Securing Containers on the High Seas

Jack Mannino @ OWASP Belgium
September 2018

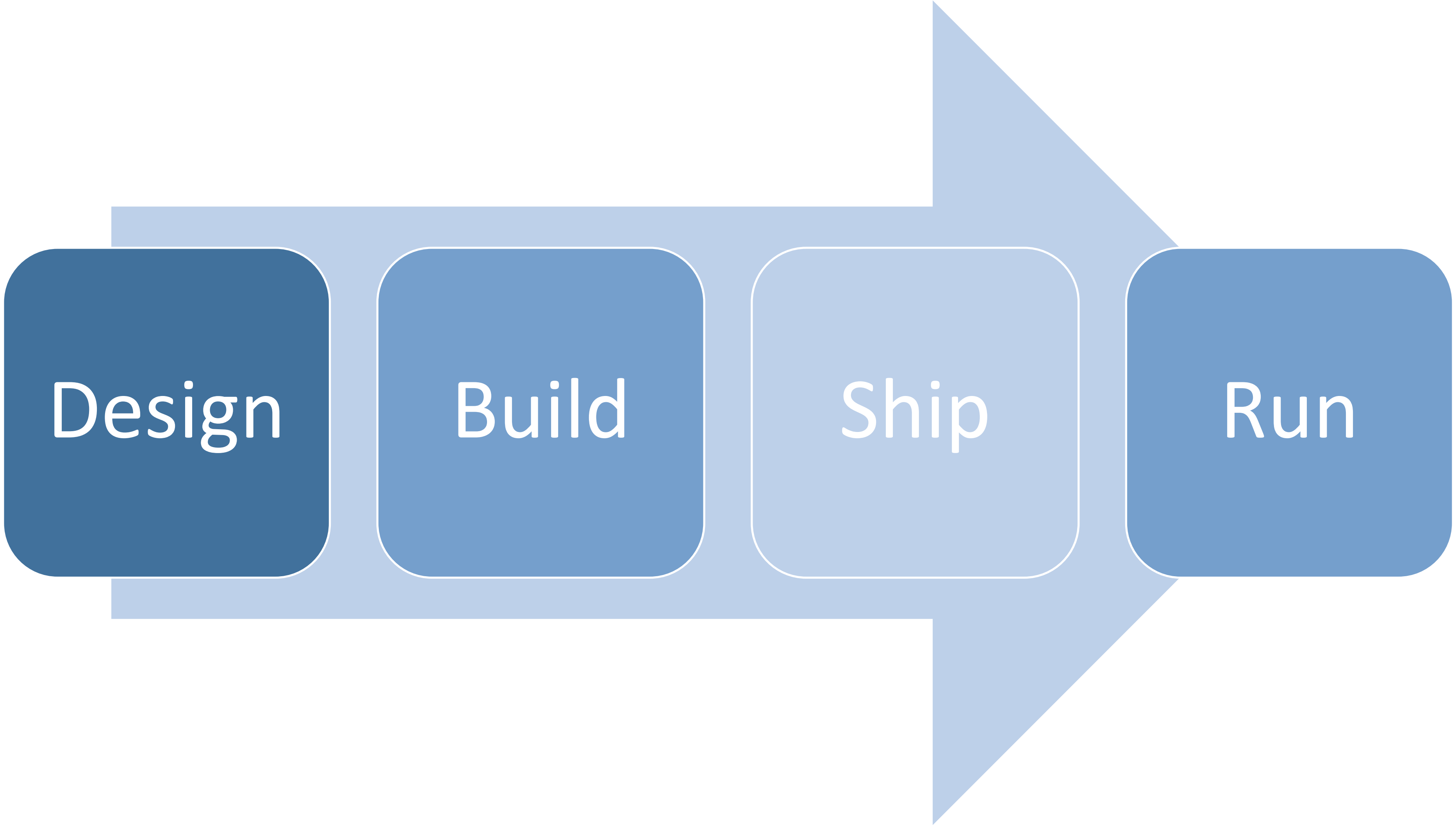
Who Am I?

Jack Mannino

- CEO at nVisium, since 2009
- Former OWASP Northern Virginia chapter leader
- Hobbies: Scala, Go and Kubernetes

nVISIUM

Container Security Lifecycle



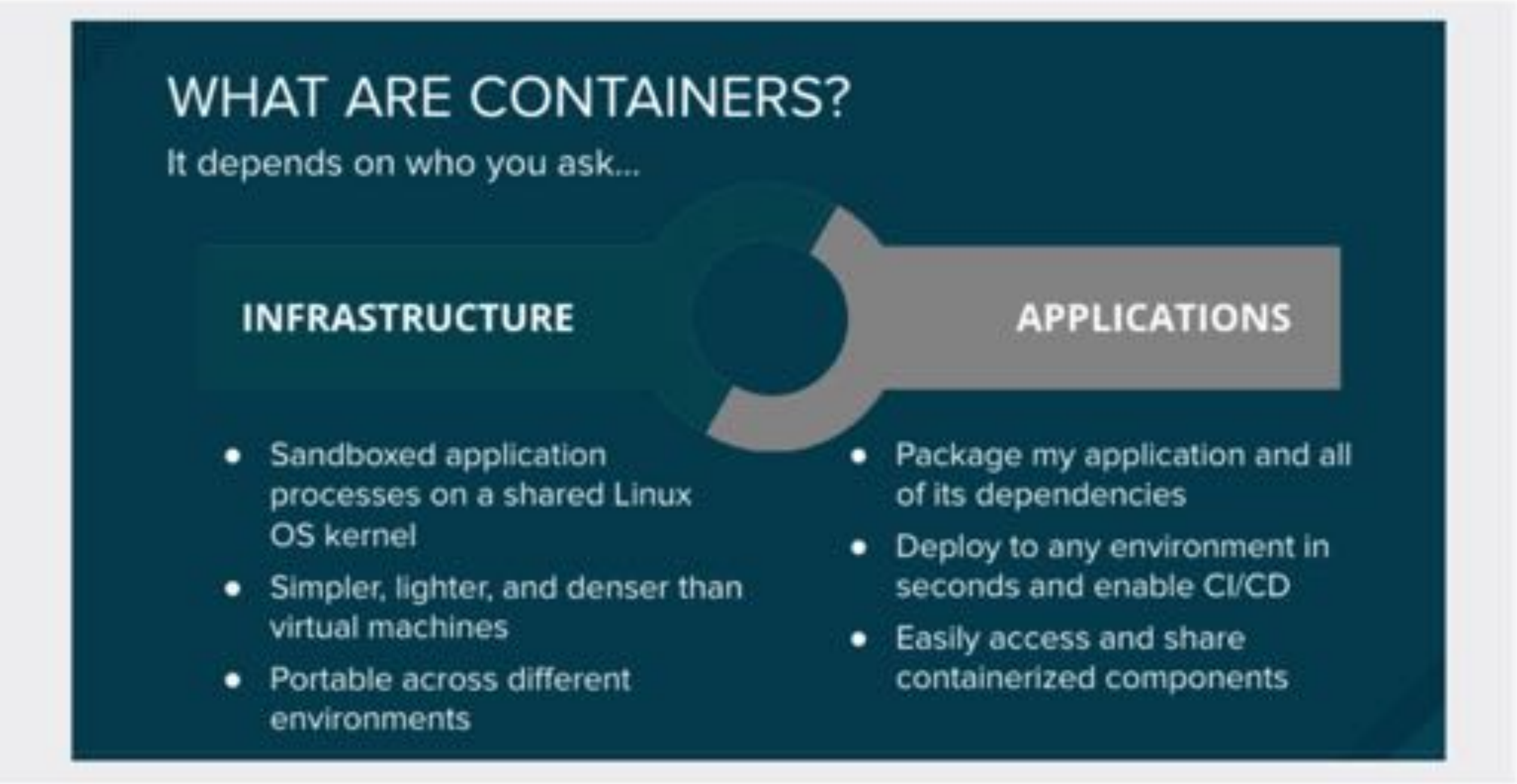
Design

Build

Ship

Run

Containers are __



WHAT ARE CONTAINERS?

It depends on who you ask...

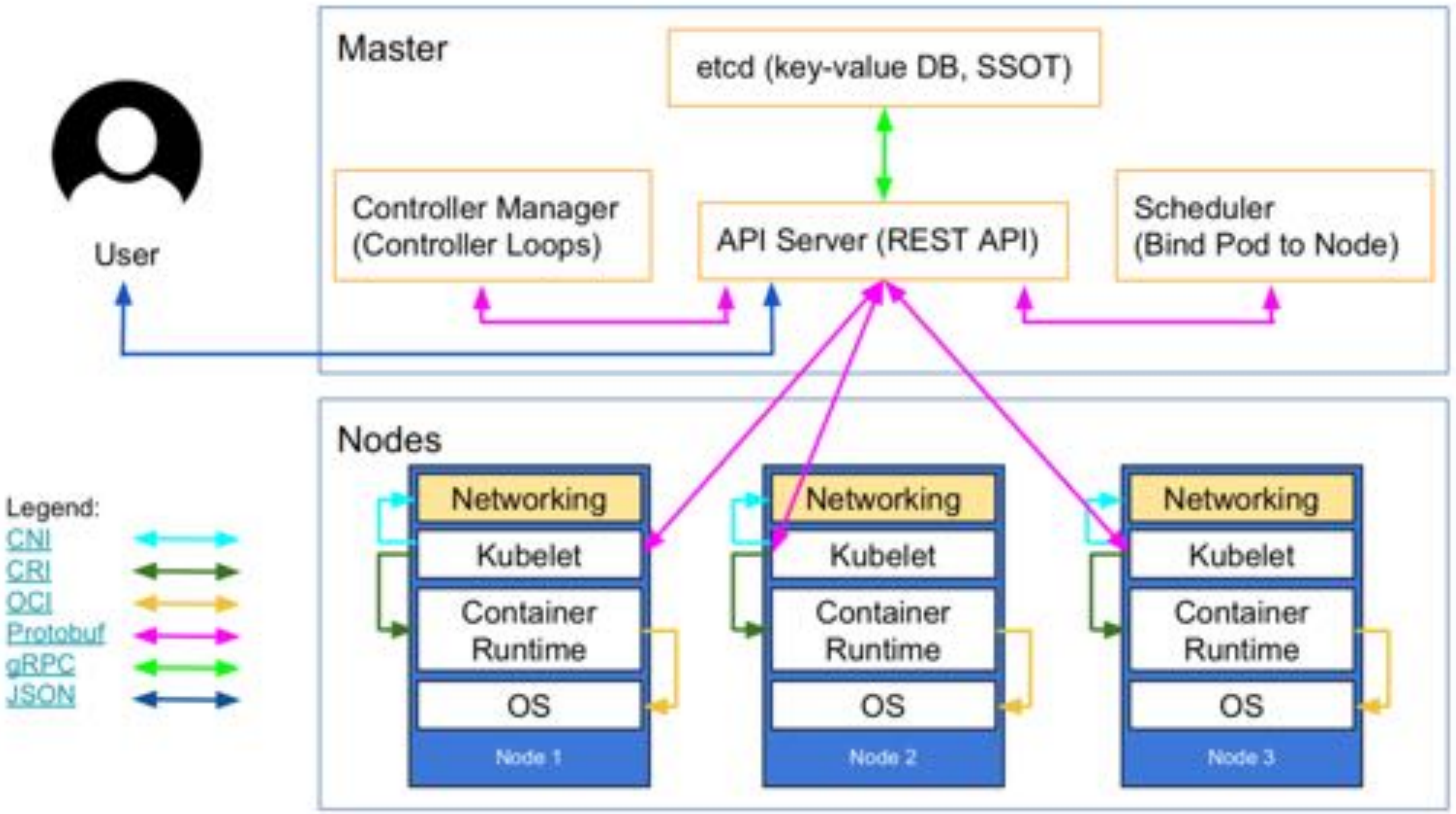
INFRASTRUCTURE

- Sandboxed application processes on a shared Linux OS kernel
- Simpler, lighter, and denser than virtual machines
- Portable across different environments

