Pragmatic IP Networking Guide

Lorenz Schori

Jan 24, 2022

CONTENTS

This short guide presents one of many ways to structure small to medium home and enterprise IP networks. A special focus lies on low complexity, support for privacy preserving technology and good network and endpoint security.

CHAPTER

ONE

INTRO

This short guide presents one of many ways to structure small to medium home and enterprise IP networks. A special focus lies on low complexity, support for privacy preserving technology and good network and endpoint security. Especially:

- The network works with end-to-end encrypted connections without modifications on the endpoints (no mandatory proxies, no custom certificate authorities).
- The network works regardless of whether endpoints use their own stub resolvers or third-party DNS services, it works for validating resolvers (DNSSEC) and it works for endpoints and applications using DoT and DoH (no split horizon DNS).
- The network allows endpoints to make use of IPv6 privacy extension or similar anti-tracking measures.

CHAPTER

TWO

REQUIREMENTS

This guide is not a tutorial. Readers should be familiar with IPv4 and IPv6 addressing and terminology.

Throughout this guide we assume that the internet uplink provides native IPv6 connectivity with a prefix sized large enough such that it can be subdivided into multiple `/64` networks. Respectable ISPs will assign static IPv6 prefixes between a `/60` (min) and `/48` (max) either by default or on request.

Deployments stuck with an IPv4 only uplink or with dynamic / dysfunctional IPv6 connectivity may choose to upgrade to static IPv6 with the help of a tunnel broker service.

CHAPTER

THREE

RECOMMENDATIONS

3.1 Segments

Unless there is only one client, networks need to be segmented into multiple subnets on different VLANs. Different classes of nodes require different services from the network. Separating them makes it easier to meet those requirements.

Use separate VLANs for (non exhaustive list):

- Printers
- IoT devices
- VoIP phones
- Servers, VMs, Containers
- Wi-Fi controllers and access points
- Laptops and Workstations
- Temporary devices / Guests
- Experiments

Hint: Segments and Autoconfiguration

Do not mix statically addressed and autoconfigured nodes in the same subnet.

Laptops and Workstations typically are configured using router advertisements and/or DHCP while static addresses should be assigned to servers. Keeping dynamic and static clients in separate subnets prevents potential address collision without additional configuration overhead. It also allows for servers to deactivate DAD (duplicate address detection).

Hint: Dual stacking only when needed

Printers placed in a separate subnet are easier to isolate from the internet (if necessary). In some situations, IPv4 might not be needed at all for a printer subnet, hence such a network segment can be operated using IPv6 exclusively. Same goes for other internal-only services (e.g., media server).

3.2 Networks and Routes

3.2.1 Subnets

An IPv6 prefix with a size of `/60` can be divided into 16 `/64` subnets, a prefix with the size of `/56` has space for 256 subnets and a `/48` prefix holds 65536 subnets. This guide is going to use `2001:db8:1020:ff00::/56` as the block used to allocate subnets from. Note, the IPv6 address range is assigned by the upstream provider. It cannot be chosen freely.

If IPv4 is still a thing when you read this guide, choose an IPv4 block from RFC 1918 big enough to be split into several `/24` networks. It is best to avoid blocks which include ranges used broadly in factory defaults like `192.168.0.0/24`, `192.168.1.0/24` and `10.0.0.0/24`. Instead opt for lesser used IPs. This guide is going to use `10.20.0.0/16` as the block used to allocate subnets from.

Hint: VLAN id == subnet id

When creating new network segments, choose the VLAN id to be identical to the subnet id. E.g. for a new network segment with VLAN id 8 allocate `2001:db8:1020:ff08::/64` for the IPv6 range and `10.20.8.0/24` for IPv4. For a new network segment with VLAN id 33 (hex: 0x21) allocate `2001:db8:1020:ff21::/64` for the IPv6 range and `10.20.33.0/24` for IPv4. Note: If the IPv6 prefix is a `/48`, then it is also possible to use the VLAN id in decimal notation for the IPv6 subnet.

