Purple Containers

Attack and defence across the entire container stack

David Grice

Slides:
https://git.io/fjoX7

I am Dave

- Technical Security Consultant & previously in governance roles
- Work in financial services
- I like motorbikes

Won't cover: CLI & command line examples nor will the demo gods will not be tested.

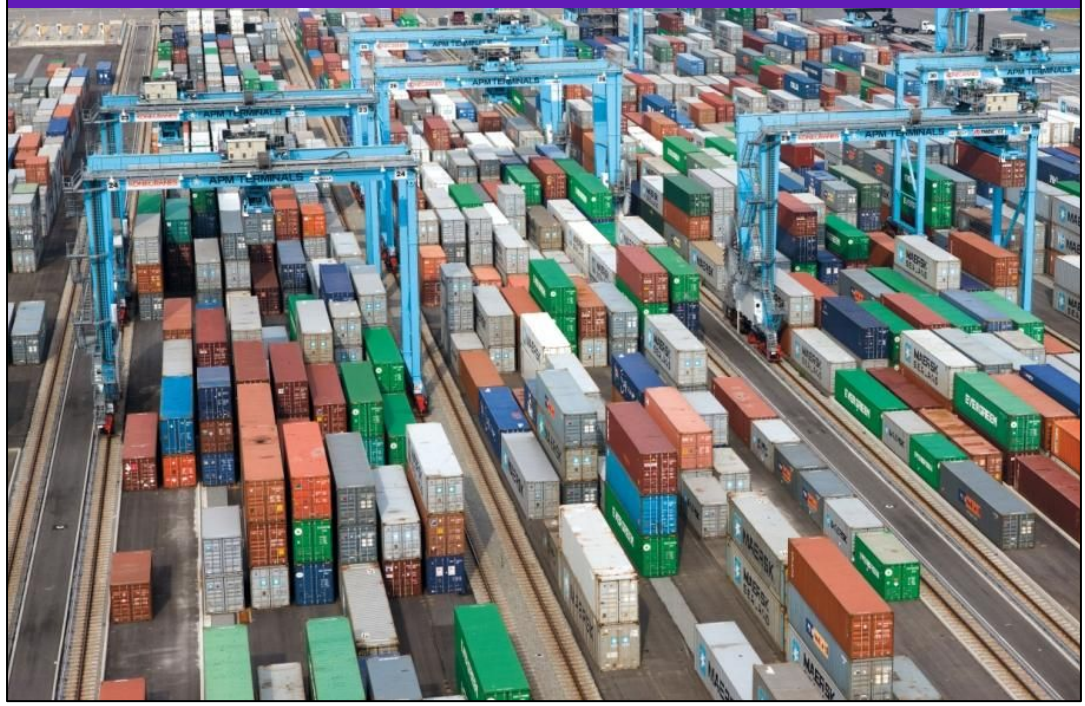
So what are we going to cover in this talk?

1. Explain the reason for this talk and get everyone on the same page in terms of container constructs
2. Then the dense middle part which breaks down attack/threats and the subsequent controls...consider yourselves warned!
3. Finally wrap up with some key summary & take-aways & questions.

All within 45 mins!

The slides will be made available at the end.

Containers have arrived...



Hands up: Running containers? Who is expecting to be running more containers in the next 12 months?

Containers have arrived
Docker was v1.0 in 2014,
forecast to grow at a CAGR of 35% between now & 2021 based on data reported through tech media.

Kubernetes went 1.0 in July 2015 and all the various tech media is call 2019 “Kubernetes for Enterprise”

And there are some pretty compelling reasons why

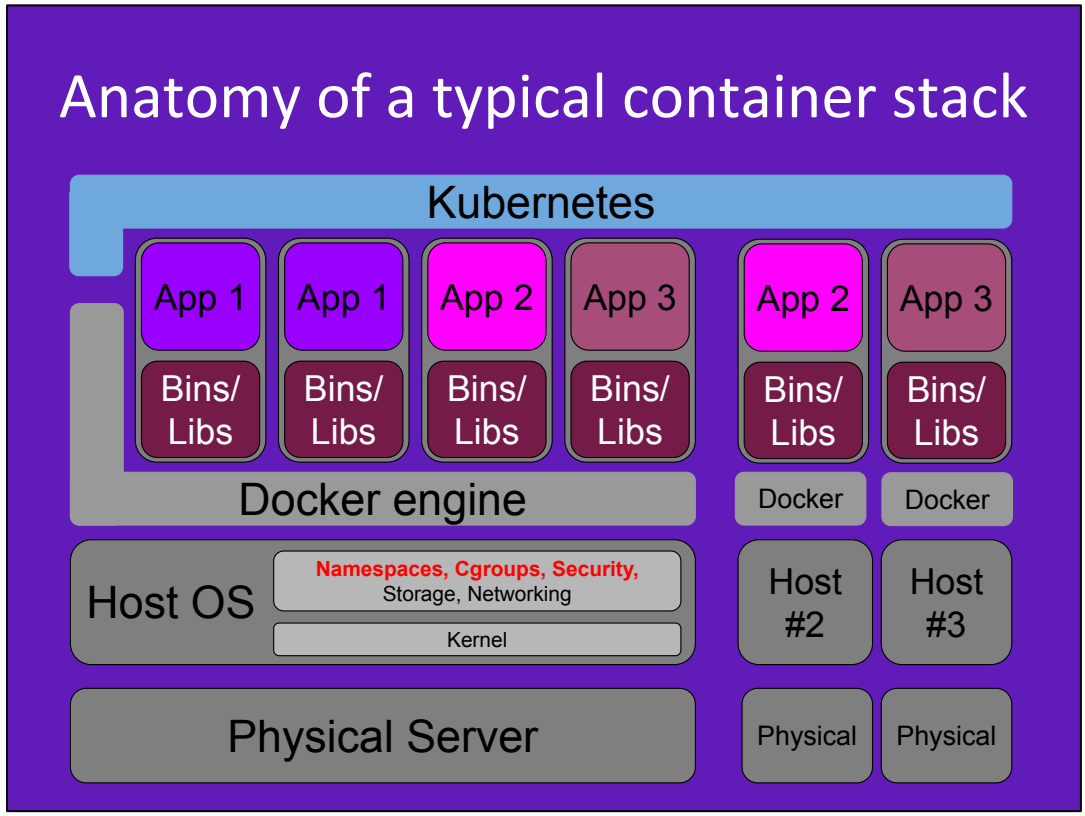
- Platform independence
- Resource efficiency and density
- *(when implemented correctly)* Effective isolation whilst also sharing resources!
- Speed
- Immense and smooth scaling. ...
- Operational simplicity. ...
- Improved developer productivity and development pipeline.

- How this talk came about:
  - Given the benefits, it’s not unreasonable that larger organisations started to get curious
  - App teams were asking for containers….cloud VMs weren’t cutting it.

- Security had the challenge of assessing how we could secure them
- I was working in a governance role, and it became quickly apparent that existing tools and process weren’t going to help us

- When leaning into this challenge, I learnt that containers are different - process, tooling all need to change
- For example:
  - HIDS, Vuln Scan, AV vs. AppArmour, Clair, Notary
  - Securing the development pipeline & eco system becomes critical
  - Infrastructure is now defined in code
  - Change in control ownership
  - 2 days is lifetime in a containerised world

- So this talk is my journey in assessing how a highly regulated financial institution could do this, pulling apart the layers of the container stack and looking at controls to address them.



Starting bottom up..

1. Physical Server: Nothing new here
2. Host OS: still not much new here, although in container land it make sense to run a container specific host OS and we will talk about that a bit later on

3. Linux namespaces: There are the secret sauce that provide the isolation (hence "container") in which we place one or more processes
   Linux cgroups: ("Control groups") provide resource limiting and accounting (CPU, memory, I/0, bandwidth, etc.)
   Security here is typically some form of Mandatory Access Control, like SELinux or AppArmour.
   Then you've got all the other bits you need; storage, networking etc.

4. Docker is by far the most popular containerisation platform. Based on previously available open source technologies Docker created a standard way to deploy Linux applications into containers which then can run in different environments as intended.

