Configurar o serviço MPLS L3VPN no roteador PE usando REST-API (IOS-XE)

Contents

Introduction

Este documento demonstra o uso da programação Python para provisionar um MPLS L3VPN em um roteador PE (Provider Edge) usando a API REST. Este exemplo usa roteadores Cisco CSR1000v (IOS-XE) como roteadores PE.

Contribuição de: Anuradha Perera

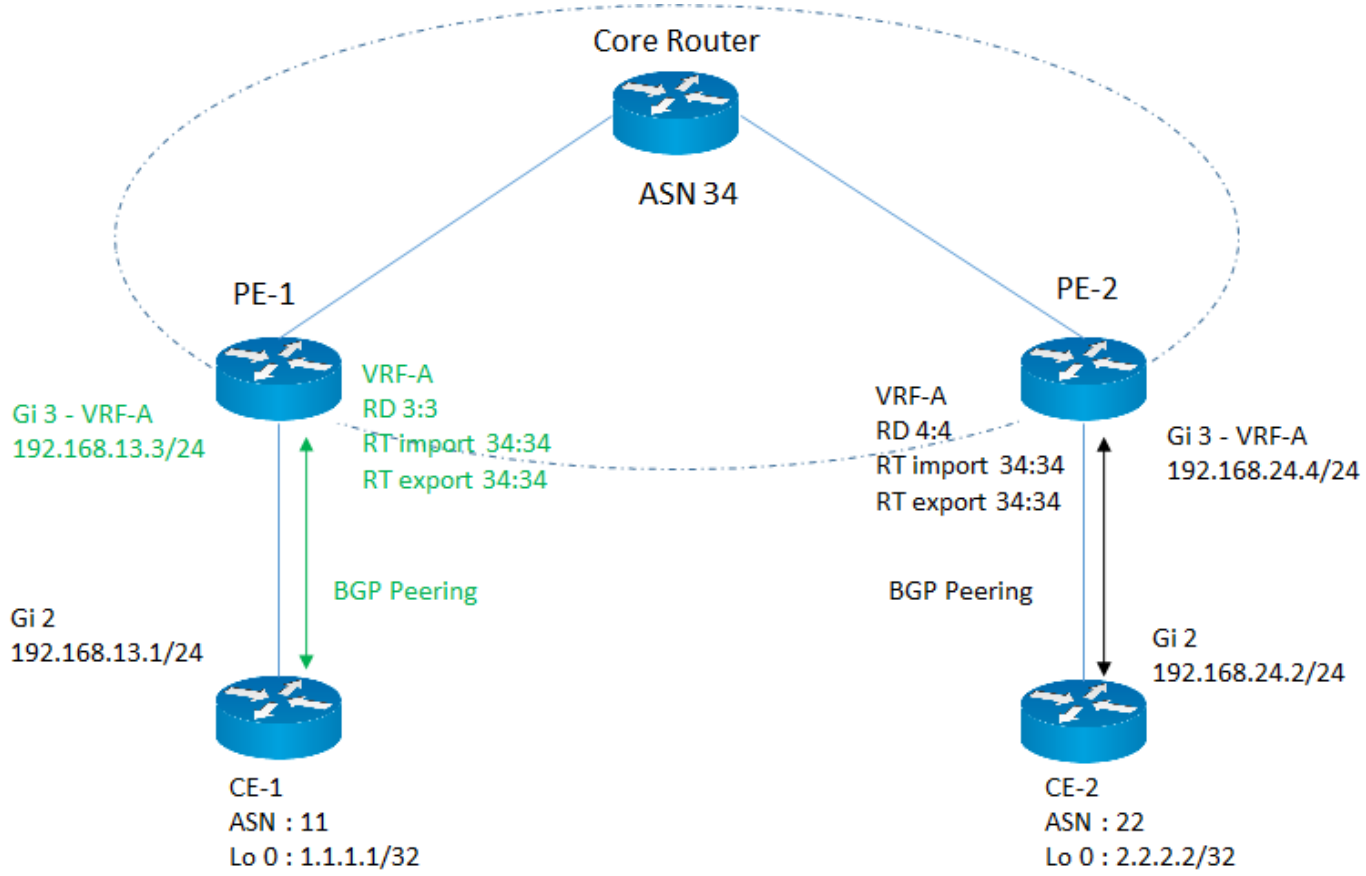
Editado por: Kumar Sridhar

Prerequisites

- Acesso de gerenciamento da API REST aos roteadores CSR1000v (consulte as referências no final deste documento).
- Python (Versão 2.x ou 3.x) e a biblioteca Python "Requests" instalados no computador usado para configurar os roteadores.
- Alguns conhecimentos básicos de programação Python.

Configuração

Diagrama de Rede



Neste exemplo, concentre-se na configuração dos parâmetros de serviço L3VPN MPLS necessários no roteador PE-1, que são destacados na cor rosa.

