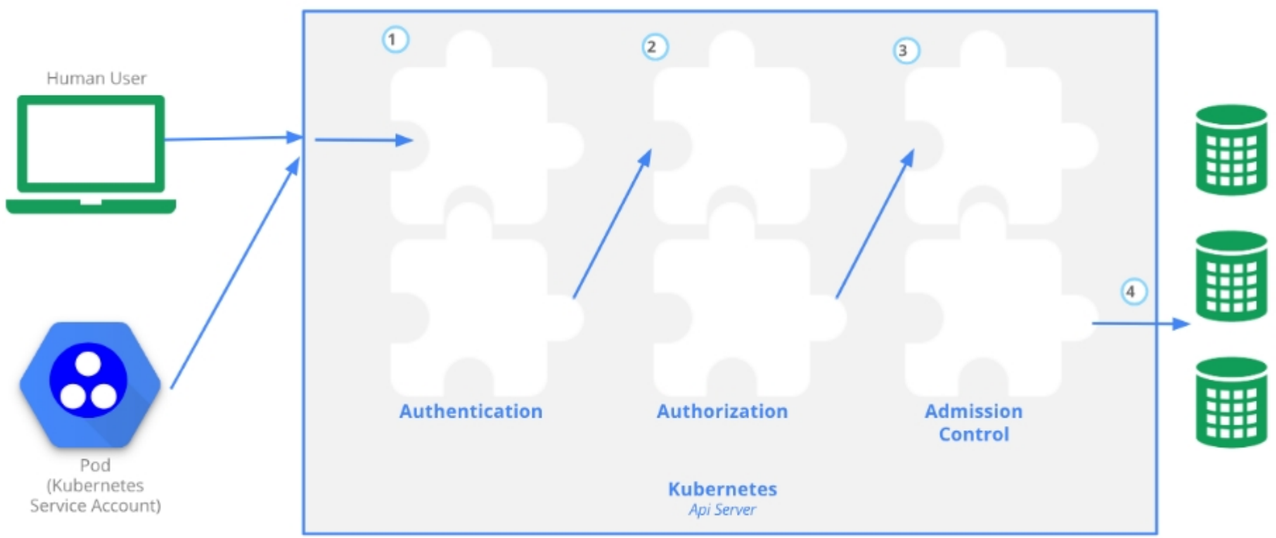
CKA Curriculum Part 5 - Security

Know how to configure authentication and authorization

We're basically talking about RBAC here.



Step 1 - Authentication

First step is Authentication which is how a user or service account identifies itself. Depending on the source, a respective authentication module is used. Authentication modules include the ability to authenticate from:

- Client Certificate
- Password
- Plain Tokens
- Bootstrap Tokens
- JWT Tokens (for service accounts)

All authentication is handled via HTTP over TLS.

Step 2 - Authorization

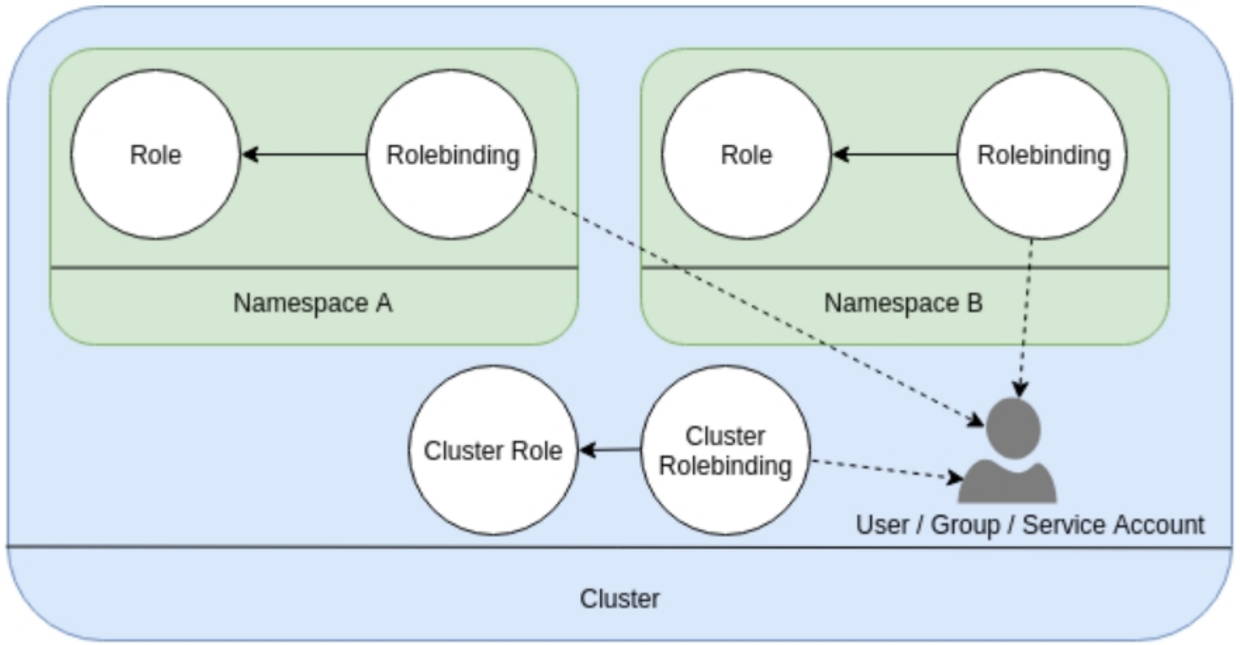
After a user or service account is authenticated, the request must then be authorized. Any authentication request is followed by some kind of action request, and the action defines the object(s) that request needs to apply to, and what the action is. For example, to list the pods in a given namespace.