APPLICATIONS

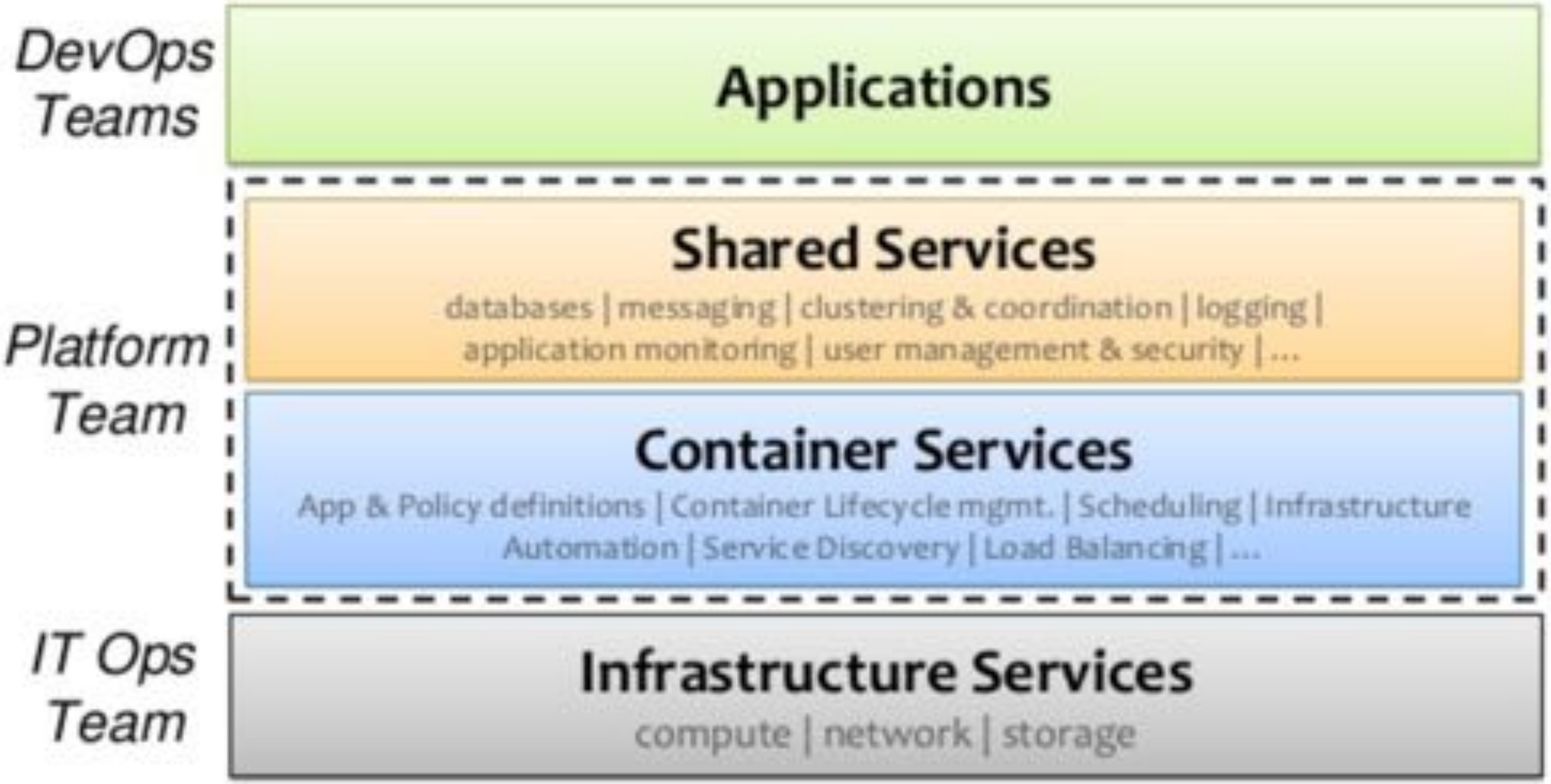
- Package my application and all of its dependencies
- Deploy to any environment in seconds and enable CI/CD
- Easily access and share containerized components

Containerized Architecture



https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/

Who Does What Now?



DevOps Teams

Applications

Platform Team

Shared Services

databases | messaging | clustering & coordination | logging | application monitoring | user management & security | ...

Container Services

App & Policy definitions | Container Lifecycle mgmt. | Scheduling | Infrastructure Automation | Service Discovery | Load Balancing | ...

IT Ops Team

Infrastructure Services

compute | network | storage

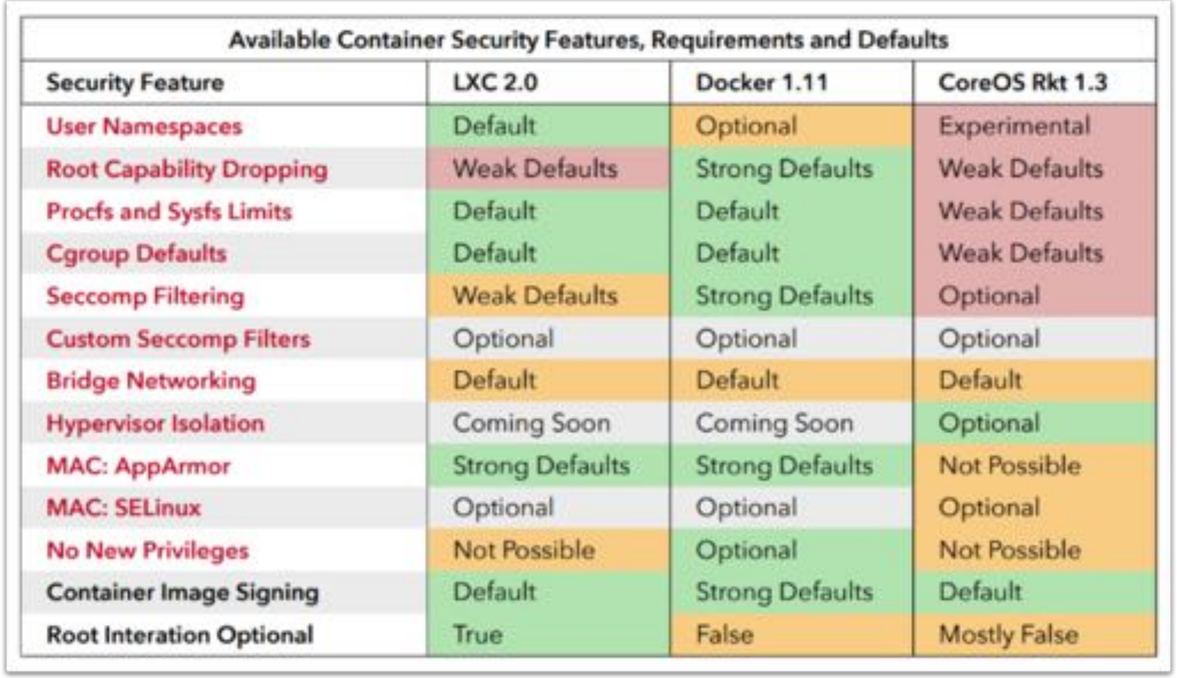
Design

Secure Architecture

- ✓ Orchestration & Management - Control Plane
- ✓ Network Segmentation & Isolation
- ✓ Encrypted communications
- ✓ Authentication (container & cluster-level)
- ✓ Identity Management & Access Control
- ✓ Secrets Management
- ✓ Logging & Monitoring

Picking the Right Container Runtime

- Open Container Initiative (OCI) spec promotes a broader set of container tech (life beyond Docker)
- Isolate containerized resources differently
- Goal is to prevent escaping from the container
- Isolation via Namespaces & Control Groups
- Isolation via Hypervisor



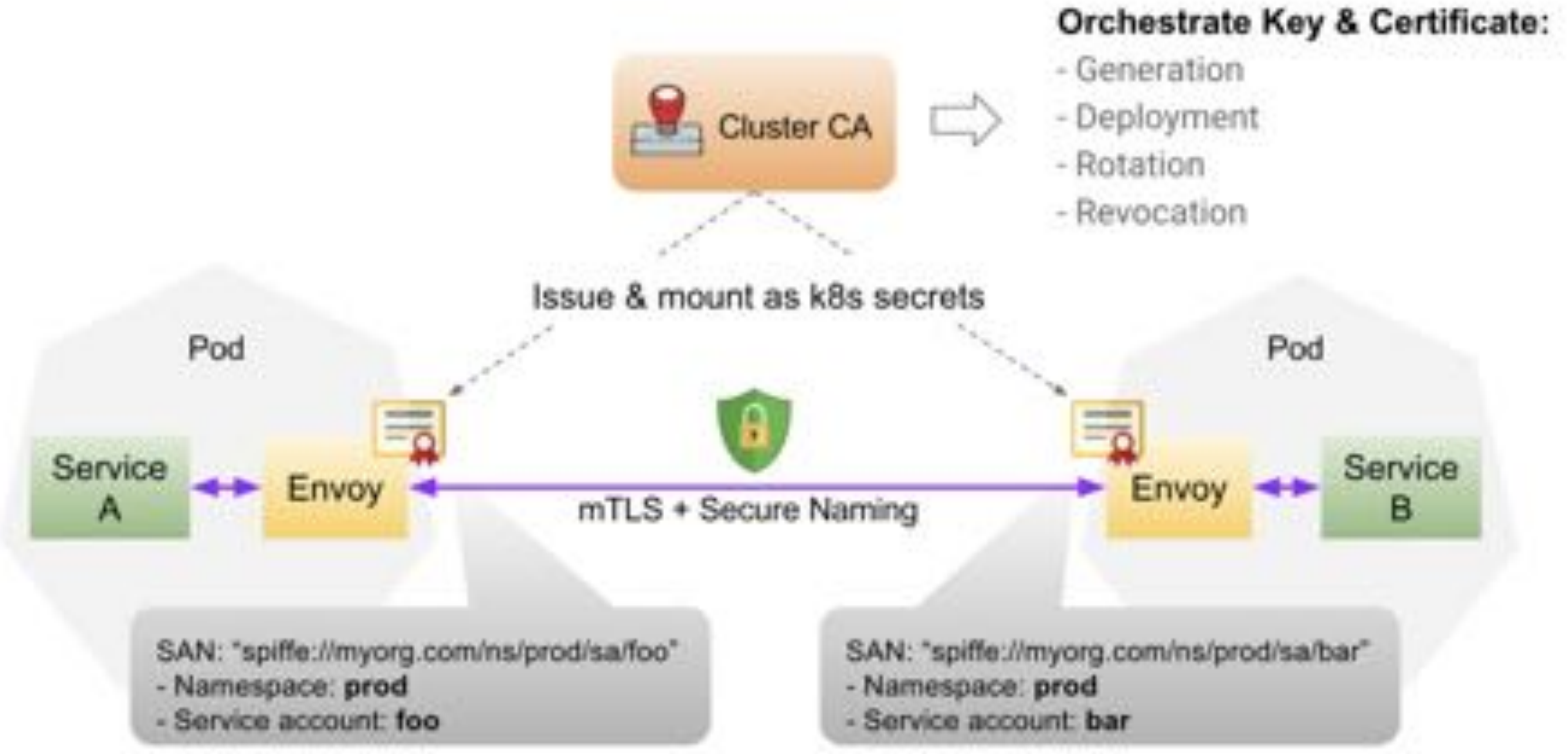
Available Container Security Features, Requirements and Defaults

Security Feature	LXC 2.0	Docker 1.11	CoreOS Rkt 1.3
User Namespaces	Default	Optional	Experimental
Root Capability Dropping	Weak Defaults	Strong Defaults	Weak Defaults
Procfs and Sysfs Limits	Default	Default	Weak Defaults
Cgroup Defaults	Default	Default	Weak Defaults
Seccomp Filtering	Weak Defaults	Strong Defaults	Optional
Custom Seccomp Filters	Optional	Optional	Optional
Bridge Networking	Default	Default	Default
Hypervisor Isolation	Coming Soon	Coming Soon	Optional
MAC: AppArmor	Strong Defaults	Strong Defaults	Not Possible
MAC: SELinux	Optional	Optional	Optional
No New Privileges	Not Possible	Optional	Not Possible
Container Image Signing	Default	Strong Defaults	Default
Root Interation Optional	True	False	Mostly False

https://blog.jessfraz.com/post/containers-security-and-echo-chambers/

Leveraging Design Patterns for Security

We can solve security issues through patterns that lift security out of the container itself. Example – Service Mesh with Istio & Envoy