Procedimento de configuração

A tarefa Configuração é dividida em várias subtarefas e cada subtarefa é implementada em uma função definida pelo usuário. Dessa forma, as funções podem ser reutilizadas quando necessário.

Todas as funções usam a biblioteca "Requests" para acessar APIs REST no roteador e o formato de dados é JSON. Nas solicitações HTTP, o parâmetro "verify" é definido como "False" para ignorar a validação do certificado SSL.

1. Recuperar token-id

Antes de prosseguir com qualquer configuração em um roteador, você precisa ter um token-id válido obtido do roteador. Essa função inicia uma solicitação HTTP para autenticar e obter uma ID de token para que possa invocar outras APIs usando esse token. A resposta desta solicitação inclui um token-id.

```
#----------------------------------

def getToken (ip, port, username, password):

solicitações de importação

importar base64

 url = "https://" + ip + ":"+ port + "/api/v1/auth/token-services"

 cabeçalhos = {

   "tipo de conteúdo": "aplicação/json",

   'authorization': "Basic " + base64.b64encode((username + ":" + password).encode('UTF-8')).decode('ascii')),
```

```
    'cache-control': "no-cache"
  }
  response = requests.request("POST", url, cabeçalhos=cabeçalhos, verify=False )
if response.status_code == 200:
  return response.json()['token-id']
else :
  retorno "failed"
#---------------------------------
```

2. Criar VRF

Essa função criará o VRF no roteador PE com o diferenciador de rota (RD) e destinos de rota de importação/exportação (RT) necessários

```
#---------------------------------
def createVRF (ip, porta, tokenID, vrfName, RD, importRT, exportRT):
solicitações de importação
  url = "https://" + ip + ":"+ porta + "/api/v1/vrf"
  cabeçalhos = {
    'tipo de conteúdo': "aplicativo/json",
    'X-auth-token': tokenID,
    'cache-control': "no-cache"
  }
  dados = {
    'name': vrfName,
    'rd': RD,
    'route-target' : [
        {
            "ação" : "importação",
```

```
            'comunidade' : importRT
        },
        {
            "ação" : "exportação",
            'comunidade' : exportRT
        }
    ]
}
 response = requests.request("POST", url, cabeçalhos=cabeçalhos, json=dados, verify=False )
if response.status_code == 201:
    retorno "bem-sucedido"
else :
    retorno "failed"
#----------------------------------
```

3. Mover a interface para um VRF

Essa função moverá uma determinada interface para um VRF.

```
#----------------------------------
def addInterfacetoVRF (ip, porta, tokenID, vrfName, interfaceName, RD, importRT, exportRT):
solicitações de importação
 url = "https://" + ip + ":"+ porta + "/api/v1/vrf/" + vrfName
 cabeçalhos = {
   'tipo de conteúdo': "aplicativo/json",
   'X-auth-token': tokenID,
   'cache-control': "no-cache"
   }
 dados = {
```

```
  'rd': RD,
  'forwarding': [ interfaceName ],
  'route-target' : [
     {
        'ação' : "importação",
        'comunidade' : importRT
     },
     {
        "ação" : "exportação",
        'comunidade' : exportRT
     }
  ]
}
 response = requests.request("PUT", url, cabeçalhos=cabeçalhos, json=dados, verify=False )


if response.status_code == 204:
  retorno "bem-sucedido"
else :
  retorno "failed"
#---------------------------------
```

4. Atribuir um endereço IP à interface

Essa função atribuirá um endereço IP à interface.

```
#---------------------------------
def assignInterfaceIP (ip, port, tokenID, interfaceName, interfaceIP, interfaceSubnet):
solicitações de importação
 url = "https://" + ip + ":"+ porta + "/api/v1/interfaces/" + interfaceName
 cabeçalhos = {
```

```
  'tipo de conteúdo': "aplicativo/json",
  'X-auth-token': tokenID,
  'cache-control': "no-cache"
  }
 dados = {
  'type': "ethernet",
  'if-name': interfaceName,
  'ip-address': interfaceIP,
  'subnet-mask': interfaceSubnet
  }
 response = requests.request("PUT", url, cabeçalhos=cabeçalhos, json=dados, verify=False )
if response.status_code == 204:
  return "successful"
else :
  return "failed"
#----------------------------------
```

5. Criar bgp sensível a VRF

Isso ativará a família de endereços VRF ipv4.

```
#----------------------------------
def createVrfBGP (ip, porta, tokenID, vrfName, ASN):
solicitações de importação
 url = "https://" + ip + ":"+ porta + "/api/v1/vrf/" + vrfName + "/routing-svc/bgp"
 cabeçalhos = {
  'tipo de conteúdo': "aplicativo/json",
  'X-auth-token': tokenID,
  'cache-control': "no-cache"
  }
```

```
 dados = {
   'routing-protocol-id': ASN
   }
 response = requests.request("POST", url, cabeçalhos=cabeçalhos, json=dados, verify=False )
if response.status_code == 201:
   retorno "bem-sucedido"
else :
   retorno "failed"
#---------------------------------
```

