

编程实现网络层数据抓包

Data Capture on Network Layer by Programming

This project receiving all the IPV4 data packages on network layer of the local host, printing the message of *head* and *data* of the package. The network traffic is monitored and displayed dynamically.

英才学院 2011 级 1 班

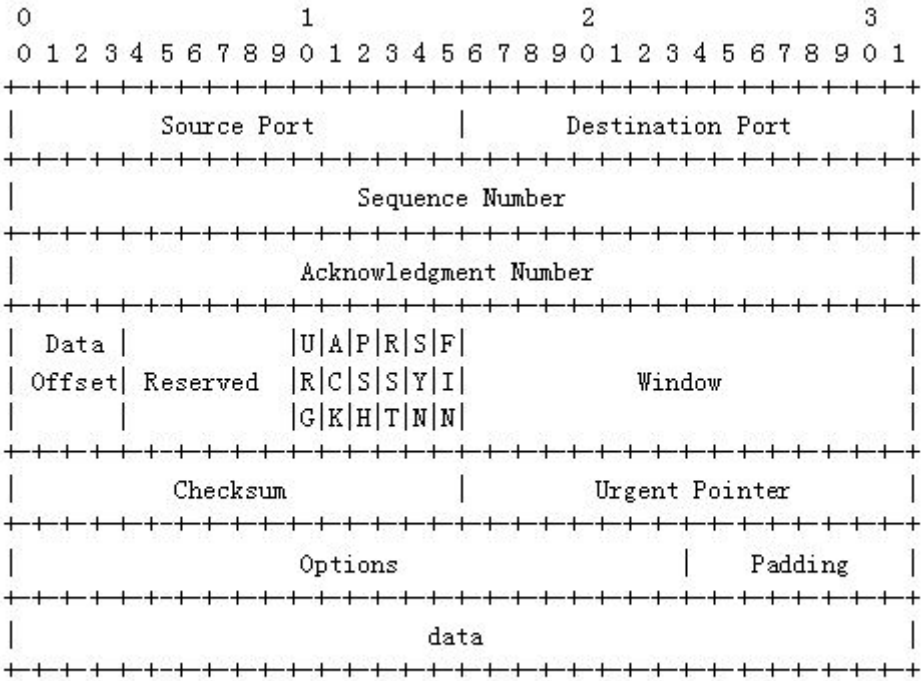
刘北北

学号 : 2011019060027

- 主要内容：**1.接收本机网络层所有 IPv4 数据分组，打印数据分组头部信息和数据信息。
- 2.动态统计展示网络流量。
- 3.按应用层协议类型，简单统计展示数据包流量。

一、基本原理：

TCP 头部：



TCP 包头格式

IP头部20 bytes

4	8	16	32 bits
Ver.	IHL	Type of service	Total length
Identification		Flags	Fragment offset
Time to live	Protocol	Header checksum	
Source address			
Destination address			
Option + Padding			
Data			
IP header structure			

二、wireshark 软件抓包：

Wireshark（前称 Ethereal）是一个网络封包分析软件。网络封包分析软件的功能是撷取网络封包，并尽可能显示出最为详细的网络封包资料。【来自百度百科】

刚连上网，我本人没有做任何操作时：

Wireshark 抓包截图显示网络流量。图中圈出了 ARP 请求和 TCP 三次握手过程。红色箭头指向数据包详情，显示以太网 II 和地址解析协议（ARP）请求。

Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: HonHaiPr_7a:5d:89 (e4:05:3d:7a:5d:89), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)

0000 ff ff ff ff ff e4 d5 3d 7a 5d 89 08 06 00 01 =z].....
0010 08 00 06 04 00 01 e4 d5 3d 7a 5d 89 b7 dc 4a ed =z]...3.
0020 00 00 00 00 00 00 b7 dc 48 01 H.

File: "C:\Users\dell\AppData\Local\Temp... Packets: 15 · Disposed: 15 (100.0%) · Dropped: 0 (0.0%) Profile: Default

用浏览器打开百度页面的时候：

Wireshark 抓包截图显示浏览器访问百度页面的网络流量。图中圈出了 TCP 三次握手和 HTTP GET 请求。红色箭头指向数据包详情，显示 HTTP 1.1 的 GET 请求。

Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: HonHaiPr_7a:5d:89 (e4:05:3d:7a:5d:89), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)

0000 ff ff ff ff ff e4 d5 3d 7a 5d 89 08 06 00 01 =z].....
0010 08 00 06 04 00 01 e4 d5 3d 7a 5d 89 b7 dc 4a ed =z]...3.
0020 00 00 00 00 00 00 b7 dc 48 01 H.

