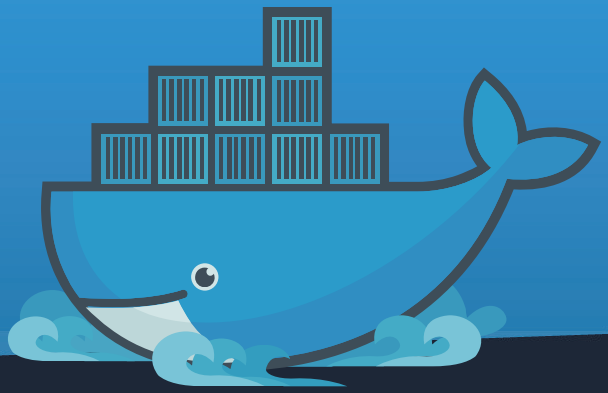
TMF – Technologie- und Methodenplattform
für die vernetzte medizinische Forschung e.V.



Dockerbank 2

Szenarien des Routinebetriebs

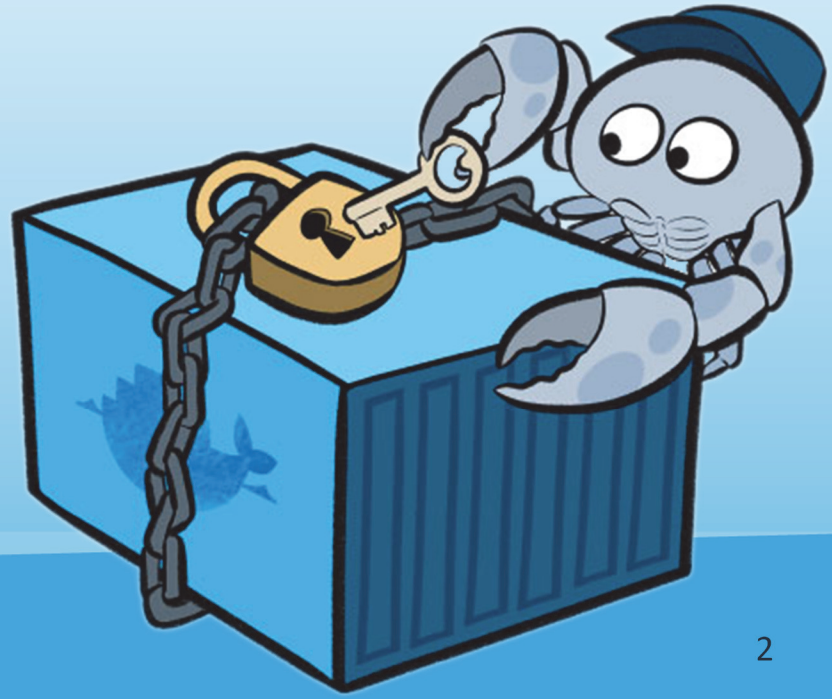
Jens Piegsa

Universitätsmedizin Greifswald

Berlin, 07.12.2016

Inhalt

- Docker im Produktiv-Einsatz
- Antipatterns
- Konfigurationsmanagement
- Docker-Containern absichern
- kontinuierliche Sicherheitsanalyse
- Secret-Management
- Datensicherung und -wiederherstellung
- Protokollieren, Überwachen, Benachrichtigen
- Fazit, weitere Materialien



DOCKER IM PRODUKTIV-EINSATZ

Was bedeutet produktiv?

- Betriebsszenarien:
  - Docker lokal betreiben
  - auf einem Intranet-Server
  - auf einem eigenen Webserver
  - auf mehreren eigenen Servern
  - auf cloud-basierter Infrastruktur
- unterscheiden sich in Zielen und Anforderungen

Docker lokal betreiben

- Ziele:
  - leichtgewichtige Instanz einer vollständigen Umgebung zu Testzwecken
  - Entwicklung / Debugging von Docker Images
- Maßnahmen:
  - Einsatz vertrauenswürdiger Images (keine Schadcode)
  - Verwendung von Volumes (kein Datenverlust zwischen mehreren Session)
  - Durchführung manueller Backups

Docker im Intranet betreiben

- Ziele:
  - dauerhafte Bereitstellung von Diensten
  - Test- / Deployment-Pipelines
- Maßnahmen:
  - periodische Backups
  - Softwareaktualisierungen
  - Einbindung von Storagelösungen
  - Ressourcen-Zuteilung
  - Monitoring

