IPK PROJECT 2021/22 - ZETA

Skuratovich Aliaksandr
Brno University of Technologies
Brno, Božetěchova 1/2
`xskura01@vutbr.cz`

April 22, 2022

ABSTRACT

This document describes an implemented software for packet capturing and processing. The name of an application is `Zniffer`. `Zniffer` means the name of an assignment (ZETA) combined with the software type (packet sniffer).

`Zniffer` itself was implemented in c++, and tested on MacOS and Ubuntu. The program was tested by comparing results with wireshark[1].

The purpose of the whole project was to implement a packet sniffer and learn more about the structure of the packets and TCP/IP model[2].

[1]https://www.wireshark.org/
[2]https://en.wikipedia.org/wiki/Network_layer

Contents

1 Introduction

1.1 Packet

In telecommunications and computer networking, a network packet is a formatted unit of data carried by a packet-switched network. A packet consists of control information and user data; the latter is also known as the payload. Control information provides data for delivering the payload (e.g., source and destination network addresses, error detection codes, or sequencing information). Typically, control information is found in packet headers and trailers.

In the seven-layer OSI model of computer networking, packet strictly refers to a protocol data unit at layer 3, the network layer. (1)

1.2 Packet sniffing

A packet analyzer, also known as a packet sniffer, protocol analyzer, or network analyzer, is a computer program or computer hardware such as a packet capture appliance, that can intercept and log traffic that passes over a computer network or part of a network. Packet capture is the process of intercepting and logging traffic. As data streams flow across the network, the analyzer captures each packet and, if needed, decodes the packet's raw data, showing the values of various fields in the packet, and analyzes its content according to the appropriate RFC or other specifications (6).

1.3 Available tools

There are many software (both open-source and proprietary) for packet sniffing and network analyzing. Here, you can see some of them.

- Wireshark - open-source live network traffic analyzer.
- tcpdump - command line packet analyzer.
- tshark - terminal-based Wireshark.

Most of the sniffers are heavyweight and complicated with hundreds of different configurations and options.

Therefore, in this project, there was an attempt to implement a simpler one. The simplicity means having no extra options and a simple intuitive interface. The intuitiveness of the interface was reached by the absence of any interface at all.

1.4 RFC standards

A Request for Comments (RFC) is a publication in a series, from the principal technical development and standards-setting bodies for the Internet, most prominently the Internet Engineering Task Force (IETF).[3]

These standards will be mentioned later in the documentation. Therefore it is worth describing them here. Only the most important parts of the standards will be shown.

1.4.1 Internet Control Message Protocol

ICMP (3) creates and sends messages to the source IP address indicating that a gateway to the internet, such as a router, service, or host, cannot be reached for packet delivery.[4]

1.4.2 User Datagram Protocol

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. (2)

1.4.3 Transmission Control Protocol

TCP is connection-oriented, and a connection between client and server is established before data can be sent. The server must be listening (passive open) for connection requests from clients before a connection is established.

[3] https://en.wikipedia.org/wiki/Request_for_Comments

[4] https://www.techtarget.com/searchnetworking/definition/ICMP

Three-way handshake (active open), re-transmission, and error detection adds to reliability but lengthens latency (7).

1.4.4 Internet Protocol version 4

IPv4 is the fourth version of the Internet Protocol (IP). It is one of the core protocols of standards-based internetworking methods in the Internet and other packet-switched networks (5)

IPV4 protocol is a layer-3 protocol (OSI model). It takes data Segments from layer-4 (Transport layer) and divides it into packets.

1.4.5 Internet Protocol, Version 6

IPV6 protocol is a successor to a previous addressing infrastructure, the IPV4 protocol mentioned above. IPV6 is based on a 128-bit addressing system. Therefore the wider variety (2^{128} addresses) of unique IP addresses is available (4).

2 Implementation

2.1 Project structure

The structure consisted of two directories, and the whole project looks as follows. `zniffer`

```
|-- CMakeLists.txt
|-- src
|   |-- Argparser.cpp
|   |-- Argparser.h
|   |-- FilterCreator.cpp
|   |-- FilterCreator.h
|   |-- Sniffer.cpp
|   |-- Sniffer.h
|   |-- main.cpp
|   |-- structures.h
|-- tests
|   |-- PyPaGen.py
|   +-- requirements.txt
```

- The `src/` directory contains `.cpp` source files and `.h` header files.
- `CMakeLists.txt` is a configuration file to compile the project.
- `tests/` directory contains `PyPaGen.py`. `PyPaGen.py` is a script for automatic packet sending and `requirements.txt` with libraries necessary for run the script. For more information about the script see subsection 3.1.

