MESSENGER EXPOSED

Investigative Report

&yet

Research led by Philipp “Fippo” Hancke

&yet Chief WebRTC Engineer

WebRTCHacks Contributor

ABOUT THIS REPORT

WebRTC is a free, open project that provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via simple APIs. The WebRTC.org library allows developers to obtain a high-quality, state of the art networking and media library that is also used in the Google Chrome browser free of charge. As such, it has been used by a number of services to implement Voice and Video calls.

Is Facebook Messenger one of those services? Read on...

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Once again, this writeup would not exist without the help and feedback from various people: Serge Lachapelle, Peter Saint-Andre, Justin Uberti, Amy Lynn Taylor, Lance Stout.
Special thanks to Mozilla's Maire Reavy for reacting swiftly to the news that the next update would kill Messenger's ability to interoperate with Firefox.

While the data capture is trivial, the interpretation is a black art.
All remaining errors are my own.

SUMMARY

Facebook Messenger is an extremely popular messaging and communications app. In making this report we have learned that:

- Facebook Messenger uses WebRTC on Web, iOS and Android.
- Facebook Messenger has cut all ties to Skype.
- Facebook Messenger is very different to Whatsapp when it comes to voice and video calling.

Messenger works in Chrome / Firefox and Opera as well as Android and iOS. The mobile versions use the WebRTC.org library and Messenger is optimized heavily for mobile usecases:

- Instead of DTLS, the older SDES encryption scheme is used between mobile clients. While not offering features such as perfect forward secrecy, it provides faster call setup time.
- The audio codec depends on platform and peer. Opus, ISAC and ISAC Low Complexity are used, with a packetization that prefers fewer, larger packets.
- VP8 is used as video codec on web, iOS and Android. For iOS, Facebook chose VP8 over H264, even though hardware acceleration might be available.

WebRTC fulfilling its promise: No plugins required

Back in 2011, Facebook launched Skype-powered video calling. It used a plugin. The need for a plugin is now gone, probably along with other parts of the integration as described in this announcement. Most platforms are supported without requiring users to install something. WebRTC is starting to fulfill its promise: no plugins needed.

GENERAL NOTES

The WebRTC rollout at Facebook has been done gradually, starting in early 2015. Chad Hart was one of the first people to notice in mid-january. I took a quick look and was not very impressed by what I found. Basically it was a simple 1-1 webchat akin to Google's apprtc sample.

Recently, there have been many Facebook announcements. They launched messenger.com as a standalone website with support for voice and video between browsers. As Tsahi Levent-Levi pointed out already, it is using WebRTC.

On the mobile side, the Messenger client had been voice only. Video calling is available in the apps as well. All of this warrants a closer look. With WhatsApp and Messenger coming from the same company, we were expecting similarities, making this report easy. Well... no.

It turns out that unlike WhatsApp, Messenger uses the webrtc.org library provided by Google. For video calls, the VP8 video codec is used. The choice of the audio codec depends on the devices used and includes ISAC and Opus. DTLS is used for encrypting

the media between the app and the browsers. Interestingly, it is not used in calls between two mobile devices.

Mobile

While Messenger looks pretty standard, there are quite a number of optimizations here. Some gems are hidden to anyone glancing through the details. Chrome's webrtc-internals are easily available and was used in the first scenarios tested. It allows looking at the implementation from a number of different angles, all the way from the captured packets, via the signaling protocol and up to the WebRTC API calls.

The iOS application offers a number of nonstandard codecs. In particular, when calling Chrome the ISAC audio codec will be used. With Firefox, which does not implement ISAC, Opus is used. In both cases, the app tweaks the codec usage by sending larger frames than the browser. That is quite an interesting hack reminiscent of the WhatsApp behaviour of switching to a larger packetization. I can only speculate that this may show larger packets to be more robust on wireless networks?

Calls between two mobile devices turn out to be more interesting. Here, the optimizations for calling a browser come to bear fully.

Currently, video calls are only supported between mobile devices, which will surely change soon. VP8 is used as the codec.