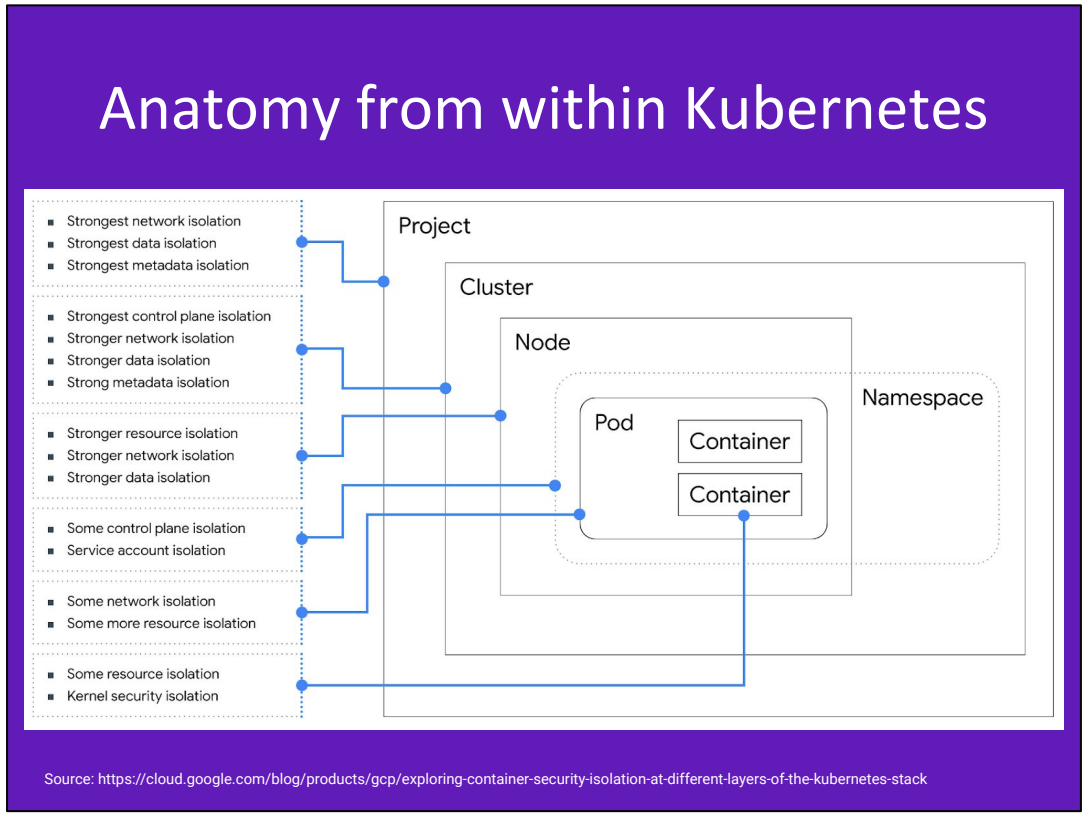
Docker Engine is a client-server application with:

- A server which is a daemon (the dockerd command)
- A command line interface (CLI) client (the docker command)
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.

5. Your application then runs on the of docker engine with its own binaries & libraries on top (that aren't shared with the OS/kernel)

6. Orchestration has a number of different technologies (Docker Swarm, Marathon, Mesos), however Kubernetes appears to be winning (or have won) the orchestration race.
Container orchestration engines all allow users to control when containers:

- start and stop,
- group them into clusters,
- coordinate all of the processes that compose an application
- automate updates
- health monitoring, and failover procedures.



Each boundary is an opportunity to harden or weaken security.
To quote google:

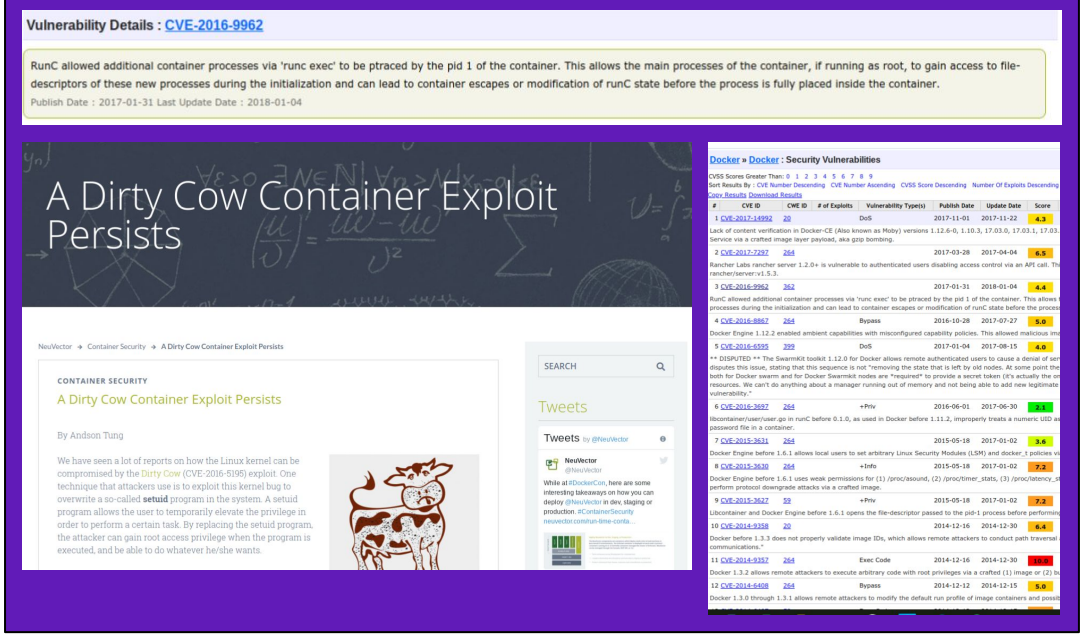
> Containers do not provide an impenetrable security boundary, nor do they aim to.
> The provide some restrictions on access to shared resources but they don't necessarily prevent a malicious attacker from circumventing these restrictions.

source:
https://cloud.google.com/blog/products/gcp/exploring-container-security-an-overview

It’s not just the tech that is different, so are the processes to build & run it.

For a traditional security team, this can present a massive challenge.
True essence of cattle; no pets
Configured in YAML files; infra as code
Effermeral infrastructure - How does that work with your CMDB?
With luck, the death of patching!

The risks are real (FUD page)



The risks are real & we are just scratching the surface as this stuff is all so new

Dec 2018: CVE-2018-1002105 - Kubernetes API server auth bypass

Feb 2019: CVE-2019-5736 - RunC container breakout to host as root

Weight Watchers

AVIVA

Until December we were ‘mostly’ concerned about default and un-hardened instances that led to bitcoin mining.

So, “we want containers” becomes a security review of:

1. The build environment (CI/CD Pipeline)
2. Underlying Operating System Security
    2a. Cloud Services Hardening and Security
3. Container Image (inc. Docker Daemon)
4. Runtime security
5. Orchestration layer (e.g. Kubernetes)
6. Network
7. Logging & Auditing

When considering the ‘we want to adopt containers’ challenge, as a highly regulated entity it is prudent you address security across the stack, rather than just one slice.

Some of these aren’t new…but none the less need to be considered in light of the new technology stack.

Also, security teams may or may not operate all the controls in a DevOps world, and may depend on App teams, but **all** need to be understood.

I think there is an opportunity for security to play a role in some of these areas and I’ll discuss them as we go along.

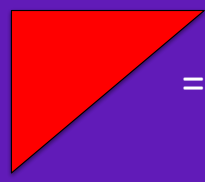
So lets just pause at this point and consider what we know:

- we have unfamiliar technology for a lot of security and traditional non-devops technology teams
- a fastly maturing technical landscape, but critically, one that is not yet mature
- evidence that the risks are real
- Statements from google that containers are not strong enough isolation on their own to protect you from a malicious actor
- and an insatiable adoption rate of this technology

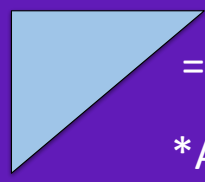
So we probably should be concerned.

The good news is there are lots of solutions as well - provided you implement them. Containers have come a long way , including the defaults to ensure they provide sane, secure config

And so, that huge and complex problem gets easier...the whole point of containers is to segment your application into bite sized manageable chunks
Automation plays an important role in enforcing securing.



