# 实验1

先清空DNS缓存：

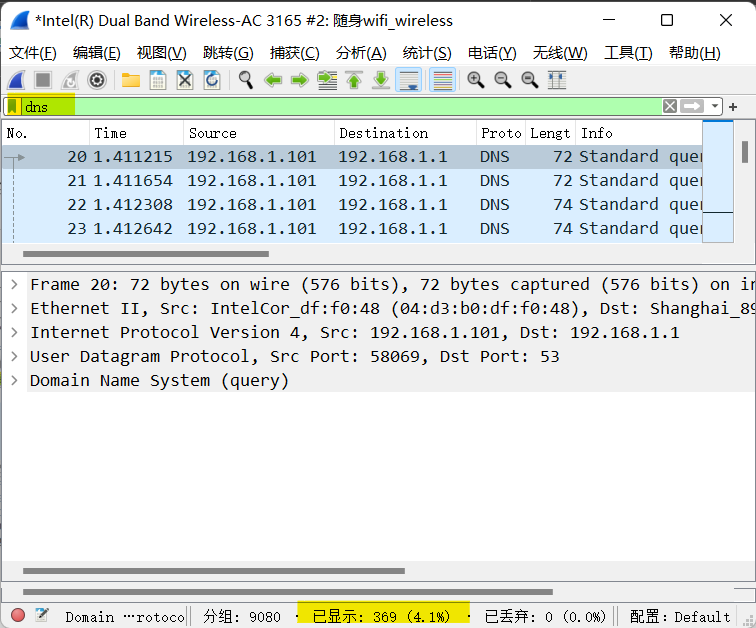
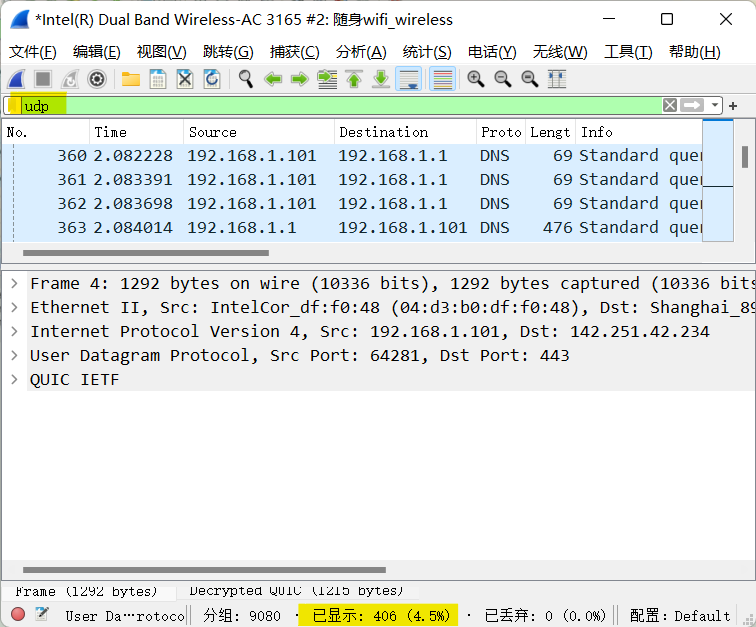


查看[www.sohu.com的IP](http://www.sohu.com的IP)地址：



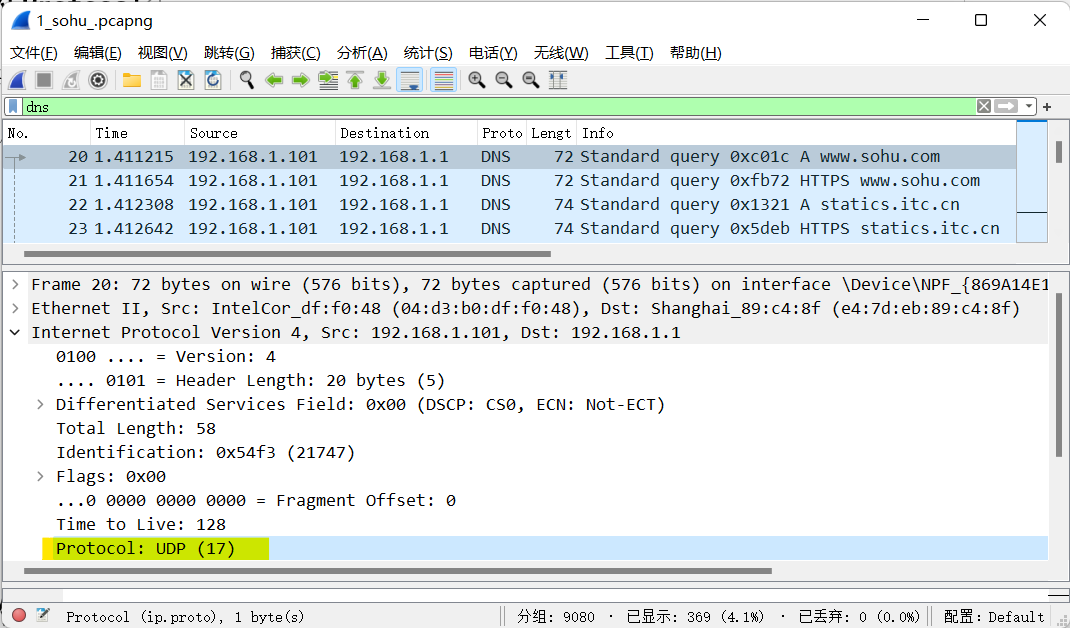
## 1.1 统计数据Displayed

如下图所示，UDP的数量为406，DNS的数量为369，UDP的数量必然大于等于DNS的数量。因为DNS作为运输层协议，在网络层与其他运输层协议复用了UDP，即DNS都使用了UDP，但UDP不仅运输DNS。这决定了UDP的数量一定大于等于DNS的数量。



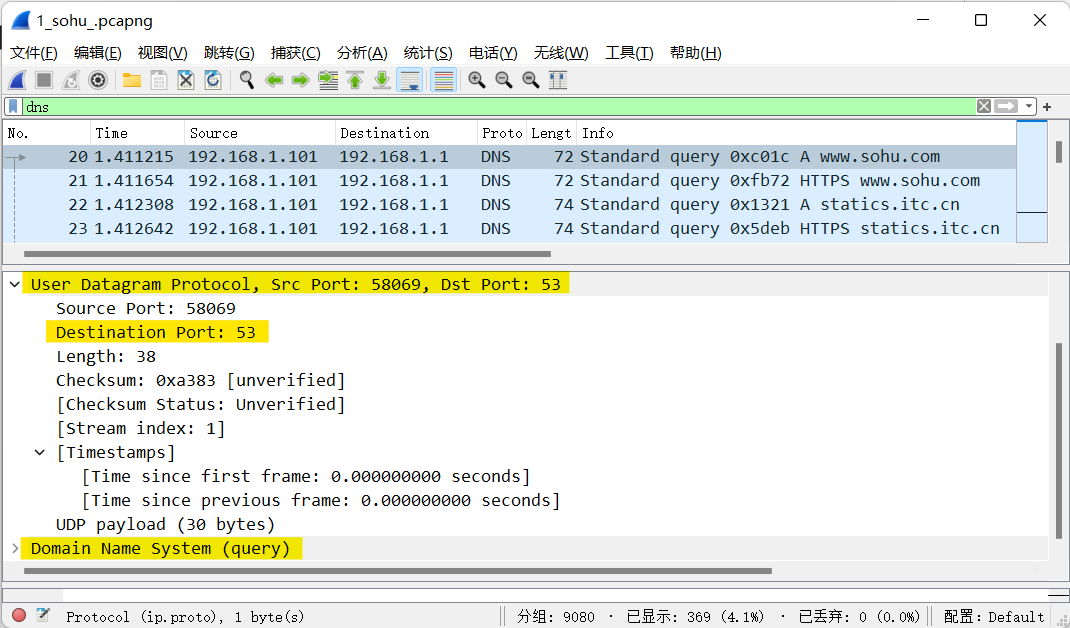
## 1.2 DNS的Protocal

IP数据报首部Protocal字段值为十六进制0x11，即十进制17，代表网络层上一层运输层的协议为UDP协议。



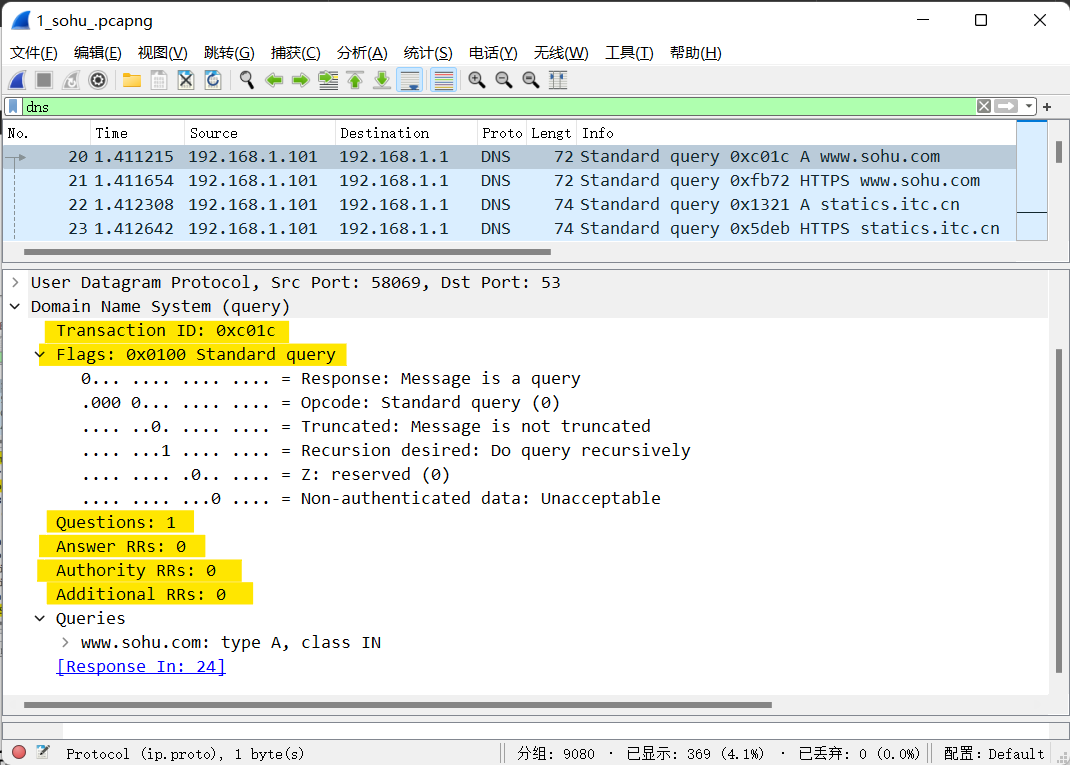
## 1.3 DNS的端口号

DNS在网络层使用UDP协议，查看网络层details，UDP目的端口为十六进制0x35，十进制53，即DNS的目的端口号为53。



## 1.4 DNS首部分析

DNS协议数据单元首部长度为12bytes，包括事务 ID、标志、问题计数、回答资源记录数、权威名称服务器计数、附加资源记录数这 6 个字段。



Transaction ID/事务ID可以区分两个DNS是否是相互匹配的同组请求应答；

Flags/标志标记了DNS包的不同特性和功能；

Questions/问题计数表示DNS 查询请求的数目；

Answer RRs/回答资源记录数表示DNS 响应的数目；

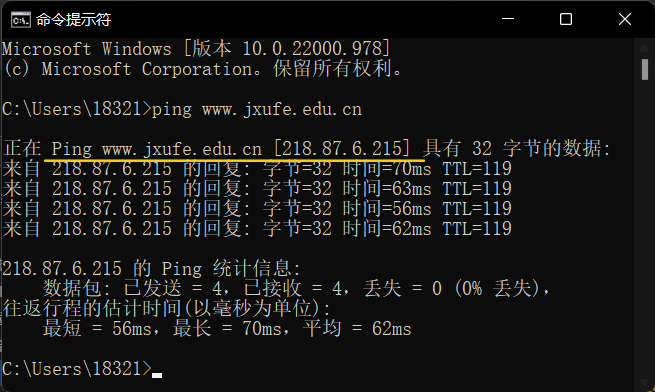
Authority RRs/权威名称服务器计数表示权威名称服务器的数目；

Additional RRs/附加资源记录数表示额外的、权威名称服务器对应 IP 地址的数目）。

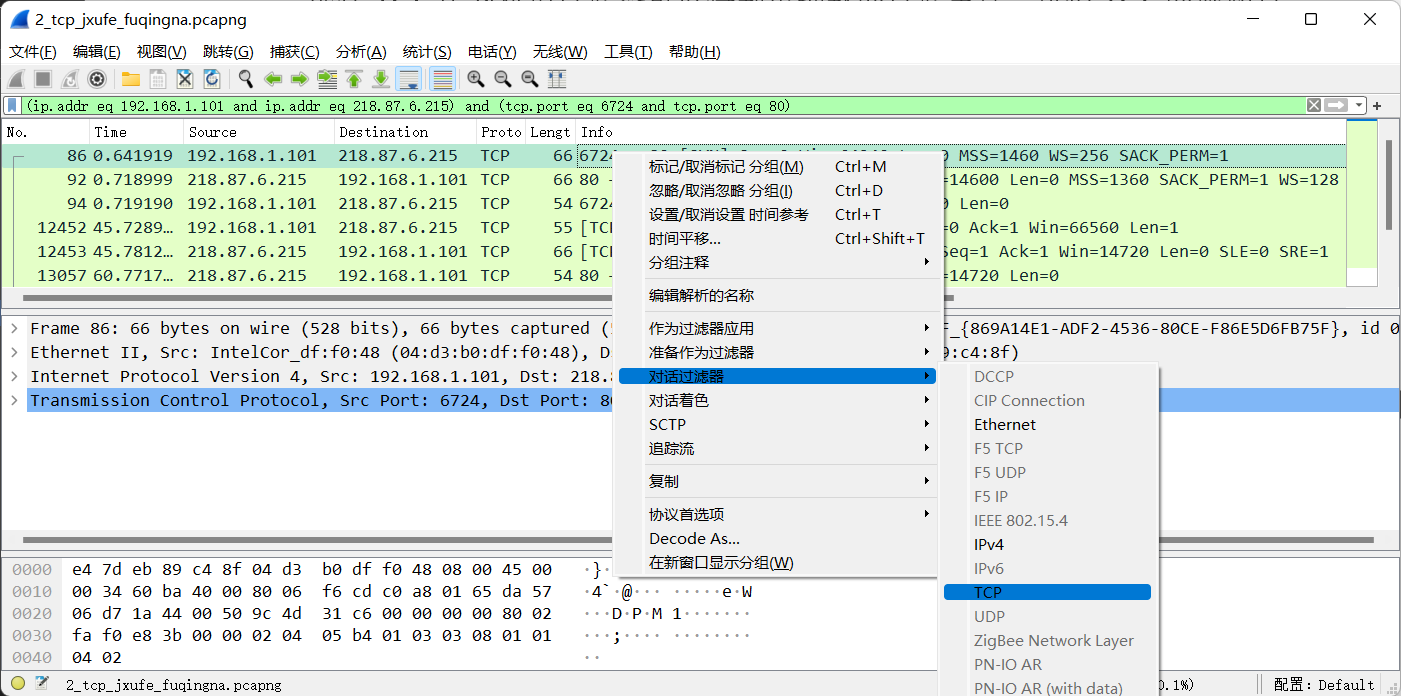
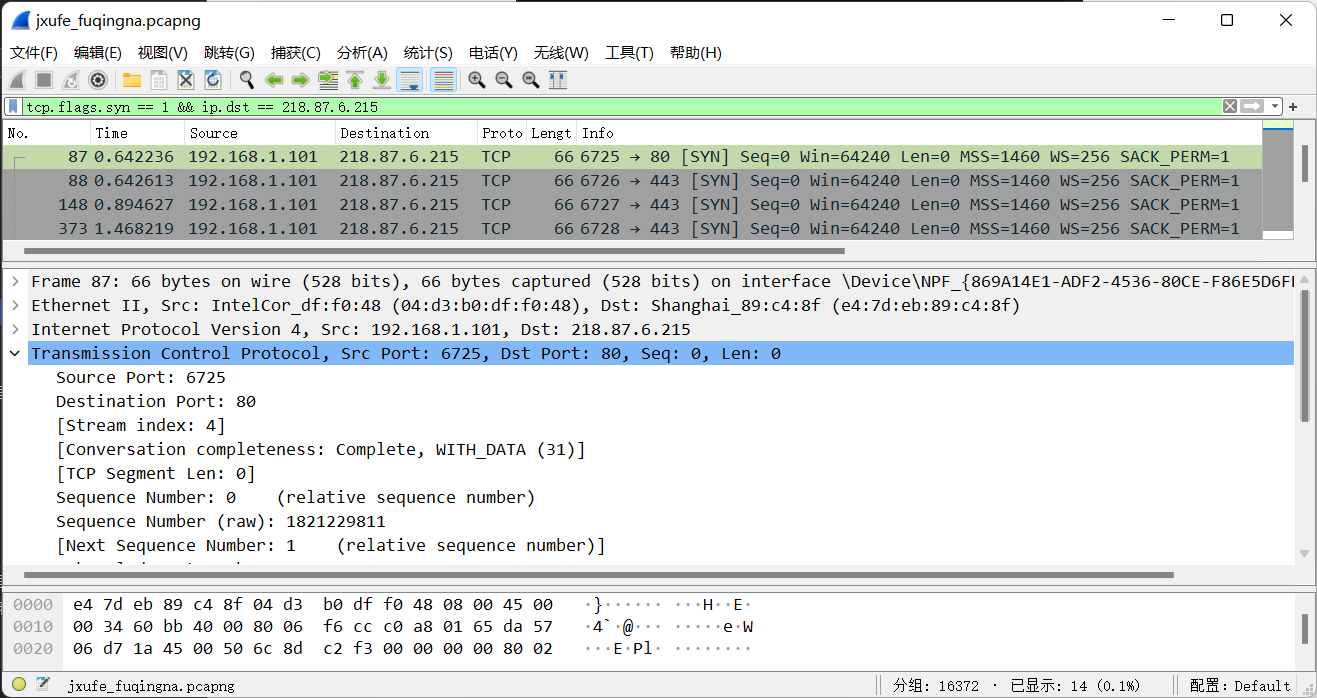
# 实验2

## 1.1 第一次握手

通过ping命令找到[www.jxufe.edu.cn的IP](http://www.jxufe.edu.cn的IP)地址：218.87.6.215。

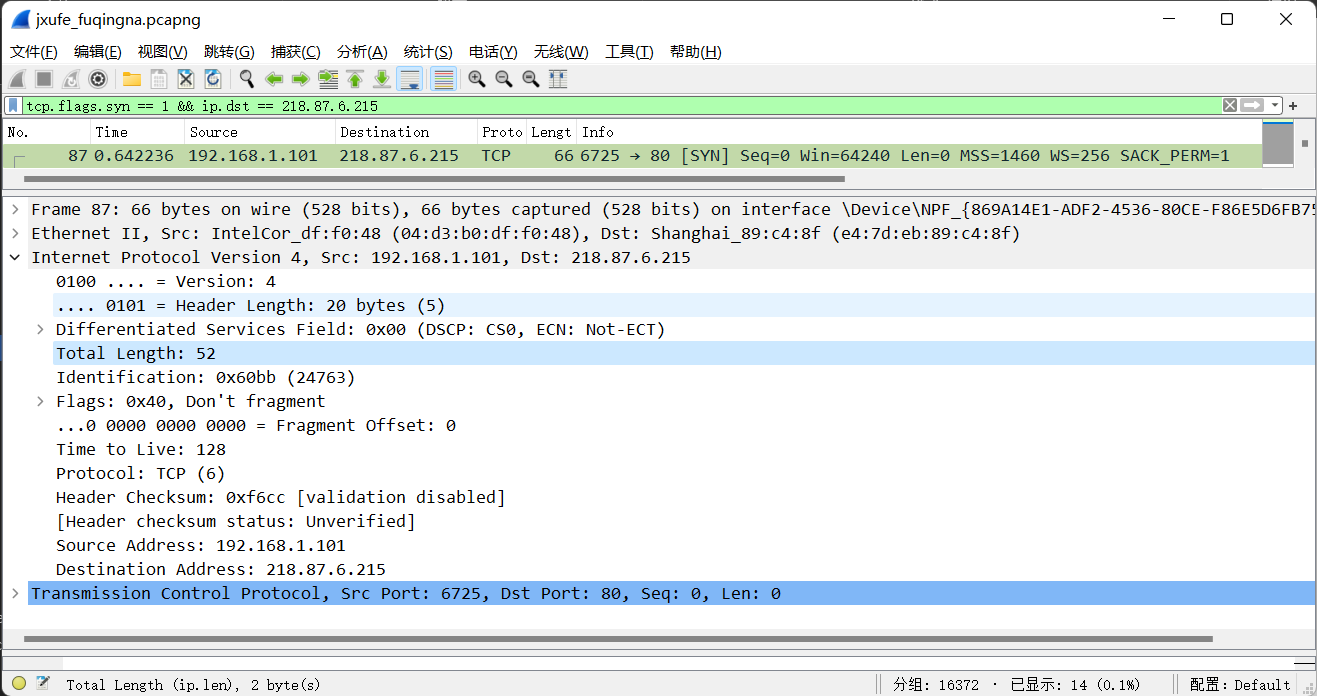
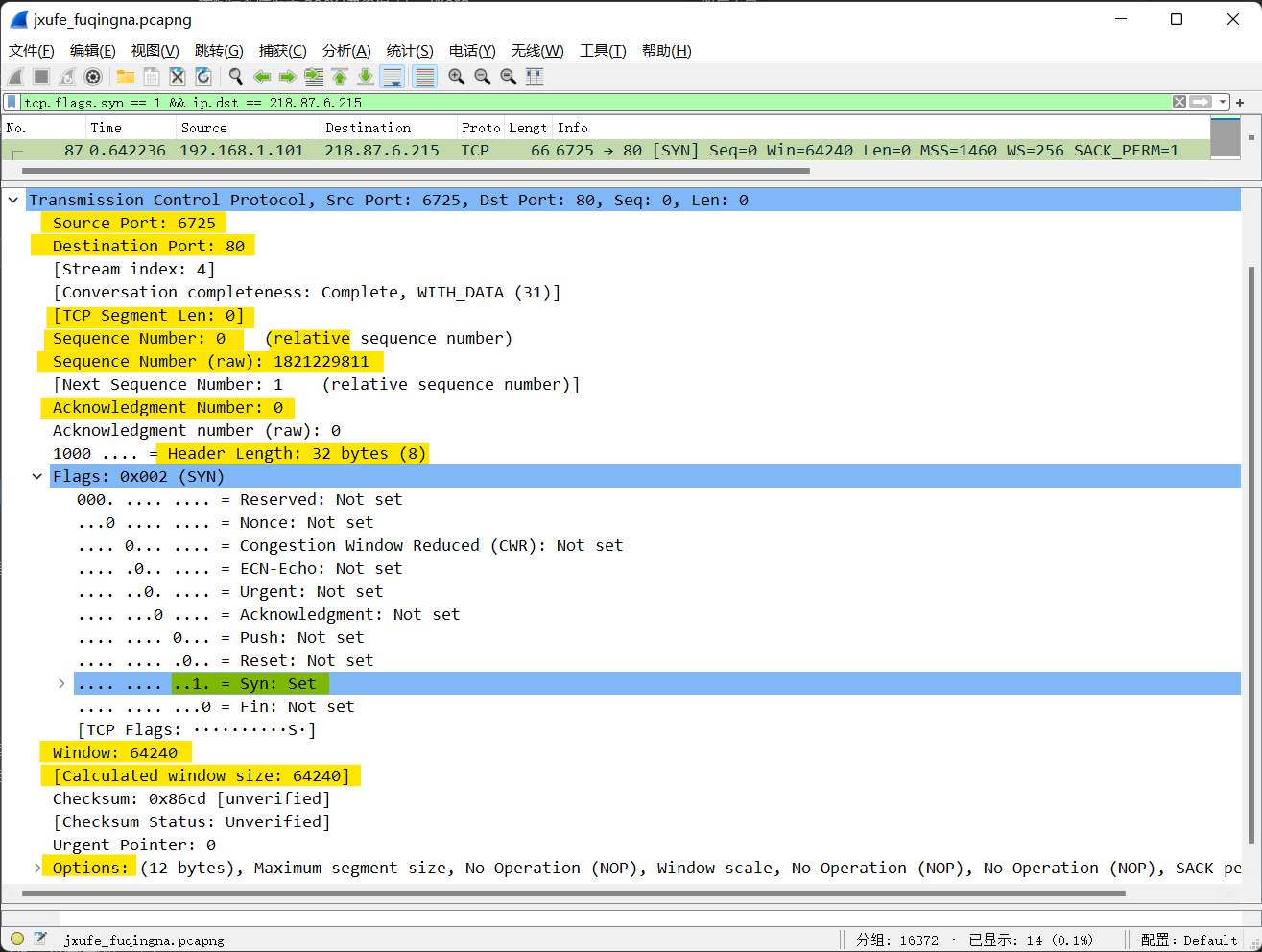


