一、前言

tcpreplay是一款强大的网络数据包重放工具，它可以将捕获到的网络流量（通常是pcap格式的文件）重新重放到网络中，实现对网络通信的重现。这在网络故障排查、安全测试、性能测试、开发调试等场景下具有广泛的应用。同时，tcpreplay不仅仅能重放TCP协议报文，**它支持重放所有协议报文**，同时支持IPv4和IPv6协议栈，不要被命名误导了，类比tcpdump的命名，tcpdump也能抓取所有协议报文而不仅仅是TCP。

TCPReplay包含几个核心组件和功能：

组件	功能	主要用途
tcpreplay	将捕获的网络流量（pcap文件）重放回网络。	控制重放速度、循环次数、输出接口等。
tcprewrite	修改pcap文件中的数据包内容。	修改IP地址、端口号、MAC地址等，用于模拟不同的网络环境或进行安全测试。
tcpprep	将pcap文件中的数据包按照客户端和服务器进行分类，为后续的重放做准备。	提高重放效率，特别是对于大型pcap文件。
tcpbridge	在两个网络之间建立桥接，将修改后的流量转发到不同的网络。	实现网络流量的隔离和转发。
tcpcapinfo	对pcap文件进行解码和调试。	分析pcap文件的内容，检查数据包的格式和内容。

本文将主要讲述前三个工具，即tcpreplay重放工具、tcprewrite重写、tcppgrep在各类应用场景中如何搭配使用。

二、完整重放 vs 筛选重放：应该选择哪种方式?

磨刀不误砍柴工，在进行重放之前，最好能预先确认是将整个pcap文件重放，还是有选择性的筛选特定报文再重放，不妨看一下各个场景的优缺点以及应该选择哪种方式重放，以及如何筛选特定报文。

1. 完整重放整个pcap文件

优点:

- 保证了原始的请求-响应序列，能更准确地模拟真实的网络交互。
- 可以全面重现包括握手、认证、数据交换等在内的所有网络行为。

缺点:

- 可能包含无关的流量，影响测试效率。

- 如果pcap文件过大，重放消耗资源大耗时较长。

2.筛选特定报文再重放

优点:

- 更加聚焦于客户端发起的请求，可以更精细地控制测试场景。
- 可以减少重放时间。

缺点:

- 如果服务器的响应对后续请求有影响，只重放请求可能会导致测试结果不准确。
- 可能会遗漏一些重要的上下文信息，如之前的握手过程或后续的交互。
- 增加操作步骤。

3.如何选择?

根据测试目的选择:

- 如果要测试客户端的发送功能，或者模拟客户端发起攻击，筛选重放客户端发送的报文更合适。
- 如果要测试整个交互过程，包括服务器的响应，完整重放更合适。

根据pcap文件大小选择:

- 如果 pcap 文件过大，筛选重放可以提高效率。

根据网络环境:

- 如果网络环境复杂，存在干扰，完整重放可以更好地模拟真实环境。

如果不确定使用哪种方式，推荐使用完整性重放。

4.如何筛选?

tcpdump或者tshark、wireshark都能做到报文筛选再写入的能力。

1.tcpdump

比如想过滤client.pcap包文件，源端IP为192.168.1.100的报文写入到client_requests.pcap，可以是:

```
tcpdump -r client.pcap -w client_requests.pcap src 192.168.1.100
```

2.tshark

过滤源端IP为192.168.1.100，并且请求的目的端协议端口为TCP 80，可以是：

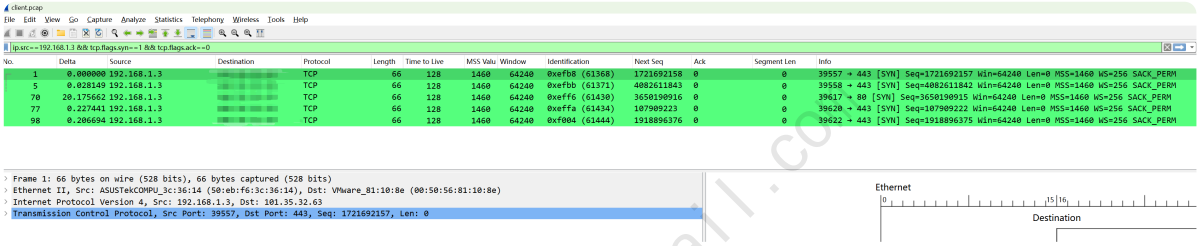
```
tshark -n -q -r client.pcap -Y 'ip.src==192.168.1.100&&tcp.dstport==80' -w
client_requests.pcap
```

3.Wireshark

Wireshark则直接输入过滤表达式后，在左上角选项栏点击 文件（File）--> 导出特定分组（Export Specified Packets）即可。

比如只想过滤客户端向外发起的SYN建联请求，过滤表达式可以是：

```
ip.src==192.168.1.3 && tcp.flags.syn==1 && tcp.flags.ack==0
```



三、安装

1.从软件源安装

发行版	安装命令
Archlinux	pacman -Sy tcpreplay
Centos/Redhat	yum install -y tcpreplay
Ubuntu/Debian	apt install tcpreplay
Gentoo	pacman --ask tcpreplay

2.二进制安装

到tcpreplay的 **releases页面** 下载最新版本，比如当前最新版为v4.5.1：

```
cd /opt
wget https://github.com/appneta/tcpreplay/releases/download/v4.5.1/tcpreplay-4.5.1.tar.gz
```

之后解压设置二进制文件的超链接：

```
tar xf tcpreplay-4.5.1.tar.gz
cd tcpreplay-4.5.1.tar.gz
ln -s /opt/tcpreplay-4.5.1/src/tcpreplay /bin/tcpreplay
ln -s /opt/tcpreplay-4.5.1/src/tcprewrite /bin/tcprewrite
```

之后便可以通过如下命令查看当前安装的版本：

```
tcpreplay -V
tcprewrite -V
```

```
┌──(root㉿kali)-[~]
└─# tcpreplay -V
tcpreplay version: 4.5.1 (build git:v4.5.1)
Copyright 2013-2024 by Fred Klassen <tcpreplay at appneta dot com> - AppNeta
Copyright 2000-2012 by Aaron Turner <aturner at synfin dot net>
The entire Tcpreplay Suite is licensed under the GPLv3
Cache file supported: 04
Not compiled with libdnet.
Compiled against libpcap: 1.10.4
64 bit packet counters: enabled
Verbose printing via tcpdump: enabled
Packet editing: disabled
Fragroute engine: disabled
Injection method: PF_PACKET send()
Not compiled with netmap
Not compiled with AF_XDP

┌──(root㉿kali)-[~]
└─# tcprewrite -V
tcprewrite version: 4.5.1 (build git:v4.5.1)
Copyright 2013-2024 by Fred Klassen <tcpreplay at appneta dot com> - AppNeta
Copyright 2000-2012 by Aaron Turner <aturner at synfin dot net>
The entire Tcpreplay Suite is licensed under the GPLv3
Cache file supported: 04
Not compiled with libdnet.
Compiled against libpcap: 1.10.4
64 bit packet counters: enabled
Verbose printing via tcpdump: enabled
Fragroute engine: disabled

┌──(root㉿kali)-[~]
└─#
```

四、实战演练

1.注意事项

1）重放的方向和报文感知

重放可以在客户端进行重放，也可以在服务端进行重放，甚至在一个隔离的实验环境下进行请求重放。

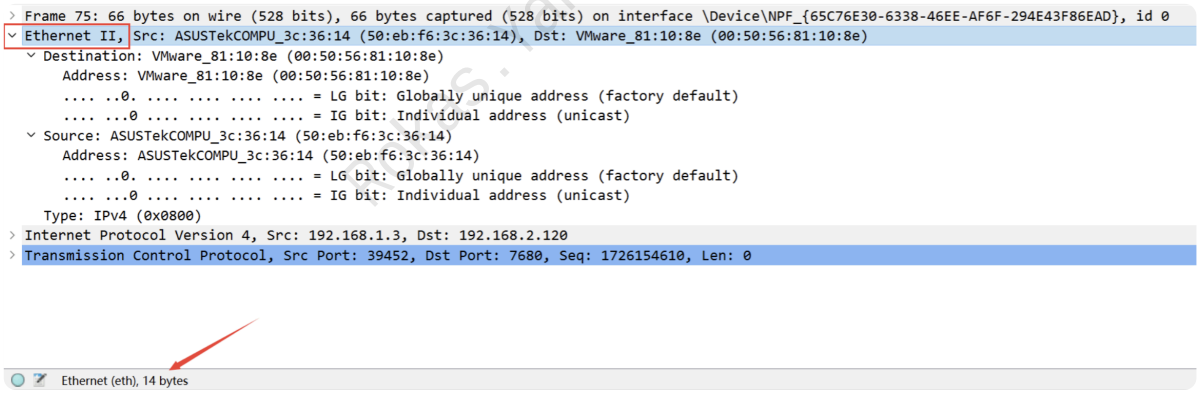
如果在客户端进行重放，重放一个完整包，以单个报文帧为维度，只有客户端往目的端发出去的方向，重放时目的端能收到这部分包，而在客户端重放目的端过来方向的包，只有客户端自己能收到，服务端感知不到；

同理，如果在服务端进行重放，以单个报文帧为维度，只有服务端往客户端传输的方向，重放时客户端能收到，在服务端重放客户端往服务端传输数据的方向，只有服务端能收到，客户端感知不到。

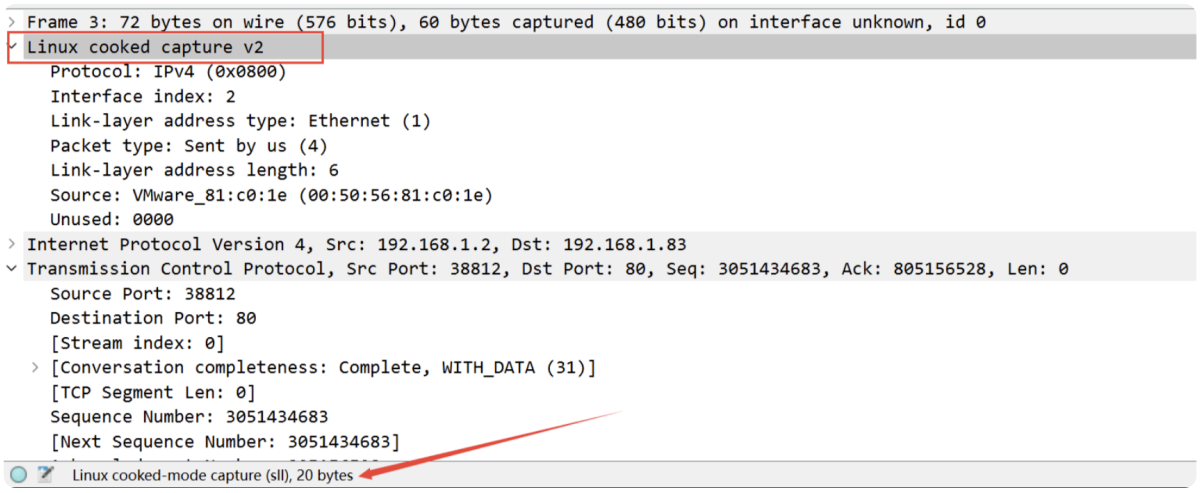
2）为什么不推荐使用-i any抓包再进行重放

在Linux上的抓包，如果抓包接口指定为**-i any**（比如tcpdump -i any），即抓取所有网卡，此时数据链路层可能不再显示为以太网，而是**Linux cooked capture v2(SLL)**，这是Linux上的伪协议，因为并不是一台机器上的所有接口都具有相同的链路层头部，参考 wireshark官网说明 。**这种情况tcpreplay是无法对这类没有以太网头部的报文进行重放的，会发现重放没有效果，发不出去包或者报错，因此重放pcap文件前，确保报文头里有以太网头部。**