= Attack / Threats



= Defence / Controls*

*Adopting all controls would be overkill…pick & choose as appropriate.

1. The build environment

- Malicious or ‘sub-optimal’ source code
- Malicious or mistaken alterations to automated build controllers/policies
- Configuration scripts with errors, or that expose credentials
- Supply chain provenance
- The addition of insecure libraries or down-rev/insecure versions of existing code

Bullet points, yeah!

Protect your build environment like your app

2FA - Enable 2FA on source code repositories, and in particular on all administration activities

Appropriate governance over code mergers

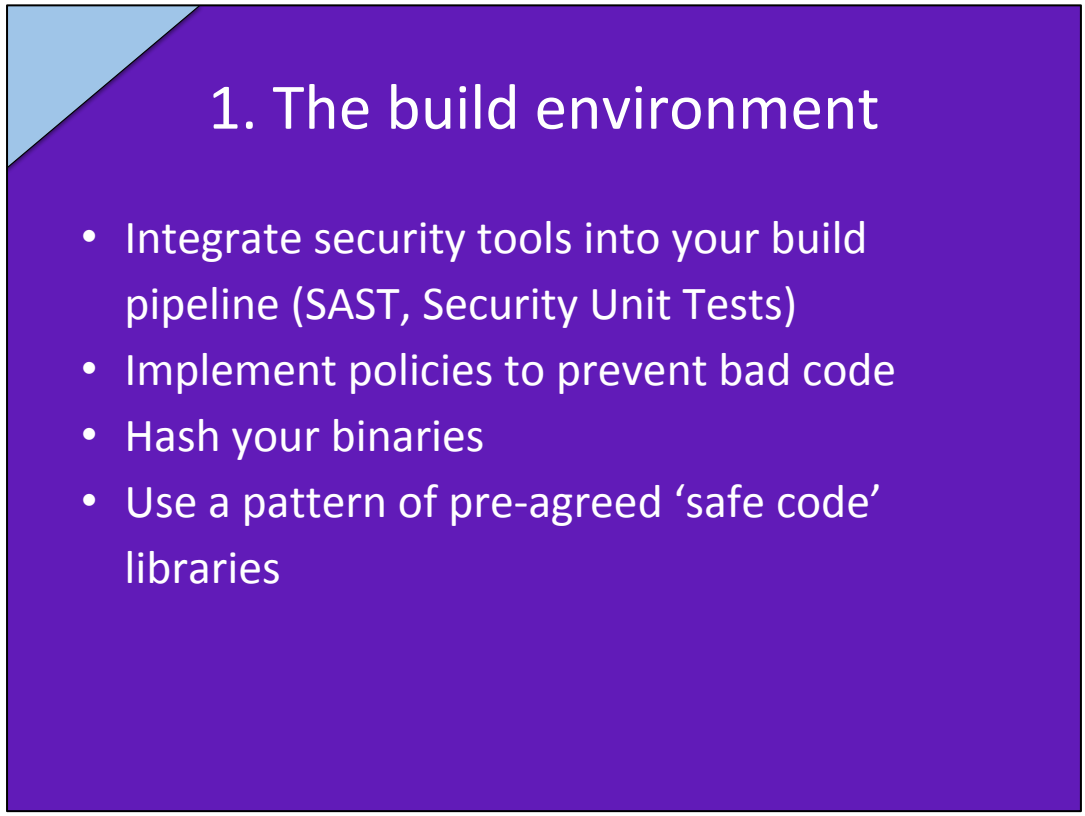
Particularly important when using 3rd party code

Principle of least privilege; restrict who has access to only those necessary

Install security testing plugins in your IDE, and scan code before it's even checked in

Commercial or open source options

Refer you to OWASPs excellent resources here: Source Code Analysis Tools https://www.owasp.org/index.php/Source_Code_Analysis_Tools



Integrate third-party solutions to scan binaries and perform white box testing for known vulns

Refer you to OWASPs excellent resources here: Source Code Analysis Tools https://www.owasp.org/index.php/Source_Code_Analysis_Tools

Implement policies to prevent malicious & vulnerable code being published

Need a feedback loop from your vuln scanner/SAST/DAST tooling that will will allow Jenkins to fail the build if the security of the build is ‘unhealthy’

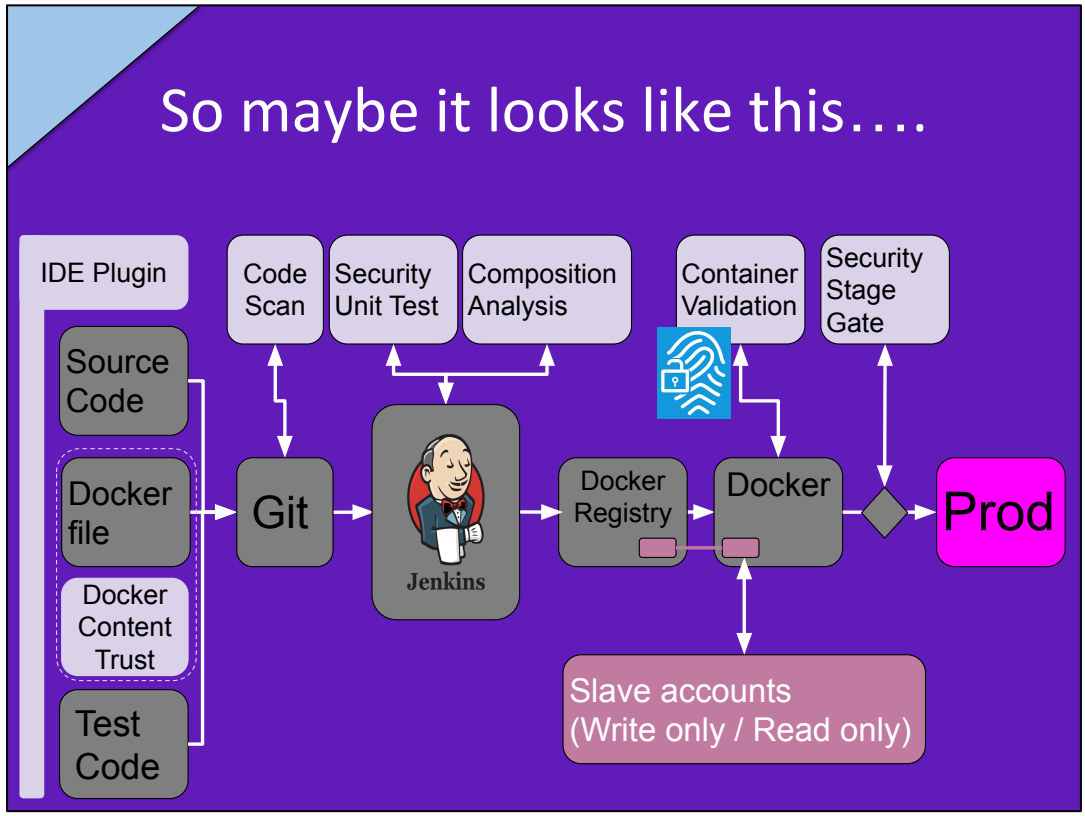
Hash your binaries (and check the hashes!)

The binaries that get deployed into production should be exactly the same as those that went through the acceptance test process—and indeed in many pipeline implementations,

this is checked by storing hashes of the binaries at the time they are created and verifying that the binary is identical at every subsequent stage in the process.

Use a pattern of pre-agreed ‘safe code’ libraries

Regularly review these, particularly when say AWS makes changes to IAM policies for example



IDE plugins before code is committed is a great idea.

Then, scanning your code in your Git repo for some kind of static code analysis is the next step.

SAST:

- Whitebox assessment of your code
- Checks for santised inputs
- OWASP top 10

Security Unit Tests:

- Unit tests are a great way to run focused test cases against specific modules of code
- typically created as your development teams find security and other bugs — without needing to build the entire product every time.
- They cover things such as **XSS and SQLi** testing of known attacks against test systems.

