10 MIN

DATE

16 FEBRUARY 2021

TELEGRAM FUZZING MEMORY-CORRUPTION

AUTHOR

POLICT

I'm a vulnerability researcher and exploit developer at Shielder. In my free time I enjoy backpacking

Hunting for bugs in Telegram's animated stickers remote attack surface

Introduction

At the end of October '19 I was skimming the Telegram's android app code, learning about the technologies in use and looking for potentially interesting features. Just a few months earlier, Telegram had introduced the animated stickers; after reading the blogpost I wondered how they worked *under-the-hood* and if they created a new image format for it, then forgot about it. Back to the skimming, I stumbled upon the **rlottie** folder and started googling. It turned out to be the Samsung native library for playing Lottie animations, originally created by Airbnb. I don't know about you but the combination of **Telegram**, **Samsung**, **native** and **animations** instantly triggered my interest in learning more 👀.

Executive summary

Research is one of Shielder's pillars – head over to our research page to learn more about our commitment to improve the security of the digital ecosystem.

What follows is my journey in researching the lottie animation format, its integration in mobile apps and the **vulnerabilities triggerable by a remote attacker against any Telegram user**. The research started in January 2020 and lasted until the end of August, with many pauses in between to focus on other projects.

During my research I have identified 13 vulnerabilities in total: 1 heap out-of-bounds write, 1 stack out-of-bounds write, 1 stack out-of-bounds read, 2 heap out-of-bound read, 1 integer overflow leading to heap out-of-bounds read, 2 type confusions, 5 denial-of-service (null-ptr dereferences).

All the issues I have found have been responsibly reported to and fixed by Telegram with updates released in September and October 2020:

- **Telegram Android v7.1.0 (2090)** (released on September 30, 2020) and later
- **Telegram iOS v7.1** (released on September 30, 2020) and later
- **Telegram macOS v7.1** (released on October 2, 2020) and later

Those updates include the fixes (the other types of clients are not affected by the vulnerabilities I have identified) – basically **if you have updated your Telegram client in the last 4 months you are safe**. If not, I recommend you to update it as soon as possible.

Table of contents

- Lottie by Airbnb
- RLottie by samsung, forked by Telegram
- Harnessing rlottie and building a corpus
- Fuzzing techniques and results
  - coverage-guided fuzzing
  - layman's guide to crash testcase minimization (excursus)
  - heap out-of-bounds write in VGradientCache::generateGradientColorTable
  - structure-aware fuzzing
- Telegram's animated stickers attack surface
  - how they patched it
- Conclusions

Lottie by Airbnb

Let's start from the original Lottie project by Airbnb, from airbnb.io/lottie:

> Lottie is a library for Android, iOS, Web, and Windows that parses Adobe After Effects animations exported as json with Bodymovin and renders them natively on mobile and on the web!

"As **json**" is particularly interesting here, I was expecting some tricky 90's proprietary binary specification but instead they chose to use one of the most common and simple formats to date. (This got me also wondering whether memory corruptions would be harder to find, but it was too early to tell!)

As we have read, a Lottie animation is defined as a JSON with some information such as the frame rate "**fr**" and the version identifier "**v**" at its root, while most of the juicy features lie in the "**layers**" array.

At its minimum, a Lottie animation looks like this:

```
{
    "v":" ",        // version identifier
    "fr":1,         // frame rate
    "ip":0,         // in-point
    "op":1,         // out-point
    "layers":[]     // the good stuff (tm)
}
```

This doesn't include any graphical element, but it's useful to have a bare-minimum example before getting complex (especially in structure-aware fuzzing, as we will discuss later).

Remember the "Adobe After Effects animations exported as json" part? If you open such an animation it contains a lot of useless information and animation's metadata, for example Adobe After Effects even supports "the Adobe ExtendScript language, which is an extended form of JavaScript" (!), which is included in the JSON but not supported by the Lottie parser we are going to talk about.

It's important to notice here that Lottie animations are widely used, though most of the time via static resources such as app's transitions and animations. Another important thing to notice is that other apps, such as **Signal**, **chose Airbnb's java/swift implementation**.

RLottie by Samsung, forked by Telegram

Here we arrive at Samsung's C++ library rlottie to parse Lottie animations. I'm not sure why Telegram's developers decided to use this implementation instead of Airbnb's, besides performance (and the chance to expose a 1-click native attack surface 😈). That being said, working with an open-source library will come in handy for setting up the fuzzing environment and triaging the crashes, something which is not as trivial to do in a black-box scenario.

