Lab 6.1 - 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 will 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, we will create and consume secrets, and we will finish with configuring a network security policy.

Set SecurityContext for a Pod and Container

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

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

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

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
apiVersion: v1
kind: Pod
```



```
metadata:
  name: secondapp
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - image: busybox
    name: secondapp
    command:
      - sleep
      - "3600"
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
    name:busy
```

Create the **secondapp** pod and verify it is 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:
  creationTimestamp: 2018-04-18T18:58:53Z
  name: secondapp
<output_omitted>
```

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
/ $ ps aux
PID   USER     TIME   COMMAND
```



```
    1 2000      0:00 sleep 3600
    8 2000      0:00 sh
   12 2000      0:00 ps aux
```

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

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

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_kil
l,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sy
s_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

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
<output_omitted>
      - sleep
      - "3600"
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
      Capabilities:                    #Add this and the following line
```



```
        add: ["NET_ADMIN", "SYS_TIME"]
    name: busy
```

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

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_kil
l,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
```

Create and Consume Secrets

Secrets are consumed in a manner similar to ConfigMaps, covered in an earlier lab. While at-rest encryption is on the way, at the moment, a secret is just base64-encoded. Begin by generating an encoded password:

```
student@ckad-1:~/app2$ echo LFTr@1n | base64
TEZUckAxbgo=
```

Create a YAML file for the object, with an API object **kind** set to **Secret**. Use the encoded key as a password parameter:

```
student@ckad-1:~/app2$ vim secret.yaml
apiVersion: v1
```



```
kind: Secret
metadata:
  name: lfsecret
data:
  password: TEZUckAxbgo=
```

Ingest the new object into the cluster:

```
student@ckad-1:~/app2$ kubectl create -f secret.yaml
secret/lfsecret created
```

Edit the **secondapp** YAML file to use the secret as a volume mounted under **/mysqlpassword**. Note that, as there is a command executed, the pod will restart when the command finishes every 3600 seconds, or every hour.

```
student@ckad-1:~/app2$ vim second.yaml
<output_omitted>
     allowPrivilegeEscalation: false
       capabilities:
         add: ["NET_ADMIN", "SYS_TIME"]
     volumeMounts:
     - mountPath: /mysqlpassword
       name: mysql
     name: busy
  volumes:
  - name: mysql
    secret:
      secretName: lfsecret

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

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

Verify the pod is running, then check if the password is mounted where expected. We will find that the password is available in its clear-text, decoded state.

```
student@ckad-1:~/app2$ kubectl get pod secondapp
NAME        READY     STATUS    RESTARTS   AGE
secondapp   1/1       Running   0          34s

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



```
/ $ cat /mysqlpassword/password
LFTr@1n
```

View the location of the directory. Note that it is a symbolic link to **../data**, which is also a symbolic link to another directory. After taking a look at the filesystem within the container, exit back to the node:

```
/ $ cd /mysqlpassword/
/mysqlpassword $ ls
password
/mysqlpassword $ ls -al
total 4
drwxrwxrwt    3 root     root           100 Apr 11 07:24 .
drwxr-xr-x   21 root     root          4096 Apr 11 22:30 ..
drwxr-xr-x    2 root     root            60 Apr 11 07:24
..4984_11_04_07_24_47.831222818
lrwxrwxrwx    1 root     root            31 Apr 11 07:24 ..data ->
..4984_11_04_07_24_47.831222818
lrwxrwxrwx    1 root     root            15 Apr 11 07:24 password ->
..data/password

/mysqlpassword $ exit
```

Implement a NetworkPolicy

An early architecture decision with Kubernetes was non-isolation - that all pods were able to connect to all other pods and nodes by design. In more recent releases, the use of a **NetworkPolicy** allows for pod isolation. The policy only has effect when the network plugins, like *Project Calico,* are capable of honoring them. If used with a plugin like *flannel*, they will have no effect. The use of **matchLabels** allows for a more granular selection within the namespace, which can be selected using a **namespaceSelector**. Using multiple labels can allow for complex application of rules. More information can be found here:
https://kubernetes.io/docs/concepts/services-networking/network-policies/

Begin by creating a default policy which denies all traffic. Once ingested into the cluster, this will affect every pod not selected by another policy, creating a mostly-closed environment. If you want to only deny ingress or egress traffic, you can remove the other **policyType**.

