# 基于Raw Socket的Sniffer实现

## 本章训练目的与要求

网络监控软件能够监控网络流量，发现网络中异常的数据流，从而帮助系统及时而有效地发现和防御网络攻击。随着互联网的快速普及和发展以及网络攻击行为的频繁出现，保证网络的安全运行成为网络监控软件的一个重要目的。除此之外，网络监控也是黑客常用的手段。因此，网络监控技术是网络安全的基本技术之一，对维护网络的安全性具有重要作用。

本章的训练目的：

① 理解网络监控技术的基本原理。

② 理解（TCP/IP）网络层次结构。

③ 掌握Raw Socket的基本原理。

④ 熟练掌握TCP/IP、ICMP等协议以及socket编程技术相关背景知识。

本章训练要求：

① 利用原始套接字编写一个网络嗅探器捕获网络数据报。

② 分析基本的数据报信息。

③ 实现简单的过滤器功能。

## 相关背景知识

### 原始套接字

Linux系统套接字主要分成3类：TCP套接字、UDP套接字和原始套接字。TCP套接字又称流式套接字，是建立在传输层TCP协议上的套接字。UDP套接字又称数据报套接字，是建立在传输层UDP协议上的套接字。原始套接字是一种比较特殊的套接字，虽然创建原始套接字的方法和TCP、UDP套接字基本相同，但是它的功能和TCP、UDP套接字相比却存在很大的差异。TCP/UDP套接字只能接收和操作传输层或者传输层之上的数据报，因为当IP层把数据报往上传给传输层的时候，下层的数据报头部信息（如IP数据报头部和Ethernet帧头部）都已经丢掉了。而原始套接字可以直接对链路层的数据报进行操作。

#### 原始套接字特点

原始套接字有如下5个特点。

1. 原始套接字可以读写ICMP、IGMP数据报。对于ICMP和IGMP等封装在IP数据报中但是又在传输层之下的数据报,系统内核不管是否已经有注册的句柄来处理这些数据报,都会将这些数据报复制一份传递给协议类型匹配的原始套接字。通过这样的方式，系统就可以在用户空间中处理这些数据报，而不用内核来处理，从而减轻内核负担。
2. 原始套接字可以读写部分的IP数据报。通常这些数据报的协议域系统内核不能识别或者系统内核没有处理。对于内核不能识别其协议类型的数据报，内核首先会对其进行必要的校验，然后查看是否有匹配的原始套接字来处理，如果有匹配的原始套接字，内核复制一份该数据报给匹配的原始套接字，否则，直接丢弃该IP数据报。
3. 对于TCP包和UDP包，因为内核有相应的句柄对这些包进行处理，所以内核并不会将其传递给任何原始套接字。因此如果想要通过原始套接字来捕获TCP和UDP包，创建原始套接字的时候，必须将第一个参数设置为PF\_PACKET，第三个参数指定为htons(ETH\_P\_IP)或者htons(ETH\_P\_ALL)，这样原始套接字将直接通过数据链路层捕获数据报。本章实现的sniffer程序就需要用这样的方式从链路层捕获所有流经本机网卡的数据报。
4. 原始套接字可以构造自己的IP报头，这样就可以构造和发送特定的IP数据报。但是必须先通过setsockopt函数设置套接字IP\_HDRINCL选项，也就是设置让程序自己填充IP报头。而不是由内核自动填充。
5. 原始套接字需要超级用户权限才能创建。

#### 原始套接字相关操作

1. 创建原始套接字：

|  |
| --- |
| int rawsock;  rawsock = socket(domain, SOCK\_RAW, protocol); |

1. 第一个参数domain表示地址族或者协议族，一般来说地址族用AF\_INET，协议族用PF\_INET. 在功能上，PF\_INET和AF\_INET在是没有区别的。PF\_INET中的PF\_代表protocol family ，而AF\_代表address family ，在最初设计的时候是计划让一个protocol family 包含多个address family，但是这个计划并没有实现。在套接字的头文件中有#define  PF\_INET  AF\_INET这样一个宏定义，因此这两个宏在数值上是相等的，在功能上也没有区别。如果需要原始套接字在链路层上捕获数据报，需要将该参数设置为PF\_PACKET。
2. 第二个参数为套接字类型，原始套接字对应类型为SOCK\_RAW。
3. 第三个参数potocol是一个常量定义，可以根据程序的需求选择相应的protocol，形式如IPPROTO\_XXX。这些宏在头文件<netinet/in.h>中都有相应的定义。如果想要通过原始套接字来捕获TCP和UDP包，创建原始套接字的时候，要将该参数指定为htons(ETH\_P\_IP)或者htons(ETH\_P\_ALL)。

创建原始套接字需要超级用户权限，否则socket函数将不能成功创建原始套接字，返回-1值，同时会将errno置为EACCES。

1. 绑定和连接操作：

通常情况下，原始套接字是不需要绑定操作的，但是也可以将原始套接字绑定在一个本地的地址上。原始套接字的绑定操作只是针对IP地址，而不会涉及到端口。也就是说，原始套接字不会绑定在一个固定的端口上。调用bind()函数之后，接收数据报时，内核只会将目的IP地址为本机IP地址的数据报传递给该原始套接字。在原始套接字发送数据报时，数据报的源IP地址会自动设置为绑定的IP地址。如果没有调用bind()函数，接收数据报时，内核会把所有的数据报传递给该原始套接字，原始套接字发送数据报时，数据报的源IP地址会自动设置为发送接口的的主IP地址。

