Chapter 4

2019-09-30

# contrail Network Policy (ch3)

## network policy introduction

if you remember in chapter 4, we’ve introduced the "flat network" requirement in kubernetes network, and in there you already learned that a flat network assumes a "allow-any-any" model - basically a kubernetes pod can reach any other pods **by default**. this surely makes the design and implementation of kubernetes networking significantly simplified and much more scalable.

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, etc. the role of it 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 to allow A to talk to pod B. from this moment on, pod A will talk to **pod B only**, because B is the only pod in the "whitelist".
3. for pod A only, any connections that are not explicitly allowed by this network policy policy1 will be rejected. you don’t need to explicitly define this in policy policy1 or any other policies, it will be enforced by the nature of kubernetes network policy. we can call this implicit policy as "deny all" policy
4. as for other pods, for example, pod B, 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 so it can communicate to all other pods in the cluster. we can call this implicit policy as "allow all" policy
5. assuming you want pod A to also be able to communicate to pod C, you need to update the network policy policy1 to **explicit** allow it. in another word, you need to keep updating the "whitelist" to allow more traffic types.
6. multiple network-polices can be applied to a pod. for example, you can add a second policy policy2 which also select pod A to apply its forwarding rules.

as you see, whenever you define a policy, essentially you will apply at least three policies in the cluster:

* policy1: the network policy you defined, with the whitelist rules of allowed traffic for the selected pod
* a "deny all" network policy: deny all other traffic that is not in the whitelist for the selected pod
* a "allow all" network policy: allow all other traffic for other pods that are not selected by the policy1 network policy

here are some highlights of kubernetes network policy:

**pod specific.**

network policy specification applies to 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 applied pod.

**implicit "allow all".**

a pod will be "affected" only if it is selected by any network policy, and it will be "affected" by the selected 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 pod is not selected by any network policy, it continues its "allow-any-any" model.

**seperation of ingress and egress.**

any policy rules need to be defined for a specific direction. the direction can be ingress, egress, none or both.

**flow based.**

once the initiating packet is allowed, the return packet in same flow will also be allowed. suppose an ingress policy applied on pod allows an ingress HTTP request, then the whole HTTP interaction will be allowed. 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 resource without a controller to implement it will have no effect. in chapter 8, you will see how these network policies looks like in contrail.

## network policy definition

like all other objects in kubernetes, network policy can be defined in a yaml file. here is an example that we’re going to test later in this chapter:

# policy-do.yaml  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: policy-do  
 namespace: dev  
spec:  
 podSelector:  
 matchLabels:  
 app: server-dev  
 policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 10.169.25.20/32  
 - namespaceSelector:  
 matchLabels:  
 project: qa  
 - podSelector:  
 matchLabels:  
 app: client1-dev  
 ports:  
 - protocol: TCP  
 port: 80  
 egress:  
 - to:  
 - ipBlock:  
 cidr: 10.169.25.21/32  
 - podSelector:  
 matchLabels:  
 app: db-dev  
 ports:  
 - protocol: TCP  
 port: 80

look at the spec part of this yaml file, it has the following structures:

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

Network policy definition can logically be divided into four sections:

1. podSelector: this defines the pods selection identifing pods where the Network policy would be applied
2. policyTypes: specify type of policy rules, Ingress, Egress or both.
3. ingress: define the ingress policy rules for the selected pods
4. egress: define the egress policy rules for the selected pods

next we’ll look at each sections.

### podSelector

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: server-dev

here the network policy would be applied to all pods which has the label app:  
server-dev. all of the following contents in spec will apply to the selected pods only.

### policyTypes

The second section defines the policyTypes for the selected pods. 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.

policyTypes:  
 - Ingress  
 - Egress

### ingress and egress

ingress and egress section defines the direction of traffic initiator, from the selected pods 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 selected pod is server-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 server-dev can accept a TCP session with a destination pod 80, initiated from pod client1-dev
* egress direction: pod server-dev can initiate a TCP session with a destination pod 8080, towards pod client1-dev.

#### network policy rule

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

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 selected 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)
* podSelector: select pods based on label of pod

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

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

for example:

policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 10.169.25.0/24  
 except:  
 - 10.169.25.21/32  
 - namespaceSelector:  
 matchLabels:  
 project: qa  
 - podSelector:  
 matchLabels:  
 app: client1-dev  
 egress:  
 - to:  
 - ipBlock:  
 cidr: 10.169.25.20/32

here:

* The ingress network points are
  + subnet 10.169.25.0/24 except 10.169.25.21/32, **or**
  + all pods in namespaces which has the label project: qa, **or**
  + pods which has the label app: client1-dev
* The egress network point is 10.169.25.20/32

#### AND vs OR

It is also possible to select only a few pods from namespaces instead of all pods. podSelector can be specified along with namespaceSelector. in that case, network endpoint would be only those pods with matching labels in the that namespaces, instead of all pods in the selected pod’s namespace. for example:

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

here, the selected pod can only communicate with those pods:

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

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

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

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

#### ports

it is also possible to specify ports for an ingress and egress rule.

Ports in ingress says that selected pods can allow incoming traffic for the specified ports. Ports in egress says that selected pods can initiate traffic to specified ports. If it is not mentioned, all ports 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:

podSelector:   
 matchLabels:   
 app: server-dev   
 policyTypes:   
 - Ingress   
 - Egress   
 ingress:   
 - from:   
 - ipBlock:   
 cidr: 10.169.25.20/32   
 - namespaceSelector:   
 matchLabels:   
 project: qa   
 - podSelector:   
 matchLabels:   
 app: client1-dev   
 ports:   
 - protocol: TCP   
 port: 80   
 egress:   
 - to:   
 - ipBlock:   
 cidr: 10.169.25.21/32   
 ports:   
 - protocol: TCP   
 port: 8080

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

* line 1-3: pod server-dev is selected by the policy, so 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 a ingress "whitelist":
      1. line 9-10: any incoming data with source IP being 10.169.25.20/32 can access the selected pod server-dev
      2. line 11-13: any pods under namespace qa can access selected pod server-dev
      3. line 14-16: any pod client1-dev can access selected pod server-dev
    - line 17-19: ports section is second (and optional) part of the same **policy rule**. only TCP port 80 (web service) on selected pod server-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 section s defines one **policy rule** in egress policy.
      1. line 21-23: these 3 lines under to: section compose a egress "whitelist", here the selected pod can send egress traffic to IP 10.169.25.21.
    - line 24: ports section is second part of the same **policy rule**. the selected pod server-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 here we are not done yet. so far we just explained the explicit part of it (the policy policy1 in our network policy introduction section) .

