Keerti Kosana

CPSC 8580

September 19th, 2018

Project 1 Report

Problem 1: Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial.

1. pcap_lookupdev: Finds a capture device to sniff on
2. pcap_lookupnet: Returns the network number and mask for the capture device
3. pcap_open_live: Starts sniffing on the capture device
4. pcap_datalink: Returns the kind of device we're capturing on
5. pcap_compile: Compiles the filter expression stored in a regular stringin order to set the filter
6. pcap_setfilter: Sets the compiled filter
7. At this point, we can either sniff one packet at a time (pcap_next) or continuously sniff (pcap_loop). Since sniffex.cuses we'll continue with pcap_loop: Sets callback function for new (filtered!) packets
8. pcap_freecode: Frees up allocated memory generated by pcap_compile
9. pcap_close: Closes the sniffing session

Problem 2: Why do you need the root privilege to run sniffex? Where does the program fail if executed without the root privilege?

We need a root privilege to run sniffex.c because sniffex will need to access a network device which is a root user privilege.

The program fails here:

```
/* find a capture device if not specified on command-line */
        dev = pcap_lookupdev(errbuf);
        if (dev == NULL) {
                fprintf(stderr, "Couldn't find default device: %s\n",
                    errbuf);
                exit(EXIT_FAILURE);
        }
```

Problem 3: Please turn on and turn off the promiscuous mode in the sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you demonstrate this.

Promiscuous mode on: all traffic passes from a network controller

Promiscuous mode off: just the traffic that was intended to be received will pass.

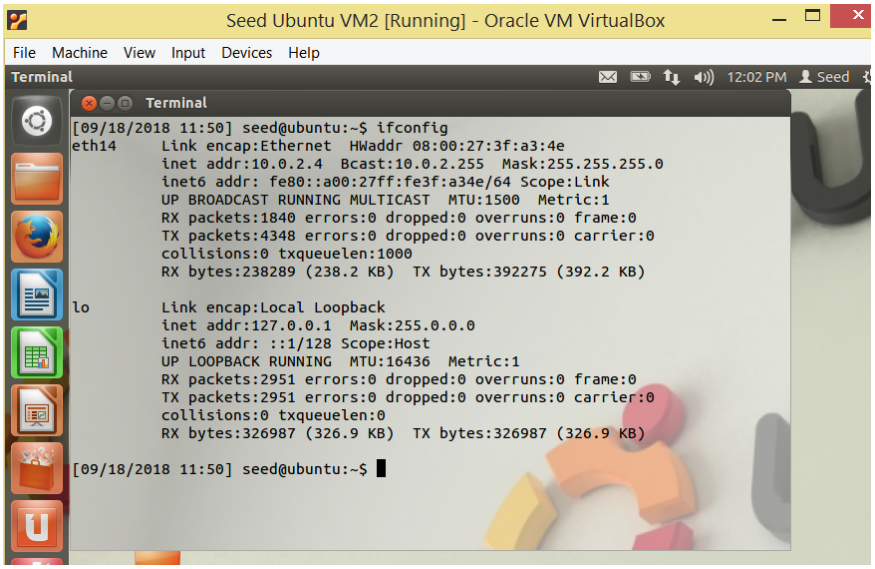
```
/* promisc mode on */
handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);

/* promisc mode off */
handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);
```

