

分工：

set1部分：刘世杰 202122460170 全部完成

set2部分：牛中原 202100460123 全部完成

Task1.1 sniffing packets

实现sniff需要以下几个步骤：

- 找到监听端口
- 过滤想要监听的内容
- 编写回调函数

sniffer.py文件如下

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

print("start")

def print_pkt(pkt):
    print_pkt.num_packets+=1
    print("\n=====packet:
{}=====\\n".format(print_pkt.num_packets))
    pkt.show()

print_pkt.num_packets=0

# pkt = sniff(iface='br-78bb728863e7',filter='tcp && src host 10.9.0.1 && dst port
23',prn=print_pkt)

pkt=sniff(iface=['br-78bb728863e7','enp0s3'],filter='icmp',prn=print_pkt)
```

task1.1A

sniffer.py捕捉到的数据包：

```
[09/23/23]seed@VM:~/../volumes$ vim sniffer.py
[09/23/23]seed@VM:~/../volumes$ sudo python3 sniffer.py
start
=====packet: 1=====
###[ Ethernet ]###
  dst      = 02:42:8a:09:00:06
  src      = 02:42:6c:78:db:3e
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 64
  id       = 63480
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x2e98
  src      = 10.9.0.1
  dst      = 10.9.0.6
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0x87fe
  id       = 0x1
  seq      = 0x1
###[ Raw ]###
  load     = '\xe2\xaa\xee\x00\x00\x00\x07\x1c\t\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&'()*+,-./01234567'
=====packet: 2=====
```

如果没有root权限运行的话会产生PermissionError:

```
seed@VM: ~/../volumes$ python3 sniffer.py
start
Traceback (most recent call last):
  File "sniffer.py", line 16, in <module>
    pkt=sniff(iface=['br-78bb728863e7','enp0s3'],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 894, in _run
    sniff_sockets.update(
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 895, in <genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  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
[09/23/23]seed@VM:~/../volumes$
```

这是因为在Linux系统上，以普通用户身份运行的程序通常没有权限访问网络设备。

task1.1B

- **Capture only the ICMP packet** 使用下面这行代码进行sniff

```
pkt=sniff(iface=['br-78bb728863e7','enp0s3'],filter='icmp',prn=print_pkt)
```

捕获数据包的结果截图在task1.1A中，可以看到IP下的数据类型是ICMP

- **Capture any TCP packet that comes from a particular IP and with a destination port number 23.**

换成下面这行代码sniff；

```
pkt = sniff(iface=['br-78bb728863e7','enp0s3'],filter='tcp && src host 10.9.0.1 && dst port 23',prn=print_pkt)
```

在attacker容器中telnet 10.9.0.6可以产生tcp数据包：

```
[09/23/23]seed@VM:~/../volumes$ docksh b
root@VM:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.

Ubuntu 20.04.1 LTS

ef90a5ca34a5 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

Documentation:  https://help.ubuntu.com
Management:    https://landscape.canonical.com
* Support:      https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Sat Sep 23 09:26:16 UTC 2023 from 10.9.0.1 on pts/1
seed@ef90a5ca34a5:~$

options = [('NOP', None), ('NOP', None), ('timestamp', (634129223, 3131941

^C[09/23/23]seed@VM:~/../volumes$ sudo python3 -i sniffer.py
start

=====packet: 1=====

###[ Ethernet ]###
dst      = 02:42:0a:09:00:06
src      = 02:42:6c:70:db:3e
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x10
len      = 53
id       = 50515
flags    = DF
frag     = 0
ttl      = 64
proto    = tcp
chksum   = 0x6147
src      = 10.9.0.1
dst      = 10.9.0.6
\options \
###[ TCP ]###
sport    = 38006
dport    = telnet
seq      = 206371060
ack      = 2642597066
dataofs  = 8
reserved = 0
flags    = PA
window   = 501
```

- Capture packets comes from or to go to a particular subnet

换成下面这行代码sniff：

```
pkt = sniff(iface=['br-78bb728863e7','enp0s3','lo'],filter='dst net
128.230.0.0/16',prn=print_pkt)
```

使用scapy构建两个数据包发送并对捕获到的数据包进行验证可以看出，成功对一段subnet进行监听，发送和接收都可以成功捕获：