正常的以太网头部：



SLL头部：



3）重放包不会模拟TCP协议栈的行为

tcpreplay只是重放报文，不会模拟TCP协议栈的行为，或者说，**它并不负责维护TCP连接的状态**。因此它不会自动处理TCP的三次握手过程和交互行为。比如客户端重放了SYN报文，服务端响应了SYN-ACK报文，但由于tcpreplay不会自动处理TCP的三次握手过程，客户端没有发送ACK报文来完成握手，导致连接被拒绝。因此看到这类TCP重放涉及到很多不能正常建立连接而会被RST的场景也不要诧异，是符合预期的。

看到这里可能还是比较抽象，不知道什么意思，不妨来看下面的几个例子。

2.完整重放

完整重放场景下，将pcap抓包文件的每一帧，重放到网络。

比如下面这个场景：

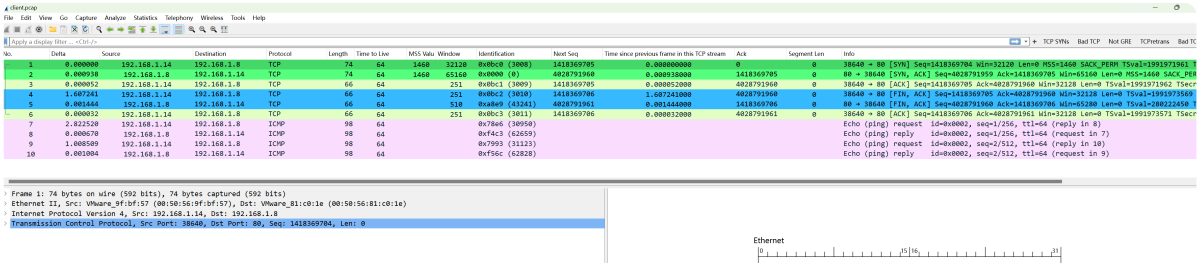
客户端	服务端	涉及协议
192.168.1.14	192.168.1.8	ICMP、80/TCP

首先我们在客户端抓取一段ICMP和TCP 80端口的报文：

```
tcpdump -i eth0 -nn host 192.168.1.8 and \( tcp port 80 or icmp \) -v -w client.pcap
```

```
┌──(root㉿kali)-[~]
└─# tcpdump -i eth0 -nn host 192.168.1.8 and \( tcp port 80 or icmp \) -v -w client.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
Got 10
```

```
Kali-2024.1 (1)  ×
└─# telnet 192.168.1.8 80
Trying 192.168.1.8...
Connected to 192.168.1.8.
Escape character is '^]'.
^]
telnet> quit
Connection closed.

┌──(root㉿kali)-[~]
└─# ping -c 2 192.168.1.8
PING 192.168.1.8 (192.168.1.8) 56(84) bytes of data.
64 bytes from 192.168.1.8: icmp_seq=1 ttl=64 time=0.704 ms
64 bytes from 192.168.1.8: icmp_seq=2 ttl=64 time=1.04 ms

--- 192.168.1.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1009ms
rtt min/avg/max/mdev = 0.704/0.871/1.038/0.167 ms
```



已经有了pcap原始抓包文件，我们使用tcpreplay将这个包文件完整重放一下，并且在客户端、服务端同时部署抓包，看看是什么表现。

首先我们需要更新下client.pcap报文中TCP下的checksum校验和，并输出为client_fix.pcap：

```
tcprewrite --infile=client.pcap --outfile=client_fix.pcap --fixcsum
```

```
┌──(root㉿kali)-[~]
└─# tcprewrite --infile=client.pcap --outfile=client_fix.pcap --fixcsum

┌──(root㉿kali)-[~]
└─#
```

1）为什么需要更新checksum?

校验和（checksum）是通过对数据进行计算得到的一个数值，发送方和接收方会对相同的数据计算出相同的校验和。如果接收方计算出的校验和与发送方提供的校验和不一致，就说明数据在传输过程中发生了错误，接收方会丢弃该数据包，比如不做csum更新的话，对端收到重放包，校验不对，是不会正常响应的，直接丢弃掉：

```
⚙ 02:27:50 # ~ tcpdump -i any -nn -s 0 host 192.168.1.14 -v
tcpdump: data link type LINUX_SLL2
dropped privs to pcap
tcpdump: listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
02:28:17.948855 ens192 In  IP (tos 0x0, ttl 64, id 3008, offset 0, flags [DF], proto TCP (6), length 60)
    192.168.1.14.38640 > 192.168.1.8.80: Flags [S], cksum 0x8395 (incorrect -> 0x3579), seq 1418369704, win 32120, options [mss 1460,sackOK,TS val 1991971961 ecr 0,nop,wscale 7]
02:28:17.950477 ens192 In  IP (tos 0x0, ttl 64, id 3009, offset 0, flags [DF], proto TCP (6), length 52)
    192.168.1.14.38640 > 192.168.1.8.80: Flags [.], cksum 0x838d (incorrect -> 0x8e99), ack 4028791960, win 251, options [nop,nop,TS val 1991971962 ecr 280220841], length 0
02:28:17.950653 ens192 In  IP (tos 0x0, ttl 64, id 3010, offset 0, flags [DF], proto TCP (6), length 52)
    192.168.1.14.38640 > 192.168.1.8.80: Flags [F.], cksum 0x838d (incorrect -> 0x8851), seq 0, ack 1, win 251, options [nop,nop,TS val 1991973569 ecr 280220841], length 0
02:28:17.950729 ens192 In  IP (tos 0x0, ttl 64, id 3011, offset 0, flags [DF], proto TCP (6), length 52)
    192.168.1.14.38640 > 192.168.1.8.80: Flags [.], cksum 0x838d (incorrect -> 0x8205), ack 2, win 251, options [nop,nop,TS val 1991973571 ecr 280222450], length 0
02:28:17.951174 ens192 In  IP (tos 0x0, ttl 64, id 30950, offset 0, flags [DF], proto ICMP (1), length 84)
    192.168.1.14 > 192.168.1.8: ICMP echo request, id 2, seq 1, length 64
02:28:17.951175 ens192 In  IP (tos 0x0, ttl 64, id 31123, offset 0, flags [DF], proto ICMP (1), length 84)
    192.168.1.14 > 192.168.1.8: ICMP echo request, id 2, seq 2, length 64
02:28:17.951236 ens192 Out IP (tos 0x0, ttl 64, id 49862, offset 0, flags [none], proto ICMP (1), length 84)
    192.168.1.8 > 192.168.1.14: ICMP echo reply, id 2, seq 1, length 64
```

通过netstat也可以统计收到的checksum错误的包量：

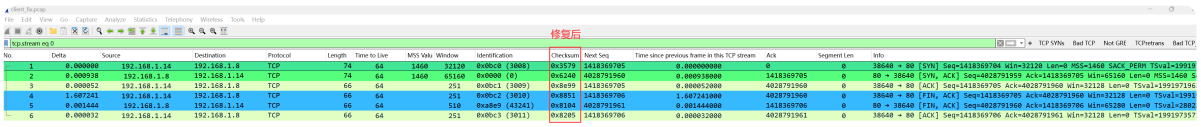
```
netstat -s|grep -i sum
```

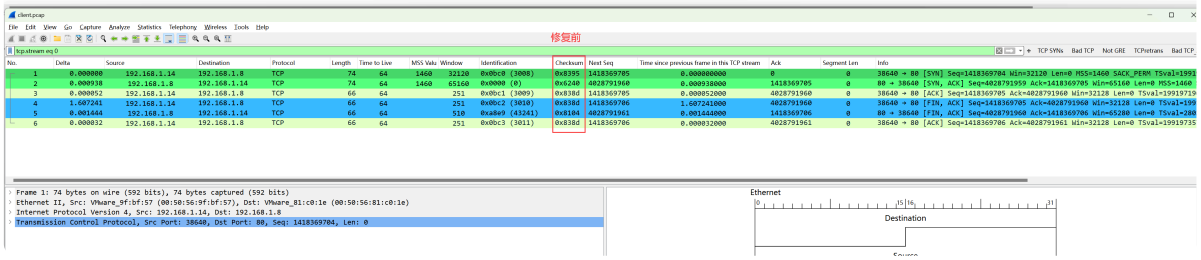
```
⚙ 02:36:49 # ~ netstat -s|grep -i sum
    InCsumErrors: 40
⚙ 02:36:54 # ~ netstat -s|grep -i sum
    InCsumErrors: 42
⚙ 02:37:15 # ~ netstat -s|grep -i sum
    InCsumErrors: 43
⚙ 02:37:17 # ~ netstat -s|grep -i sum
    InCsumErrors: 44
⚙ 02:37:18 # ~ netstat -s|grep -i sum
    InCsumErrors: 44
⚙ 02:37:19 # ~ netstat -s|grep -i sum
    InCsumErrors: 44
⚙ 02:37:20 # ~
```

```
Kali-2024.1 (1)
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0

┌──(root㉿kali)-[~]
└─# tcpreplay -vvv -i eth0 -p 1 client.pcap
reading from file ., link-type EN10MB (Ethernet), snapshot length 262144
14:16:31.000155 IP 192.168.1.14.38640 > 192.168.1.8.80: Flags [S], seq 1418369704, win 32120, options [mss 1460,sackOK,TS val 1991971961 ecr 0,nop,wscale 7], length 0
14:16:31.000156 IP 192.168.1.8.80 > 192.168.1.14.38640: Flags [S.], seq 4028791959, ack 1418369705, win 65160, options [mss 1460,sackOK,TS val 280220841 ecr 1991971961,nop,wscale 7], length 0
14:16:31.000156 IP 192.168.1.14.38640 > 192.168.1.8.80: Flags [.], ack 1, win 251, options [nop,nop,TS val 1991971962 ecr 280220841], length 0
14:16:32.000763 IP 192.168.1.14.38640 > 192.168.1.8.80: Flags [F.], seq 1, ack 1, win 251, options [nop,nop,TS val 1991973569 ecr 280220841], length 0
14:16:32.000764 IP 192.168.1.8.80 > 192.168.1.14.38640: Flags [F.], seq 1, ack 2, win 510, options [nop,nop,TS val 280222450 ecr 1991973569], length 0
14:16:32.000765 IP 192.168.1.14.38640 > 192.168.1.8.80: Flags [.], ack 2, win 251, options [nop,nop,TS val 1991973571 ecr 280222450], length 0
14:16:35.000587 IP 192.168.1.14 > 192.168.1.8: ICMP echo request, id 2, seq 1, length 64
14:16:35.000588 IP 192.168.1.8 > 192.168.1.14: ICMP echo reply, id 2, seq 1, length 64
14:16:36.000596 IP 192.168.1.14 > 192.168.1.8: ICMP echo request, id 2, seq 2, length 64
14:16:36.000597 IP 192.168.1.8 > 192.168.1.14: ICMP echo reply, id 2, seq 2, length 64
Actual: 10 packets (804 bytes) sent in 9.00 seconds
Rated: 89.3 Bps, 0.000 Mbps, 1.11 pps
Flows: 4 flows, 0.44 fps, 10 unique flow packets, 0 unique non-flow packets
Statistics for network device: eth0
        Successful packets:        10
```

