GoPro GPMF-parser Vulnerabilities

Oct 17, 2020 • Christian Reitter
ID: CVE-2020-16158, CVE-2020-16160, CVE-2020-16161, CVE-2020-16159

As part of my fuzzing research into C parsers, I took a look at the open source GoPro GPMF-parser project. The GPMF-parser software decodes custom telemetry metadata from GoPro camera video recordings. Multimedia file parsers are notoriously difficult to write safely in C, so I expected some memory security issues and saw this as a good exercise for fuzzing.

Ultimately, this analysis led to ~25 bug reports, most of them communicated confidentially to the vendor over the course of a long disclosure process due to security implications. After coordination with the vendor, I requested four CVEs in July to track several issues with a direct impact.

Contents

Consulting

I'm a freelance Security Consultant and currently available for new projects. If you are looking for assistance to secure your projects or organization, contact me.

Technical Background

The GPMF-parser documentation gives some insight into the design of the GPMF format. Action cameras store multiple telemetry datasets within the main video container alongside the compressed video and audio, apparently for performance reasons. On the technical side, this is done via the GPMF key-length-value structure and custom extensible markers such as FourCC.

In practice, the decoding of GPMF data from existing video files involves

1. GPMF data extraction from the video container (*not core functionality, but implemented for MP4*)
2. GPMF data parsing via the GPMF-parser library (*core functionality*)

The library ships with a standalone demo program to do this processing on MP4 file inputs.

Summary

The discovered issues are presented in a condensed format due to the raw number of bugs.

Please note that the security issues are treated differently depending on their code origin. Although there are existing CVEs from 2018 and 2019 for memory issues in the MP4 parsing section (*that were discovered by other researchers, see CVE-2018-18699, CVE-2019-15148*), GoPro indicated during the disclosure process that the MP4 parsing code is not actually in-scope.

I would probably not have researched/reported the related issues in depth if this fact would have been documented more explicitly.

Issues within the example program (marked "demo") were also treated as out-of-scope and are listed here for completeness.

Product Overview

Project	Source	Fix	References
GPMF-parser	GitHub	release 2.0, release 2.0.2	?

It is not documented which other internal or external projects use the GPMF-parser library.

Main Security Issues

ID	CVE	Description	Potential Impact	Location	CWE	POC
1	CVE-2020-16158	Stack-buffer-overflow out of bounds write	Crash (with stack protection) potentially arbitrary code execution	core	CWE-787	file
17	CVE-2020-16160	Division by zero	DOS via crash (confirmed)	core	CWE-369	file
18	CVE-2020-16161	Division by zero	DOS via crash (confirmed)	core (1, 2)	CWE-369	file
21	CVE-2020-16159	Heap out of bounds read	crash (confirmed) or information disclosure	core	CWE-125	file

Potential or Out-Of-Scope Security Issues

ID	Description	Potential Impact	Location	POC
2	Stack-use-after-scope	unclear, not well defined	core	file
3	Heap-buffer-overflow out of bounds read	information disclosure	demo print	see #2
4	Large memory allocation	DOS via resource exhaustion	MP4 parser	file
5	Large memory allocation	DOS via resource exhaustion	MP4 parser	file
6	Large memory allocation	DOS via resource exhaustion	MP4 parser	-
7	Null pointer read → segfault	DOS via crash	MP4 parser	file
8	Heap out of bounds read	information disclosure	MP4 parser	file
9	Division by zero	DOS via crash	MP4 parser	file
10	Heap out of bounds read	information disclosure	MP4 parser	file
11	Heap out of bounds read	DOS via crash, information disclosure?	MP4 parser	file
12	Division by zero (*difficult to reproduce*)	potential DOS via crash	MP4 parser	-
13	Long-running loops	Partial DOS via CPU resource exhaustion	MP4 parser (1, 2, 3)	file
14	Heap out of bounds read	Potentially information disclosure	core	file
15	Heap out of bounds read	Potentially information disclosure	core	file
16	Stack out of bounds write	DOS via crash or worse	demo	file
19	Heap out of bounds read	information disclosure	core	see #18
20	Heap out of bounds read	information disclosure	core (1, 2, 3)	file

Mainly Functional Issues:

ID	Description	Potential Impact	Location
GH101	memory resource leak	resource exhaustion	MP4 parser
GH102	undefined behavior	unspecified	MP4 parser
GH103	memory resource leak	resource exhaustion	MP4 parser
GH104	memory resource leak	resource exhaustion	core

Selection of Discovered Issues

CVE-2020-16158

The `GPMF_ExpandComplexTYPE()` function allows multiple 1 byte stack out of bounds writes behind the `dst` buffer. The outer `while{}` loop has some bounds checks but manipulates the relevant counters within the loop without additional checks.