File: "C:\Users\dell\AppData\Local\Temp... Packets: 15 · Disposed: 15 (100.0%) · Dropped: 0 (0.0%) Profile: Default

三、编程：

1.先定义结构数组，确定 TCP、IP、UDP、ICMP 各协议的帧头部的各个字节内容：

```

typedef struct tcphdr //tcp头部: TCP协议头最少20个字节, 包括以下的区域:
{
    unsigned short int sport; //source address: TCP源端口: 16位初始化通信的端口。源端口和源IP地址的作用是标示报文的返回地址。
    unsigned short int dport; //destination address: TCP目的端口: 16位, 定义传输的目的, 指明包文接收计算机上的应用程序地址接口。
    unsigned int th_seq; //sequence number: TCP序列号, 32位。
    unsigned int th_ack; //acknowledge number: TCP应答号。32位的序列号由接收端计算机使用, 重组分段的包文成最初形式。
    //如果设置了ACK控制位, 这个值表示一个准备接收的包的序列码。
    unsigned char th_x2:4; //header length: 数据偏移量, 4位包括TCP头大小, 指示何处数据开始。
    unsigned char th_off:4; //reserved: 保留, 6位值域, 这些位必须是0。为了将来定义新的用途所保留。
    unsigned char th_flag; //6位标志域 flags: URG ACK PSH RST SYN FIN
    unsigned short int th_win; //window size, 16位, 用来表示想收到的每个TCP数据段的大小。
    unsigned short int th_sum; //check sum, 16位校验位。
    unsigned short int th_urp; //urgent pointer, 16位, 指向后面是优先数据的字节, 在URG标志设置了时才有效。
}TCP_HDR;

struct ipheader //ip头部
{
    unsigned char h_lenver; //version & header length, 指定IP协议的版本号&包头长度
    unsigned char ip_tos; //tos(Type of Service)服务类型

    unsigned short int ip_len; //total length 包长度

    unsigned short int ip_id; //id: 每一个IP封包都有一个16位的唯一识别码
    //是封包进行重组的时候的依据
    unsigned short int ip_off; //offset: 分段偏移, 封包进行分段的时候会为各片段做好定位记录, 以便在重组的时候就能够对号入座
    unsigned char ip_ttl; //time to live: 生存时间字段设置了数据报可以经过的最多路由器数, 表示数据报在网络上生存多久。
    unsigned char ip_p; //protocol: 该封包所使用的网络协议类型
    unsigned short int ip_sum; //check sum
    unsigned int ip_src; //source address
    unsigned int ip_dst; //destination address
}IP_HDR;

typedef struct udphdr //udp头部
{
    unsigned short sport; //source port
    unsigned short dport; //destination port
    unsigned short len; //UDP length
    unsigned short cksum; //check sum(include data)
}UDP_HDR;

typedef struct icmphdr //icmp头部: Internet Control Message Protocol(Internet控制报文协议)
{
    unsigned short sport;
    unsigned short dport;
    BYTE i_type;
    BYTE i_code;
    USHORT i_cksum;
    USHORT i_id;
    USHORT i_seq;
    ULONG timestamp;
}ICMP_HDR;

```

2. 定义 TCP 标志位：

```

//定义TCP的标志位
char TcpFlag[6]={'F','S','R','P','A','U'};
/*
 * F : FIN - 结束; 结束会话
 * S : SYN - 同步; 表示开始会话请求
 * R : RST - 复位; 中断一个连接
 * P : PUSH - 推送; 数据包立即发送
 * A : ACK - 应答
 * U : URG - 紧急
 * E : ECE - 显式拥塞提醒回应
 * W : CWR - 拥塞窗口减少
 */