RLottie doesn't support all of After Effect's features, however it is still actively maintained to this day, even though I'm not 100% sure what Samsung uses rlottie for besides probably Samsung Galaxy Watch Apps. (If you do know/find out where it's used let me know at @polict_! 👍)

By checking the README it's clear that writing the harness will be trivial; by looking at Telegram's integration it's even possible to copy the initialization settings and build a 1:1 stand-alone harness.

It's important to note here also that Telegram developers chose to fork the rlottie project and maintain multiple forks of it, which makes security patching especially hard. This will turn out to be an additional problem since the Samsung's rlottie developers **do not track security issues caused by untrusted animations** in their project because they are not "the intended use case for rlottie" (quote from https://gitter.im/rLottie-dev/community).

Harnessing rlottie and building a corpus

I had almost no experience in fuzzing before this research, so I started studying and learning about two of the main players at the time: AFL++ and LibFuzzer. The majority of entry-level writeups and walkthroughs available publicly were using AFL[++] so I started with it while learning more about the alternatives available. The first version of the harness was a ctrl+c/ctrl+v frankenstein but it worked well as a starting point:

```cpp
#include <rlottie.h>
#include <iostream>
#include <string>
#include <vector>
#include <array>

int entrypoint(std::string filename){

    auto player = rlottie::Animation::loadFromFile(filename, NULL);
    if (!player) {
        printf("error: renderer initialization failed\n");
        return 1;
    }

    // metadata[0] in Telegram/TMessagesProj/jni/lottie.cpp:130
    size_t frame_count = player->totalFrame();
    printf("frame count:\t%zu\n", frame_count);

    // default width and height
    uint32_t w = 512;
    uint32_t h = 512;

    // copied from https://github.com/Samsung/rlottie/blob/master/example/lottie2gif.cpp
    auto buffer = std::unique_ptr<uint32_t[]>(new uint32_t[w * h]);

    if (frame_count < 1){
        printf("no frames to render, quitting\n");
        return 1;
    }

    printf("starting...\n");
    for (size_t frame = 0; frame < frame_count; frame++) {
        rlottie::Surface surface(buffer.get(), w, h, w * 4);
        player->renderSync(frame, surface);
    }
    printf("done!\n");

    return 0;

}

int main(int argc, char **argv){
    if (argc < 2){
        printf("usage: %s <lottie.json>\n", argv[0]);
        return 1;
    }

    return entrypoint(std::string(argv[1]));
}
```

(Only later did I discover the perf_tips AFL++ documentation, I strongly recommend it to people starting out fuzzing!)

Having verified the harness was working, I started looking for animated stickers online to build a minimal corpus to start fuzzing: Telegram channels available as a webpage on `t.me/` URLS and lottie online communities were especially useful for scraping user-generated stickers in an automated `curl-grep-gzip` fashion.

Fuzzing techniques and results

Coverage-guided fuzzing

If there's one thing I have learned the hard way in my information security experience (and later again by reading twitter heh), it is that many times doing the laziest thing would have produced the same output as a sophisticated technique, but in way less time: this research was no difference.

To say it with a meme shamelessly stolen from infosucks and twitter

After instrumenting and improving the harness and launching afl-fuzz, **crashes started to appear in a matter of seconds**. I thought that if anybody was fuzzing it, they were either exploiting the issues or still looking for ASLR-breaking gadgets – but that's just a guess! 🤷

From the first crash triage cycle it seemed some issues could be serious: heap-based out-of-bounds read/write, stack-based out-of-bounds write and high-address SEGVs all looked promising, so I started investigating them while studying the code and continuously improving and keeping the fuzzer running. Most of the remaining issues were null-pointer dereferences not useful from an exploitation perspective, however in this context - as we will see later - they might become an annoying denial-of-service bug for non-technical users.

Layman's guide to crash testcase minimization (excursus)

After triaging and prioritizing the crashes I started analyzing the root-cause of each of them. The problem was that since the library parsed JSONs and skipped useless keys, the crashing testcase included a ton of unnecessary keys and values (imagine a single line 2KB JSON with multiple nested void keys/arrays/strings/objects 😳). At the beginning I thought of writing a JSON minimizer tool in python, but remembering the "**try lazy first**" way of thinking I hacked together halfempty, ASAN and `grep` to bruteforce their way to the minimized *still-crashing-in-the-same-way* JSON, and it worked pretty well! 🃏

Let's have a look at one example fed to halfempty:

```
#!/bin/bash
timeout -k1s 4s rlottie/parser-asan /dev/stdin 2>&1 | grep -q 'WRITE of size 4 at' && exit 0 || exit 1
```

(I could have added more filters to the grep (error type, $pc, stacktrace, ...) but it wasn't really necessary here...)

