Chapter 4

2019-10-02

# contrail Network Policy (ch3)

## network policy introduction

kubernetes networking model requires all pod can access the the other pods by default. we call this a "flat network" sometimes because it follows a "allow-any-any" model - basically a kubernetes pod can reach any other pods **by default**. this makes the design and implementation of kubernetes networking significantly simplified and much more scalable.

**Note**

In chapter 4 we’ll read more about the requirements that kubernetes enforces on the networking implementation.

on the other hand, with companies large and small rapidly adopting the platform, security has emerged as an important concern. In reality, in many cases certain level of network segmentation methods are required to ensure that only certain pods can talk to each other. that is when kubernetes network policy comes into the picture. a Kubernetes "network policy" defines the access permissions for groups of pods in a way pretty much like a security group in the cloud is used to control access to VM instances.

kubernetes supports network policy via the NetworkPolicy object, which is a Kubernetes resource just like pod, service, ingress, and many others that we’ve learned earlier in this chapter. the role of NetworkPolicy object is to define how groups of pods are allowed to communicate with each other. now let’s explain the way kubernetes network policy works:

1. initially, in a kubernetes cluster, all pods are non-isolated by default. they works in "allow-any-any" model so anyone can talk to any others.
2. now you apply a network policy named policy1 to pod A. in policy policy1, you define a rule to explicitly allow A to talk to pod B. in this case we will call pod A a "target" pod, because it is the pod that the network policy will act on.
3. from this moment on, a few things happen:
   * target pod A can talk to pod B, and can talk to **pod B only**, because B is the only pod you allowed in the policy. due to the nature of the policy rules, we can call the rules a "whitelist".
   * for target pod A only, any connections that are not explicitly allowed by the "whitelist" of this network policy policy1 will be rejected. you don’t need to explicitly define this in policy policy1, it will be enforced by the nature of kubernetes network policy. we can call this an implicit policy
     + the "deny all" policy.
   * as for other non-target pods, for example, pod B or C, which are NOT applied with this network policy policy1, and not any other network policies either, will continue to follow the "allow-any-any" model. therefore they are not "affected" and can continue to communicate to all other pods in the cluster. we can call this another implicit policy - A "allow all" policy.
4. assuming you want pod A to also be able to communicate to pod C, you need to update the network policy policy1 and it’s rules to **explicit** allow it. in another word, you need to keep updating the "whitelist" to allow more traffic types.

as you see, when you define a policy, essentially at least three policies will be applied in the cluster:

* explicit policy1: the network policy you defined, with the whitelist rules allowing certain type of traffic for the selected (target) pod.
* an implicit "deny all" network policy: deny all other traffic that is not in the whitelist for the target pod.
* an implicit "allow all" network policy: allow all other traffic for other non-targeted pods that are not selected by the policy1 network policy. we’ll see "deny all" and "allow all" policies again later in chapter 8.

here are some highlights of kubernetes network policy:

**pod specific.**

network policy specification applies to one pod or a group of pods based on label, same way as RC or Deploy do.

**"whitelist" based rules.**

explicit rules compose a "whitelist", each rule describe a certain type of traffic to be allowed. all other traffic that is not described by any rules in the whitelist will be dropped for the target pod.

**implicit "allow all".**

a pod will be "affected" only if it is selected as the target by any network policy, and it will be "affected" by the selecting network policy only. absence of network policy applied on a pod indicates an implicit "allow all" policy to this pod. in another word, if a non-targeted pod continues its "allow-any-any" networking model.

**seperation of ingress and egress.**

policy rules need to be defined for a specific direction. the direction can be Ingress, Egress, none or both.

**"flow" based (vs. "packet" based).**

once the initiating packet is allowed, the return packet in same flow will also be allowed. suppose an ingress policy applied on pod A allows an ingress HTTP request, then the whole HTTP interaction will be allowed for pod A. this includes the 3 way TCP connection establishment and all data and acknowledgment in both directions.

**Note**

Network polices are implemented by the network component, so you must be using a network solution which supports network policy. Simply creating the NetworkPolicy resource without a controller to implement it will have no effect. in our book contrail is such a network components with network policy implemented. in chapter 8, you will see how these network policies works in contrail.

## network policy definition

like all other objects in kubernetes, network policy can be defined in a yaml file. let’s go ahead to look at an example of it.

**Note**

this is the same example that we’re going to read again in chapter 8.

#policy1-do.yaml  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: policy1  
 namespace: dev  
spec:  
 podSelector:  
 matchLabels:  
 app: webserver-dev  
 policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 10.169.25.20/32  
 - namespaceSelector:  
 matchLabels:  
 project: jtac  
 - podSelector:  
 matchLabels:  
 app: client1-dev  
 ports:  
 - protocol: TCP  
 port: 80  
 egress:  
 - to:  
 - podSelector:  
 matchLabels:  
 app: dbserver-dev  
 ports:  
 - protocol: TCP  
 port: 80

before explaining it in detail, let’s look at the spec part of this yaml file since other sections are self-explanatory after you’ve read yaml file of other objects. the spec has the following structure:

spec:  
 podSelector:  
 ......  
 policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 ......  
 egress:  
 - to:  
 ......

here we see that a network policy definition yaml file can logically be divided into four sections:

* podSelector: this defines the pods selection. it identifies pods where the current Network policy would be applied to.
* policyTypes: specify type of policy rules, Ingress, Egress or both.
* ingress: define the ingress policy rules for the target pods
* egress: define the egress policy rules for the target pods

next we’ll look at each sections in more detail.

### selecting target pods

when you define a network policy, kubernetes needs to know which pods you want this policy to act on. Similar to how the service select its backend pods, the network policy select which pods it will be applied to based on labels:

podSelector:  
 matchLabels:  
 app: webserver-dev

here all pods which has the label app: webserver-dev is selected to be the "target" pods by the network policy. all of the following contents in spec will apply to the target pods only.

### policy types

The second section defines the policyTypes for the target pods.

policyTypes:  
 - Ingress  
 - Egress

policyTypes: - Ingress

it can be either ingress, egress, or both. both types define specific traffic types in the form of one or more rules, which we’ll discuss next.

### policy rules

ingress and egress section defines the direction of traffic, from the selected target pods’s perspective. for example considering the following simplified example:

ingress:  
 - from:  
 - podSelector:  
 matchLabels:  
 app: client1-dev  
 ports:  
 - protocol: TCP  
 port: 80  
 egress:  
 - to:  
 - podSelector:  
 matchLabels:  
 app: client2-dev  
 ports:  
 - protocol: TCP  
 port: 8080

assuming target pod is webserver-dev pod, and there is only one pod client1-dev in the cluster having a matching label client1-dev. two things will happen:

* ingress direction: pod webserver-dev can accept a TCP session with a destination pod 80, initiated from pod client1-dev. this explains why we said kubernetes network policy is "flow" based instead of "packet" based. the TCP connection would not be able to establish if the policy would have been designed on "packet" based, becaue on receiving the incoming TCP "sync", the returning outgoing TCP "sync-ack" would have been rejected without a matching egress policy.
* egress direction: pod webserver-dev can initiate a TCP session with a destination pod 8080, towards pod client1-dev.

**Tip**

for the egress connection to go through, the other end needs to define an ingress policy to allow the incoming connection.

#### network policy rules

each from or to statement defines a rule in the network policy:

* a from statement defines an ingress policy rule
* a to statement defines an egress policy rule
* both rules can optionally has ports statement, which will be discussed later.

so you can define multiple rules to allow complex traffic mode for each direction:

ingress:  
INGRESS RULE1  
INGRESS RULE2  
egress:  
EGRESS RULE1  
EGRESS RULE2

each rule identifies the network endpoints where the target pods can communicate. Network endpoint can be identified by different methods:

* ipBlock: select pods based on ip address block
* namespaceSelector: select pods based on label of namespace (NS), all pods in the matching namespaces will be
* podSelector: select pods based on label of pod

**Note**

podSelector select different things when it is used in different places of a yaml file. previously (under spec directly) it selects pods that the network policy applies, which we’ve called "target" pods. here in a rule (under from or to), it selects which pods the target pods is communicating with. sometime we can call these pods "peering pods", or "endpoints".

so the yaml structure for a rule can look like this:

ingress:  
 - from:  
 - ipBlock:  
 .....  
 - namespaceSelector:  
 .....  
 - podSelector:  
 .....  
 ports:  
 ......

for example in our example:

ingress:  
 - from:  
 - ipBlock:  
 cidr: 10.169.25.20/32  
 - namespaceSelector:  
 matchLabels:  
 project: jtac  
 - podSelector:  
 matchLabels:  
 app: client1-dev  
 ports:  
 - protocol: TCP  
 port: 80  
 egress:  
 - to:  
 - podSelector:  
 matchLabels:  
 app: dbserver-dev  
 ports:  
 - protocol: TCP  
 port: 80

here:

* The ingress network endpoints are
  + subnet 10.169.25.20/32, **or**
  + all pods in namespaces which has the label project: jtac, **or**
  + pods which has the label app: client1-dev in current namespace (namespace of target pod)
* The egress network point is pod dbserver-dev

we’ll come to the ports part soon.

#### AND vs OR

It is also possible to specify only a few pods from namespaces instead of all pods to communicate with. in our example podSelector is used along, which assumes the same namespace as the target pod. another method is to use podSelector along with a namespaceSelector. in that case, the namespaces that the pods belongs to is those with matching labels with namespaceSelector, instead of same as the target pod’s namespace.

for example, assuming the target pod is webserver-dev and its namespace is dev, and only namespace qa has a label "project=qa" matching to the namespaceSelector:

ingress:  
- from:  
 - namespaceSelector:  
 matchLabels:  
 project: qa  
 podSelector:  
 matchLabels:  
 app: client1-qa

here, the target pod can only communicate with those pods that are:

* in namespace qa, **AND** (not **OR**) -
* with label app: client1-dev **in the namespace qa**

Please be careful it is totally different than the below definition, which allow the target pod to talk to those pods that are:

* in namespaces qa, **OR** (not **AND**) -
* with label app: client1-qa **in the target pod’s namespace dev**

ingress:  
- from:  
 - namespaceSelector:  
 matchLabels:  
 project: qa  
 - podSelector:  
 matchLabels:  
 app: client1-qa

#### protocol and ports

it is also possible to specify ports for an ingress and egress rule. protocol type can also be specified along with a protocol port. for example:

egress:  
- to:  
 - podSelector:  
 matchLabels:  
 app: dbserver-dev  
 ports:  
 - protocol: TCP  
 port: 80

ports in ingress says that target pods can allow incoming traffic for the specified ports and protocol. Ports in egress says that target pods can initiate traffic to specified ports and protocol. If ports is not mentioned, all ports and protocols are allowed.

#### line by line explanation

after explaining everything, you should find that the policy rules in our example start to makes more sense, so let’s look at it again in more detail:

podSelector:   
 matchLabels:   
 app: webserver-dev   
 policyTypes:   
 - Ingress   
 - Egress   
 ingress:   
 - from:   
 - ipBlock:   
 cidr: 10.169.25.20/32   
 - namespaceSelector:   
 matchLabels:   
 project: jtac   
 - podSelector:   
 matchLabels:   
 app: client1-dev   
 ports:   
 - protocol: TCP   
 port: 80   
 egress:   
 - to:   
 - podSelector:   
 matchLabels:   
 app: dbserver-dev   
 ports:   
 - protocol: TCP   
 port: 80

from this definition, we now can understand exactly what the network policy is trying to enforce:

* line 1-3: pod webserver-dev is selected by the policy, so it is the "target" pod, all following policy rules will apply on it, and on it only.
* line 4-6: the policy will define rules for both Ingress and Egress traffic
* line 7-19: ingress: section defines the **ingress policy**
  + line 8: from: and line 17: ports, these two sections defines one **policy rule** in ingress policy.
    - line 9-16: these 8 lines under from: section compose an ingress "whitelist":
      1. line 9-10: any incoming data with source IP being 10.169.25.20/32 can access the target pod webserver-dev
      2. line 11-13: any pods under namespace jtac can access target pod webserver-dev
      3. line 14-16: any pod client1-dev can access target pod webserver-dev
    - line 17-19: ports section is second (and optional) part of the same **policy rule**. only TCP port 80 (web service) on target pod webserver-dev is exposed and accessible. access to all other pods will be denied.
  + line 20-26: egress: section defines the **egress policy**
    - line 21: to: and line 24: ports, these two sections define one **policy rule** in egress policy.
      1. line 21-24: these 4 lines under to: section compose an egress "whitelist", here the target pod can send egress traffic to pod dbserver-dev
    - line 25: ports section is second part of the same **policy rule**. the target pod webserver-pod can only start TCP session with destination port of 8080 to other pods.

and that is not all.

if you remember in the beginning of this chapter we’ve talked about the kubernetes default "allow-any-any" network model and the implicit "deny-all", "allow-all" policies, you will realize that so far we just explained the explicit part of it (the policy policy1 in our network policy introduction section). after that, there are two more implicit policies:

* the "deny all" network policy: for the target pod webserver-dev, deny all other traffic that is other than what is explicitly allowed in the above whitelists, this implies at least two rules:
  + ingress: deny all incoming traffic destined to the target pod webserver-dev, other than what is defined in the ingress whitelist.
  + egress: deny all outgoing traffic sourcing from the target pod webserver-dev, other than what is defined in the egress whitelist.
* a "allow all" network policy: allow all traffic for other pods that are not target of this network policy, on both ingress and egress direction.

