8-SRS4.0 DTLS 握手

零声学院：音视频高级课程：https://ke.qq.com/course/468797?tuin=137bb271

本文基于 微信公众号 视频云技术 的《详解 WebRTC 传输安全机制：一文读懂 DTLS 协议》进行修改 。

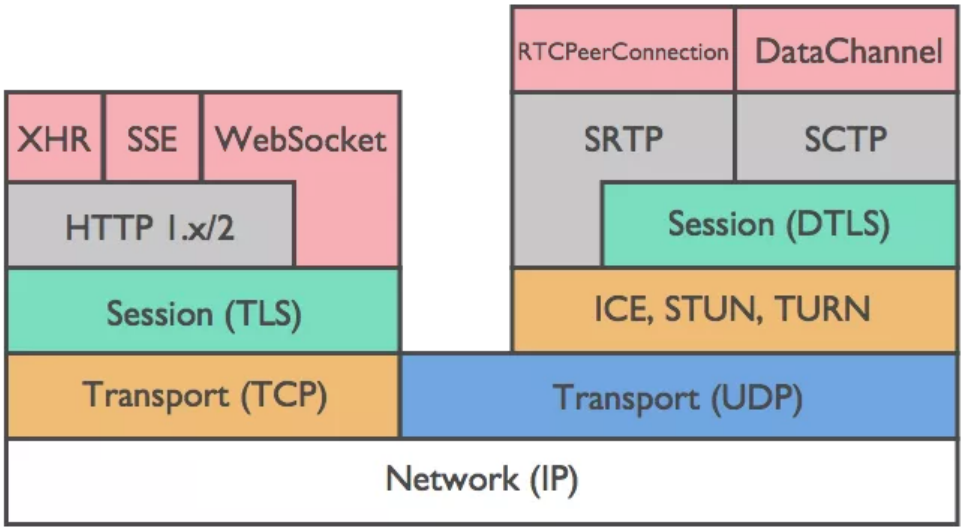
重点：

1. 掌握DTLS在数据传输的位置极其作用，
2. 了解DTLS在srs源码中的实现。

本文网页版本：https://www.yuque.com/docs/share/dc5183a0-ebba-4c2a-a8d2-0d09d6c69fb1?#（密码：oytu） 《8-SRS4.0 DTLS 握手》

原文：来之视频云技术 公众号。https://mp.weixin.qq.com/s/tHW6sWRZUzkOtI2BsRbV1w
作者 | 进学
审校 | 泰一

DTLS(Datagram Transport Layer Security数据报传输层安全) 是基于 UDP 场景下数据包可能丢失或重新排序的现实情况下，为 UDP 定制和改进的 TLS 协议。在 WebRTC 中使用 DTLS 的地方包括两部分：协商和管理 SRTP 密钥和为 DataChannel 提供加密通道。



这个图是DTLS在WebRTC网络协议栈中的位置，需要注意的是DTLS绿色部分并没有完全贯穿在ICE和SRTP/SCTP之间，这是为什么呢?

在WeRTC中使用DTLS的地方包括两部分， 一是Datachannel 数据通道，一个是音视频媒体通道。 在Datachannel数据通道中，WebRTC完全使用DTLS来进行协商和加解密，在音视频通道中WebRTC使用SRTP来进行数据的加解密，DTLS的作用仅仅是用来做密钥交换，密钥交换完就没有DTLS什么事情了，RTP/RTCP的数据加解密就交给了SRTP.

所以常说的WebRTC使用DTLS来进行加解密是不严谨的，**DTLS只是用来做密钥交换**。

- Datachannel 数据通道。在 Datachannel 数据通道中，WebRTC 完全使用 DTLS 来进行协商和加解密MediaChannel 媒体通道。
- 在RtcPeerConnection媒体通道中 WebRTC 使用 SRTP 来进行数据的加解密，DTLS 的作用仅仅是用来做密钥交换，RTP/RTCP 的数据为了与历史设备兼容性的考虑，完全通过 SRTP 来实现。

本文结合实际数据包分析 WebRTC 使用 DTLS 进行 SRTP **密钥协商的流程**。并对在实际项目中使用 DTLS 遇到的问题进行总结。

0 非对称加密

因为公钥加密的数据只有它相对应的私钥可以解开,所以你可以把公钥给人和人,让他加密他想要传送给你的数据,这个数据只有到了有私钥的你这里,才可以解开成有用的数据,其他人就是得到了,也看懂内容.同理,

如果你用你的私钥对数据进行签名,那这个数据就只有配对的公钥可以解开,有这个私钥的只有你,所以如果配对的公钥解开了数据,就说明这数据是你发的,相反,则不是.这个被称为签名.

实际应用中，**一般都是和对方交换公钥**，然后你要发给对方的数据,用他的公钥加密,他得到后用他的私钥解密,他要发给你的数据,用你的公钥加密,你得到后用你的私钥解密,这样最大程度保证了安全性.

0 证书、公钥、私钥

公钥私钥：公钥和私钥组成一个密钥对，必须配对使用。一般公钥公开，私钥自己保留。

- 公钥加密，私钥解密，一般用于传输数据；
- 私钥加密，公钥解密，一般用于数字签名、验证身份。

证书：全称是公钥证书，由第三方机构CA颁发。CA利用自己的私钥对真正的公钥施加数字签名并生成证书，客户拿到证书后，通过CA的公钥来对证书解密，拿到真正的公钥。

0 SRS DTLS握手和加解密

SRS 对应的处理源文件：

- srs_app_rtc_dtls.hpp
- srs_app_rtc_dtls.cpp

判断是不是dtls包srs_is_dtls

dtls

主要调用openssl实现DTLS

- 高层api：SrsDtls::SrsDtls
- 初始化： SrsDtlsImpl::initialize
- 数据源：SrsDtlsImpl::do_on_dtls(char* data, int nb_data)
- 握手：SrsDtlsImpl::do_handshake()
  - int r0 = SSL_do_handshake(dtls);
- 握手结果
  - int r1 = SSL_get_error(dtls, r0);

Openssl EVP(high-level cryptographic functions[1])提供了丰富的密码学中的各种函数。Openssl 中实现了各种对称算法、摘要算法以及签名/验签算法。EVP 函数将这些具体的算法进行了封装。 EVP 系列的函数的声明包含在”evp.h”里面，这是一系列封装了openssl>加密库里面所有算法的函数。通过这样的统一的封装，使得只需要在初始化参数的时候做很少的改变，就可以使用相同的代码但采用不同的加密算法进行数据的加密和解密。

EVP系列函数主要封装了加密、摘要、编码三大类型的算法，使用算法前需要调用OpenSSL_add_all_algorithms函数。

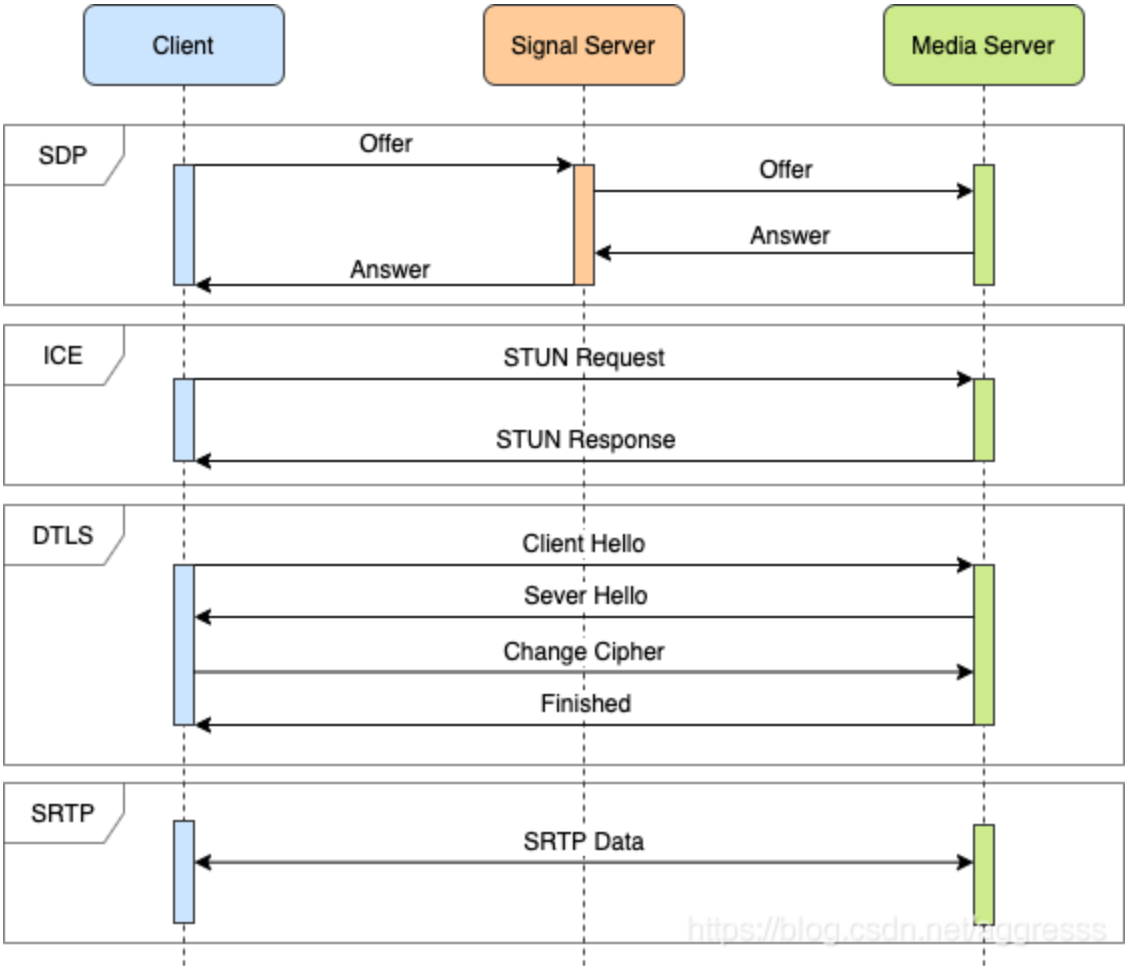
其中以加密算法与摘要算法为基本，公开密钥算法是对数据加密采用了对称加密算法，对密钥采用非对称加密（公钥加密，私钥解密）。数字签名是非对称算法（私钥签名，公钥认证）。

大致交互流程

webrtc使用sdp除了描述媒体类型，还有一些额外的字段来描述ice的连接候选项。

- 浏览器首先发送自己的offer sdp到SFU服务器，然后服务器返回answer sdp，返回的answer sdp包含ice 候选项和dtls相关的信息。
- 浏览器客户端收到sdp之后会首先进行ice连接（即一条udp链路）。
- 连接建立之后，发起dtls交互，得到远端和本地的srtp的key（分别用于解密远端到来的srtp和加密本地即将发出去的rtp数据包）。
- 然后就可以接收和发送rtp，rtcp数据了，发送之前要进行srtp加密，然后通过ice的连接发送出去。
- dtls 和 srtp 的数据包都是通过ice的udp连接进行传输的。

WebRTC交互逻辑



SRS 4.x交互DTLS握手逻辑

框图连接：https://www.processon.com/view/link/610cfda80e3e7407191c75cc

函数调用栈分析

SrsDtls::SrsDtls 创建连接的时候调用

#0 SrsDtls::SrsDtls (this=0x55555620f950, callback=0x55555620f720) at src/app/srs_app_rtc_dtls.cpp:1017

#1 0x00005555557da271 in SrsSecurityTransport::SrsSecurityTransport (this=0x55555620f720,

s=0x55555624b3f0) at src/app/srs_app_rtc_conn.cpp:95

#2 0x00005555557e33b2 in SrsRtcConnection::SrsRtcConnection (this=0x55555624b3f0, s=0x5555560b51c0,

cid=...) at src/app/srs_app_rtc_conn.cpp:1687

#3 0x000055555581bea1 in SrsRtcServer::create_session (this=0x5555560b51c0, ruc=0x55555620afa0,

local_sdp=..., psession=0x55555620a908) at src/app/srs_app_rtc_server.cpp:482

#4 0x0000555555834c03 in SrsGoApiRtcPublish::do_serve_http (this=0x5555561b7800, w=0x55555620b640,
    r=0x55555624bdd0, res=0x55555624ef80) at src/app/srs_app_rtc_api.cpp:413
#5 0x00005555558339c6 in SrsGoApiRtcPublish::serve_http (this=0x5555561b7800, w=0x55555620b640,
    r=0x55555624bdd0) at src/app/srs_app_rtc_api.cpp:287
#6 0x0000555555697a5d in SrsHttpServeMux::serve_http (this=0x5555560b4870, w=0x55555620b640,
    r=0x55555624bdd0) at src/protocol/srs_http_stack.cpp:730
#7 0x000055555569884e in SrsHttpCorsMux::serve_http (this=0x55555624ec40, w=0x55555620b640,
    r=0x55555624bdd0) at src/protocol/srs_http_stack.cpp:878
#8 0x000055555577b1e4 in SrsHttpConn::process_request (this=0x55555624f210, w=0x55555620b640,
    r=0x55555624bdd0, rid=1) at src/app/srs_app_http_conn.cpp:250
#9 0x000055555577ae29 in SrsHttpConn::process_requests (this=0x55555624f210, preq=0x55555620b718)
    at src/app/srs_app_http_conn.cpp:223
#10 0x000055555577a9af in SrsHttpConn::do_cycle (this=0x55555624f210)
    at src/app/srs_app_http_conn.cpp:177
#11 0x000055555577a3a4 in SrsHttpConn::cycle (this=0x55555624f210) at src/app/srs_app_http_conn.cpp:122
#12 0x000055555571380a in SrsFastCoroutine::cycle (this=0x55555620fe80) at src/app/srs_app_st.cpp:270
#13 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x55555620fe80) at src/app/srs_app_st.cpp:285
#14 0x000055555583bd54 in _st_thread_main () at sched.c:363
#15 0x000055555583c5f0 in st_thread_create (start=0x555555713886 <SrsFastCoroutine::pfn(void*)>,
    arg=0x55555620fe80, joinable=1, stk_size=65536) at sched.c:694
Backtrace stopped: previous frame inner to this frame (corrupt stack?)

