Chapter 4

2019-09-26

# contrail Network Policy (ch3)

if you remember in chapter 4, we’ve introduced the "flat network" requirement in kubernetes network. with a flat network basically pods can reach any other pods by default. Then how pods can be secured? The answer is network-policy. Network policy is a Kubernetes resource just like pod, service, ingress, etc. the role of this object is to define how groups of pods are allowed to communicate with each other. here are some rules about kubernetes network policy:

1. Pods are non-isolated by default - they can communicate to each other
2. Once the network policy is applied to a pod, pod will reject any connections that are not allowed by the network policy, i.e. explicit "white list" rules has to be added to allow the traffic.
3. Pods which are not part of the network policy will continue to communicate to all pods
4. Multiple network-polices can be applied to a pod.

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.

like all other objects in kubernetes, network policy can be defined in a yaml file. here is an example:

#policy-do.yaml  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: network-policy-do  
 namespace: dev  
spec:  
 podSelector:  
 matchLabels:  
 app: server-dev  
 policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 10.169.25.0/24  
 except:  
 - 10.169.25.20/32  
 - namespaceSelector:  
 matchLabels:  
 project: qa  
 - podSelector:  
 matchLabels:  
 app: client1-dev  
 ports:  
 - protocol: TCP  
 port: 8000  
 egress:  
 - to:  
 - ipBlock:  
 cidr: 10.169.25.21/32  
 ports:  
 - protocol: TCP  
 port: 8100

in this yaml file, Network policy definition can logically be divided into two sections:

1. pods selection: identify pods where the Network policy would be applied
2. ingress/egress policies: define the policies for the selected pods on ingress/egress directions

## Pods selection

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

In the above example the network policy would be applied to all pods which has the label "app: server-dev".

## Ingress and egress policies

The second section defines the policy types for the selected pods. Policy type can be "ingress", "egress" or both. "ingress" is the default policy type. policy identifies the network endpoint where the selected pods can communicate. Network endpoint can be ip address block or pods in a namespace. (all pods, group of pods, selected pods, etc)

Ingress network-endpoint has to be defined in the from section. Egress network endpoint has to be defined in the to section.

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

In the above example:

1. The ingress network points are
   1. subnet 10.85.188.0/24 except 10.85.188.21/32, or
   2. All the pods in namespaces which has the label project: qa , or
   3. Pods which has the label app: client1-dev
2. The egress network point is
   1. 10.85.188.20/32

It is also possible to select only a few pods from namespaces instead of all pods. podSelector can also be specified under namespaceSelector. in that case, network endpoint would be pods with matching labels in the selected namespaces.

The below yaml file shows an example to only allow connections from pods with label app: client-qa in namespaces with the label project: qa. Please be aware to use correct yaml syntax.

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

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

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

here is an example of policyTypes with ports specified:

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  
 ports:  
 - protocol: TCP  
 port: 8000  
 egress:  
 - to:  
 - ipBlock:  
 cidr: 10.169.25.20/32  
 ports:  
 - protocol: TCP  
 port: 8100

The above network policy says that all ingress network endpoint can reach selected pods at tcp port 8000, selected pods can reach all egress network endpoint’s tcp port 8100. all other traffic would be blocked.

to create the network policy:

$kubectl create -f policy-do.yaml -n dev  
  
$ kubectl get netpol -n dev  
NAME POD-SELECTOR AGE  
network-policy-do app=server-dev 6s  
  
$ kubectl describe netpol -n dev  
Name: network-policy-do  
Namespace: dev  
Created on: 2019-09-23 04:39:01 -0700 PDT  
Labels: <none>  
Annotations: <none>  
Spec:  
 PodSelector: app=server-dev  
 Allowing ingress traffic:  
 To Port: 8000/TCP  
 From:  
 IPBlock:  
 CIDR: 10.169.25.0/24  
 Except: 10.169.25.21/32  
 From:  
 NamespaceSelector: project=qa  
 From:  
 PodSelector: app=client1-dev  
 Allowing egress traffic:  
 To Port: 8100/TCP  
 To:  
 IPBlock:  
 CIDR: 10.169.25.20/32  
 Except:  
 Policy Types: Ingress, Egress

# Contrail – k8s Network Policy (ch8)

In chapter 3 we introduced kubernetes network policy. we’ve mentioned that simply creating network policy object won’t have any effect unless the kubernetes networking implementation support it. In this chapter we are going to see contrail’s solution for kubernetes network policy with examples.