**Note**

in chapter 8, we’ll take a deeper look at these implicit network policies and their rules in contrail implementation.

## create network policy

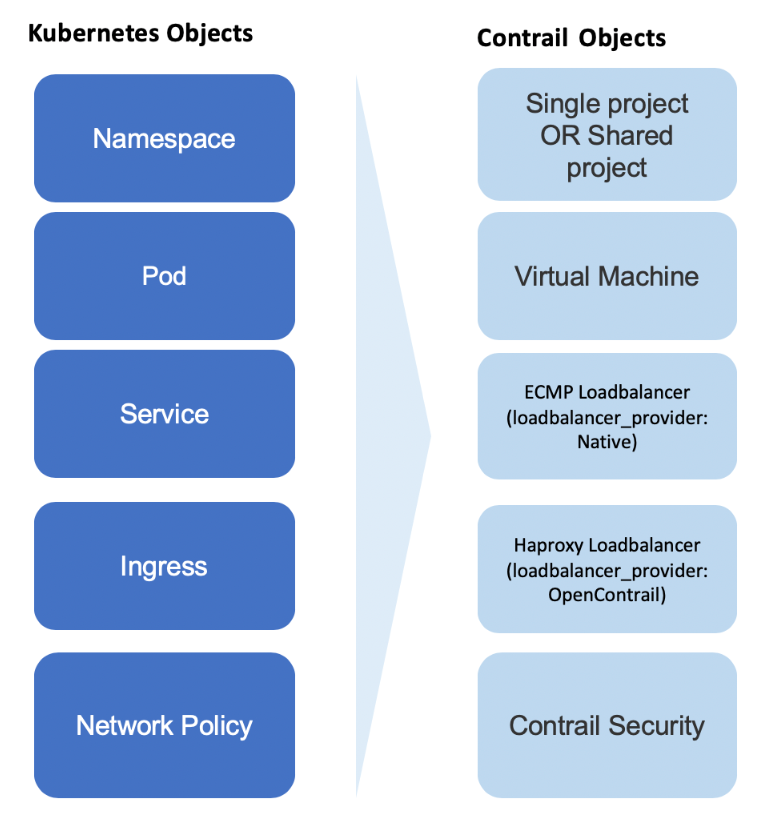
you can create and verify the network policy same way as you create other kubernetes objects:

$ kubectl apply -f policy1-do.yaml  
networkpolicy.networking.k8s.io/policy1-do created  
  
$ kubectl get netpol -n dev  
NAME POD-SELECTOR AGE  
policy1 app=webserver-dev 6s  
  
$ kubectl describe netpol policy -n dev  
Name: policy1  
Namespace: dev  
Created on: 2019-10-01 11:18:19 -0400 EDT  
Labels: <none>  
Annotations: <none>  
Spec:  
 PodSelector: app=webserver-dev  
 Allowing ingress traffic:  
 To Port: 80/TCP  
 From:  
 IPBlock:  
 CIDR: 10.169.25.20/32  
 Except:  
 From:  
 NamespaceSelector: project=jtac  
 From:  
 PodSelector: app=client1-dev  
 Allowing egress traffic:  
 To Port: 80/TCP  
 To:  
 PodSelector: app=dbserver-dev  
 Policy Types: Ingress, Egress

in chapter 8, we’ll setup a test environment to verify the effect of this network policy in more detail.

# Contrail – k8s Network Policy

in chapter 4, we’ve given the "Kubernetes to Contrail Object Mapping" table as shown below:



contrail kubernetes object mapping

the mapping highlights contrail’s implementation of kubernetes core objects: Namespace, pod, Service, Ingress and Network Policy. so far from chapter 4 through 7 we’ve pretty much explored everything except Network  
Policy. in this chapter we’ll focus on the Network Policy implementation in contrail. we’ll first introduce the contrail firewall which is the feature we used to implement kubernetes network policy; we’ll then setup a test case to verify how does kubernetes network policy works in contrail; based on the test result, in the end we’ll explore the contrail firewall policies and their rules in details to understand the contrail implementation, as well as the mapping between the two objects in the object mapping digram.

## introducing Contrail Firewall

In chapter 3 we introduced kubernetes network policy concept. we went through the yaml file definition in details and created the network policy based on it. we’ve also mentioned that simply creating network policy object won’t have any effect, unless the kubernetes networking implementation support it. contrail as a kubernetes CNI implements the kubernetes networking, and it supports the kubernetes network policy through contrail firewall. that is the focus of this chapter - we’ll demonstrate how network policy work in contrail environment through contrail firewall.

before that, let’s review some of the important concept and design in contrail to understand why we implement kubernetes network policy through contrail firewall.

**inter-VN routing.**

in contrail, virtual networks (VN) are isolated by default. that means by default Workloads in VN1 cannot communicate with workloads in another VN VN2. to allow inter-VN communications between VN1 to VN2, additional configuration is required. for example you can use a neutron router, also called "logical router" in contrail, to connect the multiple VNs so inter-VN traffic can be "routed". another commonly used method is to define a "contrail network policy" to connect VNs. contrail network policy also provides security between two virtual networks by allowing or denying specified traffic. actually in this respect, security group is a similar feature. next we’ll talk about each feature briefly.

**contrail network policy.**

A contrail network policy is used to permit inter-VN communication and to modify intra-VN traffic. it describes which traffic is permitted or not between VNs. by default, without a contrail network policy, intra-VN communication is allowed, but inter-VN traffic is denied. when you create a network policy you must associate it with a VN to have any effect.

NOTE: don’t confuse "contrail network policy" with "kubernetes network policy". these are two different security features and they work seperately.

**security group(SG).**

a security group, often abbreviated as a SG, is a group of rules that allow a user to specify the type of traffic that is allowed or not through a **port**. When a VM or pod is created in a VN, a SG can be associated with the VM when it is launched. unlike contrail network policy, which is configured "globally" and associated to the VNs, the SG is configured on the per-port basis, and it will take effect on the specific vrouter flows that is associated with the VM port.

**the limitation of SG and contrail network policy.**

in modern contrail cloud environments, sometimes it is hard to only use the existing network policy and security group to achieve desired security goal. for example: in cloud environments, workloads may move from one server to another and so most likely the IP is changing often. just relying on IP addresses to identify the endpoints to be protected is painful. Instead, users must leverage application level attributes to manipulate policies, so that the policies don’t need to be updated everytime workload moves and the associated network environment changes. also, in production, a user might need to group workloads based on combinations of tags, which is hard to translate into existing language of network policy or Security Group.

**contrail firewall security policy.**

in this chapter we’ll introduce another important feature: "contrail firewall security policy".

Contrail Firewall security policy allows decoupling of routing from security policies and provides multi dimension segmentation and policy portability, while significantly enhancing user visibility and analytics functions.

in order to implement the multi-dimension traffic segmentation, Contrail firewall introduces the concept of "tags". Tags are key-value pairs associated with different entities in the deployment. Tags can be pre-defined or custom/user defined. contrail tags are pretty much the same thing as kubernetes labels. both are used to identify the objects and workloads. as you can see, this is similar to kubernetes network policy design, and it is natural for contrail to use its firewall security policy to implement kubernetes network policy - in theory, contrail network policy or SG can be extended to do the job, but the support of tags by contrail firewall make it much simpler.

**Note**

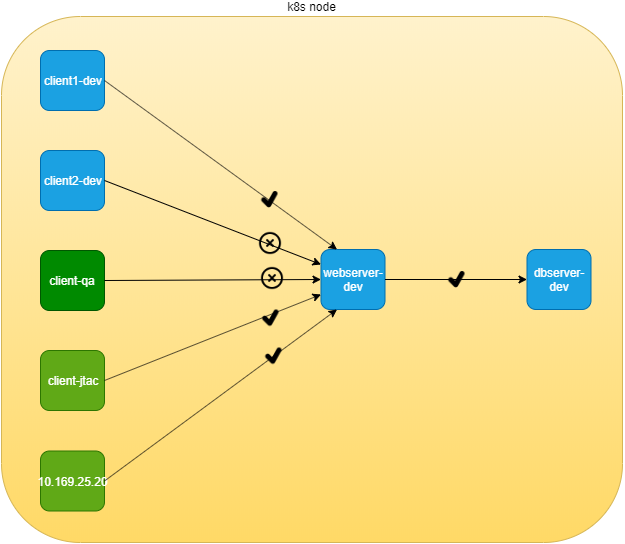
later in this chapter, we’ll sometimes refer "contrail firewall security policy" as "contrail Security", "contrail firewall", "contrail firewall security" or simply "contrail FW".

## contrail kubernetes Network Policy usage case

in this section, lets create a usage case to verify how does the kubernetes network policy works in contrail environments. we’ll start from creating a few kubernetes namespaces and pods resources that is required in the test, confirming every pod can talk to the DUT (Device Under Test) because of the default "allow-any-any" networking model, then creating network policies and observing any changes with same traffic pattern.

### network design

suppose we have this network design:

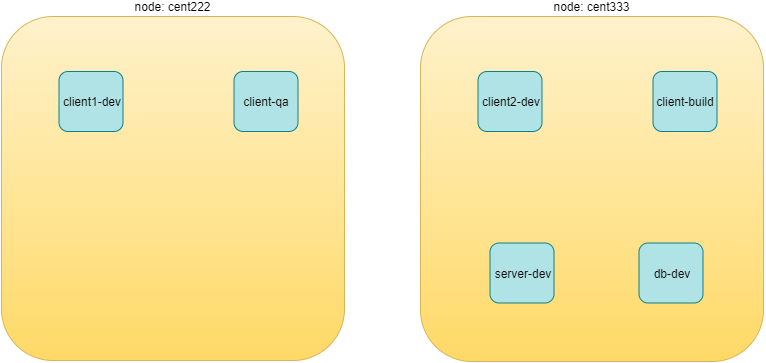


network policy: the test case design

in this didagram, 6 nodes are distributed in 3 departments: dev, qa and jtac. dev department is running a database server (dbserver-dev) holding all valuable data collected from customer. the design requires that no one should has direct access to this db server, instead, db server access is only allowed through another apache frontend server in dev department, named webserver-dev. furthermore, for security reason, the access of customer information should be granted only to authorized clients. for example, only nodes in jtac department, one node in dev department named client1-dev and source IP 10.169.25.20 can access the db via webserver. finally, the database server dbserver-dev should not initiate any connection toward other nodes.

### lab preparation

this is a very ordinary, simplified network design that you will see everywhere. if we model all these network elements in kubernetes world, it will look like this:



network policy: NS and pods

we need to create following resources:

* 3 namespaces: dev, qa, jtac
* 6 pods:
  + 2 server pods: webserver-dev, dbserver-dev
  + 2 client pods in the same namespace as of server pods: client1-dev, client2-dev
  + 2 clients pods from two different namespaces: client-qa, client-jtac
* 2 CIDRs:
  + cidr: 10.169.25.20/32, this is fabric IP of node cent222
  + cidr: 10.169.25.21/32, this is fabric IP of node cent333

kubernetes network policy test environment

|  |  |  |
| --- | --- | --- |
| NS | pod | role |
| dev | client1-dev | web client |
| dev | client2-dev | web client |
| qa | client-qa | web client |
| jtac | client-jtac | web client |
| dev | webserver-dev | webserver serving clients |
| dev | dbserver-dev | dbserver serving webserver |

Lets prepare the required k8s NS and pods resources.

here is the all-in-one yaml file defining dev, qa and jtac namespaces:

#policy-ns-pod.yaml  
##################  
# all namespaces #  
##################  
kind: Namespace  
apiVersion: v1  
metadata:  
 name: dev  
 labels:  
 project: dev  
---  
kind: Namespace  
apiVersion: v1  
metadata:  
 name: qa  
 labels:  
 project: qa  
---  
kind: Namespace  
apiVersion: v1  
metadata:  
 name: jtac  
 labels:  
 project: jtac  
---  
##################  
# all pods #  
##################  
apiVersion: v1  
kind: Pod  
metadata:  
 name: webserver-dev  
 labels:  
 app: webserver-dev  
 do: policy  
 namespace: dev  
spec:  
 containers:  
 - name: webserver  
 image: contrailk8sdayone/contrail-webserver  
 securityContext:  
 privileged: true  
 ports:  
 - containerPort: 80  
---  
apiVersion: v1  
kind: Pod  
metadata:  
 name: client1-dev  
 labels:  
 app: client1-dev  
 do: policy  
 namespace: dev  
spec:  
 containers:  
 - name: ubuntu  
 image: contrailk8sdayone/contrail-webserver  
---  
apiVersion: v1  
kind: Pod  
metadata:  
 name: client2-dev  
 labels:  
 app: client2-dev  
 do: policy  
 namespace: dev  
spec:  
 containers:  
 - name: ubuntu  
 image: contrailk8sdayone/contrail-webserver  
---  
apiVersion: v1  
kind: Pod  
metadata:  
 name: client-qa  
 labels:  
 app: client-qa  
 do: policy  
 namespace: qa  
spec:  
 containers:  
 - name: ubuntu  
 image: contrailk8sdayone/contrail-webserver  
---  
apiVersion: v1  
kind: Pod  
metadata:  
 name: client-jtac  
 labels:  
 app: client-jtac  
 do: policy  
 namespace: jtac  
spec:  
 containers:  
 - name: ubuntu  
 image: contrailk8sdayone/contrail-webserver

**Tip**

ideally, each pods may run based on different images. TCP ports usually are different between a webserver and a database server. in our case to make the test easier, we use the same exact contrail-webserver image that we’ve been using throughout the book for all pods, so clients to webserver and webserver to dbserver all share the same port number 80 served by same HTTP server. also, we add a label do: policy in all pods so that displaying all pods used in this test is also easier.