SrsDtlsImpl::initialize 连接初始化的时候调用

#0 SrsDtlsImpl::initialize (this=0x55555624c370, version="auto", role="passive")   at src/app/srs_app_rtc_dtls.cpp:435 **作为 sever，等待发起协商**
#1 0x000055555580d222 in SrsDtlsServerImpl::initialize (this=0x55555624c370, version="auto",
    role="passive") at src/app/srs_app_rtc_dtls.cpp:919

#2 0x000055555580d6e2 in SrsDtls::initialize (this=0x55555620f950, role="passive", version="auto")
    at src/app/srs_app_rtc_dtls.cpp:1037
#3 0x00005555557da465 in SrsSecurityTransport::initialize (this=0x55555620f720, cfg=0x55555624bab0)
    at src/app/srs_app_rtc_conn.cpp:109
#4 0x00005555557e4ba5 in **SrsRtcConnection::initialize** (this=0x55555624b3f0, r=0x55555620f970,
    dtls=true, srtp=true, username="65641992:6qo3") at src/app/srs_app_rtc_conn.cpp:1954
#5 0x000055555581c97b in SrsRtcServer::do_create_session (this=0x5555560b51c0, ruc=0x55555620afa0,
    local_sdp=..., session=0x55555624b3f0) at src/app/srs_app_rtc_server.cpp:571
#6 0x000055555581bec6 in SrsRtcServer::create_session (this=0x5555560b51c0, ruc=0x55555620afa0,
    local_sdp=..., psession=0x55555620a908) at src/app/srs_app_rtc_server.cpp:483
#7 0x0000555555834c03 in SrsGoApiRtcPublish::do_serve_http (this=0x5555561b7800, w=0x55555620b640,
    r=0x55555624bdd0, res=0x55555624ef80) at src/app/srs_app_rtc_api.cpp:413
#8 0x0000555558339c6 in SrsGoApiRtcPublish::serve_http (this=0x5555561b7800, w=0x55555620b640,
    r=0x55555624bdd0) at src/app/srs_app_rtc_api.cpp:287
#9 0x0000555555697a5d in SrsHttpServeMux::serve_http (this=0x5555560b4870, w=0x55555620b640,
    r=0x55555624bdd0) at src/protocol/srs_http_stack.cpp:730
#10 0x000055555569884e in SrsHttpCorsMux::serve_http (this=0x55555624ec40, w=0x55555620b640,
    r=0x55555624bdd0) at src/protocol/srs_http_stack.cpp:878
#11 0x000055555577b1e4 in SrsHttpConn::process_request (this=0x55555624f210, w=0x55555620b640,
    r=0x55555624bdd0, rid=1) at src/app/srs_app_http_conn.cpp:250
#12 0x000055555577ae29 in SrsHttpConn::process_requests (this=0x55555624f210, preq=0x55555620b718)
    at src/app/srs_app_http_conn.cpp:223
#13 0x000055555577a9af in SrsHttpConn::do_cycle (this=0x55555624f210)
    at src/app/srs_app_http_conn.cpp:177
#14 0x000055555577a3a4 in SrsHttpConn::cycle (this=0x55555624f210) at src/app/srs_app_http_conn.cpp:122
#15 0x000055555571380a in SrsFastCoroutine::cycle (this=0x55555620fe80) at src/app/srs_app_st.cpp:270
---Type <return> to continue, or q <return> to quit---

#16 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x55555620fe80) at src/app/srs_app_st.cpp:285
#17 0x000055555583bd54 in _st_thread_main () at sched.c:363
#18 0x000055555583c5f0 in st_thread_create (start=0x555555713886 <SrsFastCoroutine::pfn(void*)>,
    arg=0x55555620fe80, joinable=1, stk_size=65536) at sched.c:694
Backtrace stopped: previous frame inner to this frame (corrupt stack?)

SrsDtlsImpl::do_on_dtls

#0 SrsDtlsImpl::do_on_dtls (this=0x55555624c370, data=0x5555561ca660 "\026\376\377", nb_data=159)   at src/app/srs_app_rtc_dtls.cpp:500
#1 0x000055555580b117 in SrsDtlsImpl::on_dtls (this=0x55555624c370,
    data=0x5555561ca660 "\026\376\377", nb_data=159) at src/app/srs_app_rtc_dtls.cpp:491
#2 0x000055555580d7c7 in SrsDtls::on_dtls (this=0x55555620f950, data=0x5555561ca660 "\026\376\377",
    nb_data=159) at src/app/srs_app_rtc_dtls.cpp:1047
#3 0x00005555557da5e9 in SrsSecurityTransport::on_dtls (this=0x55555620f720,
    data=0x5555561ca660 "\026\376\377", nb_data=159) at src/app/srs_app_rtc_conn.cpp:140
#4 0x00005555557e500d in SrsRtcConnection::on_dtls (this=0x55555624b3f0,
    data=0x5555561ca660 "\026\376\377", nb_data=159) at src/app/srs_app_rtc_conn.cpp:1997
#5 0x000055555581b942 in SrsRtcServer::on_udp_packet (this=0x5555560b51c0, skt=0x7ffff7fdfb70)
    at src/app/srs_app_rtc_server.cpp:435
#6 0x00005555557b86ac in SrsUdpMuxListener::cycle (this=0x5555560fdfb0)
    at src/app/srs_app_listener.cpp:636
#7 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561b75f0) at src/app/srs_app_st.cpp:270
#8 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561b75f0) at src/app/srs_app_st.cpp:285
#9 0x000055555583bd54 in _st_thread_main () at sched.c:363
#10 0x000055555583c5f0 in st_thread_create (start=0x7ffff6e689d8 <__libc_multiple_threads>,
    arg=0x5b0000006e, joinable=119, stk_size=124) at sched.c:694
#11 0x00007ffff6e63c40 in ?? () f

SrsDtlsImpl::do_handshake

#0 SrsDtlsImpl::do_handshake (this=0x55555624c370) at src/app/srs_app_rtc_dtls.cpp:578#1 0x000055555580b3bc in SrsDtlsImpl::do_on_dtls (this=0x55555624c370,
    data=0x5555561ca660 "\026\376\377", nb_data=159) at src/app/srs_app_rtc_dtls.cpp:528

#2 0x000055555580b117 in SrsDtlsImpl::on_dtls (this=0x55555624c370,
    data=0x5555561ca660 "\026\376\377", nb_data=159) at src/app/srs_app_rtc_dtls.cpp:491
#3 0x000055555580d7c7 in SrsDtls::on_dtls (this=0x55555620f950, data=0x5555561ca660
"\026\376\377",
    nb_data=159) at src/app/srs_app_rtc_dtls.cpp:1047
#4 0x00005555557da5e9 in SrsSecurityTransport::on_dtls (this=0x55555620f720,
    data=0x5555561ca660 "\026\376\377", nb_data=159) at src/app/srs_app_rtc_conn.cpp:140
#5 0x00005555557e500d in SrsRtcConnection::on_dtls (this=0x55555624b3f0,
    data=0x5555561ca660 "\026\376\377", nb_data=159) at
src/app/srs_app_rtc_conn.cpp:1997
#6 0x000055555581b942 in SrsRtcServer::on_udp_packet (this=0x5555560b51c0,
skt=0x7ffff7fdfb70)
    at src/app/srs_app_rtc_server.cpp:435
#7 0x00005555557b86ac in SrsUdpMuxListener::cycle (this=0x5555560fdfb0)
    at src/app/srs_app_listener.cpp:636
#8 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561b75f0) at
src/app/srs_app_st.cpp:270
#9 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561b75f0) at
src/app/srs_app_st.cpp:285
#10 0x000055555583bd54 in _st_thread_main () at sched.c:363
#11 0x000055555583c5f0 in st_thread_create (start=0x7ffff6e689d8
<__libc_multiple_threads>,
    arg=0x5b0000006e, joinable=119, stk_size=124) at sched.c:694
#12 0x00007ffff6e63c40 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
#13 0x0000000000000000 in ?? ()

SrsDtlsServerImpl::on_handshake_done

#0 SrsDtlsServerImpl::on_handshake_done (this=0x5555561eb170) at
src/app/srs_app_rtc_dtls.cpp:949#1 0x000055555580bb08 in SrsDtlsImpl::do_handshake
(this=0x5555561eb170)
    at src/app/srs_app_rtc_dtls.cpp:619
#2 0x000055555580b3bc in SrsDtlsImpl::do_on_dtls (this=0x5555561eb170,
    data=0x5555561ca660 "\026\376", <incomplete sequence \375>, nb_data=576)
    at src/app/srs_app_rtc_dtls.cpp:528
#3 0x000055555580b117 in SrsDtlsImpl::on_dtls (this=0x5555561eb170,
    data=0x5555561ca660 "\026\376", <incomplete sequence \375>, nb_data=576)
    at src/app/srs_app_rtc_dtls.cpp:491
#4 0x000055555580d7c7 in SrsDtls::on_dtls (this=0x55555620f950,
    data=0x5555561ca660 "\026\376", <incomplete sequence \375>, nb_data=576)

    at src/app/srs_app_rtc_dtls.cpp:1047
#5 0x00005555557da5e9 in SrsSecurityTransport::on_dtls (this=0x5555562502b0,
    data=0x5555561ca660 "\026\376", <incomplete sequence \375>, nb_data=576)
    at src/app/srs_app_rtc_conn.cpp:140
#6 0x00005555557e500d in SrsRtcConnection::on_dtls (this=0x55555624b3f0,
    data=0x5555561ca660 "\026\376", <incomplete sequence \375>, nb_data=576)
    at src/app/srs_app_rtc_conn.cpp:1997
#7 0x000055555581b942 in SrsRtcServer::on_udp_packet (this=0x5555560b51c0,
skt=0x7ffff7fdfb70)
    at src/app/srs_app_rtc_server.cpp:435
#8 0x00005555557b86ac in SrsUdpMuxListener::cycle (this=0x5555560fdfb0)
    at src/app/srs_app_listener.cpp:636
#9 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561b75f0) at
src/app/srs_app_st.cpp:270
#10 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561b75f0) at
src/app/srs_app_st.cpp:285
#11 0x000055555583bd54 in _st_thread_main () at sched.c:363
#12 0x000055555583c5f0 in st_thread_create (start=0x7ffff6e689d8
<__libc_multiple_threads>,
    arg=0x5b0000006e, joinable=119, stk_size=124) at sched.c:694
#13 0x00007ffff6e63c40 in ?? () from /lib/x86_64-linux-gnu/libc.so.6

SrsSecurityTransport::unprotect_rtp RTP解包

#0 SrsSecurityTransport::unprotect_rtp (this=0x555556214130, packet=0x5555561ca660,
nb_plaintext=0x7ffff7fdf72c) at src/app/srs_app_rtc_conn.cpp:206
#1 0x00005555557e105d in SrsRtcPublishStream::on_rtp (this=0x555556219e10,
    data=0x5555561ca660 "\220}\363\350\336O\032Ⱶ\376{\260\276", <incomplete sequence
\336>,
    nb_data=1201) at src/app/srs_app_rtc_conn.cpp:1255
#2 0x00005555557e5dd0 in SrsRtcConnection::on_rtp (this=0x555556211d00,
    data=0x5555561ca660 "\220}\363\350\336O\032Ⱶ\376{\260\276", <incomplete sequence
\336>,
    nb_data=1201) at src/app/srs_app_rtc_conn.cpp:2148
#3 0x000055555581b85e in SrsRtcServer::on_udp_packet (this=0x5555560b51c0,
skt=0x7ffff7fdfb70)
    at src/app/srs_app_rtc_server.cpp:419
#4 0x00005555557b86ac in SrsUdpMuxListener::cycle (this=0x5555560fdfb0)
    at src/app/srs_app_listener.cpp:636
#5 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561b75f0) at
src/app/srs_app_st.cpp:270

#6 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561b75f0) at src/app/srs_app_st.cpp:285
#7 0x000055555583bd54 in _st_thread_main () at sched.c:363
#8 0x000055555583c5f0 in st_thread_create (start=0x7ffff6e689d8 <__libc_multiple_threads>,
    arg=0x5b0000006e, joinable=119, stk_size=124) at sched.c:694
#9 0x00007ffff6e63c40 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
#10 0x0000000000000000 in ?? ()

[2021-08-05 21:10:43.109][Trace][26625][1g33135q] RTC: session address init 113.246.105.182:18058[2021-08-05 21:10:43.109][Trace][26625][1g33135q] RTC: session STUN done, waiting DTLS handshake.
[2021-08-05 21:10:43.109][Trace][26625][83081r55] <- RTC RECV #9, udp 1, pps 0/0, schedule 1
[2021-08-05 21:10:43.134][Trace][26625][1g33135q] DTLS nb_data:159 收到第一次数据是什么原因?
[2021-08-05 21:10:43.135][Trace][26625][1g33135q] DTLS: State Passive RECV, done=0, arq=0/0, r0=1, r1=0, len=159, cnt=22, size=146, hs=1
[2021-08-05 21:10:43.135][Trace][26625][1g33135q] DTLS: State Passive SEND, done=0, arq=0/0, r0=-1, r1=2, len=680, cnt=22, size=82, hs=2
[2021-08-05 21:10:43.164][Trace][26625][1g33135q] DTLS nb_data:577 收到第二次数据
[2021-08-05 21:10:43.164][Trace][26625][1g33135q] DTLS: State Passive RECV, done=0, arq=0/0, r0=1, r1=0, len=577, cnt=22, size=298, hs=11
[2021-08-05 21:10:43.165][Trace][26625][1g33135q] DTLS: State Passive SEND, done=1, arq=0/0, r0=1, r1=0, len=554, cnt=22, size=466, hs=4
[2021-08-05 21:10:43.165][Trace][26625][1g33135q] RTC: DTLS handshake done.

