Thermalcircle.de

climbing the thermals

linux, kernel, routing, ipsec, netfilter, nftables, flowtables

Flowtables - Part 2: IPsec gateway in tunnel-mode

In this article series I like to take a look at *flowtables*, which is a network fastpath mechanism in the Linux kernel, based on Netfilter/Nftables, that allows accelerated handling of forwarded TCP and UDP connections. When using an acceleration feature like this, it is important to understand how it works. If you don't, then you'll have a hard time once you are going beyond just plain forwarding and start combining that acceleration with other networking features like e.g. Firewalling, NAT, advanced routing, QoS or IPsec. In this second article, I'll show how the packet flow looks like when you use a flowtable on a VPN gateway based on IPsec in tunnel-mode.

Articles of the series

- Flowtables - Part 1: A Netfilter/Nftables fastpath
- Flowtables - Part 2: IPsec gateway in tunnel-mode

Overview

The good news is that you can actually use *flowtables* in combination with IPsec. However, the gain of throughput you can achieve by doing that is limited. *Flowtables* can only accelerate forwarded traffic. On a VPN gateway like the one I'll describe here, this does not accelerate the encryption or decryption path. Received encrypted packets, which are about to be decrypted, travel on the local input path. Forwarded packets, before being sent out on the network, are encrypted and then inserted into the local output path. All *flowtables* can achieve here is to accelerate packets on the forward path, which either just have been decrypted or are about to be encrypted. However, using *flowtables* in this scenario still will increase your throughput by some degree, e.g. in case you got complex Nftables rulesets, which decrease your throughput on the slowpath and which thereby can be bypassed by the *flowtables* fastpath. Of course you need to take care that this bypassing still fits with your packet filtering intentions. Even with *flowtables*, the initial packets of each new connection will still travel on the slowpath. That might possibly be sufficient for your packet filtering use case. This article heavily builds upon knowledge which I covered in several of my other articles. I highly recommend that you take a look at those:

- Flowtables - Part 1: A Netfilter/Nftables Fastpath
- Nftables - Netfilter and VPN/IPsec packet flow
- Routing Decisions in the Linux Kernel - Part 1: Lookup and packet flow
- Connection tracking (conntrack) - Part 3: State and Examples

The Setup

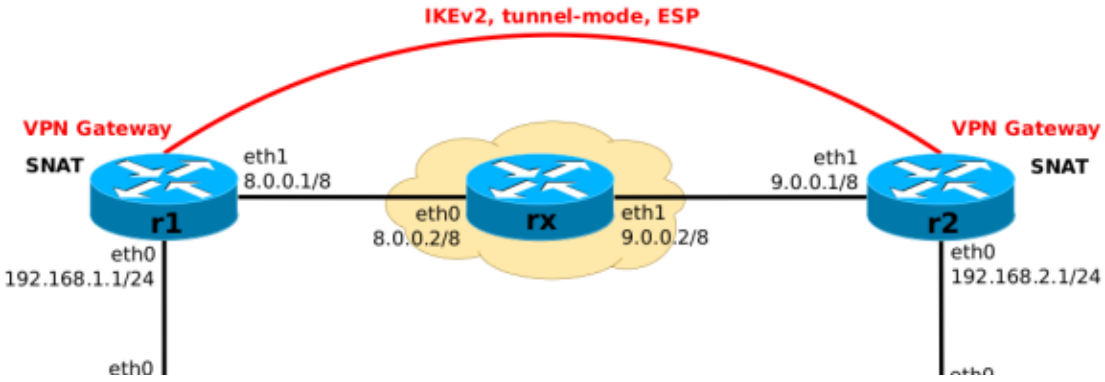

Figure 1: IPsec site-to-site example setup with VPN gateways `r1` and `r2`

I re-use the very-same site-to-site VPN gateway topology described in the mentioned VPN article as basis for explanation of the packet flow here; see Figure 1. This way you can compare in detail, how the packet flow differs when using *flowtables* compared to when not using that feature. The very same IPsec, routing and Nftables configuration also applies here. All what needs to be done to activate *flowtables* on gateway `r1` within this setup, is to create a *flowtable* and and an offload action rule in a chain in the Netfilter *forward* hook. The ruleset in Figure 2 does that. Simply add it to the existing Nftables ruleset[1].

```
table ip ft_table {
        flowtable f {
                hook ingress priority 0; devices = { eth0, eth1 };
        }
        chain y {
                type filter hook forward priority 100; policy accept;
                ip protocol { tcp, udp } flow add @f
        }
}
```

Figure 2: *Flowtable* setup on `r1`, accelerating forwarded TCP/UDP connections between `eth0` and `eth1`.

