Docker Security Workshop

Goals of this Workshop

Understand and get comfortable with Docker security technologies

Swarm Mode Security
Secrets Management
Security Scanning
Content Trust
Networking
...

Understand and get comfortable with Linux security technologies

AppArmor
seccomp
Capabilities
...

Agenda

Setting the Scene	1. Docker Security Pillars 2. Anatomy of a Container 3. Docker Client and Daemon
Docker Security Technologies	1. Trusted Code Deployment with Docker Content Trust 2. Strong Vulnerability Detection with Docker Security Scanning 3. Secure Orchestration by Default with Swarm Mode 4. Secure App-centric Networking with Docker Overlay Networks 5. Container Native Secrets Management with Docker Secrets
Linux Security Technologies	1. User Management 2. AppArmor 3. seccomp 4. Capabilities

Setting the Scene

Docker Security Pillars

The Three Pillars of Docker Security

Usable Security

Trusted delivery

Infrastructure Independent

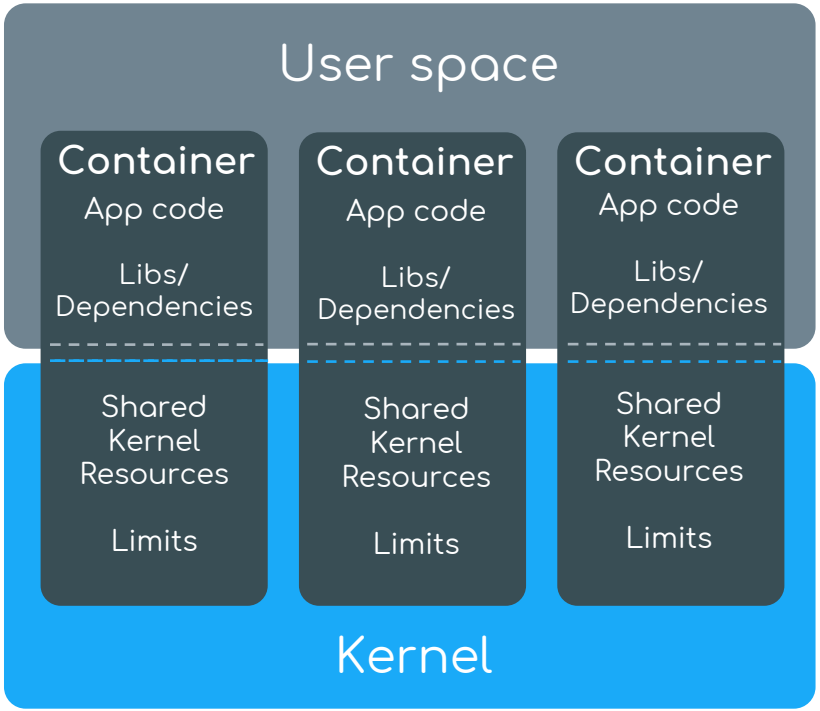
Docker Security: Aim of the Game

Secure by default

Sensible defaults
configured
out-of-the-box (OOB)

Anatomy of a Container

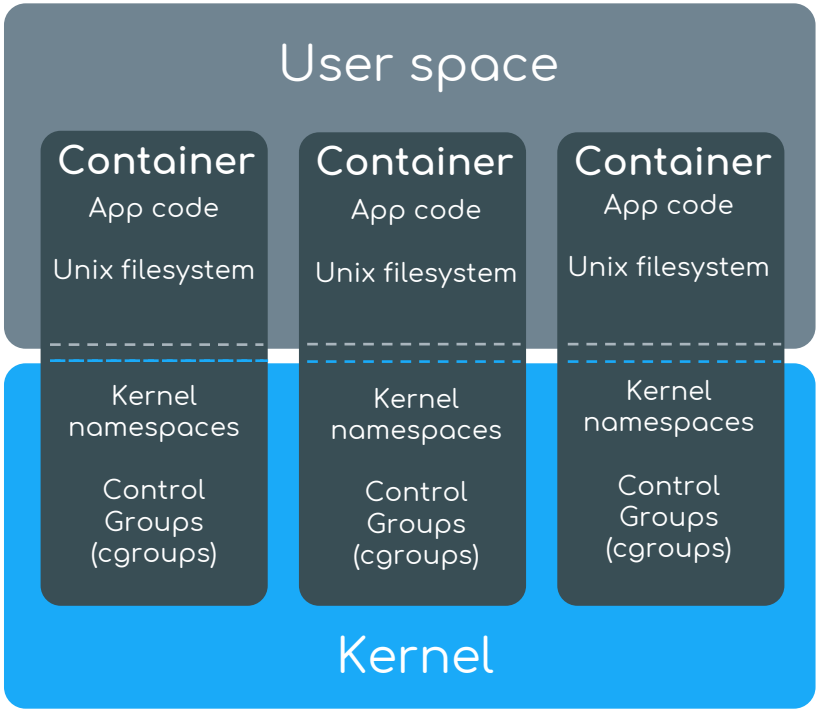
Containers: The Big Picture



User space
- Libraries
- Binaries
- Other dependencies

Ring-fenced area of OS/kernel:
- Process tree
- Filesystem root
- Network stack
- ...
- Limits on resource consumption

Containers: Linux Kernel Features



Namespace examples:

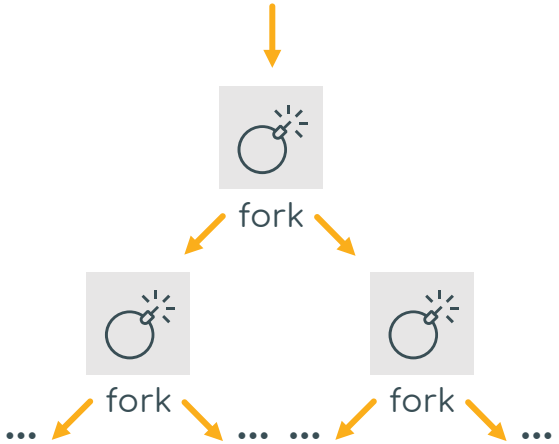
- The PID namespace stops processes in one container from seeing and interacting with processes in another container (or on the host)
- The User namespace allows containers to run processes as root inside the container but as non-privileged users outside the container (on the host)

Control Groups examples:

- Can limit the amount of CPU or memory a container can use, and prevent them from consuming all system resources

Containers: Protection Against Fork Bombs

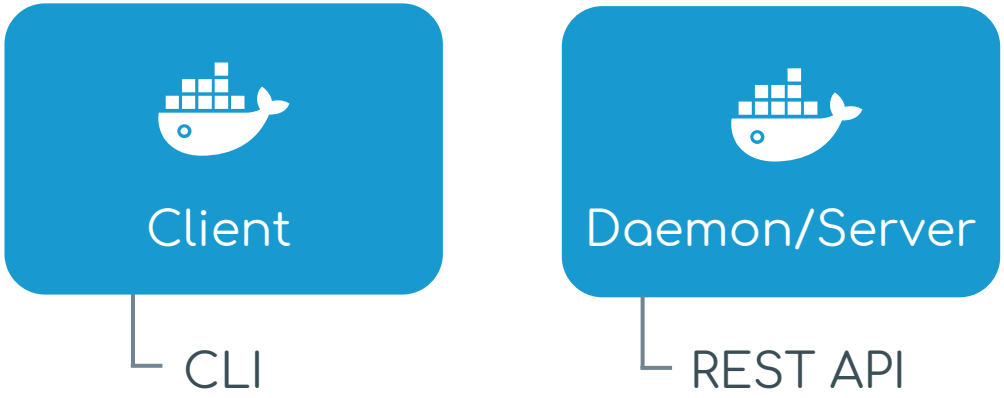
```
:(){ :|: & };:
```



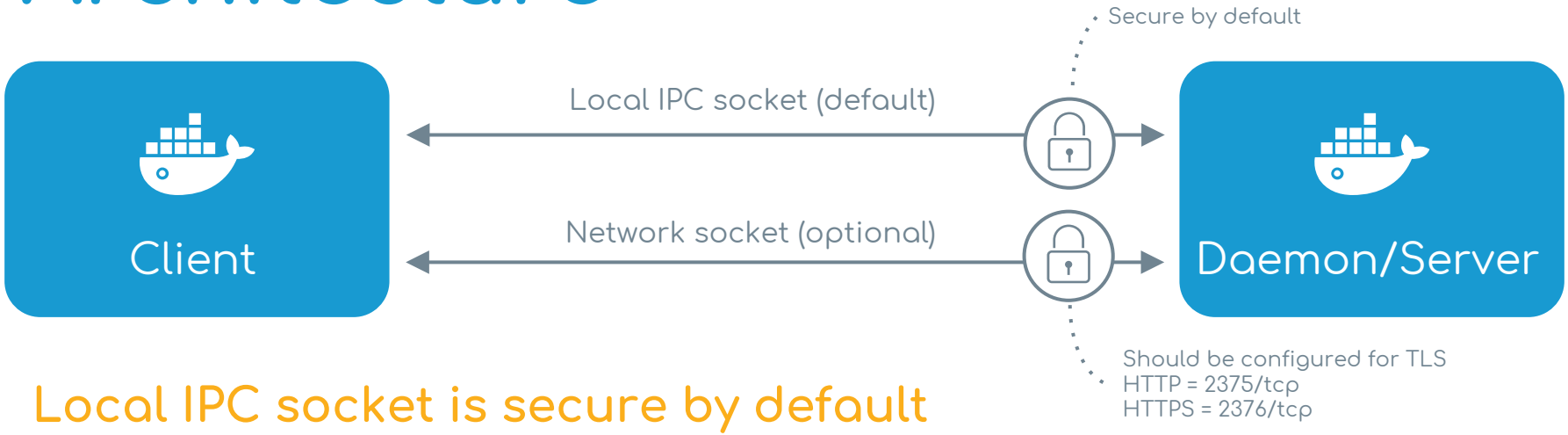
Docker Client and Daemon

The Basics

Docker: Client-server Architecture



Docker: Client-server Architecture

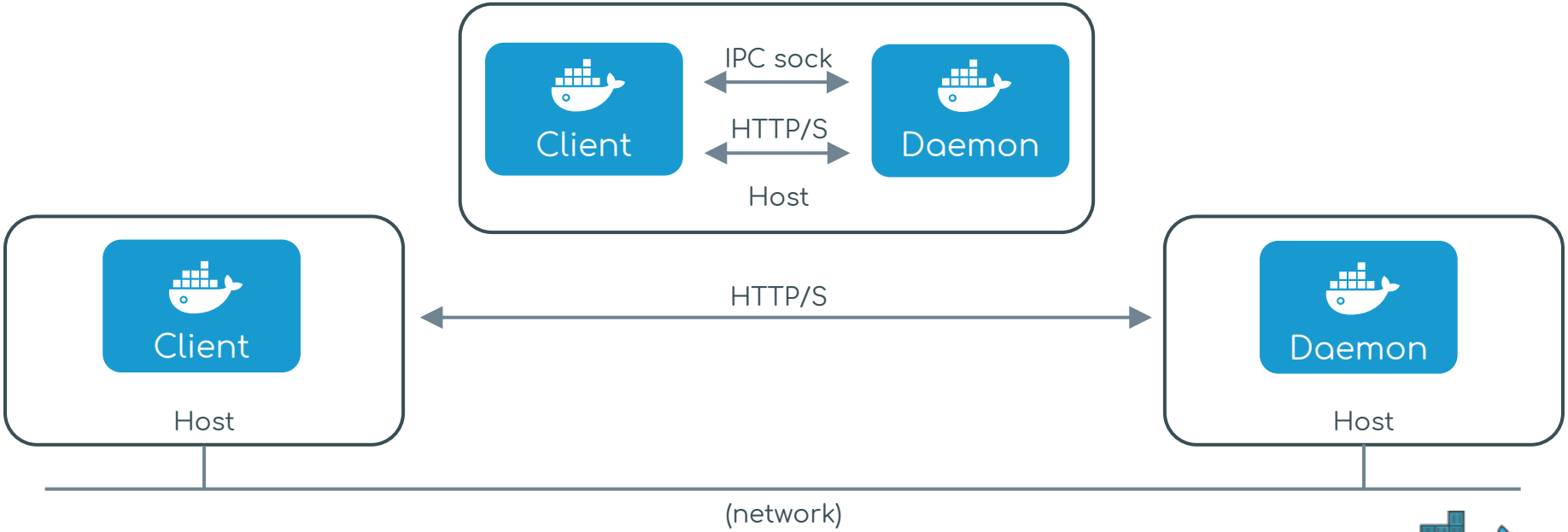


Local IPC socket is secure by default

Manual configuration required to secure the network socket

- Client mode: Client will only talk to authenticated daemons
- Daemon mode: Daemon will only talk to authenticated clients

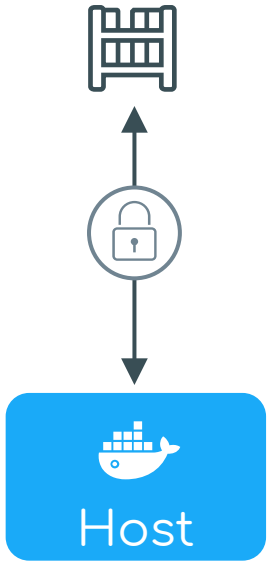
Docker: Client-server Architecture



Connecting Securely to Docker Registries

Can use TLS to secure (authenticate and encrypt) traffic between Docker and Docker Registry:

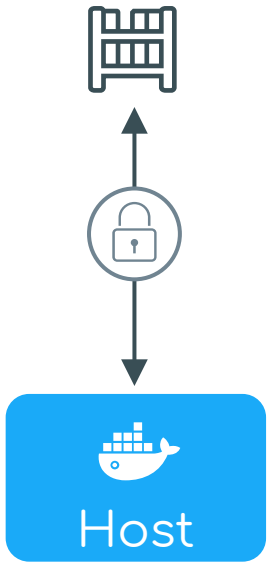
- Create a directory under /etc/docker/certs.d for the Registry
- Include client key and client certificate
- Include CA certificate



Connecting Securely to Docker Registries

```
/etc/docker/certs.d/registry.corp.internal/
  client.cert
  client.key
  ca.crt
```

- If the Registry is accessed over a specific port you must include the port in the directory name. E.g. `/etc/docker/certs.d/registry.corp.internal:5000`
- Docker expects CA certificates to have a **.crt** extension and client certificates **.cert**