3.2.2 Default Gateway

IPv6 routers advertise their link-local address and not a globally routable one (see RFC 4861 section 6.1.2). Hence, the default gateway on connected nodes is supposed to point to a link local address (i.e., an address within `fe80::/64`). Usually, IPv6 link-local addresses are derived automatically from the MAC address of the network interface. However, using generated addresses is not very practical in server subnets where hosts are configured statically. Therefore it is advisable to manually configure the link-local address for every subnet on routers.

Hint: Use fe80::1/64

Fortunately link-local addresses are link-local. Therefore, it is perfectly valid to use the same address (i.e., `fe80::1/64`) on every interface of a router (see: Blog post: fe80::1 is a Perfectly Valid IPv6 Default Gateway Address)

3.3 Names

Most people cannot be bothered to remember IPv4 addresses. IPv6 does not make things any easier.

3.3.1 DNS Hosting

DNS zones need to be hosted somewhere. SOHO routers typically provide point and click interfaces where DNS entries can be added to some internal DNS server which typically doubles as recursive DNS resolver. In order for this to work, the address of that internal DNS server needs to be configured (maunally or via autoconfiguration) on all nodes who wish to resolve those entries.

This configuration is known as split-horizon DNS. And it falls short if people start to use validating resolvers (DNSSEC) and alternative DNS resolvers, sometimes over encrypted protocols (DoT, DoH).

Hint: Place RRs in public DNS

Avoid deploying private DNS zones and split-horizon DNS. Instead place all resource records into public DNS.

3.3.2 Node Names

Every node which provides any type of service should have a name. This includes routers, servers, VMs, containers, printers, WiFi access points etc. A popular choice is to just add `AAAA` records to the domain name of the family website or the primary domain of a business.

While convenient in the beginning, this can pose problems down the road. The website might be managed by contractors while the network stays inhouse or vice-versa. Node records might be managed by an orchestrator while access to the DNS zone of the main website needs to be restricted due to policy reasons. Maintaining separate DNS zones for different purposes also simplifies gradual rollouts, e.g. of DNSSEC.

Hint: Use a dedicated domain for nodes

Thus it is recommended to register and maintain a dedicated domain and only add `AAAA` records for network nodes there. Additional records like `SSHFP` could be added as well, this will simplify node administration greatly.

Nodes need to be replaced over time. In order to simplify this process, old names should not be reused for new nodes. Instead each node keeps its name over its whole lifespan in a network. Holding on to this practice simplifies the development of a network since old and new equipment can be operated in parallel for some time.

3.3.3 Service Names

Every service should have a name. This includes webapps, file sharing, directory services, etc. A popular choice is to just use the node name where the service happens to be hosted.

Services need to keep their name, otherwise people are forced to update bookmarks and printer queues. It follows that reusing node names for services will pose problems in the long run when nodes need to be replaced.

In addition, services might be composed from several applications running on different nodes, VMs or containers. The service name is then simply pointing towards the node hosting the frontend server for TLS termination, reverse proxying and/or load balancing.

Hint: Use a dedicated domain for services

Thus it is recommended to register and maintain a dedicated domain for internal services. Either add `AAAA` records containing IPs of the nodes hosting a service or add `CNAME` pointing towards the node names.

Caution: Service enumeration via CT logs

TLS certificates issued by trusted certificate authorities are recorded in public certificate transparency logs (e.g. crt.sh). Organisations which are reusing subdomains of their main website or brand name for internal systems secured by TLS certificates might unknowingly expose this information to the public.

Using a dedicated domain name for internal services unrelated to the main website, name or brand of an organisation and deploying wildcard TLS certificate can reduce the risk of service enumeration via certificate transparency logs.

3.4 Addressing

Nodes providing services to connected clients need a fixed IP address. IPv6 addresses assigned automatically via SLAAC do not change over time. Thus, such IPs are quite suitable as a stable identifier for a given node connected to a specific network.

However, SLAAC can be problematic when used with servers and VMs. Some operating systems will not wait for autoconfiguration to complete and some server software will either fail to start or even fall back to the loopback interface to listen on when the primary interface is not ready early enough upon startup. Due to those potential race conditions it is recommended to use static IP configuration on servers.

Hint: Maintain the SLAAC IP for printers in DNS

In order to be easily reachable from clients, the SLAAC IP of every printer should be recorded in the DNS zone for nodes and a separate record should be maintained in the DNS zone for services pointing to the respective node name.