```
student@ckad-1:~/app2$ vim allclosed.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```



```
metadata:
  name: deny-default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Before we can test the new network policy, we need to make sure network access works without it applied. Update **secondapp** to include a new container running **nginx**, then test access. Begin by adding two lines for the image and name **webserver**, as found below. It takes a bit for the pod to terminate, so we'll delete, then edit the file.

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

student@ckad-1:~/app2$ vim second.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - image: nginx
    name: webserver
    ports:
    - containerPort:80
  - image: busybox
    name: secondapp
    command:
<output_omitted>
```

Create the new pod. Be aware the pod will move from **ContainerCreating** to **Error** to **CrashLoopBackOff**, as only one of the containers will start. We will troubleshoot the error in the following steps.

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

student@ckad-1:~/app2$ kubectl get pods
NAME                          READY     STATUS             RESTARTS   AGE
```



```
date-1523648520-z282m       0/1       Completed          0          3m
date-1523648580-zznjs       0/1       Completed          0          2m
date-1523648640-2h6qt       0/1       Completed          0          1m
date-1523648700-drfp5       1/1       Running            0          18s
nginx-6b58d9cdfd-9fnl4      1/1       Running            1          8d
registry-795c6c8b8f-hl5wf   1/1       Running            2          8d
secondapp                   1/2       CrashLoopBackOff   1          13s
```

Take a closer look at the events leading up to the failure. The images were pulled and the container was started. It was the execution of the container which failed.

```
student@ckad-1:~/app2$ kubectl get event
<output_omitted>
         Normal    Created                 kubelet, ckad-2-wdrq    Created
container
5m          5m           1         secondapp.1525166daeeb0e43
Pod          spec.containers{busy}            Normal    Started
kubelet, ckad-2-wdrq    Started container
20s         5m           25        secondapp.1525166e5791a7fd
Pod          spec.containers{webserver}       Warning   BackOff
kubelet, ckad-2-wdrq    Back-off restarting failed container
```

View the logs of the **webserver** container mentioned in the previous output. Note that there are errors about the user directive and not having permission to make directories.

```
student@ckad-1:~/app2$ kubectl logs secondapp webserver
2018/04/13 19:51:13 [warn] 1#1: the "user" directive makes sense only if
the master process runs with super-user privileges, ignored in
/etc/nginx/nginx.conf:2
nginx: [warn] the "user" directive makes sense only if the master process
runs with super-user privileges, ignored in /etc/nginx/nginx.conf:2
2018/04/13 19:51:13 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp"
failed (13: Permission denied)
nginx: [emerg] mkdir() "/var/cache/nginx/client_temp" failed (13:
Permission denied)
```

Delete the pods. Edit the YAML file to comment out the setting of a UID for the entire pod.

```
student@ckad-1:~/app2$ kubectl delete -f second.yaml
pod "secondapp" deleted

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



```
  name: secondapp
spec:
#  securityContext:
#   runAsUser: 1000
  containers:
  - image: nginx
    name: webserver
    ports:
    - containerPort: 80
<output_omitted>
```

Create the pod again. This time, both containers should run. You may have to wait for the previous pod to fully terminate, depending on how fast you type.

