# PROJ 1 l2tp协议分析实验与设计

**一、实验要求**

基于c++设计并实现一个简单系统，实现

1. 基于windows 7或linux抓取l2tp协议流量
2. 针对每一个目标用户，实时监控 l2tp请求，并分析、还原、呈现其连接和业务载荷
3. 支持对特定目标的静载荷替换

构建一个demo：用笔记本电脑实现对手机上网对象的以上功能

**二、实验原理**

**实验对象---L2tp协议简介**

L2TP（Layer 2 Tunneling Protocol，二层隧道协议）是 VPDN（Virtual Private Dial-up Network，虚拟私有拨号网）隧道协议的一种。VPDN 是指利用公共网络（如 ISDN 或 PSTN）的拨号功能接入公共网络，实现虚拟专用网，从而为企业、小型 ISP、移动办公人员等提供接入服务。

**协议背景**

PPP 定义了一种封装技术，可以在二层的点到点链路上传输多种协议数据包，当用户与 NAS 之间运行 PPP 协议时，二层链路端点与 PPP 会话点驻留在相同硬件设备上。

L2TP（RFC 2661）是一种对 PPP 链路层数据包进行隧道传输的技术，允许二层链路端点（LAC）和 PPP 会话点（LNS）驻留在通过分组交换网络连接的不同设备上，从而扩展了 PPP 模型，使得PPP 会话可以跨越帧中继或 Internet 等网络。

L2TP 结合了 L2F 和 PPTP 的各自优点，成为 IETF 有关二层隧道协议的工业标准

**L2TP 协议结构**

图1描述了控制通道以及PPP帧和数据通道之间的关系。PPP帧在不可靠的L2TP数据通道上进行传输，控制消息在可靠的L2TP控制通道内传输。

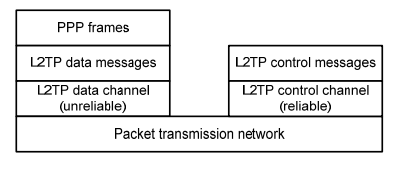


图1 L2TP协议结构

图2描述了LAC与LNS之间的L2TP数据报文的封装结构。通常L2TP数据以UDP报文的形式发送。L2TP注册了UDP 1701 端口，但是这个端口仅用于初始的隧道建立过程中。L2TP隧道发起方任选一个空闲的端口（未必是 1701）向接收方的 1701 端口发送报文；接收方收到报文后，也任选一个空闲的端口（未必是 1701），给发送方的指定端口回送报文。至此，双方的端口选定，并在隧道保持连通的时间段内不再改变。

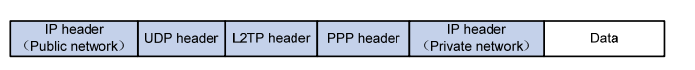


图2 L2TP报文封装结构图

**隧道和会话**

在一个 LNS 和 LAC 对之间存在着两种类型的连接。

● 隧道（Tunnel）连接：它对应了一个 LNS 和 LAC 对。

● 会话（Session）连接：它复用在隧道连接之上，用于表示承载在隧道连接中的每个 PPP 会话过程。

在同一对 LAC 和 LNS 之间可以建立多个 L2TP 隧道，隧道由一个控制连接和一个或多个会话连接组成。会话连接必须在隧道建立（包括身份保护、L2TP 版本、帧类型、硬件传输类型等信息的交换）成功之后进行，每个会话连接对应于 LAC 和 LNS 之间的一个 PPP 数据流。

控制消息和 PPP 数据报文都在隧道上传输。L2TP 使用 Hello 报文来检测隧道的连通性。LAC 和 LNS定时向对端发送 Hello 报文，若在一段时间内未收到 Hello 报文的应答，隧道断开。

**控制消息和数据消息**

L2TP 中存在两种消息：控制消息和数据消息。

● 控制消息用于隧道和会话连接的建立、维护以及传输控制。它的传输是可靠传输，并且支持对控制消息的流量控制和拥塞控制。

● 数据消息用于封装 PPP 帧，并在隧道上传输。它的传输是不可靠传输，若数据报文丢失，不予重传，不支持对数据消息的流量控制和拥塞控制。

控制消息和数据消息共享相同的报文头。L2TP 报文头中包含隧道标识符（Tunnel ID）和会话标识符（Session ID）信息，用来标识不同的隧道和会话。隧道标识相同、会话标识不同的报文将被复用在一个隧道上。报文头中的隧道标识符与会话标识符由对端分配。

**实验工具---WinPcap介绍**

WinPcap是一个基于Win32平台的，用于捕获网络数据包并进行分析的开源库.