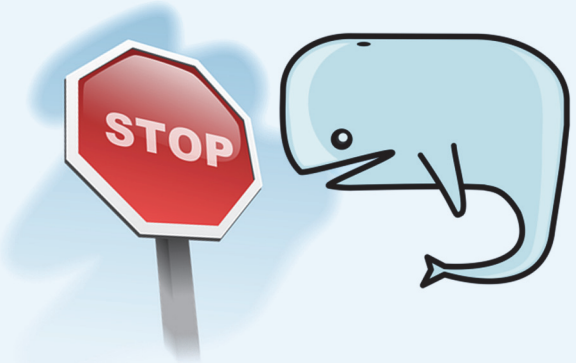
Docker auf einem Webserver

- Ziele:
  - dauerhafter Betrieb
  - kurzzeitiger Betrieb zu Test-, Demonstrationszwecken
- Maßnahmen:
  - Serverzertifikate, verschlüsselte Kommunikation
  - Monitoring, Logging
  - Hardening des Host-Systems, restriktive Firewall-Konfiguration

Docker auf mehreren Servern

- Ziele:
  - Skalierung, Ausfallsicherheit von Diensten
  - mehrere Mandanten
  - Sicherheit: Trennung durch Netzwerkzonen, VMs
  - Testumgebung
- Maßnahmen:
  - Orchestrierung
  - Konfigurationsmanagement, Secret Management
  - Private Docker Registry
  - CI/CD Pipelines

Docker produktiv: Antipatterns



- Datenhaltung im Container
- Images mit umgebungsspezifischer Konfiguration bzw. sensiblen Daten; umgebungsspezifische Tags
- Einsatz nicht reproduzierbarer Images (mittels `docker commit` erzeugt oder aus Fremdquellen)
- Container-Modifikationen (`docker cp`, `docker exec`)
- langlebige Container
- Ausführung mehrerer Prozesse innerhalb eines Containers
- Images mit vielen Schichten
- implizite oder explizite Ausführung / Abhängigkeit von `latest`
- Veröffentlichung aller Ports via `docker run -P`
- zeitliche Abhängigkeiten zwischen Containern

KONFIGURATIONSMANAGEMENT

Konfigurationsmanagement
Docker Container

- Docker Image so entwickeln, dass
  - sie unabhängig von der Umgebung (Entwicklung, Test, QA, Staging, Produktion) bleiben
  - initiale Konfigurationen über Umgebungsvariablen vorgenommen werden können, die ggf. anwendungs-abhängig über ein Entrypoint-Skript angewendet werden
  - veränderliche Konfigurationen aus Dateien, idealerweise in separaten Verzeichnissen, gelesen wird, die als (benannte) Volumes gemountet werden
  - es keiner nachträglichen Modifikationen des instanziierten Containers bedarf

Konfigurationsmanagement Hostumgebung

- Einsatz herkömmlicher Konfigurations-managementwerkzeuge wie Puppet, Chef und Ansible möglichst auf ein Minimum reduzieren
  - Installation / Update der Hostumgebung
    - docker, docker-compose, Kubernetes-, Mesos-Cluster
  - Einrichtung hybrider Umgebungen
  - Sicherheitsvorkehrungen

SICHERHEIT VON DOCKER-CONTAINERN

Sicherheit: Container vs. VMs

Kriterium	Virtuelle Maschinen (VM)	Container
Isolation	verfügen über eine zusätzliche Hypervisor-Schicht	teilen sich einen Kernel
Angriffsfläche	enthalten viele Tools, die sich Angreifer zunutze machen können	minimale Images
Kontrolle	Begrenzung des Zugriffs auf Ressourcen	Begrenzung des Zugriffs auf Ressourcen; read-only Dateisystem; Kernel Capabilities
Auditierung	langlebig; divergieren vom Basisimage; werden mittels Konfigurationsmanagement-software gepatcht	kurzlebig; Aktualisierung durch Austausch; Image-Anpassungen unter Versions-kontrolle; docker diff
Zuverlässigkeit	haben sich in der Vergangenheit bewährt	weniger Erfahrungswerte; rapide Entwicklung
Fazit	• kombinierter Einsatz von VMs und Containern: VMs zur Separierung von Container-Gruppen und Container als zusätzliche Sicherheit und wegen ihrer Features • aktuell viele Bestrebungen VMs leichtgewichtiger und Container sicherer zu machen • grundsätzlich: gestaffelte Sicherheitsvorkehrungen, Least-Privilege-Prinzip	

Bedrohungen und Maßnahmen

	Kernel-Exploits	Denial-of-Service-Angriffe	Container-Breakouts	Vergiftete Images	Verratene Geheimnisse
Images auditieren (Dockerfile, statische Analyse)				●	
Abgrenzung von Container-Gruppen durch VMs		●			
Gesamtes Container-Dateisystem read-only setzen	●	●	●		●
Sensible Volumes read-only deklarieren	●		●		
Kernel Capabilities einschränken	●		●		
CPU-Ressourcen zuweisen		●			
Arbeitsspeicher limitieren		●			
SETUID/SETGID-Dateizugriffsrechte entfernen	●		●		
Kommunikation zwischen Containern einschränken	●	●	●		
Nicht-Root USER im Dockerfile festlegen	●		●		●
Geheimnisse nicht in Umgebungsvariablen ablegen					●
Container nicht `--privileged` starten	●		●		●
Minimale Images ohne unnötige Pakete	●		●		
AppArmor, SELinux, Seccomp	●				

