

ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

ФАКУЛЬТЕТ ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра вычислительных систем

ОТЧЁТ
по курсовому проекту
по дисциплине «Сетевое программное обеспечение»
на тему
«Разработка сетевого приложения»

Выполнил:
студент группы МГ-165
Терешков Р. В.

Проверил:
доцент д.т.н.
Павский К. В.

Новосибирск – 2016

Содержание

Теоретические сведения	3
Принцип работы пакетных снифферов	3
Основы работы с libpcap	5
Реализация приложения	8
Заключение	10
Список литературы	11
Листинг программы	12

Теоретические сведения

Термин "пакет" впервые вводится на сетевом уровне. Основные протоколы этого уровня: IP (Internet Protocol - межсетевой протокол), ICMP (Internet Control Message Protocol - протокол межсетевых управляющих сообщений), IGMP (Internet Group Management Protocol - протокол управления группами Интернета) и IPsec (набор протоколов для обеспечения защиты данных, передаваемых по протоколу IP). Протоколы транспортного уровня включают в себя TCP (Transmission Control Protocol - протокол управления передачей), ориентированный на создание постоянного соединения; UDP (User Datagram Protocol - протокол пользовательских дейтаграмм), не требующий постоянного соединения; SCTP (Stream Control Transmission Protocol - протокол передачи с управлением потоком), сочетающий в себе свойства двух приведенных выше протоколов. Прикладной уровень содержит множество часто используемых протоколов, таких как: HTTP, FTP, IMAP, SMTP и множество других. [1]

Под захватом пакетов понимается сбор данных, передаваемых по сети. В любой момент, когда сетевая карта принимает Ethernet-кадр, она проверяет целевой MAC-адрес пакета на соответствие своему. В случае совпадения адресов генерируется запрос прерывания. Это прерывание впоследствии обрабатывается драйвером сетевой карты; он копирует данные из буфера сетевой карты в буфер в адресном пространстве ядра, затем проверяет поле в заголовке пакета, отвечающее за тип и передает пакет обработчику необходимого протокола в зависимости от содержания поля. Данные преодолевают стек обработчиков сетевых уровней и достигают прикладного уровня, на котором обрабатываются с помощью пользовательского приложения.

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.

Принцип работы пакетных снифферов

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.

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. [3]

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.

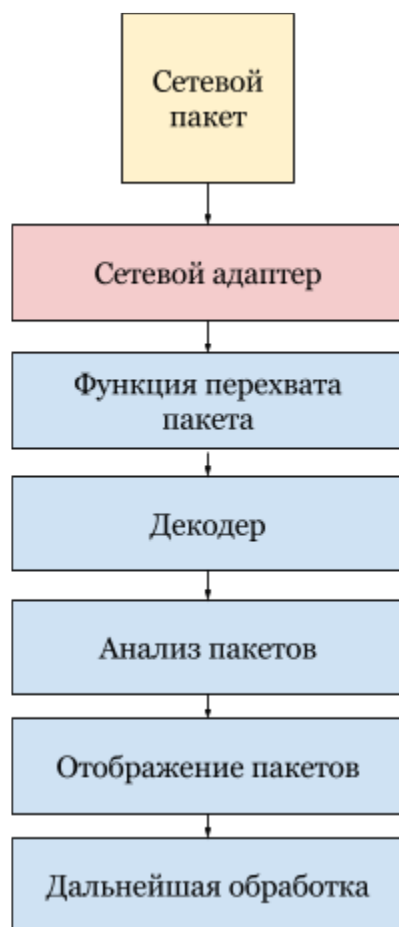


Рисунок 1 – Схема работы анализатора трафика.

Анализаторы трафика используются в различных целях, таких как:

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

Одним из самых популярных анализаторов трафика, доступных для всех платформ, является Wireshark.

В курсовом проекте рассмотрим подход по написанию своего собственного анализатора сетевого трафика для операционной системы GNU/Linux на основе библиотеки libpcap.