```
seed@V...  seed@V...  seed@V...  seed@V...  seed@V...
[09/23/23]seed@VM:~/../volumes$ sudo python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a=IP()
>>> a.src='128.230.1.1'
>>> a.dst='10.9.0.1'
>>> b=ICMP()
>>> send(a/b)
.
Sent 1 packets.
>>> a.src='10.9.0.1'
>>> a.dst='128.230.128.230'
>>> send(a/b)
.
Sent 1 packets.
>>> █
```

```

^C[09/23/23]seed@VM:~/../volumes$ sudo python3 sniffer.py
start

=====packet: 1=====
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:16:d8:4b
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 28
id       = 11816
flags    =
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0xc0c8
src      = 10.9.0.1
dst      = 128.230.1.1
\options \
###[ ICMP ]###
type     = echo-reply
code     = 0
chksum   = 0xffff
id       = 0x0
seq      = 0x0

=====packet: 2=====
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:16:d8:4b
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 28
id       = 1
flags    =
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x6f0a
src      = 10.9.0.1
dst      = 128.230.128.230
\options \

```

Task1.2 spoofing ICMP packets

使用task1中的sniffer.py文件对attacker端口监听，在python的交互式命令行中发送不同源地址的ICMP数据包到10.9.0.1，如下截图表明，成功对echo request进行响应，说明构建成功：

```

[09/23/23]seed@VM:~/../volumes$ sudo python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a=IP()
>>> a.src='1.2.3.4'
>>> a.dst='10.9.0.1'
>>> b=ICMP()
>>> send(a/b)
.
Sent 1 packets.
>>> a.src='10.9.0.6'
>>> send(a/b)
.
Sent 1 packets.
>>> █

```

```
[09/23/23]seed@vm:~/.../volumes$ vim sniffer.py
[09/23/23]seed@vm:~/.../volumes$ sudo python3 sniffer.py
start

=====packet: 1=====

###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:16:d8:4b
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 28
id       = 50588
flags    =
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0xa735
src      = 10.0.0.1
dst      = 1.2.3.4
\options \
###[ ICMP ]###
type     = echo-reply
code     = 0
chksum   = 0xffff
id       = 0x0
seq      = 0x0

=====packet: 2=====

###[ Ethernet ]###
dst      = 02:42:0a:09:00:06
src      = 02:42:6c:70:db:3e
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 28
id       = 49928
flags    =
```

Task1.3 Traceroute

在本实验中，测试到github.com(IP地址为：20.205.243.166) 的路由数，ttl从1开始，不断增加，重复发送ICMP包，通过wireshark查看何时会接收到响应包 程序分为自动发包程序ttl.py和检查路由数量程序traceroute.py代码分别如下：**ttl.py**

```
#!/usr/bin/env python3

from scapy.all import *

a = IP(dst = '20.205.243.166')

b = ICMP()

p = a/b

for i in range(1,64):
    a.ttl=i
    send(a/b)
```

traceroute.py

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
    if pkt[1].src=='10.0.2.4':
        print("ttl = {}".format(pkt[1].ttl))
        return
```

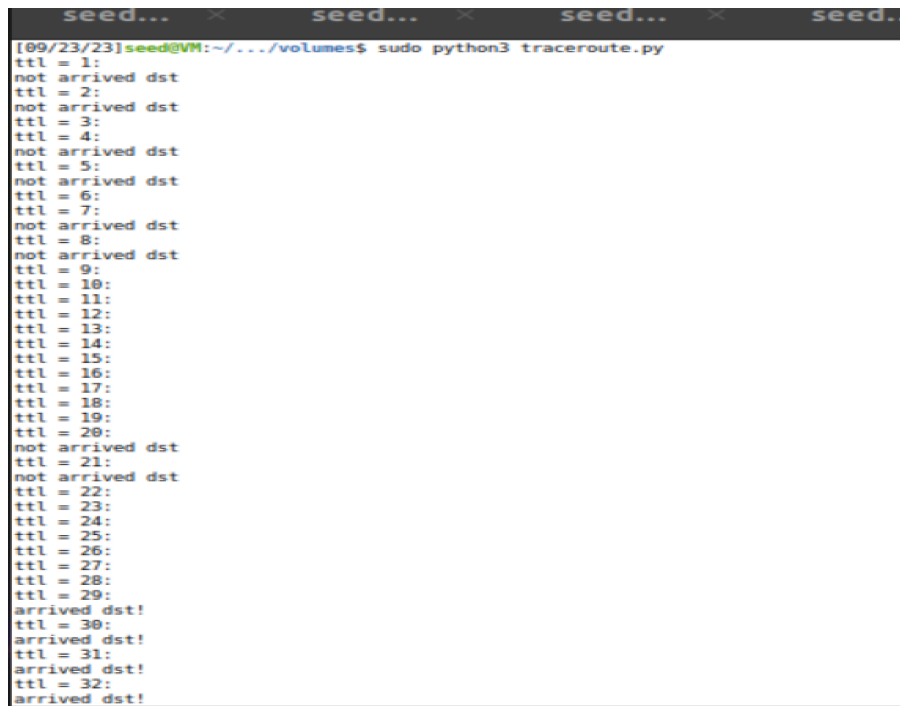
```

if pkt[1].src=='20.205.243.166':
    print("arrived dst!")
else:
    print("not arrived dst")

pkt=sniff(iface=['br-78bb728863e7','enp0s3'],filter='icmp',prn=print_pkt)