Unlike WhatsApp, Messenger can be run on multiple devices. So when calling someone who is logged in on multiple devices, all devices ring. This is nice behaviour, as it allows the called user to choose where to take the call. This makes a lot more sense than previous approaches, e.g. Google Talk attempting to let the caller determine the recipients "most available device" from a UX point of view. In a world where mobile devices are disconnected frequently and need to be woken up via push messages, this is even more important than it was a decade ago..

Speed vs Security

The most important takeaway is that instead of using DTLS, Messenger uses the older SDES encryption scheme between two mobile clients. This means media can flow as soon as ICE is done, eliminating a round-trip for the DTLS handshake. This shows a desire to reduce the session startup time.

The widely known downside of that is that the encryption keys are sent via the signaling servers and can be used to retroactively decrypt traffic. Government agencies will surely rejoice at the news of this being applicable to 10% of the mobile VoIP traffic...

THIRD-PARTY LICENSES

Facebook Messenger on iOS and Android actually shows the third party licenses in the settings menu. The list displayed is rather long, but contains a few notable entries:

- libjingle 2004-2005
- libsrtp 2001-2006
- libvpx 2010
- libyuv 2011
- opus, 2001-2011
- webrtc, 2011 copyright, including long list of individual files

This is how third-party software that is BSD-licensed (or a similar license that requires attribution) should be displayed. And it confirms that the webrtc.org library is used to implement the call functionality.

CAPTURE SETUP

The capture setup should already be familiar from previous reports but is repeated here for completeness. To capture the traffic, a Wifi Pineapple Wifi router (Mark IV, the Mark V works as well) is used.
The Android and iOS devices running the Messenger app are connected via WLAN to the Pineapple which itself is connected via Ethernet to a NAT router that connects to the internet.

Even though the Pineapple has various useful capabilities for intercepting, a tcpdump installation is all that is required. See here for installation instructions.

In this setup, the packets from the WLAN interface (wlan0) need to be forwarded to the LAN interface (eth1). This is accomplished with the wp4.sh script with the following settings:

```
wp4.sh with eth1 and wlan0 instead of defaults:
Pineapple Netmask [255.255.255.0]:
Pineapple Network [172.16.42.0/24]:
Interface between PC and Pineapple [eth0]: wlan0
Interface between PC and Internet [wlan0]: eth1
Internet Gateway [192.168.1.1]:
```

The Pineapple uses a 172.16.42.0/24 network and the Ethernet port connected to the local network at &yet's offices. The IP 192.168.1.157 is assigned to the Pineapple by DHCP.

Two mobile devices are used, an Android device with IP 172.16.42.124 and an iPad with IP 172.16.42.187. Additionally, a laptop with IP 192.168.1.202 is used running the Chrome (41) and Firefox (37) browsers using facebook.com.

Capturing is done by executing

```
tcpdump -v -i wlan0 -A -w some.pcap
```

This should usually be done in the /usb directory with an USB stick attached for storage. The captured PCAP file can then be analyzed in Wireshark or similar tools.

SESSION 1

Dump: messenger.pcap
The first capture session shows the iOS application calling the Chrome browser. Both clients are in the same network, the iOS client has the IP 172.166.42.187, the Chrome client is 192.168.1.202.
We have both the webrtc-internals dump as well as a saved version of the chrome://webrtc-internals page available.

The pcap dump looks relatively uninteresting. Since the caller is a standard browser, there is no custom stuff to be expected. An automated analysis tool showed the google-ice variant used, however. In packet #300 (filter: udp.port eq 64987) this stream can be observed, it is sent by the iOS client. This packet is a STUN allocate request. The username fragment is just 16 characters compared to the normal webrtc.org behaviour of concatenating both 16 character username fragments separated by a colon. It turns out that this is using just the local ICE username of the client.
Also, this behaviour differs from the normal behaviour of the webrtc.org library which typically tries to allocate without authentication first.

Additionally, this request contains an extra STUN attribute 0x0F with a length of four bytes. Wireshark can't interpret this attribute in this packet. It is happy to interpret it as a MAGIC COOKIE in packet #372 however. This magic cookie is from a pre-standard version of TURN which is still implemented in the webrtc.org library, very easy to find by looking for TURN_MAGIC_COOKIE_VALUE.

