Kubesec

OPA and Kubesec are both open-source tools that can be used to secure Kubernetes clusters. However, they have different strengths and weaknesses, and they are typically used together to provide a more comprehensive security solution.

OPA (Open Policy Agent) is a general-purpose policy engine that can be used to enforce a wide range of policies, including security policies. It is based on the Rego language, which is a powerful and expressive language for writing policies. OPA can be used to enforce policies on Kubernetes resources, as well as on other types of resources, such as files and databases.

Kubesec is a security tool for Kubernetes that focuses on identifying and remediating security vulnerabilities. It does this by scanning Kubernetes clusters for known vulnerabilities and then providing recommendations for how to fix them. Kubesec can also be used to enforce security policies, but it is not as flexible as OPA in this regard.

OPA and Kubesec can be used together to provide a more comprehensive security solution for Kubernetes clusters. OPA can be used to enforce policies, while Kubesec can be used to identify and remediate vulnerabilities. This combination of tools can help to protect Kubernetes clusters from a wide range of security threats.

Here are some of the key differences between OPA and Kubesec:

- OPA is a general-purpose policy engine, while Kubesec is a security tool for Kubernetes.
- OPA is based on the Rego language, while Kubesec uses its own proprietary language.
- OPA can be used to enforce policies on a wide range of resources, while Kubesec is focused on Kubernetes resources.
- OPA is more flexible than Kubesec in terms of how it can be used to enforce policies.

Here are some of the reasons why you might need to use both OPA and Kubesec:

- To get the benefits of both tools. OPA is a powerful policy engine that can be used to enforce a wide range of policies, while Kubesec is a security tool that can identify and remediate vulnerabilities.
- To cover different use cases. OPA is a general-purpose policy engine, while Kubesec is focused on Kubernetes resources. If you need to enforce policies on other types of resources, you will need to use OPA.
- To get the best of both worlds. OPA is more flexible than Kubesec in terms of how it can be used to enforce policies. However, Kubesec is easier to use and has a wider range of features. By using both tools, you can get the benefits of both worlds.

Question: If I use both of these tools to scan security for k8s, is there any overlap?
Answer:
Yes, there is a potential for overlap if you use both OPA and kubesec to scan for security issues in Kubernetes. This is because OPA already comes with some built-in policies for Kubernetes security, including policies related to secrets management, which may overlap with the functionality provided by kubesec.
However, the extent and significance of the overlap would depend on which policies you enable in OPA and how you configure kubesec. For example, if you disable the built-in Kubernetes security policies in OPA and only use it for other types of policies, you may not encounter much overlap with

kubesec. Similarly, if you only use kubesec for scanning secrets and not for other security checks, any overlap with OPA would be restricted to secret-related issues.
Overall, it's best to carefully consider which tools and policies you need for your Kubernetes security posture and configure them appropriately, to avoid duplication or unnecessary complexity.

Create file k8sec-scan.sh

```
#!/bin/bash
#kubesec-scan.sh
# using kubesec v2 api
scan_result=$(curl -sSX POST --data-binary @"k8s_deployment_service.yaml" https://v2.kubesec.io/scan)
scan_message=$(curl -sSX POST --data-binary @"k8s_deployment_service.yaml" https://v2.kubesec.io/scan | jq .[0].message -r )
scan_score=$(curl -sSX POST --data-binary @"k8s_deployment_service.yaml" https://v2.kubesec.io/scan | jq .[0].score )

# using kubesec docker image for scanning
# scan_result=$(docker run -i kubesec/kubesec:512c5e0 scan /dev/stdin < k8s_deployment_service.yaml)
# scan_message=$(docker run -i kubesec/kubesec:512c5e0 scan /dev/stdin < k8s_deployment_service.yaml | jq .[].message -r)
# scan_score=$(docker run -i kubesec/kubesec:512c5e0 scan /dev/stdin < k8s_deployment_service.yaml | jq .[].score)

  # Kubesec scan result processing
  # echo "Scan Score : $scan_score"
  if [[ "${scan_score}" -ge 4 ]]; then
    echo "Result:"
    echo "$scan_result"
    echo "Score is $scan_score"
    echo "Kubesec Scan $scan_message"
  else
    echo "Result:"
    echo "$scan_result"
    echo "Score is $scan_score, which is less than or equal to 5."
    echo "Scanning Kubernetes Resource has Failed"
    exit 1;
  fi;
```

This shell script, named **`kubesec-scan.sh`**, uses the **`kubesec`** tool to perform a security scan on a Kubernetes deployment and service configuration file (**`k8s_deployment_service.yaml`**).
To perform the scan, the script makes an HTTP POST request to the Kubesec API endpoint (**`https://v2.kubesec.io/scan`**) with the given configuration file as the input data. The scan result is stored in the **`scan_result`** variable, and other details such as the scan message and score are extracted using **`jq`** command and are stored in **`scan_message`** and **`scan_score`** variables respectively

The script then evaluates the scan score (**`scan_score`**) and if it's greater than or equal to 4, it outputs the scan result along with the score and message. Otherwise, it prints an error indicating that the scan has failed and exits with a non-zero code.
The script also provides an alternate method for performing the scan using the **`kubesec`** Docker image. This is commented out and can be used by swapping the comments with the **`curl`** method above.

