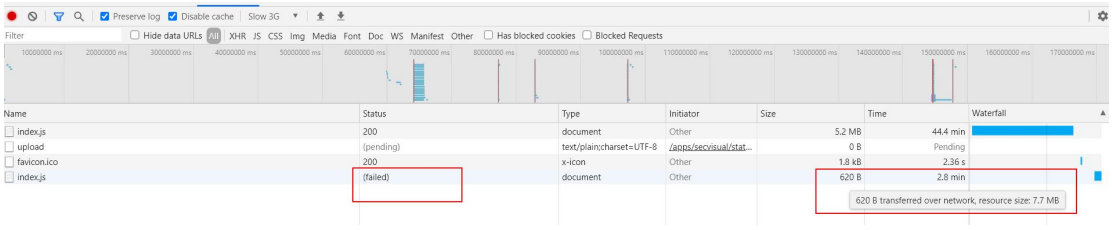
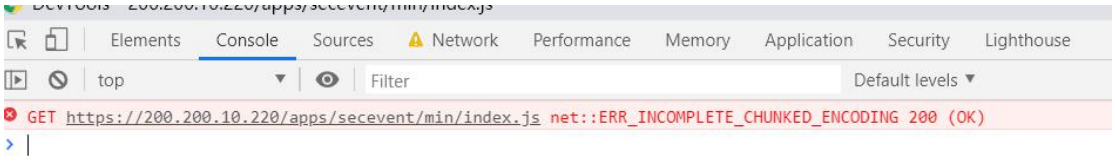
现象：

客户环境，网络很慢，任意浏览器打开处置中心都会白屏，然后报错，f12 查看是某个 js 文件一直加载失败，报错 net::ERR_INCOMPLETE_CHUNKED_ENCODING error。大概率出现，但不是必现。





问题查询 N 步走：

1. 客户出现的情况的必要条件是

(1) 网络很慢

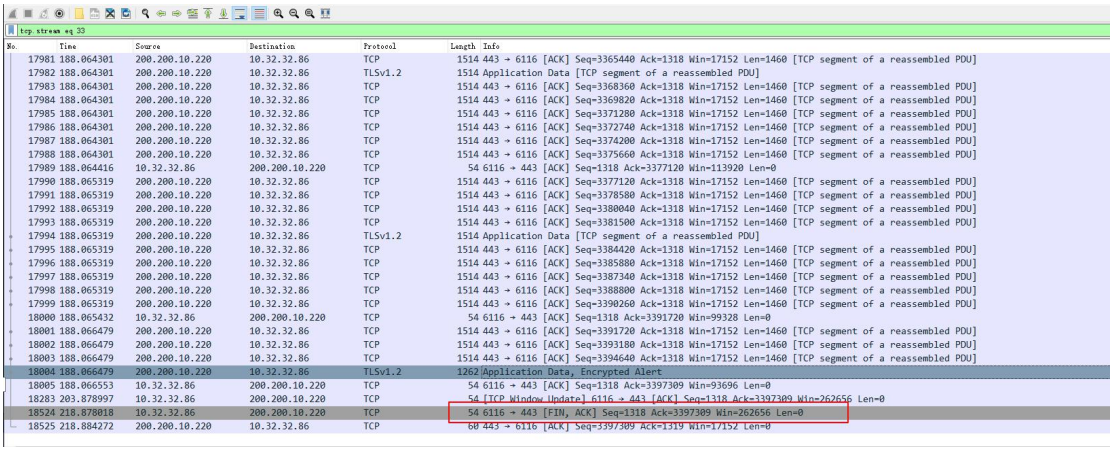
(2) js 文件比较大，达到 11M 以上。

本地模拟，在同一个 index.js 文件加上很多注释，使文件达到 20M 以上，同时 chrome 浏览器设置 slow 3G，模拟网络很慢的情况，本地复现。

2. 查看后台是 sangfor_waf+apache 的组合，reset_apache 后不会出现，说明只有 waf+apache 才会出现。监控 waf 和 apache 的进程有没有重启和有没有错误日志生成，并没有相关日志。今天前端那边说，不用 waf 能会出现。

