



CERTSWARRIOR

# Linux Foundation CKAD

**Certified Kubernetes Application Developer (CKAD)**

**Questions&AnswersPDF**

**ForMoreInformation:**

**<https://www.certsvarrior.com/>**

## **Features:**

- 90DaysFreeUpdates
- 30DaysMoneyBackGuarantee
- InstantDownloadOncePurchased
- 24/7OnlineChat Support
- ItsLatestVersion

# Latest Version: 6.0

## Question: 1

You are running a critical application on Kubernetes, and your security team has mandated the use of Pod Security Policies (PSPs) to enhance the security posture of your cluster. You have a Deployment that uses a privileged container for certain tasks. However, PSPs restrict the use of privileged containers. Describe how you can address this challenge while adhering to the security requirements imposed by PSPs.

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

### **Solution (Step by Step) :**

1. Identify the Privileged Container Tasks: Analyze your Deployment and identify the specific tasks performed by the privileged container. These tasks might involve accessing host resources like devices, manipulating network settings, or interacting with the host kernel directly.
2. Explore Alternative Solutions: Instead of relying on privileged containers, consider alternative approaches to achieve the desired functionality:
  - Host Network: If the task requires direct network access, consider using the 'hostNetwork' feature. This grants the container access to the host's network stack but doesn't require privileged mode.
  - HostPath Volumes: If the task involves accessing host files or directories, mount them into the container using 'hostPath' volumes.
  - SecurityContext: Explore the 'securityContext' options for containers. Options like 'capabilities' can grant limited access to specific host resources.
  - Dedicated Service Account: Assign a dedicated Service Account to the Deployment with limited permissions, ensuring the container can only access the required resources.
3. Implement PSP with Allowlist:
  - Create a PSP that defines a restricted set of security rules. This PSP should allow:
    - The specific tasks that require privileged operations.
    - Other essential security measures like restricting host network access, SELinux, and AppArmor configurations.
  - Apply the PSP to the namespace where your Deployment is running.
4. Update Deployment: Modify your Deployment configuration to utilize the alternative solutions identified in step 2.
  - Replace the privileged container with a non-privileged container.
  - Utilize 'hostNetwork', 'hostPath' volumes, or 'securityContext' options as needed.
  - Ensure the Deployment is properly configured to use the dedicated Service Account.
5. Test and Validate: Verify that the modified Deployment functions as expected and that the chosen alternative solutions meet the original requirements. Additionally, ensure that the PSP is enforcing the desired security policies.

Example:

Original Deployment (with privileged container):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: my-image:latest
          securityContext:
            privileged: true
```

Modified Deployment (using host network):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: my-image:latest
          securityContext:
            privileged: false
          hostNetwork: true
```

PSP with allowlist:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: my-psp
spec:
  privileged: false
  hostNetwork: true
  hostPID: false
  hostIPC: false
  selinux:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  readOnlyRootFilesystem: false
  volumes:
  - hostPath: {}
```

Note: This example illustrates one approach to address the challenge. The specific solution will depend on the nature of the privileged container tasks and the security requirements enforced by your PSP. It's essential to thoroughly understand your application's needs and implement the appropriate security measures to ensure both security and functionality. ,

## Question: 2

You are running a web application on a Kubernetes cluster, and you want to ensure that the container running your application is protected from potential security vulnerabilities. You are specifically concerned about unauthorized access to the container's filesystem. Explain how you would implement AppArmor profiles to restrict access to the container's filesystem.

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

### **Solution (Step by Step) :**

1. Define the AppArmor Profile:

- Create a new AppArmor profile file, for example, 'nginx-apparmor.conf', within your Kubernetes configuration directory.
- Within this file, define the restrictions for the container.

- For instance, to allow access to specific directories and files:

```
# include common AppArmor profile
include /etc/apparmor.d/abstractions/base/nginx.apparmor
# Allow access to specific directories
/var/www/html r,
/etc/nginx r,
# Allow access to specific files
/etc/nginx/nginx.conf r,
/usr/sbin/nginx r,
# Deny access to all other files and directories
Deny
```

2. Load the AppArmor Profile:

- Use the 'create configmap' command to create a ConfigMap containing your AppArmor profile:  
Bash

```
kubectl create configmap nginx-apparmor-profile --from-file=nginx-apparmor.conf
```

3. Apply the Profile to Your Deployment:

- Update your Deployment YAML file to include the AppArmor profile:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          securityContext:
            # Enable AppArmor and specify the profile name
            appArmor: nginx-apparmor-profile
            # ... (rest of your Deployment YAML)
```

4. Restart the Pods:

- Apply the updated Deployment YAML using 'kubectl apply -f nginx-deployment.yaml'  
- The updated deployment will restart the pods with the new AppArmor profile.

