讲师：李振良（阿良）

今天课题：《最小化微服务漏洞》

学院官网：www.aliangedu.cn

阿良个人微信

DevOps技术栈公众号

第四章 最小化微服务漏洞

- Pod安全上下文
- Pod安全策略
- OPA Gatekeeper
- Secret存储敏感数据
- 安全沙箱运行容器

Pod安全上下文

Pod安全上下文

安全上下文（Security Context）： K8s对Pod和容器提供的安全机制，可以设置Pod特权和访问控制。

Pod安全上下文

安全上下文限制维度：

- 自主访问控制（Discretionary Access Control）：基于用户ID（UID）和组ID（GID），来判定对对象（例如文件）的访问权限。
- 安全性增强的 Linux（SELinux）： 为对象赋予安全性标签。
- 以特权模式或者非特权模式运行。
- Linux Capabilities: 为进程赋予 root 用户的部分特权而非全部特权。
- AppArmor：定义Pod使用AppArmor限制容器对资源访问限制
- Seccomp：定义Pod使用Seccomp限制容器进程的系统调用
- AllowPrivilegeEscalation： 禁止容器中进程（通过 SetUID 或 SetGID 文件模式）获得特权提升。当容器以特权模式运行或者具有CAP_SYS_ADMIN能力时，AllowPrivilegeEscalation总为True。
- readOnlyRootFilesystem：以只读方式加载容器的根文件系统。

Pod安全上下文

案例1：设置容器以普通用户运行

背景：容器中的应用程序默认以root账号运行的，这个root与宿主机root账号是相同的，拥有大部分对Linux内核的系统调用权限，这样是不安全的，所以我们应该将容器以普通用户运行，减少应用程序对权限的使用。

可以通过两种方法设置普通用户：

- Dockerfile里使用USER指定运行用户
- K8s里指定spec.securityContext.runAsUser，指定容器默认用户UID

```
spec:
    securityContext:
      runAsUser: 1000   # 镜像里必须有这个用户UID
      fsGroup: 1000 # 数据卷挂载后的目录属组设置为该组
    containers:
    - image: lizhenliang/flask-demo:root
      name: web
      securityContext:
        allowPrivilegeEscalation: false  # 不允许提权
```

Pod安全上下文

案例2：避免使用特权容器

背景：容器中有些应用程序可能需要访问宿主机设备、修改内核等需求，在默认情况下，容器没这个有这个能力，因此这时会考虑给容器设置特权模式。

启用特权模式：

```
containers:
    - image: lizhenliang/flask-demo:root
      name: web
      securityContext:
        privileged: true
```

启用特权模式就意味着，你要为容器提供了访问Linux内核的所有能力，这是很危险的，为了减少系统调用的供给，可以使用Capabilities为容器赋予仅所需的能力。

Pod安全上下文

Linux Capabilities：Capabilities 是一个内核级别的权限，它允许对内核调用权限进行更细粒度的控制，而不是简单地以 root 身份能力授权。

Capabilities 包括更改文件权限、控制网络子系统和执行系统管理等功能。在 securityContext 中，可以添加或删除 Capabilities，做到容器精细化权限控制。