将Wireshark的捕获过滤设置为TCP，并用tcp.flags.syn == 1 && ip.addr == 218.87.6.215（tcp.flags.syn == 1 && ip.dst == 218.87.6.215也可以得到TCP起始点）找到与jxufe的TCP会话的起始点87号帧：

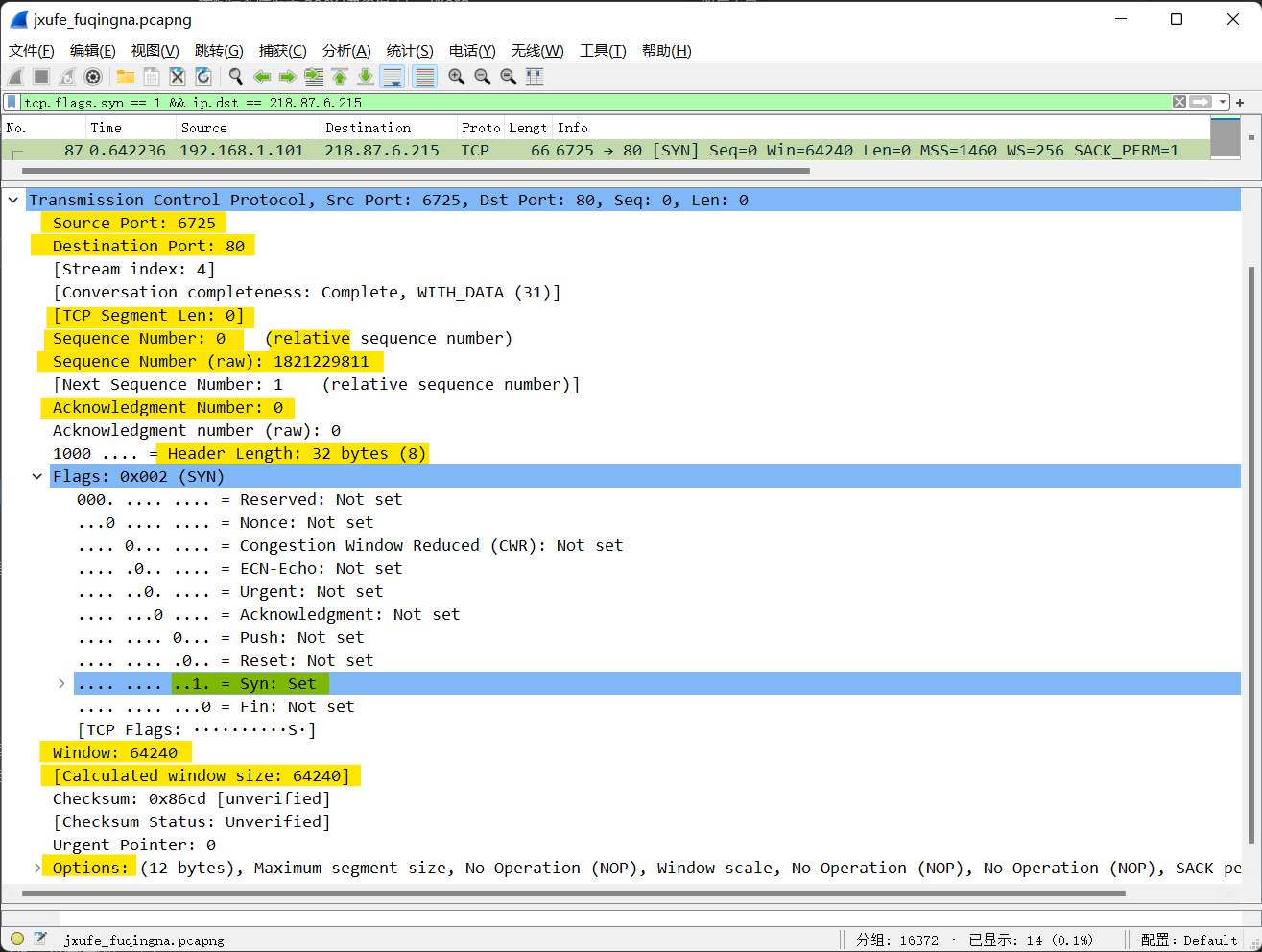


选择TCP作为对话过滤器可以将“与jxufe的TCP会话”作为过滤条件来查看包列表。可以观察到在这样的显示过滤的条件中，包括TCP的源和目的端口都被列入过滤条件。

查看87号TCP起始包的包细节：

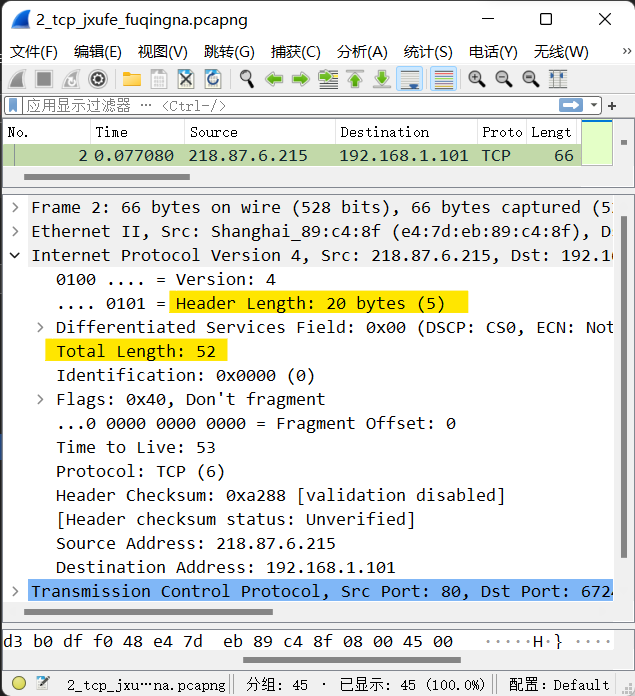
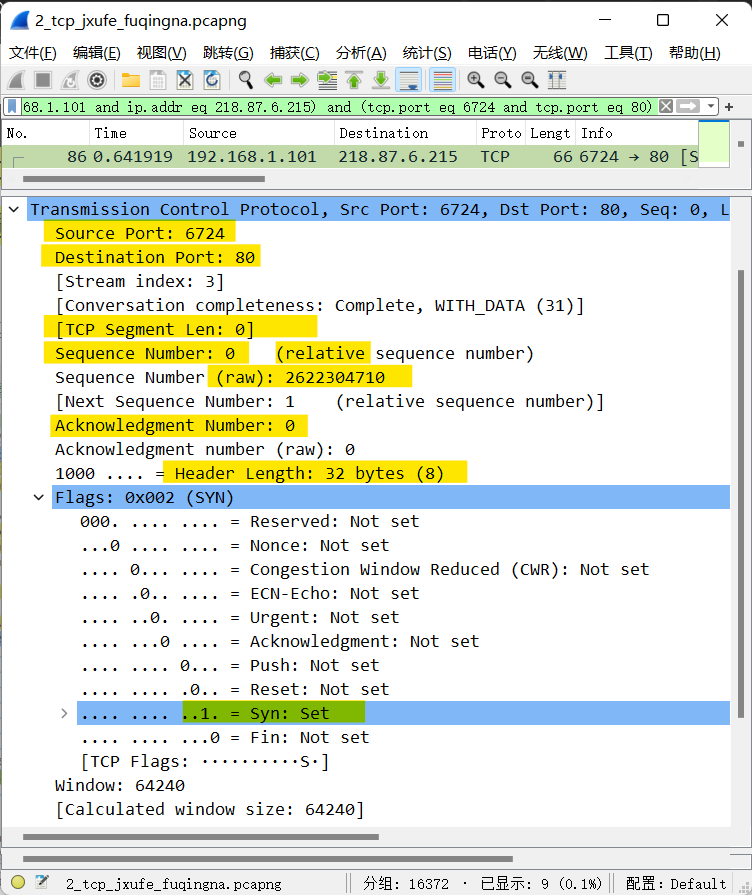


### 1.1.1 端口



源端口：6725，属于用户端口的范围；目的端口：80，说明其上层是HTTP。

### 1.1.2 TCP各部分长度



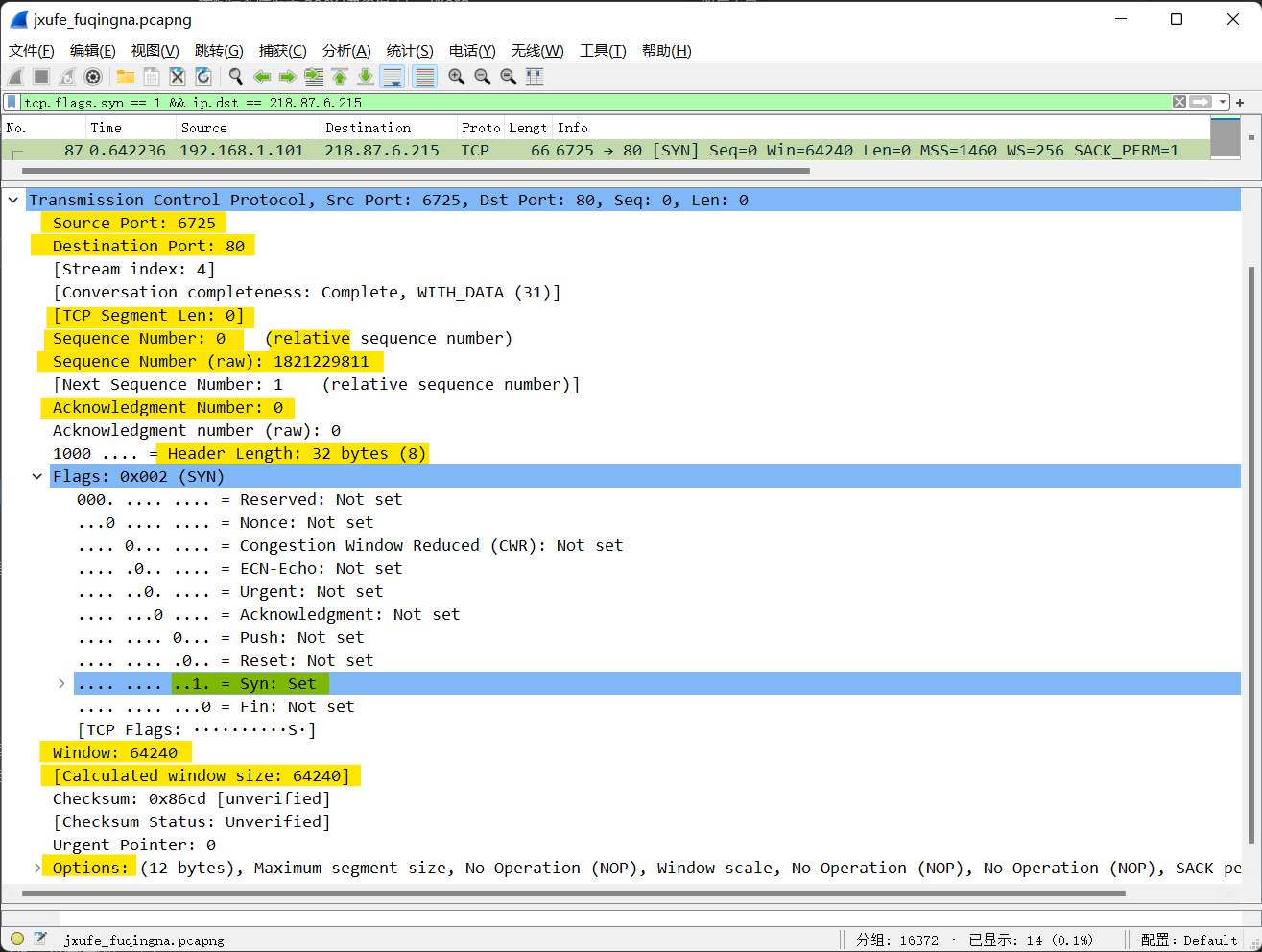
TCP首部中有Header length字段，单位为4bytes，在87号TCP起始包中，Header length为8\*4=32bytes。

TCP首部没有标注TCP的数据长度，但是wires hark提供了TCP Segment Len=0，可知TCP数据部分长度为0，因为这只是一个想要建立连接的TCP包，并没有传送数据。

TCP首部没有标注TCP的总长度，但是通过TCP Segment Len+Header Length可知TCP总长度为32 bytes，也可以通过IP数据报的IP Total length- IP Header length得到TCP总长度为32 bytes。

wires hark可以通过IP数据报的IP Total length- IP Header length得到IP数据报的数据部分长度为52-20=32，IP数据报的数据部分长度等于TCP总长度，TCP总长度-TCP Header Length=TCP数据部分长度32-32=0，从而得到TCP Segment Len=0。

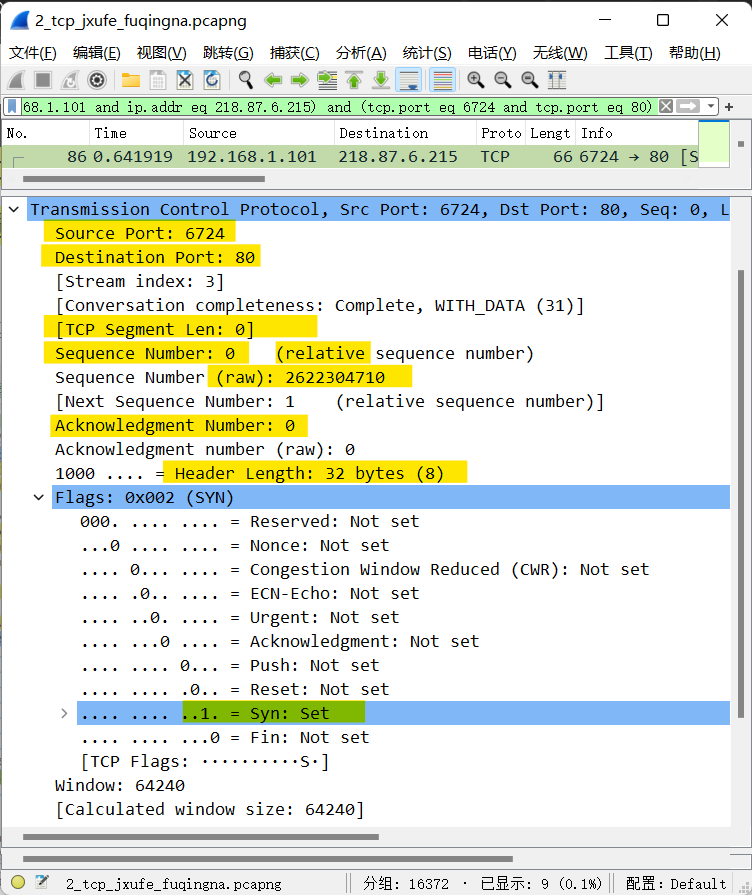
### 1.1.3 序列号和确认序号



序列号经过wires hark的处理给出了两种形式，一种是relative的相对序列号，一种是raw未经处理的绝对序列号。通过调整协议首选项protocol preference，可以选择是否关闭相对序列号relative sequence number。Raw sequence number是一个随机的数，是出于防止Ddos攻击（通过伪造IP地址，向服务器发送大量TCP syn请求，不发送回应，使得服务器占用大量资源处理这种TCP半挂起状态从而死机）而设置的，若设置随机seq，攻击方不知道seq的值，就无法假冒TCP syn包，通过Ddos攻击设备了；

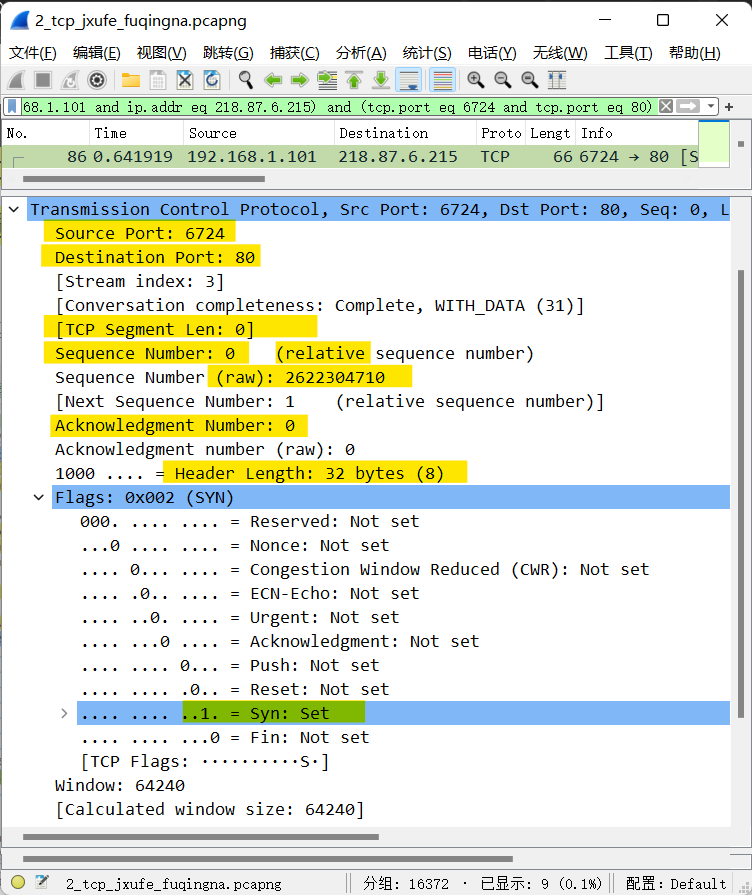
相对序列号为零，即相对于客户端的发送缓存区，它的排序为0，序列号为0。

在客户端的发送缓存中，下一个要发送的字节序列号为1。这说明第一次握手占用了一个序列号。



确认序号为0，因为服务器什么都没有返回，客户端不知道下一个从服务器发回来的包的序号应该是什么，所以填充0。

### 1.1.4 flags

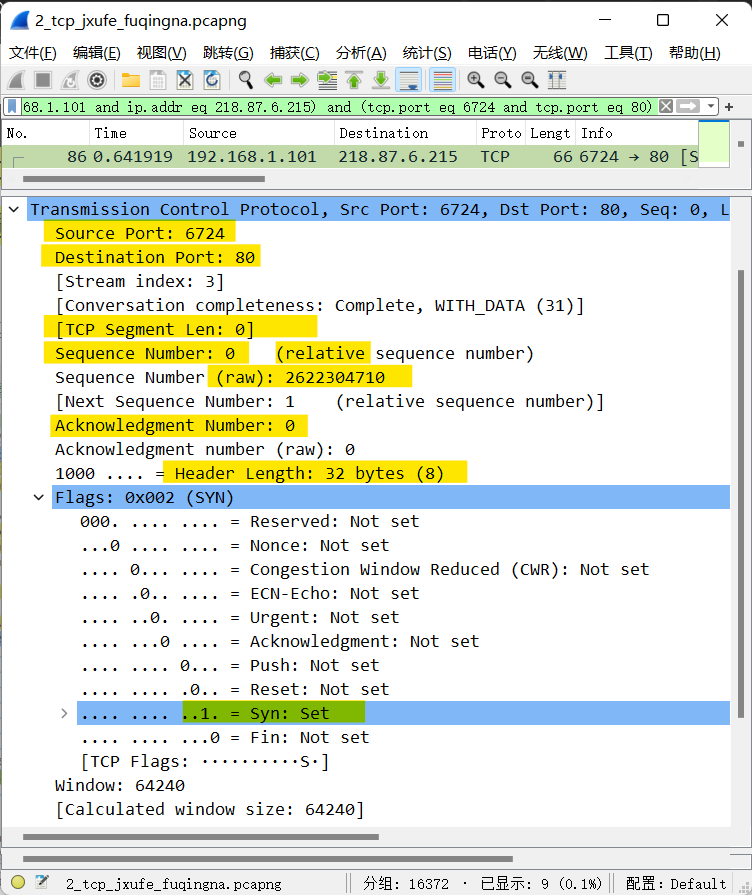


在87号TCP起始包中，flags字段为十六进制0x002，即十进制2，二进制10，倒数第二位的syn字段为1，即设置了syn，只有在前两次握手中 SYN 才为 1，而本packet就是第一次握手。

ACK=0，代表这不是一个确认的TCP包。

SYN=1， ACK=0表示连接请求报文段，即第一次握手。

### 1.1.5 窗口

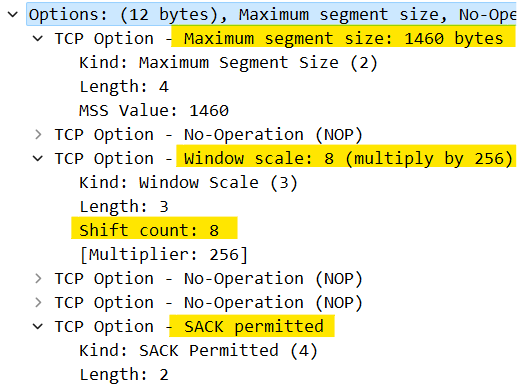


窗口大小为64240，这是wires hark根据选项字段中的Window scale调整后的结果。在本TCP中，Window scale= 28 =256 bytes。

即客户端告知服务器，下一次我（客户端）能接收64240字节的数据。

### 1.1.6 选项

选项字段长12bytes。因为TCP首部包含固定20bytes字段和变长的选项填充字段，由1.1.2可知，本TCP包的Header length为32bytes，即包含20bytes固定长度字段，和12bytes的选项与填充字段。