Using SLAAC for printers spares administrators from the tedious exercise to input verbose IPv6 addresses via single button interfaces.

Hint: Use a predictable addressing scheme for servers

An IPv6 `/64` subnet has room for 18446744073709551616 addresses. The wast size of those subnets opens the opportunity to discourage host discovery by network scans (there are many pitfalls though, see RFC 7707). In order to avoid clustering hosts around likely scanned ranges, one could use a cryptographic hash of the hostname as the basis for an IP address. Implementations of this method are available as an ansible filter (znerol/ipaddr_hash on galaxy.ansible.com) and as a JavaSrcipt isomorphic library (ipaddrhash on npmjs.com). For convenience, the fully functional example webapp can be found and used at znerol.github.io/ipaddrhash-js/

CHAPTER

FOUR

LINUX ROUTER EXAMPLE

A setup as outlined in chapter *Recommendations* can be implemented using tools built into modern Linux distros by default. Most single-router scenarios can be covered with `systemd-networkd` and `nftables` exclusively.

All configuration files are available in the github repo under examples/01-linux-router.

4.1 Network Layout

Let's assume that the ISP assigned the IPv6 prefix `2001:db8:1020:ff00::/56`. Also this example uses `10.20.0.0/16` to allocate IPv4 subnets from.

Table 1: Interfaces, Subnet IDs (VLANs) and IP Ranges

Interface	Role	Subnet ID (VLAN)	IPv6 Prefix	IPv6 Address
ens1	wan	none	Assigned via SLAAC/DHCP6	Assigned via DHCP4
ens2	trunk	none	none	none
vlan-dmz	dmz	158 (0x9e)	2001:db8:1020:ff9e::/64	none
vlan-guest	guest	214 (0xd6)	2001:db8:1020:ffd6::/64	10.20.214.1/24
vlan-staff	staff	84 (0x54)	2001:db8:1020:ff54::/64	10.20.54.1/24
vpn	staff	244 (0xf4)	2001:db8:1020:fff4::/64	10.20.244.1/24

4.2 Systemd network configuration

Network configuration is maintained in systemd.network(5) unit files under `/etc/systemd/network`. The presented configuration makes extensive use of per-network `drop-in` directories. This simplifies reuse of common configuration snippets.

```
$ tree network
network
├── lo.network
├── lo.network.d
│   ├── iface-type-loopback.conf
│   └── inet-lo.conf
├── trunk.network
├── trunk.network.d
│   ├── child-vlan-dmz.conf
│   ├── child-vlan-guest.conf
│   ├── child-vlan-staff.conf
│   └── iface-type-trunk.conf
```

(continues on next page)

(continued from previous page)

```
├── vlan-dmz.netdev
├── vlan-dmz.network
├── vlan-dmz.network.d
│   ├── iface-type-router.conf
│   └── inet-vlan-dmz.conf
├── vlan-guest.netdev
├── vlan-guest.network
├── vlan-guest.network.d
│   ├── iface-service-dhcp4.conf
│   ├── iface-service-router-adv.conf
│   ├── iface-type-router.conf
│   └── inet-vlan-guest.conf
├── vlan-staff.netdev
├── vlan-staff.network
├── vlan-staff.network.d
│   ├── iface-service-dhcp4.conf
│   ├── iface-service-router-adv.conf
│   ├── iface-type-router.conf
│   └── inet-vlan-staff.conf
├── vpn.netdev
├── vpn.netdev.d
│   ├── peer1.example.com.conf
│   └── peer2.example.com.conf
├── vpn.network
├── vpn.network.d
│   └── inet-vpn.conf
├── wan.network
└── wan.network.d
    └── inet-wan.conf

8 directories, 31 files
```

4.2.1 Interface: ens1 / wan (autoconfigured via SLAAC/DHCP)

The wan network consists of the `wan.network` unit file (containing the `Match` section) and the `wan.network.d/inet-wan.conf` drop-in (specifying how ip4/ip6 addressing is performed).