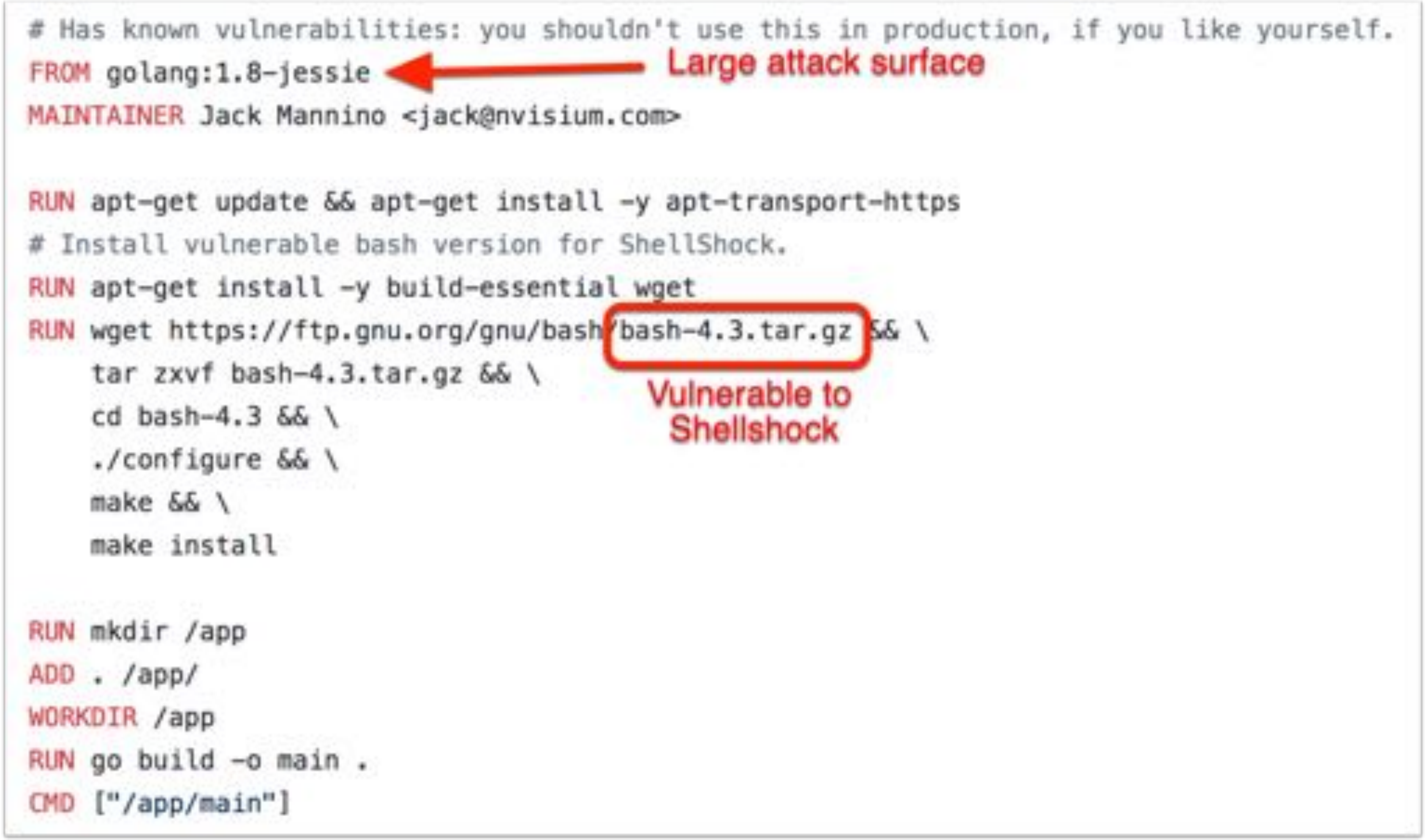


Build

Securing the Build Process

- Build steps focus on code repositories and container registries
- Run Tests -> Package Apps -> Build Image
- Build first level of security controls into containers
- Orchestration & management systems can override these controls and mutate containers through an extra layer of abstraction

Example: Insecurely Configured Docker Container



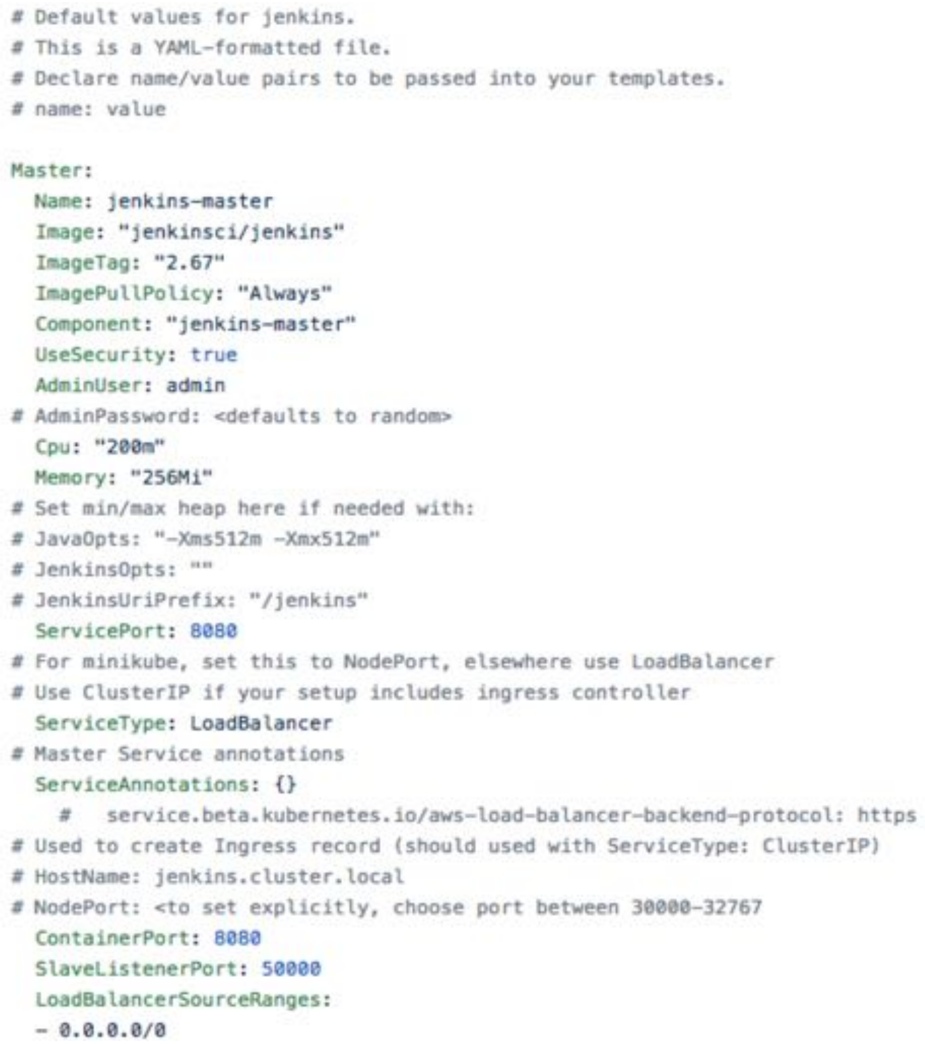
```
# Has known vulnerabilities: you shouldn't use this in production, if you like yourself.
FROM golang:1.8-jessie
MAINTAINER Jack Mannino <jack@nvisium.com>

RUN apt-get update && apt-get install -y apt-transport-https
# Install vulnerable bash version for ShellShock.
RUN apt-get install -y build-essential wget
RUN wget https://ftp.gnu.org/gnu/bash/bash-4.3.tar.gz && \
    tar zxvf bash-4.3.tar.gz && \
    cd bash-4.3 && \
    ./configure && \
    make && \
    make install

RUN mkdir /app
ADD . /app/
WORKDIR /app
RUN go build -o main .
CMD ["/app/main"]
```

Large attack surface

Vulnerable to Shellshock

Other Configuration Formats

- Your resources may be built with external tools, formats, or code
- Terraform (.tf), CloudFormation, Helm/Charts, Brigade, Metaparticle, etc.
- Create reproducible builds to streamline deployments
- Example – Helm/Charts use Go templates



```yaml
# Default values for jenkins.
# This is a YAML-formatted file.
# Declare name/value pairs to be passed into your templates.
# name: value

Master:
  Name: jenkins-master
  Image: "jenkinsci/jenkins"
  ImageTag: "2.67"
  ImagePullPolicy: "Always"
  Component: "jenkins-master"
  UseSecurity: true
  AdminUser: admin
# AdminPassword: <defaults to random>
  Cpu: "200m"
  Memory: "256Mi"
# Set min/max heap here if needed with:
# JavaOpts: "-Xms512m -Xmx512m"
# JenkinsOpts: ""
# JenkinsUriPrefix: "/jenkins"
  ServicePort: 8080
# For minikube, set this to NodePort, elsewhere use LoadBalancer
# Use ClusterIP if your setup includes ingress controller
  ServiceType: LoadBalancer
# Master Service annotations
  ServiceAnnotations: {}
    #   service.beta.kubernetes.io/aws-load-balancer-backend-protocol: https
# Used to create Ingress record (should used with ServiceType: ClusterIP)
# HostName: jenkins.cluster.local
# NodePort: <to set explicitly, choose port between 30000-32767
  ContainerPort: 8080
  SlaveListenerPort: 50000
  LoadBalancerSourceRanges:
  - 0.0.0.0/0
```

Chart for Jenkins
https://github.com/kubernetes/charts/blob/master/stable/jenkins/values.yaml

Base Image Management

- Focus on keeping the attack surface small
- Use base images that ship with minimal installed packages and dependencies
- Use version tags vs. image:latest
- Use images that support security kernel features (seccomp, apparmor, SELinux)

```
$ grep CONFIG_SECCOMP= /boot/config-$(uname -r)
$ cat /sys/module/apparmor/parameters/enabled
```

Restricting Root Capabilities

- Circa 2003, root privileges were broken into a subset of capabilities.
- This feature enables us to reduce the damage a compromised root account can do.
- Docker default profile allows 14 of 40+ capabilities.
- Open Container Initiative (OCI) spec restricts this this even further:
  - AUDIT_WRITE
  - KILL
  - NET_BIND_SERVICE

Docker Default Capabilities

- CHOWN
- DAC_OVERRIDE
- FOWNER
- FSETID
- KILL
- SETGID
- SETUID
- SETPCAP
- NET_BIND_SERVICE
- NET_RAW
- SYS_CHROOT
- MKNOD
- AUDIT_WRITE
- SETFCAP

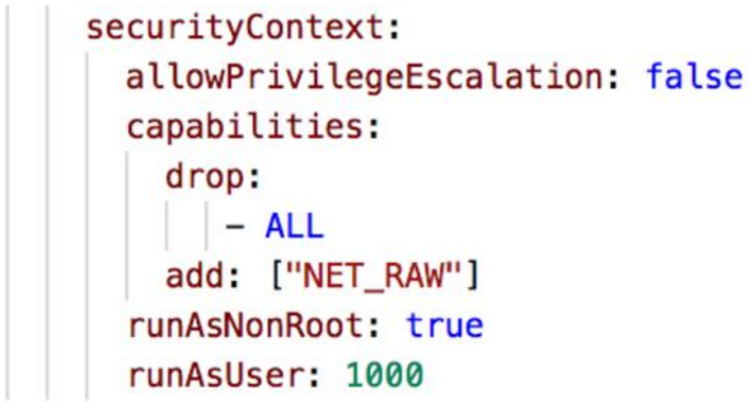
Limiting Privileges

- More often than not, your container does not need root
- Often, we only need a subset of capabilities
- Limit access to underlying host resources (network, storage, or IPC)

Example – Ping command requires CAP_NET_RAW

We can drop everything else.