capability 名称	描述
CAP_AUDIT_CONTROL	启用和禁用内核审计；改变审计过滤规则；检索审计状态和过滤规则
CAP_AUDIT_READ	允许通过 multicast netlink 套接字读取审计日志
CAP_AUDIT_WRITE	将记录写入内核审计日志
CAP_BLOCK_SUSPEND	使用可以阻止系统挂起的特性
CAP_CHOWN	修改文件所有者的权限
CAP_DAC_OVERRIDE	忽略文件的 DAC 访问限制
CAP_DAC_READ_SEARCH	忽略文件读及目录搜索的 DAC 访问限制
CAP_FOWNER	忽略文件属主 ID 必须和进程用户 ID 相匹配的限制
CAP_FSETID	允许设置文件的 setuid 位
CAP_IPC_LOCK	允许锁定共享内存片段
CAP_IPC_OWNER	忽略 IPC 所有权检查
CAP_KILL	允许对不属于自己的进程发送信号
CAP_LEASE	允许修改文件锁的 FL_LEASE 标志
CAP_LINUX_IMMUTABLE	允许修改文件的 IMMUTABLE 和 APPEND 属性标志
CAP_MAC_ADMIN	允许 MAC 配置或状态更改
CAP_MAC_OVERRIDE	覆盖 MAC(Mandatory Access Control)
CAP_MKNOD	允许使用 mknod() 系统调用
CAP_NET_ADMIN	允许执行网络管理任务：接口、防火墙和路由等
CAP_NET_BIND_SERVICE	允许绑定到小于 1024 的端口
CAP_NET_BROADCAST	允许网络广播和多播访问
CAP_NET_RAW	允许使用原始套接字
CAP_SETGID	允许改变进程的 GID
CAP_SETFCAP	允许为文件设置任意的 capabilities
CAP_SETPCAP	允许向其它进程转移能力以及删除其它进程的任意能力(只限init进程)
CAP_SETUID	允许改变进程的 UID
CAP_SYS_ADMIN	允许执行系统管理任务，如加载或卸载文件系统、设置磁盘配额等
CAP_SYS_BOOT	允许重新启动系统
CAP_SYS_CHROOT	允许使用 chroot() 系统调用
CAP_SYS_MODULE	允许插入和删除内核模块
CAP_SYS_NICE	允许提升优先级及设置其他进程的优先级

Pod安全上下文

示例1：容器默认没有挂载文件系统能力，添加SYS_ADMIN增加这个能力

```
apiVersion: v1
kind: Pod
metadata:
  name: cap-pod
spec:
  containers:
  - image: busybox
    name: test
    command:
    - sleep
    - 24h
    securityContext:
      capabilities:
        add: ["SYS_ADMIN"]
```

Pod安全上下文

案例2：只读挂载容器文件系统，防止恶意二进制文件创建

```
apiVersion: v1
kind: Pod
metadata:
  name: cap-pod
spec:
  containers:
  - image: busybox
    name: test
    command:
    - sleep
    - 24h
    securityContext:
      readOnlyRootFilesystem: true
```

Pod安全策略（PSP）

Pod安全策略（PSP）

PodSecurityPolicy（简称PSP）：Kubernetes中Pod部署时重要的安全校验手段，能够有效地约束应用运行时行为安全。

使用PSP对象定义一组Pod在运行时必须遵循的条件及相关字段的默认值，只有Pod满足这些条件才会被K8s接受。

Pod安全策略（PSP）

Pod安全策略限制维度：

配置项	描述
privileged	启动特权容器。
hostPID，hostIPC	使用主机namespaces。
hostNetwork，hostPorts	使用主机网络和端口。
volumes	允许使用的挂载卷类型。
allowedHostPaths	允许hostPath类型挂载卷在主机上挂载的路径，通过pathPrefix字段声明允许挂载的主机路径前缀组。
allowedFlexVolumes	允许使用的指定FlexVolume驱动。
fsGroup	配置Pod中挂载卷使用的辅组ID。
readOnlyRootFilesystem	约束启动Pod使用只读的root文件系统。
runAsUser，runAsGroup，supplementalGroups	指定Pod中容器启动的用户ID以及主组和辅组ID。
allowPrivilegeEscalation，defaultAllowPrivilegeEscalation	约束Pod中是否允许配置allowPrivilegeEscalation=true，该配置会控制setuid的使用，同时控制程序是否可以使用额外的特权系统调用。
defaultAddCapabilities，requiredDropCapabilities，allowedCapabilities	控制Pod中使用的Linux Capabilities。
seLinux	控制Pod使用seLinux配置。
allowedProcMountTypes	控制Pod允许使用的ProcMountTypes。
annotations	配置Pod中容器使用的AppArmor或seccomp。
forbiddenSysctls，allowedUnsafeSysctls	控制Pod中容器使用的sysctl配置。

