网络嗅探器

2012-12

课程内容

- 嗅探器概述
- Libpcap 简介、安装与程序编译
- 利用 Libpcap 进行网络嗅探的工作流程
- Libpcap 主要函数简介
- 实验题目

嗅探器概述--定义与作用

- **定义：**网络嗅探也叫网络侦听，指的是使用特定的网络协议来分解捕获到的数据包，并根据对应的网络协议识别对应数据片断。
- 作用
  - ✓管理员可以用来监听网络的流量情况
  - ✓开发网络应用的程序员可以监视程序的网络情况
  - ✓黑客可以用来刺探网络情报

嗅探器概述--网卡的工作方式

- 网卡的四种接受模式
  - 广播模式；
  - 组播模式；
  - 直接模式；
  - 混杂模式；

通常，网卡的缺省配置是支持前三种模式。

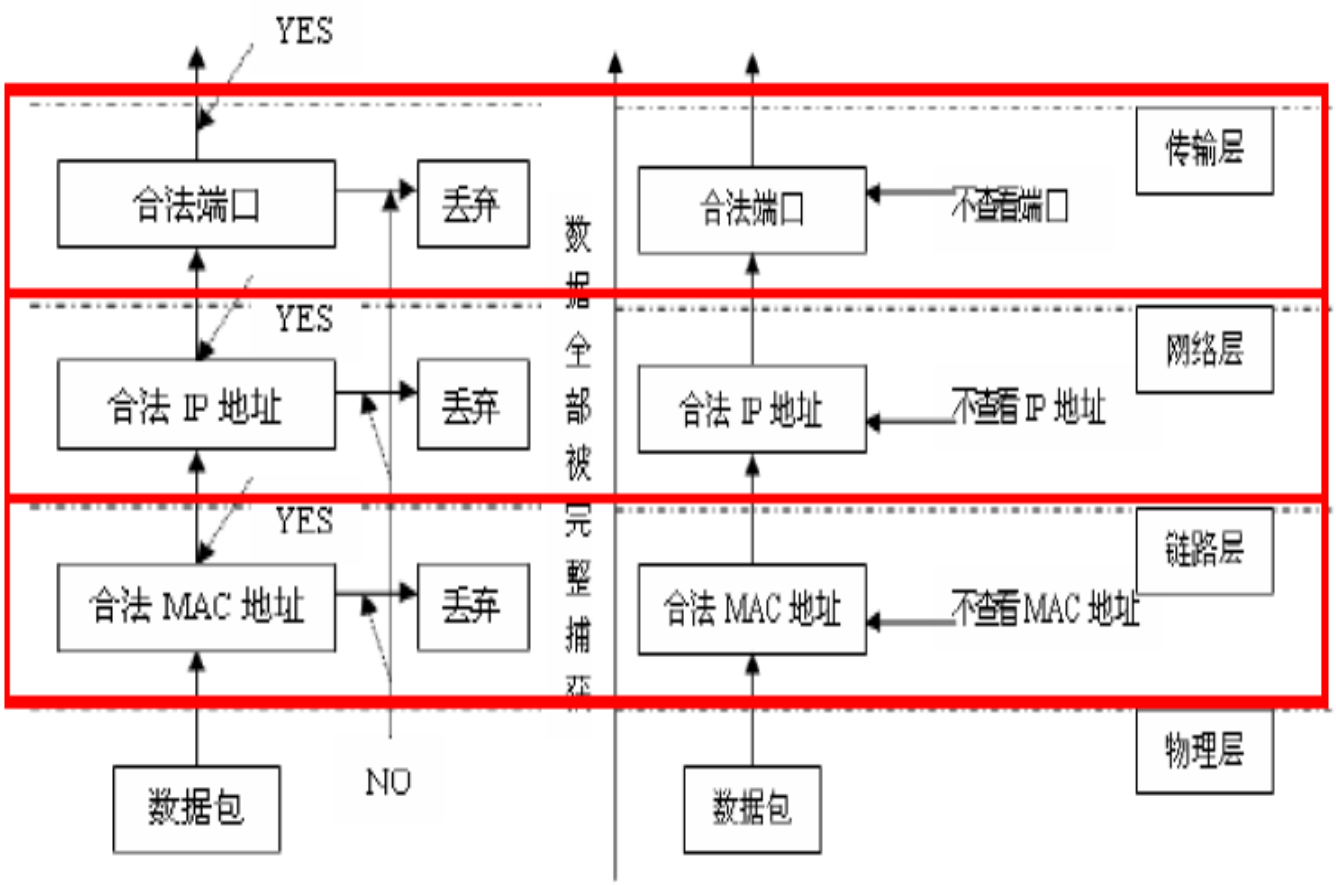
为了监听网络上的流量，必须设置为混杂模式

嗅探器概述--网络的类型

- 共享式网络
  - 通过网络的所有数据包发往每一个主机
  - 最常见的是通过 HUB 连接起来的子网

- 交换式网络
  - 通过交换机连接网络
  - 由交换机构造一个 MAC 地址 - 端口映射表
  - 发送包的时候，只发到特定的端口上

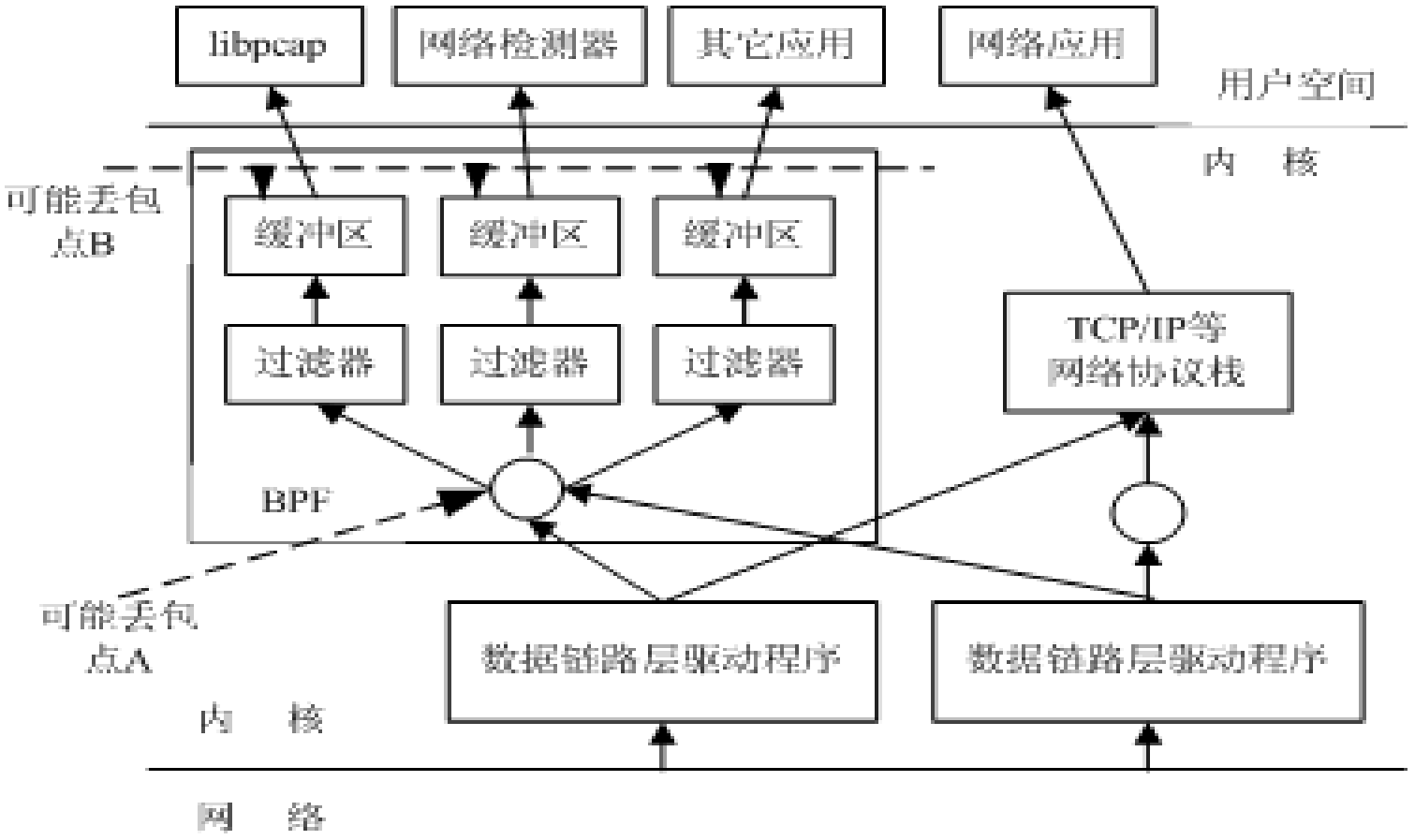
嗅探器概述--数据包的处理流程



嗅探器概述--实现机制

