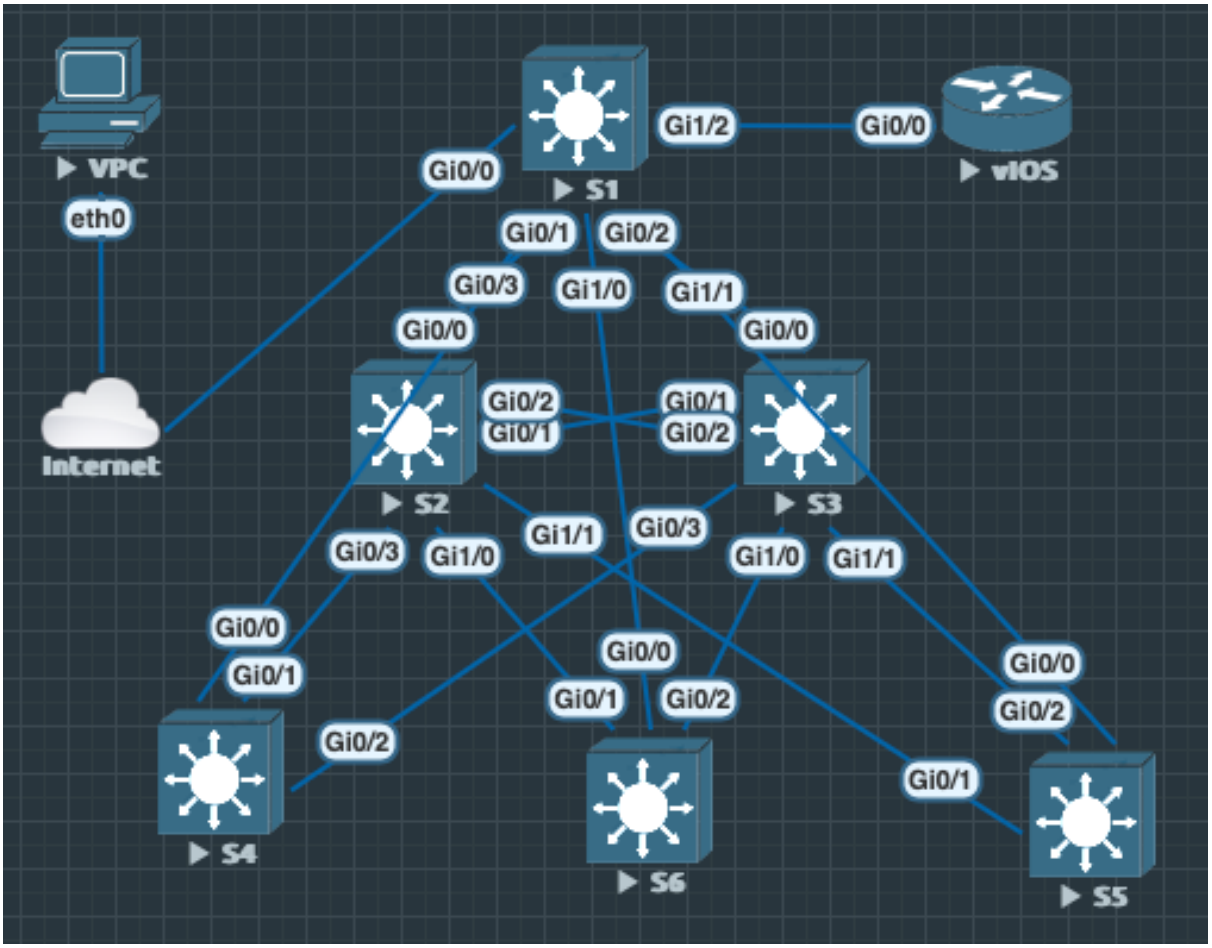


Google Cloud
Network Automation Suite

Ethan Linton

Table of Contents

The Project

This project aims to implement a network automation suite that I can use to configure network technologies without the use of my own compute power. I wanted a cloud implementation that was:

1. Free of charge
2. Provides enough compute power to run virtualized network technologies
3. Provided the ability to run Eve-ng.

This project successfully implements the above via the use of Eve-ng, Cisco CML images, and through the use of the application “Zerotier” to enable easier workflow for network automation from my local PC to the cloud network automation suite.

Technology stack

The technology stack that was used to implement my network automation suite is the following:

- Google cloud
    - Google's offering provides a free trial of $300 credit for 90 days. I made use of this free trial to build a powerful network automation suite.
- Eve-ng
    - Eve-ng was my choice of network emulator software. I preferred the web gui integration compared to the network emulator counterparts such as GNS3.
- Cisco CML Images
    - In order to get access to real Cisco routing and switching images – I utilised my existing Cisco CML subscription and exported the images from CML and imported them into my Eve-ng instance.
- NoMachine
    - The application NoMachine was used to gain access to the VM graphical environment in order to configure the backend of Eve-NG. NoMachine was installed within my VM Instance and my local machine.
- Zerotier
    - I deployed an application called "Zerotier" in order to utilise my local development workflow with my emulated Cisco routing and switching images running within Eve-ng deployed in a Compute engine instance on Google Cloud.
    - This deployment essentially allowed me to deploy my automation scripts from my local machine straight to my chosen network device.

