BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

IPK – Computer Communications and Networks

ZETA: Packet sniffer

April 24, 2022

Samuel Dobroň (xdobro23)

Contents

1 Literature

This project requires lots of networking knowledge. Definitely I should mentioned, that I had no clue how frames, packets or segments are parsed. They have structures that matches real order of bits transported by physical layer. For example structure of `IPv6` packet header [6]:

```
struct ip6_hdr
  {
    union
      {
        struct ip6_hdrctl
          {
            uint32_t ip6_un1_flow;    /* 4 bits version, 8 bits TC,
                                         20 bits flow-ID */
            uint16_t ip6_un1_plen;    /* payload length */
            uint8_t  ip6_un1_nxt;     /* next header */
            uint8_t  ip6_un1_hlim;    /* hop limit */
          } ip6_un1;
        uint8_t ip6_un2_vfc;       /* 4 bits version, top 4 bits tclass */
      } ip6_ctlun;
    struct in6_addr ip6_src;      /* source address */
    struct in6_addr ip6_dst;      /* destination address */
  };
```

To implement everything required I had to remind myself order of bites in packets and frames. Implementation is inspired by article available here [9]. If something wasn't clear, I've also used `ICMP`[2], `ICMPv6`[7], `(R)ARP`[4] documentations.

2 Implementation

The whole packet sniffer is implemented using `C` programming language. It is supported almost by all the *nix system if it provides `linux/if_arp.h` header file[1]. If not, it also should be pretty easy to complie sniffer without it, because it uses just `arphdr` structure from mentioned file.

2.1 Parsing program arguments

Entrypoint of parsing program arguments is `process_args()` function, which returns a pointer to `sniffer_options_t` structure.

2.1.1 `sniffer_options_t` structure

`sniffer_options_t` is responsible for "holding" the sniffer options. Especially 2 members of this structure are interesting – `L4` and `L3` whose holds information what packets or frames would be captured by sniffer. Bitmasking[5] is used to store these information. It provides ability to add support of filtering another protocols, not mentioned in assignment, much faster than it would require if just some `bool` variable was used.

Currently, at trasport layer `TCP`[3] and `UDP`[1] are supported. If only `--tcp|-t` argument was used, `L4` member of structure would store $1_{(10)}$ in case also `--udp | -u` was provided, `L4` would contain $3_{(10)}$. That means, first bit of `L4` integer represents `TCP` and second bit is there for `UDP`.

Bitmasking is commonly used in Kernel.

[1] Latest version is located for example here: `https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/if_arp.h`

2.1.2 getopt_long

For parsing long (--tcp, --icmp, ...) arguments is used getopt_long() function provided by unistd.h header file. The main idea of parsing arguments using this function is taken from [8].

2.2 Filtering

Filtration of captured packets is performed by set_filter() function. Well, not actually, this function just sets the filter based on provided arguments. Function set_rules() is responsible for generating BPF filter rules, whose are later compiled using pcap_compile() function. There is a interesting macro ADD_RULE which uses sprintf() function to add rules to string containing all the rules. After the compilation of rules to BPF rules, filter is set using the pcap_setfilter() funciton.

There are some limitations, it does not make sense to filter ICMP or (R)ARP frames at some port because these protocol does not use ports at all. That means, combination of --port and --arp or --icmp are not allowed, in that case, sniffer exits with error code 1.

2.3 Capturing packets

Also for capturing the packets or frames pcap's function is used. There are some steps needed to be taken by sniffer:

1. Select the device – performed by select_device() function. The interface performed in program arguments is selected if exists, if not, program ends with exit code 1. Implementation is straightforward, pcap_findalldevs() function is used to get linked list of available devices then, by comparing the name of devices with provided -i argument is selected corresponding device.
2. Open a handler – Handler is opened by pcap_open_live() function.
3. Set BPF filter
4. Start capturing – Firstly, we need to get function, which would be "processor" for incoming frames or packets. There is a function get_handler_function() that returns pointer to handling function. It is implemented in this way, because get_handler_function() can be extended of another handling function for another types of devices pretty easily.
    - Also there is some limitation caused by pcap_loop(). It has an parameter cnt[10] which sets how many packets or frames are captured and then the function returns 0 or some error code. The problem is, parameters cnt is type of int but -n argument has no maximum limitation. So the limitation is set by maximum value of integer on your system.