Pod安全策略（PSP）

Pod安全策略实现为一个准入控制器，默认没有启用，当启用后会强制实施Pod安全策略，没有满足的Pod将无法创建。因此，建议在启用PSP之前先添加策略并对其授权。

启用Pod安全策略：

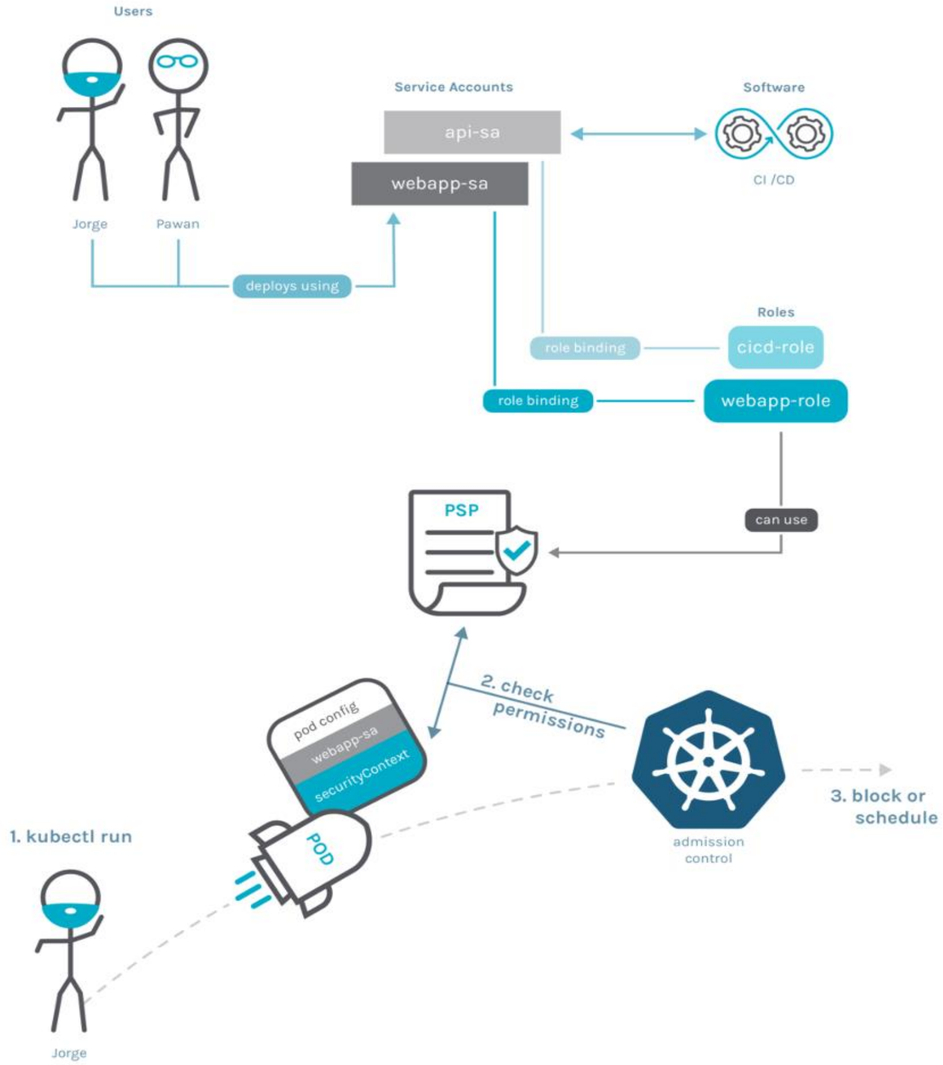
```
vi /etc/kubernetes/manifests/kube-apiserver.yaml
...
- --enable-admission-plugins=NodeRestriction,PodSecurityPolicy
...
systemctl restart kubelet
```

Pod安全策略（PSP）

右图所示，用户使用SA （ServiceAccount）创建了一个Pod，K8s会先验证这个SA是否可以访问PSP资源权限，如果可以访问，会进一步验证Pod配置是否满足PSP规则，如果不满足会拒绝部署。

因此，需要实施需要有这几点：

- 创建SA服务账号
- 创建Role并绑定SA
- 还需要创建一个Role使用PSP资源权限，再绑定SA



PSP工作流程

Pod安全策略（PSP）

示例1：禁止创建特权模式的Pod

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp-example
spec:
  privileged: false  # 不允许特权Pod
  # 下面是一些必要的字段
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

Pod安全策略（PSP）

示例1：禁止创建特权模式的Pod

```
# 创建SA
kubectl create serviceaccount aliang
# 将SA绑定到系统内置Role
kubectl create rolebinding aliang --clusterrole=edit --serviceaccount=default:aliang
# 创建使用PSP权限的Role
kubectl create role psp:unprivileged --verb=use --resource=podsecuritypolicy --resource-name=psp-example
# 将SA绑定到Role
kubectl create rolebinding aliang:psp:unprivileged --role=psp:unprivileged --serviceaccount=default:aliang
```