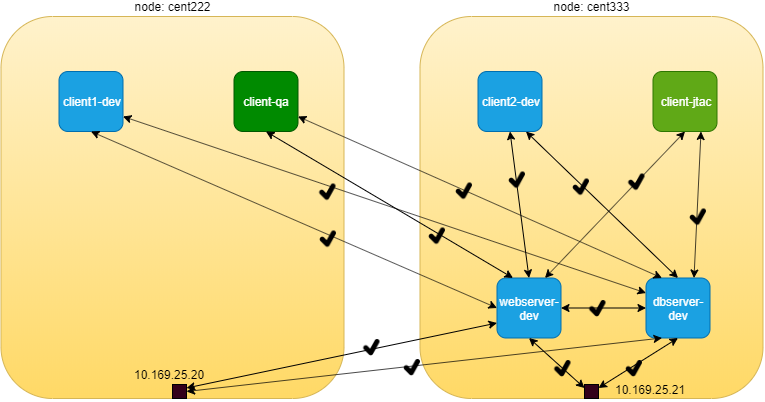
create all resources:

$kubectl create -f policy-ns-pod-do.yaml  
namespace/dev created  
namespace/qa created  
namespace/jtac created  
pod/webserver-dev created  
pod/dbserver-dev created  
pod/client1-dev created  
pod/client2-dev created  
pod/client-qa created  
pod/client-jtac created  
  
$ kubectl get pod --all-namespaces -l "do=policy" -o wide  
NAMESPACE NAME READY STATUS RESTARTS AGE IP NODE  
dev client1-dev 1/1 Running 0 33s 10.47.255.232 cent222  
dev client2-dev 1/1 Running 0 33s 10.47.255.231 cent333  
dev dbserver-dev 1/1 Running 0 33s 10.47.255.233 cent333  
dev webserver-dev 1/1 Running 0 33s 10.47.255.234 cent333  
jtac client-jtac 1/1 Running 0 33s 10.47.255.229 cent222  
qa client-qa 1/1 Running 0 33s 10.47.255.230 cent333

### traffic mode before kubernetes network policy creation

having all of the NS and pods, before we define any network policy yet, we can go ahead to send the traffic between clients and servers.

of course, kubernetes networking by default follows "allow-any-any" model, so we should expect access works between any pod, which is going to be a fully meshed access relationships. but keep in mind that the DUT in this test is webserver-dev and dbserver-dev which we are more interested to observe. to simply the verification, according to our diagram, we’ll focus on accessing the server pods from the client pods, illustrated in below figure:



network policy: pods communication before network policy creation

a few highlights here:

* all clients can access the servers, following the "permit-any-any" model
  + there is no restrictions between clients and webserver-dev pod
  + there is no restrictions between clients and dbserver-dev pod
* the communcation between client and servers are bi-directional and "symmetrical" - each end can "initiate a session" or "accept a session". these matches to the "egress policy" and "ingress policy" respectively in kubernetes network policy’s term.

obviously, these do not meet our design goal, which is exactly why we need kubernetes network policy, we’ll come to that part soon. for now let’s quickly verify the allow-any-any networking model.

first let’s verify the http server running at port 80 in webserver-dev and dbserver-dev pods:

$kubectl exec -it webserver-dev -n dev -- netstat -antp| grep 80  
tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 1/python  
$kubectl exec -it dbserver-dev -n dev -- netstat -antp| grep 80  
tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 1/python

**Tip**

as mentioned earlier, in this test all pods is with the same container image, so all pods are running the same webserver application in their containers. in this test we simply use the name to each pod to reflect their different roles in the diagram.

now we can verify accessing this http server from other pods with these commands:

**test ingress traffic.**

#from master  
dbserverIP=10.47.255.233  
webserverIP=10.47.255.234  
kubectl exec -it client1-dev -n dev -- curl http://$webserverIP -m5  
kubectl exec -it client2-dev -n dev -- curl http://$webserverIP -m5  
kubectl exec -it client-qa -n qa -- curl http://$webserverIP -m5  
kubectl exec -it client-jtac -n jtac -- curl http://$webserverIP -m5  
kubectl exec -it dbserver-dev -n dev -- curl http://$webserverIP -m5  
  
#from node cent222 (fabric interface IP: 10.169.25.20)  
curl http://$webserverIP -m5  
#from node cent333 (fabric interface IP: 10.169.25.21)  
curl http://$webserverIP -m5

these commands triggers the HTTP requests to the webserver-dev pod from all clients and the hosts of the 2 nodes. -m5 curl command option make curl to wait maximum 5 seconds for the response before it claims time out. as expected, all accesses pass through and returns the same output as below:

**from client1-dev:.**

$ kubectl exec -it client1-dev -n dev -- \  
 curl http://$webserverIP | w3m -T text/html | grep -v "^$"  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.234  
 Hostname = webserver-dev

in the command above, w3m get the output from curl which returns a webpage HTML code and renders into readable text, then send to grep to remove the empty lines. to make the command shorter we define an alias:

alias webpr='w3m -T text/html | grep -v "^$"'

now the command looks shorter:

$ kubectl exec -it client1-dev -n dev -- curl http://$webserverIP | webpr  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.234  
 Hostname = webserver-dev

similarly, we’ll get the same test results for access to dbserver-dev from any other pods.

### create kubernetes network policy

now lets create the k8s network policy to implement our design. from out initial design goal, these are what we wanted to achieve via network policy:

* client1-dev and pods under jtac NS (that is jtac-dev pod) can access webserver-dev pod
* webserver-dev pod (and only it) is allowed to access dbserver-dev pod
* all other client pods are not allowed to access the two server pods
* all other client pods can still communicate with each other

translating these requirements into language of kubernetes network policy, we’ll have this network policy yaml file:

#policy1-do.yaml  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: policy1  
 namespace: dev  
spec:  
 podSelector:  
 matchLabels:  
 app: webserver-dev  
 policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 10.169.25.20/32  
 - namespaceSelector:  
 matchLabels:  
 project: jtac  
 - podSelector:  
 matchLabels:  
 app: client1-dev  
 ports:  
 - protocol: TCP  
 port: 80  
 egress:  
 - to:  
 - podSelector:  
 matchLabels:  
 app: dbserver-dev  
 ports:  
 - protocol: TCP  
 port: 80

from the network-policy definition, based on what you’ve learned in chapter 3, you should easily tell what the policy is trying to enforce in our current setup:

* according to the ingress policy, the following clients can reach the webserver-dev server pod located in dev namespace:
  + client1-dev from dev namespace
  + all pods from jtac namespace, that is client-jtac pod in our setup
  + clients with source IP 10.169.25.20 (cent222 in our setup)
* according to the egress policy, the webserver-dev server pod in dev namespace can initiate a TCP session towards dbserver-dev pod with destination port 80 to access the data.
* for target pod server-dev, all other accesses are denied.
* communication between all other pods are not affected by this network policy.

**Tip**

actually, this is the exact network policy yaml file that we’ve demonstrated in chapter 3.

let’s create the policy and verify its effect.

$ kubectl apply -f policy1-do.yaml  
networkpolicy.networking.k8s.io/policy1 created  
  
$ kubectl get networkpolicies --all-namespaces  
NAMESPACE NAME POD-SELECTOR AGE  
dev policy1 app=webserver-dev 17s

### post kubernetes network policy creation

#### ingress policy on webserver-dev

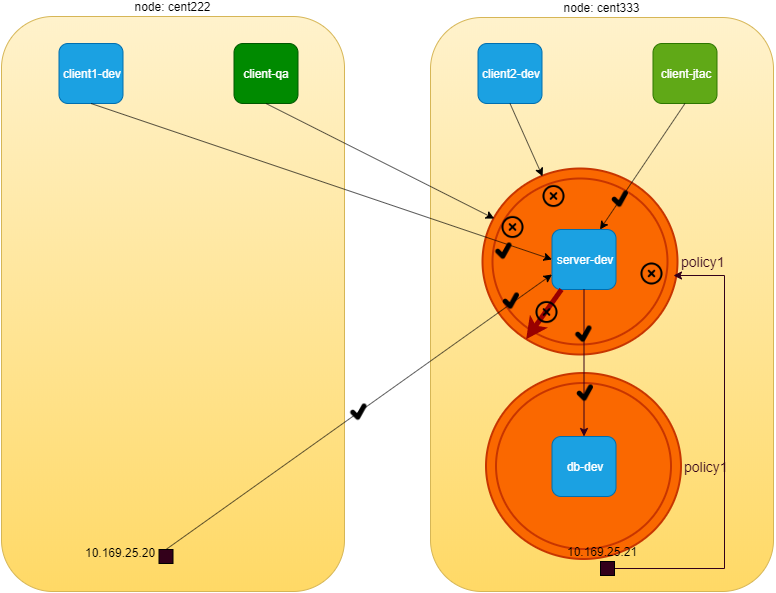
after network policy policy1 is created, let’s test the accessing of http server in webserver-dev pod from pod client1-dev, client-jtac and node cent222 host:

$ kubectl exec -it client1-dev -n dev -- curl http://$webserverIP | webpr  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.234  
 Hostname = webserver-dev

the access from these 2 pod to webserver-dev is OK and that is what we want. now if we repeat the same test from other pods client2-dev, client-qa and another node cent333 now get timed out:

$ kubectl exec -it client2-dev -n dev -- curl http://$webserverIP -m 5  
curl: (28) Connection timed out after 5000 milliseconds  
command terminated with exit code 28  
  
$ kubectl exec -it client-jtac -n jtac -- curl http://$webserverIP -m 5  
curl: (28) Connection timed out after 5000 milliseconds  
command terminated with exit code 28  
  
$ curl http://$webserverIP -m 5  
curl: (28) Connection timed out after 5000 milliseconds

the new test result after network policy applied is illustrated in this figure:



network policy: after applying policy1

detail information of the network policy object tells the same things:

$ kubectl describe netpol -n dev policy1  
Name: policy1  
Namespace: dev  
Created on: 2019-09-29 21:21:14 -0400 EDT  
Labels: <none>  
Annotations: kubectl.kubernetes.io/last-applied-configuration:  
 {"apiVersion":"networking.k8s.io/v1","kind":"NetworkPolicy",  
 "metadata":{"annotations":{},"name":"policy1","namespace":"dev"},  
 "spec":{"egre...  
Spec:  
 PodSelector: app=webserver-dev  
 Allowing ingress traffic: #<---  
 To Port: 80/TCP  
 From:  
 IPBlock:  
 CIDR: 10.169.25.20/32  
 Except:  
 From:  
 NamespaceSelector: project=jtac  
 From:  
 PodSelector: app=client1-dev  
 Allowing egress traffic:  
 To Port: 80/TCP  
 To:  
 PodSelector: app=dbserver-dev  
 Policy Types: Ingress, Egress

From the above exercise, we can conclude that k8s network policy works as expected in contrail.

but out test is not done yet. in the network policy we defined both ingress and egress policy, but so far from webserver-dev pod perspective we’ve only tested that the ingress policy of policy1 works successfully. secondly, we have not apply any policy to the other server pod dbserver-dev. according the default "allow any" policy, any pods can directly access it without a problem. obviously this is not what we wanted according to our original design. another ingress network policy is needed for dbserver-dev pod. and at last, we need to apply an egress policy to dbserver-dev to make sure it can’t connect to any other pods. so there are at least three more test items we need to confirm:

* test egress policy of policy1 applied to webserver-dev pod
* define and test ingress policy for dbserver-dev pod
* define and test egress policy for dbserver-dev pod

let’s look at the egress policy of policy1 first.

#### egress policy on webserver-dev pod

**test egress traffic.**

#test access to all pods  
kubectl exec -it webserver-dev -n dev -- curl http://$dbserverIP -m5  
kubectl exec -it webserver-dev -n dev -- curl http://<other pod IPs> -m5  
  
#test access to all ipBlock  
kubectl exec -it webserver-dev -n dev -- curl http://10.169.25.20 -m5  
kubectl exec -it webserver-dev -n dev -- curl http://10.169.25.21 -m5

the result shows only access to dbserver-dev succeeds. all other egress access is timed out.