3. 网络抓包，客户端抓包和服务器端 9443 端口抓包。

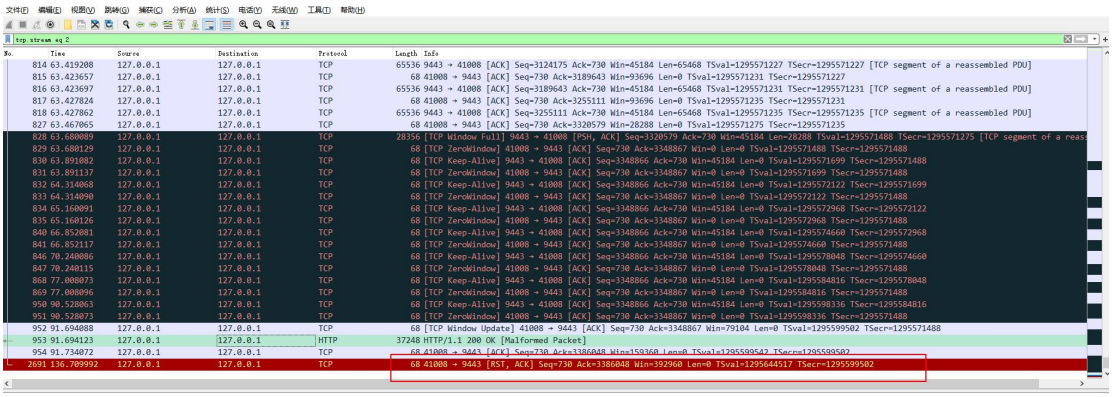
客户端抓包：



No.	Time	Source	Destination	Protocol	Length	Info
17981	188.064301	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3365440 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17982	188.064301	200.200.10.220	10.32.32.86	TLSv1.2	1514	Application Data [TCP segment of a reassembled PDU]
17983	188.064301	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3368360 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17984	188.064301	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3369820 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17985	188.064301	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3371280 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17986	188.064301	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3372740 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17987	188.064301	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3374200 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17988	188.064301	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3375660 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17989	188.064416	10.32.32.86	200.200.10.220	TCP	54	6116 → 443 [ACK] Seq=1318 Ack=3377120 Win=113920 Len=0
17990	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3377120 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17991	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3378580 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17992	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3380040 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17993	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3381500 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17994	188.065319	200.200.10.220	10.32.32.86	TLSv1.2	1514	Application Data [TCP segment of a reassembled PDU]
17995	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3384420 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17996	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3385880 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17997	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3387340 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17998	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3388800 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
17999	188.065319	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3390260 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
18000	188.065432	10.32.32.86	200.200.10.220	TCP	54	6116 → 443 [ACK] Seq=1318 Ack=3391720 Win=99328 Len=0
18001	188.066479	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3391720 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
18002	188.066479	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3393180 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
18003	188.066479	200.200.10.220	10.32.32.86	TCP	1514	443 → 6116 [ACK] Seq=3394640 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassembled PDU]
18004	188.066479	200.200.10.220	10.32.32.86	TLSv1.2	1262	Application Data, Encrypted Alert
18005	188.066553	10.32.32.86	200.200.10.220	TCP	54	6116 → 443 [ACK] Seq=1318 Ack=3397309 Win=93696 Len=0
18283	203.878997	10.32.32.86	200.200.10.220	TCP	54	[TCP Window Update] 6116 → 443 [ACK] Seq=1318 Ack=3397309 Win=262656 Len=0
18524	218.878018	10.32.32.86	200.200.10.220	TCP	54	6116 → 443 [FIN, ACK] Seq=1318 Ack=3397309 Win=262656 Len=0
18525	218.884272	200.200.10.220	10.32.32.86	TCP	60	443 → 6116 [ACK] Seq=3397309 Ack=1319 Win=17152 Len=0

