Attacking and Defending K8s Clusters

Nishaanth Guna, MDSec

>$ whoami

- Currently working as a Security Consultant with MDSec, UK focusing on cloud/infrastructure/application security
- 7+ years of penetration testing and security consulting experience
- Previous work experience include Ernst & Young, Mumbai and a mobile security startup in Bangalore conducting mobile pentests
- Reported vulnerabilities to Apple, AT&T, Microsoft, Govt. of UK and many more vulnerabilities in enterprise & popular web/mobile applications
- Chess, photography, hikes
- Conference talks include Steelcon, PHDays, InCTF
- BSides Tirana is my first workshop!

Agenda

- 0x00 — Containers and K8s internals
- 0x01 — Initial access and Attack paths
- 0x02 — War stories – Common misconfigurations
- 0x03 — Securing production-grade clusters
- 0x04 — VR, CVEs and future research

Docker 101

- Docker automates deployment of software applications inside containers by providing additional layer of abstraction and automation using OS virtualization
- Docker containers != VM
- Containers run as a process in the host operating system segregated by a namespace. Each container gets a different ns.
- Containers provide similar level of isolation at a fraction of computing power.
- Containers run based on the image which you provide. These images can be hosted on public or private repositories (docker.io, ECR and so on)
- Companies managing thousands of docker containers need to orchestrate, maintain all of them at once which is where K8s come in picture.

Demo

Objective

- Get familiar with container platform (Docker)
- Run basic commands such as listing containers, executing into the containers and so on
- Understand why there is a need to orchestrate containers

Commands

- docker run nginx -dit
- docker ps && docker ps –a
- docker exec –it <identifier> sh
- docker container prune

Demo

Objective

- Learn more about docker sockets and misconfigurations
- Under the hood of executing commands inside a docker container and seeing what it reflects in the host OS
- Remap the namespace to use a low-privileged user

Commands

- sudo systemctl status docker && ps aux |grep –I dock
- docker exec -> run sleep -> check the host OS
- cat /etc/subuid && cat /etc/subgid
- sudo systemctl stop docker
- sudo dockerd --userns-remap="dubov.daniil:dubov.daniil"

Kuberwhat??!!

- k(j)uːbərˈnɛtɪs, -ˈneɪtɪs, -ˈneɪtiːz, -ˈnɛtiːz
- koo-ber-net-ees
- K8s (8 letters between K and s)
- κυβερνήτης / kubernḗtēs: Greek for "steersman, navigator" or "guide“
- Also the etymological root of cybernetics

- Originally designed by Google (Project Borg)
- Amongst other tech, this is managed by CNCF
- Open-sourced in 2014
- What is it though?


IT CAME TO THEM WITH A MESSAGE.
BUT THEY COULD NOT UNDERSTAND ITS ALIEN LANGUAGE
Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.
F*CKING WHAT?
poorlydrawnlines.com

Demo

Objective

- Setup environmental variables
- Create a multi-node cluster
- Get information about the cluster

Commands

- NAME=tirbsid.local
- kind create cluster --name $NAME --config kind-cluster-config.yaml
- less /home/$USER/.kube/config

Why is it sought-after recently?

- Move away from traditional monolith architecture
- K8s uses microservice architecture which results in reduced (almost zero) downtime if used efficiently
- Application Server crashes all of a sudden? K8s got you!
- Easy to orchestrate a large number of containers
- Relies on desired state principle

Desired state principle

- Any created resource/object will run the exact number of times specified, at any given point of time
- If a container crashes, kube-controller-manager detects this immediately
- When a crash happens and a container goes down, there would be a mismatch between desired state and actual state declared
- A new container will be launched to obtain desired state