原始套接字可以通过调用connect()函数连接套接字。同样connect()函数也只涉及IP地址，而不涉及到端口号。调用connect()函数，连接成功后，原始套接字就可以通过write()和send()函数发送数据报。

1. 读写原始套接字：

原始套接字的写操作通常使用sendto()和sendmsg()函数。如果调用connect函数连接成功后就可以使用write()和send()函数来发送数据报。默认情况下，系统内核将自动填充IP头部。但是如果通过setsockopt()函数设置了IP\_HDRINCL选项，则必须在程序中手动填充IP头部，内核只负责填充后的IP头部的校验和。当数据报的长度大于链路中最大传输单元MTU（Maximum Transmission Unit）的时候，系统内核会自动将该数据报进行分片。

原始套接字的读操作通常使用recvfrom()和recvmsg()函数。如果调用connect()函数连接成功后就可以使用recv()和read()函数来接收数据报，同时内核也只会将源地址为connect()函数连接的IP地址的数据报传递给这个原始套接字。

#### 数据报处理

当系统收到一个数据报并且需要将其传递给原始套接字的时候，内核检查所有进程的所有套接字，按照下面的原则寻找匹配的原始套接字，然后将数据报拷贝给所有匹配的原始套接字。

1. 原始套接字的协议域和数据报的协议域完全匹配并且为非0值，内核将数据报传递给该原始套接字。
2. 如果原始套接字通过bind()函数绑定到了一个本地的IP地址上，那么内核只会将目的IP地址和套接字绑定的IP地址匹配的数据报传递给该原始套接字。
3. 如果原始套接字调用connect()函数和远程的IP地址连接，那么内核只将源IP地址和该远程IP地址匹配的数据报传递给该原始套接字。

如果一个原始套接字在创建的时候协议域为0（也就是socket()函数的第三个参数为0），并且没有调用bind()和connect函数()进行绑定和连接操作，那么该原始套接字将接收所有内核传递给其他原始套接字的数据报。另外如前面提到的，原始套接字不能接收TCP/UDP包，只能接收ICMP包、IGMP包以及一些协议域不被系统内核理解的数据报。如果想要通过原始套接字来捕获TCP和UDP包，创建原始套接字的时候，必须将第一个参数设置为PF\_ PACKET，第三个参数指定为htons(ETH\_P\_IP)或者htons(ETH\_P\_ALL)。

### TCP/IP网络协议栈结构

#### TCP/IP协议栈模型

计算机网络是具有层次的，是按照不同的通信功能划分的层次结构系统。每一层都由特定的协议来负责。常用的协议族模型主要是OSI七层协议栈模型和TCP/IP四层协议栈模型。其中TCP/IP协议栈模型是当前最流行也是最为成功的协议栈模型。本章的试验也是基于TCP/IP协议栈模型完成的。TCP/IP协议栈模型将网络划分为4层，依次是应用层（application layer）、传输层(transport layer)、互联层(internet layer)和主机-网络层(host-to-network layer)。如图6-1所示。



##### 图6-1 TCP/IP 分层体系结构

#### 常见的网络协议及其所在的层次位置

TCP/IP协议栈模型并没有对主机-网络层做具体的定义。为使网络层次结构更加清晰，通常都对主机-网络层再做一个划分，分为物理层和数据链路层。这样用五层网络体系结构即物理层、数据链路层、网络层、传输层和应用层作为分层的标准。图6-2给出了常见的网络协议及其所在的层次位置。

##### 图6-2 常用网络协议及其所在层次位置

### 数据的封装与解析

当应用程序通过协议栈向网络传送数据时，应用层的数据报要依次传递给传输层、互联层、数据链路层，最后通过物理层进入到网络中。在数据报经过每一层的时候，每一层都要在数据报增加该层的头部（和尾部）信息，以此来实现层次控制，这个过程称为数据的封装。同样，当物理层收到数据报的时候，需要依次传递给数据链路层、网络层、传输层，最后送至应用层。在数据报经过每一层的时候，每一层都需要对对应于该层的头部（和尾部）信息进行解析，最终从报文中解析出应用层数据后交给应用程序处理。该过程就称为数据拆包或解析。图6-3给出了数据的封装与解析过程。

##### 图6-3 数据的封装与解析过程

本章所要实现的sniffer程序就是利用原始套接字捕获所有流经本机网卡的数据报并且对其进行解析。按照图6-3的解析过程，由底层往上逐层解析。最终将每层解析的结果显示出来。

## 实例编程练习

### 编程练习要求

根据前面章节介绍的关于原始套接字的知识，利用原始套接字编写一个网络嗅探器捕获网络数据报并且分析基本的数据报信息，例如IP地址、端口号、协议类型、物理地址等。另外要求实现简单的过滤器功能，能够捕获指定的数据报，例如捕获指定IP地址、指定协议的数据报等。

### 编程训练设计与分析

#### 程序整体框架

现代操作系统通常都提供了对底层网络数据报捕获的机制。虽然不同的操作系统实现的底层数据报捕获机制可能会不一样，但是功能上来说却是大同小异。数据报从网络传递到主机应用程序的路径依次为网卡、设备驱动层、数据链路层、网络层、传输层，最后到达应用层程序。本文关注的数据报捕获机制主要是在数据链路层。捕获的数据报是内核对该数据报的一份拷贝，这样的数据报捕获机制不会影响到操作系统对数据报进行网络协议栈处理。整个程序的主体框架如图6-4所示。



