FACETIME 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. Clearly, Apple's Facetime is not one of those, but it has some interesting lessons anyway.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

At the risk of sounding like a broken record, thanks to the usual suspects, Serge Lachapelle, Justin Uberti, Harald Alvestrand for feedback. Also to Gustavo Garcia for enduring me while talking about some of the more boring part. And of course, Lynn Fisher for the cover page and Amy Lynn Taylor for taking care of the typesetting.

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

SUMMARY

Facetime is Apple's answer to videochat, coming preinstalled on all modern iPhones and iPads. It allows audio- and videocalls over WiFi and, since 2011, also over 3G.

In a nutshell, Facetime

- is quite impressive in terms of quality,
- requires an open port (16402) in your firewall as documented here,
- supports iOS and MacOS devices only, rings on multiple devices,
- is separate from the messaging application, unlike WhatsApp and Facebook Messenger,
- announces itself by sending metrics over an unencrypted HTTP connection. Dear Apple, have you heard about pervasive monitoring?
- presumably still uses SDES (no signs of DTLS handshakes, but I have not seen a=crypto lines in the SDP either).

Since privacy is important, it is sad to see a complete lack of encryption in the HTTP metrics call. Continuing to use SDES almost two years after the IETF at it's berlin meeting decided that WebRTC MUST NOT support it is sad. The consensus on this topic during the meeting was unanimous. For more background information, see either Victor's article on why SDES should not be used or dive into Eric Rescorla's presentation from that meeting comparing the security properties of both systems.

Facetime has been analyzed earlier, when being introduced back in 2010 as well as more recently in 2013. While the general architecture is still the same, Facetime has evolved over the years, adding new codecs like H.265. What else has changed? And how much of the changes can we observe? Is there anything those changes tell us about potential compatibility with WebRTC?

Like WebRTC, Facetime is using the ICE protocol to work around NATs and provide a seamless user experience. However, Apple is still asking users to open a certain number of ports to make things works. The interpretation of ICE is slightly different from the standard and like Whatsapps has a strong preference for using a TURN servers to provide a faster call setup. Most likely, SDES is used for encryption.

For video, both the H.264 and the H.265 codecs are supported, but only H.264 was observed when making a call on a Wifi. The reason for that is probably that, while saving bandwidth, H.265 is more computationally expensive. One of the nice features is that the optimal image size to display on the remote device is negotiated by both clients.

For audio, the AAC-ELD codec from Fraunhofer is used as outlined on the Fraunhofer website. In nonscientific testing, the codec did show behaviour of playing out static noise during wifi periods of packet loss between two updated iPhone 6 devices

The signaling is pretty interesting, using XMPP to establish a peer-to-peer connection and then using SIP to negotiate the video call over that peer-to-peer connection (without encrypting the SIP negotiation). This is a rather complicated and awkward construct that I have seen in the past when people tried to avoid making changes

to their existing SIP stack. Does that mean Apple will take a long time to make the library used by Facetime generally usable for the variety of use cases arising in the context of WebRTC? That is hard to predict, draw your own conclusions.

Facetime offers an impressive quality and user experience. Hardware and software are perfectly attuned to achieve this. As well as the networking stack as we will see...

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 iOS devices running the Facetime 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.

Locally, an iPad Mini 3 device with IP 172.16.42.187 and an iPhone 5 with IP 172.16.42.185 are used.
The remote peer at 71.230.89.52 is using an iPad as well which has a local address 192.168.42.25

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: facetime-caller.pcap

The local client (IP 172.16.42.187) is calling. The peer's public IP is 71.230.89.52, the local address is 192.168.42.25.

The first interesting packet is #60. It is a TURN ALLOCATE request. Unlike common WebRTC requests, it has the XOR-RELAYED-ADDRESS field set. And contains a reservation token. The lifetime of the requested allocation is 60s which is shorter than the default ten minutes.