**Contrail Firewall.**

Contrail has various security features to define policies between workloads. e.g.: "security groups", "network policy" and "firewall". Each has its own capabilities and a different design goal. in this chapter we’ll focus on "contrail firewall" only.

Contrail Firewall, often referred as "Contrail Security", is one of the feature that identifies the workloads based on the tags(labels). as you can see, this is similar to k8s network policy, so it is natural for contrail to use its firewall to implement k8s network policy.

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.

**Contrail k8s Network Policy test.**

Before getting into deeper into contrail firewall, lets test and see how does the k8s network policy works in contrail.

in our test we need:

* 3 namespaces,
* 5 pods
  + 1 server pod,
  + 2 client pods in the same namespace,
  + 2 clients from two different namespaces)
* 2 CIDRs to show allow and deny traffic within, across namespaces and CIDRs

Lets create the required k8s NS and pods resources.

the dev, qa and build namespaces all-in-one yaml file:

#policy-ns-do.yaml  
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: build  
 labels:  
 project: build  
---

create the 3 namespaces:

$kubectl create -f network-policy-ns-do.yaml  
namespace/dev created  
namespace/qa created  
namespace/build created

create following pods in the three NS:

|  |  |
| --- | --- |
| NS | pod |
| dev | server-dev |
| dev | client1-dev |
| dev | client2-dev |
| qa | client-qa |
| build | client-build |

here is the all-in-one yaml file:

# policy-pod-do.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
 name: server-dev  
 labels:  
 app: server-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/ubuntu  
---  
apiVersion: v1  
kind: Pod  
metadata:  
 name: client2-dev  
 labels:  
 app: client2-dev  
 do: policy  
 namespace: dev  
spec:  
 containers:  
 - name: ubuntu  
 image: contrailk8sdayone/ubuntu  
---  
apiVersion: v1  
kind: Pod  
metadata:  
 name: client-qa  
 labels:  
 app: client-qa  
 do: policy  
 namespace: qa  
spec:  
 containers:  
 - name: ubuntu  
 image: contrailk8sdayone/ubuntu  
---  
apiVersion: v1  
kind: Pod  
metadata:  
 name: client-build  
 labels:  
 app: client-build  
 do: policy  
 namespace: build  
spec:  
 containers:  
 - name: ubuntu  
 image: contrailk8sdayone/ubuntu  
---

**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 and verify all pods:

$kubectl create -f policy-pod-do.yaml  
pod/server-dev created  
pod/client1-dev created  
pod/client2-dev created  
pod/client-qa created  
pod/client-build created  
  
$ kubectl get pod -l 'do=policy' -o wide --all-namespaces  
NAMESPACE NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
build client-build 1/1 Running 0 42s 10.47.255.230 cent333 <none>  
dev client1-dev 1/1 Running 0 42s 10.47.255.233 cent333 <none>  
dev client2-dev 1/1 Running 0 42s 10.47.255.232 cent333 <none>  
dev server-dev 1/1 Running 0 42s 10.47.255.234 cent333 <none>  
qa client-qa 1/1 Running 0 42s 10.47.255.231 cent222 <none>

Before creating the k8s network policy, lets try to access the server pod from all the clients (client1-dev, client2-dev, client-qa and client-build) pods and from host of the two nodes.

check the httpserver is running at port 80 in server-dev pod:

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

accessing httpserver of server-dev 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 = server-dev</h3>  
 </body>  
 </div>  
</html>

we can repeat the same command, except replace the the client pod’s name and namespaces:

$ kubectl exec -it client2-dev -n dev -- curl http://10.47.255.234  
$ kubectl exec -it client-qa -n qa -- curl http://10.47.255.234  
$ kubectl exec -it client-build -n build -- curl http://10.47.255.234

same curl command can be sent from the host of two nodes in the cluster:

$ curl http://10.47.255.234  
$ curl http://10.47.255.234

in all tests, we’ll see exactly the same test result .

now lets create the k8s network-policy and see what is happening and any differences in test result.

here is the same exact policy yaml file we’ve showed in chapter 3:

# policy-do.yaml  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: network-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  
 ports:  
 - protocol: TCP  
 port: 8080

from the network-policy definition, we can see what the policy is trying to enforce:

* the following clients can reach the server-dev server pod located in dev namespace
  + client1-dev from dev namespace
  + client-qa from qa namespace
  + any pod in node 10.169.25.20 (cent222 in our setup)
* all other accesses are denied.

let’s create the policy and verify its effect.

