WHATSAPP EXPOSED

Investigative Report

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 Videocalls.

Is Whatsapp one of those services? Read on...

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Thanks to Adam “evilpacket” Baldwin for lending me his Pineapple device which allowed me to easily dump packets and Lynn Fisher for designing the cover. Without Chad Hart who has been running the WebRTCHacks website and the inspiration provided by his teardown of the Amazon Mayday Service this writeup would not exist.

Special thanks to Serge Lachapelle for helping on this. Also for soliciting further feedback from Niklas Blum, Henrik Lundin and Justin Uberti.

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

SUMMARY

After some rumors (e.g. on techcrunch), Whatsapp recently launched voice calls for Android. This spurred some interest in the WebRTC world with the usual suspects like Tsahi Levent-Levi chiming in and starting a heated debate.

&yet conducted an assessment from April 10 to April 14, 2015, by capturing the network traffic from Whatsapp on Android using tcpdump and analyzing the traffic patterns.

Whatsapp is using the PJSIP library to implement the voice-over-ip functionality. The capture shows no signs of DTLS, which suggests the use of SDES. Even though STUN is used, the binding requests do not contain ICE-specific attributes. RTP and RTCP are multiplexed on the same port.

The audio codec can not be fully determined. The sampling rate is 16kHz, the codec bandwidth of about 20kbit/s and the bandwidth requirement when muted is the same.

An inspection of the binary using the `strings` tool shows both PJSIP and several strings hinting at the use of elements from the webrtc.org voice engine such as the acoustic echo cancellation (AEC), AECM, gain control (AGC), noise suppression and the high-pass filter.

This report shows a series of capture sessions.

- **Session 1:** the first capture session describes this in great detail with the captured client being the caller. Subsequent sessions are used to verify assumptions raised from this one.
- **Session 2:** repeats the same scenario but switches the role to callee.
- **Session 3:** a quick test with an unanswered call which shows the allocation behaviour.
- **Session 4:** tests muting to determine if that has any effect on the codec bitrate.
- **Session 5:** blocks P2P traffic in order to determine if the client stays on the conference bridge
- **Session 6:** repeats this in the middle of the call.
- **Session 7:** an attempt to verify whether the client can work in networks where port 3478 is blocked.
- Sessions 8 and 9: are capturing two clients in the same network.

At the end of the report, a list of conference servers is given and a quick overview of the functions found in the binary is provided.

The most impressive thing here is the optimization for a fast call setup by using a relay initially and then switching to a peer-to-peer session. This also opens up the possibility for a future multi party VoIP call which would certainly be supported by this architecture. And the relay server is called “conf bridge” in the binary.

In WebRTC, we can do something similar, but it is a little more effort right now. We can setup the call with iceTransports: 'relay' which will skip host and server-reflexive candidates. Also, it is guaranteed to always work (in conditions where WebRTC will work at all).

There are some drawbacks here in terms of round-trip-times due to TURN's permission mechanism. More on that in a future post.

Since this hides the IP addresses of the parties involved from each other, we might even establish the relayed connection and do the DTLS handshake before the callee accepts the call. This is known as transport warmup, it reduces the perceived time until media starts flowing.

Once the relayed connection is established, we can call updateIce to remove the restriction to relay candidates and do an ICE restart by calling createOffer again with the iceRestart flag set to true. This would trigger an ICE restart which might determine that a P2P connection can be established. While updateIce is currently not implemented in Chrome 42 the iceTransports: 'relay' behaviour can be simulated in Javascript. It turns out this strategy is entirely possible to implement and the switching behaviour from relayed to P2P just works.

Whatsapp's usage of STUN and RTP seems a little out of date. Arguably, the way STUN is used is very straightforward and makes things like implementing the switch from relayed calls to P2P mode easier. But ICE provides methods to accomplish the same thing, in a more robust way. Using a custom TURN-like functionality that delivers raw RTP from the conference bridge saves some bytes overhead for TURN channels, but that overhead is typically negligible.

Not using DTLS-SRTP with ciphers capable of perfect forward secrecy is a pretty big issue in terms of privacy. SDES is known to have drawbacks and can be decrypted retroactively if the key (which is transmitted via the signaling server) is known. Note that the signaling exchange might still be protected the same way it is done for text messages.

In terms of user experience, the mid-call blocking of P2P showed that this scenario had been considered which shows quite some thought. The echo cancellation is a serious problem though. The webrtc.org echo cancellation is capable of a much better job and seems to be included in the binary already. Maybe the team there would even offer their help in exchange for an acknowledgement… or awesome chocolate.

