Guide to
Implementing Network Security for Kubernetes

September 2018

Contents

About this Book

Fast becoming the standard for deploying containerized workloads, Kubernetes also brings new requirements for networking and security teams. This is because applications, that are designed to run on Kubernetes, are usually architected as microservices that rely on the network. They make API calls to each other, rather than the class and function calls used in monolithic applications. There are many benefits to this modern architecture, but steps must be taken to ensure proper security protocols are in place.

This book is intended to provide guidance for implementing network security for a Kubernetes platform. Most of the examples use open source software. Some more advanced use cases, that make use of commercial software, are also included. Those use cases are clearly called out.

Introduction to Kubernetes Networking

Kubernetes has taken a different approach to networking that can be categorized into four areas.

1. Container Groups

   For logically grouped containers, the communication should be highly coupled. This is achieved using an abstraction called a Pod that contains one or more containers.

2. Communication between Pods

   Pods are the smallest unit of deployment in Kubernetes. A Pod can be scheduled on one of the many nodes in a cluster and has a unique IP address. Kubernetes places certain requirements on communication between Pods when the network has not been intentionally segmented. These requirements include:

   a. Containers should be able to communicate with other Pods without using network address translation (NAT).

   b. All the nodes in the cluster should be able to communicate with all the containers in the cluster.

   c. The IP address assigned to a container should be the same that is visible to other entities communicating with the container.

3. Pods and Services

   Since Pods are ephemeral in nature, an abstraction called a **Service** provides a long-lived virtual IP address that is tied to the service locator (e.g., a DNS name). Traffic destined for that service VIP is then redirected to one of the Pods and offers the service using that specific Pod's IP address as the destination.

4. Traffic Direction

   Traffic is directed to Pods and services in the cluster via multiple mechanisms. The most common is via an **ingress controller**, which exposes one or more service VIPs to the external network. Other mechanisms include **nodePorts** and even publicly-addressed Pods.

KUBERNETES CONCEPTS

POD

The smallest unit of deployment in Kubernetes is a Pod. A Pod encapsulates a container or a set of containers. Containers within a Pod will often share resources, such as the network stack (including IP address), storage resources, resource limits, etc. Containers within a Pod can communicate with each other via localhost, since they share the same network namespace.

A single application or microservice instance is expected to run within a Pod. In order to horizontally scale the application, multiple replicas of a Pod are created. This is typically handled by another resource in the Kubernetes object model, such as a replication controller, statefulset, daemonset, or a deployment.

CONTROLLERS

Kubernetes stores the desired state of its workloads in a distributed key-value store, such as etcd. When a deployment of an application Pod with three instances or replicas is requested by the end user, a Kubernetes scheduler component allocates those Pods to be run on a set of nodes. Managing the desired state of the required deployment is the job of a controller object.

Kubernetes offers several variants of the controller objects which provide the flexibility to define the desired state of an application.

Replica Set/Replication Controller

A replica set is the next generation of a replication controller that ensures the desired state of replicas is running on the cluster at any given time.

Deployment

A deployment controller is a higher level abstraction of a replica set or a replication controller. It provides a declarative mechanism to update Pods and replica sets. A deployment has many more features than a replica set, including where rollout strategies and scaling rules can be defined.

Daemonset

A daemonset controller ensures that the defined Pod runs on each node in the Kubernetes cluster. This model provides the flexibility to run daemon processes, such as log management, monitoring, storage providers, or to provide Pod networking for the cluster.

Statefulset

A statefulset controller ensures that the Pods deployed as part of it are provided with durable storage and a sticky identity. The PPods are also provided with a stable network ID or hostname, since the hostname is a function of the Pod name and its ordinal index.

POD NETWORKING AND CNI PLUGINS

There are seven distinct paths available in Kubernetes networking.

1. Intra-Pod communication
2. Inter-Pod to Pod communication
3. Pod-to-Node and Node-to-Pod communication
4. Pod-to-Service communication
5. Pod-to-External communication
6. External-to -Service communication
7. External-to-Pod communication

INTRA-POD COMMUNICATION

A Pod is a collection of one or more containers that share certain resources. Since they share the same network stack and IP address as the Pod, they can reach each other with localhost ports.

Adopting the principle of multiple containers sharing the same network namespace provides a few benefits. There is increased security, since the ports bound to localhost are scoped only to the Pod and not outside it. There is also simplicity in networking between the colocated containers in a Pod.

Note that this communication path does not utilize the container network interface (CNI).

POD-TO-POD COMMUNICATION

Kubernetes assumes that Pods communicate with each other over routable IP addresses. This allows Pods to communicate without translations or service-discovery tooling. This is also fundamentally different from how Docker networking works. With Docker networking, containers are provided addresses in the 172.x range and the IP address that a peer container sees is different than what it is allocated. This model also allows for easier porting of applications into the Kubernetes platform where they might be running on VMs.

Container Network Interface (CNI)

The way to achieve Inter-Pod communication is via networking Plugins, which adhere to the CNI specification.

CNI plugins allow configuration and cleaning of the networking constructs, when a Pod is created or destroyed. They follow a specification which allows for the standardization of principles and the flexibility of implementation.

A CNI plugin is implemented as an executable binary which is passed to the kubelet running on each node. This plugin binary is responsible for the management of the host-to-Pod network, which translates to connecting the Pod's network namespace to the host network namespace. This is achieved by creating some form of virtual network interface between the Pod and the underlying host (or some shared networking construct on the host, external to the Pod). The CNI interface also provides the necessary configurations to enable Pod networking external to the host, such as routing, L2 overlay constructs, and NAT configuration. It also requires a plugin configuration which should be present on each node of the cluster, typically in /etc/cni/net.d.

The CNI plugin also interfaces with IP Address Management (IPAM) which is responsible for IP Address management and assignment of IP addresses to Pods. IPAM is also responsible for adding routes for the veth created for the Pod.

A few CNI plugins include:

- Flannel: Provides Overlay networking.
- Weave: Provides Overlay networking focused on Docker integration.
- Calico: A routed network that is able to either seamlessly interwork with the underlying infrastructure, or deploy in an overlay above it. It commonly, but not always, uses BGP for the internetworking/route exchanges.

SERVICES

Pod-to-Service Communication

Because Pods are ephemeral in nature, each time a Pod is destroyed and launched, the IP address will change. A Service object in Kubernetes provides an abstraction that enables groups of Pods to be addressed based on a label selector, rather than using an IP address.

Each service is assigned a Virtual IP address, also known as the ClusterIP that can be used to communicate to the Pods. Services provide load balancing and access to the underlying Pods, and use an object called Endpoints to track changes in IP addresses of the Pods. The Endpoints object is updated whenever the IP address of a Pod changes.

Services provide four ways to access Pods:

1. ClusterIP: Allows the Service to be accessible only from within the cluster. The Service is exposed on the cluster-IP address.
2. NodePort: Selects a port from the range 30000-32767 and will proxy this port on each node and update the Service endpoint. The NodePort service is accessible by addressing it via the NodeIP and the NodePort.
3. LoadBalancer: A load balancer type allows the use of an external load balancer to route traffic to the Pods. This is typically used with public cloud platforms.
4. ExternalName: A Service that specifies an ExternalName defines an alias to an external Service outside the cluster. When the service is looked up, the CNAME of the external service is returned. This type of service does not have an Endpoints object defined.

Here is a sample Service spec.

```
apiVersion: v1
kind: Service
metadata:
 name: myapp
 labels:
   component: myapp
   role: client
spec:
 type: NodePort
 selector:
   component: myapp
   role: client
 ports:
 - name: http
   port: 9090
   protocol: TCP
Here is a sample Endpoints spec
kind: Endpoints
apiVersion: v1
metadata:
  name: myapp
subsets:
  - addresses:
      - ip: 10.32.5.4
    ports:
      - port: 9090
```

INGRESS

External-to-Internal Communication

Pods and Services are assigned IP addresses that are part of the networking setup for the cluster. They are accessible outside the cluster via a LoadBalancer or NodePort. An Ingress is a way of specifying a set of rules that allow incoming traffic to the Service. This is typically used to provide the service a URL which can be accessed externally.