MSS为1460bytes，表示在这段TCP会话中不包含TCP首部的最大数据长度为1460bytes。这样设置的原因与以太网MTU=1518有关：

1518=18以太网帧头帧尾+1500IP数据报

=18以太网帧头帧尾+20IP首部+1480IP数据部分

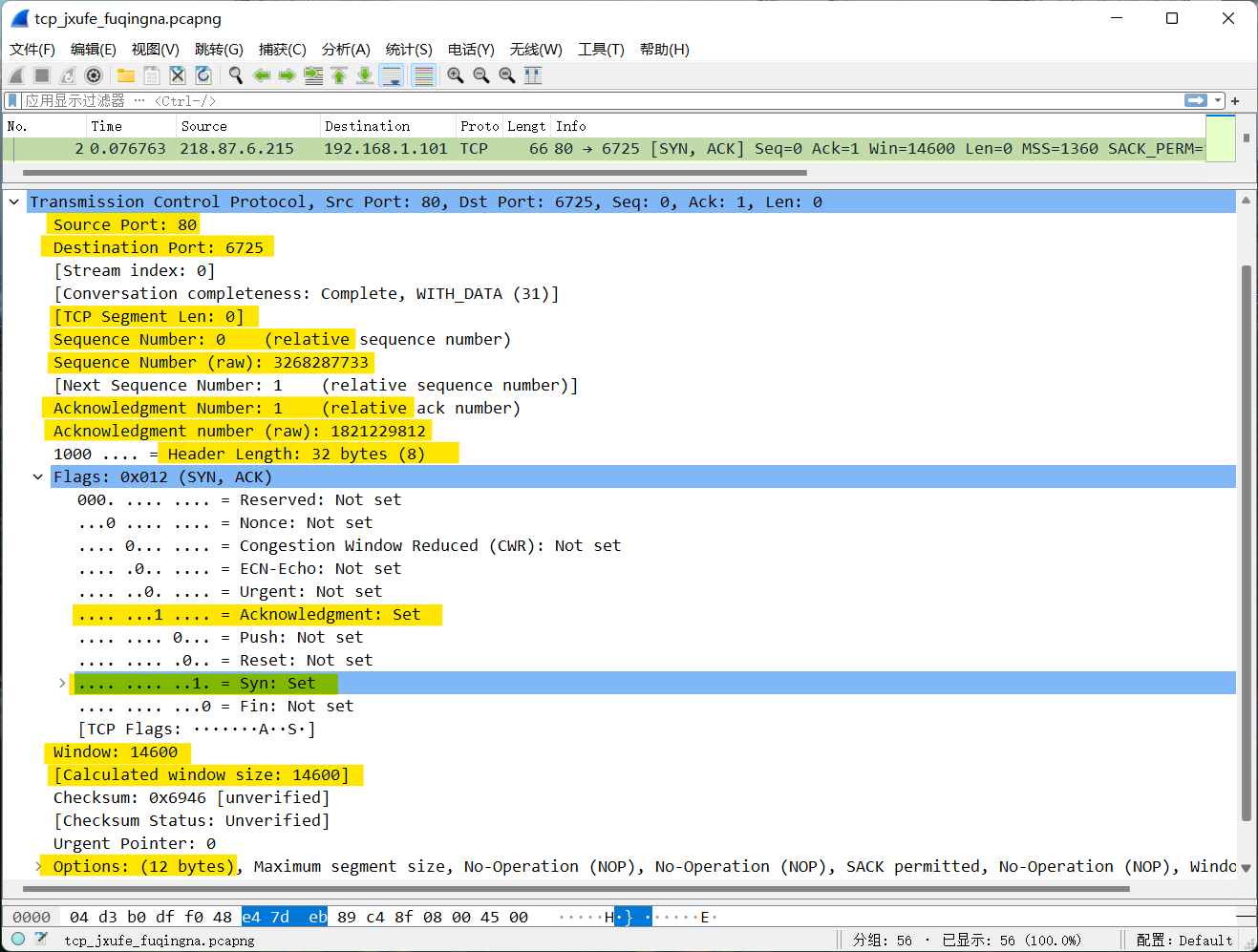
=18以太网帧头帧尾+20IP首部+20TCP固定首部+1460TCP数据部分

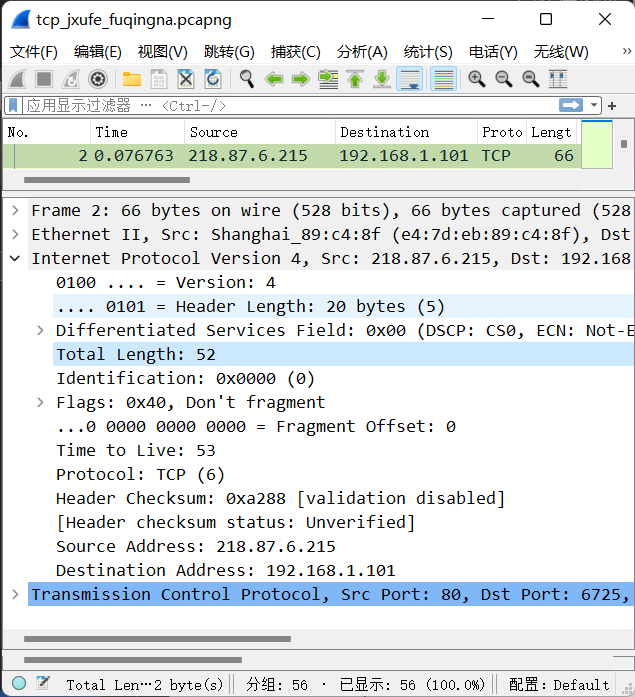
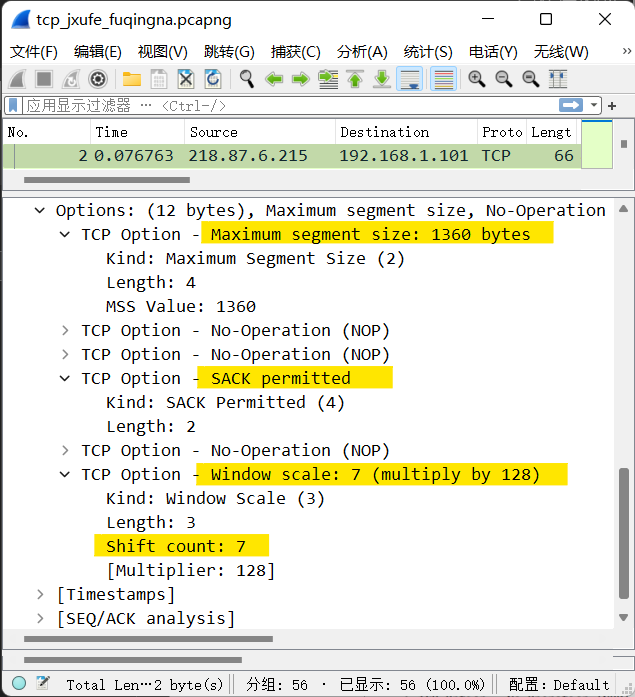
所以，设置MSS为1460bytes可以有效避免分片。

Window scale是调整窗口大小的一个参数。Window scale 种类为3时，Window scale=2 shift count ,即Window scale真正的值是根据选项字段中的shift count计算得来的。如果选项字段中设置了Window scale，意味着真正的窗口值要在原有的窗口值上乘以Window scale。在本TCP中，Window scale= 28 =256 bytes，真正的窗口值为64240bytes。

SACK设置了Sack-Permitted选择性确认允许选项。即在第一次握手中，客户端支持SACK。Sack-Permitted只可以在TCP syn中发送，即TCP建立连接的前两次握手中发送，分別表示通信双方各自是否支持SACK。一旦连接建立，该 TCP 会话就设置相应的SACK 选择性确认选项。

## 1.2 第二次握手

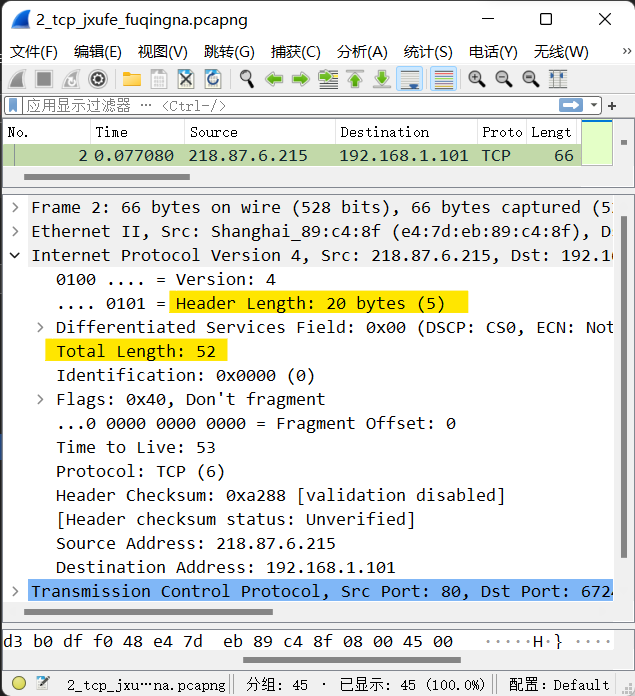
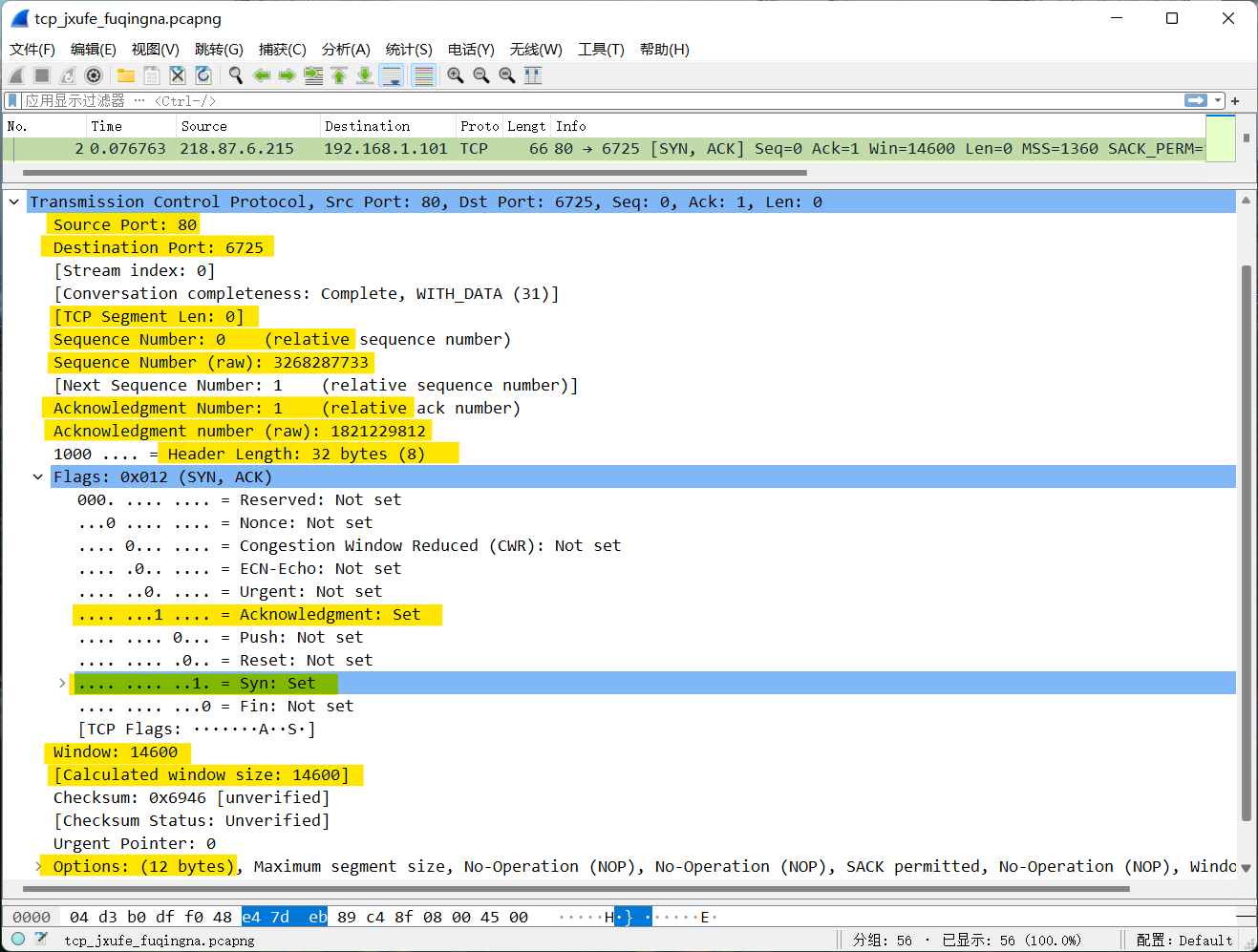




### 1.2.1 端口

源端口：80，说明其上层是HTTP；目的端口：6725，属于用户端口的范围。这里的端口是第一次握手对调的结果。

### 1.2.2 TCP各部分长度



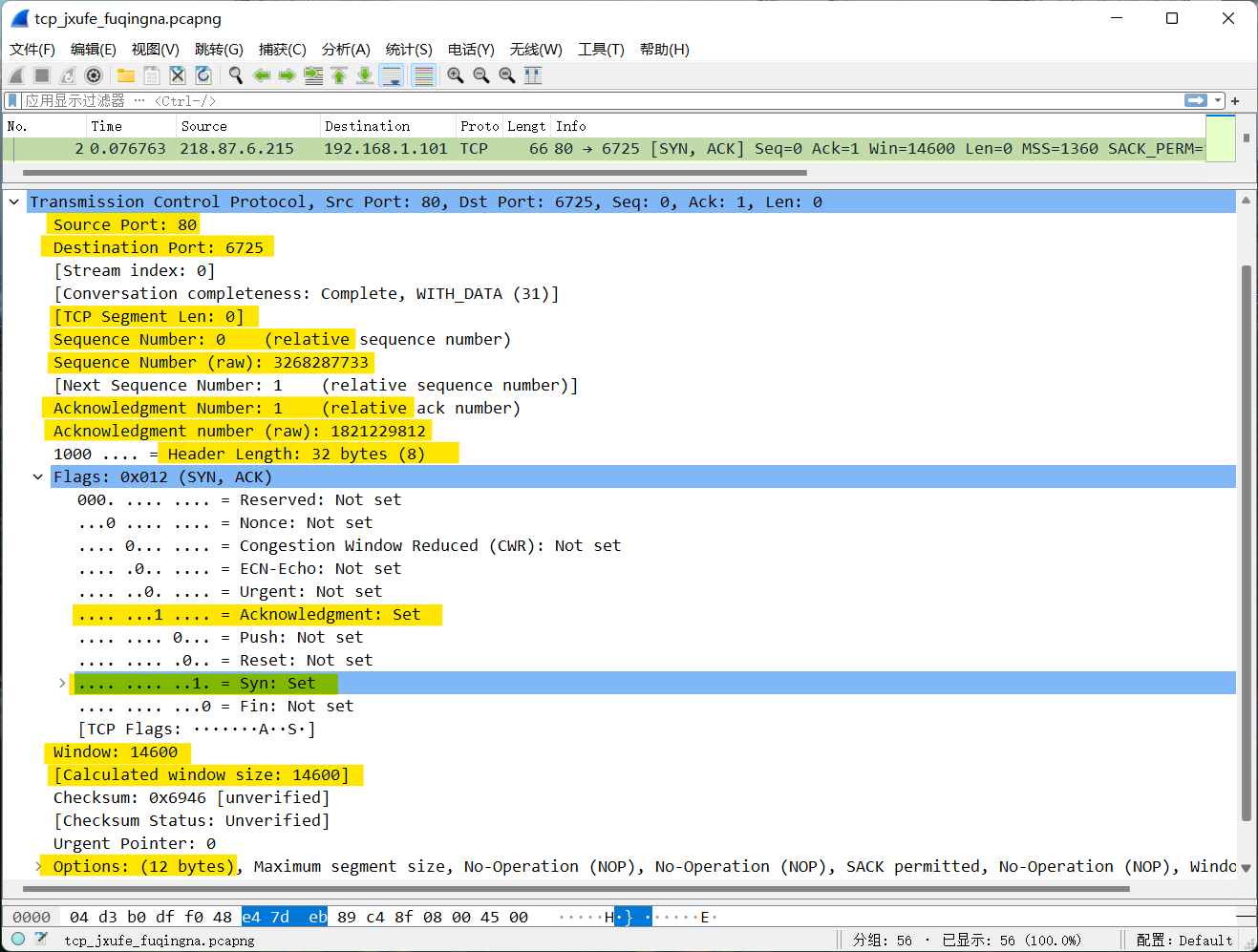
TCP首部中有Header length字段，单位为4bytes，在87号TCP起始包中，Header length为8\*4=32bytes。

TCP首部没有标注TCP的数据长度，但是wires hark提供了TCP Segment Len=0，可知TCP数据部分长度为0，因为这只是一个想要建立连接的TCP包，并没有传送数据。

TCP首部没有标注TCP的总长度，但是通过TCP Segment Len+Header Length可知TCP总长度为32 bytes，也可以通过IP数据报的IP Total length- IP Header length得到TCP总长度为32 bytes。

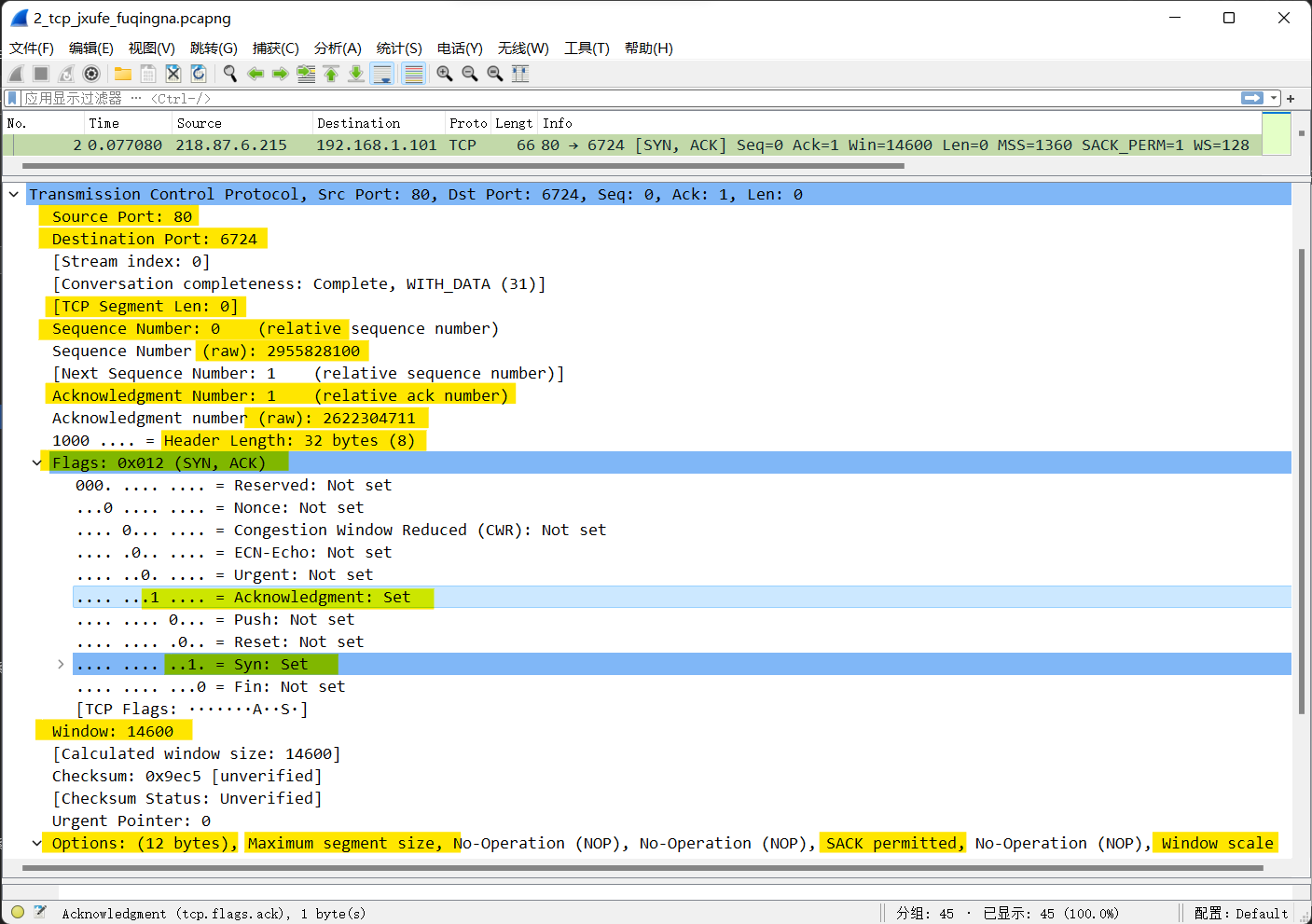
wires hark可以通过IP数据报的IP Total length- IP Header length得到IP数据报的数据部分长度为52-20=32，IP数据报的数据部分长度等于TCP总长度，TCP总长度-TCP Header Length=TCP数据部分长度32-32=0，从而得到TCP Segment Len=0。

### 1.2.3 序列号和确认序号（only relative）



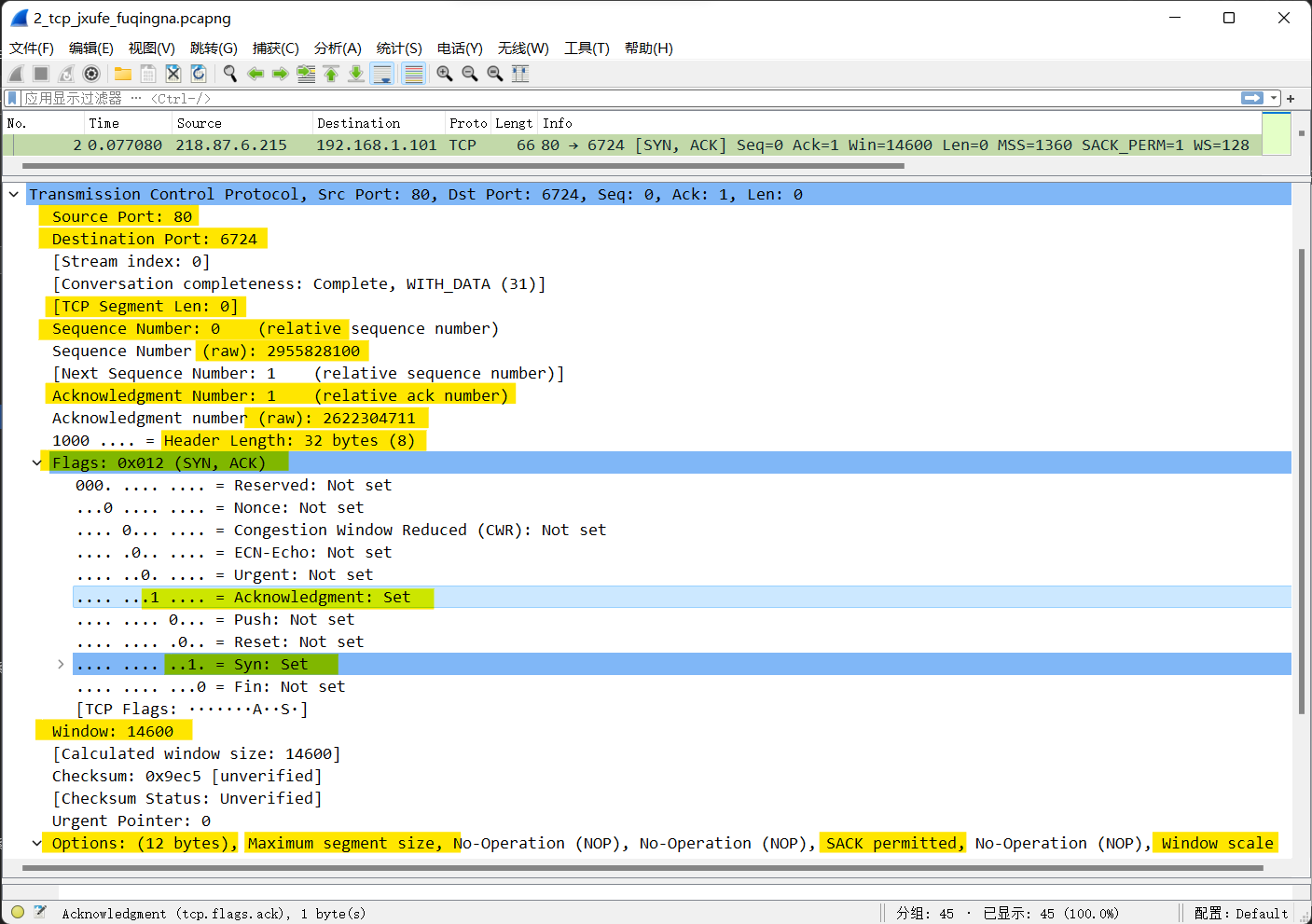
第二次握手的相对序列号为零。相对于服务器发送缓存区内的所有字节，它的排序为0，所以序列号为0。

