Jonathan

Follow 29 Followers About

Certified Kubernetes Security Specialist (CKS) Preparation Part 5 — Microservice Vulnerabilities

Jonathan Mar 3 · 7 min read

If you have not yet checked the previous parts of this series, please go ahead and check Part1, Part2, Part3 and Part4.

In this article, I would focus on the preparation around **microservice vulnerabilities** in CKS certification exam.

Manage Secrets

For knowing how Kubernetes store secrets and how Pods use secrets, we first create 2 secrets (secret1 and secret2).

- *kubectl create secret generic secret1 — from-literal=username=jonw*
- *kubectl create secret generic secret2 — from-literal=password=12345678*
- *kubectl get secrets*

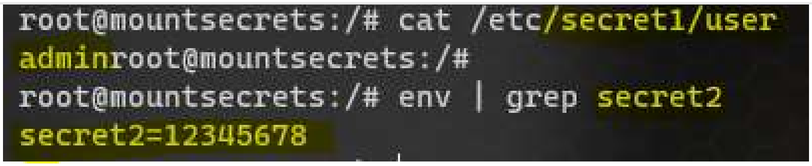
```
jonw@CKS-Master:~$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-xd7dp   kubernetes.io/service-account-token   3      46h
podsa-token-bmkk9     kubernetes.io/service-account-token   3      46h
secret1               Opaque                                1      19h
secret2               Opaque                                1      19h
```

