链滴

保障集群内节点和网络安全 (SecurityContext PodSecurityPolicy NetworkPolicy)

作者：wangjunjack

原文链接：https://ld246.com/article/1603776306468

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

一、在pod中使用宿主节点的Linux命名空间

每一个pod拥有自己的IP和端口空间，因为他们有自己的网络命名空间

每一个pod拥有自己的进程树，因为它有自己的PID命名空间

每一个pod拥有自己的IPC命名空间，仅允许同一个pod内的进程通过进程间通信机制交流

在pod使用宿主节点的网络命名空间

可以再pod spec 中的hostNetwork设置为tree，从而共享宿主机的网络命名空间，这也意味着该po没有自己的IP地址。

```
...
spec:
    hostNetwork: tree
    containers:
    - name: xxx
      image: xxx
123456
```

绑定宿主节点上的端口而不是使用宿主节点的网络命名空间

将pod端口绑定到宿主机的某一个端口，可以通过配置pod的spec.containers.ports 字段中某个容器一个端口的hostPort属性来实现。

需要注意的是与NodePort service 服务暴露的pod进行区别，使用hostPort时，到达宿主节点的端连接会直接被转发到pod对应的端口上。而在NodePort服务暴露中，到达宿主节点的端口连接会被机转发到被选取的pod。

另一个区别是，使用了hostPort的pod，仅有运行了该类pod的节点会绑定对应的端口，而NodePod类型的服务会在所有的节点上绑定端口，即使这个节点上没有pod运行。

还有一个要注意的是使用了hostPod的pod，在一个节点上只能运行一个该类pod，因为两个pod不同时绑定到宿主机上的同一个端口。

```
...
spec:
  containers:
  - name: xxx
    imange: xxx
    - containerPort: 8080  // 容器的端口
      hostPod: 9000  // 绑定到宿主机上的端口
      protocol: TCP
12345678
```

最初hostPod功能是用于暴露用过DaemonSet部署在每个节点上的系统服务，从而保证一个pod的本不会被调用到同一个节点，但是后面有更好的机制来实现该需求。

使用宿主节点的PID和IPC命名空间

pod spec中的hostPID和hostIPC选项与hostNetwork相似，当它们被设置成tree时，pod中的容器使用宿主节点的PID和IPC命名空间，分别允许它们看到宿主机上的全部进程，或通过IPC与宿主节点的进程进行通信

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: xxx
spec:
  hostPID: true
  hostIPC: true
  containers:
  - name: xxx
   iamge: xxx
12345678910
```

通常容器只能看到内容的进程，但这个pod可以列出宿主机上的所有进程。

二、配置节点的安全上下文

除了让pod使用宿主节点的Linux命名空间，还可以在pod或其所属容器的描述中通过security-contex选项配置其他安全相关的特性，这个选项可以运用于整个pod，或者每个pod中的单独容器。

安全上下文可以配置的内容：

- 指定容器运行进程的用户（用户ID）
- 阻止容器用root用户启动（容器的默认运行用户通常可以在其镜像中指定，所以可能需要阻止容器以oot用户运行）
- 使用特权模式运行容器，使其对宿主节点的内核具有完全的访问权限
- 与上相反，通过添加禁用内核功能，配置细粒度的内核访问权限
- 设置SELinux选项，加强对容器的限制
- 阻止进程写入容器的根文件系统

运行pod而不配置安全上下文

可以通过 kubectl exec podname id 查看没有任何安全上下文配置的pod

可以看到这个容器在用户ID（UID）为0的用户，即root用户，用户组ID（gid）为0（同样是root）用户组下运行。同样它还属于一些其他的用户组

注意：容器运行时使用的用户是在镜像中指定，在Dockerfile中，是通过使用USER命令实现的，如果命令被忽略，容器将使用root用户运行。

使用指定用户来运行容器

为了使用一个与镜像中不同的用户ID来运行pod，需要设置该pod的securityContext.runAsUser 选项

```
apiVersion: v1
kind: Pod
metadata:
  name: xxx
spec:
  containers:
  - name: xxx
    image: alpine
    command: ["/bin/sleep", "9999"]
    securityContext:
      runAsUser: 405    // 需要的是用户ID，而不是用户名（ID 405对应guest用户）
1234567891011
```

kubectl exec podname id 查看结果 uid=405(guest) gid=100(users)

阻止容器以root用户运行

如果攻击者获取了访问镜像仓库的权限，并上传一个标签一样的镜像，并以root用户运行。kubernete的调度器运行该pod实例，kubelet会拉取攻击者的镜像，并运行该镜像中的任何代码。

虽然容器与宿主节点基本上是隔离的，使用root用户运行容器中的进程然然是不好的实践。例如，当主节点上的一个目录被挂载到容器中，如果该容器使用了root用户运行，那它就用于该目录的所有权，如果是非root用户运行，它就只有部分权限。