刚开始只看到一个 FIN 包，而且是客户端发出，难道是客户端先发的 close 请求？感觉很奇怪，在这里先不急下结论

服务器端 9443 端口抓包：



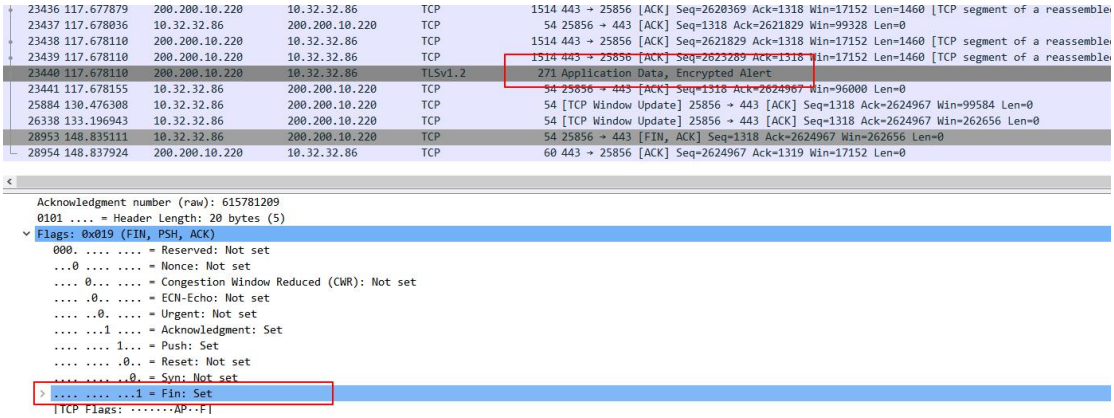
No.	Time	Source	Destination	Protocol	Length	Info
814	63.419208	127.0.0.1	127.0.0.1	TCP	65536	9443 → 41008 [ACK] Seq=3124175 Ack=730 Win=45184 Len=65468 TSval=1295571227 TSecr=1295571227 [TCP segment of a reassembled PDU]
815	63.423657	127.0.0.1	127.0.0.1	TCP	68	41008 → 9443 [ACK] Seq=730 Ack=3189643 Win=93696 Len=0 TSval=1295571231 TSecr=1295571227
816	63.423697	127.0.0.1	127.0.0.1	TCP	65536	9443 → 41008 [ACK] Seq=3189643 Ack=730 Win=45184 Len=65468 TSval=1295571231 TSecr=1295571231 [TCP segment of a reassembled PDU]
817	63.427824	127.0.0.1	127.0.0.1	TCP	68	41008 → 9443 [ACK] Seq=730 Ack=3255111 Win=93696 Len=0 TSval=1295571235 TSecr=1295571231
818	63.427862	127.0.0.1	127.0.0.1	TCP	65536	9443 → 41008 [ACK] Seq=3255111 Ack=730 Win=45184 Len=65468 TSval=1295571235 TSecr=1295571235 [TCP segment of a reassembled PDU]
827	63.467065	127.0.0.1	127.0.0.1	TCP	68	41008 → 9443 [ACK] Seq=730 Ack=3320579 Win=28288 Len=0 TSval=1295571275 TSecr=1295571235
828	63.680089	127.0.0.1	127.0.0.1	TCP	28356	[TCP Window Full] 9443 → 41008 [PSH, ACK] Seq=3320579 Ack=730 Win=45184 Len=28288 TSval=1295571488 TSecr=1295571275 [TCP segment of a reas
829	63.680129	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295571488 TSecr=1295571488
830	63.891082	127.0.0.1	127.0.0.1	TCP	68	[TCP Keep-Alive] 9443 → 41008 [ACK] Seq=3348866 Ack=730 Win=45184 Len=0 TSval=1295571699 TSecr=1295571488
831	63.891137	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295571699 TSecr=1295571488
832	64.314068	127.0.0.1	127.0.0.1	TCP	68	[TCP Keep-Alive] 9443 → 41008 [ACK] Seq=3348866 Ack=730 Win=45184 Len=0 TSval=1295572122 TSecr=1295571699
833	64.314090	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295572122 TSecr=1295571488
834	65.160091	127.0.0.1	127.0.0.1	TCP	68	[TCP Keep-Alive] 9443 → 41008 [ACK] Seq=3348866 Ack=730 Win=45184 Len=0 TSval=1295572968 TSecr=1295572122
835	65.160126	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295572968 TSecr=1295571488
840	66.852081	127.0.0.1	127.0.0.1	TCP	68	[TCP Keep-Alive] 9443 → 41008 [ACK] Seq=3348866 Ack=730 Win=45184 Len=0 TSval=1295574660 TSecr=1295572968
841	66.852117	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295574660 TSecr=1295571488
846	70.240086	127.0.0.1	127.0.0.1	TCP	68	[TCP Keep-Alive] 9443 → 41008 [ACK] Seq=3348866 Ack=730 Win=45184 Len=0 TSval=1295578048 TSecr=1295574660
847	70.240115	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295578048 TSecr=1295571488
868	77.080073	127.0.0.1	127.0.0.1	TCP	68	[TCP Keep-Alive] 9443 → 41008 [ACK] Seq=3348866 Ack=730 Win=45184 Len=0 TSval=1295584816 TSecr=1295578048
869	77.080096	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295584816 TSecr=1295571488
950	90.528063	127.0.0.1	127.0.0.1	TCP	68	[TCP Keep-Alive] 9443 → 41008 [ACK] Seq=3348866 Ack=730 Win=45184 Len=0 TSval=1295598336 TSecr=1295584816
951	90.528073	127.0.0.1	127.0.0.1	TCP	68	[TCP ZeroWindow] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=0 Len=0 TSval=1295598336 TSecr=1295571488
952	91.694088	127.0.0.1	127.0.0.1	TCP	68	[TCP Window Update] 41008 → 9443 [ACK] Seq=730 Ack=3348867 Win=79104 Len=0 TSval=1295599502 TSecr=1295571488
953	91.694123	127.0.0.1	127.0.0.1	HTTP	37248	HTTP/1.1 200 OK [Malformed Packet]
954	91.734072	127.0.0.1	127.0.0.1	TCP	68	41008 → 9443 [ACK] Seq=730 Ack=3386048 Win=159360 Len=0 TSval=1295599542 TSecr=1295599502
2691	136.709992	127.0.0.1	127.0.0.1	TCP	68	41008 → 9443 [RST, ACK] Seq=730 Ack=3386048 Win=392960 Len=0 TSval=1295644517 TSecr=1295599502