Comment

Instead of vending authentication nonces as described in draft-uberti-behave-turn-rest, the signaling server seems to give out candidates directly. This avoid several roundtrips in comparison as the candidate can be signaled to the peer without any delay. Given the lack of a username it is unclear how authentication works. The reservation token might be generated and verified using a similar shared-secret token scheme however.

Compared to the typical WebRTC procedure of creating an allocation, there is another difference. A WebRTC client such as Chrome or Firefox will follow the recommendation outlined in RFC 5389 when creating an allocation, i.e. it will send an allocate request without a USERNAME attribute. In response to that, the server will challenge the client to repeat the request with a nonce and an authentication realm. Due to this, creating an allocation in WebRTC takes two full roundtrip times.

The TURN server responds to this request in packet #82.
Next, the caller creates a TURN channel with the peer's public ip address and port 16402 in packet #121. This seems to be caused by the signaling traffic received in packets #110 and acked in #120.

Comment

The signaling traffic runs on TCP port 5223 which suggests it is using XMPP over TLS. It seems odd to create a channel for the peer's public ip and the default port used, unless the caller knows the callee can send data from that IP and port.

The caller immediately sends a CHANNELDATA TURN message on the newly created channel without waiting for a success response. Wrapped inside this is an STUN binding request, using a 20 byte binary username, four unknown attributes (numbers 0x8001, 0x8002, 0x8003 and 0x8009) and an ice-controlling (0x802a) attribute.

Comment

The 20 byte username is an indication that a pre-RFC 5245 version of ICE is used, as usernames generated according to that always contain a colon to separate the local and remote username fragments.

Packets #132 and #142 show further attempts to send binding requests. Note that the client has not yet received a response to the channel binding requests so it retries that in packet #156 and gets a response in #158.

In packet #160 the client receives a CHANNELDATA message from the TURN server. It contains a BINDING RESPONSE from the other peer. The username is reversed from the original request, the packet also contains a MAPPED-ADDR (with a value containing the public ip of the TURN server and a port of 3489) and several unknown attributes (0x8001, 0x8003, 0x8004, 0x8005 and 0x8009).

Packet #161 shows the client sending a binding requests to the peer's local IP address.

Comment

The username fragments are different from the ones observed in packets #122 and #160. This suggests that the username fragments are different per candidate, which, again, indicates a pre-RFC 5245 version of ICE.

Packet #162 shows another CHANNELDATA message. This is a binding requests from the peer and shows (in addition to some of the unknown attributes an ICE-CONTROLLED attribute.

Packet #164 shows yet another CHANNELDATA message. This one contains a binding request with the USE-CANDIDATE set. So the clients have converged on a candidate. At this point, the clients have a relayed connection.

Packet #169 shows another CHANNELDATA message being sent by the client. It contains a SIP INVITE request, sent unencrypted. This is not a surprise given the earlier analyses. The INVITE request uses compressed header names and an SDP which is encoded using the deflate mechanism.