```

最终得到结果，当ttl=29时获得来自github.com的响应：



```

[09/23/23]seed@VM:~/.../volumes$ sudo python3 traceroute.py
ttl = 1:
not arrived dst
ttl = 2:
not arrived dst
ttl = 3:
not arrived dst
ttl = 4:
not arrived dst
ttl = 5:
not arrived dst
ttl = 6:
not arrived dst
ttl = 7:
not arrived dst
ttl = 8:
not arrived dst
ttl = 9:
not arrived dst
ttl = 10:
not arrived dst
ttl = 11:
not arrived dst
ttl = 12:
not arrived dst
ttl = 13:
not arrived dst
ttl = 14:
not arrived dst
ttl = 15:
not arrived dst
ttl = 16:
not arrived dst
ttl = 17:
not arrived dst
ttl = 18:
not arrived dst
ttl = 19:
not arrived dst
ttl = 20:
not arrived dst
ttl = 21:
arrived dst!
ttl = 22:
arrived dst!
ttl = 23:
arrived dst!
ttl = 24:
arrived dst!
ttl = 25:
arrived dst!
ttl = 26:
arrived dst!
ttl = 27:
arrived dst!
ttl = 28:
arrived dst!
ttl = 29:
arrived dst!
ttl = 30:
arrived dst!
ttl = 31:
arrived dst!
ttl = 32:
arrived dst!

```

Task1.4 Sniffing and-then Spoofing

```

#!/usr/bin/python
from scapy.all import *

def send_packet(pkt):

    if ICMP in pkt and pkt[ICMP].type == 8:
        print("request: src {} dst {}".format(pkt[1].src,pkt[1].dst))
        ip = IP(src=pkt[1].dst,dst=pkt[1].src)
        if pkt[1].dst=='8.8.8.8':
            ip.src='8.8.8.9'
        icmp = ICMP(type=0,id=pkt[2].id,seq=pkt[2].seq)

        data=pkt[3].load
        newpkt = ip/icmp/data
        print("response: src {} dst {}".format(pkt[1].dst,pkt[1].src))
        send(newpkt,verbose=0)

interfaces = ['enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=send_packet)

```

ping的三个地址如下：

```

seed... x seed... x seed... x seed... x seed... x seed... x
root@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=19.0 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=17.5 ms
^C
--- 1.2.3.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 17.516/18.277/19.038/0.761 ms
root@VM:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.9: icmp_seq=1 ttl=64 time=18.3 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=109 time=52.3 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, +1 duplicates, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 18.295/35.293/52.291/16.998 ms
root@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
^C
--- 10.9.0.99 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3071ms
pipe 3
root@VM:/#

```

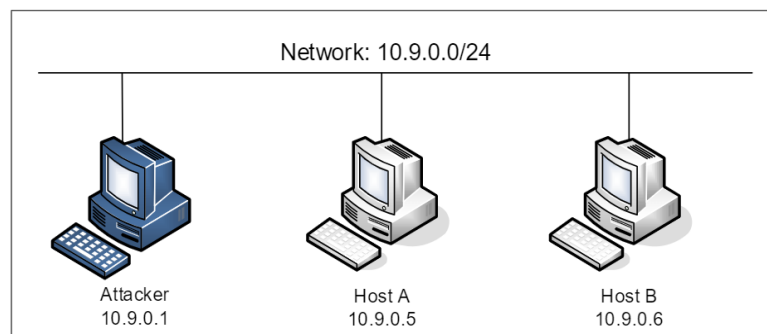
程序运行打印结果如下：

```

response: src 8.8.8.8 dst 10.0.2.4
^C[09/23/23]seed@VM:~/.../volumesvimn3 sniffing_and_spoofing.py
[09/23/23]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing.py
request: src 10.0.2.4 dst 1.2.3.4
response: src 1.2.3.4 dst 10.0.2.4
request: src 10.0.2.4 dst 1.2.3.4
response: src 1.2.3.4 dst 10.0.2.4
request: src 10.0.2.4 dst 8.8.8.8
response: src 8.8.8.8 dst 10.0.2.4

