中南大学

《SEED PROJECT》

实验报告

学生姓名 陈文勇

指导教师 王伟平

学　　院 信息科学与工程学院

专业班级 信息安全 1402

完成时间 2016.12

实验二　Sniffing_Spoofing

一、实验原理

Sniffing 就是一种能将本地网卡状态设成‘混杂’状态的模式，当网卡处于这种“混杂”方式时，该网卡具备“广播地址”，它对遇到的每一个帧都产生一个硬件中断以便提醒操作系统处理流经该物理媒体上的每一个报文包。（绝大多数的网卡具备置成混杂模式的能力）

可见，Sniffer 工作在网络环境中的底层，它会拦截所有的正在网络上传送的数据，并且通过相应的软件处理，可以实时分析这些数据的内容，进而分析所处的网络状态和整体布局。值得注意的是：Sniffer 是极其安静的，它是一种消极的安全攻击。

嗅探器在功能和设计方面有很多不同。有些只能分析一种协议，而另一些可能能够分析几百种协议。一般情况下，大多数的嗅探器至少能够分析下面的协议：标准以太网、TCP/IP、IPX。

sniffing 是作用在网络基础结构的底层。通常情况下， 用户并不直接和该层打交道，有些甚至不知道有这一层存在。所以，应该说 Sniffer 的危害是相当之大的，通常，使用 Sniffer 是在网络中进行欺骗的开始。它可能造成的危害：

嗅探器能够捕获口令。这大概是绝大多数非法使用 Sniffer 的理由，Sniffer 可以记录到明文传送的用户名和口令。能够捕获专用的或者机密的信息。比如金融帐号，许多用户很放心在网上使用自己的信用卡或现金帐号，然而 Sniffer 可以很轻松截获在网上传送的用户姓名、口令、信用卡号码、截止日期、帐号和 pin。比如偷窥机密或敏感的信息数据，通过拦截数据包，入侵者可以很方便记录别人之间敏感的信息传送，或者干脆拦截整个的 email 会话过程。可以用来危害网络邻居的安全，或者用来获取更高级别的访问权限窥探低级的协议信息。

一般来说，sniffing 和 poofing 会联合起来使用。当攻击者嗅探到关键信息时，通常会使用 poofing 技术来构造数据包来劫持会话或者去获取更多信息，通常会造成很大的危害。Poofing 技术就是攻击者自己构造数据包的 ip/tcp 数据包帧头部数据来达到自己的目的。

本次实验就是基于以上原理，在 linux 下模拟整个过程。

二、实验器材

1．Ubuntu12.04。

2．Wireshark 等常用捕包工具。

三、实验步骤

Task1．编写嗅探程序

嗅探程序可以很容易地使用 pcap 库。利用 PCAP，嗅探器的任务变得在 pcap 库调用一系列简单的程序。在序列结束时，数据包将被放置在缓冲区中，以进一步处理，只要它们被捕获。所有的数据包捕获的细节由 pcap 库处理。Tim Carstens 写了一个教程如何使用 pcap 库写的嗅探程序。

Task2．包欺骗

在正常的情况下，当一个用户发送一个数据包时，操作系统通常不允许用户设置所有的在协议头字段（如 TCP，UDP，和 IP 报头）。操作系统将大部分的领域，而

只允许用户设置几个字段，如目标 IP 地址、目标端口号等。但是当用户有有 root 权限，他们可以在数据包标头设置为任意字段。这就是所谓的包欺骗，它可以通过原始套接字完成。

