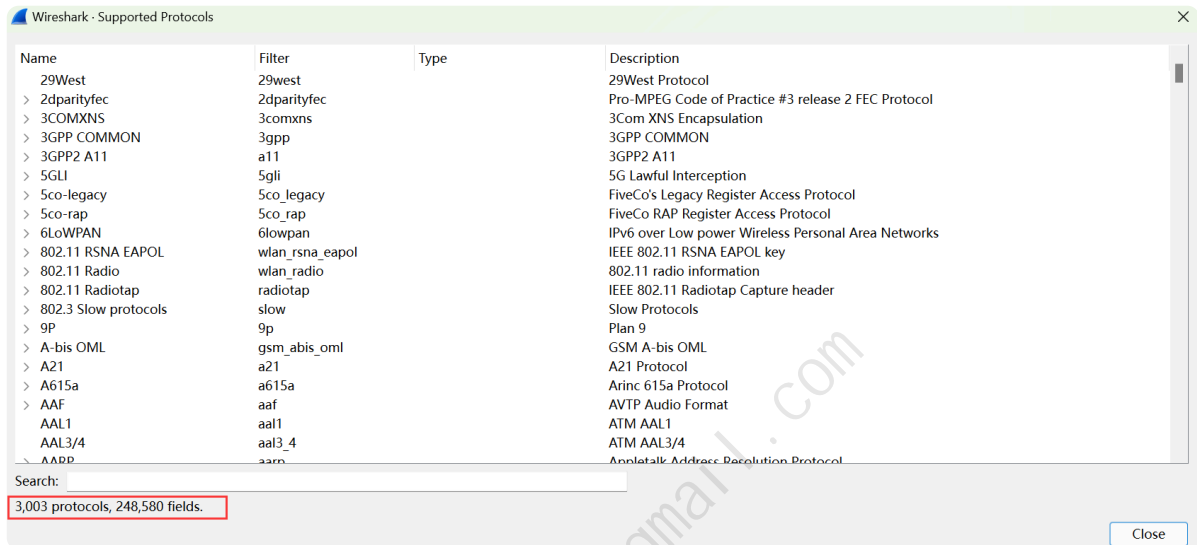


一、前言

Wireshark是一款强大的网络协议分析工具，能够捕获并分析网络中的数据包。本文将详细介绍如何通过Wireshark的精准过滤规则，帮助用户从海量数据报文中精确提取出所需的数据包，从而更有效地进行网络故障排查和安全分析。

本文由两大部分组成，第一部分介绍Wireshark的过滤字段的技巧，第二部分则具体化到各类应用场景中进行案例分析。Wireshark支持的协议有3000多个，过滤字段24万多个，因此本文不可能每一个都能覆盖到，但过滤方法是一层不变的，需要什么过滤什么。



任何教程手册都无法超越官方文档，强烈建议读者读完本篇后如需更深入全面了解Wireshark，可阅读官方文档向导：

文档名称	文档链接
wireshark-filter(4) Manual Page	https://www.wireshark.org/docs/man-pages/wireshark-filter.html
Wireshark User's Guide	https://www.wireshark.org/docs/wsug_html_chunked/
文档下载页	https://www.wireshark.org/download/docs/

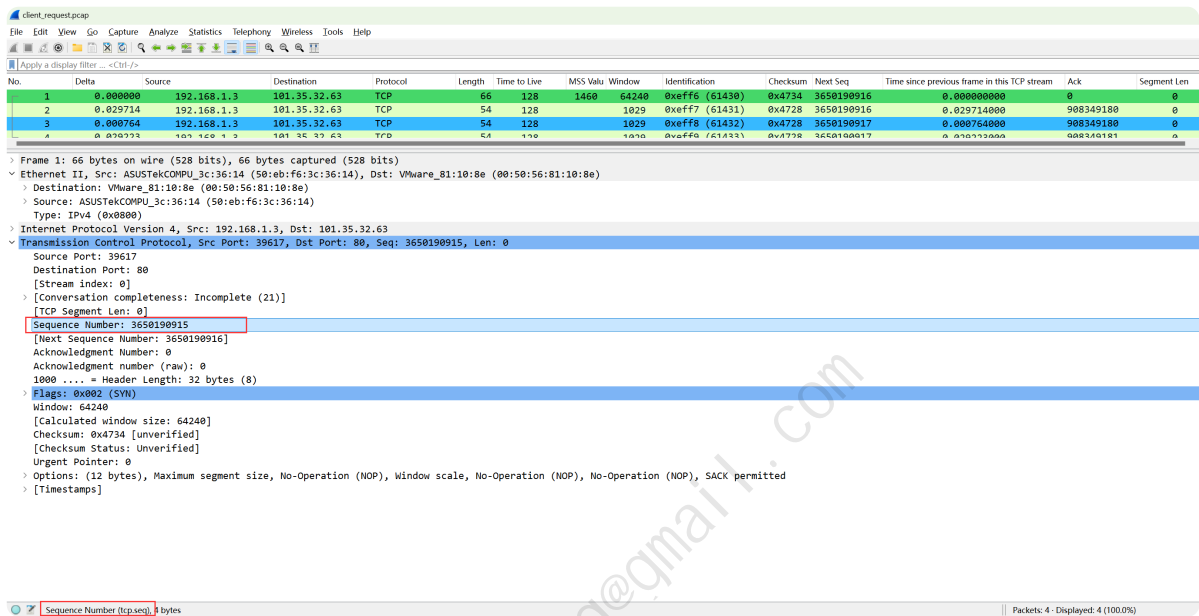
同时，你能在Wireshark使用的过滤命令，在Tshark中也可以同样使用，Tshark为Wireshark官方出品组件，可理解为CLI版的Wireshark，Wireshark和Tshark都是基于libpcap库的工具，共享相同的过滤语法，称为pcap-filter。如需了解Tshark的用法案例，可参考笔者的[这篇文章](#)。

二、过滤技巧、操作符、表达式

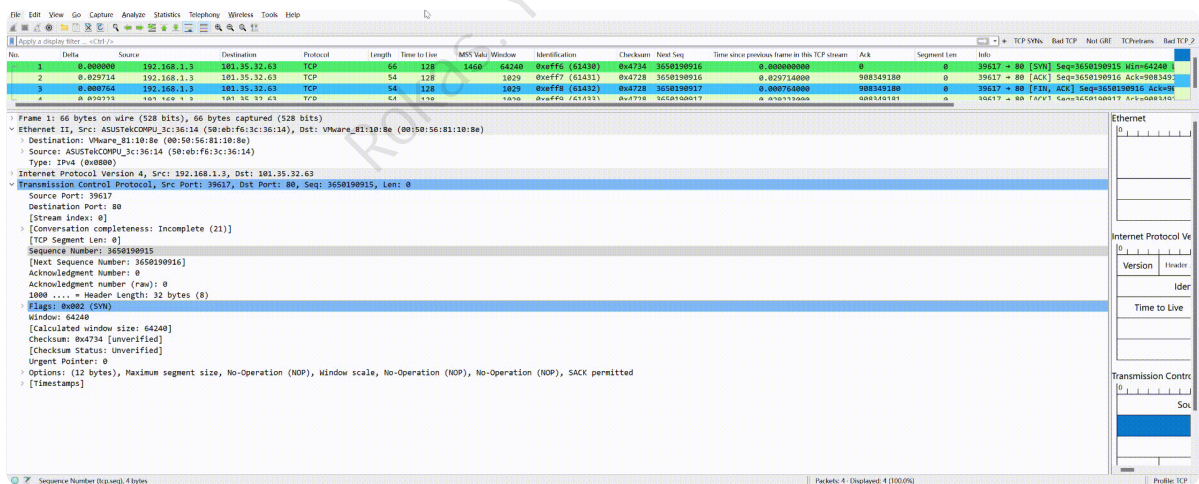
2.1 任何字段都能成为过滤条件

2.1.1 鼠标拖动任意字段过滤

将报文展开后，你鼠标所点击的任何字段，都能作为过滤条件，比如鼠标点到Sequence Number这个字段，最下面左下角会展示对应的字段过滤语法：



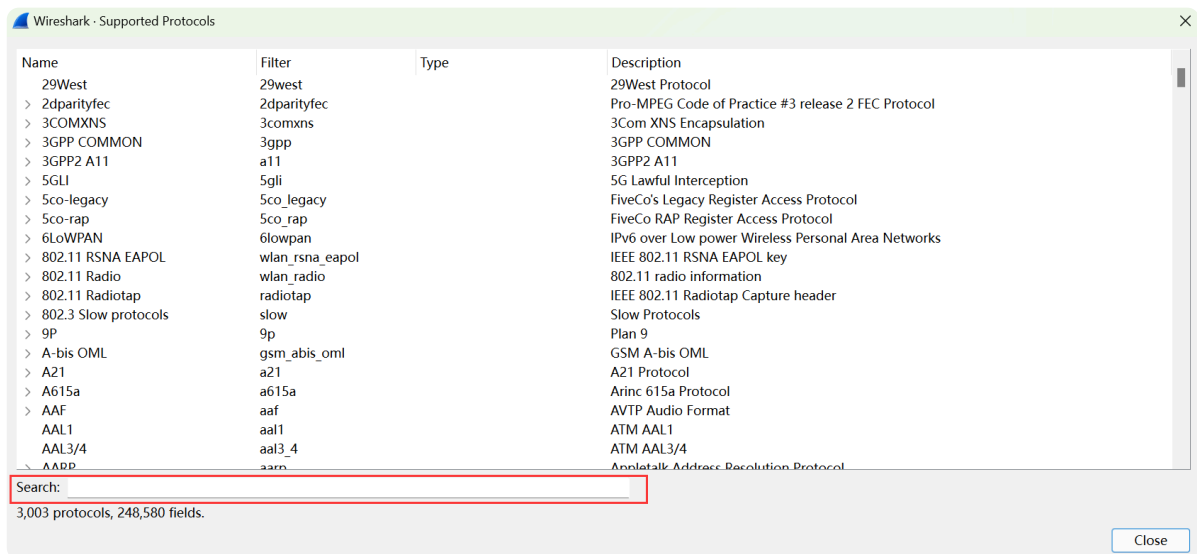
你甚至不需要手敲这个过滤条件，直接使用鼠标把它拖动到最上面过滤框中：



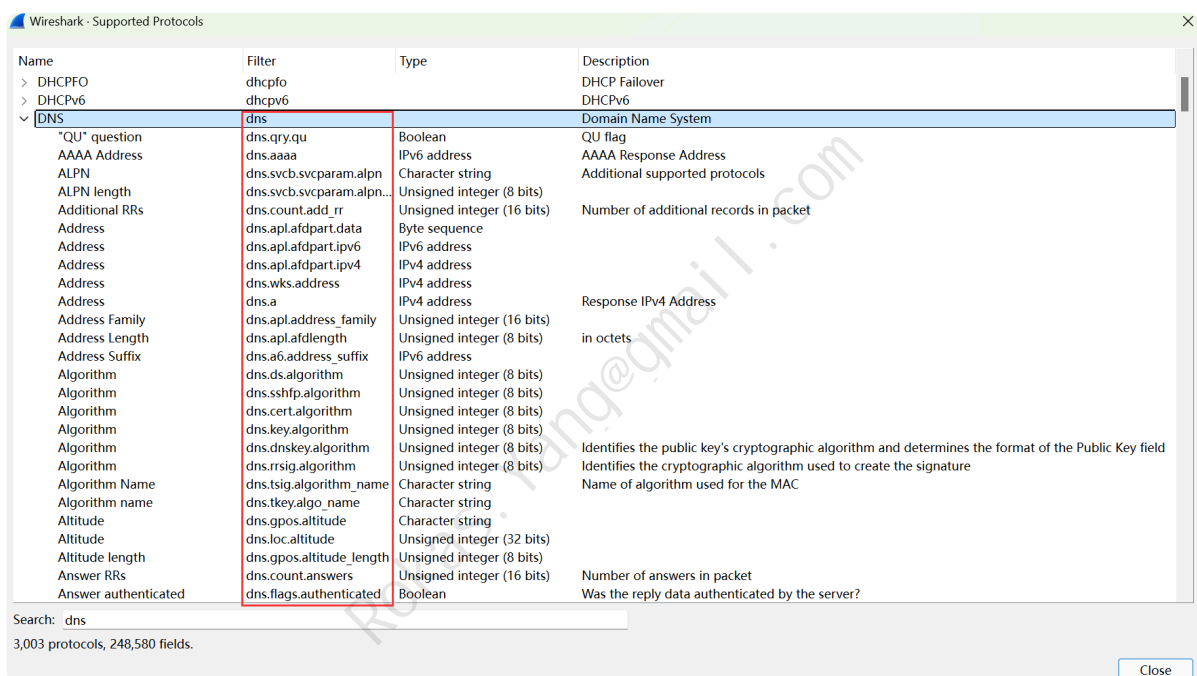
其它任何字段都是同理。

2.1.2 查看并搜索过滤字段

在视图 (View) --> 内部 (Internals) --> 支持的协议 (Supported Protocols) 即可查看支持的所有协议，在搜索框中键入你想要搜索的字段：



将会展示所有匹配的协议字段，即使协议子字段能匹配搜索的内容也会展示出来，比如搜索dns，注意不要回车，等待几秒后将会输出结果：



通常，通过协议分析排错，首先需要了解协议的正常交互行为，了解每个报文字段的作用，才能更进一步去筛选异常的交互请求、响应报文。

2.2 逻辑操作符

操作符	类C风格	描述	示例
and	&&	逻辑与	ip.src==10.0.0.5 and tcp.flags.fin==1
or		逻辑或	ip.src==10.0.0.5 or ip.src==192.1.1.1
xor	^^	逻辑异或	tr.dst[0:3] == 0.6.29 xor tr.src[0:3] == 0.6.29
not	!	逻辑非	!udp
[...]	不涉及	子序列/切片操作符	eth.src[:4] == 00:00:83:00
in	不涉及	设定集合里的成员	http.request.method in {"HEAD", "GET"}

注意：逻辑操作符是区分大小写的，一定要小写，或者使用类C风格写法。

与或非三种基本运算这里不赘述，可以参照上面表格中的示例，主要讲讲异或、子序列、集合三种操作符。

2.2.1 异或 (xor)

当且仅当满足其中一个条件，并且是两个条件不能同时满足时，为真，过滤出对应的数据包。

比如下面这个表达式：

```
ip.addr == 192.168.1.1 xor ip.addr == 10.10.0.100
```

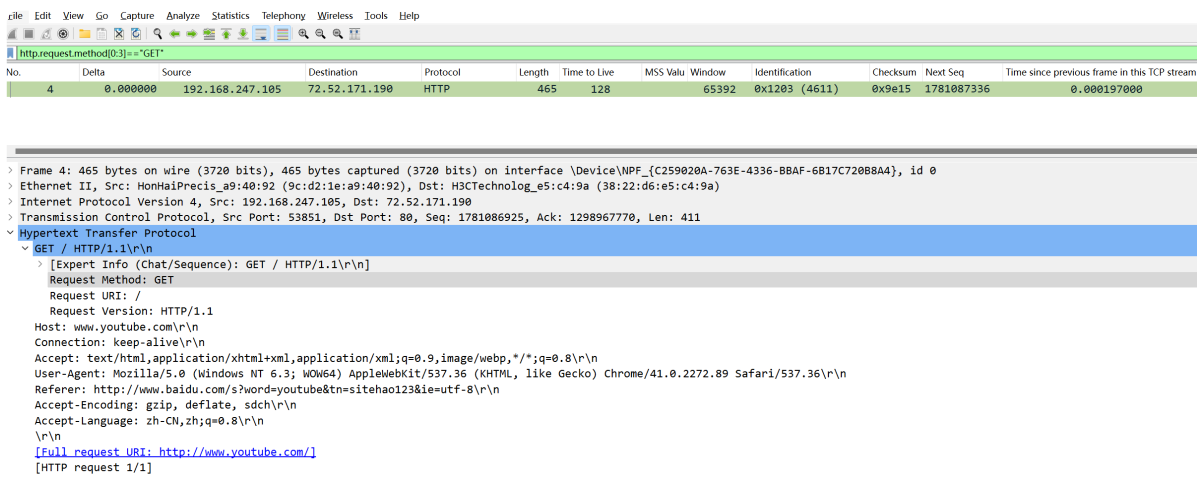
筛选ip地址为192.168.1.1的报文，或者ip.addr为10.10.0.100的报文，但不能同时满足两个条件，也就是192.168.1.1和10.10.0.100之间的交互请求，不会被匹配到，但它们两和其它IP的交互，能正常匹配。

2.2.2 子序列 ([...])

类似于数组用法，或者Python里的切片用法。

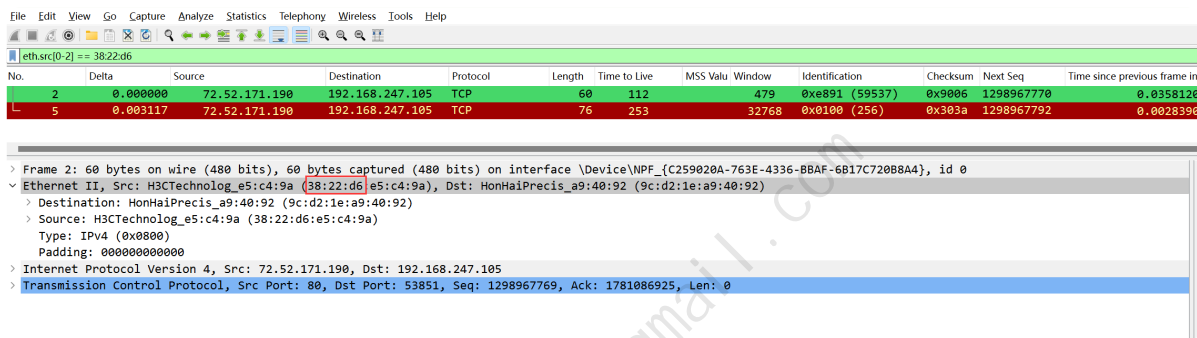
比如下面这个示例，过滤http请求方法，前三个字符为GET的报文：

```
http.request.method[0:3]=="GET"
```

又或者指定源mac地址前三位可以是：

```
eth.src[0-2] == 38:22:d6
```



2.2.3 集合 (in)

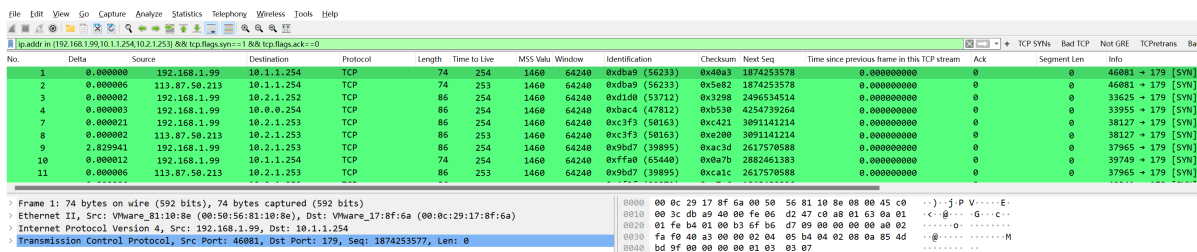
在过滤条件的基础上，只要符合集合内的元素，全部匹配。

同时筛选多个IP地址，可以是：

```
ip.addr in {192.168.1.99, 10.1.1.254, 10.2.1.253}
```

再结合上面的and逻辑操作符，只筛选SYN请求的：

```
ip.addr in {192.168.1.99, 10.1.1.254, 10.2.1.253} && tcp.flags.syn==1 && tcp.flags.ack==0
```



同理，想一次性筛选多个端口的报文并且TCP流里面没有拿到对端SYN-ACK第二次握手包，可以是：

```
tcp.port in {80,8080,443} && tcp.completeness.syn-ack==0
```

