CKS - Certified Kubernetes Security Specialist

17 May 2021

01 - Introduction

Course Introduction

- Kodekloud now a certified CNCF training partner
- Course structure:
    - Lectures
    - Demos
    - Quizzes
    - Hands-on-labs
    - Slack Channel available
    - Q&A available
- Prerequisites:
    - CKA Certification - Must be done as relies on many topics covered.
    - Docker
    - General DevOps and Linux knowledge
- Course Objectives
    - Aligned with exam objectives
    - Course structure not directly corresponding to CKS curriculum in terms of order, but all content is included.
- Mock Exams are “as close to the real exams as possible” and can be attempted as many times as desired.

02 - Understanding the Kubernetes Attack Surface

The Attack

- Demonstration of attacks on a Kubernetes environments
- Consider a Kubernetes application set running online, with two applications running in a domain each
- Suppose someone wanted to mess with these applications, the only initial information they have is the domain names, the architecture remains unknown
- Determining the IP addresses: ping <domain name>
    - The same IP address is returned for each, implying they are on the same server/infrastructure set
- Port scan of the IP address: zsh port-scan.sh <IP address>
    - All bar one port has a successful connection, Docker, implying that the applications are container-based
- Knowing the docker port is available, the next consideration is whether any authentication and authorization measures are in place within Docker
    - This is not enabled by default, IT MUST BE PRE-CONFIGURED.
- Testing the waters for the Docker infrastructure:
    - Docker ps -H <domain name>
        - The -H flag is used to specify the host, if not specified it will default to localhost
    - As no Docker authentication and authorization measures are in place, this shows all the containers within the infrastructure
- Docker version can also be identified by running docker -H <domain name> version
- To access the environment and other containers:
    - Docker -H <domain name> run --privileged -it ubuntu bash
    - Runs a basic ubuntu container with escalated privileges within the infrastructure, allowing ssh'ing into other containers within the infrastructure
- Suppose the hacker already has a script capable of exploiting the infrastructure's vulnerability(ies), they should be able to download it to their privileged container without issue:
    - Curl not found

    - Wget not found
    - Apt-get install <curl>
    - Apt-get update
    - Since the authentication isn't in place, the packages can be successfully installed and the hacker can run their script(s) to infiltrate the underlying infrastructure and mess around with the other applications, or learn additional information about the system e.g.:
        - Volume mounts: df -h
        - Username currently on system: uname
        - Host name: hostname
        - Additional containers: sudo docker ps
- Running these commands allows identification of a Kubernetes workload running in a container with Kubernetes-dashboard
- The Kubernetes dashboard must be running on a port somewhere, run sudo iptables -L -t nat | grep kubernetes-dashboard
    - Shows dashboard running on 30080
    - Dashboard can be accessed and viewed easily if there's no authentication and security controls setup
    - The dashboard provides information about pretty much everything within your Kubernetes cluster relating to storage, workloads, etc
    - Using this, information can be identified for the database container such as the authentication parameters, in this case they're listed as environment variables
    - From here the database container can be accessed and manipulated to the hacker's desire
- ALL OF THIS can be resolved via implementation of Kubernetes security measures discussed within this course.

The 4C's of Cloud Native Security

- The 4C's of Cloud Native Security were all exploited in the previous demo:
    - Cloud:
        - The infrastructure hosting the kubernetes cluster
        - Not properly secured, allowed access to the pods on the cluster

        - Could have been resolved by introducing network firewalls
    - Cluster
        - Relates to security via the Docker Daemon, Kubernetes API, etc
        - Relates to security aspects such as:
            - Authentication
            - Authorization
            - Admission
            - Network Policy
    - Container
        - Relating to security at a container-level, restrictions can be put in place to secure containers from particular repos, user privileges, etc.
        - Aspects covered include:
            - Image restriction - Only able to run images from a particular repository
            - Supply Chain
            - Sandboxing
            - Privileged - Certain activities should require escalated privileges
    - Code
        - The application code itself
        - Sensitive data should not be exposed directly in the code
        - Not covered directly in the course, though some practices such as utilising environment variables, key vaults such as Azure and HashiCorp vault, are touched on.

05 June 2021

03 - Cluster Setup and Hardening

CIS Benchmarks

- Security Benchmark - Predefined standards and best practices that should be implemented for server (or other appropriate device) security.
- Areas of consideration include:
    - Physical device configuration and limitation - USB ports that aren't expected to be used frequently / at all must have their access managed appropriately
    - Access configurations - What user accounts need to be configured? Can users log in as root?
        - Recommended that root is disabled by default and admins use sudo where required
        - Leads into only certain users having access to sudo, amongst other configurations
    - Network configuration
        - Firewall & IP Tables
        - Port Security
    - Service configuration
        - Ensure only certain services are allowed
    - Filesystem Configuration
        - All required permissions are set to the desired files
    - Auditing
        - Make sure all changes are logged for auditing purposes
    - Logging
- CIS - Centre for Internet Security
    - Commonly used tool to check if security best practices are implemented
    - Available on Linux, Windows, Cloud platforms, Mobile and many other platforms as well as Kubernetes.
- Can be downloaded from https://www.cisecurity.org/cis-benchmarks/

- Guides come with predefined instructions for your associated platform(s) best practices and how to implement them (commands included)
- CIS Provide tools such as the CIS-CAT tool to automate the assessment of best practices implementation
    - If any best practices aren't implemented, they are logged in the resultant HTML output report in a detailed manner.

Lab - Run CIS Benchmark Assessment Tool on Ubuntu

Q1: What is full form of CIS?

A: Center for Internet Security

Q2: What is not a function of CIS?

A: Monitor Global Internet Traffic

Q3: We have installed the CIS-CAT Pro Assessor tool called Assessor-CLI, under `/root`. Please run the assessment with the `Assessor-CLI.sh` script inside `Assessor-CLI` directory and generate a report called `index.html` in the output directory `/var/www/html/`.Once done, the report can be viewed using the `Assessment Report` tab located above the terminal.

Run the test in interactive mode and use below settings:
Benchmarks/Data-Stream Collections: : `CIS Ubuntu Linux 18.04 LTS Benchmark v2.1.0`
Profile : `Level 1 - Server`

`A:`

- `Run Assessor-CLI.sh`
- `Note options:`
    - `-i (interactive)`
    - `--rd <reports dir>`
    - `--rp <report prefix>`

- Run Assessor-CLI.sh with options for /var/www/html/ and index.html respectively (and --nts) and -i
    - Note: Run options in order that they are displayed
- Apply conditions for benchmark setting and profiles

Q4:How many tests failed for 1.1.1 Disable unused filesystems?

A: View report via tab and note - 6

Q5: How many tests passed for 2.1 Special Purpose Services?

A: ditto - 18

Q6:What parameters should we set to fix the failed test 5.3.10 Ensure SSH root login is disabled?

A: PermitRootLogin no

Q7:Fix the failed test - 1.7.6 Ensure local login warning banner is configured properly?

A: Find in CIS Report, run the associated command

Q8:Fix the failed test - 4.2.1.1 Ensure rsyslog is installed

A: Find area in report and run command

Q9: Fix the failed test - 5.1.2 Ensure permissions on /etc/crontab are configured

A: Find area in report and run command

Q10: In the previous questions we fixed the below 3 failed tests. Now run CIS-CAT tool test again and verify that all the below tests pass.

- 1.7.6 Ensure local login warning banner is configured properly
- 4.2.1.1 Ensure rsyslog is installed
- 5.1.2 Ensure permissions on /etc/crontab are configured

Run below command again to confirm that tests are passing now

```
sh ./Assessor-CLI.sh -i -rd /var/www/html/ -nts -rp index
```

Use below setting while running tests

Benchmarks/Data-Stream Collections: : `CIS Ubuntu Linux 18.04 LTS Benchmark v2.1.0`

Profile : `Level 1 - Server`

`A: Copy command and check associated tests pass - revisit questions if failed`

CIS Benchmarks in Kubernetes

- Kubernetes CIS benchmarks are readily available for download
- At the time of writing, the best practices are defined for versions 1.16-1.18
- The benchmarks are specifically aimed at system and application administrators, security specialists, auditors, help desk and platform deployment personnel who plan to develop, deploy, assess or secure solutions involving Kubernetes.
- Covers various security practices for master/control plane and worker nodes, such as permissions and configurations for API Server pods.
- Additional details provided to check recommended settings are in place and how to rectify any errors.
- CIS CAT Lite only supports certain OS's and benchmarks like Windows, Ubuntu, Kubernetes not included.
- CIS-CAT Pro includes Kubernetes support.

CIS Benchmarks Link: https://www.cisecurity.org/cis-benchmarks/#kubernetes

CIS CAT Tool: https://www.cisecurity.org/cybersecurity-tools/cis-cat-pro/cis-benchmarks-supported-by-cis-cat-pro/

Kube-Bench

- An open-source tool from Acqua security that can automate assessment of Kubernetes deployments in line with best practices

- The assessment occurs against the CIS Benchmarks
- To get started can either:
    - Deploy via Docker Container
    - Deploy as a pod in Kubernetes
    - Install the associated binaries
    - Compile from source

Lab - Kube-Bench

Q1: Kube-Bench is a product of which company?

A: Aqua Security

Q2: What should Kube-Bench be used for?

A: To check whether Kubernetes is deployed in line with best practices for security

Q3: Install Kube Bench in /root - version 0.4.0, Download file: `kube-bench_0.4.0_linux_amd64.tar.gz`

A:

curl -L https://github.com/aquasecurity/kube-bench/releases/download/v0.4.0/kube-bench_0.4.0_linux_amd64.tar.gz -o kube-bench_0.4.0_linux_amd64.tar.gz

tar -xvf kube-bench_0.4.0_linux_amd64.tar.gz

Q4: Run a kube-bench test now and see the results

Run below command to run kube bench

./kube-bench --config-dir `pwd`/cfg --config `pwd`/cfg/config.yaml

A: Follow instructions

Q5: How many tests passed for `Etcd Node Configuration`?

A: Review and report back - Section 2 and 7 pass

Q6: How many tests failed for `Control Plane Configuration`?

A: Review and report back - Section 3 and 0 failed

Q7: Fix this failed test `1.3.1 Ensure that the --terminated-pod-gc-threshold argument is set as appropriate`

Follow exact commands given in Remediation of given test

A: Find section in output and follow command

Edit the Controller Manager pod specification file /etc/kubernetes/manifests/kube-controller-manager.yaml

on the master node and set the --terminated-pod-gc-threshold to an appropriate threshold,

for example:

--terminated-pod-gc-threshold=10

Q8: Fix this failed test `1.3.6 Ensure that the RotateKubeletServerCertificate argument is set to true`

Follow exact commands given in Remediation of given test

A: Find section and follow instructions: Edit the Controller Manager pod specification file /etc/kubernetes/manifests/kube-controller-manager.yaml

on the master node and set the --feature-gates parameter to include RotateKubeletServerCertificate=true.

--feature-gates=RotateKubeletServerCertificate=true

Q9: Fix this failed test `1.4.1: Ensure that the --profiling argument is set to false`

A: Follow exact commands given in Remediation of given test

Q10:

Run the kube-bench test again and ensure that all tests for the fixes we implemented now pass

- `1.3.1 Ensure that the --terminated-pod-gc-threshold argument is set as appropriate`
- `1.3.6 Ensure that the RotateKubeletServerCertificate argument is set to true`
- `1.4.1: Ensure that the --profiling argument is set to false`

A: Run command - ./kube-bench --config-dir `pwd`/cfg --config `pwd`/cfg/config.yaml

And verify

Kubernetes Security Primitives

- Securing the hosts can be handled via methods such as disabling password authentication and allowing only SSH Key authentication
- Controlling access to API Server is the top priority - All Kubernetes operations depend upon this.
- Need to define:
    - Who can access the API Server?
    - What can they do with the API Server?
- For access, could use any of:
    - Files
    - Certificates
    - External authentication providers
    - Service Accounts
- For Authorization:
    - RBAC - Role-based access control
    - ABAC - Attribute-based access control
    - Node Authorization
- By default, all pods within a cluster can access one another
- This can be restricted via the introduction of network policies

Authentication

- In general, there are two types of accounts available:
    - User accounts - Admins, developers, etc
    - Service accounts - machine-based accounts
- All access to Kubernetes is handled via the Kube-API Server
- Authentication Mechanisms Available in Kubernetes for the API Server:
    - Static Password File - Not recommended
    - Static Token File
    - Certificates
    - Identity Services (3rd Party services e.g. LDAT)

- Suppose you have the user details in a file, you can pass this file as an option for authentication in the kube-apiserver.service file adding the flag:
  --basic-auth-file=user-details.csv

kube-apiserver.service

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --runtime-config=api/all \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --v=2
```

    - Restart the service / the server after this change is done
- If the cluster is setup using Kubeadm, edit the yaml file and add the same option

/etc/kubernetes/manifests/kube-apiserver.yaml

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.107
    - --allow-privileged=true
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true

    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
```

    - Kubernetes will automatically update the apiserver once the change is made
- To authenticate using the credentials specified in this manner run a curl command similar to:

  curl -v -k https://master-node-ip:6443/api/v1/pods -u "username:password"

- Could also have a token file, specify using **--token-auth-file=user-details.csv** in the api server .service or yaml file as appropriate
    - Note: the token can also be included in the curl request via **--header "Authorization: Bearer <TOKEN>"**
- **Note:** These are not recommended authentication mechanisms
    - Should consider a volume mount when providing the auth file in a **kubeadm** setup
    - Setup RBAC for new users

- Could also setup RBAC using YAML files to create rolebindings for each user

Service Accounts

- In Kubernetes there are two types of accounts:
    - **User** - Used by Humans e.g. Administrators
    - **Service** - Used by application services for various tasks e.g. Monitoring, CI/CD tools like Jenkins
- For an application to query the Kubernetes API, a service account is required
- Creation: **kubectl create serviceaccount <serviceaccount name>**
- View: **kubectl get serviceaccount**
- For detailed information: **kubectl describe serviceaccount <name>**
- When a serviceaccount is created, an authentication token is automatically created for it and stored in a Kubernetes secret.
    - Serviceaccount token used for application authentication to the kube-api server
    - The secret can be viewed via **kubectl describe secret <secret name>**
    - This token can then be passed as an Authorization Bearer token when making a curl request.
    - E.g **curl https://<IP-ADDRESS> -insecure --header "Authorization: Bearer <token>"**
- For 3rd party applications hosted on your own Kubernetes cluster:
    - The exporting of the service account token is not required
    - The service account secret can be mounted as a volume inside the pod hosting the application
- For every namespace, a default service account and its token are automatically assigned to a pod unless specified otherwise.
- Secret mounted as a volume for each pod - secret location viewable via **kubectl describe pod**
- **Note:** The default service account can only run basic Kubernetes API queries
- To specify a particular service account for a pod, add in the pod's spec field: **serviceAccount: name**
    - You must delete and recreate a pod if you wish to edit the service account on a pod.

TLS Basics

- Certificates used to guarantee trust between two parties communicating with one another, leading to a secure and encrypted connection
- Data involved in transmission should be encrypted via the use of encryption keys
- Encryption Methods:
    - Symmetric: Same key used for encryption and decryption - insecure
    - Asymmetric encryption: A public and private key are used for encryption and decryption specifically
        - Private key can only be used for decryption
- SSH Assymetric Encryption: run **ssh-keygen**
    - Creates **id_rsa** and **id_rsa.pub** (public and private keys)
    - Servers can be secured by adding public key to authorized key file at **~/.ssh/authorized_keys**
    - Access to the server is then allowed via **ssh -i id_rsa username@server**
    - For the same user, can copy the public key to any other servers
- To securely transfer the key to the server, use **asymmetric encryption**
- Can generate keys with: **openssl genrsa -out <name>.key 1024**
    - Can create public variant with: **openssl rsa -in <name>.key -pubout > <name>.pem**
- When the user first accesses the web server via HTTPS, they get the public key from the server
    - Hacker also gets a copy of it
- The users browser encrypts the servers symmetric key using their public key - securing the symmetric public key from the server
    - Hacker gets copy
- Server uses private key to decrypt user's private key - allows user to access server securely
    - Hackers don't have access to the servers private key, and therefore cannot encrypt it.
- For the hacker to gain access, they would have to create a similar website and route your requests
    - As part of this, the hacker would have to create a certificate

    - In general, certificates must be signed by a proper authority
    - Any fake certificates made by hackers must be self-signed
        - Web browsers have built-in functionalities to verify if a connection is secure i.e. is certified
- To ensure certificates are valid, the Certificate Authorities (CAs) must sign and validate the certs.
    - **Examples: Symanteg, Digicert, GlobalSign**
- To validate a certificate, one must first generate a certificate signing request to be sent to the CA: **openssl req -new -key <name>.key -out <name>.csr -subj "/C=US/ST=CA/O=MyOrg, Inc./CN=mybank.com"**
- CAs have a variety of techniques to ensure that the domain is owned by you
- How does the browser know what certificates are valid? E.g. if the certificate was assigned by a fake CA?
    - CAs have a series of public and private keys built in to the web browser(s) used by the user,
    - The public key is then used for communication between the browser and CA to validate the certificates
- **Note:** The above are described for public domain websites
- For private websites, such as internal organisation websites, private CAs are generally required and can be installed on all instances of the web browser within the organisation. Organizations previously outlined generally offer enterprise solutions.
- **Note:**
    - Certificates with a public key are named with the extension .crt or .pem, with the prefix of whatever it is being communicated with
    - Private keys will have the extension of either **.key** or **-key.pem**
- **Summary:**
    - Public key used for encryption only
    - Private keys used for decryption only

TLS In Kubernetes

- In the previous section, three types of certificates were discussed, for the purposes of discussing them in Kubernetes, how they're referred to will change:

    - Public and Private Keys used to secure connectivity between the likes of web browsers and servers: **Server Certificates**
    - Certificate Authority Public and Private Keys for signing and validating certificates: **Root Certificates**
    - Servers can request a client to verify themselves: **Client Certificates**
- **Note:**
    - Certificates with a public key are named with the extension .crt or .pem, with the prefix of whatever it is being communicated with
    - Private keys will have the extension of either **.key** or **-key.pem**
- All communication within a Kubernetes cluster must be secure, be it:
    - Pods interacting with one another
    - Services with their associated clients
    - Accessing the APIServer using the Kubectl utility
- Secure TLS communication is a requirement
- Therefore, it is required that the following are implemented:
- Server Certificates for Servers
    - Client Certificates for Clients
- **Server Components:**
    - Kube-API Server
        - Exposes an HTTPS service that other components and external users use to manage the Kubernetes cluster
        - Requires certificates and a key pair to be generated
            - apiserver.crt (public) and apiserver.key (private key)
    - ETCD Server
        - Stores all information about the cluster
        - Requires a certificate and key pair
            - Etcdserver.crt and apiserver.key
    - Kubelet server:
        - Exposes HTTPS API Endpoint that the Kube-API Server uses to interact with the worker nodes
            - kubelet.crt and kubelet.key
- **Client Certificates:**
    - All of the following require access to the Kube-API Server

    - Admin user
        - Requires certificate and key pair to authenticate to the API Server
        - admin.crt and admin.key
    - Scheduler
        - Client to the Kube-APIServer for object scheduling pods etc
            - scheduler.crt and scheduler.key
    - Kube-Controller:
        - controller-manager.crt and controller-manager.key
    - Kube-Proxy
        - kube-proxy.crt and kube-proxy.key
- **Note**: The API Server is the only component that communicates with the ETCD server, which views the API server as a client
    - The API server can use the same keys as before for serving itself as a service OR a new pair of certificates can be generated specifically for ETCD Server Authentication
    - The same principle applies for the API Server connecting to the Kubelet service
- To verify the certificates, a CA is required. Kubernetes requires at least 1 CA to be present; which has its own certificate and key (**ca.crt** and **ca.key**)

06 June 2021

TLS In Kubernetes - Certificate Creation

- Tools available for certificate creation include:
    - EASYRSA
    - OPENSSL - The more common one (used in this example)
    - CFSSL
- **Steps - Server Certificates: CA Example**
    - Generate the private key: **openssl genrsa -out ca.key 2048**
        - The number "2048" in the above command indicates the size of the private key. You can choose one of five sizes: 512, 758, 1024, 1536 or 2048 (these numbers represent bits). The larger sizes offer greater

security, but this is offset by a penalty in CPU performance. We recommend the best practice size of 1024.