Afterwards I could simply run halfempty to bruteforce a minimized testcase:

```
halfempty --stable --zero-char=0x20 --output=min.json run_and_grep_hbof4write.bash raw.json
```

This helped because, without further checks besides checking for a SIGSEGV (`test $? -eq 139`), halfempty would have produced a minimized testcase which crashed rlottie with a null-pointer dereference (still a SIGSEGV but not what I was looking for).

Back to the fuzzing now...

Heap out-of-bounds write in VGradientCache::generateGradientColorTable

Let's walk through one of the most impactful issues I have found: a 4-bytes heap out-of-bounds write in `VGradientCache::generateGradientColorTable`. Here's a sample ASAN report snippet with a bit of context:

```
==24332==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x621000001130 at pc 0x0000005652a4 bp 0x7ffef2d69190 sp
0x7ffef2d69188
WRITE of size 4 at 0x621000001130 thread T0
    #0 0x5652a3 in VGradientCache::generateGradientColorTable(std::vector<std::pair<float, VColor>,
std::allocator<std::pair<float, VColor> > > const&, float, unsigned int*, int) rlottie/src/vector/vdrawhelper.cpp:159:25
    #1 0x574d5c in VGradientCache::addCacheElement(long, VGradient const&) rlottie/src/vector/vdrawhelper.cpp:125:30
    #2 0x573645 in VGradientCache::getBuffer(VGradient const&) rlottie/src/vector/vdrawhelper.cpp:87:24
    #3 0x569a39 in VSpanData::setup(VBrush const&, VPainter::CompositionMode, int) rlottie/src/vector/vdrawhelper.cpp:761:46
    #4 0x53b528 in VPainter::setBrush(VBrush const&) rlottie/src/vector/vpainter.cpp:140:22
    #5 0x5c2a15 in LOTLayerItem::render(VPainter*, VRle const&, VRle const&) rlottie/src/lottie/lottieitem.cpp:332:18
    #6 0x5c841e in LOTCompLayerItem::renderHelper(VPainter*, VRle const&, VRle const&)
rlottie/src/lottie/lottieitem.cpp:651:28
    #7 0x5c7744 in LOTCompLayerItem::render(VPainter*, VRle const&, VRle const&) rlottie/src/lottie/lottieitem.cpp:602:9
    #8 0x5c0348 in LOTCompItem::render(rlottie::Surface const&) rlottie/src/lottie/lottieitem.cpp:198:17
    #9 0x59107d in AnimationImpl::render(unsigned long, rlottie::Surface const&)
rlottie/src/lottie/lottieanimation.cpp:107:16
    #10 0x5922a5 in rlottie::Animation::renderSync(unsigned long, rlottie::Surface&)
rlottie/src/lottie/lottieanimation.cpp:206:8
    #11 0x68b146 in entrypoint(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
rlottie_parser.cpp:40:17
    #12 0x68b40e in main rlottie_parser.cpp:60:16
    #13 0x7f22916cebf6 in __libc_start_main /build/glibc-S9d2JN/glibc-2.27/csu/../csu/libc-start.c:310
    #14 0x41e439 in _start (rlottie/parser-asan+0x41e439)
```

The vulnerability stems from an incorrectly bounded loop (comments are mine):

```
bool VGradientCache::generateGradientColorTable(const VGradientStops &stops,
                                                float                 opacity,
                                                uint32_t *colorTable, int size)
{
    int                 dist, idist, pos = 0, i;
    bool                alpha = false;
    int                 stopCount = stops.size();
    const VGradientStop *curr, *next, *start;
    uint32_t             curColor, nextColor;
    float               delta, t, incr, fpos;

    if (!vCompare(opacity, 1.0f)) alpha = true;

    start = stops.data();
    curr = start;
    if (!curr->second.isOpaque()) alpha = true;
    curColor = curr->second.premulARGB(opacity);  // out-of-bounds value, curr->second is controlled
    incr = 1.0 / (float)size;                     // static
    fpos = 1.5 * incr;                            // static

    colorTable[pos++] = curColor;

    while (fpos <= curr->first) {                 // curr->first is controlled and pos is not checked to be < size,
leading to
        colorTable[pos] = colorTable[pos - 1];    // out-of-bounds write
        pos++;
        fpos += incr;
    }
    [...]
```

As we can see in the snippet, `pos` is not checked against `size` (the `colorTable` array size), leading to writing out-of-bounds 4 bytes after the end of the `colorTable` array allocated in heap memory.

