OPA - k8s

Kubernetes security concepts:

- **`runAsUser`**: This is a Kubernetes security setting that specifies the user ID that a container should run as. By default, containers in Kubernetes will run as the **`root`** user, which can pose potential security risks. Setting **`runAsUser`** to a non-root UID ensures that if the container is compromised, the attacker will not have root privileges.
- **`runAsNonRoot`**: By default, containers in Kubernetes will run as the **`root`** user inside the container. This can lead to security vulnerabilities as **`root`** is the highest privileged user. The **`runAsNonRoot`** setting ensures that a container is run with a non-root user, which reduces the attack surface and limits the damage an attacker can do if they gain access to a compromised container.
- **`readOnlyRootFilesystem`**: This is a security setting in Kubernetes that restricts write access to the root filesystem of a container. When this setting is enabled, any attempt to modify the container's root filesystem will result in an error. This helps prevent malicious code from being injected into the container and also reduces the risk of data loss or corruption.

Overall, these concepts help improve the security of Kubernetes environments by reducing the attack surface and limiting access and privileges to containers running within the Kubernetes cluster.

Create next stage in Jenkins pipeline:

```
stage('Vulnerability Scan - Kubernetes') {
    steps {
      sh 'docker run --rm -v $(pwd):/project openpolicyagent/conftest test --policy opa-k8s-security.rego 
k8s_deployment_service.yaml'
    }
  }
```

Create OPA k8s rego file:

```
package main

deny[msg] {
  input.kind = "Service"
  not input.spec.type = "NodePort"
  msg = "Service type should be NodePort"
}

deny[msg] {
  input.kind = "Deployment"
  not input.spec.template.spec.containers[0].securityContext.runAsNonRoot = true
  msg = "Containers must not run as root - use runAsNonRoot wihin container security context"
}
```

k8s deployment file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: devsecops
  name: devsecops
spec:
  replicas: 2
  selector:
    matchLabels:
      app: devsecops
  strategy: {}
  template:
    metadata:
      labels:
        app: devsecops
    spec:
      containers:
      - image: replace
        name: devsecops-container
        securityContext:
          runAsNonRoot: true
          runAsUser: 100
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: devsecops
  name: devsecops-svc
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: devsecops
  type: NodePort
```

Reference:
https://kubernetes.io/docs/tasks/configure-pod-container/security-context/
https://snyk.io/blog/10-kubernetes-security-context-settings-you-should-understand/

Create script k8s deployment

```
#!/bin/bash
#k8s-deployment.sh
sed -i "s#replace#${imageName}#g" k8s_deployment_service.yaml
kubectl -n default get deployment ${deploymentName} > /dev/null
if [[ $? -ne 0 ]]; then
    echo "deployment ${deploymentName} doesnt exist"
    kubectl -n default apply -f k8s_deployment_service.yaml
else
    echo "deployment ${deploymentName} exist"
    echo "image name - ${imageName}"
    kubectl -n default set image deploy ${deploymentName} ${containerName}=${imageName} --record=true
fi
```

The code is a Bash script that updates a Kubernetes deployment with a new container image. Here are the explanations of each line:

- **`#!/bin/bash`**: This line indicates the interpreter to be used, which is Bash.
- **`sed -i "s#replace#${imageName}#g" k8s_deployment_service.yaml`**: This line replaces the placeholder string "replace" with the value of the **`imageName`** variable in the **`k8s_deployment_service.yaml`** file using the **`sed`** utility.
- **`kubectl -n default get deployment ${deploymentName} > /dev/null`**: This line queries the Kubernetes default namespace for the **`${deploymentName}`** deployment. The **`> /dev/null`** redirects the output to null, so it doesn't print anything to the console.
- **`if [[ $? -ne 0 ]]; then`**: This line checks the return value of the previous command (i.e., the deployment exists or not). If the value is not zero (i.e., the deployment does not exist), then it goes to the **`if`** block.
- **`echo "deployment ${deploymentName} doesnt exist"`**: This line prints the message to the console, indicating that the deployment does not exist.
- **`kubectl -n default apply -f k8s_deployment_service.yaml`**: This line applies the deployment and service configuration from the **`k8s_deployment_service.yaml`** file to create the deployment.
- **`else`**: If the **`if`** block condition is not satisfied, the script proceeds to the **`else`** block.
- **`echo "deployment ${deploymentName} exist"`**: This line prints the message to the console, indicating that the deployment already exists.
- **`echo "image name - ${imageName}"`**: This line prints the name of the new container image.
- **`kubectl -n default set image deploy ${deploymentName} ${containerName}=${imageName} --record=true`**: This line updates the existing deployment by changing the container image of the **`${containerName}`** container to **`${imageName}`**. The **`--record=true`** records the change in the revision history of the deployment.

The **`kubectl -n default`** option specifies the namespace in which to execute the given Kubernetes command.
A Kubernetes namespace is an abstraction used to partition a cluster into logical parts. It enables multiple virtual clusters to be created on top of a physical cluster. The **`default`** namespace is the starting namespace when you first create a Kubernetes cluster. Every object that created without a namespace specified goes into the **`default`** namespace.
When you execute **`kubectl`** commands, by default the context of the current namespace is taken into account. But if you want to operate on a different namespace other than the current one, you need to specify the namespace using the **`-n`** flag followed by the namespace name.
In the script, the **`kubectl -n default`** option is used to specify the namespace **`default`** to execute the **`get`**, **`apply`**, and **`set`** commands specific to the namespace where the Kubernetes deployment is located.

The **`> /dev/null`** portion of the code is a Linux and Unix shell feature to discard the output generated by a command.
In the script above, **`kubectl -n default get deployment ${deploymentName}`** is used to check if the deployment with the name **`${deploymentName}`** exists in the **`default`** namespace. If the deployment exists, we do not necessarily need to print the output since it is not critical in the context of the script. Therefore, the **`> /dev/null`** redirects any output produced by the **`get`** command to **`null`** or nowhere, meaning that any output generated by the **`get`** command is entirely discarded.
This way, the redirection **`> /dev/null`** reduces unwanted messages printed to the console, resulting in a cleaner console output.

Create script k8s rollback