Syntax der Übungsbeispiele

```
mkdir ~/foo && \
echo "Beispiel" | \
  tee -a ~/foo/bar && \
cat ~/foo/bar
```

Schreibzugriffe einschränken

- gesamtes Dateisystem auf read-only setzen:

```
docker run --read-only --rm alpine touch x
```

(kombinierbar mit Schreibrechten für Volumes)

- sensibles Volume read-only deklarieren:

```
docker run -v $(pwd):/secrets:ro --rm \
 alpine touch /secrets/x
```

Kernel Capabilities einschränken

- bestimmte Capabilities deaktivieren:

```
docker run --cap-drop SETGID --cap-drop SETUID \
 --rm alpine su -c whoami postgres
```

- sämtliche Capabilities abschalten:

```
docker run --cap-drop ALL --rm alpine \
 ping localhost -c1
```

- Whitelist- und Blacklist-Verfahren möglich durch Kombination mit `--cap-add`
- siehe Man-Pages: `man capabilities`

CPU-Ressourcen zuweisen

- erfolgt anhand von relativer Wichtungen
- Wichtung beträgt standardmäßig 1024
- Beispiel erzeugt drei Container mit den Last-wichtungen 50%, 25% und 25%:

```
docker run -d alpine sh -c 'yes>/dev/null' && \
docker run -d -c 512 alpine sh -c 'yes>/dev/null' && \
docker run -d -c 512 alpine sh -c 'yes>/dev/null' && \
top
```

- Container entfernen mit `docker rm -f …`

Arbeitsspeicher limitieren

- einmalig *Memory* und *Swap Limit Capabilities* im Host aktivieren (erfordert Neustart):

```
echo 'GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"'| \
 sudo tee -a /etc/default/grub && \
sudo update-grub && sudo reboot
```

- Container auf 42MB Arbeits- und 42MB Swap-Speicher beschränken:

```
docker run -m 42m -d alpine tail -f /dev/null && \
docker stats
```

(Abbruch mittels der Tastenkombination Strg+C)

Ausführbare Dateien mit setuid-/setgid-Zugriffsrechten entschärfen

1. Alle privilegierten Dateien finden:

```
docker run --rm debian find / -type f -perm /6000 \
 -exec stat -c "%A %a %n" {} \; 2>/dev/null
```

2. Neues `Dockerfile` erstellen, das bei der Image-Erstellung allen Dateien die kritischen Zugriffsrechte entzieht:

```
FROM debian:8.6
RUN find / -type f -perm /6000 -exec chmod a-s {} \; || true
USER daemon
```

3. Container auf Basis des neuen Image erstellen und testen:

```
docker build -t mostly-harmless . && \
docker run --rm mostly-harmless find / -type f -perm /6000 \
-exec stat -c "%n" {} \; 2>/dev/null | wc -l
```

Absicherung der Kommunikation zwischen Containern (I)

- Docker-Terminologie *(expose / publish ports)* lässt vermuten, dass die Kommunikation zwischen Containern standardmäßig eingeschränkt ist

- Experiment:

```
docker inspect $(docker run -d nginx:alpine)
docker run --rm alpine wget -qO - -T10 -t1 <IPAddress>:80
```

Absicherung der Kommunikation zwischen Containern (II)

- generelle Deaktivierung mit Ausnahme bei Container-Verlinkung:

```
echo 'DOCKER_OPTS="--icc=false --iptables"' | \
 sudo tee -a /etc/default/docker
```

- unter Debian/Ubuntu zusätzlich notwendige Anpassung mittels `sudoedit /lib/systemd/system/docker.service` vornehmen:

```
...
[Service]
ExecStart=/usr/bin/docker -d -H fd:// $DOCKER_OPTS
EnvironmentFile=/etc/default/docker
...
```

- Neustart und Prüfen der Argumente mittels der Prozessliste:

```
sudo systemctl daemon-reload && \
sudo systemctl restart docker && \
ps aux | grep dockerd
```

Absicherung der Kommunikation zwischen Containern (III)

- Experiment wiederholen:

```
docker inspect $(docker run -d nginx:alpine)
docker run --rm alpine wget -qO - -T10 -t1 <IPAddress>:80
```

- stattdessen Port-Publishing über Host nutzen:

```
docker run -d -p 10080:80 nginx:alpine && \
docker run --rm alpine wget -qO - -T10 -t1 192.168.56.20:10080
```