```

- ping 1.2.3.4时，需要把请求传给网关，网关接收到请求后伪造一个响应传给ping的src
- ping 10.9.0.99 时，属于子网内部请求，不经过网关，所以不会被检测程序发现这个请求，然而子网内又没有这个地址，所以发生unreachable错误



- ping 8.8.8.8 时，虽然在真实网络环境中也可ping通，但是返回的实际上是我们伪造的请求包，特意将8.8.8.8修改为8.8.8.9可以看出这一点。

Gain and experience

1. 熟悉scapy的使用方法
2. 熟悉了网络数据包的结构和转发过程
3. 了解到许多docker和linux下的命令
4. 掌握利用scapy捕获修改数据包的内容

Task 2:Writing programs to sniff and spoof packets

Task 2.1:Writing packet sniff program

- sniff 程序需要能够识别捕获的数据包的类型，并提取出关键信息，如源目IP地址，报文类型等
- 为达到上述目的，需要定义以太网帧的头部，IP数据报的头部等信息，我们可以使用结构体定义：

```
/* IP Header */
struct ipheader {
    unsigned char    iph_ihl:4, //IP header length
                    iph_ver:4; //IP version

    unsigned char    iph_tos; //Type of service
    unsigned short int iph_len; //IP Packet length (data + header)
    unsigned short int iph_ident; //Identification
    unsigned short int iph_flag:3, //Fragmentation flags
                    iph_offset:13; //Flags offset

    unsigned char    iph_ttl; //Time to Live
    unsigned char    iph_protocol; //Protocol type
    unsigned short int iph_chksum; //IP datagram checksum
    struct in_addr    iph_sourceip; //Source IP address
    struct in_addr    iph_destip;  //Destination IP address
};

/* Ethernet header */
struct ethheader {
    u_char ether_dhost[6]; /* destination host address */
    u_char ether_shost[6]; /* source host address */
    u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};
```

如上述，定义了IP头部和以太网帧头部的各个字段。

- 捕获数据包时，最重要的函数之一是`pcap_loop()`，它需要一个回调函数，以便每次嗅探到数据包时调用该函数，该回调函数负责完成数据包解析，打印字段等任务。

我们将回调函数命名为`got_packet ()`。

主函数部分

在函数主体，我们需要做的事情有如下几个：

- 设置要嗅探的网络设备
- 设置过滤器

- 创建嗅探会话
- 保持运行嗅探程序

```
handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
```

用于打开一个嗅探会话并获取会话句柄；

```
if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
    fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp,
pcap_geterr(handle));
    return(2);
}
if (pcap_setfilter(handle, &fp) == -1) {
    fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp,
pcap_geterr(handle));
    return(2);
}
```

用于编译一个过滤器表达式，并将结果保存在过滤器程序中，以便后续在抓包过程中使用。

```
pcap_loop(handle, -1, got_packet, NULL);

pcap_close(handle); //Close the handle
```

运行嗅探程序以及关闭。

Capture the ICMP packets between two specific hosts

设置过滤器如下：

```
char filter_exp[PCAP_FILTER_SIZE]; /* Filter string */
struct bpf_program fp; /* The compiled filter */
//char filter_exp[] = "port 23"; /* The filter express
//char filter_exp[] = "icmp";
char filter_exp[]="icmp";

bpf_u_int32 mask; /* Our netmask */
bpf_u_int32 net; /* Our IP */
struct pcap_pkthdr header; /* The header that pcap gives
const u_char *packet; /* The actual packet */

/* Define the device */
```

用另一个终端进入docker HostA, ping 10.9.0.1 · 嗅探结果如下：

```

[09/23/23]seed@VM:~/Desktop$ sudo ./sniff
or-87102073d3bd
start to sniff...
    From: 10.9.0.5
    To: 10.9.0.1
Protocol: ICMP

    From: 10.9.0.1
    To: 10.9.0.5
Protocol: ICMP

    From: 10.9.0.5
    To: 10.9.0.1
Protocol: ICMP

    From: 10.9.0.1
    To: 10.9.0.5
Protocol: ICMP