* the "deny all" network policy: for the selected pod server-pod, 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 traffic destined to the selected pod server-pod, other than what is defined in the ingress whitelist.
  + egress: deny all traffic destined to the selected pod server-pod, other than what is defined in the egress whitelist.
* a "allow all" network policy: allow all traffic for other pods that are not selected by this network policy
  + ingress: allow all traffic
  + egress: allow all traffic

**Note**

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

## create network policy

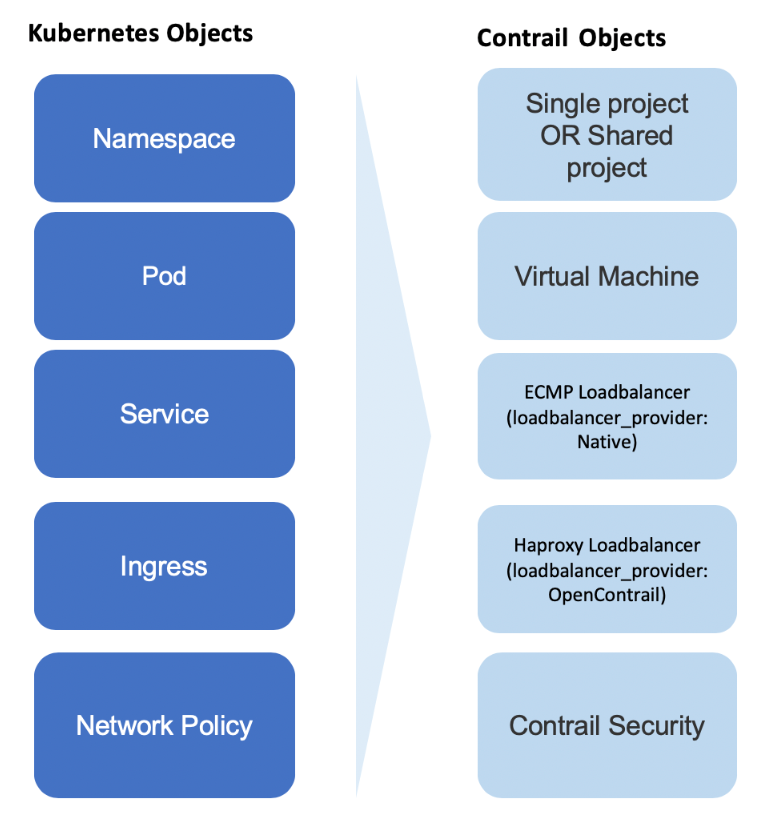
you can create and verify the network policy same way as you create other kubernetes objects:

$ kubectl apply -f policy-do.yaml  
networkpolicy.networking.k8s.io/policy-do created  
  
$ kubectl get netpol -n dev  
NAME POD-SELECTOR AGE  
network-policy-do app=server-dev 6s  
  
$ kubectl describe netpol policy-do -n dev  
Name: policy-do  
Namespace: dev  
Created on: 2019-09-25 23:16:31 -0400 EDT  
Labels: <none>  
Annotations: kubectl.kubernetes.io/last-applied-configuration:  
 {"apiVersion":"networking.k8s.io/v1","kind":"NetworkPolicy",  
 "metadata":{"annotations":{},"name":"policy-do",  
 "namespace":"dev"},"spec":{"eg...  
Spec:  
 PodSelector: app=server-dev  
 Allowing ingress traffic:  
 To Port: 80/TCP  
 From:  
 IPBlock:  
 CIDR: 10.169.25.20/32  
 Except:  
 From:  
 NamespaceSelector: project=qa  
 From:  
 PodSelector: app=client1-dev  
 Allowing egress traffic:  
 To Port: 8080/TCP  
 To:  
 IPBlock:  
 CIDR: 10.169.25.21/32  
 Except:  
 Policy Types: Ingress, Egress

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

# Contrail – k8s Network Policy

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



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.

## 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. . remember in there 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 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 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 permitted, to pass between virtual networks(VN). 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. several policies may be associated with one VN at the same time, and each policy contains a list of rules that are evaluated in the top-down fashion. the evaluation ends when the first match is found, which is known as "terminal behavior"

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 allowed) to pass through a **port**. When a VM or pod is created in a VN, a security group 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.**

As the contrail environment has grown and become more complex, it has become harder to achieve desired security results with the existing network policy and security group constructs. for example: In modern cloud environments, workloads are moving from one server to another, one rack to another and so on. Therefore, users must rely less on using IP addresses or other network coordinates to identify the endpoints to be protected. Instead, users must leverage application attributes to author policies, so that the policies don’t need to be updated on account of workload mobility.

Additionally, a user might need to group workloads based on combinations of tags. These intents are hard to express with existing network policy constructs or Security Group constructs. Besides, existing policy constructs leveraging the network coordinates, must continually be rewritten or updated each time workloads move and their associated network coordinates change.

**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.

Contrail FW Security Policy introduces the concept of tags to achieve multi-dimension traffic segmentation among various entities, with security features. Tags are key-value pairs associated with different entities in the deployment. Tags can be pre-defined or custom/user defined. contrail tags are same as kubernetes labels and both are used to identify the workloads. as you can see, this is similar to kubernetes network policy design, so it is natural for contrail to use its firewall security policy to implement kubernetes network policy.

**Note**

While Kubernetes network policy can be implemented using other security objects in Contrail like Security Groups, Contrail network policies etc, the support of tags by Contrail FW Security Policy aids in the simplification and abstraction of workloads.

the implementation has the following advanatages:

1. Workloads can be represented and grouped by tags.
2. Combinational tags can be used in policies.
3. Untagged workloads can be specified in policies.
4. Policies can be applied in various layers.

**Note**

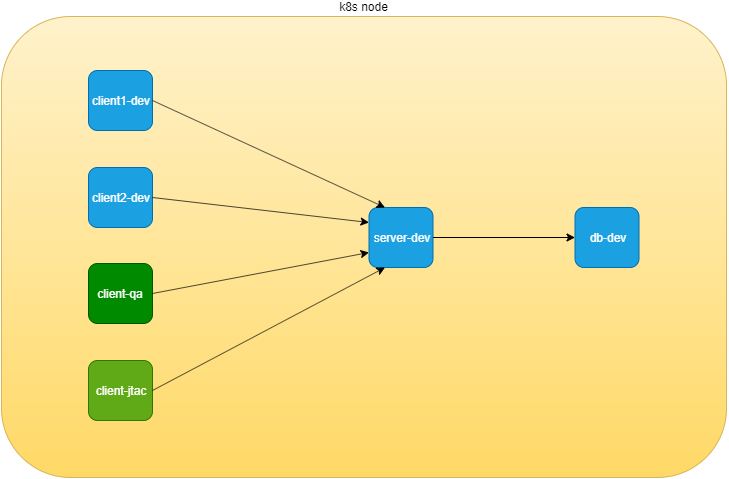
later in this chapter, we’ll sometimes refer "contrail firewall security policy" as "contrail Security", "contrail firewall", or "contrail FW".

## kubernetes Network Policy test in contrail

in this section, lets 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.

### lab preparation

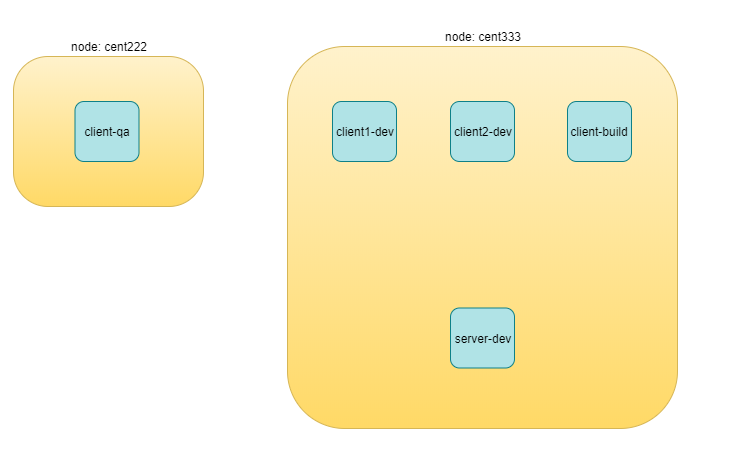
suppose we have this network design:



NS and pods to test network policy

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 cases and data collected from customer. the design requires that no one should has direct access to this db server, instead, db 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 one node in dev department named client1-dev and all nodes in department qa can access the db via webserver.

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:



NS and pods to test network policy

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 web server |

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**

we use the contrail-webserver and ubuntu image that we’ve been using throughout the book, for server and clients repectively. also, we add a label do: policy in all pods so that displaying all pods used in this test is easier.

