(Ab)using mesh networks for easy remote support

Amolith

2021-11-01T02:51:00-04:00

Contents

- Nebula . . . 2
  - Getting started . . . 2
  - Creating a Certificate Authority . . . 2
  - Generating lighthouse credentials . . . 3
  - Creating a config file . . . 3
  - Setting the lighthouse up . . . 5
  - Setting individual nodes up . . . 6
- X11vnc . . . 8
- Remmina . . . 8
- SSH . . . 9
- Going further with Nebula . . . 10

One of the things many of us struggle with when setting friends and family up with Linux is remote support. Commercial solutions like RealVNC and RustDesk do exist and function very well, but are often more expensive than we would like for answering the odd "I can't get Facebook open!" support call. I've been on the lookout for suitable alternatives for a couple years but nothing has been satisfying. Because of this, I have held off on setting others up with any Linux distribution, even the particularly user-friendly options such as Linux Mint and elementary OS; if I'm going drop someone in an unfamiliar environment, I want to be able to help with any issue within a couple hours, not days and *certainly* not weeks.

Episode 421 of LINUX Unplugged gave me an awesome idea to use Nebula, a networking tool created by Slack, X11vnc, a very minimal VNC server, and Remmina, a libre remote access tool available in pretty much every Linux distribution, to set up a scalable, secure, and simple setup reminiscent of products like RealVNC.

Nebula

The first part of our stack is Nebula, the tool that creates a network between all of our devices. With traditional VPNs, you have a client with a persistent connection to a central VPN server and other clients can communicate with the first by going through that central server. This works wonderfully in most situations, but there are a lot of latency and bandwidth restrictions that would make remote support an unpleasant experience. Instead of this model, what we want is a *mesh* network, where each client can connect directly to one another *without* going through a central system and slowing things down. This is where Nebula comes in.

In Nebula's terminology, clients are referred to as *nodes* and central servers are referred to as *lighthouses*, so those are the terms I'll use going forward.

Mesh networks are usually only possible when dealing with devices that have static IP addresses. Each node has to know *how* to connect with the other nodes; John can't meet up with Bob when Bob moves every other day without notifying anyone of his new address. This wouldn't be a problem if Bob phoned Jill and told her where he was moving; John would call Jill, Jill would tell him where Bob is, and the two would be able to find each other

With Nebula, nodes are Bob and John and Jill is a lighthouse. Each node connects to a lighthouse and the lighthouse tells the nodes how to connect with one another when they ask. It *facilitates* the P2P connection then *backs out of the way* so the two nodes can communicate directly with each other.

It allows any node to connect with any other node on any network from anywhere in the world, as long as one lighthouse is accessible that knows the connection details for both peers.

Getting started

The *best* resource is the official documentation, but I'll describe the process here as well.

After installing the required packages, make sure you have a VPS with a static IP address to use as a lighthouse. If you want something dirt cheap, I would recommend one of the small plans from BuyVM. I do have a referral link if you want them to kick me a few dollars for your purchase. Hetzner (referral: `ckGrk4J45WdN`) or netcup (referral: `36nc15758387844`) would also be very good options; I've used them all and am very comfortable recommending them.

Creating a Certificate Authority

After picking a device with a static IP address, it needs to be set up as a lighthouse. This is done by first creating a Certificate Authority (CA) that will be used for signing keys and certificates that allow our other devices into the network. The `.key` file produced by the following command is incredibly

sensitive; with it, anyone can authorise a new device and give it access to your network. Store it in a safe, preferably encrypted location.

```
nebula-cert ca -name "nebula.example.com"
```

I'll explain why we used a Fully-Qualified Domain Name (FQDN) as the CA's name in a later section. If you have your own domain, feel free to use that instead; it doesn't really matter what domain is used as long as the format is valid.

Generating lighthouse credentials

Now that we have the CA's `.crt` and `.key` files, we can create and sign keys and certificates for the lighthouse.

```
nebula-cert sign -name "buyvm.lh.nebula.example.com" -ip
    "192.168.100.1/24"
```

Here, we're using a FQDN for the same reason as we did in the CA. You can use whatever naming scheme you like, I just prefer `<vps-host>.lh.nebula...` for my lighthouses. The IP address can be on any of the following private IP ranges, I just happened to use `192.168.100.X` for my network.

IP Range	Number of addresses
10.0.0.0 – 10.255.255.255	16 777 216
172.16.0.0 – 172.31.255.255	10 48 576
192.168.0.0 – 192.168.255.255	65 536

Creating a config file

The next step is creating our lighthouse's config file. The reference config can be found in Nebula's repo. We only need to change a few of the lines for the lighthouse to work properly. If I don't mention a specific section here, I've left the default values.