TERMS OF SERVICE

The WhatsApp Terms of Service located at http://www.whatsapp.com/legal/ do not forbid the usage of tcpdump:

> While we don't disallow the use of sniffers such as Ethereal, tcpdump or HTTPWatch in general, we do disallow any efforts to reverse-engineer our system, our protocols, or explore outside the boundaries of the normal requests made by WhatsApp clients.

The mention of Ethereal seems odd, it is the name of the Wireshark sniffer before 2006.

The download page of Whatsapp for Android shows a third party LGPL notice.

CAPTURE SETUP

To capture the traffic, a Wifi Pineapple Wifi router (Mark IV, the Mark V works as well) is used.
The Android device with Whatsapp installed is 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 &yet network. The IP 192.168.1.157 is assigned to the Pineapple by DHCP.

The Android device for running Whatsapp has the IP 172.16.42.124 unless indicates otherwise.

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.

GENERAL NOTES

Whatsapp is known to use an uglified variant of the XMPP protocol for signaling (see e.g. here). This can be seen e.g. in the PCAP for session 2 which shows TCP traffic on the standard XMPP port 5222. A custom encryption is used instead of TLS.

The overall session setup is as follows:

- query "TURN" servers / conference bridge
- establish session using conference bridge
- establish P2P session
- switch media stream to P2P session if that exists

SESSION 1

Dump: whatsapp.pcap
The calling party is located in Richland, WA. The called party is located near Stockholm. There was a faint far-side echo.

The session is started by the client querying the DNS record for e2.whatsapp.net in packet #10. It subsequently established a TCP connection to that server.

The voice-over-ip session begins about 20 seconds into the capture. In packets #70 to #85 the client is sending TURN ALLOCATE requests to eight servers:

```
185.60.216.2
185.60.218.2
31.13.82.48
31.13.84.48
173.252.75.2
31.13.90.48
31.13.66.48
31.13.86.48
```

A geoip lookup using the Maxmind GeoIP database locates most of those servers in Ireland, except for 173.252.75.2 which is located in Menlo Park, CA.

Notable is that each of those TURN ALLOCATE requests contains a single attribute of type 0x4000 (which is in the comprehension-required range defined by RFC 5389) with 66 bytes of data. The attribute data is the same for each request, packets #78 to #85 are resends.
With 106 bytes this violates the RFC 5389 rule that the packet length must be divisible by 4.

Packets #87 to #105 show the responses to those requests. In addition to the XOR-MAPPED-ADDR they contain an attribute 0x4002 with 8 bytes of data. The last byte

of the attribute value seems to be different for different servers. Unlike standard TURN, this does not contain an XOR-RELAYED-ADDR field.

Comment

Subsequently, one of those 'TURN' servers, 31.13.90.48, is used as a relay. This is not the server which the lowest RTT, the response time of 0.18s (#94, #95) is way higher than for another of the irish servers (packet #89, 0.09s).

The 0x400 / 0x4002 attributes might be some kind of authentication or session establishment.

In terms of the overall service architecture it would be interesting to find out whether those servers are the same for all users or if each user gets his own set, possibly based on the telephone number or something else.

Assuming they're sent by the signaling server, eight seems too many. Also the signaling server might be smarter and assign the client to a data center. The servers used vary per session (see other sessions), more than eight servers total but eight per session.

Packets #106 to #131 show some signaling traffic. Most likely, after gathering candidates, the client is sending an offer to the peer.

Packet #132 marks the beginning of the next phase. Note that this is sent on the STUN port by one of the STUN servers. At first, this seemed to be confusing. Using Wireshark's "Decode as" to decode this as RTCP reveals this and packet #133 to be two Receiver Reports sent by the callee and caller respectively. This might be some kind of handshake.

Packet #134 is the first actual RTP packet with an SSRC value of 0x0088a82d. More on the RTP streams later. Packet #135 can not be decoded by Wireshark.

Packet #146 is the first RTP packet from the callee, relayed with the TURN server. It is again followed by an undecodable packet in #147.

The RTP data is flowing on this udp port pair for about three seconds until packet #309. The traffic can be filtered with

```
(ip.addr eq 31.13.90.48 and ip.addr eq 172.16.42.124) and (udp.port
eq 3478 and udp.port eq 45395)
```

Several packets turn out to be RTCP Sender Reports. This means that RTCP-MUX (RFC 5761) is used.

