Configuring Split Horizon DNS with Pi-Hole and Tailscale

Ben Tasker — 2025-07-05 16:05

I've long had some form of VPN for my devices to use when I'm out and about.

Although I used to run OpenVPN, I moved to Tailscale a little while back. Tailscale builds a mesh network using Wireguard protocol and so is able to connect and run quite a bit faster than OpenVPN.

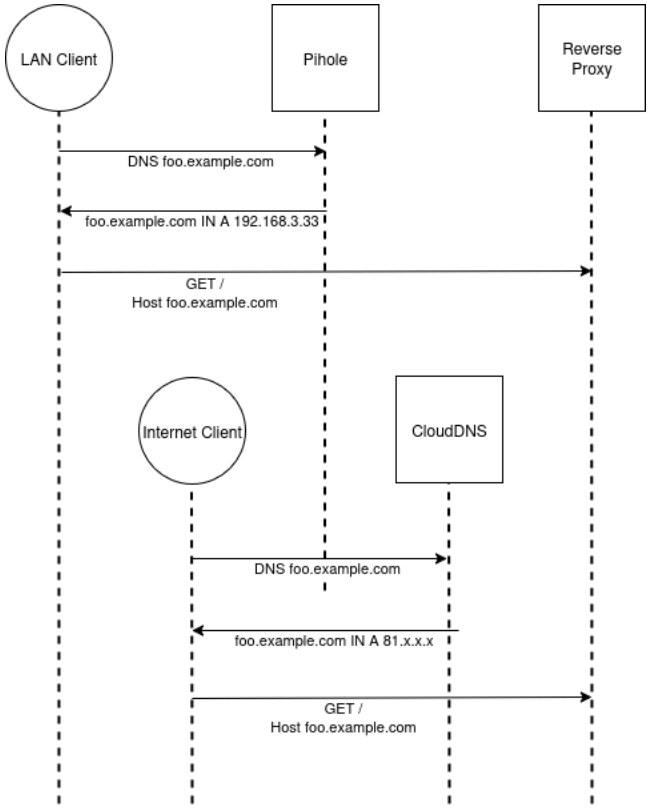
Side note: for those wondering, Tailscale is *Canadian* and can't see the content of connections (although if you're worried about this it's also possible to self-host using Headscale).

Although the tailnet has been up for some time, I hadn't got around to setting up split horizon DNS for clients on the tailnet. I was in a bit of a hurry when first setting up and so configured my reverse proxy box to advertise a route to it's own LAN IP.

This post talks about configuring my Pi-hole to implement a split horizon: returning the tailnet IP to tailnet clients and the LAN IP to LAN clients.

Splitting my Split Horizon

Many of the DNS names that I wanted to do this for *already* had a split horizon:



Clients on both the LAN and the wider internet connect to the same reverse proxy in my DMZ, but LAN clients connect using the proxy's local IP.

The reverse proxy fronts multiple services, most of which have authentication built in. However, it also requires that outside connections pass a separate (and valid) set of authentication credentials before it'll pass their connection on.

Having to authenticate twice is a little annoying though, and the split horizon makes it easy to disable the additional authentication when LAN clients connect:

```
satisfy any;
allow 192.168.3.0/24;
deny all;
auth_basic "Authenticate you must";
auth_basic_user_file /etc/nginx/wanaccess.htpasswd;
```

This extra authentication means that I'm not exposing any element of the backing service's authentication stack to the outside world. The underlying idea is that it *shouldn't matter* that there's an auth bypass zero day in (say) Grafana, because the wider world needs to get past my auth prompt before they can try to detect or exploit it.

You've Got Access: Why Make The Tailnet Special?

Given that there's an ability to access services via the WAN, you might be wondering why it is that I felt that I needed to do something specifically for the tailnet.

Unfortunately, the proxy *can't* enforce additional authentication for some services because those services clients don't support it.

