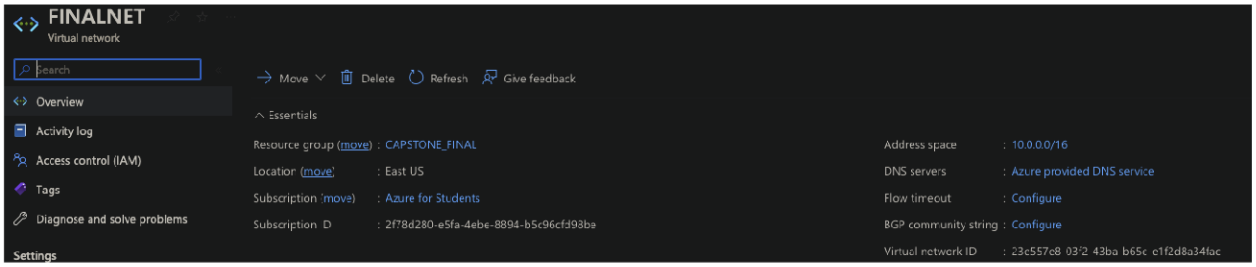
Technical Report of Wireguard, PiHole, and AD Deployment in Azure

Group 42

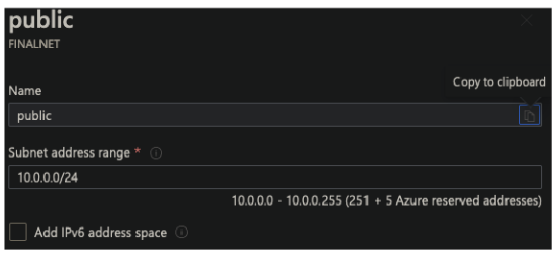
Creating the Azure Cloud Network Infrastructure

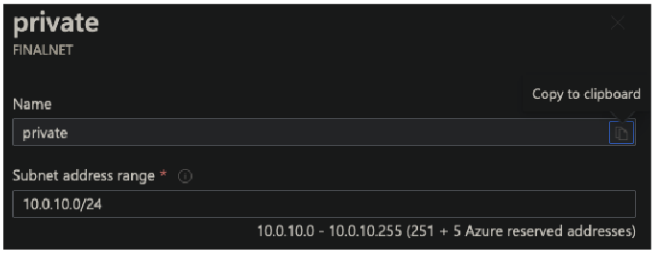
We began by creating a resource group for our project. We called it **Capstone_Final** to differentiate it from other projects we are working on.

We then created a subnet, **FINALNET**, to allow multiple servers to communicate both locally and externally. **FINALNET** has an address space of 10.0.0.0/16 to allow the creation of two subnets for security purposes.

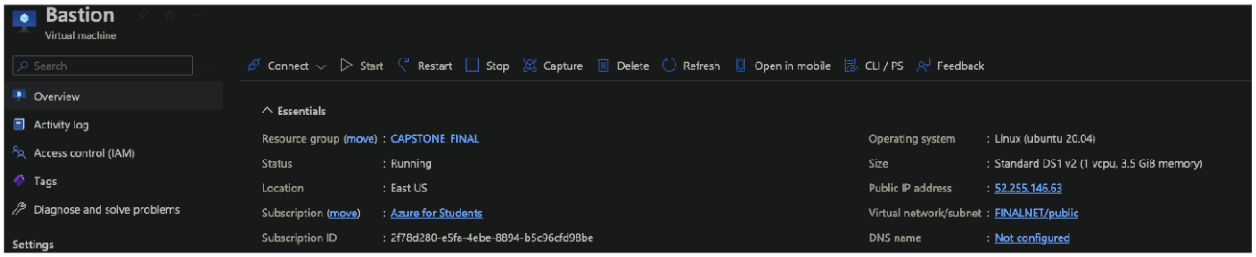


We created two subnets on **FINALNET**, one called **Public** with address space of 10.0.0.0/24 and one called **Private** with address space of 10.0.1.0/24, to keep the information in our database secure.





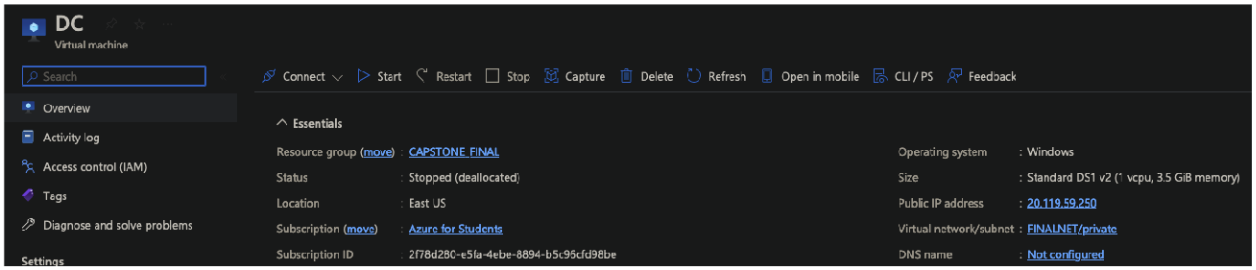
We then provisioned an **Ubuntu 22.04** machine named **Bastion** on the **Public** subnet to serve as our server host for both PiHole and to allow vpn protected access through Wireguard to the **Active Directory** network on the **Private**. We configured it initially to allow ssh access and saved the public key.



We configured the firewall rule on the machines to allow traffic through port 22 and 80 so that we could connect with it and access the internet.

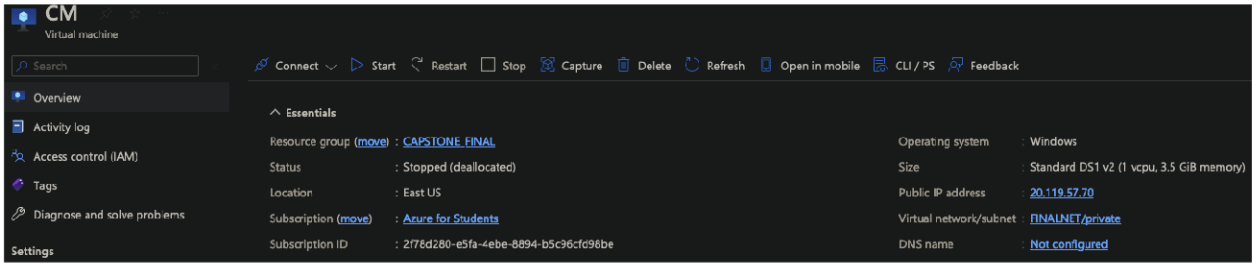
300	SSH	22	TCP	Any	Any	Allow
320	AllowAnyCustom80Inbound	80	Any	Any	Any	Allow

We provisioned a **WindowsServer** machine which we named **DC** running **2019 Datacenter Core** on the **Private** subnet. We opened **port 3389** for **RDP** and created credentials for an administrator account



300	RDP	3389	TCP	Any	Any	Allow

For our client machine we provisioned a machine which we named **CM** running **Windows 10** Pro on the **Private** subnet. We opened **port 3389** for **RDP** and created credentials for an administrator account



At this point the infrastructure setup was complete and we could begin working on the servers themselves.