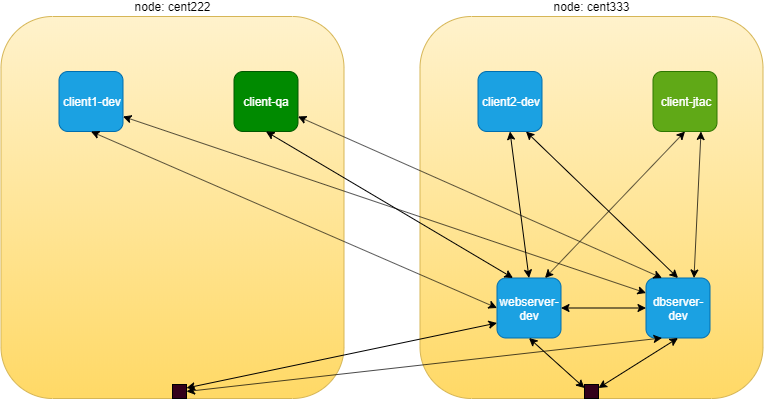
create all resources:

$kubectl create -f policy-ns-pod-do.yaml  
namespace/dev created  
namespace/qa created  
namespace/jtac created  
pod/webserver-dev created  
pod/client1-dev created  
pod/client2-dev created  
pod/client-qa created  
pod/client-jtac created  
  
$ kubectl get pod -l 'do=policy' -o wide --all-namespaces  
NAMESPACE NAME READY STATUS RESTARTS AGE IP NODE  
jtac client-jtac 1/1 Running 0 42s 10.47.255.230 cent333  
dev client1-dev 1/1 Running 0 42s 10.47.255.233 cent333  
dev client2-dev 1/1 Running 0 42s 10.47.255.232 cent333  
dev webserver-dev 1/1 Running 0 42s 10.47.255.234 cent333  
qa client-qa 1/1 Running 0 42s 10.47.255.231 cent222

### test traffic 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 a fully meshed access relationships. to simply the verification, according to our diagram, we’ll focus on accessing the server pods from the client pods, illustrated in below figure:



pods communication before network policy creation

a few highlights here:

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

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

first let’s verify the httpserver 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**

actually 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 assign different roles to each pod.

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

**test ingress traffic.**

#from master  
kubectl exec -it client1-dev -n dev -- curl http://10.47.255.234 -m5  
kubectl exec -it client2-dev -n dev -- curl http://10.47.255.234 -m5  
kubectl exec -it client-qa -n qa -- curl http://10.47.255.234 -m5  
kubectl exec -it client-jtac -n jtac -- curl http://10.47.255.234 -m5  
kubectl exec -it dbserver-dev -n dev -- curl http://10.47.255.234 -m5  
  
#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

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://10.47.255.234  
<html>  
<style>  
 h1 {color:green}  
 h2 {color:red}  
</style>  
 <div align="center">  
 <head>  
 <title>Contrail Pod</title>  
 </head>  
 <body>  
 <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b>  
 pod</h2><br><h3>IP address = 10.47.255.234<br>Hostname = webserver-dev</h3>  
 </body>  
 </div>  
</html>

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

### 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
  + client-jtac from jtac namespace
  + 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 IP 10.169.25.21 with destination port 8080.
* for selected 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/network-policy-do created  
$ kubectl get networkpolicies --all-namespaces  
NAMESPACE NAME POD-SELECTOR AGE  
dev network-policy-do app=webserver-dev 17s

### post kubernetes network policy creation

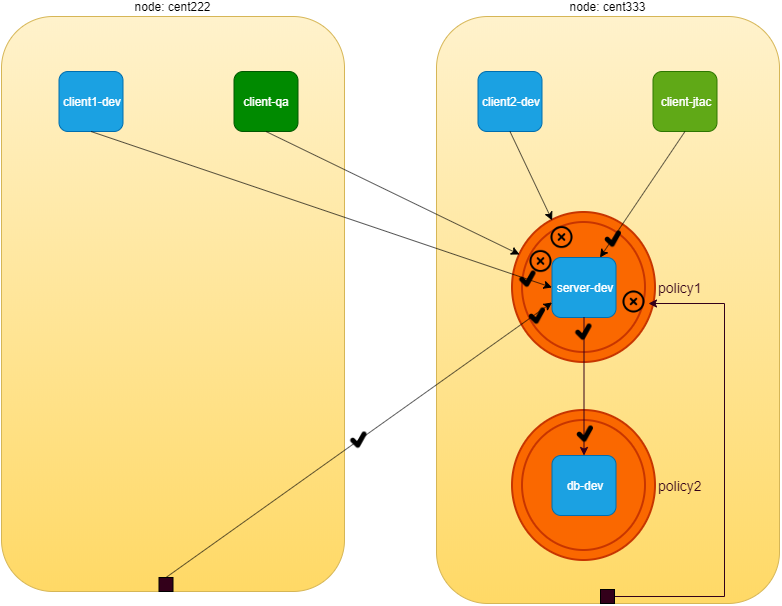
accessing httpserver of webserver-dev pod from pod client1-dev, client-jtac and node cent222 host:

$ kubectl exec -it client1-dev -n dev -- curl http://10.47.255.234  
<html>  
<style>  
 h1 {color:green}  
 h2 {color:red}  
</style>  
 <div align="center">  
 <head>  
 <title>Contrail Pod</title>  
 </head>  
 <body>  
 <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b>  
 pod</h2><br><h3>IP address = 10.47.255.234<br>Hostname = webserver-dev</h3>  
 </body>  
 </div>  
</html>

repeat the same test from from pod client2-dev, client-qa and another node cent333 now get timed out:

$ kubectl exec -it client2-dev -n dev -- curl http://10.47.255.234 -m 10  
curl: (28) Connection timed out after 10000 milliseconds  
command terminated with exit code 28  
  
$ kubectl exec -it client-jtac -n jtac -- curl http://10.47.255.234 -m 10  
curl: (28) Connection timed out after 10000 milliseconds  
command terminated with exit code 28  
  
$ curl http://10.47.255.234:80 -m 10  
curl: (28) Connection timed out after 10000 milliseconds

the test result is illustrated in this figure:



network policy

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.

### egress policy

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 is applied successfully. also we have not apply any policy to the other server pod dbserver-dev. according the default "allow any" policy, therefore any pods can directly access it without a problem. obviously this is not what we’ve designed. another ingress network policy is needed for dbserver-dev pod.

there are two more test items:

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

**test egress traffic.**

#test access to all pods  
kubectl exec -it webserver-dev -n dev -- curl http://10.47.255.229 -m5  
kubectl exec -it webserver-dev -n dev -- curl http://10.47.255.230 -m5  
kubectl exec -it webserver-dev -n dev -- curl http://10.47.255.231 -m5  
kubectl exec -it webserver-dev -n dev -- curl http://10.47.255.232 -m5  
kubectl exec -it webserver-dev -n dev -- curl http://10.47.255.233 -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 (10.47.255.233) succeeds. all other egress access is timed out.