$ kubectl apply -f policy-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=server-dev 17s

accessing httpserver of server-dev pod from pod client1-dev, client-qa 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 = server-dev</h3>  
 </body>  
 </div>  
</html>

repeat the same test from from pod client2-dev, client-build 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-build -n build -- curl http://10.47.255.234:80 -m 30  
curl: (28) Connection timed out after 10001 milliseconds  
command terminated with exit code 28  
  
$ curl http://10.47.255.234:80 -m 30  
curl: (28) Connection timed out after 10001 milliseconds

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

## Deeper view

Contrail Firewall is designed with a hierarchical structure:

* the top level object is named “Application Policy Set(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).

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 will do the following things for k8s network policy:

1. It will create a APS with Kubernetes cluster name during the initialization.
2. It registers to k8s api server to watch network policies CRUD events.
3. Whenever namespace is created, it will associate the k8s APS to the virtual-networks belonging to the namespace.
4. Whenever pod or namespace is created, tag will be created for the k8s labels in contrail.
5. Whenever network policy is created, firewall policy will be created with matching firewall rules and network endpoints.

**Note**

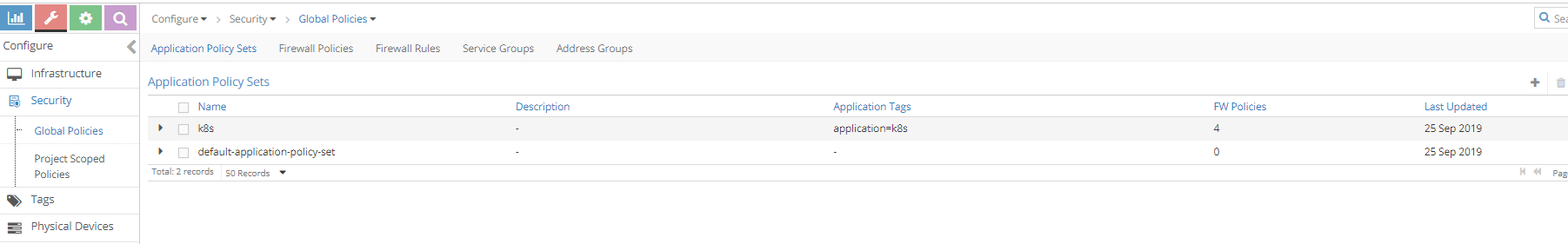
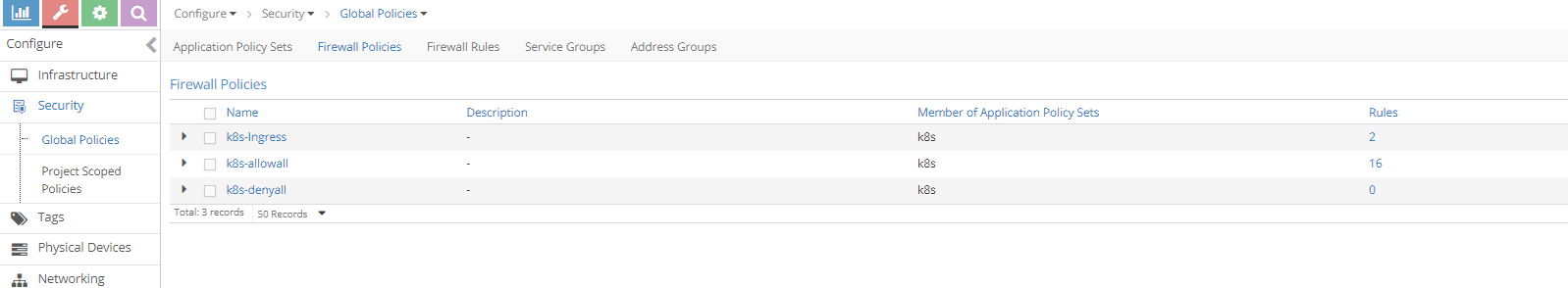
APS can be associated to different contrail objects, e.g.