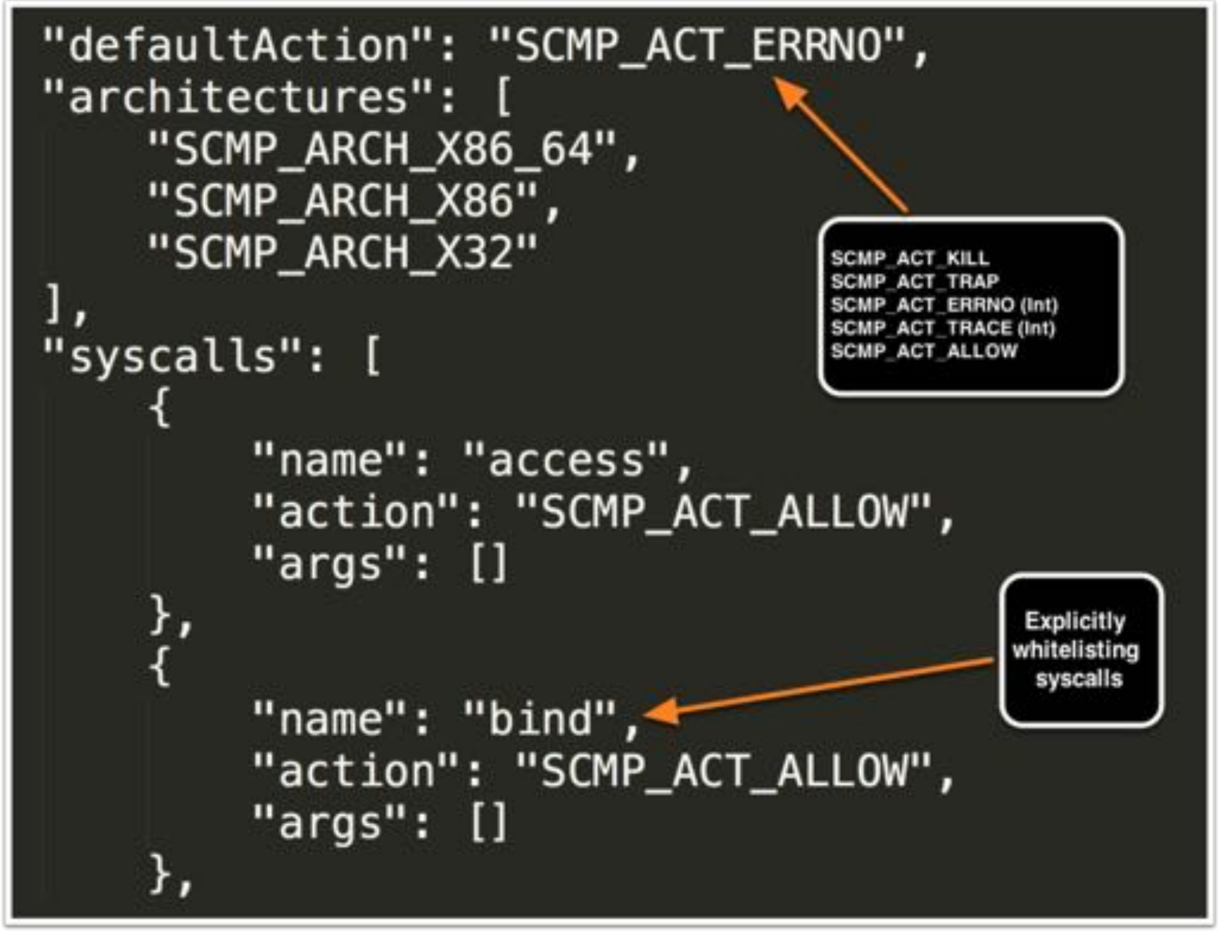
```
docker run -d --cap-drop=all --cap-add=net_raw my-image
```



```
securityContext:
  allowPrivilegeEscalation: false
  capabilities:
    drop:
      - ALL
    add: ["NET_RAW"]
  runAsNonRoot: true
  runAsUser: 1000
```

Kernel Hardening

- Restrict the actions a container can perform
- Seccomp is a linux kernel feature that allows you to filter dangerous syscalls
- Docker has a great default profile to get started



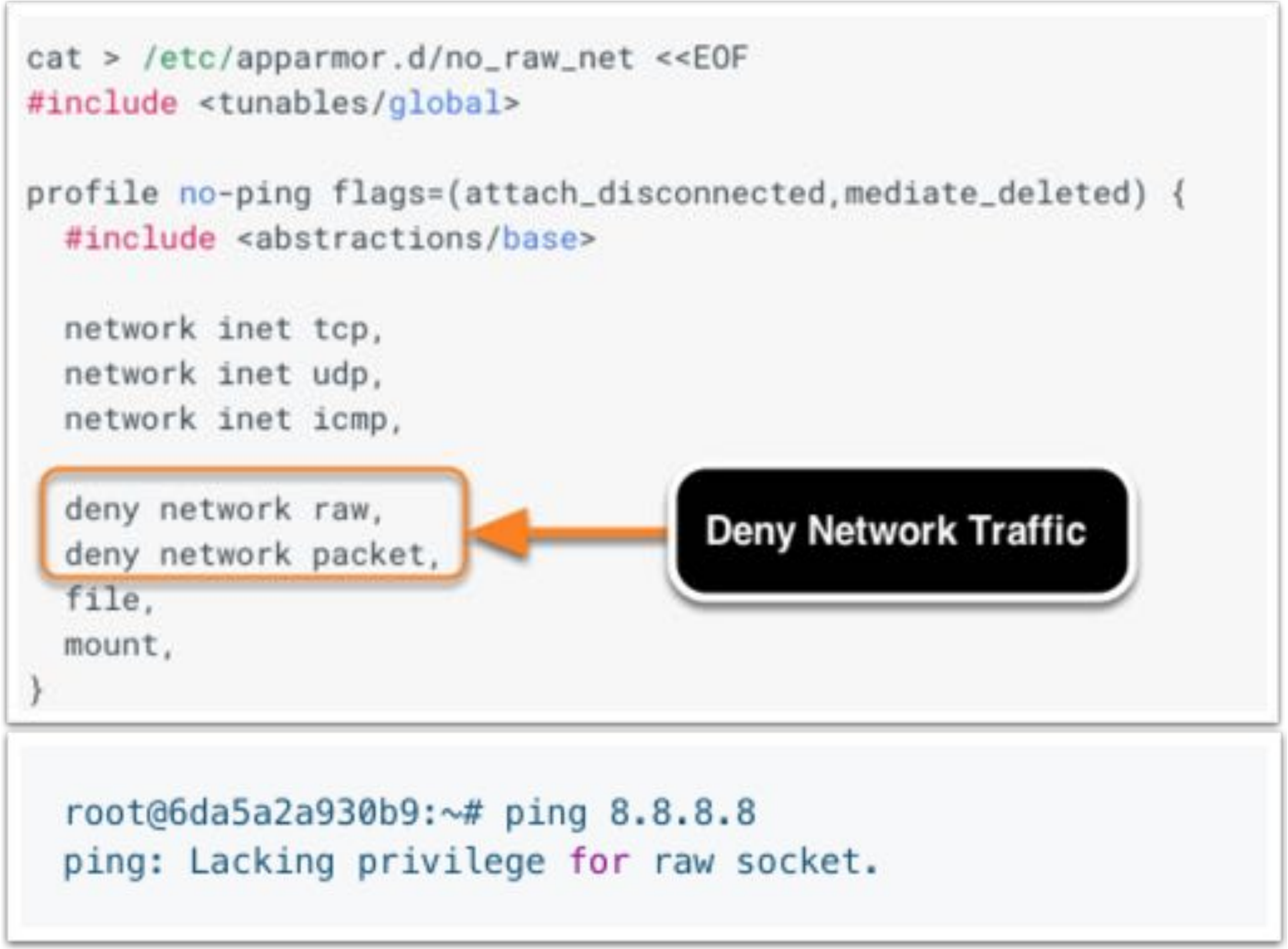
```
"defaultAction": "SCMP_ACT_ERRNO",
"architectures": [
    "SCMP_ARCH_X86_64",
    "SCMP_ARCH_X86",
    "SCMP_ARCH_X32"
],
"syscalls": [
    {
        "name": "access",
        "action": "SCMP_ACT_ALLOW",
        "args": []
    },
    {
        "name": "bind",
        "action": "SCMP_ACT_ALLOW",
        "args": []
    },
```

SCMP_ACT_KILL
SCMP_ACT_TRAP
SCMP_ACT_ERRNO (Int)
SCMP_ACT_TRACE (Int)
SCMP_ACT_ALLOW

Explicitly whitelisting syscalls

Mandatory Access Control (MAC)

- SELinux and AppArmor allow you to set granular controls on files and network access.
- Limits what a process can access or do
- Logging to identify violations (during testing and production)
- Docker leads the way with its default AppArmor profile



```
cat > /etc/apparmor.d/no_raw_net <<EOF
#include <tunables/global>

profile no-ping flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>

  network inet tcp,
  network inet udp,
  network inet icmp,

  deny network raw,
  deny network packet,
  file,
  mount,
}
```

Deny Network Traffic

```
root@6da5a2a930b9:~# ping 8.8.8.8
ping: Lacking privilege for raw socket.
```

Container Package Management

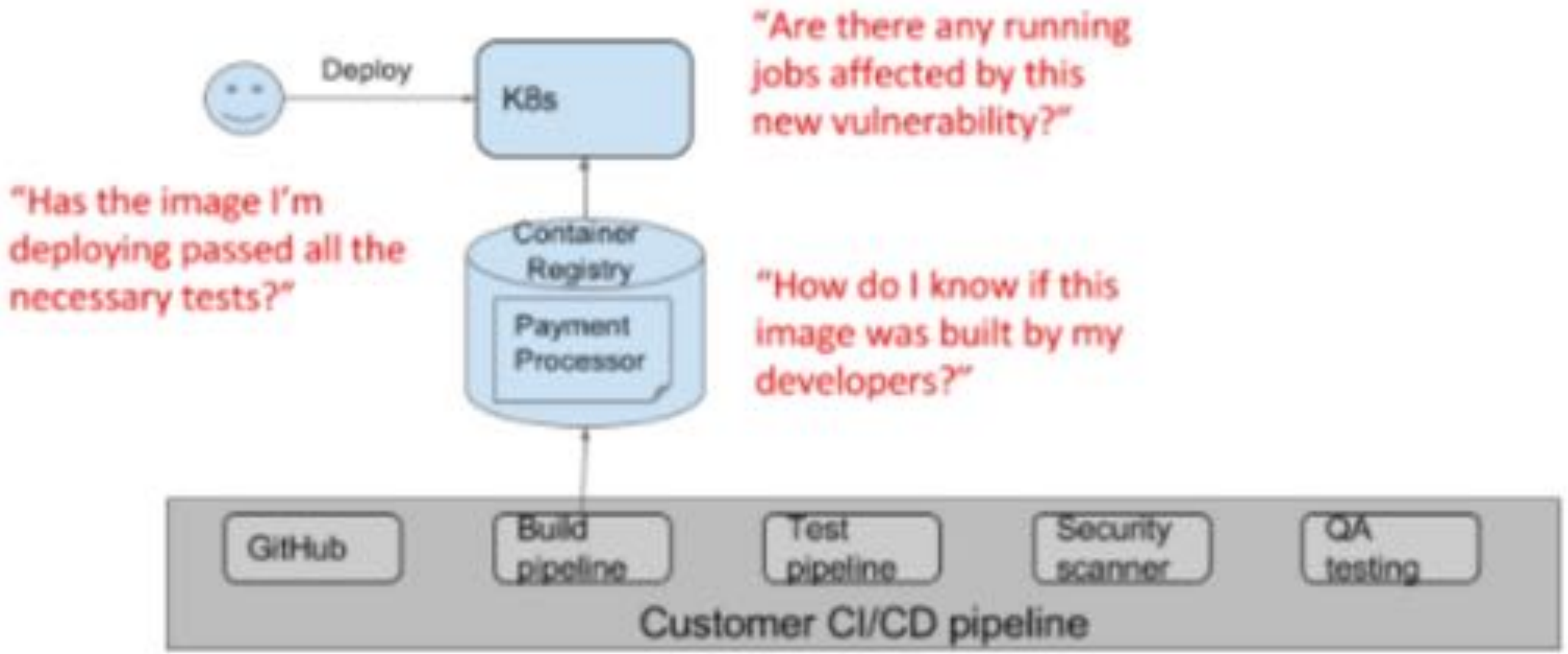
- Vulnerabilities can possibly exist in:
  - Container configurations
  - Container packages
  - Application Code & Libraries
- Solutions:
  - Clair
  - Dependency Check
  - Brigade
  - Commercial tools

