Run Multiple Container Runtimes

Learn how to run multiple container runtimes in Kubernetes.

We'll cover the following

Overview

With the help of the CRI, we can let `kubelet` talk with a variety of container runtimes. In this lesson, we will show how to configure containerd to run both the runc and gVisor containers. **runc** is a low-level container engine that is used by containerd to manage containers. On the other hand, **gVisor** is an application kernel that provides a secure environment for containers.

Why different container runtimes

Normally, a Kubernetes cluster is shared by many users and workloads—this could bring in some security concerns. Not all the applications running in the cluster are trusted. Some of them may use malicious container images, which may incur unexpected risks. Moreover, the commonly used container runtime Docker/containerd has an inescapable security issue; containers sharing the same kernel with the host, which makes it exploitable with the same vulnerabilities when the host system kernel gets affected. If some containers are running in privilege mode, the host system kernel can be exploited through those containers as well.

The common containerization approaches (such as containerd) rely on Linux `namespaces` and `cgroups`, which are fairly suitable to run trusted application containers. However, hypervisor-based containerization implementations can provide stronger isolation, which can help mitigate the vulnerabilities with hosting untrusted containers.

In Kubernetes, the `RuntimeClass` API is provided to allow the running of workloads with a selected container runtime.

Configure containerd with gVisor

Below is the Kubernetes environment we are using:

gvisor-runtimeclass.yml ×

Search in directory...

/
env-init.sh
nginx-gvisor-pod.yml
gvisor-runtimeclass.yml

```yaml
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
```

Our Kubernetes environment

First, let’s run the commands below to install the container runtime containerd and gVisor.

```
./env-init.sh
```

Commands to initialize our environment

Next, we need to configure `kubelet` to switch the runtime to `containerd`. Run the following command in the terminal above:

```
echo 'KUBELET_KUBEADM_ARGS="--network-plugin=cni --pod-infra-container-image=k8s.gcr.io/pause:3.6 --v=4
```

Use containerd as the container runtime

Now comes the most important part. Let’s configure `containerd` with gVisor. With the following commands running in the terminal above, gVisor is successfully enabled:

```
cat <<EOF | patch -d /etc/containerd
diff a/config.toml b/config.toml
--- a/config.toml
+++ b/config.toml
```

```
@@ -99,6 +99,12 @@

       [plugins."io.containerd.grpc.v1.cri".containerd.runtimes]

+        [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc]
+          runtime_type = "io.containerd.runsc.v1"
+
+          [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc.options]
+            SystemdCgroup = true
+
         [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
           base_runtime_spec = ""
           cni_conf_dir = ""
@@ -122,7 +128,7 @@
             NoPivotRoot = false
             Root = ""
             ShimCgroup = ""
-            SystemdCgroup = false
+            SystemdCgroup = true

       [plugins."io.containerd.grpc.v1.cri".containerd.untrusted_workload_runtime]
         base_runtime_spec = ""
EOF
```

Apply patch for gVisor in containerd

Finally, let's restart `containerd` and `kubelet` by running the following commands in the terminal above:

```
systemctl restart containerd
systemctl restart kubelet
```

Restart the services

After a while (about 20s to 1min), we can see that the `kubelet` node becomes ready. By issuing the command `kubectl get node`, we can list all the nodes and view the statuses. The output will be as follows:

```
NAME               STATUS                     ROLES                  AGE     VERSION
73b67756424ddc50   Ready,SchedulingDisabled   control-plane,master   2m21s   v1.23.8
```

An example output of listing nodes

Let's enable the node for scheduling by running following the command in the terminal above:

```
NODE=`kubectl get nodes -o custom-columns=NAME:.metadata.name --no-headers`
kubectl uncordon $NODE
```

Uncordon the node for scheduling

Test it out

Now, we can test it out. Firstly, let's create a `RuntimeClass` object called `gvisor`. Run the following command in the terminal above:

```
kubectl apply -f /usercode/gvisor-runtimeclass.yml
```

Create the RuntimeClass object gvisor

We can create a Pod with this object `gvisor`. Run the following command in the terminal above:

```
kubectl apply -f /usercode/nginx-gvisor-pod.yml
```

Create Pod using gVisor as container runtime

We can list the Pods to view the statuses by running the command below:

```
kubectl get pod -o wide
```

Command to list Pods

The output will be as follows:

```
NAME           READY   STATUS    RESTARTS   AGE     IP            NODE               NOMINATED NODE   REA
nginx-gvisor   1/1     Running   0          3m40s   10.244.0.6    6550573b8bf5a716   <none>           <nc
```

The output of Pods listing

Let's view the `RuntimeClass` with the JSONPath support of `kubectl`. Run the following command in the terminal above:

```
kubectl get pod -o custom-columns=NAME:metadata.name,STATUS:.status.phase,RUNTIME_CLASS:.spec.runtimeCla
```

View the RuntimeClass of Pods

The output will be as follows:

```
NAME           STATUS    RUNTIME_CLASS
nginx-gvisor   Running   gvisor
```

The RuntimeClass of Pods

Ta-da! Running multiple container runtimes in Kubernetes works!

Back

How the CRI Works

Next

Quiz on the Container Runtime Interface