Setting Up the Active Directory Server

Once we had RDPed in we installed the Active Directory Domain Services (ADDS) via Powershell and made sure to include the management tools.

```
Install-WindowsFeature AD-Domain-Services -IncludeManagementTools
```

With ADDS installed, we created our first domain forest and named it **nerdplatoon.local**.

```
Install-ADDSForest -DomainName nerdplatoon.local
```

We choose a password for the safe mode administrator. If we need to recover the domain when nobody else is able to, we will still have this password. After setting the password, the server rebooted to complete the setup.

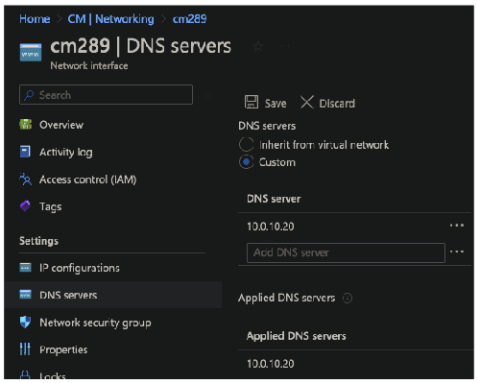
We then installed the DNS Server role with:

Install-WindowsFeature DNS

At that point we have the domain controller set up enough that we can add the workstation and administer the server from there.

Setting Up the Client Machine

We first changed the the DNS setting of **CM in Azure** to make sure it uses **DC** as it's DNS server. Make sure you do this in azure and not in the settings of the machine itself.

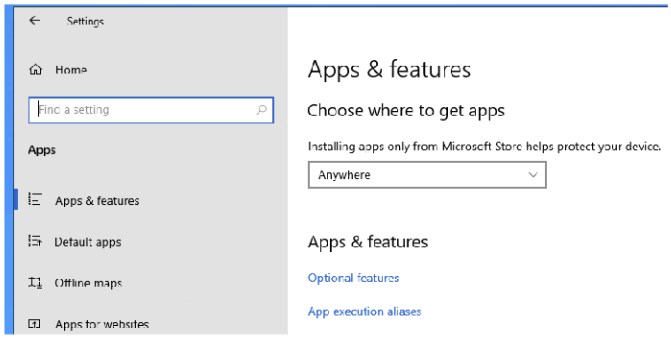


To add the computer to the domain, we used RDP to access the client machine and then ran the following command in Powershell

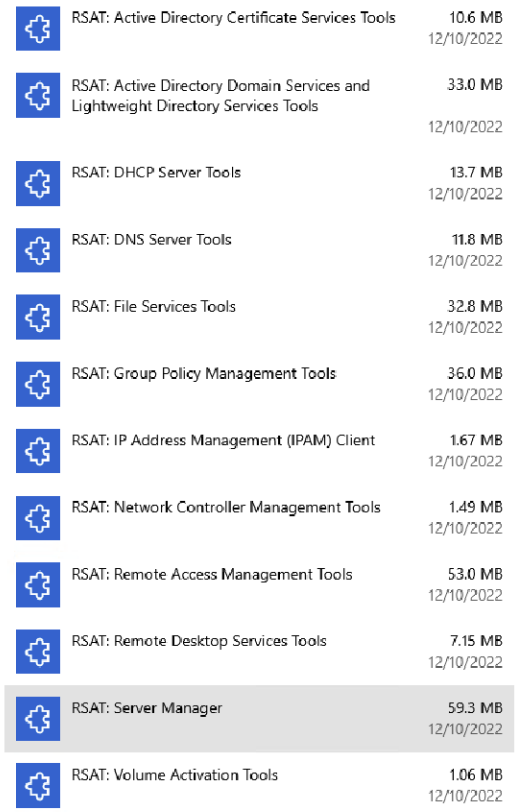
Add-Computer -DomainName **nerdplatoon.local** -Restart

It asked for credentials at this point, and we used the admin credentials for **DC**. At that point the computer restarted and when we connected over RDP we now used the admin credentials for

DC. Once we were in it was time to start setting up the services needed to remotely administer our server. We navigated to Windows Settings, then to the apps section and then clicked optional features.

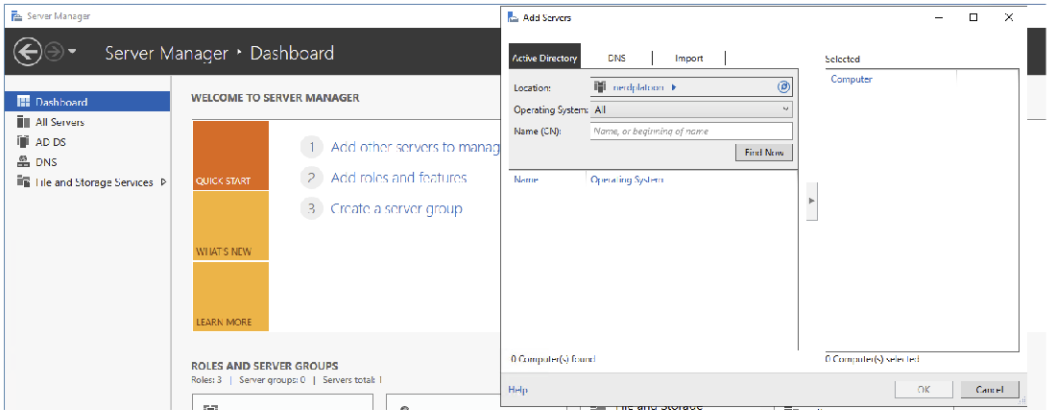


We then installed the following optional features for the server.

