UNIVERSITY OF LIMOGES

Faculty of Science and Technology

Master 1

Cryptology and Information Security (CRYPTIS)

Informatic Course

Network Audit and Security - Semester 2

IPv6 $\Longleftrightarrow$ IPv4 translator for TCP protocol

DUONG Dang-Hung

Professor: Pierre-François BONNEFOI

April 17, 2023

Table of contents

Chapter 1

Project description

The project's objective is to enable transparent TCP communication between machines running only the TCP/IPv6 stack and those running IPv4 systems.

Specifically, the process follows these steps:

- Create a net namespace `h1`which is a machine in an IPv6 local network whose prefix is `2001:2:3:4501::/64`
- Create a switch `s1` which represents the local IPv6 network
- The real Linux machine plays the role of the router `r1`
- The host machine `h1` communicates with the local IPv6 network through its interface `eth0`, while the router `r1` communicates with the local IPv6 network through interface `bridge_ipv6` and with the Internet through its interface `wlo1`
- The internet protocol used inside the local network is IPv6, whereas the Linux machine, which now acts as a router, communicate with the Internet using IPv4. Henceforth:
  - Any packets goes from the local network outside must first be decapsulated from the IPv6 header and encapsulated with the appropriate IPv4 header to be sent to the Internet
  - Any packets comes from the Internet must first be decapsulated from the IPv4 header and encapsulated with the appropriate IPv6 header to be sent inside the local network

For this project, since it was specified that the program should work with only a given IPv6 address and a given IPv4 address, `google.com` and port 80 (HTTP) was chosen to be the external server to be communicated.

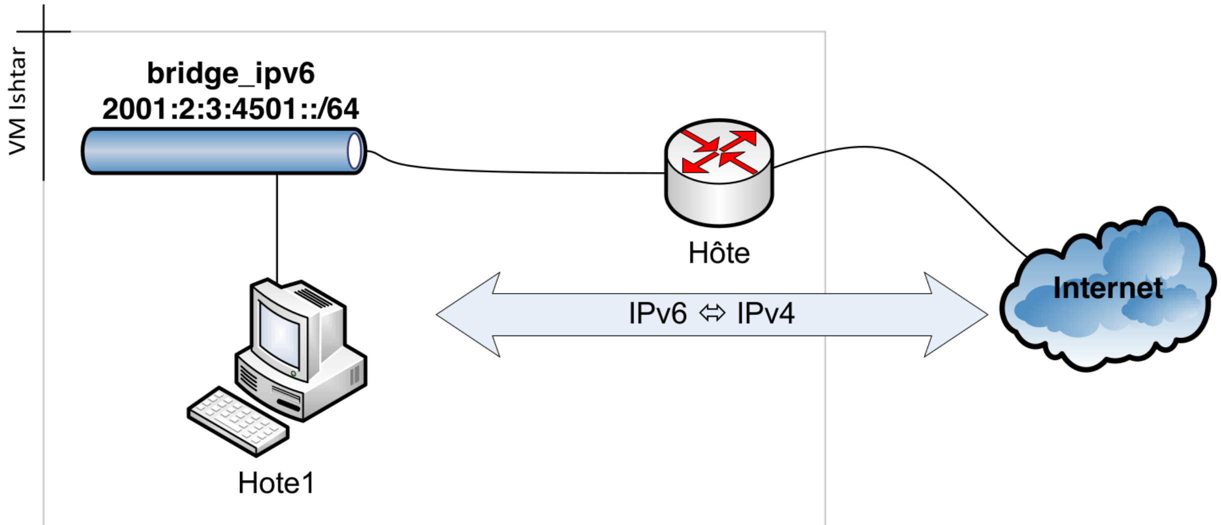


Figure 1.1: Network architecture specified in the document

Based on my understanding, I have redrawn the architecture of the project as below, and try to construct the network based on it:

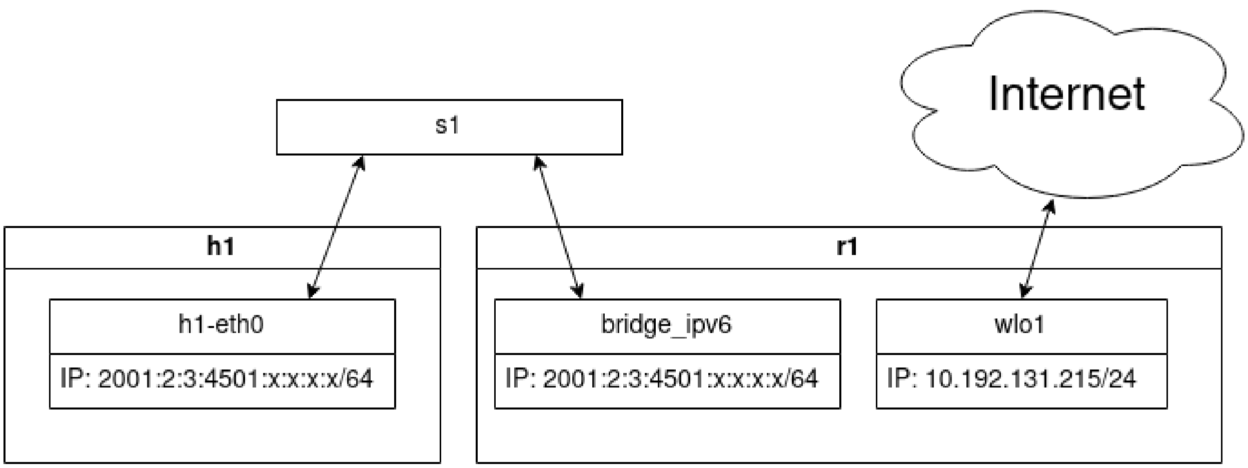


Figure 1.2: Network architecture built

where `h1` is the net namespace, `s1` is the switch connecting the local network, and `r1` is the router, which is in fact the real Linux machine.

