Hiding your network behind N external IP addresses

José María Foces Vivancos

Whoami

- Computer freak since I can remember
- IT Security Engineer
- 2012 – 2014 Software developer by Indra @ Incibe
- 2014 – 2017 Pentester – Security Analyst & Architect
- 2017 – Now Security Specialist @ Telefónica - CSIRT-ES
- https://github.com/Jmfoces

Intro

- Starting point:
  - We have a bunch of VPN services allowing multiple concurrent connections.
  - Linux box (For example a clean base install of Debian Buster)
  - Base knowledge about Linux networking or almost nothing
- Objective
  - To build a load balancer that route each connection through a random outbound interface.
  - Meanwhile, explain the concepts involved about TCP/IP, Linux networking and Nftables.

Index

- VPN – Use case
- Linux Networking
  - Iproute2
  - Nftables
- TCP/IP Multiplexing rules
- Loadbalancing

VPN – Use Case

...

LAN

TLS Channels wrapping up tun{0,1,…,N}

tun0

tun1

tunN

VPN

VPN

VPN

VPN Servers

NETFLIX

Google

Setting the VPNs: General config

- Using TCP rather than UDP
  - In fact DTLS will do the work of TCP.
  - `proto tcp`
- TCP parameters for minimal latency
  - `socket_flags TCP_NODELAY`
    - Disables Nagle's Algorithm
    - If (Wsize & pendingdata >= MSS) → send.
  - Disable buffering:
    - `sndbuf 0`
    - `rcvbuf 0`
- TLS
  - `cipher AES-256-CBC`
- Disable routing table updates:
  - `route-noexec`

Setting them up

```
root@installer:/etc/openvpn/client# ls | grep nord
credentials.nordvpn
nordvpnar3.nordvpn.com.tcp443.conf
nordvpnau22.nordvpn.com.tcp443.conf
nordvpnch16.nordvpn.com.tcp443.conf
nordvpnde70.nordvpn.com.tcp443.conf
nordvpnru10.nordvpn.com.tcp443.conf
```

