ФЕДЕРАЛЬНОЕ АГЕНСТВО СВЯЗИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И
ИНФОРМАТИКИ» (ФГОБУ «СибГУТИ»)

# ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине

“Сетевое программное обеспечение”

на тему

РЕАЛИЗАЦИЯ СРЕДСТВ АНАЛИЗА СЕТЕВОГОГО ТРАФИКА

Вариант 32 Анализатор сетевого трафика (снифер)

Выполнил студент Марков В.А.
Ф.И.О.

Группы МГ-165

Работу принял д.т.н. К.В. Павский
подпись

Защищена Оценка

Новосибирск – 2016

СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Сниффер может анализировать только то, что проходит через его сетевую карту. Внутри одного сегмента сети Ethernet все пакеты рассылаются всем машинам, из-за этого возможно перехватывать чужую информацию. Использование коммутаторов (switch, switch-hub) и их грамотная конфигурация уже является защитой от прослушивания. Между сегментами информация передаётся через коммутаторы. Коммутация пакетов – форма передачи, при которой данные, разбитые на отдельные пакеты, могут пересылаться из исходного пункта в пункт назначения разными маршрутами. Так что если кто-то в другом сегменте посылает внутри него какие-либо пакеты, то в ваш сегмент коммутатор эти данные не отправит [1].

Перехват трафика может осуществляться:

- обычным «прослушиванием» сетевого интерфейса (метод эффективен при использовании в сегменте концентраторов (хабов) вместо коммутаторов (свитчей), в противном случае метод малоэффективен, поскольку на сниффер попадают лишь отдельные фреймы);
- подключением сниффера в разрыв канала;
- ответвлением (программным или аппаратным) трафика и направлением его копии на сниффер (Network tap);
- через анализ побочных электромагнитных излучений и восстановление таким образом прослушиваемого трафика;
- через атаку на канальном (MAC-spoofing) или сетевом уровне (IP-spoofing), приводящую к перенаправлению трафика жертвы или всего трафика сегмента на сниффер с последующим возвращением трафика в надлежащий адрес.

1. Принцип работы пакетных сниферов

Сниффер – программа, которая работает на канальном уровне и скрытым образом перехватывает весь трафик. Поскольку снифферы работают на канальном уровне модели OSI, они не должны играть по правилам протоколов более высокого уровня. Снифферы обходят механизмы фильтрации (адреса, порты и т.д.), которые драйверы Ethernet и стек TCP/IP используют для интерпретации данных. Пакетные снифферы захватывают из «провода» все, что по нему приходит. Снифферы могут сохранять кадры в двоичном формате и позже расшифровывать их, чтобы раскрыть информацию более высокого уровня, спрятанную внутри (рисунок 1).

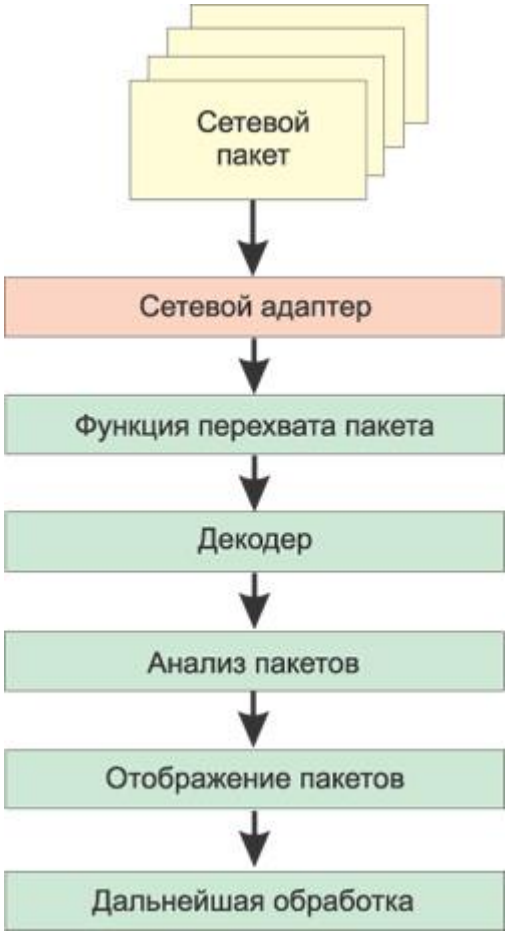


Рисунок 1 – Схема работы сниффера

Для того чтобы сниффер мог перехватывать все пакеты, проходящие через сетевой адаптер, драйвер сетевого адаптера должен поддерживать режим функционирования promiscuous mode (беспорядочный режим). Именно в этом

