Lab 3
Packet Sniffing and Spoofing Lab

Author：57119108 吴桐

Date：2022.8.15

Environment Setup using Container

在本次实验中，我们将使用虚拟机内的两个容器（host-10.9.0.5 和 seed-attacker），如图 1~2 所示。我们将在 attacker 容器上施行攻击，host 容器作为被攻击主机。

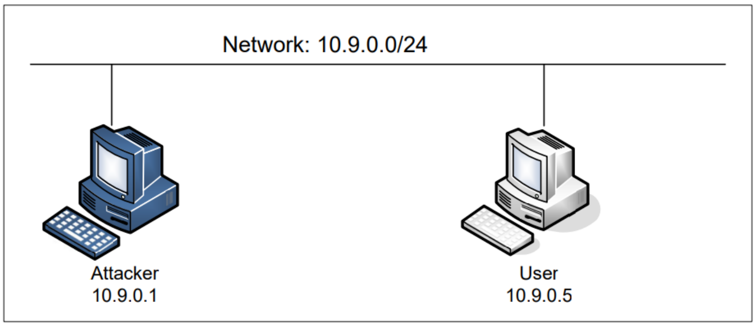

图 1

```
[08/15/22]seed@VM:~/.../Labsetup$ dockps
2596bef042a4  host-10.9.0.5
f19df684f462  seed-attacker
```

图 2

attacker 容器中设置有一个共享文件夹 volumes，用于在主机和攻击机之间传输文件。攻击者需要嗅探数据包，但是在容器中运行嗅探程序会出现问题。容器通过虚拟交换机连接到主机，因此它只能看到属于自己的流量，而不可能看到其他容器中的数据包。为了解决这个问题，我们在 attacker 主机上启动“host”模式，使其可以嗅探主机所有网络接口上的数据包。

当我们使用 Compose 文件创建容器时，会创建一个新的网络（10.9.0.0/24）来连接主机和多个容器，其中，主机的 IP 地址为 10.9.0.1。如图 3 所示，在宿主机中运行 ifconfig 命令查看新建网络中对应的网络接口名称，以便完成之后的实验。

```
[08/15/22]seed@VM:~$ ifconfig
br-a67c874d674a: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:6dff:fec3:f933  prefixlen 64  scopeid 0x20<link>
        ether 02:42:6d:c3:f9:33  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 42  bytes 5193 (5.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 3

Lab Tasks

Task 1: Using Scapy to Sniff and Spoof Packets

在此任务中，我们将基于 Scapy 实现数据包的嗅探和伪造。注意，在此任务中编写的所有 Python 程序都需要使用 root 权限来运行，因为数据包嗅探需要 root 权限。

Task 1.1: Sniffing Packets

在此任务中，我们将学习如何在 Python 程序中使用 Scapy 实现数据包嗅探。如图 4 所示，编写 sniffer.py 程序：

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-a67c874d674a', filter='icmp', prn=print_pkt)
```

图 4

Task 1.1A

进入 attacker 容器，运行 sniffer.py（图 5）。此时攻击机开始监听网络上的数据包，如图 4 代码中的 filter 参数所示，我们只接收 ICMP 报文。

```
root@VM:/volumes# chmod a+x sniffer.py
root@VM:/volumes# sniffer.py
```

图 5

在 host 容器中，运行 PING 指令（图 6）。同时，攻击机中也产生了相应的输出（图 7）。

```
root@2596bef042a4:/# ping www.baidu.com
PING www.a.shifen.com (36.152.44.95) 56(84) bytes of data.
64 bytes from 36.152.44.95 (36.152.44.95): icmp_seq=1 ttl=54 time=22.3 ms
64 bytes from 36.152.44.95 (36.152.44.95): icmp_seq=2 ttl=54 time=7.25 ms
```