SrsSRTP::initialize SRTP相关的key

#0 SrsSRTP::initialize (this=0x555556214080, recv_key="%`\307\071\315&L\334\362\203\270\225\204\327a\273\267蔪o\241RG\363[O*.m",

send_key="\303c\301\341%\375\244\332\346D\365\231\bu\213\301\214Y\270i\300\334\373\244\221\354\301^n\263") at src/app/srs_app_rtc_dtls.cpp:1073
#1 0x00005555557da8ee in SrsSecurityTransport::srtp_initialize (this=0x5555562143a0)
    at src/app/srs_app_rtc_conn.cpp:187
#2 0x00005555557da777 in SrsSecurityTransport::on_dtls_handshake_done (this=0x5555562143a0)
    at src/app/srs_app_rtc_conn.cpp:160

```
#3  0x000055555580d3bf in SrsDtlsServerImpl::on_handshake_done (this=0x555556213fa0)
    at src/app/srs_app_rtc_dtls.cpp:952
#4  0x000055555580bbae in SrsDtlsImpl::do_handshake (this=0x555556213fa0)
    at src/app/srs_app_rtc_dtls.cpp:619
#5  0x000055555580b43e in SrsDtlsImpl::do_on_dtls (this=0x555556213fa0,
    data=0x5555561ca790 "\026\376", <incomplete sequence \375>, nb_data=580)
    at src/app/srs_app_rtc_dtls.cpp:528
#6  0x000055555580b117 in SrsDtlsImpl::on_dtls (this=0x555556213fa0,
    data=0x5555561ca790 "\026\376", <incomplete sequence \375>, nb_data=580)
    at src/app/srs_app_rtc_dtls.cpp:491
#7  0x000055555580d86d in SrsDtls::on_dtls (this=0x555556213320,
    data=0x5555561ca790 "\026\376", <incomplete sequence \375>, nb_data=580)
    at src/app/srs_app_rtc_dtls.cpp:1047
#8  0x00005555557da5e9 in SrsSecurityTransport::on_dtls (this=0x5555562143a0,
    data=0x5555561ca790 "\026\376", <incomplete sequence \375>, nb_data=580)
    at src/app/srs_app_rtc_conn.cpp:140
#9  0x00005555557e500d in SrsRtcConnection::on_dtls (this=0x555556211e20,
    data=0x5555561ca790 "\026\376", <incomplete sequence \375>, nb_data=580)
    at src/app/srs_app_rtc_conn.cpp:1997
#10 0x000055555581b9e8 in SrsRtcServer::on_udp_packet (this=0x5555560b51c0,
skt=0x7ffff7fdfb70)
    at src/app/srs_app_rtc_server.cpp:435
#11 0x00005555557b86ac in SrsUdpMuxListener::cycle (this=0x5555560fdfb0)
---Type <return> to continue, or q <return> to quit---
    at src/app/srs_app_listener.cpp:636
#12 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561b75f0) at
src/app/srs_app_st.cpp:270
#13 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561b75f0) at
src/app/srs_app_st.cpp:285
#14 0x000055555583bdfa in _st_thread_main () at sched.c:363
#15 0x000055555583c696 in st_thread_create (start=0x7ffff6e689d8
<__libc_multiple_threads>,
    arg=0x5b0000006e, joinable=119, stk_size=124) at sched.c:694
#16 0x00007ffff6e63c40 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
#17 0x0000000000000000 in ?? ()

(gdb) bt#0  SrsDtlsCertificate::initialize (this=0x5555560afd60) at
src/app/srs_app_rtc_dtls.cpp:228
#1  0x000055555581dc57 in RtcServerAdapter::initialize (this=0x5555560b51a0)
    at src/app/srs_app_rtc_server.cpp:695
```

#2 0x00005555557d6dca in SrsHybridServer::initialize (this=0x5555560af510)
    at src/app/srs_app_hybrid.cpp:272
#3 0x000055555583b0cd in run_hybrid_server () at src/main/srs_main_server.cpp:485
#4 0x000055555583ac7e in run_directly_or_daemon () at src/main/srs_main_server.cpp:425
#5 0x00005555558395b7 in do_main (argc=3, argv=0x7fffffffe408) at
src/main/srs_main_server.cpp:219
#6 0x0000555555839777 in main (argc=3, argv=0x7fffffffe408) at
src/main/srs_main_server.cpp:228

SrsDtlsServerImpl::initialize

Breakpoint 6, SrsDtlsServerImpl::initialize (this=0x555556216360, version="auto",
role="passive")   at src/app/srs_app_rtc_dtls.cpp:916
916 {
(gdb) bt
#0 SrsDtlsServerImpl::initialize (this=0x555556216360, version="auto", role="passive")
    at src/app/srs_app_rtc_dtls.cpp:916
#1 0x000055555580d788 in SrsDtls::initialize (this=0x555556213210, role="passive",
version="auto")
    at src/app/srs_app_rtc_dtls.cpp:1037
#2 0x00005555557da465 in SrsSecurityTransport::initialize (this=0x555556214290,
cfg=0x555556211da0)
    at src/app/srs_app_rtc_conn.cpp:109
#3 0x00005555557e4ba5 in SrsRtcConnection::initialize (this=0x5555562116e0,
r=0x555556210e90,
    dtls=true, srtp=true, username="38a8m3lv:Gx2w") at src/app/srs_app_rtc_conn.cpp:1954
#4 0x000055555581ca21 in SrsRtcServer::do_create_session (this=0x5555560b51c0,
ruc=0x55555620af60,
    local_sdp=..., session=0x5555562116e0) at src/app/srs_app_rtc_server.cpp:571
#5 0x000055555581bf6c in SrsRtcServer::create_session (this=0x5555560b51c0,
ruc=0x55555620af60,
    local_sdp=..., psession=0x55555620a8c8) at src/app/srs_app_rtc_server.cpp:483
#6 0x0000555555834ca9 in SrsGoApiRtcPublish::do_serve_http (this=0x5555561b7800,
w=0x55555620b600,
    r=0x55555620d390, res=0x55555620e1d0) at src/app/srs_app_rtc_api.cpp:413
#7 0x0000555555833a6c in SrsGoApiRtcPublish::serve_http (this=0x5555561b7800,
w=0x55555620b600,
    r=0x55555620d390) at src/app/srs_app_rtc_api.cpp:287
#8 0x0000555555697a5d in SrsHttpServeMux::serve_http (this=0x5555560b4870,
w=0x55555620b600,
    r=0x55555620d390) at src/protocol/srs_http_stack.cpp:730

#9 0x000055555569884e in SrsHttpCorsMux::serve_http (this=0x5555561faa00,
w=0x55555620b600,
    r=0x55555620d390) at src/protocol/srs_http_stack.cpp:878
#10 0x000055555577b1e4 in SrsHttpConn::process_request (this=0x5555561da7d0,
w=0x55555620b600,
    r=0x55555620d390, rid=1) at src/app/srs_app_http_conn.cpp:250
#11 0x000055555577ae29 in SrsHttpConn::process_requests (this=0x5555561da7d0,
preq=0x55555620b6d8)
    at src/app/srs_app_http_conn.cpp:223
#12 0x000055555577a9af in SrsHttpConn::do_cycle (this=0x5555561da7d0)
    at src/app/srs_app_http_conn.cpp:177
#13 0x000055555577a3a4 in SrsHttpConn::cycle (this=0x5555561da7d0) at
src/app/srs_app_http_conn.cpp:122
#14 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561faa60) at
src/app/srs_app_st.cpp:270
#15 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561faa60) at
src/app/srs_app_st.cpp:285
#16 0x000055555583bdfa in _st_thread_main () at sched.c:363
---Type <return> to continue, or q <return> to quit---
#17 0x000055555583c696 in st_thread_create (start=0x555555713886
<SrsFastCoroutine::pfn(void*)>,
    arg=0x5555561faa60, joinable=1, stk_size=65536) at sched.c:694

SrsDtls::start_active_handshake

#0 SrsDtls::start_active_handshake (this=0x555556213210) at
src/app/srs_app_rtc_dtls.cpp:1042
#1 0x00005555557da4e8 in SrsSecurityTransport::start_active_handshake
(this=0x555556214290)
    at src/app/srs_app_rtc_conn.cpp:114
#2 0x00005555557e8602 in SrsRtcConnection::on_binding_request (this=0x5555562116e0,
r=0x7ffff7fdf980)
    at src/app/srs_app_rtc_conn.cpp:2647
#3 0x00005555557e4f8a in SrsRtcConnection::on_stun (this=0x5555562116e0,
skt=0x7ffff7fdfb70,
    r=0x7ffff7fdf980) at src/app/srs_app_rtc_conn.cpp:1988
#4 0x000055555581b80c in SrsRtcServer::on_udp_packet (this=0x5555560b51c0,
skt=0x7ffff7fdfb70)
    at src/app/srs_app_rtc_server.cpp:406
#5 0x00005555557b86ac in SrsUdpMuxListener::cycle (this=0x5555560fdfb0)
    at src/app/srs_app_listener.cpp:636

#6 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561b75f0) at src/app/srs_app_st.cpp:270
#7 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561b75f0) at src/app/srs_app_st.cpp:285
#8 0x000055555583bdfa in _st_thread_main () at sched.c:363
#9 0x000055555583c696 in st_thread_create (start=0x7ffff6e689d8 <__libc_multiple_threads>,
    arg=0x5b0000006e, joinable=119, stk_size=124) at sched.c:694
#10 0x00007ffff6e63c40 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
#11 0x0000000000000000 in ?? ()

SrsDtlsCertificate类

SrsDtlsCertificate::get_cert
SrsDtlsCertificate::get_public_key
SrsDtlsCertificate::get_ecdsa_key
SrsDtlsCertificate::get_fingerprint

SrsDtlsCertificate::get_fingerprint

#0 SrsDtlsCertificate::get_fingerprint[abi:cxx11]() (this=0x5555560afd60) at src/app/srs_app_rtc_dtls.cpp:384
#1 0x000055555581c438 in SrsRtcServer::do_create_session (this=0x5555560b51c0, ruc=0x55555620af60, local_sdp=...,
    session=0x555556211ca0) at src/app/srs_app_rtc_server.cpp:529
#2 0x000055555581bf6c in SrsRtcServer::create_session (this=0x5555560b51c0, ruc=0x55555620af60, local_sdp=...,
    psession=0x55555620a8c8) at src/app/srs_app_rtc_server.cpp:483
#3 0x0000555555834ca9 in SrsGoApiRtcPublish::do_serve_http (this=0x5555561b7800, w=0x55555620b600, r=0x55555620d370, res=
    0x55555620e1b0) at src/app/srs_app_rtc_api.cpp:413
#4 0x0000555555833a6c in SrsGoApiRtcPublish::serve_http (this=0x5555561b7800, w=0x55555620b600, r=0x55555620d370)
    at src/app/srs_app_rtc_api.cpp:287
#5 0x0000555555697a5d in SrsHttpServeMux::serve_http (this=0x5555560b4870, w=0x55555620b600, r=0x55555620d370)
    at src/protocol/srs_http_stack.cpp:730
#6 0x000055555569884e in SrsHttpCorsMux::serve_http (this=0x5555561fa9e0, w=0x55555620b600, r=0x55555620d370)
    at src/protocol/srs_http_stack.cpp:878

#7 0x000055555577b1e4 in SrsHttpConn::process_request (this=0x5555561da7b0, w=0x55555620b600, r=0x55555620d370, rid=1)
    at src/app/srs_app_http_conn.cpp:250
#8 0x000055555577ae29 in SrsHttpConn::process_requests (this=0x5555561da7b0, preq=0x55555620b6d8)
    at src/app/srs_app_http_conn.cpp:223
#9 0x000055555577a9af in SrsHttpConn::do_cycle (this=0x5555561da7b0) at src/app/srs_app_http_conn.cpp:177
#10 0x000055555577a3a4 in SrsHttpConn::cycle (this=0x5555561da7b0) at src/app/srs_app_http_conn.cpp:122
#11 0x000055555571380a in SrsFastCoroutine::cycle (this=0x5555561faa40) at src/app/srs_app_st.cpp:270
#12 0x00005555557138a6 in SrsFastCoroutine::pfn (arg=0x5555561faa40) at src/app/srs_app_st.cpp:285
#13 0x000055555583bdfa in _st_thread_main () at sched.c:363
#14 0x000055555583c696 in st_thread_create (start=0x555555713886 <SrsFastCoroutine::pfn(void*)>, arg=0x5555561faa40,
    joinable=1, stk_size=65536) at sched.c:694

SrsDtlsCertificate::get_cert

#0 SrsDtlsCertificate::get_cert (this=0x5555560afd60) at src/app/srs_app_rtc_dtls.cpp:369#1 0x000055555580a0aa in srs_build_dtls_ctx (version=SrsDtlsVersionAuto, role="passive")
    at src/app/srs_app_rtc_dtls.cpp:178
#2 0x000055555580aecd in SrsDtlsImpl::initialize (this=0x555556213aa0, version="auto", role="passive")
    at src/app/srs_app_rtc_dtls.cpp:446
#3 0x000055555580d2c8 in SrsDtlsServerImpl::initialize (this=0x555556213aa0, version="auto", role="passive")
    at src/app/srs_app_rtc_dtls.cpp:919
#4 0x000055555580d788 in SrsDtls::initialize (this=0x5555562131a0, role="passive", version="auto")
    at src/app/srs_app_rtc_dtls.cpp:1037
#5 0x00005555557da465 in SrsSecurityTransport::initialize (this=0x555556214250, cfg=0x555556212360)
    at src/app/srs_app_rtc_conn.cpp:109
#6 0x00005555557e4ba5 in SrsRtcConnection::initialize (this=0x555556211ca0, r=0x555556210e60, dtls=true, srtp=true,
    username="0t136d95:fUhd") at src/app/srs_app_rtc_conn.cpp:1954

