Project Router

- Introduction: Novice to Network Admin
- LXC Guest Setup
- Initial Network Setup
- DNS: Recursive DNS
- Logging in LXC
- IPv4
    - Firewall Setup
    - DHCP and DNS Cache
- IPv6
    - IPv6 Intro
    - Firewall Setup
    - Prefix Delegation
    - DHCP and SLAAC
- Virtual Private Networking
    - Wireguard
    - Route Subnet Through Wireguard Interface
    - Remote Access
- Bonded Interface
- Network Intrusion Detection
- Traffic Graphing/Monitoring

Introduction: Novice to Network Admin

Introduction

If you're just looking to get started reading about how I setup everything you can skip down to the Goals section or go straight to Guest Setup to get started.

Background

From when I first started using computers as a kid I treated all things related to networking as a black box. I had a rudimentary understanding of IP addresses but had no real idea how data got from my computer to a server other than the high level concept of "*my computer is sending data to that address*". It is similar to how most of us don't really know how the US Postal Service works. We have a vague notion that we drop a letter in a mailbox to be picked up and then "*my letter gets delivered to the address on the envelope.*" In reality how mail gets picked up, sorted, tracked, bundled, routed, and delivered is much more involved than we ever think about. Up until about three years ago my knowledge hadn't progressed much past knowing network packets existed and for a computer to get on the Internet it had to have an IP address, a subnet mask (not that I really understood what this was), and have a router address.

Like most computer users I never delved into networking. I'd plug my router (always an Apple AirPort into the modem and go with mostly the defaults. I used *Linux* for years but strictly as a server for web apps I programmed and never ventured very far off that path.

That all started to change in 2018 when I decided to build a server (really just a fast PC at the time) to play around with. It also coincided with me starting a new job that put me adjacent to some networking topics that started to peak my interest. Pretty soon I was self hosting a several applications accessible from the Internet and wanted to take the plunge into setting up a DMZ to try and keep my local network safe from the ever increasing number of things I was making publically accessible. Looking into how best to cordon off my applications I saw that putting public applications on a different subnet and often a VLAN is the best practice. But my AirPort didn't support that so I went searching for a router/firewall that would work better. The general consensus at the time seemed to be to use a

Unifi Security Gateway or run pfSense on an old PC. Not having a PC to run it on I first tried virtualizing my router by running it as a virtual machine on my "server." For a newbie this was more complicated than necessary and had the risk that if not done correctly could expose my internal network to the Internet. I liked *pfSense* but I quickly found out the downside of running your gateway to the Internet on your server is that when your server has a problem you likely won't have the Internet to fix it which is quite frustrating.

So I bought a mini PC (Protecli Vault) and started running pfSense on there and was happy for a year. I continued to self-host more applications and eventually got to self-hosting DNS with Pi-hole. Pretty soon I ran into my old problem of when the server is down there is no DNS and so the Internet pretty much stops working. I briefly considered buying buyin a Raspberry Pi which would have been a great solution but I decided to treat the mini PC as my "network infrastructure server" and instead of just running pfSense on there I'd use Proxmox VE and virtualize both *pfSense* and *Pi-hole* the same way I had been virtualizing Pi-hole on my server.

This worked great except I eventually noticed that my maximum download speeds were slower than they had been when only *pfSense* was running on the min PC. After a bunch of testing and attempted workarounds I realized I ~~needed~~ *wanted* a new plan. So once again I bought a mini PC that was pretty much just a more powerful version of the one I already had to try and fix the problem. I was disappointed to see that my upgrade helped but wasn't enough to overcome the overhead that came along with virutalizing *pfSense*. Additionally I was starting to push into things that *pfSense* didn't support yet like using Wireguard.

The next step was to investigate if a *Linux* based firewall would perform better while virtualized. The answer turns out to be no since both *pfSense* and *Linux* both implement *Virtio* network drivers that work very similarly and the real problem seems to just be the result of the additional layers a packet has to travel up through. Each packet coming in must be processed by the hypervisor kernel then redirected to the virtualized router kernel and if the packet was destined somewhere else had to go through the same layers in reverse to exit. That seems to be jus too much for a device with modest CPU and memory performance.

Then it dawned on me that I could get around those layers of virtualization by using the same containerization I had been using for servers I had virtualized to run all my self-hosted projects. I prefer to run my guest machines as LXC (Linux Container) guests instead of virtual machines. A *LXC* guest uses the same *Linux* Kernel as the host Operating System so there are no additional layers of virtualization overhead to deal with. Pretty neat!

When I was looking at *Linux* based firewalls I came across VyOS which is based on Debian Linux and allowed me to peek behind the curtain of the types of tools you use for a *Linux* firewall. The seed of an idea had been planted that would use all the knowledge I gained over the last 3 years tinkering with my home network-- *Linux*, *Proxmox*, *Wireguard*, and virtualization.

I could run a *LXC* guest with *Debian* that would have zero virtualization overhead and provide all the functionality I needed. Best of all I'd be able to customize the firewall/router because it is just a *Linux* machine!

So I made a list of features I had been using on *pfSense* and *VyOS* to see what I'd have to implement.