- *kubectl run mountsecrets — image=nginx -o yaml — dry-run=client > pod-mountsecrets.yaml*
- *nano pod-mountsecrets.yaml*
- *kubectl create -f pod-mountsecrets.yaml*

```
jonw@CKS-Master:~$ cat pod-mountsecrets.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: mountsecrets
  name: mountsecrets
spec:
  containers:
  - image: nginx
    name: mountsecrets
    resources: {}
    env:
      - name: secret2
        valueFrom:
          secretKeyRef:
            name: secret2
            key: pass
    volumeMounts:
    - name: secret1
      mountPath: "/etc/secret1"
      readOnly: true
  volumes:
  - name: secret1
    secret:
      secretName: secret1
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

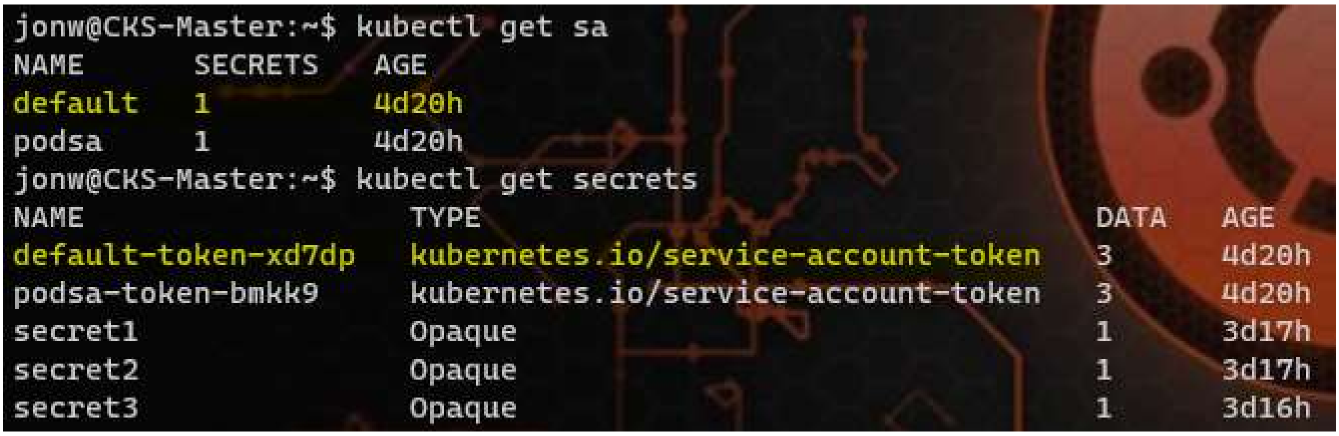
After the pod gets created, we could execute into a shell inside of the Pod and try to read the secrets.

- *kubectl exec mountsecrets -it — bash*
- *cat /etc/secret1/user*



The permission each Pod leveraging is the default service account created in the namespace. In this example, it would be service account "default" in the namespace "default". Each service account would automatically generate a token, the name would be similar to "<service account name>-token-<randon-generated-string>".

- *kubectl get sa*
- *kubectl get secrets*



If executing into one of the shell within Pods(containers), the token would be shown under file mount.

- *mount | grep serviceaccount*
- *cat /run/secrets/kubernetes.io/serviceaccount/token*

If any endpoint needs to be access with this token, this could be easily done with

Open in app

ignore the fact that it is returning HTTP status 403 as the service account does not have proper permissions.**

If we try to get secrets through Docker, we would need to get to the node that hosts this pod. In this case, it would be one of the worker nodes. After getting into the terminal of the worker node, we locate the container by using the image the pod leverages and execute approximately the same commands executed in K8s pods.

- *docker ps | grep nginx*
- *docker exec -it <container ID> sh*
- *cat <secret file path>/<secret key>*
- *cat /run/secrets/kubernetes.io/serviceaccount/token*
- *curl <target endpoint> -H "Authorization: Bearer <token content>" -k*

If we try to get the secrets from ETCD, we first query all the certificates and keys kube-apiserver uses to communicate with ETCD.

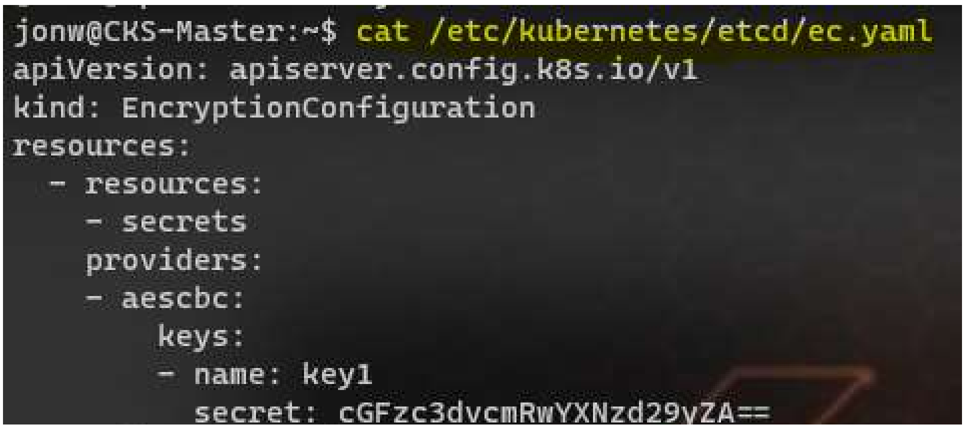
- *cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep etcd*

Then, we execute ETCD CLI with the queried certificates and keys above to get the secrets are showing in plain format.

- *ETCDCTL_API=3 etcdctl — cacert="/etc/kubernetes/pki/etcd/ca.crt" — cert="/etc/kubernetes/pki/apiserver-etcd-client.crt" — key="/etc/kubernetes/pki/apiserver-etcd-client.key" get /registry/secrets/<namespace>/<secret name>*

By now, we understand it would be "safer" if we could have ETCD encrypted even at rest and the way to do that is by enabling it in kube-apiserver configuration file. First, we create an ETCD encryption configuration file somewhere under /etc/kubernetes. In this case, we would create a file named "ec.yaml" in a folder named "etcd" under that path. For the secret section, please use base 64 encoded form of whatever string provided. Please check here for more ETCD encryption information.

- *cd /etc/kubernetes*
- *mkdir etcd*
- *nano etcd/ec.yaml*



After creating the ETCD configuration file, head over to configure kube-apiserver file for using the newly created configuration file. Also, remember to add a hostVolume and volumeMount in the bottom part of the configuration file.

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

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 192.168.1.4:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --encryption-provider-config=/etc/kubernetes/etcd/ec.yaml
    - --advertise-address=192.168.1.4
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
```

```
    volumeMounts:
    - mountPath: /etc/ssl/certs
      name: ca-certs
      readOnly: true
    - mountPath: /etc/ca-certificates
      name: etc-ca-certificates
      readOnly: true
    - mountPath: /etc/kubernetes/pki
      name: k8s-certs
      readOnly: true
    - mountPath: /usr/local/share/ca-certificates
      name: usr-local-share-ca-certificates
      readOnly: true
    - mountPath: /usr/share/ca-certificates
      name: usr-share-ca-certificates
      readOnly: true
    - mountPath: /etc/kubernetes/etcd
      name: k8s-etcd
      readOnly: true
```

```
  - hostPath:
      path: /etc/ssl/certs
      type: DirectoryOrCreate
    name: ca-certs
  - hostPath:
      path: /etc/ca-certificates
      type: DirectoryOrCreate
    name: etc-ca-certificates
  - hostPath:
      path: /etc/kubernetes/pki
      type: DirectoryOrCreate
    name: k8s-certs
  - hostPath:
      path: /usr/local/share/ca-certificates
      type: DirectoryOrCreate
    name: usr-local-share-ca-certificates
  - hostPath:
      path: /usr/share/ca-certificates
      type: DirectoryOrCreate
    name: usr-share-ca-certificates
  - hostPath:
      path: /etc/kubernetes/etcd
      type: DirectoryOrCreate
    name: k8s-etcd
```