Q&A

Docker Security Technologies

Trusted Code Deployment

With Docker Content Trust

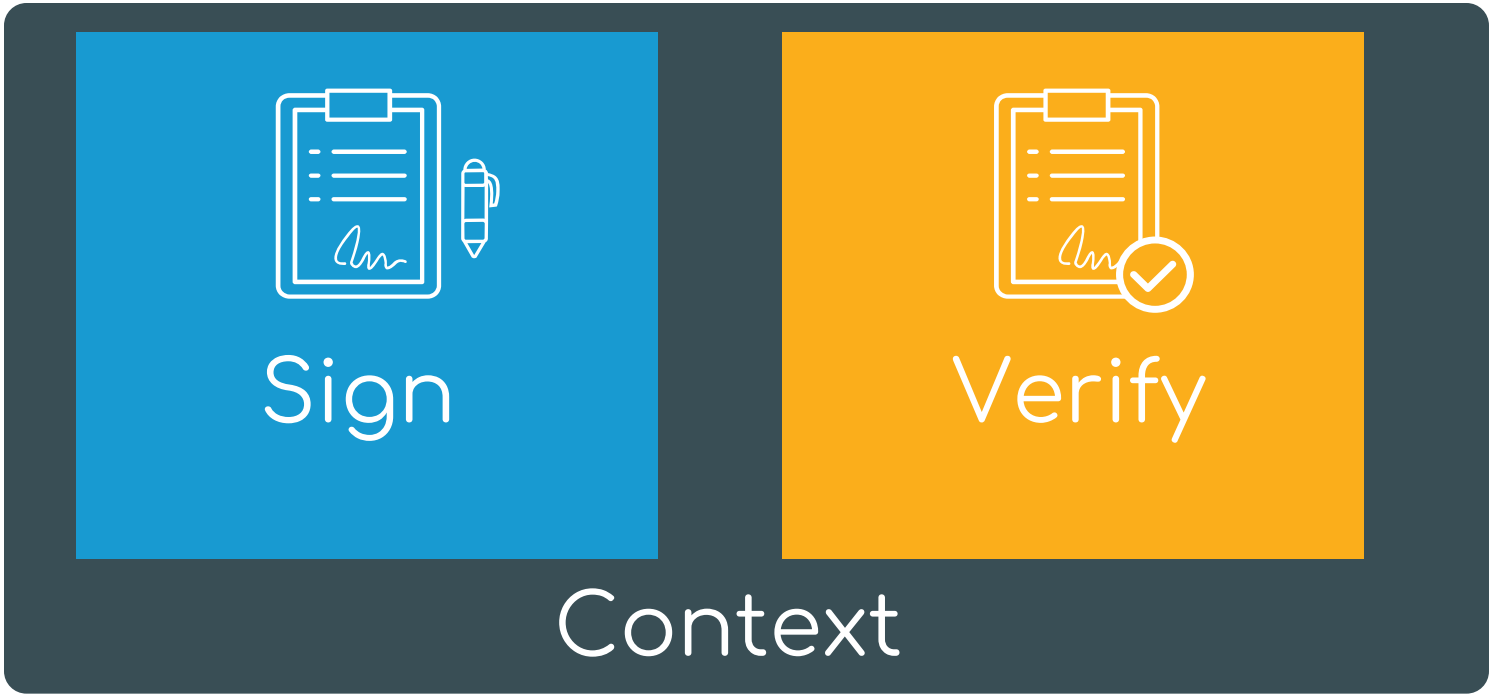
Background: Trust is Vital!

Applications are vital to businesses

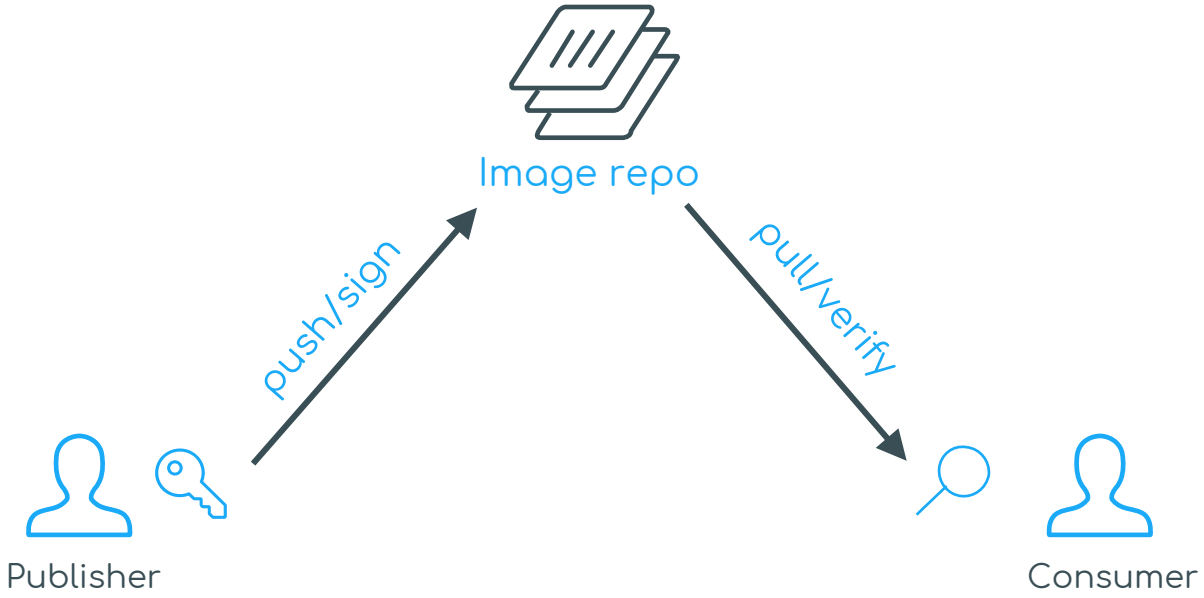
Untrusted networks like the internet are like the Wild West

Goal: Make it simple to verify and trust the software you deploy

The Big Picture



Docker Content Trust: Pushing and Pulling

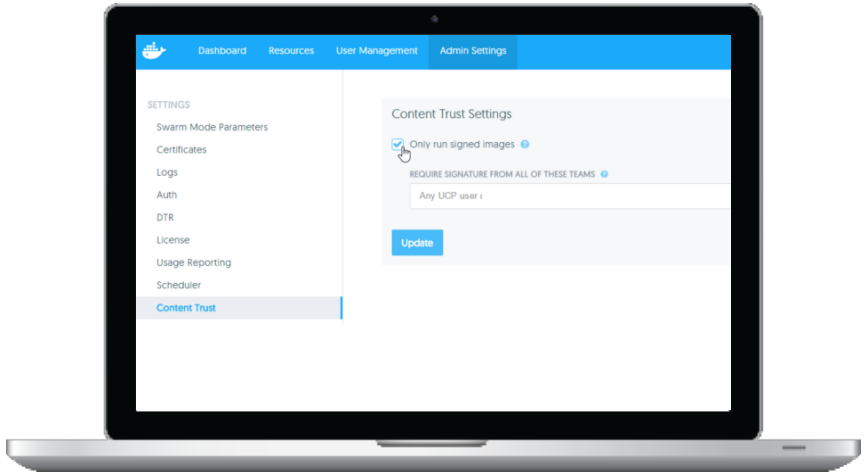


Docker Content Trust Provides...

Docker Content Trust: Easy to Enable

```
$ export DOCKER_CONTENT_TRUST=1
```

Docker Content Trust: Enable in UCP



Cluster-wide

push

pull

build

run

docker17 con

Docker Content Trust: Unsigned Images

```
$ docker pull repo/image:unsigned
...
Error: No trust data for unsigned
```

Docker client

Error creating service x
image did not meet required signing policy

Universal Control Plane Web UI

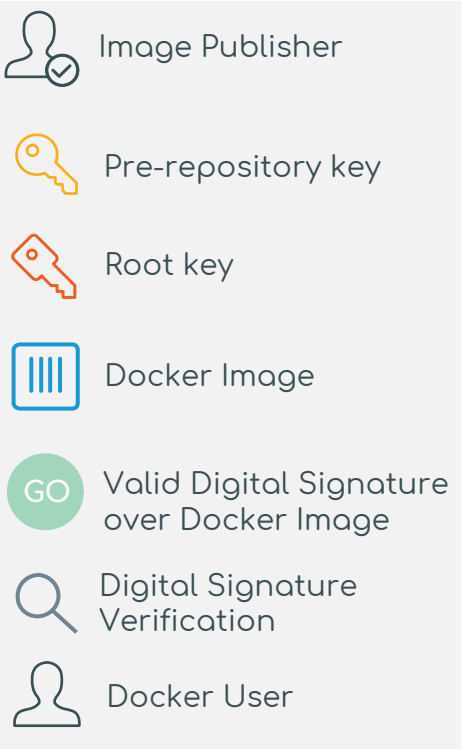
Docker Content Trust: Malicious Images

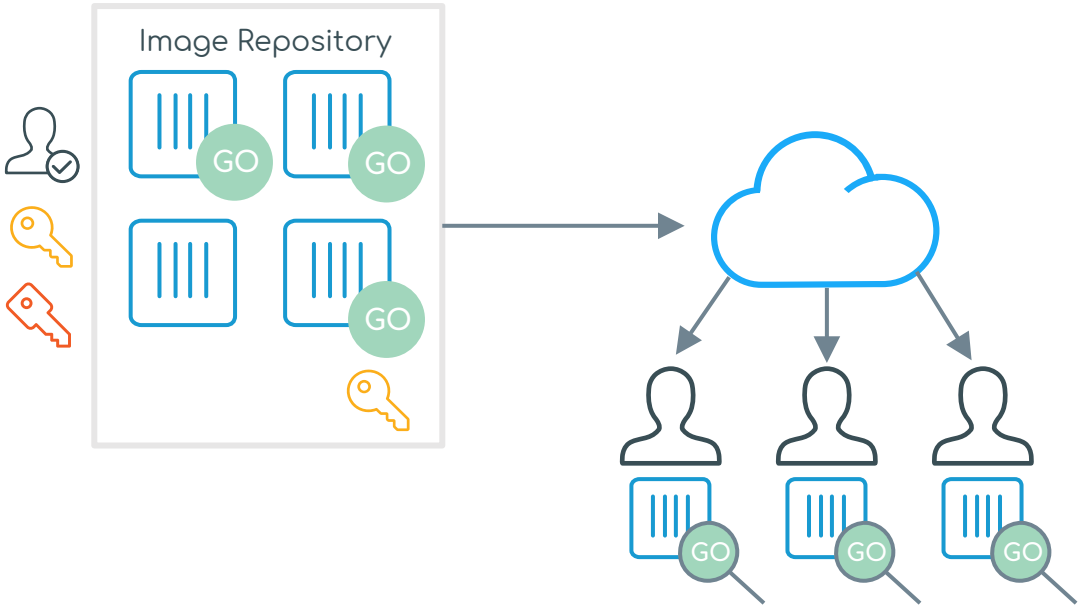
```
$ docker pull repo/image:fakesignature

Warning: potential malicious behavior - trust data has insufficient
signatures for remote repository docker.io/repo/image: valid signatures
did not meet threshold
```

Docker Content Trust: Stale Images

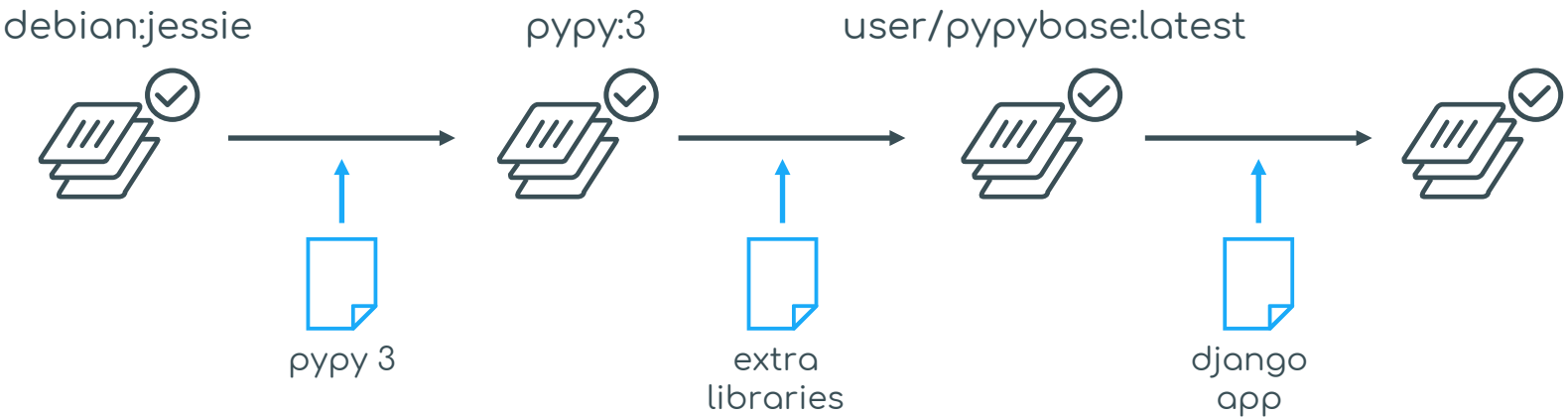
```
$ docker pull repo/image:stale

Error: remote repository docker.io/repo/image out-of-date: targets
expired at Sun Mar 26 03:56:12 PDT 2017
```

Docker Content Trust: How it Works





Docker Content Trust: Signing the Entire Chain



Docker Datacenter: Taking DCT to the Next Level

Built-in Notary Server

Simplifies deployment
Integrates with Docker Trusted Registry (DTR)

Notary is a client-server app that implements The Update Framework (TUF) that underpins Docker Contents Trust

- Publishes and manages your trusted collections
  - Delegations
  - Freshness
  - Trust thresholds
  - Survives key compromise

Docker Datacenter: Taking DCT to the Next Level

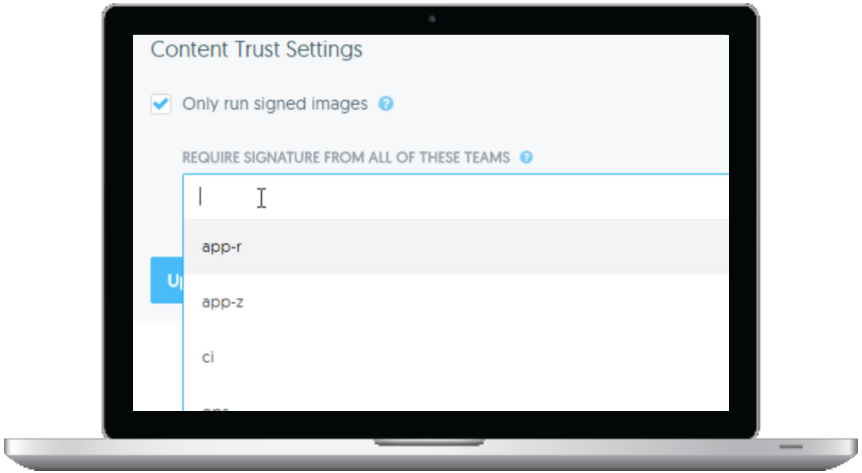
Built-in Notary Server

Simplifies deployment
Integrates with Docker Trusted Registry (DTR)

Simple Trust Thresholds

Choose UCP users and teams as authorized signers

Docker Datacenter: Taking DCT to the Next Level



Universal Control Plane web UI

- Easily create a list of required signers

Image Best Practice: Use Official Images and Use Small Images

Use minimalist base images

- Smaller images reduce the attack surface
- The official Alpine base image is <5MB‘