- oder alternativ mittels Container-Verlinkung:

```
docker run -d --name nginx nginx:alpine && \
docker run --rm --link nginx:web alpine sh -c \
 'wget -qO - -T10 -t1 $WEB_PORT_80_TCP_ADDR'
```

KONTINUIERLICHE SICHERHEITSANALYSE FÜR CONTAINER IMAGES

Sicherheitsanalyse

- automatisierte Analyse von Image-Inhalten anhand Liste bekannter Sicherheitslücken und Schwachstellen (Common Vulnerabilities and Exposures; CVE)
- unterschiedliche Techniken
  - statische vs. dynamische Analyse
- verschiedene Integrationsmöglichkeiten
  - manuelle Scans
  - Registry- und Build-Pipeline-Integration
  - Echtzeitbenachrichtigung

Kriterien zur Evaluierung von container-basierten Sicherheitsanalysewerkzeugen

- Technik
  - Wie werden in einem Image enthaltene Softwareversionen detektiert? (auf Basis des Paketmanagers oder binär)
  - Welche Quellen u. Datenbanken bekannter Sicherheitslücken werden genutzt?
  - Liegt der Analyseschwerpunkt eher auf anwendungs- oder docker-spezifischen Schwachstellen?
  - Inwiefern werden in Produduktiveinsatz befindliche Images kontinuierlich gescannt?
  - Welche Container-Technologien werden unterstützt?
- Integrierbarkeit
  - Wo ist der Ansatzpunkt des Scanners in der eigenen Pipeline?
  - Mit welchen Anwendungen arbeitet der Scanner zusammen (Registries, CI/CD-Lösungen, cloud-basierten / selbstbetriebenen Orchestrierungsplattformen)?
  - Wird neben der statischen Image-Analyse auch Auditing der Host-Umgebung unterstützt (Konfiguration, Container)?
  - Gibt es eine API für eine individuelle Integration (Erweiterbarkeit, Benachrichtigungsmechanismen)?

Sicherheitsanalysewerkzeuge

- Docker Bench for Security (Docker, Apache-Lizenz)
  - prüft Docker- und Container-Konfigurationen auf Einhaltung von "Best Practices" aufgestellt vom Zentrum für Internet Security (CIS)
- Clair (CoreOS; Apache-Lizenz)
  - detektiert installierte Software über bekannte Paketmanager
  - manuell installierte Komponenten bleiben unberücksichtigt
- Docker Security Scanning (Docker Inc.; kommerzielle Lizenz)
  - Erkennung wird auf Binär-Ebene durchgeführt, unabhängig vom Paketmanager
  - integriert in Docker Cloud und Docker Datacenter, jedoch nicht unabhängig einsetzbar
- Twistlock Trust (sowohl freie, als auch kommerzielle Lizenz)
  - Erkennung auf Binärebene
  - berücksichtigt Zero-Day-Feeds
- OpenSCAP + Atomic Scan (Red Hat; GPL3-Lizenz)
  - spezialisiert auf Red-Hat-Linuxdistributionen
- Bluemix Vulnerability Advisor (IBM; kommerziell)
  - integriert in cloud-basierte Plattform Bluemix

❖ siehe auch: https://www.alfresco.com/blogs/devops/2015/12/03/docker-security-tools-audit-and-vulnerability-assessment/

Docker Bench for Security

- Skript, das automatisiert auf Schwachstellen bzw. Einhaltung von "Best Practices" beim Produktiveinsatz von Docker prüft
- basiert auf dem Docker-Security-Benchmark des Center for Internet Security (CIS): https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.11.0_Benchmark_v1.0.0.pdf

```
sudo apt-get install auditd
cd ~
git clone https://github.com/docker/docker-bench-security.git
cd ~/docker-bench-security
sudo sh docker-bench-security.sh
```

- generierter Testbericht enthält Verweise auf die Kapitel im Dokument mit den entsprechenden Handlungsempfehlungen

Clair lokal aufsetzen


```
mkdir -p ~/clair/clair_config && \
cd ~/clair && \
curl -L https://raw.githubusercontent.com/coreos/clair/v1.2.6/docker-compose.yml -o docker-compose.yml && \
curl -L https://raw.githubusercontent.com/coreos/clair/v1.2.6/config.example.yaml \
 -o clair_config/config.yaml && \
sed -i.bak 's*    source:*    source: postgresql://postgres:password@postgres:5432?sslmode=disable*g' \
 clair_config/config.yaml && \
printf '\n' >> docker-compose.yml && \
sed -i.bak -e 's/^$/    restart: unless-stopped/g' docker-compose.yml && \
docker-compose up -d && \
sudo apt-get -y install golang-go && \
printf '\nexport GOPATH=/usr/share/go/\nexport PATH=$PATH:/usr/lib/go/bin\n' | sudo tee -a /etc/profile && \
sudo su -c 'go get -x -v -u github.com/coreos/clair/contrib/analyze-local-images' && \
docker-compose logs -f # wait until the vulerability database is updated
```