Here is a sample Ingress resource spec.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /myapp
spec:
  rules:
  - http:
      paths:
      - path: /myapp
        backend:
          serviceName: myapp-service
          servicePort: 9090
```

In this example, traffic is redirected to the myapp-service on the servicePort 9090 when the /myapp path is accessed.

An Ingress controller is required to fulfill the ingress rules specified for the service. An Ingress controller runs separately and isn't part of the controller manager that is run on the Kubernetes master.

Networking with Calico

ARCHITECTURE

Calico creates and manages a Layer 3 network that provides inter-Pod communication in the Kubernetes cluster. It provides routable IP addresses to Pods that enable easier interoperability. Calico allows enforcement of network security policies that provide fine-grained control over the communications between Pods.

Calico uses the following components to achieve this.

- calico/node: The agent that runs as part of the Calico daemonset Pod. It manages interfaces, routes, and status reporting of the node and enforces policies.
- BIRD: A BGP client that broadcasts routes that are programmed by Felix
- Etcd: An optional distributed datastore
- Calico Controller: The Calico policy controller

CALICO/NODE

calico/node is a pod with two containers.

1. A calico/node container that runs two daemon processes:
   a. Felix
   b. the Bird BGP daemon (optional)
2. A Calico-CNI plugin container (optional) that respond to CNI requests from the kubelet on the node

The Felix component is at the heart of networking with Calico. It runs on every node of the cluster and is responsible for interface and route management, state reporting, and policy enforcement.

Interface and Route Management

The Felix daemon is responsible for programming the interface and creating routes in the kernel route table to provide routable IP addresses for Pods when they get created. Felix creates a virtual network interface and assigns an IP address from the Calico IPAM for each Pod. This interface carries the prefix, *cali* unless specified otherwise.

This ensures that the Pods carry a routable IP address and the packets are routed appropriately. It also is responsible for cleaning up the interfaces when a Pod is evicted.

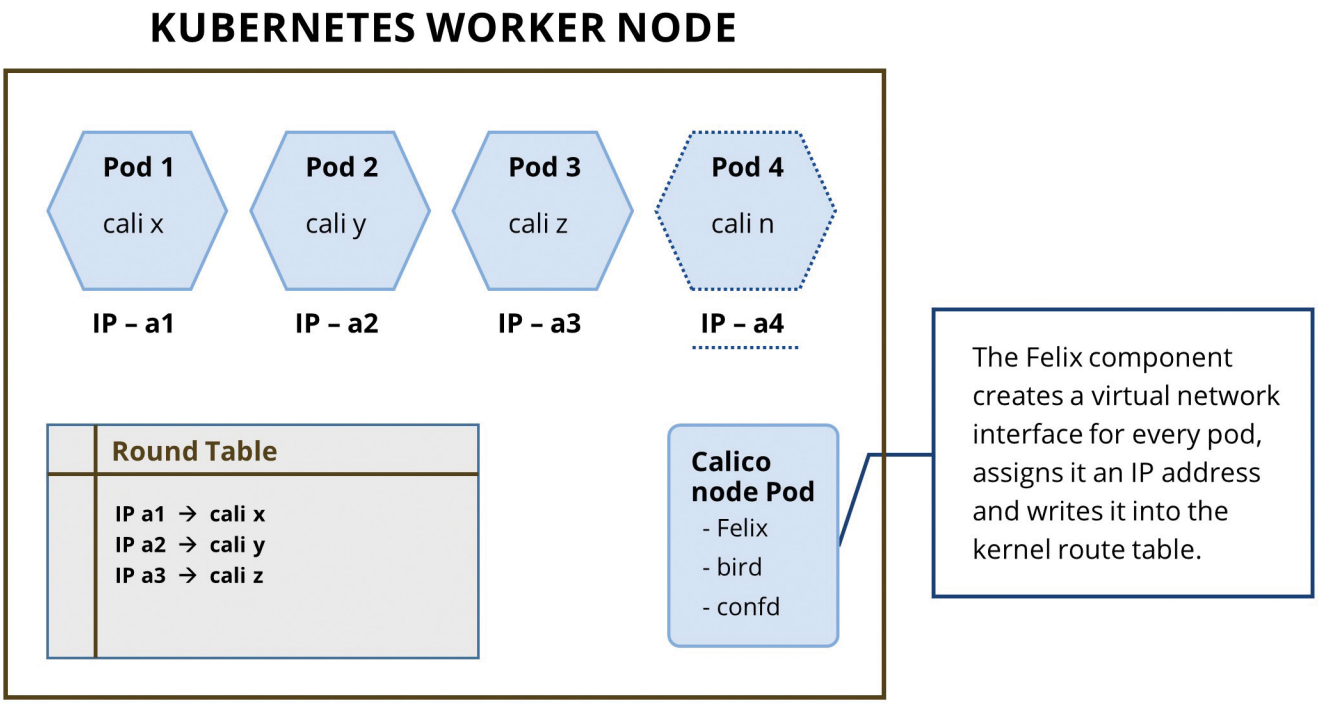
State Reporting

Felix exposes metrics that are used for instance state reporting via a monitoring tool, such as Prometheus.

Policy Enforcement

Felix is responsible for network policy enforcement. Felix monitors the labels on the Pods and compares against the defined network policy objects to decide whether to allow or deny traffic to the Pod.

Felix writes information about the interfaces with their IP Addresses and the state of the host network to etcd.



BIRD

BIRD is a BGP daemon that distributes routing information written by Felix to other BIRD agents on the cluster nodes. The BIRD agent is installed with the Calico daemonset Pod. This ensures that traffic is routed across nodes. Calico, by default, creates a full mesh topology. This means that every BIRD agent needs to be connected to every other BIRD agent in the cluster.

For larger deployments, BIRD can be configured as a route reflector. The route reflector topology allows BIRD to be set up as a centralized point which other BIRD agents communicate. It also reduces the number of open connections for each BGP agent.

ETCD

Calico uses a distributed datastore called etcd that stores the Calico resource configurations and network policy rules. The Felix daemon communicates with the etcd datastore for publishing routes, node, and interface information for every node among other information.

For higher availability, a multi-node etcd cluster should be set up for larger deployments. In this setup, etcd ensures that the Calico configurations are replicated across the etcd cluster enabling them to always be in the last known good state.

An optional deployment model is to use the Kubernetes API server as the distributed datastore, eliminating the need to stand-up and maintain an etcd datastore.

Bringing it all together :

An example of a deployment of three nginx Pods on a Kubernetes cluster demonstrates how these components work together to provide networking.

1. When one of the Nginx Pods is scheduled on the Kubernetes node, Felix will create a virtual interface with the cali prefix and assigns it a /32 IP address.

```
ubuntu@ip-172-31-59-6:~$ kubectl get pods -nweb -o wide
NAME                    READY     STATUS    RESTARTS   AGE       IP                NODE
nginx-8586cf59-759hb    1/1       Running   0          28m       192.168.162.146   ip-172-31-51-217
nginx-8586cf59-rv5gm    1/1       Running   0          28m       192.168.91.20     ip-172-31-53-121
nginx-8586cf59-s9qxq    1/1       Running   0          28m       192.168.88.23     ip-172-31-49-195
ubuntu@ip-172-31-59-6:~$ calicoctl get wep -nweb
NAMESPACE   WORKLOAD                NODE                NETWORKS             INTERFACE
web         nginx-8586cf59-s9qxq    ip-172-31-49-195    192.168.88.23/32     calib4f54adde68
web         nginx-8586cf59-759hb    ip-172-31-51-217    192.168.162.146/32   cali30c9d0d9912
web         nginx-8586cf59-rv5gm    ip-172-31-53-121    192.168.91.20/32     cali09f9c3884e9
```

Note the interface *cali09f9c3884e9* created for the Pod *nginx-8586cf59-rv5gm* scheduled on node *ip-172-31-51-121.*

```
ubuntu@ip-172-31-53-121:~$ ip route show
default via 172.31.48.1 dev eth0
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1 linkdown
172.31.48.0/20 dev eth0  proto kernel  scope link  src 172.31.53.121
192.168.88.0/26 via 172.31.49.195 dev tunl0  proto bird onlink
blackhole 192.168.91.0/26  proto bird
192.168.91.9 dev cali3f303ce6db3  scope link
192.168.91.15 dev calid3421b9ad70  scope link
192.168.91.16 dev cali01e9b155330  scope link
192.168.91.20 dev cali09f9c3884e9  scope link
192.168.162.128/26 via 172.31.51.217 dev tunl0  proto bird onlink
192.168.185.0/26 via 172.31.59.6 dev tunl0  proto bird onlink
```

Routes on the host where the Pod *nginx-8586cf59-rv5gm* is scheduled.

```
ubuntu@ip-172-31-53-121:~$ ip a show dev cali09f9c3884e9
32: cali09f9c3884e9@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netnsid 3
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
```

Cali Interface on the host where the Pod *nginx-8586cf59-rv5gm* is scheduled.

```
ubuntu@ip-172-31-59-6:~$ kubectl exec -it nginx-8586cf59-rv5gm -nweb -- /bin/sh
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1
    link/ipip 0.0.0.0 brd 0.0.0.0
4: eth0@if32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 72:9a:fd:a0:f4:9e brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.91.20/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::709a:fdff:fea0:f49e/64 scope link
       valid_lft forever preferred_lft forever
# arp
Address                  HWtype  HWaddress           Flags Mask            Iface
169.254.1.1              ether   ee:ee:ee:ee:ee:ee   C                     eth0
#
```

The interfaces in the container *nginx-8586cf59-rv5gm.* Arp, also shows that the eth0 interface within the Pod is mapped to the MAC address of the *cali09f9c3884e9* interface created by Felix. This is done by the host responding via proxy arp and returning the mac address of the cali interface.

This allows for the Pod traffic to be routed to the host.

Note: In some cases, the kernel is not able to create a persistent mac address for the interface. Since Calico uses point-to-point routed interfaces, traffic does not reach the data link layer the MAC Address is never used and can, therefore, be the same for all the cali interfaces which is ee:ee:ee…*

2. The BIRD BGP daemon realizes that there is a new network interface that has come up and it advertises that to the other peers.

```
ubuntu@ip-172-31-49-195:~$ ip route show
default via 172.31.48.1 dev eth0
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1 linkdown
172.31.48.0/20 dev eth0  proto kernel  scope link  src 172.31.49.195
blackhole 192.168.88.0/26  proto bird
192.168.88.18 dev calie219e8ce546  scope link
192.168.88.19 dev cali84e0bac56c0  scope link
192.168.88.23 dev calib4f54adde68  scope link
192.168.91.0/26 via 172.31.53.121 dev tunl0  proto bird onlink
192.168.162.128/26 via 172.31.51.217 dev tunl0  proto bird onlink
192.168.185.0/26 via 172.31.59.6 dev tunl0  proto bird onlink
```

Note the route with 192.168.91.0/26 routed via the IP address of the host *172.31.52.121* where the *nginx-8586cf59-rv5gm* Pod is.

Note: Notice that the route shows tunl0. This is because of the use of IPIP. Calico allows the use of a smarter way of defining IPIP via an option called cross-subnet. If IPIP is disabled, the route would show the actual interface name. E.g., eth0. For a discussion of IPIP vs direct networking, please see the 'Calico IP-in-IP mode' section below.

INSTALLATION

Installing Calico for networking with a Kubernetes cluster has a few prerequisites.

PRE-REQUISITES

- **kube-proxy** configured to run without the --masquerade-all option, since this conflicts with Calico.

  Note: If kubeadm is used to setup a Kubernetes cluster, an easier way to verify this would be to run the following command and verify the options under ipTables

  kubectl get cm kube-proxy -n kube-system -o yaml

- **kubelet** must be configured to use CNI plugins for networking, `--network-plugin=cni` as the option.

  Note: If kubeadm was used to set up a cluster, verify using the command `cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf`

- **kube-proxy** must run with the proxy mode of iptables. This is the default option.

INSTALL CALICO

Calico is deployed as a daemonset on the Kubernetes cluster. The daemonset construct of Kubernetes ensures that Calico runs on each node of the cluster. The installation is performed by applying a specification, which defines the required Kubernetes resources for it to function correctly.

Note: If the Kubernetes cluster is RBAC enabled, make sure the necessary RBAC policies are applied on the cluster for the Calico-kube-controller to function correctly.

kubectl apply -f https://docs.projectcalico.org/v3.1/getting-started/kubernetes/installation/rbac.yaml

The deployment manifest can be found at Calico Deployment Manifest and is applied via kubectl.

A quick inspection of the manifest shows the following Kubernetes resources defined.

- The Calico-config ConfigMap which contains parameters for configuring the install. This holds the etcd configuration parameter `etcd_endpoints` used to specify the etcd store for Calico.
- A calico/node Pod which contains the following containers:
  - A container that installs the CNI binaries and the CNI config in the standard locations as per the CNI specification (/etc/cni/net.d and /opt/cni/bin).
  - The calico/node container which manages routes and network policy.
- A Calico-etcd-secrets secret, which allows for providing etcd TLS assets.
- A Calico-kube-controller, which runs control loops to watch over node changes, policy changes and Pod label changes, to name a few.

Here is a sample screenshot showing what the cluster looks like after applying the Calico manifest.