Listing 1: wan.network

```
[Match]
Name = ens1
```

Listing 2: wan.network.d/inet-wan.conf

```
[Network]
IPv6AcceptRA = yes
DHCP = yes
```

4.2.2 Interface: ens2 / trunk

The trunk network consists of the `trunk.network` unit file (containing the `Match` section) and several drop-ins.

Listing 3: trunk.network

```
[Match]
Name = ens2
```

`trunk.network.d/iface-type-trunk.conf` simply disables IPv6 link-local addresses.

Listing 4: trunk.network.d/iface-type-trunk.conf

```
[Network]
LinkLocalAddressing = no
IPv6AcceptRA = no
```

For each VLAN, a `child-vlan-XXX.conf` ensures that the specified VLAN is added to the trunk interface.

Listing 5: trunk.network.d/child-vlan-dmz.conf

```
[Network]
VLAN = vlan-dmz
```

Listing 6: trunk.network.d/child-vlan-guest.conf

```
[Network]
VLAN = vlan-guest
```

Listing 7: trunk.network.d/child-vlan-staff.conf

```
[Network]
VLAN = vlan-staff
```

4.2.3 VLAN: vlan-dmz (IPv6 only, static addressing)

VLAN devices are created using systemd.netdev(5) units.

Listing 8: vlan-dmz.netdev

```
[NetDev]
Name = vlan-dmz
Kind = vlan
```

(continues on next page)

(continued from previous page)

```
[VLAN]
Id = 158
```

The specified device name is then used in the `Match` section of the corresponding `network` unit.

Listing 9: vlan-dmz.network

```
[Match]
Name = vlan-dmz
```

As pointed out in chapter *Recommendations* it can be beneficial to use a fixed `fe80::1` link-local address on router interfaces. The drop-in `iface-type-router.conf` provides the necessary settings. Additionally it disables acceptance of router advertisements on this interface and enables forwarding.

Listing 10: vlan-dmz.network.d/iface-type-router.conf

```
[Network]
IPv6LinkLocalAddressGenerationMode = none
Address = fe80::1/64

IPv6AcceptRA = no

IPForward = yes
```

The second drop-in `inet-vlan-dmz.conf` adds a route to the DMZ subnet (`2001:db8:1020:ff9e::/64`) to this interface. Note that link-local addresses are used for routing in IPv6. Hence, it is not necessary to actually assign a globally routed address to router interfaces.

Since this network segment is IPv6 only, there is no need to add IPv4 addresses / routes to this interface.

Listing 11: vlan-dmz.network.d/inet-vlan-dmz.conf

```
[IPv6Prefix]
Prefix = 2001:db8:1020:ff9e::/64

[Route]
Destination = 2001:db8:1020:ff9e::/64
```

4.2.4 VLAN: vlan-staff (Dual-stack, SLAAC und DHCP4)

Base files like `vlan-staff.netdev` and `vlan-staff.network` work analogous to the example above. The `iface-type-router.conf` drop-in can be reused without modification. Since this interface needs to provide IPv4 connectivity, an appropriate address needs to be supplied via `inet-vlan-staff.conf` drop-in.

Listing 12: vlan-staff.network.d/inet-vlan-staff.conf

```
[IPv6Prefix]
Prefix = 2001:db8:1020:ff54::/64

[Route]
Destination = 2001:db8:1020:ff54::/64
```

(continues on next page)

(continued from previous page)

```
[Network]
Address=10.20.84.1/24
```

Another set of drop-ins is used to configure router advertisements:

Listing 13: vlan-staff.network.d/iface-service-router-adv.conf

```
[Network]
IPv6SendRA = yes
DHCPv6PrefixDelegation = no

[IPv6SendRA]
RouterLifetimeSec = 1800
DNSLifetimeSec = 1200

EmitDNS = yes
DNS = 2606:4700:4700::1111 2606:4700:4700::1001
```

And DHCP4 server:

Listing 14: vlan-staff.network.d/iface-service-dhcp4.conf

```
[Network]
DHCPServer = yes

[DHCPServer]
EmitDNS = yes
DNS = 1.1.1.1 1.0.0.1
```

Note that neither `iface-service-router-adv.conf` nor `iface-service-dhcp4.conf` contain any interface specific configuration. Hence, they can be reused again for the `vlan-guest` interface.