刚开始也只看到 sangfor_waf 向 9443 端口发送 RST 包，结合客户端先发 FIN，难道是客户端发了 FIN，然后 sangfor_waf 才发送 RST 包？然后连接中断？为什么客户端会发 FIN 包。

同时在 firefox 进行模拟，一直没有出现，所以结论是 chrome 版本比较低，客户端问题，先发 FIN 包吗？

4. 开始解答

客户端抓包：



23436	117.677879	200.200.10.220	10.32.32.86	TCP	1514	443 → 25856 [ACK] Seq=2620369 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassemble
23437	117.678036	10.32.32.86	200.200.10.220	TCP	54	25856 → 443 [ACK] Seq=1318 Ack=2621829 Win=99328 Len=0
23438	117.678110	200.200.10.220	10.32.32.86	TCP	1514	443 → 25856 [ACK] Seq=2621829 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassemble
23439	117.678110	200.200.10.220	10.32.32.86	TCP	1514	443 → 25856 [ACK] Seq=2623289 Ack=1318 Win=17152 Len=1460 [TCP segment of a reassemble
23440	117.678110	200.200.10.220	10.32.32.86	TLSv1.2	271	Application Data, Encrypted Alert
23441	117.678155	10.32.32.86	200.200.10.220	TCP	54	25856 → 443 [ACK] Seq=1318 Ack=2624967 Win=96000 Len=0
25884	130.476308	10.32.32.86	200.200.10.220	TCP	54	[TCP Window Update] 25856 → 443 [ACK] Seq=1318 Ack=2624967 Win=99584 Len=0
26338	133.196943	10.32.32.86	200.200.10.220	TCP	54	[TCP Window Update] 25856 → 443 [ACK] Seq=1318 Ack=2624967 Win=262656 Len=0
28953	148.835111	10.32.32.86	200.200.10.220	TCP	54	25856 → 443 [FIN, ACK] Seq=1318 Ack=2624967 Win=262656 Len=0
28954	148.837924	200.200.10.220	10.32.32.86	TCP	60	443 → 25856 [ACK] Seq=2624967 Ack=1319 Win=17152 Len=0