Demo

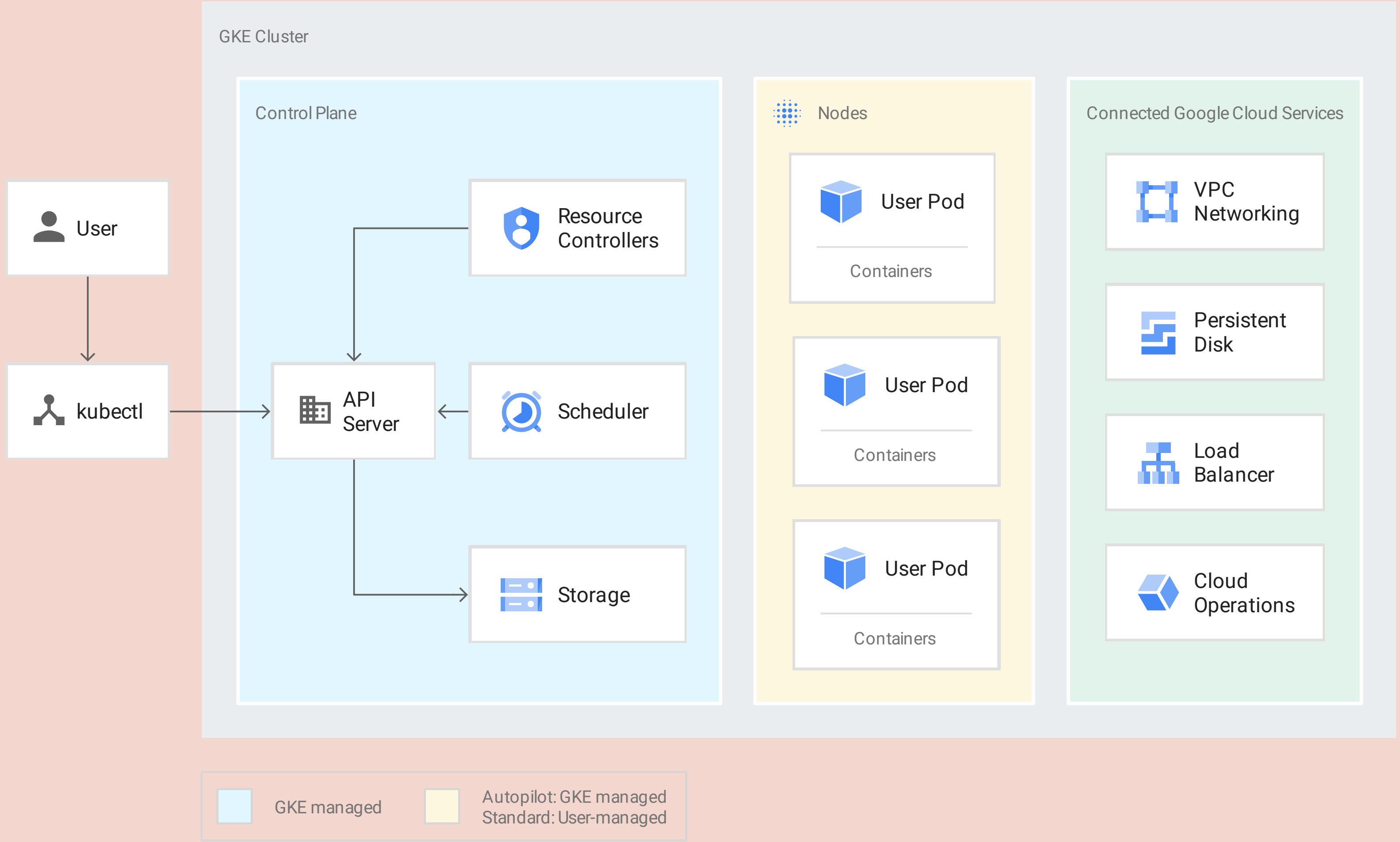
Objective

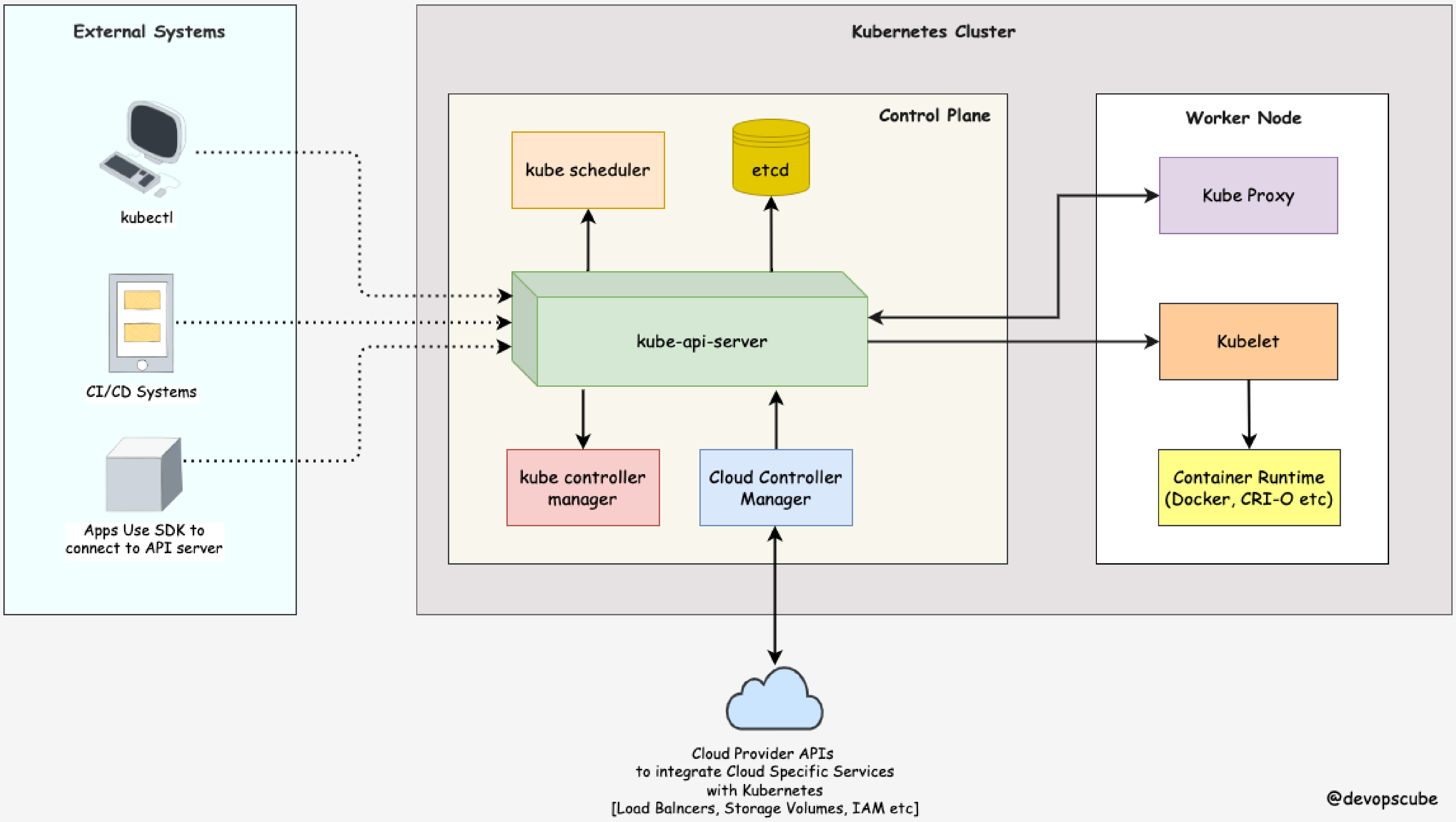
- Understand desired state principle
- Run nginx container as a ReplicaSet
- Crash one of the pods intentionally and see it in effect

Commands

- kubectl apply –f nginx-replica.yaml
- kubectl delete pod name
- kubectl get events | less

