中南大學
CENTRAL SOUTH UNIVERSITY

网络安全课外实验

实验报告

学生姓名　　张昱邦

学　　号　　0906140219

专业班级　　信安 1402

指导教师　　王伟平

学　　院　　信息科学与工程学院

完成时间　　2016 年 12 月

目 录

实验一 HeartBleed Atrack..... 错误!未定义书签。
一、实验目的 ..... 错误!未定义书签。
二、实验内容 ..... 错误!未定义书签。
三、实验原理 ..... 错误!未定义书签。
四、实验环境 ..... 错误!未定义书签。
五、实验过程 ..... 错误!未定义书签。
七、攻击原理分析..... 错误!未定义书签。
八、实验总结 ..... 错误!未定义书签。
实验二 Packtes Sniffing and Spoofing.....1
一、实验目的 1
二、实验内容 1
三、实验原理 1
四、实验环境 1
五、实验过程 1
六、实验感想 3

实验二 Packtes Sniffing and Spoofing

一、实验目的

通过实验了解包嗅探以及包哄骗的过程，加深对包嗅探以及哄骗的原理的理解，增强动手实验能力。

二、实验内容

在 SEED Project 网站的指导下，通过查询资料，独立完成包嗅探以及哄骗攻击实验。

三、实验原理

1.包嗅探：将网卡设置为“混杂模式“就可以进行利用嗅探器嗅探，可以窃听网络上流经的数据包，捕获数据包后，对数据包中的信息进行分析。

2. 包欺骗：当攻击者嗅探到关键信息时，通常会使用 spoofing 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。Spoofing 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。