To explore the packet flow through `r1` in this setup, I created a simple TCP connection (actually an HTTP connection) between `h1` and `h2`, with `h1` being client and `h2` being server. I recorded a PCAP trace on `rx` and used a NULL cipher in the IPsec setup, so that all protocol headers (also the ones which else would be encrypted) could be fully observed by Wireshark. Additionally, I created an *ftrace* record on `r1` to follow the packet flow, function call by function call. *Ftrace* creates a huge amount of output, but I did cut out all irrelevant parts and left only the function graph traces of the packet traversal.

- pcap on `rx`: trace_r1_fwd_tcp_ipsec_tunnel_flowtable.pcapng
- ftrace on `r1`: trace_r1_fwd_tcp_ipsec_tunnel_flowtable.txt

This TCP connection consists of 10 packets in total. Packets 1 till 4 travel on the slowpath. Packets 1, 2, 3 together are the TCP 3-way handshake and packet 4 is the first packet carrying payload data. Here it is not packet 2 which triggers the offload action in the *forward* chain, like it was the case in the previous flowtables article. It is actually packet 3 here which triggers the offload. That difference is caused by IPsec, and I'll explain it below. Packets 5, 6, 7, and 8 travel on the flowtables fastpath. The final two packets, 9 and 10, travel on the slowpath. This is because packet 9 carries the TCP `FIN` flag, which triggers *teardown* of the *flowtable* entry.

> But wait a minute… if packet 3 triggers the offload action, why does packet 4 still travel on the slowpath? All packets following packet 3 should travel on the fastpath, right? In general, that indeed is true. The reason this does not happen here is quite special and has not much to do with *flowtables* or with IPsec. This is why I describe it separately here. Packet 4 would indeed travel on the fastpath, if the circumstances where just slightly different. The situation in this example TCP connection here is, that packets 3 and 4 travel in the same flow direction (from *h1* to *h2*) and are sent out nearly simultaneously by *h1*. It is common that received packets are handled not one by one but list-based on OSI layer 3. That is done by function ip_list_rcv() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L613][2], and that is the case on *r1* for those packets. As a consequence, by the time the offload action is triggered by packet 3, both packets 3 and 4 are already travelling on the slowpath and both already have traversed the Netfilter *Ingress* hook. This is why packet 4 still travels on the slowpath.

Encapsulate+Encrypt path

Let's take a look at one of the packets of the described TCP connection, which is sent from `h1` to `h2` and traverses the kernel

network stack on `r1`, being accelerated by *flowtables* and encrypted+encapsulated by IPsec. Figure 3 illustrates this packet traversal. As you can see, the *flowtables* fastpath accelerates the traffic by bypassing steps 5…9, but the actual IPsec encapsulate+encrypt step and all following steps after that stay the same.

