Security Best Practices for Kubernetes Deployment

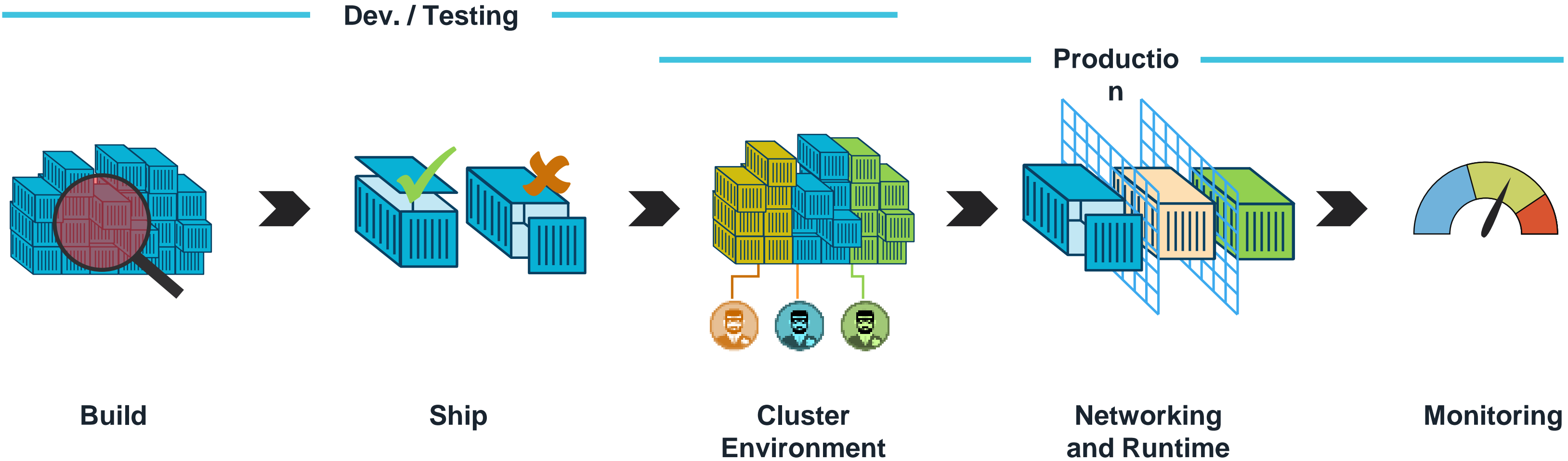
Michael Cherny
Head of Research, Aqua Security

Copyright @ 2017 Aqua Security Software Ltd. All Rights Reserved.

WHO AM I

- Head of Security Research at Aqua Security, a lead in container security
- 20 years of building security products, developmen and research
- Held senior security research positions at Microsoft, Aorato
  and Imperva.
- Presented at security conferences, among them, BlackHat Europe, RSA Europe and Virus Bulleting.