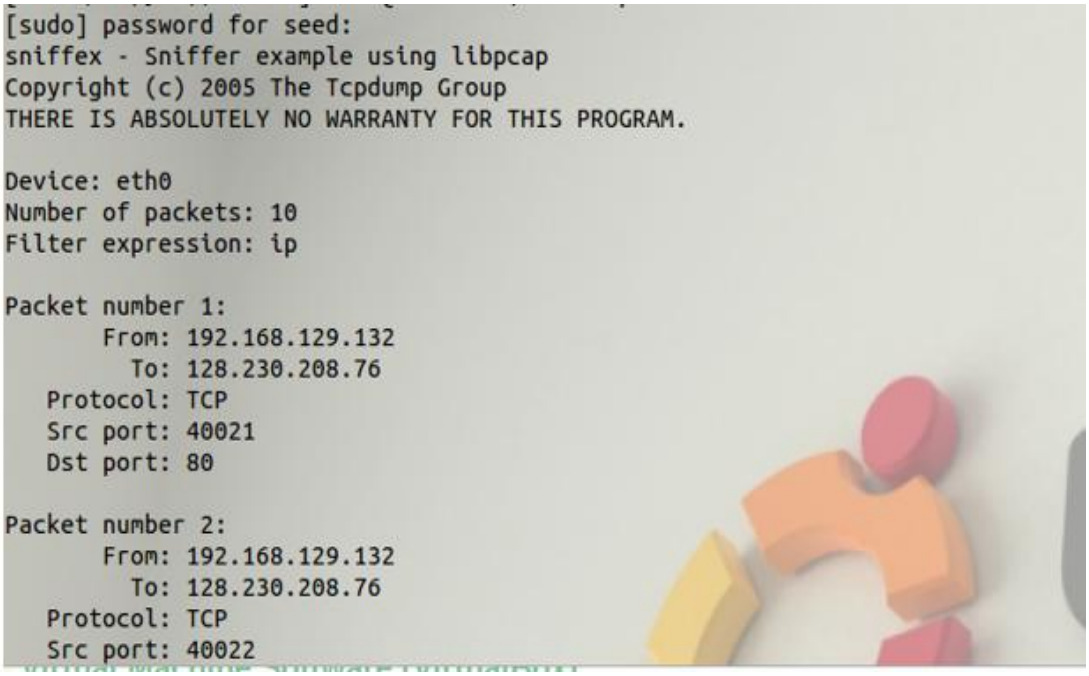
原始套接字给程序员的数据包结构的绝对控制，允许程序员构建任何任意的数据包，包括设置头字段和有效载荷。使用原始套接字是相当简单的，它包括四个步骤：（1）创建一个原始套接字，（2）设置套接字选项，（3）构建数据包，和（4）通过原始套接字发送数据包。有许多在线教程，可以教你如何使用原始套接字在 C 编程。我们已经把一些教程与实验室的网页联系起来了。请阅读它们，并学习如何写一个 spoonfing 程序包。我们展示了一个简单的的程序。

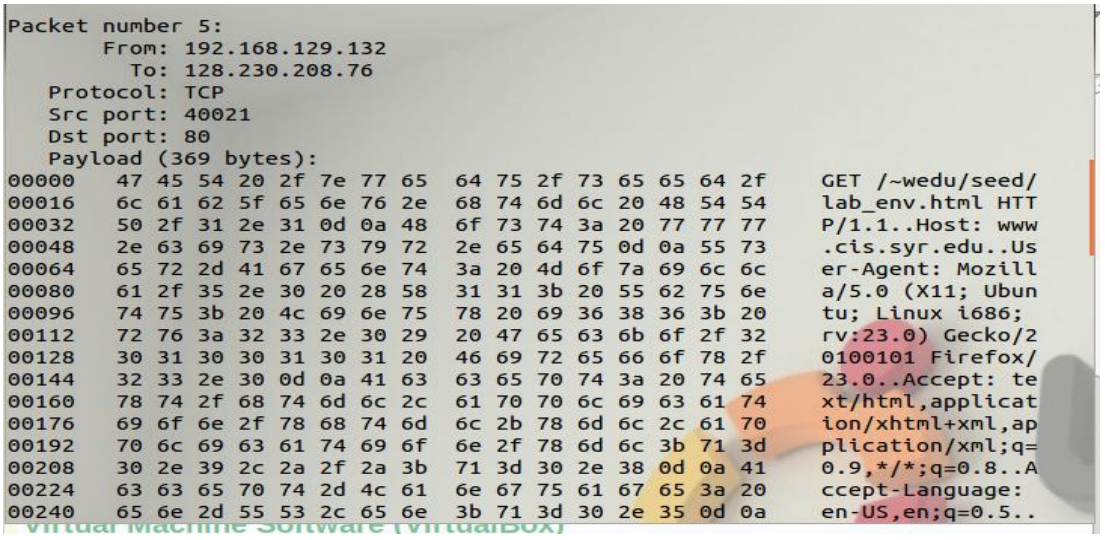
Task3：综合使用

在这个任务中，你将嗅探和欺骗技术实现连接，并实现程序。你需要在同一局域网两虚拟机。从 VMA ping 另一个 VM 的 IP，这将产生一个 ICMP 回送请求报文。如果 X 是活着的，ping 程序将收到一个回音答复，并打印出响应。你嗅探到数据包然后伪造程序运行在虚拟机 B、监控网络数据包嗅探。每当它看到 ICMP 回送请求，不管目标 IP 地址是什么，你的程序应该立即发出回声应答数据包欺骗技术的使用。因此，考虑到机器 X 是否是活的，这个程序将总是收到一个回复，这表明 X 是活的。你要写这样一个程序，包括在你显示你的程序的工作报告 screendumps。请在你的报告中附上代码。