The decompressed version is shown here:

```
v=0
o=GKVoiceChatService 0 0 IN IP4 71.15.163.132
s=mobile
c=IN IP4 71.15.163.132
b=AS:2000
```

```
t=0 0
a=FLS;VRA:0;MVRA:0;RVRA1:1;AS:2;MS:-1;LTR;CABAC;CR:3;LF:-1;PR;AR:4/3
,3/4;XR;
a=X_FLS:123 FLS;VRA:0;MVRA:0;RVRA1:1;AS:2;MS:-1;LTR;CABAC;CR:3;LF:-1
;PR;AR:4/3,3/4;XR;
a=X_FLS:100 FLS;RVRA1:0;PR;LF:-1;CR:1;CF:2;AR:4/3,3/4;XR;
a=DMBR
a=CAP
m=audio 16402 RTP/AVP 104 105 106 9 0 124 122 121 119
a=rtcp:16402
a=fmtp:AAC SamplesPerBlock 480
a=rtpID:1911967383
a=au:65792
a=fmtp:104 sbr;block 480
a=fmtp:105 sbr;block 480
a=fmtp:106 sec;sbr;block 480
a=fmtp:122 sec
a=fmtp:121 sec
a=fmtp:119 sec
m=video 16402 RTP/AVP 126 123 100
a=rtcp:16402
a=rtpID:2853556652
a=rtpmap:126 X-H264/90000
a=fmtp:126 imagesize 0 rules 15:320:240:320:240:15
a=rtpmap:123 H264/90000
a=fmtp:123 imagesize 0 rules 15:320:240:320:240:15
a=rtpmap:100 HEVC/90000
a=fmtp:100 imagesize 0 rules 15:320:240:320:240:15
a=imageattr:126 send [x=320,y=240,fps=30] [x=480,y=368,fps=30]
[x=640,y=480,fps=30] [x=1024,y=768,fps=30] [x=1280,y=720,fps=30]
recv [x=320,y=240,fps=15] [x=640,y=480,fps=30] [x=1024,y=768,f-
ps=30,q=1.00] [x=1280,y=720,fps=30]
a=imageattr:123 send [x=320,y=240,fps=30] [x=480,y=368,fps=30]
[x=640,y=480,fps=30] [x=1024,y=768,fps=30] [x=1280,y=720,fps=30]
[x=320,y=240,fps=15,i=1] recv [x=320,y=240,fps=15]
[x=640,y=480,fps=30] [x=1024,y=768,fps=30,q=1.00]
[x=1280,y=720,fps=30] [x=320,y=240,fps=15,i=1]
a=imageattr:100 recv [x=320,y=240,fps=15,i=1]
```

Compared to the 2013 analysis, the most significant change is the imageattr. This negotiates the supported video resolutions for H264. The HEVC codec, also known as H265, was added but without negotiating resolutions.

Packets #170, #171, #172 and #173 show a series of SIP 100 (trying), 180 (ringing) and 200 (Ok) and an ACK . The SDP body of the 200 response is shown:

```
v=0
o=GKVoiceChatService 0 0 IN IP4 71.230.89.52
s=mobile
c=IN IP4 71.230.89.52
b=AS:2000:2000
t=0 0
a=X_FLS:123 FLS;VRA:0;MVRA:0;RVRA1:1;AS:2;MS:-1;LTR;CABAC;CR:3;LF:-1
;PR;AR:4/3,3/4;XR;
a=X_FLS:100 FLS;RVRA1:0;PR;LF:-1;CR:1;CF:2;AR:4/3,3/4;XR;
a=DMBR
a=CAP
m=audio 16402 RTP/AVP 104 106 121 122 119
a=rtcp:16402
a=fmtp:AAC SamplesPerBlock 480
a=rtpID:3323451416
a=au:65792
a=fmtp:104 sbr;block 480
a=fmtp:106 sec;sbr;block 480
a=fmtp:121 sec
a=fmtp:122 sec
a=fmtp:119 sec
m=video 16402 RTP/AVP 123 100
a=rtcp:16402
a=rtpID:3201069193
a=rtpmap:123 H264/90000
a=fmtp:123 imagesize 0 rules 15:320:240:320:240:15
a=rtpmap:100 HEVC/90000
a=fmtp:100 imagesize 0 rules 15:320:240:320:240:15
a=imageattr:123 send [x=1024,y=768,fps=30] [x=320,y=240,fps=15,i=1]
recv [x=1024,y=768,fps=30,q=1.00] [x=320,y=240,fps=15,i=1]
```

Compared to the offer, the clients have picket a set of resolutions here and also removed the X-H264 codec as well as some audio codecs like G.722 and PCMU.

Comment

Note that this 'SDP' actually contains spec several violations such as 'b=AS:2000:2000', including codecs for which there are no rtmpap lines or invalid usage such as a=fmtp:AAC.

Let's take a step backward and look what just happened. Both clients have negotiated a data connection via a TURN and established an ICE connection. On this connection, they are now using SIP to establish an audio-video connection.

