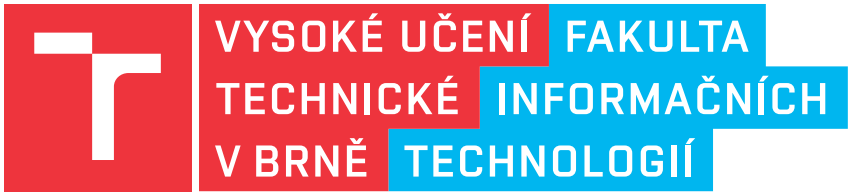

Packets sniffer

IPK - Computer Communications and Networks

April 24, 2022 Shchapaniak Andrei (xshcha00)

Contents

1 Introduction **2**

2 Implementation **2**
2.1 Files 2
2.2 Command line parameters parsing 2
2.3 Filter creating 3
2.4 Timestamp converting 3
2.5 Packets processing 3

3 Tests **4**
3.1 Test file packets.py 4
3.2 Testing results 4

Bibliography **7**

1 Introduction

The goal of the project was to implement a network analyzer, which would be able to capture and filter packets on a certain network interface. This project was implemented in C++ language, because it has more useful functionality which makes programming more efficient. The program must be run with sudo privileges to allow raw packet captures and network interface manipulation.

2 Implementation

2.1 Files

- `sniffer.cpp` contains the function `main()` where the filter for the network interface is created. In the section 2.3 it is described in detail.
- `disassembler.cpp` is the main source file which contains a function `packets_handler()` that processes captured packets 2.5. It also includes the implementation of all functions for the program.
- `disassembler.h` contains the declaration of the class `Rule` for the rules of the filter and functions with appropriate comments for the doxygen[1] documentation, and definition of macros.
- `Makefile` is for effective and automation compiling of the program.
- `packets.py` is for generating tests. A detailed description of it can be found in the section 3.1.
- `README.md` file with a brief description of the program and running examples.
- `manual.pdf` is documentation containing parts of the implementation and testing of the program.

2.2 Command line parameters parsing

It was implemented using the function `getopt()`. In case of entering only one parameter `--help` will be written a help message. In the case of a wrong sequence of arguments, a usage message will be written to the standard output. If the user has not specified a network interface, all available interfaces on the machine will be printed.

Brief description of the meaning of the parameters:

- `-i interface`: name of the interface to listen to.
- `-n number`: count of the captured packets. By default, it is set to one packet.
- `-p port`: on this port will be packet filtering performed. The given port can occur in both the source and destination part.
- `-t --tcp`: display only TCP packets [3].
- `-u --udp`: display only UDP packets [1].
- `--arp`: display only ARP packets [4].
- `--icmp`: display only ICMPv4 [2], [5] a ICMPv6 packets [6].

[1] https://www.doxygen.nl/index.html

2.3 Filter creating

Based on the command line arguments is generated a rule[2] for the filter. I used library `<pcap/pcap.h>` to create a filter for network interface, name of which must be set as input argument. Using function `pcap_open_live()` network interface will be set to promiscuous mode[3] and will be set a buffer timeout to 1 second. Then function `pcap_loop()` waits for incoming packets and call a callback function `packets_handler()`.

2.4 Timestamp converting

When extracting the timestamp from the structure `pcap_pkthdr` and converting it to the format according to RFC3339 [7], I was inspired by the solution from stackoverflow[4]. The most interesting part was an extraction of the time zone from the structure `timeval`. Code of this function is shown on the listing 1.

```
void print_time(struct timeval ts) {
    char tmbuf[90], zone[10];
    time_t timestamp = ts.tv_sec;
    struct tm* tm_loc = localtime(&timestamp);
    std::string zone_f;

    strftime(tmbuf, sizeof(tmbuf), "%Y-%m-%dT%H:%M:%S", tm_loc);
    strftime(zone, sizeof(zone), "%z", tm_loc);
    zone_f.insert(zone_f.end(), {zone[0], zone[1], zone[2], ':', zone[3], zone[4]});

    zone_f = !strcmp("+0000", zone) ? std::string(1, 'Z') : zone_f;
    printf("timestamp: %s.%06ld%s\n", tmbuf, ts.tv_usec, zone_f.c_str());
}
```

Listing 1: Time converting

2.5 Packets processing

For these purposes is implemented function `packets_handler()`, which is called every time the packet arrives at the network interface. This function analyzes incoming packets and prints appropriate information from the packet's frames. Information about every packet must consist of:

- `timestamp`
- `[src|dst] MAC addr`
- `length of the packet in bytes`
- `[src|dst] IP4/6 addr (if exists)`
- `[src|dst] port (if exists)`
- `type (if icmp/icmpv6)`
- `code (if icmp/icmpv6)`
- `package contents`

An example with these values is shown in the figure 5.