What is jq ?
jq is short for "JSON Query". It is a command-line tool for processing and manipulating JSON data. jq is written in the C programming language and is available for most operating systems. It is free and open-source software.

jq can be used to extract specific data from a JSON file, filter out unwanted data, and convert JSON data into a different format. jq is often used in conjunction with other command-line tools to process JSON data. For example, jq can be used to extract data from a JSON file and then use that data to update a database.

`jq .[0].message -r ` is extracting the 'message' key from the first object in the JSON array (the security scan result), and with the -r flag, it is returning the raw output without quotes.
`jq .[0].score` is extracting the 'score' key from the same object.

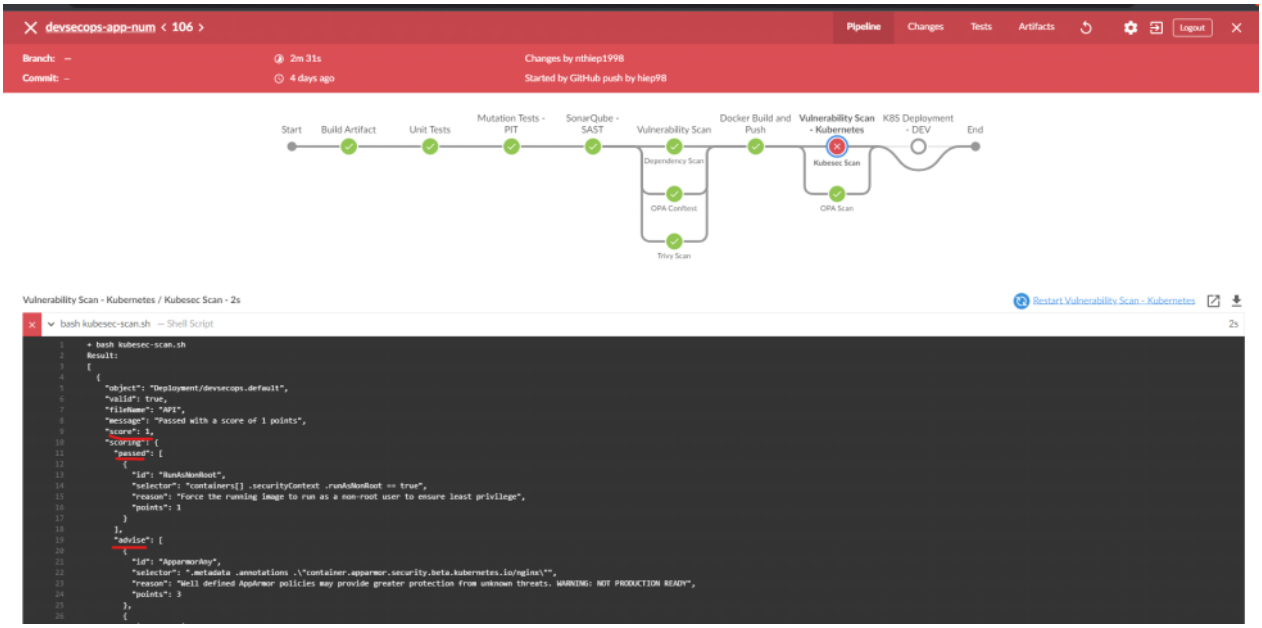
Why need use [0] in jq ?
In the given script, **`jq .[0].message -r`** and **`jq .[0].score`** are using the **`[0]`** index to extract information from the first object in the JSON array returned by the API.
The reason for using **`[0]`** is that the API response is an array of JSON objects, where each object represents a scan result for a Kubernetes resource. We are interested in the first object as it represents the scan result for the deployment and service YAML file that we have submitted for scanning.
If we do not use **`[0]`** and instead use **`jq .message`**, then **`jq`** will attempt to find the 'message' key from all the objects in the JSON array. This will not work in this case, as the 'message' key exists only inside the first object of the array. Therefore, we need to first select the first object using **`[0]`** and then extract information from it using the desired key.

Reference jq:
https://exercism.org/tracks/jq
https://www.baeldung.com/linux/jq-command-json

Edit jenkinsfile

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

And build



After build, The log will show your score, the criteria you achieved and advice to improve security for your k8s.