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

student@ckad-1:~/app2$ kubectl get pods
NAME                        READY     STATUS      RESTARTS    AGE
<output_omitted>
secondapp                   2/2       Running     0           5s
```

Expose the **webserver** using a **NodePort** service. Expect an error due to the lack of labels:

```
student@ckad-1:~/app2$ kubectl expose pod secondapp --type=NodePort
--port=80
error: couldn't retrieve selectors via --selector flag or introspection:
the pod has no labels and cannot be exposed
See 'kubectl expose -h' for help and examples.
```

Edit the YAML file to add a label in the metadata, adding the **example: second** label right after the pod name. Note that you can delete several resources at once by passing the YAML file to the **delete** command. Delete and recreate the pod. It may take up to a minute for the pod to shut down:

```
student@ckad-1:~/app2$ kubectl delete -f second.yaml
pod "secondapp" deleted

student@ckad-1:~/app2$ vim second.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
  labels:
```



```
    example: second
spec:
#  securityContext:
#    runAsUser: 1000
<output_omitted>

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

student@ckad-1:~/app2$ kubectl get pods
NAME                        READY     STATUS      RESTARTS    AGE
<output_omitted>
secondapp                   2/2       Running     0           15s
```

This time we will expose the **NodePort** again, and create the service separately, then add a label to illustrate how labels are essential for tying resources together inside of Kubernetes.

```
student@ckad-1:~/app2$ kubectl create service nodeport secondapp --tcp=80
service/secondapp created
```

Look at the details of the service. Note the selector is set to **app: secondapp**. Also, take the node of the **nodePort**, which is **31655** in the example below; yours may be different:

```
student@ckad-1:~/app2$ kubectl get svc secondapp -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2018-04-19T22:07:25Z
  labels:
    app: secondapp
  name: secondapp
  namespace: default
  resourceVersion: "216490"
  selfLink: /api/v1/namespaces/default/services/secondapp
  uid: 0aeaea82-441e-11e8-ac6e-42010a800007
spec:
  clusterIP: 10.97.96.75
  externalTrafficPolicy: Cluster
  ports:
  - name: "80"
    nodePort: 31655
    port: 80
    protocol: TCP
```



```
    targetPort: 80
  selector:
    app: secondapp
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

Test access to the service using **curl** and the ClusterIP shown in the previous output. As the label does not match any other resources, the **curl** command should hang, and eventually time out.

```
student@ckad-1:~/app2$ curl http://10.97.96.75
```

Edit the service. We will change the label to match **secondapp**, and set the **nodePort** to a new port, one that may have been specifically opened by our firewall team, port **32000**.

```
student@ckad-1:~/app2$ kubectl edit svc secondapp
<output_omitted>
  ports:
  - name: "80"
    nodePort: 32000       ## Edit this line
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    example: second       ## Edit this line, too
  sessionAffinity: None
<output_omitted>
```

Verify the updated port number is showing properly, and take note of the **ClusterIP**. The example below shows a **clusterIP** of **10.97.96.75** and a port of **32000** as expected.

```
student@ckad-1:~/app2$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
<output_omitted>
secondapp    NodePort    10.97.96.75      <none>        80:32000/TCP    5m
```

Test access to the high port. You should get the default **nginx** page both if you test from the node to the **ClusterIP:low-port** and from the exterior **hostIP:hightport**. As the high port is randomly generated, make sure it is available. Both of your nodes should be exposing the web server on port **32000**:

```
student@ckad-1:~/app2$ curl http://10.97.96.75
```



```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>

[serewicz@laptop ~]$ curl http://35.184.219.5:32000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

Now, test egress from a container to the outside world. We'll use the **netcat** command to verify access to a running webserver on port 80. First, test local access to nginx, then a remote server.