从上图不难发现，没做checksum更新时，直接重放client.pcap，客户端发了四个TCP包给服务端，但服务端校验csum不正确，并没有响应客户端，此时InCsumErrors值也正好增加了4个。

对比修复前、修复后的两个pcap文件，可以看到checksum字段明显发生了变化，其它字段保持不变：





2）开始完整重放

更新校验和后，我们将client_fix.pcap完整重放：

```
tcpreplay -v -t -i eth0 client_fix.pcap #-v参数可以看到重放的每一帧的细节；-t参数尽可能快的重放数据包
```

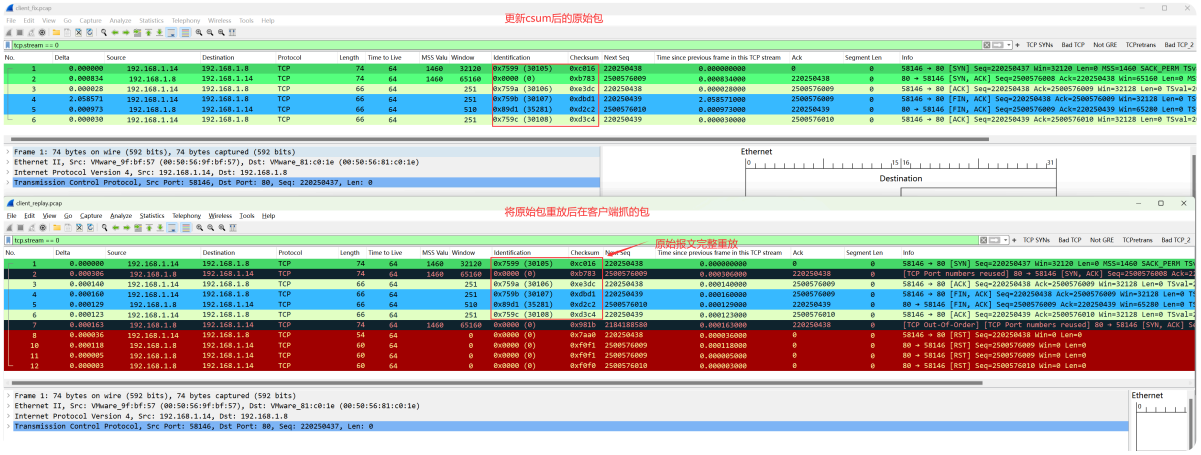
```
┌──(root㉿kali)-[~/tmp]
└─# tcpdump -i eth0 -nn host 192.168.1.8 and \( tcp port 80 or icmp \) -v -w client_replay.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
Got 17
```

```
┌──(root㉿kali)-[~/tmp]
└─# tcpreplay -v -t -i eth0 client_fix.pcap
reading from file -, link-type EN10MB (Ethernet), snapshot length 65535
22:46:54.000940 IP 192.168.1.14.58146 > 192.168.1.8.80: Flags [S], seq 220250437, win 32120, options [mss 1460,sackOK,TS val 2022595747 ecr 0,nop,wscale 7], length 0
22:46:54.000941 IP 192.168.1.8.80 > 192.168.1.14.58146: Flags [S.], seq 2500576008, ack 220250438, win 65160, options [mss 1460,sackOK,TS val 310844630 ecr 2022595747,
22:46:54.000941 IP 192.168.1.14.58146 > 192.168.1.8.80: Flags [.], ack 1, win 251, options [nop,nop,TS val 2022595748 ecr 310844630], length 0
22:46:57.000000 IP 192.168.1.14.58146 > 192.168.1.8.80: Flags [F.], seq 1, ack 1, win 251, options [nop,nop,TS val 2022597806 ecr 310844630], length 0
22:46:57.000001 IP 192.168.1.8.80 > 192.168.1.14.58146: Flags [F.], seq 1, ack 2, win 510, options [nop,nop,TS val 310846689 ecr 2022597806], length 0
22:46:57.000001 IP 192.168.1.14.58146 > 192.168.1.8.80: Flags [.], ack 2, win 251, options [nop,nop,TS val 2022597807 ecr 310846689], length 0
22:46:58.000591 IP 192.168.1.14 > 192.168.1.8: ICMP echo request, id 5, seq 1, length 64
22:46:58.000591 IP 192.168.1.8 > 192.168.1.14: ICMP echo reply, id 5, seq 1, length 64
22:46:59.000592 IP 192.168.1.14 > 192.168.1.8: ICMP echo request, id 5, seq 2, length 64
22:46:59.000593 IP 192.168.1.8 > 192.168.1.14: ICMP echo reply, id 5, seq 2, length 64
Actual: 10 packets (804 bytes) sent in 0.009370 seconds
Rated: 85805.7 Bps, 0.686 Mbps, 1067.23 pps
Flows: 4 flows, 426.89 fps, 10 unique flow packets, 0 unique non-flow packets
Statistics for network device: eth0
        Successful packets:        10
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0

┌──(root㉿kali)-[~/tmp]
└─#
```

最下面会展示重放的包量、耗时、发包速率pps、带宽、流数量、成功包量、失败包量、截断包量、重传报量等统计信息。

重放报文的同时，我们在客户端、服务端同时抓包。

此时我们先对比下更新checksum后的原始包client_fix.pcap，和重放原始包后在客户端抓到的包client_replay.pcap，有什么区别：



- 1-6 帧，客户端已经重放完了原始报文，这是两个包的相同部分。
- 第7帧开始，客户端收到了来自服务端发出来的SYN,ACK第二次握手的报文，是回复给第2帧的，这也是符合预期的，服务端收到了客户端的SYN请求，回复SYN,ACK很合理，只不过回复速度比客户端重放速度慢，因为客户端并不需要等待连接建立，它仅仅是把pcap里面的报文以最快的速度重放一遍，发包速度肯定会非常快，也不涉及什么tcp 0.2s的定时器和超时重传等逻辑，前面注意事项也说过，**它不会模拟TCP协议栈的行为。**
- 到了第8帧，客户端主动发送了RST来响应服务端回复的第7帧，在客户端看来，此时我已经没有任何SYN_SENT状态的TCP连接，服务端给我发送一个SYN,ACK是什么意思，直接RST拒绝掉。
- 第10、11帧，通过RST的SEQ序列号可以发现，客户端收到了服务端的RST，是RST给第5帧FIN,ACK的，因为服务端也觉得莫名其妙，这条TCP连接并没有正常建立，客户端给我FIN,ACK是什么意思，我要结束挥手哪条连接？不认识，直接RST拒绝。
- 同理，第12帧，也是服务端响应RST给客户端的第6帧ACK的，在服务端来看，莫名其妙给我发送一个ACK过来，RST拒绝。

为什么会RST?

回到前面说的注意事项，因为tcpreplay重放的数据包并不会模拟TCP协议栈的行为和维护TCP连接的状态，只是单纯将pcap里的报文发完，此时两端对这些报文都会产生疑惑，明明TCP连接没有Established，为什么给我发一些奇奇怪怪标志位的包，无法正常处理这些包，那么就用RST来回复你。

顺便穿插下，其实用nping指定flag标志位也能模拟测试这类RST的场景，比如客户端向服务端发送一些莫名其妙的标志位的包：

```
nping --tcp -p 80 --flag ACK 192.168.1.8       # 发ACK包给服务端
nping --tcp -p 80 --flags SYN,ACK 192.168.1.8  # 发SYN,ACK包给服务端
nping --tcp -p 80 --flags FIN,ACK 192.168.1.8  # 发FIN,ACK包给服务端
```

```
┌──(root㉿kali)-[~]
└─# nping --tcp -p 80 --flag ACK 192.168.1.8 -c 1

Starting Nping 0.7.94SVN ( https://nmap.org/nping ) at 2024-09-27 23:26 EDT
SENT (0.0163s) TCP 192.168.1.14:46323 > 192.168.1.8:80 A ttl=64 id=6170 iplen=40  seq=3021458908 win=1480
RCVD (0.0172s) TCP 192.168.1.8:80 > 192.168.1.14:46323 R ttl=64 id=0 iplen=40  seq=2417132998 win=0

Max rtt: 0.837ms | Min rtt: 0.837ms | Avg rtt: 0.837ms
Raw packets sent: 1 (40B) | Rcvd: 1 (46B) | Lost: 0 (0.00%)
Nping done: 1 IP address pinged in 1.04 seconds

┌──(root㉿kali)-[~]
└─# nping --tcp -p 80 --flags SYN,ACK 192.168.1.8 -c 1

Starting Nping 0.7.94SVN ( https://nmap.org/nping ) at 2024-09-27 23:26 EDT
SENT (0.0207s) TCP 192.168.1.14:3626 > 192.168.1.8:80 SA ttl=64 id=58748 iplen=40  seq=4113288018 win=1480
RCVD (0.0220s) TCP 192.168.1.8:80 > 192.168.1.14:3626 R ttl=64 id=0 iplen=40  seq=312378617 win=0

Max rtt: 1.195ms | Min rtt: 1.195ms | Avg rtt: 1.195ms
Raw packets sent: 1 (40B) | Rcvd: 1 (46B) | Lost: 0 (0.00%)
Nping done: 1 IP address pinged in 1.06 seconds

┌──(root㉿kali)-[~]
└─# nping --tcp -p 80 --flags FIN,ACK 192.168.1.8 -c 1

Starting Nping 0.7.94SVN ( https://nmap.org/nping ) at 2024-09-27 23:26 EDT
SENT (0.0280s) TCP 192.168.1.14:18065 > 192.168.1.8:80 FA ttl=64 id=4937 iplen=40  seq=2631346959 win=1480
RCVD (0.0292s) TCP 192.168.1.8:80 > 192.168.1.14:18065 R ttl=64 id=0 iplen=40  seq=3096479861 win=0

Max rtt: 1.096ms | Min rtt: 1.096ms | Avg rtt: 1.096ms
Raw packets sent: 1 (40B) | Rcvd: 1 (46B) | Lost: 0 (0.00%)
Nping done: 1 IP address pinged in 1.06 seconds
```