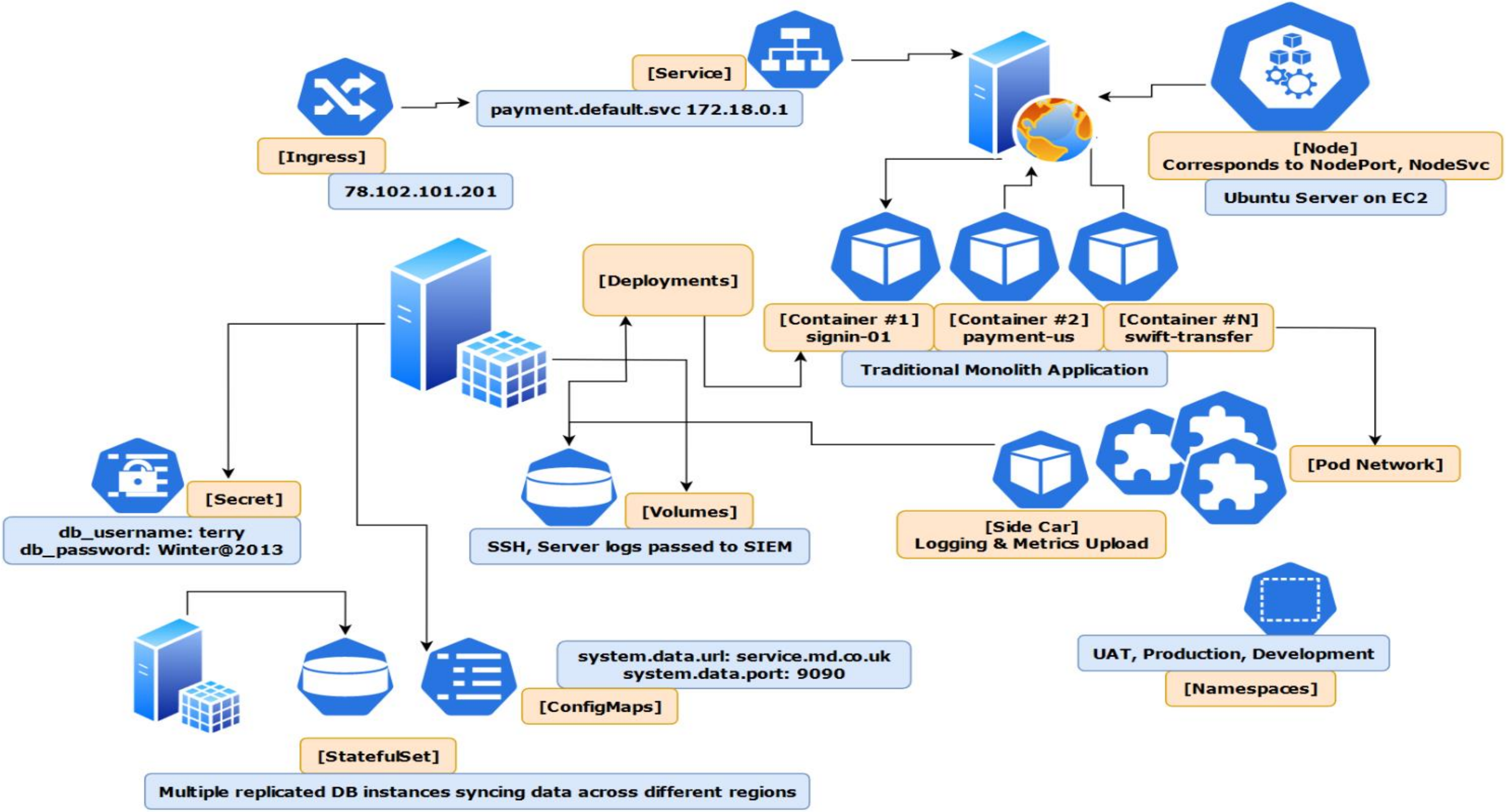

Me trying to delete a pod
❯ kubectl delete pod my-app-6b97-2j
pod
pod
pod
pod
pod
pod
pod
replicas: 12
deploy
pod
pod
pod


GKE Cluster
Control Plane
Nodes
Connected Google Cloud Services
User
kubectl
API Server
Resource Controllers
Scheduler
Storage
User Pod
Containers
User Pod
Containers
User Pod
Containers
VPC Networking
Persistent Disk
Load Balancer
Cloud Operations
GKE managed
Autopilot: GKE managed
Standard: User-managed


External Systems
kubectl
CI/CD Systems
Apps Use SDK to connect to API server
Kubernetes Cluster
Control Plane
kube scheduler
etcd
kube-api-server
kube controller manager
Cloud Controller Manager
Worker Node
Kube Proxy
Kubelet
Container Runtime (Docker, CRI-O etc)
Cloud Provider APIs
to integrate Cloud Specific Services
with Kubernetes
[Load Balncers, Storage Volumes, IAM etc]
@devopscube

```
$ kubectl api-resources --sort-by name -o wide
NAME                              SHORTNAMES   APIVERSION
NAMESPACED   KIND                             VERBS
CATEGORIES
apiservices                                    apiregistration.k8s.io/v1            false
APIService                       create,delete,deletecollection,get,list,patch,update,watch
api-extensions
assign                                         mutations.gatekeeper.sh/v1           false
Assign                           delete,deletecollection,get,list,patch,create,update,watch
assignimage                                    mutations.gatekeeper.sh/v1alpha1     false
AssignImage                      delete,deletecollection,get,list,patch,create,update,watch
assignmetadata                                 mutations.gatekeeper.sh/v1           false
AssignMetadata                   delete,deletecollection,get,list,patch,create,update,watch
bindings                                       v1                                   true
Binding                          create

certificatesigningrequests        csr          certificates.k8s.io/v1               false
CertificateSigningRequest        create,delete,deletecollection,get,list,patch,update,watch
clusterrolebindings                            rbac.authorization.k8s.io/v1         false
……
```