图 6

```
root@VM:/volumes# sniffer.py
###[ Ethernet ]###
  dst       = 02:42:6d:c3:f9:33
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 49505
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x1e43
     src       = 10.9.0.5
     dst       = 36.152.44.95
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xa0b4
        id        = 0x18
        seq       = 0x1
###[ Raw ]###
           load      = '\xd8\xfa\xf9b\x00\x00\x00\x00\xc0\x01\x06\x00\x00\x00\x00\x00\x10
\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

图 7

如果不使用 root 权限运行程序，我们会看到程序运行失败（图 8），因为数据包嗅探需要 root 权限。

```
root@VM:/volumes# su seed
seed@VM:/volumes$ sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, in <module>
    pkt = sniff(iface='br-a67c874d674a', 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
```

图 8

Task 1.1B

大多数情况下，当我们嗅探数据包时，只会对某些类型的数据包感兴趣。我们可以通过调整 sniff 函数中的 filter 参数来做到这一点。

（1）监听 ICMP 报文

过程与 Task 1.1A 相同。

（2）监听来自特定 IP 地址且目的端口为 23 的 TCP 报文

端口 23 为 Telnet 协议的默认端口，我们选择监听从主机（10.0.9.1）发出的 Telnet 报文。将 sniffer.py 中的 filter 参数修改如下：

```
filter='src host 10.9.0.5 and tcp dst port 23'
```

图 9

进入 attacker 容器，运行 sniffer.py。此时攻击机开始监听网络上的数据包。在 host 容器中运行 telnet 命令连接主机（图 10）。同时，攻击机中也产生了相应的输出（图 11）。

```
root@2596bef042a4:/# telnet 10.9.0.1
Trying 10.9.0.1...
Connected to 10.9.0.1.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login:
```

图 10

```
root@VM:/volumes# sniffer.py
###[ Ethernet ]###
  dst       = 02:42:6d:c3:f9:33
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 64729
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x29bb
     src       = 10.9.0.5
     dst       = 10.9.0.1
     \options   \
###[ TCP ]###
        sport     = 34462
        dport     = telnet
        seq       = 786385377
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 64240
        chksum    = 0x1446
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (4248713721, 0)),
 ('NOP', None), ('WScale', 7)]
```

图 11

（3）监听发往或来自某子网的数据包

我们选择发往或来自 128.230.0.0/16 网段的数据包，将 sniffer.py 中的 filter 参数修改如下：

```
filter='net 128.230.0.0/16'
```

图 12

进入 attacker 容器，运行 sniffer.py。此时攻击机开始监听网络上的数据包。在 host 容器中运行 PING 指令访问 128.230.0.1（图 13）。同时，攻击机中也产生了相应的输出（图 14）。

```
root@2596bef042a4:/# ping 128.230.0.1
PING 128.230.0.1 (128.230.0.1) 56(84) bytes of data.
64 bytes from 128.230.0.1: icmp_seq=1 ttl=44 time=260 ms
64 bytes from 128.230.0.1: icmp_seq=2 ttl=44 time=233 ms
```

图 13

```
root@VM:/volumes# sniffer.py
###[ Ethernet ]###
  dst       = 02:42:6d:c3:f9:33
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 5252
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9b30
     src       = 10.9.0.5
     dst       = 128.230.0.1
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xeae2
        id        = 0x1b
        seq       = 0x1
###[ Raw ]###
           load      = 'R\xfd\xf9b\x00\x00\x00\x00\xfd\xcd\x04\x00\x00\x00\x00\x00\x10
\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

图 14

Task 1.2: Spoofing ICMP Packets

在此任务中，我们将使用 Scapy 构造具有任意源 IP 地址的 ICMP Echo 请求报文，并将其发送给同一网络上的主机。

程序 sniffer.py 的内容如图 15 所示。进入 attacker 容器，运行 sniffer.py。此时攻击机开始监听网络上的 ICMP 数据包。

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-a67c874d674a', filter='icmp', prn=print_pkt)
```

图 15

程序 spoof_icmp.py 的内容如图 16 所示。其中，scapy 库中的 IP()、ICMP()和 send()函数，分别提供构造报文和发送功能；scapy 库中重载了除法运算符“/”，起到报文连接运算符的作用。

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

a=IP()
a.src='10.0.2.4'
a.dst='10.9.0.1'
b=ICMP()
p=a/b
send(p)
```

