

数据包嗅探与欺骗

实验报告

|  |  |
| --- | --- |
| 学生姓名 | maybeLocalhost |
| 学 号 |  |
| 专业班级 |  |
| 指导教师 |  |
| 学 院 | 计算机学院 |
| 完成时间 | 2020.10 |

目录

[一、实验目的 1](#_Toc54084519)

[二、实验设备与环境 1](#_Toc54084520)

[三、实验内容 1](#_Toc54084521)

[四、实验过程 2](#_Toc54084522)

[1. 使用scapy进行数据包嗅探 2](#_Toc54084523)

[2. 使用scapy实现特定数据包嗅探功能 4](#_Toc54084524)

[3. Traceroute 6](#_Toc54084525)

[4. 使用scapy实现数据包欺骗功能 7](#_Toc54084526)

[5. 使用scapy实现数据包嗅探与欺骗功能 8](#_Toc54084527)

[6. 使用pcap进行数据包嗅探与欺骗 9](#_Toc54084528)

[五、实验总结 13](#_Toc54084529)

# 一、实验目的

数据包Sniffing和Spoofing是网络通信中的两大威胁，理解这两种威胁对于理解网络中的安全措施至关重要。目前有许多数据包Sniffing和Spoofing工具，如Wireshark，Tcpdump，Netwox等，其中一些工具被安全专家和攻击者广泛使用。本实验帮助了解这些工具的工作原理，即如何在软件中实现数据包Sniffing和Spoofing。

# 二、实验设备与环境

Ubuntu 20.0.4，Python 3.8.5，gcc version 9.3.0，Visual Studio Code

# 三、实验内容

1. **使用scapy进行数据包嗅探**

* 有许多工具可以实现数据包Sniffing和Spoofing，但大多数只提供了固定的功能。本任务使用scapy来实现自定义的数据包处理功能。Scapy是一个python库，不仅仅是一个数据包处理工具，也可以作为构建其他包Sniffing & Spoofing工具的模块。请参照《Sniffing\_Spoofing.pdf》中的描述尝试使用scapy库进行IP数据包的Sniffing。

1. **使用scapy编程实现特定数据包嗅探功能**

* 本任务要求利用scapy库来编写python程序实现数据包的Sniffing。请参照《Sniffing\_Spoofing.pdf》中的描述，捕获特定类型的数据包，如捕获指定IP的TCP数据包等。

1. **Traceroute**

* 该部分利用TTL（Time-To-Live）的特性，使用Scapy计算数据包出发点与目标点之间的距离，该距离使用路由的结点数量表示。

1. **使用scapy编程实现数据包欺骗功能**

* 作为数据包spoofing工具，scapy允许任意修改IP数据包内的字段值。如《Sniffing\_Spoofing.pdf》中所述，本任务要求利用scapy来伪造ICMP请求数据包发送给另一台主机，并使用wireshark工具来观察ICMP请求及响应数据包。

1. **使用spacy实现数据包先嗅探再欺骗功能**

* 在之前的任务的基础上，本任务使用两台虚拟机A和B，并且结合使用数据包Sniffing以及Spoofing功能。任务要求是当A ping任意主机时，B都能接收到ICMP请求数据包，并返回相应的ICMP响应包给A。

1. **使用pcap进行数据包嗅探与欺骗**

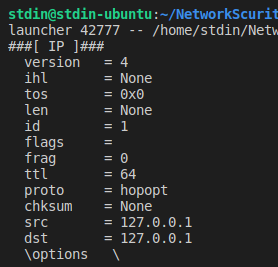
* 如《Sniffing\_Spoofing.pdf》所述，本任务同样需要完成上述几个功能，不过此处不使用scapy而是使用C语言的pcap库来实现Sniffing & Spoofing。

# 四、实验过程

先使用scapy建立一个数据包，检测实验环境是否已经搭建好：

*#!/usr/bin/python3*  
from scapy.all import \*  
a = IP()  
a.show()