6. Defina o vizinho BGP na família de endereços VRF

Essa função definirá o vizinho BGP na família de endereços VRF IPV4.

```
#---------------------------------
def defineVrfBGPNeighbor (ip, porta, tokenID, vrfName, ASN, neighborIP, remoteAS):
solicitações de importação
 url = "https://" + ip + ":"+ porta + "/api/v1/vrf/" + vrfName + "/routing-svc/bgp/" + ASN +"/neighbors"
 cabeçalhos = {
   'tipo de conteúdo': "aplicativo/json",
   'X-auth-token': tokenID,
   'cache-control': "no-cache"
   }
 dados = {
   'routing-protocol-id': ASN,
   'address': neighborIP,
   'remote-as': remoteAS
   }
 response = requests.request("POST", url, cabeçalhos=cabeçalhos, json=dados, verify=False )
 if response.status_code == 201:
```

```
    retorno "bem-sucedido"
else :
    retorno "failed"
#---------------------------------
```

Descrição e valores dos parâmetros de entrada

```
ip = "10.0.0.1"                          # endereço ip do roteador
porta = "55443"                          # Porta API REST no roteador
nome de usuário = "cisco"                # nome de usuário para login. Deve ser configurado com nível de privilégio 15.
senha = "cisco"                          # senha associada ao nome de usuário
tokenID = <valor retornado> # Token ID obtido do roteador com a função getToken
vrfName = "VRF-A" # nome do VRF
RD = "3:3" # Distinguidor de rota para VRF
importRT = "34:34" # Destino da rota de importação
exportRT = "34:34" #export Route Target
interfaceName = "GigabitEthernet3" # nome da interface voltada para a borda do cliente (CE)
interfaceIP = "192.168.13.3" # Endereço IP da interface voltada para o CE
interfaceSubnet = "255.255.255.0" # sub-rede da interface voltada para o CE
ASN = "34" # BGP AS número do roteador PE
neighborIP = "192.168.13.1" # BGP peering IP do roteador CE
remoteAS = "11" # AS número do roteador CE
```

Em todas as funções acima, APIs dedicadas foram chamadas para cada etapa da configuração. O exemplo abaixo demonstra como passar a CLI do IOS-XE, em geral, no corpo da chamada à API REST. Isso pode ser usado como uma solução alternativa para automatizar se uma API específica não estiver disponível. Nas funções acima, 'content-type' está definido como 'application/json', mas no exemplo abaixo, 'content-type' está definido como 'text/plain', pois está analisando a entrada CLI padrão.

Este exemplo define a descrição da interface para a interface GigabitEthernet3. A configuração pode ser personalizada alterando o parâmetro "cliInput".

```
#---------------------------------
def passCLIInput (ip, porta, tokenID):
```

```
solicitações de importação

 url = "https://" + ip + ":"+ port + "/api/v1/global/running-config"

 cabeçalhos = {

   'content-type': "text/plain",

  'X-auth-token': tokenID,

   'cache-control': "no-cache"

   }

 line1 = "Interface GigabitEthernet 3"

 line2 = "description Customer Facing Interface"

 cliInput = linha1 + "\r\n" + linha2

 response = requests.request("PUT", url, cabeçalhos=cabeçalhos, data=cliInput, verify=False )

 print(response.text)

if response.status_code == 204:

   retorno "bem-sucedido"

else :

   retorno "failed"

#---------------------------------
```

Referências

- Guia de configuração de software do roteador de serviços em nuvem Cisco CSR 1000v Series

https://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/configuration/b_CSR1000v_Configuration_Guide /b_CSR1000v_Configuration_Guide_chapter_01101.html

- Guia de referência de gerenciamento da API REST do Cisco IOS XE

https://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/restapi/restapi.html

Acrônimos utilizados:

MPLS - Multi Protocol Label Switching (MPLS - Comutação de rótulo multiprotocolo)

L3 - Camada 3

VPN - Rede virtual privada

VRF - Encaminhamento de rota virtual

BGP - Protocolo de gateway de borda

REST - Transferência de Estado Representacional

API - Interface de programa de aplicativo

JSON - Notação de Objeto de Script Java

HTTP - Hyper Text Transfer Protocol (Protocolo de Transferência de Hipertexto)

Sobre esta tradução
A Cisco traduziu este documento com a ajuda de tecnologias de tradução automática e humana para oferecer conteúdo de suporte aos seus usuários no seu próprio idioma, independentemente da localização.

Observe que mesmo a melhor tradução automática não será tão precisa quanto as realizadas por um tradutor profissional.

A Cisco Systems, Inc. não se responsabiliza pela precisão destas traduções e recomenda que o documento original em inglês (link fornecido) seja sempre consultado.