Pod安全策略（PSP）

示例2：禁止没指定普通用户运行的容器（runAsUser）

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp-example
spec:
  privileged: false  # 不允许特权Pod
  # 下面是一些必要的字段
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: MustRunAsNonRoot
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

OPA Gatekeeper 策略引擎

OPA 策略引擎介绍

PSP不足与状况：

- 将再1.21版本弃用PSP，在1.25版本删除PSP
- 仅支持Pod策略
- 使用复杂，权限模型存在缺陷，控制不明确

弃用文章：https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/

替代提案：https://github.com/kubernetes/enhancements/issues/2579

OPA 策略引擎介绍

OPA（Open Policy Agent）：是一个开源的、通用策略引擎，可以将策略编写为代码。提供一个种高级声明性语言-Rego来编写策略，并把决策这一步骤从复杂的业务逻辑中解耦出来。

OPA可以用来做什么?

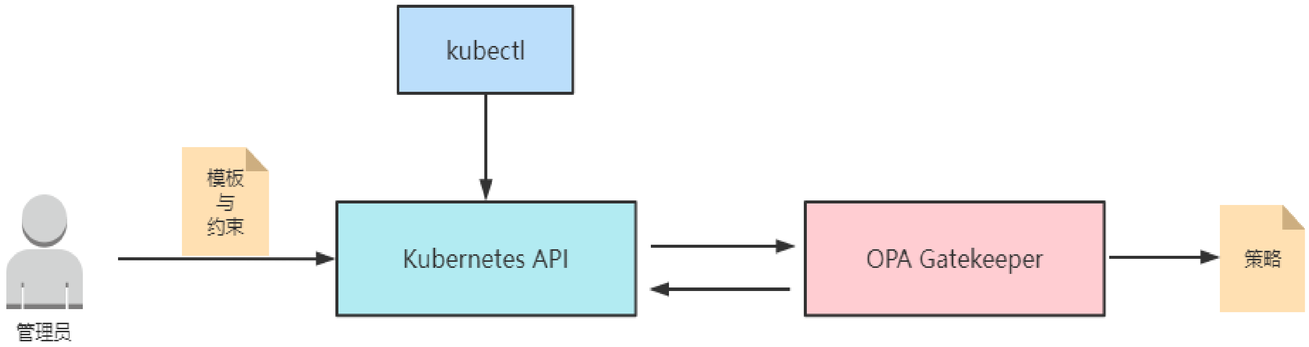
- 拒绝不符合条件的YAML部署
- 允许使用哪些仓库中的镜像
- 允许在哪个时间段访问系统
- 等...

OPA Gatekeeper 策略引擎

Gatekeeper 是基于 OPA的一个 Kubernetes 策略解决方案，可替代PSP或者部分RBAC功能。

- OPA官网：https://www.openpolicyagent.org/
- Gatekeeper项目：https://github.com/open-policy-agent/gatekeeper
- Gatekeeper文档：https://open-policy-agent.github.io/gatekeeper/website/docs/howto/

当在集群中部署了Gatekeeper组件，APIServer所有的创建、更新或者删除操作都会触发Gatekeeper来处理，如果不满足策略则拒绝。



工作流程图

OPA Gatekeeper 策略引擎

部署Gatekeeper：

kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.7/deploy/gatekeeper.yaml

Gatekeeper的策略由两个资源对象组成：

- Template：策略逻辑实现的地方，使用rego语言
- Contsraint：负责Kubernetes资源对象的过滤或者为Template提供输入参数

OPA Gatekeeper 策略引擎

案例1：禁止容器启用特权

模板：

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: privileged
spec:
  crd:
    spec:
      names:
        kind: privileged
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package admission
        violation[{"msg": msg}] {  # 如果violation为true（表达式通过）说明违反约束
          containers = input.review.object.spec.template.spec.containers
          c_name := containers[0].name
          containers[0].securityContext.privileged # 如果返回true，说明违反约束
          msg := sprintf("提示：'%v'容器禁止启用特权！",[c_name])
        }