No.	Time	Source	Destination	Protocol	Length	Time to Live	MSS	Win	Checksum	Next Seq	Time since previous frame in this TCP stream	ACK	Segment Len	Info
4	0.000000	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x1180 286362919	0.000000000	0	11747 → 8080 [SYN] Seq=286362918 Win=1480 Len=0	
5	0.002787	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x1760 0	0.001787000	286362919	0	8080 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
6	0.009276	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x1180 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 8080 [SYN] Seq=286362918 Win=1480 Len=0	
7	0.001589	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x1760 0	0.001589000	286362919	0	8080 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
11	2.003193	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x2f80 286362919	0.000000000	0	11747 → 443 [SYN] Seq=286362918 Win=1480 Len=0	
12	0.002225	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x3535 0	0.002225000	286362919	0	443 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
13	1.005746	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x1180 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 8080 [SYN] Seq=286362918 Win=1480 Len=0	
14	0.001386	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x1760 0	0.001386000	286362919	0	8080 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
22	2.000948	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x2f80 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 443 [SYN] Seq=286362918 Win=1480 Len=0	
23	0.001250	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x3535 0	0.001250000	286362919	0	443 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
24	1.001744	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x1180 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 8080 [SYN] Seq=286362918 Win=1480 Len=0	
25	0.002209	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x1760 0	0.002209000	286362919	0	8080 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
29	2.000948	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x2f80 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 443 [SYN] Seq=286362918 Win=1480 Len=0	
30	0.001382	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x3535 0	0.001382000	286362919	0	443 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
31	1.004456	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x1180 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 8080 [SYN] Seq=286362918 Win=1480 Len=0	
32	0.001643	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x1760 0	0.001643000	286362919	0	8080 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
36	2.011345	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x2f80 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 443 [SYN] Seq=286362918 Win=1480 Len=0	
37	0.002238	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x3535 0	0.002238000	286362919	0	443 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	
38	1.002461	192.168.1.14	192.168.1.8	TCP	54	64	0	1480	0x609f (5691)	0x1180 286362919	0.000000000	0	[TCP port numbers reused] 11747 → 8080 [SYN] Seq=286362918 Win=1480 Len=0	
39	0.001364	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)	0x1760 0	0.001364000	286362919	0	8080 → 11747 [RST, ACK] Seq=286362918 Win=0 Len=0	

Frame 4: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface eth0
Ethernet II, Src: VMware_9f:df:5f (00:50:56:9f:df:5f), Dst: VMware_81:c0:1e (00:50:56:81:c0:1e)
Internet Protocol Version 4, Src: 192.168.1.14, Dst: 192.168.1.8
Transmission Control Protocol, Src Port: 11747, Dst Port: 8080, Seq: 286362918, Len: 0

可以看到只有443、8080端口符合筛选条件，测试的对端服务器没有监听这两个端口，所以响应了RST-ACK。

```
(root@kali)~# nping --tcp -p 80,8080,443 192.168.1.8 -c 1
Starting Nping 0.7.94SVN ( https://nmap.org/nping ) at 2024-10-13 12:12 EDT
SENT (0.0172s) TCP 192.168.1.14:13993 > 192.168.1.8:80 S ttl=64 id=42215 iplen=40 seq=516099702 win=1480
RCVD (0.0309s) TCP 192.168.1.8:80 > 192.168.1.14:13993 SA ttl=64 id=0 iplen=44 seq=1942392822 win=64240 <mss 1460>
SENT (1.0180s) TCP 192.168.1.14:13993 > 192.168.1.8:443 S ttl=64 id=42215 iplen=40 seq=516099702 win=1480
RCVD (1.0198s) TCP 192.168.1.8:443 > 192.168.1.14:13993 [RA] ttl=64 id=0 iplen=40 seq=0 win=0
SENT (2.0196s) TCP 192.168.1.14:13993 > 192.168.1.8:8080 S ttl=64 id=42215 iplen=40 seq=516099702 win=1480
RCVD (2.0237s) TCP 192.168.1.8:8080 > 192.168.1.14:13993 [RA] ttl=64 id=0 iplen=40 seq=0 win=0
RST-ACK
Max rtt: 13.707ms | Min rtt: 1.661ms | Avg rtt: 6.461ms
Raw packets sent: 3 (120B) | Rcvd: 3 (138B) | Lost: 0 (0.00%)
Nping done: 1 IP address pinged in 2.06 seconds
(root@kali)~#
```

集合的用法也可以是连续性的，比如过滤443端口和4430~4434端口：

```
tcp.port in {443, 4430..4434}
```

或者IP地址范围：

```
ip.addr in {10.0.0.5 .. 10.0.0.9, 192.168.1.1..192.168.1.9}
```

2.3 比较操作符

操作符	别名	类C风格	描述	示例
eq	any_eq	==	等于	ip.src == 10.0.0.5
ne	all_ne	!=	不等于	ip.src != 10.0.0.5
	all_eq	===	全等	ip.src === 10.0.0.5
	any_ne	!==	不全等	ip.src !== 10.0.0.5
gt		>	大于	frame.len > 10
lt		<	小于	frame.len < 128
ge		>=	大于或等于	frame.len ge 0x100
le		<=	小于或等于	frame.len <= 0x20
contains			协议、字段或切片包含某个值	sip.To contains "a1762"
matches		~	使用Perl正则 (PCRE) 匹配一个协议或文本字段	http.host matches "acme.(org com net)"

上面的等于、不等于、大于、小于、大于等于、小于等于等基本操作符不一一介绍，介绍几个不常见但又很有用的操作符。

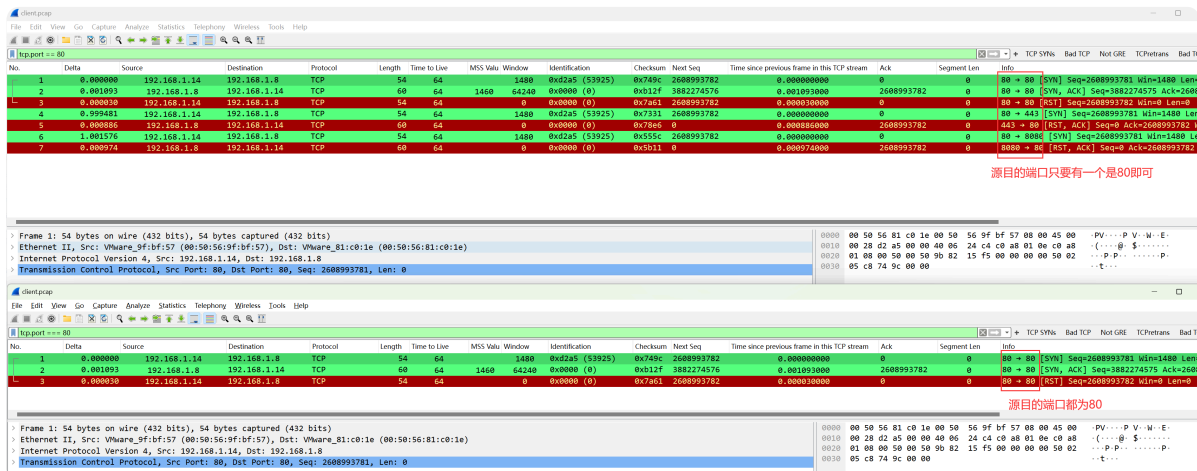
2.3.1 全等 (===)

等于 (==) 和全等 (===) 的区别：

- == 是一种宽松的比较，只要有一个值匹配即可 (any if more than one) ；
- === 是一种严格的比较，所有可能的值都必须匹配 (all if more than one) 。

比如下面这个示例，过滤TCP端口号为80的请求：

```
tcp.port == 80 # 等于的写法
tcp.port === 80 # 全等的写法
```



同一个包，使用等于(=)可以过滤出源或目的端口为80报文，使用全等(===)则会匹配源目的端口都为80的报文。

因为源目的端口，上面的过滤条件一写上去，都属于tcp.port需要校验的字段，而=、===按照各自的宽松程度进行匹配筛选。

再比如下面这个例子，过滤源目的IP必须为10.0.0.0/8网段的：

ip.addr === 10.0.0.0/8

Frame 6: 184 bytes on wire (832 bits), 184 bytes captured (832 bits) on interface 0
Ethernet II, Src: S2:54:00:21:21:23 (52:54:00:21:21:23), Dst: fe:ee:d8:21:22:23 (fe:ee:d8:21:22:23)
Internet Protocol Version 4, Src: 10.120.9.130, Dst: 10.120.9.216
User Datagram Protocol, Src Port: 2745, Dst Port: 4789
Virtual Extensible Local Area Network
Ethernet II, Src: fe:ee:d8:21:22:23 (fe:ee:d8:21:22:23), Dst: 52:54:00:47:fa:1c (52:54:00:47:fa:1c)
Internet Protocol Version 4, Src: 10.120.9.62, Dst: 10.120.9.130
Transmission Control Protocol, Src Port: 26886, Dst Port: 51801, Seq: 672187657, Ack: 2716224070, Len: 0

上面这条全等语句，限定了源目的IP地址都必须为10网段，等同于：

ip.src==10.0.0.0/8 && ip.dst==10.0.0.0/8

2.3.2 不全等 (!=)

顾名思义，还是上面的过滤端口号为例，如果过滤字段有一个不等于，那么就满足条件。

比如，下面这个过滤条件：

tcp.port != 80

Frame 1: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
Ethernet II, Src: VMware_9f:bf:57 (00:50:56:9f:bf:57), Dst: VMware_B1:c0:1e (00:50:56:81:c0:1e)
Internet Protocol Version 4, Src: 192.168.1.14, Dst: 192.168.1.8
Transmission Control Protocol, Src Port: 80, Dst Port: 80, Seq: 268893781, Len: 0

Frame 4: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
Ethernet II, Src: VMware_9f:bf:57 (00:50:56:9f:bf:57), Dst: VMware_B1:c0:1e (00:50:56:81:c0:1e)
Internet Protocol Version 4, Src: 192.168.1.14, Dst: 192.168.1.8
Transmission Control Protocol, Src Port: 80, Dst Port: 443, Seq: 268893781, Len: 0

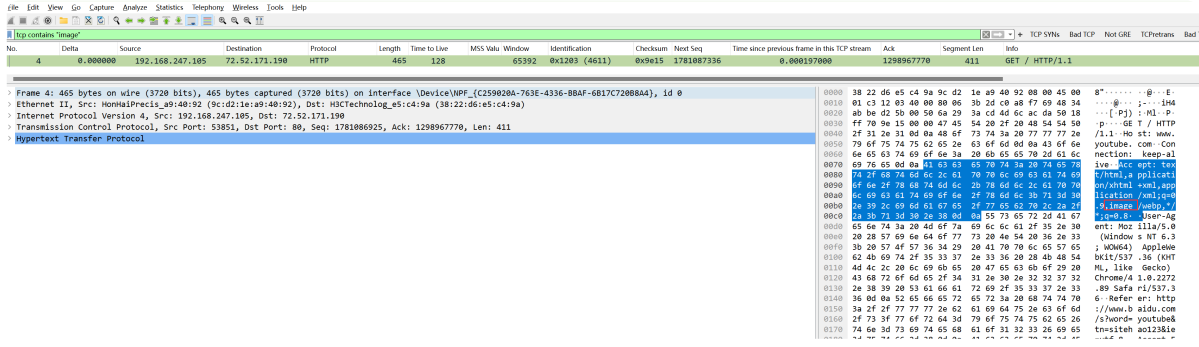
对比等于(=)，不全等(!=)排除了当源目的端口都是80的情况，以帧为维度，源或目的端口，只能满足一个是80的。

2.3.3 包含 (contains)

当要过滤某个字段是否包含指定的字符串时, 可以用contains.

比如tcp连接中, 包含字符串"image"的请求, 过滤方式可以是:

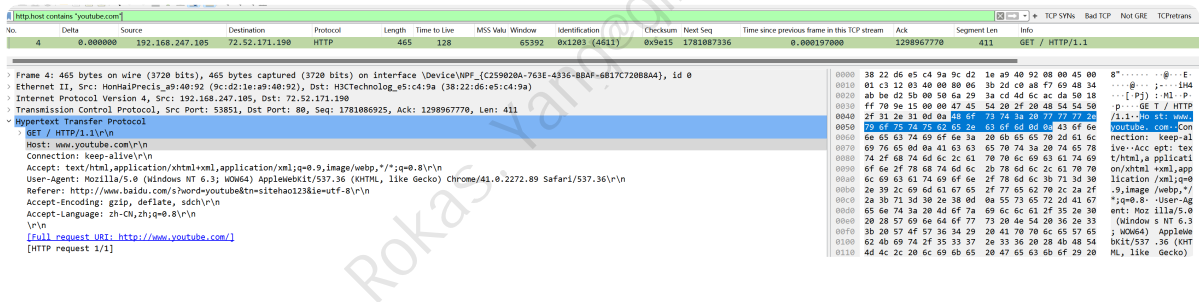
```
tcp contains "tcp"
```



当然, 把tcp替换成你想要的任何协议或字段都可以, 比如: udp contains、frame contains、http contains, 前提数据类型为string, 所以tcp.port、ip.addr这些则用不了contains.

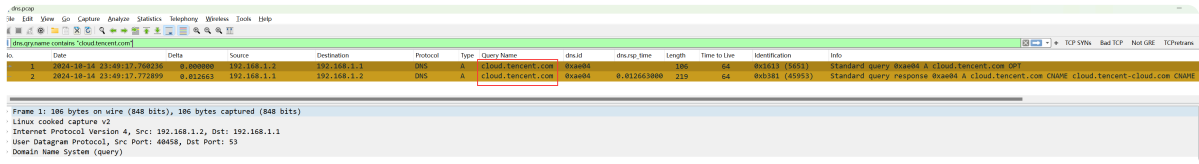
比如过滤http请求的主机名host为youtube.com的可以是:

```
http.host contains "youtube.com"
```



过滤dns查询请求包含cloud.tencent.com的可以是:

```
dns.qry.name contains "cloud.tencent.com"
```



又或者直接过滤Payload的数据, 比如nping发起一个UDP的协议探测, 发送一个测试字符串"test":

```
nping --udp -p 2115 --data-string "test" 192.168.1.72
```

服务端监听2115 udp端口, 并且收到什么回显什么:

```
socat -v udp-l:2115,fork exec:'/bin/cat'
```

```

23:57:53 ~ nping --udp -p 2115 --data-string "test" 192.168.1.72
Starting Nping 0.7.94 ( https://nmap.org/nping ) at 2024-10-14 23:58 CST
SENT (0.0177s) UDP 192.168.1.2:53 > 192.168.1.72:2115 ttl=64 id=18968 iplen=32
RCVD (0.0222s) UDP 192.168.1.72:2115 > 192.168.1.2:53 ttl=64 id=15257 iplen=32
SENT (1.0185s) UDP 192.168.1.2:53 > 192.168.1.72:2115 ttl=64 id=18968 iplen=32
RCVD (1.0228s) UDP 192.168.1.72:2115 > 192.168.1.2:53 ttl=64 id=38685 iplen=32
SENT (2.0200s) UDP 192.168.1.2:53 > 192.168.1.72:2115 ttl=64 id=18968 iplen=32
RCVD (2.0248s) UDP 192.168.1.72:2115 > 192.168.1.2:53 ttl=64 id=10447 iplen=32
SENT (3.0221s) UDP 192.168.1.2:53 > 192.168.1.72:2115 ttl=64 id=18968 iplen=32
RCVD (3.0262s) UDP 192.168.1.72:2115 > 192.168.1.2:53 ttl=64 id=52465 iplen=32
SENT (4.0244s) UDP 192.168.1.2:53 > 192.168.1.72:2115 ttl=64 id=18968 iplen=32
RCVD (4.0277s) UDP 192.168.1.72:2115 > 192.168.1.2:53 ttl=64 id=59197 iplen=32

Max rtt: 4.659ms | Min rtt: 3.268ms | Avg rtt: 4.088ms
Raw packets sent: 5 (160B) | Rcvd: 5 (230B) | Lost: 0 (0.00%)
Nping done: 1 IP address pinged in 4.08 seconds

23:58:19 ~
Debian dnscrypt&dnsmasq&nsd x
23:57:47 ~ socat -v udp-l:2115,fork exec:'/bin/cat'
> 2024/10/14 23:58:15.129980 length=4 from=0 to=3
test< 2024/10/14 23:58:15.131937 length=4 from=0 to=3
test2024/10/14 23:58:15 socat[1797817] E read(5, 0x5617b9225000, 8192): Connection refused
> 2024/10/14 23:58:16.130988 length=4 from=0 to=3
test< 2024/10/14 23:58:16.132485 length=4 from=0 to=3
test2024/10/14 23:58:16 socat[1797827] E read(5, 0x5617b9225000, 8192): Connection refused
> 2024/10/14 23:58:17.132484 length=4 from=0 to=3
test< 2024/10/14 23:58:17.134507 length=4 from=0 to=3
test2024/10/14 23:58:17 socat[1797837] E read(5, 0x5617b9225000, 8192): Connection refused
> 2024/10/14 23:58:18.133892 length=4 from=0 to=3
test< 2024/10/14 23:58:18.135850 length=4 from=0 to=3
test2024/10/14 23:58:18 socat[1797847] E read(5, 0x5617b9225000, 8192): Connection refused
> 2024/10/14 23:58:19.136606 length=4 from=0 to=3
test< 2024/10/14 23:58:19.137436 length=4 from=0 to=3
test2024/10/14 23:58:19 socat[1797857] E read(5, 0x5617b9225000, 8192): Connection refused

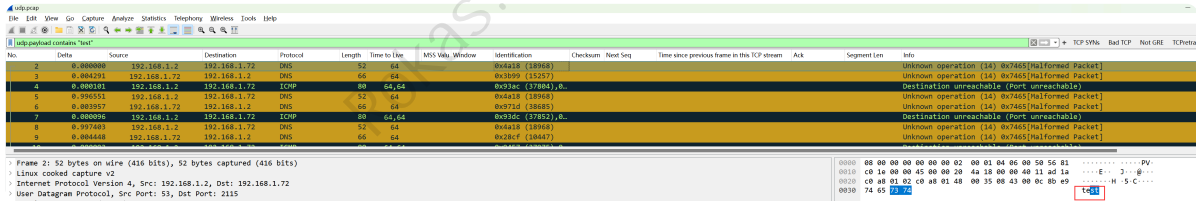
```

使用如下两个语句都能正常过滤到包含test的payload:

```

udp contains "test"
udp.payload contains "test"

```



2.3.4 匹配 (matches)

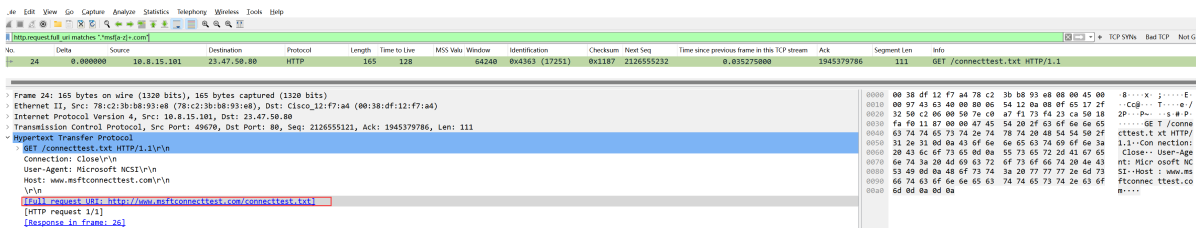
匹配模式，支持Perl正则 (PCRE) 匹配一个协议或文本字段，比contains更具灵活性。

示例1，过滤请求URI中包含msf字符串的.com网站:

```

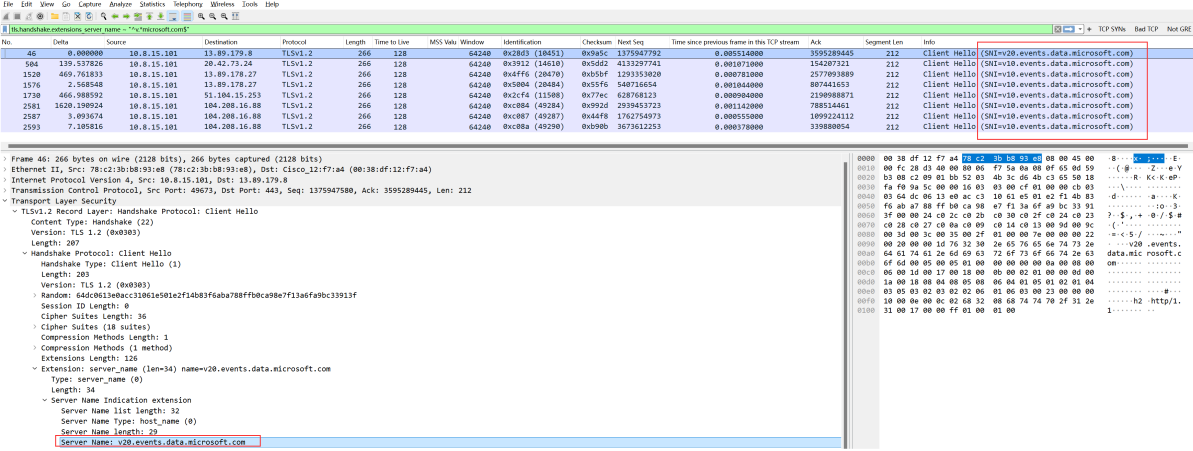
http.request.full_uri matches ".*msf[a-z]+.com"