Goals

- Basic *Linux* install in *LXC* guest
- Firewall protection of my local network from the Internet
- VLAN separation for added isolation between my subnets and the Internet
- DHCP to provide each subnet with IPv4 address assignment and local DNS resolution
- Recursive DNS for added security, privacy and removing reliance on external entities
- IPv6 stack support (DHCPv6, Router Advertisements (NDP), Prefix Delegation)
- *Wireguard* support

Extras

- Create bonded Ethernet interface to remove bandwidth bottleneck between router and switch
- Add intrusion prevension system like Snort
- Add bandwidth monitor and graphing

LXC Guest Setup

As I discussed in Introduction: Novice to Network Admin the goal is to run a router/firewall inside a *LXC* guest so there is little to no overhead when routing packets. So I created an **unpriviledged** *LXC* guest with a *Debian 10* template.

Resources

The mini PC this will run on isn't a powerhouse but should provide more than enough resources and have a few spare cycles leftover for something useful like running *Pi-hole*.

Cores	*unlimited* (4 cores)
Memory	2048 MiB
Swap	512 MiB
Root Disk	2 GB

Networking

Device	ID	Name
physical	net0	eth0
virtio	net1	eth1

Because there is some overhead with using an Ethernet Bridge I only wanted to use one where it made the most sense. Since the Ethernet connection from the modem will only ever talk directly to this *LXC* guest I am "passing" one of the physical Ethernet interfaces from *Proxmox* to this *LXC* guest. This makes it unavailable to the host and allows the *LXC* guest direct access to it similar to how PCI(e) Passthrough would work on a virtual machine.

This can be accomplished with *Proxmox*/*LXC* configuration similar to what is shown below.

```
# /etc/pve/lxc/100.conf
net1: name=eth1,bridge=vmbr0,hwaddr=D6:A9:67:D5:66:22,type=veth
```

```
+ lxc.net.0.type: phys
+ lxc.net.0.link: enp1s0
+ lxc.net.0.name: eth0
```

> Be careful to not reuse the same index for `lxc.net.[index]` and `net[index]` values or the guest will fail to boot.

Operating System

I didn't have to do much to the system itself other than making sure the timezone was correct and that it was up to date.

```
$ dpkg-reconfigure tzdata
$ apt update
$ apt upgrade
```

Initial Network Setup

Configure Interfaces

I need Internet access to download all the packages necessary so I setup *DHCP* on the WAN connection `eth0`.

> Setting all the local network interfaces to `manual` and not providing any addresses prevents any accidental routing before everything is secured.

```
# /etc/network/interfaces
   auto eth0
   iface eth0 inet dhcp
+
+  auto eth1
+    iface eth1 inet manual
+
+   auto eth1.8
+   iface eth1.8 inet manual
+          vlan-raw-device eth1
+
+   auto eth1.9
+   iface eth1.9 inet manual
+          vlan-raw-device eth1
```

Then I restart the networking service to apply the changes and create the new interfaces.

```
$ systemctl restart networking
```

Initial Security

I won't have SSH access allowed from the Internet when I am done but in the interim I want to install `fail2ban`. It doesn't hurt to have running even once the firewall is fully setup and provides just one more layer of defense.

```
$ apt install fail2ban
```

DNS: Recursive DNS

Option 1: Unbound

```
$ apt install unbound
```

Recursive DNS can sometimes sacrifice speed for security so the `unbound` server is going to be limited to only serve DNS requests from loopback addresses. Everyone else will have to go through a DNS caching server (*dnsmasq*) that I'll setup later to perform DNS queries.

```
# /etc/unbound/unbound.conf.d/local.conf
+ server:
+ interface: 127.0.0.1
+ interface: ::1
+ access-control: 127.0.0.1 allow
+ access-control: ::1 allow
```

```
$ systemctl restart unbound
```

Option 2: Public Recursive Name Server

I don't have to do anything since dnsmasq will be setup to query a public recursive DNS server like *Cloudflare*'s `1.1.1.1` and `1.0.0.1`.

Logging in LXC

Logging

One problem I ran into is that access to kernel logging is limited or unavailable from inside of a LXC container. For some usecases (like *netfilter*'s `LOG` action) any logging that happens in a LXC container will be blackholed and not recorded anywhere without a change on the host. Most often the solution to these permission/security problems is to find a way to allow access to these things from userspace.

ulogd2

I solved the *netfilter* `LOG` problem by simply using *ulogd2* to replace kernel logging with userspace logging. After installing and configuring *ulogd2* I just replaced any references to `LOG` with `NFLOG` in my *netfilter/iptables* rules. Don't worry if this doesn't make sense right now I'll talk about this more in the Firewall Setup section.

> ulogd is a userspace logging daemon for netfilter/iptables related logging. This includes per-packet logging of security violations, per-packet logging for accounting, per-flow logging and flexible user-defined accounting.

Installation

```
apt install ulogd2
```

Configuration

To get the output I wanted I had to edit the *ulogd2* config...

```
# /etc/ulogd2.conf
- stack=log1:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPKT,emu1:LOGEMU
+ #stack=log1:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPKT,emu1:LOGEMU
...
+ stack=log1:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPKT,firewall:LOGEMU
```

```
+ stack=log2:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPKT,firewall:LOGEMU
+ stack=log3:NFLOG,base1:BASE,ifi1:IFINDEX,ip2str1:IP2STR,print1:PRINTPKT,firewall:LOGEMU
+
+ [firewall]
+ file="/var/log/ulog/firewall.log"
+ sync=1
```

Connection Tracking

Similarly to *netfilter* logging connection tracking in a LXC container is more limited due to not having access to the host's `/proc/` filesystem. But I can install *conntrack* to provide a way to see connection tracking from userspace.

conntrack

> The conntrack utilty provides a full featured userspace interface to the Netfilter connection tracking system that is intended to replace the old /proc/net/ip_conntrack interface. This tool can be used to search, list, inspect and maintain the connection tracking subsystem of the Linux kernel.