Clair mittels CLI einsetzen

- Workflow:
  1. mittels `docker pull` oder `docker build` Image lokal bereitstellen bzw. mittels `docker images` auffinden
  2. Einsatz: `analyze-local-images myimage`
  3. Recherche anhand des Berichtes durchführen
  4. Optionen:
     a) keine Korrektur, bei geringfügigem Risiko
     b) Image Maintainer um Korrektur bitten
     c) Schwachstelle selbst beheben und ggf. dem Maintainer rückmelden
     d) alternatives Image einsetzen
  5. Analyse regelmäßig wiederholen

Secret-Management

Wo können Geheimnisse aufbewahrt werden?

- im Docker Image
- in Umgebungsvariablen
- in Dateien innerhalb von Volumes
- in einem Secret Store

DATENSICHERUNG UND -WIEDERHERSTELLUNG

Datenpersistenz in Containern

- Volume-Arten: anonyme, host-gebundene und benannte Volumes
- wird ein *benanntes Volume* impliziert bei Container-Erzeugung angelegt werden vorhandene Dateien aus dem zugeordneten Verzeichnis des Image in das Volume kopiert
- wird bei Container-Erzeugung ein bestehendes Volume gemountet findet kein initialer Kopiervorgang statt
- mit dem Stopp eines Containers werden keine Inhalte gelöscht
- mit dem Löschen eines Containers gehen alle Inhalte des Container-Dateisystems verloren, Volumes bleiben jedoch erhalten
  - anonyme Volumes lassen sich anschließend mittels `docker volume ls` und `docker volume inspect VID` wieder auffinden und nachnutzen
  - wird `docker rm` mit dem Schalter `-v` ausgeführt, werden alle anonymen Volumes eines Containers gelöscht, es sei denn, sie sind mittels `--volumes-from` an weitere Container gebunden

Sicherungsmöglichkeiten

- Sicherung und Wiederherstellung von
  - Docker-Images
  - Docker-Containern
  - Dateien / Verzeichnissen in Containern ohne Volume
  - Dateien / Verzeichnissen aus anonymen Volumes
  - benannten Volumes
  - Datenbank-Dumps
    - manueller Backup-Container
    - kurzlebiger Backup-Container über cronjob auf dem Host
    - permanenter Backup-Container mit cron im Vordergrund

Sicherung und Wiederherstellung von Dateien in Containern ohne Volume

- Sicherungskopie des Ordners `/data` des Containers `SOURCE` im aktuellen Arbeitsverzeichnis des Hostsystems anlegen:

```
docker cp SOURCE:/data $(pwd)/data
```

- Kopie des Ordners `data` im aktuellen Arbeitsverzeichnis des Hostsystems in das Verzeichniss `/data` des `TARGET`-Containers:

```
docker cp $(pwd)/data TARGET:/data
```

- Nachteil: Applikation und Daten sind eng gekoppelt
- besser: Einsatz von Volumes vereinfacht Aktualisierung von Containern und schafft Transparenz

Sicherung und Wiederherstellung einzelner Verzeichnisse aus anonymen Volumes

- Sicherung des Ordners `/data` des Containers `SOURCE` als Archivdatei in das aktuelle Arbeitsverzeichnis des Hostsystems:

```
docker run --rm --volumes-from SOURCE:ro \
 -v $(pwd):/backup alpine \
 tar cvzf /backup/backup_$(date +%Y-%m-%d_%H%M).tar.gz /data
```

- Wiederherstellung des Ordners `/data` aus Archivdatei im aktuellen Arbeitsverzeichnis des Hostsystems in den Containers `TARGET`:

```
docker run --rm --volumes-from TARGET \
 -v $(pwd):/backup:ro alpine \
 tar xvf /backup/backup_2016-12-07_13-30.tar
```

- Nachteile: anonyme Volumes sind schwer zuzuordnen; Backup-Routine hat Zugriff auf alle Volumes
- besser: benannte Volumes

Sicherung und Wiederherstellung von benannten Volumes

- Sicherung aller Dateien des benannten Volumes **SOURCE** als Archivdatei in das aktuelle Arbeitsverzeichnis des Hostsystems:

```
docker run --rm -v SOURCE:/data:ro \
 -v $(pwd):/backup alpine \
 tar cvzf /backup/backup_$(date +%Y-%m-%d_%H%M).tar.gz /data
```

- Wiederherstellung aller Dateien aus Archivdatei im aktuellen Arbeitsverzeichnis des Hostsystems in das Volume **TARGET**:

```
docker run --rm -v TARGET:/data \
 -v $(pwd):/backup:ro alpine \
 tar xvf /backup/backup_2016-12-07_13-30.tar.gz
```

- Nachteile: Volume-Namensschema bei Skalierung?