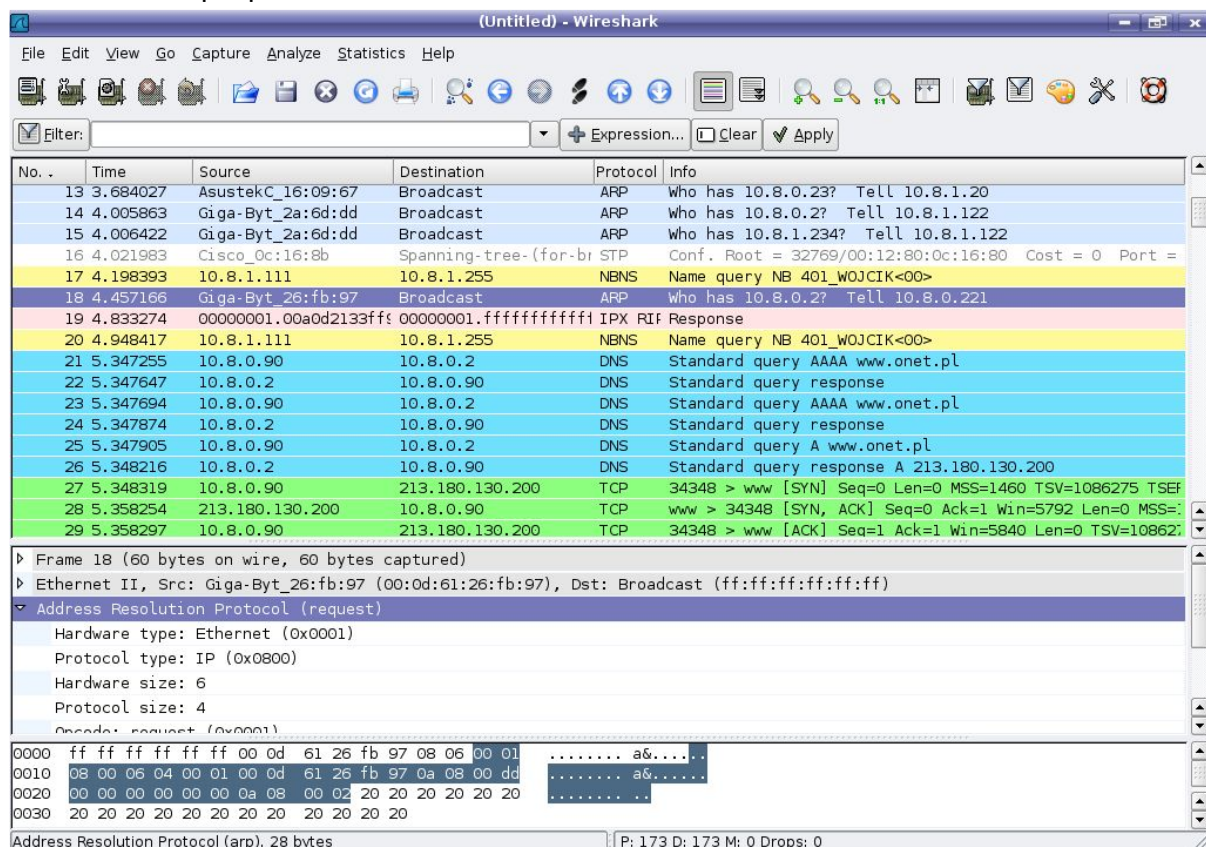


Рисунок 2 – Программа Wireshark.

Основы работы с libpcap

Библиотека libpcap является платформо независимой библиотекой с открытым исходным кодом (версия для Windows носит название winpcap). Известные снифферы tcpdump и Wireshark используют эту библиотеку для работы.