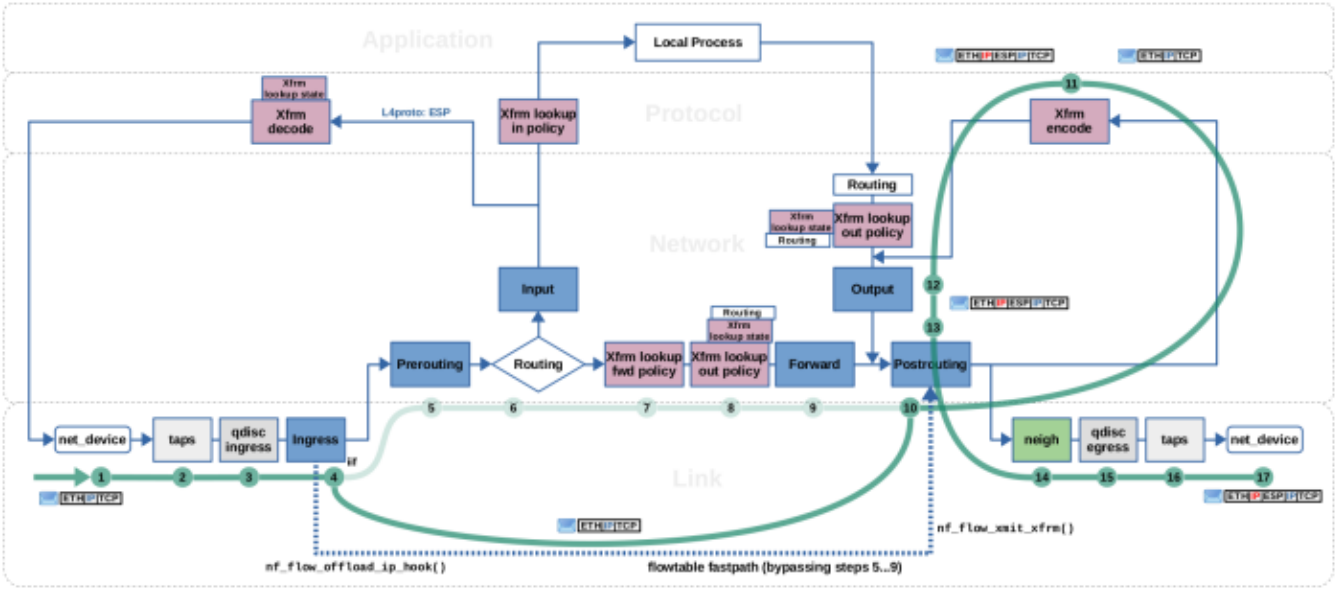

Figure 3: TCP packet sent from `h1` to `h2`, traversing `r1`, being accelerated by *flowtable* on *forward* path and then encrypted+encapsulated by IPsec (click to enlarge).

Step (1…3): The packet is received on `eth0`. It traverses *taps* (where e.g. Wireshark/tcpdump could listen) and then the *ingress* queueing discipline of the network packet scheduler (tc).

Step (4): The Packet traverses the Netfilter *Ingress* hook of `eth0`. The registered *flowtables* fastpath callback function `nf_flow_offload_ip_hook() [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_ip.c#L328]` is called. Its lookup into the flowtable results in a match. Some checks and minor things, like decrementing TTL, are done to the packet. In this case the *routing decision* cached by the flowtable for a packet of this flow and this flow direction is actually an “Xfrm bundle”. I described those in the VPN article. It is attached to the packet. This “bundle”, in addition to the *routing decision* itself, contains instructions for the Xfrm framework on how to encrypt and encapsulate this packet later. The *routing decision* within the “bundle” contains the usual function pointer `(*output)() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L36]` in `struct dst_entry [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25]`, which in this case points to function `xfrm4_output() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/xfrm4_output.c#L31]`, which represents the IPsec encrypt path. The *flowtables* function `nf_flow_xmit_xfrm() [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_ip.c#L232]` is called, which in turn calls `dst_output() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L446]`, which dereferences and calls function pointer `(*output)()`. This calls `xfrm4_output() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/xfrm4_output.c#L31]` and thereby the packet is being re-inserted into the slowpath. I described `dst_output()` and the semantics of this function pointer in detail in the article about routing decisions.

Step (5…9): Bypassed by *flowtables* fastpath.