运行结果如下，可以看出实验环境已经搭建完成：

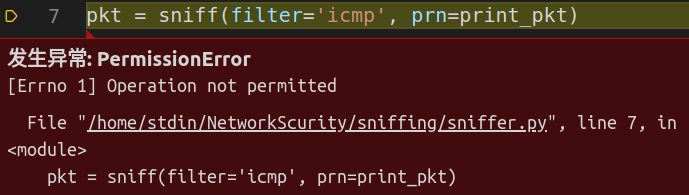


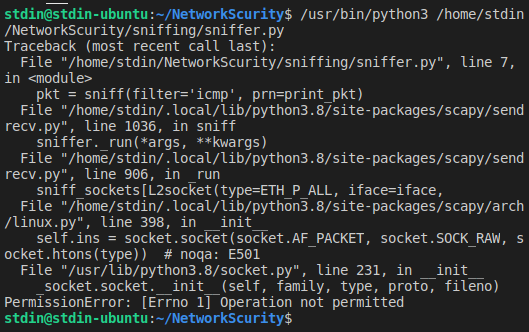
## 1. 使用scapy进行数据包嗅探

sniffer.py：

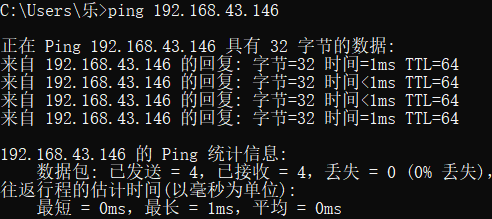
*#!/usr/bin/python*  
from scapy.all import \*  
  
**def** print\_pkt(pkt):   
 pkt.show()  
  
pkt = sniff(filter='icmp', prn=print\_pkt)

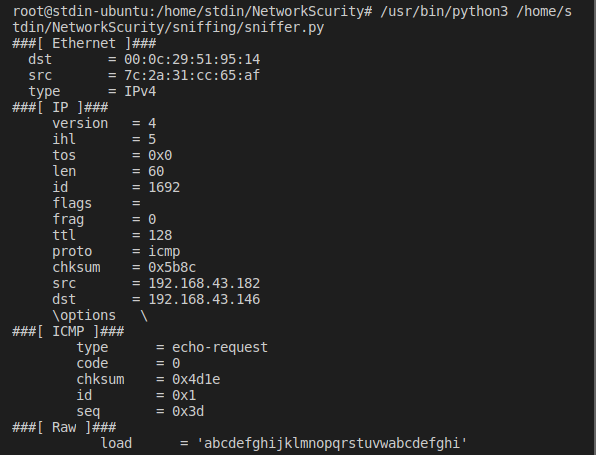
分别在root与非root环境下运行sniffer.py。在非root环境下运行sniffer.py时，由于权限不够会产生如下报错：