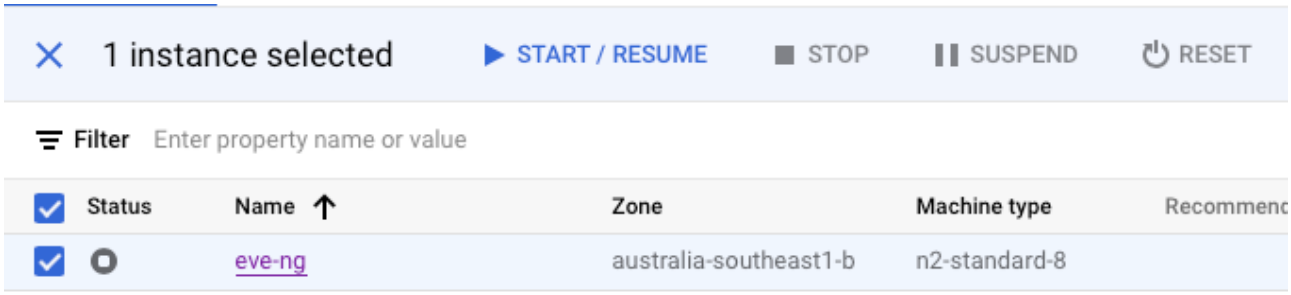
Architecture component overview

This section covers each individual component that makes up my network automation suite in more detail.

Google Cloud

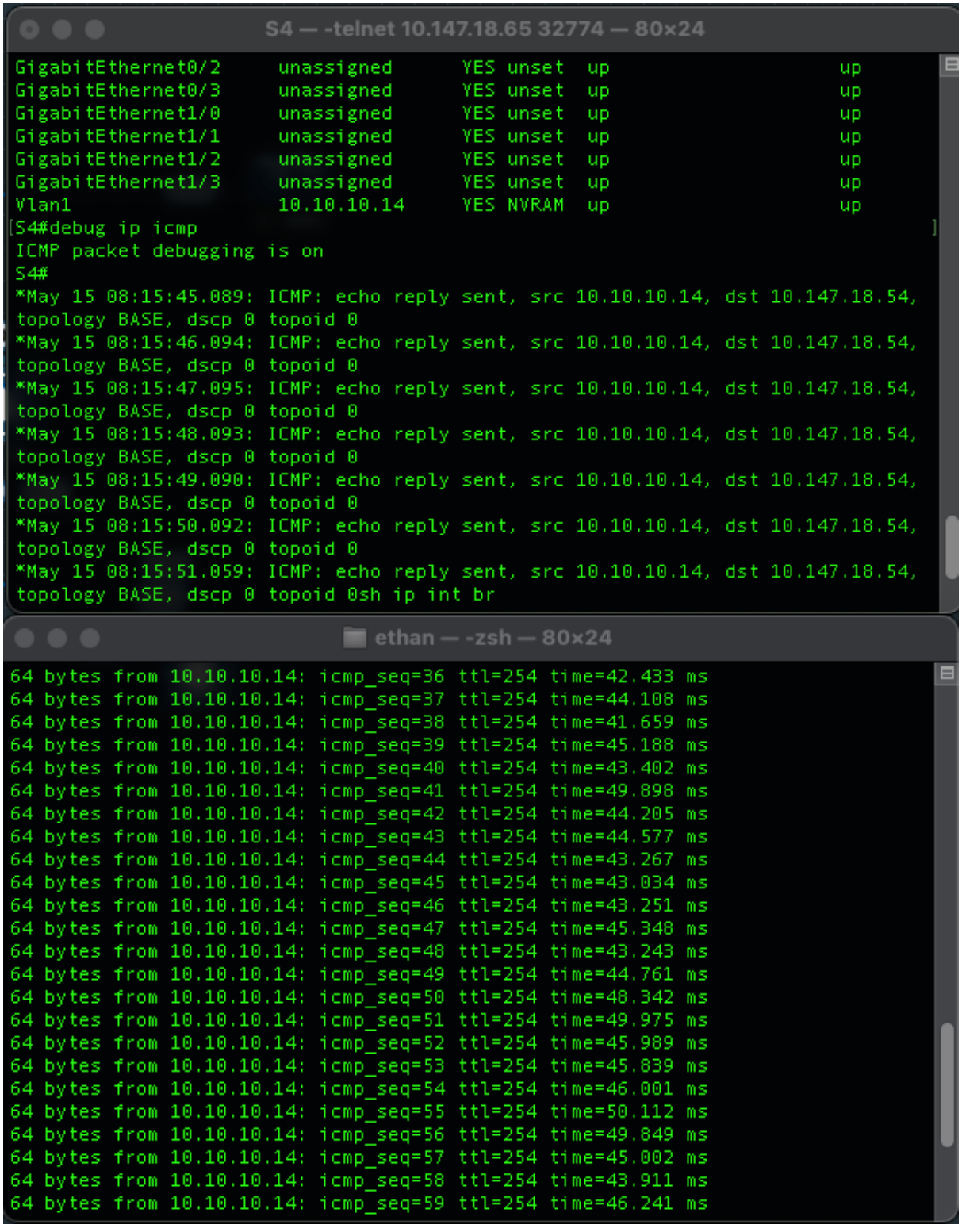
The decision of using Google Cloud as opposed to other cloud providers was due to the other providers having issues with sub-virtualization (running Qemu nodes), and kernel restrictions. With the Google Cloud free trial allowance, I was able to utilise computing power such as the n2-standard-8 machine configuration offering in-which provides 8 vCPU, 4 core and 32 GB memory.

For my Google Cloud compute engine instance, I firstly created a Ubuntu LTS Compute Engine image using the integrated cloud shell. Upon successful compute engine image creation, I created a VM Instance, set the new instance to "n2-standard-8", selected my custom Ubuntu image I created previously, and allowed http and https.