```
root@k8s-master:~# kubectl get pods -nkube-system -o wide
NAME                                   READY     STATUS    RESTARTS   AGE       IP                NODE
calico-etcd-f69gb                      1/1       Running   0          3m        10.128.0.2        k8s-master
calico-kube-controllers-5449fdfcd-wfmz7   1/1       Running   0          3m        10.128.0.3        k8s-n1
calico-node-7q4p7                      2/2       Running   0          3m        10.128.0.2        k8s-master
calico-node-9sx6z                      2/2       Running   0          3m        10.128.0.3        k8s-n1
calico-node-jpn9x                      2/2       Running   0          3m        10.128.0.4        k8s-n2
etcd-k8s-master                        1/1       Running   0          1h        10.128.0.2        k8s-master
kube-apiserver-k8s-master              1/1       Running   0          1h        10.128.0.2        k8s-master
kube-controller-manager-k8s-master     1/1       Running   0          1h        10.128.0.2        k8s-master
kube-dns-86f4d74b45-lslfc              3/3       Running   0          1h        192.168.111.193   k8s-n2
kube-proxy-fjht2                       1/1       Running   0          1h        10.128.0.2        k8s-master
kube-proxy-rjpqw                       1/1       Running   0          1h        10.128.0.4        k8s-n2
kube-proxy-xrhd5                       1/1       Running   0          1h        10.128.0.3        k8s-n1
kube-scheduler-k8s-master              1/1       Running   0          1h        10.128.0.2        k8s-master
```

INSTALL AND CONFIGURE CALICOCTL

Calico provides a command line utility called calicoctl that is used to manage Calico configurations. The host where the calicoctl utility is run requires connectivity to the Calico etcd datastore. Alternatively, calicoctl can be configured to connect to the Kubernetes API datastore.

calicoctl can also be deployed as a standalone container or a Pod.

Install calicoctl as a stand-alone binary on any host that you may use for management.

curl -O -L https://github.com/projectcalico/calicoctl/releases

chmod +x calicoctl

sudo mv calicoctl /usr/local/bin

`calicoctl` requires information of the etcd datastore to communicate with it. This can be provided in two different ways.

1. Configuration file:

   `calicoctl` by default looks for a configuration file to be present at /etc/calico/calicoctl.cfg. The format of the calicoctl.cfg can either be YAML or JSON.

   A valid example configuration file looks like the one below.

```
apiVersion: projectcalico.org/v3
kind: CalicoAPIConfig
metadata:
spec:
  etcdEndpoints: https://etcd1:2379,https://etcd2:2379,https://etcd3:2379
  etcdKeyFile: /etc/calico/key.pem
  etcdCertFile: /etc/calico/cert.pem
  etcdCACertFile: /etc/calico/ca.pem
```

2. Environment Variables:

   In the absence of a configuration file, calicoctl can accept environment variables to connect to the etcd datastore.

   The table below defines the environment variables for a dedicated etcd datastore.

Environment Variable	Description	Schema
DATASTORE_TYPE	Indicates the datastore to use. If unspecified, defaults to etcdv3. (optional)	Kubernetes, **etcdv3**
ETCD_ENDPOINTS	This is a required parameter. The Calico etcd datastore URL endpoints. For example: http://etcd1:2379, http://etcd2:2379 where etcd1/etcd2 are either DNS names or IP Addresses.	string
ETCD_USERNAME	Username for RBAC. Example: user (optional)	string
ETCD_PASSWORD	Password for the given username. (optional)	string
ETCD_KEY_FILE	The path to the etcd key file. Example: /etc/calico/key.pem (optional)	string
ETCD_CERT_FILE	The path to the etcd client certificate, Example: /etc/calico/cert.pem (optional)	string
ETCD_CA_CERT_FILE	The path to the etcd Certificate Authority file. Example: /etc/calico/ca.pem (optional)	string

The table below defines the list of environment variables for the Kubernetes datastore.

Environment Variable	Description	Schema
DATASTORE_TYPE	Indicates the datastore to use. [Default: etcdv3]	**Kubernetes**, etcdv3
KUBECONFIG	When using the Kubernetes datastore, the location of a kubeconfig file to use, e.g. /path/to/kube/config.	string
K8S_API_ENDPOINT	Location of the Kubernetes API. Not required if using kubeconfig. [Default: https://kubernetes-api:443]	string
K8S_CERT_FILE	Location of a client certificate for accessing the Kubernetes API, e.g., /path/to/cert.	string
K8S_KEY_FILE	Location of a client key for accessing the Kubernetes API, e.g., /path/to/key.	string
K8S_CA_FILE	Location of a CA for accessing the Kubernetes API, e.g., /path/to/ca.	string
K8S_TOKEN	Token to be used for accessing the Kubernetes API.	string

Tip: The name of the configuration options to be used in the configuration file can be arrived at by camel casing the environment variable without the underscore.

For example: `*ETCD_CA_CERT_FILE*` as the environment variable can be used as `*etcdCACertFile*` in the configuration file.

`*K8S_API_ENDPOINT*` as the environment variable can be used as `*k8sAPIEndpoint*` in the configuration file.

Install calicoctl as a Pod in the Kubernetes Cluster.

To use a dedicated etcd datastore (recommended)
kubectl apply -f https://docs.projectcalico.org/master/getting-started/kubernetes/installation/hosted/calicoctl.yaml

To use the Kubernetes datastore

kubectl apply -f https://docs.projectcalico.org/master/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calicoctl.yaml

POST INSTALL VERIFICATION

The following steps describe the quickest way to verify a successfully running Calico installation.

Create a nginx deployment with two replicas.

`kubectl run nginx --image=nginx --replicas=2`

This creates two nginx Pods scheduled on the Kubernetes worker nodes.

```
root@k8s-master:~# kubectl run nginx --image=nginx --replicas=2
deployment.apps "nginx" created
root@k8s-master:~# kubectl get pods -o wide
NAME                     READY     STATUS    RESTARTS   AGE       IP                NODE
nginx-65899c769f-gtmtb   1/1       Running   0          9s        192.168.111.196   k8s-n2
nginx-65899c769f-xvntk   1/1       Running   0          9s        192.168.215.67    k8s-n1
```

Verify Calico node status via calicoctl

`calicoctl node status`

```
root@k8s-master:~# calicoctl node status
Calico process is running.

IPv4 BGP status
+--------------+-------------------+-------+------------+-------------+
| PEER ADDRESS |     PEER TYPE     | STATE |   SINCE    |    INFO     |
+--------------+-------------------+-------+------------+-------------+
| 10.128.0.3   | node-to-node mesh | up    | 2018-03-31 | Established |
| 10.128.0.4   | node-to-node mesh | up    | 2018-03-31 | Established |
+--------------+-------------------+-------+------------+-------------+

IPv6 BGP status
No IPv6 peers found.
```

Verify the workload endpoints via `calicoctl`

`calicoctl get workLoadEndpoints`.

```
root@k8s-master:~# calicoctl get workLoadEndpoints
WORKLOAD                  NODE     NETWORKS             INTERFACE
nginx-65899c769f-xvntk    k8s-n1   192.168.215.67/32    cali25b3cd8343b
nginx-65899c769f-gtmtb    k8s-n2   192.168.111.196/32   cali220fbec450c
```

The Pods show a routable IP address for each running Pod.

Login to the hosts that are running the nginx Pods to inspect the route table.

```
root@k8s-n1:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.128.0.1      0.0.0.0         UG    0      0        0 ens4
10.128.0.1      0.0.0.0         255.255.255.255 UH    0      0        0 ens4
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.111.192 10.128.0.4      255.255.255.192 UG    0      0        0 tunl0
192.168.215.64  0.0.0.0         255.255.255.192 U     0      0        0 *
192.168.215.67  0.0.0.0         255.255.255.255 UH    0      0        0 cali25b3cd8343b
192.168.235.192 10.128.0.2      255.255.255.192 UG    0      0        0 tunl0
root@k8s-n1:~#
```

```
root@k8s-
root@k8s-n2:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.128.0.1      0.0.0.0         UG    0      0        0 ens4
10.128.0.1      0.0.0.0         255.255.255.255 UH    0      0        0 ens4
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.111.192 0.0.0.0         255.255.255.192 U     0      0        0 *
192.168.111.193 0.0.0.0         255.255.255.255 UH    0      0        0 calid5f3280f76c
192.168.111.196 0.0.0.0         255.255.255.255 UH    0      0        0 cali220fbec450c
192.168.215.64  10.128.0.3      255.255.255.192 UG    0      0        0 tunl0
192.168.235.192 10.128.0.2      255.255.255.192 UG    0      0        0 tunl0
```

The Felix daemon in the Calico node Pod has created a virtual network interface with a routable IP address for each running Pod on the node. The name of the interface is named cali appended by an alphanumeric number.

`*cali25b3cd8343b*` is the name of one of the interfaces on the nginx Pod.

OVERLAY AND NON-OVERLAY NETWORKING

Calico provides a standard networking model based on the principles of the Internet. It allows for standard BGP peering for on-premises solutions as well as IP-in-IP encapsulation. Calico also can be deployed as a network policy enforcement engine alongside Flannel as the CNI plugin, which provides the required Overlay network.

CALICO IP-IN-IP MODE

Calido uses IP-in-IP mode when

- There is no other way to share routes between the infrastructure;
- The network architecture performs validates addresses and discards all traffic that is not recognized;
- The Pods need to communicate across subnets.

Public cloud environments usually impose these restrictions.

Calico can be configured to use IP-in-IP encapsulation by setting the IPIP configuration parameter on the IP Pool resource. When enabled, Calico encapsulates the packets originating from a Pod in a packet which carries the header containing the source IP address of the host and the destination IP of the host where the target Pods are running. This IP-in-IP encapsulation is performed by the Linux kernel using stateless IP-in-IP encapsulation. The routes in this mode are not pre-created. It allows for dynamic IP-in-IP encapsulation. This allows the router to make the route decision following standard network routing.

Calico IP-in-IP provides three modes of operation to allow network traffic to use IP-in-IP encapsulation. These options are as follows.

1. ipip Mode: Always.
2. ipip Mode: Cross Subnet
3. ipip Mode: Off

When ipip Mode is set as Always, all packets originating from the Pod are encapsulated using the IPIP mechanism.

With ipip Mode set to CrossSubnet, Calico provides the ability to perform encapsulation only when it is necessary. IP-in-IP encapsulation in this mode is performed only when the traffic needs to cross over subnet boundaries.

Applying the following spec would enable CrossSubnet mode for the IP Pool my.ippool

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: my.ippool
spec:
  cidr: 10.1.0.0/16
  ipipMode: CrossSubnet
  natOutgoing: true
  disabled: false
```

Note: natOutgoing is always required if IPIP is enabled and set to Always or CrossSubnet mode. If IPIP is not enabled, or the mode is set to off, natOutgoing may not be necessary. This depends on the network and addressing architecture.

IP ADDRESS MANAGEMENT

The Calico IP Address Management (IPAM) is responsible for IP address management for Pods on each node of the Kubernetes cluster.