Any and all requests are facilitated providing an existing policy gives the user those permissions.

Steps 3 & 4 - Admission Control

Admission Control Modules are software modules that can modify or reject requests. In addition to the attributes available to Authorization Modules, Admission Control Modules can access the contents of the object that is being created or updated. They act on objects being created, deleted, updated or connected (proxy), but not reads.

Role and Rolebindings



A role grants access to resources within a single namespace

A rolebinding grants the permissions from a role to a user, group or service accounts.

Cluster roles and rolebindings operate in a similar way, but obviously provide access to cluster-wide resources.

As an example, lets create a new user (VT) that authenticates via certificate and has the ability to read pods from the namespace “RBAC”.

Create the namespace

```
kubectl create namespace vt
```

Create the user

Create the private key for the user:

```
openssl genrsa -out vt.key 2048
```

Create the corresponding CSR:

```
openssl req -new -key vt.key -out vt.csr -subj "/CN=vt/O=virtualthoughts"
```

Create the certificate:

```
openssl x509 -req -in vt.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out vt.crt -days 500
```

**Create the user: **

```
kubectl config set-credentials vt --client-certificate=/etc/kubernetes/pki/vt.crt  --client-key=/etc/kubernetes/pki/vt.key
```

Create the Role

Roles do not directly grant permissions, think of these as a template for the type of access you want to implement to a given namespace or cluster.

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

Create the Rolebinding

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: readonly-pods
  namespace: vt
subjects:
- kind: User
  name: vt
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: readonly-pods # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

Understand Kubernetes security primitives

For this section we're looking at Pod Security Policies, which goven mechanisms around sensitive aspects of the pod spec. It is a cluster level resource and allows a cluster admin to configure a number of pod configuration items, which are listed at https://kubernetes.io/docs/concepts/policy/pod-security-policy/

Some examples are:

- readOnlyRootFilesystem** **- Containers cannot write locally
- Privileged - Determines if a container can enable privileged mode

Enabling PodSecurityPolicy

PodSecurityPolicy *is* an admission controller. Admission controllers in k8s are additional functionality we add in to the cluster, so that after authorization has occurred, we can ask admission controllers whether or not that user is able to do what they're requesting.

Firstly, we need to check the configuration of the kube-api server

```
sudo cat /etc/kubernetes/manifests/kube-apiserver.yaml
```

In particular, we're interested in the following flag:

```
- --enable-admission-plugins=NodeRestriction
```

To which we need to append PodSecurityPolicy to:

```
- --enable-admission-plugins=NodeRestriction,PodSecurityPolicy
```

Adding a Policy

The following policy prohibits the use of privileged containers:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
 name: psp-denyprivileged
spec:
 privileged: false  # Don't allow privileged pods!
 # The rest fills in some required fields.
 seLinux:
```

```
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

Using a Policy

Pods are created either via a service account or a user. The example below uses a service account to mock a non-admin user.

```
kubectl create namespace psp-example
kubectl create serviceaccount -n psp-example fake-user
kubectl create rolebinding -n psp-example fake-editor --clusterrole=edit --serviceaccount=psp-example:fake-user
```

Attempts to use the policy will initially fail, as by default policies do not do anything, they need a rolebinding.

```
kubectl --as=system:serviceaccount:psp-example:fake-user -n psp-example create -f-  <<EOF
apiVersion: v1
kind: Pod
metadata:
  name:      pause
spec:
  containers:
    - name:  pause
      image: k8s.gcr.io/pause
EOF

Error from server (Forbidden): error when creating "STDIN": pods "pause" is forbidden: unable to validate against any pod security policy: []
```

To bind this policy:

```
kubectl -n psp-example create role psp:unprivileged \
    --verb=use \
    --resource=podsecuritypolicy \
    --resource-name=psp-denyprivileged
role "psp:unprivileged" created

kubectl -n psp-example create rolebinding fake-user:psp:unprivileged \
    --role=psp:unprivileged \
    --serviceaccount=psp-example:fake-user
rolebinding "fake-user:psp:unprivileged" created
```

Subsequent attempts from the service account to create a privileged container should fail

```
kubectl --as=system:serviceaccount:psp-example:fake-user -n psp-example create -f-  <<EOF
apiVersion: v1
kind: Pod
metadata:
  name:      pause
spec:
  containers:
    - name:  pause
      image: k8s.gcr.io/pause
EOF

Error from server (Forbidden): error when creating "STDIN": pods "privileged" is forbidden: unable to validate against any pod security policy: [spec.containers[0].securityContext.privileged: Invalid value: true: Privileged containers are not allowed]
```