$ kubectl exec -it webserver-dev -n dev -- curl 10.47.255.234  
  
<html>  
<style>  
 h1 {color:green}  
 h2 {color:red}  
</style>  
 <div align="center">  
 <head>  
 <title>Contrail Pod</title>  
 </head>  
 <body>  
 <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b> pod</h2><br><h3>IP address = 10.47.255.234<br>Hostname = webserver-dev</h3>  
 </body>  
 </div>  
</html>  
  
$ 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

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://10.47.255.233 -m5  
kubectl exec -it client1-dev -n dev -- curl http://10.47.255.233 -m5  
kubectl exec -it client2-dev -n dev -- curl http://10.47.255.233 -m5  
kubectl exec -it client-jtac -n dev -- curl http://10.47.255.233 -m5  
kubectl exec -it client-qa -n dev -- curl http://10.47.255.233 -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://10.47.255.233 -m5 | \  
 w3m -T text/html | grep -v "^$"  
 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:

#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 is much simpler - the policyTypes only has Ingress in the list so it will only define a 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 one allowed to initiate the TCP connection toward selected pod dbserver-dev on port 80.

now the access to dbserver-dev pod is secured!

## contrail implementation details

Network policies created in a Kubernetes environment are implemented by using Contrail Security Policy framework. Labels from the Kubernetes environment are exposed as tags in Contrail. you can define tags for a Kubernetes environment. Contrail security policy uses these tags to implement specified Kubernetes policies. You can define tags in the UI or upload configurations in JSON format. The newly-defined tags can be used to create and enforce policies in Contrail Security.

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).

### mappings

Below table would represent network policy construtcs in k8s and mapping constructs in contrail and the mapping is done by the kube-manager.

|  |  |
| --- | --- |
| 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 | Firewall Rule (one per k8s ingress/egress policy) |
| CIDR | Address Group(one per k8s network policy CIDR ) |
| Label | Tag (one for each k8s label) |

contrail-kube-manager, the KM, as we’ve read many time earlier in this book, does all the translations between kubernetes and contrail world.

basically the following will happen for k8s network policy:

1. KM will create a APS with Kubernetes cluster name during the initialization.
2. KM registers to k8s api server to watch network policies events.
3. Whenever namespace is created, KM will associate the APS to the VNs belonging to the namespace.
4. Whenever pod is created, in contrail corresponding tag will be created.
5. Whenever network policy is created, contrail firewall policy will be created with matching firewall rules and network endpoints.

**Note**

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

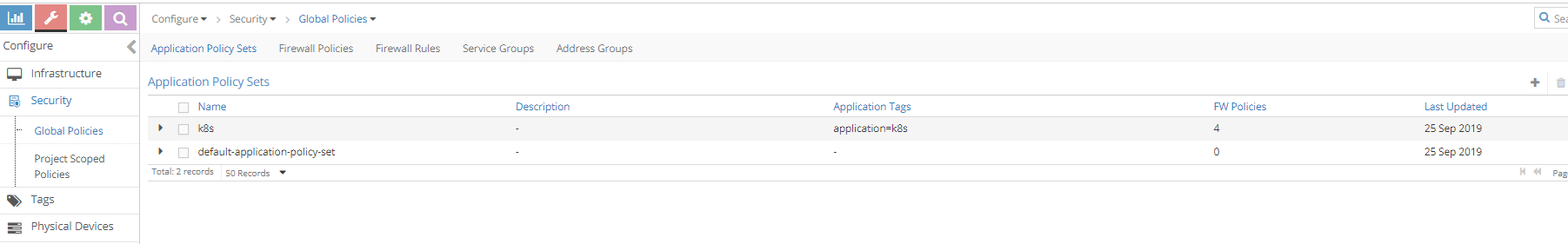
* VMI(virtual-machine-interface)
* VM(virtual-machine)
* virtual-network
* project

In contrail-k8s 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.

Lets focus on the contrail objects which are created for k8s network policy.

### Application Policy Set (APS)

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



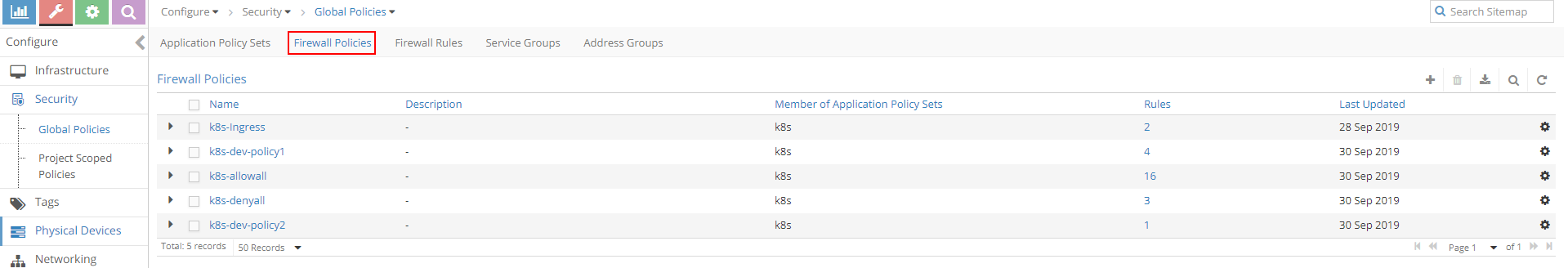
contrail UI: APS: configure → Security → Global Policies → "Application Policy Sets"

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



contrail UI:"Firewall Policies"

#### contrail firewall policy naming convention

the k8s-dev-policy1 and k8s-dev-policy2 policies are what we’ve created, even though they are with different names. it is easy to tell from the name that when KM creates the contrail firewall policies from the kubernetes network policies we created, it prefixes the firewall policy name with cluster name plus the namespace in front of our network policy name:

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

this sounds familiar after you’ve seen earlier how KM name the VN in UI after the VN objects we created in yaml file. the K8s-ingress firewall policy is created for the ingress loadbalancer which is out of the scope of this book, so we can ignore it here.

#### k8s-allowall k8s-denyall

but the bigger question is that why we see 3 more firewall policies here, since we had never created any network policies like Ingress, allowall, or denyall?

remember when we introduce kubernetes network policy in chapter 3, we’ve mentioned the "whitelist" access control method and implicit "allow all" rules. the nature of "whitelist" method indicate "deny all" other than what is in the whitelist, while the implicit "allow all" behavior make sure a pod that is not involved in any network policies 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 behaviors are honored by two additional explicitly generated policies by KM.

#### sequence number

but this may not clear all the puzzles…​ with multiple firewall policies, which one should be evaluated first and which ones afterward? in another word, in what "sequence" shall contrail evaluate each policies? firewall policies evaluation with a different sequence will lead to completely different result. imagine these two sequences "denyall - allowall" vs "allowall- denyall", the former give a pass to all other pods, while the latter give a stop.