Comment

Ironically, doing something similar with WebRTC is described in [this WebRTCHacks article] I wrote recently. Negotiate a datachannel via a TURN server, then negotiate a second connection over that channel.

Also, this design is similar to the one described in the [SOX extension] to the XMPP protocol proposed by Peter Saint-Andre, Joe Hildebrand and Cullen Jennings. One of the main advantages of that approach was that it could work with existing SIP stacks with minor modification, replacing the UDP transport with an XMPP one.

Apple seems to have taken a similar approach, maybe we are seeing the heritage of the old [iChat AV] even.

Subsequently, in stun packets #175 to to #279, the clients are exchanging RTP data wrapped in channel data messages. No DTLS fingerprint is exchanged which implies the usage of SDES. Note that the key for SDES is not established in the SDP here. That is good because that SDP is exchanged without any transport encryption.

In packet #280, the client receives a STUN binding request from the peer's public IP address. It responds and subsequently sends a binding request of its own in packet #283, containing a USE-CANDIDATE attribute. Meanwhile, both clients continue to send data via the TURN server.
The client receives a response in packet #299 and switches the media stream to the non-relayed connection. Note that there is no signaling traffic so this is not an ICE restart.

Comment

We have seen this pattern before with Whatsapp. The client uses a TURN server first, then establishes a P2P connection and switches the RTP traffic to that. Unlike Whatsapp, this is done using standard standard TURN and ICE.

Decoding the RTP traffic got significantly easier at this point. The filter for this is:

```
(ip.addr eq 71.230.89.52 and ip.addr eq 172.16.42.187) and (udp.port
eq 16402)
```

Note that unlike when WebRTC is used, we are not seeing any STUN BINDING requests for consent check after the initial exchange.

What we are seeing are the following payload types which can be mapped using the SDP observed earlier:

- 104 -- even though the rtpmap line is missing, the a=fmtp:104 sbr (for spectral band replication) implies the AAC-ELD audio codec
- 119 -- another audio codec; since this is not in the SDP it is hard to tell what it is
- 123 -- we can actually see an rtpmap line for this, it is H.264. Given that H265 is supported and negotiated by both clients one would have expected the more modern codec here.

RTCP packets are sent roughly every five seconds by both clients. This can be filtered with

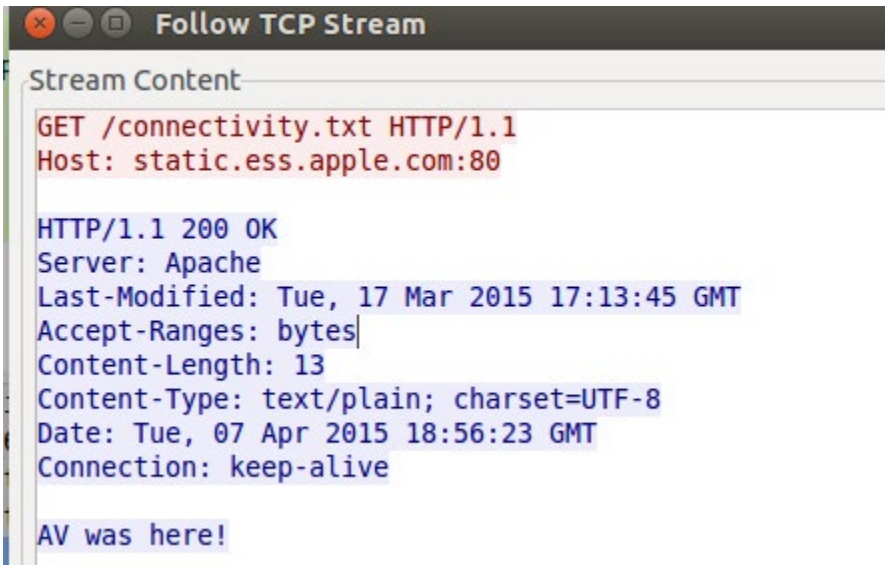
```
udp.port eq 16402 and rtcp
```

which shows some packets (e.g. packet #777) containing both a sender report (SR) and source description (SDES).

