

Adding Secondary CIDR Range to EKS-Cluster

Jaswanth Kumar Jonnalagadda

Amazon Web Services

October 2019



© 2019 Amazon Web Services, Inc. or its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without

prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited.

All trademarks are the property of their owner

Table of Contents

PreRequisite:	3
CONFIGURE CNI:	6
Configure Custom Networking:.....	6
CREATE CRDS:	8
Launching PODS in secondary CIDR Network:	10

[Amazon Elastic Container Service for Kubernetes \(EKS\)](#) now allows clusters to be created in a Amazon VPC addressed with additional IPv4 CIDR blocks in the 100.64.0.0/10 and 198.19.0.0/16 ranges. This allows customers additional flexibility in configuring the networking for their EKS clusters.

Previously, EKS customers could only create clusters in VPCs that were addressed with [RFC 1918 private IP address ranges](#). This meant customers were often unable to allocate sufficient private IP address space to support the number of Kubernetes pods managed by EKS

Now, customers can create EKS clusters in Amazon VPCs addressed with CIDR blocks in the 100.64.0.0/10 and 198.19.0.0/16 ranges. This gives customers more available IP addresses for their pods managed by Amazon EKS and more flexibility for networking architectures. Additionally, by [adding secondary CIDR blocks to a VPC](#) from the 100.64.0.0/10 and 198.19.0.0/16 ranges, in conjunction with the [CNI Custom Networking feature](#), it is possible for pods to no longer consume any RFC 1918 IP addresses in a VPC.

In this document, we will walk you through the configuration that is needed so that one can launch the Pod networking on top of secondary CIDRs

PreRequisite:

1. With the following commands you add 100.64.0.0/16 to your EKS cluster VPC

```
VPC_ID=$(aws ec2 describe-vpcs --filters Name=tag:Name,Values=<Name of EKS Cluster>* | jq -r '.Vpcs[].VpcId')
```

```
aws ec2 associate-vpc-cidr-block --vpc-id $VPC_ID --cidr-block 100.64.0.0/16
```

Sample Output:

```
{
  "CidrBlockAssociation": {
    "AssociationId": "vpc-cidr-assoc-0fc2518619e001a5d",
    "CidrBlock": "100.64.0.0/16",
    "CidrBlockState": {
      "State": "associating"
    }
  },
  "VpcId": "vpc-0674822c23b03659e"
}
```

2. Next step is to create subnets.

- a. We need to know how many subnets we are consuming, with the command

```
aws ec2 describe-instances --filters "Name=tag: alpha.eksctl.io/cluster-name,Values=<Name of EKS Cluster>*" --query 'Reservations[*].Instances[*].PrivateDnsName,Tags[?Key==`Name`.Value][0],Placement.AvailabilityZone,PrivateIpAddress,PublicIpAddress' --output table
```

Sample Output:

DescribeInstances									
ip-10-3-3-229.us-west-2.compute.internal	EFX-CIDR-TST-eks-system-workload-Node	us-west-2b	10.3.3.229	54.213.9.45					
ip-10-3-13-241.us-west-2.compute.internal	EFX-CIDR-TST-eks-private-workloads-tenanttwo-Node	us-west-2b	10.3.13.241	None					
ip-10-3-13-214.us-west-2.compute.internal	EFX-CIDR-TST-eks-private-workloads-tenantone-Node	us-west-2b	10.3.13.214	None					
ip-10-3-12-121.us-west-2.compute.internal	EFX-CIDR-TST-eks-private-workloads-tenantone-Node	us-west-2a	10.3.12.121	None					
ip-10-3-12-54.us-west-2.compute.internal	EFX-CIDR-TST-eks-private-workloads-tenanttwo-Node	us-west-2a	10.3.12.54	None					

Here I have 5 instances running on 2 subnets. For the purpose of demo, we will use the same AZ's and create 2 secondary CIDR subnets. One can customize according to the network requirements.