Once the VM instance had started, I utilised SSH to connect into the instance, and installed EVE-NG i.e. *wget -O -https://www.eve-ng.net/focal/install-eve.sh | bash -I*, and performed a restart. Once the VM booted up again – I changed the sudo password, and continued through the standard Eve-NG installation. Upon successful install – I rebooted the VM.

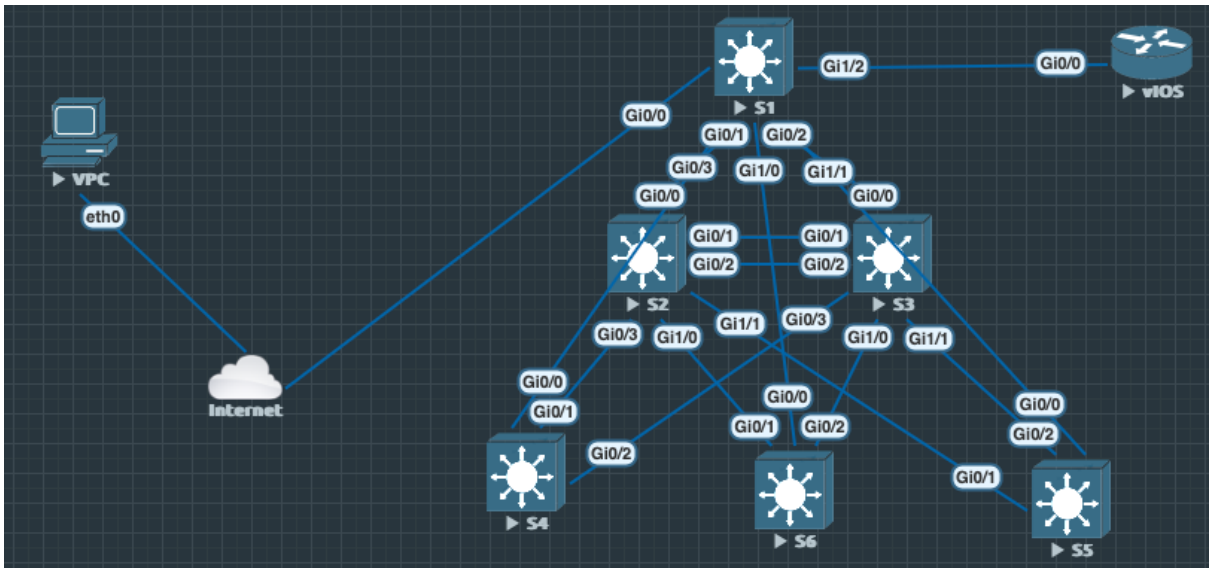
Upon configuring Zerotier (explained below) I navigated to my VPC firewall settings within Google Cloud and disabled all rules relating to SSH, HTTP/S, etc. This is because I now have a secure, and direct connection into the Eve-ng VM Instance and subsequent emulated network topology nodes. I now use IP addresses allocated throughout alternative cloud interfaces, and Zerotier networks to gain direct access. For example, the below image displays direct ping between my local machine and emulated network topology nodes within my Eve-ng instance running via Ubuntu on a VM Instance within Google Cloud.


S4 — -telnet 10.147.18.65 32774 — 80×24
GigabitEthernet0/2 unassigned YES unset up up
GigabitEthernet0/3 unassigned YES unset up up
GigabitEthernet1/0 unassigned YES unset up up
GigabitEthernet1/1 unassigned YES unset up up
GigabitEthernet1/2 unassigned YES unset up up
GigabitEthernet1/3 unassigned YES unset up up
Vlan1 10.10.10.14 YES NVRAM up up
S4#debug ip icmp
ICMP packet debugging is on
S4#
*May 15 08:15:45.089: ICMP: echo reply sent, src 10.10.10.14, dst 10.147.18.54, topology BASE, dscp 0 topoid 0
*May 15 08:15:46.094: ICMP: echo reply sent, src 10.10.10.14, dst 10.147.18.54, topology BASE, dscp 0 topoid 0
*May 15 08:15:47.095: ICMP: echo reply sent, src 10.10.10.14, dst 10.147.18.54, topology BASE, dscp 0 topoid 0
*May 15 08:15:48.093: ICMP: echo reply sent, src 10.10.10.14, dst 10.147.18.54, topology BASE, dscp 0 topoid 0
*May 15 08:15:49.090: ICMP: echo reply sent, src 10.10.10.14, dst 10.147.18.54, topology BASE, dscp 0 topoid 0
*May 15 08:15:50.092: ICMP: echo reply sent, src 10.10.10.14, dst 10.147.18.54, topology BASE, dscp 0 topoid 0
*May 15 08:15:51.059: ICMP: echo reply sent, src 10.10.10.14, dst 10.147.18.54, topology BASE, dscp 0 topoid 0sh ip int br
ethan — -zsh — 80×24
64 bytes from 10.10.10.14: icmp_seq=36 ttl=254 time=42.433 ms
64 bytes from 10.10.10.14: icmp_seq=37 ttl=254 time=44.108 ms
64 bytes from 10.10.10.14: icmp_seq=38 ttl=254 time=41.659 ms
64 bytes from 10.10.10.14: icmp_seq=39 ttl=254 time=45.188 ms
64 bytes from 10.10.10.14: icmp_seq=40 ttl=254 time=43.402 ms
64 bytes from 10.10.10.14: icmp_seq=41 ttl=254 time=49.898 ms
64 bytes from 10.10.10.14: icmp_seq=42 ttl=254 time=44.205 ms
64 bytes from 10.10.10.14: icmp_seq=43 ttl=254 time=44.577 ms
64 bytes from 10.10.10.14: icmp_seq=44 ttl=254 time=43.267 ms
64 bytes from 10.10.10.14: icmp_seq=45 ttl=254 time=43.034 ms
64 bytes from 10.10.10.14: icmp_seq=46 ttl=254 time=43.251 ms
64 bytes from 10.10.10.14: icmp_seq=47 ttl=254 time=45.348 ms
64 bytes from 10.10.10.14: icmp_seq=48 ttl=254 time=43.243 ms
64 bytes from 10.10.10.14: icmp_seq=49 ttl=254 time=44.761 ms
64 bytes from 10.10.10.14: icmp_seq=50 ttl=254 time=48.342 ms
64 bytes from 10.10.10.14: icmp_seq=51 ttl=254 time=49.975 ms
64 bytes from 10.10.10.14: icmp_seq=52 ttl=254 time=45.989 ms
64 bytes from 10.10.10.14: icmp_seq=53 ttl=254 time=45.839 ms
64 bytes from 10.10.10.14: icmp_seq=54 ttl=254 time=46.001 ms
64 bytes from 10.10.10.14: icmp_seq=55 ttl=254 time=50.112 ms
64 bytes from 10.10.10.14: icmp_seq=56 ttl=254 time=49.849 ms
64 bytes from 10.10.10.14: icmp_seq=57 ttl=254 time=45.002 ms
64 bytes from 10.10.10.14: icmp_seq=58 ttl=254 time=43.911 ms
64 bytes from 10.10.10.14: icmp_seq=59 ttl=254 time=46.241 ms