##### 图6-4 程序整体框架图

本程序主体结构主要分成3部分，由下而上，分别是数据捕获模块、数据报过滤模块、协议解析模块。首先由数据捕获模块从网络中捕获数据报，然后在数据报过滤模块中按照设定的过滤规则将数据报进行过滤，最后再将过滤后的数据报通过协议解析模块解析显示出来。

本实验要求用原始套接字来捕获数据报。原始套接字不仅可以捕获数据报，还可以发送指定的数据报。但是在本实验的程序中只需要用到原始套接字捕获数据报的功能。通过原始套接字捕获数据报的流程如图6-5所示。



##### 图6-5 原始套接字捕获数据报流程

#### 程序核心代码

程序定义了两个类，一个是rawsocket类，另一个是rawsocsniffer类。rawsocket类对原始套接字的一些设置和操作进行了封装，避开了很多繁琐的原始套接字操作，使得程序更加直观，也使得对原始套接字的操作和扩充变得更加方便。rawsocsniffer类对整个嗅探器的主要功能进行了封装，包括创建原始套接字，设置混杂模式，捕获数据报并且设置过滤，对数据报进行分析等。

下面给出各个模块的核心代码，为了使程序的流程更加清晰，代码没有做很严格的错误处理。实际中应该增加更为严格的错误处理，以此来保证程序的健壮性。

1. rawsocket类：

rawsocket类的定义如下。

|  |
| --- |
| class rawsocket  {  private:  int sockfd;  public:  rawsocket(const int protocol);  ~rawsocket() ;  //set the promiscuous mode.  bool dopromisc(char \*nif);  //capture packets.  int receive(char \*recvbuf,int buflen,struct sockaddr\_in \*from,int \*addrlen);  }; |

rawsocket类维护1个数据成员sockfd，即原始套接字句柄。维护4个成员函数，构造函数根据传入的参数构造原始套接字，在析构函数中关闭该原始套接字。dopromisc()函数对ioctl函数进行了封装，用于设置网卡混杂模式。receive()函数是对原始套接字的recvfrom ()函数进行了封装，用于捕获数据报。对原始的套接字的其他的设置和操作都可以封装在该类中。由于在本章的程序中，只需要用到上面这些操作。所以其他的操作没有封装在该类中，有兴趣的读者可以对其进行扩充。

1. 构造函数：

构造函数根据参数传入的协议类型创建原始套接字。socket()函数的第一个参数必须设置为PF\_PACKET，这样才能直接从链路层捕获数据。

|  |
| --- |
| rawsocket::rawsocket(const int protocol)  {  sockfd=socket(PF\_PACKET,SOCK\_RAW,protocol);  if(sockfd<0)  {  perror("socket error: ");  }  } |

1. dopromisc()函数：

dopromisc()函数设置网卡为混杂模式。该函数以网卡的名字为参数。要对指定的网卡设置混杂模式，只要以该网卡名字为参数调用dopromisc()函数即可。如果需要恢复网卡模式只需将代码“ifr.ifr\_flags |= IFF\_PROMISC”中的“|=”改为“&=”即可。

|  |
| --- |
| bool rawsocket::dopromisc(char\*nif)  {  struct ifreq ifr;  strncpy(ifr.ifr\_name, nif,strlen(nif)+1);  if((ioctl(sockfd, SIOCGIFFLAGS, &ifr) == -1))  {  perror("ioctlread: ");  return false;  }  ifr.ifr\_flags |= IFF\_PROMISC;  if(ioctl(sockfd, SIOCSIFFLAGS, &ifr) == -1 )  {  perror("ioctlset: ");  return false;  }  return true;  } |

1. receive()函数：

receive()函数是对原始套接字的recvfrom ()函数进行的封装，实现了对原始套接字的recvfrom ()函数更方便地调用。当成功地捕获一个数据报后，recvfrom()函数的返回值为接收到的数据报的长度。

|  |
| --- |
| int rawsocket::receive(char \*recvbuf,int buflen, struct sockaddr\_in \*from,int \*addrlen)  {  int recvlen;  recvlen=recvfrom(sockfd,recvbuf,buflen,0,(struct sockaddr \*)from,(socklen\_t \*)addrlen);  recvbuf[recvlen]='\0';  return recvlen;  } |

1. rawsocsniffer类：

rawsocket类的定义如下。

|  |
| --- |
| class rawsocsniffer:public rawsocket  {  private:  filter simfilter;  char \*packet;  const int max\_packet\_len;  public:  rawsocsniffer(int protocol);  ~rawsocsniffer();  bool init();  void setfilter(filter myfilter);  bool testbit(const unsigned int p, int k);  void setbit(unsigned int &p,int k);  void sniffer();  void analyze();  void ParseRARPPacket();  void ParseARPPacket();  void ParseIPPacket();  void ParseTCPPacket();  void ParseUDPPacket();  void ParseICMPPacket();  void print\_hw\_addr(const unsigned char \*ptr);  void print\_ip\_addr(const unsigned long ip);  }; |

rawsocsniffer类对嗅探器主要功能进行了封装。该类继承自rawsocket类。rawsocsniffer类维护3个数据成员，一个filter类型的数据结构simfilter用于设置过滤条件。一个char 型指针用于存储数据报，另一个int型数据记录最大数据报长度。

1. 设置过滤：

过滤功能主要是通过simfilter数据成员来实现的，simfilter结构如下。

|  |
| --- |
| typedef struct filter  {  unsigned long sip;  unsigned long dip;  unsigned int protocol;  } filter; |