4.2.5 Wireguard: vpn

Wireguard specific settings are mantained in `netdev` units. This includes key material, the listen port and peer definitions.

Listing 15: vpn.netdev

```
[NetDev]
Name = vpn
Kind = wireguard
Description = wireguard server

[WireGuard]
ListenPort = 51820
PrivateKeyFile = /etc/wireguard/private.key
```

In order to simplify management of peers, configuration for each peer should be maintained in a separate `drop-in` file.

Listing 16: vpn.netdev.d/peer1.example.com.conf

```
[WireGuardPeer]
PublicKey = xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
AllowedIPs = 10.20.244.170/32, 2001:db8:1020:fff4:f4c1:f2ed:a58f:a3aa/128
```

Listing 17: vpn.netdev.d/peer2.example.com.conf

```
[WireGuardPeer]
PublicKey = TrMvSoP4jYQlY6RIzBgbssQqY3vxI2Pi+y71lOWWXX0=
AllowedIPs = 10.20.244.241/32, 2001:db8:1020:fff4:9224:9412:67c5:f9f1/128
```

Caution: netdev configuration cannot be reloaded

Most configuration can be applied at runtime using `networkctl reload`. However, configuration for `netdev` units is only applied upon creation of virtual devices. As a result, in order to apply wireguard configuration after a peer was added or removed, it is regrettably necessary to completely remove the wireguard interface before `networkd` picks up the new config. I.e.:

```
ip link del vpn
networkctl releoad
```

See systemd/systemd#9627 for more details.

Network configuration via network units / drop-ins for wireguard interface follows the same pattern as all examples presented here. The `Match` section matches the name from the `netdev` unit. The `inet-vpn.conf` drop-in adds IPv6 and IPv4 addresses acting as the gateway for connected clients.

Listing 18: vpn.network

```
[Match]
Name = vpn
```

Listing 19: vpn.network.d/inet-vpn.conf

```
[Network]
Address = 10.20.244.1/24
Address = 2001:db8:1020:fff4::1/64
```

4.2.6 Loopback: lo

Note that no static globally routable IPv6 address was assigned to any interface (except for the vpn gateway). In order to access services on the router (including SSH), a static IPv6 address needs to be present at some interface.

Networking folks developed the habit to assign a routable IP on the loopback interface. This is especially useful on nodes with many interfaces in the context of dynamic routing. The loopback interface never goes down, and thus an IP assigned to `lo` will be reachable as long as there is at least one route.

Analogous to earlier examples `lo.network` simply matches the loopback device. The `iface-type-loopback.conf` drop-in is responsible for device type specific config. Notable `KeepConfiguration = static` preserves existing IP addresses and routes (i.e., `127.0.0.1/8` and `::1/128` configured during system bootup).

Listing 20: lo.network.d/iface-type-loopback.conf

```
[Network]
KeepConfiguration = static
IPv6LinkLocalAddressGenerationMode = none
IPv6AcceptRA = no
```

The `inet-lo.conf` drop-in just assigns the routers IP:

Listing 21: lo.network.d/inet-lo.conf

```
[Network]
Address = 2001:db8:1020:ff39:5ed7:b1d4:c5d:e994/128
```

This IP should be recorded in DNS. It can be used to `ping` and `ssh` from wherever there is IPv6 connectivity - as long as filter rules allow it.

4.3 Nftables Ruleset

All configuration files are available in the github repo under examples/01-linux-router.

Documentation on nftables regrettably isn't that comprehensive yet. The nft(8) manpage provides up-to-date reference material. Some usage examples and recipes are available from the nftables wiki and also from various Linux distro wiki pages (quality of content varies). It is essential to understand packet flow through netfilter hooks and to keep in mind the following rule when reasoning about rulesets:

Hint: Evaluation of Rules

In order to be delivered, a packet must be `accept`ed by every base chain in all traversed hooks.

A packet is discarded immediately as soon as it is `drop`ed or `reject`ed. None of the rules in later chains and hooks will have the opportunity to further handle it.

It is possible to leverage this behavior and design rulesets which are quite modular and easy to maintain by isolating reusable logic into generic tables and chains.