Nextcloud is a great example of this: the Nextcloud Desktop sync client authenticates with Nextcloud, but

- It uses the `Authorization` header to present it's bearer token, so the reverse proxy will see an unexpected (and, to it, invalid) set of credentials
- The client doesn't expose a way to add custom headers to the requests that it makes, so I can't simply send a shared secret and have the proxy check a different header

Having the reverse proxy require additional auth breaks off-net Nextcloud clients (and Nextcloud isn't the only service with this issue).

Geoblocking

Originally, I left the affected services accessible to the world.

Unfortunately, I *sometimes* seem to upset people enough to trigger prolonged attempts at compromising my services.

After one such attempt, I decided to reduce attack surface by adding geo-blocking to my reverse proxy, essentially restricting access to areas that I thought we'd be likely to connect from (or *at least* appear to).

This, of course, comes at a cost in flexibility, with access failing if any of the following are true:

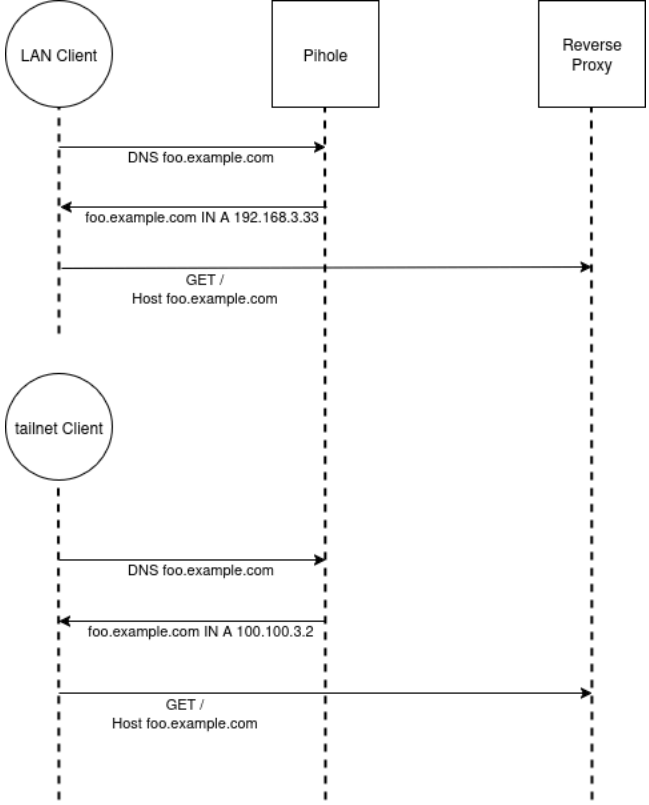
- We connected from an IP that doesn't have a location in the GeoDB (or is mislocated)
- The ISP that we're connecting from does funky routing stuff and/or uses CGNAT
- We've travelled somewhere that we wouldn't normally

Adding split horizon DNS to the tailnet allows me to avoid these scenarios, because the tailnet subnet can be special cased in *exactly* the same way that the LAN is.

It also increases the likelihood that I can close WAN access off and require that a client be on either the LAN or tailnet.

The Plan

The idea was that a tailnet client would also speak to the Pi-hole, but that names would resolve to a tailnet IP:



This is possible because Pi-hole is underpinned by a fork of `dnsmasq` called `pihole-FTL` which has inherited the setting `localise-queries` (in Pi-hole, this is enabled by default).

The man page for dnsmasq describes the setting as follows (line breaks mine):

> Return answers to DNS queries from /etc/hosts and --interface-name and --dynamic-host which depend on the interface over which the query was received.
>
> If a name has more than one address associated with it, and at least one of those addresses is on the same subnet as the interface to which the query was sent, then return only the address(es) on that subnet and return all the available addresses otherwise.
>
> This allows for a server to have multiple addresses in /etc/hosts corresponding to each of its interfaces, and hosts will get the correct address based on which network they are attached to.
>
> Currently this facility is limited to IPv4.