在服务器的发送缓存中，下一个要发送的字节序列号为1。这说明第二次握手占用了一个序列号。



确认序号为1，因为客户端在第一次握手中的seq=0，虽然第一次握手的TCP数据部分为0，即什么都没发送，但是它仍然占用掉了一个序列号。所以服务器希望下一个从客户端发回来的包的序号应该是1，所以填充1。

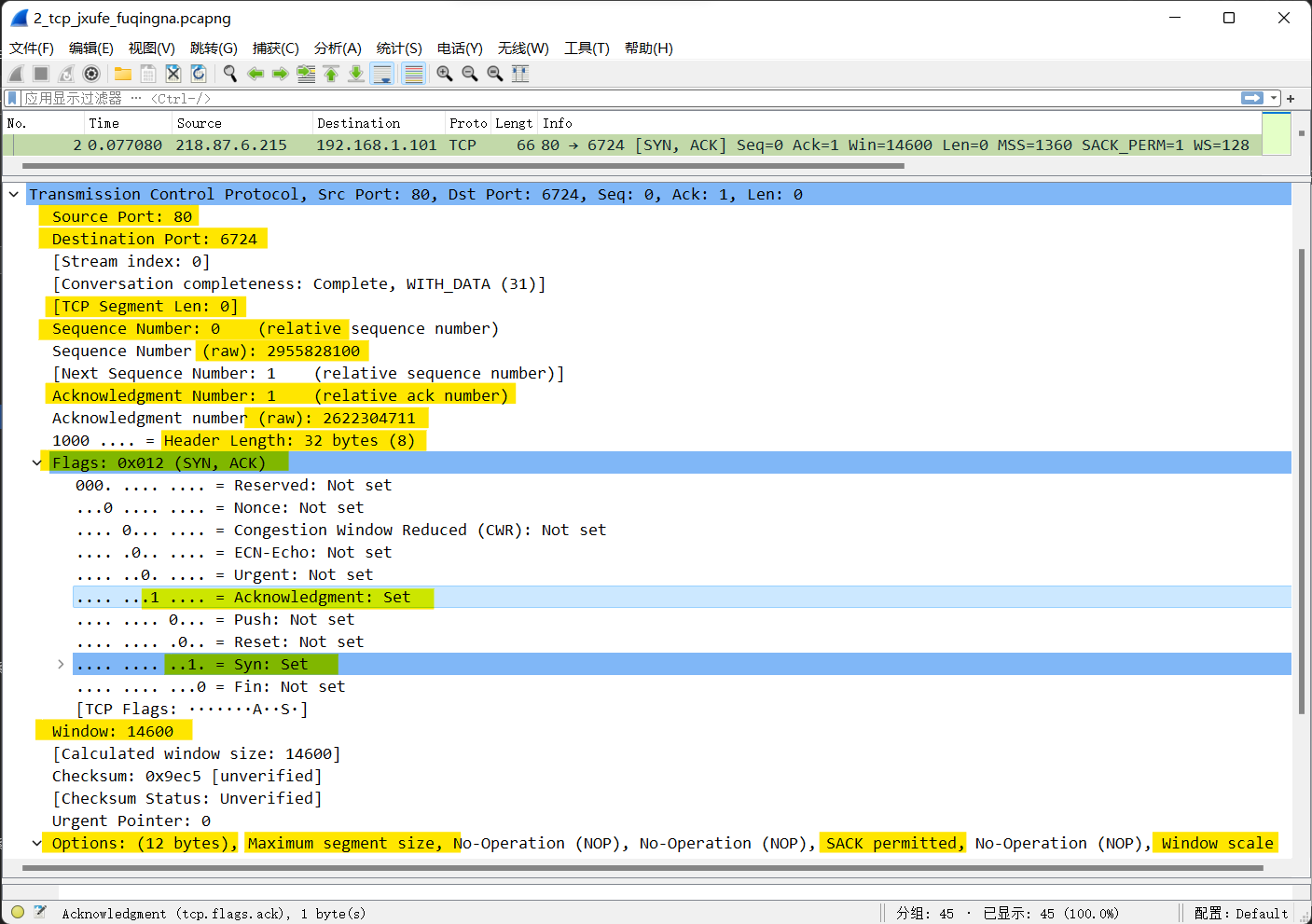
### 1.2.4 flags（seq的占用/seq与ack关系）



ACK=1。代表这是一个确认的TCP包，所以ACK=1。

SYN=1，ACK=1表示同意建立连接。即第二次握手。

### 1.2.5 窗口



窗口大小为14600 bytes，这是wires hark根据选项字段中的Window scale调整后的结果。在本TCP中，Window scale= 27 =128 bytes。

即本设备告知客户端，下一次我（服务器）能接收14600字节的数据。

### 1.2.6 选项

选项字段长12bytes。因为TCP首部包含固定20bytes字段和变长的选项填充字段，由1.2.2可知，本TCP包的Header length为32bytes，即包含20bytes固定长度字段，和12bytes的选项与填充字段。

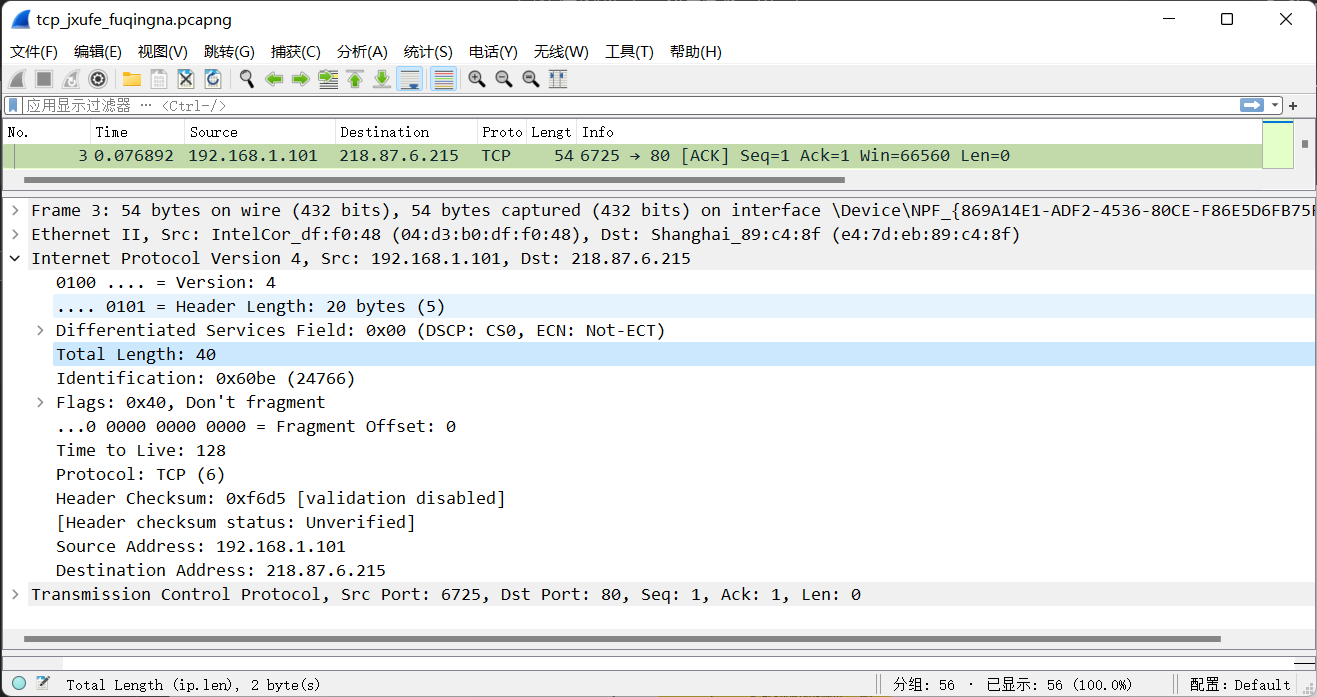
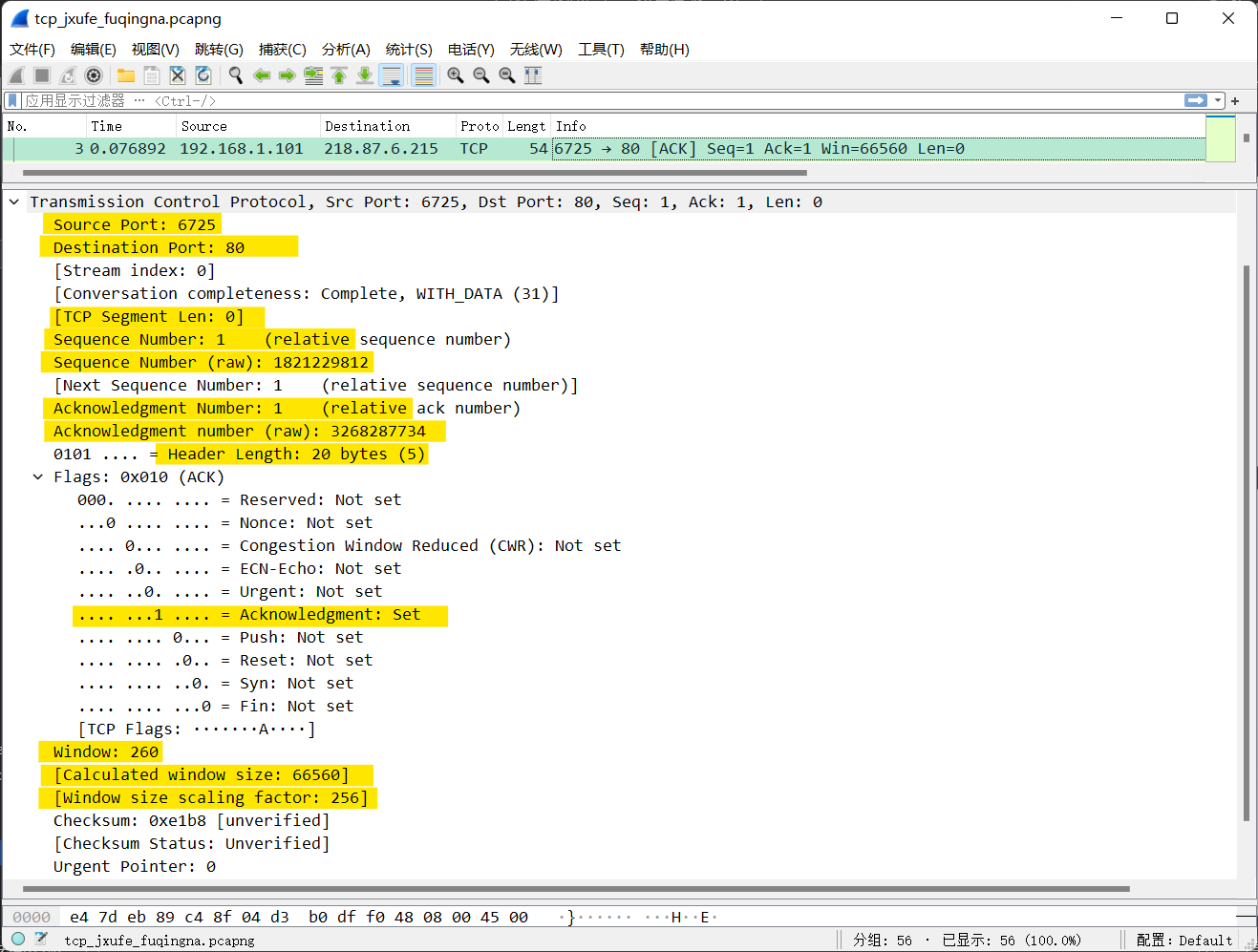


MSS为1360bytes，表示在这段TCP会话中不包含TCP首部的最大数据长度为1360bytes。在第一次握手时MSS为1460bytes，可以有效避免分片。第二次握手将这一数值缩小了。

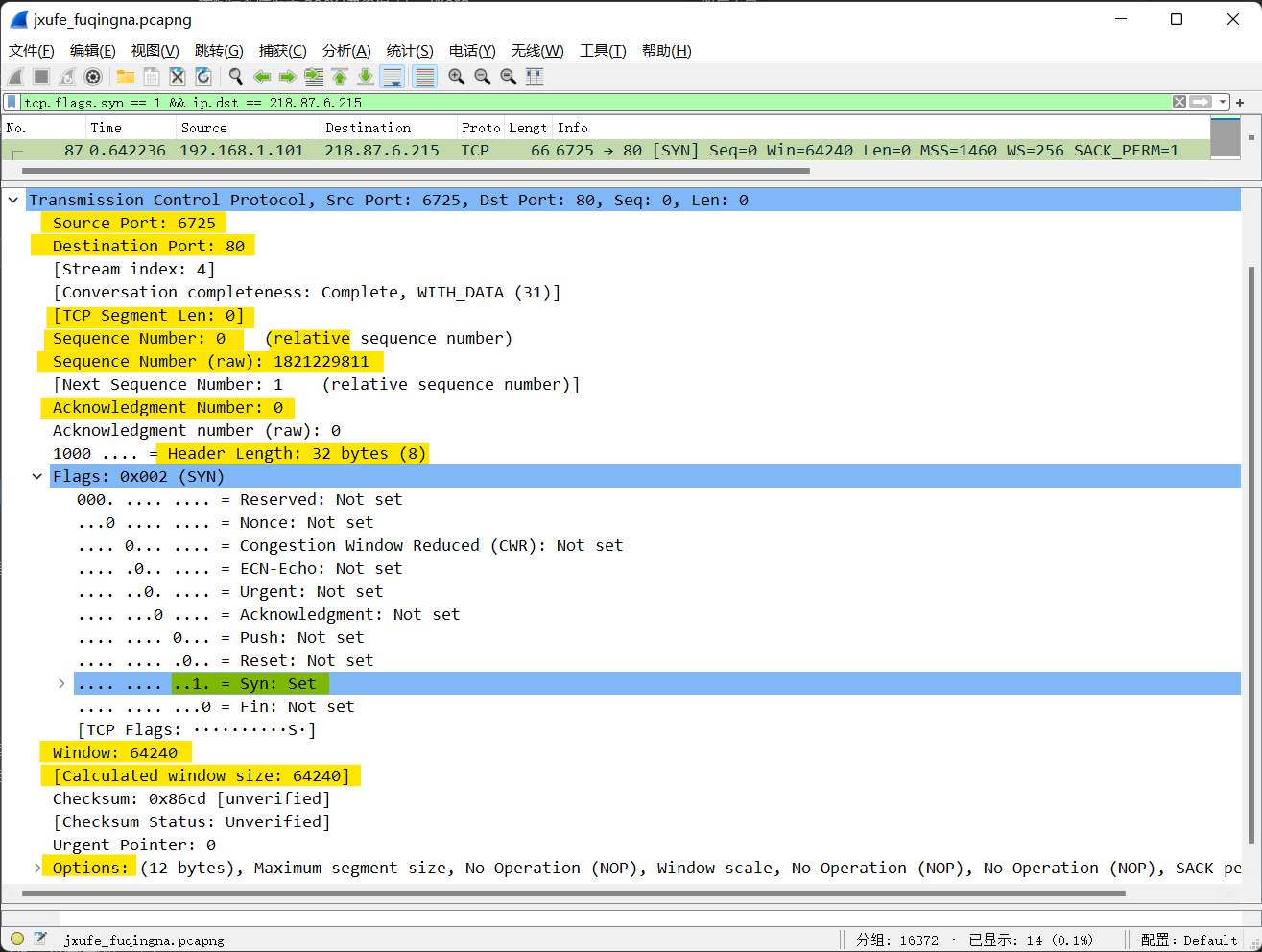
SACK设置了Sack-Permitted选择性确认允许选项。即在第二次握手中，服务器支持SACK。Sack-Permitted只可以在TCP syn中发送，即TCP建立连接的前两次握手中发送，分別表示通信双方各自是否支持SACK。一旦连接建立，该TCP 会话就设置相应的SACK 选择性确认选项。

Window scale是调整窗口大小的一个参数。Window scale 种类为3时，Window scale=2 shift count ,即Window scale真正的值是根据选项字段中的shift count计算得来的。如果选项字段中设置了Window scale，意味着真正的窗口值要在原有的窗口值上乘以Window scale。在本TCP中，Window scale= 27 =128 bytes，真正的窗口值为14600 bytes。

## 1.3 第三次握手

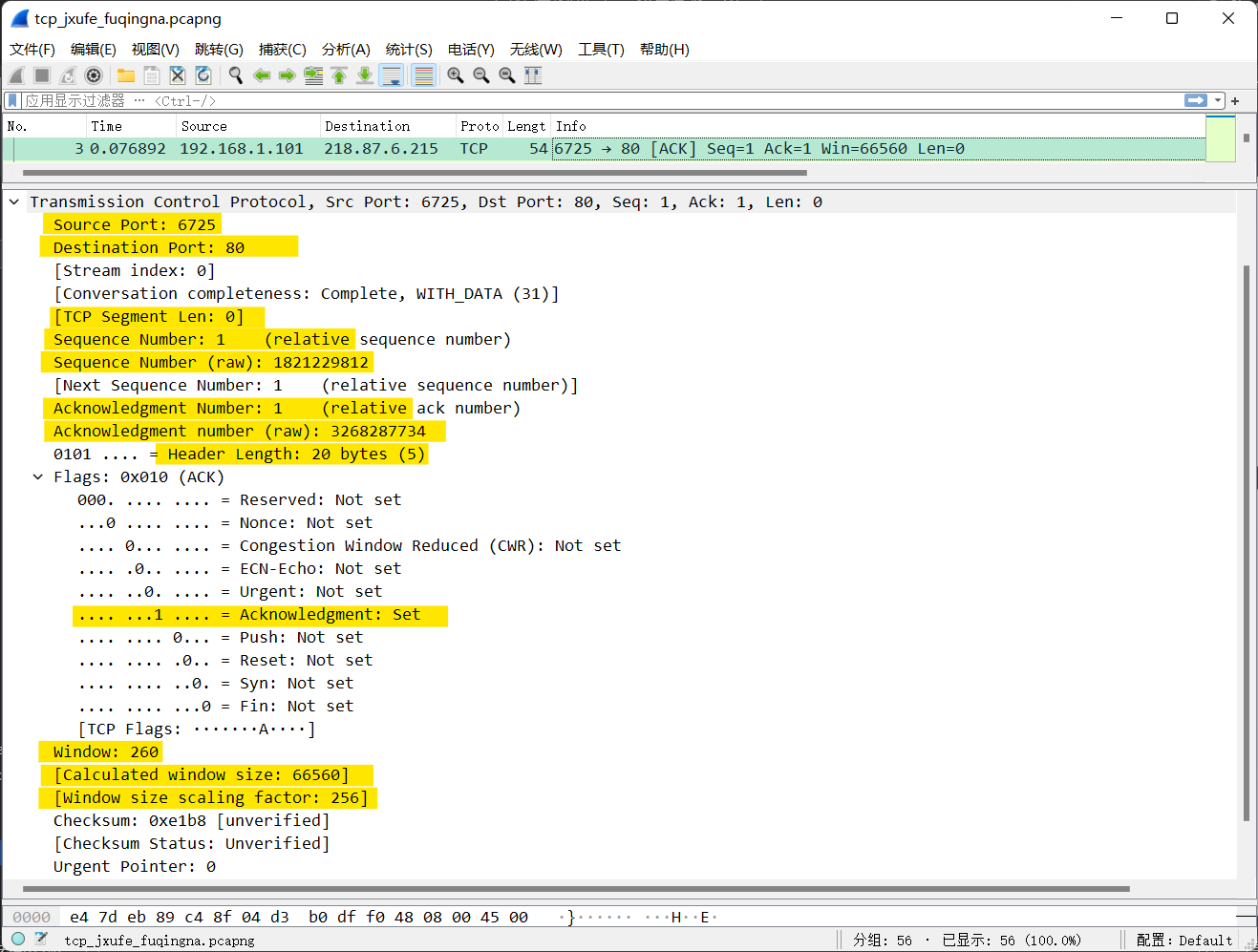


### 1.3.1 端口



源端口：6725，属于用户端口的范围；目的端口：80，说明其上层是HTTP。

### 1.3.2 TCP各部分长度



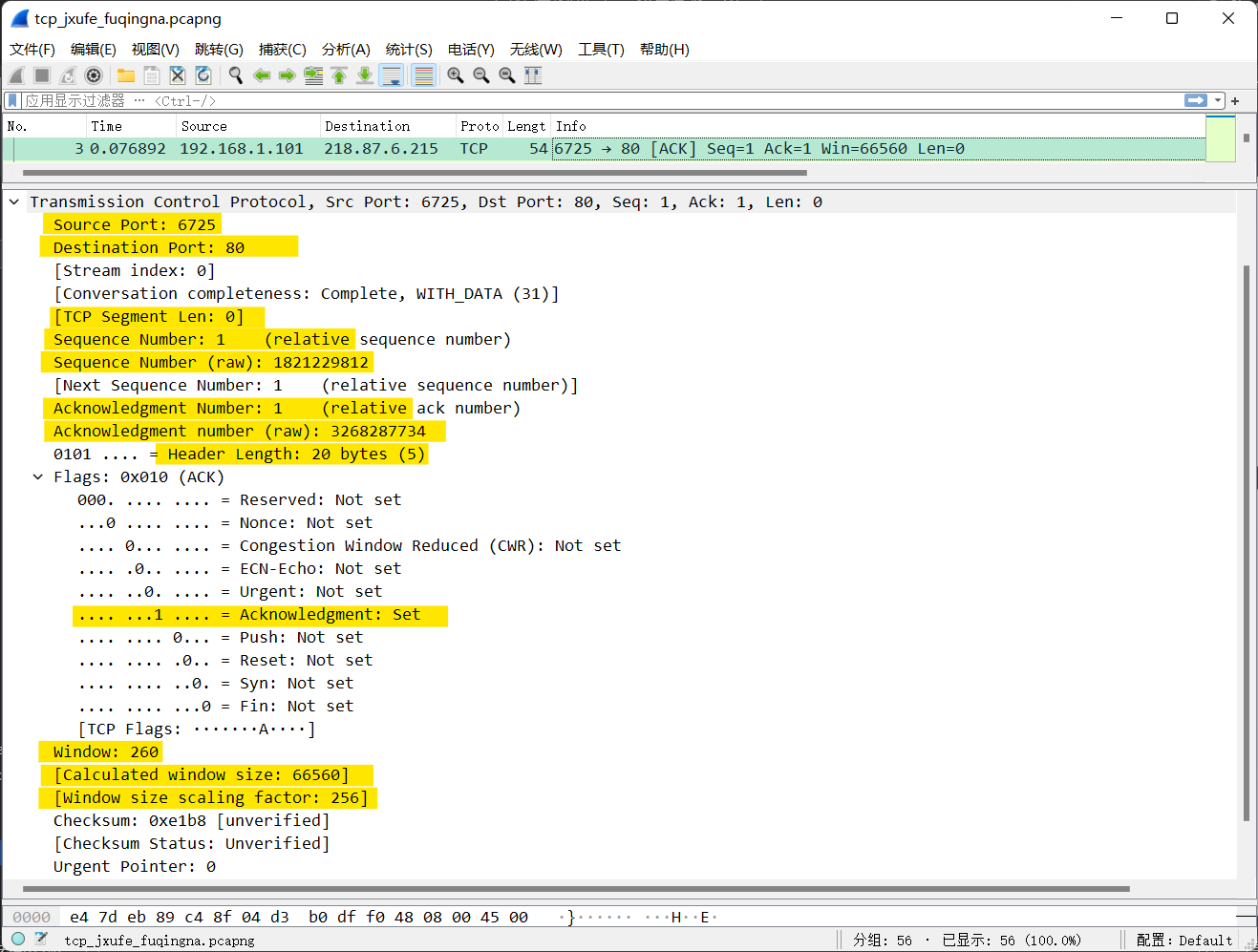
TCP首部中有Header length字段，单位为4bytes，在87号TCP起始包中，Header length为5\*4=20bytes。