```



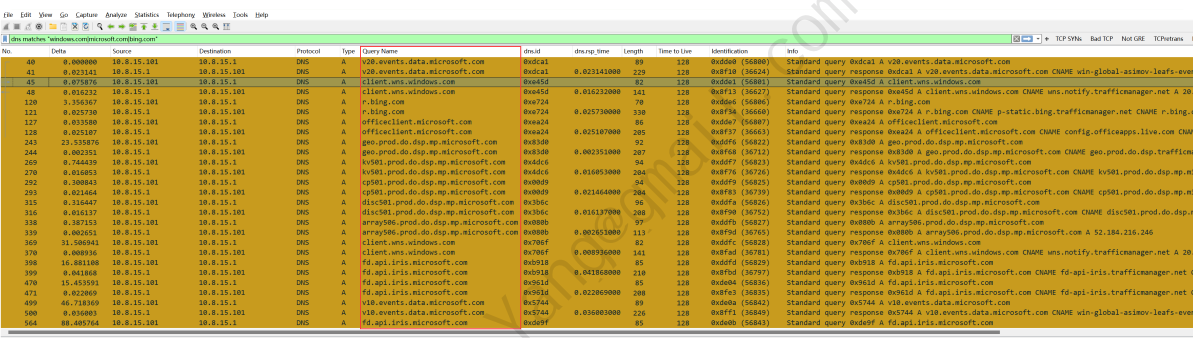
示例2，过滤TLS握手中client hello阶段的域名，必须以v字符开头，microsoft.com结尾的报文:

tls.handshake.extensions_server_name ~ ".*.microsoft.com\$"

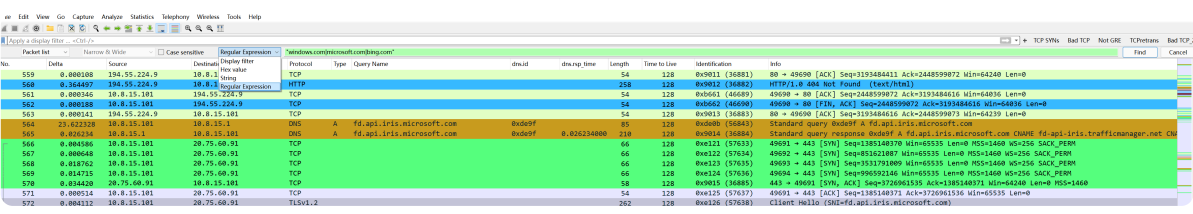


示例3, 过滤DNS协议中, 匹配其中三个网站的报文:

dns.matches "windows.com|microsoft.com|bing.com" #dns写成dns.qry.name也是没问题的



除此之外, 你可以用快捷键 **Ctrl + F** 来呼出搜索框, 支持正则、十六进制、字符串、过滤器, 并且可以设置是否大小写敏感, 搜索不会帮你筛选过滤报文, 每点击一次查找, 从上到下按顺序, 每次定位到一个符合要求的数据帧。



2.4 层级操作符 (隧道封装场景的多层级过滤)

层级操作符 (#) 后面跟十进制数, 可以将字段限制到协议栈中的某一层。

2.4.1 VXLAN场景

比如下面这个VXLAN的封包，通过UDP封装了内层IP头，再加上报文本身自带的IP头，因此出现了内、外两层：

The image shows a Wireshark packet capture of a VXLAN packet. The packet structure is as follows:

- Frame 6: 184 bytes on wire (832 bits), 184 bytes captured (832 bits)
- Ethernet II, Src: [redacted], Dst: [redacted]
- Internet Protocol Version 4, Src: 10.120.9.130, Dst: 10.120.9.216
- User Datagram Protocol, Src Port: 2745, Dst Port: 4789
- Virtual extensible Local Area Network
- Ethernet II, Src: [redacted], Dst: [redacted]
- Internet Protocol Version 4, Src: 10.120.9.62, Dst: 10.120.9.130
- Transmission Control Protocol, Src Port: 26086, Dst Port: 51801, Seq: 672107657, Ack: 2716224870, Len: 0

此时想通过过滤内层（第二层）的IP，那么可以加上层操作符#2，比如过滤内层IP为10.120.9.130的包，并且tcp目的端口为51801的报文，可以这么写：

```
ip.dst#2 == 10.120.9.130 && tcp.dstport == 51801
```

The image shows the results of applying the filter `ip.dst#2 == 10.120.9.130 && tcp.dstport == 51801`. The packet list shows multiple packets from source 10.120.9.62 to destination 10.120.9.130, all with destination port 51801. The packet details pane shows the nested structure of the packet, including the inner IP header (src: 10.120.9.62, dst: 10.120.9.130) and the outer IP header (src: 10.120.9.130, dst: 10.120.9.216).

而如果想通过第一层源IP来过滤，则默认使用ip.src或者加上#1都是可以的，再组合上面所讲的全等符号，要求外层源目的都是10网段，可以是：

```
ip.src#1 == 10.120.9.130 && tcp.dstport == 51801 && ip.addr#2 == 10.0.0.0/8
```

The image shows the results of applying the filter `ip.src#1 == 10.120.9.130 && tcp.dstport == 51801 && ip.addr#2 == 10.0.0.0/8`. The packet list shows multiple packets from source 10.120.9.130 to destination 10.120.9.130, all with destination port 51801. The packet details pane shows the nested structure of the packet, including the inner IP header (src: 10.120.9.130, dst: 10.120.9.130) and the outer IP header (src: 10.120.9.130, dst: 10.120.9.216).

2.4.2 GRE场景

GRE的封装同理，比如下面这个报文，GRE也封了一层IP头，再加上原始报文自带的IP头，因此出现内两层：

The image shows a Wireshark packet capture of a GRE packet. The packet structure is as follows:

- Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface eth0, id 0
- Ethernet II, Src: [redacted], Dst: [redacted]
- Internet Protocol Version 4, Src: 10.249.100.15, Dst: 10.133.100.21
- Internet Protocol Version 4, Src: 10.249.100.15, Dst: 10.133.100.21
- Internet Protocol Version 4, Src: 10.249.100.15, Dst: 10.133.100.21
- Internet Protocol Message Protocol

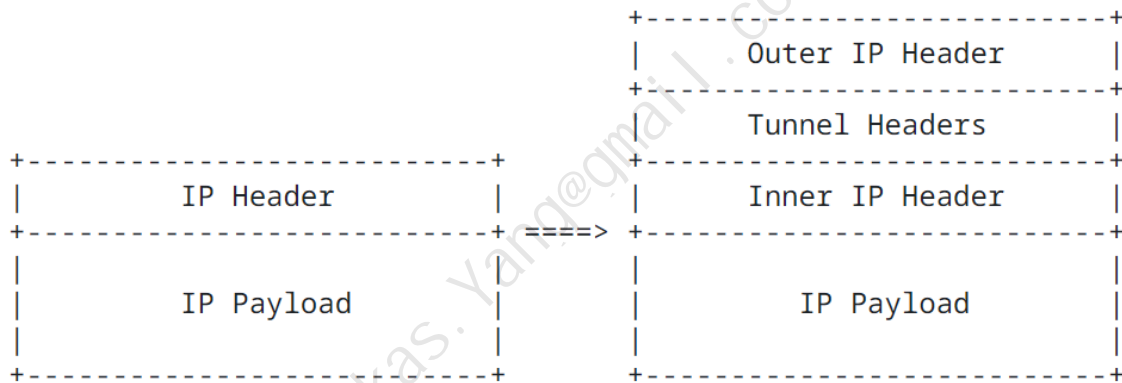
配合前面所讲的匹配操作符 (~) 和全等操作符 (===), 过滤外层源目的IP为10网段或者11网段开头, 同时外层源目的IP均为10网段, 那么可以是:

```
string(ip.addr) ~ "^10|^11" && ip.addr#2 === 10.0.0.0/8
```

这里把外层ip.addr数据类型通过string()函数转化为了字符串, 再通过匹配操作符去匹配正则表达式。

2.4.3 IP封装

IP封装亦是同理, 分内层和外层IP头, 结构如下:



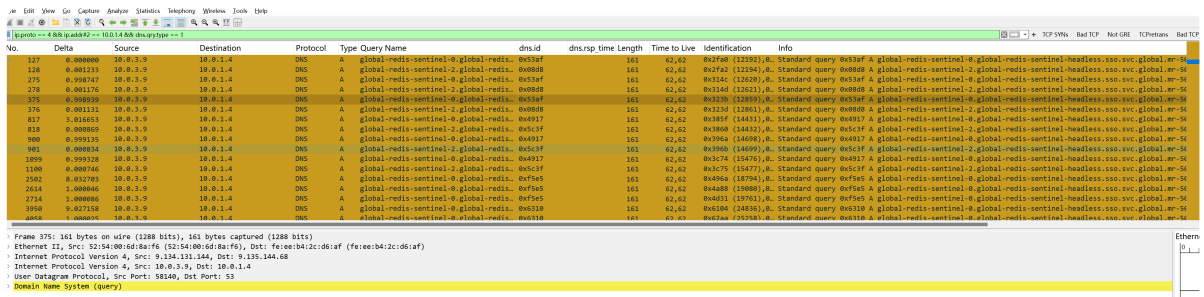
首先我们在wireshark过滤IP封装的包, 可以使用下面的语法:

```
ip.proto == 4
```

从过滤语法不难看出, IP封装 (IP Encapsulation within IP) 处于第4号协议, 协议对应的数字序号, 可以参照 IANA机构文档。过滤到IP封装报文后, 接下来可以不同层级的IP头来过滤符合需求的报文。

比如过滤内层IP头包含10.0.0.2并且为DNS记录的查询请求, 可以是:

```
ip.proto == 4 && ip.addr#2 == 10.0.1.4 && dns.qry.type == 1
```



2.5 转换类函数过滤器

Wireshark支持很多转换类的函数，参照表格：

函数	描述
upper	将字符串字段转换为大写
lower	将字符串字段转换为小写
len	返回字符串字段或字节字段的字节长度
count	返回帧中字段的出现次数
string	将非字符串字段转换为字符串
vals	将字段值转换为其值字符串（如果有）
dec	将无符号整数字段转换为十进制字符串
hex	将无符号整数字段转换为十六进制字符串
max	返回参数的最大值
min	返回参数的最小值
abs	返回参数的绝对值

上述函数不一一举例，讲述几个较为常用的函数。

2.5.1 upper()/lower()函数

可以使用这两个函数，将字符串转化为大小写，再进行正则匹配，做到不区分大小写的功能。

比如过滤HTTP响应头的server字段为apache的，可以是：

```
lower(http.server) ~ "apache"
```

No.	Delta	Source	Destination	Protocol	Length	Time to Live	MSS	Window	Identification	Checksum	Next Seq	Time since previous frame in this TC	ACK	Segment Len	Info
441	0.000000	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x8f64 (36828)	0xe0d3	4023228232	0.434111000	244910679	196	HTTP/1.0 404 Not Found (text/html)	
453	0.000004	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x8f64 (36828)	0xe0d7	623872354	0.593270000	412267241	196	HTTP/1.0 404 Not Found (text/html)	
465	0.528513	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x8f68 (36832)	0xe0e6	2799323841	0.345152000	3698826694	204	HTTP/1.0 404 Not Found (text/html)	
547	60.539435	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x900c (36876)	0xf9e0	2842480264	0.347458000	3794858661	204	HTTP/1.0 404 Not Found (text/html)	
560	60.522817	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9012 (36882)	0x6c69	3138484616	0.364497000	2448599072	204	HTTP/1.0 404 Not Found (text/html)	
606	60.546355	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9056 (36900)	0x2261	6100819429	0.364857000	3183789056	204	HTTP/1.0 404 Not Found (text/html)	
1322	60.738188	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x91ed (37557)	0x0b2e	923396664	0.397805000	292491118	204	HTTP/1.0 404 Not Found (text/html)	
1339	60.535760	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x91f7 (37567)	0x1e4d	3481339181	0.347130000	3903817844	204	HTTP/1.0 404 Not Found (text/html)	
1354	60.539320	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9200 (37576)	0x0921	868223259	0.356200000	3185379056	204	HTTP/1.0 404 Not Found (text/html)	
1372	60.549078	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9209 (37585)	0x9090	3221817941	0.366454000	3166258364	204	HTTP/1.0 404 Not Found (text/html)	
1386	60.530159	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9210 (37592)	0xc51d	805145582	0.408519000	540584423	204	HTTP/1.0 404 Not Found (text/html)	
1610	60.548634	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x920e (37578)	0xc0cf	2740232523	0.342194000	2488344249	204	HTTP/1.0 404 Not Found (text/html)	
1623	60.548781	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9207 (37573)	0x3a0b	1832847445	0.355199000	3224908812	204	HTTP/1.0 404 Not Found (text/html)	
1639	60.526088	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92aa (37546)	0x4220	313862334	0.357391000	2295866761	204	HTTP/1.0 404 Not Found (text/html)	
1653	60.556886	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9201 (37553)	0xf7fd	2231707761	0.362852000	71394013	204	HTTP/1.0 404 Not Found (text/html)	
1665	60.579698	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9208 (37566)	0x0920	1768070757	0.389508000	2624208818	204	HTTP/1.0 404 Not Found (text/html)	
1687	60.529149	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x920e (37566)	0x9090	2015833653	0.346507000	554940789	204	HTTP/1.0 404 Not Found (text/html)	
1704	60.515745	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92c7 (37575)	0x9470	1381256678	0.342085000	1626584484	204	HTTP/1.0 404 Not Found (text/html)	
1720	60.527044	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92c1f (37583)	0x8408	3462177381	0.341640000	3419737932	204	HTTP/1.0 404 Not Found (text/html)	
1766	60.508641	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9200 (37566)	0x0900	1935419277	0.361345000	244604462	204	HTTP/1.0 404 Not Found (text/html)	
1822	60.498200	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9200 (37566)	0x0900	1935419277	0.361345000	244604462	204	HTTP/1.0 404 Not Found (text/html)	

过滤http请求方法为POST或GET:

```
upper(http.request.method) ~ "post|get"
```

No.	Delta	Source	Destination	Protocol	Length	Time to Live	MSS	Window	Identification	Checksum	Next Seq	Time since previous frame in this TC	ACK	Segment Len	Info
24	0.000000	10.8.15.101	23.47.50.80	HTTP	165	128	64240	0x4363 (17251)	0x1187	2126555232	0.035275000	1945379786	111	GET /connecttest.txt HTTP/1.1	
439	84.208800	10.8.15.101	194.55.224.9	HTTP	388	128	64240	0xb488 (46664)	0x139c	244910679	0.000115000	4023228036	334	POST /luiz/five/fre.php HTTP/1.0	
451	0.640380	10.8.15.101	194.55.224.9	HTTP	278	128	64240	0xb08a (46670)	0x01ac	412427241	0.000113000	623872354	224	POST /luiz/five/fre.php HTTP/1.0	
463	0.577086	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb454 (46676)	0x110f	3698826694	0.000146000	279322836	197	POST /luiz/five/fre.php HTTP/1.0	
545	60.537139	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb45a (46682)	0x593d	3794858661	0.000130000	2842480859	197	POST /luiz/five/fre.php HTTP/1.0	
568	60.505726	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb460 (46688)	0x3247	2448599072	0.000203000	3193484411	197	POST /luiz/five/fre.php HTTP/1.0	
604	60.546007	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb466 (46694)	0x7930	1832847445	0.000213000	630269224	197	POST /luiz/five/fre.php HTTP/1.0	
1320	60.685159	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb46c (46700)	0x0f8f	292491118	0.000205000	923396459	197	POST /luiz/five/fre.php HTTP/1.0	
1337	60.576448	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb472 (46706)	0x099e	3583817844	0.000178000	3481538976	197	POST /luiz/five/fre.php HTTP/1.0	
1352	60.559147	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb478 (46712)	0x1183	1863739150	0.000255000	868232894	197	POST /luiz/five/fre.php HTTP/1.0	

2.5.2 len()函数

len()函数将返回字段的字节大小，因此可以和比较操作符配合使用，过滤某个报文字段符合大小要求的报文。

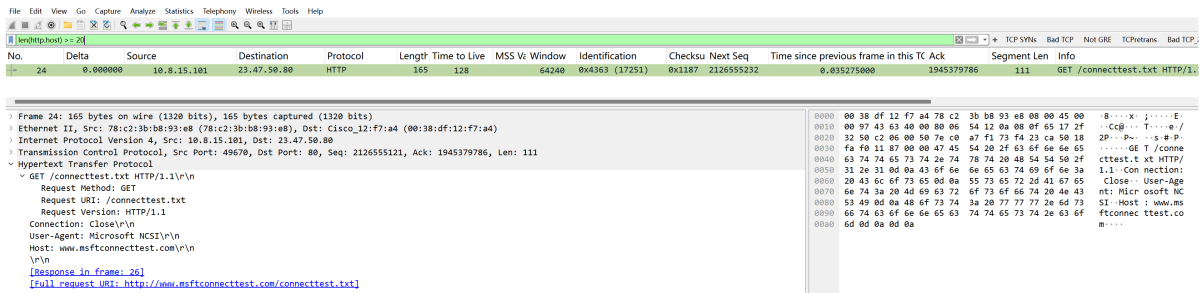
过滤http头部的URI字段，大于等于10字节的报文可以是:

```
len(http.request.uri) >= 10
```

No.	Delta	Source	Destination	Protocol	Length	Time to Live	MSS	Window	Identification	Checksum	Next Seq	Time since previous frame in this TC	ACK	Segment Len	Info
24	0.000000	10.8.15.101	23.47.50.80	HTTP	165	128	64240	0x4363 (17251)	0x1187	2126555232	0.035275000	1945379786	111	GET /connecttest.txt HTTP/1.1	
439	84.208800	10.8.15.101	194.55.224.9	HTTP	388	128	64240	0xb488 (46664)	0x139c	244910679	0.000115000	4023228036	334	POST /luiz/five/fre.php HTTP/1.0	
451	0.640380	10.8.15.101	194.55.224.9	HTTP	278	128	64240	0xb08a (46670)	0x01ac	412427241	0.000113000	623872354	224	POST /luiz/five/fre.php HTTP/1.0	
463	0.577086	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb454 (46676)	0x110f	3698826694	0.000146000	279322836	197	POST /luiz/five/fre.php HTTP/1.0	
545	60.537139	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb45a (46682)	0x593d	3794858661	0.000130000	2842480859	197	POST /luiz/five/fre.php HTTP/1.0	
568	60.505726	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb460 (46688)	0x3247	2448599072	0.000203000	3193484411	197	POST /luiz/five/fre.php HTTP/1.0	
604	60.546007	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb466 (46694)	0x7930	1832847445	0.000213000	630269224	197	POST /luiz/five/fre.php HTTP/1.0	
1320	60.685159	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb46c (46700)	0x0f8f	292491118	0.000205000	923396459	197	POST /luiz/five/fre.php HTTP/1.0	
1337	60.576448	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb472 (46706)	0x099e	3583817844	0.000178000	3481538976	197	POST /luiz/five/fre.php HTTP/1.0	
1352	60.559147	10.8.15.101	194.55.224.9	HTTP	251	128	64240	0xb478 (46712)	0x1183	1863739150	0.000255000	868232894	197	POST /luiz/five/fre.php HTTP/1.0	

过滤HTTP主机名大于等于20字节的报文:

```
len(http.host) >= 20
```

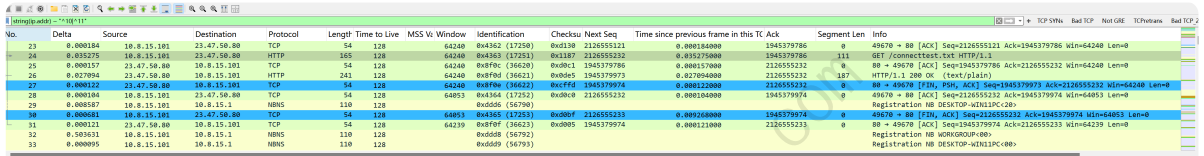


2.5.3 string()函数

较为常用，当字段为非字符串类型，而又想转换为字符串字段再进行正则匹配时，string()函数则帮了大忙。

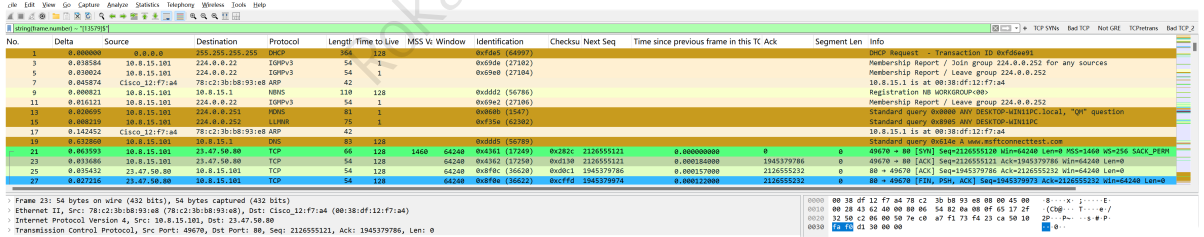
比如过滤IP为10网段开头或者23网段开头的IP，可以是：

```
string(ip.addr) ~ "^10|^23"
```