And base on result, we can add serviceAccountName and readOnlyRootFilesystem into file k8s_deployment_service.yaml to ensure it passes the kubesec test.

```
spec:
    serviceAccountName: default
    containers:
    - image: replace
      name: devsecops-container
      securityContext:
        runAsNonRoot: true
        runAsUser: 100
        readOnlyRootFilesystem: true
```

Because our script can't detect and auto-deploy when we update k8s_deployment_service.yaml file. So we will update the k8s-deployment.sh script

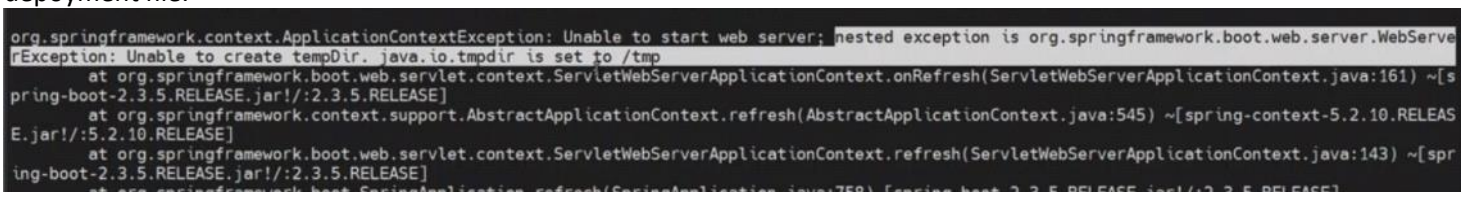
```
#!/bin/bash
#k8s-deployment.sh
DIFF_OUTPUT=$(kubectl diff -f k8s_deployment_service.yaml)
# if [ -n "$DIFF_OUTPUT" ]; then
#   # DIFF_OUTPUT is not empty, run your code here
# else
#   # DIFF_OUTPUT is empty, do nothing
# fi
#! $(kubectl diff -f k8s_deployment_service.yaml  >/dev/null 2>&1)

sed -i "s#replace#${imageName}#g" k8s_deployment_service.yaml
kubectl -n default get deployment ${deploymentName} > /dev/null
if [[ $? -ne 0 || ! $(kubectl diff -f k8s_deployment_service.yaml)="" ]]; then
   echo "deployment ${deploymentName} doesn't exist"
   kubectl -n default apply -f k8s_deployment_service.yaml
else
   echo "deployment ${deploymentName} exists"
   echo "image name - ${imageName}"
   kubectl -n default set image deploy ${deploymentName} ${containerName}=${imageName} --record=true
Fi
```

Alternatively, we can all add the below code at the end of the file to check the description of all the pod

```
# Get all pod names in current namespace
pods=$(kubectl get pods -o jsonpath='{.items[*].metadata.name}')
# Loop through each pod and describe its events
for pod in $pods; do
  echo "Getting events for pod: $pod"
  kubectl get events --field-selector involvedObject.name=$pod --all-namespaces | awk -v var="$pod" '$0 ~ var'
done
```

After update script and build again, If you see this error in your pod, you need update your k8s depoyment file.



Next, update volumes in your deployment file like below.

```
spec:
    volumes:
    - name: vol
      emptyDir: {}
    serviceAccountName: default
    containers:
    - image: replace
      name: devsecops-container
      volumeMounts:
      - mountPath: /tmp
        name: vol
      securityContext:
        runAsNonRoot: true
        runAsUser: 100
        readOnlyRootFilesystem: true
```

specifies an emptyDir volume named "vol". An emptyDir volume is a temporary volume that is created when a Pod is created and deleted when the Pod is deleted. The emptyDir volume is initially empty, and all containers in the Pod can read and write to the same files in the emptyDir volume.

The emptyDir volume is created on the node that the Pod is assigned to. The emptyDir volume is not persistent, and the data in the emptyDir volume is lost when the Pod is deleted.

The emptyDir volume is a good choice for temporary data storage, such as scratch space for a disk-based merge sort or checkpointing a long computation for recovery from crashes.

The emptyDir volume can be mounted in a container using the following syntax:

```
volumes:
  - name: vol
    emptyDir: {}
```

```
containers:
 - name: my-container
   image: busybox
   volumeMounts:
     - name: vol
       mountPath: /vol
```

In this example, the "my-container" container will have access to the emptyDir volume at the path "/vol".
In Kubernetes, curly braces `{}` with nothing inside them represent an empty object or an empty dictionary. In the context of an `emptyDir` volume in a Kubernetes deployment file, an empty dictionary `{}` indicates that there are no specialized configuration options for this particular `emptyDir`.
In other words, using an empty dictionary for an `emptyDir` volume means that Kubernetes should create a new, empty directory for the volume to be mounted on the container, using default settings. This directory is then isolated to the pod's lifecycle.

Reference: https://kubernetes.io/docs/concepts/storage/volumes/