Ship

Ship

- Securely move the container from registry -> runtime environment
- Controlled container promotion and deployment
- Validate the integrity of the container
- Validate security pre-conditions

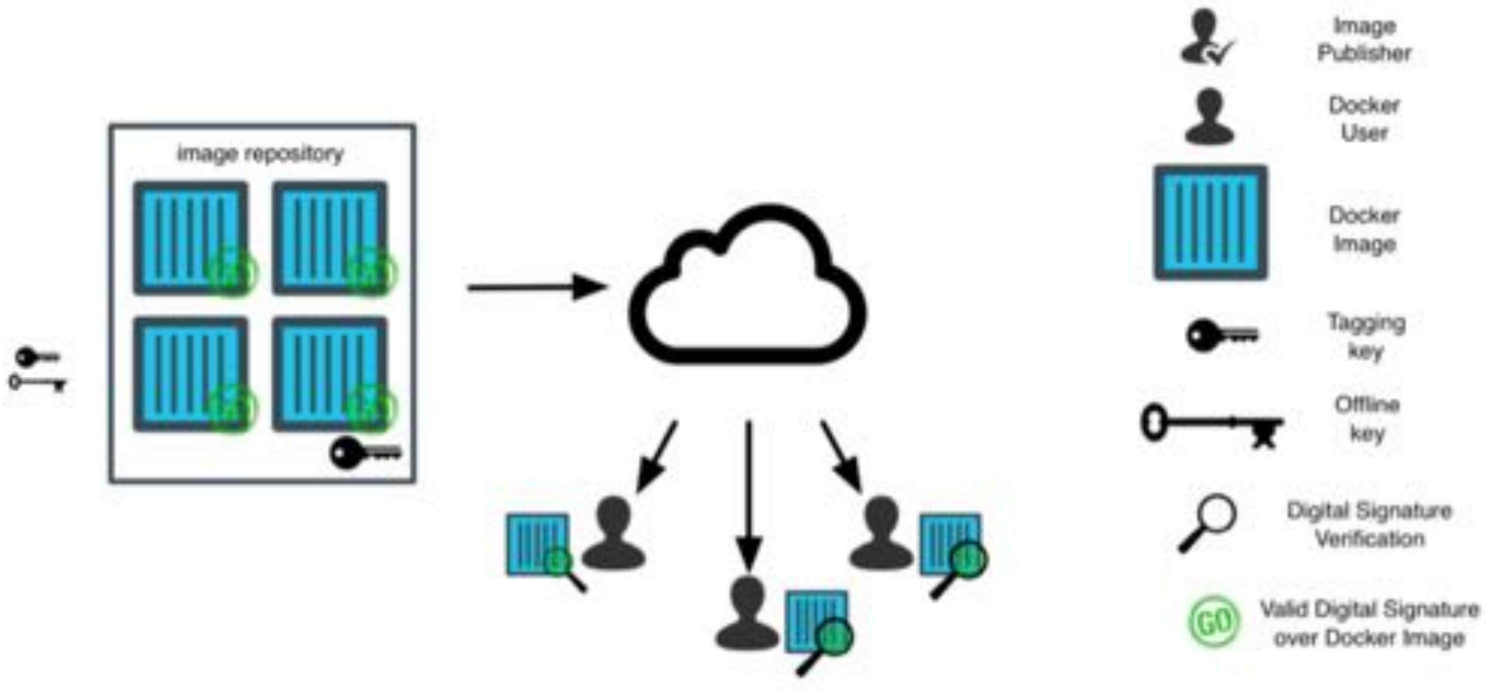
What Am I Even Shipping?


Deploy
K8s
"Are there any running jobs affected by this new vulnerability?"
"Has the image I'm deploying passed all the necessary tests?"
Container Registry
Payment Processor
"How do I know if this image was built by my developers?"
GitHub
Build pipeline
Test pipeline
Security scanner
QA testing
Customer CI/CD pipeline

https://kubernetes.io/blog/2017/11/securing-software-supply-chain-grafeas/

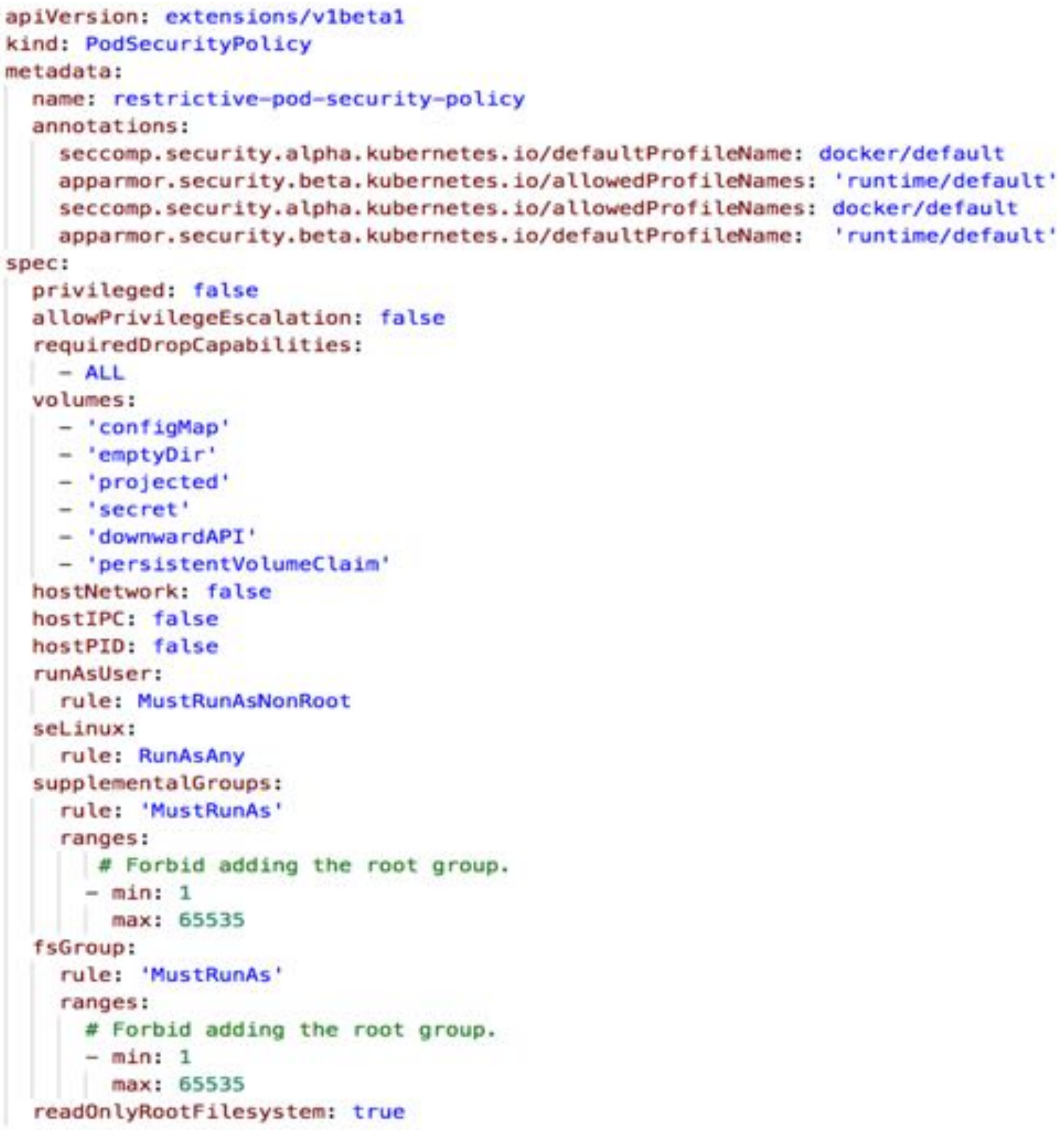
Validating Integrity & Signing Builds

- Ensures integrity of the images and publisher attestation
- Sign to validate pipeline phases
- Example – Docker Content Trust & Notary, GCP's Binary Authorization
- Consume only trusted content for tagged builds



Validating Security Pre-Conditions

- Allow or deny a container's cluster admission
- Centralized interfaces and validation
- Mutate a container's security before admission
- Example – Kubernetes calls this a *PodSecurityPolicy*



```yaml
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive-pod-security-policy
  annotations:
    seccomp.security.alpha.kubernetes.io/defaultProfileName: docker/default
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: docker/default
    apparmor.security.beta.kubernetes.io/defaultProfileName:  'runtime/default'
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: MustRunAsNonRoot
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  readOnlyRootFilesystem: true
```

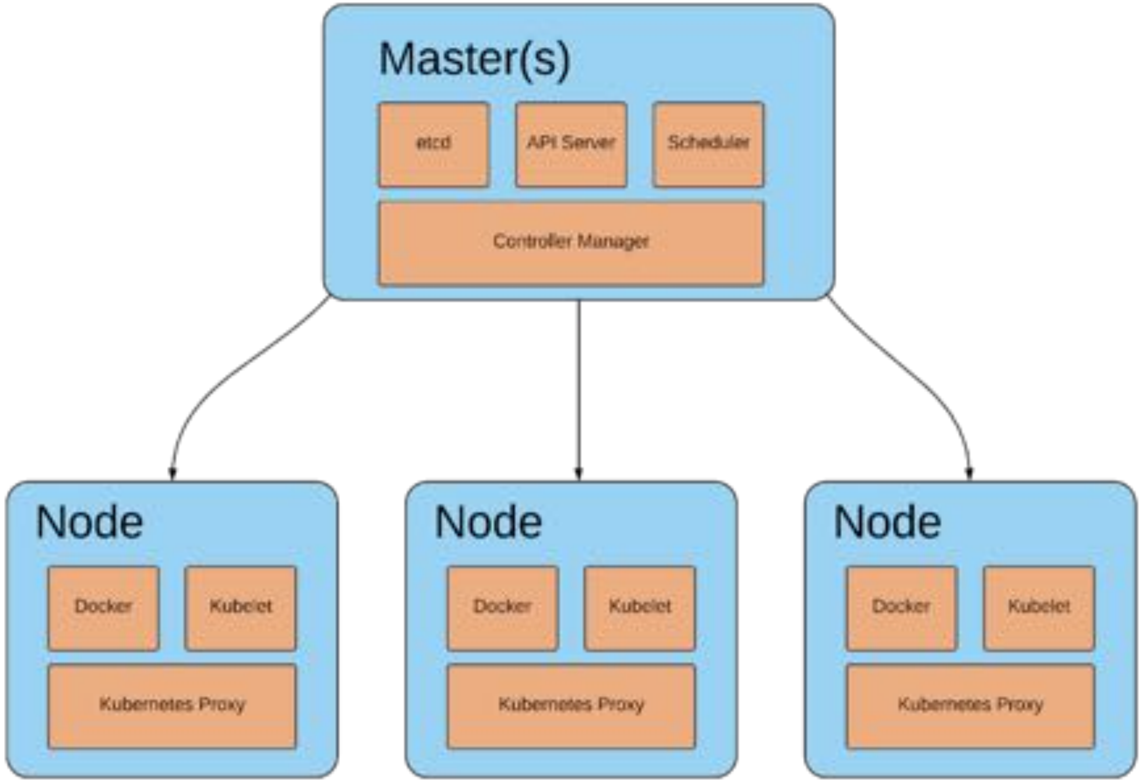
Run

Run

Typically, containers are managed, scheduled, and scaled through orchestration systems.