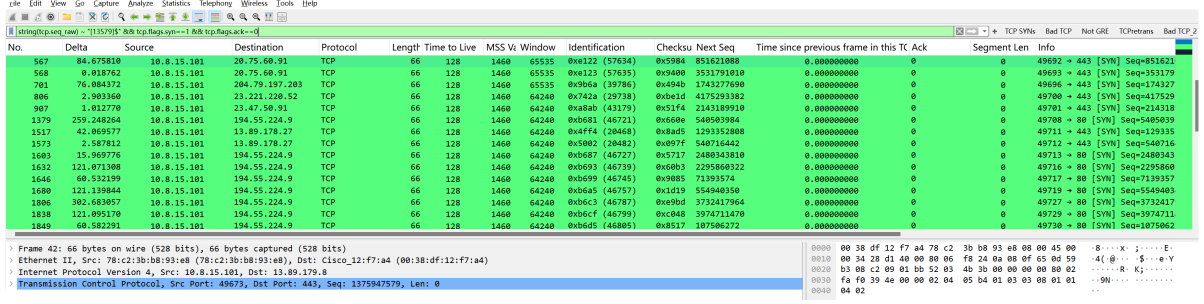
过滤奇数帧，即1/3/5/7/9结尾的帧：

```
string(frame.number) ~ "[13579]$"
```



同理，过滤原始seq序列号为奇数并且标志位为SYN的报文可以是：

```
string(tcp.seq_raw) ~ "[13579]$" && tcp.flags.syn==1 && tcp.flags.ack==0
```



匹配目的IP中以255结尾的IP地址(172.16到172.31) :

```
string(ip.dst) matches r"^(172\.(1[6-9]|2[0-9]|3[0-1])\.)\.[0-9]{1,3}\.255"
```

2.5.4 max()、min()函数

函数max()和min()接受相同类型的任意数量的参数，并分别返回集合中最大/最小的参数。

过滤tcp源端口、目的端口，最大不能超过1024的报文：

```
max(tcp.srcport, tcp.dstport) <= 1024
```

过滤tcp源端口+目的端口大于等于1024的报文：

```
min(tcp.srcport+tcp.dstport) >= 1024
```

2.6 字段引用/动态过滤

2.6.1 单变量引用

形式为\${proto.field}的表达式称为字段引用。其值从鼠标选到的当前帧中的相应字段读取。这是一种构建动态过滤器的方法。

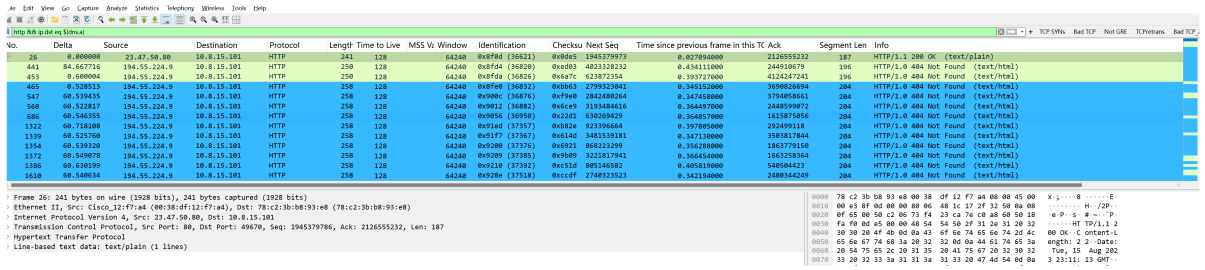
比如下面这个例子，首先我通过过滤语句 dns.a 过滤出大量DNS报文，此时我鼠标选了第一条帧：

```
dns.a # 这条语句含义是过滤dns响应报文中的地址字段
```

第一条帧的DNS response返回的解析记录是：10.85.15.101这个IP地址。

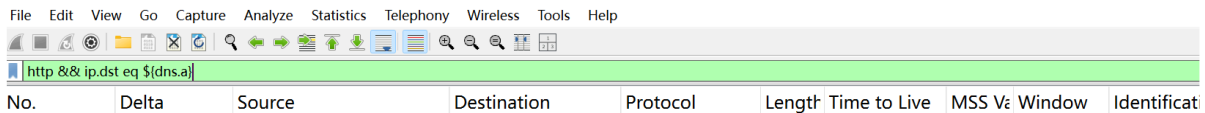
做完上面这个动作，Wireshark实际已经把上面这个IP，赋值引用给了\${dns.a}，此时我们来引用它，过滤http请求，并且目的IP为我们刚刚鼠标点选引用的字段值：

```
http && ip.dst eq ${dns.a}
```



可以看到过滤到了符合要求的HTTP请求。

此时我们再进行回车一次：

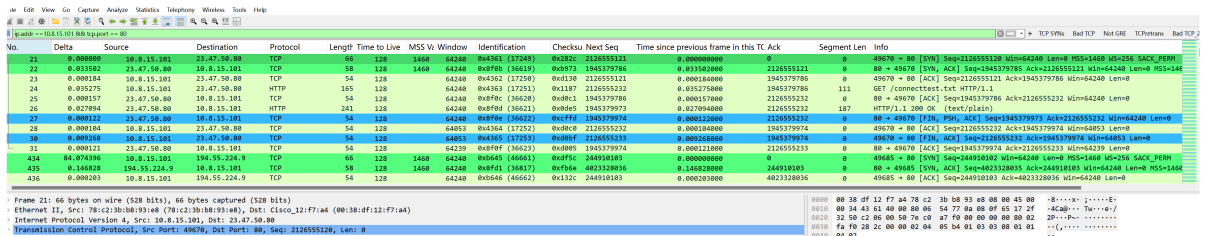


会发现这次返回为空，因为引用一次后\${dns.a}就会被清空，要想继续引用，则用dns.a语句以及鼠标重新选帧。

2.6.2 多变量引用

上面\${}这种写法类似于bash里的变量写法，那么拓展一下，也可以一次性进行多变量的引用，比如同时使用ip.addr和tcp.port两个过滤字段：

```
ip.addr == 10.8.15.101 && tcp.port == 80
```



此时我们鼠标点选一帧。

再进行变量引用，并且只过滤http：

```
ip.addr == ${ip.addr} && tcp.port == ${tcp.port} && http
```

No.	Delta	Source	Destination	Protocol	Length	Time to Live	MSS	Window	Identification	Checksum	Next Seq	Time since previous frame in this TC	Ack	Segment Len	Info
24	0.000000	10.8.15.101	23.47.50.80	HTTP	165	128			64240 0x4363 (37251)	0x1187	212655232	0.035275000	1945379786	111	GET /connecttest.txt HTTP/1.1
26	0.027251	23.47.50.80	10.8.15.101	HTTP	241	128			64240 0x8f0d (36821)	0xb065	1945379973	0.027094000	212655232	187	HTTP/1.1 200 OK (text/plain)
439	84.233549	10.8.15.101	194.55.224.9	HTTP	388	128			64240 0xb648 (46664)	0x39c7	244910679	0.000115000	4023328036	334	POST /Luiz/Five/fre.php HTTP/1.0
441	0.434167	194.55.224.9	10.8.15.101	HTTP	250	128			64240 0x8f4d (36820)	0xad03	4023328232	0.434110000	244910679	196	HTTP/1.0 404 Not Found (text/html)
451	0.206213	10.8.15.101	194.55.224.9	HTTP	278	128			64240 0xb04e (46678)	0x11c1	4124247241	0.000113000	623872158	224	POST /Luiz/Five/fre.php HTTP/1.0
453	0.393791	194.55.224.9	10.8.15.101	HTTP	250	128			64240 0x8f4d (36820)	0x87c7	623872354	0.393727000	4124247241	196	HTTP/1.0 404 Not Found (text/html)
463	0.183295	10.8.15.101	194.55.224.9	HTTP	251	128			64240 0xb654 (46676)	0x1abf	3698826694	0.000146000	2799322836	197	POST /Luiz/Five/fre.php HTTP/1.0
465	0.345218	194.55.224.9	10.8.15.101	HTTP	258	128			64240 0x8f60 (36832)	0xb065	2799323841	0.345152000	3698826694	204	HTTP/1.0 404 Not Found (text/html)
445	60.103921	10.8.15.101	194.55.224.9	HTTP	251	128			64240 0xb65a (46682)	0x5936	3794089661	0.000130000	2384208059	197	POST /Luiz/Five/fre.php HTTP/1.0
547	0.347514	194.55.224.9	10.8.15.101	HTTP	258	128			64240 0x8f60 (36832)	0xf9e8	2442480264	0.347450000	3794089661	204	HTTP/1.0 404 Not Found (text/html)
558	60.158212	10.8.15.101	194.55.224.9	HTTP	251	128			64240 0xb660 (46688)	0xc347	2448599072	0.000203000	3193484411	197	POST /Luiz/Five/fre.php HTTP/1.0
560	0.364405	194.55.224.9	10.8.15.101	HTTP	258	128			64240 0x8f62 (36834)	0xb0ce	3193484616	0.364370000	2448599072	204	HTTP/1.0 404 Not Found (text/html)
604	60.181402	10.8.15.101	194.55.224.9	HTTP	251	128			64240 0xb666 (46694)	0x7938	1615875856	0.000130000	638265224	197	POST /Luiz/Five/fre.php HTTP/1.0

正常拿到了我们想要的的数据。

2.7 搜索过滤协议和字段

点击 **分析(Analyze)** --> **显示过滤表达式(Display Filter Expression)** 可以进入到过滤表达式页面，在这里你可以搜索想要的任何协议或字段，并且鼠标点选你需要的表达式，Wireshark会帮你把完整语句写出来。

比如搜索 `dns.a`，会出来所有全文本匹配 `dns.a` 的协议或字段：

Wireshark - Display Filter Expression

Field Name

- > ALCAP - AAL type 2 signalling protocol (Q.2630)
- > CFLOW - Cisco NetFlow/IPFIX
- > Diameter - Diameter Protocol
- > DNS - Domain Name System
 - dns.a - Address
 - dns.a.ch.addr - Chaos Address
 - dns.a.ch.domain - Chaos Domain
 - dns.a.class.undecoded - Undecoded class
 - dns.a6.address.suffix - Address Suffix
 - dns.a6.prefix.len - Prefix len
 - dns.a6.prefix.name - Prefix name
 - dns.aaaa - AAAA Address
 - dns.afsdb.hostname - Hostname
 - dns.afsdb.subtype - Subtype
 - dns.apl.address.family - Address Family
 - dns.apl.aflength - Address Length
 - dns.apl.aflpart.data - Address
 - dns.apl.aflpart.ipv4 - Address
 - dns.apl.aflpart.ipv6 - Address
 - dns.apl.coded.prefix - Prefix Length
 - dns.apl.negation - Negation Flag
- > ECAT_MAILBOX - EtherCAT Mailbox Protocol
- > ERF - Extensible Record Format
- > GSM SIM - GSM SIM 11.11
- > HICP - Host IP Configuration Protocol
- > LDAP - Lightweight Directory Access Protocol
- > NGAP - NG Application Protocol
- > PPP IPCP - PPP IP Control Protocol
- > RADIUS - RADIUS Protocol
- > SHICP - Secure Host IP Configuration Protocol
- > X509CE - X.509 Certificate Extensions
- > xIRI - X.121 payload

Relation

- is present
- ==
- !=
- ===
- !==
- >
- <
- >=
- <=
- contains

Quantifier

Any All

Value

Predefined Values

Range (offset:length)

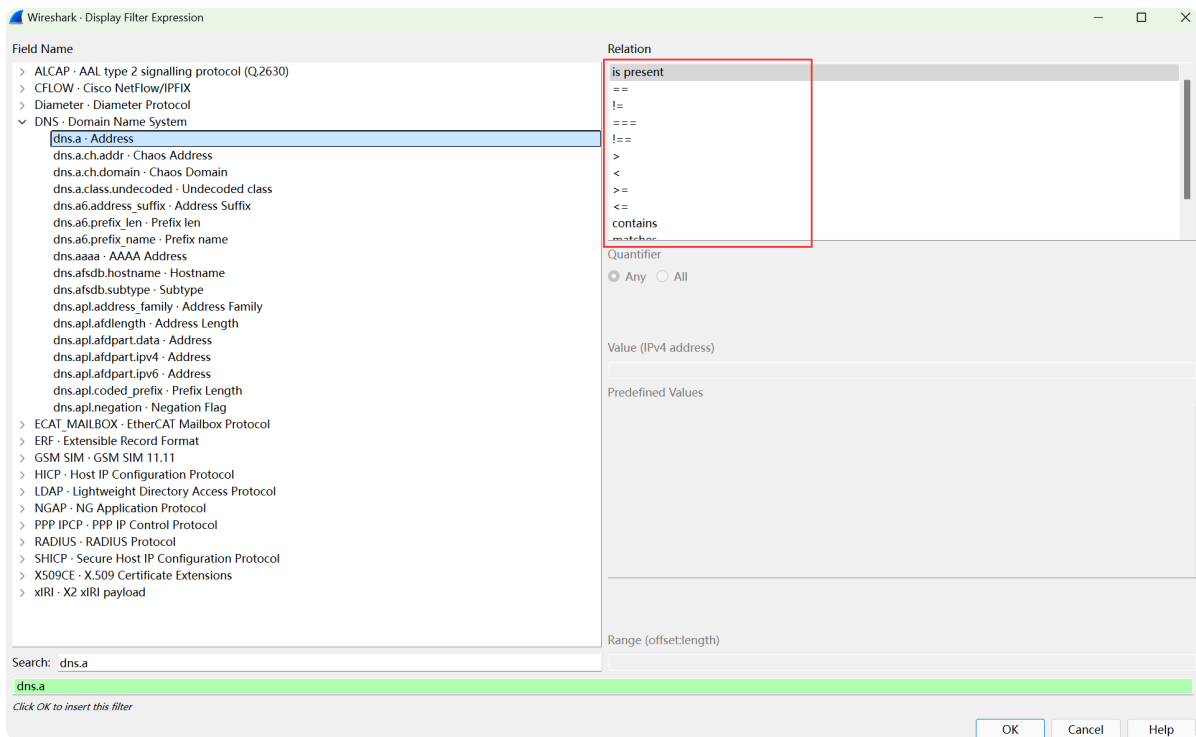
Search: dns.a

No display filter

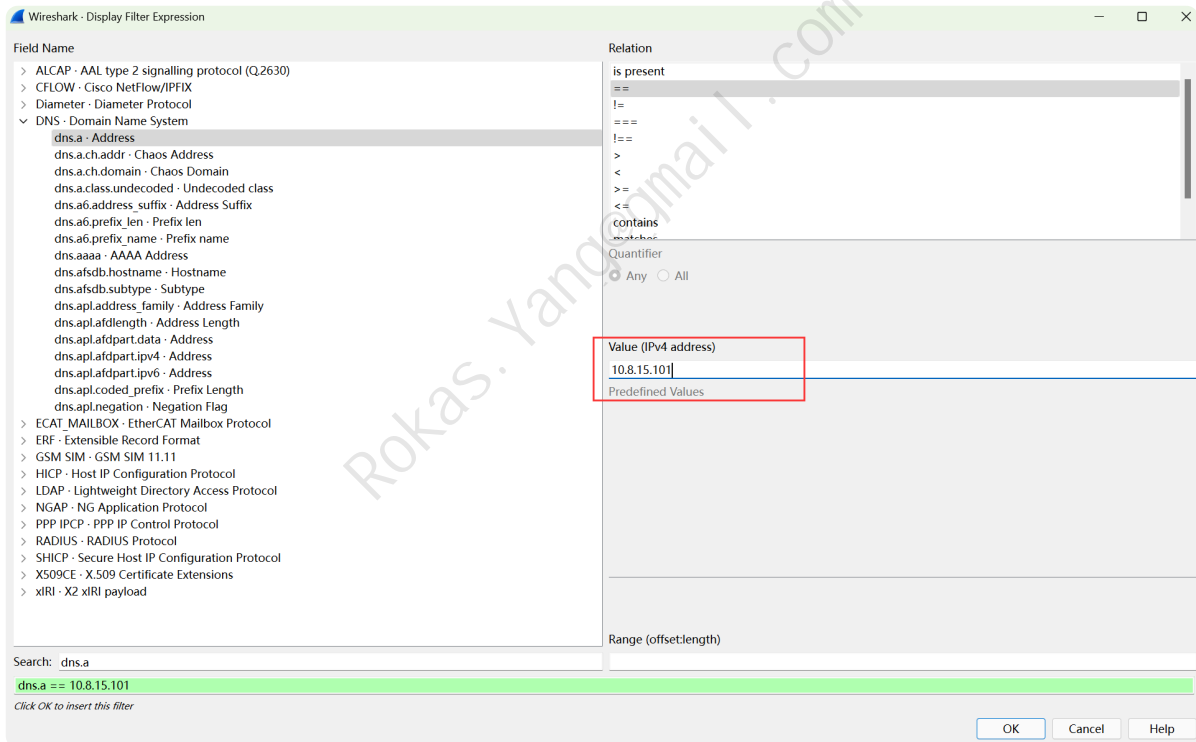
Select a field name to get started

OK Cancel Help

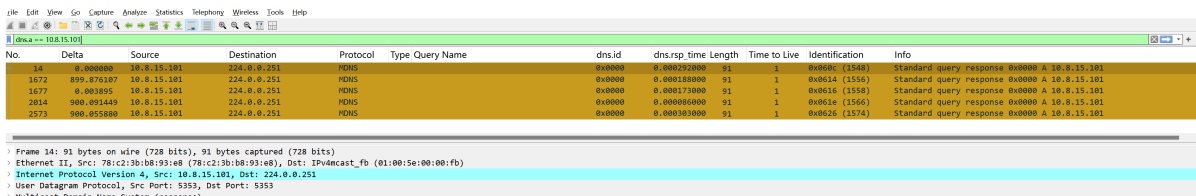
此时我们点选第一个，即 `dns.a`，右上方可以选择关系符：



紧接着，在Value这一段写上要过滤的字段值IP即可：



点OK后，并且回车，将应用此过滤语句：



其它任何协议或字段过滤方法同理。

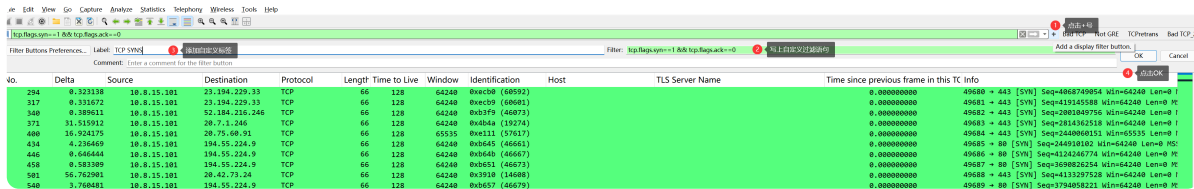
2.8 将过滤语句添加为按钮

比如想过滤TCP第一次握手的SYN包，并且这个功能经常用，需要把这条过滤语句添加为按钮，方便下次鼠标点击时即可应用这条语句，不需要每次手动输入，特别是对于比较长的语句或逻辑比较复杂的语句帮助很大。

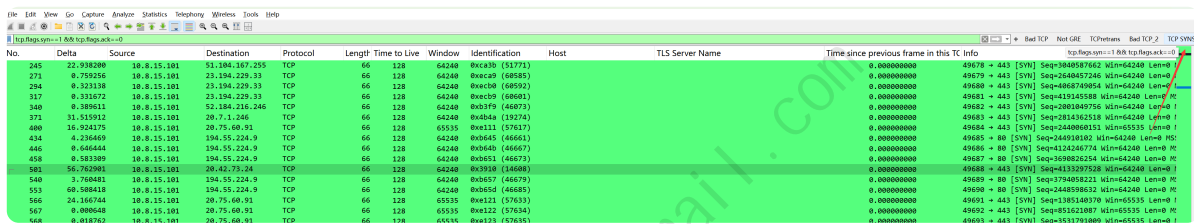
首先明确过滤TCP第一次握手的SYN包对应的过滤语句是：

```
tcp.flags.syn==1 && tcp.flags.ack==0
```

