

网络技术与应用课程报告

第二次实验报告

学号：21111033 姓名：艾明旭 年级：2021级 专业：信息安全

一、实验内容说明

IP数据报捕获与分析编程实验

要求如下：

1. 了解NPcap的架构
2. 学习NPcap的设备列表获取方法、网卡设备打开方法，以及数据包捕获方法
3. 通过NPcap编程，实现本机的IP数据报捕获，显示捕获数据帧的源MAC地址和目的MAC地址，以及类型/长度字段的值
4. 捕获的数据报不要求硬盘存储，但应以简单明了的方式在屏幕上显示。必显字段包括源MAC地址、目的MAC地址和类型/长度字段的值
5. 编写的程序应结构清晰，具有较好的可读性

二、前期准备

(1)NPcap架构

Npcap是致力于采用 Microsoft Light-Weight Filter (NDIS 6 LWF) 技术和 Windows Filtering Platform (NDIS 6 WFP) 技术对当前最流行的WinPcap工具包进行改进的一个项目。Npcap项目是最初2013年由 Nmap网络扫描器项目（创始人Gordon Lyon）和 北京大学罗杨博士发起，由 Google公司的Summer of Code计划赞助的一个开源项目，遵循MIT协议（与WinPcap一致）。

Npcap基于WinPcap 4.1.3源码基础上开发，支持32位和64位架构，在 Windows Vista 以上版本的系统中，采用NDIS 6技术的Npcap能够比原有的WinPcap数据包（NDIS 5）获得更好的抓包性能，并且稳定性更好。

NPcap提供两个不同的库：packet.dll与npcap.dll。第一个库提供一个底层的API，可用来直接访问驱动程序的函数，提供一个独立于微软的不同操作系统的编程接口。第二个库导出了更强大的、

更高层的捕获函数接口，并提供与UNIX捕获库libpcap的兼容性。这些函数使得数据包的捕获能独立于底层网络硬件与操作系统。

大多数网络应用程序通过被广泛使用的操作系统元件来访问网络，比如 sockets——这是一种简单的实现方式，因为操作系统已经妥善处理了底层具体实现细节（比如协议处理，封装数据包等工作），并且提供了一个与读写文件类似的，令人熟悉的接口；但是有些时候，这种“简单的实现方式”并不能满足需求，因为有些应用程序需要直接访问网络中的数据包：也就是说原始数据包——即没有被操作系统利用网络协议处理过的数据包。而 NpCap 则为 Win32 应用程序提供了这样的接口：

- 捕获原始数据包：无论它是发往某台机器的，还是在其他设备（共享媒介）上进行交换的
- 在数据包发送给某应用程序前，根据指定的规则过滤数据包
- 将原始数据包通过网络发送出去
- 收集并统计网络流量信息

(2)Npcap捕获数据包

设备列表获取方法——`pcap_findalldevs_ex`

NpCap 提供了 `pcap_findalldevs_ex` 和 `pcap_findalldevs` 函数来获取计算机上的网络接口设备的列表；此函数会为传入的 `pcap_if_t` 赋值——该类型是一个表示了设备列表的链表头；每一个这样的节点都包含了 `name` 和 `description` 域来描述设备。

除此之外，`pcap_if_t` 结构体还包含了一个 `pcap_addr` 结构体；后者包含了一个地址列表、一个掩码列表、一个广播地址列表和一个目的地址的列表；此外，`pcap_findalldevs_ex` 还能返回远程适配器信息和一个位于所给的本地文件夹的 `pcap` 文件列表。

网卡设备打开方法——`pcap_open`

用来打开一个适配器，实际调用的是 `pcap_open_live`；它接受五个参数：

- `name`：适配器的名称（GUID）
- `snaplen`：制定要捕获数据包中的哪些部分。在一些操作系统中（比如 xBSD 和 Win32），驱动可以被配置成只捕获数据包的初始化部分：这样可以减少应用程序间复制数据的量，从而提高捕获效率；本次实验中，将值定为 65535，比能遇到的最大的 MTU 还要大，因此总能收到完整的数据包。
- `flags`：主要的意义是其中包含的混杂模式开关；一般情况下，适配器只接收发给它自己的数据包，而那些在其他机器之间通讯的数据包，将会被丢弃。但混杂模式将会捕获所有的数据包——因为我们需要捕获其他适配器的数据包，所以需要打开这个开关。
- `to_ms`：指定读取数据的超时时间，以毫秒计；在适配器上使用其他 API 进行读取操作的时候，这些函数会在这里设定的时间内响应——即使没有数据包或者捕获失败了；在统计模式

