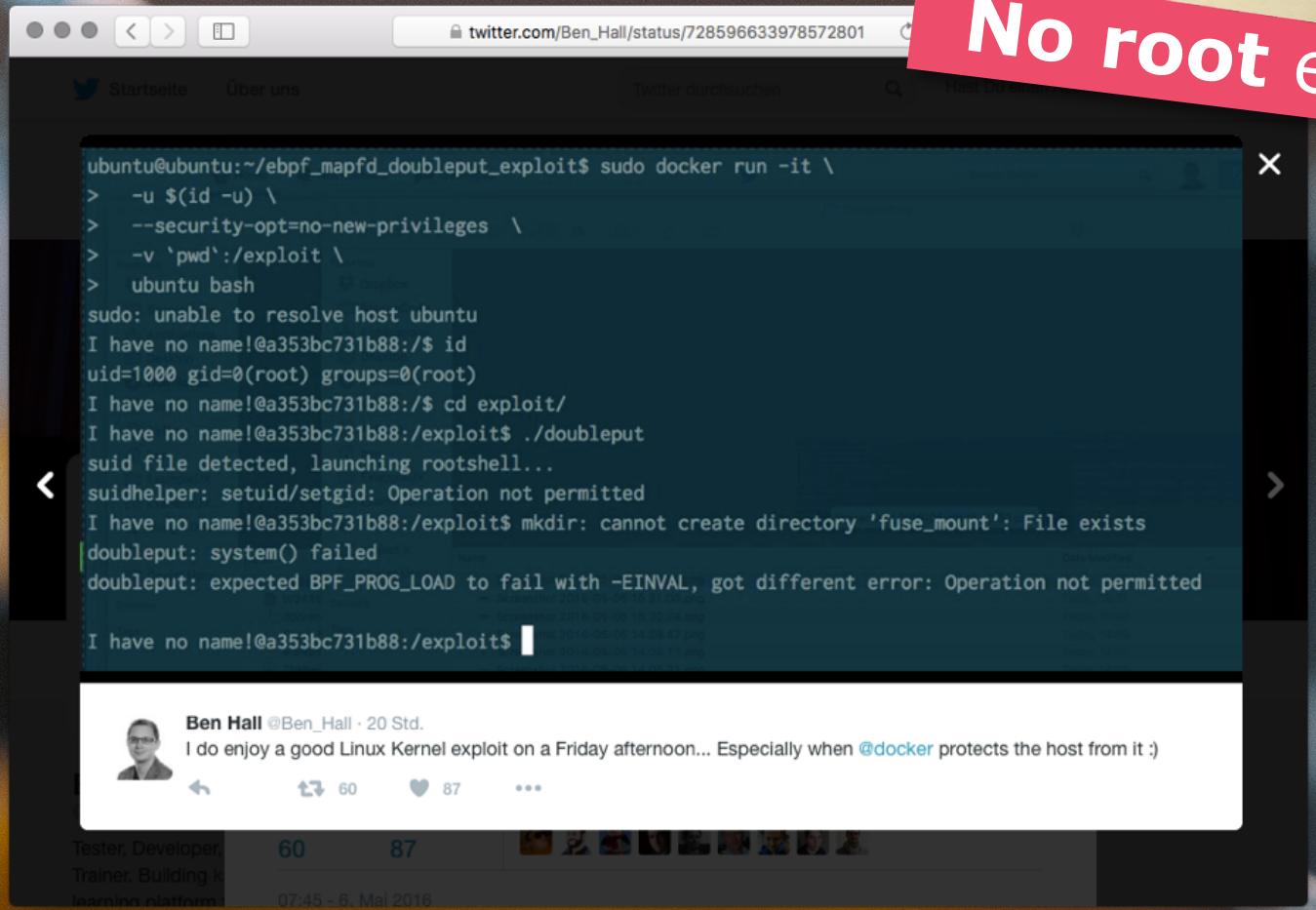
```
ubuntu@ubuntu:~/ebpf_mapfd_doubleput_exploit$ ./doubleput
starting writev
woohoo, got pointer reuse
writev returned successfully. if this worked, you'll have a root shell in <=60 seconds.
suid file detected, launching rootshell...
we have root privs now...
root@ubuntu:~/ebpf_mapfd_doubleput_exploit# whoami
root
```

Ben Hall @Ben_Hall · 20 Std.

I do enjoy a good Linux Kernel exploit on a Friday afternoon... Especially when @docker protects the host from it :)

60 87

Source: https://twitter.com/Ben_Hall/status/728596633978572801



Source: https://twitter.com/Ben_Hall/status/728596633978572801

Zusammenfassung

- Container != VM, != Sandbox
  - Sicherheit **kann** mit Containern besser als *bare metal* sein
- Inzwischen sehr viele Möglichkeiten zur Absicherung
  - Features wollen genutzt werden! Und manchmal erst aktiviert ;)
  - Vertrauensvolle Images + sicher konfigurierter Host + limitierter Zugriff + gesunder Menschenverstand...
  - ... - Faktor Mensch wie so oft die schwächste Komponente
- Etablierte Prozesse und Verfahren anwenden
  - Nicht weil es „so leicht“ geht bewährte Konzepte über Bord werfen

Wa(h)lwerbung?

Source: cdn2.spiegel.de/images/image-806145-galleryV9-ygfz.jpg

Thanks

For more information please contact:
Holger Gantikow
T +49 7071 94 57-503
h.gantikow@atos.net
h.gantikow@science-computing.de

Atos, the Atos logo, Atos Codex, Atos Consulting, Atos Worldgrid, Worldline, BlueKiwi, Bull, Canopy the Open Cloud Company, Unify, Yunano, Zero Email, Zero Email Certified and The Zero Email Company are registered trademarks of the Atos group. April 2016. © 2016 Atos. Confidential information owned by Atos, to be used by the recipient only. This document, or any part of it, may not be reproduced, copied, circulated and/or distributed nor quoted without prior written approval from Atos.

Atos