The CNI plugin is responsible for the creation of the communication channel from the Pod to the network. In order to do so, the CNI plugin points to an IPAM module which is responsible for assigning IP Addresses to Pods.

The Calico CNI configuration uses Calico-ipam as the default configuration for the CNI plugin.

Here is an example of a Calico CNI configuration file.

```
{
  "name": "k8s-pod-network",
  "cniVersion": "0.3.0",
  "plugins": [
    {
      "type": "calico",
      "etcd_endpoints": "http://10.96.232.136:6666",
      "log_level": "info",
      "mtu": 1500,
      "ipam": {
          "type": "calico-ipam"
      },
      "policy": {
          "type": "k8s",
           "k8s_api_root": "https://10.96.0.1:443",
           "k8s_auth_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZ-
pY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN-
5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJjYWxpY28tY-
25pLXBsdWdpbi10b2tlbi03MnM0YiIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2Vydml-
jZS1hY2NvdW50Lm5hbWUiOiJjYWxpY28tY25pLXBsdWdpbiIsImt1Ym"
      },
      "kubernetes": {
          "kubeconfig": "/etc/cni/net.d/calico-kubeconfig"
      }
    },
    {
      "type": "portmap",
      "snat": true,
      "capabilities": {"portMappings": true}
    }
  ]
}
```

The kubelet on each node uses this CNI configuration to set up network connectivity for the Pods running on that host.

IP POOLS

IP Pools are a collection of IP Addresses that are used for assigning IP addresses to Pods. The IPPools Calico resource can be managed using the calicoctl command line.

The IPPools are created during the deployment of the Calico daemonset. The parameter, CALICO_IPV4POOL_CIDR defines the IPPool addresses.

```
root@k8s-master:~# calicoctl get ippools
NAME                  CIDR
default-ipv4-ippool   192.168.0.0/16
```

Multiple IP Pools

Calico supports the creation of multiple IP Pools in the IPAM. Calico can assign IP Addresses to Pods from a specific IP Pool based on the Pod annotation. This allows for fine-grained control over the IP Address assignment of Pods.

Using the Calico IPAM feature, it is possible to create multiple IP Pools – one of them routable and the other with a private address range IP Pool. Then, a subset of Pods is allowed to be assigned IP addresses from the routable pool whereas the rest of the Pods to be assigned IP addresses from the private pool.

IP Pool per Pod

Calico IPAM allows specification of an IP Pool to be used Per Pod in addition to specifying the IP Pools in the CNI config. This is achieved by using Kubernetes Pod Annotations.

This annotation is used for IPv4 annotations:

"cni.projectcalico.org/ipv4pools": "[\"10.1.0.0/16\"]"

This annotation is used for IPv6 annotations:

"cni.projectcalico.org/ipv6pools": "[\"2001:db8::1/120\"]"

Note: The IP Pool needs to be created before using it as part of the Pod annotation.

Here is an example Pod specification with the IPPool annotation

```
apiVersion: v1
kind: Pod
metadata:
  name: private-nginx
  labels:
    app: private-nginx
  annotations:
    "cni.projectcalico.org/ipv4pools": "[\"10.1.0.0/16\"]"
spec:
  containers:
  - image: nginx
    name: private-nginx
```

Manual IP Per Pod

Calico allows requesting for a specific IP address from the IP Pool for a Pod. The requested IP addresses will be assigned from Calico IPAM and must exist within a configured IP pool.

The following annotation is used.

Annotations:
"cni.projectcalico.org/ipAddrs": "[\"192.168.0.1\"]"

Calico also allows a means to bypass the IPAM to request a specific IP address. Calico only distributes routes for the IP addresses, which are part of the Calico IP Pool. The routing or conflict resolution needs to be taken care of manually or via another mechanism.

Annotations:
"cni.projectcalico.org/ipAddrsNoIpam": "[\"10.0.0.1\"]"

The annotations ipAddrs and ipAddrsNoIpam cannot be used simultaneously on the Pod Specification.

NAT

Calico allows setting outbound Internet access to Pods on the IP Pool. Calico will then perform outbound NAT on the node where the Pod is scheduled. This is achieved by setting the parameter.

Set `natOutgoing` on the IP Pool configuration to true. This parameter accepts a Boolean value with the default being `false`.

A sample IP Pool definition

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: my.ippool
spec:
  cidr: 10.1.0.0/16
  ipipMode: CrossSubnet
  natOutgoing: true
  disabled: false
```

ADDRESS BLOCKS

Calico does dynamic address management by allocating a /26 address block to each node when the Calico node runs successfully.

This /26 space address block is written to the etcd datastore to reserve it. All Pods scheduled on that host then use IP addresses from that address block.

A /26 address space allocation allows assignment of IP addresses to 64 Pods. If an address block gets exhausted, then another /26 address block is associated to the node.

Calico follows this model to avoid wasting IP Addresses. This helps optimize the IP address management and Calico will simply request another /26 address block when the Node exhausts all available IP addresses.

The following screenshow shows the /26 block reservation against the node k8s-n1 and the Pod running on that node allocated an IP address from that address range.

```
root@k8s-master:~# kubectl get pods -o wide
NAME                     READY     STATUS    RESTARTS   AGE       IP                NODE
nginx-65899c769f-gtmtb   1/1       Running   0          1h        192.168.111.196   k8s-n2
nginx-65899c769f-xvntk   1/1       Running   0          1h        192.168.215.67    k8s-n1
root@k8s-master:~#
```

```
root@k8s-n1:~# ip route show
default via 10.128.0.1 dev ens4
10.128.0.1 dev ens4  scope link
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1 linkdown
192.168.111.192/26 via 10.128.0.4 dev tunl0  proto bird onlink
blackhole 192.168.215.64/26  proto bird
192.168.215.67 dev cali25b3cd8343b  scope link
192.168.235.192/26 via 10.128.0.2 dev tunl0  proto bird onlink
```

ADDRESS BORROWING

In situations where no more /26 blocks are available, Calico uses a method called Address Borrowing.

In a very large cluster, Calico borrows the specific /32 IP Address from the address block of another node and uses that for IP assignment to the Pod. It then advertises that on the cluster. A very large cluster is defined as where the IP Pool consists of a /16 block and individual nodes are allocated /26 address blocks and there are no more /26 address blocks to be allocated for a new Pod on a node, but there are individual /32 address on another node (the IP addresses of the /26 block are not exhausted on another node)Pod.

Due to the general network policy of routing where network traffic always tries to find the most specific IP addresses. Then, it rolls up if that address is not directly accessible. Communication to the Pod IP happens directly due to its allocated /32 Address.

INSTALLATION INSTRUCTIONS

Please see the current installation documents for a discussion of the various ways to install and configure Calico on either public or private cloud. Further discussion about considerations specific to a given public cloud can be found in the reference documents section titled *configuration on public clouds*. A discussion of private cloud network architectures can be found in the network architecture documents, with the most common deployment model being AS-per-Rack.

NETWORK POLICY FOR NETWORK SEGMENTATION

Network policy allows you to specify access control rules for network communication between workloads using a declarative model. Following are some details of network policy.

MOTIVATION FOR NETWORK POLICY

As Kubernetes is adopted as part of an organization's IT environment. Security controls will need to be applied to:

- Manage fine-grained access controls to application workloads (Pods).
- Manage these restrictions dynamically on a running workload to allow or deny traffic

Kubernetes Pods, by default, do not restrict any incoming traffic. There are no firewall rules for inter-Pod communication. The need to have proper access restrictions to the workloads becomes imperative when Kubernetes is used as a multi-tenant platform or as a shared PaaS.

In order to provide a simple yet native interface for management of these access controls, a network policy object was introduced as a Kubernetes resource. This network policy object allows a user to define a set of access rules for to a set of workloads. The network policy resource requires a CNI plugin that supports the policy feature to enable enforcement of these rules.

The Calico CNI plugin has the capability to define and enforce network policies that can be used in conjunction with the Kubernetes network policy object.

Network policy definitions are implemented for the following use cases:

- Granular control over a multi-tiered application, such as allowing network traffic to a set of database Pods originating from a specific set of web application Pods.
- Allowing communication to the caching Pods only from a set of Pods running the UI application.

LABELS AND LABEL SELECTORS

Access control mechanisms allowing and denying traffic between Pods is achieved by using label selectors as part of the network policy definition.

Labels are a set of key/value pairs that can be applied to an object, typically a Pod, and are used to define the identity of the Kubernetes object. They also help organize and group these objects.

Labels have reasonable restrictions on the length and value that can be used. They do not have a lifecycle of their own and are applied to the Pod with the objective of providing a meaningful definition of the Pod. They can be applied during the definition of the Pod or later while it is running.

A label selector is a condition which selects an object based on the labels that are applied to it.

The following table shows the primitives that can be used as part of the label selector in the Calico network policy object. They can all be combined together by the standard `&&` primitive.

Syntax	Schema
all()	Match all resources.
k == 'v'	Matches any resource with the label 'k' and value 'v'.
k != 'v'	Matches any resource with the label 'k' and value that is not 'v'.
has(k)	Matches any resource with label 'k', independent of value.
!has(k)	Matches any resource that does not have label 'k'
k in { 'v1', 'v2' }	Matches any resource with label 'k' and value in the given set
k not in { 'v1', 'v2' }	Matches any resource without label 'k' or any with label 'k' and value not in the given set

DEFINING POLICY [K8S POLICY API, CALICO NETWORK POLICY]

Network Policy in Kubernetes

Network policy objects are namespaced. They follow the standard specification model of defining Kubernetes objects.

A sample Kubernetes network policy object is as described below.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-network-policy
  namespace: np-demo
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

The apiVersion, kind and metadata fields are mandatory similar to other object definitions with Kubernetes.

The spec section is the definition of the network policy for the provided namespace (np-demo in the example above).

policyTypes: Each network policy includes a policyTypes list which may include either Ingress, Egress, or both.

- Ingress: defines rules for incoming traffic. Read as `Allow traffic from`
- Egress: defines rules for outgoing traffic. Read as `Allow traffic to`

If no `*policyTypes*` parameter is specified, Ingress is applied.

Ingress: Each rule allows traffic which matches both the from and ports sections. In the above example, the policy contains a single rule, which matches traffic on the port 6379, from one of three sources: specified via an ipBlock, a namespaceSelector and a PodSelector.

Egress: Each network policy may include a list of whitelist egress rules. Each rule allows traffic which matches both the to and ports sections. The example policy contains a single rule, which matches traffic on a single port to any destination in 10.0.0.0/24.

Calico Network Policy

Calico provides the capability to define network policy and global network policy objects. They are applied to a workload endpoint object in Calico to define access restrictions for Pod workloads.

Workload Endpoint:

A workload endpoints resource in Calico is the object which holds information about the virtual cali interface associated to the Pod. The workload endpoint resource is a namespaced object. Fora network policy to be applied on this endpoint, it needs to be in the same namespace.

The workload endpoint resource can define a set of labels and profiles which are used by Calico to enforce network policy on the Pod. Defining a profile is an alternate way of applying access rules on individual endpoints.

An example definition of the workload endpoint resource:

```
apiVersion: projectcalico.org/v3
kind: WorkloadEndpoint
metadata:
  name: k8s-node1-k8s-frontend--5gs43-eth0
  namespace: myproject
  labels:
    app: frontend
    projectcalico.org/namespace: myproject
    projectcalico.org/orchestrator: k8s
spec:
  node: k8s-node1
  orchestrator: k8s
  endpoint: eth0
```

```
containerID: 13372515694203141592653 5
pod: my-nginx-b1337a
endpoint: eth0
interfaceName: cali0ef25fa
mac: ca:fe:1d:52:bb:e9
ipNetworks:
- 192.168.0.0/16
profiles:
- frontend-profile
ports:
- name: connect-port
  port: 1234
  protocol: tcp
- name: another-port
  port: 5432
  protocol: udp
```

Network Policy and Global Network Policy:

Calico, like Kubernetes, allows the definition of a network policy resource which is a namespaced resource and is applied to the workload endpoints resource.

Network Policy:

A sample Calico network policy resource definition:

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: allow-tcp-6379
  namespace: myproject
spec:
  selector: app == 'database'
  types:
  - Ingress
  - Egress
  ingress:
  - action: Allow
    protocol: TCP
    source:
      selector: app == 'frontend'
    destination:
      ports:
      - 6379
  egress:
  - action: Allow
```

The above policy allows incoming traffic on port 6379 for the endpoints that carry the label `app: database` and when the source is from the endpoints which carry the labels `app: frontend`. This allows a declarative way of specifying rules between a group of endpoints matched based on the label selector.

The Spec section describes the rules to be applied for the type of traffic, which is applied based on labels of the workload.

Field	Description	Accepted Values	Schema	Default
order	Controls the order of precedence. Calico applies the policy with the lowest value first.		float	
selector	Selects the endpoints to which this policy applies.		selector	all()
types	Applies the policy based on the direction of the traffic. To apply the policy to inbound traffic, set to Ingress. To apply the policy to outbound traffic, set to Egress. To apply the policy to both, set to Ingress, Egress.	Ingress, Egress	List of strings	Depends on the presence of ingress/ egress rules*
ingress	An ordered list of ingress rules applied by policy.		List of Rule	
egress	An ordered list of egress rules applied by this policy.		List of Rule	
doNotTrack*	Indicates to apply the rules in this policy before any data plane connection tracking, and that packets allowed by these rules should not be tracked.	true, false	boolean	false
preDNAT*	Indicates to apply the rules in this policy before any DNAT.	true, false	boolean	false
applyOnForward*	Indicates to apply the rules in this policy on forwarded traffic as well as to locally terminated traffic.	true, false	boolean	false

Types

The *types* field defines the type of traffic for which the policy is applied. In the absence of the types field, the following defaults apply.

Ingress Rules Present	Egress Rules Present	Type Value
No	No	Ingress
Yes	No	Ingress
No	Yes	Egress
Yes	Yes	Ingress, Egress

Rule

A Rule is the definition of the action to be taken with the Source and the Destination.

Field	Description	Accepted Values	Schema
action	Action to perform when matching this rule.	Allow, Deny, Log, Pass	string
protocol	Positive protocol match.	TCP, UDP, ICMP, ICMPv6, SCTP, UDPLite, 1-255	string | integer
notProtocol	Negative protocol match.	TCP, UDP, ICMP, ICMPv6, SCTP, UDPLite, 1-255	string | integer
icmp	ICMP match criteria.		ICMP
notICMP	Negative match on ICMP.		ICMP
source	Source match parameters.		EntityRule
destination	Destination match parameters.		EntityRule
ipVersion	Positive IP version match	4, 6	integer
http	Match HTTP request parameters. Application layer policy must be enabled to use this field.		HTTPMatch

If the action is specified as *Pass*, further policies would be skipped and the Profile rules specified against the endpoint would be executed. If there are no Profiles set for the workload endpoint, the default applied action is *Deny.*

Entity Rule

An Entity rule helps match the workload endpoints based on selectors.

Field	Description	Accepted Values	Schema
nets	Match packets with IP in any of the listed CIDRs.	List of valid IPv4 or IPv6 CIDRs	list of CIDRs
notNets	Negative match on CIDRs. Match packets with IP not in any of the listed CIDRs.	List of valid IPv4 or IPv6 CIDRs	list of CIDRs
selector	Positive match on selected endpoints. If a namespaceSelector is also defined, the set of endpoints this applies to is limited to the endpoints in the selected namespaces.	Valid selector	selector
notSelector	Negative match on selected endpoints. If a namespaceSelector is also defined, the set of endpoints this applies to is limited to the endpoints in the selected namespaces.	Valid selector	selector
namespaceSelector	Positive match on selected namespaces. If specified, only workload endpoints in the selected Kubernetes namespaces are matched. Matches namespaces based on the labels that have been applied to the namespaces. Defines the context that selectors will apply to, if not defined then selectors apply to the network policy's namespace.	Valid selector	selector
ports	Positive match on the specified ports		list of ports
notPorts	Negative match on the specified ports		list of ports
serviceAccounts	Match endpoints running under service accounts. If a namespaceSelector is also defined, the set of service accounts this applies to is limited to the service accounts in the selected namespaces. Application layer policy must be enabled to use this field.		Service AccountsMatch

A global network policy provides the ability to define a network policy which is non-namespaced. A global network policy, similar to a network policy, is applied on a group of endpoints matched by the label selector. The difference is it applies to all endpoints regardless of the namespace that they are in.

Global Network Policy

A global network policy also applies policy rules to a host endpoint. A host endpoint resource carries the information about the interface attached to the host which is running Calico. If a host endpoint carries a label which is matched by the global network policy, the policy rules get applied to it.

This capability allows the host network interface to be secured by applying declarative rules to it as well.

A sample global network policy object definition:

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: allow-tcp-6379
spec:
  selector: app == 'database'
  types:
  - Ingress
  - Egress
  ingress:
  - action: Allow
    protocol: TCP
    source:
      selector: app == 'frontend'
    destination:
      ports:
      - 6379
  egress:
  - action: Allow
```

This policy allows incoming traffic to the endpoints with the label `app: database` on port 6379 originating from endpoints with the label `app: frontend` and is applied across namespaces.

POLICY ENFORCEMENT

Declaring network policy and global network policy, Calico resources provide the flexibility to define access rules for an individual endpoint or to a group of endpoints either in a namespace or across namespaces. This is handled by label selectors that are present on the endpoints or the host endpoint resource.

Calico enforces these policy rules as defined in the above resources by translating them into iptable and ipset rules.

The Felix agent that runs as part of the Calico daemonset on Kubernetes reads the declared network policies and translates them into iptable rules and ipsets on the host where the Pod runs.

Calico policies are applied on workload endpoints. Workload endpoints already hold information about the Calico interface, the Pod using that interface, and the node on which the Pod is scheduled. This enables Calico to provide security and isolation using the native kernel iptable rules. No custom modules are required to achieve this.

Calico runs a Calico-kube-controller Pod as part of installation. The controller Pod has several controller loops, including a policy controller. The policy controller watches over network policies and programs Calico policy objects. Removal or changes to the network policies are reflected by the controller loop which ensures the policy changes are applied to workloads in real time.

HIERARCHICAL POLICIES [COMMERCIAL SOFTWARE CAPABILITY]

Hierarchical policies can be setup using Tigera Secure Enterprise Edition (TSEE).

TSEE introduces a resource object called Tier that represents an ordered collection of network policies and global network policies. A Tier helps divide the network policies into groups of different priorities. Policies within a Tier are ordered based on the order field on the network policy. The order of execution of a Tier is dependent on the order field defined on the Tier object. It is important to note that this evaluation, like all rules in Calico and TSEE, is evaluated at creation or modification time and not at packet time.

Important flexibility provided by the Tier resource is when a Pass action is encountered on the network policy rule, the next Tier is executed. This allows network policies to be modeled in a hierarchy.

The introduction of Tier helps in the following ways.

- Allowing privileged users to specify Policy rules which take precedence over other policies
- A physical firewall hierarchy can be translated directly in terms of Network Policies defined via Calico

The way a hierarchical policy is evaluated is as follows. When a new endpoint access is processed by Calico, every Tier that has a network policy defined for the endpoint processes the request. A Tier with the least value of the order field takes precedence over a higher value. Policies within each Tier are then processed on the value of the order field of the Policy object.

- If a network policy or global network policy in the Tier Allows or Denies the packet, then the packet is handled accordingly.
- If a network policy or global network policy in the Tier Passes the packet, the next Tier containing a policy that applies to the endpoint processes the packet.

If the last Tier (the one with the highest value in the order field) applying to the workload endpoint Passes the packet, that is evaluated.

An example of the Tier resource:

```
apiVersion: projectcalico.org/v3
kind: Tier
metadata:
  name: internal-access
spec:
  order: 100
```

Network policy and global network policy objects are added to a Tier by specifying the name of the Tier under the metadata section. The requirement is that the name should follow the format of *<tier name>.<policy name>*. If the Tier name is not specified, the calicoctl command would not allow the policy object to be created.

Kubernetes network policies are always created in the default Tier. The default Tier carries the order as <nil>, which is treated as infinite and hence is the lowest in precedence when using Tiers.

SECURING KUBERNETES CONNECTIVITY

There are a number of steps that can be performed to secure application workloads on Kubernetes. Some of them are as below.

The API service of Kubernetes is the brain of all the workloads that are being run on the cluster. By default, the API server traffic is TLS enabled. There needs to be a review of the installation mechanism to verify the API server traffic is secure, always authenticated and authorized via RBAC (Role-Based Access Control).

Kubernetes also distinguishes between a user account and a service account. Enabling RBAC on the Kubernetes cluster is imperative to secure workloads. Service accounts are typically namespaced and created by the application to run the Pods. The right API-level access restrictions must also be provided on the service accounts which run the applications.

The kubelet is responsible for running application workloads on its respective node. It is also one of the components that talks to the API server to receive the workload definitions for itself. The kubelet exposes an HTTPS endpoint which grants access over the node and the Pods running on that node. The Kubernetes cluster should enable kubelet authentication and authorization.

Enabling resource and limit quotas helps to address starvation of resources. Quotas can also be set up on the number of Pods that can be created in a namespace.

Pods can request privileged access by using the Pod security context where a specific user can be defined to run the Pod. Pod security policies enable fine-grained control over Pod creation. This makes it possible to restrict the running of privileged Pods.

Pod placement, with Affinity and Anti-Affinity rules, taint and tolerations-based evictions, helps control where the Pods can be scheduled via Kubernetes.

With all of the above measures, the approach to securing the Kubernetes cluster or its workloads can best be described as enforcing a perimeter over what actions can be performed or not.

DEFICIENCIES IN THE TRADITIONAL APPROACH

With the adoption of a microservices- based architecture for developing applications, an enterprise can potentially have services which run in the thousands. Each of which these can be independently scaled, versioned and deployed. These independently- deployed services have well-defined communication points with other services, which typically is via consuming REST APIs.

Enterprises also require consumption of these services over multiple channels, such as mobile devices, and intranet/extranet applications, as well as upstream and downstream applications interacting with the API endpoints.

One of the assumptions of the perimeter security model is that the boundaries of the application largely do not change.

The boundaries of the application continue to be pushed with the adoption of cloud services for deployment of applications, them applications spanning across regions, and the need to provide a unified experience across mobile devices, all of these factors keep pushing the boundaries of the application. With the adoption of public cloud infrastructure, applications are designed to be elastically scalable thereby creating a dynamic workload. Perimeter security does not scale well with workloads that largely are dynamic or elastic in nature.

This problem is further compounded wWith the adoption of hybrid cloud environments where part of application workloads run on a Private private cloud and other workloads run on Public public clouds this problem compounds further.

Another assumption with providing perimeter security is that the traffic within the perimeter is assumed to be safe and a level of trust is placed on the communication of workloads within the network. Spear-phishing, privileged user access, and virtualized environments all work on a trusted network where finding the root cause of the breach can take a long time to discover.

The perimeter model, thus although necessary, isn't usually sufficient enough to deal with this kind of dynamic infrastructure and heterogeneous workloads.

ZERO-TRUST APPROACH

The traditional security models, which depend on placing a level of trust in the network, are being migrated to a zero-trust model. The guiding principle of the zero trust model is as follows.

- Trust is only established once the identity of the application or a user is verified.
- The Zero zero-trust model assumes that the intranet or internal network is as hostile as the external network.
- A local network does not mean that it is trustworthy.
- Policies need to be dynamic in nature to accommodate the dynamic nature of workloads being deployed.
- Policies need to be enforced at multiple points in the infrastructure.

Kubernetes makes few assumptions about the security models that can be placed on the cluster. In the case of Kubernetes and Calico, the assumptions of the zero-trust model could be fulfilled as below.

- Calico allows enabling host protection on the interfaces of your VMs where the Kkubernetes workloads run with policy management for the ingress and egress traffic.
- Role-based access controls are enforced on the workloads that run on the cluster.
- Sensitive workloads should be deployed with TLS/SSL capabilities even within a trusted network.
- Calico Network network Policy policy enforcement provides an abstraction above IP addresses of workloads thus providing dynamic policy management.
- With the nature of kKubernetes, it is always assumed that IP addresses are not a durable identity and can change, h. Hence, network policies need to be designed at a higher abstraction.

MONITORING AND TROUBLESHOOTING

MONITORING

As described earlier, Calico gets deployed as a daemonset in the Kubernetes cluster to ensure the calico/node Pod runs on each member of the cluster. The Felix agent runs as part of the calico/node Pod and carries out several functions, such as route programming, access control list (ACL) enforcement and interface management on the host. This agent also allows scraping of metrics, via Prometheus, to enable proactive monitoring of the agent.

Felix can be configured using the FelixConfiguration object resource. Calico creates a default FelixConfiguration which represents cluster level settings. individual node.<nodename> contain node specific overrides.

To enable Prometheus metrics reporting, turn the prometheusMetricsEnabled field to true.

An example default FelixConfiguration spec:

```
apiVersion: projectcalico.org/v3
kind: FelixConfiguration
metadata:
  name: default
spec:
  ipipEnabled: true
  logSeverityScreen: Info
  reportingInterval: 0s
  prometheusMetricsEnabled: true
prometheusMetricsPort: 9081
```

This enables Prometheus stats reporting at the port 9081 for every node.

Calico currently enables the following metrics to be reported to Prometheus.

Name	Description
felix_active_local_endpoints	The number of active endpoints on this host.
felix_active_local_policies	The number of active policies on this host.
felix_active_local_selectors	The number of active selectors on this host.
felix_active_local_tags	The number of active tags on this host.
felix_calc_graph_output_events	The number of events emitted by the calculation graph.
felix_calc_graph_update_time_seconds	Seconds to update calculation graph for each datastore OnUpdate call.
felix_calc_graph_updates_processed	Number of datastore updates processed by the calculation graph.
felix_cluster_num_host_endpoints	The total number of host endpoints cluster-wide.
felix_cluster_num_hosts	The number of CNX hosts in the cluster.
felix_cluster_num_workload_endpoints	The total number of workload endpoints cluster-wide.
felix_exec_time_micros	Summary of time taken to fork/exec child processes
felix_int_dataplane_addr_msg_batch_size	The number of interface address messages processed in each batch. Higher values indicate we're doing more batching to try to keep up.
felix_int_dataplane_apply_time_seconds	Time in seconds that it took to apply a dataplane update.
felix_int_dataplane_failures	The number of times dataplane updates failed and will be retried.
felix_int_dataplane_iface_msg_batch_size	The number of interface state messages processed in each batch. Higher values indicate we're doing more batching to try to keep up.
felix_int_dataplane_messages	Number dataplane messages by type.
felix_int_dataplane_msg_batch_size	The number of messages processed in each batch. Higher values indicate we're doing more batching to try to keep up.
felix_ipset_calls	The number of ipset commands executed.
felix_ipset_errors	The number of ipset command failures.
felix_ipset_lines_executed	The number of ipset operations executed.
felix_ipsets_calico	The number of active CNX IP sets.
felix_ipsets_total	The total number of active IP sets.

Name	Description
felix_iptables_chains	The number of active iptables chains.
felix_iptables_lines_executed	The number of iptables rule updates executed.
felix_iptables_restore_calls	The number of iptables-restore calls.
felix_iptables_restore_errors	The number of iptables-restore errors.
felix_iptables_rules	The number of active iptables rules.
felix_iptables_save_calls	The number of iptables-save calls.
felix_iptables_save_errors	The number of iptables-save errors.
felix_resync_state	Current datastore state.
felix_resyncs_started	The number of times Felix has started resyncing with the datastore.
felix_route_table_list_seconds	Time taken to list all the interfaces during a resync.
felix_route_table_per_iface_sync_ seconds	Time taken to sync each interface

The Prometheus Operator is used by TSEE for setting up a Prometheus server along with an AlertManager. The following section talks about Prometheus monitoring used by TSEE.

The Prometheus Operator deploys three CustomResources which help monitor the TSEE nodes based on the above metrics.

- Prometheus
- ServiceMonitor
- AlertManager

The Prometheus CustomResource (CRD) defines a Prometheus setup to be run on the cluster providing it the desired state. The Prometheus instance set up by the custom resource deploys a statefulset in the same namespace. The CRD defines which service monitors should be selected for scraping metrics based on label selectors.

The ServiceMonitor CRD defines the service(s) which are used to scrape for metrics by the Prometheus Pods. This selection of services is achieved by means of a label selector. The ServiceMonitor configures Prometheus to monitor the Pod(s) selected by the service(s). The ServiceMonitor should reside in the same namespace as the Prometheus resource, but it allows service selection across namespaces. This is achieved by means of a namespace selector.

The AlertManager CRD allows defining an AlertManager cluster to be run as part of the cluster.

The AlertManager is used to manage alerts generated by Prometheus. Prometheus' configuration allows inclusion of rules which contain alerting specifics. When an alerting rule is fired, it alerts all the AlertManagers in the cluster. The AlertManager acts on the fired rule and can perform the following operations.

- Deduplicate alerts.
- Silence alerts
- Route and send grouped notifications via providers such as PagerDuty, OpsGenie etc.

CONNECTIVITY AND ROUTING

In order to provide connectivity across workload endpoints, the calico/node Pod must gather information about the name of the node (host) on which it runs.

When the calico/node Pod is scheduled on the host, it gathers this information and creates a Node resource object. The verification, if the Node object exists, is done based on the hostname of the node. BGPConfiguration and FelixConfiguration provide node based overrides, which are associated based on the name of the node as well.

If there is a workload that does not have network connectivity, the likely case would be that the node name for the workload endpoint does not match with the node registered with Calico.

In that case, query the workload endpoints for the workload and make sure that the nodename is the same as that from the output of the command calicoctl get nodes.

```
ubuntu@ip-172-31-59-6:~$ calicoctl get wep
WORKLOAD               NODE                NETWORKS           INTERFACE
nginx-8586cf59-2gkg9   ip-172-31-49-195    192.168.88.20/32   caliebdbc8fda6b
nginx-8586cf59-c4qsc   ip-172-31-53-121    192.168.91.17/32   cali58fe95eecda

ubuntu@ip-172-31-59-6:~$ calicoctl get nodes
NAME
ip-172-31-49-195
ip-172-31-51-217
ip-172-31-53-121
ip-172-31-59-6
```

In the case where there are nodename mismatches, the following steps would need to be performed to allow calico Calico to provide network connectivity to the workload.

1. Cordon the node to disallow further workloads to be scheduled on it.
   - kubectl cordon <node-name>
2. Ensure there are no workloads running on the node by draining it. This will ensure the workloads running on the node get scheduled to other nodes and Kubernetes would take care of the same.
   - kubectl drain <node-name> --ignore-daemonsets
3. Set the right hostname on the node. This command must be run on the node itself.
   - sudo hostnamectl set-hostname <new-node-name>
4. Delete the node definition from Calico
   - calicoctl delete node <node-name>
5. Restarting the daemonset calico-node Pod should allow it to pick up the changes to the nodename.
   - kubectl delete pod -n kube-system <calico-node Pod on the node>
6. Uncordon the node which allows scheduling of workloads on it.
   - kubectl uncordon <new-node-name>

If there is network connectivity between the workloads running on the same host but inter-host connectivity fails, the possible cause might be a misconfigured BGPConfiguration, if in a private network, or if using BGP with AWS. If in public cloud (and using their CNI), there is probably an issue with the underlying SDN provided by the cloud provider and their CNI plugin.

If BGP is in use, ensure that the BGP port 179 is open across the nodes within the cluster.

Verify BGP peering by running:

`calicoctl node status`. This command requires superuser permissions to execute.

The output should be the one mentioned below.

```
root@ip-172-31-59-6:/home/ubuntu# calicoctl node status
Calico process is running.