Packets #5786 and #5790 show the SIP BYE and the corresponding ack which end this session. The TURN channel is deallocated by the client in packet #5791 by sending a refresh request with a lifetime of 0.

Additionally, the capture show some UDP traffic on port 16403 before the call. The data part of the UDP packet is always padded (with 0xFF) to 16 bytes. While it is not clear what this traffic is used for, the port is listed both for the Gamecenter and IChat AV, not Facetime, in this list of UDP and TCP ports used by Apple software. It is probably unrelated to the call.

With all this udp traffic it is very easy to miss an HTTP request in the middle of all this (packet #64):



Comment

Seriously Apple, announcing the fact to someone uses facetime to anyone who can monitor unencrypted traffic on the internet... have you heard of pervasive monitoring?

While the use of an VoIP client is already obvious to an attacker that can listen through the use of SIP and RTP using unencrypted HTTP for metrics is just unnecessary.

SESSION 2:

Dump: facetime-callee.pcap

The local client (IP 172.16.42.187) is called. The peer's public IP is 71.230.89.52, the local address is 192.168.42.25.

Overall, this session is similar to the first one. Notably the SIP offer is sent from the same client as before, despite the role reversal.

Noteworthy that the client (as callee) is continuing to send STUN binding requests to the callers local ip which can be seen by setting the filter to:

```
stun && ip.dst_host == 192.168.42.25
```

The ICE-CONTROLLING flag is set here. This might be showing that the client is searching for a lower latency path.

SESSION 3:

Dump: facetime-samenet.pcap

This session captures two clients on the same network. One of the clients is the aleady familiar 172.16.42.187, acting as callee, the calling peer has the IP address 172.16.42.185.

In packet #71, the client is sending a binding request to the peer's public IP address on port 16402. The role for this is ICE-CONTROLLED.
The client is also quickly sending a binding request to the peer's local ip address in packet #82 and 101. This is unexpected from the first two sessions. Maybe the fact that, from the ip addresses both clients might be on the same network influences the timing for this.

It takes quite a while until these are answered in packets #169 and #171. Possibly this indicates that the peer needs to get the ice-ufrag and ice-pwd via the signaling channel before it is able to answer.
With those requests being unanswered for now, the client starts allocating a relayed candidate in packet #135. The response in packet #138.

In the STUN packets following the responses in packets #169 both clients establish an ICE connection. Yet, the second client starts allocating a TURN candidate in packet #178.

Packet #182 shows the USE-CANDIDATE attribute. Both clients have established an ice connection now. The first client (ICE-CONTROLLED) deallocates the TURN channel in packet 229. Note that the controlling client does not do this, it even sends a channel bind request for the public IP of the NAT router (and port 1024) behind which both clients are in #252. It is unsuccessful though, getting an allocation mismatch. It gives up shortly afterward by sending a refresh request with a lifetime of 0.

Subsequently, both clients keep sending stun binding requests to the public IP of the router on port 16402.

In packet #192 the SIP invite is sent and subsequently acknowledged. From there, this dump shows the already familiar behaviour.

SESSION 4

Dump: facetime-samenet-nop2p.pcap

This session captures two clients on the same network with direct P2P traffic supposedly blocked. One of the clients is the aleady familiar 172.16.42.187, acting as caller this time, the called peer has the IP address 172.16.42.185.
iptables is used to block P2P traffic:

```
iptables -I INPUT -s 172.16.42.185 -d 172.16.42.187 -j DROP
iptables -I INPUT -d 172.16.42.185 -s 172.16.42.187 -j DROP
```

Using ‘stun’ as wireshark filter, we can see that the clients attempt to establish a P2P session by sending binding requests. In packet #357, a binding request is sent to the peer’s local ip address which is dropped by the iptables rules.
In packet #364, a binding request is sent to the public IP of the peer which is responded to with an ICMP destination unreachable in #366. Since the client uses a hardcoded port 16402 this is possible without sending a STUN binding request to a STUN server. The public IP is probably obtained from the signaling server instead, possibly using a mechanism like as described here.
Packets #417 and #418 show the TURN allocation from the first client. Note the XOR-MAPPED-ADDR which equals to the public ip of the client and port 16402.