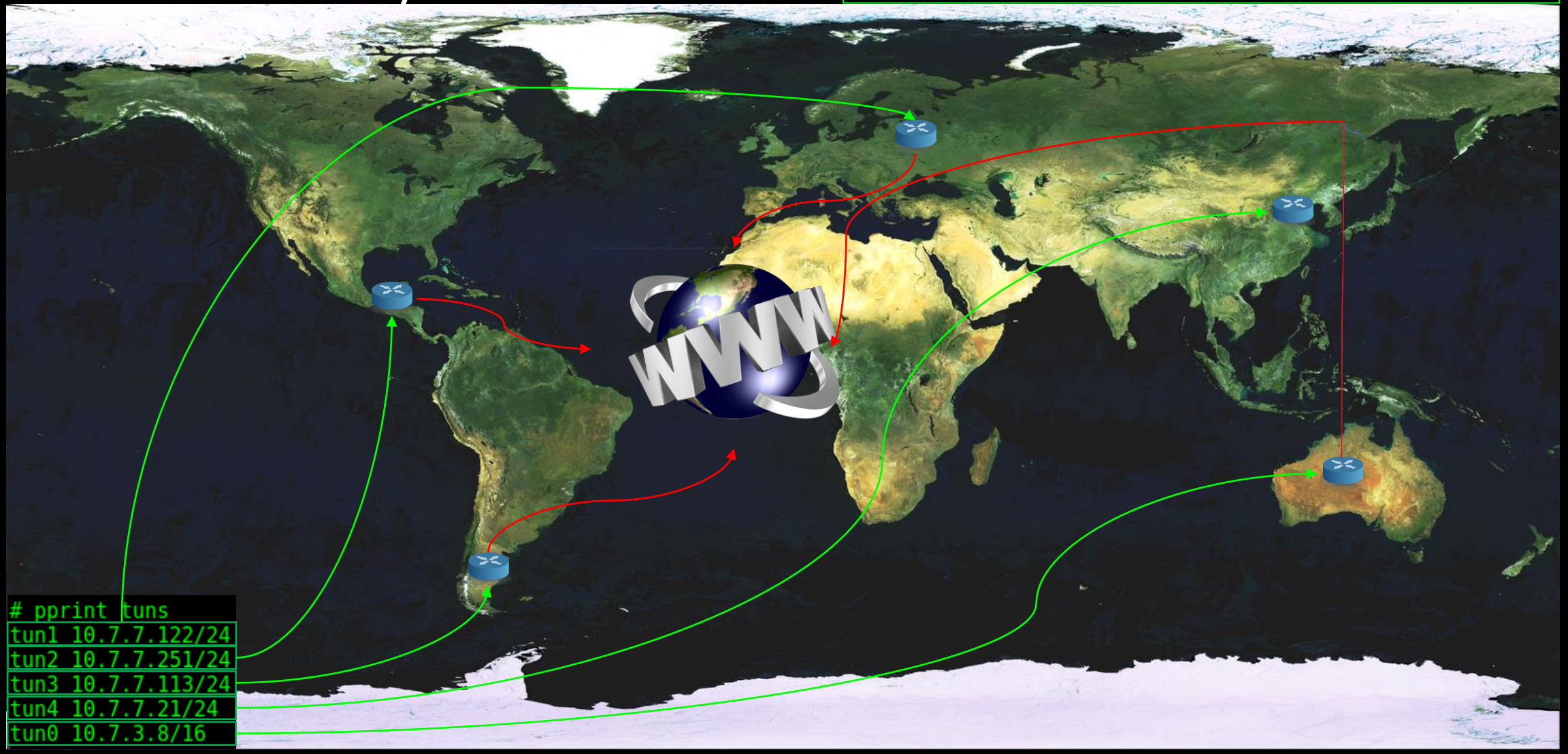
```
## Example conf
client
dev tun
proto tcp
remote <VPN_Server_addr> 443
...
```

```
function connect_vpns(){
    find $VPN_CLICFG_FOLDER -name "*.conf"  -printf "%f\n" | while read cfg_file
    do
        local service_name=`echo $cfg_file|perl -pe 's/.conf//g'`
        local com_servicename="openvpn-client@${service_name}.service"
        if ! ( systemctl start $com_servicename )
            then
                echo "Failed to establish tun with $com_servicename"
                continue;
            fi
        while ! (systemctl status $com_servicename| tail -2 |\
            grep "Initialization Sequence Completed" &>> /dev/null)
        do
            sleep 1
        done
        echo "$service_name started"
    done
}
```

```
root@installer:~/NFT_LB_home# connect_vpns
nordvpnde70.nordvpn.com.tcp443 started
nordvpnau22.nordvpn.com.tcp443 started
nordvpnch16.nordvpn.com.tcp443 started
nordvpnru10.nordvpn.com.tcp443 started
nordvpnar3.nordvpn.com.tcp443 started
```

In summary:

```
function pprint_tuns(){
    ip -o -4 addr |\
    grep -oP "(tun+[0-9]+)\s+inet ([0-9\.]+/[0-9]+)" |\
    perl -pe "s/\s+inet (([0-9]+\.){3})([0-9\/]+)/ \1\3/"
}
```



Testing the VPNs

```
function pprint_tracert(){
    local host=$1
    local interfaces=${@:2}
    for i in $interfaces
    do
        echo "Tracing through $i"
        traceroute -i $i $host |grep -v "traceroute to"| perl -pe 's/.*\((.*)\).*(\n)/\1 -> /g';
        echo ""
    done
}
```

```
root@installer:~# pprint_tracert 8.8.8.8 tun0 tun1 tun2
Tracing through tun0
10.7.0.1 -> 185.130.205.174 -> 80.81.192.108 -> 108.170.251.129 -> 108.170.227.57 -> 8.8.8.8 ->
Tracing through tun1
10.7.7.1 -> 169.254.175.50 -> 168.1.18.132 -> 50.97.19.148 -> 45.127.172.73 -> 108.170.247.65 -> 209.85.244.15 -> 8.8.8.8 ->
Tracing through tun2
10.7.7.1 -> 84.39.112.121 -> 176.10.83.58 -> 91.206.52.74 -> 209.85.243.125 -> 8.8.8.8 ->
```

```
root@installer:~# tshark -i tun0 -i tun1 -i tun2  -c 100 2>/dev/null |grep UDP | awk -F" " '{print $3,$4,$5,$6}'  | sort | uniq
10.7.3.8 → 8.8.8.8 UDP
10.7.7.122 → 8.8.8.8 UDP
10.7.7.251 → 8.8.8.8 UDP
```

Linux Networking – Firewall

- Iproute2. `aptitude show iproute2:`
  - The iproute2 suite is a collection of utilities for networking and traffic control. These tools communicate with the Linux kernel via the (rt)netlink interface, providing **advanced features** not available through the legacy net-tools commands 'ifconfig' and 'route'.
  - This allow us to change the routing decisions based on packets data or metadata.
- Nftables. `aptitude show nftables:`
  - These are the user-space administration tools for the Linux kernel's netfilter and nftables. netfilter and nftables provide a framework for stateful and stateless packet filtering, network and port address translation, and other IP packet manipulation. The framework is the successor to iptables.
  - Netfilter and nftables are used in applications such as Internet connection sharing, firewalls, IP accounting, transparent proxying, **advanced routing and traffic control**.
  - This allow us to tamper with packets data and metadata.

