Computer Communications and Networks

Project 2 Documentation

ZETA: Packet Sniffer

David Mihola (xmihol00)

24. 4. 2021

Contents

1 Introduction

This document describes the logical structure of the Packet Sniffer source code and the approach used to accomplish the assigned task. The more technical details (f.e. signature of implemented functions) are documented directly in the source code.

2 Libraries

The main functionality of the program is implemented with the use of the `pcap` library [1]. I chose this library other than the `libnet` library mainly because of better documentation. Other libraries than the standard `C` and `C++` were not used.

3 File Structure

The whole file structure contains seven files. The program itself consist only of three of them, which are:

- `main.cpp` – the main function of the program.
- `sniffer.h` – constant values definitions, function declarations, inline function definitions and their documentation.
- `sniffer.cpp` – constant values definitions and function definitions.

4 Functions

Implemented functions are divided into non inline and inline. The non inline functions implement some necessary functionality and are usually of a larger span. On the other hand the inline functions are short and used as auxiliary functions mainly for converting the byte representation of a packet to some more useful representation (f.e. string format).

4.1 Non Inline Functions

- `free_resources()` – frees allocated resources by the `pcap_open_live()` function.
- `sig_int_handler()` – handles the program termination by the `SIGINT (ctrl + c)` signal.
- `get_connection()` – retrieves current live connection established by the `pcap_open_live()` function.

- `parse_arguments()` – parses command line arguments using the `getopt` library. Creates a packet filter according to the `pcap-filter` specification from the specified packet types by the command line arguments.
- `print_packet()` – prints packet data in a format specified by the assignment with a use of `std::cout` manipulation. The `std::cout` is changed to print integers as two hexadecimal values, each byte of the packet data is then converted to an integer and printed.
- `packet_parser()` – parses received packets. Firstly, the packet arrival time is parsed with a `parse_time()` function. Secondly, parses the data link layer, in our case only the ethernet type, with a function `parse_ethernet_header()`. Thirdly, the network layer is parsed with a function `parse_network_layer()`. Parsing of a packet also includes navigating through the IP extension headers, if there are any, which is done by a function `parse_extension_headers()`. Lastly, the destination and source ports are retrieved from the transport layer by a function `parse_transport_layer()`, if possible. And the packet is printed either with the retrieved ports or without them with the use of the `print_packet()` function.
- `create_pcap_connection()` – opens a live pcap connection with a specified packet filter by the command line arguments and checks that the interface uses an ethernet data link layer. Functions `pcap_lookupnet()`, `pcap_compile()`, `pcap_setfilter()`, `pcap_freecode()` and `pcap_datalink()` are used to achieve that.
- `parse_network_layer()` – parses the network layer, which in our case includes the IPv4 [2], IPv6 [3] and ARP [4] protocols. Text representation of IPv4 and IPv6 addresses is retrieved from the packet with functions `inet_ntoa()` and `inet_ntop()` respectively and the IPv4 address with `sprintf()` in case of an ARP packet. The header type of the following header, which can also be an extension header, is retrieved as well.
- `parse_transport_layer()` – retrieves the destination and source ports in case of TCP [5] and UDP [6] headers with a function `parse_tcp_udp()`, otherwise informs that the source and destination ports cannot be obtained.
- `print_all_interfaces()` – prints all accessible interfaces on a machine to `STDOUT` with the use of a `pcap_findalldevs()` function.

4.2 Inline Functions

- `print_usage_message()` – prints a usage message on `STDOUT` when desired by the `--help` option or to `STDERR`, when the program arguments are entered incorrectly.

- `parse_mac_address()` – parses a MAC address from a byte representation to adequate text representation.
- `parse_ethernet_header()` – parses the ethernet data link layer [7] and retrieves the type [8] of the following network layer and MAC addresses. The source and the destination MAC address is parsed by the function `parse_mac_address()`.
- `parse_time()` – parses the packet receival time to a format satisfactory by the `RFC3339` [9] standard.
- `parse_tcp_udp()` – retrieves the source and the destination port from TCP and UDP headers.
- `parse_extension_headers()` – parses the IPv6 extension headers [10], which are `Hop-by-Hop Options`, `Destination Options`, `Routing Header`, `Fragment Header`, `No Next Header` and the extension headers shared between IPv4 and IPv6, which are `Authentication Header` [11] and `Encapsulation Security Payload Header` [12]. Retrieves the header type of the following transport layer, if it can be obtained, otherwise `No Next Header` is used.