5. Verify the Profile:

- Check the status of the pods with 'kubectl describe pod'  
- Look for the "Security Context" section and verify that the AppArmor profile is correctly applied.

6. Test the Restrictions:

- Try to access files or directories that are not allowed by your AppArmor profile.
- This will help you confirm that the profile is effectively restricting access.

### Question: 3

You are running a microservices application on Kubernetes, and you need to restrict the communication between your services to specific ports. For example, your 'frontend' service should only be allowed to communicate with the 'backend' service on port 8080. How would you configure this using NetworkPolicy in Kubernetes?

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

**Solution (Step by Step) :**

1. Define the NetworkPolicy:

- Create a new YAML file (e.g., 'frontend-network-policy.yaml') to define the network policy.
- Specify the name of the NetworkPolicy and the namespace where it will be applied.
- Include the following elements within the 'spec' section:
  - 'podSelector' to target the 'frontend' pods.
  - 'ingress' section to define inbound traffic rules.
  - 'egress' section to define outbound traffic rules.

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-network-policy
  namespace: your-namespace
spec:
  podSelector:
    matchLabels:
      app: frontend
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: backend
      ports:
        - protocol: TCP
          port: 8080
  egress:
    - to:
      - ipBlock:
          cidr: 0.0.0.0/0
      ports:
        - protocol: TCP
          port: 8080

```

2. Apply the NetworkPolicy:

- Apply the NetworkPolicy to your cluster using the following command:

bash

```
kubectl apply -f frontend-network-policy.yaml
```

3. Verify the NetworkPolicy:

- Use the 'kubectl get networkpolicy' command to list the applied NetworkPolicies and confirm the status.

4. Test the Restrictions:

- From a 'frontend' pod, attempt to connect to the 'backend' service on port 8080.
- Attempt to connect to other services or ports on the backend or external networks.
- Verify that the communication restrictions defined in the NetworkPolicy are working as expected.

## Question: 4

You are running a Kubernetes cluster with a limited number of nodes, and you want to deploy a new application that requires a lot of resources. You are concerned about potential resource contention and performance issues with other existing applications. How would you use resource quotas to manage resource usage and prevent potential issues?

A. See the solution below with Step by Step Explanation.

## Answer: A

Explanation:

### **Solution (Step by Step) :**

#### 1. Create a Resource Quota:

- Create a new YAML file (e.g., 'resource-quota.yaml') to define your resource quota.
- Specify the name of the resource quota and the namespace where it will be applied.
- Define the resource limits for the quota. For instance, you can set limits for CPU, memory, pods, services, etc.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: resource-quota
  namespace: your-namespace
spec:
  limits:
    requests.cpu: "2"
    requests.memory: "4Gi"
    pods: "10"
    services: "5"
```

#### 2. Apply the Resource Quota:

- Apply the resource quota to your cluster using the following command:

bash

```
kubectl apply -f resource-quota.yaml
```

#### 3. Verify the Resource Quota:

- Use the 'kubectl get resourcequota' command to list the applied resource quotas and confirm their status.

#### 4. Deploy Applications with Resource Requests:

- When deploying your applications, ensure that you specify resource requests and limits in your Deployment YAML files.
- This will help enforce the resource limits defined by your quota.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: resource-intensive-app
spec:
  replicas: 2
  template:
    spec:
      containers:
      - name: resource-intensive-app
        image: example/resource-intensive-app:latest
        resources:
          requests:
            cpu: "1"
            memory: "2Gi"
          limits:
            cpu: "1.5"
            memory: "3Gi"
```

#### 5. Monitor Resource Usage:

- Use monitoring tools (e.g., Prometheus, Grafana) to track resource usage in your namespace and ensure that applications are staying within the resource limits defined by your quota.

### Question: 5

You are deploying a sensitive application that requires strong security measures. You need to implement a solution to prevent unauthorized access to the container's runtime environment. How would you use Seccomp profiles to enforce security policies at the container level?

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

#### **Solution (Step by Step) :**

##### 1. Create a Seccomp Profile:

- Create a new YAML file (e.g., 'seccomp-profile.yaml') to define your Seccomp profile.
- Specify the name of the Seccomp profile and the namespace where it will be applied.
- Define the allowed syscalls for the container. You can use the 'seccomp' tool or the 'k8s.io/kubernetes/pkg/security/apparmor/seccomp' package to generate the profile.

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  name: seccomp-profile
spec:
  selinuxContext:
    type: RuntimeDefault
  seccompProfile:
    type: Localhost
    localhostProfile:
      # Define the allowed syscalls
      # For example, allow only a few essential syscalls
      # for a minimal runtime environment
      allow:
        - read
        - write
        - open
        - close
        - fstat
        - stat
        - lstat
        - ioctl
        - mmap
        - mprotect
        - munmap
        - fcntl
        - getpid
        - getppid
        - getuid
        - geteuid
        - getgid
        - getegid
        - clock_gettime
        - gettimeofday
        - time
        - nanosleep
        - setrlimit
        - getrlimit
        - prctl
        - brk
        - exit
        - exit_group
        - kill
        - sigaction
        - sigprocmask
        - getuid
        - getgid
        - getppid
        - getpid
  default:
    - ALLOW
```