Use official images as base images

- All official images are scanned for vulnerabilities
- Usually follow best practices

Image Best Practice: Use Official Images and Use Small Images

Pull images by digest

- Image digests are a hash of the image's config object
  - This makes them immutable
  - If the contents of the image are changed/tampered with, the digest will be different

```
$ docker pull alpine@sha256:3dcdb92...b313626d99b889d0626de158f73a

sha256:3dcdb92d7432d...e158f73a: Pulling from library/alpine
e110a4a17941: Pull complete
Digest: sha256:3dcdb92d7432d56604...47b313626d99b889d0626de158f73a
Status: Downloaded newer image for alpine@sha256:3dcd...b889d0626de158f73a
```

If Docker Content Trust is enabled all images are automatically pulled by digest

Q&A

Lab

Enabling and Testing Docker Content Trust

Strong Vulnerability Detection

With Docker Security Scanning

What Security Scanning

Tool/service that scans images for vulnerabilities

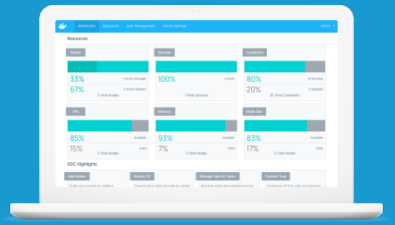
- Operates in the background
- Performs **deep** binary-level scanning of image layers
- Checks against database(s) of known vulnerabilities
- Provides detailed vulnerability report

Helps protect software and achieve software compliance

Security Scanning Offerings

Hosted

Available for **private repositories** on Docker Hub and Docker Cloud



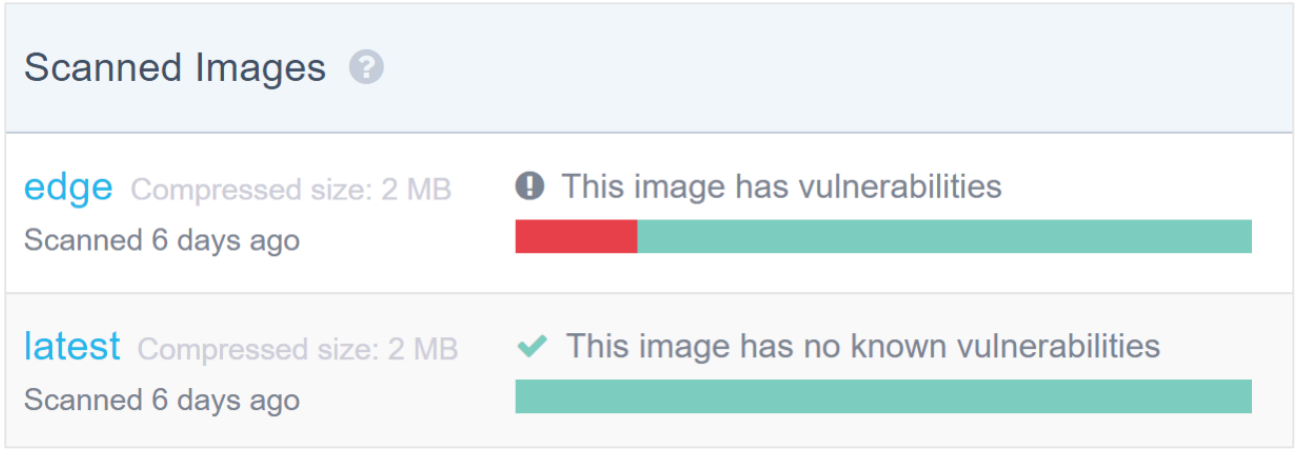
On premises

Available as part of Docker Datacenter

Security Scanning: Vulnerability Reports

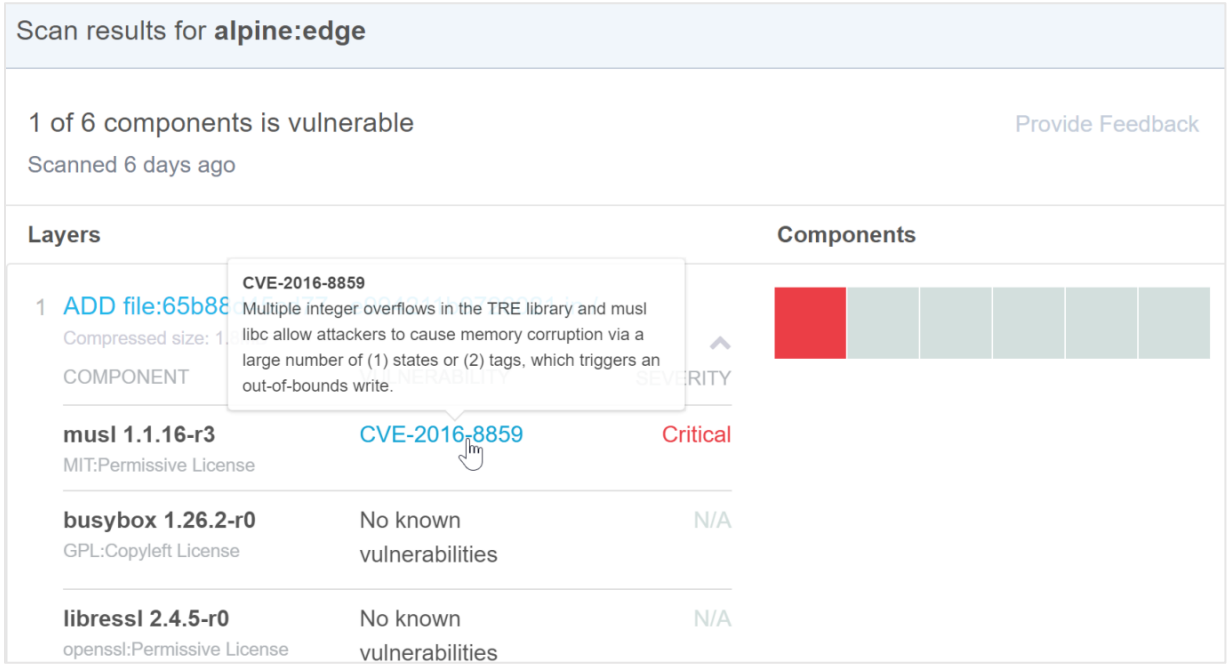
alpine:edge

alpine:latest

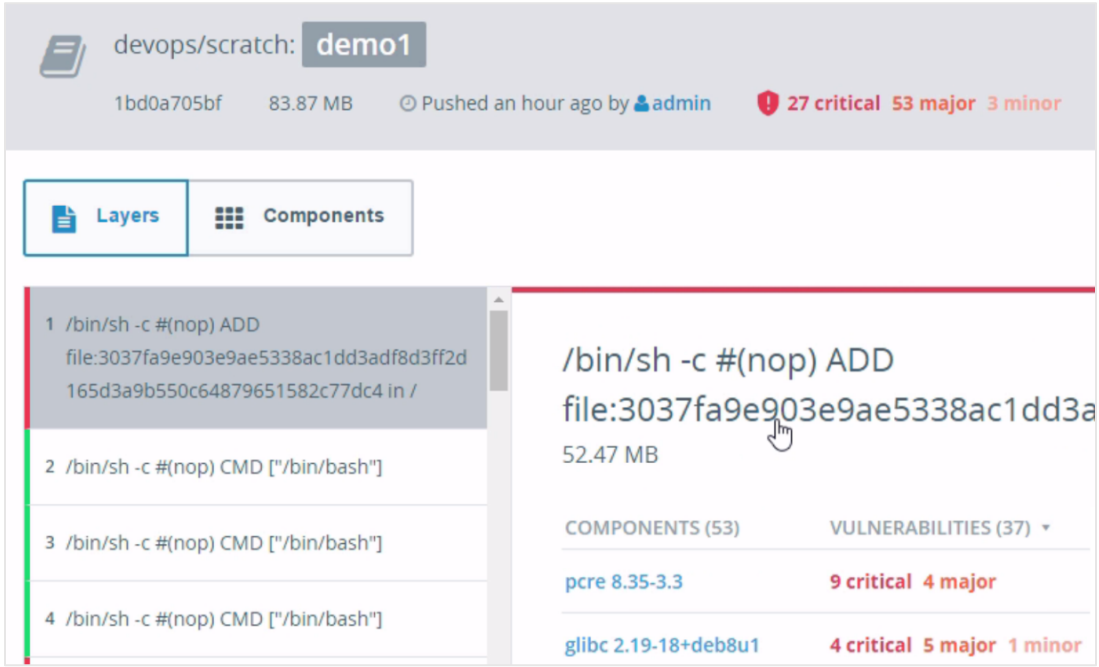


Useful high-level reports

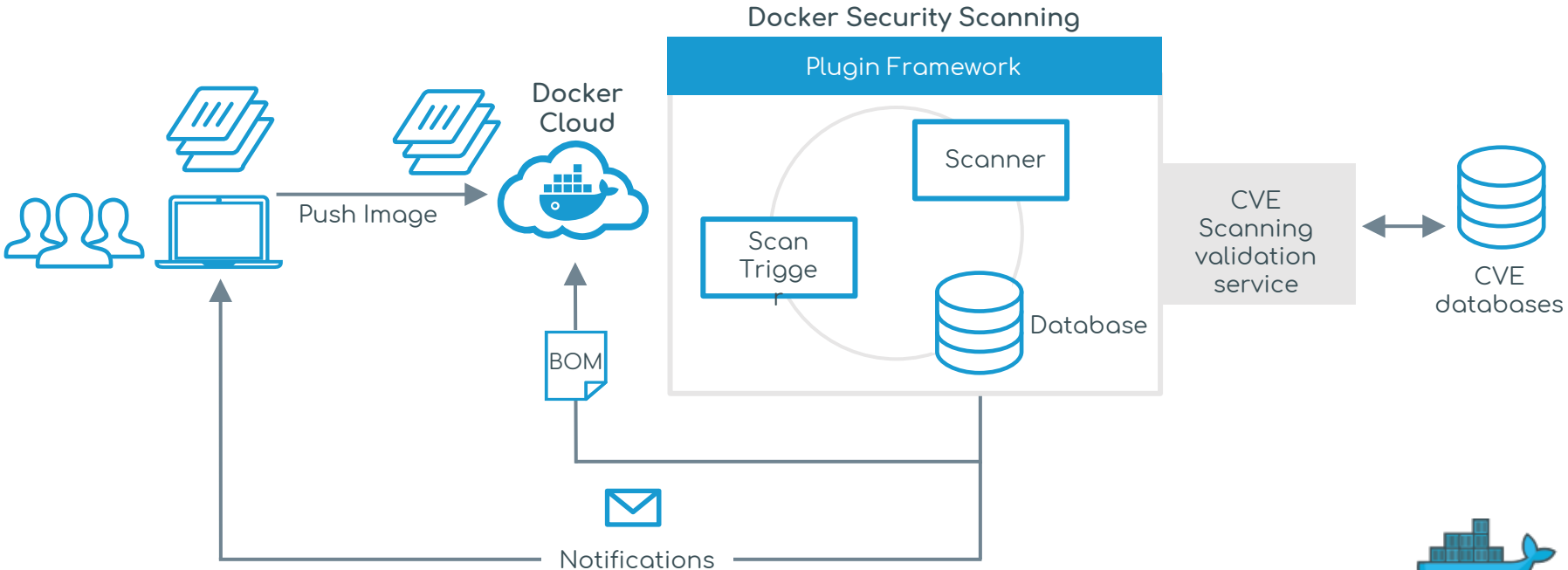
Security Scanning: Vulnerability Reports



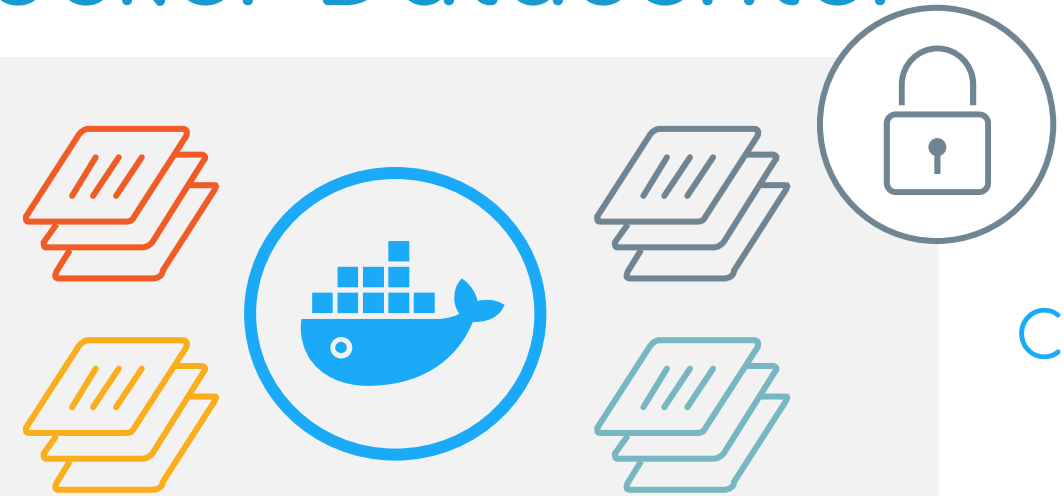
Security Scanning: DDC/DTR Vulnerability Reports



Security Scanning: How it Works



Security Scanning with Docker Datacenter



Docker Trusted Registry (DTR)
On premises

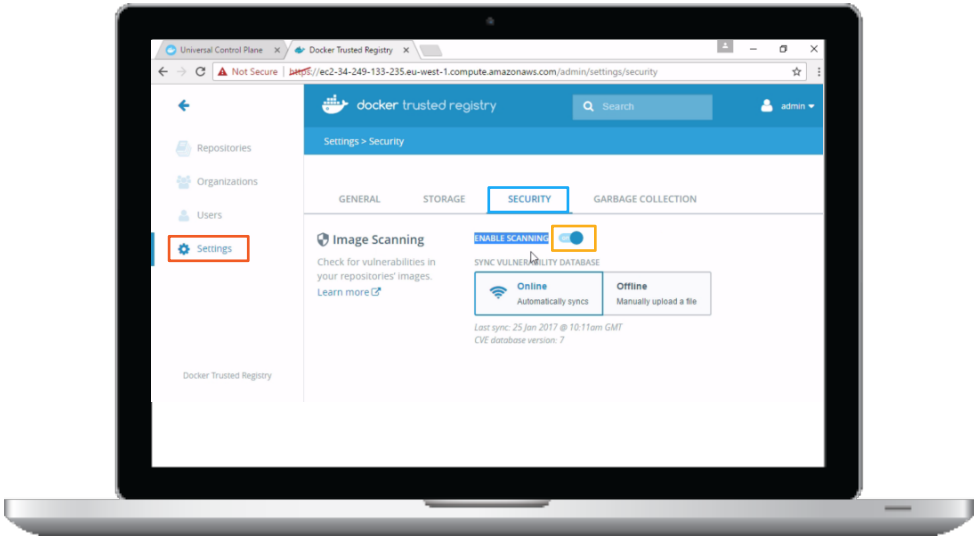
Configured via DTR

Security Scanning with Docker Datacenter

Click
<Settings>

Click
<SECURITY>

Click <on>

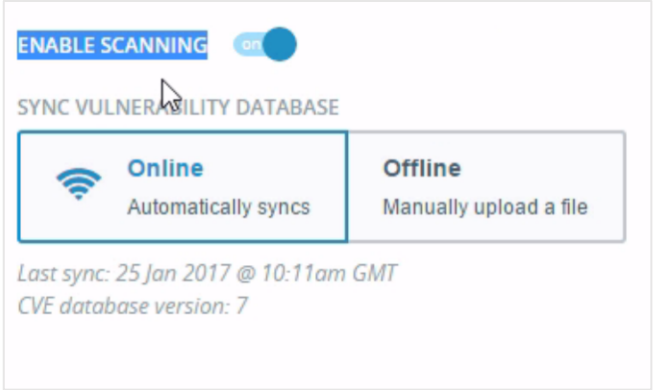


Security Scanning with Docker Datacenter

Online: Will automatically sync the vulnerability database over the internet

Offline: Will not update vulnerability database over the internet. Allows admins to manually upload .tar files.