Other than that, we see pretty much standard WebRTC behaviour here. Using "dtls" as a filter makes it easy to identify the peer-to-peer connection between both clients that can be filtered by

(ip.addr eq 172.16.42.187 and ip.addr eq 192.168.1.202) and

(udp.port eq 54106 and udp.port eq 50927)

The first part of the DTLS handshake is happening via the relay server even and switches to being p2p in the middle of the handshake.

The rest of the conversation shows RTP packets with payload types of 103 and 105 respectively. With access to the SDP it is trivial to identify those payloads as ISAC and comfort noise (CN) respectively. The codec bitrate varies strongly, on average it is about 25kbit/s for both directions with peaks of 50kbit/s. This variation in bandwidth is quite characteristic for ISAC when used with voice activity detection. You can see this yourself in this demo from the WebRTC samples repository.

The iOS app is sending an RTP stream with SSRC 0x242cc502 at a rate of about 18 packets per second and a timestamp difference of 960 which implies 60ms frames as described in this draft.

The RTP stream sent by Chrome with SSRC 0x89347586 is sending at a rate of 33 packets per second and a timestamp difference of 480 which implies 30ms frames.

Since Messenger is compatible with the Facebook web application, we can use the chrome://webrtc-internals page to look at what is happening.
The incoming call contains the following offer in the first setRemoteDescription call:

```
o=- 2154911593180001634 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE audio
a=msid-semantic: WMS stream_label
m=audio 1 RTP/SAVPF 124 103 96 120 122 121 119 117 116 125 0 8 105 127
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:djzkVJ869EKVLi/h
a=ice-pwd:U1yLJRKLxSHqEjYTBMkupQaL
a=ice-options:google-ice fb-force-5245
a=fingerprint:sha-256 2E:55:72:F5:3C:1D:32:51:8B:65:A4:8A:73:C5:1B:81:6
7:F5:29:1E:EF:24:76:F4:7D:6C:78:41:22:64:C4:1F
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=sendrecv
a=mid:audio
a=rtcp-mux
a=crypto:0 AES_CM_128_HMAC_SHA1_32 inline:AhYyjqKjqbKrkHqKpMZK2+JHPfx-
4J53vylS8xeUn
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:DYBtVGMBQ20qMnX3SYke04Ir-
Jqt4/Ah2PhpE1BYN
a=rtpmap:124 SPEEX_ISAC/16000
a=rtpmap:103 ISAC/16000
a=rtpmap:96 speex/8000
a=fmtp:96 speexusejitterbit=
a=rtpmap:120 opus/48000/2
a=fmtp:120 sprop-stereo=0; stereo=0; maxaveragebitrate=12000; minp-
time=10
a=rtpmap:122 ISPXE1/16000
a=rtpmap:121 ISPXE1_WB/16000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:105 CN/16000
a=rtpmap:127 red/8000
a=fmtp:127 useadaptivefec=
a=maxptime:60
a=ssrc:606913794 cname:pmKQ9jBJmHmiw+Iy
a=ssrc:606913794 msid:stream_label audio_label
a=ssrc:606913794 mslabel:stream_label
a=ssrc:606913794 label:audio_label
```

There are some interesting things in this offer. For the rest, see this great article on webrtcHacks for a description the "standard" WebRTC SDP.

- In addition to the old google-ice ice-option, there is a fb-force-5245 flag. RFC 5245 is the IETF specification for the ICE protocol. It is not quite clear why this would have to be forced, since the default behaviour of the WebRTC library is to ignore the old google-ice protocol (which was based on an early draft of RFC 5245) if both sides support the standard variant.
- The offer contains both a DTLS fingerprint in the a=fingerprint and two a=crypto lines for SDES. This was the default behaviour of the webrtc.org library in older versions. As we will see later, this is actually intentional.
- The preferred codec is SPEEX_ISAC/16000. Both Speex and ISAC are audio codecs which makes this codec name confusing... one can't mix audio codecs.
- Next is ISAC, the internet speech audio codec developed by GIPS. This codec is still supported by Chrome but not by Firefox.
- The third codec is Speex at a sampling rate of 8khz. In addition to the a=rtpmap line there is an a=fmtp line which sets a usejitterbit option (with a spurious equal sign after the option which might indicate an improperly done mapping of non-key-value parameters here).
- The next codec is Opus, with a set of options in the a=ftmp line. This is an attempt to tune Opus to the usage on a mobile device by restricting the complexity. See here for the specification which contains full explanations:
  - the sprop-stereo=0 means the offerer is unlike to produce stereo
  - stereo=0 means the offerer prefers to receive mono signals.
  - maxaveragebitrate=12000 sets the bitrate the mobile client wants to receive to 12kbit/second which is lower than Opus default bitrate of roughly 40kbit/s.
  - minptime=10 sets the minimium packetization time for opus to 10ms.
- ISPXE1 and ISPXE1_WB (for wideband) are probably two proprietary codecs.
- red/8000 shows RFC 2198 redundant audio data at a clock rate of 8000 and an associated a=ftmp line which enables something called "adaptivefec".
- the a=ssrc lines show a mediastream id of "stream_label audio_label", i.e. a stream with a label "stream_label" and a track with label "audio_label". This is probably taken directly from Google's peerconnection_client example.