Specifically, while `fpos`, `size` and `incr` are static, `curr->first` and `curr->second` come directly from the animated sticker but `colorTable` is an uint32_t array of static size 1024, hence it is possible to overwrite an arbitrary amount of heap memory after it by carefully using a float number as `curr->first` in the animated sticker file.

The written bytes are controlled via the sticker file too, but constrained to ARGB encoding performed in premulARGB() and getColorReplacement().

While it's probably only useful in 32bit environments, coupled with an additional ASLR-bypass gadget it might lead to remote code execution. That being said, during my research I couldn't find memory-probing oracles or remote infoleaks to overcome this protection so I didn't investigate further.

The advisories for my other issues are available at shielder.it/advisories!

Structure-aware fuzzing

While analyzing the coverage traces I noticed that most of the mutated testcases were breaking the JSON syntax or messing up the few required JSON keys, reaching very shallow code. But in those same days I learnt about structure-aware fuzzing, which looked like what I was after: since rlottie parses structured data (JSONs), i needed some way to mutate the animations without breaking its syntax; also, I wasn't much interested in fuzzing the JSON decoding because it was handled by rapidjson inside rlottie itself. While the `-x` dictionary flag in AFL++ improved the situation, it didn't instruct the fuzzer how to add or remove meaningful elements to the animation.

Let's have a little introduction on structure- / grammar-aware fuzzing for who's not familiar with it (feel free to skip this paragraph if you do!). From the structure-aware fuzzing wiki I linked earlier:

> Coverage-guided mutation-based fuzzers, such as libFuzzer or AFL, are not restricted to a single input type and do not require grammar definitions. Thus, mutation-based fuzzers are generally easier to set up and use than their generation-based counterparts. But the lack of an input grammar can also result in inefficient fuzzing for complicated input types, where any traditional mutation (e.g. bit flipping) leads to an invalid input rejected by the target API in the early stage of parsing.

As an example let's imagine we feed to AFL++ a corpus made of JSONs and point it against the harness we have seen earlier, what testcases would it produce? ... 💩 ... **Mostly broken JSONs**. :(This is because by applying "standard mutations" (e.g. bit flipping) it might mutate a char responsible for the JSON structure, breaking its syntax. This will lead to shallow code coverage, because the parser will exit once it detects the JSON is malformed, and to a lot of wasted executions, because they couldn't advance the coverage. But if we instead create a grammar definition about how are lottie animations actually structured, we'd be able to have more control about the testcase mutations. This is where protobuf and libprotobuf-mutator come in the picture: by creating a grammar definition in the protobuf syntax and using libprotobuf-mutator to instruct the fuzzer how to mutate a protobuf message, we can produce **always syntactically valid testcases** (i.e. in this case valid JSONs) to feed the target harness.

Let's see an example protobuf message I have written for the main structure by reading the source code and mattbas's python-lottie project documentation:

```
message RLottieProto {
    // required string version = 1;           // "v"
    required LayersAttribute layers = 2;      // "layers"
    optional double frame_rate = 3;           // "fr"
    required double in_point = 4;             // "ip"
    required double out_point = 5;            // "op"
    required uint32 width = 6;                // "w"
    required uint32 height = 7;               // "h"
    optional string comp_name = 8;            // "nm"
    optional bool ddd = 9;                    // "ddd"
    optional AssetsAttribute assets = 10;     // "assets"
}
```

Writing the rlottie protobuf grammar to use as an intermediate format turned out to be particularly time consuming: while the library code was easily readable, it required some tricky design decisions (proto2 or proto3? multiple types with repeated keys or minimal type + add-ons? etc...) not trivial as setting up the coverage-guided harness, leading to a **~1k LOC harness**. Moreover (probably because of that monster harness) the fuzzer was way slower than "simple" coverage-guided benchmarks (x4 slowdown on the same hardware).

To sum up, the structure-aware fuzzer turned out to be faster than the "simple" coverage-guided strategy in finding the same bugs, but required a bigger time investment upfront just to start it, so I'm happy for the knowledge I have acquired but I'd probably recommend and use it against more complex codebases than rlottie, e.g. browser's IPC. 😀

Telegram's animated stickers attack surface

So how are animated stickers implemented? They are basically files uploaded to Telegram's cloud drive and referenced in messages by setting the `application/x-tgsticker` mime type and attaching the cloud coordinates. A curious limitation I noticed is that in unencrypted chats (the default mode for chats, i.e. not "secret chats") during my testing I couldn't receive the malicious sticker to my other testing accounts; this got me wondering whether Telegram servers were doing any kind of parsing/filtering of the stickers I uploaded, but that's hard to tell since **Telegram's server-side code is not open-source** (yet?). This also limited the potential impact since only secret chats were usable to send an arbitrary animated sticker, probably because the file uploads are E2E encrypted too.