2. Apply the Seccomp Profile:

- Apply the Seccomp profile to your cluster using the following command:

bash

kubectl apply -f seccomp-profile.yaml

3. Deploy Applications with Seccomp Profile:

- Update your Deployment YAML file to include the Seccomp profile:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sensitive-app
spec:
  replicas: 2
  template:
    spec:
      containers:
      - name: sensitive-app
        image: example/sensitive-app:latest
        securityContext:
          # Enable Seccomp and specify the profile name
          seccompProfile:
            type: Localhost
            localhostProfile: seccomp-profile
```

4. Verify the Seccomp Profile:

- Check the status of the pods with 'kubectl describe pod

- Look for the "Security Context" section and verify that the Seccomp profile is correctly applied.

5. Test the Restrictions:

- Try to access system resources or make syscalls that are not allowed by your Seccomp profile.

- Verify that the profile is effectively restricting the container's access to system resources.

## Question: 6

You need to implement a strategy to manage and control the access of pods to specific resources in your Kubernetes cluster. Explain how you would use PodSecurityPolicies to enforce fine-grained access control.

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

**Solution (Step by Step) :**

1 . Create a PodSecurityPolicy:

- Create a new YAML file (e.g., 'pod-security-policy.yaml') to define your PodSecurityPolicy.

- Specify the name of the PodSecurityPolicy and the namespace where it will be applied.
- Define the security policies for the PodSecurityPolicy. You can use the 'kubectl create -f pod-security-policy.yaml' command to apply the PodSecurityPolicy.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted-pod-security-policy
spec:
  # Control pod creation based on user and group IDs
  runAsUser:
    rule: "RunAsAny"
  # Control the capabilities allowed in containers
  # Restrict the allowed capabilities
  # The "CAP_NET_BIND_SERVICE" capability enables a container to bind to privileged ports, which is usually undesirable.
  # Removing this capability adds an additional layer of security to the cluster.
  # The capability "CAP_CHMOD" allows containers to change the ownership of files and directories within the pod. This can be a security risk if the container is compromised.
  # Removing this capability helps to limit the potential damage that a compromised container can cause.
  # Note that this is a general example and you may need to adjust these capabilities based on your specific security requirements.
  allowedCapabilities:
    - "CAP_NET_BIND_SERVICE"
    - "CAP_CHMOD"
  # Control the SELinux context of pods
  selinux:
    rule: "RunAsAny"
  # Control the volume types that pods can use
  volumes:
    - "hostPath"
    - "emptyDir"
    - "projected"
    - "secret"
    - "configMap"
    - "downwardAPI"
    - "persistentVolumeClaim"
  # Control the ability to access host network and ports
  hostNetwork: false
  # Control the ability to bind to privileged ports
  # Restrict the ability of containers to bind to privileged ports (ports < 1024).
  # This helps to prevent applications from interfering with system services.
  privileged: false
  # Control the allowed security context of pods
  # The "runAsAny" option allows containers to run with any user ID and group ID.
  # Setting this to "RunAsAny" might be a security risk if the container is compromised.
  # You can restrict this to specific user IDs and group IDs or use other options like "RunAsNonRoot" or "RunAsUser" to control user and group IDs.
  # The securityContext field controls several security-related settings for the pod, such as the ability to run as a privileged user, the container's network access, and the container's security context.
  # To make your cluster more secure, you can configure the securityContext field to restrict these settings. For example, you can prevent containers from running as privileged users or from accessing host networking.
  fsGroup:
    rule: "RunAsAny"
  supplementalGroups:
    rule: "RunAsAny"
  readOnlyRootFilesystem: false
```