The offline method is ideal for security sensitive scenarios where DTR and other systems are air-gapped from the internet

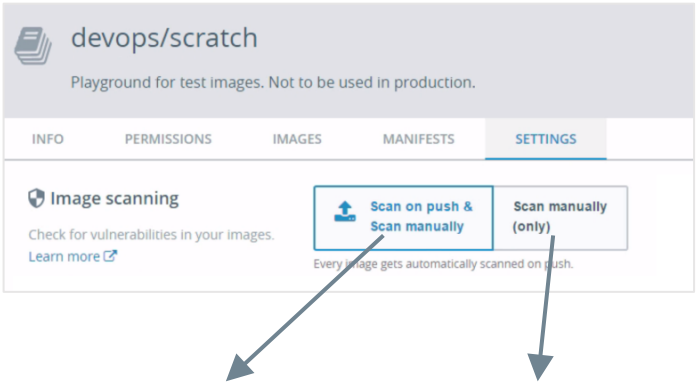


Security Scanning with Docker Datacenter

Scanning configured on a per-repo basis

Default is to scan every new image that is pushed

Can configure a repo to only support manual scans (if you don't want to trigger a scan every time an image is pushed)



Scan on every **push**

Do not scan on every **push**

Security Scanning: Summary

Official repos/images automatically scanned

Binary level scans pick up statically linked bins

Checks against CVE databases

Provides comprehensive bill of materials (BOM)

Q&A

Lab

Testing Security Scanning

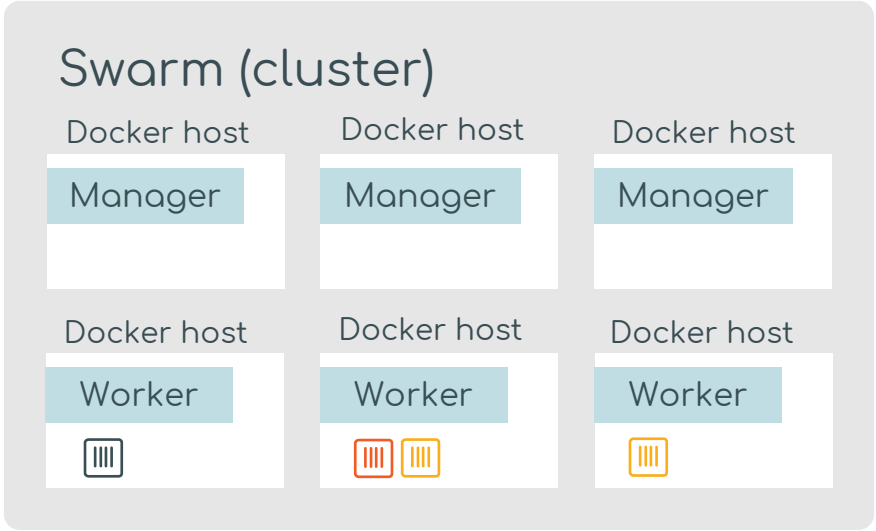
Secure Orchestration by Default

With Swarm Mode

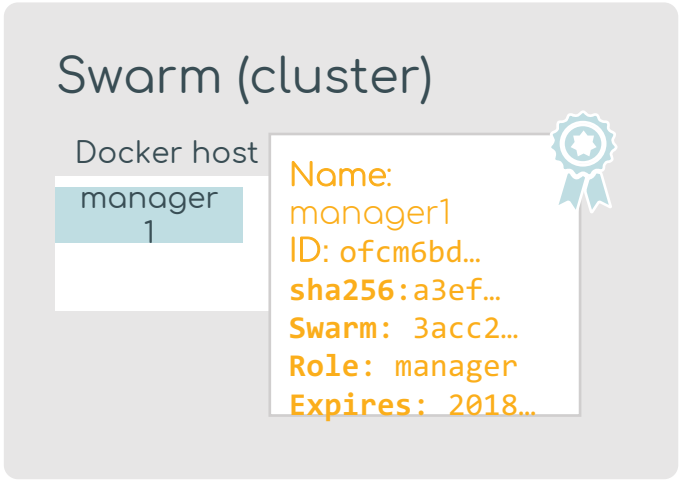
Swarm Mode: Overview

Native clustering of Docker Hosts

- One or more **Managers** (control plane)
- One or more **Workers** (data plane)
  - Run user workloads
- Strong default security (out-of-the-box)



Swarm Mode: Client Certificates

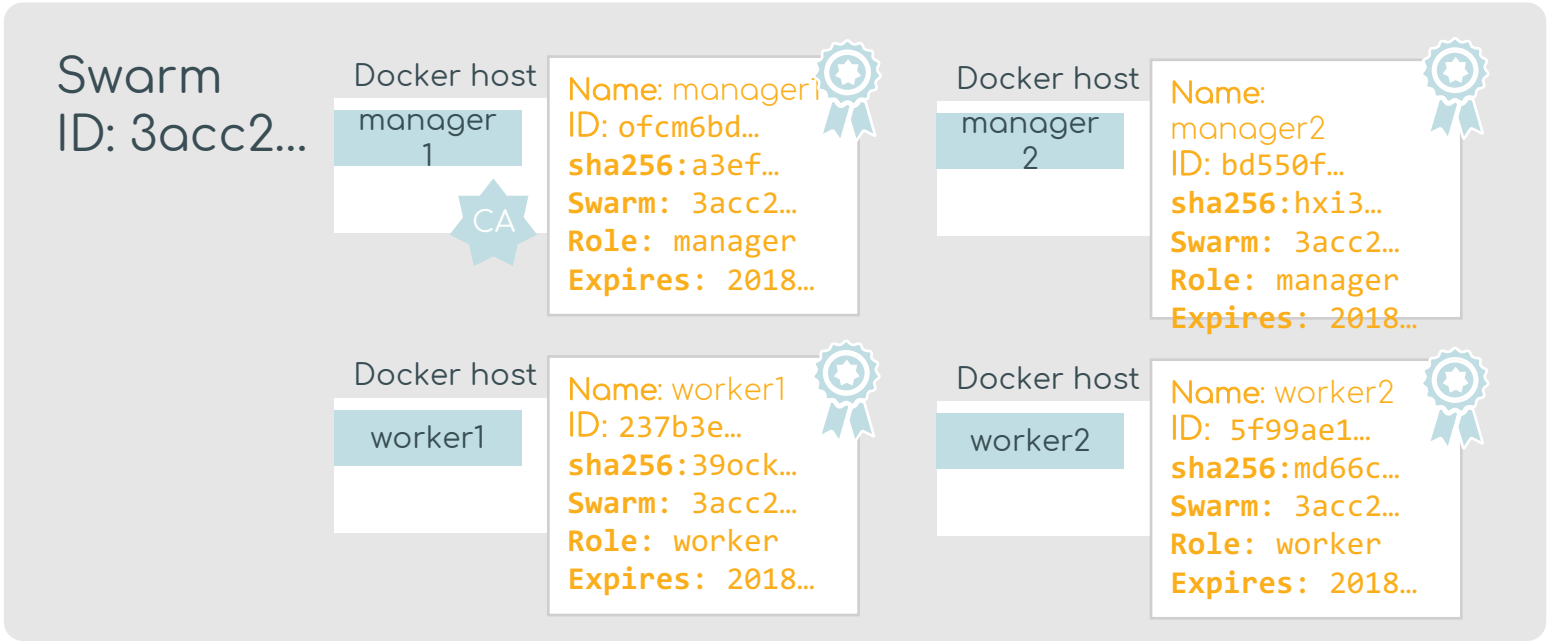


Every node gets a Client cert that identifies:

- The node
- The Swarm that it's a member of
- Its role in the Swarm

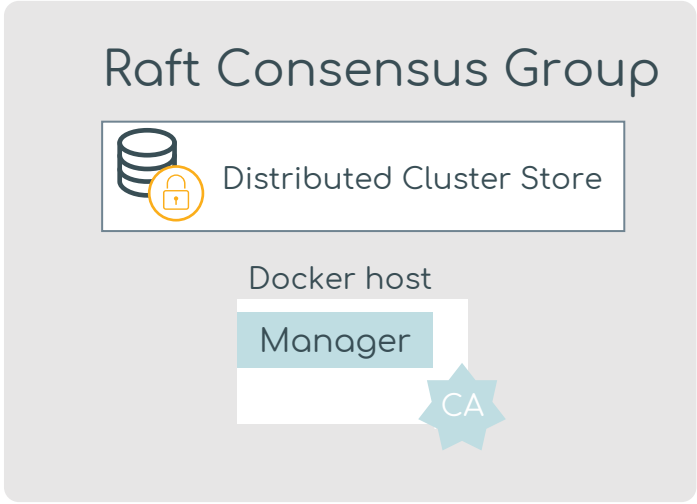
docker17 con

Swarm Mode: Cryptographic Guarantees



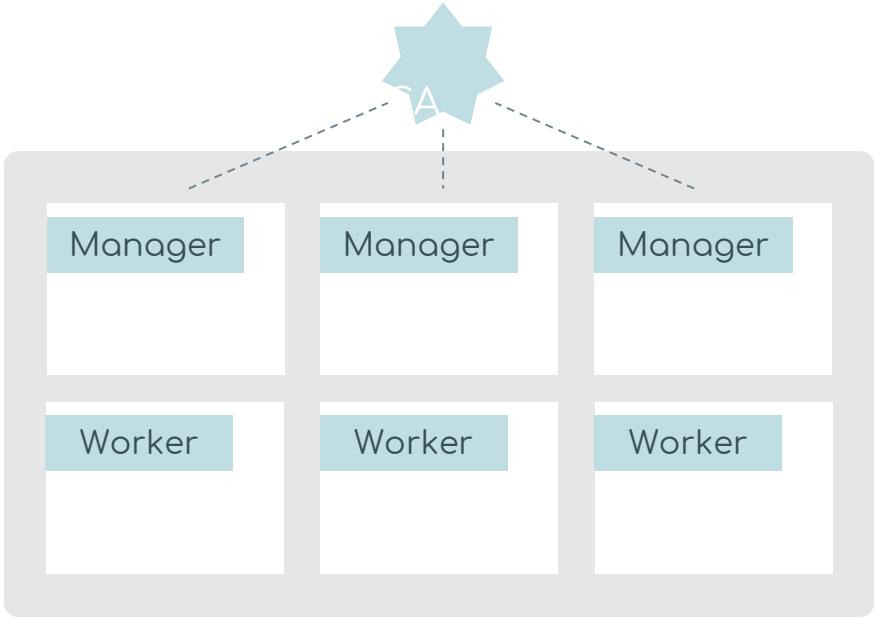
Creating a New Swarm

```
$ docker swarm init

Swarm initialized: current node
(ofcm6bdy5qcrlievawsw9wqfp) is now a manager.

To add a worker to this swarm, run the following
command:

    docker swarm join \
    --token SWMTKN-1-
31fxss83n3puc6bd11wm8vxged2ul94fxfbckjdy0rj37agk
ko-bz14m6jyeakhzvccs7wnbmmof \
    172.31.45.44:2377

To add a manager to this swarm, run 'docker
swarm join-token manager' and follow the
instructions.
```



Using and External Root CA

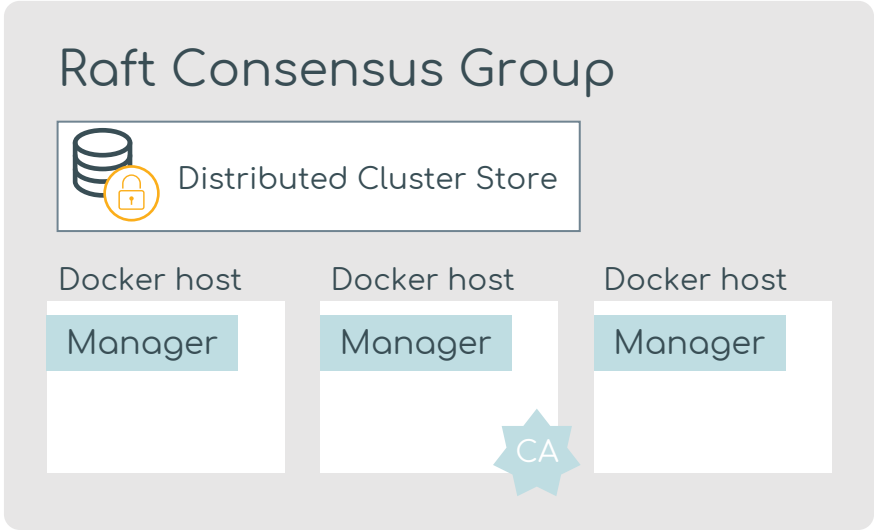
- Swarm supports using external CAs
- Pass the `--external-ca` flag to the docker `swarm init` command



Adding More Managers

```
$ docker swarm join-token manager

To add a manager to this swarm, run the
following command:

    docker swarm join \
    --token SWMTKN-1-31fx-8z0l... \
    172.31.45.44:2377
```

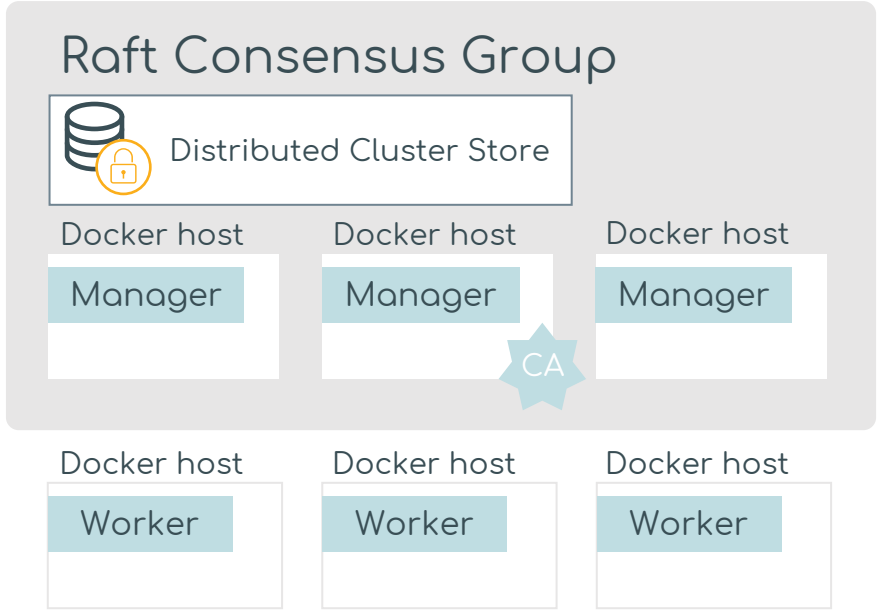
```
$ docker swarm join \
>  --token SWMTKN-1-31fx-8z0l... \
>  172.31.45.44:2377

This node joined a swarm as a manager.
```



Adding Workers

```
$ docker swarm join-token worker

To add a worker to this swarm, run the
following command:

    docker swarm join \
    --token SWMTKN-1-31fx-bz14... \
    172.31.45.44:2377
```

```
$ docker swarm join \
>  --token SWMTKN-1-31fx-bz14... \
>  172.31.45.44:2377

This node joined a swarm as a worker.
```



Protect your Join Tokens

Only approved nodes should be allowed to join your Swarm!

```
$ docker swarm join \
>  --token SWMTKN-1-31fx-bz14... \
>  172.31.45.44:2377

This node joined a swarm as a worker.
```

To join a Swarm as a **manager**, a node must specify the **manager join token.** Keep it safe!