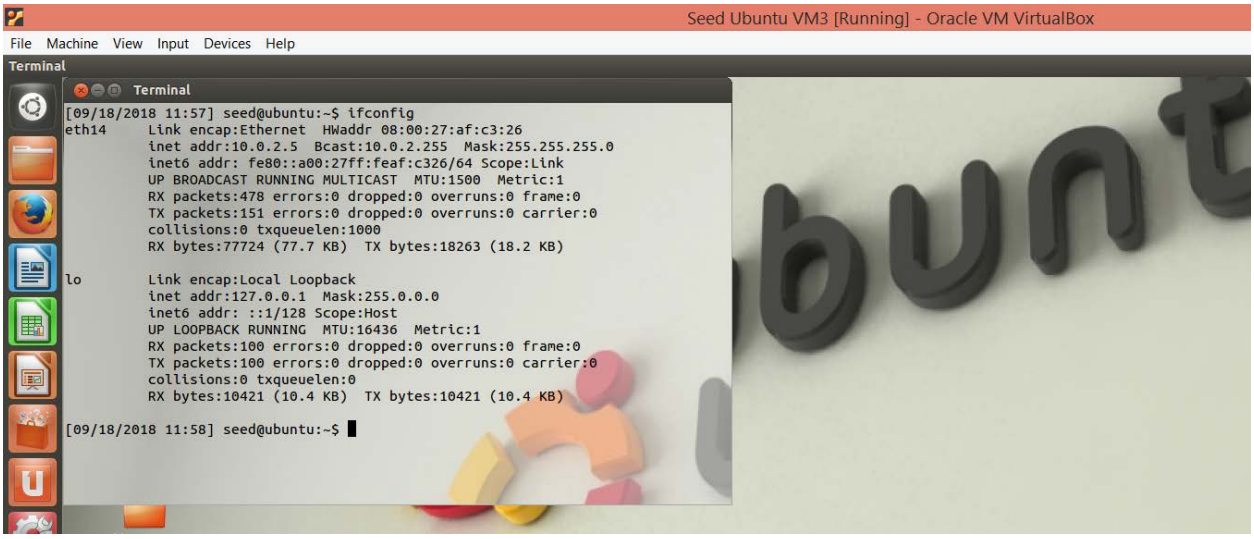
promiscuous mode on

IP Address of VM1

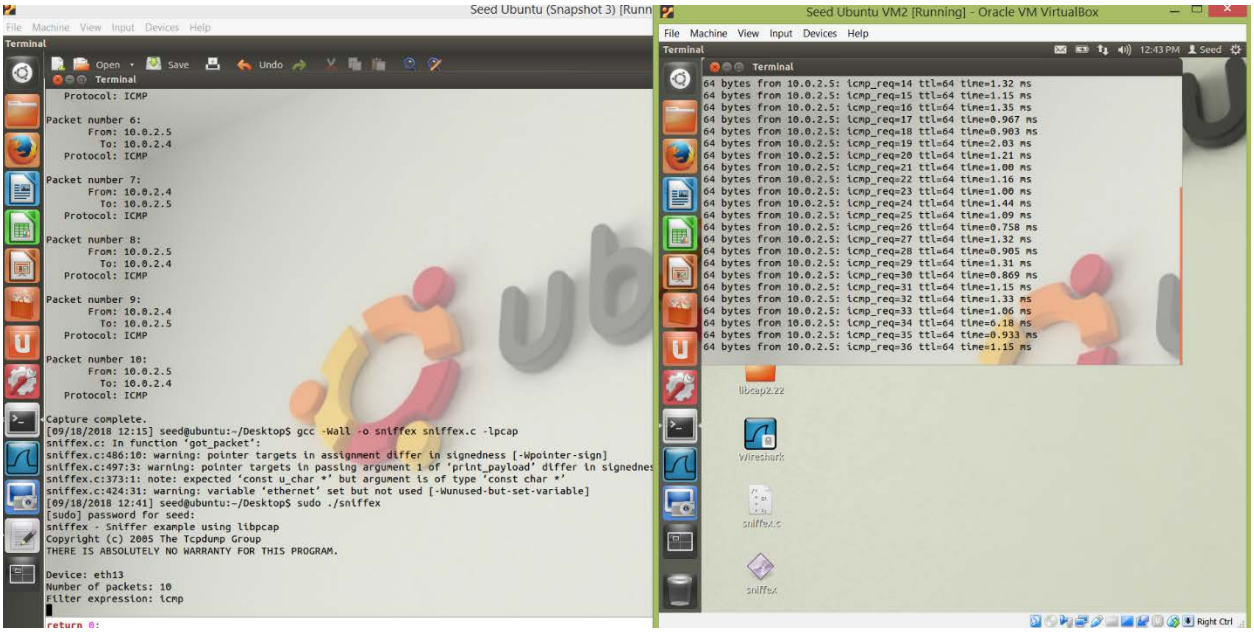
IP Address of VM2