BUILD SECURITY ALIGNED WITH DEVOPS

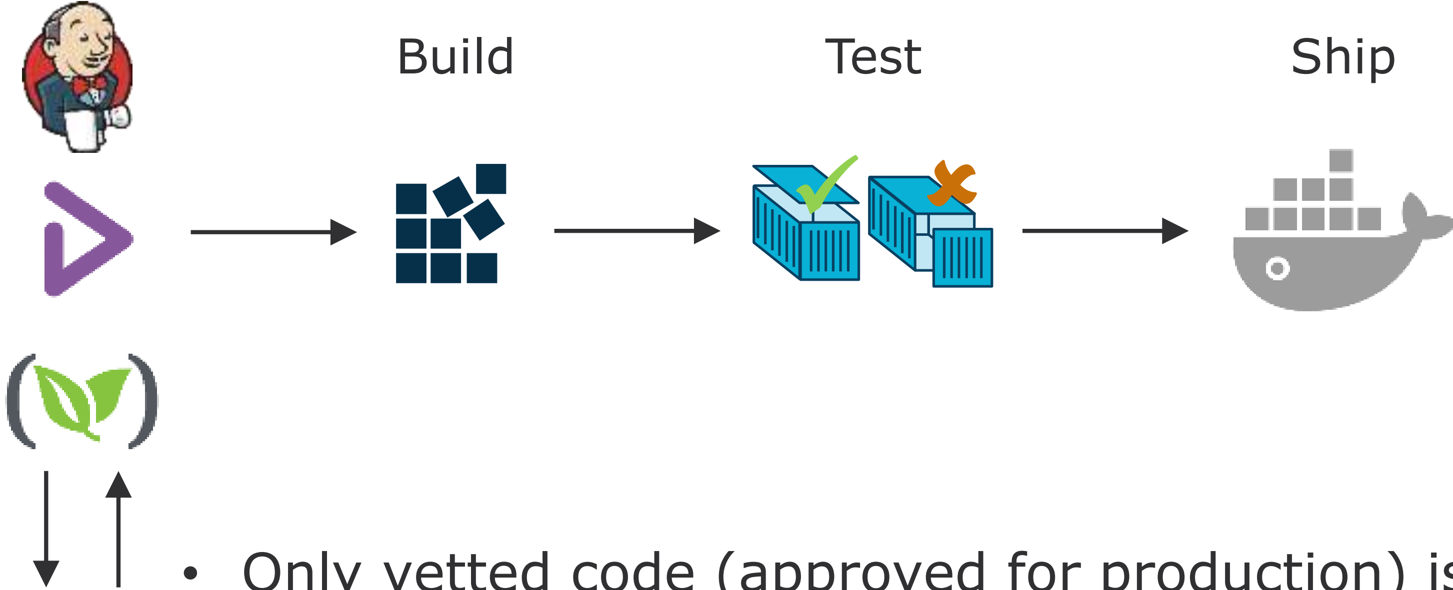


BUILD AND SHIP

BUILD AND SHIP

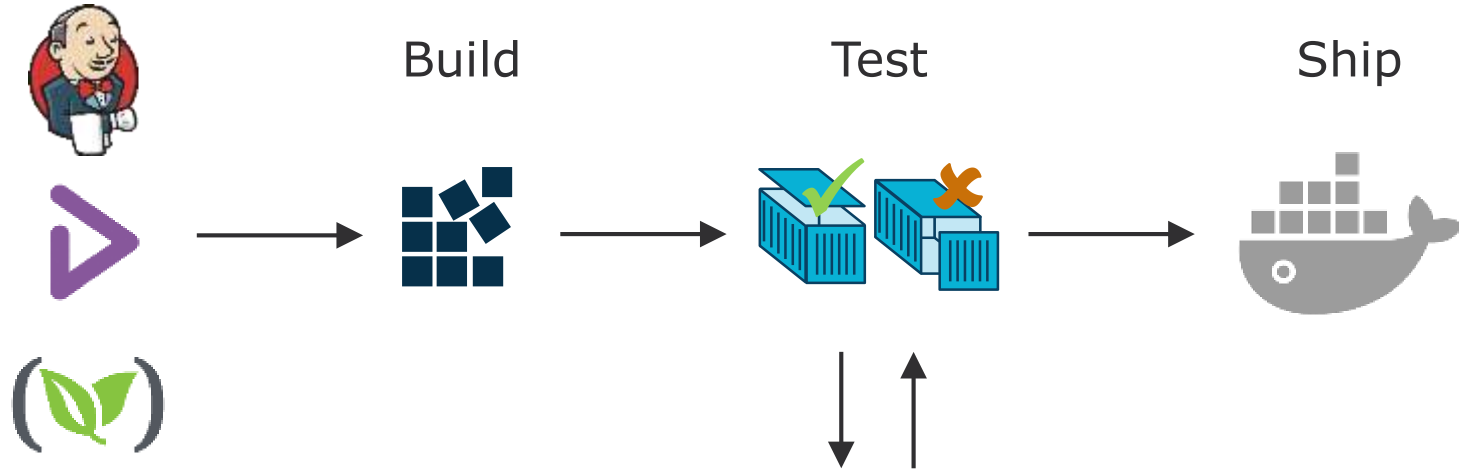
- Ensure That Only Authorized Images are Used in Your Environment
- Ensure That Images Are Free of Vulnerabilities
- Integrate Security into your CI/CD pipeline