Looking further down, the addIceCandidate calls turn out to be interesting, too. They're added in the following order:

- relay
- srflx
- host

Whether that is an attempt to go for the relay first is unclear. Another interesting observation is that there are no RTCP candidates. When assuming that both sides understand rtcp-muxing (RFC 5761) this reduces the number of pairs ICE has to check. Controlling this behaviour at the sender is currently under consideration for the JSEP specification as "rtcpMuxPolicy".

Last but not least, we can even capture the signaling traffic in the browser. Unlike when decoding Firefox Hello we don't even need a man-in-the-middle proxy for this, because the signaling traffic is readily available in the network tab of the browser.

The log contains too much information bound to the account and is therefore not made available. The signaling connection is easy to identify by searching for a request to

https://2-edge-chat.facebook.com/pull?channel=

The request consists of JavaScript objects which contain a payload like the following one:

```
"payload":"{\"type\":\"ice_candidate\",\"msg_id\":3022461217,\"-
call_id\":756411875,\"capability\":10,\"version\":1,\"sdp\":\"a=-
candidate:2481683196 1 udp 2113937151 10.199.34.94 58173 typ host
generation 0 username EH8pPyN9wK7vw3hp password +tlt1xdBYv0U-
WUo9LvW8Y8g3\\r\\n\",\"sdp_mline_idx\":0,\"label\":\"0\",\"sdp_
mid\":\"audio\"}"
```

as well as attributes like

```
"call_id":"756411875","msg_type":"ice_candidate","source":"mobile","-
realtime_viewer_fbid":"id of the caller?","type":"webrtc"}]}
```

So in addition to the candidate information provided by the WebRTC API, there is some metadata like the call id and the source present. Of particular importance here is probably the "capability". At the time of writing this section, the mobile client only allowed audio calls which is signaled here probably.

The actual offer contains, in addition to the SDP, the following extra attributes in the payload:

```
\"pranswer\":false,\"icerestart\":false,\"handlescollision\":true,\"f-
bexp\":\"ping_timeout=20000|w_tmot=8000|weak_wrtbl=1|i2s_trig-
ger=5000|spx_rate=8000|spx_rtt=2000\",\"videoon\":false}
```

The extra flags are

- pranswer: the protocol apparently supports provisional answers as defined in the JSEP specification.
- icerestart: this hints that ICE restarts (which are necessary when the network conditions change) might be possible.
- handlecollision might indicate support for glare handling, i.e. a way to deal with concurrent offers when upgrading from an audio call to a video chat.
- fbex seems to be a list of key-value pairs separated by | -- after all, why use JSON objects when you can write your own key-value parser?
- videoon: possibly this signals whether video is not only available bot also used. While this information is part of the SDP, it might be repeated here for convenience reasons.

Also, the signaling traffic show a pretty consistent behaviour with sending candidates before the actual offer. Those candidates don't show up in WebRTC-internals. This is a classic bug (in-order processing is hard in message delivery).

SESSION 2

Dump: messenger-firefox.pcap

This session captures an audio call between the iOS application and Firefox. Of particular interest is the audio stream. From the SDP one would expect a low-bitrate Opus stream.
The P2P connection can be filtered with

> (ip.addr eq 192.168.1.202 and ip.addr eq 172.16.42.187) and (udp.port eq 41260 and udp.port eq 54005)

and decoding those packets as RTP shows a payload type 120. Which is Opus in the previously captured SDP. We have two SSRCs and streams

- 0x87146579 sent from the iOS app with about 17kbit/s
- 0x30d6710a sent from Firefox starting at 20kbit/s and dropping down to 12kbit/s after a few seconds.