Installation

```
apt install conntrack
```

IPv4

IPv4

Firewall Setup

Install Shorewall

To manage `nftables/iptables` I decided to go with Shorewall since it is easy to configure and very mature. At some point I may look into switching to FireHol since it looks even simpler to configure but I wanted something I knew I'd be able to make do everything I needed.

I started by installing *shorewall* as my firewall, *shorewall-doc* which includes examples, and *shorewall-init* which can lockdown the system at boot before *Shorewall* has had a chance to configure the firewall.

```
# apt install shorewall shorewall-doc shorewall-init
```

Then I update the *shorewall* configuration to reflect that I'm using *ulogd2* for logging and that I want IPv4 forwarding enabled when *shorewall* starts.

```
# /etc/shorewall/shorewall.conf
- LOG_LEVEL="info"
+ LOG_LEVEL="NFLOG(1,0,1)"
...
- LOGFILE=/var/log/messages
+ LOGFILE=/var/log/firewall.log
...
- IP_FORWARDING=Keep
+ IP_FORWARDING=Yes
```

All my configuration files are adapted from the examples that *shorewall-doc* makes available under `/usr/share/doc/shorewall/examples`.

Setting up the zones is pretty self-explanitory. The only addition I made is I have a `warp` zone which I will use later when I am setting up my VPN.

```
# /etc/shorewall/zones
+ #------------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-zones"
```

```
+ #
+ # See http://shorewall.net/manpages/shorewall-zones.html for more information
+
#############################################################################
############
+ #ZONE   TYPE    OPTIONS             IN                  OUT
+ #                                OPTIONS             OPTIONS
+ fw      firewall
+ wan     ipv4
+ lan     ipv4
+ dmz     ipv4
+ warp    ipv4
```

Setting up the interfaces and assiging them zones is also pretty self-explanatory.

```
# /etc/shorewall/interfaces
+ #-------------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-interfaces"
+ #
+ # See http://shorewall.net/manpages/shorewall-interfaces.html for more information
+
#############################################################################
############
+ ?FORMAT 2
+
#############################################################################
############
+ #ZONE□□INTERFACE       OPTIONS
+ wan□□WAN_IF□□□tcpflags,dhcp,nosmurfs,routefilter,logmartians,sourceroute=0,physical=eth0
+ lan□□LAN_IF□□□tcpflags,dhcp,nosmurfs,routefilter,logmartians,physical=eth1
+ dmz□□DMZ_IF□□□tcpflags,dhcp,nosmurfs,routefilter,logmartians,physical=eth1.8
+ warp□□WARP_IF□□□tcpflags,dhcp,nosmurfs,routefilter,logmartians,physical=eth1.9
```

My real `/etc/shorewall/policy` file is less liberal than what is shown below (`lan` being allowed to access whatever it wants) but I wanted to show a reasonably secure policy that allowed me to have a very simple `/etc/shorewall/rules` config below.

```
# /etc/shorewall/policy
+ #-------------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-policy"
+ #
```

```
+ # See http://shorewall.net/manpages/shorewall-policy.html for more information
+
############################################################################
############
+ #SOURCE□DEST□□POLICY□□LOGLEVEL□RATE    CONNLIMIT
+
+ $FW□□all□□□ACCEPT
+ lan□□all□□□ACCEPT
+ dmz□□$FW,wan□□ACCEPT
+ warp□□$FW□□□ACCEPT
+
+ wan□□all□□□DROP□□$LOG_LEVEL
+ # THE FOLLOWING POLICY MUST BE LAST
+ all□□all□□□REJECT□□$LOG_LEVEL
```

Because my example policy is pretty open, my rules in this example are pretty sparse.

```
# /etc/shorewall/rules
+ #------------------------------------------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-rules"
+ #
+ # See http://shorewall.net/manpages/shorewall-rules.html for more information
+
############################################################################
############################################################################
############################################################
+ #ACTION        SOURCE         DEST           PROTO   DEST    SOURCE         ORIGINAL       RATE         USER/
MARK    CONNLIMIT       TIME           HEADERS        SWITCH         HELPER
+ #                                             PORT    PORT(S)        DEST           LIMIT        GROUP
+ ?SECTION ALL
+ ?SECTION ESTABLISHED
+ ?SECTION RELATED
+ ?SECTION INVALID
+ ?SECTION UNTRACKED
+ ?SECTION NEW
+
+ #       Don't allow connection pickup from the net
+ Invalid(DROP)    wan            all            tcp
+
+ DNS(ACCEPT)     all!wan,warp    $FW
```

```
+ DNS(ACCEPT)     $FW,dmz        lan:10.0.1.2
+
+ Web(ACCEPT)     dmz            $FW
+ Web(DNAT)       wan            dmz:10.0.8.2
```

Lastly is the magic that allows private addresses to access the Internet by masquerading them all as my one public IPv4 address I am assigned. The following just says all traffic heading out of `WAN_IF` (`eth0`) coming from a private IP range should be masqueraded.

```
# /etc/shorewall/snat
+ #------------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-snat"
+ #
+ # See http://shorewall.net/manpages/shorewall-snat.html for more information
+
################################################################################
################################################################################
#####
+ #ACTION              SOURCE              DEST         PROTO   PORT    IPSEC   MARK    USER
SWITCHORIGDEST PROBABILITY
+ MASQUERADE          10.0.0.0/8,\
+                     169.254.0.0/16,\
+                     172.16.0.0/12,\
+                     192.168.0.0/16       WAN_IF
```

Now that I have everything configured it might be wise to run `shorewall check` just to make sure I didn't have any typos.