Eve-ng

After successful install of Eve-ng within my Google Cloud VM Instance– I was able to navigate to the web GUI via pasting the public IP of the VM into the search bar. Upon login to the Eve-ng web GUI I changed the password.

After configuration of NoMachine, Zerotier, and the import of CML images (all explained below) – I tested out my new network automation suite. I deployed the following topology within Eve-ng utilising the web GUI that I can now access via my configured Zerotier network. This topology is simple, and consists of a few switches and a router. They are all connected to my cloud internet device thus enabling connection to my local machine via Zerotier. All devices have been preconfigured to allow for remote connection via telnet (I would not recommend telnet in a production network, SSH is recommended as an alternative protocol), and IP addresses have been allocated to all devices within the management subnet to enable connectivity.



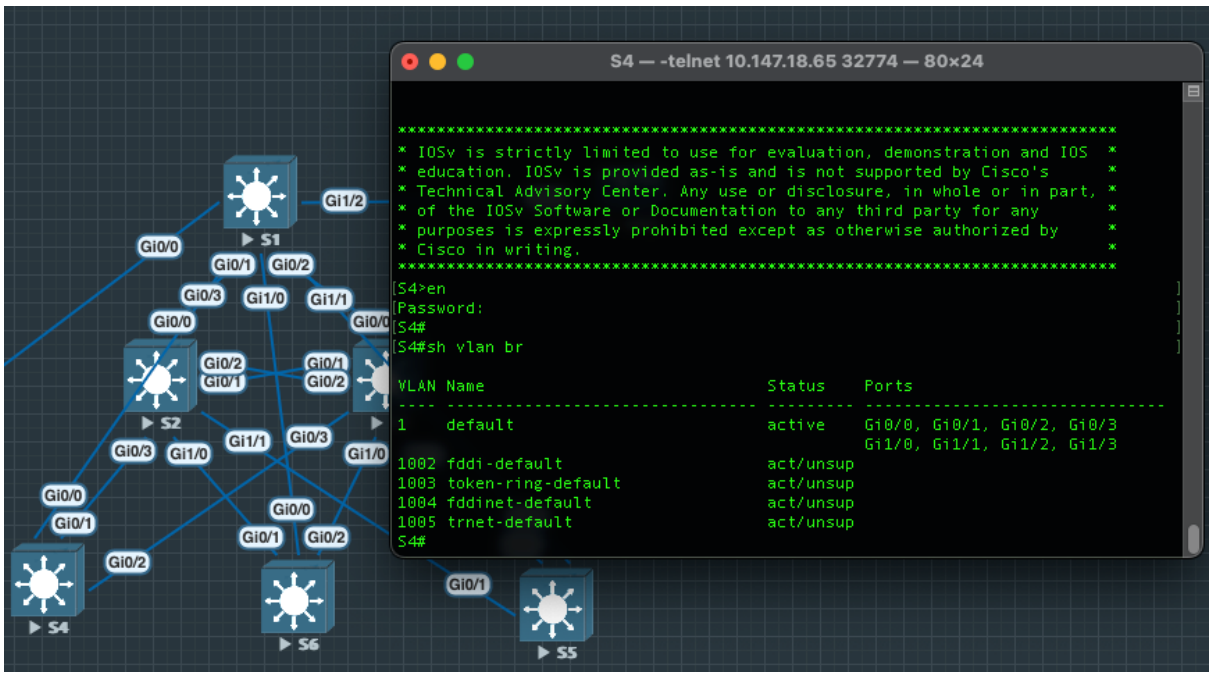
The below script is a simple script that I wrote to test out the functionality of my new network automation suite. The below script simply grabs a list of IP addresses from a .txt file – telnets into the devices thus prompting for a username and password (Telnet as opposed to SSH was implemented as this is a test scenario), configured VLANS 2-10 via a for loop, and then saved the running configuration.