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，封装数据包等等），并且提供了一个与读写文件类似的，令人熟悉的接口。

然而，有些时候，这种“简单的方式”并不能满足任务的需求，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，那些应用程序需要访问原始数据包，即没有被操作系统利用网络协议处理过的数据包。

WinPcap产生的目的，就是为Win32应用程序提供这种访问方式； WinPcap提供了以下功能:

● 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，根据用户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

以上这些功能需要借助安装在Win32内核中的网络设备驱动程序才能实现，再加上几个动态链接库DLL。所有这些功能都能通过一个强大的编程接口来表现出来,易于开发，并能在不同的操作系统上使用。

1. **环境搭建**

**四、实验过程**

**1.获取设备列表并打印信息**

WinPcap提供了 pcap\_findalldevs\_ex() 函数来实现获取设备列表的功能: 这个函数返回一个 pcap\_if 结构的链表, 每个这样的结构都包含了一个适配器的详细信息。数据域 name 和 description 表示一个适配器名称和一个可以让人们理解的描述。

下列代码能获取适配器列表，并在屏幕上显示出来，如果没有找到适配器，将打印错误信息。

pcap\_if\_t\* alldevs;

pcap\_if\_t\* d;

pcap\_t\* adhandle;

char errbuf[PCAP\_ERRBUF\_SIZE];

char packet\_filter[] = "ether proto 0x0800 or ether proto 0x86DD";

struct bpf\_program fcode;

u\_int netmask = 0;

/\* 获取本机设备列表 \*/

if (pcap\_findalldevs\_ex(PCAP\_SRC\_IF\_STRING, NULL, &alldevs, errbuf) == -1)

{

fprintf(stderr, "Error in pcap\_findalldevs: %s\n", errbuf);

exit(1);

}

/\* 打印列表 \*/

for (d = alldevs; d; d = d->next)

{

printf("%d. %s", ++i, d->name);

if (d->description)

printf(" (%s)\n", d->description);

else

printf(" (No description available)\n");

}

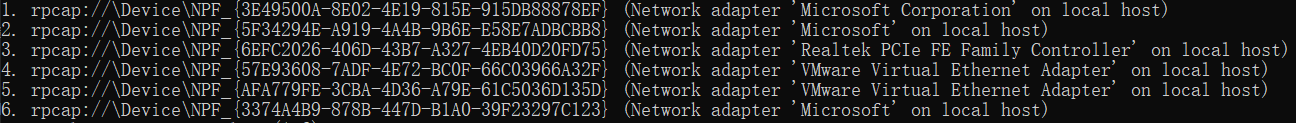
if (i == 0)

{

printf("\nNo interfaces found! Make sure WinPcap is installed.\n");

return -1;

}



**2.设置过滤器**

WinPcap最强大的特性之一，是拥有过滤数据包的引擎。 它提供了有效的方法去获取网络中的某些数据包，这也是WinPcap捕获机制中的一个组成部分。 用来过滤数据包的函数是 pcap\_compile() 和 pcap\_setfilter().

pcap\_compile() 将一个高层的布尔过滤表达式编译成一个能够被过滤引擎所解释的低层的字节码。这里布尔过滤表达式的语法是基于柏克莱封包过滤器（Berkeley Packet Filter，缩写 BPF）的。BPF的过滤功能是以BPF虚拟机机器语言的解释器的形式实现的，这种语言的程序可以抓取封包数据，对封包中的数据采取算术操作，并将结果与常量或封包中的数据或结果中的测试位比较，根据比较的结果决定接受还是拒绝封包。

pcap\_setfilter() 将一个过滤器与内核捕获会话向关联。当 pcap\_setfilter() 被调用时，这个过滤器将被应用到来自网络的所有数据包，并且，所有的符合要求的数据包 (即那些经过过滤器以后，布尔表达式为真的包) ，将会立即复制给应用程序。

以下代码展示了如何编译并设置过滤l2tp协议流量的过滤器。

// L2TP over udp port 1701

char filter\_exp[] = "udp port 1701";