While the first RTP traffic is flowing via the "TURN" server (which is not really acting as a TURN server here since it is accepting raw RTP not wrapped in a TURN send/data indication or a channel) the client is sending P2P binding requests. Packet #198 shows the first attempt to reach the ip address 192.168.0.106 (which is the local ip of the callee). The traffic can be filtered with

```
(ip.addr eq 172.16.42.124 and ip.addr eq 192.168.0.106) and (udp.
port eq 45395 and udp.port eq 35574)
```

Only the message integrity field is present, this means ICE (RFC 5245) is not used.

In packet #294 the client is attempting to send a binding request to the public ip of the peer. The filter for this traffic is

```
(ip.addr eq 172.16.42.124 and ip.addr eq 83.209.197.82) and (udp.port eq 45395 and udp.port eq 35574)
```

The first binding response is received in packet #300.

The next interesting packet is an RTP packet. It carries the same SSRC 0x0088a82d that was previously used on the relayed connection with the TURN server. The RTP sequence number is 12763. Clearing the filter shows that the caller has just switched away from the relayed connection, no further RTP packets are sent.

The callee sends its own binding requests in packets #302, #306 and #311 respectively. It then switches away from the relayed connection for its SSRC 0x6b0226ec in packet #315.

Comment

This call setup process is highly interesting. Basically, the client is using a relayed path which is considered the last resort by ICE first and switching away from it very soon.

The delay between the binding request to the public ip and the last signaling packet is quite notable. The client could probably start this check earlier or could have checked the server reflexive candidate before the host candidate. However, with RTP data already flowing there is no perceived call setup latency.

It is unclear how the servers route the RTP packets. Possibly this information is encoded in the 0x4000 / 0x4002 attributes used in the binding request and response.

This approach seems less robust than using TURN, but saves some RTT for channel allocation and permission creation in comparison.

The session is terminated around packet #500. In packets #504 to #511 the client sends STUN requests of type 0x0800 to each of the eight TURN servers. The value of this extension is the same as the value in the initial binding request.

RTP flows

Let's look at the RTP data flowing from the caller to the callee. The easiest way to look at the complete flow is by first setting the filter to

```
udp.port eq 45395
```

then choose "decode as" RTP and filtering for

```
rtp and rtp.ssrc == 0x0088a82d
```

Wireshark's RTP analysis tools seem to be unable to deal with the change of sender ip in this session.

The RTP payload type for this is 120. The DSCP flag is not set.

While the session is relayed via the TURN server, the RTP timestamp delta is 320 and the packet size is between 95 and 110 bytes.
In packet #283 the packet size increases to around 400 bytes and the marker bit is set. The RTP timestamp delta also changes to 2560 at 6–7 packets per second and goes back to 320 at in packet #392 with 50 packets per second. This implies a sampling rate of 16kHz.

> **Comment**
>
> The timing is quite interesting here. The packet size increases shortly before the serverreflexive peer candidate is checked (packet #294). It is quite interesting that there is no such attempt to do this when checking the host candidate, but on a LAN packet loss is quite rare.
>
> It is not clear why this is done. Other dumps showed the same behaviour after the switch so this might be unrelated. Blocking P2P traffic may shed some light on this behaviour.

The codec bitrate is roughly 20kbit/s in longer sessions, the net bitrate is around 50kbit/s and direction (which is also the result given in here).

AMR-WB matches this pretty well with a 16kHz sampling rate and a 19.85kbps rate (http://en.wikipedia.org/wiki/Adaptive_Multi-Rate_Wideband#AMR_modes).

However, AMR-WB usually has a constant RTP payload length whereas the payload length of the codec used here varies. The payload varies between 24 bytes and (for 20ms frames) to roughly 78 bytes.

Opus is typically used with a 48kHz sampling rate and requires less bandwidth when muted which is not the case here (see session 4). The same is true for ISAC, even though that can operate at 16k and is also used by Facebook Messenger.

SESSION 2

Dump: whatsapp2.pcap
This session reverses the roles, caller is located in Stockholm.

After the already familiar TURN allocation sequence, the RTP traffic starts in packet #89 with an RTCP RR. Unlike the behaviour in session 1, the next packet is not an RTCP RR from the peer but shows the client sending RTP traffic. The RTCP RR from the peer arrives in packet #99

> **Comment**
>
> This still supports the hypothesis that the client sends an RTCP RR before its own traffic.

The switch from relayed mode to P2P mode occurs in packet #213.