# 查看资源
kubectl get ConstraintTemplate
```

约束：

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: privileged
metadata:
  name: privileged
spec:
  match:  # 匹配的资源
    kinds:
      - apiGroups: ["apps"]
        kinds:
        - "Deployment"
        - "DaemonSet"
        - "StatefulSet  “


# 查看资源
kubectl get constraints
```

OPA Gatekeeper 策略引擎

案例2：只允许使用特定的镜像仓库

模板：

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: image-check
spec:
  crd:
    spec:
      names:
        kind: image-check
      validation:
        openAPIV3Schema:
          properties:  # 需要满足条件的参数
            prefix:
              type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package image
        violation[{"msg": msg}] {
          containers = input.review.object.spec.template.spec.containers
          image := containers[0].image
          not startswith(image, input.parameters.prefix) # 镜像地址开头不匹配并取反则为true，
说明违反约束
          msg := sprintf("提示：'%v'镜像地址不在可信任仓库！", [image])
        }
```

约束：

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: image-check
metadata:
  name: image-check
spec:
  match:
    kinds:
      - apiGroups: ["apps"]
        kinds:
        - "Deployment"
        - "DaemonSet"
        - "StatefulSet"
  parameters:  # 传递给opa的参数
    prefix: "lizhenliang/"
```

Secret存储敏感数据

Secret存储敏感数据

Secret是一个用于存储敏感数据的资源，所有的数据要经过base64编码，数据实际会存储在K8s中Etcd，然后通过创建Pod时引用该数据。

应用场景：凭据

Pod使用secret数据有两种方式：

- 变量注入
- 数据卷挂载

kubectl create secret 支持三种数据类型：

- docker-registry：存储镜像仓库认证信息
- generic：从文件、目录或者字符串创建，例如存储用户名密码
- tls：存储证书，例如HTTPS证书