在root环境下运行sniffer.py，本地主机对该虚拟机使用ping命令，可以看到运行sniffer.py的虚拟机将循环打出ICMP包的相关信息：

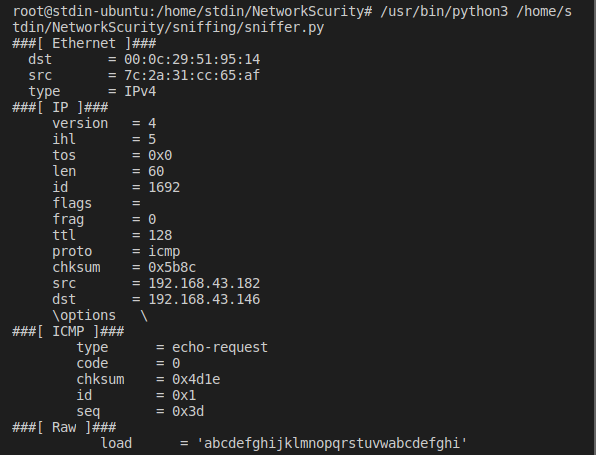




## 2. 使用scapy实现特定数据包嗅探功能

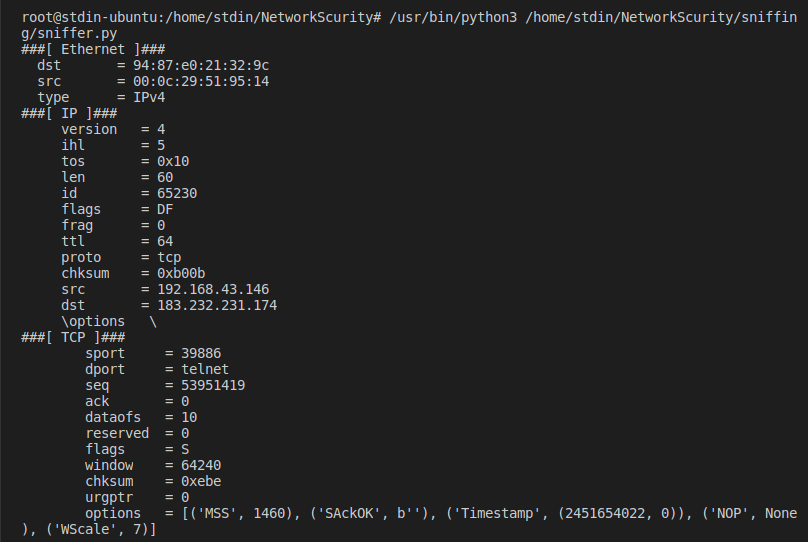
scapy filter默认情况下使用BPF作为自己的句法，可通过该句法设置某些条件的过滤器，以捕获特定的数据包。

* 捕获ICMP数据包
* *#!/usr/bin/python*   
  from scapy.all import \*   
     
  **def** print\_pkt(pkt):   
   pkt.show()   
     
  pkt = sniff(filter='icmp', prn=print\_pkt)

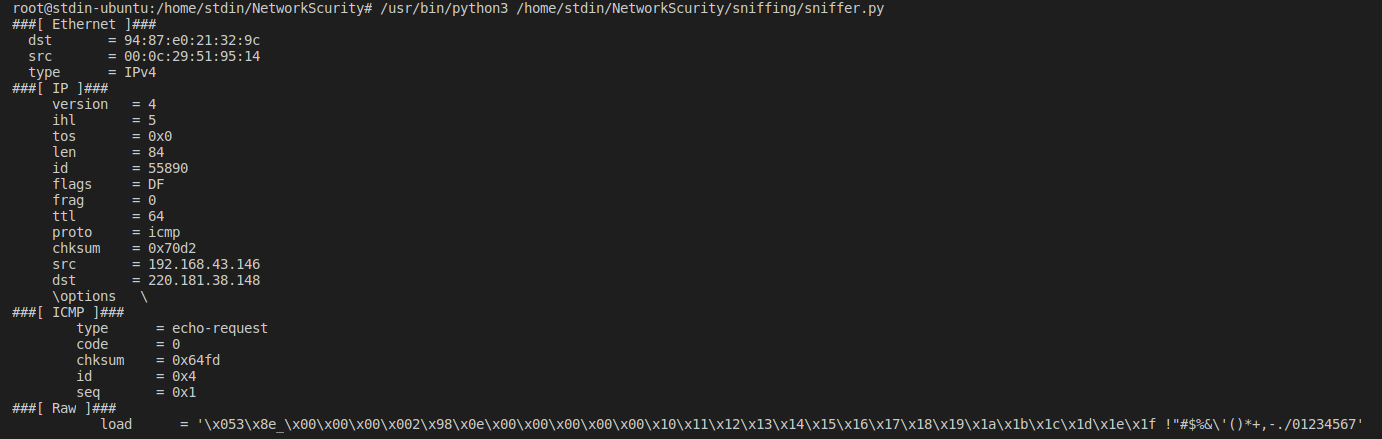


* 捕获来自特定IP且目标端口号为23的任何TCP数据包。
* *#!/usr/bin/python*  
  from scapy.all import \*  
    
  **def** print\_pkt(pkt):   
   pkt.show()  
    
  pkt = sniff(filter = 'tcp and port 23 and host 192.168.43.146', prn = print\_pkt)