    - Generate certificate signing request: **openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr**
    - **Note: "/CN=<NAME>"** outlines the name of the component the certificate is for.
    - Sign certificates, this is self-signed via the ca key pair: **openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt**
        - Results in CA having a private key and root certificate file
- **Client Certificate Generation Steps: Admin User Example**
    - Generate the keys: **openssl genrsa -out admin.key 2048**
    - Generate certificate signing request: **openssl req -new -key admin.key -subj "/CN=kube-admin" -out admin.csr**
    - Sign the certificate: **openssl x509 -req -in admin.csr -CAkey ca.key -out admin.crt**
        - The CA key pair is used for signing the new certificate, thus proving its validity
- When the admin user attempts to authenticate to the cluster, it is the certificate **admin.crt** that will be used for this
- For **non-admin users**, need to add group details to the certificate signing request to signal this.
    - Group called **SYSTEM:MASTERS** has administrative privileges, to specify this for an admin user, the signing request should be like: **openssl req -new -key admin.key -subj "/CN=kube-admin/O=system:masters" -out admin.csr**
- The same procedure would be followed for client certificates for the **Scheduler, Controller-Manager and Kubelet** - prefix with SYSTEM
- The certificates generated could be applied in different scenarios:
    - Could use the certificate instead of usernames and password in a REST API call to the api server (via a **curl** request).
        - To do so, specify the key and the certs involved as options following the request e.g. **--key admin.key --cert admin.crt and --cacert ca.crt**
    - Alternatively, all the parameters could be moved to the kube config yaml file, which acts as a centralized location to reference the certificates

- **Note:** For each of the Kubernetes components to verify one another, they need a copy of the CA's root certificate
- **Server Certificate Example: ETCD Server**
    - As ETCD server can be deployed as a cluster across multiple servers, you must secure communication between the cluster members, or peers as they're commonly known as.
    - Once generated, specify the certificates when starting the server
    - In the etcd yaml file, there are options to specify the peer certificates

```
cat etcd.yaml
- etcd
    - --advertise-client-urls=https://127.0.0.1:2379
    - --key-file=/path-to-certs/etcdserver.key
    - --cert-file=/path-to-certs/etcdserver.crt
    - --client-cert-auth=true
    - --data-dir=/var/lib/etcd
    - --initial-advertise-peer-urls=https://127.0.0.1:2380
    - --initial-cluster=master=https://127.0.0.1:2380
    - --listen-client-urls=https://127.0.0.1:2379
    - --listen-peer-urls=https://127.0.0.1:2380
    - --name=master
    - --peer-cert-file=/path-to-certs/etcdpeer1.crt
    - --peer-client-cert-auth=true
    - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
    - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    - --snapshot-count=10000
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

- **Server Certificate Example: API Server:**
    - Same procedure involved, but due to the nature of the API Server, with essentially every operation running via it, and it being referred to by multiple names, requires a lot more information included; requiring an **openssl config file**
        - Using the config file, specify DNS and IP aliases for the components
            - Acceptable names:
                - Kubernetes
                - Kubernetes.default
                - Kubernetes.fefault.svc

        - kubernetes.default.svc.cluster.local
    - When generating the signing request, you can reference the config file by appending: **--config <config name>.cnf** to the signing request command
    - From this, the certificate can be signed using the ca.crt and ca.key file as normal
    -
- The location of all certificates are passed into the exec start file for the api server or the service's configuration file, specifically:

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/apiserver-kubelet-client.crt \\
  --kubelet-client-key=/var/lib/kubernetes/apiserver-kubelet-client.key \\
  --kubelet-https=true \\
  --runtime-config=api/all \\
  --service-account-key-file=/var/lib/kubernetes/service-account.pem \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --tls-cert-file=/var/lib/kubernetes/apiserver.crt \\
  --tls-private-key-file=/var/lib/kubernetes/apiserver.key \\
  --v=2
```

    - Etcd:
        - CA File = --etcd-cafile
        - ETCD Certificate = --etchd-certfile
        - ETCD Private Key File = --etcd-keyfile
    - Kubelet
        - CA File = --kubelet-certificate-authority
        - Client Certificate = --kubelet-client-certificate
        - Client Key = --kubelet-client-key
    - Client CA = --client-ca-file
    - API Server Cert and Private Key = --tls-cert-file, --tls-private-key-file

- **Kubelet Server:**
    - A HTTPS API server on each worker node to help manage the nodes
    - Key-Certificate pair required for each worker node in the cluster
    - Named after each node
    - Must be referenced in the kubelet config file for each node, specifically:
        - **Client CA file**
        - **TLS Certificate file** (kubelet-node01.crt for example)
        - **TLS Private Key File** (kubelet-node01.key for example)
    - Client certificates used to authenticated to the Kube API Server
        - Naming convention should be **system:node:nodename**
        - Nodes must also be added to **system:nodes** group for associated privileges

View Certificate Details

- The generation of certificates depends on the cluster setup
    - If setup manually, all certificates would have to be generated manually in a similar manner to that of the previous sections
        - Components deployed as native services in this manner
    - If setup using a tool such as kubeadm, this is all pre-generated
        - Components deployed as pods in this manner
- For **Kubeadm clusters**:
    - Component found in **/etc/kubernetes/pki/** folder
        - Certificate file paths located within component's yaml files
        - Example: **apiserver.crt**
        - Use **openssl x509 -in /path/to/.crt file -text -noout**
    - Can check the certificate details such as name, alternate names, issuer and expiration dates
- Note: Additional details available in the documentation for certificates
- Use **kubectl logs <podname>** on kubeadm if any issues are found with the components
- If kubectl is unavailable, use Docker to get the logs of the associated container:
    - Run **docker ps - a** to identify the container ID
    - View the logs via **docker logs <container ID>**

A sample health check spreadsheet can be found here:
https://github.com/mmumshad/kubernetes-the-hard-way/tree/master/tools

Lab - View Certificates

Certificates API

- All certificates have an expiration date, whenever the expiry happens, keys and certificates must be re-generated
- As discussed, the signing of the certificates is handled by the CA Server
- The CA server in reality is just a pair of key and certificate files generated
- Whoever has access to these files can sign any certificate for the Kubernetes environment, create as many users they want and set their permissions
- Based on the previous point, it goes without saying these files need to be protected
    - Place the files on a fully secure server
    - The server that securely hosts these files becomes the “CA Server”
    - Any time you want to sign a certificate, it is the CA server that must be logged onto/communicated with
- For smaller clusters, it’s common for the CA server to actually be the master node
    - The same applies for a kubeadm cluster, which creates a CA pair of files and stores that on the master node
- As clusters grow in users, it becomes important to automate the signing of certificate requests and renewing expired certificates; this is handled via the **Certificates API**
    - When a certificate needs signing, a **Certificate Signing Request** is sent to Kubernetes directly via an API call
    - Instead of an admin logging onto the master node and manually signing the certificate, they create a Kubernetes API object called CertificateSigningRequest
    - Once the API object is created, any requests like this can be seen by administrators across the cluster
    - From here, the request can be reviewed and approved using kubectl, the resultant certificate can then be extracted and shared with the user

- Steps:
    - User generates key: **openssl genrsa -out <keyname>.key 2048**
    - User generates certificate signing request and sends to administrator: **openssl req -new -key <key>.name -subk "/CN=name" -out name.csr**
    - Admin receives request and creates the API object using a manifest file, where the spec file includes the following:
        - Groups - To set the permissions for the user
        - Usages - What is the user able to do with keys with this certificate to be signed?
        - Request - The certificate signing request associated with the user, which must be encoded in base64 language first i.e. **cat cert.crt | base64**
        - Admins across the cluster can view certificate requests via: **kubectl get csr**
        - If all's right with the csr, any admin can approve the request with: **kubectl certificate approve <name>**
        - You can view the CSR in a YAML form, like any Kubernetes object by appending **-o yaml** to the **kubectl get** command i.e. **kubectl get csr <user> -o yaml**
            - Note: The certificate will still be in base64 code, so run: **echo "CODED CERTIFICATE" | base64 --decode**
- **Note:** The controller manager is responsible for all operations associated with approval and management of CSR
- The controller manager's YAML file has options where you can specify the key and certificate to be used when signing certificate requests:
    - **--cluster-signing-cert-file**

- **--cluster-signing-key-file**

```
cat /etc/kubernetes/manifests/kube-controller-manager.yaml
spec:
  containers:
  - command:
    - kube-controller-manager
    - --address=127.0.0.1
    - --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
    - --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
    - --controllers=*,bootstrapsigner,tokencleaner
    - --kubeconfig=/etc/kubernetes/controller-manager.conf
    - --leader-elect=true
    - --root-ca-file=/etc/kubernetes/pki/ca.crt
    - --service-account-private-key-file=/etc/kubernetes/pki/sa.key
    - --use-service-account-credentials=true
```

Lab - Certificates API

KubeConfig

- Files containing information for different cluster configurations, such as:
  - **--server**
  - **--client-key**
  - **--client-certificate**
  - **--certificate-authority**
- The existence of this file removes the need to specify the option in the CLI
- File located at **$HOME/.kube/config**
- KubeConfig Files contain 3 sections:
  - **Clusters** - Any cluster that the user has access to, local or cloud-based
  - **Users** - User accounts that have access to the clusters defined in the previous section, each with their own privileges
  - **Contexts** - A merging of clusters and users, they define which user account can access which cluster
- These config files do not involve creating new users, it's simply configuring what existing users, given their current privileges, can access what cluster
- This removes the need to specify the user certificates and server addresses in each kubectl command

    - --server spec listed under clusters
    - User keys and certificates listed in Users section
    - Context created to specify that the user “MyKubeAdmin” is the user that is used to access the cluster “MyKubeCluster”
- **Config file defined in YAML file**
    - ApiVersion = v1
    - Kind = Config
    - Spec includes the three sections defined previously, all of which are arrays
    - Under clusters: specify the cluster name, the certificate authority associated and the server address
    - Under users, specify username and associated key(s) and certificate(s)
    - Under contexts:
        - Name format: username@clustername
        - Under context specify cluster name and users
    - Repeat for all clusters and users associated
- The file is automatically read by the kubectl utility
- Use current-context field in the yaml file to set the current context
- CLI Commands:
    - View current config file being used: **kubectl config view**
        - Default file automatically used if not specified
        - To view non-default config files, append: --kubeconfig=/path/to/file
    - To update current context: **kubectl config use-context <context-name>**
    - Other commands available via **kubectl config -h**
    - Default namespaces for particular contexts can be added also under the context area in the Config file
- **Note**: for certificates in the config file, use the full/absolute path to specify the location
    - Alternatively use **certificate-authority-data** to list certificate in **base64** format

Labs - KubeConfig

API Groups

- API Server accessible at master node IP address at port 6443:
  curl https://kube-master:6443/
    - To get the version, append /version to a curl request to the above IP address
    - To get a list of pods, append /api/v1/pods
- Kubernetes' API is split into multiple groups depending on the group's purpose such as
    - **/api** - core functionalities e.g. pods, namespaces, secrets
    - **/version** - viewing the version of the cluster
    - **/metrics** - used for monitoring cluster health
    - **/logs** - for integration with 3rd-party logging applications
    - **/apis** - named functionalities added to kubernetes over time such as deployments, replicasets, extensions
        - Each group has a version, resources, and actions associated with them
    - **/healthz** - used for monitoring cluster health
- Use curl http://localhost:6443 -k to view the api groups, then append the group and **grep name** to see the subgroups within
- **Note**: Need to provide certificates to access the api server or use **kubectl proxy** to view
- **Note**: kubectl proxy is not the same as kube proxy, the former is an http proxy service to access the api server.
  Kube proxy is used to allow connectivity between kubernetes pods and services across nodes in a cluster.

Authorization

- When adding users, need to ensure their access levels are sufficiently configured, so they cannot make any unwanted changes to the cluster
- This applies to any **physical users**, like developers, or **virtual users** like applications e.g. Jenkins
- Additional measures must be taken when sharing clusters with organizations or teams, so that they are restricted to their specific namespaces
- Authorization mechanisms available are:

    - Node-based
    - Attribute-Based
    - Rule-Based
    - WebHook-based
- **Node-Based:**
    - Requests to the **kube-apiserver** via users and the **kubelet** are handled via the Node Authorizer
    - Kubelets should be part of the **system:nodes group** i.e. **system:node:nodename**
    - Any requests coming from a user with the name system-node and is part of the system nodes group is authorized and granted access to the apiserver
- **ABAC - Attribute-Based:**
    - For users wanting to access the cluster, you should create a policy in a **JSON** format to determine what privileges the user gets, such as namespace access, resource management and access, etc - this can then be passed to the API server for authorization
    - Repeat for each users
    - Each policy must be edited manually for changes to be made, the **kube apiserver** must be restarted to make the changes take effect
    - Not generally used as difficult to manage
- **RBAC:**
    - Instead of associating each user with a set of permissions, can create a role which outlines a particular set of permissions
    - Assign users to the role
    - If any changes are to be made, it is just the role configuration that needs to be changed, rather than restarting the entire api server again.
- **Webhook:**
    - Use of third-party tools to help with authorization e.g. Open Policy Agent
    - If any requests are made to say the APIserver, the third party can verify if the request is valid or not
- **Note:** Additional authorization methods are available:
    - AlwaysAllow - Allows all requests without checks
    - AlwaysDeny - Denies all requests without checks

- Authorizations set by **--authorization** option in the **apiserver's .service or .yaml file**
- Can set modes for multiple-phase authorization, use **--authorization-mode** and list the authorization methods
    - Authorization will be done in the order of listing in this option

RBAC

- To create a role, create a YAML file
- **apiVersion: rbac.authorization.k8s.io/v1**
- Spec replaced with **rules**
    - Covers apiGroups, resources and verbs
- Multiple rules added by - apiGroups for each
    - Create the role using **kubectl create -f**
- To link the user to the role, need to create a **Role Binding**
- Under metadata:
    - Specify **subjects** - Users to be affected by the rolebinding, their associated apiGroup for authorization
    - **RoleRef** - The role to be linked to the subject
- To view roles: **kubectl get roles**
- To view rolebindings: kubectl get rolebindings
- To get additional details: kubectl describe role/rolebinding <name>
- To check access level: **kubectl auth can-i <command/activity>**
- To check if a particular user can do an activity, append **--as <username>**
- To check if an activity can be done via a user in a particular namespace, append **--namespace <namespace>**
- Note: Can restrict access to particular resources by adding **resourceNames: ["resource1", "resource2", ...]** to the role yaml file

Labs - RBAC

Cluster Roles and Role Bindings

- Roles and role bindings are created for particular namespaces and control access to resources in that particular namespace

- By default, roles and role bindings are applied to the default namespace
- In general, resources such as pods, replicasets are namespaced
- Cluster-scoped resources are resources that cannot be associated to any particular namespace, such as:
    - Persistentvolumes
    - Nodes
- To switch view namespaced/cluster-scoped resources: **kubectl api-resources --namespaced=TRUE/FALSE**
- To authorize users to cluster-scoped resources, use **cluster-roles** and **cluster-rolebindings**
    - Could be used to configure node management across a cluster, such as cluster or cluster storage administrator(s).
- Cluster roles and role bindings are configured in the exact same manner as roles and rolebindings; the only difference is the kind (ClusterRole and ClusterRoleBinding)
- Note: Cluster roles and role bindings can be applied to namespaced resources, this will allow users to have access to particular resources for anywhere in the cluster.
- Many cluster roles are created via Kubernetes by default.

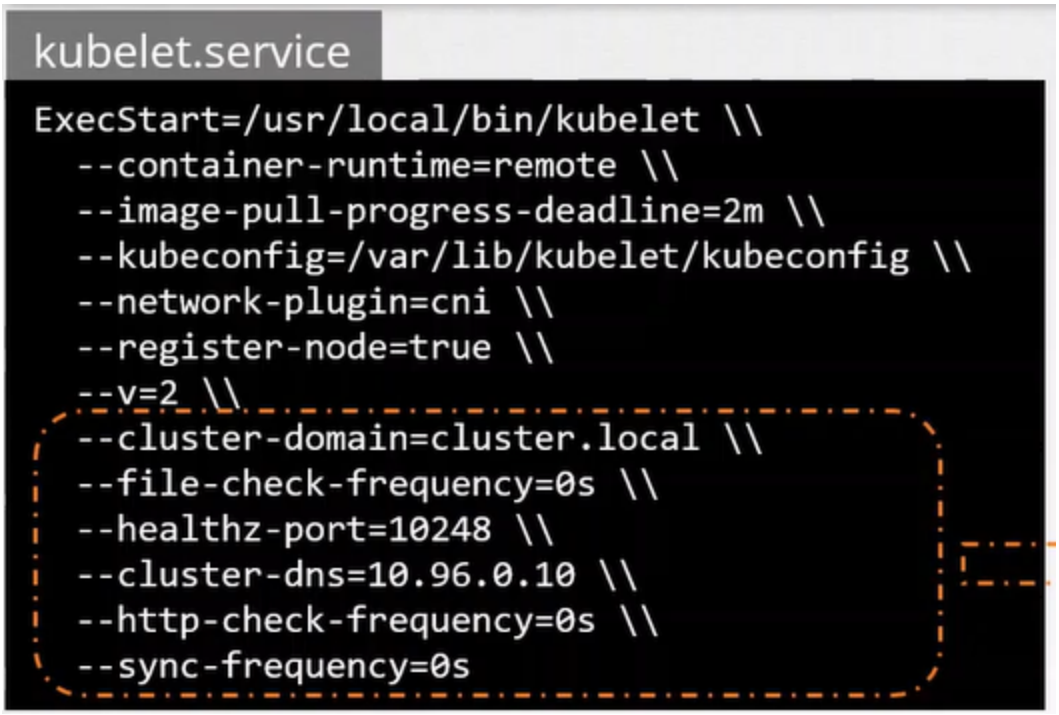
Labs - Cluster Roles and Role Bindings

12 Sept 2021

Kubelet Security

- Kubelet leads all the activities on a node to manage and maintain the node, carrying out actions such as:
    - Loading or unloading containers based on the kube-schedulers demands
    - Sending regular reports on worker node status to the api server
- Due to the importance of the kubelet, it's highly important to secure it and the communications between the kubernetes master node, api server, and worker nodes are secure.
- To refresh, the Kubelet, in worker nodes:
    - Registers the node with the cluster

    - Carries out instructions to run containers and container runtime
    - Monitor node and pod status on a regular basis
    - The kubelet can be installed as a binary file via a wget
        - Via kubeadm, it is automatically downloaded but not automatically deployed
- The kubelet configuration file varies in appearance. Previously, it was viewable as a .service file:



- Since kubernetes v1.10, options from --cluster-domain were moved to kubelet-config.yaml for ease of configuration and management
- In the kubelet.service file, the kubelet-config file path is passed via the --config flag.
    - Note: any flags at CLI-level will override the .service file's value
- The kubeadm tool does not download or install the kubelet, but it can help manage the kubelet configuration
    - Suppose there is a large number of worker nodes, rather than manually creating the config file in each of the nodes, the kubeadm tool can help automatically configure the kublet-config file associated with those nodes when joining them to the master node.
- Once kubelet is configured, there a number of frequent commands that can be used, including:

    - View the kubelet options:
        - **ps -aux | grep kubelet**
          Views the associated options for the kubelet
        - **cat /var/lib/kublet/config.yaml**
          View the config yaml for the kubelet
- Securing the kubelet, at a high level, involves taking actions to ensure the kubelet responds only to the kube-apiserver's requests
- Kubelet serves on 2 ports:
    - 10250 - Serves API allowing full access
    - 10255 - Serves an API that allows unauthenticated read-only access
- By default, kubelet allows anonymous access to the api, e.g.
    - Running **curl -sk htpps://localhost:10250/pods/** reproduces a list of all the pods
    - Running **curl -sk https://localhost:10250/logs/syslogs** returns the system logs of the node that kubelet is running on
- For the service at 10255 - this provides access read-only access to metrics to any unauthorized clients
- The above two services pose significant security risks, as anyone knowing the host IP address for the node can identify information regarding the node and cause damage if desired based on the various API calls available.
- Securing the kubelet boils down to authentication and authorization
    - Authentication determines whether the user can access the kubelet API
    - Authorization determines whether the user has sufficient permissions to perform a particular task with the API
- Authentication:
    - By default, the kubelet permits all requests without authentication
    - Any requests are labelled to be from user groups "anonymous" part of an unauthenticated group.
    - This can be disabled in either the kubelet.service file or the yaml file by setting --anonymous-auth to false, as shown below:

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  ...
  --anonymous-auth=false
  ...
```

kubelet-config.yaml

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authentication:
  anonymous:
    enabled: false
```

- Following the disabling of anonymous access, a recommended authentication method needs to be enabled. Generally there are two to choose from:
    - Certificates (X509)
    - API Bearer Tokens
- Following the creation of a pair of certificates, the ca file should be provided via the following option in the kubelet service file:
  --client-ca-file=/path/to/ca.crt

  Or to the kubelet-config.yaml file as shown below:

kubelet-config.yaml

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authentication:
  x509:
    clientCAFile: /path/to/ca.crt
```

- Now that the certificate is configured, the client certificates must be supplied in any curl commands made to the API i.e.:
  curl -sk htpps://localhost:10250/pods/ -key kubelet-key.pem -cert kubelet-cert.pem
- As far as the kubelet is concerned, the kube-apiserver is a client, therefore the apiserver should also have the kubelet client certificate and key configured in the

apiserver's service configuration, as shown below:

```
cat /etc/systemd/system/kube-apiserver.service
[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  ...
  --kubelet-client-certificate=/path/to/kubelet-cert.pem \\
  --kubelet-client-key=/path/to/kubelet-key.pem \\
```

- Remember: the kube-apiserver is itself a server and therefore has its own set of certificates, all other kubernetes certificates will also require sufficient configurations
- Note: Kubeadm also uses this approach when trying to secure the kubelet
- Note: If neither of the above authentication mechanisms explicitly reject a request, the default behaviour of kubelet is to allow the request under the username system:anonymous and group system:unauthenticated
- Authorization:
    - Once the user has access to the kubelet, what can they do with it?
    - By default, the mode AlwaysAllow is set, allowing all access to the API

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  ...
  --authorization-mode=AlwaysAllow
  ...
```

kubelet-config.yaml

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authorization:
  mode: AlwaysAllow
```

    - To change / rectify this, the authorization mode can be set to Webhook - the kubelet makes a call to the API server to determine if the request can be granted or not.
- Considering the metrics service running on 10255/metrics:
    - By default, this runs due to the kubelet config or service file having the --read-only-port set to 10255
    - If set to 0, the service is disabled and users cannot access it
- Summary:
    - By default, the kubelet allows anonymous authentication
    - To prevent this, you can set the anonymous flag to false - this can be done either in the kubelet.service or kubelet-config.yaml files
    - Kubelet supports two authentication methods:
        - Certification-based authentication
        - API Bearer Tokens
    - By default, the authorization mode is set to “always allow” - this can be prevented by changing the mode to webhook to authorize via webhook calls to the kube-api server
    - By default, the readonly port is set to 10255, allowing unauthorized access to critical kubernetes metrics, this can be disabled by setting the --read-only-port to 0.

Labs - Kubelet Security

07 Oct 2021

Note: The previous date entry was spent catching up on all the notes prior to the 12th September due to an unintentional hiatus, I didn’t literally just do 1 lecture on that day!

Kubectl Proxy & Port Forward

- In the CKA Course, it was learned that the kubectl tool can be used to interact with the Kubernetes API server
- No API authentication required as it was applied in the kube config file

- Kubectl can be found anywhere e.g. master node, a personal laptop (with a cluster in a VM)
- No matter where the cluster is, so long as the kubeconfig is appropriately configured with the security credentials, the kubectl tool can be used to manage it.
- Note: the api server could also be accessed via a curl command to the IP address and port 6443
    - If done via this method, would also need to supply the certificate and key files, such that the curl command would be similar to:

      Curl http://<kube-api-server-ip>:6443 -k
      --key admin.key
      --cert admin.crt
      --cacert car.crt
- Alternatively, one can start a proxy client using kubectl
  Kubectl proxy -> starts to serve on **localhost:8001**
    - Launches a proxy service and uses the credentials from the config file to access the API server
- Proxy only runs on laptop and is only accessible from this device
- By default, accepts traffic from the loopback address localhost -> not accessible from outside of the laptop.
- Can use kubectl proxy to make ANY request to the API server and services running within it.
- Consider an nginx pod exposed as a service only accessible within the cluster (if clusterIP service):
    - Use kubectl proxy as part of a curl command:

      curl
      http://localhost:8001/api/v1/namespaces/default/services/nginx/proxy/
    - Allows access to remote clusters as if they were running locally
- Alternative: Port-Forward
    - Takes a resource (pod, deployment, etc) as an argument and specifies a port on the host that you would like traffic to be forwarded to (and the service port)

    - Example: kubectl port-forward service/nginx 28080:80
    - The service can then be accessed via curl http://localhost:28080/
- Allows remote access to any cluster service you have access to.

Labs - Kubectl Proxy & Port Forward

20/10/2021

Kubernetes Dashboard

- A kubernetes sub-project used for a variety of functions including:
    - Get a graphical representation of your cluster
    - Monitor resource performance
    - Provision resources
    - View and manage secrets or not easily viewable resources.
- Due to these functionalities, it's important to ensure that the dashboard is secured appropriately.
- In earlier releases, access control was limited - led to many dashboard cyberattacks e.g. Tesla and Cryptocurrency mining.
- Deployment:
    - Can deploy by applying the recommended configuration from https//<path-to-kubernetes-dashboard>/recommended.yaml
    - Deploys:
        - Namespace - kubernetes-dashboard
        - Service - kubernetes-dashboard
        - Secrets - Certificates associated with the dashboard
    - Note: The service is not set to LoadBalancer by default, instead ClusterIP - this is in line with best practices so it is only accessible within the cluster VMs.
- Accessing the Dashboard:
    - Would typically want to access from the users laptop / device separate to the cluster.
    - Access will require the kubectl utility and kubeconfig file, as well as kube proxy.

- Run kubectl proxy - proxies all the requests to the api server on the cluster to the machine running the proxy
    - Can then access the dashboard via localhost:8001/<URL to Dashboard> e.g.:

      https://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/htpps:kubernetes-dashboard:/proxy

- Accessing the dashboard via a proxy is not applicable for teams requesting access to the dashboard.
    - Allowing access requires additional configuration to ensure only users of sufficient permissions have access - this is due to the ClusterIP service type.
    - Possible solution: set service type to LoadBalancer - not recommended as would make the dashboard public.
    - Possible solution: set service type to NodePort
        - This allows the dashboard to be accessible via the ports on the node, more advised than the previous suggestion assuming a sufficiently secure network.
    - Possible solution: Configure an Authentication Proxy e.g. OAuth2
        - Out of scope for the course, but a highly recommended option.
        - If users authenticate appropriately, traffic routed to Dashboard.

Securing Kubernetes Dashboard

Various authentication methods available:

- Token
    - Requires user creation with sufficient permissions via RBAC
    - Kubernetes dashboard documentation provides instructions for this, but this is particularly geared towards cluster admins only.
    - In general, requires creation of user service accounts, roles and role bindings

    - Once created, the token can be found by viewing the secret created relating to the service account.
- Kubeconfig
    - Requires passing of appropriate kubeconfig file that has the sufficient credentials to authenticate.

References:

- https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/
- https://redlock.io/blog/cryptojacking-tesla
- https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/
- https://github.com/kubernetes/dashboardhttps://www.youtube.com/watch?v=od8TnIvuADg
- https://blog.heptio.com/on-securing-the-kubernetes-dashboard-16b09b1b7aca
- https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md

Labs - Securing Kubernetes Dashboard

Verify Platform Binaries Before Deployment

- Kubernetes platform binaries are available via the Kubernetes Github repo's release
- As part of security best practices, it's important to ensure said binaries are safe for use
    - This can be done by comparing the binary checksum with the checksum listed on the website.
    - The reason for this check needing to occur is due to the possibility of the download being intercepted by attackers.
    - Even the smallest of changes can cause a complete change to the hash
- Binaries downloaded by curl command i.e. **curl <url> -L -o <filename>**
- Checksum can be viewed by using the shasum utility **(Mac): shasum -a 512 <filename>** or **sha512sum <filename> (Linux)**
    - Checksum can then be compared against the release page checksum
- Reference Links:

- https://kubernetes.io/docs/setup/release/notes
- https://github.com/kubernetes/kubernetes/tree/master/CHANGELOG

Labs - Verify Platform Binaries Before Deployment

27/10/2021

Kubernetes Software Versions

- API Versions
  - When installing a Kubernetes cluster, a particular version of Kubernetes is installed
  - Viewable via Kubectl get nodes
- Kubernetes versions are done via <Major>.<Minor>.<Patches>
- This is the standard software release pattern.
- Alpha and Beta versions used to test and integrate features into main stable releases.
- View Kubernetes GitHub repo's release page for details.
  - Packages contain all the required components of the same version.
  - For components like CoreDNS and ETCD, details on supported versions are provided as they are separate projects.

Reference Links

- https://github.com/kubernetes/kubernetes/releases
- https://github.com/kubernetes/community/blob/master/contributors/design-proposals/release/versioning.md
- https://github.com/kubernetes/community/blob/master/contributors/design-proposals/api-machinery/api-group.md
- https://blog.risingstack.com/the-history-of-kubernetes/
- https://kubernetes.io/docs/setup/version-skew-policy

Cluster Upgrade Process

- The kubernetes components don't all have to be at the same versions

    - No component should be at a version higher than the kube-api server
    - If Kube-API Server is version X (a minor release), then the following ranges apply for the other components for support level:
        - Controller manager: X-1
        - Kube-Scheduler: X-1
        - Kubelet: X-2
        - Kube-Proxy: X-2
        - Kubectl: X-1 - X+1
- At any point, Kubernetes only supports the 3 most recent minor releases e.g. 1.19 - 1.17
- It's better to **upgrade iteratively** over minor releases e.g. 1.17 - 1.18 and so on
- Upgrade process = ***Cluster-Dependent***
- If on a cloud provider, built-in functionality available
- If on kubeadm/manually created cluster, must use commands:
    - **kubeadm upgrade plan**
    - **kubeadm upgrade apply**
- Cluster upgrades involve two steps:
    - **Upgrade the master node**
        - All management components go down temporarily during the processes
        - Doesn't impact the current node workloads (only if you try to do anything with them)
    - **Upgrade the worker nodes**
        - Can be done all at once - Results in downtime
        - Can be done iteratively - Minimal downtime by draining nodes as they get upgraded one after another
        - Could also add new nodes with the most recent software versions
            - Proves especially inconvenient when on a cloud provider
- **Upgrading via Kubeadm:**
    - ***kubeadm upgrade plan***
        - Lists latest versions available
        - Components that must be upgraded manually
        - Command to upgrade kubeadm

    - Note: kubeadm itself must be upgraded first: **apt-get upgrade -y kubeadm=major.minor.patch_min-patch_max**
- Check upgrade success based on CLI output and kubectl get nodes
- If Kubelet is running on Master node, this must be upgraded next the master node and restart the service:
    - **apt-get upgrade -y kubelet=1.12.0-00**
    - **systemctl restart kubelet**
- Upgrading the worker nodes:
    - Use the drain command to stop and transfer the current workloads to other nodes: kubectl drain <node>, then upgrade the following for each node (***ssh*** into each one):
        - Kubeadm - ***apt-get upgrade -y kubeadm=major.minor.patch_min-patch_max***
        - Kubelet - ***apt-get upgrade -y kubelet=major.minor.patch_min-patch_max***
        - Node config: ***kubeadm upgrade node config --kubelet-version major.minor.patch***
    - Restart the service: **systemctl restart kubelet**
- Make sure to uncordon each node after each upgrade! ***kubectl uncordon <node>***
- Instructions Available Here: [Upgrade A Cluster | Kubernetes]

Labs - Cluster Upgrade Process

Kubectl Network Policies

- Consider an application comprised of the following servers with traffic throwing through each:
    - Web - Serves frontend to users
    - Application - Serves backend API
    - Database - Serves information utilised by API
- In this case:
    - Web user requests content from web server on port 80
    - Web server makes request to API
    - API makes request to DB and info to be given to user

- In terms of traffic, there are two to be considered:
    - Ingress - Incoming to resource
    - Egress - Outgoing from resource
    - In the case of the example above:
        - Web Server:
            - Ingress - 80
            - Egress - 5000
        - API
            - Ingress - 5000
            - Egress - 3306
        - DB:
            - Ingress - 3306
- Considering this from a Kubernetes perspective:
    - Each node, pod and service within a cluster has its own IP address
    - When working with networks in Kubernetes, it's expected that pods should be able to communicate with one another, regardless of the solution to the project i.e. No additional configuration should be required.
    - By default, Kubernetes has an "All Allow" rule -> Allows communication between any pod in the cluster.
- Applying this knowledge to the example earlier:
    - Each server runs on a pod
    - Services are introduced to allow communication between pods and the user
    - By default, each pods can communicate with one another
- In the event of wanting to prevent a pod from communicating with a particular other pod, such as in a production scenario, Rules can be applied to control what Ingress and Egress traffic an pod can allow - This is a Network Policy.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector: # Define pod(s) to apply network policy to
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  Ingress: # define pod associated with acceptable ingress traffic
  - 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: # define what pod(s) the pod the policy is being applied to can send communications to
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

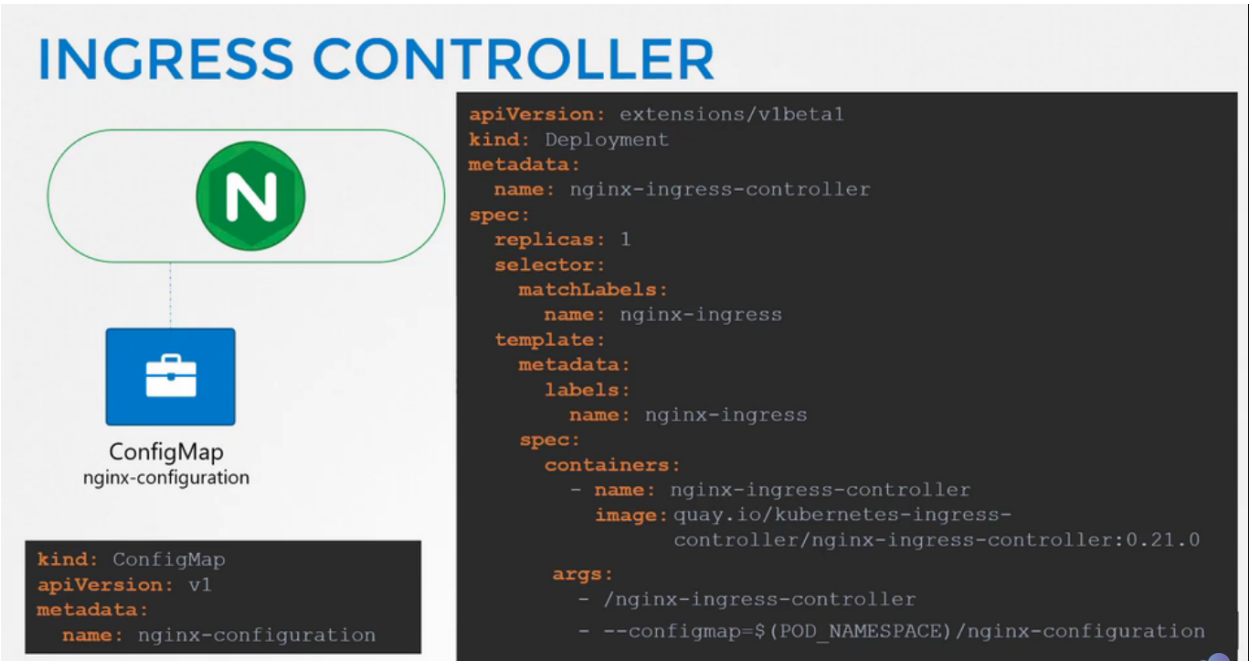
- To create the network policy, use kubectl apply as per:
    - kubectl apply -f <policy>.yaml
- Network policies are enforced by the network solution implemented on the cluster
- Many solutions support network policies, such as kube-router, Calico, and Weave-Net
    - Flannel doesn't support network policies - they can still be created, but will not be enforced.
- Selectors available:
    - podSelector
    - namespaceSelector
    - ipBlock

Ingress

- To understand the importance of Ingress service, it's best to consider an example:
    - Suppose an application is built into a Docker image and deployed as a pod via Kubernetes
    - Due to the application's nature, a database is required - a MySQL database is deployed as a pod and exposed as a ClusterIP service to allow for inter-pod communication.
    - To expose the application to external users, the application can be exposed as a NodePort service.
        - Application becomes available of one of the nodes ports
        - This is fine for small non-production apps
    - To access the URL, users go to ***http:<node-ip>:<nodeport>***
    - Increased demand to the application can be handled by increasing the number of replicasets & automatic service configuration for load balancing purposes
    - For production purposes, wouldn't want to provide the IP and port number for access
        - To resolve, configure a DNS server so a custom url can be used to direct users to the IP address
    - Similarly, users shouldn't have to provide ports when accessing the application
        - To resolve, integrate a proxy server between the DNS Server and cluster to proxy requests to the NodePort
        - The DNS is pointed to this Proxy Server
- The above steps would be applicable for hosting an app on an on-prem data center
- If working on a public cloud application, NodePort can be replaced by LoadBalancer
    - Kubernetes still carries out all the NodePort functionalities, but makes a request to the cloud platform to provision a load balancer configured to route the service ports of all the nodes.
    - The cloud provider's load balancer would have its own external IP which users request access via.
- Suppose as the application grows, you want to add another service.

    - Users should be able to access the new service via a new domain under the general URL
    - In the past, developers would develop the new service as a completely different application. To share the cluster resources, this would be deployed as a new deployment within the cluster; requiring a new service, load balancer, etc.
        - This can cause issues, as each of the load balancers add to the cloud bill
    - To map traffic between the two new services and load balancers, one would have to set up a new proxy or load balancer.
        - This would have to be repeated each time a new service is added, along with setting up SSL communication for the end-users.
        - SSL should be configured in one central location.
- The whole process outlined above has numerous issues. In addition to requiring proxy configuration per service, one must also consider cost - each additional LoadBalancer costs to start up.
    - Additionally, the configurations require extra management -> leads to extra difficulty.
- Kubernetes offers a central solution to this via Ingress resources.
    - Allows user access via a single URL
    - URL can be configured to route different services depending on URL parts / domains
    - SSL security can automatically be implemented via Ingress resources too.
    - Ingress - Essentially a layer 7 load balancer built-in to Kubernetes clusters, configurable via Kubernetes primitives
- Note: Even with Ingress resource(s) in place, one must still expose an application via a NodePort service or Load Balancer; this is a one-time configuration.
- Once exposed, all load-balancing, authentication, SSL and URL routing configurations are managed via the Ingress controller, the rules are defined by Ingress resources.
- Ingress controllers aren't set up with a Kubernetes cluster by default, one must be deployed such as:
    - GCE

    - Nginx
    - Traefik
- It should be noted that the load balancer isn't the only component of an Ingress Controller - Additional functionality is available for monitoring the cluster and resources for new ingress resources or definitions.
- To create, one needs to write a definition file:
    - First for an nginx ingress controller
    - As working with Nginx, need to configure options such as log path, SSL settings, etc.
    - To decouple this from the controller image, one should write a configmap definition file to be referenced.
        - Allows easier modification rather than editing a massive file.
    - Additionally, one needs an ingress service definition file to facilitate external communication.



