Lesson Plan

Understanding Calico Native Service Handling

Summary

1. Lab Setup
2. About Calico eBPF
3. NodePort without source IP preservation
4. Enable Calico eBPF
5. Source IP preservation
6. Direct Server Return (DSR)

Lab Setup

This lab assumes you already have the "Yet Another Online Bank" (yaobank) installed, as created in the “Installing the Sample Application” module in Week 1. If you don’t already have it installed then go back and do so now.

For reference, this is architecture of YAOBank:

If you are not already on host1, you can enter host1 by using the multipass shell command.

```
multipass shell host1
```

About Calico eBPF

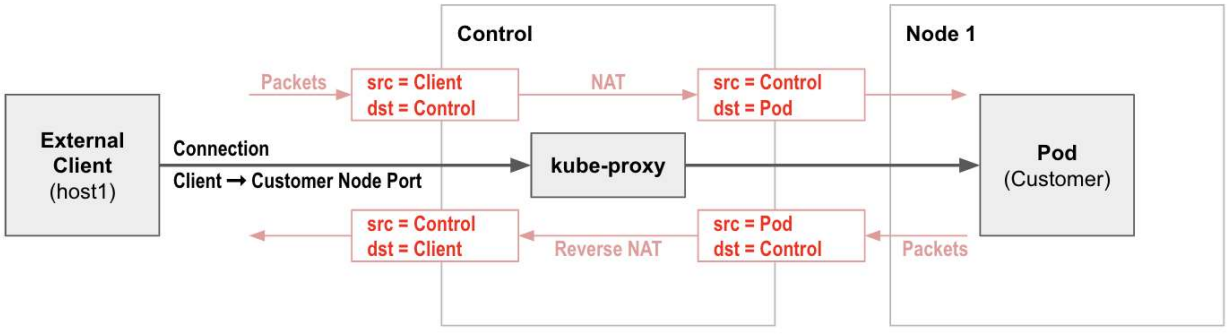
Calico's eBPF dataplane is an alternative to the default standard Linux dataplane (which is iptables based). The eBPF dataplane has a number of advantages:

- It scales to higher throughput.
- It uses less CPU per GBit.
- It has native support for Kubernetes services (without needing kube-proxy) that:
    - Reduces first packet latency for packets to services.
    - Preserves external client source IP addresses all the way to the pod.
    - Supports DSR (Direct Server Return) for more efficient service routing.
    - Uses less CPU than kube-proxy to keep the dataplane in sync.

The eBPF dataplane also has some limitations, which are described in the Enable the eBPF dataplane guide in the Calico documentation.

NodePort without source IP preservation

Before we enable Calico's eBPF based native service handling, let's take a closer look at how kube-proxy handles node ports, and show that the client source IP is not preserved all the way to the pod backing the service. As we covered in the previous module, kube-proxy uses NAT to map the destination IP to the chosen backing pod (DNAT), and map the source IP to the node IP of the ingress node (SNAT). It does the SNAT so that standard Linux networking routes the return packets back to the ingress node so it can reverse the NAT.



For this lab we will need two shells on host1 so we can easily run two commands in parallel.

In a new terminal window, open another shell on host1 (you will have two instances to host1 now):

```
multipass shell host1
```

Once you reach the command prompt of host1, run the following command which will monitor the customer pod logs, showing us new logs as they happen:

```
kubectl logs -n yaobank -l app=customer --follow
```

While the above command is running, access YAO Bank from your other host1 shell, using the node port via the control node:

```
curl 198.19.0.1:30180
```

You should see these logs from the customer pod appear:

```
198.19.0.1 - - [20/Oct/2020 23:58:06] "GET / HTTP/1.1" 200 -
198.19.0.1 - - [21/Oct/2020 00:03:08] "GET / HTTP/1.1" 200 -
198.19.0.1 - - [21/Oct/2020 00:11:31] "GET / HTTP/1.1" 200 -
198.19.0.1 - - [21/Oct/2020 00:30:09] "GET / HTTP/1.1" 200 -
```

Note that the source IP that the pod sees is 198.19.0.1, which is the control node, the node we accessed the node port via. When traffic arrives from outside the cluster, kube-proxy applies SNAT to the traffic on the ingress node; from the pod's point of view, this makes the traffic appear to come from the ingress node (198.19.0.1 in this case). The SNAT is required to make sure the return traffic flows back through the same node so that NodePort DNAT can be undone.

Once you're done with the logs, hit Ctrl-C to stop the kubectl command, but leave the shell up since we will be using it again later in this module.