Для разработки нашей программы, нам необходим сетевой интерфейс, на котором будет производиться захват пакетов. Мы можем назначить это устройство самостоятельно или воспользоваться функцией, предоставляемой libpcap:

```
char *pcap_lookupdev(char *errbuf)
```

Эта функция возвращает указатель на строку, содержащую название первого сетевого интерфейса, пригодного для захвата пакетов; в случае ошибки возвращается NULL

(это справедливо и для других функций `libpcap`). Аргумент `errbuf` предназначен для пользовательского буфера, в который будет помещено сообщение об ошибке в случае возникновения - это очень удобно при отладке программ. Размер этого буфера должен быть не меньше `PCAP_ERRBUF_SIZE` (на данный момент 256) байт.

Для вывода всех доступных для sniffинга устройств можно использовать функцию

```
int pcap_findalldevs(pcap_if_t **alldevsp, char* errbuf)
```

Работа с устройством

Далее мы открываем выбранное сетевое устройство, используя функцию

```
pcap_t *pcap_open_live(const char *device, int snaplen,  
    int promisc, int to_ms, char *errbuf) .
```

Она возвращает идентификатор устройства, представленный в виде переменной типа `pcap_t`, которая может использоваться в других функциях `libpcap`. Первый аргумент - сетевой интерфейс, с которым мы хотим работать, второй - максимальный размер буфера для захвата данных в байтах.

Установка минимального значения второго параметра полезна в том случае, когда необходимо производить захват только заголовков пакетов. Размер Ethernet-кадра равен 1518 байтам. Значения 65535 будет достаточно для захвата любого пакета из любой сети. Аргумент `promisc` устанавливает, будет ли устройство работать в `promiscuous`-режиме или нет. (В `promiscuous`-режиме сетевая карта будет генерировать прерывания для всех данных, которые она получает, а не только для тех, которые подходят по значению MAC-адреса).

Аргумент `to_ms` сообщает ядру время ожидания в миллисекундах перед копированием информации из пространства ядра в пространство пользователя. Передача значения 0 приведет к тому, что операция чтения данных будет ожидать до тех пор, пока не будет собрано достаточное количество пакетов.

Захват данных

Теперь нам необходимо начать захват пакетов. Используем функцию

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback,  
    u_char *user)
```

которая используется для сборки пакетов и их обработки. Она возвращает количество пакетов, заданное аргументом `cnt`. Функция обратного вызова используется для обработки принятых пакетов (нам необходимо будет задать эту функцию). Для передачи дополнительной информации в эту функцию мы будем использовать параметр `*user`, являющийся указателем на переменную типа `u_char` (нам

необходимо будет самостоятельно производить приведение типов в зависимости от необходимого типа данных, передаваемых функции).

Прототип функции обратного вызова выглядит следующим образом:

```
void callback_function(u_char *arg, const struct pcap_pkthdr*  
pkthdr, const u_char* packet)
```

Первый аргумент является аргументом `*user`, переданным функции `pcap_loop()`; следующий аргумент является указателем на структуру, содержащую информацию о принятом пакете. Поля структуры `struct pcap_pkthdr` представлены ниже (взято из файла `pcap.h`):

```
struct pcap_pkthdr {  
    struct timeval ts; /* отметка времени */  
    bpf_u_int32 caplen; /* размер принятых данных */  
    bpf_u_int32 len; /* размер данного пакета */  
}
```

Альтернативой функции `pcap_loop()` является функция

```
pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback,  
u_char *user)
```

Единственным отличием является то, что это функция возвращает результат по истечении времени, заданного при вызове `pcap_open_live()`.

Фильтрация трафика

Для того чтобы отфильтровывать трафик приходящий на заданный порт, мы можем использовать функцию