NAME

ip-route - routing table management

THE BIBLE!

Route tables: Linux-2.x can pack routes into several routing tables identified by a number in the range from 1 to 2^31 or by name from the file **/etc/iproute2/rt_tables** By default all normal routes are inserted into the **main** table (ID 254) and the kernel only uses this table when calculating routes. Values (0, 253, 254, and 255) are reserved for built-in use.

```
$ cat /etc/iproute2/rt_tables
#
# reserved values
#
255     local
254     main
253     default
0       unspec
```

DEFAULTS!

Actually, one other table always exists, which is invisible but even more important. It is the **local** table (ID 255). This table consists of routes for local and broadcast addresses. The kernel maintains this table automatically and the administrator usually need not modify it or even look at it.

The multiple routing tables enter the game when policy routing is used.

```
# ip route show table local | grep tun0
broadcast 10.7.0.0 dev tun0 proto kernel scope link src 10.7.3.8
local 10.7.3.8 dev tun0 proto kernel scope host src 10.7.3.8
broadcast 10.7.255.255 dev tun0 proto kernel scope link src 10.7.3.8
```

```
# ip route show table main
default via 192.168.2.1 dev eth0 onlink
10.7.0.0/16 dev tun0 proto kernel scope link src 10.7.3.8
10.7.7.0/24 dev tun1 proto kernel scope link src 10.7.7.122
10.7.7.0/24 dev tun2 proto kernel scope link src 10.7.7.251
10.7.7.0/24 dev tun3 proto kernel scope link src 10.7.7.113
10.7.7.0/24 dev tun4 proto kernel scope link src 10.7.7.21
192.168.2.0/24 dev eth0 proto kernel scope link src 192.168.2.24
```

DEFAULTS!

NAME

ip-rule - routing policy database management

…

Ip rule manipulates rules in the routing policy database control the route selection algorithm.

…

In some circumstances we want to route packets differently depending not only on destination addresses, but also on other packet fields: source address, IP protocol, transport protocol ports or even packet payload. **This task is called 'policy routing'**.

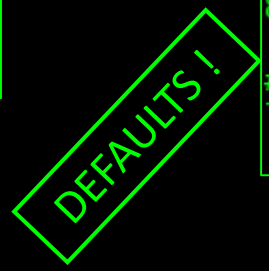
To solve this task, the conventional destination based routing table, ordered according to the longest match rule, is replaced with a **'routing policy database'** (or **RPDB**), which selects routes by executing some set of rules. Each policy routing rule consists of a selector and an action predicate.

The RPDB is scanned in order of decreasing priority.

The **selector** of each rule is applied to {…**fwmark**} and, if the **selector matches** the packet, the **action is performed**. …

Semantically, the natural action is to select the nexthop and the output device.

```
# ip rule show
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```




```
# ip route get 8.8.8.8
8.8.8.8 via 192.168.2.1 dev eth0 src 192.168.2.24
    cache
# ip route get 127.0.0.1
local 127.0.0.1 dev lo table local src 127.0.0.1
    cache <local>
```

Iproute2 – Simple PoC:

```
root@installer:~# ip route show table tun0
default dev tun0 scope link
root@installer:~# ip route show table main
default via 192.168.2.1 dev eth0 onlink
10.7.0.0/16 dev tun0 proto kernel scope link src 10.7.3.8
10.7.7.0/24 dev tun1 proto kernel scope link src 10.7.7.122
10.7.7.0/24 dev tun2 proto kernel scope link src 10.7.7.251
10.7.7.0/24 dev tun3 proto kernel scope link src 10.7.7.113
10.7.7.0/24 dev tun4 proto kernel scope link src 10.7.7.21
192.168.2.0/24 dev eth0 proto kernel scope link src 192.168.2.24
```