This means that we can create the following record set in `/etc/pihole/custom.list`:

```
192.168.3.33 foo.example.com
100.100.3.2  foo.example.com
```

If a query is received over an interface in one of these subnets, only the matching record will be returned (otherwise, both will be returned):

Receiving Interface IP	Response
192.168.3.13/24	192.168.3.33
100.100.3.13/24	100.100.3.2
10.8.0.0/24	192.168.3.33, 100.100.3.2

One small drawback with this is that the records must be in the `hosts` format file - most of my records were in dnsmasq format files, so I had to migrate the ones that I wanted to split.

Re-Jigging My Docker Container

There was, however, a catch.

When I first created my pihole container, the docker invocation looked something like this:

```
docker run \
-d \
--name=pihole \
--hostname=pihole \
--restart=unless-stopped \
--e ServerIP=0.0.0.0 \
--e WEBPASSWORD='NotMyRealPass' \
-v $PWD/pihole/conf:/etc/pihole \
-v $PWD/pihole/dnsmasq.d:/etc/dnsmasq.d/ \
-p 53:53 -p 53:53/udp \
-p 8080:80 \
pihole/pihole
```

This meant that the container was using bridged networking, depriving Pi-hole of the means to see which physical interface a query arrived on: it simply saw the other side of a single bridge interface.

So, I killed the container and started a new one using host networking:

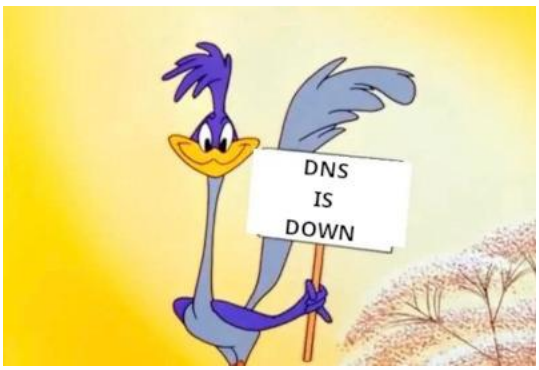
```
docker run \
-d \
--network=host \
--name=pihole \
--hostname=pihole \
--restart=unless-stopped \
-e ServerIP=0.0.0.0 \
-e WEBPASSWORD='NotMyRealPass' \
-v $PWD/pihole/conf:/etc/pihole \
-v $PWD/pihole/dnsmasq.d:/etc/dnsmasq.d/ \
pihole/pihole
```

However the container failed to start: Pihole's web interface was trying to bind to port 80 which already had something bound to it.

As I'd previously mapped 8080 into the container (`-p 8080:80`), I used the environment variable `WEB_PORT` to tell Pi-hole to bind to that port instead:

```
docker run \
-d \
--network=host \
-e WEB_PORT=8080 \
--name=pihole \
--hostname=pihole \
--restart=unless-stopped \
--env=ServerIP=0.0.0.0 \
--env='WEBPASSWORD=NotMyRealPass' \
-v $PWD/pihole/conf:/etc/pihole \
-v $PWD/pihole/dnsmasq.d:/etc/dnsmasq.d/ \
-p 53:53 -p 53:53/udp \
-p 8080:80 \
pihole/pihole
```

DNS Outage



Pi-hole came up, but it wasn't responding to queries.

Netstat showed `pihole-FTL` listening and bound to all interfaces:

```
$ sudo netstat -lnp | grep :53
tcp        0      0 0.0.0.0:53              0.0.0.0:*               LISTEN      2653543/pihole-FTL
tcp6       0      0 :::53                   :::*                    LISTEN      2653543/pihole-FTL
udp        0      0 0.0.0.0:53              0.0.0.0:*                           2653543/pihole-FTL
udp6       0      0 :::53                   :::*                                2653543/pihole-FTL
```