[2] https://linux.die.net/man/7/pcap-filter
[3] https://en.wikipedia.org/wiki/Promiscuous_mode
[4] https://stackoverflow.com/questions/2408976/struct-timeval-to-printable-format

3 Tests

For the program testing were used different tools that work with network packets. For accurate results were used TCP and UDP ports that are not reserved for the operation of network applications.

3.1 Test file packets.py

I have decided to generate packets using Python library `scapy`[5]. It helps me to know accurate information, which packet must contain. After tests are generated i can replay traffic in these packets to my network interface using a tool `tcpreplay`[6]. Generated traffic you can find in the script `packets.py`.

3.2 Testing results

```
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.104.128  netmask 255.255.255.0  broadcast 192.168.104.255
        inet6 fe80::9228:300e:dd2c:9351  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:a1:dc:c1  txqueuelen 1000  (Ethernet)
        RX packets 80  bytes 18164 (18.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 117  bytes 13288 (13.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 1: Network interface on which traffic will be captured.

```
root@student-vm:/home/student/ipk_proj2# grep "\['example'\]" packets.py
    dict_dict_pkts['example']['layer2']  = Ether(dst=mac_addr_interface, src="00:11:22:33:44:55")
    dict_dict_pkts['example']['layer3']  = IP(dst=ip_addr_interface, src="190.190.13.1")
    dict_dict_pkts['example']['layer4']  = TCP(sport=60001, dport=12345)
    dict_dict_pkts['example']['message'] = "()()(_-_-_!!!Andrei Shchapaniak, xshcha00!!!_-_-_)()()"
```

Figure 2: Contents of the packet for test.

```
root@student-vm:/home/student/ipk_proj2# sudo tcpreplay -i ens33 pkt_example.pcap
Actual: 1 packets (108 bytes) sent in 0.000003 seconds
Rated: 36000000.0 Bps, 288.00 Mbps, 333333.33 pps
Statistics for network device: ens33
        Successful packets:        1
        Failed packets:            0
        Truncated packets:         0
```

Figure 3: The output of the tcpreplay tool. The packet was sent successfully.

[5] https://scapy.readthedocs.io/en/latest/usage.html
[6] https://linux.die.net/man/1/tcpreplay

```
root@student-vm:/home/student# tshark -f "tcp port 60001" -T json -i ens33
 -a packets:1 > tshark_pkt.json
Running as user "root" and group "root". This could be dangerous.
Capturing on 'ens33'
1
root@student-vm:/home/student# grep -E '"frame.time"|"frame.len"|"eth.dst"
|"eth.src"|"ip.dst"|"ip.src"|"tcp.srcport"|"tcp.dstport"' tshark_pkt.json
          "frame.time": "Apr 19, 2022 23:47:34.136371806 CEST",
          "frame.len": "108",
          "eth.dst": "00:0c:29:a1:dc:c1",
          "eth.src": "00:11:22:33:44:55",
          "ip.src": "190.190.13.1",
          "ip.dst": "192.168.104.128",
          "tcp.srcport": "60001",
          "tcp.dstport": "12345",
```

Figure 4: The output of the tshark tool.

```
root@student-vm:/home/student/ipk_proj2# ./ipk-sniffer --tcp -p 60001 -i ens33
********************************************************************************

timestamp: 2022-04-19T23:47:34.136371+02:00
src MAC: 00:11:22:33:44:55
dst MAC: 00:0c:29:a1:dc:c1
frame length: 108 bytes
src IP: 190.190.13.1
dst IP: 192.168.104.128
src port: 60001
dst port: 12345

0x0000  00 0c 29 a1 dc c1 00 11 22 33 44 55 08 00 45 00  ..)....."3DU..E.
0x0010  00 5e 00 01 00 00 40 06 85 b1 be be 0d 01 c0 a8  .^....@.........
0x0020  68 80 ea 61 30 39 00 00 00 00 00 00 00 00 50 02  h..a09........P.
0x0030  20 00 ba 9d 00 00 28 29 28 29 28 5f 2d 5f 2d 5f   .....()()(_-_-_
0x0040  21 21 21 41 6e 64 72 65 69 20 53 68 63 68 61 70  !!!Andrei Shchap
0x0050  61 6e 69 61 6b 2c 20 78 73 68 63 68 61 30 30 21  aniak, xshcha00!
0x0060  21 21 5f 2d 5f 2d 5f 29 28 29 28 29 00           !!_-_-_)()()