Wait for the kube-apiserver to restart and test whether newly created secrets are now being encrypted even at rest. So, secret3 is created after enabling the at-rest encryption.

- *ETCDCTL_API=3 etcdctl — cacert="/etc/kubernetes/pki/etcd/ca.crt" — cert="/etc/kubernetes/pki/apiserver-etcd-client.crt" — key="/etc/kubernetes/pki/apiserver-etcd-client.key" get /registry/secrets/<namespace>/<secret name>*

```
jonw@CKS-Master:~$ sudo ETCDCTL_API=3 etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-client.key get /registry/secrets/default/secret3
/registry/secrets/default/secret3
k8s:enc:aescbc:v1:key1:…
```

Container Runtime Interface (CRI)

place. Before open container initiative (OCI) proposing to have CRI, the communication between containers and Kubernetes (K8s) is relying on dockershim/rkt provided and maintained by Docker. However, when containers and K8s are getting more and more sophisticated, the maintenance cost of dockershim/rkt becomes higher and higher. Therefore, having an interface that opens to the open source community and for solely dealing with container runtime becomes the answer to this challenging situation.

For CKS exam, we would need to know how to create RuntimeClass and how to create Pods using the specific RuntimeClass to communicate with K8s. For more information, please check this site.

- *kubectl create -f <runtimeclass.yaml>*

```
apiVersion: node.k8s.io/v1  # RuntimeClass is defined in the node.k8s.io API group
kind: RuntimeClass
metadata:
  name: myclass  # The name the RuntimeClass will be referenced by
  # RuntimeClass is a non-namespaced resource
handler: myconfiguration  # The name of the corresponding CRI configuration
```

Image Credit: kubernetes.io

- *kubectl get runtimeclass*
- *kubectl create -f <pod.yaml>*

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  runtimeClassName: myclass
  # ...
```

Image Credit: kubernetes.io

Container with Proper Permissions

not set with proper permissions, the hosting environment are exposed to potential threat. Luckily, there are many configuration that could be implemented inside container to not running into that situation.

runAs

By adding these security contexts, whenever administrators execute into one of the Pod's terminal, the associated identity would be user:1000 and group:3000. Please check here for more information.

- *kubectl create -f <pod.yaml>*

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: runas
  name: runas
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
  containers:
  - command:
    - sh
    - -c
    - sleep 1d
    image: busybox
    name: runas
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

- *kubectl exec <pod name> -it — sh*

- *id*