```
function set_rpdb_poc(){
    cat <<EOF > /etc/iproute2/rt_tables
# reserved values
#
255 local
254 main
253 default
EOF
    echo 21 tun0 >> /etc/iproute2/rt_tables
    ip rule add prio 1 fwmark 21 table tun0
    ip route replace default dev tun0 table tun0
}
```

```
root@installer:~# ip rule show
0:      from all lookup local
1:      from all fwmark 0x15 lookup tun0
32766:  from all lookup main
32767:  from all lookup default
```

Standard lookup on main table

```
# ip route get 8.8.8.8
8.8.8.8 via 192.168.2.1 dev eth0 src 192.168.2.24
```

```
# ping 8.8.8.8 -c 1
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=55 time=2.77 ms
```

Force lookup on new table tun0

```
# ip route get 8.8.8.8 mark 21
8.8.8.8 dev tun0 table 21 src 10.7.3.8 mark 0x15
```

```
# ping 8.8.8.8 -c 1 -m 21
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=61 time=39.1 ms
```

```
# tshark -i eth0 -i tun0 -f icmp -a duration:30 2>/dev/null |grep ICMP | awk -F" " '{print $3,$4,$5,$6}'
10.7.3.8 → 8.8.8.8 ICMP
8.8.8.8 → 10.7.3.8 ICMP
192.168.2.24 → 8.8.8.8 ICMP
8.8.8.8 → 192.168.2.24 ICMP
```

nft(8)

NAME

nft - Administration tool for packet filtering and classification

- Allows the user to set and match packet **meta information**
- From Nftables wiki →
- …

IPv4/IPv6/Inet address family hooks

Hook	Description