```
    env:
      - name: POD_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
      - name: POD_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace

    ports:
      - name: http
        containerPort: 80
      - name: https
        containerPort: 443
```

- NodePort service should also be created to facilitate external communication and expose the ingress resource.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

- As mentioned previously, ingress controllers have additional functionality available for monitoring the cluster for ingress resources and applying configurations when changes are made.
- For the controller to do this, a service account must be created. With associated role and role bindings.
- Once an ingress controller is in place, can create ingress resources:

    - A set of rules and configurations applied to the ingress controller.
        - Example: could configure a route to forward all traffic to the application, or to a different set of applications based on a URL
        - Alternatively, could route based on DNS.
- As per, ingress resources are configured via a definition file:

```
apiVersion: extensions/v1beta1
Kind: Ingress
metadata:
  Name: Ingress-wear
Spec:
  backend:
    Service: wear-service
    servicePort: 80
```

- After applying the configuration using kubectl apply, can view ingress resources via kubectl get ingress.
- To route traffic in a conditional form, use ingress routes:
    - E.g. routing based on DNS
- Within each rule, can configure additional paths to rout to additional services / applications.
- To implement, using the 2-service application example:

```
apiVersion: extensions/v1beta1
Kind: Ingress
metadata:
  Name: Ingress-wear
Spec:
  Rules:
  - http:
      Paths: # array detailing paths for each url possibility
      - path: /user
        Backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
         backend:
           serviceName: watch-service
           servicePort: 80
```

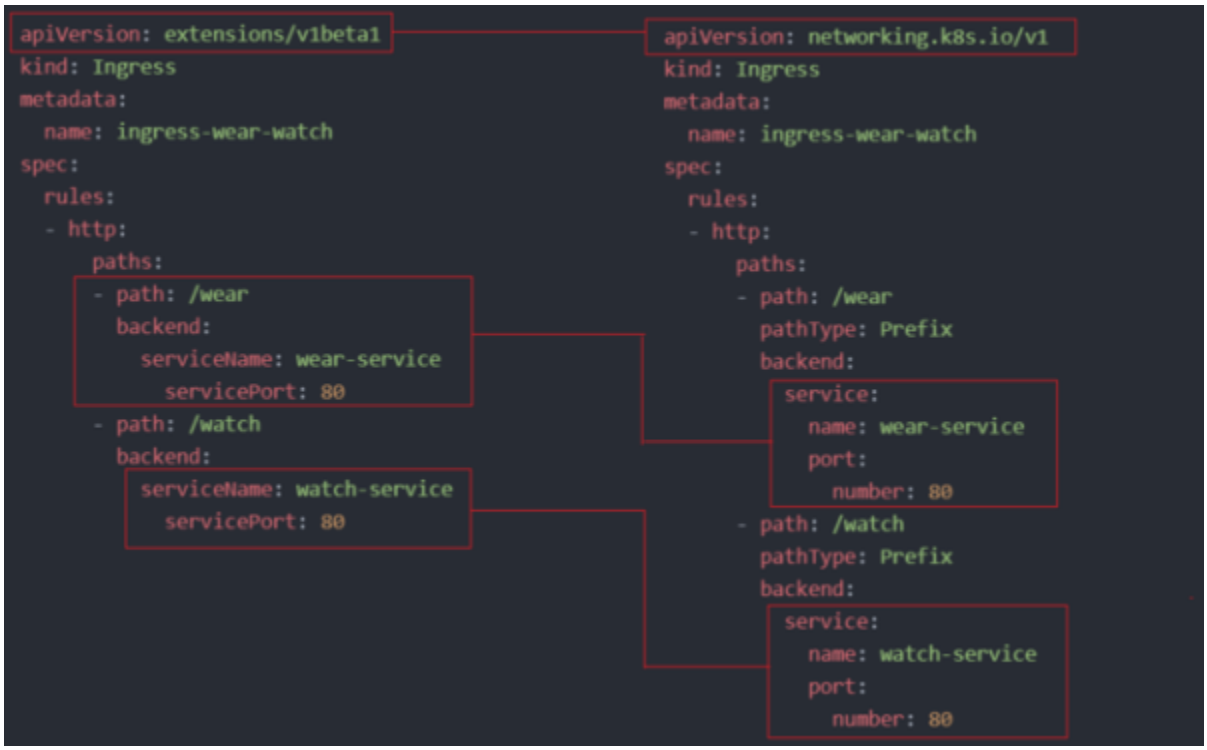
- Resource creatable via kubectl apply or create.
- To view detailed information:

  Kubectl describe ingress <ingress resource>
- Note: In the description, a default backend is described
    - In the event a user enters a path not matching any of the rules, they will be routed to the 'default backend' service
    - A default backend must be deployed for this however.
- If wanting to split traffic via domain name, fill out a definition file similar to:

```
apiVersion: extensions/v1beta1
Kind: Ingress
metadata:
  Name: Ingress-wear
Spec:
  Rules:
  - host: wear.my-online-store.com
    http:
     Paths: # array detailing paths for each url possibility
     - paths:
       Backend:
         serviceName: wear-service
         servicePort: 80
     -  host: watch.my-online-store.com
        paths:
         backend:
          serviceName: watch-service
          servicePort: 80
```

- When splitting by URL, had 1 rule and split the traffic by 2 parts.
- When splitting by hostname, use 2 rules with a path for each
- Note: If not specifying a host field, it will assume it to be a * and accept all incoming traffic without matching the hostname; this is acceptable for a single backend.

Latest Kubernetes versions:



- in k8s version 1.20+ we can create an Ingress resource from the imperative way like this:-
- Format - kubectl create ingress <ingress-name> --rule="host/path=service:port"
- Example - kubectl create ingress ingress-test --rule="wear.my-online-store.com/wear*=wear-service:80"
- Find more information and examples in the below reference link:-
    - https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-em-ingress-em-
- References:-
    - https://kubernetes.io/docs/concepts/services-networking/ingress
    - https://kubernetes.io/docs/concepts/services-networking/ingress/#path-types

Ingress - Annotations and Rewrite-Target

Different ingress controllers have different options that can be used to customise the way it works. NGINX Ingress controller has many options that

can be seen here. I would like to explain one such option that we will use in our labs. The Rewrite target option.

Our watch app displays the video streaming webpage at http://<watch-service>:<port>/

Our wear app displays the apparel webpage at http://<wear-service>:<port>/

We must configure Ingress to achieve the below. When user visits the URL on the left, his request should be forwarded internally to the URL on the right. Note that the /watch and /wear URL path are what we configure on the ingress controller so we can forwarded users to the appropriate application in the backend. The applications don't have this URL/Path configured on them:

http://<ingress-service>:<ingress-port>/watch –> http://<watch-service>:<port>/

http://<ingress-service>:<ingress-port>/wear –> http://<wear-service>:<port>/

Without the rewrite-target option, this is what would happen:

http://<ingress-service>:<ingress-port>/watch –>
http://<watch-service>:<port>/watch

http://<ingress-service>:<ingress-port>/wear –>
http://<wear-service>:<port>/wear

Notice watch and wear at the end of the target URLs. The target applications are not configured with /watch or /wear paths. They are different applications built specifically for their purpose, so they don't expect /watch or /wear in the URLs. And as such the requests would fail and throw a 404 not found error.

To fix that we want to "ReWrite" the URL when the request is passed on to the watch or wear applications. We don't want to pass in the same path that user typed in. So we specify the rewrite-target option. This rewrites the URL by replacing whatever is under rules->http->paths->path which happens to be

/pay in this case with the value in rewrite-target. This works just like a search and replace function.

For example: replace(path, rewrite-target)

In our case: replace("/path","/")

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  namespace: critical-space
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /pay
        backend:
          serviceName: pay-service
          servicePort: 8282
```

In another example given here, this could also be:

```
replace("/something(/|$)(.*)", "/$2")
```

```
apiVersion: extensions/v1beta1
kind: Ingress
```

```
metadata:
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
  name: rewrite
  namespace: default
spec:
  rules:
  - host: rewrite.bar.com
    http:
      paths:
      - backend:
          serviceName: http-svc
          servicePort: 80
        path: /something(/|$)(.*)
```

Docker Service Configuration

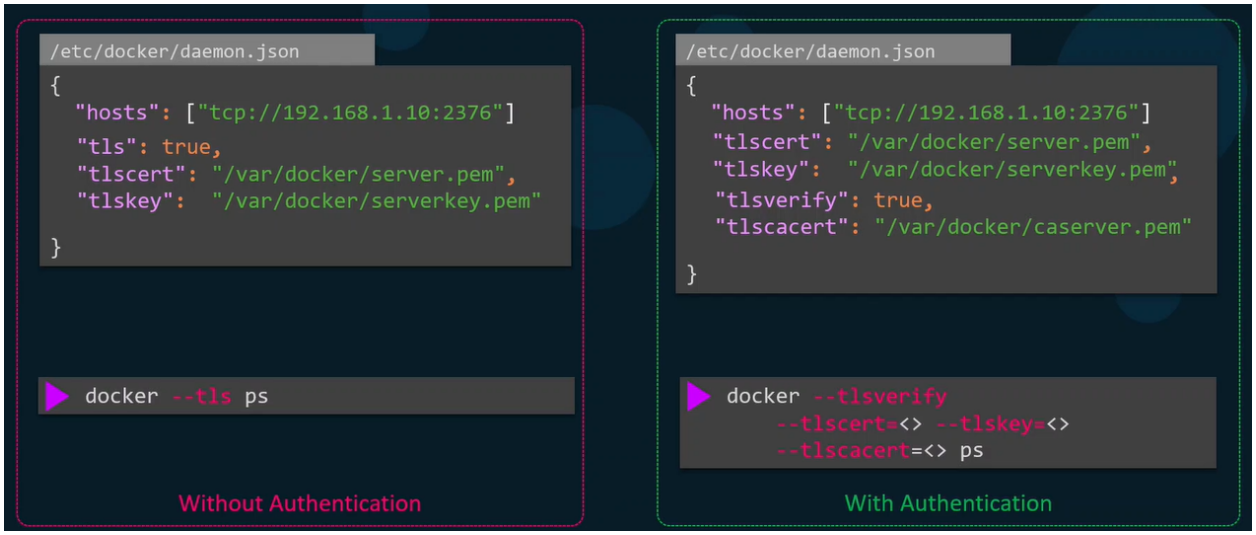
- Docker service can be managed via the systemctl command:
    - Start - start the service
    - Status - check the service status
    - Stop - stop the service
- The Docker Daemon can be started manually via running ***dockerd***
    - Usually done for troubleshooting or debugging purposes
    - Prints dockerd logs
    - Additional logs printable via the --debug flag

- The docker daemon starts and listens on an internal unix socket at /var/run/docker.sock
    - Unix Socket = Inter-Process Communication (IPC) mechanism used for communication between different processes on the same host
    - Implies the docker daemon is only intractable within the same host and docker CLI is only configured to interact with the docker daemon on this host.
- In the event you want to establish connection to the docker daemon from another host running the Docker CLI e.g. a Cloud VM running docker daemon.
    - Not set up by default.
    - One can instruct the daemon to listen on a TCP interface by adding a flag to the dockerd command:
      --host=tcp://<IP>:2375
    - The host at IP can now interact with the Docker Daemon using the Docker CLI, first by setting the environment variable
      DOCKER_HOST="tcp://<IP>:2375"
    - Note: This is disabled by default - For good reason! This is because by making the API server available on the internet, anyone can create and manage containers on the Daemon host - no security is set up by default.
    - Docker Daemon sets unencrypted communication by default, to set:
        - Create a pair of TLS certificates
        - Add the following flags to the dockerd command:
            - --tls=true
            - --tlscert=/var/docker/server.pem (path to certificate)
            - --tlskey=/var/docker/serverkey.pem (path to private key)
    - With TLS enabled, the standard port becomes 2376 - encrypted traffic; 2375 remains for unencrypted traffic.
- All the options specified in this section can be added to a configuration file for ease of use; typically located at /etc/docker/daemon.json
    - This must be created if it doesn't already exist; it's not included by default
    - Note: Hosts is an array i.e. ["host 1", "host 2", ...]
    - This configuration can be referenced when using the systemctl utility to start the Docker server.

Docker - Securing the Daemon

- The Docker Daemon / Engine / API Service needs to be secured appropriately, otherwise unauthorized users could:
    - Delete existing containers hosting applications
    - Delete volumes storing data
    - Run containers of their own e.g. bitcoin mining
    - Gain root access to the host system via a privileged container
        - Can lead to targeting of other systems in the infrastructure
- Reminder:
    - The Docker Daemon is by default exposed on the host only on a UNIX socket at /var/run/docker.sock
    - No one outside of the host can access the docker daemon by default
    - Note: The applications can still be accessed so long as the ports are published
- First Area of Security Consideration - Host:
    - Actions that can be taken include:
        - Disable Password-based authentication
        - Enable SSH key-based authentication
        - Determine which users need access to the servers
        - Disabling unused ports
    - External access needs to be configured only if absolutely necessary - achievable by adding to the “hosts” array in the docker daemon json configuration file.
        - Any hosts added must be private and only accessible within your organisation
        - TLS Certificates must also be configured for the server in this scenario, requiring:
            - A certificate authority
            - A TLS certificate
            - A TLS Key

        - TLS Configuration requires setting the port value for hosts in the config file to 2376 for encrypted TLS communication; as well as tls = true.
        - On the host accessing the docker daemon, the following must also be set as environment variables:
            - DOCKER_TLS=true
            - DOCKER_HOST="tcp://<IP>:2376"
- The configurations so far are acceptable but no authentication is enforced. To do so, certificate-based authentication can be enabled by copying the certificate authority to the docker daemon server.
    - The docker daemon config file can have the following added:
        - Tlsverify = true - enforces the need for clients to authenticate
        - Tlsecacert = "path/to/cacert.pem"
- To authenticate, clients need to have their own certificates signed by the CA, generated to give client.pem and clientkey.pem
    - Sharing these with the cacert on the host
- On the client side, these can be passed via --tlscert, --tlskey, --tlscacert flags or added to the ./docker folder on the system.
- Summary:
    - TLS alone only enforces encryption, TLS verify enforces authentication



30/10/2021

04 - System Hardening

Least Privilege Principle

- Airport Analogy
    - Consider an airport passenger, what procedure must they go through?
        - Baggage drop-off and ticket validation
        - Travel document verification
        - Security / Carry on check
        - Move to a specific departure gate; users can access other areas during this time, but some restrictions still apply.
    - This suggests that in security management for a system, there are multiple entities involved; each with their own permissions and privileges
    - Considering the privileges of each of the entities:
        - Traveller - Unrestricted access to public spaces in the airport after check-in and the designated gate
        - Baggage - Has access to the travellers documents and the airlines information
        - Security officers - can inspect the belongings of travellers in the area
        - Store employees - public and restricted areas
        - Boarding gate - access to the documents
        - Cleaners and cargo workers - access to public and certain restricted areas
        - Pilots / Flight crew - complete aircraft access and certain areas of the airport
            - They have access only to the plane(s) of their airlines
- In this case, each entity has the least / minimum amount of privileges required to perform their role.
- This principle is applicable to computer systems and K8s clusters.
- Kubernetes cluster security, via the least privilege principle, can be enforced via multiple options including:

    - Limit access to nodes
    - RBAC Access
    - Remove obsolete packages and services
    - Restrict network access
    - Restrict obsolete kernel modules
    - Identify and fix open ports
-

Minimize Host OS Footprint Intro

- Reducing the attack service
    - Achievable by keeping all systems in the cluster in a simple and consistent state
    - Any inconsistent or weak nodes could be attacked
        - Consistency can be enforced by Least Privilege Principle

Limit Node Access

- As standard practice, exposure to the internet of the control plane and nodes should be limited.
    - For self-hosted clusters or managed K8s clusters, this can be achieved via a private network; access can then be achieved by a VPN
    - If running in the cloud, control plane access may be unlikely
- If implementing a VPN solution is impossible, one can enable authorized networks via the infrastructure firewall
    - Allows/denies access from particular IP addresses
- Note: Attacks may not come from external sources, entities with network access within the cluster may allow internal attacks.
    - One must therefore consider restricting access within the cluster.
    - Example - SSH Access:
        - Administrators require it
        - Developers do not require it usually and therefore they should be restricted.
        - End users shouldn't have access too.