解决方法：添加一个直接与网卡驱动程序接口的驱动的模块，作为网卡驱动程序应用程序的"中间人"。

Libpcap 简介



BPF结构原理

Libpcap 的安装

1. 判断系统是否有安装 Libpcap 库

   rpm –qa|grep pcap
2. 到 www.tcpdump.org 下载 Libpcap 安装包
3. 将源码包 libpcap.tar.gz 拷贝到 /usr 目录下
4. 释放源码包的内容

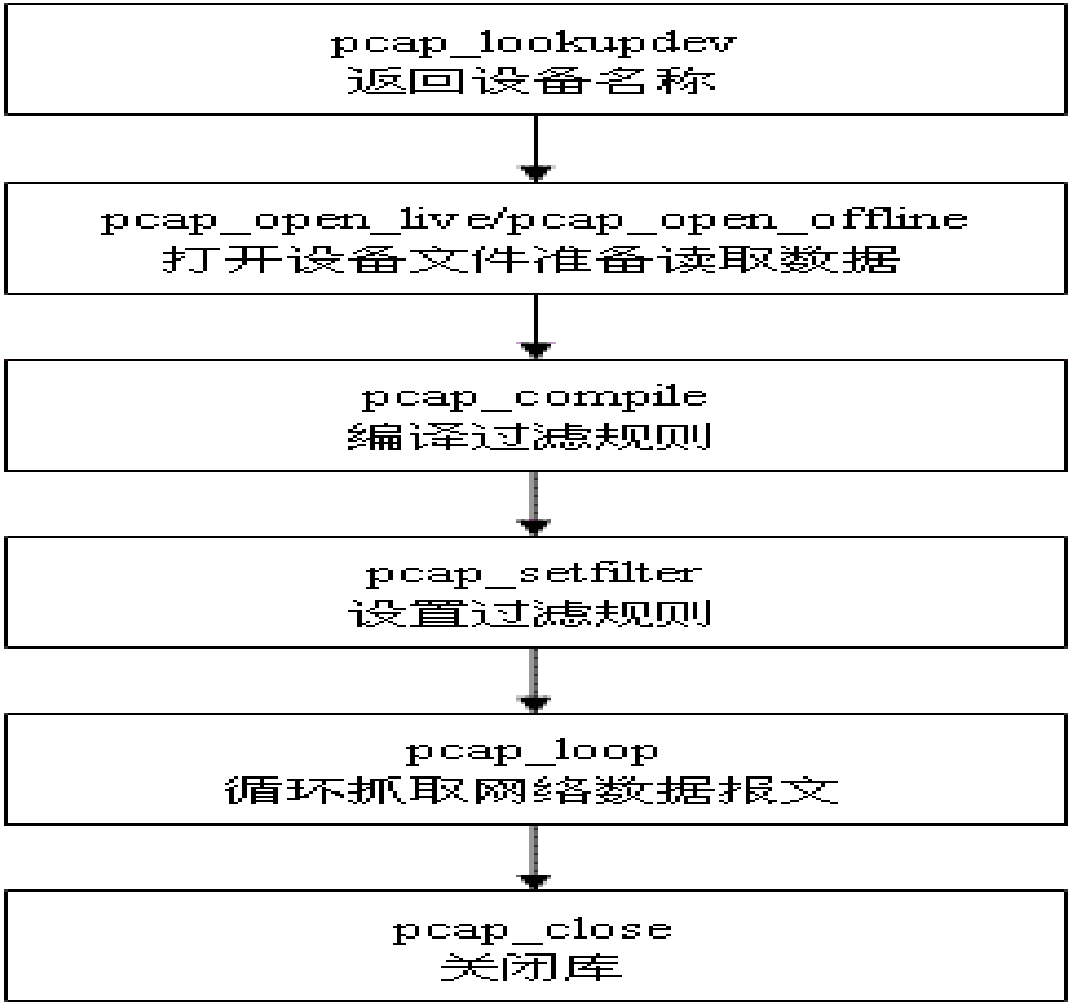
   tar xvzf libpcap.tar.gz
5. 检查系统配置生成 Libpcap 的文件 make