```
int pcap_compile(pcap_t *p, struct bpf_program *fp,  
const char *str, int optimize, bpf_u_int32 mask)
```

Первый аргумент аналогичен во всех функциях библиотеки и рассмотрен ранее; второй аргумент представляет собой указатель на составленную версию фильтра. Следующий - это выражение для фильтра. Это выражение может быть названием протокола, таким как ARP, IP, TCP, UDP и т.д. Следующий аргумент устанавливает состояние оптимизации (0 - не оптимизировать, 1 - оптимизировать). Далее идет маска сети, с которой работает фильтр. Функция возвращает -1 в случае ошибки (в том случае, если обнаружена ошибка в выражении).

Чтобы применить фильтр используется функция

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

Второй аргумент функции - составленная версия выражения для фильтрации трафика.

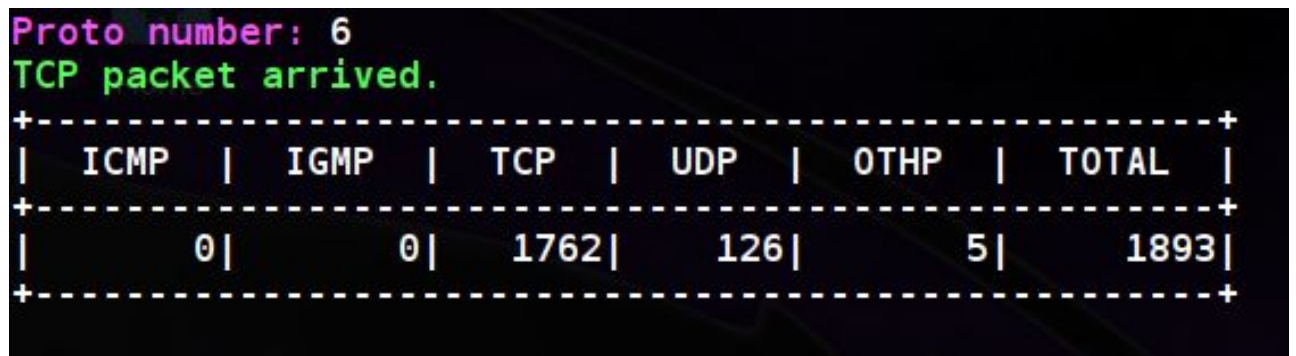
Получение информации IPv4

```
int pcap_lookupnet(const char *device, bpf_u_int32 *netp,  
                  bpf_u_int32 *maskp, char *errbuf)
```

При помощи этой функции можно получить сетевой адрес IPv4 и маску сети, присвоенные данному сетевому интерфейсу. Сетевой адрес будет записан по адресу *netp, а маска сети - по адресу *maskp. [2]

Реализация приложения

При запуске сниффера, пользователю либо предлагается выбрать любое устройство для прослушивания, которое присутствует в его системе, либо он может выбрать его самостоятельно, указав в качестве аргумента командной строки (например `eth0`). Затем ему необходимо указать целевое слово. Перехватываться будут только те пакеты, которые его содержат. Далее указывается фильтр (например `ip, tcp, tcp port 80` и тд) и начинается сбор информации с этого устройства. При перехвате каждого нового пакета, увеличивается счётчик для этого пакета, а также счётчик общих полученных пакетов. По этим данным строится и обновляется таблица перехваченных пакетов. Пример вывода данной таблицы изображён на рисунке 3.



Proto number: 6 TCP packet arrived.						
ICMP	IGMP	TCP	UDP	OTHER	TOTAL	
0	0	1762	126	5	1893	

Рисунок 3 – Вывод таблицы перехваченных пакетов.

Вся информация о каждом из перехваченных пакетов хранится в специальном log-файле.

Заключение

В результате выполнения курсовой работы осуществлено базовое ознакомление с библиотекой `librsar`, а также реализован анализатор трафика на основе данной библиотеки.

Список литературы

1. Фейт С. TCP/IP: Архитектура, протоколы, реализация (включая IP версии 6 и IP Security). — М.: Лори, 2000. — 424 с.
2. Linux по-русски // виртуальная энциклопедия: Захват пакетов при помощи библиотеки libpcap, — [ссылка](#).
3. Компьютер пресс // электронный журнал: Анализаторы сетевых пакетов, — [ссылка](#).

Листинг программы

```
/****** sniffer.c *****/
/******
*** gcc -lpcap sniffer.c -o sniffer ***
*****/