4.3.1 nftables.conf

The main entrypoint is `/etc/nftables.conf` which simply includes definitions and tables in the correct order. Note that with this design, features can be added to the firewall by simply dropping more table files into the appropriate directory.

Listing 22: nftables.conf

```
flush ruleset

include "/etc/nftables/defines/*.nft"
include "/etc/nftables/tables/*.nft"
```

The rest of the configuration gets collected using includes from `/etc/nftables` directory:

```
nftables
├── defines
│   ├── nics.nft
│   └── zones.nft
├── inet-filter
│   ├── hook-forward-filter.nft
│   ├── hook-input-filter.nft
│   └── hook-output-filter.nft
├── inet-lib
│   ├── chains-autoconf.nft
│   └── chains-essentials.nft
├── inet-zones
│   ├── zone-autoconfiguration.nft
│   ├── zone-management.nft
│   ├── zone-public.nft
│   └── zone-wan.nft
└── tables
    ├── inet-filter-martians.nft
    ├── inet-filter.nft
    └── ip4-nat.nft

5 directories, 14 files
```

Main entry points are the tables, thus let's go through these first.

4.3.2 Martians

Reverse path filtering (aka uRPF, aka BCP38, aka RFC 2827) can be implemented using nftables for both IPv4 and IPv6. As long as routes are symetric, the following ruleset will ensure that packets entering a given interface do have a plausible source address.

Listing 23: nftables/tables/inet-filter-martians.nft

```
# Implements BCP38 / RFC2827 / reverse path filtering using the fib.
# * http://www.bcp38.info/index.php/Main_Page
# * https://datatracker.ietf.org/doc/html/rfc2827
# * https://manpages.debian.org/bullseye/nftables/nft.8.en.html#FIB_EXPRESSIONS
#
# Note: Applies to all interfaces on the system.
table inet managed-by-ansible.inet-filter-martians {
    chain prerouting-raw-rpfilter {
        type filter hook prerouting priority raw; policy drop;

        # Lookup the tuple (saddr, iif) in the fib and extract the oif from the
        # resulting entry. Accept the packet if that information exists.
        fib saddr . iif oif exists accept

        log group 0 prefix "prerouting-raw-rpfilter:drop-martian"
    }
}
```

This example uses the `fib` (forward information base). The nftables wiki has additional examples on matching routing information.

Hint: Table names and chain names

Tables are merely containers for chains and associated state. Chains are containers for rules. Their names do not have any significance during rule execution. Only the table family (e.g., `inet`) and the `type ... hook ... priority` line are relevant (and the rules of course).

It might help to think about tables as namespaces. Names can help avoid collisions when combining tables from multiple sources in one ruleset and they make it easier to navigate the output of `nft list ruleset`.

4.3.3 Zoned Firewall

Chains included in `inet-filter.nft` file form a flexible zoned firewall.

Listing 24: nftables/tables/inet-filter.nft

```
table inet managed-by-ansible.inet-filter {
    include "/etc/nftables/inet-lib/*.nft"
    include "/etc/nftables/inet-zones/*.nft"
    include "/etc/nftables/inet-filter/*.nft"
}
```

The goal of the presented design is that new VLANs can easily be added to existing zones (via the `zones.nft` file). Also adding new rules to existing zones is a matter of adding them to the appropriate chain in one of the `zone-XX.nft` files.

Listing 25: nftables/defines/zones.nft

```
# interfaces with autoconfigurated clients
define zone_autoconfig = {
    $nic_guest,
    $nic_staff,
}

# management zone
define zone_management_source = {
    $nic_staff,
    $nic_vpn,
}
define zone_management_dest = {
    $nic_dmz,
}

# public services zone
define zone_public_source = {
    $nic_guest,
    $nic_staff,
    $nic_wan,
}
define zone_public_dest = {
    $nic_dmz,
}
```

(continues on next page)

(continued from previous page)

```
# restricted wan access zone
define zone_restricted_wan_source = {
    $nic_dmz,
}

# unrestricted wan access zone
define zone_unrestricted_wan_source = {
    $nic_guest,
    $nic_staff,
    $nic_vpn,
}
```

Some zones are only relevant in `input` and `output` hooks (e.g., `autoconfiguration`). Others are used when `forward`-ing traffic (e.g., `public`) and some are hooked into `input` and `forward` (e.g., management).