```
student@ckad-1:~/app2$ kubectl exec -it -c busy secondapp sh
/ $ nc -vz 127.0.0.1 80
127.0.0.1 (127.0.0.1:80) open

/ $ nc -vz www.linux.com 80
www.linux.com (151.101.185.5:80) open

/ $ exit
```

Now that we have tested both ingress and egress, we can implement the network policy:

```
student@ckad-1:~/app2$ kubectl create -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default created
```

Use the ingress and egress tests again. Three of the four should eventually time out. Start by testing from outside the cluster:

```
[serewicz@laptop ~]$ curl http://35.184.219.5:32215
curl: (7) Failed to connect to 35.184.219.5 port 32000: Connection timed
out
```

Then, test from the host to the container:

```
student@ckad-1:~/app2$ curl http://10.97.96.75:80
curl: (7) Failed to connect to 10.97.96.75 port 80: Connection timed out
```


Now, test egress. From container to container should work, as the filter is outside of the pod; then, test egress to an external web page. It should eventually time out:

```
student@ckad-1:~/app2$ kubectl exec -it -c busy secondapp sh
/ $ nc -vz 127.0.0.1 80
127.0.0.1 (127.0.0.1:80) open

/ $ nc -vz www.linux.com 80
nc: bad address 'www.linux.com'

/ $ exit
```

Update the **NetworkPolicy** and remove the **Egress** line. Then, replace the policy:

```
student@ckad-1:~/app2$ vim allclosed.yaml
….
# - Egress

student@ckad-1:~/app2$ kubectl replace -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default replaced
```

Test egress access to an outside site. Get the IP address of the **eth0** inside the container while logged in. The IP is **192.168.55.91** in the example below, but yours may be different:

```
student@ckad-1:~/app2$ kubectl exec -it -c busy secondapp sh
/ $ nc -vz www.linux.com 80
www.linux.com (151.101.185.5:80) open
/ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
4: eth0@if59: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 1e:c8:7d:6a:96:c3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.55.91/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::1cc8:7dff:fe6a:96c3/64 scope link
       valid_lft forever preferred_lft forever
```



```
/ $ exit
```

Now, add a selector to allow ingress to only the **nginx** container. Use the IP from the **eth0** range:

```
student@ckad-1:~/app2$ vim allclosed.yaml
<output_omitted>
policyTypes:
 - Ingress
 ingress:
 - from:
   - ipBlock:
       cidr: 192.168.0.0/16
```

Recreate the policy, and verify its configuration.

```
student@ckad-1:~/app2$ kubectl replace -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default replaced

student@ckad-1:~/app2$ kubectl get networkpolicy
NAME           POD-SELECTOR     AGE
deny-default   example=second   15s

student@ckad-1:~/app2$ kubectl get networkpolicy -o yaml
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: NetworkPolicy
  metadata:
<output_omitted>
```

Test access to the container both using **curl**, as well as **ping**, the IP address to use was found from **ip a** inside the container:

```
student@ckad-1:~/app2$ curl http://192.168.55.91
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>

student@ckad-1:~/app2$ ping -c5 192.168.55.91
PING 192.168.55.91 (192.168.55.91) 56(84) bytes of data.
64 bytes from 192.168.55.91: icmp_seq=1 ttl=63 time=1.11 ms
```



```
64 bytes from 192.168.55.91: icmp_seq=2 ttl=63 time=0.352 ms
64 bytes from 192.168.55.91: icmp_seq=3 ttl=63 time=0.350 ms
64 bytes from 192.168.55.91: icmp_seq=4 ttl=63 time=0.359 ms
64 bytes from 192.168.55.91: icmp_seq=5 ttl=63 time=0.295 ms

--- 192.168.55.91 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4054ms
rtt min/avg/max/mdev = 0.295/0.495/1.119/0.312 ms
```

Update the policy to only allow ingress for TCP traffic on port 80, then test with **curl**, which should work. The **ports** entry should line up with the **from** entry a few lines above:

```
student@ckad-1:~/app2$ vim allclosed.yaml
<output_omitted>
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 192.168.0.0/16
    ports:
    - port: 80
      protocol: TCP

student@ckad-1:~/app2$ kubectl replace -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default replaced

student@ckad-1:~/app2$ curl http://192.168.55.91
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

All five pings should fail, with zero received:

```
student@ckad-1:~/app2$ ping -c5 192.168.55.91
PING 192.168.55.91 (192.168.55.91) 56(84) bytes of data.

--- 192.168.55.91 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4098ms
```