- b. Create two subnets with the following commands

```
export AZ1=us-west-2a
export AZ2=us-west-2b
```

```
CGNAT_SNET1=$(aws ec2 create-subnet --cidr-block 100.64.0.0/19 --vpc-id $VPC_ID --availability-zone $AZ1 | jq -r .Subnet.SubnetId)
```

```
CGNAT_SNET2=$(aws ec2 create-subnet --cidr-block 100.64.32.0/19 --vpc-id $VPC_ID --availability-zone $AZ2 | jq -r .Subnet.SubnetId)
```

We can login to the AWS console and see two new subnets created.

EFX-EKS-CIDR-TST-PublicSubnet04

EFX-EKS-CIDR-TST-PublicSubnet03

subnet-0e5b2b3df787de0f8

subnet-04804a120c6566bea

Subnet: subnet-04804a120c6566bea

Description

Flow Logs

Route Table

Network ACL

Tags

Sharing

Subnet ID

subnet-04804a120c6566bea

🔗

State

available

VPC

vpc-0674822c23b03659e

| EFX-EKS-CIDR-TST-VPC

IPv4 CIDR

100.64.0.0/19

Available IPv4 Addresses

8166

IPv6 CIDR

-

Availability Zone

us-west-2a (usw2-az2)

Route Table

rtb-0e08fa9d736e90ab1

| Public Subnets

Network ACL

acl-0eed99fa674360c32

Default subnet

No

Auto-assign public IPv4 address

No

Auto-assign IPv6 address

No

Owner

682651395775

3. Add necessary tags on created subnets.
 - a. Retrieve the tags by querying current subnets

```
aws ec2 describe-subnets --filters "Name=tag:aws:cloudformation:stack-name,Values=<cf-stack>" "Name=cidr-block,Values=<cidr-block>" --output text
```

Sample Output:

```
TAGS aws:cloudformation:logical-id PublicSubnet01
TAGS Name EFX-EKS-CIDR-TST-PublicSubnet01
TAGS kubernetes.io/cluster/EFX-CIDR-TST shared
TAGS kubernetes.io/role/elb 1
TAGS aws:cloudformation:stack-name EFX-EKS-CIDR-TST
TAGS aws:cloudformation:stack-id arn:aws:cloudformation:us-west-2:682651395775:stack/EFX-EKS-CIDR-TST/03f06380-f862-11e9-a2e2-0a134f485a1c
```

- b. Add these tags through AWS Console or through AWS CLI.

Using AWS CLI, with the commands

```
aws ec2 create-tags --resources $CGNAT_SNET1 --tags Key= Name ,Value= EFX-EKS-CIDR-TST-PublicSubnet03
aws ec2 create-tags --resources $CGNAT_SNET1 --tags Key=kubernetes.io/cluster/<Name of EKS Cluster>*,Value=shared
aws ec2 create-tags --resources $CGNAT_SNET1 --tags Key=kubernetes.io/role/elb,Value=1
```

Similarly add tags to other created subnets

4. Next step, we need to associate the subnets into the route table. For the demo purpose we will be adding the subnets to public route table that had connectivity to internet gateway
 - a. Retrieving the subnet id from existing subnet with the command

```
SNET1=$(aws ec2 describe-subnets --filters "Name=tag:aws:cloudformation:stack-name,Values=EFX-EKS-CIDR-TST" "Name=cidr-block,Values=10.3.2.0/24" | jq -r .Subnets[].SubnetId)
```

- b. Retrieve route table id with the command

```
RTASSOC_ID=$(aws ec2 describe-route-tables --filters  
Name=association.subnet-id,Values=$SNET1 | jq -r  
.RouteTables[].RouteTableId)
```

- c. Associate the route table id to the newly created subnets.

```
aws ec2 associate-route-table --route-table-id $RTASSOC_ID --subnet-id  
$CGNAT_SNET1
```

```
aws ec2 associate-route-table --route-table-id $RTASSOC_ID --subnet-id  
$CGNAT_SNET2
```