#include <arpa/inet.h>
#include <ctype.h>
#include <errno.h>
#include <net/ethernet.h>
#include <netinet/if_ether.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <pcap.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define BLK "\033[1;30m"
#define RED "\033[1;31m"
#define GRN "\033[1;32m"
#define YLW "\033[1;33m"
#define BLU "\033[1;34m"
#define MAG "\033[1;35m"
#define CYN "\033[1;36m"
#define WHT "\033[1;37m"

FILE *logfile;

u_char target[BUFSIZ] = "duck";
u_char chbuff[BUFSIZ];

void dumperr(char *str)
{
    perror(str);
    exit(1);
}

int checkinfo(const u_char *data, int size)
{
    int i, j, z = 0;
    for (i = 0; i < size; ++i) {
        if (i && !(i % 16)) {
            for (j = i - 16; j < i; ++j, ++z) {
                if (data[j] >= 32 && data[j] <= 128)
                    chbuff[z] = data[j];
                else
                    chbuff[z] = '.';
            }
        }
        if (i == size - 1) {
            for (j = i - i % 16; j <= i; ++j, ++z) {
```

```

        if (data[j] >= 32 && data[j] <= 128)
            chbuff[z] = data[j];
        else
            chbuff[z] = '.';
    }
}
chbuff[z] = '\0';
printf("\nBUF : %s\n\n", (u_char *)chbuff);
if (strstr(chbuff, target) != NULL)
    return 1;
else
    return 0;
}

void dumpinfo(const u_char *data, int size)
{
    int i, j;
    // if (strstr(data, target) != NULL) {
    for (i = 0; i < size; ++i) {
        if (i && !(i % 16)) {
            fprintf(logfile, "          ");
            for (j = i - 16; j < i; ++j) {
                if (data[j] >= 32 && data[j] <= 128)
                    fprintf(logfile, "%c", (u_char) data[j]);
                else
                    fprintf(logfile, ".");
            }
            fprintf(logfile, "\n");
        }

        if (!(i % 16)) fprintf(logfile, "      ");

        fprintf(logfile, " %02X", (u_int) data[i]);

        if (i == size - 1) {
            for (j = 0; j < 15 - i % 16; ++j)
                fprintf(logfile, "   ");

            fprintf(logfile, "          ");

            for (j = i - i % 16; j <= i; ++j) {
                if (data[j] >= 32 && data[j] <= 128)
                    fprintf(logfile, "%c", (u_char) data[j]);
                else
                    fprintf(logfile, ".");
            }
            fprintf(logfile, "\n");
        }
    }
    // }
    // else printf("NOT COMPARE\n");
}

void print_icmp_packet(const u_char *buffer, int size)
{
    u_short iphdrlen;

    struct iphdr *iph = (struct iphdr *) (buffer + sizeof(struct ethhdr));
    iphdrlen = iph->ihl * 4;

```

```

    struct icmphdr *icmph = (struct icmphdr *) (buffer + iphdrlen + sizeof(struct
ethhdr));
    int header_size = sizeof(struct ethhdr) + iphdrlen + sizeof(icmph);

    fprintf(logfile, "\n\n ***** ICMP PACKET *****\n\n");
    fprintf(logfile, "ICMP Header:\n");
    fprintf(logfile, " > Type\t:\t%d", (u_int) icmph->type);
    switch (icmph->type) {
        case 0:
            fprintf(logfile, " (ICMP Echo Request)\n");
            break;
        case 8:
            fprintf(logfile, " (ICMP Echo Reply)\n");
            break;
        case 11:
            fprintf(logfile, " (TTL Expired)\n");
            break;
        case 12:
            fprintf(logfile, " (IP Header Error)\n");
            break;
        default:
            fprintf(logfile, " (Unknown Error)\n");
            break;
    }
    fprintf(logfile, " > Code\t:\t%d\n", (u_int) icmph->code);
    fprintf(logfile, "\n\n ++++++++ DATA PAYLOAD ++++++++\n\n");
    dumpinfo(buffer + header_size, size - header_size);
    fprintf(logfile,
"\n=====");
}

void print_tcp_packet(const u_char *buffer, int size)
{
    u_short iphdrlen;

    struct iphdr *iph = (struct iphdr *) (buffer + sizeof(struct ethhdr));
    iphdrlen = iph->ihl * 4;

    struct tcphdr *tcph = (struct tcphdr *) (buffer + iphdrlen + sizeof(struct ethhdr));
    int header_size = sizeof(struct ethhdr) + iphdrlen + tcph->doff * 4;