Secret存储敏感数据

示例：将Mysql用户密码保存到Secret中存储

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysql
type: Opaque
data:
  mysql-root-password: "MTIzNDU2"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: db
        image: mysql:5.7.30
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql
              key: mysql-root-password
```

安全沙箱运行容器

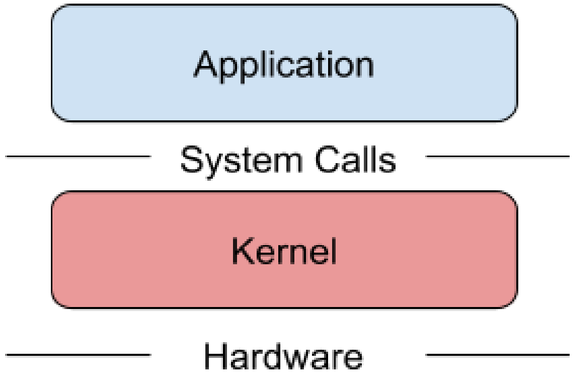
安全沙箱运行容器：gVisor介绍

所知，容器的应用程序可以直接访问Linux内核的系统调用，容器在安全隔离上还是比较弱，虽然内核在不断地增强自身的安全特性，但由于内核自身代码极端复杂，CVE 漏洞层出不穷。
所以要想减少这方面安全风险，就是做好安全隔离，阻断容器内程序对物理机内核的依赖。
Google开源的一种gVisor容器沙箱技术就是采用这种思路，gVisor隔离容器内应用和内核之间访问，提供了大部分Linux内核的系统调用，巧妙的将容器内进程的系统调用转化为对gVisor的访问。

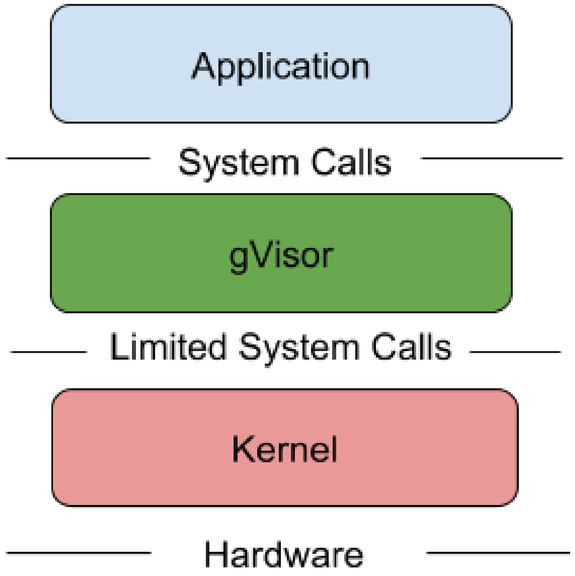
gVisor兼容OCI，与Docker和K8s无缝集成，很方面使用。

项目地址：https://github.com/google/gvisor

安全沙箱运行容器：gVisor介绍



容器中应用程序直接与Linux内核交互

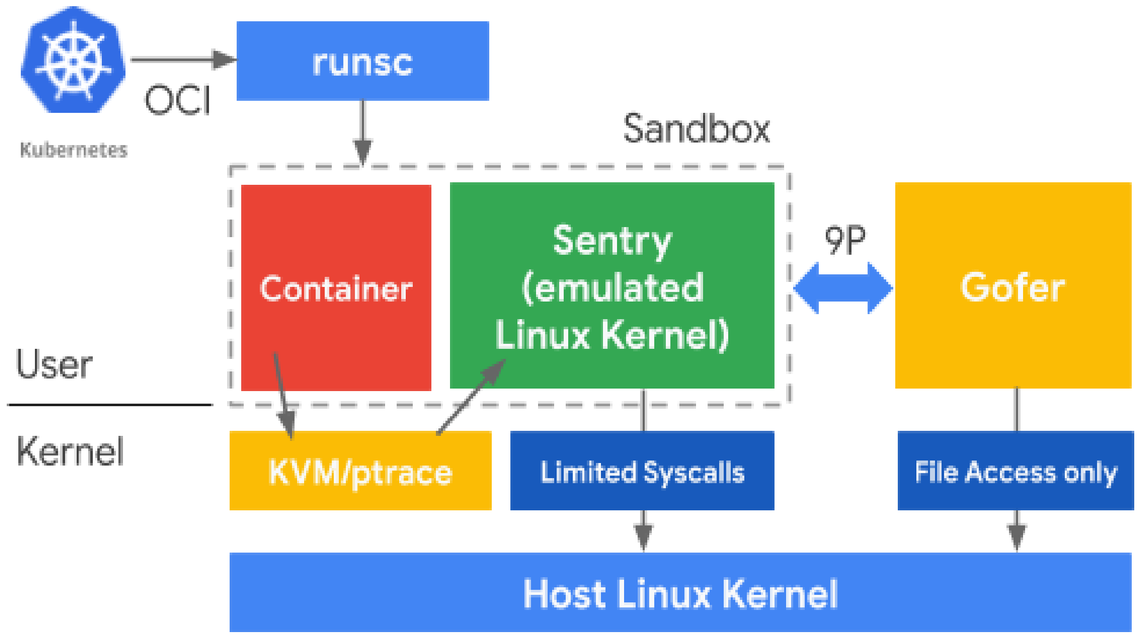


容器中应用程序需要通过gVisor才能使用资源

安全沙箱运行容器：gVisor架构

gVisor 由 3 个组件构成：

- Runsc 是一种 Runtime 引擎，负责容器的创建与销毁。
- Sentry 负责容器内程序的系统调用处理。
- Gofer 负责文件系统的操作代理，IO 请求都会由它转接到 Host 上。



gVisor架构图

安全沙箱运行容器：gVisor与Docker集成

gVisor内核要求：Linux 3.17+

如果用的是CentOS7则需要升级内核，Ubuntu不需要。

CentOS7内核升级步骤：

```
rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org
rpm -Uvh http://www.elrepo.org/elrepo-release-7.0-2.el7.elrepo.noarch.rpm
yum --enablerepo=elrepo-kernel install  kernel-ml-devel kernel-ml –y
grub2-set-default 0
reboot
uname -r
```

安全沙箱运行容器：gVisor与Docker集成

1、准备gVisor二进制文件

```
sha512sum -c runsc.sha512
rm -f *.sha512
chmod a+x runsc
mv runsc /usr/local/bin
```

2、Docker配置使用gVisor

```
runsc install  # 查看加的配置/etc/docker/daemon.json
systemctl restart docker
```

参考文档：https://gvisor.dev/docs/user_guide/install/

安全沙箱运行容器：gVisor与Docker集成

使用runsc运行容器：