紧接着，按照图示顺序添加自定义过滤按钮：



点击OK后可以看到右边多出来一个我们自定义的标签按钮：TCP SYN：

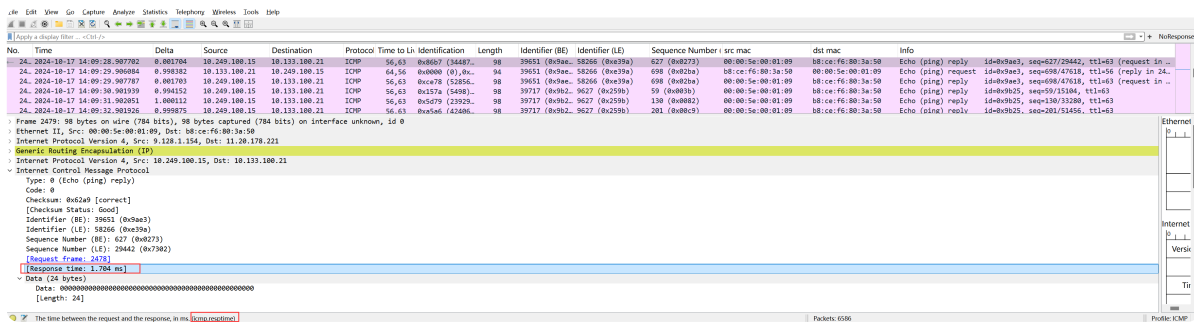


每次点击它，Wireshark会自动应用我们定义的过滤语句。

2.9 将任意字段应用为列

比如ICMP场景，想在打开报文的时候，就能清晰展示每个报文的ping耗时。

首先需要找到ICMP响应耗时的字段，将ICMP Reply报文展开，找到Response time字段：



鼠标右击这个字段，然后右击它 --> 应用为列 (Apply as Column)：

24...	2024-10-17 14:09:42.904869	1.000204	10	Expand Subtrees	56,63	0x
24...	2024-10-17 14:09:43.905288	1.000419	10	Collapse Subtrees	56,63	0x
24...	2024-10-17 14:09:44.905575	1.000287	10	Expand All	56,63	0x
24...	2024-10-17 14:09:45.905363	0.999788	10	Collapse All	56,63	0x
24...	2024-10-17 14:09:46.905523	1.000160	10	Apply as Column	56,63	0x
24...	2024-10-17 14:09:47.905484	0.999961	10	Apply as Filter	56,63	0x
25...	2024-10-17 14:09:48.905572	1.000088	10	Prepare as Filter	56,63	0x
25...	2024-10-17 14:09:49.905711	1.000139	10	Conversation Filter	56,63	0x

```

Frame 2479: 98 bytes on wire (784 bits), 98 byte
: Ethernet II, Src: 00:00:5e:00:01:09, Dst: b8:ce:
: Internet Protocol Version 4, Src: 9.128.1.154, D
: Generic Routing Encapsulation (IP)
: Internet Protocol Version 4, Src: 10.249.100.15,
: Internet Control Message Protocol
  Type: 0 (Echo (ping) reply)
  Code: 0
  Checksum: 0x62a9 [correct]
  [Checksum Status: Good]
  Identifier (BE): 39651 (0x9ae3)
  Identifier (LE): 58266 (0xe39a)
  Sequence Number (BE): 627 (0x0273)
  Sequence Number (LE): 29442 (0x7302)
  [Request frame: 2478]
  [Response time: 1.704 ms]

```

- Expand Subtrees
- Collapse Subtrees
- Expand All
- Collapse All
- Apply as Column **Ctrl+Shift+I**
- Apply as Filter
- Prepare as Filter
- Conversation Filter
- Colorize with Filter
- Follow
- I/O Graph
- Copy
- Show Packet Bytes... **Ctrl+Shift+O**
- Export Packet Bytes... **Ctrl+Shift+X**
- Wiki Protocol Page
- Filter Field Reference
- Protocol Preferences
- Decode As... **Ctrl+Shift+U**
- Go to Linked Packet
- Show Linked Packet in New Window

✓ Data (24 bytes)
 Data: 00
 [Length: 24]

之后可以看到报文的最右边多出一列：Response time，我们把它拖动到中间显眼部分，这样每个ICMP Reply响应包都展示了各自的耗时：

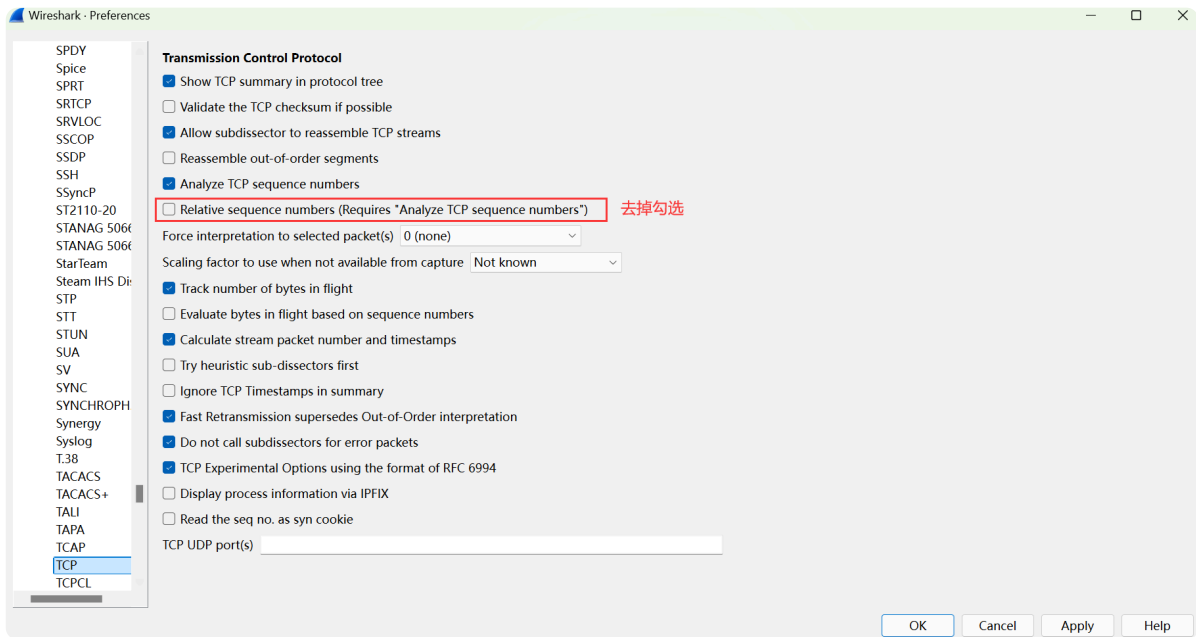
No.	Time	Delta	Source	Destination	Protocol	Time to Live	Identification	Length	Identifier (BE)	Identifier (LE)	Sequence Number (BE)	Response time	src mac	dst mac	Info
24	2024-10-17 14:08:31.98534	0.998875	10.133.100.21	10.249.100.15	ICMP	64,56	0x0000 (0),0x...	94	39317 (0x9999,38297 (0x9599))	138 (0x0882)			b8:ce:fe:80:3a:50	00:00:5e:00:01:09	Echo (ping) request
24	2024-10-17 14:08:31.987246	0.000172	10.249.100.15	10.133.100.21	ICMP	56,63	0x0000 (0),0x...	98	39317 (0x9999,38297 (0x9599))	139 (0x0883)	1.712		b8:ce:fe:80:3a:50	00:00:5e:00:01:09	Echo (ping) reply
24	2024-10-17 14:08:32.985498	0.999252	10.133.100.21	10.249.100.15	ICMP	64,56	0x0000 (0),0x...	94	39317 (0x9999,38297 (0x9599))	201 (0x08c5)			00:00:5e:00:01:09	b8:ce:fe:80:3a:50	Echo (ping) request
24	2024-10-17 14:08:32.987068	0.001570	10.249.100.15	10.133.100.21	ICMP	56,63	0x277f (10411)	98	39317 (0x9999,38297 (0x9599))	201 (0x08c5)	1.57		00:00:5e:00:01:09	b8:ce:fe:80:3a:50	Echo (ping) reply
24	2024-10-17 14:08:33.987360	0.999692	10.133.100.21	10.249.100.15	ICMP	64,56	0x0000 (0),0x...	94	39317 (0x9999,38297 (0x9599))	272 (0x0b18)			b8:ce:fe:80:3a:50	00:00:5e:00:01:09	Echo (ping) request
24	2024-10-17 14:08:33.987339	0.000379	10.249.100.15	10.133.100.21	ICMP	56,63	0x714f (29087)	98	39317 (0x9999,38297 (0x9599))	272 (0x0b18)	1.57		00:00:5e:00:01:09	b8:ce:fe:80:3a:50	Echo (ping) reply
24	2024-10-17 14:08:34.987671	0.998431	10.133.100.21	10.249.100.15	ICMP	64,56	0x0000 (0),0x...	94	39317 (0x9999,38297 (0x9599))	343 (0x0b17)			b8:ce:fe:80:3a:50	00:00:5e:00:01:09	Echo (ping) request
24	2024-10-17 14:08:34.987663	0.000792	10.249.100.15	10.133.100.21	ICMP	56,63	0x004d (47883)	98	39317 (0x9999,38297 (0x9599))	343 (0x0b17)	1.702		00:00:5e:00:01:09	b8:ce:fe:80:3a:50	Echo (ping) reply
24	2024-10-17 14:08:35.985881	0.998418	10.133.100.21	10.249.100.15	ICMP	64,56	0x0000 (0),0x...	94	39317 (0x9999,38297 (0x9599))	414 (0x013e)			00:00:5e:00:01:09	b8:ce:fe:80:3a:50	Echo (ping) request
24	2024-10-17 14:08:35.987446	0.001565	10.249.100.15	10.133.100.21	ICMP	56,63	0x0387 (775)...	98	39317 (0x9999,38297 (0x9599))	414 (0x013e)	1.565		00:00:5e:00:01:09	b8:ce:fe:80:3a:50	Echo (ping) reply
24	2024-10-17 14:08:36.985904	1.000058	10.133.100.21	10.249.100.15	ICMP	64,56	0x0000 (0),0x...	94	39317 (0x9999,38297 (0x9599))	485 (0x01a5)			b8:ce:fe:80:3a:50	00:00:5e:00:01:09	Echo (ping) request
24	2024-10-17 14:08:36.918076	0.801572	10.249.100.15	10.133.100.21	TCP	64,61	0x6a27/30014	94	39317 (0x9999,38297 (0x9599))	485 (0x01a5)	1.572		00:00:5e:00:01:09	b8:ce:fe:80:3a:50	Echo (ping) request

这里不再举例，任何协议的字段都能使用此方法应用为列，让Wireshark更直观的展示我们想要关注的重点字段。

2.10 修改相对seq为原始seq

原始序列号（字段：tcp.seq_raw）也叫绝对序列号，默认情况下，Wireshark展示的序列号都是以每一条TCP流为维度的相对序列号（比如第一次握手SYN的seq从0开始计算），而实际交互中的seq序列号都是原始序列号，不会从0开始，所以使用相对序列号并不利于点到点、端到端或者全链路对于同一条TCP流的分析。

此时可以在编辑 (Edit) --> 首选项 (Preferences) --> 协议 (Protocols) --> TCP 页面中，取消勾选相对序列号 (Relative sequence numbers)：



之后点击应用，可以看到展示的seq均为原始seq，后续的对流分析可以通过原始seq进行过滤匹配。

No.	Delta	Source	Destination	Protocol	Length	Time to Live	Window	Identification	Status Code	Seq(raw)	Time since previous frame in this TCP stream	Info
21	0.000000	10.8.15.101	23.47.50.80	TCP	66	128	64240	0x4361 (17249)		212655120	0.000000000	49670 → 80 [SYN] Seq=212655120 Win=64240 Len=0 MSS=1460 WS=2
22	0.033562	23.47.50.80	10.8.15.101	TCP	54	128	64240	0x8F80 (36639)		1945379785	0.033562000	80 → 49670 [SYN, ACK] Seq=1945379785 Ack=212655121 Win=64240
23	0.000184	10.8.15.101	23.47.50.80	TCP	54	128	64240	0x4362 (17250)		212655121	0.000184000	49670 → 80 [ACK] Seq=212655121 Ack=1945379786 Win=64240 Len=0
24	0.035275	10.8.15.101	23.47.50.80	HTTP	165	128	64240	0x4363 (17251)		212655121	0.035275000	GET /connecttest.txt HTTP/1.1
25	0.000157	23.47.50.80	10.8.15.101	TCP	54	128	64240	0x8F80 (36620)		1945379786	0.000157000	80 → 49670 [ACK] Seq=1945379786 Ack=212655122 Win=64240 Len=0
26	0.027094	23.47.50.80	10.8.15.101	HTTP	241	128	64240	0x8F80 (36621)	200	1945379786	0.027094000	HTTP/1.1 200 OK (text/plain)
27	0.000122	23.47.50.80	10.8.15.101	TCP	54	128	64240	0x8F80 (36622)		1945379787	0.000122000	80 → 49670 [FIN, PSH, ACK] Seq=1945379787 Ack=212655122 Win=0
28	0.000104	10.8.15.101	23.47.50.80	TCP	54	128	64085	0x4364 (17252)		212655122	0.000104000	49670 → 80 [ACK] Seq=212655122 Ack=1945379787 Win=64085 Len=0
29	0.000268	10.8.15.101	23.47.50.80	TCP	54	128	64085	0x4365 (17253)		212655122	0.000268000	49670 → 80 [FIN, ACK] Seq=212655122 Ack=1945379786 Win=64085
31	0.000121	23.47.50.80	10.8.15.101	TCP	54	128	64239	0x8F8F (36623)		1945379974	0.000121000	80 → 49670 [ACK] Seq=1945379974 Ack=212655123 Win=64239 Len=0
42	2.534000	10.8.15.101	13.89.179.8	TCP	66	128	64240	0x2861 (10449)		1375947579	0.000000000	49673 → 443 [SYN] Seq=1375947579 Win=64240 Len=0 MSS=1460 WS=2
43	0.000164	13.89.179.8	10.8.15.101	TCP	58	128	64240	0x8F11 (36625)		3595289444	0.000164000	443 → 49673 [SYN, ACK] Seq=3595289444 Ack=1375947580 Win=64240
44	0.000187	10.8.15.101	13.89.179.8	TCP	54	128	64240	0x2862 (10450)		1375947580	0.000187000	49673 → 443 [ACK] Seq=1375947580 Ack=3595289445 Win=64240 Len=0
46	0.000514	10.8.15.101	13.89.179.8	TLSv1.2	266	128	64240	0x2863 (10451)		1375947580	0.000514000	Client Hello (SSLv3/TLSv1.2) [Application Data]
47	0.000164	13.89.179.8	10.8.15.101	TCP	58	128	64240	0x8F12 (36626)		3595289445	0.000164000	443 → 49673 [ACK] Seq=3595289445 Ack=1375947580 Win=64240 Len=0
49	0.017983	10.8.15.101	20.71.1.246	TCP	66	128	64240	0x4330 (15255)		616795611	0.000000000	49674 → 443 [SYN] Seq=616795611 Win=64240 Len=0 MSS=1460 WS=2
50	0.000081	20.71.1.246	10.8.15.101	TCP	54	128	64240	0x8F14 (36628)		3747292171	0.000081000	443 → 49674 [ACK] Seq=3747292171 Ack=616795611 Win=64240 Len=0
51	0.000000	10.8.15.101	20.71.1.246	TCP	54	128	0	0x433C (15260)		3060972161	0.000000000	40674 → 443 [RST] Seq=3060972161 Win=0 Len=0

2.11 Wireshark的箭头符号

Wireshark会在帧的最左边，用箭头符号标记请求帧 (→) 和响应帧 (←)：

No.	Delta	Source	Destination	Protocol	Length	Time to Live	Window	Identification	Segment Len	Seq(raw)	Ack	Time since previous frame in this TCP stream	Info
21	0.000000	10.8.15.101	23.47.50.80	TCP	66	128	64240	0x4361 (17249)	0	212655120	0	0.000000000	49670 → 80 [SYN] Seq=212655120 Win=64240 Len=0 MSS=1460 WS=2
22	0.033562	23.47.50.80	10.8.15.101	TCP	58	128	64240	0x8F80 (36639)	0	1945379785	212655121	0.033562000	80 → 49670 [SYN, ACK] Seq=1945379785 Ack=212655121 Win=64240
23	0.000184	10.8.15.101	23.47.50.80	TCP	54	128	64240	0x4362 (17250)	111	212655121	1945379786	0.000184000	49670 → 80 [ACK] Seq=212655121 Ack=1945379786 Win=64240 Len=0
24	0.035275	10.8.15.101	23.47.50.80	HTTP	165	128	64240	0x4363 (17251)	111	212655121	1945379786	0.035275000	GET /connecttest.txt HTTP/1.1
25	0.000157	23.47.50.80	10.8.15.101	TCP	54	128	64240	0x8F80 (36620)	0	1945379786	212655122	0.000157000	80 → 49670 [ACK] Seq=1945379786 Ack=212655122 Win=64240 Len=0
26	0.027094	23.47.50.80	10.8.15.101	HTTP	241	128	64240	0x8F80 (36621)	187	1945379786	212655122	0.027094000	HTTP/1.1 200 OK (text/plain)
27	0.000122	23.47.50.80	10.8.15.101	TCP	54	128	64240	0x8F80 (36622)	0	1945379973	212655122	0.000122000	80 → 49670 [FIN, PSH, ACK] Seq=1945379973 Ack=212655122 Win=0
28	0.000104	10.8.15.101	23.47.50.80	TCP	54	128	64085	0x4364 (17252)	0	212655122	1945379974	0.000104000	49670 → 80 [ACK] Seq=212655122 Ack=1945379974 Win=64085 Len=0
29	0.000268	10.8.15.101	23.47.50.80	TCP	54	128	64085	0x4365 (17253)	0	212655122	1945379974	0.000268000	49670 → 80 [FIN, ACK] Seq=212655122 Ack=1945379974 Win=64085
31	0.000121	23.47.50.80	10.8.15.101	TCP	54	128	64239	0x8F8F (36623)	0	1945379974	212655123	0.000121000	80 → 49670 [ACK] Seq=1945379974 Ack=212655123 Win=64239 Len=0

对于帧长度太大导致被截断的报文，看不出哪个是请求或响应帧特别有用，比如下面的示例：

No.	Delta	Source	Destination	Protocol	Length	Time to Live	Window	Identification	Segment Len	Seq(raw)	Ack	Time since previous frame in this TCP stream	Info
1	0.000000	192.168.1.1	10.0.0.1	TCP	66	128	8152	0x7C9F (31989)	0	2984488208	0	0.000000000	61300 → 80 [SYN] Seq=2984488208 Win=8192 Len=0 MSS=1460 WS=256 SACK
2	0.000187	10.0.0.1	192.168.1.1	TCP	60	128	64148	0x4366 (15866)	0	1566413354	2984488241	0.000187000	80 → 61300 [SYN, ACK] Seq=1566413354 Ack=2984488241 Win=64148 Len=0
3	0.000664	192.168.1.1	10.0.0.1	TCP	60	128	64	0x7F6B (53147)	0	2984488241	1566413355	0.000664000	61300 → 80 [ACK] Seq=2984488241 Ack=1566413355 Win=16384 Len=0
4	0.000853	192.168.1.1	10.0.0.1	HTTP	1514	128	64	0x0000 (20480)	1460	2984488241	1566413355	0.000853000	Continuation[Packet size limited during capture]
5	0.000000	192.168.1.1	10.0.0.1	HTTP	89	128	64	0x400B (20155)	35	2984489701	1566413355	0.000000000	Continuation
6	0.186550	10.0.0.1	192.168.1.1	TCP	60	111	64240	0x0204 (740)	0	1566413355	2984489736	0.186550000	80 → 61300 [ACK] Seq=1566413355 Ack=2984489736 Win=64240 Len=0
7	0.000100	192.168.1.1	10.0.0.1	TCP	60	128	64	0x070E (52864)	1	2984489735	1566413355	0.000100000	[TCP Keep-Alive] 61300 → 80 [ACK] Seq=2984489735 Ack=1566413355 Win=0
8	0.000445	10.0.0.1	192.168.1.1	TCP	60	111	64240	0x090F (38335)	0	1566413355	2984489736	0.000445000	[TCP Keep-Alive] 61300 → 61300 [ACK] Seq=1566413355 Ack=2984489736
9	0.000475	10.0.0.1	192.168.1.1	TCP	60	128	64	0x070C (52864)	1	2984489735	1566413355	0.000475000	[TCP Keep-Alive] 61300 → 80 [ACK] Seq=2984489735 Ack=1566413355 Win=0
10	0.000475	10.0.0.1	192.168.1.1	TCP	60	111	64240	0x090E (38334)	0	1566413355	2984489736	0.000475000	[TCP Keep-Alive] 61300 → 61300 [ACK] Seq=1566413355 Ack=2984489736
11	18.454900	10.0.0.1	192.168.1.1	HTTP	572	111	64240	0x0C0F (52723)	518	1566413355	2984489736	18.454900000	HTTP/1.1 200 OK [Packet size limited during capture]
12	0.001620	192.168.1.1	10.0.0.1	TCP	60	128	62	0x050F (19355)	0	2984489735	1566413355	0.001620000	61300 → 80 [ACK] Seq=2984489735 Ack=1566413355 Win=16384 Len=0
13	0.001097	10.0.0.1	192.168.1.1	HTTP	640	111	64240	0x0000 (14400)	1460	1566413357	2984489736	0.001097000	Continuation[Packet size limited during capture]
14	0.000721	192.168.1.1	10.0.0.1	TCP	60	128	64	0x050E (19354)	0	2984489736	1566413355	0.000721000	61300 → 80 [ACK] Seq=2984489736 Ack=1566413355 Win=16384 Len=0
15	0.002599	10.0.0.1	192.168.1.1	HTTP	1514	111	64240	0x0C0F (52825)	1460	1566413359	2984489736	0.002599000	Continuation[Packet size limited during capture]
16	0.000178	10.0.0.1	192.168.1.1	TCP	1514	111	64240	0x050F (19355)	1460	1566413359	2984489736	0.000178000	Continuation