图 16

spoof_icmp.py 需要在 host 容器中运行，然而 host 容器并没有挂载共享文件夹，直接在容器内编辑文件较为麻烦，因此我们可以在主机上完成代码编写，使用 docker cp 命令将文件从宿主机拷贝到容器内（图 17）。

```
[08/15/22]seed@VM:~/.../volumes$ docker cp spoof_icmp.py 2596bef042a4:/spoof_icmp.py
```

图 17

进入 host 容器，运行 spoof_icmp.py（图 18）。

```
root@2596bef042a4:/# chmod a+x spoof_icmp.py
root@2596bef042a4:/# spoof_icmp.py
.
Sent 1 packets.
```

图 18

在 attacker 容器中收到如下数据包，可见该数据包的源 IP 地址为 10.0.2.4，正是我们在在 spoof_icmp.py 中计划的源 IP 地址。

```
root@VM:/volumes# sniffer.py
###[ Ethernet ]###
  dst       = 02:42:6d:c3:f9:33
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x64d3
     src       = 10.0.2.4
     dst       = 10.9.0.1
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
        id        = 0x0
        seq       = 0x0
```

图 19

Task 1.3: Traceroute

在此任务中，我们将使用 Scapy 来估计本地主机与某目标主机之间的距离，即路由器数量。原理非常简单：将一个数据包发往目的地，其 TTL 字段设置为 1。该报文经过路由器时，TTL 字段的数值递减，直到 TTL 等于 0 时路由器返回一个 ICMP 差错报告报文“Time-to-live exceeded”。这样，我们就获得了第一个路由器的 IP 地址。此后，我们将 TTL 字段增加到 2，发送第二个数据包，并获得第二个路由器的 IP 地址。重复这个过程，直到我们的数据包到达目的地。注意，使用上述方法只能得到一个估计值。理论上，发出的多个数据包可能经过不同的路由。

如图 20 所示，代码 try_ttl.py 从 1 开始（直到 30），测试合理的 TTL 值。我们选择 8.8.8.8 作为目标主机，构造并发送相应的 ICMP Echo 请求报文。

注意，我们需要选择本地能够达到的目标主机（图 21）。

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

# in most situations, 30 steps is enough
for i in range(1,30):
        a=IP()
        a.dst='8.8.8.8'
        a.ttl=i
        b=ICMP()
        p=a/b
        send(p)
```

图 20

```
root@2596bef042a4:/# 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=111 time=82.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=111 time=77.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=111 time=87.5 ms
```

图 21

若是选择了不存在的主机，在使用 WireShark 观察数据包时，我们就会发现发出的报文没有回复（no response），而 ICMP 差错报告报文（Time-to-live exceeded）仅返回了几次（图 22）。我们猜测可能是因为网络中设置了 ICMP 差错报告报文的数量限制或者一个流量黑洞，如果一直无法找到前往目的地址的路由，则将流量引向黑洞且不返回 ICMP 差错报告报文。

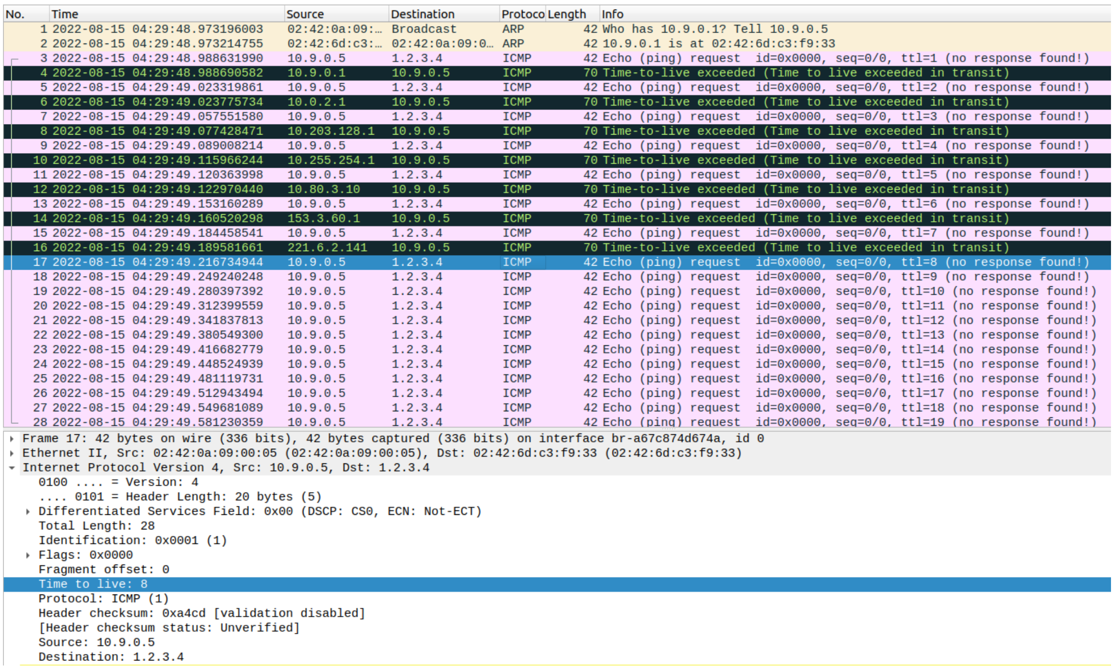

图 22

try_ttl.py 需要在 host 容器中运行，然而 host 容器并没有挂载共享文件夹，直接在容器内编辑文件较为麻烦，因此我们可以在主机上完成代码编写，使用 docker cp 命令将文件从宿主机拷贝到容器内（图 23）。