2.2 Program pipeline

The program pipeline consists of four parts.

1. **Parse command-line arguments:**
   The first program block is implemented in `Argparse.cpp,h` files. Library `getopt` is used.

```cpp
namespace Argparser {

    struct program_arguments_t {
        const uint8_t flags;
        const uint32_t port;
        bool port_set;
        const int number_of_packets;
        const std::string interface;
    };

    struct program_arguments_t argparser(int, char **, int &);
}
```

   Function `argparser` parses command-line arguments and returns a structure `program_arguments_t`.

2. **Create a filter:**
   Then the fields of the `program_arguments_t` structure are passed to a `get_filter_string()` function, where they are processed in an efficient way. A `std::string` with a filter specification is returned. The function itself looks as follows:

```cpp
std::string get_filter_string(const uint8_t flags, const uint32_t port, const bool port_set) {
    const std::string opts[] = {
        "tcp", "udp", "arp", "icmp or icmp6"
    };
    // all flags options are set if no options provided.
    uint8_t i = flags == 0 ? 0b1111 : flags;
    int32_t j = 0;
    bool insert_or = false;
```

```
        bool enclose_in_braces;
        std::string opt_string;
        // bitwice tiktonik
        for (; i != 0b0000; i >>= 1, j++) {
            if ((i & 0b0001) == 1) {
                opt_string += insert_or ? std::string(" or ") + opts[j] : opts[j];
                // here, we know we'll need to insert "or" for the filter
                insert_or = true;
            }
        }
        enclose_in_braces = j > 1;
        if (enclose_in_braces)
            opt_string = std::string("(") + opt_string + std::string(")");
        if (port_set)
            opt_string += (std::string(" and port ") + std::to_string(port));
        return opt_string;
    }
```

3. **Packet sniffing**
   The first step is to create and configure a sniffer. Functions `pcap_open_live()` and `pcap_compile()` were used for these purposes.

   The second step is to start "listening" and capturing packets. For these purposes the function `pcap_lopp()` was used. Callback function accepted by a `pcap_loop()` as a parameter is called `sniff_packets()`.

4. **Packet disassembling**
   The function processes the individual packets, disassembles it, and prints all necessary information.

```
timestamp:  <RFC3339 formatted packet timestamp>
src MAC: <mac address>
dst MAC: <mac address>
frame length:  <whole number>B
src IP: <ip address>
dst IP: <ip address>
src port:  <port number>
dst port:  <port number>
<offset>   <8 bytes in hexa>  <8 bytes in hexa>    <printable ascii or .>
```

   The first step is to print a timestamp, MAC addresses, and the length of a captured packet. Then, accordingly to the type of the ethernet header, an appropriate function is called. List and the sequence of action for supported ethernet header types:

   (a) `ETHERTYPE_IP`,`ETHERTYPE_IPV6`: In these two cases, a function `handle_ip4,6_packet()` is called. The function prints source and destination IP addresses in an appropriate format. Then, source and destination ports are printed if IP protocol is `TCP` or `UDP`. In case of `ICMP` protocol, function returns.

   (b) `ETHERTYPE_ARP`: In the case of an ARP header, source and destination IP and MAC addresses are printed.

3 Testing

Wireshark tool was used to control the correctness of outputs provided by a `Zniffer` program. Also a script `PyPaGen.py` was implemented to generate packets.