[Service]
payment.default.svc 172.18.0.1
[Ingress]
78.102.101.201
[Node]
Corresponds to NodePort, NodeSvc
Ubuntu Server on EC2
[Deployments]
[Container #1]
signin-01
[Container #2]
payment-us
[Container #N]
swift-transfer
Traditional Monolith Application
[Pod Network]
[Secret]
db_username: terry
db_password: Winter@2013
[Volumes]
SSH, Server logs passed to SIEM
[Side Car]
Logging & Metrics Upload
system.data.url: service.md.co.uk
system.data.port: 9090
[ConfigMaps]
UAT, Production, Development
[Namespaces]
[StatefulSet]
Multiple replicated DB instances syncing data across different regions

Demo

Objective

- Create a new namespace to be used for the workshop
- Set aliases for context switching to cluster-admin

Commands

- kubectl create ns Tirana
- alias switch-admin="kubectl config use-context cluster-admin"

Authentication into the Cluster

- Users are of 2 types in K8s – SAs which are managed by K8s and users
- Normal users cannot be added to the cluster through an API call
- Any user that presents a valid certificate signed by the cluster's CA is considered authenticated
- From here, RBAC sub-system kicks in and validates if the authenticated user has privileges on the resource and the namespace
- SAs are users managed by the Kubernetes API, created automatically by API server and bound to a namespace.
- SAs are tied to set of credentials stored as *Secrets,* which are then mounted into pods allowing in-cluster processes to talk to the API Server
- API requests are tied to normal user or a service account or anonymous
- K8s uses client certificates, bearer tokens, or an authenticating proxy to authenticate API requests through plugins

Authentication into the Cluster

- Auth strategies include X509 certs, static token file, bearer token, bootstrap tokens, SA tokens
- External identity provider can also be used with OpenID connect tokens. This can be integrated with Kerberos, AAD, Salesforce and Google.
- OIDC authentication is rigid way of authenticating into production clusters since the user has to be in the AD, cloud providers (and possible with OTP)
- Webhook token authentication, authenticating proxy can also be used
- Anonymous requests which are part of *system:anonymous* or *system:unauthenticated*
- Flag --anonymous-auth has set to be set true in the API server
- User impersonation is possible, impersonate as a user, group or UID

Demo

Objective

- Understand the authentication process involved between K8s API Server and kubectl
- Differentiate between SA and certificate authentication

Commands

- kubectl get nodes
- kubectl get nodes -v9
- kubectl get nodes --token $SA

RBAC Authorization

- Role-Based Authorization (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users.
- RBAC uses *rbac.authorization.k8s.io* API group
- Role and ClusterRole
- RoleBinding and ClusterRoleBinding
- Decide on the namespace of the resource -> Create a Role or ClusterRole -> Bind it a subject -> Validate if RBAC is applied
- Some of the default ClusterRoles include system:basic-user, system:discovery, system:public-info-viewer, cluster-admin, admin, edit, view
- Some ClusterRoles are not system: prefixed. These are user-facing roles. For example, cluster-admin

Roles and ClusterRole

- An RBAC *Role* or *ClusterRole* contains rules that represent a set of permissions. Permissions are additive (there are no "deny" rules)
- A *Role* sets permissions within the mentioned namespace; whenever you create a role, the namespace has to be mentioned
- *ClusterRole* is a non-namespaced resource. Set of permissions are applied across the cluster
- If you want to define a role within a namespace, use a *Role;* if you want to define a role cluster-wide, use a *ClusterRole*
- ClusterRole can be used to grant permissions on cluster-scoped resources like nodes, healthz endpoints)
- Should follow path segment names rule. In other words, the name cannot contain "." or ".." or "/" or "%"

RoleBinding and ClusterRoleBinding

- A *RoleBinding* grants permissions defined in a *Role*. It has list of subjects (users, groups or service accounts), and a reference to the role being granted
- A *RoleBinding* grants permissions within a specific namespace whereas a *ClusterRoleBinding* grants it cluster-wide
- A *RoleBinding* may reference any *Role* in the same namespace
- Alternatively, a *RoleBinding* can reference a ClusterRole and bind that *ClusterRole* to the namespace of the *RoleBinding*
- If you want to bind a *ClusterRole* to all the namespaces in your cluster, you can use a *ClusterRoleBinding*
- This kind of reference lets you define a set of common roles across the cluster, then reuse them in other namespacess

Demo

Objective

- Create a new user in the Kubernetes environment
- Create a new role to allow access to a specific namespace
- Create a new rolebinding to bind the created role to the user

Commands

- openssl genrsa -out dubov.key 2048
- openssl req -new -key dubov.key -out dubov.csr –subj "/CN=dubov"
- docker cp container-name:/etc/kubernetes/pki/ca.key .
- docker cp fc25868e5b43:/etc/kubernetes/pki/ca.crt .