```
[08/15/22]seed@VM:~/.../volumes$ docker cp try_ttl.py 2596bef042a4:/try_ttl.py
```

图 23

如图 24 所示，在主机上开启 WireShark 监听创建容器时相应的网桥。

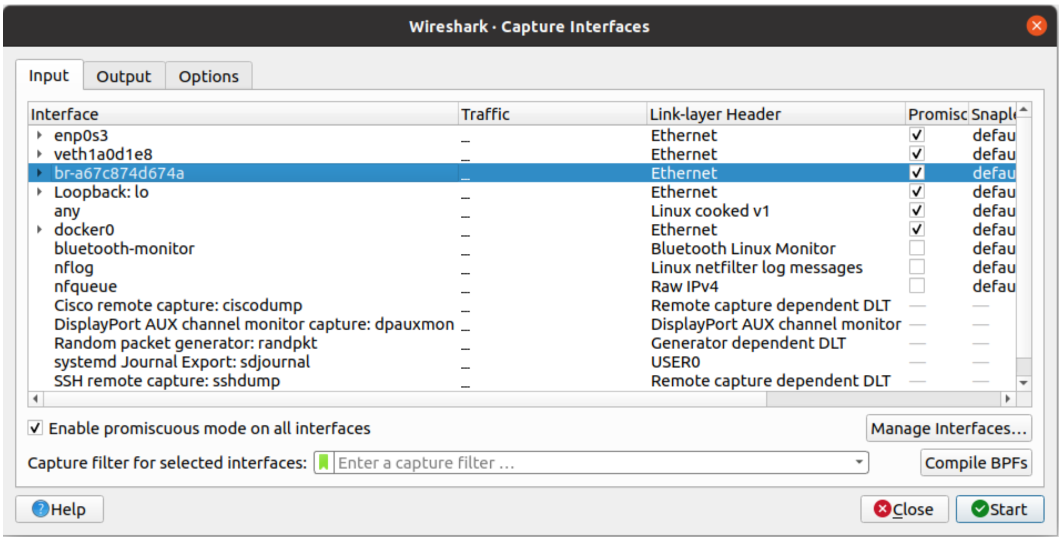

图 24

该网桥用于容器之间的通信，如果数据包从 host 容器发送至外网，可能不经过网桥，从而导致 WireShark 监听不到数据包。因此我们需要在 host 容器上运行 try_ttl.py（图 25）。