* 捕获发去或来自于特定子网的数据包. 可以选择任意子网, 但是不要选择那些 VM 可以接触到的。
* *#!/usr/bin/python*   
  from scapy.all import \*   
     
  **def** print\_pkt(pkt):   
   pkt.show()   
     
  pkt = sniff(filter = 'host 220.181.38.148', prn = print\_pkt)



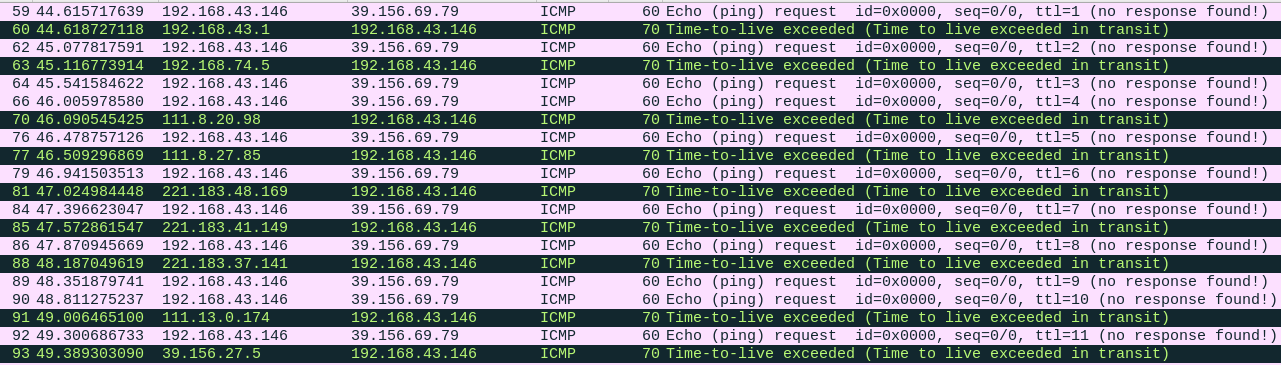
## 3. Traceroute

该部分使用Scapy计算数据包出发点与目标点之间的距离，该距离使用路由的结点数量表示。原理是利用TTL（Time-To-Live）的特性，每经过一个路由器，TTL值就会减去1。当TTL减到0时，该数据包就会被丢弃，同时该路由器会回发一个ICMP的错误信息。但由于每次的路由路径可能会不一样，所以该方法只能估计源主机与目的主机之间的距离。

可以写一个循环，向目标点不断地发送ICMP数据包。一开始设置TTL为为1，即发出的ICMP数据包经历一个路由结点后，就会被抛弃。

from scapy.all import \*  
a = IP()  
b = ICMP()  
a.dst = "39.156.69.79"  
**for** i **in** range(30):  
 a.ttl = i + 1  
 p = a / b  
 send(p)

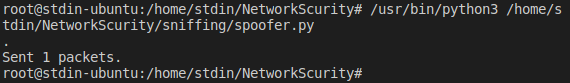
运行该程序可以在Wireshark中查看到达目的主机39.156.69.79前经过了那些路由结点：

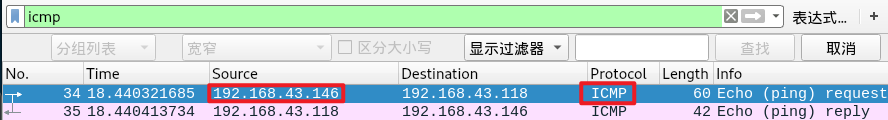


## 4. 使用scapy实现数据包欺骗功能

使用该Ubuntu虚拟机伪装ICMP数据包，并发送给同一子网下的另一台虚拟机，通过Wireshark去观察：

*#!/usr/bin/python*   
from scapy.all import \*   
a = IP()   
a.dst = '192.168.43.118'   
b = ICMP()   
p = a / b   
send(p)