Step (10): The packet traverses the Netfilter *Postrouting* hook. This is kind of special. “Normal” non-IPsec packets would get re-inserted into the slowpath by flowtables calling `neigh_xmit() [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/neighbour.c#L2991]` which inserts them into the *neighboring subsystem* and thus they would not traverse Netfilter *Postrouting*. However, `xfrm4_output() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/xfrm4_output.c#L31]` let's the packet traverse *Postrouting*, before doing the actual Xfrm transformation (step 11). This is an important subtle difference to remember. It means that in IPsec tunnel-mode, flowtable-accelerated packets on the encrypt+encapsulate path, while bypassing the Netfilter *Prerouting* and *Forward* hooks, still traverse the Netfilter *Postrouting* hook two times, once in still unencrypted form (step 10) and then once again in encrypted+encapsulated form (step 13). Packets of the opposite flow direction, which are decrypted+decapsulated, however, do not traverse *Postrouting* at all, if they are accelerated by *flowtables*.

Step (11): The Xfrm framework transforms the packet according to the instructions in the attached “bundle”. In this case this means encapsulating the IP packet into a new outer IP packet with source IP address `8.0.0.1` and destination IP addresses `9.0.0.1`

and then encapsulating the inner IP packet into ESP protocol and encrypting it and its payload. After that, the transformation instructions are removed from the "bundle", leaving only the routing decision for the new outer IP packet attached to the packet.

Step (12…17): The packet is re-inserted into the local output path. It traverses the Netfilter *Output* hook and then again the Netfilter *Postrouting* hook. It traverses the *neighboring subsystem* which in this case resolves the *nexthop* gateway ip address `8.0.0.2` from the attached *routing decision* into a MAC address (by doing ARP lookup, if address not yet in cache). Finally, it traverses the *egress* queueing discipline of the network packet scheduler (`tc`), *taps* (where e.g. Wireshark/tcpdump could listen) and then is sent out on `eth1`.

Decapsulate+Decrypt path

Let's take a look at one of the packets of the described TCP connection, which is sent from `h2` to `h1` and traverses the kernel network stack on `r1`, being decrypted+decapsulated by IPsec and then accelerated by *flowtables*. Figure 4 illustrates this packet traversal. As you can see, the *flowtables* fastpath accelerates the traffic by bypassing steps 13…18, but all other steps, which includes the actual IPsec decapsulate+decrypt step, stay the same.