```
apiVersion: v1
kind: Pod
metadata:
  name: xxx
spec:
  containers:
  - name: xxx
    image: alpine
    command: ["/bin/sleep", "9999"]
    securityContext:
      runAsNonRoot: true
1234567891011
```

使用特权模式运行pod

有时pod需要做它们的宿主节点上能够做的任何事情，例如操作被保护的系统设备，或使用其他通常器中不能使用的内核功能。

这种pod的一个例子就是kube-proxy，该pod需要修改主机的iptables

规则来让kubernetes中服务规则生效。为获取宿主机完整的内核权限，该pod需要在特权模式下运行可以将容器的securityContext中的privileged设置为true实现

```
apiVersion: v1
kind: pod
metadata:
  name: xxx
spec:
  containers:
  - name: xxx
    image: alpine
    command: ["/bin/sleep", "9999"]
    securityContext:
      privileged: true  // 指定给容器在特权模式下运行
1234567891011
```

部署该pod，可以跟没有使用特权模式运行的pod进行对比。

可以通过查看/dev 这个目录进行对比（或其他的方式也可以，只要能体现不一样就行）

进入在非特权模式下运行的pod中，ls /dev 可以看到一个比较短的列表

进入以特权模式运行的pod中，ls /dev 可以看到一个很长的列表，其中就包含了宿主节点上所有的备。这也意味着它可以自由使用任何设备。

举个例子，如果要再一个树莓派上运行一个pod，用这个pod来控制相连的LED, 那么必须使用特权模来运行该pod。

为容器单独添加内核功能

传统的Linux值区分特权和非特权进程，但是经过多件的发展，Linux已经可以通过内核功能支持更细度的权限系统。

相对于让容器在特权模式下给予所有的权限，一个更加安全的做法是给予它需要的内核功能特权，kubernetes允许为特定的容器添加内核功能，或禁用部分内核功能，以允许容器进行更加精细的权限控，限制攻击者潜在的影响。

例如，一个容器是不允许修改系统时间的（硬件时钟的时间）。可以通过在一个非特权模式运行的po中来实验。会发现提示没有权限修改。

如果需要容器修改系统时间，可以再容器的capabilities 里add一项名为CAP_SYS_TIME的功能

```
apiVersion: v1
kind: Pod
metadata:
  name: xxx
spec:
  containers:
  - name: xxx
    image: alpine
    commadn: ["/bin/sleep", "9999"]
    securityContext:
        capabilities:
            add:
            - SYS_TIME
12345678910111213
```

此时就可在pod中修改系统时间了

注意：Linux内核功能的名称通常是以CAP_开头的，但是在pod spec中指定内核功能时，必须省略CA_前缀。

警告：自行尝试修改系统时间，可能导致节点不可用

添加指定的内核功能项，固然比使用privileged: true更好，可以在Linux手册送查看Linux内核功能列表

在容器中禁用内核功能

前面我们已经了解了如何给容器添加内核功能，同样我们可以禁用容器使用某项内核功能。例如，默情况下容器拥有CAP_CHOWN内核功能，可以修改文件系统中文件的所有者。为阻止容器的此种行，可以在容器的securityContext.capabilities.drop

列表中添加此项，以禁用这个修改文件所有者的内核功能。

```
apiVersion: v1
kind: Pod
metadata:
  name: xxx
spec:
  containers:
  - name: xxx
    image: alpine
    command: ["/bin/sleep", "9999"]
    securityContext:
        capabilities:
            drop:
            - CHOWN
```

12345678910111213

阻止对容器根文件系统的写入

因为安全原因，你可以能需要阻止容器中的进程对容器的根文件系统进行写入，仅允许它们写入挂载存储卷。

假如你在运行一个有隐藏的漏洞，可以允许攻击者写入文件系统的应用（使用非编译型语言编写的应），这些应用文件在构建容器时放入容器镜像中，并且在容器根文件系统中提供服务。由于漏洞的存，攻击者可以修改这些文件，在其中注入恶意代码。

这一类攻击可以通过阻止容器写入自己的根文件系统（应用的可执行代码通常存储的位置）来阻止。容器的securityContext.readOnlyRootFilesystem 设置成true来实现