режиме работы сетевого адаптера сниффер способен перехватывать все пакеты. Данный режим работы сетевого адаптера автоматически активизируется при запуске сниффера или устанавливается вручную соответствующими настройками сниффера.

Весь перехваченный трафик передается декодеру пакетов, который идентифицирует и расщепляет пакеты по соответствующим уровням иерархии. В зависимости от возможностей конкретного сниффера представленная информация о пакетах может впоследствии дополнительно анализироваться и отфильтровываться [2].

Анализаторы трафика используются в различных целях, таких как:

- анализ протоколов;
- мониторинга сети;
- оценка безопасности сети.

Например, Wireshark является самым популярным анализатором трафика, доступен для всех платформ.

В курсовом проекте рассмотрим два подхода по написанию своего собственного анализатора сетевого трафика для операционной системы GNU/Linux:

- на основе библиотеки libpcap
- на основые сырых сокетов (raw socket)

## 1.1. Анализатор трафика на основе libpcap

Библиотека Pcap (Packet Capture) позволяет создавать программы анализа сетевых данных, поступающих насетевую карту компьютера. Примером программного обеспечения, использующего библиотеку Pcap, служит программа Wireshark. Разнообразные программы мониторинга и тестирования сети, снифферы используют эту библиотеку. Она предназначена для использования совместно с языками C/C++, а для работы с библиотекой на других языках, таких как Java, .NET, используют обёртки. Для Unix-подобных систем это библиотека libpcap, а для Microsoft Windows – WinPcap. Программное обеспечение сетевого

мониторинга может использовать libpcap или WinPcap, чтобы захватить пакеты, путешествующие по сети, и (в более новых версиях) для передачи пакетов в сети. Libpcap и WinPcap также поддерживают сохранение захваченных пакетов в файл и чтение файлов, содержащих сохранённые пакеты. Программы, написанные на основе libpcap или WinPcap, могут захватить сетевой трафик, анализировать его. Файл захваченного трафика сохраняется в формате, понятном для приложений, использующих Pcap [3].

```
#include <pcap.h>

pcap_t *handle = NULL;

//Open the device for sniffing
handle = pcap_open_live("eth0", BUFSIZ, 0, -1, errbuf);
if (handle == NULL) {
    // print error
    exit(1);
}

//Put the device in sniff loop (100 packets)
pcap_loop(handle, 100, callback, NULL);

// Close the device
pcap_close(handle);
```

## 1.2. Анализатор трафика на основе сырых сокетов

Linux позволяет использовать сырые сокеты не только для отправки, но и для получения данных.

```
/*
 * Example how to do with raw socket:
 * 1. Sniff both incoming and outgoing traffic.
 * 2. Sniff ALL ETHERNET FRAMES, which includes all kinds of
 * IP packets and even more if there are any.
 * 3. Provides the Ethernet headers too, which contain the mac addresses.
 */
 int sock_raw = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
 /*
  * Optional:
  * Its important to provide the correct interface name
  * to setsockopt, eth0 in this case and in most cases.
  *
  * setsockopt(sock_raw, SOL_SOCKET, SO_BINDTODEVICE,
  *            "eth0", strlen("eth0")+1);
  */

if (sock_raw < 0) {
    //Print the error
    return 1;
  }
```

```
while (1) {
  //Receive a packet
  if (recvfrom(sock_raw, buffer) < 0) {
      //Print the error
  }
  callback(args); //process the packet
}
close(sock_raw);
```

## 2. Структура сетевых протоколов в GNU/Linux

Раздел сетевых протоколов определяет отдельные доступные сетевые протоколы (такие как TCP, UDP и так далее). Они инициализируются в начале дня в функции inet_init в linux/net/ipv4/af_inet.c (так как TCP и UDP относятся к семейству протоколовinet). Функция inet_init регистрирует каждый из встроенных протоколов, использующих функцию proto_register. Эта функция определена в linux/net/core/sock.c, и кроме добавления протокола в список действующих, если требуется, может выделять один или более slab-кэшей.