下，to_ms 还可以用来定义统计的时间间隔：设置为 0 说明没有超时——如果没有数据包到达，则永远不返回；对应的还有 -1：读操作立刻返回。

- errbuf：用于存储错误信息字符串的缓冲区

该函数返回一个 pcap_t 类型的 handle。

数据包捕获方法——pcap_loop

虽然在课本上演示用的是 pcap_next_ex 函数，但是他并不会使用回调函数，不会把数据包传递给应用程序，所以在本次实验中我采取的是 pcap_loop 函数。

API 函数 pcap_loop 和 pcap_dispatch 都用来在打开的适配器中捕获数据包；但是前者会已知捕获直到捕获到的数据包数量达到要求数量，而后者在到达了前面 API 设定的超时时间之后就会返回（尽管这得不到保证）；前者会在一小段时间内阻塞网络的应用，故一般项目都会使用后者作为读取数据包的函数；虽然在本次实验中，使用前者就够了。

这两个函数都有一个回调函数；这个回调函数会在这两个函数捕获到数据包的时候被调用，用来处理捕获到的数据包；这个回调函数需要遵从特定的格式。但是需要注意的是我们无法发现 CRC 冗余校验码——因为帧到达适配器之后，会经过校验确认的过程；这个过程成功，则适配器会删除 CRC；否则，大多数适配器会删除整个包，因此无法被 Npcap 确认到。

三、实验过程

(1)项目设计思路

本次实验还是非常友善的，首先在实验指导书上，已经把大体思路都设计出来了，需要我们完成的是把这些部分组合起来，并实现我们想要实现的功能，即输出显示捕获数据帧的源MAC地址和目的MAC地址，以及类型/长度字段的值。

所以按照课本上的要求，依次完成：

1. 获取设备列表
2. 打开想要嗅探的设备
3. 捕获数据包

需要注意的是我们需要按照书上的介绍，将捕获的结构体强制转化成我们所需要的格式——即标准数据报所具有的格式，因为其是按字节划分的，所以需要用到pack()函数，打包过程如下代码：

```

typedef struct IPheader_t {          //IP首部
    u_int SrcIP;//源IP
    u_int DstIP;//目的IP
    WORD TotalLen;//总长度
    WORD ID;//标识
    WORD Flag_Segment;//标志 片偏移
    WORD Checksum;//头部校验和
    BYTE TTL;//生存周期
    BYTE Protocol;//协议
    BYTE Ver_HLen;//IP协议版本和IP首部长度：高4位为版本，低4位为首部的长度
    BYTE TOS;//服务类型

}IPheader_t;

typedef struct FrameHeader_t {      //帧首部
    BYTE DesMAC[6];//目的地址
    BYTE SrcMAC[6];//源地址
    WORD FrameType;//帧类型
}FrameHeader_t;

typedef struct Data_t {            //数据包
    FrameHeader_t FrameHeader;
    IPheader_t IPheader;
}Data_t;

```

捕获完数据包后就可以根据以上结构体，输出我们想要的数​​据，比如 `IPPacket-`

`>FrameHeader.SrcMAC`、`IPPacket->FrameHeader.DesMAC`、`IPPacket->FrameHeader.FrameType` 等，但一定要注意格式。