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

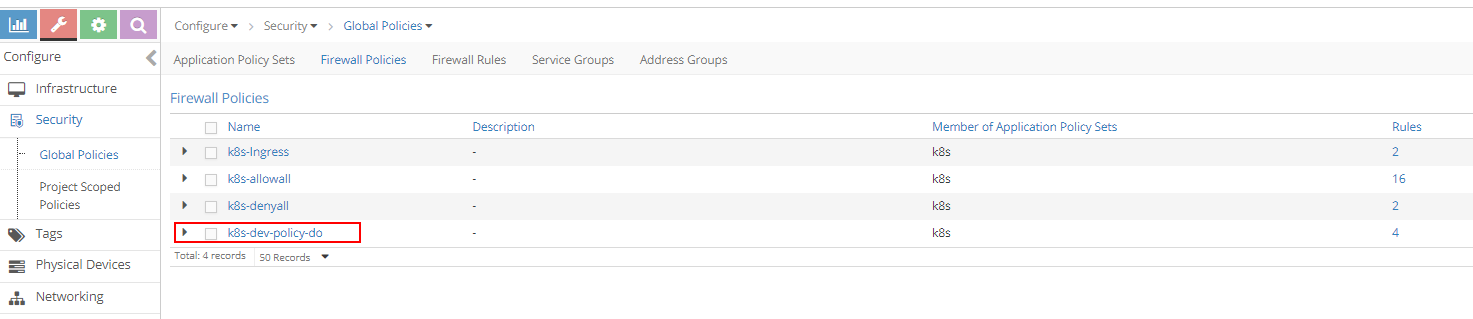
In contrail-k8s cluster, it is associated to virtual network. Whenever traffic goes on those networks, firewall policies associated on the APS would be evaluated and respective action would be taken for the traffic.

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

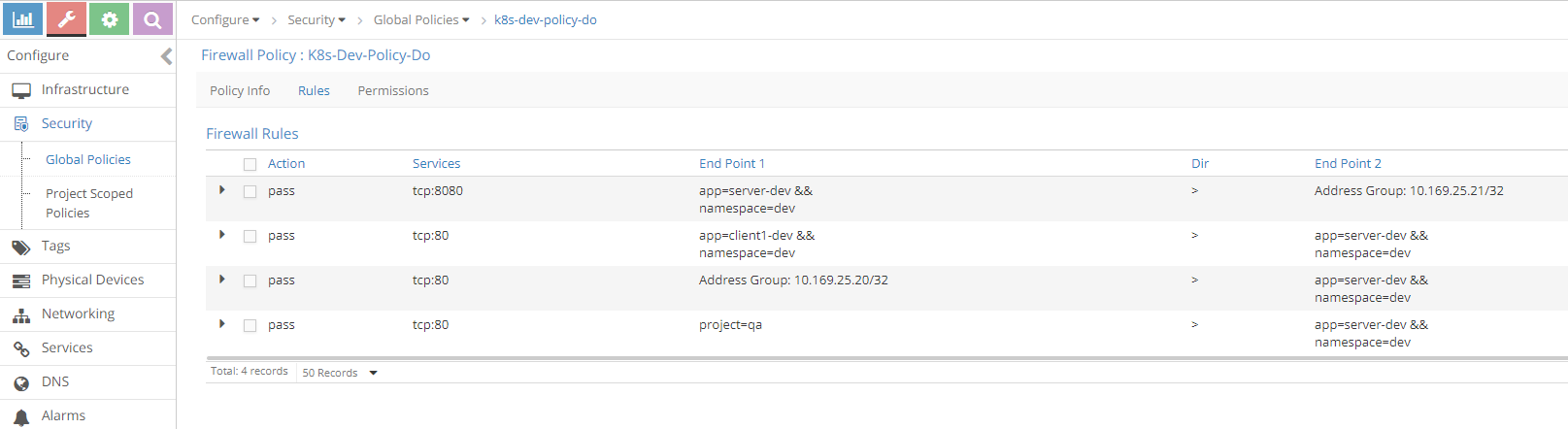
1. As mentioned in the above section, APS with k8s cluster name will be created by contrail-kube-manager during the initialization. Since the cluster name is k8s by default, we can see the APS k8s in the below contrail ui. There is one more APS default-application-policy-set which is created by default. By default firewall policy k8s-denyall and k8s-allowall will be created. Since k8s by default "allows all" and contrail firewall by default "denies all", contrail has to have rules to match k8s default bahaviour. So by default firewalls rules are added in k8s-allowall firewall policy. Below contrail ui snapshots shows it.