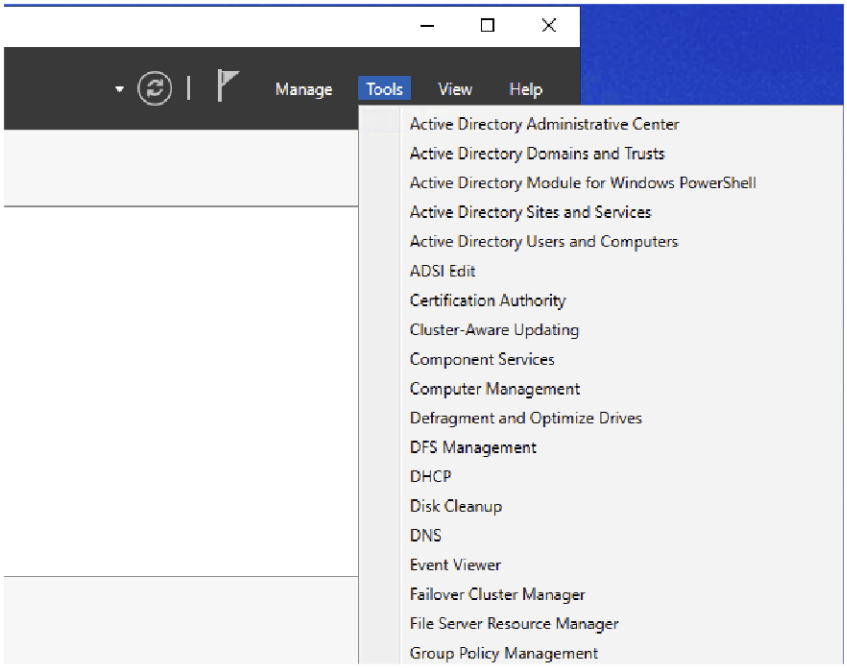


Setting Up the Domain

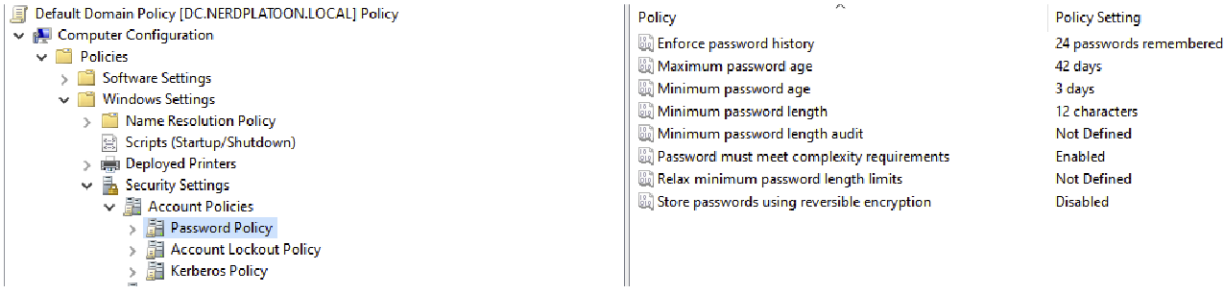
From here on out we used the Server Manager App located in the start menu to set the needed features up. We began by adding the DC server to the server manager with the new server button. We entered DC into the name field and added the server.



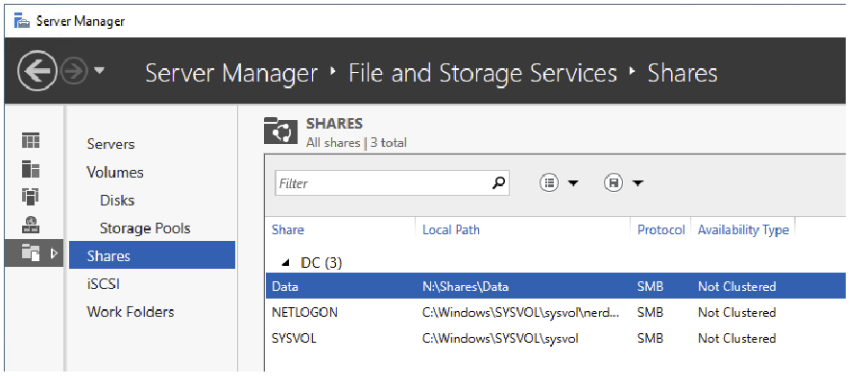
The next step was to harden the harden the AD environment using standard policies. We opened the Group policy Management panel through the tools option in the top left of server manager.



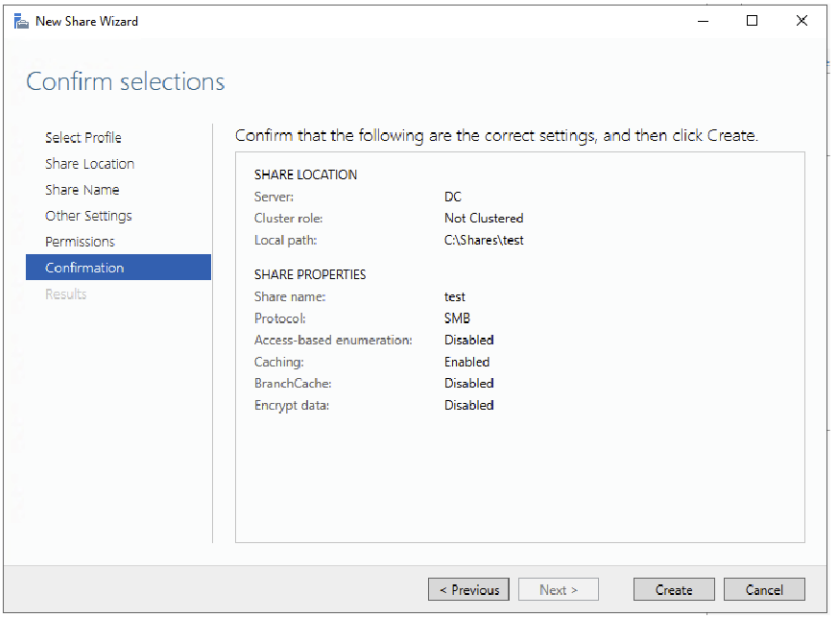
We then right clicked the default domain policy selected edit and then navigated to the password policy settings and made the shown changes.



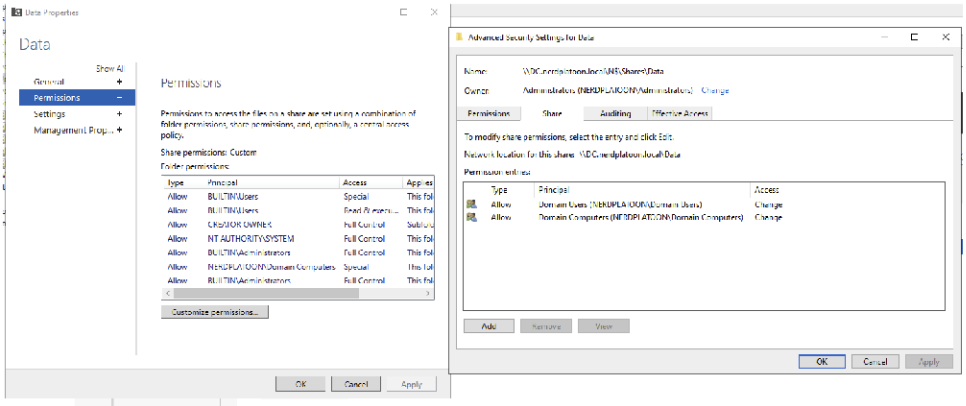
The next step was creating a shared drive for use by all domain users. We navigated to the File and storage services tab and then selected the shares tab



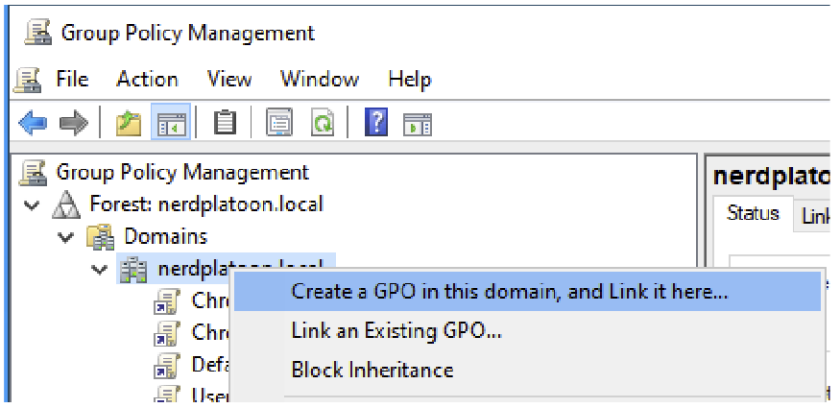
We right clicked in the shares field and selected create a new share. The default options are fine in all cases the final screen should look like this.



