Project Documentation

Computer Communications and Networks

Lukáš Foltyn

Zeta - Packet sniffer
2020/2021

Contents

1 Introduction

1.1 Project description

The goal of this project was to implement a light version of packet sniffer. Task has been given in Computer Communications and Networks course at VUT Faculty of information technology.

1.2 Project structure

Project consists of four source code files. The `ipk-sniffer.cpp`'s functionality is mostly to parse command line arguments and catch the raw packets, that are parsed in `PacketInfo` class that is declared in `packet_info.h` and defined in `packet_info.cpp`. For its correct functionality are used structures that can be found in `defined_headers.h`.

2 Implementation

Packet sniffer is implemented in C++ language. For parsing command line arguments is used a `getopt` function [4]. For getting correct offsets in array of bytes (representing a given packet) are used structures from a network header files: `netinet/{ip, in, tcp, udp, ip_icmp, ip6, if_ether}.h`. Catching user's CTRL+C termination signal is handled by functions from `signal.h`, so that the program ends correctly even when unexpected and forced termination is required. Lastly but most importantly, for capturing the packets was installed a libcap library [8], which provides a wide variety of functions to do so. Here is the list of used libcap functions:

- `pcap_findalldevs()`
- `pcap_freealldevs()`
- `pcap_open_live()`
- `pcap_lookupnet()`
- `pcap_compile()`
- `pcap_setfilter()`
- `pcap_loop()`
- `pcap_geterr()`
- `pcap_freecode()`
- `pcap_close()`

2.1 PacketInfo class and its functions

Class consisting of nine functions, that are able to find a particular information contained in the packet and one function that prints out the whole packet with all desired info. It only needs a pointer to packet data and a pointer to `pcap_pkthrd` structure that holds the information about packet lenght as well as the time when the packet was received.

2.1.1 get_network_protocol()

Finds out what kind of network protocol follows after the data link layer. Works only if a given packet has ethernet header, which means 14 bytes offset from the beginning of the packet.

2.1.2 get_transport_protocol()

Finds out what kind of transport protocol follows after the network layer. Both IPv4 and IPv6 are supported as network protocols. Even extension headers are allowed. This functionality is done by looping through headers until a transport protocol header is reached or exception is thrown if unexpected protocol occurs.

2.1.3 get_network_header()

Searches for the position in the packet where a network protocol header starts and returns a pointer to it. Again, as in `get_network_protocol()` function, this works only with ethernet header.

2.1.4 get_transport_header()

Searches for the position in the packet where a transport protocol header starts. Works exactly the same way as a `get_transport_protocol()` function, but instead of protocol type, pointer to a transport protocol header is returned.

2.1.5 get_source_ip() and get_destination_ip()

Functions that find and return source/destination ip address contained in the packet if it's valid, otherwise an exception is thrown. Both class functions `get_network_{protocol/header}` are used here. Ip addresses are converted into a correct format with help of `inet_ntoa()` function for IPv4, `inet_ntop()` function for IPv6. From **ARP** packets the **sender/target** ip address is obtained by bitwise operations.

2.1.6 get_source_port() and get_destination_port()

Functions that find and return source/destination port contained in the packet if it's valid, otherwise an exception is thrown. Two more class functions are used here - `get_transport_header()` and `get_transport_protocol()`. In this case, valid protocols are TCP and UDP.

2.1.7 get_timestamp()

Looks into the `pcap_pkthrd` structure for the time when the packet was received and returns it converted in a RFC3339 format [5].

2.1.8 print_packet()

This function is used for printing the whole packet in a similar format as Wireshark [9]. On the first line we can found time when the packet was received then source ip address/port followed by destination ip address/port and the length of the captured packet. Then the packet data are printed from the first to the last byte in hexadecimal format on the left side as well as in ASCII format on the right side. Non-printable characters are replaced with dot.

```
2021-04-25T14:46:35.751+1:00 13.32.11.231 : 80 > 192.168.15.112 : 52956, lenght 68
0x0000:  74 70 fd 26 86 94 60 e3  27 9f 70 9a 08 00 45 00 tp.&..`. '.p...E.
0x0010:  00 34 f4 96 00 00 f2 06  eb 0d 0d 20 0b e7 c0 a8 .4...... ... ....
0x0020:  0f 70 00 50 ce dc 7d af  f6 47 b9 5f 11 1d 80 10 .p.P..}. .G._....
0x0030:  00 83 02 2e 00 00 01 01  08 0a 31 e7 5b 5a fc df ........ ..1.[Z..
0x0040:  f3 2a c5 ba                                     .*..
```