#7 0x000055555581ca21 in SrsRtcServer::do_create_session (this=0x5555560b51c0, ruc=0x55555620af60, local_sdp=...,
    session=0x555556211ca0) at src/app/srs_app_rtc_server.cpp:571
#8 0x000055555581bf6c in SrsRtcServer::create_session (this=0x5555560b51c0, ruc=0x55555620af60, local_sdp=...,
    psession=0x55555620a8c8) at src/app/srs_app_rtc_server.cpp:483
#9 0x0000555555834ca9 in SrsGoApiRtcPublish::do_serve_http (this=0x5555561b7800, w=0x55555620b600, r=0x55555620d370,
    res=0x55555620e1b0) at src/app/srs_app_rtc_api.cpp:413
#10 0x0000555555833a6c in SrsGoApiRtcPublish::serve_http (this=0x5555561b7800, w=0x55555620b600, r=0x55555620d370)
    at src/app/srs_app_rtc_api.cpp:287
#11 0x0000555555697a5d in SrsHttpServeMux::serve_http (this=0x5555560b4870, w=0x55555620b600, r=0x55555620d370)
    at src/protocol/srs_http_stack.cpp:730
#12 0x000055555569884e in SrsHttpCorsMux::serve_http (this=0x5555561fa9e0, w=0x55555620b600, r=0x55555620d370)
    at src/protocol/srs_http_stack.cpp:878
#13 0x000055555577b1e4 in SrsHttpConn::process_request (this=0x5555561da7b0, w=0x55555620b600, r=0x55555620d370, rid=1)
    at src/app/srs_app_http_conn.cpp:250
#14 0x000055555577ae29 in SrsHttpConn::process_requests (this=0x5555561da7b0, preq=0x55555620b6d8)
    at src/app/srs_app_http_conn.cpp:223

SrsDtlsCertificate::get_public_key

Breakpoint 5, SrsDtlsCertificate::get_public_key (this=0x5555560afd60) at src/app/srs_app_rtc_dtls.cpp:374374   return dtls_pkey;
(gdb) bt
#0 SrsDtlsCertificate::get_public_key (this=0x5555560afd60) at src/app/srs_app_rtc_dtls.cpp:374
#1 0x000055555580a0ef in srs_build_dtls_ctx (version=SrsDtlsVersionAuto, role="passive")
    at src/app/srs_app_rtc_dtls.cpp:179
#2 0x000055555580aecd in SrsDtlsImpl::initialize (this=0x555556213aa0, version="auto", role="passive")
    at src/app/srs_app_rtc_dtls.cpp:446
#3 0x000055555580d2c8 in SrsDtlsServerImpl::initialize (this=0x555556213aa0, version="auto", role="passive")
    at src/app/srs_app_rtc_dtls.cpp:919

#4 0x000055555580d788 in SrsDtls::initialize (this=0x5555562131a0, role="passive", version="auto")
    at src/app/srs_app_rtc_dtls.cpp:1037
#5 0x00005555557da465 in SrsSecurityTransport::initialize (this=0x555556214250, cfg=0x555556212360)
    at src/app/srs_app_rtc_conn.cpp:109
#6 0x00005555557e4ba5 in SrsRtcConnection::initialize (this=0x555556211ca0, r=0x555556210e60, dtls=true, srtp=true,
    username="0t136d95:fUhd") at src/app/srs_app_rtc_conn.cpp:1954
#7 0x000055555581ca21 in SrsRtcServer::do_create_session (this=0x5555560b51c0, ruc=0x55555620af60, local_sdp=...,
    session=0x555556211ca0) at src/app/srs_app_rtc_server.cpp:571
#8 0x000055555581bf6c in SrsRtcServer::create_session (this=0x5555560b51c0, ruc=0x55555620af60, local_sdp=...,
    psession=0x55555620a8c8) at src/app/srs_app_rtc_server.cpp:483
#9 0x0000555555834ca9 in SrsGoApiRtcPublish::do_serve_http (this=0x5555561b7800, w=0x55555620b600, r=0x55555620d370,
    res=0x55555620e1b0) at src/app/srs_app_rtc_api.cpp:413
#10 0x0000555555833a6c in SrsGoApiRtcPublish::serve_http (this=0x5555561b7800, w=0x55555620b600, r=0x55555620d370)
    at src/app/srs_app_rtc_api.cpp:287
#11 0x0000555555697a5d in SrsHttpServeMux::serve_http (this=0x5555560b4870, w=0x55555620b600, r=0x55555620d370)
    at src/protocol/srs_http_stack.cpp:730
#12 0x000055555569884e in SrsHttpCorsMux::serve_http (this=0x5555561fa9e0, w=0x55555620b600, r=0x55555620d370)
    at src/protocol/srs_http_stack.cpp:878
#13 0x000055555577b1e4 in SrsHttpConn::process_request (this=0x5555561da7b0, w=0x55555620b600, r=0x55555620d370, rid=1)
    at src/app/srs_app_http_conn.cpp:250
#14 0x000055555577ae29 in SrsHttpConn::process_requests (this=0x5555561da7b0, preq=0x55555620b6d8)
    at src/app/srs_app_http_conn.cpp:223

01 DTLS 协议简介

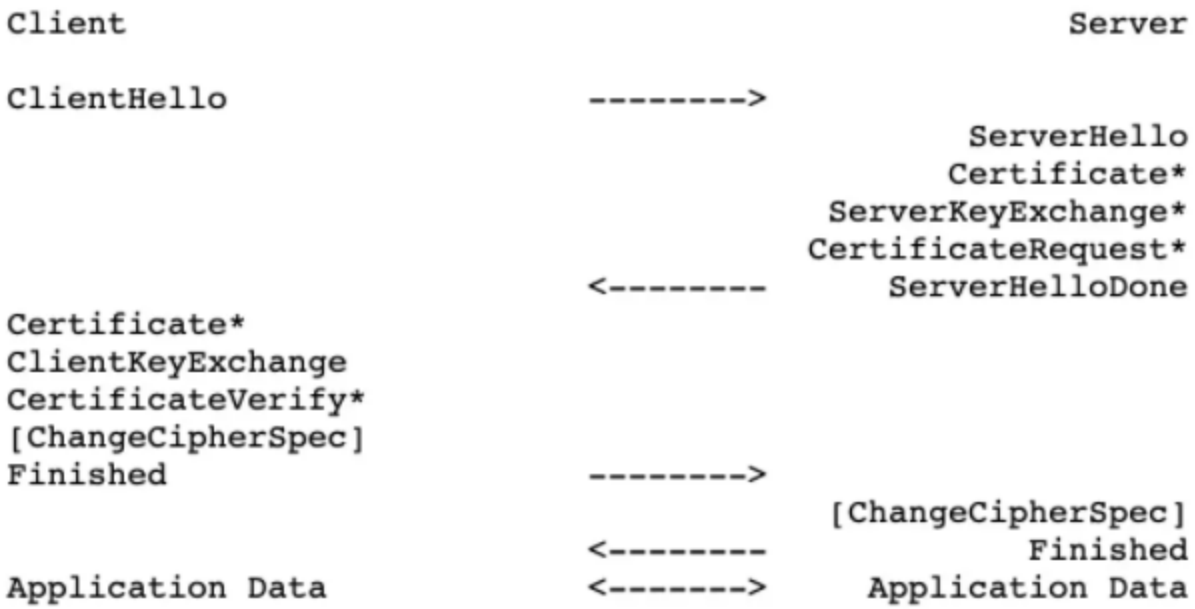
在分析 DTLS 在 WebRTC 中的应用之前，先介绍下 DTLS 协议的基本原理。

DTLS 协议由两层组成: **Record 协议 和 Handshake 协议。**

1. Record 协议: 使用对称密钥对传输数据进行加密，并使用 HMAC 对数据进行完整性校验，实现了数据的安全传输。
2. Handshake 协议：使用非对称加密算法，完成 Record 协议使用的对称密钥的协商。

1. HandShake

TLS 握手协议流程如下，参考 RFC5246 (https://www.rfc-editor.org/rfc/rfc5246.html)**The Transport Layer Security (TLS) Protocol Version 1.2**。



DTLS 握手协议流程如下，参考 RFC6347(https://www.rfc-editor.org/rfc/rfc6347.html) **Datagram Transport Layer Security Version 1.2**。

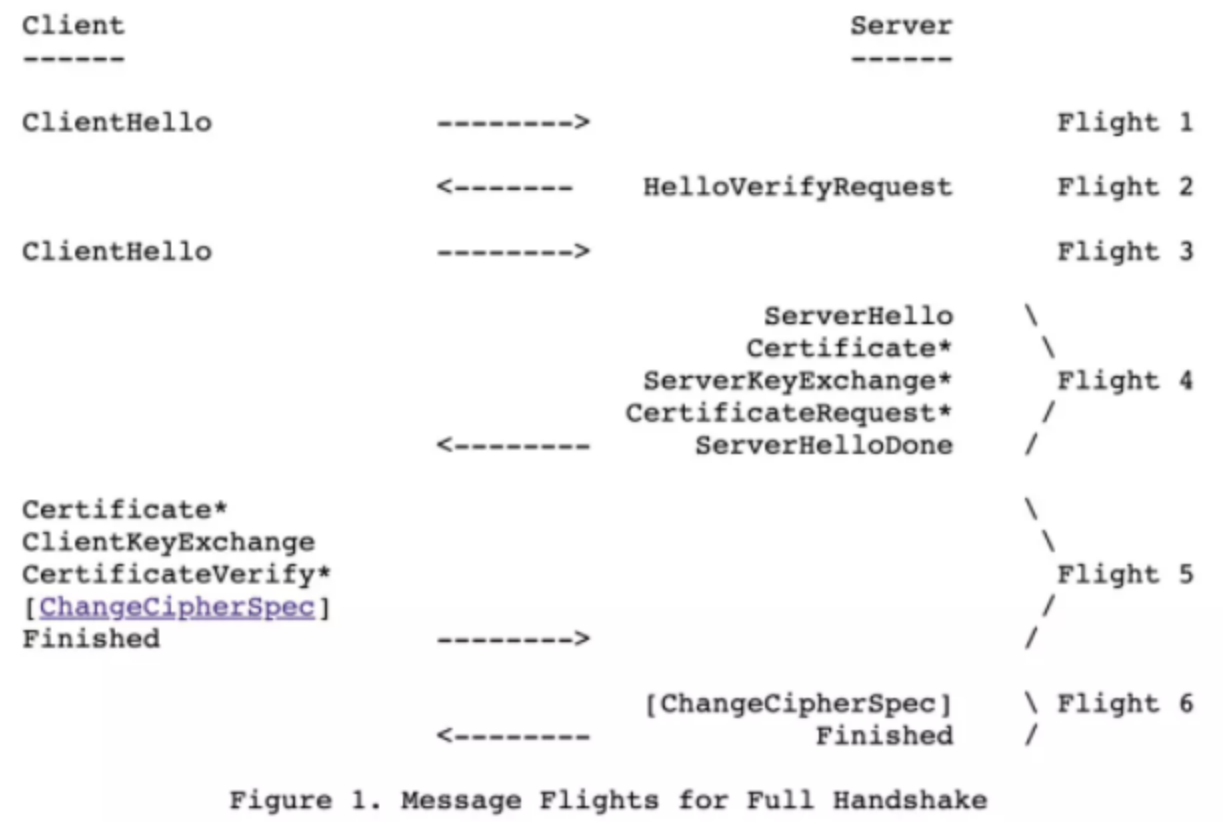
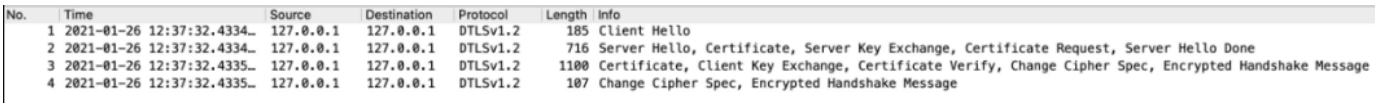


Figure 1. Message Flights for Full Handshake



No.	Time	Source	Destination	Protocol	Length	Info
1	2021-01-26 12:37:32.4334…	127.0.0.1	127.0.0.1	DTLSv1.2	185	Client Hello
2	2021-01-26 12:37:32.4334…	127.0.0.1	127.0.0.1	DTLSv1.2	716	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
3	2021-01-26 12:37:32.4335…	127.0.0.1	127.0.0.1	DTLSv1.2	1100	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message
4	2021-01-26 12:37:32.4335…	127.0.0.1	127.0.0.1	DTLSv1.2	107	Change Cipher Spec, Encrypted Handshake Message

TLS 和 DTLS 的握手过程基本上是一致的，差别以及特别说明如下：

- DTLS 中 HelloVerifyRequest 是为防止 DoS 攻击增加的消息。
- TLS 没有发送 CertificateRequest，这个也不是必须的，是反向验证即服务器验证客户端。
- DTLS 的 RecordLayer 新增了 SequenceNumber 和 Epoch，以及 ClientHello 中新增了 Cookie，以及 Handshake 中新增了 Fragment 信息（防止超过 UDP 的 MTU），都是为了适应 UDP 的丢包以及容易被攻击做的改进。参考 RFC 6347。

- DTLS 最后的 Alert 是将客户端的 Encrypted Alert 消息，解密之后直接响应给客户端的，实际上 Server 应该回应加密的消息，这里我们的服务器回应明文是为了解析客户端加密的那个 Alert 包是什么。

RecordLayer 协议是和 DTLS 传输相关的协议，UDP 之上是 RecordLayer，RecordLayer 之上是 Handshake 或 ChangeCipherSpec 或 ApplicationData。RecordLayer 协议定义参考 RFC4347，实际上有三种 RecordLayer 的包:

- DTLSPlaintext，DTLS 明文的 RecordLayer。
- DTLSCompressed，压缩的数据，一般不用。
- DTLSCiphertext，加密的数据，在 ChangeCipherSpec 之后就是这种了。