该程序实现了针对源IP，目的IP和协议类型进行过滤的功能。filter结构维护了3个变量：源IP、目的IP、协议类型。程序初始化时会对rawsocsniffer类的simfilter结构进行设置来设定过滤条件。例如，如果只要求捕获来自ip地址为“192.168.0.45”的数据报，则将simfilter结构中的sip设置为192.168.0.45。同样如果要对目的IP进行过滤，则需要设置simfilter的dip为指定IP。对协议类型进行过滤是通过对protocol变量进行操作来实现的。protocol变量中的每一位对应一种协议类型。某位为1则表示接收该位所对应协议的数据报，为0则表示不接收。protocol变量中哪一位对应哪种协议类型则可以按照自定义的规则来设置。在本程序中，arp协议对应protocol的第1位，tcp协议对应第2位。udp协议对应第3位，icmp协议对应第4位。在按协议类型分析数据报之前，先查看protocol变量中该协议对应的位是否为1，如果为1则分析该数据报，否则丢弃。例如，如果protocol变量为0x0002,二进制为（0000000000000010），第2位为1，其他位全为0，表示程序只分析tcp包。如果protocol变量为0x0005，二进制为（0000000000000101），第1位和第3位为1，其余位都为0，表示程序只分析udp包和arp包。如果protocol等于0x0000，所有位都为0，也就是没有设置过滤，那么程序将分析所有捕获的数据报。读者可以对该过滤机制进行扩充，以实现更多的过滤功能。rawsocsniffer类有3个函数对simfilter变量进行维护。

①setfilter()函数：

setfilter()函数根据传入的参数设置simfilter。

|  |
| --- |
| void rawsocsniffer::setfilter(filter myfilter)  {  simfilter.protocol=myfilter.protocol;  simfilter.sip=myfilter.sip;  simfilter.dip=myfilter.dip;  } |

②testbit()函数：

testbit()函数用于测试某一无符号整型变量的指定位是否为1.

|  |
| --- |
| bool rawsocsniffer::testbit(const unsigned int p,int k)  {  if((p>>(k-1))&0x0001)  return true;  else  return false;  } |

③setbit()函数：

setbit()函数用于将某一无符号整型变量的指定位置为1.

|  |
| --- |
| void rawsocsniffer::setbit(unsigned int &p,int k)  {  p=(p)|((0x0001)<<(k-1));  } |

1. 数据报捕获：

sniffer()函数用于启动数据报捕获过程，该函数通过循环调用基类rawsocket 的receive函数来捕获数据报，receive函数返回值为捕获到的数据报长度，所以当返回值大于0时表示捕获到数据报，然后调用analyze()函数对捕获到的数据报进行处理。

|  |
| --- |
| void rawsocsniffer::sniffer()  {  struct sockaddr\_in from;  int sockaddr\_len=sizeof(struct sockaddr\_in);  int recvlen=0;  while(1)  {  recvlen=receive(packet,max\_packet\_len,&from,&sockaddr\_len);  if(recvlen>0)  {  analyze();  }  else  {  continue;  }  }  } |

rawsocsniffer类维护了一个数据成员packet来临时存储捕获到的数据报。packet是一个char型指针，rawsocsniffer类的构造函数给该指针分配了一块大小为max\_packet\_len的内存空间。在本程序中max\_packet\_len设置为2048。

#### 数据报头部格式

本程序只分析了一些常见的协议，比如ARP协议、IP协议、TCP协议、UDP协议、ICMP协议。各种协议的的数据报头部结构定义如下。

1. 以太网帧头部：

以太网帧头部结构定义如下。

|  |
| --- |
| typedef struct ether\_header\_t{  BYTE des\_hw\_addr[6]; //目的MAC地址  BYTE src\_hw\_addr[6]; //源MAC地址  WORD frametype; //数据长度或类型  } ether\_header\_t; |

1. IP包头部：

IP包头部结构定义如下。

|  |
| --- |
| typedef struct ip\_header\_t{  BYTE hlen\_ver; //头部长度和版本信息  BYTE tos; //8位服务类型  WORD total\_len; //16位总长度  WORD id; //16位标识符  WORD flag; //3位标志+13位片偏移  BYTE ttl; //8位生存时间  BYTE protocol; //8位上层协议号  WORD checksum; //16位校验和  DWORD src\_ip; //32位源IP地址  DWORD des\_ip; //32位目的IP地址  } ip\_header\_t; |

1. ARP包头部：

ARP包头部结构定义如下。

|  |
| --- |
| typedef struct arp\_header\_t{  WORD hw\_type; //16位硬件类型  WORD prot\_type; //16位协议类型  BYTE hw\_addr\_len; //8位硬件地址长度  BYTE prot\_addr\_len; //8位协议地址长度  WORD flag; //16位操作码  BYTE send\_hw\_addr[6]; //源Ethernet网地址  DWORD send\_prot\_addr; //源IP地址  BYTE des\_hw\_addr[6]; //目的Ethernet网地址  DWORD des\_prot\_addr; //目的IP地址  } arp\_header\_t; |

1. TCP包头部：

TCP包头部结构定义如下。

|  |
| --- |
| typedef struct tcp\_header\_t{  WORD src\_port; //源端口  WORD des\_port; //目的端口  DWORD seq; //seq号  DWORD ack; //ack号  BYTE len\_res; //头长度  BYTE flag; //标志字段  WORD window; //窗口大小  WORD checksum; //校验和  WORD urp; //紧急指针  } tcp\_header\_t; |