Chapter 2

Programming Implementation

2.1 Build the architecture

The project starts with building the architecture. This includes creating the net namespace, the switch, link the interfaces with the switch, etc. `radvd` (which stands for Router ADVertisement Daemon) is also utilized to periodically multi-cast Router Advertisement (RA) message to all machines in the same network segment, so they can be assigned an IPv6 address with the prefix specified . All of these works are carried out in the file `build_architecture`. Run the command:

```
 $ sudo ./build_architecture
```

After a while the router would multicast RA messages, and each machine (in this case, only `h1`) will auto-config its IPv6 address. Check using the command `ip address`.
On the router:

```
$ ip address show bridge_ipv6
12: bridge_ipv6@s1-r1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    noqueue state UP group default qlen 1000
     link/ether 62:11:c5:71:e4:12 brd ff:ff:ff:ff:ff:ff
     inet6 2001:2:3:4501:fab6:a7df:837e:5c57/64 scope global temporary
    dynamic
        valid_lft 86400sec preferred_lft 14400sec
     inet6 2001:2:3:4501:6011:c5ff:fe71:e412/64 scope global dynamic
    mngtmpaddr
        valid_lft 86400sec preferred_lft 14400sec
     inet6 fe80::6011:c5ff:fe71:e412/64 scope link
        valid_lft forever preferred_lft forever
```

On `h1`:

```
$ sudo ip netns exec h1 ip address show eth0
14: eth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    state UP group default qlen 1000
     link/ether 12:02:b0:0c:87:c4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
     inet6 2001:2:3:4501:1002:b0ff:fe0c:87c4/64 scope global dynamic
    mngtmpaddr
        valid_lft 86392sec preferred_lft 14392sec
     inet6 fe80::1002:b0ff:fe0c:87c4/64 scope link
        valid_lft forever preferred_lft forever
```

2.2 Build the IPv4 $\Longleftrightarrow$ IPv6 translator

As described in Chapter 1, the program needs to be able to decapsulate a packet from IPv6 header and encapsulate it with IPv4 header if the packet is going outside from the local network, and vice versa. The 2 main Python libraries used here are `Scapy` and `NetfilterQueue`

A class `Translator` was created in Python:

```python
#!/usr/bin/python3

from scapy.all import *
from netfilterqueue import NetfilterQueue
import threading

class Translator:
    def __init__(self, server_ipv4, server_ipv6, client_mac, client_ipv6, internet_iface, local_ipv6_iface) -> None:
        self.__server_ipv4 = server_ipv4
        self.__server_ipv6 = server_ipv6
        self.__client_mac = client_mac
        self.__client_ipv6 = client_ipv6
        self.__internet_iface = internet_iface
        self.__local_ipv6_iface = local_ipv6_iface
```

2 functions are created for this class: `process_outgoing_packets()` which translate an IPv6 packet to an IPv4 packet then send it outside, and `process_incoming_packets()` do it the other way around. To be able to appropriately create a new header for a packet, we rely in the comparisons below:

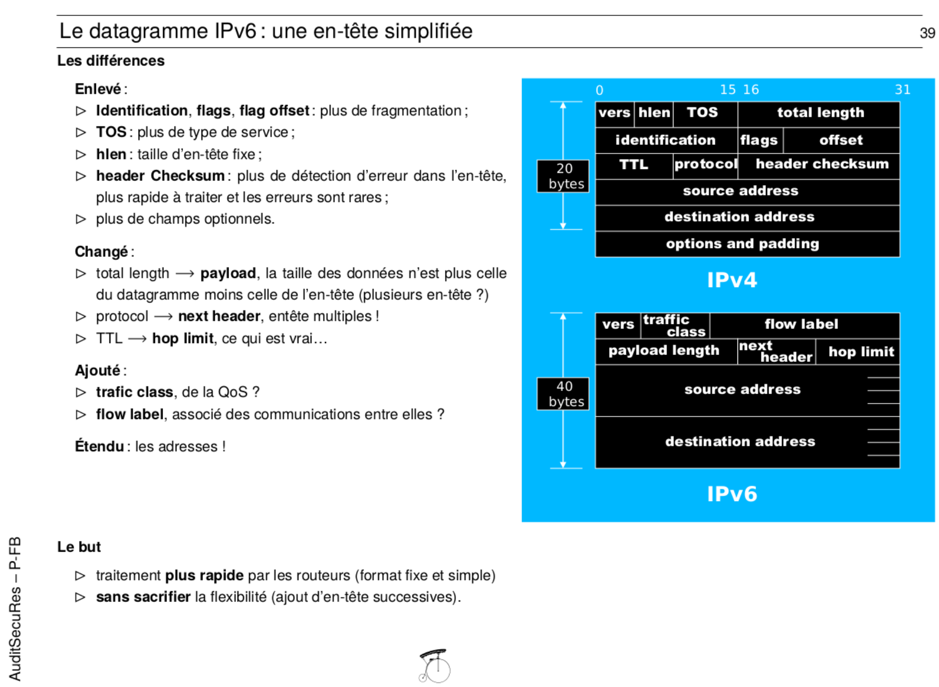


Figure 2.1: IPv4 vs IPv6 header

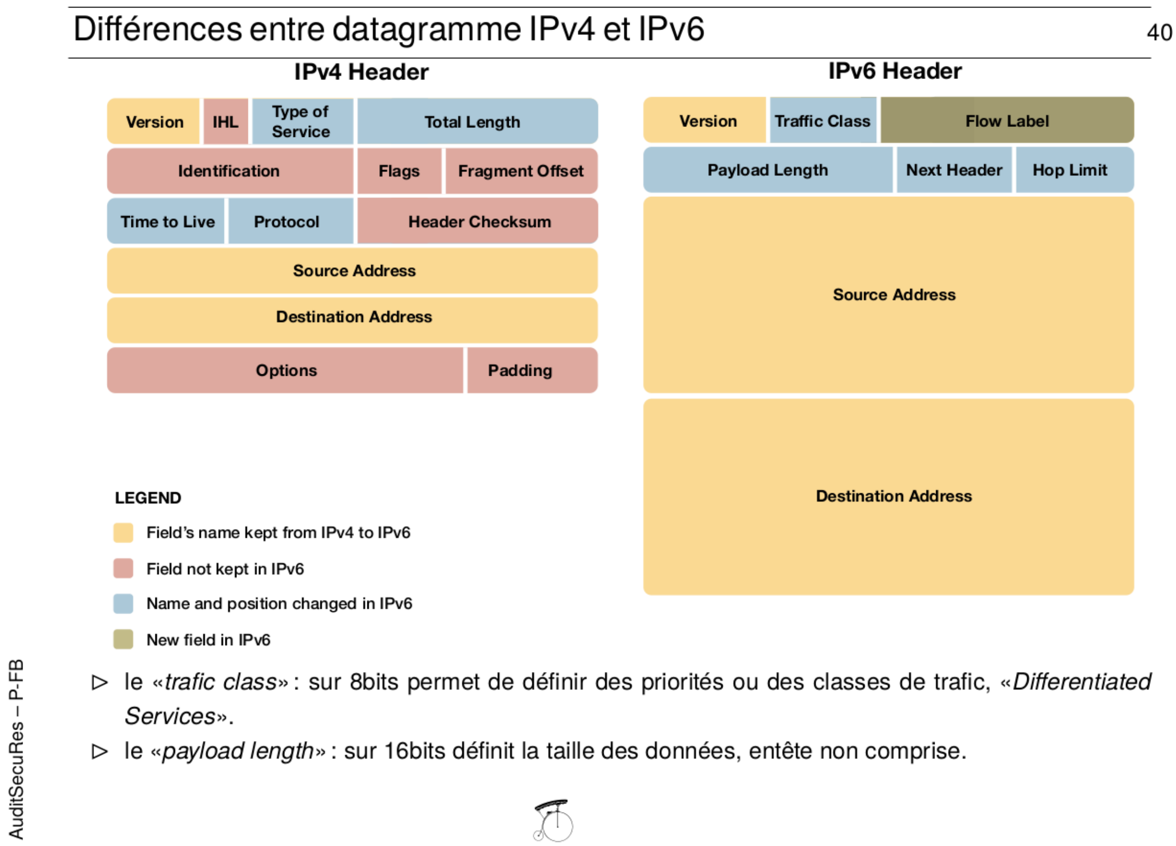


Figure 2.2: IPv4 vs IPv6 header (continue)

Studying these comparisons, to create an IPv4 header corresponding to the original IPv6 header, these fields have to be migrated:

- Traffic Class ⟶ Type of Service
- Payload Length ⟶ Total Length (is auto-configured by `Scapy`)
- Next Header ⟶ Protocol
- Hop Limit ⟶ Time to Live
- Source Address (IPv6) ⟶ Source Address (IPv4) (is auto-configured by `Scapy`)
- Destination Address (IPv6) ⟶ Destination Address (IPv4)

Other fields not mentioned are auto-configured by `Scapy` when constructing a new header, hence can be ignored.

After having constructed the new corresponding IPv4 header, we can use it to encapsulate the TCP packet, and the whole packet is encapsulated as an Ethernet frame, then, it is sent to the Internet through the `wlo1` interface.

The implementation of the process described above in Python is as below:

```
    def process_outgoing_packets(self, p):
        data = p.get_payload()

        ipv6_packet = IPv6(data)
        tc = ipv6_packet.tc
        hlim = ipv6_packet.hlim

        p_tcp = ipv6_packet[TCP]
        del p_tcp.chksum # If not do this, receive no SYN/ACK

        ipv4_header = IP(tos=tc, ttl=hlim, proto='tcp', dst=self.__server_ipv4)

        ipv4_packet = Ether() / ipv4_header / p_tcp

        sendp(ipv4_packet, iface=self.__internet_iface)
```

Applying the reverse process, the function `process_incoming_packets()` used to handle IPv4 packets from outside and translate them to IPv6 then forward it inside is implemented:

```
    def process_incoming_packets(self, p):
        data = p.get_payload()

        ipv4_packet = IP(data)

        p_tcp = ipv4_packet[TCP]
        tos = ipv4_packet.tos
        ttl = ipv4_packet.ttl
        proto = ipv4_packet.proto
        del p_tcp.chksum

        ipv6_header = IPv6(tc=tos, hlim=ttl, nh=proto, src=self.__server_ipv6, dst=self.__client_ipv6)

        ipv6_packet = Ether(dst=self.__client_mac) / ipv6_header / p_tcp

        sendp(ipv6_packet, iface=self.__local_ipv6_iface)
```

Finally, a function is constructed to run the 2 functions created above, in order to handle both outgoing and incoming packets. Thanks to `NetfilterQueue`, all outgoing packets will be put in queue number 1, which then is handled by `process_outgoing_packets()`, and all incoming packets will be put in queue number 2, which is handled by `process_incoming_packets()`. They are executed concurrently in 2 separate threads. The process of putting outgoing and incoming packets in queue 1 and queue 2 is carried out by NetFilter, which will be indicated later.

The `run()` function is implemented as below:

```
    def run(self):
        q1 = NetfilterQueue()
        q1.bind(1, self.process_outgoing_packets)

        q2 = NetfilterQueue()
        q2.bind(2, self.process_incoming_packets)

        t1 = threading.Thread(target=q1.run)
        t2 = threading.Thread(target=q2.run)
        t1.start()
        t2.start()
```

2.3 Build the main program

The only work left to do is to use the translator programmed in Section 2.2 and run it to handle the packets, which is somehow quite similar to a "test drive" program of the Translator.
Some constant arguments are set such as the external interface that connects to the Internet, the internal interface that connects to local IPv6 network, IPv6 address of the server (google.com, in this case), IPv4 address of the server, IPv6 address of the client, MAC address of the client:

```
#!/usr/bin/python3

import subprocess
import sys
from translator import Translator

INTERNET_IFACE = "wlo1"
LOCAL_IPV6_IFACE = "bridge_ipv6"
IPV6_PREFIX = "2001:2:3:4501:"

## Find the MAC and IPv6 address of the netns h1
p1 = subprocess.Popen("ip netns exec h1 ip address show eth0".split(" "), stdout=subprocess.PIPE)
p2 = subprocess.Popen(["grep", "-E", "link/ether|{}".format(IPV6_PREFIX)], stdin=p1.stdout, stdout=subprocess.PIPE)
CLIENT_MAC, CLIENT_IPV6  = [line.strip().split()[1] for line in p2.communicate()[0].decode().splitlines()]
p1.stdout.close()
CLIENT_IPV6 = CLIENT_IPV6[:-3]
```