这类包无不例外都被RST掉了。

接着，我们对比客户端抓的包client_replay.pcap和服务端抓的包server.pcap：

对比ip.id和checksum，可以发现几个特点：

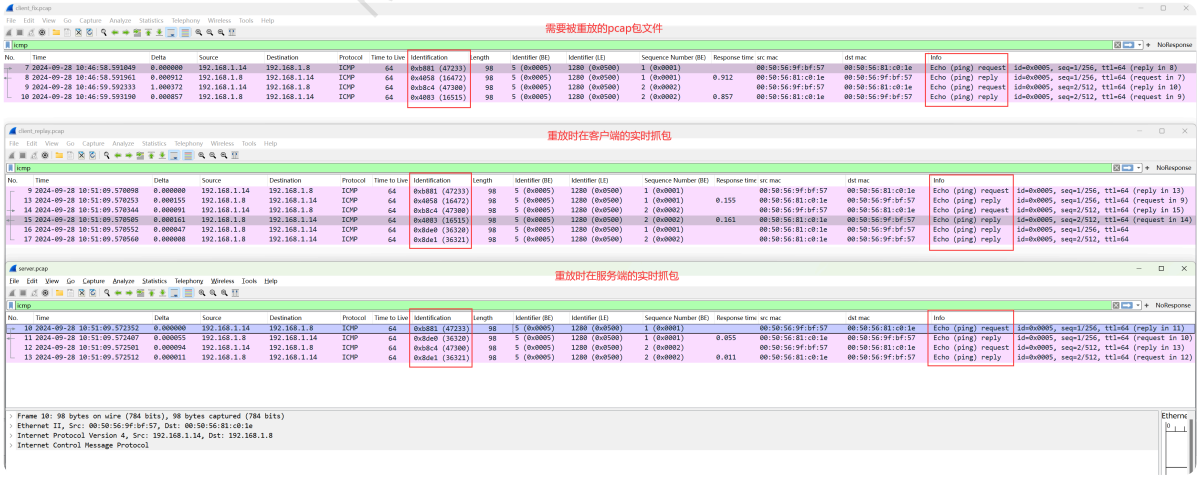
客户端重放的包，如果方向是服务端往客户端发送的，客户端重放只会重放给自己，没办法给服务端也发送一份，能重放给服务端的，只有客户端往服务端发送的这个方向，比如上图server.pcap中的SYN（第1帧）、ACK（第3帧）、FIN,ACK（第4帧），ACK（第7帧），其余的几帧，比如SYN,ACK（第2帧）是服务端实时响应的，而非重放产生的，重放的SYN,ACK checksum是0xb783，服务端实时响应的SYN,ACK checksum是0x8395，其余的同理。

因此得出一个结论：

- 客户端重放完整的数据包，不需要服务端去响应这些重放包，而是完整把这些包发一遍；
- 客户端重放数据包时，对于客户端往服务端发送方向的数据，服务端是能收到的，服务端往客户端方向发送的数据，客户端只能重放给自己，没能力要求服务端传送这样的一个数据包过来；
- 在服务端看来，因为重放过来的包不具备真正意义上的建联属性，服务端即使响应第二次握手SYN,ACK，连接也建立不起来，因此对于后续客户端发送的ACK、FIN,ACK等都会发送RST响应这类报文。

TCP部分已经看完，我们再来看ICMP部分。

同时打开需要被重放的包client_fix.pcap、重放时在客户端的实时抓包client_replay.pcap、重放时在服务端的实时抓包server.pcap：



对比ip.id不难发现，和上面的结论有一些共同之处：

- 客户端重放完整的数据包时，过去的方向icmp request和过来方向的icmp reply都会重放；
- 对于icmp request，确确实实已经重放给服务端了，服务端已经收到了，并且响应了；
- 对于重放的icmp reply，客户端会重放给自己，再加上服务端实时响应的icmp reply，因此客户端的实时抓包，出现了重复的icmp reply响应。

3.重放单方向的包

只重放客户端发出去的包，还是以client.pcap为例，我们将客户端发出去的方向的包过滤出来：

```
tcpdump -r client.pcap src 192.168.1.14 -w client_requests.pcap
```

```
┌──(root㉿kali)-[~]
└─# tcpdump -r client.pcap src 192.168.1.14 -w client_requests.pcap
reading from file client.pcap, link-type EN10MB (Ethernet), snapshot length 262144

┌──(root㉿kali)-[~]
└─# tcpdump -n -r client_requests.pcap
reading from file client_requests.pcap, link-type EN10MB (Ethernet), snapshot length 262144
15:28:15.338423 IP 192.168.1.14.38122 > 192.168.1.8.80: Flags [S], seq 2337782632, win 32120, options [mss 1460,sackOK,TS val 1996276144 ecr 0,nop,wscale 7], length 0
15:28:15.339304 IP 192.168.1.14.38122 > 192.168.1.8.80: Flags [.], ack 2167779438, win 251, options [nop,nop,TS val 1996276145 ecr 284525027], length 0
15:28:17.471992 IP 192.168.1.14.38122 > 192.168.1.8.80: Flags [F.], seq 0, ack 1, win 251, options [nop,nop,TS val 1996278278 ecr 284525027], length 0
15:28:17.473477 IP 192.168.1.14.38122 > 192.168.1.8.80: Flags [.], ack 2, win 251, options [nop,nop,TS val 1996278279 ecr 284527161], length 0
15:28:18.645545 IP 192.168.1.14 > 192.168.1.8: ICMP echo request, id 4, seq 1, length 64
15:28:19.646955 IP 192.168.1.14 > 192.168.1.8: ICMP echo request, id 4, seq 2, length 64
```

进行csum修复：

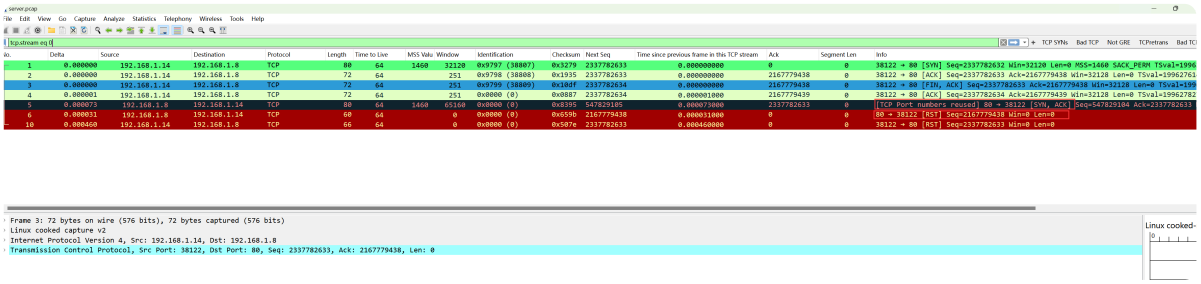
```
tcprewrite --infile=client_requests.pcap --outfile=client_fix.pcap --fixcsum
```

重放更新csum后的包给服务端：

```
tcpreplay -v -i eth0 -p 1000 client_fix.pcap # -p指定每秒发包速率
```

此时在服务端抓包可以看到，客户端自顾自一次性重放完了SYN（第一次握手）、ACK（第三次握手）、FIN,ACK（申请挥手）、ACK（确认挥手）等单方向的包，此时服务端处理也很有意思：

- 服务端回复了SYN,ACK来回应客户端的第一次握手SYN，因为中间链路也存在耗时的原因，在客户端看来，这个SYN,ACK明显没有客户端的发包速度快，因此在倒数的几个帧里；
- 紧接着，服务端回复了RST去响应客户端发出来的FIN,ACK，因为在服务端来看，这条TCP连接并没有正常建立成功，客户端发个FIN,ACK，服务端不理解什么意思，直接RST拒绝；
- 同时客户端也发送RST来回复服务端发出来的SYN,ACK，因为在客户端看来，也不懂对端发的SYN,ACK是什么含义，自己并没有SYN_SENT状态的连接。

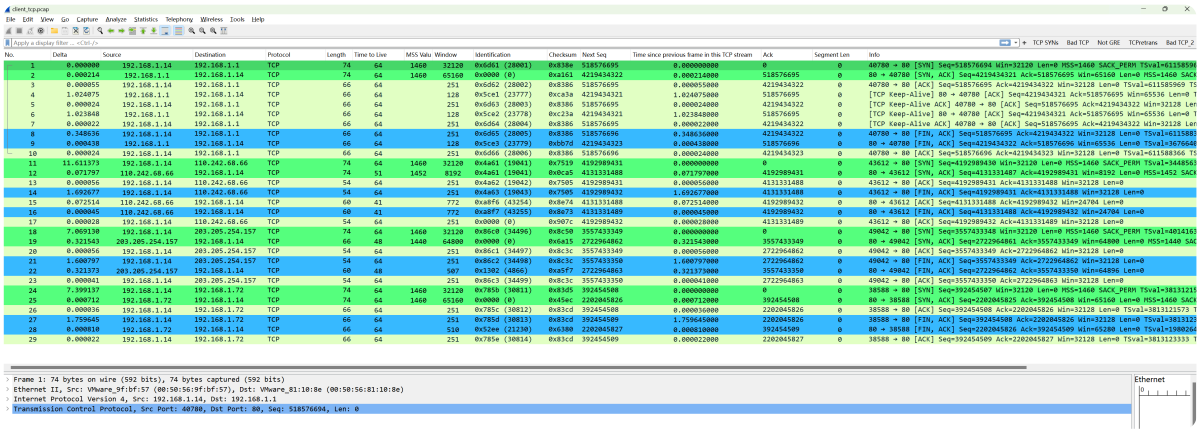


4.重放特定报文

重放特定报文，对于环境较为复杂的业务场景和自研应用层协议场景非常有用，排除了一些干扰因素，让重放实验更纯净简洁，以下通过第一次握手SYN的重放和dns query请求重放进行举例。

1）重放第一次握手的SYN报文

比如client_tcp.pcap报文里有一些和内外网服务端80端口建联的完整请求：



我们只想重放第一次SYN握手的部分，首先需要把SYN标志位为1并且ACK为0的报文过滤出来保存为client_syn.pcap：

tcpdump可以是：

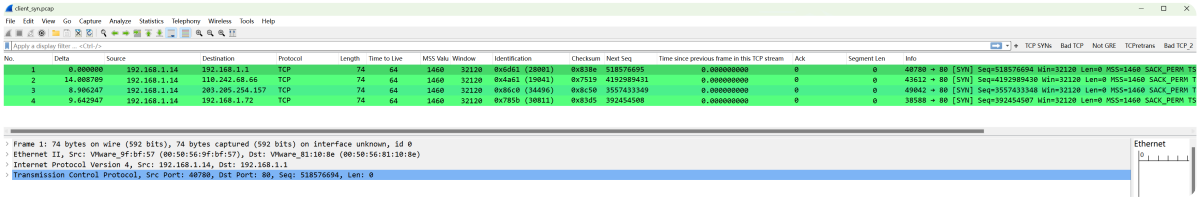
```
tcpdump -r client_tcp.pcap 'tcp[tcpflags] & (tcp-syn) != 0 and tcp[tcpflags] & (tcp-ack) == 0' -w client_syn.pcap
```