TCP首部没有标注TCP的数据长度，但是wires hark提供了TCP Segment Len=0，可知TCP数据部分长度为0，因为这只是一个想要建立连接的TCP包，并没有传送数据。

TCP首部没有标注TCP的总长度，但是通过TCP Segment Len+Header Length可知TCP总长度为20 bytes，也可以通过IP数据报的IP Total length- IP Header length得到TCP总长度为20 bytes。

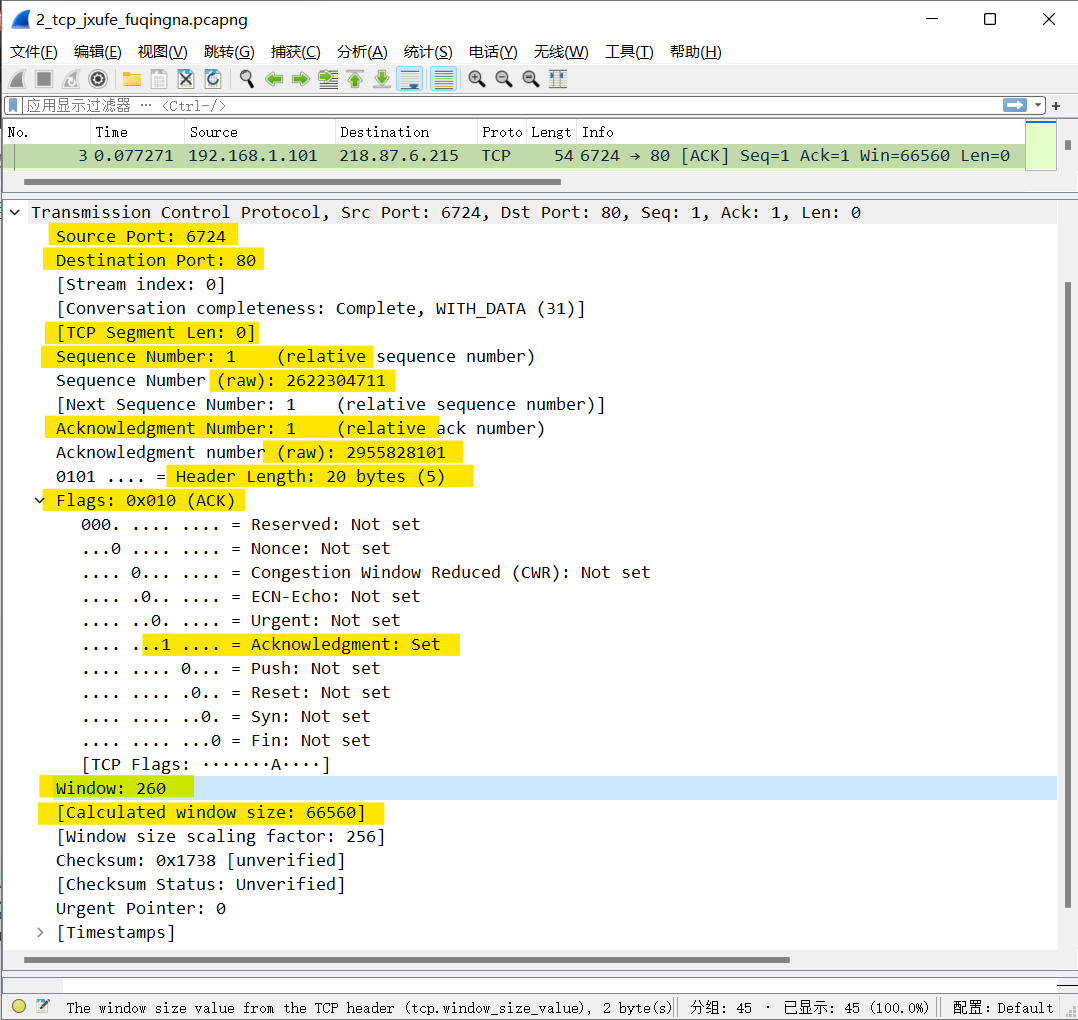
wires hark可以通过IP数据报的IP Total length- IP Header length得到IP数据报的数据部分长度为40-20=20，IP数据报的数据部分长度等于TCP总长度，TCP总长度-TCP Header Length=TCP数据部分长度20-20=0，从而得到TCP Segment Len=0。

### 1.3.3 序列号和确认序号（only relative）



第三次握手的相对序列号为1，即相对于客户端发送缓存区内的所有字节，它的排序为0，所以序列号为0。

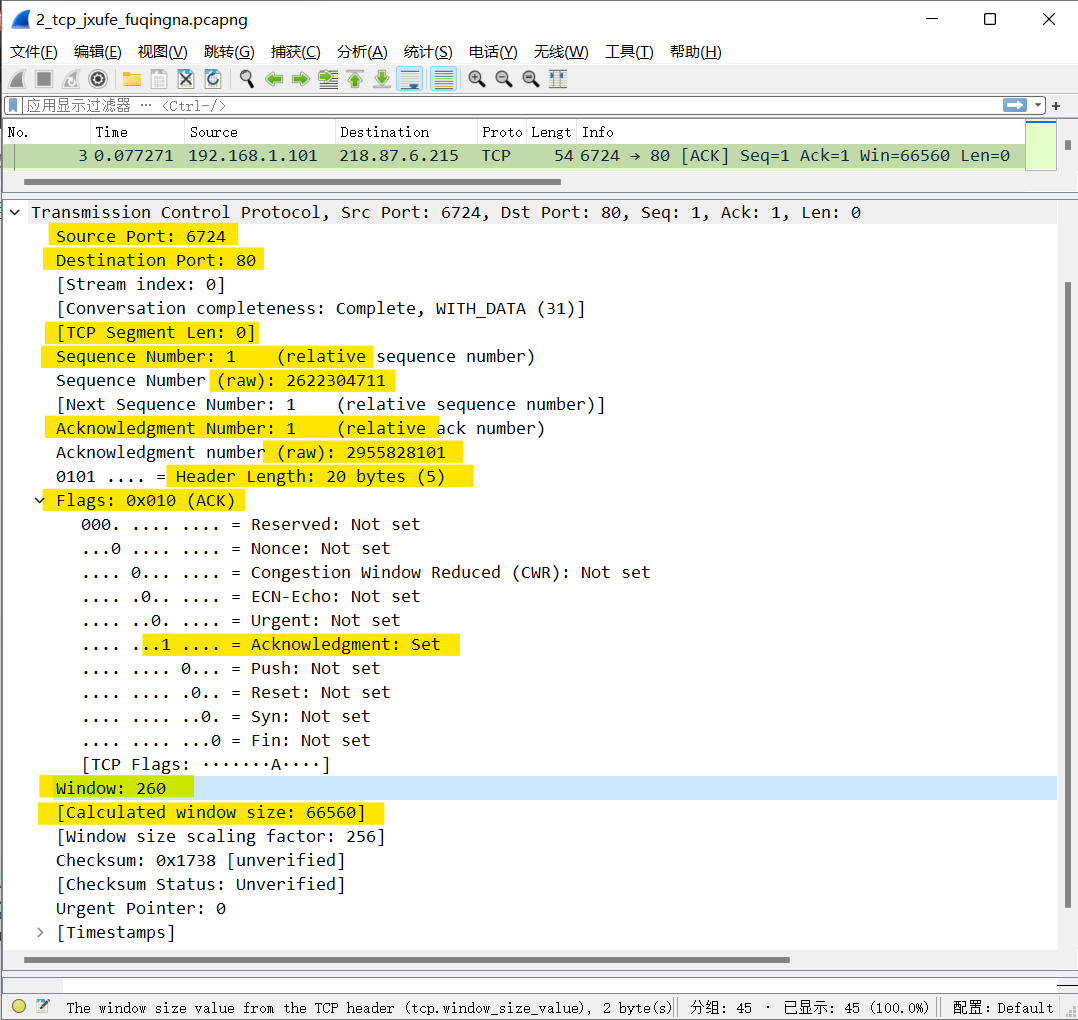
在客户端的发送缓存中，下一个要发送的字节序列号为1。这说明第三次握手是没有占序列号的。



确认序号为1。因为服务器在第二次握手中的seq=0，虽然第二次握手的TCP数据部分为0，即什么都没发送，但是它仍然占用掉了一个序列号。所以客户端希望下一个从服务器发回来的包的序号应该是1，所以填充1。

Get的确认序号也为1。第三次握手是没有占序列号的，所以第三次握手seq=1，get的ack仍然是第三次握手的序列号ack\_get=seq\_3=1。

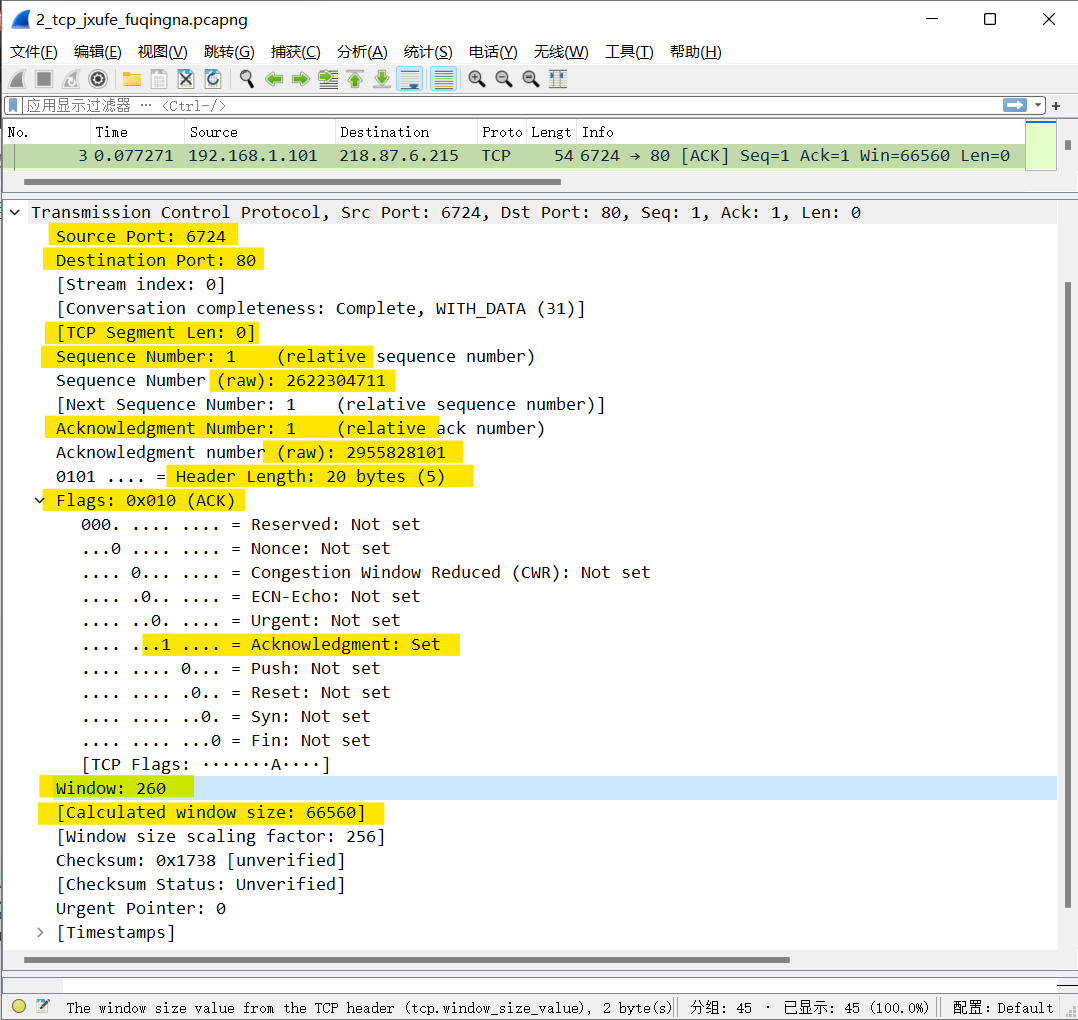
### 1.3.4 flags（seq的占用/seq与ack关系）



ACK=1。代表这是一个确认的TCP包，所以ACK=1。

SYN=0。SYN未设置，因为这已是第三次握手，不用SYN来同步序号了。

### 1.3.5 窗口



窗口大小260，计算后的窗口大小为66560。在第一次握手中，客户端声明了Window scale=28 =256 bytes，wires hark客户端声明的Window scale= 256 乘以window=260，得到66560。

即客户端告知服务器，下一次我（客户端）能接收66560字节的数据。

### 1.3.6 选项

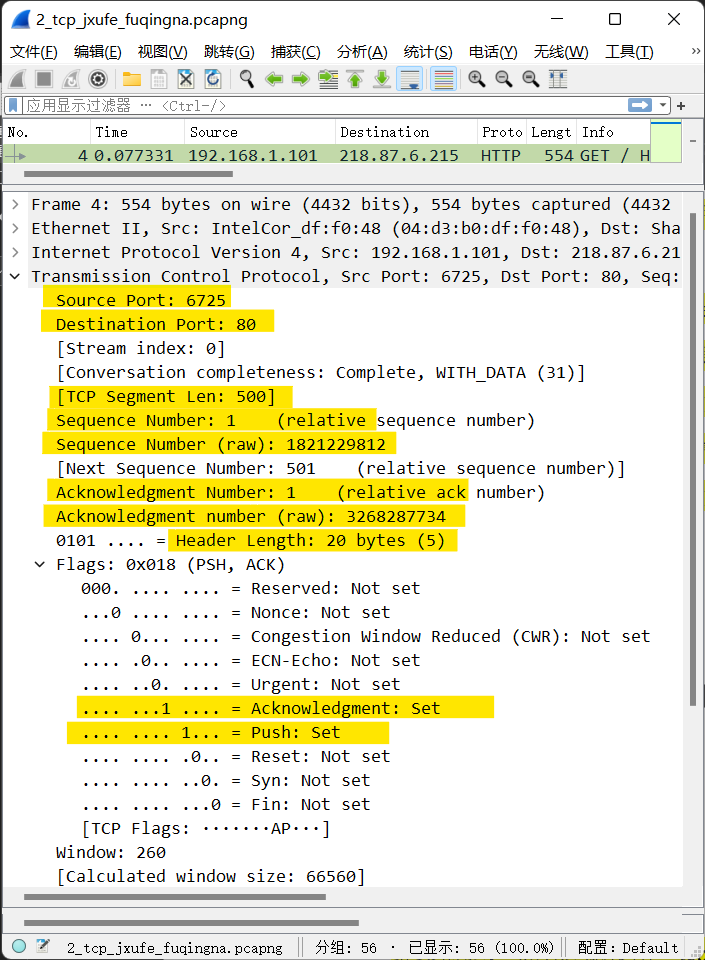
第三次握手没有“选项”字段。因为TCP首部包含固定20bytes字段和变长的选项填充字段，由1.3.2可知，本TCP包的Header length为20bytes，即只包含20bytes固定长度字段，无选项与填充字段，选项与填充字段长度为0。

MSS=1360。MSS在TCP连接建立时，收发双方协商报文段所能承载的最大数据长度。在本实验中，客户端的MSS=1460，服务器的MSS=1360。通讯双方的MSS不相等，就会根据双方提供的MSS值得最小值确定为这次连接的MSS值。即取MSS=1360。

SACK开启。因为第一次和第二次握手中，都设置了Sack-Permitted选择性确认允许选项，表示客户端和服务器都支持选择性确认。

# 实验3

## 3.1 http\_get



源端口：6725，属于用户端口的范围；目的端口：80，说明其上层是HTTP。

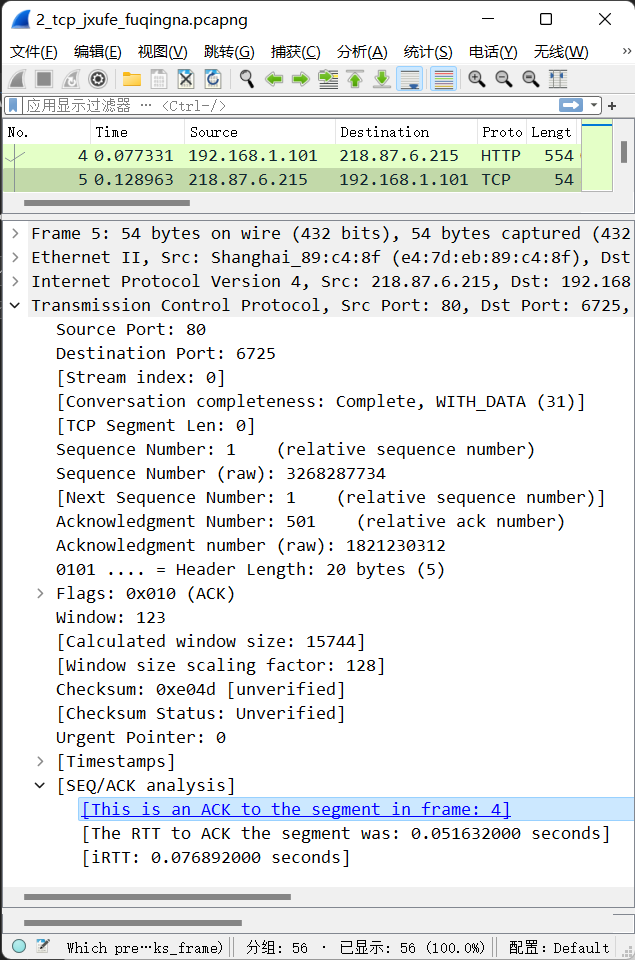
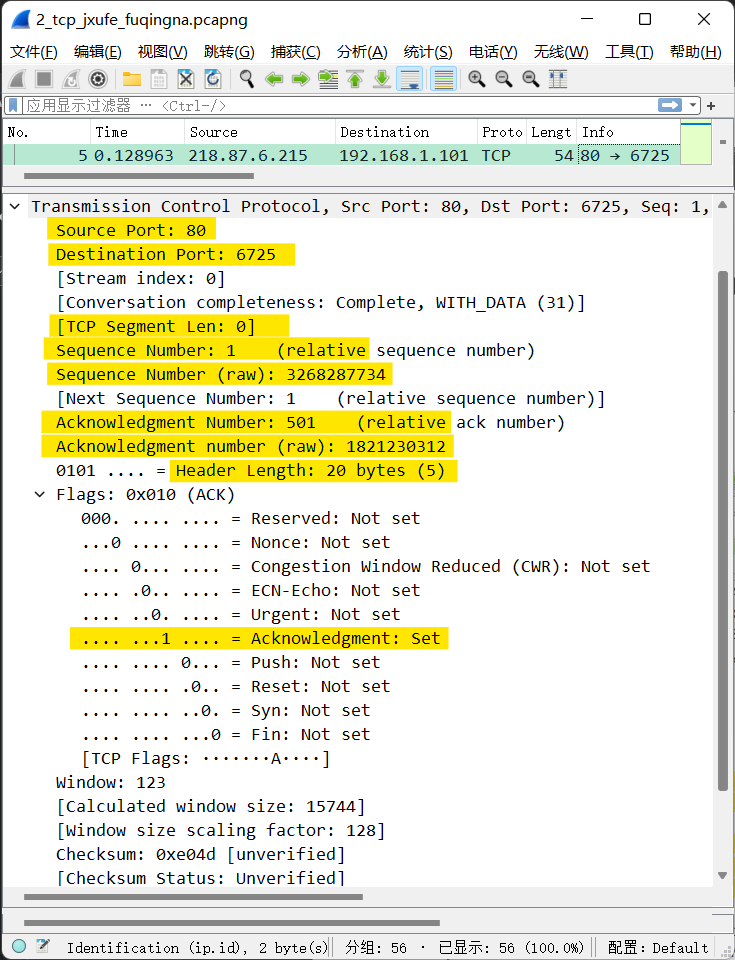
相对序列号为1，即相对于客户端发送缓存区内的所有字节，它的排序为1，所以序列号为1。因为在第三次握手中，客户端向服务器发送的TCP中seq=1，并且1号seq没有被第三次握手占用，所以http的seq仍然是1。

确认序号为1。因为第三次握手的seq=1，而第三次握手是没有占序列号的，所以http\_get的ack仍然是第三次握手的序列号ack\_get=seq\_3=1。

TCP首部长度为20，没有选项字段。

Flag字段中设置了ACK和PUSH。设置了ACK，即ACK=1，此时确认号字段才有效。客户端将PUSH置为1，发送该TCP报文段，服务器收到后发现PUSH置1，会尽快地将次TCP交付给接收应用进程，而不再等到整个缓冲都填满了再向上交互其效果类似于插队。

## 3.2 服务器回复TCP\_1



5号帧是对4号帧的一个确认，所以5号帧的TCP segment Len=0，TCP数据部分为0。5号帧的seq=1，Next seq仍然是1，说明这个确认是不占用序号的。

源端口：80，说明其上层是HTTP；目的端口：6725，属于用户端口的范围。这里的端口是http\_get对调的结果。

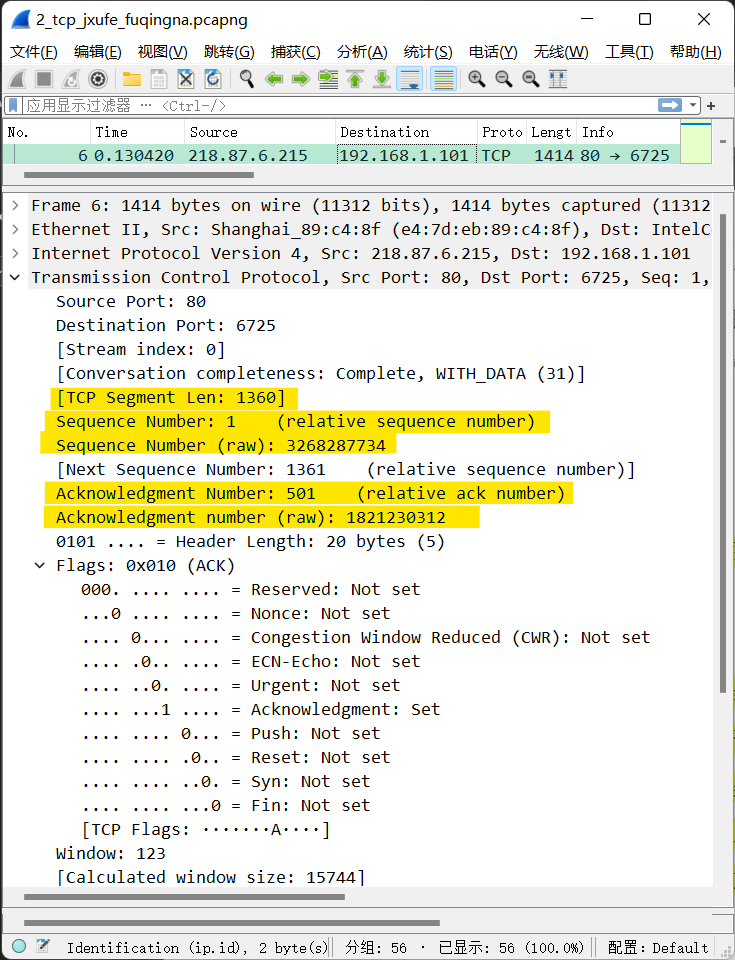
相对序列号为1，即相对于服务器发送缓存区内的所有字节，它的排序为1，所以序列号为1。因为在第二次握手中，服务器向客户端发送的TCP中seq=0，并且0号seq被占用了，所以服务器的回复seq=1。

确认序号为501。因为http\_get的seq=1，数据长度TCP segment Len=500，从1到500正好是500字节，那么http\_get之后服务器发出的确认号应该是500+1=501。