### 3. Apply the PodSecurityPolicy to Deployments:

- Update the 'podSecurityContext' field in your Deployment YAML to specify the PodSecurityPolicy.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app:latest
          # Add this section to the deployment's template
          securityContext:
            # The name of the PodSecurityPolicy to apply to the pod
            podSecurityPolicy: "restricted-pod-security-policy"
            # ... (rest of your deployment YAML)
```

### 4. Verify the PodSecurityPolicy:

- Use the 'kubectl get podsecuritypolicy' command to list the applied PodSecurityPolicies and confirm their status.

### 5. Test the Restrictions:

- Try to create pods that violate the rules defined in the PodSecurityPolicy.
- Verify that the PodSecurityPolicy is effectively preventing the creation of pods that do not meet the defined security policies.,

## Question: 7

You are running a web application in a Kubernetes cluster. You have a deployment named 'web- app' with two replicas. You need to implement a Network Policy that allows only traffic from pods with the label app: database' to access the 'web-app' deployment on port 8080. You also need to block all other traffic to the 'web-app' deployment.

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

**Solution (Step by Step) :**

1. Create the Network Policy:

- Create a YAML file named 'web-app-network-policy.yaml' with the following content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: web-app-network-policy
  namespace: default # Replace with your namespace
spec:
  podSelector:
    matchLabels:
      app: web-app
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: database
      ports:
        - protocol: TCP
          port: 8080
  egress:
    - to:
        - ipBlock:
            cidr: 0.0.0.0/0
      ports:
        - protocol: TCP
          port: 8080
```

2. Apply the Network Policy:

- Apply the Network Policy to your cluster:

bash

```
kubectl apply -f web-app-network-policy.yaml
```

3. Verify the Network Policy:

- Verify that the Network Policy has been applied correctly by listing the Network Policies in your namespace:

bash

```
kubectl get networkpolicies -n default # Replace with your namespace
```

You should see the 'web-app-network-policy' listed.

4. Test the Network Policy:

- From a pod with the label 'app: database' , try to access the 'web-app' deployment on port 8080. This should be successful.

- From any other pod, try to access the 'web-app' deployment on port 8080. This should be blocked.

- The 'podSelector' in the Network Policy specifies that it applies to pods with the label 'app: web-app'.

- The 'ingress' section defines the allowed incoming traffic. In this case, it allows traffic from pods with the label

'app: database' on port 8080.

- The 'egress' section defines the allowed outgoing traffic. In this case, it allows all outgoing traffic except on port 8080. This ensures that only pods with the 'app: database' label can access the 'web-app' deployment on port 8080.

Note:

- You may need to update the 'namespace' in the Network Policy YAML file to match the namespace where your 'web-app' deployment is running.

- Make sure that pods with the label 'app: database' are allowed to access the 'web-app' deployment by other means, such as Service or Ingress, if needed.,

## Question: 8

You have a Kubernetes cluster with a Deployment named 'my-app' that runs a web application. You want to restrict access to this application to only specific users within your organization. How would you use Service Accounts and RBAC to implement this?

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

**Solution (Step by Step) :**

1 . Create a Service Account:

- Create a new Service Account specifically for your application:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-app-sa
```

- Apply this YAML file using 'kubectl apply -f my-app-sa.yaml'.

2. Create a Role:

- Define a Role that grants specific permissions to the Service Account. For example, you might want to grant read access to the Deployment's secrets:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: my-app-reader
  namespace: default
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["core"]
  resources: ["secrets"]
  verbs: ["get"]
```

- Apply this YAML file using 'kubectl apply -f my-app-reader.yaml'

3. Bind the Role to the Service Account:

- Create a RoleBinding that associates the 'my-app-reader' Role with the 'my-app-sa' Service Account:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: my-app-sa-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: my-app-reader
subjects:
- kind: ServiceAccount
  name: my-app-sa
  namespace: default
```

- Apply this YAML file using 'kubectl apply -f my-app-sa-binding.yaml'

4. Update the Deployment:

- Update the 'my-app' Deployment to use the new Service Account:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      serviceAccountName: my-app-sa
```

- Apply the updated Deployment configuration using 'kubectl apply -f my-app.yaml'.

#### 5. Verify:

- Ensure that pods within the 'my-app' Deployment are running with the correct Service Account. You can use

'kubectl get pods -l app=my-app -o wide' to inspect the pod details.