Backup-Beispielcontainer

- tutum/mysql-backup:

```
docker run -d \
    --env MYSQL_HOST=mysql.host \
    --env MYSQL_PORT=27017 \
    --env MYSQL_USER=admin \
    --env MYSQL_PASS=password \
    --volume host.folder:/backup
    tutum/mysql-backup
```

- spoqa/postgresql-backup
- osixia/openldap-backup

PROTOKOLLIERUNG, ÜBERWACHUNG, BENACHRICHTIGUNG

Ziele

- Zuverlässigkeit bereitgestellter Dienste gewährleisten
  - Ausfallzeiten reduzieren
  - Softwarefehler frühzeitig erkennen
  - auf Performance-Engpässe reagieren
  - Dateneingaben auditieren
- Einbruchsschutz
  - Schwachstellen identifizieren
  - Angriffe frühzeitig erkennen
  - Schadensausmaß bestimmen / reduzieren
- Prozessoptimierung
  - Anwenderverhalten auswerten

Möglichkeiten der Protokollierung

- `docker logs` ist die einfachste Lösung, hat jedoch für den Produktiveinsatz einige Nachteile:
  - kann nur mit STDOUT- und STDERR-Ausgaben umgehen
  - Protokolle gehen mit dem Entfernen eines Containers verloren
  - das intern genutzte JSON-Format ist speicherintensiv, erschwert einfache Suchen und zerlegt Stacktraces zeilenweise in mehrere Abschnitte
  - unterstützt keine Log-Rotation
- Log-Dateien in ein Docker Volume schreiben
- spezialisierte Logging-Dienste einsetzen
  - z.B. syslog-ng, rsyslog, logstash, logspout, filebeat, fluentd
- Komplettlösungen:
  - ELK-Stack: Elasticsearch, Logstash, Kibana
  - Logentries (kommerzielle Lizenz)

Welche Protokolle werden docker-intern angelegt

- Docker-Daemon-Logs
  - abhängig von der Linux-Distribution
  - für Debian / Ubuntu aktuell:

```
journalctl -u docker -n100
```

- Container-Logs
  - für alle Container anzeigen:

```
sudo su -c 'ls -Ralph /var/lib/docker/containers/*/*.log'
```

Container-Protokollierung mittels logrotate begrenzen

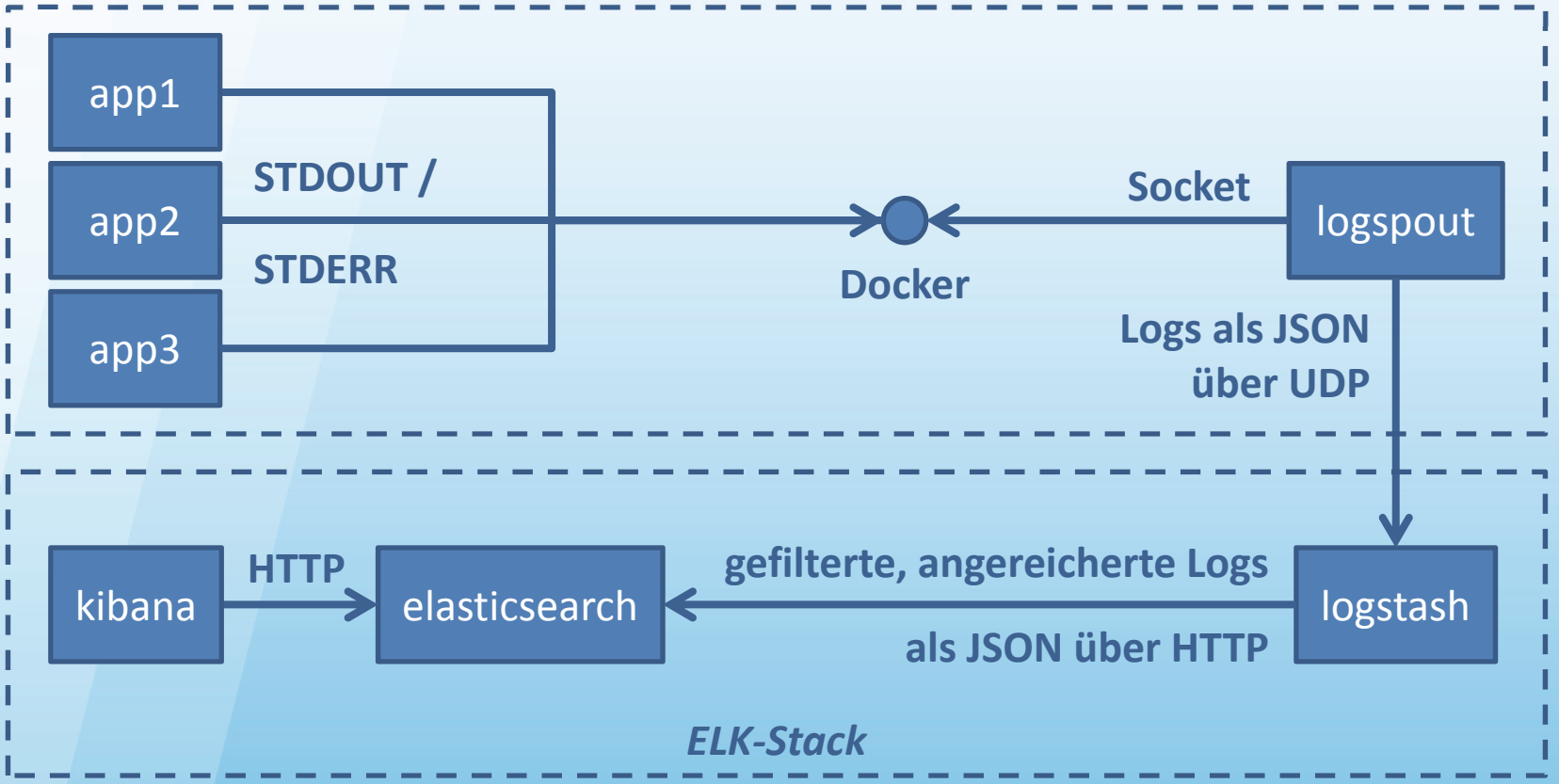
- `/etc/logrotate.d/docker` anlegen:

```
/var/lib/docker/containers/*/*.log {
  daily
  rotate 5
  compress
  delaycompress
  missingok
  copytruncate
}
```