IP Address of VM3

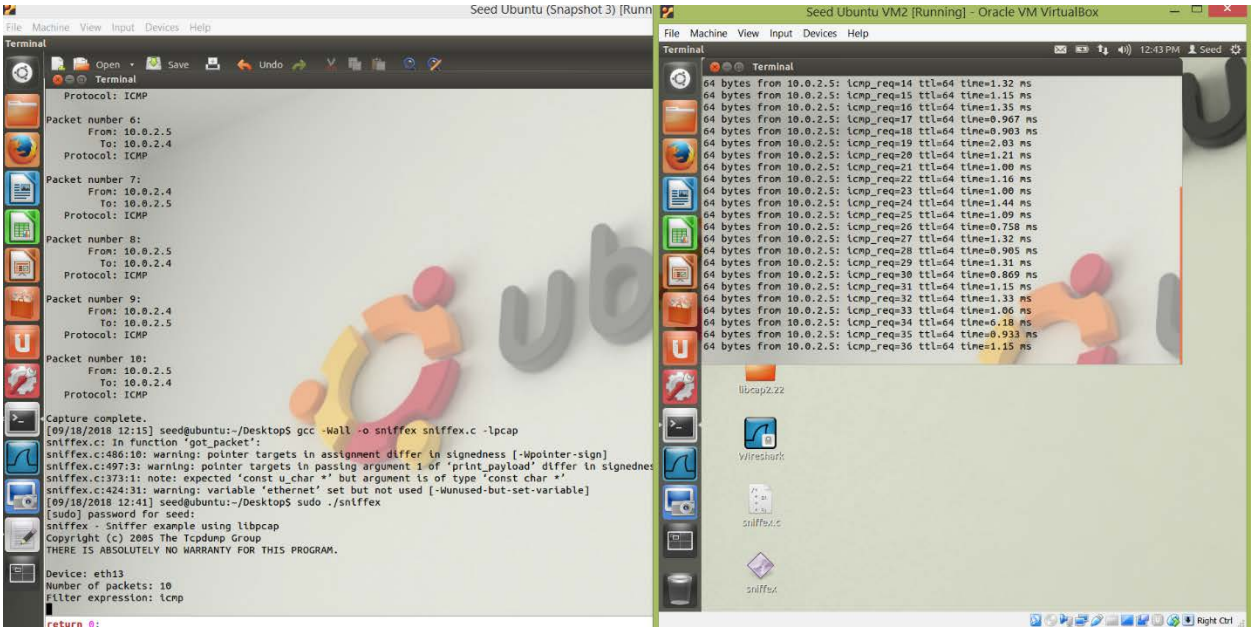


Changed the 0 to 1, ping from VM2 to VM3, and ran sniffex.c on sudo in VM1:

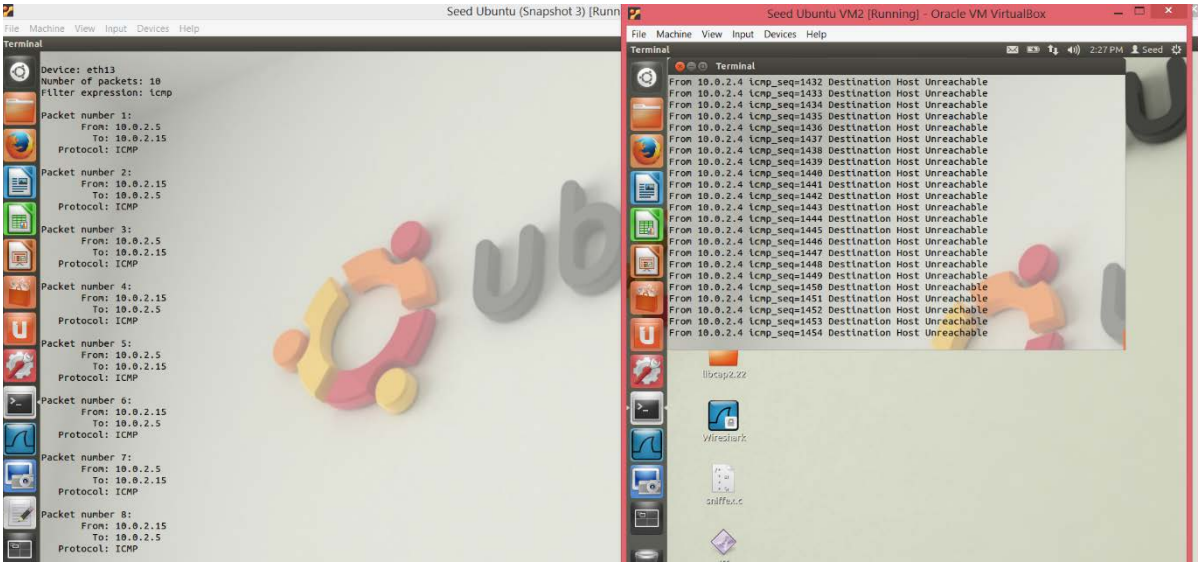


Promiscuous mode off

Changed 1 to 0, ping from VM2 to VM3, and ran sniffex.c on sudo in VM1:

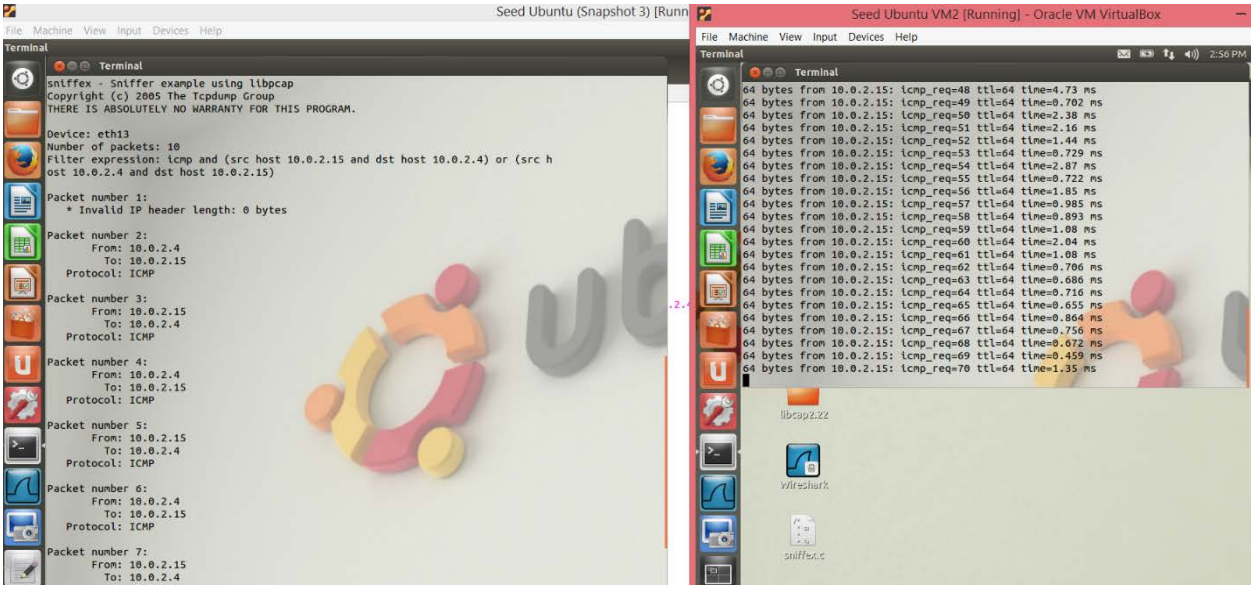


Ping from VM2 to VM1:



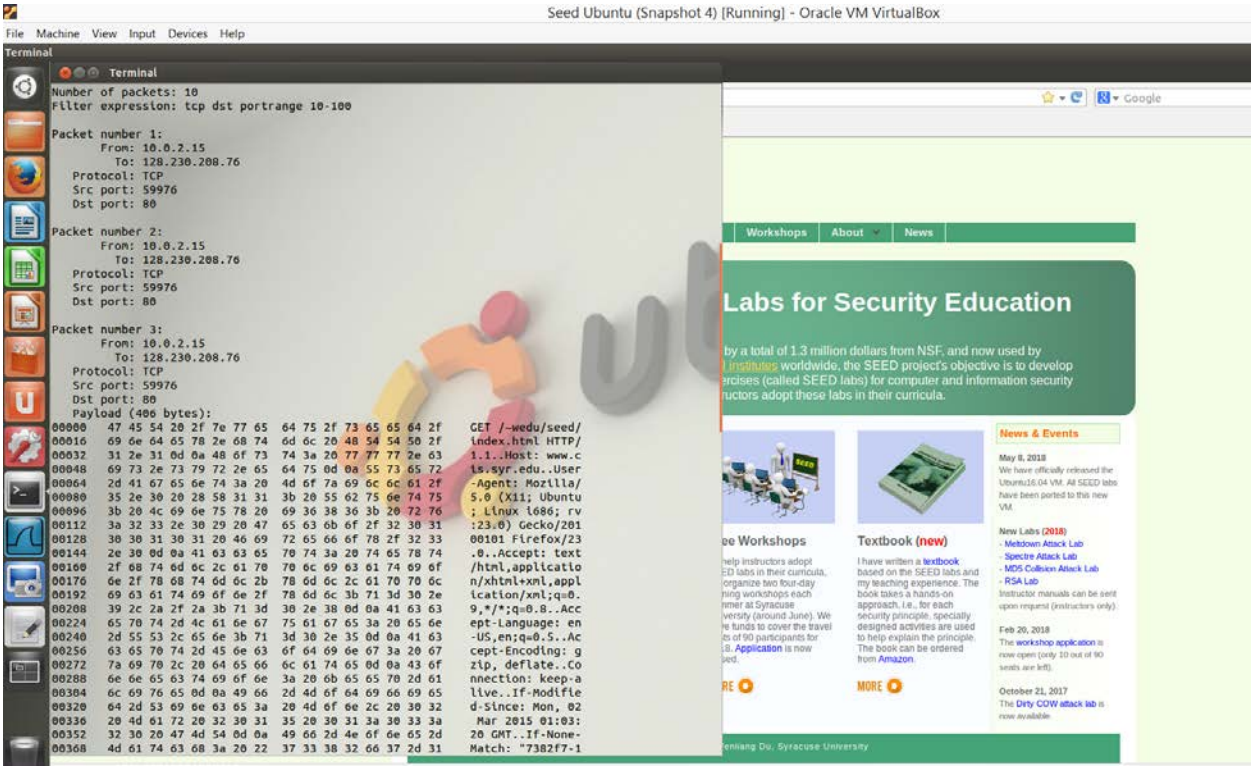
Problem 4: Please write filter expressions to capture each of the followings. In your lab reports, you need to include screendumps to show the results of applying each of these filters.

• Capture the ICMP packets between two specific hosts.



• Capture the TCP packets that have a destination port range from to port 10 - 100.

Ran sniffiex.c in VM1 and opened up a webpage in Firefox



```
/*
 * sniffex.c
 *
 * Sniffer example of TCP/IP packet capture using libpcap.
 *
 * Version 0.1.1 (2005-07-05)
 * Copyright (c) 2005 The Tcpdump Group
 *
 * This software is intended to be used as a practical example and
 * demonstration of the libpcap library; available at:
 * http://www.tcpdump.org/
 *
 ****************************************************************************
 *
 * This software is a modification of Tim Carstens' "sniffer.c"
 * demonstration source code, released as follows:
 *
 * sniffer.c
 * Copyright (c) 2002 Tim Carstens
 * 2002-01-07
 * Demonstration of using libpcap
 * timcarst -at- yahoo -dot- com
 *
 * "sniffer.c" is distributed under these terms:
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
```

* PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,

* REPAIR OR CORRECTION.

*

* IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

* WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR

* REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,

* INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING

* OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED

* TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY

* YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER

* PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE

* POSSIBILITY OF SUCH DAMAGES.