    // if (strstr(buffer + header_size, target) != NULL) {
    if (checkinfo(buffer + header_size, size - header_size)) {
        fprintf(logfile, "\n\n ***** TCP PACKET *****\n\n");
        fprintf(logfile, "TCP Header:\n");
        fprintf(logfile, " > Src Port\t:\t%u\n", ntohs(tcph->source));
        fprintf(logfile, " > Dst Port\t:\t%u\n", ntohs(tcph->dest));
        fprintf(logfile, " > Hdr Len\t:\t%d bytes\n", (u_int) tcph->doff * 4);
        fprintf(logfile, "\n\n ++++++++ DATA PAYLOAD ++++++++\n\n");
        dumpinfo(buffer + header_size, size - header_size);
        fprintf(logfile,
"\n=====");
    }
}

void print_udp_packet(const u_char *buffer, int size)
{
    u_short iphdrlen;

    struct iphdr *iph = (struct iphdr *) (buffer + sizeof(struct ethhdr));
    iphdrlen = iph->ihl * 4;

```

```

struct udphdr *udph = (struct udphdr *) (buffer + iphdrlen + sizeof(struct ethhdr));
int header_size = sizeof(struct ethhdr) + iphdrlen + sizeof(udph);

// if (strstr(buffer + header_size, target) != NULL) {
//   if (checkinfo(buffer + header_size, size - header_size)) {
//     fprintf(logfile, "\n\n ***** UDP PACKET *****\n\n");
//     fprintf(logfile, "UDP Header:\n");
//     fprintf(logfile, " > Src Port\t:\t%u\n", ntohs(udph->source));
//     fprintf(logfile, " > Dst Port\t:\t%u\n", ntohs(udph->dest));
//     fprintf(logfile, " > UDP Len\t:\t%d\n", ntohs(udph->len));
//     fprintf(logfile, "\n\n ++++++ DATA PAYLOAD ++++++\n\n");
//     dumpinfo(buffer + header_size, size - header_size);
//     fprintf(logfile,
"\n=====
    }
  }

void process_packet(u_char *args, const struct pcap_pkthdr *pkth, const u_char *packet)
{
    static int icmp = 0, igmp = 0, tcp = 0, udp = 0, othp = 0, total = 0;

    int size = pkth->len;
    struct iphdr *iph = (struct iphdr *) (packet + sizeof(struct ethhdr));

    ++total;

    printf("\033[2J\033[1;1H");
    printf(MAG);
    printf("Proto number: ");
    printf(WHT);
    printf("%d\n", iph->protocol);

    switch (iph->protocol) {
    case 1:
        // icmp
        ++icmp;
        printf(GRN);
        printf("ICMP packet arrived.\n");
        printf(WHT);
        print_icmp_packet(packet, size);
        break;
    case 2:
        // igmp
        ++igmp;
        printf(GRN);
        printf("IGMP packet arrived.\n");
        printf(WHT);
        print_igmp_packet(packet, size);
        break;
    case 6:
        // tcp
        ++tcp;
        printf(GRN);
        printf("TCP packet arrived.\n");
        printf(WHT);
        print_tcp_packet(packet, size);
        break;
    case 17:
        // udp
        ++udp;
        printf(GRN);

```