tshark可以是：

```
tshark -n -q -r client_tcp.pcap -Y 'tcp.flags.syn==1&&tcp.flags.ack==0' -w client_syn.pcap
```

```
┌──(root㉿kali)-[~]
└─# tcpdump -r client_tcp.pcap 'tcp[tcpflags] & (tcp-syn) != 0 and tcp[tcpflags] & (tcp-ack) == 0' -w client_syn.pcap
reading from file client_tcp.pcap, link-type EN10MB (Ethernet), snapshot length 262144

┌──(root㉿kali)-[~]
└─# tcpdump -n -r client_syn.pcap
reading from file client_syn.pcap, link-type EN10MB (Ethernet), snapshot length 262144
05:21:31.279976 IP 192.168.1.14.40780 > 192.168.1.1.80: Flags [S], seq 518576694, win 32120, options [mss 1460,sackOK,TS val 611585969 ecr 0,nop,wscale 7], length 0
05:21:45.288685 IP 192.168.1.14.43612 > 110.242.68.66.80: Flags [S], seq 4192989430, win 32120, options [mss 1460,sackOK,TS val 3448563492 ecr 0,nop,wscale 7], length 0
05:21:54.194932 IP 192.168.1.14.49042 > 203.205.254.157.80: Flags [S], seq 3557433348, win 32120, options [mss 1460,sackOK,TS val 401416380 ecr 0,nop,wscale 7], length 0
05:22:03.837879 IP 192.168.1.14.38588 > 192.168.1.72.80: Flags [S], seq 392454507, win 32120, options [mss 1460,sackOK,TS val 3813121571 ecr 0,nop,wscale 7], length 0

┌──(root㉿kali)-[~]
└─# tshark -n -q -r client_tcp.pcap -Y 'tcp.flags.syn==1&&tcp.flags.ack==0' -w client_syn.pcap
Running as user "root" and group "root". This could be dangerous.

┌──(root㉿kali)-[~]
└─# tshark -n -r client_syn.pcap
Running as user "root" and group "root". This could be dangerous.
    1   0.000000 192.168.1.14 → 192.168.1.1  TCP 74 40780 → 80 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=611585969 TSecr=0 WS=128
    2  14.008709 192.168.1.14 → 110.242.68.66 TCP 74 43612 → 80 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=3448563492 TSecr=0 WS=128
    3  22.914956 192.168.1.14 → 203.205.254.157 TCP 74 49042 → 80 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=401416380 TSecr=0 WS=128
    4  32.557903 192.168.1.14 → 192.168.1.72 TCP 74 38588 → 80 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=3813121571 TSecr=0 WS=128
```

过滤出来的client_tcp.pcap内容如下：



接下来，我们使用tcprewrite来更新一下checksum：

```
tcprewrite --infile=client_syn.pcap --outfile=client_syn_fix.pcap --fixcsum
```

一切准备就绪，准备开始重放，在此之前我们在客户端部署下抓包：

```
tcpdump -i eth0 -nn -s 0 tcp port 80 -v -w client_syn_replay.pcap
```

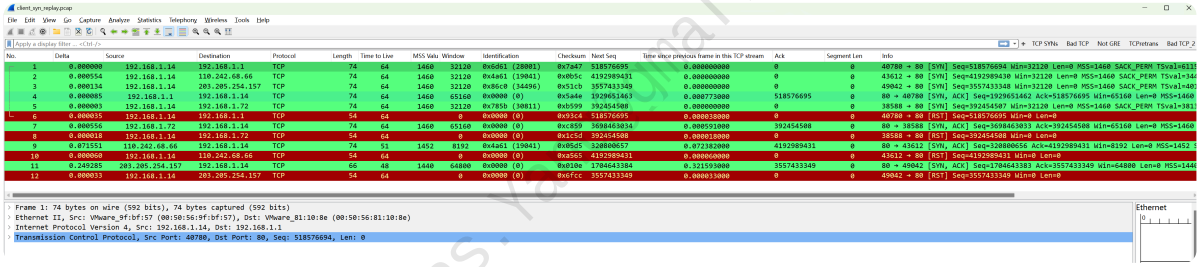
紧接着，使用tcpreplay开始重放：

```
tcpreplay -v -t -i eth0 client_syn_fix.pcap
```

```
┌──(root㉿kali)-[~]
└─# tcpdump -i eth0 -nn -s 0 tcp port 80 -v -w client_syn_replay.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
Got 12
```

```
Kali-2024.1 (1)
┌──(root㉿kali)-[~]
└─# tcpreplay -v -t -i eth0 client_syn_fix.pcap
reading from file -, link-type EN10MB (Ethernet), snapshot length 65535
05:21:31.000279 IP 192.168.1.14.40780 > 192.168.1.1.80: Flags [S], seq 518576694, win 32120, options [mss 1460,sackOK,TS val 611585969 ecr 0,nop,wscale 7], length 0
05:21:45.000288 IP 192.168.1.14.43612 > 110.242.68.66.80: Flags [S], seq 4192989430, win 32120, options [mss 1460,sackOK,TS val 3448563492 ecr 0,nop,wscale 7], length 0
05:21:54.000194 IP 192.168.1.14.49042 > 203.205.254.157.80: Flags [S], seq 3557433348, win 32120, options [mss 1460,sackOK,TS val 401416380 ecr 0,nop,wscale 7], length 0
05:22:03.000837 IP 192.168.1.14.38588 > 192.168.1.72.80: Flags [S], seq 392454507, win 32120, options [mss 1460,sackOK,TS val 3813121571 ecr 0,nop,wscale 7], length 0
Actual: 4 packets (296 bytes) sent in 0.009168 seconds
Rated: 32286.2 Bps, 0.258 Mbps, 436.30 pps
Flows: 4 flows, 436.30 fps, 4 unique flow packets, 0 unique non-flow packets
Statistics for network device: eth0
        Successful packets:        4
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0

┌──(root㉿kali)-[~]
└─#
```

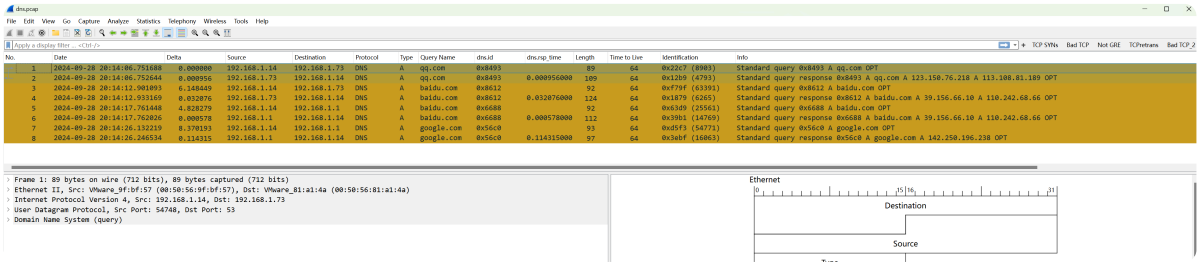
我们观察下客户端的实时抓包：



四次SYN重放，都成功拿到了对端的SYN,ACK响应，紧接着客户端RST了对端的SYN,ACK，因为客户端重放报文，不存在TCP连接的建立，不会模拟TCP协议栈的行为，因此，tcpreplay不会自动处理TCP的三次握手过程，看过前面的注意事项就知道，发RST，非常合理。

2）重放dns query查询请求

比如dns.pcap这个抓包文件，里面有一系列DNS查询请求，我们想把dns查询请求单独过滤出来并进行重放。



首先把dns query请求筛选出来另存为dns_query.pcap：

```
tcpdump -n -r dns.pcap 'udp port 53 and udp[10] & 0x80 == 0' -w dns_query.pcap
```

或者：

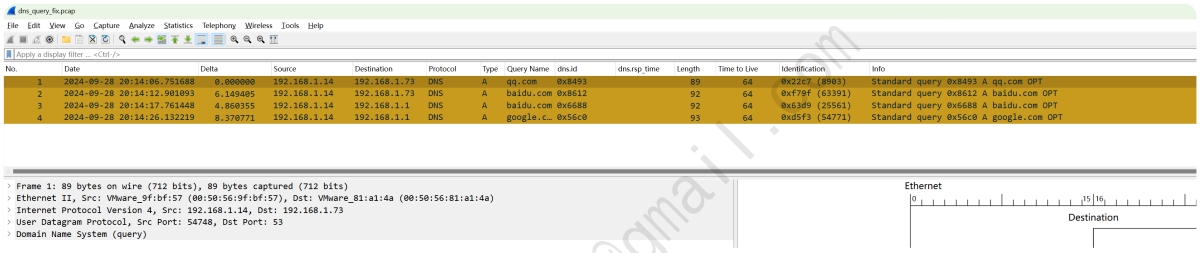
```
tshark -n -q -r dns.pcap -Y 'dns.flags.response == 0' -w dns_query.pcap
```

```
┌──(root㉿kali)-[~]
└─# tshark -n -q -r dns.pcap -Y 'dns.flags.response == 0' -w dns_query.pcap
Running as user "root" and group "root". This could be dangerous.

┌──(root㉿kali)-[~]
└─# tshark -n -r dns_query.pcap
Running as user "root" and group "root". This could be dangerous.
    1   0.000000 192.168.1.14 → 192.168.1.73 DNS 89 Standard query 0x8493 A qq.com OPT
    2   6.149405 192.168.1.14 → 192.168.1.73 DNS 92 Standard query 0x8612 A baidu.com OPT
    3  11.009760 192.168.1.14 → 192.168.1.1  DNS 92 Standard query 0x6688 A baidu.com OPT
    4  19.380531 192.168.1.14 → 192.168.1.1  DNS 93 Standard query 0x56c0 A google.com OPT
```

更新checksum，输出为dns_query_fix.pcap：

```
tcprewrite --infile=dns_query.pcap --outfile=dns_query_fix.pcap --fixcsum
```

之后我们便拿到了可以用于重放的pcap包：



一共四条A记录的查询，对应两个内网DNS服务器。

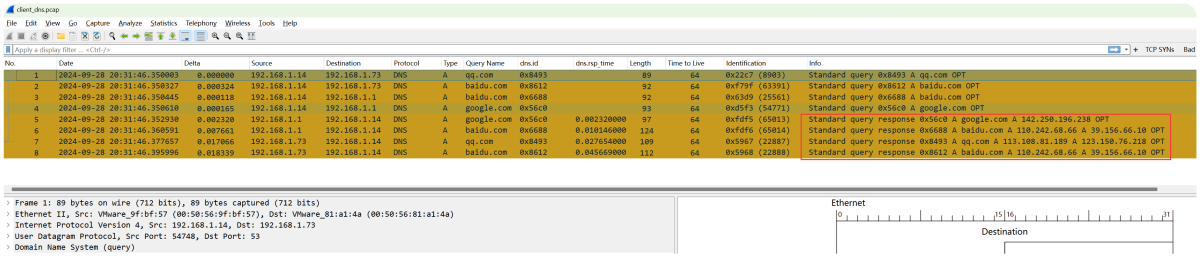
到此，可以开始重放dns_query_fix.pcap，同时我们在客户端抓包，看看能否拿到服务端的响应：

```
tcpdump -i eth0 -nn -s 0 udp port 53 -v -w client_dns.pcap
tcpreplay -v -t -i eth0 dns_query_fix.pcap
```

```
┌──(root㉿kali)-[~]
└─# tcpdump -i eth0 -nn -s 0 udp port 53 -v -w client_dns.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
Got 8
```

```
Kali-2024.1 (1)
┌──(root㉿kali)-[~]
└─# tcpreplay -v -t -i eth0 dns_query_fix.pcap
reading from file -, link-type EN10MB (Ethernet), snapshot length 65535
08:14:06.000751 IP 192.168.1.14.54748 > 192.168.1.73.53: 33939+ [1au] A? qq.com. (47)
08:14:12.000901 IP 192.168.1.14.60773 > 192.168.1.73.53: 34322+ [1au] A? baidu.com. (50)
08:14:17.000761 IP 192.168.1.14.39197 > 192.168.1.1.53: 26248+ [1au] A? baidu.com. (50)
08:14:26.000132 IP 192.168.1.14.48231 > 192.168.1.1.53: 22208+ [1au] A? google.com. (51)
Actual: 4 packets (366 bytes) sent in 0.005407 seconds
Rated: 67690.0 Bps, 0.541 Mbps, 739.78 pps
Flows: 4 flows, 739.78 fps, 4 unique flow packets, 0 unique non-flow packets
Statistics for network device: eth0
        Successful packets:        4
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0

┌──(root㉿kali)-[~]
└─#
```