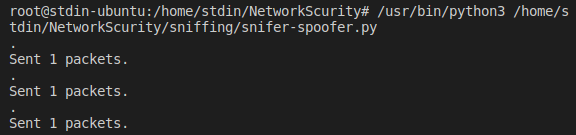
## 5. 使用scapy实现数据包嗅探与欺骗功能

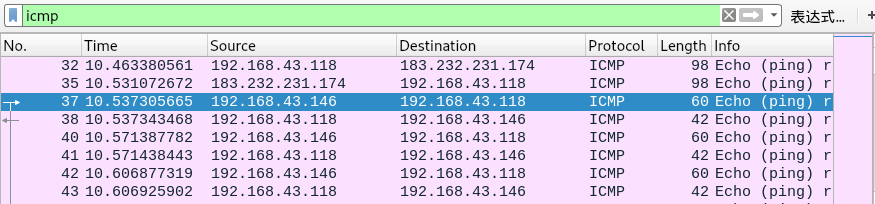
实现数据包的嗅探和欺骗功能需要两台虚拟机，让虚拟机A去ping一个IP，虚拟机B运行嗅探程序，监控LAN中的通信状态。理论上，只要A发出了ICMP包，B就会感知，并且发出伪造的响应。

*#!/usr/bin/python*  
from scapy.all import \*  
**def** spoof\_pkt(pkt):  
 a = IP()   
 b = ICMP()   
 a.dst = '192.168.43.118'  
 send(a / b)   
  
pkt = sniff(filter='icmp and host 192.168.43.118', prn=spoof\_pkt)

spoof\_pkt(pkt)

本次实验中，IP为192.168.43.118的虚拟机将尝试去ping www.baidu.com，而IP为192.168.43.146的虚拟机将监听LAN中的通信，当其嗅探到192.198.43.118的ICMP请求包时，就会自动发送一个ICMP响应包。





## 6. 使用pcap进行数据包嗅探与欺骗

该实验主要用到了libpcap中的几个函数：

* pcaplookupdev()：该函数用于查找网络设备，返回可被pcapopen\_live()函数调用的网络设备名指针。
* pcap*open*live()：该函数用于打开网络设备，并且返回用于捕获网络数据包的数据包捕获描述字。对于此网络设备的操作都要基于此网络设备描述字。
* pcap\_lookupnet()：该函数获得指定网络设备的网络号和掩码。
* pcap\_compile()：该函数用于将用户制定的过滤策略编译到过滤程序中。
* pcap\_setfilter()：该函数用于设置过滤器。
* pcap\_loop()：该函数用来捕获数据包。
* pcap\_close()：该函数用于关闭网络设备，释放资源。

在伪造回复数据包时，其长度需大于等于接收到的数据包，即将收到的数据包中的Data复制后回送。否则在受害机中会出现 8 bytes from 1.2.3.4: icmp\_seq=17 ttl=128 (truncated) 问题。因此，构造一个ICMP包：

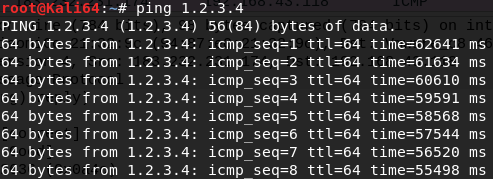
**struct** spoofed\_packet   
{   
 **struct** ip iph;   
 **struct** icmp icmph;   
};

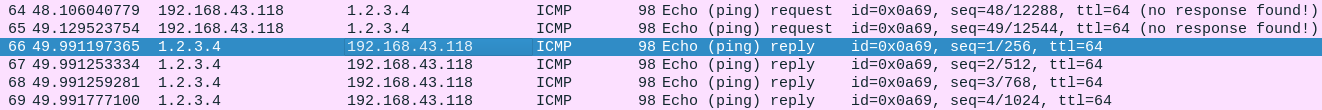
构造欺骗包：

*//计算ip标头偏移量*   
iph = (**struct** ip\*)(packet + SIZE\_ETHERNET);   
size\_ip = iph->ip\_hl\*4; *// ip头大小*  
   