Можно увидеть, как отдельные протоколы идентифицируют сами себя посредством структуры proto в файлах tcp_ipv4.c, udp.c и raw.c, в linux/net/ipv4/. Каждая из этих структур протоколов отображается в виде типа и протокола в inetsw_array, который приписывает встроенные протоколы их операциям. Структура inetsw_array и его связи показаны на рисунке 2. Каждый из протоколов в этом массиве инициализируется в начале дня в inetsw вызовом inet_register_protosw из inet_init. Функция inet_init также инициализирует различные модули inet, такие как ARP, ICMP, IP-модули и TCP и UDP-модули.

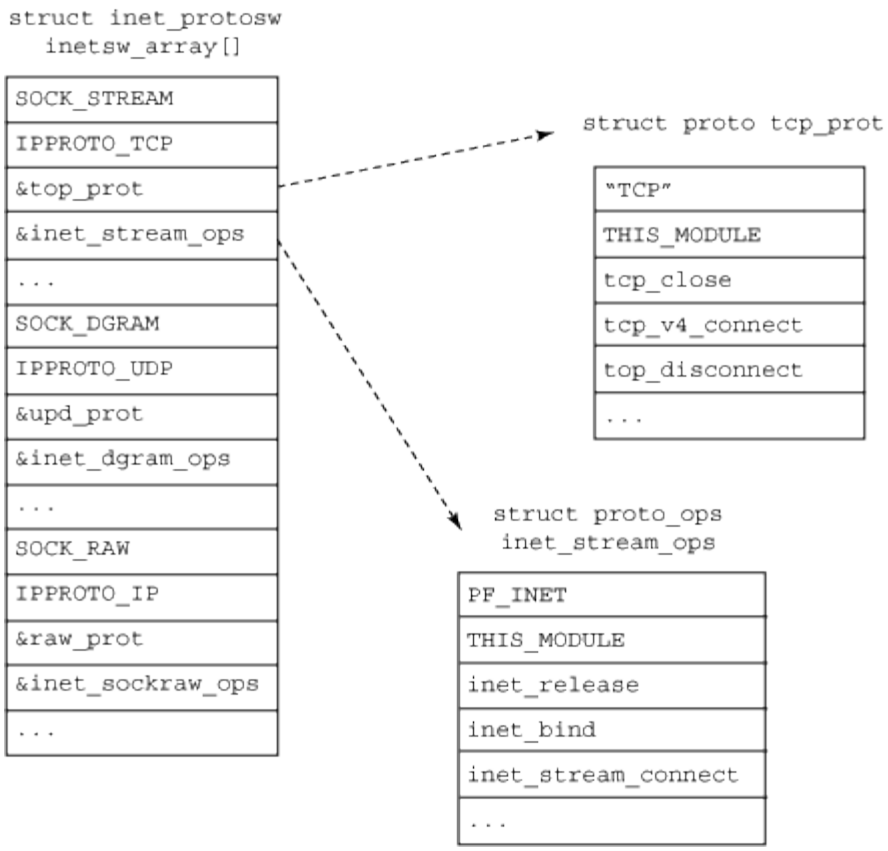


Рисунок 2 – Структура стека протоколов

Как можно заметить, рисунок 3, несколько структур sk_buff для данного соединения могут быть связаны вместе. Каждая из них идентифицирует структуру устройства (net_device), которому пакет посылается или от которого получен. Так как каждый пакет представлен bsk_buff, заголовки пакетов удобно определены набором указателей (th, iph и mac для Управления доступом к среде (заголовок Media Access Control или MAC). Поскольку структуры sk_buff являются центральными в организации данных сокета, для управления ими был создан ряд функций поддержки. Существуют функции для создания, разрушения, клонирования и управления очередностью sk_buff.