用wireshark打开客户端抓包文件client_dns.pcap：



客户看到四条A记录的重放，服务端都一一响应了。dnsmasq服务器也能查询到此次日志：

```
20:35:34 ~ tail -f /var/log/dnsmasq.log
Sep 28 20:31:46 dnsmasq[14914]: forwarded qq.com to 114.114.114.114
Sep 28 20:31:46 dnsmasq[14914]: nameserver 114.114.114.114 refused to do a recursive query
Sep 28 20:31:46 dnsmasq[14914]: reply qq.com is 113.108.81.189
Sep 28 20:31:46 dnsmasq[14914]: reply qq.com is 123.150.76.218
Sep 28 20:31:46 dnsmasq[14914]: query[A] baidu.com from 192.168.1.14
Sep 28 20:31:46 dnsmasq[14914]: forwarded baidu.com to 192.168.1.72
Sep 28 20:31:46 dnsmasq[14914]: forwarded baidu.com to 114.114.114.114
Sep 28 20:31:46 dnsmasq[14914]: nameserver 114.114.114.114 refused to do a recursive query
Sep 28 20:31:46 dnsmasq[14914]: reply baidu.com is 110.242.68.66
Sep 28 20:31:46 dnsmasq[14914]: reply baidu.com is 39.156.66.10
```

5.重写源IP/目的IP/源MAC/目的MAC再进行重放

已知client.pcap文件涉及到的信息如下：

客户端	服务端	涉及协议
192.168.1.14	192.168.1.8	ICMP、80/TCP

假设因为生产环境的原因，1.14生产机器不能再用于重放，此时我们想要在别的客户端和服务端进行重放实验，譬如192.168.1.12进行重放，重放给192.168.1.72，信息如下：

客户端	客户端MAC	服务端	服务端MAC	涉及协议
192.168.1.12	00:50:56:81:8e:44	192.168.1.72	00:50:56:81:be:58	ICMP、80/TCP

因为tcprewrite重写的方向是每一帧维度的源端和目的端，不区分IP或mac，只区分左右方向，比如一条交互报文如下：

```
A --icmp request--> B
B --icmp reply  --> A
```

这两个正常交互报文，方向不一样，在tcprewrite眼里，第一条报文的源是A目的是B，第二条报文的源是B目的是A，如果直接使用tcprewrite修改源地址和目的地址，比如源重写为a，目的重写为b，会造成如下效果：

```
a --icmp request--> b
a --icmp reply  --> b
```

第一条报文没问题，第二条报文就有问题了，方向发生了变化，正确方向应该是：

```
b --icmp reply --> a
```

因此，要正确的重写整个报文里的源目的，并且保证方向正确，要做的步骤稍微有点繁琐：

1）将报文拆成两个方向的包

拆包使用tcpdump、tshark、wireshark都可以，将我们要的报文方向过滤出来写入到新pcap文件即可；比如客户端出去的方向，tcpdump可以是：

```
tcpdump -r client.pcap src 192.168.1.14 -w client_src.pcap
```

tshark可以是：

```
tshark -n -q -r client.pcap -Y 'ip.src==192.168.1.14' -w client_src.pcap
```

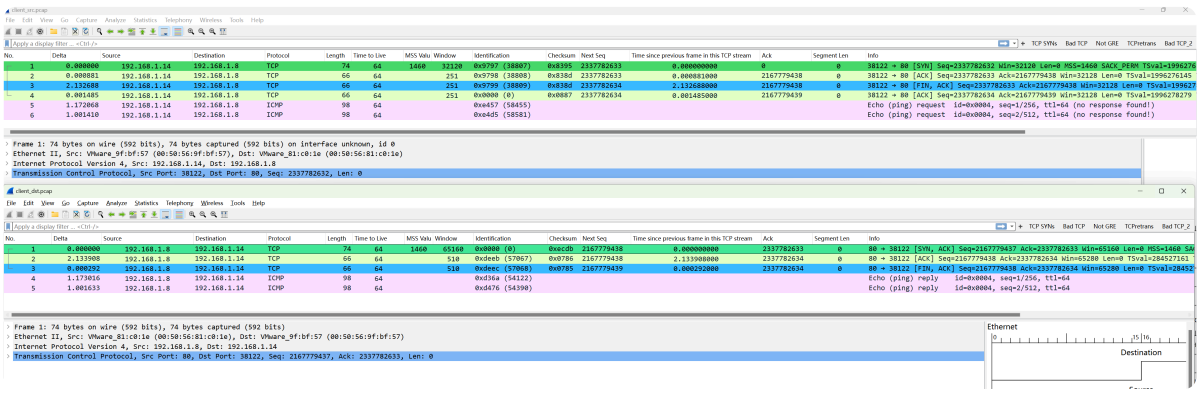
以上方式二选一即可，tcpdump更常用。

服务端进来的方向：

```
tcpdump -r client.pcap dst 192.168.1.14 -w client_dst.pcap
```

```
┌──(root㉿kali)-[~]
└─# tshark -n -q -r client.pcap -Y 'ip.src==192.168.1.14' -w client_src.pcap
Running as user "root" and group "root". This could be dangerous.

┌──(root㉿kali)-[~]
└─# tcpdump -r client.pcap dst 192.168.1.14 -w client_dst.pcap
reading from file client.pcap, link-type EN10MB (Ethernet), snapshot length 262144

┌──(root㉿kali)-[~]
└─# ls -l client_src.pcap client_dst.pcap
-rw-r--r-- 1 tcpdump tcpdump 506 Sep 27 17:11 client_dst.pcap
-rw-r--r-- 1 root    root    788 Sep 27 17:11 client_src.pcap

┌──(root㉿kali)-[~]
└─#
```

拆成两个方向的包后可以看到每个包都是单方向的报文：



2）分别对两个包进行源目的重写

首先对源方向的包重写源IP、目的IP、源MAC、目的MAC：

重写后源IP	重写后目的IP	重写后源MAC	重写后目的MAC
192.168.1.12	192.168.1.72	00:50:56:81:8e:44	00:50:56:81:be:58

对应的tcprewrite写法可以是：

```
tcprewrite --infile=client_src.pcap --outfile=client_src_rewrite.pcap -S
0.0.0.0/0:192.168.1.12 -D 0.0.0.0/0:192.168.1.72 --enet-smac=00:50:56:81:8e:44 --enet-
dmac=00:50:56:81:be:58
```

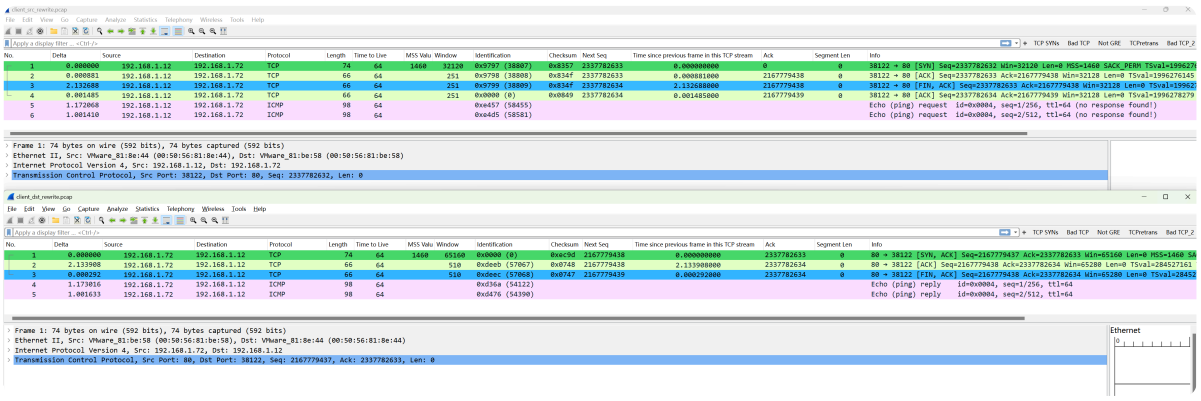
同理，我们再对服务端过来方向的包进行重写：

重写后源IP	重写后目的IP	重写后源MAC	重写后目的MAC
192.168.1.72	192.168.1.12	00:50:56:81:be:58	00:50:56:81:8e:44

```
tcprewrite --infile=client_dst.pcap --outfile=client_dst_rewrite.pcap -S
0.0.0.0/0:192.168.1.72 -D 0.0.0.0/0:192.168.1.12 --enet-smac=00:50:56:81:be:58 --enet-
dmac=00:50:56:81:8e:44
```

```
┌──(root㉿kali)-[~]
└─# tcprewrite --infile=client_src.pcap --outfile=client_src_rewrite.pcap -S 0.0.0.0/0:192.168.1.12 -D 0.0.0.0/0:192.168.1.72 --enet-smac=00:50:56:81:8e:44 --enet-dmac=00:50:56:81:be:58

┌──(root㉿kali)-[~]
└─# tcprewrite --infile=client_dst.pcap --outfile=client_dst_rewrite.pcap -S 0.0.0.0/0:192.168.1.72 -D 0.0.0.0/0:192.168.1.12 --enet-smac=00:50:56:81:be:58 --enet-dmac=00:50:56:81:8e:44

┌──(root㉿kali)-[~]
└─# ls -l client_src_rewrite.pcap client_dst_rewrite.pcap
-rw-r--r-- 1 root root 506 Sep 27 17:26 client_dst_rewrite.pcap
-rw-r--r-- 1 root root 588 Sep 27 17:26 client_src_rewrite.pcap

┌──(root㉿kali)-[~]
└─#
```

重写后，源目的IP地址、源目的MAC地址，从生产机器变为了我们的实验机器：



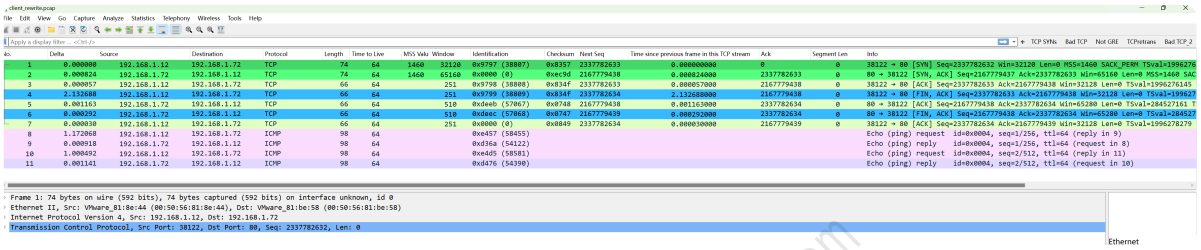
3）合并重写后的两个包

使用mergecap合并即可，默认会按照时间顺序合并：

```
mergecap -w client_rewrite.pcap client_src_rewrite.pcap client_dst_rewrite.pcap
```

```
┌──(root㉿kali)-[~]
└─# mergecap -w client_rewrite.pcap client_src_rewrite.pcap client_dst_rewrite.pcap

┌──(root㉿kali)-[~]
└─# ls -l client_rewrite.pcap
-rw-r--r-- 1 root root 1384 Sep 27 17:31 client_rewrite.pcap

┌──(root㉿kali)-[~]
└─# _
```