* 
* 
* **Note**
* When firewall polices are evaluated for the APS, it is evaluated in a sequence. When there is a matching policy, it will be executed. So all firewall polices and allfirewall rules has a sequence number. contrail-Kube-manager allocates the right sequence number for firewall policies and firewall rules to keep the k8s network-policy intact in contrail. K8s-ingress firewall policy is created for the ingress loadbalancer which is out of the scope of this book.

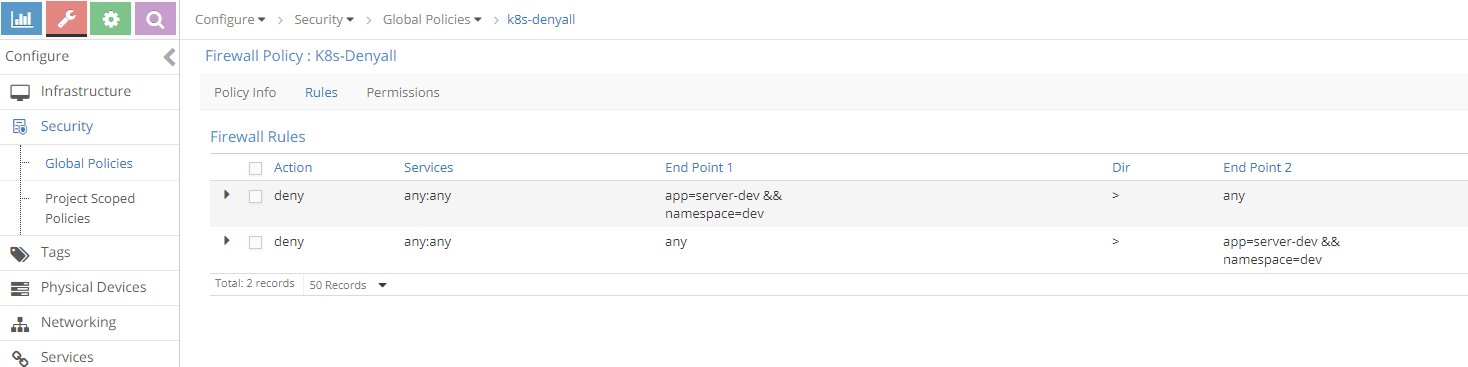
1. After the network policy creation, k8s APS will have the new firewall policy for the k8s netwok policy.