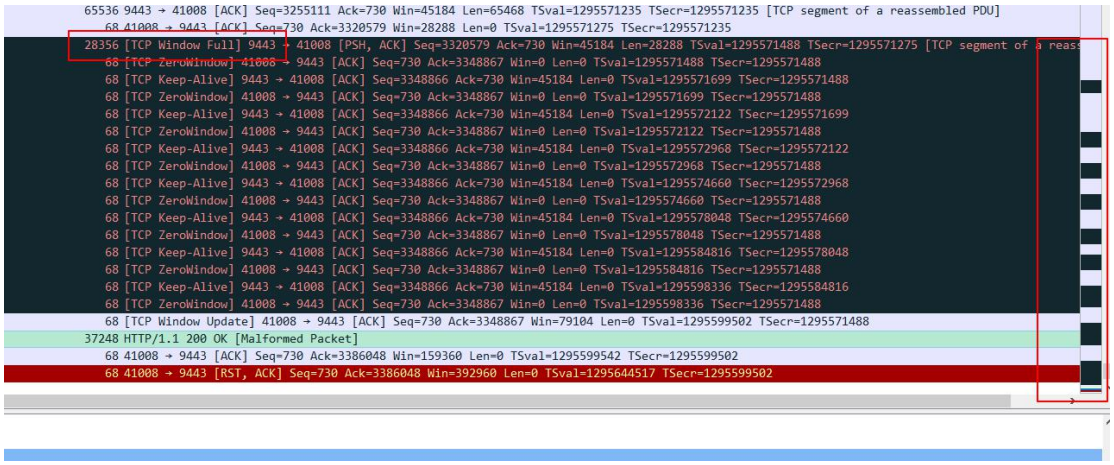

```
Acknowledgment number (raw): 615781209
0101 .... = Header Length: 20 bytes (5)
Flags: 0x019 (FIN, PSH, ACK)
  000. .... .... = Reserved: Not set
  ...0 .... .... = Nonce: Not set
  .... 0... .... = Congestion Window Reduced (CWR): Not set
  .... .0.. .... = ECN-Echo: Not set
  .... ..0. .... = Urgent: Not set
  .... ...1 .... = Acknowledgment: Set
  .... .... 1... = Push: Set
  .... .... .0.. = Reset: Not set
  .... .... ..0. = Syn: Not set
  .... .... ...1 = Fin: Set
  [TCP Flags: ·····AP···F]
```

重新仔细看一下这个抓包，这两个包的颜色有点特别，google 一下，原来 Encrypted Alert 也是中断的意思，但是没有 FIN 的标记，然后打开传输层的请求头，发现真的有 FIN 设置，所以是服务器端先发的 FIN 请求，并非客户端。

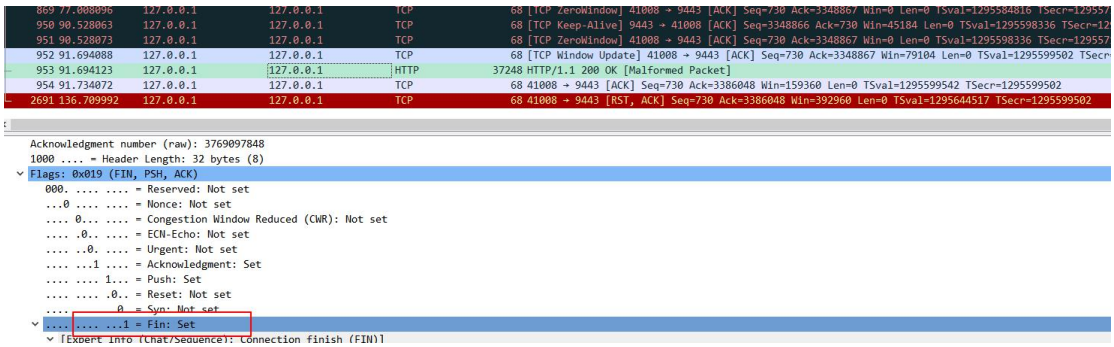
打开 sangfor_waf 的 debug 日志

可以看到是 server 端先 close 的，但是之前的抓包是 waf 发的 RST 包，然后去查看 waf 的代码，发现并没有什么异常之处。同时看到 waf 有个 monitor 超时监控的线程，是不是这个线程干的？查看相关代码，发现只是定时清理任务而已。