```

3. WSA 初始化，获得主机名，并配置地址信息：


```

//WSA初始化, 并建立套接字sock, 其中AF_INET代表的是IPv4协议
WSAStartup(MAKEWORD(2,1), &wsd);
if((sock = socket(AF_INET, SOCK_RAW, IPPROTO_IP)) == SOCKET_ERROR)
{
    exit(0);
}

//调用gethostname获得主机名, 并通过主机名获得包含主机名和地址信息的hostent结构指针
/*struct hostent
{
    char FAR * h_name;
    char FAR * FAR * h_aliases;
    short h_addrtype;
    short h_length;
    char FAR * FAR * h_addr_list;
};*/
char FAR name[MAX_HOSTNAME_LEN];
gethostname(name, MAX_HOSTNAME_LEN);
struct hostent FAR * pHostent;
pHostent = (struct hostent *) malloc(sizeof(struct hostent));
pHostent = gethostbyname(name);

//配置地址信息
SOCKADDR_IN sa;
sa.sin_family = AF_INET;
sa.sin_port = htons(6000);
memcpy(&sa.sin_addr.S_un.S_addr, pHostent->h_addr_list[0], pHostent->h_length);
//从sa.sin_addr.S_un.S_addr所指的内存地址的起始位置开始拷贝pHostent->h_length个字节,
//到pHostent->h_addr_list[0]所指的内存地址的起始位置中
//即将配置好的地址放入pHostent相应的位置中

```

4. 套接字与地址绑定, 并定义报头指针:

```

//将套接字sock和地址sa绑定
bind(sock, (SOCKADDR *)&sa, sizeof(sa));

if ((WSAGetLastError()) == 10013) //试图使用被禁止的访问权限去访问套接字
    exit(0);

WSAIoctl(sock, SIO_RCVALL, &optval, sizeof(optval), NULL, 0, &dwBytesRet, NULL, NULL);

//定义了指向那些协议对应的头部的指针, 并初始化
struct udphdr *pUdpheader;
struct ipheader *pIpheader;
struct tcpheader *pTcpheader;
struct icmphdr *pIcmpheader;

char szSourceIP[MAX_ADDR_LEN], szDestIP[MAX_ADDR_LEN]; //源IP和目的IP
SOCKADDR_IN saSource, saDest;

pIpheader = (struct ipheader *) RecvBuf; //指针初始化
pTcpheader = (struct tcpheader *) (RecvBuf + sizeof(struct ipheader));
pUdpheader = (struct udphdr *) (RecvBuf + sizeof(struct ipheader));
pIcmpheader = (struct icmphdr *) (RecvBuf + sizeof(struct ipheader));

int iIpLen = sizeof(unsigned long) * (pIpheader->h_lenver & 0x0f);

```

5. 抓包前的准备:

```

//开始抓包前的准备工作：|
while (1)
{
    //把RecvBuf清零
    memset(RecvBuf, 0, sizeof(RecvBuf));
    recv(sock, RecvBuf, sizeof(RecvBuf), 0);

    //将按照网络字节顺序存储的源IP地址缀到szSourceIP后
    saSource.sin_addr.s_addr = pIpheader->ip_src; //s_addr是按照网络字节顺序存储IP地址
    strncpy(szSourceIP, inet_ntoa(saSource.sin_addr), MAX_ADDR_LEN);

    //将按照网络字节顺序存储的目的IP地址缀到szDestIP后
    saDest.sin_addr.s_addr = pIpheader->ip_dst;
    strncpy(szDestIP, inet_ntoa(saDest.sin_addr), MAX_ADDR_LEN);

    //计算将网络字节转化为主机字节后的各个协议头部长度
    lenip=ntohs(pIpheader->ip_len);
    lentcp =(ntohs(pIpheader->ip_len)-(sizeof(struct ipheader)+sizeof(struct tcpheader)));
    lenudp =(ntohs(pIpheader->ip_len)-(sizeof(struct ipheader)+sizeof(struct udphdr)));
    lenicmp =(ntohs(pIpheader->ip_len)-(sizeof(struct ipheader)+sizeof(struct icmp_hdr)));
}