Composition Analysis:

- Check libraries and supporting code against the **CVE (Common Vulnerabilities and Exposures)** database to determine whether you are using vulnerable code.
- Maybe your SAST tool will do this, but that is not always the case

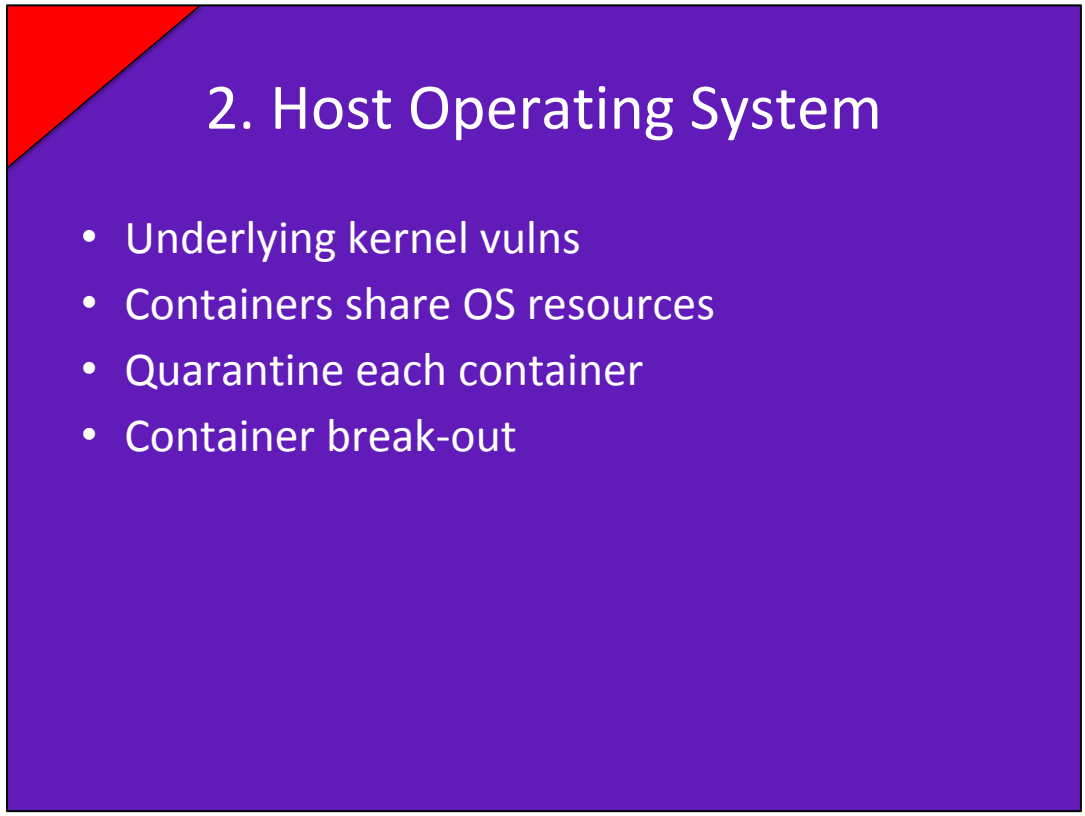
Container Validation:

- Is this the container I was expecting?
- You want to ensure your entire process was followed, and that nowhere along

- the way did a well-intentioned developer subvert your process with untested code.
- You can accomplish this by creating a cryptographic digest of all image contents, and then track it though your container lifecycle to ensure that no unapproved images run in your environment.
- Some container management platforms offer tools to digitally fingerprint code at each phase of the development process, alongside tools to validate the signature chain. But these capabilities are seldom used, and platforms such as Docker may only optionally produce signatures.
- While all code should be checked prior to being placed into a registry or container library, signing images and code modules happens during building.
- You will need to create specific keys for each phase of the build, sign code snippets on test completion but before code is sent on to the next step in the process, and (most important) keep these keys secured so attackers cannot create their own trusted code signatures.

Security Stage Gate:

- A number of third-party products perform automated binary and white box testing, rejecting builds when critical issues are discovered.
- It is recommend you implement some form of code scanning to verify the code you build into containers is secure.
- Many newer tools offer full RESTful API integration within the software delivery pipeline.
- These tests usually take a bit longer to run but still fit within a CI/CD deployment framework.



Underlying kernel vulns

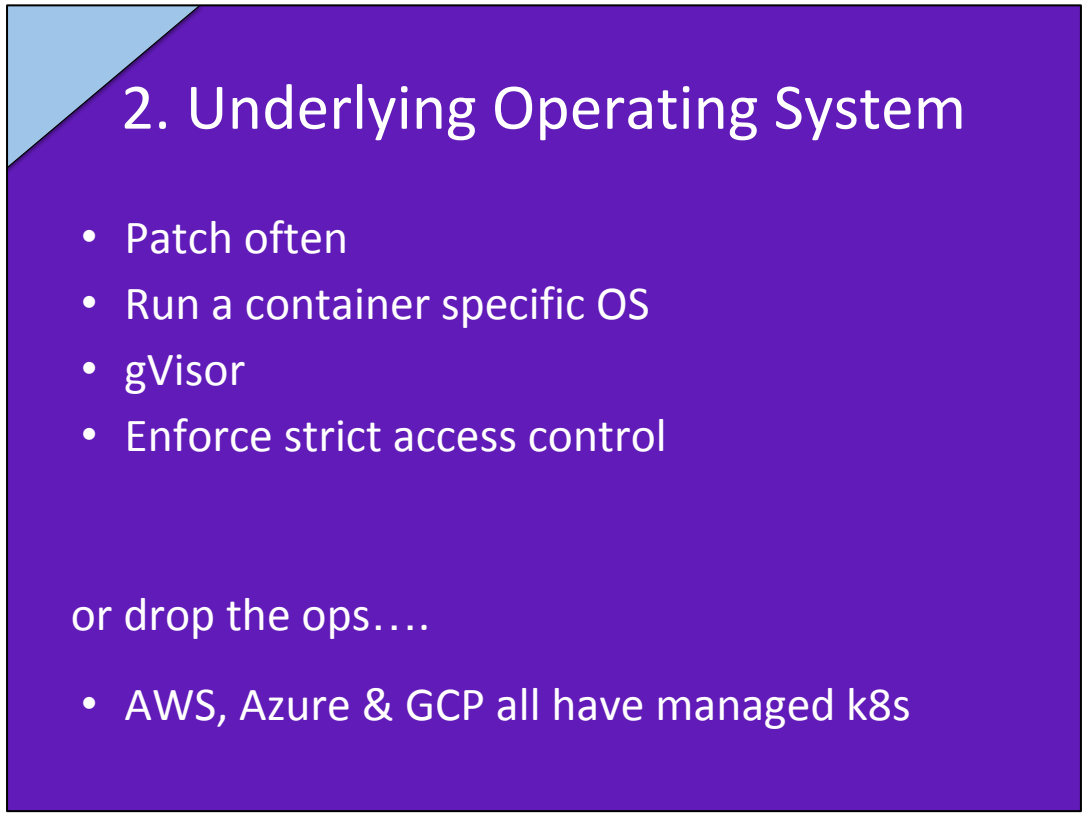
Containers share OS resources

although containers provide strong software-level isolation of resources, the use of a shared kernel invariably results in a larger inter-object attack surface than seen with hypervisors, even for container-specific OSs.

In other words, the level of isolation provided by container runtimes is not as high as that provided by hypervisors.

Container break-out

this is akin to Guest to Host breakout in a VM environment



Patch often

Container specific OSes have ‘auto-update’ features.

Run a container specific OS

First, it must be said that container specific OS are great.

Container-specific OSs have a much smaller attack surface than that of general-purpose OSs. For example, they do not contain libraries and package managers that enable a general-purpose OS to directly run database and web server apps. However,

Some of the big things they do are:
read-only /usr
stateful read/write on the / (root) file system

So they make sure you don’t put anything in /usr/bin and thus it forces you to run containers

Recommended: NIST 800-190
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf

gVisor

It’s a sandbox the brokers user-space to kernel system calls
It’s open source, but under very active devlopment
Right now, there are some trade-offs (e.g. no Istio), but there is a roadmap to address that fairly soon.

Enforce strict access control

Or drop the ops…..

So by leveraging a managed Kubernetes service you hand the ops of the underlying OS to the cloud provider

It goes without saying to chose wisely here, especially with smaller players that can't offer you a SOC 2 Type II certification of the service.