// add filter of udp:1701

if (pcap\_compile(handle, &filter, filter\_exp, 0, ip) == PCAP\_ERROR) {

printf("Bad filter - %s\n", pcap\_geterr(handle));

return CONPILE\_ERROR;

}

if (pcap\_setfilter(handle, &filter) == PCAP\_ERROR) {

printf("Error setting filter - %s\n", pcap\_geterr(handle));

return SET\_FILTER\_ERROR;

}

在这段代码片断中，传递给 pcap\_compile() 的过滤器是" udp port 1701"，这是因为l2tp的协议流量都是通过udp协议在端口1701间传送的。

**打开设备，开始监听流量**

打开设备的函数是 pcap\_open()。

参数 snaplen 制定要捕获数据包中的哪些部分。本例中，我们将值定为65535，它比我们能遇到的最大的MTU还要大。因此，我们确信我们总能收到完整的数据包。

参数flag用来指示适配器是否要被设置成混杂模式。 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。 相反，如果适配器是混杂模式，那么不管这个数据包是不是发给我的，我都会去捕获。也就是说，我会去捕获所有的数据包。这意味着在一个共享媒介(比如总线型以太网)，WinPcap能捕获其他主机的所有的数据包。因此本例中必须将适配器设置成混杂模式。

参数to\_ms 指定读取数据的超时时间，以毫秒计(1s=1000ms)。在适配器上进行读取操作都会在 to\_ms 毫秒时间内响应，即使在网络上没有可用的数据包。将 to\_ms 设置为0意味着没有超时，那么如果没有数据包到达的话，读操作将永远不会返回。 如果设置成-1，则情况恰好相反，无论有没有数据包到达，读操作都会立即返回。

当适配器被打开，捕获工作就可以用 pcap\_dispatch() 或 pcap\_loop() 进行。 这两个函数非常的相似，区别就是 pcap\_ dispatch() 当超时时间到了(timeout expires)就返回 (尽管不能保证) ，而 pcap\_loop() 不会因此而返回，只有当 cnt 数据包被捕获，所以，pcap\_loop()会在一小段时间内，阻塞网络的利用。 pcap\_loop() 对于我们这个demo来说，可以满足需求。

下面的代码展示了如何打来一个适配器并开始抓取数据包。

if (i == 0)

{

printf("\nNo interfaces found! Make sure WinPcap is installed.\n");

return -1;

}

printf("Enter the interface number (1-%d):", i);

scanf("%d", &inum);

if (inum < 1 || inum > i)

{

printf("\nInterface number out of range.\n");

/\* 释放设备列表 \*/

pcap\_freealldevs(alldevs);

return -1;

}

/\* 跳转到选中的适配器 \*/

for (d = alldevs, i = 0; i < inum - 1; d = d->next, i++);

/\* 打开设备 \*/

if ((adhandle = pcap\_open(d->name, // 设备名

65536, // 65535保证能捕获到不同数据链路层上的每个数据包的全部内容

PCAP\_OPENFLAG\_PROMISCUOUS, // 混杂模式

1000, // 读取超时时间

NULL, // 远程机器验证

errbuf // 错误缓冲池

)) == NULL)

{

fprintf(stderr, "\nUnable to open the adapter. %s is not supported by WinPcap\n", d->name);

/\* 释放设备列表 \*/

pcap\_freealldevs(alldevs);

return -1;

}

/\* 释放设备列表 \*/

pcap\_freealldevs(alldevs);

/\* 开始捕获 \*/

pcap\_loop(adhandle, 0, my\_protocol\_packet\_callback, NULL);

**4.对捕获到的数据包进行分析、还原、呈现**

pcap\_loop都有一个 callback 参数, packet\_handler, 指向一个可以接收数据包的函数。这个函数会在收到每个新的数据包并收到一个通用状态时被libpcap所调用。packet\_handler有三个参数，分别如下：

● \_char \*param：数据包存储的文件指针

● struct pcap\_pkthdr \* header： 并非是数据包的指针，只是与数据包捕获驱动有关的一个header ，是堆文件包的结构体首部指针。可以得到时间值，数据包长度。