```

6. 确认协议符合要求，开始抓包：

```

//确认IP头部的协议类型是IPPROTO_TCP，且头部长度不为零，就可以开始抓包了！
if((pIpheader->ip_p)==IPPROTO_TCP&&lentcp!=0)
{
    pCount++; //计数：正在抓第几个数据包
    dataip=(unsigned char *) RecvBuf;
    datatcp=(unsigned char *) RecvBuf+sizeof(struct ipheader)+sizeof(struct tcpheader); //data
    entity_content[65535]=*datatcp;
    |
    //打印数据包字节数据：
    //之前已经通过套接字sock将这些头部与本地地址连接，获取了这些协议头部的信息，现在只需要打出来即可

    printf("\n#####数据包[%i]=%d字节数据#####",pCount,lentcp);
    printf("\n#####IP协议头部#####\n");
    printf("标识:%i\n",ntohs(pIpheader->ip_id));
    printf("总长度:%i\n",ntohs(pIpheader->ip_len));
    printf("偏移量:%i\n",ntohs(pIpheader->ip_off));
    printf("生存时间:%d\n",pIpheader->ip_ttl);
    printf("服务类型:%d\n",pIpheader->ip_tos);
    printf("协议类型:%d\n",pIpheader->ip_p);
    printf("检验和:%i\n",ntohs(pIpheader->ip_sum));
    printf("源IP地址:%s ",szSourceIP);
    printf("\n目的IP地址:%s ",szDestIP);
    printf("\n#####TCP协议头部#####\n");
    printf("源端口:%i\n",ntohs(pTcpheader->sport));
    printf("目的端口:%i\n",ntohs(pTcpheader->dport));
    printf("序列号:%i\n",ntohs(pTcpheader->th_seq));
    printf("应答号:%i\n",ntohs(pTcpheader->th_ack));
    printf("检验和:%i\n",ntohs(pTcpheader->th_sum));
    printf("标志位: ");
    unsigned char FlagMask=1;
    int t=0,n=0,i5=0;
}

```

标志位的打印比较特殊：

```

printf("校验和:%i\n",ntohs(pTcpheader->th_sum));
printf("标志位: ");
unsigned char FlagMask=1;
int t=0,p=0,i5=0;
int lenhttp=0;
for(k=0;k<6;k++) //打印标志位时,用了FlagMask类似掩码的作用
{
    if((pTcpheader->th_flag)&FlagMask)
        printf("%c",TcpFlag[k]);
    else
        printf(" ");
    FlagMask=FlagMask<<1;
}

```

8. 运行结果：

由于网络数据交换比较快，不管是抓包软件还是程序都很快运行大量的数据，因此对程序加了小改动，

将 while (1) 改为了 for 循环，设置有限运行次数。

```

//开始抓包前的准备工作:
//while (1)
int cont;
for(cont=0;cont<50;cont++)
{
    //把RecvBuf清零
    memset(RecvBuf, 0, sizeof(Re

```

运行结果如下：

```
"E:\MYPROJECTS\package_capture\Debug\package_capture.exe"
检验和:38147
标志位:      A
#####数据包[17]=36字节数据#####
*****IP协议头部*****
标识:19168
总长度:76
偏移量:16384
生存时间:64
服务类型:0
协议类型:6
检验和:5813
源IP地址:183.220.77.210
目的IP地址:111.13.100.91
*****TCP协议头部*****
源端口:54596
目的端口:80
序列号:36967
应答号:50465
检验和:36707
标志位:      A
#####数据包[18]=28字节数据#####
*****IP协议头部*****
标识:19169
总长度:68
偏移量:16384
生存时间:64
服务类型:0
协议类型:6
检验和:5820
源IP地址:183.220.77.210
目的IP地址:111.13.100.91
*****TCP协议头部*****
源端口:54596
目的端口:80
序列号:36967
应答号:50465
检验和:47583
标志位:      A
#####数据包[19]=20字节数据#####
*****IP协议头部*****
标识:19170
总长度:60
偏移量:16384
```