* <end of "sniffex.c" terms>

*

****

*

* Below is an excerpt from an email from Guy Harris on the tcpdump-workers

* mail list when someone asked, "How do I get the length of the TCP

* payload?" Guy Harris' slightly snipped response (edited by him to

* speak of the IPv4 header length and TCP data offset without referring

* to bitfield structure members) is reproduced below:

*

```
* The Ethernet size is always 14 bytes.
*
* <snip>...</snip>
*
* In fact, you *MUST* assume the Ethernet header is 14 bytes, *and*, if
* you're using structures, you must use structures where the members
* always have the same size on all platforms, because the sizes of the
* fields in Ethernet - and IP, and TCP, and... - headers are defined by
* the protocol specification, not by the way a particular platform's C
* compiler works.)
*
* The IP header size, in bytes, is the value of the IP header length,
* as extracted from the "ip_vhl" field of "struct sniff_ip" with
* the "IP_HL()" macro, times 4 ("times 4" because it's in units of
* 4-byte words).  If that value is less than 20 - i.e., if the value
* extracted with "IP_HL()" is less than 5 - you have a malformed
* IP datagram.
*
* The TCP header size, in bytes, is the value of the TCP data offset,
* as extracted from the "th_offx2" field of "struct sniff_tcp" with
* the "TH_OFF()" macro, times 4 (for the same reason - 4-byte words).
* If that value is less than 20 - i.e., if the value extracted with
* "TH_OFF()" is less than 5 - you have a malformed TCP segment.
*
* So, to find the IP header in an Ethernet packet, look 14 bytes after
* the beginning of the packet data.  To find the TCP header, look
* "IP_HL(ip)*4" bytes after the beginning of the IP header.  To find the
* TCP payload, look "TH_OFF(tcp)*4" bytes after the beginning of the TCP
```