Another interesting thing I noticed about secret chats is that, besides the macOS client, it's not possible to configure the client to prevent secret chats from being automatically accepted on that device. This allowed me to automatically start a secret chat and send animated stickers to anyone via Frida (thanks @thezero for the help with the JavaScript code!), until after my reports Telegram introduced the "Filter New Chats from Non-Contacts" setting (which is still non-default so probably not enabled by everyone).

~~Un~~Fortunately the animated stickers are parsed and rendered only when the chat is opened, making these vulnerabilities reachable only if the chat is opened by clicking on it. Furthermore, since the animated sticker is downloaded on the device, everytime the chat is opened the issue triggers; this turned useless memory corruptions (such as null-pointer dereferences) into an annoyingly persistent crash which would have prevented non-technical victims from accessing the previous messages in the chat. (Tech-savvy people could have extracted them from the local Telegram's database, or used another client altogether.)

How they patched it

After my reports, Telegram introduced an interesting way to prevent such attack surface from being available remotely in a single click, without breaking the end-to-end encryption altogether: each and every animated sticker received in a secret chat (remember that malicious stickers in normal chats are filtered) are verified to be actually part of a sticker set (or "sticker pack", i.e. a collection of stickers of a specific theme/topic). This probably comes from my own proof-of-concepts where I faked sticker sets references, but at the end of the day it successfully prevents malicious stickers from being decoded on the victim device since during the creation of a sticker set every sticker is parsed (yes, I guess the issues I have found could have been used against Telegram servers themselves in the creation of a sticker pack, but again since the server-side code is not open-source that's just a guess 🤷).

We can see an example implementation of these new checks in verifyAnimatedStickerMessage, part of Telegram's Android source code:

```
TLRPC.Document document = MessageObject.getDocument(message);
String name = MessageObject.getStickerSetName(document);
if (TextUtils.isEmpty(name)) {
    return;
}
TLRPC.TL_messages_stickerSet stickerSet = stickerSetsByName.get(name);
if (stickerSet != null) {
    for (int a = 0, N = stickerSet.documents.size(); a < N; a++) {
        TLRPC.Document sticker = stickerSet.documents.get(a);
        if (sticker.id == document.id && sticker.dc_id == document.dc_id) {
            message.stickerVerified = 1;
            break;
        }
    }
    return;
}
```

`sticker.id == document.id` verifies that the unique Telegram cloud file identifier (used to reference also stickers, even in secret chats) equals the identifier of a sticker in a public sticker set, while `sticker.dc_id == document.dc_id` verifies that the datacenter identifiers match (I'm not 100% sure this was necessary). This way a potential attacker not only needs to find additional issues in the rlottie forks, but also a bypass for these new authenticity checks.

Conclusions

Before starting this research in 2019 I would have been pretty skeptical if you had asked me whether the following year I'd find a single memory corruption in Telegram. Today I shared with you the story of how I have found 13, some with a higher impact than others but all which were promptly fixed by Telegram for all the device families supporting secret chats: Android, iOS and macOS. This research helped me understand once more that **it's not trivial to limit attack surfaces at scale in end-to-end encrypted contexts** without losing functionalities. I hope that this blogpost inspired you in learning more about fuzzing and information security in general. If you have any comment or tip for improvement it would be greatly appreciated: you can reach me at @polict_ – until next time! 👋

Psst, do you feel these issues could have been found in your app? 🤓 Let's arrange a security assessment!

Previous post

Re-discovering a JWT Authentication Bypass in ServiceStack

Next post

QilingLab – Release

INFO

Shielder S.r.l.

P.I. 11435310013

REA TO - 1213132

Registered Capital: 81.000,00 €

Via Palestro, 1/C
10064 Pinerolo (TO) Italy

CONTACTS

info@shielder.com

Landline: (+39) 0121 - 39 36 42

Commercial: (+39) 345 - 30 31 983

Technical: (+39) 393 - 16 66 814

SITEMAP

Home

Company

Services

Advisories

Blog

Careers

Contacts

Disclosure policy

Privacy policy