I hooked *shorewall* into the boot process to make sure the system is secure during boot by enabling *shorewall-init.service* and *shorewall.service*. First I told *shorewall-init* that it needs to account for *shorewall* when it runs.

```
# /etc/default/shorewall-init
- PRODUCTS=""
+ PRODUCTS="shorewall"
```

Then I simply told those services to start at boot.

```
# systemctl enable shorewall
# systemctl enable shorewall-init
```

Modify Interfaces

Now that *Shorewall* will secure everything at bootup it is safe to update `/etc/networking/interfaces` and add their IPv4 addresses.

```
# /etc/networking/interfaces
 auto eth1
- iface eth1 inet manual
+ iface eth1 inet static
+         address 10.0.1.1/21


  auto eth1.8
- iface eth1.8 inet manual
+ iface eth1.8 inet static
          vlan-raw-device eth1
+         address 10.0.8.1/24


  auto eth1.9
- iface eth1.9 inet manual
+ iface eth1.9 inet static
          vlan-raw-device eth1
+         address 10.0.9.1/24
```

Now if I reboot the system all my interfaces will come up configured and the system will be protected by *nftables/iptables* configured by *Shorewall*.

> Be sure to sanity check the configuration so Shorewall doesn't block SSH access if that is needed.

```
# reboot
```

IPv4

DHCP and DNS Cache

Install dnsmasq

I decided to use dnsmasq since it can fulfull multiple roles as both a DHCP and DNS cache. I'll first configure it for IPv4 and then later add in the few extra IPv6 lines needed.

Setup DHCP

The following can look complicated but that is just becuase there are a ton of MAC Addresses and IP Addresses mixed throughout. If you look closely you can see that there are only four types of lines.

1. `no-dhcp-interface=eth0,lo` prevents DHCP binding on our loopback address and `eth0` which is the interface facing the Internet.
2. `dhcp-range=` declares a start and stop address and lease lifetime for each subnet. I am also setting an optional tag for each so I can target them later if I want.
3. `dhcp-option=` allows me to set specific DHCP options. The `tag:` allows me to target addresses matching a specific tag. I am overriding the default DNS servers because I want `lan` and `dmz` to use my *Pi-hole* server and `warp` should use a public DNS server since any device on that subnet is routed through a VPN tunnel so it doesn't have local network access.
4. `dhcp-host=` defines what IP addresses and hostnames get assigned to which network device with a specific MAC address

```
# /etc/dnsmasq.d/dhcp.conf
+ no-dhcp-interface=eth0,lo
+
+ dhcp-range=set:lan,10.0.5.1,10.0.7.254,12h
+ dhcp-range=set:dmz,10.0.8.1,10.0.8.254,12h
+ dhcp-range=set:warp,10.0.9.1,10.0.9.254,5m
+
+ dhcp-option=tag:lan,option:dns-server,10.0.1.2
+ dhcp-option=tag:lan,option:dns-server,10.0.1.2
```

```
+ dhcp-option=tag:warp,option:dns-server,1.1.1.1,1.0.0.1
+
+ # LAN - network infrastructure
+ dhcp-host=aa:af:57:f3:4e:90,10.0.1.2,pihole□□□# pihole
+ dhcp-host=b4:fb:e4:8f:f9:74,10.0.1.3,unifi-switch-8□# unifi-switch-8
+
+ # LAN  - proxmox
+ dhcp-host=e0:d5:5e:63:fe:30,10.0.3.2,blackbox□□□# blackbox
+ dhcp-host=70:85:c2:fe:4c:b7,10.0.3.3,mini□□□□# mini
+ dhcp-host=6e:91:84:4a:74:f1,10.0.3.4,backup□□□# backup
+
+ # LAN - assigned devices
+ dhcp-host=d0:a6:37:ed:8c:7f,10.0.4.4,silverbook□□# silverbook
+ dhcp-host=82:13:00:9c:c7:00,10.0.4.5,thunderbolt□□# thunderbolt
+ dhcp-host=34:36:3b:7f:18:1e,10.0.4.8,jess□□□□# jess
+ dhcp-host=96:64:5F:1C:A6:2C,10.0.5.6,refuge□□□# refuge
+ dhcp-host=7A:BC:46:D1:A3:1B,10.0.5.9,unifi□□□# unifi
+
+ # DMZ - assigned devices
+ dhcp-host=62:59:92:A7:1D:F1,10.0.8.5,bitcoin□□□# bitcoin
+ dhcp-host=32:cc:fb:a3:1a:57,10.0.8.2,contained□□# contained
```

Setup DNS Caching

Everything here is commented with an explanation of what it does. The only thing slightly interesting is I have two `server=` parameters pointing to the IPv4 loopback addresses which is where *Unbound* is listening. If *Unbound* wasn't being used I'd either remove `no-resolv` and use the system namesevers or change the `server=` parameters to point to a public recursive name sever.

```
# /etc/dnsmasq.d/dns.conf
+ # Add the domain to simple names (without a period) in /etc/hosts in the same way as for DHCP-derived
names.
+ expand-hosts
+
+ # Log the results of DNS queries handled by dnsmasq.
+ log-queries
+
+ # Do not listen on the specified interface.
+ except-interface=eth0,lo
```