IPv4 BGP status
+---------------+-------------------+-------+------------+-------------+
| PEER ADDRESS  |     PEER TYPE     | STATE |   SINCE    |    INFO     |
+---------------+-------------------+-------+------------+-------------+
| 172.31.49.195 | node-to-node mesh | up    | 2018-04-25 | Established |
| 172.31.51.217 | node-to-node mesh | up    | 2018-04-25 | Established |
| 172.31.53.121 | node-to-node mesh | up    | 2018-04-25 | Established |
+---------------+-------------------+-------+------------+-------------+

IPv6 BGP status
No IPv6 peers found.
```

POLICY ISSUES [COMMERCIAL SOFTWARE CAPABILITY]

Tigera Secure Enterprise Edition (TSEE) adds policy management and monitoring on top of the Calico deployment. It provides a TSEE manager, which is a web interface for creating and managing hierarchical policies.

It also provides a policy query tool, namely calicoq, which allows inspecting CNX policies being enforced on the workloads as well as the Pods which are affected by it.

The calicoq command line interface provides several subcommands to verify that the policies are applied as per the desired effect.

A few of the subcommands are as listed below.

- The ***endpoint*** command shows you the CNX policies and profiles that relate to specified endpoints.
- The ***eval*** command displays the endpoints that a selector selects.
- The ***host*** command displays the policies and profiles that are relevant to all endpoints on a given host.
- The ***policy*** command shows the endpoints that are relevant to a given policy.

The calicoq endpoint subcommand allows searching for CNX policies that relate to a specific endpoint.

The command takes the form as :

`calicoq endpoint <substring>`

The <substring> does a contains match for the endpoints full ID which is formed as <host-name>/<orchestrator>/<workload-name>/<endpoint-name>.

For each endpoint, it displays the following.

- The policies and profiles that apply to that endpoint in the order that they apply.
- The policies and profiles whose rule selectors match that endpoint (that allow or disallow that endpoint as a traffic source or destination).

calicoq policy <policy-name> shows the endpoints that are relevant to the named policy, comprising:

- The endpoints that the policy applies to.
- The endpoints that match the policy's rule selectors.
- A policy that specifies:

  policy selector: role=='database'; rule: allow from role == 'webserver'

  Then the "policy applies to" selector is role == `database` and the "policy's rule selector" is role == 'webserver'.)

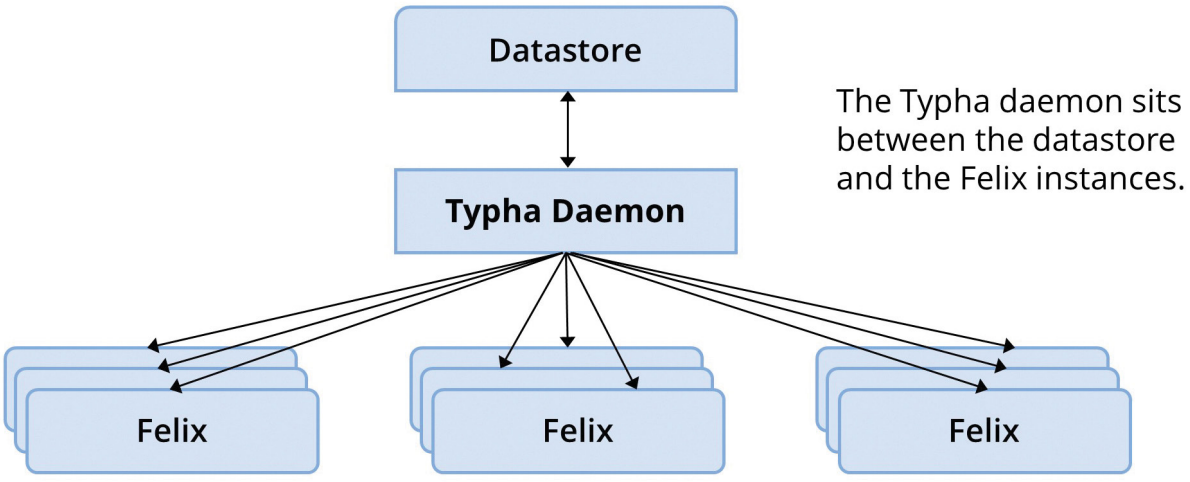
These subcommands help identify if the right policies are applied to the right endpoints

ADVANCED TOPICS

SCALE CONSIDERATIONS

Typha

With a large Kkubernetes cluster with (somewhere in the order of 100+ nodes), there is a lot of load generated on the datastore by the Felix daemon. Each Felix daemon connects to the datastore, to which is the single source of truth. In consideration wWith a cluster of this scale, Calico introduces a daemon which that sits between the datastore and Felix.



This setup has the following benefits.

- Typha helps reduce the load on the datastore drastically. This is especially important and beneficial when Calico is deployed with the Kubernetes datastore driver.
- The Kubernetes datastore generates a lot of updates that might not be relevant to Felix. Typha helps filter these updates, thus helping improve Felix's footprint on the CPU usage. The API server updates would increase dramatically with more nodes or workloads added to it which is filtered out by Typha.

The configuration of Typha is read from the following.

- Environment Variables variables prefixed with TYPA_
- A config file located at /etc/calico/typha.cfg

The environment variables take precedence over the configuration file.

Typha also supports metrics reporting to Prometheus. This allows for a unified monitoring dashboard with all the Calico components.

Following are the full list of parameters that can be set for configuration of Typha.

Environment Variable	Description	Schema
TYPHA_DATASTORETYPE	The datastore that Typha should read endpoints and policy information from. [Default: etcdv3]	etcdv3, Kubernetes
TYPHA_HEALTHENABLED	When enabled, exposes Typha health information via an HTTP endpoint.	Boolean
TYPHA_LOGFILEPATH	The full path to the Typha log. Set to none to disable file logging. [Default: /var/log/calico/typha.log]	string
TYPHA_LOGSEVERITYFILE	The log severity above which logs are sent to the log file. [Default: Info]	Debug, Info, Warning, Error, Fatal
TYPHA_ LOGSEVERITYSCREEN	The log severity above which logs are sent to the stdout. [Default: Info]	Debug, Info, Warning, Error, Fatal
TYPHA_LOGSEVERITYSYS	The log severity above which logs are sent to the syslog. Set to "" for no logging to syslog. [Default: Info]	Debug, Info, Warning, Error, Fatal
TYPHA_PROMETHEUS GOMETRIC SENABLED	Set to false to disable the Go runtime metrics collection, which the Prometheus client does by default. This reduces the number of metrics reported, reducing Prometheus load. [Default: true]	Boolean
TYPHA_ PROMETHEUSMETRICS ENABLED	Set to true to enable the Prometheus metrics server in Typha. [Default: false]	Boolean
TYPHA_PROMETHEUS METRICSPORT	Experimental: TCP port that the Prometheus metrics server should bind to. [Default: 9091]	int
TYPHAPROMETHEUS PROCESS METRICSENABLED	Set to false to disable process metrics collection, which the Prometheus client does by default. This reduces the number of metrics reported, reducing Prometheus load. [Default: true]	Boolean

To deploy Typha on a Kubernetes cluster follow the steps below.

```
apiVersion: v1
kind: Service
metadata:
  name: calico-typha
  namespace: kube-system
  labels:
    k8s-app: calico-typha
spec:
  ports:
    - port: 5473
      protocol: TCP
      targetPort: calico-typha
      name: calico-typha
  selector:
    k8s-app: calico-typha
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: calico-typha
  namespace: kube-system
  labels:
    k8s-app: calico-typha
spec:
  replicas: 3
  revisionHistoryLimit: 2
  template:
    metadata:
      labels:
        k8s-app: calico-typha
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ''
    spec:
      tolerations:
      - key: CriticalAddonsOnly
        operator: Exists
      hostNetwork: true
      containers:
      - image: calico/typha:v0.2.2
        name: calico-typha
```

```
      ports:
      - containerPort: 5473
        name: calico-typha
        protocol: TCP
      env:
        - name: TYPHA_LOGFILEPATH
          value: “none”
        - name: TYPHA_LOGSEVERITYSYS
          value: “none”
        - name: TYPHA_LOGSEVERITYSCREEN
          value: “info”
        - name: TYPHA_PROMETHEUSMETRICSENABLED
          value: “true”
        - name: TYPHA_PROMETHEUSMETRICSPORT
          value: “9093”
        - name: TYPHA_DATASTORETYPE
          value: “kubernetes”
        - name: TYPHA_CONNECTIONREBALANCINGMODE
          value: “kubernetes”
      volumeMounts:
      - mountPath: /etc/calico
        name: etc-calico
        readOnly: true
      resources:
        requests:
          cpu: 1000m
    volumes:
    # Mount in the Calico config directory from the host.
    - name: etc-calico
      hostPath:
        path: /etc/calico
```

Felix needs to be configured to talk to Typha, which can be done by editing the calico/node Pod spec and updating the following environment variable parameter. This allows Felix to discover Typha using the Kubernetes API endpoint.

```
- name: FELIX_TYPHAK8SSERVICENAME
  value: “calico-typha”
```

Note: Felix needs to connect to the same datastore as Typha to read its node configuration first before connecting to Typha.

Route Reflector

The route reflector topology allows setting up BIRD as a centralized point which other BIRD agents communicate with. This largely reduces the number of open connections for each BGP agent.

In a full node-to-node mesh setup, each calico/node opens up TCP connections to every other calico/node. This setup quickly becomes quite complex with a large deployment.

To install a route reflector and configure other BGP nodes to connect to the route reflector, follow the steps below.

Prerequisites

A machine running either Ubuntu or RHEL that is not already being used as a compute host.

1. Install BIRD

The BIRD package is present in the following repository for Ubuntu

sudo add-apt-repository ppa:cz.nic-labs/bird

For CentOS the repository is epel-release

sudo yum install epel-release

2. Configure BIRD

The BIRD configuration requires the AS Number setup for the BGP nodes. Create the BIRD configuration file /etc/bird/bird.conf on the route reflector system and replace <routereflector_ip> with the IPv4 address of the node:

```
# Configure logging
log syslog { debug, trace, info, remote, warning, error, auth, fatal, bug };
log stderr all;

# Override router ID
router id <routereflector_ip>;


# Turn on global debugging of all protocols
debug protocols all;
```

For each compute node in the cluster add the block

```
protocol bgp <node_shortname> {
  description "<node_ip>";
  local as <as_number>;
  neighbor <node_ip> as <as_number>;
  multihop;
  rr client;
  graceful restart;
  import all;
  export all;
}
```

<*node_shortname*> is a unique name for each node (this takes only alphabets as the input and must be unique for each host. The parameter is only used with the route reflector config:

<*node_ip*> with the IPv4 address of the node.

<as_number> with the AS number being used

Restart the BIRD service.