Enable Calico eBPF

To enable Calico eBPF we need to:

- Configure Calico so it knows how to connect directly to the API server (rather than relying on kube-proxy to help it connect)
- Disable kube-proxy
- Configure Calico to switch to the eBPF dataplane

Configure Calico to connect directly to the API server

In eBPF mode, Calico replaces kube-proxy. This means that Calico needs to be able to connect directly to the API server (just as kube-proxy would normally do). Calico supports a ConfigMap to configure these direct connections for all of its components.

Note: It is important the config map points at a stable address for the API server(s) in your cluster. If you have a HA cluster, the config map should point at the load balancer in front of your API servers so that Calico will be able to connect even if one control plane node goes down. In clusters that use DNS load balancing to reach the API server (such as kops and EKS clusters) you should configure Calico to talk to the corresponding domain name.

In our case, we have a single control node hosting the Kubernetes API service. So we will just configure the control node IP address directly:

```
cat <<EOF | kubectl apply -f -
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: kubernetes-services-endpoint
  namespace: tigera-operator
data:
  KUBERNETES_SERVICE_HOST: "198.19.0.1"
  KUBERNETES_SERVICE_PORT: "6443"
EOF
```

Example output:

```
configmap/kubernetes-services-endpoint created
```

ConfigMaps can take up to 60s to propagate; wait for 60s and then restart the operator, which itself also depends on this config:

```
kubectl delete pod -n tigera-operator -l k8s-app=tigera-operator
```

If you watch the Calico pods, you should see them get recreated with the new configuration:

```
watch kubectl get pods -n calico-system
```

Disable kube-proxy

Calico's eBPF native service handling replaces kube-proxy. You can free up resources from your cluster by disabling and no longer running kube-proxy. When Calico is switched into eBPF mode it will try to clean up kube-proxy's iptables rules if they are present.

Kube-proxy normally runs as a daemonset. So an easy way to stop and remove kube-proxy from every node is to add a nodeSelector to that daemonset which excludes all nodes.

In some environments, kube-proxy is installed/run via a different mechanism. For example, a k3s cluster typically doesn't have a kube-proxy daemonset, and instead is controlled by install time

k3s configuration. In this case, you would still want to get rid of kube-proxy, but if you just wanted to try out Calico eBPF quickly on such a cluster, you can tell Calico to not try to tidy up kube-proxy's iptables rules, and instead allow them to co-exist. (Calico eBPF will still bypass the iptable rules, so they have no effect on the traffic.)

Let's do that now in this cluster by running this command on host1:

```
calicoctl patch felixconfiguration default --patch='{"spec":
{"bpfKubeProxyIptablesCleanupEnabled": false}}'
```

Switch on eBPF mode

You're now ready to switch on eBPF mode. To do so, on host1, use calicoctl to enable the eBPF mode flag:

```
calicoctl patch felixconfiguration default --patch='{"spec": {"bpfEnabled":
true}}'
```

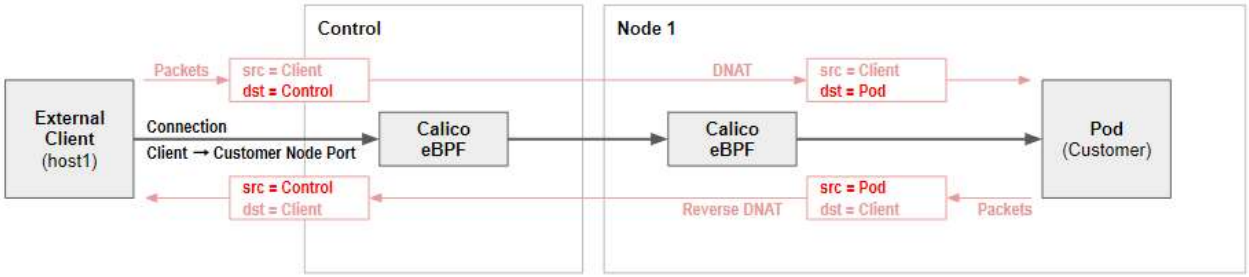
Since enabling eBPF mode can disrupt existing connections, restart YAO Bank's customer and summary pods:

```
kubectl delete pod -n yaobank -l app=customer
kubectl delete pod -n yaobank -l app=summary
```

You're now ready to proceed to the next section to verify connectivity.

Source IP preservation

Now that we've switch to the Calico eBPF data plane, Calico native service handling handles the service without needing kube-proxy. As it is handling the packets with eBPF, rather than the standard Linux networking pipeline, it is able to special case this traffic in a way that allows it to preserve the source IP address, including special handling of return traffic so it still is returned from the original ingress node.