SECURITY INTEGRATED WITH CI/CD



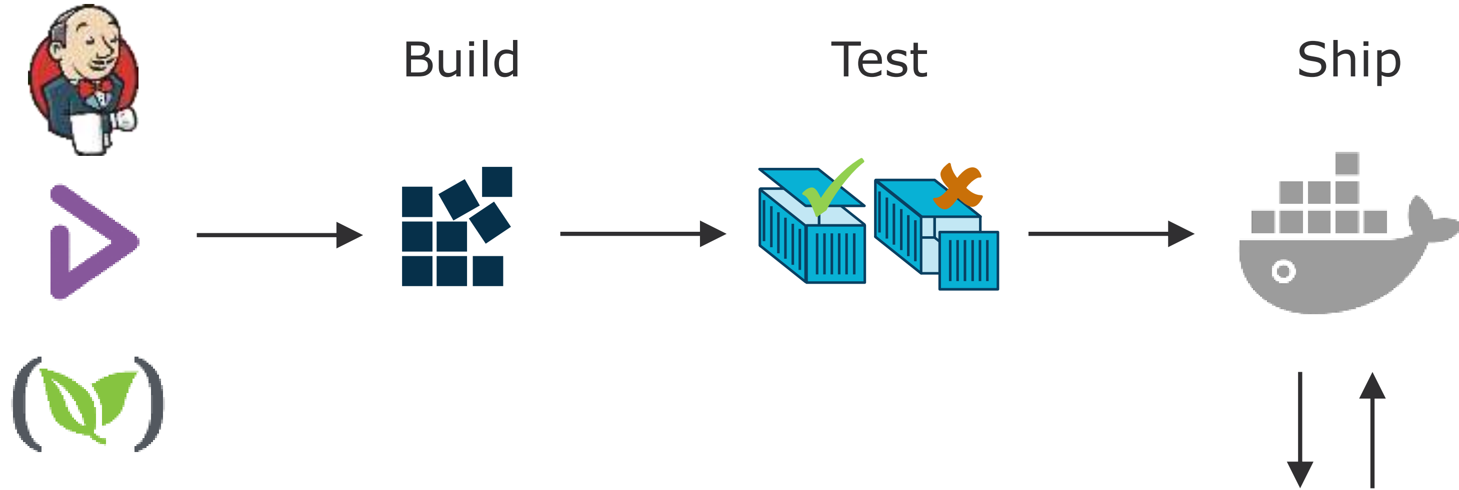
- Only vetted code (approved for production) is used for building the images

SECURITY INTEGRATED WITH CI/CD



- Implement Continuous Security Vulnerability Scanning
- Regularly Apply Security Updates to Your Environment

SECURITY INTEGRATED WITH CI/CD



- Use private registries to store your approved images - make sure you only push approved images to these registries

CLUSTER ENVIRONMENT

LIMIT DIRECT ACCESS TO KUBERNETES NODES

- Limit SSH access to Kubernetes nodes
- Ask users to use “kubectl exec”

CONSIDER KUBERNETES AUTHORIZATION PLUGINS

- Enables define fine-grained-access control rules
  - Namespaces
  - Containers
  - Operations
- ABAC mode
- RBAC mode
- Webhook mode
- Custom mode

CREATE ADMINISTRATIVE BOUNDARIES BETWEEN RESOURCES

- Limits the damage of mistake or malicious intent
- Partition resources into logical groups
- Use Kubernetes namespaces to facilitate resource segregation
- Kubernetes Authorization plugins to segregate user's access to namespace resources

CREATE ADMINISTRATIVE BOUNDARIES BETWEEN RESOURCES

- Example: allow ‘alice’ to read pods from namespace ‘fronto’

```
{
	"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
	"kind": "Policy",
	"spec": {
		"user": "alice",
		"namespace": "fronto",
		"resource": "pods",
		"readonly": true
	}
}
```

aqua

DEFINE RESOURCE QUOTA

- Resource-unbound containers in shared cluster are bad practice
- Create resource quota policies
  - Pods
  - CPUs
  - Memory
  - Etc.
- Assigned to namespace