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your telnet username: ")
password = getpass.getpass()

f = open ('myswitches')

for IP in f:
    IP=IP.strip() # Strips all spaces after the IP address
    print ("Configuring Switch" + (IP))
    HOST = IP
```

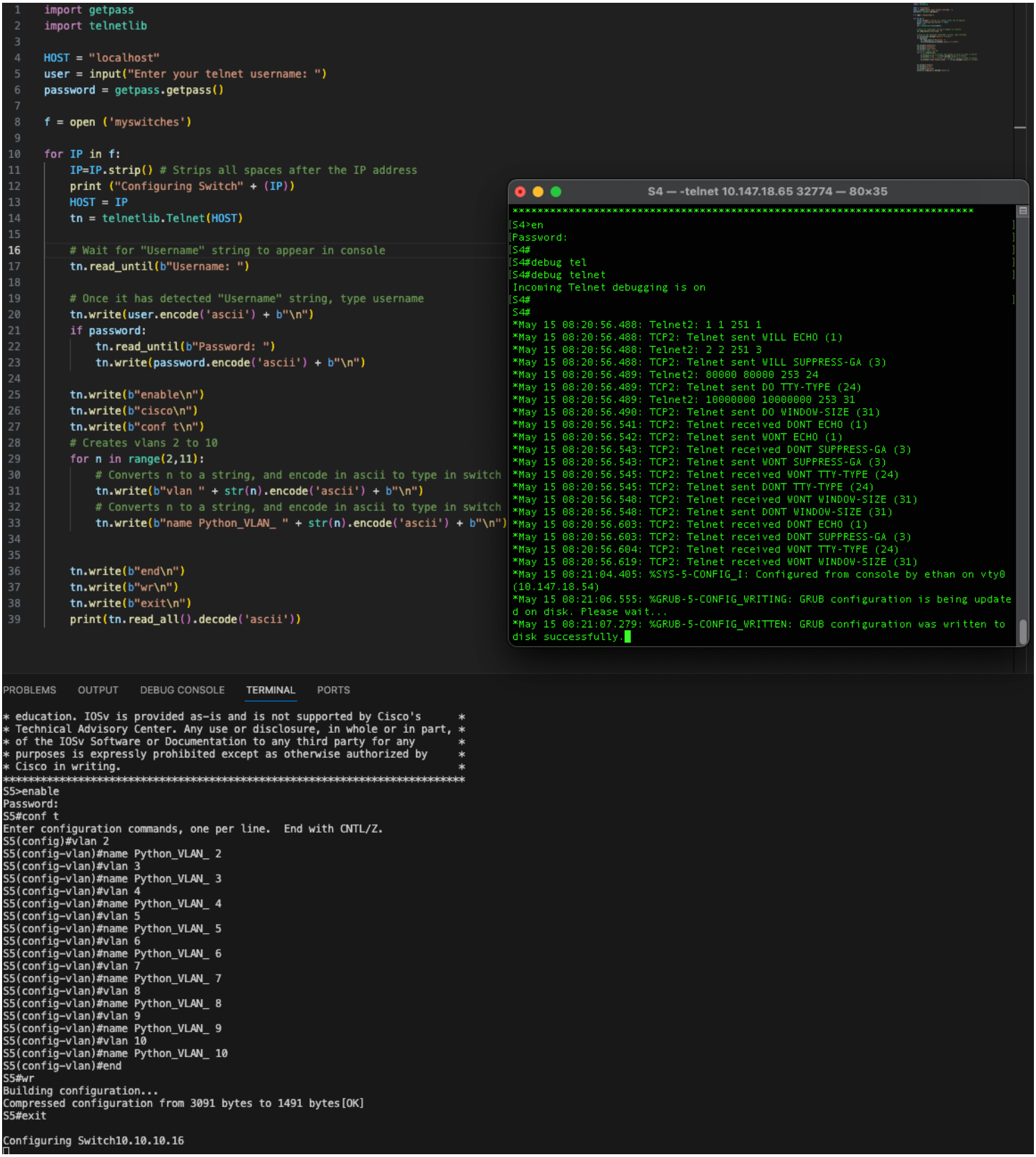
```
tn = telnetlib.Telnet(HOST)

# Wait for "Username" string to appear in console
tn.read_until(b"Username: ")

# Once it has detected "Username" string, type username
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"enable\n")
tn.write(b"cisco\n")
tn.write(b"conf t\n")
# Creates vlans 2 to 10
for n in range(2,11):
    # Converts n to a string, and encode in ascii to type in switch
    tn.write(b"vlan " + str(n).encode('ascii') + b"\n")
    # Converts n to a string, and encode in ascii to type in switch
    tn.write(b"name Python_VLAN_ " + str(n).encode('ascii') + b"\n")