$ kubectl exec -it webserver-dev -n dev -- curl $dbserverIP -m5 | webpr  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.233  
 Hostname = dbserver-dev  
$ kubectl exec -it webserver-dev -n dev -- curl 10.47.255.232 -m5  
curl: (28) Connection timed out after 5001 milliseconds  
command terminated with exit code 28

#### network policy on dbserver-dev pod

so far so good. let’s look at the second test items: ingress access to dbserver-dev pod from other pods other than webserver-dev pod:

**test egress traffic.**

#test access to all pods  
kubectl exec -it webserver-dev -n dev -- curl http://$dbserverIP -m5  
kubectl exec -it client1-dev -n dev -- curl http://$dbserverIP -m5  
kubectl exec -it client2-dev -n dev -- curl http://$dbserverIP -m5  
kubectl exec -it client-jtac -n dev -- curl http://$dbserverIP -m5  
kubectl exec -it client-qa -n dev -- curl http://$dbserverIP -m5  
  
#test access to all ipBlock  
#from node cent222 (fabric interface IP: 10.169.25.20)  
curl http://10.47.255.234 -m5  
#from node cent333 (fabric interface IP: 10.169.25.21)  
curl http://10.47.255.234 -m5

all pods can access dbserver-dev pod directly:

$ kubectl exec -it client1-dev -n dev -- curl http://$dbserverIP -m5 | webpr  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.233  
 Hostname = dbserver-dev

our design is to block access from all pods except webserver-dev pod. for that we need to apply another policy. here is the yaml file of the second policy:

#policy-do2.yaml  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: policy2  
 namespace: dev  
spec:  
 podSelector:  
 matchLabels:  
 app: dbserver-dev  
 policyTypes:  
 - Ingress  
 ingress:  
 - from:  
 - podSelector:  
 matchLabels:  
 app: webserver-dev  
 ports:  
 - protocol: TCP  
 port: 80

this network policy policy2 is pretty much like the previous network policy policy1, except that it looks simpler - the policyTypes only has Ingress in the list so it will only define an ingress policy. and that ingress policy defines a whitelist using only a podSelector. in our test case, only one pod webserver-dev has the matching label with it so it will be the only one allowed to initiate the TCP connection toward target pod dbserver-dev on port 80. let’s create the policy policy2 now and very the result again:

$ kubectl exec -it webserver-dev -n dev -- curl http://$dbserverIP -m5 | webpr  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.233  
 Hostname = dbserver-dev  
  
$ kubectl exec -it client1-dev -n dev -- curl http://$dbserverIP -m5 | webpr  
command terminated with exit code 28  
curl: (28) Connection timed out after 5002 milliseconds

now the access to dbserver-dev pod is secured!

#### egress policy on dbserver-dev

there is the one last requirement we haven’t meet in our designed goal. server dbserver-dev should not be able to initiate any connection toward other nodes. which is the final item we need to test in our plan.

now when you review our policy policy2, you may wonder how do we make that happen. we’ve highlighted in chapter 3 that network policy is "whitelist" based only by design. so what ever you put in the whitelist means "allowed". only a "blacklist" gives a "deny", but even with a blacklist you won’t be able to list all other pods just to get them denied.

another thinking is to make the use of the "deny all" implicit policy. so assuming this sequence of policies in current kubernetes network policy design:

1. policy2 on dbserver-dev
2. deny all for dbserver-dev
3. "allow all" for other pods

it looks like if we give an "empty" whitelist in egress policy of dbserver-dev, then nothing will be allowed and the 'deny all' policy for target pod will come into play. problem is how do we define an "empty" whitelist?

apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: policy2-tryout  
 namespace: dev  
spec:  
 podSelector:  
 matchLabels:  
 app: dbserver-dev  
 policyTypes:  
 - Ingress  
 ingress:  
 - from:  
 - podSelector:  
 matchLabels:  
 app: webserver-dev  
 ports:  
 - protocol: TCP  
 port: 80  
 engress: #<---

turn out this doesn’t work as expected.

$ kubectl exec -it dbserver-dev -n dev -- curl http://10.47.255.232 -m5 | webpr  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.232  
 Hostname = client1-dev

checking into the policy object detail does not uncover anything obviously wrong.

$ kubectl describe netpol policy2-tryout -n dev  
Name: policy2-tryout  
Namespace: dev  
Created on: 2019-10-01 17:02:18 -0400 EDT  
Labels: <none>  
Annotations: kubectl.kubernetes.io/last-applied-configuration:  
 {"apiVersion":"networking.k8s.io/v1","kind":"NetworkPolicy",  
 "metadata":{"annotations":{},"name":"policy2-tryout",  
 "namespace":"dev"},"spec"...  
Spec:  
 PodSelector: app=dbserver-dev  
 Allowing ingress traffic:  
 To Port: 80/TCP  
 From:  
 PodSelector: app=webserver-dev  
 Allowing egress traffic:  
 <none> (Selected pods are isolated for egress connectivity) #<---  
 Policy Types: Ingress

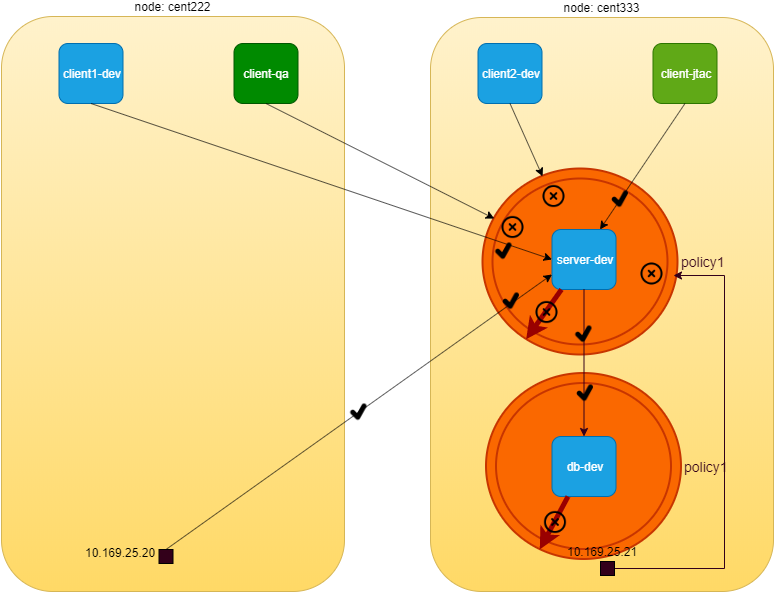
the problem is on the policyTypes here. we haven’t added the Egress in, so whatever configured in egress policy will be ignored. simply adding - Egress in policyTypes will fix it. furthermore, to express an "empty" whitelist the egress: keyword is optional and not required. below is the new policy yaml file:

apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: policy2-egress-denyall  
 namespace: dev  
spec:  
 podSelector:  
 matchLabels:  
 app: dbserver-dev  
 policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - podSelector:  
 matchLabels:  
 app: webserver-dev  
 ports:  
 - protocol: TCP  
 port: 80

now delete the old policy policy2 and apply this new policy, request from dbserver-dev to any other pods (for example pod client1-dev) will be blocked:

$ kubectl exec -it dbserver-dev -n dev -- curl http://10.47.255.232 | webpr  
command terminated with exit code 28  
curl: (7) Failed to connect to 10.47.255.232 port 80: Connection timed out

here is the final diagram illustrating our network policy test result:



network policy: after applying an empty egress policy on dbserver-dev pod

#### the drop action in flow table

before we conclude the test, lets take a look at the vrouter flow table when a traffic is dropped by the policy.

on node cent333 where pod dbserver-dev is located:

$ docker exec -it vrouter\_vrouter-agent\_1 flow --match 10.47.255.232:80  
Flow table(size 80609280, entries 629760)  
  
Entries: Created 33 Added 33 Deleted 30 Changed 54Processed 33 Used Overflow entries 0  
(Created Flows/CPU: 7 9 11 6)(oflows 0)  
  
Action:F=Forward, D=Drop N=NAT(S=SNAT, D=DNAT, Ps=SPAT, Pd=DPAT, L=Link Local Port)  
 Other:K(nh)=Key\_Nexthop, S(nh)=RPF\_Nexthop  
 Flags:E=Evicted, Ec=Evict Candidate, N=New Flow, M=Modified Dm=Delete Marked  
TCP(r=reverse):S=SYN, F=FIN, R=RST, C=HalfClose, E=Established, D=Dead  
  
Listing flows matching ([10.47.255.232]:80)  
  
 Index Source:Port/Destination:Port Proto(V)  
 ----------------------------------------------------------------------------------  
 158672<=>495824 10.47.255.232:80 6 (5)  
 10.47.255.233:42282  
(Gen: 1, K(nh):59, Action:D(Unknown), Flags:, TCP:Sr, QOS:-1, S(nh):63,  
 Stats:0/0, SPort 54194, TTL 0, Sinfo 0.0.0.0)  
  
 495824<=>158672 10.47.255.233:42282 6 (5)  
 10.47.255.232:80  
(Gen: 1, K(nh):59, Action:D(FwPolicy), Flags:, TCP:S, QOS:-1, S(nh):59,  
 Stats:3/222, SPort 52162, TTL 0, Sinfo 8.0.0.0)

the Action:D is set to D(FwPolicy) which means DROP due to Firewall Policy. meanwhile, in the other node cent222 where the pod client1-dev is located, we don’t see any flow generated, indicating the packet does not arrive.

$ docker exec -it vrouter\_vrouter-agent\_1 flow --match 10.47.255.233  
Flow table(size 80609280, entries 629760)  
  
......  
  
Listing flows matching ([10.47.255.233]:\*)  
  
 Index Source:Port/Destination:Port Proto(V)  
 ----------------------------------------------------------------------------------

## contrail implementation details

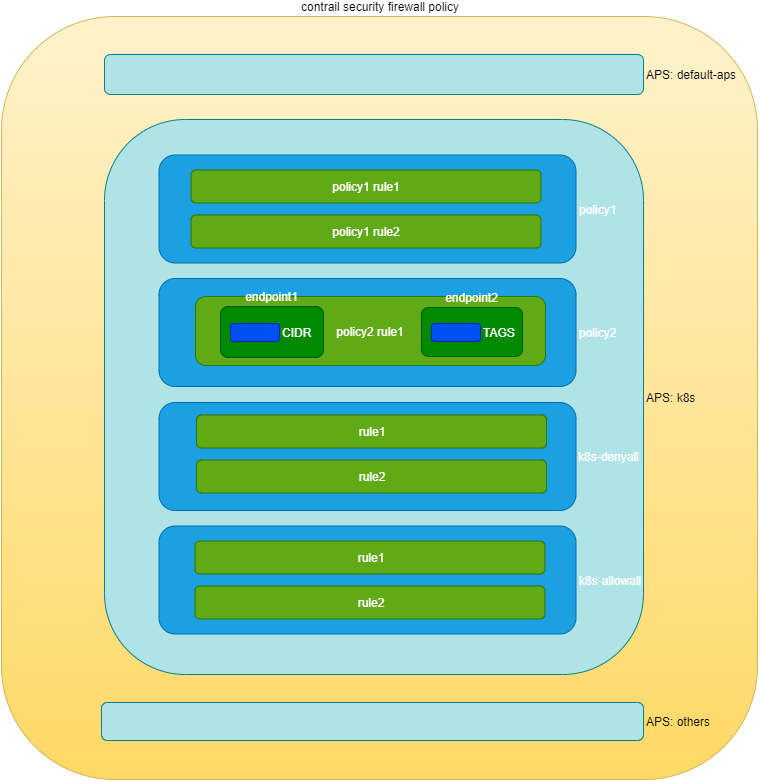
we’ve introduce that contrail implements kubernetes network policy by Contrail firewall security Policy. you also knows that Kubernetes labels are exposed as tags in contrail. these tags are used by contrail security policy to implement specified Kubernetes policies. tags will be created automatically from kubernetes objects labels or created in the UI manually.

In this section we’ll take a closer look at the contrail firewall policies, policy rules, and the tags. especially, we’ll examine the mapping relationships between the kubernetes object that we created and tested in the last section, and the corresponding contrail objects in contrail firewall system.

Contrail Firewall is designed with a hierarchical structure:

* the top level object is named "Application Policy Set", abbreviated as APS
* APS has Firewall Policies;
* Firewall Policy has Firewall Rules;
* Firewall rules has the endpoints;
* Endpoints can be identified via tags or address groups (CIDRs).

the structure is illustrated in this figure:



contrail firewall

### construct mappings

Kubernetes network policy and contrail firewall policy are two different entities in terms of the semantics in which network policy is specified in each. in order for contrail firewall to implement kubernetes network policy, contrail needs to implement the one to one mapping for a lot of data construct from kubernetes to contrail firewall. these data constructs are the basic building blocks of kubernetes network policy and the corresponding contrail firewall policy.

Below table represents kubernetes network policy constructs and the corresponding constructs in contrail:

|  |  |
| --- | --- |
| K8s Network Policy Construct | Contrail Firewall Construct |
| Cluster Name | APS (one per k8s cluster) |
| Network Policy | Firewall Policy (one per k8s network policy) |
| Ingress and Egress policy rule | Firewall Rule (one per k8s ingress/egress policy rule) |
| CIDR | Address Group(one per k8s network policy CIDR ) |
| Label | Tag (one for each k8s label) |
| Namespace | Custom Tag (one for each namespace) |

contrail-kube-manager, the KM, as we’ve read many time earlier in this book, does all of the translations between the two worlds. basically the following will happen in the context of kubernetes network policy:

1. KM will create a APS with Kubernetes cluster name during its initialing process. typically the default kubernetes cluster name is k8s, so you will see an APS with the same name in your cluster
2. KM registers to kube-apiserver to watch the network policies events.
3. Whenever a kubernetes network policy is created, a corresponding contrail firewall policy will be created with all matching firewall rules and network endpoints.
4. for each label in kubernetes object there will be a corresponding contrail tag created
5. based on the tag, the corresponding contrail objects (VN, pods, VMI, projects, etc) can be located.
6. contrail will then apply the contrail firewall policies and rules in the APS on the contrail objects, this is how the specific traffic is permited or denied.