Demo

Commands

- openssl x509 -req -in dubov.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out dubov.crt -days 300
- kubectl config set-credentials dubov --client-certificate=dubov.crt --client-key=dubov.key
- kubectl config set-context dubov --cluster=kind-bsides --user=dubov --server=https://127.0.0.1:35605

Commands

- kubectl config use-context dubov
- kubectl get pods & kubectl auth can-i --list
- kubectl apply -f rbac-roles.yaml
- kubectl apply -f rbac-rolebinding.yaml

Demo

Commands

- Dump necessary files for roles, cluster role and corresponding bindings
- Identify risky RBAC permissions using rbac-audit
- Identify attack paths (plugins, resources) using rbac-police

Commands

- kubectl get roles -A -o json > roles.json
- kubectl get clusterroles -A -o json > clusterroles.json
- kubectl get rolebindings -A -o json > rolebindings.json
- kubectl get clusterrolebindings -A -o json > clusterrolebindings.json

```
─<%>─ python3 ExtensiveRoleCheck.py --clusterRole /tmp/clusterroles.json --role /tmp/roles.json --cluseterolebin
dings /tmp/clusterrolebindings.json --rolebindings /tmp/rolebindings.json

[*] Started enumerating risky ClusterRoles:
[!][ClusterRole]→ gatekeeper-manager-role Has permission to use list on any resource!
[!][ClusterRole]→ gatekeeper-manager-role Has permission to use delete on any resource!
[!][ClusterRole]→ gatekeeper-manager-role Has permission to use delete on any resource!
[!][ClusterRole]→ gatekeeper-manager-role Has permission to use delete on any resource!
[!][ClusterRole]→ gatekeeper-manager-role Has permission to use delete on any resource!
[!][ClusterRole]→ get-secrets-everywhere Has permission to list secrets!
[!][ClusterRole]→ local-path-provisioner-role Has permission to access pods with any verb!
[*] Started enumerating risky Roles:
[!][Role]→ create-pod-ns Has permission to create pods!
[!][Role]→ gatekeeper-manager-role Has permission to list secrets!
[!][Role]→ create-pod-ns Has permission to create pods!
[*] Started enumerating risky ClusterRoleBinding:
[!][ClusterRoleBinding]→ gatekeeper-manager-rolebinding is binded to gatekeeper-admin ServiceAccount.
[!][ClusterRoleBinding]→ local-path-provisioner-bind is binded to local-path-provisioner-service-account Service
Account.
[!][ClusterRoleBinding]→ rbac-clusterrole-dubov is binded to the User: dubov!
[*] Started enumerating risky RoleBindings:
[!][RoleBinding]→ rbac-role-binding-role-binding is binded to the User: dubov!
[!][RoleBinding]→ rbac-role-dubov is binded to the User: dubov!
[!][RoleBinding]→ gatekeeper-manager-rolebinding is binded to gatekeeper-admin ServiceAccount.
─<ggwp@metaaoh:~/k8s-content/kubernetes-rbac-audit [master]>─                                          ─<pts/0>─
```

```
─<ggwp@metaaoh:~/k8s-content/rbac-police [main]>─                                                    ─<pts/0>─
─<%>─ rbac-police eval lib/
{
    "policyResults": [
        {
            "policy": "lib/assign_sa.rego",
            "severity": "Critical",
            "description": "Identities that can create pods or create, update or patch pod controllers (e.g. Dae
monSets, Deployments, Jobs) in privileged namespaces (kube-system), may assign an admin-equivalent SA to a pod i
n their control",
            "violations": {
                "serviceAccounts": [
                    {
                        "name": "local-path-provisioner-service-account",
                        "namespace": "local-path-storage",
                        "nodes": [
                            {
                                "bsides-control-plane": [
                                    "local-path-provisioner-6bc4bddd6b-xr4ks"
                                ]
                            }
                        ]
                    }
                ]
            }
```

```
        {
            "policy": "lib/list_secrets.rego",
            "severity": "Medium",
            "description": "Identities that can list secrets cluster-wide may access confidential information, a
nd in some cases serviceAccount tokens",
            "violations": {
                "serviceAccounts": [
                    {
                        "name": "gatekeeper-admin",
                        "namespace": "gatekeeper-system",
                        "nodes": [
                            {
                                "bsides-control-plane": [
                                    "gatekeeper-audit-6668847c5c-549wz",
                                    "gatekeeper-controller-manager-7fff77f764-c2bm7",
                                    "gatekeeper-controller-manager-7fff77f764-nkmml",
                                    "gatekeeper-controller-manager-7fff77f764-qv2sl"
                                ]
                            }
                        ]
                    }
                ]
            }
        },
```

K8s Vulns Overview

0x00

Deploying resources in default namespace

0x01

Bad Pods – Container to Host Compromise

0x02

Malicious Container Images (Vulnerable CI/CD, Jenkins)

0x03

Insecure RBAC (*.* on resources, verbs)

0x04

Overly-permissive and powerful third-party plugins

Initial Access

- Exposed cloud credentials – Compromised cloud credentials used by exposed AKS/GKE/EKS can lead to cluster takeover if keys are scoped
- Compromised image in registry – Compromising a private registry to plant malware in the base image pulled regularly by developers
- Exposed Kubeconfig – If attackers get access to kubeconfig file via a compromised client or any others means, this can be used to access K8s.
- Application vulnerability – Running a public-facing vulnerable application in a cluster which is vulnerable to RCE, LFI can lead to compromise. An attacker can read the SA mounted into the pod, breakout to attack the host
- Exposed Interfaces – Software like K8s dashboard, Apache NiFi, Kubeflow, Argo were never meant to be exposed publicly.

Demo

Objective

- Understand why deploying resources in default namespace is insecure
- Understand how SAs integrate into the K8s environment
- Exploit privileged SA to list secrets on the cluster

Commands

- kubectl run nginx --image=nginx
- kubectl exec -it nginx – bash
- cat /var/run/secrets/kubernetes.io/serviceaccount/token
- kubectl auth can-i --list --token $SA
- kubectl get nodes

Demo

Objective

- Understand why running pods with host mounts can be dangerous in an environment

Commands

- docker exec -it id bash
- kubectl run nginx --image=nginx
- kubectl apply –f kryptopod.yaml
- kubectl exec -it hot-pod -- bash

Privilege Escalation – Attack Paths

Scenario – 1

{-} Compromised credentials has untethered permissions on the entire cluster
{-} Get the list of nodes running in the environment to differentiate between master and worker nodes
{-} Create a pod with all the sensitive host mounts, capabilities and PrivEsc
{-} Using nodeName value, deploy the pod in one of the master nodes
{-} From inside the privileged pod, chroot to the host filesystem and search through the files and folders in the system
{-} Environmental variable contains a Service Account token and location to kubeconfig file
{-} Steal them to get cluster-admin
{-} Pwn!