Figure 1: Example of captured packet and its format

3 Testing

This project was not tested by any other program or script. In this case, more of a static testing approach was chosen. Both **ipk-sniffer** and **Wireshark** were run and with the help of a small python script, different kinds of packets were generated and sent. Then the outputs were compared. You can see some examples on the next pages.

```
2021-04-25T15:58:03.269+1:00 ff02::16 : 854 > fe80::8ac:68e9:d469:d421 : 5541, lenght 296
0x0000:  ff ff ff ff ff ff 74 70  fd 26 86 94 86 dd 60 00 ......tp .&....`.
0x0010:  00 00 00 f2 00 40 ff 02  00 00 00 00 00 00 00 00 .....@.. ........
0x0020:  00 00 00 00 00 16 fe 80  00 00 00 00 00 00 08 ac ........ ........
0x0030:  68 e9 d4 69 d4 21 2b 04  01 02 00 00 c9 10 00 00 h..i.!+. ........
0x0040:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00 ........ ........
0x0050:  c2 04 40 00 00 00 05 02  00 00 01 02 00 00 06 08 ..@..... ........
0x0060:  00 04 00 00 00 00 fe 80  00 00 00 00 00 00 08 ac ........ ........
0x0070:  68 e9 d4 69 d4 21 aa ab  00 00 00 00 00 00 08 ac h..i.!.. ........
0x0080:  68 e9 d4 69 d4 21 fe 80  00 00 00 00 00 00 08 ac h..i.!.. ........
0x0090:  68 e9 d4 69 cc da aa aa  00 00 00 00 00 00 00 00 h..i.... ........
0x00a0:  00 00 00 00 aa aa 03 56  15 a5 00 00 00 00 00 00 .......V ........
0x00b0:  00 00 50 02 20 00 54 17  00 00 54 45 53 54 20 2d ..P. .T. ..TEST -
0x00c0:  20 49 50 76 36 20 77 69  74 68 20 48 6f 70 20 62  IPv6 wi th Hop b
0x00d0:  79 20 48 6f 70 20 65 78  74 65 6e 73 69 6f 6e 20 y Hop ex tension
0x00e0:  68 65 61 64 65 72 20 61  6e 64 20 6f 70 74 69 6f header a nd optio
0x00f0:  6e 73 20 61 6e 64 20 77  69 74 68 20 52 6f 75 74 ns and w ith Rout
0x0100:  69 6e 67 20 65 78 74 65  6e 73 69 6f 6e 20 68 65 ing exte nsion he
0x0110:  61 64 65 72 20 61 6e 64  20 61 64 64 72 65 73 73 ader and  address
0x0120:  65 73 20 2b 20 54 43 50                          es + TCP
```

```
▸ Frame 412: 296 bytes on wire (2368 bits), 296 bytes captured (2368 bits) on interface wlo1, id 0
▸ Ethernet II, Src: IntelCor_26:86:94 (74:70:fd:26:86:94), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Internet Protocol Version 6, Src: ff02::16, Dst: fe80::8ac:68e9:d469:d421
▸ Transmission Control Protocol, Src Port: 854, Dst Port: 5541, Seq: 0, Len: 110
▸ Data (110 bytes)
```

```
0000  ff ff ff ff ff ff 74 70  fd 26 86 94 86 dd 60 00   ......tp ·&····`·
0010  00 00 00 f2 00 40 ff 02  00 00 00 00 00 00 00 00   .....@·· ········
0020  00 00 00 00 00 16 fe 80  00 00 00 00 00 00 08 ac   ········ ········
0030  68 e9 d4 69 d4 21 2b 04  01 02 00 00 c9 10 00 00   h··i·!+· ········
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00   ········ ········
0050  c2 04 40 00 00 00 05 02  00 00 01 02 00 00 06 08   ··@····· ········
0060  00 04 00 00 00 00 fe 80  00 00 00 00 00 00 08 ac   ········ ········
0070  68 e9 d4 69 d4 21 aa ab  00 00 00 00 00 00 08 ac   h··i·!·· ········
0080  68 e9 d4 69 d4 21 fe 80  00 00 00 00 00 00 08 ac   h··i·!·· ········
0090  68 e9 d4 69 cc da aa aa  00 00 00 00 00 00 00 00   h··i···· ········
00a0  00 00 00 00 aa aa 03 56  15 a5 00 00 00 00 00 00   ·······V ········
00b0  00 00 50 02 20 00 54 17  00 00 54 45 53 54 20 2d   ··P· ·T· ··TEST -
00c0  20 49 50 76 36 20 77 69  74 68 20 48 6f 70 20 62    IPv6 wi th Hop b
00d0  79 20 48 6f 70 20 65 78  74 65 6e 73 69 6f 6e 20   y Hop ex tension
00e0  68 65 61 64 65 72 20 61  6e 64 20 6f 70 74 69 6f   header a nd optio
00f0  6e 73 20 61 6e 64 20 77  69 74 68 20 52 6f 75 74   ns and w ith Rout
0100  69 6e 67 20 65 78 74 65  6e 73 69 6f 6e 20 68 65   ing exte nsion he
0110  61 64 65 72 20 61 6e 64  20 61 64 64 72 65 73 73   ader and  address
0120  65 73 20 2b 20 54 43 50                           es + TCP
```