* header.
*
* To find out how much payload there is:
*
* Take the IP *total* length field - "ip_len" in "struct sniff_ip"
* - and, first, check whether it's less than "IP_HL(ip)*4" (after
* you've checked whether "IP_HL(ip)" is >= 5). If it is, you have
* a malformed IP datagram.
*
* Otherwise, subtract "IP_HL(ip)*4" from it; that gives you the length
* of the TCP segment, including the TCP header. If that's less than
* "TH_OFF(tcp)*4" (after you've checked whether "TH_OFF(tcp)" is >= 5),
* you have a malformed TCP segment.
*
* Otherwise, subtract "TH_OFF(tcp)*4" from it; that gives you the
* length of the TCP payload.
*
* Note that you also need to make sure that you don't go past the end
* of the captured data in the packet - you might, for example, have a
* 15-byte Ethernet packet that claims to contain an IP datagram, but if
* it's 15 bytes, it has only one byte of Ethernet payload, which is too
* small for an IP header. The length of the captured data is given in
* the "caplen" field in the "struct pcap_pkthdr"; it might be less than
* the length of the packet, if you're capturing with a snapshot length
* other than a value >= the maximum packet size.
* <end of response>
*
****

```
*
* Example compiler command-line for GCC:
*   gcc -Wall -o sniffex sniffex.c -lpcap
*
****************************************************************************
*
* Code Comments
*
* This section contains additional information and explanations regarding
* comments in the source code. It serves as documentaion and rationale
* for why the code is written as it is without hindering readability, as it
* might if it were placed along with the actual code inline. References in
* the code appear as footnote notation (e.g. [1]).
*
* 1. Ethernet headers are always exactly 14 bytes, so we define this
* explicitly with "#define". Since some compilers might pad structures to a
* multiple of 4 bytes - some versions of GCC for ARM may do this -
* "sizeof (struct sniff_ethernet)" isn't used.
*
* 2. Check the link-layer type of the device that's being opened to make
* sure it's Ethernet, since that's all we handle in this example. Other
* link-layer types may have different length headers (see [1]).
*
* 3. This is the filter expression that tells libpcap which packets we're
* interested in (i.e. which packets to capture). Since this source example
* focuses on IP and TCP, we use the expression "ip", so we know we'll only
* encounter IP packets. The capture filter syntax, along with some
* examples, is documented in the tcpdump man page under "expression."
```

```
* Below are a few simple examples:
*
* Expression                Description
* ----------                -----------
* ip                                Capture all IP packets.
* tcp                               Capture only TCP packets.
* tcp port 80               Capture only TCP packets with a port equal to 80.
* ip host 10.1.2.3          Capture all IP packets to or from host 10.1.2.3.
*
******************************************************************************
*
*/

#define APP_NAME          "sniffex"
#define APP_DESC          "Sniffer example using libpcap"
#define APP_COPYRIGHT     "Copyright (c) 2005 The Tcpdump Group"
#define APP_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS
PROGRAM."

#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN	6

/* Ethernet header */
struct sniff_ethernet {
	u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
	u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
	u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip {
	u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
	u_char  ip_tos;                 /* type of service */
	u_short ip_len;                 /* total length */
	u_short ip_id;                  /* identification */
	u_short ip_off;                 /* fragment offset field */
	#define IP_RF 0x8000            /* reserved fragment flag */
	#define IP_DF 0x4000            /* dont fragment flag */
	#define IP_MF 0x2000            /* more fragments flag */
	#define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
```

```
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
        u_char  th_offx2;               /* data offset, rsvd */
#define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
        #define TH_RST  0x04
        #define TH_PUSH 0x08
        #define TH_ACK  0x10
        #define TH_URG  0x20
        #define TH_ECE  0x40
        #define TH_CWR  0x80
        #define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
```

```
        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
};

void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);

void
print_payload(const u_char *payload, int len);

void
print_hex_ascii_line(const u_char *payload, int len, int offset);

void
print_app_banner(void);

void
print_app_usage(void);

/*
 * app name/banner
 */
void
print_app_banner(void)
{

        printf("%s - %s\n", APP_NAME, APP_DESC);
```

```
        printf("%s\n", APP_COPYRIGHT);
        printf("%s\n", APP_DISCLAIMER);
        printf("\n");

return;
}

/*
 * print help text
 */
void
print_app_usage(void)
{

        printf("Usage: %s [interface]\n", APP_NAME);
        printf("\n");
        printf("Options:\n");
        printf("    interface    Listen on <interface> for packets.\n");
        printf("\n");

return;
}

/*
 * print data in rows of 16 bytes: offset   hex   ascii
 *
 * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET / HTTP/1.1..
 */
```

```
void
print_hex_ascii_line(const u_char *payload, int len, int offset)
{

        int i;
        int gap;
        const u_char *ch;

        /* offset */
        printf("%05d   ", offset);

        /* hex */
        ch = payload;
        for(i = 0; i < len; i++) {
                printf("%02x ", *ch);
                ch++;
                /* print extra space after 8th byte for visual aid */
                if (i == 7)
                        printf(" ");
        }
        /* print space to handle line less than 8 bytes */
        if (len < 8)
                printf(" ");

        /* fill hex gap with spaces if not full line */
        if (len < 16) {
                gap = 16 - len;
                for (i = 0; i < gap; i++) {
```

```
                printf("   ");
            }
        }
        printf("   ");

        /* ascii (if printable) */
        ch = payload;
        for(i = 0; i < len; i++) {
            if (isprint(*ch))
                printf("%c", *ch);
            else
                printf(".");
            ch++;
        }

        printf("\n");

return;
}

/*
 * print packet payload data (avoid printing binary data)
 */