prerouting	All packets entering the system are processed by the prerouting hook. It is invoked before the routing process and is used for early filtering or changing packet attributes that affect routing.
input	Packets delivered to the local system are processed by the input hook.
forward	Packets forwarded to a different host are processed by the forward hook.
output	Packets sent by local processes are processed by the output hook.
postrouting	All packets leaving the system are processed by the postrouting hook.

SEE ALSO

iptables(8), ip6tables(8), arptables(8), ebtables(8), ip(8), tc(8)

There is an official wiki at: http://wiki.nftables.org

Nftables Chains & Hooks

Chains

type refers to the kind of chain to be created. Possible types are:

- *filter*: Supported by *arp*, *bridge*, *ip*, *ip6* and *inet* table families.
- *route*: Mark packets (like mangle for the *output* hook, for other hooks use the type *filter* instead), supported by *ip* and *ip6*.
- *nat*: In order to perform Network Address Translation, supported by *ip* and *ip6*.

hook refers to an specific stage of the packet while it's being processed through the kernel. More info in Netfilter hooks.

- The hooks for *ip*, *ip6* and *inet* families are: *prerouting*, *input*, *forward*, *output*, *postrouting*.
- The hooks for *arp* family are: *input*, *output*.
- The *bridge* family handles ethernet packets traversing bridge devices.