- Account Management - There are 4 types of accounts:
    - User accounts - Any individuals needing access to the system e.g. developers system administrators
    - Superuser account - Root Account, has complete access and control over the system
    - System Accounts - Created during the system development, used by software such as SSH and Mail
    - Service Accounts - Similar to system accounts, created when services are installed on Linux e.g. Nginx
- Viewing user details - Commands:
    - Id - provides details regarding user and group id
    - Who - lists who is currently logged in
    - Last - last time users logged into the system
- Access control files:
    - All stored under **/etc/** folder
        - **/passwd** - contains basic information about system users, including user id and default directory
        - **/shadow** - contains the passwords for users, stored in hash format
        - **/group** - stores information about all user groups in the system
- The above commands and configuration files should be used to investigate user permissions and access; limiting as appropriate based on least privilege principle.
    - Disable user account by setting default shell to a nologin shell:
      Usermod -s /bin/nologin <username>
    - Delete user - **userdel <username>**
    - Delete user and remove from group - **userdel <username> <groupname>**

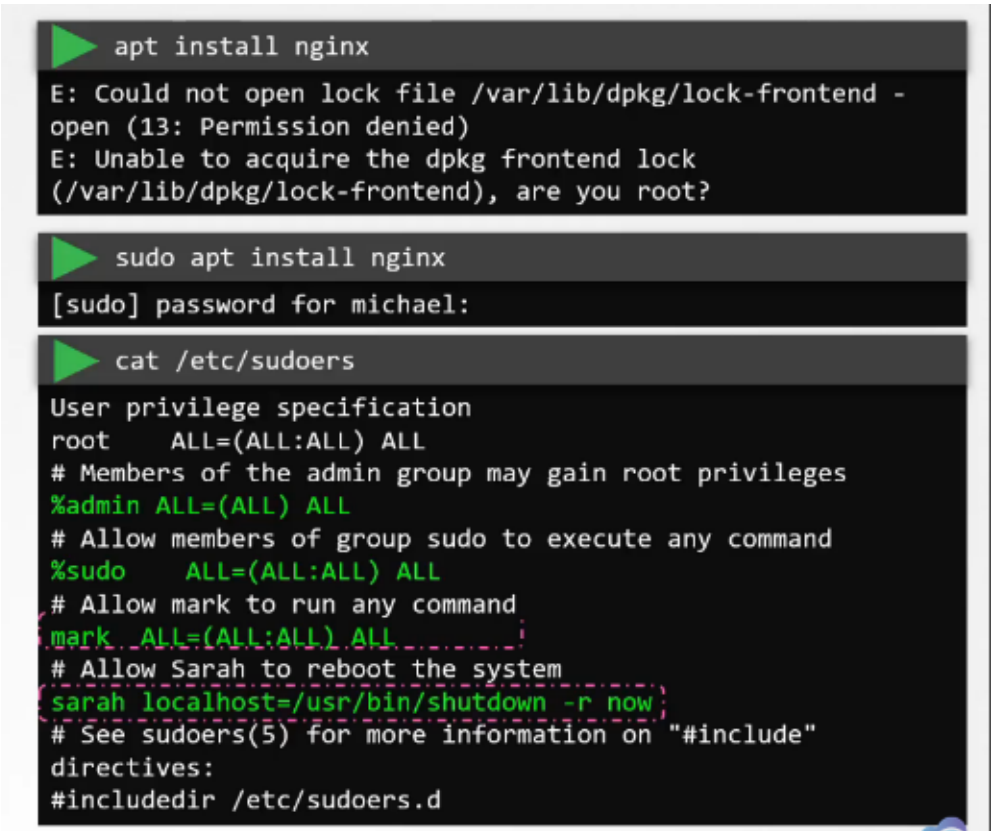
SSH Hardening

- SSH used for logging into and executing commands for remote servers
- General access via any of the following commands:
  Ssh <hostname or IP address>
  Ssh <user>@<hostname or IP address>
  Ssh -l <user> <hostname or IP address>

- Remote server must have an SSH service running and port 22 opened for communication
- Additionally, a valid username and password for the user should be created on the remote server; or an SSH key.
- To maximise security for access:
    - Generate key pair (public and private key) on the client system
    - Public key shared with remote server
    - Creation: ***ssh-keygen -t rsa***
        - Enter passphrase = optional, but enhances security
        - Public key and private key stored at /home/user/.ssh/id_rsa.pub and /home/user/.ssh/id_rsa respectively
    - Copy public key to remote server: **ssh-copy-id <username>@<hostname>**
        - Password required for authentication on the server; shouldn't be required after key copying
    - On the remote server, the keys should be stored at **/home/user/.ssh/authorized_keys**
- Hardening the SSH service:
    - On the remote server; can configure the SSH service to enhance security.
    - Possible actions:
        - Disable SSH for root account - no one can log into the system via the root account, only their personal user or system accounts
            - In line with the principle of least privilege
        - Requires updating of the ssh config file, located at **/etc/ssh/sshd_config -> set PermitRootLogin** to no
        - Could also disable password-based services if using SSH key based authentication -> set **PasswordAuthentication** to no
    - Once changes are applied, restart the service: **systemctl restart sshd**
- **Additional information available via section 5.2 of the CIS Benchmarks https://www.cisecurity.org/cis-benchmarks/**

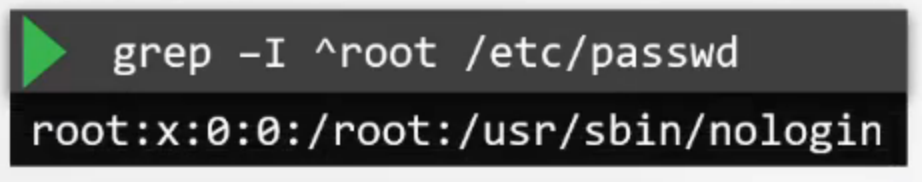
Privilege Escalation in Linux

- Following disabling of root user access via ssh, need to consider when users may need root / administrative access credentials; this is provided by the sudo functionality (prefixed before any command)
- Default configuration for sudo is under **/etc/sudoers;** it can only be edited via visudo
- The sudoers file defines user privileges for sudo



- Administrators can give granular permissions
    - User mark has all permissions
    - User sarah has sudo permissions only for restarting the system
- Any sudo commands ran are executed in the users local shell rather than root shell -> can eliminate the need for ever logging in as a root user directly

- This can be done by editing the /etc/passwd for root user like below



- Sudoers syntax
    - Anything beginning with # or $ is a comment
    - Content split into fields:
        - User or group the privilege(s) are to be granted to,
            - Groups are prefixed with %
        - Host the user can make use of the privileges on
            - By default, set to all, applies to localhost in most cases
        - Users and Groups the user in field 1 can run the command as
        - Command(s) runnable by the user(s) via sudo

31/10/2021

Remove Obsolete Packages and Services

- In general, systems should be kept as lean as possible, making sure that only the necessary software is installed and they are kept updated to address security fixes
- This can be applied for standalone systems, Kubernetes systems, etc.
- Any software that isn't used should be uninstalled for general reasons including:
    - Increased complexity of the system - one more package to maintain
    - It may remain unused and take up unwanted space
    - If not maintained, additional security vulnerabilities may arise that could be taken advantage of
- Removing services:
    - Services start applications upon Linux system booting
    - Managed by the Systemd tool and the systemctl utility
    - E.g. ***systemctl status <service>***
        - Provides details regarding the service, including where its configuration file(s) are located

    - As with packages, only necessary services should be kept running or available on the system
        - To view the system services: ***systemctl list-units --type service***
        - To remove a service: ***systemctl stop <service>***
- Once services for unwanted packages are stopped, one can remove the associated package(s) via ***apt remove <package name>***
- Additional information available in section 2 of the CIS Benchmarks.

Restrict Kernel Modules

- Linux kernel modules has a modular design - its functionality can be extended via the addition of extra modules
- Example - Hardware support for video drivers via graphics cards
- Modules can be added manually - **modprobe <module name>**
- List modules in the Kernel - **lsmod**
- Note: When kubernetes workloads are running, non-root processes on pods may cause network-related modules to be installed on the Kernel - This can lead to vulnerabilities exploited by attackers.
- To mitigate, modules on cluster nodes can be blacklisted - achievable by creating a .conf file under **/etc/modprobe.d**
    - Any modules to be blacklisted should be included as an entry via: **blacklist <modulename>**
    - Upon entry addition, the system should be restarted and tests should be ran to see if the module is still running:
      Shutdown -r now
      Lsmod | grep <modulename>
- Example modules that should be blacklisted - **sctp** and **dccp**
- Additional information available in section 3.4 of the CIS Benchmarks

Identify and Disable Open Ports

- Several processes and services bind themselves to a network port on the system - an addressable location in the OS to allow for segregation of network traffic.
- Example - TCP port 22 is only used for SSH processes

- To understand if a port is in used and listening for a connection request:
    - ***natstat -an | grep -w LISTEN***
- To understand what each service or port is being used for, can check in the services file under /etc
    - Example: **cat /etc/services | grep -w 53**
- This begs the question, what ports should be open for nodes on the Kubenetes cluster? Answer: Review the documentation!
- https://kubernetes/doc.setup/production-environment/tools/production-environment/tools/kubeadm/install-kubeadm/#check-required-ports

Minimize IAM Roles

- As discussed in the section ***Limit Node Access,*** it's never a good idea to use the Root account to perform daily tasks, SSH hardening can prevent root access and force users to using their own user accounts.
- Root accounts are equivalent to Admin accounts in Windows and Root Accounts in public cloud platforms e.g. AWS
- Note: AWS used as an example for this section but topics apply
    - 1st account (Root account) created - User can log into management credentials
        - Any and all functions can be carried out by this account on the management account
    - Root account, in line with the least privilege principle, should be used to create new users and assign them the appropriate permissions
    - The credentials for the root account should be saved and secured as appropriate
- When a new user is created, the least privilege is assigned depending on the associated IAM (Identity Access Management) policy.
    - E.g. developers have ability to create EC2 instances, but can only view the S3 Buckets
    - For further ease of assignment, users could be added to particular role groups or IAM groups - an IAM policy can then be attached to the group and assigned automatically.

- For resources and services, by default, no permissions are allowed. This cannot be achieved via IAM policies, roles must be developed.
    - Allows secure access to an AWS Resource(s)
    - Example - Allow EC2 Instance Access to an S3 Bucket.
- Note: Programmatic access can be configured, but this is typically less secure than via IAM methods.

27/11/2021

Minimize External Access to the Network

- To view the ports that are open and listening for requests on your system, utilize the netstat command:
  netstat -an | grep -w LISTEN
- By default, many of these ports can accept connections from any other device on the network - undesirable via the principle of least privilege
- In production environments, with multiple clients and servers involved, it's beyond important to implement network security to only allow or restrict access to various services
- Security can be provided via network-wide or external firewalls e.g.
    - Cisco ASA
    - Juniper NGFW
    - Barracuda NGFW
    - Fortinet
- Alternatively, firewalls can be applied per server.

UFW Firewall Basics

- UFW = Uncomplicated Firewall
- Considering an application setup where one needs to restrict access to an Ubuntu Server - call it app01
- The server should only accept SSH connections on Port 22 from a particular IP address
    - Additionally, the server runs a web server on http port 80

    - All other users in a particular IP range should be allowed
- Any other unused ports should be closed
- To help with this, can use the netfilter CLI tool and IPTables
    - IPTables - Most common for Linux, but requires a steep learning curve
    - UFW = Alternative that provides a frontend for IPTables
- On the machine:

  Netstat -an | grep -w LISTEN
- The UFW tool can be installed via the following:
    - **Apt-get update**
    - **Apt-get install ufw**
    - **Systemctl enable ufw**
    - **Systemctl start ufw**
- UFW Rules:
    - Check the UFW Status: **ufw status**
    - In example, want to allow 80 and 22 from particular sources, but no outgoing ports need closing:
        - Allow outgoing by default:
            - **Ufw allow outgoing**
        - Deny ingress by default (before making exceptions)
            - Ufw default deny incoming
        - Allow from particular IP address on particular port e.g. SSH 22
            - **ufw allow from 172.16.238.5 to any port 22 proto tcp**
            - Allows inbound connections from the IP address defined to port 22 on any reachable IP address on app01
            - Ufw allow from **172.16.238.5 to any port 80 proto tcp**
                - **Allows connections on port 80 from IP address**
            - **Ufw allow from 172.16.100.0/28 to any port 80 proto tcp**
                - **Allows access on port 80 from IPs in any range**
            - **Ufw deny 8080**
                - **Denies all access on port 8080 (not needed if denying access by default)**
        - To enable: ufw enable
        - Ufw status to check implementation

- To delete rule: **ufw delete <rule> or <rule number>**

Linux Syscalls

- To understand how Syscalls are made, need to understand how a process runs:
  - Kernel - Major OS component that interfaces between hardware and processes
  - Kernel can be split into Kernel and User Spaces
    - Kernel Space - OS-related applications and software e.g. drivers
    - User Space - Where applications are ran
  - When a program runs in the user space, suppose one that wants to write data to a file:
    - Applications make system calls to the Kernel Space for tasks e.g.:
      - Open a file
      - Write to a file
      - Define a variable
      - open()
      - close()
    - Tracing Syscalls:
      - **Which strace**:
        - Installed by default
        - Used to inspect / provide output of the syscalls made when running a command
      - Output presented in form of:
        - syscall(<path to executable file>, [arg 1, arg 2, ...], ...)
      - To trace the syscalls of a running process:
        - Get process ID (PID) - pid <process>
        - Get Syscalls: strace -p PID
    - Note: the -c flag can be used for more detailed outputs.

AquaSec Tracee

- Open-source tool used for tracing container syscalls

- Makes use of EBPF to trace the system at runtime without interfering with the Kernel:
    - extended Berkeley Packet filter
- Tracee can be easily ran as a container:
- Prerequisites:
    - As ran as a container, it runs the EBFP program and stores it at /tmp/tracee by default
    - To ensure built once and usable for successive runs, a set number of bind mounts are required:

Mount	Purpose
/tmp/tracee	Default workspace
/lib/modules	Kernel Header Access
/usr/src	Kernel Header Access

    - Addtionally, due to tracing syscalls, Tracee requires privileged capabilities:
- Using Tracee:
    - To observe syscalls generated by a single command:
      **Docker run --name tracee --rm --privileged --pid=host **
      **-v /lib/modules/:/lib/modules/:ro -v /usr/src:/usr/src:ro **
      -v /tmp/tracee:/tmp/tracee aqasec/tracee:0.4.0 --trace comm=ls
    - Returns syscalls made during the run of the ls command
    - To observe syscalls for all new processes on the host:
      **Docker run --name tracee --rm --privileged --pid=host **
      **-v /lib/modules/:/lib/modules/:ro -v /usr/src:/usr/src:ro **
      -v /tmp/tracee:/tmp/tracee aqasec/tracee:0.4.0 --trace pid=new
    - For new containers:
      **Docker run --name tracee --rm --privileged --pid=host **
      **-v /lib/modules/:/lib/modules/:ro -v /usr/src:/usr/src:ro **
      -v /tmp/tracee:/tmp/tracee aqasec/tracee:0.4.0 --trace container=new

Restricting Syscalls Using Seccomp

- Previously seen how to view syscalls in an os.
- In practice, ~435 syscalls are available in Linux, all can be used by applications in the user space.
- In reality, applications wouldn't need to use anywhere near as many syscalls - application should only be able to use the required syscalls for their application
- By default, the Linux Kernel allows any syscall to be invoked by programs in the user space - increasing attack surface
- To resolve, can utilise seccomp
    - Secure Computing - A linux kernel-native feature designed to constrain applications to only make the required syscalls
    - To check if host supports it, look in the boot config file:
      Grep -i seccomp /boot/config-$(uname -r)
    - If CONFIG_SECCOMP = y -> seccomp is supported
- To demonstrate how applications can be finetuned, consider running a container e.g. docker/whalesay:
    - Docker run docker/whalesay cowsay hello!
    - Knowing the app works - can run the container and exec into it:
      Docker run -it --rm docker/whalesay /bin/sh
        - If users wanted to change the time for example, it would not be allowed
            - Process running is shell/bin/sh
            - Inspecting the process id -> PID = 1
            - Inspecting the PID can check via seccomp:
                - Grep seccomp /prc/1/status
                - If value = 2 -> seccomp enabled:
                - Modes:
                    - 0 = Disabled
                    - 1 = Strict -> Blocks all syscalls except read, write, exit, rt_sigreturn
                    - 2 => Selectively filters syscalls
        - How was the seccomp filter applied? What restrictions has it got applied?

- Docker has seccomp enabled by default and restricts ~60 of the 435 syscalls
- Defined by json file(s)
    - Defines architectures for files
    - Syscall arrays -> names and actions (allow or deny)
    - Default action -> what to do with syscalls not defined in the syscall array
- Json files can act as whitelists or blacklists
    - Whitelists can be very restrictive as any which you do want to run have to be added
    - Blacklists -> allows all by default bar any in the array; more open than whitelists but more susceptible to attacks
    - Default docker seccomp profiles block calls such as reboot, mount and unmount, clock time managements
    - Default seccomp profiles are good but custom files are better for particular scenarios
- To utilise a custom seccomp profile, can add associated flag to the docker run command i.e.:
  Docker run -it --rm --security-opt seccomp=/path/to/file.json \ docker/whalesay /bin/sh
- Seccomp can be disabled for containers completely by setting the flag to **--security-opt seccomp=unconfined**
    - SHOULD NEVER BE USED

Implementing Seccomp in Kubernetes

- By default, Docker supplies its own default seccomp profile in containers in mode 2
- When running in kubernetes, the number of blocked syscalls and seccomp enablement may be different -> seccomp may not be enabled by default in Kubernetes (as of v1.20)
- To implement seccomp - use a pod definition file, to apply, need to add an appropriate field in the securityContext area:

securityContext:

Seccompprofile:

Type: RuntimeDefault

- Note: When adding the profile in this manner, it is advised to add to the container's securityContext field a disabling of privilege escalation i.e.:
  Containers:

  - securityContext:

  allowPrivilegeEscalation: false
- This helps the application run ONLY with the Syscalls it requires for the process
- Note: Type can be set to Unconfined, but this is not recommended and is the default profile
- For custom profiles in the pod security context::
  securityContext:

  Seccompprofile:

  Type: Localhost

  localhostProfile: <path to file>
- Note: The profile path must be relative to the default seccomp file in **/var/lib/kubelet/seccomp**
- Example:
  - Creating an audit seccomp profile with defaultAction **SCMP_ACT_LOG**
  - Syscalls from the container will be stored in /var/log/syscall
    - Logs user id, process ID, etc about the processes used
  - To help map the syscall numbers to syscalls is to check the following:
    Grep -w 35 /usr/include/asm/unistd_64.sh
  - Where 35 can be replaced with the PID
- Tracee can also be used for this as discussed previously
- Consider another seccomp profile that rejects by default (defaultAction = SCMP_ACT_ERRNO) => any pod created with this cannot run as all Syscalls are blocked by default
- Once the Syscalls are analysed and audited, users can identify the allowed syscalls and add them to a custom seccomp profile => recommended to block all by default and allow only ones needed.

- Note: A custom seccomp profile won't be required to be created from scratch, but a template one may need to be attached to a pod/deployment etc.

29/11/2021

AppArmor

- Although Seccomp works well to restrict the syscalls a container makes (or pods/objects in Kubernetes), it cannot be used to restrict a programs access to objects like files and directories.
- Apparmor = Linux Security module, used to confine a program to a limited set of resources
    - Installed by default for most distributions
    - Use **systemctl status apparmor** to check status
    - To use, the associated Kernel module must first be loaded on all nodes where containers can run
        - Check enabled file under: **/sys/module/apparmor/parameters/enabled**

          Shows Y or N depending on enablement
    - Similar to Seccomp, apparmor is applied via profiles, which must be loaded into the Kernel; checkable via **/sys/kernel/security/apparmor/profiles**
    - Profiles are simple text files that determines what capabilities are allowed or restricted

Example:

apparmor-deny-write

```
profile apparmor-deny-write flags=(attach_disconnected) {
  file,
  # Deny all file writes.
  deny /** w,
}
```

- Above contains 2 rules:

- File -> allow file system -> allows complete access to filesystem
- Deny rule -> denies write access to all files under root system including subdirectories

Example 2 - Deny write access to only files under a particular subdirectory:

apparmor-deny-proc-write

```
profile apparmor-deny-proc-write flags=(attach_disconnected) {
  file,
  # Deny all file writes to /proc.
  deny /proc/* w,
}
```

- Apparmor profiles can be created via various tools
- The status of the profiles loaded can be checked using **aa-status**
- Profiles can be loaded in 3 modes:
  - Enforce - Apparmor will enforce the rules on any application that fits the profile
  - Complain - Apparmor will allow the actions without restrictions but the actions are logged as events
  - Unconfined - Any tasks are allowed and no logging occurs

Creating AppArmor Profiles

Example Script used in lecture - creates a directory and writes a file to said directory, the event of creation is logged in “create.log”

add_data.sh

```
#!/bin/bash
data_directory=/opt/app/data
mkdir -p ${data_directory}
echo "=>  File created at `date`" | tee ${data_directory} create.log
```

```
./add_data.sh
=>  File created at Mon Mar 12 03:29:22 UTC 2021
```

```
cat /opt/app/data/create.log
=>  File created at Mon Mar 12 03:29:22 UTC 2021
```

- Tools provided by apparmor can be used to facilitate the creation of apparmor profiles
- To install the utilities for apparmor: **apt-get install -y apparmor-utils**
- Once done, can generate a profile via:
  aa-genprof /path/to/script.sh
    - This prepares apparmor to develop an accurate profile by scanning the system during the application's runnin
    - Once the application has run in a separate window, in the original window, enter "S" to scan the system for AppArmor events
        - This will generate a series of questions that the user must answer to help develop the profile
            - E.g. for the example above, need to think about allowing the script access to the mkdir command or the tee command
    - Each event is reviewed on severity level of 1-10 (10 = most severe)
    - Resultant actions can be allowed/denied/inherited etc as required
    - The new profile can be viewed and validated by aa-status
    - The profiles are stored under **/etc/apparmor.d/**
- It's often advised to rerun the application with slightly different parameters to check the enforcement of the apparmor profile.