The section below is where we'll define certificates and keys. `ca.crt` will remain `ca.crt` when we copy it over but I like to leave the node's cert and key files named as they were when generated; this makes it easy to identify nodes by their configs. Once we copy everything over to the server, we'll add the proper paths to the `cert` and `key` fields.

```
pki:
  ca: /etc/nebula/ca.crt
  cert: /etc/nebula/
  key: /etc/nebula/
```

The next section is for identifying and mapping your lighthouses. This needs to be present in *all* of the configs on *all* nodes, otherwise they won't know how to reach the lighthouses and will never actually join the network. Make sure you replace `XX.XX.XX.XX` with whatever your VPS's public IP address is. If you've used a different private network range, those changes need to be reflected here as well.

```
static_host_map:
  "192.168.100.1": ["XX.XX.XX.XX:4242"]
```

Below, we're specifying how the node should behave. It is a lighthouse, it should answer DNS requests, the DNS server should listen on all interfaces on port 53, it sends its IP address to lighthouses every 60 seconds (this option doesn't actually have any effect when `am_lighthouse` is set to `true` though), and this lighthouse should not send reports to other lighthouses. The bit about DNS will be discussed later.

```
lighthouse:
  am_lighthouse: true
  serve_dns: true
  dns:
    host: 0.0.0.0
    port: 53
  interval: 60
  hosts:
```

The next bit is about hole punching, also called *NAT punching*, *NAT busting*, and a few other variations. Make sure you read the comments for better explanations than I'll give here. `punch: true` enables hole punching. I also like to enable `respond` just in case nodes are on particularly troublesome networks; because we're using this as a support system, we have no idea what networks our nodes will actually be connected to. We want to make sure devices are available no matter where they are.

```
punchy:
  punch: true
  respond: true
  delay: 1s
```

`cipher` is a big one. The value *must* be identical on *all* nodes *and* lighthouses. `chachapoly` is more compatible so it's used by default. The devices *I* want to connect to are all x86 Linux, so I can switch to `aes` and benefit from a small performance boost. Unless you know *for sure* that you won't need to work with *anything* else, I recommend leaving it set to `chachapoly`.

```
cipher: chachapoly
```

The last bit I modify is the firewall section. I leave most everything default but *remove* the bits after `port: 443`. I don't *need* the `laptop` and `home` groups (groups will be explained later) to access port `443` on this node, so I shouldn't include the statement. If you have different needs, take a look at the comment explaining how the firewall portion works and make those changes.

Again, I *remove* the following bit from the config.

```
- port: 443
  proto: tcp
  groups:
    - laptop
    - home
```

Setting the lighthouse up

We've got the config, the certificates, and the keys. Now we're ready to actually set it up. After SSHing into the server, grab the latest release of Nebula for your platform, unpack it, make the `nebula` binary executable, then move it to `/usr/local/bin` (or some other location fitting for your platform).

```
wget
    https://github.com/slackhq/nebula/releases/download/vX.X.X/nebula-PLATFORM-ARCH.tar.gz
tar -xvf nebula-*
chmod +x nebula
mv nebula /usr/local/bin/
rm nebula-*
```

Now we need a place to store our config file, keys, and certificates.

```
mkdir /etc/nebula/
```

The next step is copying the config, keys, and certificates to the server. I use `rsync` but you can use whatever you're comfortable with. The following four files need to be uploaded to the server.

- `config.yml`
- `ca.crt`
- `buyvm.lh.nebula.example.com.crt`
- `buyvm.lh.nebula.example.com.key`

With `rsync`, that would look something like this. Make sure `rsync` is also installed on the VPS before attempting to run the commands though; you'll get an error otherwise.

```
rsync -avmzz ca.crt user@example.com:
rsync -avmzz config.yml user@example.com:
rsync -avmzz buyvm.lh.* user@example.com:
```

SSH back into the server and move everything to `/etc/nebula/`.

```
mv ca.crt /etc/nebula/
mv config.yml /etc/nebula/
mv buyvm.lh* /etc/nebula/
```

Edit the config file and ensure the `pki:` section looks something like this, modified to match your hostnames of course.

```
pki:
  ca: /etc/nebula/ca.crt
  cert: /etc/nebula/buyvm.lh.nebula.example.com.crt
  key: /etc/nebula/buyvm.lh.nebula.example.com.key
```

Run the following command to make sure everything works properly.

```
nebula -config /etc/nebula/config.yml
```

The last step is daemonizing Nebula so it runs every time the server boots. If you're on a machine using systemd, dropping the following snippet into `/etc/systemd/system/nebula.service` should be sufficient. If you're using something else, check the the examples directory for more options.

```
[Unit]
Description=nebula
Wants=basic.target
After=basic.target network.target
Before=sshd.service

[Service]
SyslogIdentifier=nebula
ExecReload=/bin/kill -HUP $MAINPID
ExecStart=/usr/local/bin/nebula -config /etc/nebula/config.yml
Restart=always

[Install]
WantedBy=multi-user.target
```

We're almost done!