```

        printf("UDP packet arrived.\n");
        printf(WHT);
        print_udp_packet(packet, size);
        break;
default:
    // unknown protocol
    ++othp;
    printf(RED);
    printf("Unknown packet arrived.\n");
    printf(WHT);
    break;
}

printf("+-----+\n");
printf("|  ICMP  |  IGMP  |  TCP  |  UDP  |  OTHP  |  TOTAL  |\n");
printf("+-----+\n");
printf("|%8d|%8d|%7d|%7d|%8d|%9d|\n", icmp, igmp, tcp, udp, othp, total);
printf("+-----+\n");
}

void file_check()
{
    // if ((logfile = fopen("log.txt", "r")) == NULL)
    //     dumperr("fopen");

    long res = 0;
    char buf[BUFSIZ];
    while (1) {
        while (!feof(logfile)) {
            fgets(buf, BUFSIZ, logfile);
            if (strcmp(buf, "duckduckgo") == 0) {
                printf("Got it! Position [%d] in the logfile.\n",
                    ftell(logfile));
                fclose(logfile);
                return;
            }
        }
        printf("Still nothing til [%d] ...\n", ftell(logfile));
        sleep(1);
    }
}

int main(int argc, char **argv)
{
    pthread_t fchecker;

    int cnt = 1, i, j, n, npacks = 0;
    char *devname = NULL, devs[100][100];
    char errbuf[PCAP_ERRBUF_SIZE];
    char filter_exp[32] = "ip";

    pcap_if_t      *alldevsp, *dev;
    pcap_t          *devhandler;

    const u_char *packet;
    struct pcap_pkthdr hdr;
    struct ether_header *eptr;
    struct bpf_program fp;

    bpf_u_int32 mask;
    bpf_u_int32 net;

```



```

if ((logfile = fopen("log.txt", "w")) == NULL)
    dumperr("fopen");

/**** CHOOSING DEVICE STAGE ****/

if (argc == 2)
    devname = argv[1];
else if (argc > 3) {
    fclose(logfile);
    dumperr("arguments");
}
else {
    printf("Finding available devices ... ");
    if (pcap_findalldevs(&alldevsp, errbuf))
        dumperr("pcap_findalldevs");
    printf("Done!\n");

    printf(YLW);
    printf("***** Devices *****\n");
    printf(WHT);
    for (dev = alldevsp; dev != NULL; dev = dev->next, cnt++) {
        printf(" %2d. %s - %s\n", cnt, dev->name, dev->description);

        if (dev->name != NULL)
            strcpy(devs[cnt], dev->name);
    }
    printf(YLW);
    printf("*****\n");
    printf(WHT);

    printf("Enter the number of device you want to sniff -> ");
    scanf("%d", &n);
    devname = devs[n];
}

/**** FILTERING STAGE ****/

printf("Input target word: ");
scanf("%s", &target);

printf("Input number of packets you want to sniff (-1 for inf): ");
scanf("%d", &npacks);

printf("Input filter expression: ");
scanf("%s", &filter_exp);

if (pcap_lookupnet(devname, &net, &mask, errbuf) == -1) {
    dumperr("pcap_lookupnet");
    net = 0;
    mask = 0;
}

/**** START SNIFFING ****/

printf("Opening device %s for sniffing word %s by %d packets with filter=%s ... ",
    devname, target, npacks, filter_exp);

sleep(2);

if ((devhandler = pcap_open_live(devname, BUFSIZ, 1, 0, errbuf)) == NULL)
    dumperr("pcap_open_live");
printf("Done!\n");

```

```
    if (pcap_compile(devhandler, &fp, filter_exp, 0, net) == -1)
        dumperr("pcap_compile");

    if (pcap_setfilter(devhandler, &fp) == -1)
        dumperr("pcap_setfilter");

//    pthread_create(&fchecker, NULL, (void *)file_check, NULL);
    pcap_loop(devhandler, npacks, process_packet, NULL);

    /*** END IT WHEN GET ALL PACKS ***/

    pcap_freecode(&fp);
    pcap_close(devhandler);
    fclose(logfile);

    printf("\nCapture complete!\n");

    return 0;
}
```