```
apiVersion: v1
kind: Pod
metadata:
  name: xxx
spec:
  containers:
  - name: xxx
    image: alpine
    command: ["/bin/sleep", "9999"]
    securityContext:
      readOnlyRootFilesystem: true  // 设置禁止修改容器的根文件系统
    volumeMounts:
    - name: my-volume
      mountPath: /volume
      readOnly: false  // 运行写入挂载卷
  volumes:
  - name: my-volume
    emptyDir:
```

12345678910111213141516171819

实验

kubectl exec -it podname touch /new-file // 会提示Read-only file system

kubectl exec -it podname touch /volume/newfile

kubectl exec -it podname – ls -la /volumenewfile // 可以看到创建成功了

如上例子，如果容器的根文件系统是只读的，那很可能要为应用程序会写入的每一个目录（如日志、盘缓存）挂载一个数据卷了

提示：为了增强安全性，建议生产环境中容器的readOnlyRootFilesystem设置成true

设置pod级别的安全上下文

上的例子都可以单独对容器的安全上下文进行设置。这些选项中的一部分也可以从pod级别进行设置可设置pod.spec.securityContext

属性进行设置。它们会作为pod中每一个容器的安全上下文，但是会被容器的安全上下文所覆盖。下来介绍一些pod级别独有的安全上下文属性。

容器使用不同用户运行时共享存储卷

在pod中不同的容器可以共享一个挂载存储卷，在一个容器中写，在另外一个容器中读。但这是因为

个容器都是以root用户运行的，拥有存储卷中文件的所有权限。如果使用了runAsUser选项。可能要一个pod中用两个不同的用户来运行两个容器了（可能是一个第三方的容器，需要以它们特定的用户运行进程），如果这样两个容器共享同一个存储卷，它们并不一定能读写另一个容器的文件。

因此，kubernetes允许为pod中所有的容器指定 supplemental组，以允许它们无论以哪个用户ID运都可以共享文件，可以通过以下两个属性进行设置：

- fsGroup
- supplementalGroups

```
apiVersion: v1
kind: Pod
metadata:
  name: xxx
spec:
  securityContext:
    fsGroup: 555
    supplementalGroups: [666, 777]
  containers:
  - name: c1
    image: alpine
    command: ["/bin/sleep", "9999"]
    volumeMounts:
    - name: shared-volume
      mountPath: /volume
      readOnly: false
    securityContext:
      runAsUser: 1111
  - name: c2
    image: alpine
    command: ["/bin/sleep", "9999"]
    securityContext:
      runAsUser: 2222
    volumeMounts:
      -name: shared-volume
       mountPath: /volume
       readOnly: false
  volumes:
  - name: shared-volume
    emptyDir:
12345678910111213141516171819202122324252627282930
```

查看容器信息