没有明确的字段说明是哪种消息，不过可以根据上下文以及内容判断。比如 ChangeCipherSpec 是可以通过类型，它肯定是一个 Plaintext。除了 Finished 的其他握手，一般都是 Plaintext。

02 SRTP 密钥协商

2.1 角色协商

在 DTLS 协议，通信的双方有 Client 和 Server 之分。在 WebRTC 中 DTLS 协商的身份是在 SDP 中描述的。描述如下，参考 SDP-Anatomy 中 DTLS 参数。

a=setup:active

setup 属性在 rfc4145（https://www.rfc-editor.org/rfc/rfc4145.html）**TCP-Based Media Transport in the Session Description Protocol (SDP)**,

setup:active，作为 client，主动发起协商；

setup:passive, 作为 sever，等待发起协商；

setup:actpass, 作为 client，主动发起协商。作为 server，等待发起协商。

WebRTC的sdp交互范例（基于SFU模型）

客户端offer sdp

a=setup:actpass 作为 client，主动发起协商。

a=fingerprint:sha-256
92:1F:0B:5E:BB:04:B2:A6:6B:90:B1:7E:5B:F2:2B:C8:69:8A:FB:7D:39:59:69:AA:08:37:4A:48:07:FE:14:E0

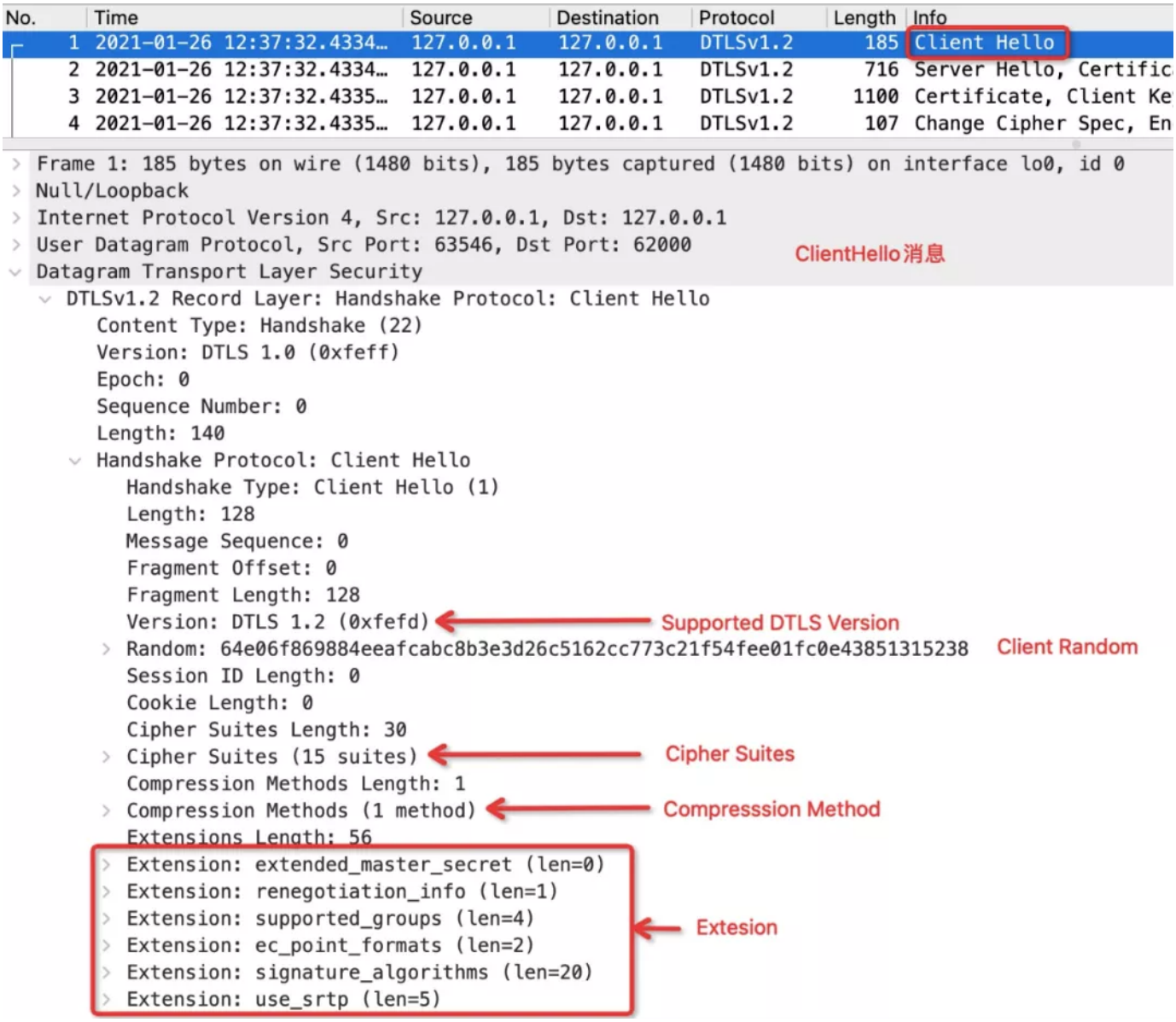
服务器 answer sdp

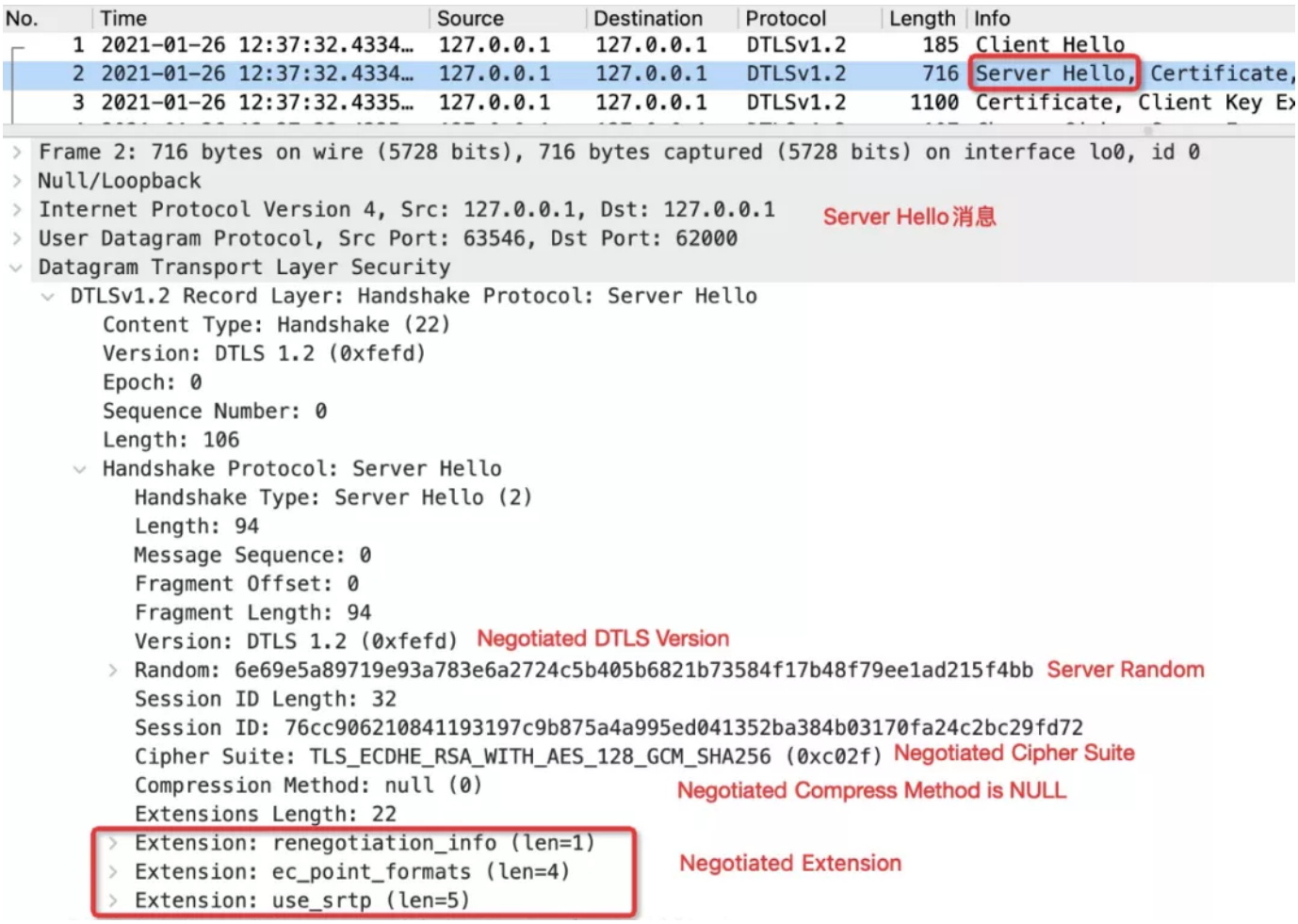
a=setup:passive作为 sever，等待发起协商

a=fingerprint 的内容是证书的摘要签名，用于验证证书的有效性，对方收到证书后，基于相同的hash算法计算一遍哈希值，如果这次计算的值和信令传过来的值相等，说明证书是完整有效的。

2.2 算法协商–Hello 消息

ClienHello 和 ServerHello 协商 DTLS 的 Version、CipherSuites、Random、以及 Extensions。


No. Time Source Destination Protocol Length Info
1 2021-01-26 12:37:32.4334… 127.0.0.1 127.0.0.1 DTLSv1.2 185 Client Hello
2 2021-01-26 12:37:32.4334… 127.0.0.1 127.0.0.1 DTLSv1.2 716 Server Hello, Certific
3 2021-01-26 12:37:32.4335… 127.0.0.1 127.0.0.1 DTLSv1.2 1100 Certificate, Client Ke
4 2021-01-26 12:37:32.4335… 127.0.0.1 127.0.0.1 DTLSv1.2 107 Change Cipher Spec, En
Frame 1: 185 bytes on wire (1480 bits), 185 bytes captured (1480 bits) on interface lo0, id 0
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 63546, Dst Port: 62000
ClientHello消息
Datagram Transport Layer Security
DTLSv1.2 Record Layer: Handshake Protocol: Client Hello
Content Type: Handshake (22)
Version: DTLS 1.0 (0xfeff)
Epoch: 0
Sequence Number: 0
Length: 140
Handshake Protocol: Client Hello
Handshake Type: Client Hello (1)
Length: 128
Message Sequence: 0
Fragment Offset: 0
Fragment Length: 128
Version: DTLS 1.2 (0xfefd)
Supported DTLS Version
Random: 64e06f869884eeafcabc8b3e3d26c5162cc773c21f54fee01fc0e43851315238
Client Random
Session ID Length: 0
Cookie Length: 0
Cipher Suites Length: 30
Cipher Suites (15 suites)
Cipher Suites
Compression Methods Length: 1
Compression Methods (1 method)
Compresssion Method
Extensions Length: 56
Extension: extended_master_secret (len=0)
Extension: renegotiation_info (len=1)
Extension: supported_groups (len=4)
Extension: ec_point_formats (len=2)
Extension: signature_algorithms (len=20)
Extension: use_srtp (len=5)
Extesion



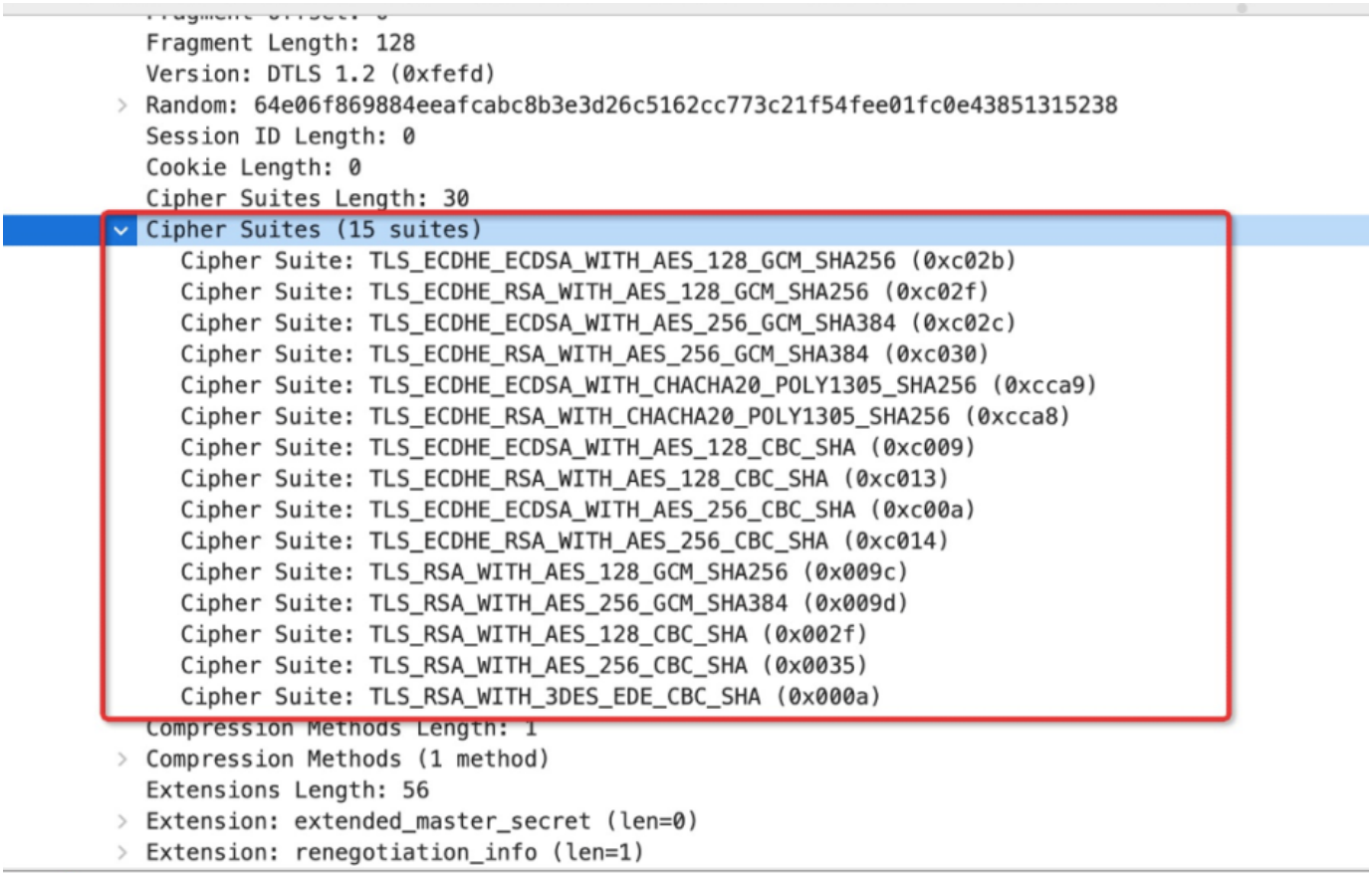
- Version：Client 给出自己能支持的、或者要使用的最高版本，比如 DTLS1.2。Server 收到这个信息后，根据自己能支持的、或者要使用的版本回应，比如 DTLS1.0。最终以协商的版本也就是 DTLS1.0 为准。