+
+ # Accept DNS queries only from hosts whose address is on a local subnet, ie a subnet for which an interface exists on the server.
+ local-service
+
+ # Dnsmasq binds the address of individual interfaces, allowing multiple dnsmasq instances, but if new interfaces or addresses appear, it automatically listens on those
+ bind-dynamic
+
+ # Return answers to DNS queries from /etc/hosts and --interface-name which depend on the interface over which the query was received.
+ localise-queries
+
+ # All reverse lookups for private IP ranges (ie 192.168.x.x, etc) which are not found in /etc/hosts or the DHCP leases file are answered with "no such domain"
+ bogus-priv
+
+ # Later versions of windows make periodic DNS requests which don't get sensible answers from the public DNS and can cause problems by triggering dial-on-demand links.
+ filterwin2k
+
+ # Enable code to detect DNS forwarding loops
+ dns-loop-detect
+
+ # Reject (and log) addresses from upstream nameservers which are in the private ranges.
+ stop-dns-rebind
+
+ # Exempt 127.0.0.0/8 and ::1 from rebinding checks.
+ rebind-localhost-ok
+
+ # Tells dnsmasq to never forward A or AAAA queries for plain names, without dots or domain parts, to upstream nameservers.
+ domain-needed
+
+ # Specifies DNS domains for the DHCP server.
+ domain=hermz.io
+
+ # Don't read /etc/resolv.conf. Get upstream servers only from the command line or the dnsmasq configuration file.

```
+ no-resolv
+
+ server=127.0.0.1
+ server=::1
```

Resolve Static Clients

One problem I ran into was that static clients never use DHCP so the DHCP server doesn't register their hostname with their intended IP address. To work around this limitation I just added those entries to the `/etc/hosts` file since by default *dnsmasq* will resolve using those entries too.

```
# /etc/hosts
+ 10.0.1.1       ember
+ 10.0.1.2       pihol
+ 10.0.1.3       unifi-switch-8
+ 10.0.3.2       blackbox
+ 10.0.3.3       mini
+ 10.0.3.4       backup
+ 10.0.3.5       edge


  # --- BEGIN PVE ---
```

Reboot

Now that *dnsmasq* is fully configured I just restart it using *systemctl*

```
# systemctl restart dnsmasq.service
```

IPv6

IPv6 Intro

Refresher

For a quick crash course into IPv6 checkout my IPv6 Quick Explainer guide.

Why Did I Setup IPv6?

Beyond just being good to know because it'll be what we're all using sooner than later there are a few practical advantages of IPv6 over IPv4. Most important to me though is being able to have IP addresses that don't have to be masqueraded by the router. This has several knock-on effects I appreciate.

No Need for Hairpin NAT

I don't have to masquerade IP addresses which means that when I access a device from my LAN I can use the same IP address that is used when people access a device from the WAN. I don't have to setup a hacky Hairpin NAT or necessarily use Split-horizen DNS to just have everything work. The less janky configurations I have to create and maintain to paper over problems of IPv4 the better.

Fine-grained DNS Control

Because each device can have a publically routable address I can setup subdomains to **actually** point to different addresses. As an example I can have `wireguard.swigg.net` point to my router IP address for VPN access while `*.swigg.net` can point to my server IP address I am running in my DMZ. With IPv4 I had to have them both point to my router public IP address and then use some sort of proxy to forward based on hostname plus do something janky like above.

Firewall Setup

Install Shorewall6

Configuring *Shorewall* for IPv6 is nearly identical to how I did it for IPv4. The biggest different is I can skip most things related to *masquerading* since that is less often necessary in the world of IPv6.

The only changes that need to be made is installing and configuring *shorewall6*. I am not going to go over everything again since it is nearly identical to Firewall Setup under IPv4 but pay close attention to the path is now `/etc/shorewall6`

```
# apt install shorewall6
```

```
# /etc/shorewall6/shorewall.conf
- LOG_LEVEL="info"
+ LOG_LEVEL="NFLOG(1,0,1)"
...
- LOGFILE=/var/log/messages
+ LOGFILE=/var/log/firewall.log
...
- IP_FORWARDING=Keep
+ IP_FORWARDING=Yes
```

```
# /etc/shorewall6/zones
+ #----------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-zones"
+ #
+ # See http://shorewall.org/manpages/shorewall-zones.html for more information
+
###############################################################################
############
+ #ZONE   TYPE    OPTIONS             IN                  OUT
+ #                                   OPTIONS             OPTIONS
+ fw      firewall
```

+ wan   ipv4
+ lan   ipv4
+ dmz   ipv4
+ warp   ipv4

/etc/shorewall6/interfaces
+ #--
+ # For information about entries in this file, type "man shorewall6-interfaces"
+ #
+ # See http://shorewall.org/manpages/shorewall-interfaces.html for more information
+
###
#############
+ ?FORMAT 2
+
###
#############
+ #ZONE□INTERFACE□OPTIONS
+ wan□□WAN_IF□□tcpflags,dhcp,forward=1,accept_ra=2,sourceroute=0,physical=eth0
+ lan□□LAN_IF□□tcpflags,dhcp,forward=1,physical=eth1
+ dmz□□DMZ_IF□□tcpflags,dhcp,forward=1,physical=eth1.8
+ warp□□WARP_IF□□tcpflags,dhcp,forward=1,physical=eth1.9