- To load a profile:
    - **Apparmor_parser path/to/profile**
        - No output = success
- Disable profile:
    - **apparmor_parser -R path/to/profile**
        - No output = success
    - Need to create a symlink to the profile and the apparmor disable directory:
        - **Ln -s /path/to/profile /etc/apparmor.d/disable/**

AppArmor in Kubernetes

- AppArmor can also be used to restrict the functions of containers orchestrated by Kubernetes
- At the time of writing / when the recording was done, Kubernetes support for it was in beta
- **Prerquisites:**
    - The AppArmor Kernel module must be enabled on all nodes in the K8s cluster where pods are expected to run
    - AppArmor profiles must be loaded in the Kernel
    - The container runtime should support AppArmor - Generally expected, Docker know to support.

Example:

ubuntu-sleeper.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
spec:
  containers:
  - name: hello
    image: ubuntu
    command: [ "sh", "-c", "echo 'Sleeping for an hour!' && sleep 1h" ]
```

apparmor-deny-write

```
profile apparmor-deny-write flags=(attach_disconnected) {
  file,
  # Deny all file writes.
  deny /** w,
}
```

- Ensure that the profile is loaded on all the nodes with aa-status if the profile is listed in the required director
- As the support is only in beta, need to link it by annotations:

  Annotations:

  container.apparmor.security.beta.kubernetes.io/<container_name>: localhost/<profile name>
- When the pod is created, you can inspect and test the pods capabilities to test the profile enforcement.

Linux Capabilities

- Previously seen that even when running a container with an unconfined seccomp profile, it couldn't be manipulated
- The same is applicable to Kubernetes even though it doesn't use seccomp by default
- To understand this, need to review how processes run in Linux:
  - Processes are separated into privileged and unprivileged (for kernel versions less than 2.2)
  - For 2.2 onwards, processes are split into capabilities:
    - Privileged processes possess a number of capabilities e.g.:

        - CAP_CHOWN
        - CAP_NETMODE
        - CAP_SYS_BOOT - Allows reboots
    - To check the required capabilities for a command:
        - getcap /usr/bin/ping for example
    - For a process:
        - Get the PID: ps -ef | grep /usr/sbin/sshd | grep -v grep
        - Getpcaps <PID>
- Pivoting back to the pod in the example, the reason the certain commands like the change time couldn't work, the operation wasn't permitted
    - Even if ran as a root user, the container is only started with limited capabilities (14 if Docker = runtime)
- Capabilities for a container can be managed at the container's security context:

ubuntu-sleeper.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
spec:
  containers:
  - name: ubuntu-sleeper
    image: ubuntu
    command: [ "sleep", "1000" ]
    securityContext:
      capabilities:
        add: ["SYS_TIME"]
```

- To drop a capability, add drop: ["Capability 1", "Capability 2", ...]

30/11/2021

05 - Minimize Microservice Vulnerabilities

Security Contexts

- When running docker containers, can specify security standards such as the ID of the user to run the container
- The same security standards can be applied to pods and their associated containers
- Configurations applied a pod level will apply to all containers within
- Any container-level security will override pod-level security
- To add security contexts, add securityContext to either or both the POD and Container specs; where user IDs and capabilities can be set
    - Note: Capabilities are only supported at container level

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
      - name: ubuntu
        image: ubuntu
        command: ["sleep", "3600"]
        securityContext:
          runAsUser: 1000
          capabilities:
              add: ["MAC_ADMIN"]
```

Admission Controllers

- Commands are typically ran using the kubectl utility, these commands are sent to the kubeapi server and applied accordingly
    - To determine the validity of the command, it goes through an authentication process via certificates
    - Any kubectl commands come from users with a kubeconfig file containing the certificates required
    - The determination of whether the process has permission to carry the task out is handled by RBAC authorization
        - Kubernetes Roles are used to support this.
- With RBAC, restrictions can be placed on resources for:
    - Operation-wide restrictions
    - Specific operation restrictions e.g. create pod of specific names
    - Namespace-scoped restrictions
- What happens if you want to go beyond this? For example:
    - Allow images from a specific registry
    - Only run as a particular user
    - Allow specific capabilities
    - Constrain the metadata to include specific information
- The above is handled by Admission controllers
    - Various pre-built admission controllers come with K8s:
        - AlwaysPullImages
        - DefaultStorageClass
        - EventRateLimit
        - NamespaceExists
- Example - NamespaceExists:
    - If creating a pod in a namespace that doesn't exist:
        - The request is authenticated and authorized
        - The request is then denied as the admission controller acknowledges that the namespace doesn't exist -> Request is denied
- To view admission controllers enabled by default:
  Kube-apiserver -h | grep enable-admission-plugins

    - Note: For kubeadm setups, this must be run from the kube-apiserver control plane using kubectl
- Admission controllers can either be added to the .service file or the .yaml manifest depending on the setup:

kube-apiserver.service

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --runtime-config=api/all \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --v=2
  --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision
```

- /etc/kubernetes/manifests/kube-apiserver.yaml

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.107
    - --allow-privileged=true
    - --enable-bootstrap-token-auth=true
    - --enable-admission-plugins=NodeRestriction,NamespaceAutoPrc
    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
```

    - The 2nd version is applicable for a Kubeadm setup
- To disable a plugin use --disable-admission-plugins=<plugin1>,<plugin2>, ...
- Note: NamespaceAutoProvision is not enabled by default - it can be enabled by the above menthod

Validating and Mutating Admission Controllers

- Validating Admission Controller - Allows or Denies a request depending on the controllers functionality/conditions
    - Example: NamespaceExists
- Mutating Admission Controller: If an object is to be created and a required parameter isn't specified, the object is modified to use the default value prior to creation
    - Example: DefaultStorageClass
- Note: Certain admission controllers can do both mutation and validation operations
- Typically, mutation admission controllers are called first, followed by validation controllers

- Many admission controllers come pre-packaged with Kubernetes, but could also want custom controllers:
    - To support custom admission controllers, Kubernetes has 2 available for use:
        - MutatingAdmissionWebhook
        - ValidatingAdmissionWebhook
    - Webhooks can be configured to point to servers internal or external to the cluster
        - Servers will have their own admission controller webhook services running the custom logic
        - Once all the built-in controllers are managed, the webhook is hit to call to the webhook server by passing a JSON object regarding the request
        - The admission webhook server then responds with an admissionreview object detailing the response
- To set up, the admission webhook server must be setup, then the admission controller should be setup via a webhook configuration object
    - The server can be deployed as an api server in any programming language desired e.g. Go, Python, the only requirement is that it must be able to accept and handle the requests
        - Can have a validate and mutate call
    - Note: For exam purposes, need to only understand the functionality of the webhook server, not the actual code
- The webook server can be ran in whatever manner desired e.g. a server, or a deployment in kubernetes
    - Latter requires it to be exposed as a service for access
- The webhook configuration object then needs to be created (validating example follows):
- Each configuration object contains the following:
    - Name - id for server
    - Clientconfig - determines how the webhook server should be contacted - via URL or service name
        - Note: for service-based configuration, communication needs to be authenticated via a CA, so a caBundle needs to be provided

- Rules
  - Determines when the webhook server needs to be called i.e. for what sort of requests should invoke the call to the webhook server
  - Attributes detailed include API Groups, namespaces, and resources.

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
   name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  clientConfig:
      service:
        namespace: "webhook-namespace"
        name: "webhook-service"
      caBundle: "Ci0tLS0tQk.....tLS0K"
  rules:
    - apiGroups:   [""]
      apiVersions: ["v1"]
      operations:  ["CREATE"]
      resources:   ["pods"]
      scope:       "Namespaced"
```

06/12/2021

Pod Security Policies

- When developing pods, there may be configurations you wish to prevent users from using / applying on the cluster

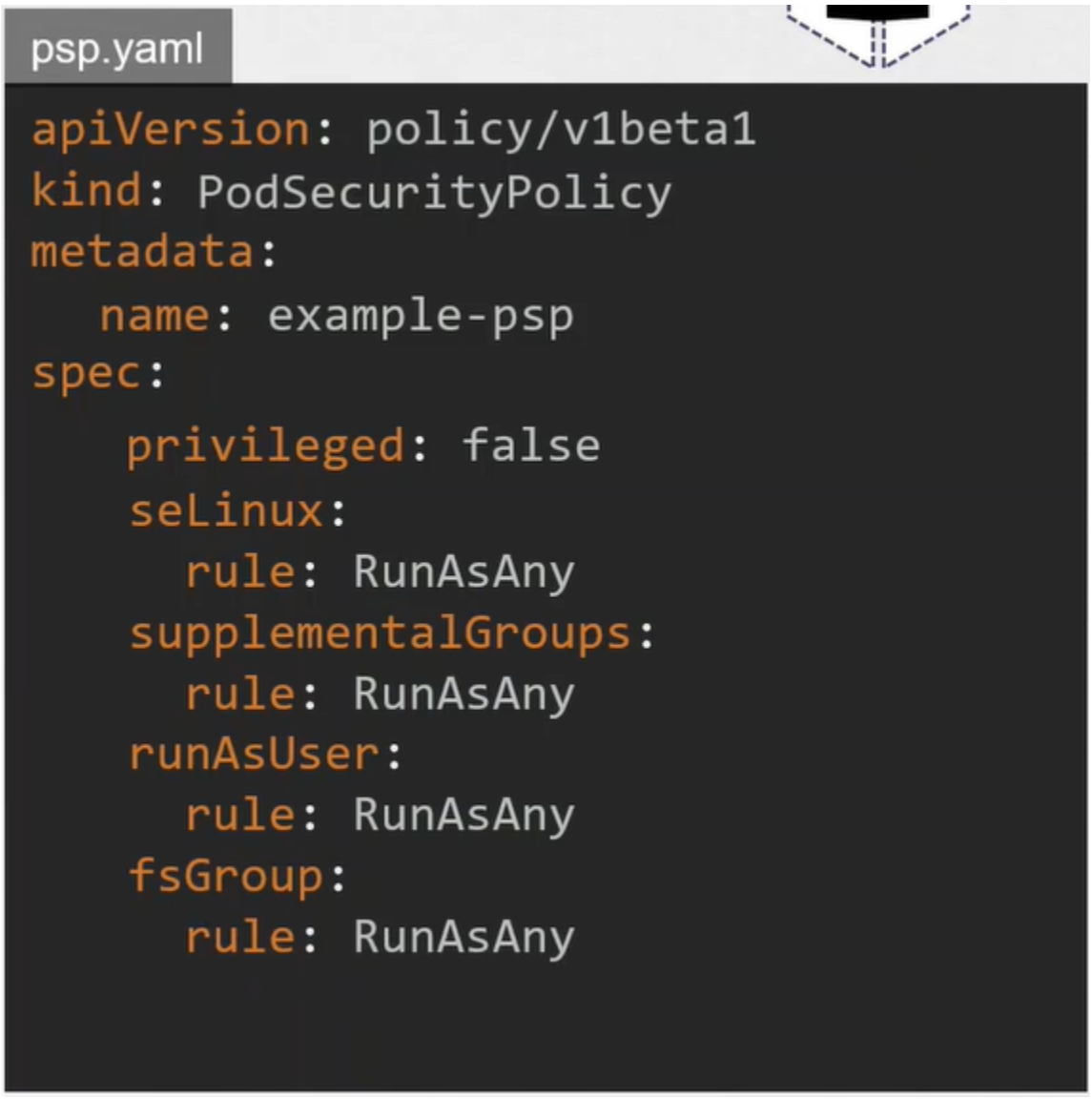
    - Examples:
        - Prevent container from having root access to the underlying system
        - Prevent running as root user
        - Prevent certain capabilities
    - These restrictions can be enforced by Pod Security Policies
- There is a pod security policy Admission controller that comes as part of Kubernetes by default, though it is not enabled by default.
    - Check if disabled by:: **kube-apiserver -h | grep enable-admission-plugins**
- It can be enabled by updating the --enable-admission-plugins flag with the kube-apiserver.service or kube-apiserver.yaml files:

/etc/kubernetes/manifests/kube-apiserver.yaml

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.107
    - --allow-privileged=true
    - --enable-bootstrap-token-auth=true
    - --enable-admission-plugins=PodSecurityPolicy
    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
```

- Once enabled, one can create a Pod Security Policy Object, outlining the requirements / restrictions. An example of restricting running as privileged user follows:

psp.yaml



```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example-psp
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
```

- Note:
    - seLinux, supplementalGroups, runAsUser and fsGroup are all mandatory fields.
- Once deployed, any pod definition files will be checked against the pod security policy.
- **Note:** Need to ensure that the security policy api must be authenticated / authorized for the admission controller to work.
    - This can be achieved via RBAC
    - Every pod has a serviceAccount associated with it (default if not specified)

- Can therefore create a role and bind the serviceAccount to allow access to the pod security policy api

psp-example-role.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: psp-example-role
rules:
- apiGroups: ["policy"]
  resources: ["podsecuritypolicies"]
  resourceNames: [”example-psp"]
  verbs: ["use"]
```

psp-example-rolebinding.yaml

```yaml
apiVersion: rbac.authorization.k8s.i
kind: RoleBinding
metadata:
  name: psp-example-rolebinding
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
roleRef:
  kind: Role
  name: psp-example-role
  apiGroup: rbac.authorization.k8s.i
```

- In Pod Security Policies, one can:
    - Force the user to run as “non-root”
    - Force capabilities to be added or removed
    - Restrict the types of volumes

Open Policy Agent (OPA)

- Consider a user logging into a web server, there are typically multiple steps for logging in:
    - Authentication - Checking identity
    - Authorization - Checking / restricting
- OPA exists to handle authorization requests / verification.
- Without OPA, any time a new service will be deployed, one will have to manually configure all the routes and authorization mechanisms / policies between it and the other services => VERY TEDIOUS!
- In practice, OPA is deployed within the environment and policies are configured within it.
    - When a service wants to make a request to another service, the request first goes to OPA, which either allows or denies it.
- Step 1:
    - Download from Github
    - Execute from within directory: ./opa run -s
- Step 2: Load Policies
    - Policies defined in .rego language (Example follows):

```
package httpapi.authz

# HTTP API request
import input

default allow = false

allow {
  input.path == "home"
  input.user == "john"
}
```

        - {} determines acceptance conditions

    - Once ready, can load the policy via curl to use a PUT request e.g.:
      Curl -X PUT --data-binary @example.rego
      http://localhost:8181/v1/policies/example1
    - To view the list of existing policies:
      Curl http://localhost:8181/v1/policies
- Policies can then be called within programs by making a post request to the API and passing the required input parameters -> If allow conditions are met, OPA allows the request, if not, rejected.
- Note: A guide on utilising the rego language is available via the documentation and a "playground" for policy testing

OPA in Kubernetes

- Kubernetes currently has RBAC for various functions except:
    - Only permit images from certain registries
    - Forbid "runAs Root User"
    - Only allow certain capabilities
    - Enforce pod labelling
- Admission Controllers go some way to covering these failures, with many pre-packaged with Kubernetes and capable of mutation and validation operations however even they have their limitations.
    - For custom admission controllers, would have to build a separate admission controller webhook server
    - This need can be removed and centralized by using OPA in kubernetes.
- Assuming OPA is pre-installed in Kubernetes, one can create a ValidatingWebhookConfiguration:
    - If OPA is not installed in the Cluster, set the ClientConfig URL to the server
    - If installed on the server, one needs to define:
        - The OPA caBundle
        - The OPA Kubernetes service details
- As discussed, the webhook configuration sends an admission review request to the OPA endpoint - in this case, the request is checked against the .rego policy stored within OPA.
    - This can be for operations such as checking image registries.

- For policies where information about other pods is required, need to utilise an import statement:
  Import data.kubernetes.pods
- To help OPA understand the state / definition of the Kubernetes cluster in OPA, one can use the kube-mgmt service
    - Service deployed as a sidecar container alongside OPA
    - Used to:
        - Replicate Kubernetes resources to OPA
        - Load policies into OPA via Kubernetes
            - If creating an OPA policy, can create a ConfigMap with the following label: openpolicyagent.org/policy: rego

              And define the policy logic under the configMap's data field.
- How is OPA deployed on Kubernetes?
    - Deployed with kube-mgmt as a deployment
    - Roles and rolebindings deployed
    - Service to expose OPA service to Kubernetes API Server
    - Deployed in an OPA namespace.
- Within the cluster, one can now create a validating/mutating admission webhook and reference the OPA service

```yaml
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: opa-validating-webhook
webhooks:
  - name: validating-webhook.openpolicyagent.org
    rules:
      - operations: ["CREATE", "UPDATE"]
        apiGroups: ["*"]
        apiVersions: ["*"]
        resources: ["*"]
    clientConfig:
      caBundle: $(cat ca.crt | base64 | tr -d '\n')
      service:
        namespace: opa
        name: opa
```

- Note: The above is the “old” way of implementing OPA to Kubernetes, nowadays a gatekeeper service is used.

Mange Kubernetes Secrets

- Used to store sensitive information e.g. passwords and keys
- Similar to configmaps but stored in hashed format
- Secrets must first be created before injection
- Creation via commands or a definition file:
    - Imperative: kubectl create secret generic <secret name> --from-literal=<key>=<value>, ...

      Or use: --from-file=/path/to/file
    - Declarative:
        - Create a secret.yaml file
            - Under data - add secrets in key-value pairs
- To encode secrets: echo -n ‘secret’ | base64
- Secrets viewable and manageable via kubectl < task > secrets
- To view secrets values: kubectl get secret <secret> -o yaml
- To decode, use the **| base64** command with **--decode flag**

- Secrets can then be injected into pods as environment variables using the envFrom field
    - One can also reference the secret directly using env // valueFrom
    - Alternatively, volumes can be used, but this isn't recommended.

Container Sandboxing

- All containers running on a server share the same underlying kernel. From the host perspective, this is just another process isolated from the host and other containers running it.
- When running a container running a process, the process will have a different process ID within the container and on the host OS.
    - This is process ID namespacing
        - Allows containers to isolate processes from one another
        - Note: If process killed on the host, the container would also be stopped
- For containers to run in the user space, they need to make syscalls to the hardware -> this leads to issues
    - Unlike VMs, which have dedicated Kernels, share the same Kernel on the host OS
    - This can pose massive security risks as it means that containers could interact and affect one another.
    - Sandboxing can resolve this
        - Sandboxing techniques include:
            - Seccomp - Docker default profiles and custom profiles (K8s)
            - Apparmor
        - Both of the above work by whitelisting and blacklisting actions and calls:
            - Whitelist - Block by default and allow particular calls / actions
            - Blacklist - Allow by default and block particular calls/actions
        - Both have varying use cases, there is no set case for what works best.