So we can see the effect of source IP preservation, tail the logs of YAO Bank's customer pod again in your second shell window:

```
kubectl logs -n yaobank -l app=customer --follow
```

While the above command is running, access YAO Bank from your other host1 shell, using the node port via the control node:

```
curl 198.19.0.1:30180
```

You should see these logs from the customer pod appear:

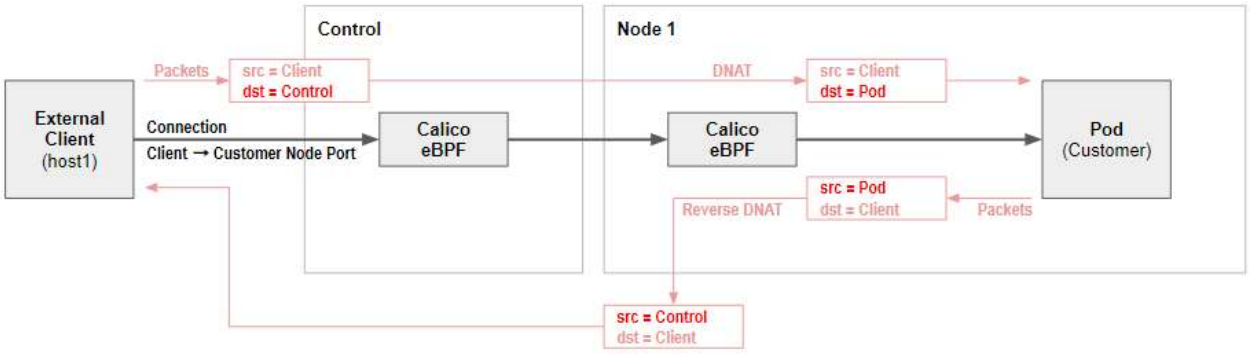
```
198.19.15.254 - - [05/Oct/2020 10:04:37] "GET / HTTP/1.1" 200 -
198.19.15.254 - - [05/Oct/2020 10:04:41] "GET /logout HTTP/1.1" 404 -
198.19.15.254 - - [05/Oct/2020 10:04:45] "GET / HTTP/1.1" 200 -
```

This time the source IP that the pod sees is 198.19.15.254, which is host1, which was the real source of the request, showing that the source IP has been preserved end-to-end.

This is great for making logs and troubleshooting easier to understand, and means you can now write network policies for pods that restrict access to specific external clients if desired.

Enable Direct Server Return (DSR)

Calico's eBPF dataplane also supports DSR (Direct Server Return). DSR allows the node hosting a service backing pod to send return traffic directly to the external client rather than taking the extra hop back via the ingress node (the control node in our example).



DSR requires a network fabric with suitably relaxed RPF (reverse path filtering) enforcement. In particular the network must accept packets from nodes that have a source IP of another node. In addition, any load balancing or NAT that is done outside the cluster must be able to handle the DSR response packets from all nodes.

Snoop traffic without DSR

To show the effect of this, let's snoop the traffic on the control node.

SSH into the control node:

```
ssh control
```

Snoop the traffic associated with the node port:

```
sudo tcpdump -nvi any 'tcp port 30180'
```

Example output:

```
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked v1), capture
size 262144 bytes
```

While the above command is running, access YAO Bank from your other host1 shell, using the node port via the control node:

```
curl 198.19.0.1:30180
```