The RTP packets from the iOS application are sent at a rate of 18 per second with a timestamp difference of 2660 and a raw codec bitrate of about 12kbit/s -- which matches the expectation from the offer SDP.
The packets sent by Firefox are sent more frequently, 50 packets per second with a timestamp difference of 960. The codec bitrate starts at about 20kbit/s and then drops to the desired 12kbit/s as well.

> **Comment**
>
> Again, we're seeing some tuning here, this time of Opus. A low bitrate stream with a large packetization is chosen over the default. We have seen this preference for fewer, larger packets in the previous session with ISAC.

Another issue that was noticed during testing was that calls from the app to Firefox 38 did not work. At &yet we had seen similar behaviour with our old Talky iOS app. Firefox 38 requires perfect forward secrecy ciphers and this makes the DTLS handshake fail. Fortunately, this could be worked around before Firefox 38 rolled out and has also been fixed by the Messenger team.

SESSION 3

Dump: messenger-ios-android.pcap

Since Messenger is supported both on iOS and Android, let's test a call between those two platforms on the LAN. We cannot see the SDP like we could in the browser, but we roughly know what the offer should look like.
The iOS client is running on IP 172.16.42.187 and the Android client on 172.16.42.124.

Looking at the dump, we can see the clients discovering each other quickly using

> (ip.addr eq 172.16.42.187 and ip.addr eq 172.16.42.124) and (udp.port eq 63339 and udp.port eq 60198)

as a filter. Packet #153 shows the first binding request and packet #186 the response. Both clients continue exchanging binding requests for quite a while (three seconds) before the first non-STUN packet is sent.

Surprisingly, it is not a DTLS client hello.

Comment

The lack of DTLS between mobile clients is somewhat surprising. Given Facebook's signaling protocol this allows them to retrospectively decrypt all sessions, one of the big arguments made against SDES at the IETF meeting in Berlin a few years ago. However, SDES does not require an additional handshake at the P2P level which can take quite long if there is packet loss. Maybe something new is needed that combines the PFS properties of DTLS-SRTP with the fast setup of SDES?

Instead, packet #241 shows an RTP packet with a payload type 103. This is, as we know from earlier sessions where we had the SDP available, the ISAC codec. The timestamp difference is 960 with about 11 packets sent per second, so 60ms frames again. The bitrate varies again, about 18kbit/s for both directions on average.

SESSION 4

Dump: messenger-android-ios.pcap

For the sake of completeness, let's reverse the roles and let the Android device call the iOS device. Not many surprises to be expected, right?

Well, as before the P2P stream is quickly identified with a filter of

> (ip.addr eq 172.16.42.187 and ip.addr eq 172.16.42.124) and (udp.port eq 53297 and udp.port eq 43984)

Again, no DTLS is used. The RTP payload type is 119, the timestamp difference is 960 and there are on average 11 packets per second (up to 17). Let's scroll back a few pages and look at the SDP...

```
m=audio 1 RTP/SAVPF 124 103 96 120 122 121 119 117 116 125 0 8 105 127
```