TCP首部长度为20，没有选项字段。

Flag字段中设置了ACK。设置了ACK，即ACK=1，此时确认号字段才有效。

## 3.3 服务器回复TCP\_2



源端口：80，说明其上层是HTTP；目的端口：6725，属于用户端口的范围。这里的端口是http\_get对调的结果。

相对序列号为1，即相对于客户端发送缓存区内的所有字节，它的排序为1，所以序列号为1。因为5号帧的seq=1，Next seq仍然是1，说明5号帧的确认是不占用序号的。所以服务器的回复seq=1。

确认序号为501。因为http\_get的seq=1，数据长度TCP segment Len=500，从1到500正好是500字节，那么http\_get之后服务器发出的确认号应该是500+1=501。

数据长度为1360，因为TCP segment Len=1360。

# 实验4

## 4.1 三次还是四次？

不管是客户端还是服务器，都可以主动发起四次挥手，释放连接。本实验中是服务器主动发起四次挥手。

四次挥手的步骤：

**1. ACK+FIN：**服务器打算关闭连接，发送一个FIN 标志位被置为1的TCP报文，即FIN报文，服务器进入 FIN\_WAIT\_1 状态。。

**2. ACK：**客户端收到FIN报文后，向服务器发送 ACK 应答报文，表示服务器的FIN我已收到。接着客户端进入等待关闭的Closed\_wait状态。服务器收到客户端的ACK应答报文后，进入 FIN\_WAIT\_2 状态。

可能仍然有未发送的数据，数据发送完毕后进入3。

**3. FIN：**等待客户端处理完数据后，也向服务器发送 FIN 报文，之后客户端进入 LAST\_ACK 状态。

**4. ACK：**服务端收到客户端的 FIN 报文后，回一个ACK应答报文，之后进入 TIME\_WAIT 状态。客户端收到了 ACK 应答报文后，就进入了 CLOSE 状态，至此客户端已经完成连接的关闭。服务端在经过2MSL一段时间后，自动进入 CLOSE 状态，至此服务端也完成连接的关闭。

服务器收到客户端的 FIN 报文时，会马上回一个 ACK 应答报文， 但是服务端应用程序可能还有数据要发送，所以并不能马上发送 FIN 报文，而是将发送 FIN 报文的控制权交给服务端应用程序。所以ACK和FIN一般都会分开发送，产生四次握手。

三次挥手的步骤将四次挥手的2和3合并了，即ACK+FIN→ACK+FIN→ACK。当客户端（被动关闭方）在 TCP 挥手过程中，没有数据要发送并且开启了 TCP 延迟确认机制（默认开启），那么第二和第三次挥手就会合并传输，这样就出现了三次挥手。

## 4.2 FIN和ACK设置情况



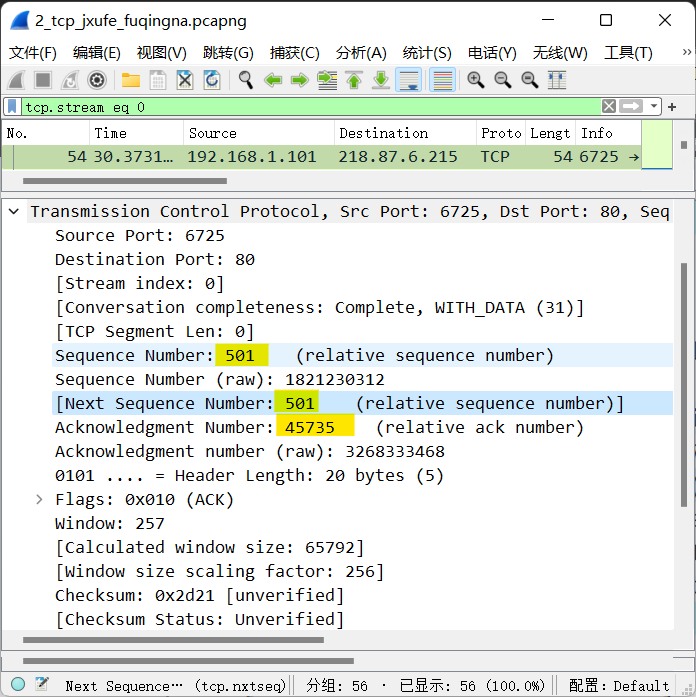
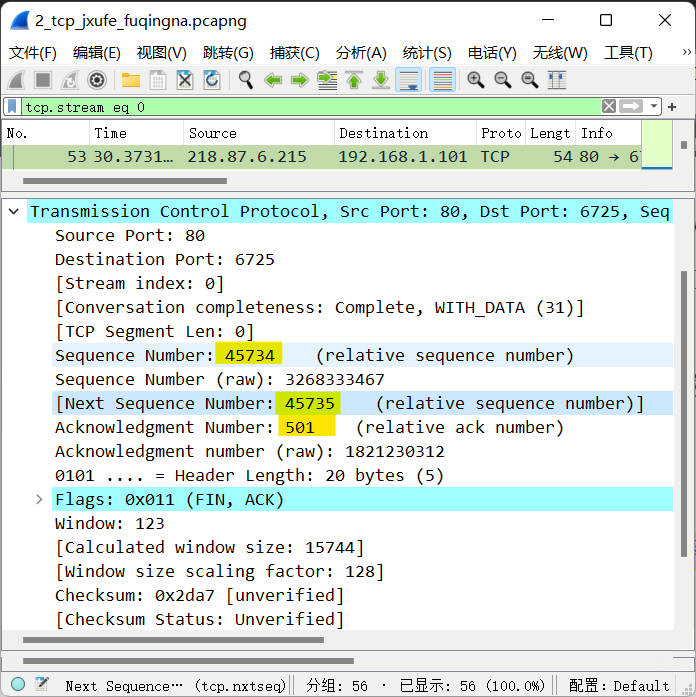
第一次挥手：服务器设置了FIN和ACK，请求断开连接。

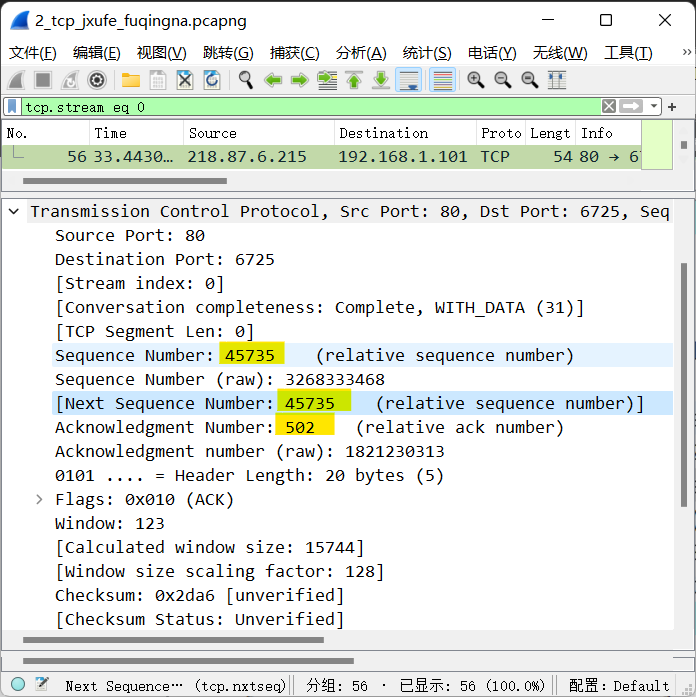
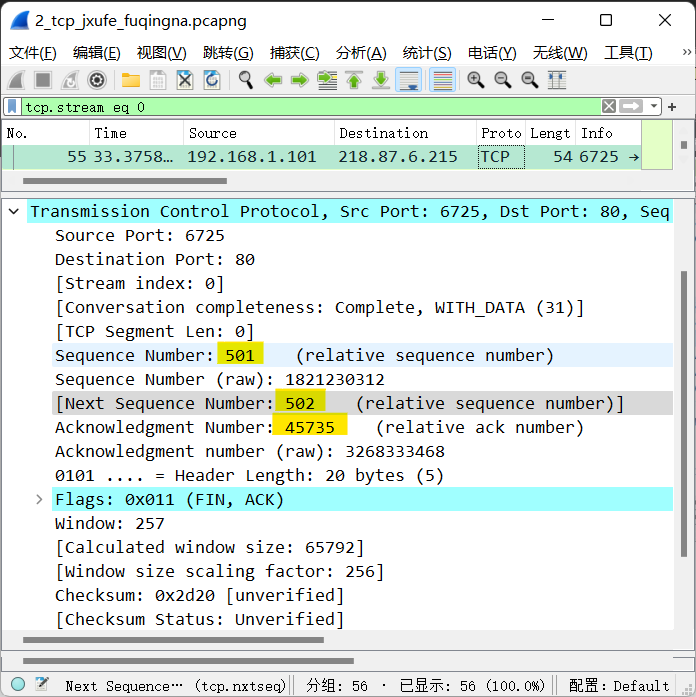
第二次挥手：客户端设置了ACK，确认收到了FIN。

第三次挥手：客户端设置了FIN和ACK，客户端处理数据后，也请求断开连接。

第四次挥手：服务器设置ACK，确认收到FIN。

## 4.3 是否占序号？





观察源和目的端口可知，53、56号帧是服务器（发起方）向客户端（响应方）的方向，54、55号帧是客户端（响应方）向服务器（发起方）的方向。

53号帧（fin）的seq=45734，Next seq=45735，客户端接收到53号帧后给出的确认号也是45735，说明53号帧（fin）占用了序列号；56号帧的seq=45735，Next seq=45735，说明56号帧不占用序列号，因为56号帧发送之后就不会收到和发出TCP包了；

54号帧的seq=501，Next seq=501，说明54号帧不占用序列号，因为它只是对FIN的一个确认；55号帧（分）的seq=501，Next seq=502，服务器接收到55号帧后给出的确认号也是502，说明55号帧占用了序列号。

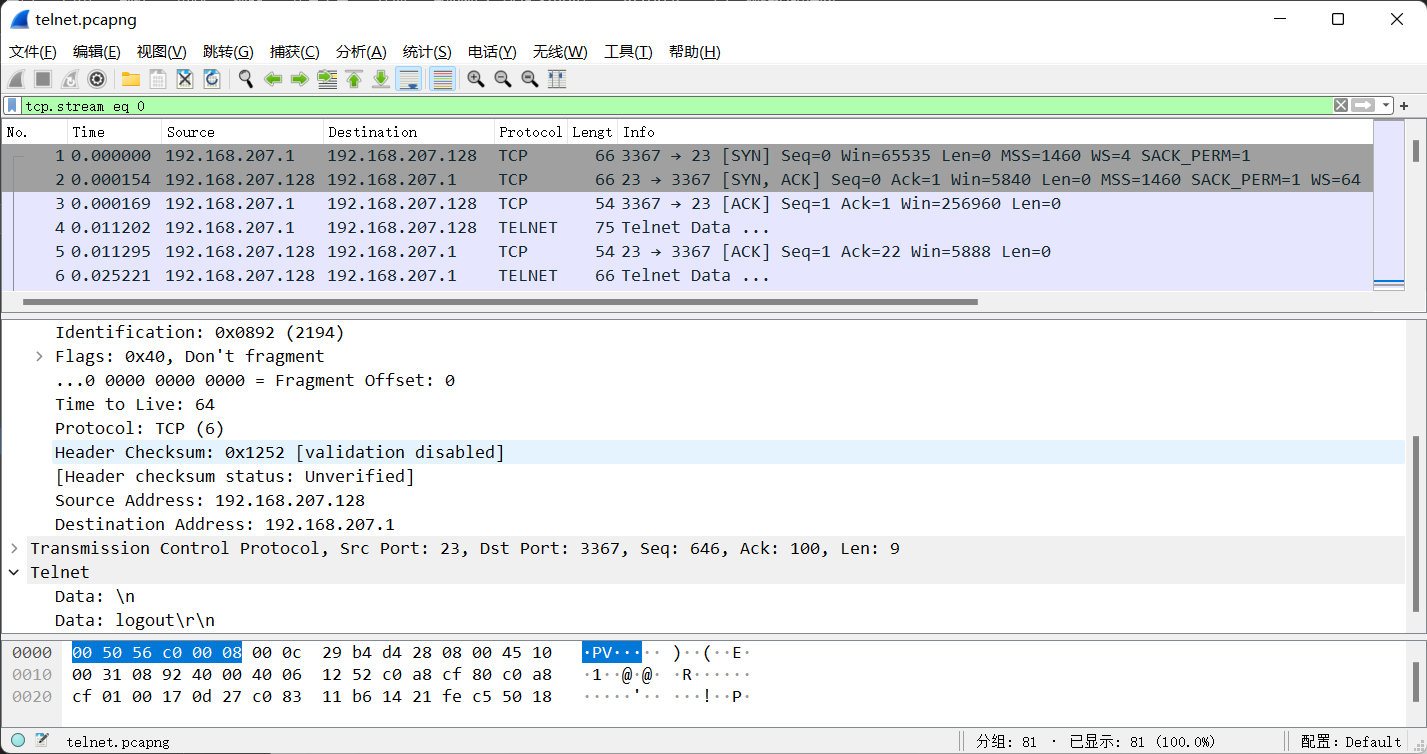
综上，两个方向的fin报文都会占据一个序列号。

# 实验5

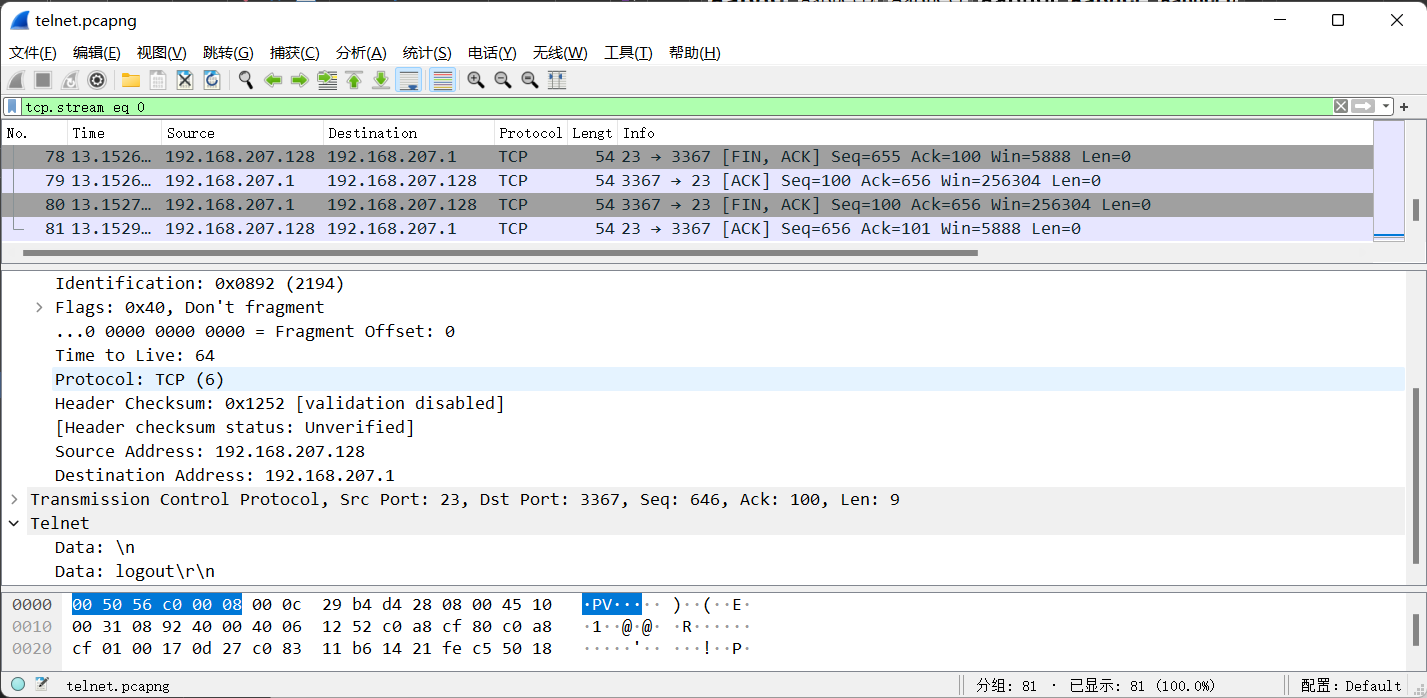
## 5.1 telnet\_off

### 5.1.1 建立和断开连接

三次握手：



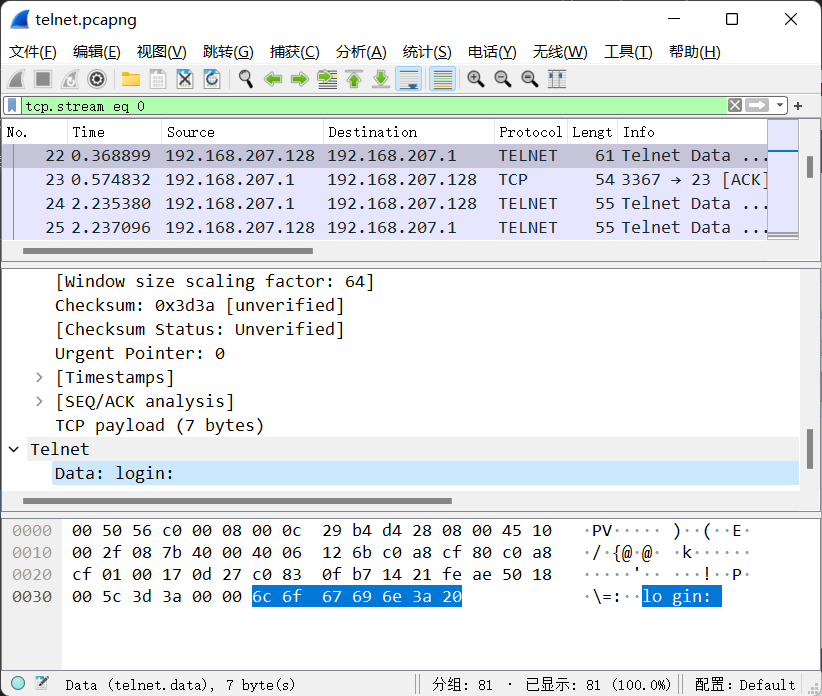
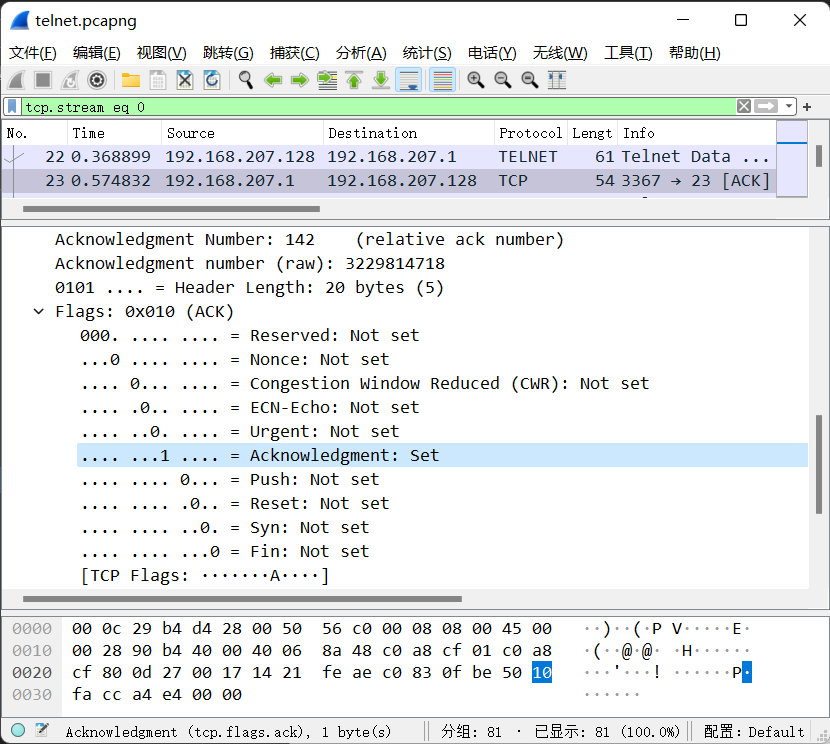
四次挥手：



如上图所示，telnet通过TCP协议建立和断开连接。

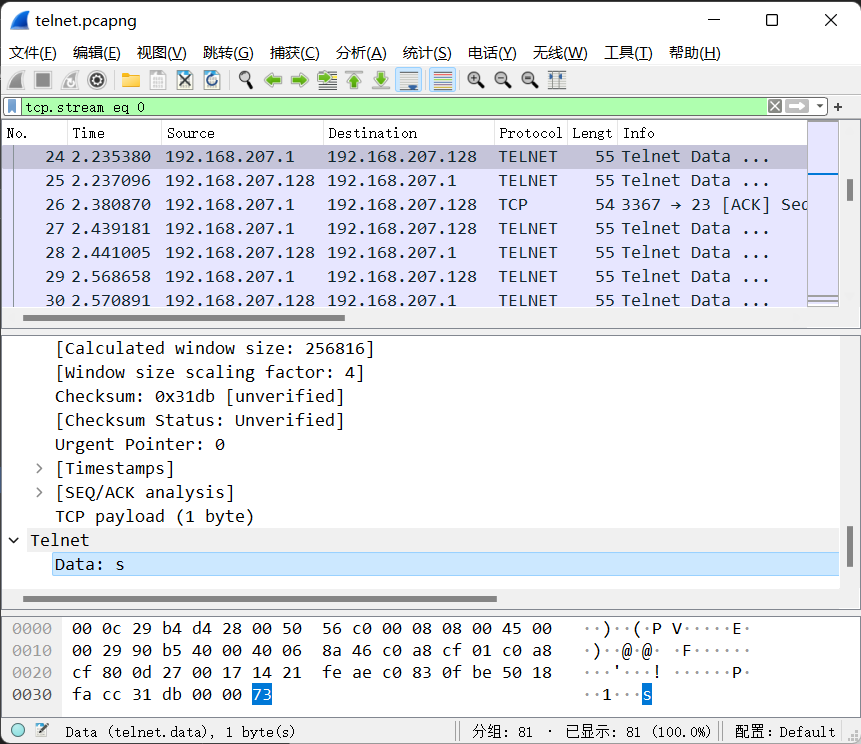
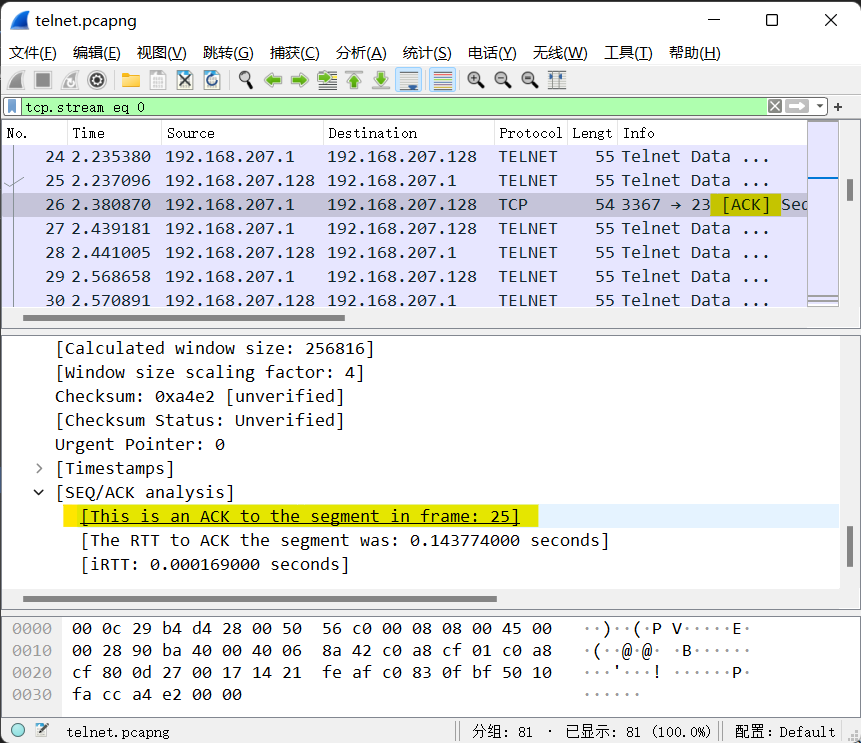
### 5.1.2 身份验证