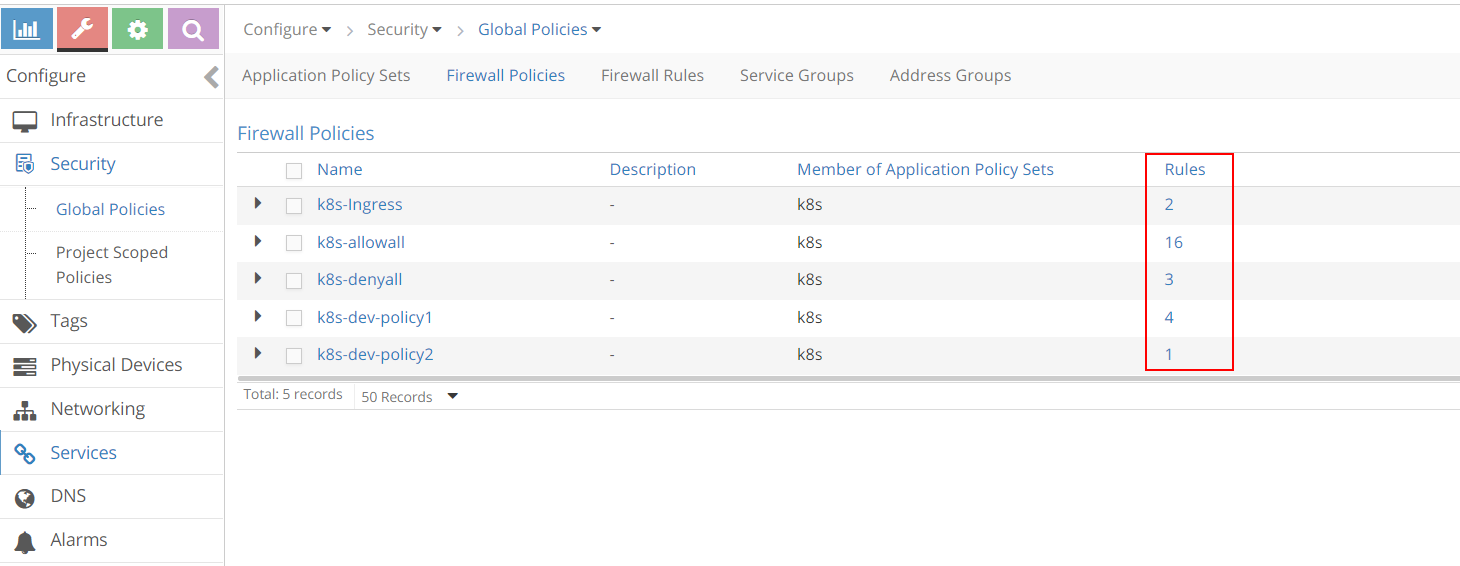
the anwer is the "sequence number".

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 them has a sequence number. When there is a matching policy, it will be executed. it is again contrail-Kube-manager that allocates the right sequence number for firewall policies and firewall rules, so that things will work in correct order. the process is automatically done without manual intervention, so that 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.

### rules

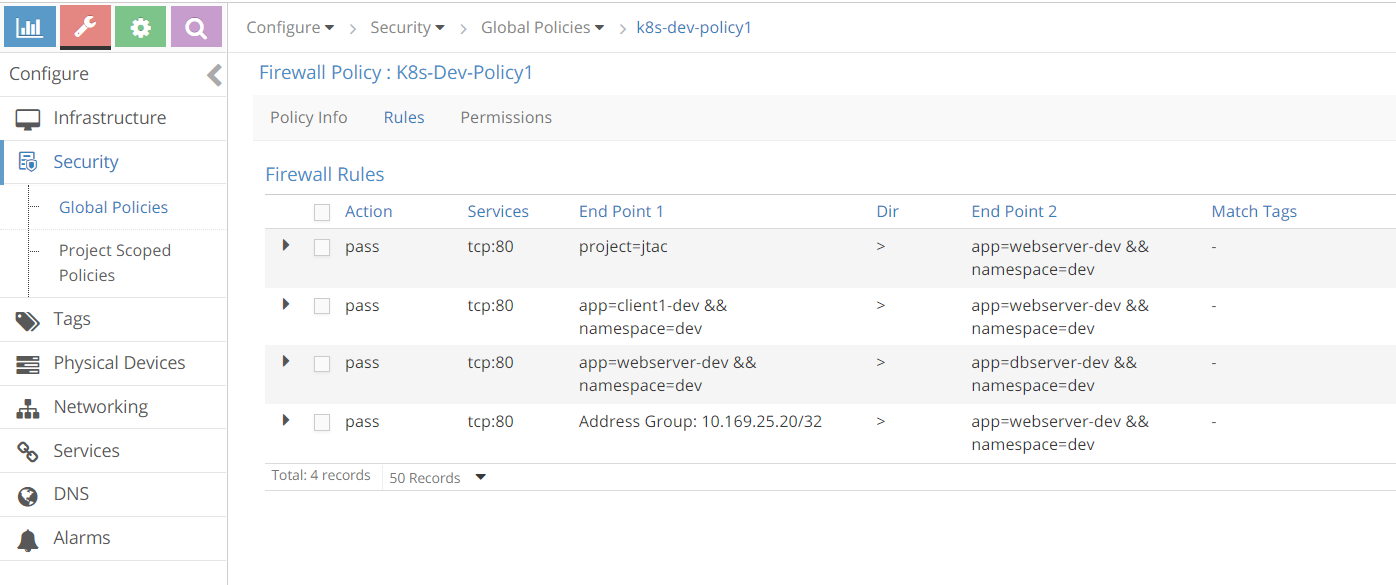
in the same view of "Firewall policies" list, in the right side we see number of "Rules" for each policy:



contrail UI:"Firewall Policy rules"

#### rules in k8s-dev-policy1

there are 4 rules for our 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 the kubernetes network policy policy1 that we’ve defined? 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 | - |

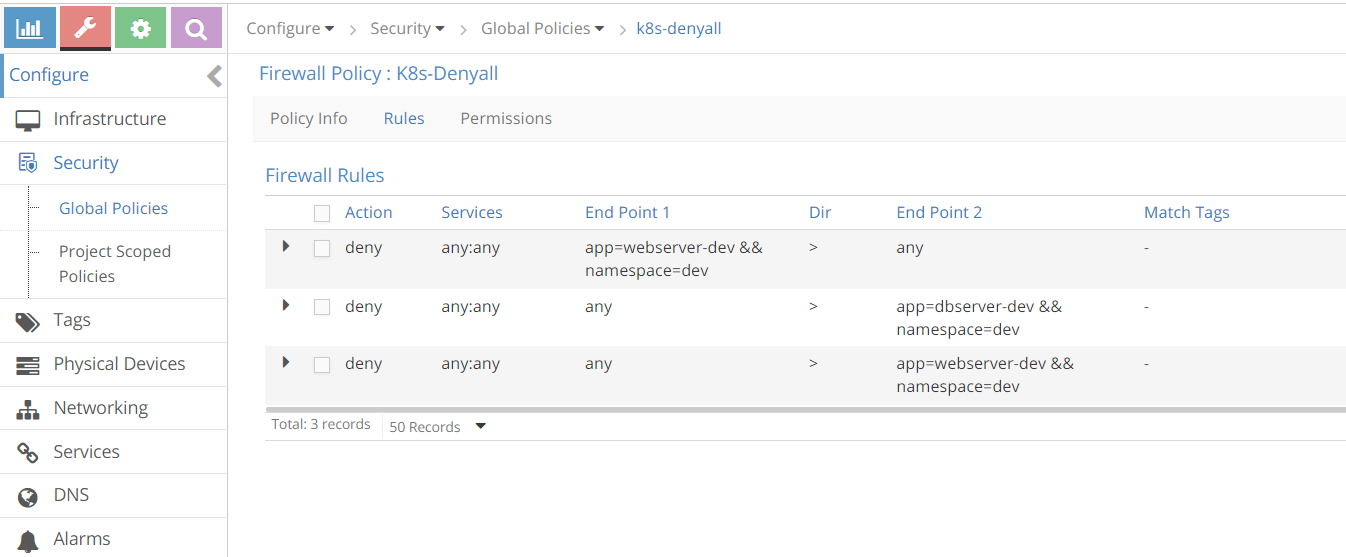
the first column is the rule number that we added, all other columns are imported from the UI screenshot. now compare it with what we’ve seen in kubernetes object in detail:

$ 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 matches between kubernetes network policy policy1 and contrail firewall policy k8s-dev-policy1!