```
kubectl  exec -it podname -c c1 sh
$ id
uid=1111 gid=0(root) groups=555,666,777
$ ls -l / grep volume
total 4
-rw-r--r--    1 1111    555  ...
123456
```

可以看到该容器以ID为 1111的用户运行，用户组为0（root），但同时555，666，777的用户组也联到了该用户下。在pod中定义fsGroup为555，所以挂载的存储卷属于用户组ID为555的

此时在/volume

目录下创建一个文件，查看文件属性，该文件所属用户情况跟通常设置下的新建文件不同。在通常情下，某一个用户创建的文件所属的用户组ID，应该与创建用户的所属组ID一致，在这种情况下应该为。在这个容器的根文件系统创建一个文件，可以验证这一点。

由此可见，安全上下文中的fsGroup属性当进程在存储卷中创建文件时起到了作用，而supplemental性定义了某个用户所关联的额外的用户组。

三、限制pod使用安全相关的特性

在上面已经介绍了如何在部署一个pod时在一个宿主节点上做任何想做的事情，很明显要有一种机制限制用户使用其中部分功能，集群管理员可以使用 PodSecurityPolicy资源来限制对以上安全相关的性的使用。

PodSecurityPolicy 资源介绍

PodSecurityPolicy是一种集群资源（无命名空间）的资源，它定义了用户能否在pod中使用各种安相关的特性。维护PodSecurityPolicy资源中的配置策略的工作由集成在API服务器中的PodSecurityPolicy准入插件来完成。

注意：你的集群不一定开启了PodSecurityPoicy 准入控制插件，需要开启它才行

当向API服务器发送pod资源时，PodSecurityPolicy准入控制插件会将这个pod与已经配置的PodSecrityPolicy进行校验，如果这个pod符合集群中已有的安全策略，它会被存入etcd。如果不符合将会被接拒绝。这个插件也会根据安全策略中配置的默认值对pod进行修改。

了解PodSecurityPolicy可以做的事情

一个PodSecurityPolicy资源可以定义以下事项：

- 是否允许pod使用宿主节点的PID、IPC和网络命名空间
- pod允许绑定宿主节点端口
- 容器运行时允许使用用户的ID
- 是否允许拥有特权模式容器的pod
- 允许添加哪些内核功能、默认使用哪些内核功能，总是禁用哪些内核功能
- 允许使用哪些SELinux选项
- 容器是否允许使用可写入的根文件系统
- 允许容器在哪些文件系统组下运行
- 允许pod使用哪些类型的存储卷

查看一个PodSecurityPolicy样例

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: default
spec:
  hostPID: false      // 禁用宿主节点的PID命名空间
  hostIPC: false      // 禁用宿主节点的IPC命名空间
  hostNetwork: false  // 禁用宿主节点的网络命名空间
  hostPorts:   // 限制容器可以绑定宿主节点的哪些端口
  - min: 10000       // 10000-110000 可以绑定
```

```
    max: 11000
  - min: 13000       // 13000-14000 可以绑定
    max: 14000
  privileged: false  // 禁止使用特权模式的容器运行
  readOnlyRootFilesystem: true  // 容器强制使用只读的根文件系统
  runAsUser:         // 以supplemental、fsGroup
    rule: RunAsAny  // 一起允许用户使用任何的用户或用户组运行容器
  fsGroup:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  seLinux:          // 可以使用SELinux的任何选项
    rule: RunAsAny
  volumes:          // 可以使用任何类型的存储卷
  - '*'
12345678910111213141516171819202122232425
```

部署了该PodSecurityPolicy后，集群创建新的资源时将要符合该安全策略才行，比如不能使用privileged 特权模式运行容器等。

了解runAsUser、fsGroup、supplementalGroup策略

前面的例子中没有对runAsUser、fsGroup、supplementalGroup等字段进行任何限制，如果要限制器可以使用的用户和用户组ID，可以将规则改为MustRunAs，并制定允许使用的ID范围

使用MustRunAs规则

```
runAsUser:
  rule: MustRunAs
  range:
  - min: 2   // min=max 可以指定一个特定的ID
    max: 2
fsGroup:
  rule: MustRunAs
  range:    // 指定了用户组ID在 2-10，20-30（包含临界值） 范围内用户组ID
  - min: 2
    max: 10
  - min: 20
    max: 30
supplementalGropus:
  rule: MustRunAs
  range:
  - min: 2
    max: 10
  - min: 20
    max: 30
1234567891011121314151617181920
```

如果pod spec中任何一个字段向设置该范围以外的值，那该pod将不会被API服务器接收。

注意：PodSecurityPolicy策略对已存在的pod无效，因为PodeSecurityPolicy资源只在创建和升级pd的时候生效

部署镜像中用户ID在指定范围以外的pod

通过前面我们知道在pod

spec中使用范围以外的用户ID运行，会被API服务器拒绝。如果在容器镜像文件中Dockerfile

USER指定的用户ID在范围以外，此时容器能被部署，但是进入该容器中可以查看ID，会发现用户运时使用的ID为PodSecurityPolicy策略中指定的用户ID。PodSecurityPolicy可以将编码覆盖到镜像中用户ID。

在runAsUser字段中还可以指定另一种规则：mustRunAsNonRoot。此策略会阻止用户部署以root户运行容器。该情况下，必须在spec中为容器指定runAsUser字段，且不能为0（0为root用户），或在镜像中USER指定一个非0的用户ID。

配置允许、默认添加、禁止使用的内核功能

容器可以运行在特权模式下，也可以通过对每个容器添加或禁用Linux内核功能来定义更加细粒度的限设置。以下三个字段会影响容器使用内核功能。

- allowedCapabilities 指定容器中可以添加的内核功能
- defaultAddCapabilities 指定容器中默认添加的内核功能
- requiredDropCapabilities 指定容器中禁止使用的内核功能