Kubernetes, Mesos, Docker Swarm, AWS ECS, etc.

- Cluster/Service authentication
- Identity Management & Access Control
- Policy & Constraint Enforcement
- Propagation of secrets
- Logging & Monitoring



Example – Kubernetes Control Plane

Control Plane Hardening

- The Control Plane manages the cluster’s state and schedules containers.
- A privileged attack against a control plane node or pod can have serious consequences.
- Managed services such as Azure AKS, AWS EKS and Google Cloud Platform’s GKE abstract away the control plane for you.

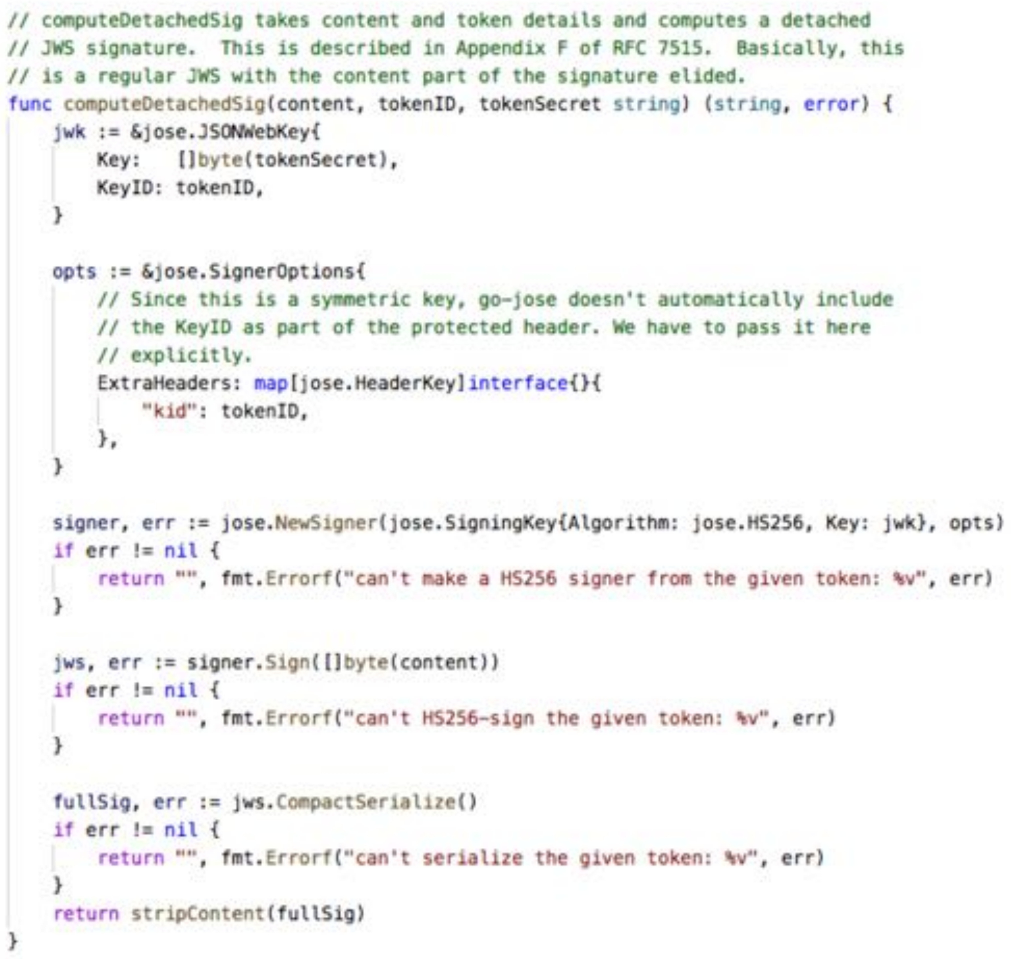
Management APIs

- Deploy, modify, and kill services
- Run commands inside of containers
- Kubernetes, Marathon, and Swarm APIs work similarly
- *Frequently deployed without authentication or access control*

Authentication

- Authenticate subjects (users and service accounts) to the cluster
- Authentication occurs at several layers
  - Authenticating API subjects
  - Authenticating nodes to the cluster
  - Authenticating services to each other

Avoid sharing service accounts across multiple services!



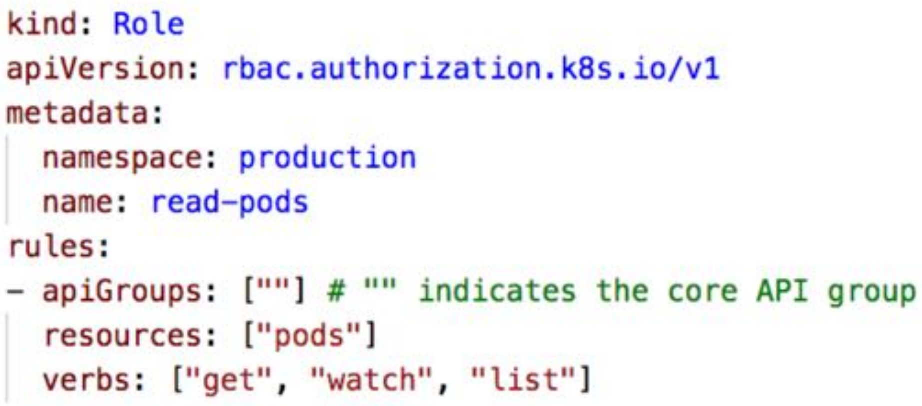
```go
// computeDetachedSig takes content and token details and computes a detached
// JWS signature.  This is described in Appendix F of RFC 7515.  Basically, this
// is a regular JWS with the content part of the signature elided.
func computeDetachedSig(content, tokenID, tokenSecret string) (string, error) {
    jwk := &jose.JSONWebKey{
        Key:    []byte(tokenSecret),
        KeyID: tokenID,
    }

    opts := &jose.SignerOptions{
        // Since this is a symmetric key, go-jose doesn't automatically include
        // the KeyID as part of the protected header. We have to pass it here
        // explicitly.
        ExtraHeaders: map[jose.HeaderKey]interface{}{
            "kid": tokenID,
        },
    }

    signer, err := jose.NewSigner(jose.SigningKey{Algorithm: jose.HS256, Key: jwk}, opts)
    if err != nil {
        return "", fmt.Errorf("can't make a HS256 signer from the given token: %v", err)
    }

    jws, err := signer.Sign([]byte(content))
    if err != nil {
        return "", fmt.Errorf("can't HS256-sign the given token: %v", err)
    }

    fullSig, err := jws.CompactSerialize()
    if err != nil {
        return "", fmt.Errorf("can't serialize the given token: %v", err)
    }
    return stripContent(fullSig)
}
```

Example – K8s JWT Generator

Authorization & Access Control

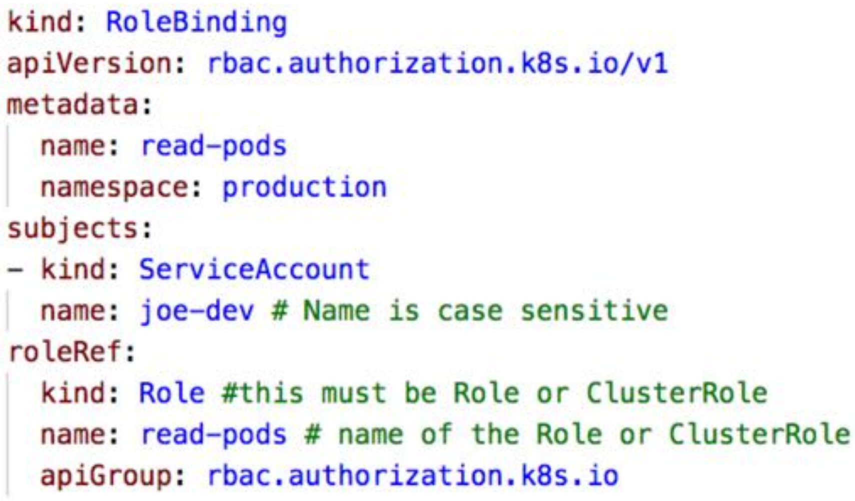
- Subjects should only have access to the resources they need
- Limit what a single hostile user or container can achieve)
- Multiple vantage points - to the API, between containers, between control plane components