四、实验结果

Task 1 运行结果如下：



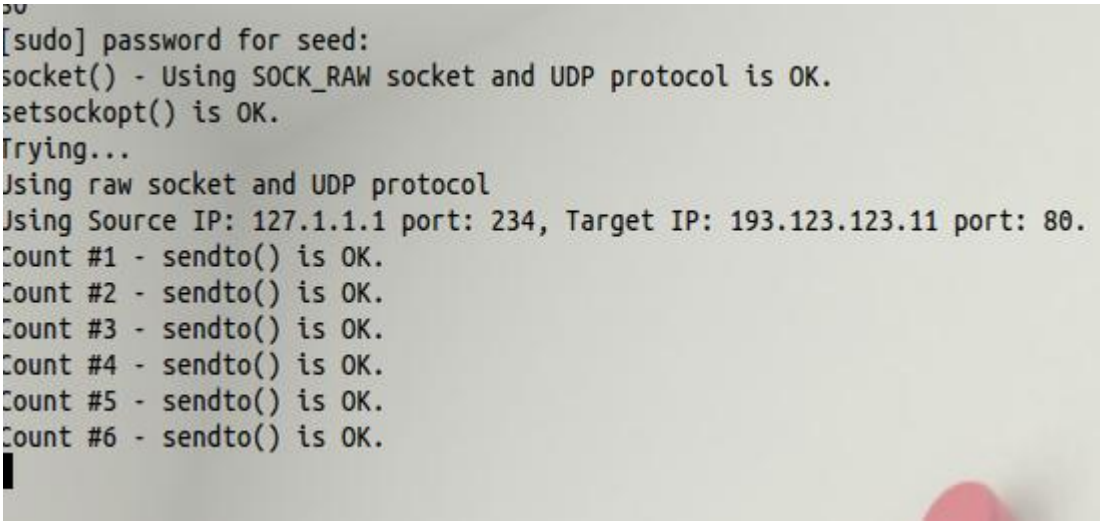


Tsak 1 运行截图

在程序中预设捕获 10 个数据包，当捕获数据包之后会将数据包进行处理，会下显示数据包的类型，还有数据包的源 ip 和目的 ip，源端口和目的端口，当有数据时还会显示数据。

对于任务一的 2，主要是修改 filter 中的过滤条件，要实现只捕获 ICMP 类型的数据包，只需要将 char filter_exp[] = "ip"中的 ip 改为 ICMP，然后要捕获端口在 10-100 之间的 tcp 数据包，同理，将这条语句中的条件改为‘tcp and dst portrange 10-100’即可。

Task 2 运行结果如下：



可以看到成功向 193.123.123.11 的 80 端口发送了伪造的的源 IP 为 127.1.1.1 且端口的 234 的数据包，这就实现包欺骗的过程。

实验心得：

在这次的网上实验当中，我觉得我学到了很多的东西，虽然这个实验对于我来说有很大的难度，但是在我一直问着同学，得到了很多同学的帮助，从而让我完成了这个实验。我对于我们专业的知识还是掌握的不是很到位，我翻阅了很多的资料，也有参考着文献，也参考着实验指导，做出了一大部分。我对于这个是满足的，至少我在这个实验当中学到了很多知识，这个对于我的以后来说有很大的帮助。古人云，磨刀不误砍柴工。前期的知识储备、文献储备、材料准备、方法准备可以避免手忙脚乱，充分的预实验使你充满信心。一步一个脚印，就不必“从头再来”。最不能容忍的是在开始的几步偷懒，造成后面总有一些无法排除的障碍。所以，对于我来说，我应该好好的掌握这些基础的知识，来好好的为以后做准备。