```

Capture the TCP packets with a destination port number in the range from 10 to 100

过滤器设置如下：

```

struct bpf_program tp, /* The compiled filter */
//char filter_exp[] = "port 23"; /* The filter expression */
//char filter_exp[] = "icmp";
char filter_exp[]="tcp and dst portrange 10-100";

bpf_u_int32 mask; /* Our netmask */
bpf_u_int32 net; /* Our IP */
struct pcap_pkthdr header; /* The header that pcap gives us */
const u_char *packet; /* The actual packet */

/* Define the device */
dev = pcap_lookupdev(errbuf);
printf("%s\n", dev);
if (dev == NULL) {
    printf("Error: %s\n", errbuf);
}

```

从docker HostA 发送telnet连接请求命令（TCP类型），得到结果如下：

```
[09/23/23]seed@VM:~/Desktop$ sudo ./sniff
br-87102073d3bd
start to sniff...
    From: 10.9.0.5
    To: 10.9.0.1
Protocol: TCP

    From: 10.9.0.5
    To: 10.9.0.1
Protocol: TCP

    From: 10.9.0.5
    To: 10.9.0.1
Protocol: TCP

    From: 10.9.0.5
    To: 10.9.0.1
Protocol: TCP
```

Sniffing Passwords

我们可以借助scapy库，过滤出telnet的报文，并且打印报文原始内容

```
1 from scapy.all import *
2 def print_pkt(pkt):
3     pkt.show()
4 print(sniff(iface="br-87102073d3bd", filter="tcp port 23", prn=print_pkt))
```

从docker Host A使用telnet命令尝试连接10.9.0.1，这时，在docker HostA中输入的字符，全部会作为数据报发送到10.9.0.1，也能被嗅探到，这时在嗅探程序上，就能看到输入的用户名及密码。

```
#### Raw #####
load = 'Password: '
```

```
#### Ethernet #####
dst = 02:42:92:e6:2e:29
src = 02:42:0a:09:00:05
type = IPv4
```

```
#### IP #####
version = 4
ihl = 5
tos = 0x10
len = 52
id = 51578
flags = DF
frag = 0
```

```
#### Raw #####
load = 'd'
```

```
#### Ethernet #####
dst = 02:42:0a:09:00:05
src = 02:42:92:e6:2e:29
type = IPv4
```

```
#### IP #####
version = 4
ihl = 5
tos = 0x10
len = 52
id = 52042
flags = DF
frag = 0
```

```
###[ Raw ]###
load      = 'e'

###[ Ethernet ]###
dst       = 02:42:92:e6:2e:29
src       = 02:42:0a:09:00:05
type     = IPv4
###[ IP ]###
version   = 4
ihl       = 5
tos       = 0x10
len       = 52
id        = 51573
flags     = DF
frag      = 0
ttl       = 64
```

answer questions:

1, Question: Please use your own words to describe the sequence of the library calls that are essential for sniffer programs.

Answer:

```
dev = pcap_lookupdev(errbuf)
pcap_lookupnet(dev, &net, &mask, errbuf)
handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf)
pcap_compile(handle, &fp, filter_exp, 0, net)
pcap_setfilter(handle, &fp)
pcap_loop(handle, -1, got_packet, NULL);
pcap_close(handle); //Close the handle
```

大致为如上几步：确定网络设备，打开网络设备，设置过滤器，进行嗅探，关闭嗅探程序。

2, Question: Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

Answer:

嗅探数据包的过程很可能涉及到隐私数据，这部分数据除非由特权级的指令执行，否则不能访问，这涉及到隐私数据安全的问题，当用户不是特权级时，不能监听网络设备。

3, Question: Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in pcap open live() turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this

mode is on and off? Please describe how you can demonstrate this.

Answer:

当 promiscuous mode (混杂模式) 开启时，网络接口会接收到所有经过的数据包，无论这些数据包的目标地址是否匹配当前主机的 MAC 地址。而当 promiscuous mode 关闭时，网络接口仅接收与当前主机相关的数据包。

Task 2.2 Spoofing

Task 2.2 A: Write a spoofing program.