Privilege Escalation – Attack Paths

Scenario – 2

{-} Cordon preventing unauthorized pods in the master nodes
{-} Similar to the previous environment, create a pod on the worker node instead of the master node
{-} From the container, chroot inside the worker node and enumerate all the pods/namespaces running using ps –ef
{-} Write a script to enter all the namespaces and save the SA token
{-} One of the pods running database manager had super privileges due to the nature of the pod. Was one of the centralized key-value managers
{-} Use the database manager pod's SA token to obtain cluster-admin
{-} Pwn!

Privilege Escalation – Attack Paths

Scenario – 3

{-} Internal K8s assessment with OIDC login to the cluster
{-} Production environment completely locked out rendering the stolen keys completely useless.
{-} Key only has GET, LIST permissions on a few resources
{-} Take a step back and view the kubeconfig file once again
{-} Notice multiple contexts in the configuration file. One of the context points to staging cluster
{-} Re-use the keys in the staging cluster to discover the same keys have extra privileges including CREATE pods on lot of sensitive namespaces
{-} Create a pod with hostPath flag to steal the cluster-admin credentials
{-} Pwn!

Privilege Escalation – Attack Paths

Scenario – 4

{-} Initial access through Local File Read
{-} Compromised pod has SA with the ability to CREATE pods in a specific ns
{-} Pods cannot run as privileged due to PodSecurityPolicy.
Pods cannot run be deployed with hostNetwork/hostPath flags due to additional security controls in the environment to prevent breakouts.
{-} Can be bypassed by creating a pod with hostPID or hostIPC flags
{-} Using hostPID, list the processes on the host machine
{-} Identify a process related to cloud environment running with AWS access and secret keys as arguments
{-} AWS keys had access to the entire cloud estate == Pwn!

Privilege Escalation – Attack Paths

Scenario – 5

{-} Initial Access through SSRF from a web application
{-} Request the AWS Instance Metadata service (169.254.169.254)
{-} Keys are privileged due to a design flaw in AWS environment. Using the obtained keys you can pull ECR images and list few S3 buckets
{-} Export the keys to the local machine and verify the identity of the key
{-} Enumerate S3 buckets of the organization and use the stolen key to fetch the objects of the bucket
{-} One of the objects in the bucket include ETCD backup of the prod cluster
{-} Extract the ETCD store to discover a trove of SA tokens and AWS keys
{-} One of the keys had cluster-admin access in the cluster
{-} Pwn!

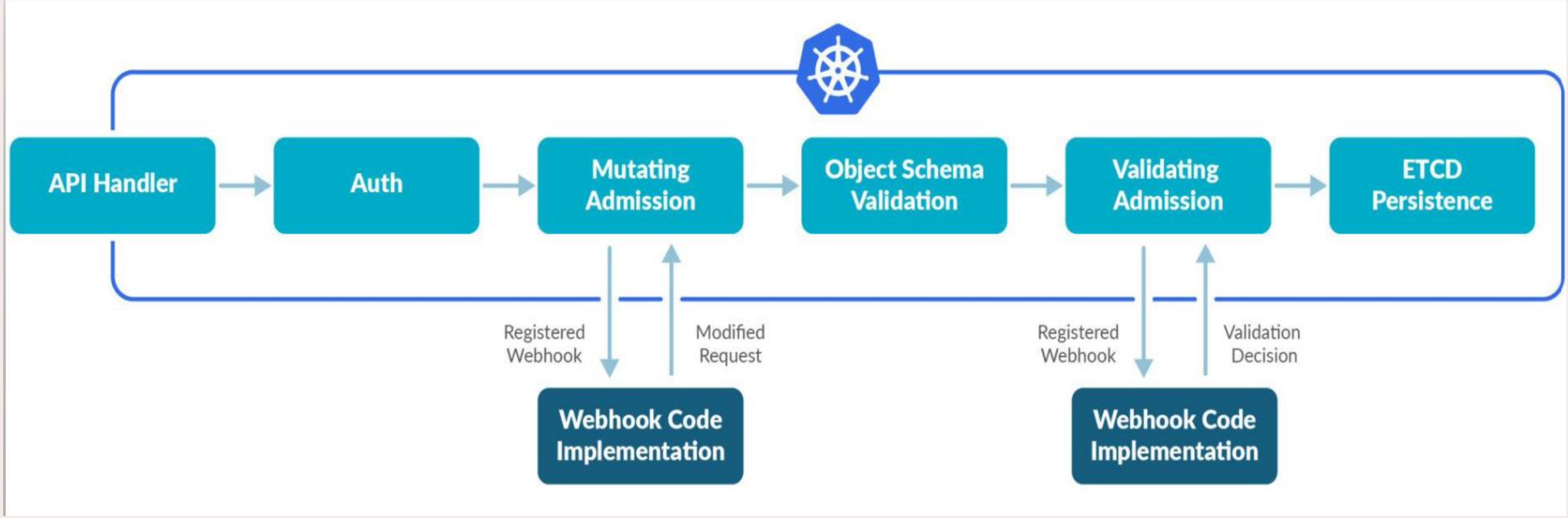
AWS EKS PrivEsc w SSRF

- Able to reach AWS Metadata API from inside a container running on a node and no default mechanism to stop that
- On a low-privileged EKS pod, the temporary AWS keys has same level of privilege as underlying EC2 machine
- Compromising a single pod using the keys from 169.254.169.254 can lead to potential compromise of AWS estate
- Block pods from reaching the AWS Metadata API
- Issue can be fixed by using IMDSv2
- Similar issues on GKE, but detailed documentation on how to prevent this vulnerability