tn.write(b"end\n")
tn.write(b"wr\n")
tn.write(b"exit\n")
print(tn.read_all().decode('ascii'))
```

Please see below for the outputs relating to the test network automation script I created above.

I will be using S4 (Switch 4) as the example switch. The following image displays VLAN configuration on switch 4 before the above network automation script is run.



I ran the above script via Visual Studio Code located on my local machine. The below image displays the outputs from when I am running the script. This image shows the output right after configuration of Switch 4.

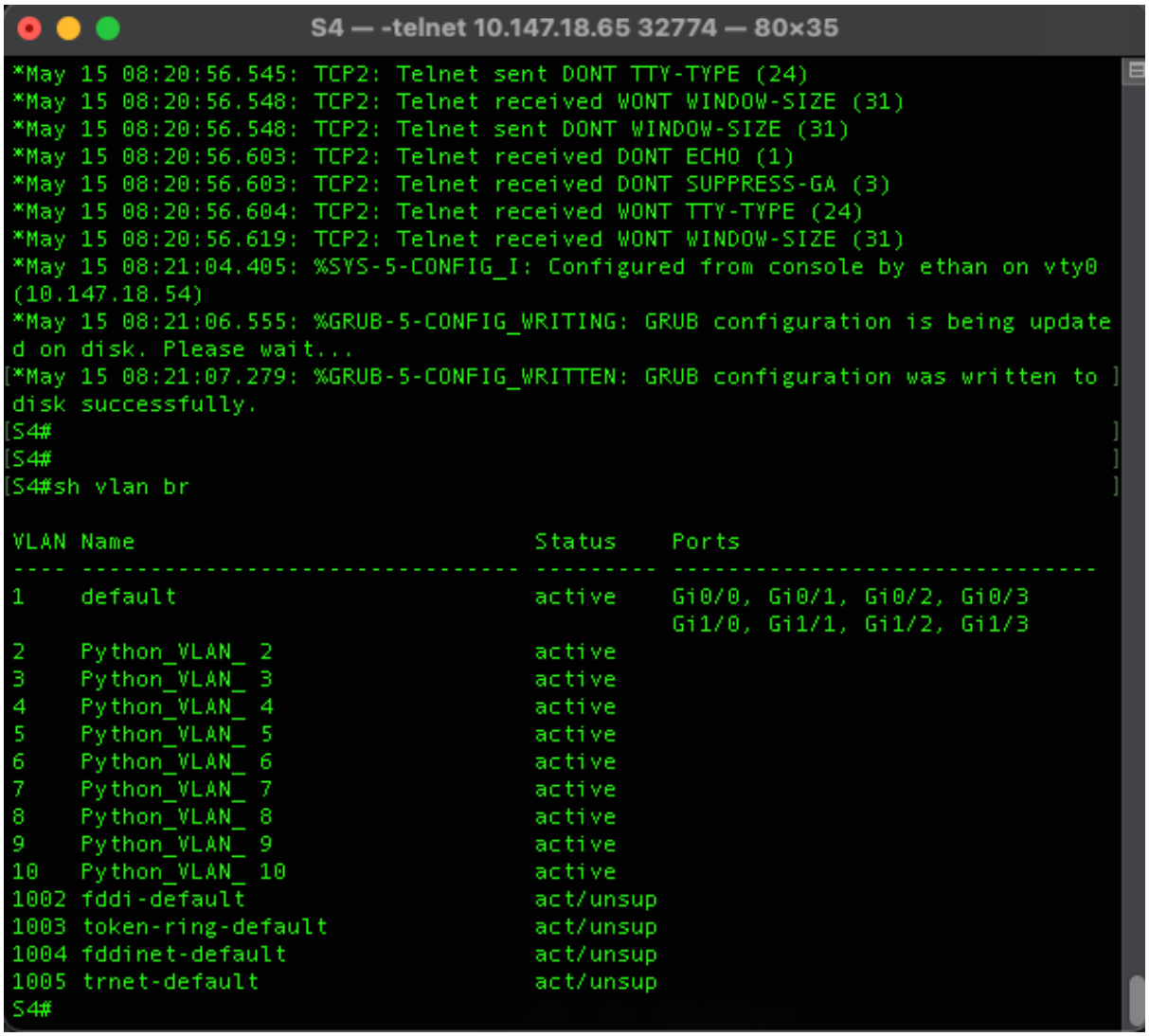


```python
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your telnet username: ")
password = getpass.getpass()

f = open ('myswitches')

for IP in f:
    IP=IP.strip() # Strips all spaces after the IP address
    print ("Configuring Switch" + (IP))
    HOST = IP
    tn = telnetlib.Telnet(HOST)

    # Wait for "Username" string to appear in console
    tn.read_until(b"Username: ")

    # Once it has detected "Username" string, type username
    tn.write(user.encode('ascii') + b"\n")
    if password:
        tn.read_until(b"Password: ")
        tn.write(password.encode('ascii') + b"\n")

    tn.write(b"enable\n")
    tn.write(b"cisco\n")
    tn.write(b"conf t\n")
    # Creates vlans 2 to 10
    for n in range(2,11):
        # Converts n to a string, and encode in ascii to type in switch
        tn.write(b"vlan " + str(n).encode('ascii') + b"\n")
        # Converts n to a string, and encode in ascii to type in switch
        tn.write(b"name Python_VLAN_ " + str(n).encode('ascii') + b"\n")


    tn.write(b"end\n")
    tn.write(b"wr\n")
    tn.write(b"exit\n")
    print(tn.read_all().decode('ascii'))
```