```
docker run -d --runtime=runsc nginx
```

使用dmesg验证：

```
docker run --runtime=runsc -it nginx dmesg
```

已经测试过的应用和工具：https://gvisor.dev/docs/user_guide/compatibility/

安全沙箱运行容器：gVisor与Containerd集成

切换Containerd容器引擎

1、准备配置

```
cat > /etc/sysctl.d/99-kubernetes-cri.conf << EOF
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
sysctl -system
```

2、安装

```
cd /etc/yum.repos.d
wget http://mirrors.aliyun.com/docker-
ce/linux/centos/docker-ce.repo
yum install -y containerd.io
```

3、修改配置文件

- pause镜像地址
- Cgroup驱动改为systemd
- 增加runsc容器运行时
- 配置docker镜像加速器

3、修改配置文件

```
mkdir -p /etc/containerd
containerd config default > /etc/containerd/config.toml
vi /etc/containerd/config.toml
  ...
  [plugins."io.containerd.grpc.v1.cri"]
    sandbox_image = "registry.aliyuncs.com/google_containers/pause:3.2"
        ...
      [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
            SystemdCgroup = true
            ...
        [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc]
          runtime_type = "io.containerd.runsc.v1"
      [plugins."io.containerd.grpc.v1.cri".registry.mirrors."docker.io"]
        endpoint = ["https://b9pmyelo.mirror.aliyuncs.com"]
    ...
systemctl restart containerd
```

4、配置kubelet使用containerd

```
vi /etc/sysconfig/kubelet
KUBELET_EXTRA_ARGS=--container-runtime=remote --container-runtime-
endpoint=unix:///run/containerd/containerd.sock --cgroup-driver=systemd
systemctl restart kubelet
```

5、验证

```
kubectl get node -o wide
```

安全沙箱运行容器：gVisor与Containerd集成

containerd也有 ctr 管理工具，但功能比较简单，一般使用crictl工具检查和调试容器。

项目地址：https://github.com/kubernetes-sigs/cri-tools/

准备crictl连接containerd配置文件：

```
cat > /etc/crictl.yaml << EOF
runtime-endpoint: unix:///run/containerd/containerd.sock
EOF
```

下面是docker与crictl命令对照表：

镜像相关功能	Docker	Containerd
显示本地镜像列表	docker images	crictl images
下载镜像	docker pull	crictl pull
上传镜像	docker push	无，例如buildk
删除本地镜像	docker rmi	crictl rmi
查看镜像详情	docker inspect	crictl inspecti

容器相关功能	Docker	Containerd
容器列表	docker ps	crictl ps
创建容器	docker create	crictl create
启动容器	docker start	crictl start
停止容器	docker stop	crictl stop
删除容器	docker rm	crictl rm
容器详情	docker inspect	crictl inspect
附加终端	docker attach	crictl attach
执行命令	docker exec	crictl exec
查看日志	docker logs	crictl logs
资源统计	docker stats	crictl stats

POD 相关功能	Docker	Containerd
显示 POD 列表	无	crictl pods
查看 POD 详情	无	crictl inspectp
运行 POD	无	crictl runp
停止 POD	无	crictl stopp

安全沙箱运行容器：K8s使用gVisor运行容器

RuntimeClass 是一个用于选择容器运行时配置的特性，容器运行时配置用于运行 Pod 中的容器。

创建RuntimeClass：

```
apiVersion: node.k8s.io/v1  # RuntimeClass 定义于 node.k8s.io API 组
kind: RuntimeClass
metadata:
  name: gvisor # 用来引用 RuntimeClass 的名字
handler: runsc # 对应的 CRI 配置的名称
```

安全沙箱运行容器：K8s使用gVisor运行容器

创建Pod测试gVisor：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-gvisor
spec:
  runtimeClassName: gvisor
  containers:
  - name: nginx
    image: nginx

kubectl get pod nginx-gvisor -o wide
kubectl exec nginx-gvisor -- dmesg
```

课后作业

1、创建一个PSP策略，防止创建特权Pod，再创建一个ServiceAccount，使用kubectl –as验证PSP策略效果

2、使用containerd作为容器运行时，准备好gVisor，创建一个RuntimeClass，创建一个Pod在gVisor上运行

谢谢

阿良个人微信

DevOps技术栈公众号

阿良教育：www.aliangedu.cn