Буферы сокетов разработаны таким образом, чтобы связываться друг с другом для данного сокета и включать большой объем информации, в том числе ссылки на заголовки протоколов, временные метки (когда пакет был отправлен или получен) и соответствующее устройство.

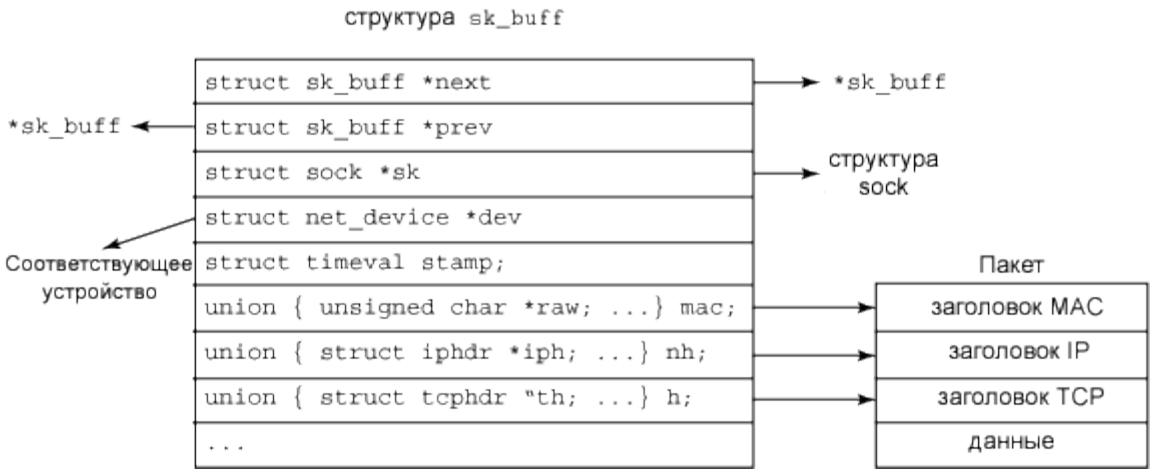


Рисунок 3 – Структура буфера сокета

# ЗАКЛЮЧЕНИЕ

В результате выполнения работы реализован анализатор трафика на базе библиотеки libpcap, и сырых сокетов.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. ru.wikipedia.org/wiki/Анализатор_трафика
2. http://compress.ru/article.aspx?id=16244
3. ru.wikipedia.org/wiki/Pcap
4. Стивенс У.Р., Феннер Б., Рудофф Э. M.UNIX: разработка сетевых приложений. - 3-е изд. - СПб. : ПИТЕР, 2007. - 1038с.
5. Фейт С. TCP/IP: Архитектура, протоколы, реализация (включая IP версии 6 и IP Security). – М.: Лори, 2000. – 424 с.