以上即为主要思路，接下来进行关键代码分析。

(2)关键代码pcap类型的分析

按照项目设计思路开始编写代码，一开始需要引入头函数 `<pcap.h>` ,然后是获取设备列表，先定义接口指针以及将来会用到选设备时候的int型变量和一个错误信息缓冲区，然后就可以利用 `pcap_findalldevs_ex` 函数来获取计算机上的网络接口设备的列表，如果返回值为-1——即出现异常的话，则会显示异常信息并结束进程，具体代码如下：

```

    pcap_if_t* alldevs;//指向设备链表首部的指针
pcap_if_t* d;
char errbuf[PCAP_ERRBUF_SIZE];    //错误信息缓冲区
int num = 0;//接口数量
int n;
int read_count;    //获得本机的设备列表
int s = 0;
if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING,    //获取本机的接口设备
    NULL,    //无需认证
    &alldevs,    //指向设备列表首部
    errbuf    //出错信息保存缓存区
) == -1)

```

接下来是由用户选择想要嗅探的设备，首先将刚刚捕获到的接口设备信息打印在进程中，然后由用户输入想要嗅探的接口并对其选择的数字做合法性检测，并跳转到此设备出进行数据包的嗅探，如果嗅探成功则开始返回手动设置输出的信息，如果失败则会显示错误信息并结束进程，具体代码如下所示：

字

```
for (d = alldevs; d != NULL; d = d->next)
{
    num++;
    cout << dec << num << ":" << d->name << endl; //利用d->name获取该网络接口设备的名称

    if (d->description == NULL) //利用d->description获取该网络接口设备的描述信息
    {
        cout << "无信息" << endl;
    }
    else
    {
        cout << d->description << endl;
    }
}
if (num == 0)
{
    cout << "无可用的接口" << endl;
    return 0;
}
{
    cout << "请输入要打开的网络接口号" << " (1~" << num << ") : " << endl;
    cin >> n;
    num = 0;
    for (d = alldevs; num < (n - 1); num++)
    {
        d = d->next;
    } //跳转到选中的网络接口号
    pcap_t* adhandle;
    adhandle = pcap_open(d->name, //设备名
        65536, //要捕获的数据包的部分
        PCAP_OPENFLAG_PROMISCUOUS, //混杂模式
        1000, //超时时间
        NULL, //远程机器验证
        errbuf //错误缓冲池
    );
    if (adhandle == NULL)
    {
        cout << "错误，无法打开" << endl;
        pcap_freealldevs(alldevs);
        return 0;
    }
    else
    {
        cout << "监听: " << d->description << endl;
        pcap_freealldevs(alldevs);
    }
}
if (s == 0)
{
    cout << "要捕获的数据包的个数: " << endl;
    cin >> read_count;
```

```
        if (read_count == 0)
        {
            pcap_close(adhandle);
        }
        else
        {
            pcap_loop(adhandle, read_count, (pcap_handler)PacketHandle, NULL);
            read_count = 0;
        }
    }
}
return 0;
}
```

最后是捕获数据包，本次实验用的是pcap_loop函数来捕获并产生回调信息，在这个函数中会调用另一个函数并一直循环，这也是它叫做loop的原因，相当于开启了又一个线程，用其开启自定义的packet_handler函数，这个函数的参数本人是仿照标准获取信息的参数来设置的，并在函数体中首先对时间输出标准化，输出时间、长度的相关信息，接下来对源MAC地址和目的MAC地址，以及类型/长度字段进行输出，在输出源MAC地址和目的MAC地址的时候需要使用%02x获得统一格式的输出生，在输出类型的时候要对捕获到的具体数值做出判断，通过switch来确定其究竟是哪一种类型，并输出相应字段的值，实现的具体代码如下：

```

void PacketHandle(u_char* argument, const struct pcap_pkthdr* pkt_head, const u_char*
pkt_data)
{
    FrameHeader_t* ethernet_protocol;          //以太网协议
    u_short ethernet_type;                     //以太网类型
    u_char* mac_string;                       //以太网地址
    //获取以太网数据内容
    ethernet_protocol = (FrameHeader_t*)pkt_data;
    ethernet_type = ntohs(ethernet_protocol->FrameType);
    printf("以太网类型为 : \t");
    printf("%04x\n", ethernet_type);
    switch (ethernet_type)
    {
    case 0x0800:
        printf("网络层IPv4协议\n");
        break;
    case 0x0806:
        printf("网络层ARP协议\n");
        break;
    case 0x8035:
        printf("网络层RARP协议\n");
        break;
    default:
        printf("网络层协议未知\n");
        break;
    }
    mac_string = ethernet_protocol->SrcMAC;
    printf("Mac源地址: \n");
    printf("%02x:%02x:%02x:%02x:%02x:%02x:\n",
        *mac_string,
        *(mac_string + 1),
        *(mac_string + 2),
        *(mac_string + 3),
        *(mac_string + 4),
        *(mac_string + 5)
    );
    mac_string = ethernet_protocol->DesMAC;
    printf("Mac目的地址: \n");
    printf("%02x:%02x:%02x:%02x:%02x:%02x:\n",
        *mac_string,
        *(mac_string + 1),
        *(mac_string + 2),
        *(mac_string + 3),
        *(mac_string + 4),
        *(mac_string + 5)
    );
    if (ethernet_type == 0x0800)
    {
        IP_Packet_Handle(pkt_head, pkt_data);
    }
}