**Note**

APS can be associated to different contrail objects, e.g.

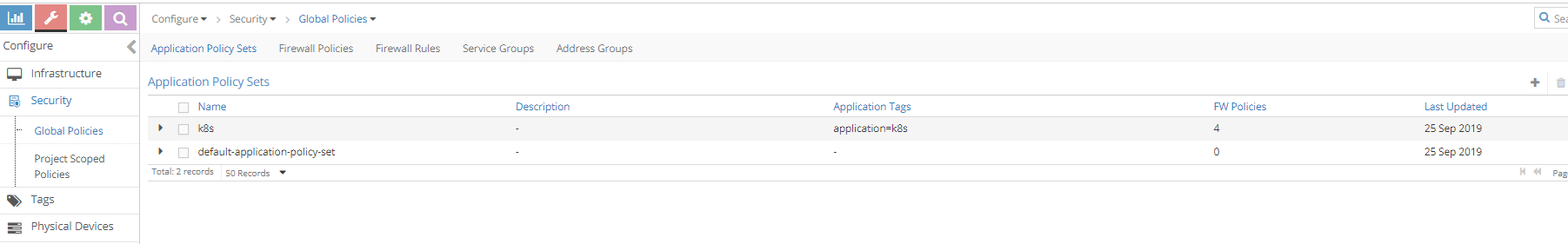
* VMI(virtual machine interface)
* VM (virtual machine) or pods
* VN (virtual network)
* project

In contrail kubernetes cluster, it is associated to virtual network. Whenever traffic goes on those networks, firewall policies associated on the APS would be evaluated and respective action would be taken for the traffic.

in the previous section, we have created two kubernetes network policies in our usage case. now lets explore the contrail objects that are created for these kubernetes network policies.

### Application Policy Set (APS)

As mentioned above, contrail-kube-manager will create an Application Policy Set(APS) using the kubernetes cluster name during the initialization stage. in chapter 3 when we introduce "Contrail Namespaces and Isolation", we’ve learned the cluster name is k8s by default in contrail. therefore the APS name will also be k8s in the contrail UI.



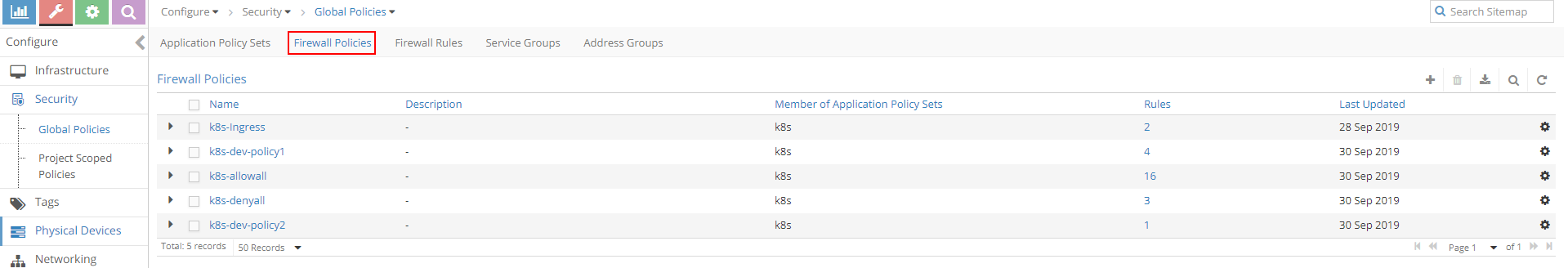
contrail UI: APS: configure → Security → Global Policies → "Application Policy Sets"

There is one more APS default-application-policy-set which is created by default.

### policies

now clicking on the "Firewall Policies" to display all firewall polices in the cluster. in our test environment, you will find the following policies available:

* k8s-dev-policy1
* k8s-dev-policy2
* k8s-denyall
* k8s-allowall
* k8s-Ingress



contrail UI:"Firewall Policies"

#### contrail firewall policy naming convention

the k8s-dev-policy1 and k8s-dev-policy2 policies are what we’ve created. although they looks different from the object name we gave in our yaml file, it is easy to tell which one is which. when KM creates the contrail firewall policies based on the kubernetes network policies, it prefixes the firewall policy name with the cluster name, plus the namespace, in front of our network policy name:

<cluster name>-<namespace-name>-<kubernetes network policy name>

this sounds familiar. earlier we’ve showed how KM name the VN in contrail UI after the kubernetes VN objects name we created in yaml file.

the K8s-ingress firewall policy is created for the ingress loadbalancer. basically this is to ensure the Ingress to work properly in contrail. the detail explanation is out of the scope of this book, so we can ignore it here.

#### the k8s-allowall and k8s-denyall firewall policy

but the bigger question is that why we still see 2 more firewall policies here, because we had never created any network policies like allowall, or denyall?

remember when we introduce kubernetes network policy in chapter 3, we’ve mentioned kubernetes network policy uses "whitelist" method and the implicit "deny all" and "allow all" policies. the nature of "whitelist" method indicates "deny all" action for all traffic other than what is added in the whitelist, while the implicit "allow all" behavior make sure a pod that is not involved in any network policies can continue its "allow-any-any" traffic model. the problem with contrail firewall regarding these implicitness is that, by default it follows a "deny all" model - anything that is not explicitly defined will be blocked. that is why in contrail implementation, these two corresponding implicit network policies are honored by two explicitly policies generated by the KM module.

one question may be raised at this point. with multiple firewall policies, which one should be applied and evaluated first and which ones afterward? in another word, in what "sequence" shall contrail apply and evaluate each policy? firewall policies evaluation with a different sequence will lead to completely different result. just imagine these two sequences "denyall - allowall" vs "allowall- denyall", the former give a pass to all other pods, while the latter give a stop.

the anwer is the "sequence number".

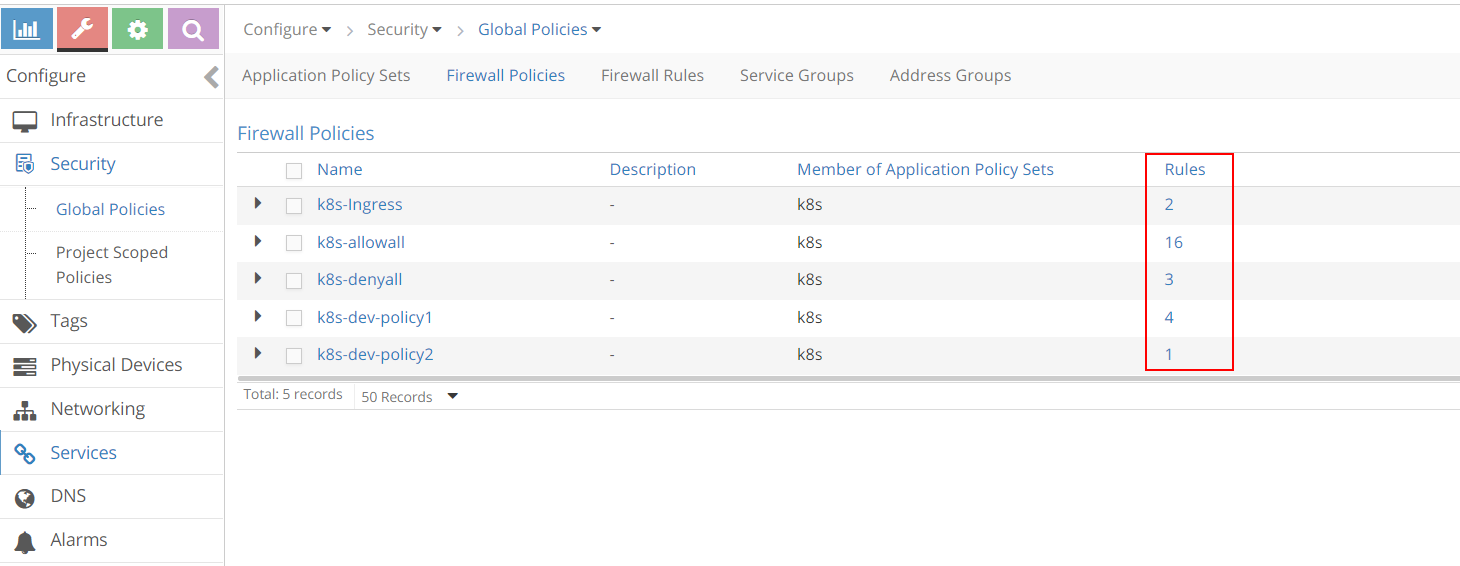
#### sequence number

When firewall polices in an APS are evaluated, it has to be evaluated in a certain sequence. all firewall polices and all firewall rules (will come to this soon) in each of the policy has a sequence number. When there is a matching policy, it will be executed, and the evaluation will stop. it is again contrail-Kube-manager that allocates the right sequence number for all firewall policies and firewall rules, so that everythings works in correct order. the process is automatically done without manual intervention. we don’t have to worry about these things when we create the kubernetes network policies.

we’ll visit sequence number again later, now let’s look at the rules defined in the firewall policy.

### firewall policy rules

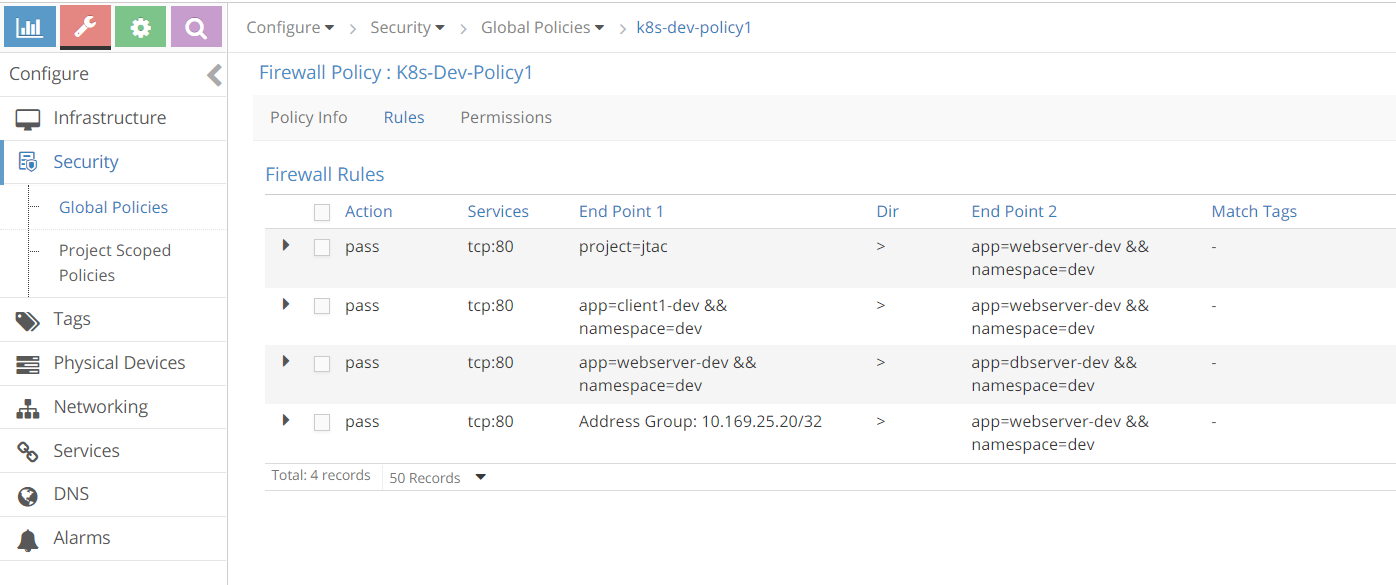
in the same view of "Firewall policies" list, in the right side we see number of "Rules" for each policy:



contrail UI:"Firewall Policy rules"

#### rules in k8s-dev-policy1 firewall policy

there are 4 rules for the k8s-dev-policy1 policy. Clicking at it we will see the rules in detail:



contrail UI:"k8s-dev-policy1" rules

doesn’t it look familiar with our kubernetes network policy policy1 that we’ve tested? let’s put the rules displayed in the screnshot into a table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rule# | Action | Services | End Point1 | Dir | End Point2 | Match Tags |
| 1 | pass | tcp:80 | project=jtac | > | app=webserver-dev && namespace=dev | - |
| 2 | pass | tcp:80 | app=client1-dev && namespace=dev | > | app=webserver-dev && namespace=dev | - |
| 3 | pass | tcp:80 | app=webserver-dev && namespace=dev | > | app=dbserver-dev && namespace=dev | - |
| 4 | pass | tcp:80 | Address Group: 10.169.25.20/32 | > | app=webserver-dev && namespace=dev | - |