```
systemctl restart bird
systemctl enable bird
```

3. Configure compute nodes.

Disable the node-to-node mesh configuration

```
cat << EOF | calicoctl create -f -
 apiVersion: projectcalico.org/v3
 kind: BGPConfiguration
 metadata:
   name: default
 spec:
   logSeverityScreen: Info
   nodeToNodeMeshEnabled: false
```

Create a global BGP peer for the route reflector

```
cat << EOF | calicoctl create -f -
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global-1
spec:
  peerIP: 192.20.30.40 #< IP Address of the Route reflector node>
  asNumber: 64567  #< AS Number as defined for the deployment>
EOF
```

This will setup BGP peering of the calico/nodes with the route reflector.

HOST PROTECTION

Calico allows securing the network interface of the host/node similar to how it allows securing virtual interfaces for the workload endpoints that run on the host. It allows application of the same network security model, which is applied to the workload endpoints even to a host interface.

This is managed by representing the host interface as a Calico resource, namely, host endpoint.

An example configuration for HostEndpoint.

```
apiVersion: projectcalico.org/v3
kind: HostEndpoint
metadata:
  name: dev-vm1-interface
```

```
  labels:
    type: development
spec:
  interfaceName: eth0
  node: devv1
  expectedIPs:
  - 192.168.0.1
  - 192.168.0.2
    ports:
  - name: web
    port: 9090
    protocol: TCP
  - name: healthcheck
    port: 7071
    protocol: TCP
```

Calico distinguishes workload endpoints by a configurable prefix, the default being cali for the interface. This option can be changed via the FelixConfiguration by setting the `InterfacePrefix` field. Interfaces that start with a value listed in `InterfacePrefix` field are assumed to be workload interfaces, while the others are treated as host interfaces.

Calico blocks all ingress/egress traffic for the workload interfaces by default unless the interface is a known interface or a network policy allows it.

In the case of the host interface, Calico only applies restrictions to the ingress/egress traffic to the interfaces which are configured with the host endpoints resource.

Calico can be used to secure a NAT gateway or a router. Following the model of applying selector based policies depending on the labels configured on the host endpoints allows for security policies being applied dynamically.

Host endpoint policies can be applied to three types of traffic:

1. Traffic that is terminated locally.
2. Traffic that is forwarded between host endpoints.
3. Traffic that is forwarded between a host endpoint and a workload endpoint on the same host.

SERVICE MESH

Istio and Project Calico

Istio is an opensource project as an implementation of the service mesh concept. A service mesh is referred to as a network of microservices which communicate with each other. The mesh of these services communicating with each other can grow fairly quickly as the services get enhanced and developed further.

Istio helps provide:

- Service discovery
- Traffic management and application level load balancing.
- Circuit breaking and fault recovery.
- Request tracing.
- Monitoring.
- Network policy enforcement for enhanced security.
- Telemetry and reporting

Istio service mesh logically has components which provide the control plane and the data plane

- The control plane runs the Istio controller Pod as part of a Kubernetes deployment. It enforces policies and programs the dataplane components to route traffic.
- The data plane component runs a sidecar container within each Pod of an application. This is an envoy proxy container that deals with ingress and egress traffic.

Data Plane Components:

Envoy

Envoy is deployed as a sidecar to the application podPods within kKubernetes. The envoy proxy provides a rich feature set of dynamic service discovery, load balancing, health checks, staged rollouts, etc., which is leveraged by Istio.

Control Plane Components:

Mixer

Mixer is designed to be a platform-independent component, which is responsible for policy enforcement on the service mesh. It also collects telemetry data from the envoy proxy.

Pilot

Pilot is the component that provides service discovery for the Envoy sidecar containers. It is also responsible for programming the sidecar proxy with routing rules.

Istio-Auth

Istio-Auth allows services to communicate with each other using mutual TLS.

Istio operates at the service layer (Layer 7) of the application, which is typically HTTP and the Istio proxy operates at that level. This allows the Istio proxy to make policy decisions based on HTTP headers and route traffic based on them as well as other things. The Calico network policy operates at the Layer 3 of the OSI model, making it much more universal. This enables enforcement of the policy at traffic which does not use HTTP.

With Istio, the policy enforcement is performed by the sidecar container injected into the application Pods which run in the same network namespace as that of the Pod. The Calico network policy is enforced outside the network namespace of the Pods. This ensures that the policy enforcement cannot be bypassed.

Calico policy can extend beyond the service mesh as well allow enforcement on VMs and bare metal which are outside Kubernetes.

One of the very important features of Istio is that it allows the services to communicate over TLS. The sidecar Pods encrypt traffic where the service still believes it is communicating over HTTP. The Istio-Auth component takes away the hassle of managing certificates. The envoy proxy mutually authenticates with services thus creating an identity which can be used to identify outside traffic.

In conjunction with the network policy of Calico, this provides two-layered security. Calico allows specifying policies on egress traffic to provide fine-grained isolation and ensure that a malicious attacker who has gained access to a Pod cannot attack other Pods other than the ones that are visible via egress to him alone.

One area where Project Calico and Istio have a deep integration is in application layer policy. As can be seen in the policy objects in the above network policy section, Calico policy now can match on both service accounts (which map to TLS certificates in Istio) and HTTP actions (such as *PUT, POST,* and *GET* as well as an event the URI targets of those actions. This allows a single policy to encompass not only Layers 3 and 4 matching, but also Layer 5 to 7 behaviors and TLS encryption.

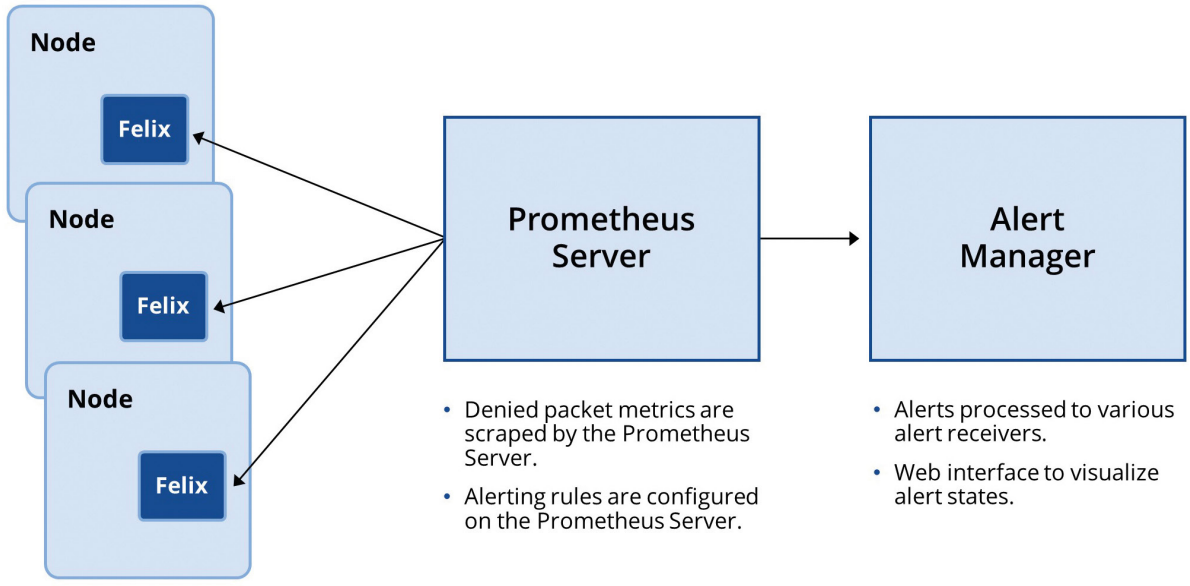
COMPLIANCE

Policy Violation and Alerting

CNX provides the ability to monitor violations of policies configured in the cluster.

Denied traffic is continuously monitored and alerts are configured based on a set of rules and thresholds.

Architecture



Policy violation and reporting follows the workflow defined below.

- A CNX-specific Felix binary running inside calico/node container monitors the host for Denied packets and collects metrics.
- A Prometheus server(s), which gets deployed as part of the CNX manifest, keeps scraping the calico/node.
- An AlertingRule is configured in Prometheus, which triggers an alert based on the denied packet metrics.

- The AlertingRule fires are acted upon by an AlertManager, which is deployed as part of the CNX manifest. The AlertManager receives the alert and forwards alerts to the configured alerting provider. E.g., OpsGenie, Pagerduty.

The PrometheusReporterEnabled and PrometheusReporterPort attributes need to be set on Felix to enable reporting of the denied packets metrics.

The metrics generated are:

- calico_denied_packets - Total number of packets denied by CNX policies.
- calico_denied_bytes - Total number of bytes denied by CNX policies.

The denied packets, as well as denied bytes metrics, allow identification of the source IP address of the traffic as well as the policy in play.

Each one of these metrics is available as a combination of {policy, srcIP}. An HTTP GET request to retrieve metrics from a calico-node container provides the metrics output as below

```
# HELP calico_denied_bytes Total number of bytes denied by calico policies.
# TYPE calico_denied_bytes gauge
calico_denied_bytes{policy="profile/k8s_ns.ns-0/0/deny",srcIP="10.245.13.133"} 300
calico_denied_bytes{policy="profile/k8s_ns.ns-0/0/deny",srcIP="10.245.13.149"} 840

# HELP calico_denied_packets Total number of packets denied by calico policies.
# TYPE calico_denied_packets gauge
calico_denied_packets{policy="profile/k8s_ns.ns-0/0/deny",srcIP="10.245.13.133"} 5
calico_denied_packets{policy="profile/k8s_ns.ns-0/0/deny",srcIP="10.245.13.149"} 14
```

This means that the profile k8s_ns.ns-0 denied five packets (totaling 300 bytes) originating from the IP Address “10.245.13.133” and the same profile denied 14 packets originating from the IP Address “10.245.13.149”.

A metric is generated only when there are packets which were being actively denied at an endpoint via a policy. The time to live for the metric is 60 seconds after the last packet was denied.

Prometheus expires the metric considering it stale if there are no further updates on the metric for some time. This is a configurable option and the time until the metric is considered as stale can be defined with Prometheus.

Policy Auditing

CNX adds a field DropActionOveride to the Felix configuration, which allows defining how the Deny action in a network policyNetworkPolicy rule should be treated.

This is a CNX only option.

the The DropActionOveride field carries the following values:

- Drop
- Accept
- LogAndDrop
- LogAndAccept

Typically, the Drop and the LogAndDrop values are used with Felix, since the Deny action does specify that in the network policyNetworkPolicy Rule. The Accept / LogAndAccept options would be useful while debugging an issue. These options allow traffic even when the action on the Rule is Deny.

When the LogAndDrop or LogAndAccept values are used, there is a syslog entry created for every packet that passes through this rule.

The Felix agent keeps track of these denied packets and publishes the count of denied packets as Prometheus metrics on the port configured by the PrometheusReporterPort setting. The reporting of denied packets is unaffected by the DropActionOverride setting on Felix. Even with the values being set as Accept or LogAndAccept, those packets are still considered as denied packets for metrics reporting.

An example Felix Configuration spec with the DropActionOverride option:

```
apiVersion: projectcalico.org/v3
kind: FelixConfiguration
metadata:
  name: node.myhost
spec:
  defaultEndpointToHostAction: Return
  dropActionOverride: LogAndDrop
```