```



```

    }
}

```

IP地址的捕获，这里需要我们对上述的函数做简单的修改。

```

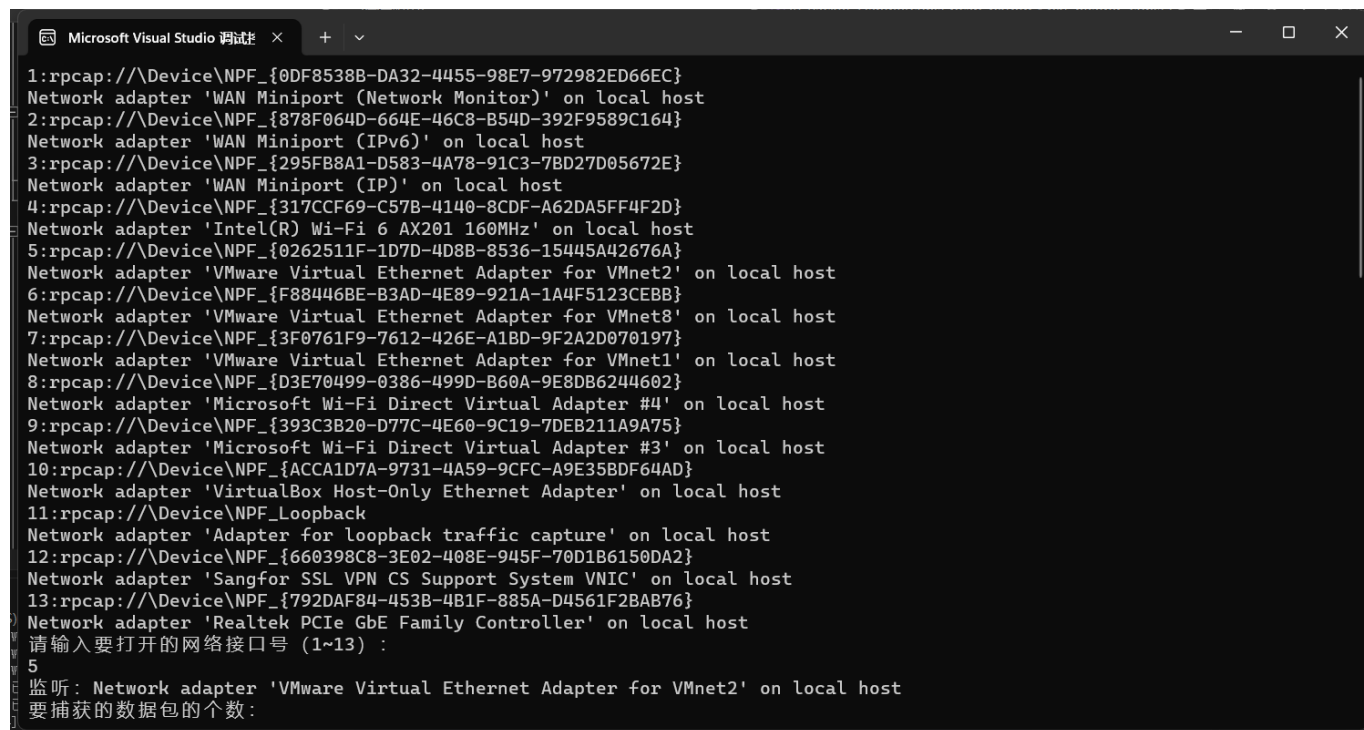
void IP_Packet_Handle(const struct pcap_pkthdr* pkt_header, const u_char* pkt_data)
{
    IPheader_t* IPheader;
    IPheader = (IPheader_t*)(pkt_data + 14); // IP包的内容在原有物理帧后14字节开始
    sockaddr_in source, dest;
    char sourceIP[16], destIP[16];
    source.sin_addr.s_addr = IPheader->SrcIP;
    dest.sin_addr.s_addr = IPheader->DstIP;
    strncpy(sourceIP, inet_ntoa(source.sin_addr), 16);
    strncpy(destIP, inet_ntoa(dest.sin_addr), 16);
    printf("版本: %d\n", IPheader->Ver_HLen >> 4);
    printf("IP协议首部长度: %d Bytes\n", (IPheader->Ver_HLen & 0x0f) * 4);
    printf("服务类型: %d\n", IPheader->TOS);
    printf("总长度: %d\n", ntohs(IPheader->TotalLen));
    printf("标识: 0x%.4x (%i)\n", ntohs(IPheader->ID));
    printf("标志: %d\n", ntohs(IPheader->Flag_Segment));
    printf("片偏移: %d\n", (IPheader->Flag_Segment) & 0x8000 >> 15);
    printf("生存时间: %d\n", IPheader->TTL);
    printf("协议号: %d\n", IPheader->Protocol);
    printf("协议种类: ");
    switch (IPheader->Protocol)
    {
    case 1:
        printf("ICMP\n");
        break;
    case 2:
        printf("IGMP\n");
        break;
    case 6:
        printf("TCP\n");
        break;
    case 17:
        printf("UDP\n");
        break;
    default:
        break;
    }
    printf("首部检验和: 0x%.4x\n", ntohs(IPheader->Checksum));
    printf("源地址: %s\n", sourceIP);
    printf("目的地址: %s\n", destIP);
    cout << "-----" << endl;
}