服务器端抓包：



可以看到 apache 这边的 tcp 滑动窗口是满的，并且每次满了后等待时间一直在增加，猜测是不是等待时间增加，维持 http 长链接然后超时？观察到 Malformed Packet 这个包有点奇怪，google 了一下，只是 wireshark 无法解析而已，并不是一个坏包。然后打开这个包的传输层的头，看一下



我日，居然也是 FIN 包，到这里已经证据确凿是 apache 先 close 的了，然后 sangfor_waf 才会发 RST 包。

5. 由第 4 步知道是 apache 先 close 的 socket 导致一系列操作，下面分析 apache，查看 apache 所有 timeout 的地方

```
SIS3.0.52.0 /sf/cfg/3.0.1/public/sn # grep -i "time" /home/fantom/apps/super/default/conf/fantom/httpd.conf
SSLSessionCacheTimeout  300
    ProxyTimeout 1200
FileEtag Mtime Size
LoadModule reqtimeout_module modules/mod_reqtimeout.so
<IfModule reqtimeout_module>
    RequestReadTimeout header=20,MinRate=500 body=200,MinRate=500
# Description:  Amount of time the server will wait for certain events before failing a request
# Syntax:   TimeOut seconds
# Default:  TimeOut 300
Timeout 1000
# Description:  Amount of time the server will wait for subsequent requests on a persistent connection
# Syntax:   KeepAliveTimeout seconds
# Default:  KeepAliveTimeout 5
KeepAliveTimeout 5
```

发现有很多处地方用到，猜测是 Timeout 或者 KeepAliveTimeout 两个参数导致

TimeOut Directive

Description:	Amount of time the server will wait for certain events before failing a request
Syntax:	`TimeOut` *seconds*
Default:	`TimeOut` 60
Context:	server config, virtual host
Status:	Core
Module:	core

The `TimeOut` directive defines the length of time Apache httpd will wait for I/O in various circumstances:

- When reading data from the client, the length of time to wait for a TCP packet to arrive if the read buffer is empty.

  For initial data on a new connection, this directive doesn't take effect until after any configured `AcceptFilter` has passed the new connection to the server.

- When writing data to the client, the length of time to wait for an acknowledgement of a packet if the send buffer is full.
- In `mod_cgi` and `mod_cgid`, the length of time to wait for any individual block of output from a CGI script.
- In `mod_ext_filter`, the length of time to wait for output from a filtering process.
- In `mod_proxy`, the default timeout value if `ProxyTimeout` is not configured.

KeepAliveTimeout Directive

Description:	Amount of time the server will wait for subsequent requests on a persistent connection
Syntax:	`KeepAliveTimeout` *num[ms]*
Default:	`KeepAliveTimeout` 5
Context:	server config, virtual host
Status:	Core
Module:	core

The number of seconds Apache httpd will wait for a subsequent request before closing the connection. By adding a postfix of ms the timeout can be also set in milliseconds. Once a request has been received, the timeout value specified by the `Timeout` directive applies.

Setting `KeepAliveTimeout` to a high value may cause performance problems in heavily loaded servers. The higher the timeout, the more server processes will be kept occupied waiting on connections with idle clients.

If `KeepAliveTimeout` is **not** set for a name-based virtual host, the value of the first defined virtual host best matching the local IP and port will be used.

老实说，看不出来是哪个的问题，那就一个一个来。测试发现，把 Timeout 时间改大，是可以解决这个问题的

加载一个 30M 左右的 js，居然用了 44min，不像之前那样子 fail 了，到这里，已经解决这个问题了。