如果不看箭头符号，并不好直观看出哪个帧是HTTP请求，而Wireshark则默认标记出来了，第4帧是请求，第11帧是响应，前提是鼠标要选择请求帧或响应帧，Wireshark才会给你完整标记出请求帧和响应帧。

DNS、ICMP也是同理，比如鼠标选中其中一个Dns query，Wireshark会同时把所选中的query对应的response用箭头符号都标记出来。

No.	Delta	Source	Destination	Protocol	Type	Query Name	dns.id	dns.rsp.time	Length	Time to Live	Identification	Info
19	0.000000	10.8.15.101	10.8.15.1	DNS	A	www.microsoft.com	0x630e		81	128	0x0005 (56789)	Standard query 0x630e A www.microsoft.com
20	0.022121	10.8.15.1	10.8.15.101	DNS	A	www.microsoft.com	0x630e	0.022121000	227	128	0x000a (36618)	Standard query response 0x630e A www.microsoft.com CHAME ncsi.geo.trafficmanager.net

三、常见应用场景的过滤方法

3.1 过滤HTTP/HTTPS/TLS请求域名

使用如下过滤语句筛选HTTP请求的请求HOST，以及TLS握手的client hello阶段的请求主机名：

```
tls.handshake.type == 1 || http.request
```

No.	Delta	Source	Destination	Protocol	Length	Time to Live	Window	Identification	Host	TLS Server Name	Time since previous frame in this TCP info
24	0.000000	10.8.15.101	23.67.58.90	HTTP	235	128	64240	0x2803 (12751)	www.microsoft.com	v2b.events.data.microsoft.com	0.005518000
46	2.635333	10.8.15.101	13.89.179.8	TLSv1.2	266	128	64240	0x2803 (10481)		client.ums.windows.com	0.002548000

HTTP请求域名在http.host字段内，TLS/HTTPS请求域名在client hello阶段的SN扩展字段内有展示，因此如上图，可以把这两个字段应用为列，可以更清晰看到HTTP/HTTPS/TLS报文携带的请求域名。

3.2 过滤HTTP状态码或状态码范围

我们比较想知道HTTP响应4xx、5xx状态码报文，可以这么做：

```
string(http.response.code) ~ "^[40-9]{2}$|^5[0-9]{2}$"
```

No.	Delta	Source	Destination	Protocol	Length	Time to Live	Window	Identification	Status Code	Time since previous frame in this TCP stream	Info
441	0.000000	194.55.224.9	10.8.15.101	HTTP	250	128	64240	0x8f04 (36820)	404	0.434118000	HTTP/1.0 404 Not Found (text/html)
453	0.690004	194.55.224.9	10.8.15.101	HTTP	250	128	64240	0x8f04 (36826)	404	0.393727000	HTTP/1.0 404 Not Found (text/html)
465	0.528513	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x900c (36832)	404	0.345152000	HTTP/1.0 404 Not Found (text/html)

首先使用string()函数将HTTP状态码字段转换为字符串类型，之后使用PCRE正则匹配4xx和5xx状态码。如果只想限定查找几个固定的状态码，那就更简单了：

```
http.response.code in {403, 404, 502, 503, 504}
```

No.	Delta	Source	Destination	Protocol	Length	Time to Live	Window	Identification	Status Code	Time since previous frame in this TCP stream	Info
441	0.000000	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x8fd4 (36820)	404	0.43411000	HTTP/1.0 404 Not Found (text/html)
453	0.600004	194.55.224.9	10.8.15.101	HTTP	250	128	64240	0x8fd4 (36826)	404	0.393727000	HTTP/1.0 404 Not Found (text/html)
465	0.528513	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x8f80 (36832)	404	0.345152000	HTTP/1.0 404 Not Found (text/html)
547	0.539435	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9006 (36876)	404	0.347458000	HTTP/1.0 404 Not Found (text/html)
560	0.522817	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9012 (36882)	404	0.364497000	HTTP/1.0 404 Not Found (text/html)
686	0.546355	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9095 (36950)	404	0.364857000	HTTP/1.0 404 Not Found (text/html)
1322	0.718108	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x91e1 (37357)	404	0.397805000	HTTP/1.0 404 Not Found (text/html)
1339	0.525760	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x91f7 (37367)	404	0.347130000	HTTP/1.0 404 Not Found (text/html)
1354	0.539320	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9200 (37376)	404	0.356280000	HTTP/1.0 404 Not Found (text/html)
1372	0.549078	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9200 (37385)	404	0.366454000	HTTP/1.0 404 Not Found (text/html)
1386	0.630199	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9210 (37392)	404	0.405819000	HTTP/1.0 404 Not Found (text/html)
1610	0.540834	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x928e (37518)	404	0.342294000	HTTP/1.0 404 Not Found (text/html)
1623	0.542761	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x9297 (37527)	404	0.355199000	HTTP/1.0 404 Not Found (text/html)
1639	0.526086	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92ba (37546)	404	0.357391000	HTTP/1.0 404 Not Found (text/html)
1653	0.556896	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92b1 (37553)	404	0.362852000	HTTP/1.0 404 Not Found (text/html)
1665	0.579698	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92b8 (37560)	404	0.389308000	HTTP/1.0 404 Not Found (text/html)
1687	0.528149	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92b8 (37566)	404	0.346907000	HTTP/1.0 404 Not Found (text/html)
1704	0.515745	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92c7 (37575)	404	0.342065000	HTTP/1.0 404 Not Found (text/html)
1720	0.527944	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92c7 (37583)	404	0.341640000	HTTP/1.0 404 Not Found (text/html)
1766	0.568641	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92e9 (37609)	404	0.361945000	HTTP/1.0 404 Not Found (text/html)
1778	0.542839	194.55.224.9	10.8.15.101	HTTP	258	128	64240	0x92f8 (37624)	404	0.367181000	HTTP/1.0 404 Not Found (text/html)

如果是HTTPS/TLS加密后的报文想过滤状态码，使用此方法则不行，因为数据已经被加密了在tls握手后看不到任何明文字段，除非解密后去过滤对应字段，如何解密可以参考 [这篇文章](#)。

3.3 过滤TCP握手失败的SYN报文

过滤第一次握手失败的请求，并且没有拿到SYN-ACK响应的，可以是：

```
tcp.completeness.syn==1&tcp.completeness.syn-ack==0&tcp.completeness.ack==0&tcp.completeness.data==0&tcp.completeness.fin==0
```

No.	Time	Delta	Source	Destination	Protocol	Length	MSS Value	Window	Identification	Checksum	Time to	Time since previous frame in this TCP stream	Source or I
1	0.000000	0.000000	192.168.1.14	119.29.29.29	TCP	54	1480	0x5f4c	(...)	0x0dc10	64	0.000000000	
2	1.000418	1.000418	192.168.1.14	119.29.29.29	TCP	54	1480	0x5f4c	(...)	0x0dc10	64	1.000418000	
3	2.001786	1.001368	192.168.1.14	119.29.29.29	TCP	54	1480	0x5f4c	(...)	0x0dc10	64	1.001368000	
4	3.003213	1.001427	192.168.1.14	119.29.29.29	TCP	54	1480	0x5f4c	(...)	0x0dc10	64	1.001427000	
5	4.005475	1.002262	192.168.1.14	119.29.29.29	TCP	54	1480	0x5f4c	(...)	0x0dc10	64	1.002262000	

tcp.completeness.syn-ack这条语句，用于过滤在一个tcp连接中的标志位中是否涵盖syn-ack的报文，其它标志位同理，[官方文档](#)的解释如下：

TCP Conversation Completeness

TCP conversations are said to be complete when they have both opening and closing handshakes, independently of any data transfer. However, we might be interested in identifying complete conversations with some data sent, and we are using the following bit values to build a filter value on the tcp.completeness field :

- 1 : SYN
- 2 : SYN-ACK
- 4 : ACK
- 8 : DATA
- 16 : FIN

可以看到下面的完整报文，实际都被RST-ACK拒绝连接了，因为目的服务器端口没有监听：

No.	Delta	Source	Destination	Protocol	Length	Time to Live	Window	Identification	Status Code	Time since previous frame in this TC	Info
1	0.000000	192.168.1.14	192.168.1.8	TCP	54	64	1480	0x0000 (0)		0.000000000	43496 → 443 [SYN] Seq=333275029 Win=1480 Len=0
2	0.001067	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)		0.001067000	443 → 43496 [RST, ACK] Seq=333275030 Win=0 Len=0
3	0.002066	192.168.1.14	192.168.1.8	TCP	54	64	1480	0x0000 (0)		0.002066000	[TCP Port numbers reused] 43496 → 443 [SYN] Seq=333275029 Win=1480 Len=0
4	0.001333	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)		0.001333000	443 → 43496 [RST, ACK] Seq=0 Ack=333275030 Win=0 Len=0
5	1.000453	192.168.1.14	192.168.1.8	TCP	54	64	1480	0x0000 (0)		0.000000000	[TCP Port numbers reused] 43496 → 443 [SYN] Seq=333275029 Win=1480 Len=0
6	0.001265	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)		0.001265000	443 → 43496 [RST, ACK] Seq=0 Ack=333275030 Win=0 Len=0
7	1.000441	192.168.1.14	192.168.1.8	TCP	54	64	1480	0x0000 (0)		0.000000000	[TCP Port numbers reused] 43496 → 443 [SYN] Seq=333275029 Win=1480 Len=0
8	0.001233	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)		0.001233000	443 → 43496 [RST, ACK] Seq=0 Ack=333275030 Win=0 Len=0
9	1.000383	192.168.1.14	192.168.1.8	TCP	54	64	1480	0x0000 (0)		0.000000000	[TCP Port numbers reused] 43496 → 443 [SYN] Seq=333275029 Win=1480 Len=0
10	0.000212	192.168.1.8	192.168.1.14	TCP	60	64	0	0x0000 (0)		0.000212000	443 → 43496 [RST, ACK] Seq=0 Ack=333275030 Win=0 Len=0

而如果我们只是简单粗暴的过滤syn为1并且ack等于0的报文：

```
tcp.flags.syn==1 && tcp.flags.ack==0
```

这个逻辑仅仅是以帧为维度，只要满足SYN标志位为1，并且ACK标志为0的报文，都会展示出来，没有TCP流/TCP会话的概念，某些场景可能也能过滤出来，但更推荐tcp.completeness.xxx语句，更加精准的命中需求。

3.4 过滤超时/丢包重传的报文

过滤TCP超时或丢包重传的报文的可以是：

```
tcp.analysis.retransmission
```

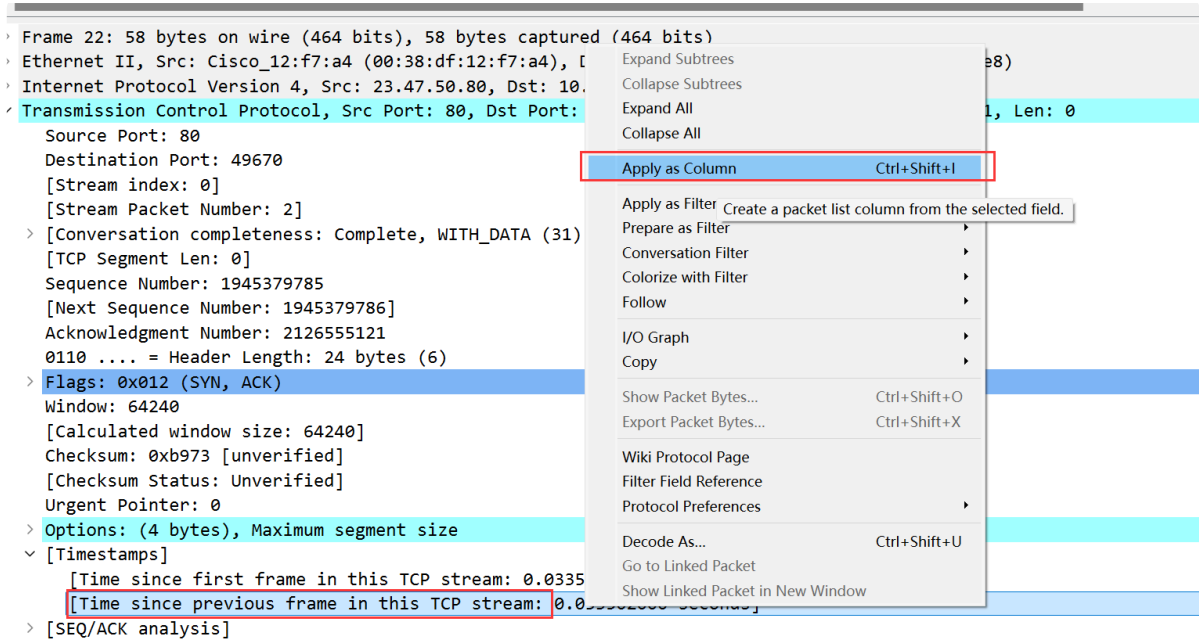
No.	Date	Time	Delta	Source	Destination	Protocol	Length	Time to Live	MSS Vs Window	Identification	Checksum	APP	Seq(raw)	Next Seq	Time since previous frame in this TC	ACK
2	2024-10-20 08:11:03.745717	1.000458	0.000000	192.168.1.14	119.29.29.29	TCP	54	64	1480	0x5F4c (24396)	0xdc10		2593727778	2593727771	1.000418000	0
3	2024-10-20 08:11:04.074705	2.001296	1.001568	192.168.1.14	119.29.29.29	TCP	54	64	1480	0x5F4c (24396)	0xdc10		2593727778	2593727771	1.001568000	0
4	2024-10-20 08:11:05.748512	3.003213	1.901427	192.168.1.14	119.29.29.29	TCP	54	64	1480	0x5F4c (24396)	0xdc10		2593727778	2593727771	1.001427000	0
5	2024-10-20 08:11:06.750774	4.005475	1.002262	192.168.1.14	119.29.29.29	TCP	54	64	1480	0x5F4c (24396)	0xdc10		2593727778	2593727771	1.002262000	0

也可以再加一个限定条件，过滤TCP重传并且时间差大于等于0.2s的报文：

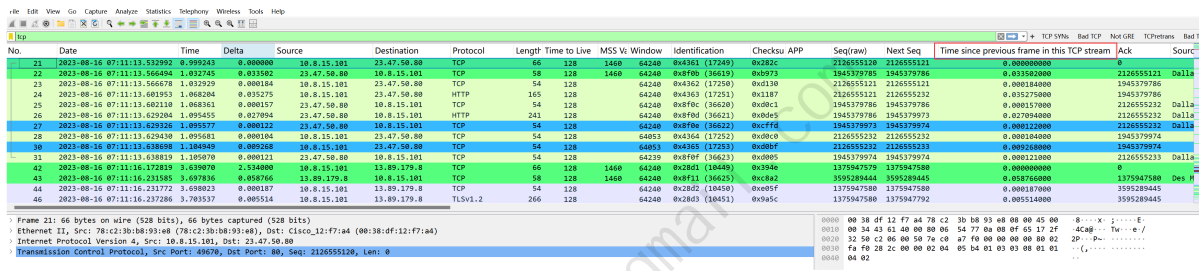
```
tcp.analysis.retransmission && tcp.time_delta >= 0.2
```

3.5 在同一个TCP流中的帧按间隔时间排序

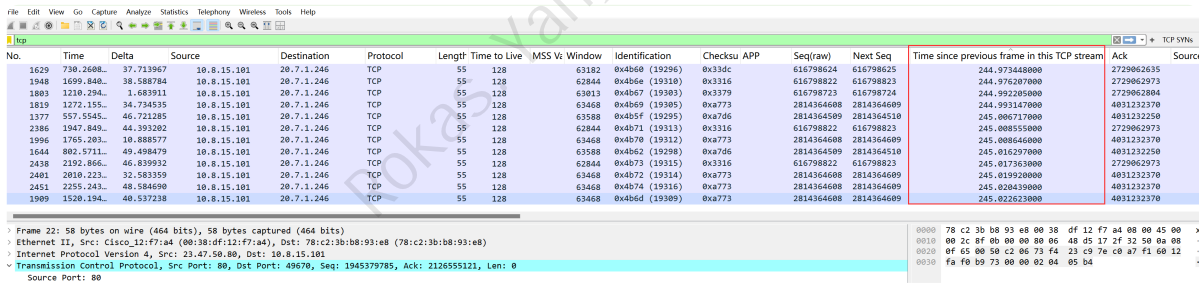
首先点选任意一个TCP帧，找到时间戳字段，右击"Time since previous frame in this TCP stream"字段（含义为在TCP流中相对上一帧的时间） --> 应用为列（Apply as Column）：



之后把这个字段拖动到视野能及的地方：

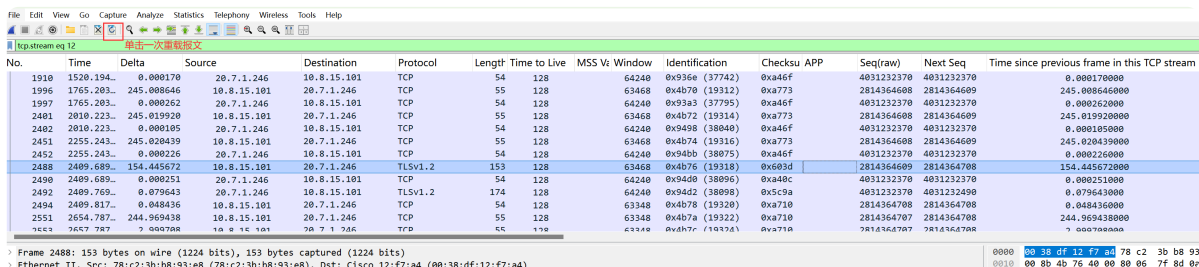


单击此字段会按照由下之上从大到小排列报文，单位为秒：



再单击一下，则从小到大排列报文。

找到耗时异常的帧并右击追踪流后，别忘了将报文重载，让它恢复报文顺序：



不然还是会将这个字段进行大小排序。

如果想要过滤这个字段大于某个时间间隔的报文，比如帧间隔大于100秒的可以是：



No.	Time	Delta	Source	Destination	Protocol	Length	Time to Live	MSS W/Window	Identification	Checksum	APP	Seq(raw)	Next Seq	Time since previous frame in this TCP stream	Ack	Source Geof
1989	1520.194..	65.452941	10.8.15.101	20.7.1.246	TCP	55	128	63468	0xb66d (19309)	0xa773		2814364608	2814364609	245.02262000		4031232370
1972	1524.025..	0.811248	20.7.1.246	10.8.15.101	TCP	54	128	62420	0xb378 (3712)	0xc5c5		616798822	616798823	244.97200700		2729082973
1944	1529.840..	115.814085	10.8.15.101	20.7.1.246	TCP	55	128	62844	0xb46e (19310)	0x331e		616798822	616798823	245.00865000		4031232370
1996	1765.203..	65.382763	10.8.15.101	20.7.1.246	TCP	55	128	63468	0xb478 (19312)	0x331e		2814364608	2814364609	245.00865000		4031232370
2386	1947.849..	182.646128	10.8.15.101	20.7.1.246	TCP	55	128	62844	0xb471 (19313)	0x331e		616798822	616798823	245.00865000		2729082973
2440	1977.639..	27.970969	20.7.1.246	10.8.15.101	TCP	54	128	62420	0xb378 (3712)	0xc5c5		616798822	616798823	244.97200700		2729082973
2401	2010.223..	32.583359	10.8.15.101	20.7.1.246	TCP	55	128	63468	0xb472 (19314)	0xa773		2814364608	2814364609	245.01202000		4031232370
2438	2192.866..	182.643575	10.8.15.101	20.7.1.246	TCP	55	128	62844	0xb473 (19315)	0x331e		616798822	616798823	245.01736000		2729082973
2451	2255.243..	62.376969	10.8.15.101	20.7.1.246	TCP	55	128	63468	0xb474 (19316)	0xa773		2814364608	2814364609	245.02049000		4031232370
2467	2409.609..	154.645802	10.8.15.101	20.7.1.246	TLSv1.2	132	128	62844	0xb476 (19317)	0xc855		616798822	616798823	245.02262000		2729082973
2488	2489.689..	0.680096	10.8.15.101	20.7.1.246	TLSv1.2	153	128	63468	0xb479 (19318)	0xc85d		2814364609	2814364708	154.44567000		4031232370
2550	2654.771..	245.081778	10.8.15.101	20.7.1.246	TCP	55	128	64240	0xb479 (19321)	0xc2c7		616798920	616798921	244.95347000		2729082973
2621	3564.797..	0.970008	10.8.15.101	20.7.1.246	TCP	55	128	62844	0xb476 (19322)	0xa773		616798822	616798823	245.02262000		4031232370