The main program takes the domain name server as an argument, then its IPv4 address and IPv6 address are found by sending DNS request using the `dig` command:

```
if __name__ == "__main__":
    ## Find the IPv4 and IPv6 addresses of the server by sending DNS Request using dig command, mode A (to find IPv4) and AAAA (to find IPv6)
    SERVER_DOMAIN_NAME = sys.argv[1]
    SERVER_IPV4, SERVER_IPV6 = [subprocess.run(["dig", "+short", SERVER_DOMAIN_NAME, dig_type], capture_output=True).stdout.decode().strip("\n") for dig_type in ["A", "AAAA"]]
```

It is also essential to set the NetFilter rules using `iptables` and `ip6tables` commands. For the outgoing packets, since the router receives IPv6 packets, `ip6tables` therefore should be applied, on the other hands, for the incoming packets, the router receives IPv4 packets from the Internet, thus we use `iptables` for them.

All packets coming from interface `bridge_ipv6` that uses TCP protocol, having source IPv6 address with prefix `2001:2:3:4501::/64` should be put in queue 1. This correspond to the rule:

```
$ ip6tables -t mangle -A PREROUTING -i bridge_ipv6 -p tcp -s 2001:2:3:4501::/64 -j NFQUEUE --queue-num 1
```

which in Python is executed using `subprocess`:

```
    subprocess.run(["ip6tables", "-t", "mangle", "-A", "PREROUTING", "-i", LOCAL_IPV6_IFACE, "-p", "tcp", "-s", IPV6_PREFIX + ":/64", "-j", "NFQUEUE", "--queue-num", "1"])
```

All packets coming from interface `wlo1` that uses TCP protocol, having source IPv4 address equals to <`google.com IPv4 address`>, from source port 80 (HTTP) should be put in queue 2. This might not be the best way to handle incoming packets, but no better idea has been come up with. This correspond to the rule:

```
$ iptables -t mangle -A PREROUTING -i wlo1 -p tcp -s <google.com IPv4 address> --sport 80 -j NFQUEUE --queue-num 2
```

which in Python is executed using `subprocess`:

```
    subprocess.run(["iptables", "-t", "mangle", "-A", "PREROUTING", "-i", INTERNET_IFACE, "-p", "tcp", "-s", SERVER_IPV4, "--sport", "80", "-j", "NFQUEUE", "--queue-num", "2"])
```

Finally we run the Translator:

```
    if len(SERVER_IPV4) == 0 or SERVER_IPV4 is None:
        raise Exception("This domain has no IPv4 address")
    if len(SERVER_IPV6) == 0 or SERVER_IPV6 is None:
        raise Exception("This domain has no IPv6 address")
    t = Translator(SERVER_IPV4, SERVER_IPV6, CLIENT_MAC, CLIENT_IPV6, INTERNET_IFACE, LOCAL_IPV6_IFACE)
    t.run()
```

Chapter 3

Result

To test the program, `socat` is used to establish a TCP communication to the `google.com` server. The client netns `h1` will try to send a HTTP Request method GET to `google.com` port 80.

First, find the IP address of the domain name server:

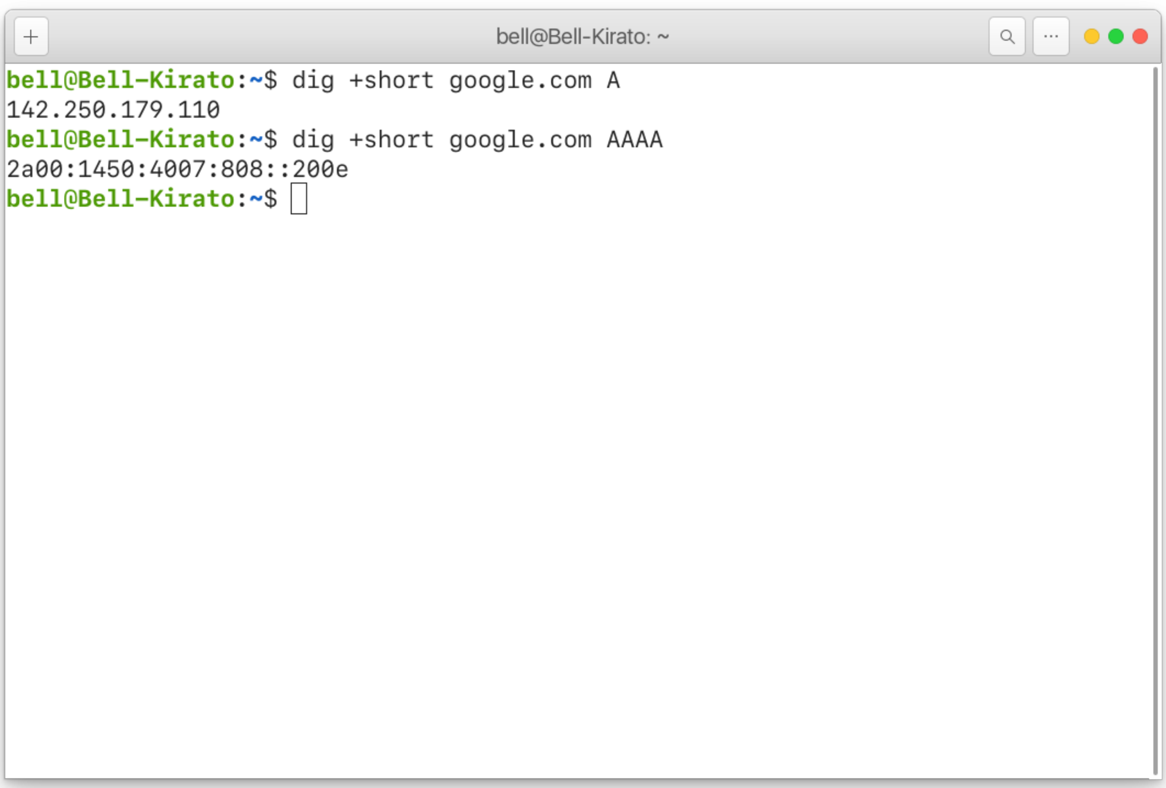


Figure 3.1: Send DNS request to `google.com`

Run the main program to handle outgoing and incoming packets:

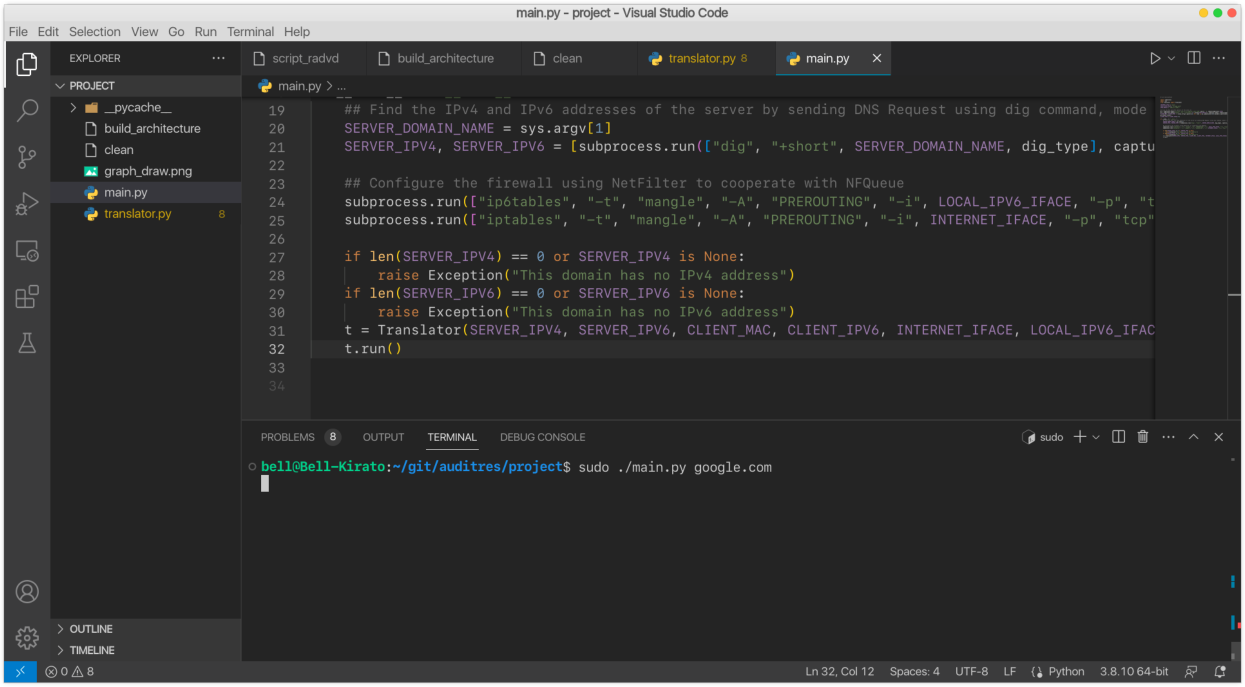


Figure 3.2: Run `main.py`

Run `tcpdump` to sniff packets going to/coming from `google.com` on interface `wlo1` of the router:

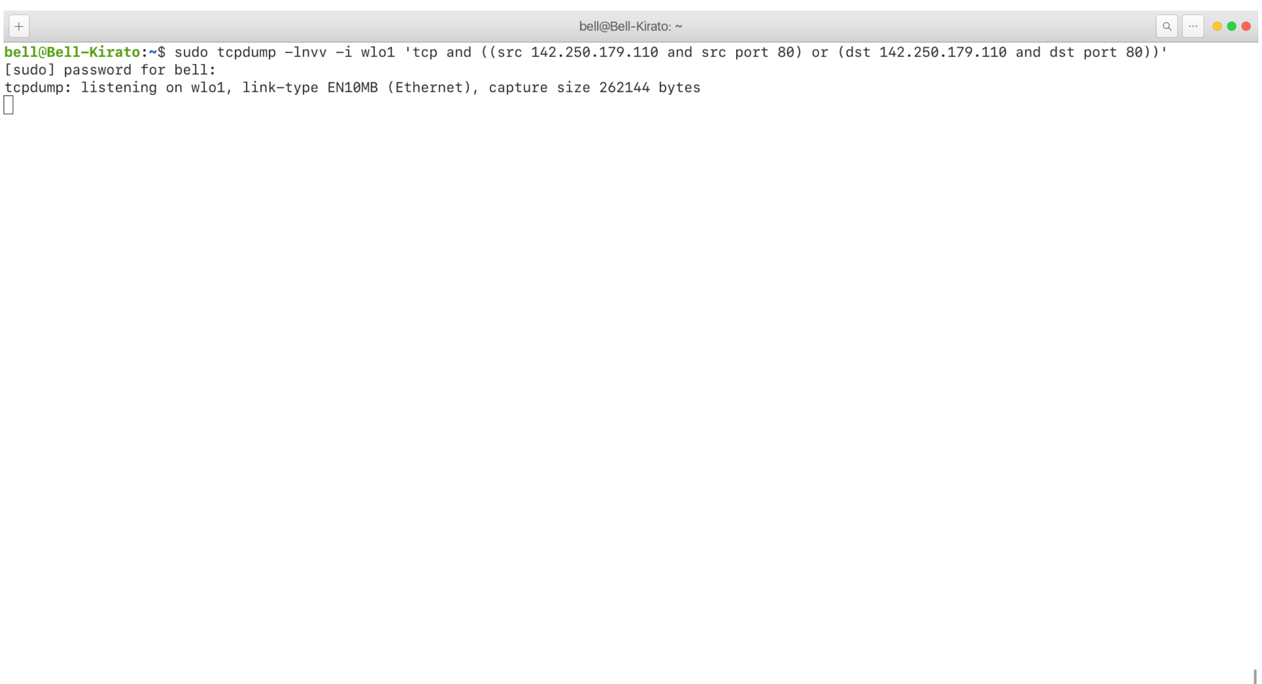


Figure 3.3: Sniff IPv4 packets

Run `tcpdump` to sniff packets going to/coming from `google.com` on interface `eth0` of the client:

Figure 3.4: Sniff IPv6 packets

Use `socat` to establish a connection to `google.com` (using its IPv6 address) from the client:

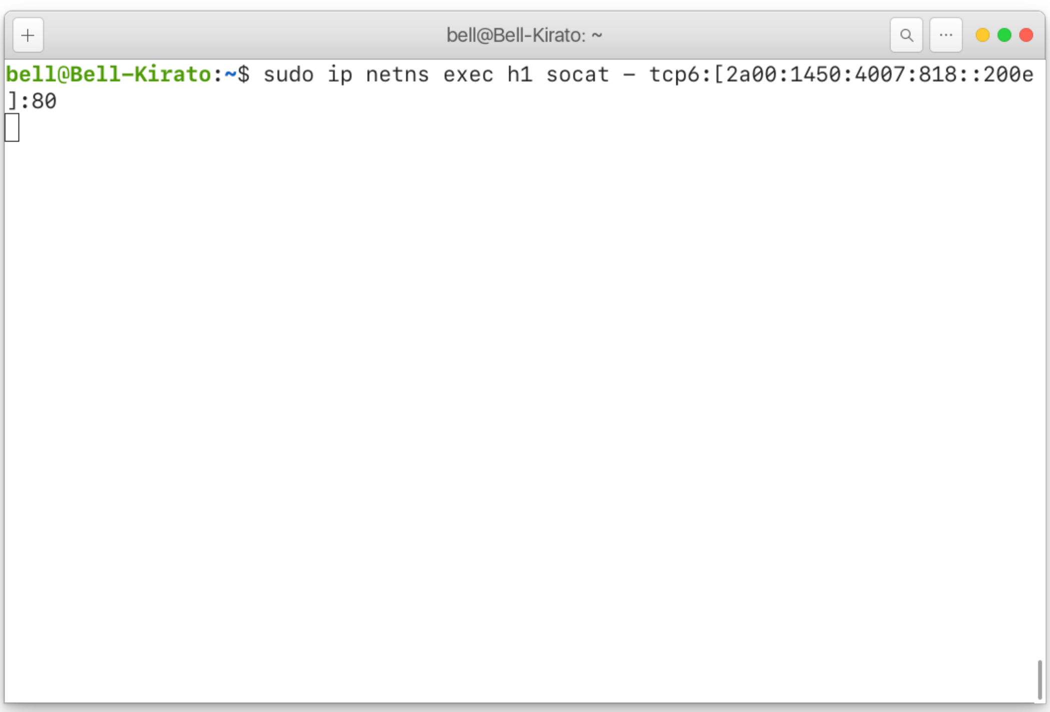
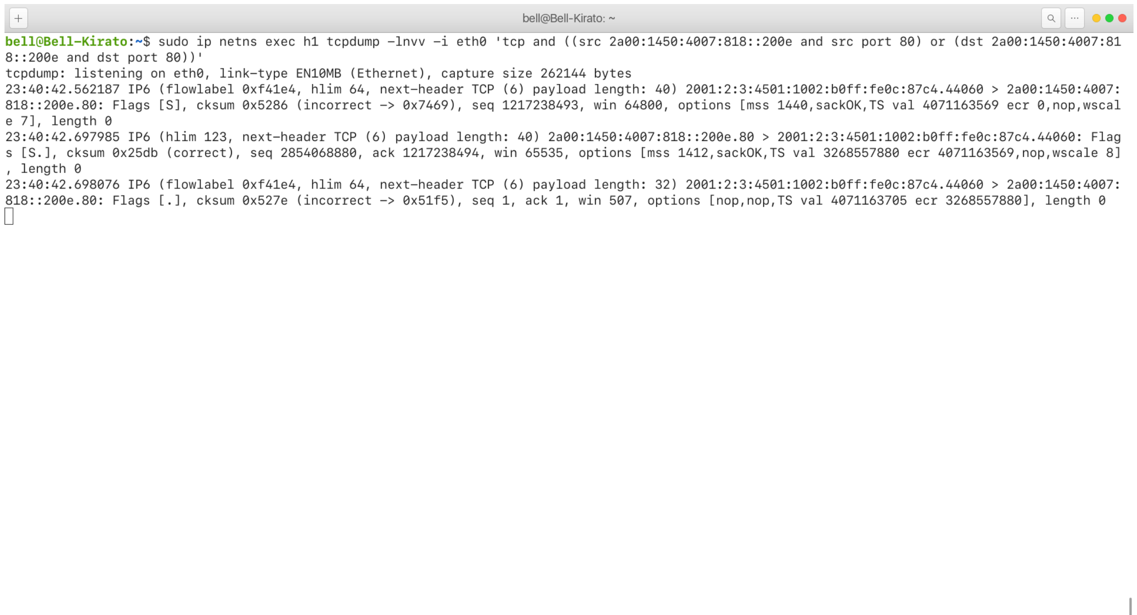