Packets #439 and #448 show the TURN allocation from the second client. Note that the XOR-MAPPED-ADDR is equal to the public ip of the client and port 16402 again.

Comment

For this to work the NAT router needs to be able to map to the internal IP based on the senders public ip AND port for the same destination ip and port.

Packet #475 shows a TURN channel binding. No TURN permissions are created, this mechanism is not used. The channel is created for the server-reflexive candidate of the peer.

Comment

This is surprising. Creating the channel for the server-reflexive candidate of the peer would not work for the (rare) scenarios where there is a TURN (TCP/TLS) servers involved on each side.

Packet #508 shows another channel binding, similar to packet #475. The response in packet #589 arrives somewhat late, about 0.8 seconds after the request. Actually it seems like the original request is lost and this is resent (without changing the transaction ID) in packet #578. This does not stop the client from assuming that the channel binding is a success and send messages on the channel already.
Following the conversation with this TURN server further shows that the client sends a channel bind refresh with a lifetime of 0 in packet #585 to release the channel.
It turns out that despite blocking the direct traffic with iptables, the clients were able to find a way around this without resorting to using a TURN server. Here is what likely happened...

In packet #556 we see a BINDING request that the other client responds to in packet #557. Previous packets by the same client had not been answered as can be seen when setting the filter to

```
    (ip.addr eq 172.16.42.187 and ip.addr eq 172.16.42.185) and udp.
 port eq  16402
```

which also shows that the other client needed to send a binding request of its own before the binding request got through. Now remember our network setup. The pineapple is doing NAT from the 172.16.42.x network. So the clients just punched a hole in the NAT here.

So actually this session is a hands-on demonstration of how NAT works and how ICE works around it. Even if we managed to block P2P at the Pineapple, the NAT router in front of it would still be tricked...

SESSION 5

Dump: facetime-dropmidcall.pcap

In this session, the P2P traffic to the peer's public IP address (the same as in session 1) is blocked in the middle of the call using the following two iptables rules:

```
iptables -I FORWARD -s 71.230.89.52 -j DROP
iptables -I FORWARD -d 71.230.89.52 -j DROP
```

In terms of user experience, this is rather nicely done. The peer's video is blurred at first and then removed, and a spinning wheel appears during the reconnect.

The call starts as usual. The P2P traffic is blocked in packet #6216, it is the last packet received from the peer for a while. It takes the client about three seconds until it realizes something is wrong and starts using the TURN server again in packet #6893. Then it starts sending TURN CHANNELDATA messages to the TURN server using the same channel previously used.

Comment

Note that the allocation on which this channel has been created previously in packet #92 has been released in packet #150. I would have expected this to mean that the TURN server can release the socket. Given the architecture of the TURN servers used and the small number of predetermined ports given here it is quite likely that the TURN server never closes the socket and the knowledge of the peer's address associated with the channel might be enough to route packets. It does not seem like those packets reach their destination though.

Unwrapping the channel data it turns it this is an RTP packet with a sequence number 51593. Which is the same RTP sequence number used in packet #6892. So the client is trying to send the same packet both directly as well as over the TURN channel. This can be observed for many of the packets following this one.

Comment

This seems like a bad idea as it will send the audio and video data via the same upstream twice. If the P2P and TURN connections were on different routes this might make sense but as is it does not.

It does not seem like the data reaches the peer, at least no data from the peer reaches the client. Possibly this is due to the release of the allocation earlier.

In packet #9252 (sixteen seconds after P2P traffic was blocked) we can see some signaling traffic, also subsequently.