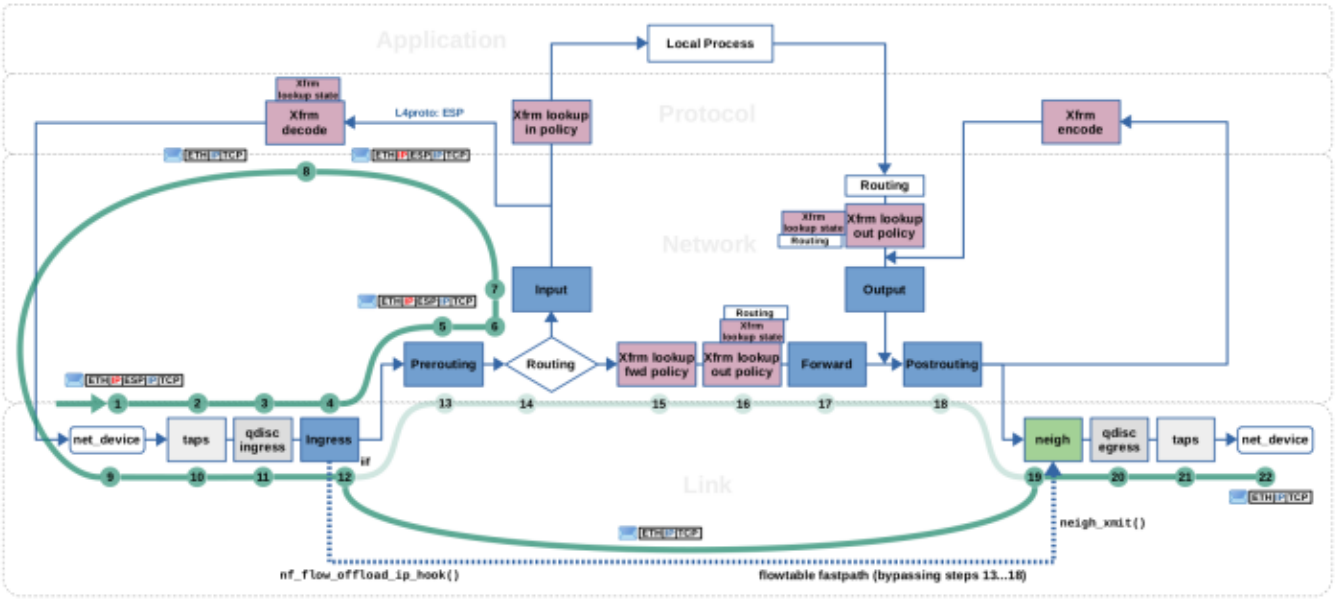

Figure 4: TCP packet sent from `h2` to `h1`, traversing `r1`, being decrypted+decapsulated by IPsec and then accelerated by *flowtable* on *forward* path (click to enlarge).

Step (1…7): The packet is received on `eth1` in its encrypted+encapsulated form. It traverses taps (where e.g. Wireshark/tcpdump could listen), the ingress queueing discipline of the network packet scheduler (`tc`), and the Netfilter *Ingress* hook of `eth1`. There is no flowtable match here, as *flowtables* can only see the outer IP packet at this point. The packet then traverses the Netfilter *Prerouting* hook. Now the routing lookup is performed. In this case, the routing subsystem determines that this packet's destination IP `8.0.0.1` matches the IP address of interface `eth1` of this host `r1`. Thus, this packet is destined for local reception and the lookup attaches an according *routing decision* to it. As a result, the packet then traverses the Netfilter *Input* hook.

Step (8…11): The Xfrm framework has a layer 4 receive handler waiting for incoming ESP packets at this point. It parses the SPI value from the ESP header of the packet and performs a lookup into the SAD for a matching IPsec SA (lookup based on SPI and destination IP address). A matching SA is found, which specifies the further steps to take here for this packet. It is decrypted and the ESP header is decapsulated. Now the internal IP packet becomes visible. The SA specifies tunnel-mode, so the outer IPv4 header is decapsulated. A lot of packet meta data is changed here, e.g. the attached routing decision (of the outer IP packet, which is now removed) is stripped away, the reference to connection tracking is removed, and a pointer to the SA, which has been used here to transform the packet, is attached (via skb extension `struct sec_path` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/xfrm.h#L1021]). As a result, other kernel components can later recognize that this packet has been decrypted by Xfrm. Finally, the packet is now re-inserted into the OSI layer 2 receive path of `eth1`. Now history repeats… the packet once again traverses *taps* and the *ingress* queueing discipline.

Step (12): The packet traverses the Netfilter *Ingress* hook of `eth1`. The registered *flowtables* fastpath callback function

`nf_flow_offload_ip_hook()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nf_flow_table_ip.c#L328] is called. Its lookup into the flowtable results in a match. In this case the *routing decision* cached by the *flowtable* for a packet of this flow and this flow direction is actually no "Xfrm bundle", but the normal combination of `struct rtable` and `struct dst_entry`, which I described in the routing article. It is attached to the packet. Some more checks and minor things, like decrementing TTL, are done to the packet and then it is "re-inserted" into the slowpath by calling function `neigh_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/neighbour.c#L2991]. This re-inserts it into the *neighboring subsystem* (see step 19). Thus, in this flow direction, *flowtables* does not handle the packet in any special IPsec-related way. Flowtables is not IPsec-"aware" in this flow direction and just handles the packet like any other "normal" offloaded packet. This also means, that Netfilter *Postrouting* is not traversed by offloaded packets in this flow direction.

Step (13…18): Bypassed by *flowtables* fastpath.

Step (19): The packet, coming from the *flowtables* fastpath, is re-inserted into the *neighboring subsystem* by function `neigh_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/neighbour.c#L2991] (as described in step 12). The *neighboring subsystem* does resolve the *nexthop* IP address, in this case the destination IP address of the packet, into a MAC address (by doing ARP lookup, if address not yet in cache).

Step (20…22): The packet traverses the *egress* queueing discipline of the network packet scheduler (`tc`), *taps* (where e.g. Wireshark/tcpdump could listen) and is then sent out on `eth0`.