实验源代码：

Task1 代码：

```
#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) /*与 15 与运算*/
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)/*ip_vhl 的各二进位全部右移 4 位*/

/* 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);

void /*输出相关信息*/
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;
}
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;
}
void
print_hex_ascii_line(const u_char *payload, int len, int offset)
{
	int i;
	int gap;
	const u_char *ch;
	printf("%05d	", offset);
	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(" ");
	if (len < 16) {
		gap = 16 - len;
		for (i = 0; i < gap; i++) {
			printf("	");
		}
	}
	printf("	");
	ch = payload;
	for(i = 0; i < len; i++) {
		if (isprint(*ch))
			printf("%c", *ch);
		else
			printf(".");
		ch++;
	}
    printf("\n");
return;
```

```
}
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;
	if (len <= line_width) {
		print_hex_ascii_line(ch, len, offset);
		return;
	}
	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;
}
```

```
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;
    }
    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] */
    struct bpf_program fp;          /* compiled filter program (expression) */
```

```
bpf_u_int32 mask;              /*  子网掩码  */
bpf_u_int32 net;               /* IP 地址  */
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, 1, 1000, errbuf);
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);
    }
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {/*过滤表达式*/
        fprintf(stderr, "Couldn't parse filter %s: %s\n",
            filter_exp, pcap_geterr(handle));
        exit(EXIT_FAILURE);
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s\n",
            filter_exp, pcap_geterr(handle));
        exit(EXIT_FAILURE);
    }
    pcap_loop(handle, num_packets, got_packet, NULL);
    pcap_freecode(&fp);
    pcap_close(handle);
    printf("\nCapture complete.\n");
return 0;
}
```

Task2 代码：

```
#include<unistd.h>
#include<stdio.h>
#include<sys/socket.h>
#include<netinet/ip.h>
#include<netinet/udp.h>
#include<stdlib.h>
#define PCKT_LEN 8192
struct ipheader {
  unsigned char      iph_ihl:5, iph_ver:4;
  unsigned char      iph_tos;
```

```
  unsigned short int iph_len;
  unsigned short int iph_ident;
  unsigned char          iph_flag;
  unsigned short int iph_offset;
  unsigned char          iph_ttl;
  unsigned char          iph_protocol;
  unsigned short int iph_chksum;
  unsigned int           iph_sourceip;
  unsigned int           iph_destip;
};
  // UDP header's structure
struct udpheader {
  unsigned short int udph_srcport;
  unsigned short int udph_destport;
  unsigned short int udph_len;
  unsigned short int udph_chksum;
};
// total udp header length: 8 bytes (=64 bits)

// Function for checksum calculation. From the RFC,
// the checksum algorithm is:
//    "The checksum field is the 16 bit one's complement of the one's
//    complement sum of all 16 bit words in the header.    For purposes of
//    computing the checksum, the value of the checksum field is zero."
unsigned short csum(unsigned short *buf, int nwords)
{            //
             unsigned long sum;
             for(sum=0; nwords>0; nwords--)
                          sum += *buf++;
             sum = (sum >> 16) + (sum &0xffff);
             sum += (sum >> 16);
             return (unsigned short)(~sum);
}
// Source IP, source port, target IP, target port from the command line arguments
int main(int argc, char *argv[])
{
int sd;
```

```
// No data/payload just datagram
char buffer[PCKT_LEN];
// Our own headers' structures
struct ipheader *ip = (struct ipheader *) buffer;
struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
// Source and destination addresses: IP and port
struct sockaddr_in sin, din;
int one = 1;
const int *val = &one;

memset(buffer, 0, PCKT_LEN);

if(argc != 5)
{
printf("- Invalid parameters!!!\n");
printf("- Usage %s <source hostname/IP> <source port> <target hostname/IP> <target port>\n", argv[0]);
exit(-1);
}

// Create a raw socket with UDP protocol
sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
if(sd < 0)
{
perror("socket() error");
// If something wrong just exit
exit(-1);
}
else
printf("socket() - Using SOCK_RAW socket and UDP protocol is OK.\n");

// The source is redundant, may be used later if needed
// The address family
sin.sin_family = AF_INET;
din.sin_family = AF_INET;
// Port numbers
sin.sin_port = htons(atoi(argv[2]));
```

```
din.sin_port = htons(atoi(argv[4]));
// IP addresses
sin.sin_addr.s_addr = inet_addr(argv[1]);
din.sin_addr.s_addr = inet_addr(argv[3]);

// Fabricate the IP header or we can use the
// standard header structures but assign our own values.
ip->iph_ihl = 5;
ip->iph_ver = 4;
ip->iph_tos = 16; // Low delay
ip->iph_len = sizeof(struct ipheader) + sizeof(struct udpheader);
ip->iph_ident = htons(54321);
ip->iph_ttl = 64; // hops
ip->iph_protocol = 17; // UDP
// Source IP address, can use spoofed address here!!!
ip->iph_sourceip = inet_addr(argv[1]);
// The destination IP address
ip->iph_destip = inet_addr(argv[3]);

// Fabricate the UDP header. Source port number, redundant
udp->udph_srcport = htons(atoi(argv[2]));
// Destination port number
udp->udph_destport = htons(atoi(argv[4]));
udp->udph_len = htons(sizeof(struct udpheader));
// Calculate the checksum for integrity
ip->iph_chksum = csum((unsigned short *)buffer, sizeof(struct ipheader) +
sizeof(struct udpheader));
// Inform the kernel do not fill up the packet structure. we will build our own...
if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one)) < 0)
{
perror("setsockopt() error");
exit(-1);
}
else
printf("setsockopt() is OK.\n");

// Send loop, send for every 2 second for 100 count
```

```
printf("Trying...\n");
printf("Using raw socket and UDP protocol\n");
printf("Using Source IP: %s port: %u, Target IP: %s port: %u.\n", argv[1], atoi(argv[2]), argv[3], atoi(argv[4]));

int count;
for(count = 1; count <=20; count++)
{
if(sendto(sd, buffer, ip->iph_len, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
// Verify
{
perror("sendto() error");
exit(-1);
}
else
{
printf("Count #%u - sendto() is OK.\n", count);
sleep(2);
}
}
close(sd);
return 0;
}
```