Codec 119 is there but has lower priority than ISAC (103). So why is codec 119 chosen here?
Let's start a different session from Android to the browser and look at the offer SDP:

```
type: offer, sdp: v=0
o=- 6513330831953853535 2 IN IP4 127.0.0.1
s=-
t=0 0
```

```
a=group:BUNDLE audio
a=msid-semantic: WMS stream_label
m=audio 1 RTP/SAVPF 119 124 103 96 120 122 121 117 116 125 0 8 105 127
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:Uhznx0RRftqvzyiJ
a=ice-pwd:doJXZm0LqW00TW2gKqH6VB6L
a=ice-options:google-ice fb-force-5245
a=fingerprint:sha-256 6B:F4:A1:D1:E1:FA:D6:D5:AF:7C:B4:B2:2D:72:55:1D:9
3:2B:6E:47:82:DA:9B:9B:DD:F6:38:00:D9:EB:56:04
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=sendrecv
a=mid:audio
a=rtcp-mux
a=crypto:0 AES_CM_128_HMAC_SHA1_32 inline:hweP1A8b+BcmFBof5pwqvCH01ei/
wskd11by7Mgc
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:LlzD5S/+xcqcyp+RCPAV6Fjlov-
JRCbHYaaW6XgZv
a=rtpmap:124 SPEEX_ISAC/16000
a=rtpmap:103 ISAC/16000
a=rtpmap:96 speex/8000
a=fmtp:96 speexusejitterbit=
a=rtpmap:120 opus/48000/2
a=fmtp:120 sprop-stereo=0; minptime=10; maxaveragebitrate=12000; ste-
reo=0
a=rtpmap:122 ISPXE1/16000
a=rtpmap:121 ISPXE1_WB/16000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:105 CN/16000
a=rtpmap:127 red/8000
a=fmtp:127 useadaptivefec=
a=maxptime:60
a=ssrc:1788159847 cname:TFQ+rHHDQD4Ew5K+
a=ssrc:1788159847 msid:stream_label audio_label
a=ssrc:1788159847 mslabel:stream_label
a=ssrc:1788159847 label:audio_label
```

The codec preference is clearly different from iOS. Let's put them close to each other for comparison (iOS first, then Android)

```
m=audio 1 RTP/SAVPF 124 103 96 120 122 121 119 117 116 125 0 8 105 127
```

```
m=audio 1 RTP/SAVPF 119 124 103 96 120 122 121 117 116 125 0 8 105 127
```

Both clients support the same set of codecs, but have different preferences. Also, it turns out that this codec 119 has no associated a=rtpmap line, so we don't have a name for it even.

When not knowing what codec is used, we can look at the relative distribution of the RTP payload lengths:

This is quite characteristic. Many small packets with a payload length of 22 bytes, a number of groups of packets which are about 10-11 bytes apart in terms of payload size and another big spike at 253 bytes.

Sometimes, using a search engine to comb the Internet can be more effective. When you search for "webrtc codec 119", this leads to the subversion history of the webtrc.org library:

cricket::AudioCodec(119, "ISACLC", 16000, 40000, 1, 16),

This is a low-complexity variant of ISAC. Better suited for old and cheap Android hardware. It all makes sense.

SESSION 5

Dump: messenger-video-ios-android.pcap & messenger-video-android-ios.pcap

While writing this, Facebook rolled out video calling. The big question here is what video codec is used. VP8 is rejected by some parties for patent liability reasons. Check the notes from the Hawaii IETF meeting if you are really interested.
On iOS devices, H.264 hardware encoders might be available whereas VP8 hardware encoders might work on Android. So do we see different codecs?

First, let's look at a video call from the iOS app to the browser so we can see the SDP:

```
type: offer, sdp: v=0
o=- 7780344328953428983 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE audio video
a=msid-semantic: WMS stream_label
m=audio 1 RTP/SAVPF 124 103 96 120 122 121 119 117 116 125 97 0 8 105
127
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:f2lzrp6v4XaE3IhB
a=ice-pwd:HQFn6agwB83M8+BPXzxO+8F6
a=ice-options:google-ice fb-force-5245
a=fingerprint:sha-256 2E:55:72:F5:3C:1D:32:51:8B:65:A4:8A:73:C5:1B:81:6
7:F5:29:1E:EF:24:76:F4:7D:6C:78:41:22:64:C4:1F
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=sendrecv
a=mid:audio
a=rtcp-mux
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:6RNtdNQ90bFPeMBDLzGn2QSc6q/
EUOygytdYqddN
a=rtpmap:124 SPEEX_ISAC/16000
a=rtpmap:103 ISAC/16000
a=rtpmap:96 speex/8000
a=fmtp:96 speexusejitterbit=
a=rtpmap:120 opus/48000/2
```

```
a=fmtp:120 maxaveragebitrate=12000; minptime=10; stereo=0; sprop-ste-
reo=0
a=rtpmap:122 ISPXE1/16000
a=rtpmap:121 ISPXE1_WB/16000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:105 CN/16000
a=rtpmap:127 red/8000
a=fmtp:127 useadaptivefec=
a=maxptime:60
a=ssrc:2351689754 cname:x2I4t0ECzvKW5mvG
a=ssrc:2351689754 msid:stream_label audio_label
a=ssrc:2351689754 mslabel:stream_label
a=ssrc:2351689754 label:audio_label
m=video 0 RTP/SAVPF 100 126 123
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:f2lzrp6v4XaE3IhB
a=ice-pwd:HQFn6agwB83M8+BPXzxO+8F6
a=ice-options:google-ice fb-force-5245
a=fingerprint:sha-256 2E:55:72:F5:3C:1D:32:51:8B:65:A4:8A:73:C5:1B:81:6
7:F5:29:1E:EF:24:76:F4:7D:6C:78:41:22:64:C4:1F
a=extmap:2 urn:ietf:params:rtp-hdrext:toffset
a=recvonly
a=mid:video
a=rtcp-mux
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:6RNtdNQ90bFPeMBDLzGn2QSc6q/
EUOygytdYqddN
a=rtpmap:100 VP8/90000
a=rtcp-fb:100 ccm fir
a=rtcp-fb:100 goog-remb
a=rtcp-fb:100 nack
a=rtpmap:126 red/90000
a=rtpmap:123 ulpfec/90000
a=ssrc:3764505106 cname:x2I4t0ECzvKW5mvG
a=ssrc:3764505106 msid:stream_label video_label
a=ssrc:3764505106 mslabel:stream_label
a=ssrc:3764505106 label:video_label
```