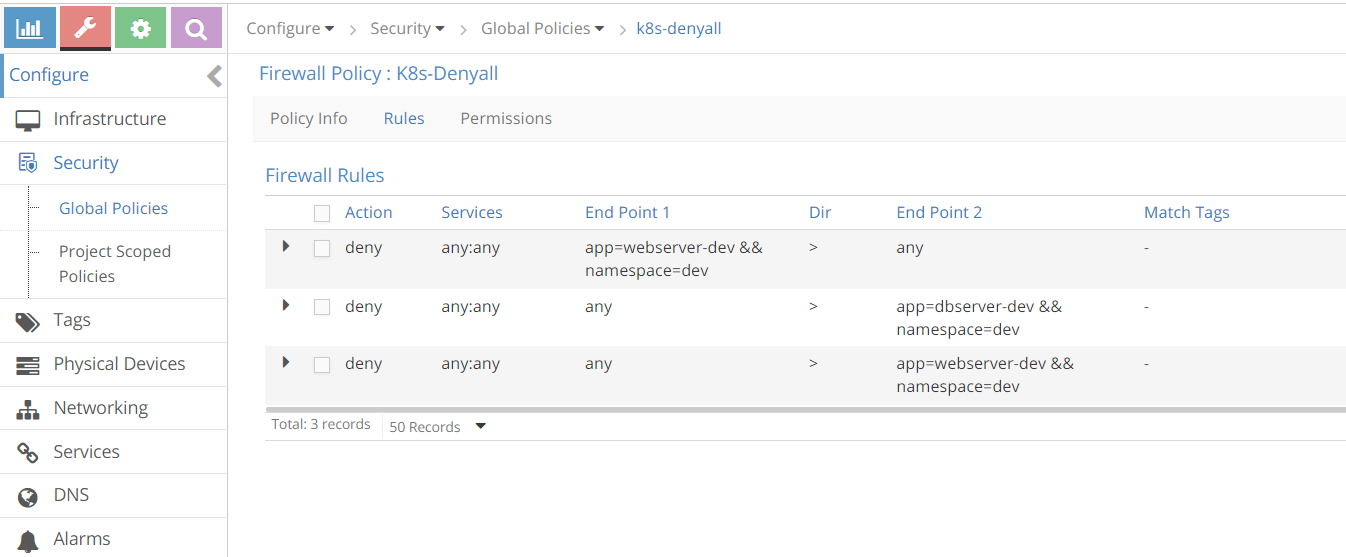
the first column is the rule number that we added, all other columns are imported from the UI screenshot. now compare it with the kubernetes object information:

$ kubectl get netpol --all-namespaces -o yaml  
apiVersion: v1  
items:  
- apiVersion: extensions/v1beta1  
 kind: NetworkPolicy  
 metadata:  
 ......  
 spec:  
 egress:  
 - ports:  
 - port: 80  
 protocol: TCP  
 to:  
 - podSelector: #<---rule#3  
 matchLabels:  
 app: dbserver-dev  
 ingress:  
 - from:  
 - ipBlock: #<---rule#4  
 cidr: 10.169.25.20/32  
 - namespaceSelector: #<---rule#1  
 matchLabels:  
 project: jtac  
 - podSelector: #<---rule#2  
 matchLabels:  
 app: client1-dev  
 ports:  
 - port: 80  
 protocol: TCP  
 podSelector:  
 matchLabels:  
 app: webserver-dev  
 policyTypes:  
 - Ingress  
 - Egress

all rules we see in firewall policy k8s-dev-policy1 matches with rules in kubernetes network policy policy1.

#### rules in k8s-denyall firewall policy

now let’s go back and exmine the rules in k8s-denyall policy that KM generated for our kubernetes network policies.

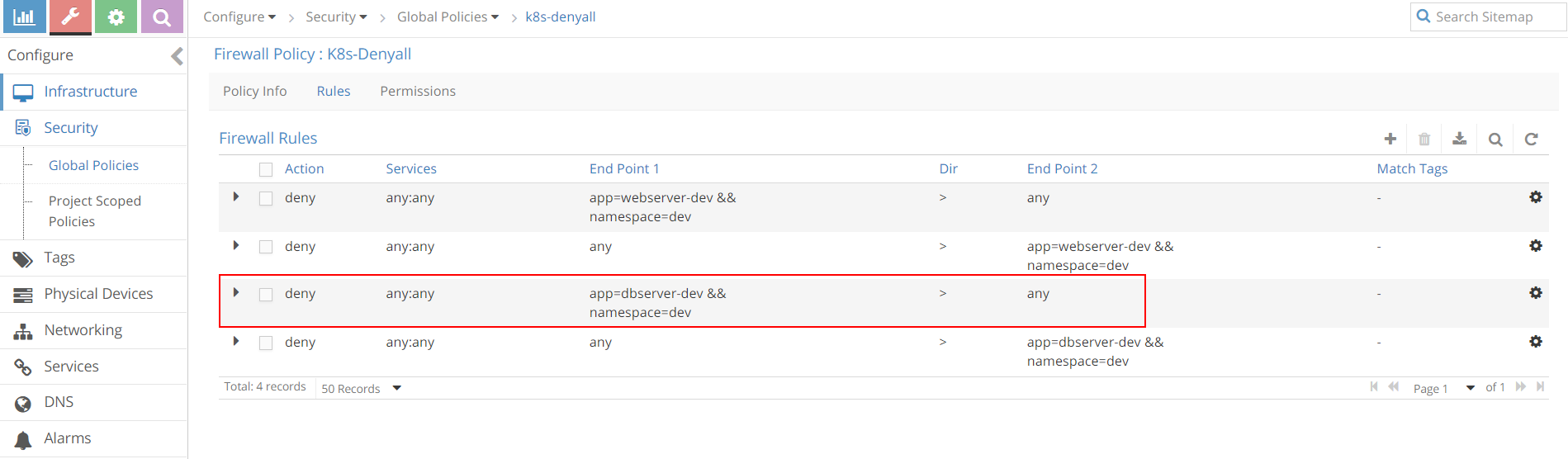


contrail UI:"k8s-denyall" rules

again we convert that into a table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rule# | Action | Services | End Point1 | Dir | End Point2 | Match Tags |
| 1 | deny | any:any | app=webserver-dev && namespace=dev | > | any | - |
| 2 | deny | any:any | any | > | app=dbserver-dev && namespace=dev | - |
| 3 | deny | any:any | any | > | app=webserver-dev && namespace=dev | - |

the k8s-alldeny rules are simple. it just tell contrail to deny communication with all other pods that is not in the whitelist. one thing worth to mention is that there is a rule in the direction from app=webserver-dev && namespace=dev to any, so that egress traffic is denied for webserver-dev pod, while there is no such a rule from app=dbserver-dev && namespace=dev to any. if you review our test in last section, in the original policy policy2, we did not define an Egress option in policyTypes to deny egress traffic of dbserver-dev, that is why when translated into contrail firewall there is no such a rule either. if we change policy2 to the new policy policy2-egress-denyall and examine the same, we’ll see the "missing" rule now:

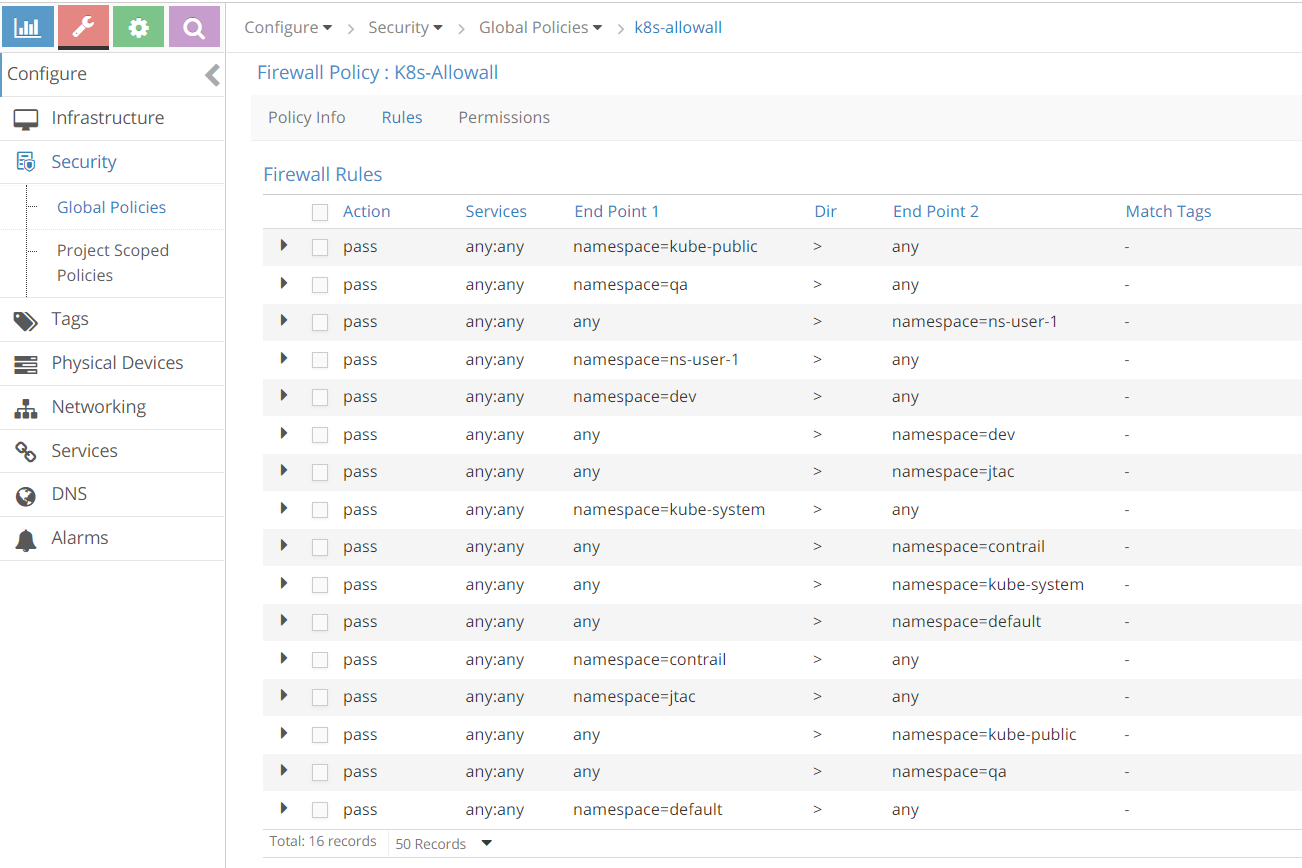


contrail UI:"k8s-denyall" rules

please pay attention that the "k8s-denyall" policy only apply to those "target" pods - pods that are selected by the network policies. in our case it only applies to pods webserver-dev and dbserver-dev. other pods like client-jtac or client-qa will not be affected. instead, those pods will be applied by k8s-allowany policy, which we will examine the next.

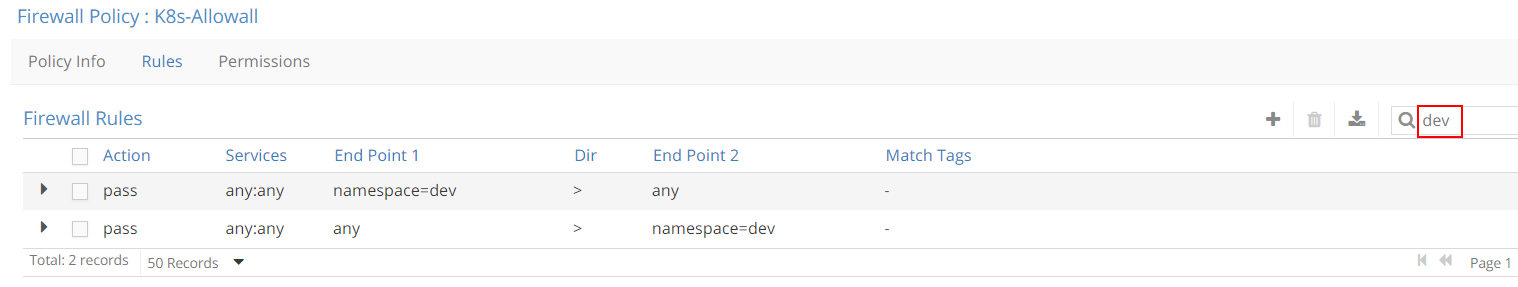
#### rules in k8s-allowall firewall policy

the k8s-allowall policy seems to have more rules than other policies:

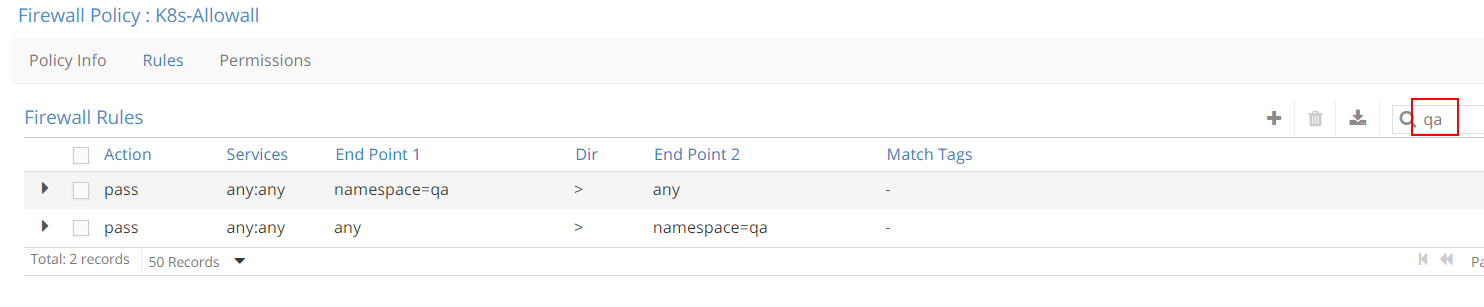


contrail UI:"k8s-allowall" rules

despite the number of rules, in fact k8s-allowall is the simplest one. it works at the NS level and simply has two rules for each NS. in UI within the "search" box, apply a namespace as the filter e.g. "jtac" or "qa", we’ll see these results:



contrail UI:"k8s-allowall" rules filtered by NS "jtac"



contrail UI:"k8s-allowall" rules filtered by NS "qa"

basically what this policy says is: for those pods that does not has any network policy applied yet, let’s continue the kubernetes default "allow-any-any" networking model and allow everything!