   ./configure
6. 编译 Libpcap 库： make
7. 安装库文件： make install

Libpcap　程序编译

gcc –o 目标文件 源文件 *– l pcap*

使用 Libpcap 编写程序流程



Libpcap API 介绍

- pcap_lookupdev( )
- pcap_lookupnet( )
- pcap_open_live( )
- pcap_setfilter( )
- pcap_compile( )
- pcap_next( )
- pcap_loop( )

pcap_lookupdev()

- char *pcap_lookupdev(char *errbuf)
- return a pointer to a network device suitable for use with *pcap_open_live()* and *pcap_lookupnet()*
- return *NULL* indicates an error
- reference: lookupdev.c

pcap_lookupnet()

- int pcap_lookupnet(

  const char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp,

  char *errbuf)
- determine the *network number* and *mask* associated with the network device
- return *-1* indicates an error
- reference: lookupnet.c

pcap_open_live( ) (1/2)

- pcap_t *pcap_open_live(

  const char *device, int snaplen,

  int promisc, int to_ms, char *errbuf)
- obtain a packet capture descriptor to look at packets on the network
- snaplen: maximum number of *bytes* to capture

pcap_open_live( ) (2/2)

- promisc: true, set the interface into *promiscuous mode*; false, only bring packets intended for you
- to_ms: *read timeout* in milliseconds; *zero*, cause a read to wait forever to allow enough packets to arrive
- return *NULL* indicates an error

pcap_compile() (1/2)

- int pcap_compile(pcap_t *p,
  struct bpf_program *fp, char *str,
  int optimize, bpf_u_int32 netmask)
- compile the str into a filter program
- str: filter string
- optimize: 1, optimization on the resulting code is performed; 0, false

pcap_compile( ) (2/2)

- netmask: specify network on which packets are being captured
- return *-1* indicates an error

规则的设定

- 规则是由标识和修饰符与逻辑符组成的。
- 修饰符
  - ✓ 确定方向的修饰符；
  - ✓ 确定类别的修饰符；
  - ✓ 确定协议的修饰符；
- 逻辑符
  - ✓ and 或 &&
  - ✓ not 或 !
  - ✓ or 或 ||

类型修饰符

- host: 指定操作的对象是一台主机 如： host 192.168.0.1
- net : 指定操作的对象是一个网络 如： net 192.168.0.0
- port: 指定操作的对像是一个端口 如： port 21