Setting individual nodes up

This process is almost exactly the same as setting lighthouses up. All you'll need to do is generate a couple of certs and keys then tweak the configs a bit.

The following command creates a new cert/key for USER's node with the IP address `192.168.100.2`. The resulting files would go on the *remote* node not yours. Replace `HOST` and `USER` with fitting values.

```
nebula-cert sign -name "HOST.USER.nebula.example.com" -ip
    "192.168.100.2/24"
```

The following command will create a *similar* cert/key but it will be part of the `support` group. The files resulting from this should go on *your* nodes. With the config we'll create next, nodes in the `support` group will be able to VNC and SSH into other nodes. Your nodes need to be in the `support` group so you'll have access to the others.

```
nebula-cert sign -name "HOST.USER.nebula.example.com" -ip
    "192.168.100.2/24" -groups "support"
```

On to the config now. This tells the node that it is *not* a lighthouse, it should *not* resolve DNS requests, it *should* ping the lighthouses and tell them its IP address every 60 seconds, and the node at `192.168.100.1` is one of the lighthouses it should report to and query from. If you have more than one lighthouse, add them to the list as well.

```
lighthouse:
  am_lighthouse: false
  #serve_dns: false
  #dns:
  #host: 0.0.0.0
  #port: 53
  interval: 60
  hosts:
    - "192.168.100.1"
```

The other bit that should be modified is the `firewall:` section and this is where the groups we created earlier are important. Review its comments and make sure you understand how it works before proceeding.

We want to allow inbound connections on ports 5900, the standard port for VNC, and 22, the standard for SSH. Additionally, we *only* want to allow connections from nodes in the `support` group. Any *other* nodes should be denied access.

Note that including this section is not necessary on *your* nodes, those in the `support` group. It's only necessary on the remote nodes that you'll be connecting to. As long as the `outbound:` section in the config on *your* node allows any outbound connection, you'll be able to access other nodes.

```
- port: 5900
  proto: tcp
  groups:
    - support

- port: 22
  proto: tcp
  groups:
    - support
```

The certs, key, config, binary, and systemd service should all be copied to the same places on all of these nodes as on the lighthouse.

X11vnc

Alright. The hardest part is finished. Now on to setting `x11vnc` up on the nodes you'll be supporting.

All you should need to do is install `x11vnc` using the package manager your distro ships with, generate a 20 character password with `pwgen -s 20 1`, run the following command, paste the password, wait for `x11vnc` to start up, make sure it's running correctly, press `Ctrl` + `C`, then add the command to the DE's startup applications!

```
x11vnc --loop -usepw -listen <nebula-ip> -display :0
```

`--loop` tells `x11vnc` to restart once you disconnect from the session. `-usepw` is pretty self-explanatory. `-listen <nebula-ip>` is important; it tells `x11vnc` to only listen on the node's Nebula IP address. This prevents randos in a coffee shop from seeing an open VNC port and trying to brute-force the credentials. `-display :0` just defines which X11 server display to connect to.

Some distributions like elementaryOS and those that use KDE and GNOME will surface a dialogue for managing startup applications if you just press the Windows (Super) key and type `startup`. If that doesn't work, you'll have to root around in the settings menus, consult the distribution's documentation, or ask someone else that might know.

After adding it to the startup application, log out and back in to make sure it's running in the background.

Remmina

Now that our network is functioning properly and the VNC server is set up, we need something that connects to the VNC server over the fancy mesh network. Enter Remmina. This one goes on *your* nodes.

Remmina is a multi-protocol remote access tool available in pretty much ever distribution's package archive as `remmina`. Install it, launch it, add a new connection profile in the top left, give the profile a friendly name (I like to use the name of the person I'll be supporting), assign it to a group, such as `Family` or `Friends`, set the Protocol to `Remmina VNC Plugin`, enter the node's Nebula IP address in the Server field, then enter their username and the 20 character password you generated earlier. I recommend setting the quality to Poor, but Nebula is generally performant enough that any of the options are suitable. I just don't want to have to disconnect and reconnect with a lower quality if the other person happens to be on a slow network.

Save and test the connection!

If all goes well and you see the other device's desktop, you're done with the VNC section! Now on to SSH.