K8s - Create a Role



```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: production
  name: read-pods
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

K8s - Bind a Subject to the Role



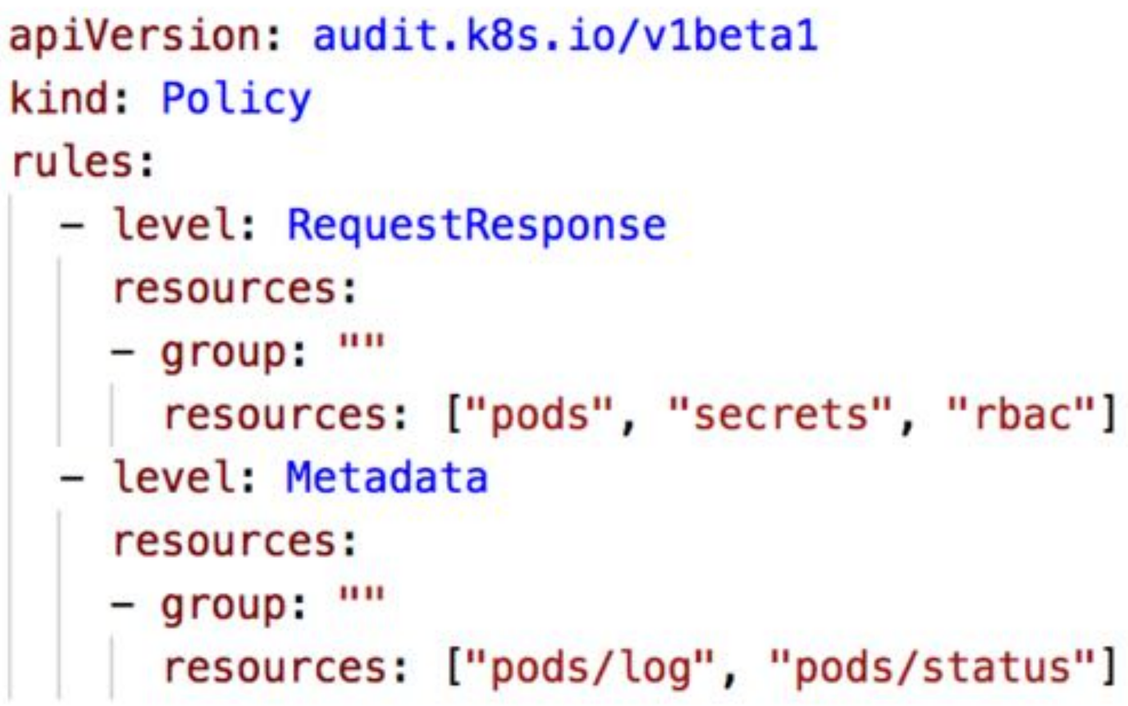
```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: production
subjects:
- kind: ServiceAccount
  name: joe-dev # Name is case sensitive
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: read-pods # name of the Role or ClusterRole
  apiGroup: rbac.authorization.k8s.io
```

Logging and Monitoring

- OWASP Top 10 2017 – A10 = Insufficient Logging & Monitoring
- Container lifecycle is short and unpredictable
- Visibility through telemetry and logs
- Tag and label assets for context and de-duplication
- Focus on visibility at these levels
  - Application-level logging
  - Container-level logging
  - Orchestration/Scheduler logging
  - Cloud/Infrastructure logging (services and systems)

Example - Creating a K8s Audit Policy

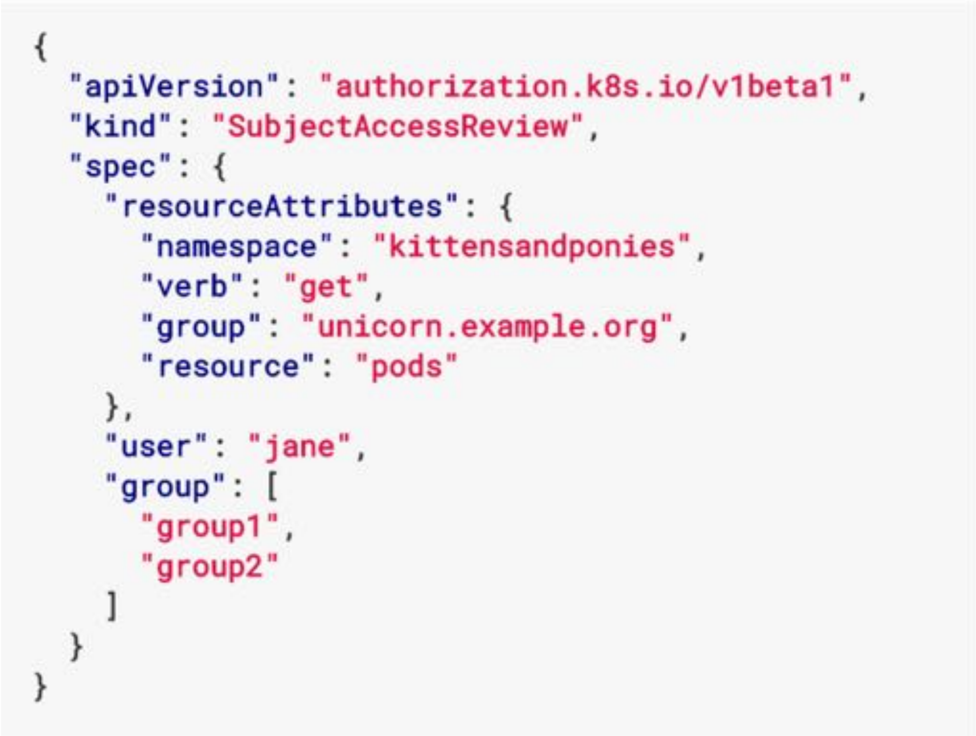
- Building an audit policy
  - API accessible via the audit.k8s.io group
  - *Metadata* – user, timestamp, verb, resources but no request or response
  - *Request* – request only
  - *RequestResponse* – request and response
  - *None* - do not log



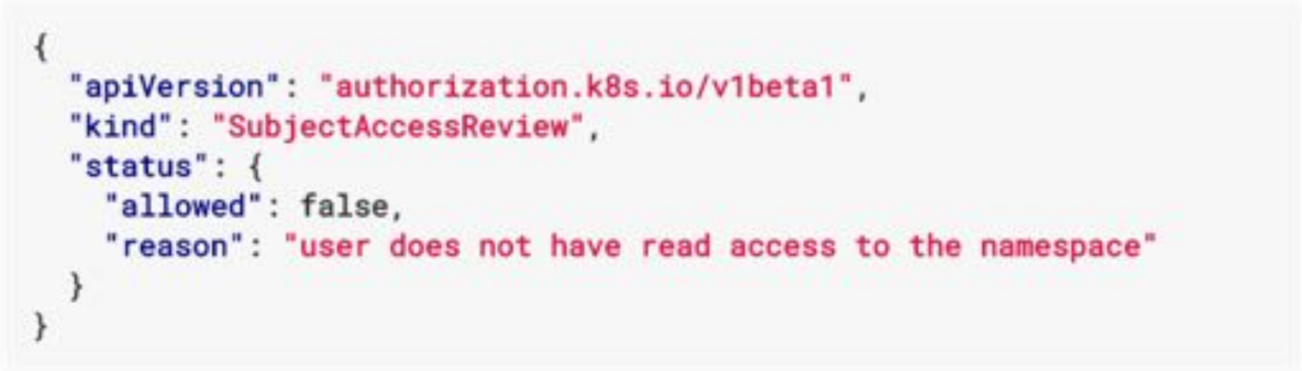
```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
  - level: RequestResponse
    resources:
    - group: ""
      resources: ["pods", "secrets", "rbac"]
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]
```

Webhooks

- Send security relevant events to a Webhook endpoint
  - **--authorization-webhook-config-file=webhook.config**



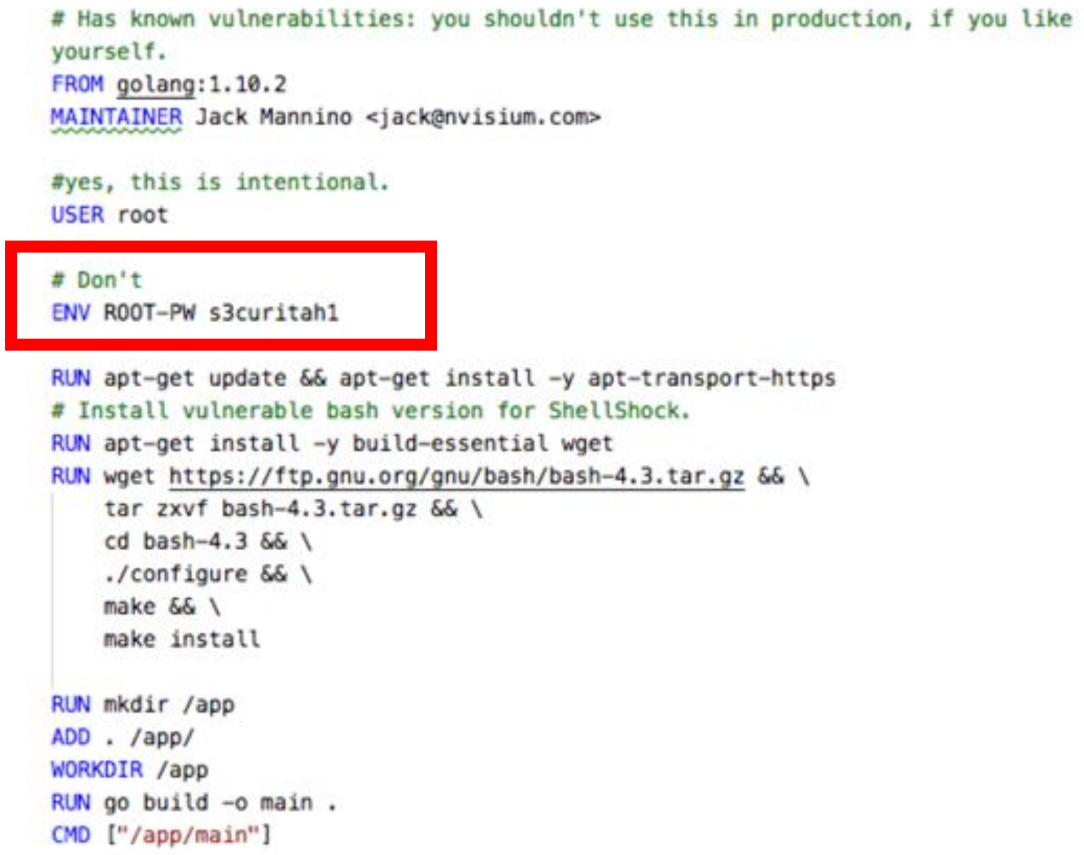
```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "kittensandponies",
      "verb": "get",
      "group": "unicorn.example.org",
      "resource": "pods"
    },
    "user": "jane",
    "group": [
      "group1",
      "group2"
    ]
  }
}
```



```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "reason": "user does not have read access to the namespace"
  }
}
```

Secrets Management

- Safely inject secrets into containers at runtime
- Reduced footprint for leaking secrets
- Dynamic key generation and rotation is ideal
- Anti-patterns:
  - Hardcoded
  - Environment variables
- Limit the scope of subjects that can retrieve secrets



```
# Has known vulnerabilities: you shouldn't use this in production, if you like
yourself.
FROM golang:1.10.2
MAINTAINER Jack Mannino <jack@nvisium.com>

#yes, this is intentional.
USER root

# Don't
ENV ROOT-PW s3curitah1

RUN apt-get update && apt-get install -y apt-transport-https
# Install vulnerable bash version for ShellShock.
RUN apt-get install -y build-essential wget
RUN wget https://ftp.gnu.org/gnu/bash/bash-4.3.tar.gz && \
    tar zxvf bash-4.3.tar.gz && \
    cd bash-4.3 && \
    ./configure && \
    make && \
    make install

RUN mkdir /app
ADD . /app/
WORKDIR /app
RUN go build -o main .
CMD ["/app/main"]
```

Secrets Management

Docker

~~docker run –it –e “DBUSER=dbuser” –e “DBPASSWD=dbpasswd” mydbimage~~

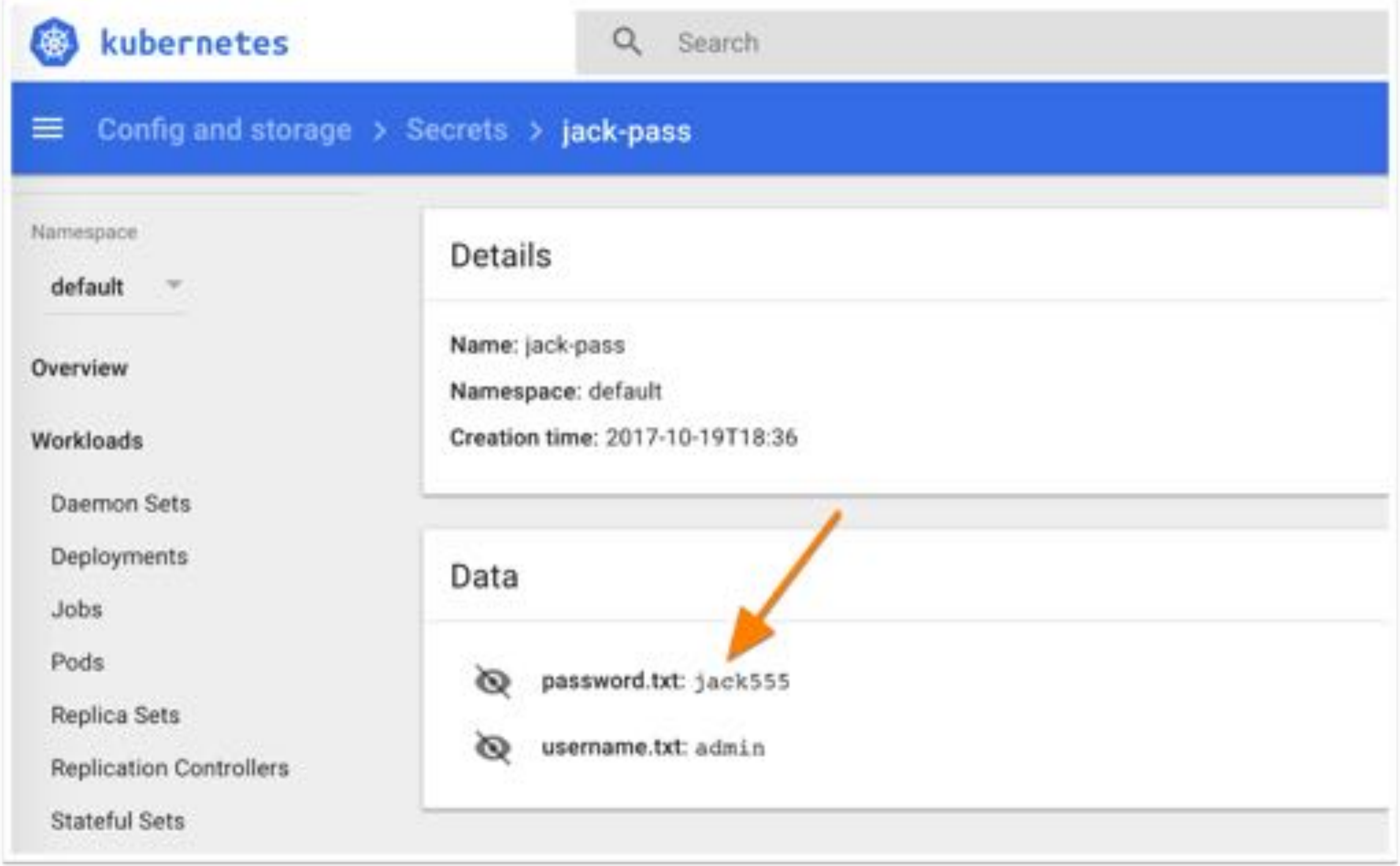
```
echo <secret> | docker secret create some-secret
```

Kubernetes