# ПРИЛОЖЕНИЕ

# ИСХОДНЫЙ ТЕКСТ ПРОГРАММЫ

```
#include "header.h"

uint16_t checksum(uint8_t *buf, uint16_t len)
{
        uint32_t sum = 0;

        // build the sum of 16bit words
        while(len >1) {
            sum += 0xFFFF & (*buf<<8|*(buf+1));
            buf+=2;
            len-=2;
        }
        // if there is a byte left then add it (padded with zero)
        if (len){
            sum += (0xFF & *buf)<<8;
        }
        // now calculate the sum over the bytes in the sum
        // until the result is only 16bit long
        while (sum>>16) {
            sum = (sum & 0xFFFF)+(sum >> 16);
        }
        // build 1's complement:
        return( (uint16_t) sum ^ 0xFFFF);
}


unsigned short csum(unsigned short* vdata, int length)
{
    // Cast the data pointer to one that can be indexed.
    char* data=(char*)vdata;

    // Initialise the accumulator.
    uint32_t acc=0xffff;

    // Handle complete 16-bit blocks.
    size_t i=0;
    for (i = 0; i+1 < length; i += 2) {
        uint16_t word;
        memcpy(&word,data+i,2);
        acc+=ntohs(word);
        if (acc>0xffff) {
            acc-=0xffff;
        }
    }

    // Handle any partial block at the end of the data.
    if (length&1) {
        uint16_t word=0;
        memcpy(&word,data+length-1,1);
        acc+=ntohs(word);
        if (acc>0xffff) {
            acc-=0xffff;
        }
    }
```

```
    // Return the checksum in network byte order.
    return ~acc;
}

// set tcp checksum: given IP header and tcp segment
void compute_tcp_checksum(struct iphdr *pIph, unsigned short *ipPayload)
{
    register unsigned long sum = 0;

    unsigned short tcpLen = ntohs(pIph->tot_len) - (pIph->ihl<<2);

    struct tcphdr *tcphdrp = (struct tcphdr*)(ipPayload);

    //add the pseudo header

    //the source ip
    sum += (pIph->saddr>>16)&0xFFFF;
    sum += (pIph->saddr)&0xFFFF;

    //the dest ip
    sum += (pIph->daddr>>16)&0xFFFF;
    sum += (pIph->daddr)&0xFFFF;

    //protocol and reserved: 6
    sum += htons(IPPROTO_TCP);

    //the length
    sum += htons(tcpLen);

    //add the IP payload
    //initialize checksum to 0
    tcphdrp->check = 0;
    while (tcpLen > 1) {
        sum += htons(*ipPayload++);
        tcpLen -= 2;
    }
    //if any bytes left, pad the bytes and add
    if(tcpLen > 0) {
        //printf("++++++++++padding, %d\n", tcpLen);
        sum += ((*ipPayload)&htons(0xFF00));
    }
      //Fold 32-bit sum to 16 bits: add carrier to result
      while (sum>>16) {
          sum = (sum & 0xffff) + (sum >> 16);
      }
      sum = ~sum;
    //set computation result
    tcphdrp->check = (unsigned short)sum;
    printf("--->%d", htons(tcphdrp->check));
}

unsigned short csum1(unsigned short* vdata, int length, struct iphdr *pIph)
{
    // Cast the data pointer to one that can be indexed.
    char* data=(char*)vdata;

    int acc = 0;

    unsigned short tcpLen = ntohs(pIph->tot_len) - (pIph->ihl<<2);
```

```
    //add the pseudo header

    //the source ip
    acc += pIph->saddr;
    acc += pIph->saddr;

    //the dest ip
    acc += pIph->daddr;
    acc += pIph->daddr;

    //protocol and reserved: 6
    acc += htons(IPPROTO_TCP);

    //the length
    acc += htons(tcpLen);

    // Handle complete 16-bit blocks.
    size_t i = 0;
    for (i = 0; i+1 < length; i += 2) {
        uint16_t word;
        memcpy(&word,data+i,2);
        acc+=ntohs(word);
        if (acc>0xffff) {
            acc-=0xffff;
        }
    }

    // Handle any partial block at the end of the data.
    if (length&1) {
        uint16_t word=0;
        memcpy(&word,data+length-1,1);
        acc+=ntohs(word);
        if (acc>0xffff) {
            acc-=0xffff;
        }
    }

    // Return the checksum in network byte order.
    return ~acc;
}

#ifndef HEADER_H
#define HEADER_H

#include <pcap.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>

#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/icmp.h>
#include <linux/if_ether.h>

#include <arpa/inet.h>
#include <netinet/in.h>