1. UDP包头部：

UDP包头部结构定义如下。

|  |
| --- |
| typedef struct udp\_header\_t{  WORD src\_port; //源端口  WORD des\_port; //目的端口  WORD len; //数据报总长度  WORD checksum; //校验和  } udp\_header\_t; |

1. ICMP包头部：

ICMP包头部结构定义如下。

|  |
| --- |
| typedef struct icmp\_header\_t{  BYTE type; //8位类型  BYTE code; //8位代码  WORD checksum; //16位校验和  WORD id; //16位标识符  WORD seq; //16位序列号  } icmp\_header\_t; |

#### 数据报解析

数据报解析过程就是对捕获的数据报按照数据链路层（MAC）、网络层（IP、ARP/RARP）、传输层（TCP、UDP、ICMP）和应用层（HTTP等）的层次结构自底向上进行解析，分析各个协议的字段，最后将解析结果显示输出。数据报的解析过程如图6-6所示。本程序只分析了一些常见的协议，比如ARP协议、IP协议、TCP协议、UDP协议和ICMP协议。其他的协议读者可以根据需求对程序进行扩充。



##### 图6-6 数据报解析流程

程序每捕获到一个数据报就调用analyze()函数对该数据报进行解析。analyze()函数根据以太帧的类型调用相应的上层解析函数对数据报进行解析。帧类型0x0800表示上层为IP包，0x0806表示上层为ARP包，0x0835表示上层为RARP包。调用上层解析函数前先根据过滤器的设置判断是否要对该协议的数据报进行解析。

|  |
| --- |
| void rawsocsniffer::analyze()  {  ether\_header\_t \*etherpacket=(ether\_header\_t \*)packet;  if(simfilter.protocol==0)  simfilter.protocol=0xff;  switch (ntohs(etherpacket->frametype))  {  case 0x0800:  if(((simfilter.protocol)>>1))  {  cout<<"\n\n/\*---------------ip packet--------------------\*/"<<endl;  ParseIPPacket();  }  break;  case 0x0806:  if(testbit(simfilter.protocol,1))  {  cout<<"\n\n/\*--------------arp packet--------------------\*/"<<endl;  ParseARPPacket();  }  break;  case 0x0835:  if(testbit(simfilter.protocol,5))  {  cout<<"\n\n/\*--------------RARP packet--------------------\*/"<<endl;  ParseRARPPacket();  }  break;  default:  cout<<"\n\n/\*--------------Unknown packet----------------\*/"<<endl;  cout<<"Unknown ethernet frametype!"<<endl;  break;  }  } |

1. 解析ARP包：

程序解析了ARP包几个主要字段，包括硬件地址长度、协议地址长度、协议类型、操作类型以及源IP地址、源MAC地址、目的IP地址、目的MAC地址。其中操作类型为0x0001表示ARP请求，0x0002表示ARP应答。

|  |
| --- |
| void rawsocsniffer::arppacket\_analyze()  {  arp\_packet\_t \*arppacket=(arp\_packet\_t \*)packet;  print\_hw\_addr(arppacket->arpheader.des\_hw\_addr);  print\_hw\_addr(arppacket->arpheader.send\_hw\_addr);  cout<<endl;  print\_ip\_addr(arppacket->arpheader.send\_prot\_addr);  print\_ip\_addr(arppacket->arpheader.des\_prot\_addr);  } |

1. 解析IP包：

程序首先判断过滤条件，根据过滤器的源IP和目的IP对数据报进行过滤。然后再根据IP层的协议域字段的值来调用对应的上层协议解析函数对数据报进行解析。其中协议域字段值为1表示上层为ICMP包，6表示TCP，17表示UDP。

|  |
| --- |
| void rawsocsniffer::ParseIPPacket()  {  ip\_packet\_t \*ippacket=(ip\_packet\_t \*)packet;  cout<<"ipheader.protocol: "<<int(ippacket->ipheader.protocol)<<endl;  if(simfilter.sip!=0)  {  if(simfilter.sip!=(ippacket->ipheader.src\_ip))  return;  }  if(simfilter.dip!=0)  {  if(simfilter.dip!=(ippacket->ipheader.des\_ip))  return;  }  switch (int(ippacket->ipheader.protocol))  {  case 1:  if(testbit(simfilter.protocol,4))  {  cout<<"Received an ICMP packet"<<endl;  ParseICMPPacket();  }  break;  case 6:  if(testbit(simfilter.protocol,2))  {  cout<<"Received an TCP packet"<<endl;  ParseTCPPacket();  }  break;  case 17:  if(testbit(simfilter.protocol,3))  {  cout<<"Received an UDP packet"<<endl;  ParseUDPPacket();  }  break;  /\*省略针对其他协议的分析\*/  }  } |

1. 解析ICMP包：

程序解析了ICMP 包的类型、编码、标示符和序列号字段。

|  |
| --- |
| void rawsocsniffer::icmppacket\_analyze()  {  icmp\_packet\_t \*icmppacket=(icmp\_packet\_t \*)packet;  cout<<setw(20)<<"MAC address: from ";  print\_hw\_addr(icmppacket->etherheader.src\_hw\_addr);  cout<<"to ";  print\_hw\_addr(icmppacket->etherheader.des\_hw\_addr);  cout<<endl<<setw(20)<<"IP address: from ";  print\_ip\_addr(icmppacket->ipheader.src\_ip);  cout<<"to ";  print\_ip\_addr(icmppacket->ipheader.des\_ip);  cout<<endl;  cout<<setw(12)<<"icmp type: "<<int(icmppacket->icmpheader.type)<<" icmp code: "<<int(icmppacket->icmpheader.code)<<endl;  cout<<setw(12)<<"icmp id: "<<ntohs(icmppacket->icmpheader.id)<<" icmp seq: "<<ntohs(icmppacket->icmpheader.seq)<<endl;  } |