They will all leverage a container specific OS and have 'sane' defaults already applied, but do your due diligence & confirm

You still maintain some responsibility though as cloud is a shared responsibility, and particularly around IAM policies and enforcing the principle of least privilege

3. Container Image

- Images need hardening
- Defaults are not exhaustive, particularly in older versions
- Provenance over supply chain: is this the container I was expecting?

CIS Benchmark
Docker Bench Security

- Need to be understood as a collective.
- This is a hand in glove situation
- The Docker Bench for Security is a script that checks for dozens of common best-practices around deploying Docker containers in production.
- The tests are all automated, and are inspired by the CIS Docker Community Edition Benchmark v1.1.0
- Note that this container is being run with a lot of privilege -- sharing the host's filesystem, pid and network namespaces, due to portions of the benchmark applying to the running host. Don't forget to adjust the shared volumes according to your operating system, for example it might not use systemd.

Container Signing and Chain of Custody

As discussed in the previous slides
DCT is good for ‘official’ images, but what about your own?
Add a hash for the total of the image once you add your binaries and libraries on top and check that hash again before pushing to prod

Remove all unneeded modules, packages and files

Pretty self-explanatory...just trying to minimise the attack surface

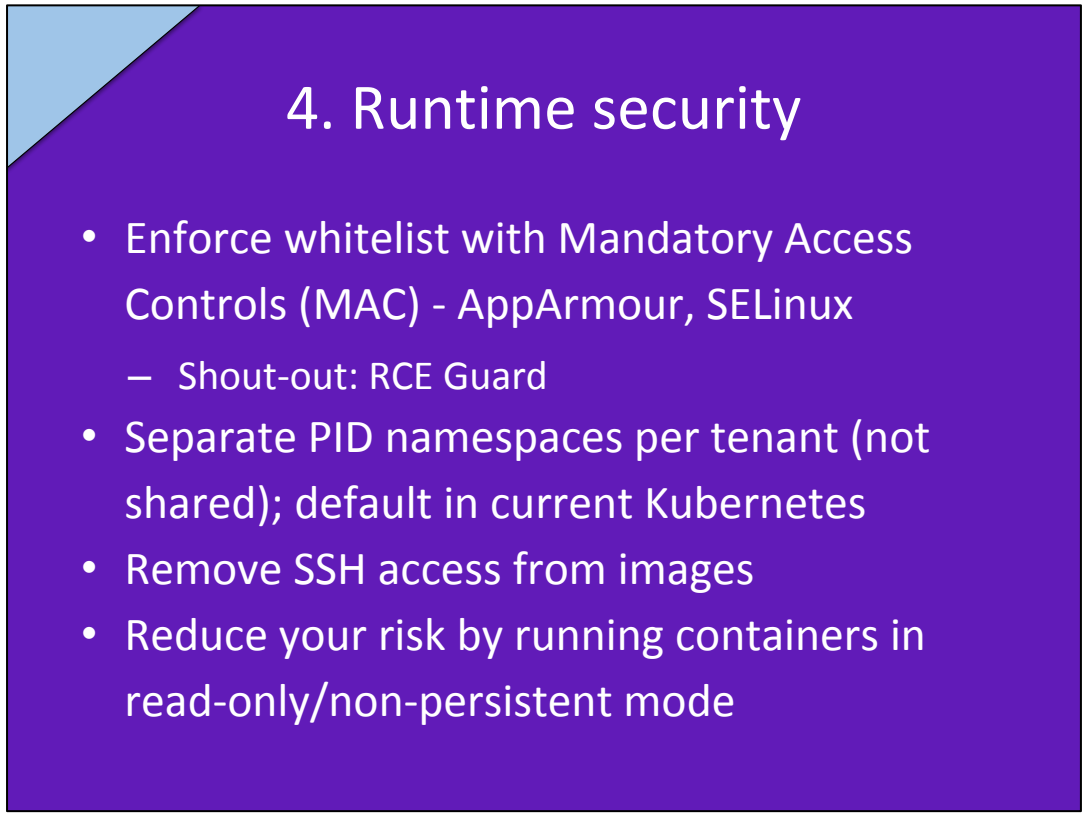
Fail vulnerable builds

Trick is to balance developer productivity & security
Have a strategy for whitelisting

4. Runtime security

- Port scanning
- Hijacked processes
- Data exfiltration
- Backdoor
- Network lateral movement
- Brute force attacks
- Container breakout

For an exhaustive list, check out the Mitre Att&ck Framework
https://attack.mitre.org/wiki/Main_Page



Enforce whitelist with Manditory Access Controls (MAC) - AppArmour, SELinux (Project Atomic)

Prevent writes to the /proc /proc/sys /sys directories

Prevent mounts

It' recommended you generate your security filters/profile whitelist at build (not at runtime)

Separate namespace per tenant (not shared)

Remove SSH Port 22 Access

If you have to, limit what can access via Port 22 via a bastion host. Then, also mark the container as tainted.

The gold standard though is to live in a world with 'root'

Reduce your risk by running application containers in read-only/non-persistent mode

Recommended reading - Jess Frazelle's talk: Security in a Containerized World - DevOps Days

4. Runtime security

- Enable Docker's seccomp profile
- V1.8+ defaults block a lot of nasty stuff: add_key, keyctl, request_key, clone, unshare
- Implement NO_NEW_PRIVS (Default in k8s)
- Enforce time-to-live thresholds
- Continue to vuln scan against running instances
- Super Paranoid? Hardware root of trust

Enable Docker's seccomp profile

Secure computing mode (seccomp) is a Linux kernel feature. You can use it to restrict the actions available within the container

Think of this as AppArmour but instead of operating at the kernel, it operates within the container.

There is a 'gotcha'…..This feature is available only if Docker has been built with seccomp **AND** the kernel is configured with CONFIG_SECCOMP enabled.

V1.8+ defaults block: add_key, keyctl, request_key, clone, unshare

These features have often been associated with a range of CVEs and attacks on containers.

Implement NO_NEW_PRIVS

It's not on by default in Docker.

If a binary has been given a SUID it retains the ability to 'fork', 'clone', execve' which executes the program pointed to by filename; setting NO_NEW_PRIVS prevents these privileges being added to the SUID.

Kubernetes turns this on by default.

Enforce time-to-live thresholds

Vulns come out every day, so consider how long do you want a container running before it's torn down and rebuilt?

Hardware root of trust*

Consider using hardware-based countermeasures to provide a basis for trusted computing

Hardware root of trust is not a concept unique to containers
Container specific OSes make this easier

NIST has some really good details on this

The currently available way to provide trusted computing is to:
1. Measure firmware, software, and configuration data before it is executed using a Root of Trust for Measurement (RTM).
2. Store those measurements in a hardware root of trust, like a trusted platform module (TPM).
3. Validate that the current measurements match the expected measurements.

If so, it can be attested that the platform can be trusted to behave as expected.