RTP Flows

The RTP flow with rtp.ssrc == 0x45d3e806 sent by the client is particularly interesting. It starts in packet #90 and sends some packets (#90 - #98) to the relay server. This is followed by a rather lengthy gap until packet #258 almost 2.7 seconds later. This is significant as packet #258 is the first packet sent by the callee on the P2P session and also switches the packetization. It seems that the callee is triggering the behaviour and the callers stream with rtp.ssrc == 0x163a906 responds to this in packet #278

> **Comment**
>
> This looks like this behaviour might be triggered by the callee, after not sending packets for an unusually long period of time. Note that there is no apparent gap in the RTP timestamp or sequence number between packets #98 and #258.

SESSION 3

Dump: whatsapp-noanswer.pcap
This session tests the behaviour when the caller does not accept the call.

This shows a one-sided allocation of a pseudo-TURN channel (packets #29-44 and #51-66) and the closing of the channel in packets #93 - #100. No media is sent.

SESSION 4

Dump: whatsapp-mute.pcap
This session was testing whether muting and unmuting affect the codec bitrate.

The switch happens faster than in the first session here, in packet #173 roughly 0.8 seconds after the first RTP packet. The increase of the timestamp delta from 320 to 2560 happens on the P2P connection.

> **Comment**
>
> This seems to be independent of the switch to the P2P connection. See below.

RTP Flows

There are two RTP flows there: rtp.ssrc == 0x1b841a7c which is the first stream sent and rtp.ssrc == 0x489e2b26.
In the first stream we have again a large gap (2.45 seconds) here, between packets #124 and #282 which marks the start of a series of packets with a timestamp delta of 2560 as already described in session 2.

Notable about both streams is that the muting is not visible in the codec bandwidth required which stays at 20kbit/s. This is different from the behaviour of e.g. OPUS.

SESSION 5

Dump: whatsapp-drop.pcap
This session is a test of what happens when the P2P connection is blocked.

This is trivial to achieve on the Pineapple using iptables before starting the call:

```
iptables -I FORWARD -s xx.xx.xx.xx -j DROP
iptables -I FORWARD -d xx.xx.xx.xx -j DROP
```

The P2P connectivity checks start in packet #148, this can be best seen by setting the filter to stun. In packets #148 and #149, two binding requests are sent to the peers local and server-reflexive candidate respectively. Subsequently (after a 0.5 second delay), only the host candidate is checked and only after giving up on this (packet #278) after 2.5 seconds the server-reflexive candidate is checked again (stun packets #279 until #324).

> **Comment**
>
> This seems like a bug.

The call is relayed during the whole time.

> **Comment**
>
> This supports the hypothesis that the purpose of this behaviour is to establish a connection using a failsafe relay channel first and then optimize by switching to P2P.

RTP Flows

The stream sent by the client with rtp.ssrc == 0x264895ff starts behaving oddly in packet #356. It sends each RTP packet twice, with the same sequence number and timestamp.

The stream sent by the peer with rtp.ssrc == 0x536DE9EC is using larger packetization for a short period of time in packet #3254 until packet #4164.

> **Comment**
>
> At the same time the other client is not attempting to do this so it is not a symmetric behaviour.

SESSION 6

Dump: whatsapp-drop-midcall.pcap
For this session, the P2P traffic is blocked mid-session to investigate the client's reaction to that. The call went silent, the application made a signal noise and the call was reestablished – in relayed mode. Echo got significantly worse at that point.

Before blocking, we again observe the by-now known behaviour of switching from the relay to P2P mode. The use of iptables can be directly observed in the RTP stream from the peer which can be filtered with rtp.ssrc == 0xbb48baa. The drop starts at packet #1864. It takes about five seconds until RTP packets are again received from the relay in packet #2426.
Packets #1889 and #1890 are again the malformed "AVB RTCP packets" which were already observed in the first session.

In packet #2389, five seconds after receiving the last RTP packet from the peer, the client starts to send data to the relay (duplicated packets again). In packet #2397 a P2P binding request is sent, this times out after three seconds. No binding request is sent to the peers local ip address.
The signaling channel is not used during this time period which differentiates this from an ICE restart. The last signaling packet is in #493, the next in #8236.

Comment

Falling back to the relay path might be something to try if an ICE restart does not result in any candidates from the peer. That way, both clients will end up with a P2P connection even if the signaling channel fails as well.

SESSION 7

Dump: whatsapp-stunblock.pcap
This session tests the behaviour when port 3478 UDP is blocked by running

```
iptables -I FORWARD -p udp --destination-port 3478 -j DROP
```

The client repeats the attempt to get an allocation but does not seem to fall back to TCP. The call is terminated by caller before callee accepts.

SESSION 8

Dump: whatsapp-both.pcap
This session captures two clients on the same network. This shows that both clients are using the same bridge, 173.252.75.2 and switch to a peer-to-peer connection afterwards.