#### rules in k8s-denyall

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



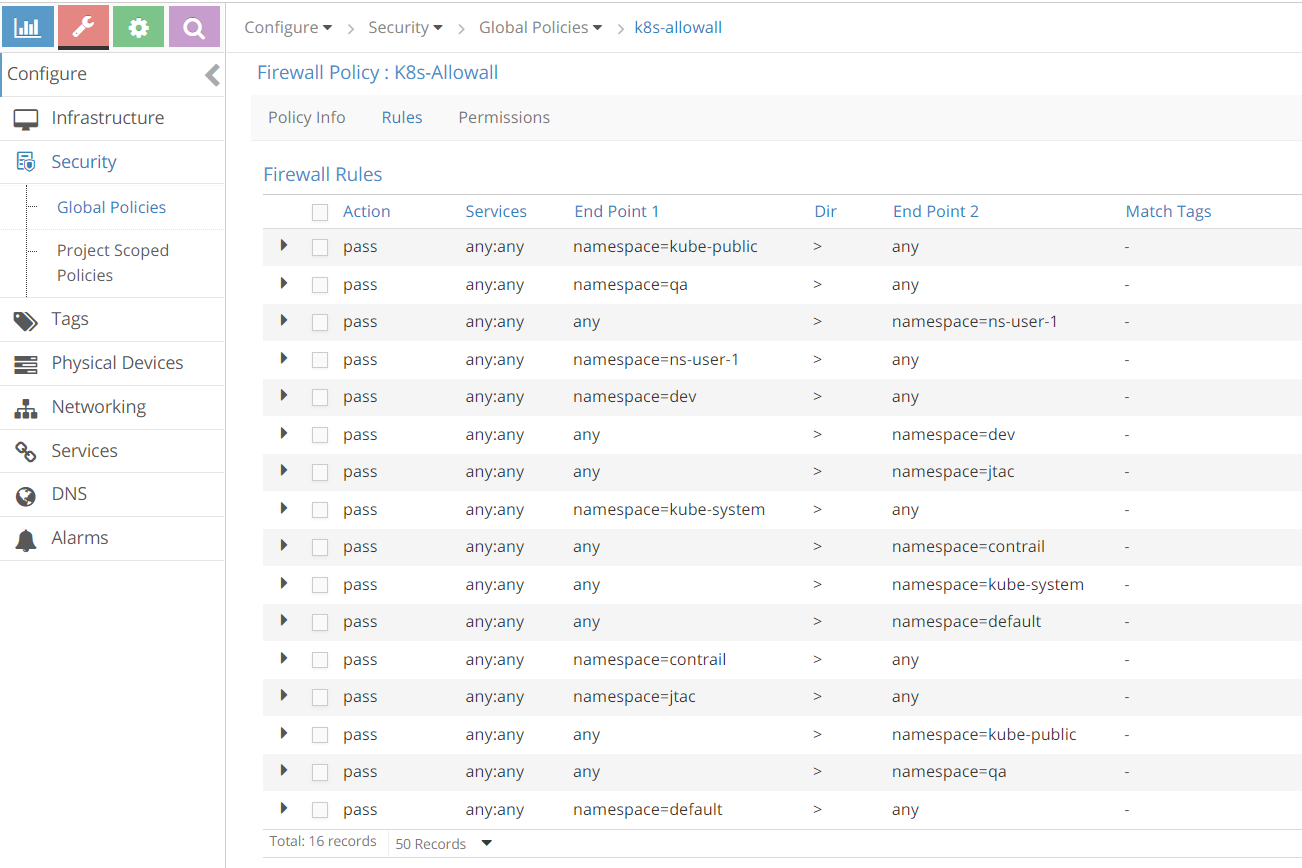
contrail UI:"k8s-denyall" rules

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 other than the ones allowed in all our defined network policies. please pay attention that the "k8s-denyall" policy only apply to those 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, client-qa will not be affected by k8s-denyall policy. instead, those other pods will be applied by k8s-allowany policy, which we will examine the next.

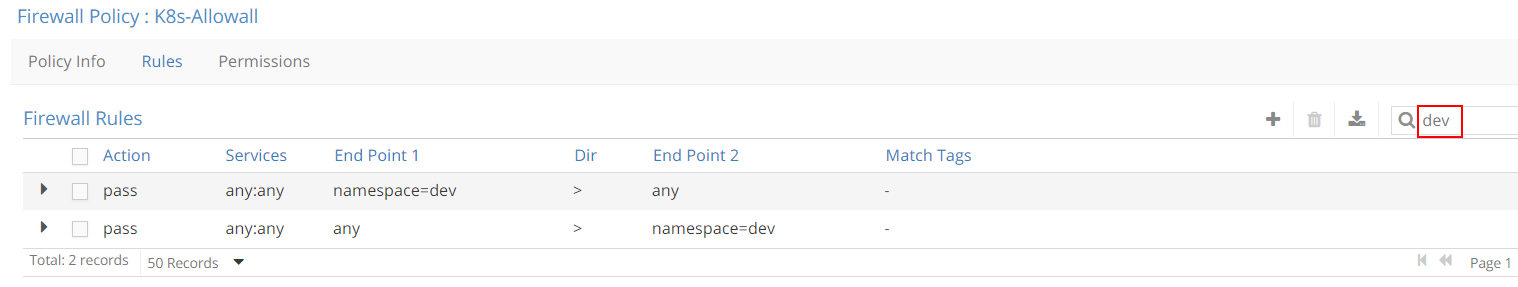
#### rules in k8s-allowall

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

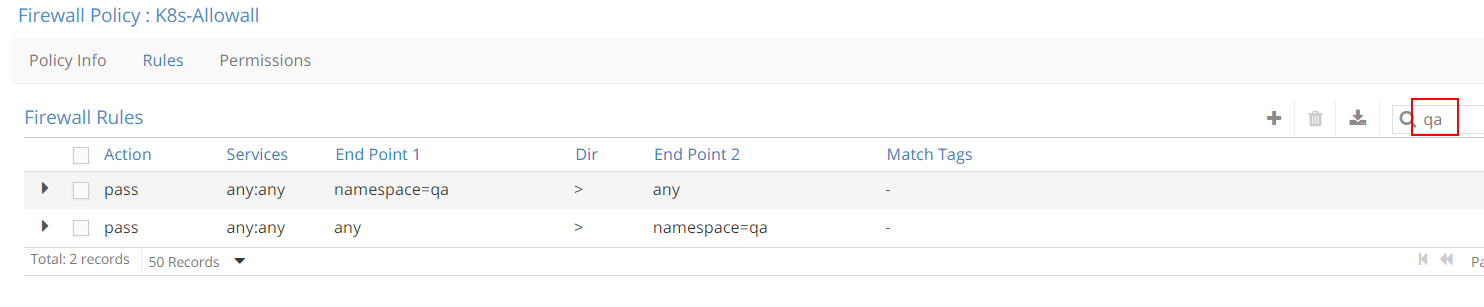


contrail UI:"k8s-allowall" rules

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 US apply a namespace e.g. "jtac" as filter with in the "search" button, we’ll see this figure:



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



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

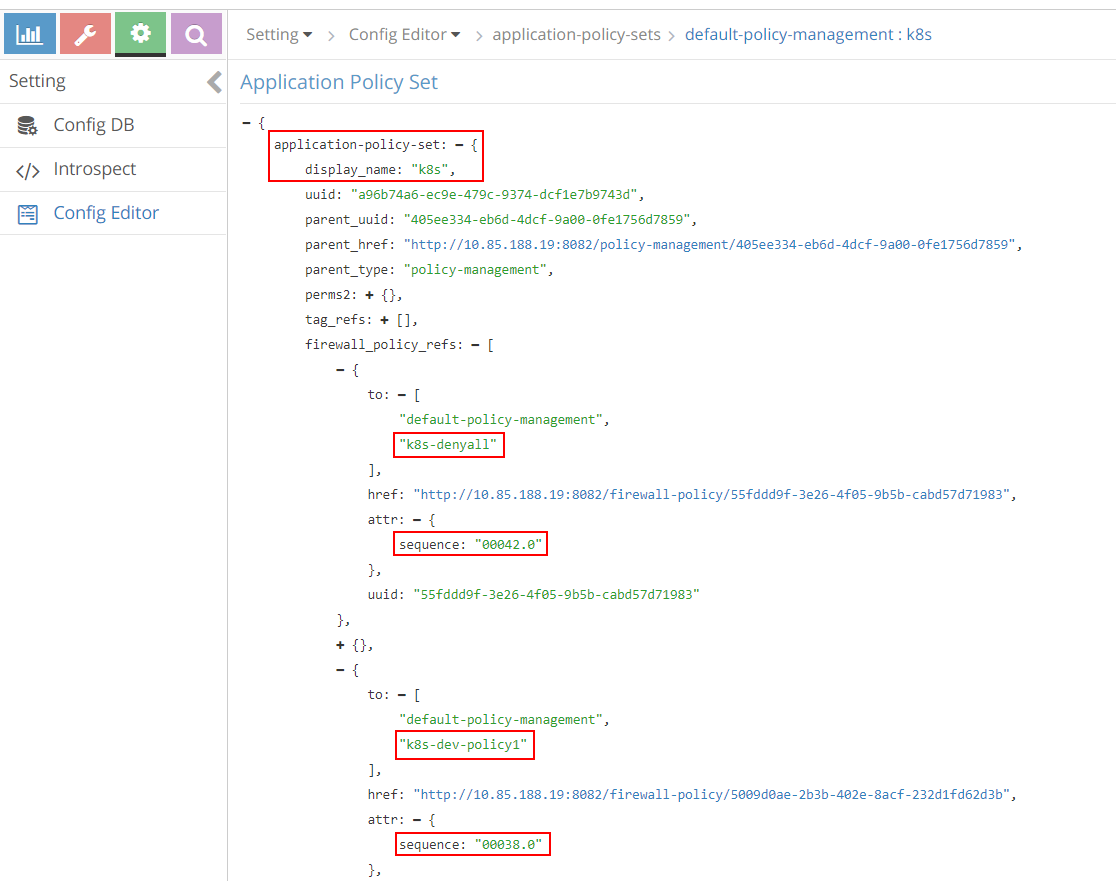
it just says: for those who does not has any network policy applied yet, let’s continue the kubernetes default "allow-any-any" networking model and allow everything!

### sequence number

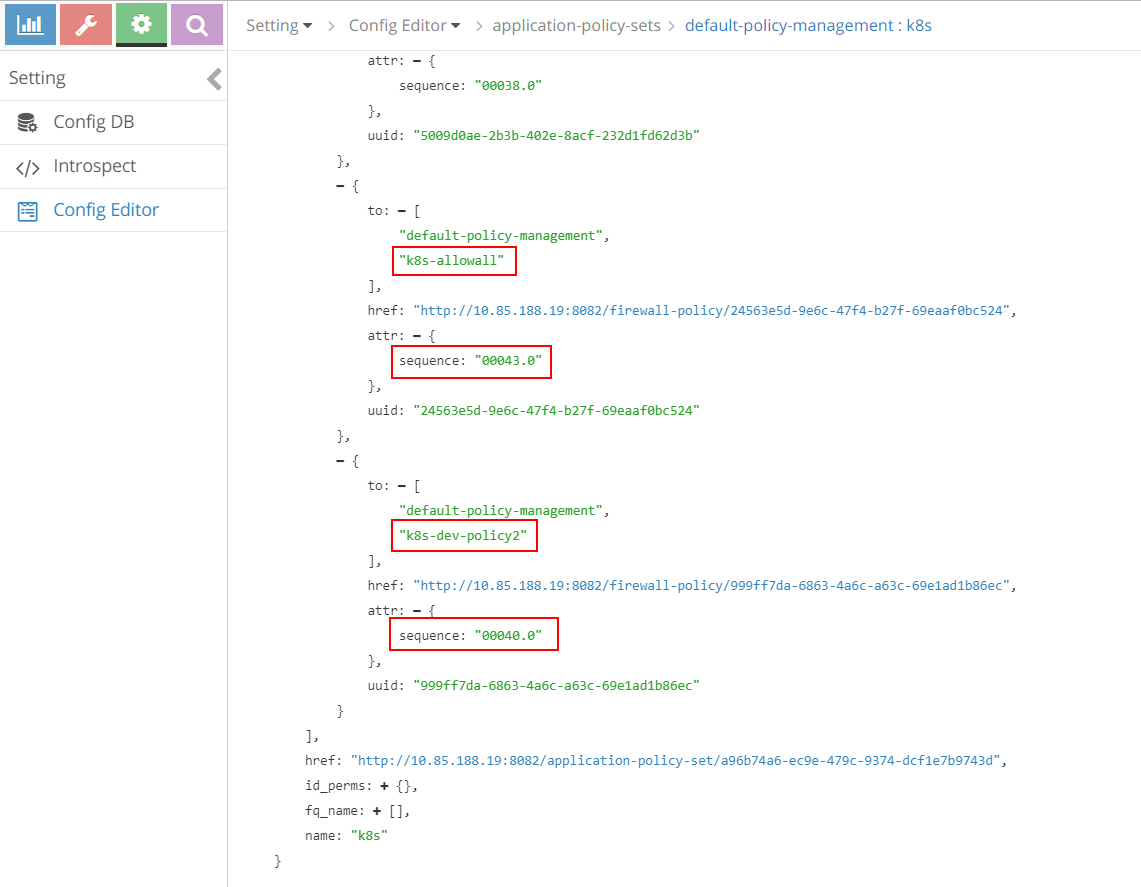
after having explored the contrail firewall policy rules, let’s come back to the sequence number and see how it works exactly.

sequence number is a number attached in all firewall-policies and their rules. it decides the order in which all rules are applied and evaluated. the lower the sequence number the higher the priority. let’s look at our setup.

#### sequence number in firewall policies



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



contrail UI:sequence number for policies: (continue)

in this screenshot, under APS k8s there are the 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 selected pod will be denied. this is the exact goal that we want to achieve.