- 写一个spoof程序，与前一个任务相同，都需要定义数据报的头部信息，但不同的是前者的目的是识别，后者的目的是创建一个新的数据报。

我们定义几个头部的结构体：

```
/* Ethernet header */
struct ethheader {
    u_char ether_dhost[6];    /* destination host address */
    u_char ether_shost[6];    /* source host address */
    u_short ether_type;        /* IP? ARP? RARP? etc */
};

/* IP Header */
struct ipheader {
    unsigned char    iph_ihl:4, //IP header length
                    iph_ver:4; //IP version

    unsigned char    iph_tos; //Type of service
    unsigned short int iph_len; //IP Packet length (data + header)
    unsigned short int iph_ident; //Identification
    unsigned short int iph_flag:3, //Fragmentation flags
                    iph_offset:13; //Flags offset

    unsigned char    iph_ttl; //Time to Live
    unsigned char    iph_protocol; //Protocol type
    unsigned short int iph_chksum; //IP datagram checksum
    struct in_addr    iph_sourceip; //Source IP address
    struct in_addr    iph_destip; //Destination IP address
};

/* ICMP Header */
struct icmpheader {
    unsigned char icmp_type; // ICMP message type
    unsigned char icmp_code; // Error code
    unsigned short int icmp_chksum; //Checksum for ICMP Header and data
    unsigned short int icmp_id; //Used for identifying request
    unsigned short int icmp_seq; //Sequence number
};

/* UDP Header */
struct udpheader
```

```
{
    u_int16_t udp_sport;           /* source port */
    u_int16_t udp_dport;          /* destination port */
    u_int16_t udp_ulen;           /* udp length */
    u_int16_t udp_sum;            /* udp checksum */
};
```

我们可以构建并发送原始数据包，此过程需要如下步骤：

1. 创建原始套接字：使用 `socket()` 函数创建一个原始网络套接字，指定地址族为 `AF_INET` (IPv4)，套接字类型为 `SOCK_RAW`，协议类型为 `IPPROTO_RAW` (表示发送原始 IP 数据包)。
2. 设置套接字选项：使用 `setsockopt()` 函数设置套接字选项，将 `IPPROTO_IP` 和 `IP_HDRINCL` 作为参数传递给 `setsockopt()` 函数。`IP_HDRINCL` 表示在发送数据包时包含 IP 报头，而不是由系统自动添加 IP 报头。
3. 提供目标信息：将目标信息填充到 `sockaddr_in` 类型的结构体中，包括目标的地址族 (`AF_INET`) 和目标 IP 地址 (`iph_destip`)。
4. 发送数据包：使用 `sendto()` 函数发送数据包。数据包的内容是通过参数传递给该函数的 `ip` 结构体指针，数据包长度通过 `ntohs(ip->iph_len)` 获取。目标地址和目标地址结构体也作为参数传递给 `sendto()` 函数。
5. 关闭套接字：使用 `close()` 函数关闭所创建的原始套接字。

关键步骤如下：

```
struct sockaddr_in dest_info;
int enable = 1;

// Step 1: Create a raw network socket.
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
printf("sock: %d\n", sock);

// Step 2: Set socket option.
setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
           &enable, sizeof(enable));

// Step 3: Provide needed information about destination.
dest_info.sin_family = AF_INET;
dest_info.sin_addr = ip->iph_destip;

// Step 4: Send the packet out.
sendto(sock, ip, ntohs(ip->iph_len), 0,
       (struct sockaddr *)&dest_info, sizeof(dest_info));
close(sock);
```

函数的主体中，如果发送UDP报文，我们需要定义UDP的首部和IP的首部，并且给对应字段正确赋值。比如在UDP首部赋值时：

```
udp->udp_sport = htons(12345);
udp->udp_dport = htons(9090);
udp->udp_ulen = htons(sizeof(struct udphdr) + data_len);
udp->udp_sum = 0;
```

发送数据包时，可以在目的IP的docker监听网卡，就能监听到发送的数据包。

使用命令：

```
tcpdump -i eth0 -n
```

监听本docker的网卡。

在主机(attacker)运行spoofer程序，在HostA监听到如下内容

```
root@8948a8f2f487:/# tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
09:28:40.505181 IP 10.9.0.1.12345 > 10.9.0.5.9090: UDP, length 14
09:28:40.505200 IP 10.9.0.5 > 10.9.0.1: ICMP 10.9.0.5 udp port 9090 unreachable,
length 50
09:28:45.727362 ARP, Request who-has 10.9.0.1 tell 10.9.0.5, length 28
09:28:45.727446 ARP, Request who-has 10.9.0.5 tell 10.9.0.1, length 28
09:28:45.727454 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
09:28:45.727454 ARP, Reply 10.9.0.1 is-at 02:42:91:2c:04:d3, length 28
```