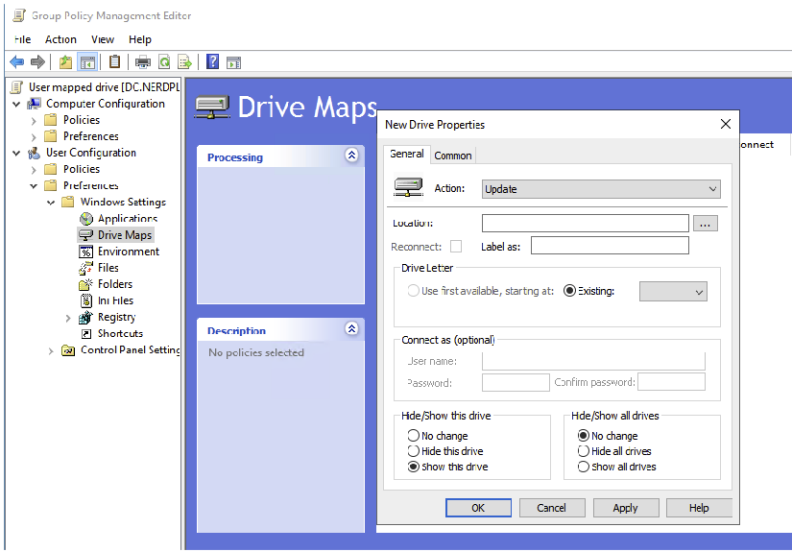
We edited the permissions for the share to allow alldomain users and computers



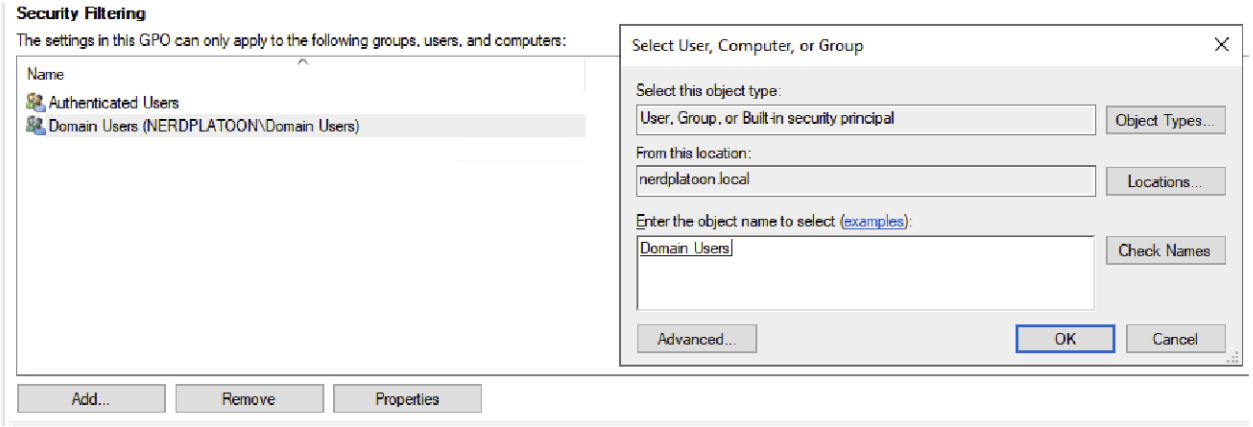
We then navigated back to the group policy management tool and created a new GPO linked to the domain by right clicking the domain and selecting Create new GPO and link it here



We named the GPO User Mapped Drive and frome the edit panel we navigated to the Drive maps section and right clicked the empty space and selected New > Mapped Drive. We entered the path of the created share as the location and selected the letter N for the drive. We also made sure to select show this drive at the bottom right of the section.

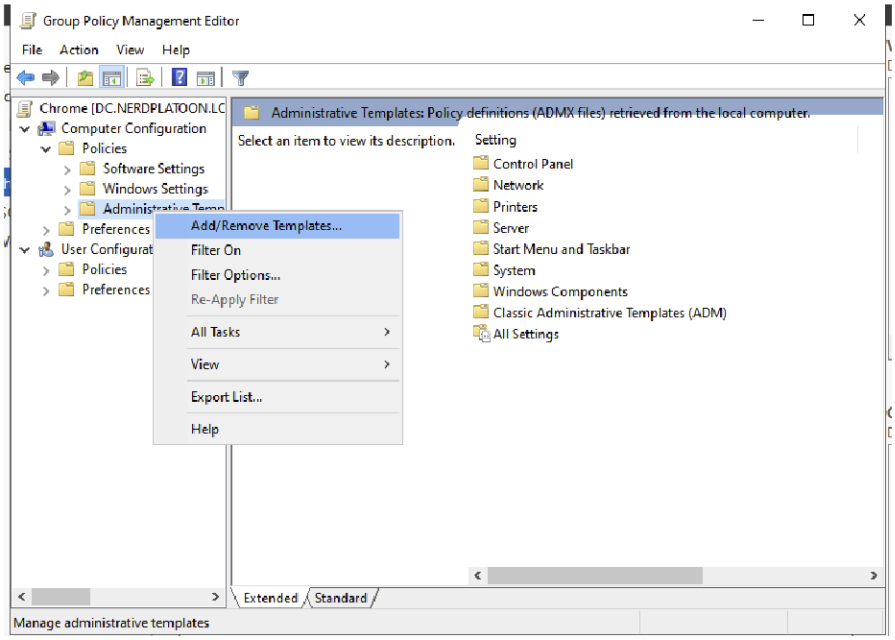


After we created the GPO we selected it and clicked add under the security filtering panel on the right. We entered Domain Users to make sure we applied the GPO correctly.

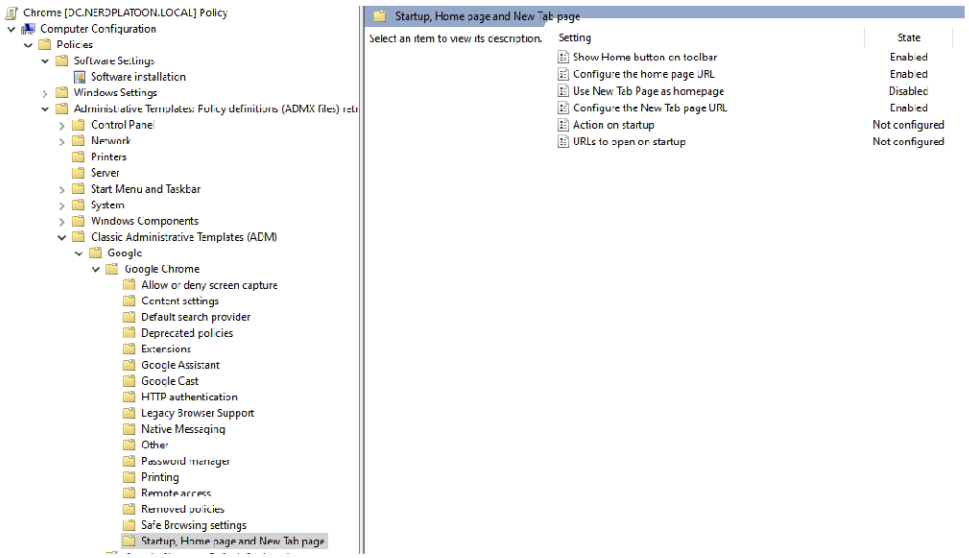


The next step was to set up an install of chrome for every new user. We downloaded the latest Chrome Browser MSI package from https://cloud.google.com/chrome-enterprise/browser/download/ and put it in the created shared network folder. We then downloaded the policy templates from https://dl.google.com/dl/edgedl/chrome/policy/policy_templates.zip and put them in the shared drive as well. We navigated back to the GPM window and created a new domain linked GPO