DEFINE RESOURCE QUOTA EXAMPLE

- computer-resource.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
    name: compute-resources
spec:
    hard:
        pods: "4‟
        requests.cpu: "1‟
        requests.memory: 1Gi
        limits.cpu: "2‟
        limits.memory: 2Gi
```

- kubectl create -f ./compute-resources.yaml --namespace=myspace

aqua

SAFELY DISTRIBUTE YOUR SECRETS

- Storing sensitive data in docker file, POD definition etc. is not safe
- Centrally and securely stored secrets
  - Manage user access
  - Manage containers/pods access
  - Store securely
  - Facilitate secrets expiry and rotation
  - Etc.

KUBERNETES SECRETS

- Secret object
  - As file
  - As Environment variable
- Risks
  - Secrets stored in plain text
  - Is at rest
  - No separation of duties: operator can see secret value
  - Secrets are available, even if there is no container using it

KUBERNETES SECRETS - EXAMPLE

- echo -n "admin" > ./username.txt
  echo -n "1f2d1e2e67df" > ./password.txt
- kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt
  secret "db-user-pass" created
- kubectl get secrets
- Kubectl describe secrets/db-user-pass

aqua

KUBERNETES SECRETS - EXAMPLE

- It call also be done manually
- echo -n "admin" | base64
  YWRtaW4=
- echo -n "1f2d1e2e67df" | base64
  MWYyZDFlMmU2N2Rm

```
apiVersion: v1
kind: Secret
metadata:
	name: mysecret
type: Opaque
data:
	username: YWRtaW4=
	password: MWYyZDFlMmU2N2Rm
```

- kubectl create -f ./secret.yaml

NETWORKING

IMPLEMENT NETWORK SEGMENTATION

- “Good neighbor” compromised application is a door open into cluster
- Network segmentation
- Ensures that container can communicate only with whom it must

IMPLEMENT NETWORK SEGMENTATION

- Cross-cluster segmentation can be achieved using firewall rules
- “dynamic” nature of container network identities makes container network segmentation a true challenge
- Kubernetes Network SIG works on pod-to-pod network policies
  - Currently only ingress policies can be defined

IMPLEMENT NETWORK SEGMENTATION: EXAMPLE

- Enable using an annotation on the Namespace

```
kind: Namespace
apiVersion: v1
metadata:
    annotations:
        net.beta.kubernetes.io/network-policy: |
        {
            "ingress": {
                "isolation": "DefaultDeny"
            }
        }
```

```
kubectl annotate ns <namespace> "net.beta.kubernetes.io/network-policy={\"ingress\": {\"isolation\": \"DefaultDeny\"}}"
```

IMPLEMENT NETWORK SEGMENTATION: EXAMPLE

```
apiVersion: extensions/v1beta1
kind: NetworkPolicy
metadata:
      name: test-network-policy
      namespace: default
spec:
      podSelector:
            matchLabels:
                  role: db

      ingress:
            - from:
                  - namespaceSelector:
                        matchLabels:
                              project: myproject
                  - podSelector:
                        matchLabels:
                        role: frontend
            ports:
                 - protocol: tcp
                  port: 6379

kubectl create -f policy.yaml
```

RUNTIME

RUNTIME

- Implement “least privileges” principal
- Security Context control security parameters to be assigned
  - Pod
  - Containers
- Pod Security Policy
- Consider Kubernetes admission controllers:
  - “DenyEscalatingExec” – for containers/pods with elevated privileges
  - “ImagePolicyWebhook” – Kubernetes support to plug external image reviewer
  - “AlwaysPullImages” – in multitenant cluster

SECURITY CONTEXT

Security Context Setting	Description
SecurityContext->runAsNonRoot	Indicates that containers should run as non-root user
SecurityContext->Capabilities	Controls the Linux capabilities assigned to the container.
SecurityContext->readOnlyRootFilesystem	Controls whether a container will be able to write into the root filesystem.
PodSecurityContext->runAsNonRoot	Prevents running a container with ‘root’ user as part of the pod

SECURITY CONTEXT: EXAMPLE