Relevant code (simplified):

```
uint32_t GPMF_ExpandComplexTYPE(char *src, uint32_t srcsize, char *dst, uint32_t *dstsize)
{
        uint32_t i = 0, k = 0, count = 0;

        while (i<srcsize && k<*dstsize)
        {
// [...], new count value is calculated here
                                uint32_t l;
                                for (l = 1; l<count; l++)
                                {
                                        dst[k] = src[i - 1];
                                        k++;
                                }
// [...]
            i++;
            k++;
        }
```

GPMF_parser.c

As you can see, the limits imposed via the `while (i<srcsize && k<*dstsize)` condition are ineffective since `k` is increased locally based on other values. The assignment `dst[k] = src[i - 1]` is therefore reachable with larger than intended `k` values.

Consider the following problematic run of `GPMF_ExpandComplexTYPE()` on a target buffer `dst` with a size of 64 bytes:

```
(gdb) print *dstsize
$7 = 64
```

Here is the memory view of the destination buffer plus following 32 bytes of memory, captured at the start of the function:
(*x86_64, no stack canary*)

```
(gdb) x/96xb dst
0x7fffffffcde0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffcde8: 0x78    0xd0    0xff    0xff    0xff    0x7f    0x00    0x00
```

```
0x7fffffffcdf0: 0x07    0x00    0x00    0x00    0xff    0x7f    0x00    0x01
0x7fffffffcdf8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffce00: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x01
0x7fffffffce08: 0x84    0xd0    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffce10: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffce18: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x01 // << end of dst
0x7fffffffce20: 0x18    0x00    0x00    0x00    0x18    0x00    0x00    0x00
0x7fffffffce28: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffce30: 0xf0    0x55    0x05    0x01    0x00    0x00    0x00    0x00
0x7fffffffce38: 0x78    0xd0    0xff    0xff    0x18    0x00    0x00    0x00
```

Here is the same memory region after exiting the `while()` loop with a problematic input:

```
(gdb) x/96xb dst
0x7fffffffcde0: 0xff    0xff    0xff    0xff    0xff    0xff    0xff    0x0f
0x7fffffffcde8: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffcdf0: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffcdf8: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffce00: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffce08: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffce10: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffce18: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f // << end of dst
0x7fffffffce20: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffce28: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffce30: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
0x7fffffffce38: 0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f    0x0f
```

In this particular case the memory is clobbered with a single `0x0f` value that is taken from `src[i - 1]`.

The classification of the security impact depends on the practical use of the GPMF-parser library and related compiler settings. On systems with stack canaries, the OOB write will be detected and the program exits, which mitigates this attack to a Denial of Service (DoS).

If the target binary is compiled without stack canaries, the vulnerability may allow an attacker to perform changes to the program flow via manipulation of adjacent memory regions behind `dst`. The changes are constrained by the nature of the out of bounds write, but a carefully crafted input might still leverage this bug to do something interesting on certain systems or architectures. In that case, the integrity and confidentiality properties of the target program will be impacted as well, although the overall attack complexity might be high or some user interaction might be involved.

CVE-2020-16160

The `GPMF_Decompress()` function includes a problematic division that can lead to a division by zero issue if `type` is not detected. This is possible since `GPMF_SizeofType()` can return zero.

```
uint32_t sizeoftype = GPMF_SizeofType(type);
uint32_t chn = 0, channels = sample_size / sizeoftype;
```

GPMF_parser.c

The impact is a denial of service through the crash, represented by the following sanitizer warning:

```
==20022==ERROR: AddressSanitizer: FPE on unknown address
```

CVE-2020-16161

The `GPMF_ScaledData()` function includes a problematic modulo operation that can lead to division by zero issue if `inputtypeelements` is zero:

```
switch (complextype[i % inputtypeelements])
```

GPMF_parser.c

The same issue is also present at a second location in the code.