1. Login :

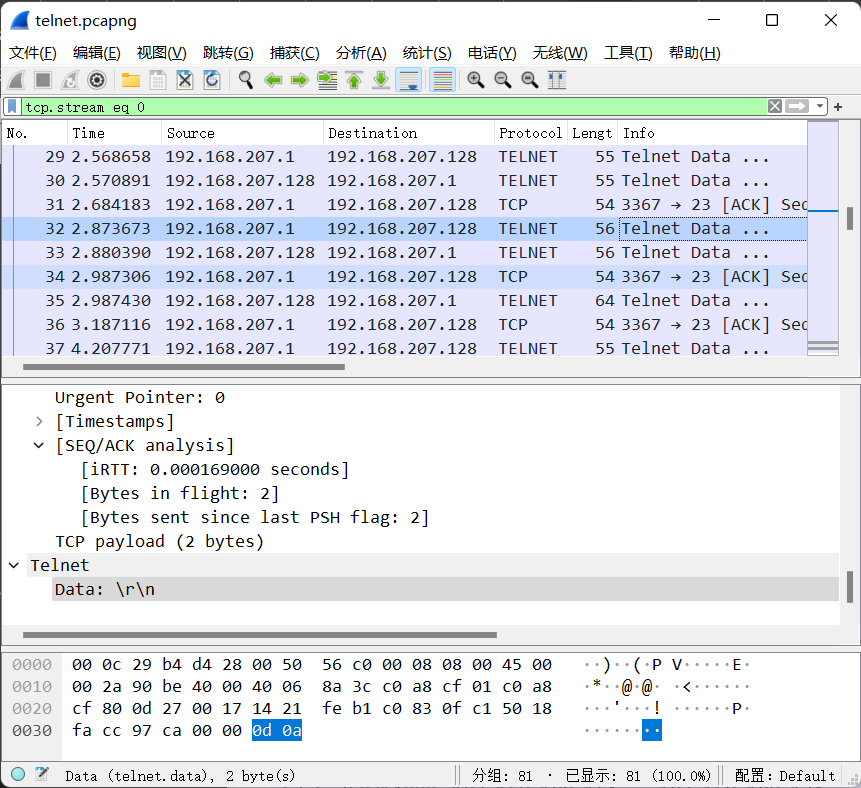
 

服务器发送login： ，等待客户端回复。观察telnet\_login包，发现它的数据部分就是个数为7的“login： ” 包含一个空格，这与TCP的payload的7字节相符。然后客户端发送了一个确认收到的ACK置1的TCP，表示已收到。

2. 输入用户名：

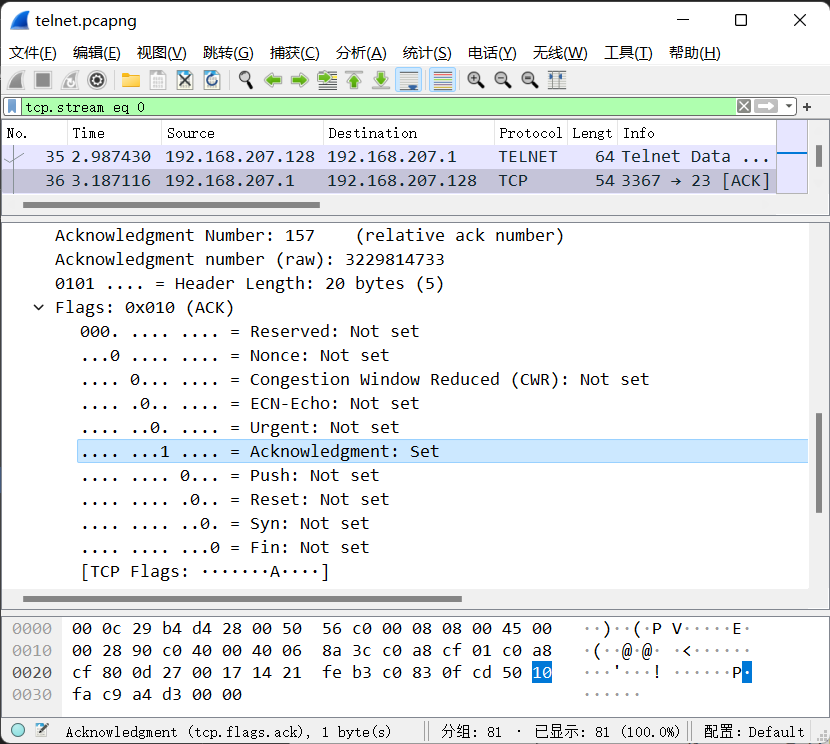
  

输入的字符将挨个发出，主机输入一个字符就发送一个Telnet报文，它的telnet数据部分是输入的那个字符。然后客户端返回一个ACK应答，它的telnet数据部分也是输入的那个字符。之后主机发送一个TCP\_ACK确认收到报文。即三个一组，①Telnet②Telnet③TCP。本实验中的数据部分拼接起来就是stu，即服务器的用户名。这个过程是明文传输的。



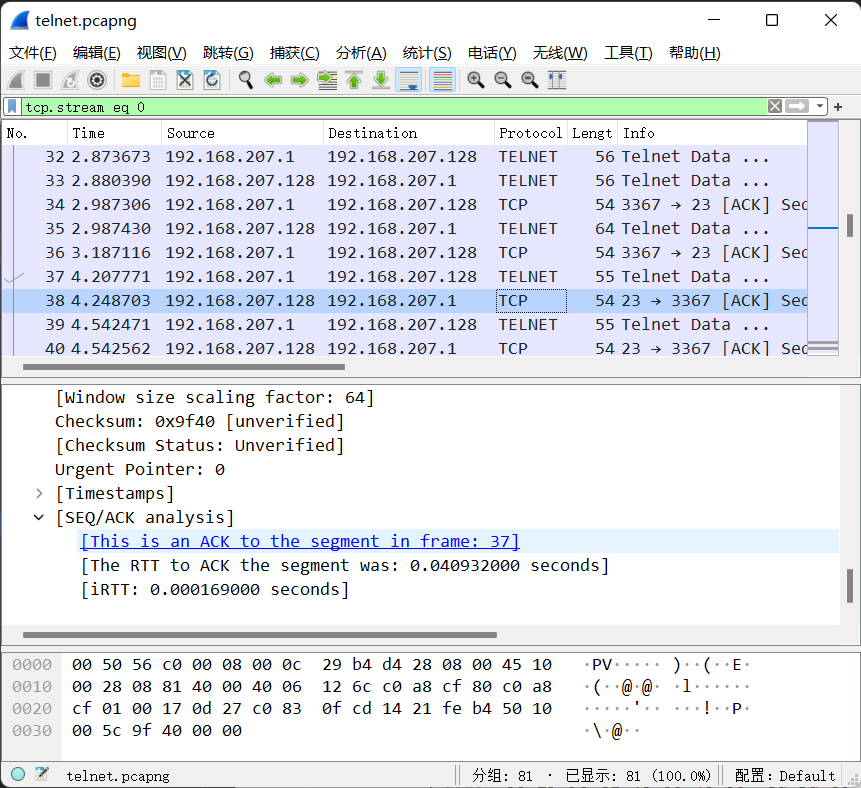
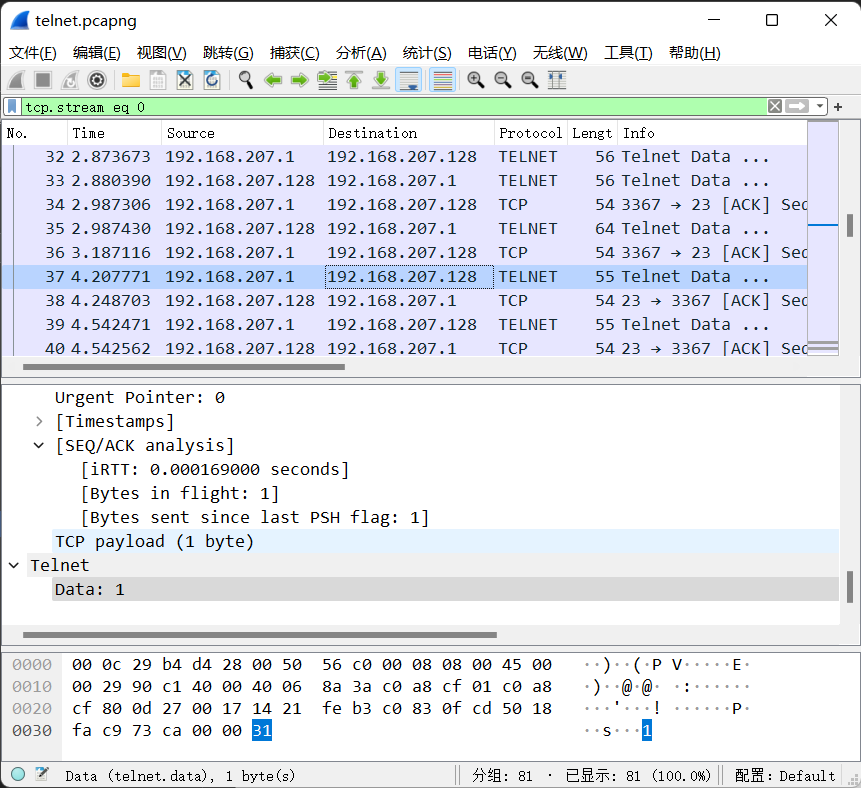
最后在键盘上键入回车，回车符\r\n也要产生三个数据包。回车符发送之后,远端立即回送一个\n\rpassword:要求输入密码。

3. Password:



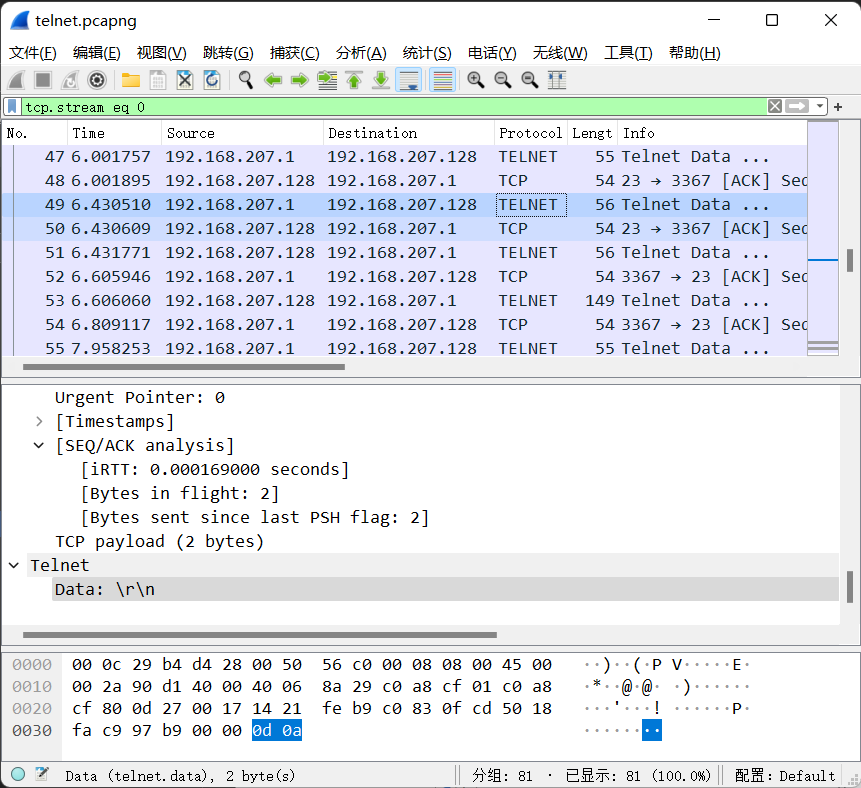
服务器发送password： ，等待客户端回复。观察telnet\_password包，发现它的数据部分就是个数为10的“password： ” 包含一个空格，这与TCP的payload的10字节相符。然后客户端发送了一个确认收到的ACK置1的TCP，表示已收到。

4. 输入密码：



密码输入过程略有不同，一个字符产生两个包，一个是Telnet，一个是TCP，密码也是明文传输。

输入的字符将挨个发出，主机输入一个字符就发送一个Telnet报文，它的telnet数据部分是输入的那个字符。然后主机发送一个TCP\_ACK确认收到报文。即两个一组，①Telnet②TCP。本实验中的数据部分拼接起来就是111111，即远程登入服务器的密码。密码也是明文传输的。



最后在键盘上键入回车，回车符\r\n也要产生两个数据包。回车符发送之后,服务器会验证密码。

### 5.1.3 命令与响应

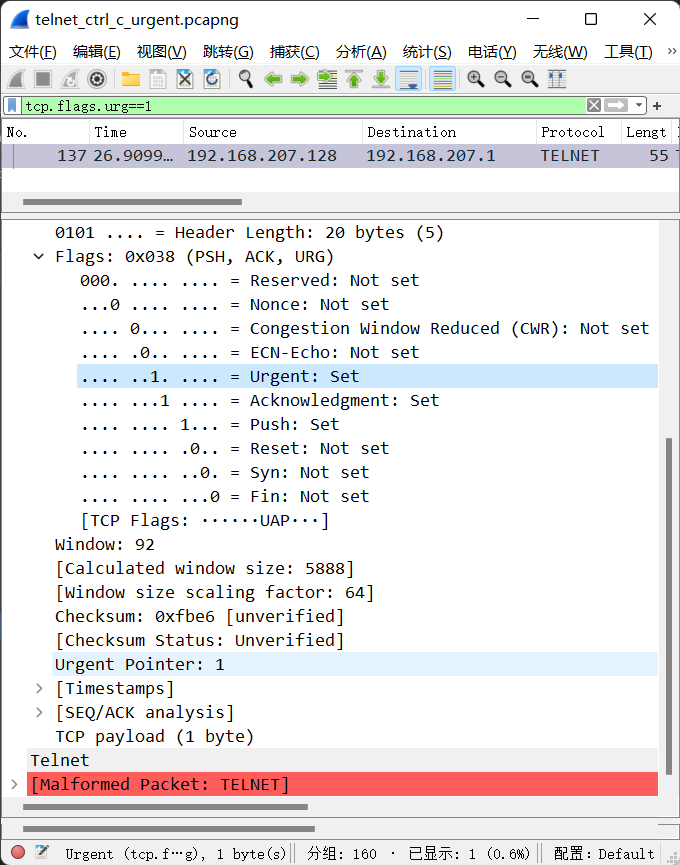
此时telnet已建立，可以自行输入命令，本实验中输入了“ls”、“exit”。仍然是单个字符传输，一个字符产生一个包含两个telnet、一个或零个tcp的组，与身份验证中的字符传输机制相似。

## 5.2 telnet\_ls

### 5.2.1 PSH

Flag中的PSH=1，表示该TCP的发送方应该立即发送TCP，接收方应该将接收缓存中的所有数据立即提交到应用层。而telnet的操作都要求快速得到反馈，这就导致了很多包的TCP首部有PSH标志。

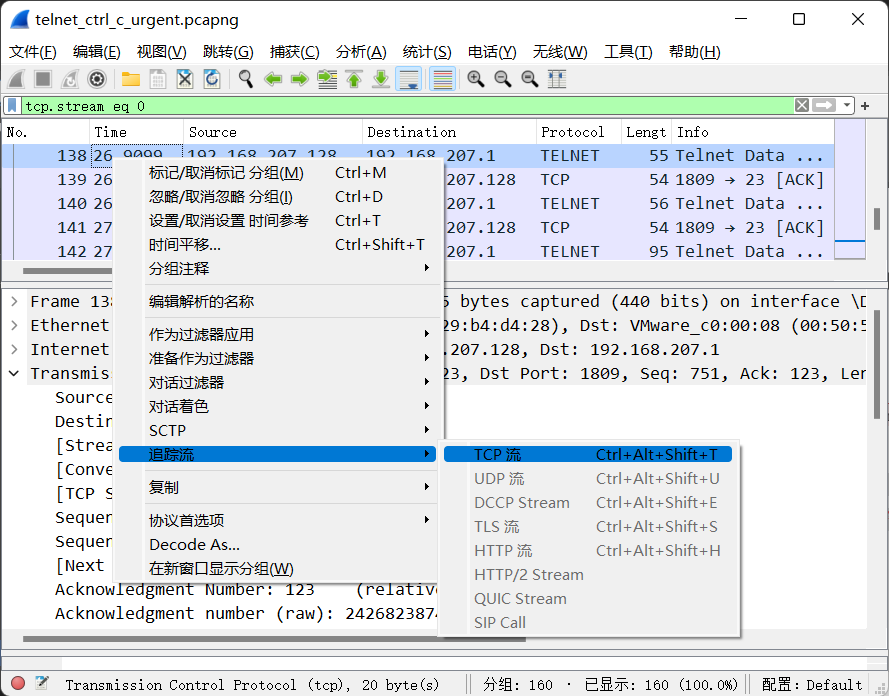
### 5.2.2 urg

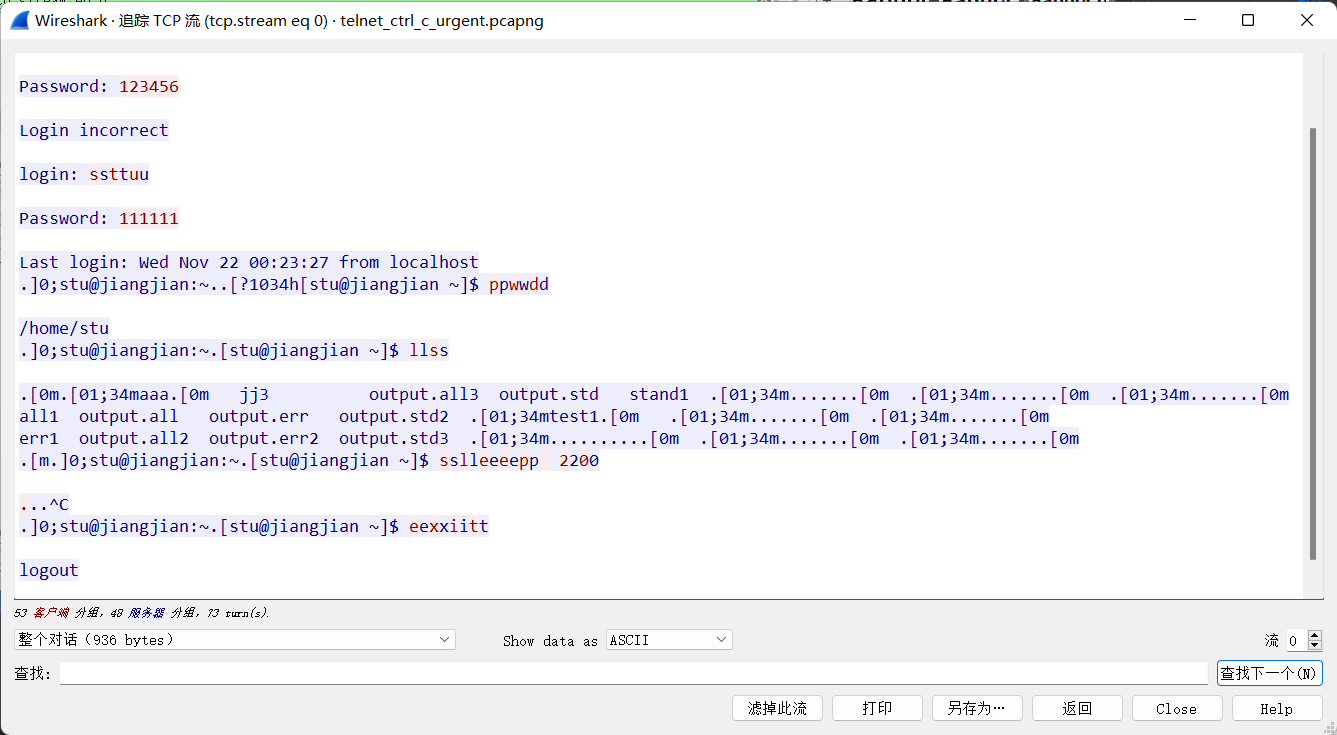


如上图所示，输入“tcp.flags.urg == 1”，滤出了“紧急”报文段。

CTRL+C强行中断当前程序的执行，只有服务器向客户端回复方向的TCP报文采用urg紧急比特。理想状态下，强行中断应该引起客户端和服务器都采用紧急比特。但是实际中的网络传输，往往是服务器向客户端发送的数据量很大，会被流量控制机制限制，若客户端接收窗口为0，服务器就无法给客户端继续发数据，所以必须采用urg确保即时性；而客户端发往服务器的数据量比较小，不会被流量控制机制限制，数据不会在发送缓存和接收缓存中等待过久，所以无需urg。

### 5.2.3 追踪TCP流



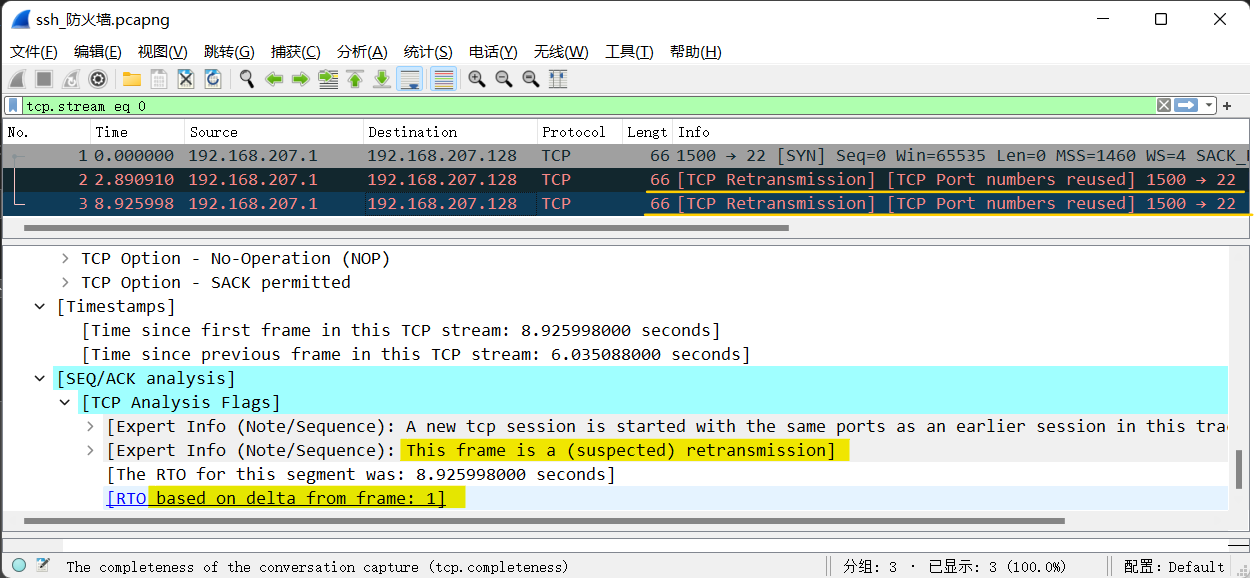


通过wires shark的follow tcp steam工具我们可以直接看到用户名、密码和用户操作。这说明telnet的明文传输很不安全，可以很简单地截获账户密码，无法保护用户的隐私信息和资料。进一步的，telnet还有没有强力认证过程，只是验证连接者的帐户和密码、没有完整性检查，传送的数据没有办法知道是否完整是否被篡改过、传送的数据都没有加密等缺点。