与 wireshark 抓包软件结果一致：

No.	Time	Source	Destination	Protocol	Length	Info
51	6.124838000	183.220.77.210	111.13.100.91	TCP	62	54596 → http [SYN] Seq=0 win=8192 Len=0 MSS=1460 SACK_PERM=1
52	6.132285000	223.87.0.29	183.220.77.210	HTTP	56	HTTP/1.1 200 OK (text/plain)
53	6.132406000	183.220.77.210	223.87.0.29	TCP	54	54594 → http [ACK] Seq=1204 Ack=214 win=64027 Len=0
54	6.153044000	223.87.0.29	183.220.77.210	TCP	269	[TCP segment of a reassembled PDU]
55	6.167228000	223.87.0.29	183.220.77.210	HTTP	56	HTTP/1.1 200 OK (text/plain)
56	6.167231000	111.13.100.91	183.220.77.210	TCP	64	http → 54596 [SYN, ACK] Seq=0 Ack=1 win=8192 Len=0 MSS=1440 SACK_PERM=1
57	6.167402000	183.220.77.210	223.87.0.29	TCP	54	54595 → http [ACK] Seq=1204 Ack=217 win=64024 Len=0
58	6.167493000	183.220.77.210	111.13.100.91	TCP	54	54596 → http [ACK] Seq=1 Ack=1 win=64800 Len=0
59	6.177514000	183.220.77.210	111.13.100.91	HTTP	848	GET / HTTP/1.1
60	6.230916000	111.13.100.91	183.220.77.210	TCP	56	http → 54596 [ACK] Seq=1 Ack=795 win=40494 Len=0
61	6.307844000	111.13.100.91	183.220.77.210	TCP	458	[TCP segment of a reassembled PDU]
62	6.310685000	111.13.100.91	183.220.77.210	TCP	72	[TCP segment of a reassembled PDU]
63	6.310752000	183.220.77.210	111.13.100.91	TCP	54	54596 → http [ACK] Seq=795 Ack=420 win=64381 Len=0
64	6.311379000	111.13.100.91	183.220.77.210	TCP	1494	[TCP segment of a reassembled PDU]
65	6.311639000	111.13.100.91	183.220.77.210	TCP	1494	[TCP segment of a reassembled PDU]
66	6.311676000	183.220.77.210	111.13.100.91	TCP	54	54596 → http [ACK] Seq=795 Ack=3300 win=64800 Len=0
67	6.311833000	111.13.100.91	183.220.77.210	TCP	1494	[TCP segment of a reassembled PDU]
68	6.321046000	111.13.100.91	183.220.77.210	TCP	64	[TCP segment of a reassembled PDU]
69	6.321048000	111.13.100.91	183.220.77.210	TCP	64	[TCP Previous segment not captured] http → 54596 [PSH, ACK] Seq=6188 Ack=795
70	6.321048000	111.13.100.91	183.220.77.210	HTTP	64	[TCP Previous segment not captured] continuation of previous HTTP request

▶ Frame 58: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
 ▶ Ethernet II, Src: HonHaiPr_7a:5d:89 (e4:d5:3d:7a:5d:89), Dst: HuaweiTe_5e:52:32 (28:6e:d4:5e:52:32)
 ▶ Internet Protocol Version 4, Src: 183.220.77.210 (183.220.77.210), Dst: 111.13.100.91 (111.13.100.91)
 ▶ Transmission Control Protocol, Src Port: 54596 (54596), Dst Port: http (80), Seq: 1, Ack: 1, Len: 0

```

0000  28 6e d4 5e 52 32 e4 d5 3d 7a 5d 89 08 00 45 00  (n.AR2...=z]...E.
0010  00 28 43 d5 40 00 06 16 e4 b7 dc 4d d2 6f 0d    (.0.0.0...M.O.
0020  64 5b d5 44 00 50 90 67 c7 22 c5 21 1f 72 50 10  d[.D.P.g :.!rP.
0030  fd 20 c7 e9 00 00                                . ....
  
```

File: "C:\Users\dell\AppData\Local\Tem... Packets: 170 · Displayed: 154 (90.6%) · Dropped: 0 (0.0%) Profile: Default

9. 流量统计：

增加一个统计流量的变量 bit_total，将所有的 lentcp 累计求和，在所有包抓完后输出总比特数。

```
//统计流量
static long int bit_total=0;
bit_total+=lentcp;

printf("\n总流量为%d bit(s)",bit_total*8);
```

结果为：

```

E:\MYPROJECTS\package_capture\Debug\package_capture.exe
目的端口:80
序列号:27695
应答号:58095
检验和:18943
标志位: Pa
总流量为235400 bit(s)
数据包[34]=1318字节数据
*****IP协议头部*****
标识:11340
总长度:1358
偏移量:16384
生存时间:64
服务类型:0
协议类型:6
检验和:13668
源IP地址:183.220.72.180
目的IP地址:111.13.100.92
*****TCP协议头部*****
源端口:49761
目的端口:80
序列号:31812
应答号:30606
检验和:41931
标志位: PH
总流量为245944 bit(s)Press any key to continue.
  