```

最后需要释放设备列表，即可退出进程。

(3)结果展示

首先是捕获到设备接口，打印设备列表的界面：



```
Microsoft Visual Studio 调试  x + v
1:rpcap://\Device\NPF_{0DF8538B-DA32-4455-98E7-972982ED66EC}
Network adapter 'WAN Miniport (Network Monitor)' on local host
2:rpcap://\Device\NPF_{878F064D-664E-46C8-B54D-392F9589C164}
Network adapter 'WAN Miniport (IPv6)' on local host
3:rpcap://\Device\NPF_{295FB8A1-D583-4A78-91C3-7BD27D05672E}
Network adapter 'WAN Miniport (IP)' on local host
4:rpcap://\Device\NPF_{317CCF69-C57B-4140-8CDF-A62DA5FF4F2D}
Network adapter 'Intel(R) Wi-Fi 6 AX201 160MHz' on local host
5:rpcap://\Device\NPF_{0262511F-1D7D-4D8B-8536-15445A42676A}
Network adapter 'VMware Virtual Ethernet Adapter for VMnet2' on local host
6:rpcap://\Device\NPF_{F88446BE-B3AD-4E89-921A-1A4F5123CEBB}
Network adapter 'VMware Virtual Ethernet Adapter for VMnet8' on local host
7:rpcap://\Device\NPF_{3F0761F9-7612-426E-A1BD-9F2A2D070197}
Network adapter 'VMware Virtual Ethernet Adapter for VMnet1' on local host
8:rpcap://\Device\NPF_{D3E70499-0386-499D-B60A-9E8DB6244602}
Network adapter 'Microsoft Wi-Fi Direct Virtual Adapter #4' on local host
9:rpcap://\Device\NPF_{393C3B20-D77C-4E60-9C19-7DEB211A9A75}
Network adapter 'Microsoft Wi-Fi Direct Virtual Adapter #3' on local host
10:rpcap://\Device\NPF_{ACCA1D7A-9731-4A59-9CFC-A9E35BDF64AD}
Network adapter 'VirtualBox Host-Only Ethernet Adapter' on local host
11:rpcap://\Device\NPF_{Loopback}
Network adapter 'Adapter for loopback traffic capture' on local host
12:rpcap://\Device\NPF_{660398C8-3E02-408E-945F-70D1B6150DA2}
Network adapter 'Sangfor SSL VPN CS Support System VNIC' on local host
13:rpcap://\Device\NPF_{792DAF84-453B-4B1F-885A-D4561F2BAB76}
Network adapter 'Realtek PCIe GbE Family Controller' on local host
请输入要打开的网络接口号 (1~13) :
5
监听: Network adapter 'VMware Virtual Ethernet Adapter for VMnet2' on local host
要捕获的数据包的个数:
```