```
jonw@CKS-Master:~$ kubectl exec runas -it -- sh
/ $ id
uid=1000 gid=3000
```

runAsNonRoot

it would need root access to run with the image. Please check here for more information.

- *kubectl create -f <pod.yaml>*

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: runasnonroot
  name: runasnonroot
spec:
  containers:
  - command:
    - sh
    - c
    - sleep 1d
    image: busybox
    name: runasnonroot
    resources: {}
    securityContext:
      runAsNonRoot: true
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

- *kubectl describe pod <pod name>*

```
Events:
  Type     Reason          Age                From               Message
  ----     ------          ----               ----               -------
  Normal   Scheduled       7s                 default-scheduler  Successfully assigned default/runasnonroot to cks-worker
  Normal   Pulled          4s                 kubelet            Successfully pulled image "busybox" in 1.311995533s
  Normal   SandboxChanged  4s                 kubelet            Pod sandbox changed, it will be killed and re-created.
  Normal   Pulling         1s (x3 over 6s)    kubelet            Pulling image "busybox"
  Warning  Failed          1s (x2 over 4s)    kubelet            Error: container has runAsNonRoot and image will run as root
  Normal   Pulled          1s                 kubelet            Successfully pulled image "busybox" in 1.376420132s
```

privileged

There is no apparent way to know whether the container is running in privileged state or not. However, this could be tested by the possibility of executing some privileged actions.

- *kubectl create -f <pod.yaml>*

```
apiVersion: v1
```

```
  labels:
    run: privileged
  name: privileged
spec:
  containers:
  - command:
    - sh
    - -c
    - sleep 1d
    image: busybox
    name: runas
    resources: {}
    securityContext:
      privileged: true
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

- *kubectl exec <pod name> -it — sh*
- *sysctl kernel.hostname=whatever*

```
jonw@CKS-Master:~$ kubectl exec privileged -it -- sh
/ # sysctl kernel.hostname=whatever
kernel.hostname = whatever
```

allowPrivilegeEscalation

By checking /proc/1/status → NoNewPrivs, we would get the information on whether the Pod could execute privileged actions, including actions shown in the previous section "sysctl kernel.hostname=whatever". If the Pod is allowed to have privilege escaltion, there should be no issues.

- *kubectl create -f <pod.yaml>*

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: allowprivilegeescalation
  name: allowprivilegeescalation
spec:
  containers:
  - command:
    - sh
```

```
    name: runas
    resources: {}
    securityContext:
      allowPrivilegeEscalation: true
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

- *kubectl exec <pod name> -it — sh*
- *cat /proc/1/status | grep NoNewPrivs*

```
jonw@CKS-Master:~$ kubectl exec allowprivilegeescalation -it -- sh
/ # cat /proc/1/status
Name:   sleep
Umask:  0022
State:  S (sleeping)
Tgid:   1
Ngid:   0
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 64
Groups: 10
NStgid: 1
NSpid:  1
NSpgid: 1
NSsid:  1
VmPeak:     1300 kB
VmSize:     1300 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:         4 kB
VmRSS:         4 kB
RssAnon:               4 kB
RssFile:               0 kB
RssShmem:              0 kB
VmData:       36 kB
VmStk:       132 kB
VmExe:       888 kB
VmLib:         0 kB
VmPTE:        28 kB
VmSwap:        0 kB
HugetlbPages:          0 kB
CoreDumping:    0
THP_enabled:    1
Threads:        1
SigQ:   1/31705
```

```
SigIgn: 0000000000000004
SigCgt: 0000000000000000
CapInh: 00000000a80425fb
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
CapBnd: 00000000a80425fb
CapAmb: 0000000000000000
NoNewPrivs:     0
```

PodSecurityPolicy

First, we enable kube-apiserver to apply Pod security policy within its configuration file. Then, we create one Pod security policy to determine what actions could be performed by Pod and what could not. Please check here for more information.