08/12/2021

gVisor

- The Linux Kernel allows applications to perform an untold number of syscalls to perform a variety of functions
- Whilst this can be great from a developer perspective, the same cannot be said for security perspective. The more opportunities for Kernel interaction there are, the more opportunities for attack.
- The core problem relates more to how each container in a multitenant environment would be interacting with the same underlying OS and Kernel -> Need to improve isolation from container-container AND from container-Kernel
- gVisor aims to implement the isolation between the container and Kernel.
    - When a program / container wants to make a syscall to the kernel, it first goes to gVisor
    - gVisor as a sandbox tool contains a variety of tools:
        - Sentry - An independent application-level Kernel dedicated for containers; intercepting and responding to syscalls
            - Sentry supports much less syscalls than the linux kernel as it's designed to support containers directly; limiting the opportunities for exploit.
        - Gofer - A file proxy that implements the logic for containers to access the filesystem
    - gVisor can also facilitate and monitor network-based operations.
    - Each container has its own gVisor Kernel acting between the container and the host kernel.
- The main con of gVisor is that its compatibility with applications can be limited, it can also result in slightly slower containers.

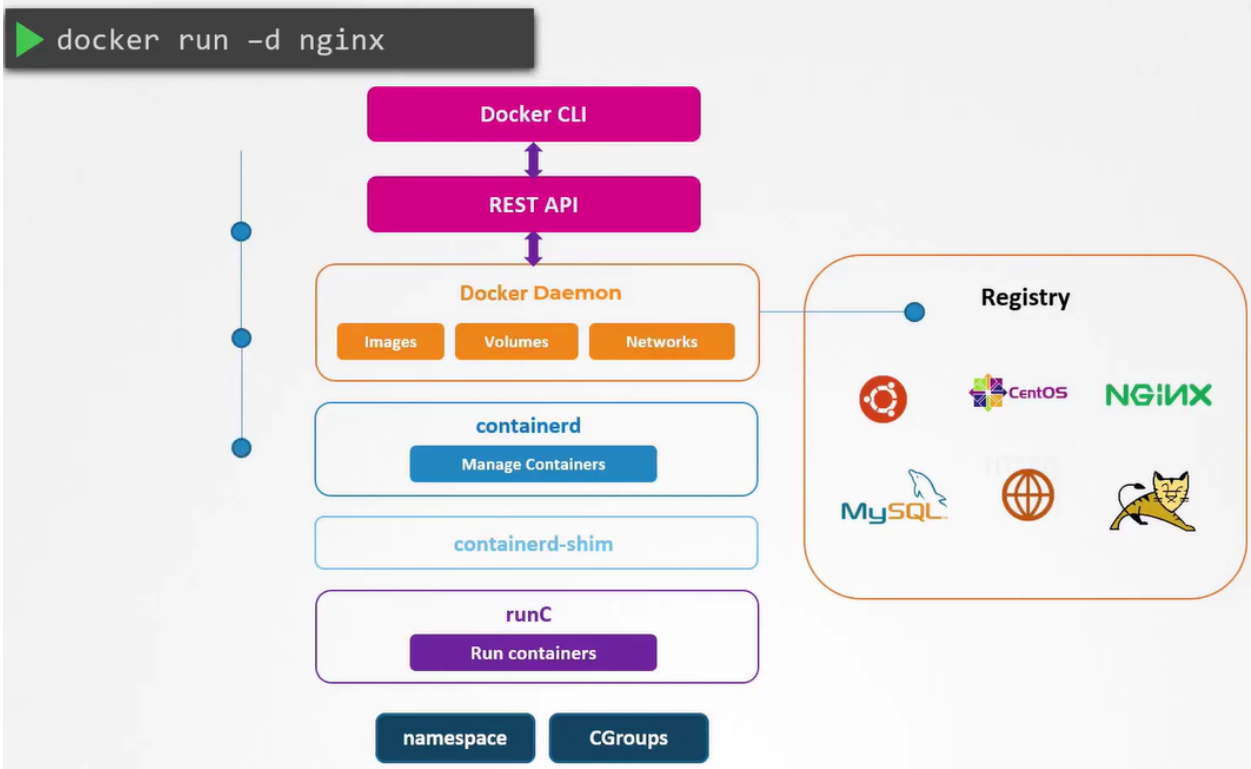
Kata Containers

- Kata aims to set the isolation at its own VM / container
- Each container will have its own dedicated kernel running inside

- Removes the issue(s) when all container apps communicate with the same host kernel -> if any issues occur, only the affected container will have issues
- The VMs created by Kata are lightweight and performance-focussed, therefore would not take long to spin up.
    - The added isolation and VM would require additional resources
    - Additionally, there are compatibility issues for virtualization
- Note: Wouldn't want to run Kata on cloud instances / not be able to, this would be nested virtualization and can lead to major performance issues.

Runtime Classes

- When running a container, the following steps from Docker CLI are applied



1. Docker client converts command to a REST API
2. REST API request passed to Docker Daemon
3. If the image is not present in the local system, it is pulled from the Docker registry
4. The call is made to containerd to start the container -> containerd converts image to OCI-compliant bundle

5. The OCI-compliant bundle is passed to the containerd-shim, triggering the runtime runC to start the container
6. RunC interacts with the Kernel's namespaces and CGroups on the Kernel to create the container

- RunC is referred to the default container runtime and adheres to OCI standards for format and runtime
- With runc installed on a server, containers could be ran on their own via runc run <image>
    - Without docker's features e.g. image management, managing the container would be very tough
    - runC is the default container runtime for many container systems
- Kata and gVisor use their own runtimes that are OCI-compatible
    - The OCI-compatibility means that kata and runsc (gvisor) could be used with Docker without issue by adding a **--runtime <runtime>** to the docker run command.

Using Runtimes in Kubernetes

- Assuming gVisor is already installed on the nodes of a Kubernetes cluster, it can be used as the runtime for containers within easily.
- To do so, a runtimeclass Kubernetes object is required, which can be created using kubectl as usual. The handler should be a valid name associated with the given runtime / application.

gvisor.yaml

```
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
```

- To specify the runtime to be used by the pod, add runtimeClassName: <runtime class name> to the pod spec
- To check if pods are running in this runtime, run pgrep -a <container name>
    - If no output is given, there is an isolation between the system and the container and it's running in the defined runtime as expected.
    - To check the runtime: pgrep -a <runtime>

One-Way SSL vs Mutual SSL (mTLS)

- One-Way SSL:
    - Receivers can only verify identities based on the information sent by clients e.g. emails, social media using usernames and passwords
- If there is no end-user to provide the information e.g. two services, what then?
    - Mutual SSl is required:
        - Client and Server verify their identities
        - When requesting data from the server, the client requests the servers public certificates
        - When the server sends it back, it requests the clients certificate and the client verifies the server certificate with the CA it uses
        - Once verified, the client sends its certificate with a symmetric key encrypted with the server's public key
        - The server then validates the client certificate via the CA.
- Once both certificates are verified, all communication can be encrypted using the symmetric key.

Implement Pod-to-Pod Encryption via mTLS

- By default, data transfer between pods is unencrypted -> This is a MAJOR security risk
- Pods can be configured to use mTLS though.
    - Similar to the previous example, but replace client and server with pod a and pod b respectively.
    - This ensures that both pods verify each other's identities.
- How would this be managed across multiple pods and nodes?

    - This is achievable by getting applications to encrypt communications by default, however this could then cause issues if there are differing encryption methods.
    - Typically, third party applications are used to facilitate the mTLS e.g. Istio and Linkerd
        - These are Service Mesh tools and aren't confined to mTLS only, they are typically used to facilitate microservice architecture
    - Tools like this run alongside pods as sidecar containers
    - When a pod sends a message to another, istio intercepts, encrypts and sends the message, where it is encrypted by the istio container running alongside the receiving pod.
    - Istio supports varying types of encryption levels / modes:
        - Permissive / Opportunistic - Will accept unencrypted traffic where possible / deemed safe
        - Enforced / Strict - No unencrypted traffic allowed at all

28/12/2021

06 - Supply Chain Security

Minimize Base Image Footprint

- Base vs Parent Image
    - Parent image - any image used to build a custom image
    - Base image - any image built from scratch
- Note: Parent vs Base image may be used interchangeably, not a major issue
- For these notes - base image = any image used to build a custom image
- Best Practices for image building:
    - Images shouldn't be built that contain multiple applications e.g. databases, applications
        - Images should be modular i.e. 1 for a DB, etc. They should each solve their own problem and have their own independent set of dependencies

        - Once deployed as containers, these images can be controlled individually for scaling
    - Persist State:
        - Data or state shouldn't be stored in containers as they are ephemeral in nature - one should be able to destroy and recreate a container without losing data
        - Store on an external volume or via a caching service
    - Choosing a base image:
        - Should always consider base images that suit the technical need of the solution being developed
        - Images can be viewed on Docker Hub
        - Areas to note when searching for a base image:
            - Is it from an official source?
            - Is it up to date?
    - Slim/Minimal Images
        - Minimizing the size of the image allows quicker pulling and building
        - General steps that can be taken:
            - Create slim/minimal images
            - Find an official minimal image that exists
            - Only install necessary packages e.g. remove shells/package managers/tools - anything which one can use to infiltrate a system
            - Ensure images are suited to the environment that they are being used for e.g. for a development environment, include debug tools, for production, ensure as lean as possible
            - Use multi-stage builds - ensures lean, production-ready images
- Distroless Docker Images
        - Contain only the application and docker runtime dependencies
        - Provided by google
- Minimizing image footprint leads to smaller area of attack for vulnerabilities -> more secure image

Image Security

- Docker images follow the naming convention where image: <image name>
    - Image name = image / repository referenced
    - I.e. library/image name
        - Library = default account where docker official images are stored
        - If referencing from a particular account - swap library with account name
    - Images typically pulled from docker registry at docker.io by default
- Private repositories can also be referenced
    - Requires login via **docker login <registry name>**
    - It can then be referenced via the full path in the private registry
    - To facilitate the authentication - create a secret of type docker-registry i.e.:

Kubectl create secret docker-registry <name> --docker-server=<registry name>
--docker-username=<username> --docker-password=<password>
--docker-email=<email>

Then, in the pod spec, add:
imagePullSecrets:

- **Name: <secret name>**

Whitelist Allowed Registries - Image Policy Webhook

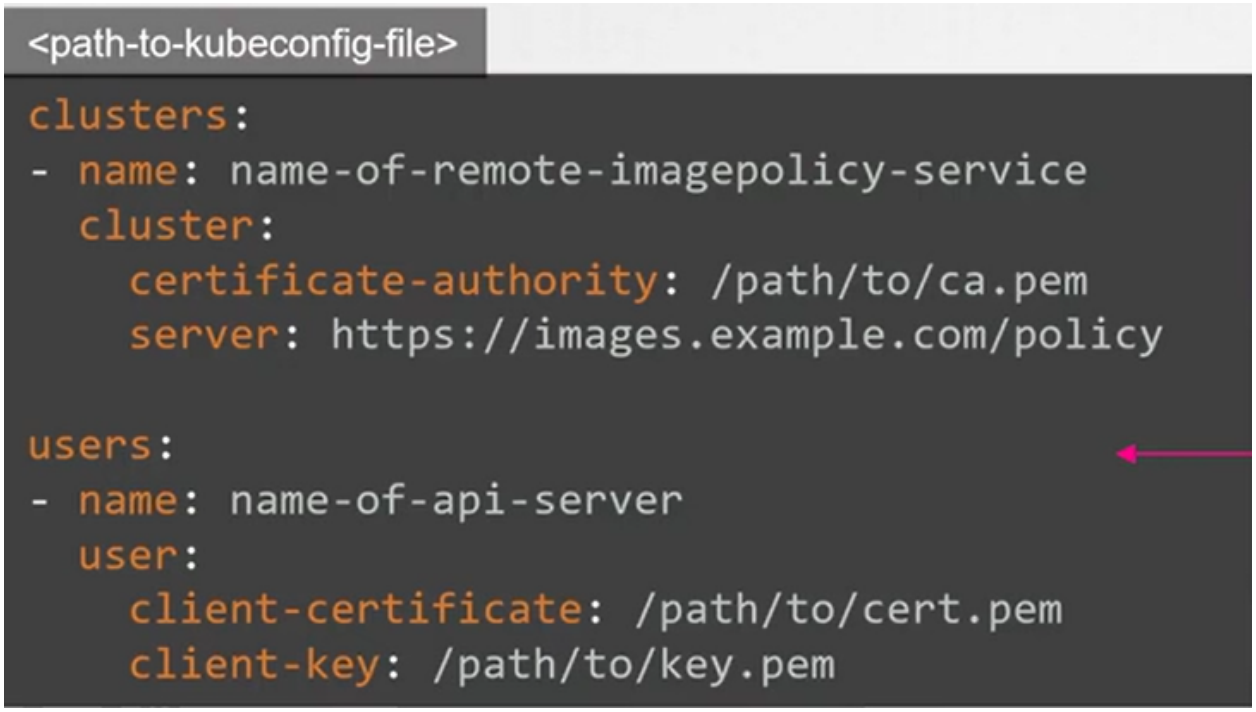
- By default, any image can be pulled from any registry (private or open) in Kubernetes
- This is a huge security risk, as if a image is pulled from a non-approved registry, it could contain multiple vulnerabilities that can be taken advantage of
- In general, need to ensure images are pulled from approved registries only i.e. sufficient governance
    - This can be handled using admission controllers as discussed previously
    - When a request comes in, the admission controller can check the registry

    - We can make an admission webhook server and configure the webhook admission controller to validate the registry provided, along with any messages
    - Alternatively:
        - Could deploy an OPA service - configure a validating webhook to connect to the opa service and check agains the rego policy
    - Alternatively again:
        - Could utilise a custom build imagepolicywebhook, which can communicate with an admission webhook server via an admission configuration file
- ImagePolicyWebHook Admission Config File:

/etc/kubernetes/admission-config.yaml

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: ImagePolicyWebhook
  configuration:
    imagePolicy:
      kubeConfigFile: <path-to-kubeconfig-file>
      allowTTL: 50
      denyTTL: 50
      retryBackoff: 500
      defaultAllow: true
```

- defaultAllow: true - if webhook not accessible, default behaviour allows pod to be created
    - Setting to false denies by default unless the exceptions are explicitly defined elsewhere
- Access to the admission webhook server is defined in the kubeconfig file



- Once setup, the admission controller can be added to the --enable-admission-plugins flag in the kube-apiserver .service or yaml file as appropriate.
    - Additionally: pass --admission-control-config-file=<path to config file>
    - This is used to help enable the admission controller webhook and authenticate it appropriately

Static Analysis of User Workloads - Kubernetes resources, Docker Files, etc

- When submitting a resource creation request in kubernetes it goes through:
    - Kubectl -> authentication -> authorisation -> admission controllers -> create pod
- What if we want to catch any security vulnerabilities before using kubectl? Static analysis can be used for this - resource files are reviewed against policies.
- Example tools - kubesec
    - Helps analyse a given resource definition file and returns a report with an associated score based on each vulnerability along with rationale regarding the vulnerability

- Kubesec installation
    - Installable via a binary
    - Run by kubesec scan <pod.yaml>
    - OR make a curl POST request:
      Curl -sSX POST --data-binary @"pod.yaml" https://v2.kubesec.io/scan
    - OR:
        - Kubecsec http 8080 &
          Runs a kubesec instance locally on the server

30/12/2021

Scan Images for Known Vulnerabilities (Trivy)

- CVE = Common Vulnerabilities and Exposures
- User-submitted database for vulnerabilities, workarounds and why it's an issue
- What constitutes a CVE?
    - Anything that can allow an attacker to bypass security checks and perform unwanted actions
    - Anything that allows attackers to seriously affect application performance
- Each CVE gets a severity rating from 0-10 - helps to understand what vulnerabilities should be shown greater focus etc
- In general, a higher score = greater vulnerability
- Example - Download from http instead of https gives a score of around 7.3
- Kubernetes clusters will have various processes and packages running at any given point, the attack area can be minimized by actions such as deleting unnecessary packages as discussed previously.
- To understand the current state of the cluster in terms of vulnerabilities across processes, containers, and so on, one can utilise CVE Scanners
    - Container scanners look for vulnerabilities in container / execution environment - applications in the container
    - Once vulnerabilities are identified, the appropriate action(s) can be taken e.g. update versions, remove packages, etc
    - In general - more packages = greater footprint = greater amount of vulnerabilities

- **Example - Trivy**
    - Provided by AquaSecurity as a simple CVE scanner for containers, artifacts, etc
    - Can be integrated with CI/CD pipelines
    - Can easily be installed as if installing a typical package
    - To scan: **trivy image <image name>:<tag>**
        - Additional flags available e.g.:
            - **--severity=<severity 1>,<severity 2>**
            - **--ignore-unfixed** (ignore any vulnerabilities that can't be fixed even if packages are updated)
        - Trivy can be used to scan images in a tar format too e.g.:
            - **Docker save <image> > <name>.tar**
            - **Trivy image --input archive.tar**
    - Reducing vulnerabilities can be done by using minimal images e.g. alpine images like nginx-alpine
- Best practices:
    - Continuously rescan images
    - Use kubernetes admission controllers to scan images
    - Use your own repository with pre-scanned images ready to go
    - Integrate container scanning into CI/CD pipelines

07 - Monitoring, Logging and Runtime Security

Perform Behavioural Analytics of Syscall Processes

- Various ways of securing kubernetes clusters have been discussed so far:
    - Securing kubernetes nodes
    - Minimizing microservices vulnerabilities
    - Sandboxing techniques
    - MTLS Encryption
    - Network Access Restriction
- No guarantee that utilising these 100% prevents the possibility of an attack, how can one prepare for the possibility?

- In general for vulnerabilities, the sooner that the possibility of a vulnerability has been exploited is noted, the better
- Actions that could be taken include:
    - Alert notifications
    - Rollbacks
    - Set limits for transactions or resource usage
- Identifying breaches that have already occurred can be done using tools like Falco
    - Syscalls can be monitored by tools like tracee
    - Need to analyse syscalls like these in real time to monitor events occurring within the system and note any of suspicious nature
        - Example - Accessing a containers bash shell and going to the /etc/shadow section, deleting logs,

31/12/21

Falco Overview and Installation

- For functionality, Falco must monitor the syscalls coming from the applications in the user space into the linux kernel.
- This is done by a particular Kernel module - this is intrusive and forbidden by some Kubernetes service providers
- A workaround is available via the eBPF in a similar manner to the Aquasec Tracee tool.
- Syscalls are analysed by Sysdig libraries in the user space and filtered by the falco policy engine based on predefined rules to determine the nature of the event.
- Alert notifications are sent via various methods including email and slack at the users discretion.
- Steps are provided in the Falco getting started documentation for running it as a service on Linux
- Note: In the event the Kubernetes cluster is compromised, Falco will still be running.
- Alternatively, Falco can be installed as a daemonset via installing the helm charts.
- Falco pods should then be running on all nodes.

Use Falco to Detect Threats

- Check that falco is running:
    - **Systemctl status falco** (if running on host)
- Creating a pod as normal, in a separate terminal, one can ssh into the node and run: **journalctl -fu falco**
    - This allows inspection of the events generated / picked up by the falco service
    - Note: the fu flag allows events to be automatically added as they appear
- In the original terminal, executing a shell in the pod generates an event picked up by falco.
    - Details displayed include pod, namespace, container name, commands ran, etc.
- The same is applied for any activity ran.
- Falco implements several rules by default to detect events, such as creating a shell and reading particular files.
    - Rules are defined in rules.yaml file.
    - Elements included: rules, lists and macros
    - Rules:
        - Defines all the conditions for which an event should be triggered
            - Rule - Name of the rule
            - Desc - What is the detailed explanation of the rule
            - Condition - filtering expression applied against events matching the rule
            - Output - output generated for any events matching the rule
            - Priority - severity of the rule

- Custom rule example - shell opening in a container anywhere not equal to the host

rules.yaml

```
- rule:  Detect Shell inside a container
  desc:  Alert if a shell such as bash is open inside the container
  condition:  container.id != host and proc.name = bash
  output:   Bash Shell Opened (user=%user.name %container.id)
  priority: WARNING
```

- Note: Conditions are used via sysdig filters e.g. container.id, fd.name (file descriptor), evt.type (event type), user.name, container.image.repository
- Outputs also can utilise filters like this.
- Priority can be any of EMERGENCY, ALERT, DEBUG, NOTICE, etc.
- Note: for a set of similar commands e.g. opening any possible shell for the container, lists can be used.

rules.yaml

```
- rule:  Detect Shell inside a container
  desc:  Alert if a shell such as bash is open inside the container
  condition:  container.id != host and proc.name in (linux_shells)
  output:   Bash Opened (user=%user.name container=%container.id)
  priority: WARNING

- list: linux_shells
  items: [bash, zsh, ksh, sh, csh]
```

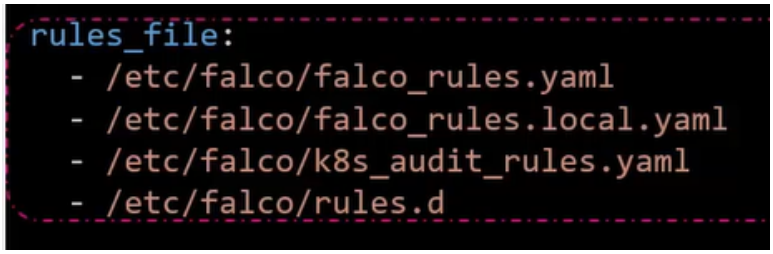
- Macros can be used to shorten filters e.g.