To join a Swarm as a **worker**, a node must specify the **worker join token.** Keep it safe!

You can rotate join tokens with:

```
$ docker swarm join-token --rotate worker|manager
```

Swarm Mode: Client Certificates

```
$ openssl x509 -in
/var/lib/docker/swarm/certificates/swarm-node.crt -text

Certificate:
...
  Issuer: CN=swarm-ca
   Validity
     Not Before: Mar  9 15:21:00 2017 GMT
     Not After : Jun  7 16:21:00 2017 GMT
   Subject: O=lgz5xj1eqg..., OU=swarm-manager, CN=ofcm6bdy...
   ...
     X509v3 Subject Alternative Name:
       DNS:swarm-manager, DNS:ofcm6bdy..., DNS:swarm-ca
     ...
-----BEGIN CERTIFICATE-----
MIICNDCCAdugAwIBAgIUCoRaj23j4h5
...
```

Swarm ID

Node Role

Node ID

All nodes get a client certificate

O = Swarm ID

OU = Role

CN = Node ID

Client certificates are used for mutual authentication and encryption.

Swarm Mode: Client Certificates

```
Certificate:
...
  Issuer: CN=swarm-ca
   Validity
     Not Before: Mar  9 15:21:00 2017 GMT
     Not After : Jun  7 16:21:00 2017 GMT
   Subject: O=lgz5xj1eqg4pcd0bib75i4fhd, OU=swarm-manager, CN=ofcm6bdy5qcrlievawsw9wqfp
     X509v3 Subject Alternative Name:
       DNS:swarm-manager, DNS:ofcm6bdy..., DNS:swarm-ca
     ...

$ docker node ls
ID                           HOSTNAME            STATUS  AVAILABILITY  MANAGER STATUS
4ckd17z0uk6fzi0tfwyxbra1g    ip-172-31-34-195    Ready   Active
ofcm6bdy5qcrlievawsw9wqfp *  ip-172-31-45-44     Ready   Active        Leader
p73dypqeyeg9p7iab9d0qzns5    ip-172-31-46-1      Ready   Active        Reachable
ubt37ywh3j171f6lpv3n5et4u    ip-172-31-43-107    Ready   Active        Reachable
uf7y3ap5qdyrwmxt9upnctxws    ip-172-31-46-102    Ready   Active
```

Swarm Info

```
$ docker info
...
Swarm: active
 NodeID: ofcm6bdy5qcrlievawsw9wqfp
 Is Manager: true
 ClusterID: lgz5xj1eqg4pcd0bib75i4fhd
 Managers: 3
 Nodes: 5
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
 Node Address: 172.31.45.44
 Manager Addresses:
  172.31.43.107:2377
  172.31.45.44:2377
  172.31.46.1:2377
```

The `docker info` command can be used to display information about the Swarm that a node belongs to.

Some security related items are shown in yellow

Simple Certificate Rotation

Automatic *client certificate* rotation

- defaults to 90 days
- Customizable

Swarm operates a whitelist of valid certificates

Renewal times are randomized to prevent overloading the CA

Name: `manager1`

ID (CN): `ofcm6bdy5qcrlievawsw9wqfp`

Swarm (O): `lgz5xj1eqg4pcd0bib75i4fhd`

Role (OU): `swarm-manager`

Not before: `Mar  9 15:21:00 2017 GMT`

Not after: `Jun  7 16:21:00 2017 GMT`

sha256: `hxi3...`

Certificate Rotation

Only client certificates can be rotated*

Use the `--cert-expiry flag` to change the rotation period

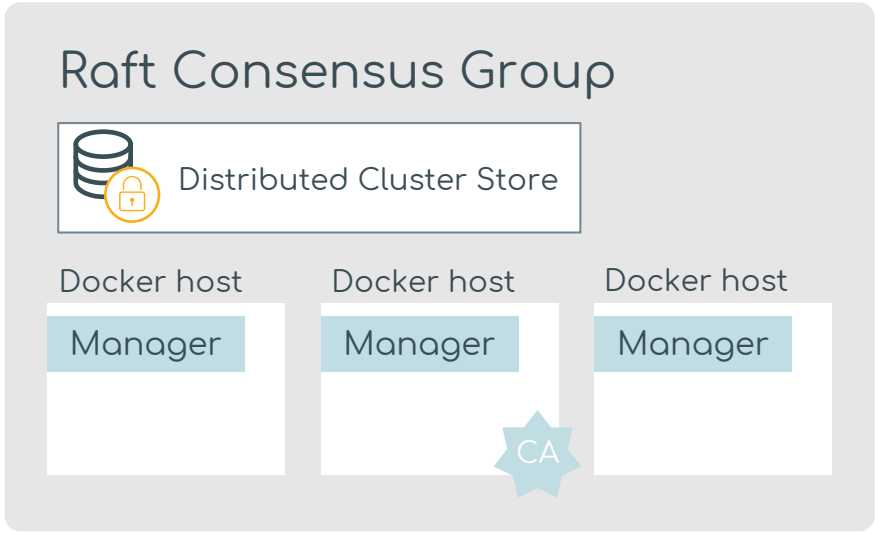
The following command will build a Swarm that rotates client certificates every 30 days

```
docker swarm init --cert-expiry 720h0m0s
```

The following command updates a Swarm to rotate client certificates every 60 days

```
docker swarm update --cert-expiry 1440h
```

Docker Swarm: Secure Cluster Store

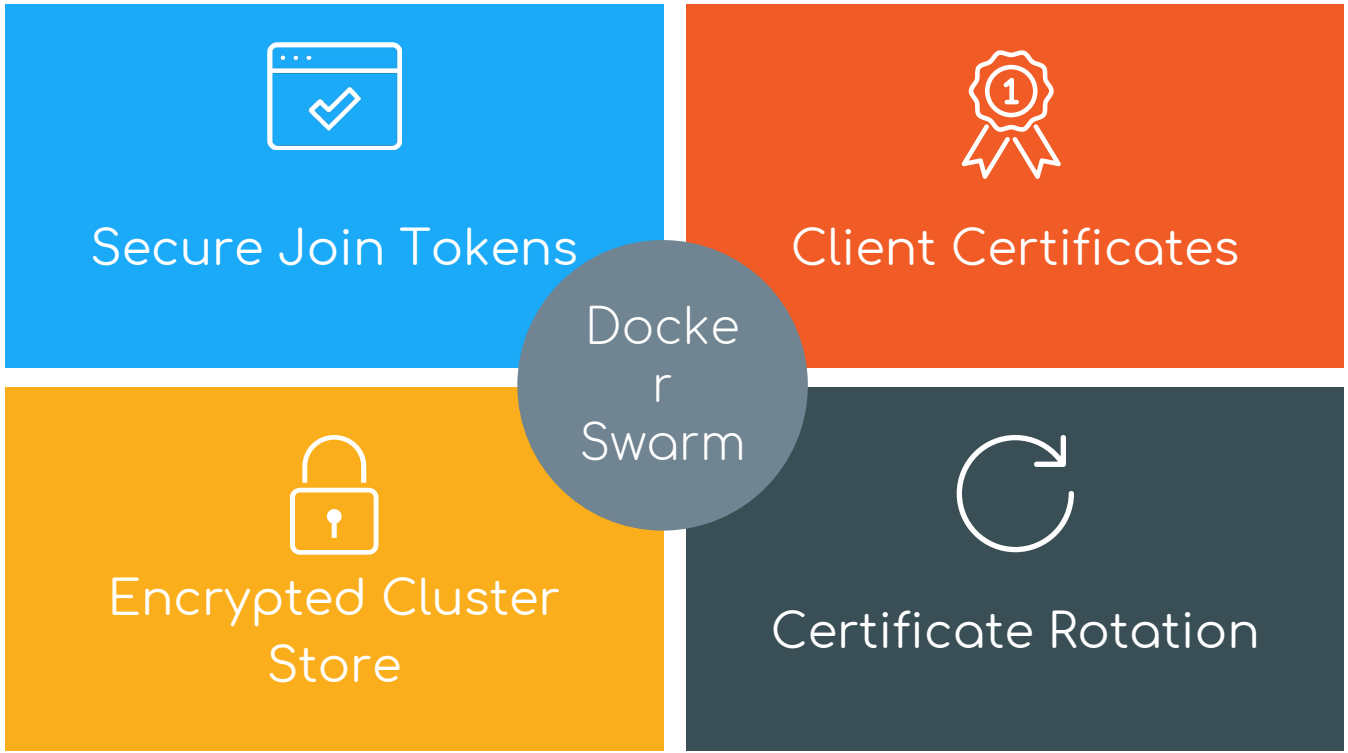


The cluster store is encrypted

- Anything stored in the cluster store is encrypted (secrets etc.)

The cluster store is distributed/replicated across all managers

Docker Swarm Security: Recap



Docker Swarm: Workload Placement

Constraints

Limit the **nodes** that service tasks can run on

Constraints

Constraints use the following:

Built-in **node attributes**	`node.id | node.hostname | node.role | …`
Built-in **Engine labels**	`engine.labels.operatingsystem | …`
User-define **node labels**	`node.labels.zone | node.labels.pcidss …`

Constraints: Only Run Tasks on Worker Nodes

```
$ docker service create \
    --name svc1 \
    --constraint ‘node.role == worker’ \
    redis:latest
```

Constraints: Only Run Tasks on Nodes Running Ubuntu

```
$ docker service create \
 --name svc1 \
 --constraint ‘engine.labels.operatingsystem == ubuntu 16.04’ \
 redis:latest
```

Constraints: User-defined Labels

```
$ docker node update \
    --label-add zone=prod1 \
    node1


$ docker service create \
 --name svc1 \
 --constraint ‘node.labels.zone == prod1’ \
 redis:latest
```

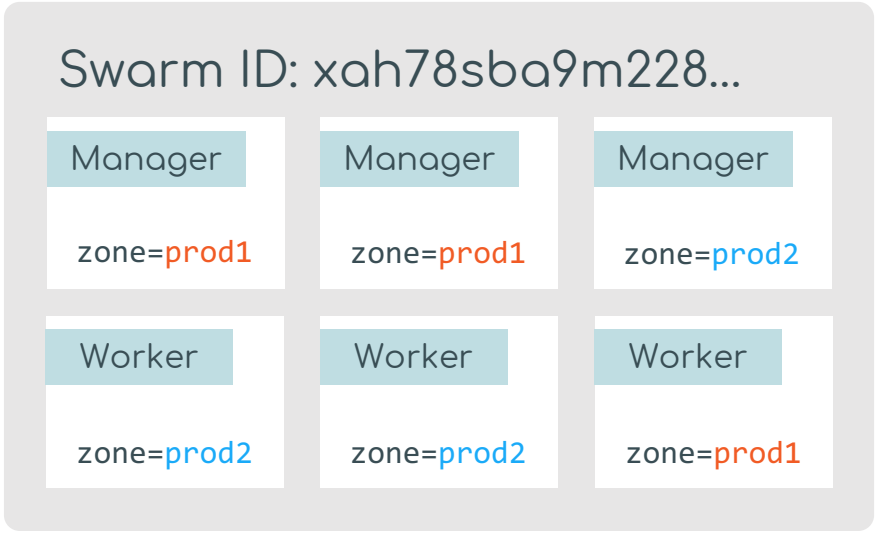
Constraints: User-defined Labels

```
$ docker node update \
    --label-add zone=prod1 \
    node1


$ docker service create \
 --name svc1 \
 --constraint ‘node.labels.zone != prod1’ \
 redis:latest
```

User-defined Labels

```
$ docker node update --label-add
```

- Simple key/value pairs
- Great way to organize nodes
- Only apply within the Swarm



PCI-DSS Example

```
docker service create \
 --name web-fe \
 --constraint 'node.labels.pcidss
== yes' \
 --replicas=3
 corp1/nginx:hardened
```

- Single Swarm with 6 nodes
- 3 nodes with label `pcidss=yes`
- 3 nodes with label `pcidss=no`
- Service deployed with constraint:
  - `node.labels.pcidss == yes`
    - Service tasks can only be scheduled on nodes with label `pcidss=yes`

Q&A

Lab

Building A Secure Swarm

Secure App-centric Networks

with Swarm Mode

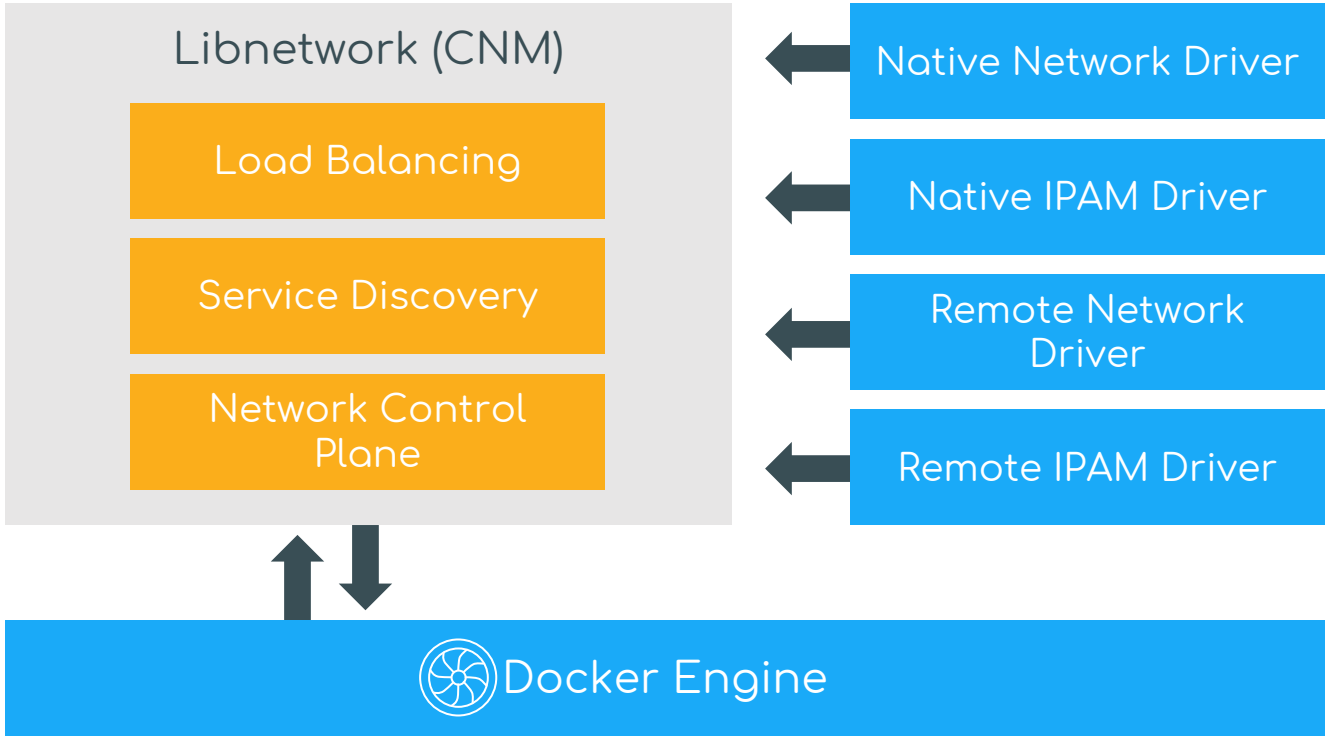
Background: Networking is Important!