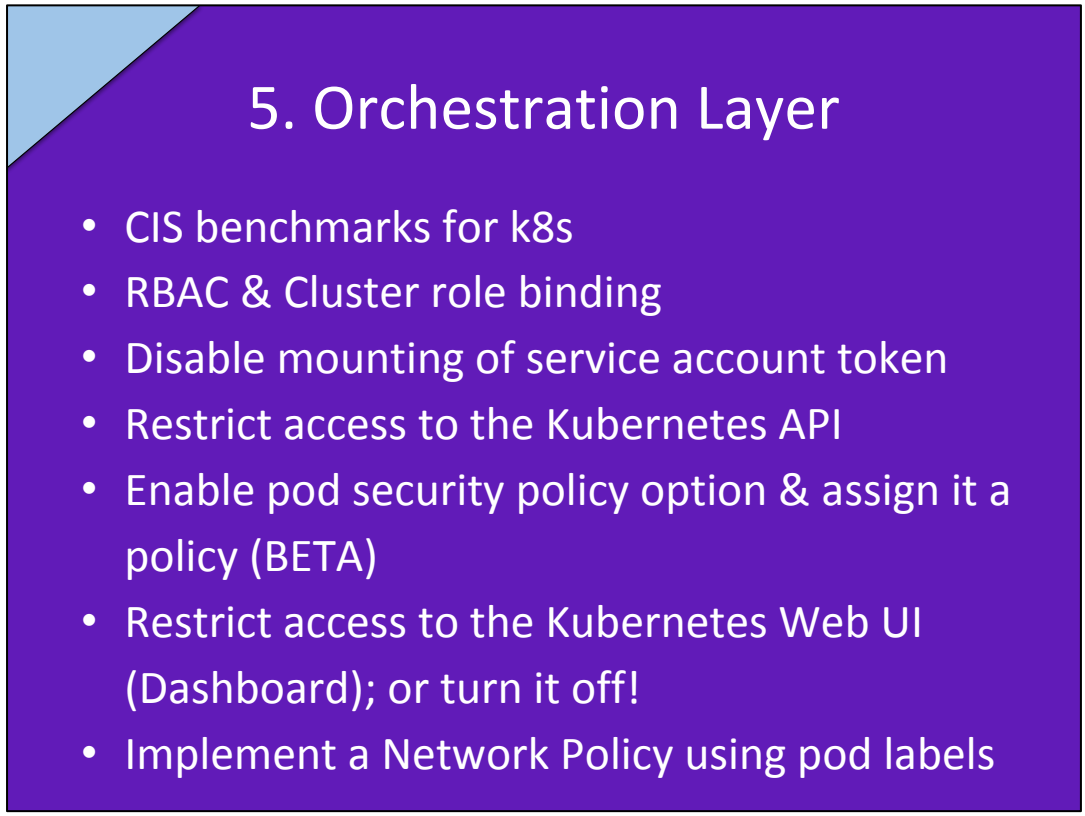
5. Orchestration Layer

- Steal secrets
- Service account access to API layer
- Access cloud providers metadata API (e.g. AWS 169.254.169.254)
- Enumeration of services
- k8s API server attacks via token replay
- Container breakout and hijack kubelet
- Container MITM

Access cloud providers metadata API (e.g. AWS 169.254.169.254)

This is a real attack vector, not just for containers but any cloud service you run

You've probably heard about CSRF, well this metadata API attack is akin to a server side request forgery



CIS benchmarks for k8s

Note the distinction between level 1 & 2, level 2 being essentially another layer of the onion

Kube-bench is the docker bench equivalent for applying the CIS benchmarks for k8s

RBAC & Cluster role binding

this is covered in the CIS benchmark but is fairly critical

RBAC is configured using standard Kubernetes resources. Users can be bound to a set of roles (ClusterRoles and Roles) through bindings (ClusterRoleBindings and RoleBindings).

From more recent versions, Users start with no permissions and must explicitly be granted access by an administrator.

Critically, think about which roles have privilege to get secrets (& that would allow an attacker to escalate privs)

All Kubernetes clusters install a default set of ClusterRoles, representing common buckets users can be placed in. The “edit” role lets users perform basic actions like deploying pods; “view” lets a user observe non-sensitive resources;

“admin” allows a user to administer a namespace; and “cluster-admin” grants access to administer a cluster.

Disable Mounting of Service Account Token

Every namespace has a default service account

Pods use service accounts to assert their identity to other workloads, including the API server

When you create a pod, if you do not specify a service account, the pod will assign a default one for that namespace

The pod mounts these service account credentials to talk to the API server, so if a pod is compromised it can be used to perform arbitrary operations.

Restrict access to the Kubernetes API

The types of things we need to care about here are

Transport Security

Authentication

Authorization

Admission Control

But the big one, is API Server Ports and IPs

You need to really restrict the IP address ranges of what can hit the API server

Enable Pod security policy

Currently in Beta

Combo of AppArmor, seccomp, RunAsNonRoot, etc.

Defined in build

Enforced in Build & at Run Time

Restrict access to the Kubernetes Web UI (Dashboard)

Or better yet turn is off completely

The Dashboard is backed by a highly privileged Kubernetes Service Account.

If you are doing this via a cloud providers managed k8s offering, then use the the Cloud Console as it provides much of the same functionality, and you don't need these permissions.

Implement a Network Policy using pod labels

This allows you to control the network traffic between k8s pods

By default, pods are non-isolated; they accept traffic from any source.

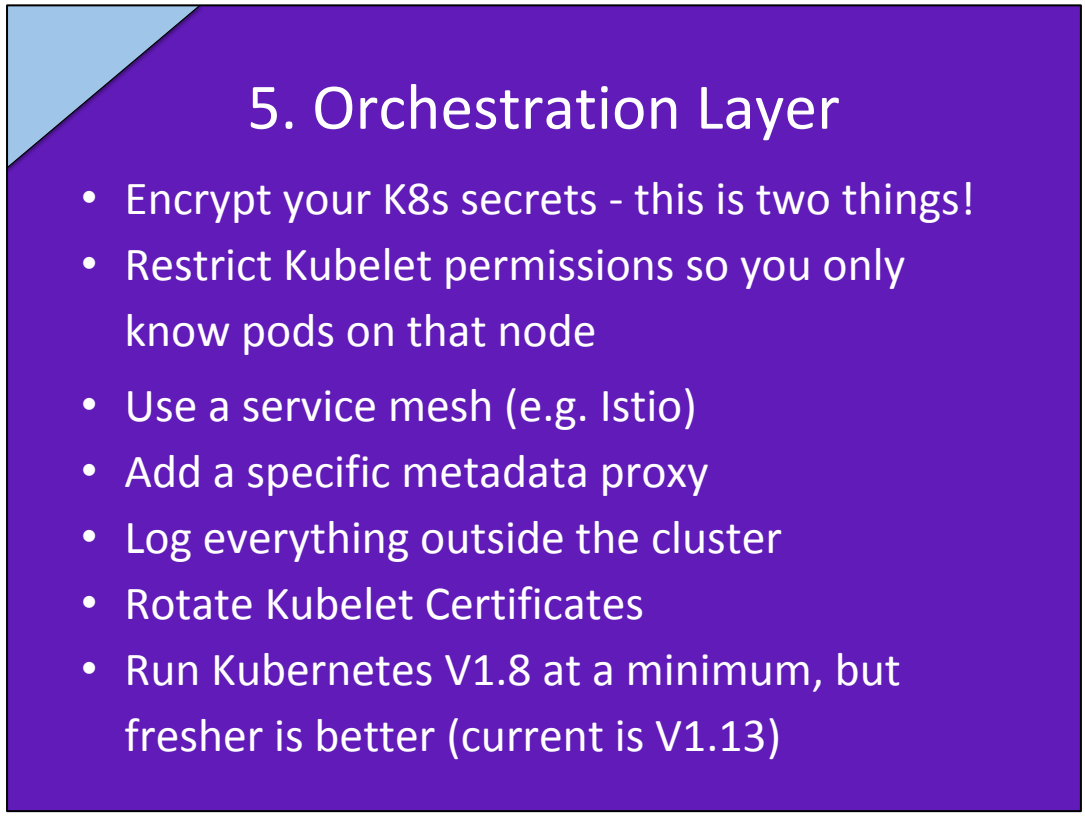
include a specific block to k8s dashboard

check that cloud providers provide built-in network policy support

Watch out for overlapping policies….one allows, the other blocks traffic…..

If you label your namespaces, this is a big help

Recommend: Securing Cluster Networking with Network Policies - Ahmet Balkan, Google https://www.youtube.com/watch?v=3gGpMmYeEO8



Encrypt your K8s secrets

By default, they are stored in plain text in your etcd folder
V1.10 introduced envelope encryption of secrets with a KMS provider, meaning that a local key is used to encrypt the secrets (known as a “data encryption key”), and that the key is itself encrypted with another key not stored in Kubernetes (a “key encryption key”).
This model means that you can regularly rotate the key encryption key without having to re-encrypt all the secrets.
Furthermore, it means that you can rely on an external root of trust for your secrets in Kubernetes—systems ike Cloud KMS or HashiCorp Vault.

Restrict Kubelet permissions so you only know pods on that node

Use a service mesh (e.g. Istio)

More on that next

Add a specific metadata proxy

Your cloud service provider make available metadata to help describe the k8s environment.