Comment

This is not fully conclusive. A conclusive test would need to provide captures from two different locations. This creates a problem of displaying both captures in a synchronized way however.

SESSION 9

Dump: whatsapp-both-noanswer.pcap
This session captures two clients on the same network again. The caller ends the call before it is accepted by the peer. It can be observed that the callee (with the IP 172.16.42.124) establishes a connection to the conference bridge already.

Comment

This was a test whether some kind of early transport warmup is used. The biggest problem with early transport warmup is revealing the user's ip address without consent which is eliminated when warming up the transport via a TURN relay. This does not happen here, might be worth trying for WebRTC however.

CONFERENCE BRIDGES

The following 21 conference bridges were seen in the seven sessions (sessions 8 and 9 are not included in this):

```
173.252.75.2, 31.13.66.48, 31.13.82.48, 185.60.216.2, 31.13.90.48,
31.13.86.48, 31.13.84.48, 185.60.218.2, 31.13.71.48, 185.60.217.2,
179.60.192.48,    179.60.195.48,    31.13.70.48,    31.13.85.48,
31.13.100.14, 31.13.76.80, 173.252.121.1, 31.13.91.48, 31.13.83.48,
31.13.64.48, 173.252.114.1
```

Seven of those are located in a data center in Menlo Park, the rest are located in Ireland. With enough dumps it should be possible to enumerate to total number of servers and get an estimate about the total number of servers.

Since each call uses a set of eight servers, there could have been 56 different bridges used. Observing only 21 suggests some reuse and we can estimate an upper bound.
This problem is known as "mark and recapture" in biology and a Bayesian estimate (assuming two visits) yields a total number of approximately 27 3 5 servers (K=16, n=14, k=9).
This assumes that things like geographic location are not taken into account which, given that a client from the US is using a bridge in Ireland, seems reasonable. Using a multi-visit model might be more robust.

BINARY ANALYSIS

The APK of Whatsapp version 21223 was unpacked and the libwhatsapp.so file (for the armeabi-v7a version) was analyzed using the 'strings' utility to obtain a list of printable characters in the file. Notable highlights:

- Using Speex AEC, Using WebRTC AEC, Using WebRTC AECM, Using the echo suppressor, Using WebRTC AGC, Using WebRTC noise suppression, Using WebRTC high-pass filtering
  - This suggests some media components of the webrtc.org library are used.
  - Note that these strings are not from webrtc.org library itself.
- Dumping PJMEDIA capabilities
  - PJMEDIA is part of the PJSIP library.
- AES_CM_128_HMAC_SHA1_32, AES_CM_128_HMAC_SHA1_80
  - This suggests SDES is used for encryption, most likely with 32 bit MAC to reduce the packet size.
- Warning: codec priority set failed for Opus!
  - also for AMR, PCMU. This looks like the client supports Opus, AMR and PCMU. Unfortunately, this does not help deciding which of these codecs is used.
- conf bridge creation failed
  - this explains the call setup observed. It might also imply that multiparty calls are supported by the architecture.
- voip network change notified
  - switching from Wifi to 3g might be supported. Capturing might be hard, capturing a switch from the the remote end might be possible.
- udp relay, tcp relay
  - this implied that blocking UDP might lead to a TCP fallback which could not be observed.
- srtp_init: done
  - this suggests libsrtp is used. Also contains a hint to pjsips transport_srtp.c.
- Failed decoding crypto key from base64
  - another hint that SDES is used. This is from transport_srtp.c again
- callSetupT et al
  - looks like call statistics are gathered, potentially written to sqlite database

The APK even contains a LICENSE.txt file. The contents suggest that the client is licensed under the Apache license which is rather confusing.
In addition, a NOTICE.txt is listing third-party licenses for a few projects. PJSIP is not contained herein which may suggest the alternative commercial license was used. libsrtp and the webrtc.org library are not mentioned here either.

CONCLUSION

We took an in-depth look at Whatsapp's voice calling feature and while it is impossible to be 100% certain by observing network traffic flows and a binary, we learned that:

1. PJSIP is heavily used. Some components of the webrtc.org project are mentioned in the binary.
2. Whatsapp makes heavy use of transport warmup (i.e. direct to relay)
3. The SDES encryption scheme is used, raising some privacy concerns.
4. We cannot come to a conclusion around the codec that is in use, but invite the reader to contribute with their own findings.
5. Relay / STUN infrastructure is based in the USA and Ireland, which is interesting considering the user base geographies of Whatsapp.