```
S4 — -telnet 10.147.18.65 32774 — 80×35
**************************************************************************
[S4>en
[Password:
[S4#
[S4#debug tel
[S4#debug telnet
Incoming Telnet debugging is on
[S4#
S4#
*May 15 08:20:56.488: Telnet2: 1 1 251 1
*May 15 08:20:56.488: TCP2: Telnet sent WILL ECHO (1)
*May 15 08:20:56.488: Telnet2: 2 2 251 3
*May 15 08:20:56.488: TCP2: Telnet sent WILL SUPPRESS-GA (3)
*May 15 08:20:56.489: Telnet2: 80000 80000 253 24
*May 15 08:20:56.489: TCP2: Telnet sent DO TTY-TYPE (24)
*May 15 08:20:56.489: Telnet2: 10000000 10000000 253 31
*May 15 08:20:56.490: TCP2: Telnet sent DO WINDOW-SIZE (31)
*May 15 08:20:56.541: TCP2: Telnet received DONT ECHO (1)
*May 15 08:20:56.542: TCP2: Telnet sent WONT ECHO (1)
*May 15 08:20:56.543: TCP2: Telnet received DONT SUPPRESS-GA (3)
*May 15 08:20:56.543: TCP2: Telnet sent WONT SUPPRESS-GA (3)
*May 15 08:20:56.545: TCP2: Telnet received WONT TTY-TYPE (24)
*May 15 08:20:56.545: TCP2: Telnet sent DONT TTY-TYPE (24)
*May 15 08:20:56.548: TCP2: Telnet received WONT WINDOW-SIZE (31)
*May 15 08:20:56.548: TCP2: Telnet sent DONT WINDOW-SIZE (31)
*May 15 08:20:56.603: TCP2: Telnet received DONT ECHO (1)
*May 15 08:20:56.603: TCP2: Telnet received DONT SUPPRESS-GA (3)
*May 15 08:20:56.604: TCP2: Telnet received WONT TTY-TYPE (24)
*May 15 08:20:56.619: TCP2: Telnet received WONT WINDOW-SIZE (31)
*May 15 08:21:04.405: %SYS-5-CONFIG_I: Configured from console by ethan on vty0
(10.147.18.54)
*May 15 08:21:06.555: %GRUB-5-CONFIG_WRITING: GRUB configuration is being update
d on disk. Please wait...
*May 15 08:21:07.279: %GRUB-5-CONFIG_WRITTEN: GRUB configuration was written to
disk successfully.
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
S5>enable
Password:
S5#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
S5(config)#vlan 2
S5(config-vlan)#name Python_VLAN_ 2
S5(config-vlan)#vlan 3
S5(config-vlan)#name Python_VLAN_ 3
S5(config-vlan)#vlan 4
S5(config-vlan)#name Python_VLAN_ 4
S5(config-vlan)#vlan 5
S5(config-vlan)#name Python_VLAN_ 5
S5(config-vlan)#vlan 6
S5(config-vlan)#name Python_VLAN_ 6
S5(config-vlan)#vlan 7
S5(config-vlan)#name Python_VLAN_ 7
S5(config-vlan)#vlan 8
S5(config-vlan)#name Python_VLAN_ 8
S5(config-vlan)#vlan 9
S5(config-vlan)#name Python_VLAN_ 9
S5(config-vlan)#vlan 10
S5(config-vlan)#name Python_VLAN_ 10
S5(config-vlan)#end
S5#wr
Building configuration...
Compressed configuration from 3091 bytes to 1491 bytes[OK]
S5#exit