● const\_char \* pkt\_data：指向数据包内容的指针，包括了协议头，可以经过计算获得IP数据包头部的位置， UDP首部的位置等。

我们需要定义自己的packet\_handler函数，对抓到的数据包进行层层解包，还原并呈现其中的信息。

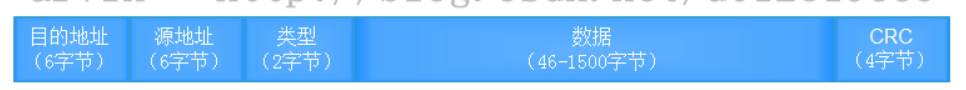
**4.1 解析以太协议头**

以太网目的地址和源地址各占6个字节，该地址即平常所说的网卡的MAC。

在地址后面有两个字节表示类型，如：0800表示此真数据是IP数据，0806表示ARP请求。

类型字段之后是数据，对于以太网，数据段大小为46-1500字节，不足46字节的数据将被自动补足到46字节。

CRC字段用于对帧内数据进行校验，保证数据传输的正确性，通常是硬件实现的。



图三 以太协议结构图

下面的代码定义了以太协议头结构体和相关的解析操作

/\*下边是以太网头的协议格式 \*/

struct ethernet\_header

{

u\_int8\_t ether\_dhost[6]; /\*目的以太地址\*/

u\_int8\_t ether\_shost[6]; /\*源以太网地址\*/

u\_int16\_t ether\_type; /\*以太网类型\*/

};

//解析以太网协议头

u\_short ethernet\_type; /\*以太网协议类型\*/

struct ethernet\_header\* ethernet\_protocol; /\*以太网协议变量\*/

const int ether\_header\_length = 14;

ethernet\_protocol = (struct ethernet\_header\*)packet\_content; /\*获得一太网协议数据内容\*/

ethernet\_type = ntohs(ethernet\_protocol->ether\_type); /\*获得以太网类型\*/

if (ethernet\_type != 0x0800) {

printf("上层不是IPV4协议，停止解析\n");

return ;

}

**4.2 解析IPV4协议头**

下面是IPV4协议头格式及其相关字段的解释



图4 IPV4协议头格式

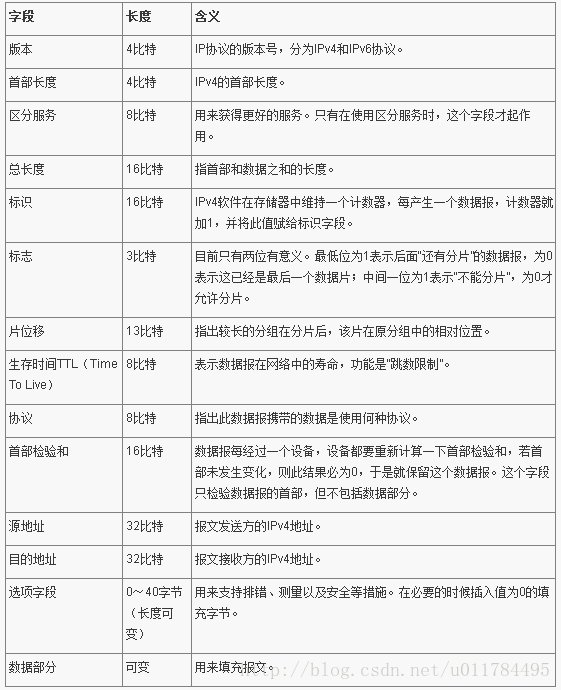


图5 IPV4协议各字段解释

知道了IPV4协议头的格式之后，我们就可以逐字段解析相关信息了。

/\*ip地址格式\*/

typedef u\_int32\_t in\_addr\_t;

/\*IPV4头结构体定义\*/

struct ipv4\_header

{

#ifdef WORKS\_BIGENDIAN

u\_int8\_t ip\_version : 4, /\*version:4\*/

ip\_header\_length : 4; /\*IP协议首部长度Header Length\*/

#else

u\_int8\_t ip\_header\_length : 4,

ip\_version : 4;

#endif

u\_int8\_t ip\_tos; /\*服务类型Differentiated Services Field\*/

u\_int16\_t ip\_length; /\*总长度Total Length\*/

u\_int16\_t ip\_id; /\*标识identification\*/

u\_int16\_t ip\_off; /\*片偏移\*/

u\_int8\_t ip\_ttl; /\*生存时间Time To Live\*/

u\_int8\_t ip\_protocol; /\*协议类型（TCP或者UDP协议）\*/

u\_int16\_t ip\_checksum; /\*首部检验和\*/

struct in\_addr ip\_source\_address; /\*源IP\*/

struct in\_addr ip\_destination\_address; /\*目的IP\*/

};