void
print_payload(const u_char *payload, int len)
{

        int len_rem = len;
```

```
int line_width = 16;                    /* number of bytes per line */
int line_len;
int offset = 0;                                 /* zero-based offset counter */
const u_char *ch = payload;

if (len <= 0)
        return;

/* data fits on one line */
if (len <= line_width) {
        print_hex_ascii_line(ch, len, offset);
        return;
}

/* data spans multiple lines */
for ( ;; ) {
        /* compute current line length */
        line_len = line_width % len_rem;
        /* print line */
        print_hex_ascii_line(ch, line_len, offset);
        /* compute total remaining */
        len_rem = len_rem - line_len;
        /* shift pointer to remaining bytes to print */
        ch = ch + line_len;
        /* add offset */
        offset = offset + line_width;
        /* check if we have line width chars or less */
        if (len_rem <= line_width) {
```

```
                        /* print last line and get out */
                        print_hex_ascii_line(ch, len_rem, offset);
                        break;
                }
        }

return;
}

/*
 * dissect/print packet
 */
void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{

        static int count = 1;                   /* packet counter */

        /* declare pointers to packet headers */
        const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
        const struct sniff_ip *ip;              /* The IP header */
        const struct sniff_tcp *tcp;            /* The TCP header */
        const char *payload;                    /* Packet payload */

        int size_ip;
        int size_tcp;
        int size_payload;
```

```
printf("\nPacket number %d:\n", count);
count++;

/* define ethernet header */
ethernet = (struct sniff_ethernet*)(packet);

/* define/compute ip header offset */
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
if (size_ip < 20) {
        printf("   * Invalid IP header length: %u bytes\n", size_ip);
        return;
}

/* print source and destination IP addresses */
printf("       From: %s\n", inet_ntoa(ip->ip_src));
printf("         To: %s\n", inet_ntoa(ip->ip_dst));

/* determine protocol */
switch(ip->ip_p) {
        case IPPROTO_TCP:
                printf("   Protocol: TCP\n");
                break;
        case IPPROTO_UDP:
                printf("   Protocol: UDP\n");
                return;
        case IPPROTO_ICMP:
                printf("   Protocol: ICMP\n");
```

```
                return;
        case IPPROTO_IP:
                printf("   Protocol: IP\n");
                return;
        default:
                printf("   Protocol: unknown\n");
                return;
}

/*
 *  OK, this packet is TCP.
 */

/* define/compute tcp header offset */
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
size_tcp = TH_OFF(tcp)*4;
if (size_tcp < 20) {
        printf("   * Invalid TCP header length: %u bytes\n", size_tcp);
        return;
}

printf("   Src port: %d\n", ntohs(tcp->th_sport));
printf("   Dst port: %d\n", ntohs(tcp->th_dport));

/* define/compute tcp payload (segment) offset */
payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);

/* compute tcp payload (segment) size */
```

```
        size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);

        /*
         * Print payload data; it might be binary, so don't just
         * treat it as a string.
         */
        if (size_payload > 0) {
                printf("   Payload (%d bytes):\n", size_payload);
                print_payload(payload, size_payload);
        }

return;
}

int main(int argc, char **argv)
{

        char *dev = NULL;                       /* capture device name */
        char errbuf[PCAP_ERRBUF_SIZE];          /* error buffer */
        pcap_t *handle;                         /* packet capture handle */

        char filter_exp[] = "ip";               /* filter expression [3] */
        //char filter_exp[] = "icmp and (src host 192.168.0.38 and dst host 8.8.8.8) or (src host
8.8.8.8 and dst host 192.168.0.38)";            /* filter expression [3] */
        //char filter_exp[] = "tcp dst portrange 10-100";               /* filter expression [3] */

        struct bpf_program fp;                  /* compiled filter program (expression) */
        bpf_u_int32 mask;                       /* subnet mask */
        bpf_u_int32 net;                        /* ip */
```

```
int num_packets = 10;                   /* number of packets to capture */

print_app_banner();

/* check for capture device name on command-line */
if (argc == 2) {
        dev = argv[1];
}
else if (argc > 2) {
        fprintf(stderr, "error: unrecognized command-line options\n\n");
        print_app_usage();
        exit(EXIT_FAILURE);
}
else {
        /* find a capture device if not specified on command-line */
        dev = pcap_lookupdev(errbuf);
        if (dev == NULL) {
                fprintf(stderr, "Couldn't find default device: %s\n",
                    errbuf);
                exit(EXIT_FAILURE);
        }
}

/* get network number and mask associated with capture device */
if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
        fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
            dev, errbuf);
        net = 0;
```

```
        mask = 0;
    }

    /* print capture info */
    printf("Device: %s\n", dev);
    printf("Number of packets: %d\n", num_packets);
    printf("Filter expression: %s\n", filter_exp);

    /* open capture device */
    handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf); //change 0 to 1 to turn on promiscuous mode
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
        exit(EXIT_FAILURE);
    }

    /* make sure we're capturing on an Ethernet device [2] */
    if (pcap_datalink(handle) != DLT_EN10MB) {
        fprintf(stderr, "%s is not an Ethernet\n", dev);
        exit(EXIT_FAILURE);
    }

    /* compile the filter expression */
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s\n",
            filter_exp, pcap_geterr(handle));
        exit(EXIT_FAILURE);
    }
```

```
        /* apply the compiled filter */
        if (pcap_setfilter(handle, &fp) == -1) {
                fprintf(stderr, "Couldn't install filter %s: %s\n",
                    filter_exp, pcap_geterr(handle));
                exit(EXIT_FAILURE);
        }

        /* now we can set our callback function */
        pcap_loop(handle, num_packets, got_packet, NULL);

        /* cleanup */
        pcap_freecode(&fp);
        pcap_close(handle);

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

return 0;
}
```