```

```
#include <linux/if_arp.h>

#define COLOR_OFF "\033[0m"

#define COLOR_RED "\033[0;31m"
#define COLOR_CYN "\033[0;36m"
#define COLOR_BLU "\033[0;34m"
#define COLOR_GRN "\033[0;32m"
#define COLOR_RED "\033[0;31m"
#define COLOR_BLC "\033[0;30m"
#define COLOR_WHT "\033[0;37m"
#define COLOR_YEL "\033[0;33m"
#define COLOR_MAG "\033[0;35m"

struct pseudo_header {
    unsigned int   src_ip;
    unsigned int   dest_ip;
    unsigned char  zeroes;
    unsigned char  protocol;
    unsigned short len;
} __attribute__ ((packed));


void print_data(const u_char *data , int size);

void print_tcp_packet(const u_char *data, int size, char *arg1, char *arg2);
void print_udp_packet(const u_char *data, int size, char *arg1, char *arg2);
void print_icmp_packet(const u_char *data, int size, char *arg1, char *arg2);

void print_other_packet(const u_char *data, int size, char *arg1, char *arg2);

void print_ethernet_header(const u_char *data);
void print_ip_header(const u_char *data, int size);
void print_tcp_header(const u_char *data, int size);
void print_arp_header(const u_char *data, int size);

unsigned short csum(unsigned short *data, int len);
uint16_t checksum(uint8_t *buf, uint16_t len);

void compute_tcp_checksum(struct iphdr *pIph, unsigned short *ipPayload);
unsigned short csum1(unsigned short* vdata, int length, struct iphdr *pIph);


#endif


/*
*   Simple packet sniffer using libpcap library by Markov V.A.
*
*   make clean && make
*   sudo ./main [flags]
*
*   flags:
*   [-i] - show icmp
*   [-u] - show udp
*   [-t] - show tcp
*   [-a] - show arp
*
*   [-h] - show header (use with -i/-t/-u/-a)
*   [-d] - show data (use with -i/-t/-u/-a)
*   [-c] - count packets
```

```
*   [-s] - show statistic
*/

#include "header.h"

/*
*   flag_mode[0] - print header
*   flag_mode[1] - count packets
*   flag_mode[2] - print data
*   flag_mode[3] - icmp
*   flag_mode[4] - tcp
*   flag_mode[5] - udp
*   flag_mode[6] - arp
*   flag_mode[7] - statisitc
*/
char *flag_mode[7] = {"q","q","q","q","q","q","q"};

/*
*   packet_cnt[0] - tcp
*   packet_cnt[1] - udp
*   packet_cnt[2] - icmp
*   packet_cnt[3] - igmp
*   packet_cnt[4] - other
*   packet_cnt[5] - total
*   packet_cnt[6] - arp
*/
int packet_cnt[7] = { 0 };

// Process the sniffed packet (callback method)
void handler(u_char *user, const struct pcap_pkthdr *pkthdr, const u_char
*buffer)
{
    int size = pkthdr->len;

    printf("\nTime of capture          : %s", ctime(&pkthdr->ts.tv_sec));
    printf("Capture lenght of packet : %d bytes\n", pkthdr->caplen);
    printf("Full lenght of packet    : %d bytes\n", pkthdr->len);

    //Get the IP Header part of this packet , excluding the ethernet header
    struct iphdr *iph = (struct iphdr*)(buffer + sizeof(struct ethhdr));

    ++packet_cnt[5]; // increment total packet

    //Check the Protocol and do accordingly...
    switch (iph->protocol)
    {
        /* ICMP Protocol */
        case 1:
        {
            ++packet_cnt[2]; // increment icmp pakset
            if (flag_mode[3] == "i") {
                print_icmp_packet(buffer, size, flag_mode[2], flag_mode[1]);
            }
            //exit(1);
            break;
        }
        /* IGMP Protocol */
        case 2:
        {
            ++packet_cnt[3]; // increment igmp packet
            break;
```

```
        }
        /* IP Protocol */
        case 4:
        {
            break;
        }
        /* TCP Protocol */
        case 6:
        {
            ++packet_cnt[0]; // increment tcp packet
            if (flag_mode[4] == "t") {
                print_tcp_packet(buffer, size, flag_mode[2], flag_mode[1]);
            }
            break;
        }
        /* UDP Protocol */
        case 17:
        {
            ++packet_cnt[1]; // increment udp packet
            if (flag_mode[5] == "u") {
                print_udp_packet(buffer, size, flag_mode[2], flag_mode[1]);
            }
            break;
        }
        /* Some Other Protocol like ARP etc. */
        default:
        {
            if (flag_mode[6] == "a") {
                print_other_packet(buffer, size, flag_mode[2], flag_mode[1]);
            }
            ++packet_cnt[4]; // increment other/arp packet
            //exit(1);
            break;
        }
    }
}

int main(int argc, char *argv[])
{
    int rc = 0;
    char errbuf[PCAP_ERRBUF_SIZE] = { 0 };
    pcap_t *handle = NULL;

    /* Default packet */
    int count_packet = 100;

    /* If no args */
    if (argc < 2) {
        fprintf(stderr, COLOR_RED);
        fprintf(stderr, "error: no argc\n");
        fprintf(stderr, "usage: %s\n", argv[0]);
        fprintf(stderr, "-c (count packets)\n");
        fprintf(stderr, "-d (print data)\n");
        fprintf(stderr, "-i (print icmp)\n");
        fprintf(stderr, "-a (print arp)\n");
        fprintf(stderr, "-t (print tcp)\n");
        fprintf(stderr, "-u (print udp)\n");
        fprintf(stderr, "-h (print header)\n");
        fprintf(stderr, "-s (show statistic)\n");
        fprintf(stderr, COLOR_OFF);
        exit(1);
```

```
}

// Set up the working mode, set up flags
while ((rc = getopt(argc, argv, "c:dhtiuas")) != -1) {
    switch (rc)
    {
        case 'c':
            flag_mode[0] = "c"; // count
            count_packet = atoi(optarg);
            printf("[show total packet %d ] ", count_packet);
            break;

        case 'd':
            flag_mode[1] = "d"; // data
            printf("[show data] ");
            break;

        case 'h':
            flag_mode[2] = "h"; // header
            printf("[show header] ");
            break;

        case 't':
            flag_mode[4] = "t"; // tcp
            printf("[show tcp] ");
            break;

        case 'i':
            flag_mode[3] = "i"; // icmp
            printf("[show icmp] ");
            break;

        case 'u':
            flag_mode[5] = "u"; // udp
            printf("[show udp] ");
            break;

        case 'a':
            flag_mode[6] = "a"; // arp
            printf("[show arp] ");
            break;

        case 's':
            flag_mode[7] = "s"; // arp
            printf("[show statistic] ");
            break;

        case '?':
            fprintf(stderr, COLOR_BLU);
            fprintf(stderr, "help\n");
            fprintf(stderr, "usage: %s\n", argv[0]);
            fprintf(stderr, "-c (count packets)\n");
            fprintf(stderr, "-s (show statistic)\n");
            fprintf(stderr, "-i (print icmp)\n");
            fprintf(stderr, "-t (print tcp)\n");
            fprintf(stderr, "-a (print arp)\n");
            fprintf(stderr, "-u (print udp)\n");
            fprintf(stderr, "-d (print data)\n");
            fprintf(stderr, "-h (print header)\n");
            fprintf(stderr, COLOR_OFF);
            exit(1);
```

```
                break;
            }
        }

        /*
         * Example how to do it without pcap library:
         * 1. Sniff both incoming and outgoing traffic.
         * 2. Sniff ALL ETHERNET FRAMES, which includes all kinds of
         * IP packets and even more if there are any.
         * 3. Provides the Ethernet headers too, which contain the mac addresses.
         *
         * int sock_raw = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
         *
         * Optional:
         * Its important to provide the correct interface name
         * to setsockopt, eth0 in this case and in most cases.
         *
         * setsockopt(sock_raw, SOL_SOCKET, SO_BINDTODEVICE,
         *            "eth0", strlen("eth0")+1);
         *
         * if (sock_raw < 0) {
         *    //Print the error
         *    return 1;
         * }
         *
         * while (1) {
         *   //Receive a packet
         *   if (recvfrom(sock_raw, buffer) < 0) {
         *       //Print the error
         *   }
         *   callback(args); //process the packet
         * }
         * close(sock_raw);
         */

        //Open the device for sniffing
        //handle = pcap_open_live("eth0", BUFSIZ, 0, -1, errbuf);