注：如果没有指定规则 缺省类型是 host

方向修饰符

- **src:　指定 IP 包中的源地址**
- **dst ：指定 IP 包中的目的地址**
- **dst or src**
- **dst and src**

注：如不指定，系统默认为 dst or src

协议修饰符

用于指定特定协议的数据包，主要包括IP 、 TCP 、 UDP 、 ARP 等协议类型。

注：如果没有指定监听的协议类型，系统将监控所有协议的数据包

过滤规则设置实例

1. 截获主机 192.168.0.2 主机收发的所有数据包

   src and host 192.168.0.2

1. 获取 192.168.0.168 接收或发送的 telnet 数据包

   tcp and port23 and host 192.168.0.168

3. 截获 192.168.0.2 与 192.168.0.3 的通信信息

   192.168.0.2 and 192.168.0.3

pcap_setfilter( )

- int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
- specify a filter program
- return *-1* indicates an error
- pcap_filter.c

pcap_next()

- const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
- read the next packet
- return *NULL* indicates an error
- pcap_next.c
- timestamp.c

Pcap_Loop

```
void my_callback(u_char *useless,const struct
    pcap_pkthdr* pkthdr,const u_char*    packet) {

        //do stuff here with packet
}
```

int **pcap_loop(pcap_t *p ,int cnt,pcap_handler callback,u_char * user);**

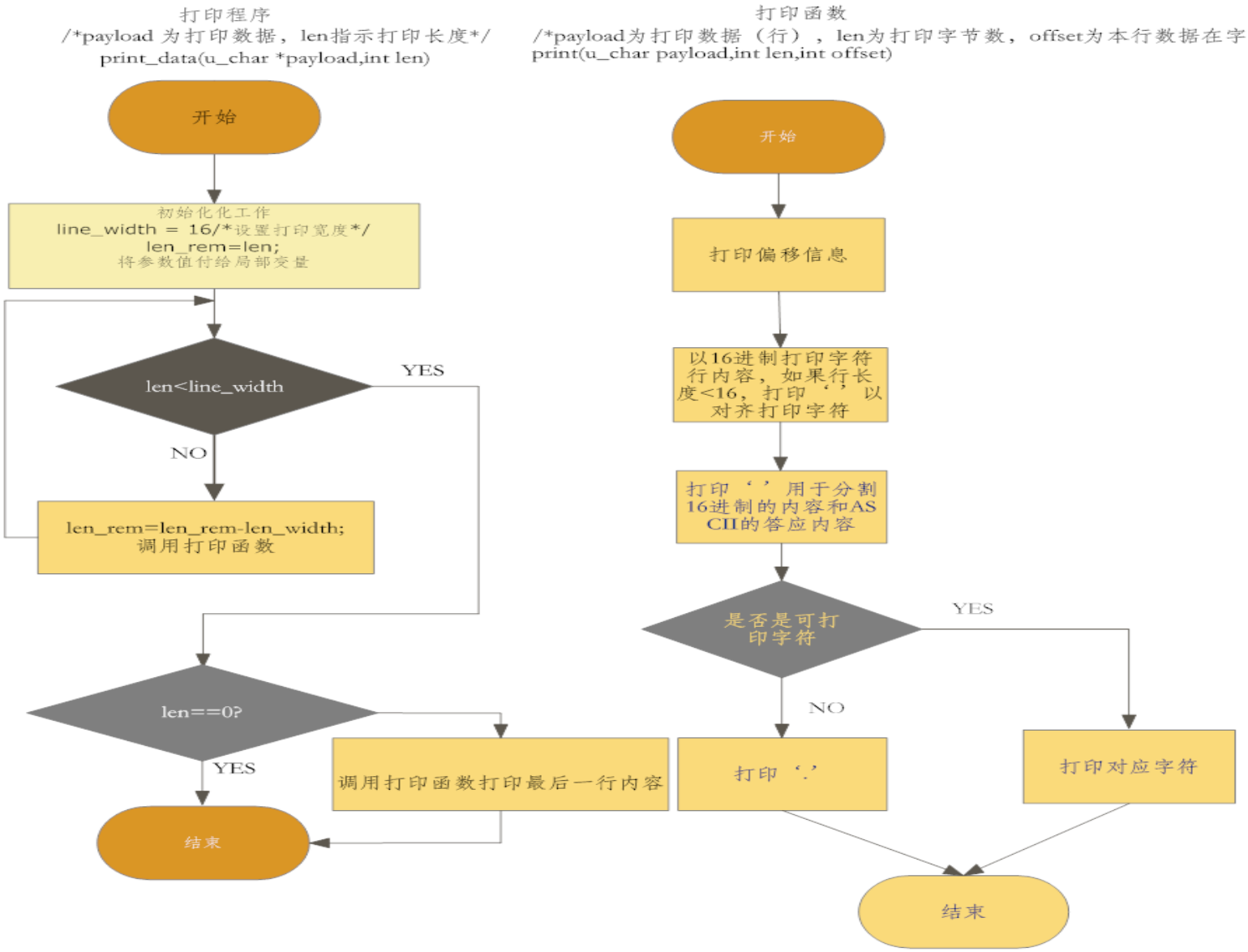
实验题目

使用 Libpcap 库捕获局域网中的 IP 包，要求:

1. 打印数据包的源与目的物理地址；
2. 打印源ＩＰ与目的ＩＰ地址；
3. 打印出上层协议类型；
4. 如果上层协议为 TCP 或 UDP 协议，打印目的与源端口信息；
5. 如果上层协议为 TCP 或 UDP 协议，将数据以 16 进制与 ASCII 的两种方式同时打印出来，不可打印字符以‘.’代替；

00000　47 45 54 20 2f 20 48 54　54 50 2f 31 2e 31 0d 0a　GET / HTTP/1.1..

思路


打印程序
/*payload 为打印数据，len指示打印长度*/
print_data(u_char *payload,int len)
开始