//解析IPV4协议头

struct ipv4\_header\* ip\_protocol; /\*ip协议变量\*/

ip\_protocol = (struct ipv4\_header\*)(packet\_content + ether\_header\_length); /\*去掉以太头部获得ip数据包的内容\*/

ip\_header\_length = ip\_protocol->ip\_header\_length \* 4; /\*获得长度\*/

if (ip\_protocol->ip\_protocol != 17) {

printf("上层不是UDP停止解析\n");

return;

}

**4.3 解析udp协议头**

udp头部结构中包含以下各部分

●16位源端口号 记录源端口号，在需要对方回信时选用。不需要时可用全0。

●16位目的端口号 记录目标端口号。这在终点交付报文时必须要使用到。

●长度 UDP数据报的长度（包括数据和首部），其最小值为8B（即仅有首部没有数据的情况）。

●校验和 检测UDP数据报在传输中是否有错，有错就丢弃。该字段时可选的，当源主机不想计算校验和，则直接令该字段为全0。当传输层从IP层收到UDP数据报时，就根据首部中的目的端口，把UDP数据报通过相应的端口，上交给进程。如果接收方UDP发现收到的报文中目的端口号不正确（即不存在对应端口号的应用进程），就丢弃该报文，并由ICMP发送“端口不可达”差错报文交给发送方。

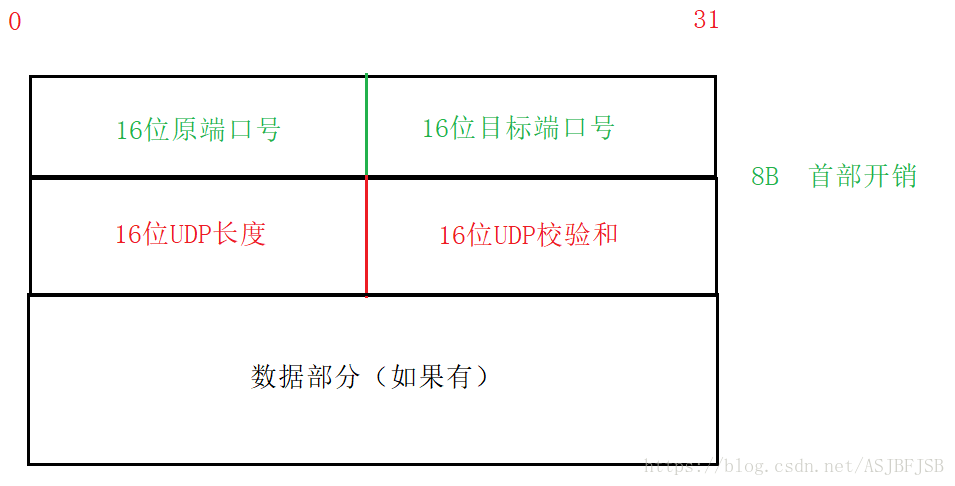


图6 UDP协议格式图

下面是关于udp协议头结构体的定义和解析代码：

/\*关于udp头部的定义\*/

struct udp\_header

{

u\_short uh\_sport;

u\_short uh\_dport;

u\_short uh\_ulen;

u\_short uh\_sum;

};

//解析UDP协议头

struct udp\_header\* udp\_protocol;

u\_short source\_port;

u\_short destination\_port;

u\_short length;

const int udp\_header\_length = 8;

/\* 获取UPD协议数据 \*/

udp\_protocol = (struct udp\_header\*)(packet\_content+ether\_header\_length+ ip\_header\_length);

/\* 获取源端口 \*/

source\_port = ntohs(udp\_protocol->uh\_sport);

/\* 获取目的端口 \*/

destination\_port = ntohs(udp\_protocol->uh\_dport);

/\* 获取长度 \*/

length = ntohs(udp\_protocol->uh\_ulen);