### sequence number

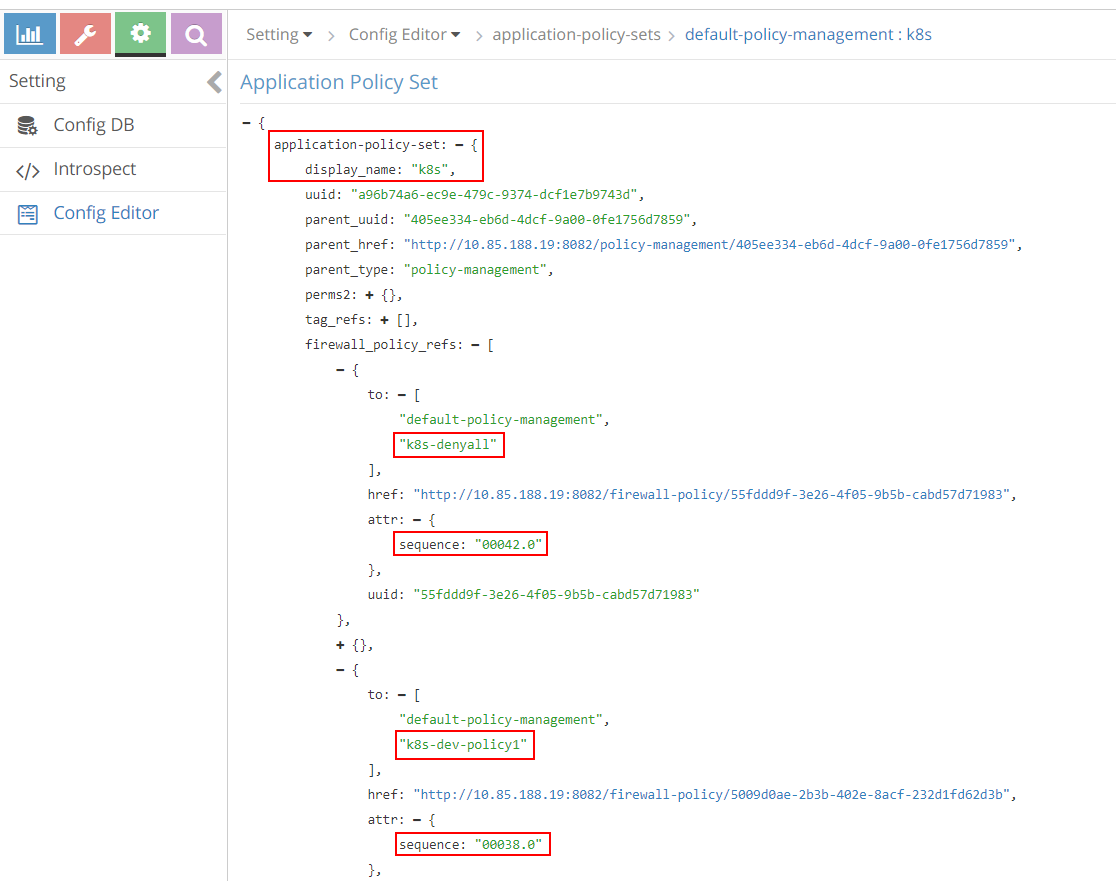
after having explored the contrail firewall policy rules, let’s come back to the sequence number and see how it works exactly.

sequence number is a number attached in all firewall-policies and their rules. it decides the order in which all policies are applied and evaluated, similarly, in one particular policy, it decides the order in which all rules are applied and evaluated. the lower the sequence number the higher the priority. to find the sequence number we have to look into the firewall policy and policy rule object attributes in contrail configuration database. first let’s explore the firewall policy object in APS to check their sequence number.

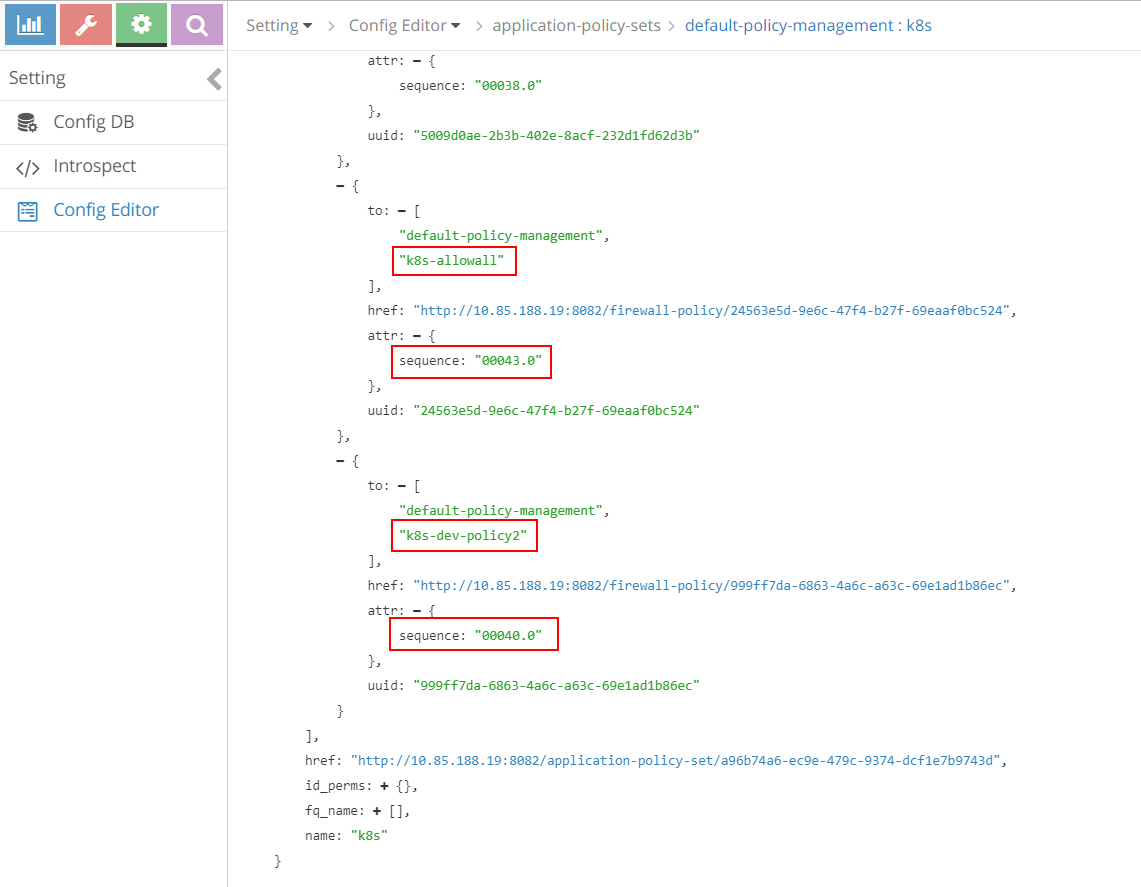
**Tip**

in chapter 5 we’ve used curl command to pull the loadbalancer object data when we introduce service. here we use "Config Editor" to do the same.

#### sequence number in firewall policies



contrail UI:sequence number for policies: "setting" → "Config Editor"



contrail UI:sequence number for policies: (continue)

in these screenshots, under APS k8s there are all the 5 policies that we’ve saw. for example, the policy k8s-dev-policy1 which maps to the kubernetes network policy policy1 that we explicitly defined, and the policy k8s-denyall which is what the system automatically generated. in the figure it shows k8s-dev-policy1 and k8s-denyall has a sequence number as "00038.0" and "00042.0" respectively. therefore, k8s-dev-policy1 has a higher priority and it will be applied and also evaluated first. that means the traffic types we defined in whitelist will be allowed first, then all other traffic to or from the target pod will be denied. this is the exact goal that we want to achieve.

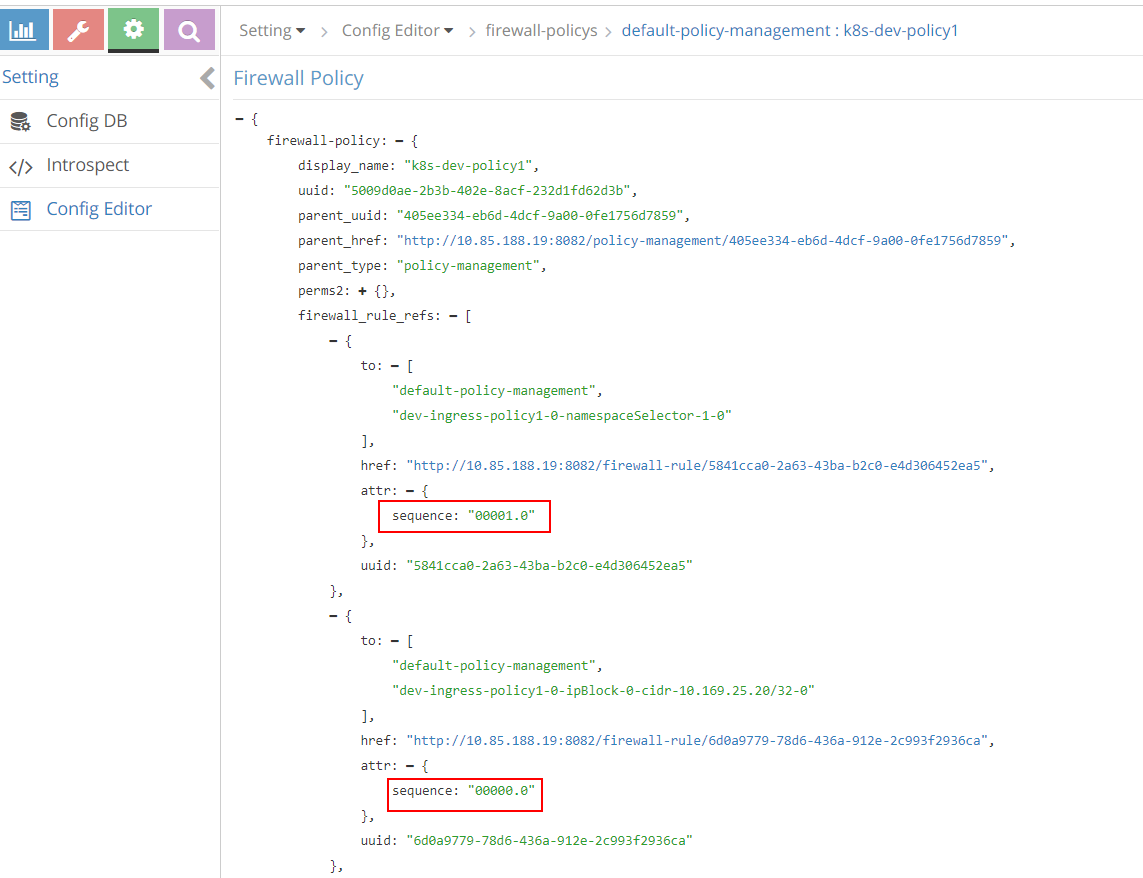
all sequence number for all firewall policies are listed in below table, from highest priority to the lowest:

|  |  |
| --- | --- |
| seq# | firewall policy |
| 00002.0 | k8s-Ingress |
| 00038.0 | k8s-dev-policy1 |
| 00040.0 | k8s-dev-policy2 |
| 00042.0 | k8s-denyall |
| 00043.0 | k8s-allowall |

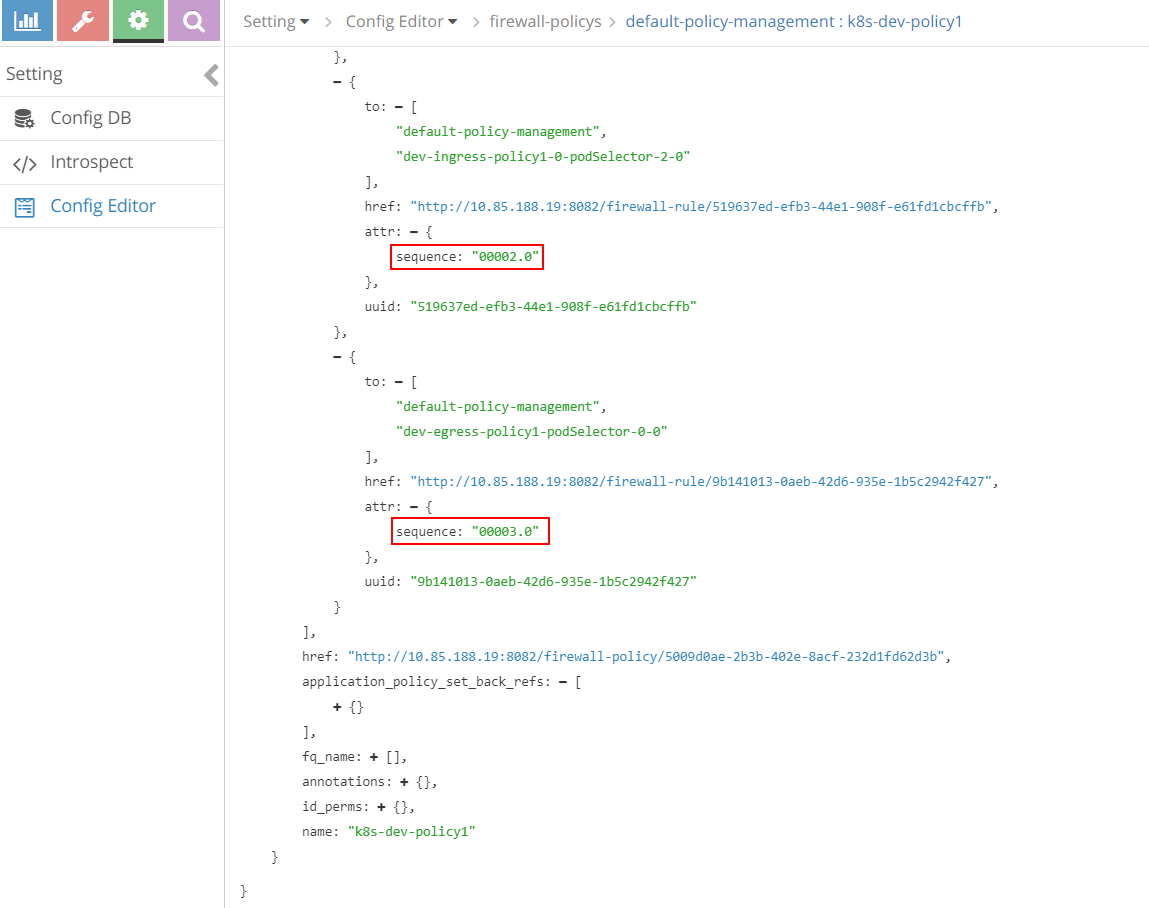
based on the sequence number, the application and evaluation order is the explicit policies first, followed by the "deny all" policy and then the "allow all" policy at the last. the same order as in kubernetes is honored. next let’s check the sequence number in policy rules.