1. 解析TCP包：

程序解析了TCP包的源端口、目的端口、序列号、ACK等字段。

|  |
| --- |
| void rawsocsniffer::tcppacket\_analyze()  {  tcp\_packet\_t \*tcppacket=(tcp\_packet\_t \*)packet;  cout<<setw(20)<<"MAC address: from ";  print\_hw\_addr(tcppacket->etherheader.src\_hw\_addr);  cout<<"to ";  print\_hw\_addr(tcppacket->etherheader.des\_hw\_addr);  cout<<endl<<setw(20)<<"IP address: from ";  print\_ip\_addr(tcppacket->ipheader.src\_ip);  cout<<"to ";  print\_ip\_addr(tcppacket->ipheader.des\_ip);  cout<<endl;  cout<<setw(10)<<"srcport: "<<ntohs(tcppacket->tcpheader.src\_port)<<" desport: "<<ntohs(tcppacket->tcpheader.des\_port)<<endl;  cout<<"seq: "<<ntohl(tcppacket->tcpheader.seq)<<" ack: "<<ntohl(tcppacket->tcpheader.ack)<<endl;  } |

1. 解析UDP包：

程序解析了UDP包的源端口、目的端口、数据报长度等字段。

|  |
| --- |
| void rawsocsniffer::udppacket\_analyze()  {  udp\_packet\_t \*udppacket=(udp\_packet\_t \*)packet;  cout<<setw(20)<<"MAC address: from ";  print\_hw\_addr(udppacket->etherheader.src\_hw\_addr);  cout<<"to ";  print\_hw\_addr(udppacket->etherheader.des\_hw\_addr);  cout<<endl<<setw(20)<<"IP address: from ";  print\_ip\_addr(udppacket->ipheader.src\_ip);  cout<<"to ";  print\_ip\_addr(udppacket->ipheader.des\_ip);  cout<<endl;  cout<<setw(10)<<"srcport: "<<ntohs(udppacket->udpheader.src\_port)<<" desport: "<<ntohs(udppacket->udpheader.des\_port)\  <<" length:"<<ntohs(udppacket->udpheader.len)<<endl;  } |

## 扩展与提高

### 使用libpcap捕获数据报

#### libpcap介绍

对数据报的捕获也可以借助于Linux下的libpcap开发包来实现。该开发包是由Berkeley大学的Van Jacobson、Craig Leres 和 Steven McCanne合作开发，它提供了丰富的API函数，可以帮助程序员快速地开发数据报捕获软件。libpcap的官方网址是：[www.tcpdump.org](http://www.tcpdump.org)。读者可以在此网站上下载该开发包。该开发包对捕获网络数据报的功能进行了封装，使用起来非常方便。并且支持自定义过滤规则，只捕获用户感兴趣的数据报。其过滤规则非常细致，可以进行各种复杂组合的过滤。由于它的过滤模块是在内核层次中实现的，所以效率非常高。很多著名的软件如抓包工具tcpdump和网络入侵检测系统snort都是基于libpcap开发的。

#### libpcap的功能

libpcap的功能可以大致归纳为以下几点：

1. 捕获数据报

捕获数据报是libpcap最基本也是最强大的功能。使用libpcap可以方便而且高效地捕获网络数据报。

1. 过滤数据报

libpcap提供了强大的过滤机制，在内核层次中对数据报进行过滤，效率非常高，而且其过滤规则非常细致，可以进行各种复杂组合，实现强大的过滤功能。

1. 分析数据报

libpcap在捕获数据报时提供了一些辅助信息，比如捕获时间，数据报长度等信息。可以帮助开发者更好地分析数据报。

1. 存储数据报

libpcap提供了将数据报存储到本地的功能，从网上捕获数据报后，可以先将其存储在本地电脑上，以后再对其进行分析。libpcap提供了一种机制对离线的数据报进行分析，也就是从libpcap保存在本地的文件中读取数据报进行分析。

#### libpcap进行数据报捕获和分析的步骤

使用libpcap进行数据报捕获和分析的步骤一般如下。

1. 使用pcap\_findalldevs()获取设备列表。
2. 使用pcap\_lookupnet()获取网络地址和子网掩码。
3. 使用pcap\_open\_live()打开指定设备。
4. 使用pcap\_compile()编译过滤规则。
5. 使用pcap\_setfilter()设置过滤规则。
6. 使用pcap\_loop()来循环捕获数据报，并在该函数中调用数据报分析模块解析数据报。
7. 使用pcap\_close()关闭设备句柄。

其流程图如图6-7所示。



##### 图6-7 数据报捕获流程

#### 使用libpcap捕获数据报的关键代码

1. 获取并且输出网络设备列表信息

首先通过pcap\_findalldevs()函数来获取本机所有网络设备列表，并且将其存储在device变量中。然后通过ifprint函数将找到的设备信息打印出来。

|  |
| --- |
| if(pcap\_findalldevs(&devices,errbuf)==-1)  {  printf("Error in pcap\_findalldevs() : %s \n",errbuf);  return -1;  }  else  {  printf("Find the following devices on your machine: \n");  ifprint(devices,devname);  } |