初始化化工作
line_width = 16/*设置打印宽度*/
len_rem=len;
将参数值付给局部变量
len<line_width
YES
NO
len_rem=len_rem-len_width;
调用打印函数
len==0?
YES
结束
调用打印函数打印最后一行内容
打印函数
/*payload为打印数据（行），len为打印字节数，offset为本行数据在字符串中的偏移*/
print(u_char payload,int len,int offset)
开始
打印偏移信息
以16进制打印字符行内容，如果行长度<16，打印‘’以对齐打印字符
打印‘’用于分割16进制的内容和ASCII的答应内容
是否是可打印字符
YES
NO
打印‘.’
打印对应字符
结束

What is an ethernet header?

From #include<netinet/if_ether.h>

```
struct ether_header {
   u_int8_t ether_dhost[ETH_ALEN]; /* 6 bytes destination */
   u_int8_t ether_shost[ETH_ALEN]; /* 6 bytes source  addr */
   u_int16_t ether_type;                    /* 2 bytes ID type */
} __attribute__ ((__packed__));
```

Some ID types:

```
#define ETHERTYPE_IP    0x0800 /* IP */
#define ETHERTYPE_ARP 0x0806 /* Address resolution */
```

Is this platform independent?

NO!

So we may need to swap bytes to read the data.

```
struct ether_header *eptr;      /* where does this go? */
```

eptr = (struct ether_header *) packet;

```
/* Do a couple of checks to see what packet type we have..*/
if (ntohs (eptr->ether_type) == ETHERTYPE_IP) {
    printf("Ethernet type hex:%x dec:%d is an IP packet\n",
        ntohs(eptr->ether_type), ntohs(eptr->ether_type));

} else if (ntohs (eptr->ether_type) == ETHERTYPE_ARP) {
    printf("Ethernet type hex:%x dec:%d is an ARP packet\n",
        ntohs(eptr->ether_type), ntohs(eptr->ether_type));
}
```