Figure 2: IPv6/TCP packet with extension headers

```
2021-04-25T16:06:11.70+1:00 123.123.123.123 : 854 > 185.51.241.138 : 5541, lenght 134
0x0000:  60 e3 27 9f 70 9a 74 70  fd 26 86 94 08 00 45 00 `.'.p.tp .&....E.
0x0010:  00 78 00 01 00 00 40 33  d8 9d 7b 7b 7b 7b b9 33 .x....@3 ..{{{{.3
0x0020:  f1 8a 11 05 00 00 00 00  00 00 00 00 00 00 61 62 ........ ......ab
0x0030:  63 64 65 66 67 68 69 6a  6b 6c 6d 6e 6f 70 03 56 cdefghij klmnop.V
0x0040:  15 a5 00 48 00 00 54 45  53 54 20 2d 20 49 50 76 ...H..TE ST - IPv
0x0050:  34 20 2b 20 41 75 74 68  65 6e 74 69 63 61 74 69 4 + Auth enticati
0x0060:  6f 6e 20 65 78 74 65 6e  73 69 6f 6e 20 68 65 61 on exten sion hea
0x0070:  64 65 72 20 77 69 74 68  20 70 61 79 6c 6f 61 64 der with  payload
0x0080:  20 2b 20 55 44 50                                + UDP
```

```
▸ Frame 49: 134 bytes on wire (1072 bits), 134 bytes captured (1072 bits) on interface wlo1, id 0
▸ Ethernet II, Src: IntelCor_26:86:94 (74:70:fd:26:86:94), Dst: Tp-LinkT_9f:70:9a (60:e3:27:9f:70:9a)
▸ Internet Protocol Version 4, Src: 123.123.123.123, Dst: 185.51.241.138
▸ Authentication Header
▸ User Datagram Protocol, Src Port: 854, Dst Port: 5541
▸ Data (64 bytes)
```

```
0000  60 e3 27 9f 70 9a 74 70  fd 26 86 94 08 00 45 00   `·'·p·tp ·&····E·
0010  00 78 00 01 00 00 40 33  d8 9d 7b 7b 7b 7b b9 33   ·x····@3 ··{{{{·3
0020  f1 8a 11 05 00 00 00 00  00 00 00 00 00 00 61 62   ········ ······ab
0030  63 64 65 66 67 68 69 6a  6b 6c 6d 6e 6f 70 03 56   cdefghij klmnop·V
0040  15 a5 00 48 00 00 54 45  53 54 20 2d 20 49 50 76   ···H··TE ST - IPv
0050  34 20 2b 20 41 75 74 68  65 6e 74 69 63 61 74 69   4 + Auth enticati
0060  6f 6e 20 65 78 74 65 6e  73 69 6f 6e 20 68 65 61   on exten sion hea
0070  64 65 72 20 77 69 74 68  20 70 61 79 6c 6f 61 64   der with  payload
0080  20 2b 20 55 44 50                                  + UDP
```