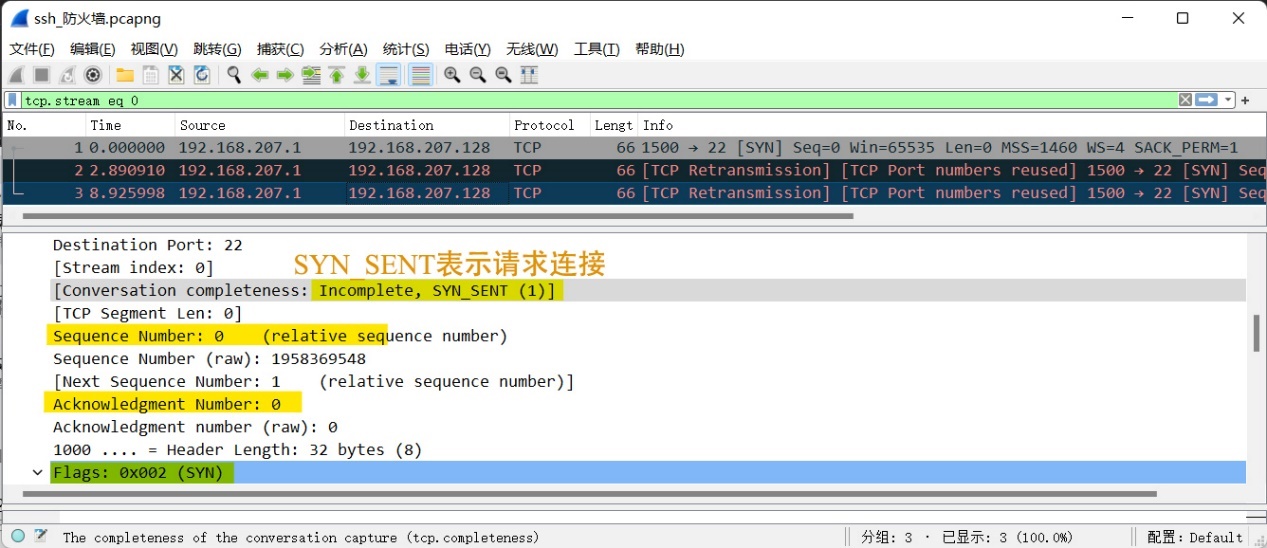
# 实验6

## 6.1 sshd\_off

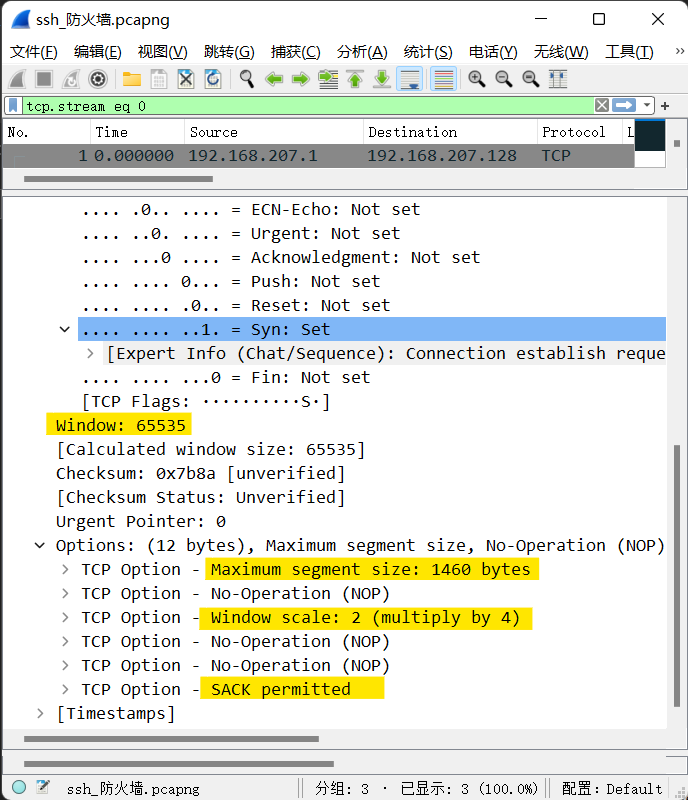
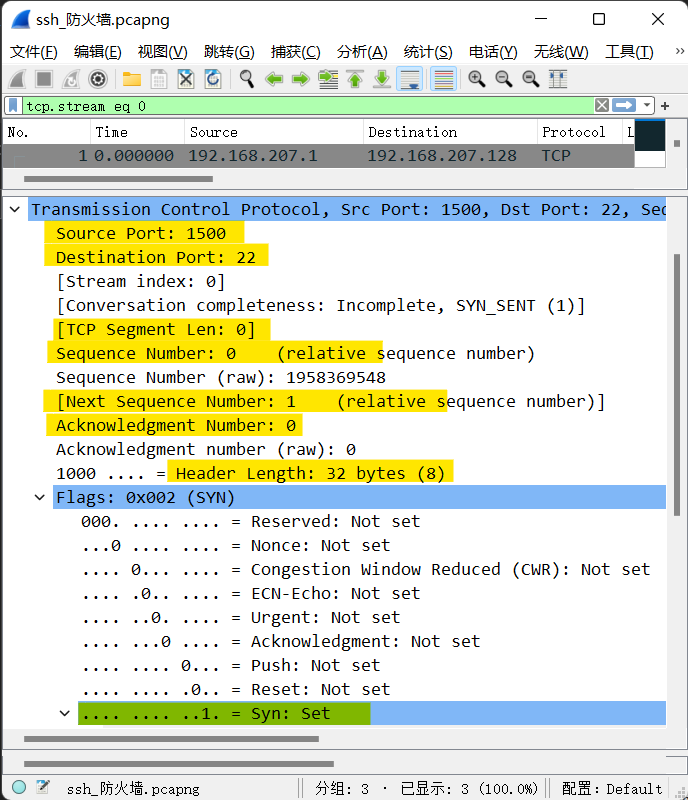
### 6.1.1 总体特点



总体上看，这三个包的内容大致相同，因为第二三个包都是对第一个包的重传，由于防火墙的sshd服务关闭了，所以这三个包都发送失败了。



这三个包的都是TCP请求建立连接/第一次握手的包，因为它们都是从用户端口传输到TCP\_22（ssh）端口，且seq=0，ack=0。因为sshd服务使用ssh协议，是应用层协议，它的传输层协议是TCP，所以在ssh登录之前必须要建立TCP连接。

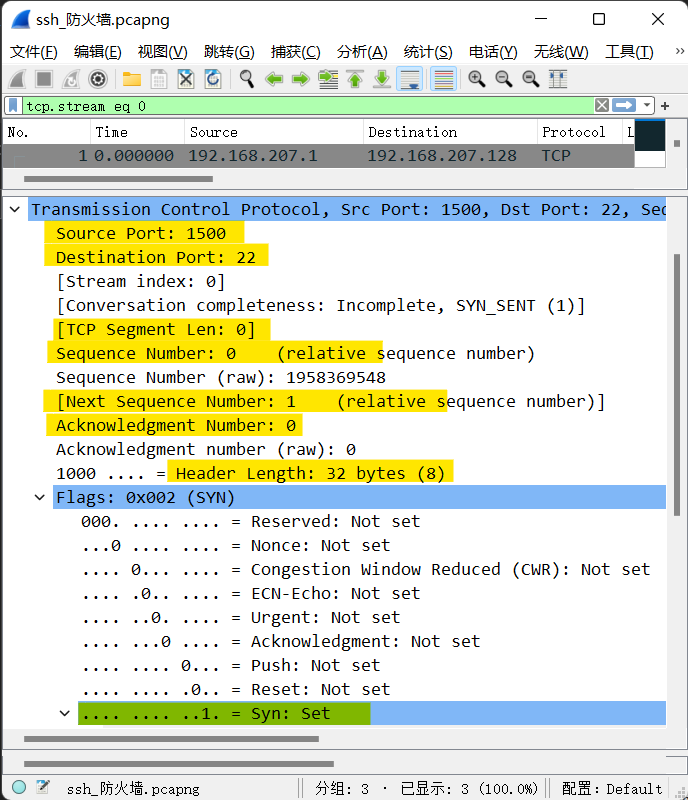


由于是后两个是第一个的重传包，所以选取1号帧分析包细节。

### 6.1.2 端口

源端口：1500，属于用户端口的范围；目的端口：22，说明采用了ssh协议。

### 6.1.3 TCP各部分长度

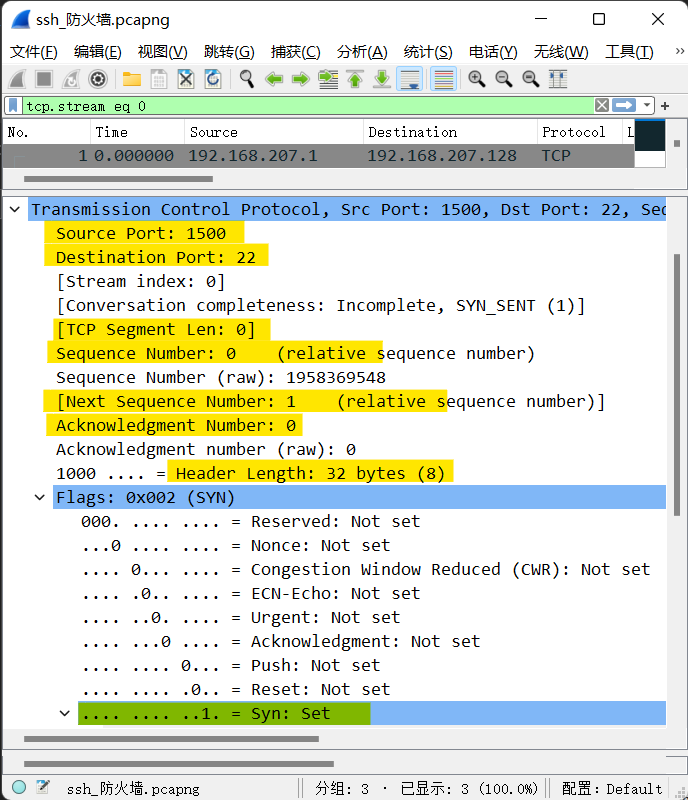


TCP首部中有Header length字段，单位为4bytes，在1号TCP包中，Header length为8\*4=32bytes。

TCP首部没有标注TCP的数据长度，但是wires hark提供了TCP Segment Len=0，可知TCP数据部分长度为0，因为这只是一个想要建立连接的TCP包，并没有传送数据。

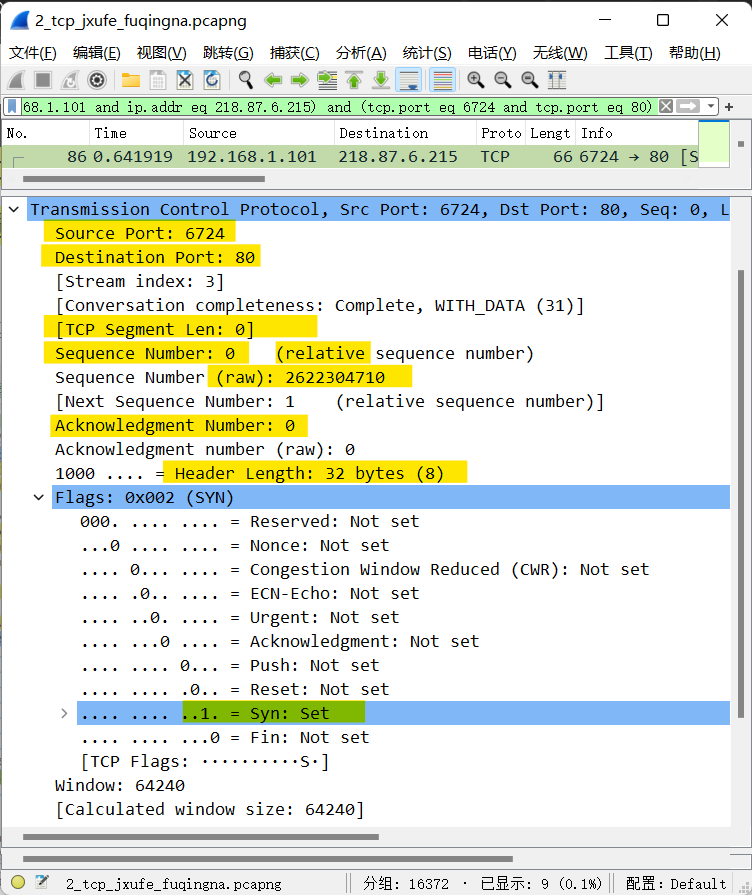
TCP首部没有标注TCP的总长度，但是通过TCP Segment Len+Header Length可知TCP总长度为32 bytes，也可以通过IP数据报的IP Total length- IP Header length得到TCP总长度为32 bytes。

### 6.1.4 序列号和确认序号



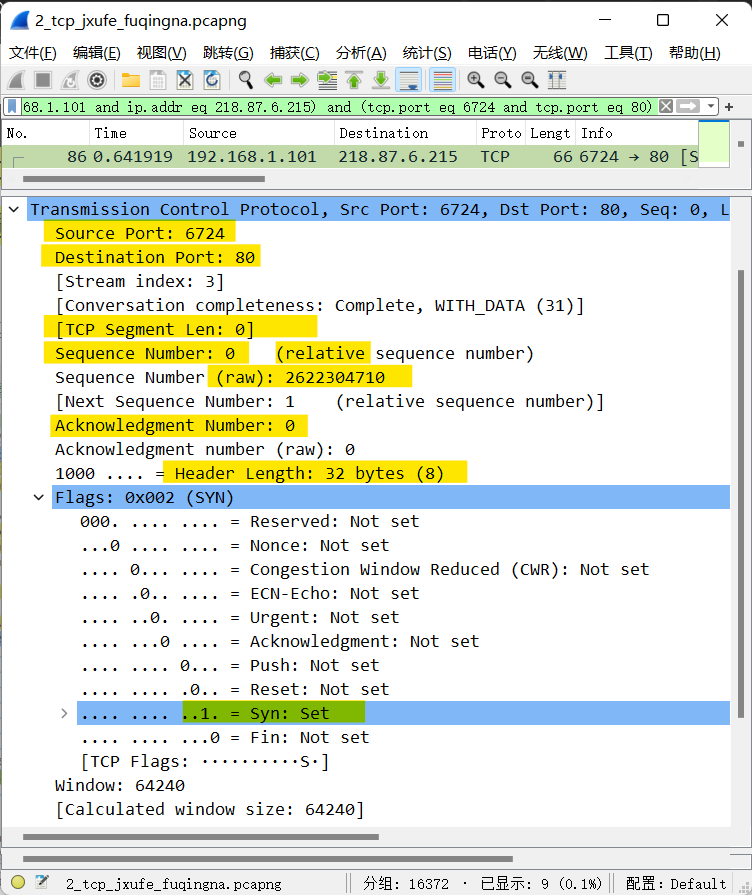
相对序列号为零，即相对于客户端的发送缓存区，它的排序为0，序列号为0。

在客户端的发送缓存中，下一个要发送的字节序列号为1。这说明第一次握手占用了一个序列号。



确认序号为0，因为服务器什么都没有返回，客户端不知道下一个从服务器发回来的包的序号应该是什么，所以填充0。

### 6.1.5 flags

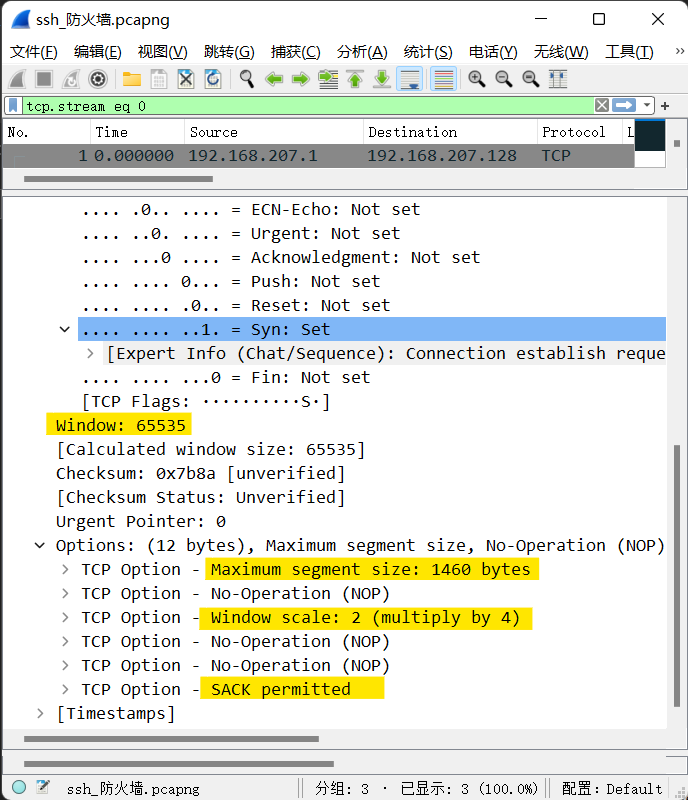


在1号TCP包中，flags字段为十六进制0x002，即十进制2，二进制10，倒数第二位的syn字段为1，即设置了syn，只有在前两次握手中 SYN 才为 1，而本packet就是第一次握手。

ACK=0，代表这不是一个确认的TCP包。

SYN=1， ACK=0表示连接请求报文段，即第一次握手。

### 6.1.6 窗口

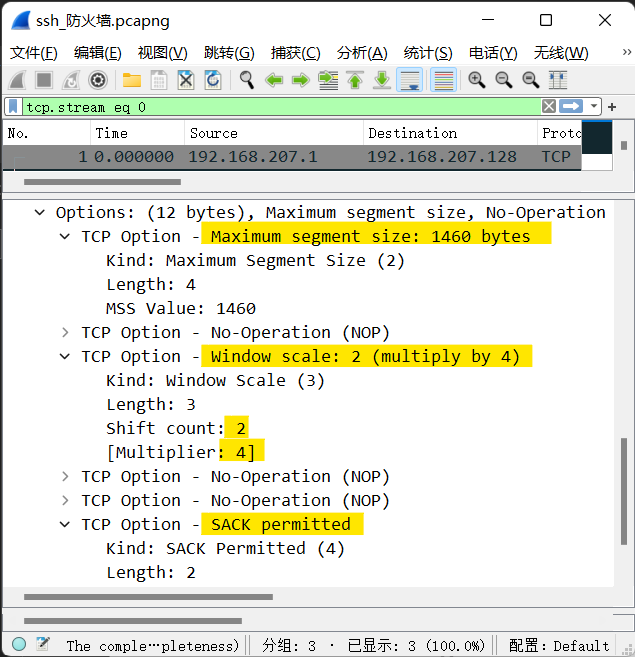


窗口大小为65535，这是wires hark根据选项字段中的Window scale调整后的结果。在本TCP中，Window scale= 22 =4 bytes。另外，这个窗口大小恰好是TCP的最大总长度（因为TCP首部长度字段长16）65535。

即客户端告知服务器，下一次我（客户端）能接收65535字节的数据。

### 6.1.7 选项

选项字段长12bytes。因为TCP首部包含固定20bytes字段和变长的选项填充字段，由6.1.3可知，本TCP包的Header length为32bytes，即包含20bytes固定长度字段，和12bytes的选项与填充字段。



MSS为1460bytes，表示在这段TCP会话中不包含TCP首部的最大数据长度为1460bytes。这样设置的原因与以太网MTU=1518有关：

1518=18以太网帧头帧尾+1500IP数据报

=18以太网帧头帧尾+20IP首部+1480IP数据部分

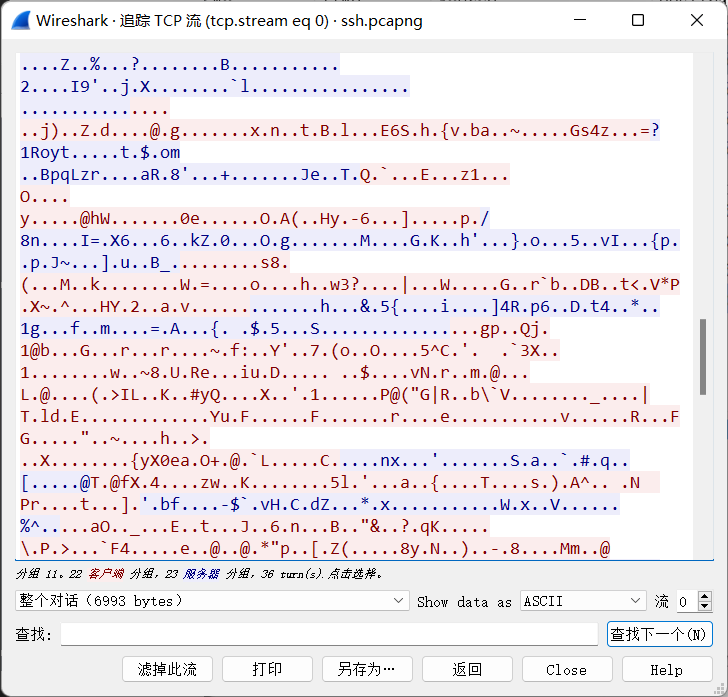
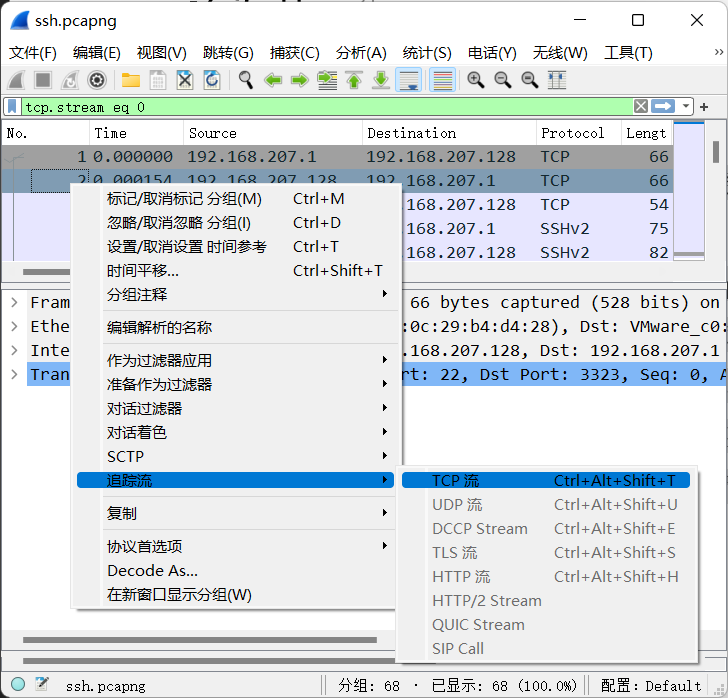
=18以太网帧头帧尾+20IP首部+20TCP固定首部+1460TCP数据部分

所以，设置MSS为1460bytes可以有效避免分片。

Window scale是调整窗口大小的一个参数。Window scale 种类为3时，Window scale=2 shift count ,即Window scale真正的值是根据选项字段中的shift count计算得来的。如果选项字段中设置了Window scale，意味着真正的窗口值要在原有的窗口值上乘以Window scale。在本TCP中，Window scale= 22 =4 bytes，真正的窗口值为65535bytes。

SACK设置了Sack-Permitted选择性确认允许选项。即在第一次握手中，客户端支持SACK。Sack-Permitted只可以在TCP syn中发送，即TCP建立连接的前两次握手中发送，分別表示通信双方各自是否支持SACK。一旦连接建立，该 TCP 会话就设置相应的SACK 选择性确认选项。

## 6.2 sshd\_on



与telnet不同，ssh追踪到的流都是乱码，因为telnet是明码传输，ssh是加密传输。

sshd的优势在于，发送方需要经过加密算法加密，接收方需要解密，安全性大大提高，不能够通过抓包分析轻易破解sshd用户的隐私。但是加密、解密的过程导致其传输速度、效率较telnet低，但是由于它具备telnet不具有的安全性，所以更普遍采用的还是sshd。