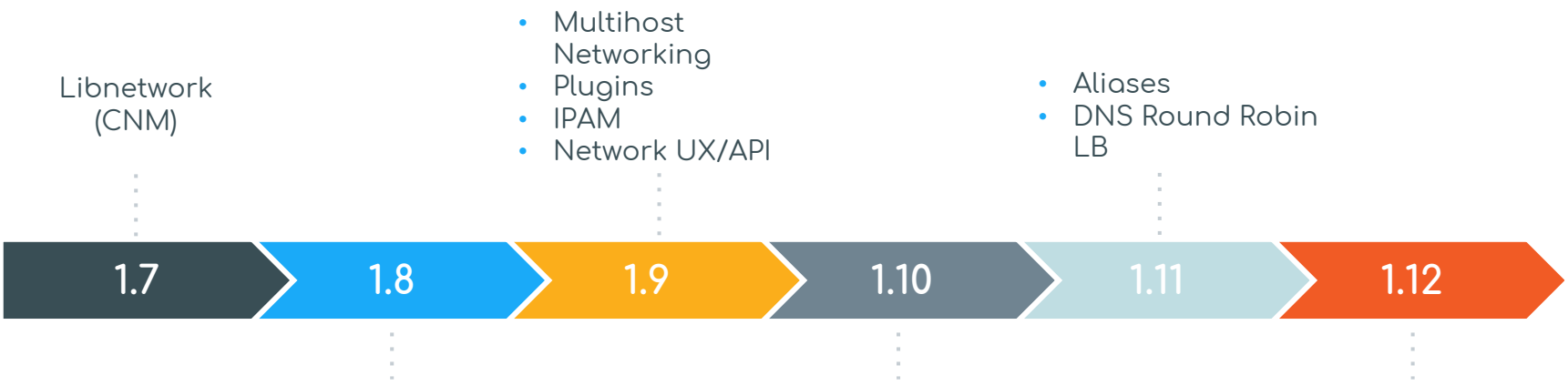
Networking is integral to distributed applications

But networking is hard, vast, and complex!

Goal: Make Docker networking SIMPLE and SECURE!

Docker Networking Architecture



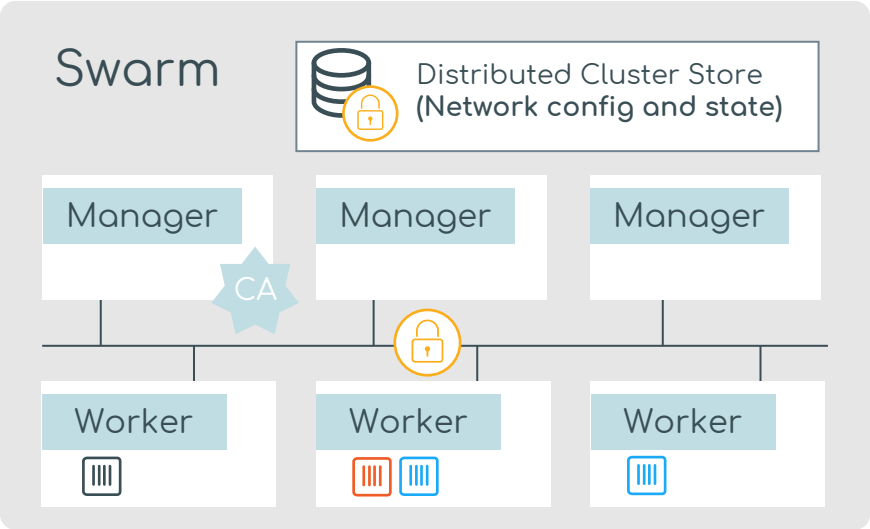


PCI-DSS Example

Every **Swarm** gets a distributed **cluster store**

- Encrypted by default
- Stores network config and state

All node-to-node communication is secured by mutual TLS



Secure Networking: Container to Container

Control Plane

Encrypted by default

- AES (GCM)
- Keys rotated every 12 hours

Data Plane

Can be easily encrypted

- `--opt encrypted`
- AES (GCM)
- Keys rotated every 12 hours

Secure Container Networking: Example

```
$ docker network create -d overlay --opt encrypted my-net
```

Control Plane encrypted

Data Plane encrypted

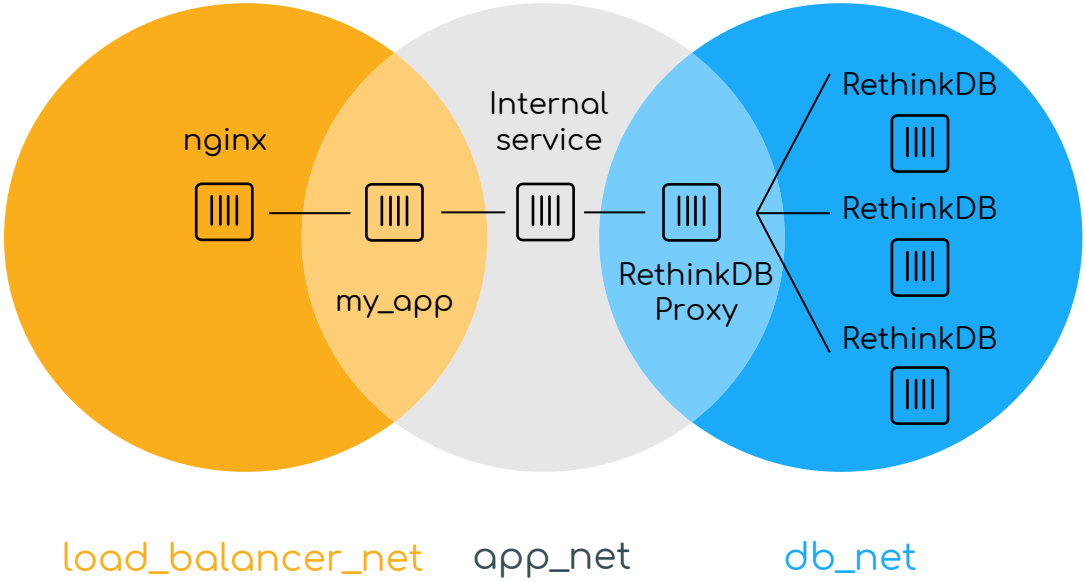
Keys automatically rotated

Config in secure cluster store

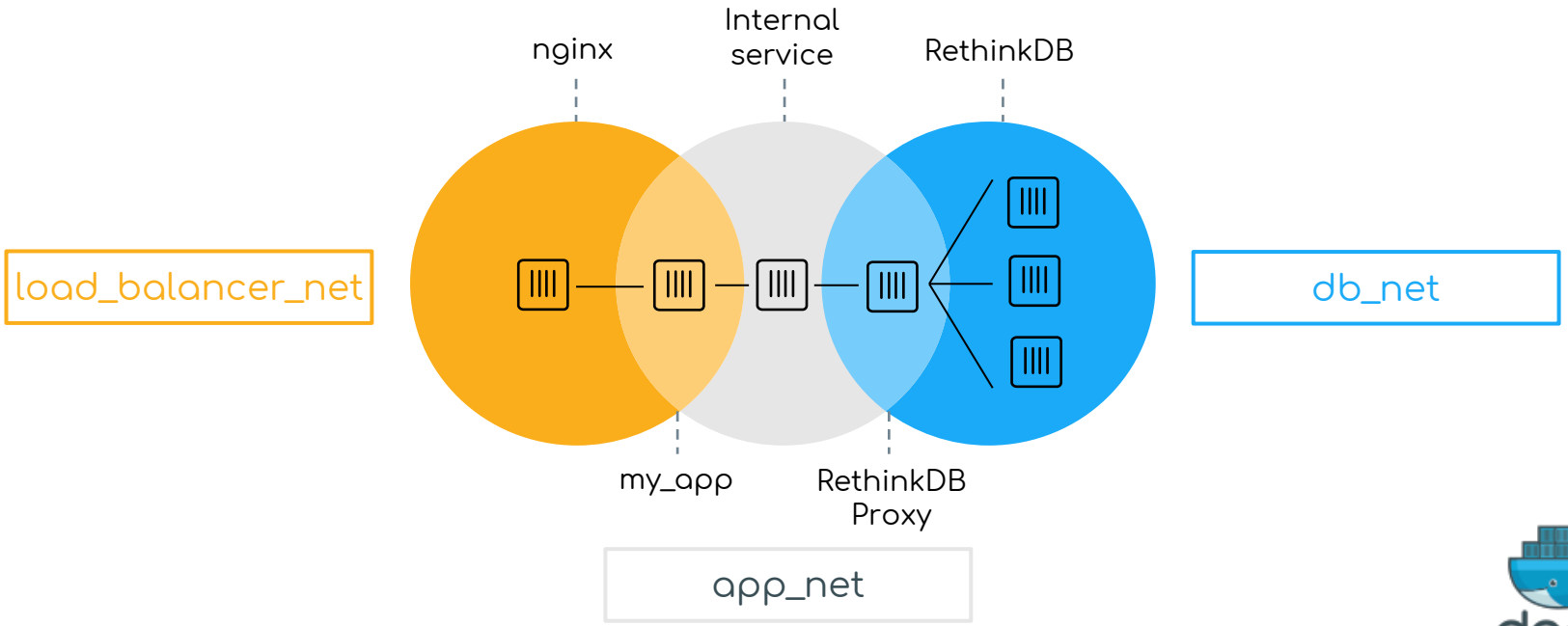
Secure Container Networking: Lazy Creation

- Newly created networks are only created on nodes that **need** them
  - Nodes that do not need them do not get them (more secure)
  - Reduces network chatter (more secure)

Secure Container Networking: Isolation

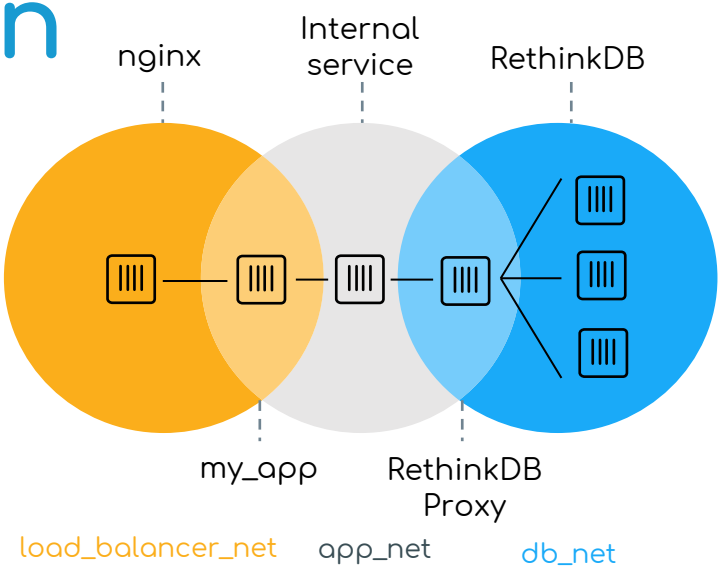


Secure Container Networking: Isolation



Secure Container Networking: Isolation

- Micro segmentation
- By default, containers can only talk to other containers on the same network
- Service Discovery is network-scoped
  - Containers cannot automatically discover services and containers on other networks



Networking Gotcha

Starting a container with the --net=host will allow the container to see **all networking traffic on the Docker host!**

```
$ docker container run --rm -it \
  --net=host \
  alpine sh
```

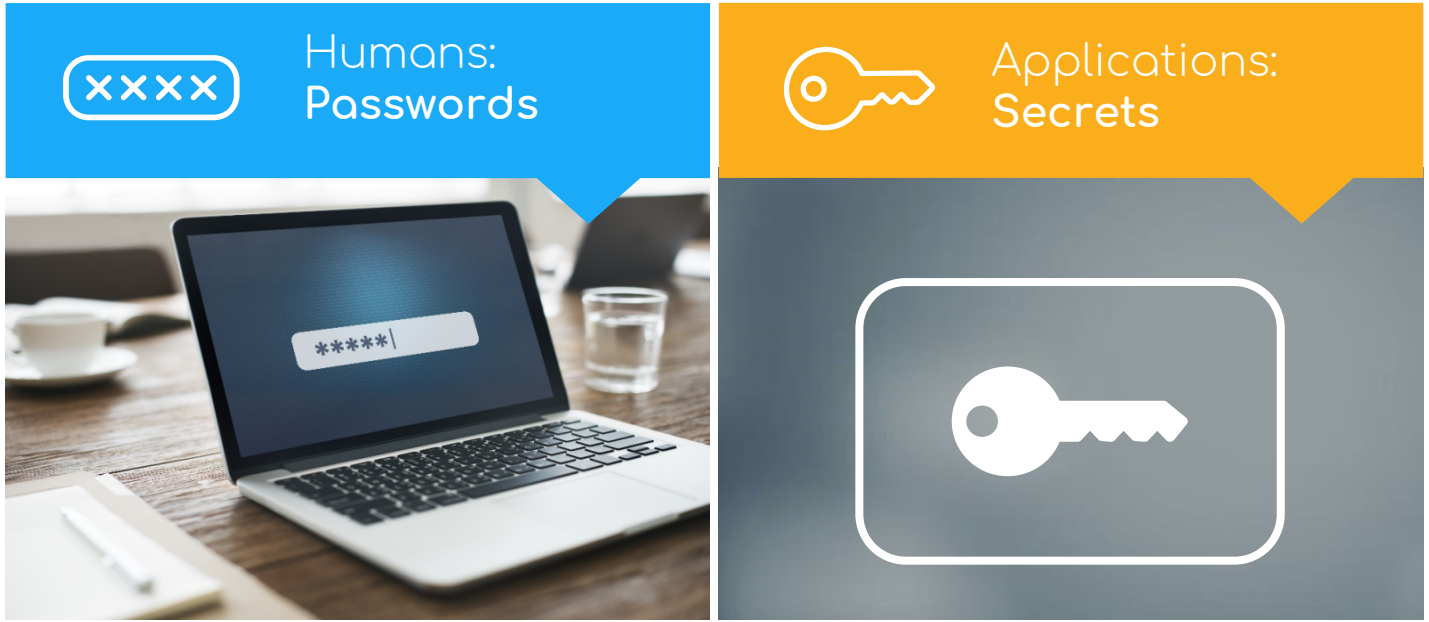
Avoid at all costs!