SSH

First off, make sure `openssh-server` is installed on the remote node; `openssh-client` would also be good to have, but from what I can tell, it's not strictly necessary. You *will* need `openssh-client` on *your* node, however. If you already have an SSH key, copy it over to `~/.ssh/authorized_keys` on the remote node. If you don't, generate one with `ssh-keygen -t ed25519`. This will create an Ed25519 SSH key pair. Ed25519 keys are shorter and faster than RSA and more secure than ECDSA or DSA. If that means nothing to you, don't worry about it. Just note than this key might not interact well with older SSH servers; you'll know if you need to stick with the default RSA. Otherwise, Ed25519 is the better option. After key generation has finished, copy `~/.ssh/id_ed25519.pub` (note the `.pub` extension) from your node to `~/.ssh/authorized_keys` on the remote node. The file *without* `.pub` is your *private* key. Like the Nebula CA certificate we generated earlier, this is extremely sensitive and should never be shared with anyone else.

Next is configuring SSH to only listen on Nebula's interface; as with `x11vnc`, this prevents randos in a coffee shop from seeing an open SSH port and trying to brute-force their way in. Set the `ListenAddress` option in `/etc/ssh/sshd_config` to the remote node's Nebula IP address. If you want to take security a step further, search for `PasswordAuthentication` and set it to `no`. This means your SSH key is *required* for gaining access via SSH. If you mess up Nebula's firewall rules and accidentally give other Nebula devices access to this machine, they still won't be able to get in unless they have your SSH key. I *personally* recommend disabling password authentication, but it's not absolutely necessary. After making these changes, run `systemctl restart sshd` to apply them.

Now that the SSH server is listening on Nebula's interface, it will actually fail to start when the machine (re)boots. The SSH server starts faster than Nebula does, so it will look for the interface before Nebula has even had a chance to connect. We need to make sure systemd waits for Nebula to start up and connect before it tells SSH to start; run `systemctl edit --full sshd` and add the following line in the `[Unit]` section, above `[Service]`.

```
1 After=nebula.service
```

Even now, there's still a bit of a hiccup. Systemd won't start SSH until Nebula is up and running, which is good. Unfortunately, even after Nebula has started, it still takes a minute to bring the interface up, causing SSH to crash. To fix *this*, add the following line directly below `[Service]`.

```
1 ExecStartPre=/usr/bin/sleep 30
```

If the `sleep` executable is stored in a different location, make sure you use that path instead. You can check by running `which sleep`.

When the SSH *service* starts up, it will now wait an additional 30 seconds before actually starting the SSH *daemon*. It's a bit of a hacky solution but it works™. If you come up with something better, please send it to me and I'll include it in the post! My contact information is at the bottom of this site's home page.

After you've made these changes, run `systemctl daemon-reload` to make sure systemd picks up on the modified service file, then run `systemctl restart sshd`. You should be able to connect to the remote node from your node using the following command.

```
ssh USER@<nebula-ip>
```

If you want to make the command a little simpler so you don't have to remember the IP every time, create `~/.ssh/config` on your node and add these lines to it.

```
Host USER
  Hostname <nebula-ip>
  User USER
```

Now you can just run `ssh USER` to get in. If you duplicate the above block for all of the remote nodes you need to support, you'll only have to remember the person's username to SSH into their machine.

Going further with Nebula

This section explains why we used FQDNs in the certs and why the DNS resolver is enabled on the lighthouse.

Nebula ships with a built-in resolver meant specifically for mapping Nebula node hostnames to their Nebula IP addresses. Running a public DNS resolver is very much discouraged because it can be abused in terrible ways. However, the Nebula resolver mitigates this risk because it *only* answers queries for Nebula nodes. It doesn't forward requests to any other servers nor does it attempt to resolve any domain other than what was defined in its certificate. If you use the example I gave above, that would be `nebula.example.com`; the lighthouse will attempt to resolve any subdomain of `nebula.example.com` but it will just ignore `example.com`, `nebula.duckduckgo.com`, `live.secluded.site`, etc.

Taking advantage of this resolver requires setting it as your secondary resolver on any device you want to be able to resolve hostnames from. If you were to add the lighthouse's IP address as your secondary resolver on your PC, you could enter `host.user.nebula.example.com` in Remmina's server settings *instead of* `192.168.1.2`.

But how you do so is beyond the scope of this post!

If you're up for some *more* shenanigans later on down the line, you could set up a Pi-Hole instance backed by Unbound and configure Nebula as Unbound's secondary resolver. With this setup, you'd get DNS-level ad blocking *and* the ability to resolve Nebula hostname. Pi-Hole would query Unbound for `host.user.nebula.example.com`, Unbound would receive no answer from the root servers because the domain doesn't exist outside of your VPN, Unbound would fall back to Nebula, Nebula would give it an answer, Unbound would cache the answer, tell Pi-Hole, Pi-Hole would cache the answer, tell your device, then your device would cache the answer, and you can now resolve any Nebula host!

Exactly how you do *that* is ***definitely*** beyond the scope of this post :P

If you set any of this up, I would be interested to hear how it goes! As stated earlier, my contact information is at the bottom of the site's home page :)