Task 3 代码

```
#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) */
#include <unistd.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netdb.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#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."
#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 */
char* dstip;
char* srcip;
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 + 14);
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));
dstip=inet_ntoa(ip->ip_src);
//printf(" desip %s",dstip);
printf("               To: %s\n", inet_ntoa(ip->ip_dst));
srcip=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 + 14 + 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 + 14 + 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[] = " icmp";        /* filter expression    */
        struct bpf_program fp;              /* compiled filter program (expression) */
        bpf_u_int32 mask;           /* subnet mask */
        bpf_u_int32 net;          /* ip */
        int num_packets = 1;          /* 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);
    handle=pcap_open_live(dev,1518,1,1000,errbuf);
    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");
        int s, i;
        char buf[400];
        struct ip *ip = (struct ip *)buf;
        struct icmphdr *icmp = (struct icmphdr *)(ip + 1);
        struct hostent *hp, *hp2;
        struct sockaddr_in dst;
      int offset;
      int on;
      int num = 100;
      if(argc < 3)
      {
          printf("\nUsage: %s <saddress> <dstaddress> [number]\n", argv[0]);
          printf("- saddress is the spoofed source address\n");
          printf("- dstaddress is the target\n");
          printf("- number is the number of packets to send, 100 is the default\n");
          exit(1);
       }
            /* Create RAW socket */
            if((s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
            {
             perror("socket() error");
             /* If something wrong, just exit */
```

```
    exit(1);
    }
    if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
    {    perror("setsockopt() for IP_HDRINCL error");
  exit(1);
    if((hp = gethostbyname(dstip)) == NULL)
{
  if((ip->ip_dst.s_addr = inet_addr(dstip)) == -1)
  {
    fprintf(stderr, "%s: Can't resolve, unknown host.\n", argv[2]);
          exit(1);
}
 }
 else
    bcopy(hp->h_addr_list[0], &ip->ip_dst.s_addr, hp->h_length);
 if((hp2 = gethostbyname(srcip)) == NULL)
    {
     if((ip->ip_src.s_addr = inet_addr(srcip)) == -1)
     {
          fprintf(stderr, "%s: Can't resolve, unknown host\n", dstip);
          exit(1);

     }
    }
    else

          bcopy(hp2->h_addr_list[0], &ip->ip_src.s_addr, hp->h_length);
    printf("Sending to %s from spoofed %s\n", inet_ntoa(ip->ip_dst), srcip);
    ip->ip_v = 4;
    ip->ip_hl = sizeof*ip >> 2;
    ip->ip_tos = 0;
    ip->ip_len = htons(sizeof(buf));
    ip->ip_id = htons(4321);
    ip->ip_off = htons(0);
    ip->ip_ttl = 255;
    ip->ip_p = 1;
    ip->ip_sum = 0; /* Let kernel fills in */
```

```
            dst.sin_addr = ip->ip_dst;
            dst.sin_family = AF_INET;
            icmp->type = 0;
            icmp->code = 0;
            icmp->checksum = htons(~(ICMP_ECHO << 8));
            ip->ip_off = htons(offset >> 3);
            if(offset < 65120)
              ip->ip_off |= htons(0x2000);
            else
               ip->ip_len = htons(418); /* make total 65538 */
            if(sendto(s, buf, sizeof(buf), 0, (struct sockaddr *)&dst, sizeof(dst)) < 0)
      {
                fprintf(stderr, "offset %d: ", offset);
                perror("sendto() error");
          }
      else
         printf("sendto() is OK.\n");

                        if(offset == 0)
            {
            icmp->type = 0;
            icmp->code = 0;
            icmp->checksum = 0;
            }
          close(s);
            }
       return 0;
}
```