```

动态流量统计

10. 按协议种类分类统计流量：

即增加一个判断协议类型的过程。在 TCPhead 中有 dport 和 sport 成员，表明目的端口号和源端口号。可以通过端口号特征判断协议类型。例如，在统计 http 协议的流量时，只有 TCPhead 中 dport 或 sport 成员是 80 时该协议时才累加。

```

//统计流量
static long int bit_total=0;
bit_total+=lentcp;

//统计http流量
static long int bit_total_http=0;
bit_total_http+=lenhttp;

//如果源端口和目的端口至少有一个是80号,说明是http协议的包
if(ntohs(pTcpheader->sport)==80||ntohs(pTcpheader->dport)==80)
for(j=0;j<lentcp;j++)
{
    //检测http协议:
    if( *(datatcp+j)==0x0d&&*(datatcp+j+1)==0x0a&&*(datatcp+j+2)==0x0d&&*(datatcp+j+3)==0x0a)
    {
        lenhttp=j;
        bit_total_http+=lenhttp;
        break;
    }
}
}
  
```

```

E:\MYPROJECTS\package_capture\Debug\package_capture.exe
应答号:63503
检验和:42483
标志位:    PA
总流量为261536 bit(s)
http总流量为40352 bit(s)
#####数据包:321=1字节数据#####
*****IP协议头部*****
标识:14848
总长度:41
偏移量:16384
生存时间:64
服务类型:0
协议类型:6
检验和:960
源IP地址:183.220.72.180
目的IP地址:221.176.30.206
*****TCP协议头部*****
源端口:49332
目的端口:5201
序列号:1429
应答号:65129
检验和:24737
标志位:    A
总流量为36514 bit(s)
http总流量为42656 bit(s)

```

【体会与收获】:

1. 通过本次实验,我初步了解了网络抓包的原理和方法,对网络协议也有了深一步的认识。尤其对加“头部”的概念有深刻领会。有的时候只有亲自写代码才能深入理解其中微妙。
2. 由于第一次接触,代码是从互联网上下载的,但经过仔细学习推敲,认真理解,有很大收获,并且能够根据题目要求适当改进,实现了功能,在模仿中学习进步。
3. 网络抓包是一种了解网络运行状况的很好手段,wireshark 以及其他类似软件有很多功能,等待我进一步学习后使用。
4. 第一次作业我用的是 linux 操作系统(在 windows 上搭的虚拟机),由于虚拟机影响电脑速度,这次使用的 windows 系统,发现从目前的水平来看,其实不同的操作系统 API 不同,编程实现功能的时候基本原理还是类似的。Windows 有一些自身的函数,如 WAS.....windows 和 linux 的具体区别还有待进一步学习摸索。

【附:源代码】:

```
#include <StdAfx.h>
```

```
#include <winsock2.h>
#include <windows.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define MAX_HOSTNAME_LAN 255
#define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)
#define MAX_ADDR_LEN 16
#pragma comment(lib,"WS2_32.lib")
```

typedef struct tcpheader //tcp 头部：TCP 协议头最少 20 个字节，包括以下的区域：

{

 unsigned short int sport; //source address：TCP 源端口：16 位初始化通信的端口。源端口和

源 IP 地址的作用是标示报文的返回地址。

 unsigned short int dport; //destination address：TCP 目的端口：16 位,定义传输的目的，

指明包文接收计算机上的应用程序地址接口。

 unsigned int th_seq; //sequence number：TCP 序列号，32 位。

 unsigned int th_ack; //acknowledge number：TCP 应答号。32 位的序列号由接收端计

算机使用，重组分段的包文成最初形式。

 //如果设置了 ACK 控制位，这个值表示一个准备接收的包的序列码。

 unsigned char th_x2:4; //header length: 数据偏移量,4 位包括 TCP 头大小，指示何处

数据开始。

 unsigned char th_off:4; //reserved:保留,6 位值域，这些位必须是 0。为了将来定义新的用

途所保留。

 unsigned char th_flag; //6 位标志域 flags: URG ACK PSH RST SYN FIN

 unsigned short int th_win; //window size :16 位 ,用来表示想收到的每个 TCP 数据段的大小。

 unsigned short int th_sum; //check sum : 16 位校验位。


```
    unsigned short int th_urp;        //urgent pointer : 16 位 , 指向后面是优先数据的字节 , 在 URG
```

标志设置了时才有效。

```
}TCP_HDR;
```

```
struct ipheader //ip 头部
```

```
{
```

```
    unsigned char h_lenver;           //version & header length : 指定 IP 协议的版本号&包
```

头长度

```
    unsigned char ip_tos;             //tos(Type of Service)服务类型
```

```
    unsigned short int ip_len;        //total length 包长度
```

```
    unsigned short int ip_id;         //id:每一个 IP 封包都有一个 16 位的唯一识别码
```

```
    //是封包进行重组的时候的依据
```

```
    unsigned short int ip_off;        //offset:分段偏移,封包进行分段的时候会为各片段做好
```