/etc/shorewall6/policy
+ #--
+ # For information about entries in this file, type "man shorewall-policy"
+ #
+ # See http://shorewall.net/manpages/shorewall-policy.html for more information
+
###
#############
+ #SOURCE□DEST□□POLICY□□LOGLEVEL□RATE   CONNLIMIT
+
+ $FW□□all□□□ACCEPT
+ lan□□all□□□ACCEPT
+ dmz□□$FW,wan□□ACCEPT
+ warp□□$FW□□□ACCEPT
+
+ wan□□all□□□DROP□□$LOG_LEVEL
+ # THE FOLLOWING POLICY MUST BE LAST

```
+ all□□all□□□REJECT□□$LOG_LEVEL
```

```
# /etc/shorewall6/rules
+ #------------------------------------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-rules"
+ #
+ # See http://shorewall.net/manpages/shorewall-rules.html for more information
+
###############################################################################
###############################################################################
#############################################################################
+ #ACTION□□□SOURCE          DEST           PROTO   DEST    SOURCE          ORIGINAL        RATE         USER/
MARK    CONNLIMIT       TIME           HEADERS         SWITCH          HELPER
+ #                                        PORT    PORT(S)         DEST            LIMIT        GROUP
+ ?SECTION ALL
+ ?SECTION ESTABLISHED
+ ?SECTION RELATED
+ ?SECTION INVALID
+ ?SECTION UNTRACKED
+ ?SECTION NEW
+
+ #       Don't allow connection pickup from the net
+ Invalid(DROP)□□wan             all             tcp
+
+ DNS(ACCEPT)□□all!wan,warp    $FW
+ DNS(ACCEPT)□□$FW,dmz         lan:2001:db8:2fa3:4848::9a57:cec2
+
+ Web(ACCEPT)□□dmz             $FW
+ Web(ACCEPT)□□wan             dmz:2001:db8:2fa3:4848:66:1cb:59a7:bbe1
```

At this point I just have an empty `/etc/shorewall6/snat` configuration because IPv6 doesn't need masqueraded to access the Internet.

```
# /etc/shorewall/snat
+ #------------------------------------------------------------------------------
+ # For information about entries in this file, type "man shorewall-snat"
+ #
+ # See http://shorewall.org/manpages/shorewall-snat.html for more information
+
###############################################################################
```

```
################################################################################
#####
+ #ACTION            SOURCE              DEST        PROTO   PORT    IPSEC   MARK    USER    SWITCH
ORIGDEST        PROBABILITY
```

Just like before it might be wise to run `shorewall6 check` just to make sure I didn't have any typos.

I already enabled *shorewall-init.service* to secure the system during boot so to hook in *shorewall6* I just needed to edit its configuration and then enable *shorewall6.service* to start at boot like I already did for *shorewall.service* and *shorwall-init.service*.

```
# /etc/default/shorewall-init
- PRODUCTS="shorewall"
+ PRODUCTS="shorewall shorewall6"
```

Then I told it to start at boot.

```
# systemctl enable shorewall6
```

Reboot

It isn't strictly neccessary to reboot but I just prefer to see my system as it would be after it starts up.

```
# reboot
```

IPv6

Prefix Delegation

I'd recommend reading about Prefix Delegation to get a better understanding of it but the gist is that using DHCPv6 it is possible to request a "prefix" where any IPv6 address starting with that will be routed to the router. Then the router can use that to configure clients on the network to each have a unique address instead of the router only one (as in IPv4) and having to share it using a hack like masquerading.

Install A Client

There are a few different DHCPv6 clients you can use that support *Prefix Delegation* but I decided to go with *wide-dhcpv6-client*. I also tried *dhcpcd* but found the configuration syntax to be a little uglier.

```
# apt install wide-dhcpv6-client
```

The config below is doing a few different things that I'll list but you can read about all the possible dhcp6c.conf configuration.

```
# /etc/wide-dhcpv6/dhcp6c.conf
+
+ interface eth0 {
+ #  send rapid-commit;
+   send ia-na 0;
+   send ia-pd 1;
+ };
+
+ id-assoc na 0 {
+
+ };
+
+ id-assoc pd 1 {
+   prefix ::/60 infinity;
+
+   prefix-interface eth1 {
```

```
+    sla-id 0;
+    sla-len 4;
+    ifid 1;
+  };
+
+  prefix-interface eth1.8 {
+    sla-id 1;
+    sla-len 4;
+    ifid 1;
+  };
+
+  prefix-interface eth1.9 {
+    sla-id 2;
+    sla-len 4;
+    ifid 1;
+  };
+ };
```

- **Lines 3-7** use `eth0` to request a "normal address" with `ia-na` and a delegrated prefix range with `ia-pd`
- **Lines 9-11** are required and correspond to the "normal addresss" I asked for, but there is no extra configuration needed
- **Lines 13 and 14** are the start of the prefix delegation block and I am specifying I want a prefix that gives me a subnet of `/60`. I know *Comcast* will give me a `/60` which is 295,147,905,179,352,825,856 (two hundred ninety five quintillion, one hundred forty seven quadrillion, nine hundred five trillion, one hundred seventy nine billion, three hundred fifty two million, eight hundred twenty five thousand, eight hundred fifty six) so that should be more than enough.
- **Lines 16-20** and the other similar blocks just specify what slice of the prefix I was delegated that I want applied to each interface. The lines `sla-id` (Site-Level Aggregation identifier) is just an index to a what is basically an IPv4 subnet, `sla-len` is the size of that subnet (I requested a /60 so if our SLA is /4 then I end up with /64 which is ideal for an IPv6 subnet), and `ifid` just defines that I want the first address available in our subnet assigned to our interface. So if the IPv6 addresses assigned to this interface was `2601:1833:a3a:100::/64` the interface would have `2601:1833:a3a:100::1/64` assigned to it.