- täglicher Cronjob meist bereits vorkonfiguriert (siehe `/etc/cron.daily/logrotate`; `/etc/crontab`)

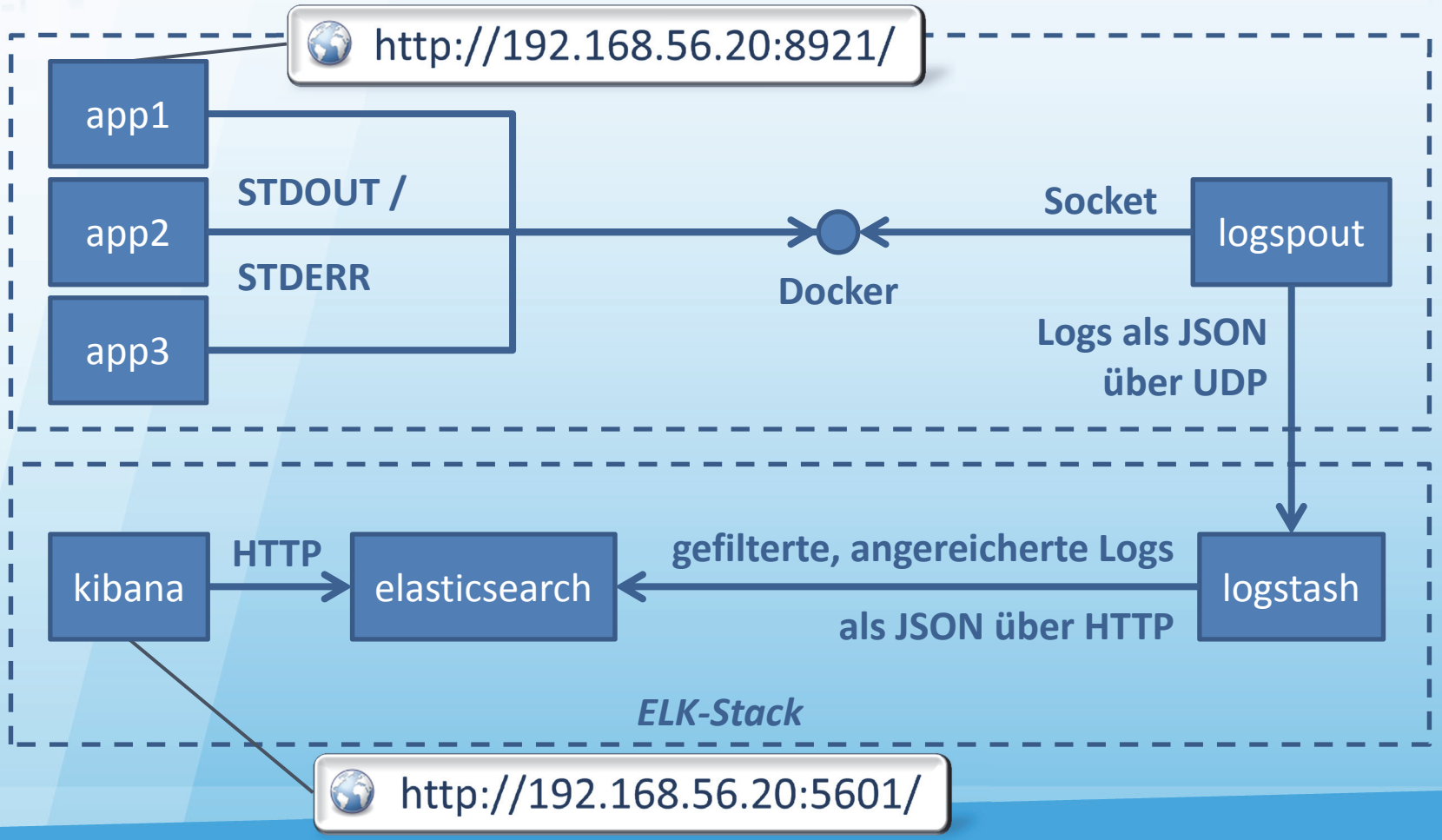
Der ELK-Stack

```
cd ~/elk && docker-compose up -d
```