```
- macro: container
  condition: container.id != host
```

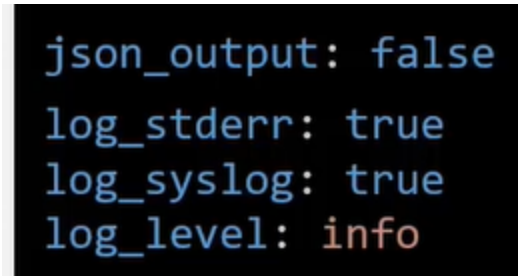
Falco Configuration Files

- Main falco configuration file is located at /etc/falco/falco.yaml
    - Vieweable via either **journalctl -fu falco** or **/usr/lib/systemd/system/falco.service**

    - Contains all the configuration parameters associated with falco e.g. display format, output channel configuration etc.
- Common options:
    - How are rules loaded? Rules_file list
    - Note: The order of the rules files is important, as this is the order that falco will check them -> stick top priority rule files first.



    - Logging of events - What format and verbosity are used?



    - Minimum priority that should be logged determined by priority key (debug is the default)
    - Output channels:
        - Stdoutput set to true by default
        - Can configure output to a particular file in a similar manner or to a particular program

```
file_output:
  enabled: true
  filename: /opt/falco/events.txt

program_output:
  enabled: true
  program: "jq '{text: .output}' | curl -d @- -X POST https://hooks.slack.com/services/XXX"
```

- HTTP Endpoints are also configurable for output:

```
http_output:
  enabled: true
  url: http://some.url/some/path/
```

- Note: For any changes made to this file, falco must be restarted to take effect.
- Rules:
    - Default file = /etc/falco/falco_rules.yaml

      Example:

/etc/falco/falco_rules.yaml

```
- rule: Terminal shell in container
  desc: A shell was used as the entrypoint/exec point into a container with an attached terminal.
  condition: >
    spawned_process and container
    and shell_procs and proc.tty != 0
    and container_entrypoint
    and not user_expected_terminal_shell_in_container_conditions
  output: >
    A shell was spawned in a container with an attached terminal (user=%user.name user_loginuid=%user.
loginuid %container.info
    shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline terminal=%proc.tty container_id=%contain
er.id image=%container.image.repository)
  priority: NOTICE
```

    - Any changes made to this file will be overwritten when updating the falco package. To avoid this, add to /etc/falco/falco)rules.local.yaml
- Hot reload can be used to avoid restarting the falco service and allow changes to take place:
    - Find the process ID of falco at /var/run/falco.pid
    - Run a kill -1 command: kill -1 $(cat /var/run/falco.pid)

Reference Links

https://falco.org/docs/getting-started/installation/

https://github.com/falcosecurity/charts/tree/master/falco

https://falco.org/docs/rules/supported-fields/

https://falco.org/docs/rules/default-macros/

https://falco.org/docs/configuration/

Mutable vs Immutable Infrastructure

- Software upgrades can be done via manual methods or using configuration tools such as custom scripts or configuration managers like ansible
- In high-availability setups, could apply the same update approach to each server running the software - in-place updates
    - Configuration remains the same, but the software has changed
    - This leads to mutable infrastructure
- If the upgrade fails for a particular server due to dependency issues like network or files, a configuration drift can occur - each server behaves slightly differently to one another.
- To workaround, can just spin up new servers with the new updated software and delete the old servers upon successful updates
    - This is the idea behind immutable infrastructure - Unchanged infrastructure
- Immutability is one of the primary thoughts on containers
    - As they are made using images, any changes e.g. version updates should be applied to an image first, then that image is used to create new containers via rolling updates
    - Note: containers can be changed during runtime e.g. copying files to and from containers - this is not in line with security best practices

Ensure Immutability of Containers at Runtime

- Even though containers are designed to be immutable by default, there are ways to do in-place updates on them:
    - Copying files to containers:
        - **kubectl cp nginx.conf nginx:/etc/nginx**
        - **Kubectl cp <file name> <container name>:<target path>**
    - Executing a shell into the container and making changes:
        - **Kubectl exec -ti nginx -- bash nginx:/etc/nginx**

- To prevent this, one could add to the pod definition file security contexts in a similar manner to start with a readonly root file system e.g.:

nginx.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    securityContext:
          readOnlyRootFilesystem: true
```

- This is generally not advisable for applications that may need to write to different directories like storing cache data, and so on

- This can be worked around by using volumes:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx

    securityContext:
          readOnlyRootFilesystem: true
    volumeMounts:
        - name: cache-volume
          mountPath: /var/cache/nginx
        - name: runtime-volume
          mountPath: /var/run

    volumes:
      - name: cache-volume
        emptyDir: {}
      - name: runtime-volume
        emptyDir: {}
```

- Considering the same file above, in the event this is being ran with privileged set to true, the read-only option will be overwritten -> even more proof that containers shouldn't run as root.
- In general:
    - Avoid setting readOnlyRootFileSysten as false
    - Avoid setting privileged to true and runAsUser to 0
- The above can be enforced via PodSecurityPolicies as discussed previously.

Use Audit Logs to Monitor Access

- Previously seen how falco can produce details for events such as namespace, pod, and commands used in Kubernetes, but how can these be audited?
- Kubernetes allows auditing by default via the kube-apiserver.
- When making a request to the cluster, all requests go to the api server
    - When the request is made - it goes through "**RequestReceived**" stage - generates an event regardless of authentication and authorization
    - Once authenticated and authorized, stage is **ResponseStarted**
    - Once request completed - stage is **ResponseComplete**
    - In the event of an error - stage is **Panic**
    - Each of the above generate events, but this is not enabled by default, as a significant amount of unnecessary events would be recorded.
- In general, want to monitor only the events we actually care about, such as deleting pods from a particular namespace.

- This can be done by creating a policy object configuration file in a similar manner to below:

audit-policy.yaml

```yaml
apiVersion: audit.k8s.io/v1
kind: Policy
omitStages: ["RequestReceived"]
rules:
  - namespaces: ["prod-namespace"]
    verbs: ["delete"]
    resources:
    - groups: " "
      resources: ["pods"]
      resourceNames: ["webapp-pod"]
    level: RequestResponse

  - level: Metadata
    resources:
    - groups: " "
      resources: ["secrets"]
```

- Note: omitStages is optional, define the stages you want to ignore in an array
- Rules: add each rule to be considered in a list, noting verbs, namespaces, resources where applicable, etc.
    - For resources, can specify the groups that they belong to and the resources within these groups
    - Additional refinement can be done via resourceNames
    - Level determines logging verbosity - none, metadata, request, etc.

- Auditing is disabled by default in Kubernetes, need to configure an auditing backend for the kube-apiserver, two types are supported: log file backend or webhook service backend.
- The path to the audit-log file path must be added to the static pod definition **--audit-log-path=/var/path/to/file**
- To point he apiserver to the policy to be referenced: **--audit-policy-file=/path/to/file**
- Additional options include:
    - **--audit-log-maxage**: what is the longest a log will be kept for?
    - **--audit-log-maxbackup**: maximum number of log files to be retained on the host
    - **--audit-log-maxisize**: maximum file size for given log files
- To test, carry out the event(s) described in the policy files.

08 - Mock Exams

Mock Exam 1

Q1:

A pod has been created in the omni namespace. However, there are a couple of issues with it.

The pod has been created with more permissions than it needs.

It allows read access in the directory /usr/share/nginx/html/internal causing an Internal Site to be accessed publicly.

To check this, click on the button called Site (above the terminal) and add /internal/ to the end of the URL.

Use the below recommendations to fix this.

Use the AppArmor profile created at /etc/apparmor.d/frontend to restrict the internal site.

There are several service accounts created in the omni namespace. Apply the principle of least privilege and use the service account with the minimum privileges (excluding the default service account).

Once the pod is recreated with the correct service account, delete the other unused service accounts in omni namespace (excluding the default service account).

You can recreate the pod but do not create a new service accounts and do not use the default service account.

Q2:

A pod has been created in the `orion` namespace. It uses secrets as environment variables. Extract the decoded secret for the `CONNECTOR_PASSWORD` and place it under `/root/CKS/secrets/CONNECTOR_PASSWORD`.

You are not done, instead of using secrets as an environment variable, mount the secret as a read-only volume at path `/mnt/connector/password` that can be then used by the application inside.

Q3:

A number of pods have been created in the `delta` namespace. Using the `trivy` tool, which has been installed on the controlplane, identify and delete pods `except` the ones with least number of `CRITICAL` level vulnerabilities.

Note: Do not modify the objects in anyway other than deleting the ones that have critical vulnerabilities.

Q4:

Create a new pod called `audit-nginx` in the default namespace using the `nginx` image. Secure the syscalls that this pod can use by using the `audit.json` seccomp profile in the `pod's` security context.

The `audit.json` is provided at `/root/CKS` directory. Make sure to move it under the `profiles` directory inside the default seccomp directory before creating the pod

Q5:

The CIS Benchmark report for the `kube-apiserver` is available at the tab called `CIS Report 1`.
Inspect this report and fix the issues reported as `FAIL`

Q6:

There is something suspicious happening with one of the pods running an `httpd` image in this cluster.
The Falco service shows frequent alerts that start with: `File below a known binary directory opened for writing`.
Identify the rule causing this alert and update it as per the below requirements:

1. Output should be displayed as: `CRITICAL File below a known binary directory opened for writing (user=user_name file_updated=file_name command=command_that_was_run)`
2. Alerts are logged to `/opt/security_incidents/alerts.log`

Do not update the default rules file directly. Rather use the `falco_rules.local.yaml` file to override.
`Note:` Once the alert has been updated, you may have to wait for up to a minute for the alerts to be written to the new log location.

Q7:

A pod called `busy-rx100` has been created in the `production` namespace. Secure the pod by recreating it using the `runtimeClass` called `gvisor`. You may delete and recreate the pod.

Note: As long as the pod is recreated with the correct runtimeClass, the task will be marked correct. This lab environment does not support gvisor at the moment so if the pod is not in a running state, ignore it and move on to the next question.

Q8:

We need to make sure that when pods are created in this cluster, they cannot use the `latest` image tag, irrespective of the repository being used.
To achieve this, a simple Admission Webhook Server has been developed and deployed. A service called `image-bouncer-webhook` is exposed in the cluster internally. This Webhook server ensures that the developers of the team cannot use the latest image tag.
Make use of the following specs to integrate it with the cluster using an `ImagePolicyWebhook`:

1. Create a new admission configuration file at `/etc/admission-controllers/admission-configuration.yaml`
2. The `kubeconfig` file with the credentials to connect to the webhook server is located at `/root/CKS/ImagePolicy/admission-kubeconfig.yaml`. Note: The directory `/root/CKS/ImagePolicy/` has already been mounted on the `kube-apiserver` at path `/etc/admission-controllers` so use this path to store the admission configuration.
3. Make sure that if the latest tag is used, the request must be rejected at all times.
4. Enable the Admission Controller.

Finally, delete the existing pod in the `magnum` namespace that is in violation of the policy and recreate it, ensuring the same image but using the tag `1.27`.

Mock Exam 2

Q1:

A pod called `redis-backend` has been created in the `prod-x12cs` namespace. It has been exposed as a service of type `ClusterIP`. Using a network policy called `allow-redis-access`, lock down access to this pod only to the following:

1. Any pod in the same namespace with the label `backend=prod-x12cs`.
2. All pods in the `prod-yx13cs` namespace.
All other incoming connections should be blocked.

Use the `existing labels` when creating the network policy.

Q2:

A few pods have been deployed in the `apps-xyz` namespace. There is a pod called `redis-backend` which serves as the backend for the apps `app1` and `app2`. The pod called `app3` on the other hand, does not need access to this `redis-backend` pod. Create a network policy called `allow-app1-app2` that will only allow incoming traffic from `app1` and `app2` to the `redis-pod`.

Make sure that all the available `labels` are used correctly to target the correct pods. Do not make any other changes to these objects.

Q3:

A pod has been created in the `gamma` namespace using a service account called `cluster-view`. This service account has been granted additional permissions as compared to the default service account and can `view` resources cluster-wide on this Kubernetes cluster. While these permissions are important for the application in this pod to work, the secret token is still mounted on this pod.

Secure the pod in such a way that the secret token is no longer mounted on this pod. You may delete and recreate the pod.

Q4:

A pod in the `sahara` namespace has generated alerts that a shell was opened inside the container.

Change the format of the output so that it looks like below:

`ALERT timestamp of the event without nanoseconds,User ID,the container id,the container image repository`

Make sure to update the rule in such a way that the changes will persists across Falco updates.

You can refer the `falco` documentation Here

Q5:

`martin` is a developer who needs access to work on the `dev-a`, `dev-b` and `dev-z` namespace. He should have the ability to carry out any operation on any `pod` in `dev-a` and `dev-b` namespaces. However, on the `dev-z` namespace, he should only have the permission to `get` and `list` the pods.

The current set-up is too permissive and violates the above condition. Use the above requirement and secure martin's access in the cluster. You may re-create objects, however, make sure to use the same name as the ones in effect currently

Q6:

On the controlplane node, an unknown process is bound to the port 8088. Identify the process and prevent it from running again by stopping and disabling any associated services. Finally, remove the package that was responsible for starting this process.

Q7:

A pod has been created in the `omega` namespace using the pod definition file located at `/root/CKS/omega-app.yaml`. However, there is something wrong with it and the pod is not in a running state.

We have used a custom seccomp profile located at `/var/lib/kubelet/seccomp/custom-profile.json` to ensure that this pod can only make use of limited syscalls to the Linux Kernel of the host operating system. However, it appears the profile does not allow the `read` and `write` syscalls. Fix this by adding it to the profile and use it to start the pod.

Q8:

A pod definition file has been created at `/root/CKS/simple-pod.yaml` . Using the `kubesec` tool, generate a report for this pod definition file and fix the major issues so that the subsequent scan report no longer fails.

Once done, generate the report again and save it to the file `/root/CKS/kubesec-report.txt`

Q9:

Create a new pod called `secure-nginx-pod` in the `seth` namespace. Use one of the images from the below which has no `CRITICAL` vulnerabilities.

1. nginx
2. nginx:1.19
3. nginx:1.17
4. nginx:1-alpine
5. gcr.io/google-containers/nginx
6. bitnami/nginx:latest
7. httpd:2-alpine

Mock Exam 3

Q1:

A kube-bench report is available at the `Kube-bench assessment report` tab. Fix the tests with `FAIL` status for `4 Worker Node Security Configuration` .

Make changes to the `/var/lib/kubelet/config.yaml`

After you have fixed the issues, you can update the published report in the `Kube-bench assessment report` tab by running `/root/publish_kubebench.sh` to validate results.

Q2:

Enable auditing in this kubernetes cluster. Create a new policy file that will only log events based on the below specifications:

Namespace: `prod`

Level: `metadata`

Operations: `delete`

Resources: `secrets`

Log Path: `/var/log/prod-secrets.log`

Audit file location: `/etc/kubernetes/prod-audit.yaml`

Maximum days to keep the logs: `30`

Once the policy is created it, enable and make sure that it works.

Q3:

Enable `PodSecurityPolicy` in the Kubernetes API server.

- Create a PSP with below conditions:

1. PSP name : `pod-psp`
2. Privilege to run as root on host: `false`
3. Allowed volumes to mount on pod: `configMap,secret,emptyDir,hostPath`
4. seLinux, runAsUser, supplementalGroups, fsGroup: `RunAsAny`

- Fix the pod definition `/root/pod.yaml` based on this PSP and deploy the pod. Ensure that the pod is running after applying the above pod security policy.

Q4:

We have a pod definition template `/root/kubesec-pod.yaml` on `controlplane` host. Scan this template using the `kubesec` tool and you will notice some failures.

Fix the failures in this file and save the success report in `/root/kubesec_success_report.json`.

Make sure that the final kubesec scan status is `passed`.

Q5;

In the `dev` namespace create below resources:

- A role `dev-write` with access to get, watch, list and create pods in the same namespace.

- A Service account called `developer` and then bind `dev-write` role to it with a rolebinding called `dev-write-binding`.

- Finally, create a pod using the template `/root/dev-pod.yaml`. The pod should run with the service account `developer`. Update `/root/dev-pod.yaml` as necessary

Q6:

Try to create a pod using the template defined in `/root/beta-pod.yaml` in the namespace `beta`. This should result in a failure.

Troubleshoot and fix the OPA validation issue while creating the pod. You can update `/root/beta-pod.yaml` as necessary.

The Rego configuration map for OPA is in `untrusted-registry` under `opa` namespace.

NOTE: In the end pod need not to be successfully running but make sure that it passed the OPA validation and gets created.

Q7:

We want to deploy an `ImagePolicyWebhook` admission controller to secure the deployments in our cluster.

- Fix the error in `/etc/kubernetes/pki/admission_configuration.yaml` which will be used by `ImagePolicyWebhook`
- Make sure that the policy is set to implicit deny. If the webhook service is not reachable, the configuration file should automatically reject all the images.
- Enable the plugin on API server.

The kubeconfig file for already created imagepolicywebhook resources is under `/etc/kubernetes/pki/admission_kube_config.yaml`

Q8:

Delete pods from `alpha` namespace which are not immutable.

Note: Any pod which uses elevated privileges and can store state inside the container is considered to be non-immutable.