1. 获取网络地址和子网掩码。

其中参数devname为网络设备的名字，获取的网络地址存储在变量net\_ip中，子网掩码存储在变量net\_mask中。

|  |
| --- |
| if(pcap\_lookupnet(devname,&net\_ip,&net\_mask,errbuf)==-1)  {  printf("Error in the pcap\_lookupnet: %s \n",errbuf);  goto error;  } |

1. 打开指定设备

其中参数devname为要打开的设备名字。

|  |
| --- |
| if((dev\_handle\_pcap=pcap\_open\_live(devname,BUFSIZ,1,0,errbuf))==NULL)  {  printf("Error in the pcap\_open\_live! \n");  goto error;  } |

1. 编译和设置过滤规则

其中参数bpf\_filter的数据类型为struct bpf\_program，参数bpf\_filter\_string是设置的过滤字符串。libpcap中支持多种类型的过滤，功能非常强大。包括基于IP地址、MAC地址、端口号等类型的过滤，并且可以通过一些关键字设置过滤表达式，组合出各种过滤类型。设置过滤器最主要的工作在于如何根据规则构造过滤表达式。

|  |
| --- |
| if((pcap\_compile(dev\_handle\_pcap,&bpf\_filter,bpf\_filter\_string,0,net\_ip))==-1)  {  printf("Error in the pcap\_compile! \n");  goto error;  }  else  {  if((pcap\_setfilter(dev\_handle\_pcap,&bpf\_filter))==-1)  {  printf("Error in the pcap\_setfilter ! \n");  goto error;  }  } |

1. 过滤表达式

过滤表达式的语法主要有如下几点。

1. 表达式支持逻辑操作符，可使用关键字and、or、 not对子表达式进行组合，同时支持使用小括号。
2. 基于协议的过滤要使用协议限定符，协议限定符可以为ip、arp、 rarp、 tcp 和 udp等。
3. 基于MAC地址的过滤要使用限定符ether（代表Ethernet网地址）。当该MAC地址仅作为源地址时过滤表达式为ether src mac\_addr，仅作为目的地址时表达式为ether dst mac\_addr，既作为源地址又作为目的地址时表达式为ether host mac\_addr。此外应注意mac\_addr应遵循由冒号分隔的十六进制格式，如00:E0:4C:E0:38:88，否则编译过滤器时会出错。
4. 基于IP地址的过滤应使用限定符host（代表主机地址）。当该IP地址仅作为源地址时过滤表达式为src host ip\_addr，仅作为目的地址时表达式为dst host ip\_addr，既作为源地址又作为目的地址时表达式为host ip\_addr。
5. 基于端口的过滤应使用限定符port。例如仅接收80端口的数据报则表达式为port 80。

设置过滤字符串的示例如下。

|  |
| --- |
| 例1：只捕获arp或icmp包  过滤表达式为：arp or (ip and icmp) ，或者简写为：arp or icmp  例2：捕获以192.168.1.27为源或目的地址的端口为80的tcp包  过滤表达式为：(ip and tcp) and (host 192.168.1.27) and (port 80)  例3：捕获主机192.168.1.27与192.168.1.22之间传递的所有udp包  过滤表达式为：(ip and udp) and ((src host 192.168.1.27 and dst host 192.168.1.22) or (dst host 192.168.1.27 and src host 192.168.1.22))  例4：捕获从mac地址00-13-D3-A1-D2-F6到00-50-56-C0-00-01之间所有的arp包  过滤表达式为：arp and (ether src 00:13:D3:A1:D2:F6 and ether dst 00:50:56:C0:00:01) |

1. 捕获数据报

|  |
| --- |
| pcap\_loop(dev\_handle\_pcap,-1,Packethandler,NULL); |

其中第一个参数是已经打开的设备接口句柄，第三个参数是数据报处理函数，该函数有固定的格式，其定义如下。

|  |
| --- |
| void Packethandler(u\_char \*argument,const struct pcap\_pkthdr\* packet\_header,const u\_char\* packet\_content) |

pcap\_loop函数捕获到一个数据报后会自动调用Packethandler函数对数据报进行处理。并且自动将该数据报的信息作为参数传递给Packethandler函数。可以在Packethandler函数中增加对数据报进行解析的代码，其中第二个参数packet\_header为libpcap在捕获数据报时增加的辅助头部信息，包括时间戳、数据报长度等，第三个参数packet\_content即为捕获到的数据报。至此，就可以按照前面介绍的数据报的解析过程对数据报进行解析。解析过程同图6-6一致。

1. 关闭设备

程序结束时注意要关闭设备，释放资源。

|  |
| --- |
| pcap\_close(dev\_handle\_pcap); |

#### 注意事项

1. 程序在最后编译时必须加上编译选项-lpcap，表示需要用到libpcap的库函数，编译才能通过。
2. libpcap只能捕获数据报而不能发送数据报（windows下的winpcap可以发送数据报），如果要想进行数据报的发送操作则需要用到Linux下的另一个开发包libnet，通过该开发包提供的一些接口函数来填充和发送数据报。