然后是输出数据报信息的画面：

```
Microsoft Visual Studio 调试 × + -
3
以太网类型为 : 0800
网络层IPv4协议
Mac源地址 :
00:50:56:c0:00:02:
Mac目的地址 :
01:00:5e:7f:ff:fa:
版本 : 15
IP协议首部长度 : 60 Bytes
服务类型 : 250
总长度 : 273
标识 : 0x29ff (-641)
标志 : 49320
片偏移 : 0
生存时间 : 239
协议号 : 255
协议种类 : 首部检验和 : 0x1301
源地址 : 69.0.0.203
目的地址 : 203.127.0.0
-----
以太网类型为 : 0800
网络层IPv4协议
Mac源地址 :
00:50:56:c0:00:02:
Mac目的地址 :
01:00:5e:7f:ff:fa:
版本 : 15
IP协议首部长度 : 60 Bytes
服务类型 : 250
总长度 : 273
```

```
Microsoft Visual Studio 调试 × + -
标识 : 0x29fe (-641)
标志 : 49320
片偏移 : 0
生存时间 : 239
协议号 : 255
协议种类 : 首部检验和 : 0x1301
源地址 : 69.0.0.203
目的地址 : 203.128.0.0
-----
以太网类型为 : 0800
网络层IPv4协议
Mac源地址 :
00:50:56:c0:00:02:
Mac目的地址 :
01:00:5e:7f:ff:fa:
版本 : 15
IP协议首部长度 : 60 Bytes
服务类型 : 250
总长度 : 273
标识 : 0x29fd (-641)
标志 : 49320
片偏移 : 0
生存时间 : 239
协议号 : 255
协议种类 : 首部检验和 : 0x1301
源地址 : 69.0.0.203
目的地址 : 203.129.0.0
-----
D:\dasanshang\wangji\shiyancehngxu\2.1\x64\Debug\2.1.exe (进程 16204)已退出, 代码为 0。
```

四、特殊情况的分析与处理

本次实验中确实遇到了一个难点，但并不是在网络的获取设备列表或者是嗅探数据报的层面，而是对各种东西进行输出的时候不知道怎么输出。

由于之前我们使用C++的时候经常使用的诗句类型不在本次实验可以输出的范围之内，因此我们需要进行输出的数据类型就会十分复杂，最终我们选择了合适的方法，成功的将word,byte等数据

输出，避免了汇编与C++转换之间的問題。

查閱資料，但類似強制類型轉換的方法，未能成功，後來改用%x來格式化後，隨即改變程序，得到了看起來有模樣的數據，但是這樣子打印的話會使得輸出的位數不全相同，因為有些小於0x10的十六進制數輸出就會只輸出一位，所以將%x換成%02x就可以獲得統一格式的輸出。

本次還遇到的一個問題就是在進行抓包的時候，許多網絡的抓包十分困難，抓包結果很慢甚至一直無法成功輸出，猜測是端口的问题，但是未能得到很好的驗證。

五、總結

本次實驗是網絡技術與應用的第二次實驗，一開始由於對於網絡服務器的編程完全不了解，對於捕獲數據包方面的知識更是只有工具操作，沒有實際原理的認識，甚至基本上沒有用過wireshark軟件，但是通過動手編程後，不僅對數據報的架構、npcap/winpcap的架構、捕獲數據包的細節等方面有了更深層的認識，在網絡方面的認知也更加豐富。