```
root@2596bef042a4:/# chmod a+x try_ttl.py
root@2596bef042a4:/# try_ttl.py
```

图 25

WireShark 的监听结果如下所示。可见当 TTL 为 18 时，发出的 ICMP Echo 请求报文收到了响应。

No.	Time	Source	Destination	Protoco	Length	Info
1	2022-08-15 04:54:09.436418203	02:42:0a:09:…	Broadcast	ARP	42	Who has 10.9.0.1? Tell 10.9.0.5
2	2022-08-15 04:54:09.436436786	02:42:6d:c3:…	02:42:0a:09:0…	ARP	42	10.9.0.1 is at 02:42:6d:c3:f9:33
3	2022-08-15 04:54:09.452779332	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=1 (no response found!)
4	2022-08-15 04:54:09.452829475	10.9.0.1	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
5	2022-08-15 04:54:09.485896765	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=2 (no response found!)
6	2022-08-15 04:54:09.486452322	10.0.2.1	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
7	2022-08-15 04:54:09.526116693	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=3 (no response found!)
8	2022-08-15 04:54:09.532274354	10.203.128.1	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
9	2022-08-15 04:54:09.564934962	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=4 (no response found!)
10	2022-08-15 04:54:09.574298048	10.255.254.1	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
11	2022-08-15 04:54:09.596774943	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=5 (no response found!)
12	2022-08-15 04:54:09.599565159	10.80.3.10	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
13	2022-08-15 04:54:09.628532617	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=6 (no response found!)
14	2022-08-15 04:54:09.635116870	153.3.60.1	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
15	2022-08-15 04:54:09.665206117	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=7 (no response found!)
16	2022-08-15 04:54:09.673771009	221.6.2.173	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
17	2022-08-15 04:54:09.697264390	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=8 (no response found!)
18	2022-08-15 04:54:09.701552843	122.96.66.97	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
19	2022-08-15 04:54:09.729407447	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=9 (no response found!)
20	2022-08-15 04:54:09.761034729	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=10 (no response found!)
21	2022-08-15 04:54:09.792858831	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=11 (no response found!)
22	2022-08-15 04:54:09.806157075	219.158.8.118	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
23	2022-08-15 04:54:09.836822173	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=12 (no response found!)
24	2022-08-15 04:54:09.846138997	219.158.96.2…	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
25	2022-08-15 04:54:09.877283496	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=13 (no response found!)
26	2022-08-15 04:54:09.895255307	219.158.16.2…	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
27	2022-08-15 04:54:09.912159723	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=14 (no response found!)
28	2022-08-15 04:54:09.923636280	72.14.213.114	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
29	2022-08-15 04:54:09.946813800	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=15 (no response found!)
30	2022-08-15 04:54:09.962889634	108.170.241.…	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
31	2022-08-15 04:54:09.984592245	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=16 (no response found!)
32	2022-08-15 04:54:09.997574140	108.170.225.…	10.9.0.5	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
33	2022-08-15 04:54:10.012670476	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=17 (no response found!)
34	2022-08-15 04:54:10.044505431	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=18 (reply in 35)
35	2022-08-15 04:54:10.048282877	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=111 (request in 34)
36	2022-08-15 04:54:10.080812357	10.9.0.5	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=19 (reply in 37)
37	2022-08-15 04:54:10.087628865	8.8.8.8	10.9.0.5	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=111 (request in 36)

```
Frame 34: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-a67c874d674a, id 0
Ethernet II, Src: 02:42:0a:09:00:05 (02:42:0a:09:00:05), Dst: 02:42:6d:c3:f9:33 (02:42:6d:c3:f9:33)
Internet Protocol Version 4, Src: 10.9.0.5, Dst: 8.8.8.8
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 28
    Identification: 0x0001 (1)
    Flags: 0x0000
    Fragment offset: 0
    Time to live: 18
    Protocol: ICMP (1)
```

图 26

Task 1.4: Sniffing and-then Spoofing

在此任务中，我们将结合 Task 1.1 和 Task 1.2 完成一个数据包嗅探和伪造程序。

首先，在 host 容器中依次访问以下三个 IP 地址，结果如图 28~30 所示。可以发现，只用 8.8.8.8 可以 PING 通。