Q&A

Container Native Secrets Management

Docker 1.13 Introduced Native Docker Secrets Management

What is a Secret



The Three Pillars of Docker Security

Secrets

Usable Security

Trusted delivery

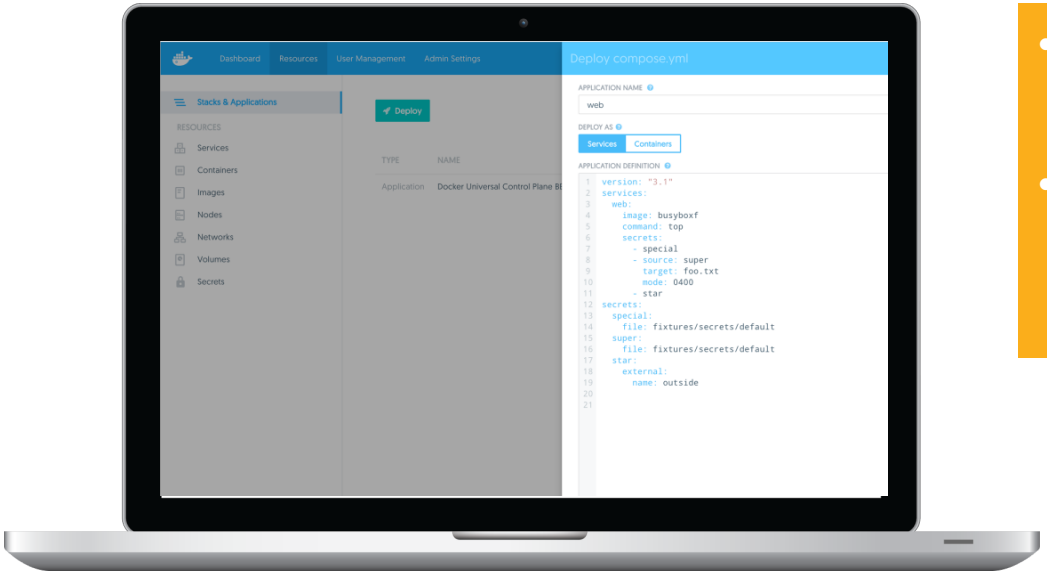
Infrastructure Independent

Secrets Management: Usable Security

Usable Security

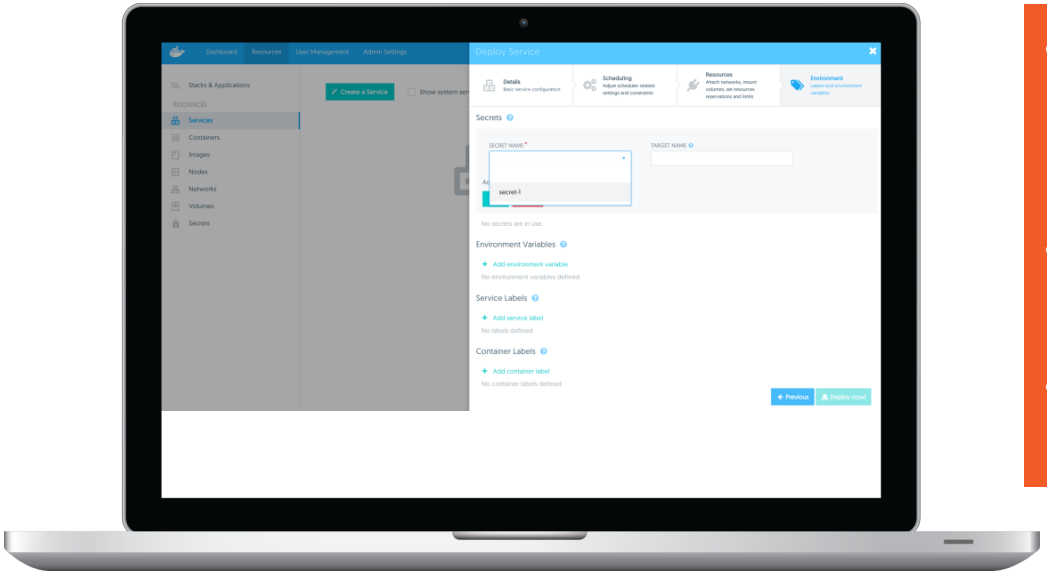
- Standardized interface for developers
- Standardized interface for operations teams
- Fits most existing methods of accessing secrets
- Leverages existing security features of Swarm Mode

Secrets Management: Usable Security (Devs)



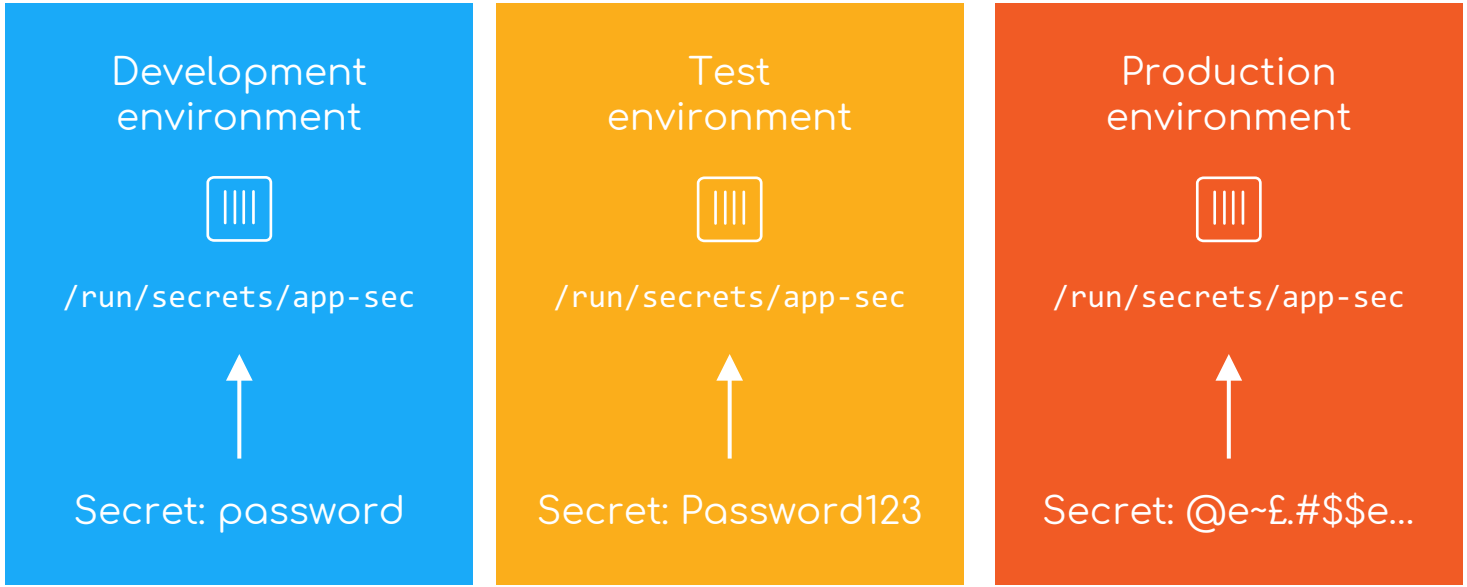
- Compose and services support for secrets
- Define services, secrets, networks and volumes **in a single file**

Secrets Management: Usable Security (Ops)



- Integrated secrets and app management in Docker Datacenter
- Deploy Compose file directly with no code changes
- Add granular access control to secrets and services

Secrets Management: Simplified Workflow (example)



Secrets Management: Trusted Delivery

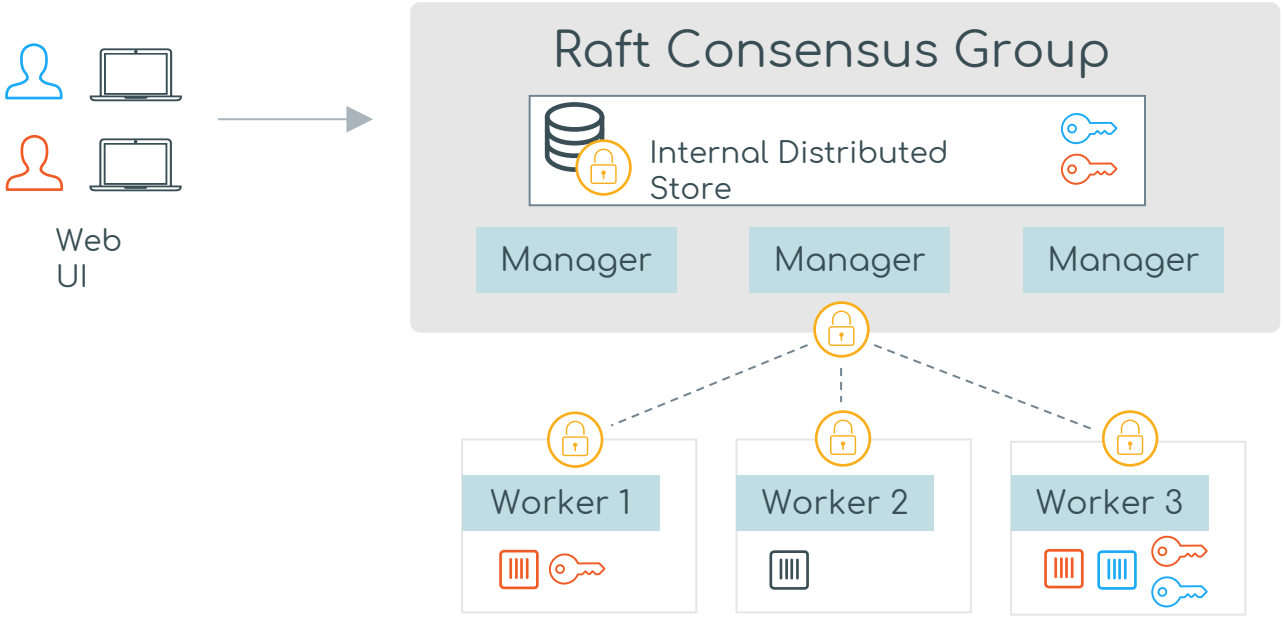
Secrets encrypted at rest in the cluster store

Secrets encrypted in-flight over the network

Secrets only available to authorized apps/services

Secrets never persisted to disk in containers or on nodes

Secrets Management: Trusted Delivery



Docker Secrets Management: Infrastructure Independence

Infrastructure Independent

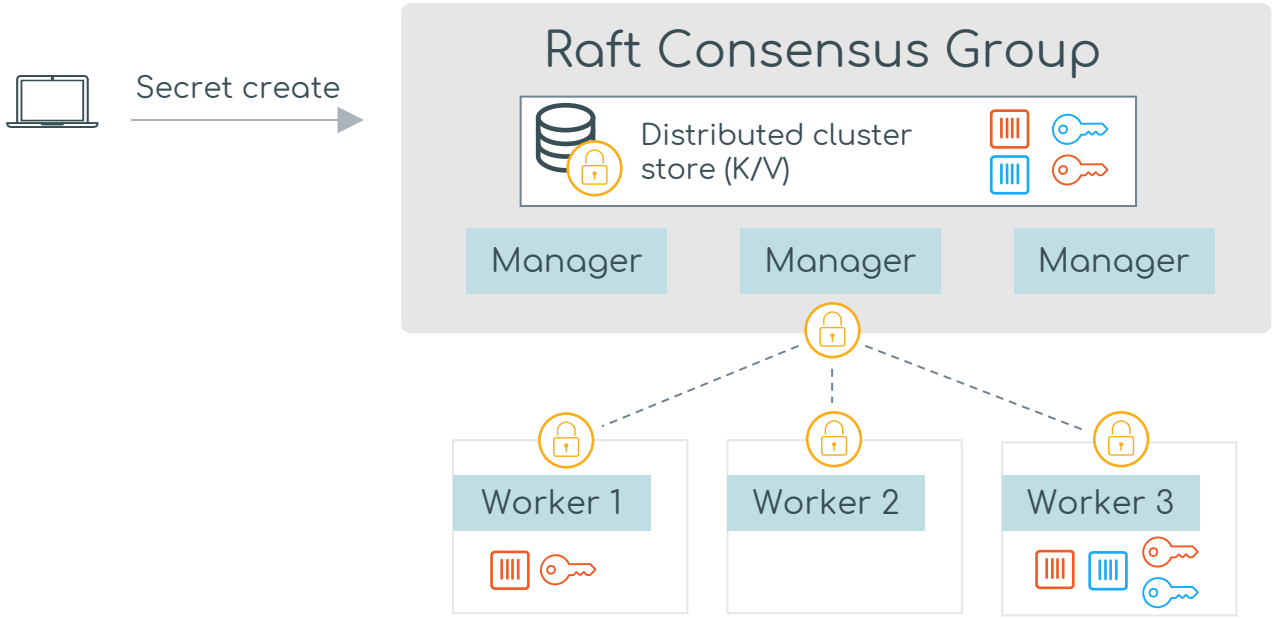
Security is inherent to the Docker platform

Security features and guarantees travel with your app across different infrastructures

Docker Secrets Management: Summary

Usable Security	Secure defaults with tooling that is native to both dev and ops
+ Trusted Delivery	Everything needed for a full functioning app is delivered safely and guaranteed to not be tampered with
+ Infrastructure Independent	All of these things in your system are in the app platform and can move across infrastructure without disrupting the app
= Safer Apps	

Secrets Management: Summary



Secrets Management: Summary

Usable Security

Trusted delivery

Infrastructure Independent

Docker Datacenter adds RBAC

Never store secrets in your app!