3.6 过滤/排序Ping耗时长或超时请求

3.6.1 按照耗时排序

首先将ICMP响应耗时段应用为列，如何操作可参考上面 2.9 部分。

之后只过滤icmp reply的请求：

```
icmp.type == 0
```

No.	Time	Delta	Source	Destination	Protocol	Time to Live	Identification	Length	Identifier (ID)	Identifier (ID)	Sequence Number (S)	Response time	src mac	dst mac	Info
2	2024-10-20 03:45:15.947553	0.000000	192.168.1.8	192.168.1.14	ICMP	64	0xb66c (1724)	98	8 (0xb008)	2048 (0xb000)	1 (0xb001)	9.139	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1725, ttl=64 (request in 1)
4	2024-10-20 03:45:16.941369	0.993796	192.168.1.8	192.168.1.14	ICMP	64	0xb670 (1824)	98	8 (0xb008)	2048 (0xb000)	2 (0xb002)	1.318	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1732, ttl=64 (request in 3)
6	2024-10-20 03:45:17.942700	1.000441	192.168.1.8	192.168.1.14	ICMP	64	0xb67c (1924)	98	8 (0xb008)	2048 (0xb000)	3 (0xb003)	1.073	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1740, ttl=64 (request in 5)
8	2024-10-20 03:45:18.944210	1.001420	192.168.1.8	192.168.1.14	ICMP	64	0xb67f (1987)	98	8 (0xb008)	2048 (0xb000)	4 (0xb004)	1.168	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1748, ttl=64 (request in 7)
10	2024-10-20 03:45:19.945786	1.002055	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	6 (0xb006)	8.472	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1296, ttl=64 (request in 9)
13	2024-10-20 03:45:20.947386	1.002880	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	2 (0xb002)	8.731	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1292, ttl=64 (request in 11)
15	2024-10-20 03:45:21.948981	1.003605	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	4 (0xb004)	8.678	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1298, ttl=64 (request in 13)
17	2024-10-20 03:45:22.950576	1.004330	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	8 (0xb008)	8.581	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1302, ttl=64 (request in 15)
19	2024-10-20 03:45:23.952171	1.005055	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	5 (0xb005)	8.471	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1306, ttl=64 (request in 17)
21	2024-10-20 03:45:24.953766	1.005780	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	8 (0xb008)	8.621	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1310, ttl=64 (request in 19)
23	2024-10-20 03:45:25.955361	1.006505	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	8 (0xb008)	8.827	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1314, ttl=64 (request in 21)

可以看到Response time字段直观列出了每个icmp响应报文的RTT耗时。

单击两下这个字段，则可以让它从大到小排列，哪个报文耗时最大可以更直观看出：

No.	Time	Delta	Source	Destination	Protocol	Time to Live	Identification	Length	Identifier (ID)	Identifier (ID)	Sequence Number (S)	Response time	src mac	dst mac	Info
23	2024-10-20 03:45:25.955361	1.006505	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	7 (0xb007)	8.827	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1312, ttl=64 (request in 23)
13	2024-10-20 03:45:19.945786	1.002055	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	2 (0xb002)	8.731	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1296, ttl=64 (request in 11)
15	2024-10-20 03:45:21.948981	1.003605	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	3 (0xb003)	8.678	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1298, ttl=64 (request in 13)
21	2024-10-20 03:45:25.955361	1.006505	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	6 (0xb006)	8.633	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1304, ttl=64 (request in 20)
17	2024-10-20 03:45:22.950576	1.004330	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	4 (0xb004)	8.581	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1302, ttl=64 (request in 15)
19	2024-10-20 03:45:23.952171	1.005055	119.29.29.29	192.168.1.14	ICMP	52	0xb6c2 (4813)	98	9 (0xb009)	2304 (0xb000)	1 (0xb001)	8.472	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb009, seq=1290, ttl=64 (request in 9)
11	2024-10-20 03:45:17.942700	1.001420	192.168.1.8	192.168.1.14	ICMP	64	0xb67f (1987)	98	8 (0xb008)	2048 (0xb000)	5 (0xb005)	8.471	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1746, ttl=64 (request in 13)
4	2024-10-20 03:45:16.941369	0.993796	192.168.1.8	192.168.1.14	ICMP	64	0xb670 (1824)	98	8 (0xb008)	2048 (0xb000)	2 (0xb002)	1.318	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1732, ttl=64 (request in 3)
8	2024-10-20 03:45:18.944210	1.001420	192.168.1.8	192.168.1.14	ICMP	64	0xb67f (1987)	98	8 (0xb008)	2048 (0xb000)	4 (0xb004)	1.168	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1748, ttl=64 (request in 7)
6	2024-10-20 03:45:17.942700	1.001421	192.168.1.8	192.168.1.14	ICMP	64	0xb67c (1836)	98	8 (0xb008)	2048 (0xb000)	3 (0xb003)	1.073	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply id=0xb008, seq=1740, ttl=64 (request in 5)

3.6.2 过滤耗时的icmp reply

紧接着，如果想过滤耗时超过8.5ms的请求，可以是：

```
icmp.resptime >= 8.5
```


以第一条9.139毫秒的icmp reply为例，想要过滤对应的icmp request请求+reply响应报文，可以是：

```
icmp.seq == 1 && icmp.ident == 8
```

3.6.3 过滤Ping超时/不响应的报文

一条语句即可搞定：

```
icmp.resp_not_found
```

可以看到ping国内公共DNS 114没有拿到响应，不排除对端禁ping的情况。

3.7 过滤/排序DNS解析耗时长或超时的请求

3.7.1 按照耗时排序

首先找到一条dns response的报文，展开后找到最底下的dns.time字段，右击应用为列，如何操作也可参考上面 2.9 节。

241	0.027454	10.8.15.1	10.8.15.101	DNS	A	www.msftconnecttest.com	0xc8b3
243	0.635448	10.8.15.101	10.8.15.1	DNS	A	geo.prod.do.dsp.mp.microsoft.com	0x83d0

```

> Frame 20: 227 bytes on wire (1816 bits), 227 bytes captured (1816 bits)
> Ethernet II, Src: Cisco_12:f7:a4 (00:38:df:12:f7:a4), Dst: 78:c2:3b:b8:93:e8 (78:c2:3b:b8:93:e8)
> Internet Protocol Version 4, Src: 10.8.15.1, Dst: 10.8.15.101
> User Datagram Protocol, Src Port: 53, Dst Port: 61984
< Domain Name System (response)
  Transaction ID: 0x614e
  > Flags: 0x8180 Standard query response, No error
  Questions: 1
  Answer RRs: 5
  Authority RRs: 0
  Additional RRs: 0
  > Queries
  > Answers

```

[Request In: 19]

[Time: 0.02231000 seconds]

- Expand Subtrees
- Collapse Subtrees
- Expand All
- Collapse All
- Apply as Column Ctrl+Shift+I
- Apply as Filter ▶
- Prepare as Filter ▶
- Conversation Filter ▶
- Colorize with Filter ▶
- Follow ▶
- I/O Graph ▶
- Copy ▶
- Show Packet Bytes... Ctrl+Shift+O
- Export Packet Bytes... Ctrl+Shift+X
- Wiki Protocol Page
- Filter Field Reference
- Protocol Preferences ▶
- Decode As... Ctrl+Shift+U
- Go to Linked Packet
- Show Linked Packet in New Window

应用为列后，过滤请求可以直接写 dns.time，之后单击两下我们刚新增的列：

No.	Delta	Source	Destination	Protocol	Type	Query Name	dns.id	dns.rsp.time	Length	Time to Live	Identification	Info
473	0.728995	10.8.15.1	10.8.15.101	DNS	A	www.msftconnecttest.com	0xc8b8	0.635400	243	128	0xc8b4 (36856)	Standard query response 0xc8b8 A www.msftconnecttest.com CNAME ncsi-101
2324	0.708138	10.8.15.1	10.8.15.101	DNS	A	dns.msfncls.com	0xfaf6	0.41177000	92	128	0xfaf1 (36113)	Standard query response 0xfaf6 A dns.msfncls.com A 131.107.255.255
399	16.922976	10.8.15.1	10.8.15.101	DNS	A	fd.api.iris.microsoft.com	0xf918	0.41168000	210	128	0xf91d (36797)	Standard query response 0xf918 A fd.api.iris.microsoft.com CNAME fd-#
733	0.628161	10.8.15.1	10.8.15.101	DNS	A	www.msn.com	0x8264	0.37924000	146	128	0x826f (36975)	Standard query response 0x8264 A www.msn.com CNAME www.msn-com-a-8083
508	22.620295	10.8.15.1	10.8.15.101	DNS	A	v10.events.data.microsoft.com	0x5744	0.31609000	226	128	0x574f (36845)	Standard query response 0x5744 A v10.events.data.microsoft.com CNAME v
2336	0.674926	10.8.15.1	10.8.15.101	DNS	A	www.bing.com	0x5cfd	0.33493600	337	128	0x5d02 (37991)	Standard query response 0x5cfd A www.bing.com CNAME www-wm.bing.com.t
2885	11.151870	10.8.15.1	10.8.15.101	DNS	A	login.live.com	0x6073	0.33495000	359	128	0x607e (37838)	Standard query response 0x6073 A login.live.com CNAME login.msa.mide#
2383	54.153941	10.8.15.1	10.8.15.101	DNS	A	www.bing.com	0x5c6e	0.33430000	337	128	0x5c71 (38820)	Standard query response 0x5c6e A www.bing.com CNAME www-wm.bing.com.t
1964	298.905981	10.8.15.1	10.8.15.101	DNS	A	licensing.mp.microsoft.com	0x9eda	0.33405000	282	128	0x9ee0 (37776)	Standard query response 0x9eda A licensing.mp.microsoft.com CNAME cons
2385	18.036618	10.8.15.1	10.8.15.101	DNS	A	dns.msfncls.com	0x3400	0.33141000	92	128	0x3406 (38922)	Standard query response 0x3400 A dns.msfncls.com A 131.107.255.255
1865	0.188808	10.8.15.1	10.8.15.101	DNS	A	ecn-us.dev.virtualsearch.net	0x1c08	0.33082000	189	128	0x1c12 (37386)	Standard query response 0x1c08 A ecn-us.dev.virtualsearch.net CNAME ee
2522	569.009044	10.8.15.1	10.8.15.101	DNS	A	www.bing.com	0x1dca	0.329417000	337	128	0x1de0 (38112)	Standard query response 0x1dca A www.bing.com CNAME www-wm.bing.com.t
1726	182.372762	10.8.15.1	10.8.15.101	DNS	A	v10.events.data.microsoft.com	0x8480	0.32787000	227	128	0x848d (37586)	Standard query response 0x8480 A v10.events.data.microsoft.com CNAME v
1865	372.589439	10.8.15.1	10.8.15.101	DNS	A	fd.api.iris.microsoft.com	0x70b3	0.32753000	215	128	0x70b9 (37669)	Standard query response 0x70b3 A fd.api.iris.microsoft.com CNAME fd-#
241	0.713086	10.8.15.1	10.8.15.101	DNS	A	www.msftconnecttest.com	0xc8b1	0.32745000	247	128	0xc8b7 (38211)	Standard query response 0xc8b1 A www.msftconnecttest.com CNAME ncsi-101

可以看到dns响应耗时已经大到小排序。

3.7.2 过滤耗时长的dns response响应

dns.time字段的单位是秒，过滤耗时超过50ms的dns响应可以是：

dns.time > 0.05

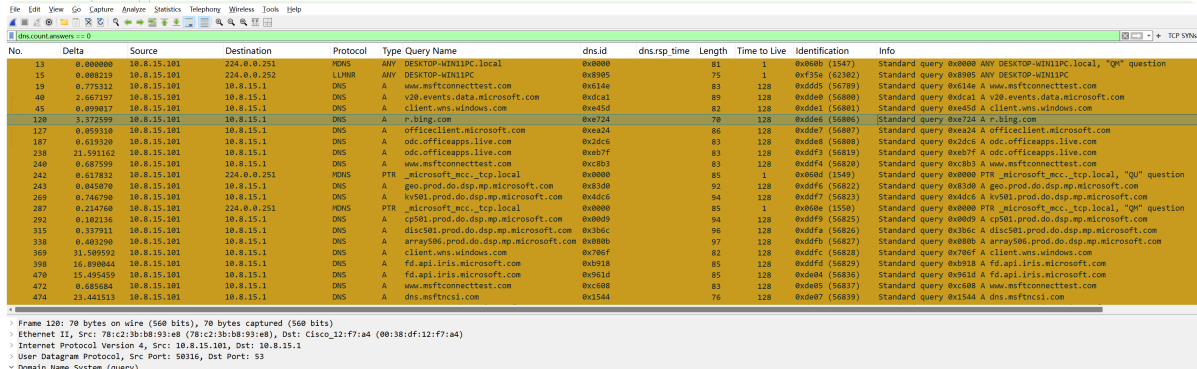
No.	Delta	Source	Destination	Protocol	Type	Query Name	dns.id	dns.rsp.time	Length	Time to Live	Identification	Info
473	0.728995	10.8.15.1	10.8.15.101	DNS	A	www.msftconnecttest.com	0xc8b8	0.635400	243	128	0xc8b4 (36856)	Standard query response 0xc8b8 A www.msftconnecttest.com CNAME ncsi-101

3.7.3 过滤没有解析到地址的报文

一条语句即可搞定：

```
dns.count.answers == 0
```

这些请求都没有解析到地址，返回的记录数量为0：

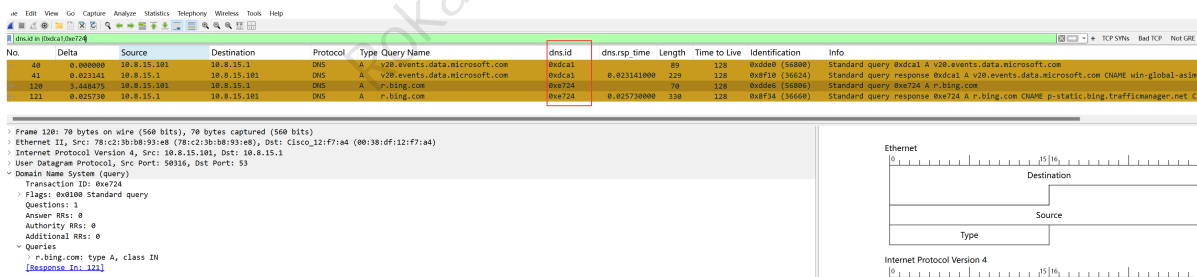


3.8 跟踪DNS请求和响应/过滤DNS解析的域名

3.8.1 跟踪dns请求 (dns.id)

可以根据dns.id字段来跟踪一条DNS请求和对应的响应，比如追踪如下两个dns.id字段：

```
dns.id in {0xdca1,0xe724}
```



3.8.2 过滤DNS解析的域名 (dns qry.name)

过滤dns解析的域名，比如cloud.tencent.com，可以是：

```
dns.qry.name == "cloud.tencent.com"
```

因为这个字段数据类型属于字符串类型，再配合前面讲到的正则匹配，匹配满足要求的多个域名，可以是：

```
dns.qry.name ~ "bing.com$|microsoft.com$"
```

No.	Delta	Source	Destination	Protocol	Type	Query Name	dns.id	dns.rsp.time	Length	Time to Live	Identification	Info
40	0.000000	10.8.15.101	10.8.15.1	DNS	A	v20.events.data.microsoft.com	0x6ca1		89	128	0x0500 (56888)	Standard query 0x6ca1 A v20.events.data.microsoft.com
41	0.001341	10.8.15.1	10.8.15.101	DNS	A	v20.events.data.microsoft.com	0x6ca1	0.021341000	228	128	0x0f20 (36624)	Standard query response 0x6ca1 A v20.events.data.microsoft.com CNAME win-glob
120	3.448475	10.8.15.101	10.8.15.1	DNS	A	r.bing.com	0xe724		78	128	0x0500 (56888)	Standard query 0xe724 A r.bing.com
121	0.025730	10.8.15.1	10.8.15.101	DNS	A	r.bing.com	0xe724	0.025730000	330	128	0x0f30 (36660)	Standard query response 0xe724 A r.bing.com CNAME p-static.bing.trafficmanage
127	0.033508	10.8.15.101	10.8.15.1	DNS	A	officeclient.microsoft.com	0xea24		96	128	0x0500 (56888)	Standard query 0xea24 A officeclient.microsoft.com
128	0.025107	10.8.15.1	10.8.15.101	DNS	A	officeclient.microsoft.com	0xea24	0.025107000	205	128	0x0f37 (36663)	Standard query response 0xea24 A officeclient.microsoft.com CNAME config.offic
243	23.535876	10.8.15.101	10.8.15.1	DNS	A	geo.prod.do.dsp.mp.microsoft.com	0x4d30		92	128	0x0500 (56888)	Standard query 0x4d30 A geo.prod.do.dsp.mp.microsoft.com
244	0.002351	10.8.15.1	10.8.15.101	DNS	A	geo.prod.do.dsp.mp.microsoft.com	0x4d30	0.002351000	207	128	0x0f60 (36712)	Standard query response 0x4d30 A geo.prod.do.dsp.mp.microsoft.com CNAME geo-p
269	0.744439	10.8.15.101	10.8.15.1	DNS	A	kv501.prod.do.dsp.mp.microsoft.com	0x4dc6		94	128	0x0500 (56888)	Standard query 0x4dc6 A kv501.prod.do.dsp.mp.microsoft.com
270	0.016065	10.8.15.1	10.8.15.101	DNS	A	kv501.prod.do.dsp.mp.microsoft.com	0x4dc6	0.016065000	204	128	0x0f70 (36720)	Standard query response 0x4dc6 A kv501.prod.do.dsp.mp.microsoft.com CNAME kv50

3.9 连接被重置 (Reset)

相信很多人或多或少遇到过访问某个网站时，浏览器提示“对端已拒绝连接”、“ERR_CONNECTION_REFUSED”、“连接被重置”等报错：



我们不妨来分析下这类场景下的报文：

No.	Time	Delta	Source	Destination	Protocol	Length	MSS	Window	Identification	Checksum	Time to Live	Time since previous frame in this TCP stream	Info
1	0.000000	0.000000	192.168.247.105	72.52.171.190	TCP	66	1460	8192	0x1203 (4699)	0x2064	112	0.000000000	53851 → 80 [SYN] Seq=1781086924 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
2	0.035812	0.035812	72.52.171.190	192.168.247.105	TCP	60	479	0x0893 (35937)	0x0986	0x112	112	0.035812000	80 → 53851 [SYN, ACK] Seq=1298967769 Ack=1781086925 Win=479 Len=0
3	0.035893	0.000081	192.168.247.105	72.52.171.190	TCP	54	6392	0x1192 (4618)	0x9275	0x128	128	0.000001000	53851 → 80 [ACK] Seq=1781086925 Ack=1298967770 Win=6392 Len=0
4	0.036008	0.000115	192.168.247.105	72.52.171.190	HTTP	465	65392	0x1203 (4611)	0x0e15	0x128	128	0.000157000	GET / HTTP/1.1
5	0.038029	0.002021	72.52.171.190	192.168.247.105	TCP	76	32708	0x0100 (256)	0x303a	0x128	253	0.002039000	80 → 53851 [RST, ACK, Reserved] Seq=1298967770 Ack=1781087335 Win=32768 Len=22