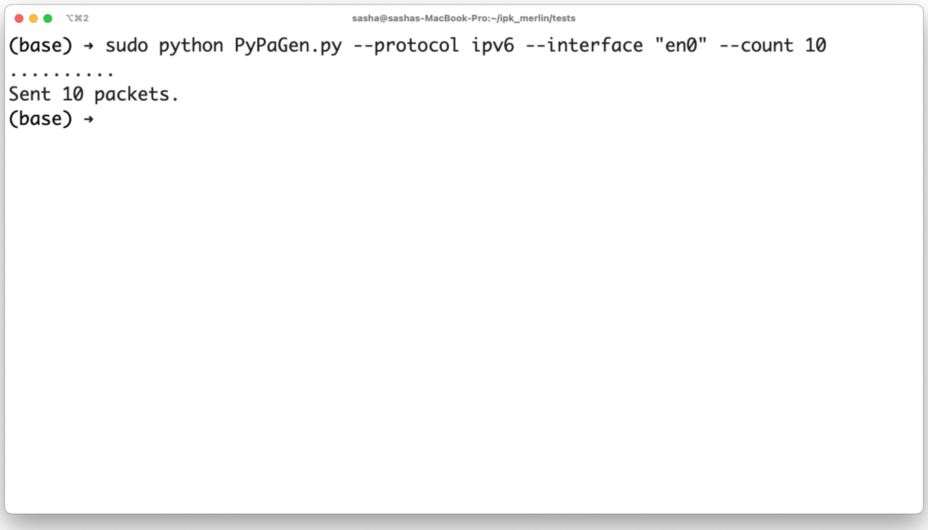
3.1 PyPaGen.py

The main purpose of the script, as its name implies, is to generate packets. The pipeline of a program is simple: specify command-line arguments and wait.

Packets are generated using `scapy` and `netifaces` libraries. You can see how the program creates and sends packets below. This script is helpful for debugging `Zniffer`. Also, it provides an opportunity to learn more about the process of packet creation in python.

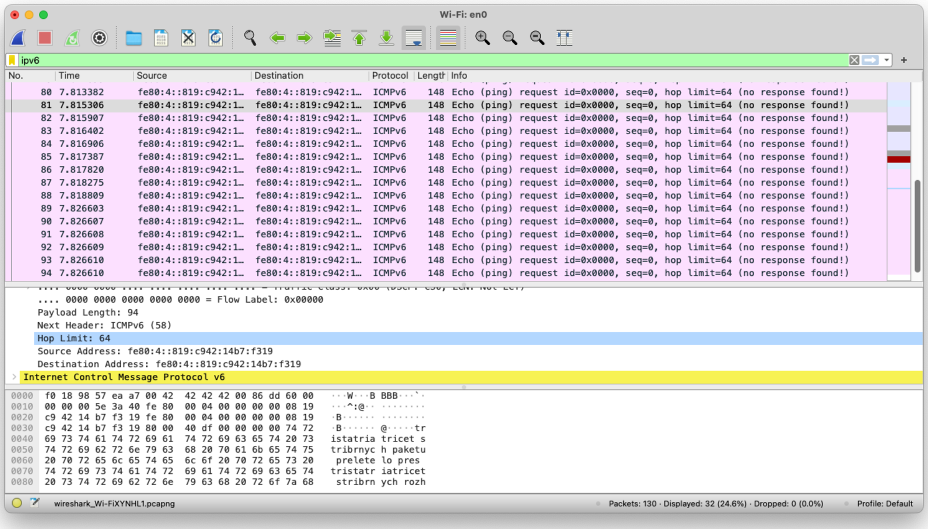
The whole testing pipeline looks as following:

1. Generate packets with `PyPaGen.py` script.
2. Compare Wireshark and `Zniffer` outputs.
3. Repeat the 2 steps above until it works well.



```
(base) → sudo python PyPaGen.py --protocol ipv6 --interface "en0" --count 10
..........
Sent 10 packets.
(base) →
```

Figure 1: Example of using `PyPaGen.py` script.