Figure 3: IPv4/UDP packet with authentication header

```
2021-04-25T16:06:11.135+1:00 123.123.123.123 : no port > 185.51.241.138 : no port, lenght 131
0x0000:  60 e3 27 9f 70 9a 74 70  fd 26 86 94 08 00 45 00 `.'.p.tp .&....E.
0x0010:  00 75 00 01 00 00 40 33  d8 a0 7b 7b 7b 7b b9 33 .u....@3 ..{{{{.3
0x0020:  f1 8a 01 04 00 00 00 00  00 00 00 00 00 00 61 62 ........ ......ab
0x0030:  63 64 65 66 67 68 69 6a  6b 6c 08 00 3c 8a 00 00 cdefghij kl..<...
0x0040:  00 00 54 45 53 54 20 2d  20 49 50 76 34 20 2b 20 ..TEST -  IPv4 +
0x0050:  41 75 74 68 65 6e 74 69  63 61 74 69 6f 6e 20 65 Authenti cation e
0x0060:  78 74 65 6e 73 69 6f 6e  20 68 65 61 64 65 72 20 xtension  header
0x0070:  77 69 74 68 20 70 61 79  6c 6f 61 64 20 2b 20 49 with pay load + I
0x0080:  43 4d 50                                         CMP
```

```
▸ Frame 50: 131 bytes on wire (1048 bits), 131 bytes captured (1048 bits) on interface wlo1, id 0
▸ Ethernet II, Src: IntelCor_26:86:94 (74:70:fd:26:86:94), Dst: Tp-LinkT_9f:70:9a (60:e3:27:9f:70:9a)
▸ Internet Protocol Version 4, Src: 123.123.123.123, Dst: 185.51.241.138
▸ Authentication Header
▸ Internet Control Message Protocol
```

```
0000  60 e3 27 9f 70 9a 74 70  fd 26 86 94 08 00 45 00   `·'·p·tp ·&····E·
0010  00 75 00 01 00 00 40 33  d8 a0 7b 7b 7b 7b b9 33   ·u····@3 ··{{{{·3
0020  f1 8a 01 04 00 00 00 00  00 00 00 00 00 00 61 62   ········ ······ab
0030  63 64 65 66 67 68 69 6a  6b 6c 08 00 3c 8a 00 00   cdefghij kl··<···
0040  00 00 54 45 53 54 20 2d  20 49 50 76 34 20 2b 20   ··TEST -  IPv4 +
0050  41 75 74 68 65 6e 74 69  63 61 74 69 6f 6e 20 65   Authenti cation e
0060  78 74 65 6e 73 69 6f 6e  20 68 65 61 64 65 72 20   xtension  header
0070  77 69 74 68 20 70 61 79  6c 6f 61 64 20 2b 20 49   with pay load + I
0080  43 4d 50                                          CMP
```

Figure 4: IPv4/ICMP packet with authentication headers

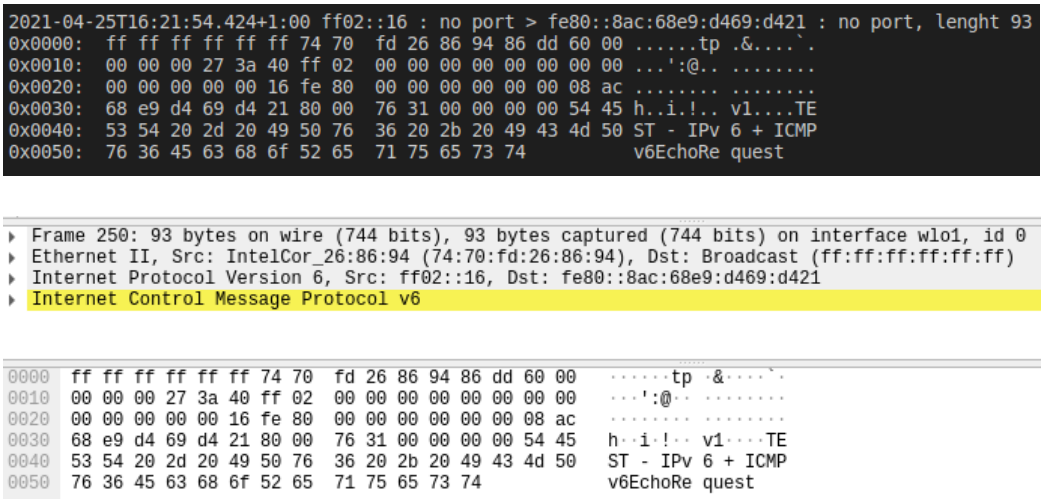

Figure 5: IPv6/ICMPv6 packet

References

[1] Tim Carstens. *Programing with pcap*. URL: https://www.tcpdump.org/pcap.html.

[2] *Filter-tcpdump*. URL: https://www.tcpdump.org/manpages/pcap-filter.7.html.

[3] *IPv6 header format*. URL: https://en.wikipedia.org/wiki/IPv6_packet.

[4] Michael Kerrisk. *Linux Programmer's Manual GETOPT(3)*. URL: https://man7.org/linux/man-pages/man3/getopt.3.html.

[5] G. Klyne and C. Newman. *Date and Time on the Internet: Timestamps*. URL: https://tools.ietf.org/html/rfc3339.

[6] Luis MartinGarcia. *Programming with Libpcap - Sniffing the network from our own application*. URL: http://recursos.aldabaknocking.com/libpcapHakin9LuisMartinGarcia.pdf.

[7] David C. Plummer. *An Ethernet Address Resolution Protocol*. URL: https://tools.ietf.org/html/rfc826.

[8] *Tcpdump*. URL: https://www.tcpdump.org/.

[9] *Wireshark*. URL: https://www.wireshark.org/.