Exercise 6.1: Set SecurityContext for a Pod and Container

Working with Security: Overview

In this lab we will implement security features for new applications, as the simpleapp YAML file is getting long and more difficult to read. Kubernetes architecture favors smaller, decoupled, and transient applications working together. We'll continue to emulate that in our exercises.

In this exercise we will create two new applications. One will be limited in its access to the host node, but have access to encoded data. The second will use a `network security policy` to move from the default all-access Kubernetes policies to a mostly closed network. First we will set `security contexts` for pods and containers, then create and consume secrets, then finish with configuring a network security policy.

1. Begin by making a new directory for our second application. Change into that directory.

```
student@ckad-1:~$ mkdir ~/app2

student@ckad-1:~$ cd ~/app2/
```

2. Create a YAML file for the second application. In the example below we are using a simple image, busybox, which allows access to a shell, but not much more. We will add a `runAsUser` to both the pod as well as the container.

```
student@ckad-1:~/app2$ vim second.yaml
```

second.yaml

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: secondapp
5  spec:
6    securityContext:
7      runAsUser: 1000
8    containers:
9    - name: busy
10     image: busybox
11     command:
12       - sleep
13       - "3600"
14     securityContext:
15       runAsUser: 2000
16       allowPrivilegeEscalation: false
```

3. Create the secondapp pod and verify it's running. Unlike the previous deployment this application is running as a pod. Look at the YAML output, to compare and contrast with what a deployment looks like. The status section probably has the largest contrast.

```
student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created

student@ckad-1:~/app2$ kubectl get  pod secondapp
```

```
NAME        READY     STATUS    RESTARTS   AGE
secondapp   1/1       Running   0          21s

student@ckad-1:~/app2$ kubectl get pod secondapp -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    cni.projectcalico.org/podIP: 192.168.158.97/32
  creationTimestamp: "2019-11-03T21:23:12Z"
  name: secondapp
<output_omitted>
```

4. Execute a Bourne shell within the Pod. Check the user ID of the shell and other processes. It should show the container setting, not the pod. This allows for multiple containers within a pod to customize their UID if desired. As there is only one container in the pod we do not need to use the **-c busy** option.

```
student@ckad-1:~/app2$ kubectl exec -it secondapp -- sh
```

On Container

```
/ $ ps aux
PID   USER     TIME  COMMAND
    1   2000      0:00   sleep 3600
    8   2000      0:00   sh
   12   2000     0:00   ps aux
```

5. While here check the capabilities of the kernel. In upcoming steps we will modify these values.

On Container

```
/ $ grep Cap /proc/1/status
CapInh:         00000000a80425fb
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         00000000a80425fb
CapAmb:         0000000000000000

/ $ exit
```

6. Use the capability shell wrapper tool, the **capsh** command, to decode the output. We will view and compare the output in a few steps. Note that there are 14 comma separated capabilities listed.

```
student@ckad-1:~/app2$ capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,
cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,
cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

7. Edit the YAML file to include new capabilities for the container. A capability allows granting of specific, elevated privileges without granting full root access. We will be setting **NET_ADMIN** to allow interface, routing, and other network configuration. We'll also set **SYS_TIME**, which allows system clock configuration. More on kernel capabilities can be read here: https://github.com/torvalds/linux/blob/master/include/uapi/linux/capability.h

   It can take up to a minute for the pod to fully terminate, allowing the future pod to be created.

```
student@ckad-1:~/app2$ kubectl delete pod secondapp
```

```
pod "secondapp" deleted

student@ckad-1:~/app2$ vim second.yaml
```

second.yaml

```
1  <output_omitted>
2        - sleep
3        - "3600"
4      securityContext:
5        runAsUser: 2000
6        allowPrivilegeEscalation: false
7        capabilities:                        #<-- Add this and following line
8          add: ["NET_ADMIN", "SYS_TIME"]
```

8. Create the pod again. Execute a shell within the container and review the Cap settings under `/proc/1/status`. They should be different from the previous instance.

```
student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created

student@ckad-1:~/app2$ kubectl exec -it secondapp -- sh
```

On Container

```
/ $ grep Cap /proc/1/status
CapInh:         00000000aa0435fb
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         00000000aa0435fb
CapAmb:         0000000000000000

/ $ exit
```

9. Decode the output again. Note that the instance now has 16 comma delimited capabilities listed. **cap_net_admin** is listed as well as **cap_sys_time**.

```
student@ckad-1:~/app2$ capsh --decode=00000000aa0435fb
0x00000000aa0435fb=cap_chown,cap_dac_override,cap_fowner,
cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,
cap_net_bind_service,cap_net_admin,cap_net_raw,cap_sys_chroot,
cap_sys_time,cap_mknod,cap_audit_write,cap_setfcap
```