```
#!/bin/bash
#k8s-deployment-rollout-status.sh
sleep 60s
if [[ $(kubectl -n default rollout status deploy ${deploymentName} --timeout 5s) != *"successfully rolled out"* ]];
then
```

```
  echo "Deployment ${deploymentName} Rollout has Failed"
  kubectl -n default rollout undo deploy ${deploymentName}
  exit 1;
else
  echo "Deployment ${deploymentName} Rollout is Success"
fi
```

This Bash script is designed to check the status of a Kubernetes deployment rollout and rollback the deployment if the rollout has failed.
Here is how the script works step by step:
The script starts by pausing for 60 seconds using the `**sleep**` command. This is done to ensure that the deployment rollback status check is not executed until the expected time for the rollout to start has passed.
The script then executes the `**kubectl**` command with the `**rollout status**` sub-command and the `**deploy**` resource type to check the status of the deployment rollout. It uses the `**deploymentName**` variable to specify the name of the deployment that needs to be checked. The `**--timeout**` option specifies the maximum time that `**kubectl**` should wait for the rollout to complete.
The `**kubectl**` command output is captured as a string and compared against the pattern `**"successfully rolled out"**`. If the pattern is not found in the output, the script proceeds to the `**then**` block.
If the pattern is not found, the script prints a message to the console indicating that the deployment rollout has failed using the `**echo**` command. It then executes the `**kubectl**` command with the `**rollout undo**` subcommand to undo the deployment rollout using the `**deploymentName**` variable. This will cause the deployment to rollback to the previous deployment version.
Finally, the script terminates with a failure status of `**1**`.
If the pattern is found in the output, the script prints a message indicating a successful deployment rollout using the `**echo**` command.
Finally, the script terminates. No error status is returned.
Overall, this script is useful for automatically checking the status of a Kubernetes deployment rollout, detecting when the rollout has failed, and rolling back to a previous deployment version to ensure that the Kubernetes application runs smoothly without interruption.

Add enviroment into pipeline

```
pipeline {
 agent any

 environment {
  deploymentName = "devsecops"
  containerName = "devsecops-container"
  serviceName = "devsecops-svc"
  imageName = "siddharth67/numeric-app:${GIT_COMMIT}"
  applicationURL = " http://devsecops-demo.eastus.cloudapp.azure.com/"
  applicationURI = "/increment/99"
 }
```

Edit pipeline:

```
stage('K8S Deployment - DEV') {
   steps {
    parallel(
     "Deployment": {
      withKubeConfig([credentialsId: 'kubeconfig']) {
        sh "bash k8s-deployment.sh"
        }
      },
      "Rollout Status": {
        withKubeConfig([credentialsId: 'kubeconfig']) {
          sh "bash k8s-deployment-rollout-status.sh"
        }
      }
    )
  }
}
```

Build pipeline.