**4.4 解析L2TP协议头**

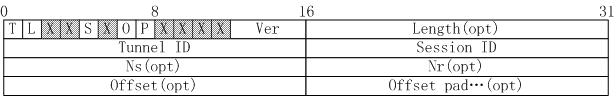


图7 L2TP协议头格式图

下面是对l2TP头各字段的介绍

● Type(T)：标识消息的类型，0表示是数据消息，1表示控制消息。

● Length(L)：置1时，说明Length域的值是存在的，对于控制消息L位必须置1。

● X bit：保留位，所有保留位均置0。

● Sequence(S)：置1时，说明Ns和Nr是存在的，对于控制消息S必须置1。

● Offset(O)：置1时，说明Offset Size域是存在的，对于控制消息O必须置0。

● Priority(P)：只用于数据消息，对于控制消息P位置0，当数据消息此位置1时，说明该消息在本列队和传输时应得到优先处理。

● Ver：必须是2，表示L2TP数据报头的版本。

● Length：标识整个报文的长度(以字节为单位)。

● Tunnel ID：标识L2TP控制链接，L2TP Tunnel标识符只有本地意义，一个Tunnel两端被分配的Tunnel ID可能会不同，报头中的Tunnel是指接收方的Tunnel ID，而不是发送方的。本端的Tunnel ID在创建Tunnel时分配。通过Tunnel ID AVPs和对端交换Tunnel ID信息。

● Session ID：标识Tunnel中的一个session，只有本地意义，一个session两端Session ID可能不同。

● Ns：标识发送数据或控制消息的序号，从0开始，以1递增，到216再从0开始。

● Nr：标识下一个期望接收到的控制消息。Nr的值设置成上一个接收到的控制消息的Ns+1。这样是对上一个接收到的控制消息的确认。数据消息忽略Nr。

● Offset Size：如果值存在的话，标识有效载荷数据的偏移。

关于l2tp头结构体的定义和解析代码如下

// L2TP头部结构

struct l2tp\_header {

Int l2tp\_type;

int l2tp\_len\_field;

int l2tp\_sequence\_field;

int l2tp\_offset\_field;

int l2tp\_priority;

int l2tp\_version;

};

// 指向L2TP头部

l2tp = (l2tp\_header\*)(packet\_content+ether\_header\_length+ip\_header\_length+udp\_header\_length);

//逐字节解析L2TP头

// l2tp header 0 byte and 1 byte

l2tp\_type = ((\*(l2tp\_header)) & 0x80) >> 7;

l2tp\_len\_field = ((\*(l2tp\_header)) & 0x40) >> 6;

l2tp\_sequence\_field = ((\*(l2tp\_header)) & 0x08) >> 3;

l2tp\_offset\_field = ((\*(l2tp\_header)) & 0x02) >> 1;

l2tp\_priority = (\*(l2tp\_header)) & 0x01;

l2tp\_version = (\*(l2tp\_header + 1)) & 0x0F;

if (l2tp\_version == 0x02)

printf("\n版本号：2\n");

else

printf("版本号错误\n");

if (l2tp\_type)

printf("这是一条控制信息\n");

else

printf("这是一条数据信息\n");

if (l2tp\_offset\_field)

printf("偏移域存在\n");

else

printf("偏移域不存在\n");

if (l2tp\_priority)

printf("优先位设置为 %d.\n", l2tp\_priority);

else

printf("优先位未设置\n");

// l2tp header 2-3 byte

int l2tp\_total\_length;

int l2tp\_total\_length\_bias;

l2tp\_total\_length\_bias = (l2tp\_len\_field) ? 2 : 0;

if (l2tp\_total\_length\_bias) {

l2tp\_total\_length = (l2tp\_len\_field) ?

(((\*(l2tp\_header + l2tp\_total\_length\_bias)) & 0xFF) << 8) | ((\*(l2tp\_header + l2tp\_total\_length\_bias + 1)) & 0xFF) : 0;

printf("长度域存在，l2tp数据总长度为 %d byte.\n", l2tp\_total\_length);

}

else

printf("长度域不存在\n");

// l2tp header 4~7 byte

int l2tp\_tunnel\_id, l2tp\_session\_id;

int l2tp\_tunnel\_id\_bias, l2tp\_session\_id\_bias;

l2tp\_tunnel\_id\_bias = (l2tp\_len\_field) ? 4 : 2; // 4 or 2(without Length)

l2tp\_session\_id\_bias = l2tp\_tunnel\_id\_bias + 2;

l2tp\_tunnel\_id = (((\*(l2tp\_header + l2tp\_tunnel\_id\_bias)) & 0xFF) << 8)

| ((\*(l2tp\_header + l2tp\_tunnel\_id\_bias + 1)) & 0xFF);

l2tp\_session\_id = (((\*(l2tp\_header + l2tp\_session\_id\_bias)) & 0xFF) << 8)

| ((\*(l2tp\_header + l2tp\_session\_id\_bias + 1)) & 0xFF);

if (l2tp\_tunnel\_id == 0 || l2tp\_session\_id == 0) {

printf("Tunnel id 或 Session id 无效.\n");

return;