- *sudo nano /etc/kubernetes/manifests/kube-apiserver.yaml*

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 192.168.1.4:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --encryption-provider-config=/etc/kubernetes/etcd/ec.yaml
    - --advertise-address=192.168.1.4
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction,PodSecurityPolicy
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
```

- *kubectl create -f psp.yaml*

```
apiVersion: policy/v1beta1
```

```
  privileged: false  # Don't allow privileged pods!
  allowPrivilegeEscalation: false
  # The rest fills in some required fields.
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

When try deploying another pods allowing privilege escalation, the console would return error message because it is contradicting to Pod security policy that was applied.

- *kubectl create -f <pod.yaml>*

```
jonw@CKS-Master:~$ kubectl create -f pod-allowPrivilegeEscalation.yaml
Error from server (Forbidden): error when creating "pod-allowPrivilegeEscalation.yaml": pods "allowprivilegeescal
ation" is forbidden: PodSecurityPolicy: unable to admit pod: [spec.containers[0].securityContext.allowPrivilegeEs
calation: Invalid value: true: Allowing privilege escalation for containers is not allowed]
```

mTLS

mTLS stands for mutual authentication, meaning client authenticates server and server does the same to client. This Medium article provides a pretty clear explanation how it works. Basically, whenever we are putting client certificate and client key in the command like "curl" or "xxxxctl", there is a high possibility that the communication is applying mTLS. In the Medium article above, if we follow all the steps until the end, we should be having an Ingress YAML set up somewhat like below.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/auth-tls-verify-client: \"on\"
    nginx.ingress.kubernetes.io/auth-tls-secret: \"default/my-certs\"
  name: meow-ingress
  namespace: default
spec:
  rules:
  - host: meow.com
    http:
```

```
          servicePort: 80
        path: /
  tls:
  - hosts:
    - meow.com
    secretName: my-certs
```

If we try to curl the HTTPS website without client credentials.

** The NGINX Ingress is setup to be exposed with NodePort

** The resolved public IP address could be either node's

- *curl https://meow.com:30908 — resolve meow.com:30908:52.183.91.218 -k*

```
jonw@CKS-Master:~$ curl https://meow.com:30908 --resolve meow.com:30908:52.183.91.218 -k
<html>
<head><title>403 Forbidden</title></head>
<body>
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

If we try to curl the HTTPS website with client credentials, we would get similar results as below. We might still see the HTTP status 403 but that is just because the client is forbidden to visit the site.

- *curl https://meow.com:30908 — resolve meow.com:30908:52.183.91.218 — cert client.crt — key client.key -kv*

```
jonw@CKS-Master:~$ curl https://meow.com:30908 --resolve meow.com:30908:52.183.91.218 --cacert ca.crt --cert client.crt --key client.key -kv
* Added meow.com:30908:52.183.91.218 to DNS cache
* Rebuilt URL to: https://meow.com:30908/
* Hostname meow.com was found in DNS cache
*   Trying 52.183.91.218...
* TCP_NODELAY set
* Connected to meow.com (52.183.91.218) port 30908 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: ca.crt
  CApath: /etc/ssl/certs
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS Unknown, Certificate Status (22):
* TLSv1.3 (IN), TLS handshake, Unknown (8):
* TLSv1.3 (IN), TLS Unknown, Certificate Status (22):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS Unknown, Certificate Status (22):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS Unknown, Certificate Status (22):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Client hello (1):
* TLSv1.3 (OUT), TLS Unknown, Certificate Status (22):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* ALPN, server accepted to use h2
* Server certificate:
*  subject: CN=meow.com
```

Kubernetes Security Preparation Cloud Native Microservices