#### 6. Restricting Access to Specific Users:

- To restrict access to the application to specific users within your organization, you would need to:
- Configure a more granular Role to grant specific access levels (e.g., read-only, edit, etc.).
- Use a Kubernetes authentication provider (such as OAuth2 or OpenID Connect) to authenticate and authorize users.
- Bind the Role to the user's identity, ensuring they have the appropriate permissions.

Important Note: This example provides a basic setup for RBAC with Service Accounts. In real-world scenarios, you might need to configure more complex RBAC rules to address your specific security requirements and user access control policies.]

## Question: 9

You have a web application that uses two different services: 'frontend' and 'backend'. You want to restrict access to the 'backend' service from all pods except those with the label 'app: frontend'. How would you configure NetworkPolicy to achieve this?

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-policy
  namespace:
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: frontend

```

- Replace with your actual namespace.

2. Apply the NetworkPolicy:

- Run the following command to apply the NetworkPolicy:

bash

kubectl apply -f backend-networkpolicy.yaml

- This NetworkPolicy defines a policy for pods with the label 'app: backend'.

- The 'ingress' rule allows traffic only from pods with the label 'app: frontend'.

- All other pods will be blocked from accessing the 'backend' service.

This ensures that only the frontend service can communicate with the 'backend' service. ,

## Question: 10

You are building a microservices application on Kubernetes, where two services, and 'service-b' , need to communicate with each other securely. 'Service-b' needs to expose a secure endpoint that is only accessible by 'service-a'. Describe how you would implement this using Kubernetes resources, including the configuration for the 'service-b' endpoint.

A. See the solution below with Step by Step Explanation.

**Answer: A**

Explanation:

**Solution (Step by Step) :**

1. Define a Kubernetes Secret:

- Create a Kubernetes secret to store the certificate and key pair for 'service-W'. This secret will be used to secure the communication.

- Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: service-b-tls
type: kubernetes.io/tls
data:
  tls.crt:
  tls.key:
```

2. Configure 'service-b' Deployment:

- Define a Deployment for 'service-b' , specifying a container that uses the secret for TLS.
- Ensure that the container has the required dependencies and configuration to use TLS.
- Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-b-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-b
  template:
    metadata:
      labels:
        app: service-b
    spec:
      containers:
        - name: service-b
          image: your-image:latest
          ports:
            - containerPort: 8443
          volumeMounts:
            - name: service-b-tls
              mountPath: /var/tls/
      volumes:
        - name: service-b-tls
          secret:
            secretName: service-b-tls
```

3. Define a Kubernetes Service for 'service-b'.

- Create a Service for 'service-b' that exposes the secure endpoint on a specific port (e.g., 8443) and uses the LoadBalancer' type for external access.
- Use the 'targetPort' field to specify the container port that 'service-b' is listening on.
- Example:

```
apiVersion: v1
kind: Service
metadata:
  name: service-b-service
spec:
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8443
      targetPort: 8443
  selector:
    app: service-b
```

#### 4. Configure 'service-a' Deployment:

- Define a Deployment for 'service-a', specifying a container that uses the secret for TLS when connecting to service-W.
- Example:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-a-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-a
  template:
    metadata:
      labels:
        app: service-a
    spec:
      containers:
        - name: service-a
          image: your-image:latest
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: service-b-tls
              mountPath: /var/tls/
          volumes:
            - name: service-b-tls
              secret:
                secretName: service-b-tls

```

#### 5. Update 'service-a' Container Configuration:

- Within the 'service-a' container, ensure the application is configured to use the certificate and key from the mounted volume ('/var/tls/') for secure communication with 'service-b'.

#### 6. Verify Secure Communication:

- Use 'kubectl get pods' to check the status of both 'service-a' and 'service-b' pods.
- Test the communication between 'service-a' and 'service-b' by sending requests from the 'service-a' pod to the secure endpoint of 'service-b'.
- Verify that the communication is secure and that 'service-a' can successfully access the endpoint.

#### Notes:

- You may need to adjust the port numbers and image names in the examples to match your specific setup.
- Make sure you have the certificate and key in the correct format and base64 encoded before creating the Secret.
- You can also use other methods like a Service Account and Role-Based Access Control (RBAC) to restrict access to the secure endpoint, if needed.
- This is a simplified example and additional security measures may be required based on your application's requirements.



# CERTSWARRIOR

## *FULL PRODUCT INCLUDES:*

Money Back Guarantee



Instant Download after Purchase



90 Days Free Updates



PDF Format Digital Download



24/7 Live Chat Support



Latest Syllabus Updates



For More Information – Visit link below:

**<https://www.certswarrior.com>**

**16 USD Discount Coupon Code: U89DY2AQ**