        handle = pcap_open_live("wlp2s0", BUFSIZ, 0, -1, errbuf);
        if (handle == NULL) {
            fprintf(stderr, "error: Couldn't open device eth0: %s\n", errbuf);
            exit(1);
        }

        //Put the device in sniff loop (100 packets)
        pcap_loop(handle, count_packet, handler, NULL);

        // Close the device
        pcap_close(handle);

        // Display statistic
        if (flag_mode[7] == "s") {
            char buf_stat[256] = { 0 };

            printf(COLOR_MAG);
            printf("+-----------------------------------------------------------------
----------+");
            printf("\n|-Capturing packets:
|\n");

            int len = sprintf(buf_stat, "|-TCP:%d, UDP:%d, ICMP:%d, IGMP:%d, Others:
%d, Total: %d",
```

```
packet_cnt[0],packet_cnt[1],packet_cnt[2],packet_cnt[3],packet_cnt[4],packet_cnt
[5]);
        printf("%s", buf_stat);

        for (int i = len; i < 74; i++) {
            printf(" ");
        }

        printf("|");
        printf("\n+-----------------------------------------------------------------
------------+\n");
        printf(COLOR_OFF);
    }

    return(0);
}

#include "header.h"

struct arp_head {
    u_char arp_sha[ETH_ALEN]; // sender hardware address
    u_char arp_sip[4];        // sender ip address
    u_char arp_tha[ETH_ALEN]; // target hardware address
    u_char arp_tip[4];        // target ip address
} __attribute__ ((packed));

void print_arp_header(const u_char *data, int size)
{
    int i;
    int header_size;

    struct arphdr *arph;
    struct arp_head *a_hd;

    unsigned short arphdrlen;

    arph = (struct arphdr*)(data + sizeof(struct ethhdr));

    arphdrlen = sizeof(struct arphdr);

    header_size = sizeof(struct ethhdr) + arphdrlen;

    a_hd = (struct arp_head*)(data + header_size);

    printf("\nARP HEADER\n");
    printf("   |-Hardware address    :Ethernet 0x%.2X\n", ntohs(arph->ar_hrd));
    printf("   |-Protocol address    :IP 0x%.2x\n", ntohs(arph->ar_pro));

    printf("   |-Hardware size       :%X\n", arph->ar_hln);
    printf("   |-Protocol size       :%X\n", arph->ar_pln);
    printf("   |-Command             :%X\n", ntohs(arph->ar_op));

    printf("\nADDITIONAL ARP HEADER\n");

    printf("   |-MAC sender ");
    for (i = 0; i < 6; i++) {
        printf(":%.2X", a_hd->arp_sha[i]);
    }

    printf("\n   |-IP sender  :");
```

```
    for (i = 0; i < 4; i++) {
        if (i == 3) {
            printf("%d", a_hd->arp_sip[i]);
        } else {
            printf("%d.", a_hd->arp_sip[i]);
        }
    }
    printf("\n   |-MAC target ");


    for (i = 0; i < 6; i++) {
        printf(":%.2X", a_hd->arp_tha[i]);
    }

    printf("\n   |-IP target  :");
    for (i = 0; i < 4; i++) {
        if (i == 3) {
           printf("%d", a_hd->arp_tip[i]);
        } else {
           printf("%d.", a_hd->arp_tip[i]);
        }
    }
}
#include "header.h"

void print_data(const u_char *data , int Size)
{
    int i, j;

    for (i = 0; i < Size; i++) {
        if (i != 0 && i%16 == 0) {  //if one line of hex printing is complete...
            printf("         ");
            for (j = i - 16; j < i; j++) {
                if (data[j] >= 32 && data[j] <= 128) {
                    printf("%c", (unsigned char)data[j]); //if its a number or
alphabet
                } else {
                    printf("."); //otherwise print a dot
                }
            }
            printf("\n");
        }

        if (i%16 == 0)
            printf("   ");

        printf(" %02X", (unsigned int)data[i]);
        if (i == Size - 1) {  //print the last spaces
            for (j = 0; j < 15 - i%16; j++)
                printf("   "); //extra spaces
            printf("         ");
            for (j = i - i%16; j <= i; j++) {
                if (data[j] >= 32 && data[j] <= 128)
                    printf("%c", (unsigned char)data[j]);
                else
                    printf(".");
            }
            printf("\n");
        }
    }
}
```