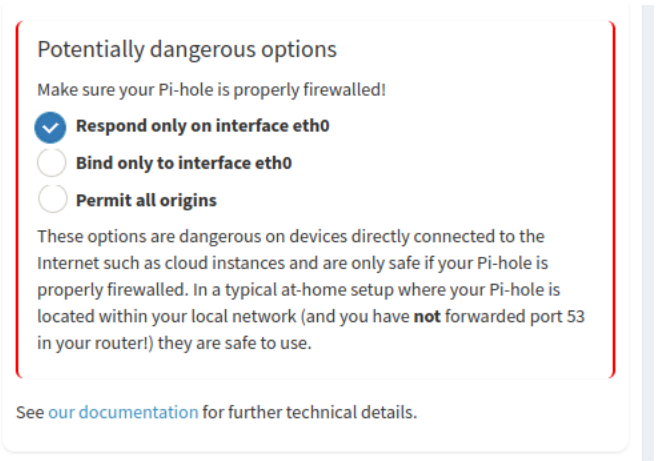
Packet captures showed that queries were coming in, but no responses were being sent.

```
$ sudo tcpdump -i any port 53
21:54:02.345555 enp0s25 In  IP 192.168.3.163.32273 > 192.168.3.5.53: 57965+ A? n-deventry.tplinkcloud.com. (44)
21:54:02.512870 enp0s25 In  IP 192.168.3.44.63761 > 192.168.3.5.53: 26967+ AAAA? lycraservice-pa.googleapis.com.home. (5
21:54:02.524346 enp0s25 In  IP 192.168.3.44.1270 > 192.168.3.5.53: 2692+ A? lycraservice-pa.googleapis.com.home. (53)
21:54:02.767189 enp0s25 In  IP6 2001:820:aa1a:c443:b9c4:44b:df15:bd8e.36925 > 2001:820:aa1a:c443::2.53: 28460+ A? a.nel.cloudflare
21:54:02.767189 enp0s25 In  IP6
```

Queries weren't triggering any activity in Pihole's logs either.

To restore service to the LAN, I killed the container and brought it back up with bridged networking - DNS sprang straight back to life.

It took me a while to figure out what the issue was, but eventually I spotted this setting in Pi-hole's web interface:



Pi-hole was configured to only respond to queries received from interface `eth0`. Resolution stopped because the box that I run pihole on doesn't have an `eth0` (it's a `udev`'y style `enp0s25`).

I switched this to `Permit all origins` and restarted the container with host networking. This time, queries were answered.

Configuring Tailscale

The box hosting pihole was already part of the tailnet, but I wanted to remove the previous route advertisement.

So I ran

```
sudo tailscale down

# Previously this was
# --advertise-routes=192.168.3.33/32
sudo tailscale set --advertise-routes=

sudo tailscale up
```

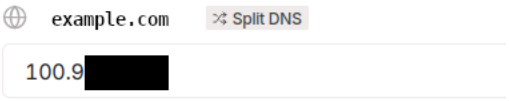
Then, from another tailnet client (my laptop), I tried resolving a name via both the LAN and tailnet address:

```
$ dig +short foo.example.com @100.99.55.55
100.100.3.2

$ dig +short foo.example.com @192.168.3.13
192.168.3.33
```

All that was left was to have tailnet clients actually use Pihole.

I logged into Tailscale's web interface and added a Split DNS entry:



When bringing tailscale up on my Linux laptop, I had to explicitly pass a flag to allow it to use the advertised server

```
sudo tailscale up --accept-dns
```

The android app has a toggle for this, but it was already on.

Conclusion

My devices now have transparent (and slightly more privileged) access to services when I'm out and about.

Because Tailscale acts as a mesh network, I don't need to worry about automatically turning the VPN off when I'm at home - devices in the same segment can direct connect to one another rather than making a round-trip via a remote coordinator.

As a result of getting this up and running, I've been able to close off WAN access to a number of services (although I still can't can't do that for any service which hosts something I might try to cast, because Chromecasts ignore local DNS... grrr).

It all works well enough that I've been able to write, proof-read and publish this post whilst off net.

As an added bonus, Tailscale seem to have partnered with Mullvad, so if I'm ever *travelling* travelling, I can have my devices route all connections via Mullvad and my tailnet.

dns howto pihole privacy tailscale vpn

Previous post