Forward zones are directional, hence it is necessary to define two sets of interfaces (source and dest).

Also note that interfaces can be part of multiple zones. E.g., `nic_dmz` is a *destination* for traffic in the management zone and in the public zone, and at the same time it is a *source* in the *restricted_wan* zone.

In some zones, there is by definition only one destination interface (e.g., in the *wan* zones). In that case, an explicit set of interfaces can be omitted and the interface name is used directly in the base chains.

Base chains are defined in `hook-forward-filter.nft`, `hook-input-filter.nft` and `hook-output-filter.nft`. Note that base chains are named according to the following pattern: *<hook-name>-<priority-keyword>*.

Listing 26: nftables/inet-filter/hook-forward-filter.nft

```
chain forward-filter {
    type filter hook forward priority filter; policy drop;

    # Accept established connections, drop invalid ones.
    jump global-conntrack-essentials

    # Forward zone: management
    # Accept selected traffic from management zone to managed services zones.
    iifname $zone_management_source oifname $zone_management_dest \
        jump forward-management

    # Forward zone: public services
    # Accept selected traffic from public clients zones to public services zones.
    iifname $zone_public_source oifname $zone_public_dest \
        jump forward-public

    # Forward zone: unrestricted wan access
    # Accept all traffic from unrestricted zones to wan.
    iifname $zone_unrestricted_wan_source oifname $nic_wan accept

    # Forward zone: restricted wan access
    # Accept selected traffic from restricted zones to wan.
    iifname $zone_restricted_wan_source oifname $nic_wan \
        jump forward-restricted-to-wan

    # Log unmatched to NFLOG group 0
```

(continues on next page)

(continued from previous page)

```
    log group 0 prefix "forward-filter:drop-default"
}
```

The structure of the forwarding rules is quite simple. In a stateful firewall, the first thing to check is conntrack metadata. The following rules simply match input and output interface metadata and apply rules defined for the respective zones.

Note, it is possible to use `iif` and `oif` instead of `iifname` and `oifname`. The former matches interface index and the latter the interface name. It follows that the short syntax only can be used if all interfaces are brought up at boot time and never change during runtime. The long syntax is useful when interfaces are created dynamically. E.g. for PPP(oE) uplinks.

Zone files can be quite simple. The `public` forwarding zone consists of one chain with one rule. More rules (e.g., for UDP traffic) could be added easily.

Listing 27: nftables/inet-zones/zone-public.nft

```
# Destination zone public (public hostings)

# Rules evaluated for all traffic entering this zone originating from public clients␣
↪zone.
chain forward-public {
    tcp dport { http, https } ct state new accept
}
```

Note `ct state new` matches conntrack metadata `new` state. After `global-conntrack-essentials`, it is not strictly required to explicitly match that (since everything else was already accepted or dropped). Stating `ct state new` explicitly on `tcp` rules is a matter of good style.

The management zone is a bit more complex since it is referenced from the `forward` as well as from the `input` hooks.

Listing 28: nftables/inet-zones/zone-management.nft

```
# Source zone management (client machines allowed to manage the hosts and network)

# Rules evaluated for all traffic leaving this zone directed towards managed services␣
↪zone.
chain forward-management {
    # Accept management.
    tcp dport { ssh, http, https } ct state new accept

    # Accept pings.
    icmp type echo-request accept
    icmpv6 type echo-request accept
}

# Rules evaluated for all traffic from this zone directed at this host.
chain host-input-management {
    # Accept management.
    tcp dport ssh ct state new accept

    # Accept pings.
    icmp type echo-request accept
    icmpv6 type echo-request accept
}
```

More examples are available in the github repo under examples/01-linux-router.

4.3.4 IPv4 NAPT

Network address and port translation for IPv4 can be added using another drop-in table. Note that this table is restricted to IPv4 (the `ip` family is specified in the table definition),

Listing 29: nftables/nftables/tables/ip4-nat.nft

```
table ip managed-by-ansible.ip4-nat {
    chain postrouting-srcnat {
        type nat hook postrouting priority srcnat;

        oifname $nic_wan masquerade
    }
}
```