The traffic will get logged by the tcpdump, for example:

```
    13:59:32.826328 IP (tos 0x0, ttl 64, id 59453, offset 0, flags [DF],
proto TCP (6), length 60)
    198.19.15.254.57064 > 198.19.0.1.30180: Flags [S], cksum 0x520e
(correct), seq 2716257334, win 64240, options [mss 1460,sackOK,TS val
3319780880 ecr 0,nop,wscale 7], length 0
13:59:32.827552 IP (tos 0x0, ttl 62, id 0, offset 0, flags [DF], proto TCP
(6), length 60)
    198.19.0.1.30180 > 198.19.15.254.57064: Flags [S.], cksum 0x287a
(correct), seq 23038493, ack 2716257335, win 65184, options [mss
1370,sackOK,TS val 2052464730 ecr 3319780880,nop,wscale 7], length 0
13:59:32.828602 IP (tos 0x0, ttl 64, id 59454, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.57064 > 198.19.0.1.30180: Flags [.], cksum 0x5394
(correct), ack 1, win 502, options [nop,nop,TS val 3319780883 ecr
2052464730], length 0
13:59:32.828602 IP (tos 0x0, ttl 64, id 59455, offset 0, flags [DF], proto
TCP (6), length 132)
    198.19.15.254.57064 > 198.19.0.1.30180: Flags [P.], cksum 0xa09d
(correct), seq 1:81, ack 1, win 502, options [nop,nop,TS val 3319780883 ecr
2052464730], length 80
13:59:32.829986 IP (tos 0x0, ttl 62, id 33835, offset 0, flags [DF], proto
```

```
TCP (6), length 52)
    198.19.0.1.30180 > 198.19.15.254.57064: Flags [.], cksum 0x533b
(correct), ack 81, win 509, options [nop,nop,TS val 2052464732 ecr
3319780883], length 0
13:59:32.853492 IP (tos 0x0, ttl 62, id 33836, offset 0, flags [DF], proto
TCP (6), length 69)
    198.19.0.1.30180 > 198.19.15.254.57064: Flags [P.], cksum 0x9345
(correct), seq 1:18, ack 81, win 509, options [nop,nop,TS val 2052464756
ecr 3319780883], length 17
13:59:32.853517 IP (tos 0x0, ttl 62, id 33837, offset 0, flags [DF], proto
TCP (6), length 784)
    198.19.0.1.30180 > 198.19.15.254.57064: Flags [FP.], cksum 0x037f
(correct), seq 18:750, ack 81, win 509, options [nop,nop,TS val 2052464756
ecr 3319780883], length 732
13:59:32.855113 IP (tos 0x0, ttl 64, id 59456, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.57064 > 198.19.0.1.30180: Flags [.], cksum 0x52ff
(correct), ack 18, win 502, options [nop,nop,TS val 3319780909 ecr
2052464756], length 0
13:59:32.855323 IP (tos 0x0, ttl 64, id 59457, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.57064 > 198.19.0.1.30180: Flags [.], cksum 0x5023
(correct), ack 751, win 501, options [nop,nop,TS val 3319780909 ecr
2052464756], length 0
13:59:32.857380 IP (tos 0x0, ttl 64, id 59458, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.57064 > 198.19.0.1.30180: Flags [F.], cksum 0x5021
(correct), seq 81, ack 751, win 501, options [nop,nop,TS val 3319780910 ecr
2052464756], length 0
13:59:32.860678 IP (tos 0x0, ttl 62, id 0, offset 0, flags [DF], proto TCP
(6), length 52)
    198.19.0.1.30180 > 198.19.15.254.57064: Flags [.], cksum 0x5012
(correct), ack 82, win 509, options [nop,nop,TS val 2052464763 ecr
3319780910], length 0
```

You can see there is traffic flowing via the node port in both directions. In this example:

- Traffic from host1 to the node port: `198.19.15.254.57064 > 198.19.0.1.30180`
- Traffic from the node port to host1: `198.19.0.1.30180 > 198.19.15.254.57064`

Leave the tcpdump command running and we'll see what difference turning on DSR makes.

Switch on DSR

Run the following command from host1 to turn on DSR:

```
calicoctl patch felixconfiguration default --patch='{"spec":
{"bpfExternalServiceMode": "DSR"}}'
```

Now access YAO Bank from host1 using the node port via the control node:

```
curl 198.19.0.1:30180
```

The traffic will get logged by the tcpdump, for example:

```
    198.19.15.254.56898 > 198.19.0.1.30180: Flags [S], cksum 0xf4ce
(correct), seq 3085566351, win 64240, options [mss 1460,sackOK,TS val
3319654555 ecr 0,nop,wscale 7], length 0
13:57:26.502803 IP (tos 0x0, ttl 64, id 29496, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.56898 > 198.19.0.1.30180: Flags [.], cksum 0xfa75
(correct), ack 3427182734, win 502, options [nop,nop,TS val 3319654556 ecr
2052338405], length 0
13:57:26.503215 IP (tos 0x0, ttl 64, id 29497, offset 0, flags [DF], proto
TCP (6), length 132)
    198.19.15.254.56898 > 198.19.0.1.30180: Flags [P.], cksum 0x477e
(correct), seq 0:80, ack 1, win 502, options [nop,nop,TS val 3319654557 ecr
2052338405], length 80
13:57:31.558591 IP (tos 0x0, ttl 64, id 29498, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.56898 > 198.19.0.1.30180: Flags [.], cksum 0xd293
(correct), ack 18, win 502, options [nop,nop,TS val 3319659613 ecr
2052343461], length 0
13:57:31.558592 IP (tos 0x0, ttl 64, id 29499, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.56898 > 198.19.0.1.30180: Flags [.], cksum 0xcfb7
(correct), ack 751, win 501, options [nop,nop,TS val 3319659613 ecr
2052343461], length 0
13:57:31.559193 IP (tos 0x0, ttl 64, id 29500, offset 0, flags [DF], proto
TCP (6), length 52)
    198.19.15.254.56898 > 198.19.0.1.30180: Flags [F.], cksum 0xcfb6
(correct), seq 80, ack 751, win 501, options [nop,nop,TS val 3319659613 ecr
2052343461], length 0
```

You should only see traffic in one direction, from host1 to the node port on the control node. The return traffic is going directly back to the client (host1) from the node hosting the customer pod backing the service (node1).

Exit back to host1

You can now ctrl-c to terminate the tcpdump and then exit from the control back to the host1 command line.

```
exit
```