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: xxx
spec:
  allowedCapabilities:
  - SYS_TIME    // 容器可以添加修改系统时间的内核功能
  defaultAddCapabilities:
  - CHOWN       // 容器默认添加可以修改文件所属者的内核功能
  requiredDropCapabilities
  - SYS_ADMIN  // 容器禁用SYS_ADMIN、SYS_MODULE 内核功能
  - SYS_MODULE
  ...
12345678910111213
```

注意：SYS_ADMIN 功能允许使用一系列的管理操作，SYS_MODULE功能允许加载或卸载内核模块

defaultAddCapabilities 字段中列出的所有内核功能将添加到每个pod的每个容器中。如果用户希望个容器不能使用其中某个内核功能，必须在容器的spec中显示禁用该内核功能。

requiredDropCapabilities字段中列出的所有禁用的内核功能，用户如果在SecurityContext.capabilites.add

字段中添加了该字段中的内核功能，将会被API服务器拒绝。PodSecurityPolicy访问控制插件将会每个容器的SecurityContext.Capabilities.drop

字段中加入这些功能。

限制pod可以使用的存储类型

PodSecurityPolicy

可以定义用户再pod中使用哪些存储卷类型。但最低限度上，一个PodSecurityPolicy应该允许pod使以下类型的存储卷：emptyDir、configMap、secret、downwardAPI、persistentVolumeClaim。

```
kind: PodSecurityPolicy
spec:
```

```
  volumes:
  - emptyDir
  - configMap
  - secret
  - downwardAPI
  - persistenVolumeClaim
12345678
```

如果有多个PodSecurityPolicy资源，pod可以使用PodSecurityPolicy中任何一个存储卷类型，实际效的是所有volume列表的合集

对不同的用户或用户组分配不同的PodSecurityPolicy

PodSecurityPolicy是集群级别的资源，它不属于存储或使用在某一个特定的命名空间上。这是否意着它总是会应用在所有的命名空间上呢？不是的，因为这样会使得它们相当难用。毕竟系统pod需要许做一些常规pod不应当做的事情。

对不同的用户或组分配不同的PodSecurityPolicy是通过RBAC机制来实现的。创建你需要的PodSecuriyPolicy资源，然后创建ClusterRole资源，并通过名称将它们指向不同的策略，一次使PodSecurityPolcy资源中的策略对不同的用户或组生效。通过ClusterRoleBinding资源将特定的用户或组绑定到ClustrRole上，当PodSecurityPolicy访问控制插件需要决定是否接纳一个pod时，它只会考虑创建pod的户可以访问到的PodSecurityPolicy中的策略。

创建一个允许部署特权容器的PodSecurityPolicy

```
apiVersion: extentsions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
spec:
  privileged: true
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  volumes:
  - '*'
12345678910111213141516
```

首先创建一个PodSecurityPolicy，允许用户创建特权模式容器的pod

```
kubectl  get psp    // psp 是PodSecurityPolicy的简写
可以看到 default 和 privileged  两个psp 其中default 中PRIV是false
12
```

在部署pod时，如果任一策略允许使用pod中使用到的任何特性，API服务器就会接受这个pod。

现在考虑两个使用该集群的用户A和B，其中A只能创建受限的非特权pod，允许B可以创建特权模式的od。可以通过让A只使用default PodSecurityPolicy，而B 使用以上两个PodSecurityPolicy来做到。

使用RBAC将不同的PodSecurityPolicy分配给不同的用户

RBAC可以给用户授予特定类型的资源的访问权限，但RBAC机制也可以通过使用其名字来授权对特定源实例的访问权限。

先创建两个ClusterRole，分别名为psp-default、psp-privileged

```
kubectl create clusterrole  psp-default --verb=use \
 --resource=podsecuritypolicy  --resource-name=psp-default
kubectl create clusterrole psp-privileged --verb=use \
 --resource=podsecuritypolicy --resource-name=privileged
1234
```

注意：这里使用的动词是use，而不是get list watch或类似的动词

现在需要将这两个策略绑定到用户上，因为ClusterRole是集群资源，所以需要使用ClusterRoleBindig资源而非RoleBinding。

要将 psp-default

ClusterRole绑定到所有已认证的用户上，而不单单是用户A。否则没有用户可以创建pod，因为PodSecurityPolicy访问控制插件会因为没有找到任何策略二拒绝创建pod。所有已认证的用户都属于system authenticated，因此要将psp-default

ClusterRole绑定到这个组。