Know how to configure network policies

Determine how pods communicate with each other. By default this is left completely open. Think of this as firewall rules. We can apply network policies by:

- Pod selectors (labels)
- Namespace selectors
- CIDR blocks ranges

If we were to run a nginx container and curl to it from a pod, it works as expected.

```
/home # curl 10.10.57.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
   body {
```

To create a network policy, in this example we're denying all traffic. When this is applied, it will do so to the default namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

After applying and redoing the curl test, it fails:

```
/home # curl 10.10.57.2
curl: (7) Failed connect to 10.10.57.2:80; Connection timed out
```

A fleshed out network policy looks like the following:

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

Create and manage TLS certificates for cluster components

To get a new cluster up and running with all the required certificates, the following is required:

Cluster Wide

- CA - Used to sign further

- Admin user
- Service account

For Master Nodes:

- Kube-controller
- Kube-scheduler
- Kube-apiserver

For Worker Nodes:

- Kube-proxy
- Kubelet

Creation of these can be accomplished by using cssl. Best way to practice this whole exercise is to do Kubernetes the hard way (https://github.com/kelseyhightower/kubernetes-the-hard-way)

Work with images securely

CVE Scanning

Most container registries employ some sort of CVE scanning for components that have known exploits. Therefore, when working with container images, ensure they're scanned regularly by a trusted scanner. You can also leverage a private, internal container registry for this purpose.

Update nodes

Ensure that master and worker nodes are frequently updated with the latest security and operating system patches. Alternatively, cycle out nodes entirely with more up to date versions of themselves, but this is dependent on the level of automation present in the environment.

Use :latest with caution (or not at all)

The :latest flag on a container image may not always be the true, newest version of that image. Where possible, specify specific versions.

Additionally, using :latest can cause deployment issues. For example, an application can be written for ubuntu:latest and is specified as such. Ubuntu then release a new container image, which differs from the previous :latest, which causes instability. However, specifying to use ubuntu:19.04 will provide consistency. When a newer version comes out, it can be tested and implemented in a much more controlled manner.

Remove superfluous components from images

If libraries/components/applications are no longer required in the image, remove them accordingly. This not only optimises the subsequent containers, but reduces the attack surface/footprint

Leverage Network Security Policies

Lock down access to pods by adopting Network Security Policies. For example, if you have a web tier that purely listens on TCP 443, limit access to those pods to that specific port.

Define Security Contexts

A security context defines privilege and access control settings for a pod or a container. This includes (but not limited to):

- Discretionary Access Control: Permission to access an object, like a file, is based on user ID (UID) and group ID (GID).
- Security Enhanced Linux (SELinux): Objects are assigned security labels.
- Running as privileged or unprivileged.
- Linux Capabilities: Give a process some privileges, but not all the privileges of the root user.
- AppArmor: Use program profiles to restrict the capabilities of individual programs.
- Seccomp: Filter a process's system calls.
- AllowPrivilegeEscalation: Controls whether a process can gain more privileges than its parent process. This bool directly controls whether the no_new_privs flag gets set on the container process. AllowPrivilegeEscalation is true always when the container is: 1) run as Privileged OR 2) has CAP_SYS_ADMIN

Security contexts are defined within the pod configuration, under "spec" for both the pod and the container.

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
```

```
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

In the configuration file, the runAsUser field specifies that for any Containers in the Pod, all processes run with user ID 1000. The runAsGroup field specifies the primary group ID of 3000 for all processes within any containers of the Pod. If this field is ommitted, the primary group ID of the containers will be root(0). Any files created will also be owned by user 1000 and group 3000 when runAsGroup is specified. Since fsGroup field is specified, all processes of the container are also part of the supplementary group ID 2000. The owner for volume /data/demo and any files created in that volume will be Group ID 2000.

We can validate this by logging into the pod:

```
kubectl exec -it security-context-demo -- sh
/ $ ps
PID   USER     TIME  COMMAND
    1 1000      0:00 sleep 1h
    6 1000      0:00 sh
   12 1000      0:00 ps
/ $
```

Secure persistent key value store

For multi tiered applications, there will likely be sensitive information that needs to be accessed, such as :

- Usernames
- Passwords
- Keys
- SSL Certificates

Which can be created on the fly with the kubectl command:

```
kubectl create secret generic my-secret --from-literal=username=someusername --from-literal=password=somepassword
```

We can then use values from these secrets within pod specs, in this example we're populating a couple of environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: password
  restartPolicy: Never

kubectl exec -it secret-env-pod sh
# printenv
KUBERNETES_PORT=tcp://10.96.0.1:443
```

KUBERNETES_SERVICE_PORT=443
HOSTNAME=secret-env-pod
REDIS_DOWNLOAD_SHA=3ce9ceff5a23f60913e1573f6dfcd4aa53b42d4a2789e28fa53ec2bd28c987dd
HOME=/root
SECRET_PASSWORD=somepassword
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
SECRET_USERNAME=someusername
REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-5.0.4.tar.gz
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_PORT_HTTPS=443
REDIS_VERSION=5.0.4
GOSU_VERSION=1.10
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/data