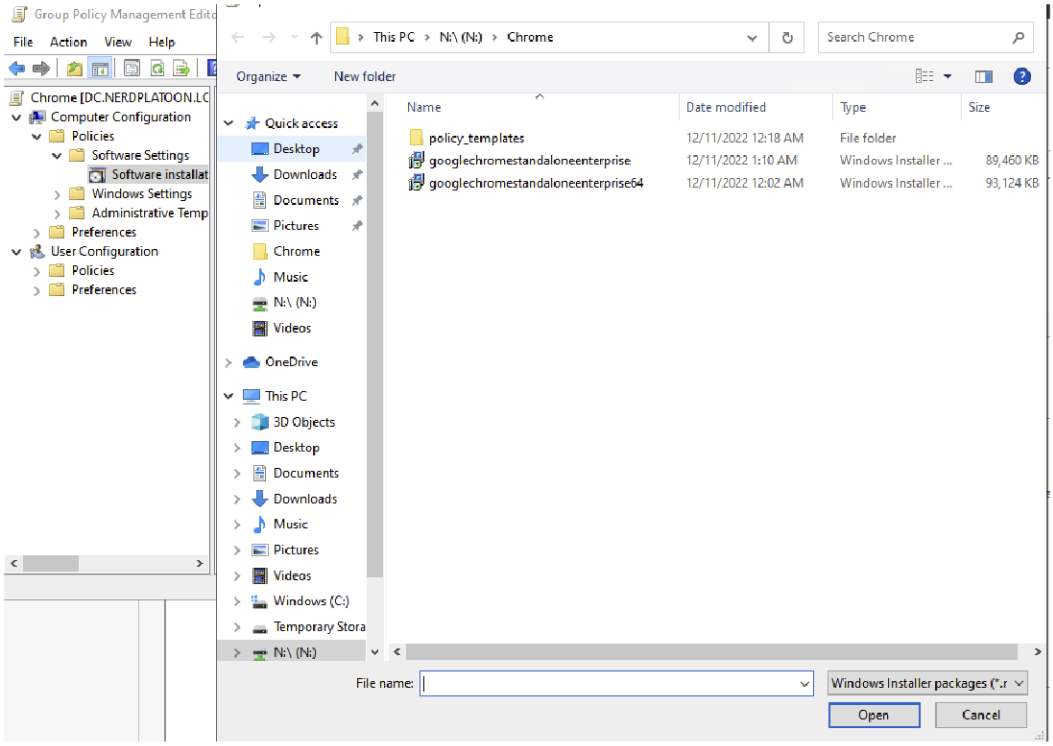
object. We named it Chrome and navigated to the administrative templates section under the computer configuration header. We right clicked the add new Administrative Template and selected the Chrome template located at \policy_templates\windows\admx\en-US



We then accessed the Chrome Template and configured the following settings, choosing https://www.parrotsec.org/ as a home page.



In the same GPO, we used the software settings to ensure the install of chrome on every machine.We also made sure to apply this GPO to ever computer in the domain.

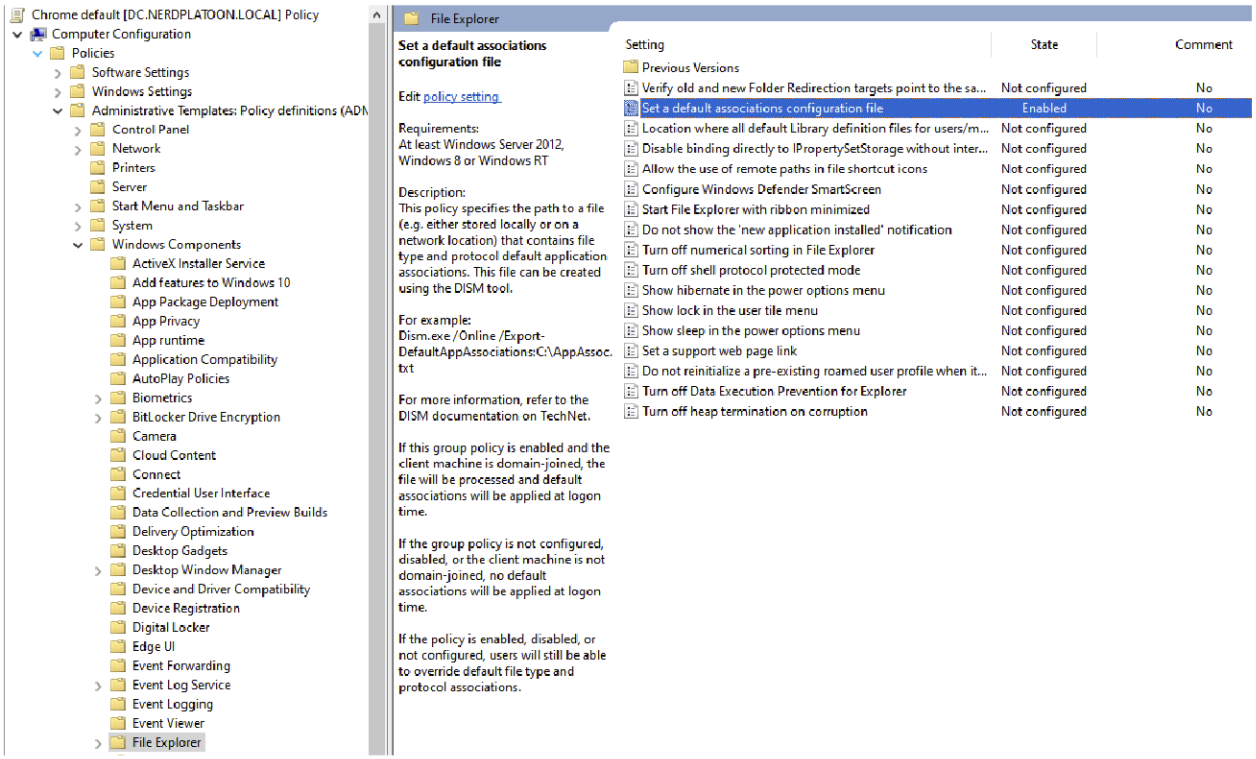


To make sure chrome is the default browser we downloaded googles default application association file from https://storage.googleapis.com/support-kms-prod/4E749252F96E04B747F4668C2E45278F42CF and added it to the shared drive.

N:\Chrome\chromedefault....

```
<?xml version="1.0" encoding="UTF-8"?>
- <DefaultAssociations>
    <Association ApplicationName="Google Chrome" ProgId="ChromeHTML" Identifier=".htm"/>
    <Association ApplicationName="Google Chrome" ProgId="ChromeHTML" Identifier=".html"/>
    <Association ApplicationName="Google Chrome" ProgId="ChromeHTML" Identifier="http"/>
    <Association ApplicationName="Google Chrome" ProgId="ChromeHTML" Identifier="https"/>
</DefaultAssociations>
```

We created a GPO (**Chrome default**) and applied it to all computers in the domain just as before. We edited it by going to **Computer Configuration > Policies > Administrative Template > Windows Components > File Explorer** and double-clicked **Set a default associations configuration file.**



We selected enabled and entered the location of the default application association XML file we downloaded. At that point the default browser would automatically be set as chrome on all computers in the domain.

Configuring Wireguard

In order to establish ad-blocking services for client machines within the network, we configured **Pi-hole** on our Bastion server. This uses **Wireguard** as a vpn service to tunnel all traffic from the internal network through a single external IP address. In addition, we needed to configure this external server similar to a VPN concentrator's functionality, whereby remote administrators could connect from the external WAN through a Wireguard tunnel and administer the virtual machines on the internal VNET. This is sometimes referred to as a **Hub-spoke network topology** in Azure, or as **point to cloud**. An understanding of the basic functionality of Wireguard is necessary to configure the larger Pi-hole and Hub-spoke network architectures.