合并后的包，五元组中的其中四元组终于被正常修改，重写成了我们想要的实验对象：



后续可以对此包进行tcpreplay重放操作。

4）重放报文

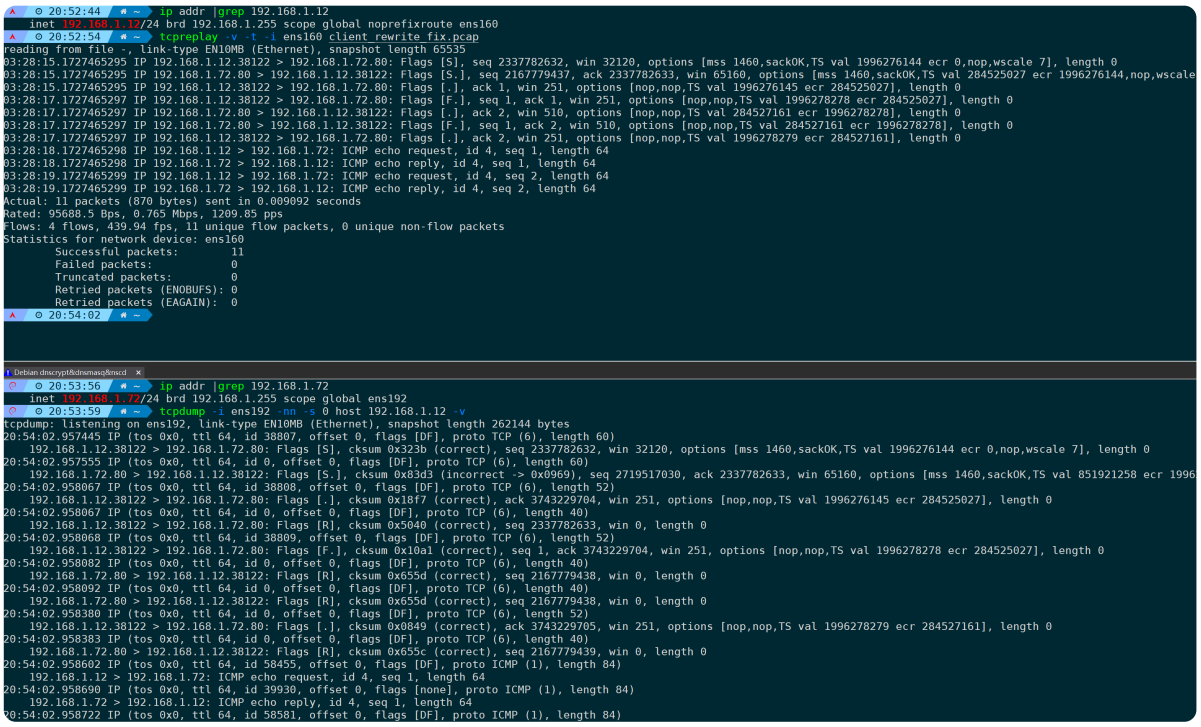
重放之前，别忘了更新一下checksum：

```
tcprewrite --infile=client_rewrite.pcap --outfile=client_rewrite_fix.pcap --fixcsum
```

之后把client_rewrite_fix.pcap需要被重放的报文，放到192.168.1.12源端上进行重放（当然你也可以放到服务端进行重放）：

```
tcpreplay -v -t -i ens160 client_rewrite_fix.pcap
```

在服务端抓包也能正常收到客户端发过来的重放包：



5）另一种可选方案（tcpprep、endpoints、cache）

上面拆包、处理包再合并包的步骤较为繁琐，还有一种方式时，使用tcpprep来配合实现。

还是下面这个场景为例，已知client.pcap文件涉及到的信息如下：

客户端	服务端	涉及协议
192.168.1.14	192.168.1.8	ICMP、80/TCP

假设因为生产环境的原因，1.14生产机器不能再用于重放，此时我们想要在别的客户端和服务端进行重放实验，譬如在1.12进行重放，重放给1.72，信息如下：

客户端	客户端MAC	服务端	服务端MAC	涉及协议
192.168.1.12	00:50:56:81:8e:44	192.168.1.72	00:50:56:81:be:58	ICMP、80/TCP

使用tcpprep将pcap文件中的数据包按照客户端和服务器进行分类。

方式一：将client.pcap文件中IP为192.168.1.14/32的设置为client端，剩余的视为为server端：

```
tcpprep -c 192.168.1.14/32 -i client.pcap -o client.cache
```

此时生成了一个cache缓存文件，后面tcprewrite重写的时候会用到这个缓存文件，生成后内容如下：

```
┌──(root㉿kali)-[~]
└─# tcpprep -c 192.168.1.14/32 -i client.pcap -o client.cache

┌──(root㉿kali)-[~]
└─# cat client.cache
tcpprep04
1-c 192.168.1.14/32 -i client.pcap -o client.cache

┌──(root㉿kali)-[~]
└─#
```

方式二：让tcpprep采用自动/client分包模式生成缓存文件：

```
tcpprep -a client -i client.pcap -o client.cache
```

内容如下：

```
┌──(root㉿kali)-[~]
└─# tcpprep -a client -i client.pcap -o client.cache

┌──(root㉿kali)-[~]
└─# cat client.cache
tcpprep04
(-a client -i client.pcap -o client.cache

┌──(root㉿kali)-[~]
└─#
```

自动模式下tcpprep认为有以下特征的IP为client端：

- 发TCP SYN包的；
- 发DNS请求包的；
- 收到ICMP-Port Unreachable端口不可达的；

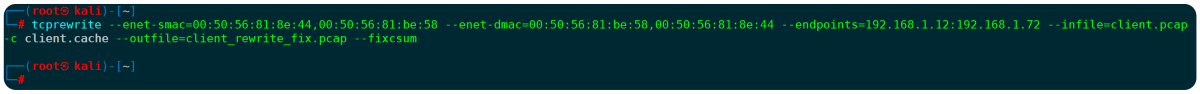
认为有以下特征的视为server端：

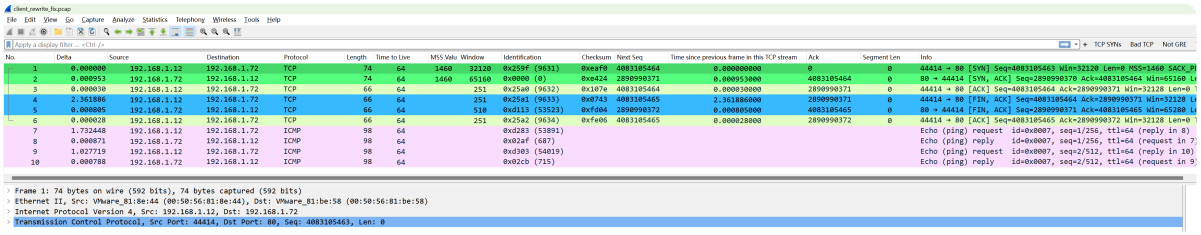
- 发TCP SYN,ACK的；
- 发DNS响应的；
- 发ICMP-Port Unreachable端口不可达的；

如果你已经清晰知道哪些IP是源IP，建议采用方式一。

紧接着，我们使用tcprewrite修改源IP/目的IP/源MAC/目的MAC，并且更新checksum：

```
tcprewrite --enet-smac=00:50:56:81:8e:44,00:50:56:81:be:58 --enet-
dmac=00:50:56:81:be:58,00:50:56:81:8e:44 --endpoints=192.168.1.12:192.168.1.72 --
infile=client.pcap -c client.cache --outfile=client_rewrite_fix.pcap --fixcsum
```





此时我们已经一次性修改完了需要重放的报文文件的源IP/目的IP/源MAC/目的MAC，一切符合预期，之后按照上面的重放步骤正常进行即可。

6.重写源端口/目的端口再进行重放

当业务端口发生变化，或者源/目的端口已经被占用，需要修改端口进行重放数据包时，tcprewrite的-r/--portmap参数就派上了用场，此参数可以重写TCP、UDP的源目的端口，用法可以是端口范围、多个具体端口修改为指定端口。

```
r string, --portmap=string
    Rewrite TCP/UDP ports.  This option may appear up to 9999 times.

    Specify a list of comma delimited port mappings consisting of colon delimited port number pairs.  Each colon delimited port pair consists of the port to match followed by the port number to rewrite.

    Examples:
        --portmap=80:8000 --portmap=8080:80    # 80->8000 and 8080->80
        --portmap=8000,8080,88888:80           # 3 different ports become 80
        --portmap=8000-8999:80                 # ports 8000 to 8999 become 80
```

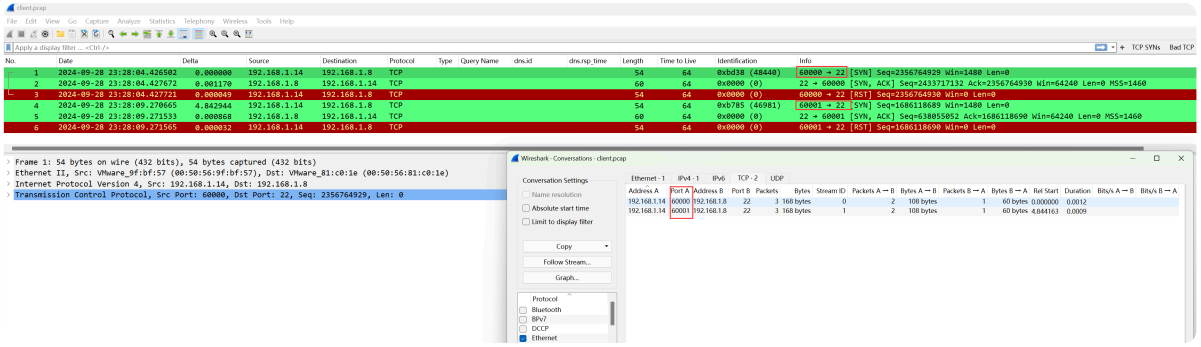
比如以下场景，pcap包里的原始业务访问五元组：

源IP	源端口	目的IP	目的端口	协议
192.168.1.14	60000,60001	192.168.1.8	22	TCP

由于某种原因，源端口60000/60001、目的端口22都被其它业务占用，重放时，需要修正为最新的业务端口：

源IP	源端口	目的IP	目的端口	协议
192.168.1.14	60002	192.168.1.8	22222	TCP

此时已知client.pcap报文内容如下：



即客户端用60000和60001作为源端口往服务端22端口分别发了一次TCP探测。

重放时，需要把60000和60001端口均修改为60002端口，目的端口22修改为22222端口输出为client_port_changed.pcap，那么tcprewrite可以这么写：

```
tcprewrite --portmap=60000-60001:60002 --portmap=22:22222 --infile=client.pcap --
outfile=client_port_changed.pcap
```

```
┌──(root㉿kali)-[~]
└─# tcprewrite --portmap=60000-60001:60002 --portmap=22:22222 --infile=client.pcap --outfile=client_port_changed.pcap

┌──(root㉿kali)-[~]
└─#
```