Der ELK-Stack

```
cd ~/elk && docker-compose up -d
```



Welche Metriken überwachen?

- Anwendungsmetriken
  - Anmeldeversuche, Nutzersitzungen, Threads, Transaktionen, Entities; Response Time, Health Check
- Containermetriken
  - Container CPU / Limits
  - Memory Limits + Allocation Fail Counters
  - Disk I/O
  - Network I/O + Network Errors
  - Events: create, destroy, die, export, kill, pause, restart, start, stop, unpause, oom
  - Logs
- Servermetriken
  - CPU
  - Speicher
  - Festplatte

Überwachung von Produktivsystemen

- docker-spezifische Lösungen:
  - docker stats + API, Google cAdvisor, Prometheus
- herkömmliche Überwachungswerkzeuge:
  - Nagios, Xymon, PRTG, Zabbix
- kombinierte Lösungen:
  - TICK-Stack: Telegraf, InfluxDB, Chronograf, Kapacitor
- eigene HTTP-Endpunkte mit weiteren Parametern
  - Anmeldeversuche, Nutzersitzungen, Threads, Transaktionen, Entities; Response Time, Health Check, …

Monitoring mit docker stats

- Echtzeitstatistik auf der Konsole:

```
docker stats $(docker ps -a --format={{.Names}})
```

Monitoring mit Google cAdvisor


```
docker run -d --name cadvisor -p 7777:8080 \
 -v /:/rootfs:ro \
 -v /var/run:/var/run:rw \
 -v /sys:/sys:ro \
 -v /var/lib/docker/:/var/lib/docker:ro \
 google/cadvisor
```

- im Browser:

http://192.168.56.20:7777/

Monitoring mit Prometheus

- Konfigurationsdatei `~/prometheus/conf.yml`:

```
global:
  scrape_interval: 1m
  scrape_timeout: 10s
  evaluation_interval: 1m

scrape_configs:

- job_name: prometheus
  scheme: http
  static_configs:
  - targets: ['192.168.56.20:9090']
```

Monitoring mit Prometheus

```
docker run -d -p 9090:9090 --name prometheus \
 -v $HOME/prometheus:/conf:ro \
 -v prom-data:/prometheus \
 prom/prometheus \
 -config.file=/conf/conf.yml
```

- im Browser:

http://192.168.56.20:9090/

Prometheus

An open-source service monitoring system and time series database.

FAZIT

Fazit

- früh starten Sicherheitsaspekt einzubeziehen
- Sicherheitsanalysewerkzeuge einsetzen
- Schritt für Schritt zur eigenen Continuous Delivery Pipeline

Weiterführende Literatur

- Adrian Mouat: *Docker – Software entwickeln und deployen mit Containern.* dpunkt.verlag 2016, ISBN 978-3-86490-384-7
- Adrian Mouat: *Docker Security. Using Containers Safely in Production. Second Release.* O'Reilly Media 2016. *http://www.oreilly.com/webops-perf/free/docker-security.csp*
- *Awesome Docker.* https://veggiemonk.github.io/awesome-docker/
- *Docker Cheat Sheet.* http://docker.jens-piegsa.com
- Docker + AppArmor. https://docs.docker.com/engine/security/apparmor/
- Docker + SELinux. http://www.projectatomic.io/docs/docker-and-selinux/
- Docker + Seccomp. https://docs.docker.com/engine/security/seccomp/

- *Docker-Illustrationen mit freundlicher Genehmingung von Docker, Inc.*

Kontakt & Fragen

- per E-Mail an:
  jens.piegsa@uni-greifswald.de