```
kubectl create secret generic db-user-pw --from-file=./username.txt --from-file=./password.txt
```

```
kubectl create –f ./secret.yaml
```

Nothing is Perfect



Details

Name: jack-pass

Namespace: default

Creation time: 2017-10-19T18:36

Data

password.txt: jack555

username.txt: admin

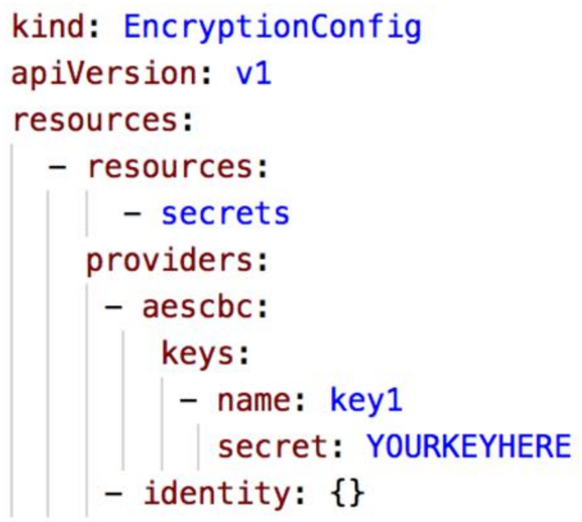
Beware of Plain Text Storage

Prior to 1.7, secrets were stored in plain text at-rest

```
$ ls /etc/foo/
username
password

$ cat /etc/foo/username

admin
$ cat /etc/foo/password
1f2d1e2e67df
```

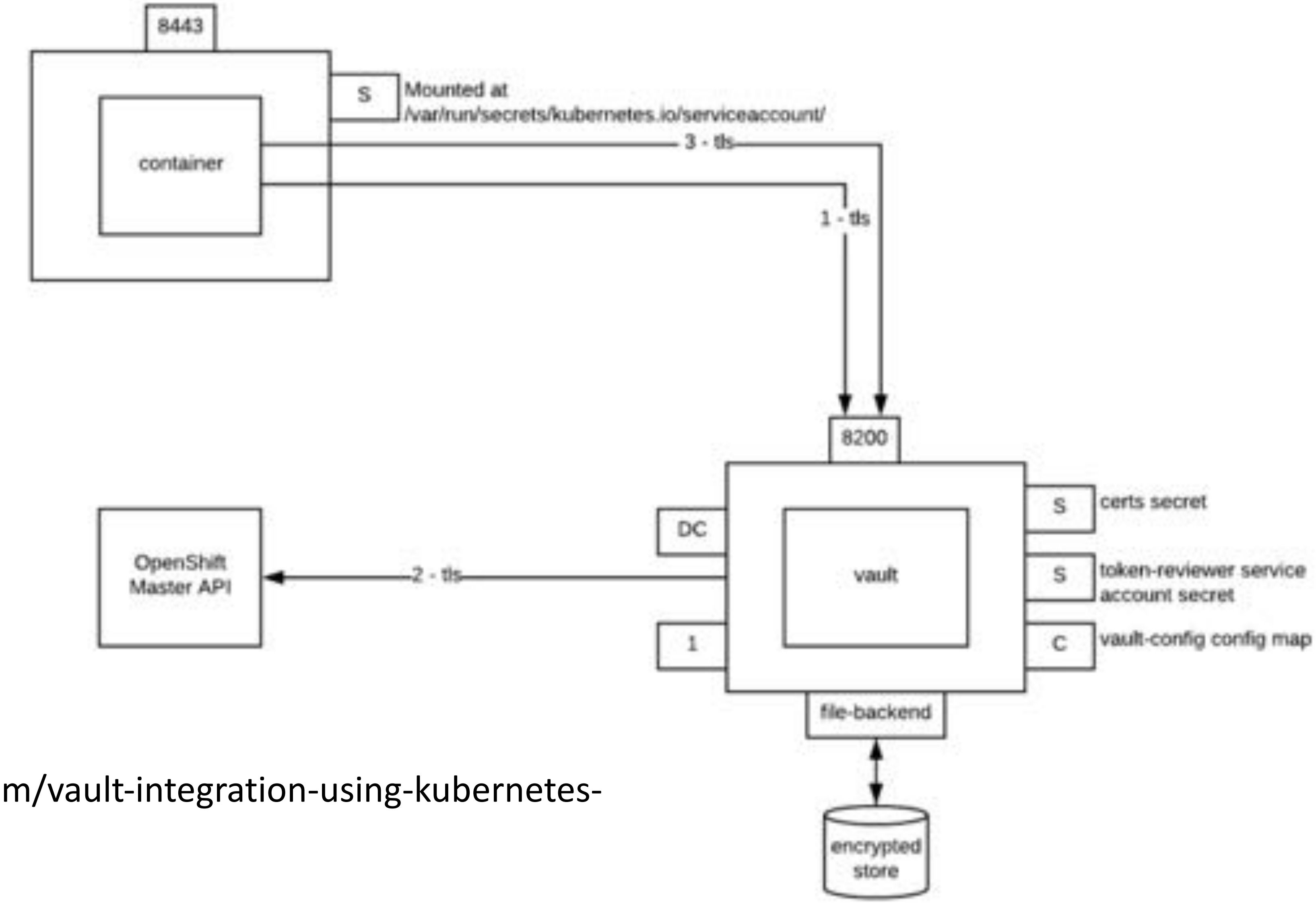
As of v1.7+, k8s can encrypt your secrets in **etcd**

Not perfect at all, either.



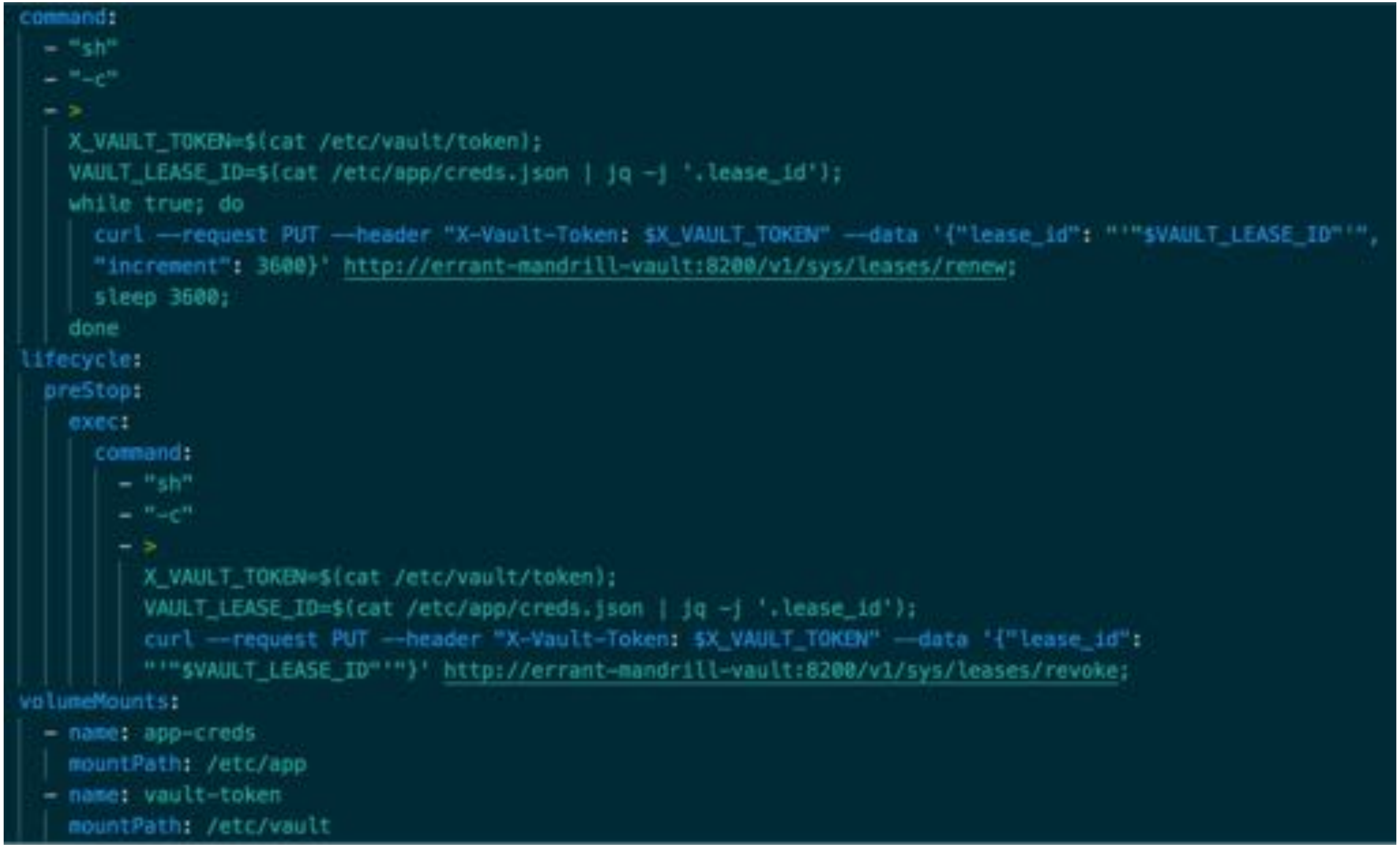
```yaml
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - aescbc:
        keys:
        - name: key1
          secret: YOURKEYHERE
    - identity: {}
```

Dynamic Loading & Rotation



https://blog.openshift.com/vault-integration-using-kubernetes-authentication-method/

Example - Retrieve and Mount a Secret



```
command:
  - "sh"
  - "-c"
  - >
    X_VAULT_TOKEN=$(cat /etc/vault/token);
    VAULT_LEASE_ID=$(cat /etc/app/creds.json | jq -j '.lease_id');
    while true; do
      curl --request PUT --header "X-Vault-Token: $X_VAULT_TOKEN" --data '{"lease_id": "'"$VAULT_LEASE_ID"'",
      "increment": 3600}' http://errant-mandrill-vault:8200/v1/sys/leases/renew;
      sleep 3600;
    done
lifecycle:
  preStop:
    exec:
      command:
        - "sh"
        - "-c"
        - >
          X_VAULT_TOKEN=$(cat /etc/vault/token);
          VAULT_LEASE_ID=$(cat /etc/app/creds.json | jq -j '.lease_id');
          curl --request PUT --header "X-Vault-Token: $X_VAULT_TOKEN" --data '{"lease_id":
          "'"$VAULT_LEASE_ID"'"}' http://errant-mandrill-vault:8200/v1/sys/leases/revoke;
volumeMounts:
  - name: app-creds
    mountPath: /etc/app
  - name: vault-token
    mountPath: /etc/vault
```

Policy & Constraint Enforcement

- Harden by applying a Security Context at the pod or container level
- Mutate the container's configuration as needed
  - i.e- overrides a Dockerfile

Setting	PodSecurityContext	SecurityContext
Allow Privilege Escalation		X
Capabilities		X
Privileged		X
Read-Only Root Filesystem		X
Run as Non Root	X	X
Run as User	X	X
SELinux Options	X	
FS Group	X	
Supplemental Groups	X	

Example – K8s Pod & Container Security Context

Conclusion

- Secure your container ecosystem and supply chain, not just the runtime
- You probably don't need root – start with minimally privileged containers
- Focus on layered security and strong isolation
- Ensure visibility from a developer's laptop to running in production

Thanks! Keep in Touch

Jack Mannino

Twitter @jack_mannino

Linkedin - https://www.linkedin.com/in/jackmannino

Email - Jack@nvisium.com