定位记录,以便在重组的时候就能够对号入座

```
    unsigned char ip_ttl;             //time to live:生存时间字段设置了数据报可以经过的最多
```

路由器数,表示数据包在网络上生存多久。

```
    unsigned char ip_p;               //protocol:该封包所使用的网络协议类型
```

```
    unsigned short int ip_sum;        //check sum
```

```
    unsigned int ip_src;              //source address
```

```
    unsigned int ip_dst;              //destination address
```

```
}IP_HDR;
```

```
typedef struct udphdr //udp 头部
```

```
{
```

```
    unsigned short sport;             //source port
```

```
    unsigned short dport;             //destination port
```

```
    unsigned short len;               //UDP length
```

```
    unsigned short cksum;             //check sum(include data)
```

```
} UDP_HDR;
```

```
typedef struct icmphdr //icmp 头部:Internet Control Message Protocol(Internet 控制报文协议)
```

```
{
    unsigned short sport;
    unsigned short dport;
    BYTE i_type;
    BYTE i_code;
    USHORT i_cksum;
    USHORT i_id;
    USHORT i_seq;
    ULONG timestamp;
}ICMP_HDR;
```

```
int main(int argc, char* argv[])
```

```
{
    SOCKET sock;
    WSADATA wsd;
    char RecvBuf[65535] = {0};
    char entity_content[65535]={0};
    char temp[65535]= {0};
    DWORD dwBytesRet;
    int pCount=0;
    unsigned int optval = 1;
    unsigned char *dataip=NULL;
    unsigned char *datatcp=NULL;
    unsigned char *dataudp=NULL;
    unsigned char *dataicmp=NULL;
    int lentcp=0, lenudp, lenicmp, lenip;
    int k;

    //定义 TCP 的标志位

    char TcpFlag[6]={'F','S','R','P','A','U'};

    /* * F : FIN - 结束; 结束会话

        * S : SYN - 同步; 表示开始会话请求

        * R : RST - 复位;中断一个连接

        * P : PUSH - 推送; 数据包立即发送

        * A : ACK - 应答

        * U : URG - 紧急
```

* E : ECE - 显式拥塞提醒回应

* W : CWR - 拥塞窗口减少

*/

//WSA 初始化，并建立套接字 sock,其中 AF_INET 代表的是 IPv4 协议

WSAStartup(MAKEWORD(2,1),&wsd);

if((sock = socket(AF_INET, SOCK_RAW, IPPROTO_IP))!=SOCKET_ERROR)

{

exit(0);

}

//调用 gethostname 获得主机名，并通过主机名获得包含主机名和地址信息的 hostent 结构指针

/*struct hostent

{ char FAR * h_name;

char FAR * FAR * h_aliases;

short h_addrtype;

short h_length;

char FAR * FAR * h_addr_list;

};*/

char FAR name[MAX_HOSTNAME_LAN];

gethostname(name, MAX_HOSTNAME_LAN);

struct hostent FAR * pHostent;

pHostent = (struct hostent *)malloc(sizeof(struct hostent));

pHostent = gethostbyname(name);

//配置地址信息

SOCKADDR_IN sa;

sa.sin_family = AF_INET;

sa.sin_port = htons(6000);

memcpy(&sa.sin_addr.S_un.S_addr, pHostent->h_addr_list[0], pHostent->h_length);