* 

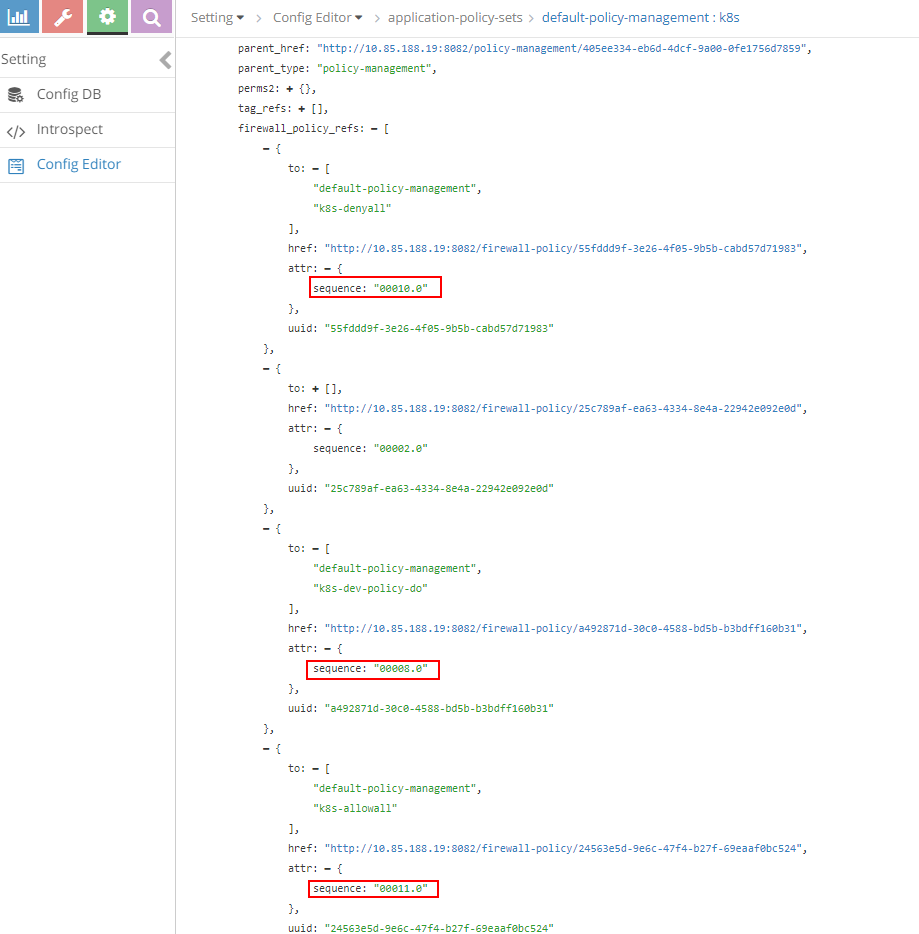
1. K8s-dev-policy-do is the new firewall policy created by contrail-kube-manager for k8s network policy policy-do. As explained in earlier chapters, cluster and namespace name are prepended to the network policy name. If we closely watch, we can see four firewall rules in K8s-dev-policy-do firewall policy and two new firewall rules for the k8s-denyall firewall policy are added. clicking at the firewall policy k8s-dev-policy-do, Firewall rules of k8s-dev-policy-do is matching to k8s network policy which can be seen:

* 

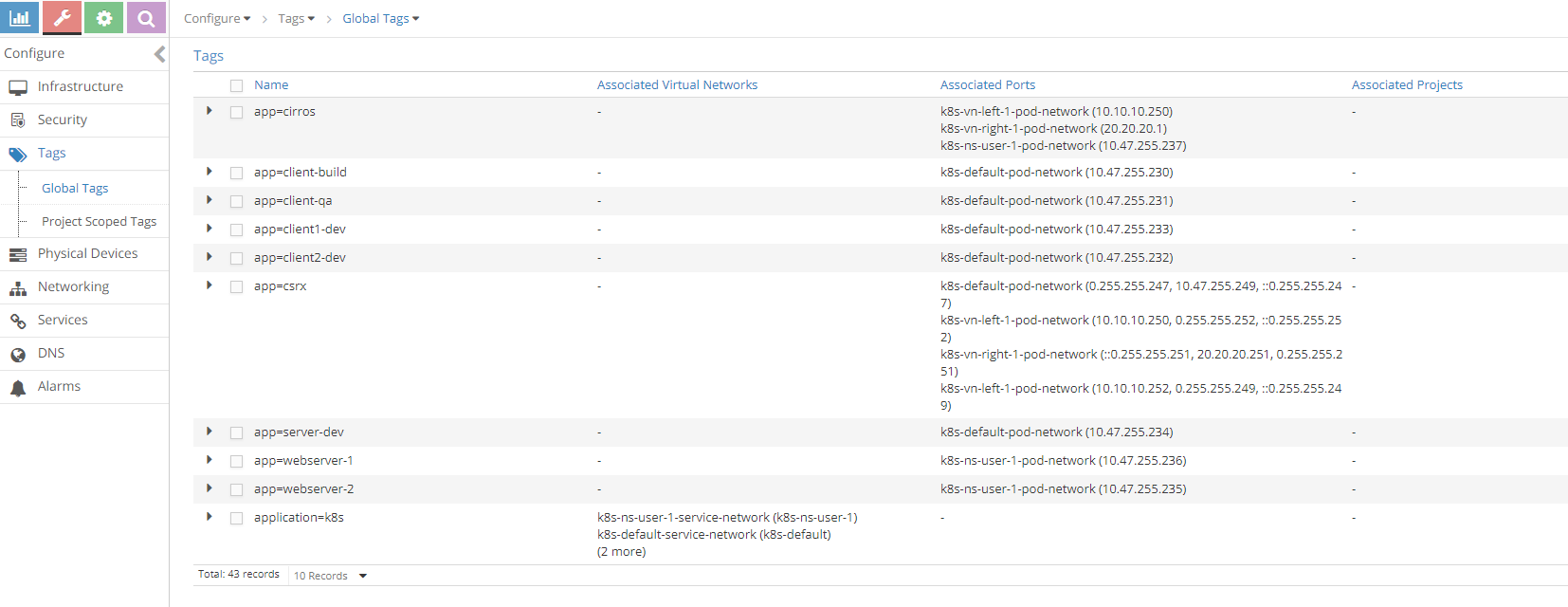
1. Newly added firewall rules for firewall policy k8s-denyall can also be seen below. These rules are needed to deny other traffic to the server-dev pod.