可以看到确实收到了UDP报文

Task 2.2 B: Spoof an ICMP Echo Request

- 当一个主机发送 ICMP Echo 请求时，它希望接收到一个 ICMP Echo Reply (回复) 消息作为响应。这样的请求通常用于测试与目标主机之间的连通性。例如，当你在命令行中使用 "ping" 命令时，实际上就是发送一个 ICMP Echo 请求，并等待目标主机返回 ICMP Echo Reply 响应。
- 在 ICMP 协议中，Echo 请求 (Type 8) 和 Echo 回复 (Type 0) 消息都需要计算校验和，校验和值设置到 ICMP 头部的 Checksum 字段中。

因此我们需要写一个校验和计算的函数，TCP校验和在伪报头上计算，其中包括 TCP报头和数据，加上IP报头的一部分。

计算校验和的部分内容如下：

```
while (nleft > 1) {
    sum += *w++;
    nleft -= 2;
```



```

}

/* treat the odd byte at the end, if any */
if (nleft == 1) {
    *(u_char *)&temp = *(u_char *)w ;
    sum += temp;
}

/* add back carry outs from top 16 bits to low 16 bits */
sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
sum += (sum >> 16);                // add carry
return (unsigned short)(~sum);

```

- 发送报文的过程与前一问类似，不同的是ICMP Echo需要构造ICMP报文头部，而前者是UDP首部。

构造UDP首部如下:

```

struct icmpheader *icmp = (struct icmpheader *)
    (buffer + sizeof(struct ipheader));
icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

// Calculate the checksum for integrity
icmp->icmp_chksum = 0;
icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
    sizeof(struct icmpheader));

```

任意设置一个源IP，即使可能不存在的IP，在HostA依然可以收到伪造的报文。

```

struct ipheader *ip = (struct ipheader *) buffer;
ip->iph_ver = 4;
ip->iph_ihl = 5;
ip->iph_ttl = 20;
ip->iph_sourceip.s_addr = inet_addr("1.1.1.1");
ip->iph_destip.s_addr = inet_addr("10.9.0.5");
ip->iph_protocol = IPPROTO_ICMP;
ip->iph_len = htons(sizeof(struct ipheader) +
    sizeof(struct icmpheader));

```

监听HostA网卡，得到如下结果：

```
root@8948a8f2f487:/# tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
01:36:43.102834 ARP, Request who-has 10.9.0.5 tell 10.9.0.1, length 28
01:36:43.102842 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
01:36:43.102850 IP 1.1.1.1 > 10.9.0.5: ICMP echo request, id 0, seq 0, length 8
01:36:43.102860 IP 10.9.0.5 > 1.1.1.1: ICMP echo reply, id 0, seq 0, length 8
01:36:48.143421 ARP, Request who-has 10.9.0.1 tell 10.9.0.5, length 28
01:36:48.143800 ARP, Reply 10.9.0.1 is-at 02:42:92:e6:2e:29, length 28
█
```

可以看到确实收到了来自**1.1.1.1**的**ICMP echo**报文

answer questions

4,Question: Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

Answer:

如果将 IP 包的长度字段设置为与实际包大小不一致的值，可能会导致网络传输过程中出现问题。接收方会根据 IP 包的长度字段来解析和处理数据，如果长度字段与实际包大小不匹配，可能会导致数据截断、解析错误或者被丢弃。

5,Question: Using the raw socket programming, do you have to calculate the checksum for the IP header?