Private and Public Key Generation and Exchange

```
wgadmin@Wireguard:~/wgdir$ wg genkey | tee wgprivate.key
```

```
wgadmin@Wireguard:~/wgdir$ cat wgprivate.key | wg pubkey > wgpublic.key
```

In order to first establish a tunnel between two Wireguard peers, one must generate a private key, and a corresponding public key for each machine using the above commands. The two machines must then exchange their public keys, preferably out-of-band, or using a secure protocol such as scp.

```
[Interface]
PrivateKey = WByR9nif7/BnzWtJ9j2TCLsqEQFHctUKLAwHDqGhF1U=
ListenPort = 11235

[Peer]
PublicKey = QW/iWFib76jnRR72j7sOldfQdkowRcZ5Ez2m7gONowE=
Endpoint =
AllowedIPs = 10.0.10.10/16, 10.0.10.20/16
```

The keys must then be recorded to a **.conf** document within **/etc/wireguard/** such as **wg0.conf** where **wg0** is the new virtual interface you will be creating. As shown above, the configuration file should be in the format found within the **wg** man pages. The public key and IP addresses are acquired during the earlier exchange. The endpoint is the IP address of the other end of the tunnel, and the **AllowedIPs** should be set to the address range of the internal subnets to which you will be connecting. During the initial configuration of wg, I attempted numerous entries for the **Allowed IPs** and the **endpoints** fields, as well as DNS. The internal network remained unresponsive during my attempts to reach it from external clients. I was however able to get an ICMP response from the public IP through the wireguard tunnel and between internal hosts.

Split-Tunnel VPN Connection

In order to allow our client machines to browse the internet and carry out other unrelated network tasks while simultaneously tunneling to the cloud infrastructure, a split tunnel must be configured in the wireguard VPN.

This is fairly simple, and it requires changes to one line in **wg0.conf**.
You must change AllowedIPs line to be equal to the internal IP address range of clients in the cloud deployment. For context, if one were to enter 0.0.0.0/0 as the AllowedIPs value, **wg-quick** would create entries in the routing table which direct all traffic destined for that IP range to the **wg0** device, through its tunnel.

Pi-hole Ad-blocking and VPN Gateway

This objective of this portion of the project was to set up an outward-facing Ubuntu server with a public IP address, through which all web traffic from the internal clients would be routed. In addition to protecting cloud virtual clients from the internet, this server would be configured with **Pi-hole**, which blocks advertisements for the entire network which has been routed through its automatically configured Wireguard tunnels.

Setting Up PiVPN

The initial startup of Pi-hole by pulling and executing a script for the installation of **pivpn**, which automatically configures wireguard connections for our Pi-hole ad-blocker:

```
curl -L https://install.pivpn.io | bash
```

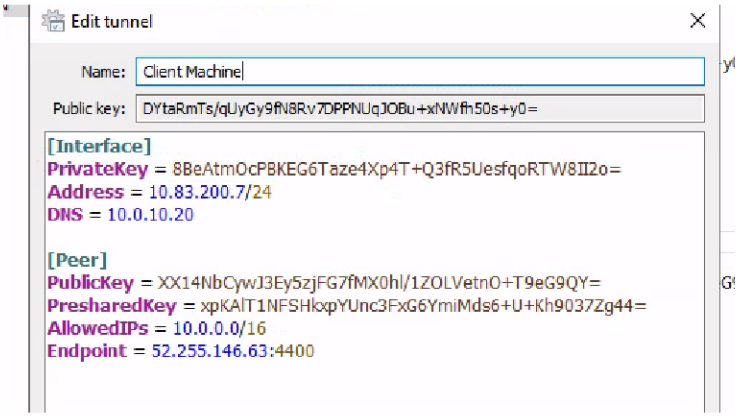
This results in several basic installation prompts. We selected our chosen wireguard port for the current configuration, to allow unattended upgrades, and whether to connect via public or private IP. (Later in the deployment, it was discovered that this would not work for all vpn need, in part because it provides a binary choice between public connection and private, and we needed both options for RDP and ad-blocking.) Afterwards, a restart is required.

The following step is simply to add new peers to the network, which will be automatically configured by the command **pivpn -a** which will ask for standard input of a peer name.

Pivpn automatically creates a new **wg0.conf** file in the folder **/etc/wireguard** writing out all of the interface and peer information for the wireguard connections. It adds new information for each new peer as the user invokes **pivpn -a**.

In addition, it creates a folder in the home directory called **configs**, populating it with automatically generated configuration files for use with each wireguard peer. The information in these files is meant to be entered into a new **/etc/wireguard/<virtualWGinterfacename>.conf**

file for linux peers, or into the GUI wireguard interface on Windows and MAC peers, as shown below:



This image displays the original configuration of **pivpn**, which was configured in the setup to connect through a public IP address. This was fixed later to establish fully functional ad-blocking. Below is displayed the first pivpn auto-configured **/etc/wireguard/wg0.conf** file for an external Kali machine.

```
[Interface]
PrivateKey = cI35WxjCLnwkm7uA4UGk9bzmLkT9OsIZ/nEssnmKy1w=
Address = 10.83.200.4/24
DNS = 9.9.9.9, 149.112.112.112

[Peer]
PublicKey = XX14NbCywJ3Ey5zjFG7fMX0hl/1ZOLVetnO+T9eG9QY=
PresharedKey = qhqJif/TUZKujQ0u9dlgkVaGkDQBskVJgM2mg7NIXGI=
Endpoint = 52.255.146.63:4400
AllowedIPs = 10.0.0.0/16
```

Originally, this configuration had 0.0.0.0/0 in the **AllowedIPs** section, but it was changed to the internal IP subnet ranges, as described in the **Split-Tunnel VPN Connection** earlier in this report.

To activate the new wireguard configuration on the linux machines, the command **sudo wg-quick up wg0** was used to quickly create the virtual wireguard interface. Once running, the command **sudo wg** can be run to display currently active interfaces and peers.

```
bastionadmin@Bastion:~/configs$ sudo wg
sudo: unable to resolve host Bastion: Name or service not known
interface: wg0
  public key: XX14NbCywJ3Ey5zjFG7fMX0hl/1ZOLVetnO+T9eG9QY=
  private key: (hidden)
  listening port: 4400

peer: ANBstrwJR88zhAnsDJSRkEpRoBao7q3bR8LXjNuw/DI=
  preshared key: (hidden)
  allowed ips: 10.83.200.2/32

peer: U3uGy96ZG1vlxgpat4F7nkAY5/xZZXTva00nlp2/PSE=
  preshared key: (hidden)
  allowed ips: 10.83.200.3/32

peer: BiULqRKHPSX+MwO3QS/mCwnYRKF80n1z1ZdYYkmI+A8=
  preshared key: (hidden)
  allowed ips: 10.83.200.4/32
bastionadmin@Bastion:~/configs$
```

This, however, does not prove that the wireguard connection is truly functional. The command **pivpn -c** displays all of the added devices, and recorded data about which ones were successfully connected.