同时，更新下csum值，输出为client_port_changed_fix.pcap：

```
tcprewrite --fixcsum --infile=client_port_changed.pcap --
outfile=client_port_changed_fix.pcap
```

此时可能会弹出警告，其中有些包不能被--fixcsum参数强制更新csum进行修改：

```
┌──(root㉿kali)-[~]
└─# tcprewrite --fixcsum --infile=client_port_changed.pcap --outfile=client_port_changed_fix.pcap
Warning: skipping packet 2 because caplen 60 minus L2 length 14 does not equal IPv4 header length 44. Consider option '--fixhdrlen'.
Warning: skipping packet 5 because caplen 60 minus L2 length 14 does not equal IPv4 header length 44. Consider option '--fixhdrlen'.

┌──(root㉿kali)-[~]
└─#
```

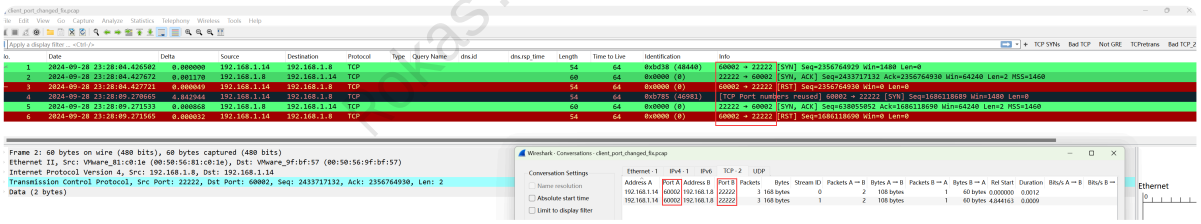
可以使用--fixhdrlen作为替代，即更改TCP/IP头部以匹配数据包长度：

```
-C, --fixcsum               Force recalculation of IPv4/TCP/UDP header checksums
    --fixhdrlen             Alter IP/TCP header len to match packet length
```

```
tcprewrite --fixhdrlen --infile=client_port_changed.pcap --
outfile=client_port_changed_fix.pcap
```

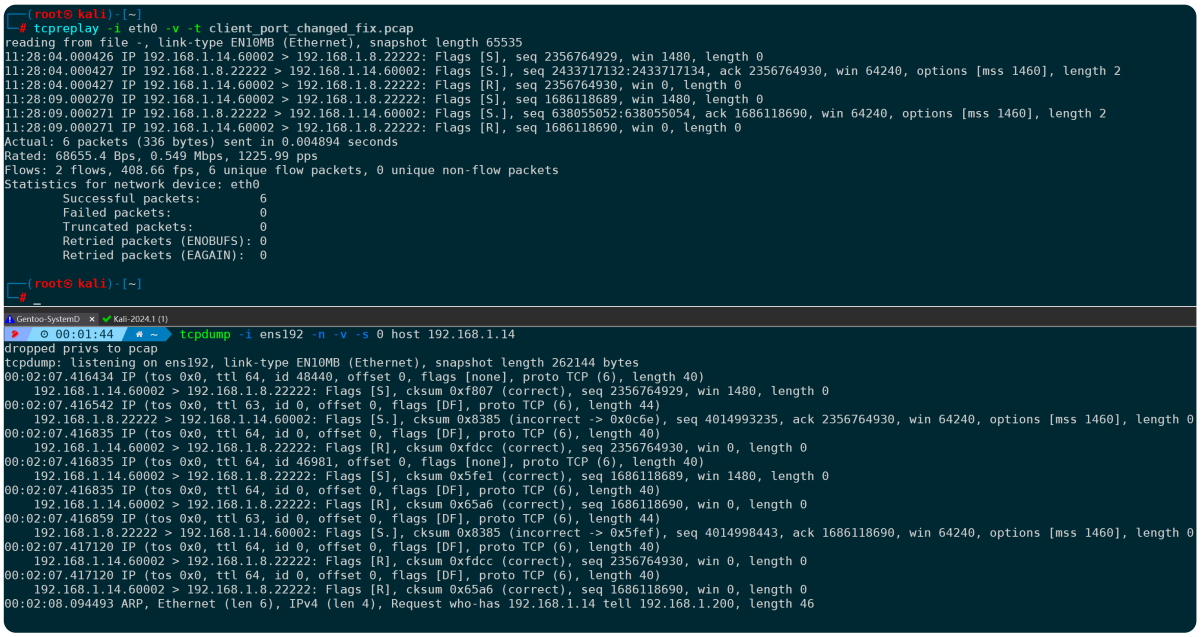
```
┌──(root㉿kali)-[~]
└─# tcprewrite --fixhdrlen --infile=client_port_changed.pcap --outfile=client_port_changed_fix.pcap

┌──(root㉿kali)-[~]
└─#
```

此时使用Wireshark看一下最终我们需要重放的包client_port_changed_fix.pcap：



源目的端口已经成功修改，接下来将这个pcap包正常进行重放即可。

```
tcpreplay -i eth0 -v -t client_port_changed_fix.pcap
```



在服务端抓包可以看到，已经接收到了客户端重放的请求，并且csum值正确。

7.一些控制重放速率和循环次数的参数

如果需要控制重放速率或循环重放的次数，下面这些参数较为有用：

参数	含义
-L number/--limit=number	设置重放的包量，重放到指定包量后停止重放；默认-1，不限制。
--duration=number	设置重放的秒数，超过指定时间后停止重放；默认-1，不限制。
-x string/--multiplier=string	设置重放的倍率，比如-x 2.0指定2倍速进行重放，-x 0.7指定0.7倍进行重放。
-p string/--pps=string	设置每秒发包量，比如-p 200，则为每秒发200个包，-p 0.25，一分钟发15个包。
-M string/--mbps=string	设置重放的带宽大小，比如-M 1000以1Gbps的速率发送，前提发包机器的带宽和机器性能不是瓶颈。
-t/--topspeed	以最快速度重放。
--pps-multi=number	每个时间间隔要发送的包量，和前面的-p参数必须一起使用，比如 -p 200 --pps-multi=10 含义为每10秒发送200个包。
-l number/--loop=number	设置循环重放的次数，默认为1次。
--loopdelay-ms=number	设置循环和循环之间的间隔时间，单位ms，这个参数必须要和-l/--loop参数一起出现。
--loopdelay-ns=number	和上一个参数唯一区别是，它的单位是ns（纳秒），必须和-l/-loop参数一起出现。

比如我想设定循环重放10次，循环和循环之间间隔100ms，并且以最快的速度重放，可以是：

```
tcpreplay -i eth0 --loop=10 --loopdelay-ms=100 -t client_replay.pcap
```

```
┌──(root㉿kali)-[~]
└─# time tcpreplay -i eth0 --loop=10 --loopdelay-ms=100 -t client_replay.pcap
Actual: 170 packets (12760 bytes) sent in 0.903379 seconds
Rated: 14124.7 Bps, 0.112 Mbps, 188.18 pps
Flows: 4 flows, 4.42 fps, 170 unique flow packets, 0 unique non-flow packets
Statistics for network device: eth0
        Successful packets:        170
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0

real    1.00s
user    0.00s
sys     0.01s
cpu     0%

┌──(root㉿kali)-[~]
└─#
```

或者，我想设置重放完10个包后停止重放，并且发包速率为每秒2个，可以是：

```
tcpreplay -i eth0 --limit=10 --pps=2 client_replay.pcap
```

```
┌──(root㉿kali)-[~]
└─# time tcpreplay -i eth0 --limit=10 --pps=2 client_replay.pcap
Actual: 10 packets (772 bytes) sent in 4.50 seconds
Rated: 171.5 Bps, 0.001 Mbps, 2.22 pps
Flows: 4 flows, 0.88 fps, 10 unique flow packets, 0 unique non-flow packets
Statistics for network device: eth0
        Successful packets:        10
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0

real    4.58s
user    1.14s
sys     3.37s
cpu     98%
```

其它参数不再一一演示，可根据需求搭配使用，有些参数和参数之间会产生互斥，不能同时出现。

8.重放前将数据包预加载到内存

一个比较重要的提高性能的参数**-K/--preload-pcap**，此选项在开始发送之前将指定的pcap加载到内存（RAM）中，以提高重放性能，同时对启动性能有一定的影响，因为在 tcpreplay开始发送数据包之前，会有一个初始的延迟，用来将所有数据包加载到内存中。此选项还抑制了每次迭代的流统计数据收集，可以显著减少内存占用。

这样做的好处是可以减少磁盘I/O操作，因为数据包直接从内存中读取而不是从磁盘读取，从而显著提高数据包发送的速度和整体重放性能，特别是对于一些pcap大包，或者需要循环多次重放的情况效果更为明显。

在配合--loop参数使用的情况下，流量统计信息是基于首次循环迭代中收集的数据和用户提供的选项来预测的，这可以显著减少内存使用量，因为不需要为每次循环都存储详细的统计数据。即使在多次循环中，数据包也只会被预加载一次，即在第一次循环开始前。

比如我想循环500次重放，循环和循环之间间隔10毫秒延时，以最快速度重放完所有包，可以是：

```
tcpreplay -i eth0 -K -t --loop=500 --loopdelay-ms=10 client_replay.pcap
```

```
┌──(root㉿kali)-[~]
└─# time tcpreplay -i eth0 -K -t --loop=500 --loopdelay-ms=10 client_replay.pcap
File Cache is enabled
Actual: 8500 packets (638000 bytes) sent in 5.12 seconds
Rated: 124545.8 Bps, 0.996 Mbps, 1659.30 pps
Flows: 4 flows, 0.78 fps, 17 unique flow packets, 0 unique non-flow packets
Statistics for network device: eth0
        Successful packets:        8500
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0

real    5.21s
user    0.01s
sys     0.08s
cpu     1%

┌──(root㉿kali)-[~]
└─#
```

第一行输出了"File Cache is enabled"，说明缓存被启用了，这个过程可能会有一定延时，是将包文件预加载到内存的耗时。

五、总结

到此，本文系统地介绍了tcpreplay及其相关工具在网络测试、安全评估和故障诊断中的应用。首先，概述了tcpreplay的主要功能，包括重放网络流量、调整重放速度和循环次数等，并强调了其在重现网络交互、全面测试网络行为等方面的重要性。接着，详细讨论了完整重放与筛选重放的优缺点及适用场景，帮助读者根据实际需求选择合适的重放策略。

同时演示了如何使用tcpdump、tshark等工具进行报文筛选并配合tcprewrite进行重写，以便更精确地控制测试流量，并通过实战演练展示了如何修改源IP、目的IP、源MAC、目的MAC等信息进行流量控制。

最后，文章总结了提高重放性能的关键参数和技巧，如预加载pcap文件到内存、设置重放倍率等。这些内容不仅提高了网络测试的效率和准确性，还为网络安全人员和测试人员提供了实用的指导。最后，希望本文对你能有所帮助。