********************************************************************************
```

Figure 5: The output of the program.

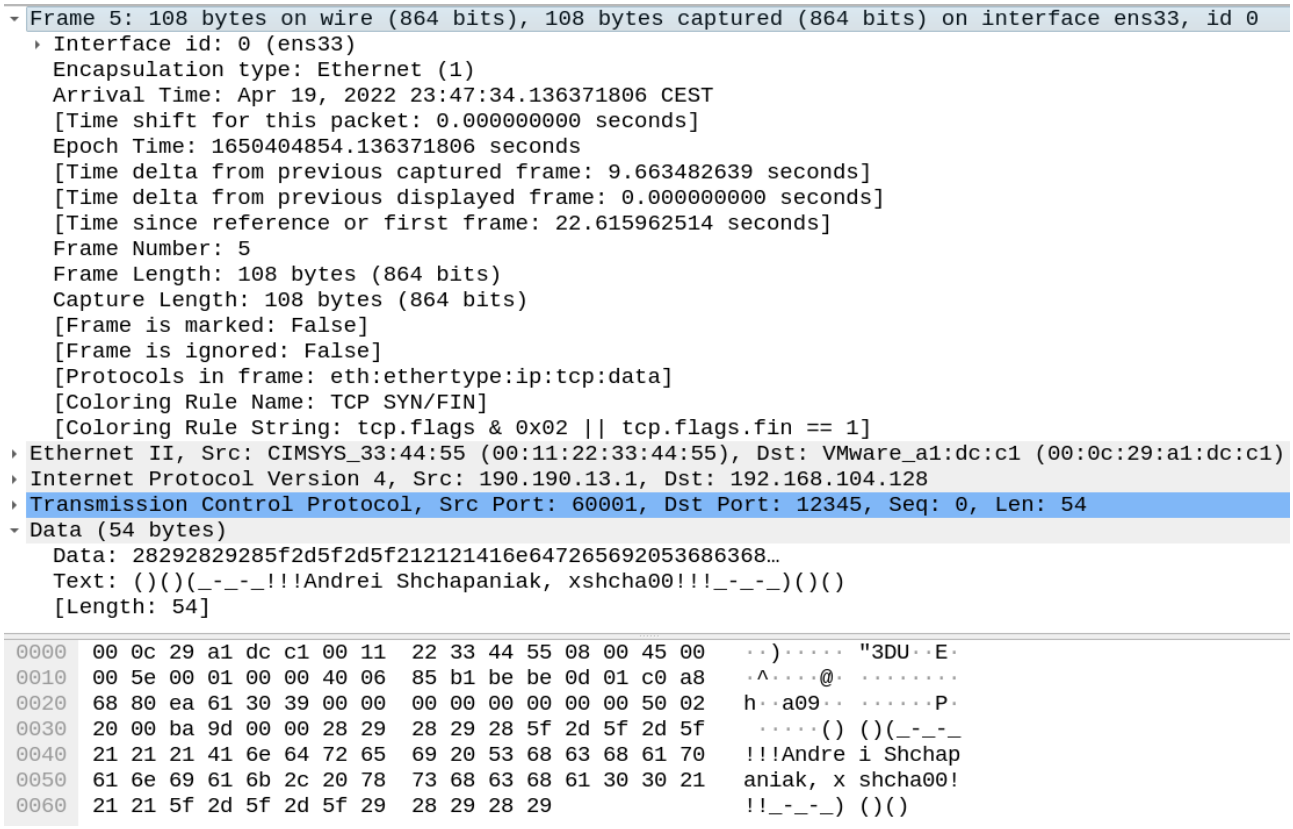

Figure 6: The output of the Wireshark tool with data in ASCII.

Bibliography

[1] *User Datagram Protocol* [RFC 768]. RFC Editor, August 1980.
Available at: https://www.rfc-editor.org/info/rfc768.

[2] *Internet Control Message Protocol* [RFC 792]. RFC Editor, September 1981.
Available at: https://www.rfc-editor.org/info/rfc792.

[3] *Transmission Control Protocol* [RFC 793]. RFC Editor, September 1981.
Available at: https://www.rfc-editor.org/info/rfc793.

[4] ATKINSON, R. a BHATTI, S. *Address Resolution Protocol (ARP) for the Identifier-Locator Network Protocol for IPv4 (ILNPv4)* [RFC 6747]. RFC Editor, November 2012.
Available at: https://www.rfc-editor.org/info/rfc6747.

[5] BONICA, R., GAN, D.-H., PIGNATARO, C. a TAPPAN, D. *Extended ICMP to Support Multi-Part Messages* [RFC 4884]. RFC Editor, April 2007.
Available at: https://www.rfc-editor.org/info/rfc4884.

[6] GUPTA, M. a CONTA, A. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification* [RFC 4443]. RFC Editor, March 2006.
Available at: https://www.rfc-editor.org/info/rfc4443.

[7] NEWMAN, C. a KLYNE, G. *Date and Time on the Internet: Timestamps* [RFC 3339]. RFC Editor, July 2002.
Available at: https://www.rfc-editor.org/info/rfc3339.