Figure 3.5: Establish connection to IPv6 `google.com` using socat

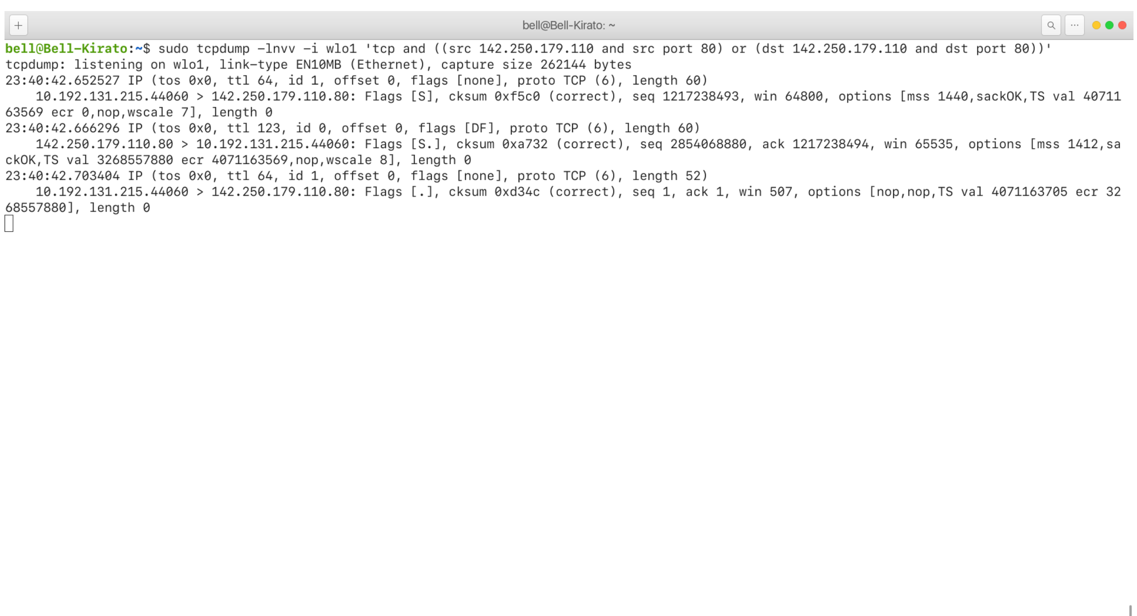
The three-way handshake packets SYN, SYN/ACK, ACK are observed on `eth0` interface of `h1`:



```
bell@Bell-Kirato:~$ sudo ip netns exec h1 tcpdump -lnvv -i eth0 'tcp and ((src 2a00:1450:4007:818::200e and src port 80) or (dst 2a00:1450:4007:81
8::200e and dst port 80))'
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
23:40:42.562187 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 40) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [S], cksum 0x5286 (incorrect -> 0x7469), seq 1217238493, win 64800, options [mss 1440,sackOK,TS val 4071163569 ecr 0,nop,wscal
e 7], length 0
23:40:42.697985 IP6 (hlim 123, next-header TCP (6) payload length: 40) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Flag
s [S.], cksum 0x25db (correct), seq 2854068880, ack 1217238494, win 65535, options [mss 1412,sackOK,TS val 3268557880 ecr 4071163569,nop,wscale 8]
, length 0
23:40:42.698076 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 32) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [.], cksum 0x527e (incorrect -> 0x51f5), seq 1, ack 1, win 507, options [nop,nop,TS val 4071163705 ecr 3268557880], length 0
```

Figure 3.6: 3-way handshake on `eth0` of `h1`

The three-way handshake packets SYN, SYN/ACK, ACK are observed on `wlo1` interface of the router:



```
bell@Bell-Kirato:~$ sudo tcpdump -lnvv -i wlo1 'tcp and ((src 142.250.179.110 and src port 80) or (dst 142.250.179.110 and dst port 80))'
tcpdump: listening on wlo1, link-type EN10MB (Ethernet), capture size 262144 bytes
23:40:42.652527 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 60)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [S], cksum 0xf5c0 (correct), seq 1217238493, win 64800, options [mss 1440,sackOK,TS val 40711
63569 ecr 0,nop,wscale 7], length 0
23:40:42.666296 IP (tos 0x0, ttl 123, id 0, offset 0, flags [DF], proto TCP (6), length 60)
    142.250.179.110.80 > 10.192.131.215.44060: Flags [S.], cksum 0xa732 (correct), seq 2854068880, ack 1217238494, win 65535, options [mss 1412,sa
ckOK,TS val 3268557880 ecr 4071163569,nop,wscale 8], length 0
23:40:42.703404 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0xd34c (correct), seq 1, ack 1, win 507, options [nop,nop,TS val 4071163705 ecr 32
68557880], length 0
```

Figure 3.7: 3-way handshake on `wlo1` of `r1`

Send a HTTP request GET method from the `h1` client then close the connection:

```
bell@Bell-Kirato:~$ sudo ip netns exec h1 socat tcp6:[2a00:1450:4007:818::200e]:
80 -
GET / HTTP/1.1
Host: www.google.com

HTTP/1.1 200 OK
Date: Sun, 16 Apr 2023 15:20:28 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Content-Security-Policy-Report-Only: object-src 'none';base-uri 'self';script-sr
c 'nonce-yDV7ZBiz96P3w0ZxATOJvQ' 'strict-dynamic' 'report-sample' 'unsafe-eval'
'unsafe-inline' https: http:;report-uri https://csp.withgoogle.com/csp/gws/other
-hp
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: AEC=AUEFqZfZQeyd7EAHtjxxHSnb2jRagLamOmdxBDl7l3v86AScsfD0YkwBRP8; exp
ires=Fri, 13-Oct-2023 15:20:28 GMT; path=/; domain=.google.com; Secure; HttpOnly
; SameSite=lax
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked
```