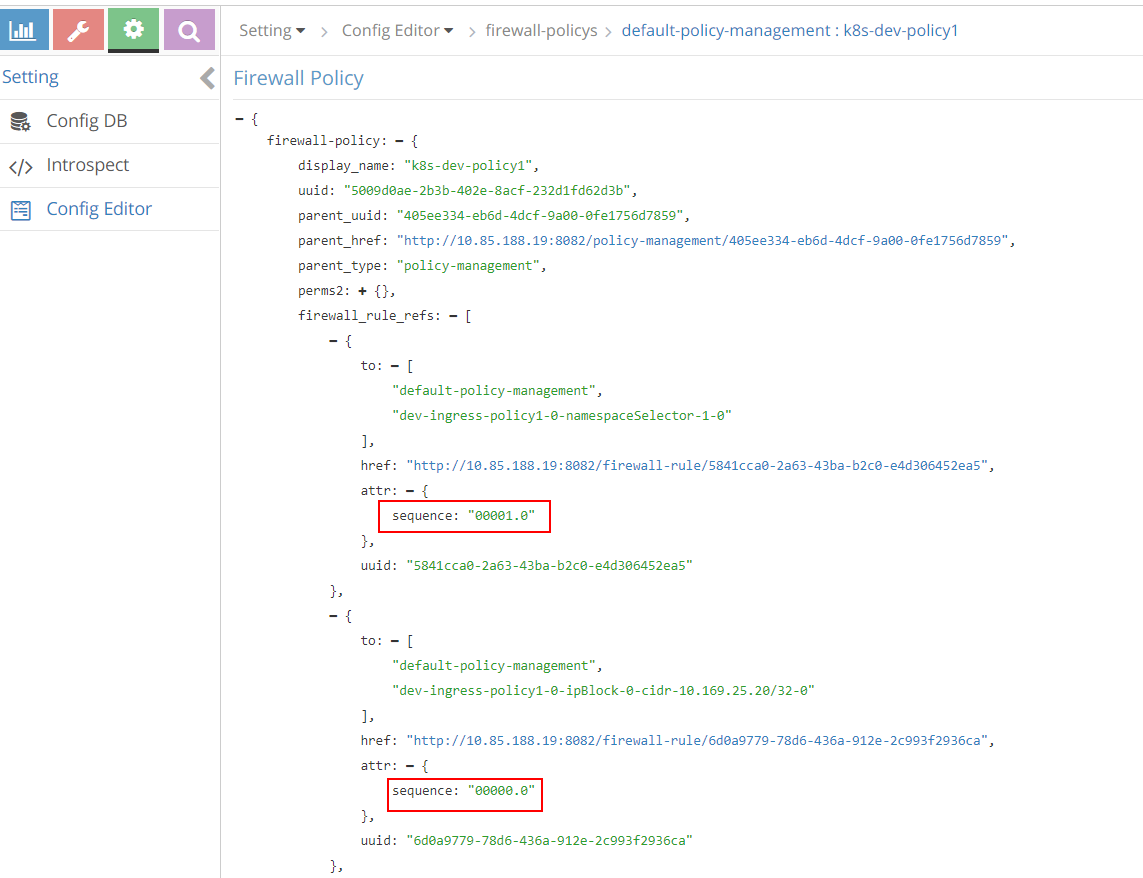
overall, this is the sequence number for all firewall policies, from highest priority to the lowest:

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

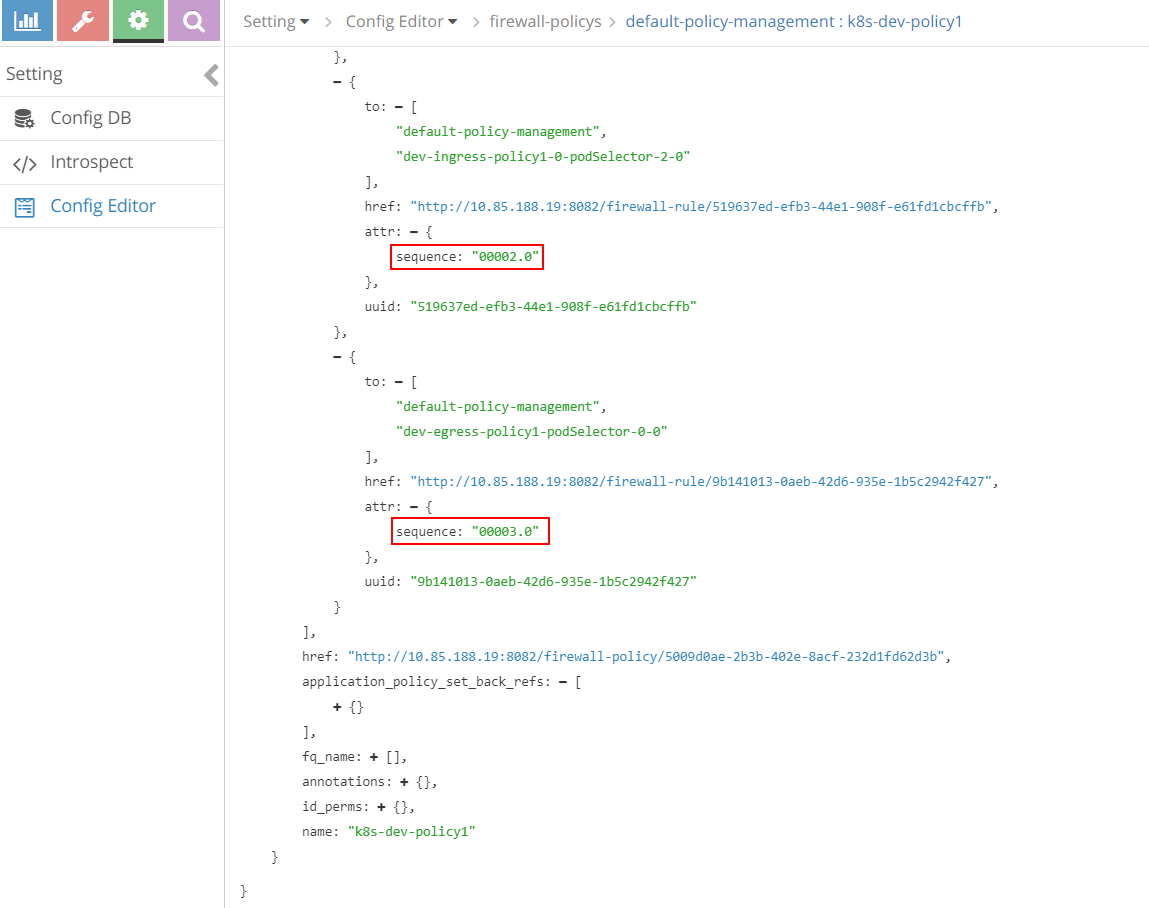
next let’s check the sequence number in policy rules.

#### sequence number in firewall policy rules

in the same firewall policy, policy rules will also have to be applied and evaluated in a certain order. that is also 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)

this is the sequence number for all policy 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 show in the definition file. in another word, rules will be applied and evaluated in the same order as they are defined.

### test case: drop action

Since action for the whitelisted case is FORWARD and flow for FORWARD is explained in the previous sections with service and ingress, lets focus on the flow for the non-whitelisted case.

$ kubectl exec -it client-jtac -n jtac -- curl -v 10.47.255.234  
\* Rebuilt URL to: 10.47.255.234/  
\* Hostname was NOT found in DNS cache  
\* Trying 10.47.255.234...  
\* connect to 10.47.255.234 port 80 failed: Connection timed out  
\* Failed to connect to 10.47.255.234 port 80: Connection timed out  
\* Closing connection 0  
curl: (7) Failed to connect to 10.47.255.234 port 80: Connection timed out  
command terminated with exit code 7

login to contrail-vrouter-agent and see the flow

$ docker exec -it vrouter\_vrouter-agent\_1 flow --match 10.47.255.234:80  
Flow table(size 80609280, entries 629760)  
  
Entries: Created 340 Added 336 Deleted 280 Changed 331Processed 340 Used Overflow entries 0  
(Created Flows/CPU: 91 84 65 100)(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 ([10.47.255.234]:80)  
  
 Index Source:Port/Destination:Port Proto(V)  
 ----------------------------------------------------------------------------------  
 11148<=>242676 10.47.255.234:80 6 (4)  
 10.47.255.230:35778  
(Gen: 1, K(nh):52, Action:D(Unknown), Flags:, TCP:Sr, QOS:-1, S(nh):52,  
 Stats:0/0, SPort 62120, TTL 0, Sinfo 0.0.0.0)  
  
 242676<=>11148 10.47.255.230:35778 6 (4)  
 10.47.255.234:80  
(Gen: 1, K(nh):65, Action:D(FwPolicy), Flags:, TCP:S, QOS:-1, S(nh):65,  
 Stats:6/444, SPort 61489, TTL 0, Sinfo 12.0.0.0)

Since client-qa is not part of the white-list for the webserver-dev pod, the action is set to D(FwPolicy) which means DROP due to Firewall Policy.