Requires Docker 1.13+ in Swarm Mode

Q&A

Lab

Docker Engine & Docker Datacenter Labs Available

Linux Security Technologies

User Management

Managing Daemon and Container Privileges

The Docker Daemon Requires Root

- The Docker daemon (**dockerd**) is in charge of starting and managing containers
- Starting and managing containers means working with **kernel** features such as namespaces.
- Working with kernel features requires root.
- Verify that your Docker daemon is running as root:

```
$ ps -aux | grep dockerd
root  22345  0.3  6.4 541936 65812 ?  Ssl  09:14  0:16
/usr/bin/dockerd -H fd://
```

Control Access to the Docker Daemon

Access to the Docker Daemon (dockerd) is via /var/run/docker.sock

- This is local non-networked Unix socket
- The group owner of the socket is the local **docker** Unix group

```
$ ls -l /var/run/docker.sock
srw-rw---- 1 root docker 0 Mar 30 09:15 /var/run/docker.sock
```

You should grant regular user accounts access to the Docker daemon (via the socket) by adding them to the local **docker** Unix group

```
$ sudo usermod -aG docker npoulton
```

By Default, Containers Run as Root

```
$ docker container run -v /bin:/host/bin -it --rm alpine sh

/ # whoami
root

/ # id
uid=0(root) gid=0(root)

/ # rm /host/bin/*
```

This will delete all files in the `/bin` directory on the **Docker host!** Don't do it!

By Default, Containers Run as Root

By default

root inside a container == root outside a container

Run containers as non-root users

```
$ docker container run --user 1000:1000 \
  -v /bin:/host/bin -it --rm alpine sh

/ $ id
uid=1000 gid=1000

/ $ rm /host/bin/sh
rm: can't remove '/host/bin/sh': Permission denied

/ $ ps
PID   USER     TIME   COMMAND
 1    1000     0:00   sh
```

The container does not have root access to the host

The process/app running in the container is not running as root inside the container

User Namespaces to the Rescue

- User namespaces:
  - Been in the Linux kernel for a while
  - Supported in Docker since 1.10
- How they work:
  - Give a container its own isolated set of UIDs and GIDs
  - These isolated UIDs and GIDs inside the container are mapped to non-privileged UIDs and GIDs on the Docker host.

User Namespaces: Example

```
$ sudo systemctl stop docker
$ sudo dockerd --userns-remap=default &
INFO[0000] User namespaces: ID ranges will be mapped to subuid/subgid…
<Snip>

$ docker run -v /bin:/host/bin -it --rm alpine sh

/ # id
uid=0(root) gid=0(root) ...

/ # rm /host/bin/sh
rm: can't remove '/host/bin/sh': Permission denied
```

Running as root inside container

NOT running as root outside container

User Namespaces: Behind the Scenes

```
$ sudo dockerd --userns-remap=default &
```

The **--userns-remp** flag uses mappings defined in **/etc/subuid** and **/etc/subgid**

```
$ cat /etc/subuid
lxd:100000:65536
root:100000:65536
ubuntu:165536:65536
dockremap:231072:65536
```

```
cat /etc/subgid
lxd:100000:65536
root:100000:65536
ubuntu:165536:65536
dockremap:231072:65536
```

Mapping to the **default** user namespace uses the **dockermap** user and group

Mappings contain three fields:

- User or group name
- Starting subordinate UID/GID
- Number of subordinate UIDs/GIDs available

User Namespaces: Behind the Scenes

When you start Docker with the `--userns-remap` flag the daemon runs within the confined user namespace.

- As part of the implementation a new Docker environment is created under `/var/lib/docker`
- The name of this new subdirectory the mapped UID and the mapped GID

```
$ sudo ls -l /var/lib/dockertotal 40
drwx------ 11 231072 231072 4096 Mar 30 11:17 231072.231072
```

This remapped daemon will operate inside of this 231072.231072 environment

- All of you previously pulled images etc will be inaccessible to this remapped daemon

User Namespaces: Behind the Scenes

You can verify the namespace that the daemon is running in with the `docker info` command

It is not recommended to regularly stop and restart the daemon in new user namespaces

- Mainly because you cannot access images etc. in other namespaces (including the global namespace)

```
$ docker info
Containers: 1
 Running: 1
 Paused: 0
 Stopped: 0
Images: 1
Server Version: 17.03.1-ce
Storage Driver: aufs
<Snip>
Docker Root Dir:

/var/lib/docker/231072.231072
...
```

User Management: Recap

The Docker daemon runs as root

- Grant regular users access via the local **docker** Unix group

By default containers run as root

- root inside a container == root outside a container (default)

User namespaces allow you to run processes as root inside a container but not be root outside of the container

Lab

User Namespaces

Q&A

AppArmor

Mandatory Access Control (MAC)

AppArmor

AppArmor is a Linux kernel security module.

You define **profiles** that control access to specific resources such as files and the network.

You can apply these profiles to applications and containers.

AppArmor

Use the docker **info** command to see if AppArmor is installed and available

```
$ docker info
Containers: 1
 Running: 1
 Paused: 0
 Stopped: 0
Images: 1
Server Version: 17.03.1-ce
<Snip>
Security Options:
 apparmor
 seccomp
  Profile: default
 userns
```

AppArmor: Default Docker Profile

- Docker creates and loads a default AppArmor profile for **containers** called **docker-default**
  - Sensible defaults
  - Based on
    - https://github.com/docker/docker/blob/master/profiles/apparmor/template.go
- A profile for the Docker daemon exists but is not installed and used by default

```
# deny write for all files directly in /proc
deny @{PROC}/* w,
# deny write to files not in /proc/<number>/** or
/proc/sys/**
deny @{PROC}/{[^1-9],[^1-9][^0-9],[^1-9s][^0-
9y][^0-9s],[^1-9][^0-9][^0-9][^0-9]*}/** w,
# deny /proc/sys except /proc/sys/k* (effectively
/proc/sys/kernel)
deny @{PROC}/sys/[^k]** w,
# deny everything except shm* in /proc/sys/kernel/
deny @{PROC}/sys/kernel/{?,??,[^s][^h][^m]**} w,
deny @{PROC}/sysrq-trigger rwklx,
deny @{PROC}/mem rwklx,
deny @{PROC}/kmem rwklx,
deny @{PROC}/kcore rwklx,
deny mount,
deny /sys/[^f]*/** wklx,
deny /sys/f[^s]*/** wklx,
deny /sys/fs/[^c]*/** wklx,
deny /sys/fs/c[^g]*/** wklx,
deny /sys/fs/cg[^r]*/** wklx,
deny /sys/firmware/** rwklx,
deny /sys/kernel/security/** rwklx,
```

AppArmor: Specifying a Profile

- You can override the default container profile (**docker-default**) with the `--security-opt flag`

```
$ docker container run --rm -it /
  --security-opt apparmor=custom-profile hello-world
```

AppArmor: Checking Status

Use the **aa-status** command see the status of AppArmor profiles

This is the **docker-default** policy

These three processes in **enforce mode** are three running containers

```
$ aa-status
apparmor module is loaded.
14 profiles are loaded.
14 profiles are in enforce mode.
   /sbin/dhclient
   /usr/bin/lxc-start
   ...
   docker-default
   <Snip>
0 profiles are in complain mode.
4 processes have profiles defined.
4 processes are in enforce mode.
   /sbin/dhclient (924)
   docker-default (26965)
   docker-default (27528)
   docker-default (27908)
```

Lab

AppArmor

seccomp

Syscall Filtering

seccomp

- seccomp is a **Linux kernel module** that acts like a firewall for **syscalls**
  - In the mainline Linux kernel since 2005
  - Supported in Docker since Docker 1.10
- Using **seccomp-bpf** (Berkley Packet Filters) is an extension that makes seccomp more flexible and granular
  - You can create policies that allow granular control of which **syscalls** are allowed and which are not
- Docker allows you to associate seccomp policies with containers
  - The aim is to control (limit) a containers access to the Docker host's kernel

Checking for seccomp

seccomp needs to be enabled in the Docker host's kernel as well as in the Docker Engine.

To check for seccomp in the kernel

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

To check for seccomp in Docker

```
$ docker info | grep seccomp
seccomp
```

Docker's Default seccomp Policy

- Docker automatically applies the **default seccomp policy** to new containers
- The aim of the default policy is to provide a sensible out-of-the-box policy
- You should consider the default policy as *moderately protective* while providing wide application compatibility
- The default policy disables over 40 syscalls (Linux has over 300 syscalls)
- The default policy is available here:

  https://github.com/docker/docker/blob/master/profiles/seccomp/default.json

Overriding the Default seccomp Policy

You can use the `--security-opt` flag to force containers to run within a custom seccomp policy

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

Docker seccomp profiles operate using a whitelist approach that specifies allowed syscalls. Only syscalls on the whitelist are permitted

Running a Container Without a seccomp Policy

You can run containers without a seccomp policy applied

- This is call running a container **unconfined**

```
$ docker run --rm -it \
  --security-opt seccomp=unconfined \
  hello-world
```

It is not recommended to run containers unconfined!

Lab

Seccomp

Capabilities

Slicing and Dicing Root Privileges

Linux Kernel Capabilities

- The Unix world has traditionally divided process into two categories:
  - Privileged (root)
  - Unprivileged (non-root)
- Privileged processes bypass all kernel permission checks (scary)
- Unprivileged process are subject to all kernel permission checks
- This **all or nothing** approach often led to processes running as root when they really only needed a small subset of the privileges assigned to root processes.
- Modern Linux kernels slice root privileges into smaller chunks called **capabilities.**
  - It is now possible to assign **some** root privileges to a process without assigning them all.

Capabilities: Web Server Example

A container running a web server that only needs to bind to a port below 1024 does not need to run as root! **Should not run as root!**

It might be enough to drop all capabilities for that container except **CAP_NET_BIND_SERVICE**.

If an intruder is able to escalate to root within the web server container they will be limited to binding to low numbered privileged ports. They won't be able to bypass file ownership checks, kill processes, lock memory, create special files, modify routing tables, set promiscuous mode, setuid, load kernel modules, chroot, renice processes, ptrace, change the clock etc...

Net result = reduced attack surface!

Docker and Capabilities

- Docker operates a **whitelist** approach to implementing capabilities.
- If a capability isn't on the whitelist it is **dropped**.
- The list on the right shows the current capabilities whitelist for the **default profile.**
  - https://github.com/docker/docker/blob/master/oci/defaults_linux.go#L62-L77
- For a full list of capabilities:
  - http://man7.org/linux/man-pages/man7/capabilities.7.html

```
s.Process.Capabilities = []string{
        "CAP_CHOWN",
        "CAP_DAC_OVERRIDE",
        "CAP_FSETID",
        "CAP_FOWNER",
        "CAP_MKNOD",
        "CAP_NET_RAW",
        "CAP_SETGID",
        "CAP_SETUID",
        "CAP_SETFCAP",
        "CAP_SETPCAP",
        "CAP_NET_BIND_SERVICE",
        "CAP_SYS_CHROOT",
        "CAP_KILL",
        "CAP_AUDIT_WRITE",
}
```

Docker and Capabilities

You can use the **`--cap-add`** and **`--cap-drop`** flags to add an remove capabilities from a container.

To drop the CAP_NET_BIND_SERVICE capability form a container:

```
$ docker container run --rm -it --cap-drop NET_BIND_SERVICE alpine sh
```

The Linux kernel prefixes capabilities with “CAP_”. E.g. CAP_CHOWN, CAP_NET_BIND_SERVICE etc. Docker does not use the “CAP_” prefix but otherwise matches the kernel names.

Docker and Capabilities

To drop all capabilities except the CAP_NET_BIND_SERVICE capability form a container:

```
$ docker container run --rm -it \
  --cap-drop ALL --cap-add NET_BIND_SERVICE \
  alpine sh
```

To add the CAP_CHOWN capability to a container:

```
$ docker container run --rm -it \
  --cap-add CHOWN \
  alpine sh
```

Docker and Capabilities

Docker cannot currently add capabilities to non-root users

- All of the examples shown in the slides have been adding and removing capabilities from containers running as root

Privilege escalation is difficult without file-related capabilities

- File-related capabilities are stored in a file's extended attributes
- Extended attributes are stripped out when Docker Images are built

Lab

Capabilities

Docker Bench

Audit Your Docker Security

Docker Bench

- Open-source tool for running automated tests
  - Inspired by the CIS Docker 1.13 benchmark
  - Regularly updated
- Checks Docker host
- Runs against containers on same host
- Checks for AppArmor, read-only volumes, etc...
- https://dockerbench.com

```
# ------------------------------------------------------------------------------
# Docker Bench for Security v1.3.0
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker 1.13 Benchmark.
# ------------------------------------------------------------------------------

Initializing Thu Jan 26 08:58:33 UTC 2017


[INFO] 1 - Host Configuration
[WARN] 1.1  - Create a separate partition for containers
[INFO] 1.2  - Harden the container host
[PASS] 1.3  - Keep Docker up to date
[INFO]      * Using 1.13.0 which is current as of 2017-01-18
[INFO]      * Check with your operating system vendor for support and security maintenance for Docker
[INFO] 1.4  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:998:ubuntu
[WARN] 1.5  - Audit docker daemon - /usr/bin/docker
[WARN] 1.6  - Audit Docker files and directories - /var/lib/docker
[WARN] 1.7  - Audit Docker files and directories - /etc/docker
[WARN] 1.8  - Audit Docker files and directories - docker.service
[WARN] 1.9  - Audit Docker files and directories - docker.socket
[WARN] 1.10 - Audit Docker files and directories - /etc/default/docker
[INFO] 1.11 - Audit Docker files and directories - /etc/docker/daemon.json
[INFO]      * File not found
[WARN] 1.12 - Audit Docker files and directories - /usr/bin/docker-containerd
[WARN] 1.13 - Audit Docker files and directories - /usr/bin/docker-runc


[INFO] 2 - Docker Daemon Configuration
[WARN] 2.1  - Restrict network traffic between containers
[WARN] 2.2  - Set the logging level
[PASS] 2.3  - Allow Docker to make changes to iptables
[PASS] 2.4  - Do not use insecure registries
[WARN] 2.5  - Do not use the aufs storage driver
[WARN] 2.6  - Configure TLS authentication for Docker daemon
[WARN]      * Docker daemon currently listening on TCP with TLS, but no verification
[INFO] 2.7  - Set default ulimit as appropriate
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.8  - Enable user namespace support
[PASS] 2.9  - Confirm default cgroup usage
[PASS] 2.10 - Do not change base device size until needed
[WARN] 2.11 - Use authorization plugin
[WARN] 2.12 - Configure centralized and remote logging
[WARN] 2.13 - Disable operations on legacy registry (v1)
[WARN] 2.14 - Enable live restore
[PASS] 2.15 - Do not enable swarm mode, if not needed
[PASS] 2.16 - Control the number of manager nodes in a swarm (Swarm mode not enabled)
[PASS] 2.17 - Bind swarm services to a specific host interface
[WARN] 2.18 - Disable Userland Proxy
[PASS] 2.19 - Encrypt data exchanged between containers on different nodes on the overlay network
[PASS] 2.20 - Apply a daemon-wide custom seccomp profile, if needed
[PASS] 2.21 - Avoid experimental features in production
```

Docker Bench

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

Runs as a container

Runs with a lot of **privileges**

- It needs to run tests against the Docker host

Thank you