```
kubectl create clusterrolebinding psp-all-user --clusterrole=psp-default \
  --group=system:authenticated
12
```

将psp-privileged ClusterRole 绑定到用户B

```
kubectl create clusterrolebinding psp-b  --clusterrole=psp-privileged \
  --user=b
12
```

作为一个已认证用户，A现在拥有default PodSecurityPolicy 的权限。而用户B拥有default 和 privileged PodSecurityPolicy的权限，A不能部署特权模式的pod，而用户B可以。

使用kubectl 创建不同用户

如何以用户A或用户B身份通过认证，而非用现在的已认证的用户。首先用kubectl 的子命令config创建两个新用户

```
kubectl config set-credentials a --username=a  --password=password
kubectl config set-credentials b --username=b  --password=password
12
```

然后使用这两个用户去创建特权模式的pod

```
kubectl --user=a create -f  pod-privileged.yaml  // 会提示创建失败
kubectl --user=a create -f  pod-privileged.yaml  // 可以创建成功
12
```

四、隔离pod网络

上面所讲的安全特性配置都是pod和pod中的容器上。现在来了解一下如何通过限制pod与pod之间信，来确保pod之间的网络安全。

是否可以进行这些配置取决于集群中使用的网络插件，如果网络插件支持，可以通过NetworkPolicy源配置网络隔离。

一个NetworkPolicy会应用在匹配它的标签选择器的pod上，指明这些允许访问这些pod的源地址，这些pod可以访问的目标地址。这些分别由入向（ingress）和出向（egress）规则指定。这两者都以匹配由标签选择器选出的pod，或者一个namespace中的所有pod，或者通过无类别域间路由CID指定的IP地址段。

在一个命名空间中启用网络隔离

```
apiVersion: network.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
            // 空的标签选择器会匹配命名空间中所有的pod
1234567
```

在任何一个命名空间内创建该NetworkPolicy之后，任何客户端都不能访问该命名空间中的pod

允许同一个命名空间中的部分pod访问一个服务端pod

```
apiVersion: network.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: xxx
spec:
  podSelector:
    matchLabels:
      app: database  // 确保了对具有app=database标签的pod的访问安全性
  ingress:
  - from:
    - podSlecetor:
        matchLabels:
          app: webserver  // 只允许具有app=webserver标签的pod访问
      ports:
      - port: 5432   // 仅允许对该端口访问
123456789101112131415
```

上面的NetworkPolicy表示，命名空间中的数据库服务，现在只允许webserver才能访问，其他的po都不能访问。并且，webserver只能访问数据库的5432端口。

客户端pod通常通过Service而非直接访问pod来访问服务端拍pod，但这对结果没有影响，Network olicy在通过Service访问时任然生效。

在不同kubernetes命名空间之间进行网络隔离

假设我们有一个多租户使用同一个kubernetes集群。每个租户都有多个命名空间，每个命名空间中有个标签指明它们属于哪个租户。例如，有一个租户manning，它所有命名空间中都有标签tenant:manning。其中的一个命名空间中运行了一个微服务testserver，该微服务具有标签

app:testserver，它只允许同一租户下所有命名空间下的所有pod访问。禁止其他租户访问。

为了确保该微服务的安全，可以创建如下的NetworkPolicy。

```
apiVersion: network.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: xxx
spec:
  podSelector:
    matchLabels:
      app: testserver  // 该策略应用于具有app=testserver标签的pod
  ingress:
  - from:
    - namespaceSelector:
      matchLabels:
        tenant: manning  // 只有具有tenant=manning标签的命名空间中运行的
      ports:                //  pod可以访问该微服务
      - port: 80
12345678910111213141 5
```

注意：在多租户的kubernetes集群中，通常租户不能为他的命名空间添加标签或注释，否则，他们可规避基于namespaceSelector的入向规则。

使用CIDR隔离网络

除了通过再pod选择器或命名空间选择器定义了哪些pod可以访问NetworkPolicy资源中指定的目标pd，还可以通过CIRD表示法指定一个IP段。例如，允许192.168.0.1/24

网段的客户端访问之前的testserver，可以再入向规则中添加一下代码

```
ingress:
- from:
  - ipBlock:
      cidr: 192.168.0.1/24   // 允许IP在该范围内的客户端访问testserve
1234
```

限制pod对外访问流量

```
spec:
  podSelector:
    matchLabels:
      app: webserver   // 这个策略应用于具有app=webserver标签的pod
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: database    // webserver的pod只能与具有app=database标签的pod通信
```