}

else {

printf("Tunnel Id: %d\n", l2tp\_tunnel\_id);

printf("Session Id: %d\n", l2tp\_session\_id);

}

// l2tp header 8-9 & 10-11 byte

int l2tp\_Ns, l2tp\_Nr;

int l2tp\_Ns\_bias, l2tp\_Nr\_bias;

l2tp\_Ns\_bias = (l2tp\_sequence\_field) ?

((l2tp\_len\_field) ? 8 : 6) : 0;

l2tp\_Nr\_bias = (l2tp\_sequence\_field) ? l2tp\_Ns\_bias + 2 : 0;

if (l2tp\_Ns\_bias || l2tp\_Nr\_bias) {

l2tp\_Ns = (((\*(l2tp\_header + l2tp\_Ns\_bias)) & 0xFF) << 8)

| ((\*(l2tp\_header + l2tp\_Ns\_bias + 1)) & 0xFF);

l2tp\_Nr = (((\*(l2tp\_header + l2tp\_Nr\_bias)) & 0xFF) << 8)

| ((\*(l2tp\_header + l2tp\_Nr\_bias + 1)) & 0xFF);

if (l2tp\_sequence\_field)

printf("序列域存在,Ns,Nr存在\n");

printf("Ns(next sequence number): %d.\n", l2tp\_Ns);

printf("Nr(next control message received): %d.\n", l2tp\_Nr);

}

else

printf("序列域不存在\n");

// l2tp header 12-13

int l2tp\_offset;

int l2tp\_offset\_bias;

l2tp\_offset\_bias = (l2tp\_offset\_field) ? (

l2tp\_len\_field ? (l2tp\_sequence\_field ? 12 : 8) : (l2tp\_sequence\_field ? 10 : 6)

) : 0;

l2tp\_offset = (((\*(l2tp\_header + l2tp\_offset\_bias)) & 0xFF) << 8)

| ((\*(l2tp\_header + l2tp\_offset\_bias + 1)) & 0xFF);

if (l2tp\_offset\_bias)

printf("偏移域存在，偏移长度为(有效数据距l2tp头的字节数): %d.\n", l2tp\_offset);

else

printf("偏移域不存在\n");

**4.5 输出有效载荷**

通过Packet\_handler的第二个参数可以得到数据包的总长度，用总长度依次减去以太网协议头长度、IPV4协议头长度、udp协议头长度、L2TP协议头长度便是有效载荷的总长度。再逐字节输出即可。

下面是实现代码

int total\_headers\_size = ethernet\_header\_length + ip\_header\_length + udp\_header\_length + l2tp\_header\_length;

printf("网络包总长度: %d bytes\n", header->caplen);

payload\_length = header->caplen -

(ethernet\_header\_length + ip\_header\_length + udp\_header\_length + l2tp\_header\_length);

printf("总载荷长度: %d bytes\n", payload\_length);

payload = packet + total\_headers\_size;

printf("\*\*\*\*\*\*载荷数据显示\*\*\*\*\*\*\n\n");

/\* Print payload in ASCII \*/

if (payload\_length > 0) {

const u\_char\* temp\_pointer = payload;

u\_char temp\_array[16] = { 0 };

int byte\_count = 0;

while (byte\_count < payload\_length) {

if (byte\_count % 16 == 0 && byte\_count) {

printf(" | ");

unsigned int index;

for (index = 0; index < 16; ++index) {

printf("%c", temp\_array[index]);

temp\_array[index] = 0; // erase

}

putchar('\n');

}

printf("%02X ", \*temp\_pointer);

temp\_array[byte\_count % 16] = \*temp\_pointer;

++temp\_pointer;

++byte\_count;