- The hook for *netdev* is: *ingress*.

Nftables – Priorities

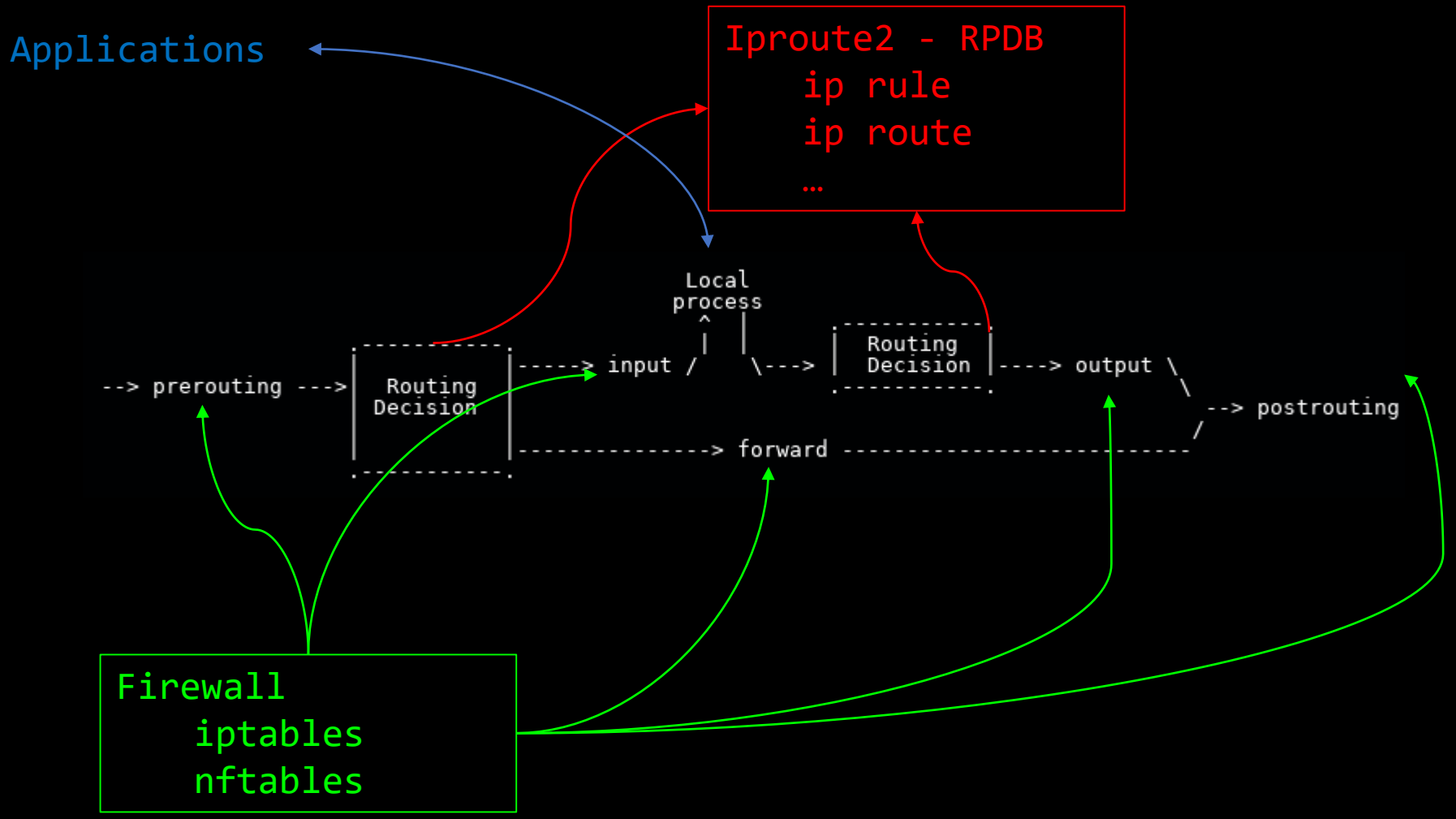
Base chain priority

The priority can be used to order the chains or to put them before or after some Netfilter internal operations. For example, a chain on the *prerouting* hook with the priority *-300* will be placed before connection tracking operations.

For reference, here's the list of different priority used in iptables:

- NF_IP_PRI_CONNTRACK_DEFRAG (-400): priority of defragmentation
- NF_IP_PRI_RAW (-300): traditional priority of the raw table placed before connection tracking operation
- NF_IP_PRI_SELINUX_FIRST (-225): SELinux operations
- NF_IP_PRI_CONNTRACK (-200): Connection tracking operations
- NF_IP_PRI_MANGLE (-150): mangle operation
- NF_IP_PRI_NAT_DST (-100): destination NAT
- NF_IP_PRI_FILTER (0): filtering operation, the filter table
- NF_IP_PRI_SECURITY (50): Place of security table where secmark can be set for example
- NF_IP_PRI_NAT_SRC (100): source NAT
- NF_IP_PRI_SELINUX_LAST (225): SELinux at packet exit
- NF_IP_PRI_CONNTRACK_HELPER (300): connection tracking at exit

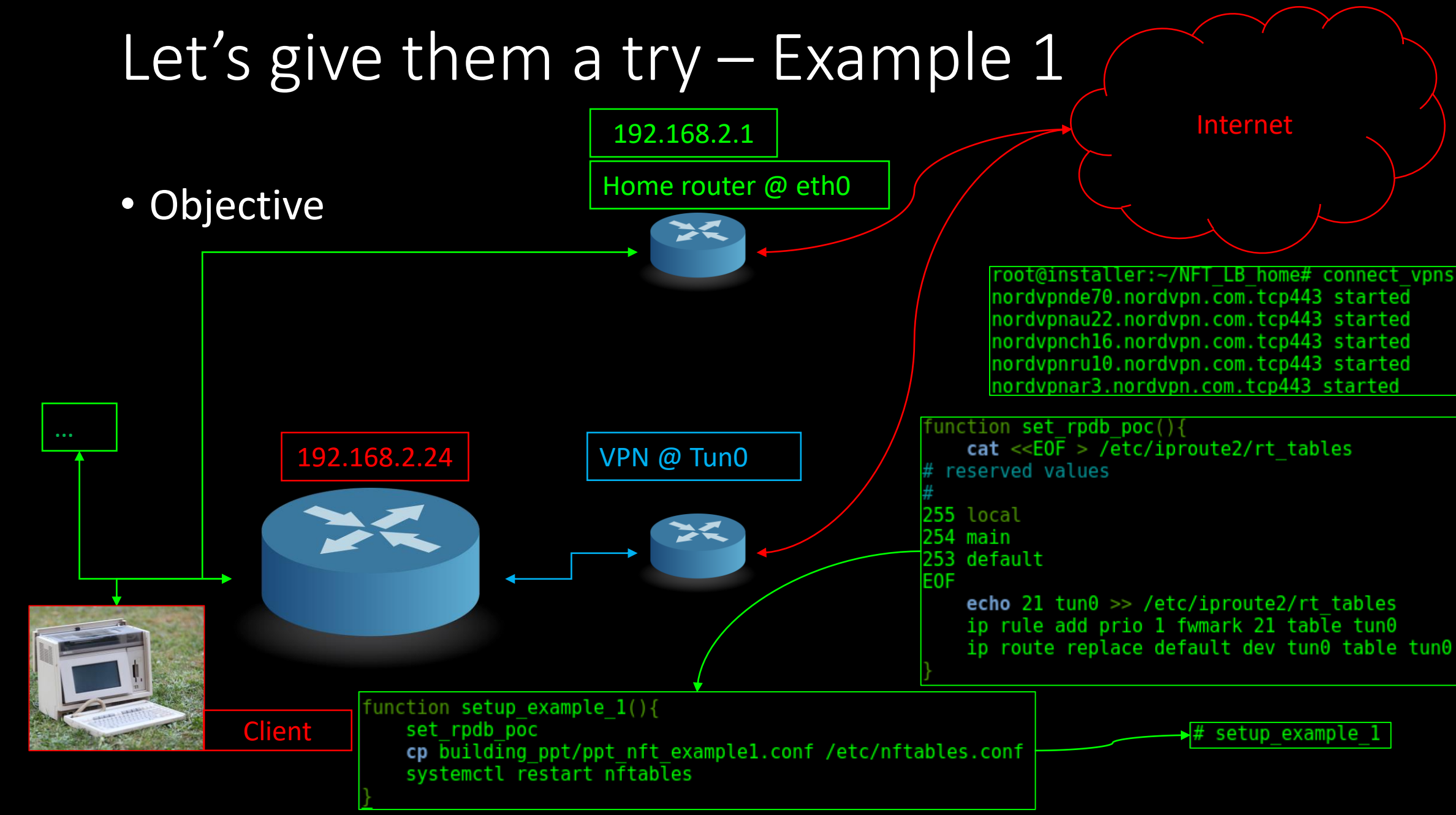
Linux Networking – Firewall (Map)



Source: Nftables Wiki

Let’s give them a try – Example 1

- Objective



Let’s give them a try – Example 1

/etc/nftables.conf