Configuring Switch10.10.10.16
```

The below image displays the new configuration of additional VLANs pushed to the emulated network node within Eve-ng via my python automation script within Visual studio code on my local machine.



This script was a simple example of what can be done via the implementation of my own network automation suite. Future plans include introducing the likes of ansible automation, and automation enabled via API.

NoMachine

Before installing NoMachine onto the VM Instance – I had to install Ubuntu Desktop. In order to install Ubuntu Desktop, I installed Tasksel, and upon opening Tasksel I selected "Ubuntu Desktop" as the package I wish to install. After Ubuntu Desktop package installation – I rebooted the VM instance.

Next, I installed NoMachine by navigating to the NoMachine website on my local machine – copying the direct URL for the NoMachine Linux download link, and typing *wget* + *<URL>* into my VM instance. After downloading the .deb file, I installed the .deb NoMachine file using *sudo apt install./<NoMachineName>.deb.* In order to access the Google Cloud VM

Instance via NoMachine, I had to enable PasswordAuthentication within the sshd_config file in Ubuntu, and create a new user for NoMachine within Ubuntu.

Upon configuration of NoMachine on the VM instance, I installed NoMachine onto my local machine – I added a new connection pointing to my public IP, and utilised the account I created previously to login.

Zerotier

In order to utilise Zerotier I had to make changes to the Ubuntu operating system. I had to turn my VM Instance into a router. Firstly I assigned an IP address within my VM Instance, and restarted networking. Secondly – I enabled IPv4 forwarding to allow Eve-ng VM to forward IP traffic to destinations it doesn't know. Thirdly – I created an outbound NAT rule for traffic leaving the Eve-ng VM Instance.

Within the VM Instance – I navigated to /etc/network/interfaces, and assigned an IP Address to the pnet1 cloud interface. I defined static IP allocation, IP address, and a subnet mask. This new interface will essentially allow for connection of up to 254 management interfaces to its own bridge interface. I restarted networking – *systemctl restart networking*, and verified the IP configured under pnet1 – *ip address*.

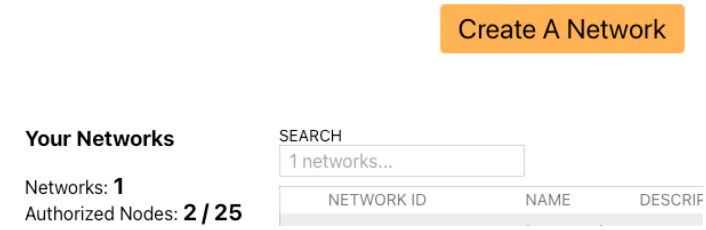
Next I had to turn on IPv4 forwarding. I opened the /etc/sysctl.conf file. This file tells the kernel if it should turn on certain features. Within this file, I commented out *net.ipv4.ip_forward=1.* Now, when it receives a packet that isn't destined for itself, it will look up its own routing table and forward as necessary. In order to load my new changes I entered *sysctl -p /etc/sysctl.conf.*

The creation of an outbound NAT IP tables rule is next. The rule command I entered is the following: *iptables -t nat -A POSTROUTING -s <management subnet just defined> -o pnet0 -j MASQUERADE.* To verify that this rule has been added – *iptables -L -nv -t nat.* This verifies that the newly created rule is within the POSTROUTING chain within my nat table.

Since Ubuntu has now been configured for Zerotier – I will now begin the process of registration and installation of the Zerotier application. The process below outlines the steps carried out to enable Zerotier:

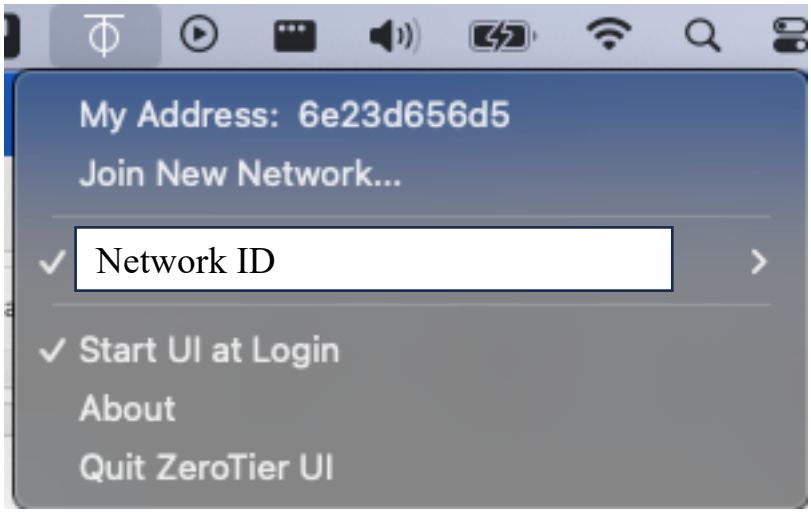
1. Register for an account on Zerotier
2. Create a Zerotier network
3. Download the clients
4. Join clients to the network I created
5. Authorise the client nodes
6. Add a managed route
7. Connect directly to my Eve-ng lab images

I firstly registered for a Zerotier account – after verification I was able to configured my first Zerotier network. Within the Zerotier dashboard, I clicked "Create a network" – this instantly created a Zerotier network without the need for any configuration. *The below image was taken after adding the required nodes to my network.*



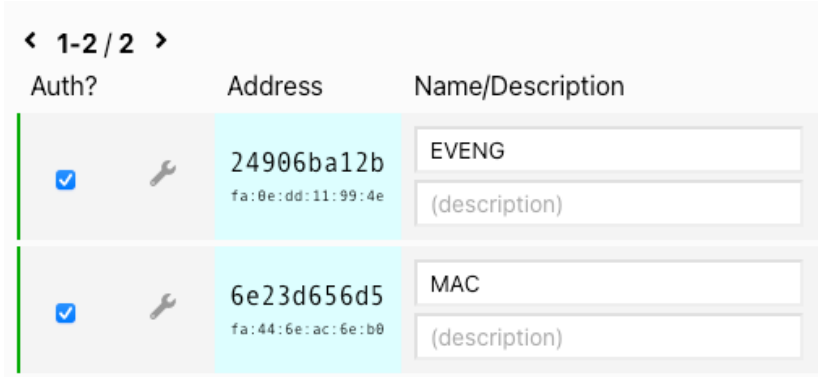
After creating the network, next I downloaded the clients for my endpoints. I downloaded Zerotier onto my Ubuntu VM instance via the bash script provided on the Zerotier website. I also installed Zerotier onto my own local machine.

Next, I joined the Zerotier network I created previously by entering the network ID that was created during network creation.



I joined the Zerotier network within the VM Instance via entering the *zerotier-cli join <network id>*.

Within the Zerotier management dashboard, I authorized both connected via check box.



All of this work so far has enabled me to connect to the Eve-ng server directly. I now wanted to enable local machine to Eve-ng network topology image direct connection to enable seamless network automation workflows. In order to achieve this – I added a new managed route within the Zerotier management dashboard. I configured the management subnet that

was created and assigned to the pnet interface previously as the destination route within my managed route, and configured that destination to be reached via the Zerotier IP address of the Eve-ng VM instance node.

Cisco CML Images

In order to utilise Cisco images that are as close to real Cisco images – I utilised Cisco CML networking images from my current Cisco CML subscription, and imported them into Eve-ng.

I downloaded the Cisco CML .iso file from the software download page on Cisco's website.

Upon download, I opened the ISO file, and extracted the ISO to an alternative location. Within this ISO file are a list of Cisco images relating to routing and switching. I utilised NoMachine to transfer my extracted Cisco images to my Eve-ng VM instance, and placed the required IOSV, and IOSVl2 images into /opt/unetlab/addons/qemu. In accordance to the Eve-ng documentation surrounding the importation of qemu Cisco images – I renamed the images to fit Eve-ng naming scheme.

Upon successfully downloading, and simply moving the images into the Eve-ng VM instance, I was now able to build my test network for network automation.

Conclusion

This project successfully implemented a network automation suite which I can utilise from my local machine. This project has a big emphasis on improving workflows relating to the interaction between my local machine and the end network topology node.

Future work surrounding this implementation includes the following:

- Ansible automation
- API Automation
- Netmiko implementation
    - Enable SSH
- NAPALM implementation
    - Configure BGP routing protocol
    - Deploy large BGP network
    - Make changes based on already existing configuration
- Scaling of the Netmiko implementation.