Problem is, attackers can use this against us.

do a search on github; GCP publish one there and lyft the ride sharing company publish theirs for AWS

Log everything outside the cluster

Not a container specific concept, but just good hygiene
Don’t give attackers an easy way to destroy their footprints

Rotate Kubelet Certificates

The kubelet uses certificates for authenticating to the Kubernetes API. By default, these certificates are issued with one year expiration so that they do not need to be renewed too frequently.

Once automed though, this could be accelerated depend on your level of paranoia

Kubernetes 1.8 contains kubelet certificate rotation, a feature that will automatically generate a new key and request a new certificate from the Kubernetes API as the current certificate approaches expiration.

Once the new certificate is available, it will be used for authenticating connections to the Kubernetes API.

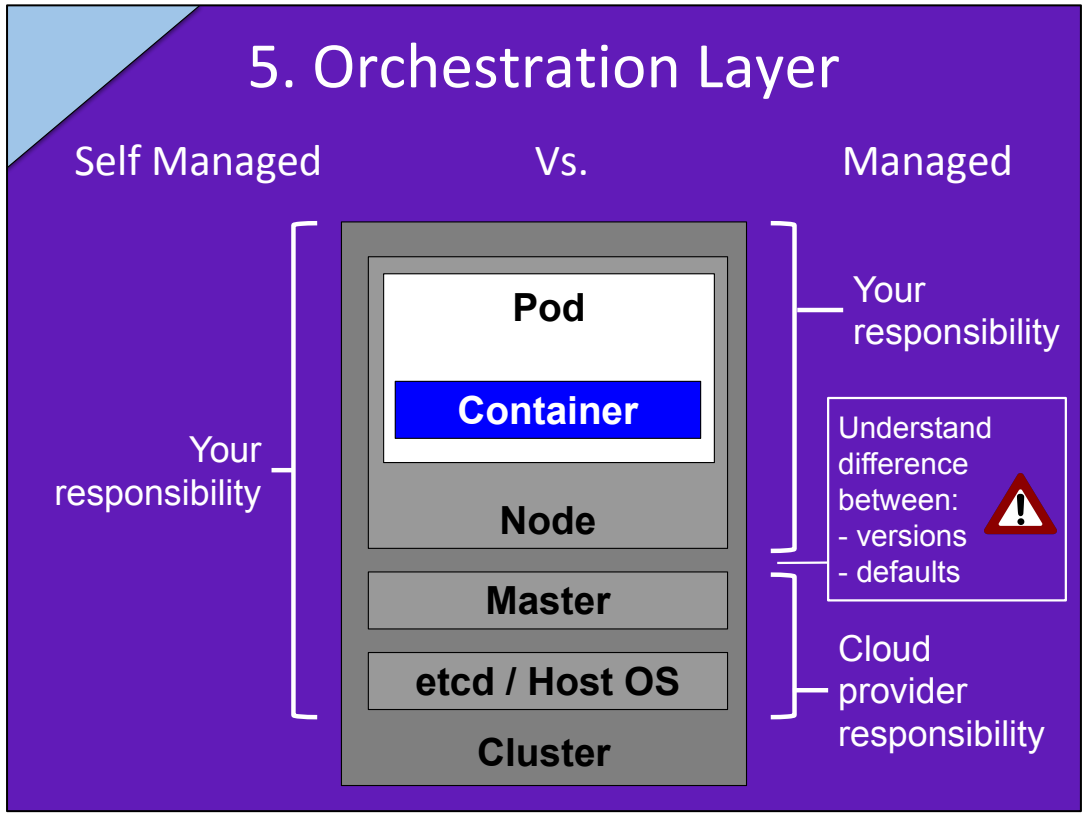
Run Kubernetes V1.8 at a minimum, but fresher is better

A lot of awesome defaults were included.

Big additions were RBAC, and change to role permission defaults

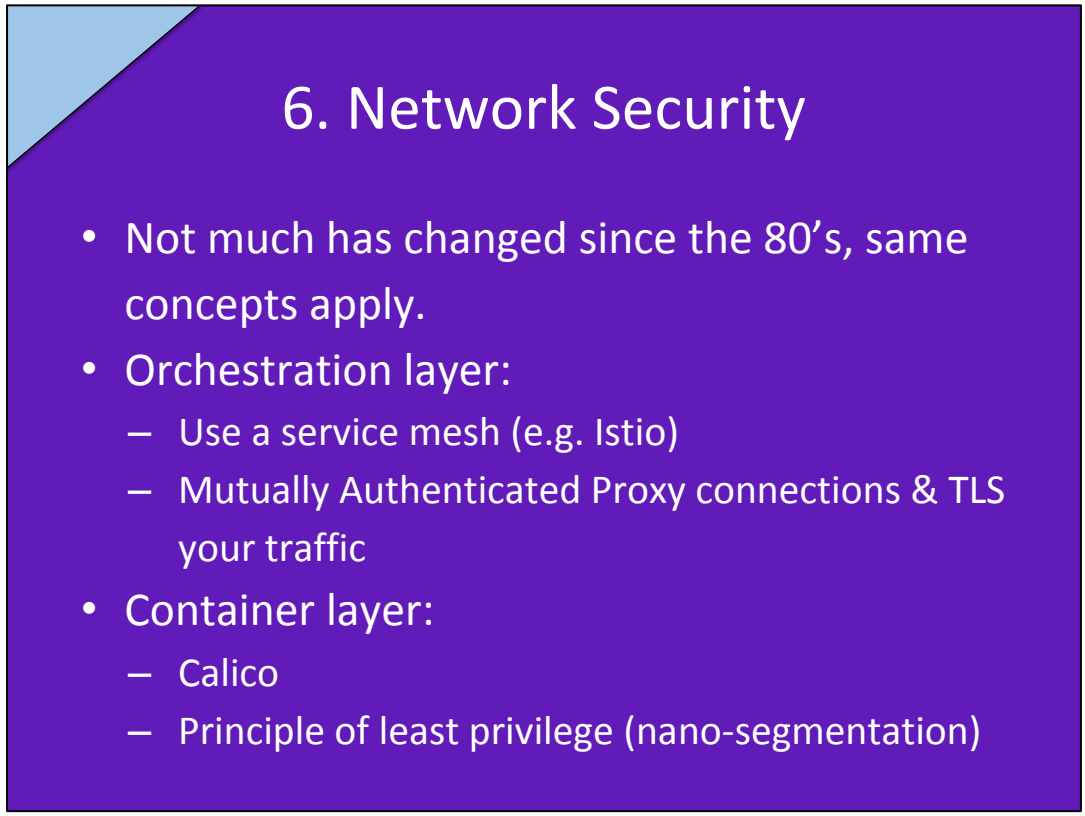
the creation of a network policy

Support to rotate certificates


5. Orchestration Layer
Self Managed
Vs.
Managed
Pod
Container
Node
Master
etcd / Host OS
Cluster
Your responsibility
Your responsibility
Understand difference between:
- versions
- defaults
Cloud provider responsibility

6. Network Security

- More surface area now that microservices are talking to each other
- Need to isolate discrete workloads
- Restrict traffic to unintended services
- Prevent egress to the internet
- Is logical isolation enough for a DMZ?



Orchestration layer:

Use a service mesh such as Istio

One great feature of Istio today is the ability to encrypt traffic in your service mesh with TLS. Your applications don't have to manage certificates or even enable TLS themselves: the sidecar intercepts and encrypts traffic before it leaves the pod. The Envoy proxies mutually authenticate connections and this makes the service mesh recognize "self" versus connections from outside the mesh. This can be enabled globally at present.

Container Layer:

Calico applies network routing and network policy rules to virtual interfaces for orchestrated containers

So consider using this to implement a principle of least privilege

The Calico Felix agent runs on each node, programs kernel routes to local workloads, and calculates and enforces the local filtering rules required by the current policies applied to the cluster.

Istio with Auth + Network Policy is like 2-factor authentication: an incoming connection has to have the right cryptographic identity and originate from the right IP.

6. Network Security

- Cloud Service:
  - Separate cloud accounts/VPCs/projects/resource groups for different workload groups
  - 0.0.0.0/0 CIDR ranges should be avoided
  - Limit privilege to change network policies
  - Segregation of duties
- Run regular checks against ‘known good’
- Limit your blast radius
- Log

Cloud service:

You have to determine the right balance for your organisation
The balance here is to perhaps have one account that all external connectivity is funneled through, and that one account can’t be changed by the app team (segregation of duties).