```
#!/usr/sbin/nft -f
flush ruleset;

table ip mangle {
    chain prerouting {
        type filter hook prerouting priority -200;
        ip protocol icmp nftrace set 1;
        ip daddr == 192.168.2.0/24 return;
        iif tun0 return;
        mark set 21;
    }
}

table ip nat {
    chain prerouting {
        type nat hook prerouting priority -200;
    }
    chain postrouting {
        type nat hook postrouting priority 100;
        ip protocol icmp nftrace set 1;
        mark != 0 masquerade;
    }
}
```

```
root@installer:~# ip rule show
0:      from all lookup local
1:      from all fwmark 0x15 lookup tun0
32766:  from all lookup main
32767:  from all lookup default
```

```
root@installer:~# ip route show table tun0
default dev tun0 scope link
root@installer:~# ip route show table main
default via 192.168.2.1 dev eth0 onlink
```

```
# ip route replace default via 192.168.2.24
```

Router – Normal Case

```
# curl https://ipinfo.io
{
  "ip": "79.153.94.226",
  "city": "Madrid",
  "region": "Madrid",
  "country": "ES",
  "loc": "40.4165,-3.7026",
  "org": "AS3352 TELEFONICA
  "postal": "28011"
}#
```

Client – Via VPN

```
# curl https://ipinfo.io
{
  "ip": "185.130.205.167",
  "city": "Frankfurt am Main",
  "region": "Hesse",
  "country": "DE",
  "loc": "50.1167,8.6833",
  "org": "AS61317 Digital Energy
  "postal": "09228"
}#
```

TCP/IP Multiplexing Rules

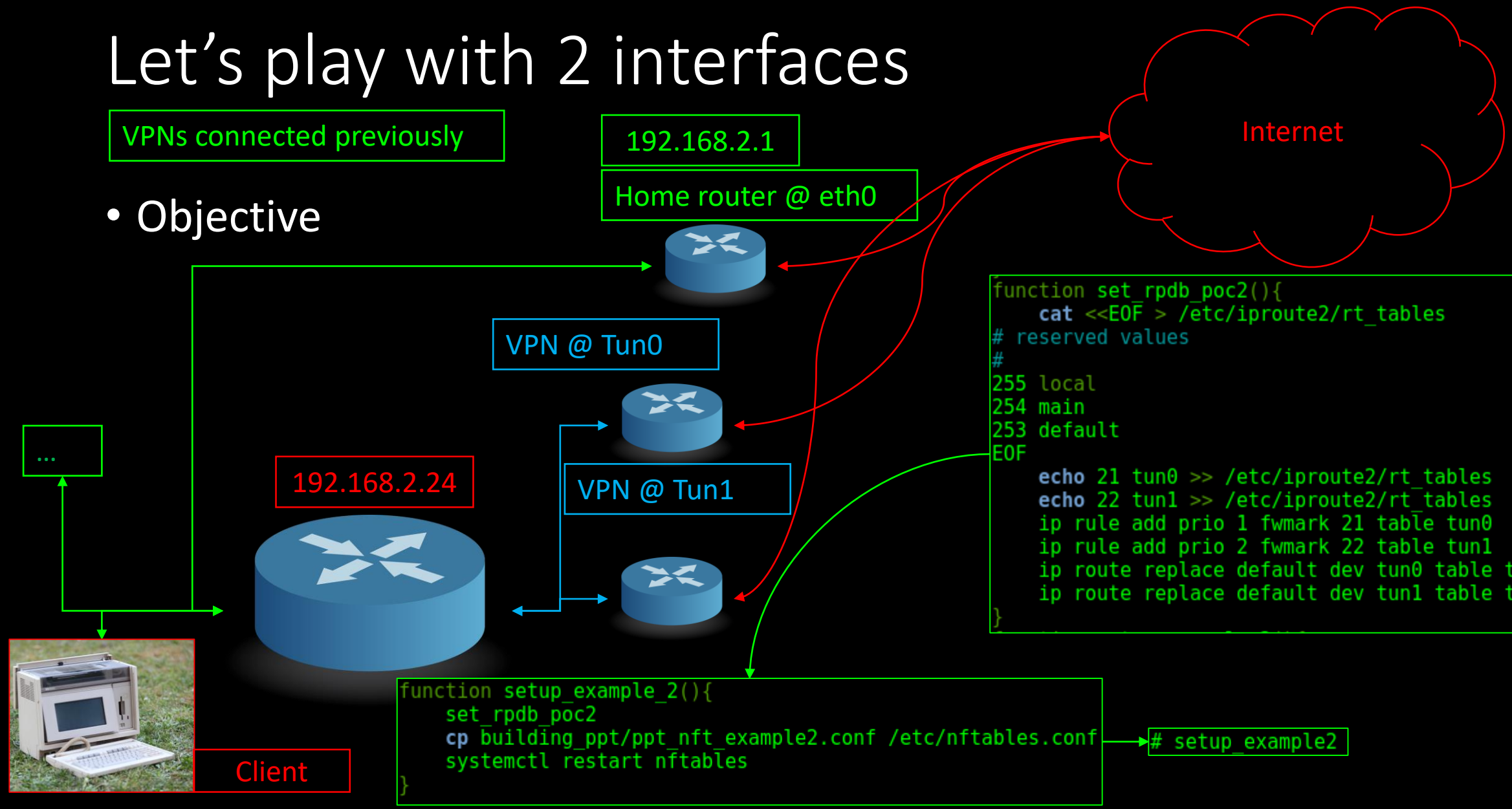

```
CONNTRACK(8)