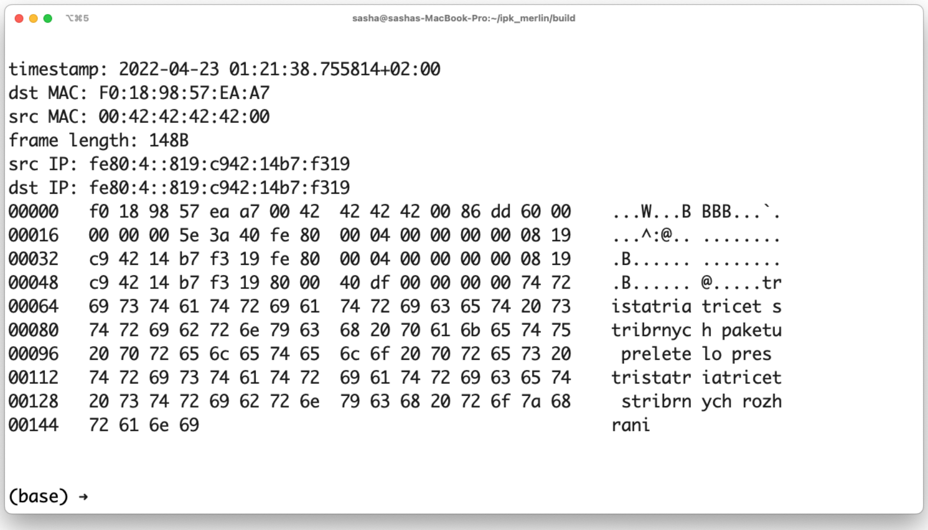
```
No.  Time      Source                Destination           Protocol Length Info
  80 7.813382  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  81 7.815306  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  82 7.815907  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  83 7.816402  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  84 7.816906  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  85 7.817387  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  86 7.817820  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  87 7.818275  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  88 7.818809  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  89 7.826603  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  90 7.826607  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  91 7.826608  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  92 7.826609  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  93 7.826610  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)
  94 7.826610  fe80:4::819:c942:1…   fe80:4::819:c942:1…   ICMPv6   148 Echo (ping) request id=0x0000, seq=0, hop limit=64 (no response found!)

    .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
    Payload Length: 94
    Next Header: ICMPv6 (58)
    Hop Limit: 64
    Source Address: fe80:4::819:c942:14b7:f319
    Destination Address: fe80:4::819:c942:14b7:f319
> Internet Control Message Protocol v6

0000  f0 18 98 57 ea a7 00 42  42 42 42 00 86 dd 60 00   ···W···B BBB···`·
0010  00 00 00 5e 3a 40 fe 80  00 04 00 00 00 00 08 19   ···^:@·· ········
0020  c9 42 14 b7 f3 19 fe 80  00 04 00 00 00 00 08 19   ·B······ ········
0030  c9 42 14 b7 f3 19 80 00  40 df 00 00 00 00 74 72   ·B······ @·····tr
0040  69 73 74 61 74 72 69 61  74 72 69 63 65 74 20 73   istatria tricet s
0050  74 72 69 62 72 6e 79 63  68 20 70 61 6b 65 74 75   tribrnyc h paketu
0060  20 70 72 65 6c 65 74 65  6c 6f 20 70 72 65 73 20    prelete lo pres
0070  74 72 69 73 74 61 74 72  69 61 74 72 69 63 65 74   tristatr iatricet
0080  20 73 74 72 69 62 72 6e  79 63 68 20 72 6f 7a 68    stribrn ych rozh

wireshark_Wi-FiXYNHL1.pcapng    Packets: 130 · Displayed: 32 (24.6%) · Dropped: 0 (0.0%)    Profile: Default
```

Figure 2: Packets generated by `PyPaGen.py` captured by Wireshark.



```
timestamp: 2022-04-23 01:21:38.755814+02:00
dst MAC: F0:18:98:57:EA:A7
src MAC: 00:42:42:42:42:00
frame length: 148B
src IP: fe80:4::819:c942:14b7:f319
dst IP: fe80:4::819:c942:14b7:f319
00000   f0 18 98 57 ea a7 00 42  42 42 42 00 86 dd 60 00   ...W...B BBB...`.
00016   00 00 00 5e 3a 40 fe 80  00 04 00 00 00 00 08 19   ...^:@.. ........
00032   c9 42 14 b7 f3 19 fe 80  00 04 00 00 00 00 08 19   .B...... ........
00048   c9 42 14 b7 f3 19 80 00  40 df 00 00 00 00 74 72   .B...... @.....tr
00064   69 73 74 61 74 72 69 61  74 72 69 63 65 74 20 73   istatria tricet s
00080   74 72 69 62 72 6e 79 63  68 20 70 61 6b 65 74 75   tribrnyc h paketu
00096   20 70 72 65 6c 65 74 65  6c 6f 20 70 72 65 73 20    prelete lo pres
00112   74 72 69 73 74 61 74 72  69 61 74 72 69 63 65 74   tristatr iatricet
00128   20 73 74 72 69 62 72 6e  79 63 68 20 72 6f 7a 68    stribrn ych rozh
00144   72 61 6e 69                                        rani


(base) →
```

Figure 3: Packets generated in `PyPaGen.py` captured by `Zniffer`.

References

[1]

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

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

[4] HINDEN, B. a DEERING, D. S. E. *Internet Protocol, Version 6 (IPv6) Specification* [RFC 2460]. RFC Editor, prosinec 1998.
Available at: https://www.rfc-editor.org/info/rfc2460.

[5] WIKIPEDIA CONTRIBUTORS. *IPv4 — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=IPv4&oldid=1081194704]. 2022. [Online; accessed 22-April-2022].

[6] WIKIPEDIA CONTRIBUTORS. *Packet analyzing*. 2022.
Available at: https://en.wikipedia.org/wiki/Packet_analyzer.

[7] WIKIPEDIA CONTRIBUTORS. *Transmission Control Protocol — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=1083491738]. 2022. [Online; accessed 22-April-2022].