**if** (iph->ip\_p != IPPROTO\_ICMP || size\_ip < 20) { *// disregard other packets*   
 **return**;   
}   
   
*//计算icmp标头偏移量*   
icmph = (**struct** icmp\*)(packet + SIZE\_ETHERNET + size\_ip);   
   
*//利用数据报的长度构造欺骗包并分配内存*   
char buf[htons(iph->ip\_len)];   
**struct** spoofed\_packet \*spoof = (**struct** spoofed\_packet \*) buf;   
   
*//通过将请求包中的所有内容复制到伪造包，初始化*   
memcpy(buf, iph, htons(iph->ip\_len));   
   
*//修改ip头*   
(spoof->iph).ip\_src = iph->ip\_dst;   
(spoof->iph).ip\_dst = iph->ip\_src;   
   
*//重新计算校验和，可以将它保留为0，因为原始套接字会计算这个值。*   
(spoof->iph).ip\_sum = 0;   
   
*//修改icmp头*   
(spoof->icmph).icmp\_type = ICMP\_ECHOREPLY;   
*//icmp\_code=0 即回显应答*   
(spoof->icmph).icmp\_code = 0;   
   
(spoof->icmph).icmp\_cksum = 0; *// 应先将值设为0以重新计算*   
(spoof->icmph).icmp\_cksum = in\_cksum((unsigned short \*) &(spoof->icmph), **sizeof**(spoof->icmph));

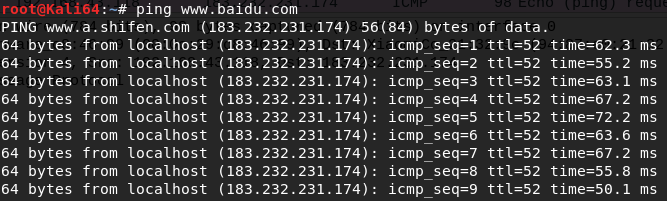
首先运行该程序，然后在另一虚拟机中去ping一个源地址并不存在的主机1.2.3.4，可以看到在这一虚拟机中接收到了reply数据包，在Wireshark中也可以看到发送的伪造数据包。

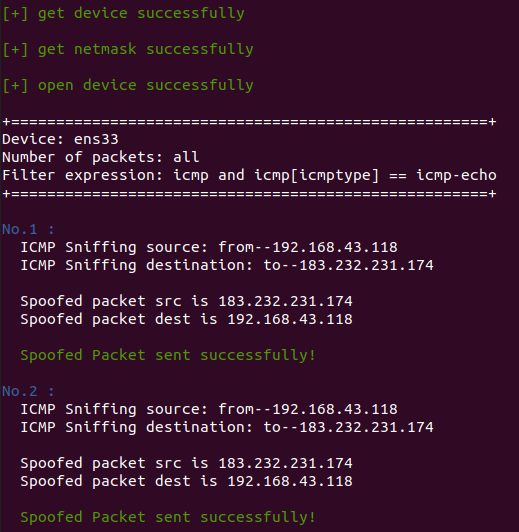


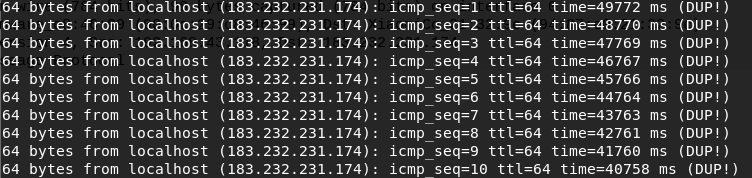




然后尝试去ping一个正常的IP，可以发现出现了DUP!，即DUPLICATE，也就是ping的时候收到多个重复值回应。







**Question 1**：嗅探程序中对应的库调用顺序关键部分代码：

*//获取网络接口*   
dev = pcap\_lookupdev(errbuf);   
**if** (dev == NULL){   
 fprintf(stderr, "Couldn't find default device: %s\n",errbuf);   
 exit(EXIT\_FAILURE);   
}   
   