Answer:

Raw Socket提供了对网络协议栈的底层访问，可以自定义 IP 包的内容和头部字段。在这种情况下，通常需要手动构建 IP 头部，并手动计算 IP 头部的检验和。

6,Question: Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

Answer:

这是因为 Raw Socket 提供了对协议栈的底层访问，使得开发者可以读写和修改低层网络协议的数据包。如果普通用户也能访问 Raw Socket，则可以导致系统安全风险，例如用户可以利用 Raw Socket 进行网络欺诈。如果在没有 root 权限的情况下尝试执行使用 Raw Socket 的程序，则通常会遇到权限不足的错误，无法创建 Raw Socket 和/或绑定到指定的网络接口上。原因是：进程需要 Root 权限才能打开 Raw Socket 以及发送/接收原始网络数据包。

Task 2.3: Sniff and then Spoof

- sniff 之后再 spoof 的思路很简单，回想起前面的两个内容，我们可以发现，每次sniff到了数据包，我们就会调用回调函数，解析数据包内容，那么这样我们自然想到，如果在解析数据包后面加上发送的过程，就能实现目的了。
- 在回调函数中，添加spoof中发送数据包函数的功能，主体的函数部分，仍然是打开嗅探会话，监听某个网络设备，每次抓到数据包，进入回调函数，我们spoof一个数据包。

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
               const u_char *packet)
```

回调函数中增加spoof的内容。

注意我们仍需计算首部检验和。

注意过滤器的设置：按照要求我们应如下设置

```
char filter_exp[] = "icmp[icmptype]==icmp-echo";
```

这个过滤器可以用来捕获 ICMP 类型为回显请求的数据包，即捕获发送给本机的 Ping 请求数据包。

got_packet()函数中加入spoof

```
printf("From: %s ", inet_ntoa(ip->iph_sourceip));
printf("To: %s ", inet_ntoa(ip->iph_destip));
if (ip->iph_protocol == IPPROTO_ICMP)
    printf("protocal: ICMP\n");
else
    printf("protocal: Others\n");

struct icmpheader *icmp_pkt = (struct icmpheader *) (packet + sizeof(struct ether_header));

if (ip->iph_protocol == IPPROTO_ICMP) {
    char buffer[1500];
    memset(buffer, 0, 1500);
```

在docker HostA ping 1.1.1.1(下半),虽然命令不可达，但是还是被嗅探到

```
[09/23/23]seed@VM:~/Share$ sudo docker exec -u 0 -it 8948a8f2f487 /bin/bash
root@8948a8f2f487:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.090 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.042 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.067 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.076 ms
64 bytes from 10.9.0.1: icmp_seq=5 ttl=64 time=0.086 ms
64 bytes from 10.9.0.1: icmp_seq=6 ttl=64 time=0.044 ms
^C
--- 10.9.0.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5195ms
rtt min/avg/max/mdev = 0.042/0.067/0.090/0.018 ms
root@8948a8f2f487:/# ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
From 192.168.250.250 icmp_seq=1 Destination Net Unreachable
From 192.168.250.250 icmp_seq=2 Destination Net Unreachable
From 192.168.250.250 icmp_seq=3 Destination Net Unreachable
_
```

嗅探到后发送数据包

```
listening on network card, ret: 0x5621351872c0...
try to compile filter...
try to set filter...
start to sniff...
From: 10.0.2.10 To: 1.1.1.1 protocol: ICMP
icmp id: 29, seq: 1
send tt source :1.1.1.1
send tt dest: 10.0.2.10
sock: 4
From: 10.0.2.10 To: 1.1.1.1 protocol: ICMP
icmp id: 29, seq: 2
send tt source :1.1.1.1
send tt dest: 10.0.2.10
sock: 4
From: 10.0.2.10 To: 1.1.1.1 protocol: ICMP
icmp id: 29, seq: 3
send tt source :1.1.1.1
send tt dest: 10.0.2.10
sock: 4
```

Gain and experience

- 通过这次实验，我基本了解了网络嗅探和伪造并发送数据包的原理和实践，使用pcap库进行嗅探的主要步骤是设置要嗅探的设备，过滤，创建会话，运行嗅探。都可以调用pcap库函数去做，学到了过滤器的语法设置。
- 再次加强了计算机网络相关知识的熟练度，通过去写和观察IP报文头部，UDP报文头部等结构的字段，对计算机网络的相关内容更加了解。
- 学习到很多Linux上的操作方法和知识，比如在docker之间的连接和通信，监听网卡，以特权身份进入docker，挂载文件目录等操作。
- 从实践的角度再次学习了不同报文的含义，比如ICMP Echo请求报文，telnet 连接请求报文，ICMP的ping 报文等，大致了解了如何构建并发送不同类型的数据报，其中检验和的计算是首部字段重要的一部分。
- 了解到很多涉及数据安全和隐私的地方，比如嗅探操作需要特权级命令，因为涉及到隐私数据的安全，让我更加明白了网络安全的重要性和意义。