```c
#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
```

```
66  #define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
67          u_short th_win;                 /* window */
68          u_short th_sum;                 /* checksum */
69          u_short th_urp;                 /* urgent pointer */
70  };
71
72  void
73  got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
    *packet);
74
75  void
76  print_payload(const u_char *payload, int len);
77
78  void
79  print_hex_ascii_line(const u_char *payload, int len, int offset);
80
81  void
82  print_app_banner(void);
83
84  void
85  print_app_usage(void);
86
87
88  /*
89   * print help text
90   */
91  void
92  print_app_usage(void)
93  {
94          printf("\n");
95          printf("Options:\n");
96          printf("    interface    Listen on <interface> for packets.\n");
97          printf("\n");
98
99          return;
100 }
101
102 /*
103  * print data in rows of 16 bytes: offset   hex   ascii
104  *
105  * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET / HTTP/1.1..
106  */
107 void
108 print_hex_ascii_line(const u_char *payload, int len, int offset)
109 {
110
111         int i;
112         int gap;
113         const u_char *ch;
114
115         /* offset */
116         printf("%05d   ", offset);
117
118         /* hex */
119         ch = payload;
120         for (i = 0; i < len; i++) {
121                 printf("%02x ", *ch);
122                 ch++;
123                 /* print extra space after 8th byte for visual aid */
124                 if (i == 7)
125                         printf(" ");
126         }
127         /* print space to handle line less than 8 bytes */
128         if (len < 8)
129                 printf(" ");
130
```

```c
        /* 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;
                }
        }
```

```
196
197             return;
198     }
199
200     /*
201      * dissect/print packet
202      */
203     void
204     got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
        *packet)
205     {
206
207             static int count = 1;                   /* packet counter */
208
209             /* declare pointers to packet headers */
210             const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
211             const struct sniff_ip *ip;              /* The IP header */
212             const struct sniff_tcp *tcp;            /* The TCP header */
213             const char *payload;                    /* Packet payload */
214
215             int size_ip;
216             int size_tcp;
217             int size_payload;
218
219             printf("\nPacket number %d:\n", count);
220             count++;
221
222             /* define ethernet header */
223             ethernet = (struct sniff_ethernet*)(packet);
224
225             /* define/compute ip header offset */
226             ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
227             size_ip = IP_HL(ip) * 4;
228             if (size_ip < 20) {
229                     printf("   * Invalid IP header length: %u bytes\n", size_ip);
230                     return;
231             }
232
233             /* print source and destination IP addresses */
234             printf("       From: %s\n", inet_ntoa(ip->ip_src));
235             printf("         To: %s\n", inet_ntoa(ip->ip_dst));
236
237             /* determine protocol */
238             switch (ip->ip_p) {
239             case IPPROTO_TCP:
240                     printf("   Protocol: TCP\n");
241                     break;
242             case IPPROTO_UDP:
243                     printf("   Protocol: UDP\n");
244                     return;
245             case IPPROTO_ICMP:
246                     printf("   Protocol: ICMP\n");
247                     return;
248             case IPPROTO_IP:
249                     printf("   Protocol: IP\n");
250                     return;
251             default:
252                     printf("   Protocol: unknown\n");
253                     return;
254             }
255
256             /*
257              *  OK, this packet is TCP.
258              */
259
260             /* define/compute tcp header offset */
```

```
261             tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
262             size_tcp = TH_OFF(tcp) * 4;
263             if (size_tcp < 20) {
264                     printf("   * Invalid TCP header length: %u bytes\n",
    size_tcp);
265                     return;
266             }
267
268             printf("   Src port: %d\n", ntohs(tcp->th_sport));
269             printf("   Dst port: %d\n", ntohs(tcp->th_dport));
270
271             /* define/compute tcp payload (segment) offset */
272             payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
273
274             /* compute tcp payload (segment) size */
275             size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
276
277             /*
278              * Print payload data; it might be binary, so don't just
279              * treat it as a string.
280              */
281             if (size_payload > 0) {
282                     printf("   Payload (%d bytes):\n", size_payload);
283                     print_payload(payload, size_payload);
284             }
285
286             return;
287     }
288
289     int main(int argc, char **argv)
290     {
291
292             char *dev = NULL;                       /* capture device name */
293             char errbuf[PCAP_ERRBUF_SIZE];          /* error buffer */
294             pcap_t *handle;                         /* packet capture handle */
295
296             char filter_exp[] = "ip";               /* filter expression [3] */
297             struct bpf_program fp;                  /* compiled filter program
    (expression) */
298             bpf_u_int32 mask;                       /* subnet mask */
299             bpf_u_int32 net;                        /* ip */
300             int num_packets = 10;                   /* number of packets to
    capture */
301
302             /* check for capture device name on command-line */
303             if (argc == 2) {
304                     dev = argv[1];
305             }
306             else if (argc > 2) {
307                     fprintf(stderr, "error: unrecognized command-line
    options\n\n");
308                     print_app_usage();
309                     exit(EXIT_FAILURE);
310             }
311             else {
312                     /* find a capture device if not specified on command-line */
313                     dev = pcap_lookupdev(errbuf);
314                     if (dev == NULL) {
315                             fprintf(stderr, "Couldn't find default device: %s\n",
316                                     errbuf);
317                             exit(EXIT_FAILURE);
318                     }
319             }
320
321             /* get network number and mask associated with capture device */
322             if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
```

```
323                     fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
324                             dev, errbuf);
325                     net = 0;
326                     mask = 0;
327             }
328
329             /* print capture info */
330             printf("Device: %s\n", dev);
331             printf("Number of packets: %d\n", num_packets);
332             printf("Filter expression: %s\n", filter_exp);
333
334             /* open capture device */
335             handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
336             if (handle == NULL) {
337                     fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
338                     exit(EXIT_FAILURE);
339             }
340
341             /* make sure we're capturing on an Ethernet device [2] */
342             if (pcap_datalink(handle) != DLT_EN10MB) {
343                     fprintf(stderr, "%s is not an Ethernet\n", dev);
344                     exit(EXIT_FAILURE);
345             }
346
347             /* compile the filter expression */
348             if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
349                     fprintf(stderr, "Couldn't parse filter %s: %s\n",
350                             filter_exp, pcap_geterr(handle));
351                     exit(EXIT_FAILURE);
352             }
353
354             /* apply the compiled filter */
355             if (pcap_setfilter(handle, &fp) == -1) {
356                     fprintf(stderr, "Couldn't install filter %s: %s\n",
357                             filter_exp, pcap_geterr(handle));
358                     exit(EXIT_FAILURE);
359             }
360
361             /* now we can set our callback function */
362             pcap_loop(handle, num_packets, got_packet, NULL);
363
364             /* cleanup */
365             pcap_freecode(&fp);
366             pcap_close(handle);
367
368             printf("\nCapture complete.\n");
369
370             return 0;
371     }
```