//从 sa.sin_addr.S_un.S_addr 所指的内存地址的起始位置开始拷贝 pHostent->h_length 个字节，

//到 pHostent->h_addr_list[0]所指的内存地址的起始位置中

//即将配置好的地址放入 pHostent 相应的位置中

//将套接字 sock 和地址 sa 绑定

```
bind(sock, (SOCKADDR *)&sa, sizeof(sa));
```

```
if ((WSAGetLastError())==10013) //试图使用被禁止的访问权限去访问套接字  
exit(0);
```

```
WSAIoctl(sock, SIO_RCVALL, &optval, sizeof(optval), NULL, 0, &dwBytesRet, NULL, NULL);
```

```
//定义了指向那些协议对应的头部的指针，并初始化
```

```
struct udphdr *pUdpheader;  
struct ipheader *pIpheader;  
struct tcpheader *pTcpheader;  
struct icmphdr *pIcmpheader;
```

```
char szSourceIP[MAX_ADDR_LEN], szDestIP[MAX_ADDR_LEN]; //源 IP 和目的 IP  
SOCKADDR_IN saSource, saDest;
```

```
pIpheader = (struct ipheader *)RecvBuf; //指针初始化
```

```
pTcpheader = (struct tcpheader *) (RecvBuf+ sizeof(struct ipheader ));  
pUdpheader = (struct udphdr *) (RecvBuf+ sizeof(struct ipheader ));  
pIcmpheader = (struct icmphdr *) (RecvBuf+ sizeof(struct ipheader ));
```

```
int iIphLen = sizeof(unsigned long) * ( pIpheader->h_lenver & 0x0f );
```

```
//开始抓包前的准备工作：
```

```
//while (1)
```

```
int cont;
```

```
for(cont=0;cont<50;cont++)
```

```
{
```

```
    //把 RecvBuf 清零
```

```
    memset(RecvBuf, 0, sizeof(RecvBuf));
```

```
    recv(sock, RecvBuf, sizeof(RecvBuf), 0);
```

```
//将按照网络字节顺序存储的源 IP 地址缀到 szSourceIP 后
```

```
saSource.sin_addr.s_addr = pIpheader->ip_src; //s_addr是按照网络字节顺序存储 IP 地址  
strncpy(szSourceIP, inet_ntoa(saSource.sin_addr), MAX_ADDR_LEN);
```



```

//将按照网络字节顺序存储的目的 IP 地址缀到 szDestIP 后
saDest.sin_addr.s_addr = pIpheader->ip_dst;
strncpy(szDestIP, inet_ntoa(saDest.sin_addr), MAX_ADDR_LEN);

//计算将网络字节转化为主机字节后的各个协议头部长度
lenip=ntohs(pIpheader->ip_len);
lentcp =(ntohs(pIpheader->ip_len)-(sizeof(struct ipheader)+sizeof(struct tcpheader)));
lenudp =(ntohs(pIpheader->ip_len)-(sizeof(struct ipheader)+sizeof(struct udphdr)));
lenicmp =(ntohs(pIpheader->ip_len)-(sizeof(struct ipheader)+sizeof(struct icmphdr)));

//确认 IP 头部的协议类型是 IPPROTO_TCP，且头部长度不为零，就可以开始抓包了！
if((pIpheader->ip_p)==IPPROTO_TCP&&lentcp!=0)
{

    pCount++; //计数：正在抓第几个数据包

    dataip=(unsigned char *) RecvBuf;
    datatcp=(unsigned char *) RecvBuf+sizeof(struct ipheader)+sizeof(struct tcpheader);
//data
    entity_content[65535]=*datatcp;

    //打印数据包字节数据：

    //之前已经通过套接字 sock 将这些头部与本地地址连接，获取了这些协议头部的信息，现在只需要打出来即可

    printf("\n#####数据包[%i]=%d 字节数据\n",pCount,lentcp);

    printf("\n*****IP 协议头部*****\n");

    printf("标识:%i\n",ntohs(pIpheader->ip_id));

    printf("总长度:%i\n",ntohs(pIpheader->ip_len));

    printf("偏移量:%i\n",ntohs(pIpheader->ip_off));

    printf("生存时间:%d\n",pIpheader->ip_ttl);

    printf("服务类型:%d\n",pIpheader->ip_tos);

```

```

printf("协议类型:%d\n",pIpheader->ip_p);

printf("检验和:%i\n",ntohs(pIpheader->ip_sum));

printf("源 IP 地址:%s ",szSourceIP);

printf("\n 目的 IP 地址:%s ",szDestIP);

printf("\n*****TCP 协议头部*****\n");

printf("源端口:%i\n",ntohs(pTcpheader->sport));

printf("目的端口:%i\n",ntohs(pTcpheader->dport));

printf("序列号:%i\n",ntohs(pTcpheader->th_seq));

printf("应答号:%i\n",ntohs(pTcpheader->th_ack));

printf("检验和:%i\n",ntohs(pTcpheader->th_sum));

printf("标志位 : ");

unsigned char FlagMask=1;
int t=0,p=0,i5=0;
int lenhttp=0;

for(k=0;k<6;k++) //打印标志位时，用了 FlagMask 类似掩码的作用
{
    if((pTcpheader->th_flag)&FlagMask)
        printf("%c",TcpFlag[k]);
    else
        printf(" ");
    FlagMask=FlagMask<<1;
}

}

}
return 0;
}

```

2013 年 12 月 1 日