The next step was just to enable and run the service.

```
# systemctl enable --now wide-dhcpv6-client
```

Then I was able to verify that I had publicaly accessible IPv6 addresses.

```
# ip -6 addr
1: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 2001:6020:ae3:1022:a4d3:f031:fb7e:e629/128 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::2b0:c9ff:fe79:cd77/64 scope link
       valid_lft forever preferred_lft forever
2: eth1@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 2601:1833:a3a:100::1/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::d45a:67ff:fec6:6688/64 scope link
       valid_lft forever preferred_lft forever
3: eth1.8@eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 2601:1833:a3a:101::1/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::d45a:67ff:fec6:6688/64 scope link
4: eth1.9@eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 2601:1833:a3a:102::1/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::d45a:67ff:fec6:6688/64 scope link
       valid_lft forever preferred_lft forever
...
```

Exactly like I hoped, I can see that using `ia-nd` to request a "normal address" for `eth0` resulted in `2001:6020:ae3:1022:a4d3:f031:fb7e:e629/128` being assigned as my routers public IPv6 address. It also looks like the prefix delegation worked since `eth1`, `eth1.8`, `eth1.9` all have the same prefix with incrementing SLA identifiers that you can see represented by `100`, `101`, `102` in their addresses.

DHCP and SLAAC

I already setup *dnsmasq* for IPv4 and so there is very little that needs to be done to add IPv6 support.

I just needed to add `dhcp-range` lines for each subnet. I am tagging them the same as before and using the `::,constructor:<interface>` syntax to tell *dnsmasq* to determine the the prefix the DHCPv6 range should be valid over from the GUAs (Global Unicast Addresse) (publically routable IPs) on each interface. These were assigned in the previous section (Prefix Delegation) by *wide-dhcpv6-client*. Declaring `ra-stateless` configures *dnsmasq* to use SLAAC to automatically configure clients in this prefix.

```
# /etc/dnsmasq.d/dhcp.conf
+ dhcp-range=set:lan,::,constructor:eth1,ra-stateless,12h
+ dhcp-range=set:dmz,::,constructor:eth1.8,ra-stateless,12h
+ dhcp-range=set:warp,::,constructor:eth1.9,ra-stateless,5m
```

Then I enabled router advertisements so *dnsmasq* will broadcast information to any potential clients on the subnet.

```
# /etc/dnsmasq.d/router-advertisements.conf
+ enable-ra
```

Virtual Private Networking

Wireguard

I had two goals I wanted to accomplish with VPNs.

1. I need to redirect all outbound traffic from a specific subnet through a VPN so any client on that subnet would have its privacy protected by the VPN.
2. Allow me to VPN into my home network from somehwere else and have access to everything as if I was sitting on my computer at home.

Both of them could have been accomlished with any VPN most likely but I went with WireGuard since it is a simple and fast VPN whose setup is similar to SSH so it was inuitive for me to setup.

Host Setup

To use *Wireguard* inside of a LXC container the host has to have *Wireguard* installed since *LXC* guests are run with the kernel of the host system. *Wireguard* was first mainlined into the *Linux* kernel in version 5.6 so with kernel versions using 5.6 or later it is already built in. Anything before 5.6 that doesn't specifically have *Wireguard* backported in will need to use kernel modules to get it working. Wireguard.com has detailed instructions on how to install it on nearly any platform. Since I am using *Proxmox* as my host it was already backported into the kernel.

Guest Setup

Additionally I needed the `wireguard-tools` package in the *LXC* guest and `resolvconf` so DNS can be configured properly.

```
# echo "deb http://deb.debian.org/debian buster-backports main" > /etc/apt/sources.list.d/buster-backports.list
# apt update
# apt install --no-install-recommends wireguard-tools
# apt install resolvconf
```

Route Subnet Through Wireguard Interface

Funneling all traffic from an Ethernet interface through a *Wireguard* interface is relatively easy once I became familar with how packets flow through *Linux*. I mostly just needed to modify my *Wireguard* `*.conf` file to add the `Table`, `PostUp`, and `PreDown` parameters.

> I also needed to setup IP masquerading of outgoing traffic on my *Wireguard* interface. See below for instructions.

Create Interface

Creating the configuration file is a bit out of the scope of this document. A VPN provider that supports *Wireguard* will likely just provide a pre-built configuration file. But I also have a brief overview of how you'd make one.

```
# /etc/wireguard/warp.conf
[Interface]
PrivateKey = ****
Address = 10.10.20.59/19, 2a03:4012:4021:80af::1f3c/64
DNS = 10.10.0.1, 2a03:4012:4021:80af::1
Table = 9
PostUp = ip rule add iif eth1.9 lookup 9; ip -6 rule add iif eth1.9 lookup 9
PreDown = ip rule del iif eth1.9 lookup 9; ip -6 rule del iif eth1.9 lookup 9

[Peer]
PublicKey = T28Qn5VFzT4wiwEPd7DscwcP3Rsmq23QcnjH1N5G/wc=
Endpoint = wireguard.vpn-provider.example:51820
AllowedIPs = 0.0.0.0/0, ::0/0...
```

Line 5: All rules/routes should be applied to a custom route table `9`. I could have also named my custom route table by running `echo "9 warp" > /etc/iproute2/rt_tables` and then say `Table = warp` for

improved readability.

Line 6: Adds rules for IPv4 and IPv6 that all traffic coming in interface `eth1.9` should use custom route table `9`. Because I defined a peer with `AllowedIPs = 0.0.0.0/0, ::0/0` a default route will be setup on custom route table `9` that redirects all traffic to the *Wireguard* interface. If I named my custom route like shown above I could have said `lookup warp` inplace of `lookup 9`.