NAME
       conntrack - command line interface for netfilter connection tracking
```

- ICMP
  - Packets that belong to the same ICMP flow should leave the router through the same outbound interface.
    - The flows are distinguished by the Id field of the packet.
- UDP
  - Packets that belong to the same UDP flow should leave the router through the same outbound interface.
    - The flows are uniquely identified by the following quadruple (Saddr,Sport,Daddr,Dport)
- TCP
  - Packets that belong to the same TCP connection should leave the router through the same outbound interface.
    - The flows are uniquely identified by the following quadruple (Saddr,Sport,Daddr,Dport)

Let’s play with 2 interfaces

VPNs connected previously

- Objective



Let’s play with 2 interfaces

/etc/nftables.conf

```
#!/usr/sbin/nft -f
flush ruleset;

table ip mangle {
    chain prerouting {
        type filter hook prerouting priority -150;
        ip protocol icmp nftrace set 1;
        iifname != eth0 return;
        mark set jhash ct mark mod 2 offset 21;
    }
}

table ip nat {
    chain prerouting {
        type nat hook prerouting priority -200;
        ip protocol icmp nftrace set 1;
        ct mark set jhash ip id mod 4294967295;
    }
    chain postrouting {
        type nat hook postrouting priority 100;
        ip protocol icmp nftrace set 1;
        mark != 0 masquerade;
    }
}
```

Avoid marking traffic coming back

Set mark on the packet. So it gets dispatched correctly

Set mark on the flow tracked by Conntrack

Check the outbound address

```
function check(){
    for i in {1..10}
    do
    curl -s https://ipinfo.io\
        | grep '"ip"' | awk -F'"' '{print $4}'
    sleep 0.1
    done
}
```

```
root@installer:~# check | sort | uniq -c
      4 103.86.96.5
      6 185.130.205.167
```

Let’s iterate the process with up to N tunnels

/etc/nftables.conf

```
#!/usr/sbin/nft -f
flush ruleset;

table ip mangle {
    chain prerouting {
        type filter hook prerouting priority -150;
        ip protocol icmp nftrace set 1;
        iifname != eth0 return;
        mark set jhash ct mark mod 55 offset 21;
    }
}

table ip nat {
    chain prerouting {
        type nat hook prerouting priority -200;
        ip protocol icmp nftrace set 1;
        #ct mark set jhash ip id mod 4294967295;
        ct mark set numgen random mod 4294967295;
    }
    chain postrouting {
        type nat hook postrouting priority 100;
        ip protocol icmp nftrace set 1;
        mark != 0 masquerade;
    }
}
```

Check the outbound address

```
function check_count(){
    check | sort | uniq -c | sort -n | perl -pe 's/^\s*//g'
}
```

```
root@nas:~# check_count
1 107.181.78.137
1 139.59.12.192
1 185.183.107.82
1 194.28.174.161
1 66.133.78.140
1 94.46.175.45
2 185.94.193.60
2 188.172.255.137
```

```
root@nas:~# check_count
1 103.25.59.82
1 107.181.78.137
1 185.104.187.114
1 185.156.174.12
1 198.144.149.169
1 207.189.27.8
1 66.171.37.103
3 103.37.95.98
```

```
root@nas:~# check_count
1 185.104.184.124
1 185.104.187.114
1 185.125.32.118
1 185.216.32.43
1 185.94.193.60
1 188.172.255.137
1 195.181.166.134
1 200.85.152.109
1 82.221.139.38
1 89.238.178.45
```

```
root@nas:~# check_count
1 103.37.95.98
1 195.181.166.134
1 198.144.149.169
1 45.64.186.188
1 46.166.172.51
1 82.221.139.38
1 89.238.131.136
1 89.46.103.148
1 94.242.62.238
1 95.213.136.45
```

Conclusions

- The routing subsystem uses the RPDB make the routing decision.
  - Remember that it’s managed by ip route/rule/…
- Nftables is the subsystem in charge of traffic classification.
  - It’s very flexible and friendlier than Iptables.
- The integration of both of them can be done in several manners.
  - While doing this work I’ve struggled many times with intermediate solutions that work but were more complex.

Improvements

- Setting the VPNs: Integration
  - Hooking after tunnel status changes,
    - This will improve the stability.
    - `up ontunup.sh → Update RPDB → Increase mod divisor`
    - `down ontundown.sh → Update RPDB → Decrease mod divisor`
    - `Script-security 2`
      - `2 -- Allow calling of built-in executables and user-defined scripts.`

Thanks for coming

Thanks to all developers of the software I've used here.