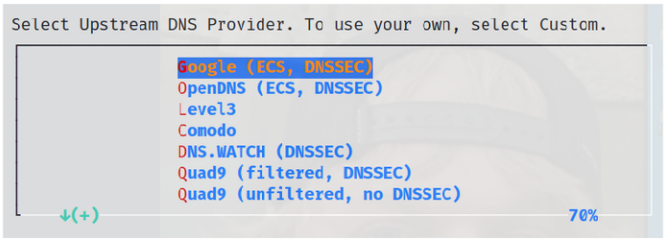
```
bastionadmin@Bastion:~/configs$ pivpn -c
sudo: unable to resolve host Bastion: Name or service not known
::: Connected Clients List :::
Name                Remote IP                Virtual IP       Bytes Receiv
ed      Bytes Sent      Last Seen
AllenMac            (none)                   10.83.200.2      0B
        0B              (not yet)
allenandroid        (none)                   10.83.200.3      0B
        0B              (not yet)
domaincontroller    (none)                   10.83.200.5      0B
        0B              (not yet)
CM2                 52.152.167.134:56361     10.83.200.7      185KiB
        3.9MiB          Dec 13 2022 - 00:31:27
kali                169.150.203.63:38385     10.83.200.4      1.5KiB
        708B            Dec 13 2022 - 00:18:17
::: Disabled clients :::
```

Installation of Pi-hole

To begin, the following script must be pulled from the pihole website, and installed on the **Bastion** machine.

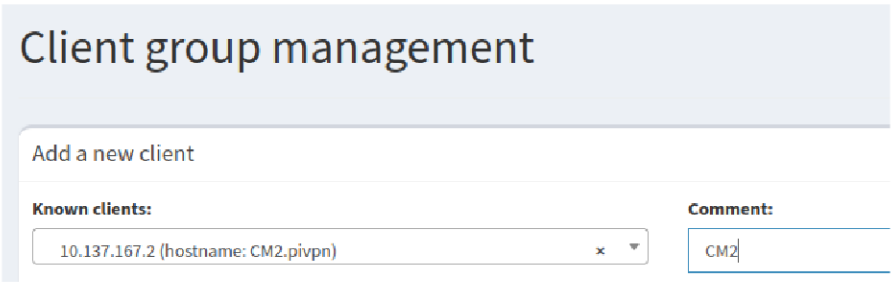
```
sudo curl -sSL https://install.pi-hole.net | bash
```

This will start a GUI menu sequence within the terminal window, where the interface, IP addresses, and gateway addresses are selected. Afterwards, the upstream DNS was chosen.



Pi-hole Configuration

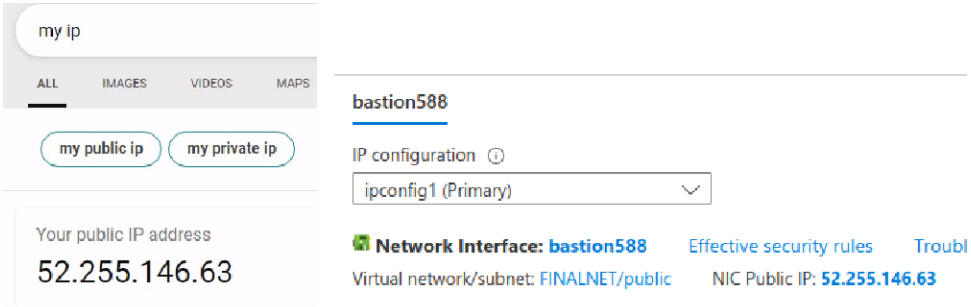
Once the Pi-hole has been installed, it was accessed by typing in the public IP of the Bastion server followed by /admin: **52.255.146.63/admin**. This will display the web interface, which contains GUI options to include one used to connect clients to the ad-blocker:



After connecting the client machine, **CM2**, it was time to ensure that it was properly routing HTTPS and DNS traffic through the right channels. First we established an RDP connection to **CM2's** currently exposed public IP address with the command:

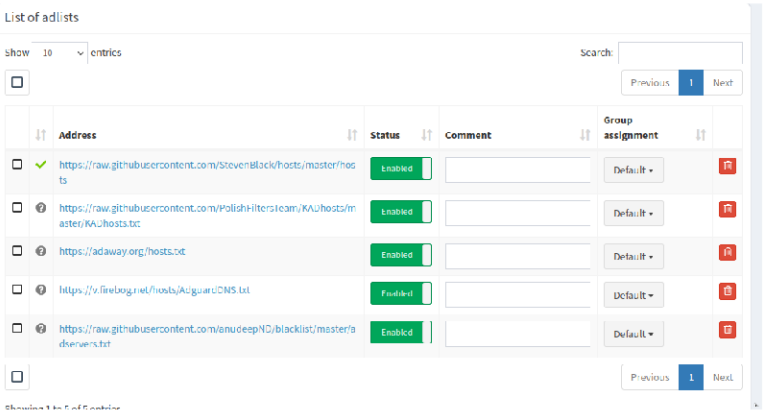
```
Xfreerdp /v:52.152.167.134 /u:dcadmin
```

From this point the public IP address of the HTTPS session was queried through a Bing search to check that the public IP displayed was that of **Bastion** rather than **CM2**.

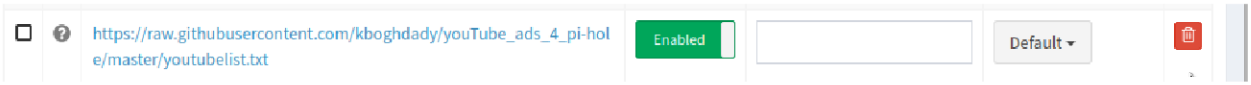


Ad-Blocking Lists

While this configuration came with a preinstalled ad-blocking list, this is not sufficient to block ads on a broad range of web domains. In order to have a more effective ad-blocking service, new lists were imported to the Pi-hole via its GUI web interface:

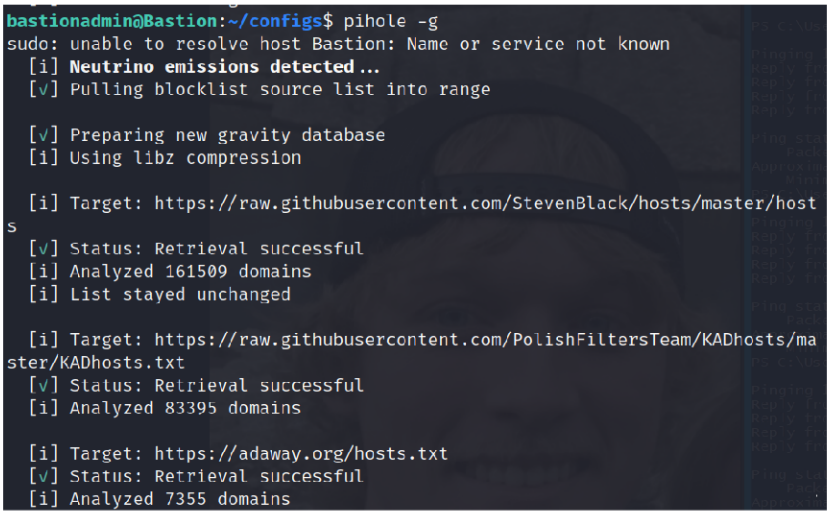


Lists such as these can be found online, and in order to block ads for a specific domain, a search for Youtube ad lists was made in order to test a targeted configuration:

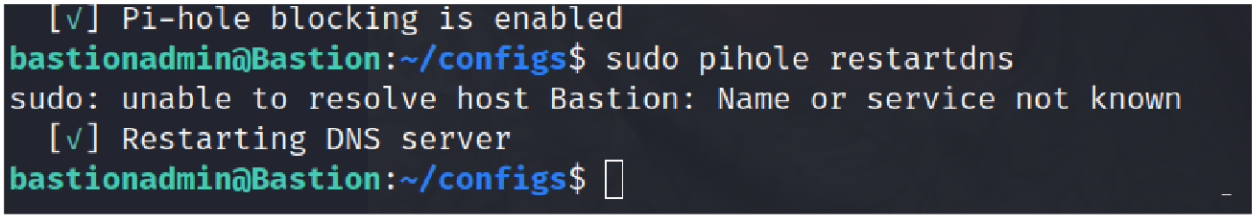


To test functionality, we browsed to Youtube in **CM2** before and after the following steps to make sure that advertisements had disappeared.