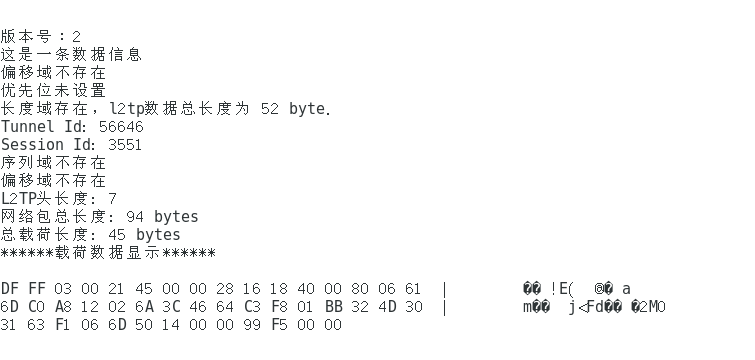
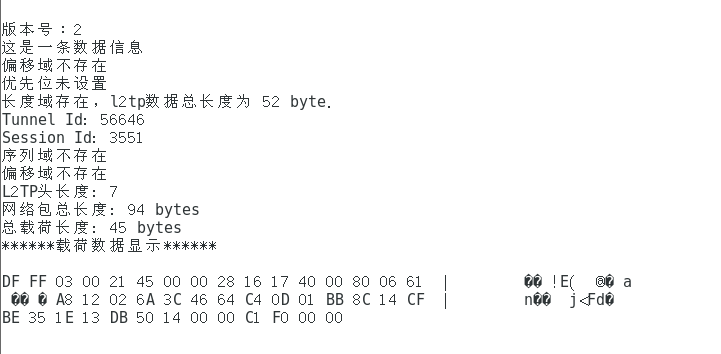
}

printf("\n");

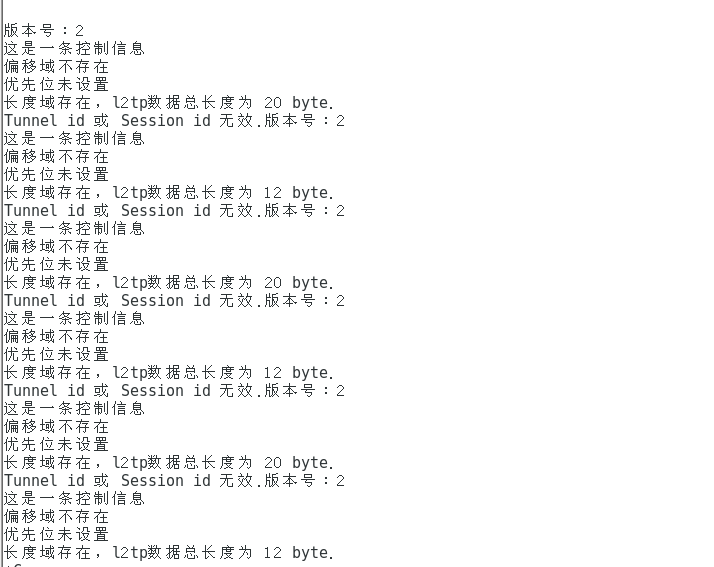
}

**五、测试用例**

1. **成功捕获的数据包**



1. **成功捕获的控制包**



1. **失败情况**

**六、问题解决**

1. **大小端问题**

在对IPV4协议头进行分析时，发现本该为4的协议号总是出错，而首部长度却一直为4，思考会不会是定义次序有问题，查阅资料后发现可能有大小端的问题。

小端模式指的是一个多字节的数据结构放进cpu，会把每个字节倒着存进去，比如存个short：01000010 10000000 00000011 11111101，小端模式会存成11111101 00000011 10000000 01000010。

version和ihl每个都是4bit,可以类比联想，之所以ip报头前两个字段错误原因可能是因为这两个4位的字段是倒着存在一个字节中的。

下面是实验代码：

#include<stdio.h>

#include<stdlib.h>

struct baite{

unsigned char ver:4;

unsigned char ihl:4;

}\*ip;

bool data[100];

int main(){

ip=(struct baite\*)data;

ip->ver=4;

ip->ihl=5;

printf("%d %d ",b->ver,b->ihl);//小端电脑输出结果是 4 5

printf("%d",\*(int \*)data);//小端电脑输出结果是84 即01010100 0101是5 0100是4

return 0;

}

电脑输出结果是84 即0101 0100，0101是5，0100是4，说明ihl（5）存在了version（4）前面，尽管你写的定义是让version存在前面的，但因为小端模式的缘故，它倒着存了。（大端模式输出是69，即0100 0101）

这说明，如果在一个字节的内部定义了两个长4位的变量，在小端模式下将倒着存，所以ip报头得区分对待大小端的定义。

**七、总结收获**