3 Testing

Sniffer was developed at Red Hat Enterprise Linux 8.5[2]. During development I was using nc[3] and tcpdump tools.

3.1 UDP packet capture

To test if capturing of UDP packets works I've started a nc server with following command:

```
$ nc -u -l 50
```

[2]Available at https://developers.redhat.com/products/rhel/download

[3]More about it at: https://linux.die.net/man/1/nc

Packet sniffer was started as:

```
$ ./ipk-sniffer -i eno1 --port 50 --udp
```

To have something to compare results with, I've used `tcpdump`[4]:

```
$ tcpdump -i eno1 udp port 50 -XX
```

It captured:

```
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eno1, link-type EN10MB (Ethernet), capture size 262144 bytes

08:04:28.425794 IP 10.0.0.1.61270 > XXXX: UDP, length 6
        0x0000:  70b5 e8ef c0e4 204e 7144 7801 0800 4500  p......NqDx...E.
        0x0010:  0022 43c2 0000 3011 f242 0a28 c00f 0a13  ."C...0..B.(....
        0x0020:  807c ef56 0032 000e 77a6 6865 6c6c 6f0a  .|.V.2..w.hello.
        0x0030:  0000 0000 0000 0000 534d c5ce            ........SM..
```

Then, to have something to capture I've wrote simple `hello` to `/dev/udp/10.0.0.1/50` with following command:

```
$ echo -n "hello" > /dev/udp/10.0.0.1/50
```

And here is output of packet sniffer:

```
timestamp: 2022-04-24T08:04:28.763121+01:00
src MAC: 20:4e:71:ff:ff:ff
dst MAC: 70:b5:e8:ff:ff:ff
frame length: 60 bytes
src IP: 10.0.0.15
dst IP: 10.0.0.1
src port: 64043
dst port: 50

0x0000: 45 00 00 22 43 c2 00 00 30 11 f2 42 0a 28 c0 0f  E.."C...0..B.(..
0x0010: 0a 13 80 7c ef 56 00 32 00 0e 77 a6 68 65 6c 6c  ...|.V.2..w.hell
0x0020: 6f 0a 00 00 00 00 00 00 00 00 53 4d c5 ce 00 00  o.........SM....
0x0030: 00 00 00 00 00 00 00 00 00 00 00 00  ............
```

Last part of MAC addresses was replaced by `ff`, similar for IP addresses. We can see little difference between `tcmpdump` and mine packet sniffer. It seems like `tcmpdump` prints some header but i couldn't found out what is going on. According to `pcap_loop` documentation [10] I am printing whole frame, strange.

3.2 Capturing another types of packets or frames

Testing basic functionality of packet sniffer – capturing the packets or frames (`TCP`, `ICMP` and `(R)ARP`) and also `IPv6` was tested in the same way as for `UDP`.

4 Extensions

No extensions are supported, but program is written to easily add new functionality. For example `IPv6` extension headers would require just implement moving pointer of segment behind extension headers.

[4] More about `tcpdump` at: `https://www.tcpdump.org/`

References

[1] *User Datagram Protocol*. RFC 768, Aug. 1980.

[2] *Internet Control Message Protocol*. RFC 792, Sept. 1981.

[3] *Transmission Control Protocol*. RFC 793, Sept. 1981.

[4] *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826, Nov. 1982.

[5] A. AGARWAL, *Bits & bitmasking*. [online], rev. 26. December 2019. [seen. 2022-04-11].

[6] FREE SOFTWARE FOUNDATION, *ipv6.h*. [online], 1991. [seen. 2022-04-15].

[7] M. GUPTA AND A. CONTA, *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. RFC 4443, Mar. 2006.

[8] T. KOENIG AND M. KERRISK, *getopt(3)*. [online], rev. 27. August 2021. [seen. 2022-04-11].

[9] NANODANO, *Using libpcap in c*. [online], rev. 14. August 2015. [seen. 2022-04-15].

[10] THE TCPDUMP GROUP, *Man page of pcap_loop*. [online], rev. 07. March 2022. [seen. 2022-04-15].