With this we had created entries for the subnets in public route tables

CONFIGURE CNI:

1. Make sure to use the latest CNI version
 - a. To view the version, with the command

```
kubectl describe daemonset aws-node --namespace kube-system | grep Image |  
cut -d "/" -f 2
```

Sample Output:
amazon-k8s-cni:v1.5.3

- b. If the version is less than 1.3 then update to the latest CNI version with the command

```
kubectl apply -f https://raw.githubusercontent.com/aws/amazon-vpc-cni-  
k8s/master/config/v1.5.3/aws-k8s-cni.yaml
```

- c. Wait for the pods are recycled. We can check the status of pods by using the command

```
kubectl get pods -n kube-system -w
```

Configure Custom Networking:

1. Edit aws-node configmap and add AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG environment variable to the node container spec and set it to true

With the command we can update the aws-node configmap

Add the bolded lines in the config and save the file.

```
spec:
  containers:
  - env:
    - name: AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG
      value: "true"
    - name: AWS_VPC_K8S_CNI_LOGLEVEL
      value: DEBUG
    - name: MY_NODE_NAME
```

2. Terminate worker nodes so that Autoscaling launches newer nodes that come bootstrapped with custom network config

NOTE: Following command will restart the worker nodes

With the following command we will be restarting the worker nodes.

```
INSTANCE_IDS=$(aws ec2 describe-instances --query
'Reservations[*].Instances[*].InstanceId' --filters "Name=tag:alpha.eksctl.io/cluster-
name,Values==<Name of EKS Cluster>" --output text `)
for i in "${INSTANCE_IDS[@]}"
do
    echo "Terminating EC2 instance $i ..."
    aws ec2 terminate-instances --instance-ids $i
done
```

Sample Output:

```
...
Terminating EC2 instance i-09570bf643ec67507 ...
{
  "TerminatingInstances": [
    {
      "InstanceId": "i-09570bf643ec67507",
      "CurrentState": {
        "Code": 32,
        "Name": "shutting-down"
      },
      "PreviousState": {
        "Code": 16,
        "Name": "running"
      }
    }
  ]
}
```

```

    }
  }
]
}
...

```

Restating might take around 20 minutes depending upon the number of instances and the autoscaling groups

CREATE CRDS:

In this phase we will add custom resources to ENIConfig custom resource definition (CRD). CRD's are extensions of Kubernetes API that stores collection of API objects of certain kind. In this case, we will store VPC Subnet and SecurityGroup configuration information in these CRD's so that Worker nodes can access them to configure VPC CNI plugin

EKS cluster when spun up will come up with a default ENIConfig CRD, we can check that with the command

```
kubectl get crd
```

Sample Output:

```

NAME                               CREATED AT
eniconfigs.crd.k8s.amazonaws.com  2019-10-27T03:54:21Z

```

If the ENIConfig is missing, we can install it with the command

```
kubectl apply -f https://raw.githubusercontent.com/aws/amazon-vpc-cni-k8s/master/config/v1.3/aws-k8s-cni.yaml
```

1. Create custom resources for each subnet by replacing the subnet id and security groups

Sample config file

```

apiVersion: crd.k8s.amazonaws.com/v1alpha1
kind: ENIConfig
metadata:
  name: group1-pod-netconfig
spec:
  subnet: $SUBNETID1
  securityGroups:
    - $SECURITYGROUPID1
    - $SECURITYGROUPID2

```


For this demo purpose we will be creating two files for two new subnets we created.