Admission Controllers

- Gatekeeps the cluster from object writes to ETCD server if a resource does not conform with a policy (e.g., reject images not pulled from internal.mdsec.docker.registry.com, do not run containers as root or with privileged flag
- Possible to create HTTP callbacks/webhooks with custom logic to decide if a resource should be let in the cluster or not
- Mutating and Validating Controllers
- Mutating Controllers can intercept an API request and PATCH the request to manipulate a struct or an object in it)
- 30+ shipped with K8s, compiled into the kube-apiserver binary (NameSpaceLifecycle, PodSecurityPolicy, ValidatingAdmissionWebhook)
- Flask Server monitoring for webhooks -> Deploy it as a SVC -> Register a Webhook Controller -> Validate if resources conform to policy

Admission Controllers - Flow



```go
// AdmissionReview describes an admission review request/response.
type AdmissionReview struct {
	metav1.TypeMeta `json:",inline"`
	// Request describes the attributes for the admission request.
	// +optional
	Request *AdmissionRequest `json:"request,omitempty" protobuf:"bytes,1,opt,name=request"`
	// Response describes the attributes for the admission response.
	// +optional
	Response *AdmissionResponse `json:"response,omitempty" protobuf:"bytes,2,opt,name=response"`
}
```

```
type AdmissionRequest struct {
		UID types.UID `json:"uid" protobuf:"bytes,1,opt,name=uid"`
		Kind metav1.GroupVersionKind `json:"kind" protobuf:"bytes,2,opt,name=kind"`
		Resource metav1.GroupVersionResource `json:"resource" protobuf:"bytes,3,opt,name=resource"`
		SubResource string `json:"subResource,omitempty" protobuf:"bytes,4,opt,name=subResource"`
		RequestKind *metav1.GroupVersionKind `json:"requestKind,omitempty"
protobuf:"bytes,13,opt,name=requestKind"`

		RequestResource *metav1.GroupVersionResource `json:"requestResource,omitempty"
protobuf:"bytes,14,opt,name=requestResource"`
		RequestSubResource string `json:"requestSubResource,omitempty"
protobuf:"bytes,15,opt,name=requestSubResource"`

		Name string `json:"name,omitempty" protobuf:"bytes,5,opt,name=name"`
		Namespace string `json:"namespace,omitempty" protobuf:"bytes,6,opt,name=namespace"`
		Operation Operation `json:"operation" protobuf:"bytes,7,opt,name=operation"`

		UserInfo authenticationv1.UserInfo `json:"userInfo" protobuf:"bytes,8,opt,name=userInfo"`

		Object runtime.RawExtension `json:"object,omitempty" protobuf:"bytes,9,opt,name=object"`

		OldObject runtime.RawExtension `json:"oldObject,omitempty" protobuf:"bytes,10,opt,name=oldObject"`

		DryRun *bool `json:"dryRun,omitempty" protobuf:"varint,11,opt,name=dryRun"`
		Options runtime.RawExtension `json:"options,omitempty" protobuf:"bytes,12,opt,name=options"`
}
```

```go
type AdmissionResponse struct {
	// UID is an identifier for the individual request/response.
	// This should be copied over from the corresponding AdmissionRequest.
	UID types.UID `json:"uid" protobuf:"bytes,1,opt,name=uid"`

	Allowed bool `json:"allowed" protobuf:"varint,2,opt,name=allowed"`

	Result *metav1.Status `json:"status,omitempty" protobuf:"bytes,3,opt,name=status"`

	Patch []byte `json:"patch,omitempty" protobuf:"bytes,4,opt,name=patch"`

	// The type of Patch. Currently we only allow "JSONPatch".
	// +optional
	PatchType *PatchType `json:"patchType,omitempty" protobuf:"bytes,5,opt,name=patchType"`

	// AuditAnnotations is an unstructured key value map set by remote admission controller (e.g. error=image-
blacklisted).
	// MutatingAdmissionWebhook and ValidatingAdmissionWebhook admission controller will prefix the keys with
	// admission webhook name (e.g. imagepolicy.example.com/error=image-blacklisted). AuditAnnotations will be
provided by

	AuditAnnotations map[string]string `json:"auditAnnotations,omitempty"
protobuf:"bytes,6,opt,name=auditAnnotations"`

	Warnings []string `json:"warnings,omitempty" protobuf:"bytes,7,rep,name=warnings"`
}
```

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com“
webhooks:
 - name: "pod-policy.example.com“
rules:
- apiGroups:   [""]
  apiVersions: ["v1"]
  operations:  ["CREATE"]
  resources:   ["pods"]
  scope:       "Namespaced“
  clientConfig:service:namespace: "example-namespace“
  name: "example-service“
  caBundle: <CA_BUNDLE>
  admissionReviewVersions: ["v1"]
  sideEffects: None
```

Demo

Objective

- Understand how Admission Controllers work
- Check what plugins are enabled in our cluster by default
- Deploy Kyverno/OPA Gatekeeper and check validation

Commands

- kubectl describe pod kube-apiserver-bsides-control-plane -n kube-system
- ls –la /etc/kubernetes/manifests
- kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/master/deploy/gatekeeper.yaml
- kubectl get ns

Demo

Commands

- kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper-library/master/library/pod-security-policy/privileged-containers/template.yaml
- kubectl apply -f priv-constraint.yaml
- kubectl run nginx-opa-test --image nginx --privileged=true

Commands

- kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper-library/master/library/general/allowedrepos/template.yaml
- kubectl apply -f imagepull-constraint.yaml
- kubectl run image-opa-test --image nginx

ETCD Best Practices

- Due to the nature of ETCD, it is a critical resource. Even a read access to the daemon is dangerous because it contains sensitive information about the secrets, state of the cluster.
- Enabling encryption at rest for ETCD secrets. Switched off by default. Enable it by using --encryption-provider-config
- Isolate the daemon behind a firewall or use network ACL to limit communication between pods and the key-value store
- Backups should be done regularly and stored in a secure location
- Audit logging and monitoring for malicious events, unauthorized entries
- Update all the components in the environment regularly and keep a look out for CVEs which affect the software in your environment.

Setup Multi-Node Cluster w Vagrant

Vagrantfile