Figure 3.8: Send GET request to `google.com` from `h1`

Data packets going through `wlo1` interface of the router:

```
23:42:54.088518 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x56e7 (correct), seq 39, ack 32201, win 501, options [nop,nop,TS val 4071295058 e
cr 3268688280], length 0
23:42:54.137741 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x50db (correct), seq 39, ack 33601, win 524, options [nop,nop,TS val 4071295134 e
cr 3268688329], length 0
23:42:54.217679 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x4ae0 (correct), seq 39, ack 35001, win 524, options [nop,nop,TS val 4071295198 e
cr 3268688396], length 0
23:42:54.321730 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x4504 (correct), seq 39, ack 36401, win 524, options [nop,nop,TS val 4071295298 e
cr 3268688396], length 0
23:42:54.496433 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x3964 (correct), seq 39, ack 39201, win 524, options [nop,nop,TS val 4071295414 e
cr 3268688456], length 0
23:42:54.653937 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x2d56 (correct), seq 39, ack 42001, win 524, options [nop,nop,TS val 4071295625 e
cr 3268688531], length 0
23:42:54.763606 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x21d2 (correct), seq 39, ack 44801, win 524, options [nop,nop,TS val 4071295729 e
cr 3268688575], length 0
23:42:54.873914 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x15f1 (correct), seq 39, ack 47601, win 524, options [nop,nop,TS val 4071295874 e
cr 3268688671], length 0
23:42:55.022650 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x0eea (correct), seq 39, ack 49139, win 528, options [nop,nop,TS val 4071295989 e
cr 3268688813], length 0
23:43:17.037597 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [F.], cksum 0xb8e0 (correct), seq 39, ack 49139, win 528, options [nop,nop,TS val 4071318013
ecr 3268688813], length 0
23:43:17.053659 IP (tos 0x0, ttl 123, id 37243, offset 0, flags [none], proto TCP (6), length 52)
    142.250.179.110.80 > 10.192.131.215.44060: Flags [F.], cksum 0x5e52 (correct), seq 49139, ack 40, win 256, options [nop,nop,TS val 3268712266
ecr 4071318013], length 0
23:43:17.080511 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 52)
    10.192.131.215.44060 > 142.250.179.110.80: Flags [.], cksum 0x5cfc (correct), seq 40, ack 49140, win 528, options [nop,nop,TS val 4071318083 e
cr 3268712266], length 0
```

Figure 3.9: PUSH and FIN packets on `wlo1` of `r1`

Data packets going through `eth0` interface of `h1`:

```
bell@Bell-Kirato: ~
23:42:54.545696 IP6 (hlim 123, next-header TCP (6) payload length: 1432) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fl
ags [.], cksum 0xe581 (correct), seq 39201:40601, ack 39, win 256, options [nop,nop,TS val 3268688531 ecr 4071294253], length 1400: HTTP
23:42:54.618280 IP6 (hlim 123, next-header TCP (6) payload length: 1432) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fl
ags [P.], cksum 0x1052 (correct), seq 40601:42001, ack 39, win 256, options [nop,nop,TS val 3268688531 ecr 4071294253], length 1400: HTTP
23:42:54.618336 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 32) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [.], cksum 0x527e (incorrect -> 0xabfe), seq 39, ack 42001, win 524, options [nop,nop,TS val 4071295625 ecr 3268688531], lengt
h 0
23:42:54.662096 IP6 (hlim 123, next-header TCP (6) payload length: 1432) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fl
ags [.], cksum 0x2a14 (correct), seq 42001:43401, ack 39, win 256, options [nop,nop,TS val 3268688575 ecr 4071294308], length 1400: HTTP
23:42:54.722460 IP6 (hlim 123, next-header TCP (6) payload length: 1432) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fl
ags [.], cksum 0xba69 (correct), seq 43401:44801, ack 39, win 256, options [nop,nop,TS val 3268688633 ecr 4071294392], length 1400: HTTP
23:42:54.722515 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 32) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [.], cksum 0x527e (incorrect -> 0xa07a), seq 39, ack 44801, win 524, options [nop,nop,TS val 4071295729 ecr 3268688575], lengt
h 0
23:42:54.788305 IP6 (hlim 123, next-header TCP (6) payload length: 1432) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fl
ags [.], cksum 0xb735 (correct), seq 44801:46201, ack 39, win 256, options [nop,nop,TS val 3268688671 ecr 4071294448], length 1400: HTTP
23:42:54.867049 IP6 (hlim 123, next-header TCP (6) payload length: 1432) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fl
ags [.], cksum 0x4c81 (correct), seq 46201:47601, ack 39, win 256, options [nop,nop,TS val 3268688747 ecr 4071294500], length 1400: HTTP
23:42:54.867143 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 32) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [.], cksum 0x527e (incorrect -> 0x9499), seq 39, ack 47601, win 524, options [nop,nop,TS val 4071295874 ecr 3268688671], lengt
h 0
23:42:54.922943 IP6 (hlim 123, next-header TCP (6) payload length: 1432) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fl
ags [.], cksum 0xfdbb (correct), seq 47601:49001, ack 39, win 256, options [nop,nop,TS val 3268688813 ecr 4071294555], length 1400: HTTP
23:42:54.982402 IP6 (hlim 123, next-header TCP (6) payload length: 170) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Fla
gs [P.], cksum 0x6ca6 (correct), seq 49001:49139, ack 39, win 256, options [nop,nop,TS val 3268688886 ecr 4071294623], length 138: HTTP
23:42:54.982460 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 32) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [.], cksum 0x527e (incorrect -> 0x8d92), seq 39, ack 49139, win 528, options [nop,nop,TS val 4071295989 ecr 3268688813], lengt
h 0
23:43:17.005655 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 32) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [F.], cksum 0x527e (incorrect -> 0x3789), seq 39, ack 49139, win 528, options [nop,nop,TS val 4071318013 ecr 3268688813], leng
th 0
23:43:17.076484 IP6 (hlim 123, next-header TCP (6) payload length: 32) 2a00:1450:4007:818::200e.80 > 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060: Flag
s [F.], cksum 0xdcfa (correct), seq 49139, ack 40, win 256, options [nop,nop,TS val 3268712266 ecr 4071318013], length 0
23:43:17.076522 IP6 (flowlabel 0xf41e4, hlim 64, next-header TCP (6) payload length: 32) 2001:2:3:4501:1002:b0ff:fe0c:87c4.44060 > 2a00:1450:4007:
818::200e.80: Flags [.], cksum 0x527e (incorrect -> 0xdba4), seq 40, ack 49140, win 528, options [nop,nop,TS val 4071318083 ecr 3268712266], lengt
h 0
```

Figure 3.10: PUSH and FIN packets on `eth0` of `h1`