Sample files with actual values

```
apiVersion: crd.k8s.amazonaws.com/v1alpha1
kind: ENIConfig
metadata:
  name: group1-pod-netconfig
spec:
  subnet: subnet-04f960ffc8be6865c
  securityGroups:
    - sg-070d03008bda531ad
    - sg-06e5cab8e5d6f16ef
```

And save the files as group1-pod-netconfig.yaml

2. Apply the created CRD's with the kubectl command

```
Kubectl apply -f group1-pod-netconfig.yaml
Kubectl apply -f group2-pod-netconfig.yaml
```

3. We can view the configs created with the command

```
kubectl get ENIConfig
```

Sample Output:

NAME	AGE
group1-pod-netconfig	9h
group2-pod-netconfig	9h

4. As the last step will annotate the nodes to take advantage of secondary ips we added. We will be annotating the nodes with the command

```
kubectl annotate node <private ip dns of ec2> k8s.amazonaws.com/eniConfig=group1-
pod-netconfig
Once annotated we can see secondary ip's for the annotated nodes as below.
```

Filter by tags and attributes or search by keyword						
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
EFX-CIDR-TST-eks-system-workload-Node	i-0e440a8b45fe050de	m5.large	us-west-2b	running	2/2 checks ...	None
Instance: i-0e440a8b45fe050de (EFX-CIDR-TST-eks-system-workload-Node) Public DNS: ec2-54-213-9-45.us-west-2.compute.amazonaws.com						
Description Status Checks Monitoring Tags						
Instance ID	i-0e440a8b45fe050de		Public DNS (IPv4)	ec2-54-213-9-45.us-west-2.compute.amazonaws.com		
Instance state	running		IPv4 Public IP	54.213.9.45		
Instance type	m5.large		IPv6 IPs	-		
Elastic IPs			Private DNS	ip-10-3-3-229.us-west-2.compute.internal		
Availability zone	us-west-2b		Private IPs	10.3.3.229, 100.64.43.206, 100.64.61.109		
Security groups	instance-50733deb-default-appf-5352, eksctl-EFX-CIDR-TST-nodegroup-eks-system-workload-SG-176EK3LSPHUSA, eksctl-EFX-CIDR-TST-cluster-ClusterSharedNodeSecurityGroup-AF5YUJJPYNPU. view inbound rules. view outbound rules		Secondary private IPs	100.64.32.224, 100.64.46.99, 100.64.35.151, 100.64.63.40, 100.64.40.25, 100.64.38.28, 100.64.56.92, 100.64.35.93, 100.64.34.158, 100.64.56.33, 100.64.45.66, 100.64.39.131, 100.64.48.3, 100.64.62.181, 100.64.58.54, 100.64.50.119, 100.64.60.154, 100.64.33.46		
Scheduled events	No scheduled events		VPC ID	vpc-0674822c23b03659e (EFX-EKS-CIDR-TST-VPC)		
AMI ID	amazon-eks-node-1.14-v20190927 (ami-05d586e6f773f6abf)		Subnet ID	subnet-0d8f3418496e1e08f (EFX-EKS-CIDR-TST-PublicSubnet02)		

With this we can observe the secondary ip range attached to the worker node.

Launching PODS in secondary CIDR Network:

Once everything is setup we can launch pods in the new ip range.

Let's launch a nginx pod and see the ip's the pod is attached with

```
kubectl run nginx --image=nginx
kubectl scale --replicas=2 deployments/nginx
kubectl expose deployment/nginx --type=NodePort --port 80
```

With this we deployed a nginx pod with 2 replicas.

View the ip's attached to the pods with the command:

```
kubectl get pods -o wide
```

Sample Output:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
NOMINATED NODE						
nginx-64f497f8fd-k962k	1/1	Running	0	40m	100.64.6.147	ip-192-168-52-113.us-east-2.compute.internal <none>
nginx-64f497f8fd-lkslh	1/1	Running	0	40m	100.64.53.10	ip-192-168-74-125.us-east-2.compute.internal <none>

Here we can observe that new pods are launched in the new CIDR range.
With this demo we can attach secondary IP range to and existing EKS Cluster.