```ruby
Vagrant.configure("2") do |config|

    config.vm.define "k8s-master" do |master|
        master.vm.box = IMAGE_NAME
        master.vm.network "private_network", ip: API_SERVER_IP
        master.vm.hostname = "#{CLUSTER_NAME}-k8s-control-plane"
        master.vm.provider :virtualbox do |vb|
            vb.name = "#{CLUSTER_NAME}-k8s-control-plane"
            vb.memory = VM_MEMORY
            vb.cpus = 2
        end
        master.vm.provision "master-common", type: "shell",
         env: {
            "API_SERVER_IP": API_SERVER_IP,
            "DNS_SERVER": DNS_SERVER,
            "KUBERNETES_VERSION": KUBERNETES_VERSION,
            "CRI_VERSION": CRI_VERSION,
            "CLUSTER_NAME": CLUSTER_NAME,
            "OS": OS
         },
         path: "init/common.sh"
```

Setup Multi-Node Cluster w Vagrant

init > $ common.sh

```
apt update -y
apt install cri-o cri-o-runc -y
systemctl daemon-reload
systemctl enable crio --now

if [ ! -d /etc/modules.load.d/ ]; then
    mkdir /etc/modules.load.d/
fi
cat <<EOF | sudo tee /etc/modules.load.d/crio.conf
overlay
br_netfilter
EOF

modprobe overlay
modprobe br_netfilter

cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
```

Setup Multi-Node Cluster w Vagrant

init > $ master.sh

```bash
#!/bin/bash
#
set -euxo pipefail

# control plane init and image pull
kubeadm config images pull
kubeadm init --apiserver-advertise-address=$API_SERVER_IP --apiserver-cert-extra-sans=
echo "[-+-+-+-+-+-+-+-+!K8S Control Panel Up!-+-+-+-+-+-+-+-+]"

# copy token script and kubeconfig from master
kubeadm token create --print-join-command | tee /vagrant/node_join.sh
cp /etc/kubernetes/admin.conf /vagrant/admin.config
echo "[-+-+-+-+-+-+-+-+!K8S Credentials copied to host OS /vagrant folder!-+-+-+-+-+-+
```

Setup Multi-Node Cluster w Vagrant

Command Prompt - vagrant up

```
0e478
    k8s-master: + echo '[-+-+-+-+-+-+-+-+!K8S Control Panel Up!-+-+-+-+-+-+-+-+]'
    k8s-master: [-+-+-+-+-+-+-+-+!K8S Control Panel Up!-+-+-+-+-+-+-+-+]
    k8s-master: + tee /vagrant/node_join.sh
    k8s-master: + kubeadm token create --print-join-command
    k8s-master: kubeadm join 192.168.13.37:6443 --token xel22w.gn1ccd2d8n9q5oxy --discovery-
810ef29a5d70491114fa7633636cfb0862d0cf50abfa97a9cde253b40e478
    k8s-master: + cp /etc/kubernetes/admin.conf /vagrant/admin.config
    k8s-master: [-+-+-+-+-+-+-+-+!K8S Credentials copied to host OS /vagrant folder!-+-+-+-+
    k8s-master: + echo '[-+-+-+-+-+-+-+-+!K8S Credentials copied to host OS /vagrant folder!
    k8s-master: + mkdir -p /home/vagrant/.kube
    k8s-master: + cp /etc/kubernetes/admin.conf /home/vagrant/.kube/config
    k8s-master: + chown -R vagrant: /home/vagrant
    k8s-master: [-+-+-+-+-+-+-+-+!K8S Credentials exported to /home/vagrant/.kube/!-+-+-+-+-
    k8s-master: + echo '[-+-+-+-+-+-+-+-+!K8S Credentials exported to /home/vagrant/.kube/!-
==> k8s-worker-1: Importing base box 'bento/ubuntu-22.04'...
==> k8s-worker-1: Matching MAC address for NAT networking...
==> k8s-worker-1: Checking if box 'bento/ubuntu-22.04' version '202303.13.0' is up to date..
==> k8s-worker-1: Setting the name of the VM: enron.corp.k8s.local-k8s-worker-node-1
==> k8s-worker-1: Fixed port collision for 22 => 2222. Now on port 2200.
==> k8s-worker-1: Clearing any previously set network interfaces...
```

Try it out!

[ ] https://github.com/nishaanthguna22/vagrant-deployment-files/tree/master/K8s-multicluster
[ ] https://t.ly/bVMv

In a Nutshell

- RBAC is critical for resource authz in the cluster
- Container breakouts often result in host compromise.
- Third-party plugins hoard critical vulnerabilities due to the excessive privileges on the SAs
- Compromise the pod with super-privileged SA equates to cluster-admin
- Do not expose the cluster’s API Server to the Internet without any firewall or network restrictions
- Certificate authentication is best suited for enterprise-grade clusters

References

- https://imgflip.com/
- https://twitter.com/memenetes
- https://cloud.google.com
- https://www.adaltas.com/en/2019/08/07/users-rbac-kubernetes/
- https://kubernetes.io
- https://microsoft.github.io/Threat-Matrix-for-Kubernetes
- https://blog.christophetd.fr/privilege-escalation-in-aws-elastic-kubernetes-service-eks-by-compromising-the-instance-role-of-worker-nodes/
- https://blog.calif.io/p/privilege-escalation-in-eks
- https://faun.pub/top-20-kubernetes-memes-b5cb4c5af395
- https://bishopfox.com/blog/kubernetes-pod-privilege-escalation