- CipherSuites：**Client 给出自己能支持的加密套件 CipherSuites**，Server 收到后选择自己能支持的回应一个，比如 TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)，加密套件确定了证书的类型、密钥生成算法、摘要算法等。

- Random：双方的随机数，参与到生成 MasterSecret。MasterSecret 会用来生成主密钥，导出共享密匙和 SRTP 密钥。详见 [导出 SRTP 密钥]。

- Extensions：Client 给出自己要使用的扩展协议，Server 可以回应自己支持的。比如 Client 虽然设置了 SessionTicket TLS 这个扩展，但是 Server 没有回应，所以最终并不会使用这个扩展。

Cipher Suite



在 Hello 消息中加密套接字使用 IANA 中的注册的名字。IANA 名字由 Protocol，Key Exchange Algorithm，Authentication Algorithm，Encryption Algorithm ，Hash Algorithm 的描述组成。

例如，TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 的含义如下：

- Protocol: Transport Layer Security (TLS)
- Key Exchange: Elliptic Curve Diffie–Hellman Ephemeral (ECDHE)
- Authentication: Rivest Shamir Adleman algorithm (RSA)

- Encryption: Advanced Encryption Standard with 128bit key in Galois/Counter mode (AES 128 GCM)

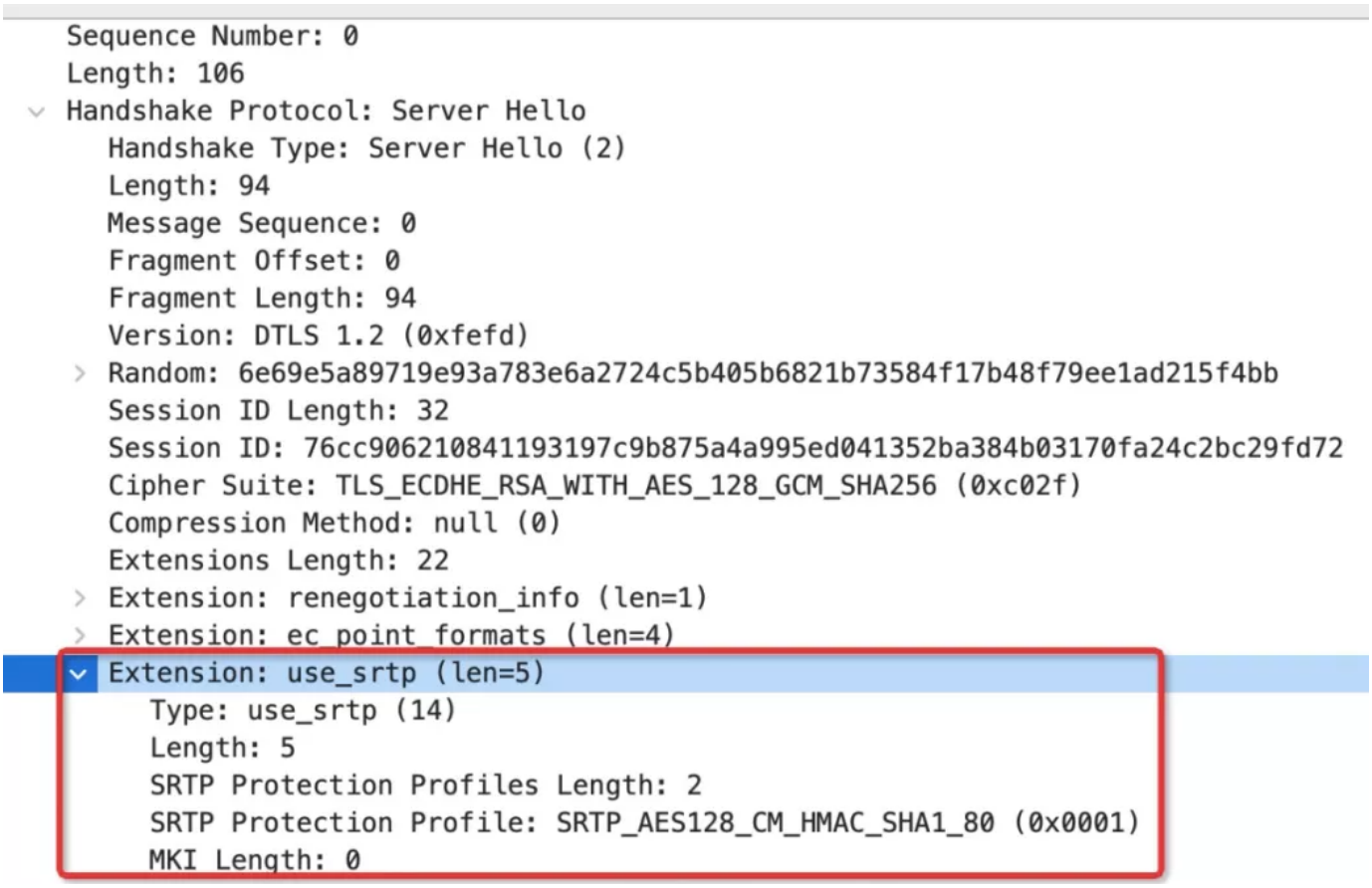
- Hash: Secure Hash Algorithm 256 (SHA256)

加密套接字在 ciphersuite.info 可以查到。在查到 IANA 名字的同时，也可以查到在 OpenSSL 中的名字。TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 在 OpenSSL 中的名字为 ECDHE-RSA-AES128-GCM-SHA256。
Note: 关于 Authentication(认证)、KeyExchange(密钥交换)、Encryption(加密)、MAC(Message Authentication Code) 消息摘要等，可以参考 RSA密钥协商。

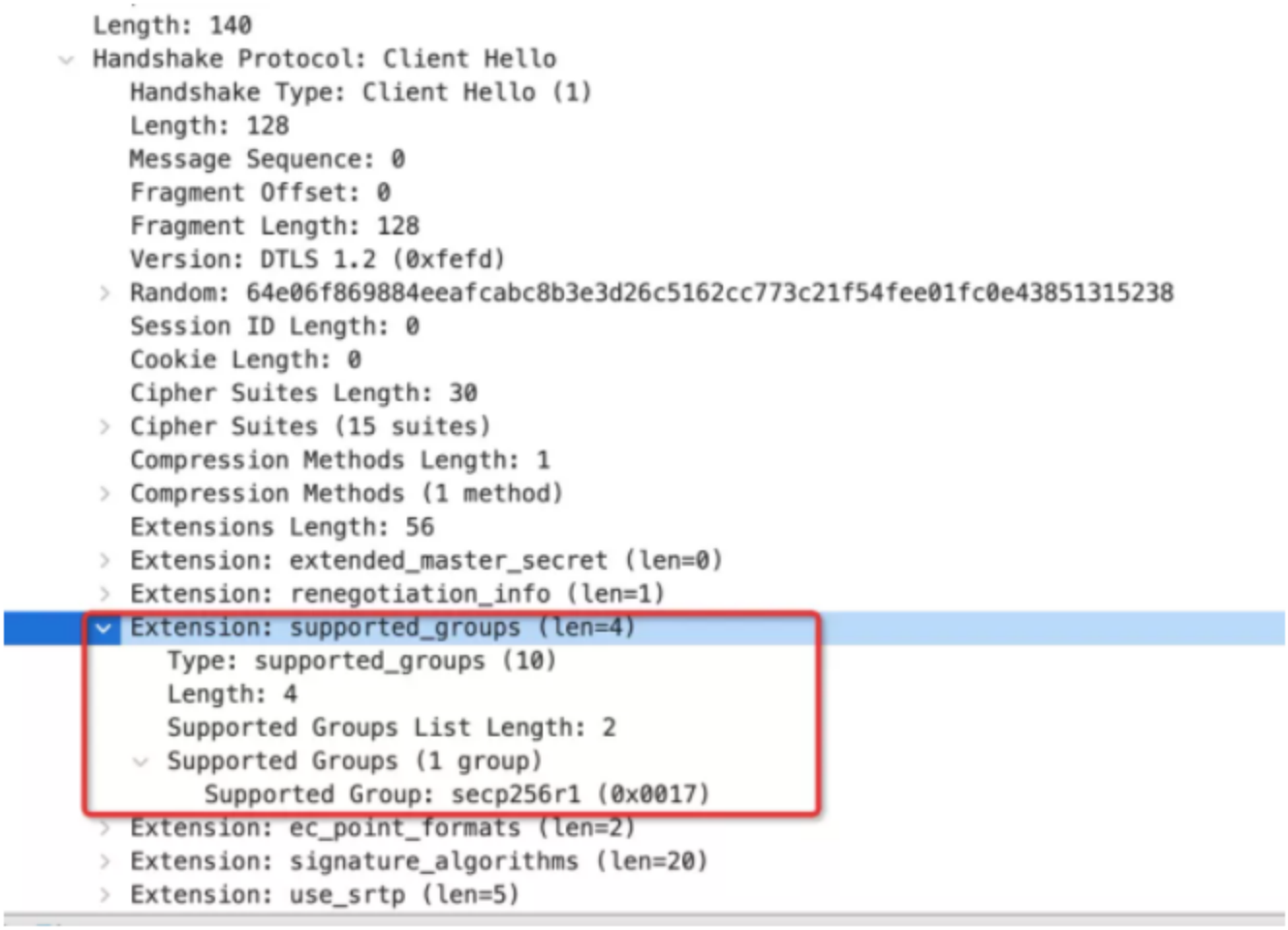
Extension

DTLS 的扩展协议，是在 ClientHello 和 ServerHello 的 Extensions 信息中指定的，所有的 TLS 扩展参考 TLS Extensions。下面列出几个 WebRTC 用到的扩展：

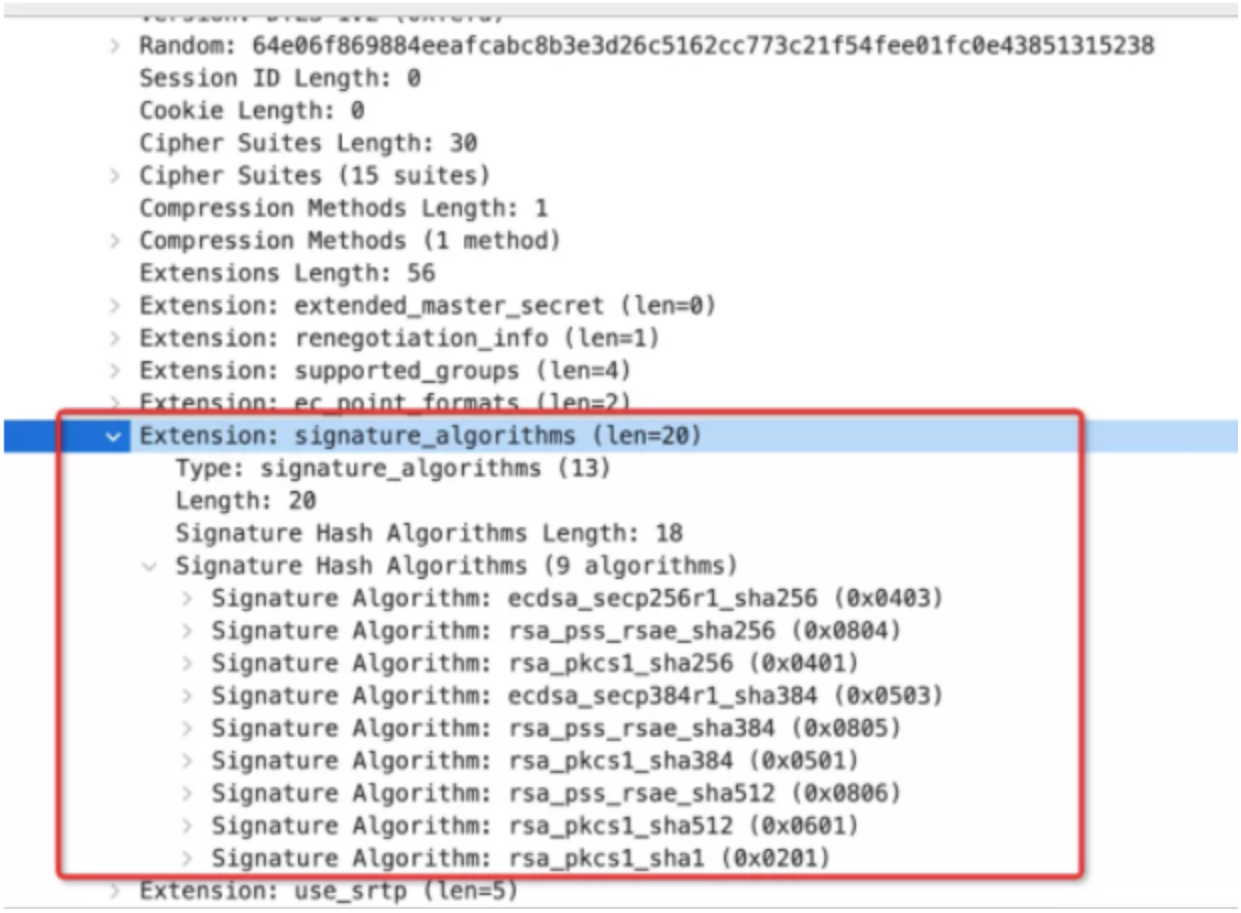
- **use_srtp**: DTLS 握手完成后 (Finished)，使用 SRTP 传输数据，DTLS 生成 SRTP 的密钥。RFC5764。ClientHello 中的扩展信息定义了 RFC5764 4.1.2. SRTP Protection Profiles 和 srtp_mki。



supported_groups，原来的名字为 elliptic_curves，描述支持的 ECC 加密算法，参考 RFC8422 5.1.1.Supported Elliptic Curves Extension，一般用的是 secp256r1。