Finally, about 18 seconds after the interruption, the client gives up by sending a SIP BYE to the peer via the TURN server in packet #9598. This is repeated in packets #9605, #9623 and #9633, no response is ever received.

In packet #9600, the call is restarted from scratch using the already familiar pattern of creating an allocation using a relay token. The ports are still blocked so no switch to P2P happens.

Comment

Instead of restarting from scratch, I would have expected an ICE restart here. Taking almost twenty seconds to detect a broken connection seems quite a long time, on the other hand this scenario is clearly engineered and might not happen in the real world often enough to optimize it.

CONCLUSIONS

We can again see some differences in the way ICE works here:

- the order in which the candidates are checked is different
- the reservation token mechanism allows signaling candidates faster
- TURN channeldata is sent without waiting for the result of the TURN binding request.

Let us look at each of those in turn.

The order of candidate checks is different from what one would expect. It seems very similar to what Whatsapp did with going to the TURN server first, but still takes local networks into account. It does not use STUN, but that is largely since it uses a hardcoded port 16402 and the public IP can be obtained by other means (such as the signaling server) then.
When we assume that TURN servers always play a role, do we even need STUN? Instead, can we skip allocating those and discover peer-reflexive candidates from the ports used to send to data to the peer's relay candidates?

Whereas WebRTC (roughly) checks in the order "host candidates, then server-reflexive, then relay" (which is the same as the priority), the order is rather "check host candidates if peers seem to be in the same network, then check TURN servers, try server-reflexive connection last". Checking host candidates first if there is a high chance of success makes sense and is a nice optimization which makes things look good if both clients are in the same network.
Basically we are seeing the same pattern we saw earlier in the Whatsapp investigation but using ICE as a framework.

The use of the TURN reservation-token mechanism is very interesting. In WebRTC, creating a TURN allocation takes two round-trips. In the first round trip, the client tries to create an allocation without username and credential and gets challenged by the server (together with a nonce to avoid replay attacks). In the second round trip, the client retries the allocation with a username, the nonce and signs the request with the password. What it gets back is the candidate ip and port. The timing of this can be seen very nicely in the candidate allocation demo. If TURN becomes more important, having a fast way to allocate TURN candidates is crucial for a fast connection setup. Optimizing one of these round trips away is desirable obviously, but what if we can signal the candidate before allocating it even? Then the signaling latency is mostly hiding any allocation latency and we can establish the connection slightly faster.
The allocation token mechanism seems like a good way to do this. However, it would require a modification of the WebRTC API because currently there is no way to pass this information about the pre-allocated candidate and token from Javascript to the ICE agent.

Sending TURN channeldata without waiting for an acknowledgement that the channel could be created is interesting in terms of latency, too. Most of the time, it is going to work. And while we have seen it fail in one of the sessions, this "opportunistic" behaviour is still useful elsewhere. For example, if the callee is creating an allocation (remember: two round trip times) it does not really have to wait for the authentication round-trip to succeed before it starts sending the data (in this case: ICE consent

checks, sent as TURN send indications). This eliminates the effect of the two-RTT behaviour for the vast majority of cases. And when it does not work, ICE will retry.

Does the architecture of Facetime tell us anything about what effort will be required from Apple to implement WebRTC? Well, there are some points here. Some standards like DTLS and TURN/TCP are not used by Facetime. That is easy to fix though. Removing the proprietary bits from the TURN usage and upgrading ICE from its pre-RFC-5245 version to the standard version is going to take some effort and it is conflicting with the Facetime usage.

The software architecture is likely a bigger issue. Using this combination of XMPP and SIP is certainly not something done lightly, as is creates a number of issues. With the SOX proposal one of the key benefits was not having to modify the SIP stack which acted as control surface for the media engine. This might imply that Apple does currently not have a strong desire to do this. But to enable WebRTC, they are going to need to adapt their media engine to a variety of use cases. Which took Google a number of years.

But we are talking about Apple, they are known for big surprises.