```
ping 1.2.3.4       # a non-existing host on the Internet
ping 10.9.0.99     # a non-existing host on the LAN
ping 8.8.8.8       # an existing host on the Internet
```

图 27

```
root@2596bef042a4:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2041ms
```

图 28

```
root@2596bef042a4:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
^C
--- 10.9.0.99 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2034ms
```

图 29

```
root@2596bef042a4:/# 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=111 time=64.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=111 time=73.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=111 time=70.3 ms
```

图 30

程序 sniff_spoof.py 的内容如图 31 所示。

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

def spoof_pkt(pkt):
        if ICMP in pkt and pkt[ICMP].type == 8:
                print("Original Packet.......")
                print("Source IP : ", pkt[IP].src)
                print("Destination IP:", pkt[IP].dst)

                ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
                icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
                data = pkt[Raw].load
                newpkt = ip/icmp/data

                print("Spoofed Packet.......")
                print("Source IP : ", newpkt[IP].src)
                print("Destination IP : ", newpkt[IP].dst)
                send(newpkt, verbose=0)

pkt = sniff(iface='br-a67c874d674a', filter='icmp', prn=spoof_pkt)
```

图 31

进入 attacker 容器，运行 sniff_ spoof.py（图 32）。

```
root@VM:/volumes# chmod a+x sniff_spoof.py
root@VM:/volumes# sniff_spoof.py
```

图 32

进入 host 容器，依次访问图 27 所示的三个 IP 地址。访问 1.2.3.4 的结果如图 33~34 所示，访问 10.9.0.99 的结果如图 35 所示，访问 8.8.8.8 的结果如图 36~37 所示。

发往 1.2.3.4 的 ICMP Echo 请求报文成功得到响应。这是因为 ICMP 包的目的地址不在本地网段，数据包经过网关时被 attacker 容器中的 sniff_spoof.py 程序嗅探到，从而伪造了响应数据包。

```
root@2596bef042a4:/# 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=62.3 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=20.5 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=22.3 ms
```

图 33

```
root@VM:/volumes# sniff_spoof.py
Original Packet.......
Source IP :  10.9.0.5
Destination IP: 1.2.3.4
Spoofed Packet.......
Source IP :  1.2.3.4
Destination IP :  10.9.0.5
```

图 34

发往 10.9.0.99 的 ICMP Echo 请求报文无法得到响应。这是因为 ICMP 包的目的地址在本地子网网段，数据包不需要经过网关，因此不能被 sniff_spoof.py 程序嗅探并伪造响应。且该地址本身不存在，所以不会获得响应。

```
root@2596bef042a4:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
6 packets transmitted, 0 received, +3 errors, 100% packet loss, time 5099ms
pipe 4
```

图 35

8.8.8.8 是存在的公网 IP 地址，因此访问该 IP 地址得到了正常响应。且当数据包经过网关时会被 sniff_spoof.py 程序嗅探并伪造响应数据包，所以产生了 DUP 标志（重复数据包）。

```
root@2596bef042a4:/# 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=61.2 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=111 time=66.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=22.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=111 time=86.8 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=21.1 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=111 time=62.1 ms (DUP!)
```

图 36

```
root@VM:/volumes# sniff_spoof.py
Original Packet.......
Source IP :  10.9.0.5
Destination IP: 8.8.8.8
Spoofed Packet.......
Source IP :  8.8.8.8
Destination IP :  10.9.0.5
```

图 37

Summary

本次实验的内容是数据包嗅探和伪造，实验难度比前两次实验更低，实验的趣味性也更强。在 Task 1.3 中，我一开始选择了不存在的目标主机，使用 WireShark 观测数据包时出现了意外的结果，在与老师交流后明白了其中的原理，也找到了正确的实验方法；在 Task 1.4

中，需要深入理解数据包嗅探和伪造的原理，才能够解释为什么在运行自主编写的嗅探&伪造程序之后，访问三个 IP 地址出现了不同的结果。