- **signature_algorithms**，DTLS1.2 的扩展，指定使用的 Hash 和 Signature 算法，参考 RFC5246 7.4.1.4.1. Signature Algorithms。DTLS1.0，RSA 用的是 md5sha1 摘要算法，DSA 用的是 sha1 摘要算法。



- **extended_master_secret**，扩展 MasterSecret 的生成方式，参考 RFC7627。在 KeyExchange 中，会加入一些常量来生成 MasterSecret。TLS 定义了扩展方式，如果用这个扩展，DTLS 的方式和 TLS 会有些不同。

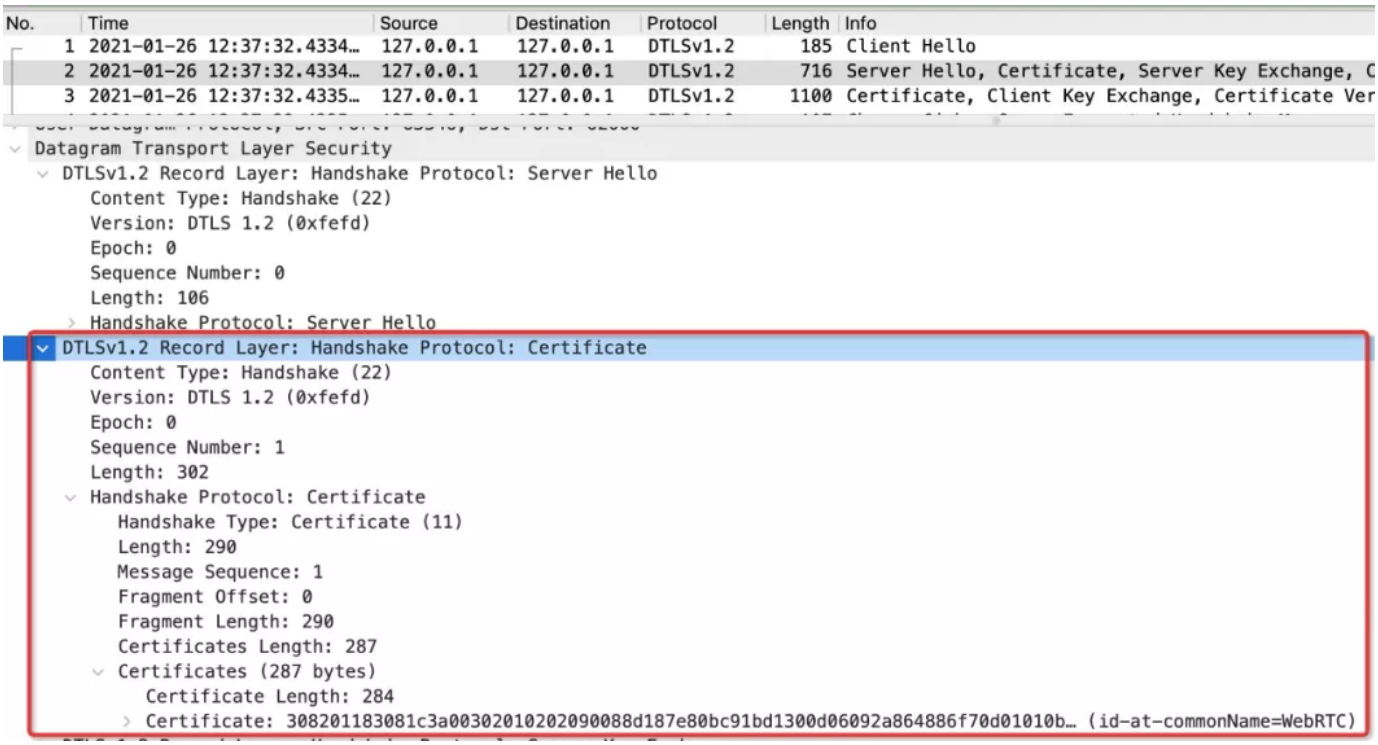
- **renegotiation_info**，参考 RFC5746。

除了这些扩展协议，和 SRTP 密钥导出相关的还有：

RFC5705: Keying Material Exporters for Transport Layer Security (TLS)，DTLS 如何从 MasterSecret 导出 Key，比如 SRTP 的 Key。

RFC5764: DTLS Extension to Establish Keys for the SRTP，DTLS 的 use_srtp 扩展的详细规范，包括 ClientHello 扩展定义、Profile 定义、Key 的计算。

2.3 身份验证-Certificate



数字证书是由一些公认可信的证书颁发机构签发的，不易伪造。数字证书可以用于接收者验证对端的身份，接收者收到某个对端的证书时，会对签名颁发机构的数字签名进行检查，一般来说，接收者事先就会预先安装很多常用的签名颁发机构的证书（含有公开密钥），利用预先的公开密钥可以对签名进行验证。

Server 端通过 Hello 消息，协商交换密钥的方法后，**将 Server 证书发送给 Client**，用于 Client 对 Server 的身份进行校验。Server 发送的证书必须适用于协商的 KeyExchange 使用的加密套接字，以及 Hello 消息扩展中描述的 Hash/Signature 算法对。

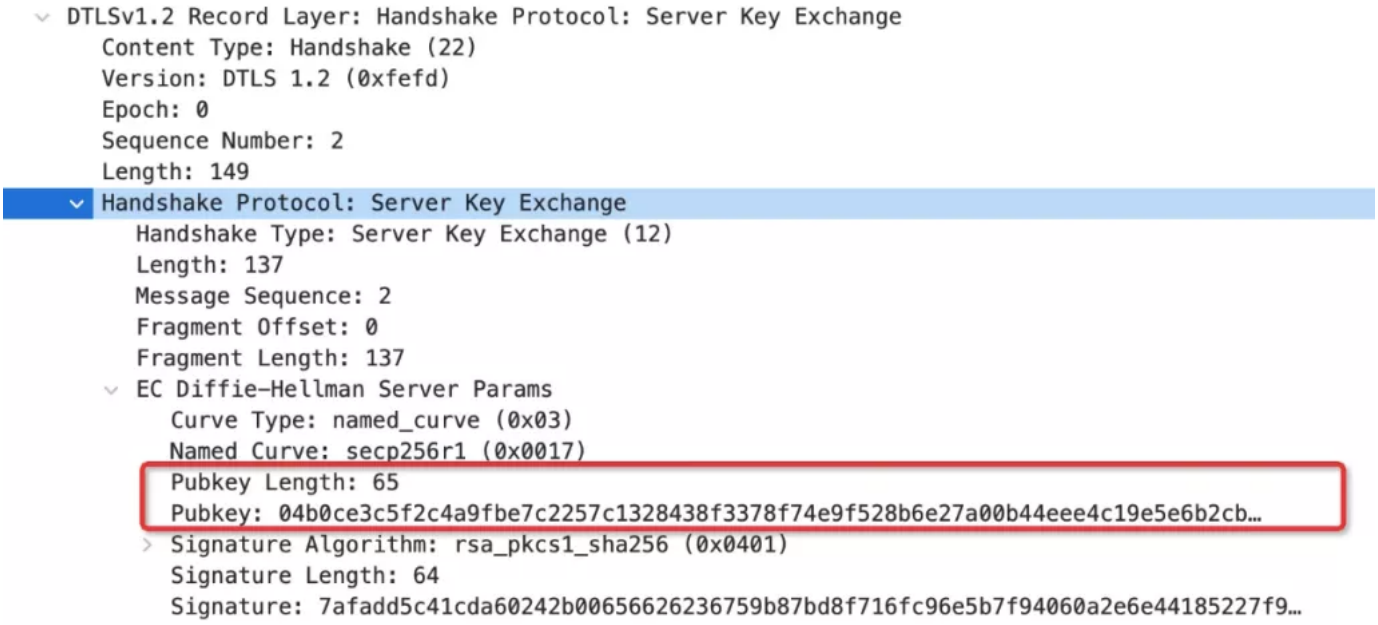
在 WebRTC 中，通信的双方通常将无法获得由知名根证书颁发机构 (CA) 签名的身份验证证书，自签名证书通常是唯一的选择。rfc4572 (https://www.rfc-editor.org/rfc/rfc4572.html)定义一种机制，通过在 SDP 中增加自签名证书的安全哈希，称为 " 证书指纹 "，在保证 SDP 安全传输的前提下，如果提供的证书的指纹与 SDP 中的指纹匹配，则可以信任自签名证书。在实际的应用场景中，SDP 在安全的信令通道 (https) 完成交换的，SDP 的安全完整是可以做到的。这样在 DTLS 协商过程中，可以使用证书的指纹，完成通信双方的身份校验。证书指纹在 SDP 中的描述如下，参考 SDP-Anatomy 中 DTLS 参数。

```
a=fingerprint:sha-
256 49:66:12:17:0D:1C:91:AE:57:4C:C6:36:DD:D5:97:D2:7D:62:C9:9A:7F:B9:A3:F4:70:03:E7:43:91:73:23:5E
```

2.4 密钥交换–KeyExchange

ServerKeyExchange 用来将 Server 端使用的公钥，发送给 Client 端。分为两种情况：

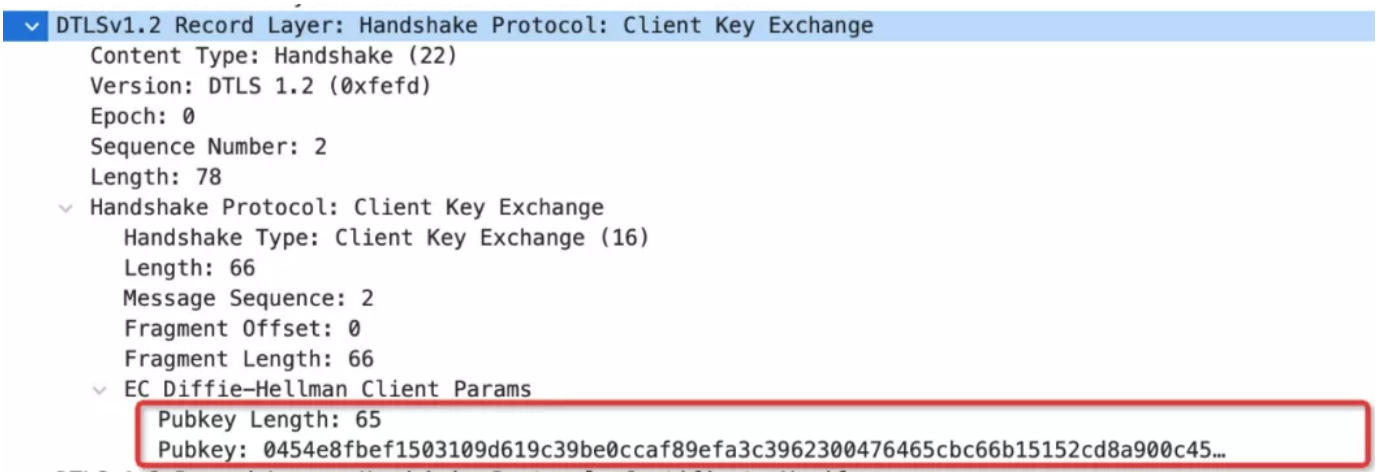
1. RSA 算法: 如果服务端使用的是 RSA 算法，可以不发送这个消息，因为 RSA 算法使用的公钥已经在 Certificate 中描述。
2. DH 算法，是根据对方的公钥和自己私钥计算共享密钥。因为 Client 和 Server 都只知道自己的私钥，和对方的公钥；而他们的私钥都不同，根据特殊的数学特性，他们能计算出同样的共享密钥。关于 DH 算法如何计算出共享密钥，参考 DH算法。