四、实验环境

Oracle VM VirtualBox

Ubuntu12.04

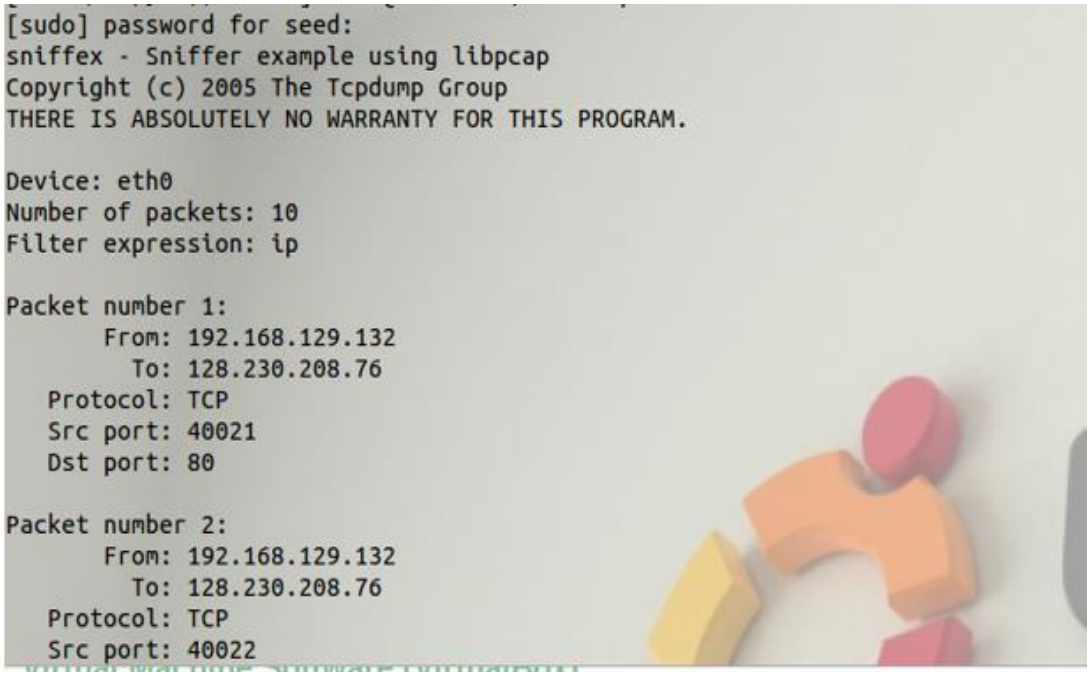
Wireshark 等常用捕包工具

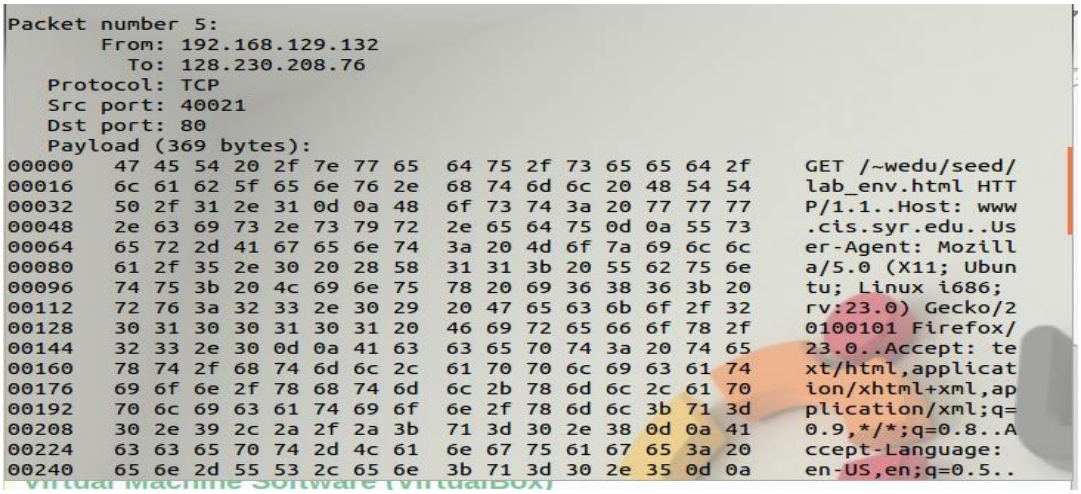
五、实验过程

1．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。

捕获的数据包如下图：

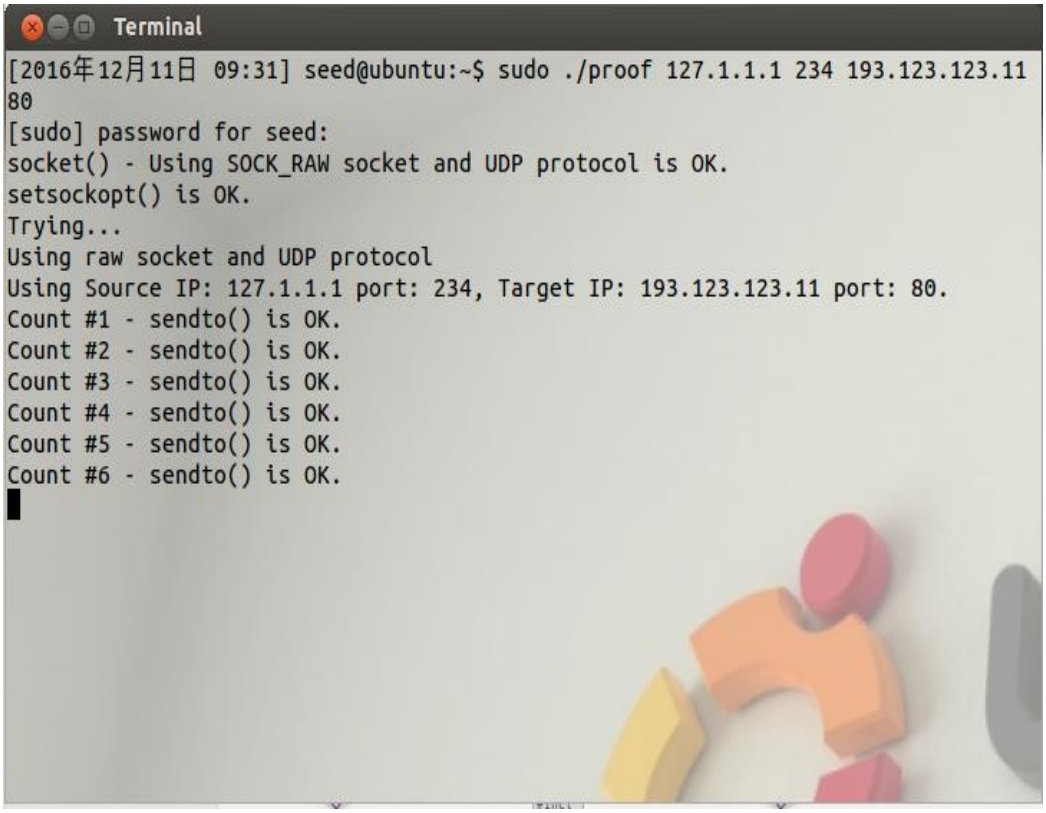




2．包欺骗

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。实验过程如下图：



3. 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。代码如附录所示。

六、实验感想

在这次的网上实验当中，我觉得我学到了很多的东西，尤其是对包窃听和哄骗的原理知识方面，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。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。在这次的网上实验当中，我觉得我学到了很多的东西，尤其是对包窃听和哄骗的原理知识方面虽然这个实

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。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。

任务三附录：

```
#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);
            }
```