5 Testing

The Packet Sniffer was tested with a use of python script `packet_generator.py` included in the file structure. It relies on a `scapy` library [13] [14] to generate packets. The constructions in the script build and generate various test packets, which cover majority of packet combinations of the OSI network and transport layers. When the Packet Sniffer as well as Wireshark, which I consider as a reliable reference, are running, the python script is executed. Then the results of such a session are compared.

6 Additional functionality

The additional functionality can be specified by command line arguments, see below. It consists of printing the source and the destination MAC address, printing the types of headers of layers of the OSI model including extension headers. Furthermore specifying the IPv4 and IPv6 addresses, on which packets are scanned. This functionality was added mainly for more convenient testing, but also the user can benefit from more specific information about a certain packet.

7 Usage

Firstly, the program must be compiled with GNU make using command `make`, binary file `ipk-sniffer` and others will be generated. Then run the program as:
`sudo ./ipk-sniffer [options ...]`

Options:

`-h, --help` Prints a usage message to `STDOUT`.

`-i, --interface <optional string>` Option can be either followed by a name of an interface on which packets are scanned, or all available interfaces are printed, if the option parameter is not specified or the option is missing entirely.

`-p <unsigned short>` A port number on which the traffic is going to be scanned.

`-t, --tcp` Only TCP and other specified packets are going to be scanned.

`-u, --udp` Only UDP and other specified packets are going to be scanned.

`--arp` Only ARP frame packets are going to be scanned.

`--icmp` Only ICMP, ICMPv6 and other specified packets are going to be scanned.

`-n <int>` The number of packets to be scanned, 1 if not specified.

`--mac` Prints the source and destination MAC addresses of each packet.

`--type` Prints the types of headers of layers of the OSI model including extension headers.

`--hostv4 <IPv4 address>` Scans only packets with either destination or source IPv4 address specified in the argument.

`--hostv6 <IPv6 address>` Scans only packets with either destination or source IPv6 address specified in the argument.

8 Conclusion

This project was quite different from other projects that I have done on this faculty. There are not that many tutorials [15] or examples regarding this topic, so I had to study the documentation of used libraries a lot more than normally and even look into RFC standards. As a result, I have learnt a lot more than by simply following tutorials and examples.

9 References

[1] Van Jacobson, Craig Leres and Steven McCanne (2020, Septemper 9) *pcap - Packet Capture library*. https://man7.org/linux/man-pages/man3/pcap.3pcap.html

[2] Postel, J. (1981, September) *Internet Protocol* [RFC0791]. https://tools.ietf.org/html/rfc791

[3] Deering & Hinden (1995, December) *Internet Protocol, Version 6 (IPv6) Specification* [RFC1883]. https://tools.ietf.org/html/rfc1883

[4] David C. Plummer (1982, November) *An Ethernet Address Resolution Protocol* [RFC0826]. https://tools.ietf.org/html/rfc826

[5] Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291 (1981, September) *TRANSMISSION CONTROL PROTOCOL* [RFC0793]. https://tools.ietf.org/html/rfc793

[6] J. Postel (1980, August 28) *User Datagram Protocol* [RFC0768]. https://tools.ietf.org/html/rfc768

[7] The Tcpdump Group (2021, January 29) *LINK-LAYER HEADER TYPES*. https://www.tcpdump.org/linktypes.html

[8] Internet Assigned Numbers Authority (2021, February 26) *Protocol Numbers*. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

[9] Klyne, et. al. (2002, July) *Date and Time on the Internet: Timestamps* [RFC3339]. https://tools.ietf.org/html/rfc3339

[10] Cisco Systems, Inc (2006, October) *IPv6 Extension Headers Review and Considerations* [PDF]. https://www.cisco.com/en/US/technologies/tk648/tk872/technologies_white_paper0900aecd8054d37d.pdf

[11] Atkinson, R. (1995, August) *IP Authentication Header* [RFC1826]. https://tools.ietf.org/html/rfc1826

[12] Atkinson, R. (1995, August) *IP Encapsulating Security Payload (ESP)* [RFC1827]. https://tools.ietf.org/html/rfc1827

[13] Philippe Biondi and the Scapy community. (2021, Apr 07) *Welcome to Scapy's documentation!*. https://scapy.readthedocs.io/en/latest/

[14] Eggert, Oliver (2012, January 19) *IPv6 Packet Creation With Scapy Documentation* [PDF]. https://www.idsv6.de/Downloads/IPv6PacketCreationWithScapy.pdf

[15] thenewboston (2015, December 29) *Python Network Packet Sniffer Tutorials* [Video Series]. YouTube. https://youtube.com/playlist?list=PL6gx4Cwl9DGDdduy0IPDDHYnUx66Vc4ed