ClientKeyExchange 用来将 Client 使用的公钥，发送给 Server 端。

1. RSA 算法：如果密钥协商使用的 RSA 算法，发送使用 server 端 RSA 公钥，对 premaster secret 加密发送给 server 端。

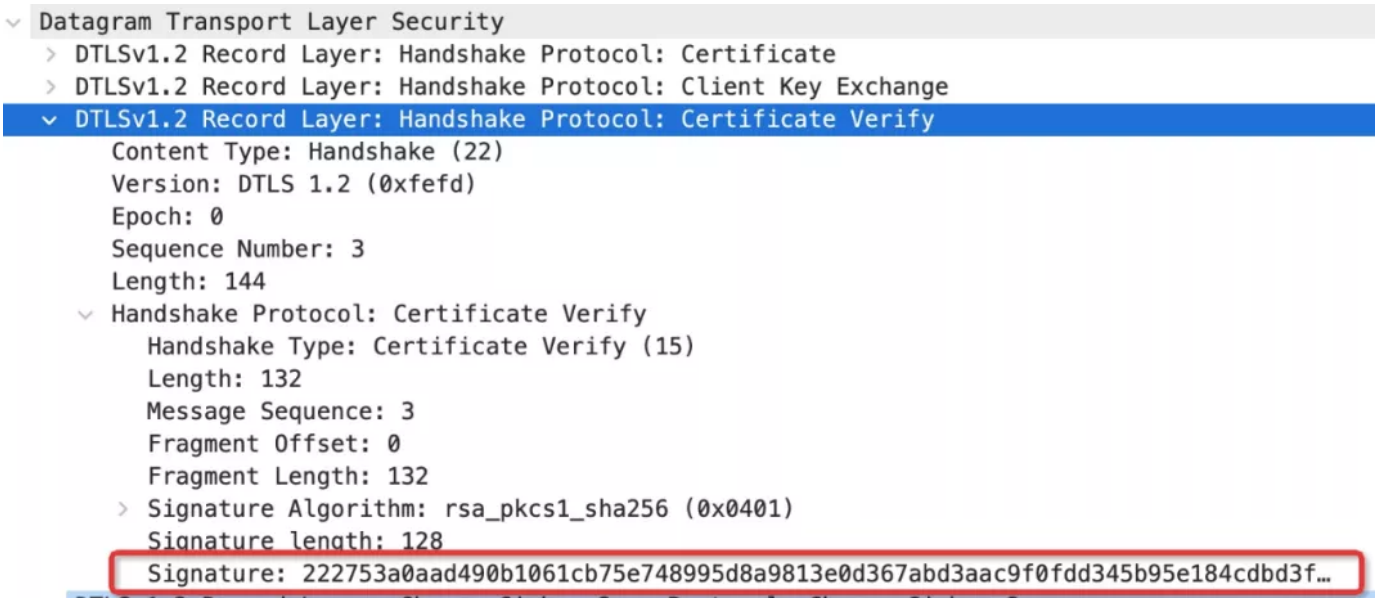
2. DH 算法：如果密钥协商使用的 DH 算法，并且在证书中没有描述，在将客户端使用的 DH 算法公钥发送给 Server 端，以便计算出共享密钥。



KeyExchange 的结果是，Client 和 Server 获取到了 RSA Key， 或通过 DH 算法计算出共享密钥。详见 [导出 SRTP 密钥] 的过程。

2.5 证书验证–CertificateVerify

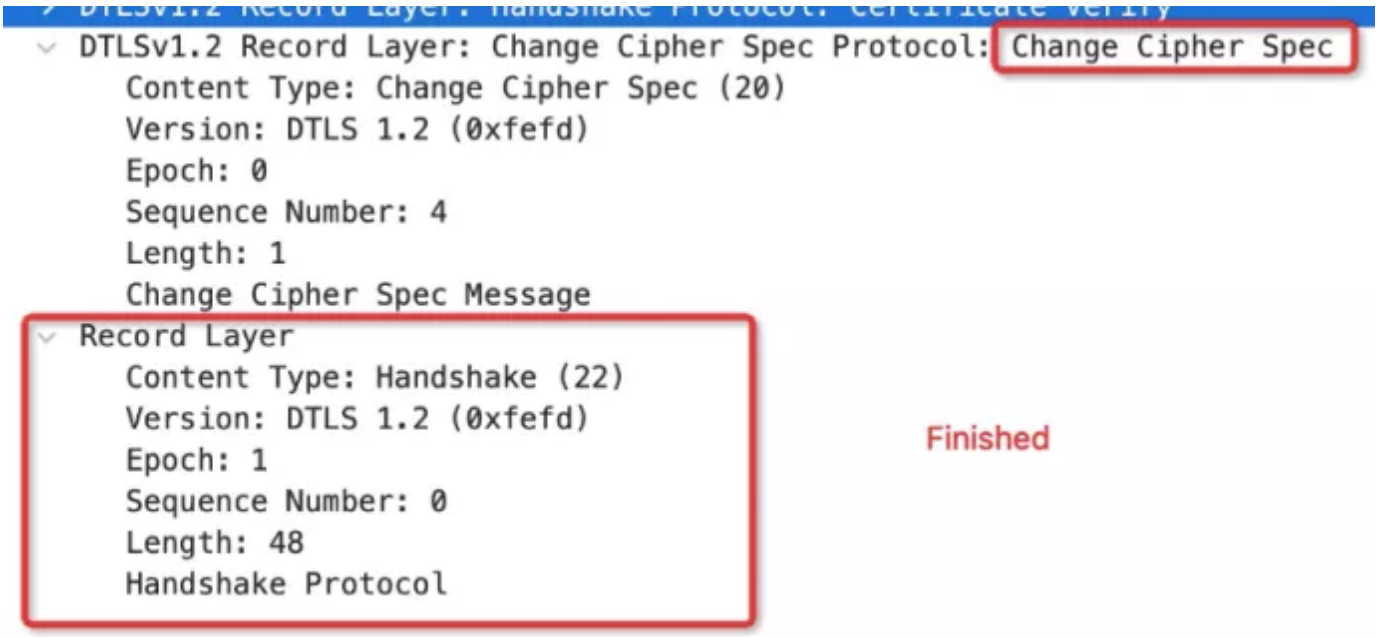
客户端使用 ClientRequest 中描述的 Hash/Signature 算法，对收到和发送的 HandShake 消息签名发送给 Server。Server 端对签名进行校验。



6. 加密验证–Finished

当 Server 和 Client 完成对称密钥的交换后，通过 ChangeCipherSpec 通知对端进入加密阶段，epoch 加 1。

随后 Client 使用交换的密钥，对 "client finished" 加密，使用 Finished 消息，发送给服务端。Server 使用交换的密钥，对 "server finished" 进行加密发送给客户端。一旦验证了 finished 消息后，就可以正常通信了。



03 导出 SRTP 密钥

上面介绍了 DTLS 的过程，以下通过结合上面例子给出的实际数据，详细说明 SRTP 密钥的导出步骤。

3.1 协商后的加密算法

加密套件：TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256(0xc02f)

椭圆曲线算法为：secp256r1，椭圆曲线的点压缩算法为：uncompressed。

椭圆曲线算法的基础知识的介绍在 ECC椭圆曲线加密算法-ECDH,ECDHE,ECDSA，由文档中我们可以知道，确定椭圆曲线加密算法有如下参数：

- 素数 p，用于确定有限域的范围。
- 椭圆曲线方程中的 a，b 参数。
- 用于生成子群的的基点 G。
- 子群的阶 n。
- 子群的辅助因子 h。

定义为六元组(p,a,b,G,n,h)

通过在 SECG-SEC2 2.4.2 Recommended Parameters secp256r1 中可以查到 secp256r1 对应的参数如下：

Basic 复制代码

```
secp256r1使用的参数如下：

使用的素数p：

p=FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
=2224(232-1)+2192+296-1

椭圆曲线E：y^2=x^3+ax+b的参数定义如下：

a=FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
b=5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B

基点G的非压缩格式：

G=046B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C2964FE342E2F
E1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5

有限域的阶n：
n=FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551

辅助因子h：
h=01
```

3.2 通过 KeyExchange 交换椭圆曲线算法公钥

ECDH Server Parameter

```
Basic | 复制代码
1  Pubkey:04b0ce3c5f2c4a9fbe7c2257c1328438f3378f74e9f528b6e27a00b44eee4c19e5e6b2
   cb6cab09f796bcf8c05102b2a4bcdc753d91cc4f431f558c845a1ba6f1ce
```

记为 Spk

ECDH Client Paramter

```
Basic | 复制代码
1  PubKey:
   0454e8fbef1503109d619c39be0ccaf89efa3c3962300476465cbc66b15152cd8a900c45d5064
   20f0123e65d8fbb70cb60b497893f81c5c2a0ef2f4bc2da996d9e
```

记为 Cpk

3.3 根据 ECDHE 算法计算共享密钥 S(pre-master-secret)

假设 Server 的使用的椭圆曲线私钥为 Ds, Client 使用的 Dc，计算共享密钥的如下，参考 ECC 椭圆曲线加密算法-ECDH,ECDHE,ECDSA

S = Ds * Cpk = Dc * Spk

这个共享密钥 S 就是我们在 RFC 文档中看到的 pre-master-secret

3.4 计算master secret

计算 master secret 过程如下，可参考 Computing the Master Secret

```
master_secret = PRF(pre_master_secret, "master secret", ClientHello.random + ServerHello.ran
dom)[0..47];
```

计算出来的 master_secret 为 48 Bytes，其中 ClientHello.random 和 ServerHello.random 在 Hello 消息中给出。PRF 是伪随机数函数（pseudorandom function），在协商的加密套件中给出。

3.5 使用 master_secrete 导出 SRTP 加密参数字节序列

使用 RFC5705 4. Exporter Definition 给出的计算方式，使用参数 master_secret，client_random，server_random 计算字节序列：

```
key_block = PRF(master_secret, "EXTRACTOR-dtls_srtp", client_random + server_random)[length]
```

在 DTLS-SRTP 4.2. Key Derivation 中描述了需要的字节序列长度。

```
2 * (SRTPSecurityParams.master_key_len + SRTPSecurityParams.master_salt_len) bytes of data
```

master_key_len 和 master_salt_len的值，在 user_srtp 描述的profile中定义。我们的实例中使用的 profile 为 SRTP_AES128_CM_HMAC_SHA1_80，对应的 profile 配置为：

```
SRTP_AES128_CM_HMAC_SHA1_80
        cipher: AES_128_CM
        cipher_key_length: 128
        cipher_salt_length: 112
        maximum_lifetime: 2^31
        auth_function: HMAC-SHA1
        auth_key_length: 160
        auth_tag_length: 80
```

也就是我们需要 (128/8+112/8)*2 = 60 bytes 字节序列。

3.6 导出SRTP密钥

计算出 SRTP 加密参数字节序列，在 DTLS-SRTP 4.2. Key Derivation 描述了字节序列的含义：

```
client_write_SRTP_master_key[SRTPSecurityParams.master_key_len]; // 128 bits
server_write_SRTP_master_key[SRTPSecurityParams.master_key_len]; // 128 bits
client_write_SRTP_master_salt[SRTPSecurityParams.master_salt_len]; // 112 bits
server_write_SRTP_master_salt[SRTPSecurityParams.master_salt_len]; // 112 bits
```

至此我们得到了，Client和Server使用的SRTP加密参数：master_key 和 master_salt。

04 DTLS 超时重传

DTLS 是基于 UDP 的，不可避免会出现丢包，需要重传。如果处理不当，会导致整个通信双方无法建立会话，通话失败。RFC6347 4.2.4 给出了超时和重传机制。

在处理重传时，以下几点需要注意：

1. 在 DTLS 协议中，为了解决丢包和重传问题，新增了 message_seq。 在发送 DTLS 重传消息时，一定要更新其中的 message_seq，这样对端将把包识别是一个重传包，响应正确消息。否则，会默默丢弃这些包，不进行响应。

2. 当 server 端收到 client 的 FINISHED 消息，并发送 FINISHED 消息给 client，更新 server 状态为协商完成，开始发送 SRTP 数据。此时发送给 client 的 FINISHED 消息，出现丢包。client 收到 SRTP 数据后丢弃。同时，再次发送 FINISHED 消息到 server，server 要正确响应。否则，会导致 DTLS 协商完成的假象，通话失败。

3. 使用 openssl 1.1.1 之前版本，无法设置 DTLS 超时重传时间，可以超时重传机制不可用，大家开始转向使用 boringssl。openssl 1.1.1 开始版本已经支持设置 DTLS 超时重传，达到和 boringssl 同样的效果。参考 DTLS_set_timer_cb。

05 OpenSSL 的 DTLS 功能

DTLS 是一个庞大的协议体系，其中包括了各种加密，签名，证书，压缩等多种算法。大多数项目是基于 OpenSSL 或 BoringSSL 实现的 DTLS 功能。在实际项目使用 OpenSSL 的 DTLS 功能，与协商有关的接口总结如下。

1. X509_digest，**计算证书 fingerprint，用在 SDP 协商中的 fingerprint 属性**。

2. SSL_CTX_set_cipher_list，设置使用的加密套件，通过设置算法的描述，影响 Hello 消息中的 cipher list。

3. SSL_CTX_set1_sigalgs_list 设置签名算法。通过设置签名算法的描述，影响 hello 消息中 signature_algorithms 扩展。signature_algorithms 对 DTLS 的 Hello消息，KeyExchange，CerficateVerify消息。signature_algorithms 设置不正确，会出现 internal error，不容易定位问题。

4. SSL_CTX_set_verify 设置是否校验对端的证书。由于在 RTC 中大多数据情况下使用自签证书，所以对证书的校验，已校验身份是需要的。

5. SSL_CTX_set_tlsext_use_srtp 设置 srtp 扩展。srtp 扩展中的 profile，影响 srtp 加密时使用密钥的协商和提取。

6. SSL_set_options 使用 SSL_OP_NO_QUERY_MTU 和 [SSL_set_mtu] 设置 fragment 的大小。默认 OpenSSL 使用 fragment 较小。通过上面两个接口，设置适合网络情况的 fragment。

7. DTLS_set_timer_cb，设置超时重传的 Callback，由 callback 设置更合理的超时重传时间。

在开源项目 SRS（https://github.com/ossrs/srs） 中已经支持了 WebRTC 的基础协议，对 DTLS 协议感兴趣的同学，可以基于 SRS 快速搭建本机环境，通过调试，进一步加深对 DTLS 的理解。

06 总结

本文通过 WebRTC 中 SRTP 密钥的协商过程，来说明 DTLS 在 WebRTC 中的应用。DTLS 协议设计的各个加密算法的知识较多，加上 TLS 消息的在各种应用场景中的扩展，难免有理解和认知不到的地方，还需要进一步深入探索。

参考文献

1. TLS 1.2
2. DTLS 1.2
3. TLS Session Hash Extension
4. TCP-Based Media Transport in the Session Description Protocol
5. TLS Extension

6. SRTP Extension for DTLS
7. OpenSSL Man
8. ECC椭圆曲线加密算法-介绍
9. ECC椭圆曲线加密算法-有限域和离散对数
10. ECC椭圆曲线加密算法-ECDH、ECDHE和ECDSA