三次握手成功建立，此时客户端发送GET请求，对端响应了RST-ACK断开连接，同时，对端发送的SYN-ACK的TTL是112，ip.id为99537，而RST-ACK的TTL突然变成了253，ip.id变成了256，很明显，这个RST-ACK报文并不是对端主动发送的。那么会是谁发送的？有几种可能：

- 域名解析到国内服务器，并且没备案，被阻断访问；
- 涉及到不合规业务，被运营商或相关部门阻断拦截；
- 对端的上层或客户端上层有安全防火墙等安全类设备或策略阻断访问，代答了RST-ACK。

这类场景下如果备案没问题，且对端和本段都没有安全墙，则建议报障当地运营商看下，或联系对端服务商反馈此问题。

3.10 指定时间范围/指定帧数范围过滤

上面讲了这么多技巧，相信你已经可以依葫芦画瓢，找到对应的字段，进行范围过滤。

3.10.1 指定时间范围 (frame.time)

一般用的是北京时间 (frame.time) 进行时间维度的过滤，当然你也可以用UTC时间 (frame.time_utc)，北京时间和UTC时间对应下面这两个时间字段：

No.	Time	Delta	Source	Destination	Protocol	Time to Lh	Identification	Length	Identifier (BE)	Identifier (LE)	Sequence Number (BE)	Response time	src mac	dst mac	Info
1	2024-10-20 03:45:15.938414	0.000000	192.168.1.14	192.168.1.8	ICMP	64	0xc25b (49755)	98	8 (0x0008)	2048 (0x0800)	1 (0x0001)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
2	2024-10-20 03:45:15.947553	0.009139	192.168.1.8	192.168.1.14	ICMP	64	0xb6bc (1724)	98	8 (0x0008)	2048 (0x0800)	1 (0x0001)	9.139	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
3	2024-10-20 03:45:16.940931	0.992478	192.168.1.14	192.168.1.8	ICMP	64	0xc328 (49960)	98	8 (0x0008)	2048 (0x0800)	2 (0x0002)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
4	2024-10-20 03:45:16.941349	0.001318	192.168.1.14	192.168.1.14	ICMP	64	0x0720 (1824)	98	8 (0x0008)	2048 (0x0800)	2 (0x0002)	1.318	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
5	2024-10-20 03:45:17.941717	1.000368	192.168.1.14	192.168.1.8	ICMP	64	0xc3ac (50902)	98	8 (0x0008)	2048 (0x0800)	3 (0x0003)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
6	2024-10-20 03:45:17.942790	0.001073	192.168.1.8	192.168.1.14	ICMP	64	0x072c (1836)	98	8 (0x0008)	2048 (0x0800)	3 (0x0003)	1.073	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
7	2024-10-20 03:45:18.943042	1.000252	192.168.1.14	192.168.1.8	ICMP	64	0xc44a (50250)	98	8 (0x0008)	2048 (0x0800)	4 (0x0004)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
8	2024-10-20 03:45:18.944210	0.001168	192.168.1.8	192.168.1.14	ICMP	64	0x075f (1887)	98	8 (0x0008)	2048 (0x0800)	4 (0x0004)	1.168	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
9	2024-10-20 03:45:27.129314	8.185104	192.168.1.14	119.29.29.29	ICMP	64	0xb6c2d (48173)	98	9 (0x0009)	2304 (0x0900)	1 (0x0001)		00:50:56:9f:bf:57	00:50:56:81:10:8e	Echo (ping) request

过滤某个时间之后：

```
frame.time >= "2024-10-20 03:45:00"
```

No.	Time	Delta	Source	Destination	Protocol	Time to Lh	Identification	Length	Identifier (BE)	Identifier (LE)	Sequence Number (BE)	Response time	src mac	dst mac	Info
1	2024-10-20 03:45:15.938414	0.000000	192.168.1.14	192.168.1.8	ICMP	64	0xc25b (49755)	98	8 (0x0008)	2048 (0x0800)	1 (0x0001)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
2	2024-10-20 03:45:15.947553	0.009139	192.168.1.8	192.168.1.14	ICMP	64	0xb6bc (1724)	98	8 (0x0008)	2048 (0x0800)	1 (0x0001)	9.139	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
3	2024-10-20 03:45:16.940931	0.992478	192.168.1.14	192.168.1.8	ICMP	64	0xc328 (49960)	98	8 (0x0008)	2048 (0x0800)	2 (0x0002)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
4	2024-10-20 03:45:16.941349	0.001318	192.168.1.14	192.168.1.14	ICMP	64	0x0720 (1824)	98	8 (0x0008)	2048 (0x0800)	2 (0x0002)	1.318	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
5	2024-10-20 03:45:17.941717	1.000368	192.168.1.14	192.168.1.8	ICMP	64	0xc3ac (50902)	98	8 (0x0008)	2048 (0x0800)	3 (0x0003)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
6	2024-10-20 03:45:17.942790	0.001073	192.168.1.8	192.168.1.14	ICMP	64	0x072c (1836)	98	8 (0x0008)	2048 (0x0800)	3 (0x0003)	1.073	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
7	2024-10-20 03:45:18.943042	1.000252	192.168.1.14	192.168.1.8	ICMP	64	0xc44a (50250)	98	8 (0x0008)	2048 (0x0800)	4 (0x0004)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
8	2024-10-20 03:45:18.944210	0.001168	192.168.1.8	192.168.1.14	ICMP	64	0x075f (1887)	98	8 (0x0008)	2048 (0x0800)	4 (0x0004)	1.168	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
9	2024-10-20 03:45:27.129314	8.185104	192.168.1.14	119.29.29.29	ICMP	64	0xb6c2d (48173)	98	9 (0x0009)	2304 (0x0900)	1 (0x0001)		00:50:56:9f:bf:57	00:50:56:81:10:8e	Echo (ping) request

过滤某个时间范围：

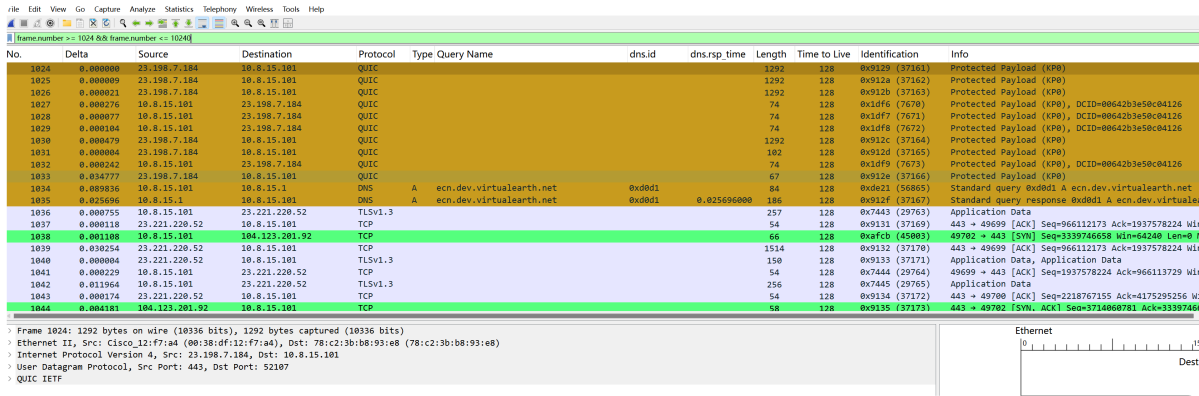
```
frame.time >= "2024-10-20 03:45:00" && frame.time <= "2024-10-20 03:45:30"
```

No.	Time	Delta	Source	Destination	Protocol	Time to Lh	Identification	Length	Identifier (BE)	Identifier (LE)	Sequence Number (BE)	Response time	src mac	dst mac	Info
1	2024-10-20 03:45:15.938414	0.000000	192.168.1.14	192.168.1.8	ICMP	64	0xc25b (49755)	98	8 (0x0008)	2048 (0x0800)	1 (0x0001)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
2	2024-10-20 03:45:15.947553	0.009139	192.168.1.8	192.168.1.14	ICMP	64	0xb6bc (1724)	98	8 (0x0008)	2048 (0x0800)	1 (0x0001)	9.139	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
3	2024-10-20 03:45:16.940931	0.992478	192.168.1.14	192.168.1.8	ICMP	64	0xc328 (49960)	98	8 (0x0008)	2048 (0x0800)	2 (0x0002)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
4	2024-10-20 03:45:16.941349	0.001318	192.168.1.14	192.168.1.14	ICMP	64	0x0720 (1824)	98	8 (0x0008)	2048 (0x0800)	2 (0x0002)	1.318	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
5	2024-10-20 03:45:17.941717	1.000368	192.168.1.14	192.168.1.8	ICMP	64	0xc3ac (50902)	98	8 (0x0008)	2048 (0x0800)	3 (0x0003)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
6	2024-10-20 03:45:17.942790	0.001073	192.168.1.8	192.168.1.14	ICMP	64	0x072c (1836)	98	8 (0x0008)	2048 (0x0800)	3 (0x0003)	1.073	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
7	2024-10-20 03:45:18.943042	1.000252	192.168.1.14	192.168.1.8	ICMP	64	0xc44a (50250)	98	8 (0x0008)	2048 (0x0800)	4 (0x0004)		00:50:56:9f:bf:57	00:50:56:81:c0:1e	Echo (ping) request
8	2024-10-20 03:45:18.944210	0.001168	192.168.1.8	192.168.1.14	ICMP	64	0x075f (1887)	98	8 (0x0008)	2048 (0x0800)	4 (0x0004)	1.168	00:50:56:81:c0:1e	00:50:56:9f:bf:57	Echo (ping) reply
9	2024-10-20 03:45:27.129314	8.185104	192.168.1.14	119.29.29.29	ICMP	64	0xb6c2d (48173)	98	9 (0x0009)	2304 (0x0900)	1 (0x0001)		00:50:56:9f:bf:57	00:50:56:81:10:8e	Echo (ping) request

3.10.2 指定帧数范围 (frame.number)

过滤1024~10240帧可以是：

```
frame.number >= 1024 && frame.number <= 10240
```



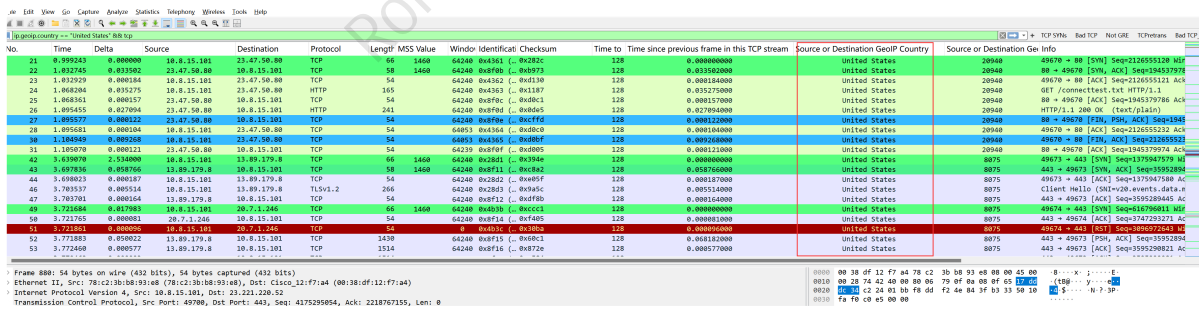
3.11 通过IP归属地/AS号过滤

这个功能需要Wireshark配置IP地址库才能实现，如何配置可参考笔者的[这篇文章](#)，并且有更详细的关于地址库字段的过滤用法，下面只简单列举较为常用的两个实例。

3.11.1 通过IP归属地过滤

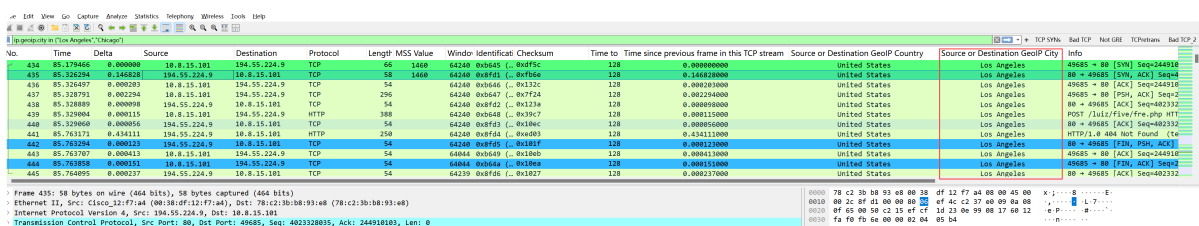
过滤指定国家的TCP报文，比如过滤IP归属地为美国的TCP报文，可以是：

```
ip.geopip.country == "United States" && tcp
```



当然也可以是城市的维度，比如只筛选归属地为洛杉矶或者芝加哥的请求，可以这么写：

```
ip.geopip.city in {"Los Angeles", "Chicago"}
```

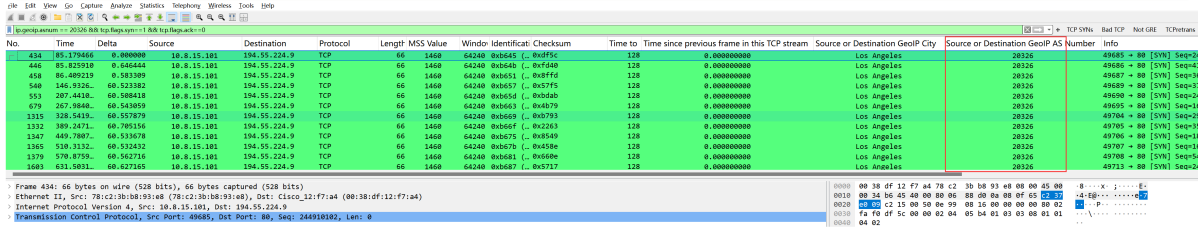


3.11.2 通过AS号过滤

通过AS号过滤和上面的过滤方式同理，找到对应的过滤字段即可。

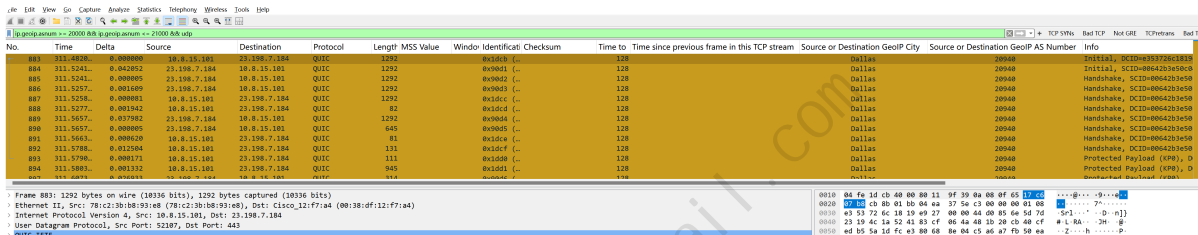
过滤某个特定的AS号的SYN请求：

```
ip.geoip.asnum == 20326 && tcp.flags.syn==1
```



过滤某个AS号范围的UDP报文：

```
ip.geoip.asnum >= 20000 && ip.geoip.asnum <= 21000 && udp
```



3.12 过滤端口复用的情况

Wireshark对于TCP端口复用的定义：

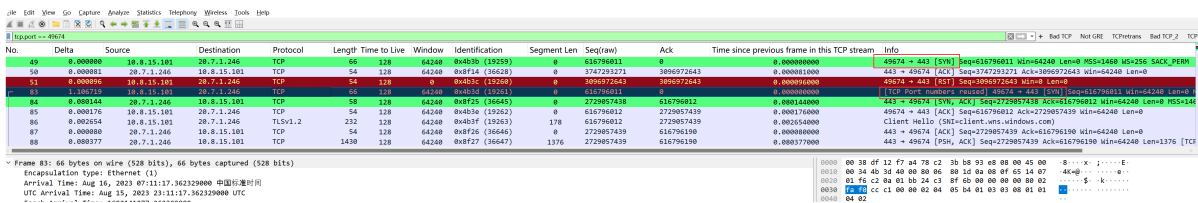
TCP Port numbers reused

Set when the SYN flag is set (not SYN+ACK), we have an existing conversation using the same addresses and ports, and the sequence number is different than the existing conversation's initial sequence number.

即当在一个抓包文件中，SYN标志位的报文（不包含SYN-ACK），有一个使用相同地址和端口的现有会话，seq与现有会话的初始seq不同时，会将此SYN报文标记为端口复用。

基于这个定义，那么即使一个报文中，TCP stream 1的SYN使用了一组源目的IP和端口，而这个包的TCP stream 100的SYN也使用了这组源目的IP和端口，即使这两条流相差十万八千里，哪怕一年十年间隔之久，只要出现在同一个抓包文件里面，Wireshark就能根据定义，把后面出现的TCP stream 100的SYN标记为端口复用。

比如下面这个例子：

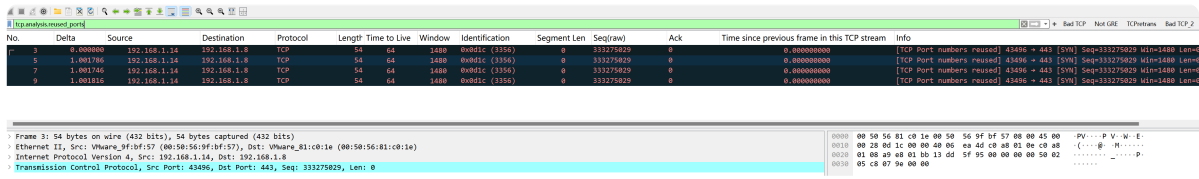


第49帧和83帧，SYN报文在不同的TCP流中，使用同一个源IP、目的IP、源端口、目的端口，所以后面出现的SYN，Wireshark标记为端口复用。

端口复用虽然Wireshark默认标记为红黑色，但并不表示异常，而是提醒用户这里有端口复用的情况，主要看对端会不会拒绝请求，如果不拒绝能正常处理响应SYN-ACK，那没什么问题，如果拒绝或者不响应，则要核实下对端是不是关闭了TCP快速回收或其它可能配置，主要取决于对端的反应。

过滤端口复用的情况，可以使用下面的过滤语句：

```
tcp.analysis.reused_ports
```



3.13 TCP Keep-Alive的定义和过滤

Wireshark对于TCP Keep-Alive的定义：

TCP Keep-Alive

Set when the segment size is zero or one, the current sequence number is one byte less than the next expected sequence number, and none of SYN, FIN, or RST are set.

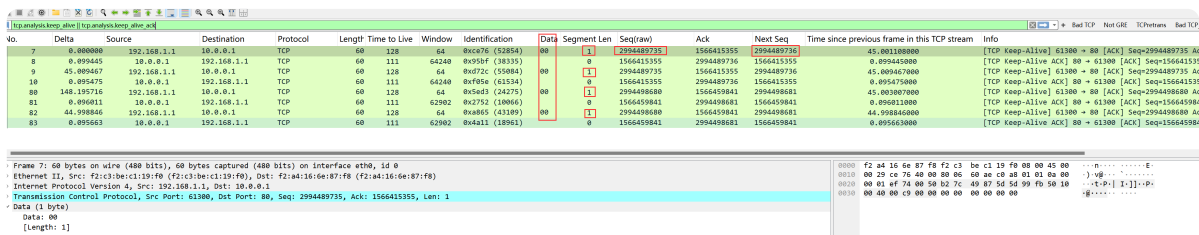
Supersedes “Fast Retransmission”, “Out-Of-Order”, “Spurious Retransmission”, and “Retransmission”.

即下面三个条件必须同时满足时，才会标记为TCP Keep-Alive：

- 段长度 (segment length) 为0或者1字节；
- 当前序列号比下一个期望的序列号小1；
- SYN、FIN、RST等标志位没有置1。

过滤这类包的语法是：

```
tcp.analysis.keep_alive || tcp.analysis.keep_alive_ack #这里把keep-alive ack回包也加入进来
```



可以看出，过滤出来的keep-alive包均同时满足上面三个条件，段长度为1的情况下，填充的数据是0，对应十六进制0x00，表示这是一个空的数据段。为什么设置为1？TCP协议要求每个数据段至少包含1个字节的有效载荷，设置长度为1字节确保了探测包满足这一要求，既节省了开销，又达到了刷新会话保活时间的能力（这里要和HTTP Keep-Alive的长连接区分开来）。