Offload Action and IPsec

There is a small but tricky detail regarding the offload action, when you are using IPsec in tunnel-mode in combination with *flowtables*. The offload action is implemented as function `nft_flow_offload_eval()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nft_flow_offload.c#L271]. This function is executed when a packet traverses the `flow add @f` statement in your *forward* chain; see again Figure 2. The general principle of the offload action is that it is usually triggered by the first "confirmed" packet of a flow (flow = a connection tracked by conntrack) and its purpose is to create a new entry within the *flowtable* and this entry must always contain the two *routing decisions*, one for each flow direction of the flow which is to be offloaded. One of those two *routing decisions* is delivered by the packet which triggered this offload action (because each packet, after traversing the routing lookup, has a *routing decision* attached to it). To obtain the second *routing decision*, which is the one for the opposite flow direction, the *flowtables* code now needs to do a routing lookup. But here lies a very subtle problem… As you saw in the sections above, in case of IPsec tunnel-mode, one of the two *routing decisions*, which the *flowtables* entry needs to remember (to cache), is an "Xfrm bundle" (the one used for the packet flow direction which is the encrypt+encapsulate path) and the other one is just a regular combination of `struct rtable` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L49] and `struct dst_entry` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25] (the one used for the flow direction which is the decrypt+decapsulate path). It cannot be predicted which packet of which flow direction will actually be the first "confirmed" packet which triggers this offload action. If the packet which triggers this offload action is actually a packet which just now has been decrypted+decapsulated by Xfrm, then this packet carries a regular *routing decision* and not an "Xfrm bundle". The *flowtables* code would save this one and perform a routing lookup to obtain the *routing decision* for the opposite flow direction. That opposite one however needs to be an "Xfrm bundle" for everything to work correctly! And a normal routing lookup does not produce an "Xfrm bundle" because it does not do an Xfrm policy lookup. The opposite situation however would work fine: If the packet which triggers this offload action is actually a packet which is yet to be encapsulated+encrypted by Xfrm, then this packet carries an "Xfrm bundle" as a *routing decision*. The *flowtables* code would save the "bundle" in the new flowtable entry and perform a routing lookup to obtain the *routing decision* for the opposite flow direction. That opposite however is just a normal one and no "Xfrm bundle". So everything works well in this case. The currently implemented solution to this tricky little problem is the following: Function `nft_flow_offload_eval()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nft_flow_offload.c#L271] actually skips the offload action for network packets, which carry the `sec_path` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/xfrm.h#L1021] skb extension (described above), which means packets that just got decrypted+decapsulated by Xfrm. So, in this case the actual "offload" of a flow into the *flowtables* fastpath happens a little later, probably with the very next packet. This is actually done in function `nft_flow_offload_skip()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/netfilter/nft_flow_offload.c#L254], which calls `skb_sec_path(skb)` [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L4457] to check for the `sec_path` extension. This is why it is packet 3 in the example TCP connection above which triggers the offload and not packet 2. Confusing? Yes, indeed it is, but it works! 😉

Context

The described behavior and implementation has been observed on a Debian 11 (bullseye) system with using Debian *backports* on *amd64* architecture.

- kernel: `5.14.7`
- nftables: `0.9.8`

Feedback

Feedback to this article is very welcome! Please be aware that I did not develop or contribute to any of the software components described here. I'm merely some developer who took a look at the source code and did some practical experimenting. If you find something which I might have misunderstood or described incorrectly here, then I would be very grateful, if you bring this to my attention and of course I'll then fix my content asap accordingly.

published 2022-08-14, *last modified 2022-08-14*

1)
Of course, it is not strictly necessary to isolate the *flowtables* setup in a separate table and create a separate base chain for it in the *forward* hook, like I did that here. You could also integrate that in the existing table and *forward* chain. However, this way here is a clear separation and probably easier to read and maintain. Priority of the base chain is intentionally set to 100, so that it is traversed AFTER the existing *forward* chain, which has priority 0.
2)
In case you like to dig deeper into that list-base packet handling: I described it to some degree in my article series about routing decisions.