Key takeaways here are:

Limit your blast radius
Log your traffic

7. Logging & Auditing

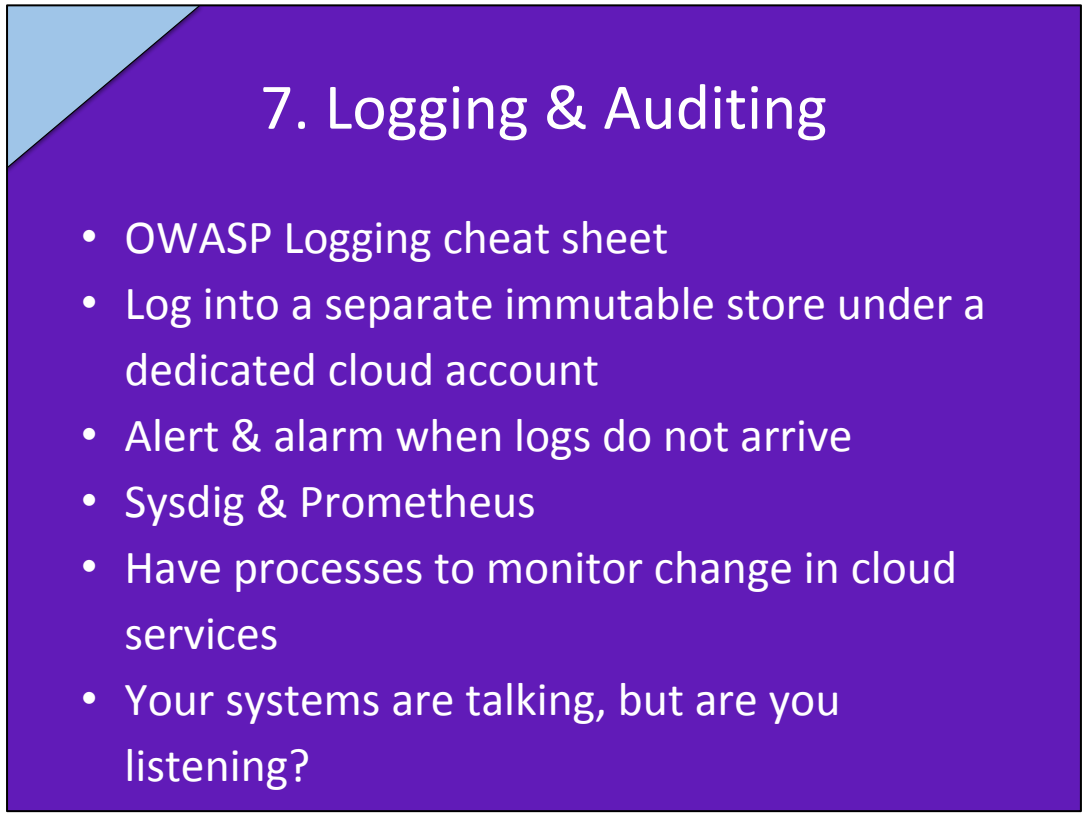
- Need to meet operational requirements
- Logging via the application may not be enough
- Containers are transient
- Need to meet forensic requirements
- Logs are not sufficiently protected
- Logs are not sanitised
- Signal to noise ratio
- Cloud services features keep changing

Pick your poison depending on the size and complexity of your environment:

Small & Simple = use a dedicated logging container in the cluster
Big & Complex = consider sidecar approach, however this is notably more complex to set up.

If you are using a managed kubernetes service from a cloud provider, this should not be quite so complex

Practice end-to-end logging - app, k8s, container, OS, cloud services
Don't forget about the full stack
You may have the underlying OS abstracted aware from you in a k8s service, but if not log it.
Also log who is logging into and out of your cloud accounts; making changes to network and IAM policies etc.



OWASP Logging cheat sheet

This is your friend!

provides developers with concentrated guidance on building application logging mechanisms, especially related to security logging.

There is also a more comprehensive guide if you need it

Log into a separate immutable store under a dedicated cloud account

Best practice dictates an immutable data store

Not optional for highly regulated entities

Just be mindful that maintaining access to old facts comes at a high price, like the cost of infinitely growing storage so use a data rotation policy as appropriate

Alert & alarm when logs do not arrive

Helps you identify at best misconfigured infrastructure or applications, or at worst ownage!

Sysdig & Prometheus

Containers by their nature introduce issues; issues with scaling, their ephemeral nature, and the sheer volume of data involved may dictate the need for dedicated container monitoring solutions like Prometheus and Sysdig.

My preference here is Sysdig, mainly because of sysdig falco (an add-on) which helps address the signal to noise ratio and focus in on the key indications such as:

- A shell is run inside a container
- A server process spawns a child process of an unexpected type
- Unexpected read of a sensitive file (like /etc/shadow)
- A non-device file is written to /dev

    - A standard system binary (like ls) makes an outbound network connection

Your systems are talking, but are you listening?

- Just remember, logs are a huge source of intel for attackers.
- They can enumerate services, in the most egregious offences that can also see sensitive customer data so be prudent about what you log
- And don’t allow logs to be turned off

Too Much?

Let’s focus on the key ones

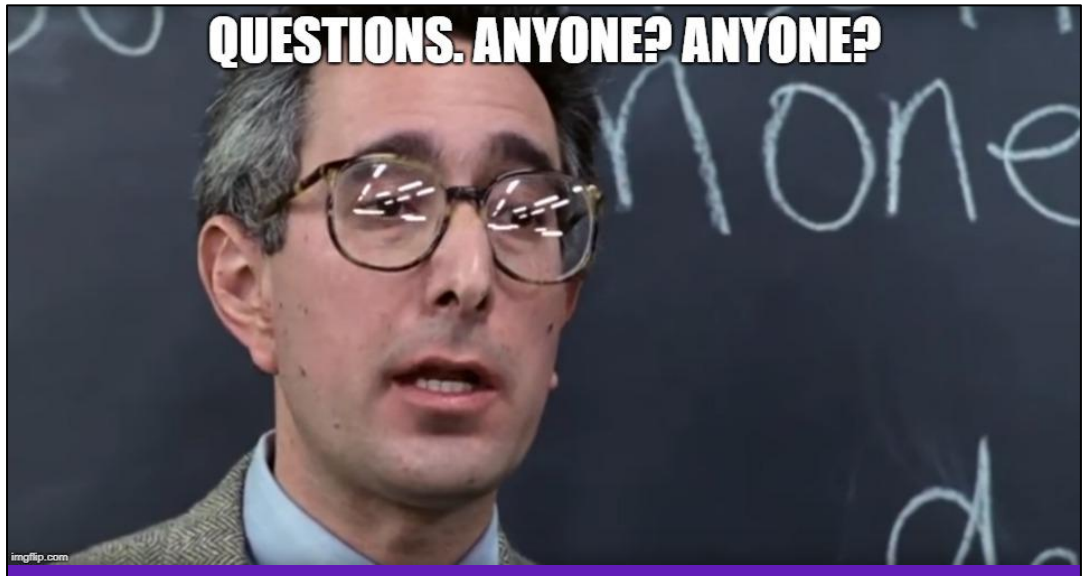
Essential Ingredients

- Pod Security Policy: non-root
- Docker Bench & Kube Bench Security
  - RBAC essential
- Run a minimal OS with no SSH; 400MB is large
- Restrictive Network Policy
- Cloud Account hardening: blast radius, segregation, CIDR ranges, metadata API proxy
- Enforce run time protection
- Protect your build environment like your app

There is a reason these two are at the top of the list

So if you have a plan to take this forward, what are some of the things to consider:

Secure defaults are pretty good...now; but not enough
Same problems; materially new ways to achieve solutions
Automation is key, not only to build but ...critically… to enforce ‘known good’ states
And that gets tricky….
Teams need to understand and adapt (both security & application teams)
This means that security teams may not be the ones operating these controls and previous segregation of duties get hard
Zero trust is possible
IMO there is value in commercial tooling that bring visibility & allows injection of control points across the full stack



www.linkedin.com/in/davidagrice

@Shh_Dontell Slides: https://git.io/fjoX7