#### sequence number in firewall policy rules

as mentioned, in the same firewall policy, policy rules will also have to be applied and evaluated in a certain order. in contrail firewall that is again ensured by the sequence number. the sequence number in rules of firewall policy k8s-dev-policy1 is displayed in below figures:



contrail UI:sequence number for rules: "setting" → "Config Editor"



contrail UI:sequence number for rules: (continue)

below table shows sequence number of all rules of firewall policy k8s-dev-policy1, from highest priority to the lowest:

|  |  |
| --- | --- |
| seq# | firewall rule |
| 00000.0 | dev-ingress-policy1-0-ipBlock-0-cidr-10.169.25.20/32-0 |
| 00001.0 | dev-ingress-policy1-0-namespaceSelector-1-0 |
| 00002.0 | dev-ingress-policy1-0-podSelector-2-0 |
| 00003.0 | dev-egress-policy1-podSelector-0-0 |

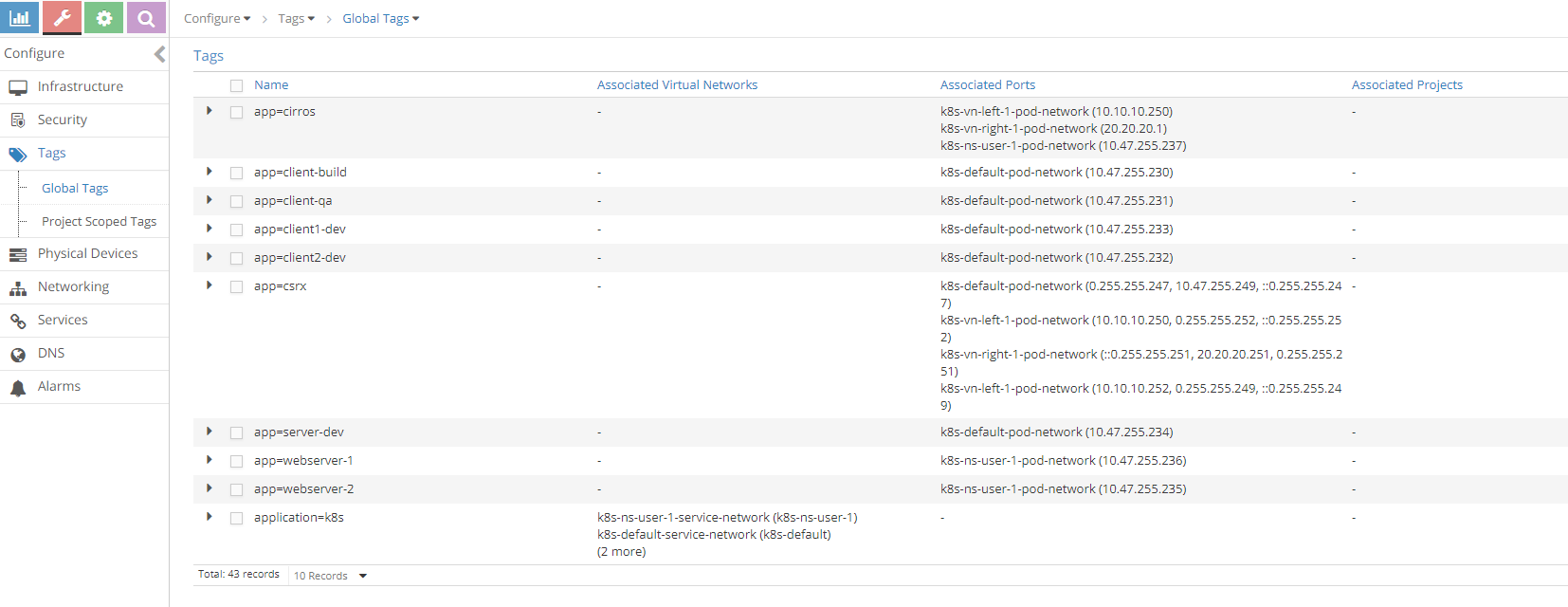
comparing with our network policy yaml file configuration:

ingress:  
 - from:  
 - ipBlock:  
 cidr: 10.169.25.20/32 #<---seq# 00000.0  
 - namespaceSelector: #<---seq# 00001.0  
 matchLabels:  
 project: jtac  
 - podSelector: #<---seq# 00002.0  
 matchLabels:  
 app: client1-dev  
 ports:  
 - protocol: TCP  
 port: 80  
 egress:  
 - to:  
 - podSelector: #<---seq# 00003.0  
 matchLabels:  
 app: dbserver-dev  
 ports:  
 - protocol: TCP  
 port: 80

we can find that the rules sequence number is consistent with the sequence they appear in the yaml file. in another word, rules will be applied and evaluated in the same order as they are defined.

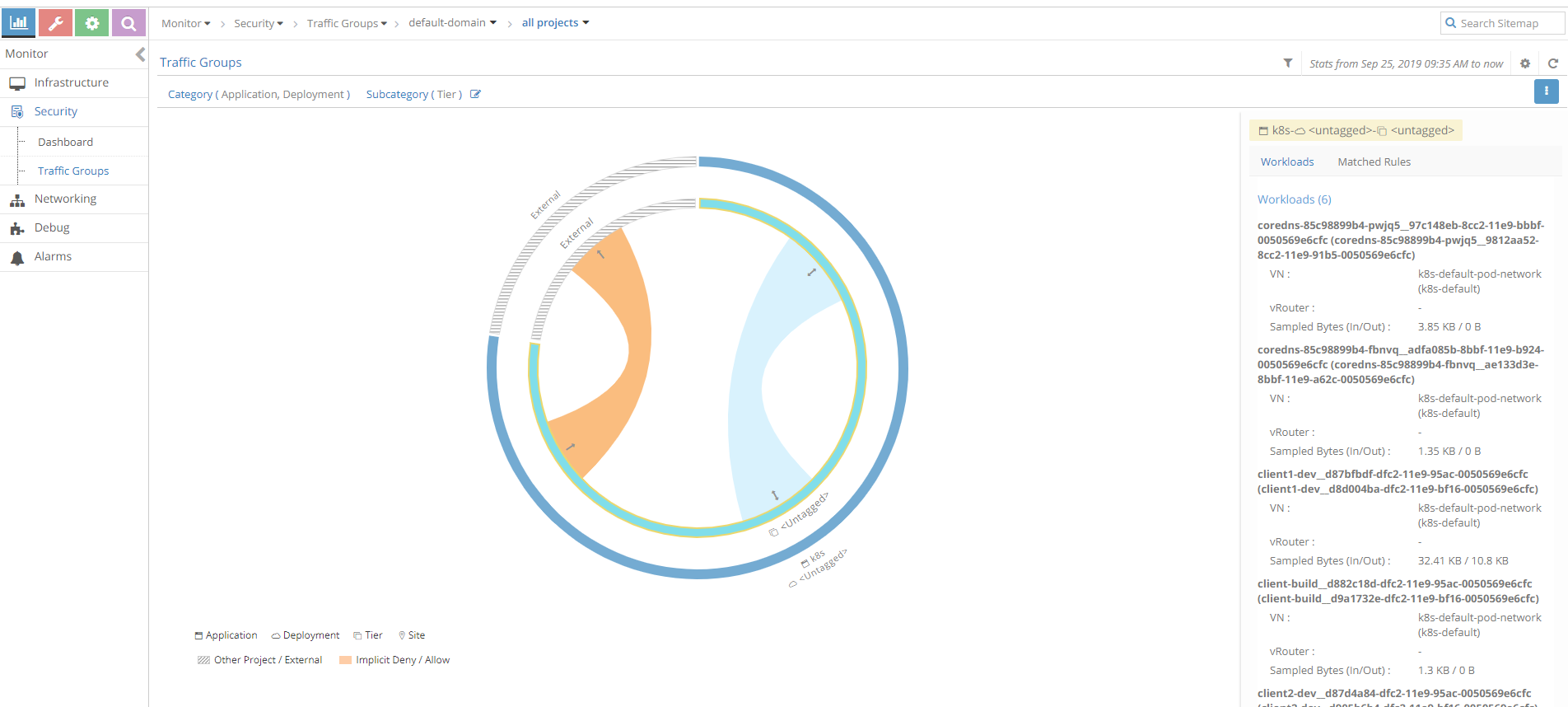
### tag

we’ve been talking about the contrail tags and we already know that contrail-kube-manager will translate each kubernetes label into a contrail tag, which is attached to the respective port of a pod.

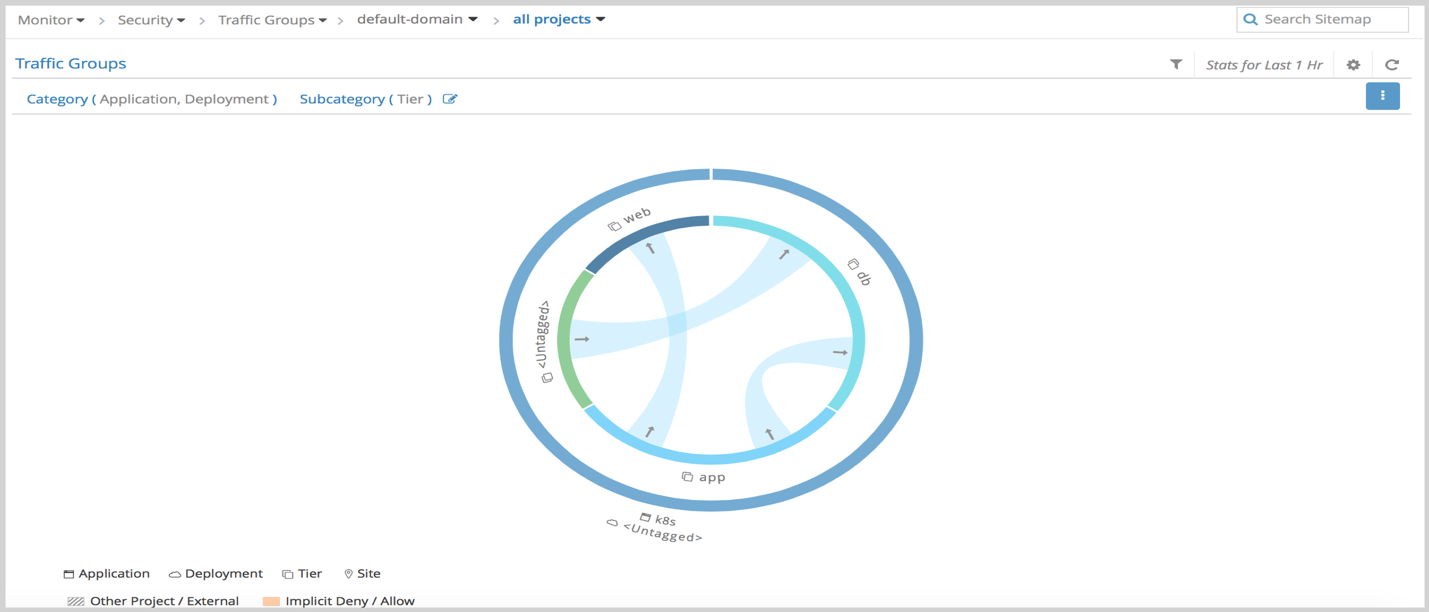


### UI visualization

Contrail UI provides nice visualization for security. It is self explanatory if you know how contrail security works.



Sample traffic visualization for the above policy with workload



Sample traffic visualization with more network policies