```
apiVersion: v1
kind: Pod
metadata:
        name: hello-world
spec:
        containers:
# specification of the pod's containers
# ...
securityContext:
        readOnlyRootFilesystem: true
        runAsNonRoot: true
```

IMAGE POLICY WEBHOOK

- api.imagepolicy.v1alpha1.ImageReview object describing the action
- Fields describing the containers being admitted, as well as any pod annotations that match *.image-policy.k8s.io/*

IMAGE POLICY WEBHOOK: EXAMPLE

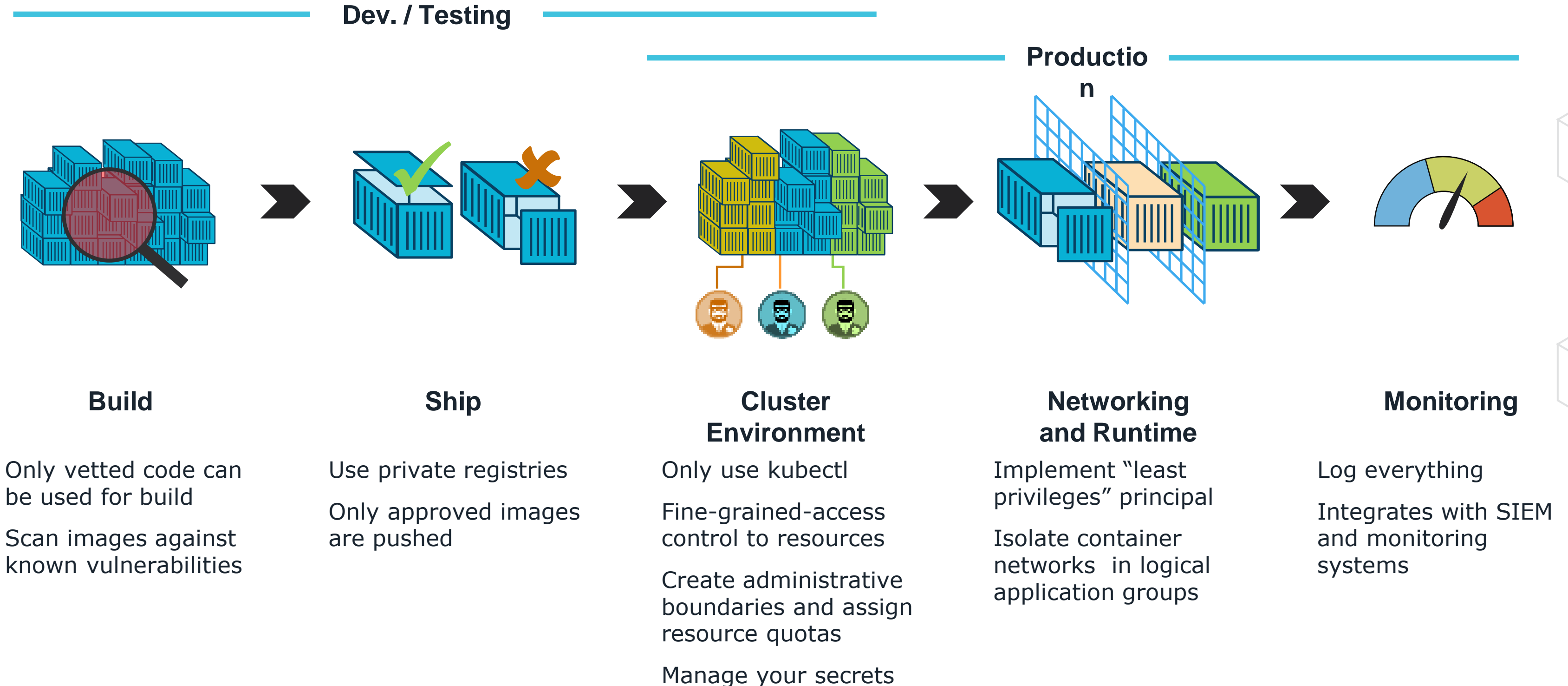
```
{
  "apiVersion":"imagepolicy.k8s.io/v1alpha1",
  "kind":"ImageReview",
  "spec":{
    "containers":[
     {
       "image":"myrepo/myimage:v1“
     },
    {

"image":"myrepo/myimage@sha256:beb6bd6a68f114c1dc2ea4b28db81bdf91de202a9014972bec5e4d9171d
90ed“
    }
   ],
   "annotations":[
     "mycluster.image-policy.k8s.io/ticket-1234": "break-glass“
   ],
   "namespace":"mynamespace“
  }
}
```

MONITORING AND VISIBILITY

- Log everything
- Cluster-based logging
  - Log container activity into a central log hub.
  - Use Fluentd agent on each node
  - Ingested logs using
    - Google Stackdriver Logging
    - Elasticsearch
  - Viewed with Kibana.

SUMMARY

SECURITY ALIGNED WITH DEVOPS



THANK YOU

Michael Cherny
cherny@aquasec.com
@chernymi
https://www.aquasec.com