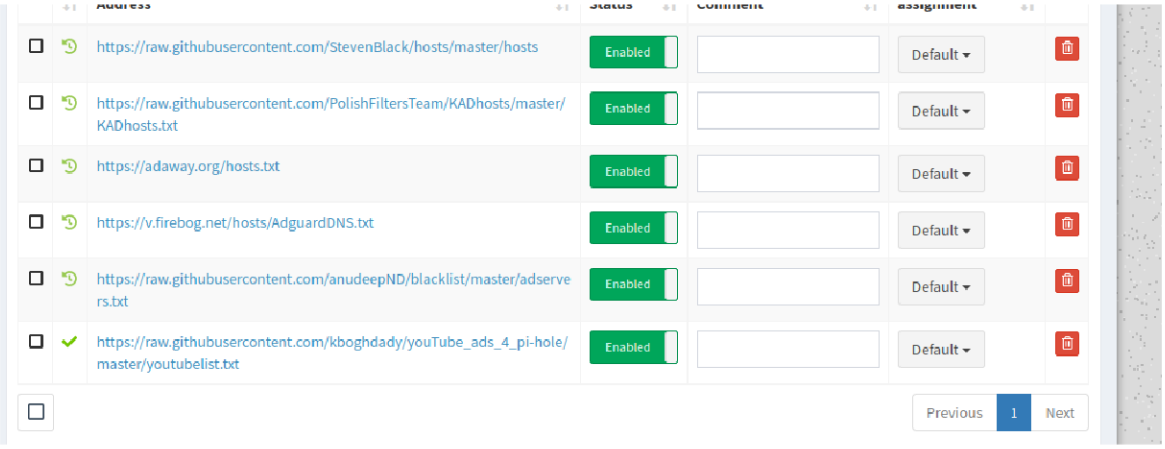
After different ad lists are loaded onto the Pi-hole GUI interface, the command **pihole -g** was used in the **Bastion** ssh session to create the Pi-hole's gravity database, which stores and manages the various ad lists:



In addition the Pi-hole DNS services needed to be restarted with the command **sudo pihole restart dns**



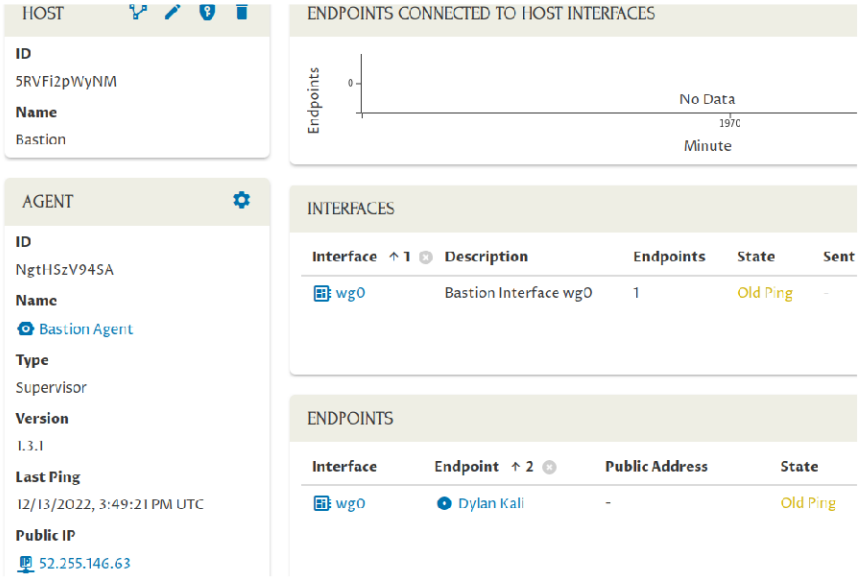
This will result in the Pi-hole web GUI showing that the lists are functional:



RDP through Wireguard Configuration

The final stage of the project involved the establishment of an RDP session with the internal **CM series** of client machines. After the original simple configuration using wireguard alone, attempts were made to ping and RDP through the **wg0** interface, however these attempts were non-responsive. The Bastion server's private IP was accessible through ping however. Later on after the pivpn configurations were established, similar results were observed. After numerous modifications of the wireguard configuration files on all three machines, clean restarts of the services, uninstalls and reinstalls, attempts from different external client operation systems, and triple-checks of firewalls on all devices, neither the basic wireguard configuration, nor configurations produced and modified from pivpn resulted in a successful connection beyond the Bastion server. As a further attempt at successful configuration, the original wireguard, pihole, and pivpn services were removed from all three devices, and **Pro**

Custodibus was installed to manage the wireguard configurations.



Screen shots from this period of the project were not retained, however numerous attempts at proper configuration were also made at many levels within the Pro Custodibus web tool, and within corresponding devices.

After determining that the problem must exist in a more foundational aspect of the cloud deployment, I made the decision to uninstall and recreate the Bastion and CM machines with another virtual network in order to design a **hub-spoke** network topology. During the process of rebuilding, I discovered that the desired architecture would not be compatible with the current layout within Azure. This was in part, because Azure does not support the movement of virtual machines between Vnets. As a workaround, I attempted to move these machines by deleting the virtual machine resource object, without deleting the hard drive. I would then create a new virtual network adapter and VM with the old storage drive selected on the desired Virtual Network. These machines would not boot, they would not be provisioned with the old hard drive as the OS drive (it would be a data drive), or they would be unresponsive to pings and RDP attempts to their public IP address. At this point it was determined that an entirely new project

would need to be started from the ground up, with a brand-new foundation of virtual networks purposed specifically for a **hub-spoke, point-to-site** topology geared for Azure.

Lessons Learned

I think that I made multiple strategic missteps from the beginning, which can be avoided in the future.

For one, I should have planned better. Before immediately installing programs and changing/experimenting with different configurations, I should have outlined exactly what the network needed to look like, including virtual nets, subnets, IP address ranges, different forms of nested connections, and the interplay between different programs.

In addition, record keeping was poor. During the previous osTicket deployment, I did not keep screen shots or complete documentation throughout construction of the network because it was fairly straight forward, and redoing anything which I had previously configured to take a screenshot was quick and easy. During this project, my lack of record-keeping resulted in poor systematization of troubleshooting. Rather than progressively getting closer to a functional network, multiple misconfigurations eventually led to the **CM** computers and the **Bastion** computers entering failed or unreachable states.

Not only did a lack of record-keeping result in my failure to recall past actions, my lack of backups, snapshots, and other proactive system recovery tools led to an inability to quickly recover from failed states and broken connections. This resulted in countless wasted hours.

Ultimately, while the RDP connection was never made, we learned a lot during this project, and it was a very beneficial experience for both of us.