* 

1. Lets see the sequence number of the firewall policies. Since firewall policy k8s-dev-policy-do has high priority (indicated by a lower sequence number) than k8s-denyall and k8s-allowall, all matching traffic for the endpoints are allowed. similarly, k8s-denyall has high priority than k8s-allowall, so all other traffic will be dropped.

* 

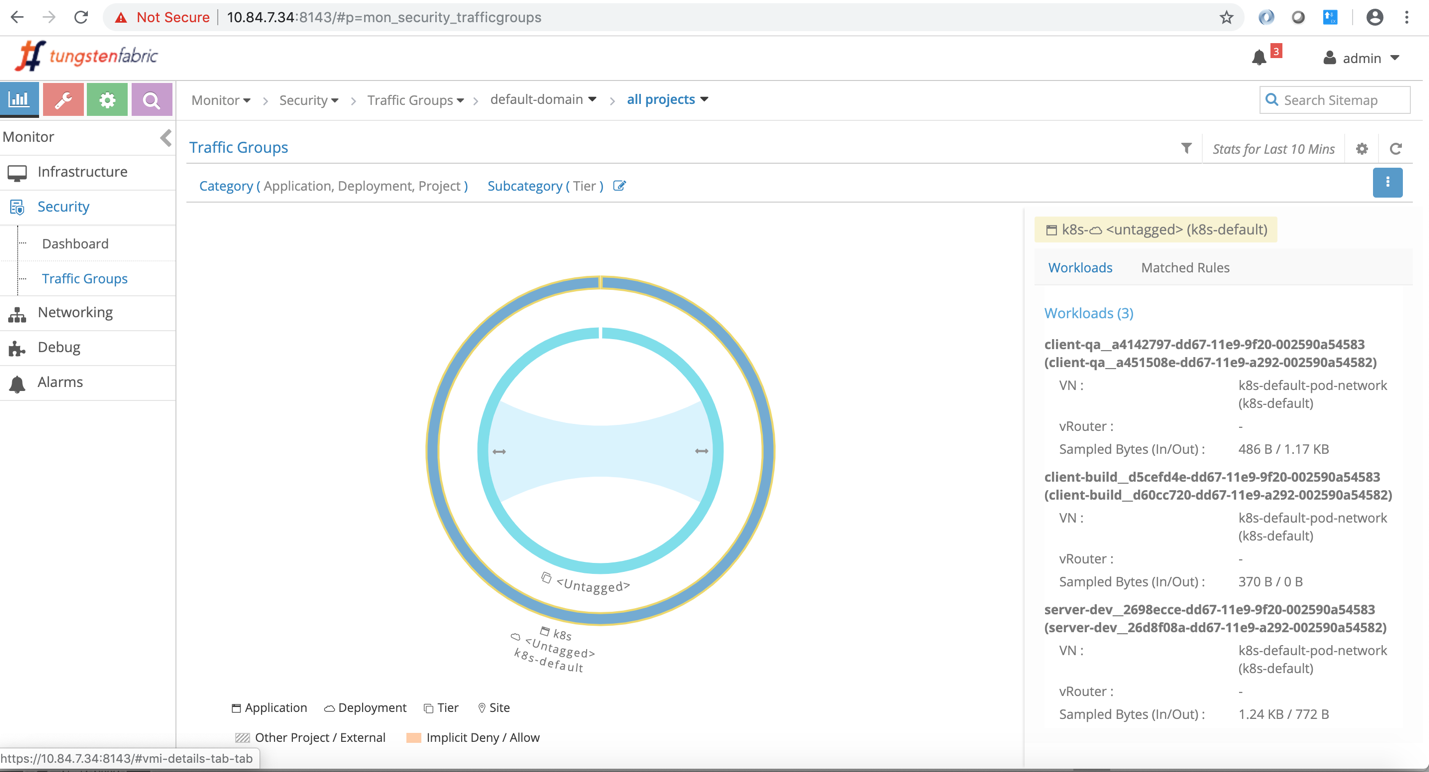
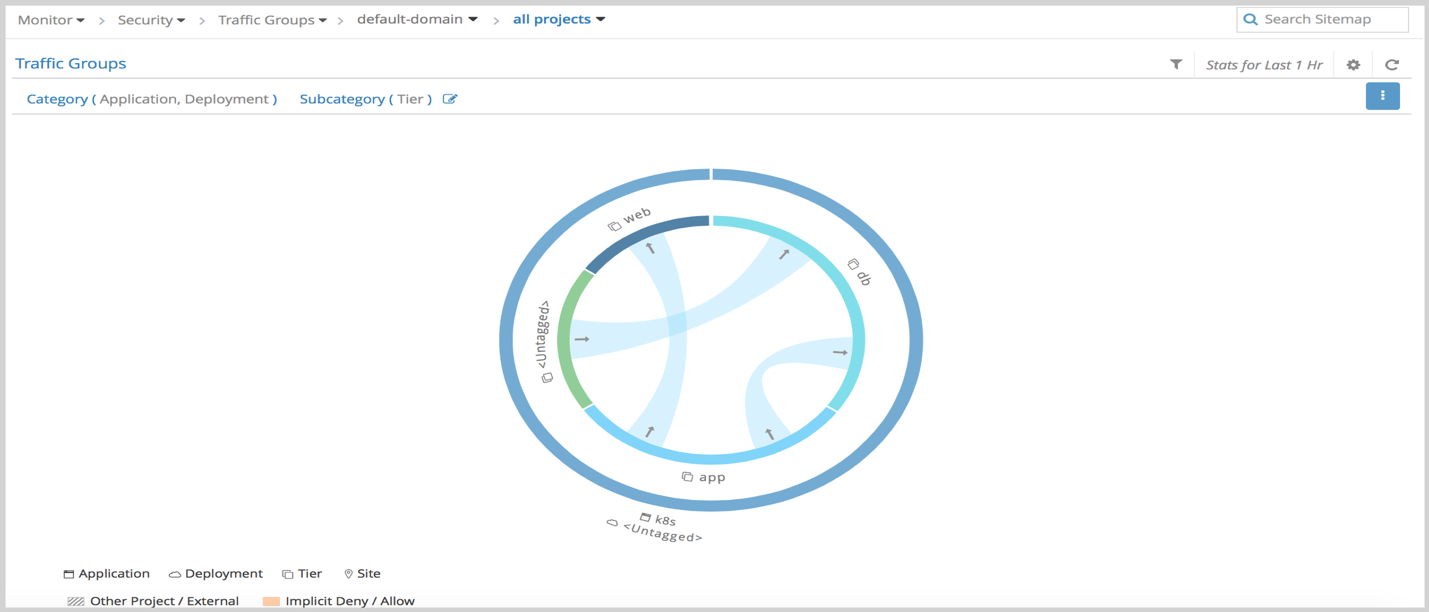
1. Also look at the tags and associated ports(vmi). For each pod label, tag can be seen and it is attached to the respective pod port which is done by the kube-manager.

* 

1. 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-build -n build -- 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-build is not part of the white-list for the server-dev pod, the action is set to D(FwPolicy) which means DROP due to Firewall Policy.

1. So far we have discussed the ingress policy type. There is no change for egress policy type. So whatever we discussed it can be applied to egress policy type.
2. 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