### 使用tcpdump捕获数据报

#### tcpdump介绍

tcpdump 是一款功能非常强大的网络数据截取分析工具。它支持针对网络层、协议、主机、网络或端口的过滤，支持基于正则表达式的过滤方式，具有强大的功能和灵活的截取策略。主要用于网络的分析、维护、统计、检测等方面，例如定位网络瓶颈、统计网络流量使用情况等。更重要的是tcpdump是开源项目，提供了源代码，公开了接口，因此具备很强的可扩展性，对于网络维护和管理者来说都是非常有用的工具。大多数的Linux操作系统都将tcpdump集成在内。因为tcpdump需要将网卡设置为混杂模式，所以需要root权限才能执行tcpdump命令。tcpdump通过将网络接口设置为混杂模式，绕过标准TCP/IP堆栈对数据报的截取分析。

#### tcpdump命令

该命令使用语法如下。

tcpdump [ -adeflnNOpqRStuvxX ] [ -c count ] [ -C file\_size ] [ -F file ]

[ -i interface ] [ -m module ] [ -r file ] [ -s snaplen ] [ -T type ] [ -w file ]

[ -E algo:secret ] [ expression ]

可以通过对 tcpdump 选项、表达式进行组合，以及对参数进行设定，从大量的网络数据中过滤出有用的信息来分析网络问题。

#### tcpdump选项（option）

tcpdump可用选项如表6-1所示。

##### 表6-1 tcpdump 选项

|  |  |
| --- | --- |
| 选项 | 含义 |
| -a | 把网络地址和广播地址转换成名字； |
| -dd | 将匹配数据报代码以c程序格式输出； |
| -ddd | 将匹配的数据报代码以十进制形式输出 |
| -e | 输出数据链路层头部信息； |
| -f | 将外部的Internet地址以数字的形式打印出来； |
| -l | 对标准输出进行缓冲，可以用于捕捉数据的同时查看数据； |
| -n | 不把网络地址转换成主机名； |
| -N | 不输出主机名中的域名部分，例如“tcpdump.org”只输出“tcpdump”； |
| -O | 不运行数据报匹配模板的优化器； |
| -p | 不将网络接口设置成混杂模式； |
| -q | 快速输出，只输出较少的协议信息； |
| -S | 将tcp的序列号以绝对值形式输出，而不是相对值； |
| -t | 不在输出的每一行打印时间戳； |
| -u | 输出未解码的NFS句柄； |
| -v | 输出比较详细的信息，例如在ip包中可以包括ttl和服务类型的信息； |
| -vv | 输出详细的报文信息； |
| -vvv | 输出更为详细的报文信息； |
| -c count | 指定监听数据报数量，当收到指定的包的数目后推出tcpdump； |
| -C file\_size | 限定数据报写入文件大小； |
| -F file | 从指定的文件中读取过滤正则表达式,忽略命令行中的表达式； |
| -i interface | 指定监听网络接口； |
| -m module | 打开指定的SMI MIB组件； |
| -r file | 从指定的文件中读取数据报(这些数据报一般通过-w选项产生)； |
| -s snaplen | 从每个数据报中读取最开始的snaplen个字节，而不是默认的68个字节；将截取的数据报直接解释为指定类型的报文。 |
| -T type | 把捕获的数据报解析成指定的type。目前已知的类型有：rpc，rtp，rtcp ，vat和wb 。 |
| -w file | 将捕获的数据报写入文件，不分析和打印数据报； |
| -E algo:secret | 用algo:secret解密IPsec ESP数据报. |

#### tcpdump表达式（expression）

tcpdump利用正则表达式来过滤数据报。如果没有指定表达式，则捕获全部数据报， 否则，只捕获满足表达式为真的数据报。表达式有3类常用的关键字：type、dir和proto。

1. type 是指定类型的关键字，包括host、net和port。缺省为host。

例如：tcpdump host 192.168.1.27 表示捕获ip为192.168.1.27的主机收发的所有数据报。

1. dir 是指定数据报传输方向的关键字，包括src、dst、src or dst和src and dst。缺省为src or dest。

例如：tcpdump dst net 192.168.1.28表示捕获目标网络地址为192.168.1.28的所有数据报。

1. proto 是指定协议的关键字，包括ether、fddi、tr、ip、ip6、arp、rarp、decnet、tcp和udp。

例如：tcpdump udp表示捕获所有udp协议的数据报。

1. 其他重要的关键字还有gateway、broadcast、less、greater，3种逻辑运算符（取非运算符 “not”或者“!”， 与运算符 “and ”或者“&& ”；或运算 是“or ”或者“|| ”）将这些关键字组合就能灵活地构造出各种复杂的过滤条件。

#### 简单示例

1. 捕获arp 和udp包：

tcpdump arp or udp

1. 捕获除了主机名为alice之外的所有IP包：

tcpdump ip host not alice

1. 捕获主机IP为192.168.1.102接受和发出的telnet数据报：

tcpdump tcp and host 192.168.1.102 and port 23

1. 捕获来源IP地址为192.168.1.1，目地IP地址为192.168.1.102，端口为80的数据报：

tcpdump tcp and src host 192.168.1.1 and dst host 192.168.1.102 and port 80

1. 捕获主机IP为192.168.1.1和192.168.1.102之间传递的所有tcp包：

tcpdump ip and tcp and \(\(src host 192.168.1.1 and dst host 192.168.1.102\) or \(dst host 192.168.1.1 and src host 192.168.1.102\)\)

#### 注意事项

在表达式中用到括号时，一定要在括号前加反斜杠。以上示例只是利用正则表达式组合出一些简单的过滤条件，读者完全可以根据自己的需求利用正则表达式组合出更复杂的过滤条件。