Line 7: Just the inverse of line 5 to clean up after myself when taking down the *Wireguard* interface.

Setup IP Masquerading

> IP Masquerading is a technique that hides an entire IP address space, usually consisting of private IP addresses, behind a single IP address in another, usually public address space.

Source: Wikipedia

Configuration

The easiest way to set this up are to append some *netfilter* rules to the `PostUp` and `PreDown` parameters.

```
...
PostUp = ...; iptables -t nat -A POSTROUTING -o wg0 -j MASQUERADE
PreDown = ...; iptables -t nat -D POSTROUTING -o wg0 -j MASQUERADE

[Peer]
...
```

Although this works fine there is a risk of the *iptables/netfilter* rules getting squashed by *Shorewall* if it is restarted while the *Wireguard* interface exists. It is best to have *Shorewall* setup the masquerading by making a simple declaration in `/etc/shorewall/snat`. I've included the other *Shorewall* configuration files that would be necessary to make this setup work.

First I define the `wg` zone...

```
# /etc/shorewall/zones
  #ZONE   TYPE    OPTIONS             IN                  OUT
```

```
 #                              OPTIONS              OPTIONS
 warp    ipv4
+ wg      ipv4
```

Then I define the interface `WG_IF` and put it in the `wg` zone...

```
# /etc/shorewall/interfaces
 #ZONE INTERFACE OPTIONS
 warp WARP_IF tcpflags,nosmurfs,routefilter=2,logmartians,physical=eth1.9
+ wg WG_IF physical=wg0
```

This tells Shorwall to masquerade all IPs going out on `WG_IF`...

```
# /etc/shorewall/snat
 #ACTION SOURCE DEST
+ MASQUERADE 0.0.0.0/0 WG_IF
```

Then I allow the `warp` zone to send packets to the `wg` zone. The `warp` zone isn't allowed to send packets to any other subnet or the `wan`. This prevents any data/privacy spills from happening if the *Wireguard* interface ever goes down. It is always best to fail into a state that protects security and privacy.

```
# /etc/shorewall/policy
 #SOURCE DEST POLICY         LOGLEVEL       RATE    CONNLIMIT
- warp $FW ACCEPT          $LOG_LEVEL
+ warp $FW,wg ACCEPT         $LOG_LEVEL
```

Remote Access

Allowing remote access is just a matter of setting up a new *Wireguard* interface, allowing incoming traffic to that interface, and making sure the firewall allows that traffic to connect to the rest of the network.

Create Interface

```
# cd /etc/wireguard
# umask 077
# wg genkey | tee guard.key | wg pubkey > guard.pub
# printf "[Interface]\PrivateKey = %s\n" `cat guard.key`
```

Then I modified my file to finish configuring the interface and allow a `[Peer]` for my laptop.

```
# /etc/wireguard/guard.conf
[Interface]
PrivateKey = ****
+ Address = 10.0.2.1/28, 2001:db8:2ebf:2::1/64
+ ListenPort = 51820
+
+ [Peer]
+ PublicKey = Iz5ceR0+tCN3BLTWehZxSplzdbABRT8geqifFxubHUA=
+ AllowedIPs = 10.0.2.4/32, 2001:db8:2ebf:1::4/128
+ PresharedKey = ***
```

Line 4: Sets an IPv4 and IPv6 address for this interface. These will be the servers IPs on each virtual subnet.

Line 5: Sets the port to listen to for this interface. It is just the default *Wirgaurd* port and I'll allow traffic through the firewall for it soon.

Line 7-10: Declare a peer, define the public key to use when communicating and validaing any connections, set what IPs the peer is allowed to use on each virtual subnet, and configure a pre-shared key for additional secuirty.

> A preshard key can be generated by running `wg genpsk` and must be the same on both the `[Peer]` block on the server and the `[Interface]` block on the client.

Firewall Configuration

First I had to declare a new interface and since I want it to be as if I was sitting on my laptop at home, I put it in the `lan` zone.

```
# /etc/shorewall/interfaces
...
 #ZONE□INTERFACE□OPTIONS
...
 wg□WGAZSE1_IF□tcpflags,nosmurfs,routefilter,logmartians,physical=wgazse1
+ lan□WGGUARD_IF□tcpflags,nosmurfs,routefilter,logmartians,physical=wgguard
```

```
# /etc/shorewall/interfaces
...
 #ZONE□INTERFACE□OPTIONS
...
 wg□WGAZSE1_IF□tcpflags,nosmurfs,routefilter,logmartians,physical=wgazse1
+ lan□WGGUARD_IF□tcpflags,forward=1,physical=wgguard
```

For outside clients to connect I need to add a rule that allows them to connect to the firewall on port 51820.

```
# /etc/shorewall[6]/rules
+ ACCEPT         wan,lan         $FW             udp             51820
```

The last step is to once again setup masquerading so traffic from clients on the *Wireguard* subnet appear to be originating from the `wgguard` interface which is in the `lan` zone.

```
# /etc/shorewall/snat
+ MASQUERADE□□10.0.2.0/28□□□□WAN_IF,LAN_IF,DMZ_IF
```

```
# /etc/shorewall6/snat
+ MASQUERADE□□fde9:2375:2ebf:2::/64□WAN_IF,LAN_IF,DMZ_IF
```

Bonded Interface

work in progress

Network Intrusion Detection

work in progress

Traffic Graphing/Monitoring

work in progress