So we have a video m-line but it is set to recvonly, i.e. the device does not want to receive video. Yet, it seems to have added a local video stream as can be seen by the a=ssrc lines.
And indeed, the call is shown as audio-only by the browser. For now at least!
In the answer, the browser rejects the video m-line by setting the port to 0.

There is just one codec here and it is VP8 with an RTP payload type of 100 and redundant data with payload type 126.

Let's go back to calls between mobile devices. First is iOS calling Android. The video quality is good, the framerate is high enough to appear fluent.

We can see two SSRCs here:

- 0xed85e125 sent from the Android device and
- 0x29f266b4 sent from the iOS device.
- The codec bandwidth ramps up slowly from 125kbit/second to about 500kbit/s during the 40 seconds of the call.

In the other direction, we have the SSRCs

- 0xf47f8251 sent from the Android device and
- 0xfb796208 sent from the iOS devices.

Note that video is sent by the Android device only for thirty seconds. At that point the camera was turned off which stopped the video packets. The bitrate of the iOS device continues to ramp up to about 800kbit/s which suggests there is no attempt to artificially limit this via a b=AS setting. Ironically, this is still combined with the ISAC LC codec.

CONCLUSIONS

The optimization for the user experience on mobile is quite impressive here, and shows great knowledge of the internals of the webrtc.org library. In addition, a great deal of effort is made to ensure interoperability among all platforms.

There are three major items which are worth pointing out.

First, the tuning of both the Opus and ISAC codecs is interesting. Is this an optimization based on Facebook's experience with mobile platforms? The increased framesize leads to larger packets, which might get dropped less frequently on mobile networks. The preference of ISAC over OPUS and the tuning of Opus to low-bitrates and mono seems to be an attempt to reduce the complexity for low-end mobile devices, as well.

Second, the use of the old relay protocol saves some round-trips compared to standard TURN. In order to authenticate, TURN requires a nonce value from the server in order to authenticate which means that in practice it takes two full round-trips between the client and the server to get a relay candidate. Given the widespread usage of nonce-like credentials this might be something that can be optimized further to decrease the connection startup time. Combined with a relay-first strategy as used by WhatsApp this could even allow warming up the transport before the called user accepts the call. The security concern here is the disclosure of the user's IP address. Disclosing the TURN server's IP is less of a concern, obviously.

Finally, the lack of DTLS for calls between mobile clients is unfortunate. It might be a reaction to DTLS being slow when there is packet loss or when fragmentation of the client hello is necessary. SDES is a solution to this problem, however it lacks important properties such as Perfect Forward Secrecy. Additionally, the signaling server knows the key needed to decrypt a session, which could create headaches for Facebook since law enforcement agencies might ask for the logs containing a key to retroactively decrypt a wiretapped session.