*//获取与捕获设备关联的网络号和掩码*   
**if** (pcap\_lookupnet(dev, &net, &mask, errbuf) == -1) {   
 fprintf(stderr, "Couldn't get netmask for device %s: %s\n",dev, errbuf);   
 net = 0;   
 mask = 0;   
}   
   
*//打开网络接口*   
handle = pcap\_open\_live(dev, SNAP\_LEN, 1, 0, errbuf); *//0表示非混杂模式，任何其他值表示混杂模式，第三个参数指timeout指定需要等待的毫秒数，超过这个数值后，第3步获取数据包的这几个函数就会立即返回。0表示一直等待直到有数据包到来*   
**if** (handle == NULL) {   
 fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);   
 exit(EXIT\_FAILURE);   
}   
   
*//确保捕获的是以太网设备*   
*//返回链路层的类型，DLT\_EN10MB： 以太网（10Mb, 100Mb, 1000Mb, 或者更高）*   
**if** (pcap\_datalink(handle) != DLT\_EN10MB) {   
 fprintf(stderr, "%s is not an Ethernet\n", dev);   
 exit(EXIT\_FAILURE);   
}   
   
*//编译filter*   
**if** (pcap\_compile(handle, &fp, filter\_exp, 0, net) == -1) {   
 fprintf(stderr, "Couldn't parse filter %s: %s\n",filter\_exp, pcap\_geterr(handle));   
 exit(EXIT\_FAILURE);   
}   
   
*//设置filte*   
**if** (pcap\_setfilter(handle, &fp) == -1) {   
 fprintf(stderr, "Couldn't install filter %s: %s\n",filter\_exp, pcap\_geterr(handle));   
 exit(EXIT\_FAILURE);   
}   
   
*//开始循环抓包*   
pcap\_loop(handle, num\_packets, call\_back, NULL); *//类似python中的sniff函数*   
   
*//释放资源*   
pcap\_freecode(&fp);   
pcap\_close(handle);

**Question 2**：嗅探程序必须使用root用户来运行，如果没有root privilege，执行程序会发生segmentation fault。因为程序访问了其没有权限访问的内存，我认为具体原因应该是程序访问网卡需要root权限。

**Question 3**：修改 pcap\_open\_live 的第三个参数 int promisc 可以开启和关闭混杂模式。若开启混杂模式，该虚拟机A则可以监听到另一个虚拟机B ping 任何主机的信息。若关闭，则只能接受目标IP是A自身的ICMP包。

# 五、实验总结

通过本次实验，我学习了 scapy 和 pcap 的基本使用方法，加深了对IP、TCP、ICMP等协议数据包格式的了解，并能够编写简单的嗅探欺骗程序。其中，嗅探阶段主要是利用网卡的混杂模式，接收其他目的IP不是自身的数据包，从而实现嗅探，同时还可利用过滤规则BPF进行数据包的过滤，以嗅探指定规则的数据包。欺骗阶段则主要是对数据包格式的正确构造，以及如何将构造完成的数据包发送到受害机上。

最后，根据实验过程，我们可以根据该攻击的两个阶段来制定相应的预防措施。由于绝大多数路由器和交换机都添加了安全机制，一般情况下只向目标转发数据包，网卡开启混杂模式不能再轻易地嗅探到网络中的数据包。另一方面，可以通过网络层安全传输协议，如IPsec，对数据包进行加密，使得即使攻击者获取了数据包，也无法获取其中的内容。除此外，还可以通过使用随机化的初始序列号、避免采用基于IP地址的信任策略等措施来预防数据包欺骗。