The impact is a denial of service through the crash, represented by the following sanitizer warning:

```
==20022==ERROR: AddressSanitizer: FPE on unknown address
```

CVE-2020-16159

`GPMF_ScaledData()` calls macros to read and convert input data into specific types.

This can lead to a heap out of bounds read and following Segmentation fault in the MACRO_BSWAP_CAST_UNSIGNED_SCALE handling when operating on the end of `ms->buffer`:

```
==14571==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x606000000060 at pc 0x0000005750ce bp 0x7fff70e5b6d0 sp 0x7fff70e5b6c8
READ of size 1 at 0x606000000060 thread T0
    #0 0x5750cd in GPMF_ScaledData /gpmf-parser/demo/../GPMF_parser.c:1694:37
    #1 0x4c9a53 in main /gpmf-parser/demo/GPMF_demo.c:253:7
    #2 0x7f346c0e0bba in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x26bba)
    #3 0x41e2d9 in _start (/gpmf-parser/demo/gpmfdemo+0x41e2d9)

0x606000000060 is located 0 bytes to the right of 64-byte region
[0x606000000020,0x606000000060)

==14571==ERROR: AddressSanitizer: SEGV on unknown address 0x606000010000
(pc 0x0000005750da bp 0x7fff70e75ab0 sp 0x7fff70e5b6e0 T0)
==14571==The signal is caused by a READ memory access.
    #0 0x5750da in GPMF_ScaledData /gpmf-parser/demo/../GPMF_parser.c:1694:37
```

The main observed impact of this is a denial of service via the segfault.

Depending on the usage of the GPMF-parser library, it might be possible to avoid the segfault and leverage this for some sort of information disclosure via uninitialized heap memory.

POC

Crafted example inputs for the individual issues were provided to the vendor and are linked in the bug summary overview at the beginning of the article.

To reproduce:

- Compile an older version of the GPMF-parser demo application
  - Optionally add appropriate sanitizers such as `-fsanitize=address` and other compile flags for error detection
- Run the example program against the `poc_issue*.mp4` file
- Use special compile-time and run-time flags such as `-fsanitize-recover=adddress`, `ASAN_OPTIONS=halt_on_error=0` to proceed beyond the initial errors on inputs that trigger multiple issues

Disclosure

The disclosure process for these bugs was long and exhausting.

One of the reasons for this is my decision to fuzz the GPMF parsing code “all the way” through the MP4 file parsing interface. While this allowed me to present MP4-based POCs that directly worked as input for the unmodified demo application (better reproducibility for the developers), it also doubled the number of bugs that I had to analyze, report and patch away. Additionally, many bugs happened on similar inputs, which increased the complexity of preparing useful reports due to overlapping crashes.

Another major factor is the perceived unfamiliarity of the vendor with the requested type of coordinated disclosure for their open source code. For example, there is insufficient public documentation regarding disclosure contacts, guidelines and processes. A lot of back and forth via email was required to coordinate the relevant aspects, and this was not always successful.

The main patch release (2.0) was released around a month after the disclosure to the vendor. While this is not a bad response time for a likely nonessential vendor project, 9 of the 21 confidentially reported issues were missed during patching. The vendor quickly followed up with another release (2.0.2) a few days after I reported this.

To my knowledge, no public credit has been given for the 21 vulnerability reports despite a number of code changes and two security releases, which is disappointing.

Partial Timeline

Note: this timeline excludes confidential data points and related information.

Date	information
2020-06-20	Initial communication to developer address and request for security contact
2020-06-23	Request to 2nd developer address
2020-06-25	GitHub issue to ask for official security contact
2020-06-25	Initial email communication with GoPro security
2020-07-10 to 2020-07-14	Public GitHub bug issues GH101 to GH104
2020-07-11 to 2020-07-15	Confidential coordinated disclosure of issues #1 to #21 with POCs
2020-07-15	Shared fuzzing corpus with GoPro for testing
2020-07-23	GoPro agrees to proposed CVE request process
2020-07-27	GoPro acknowledges proposed CVE descriptions
2020-07-28	Request of CVEs from MITRE
2020-07-30	MITRE assigns CVEs
2020-08-20	GPMF-parser 2.0.0 is released with security fixes
2020-08-22	Note to GoPro about incomplete fixes for issues #3, #4, #7, #8, #15, #16, #19, #20 and #21
2020-08-27	GPMF-parser 2.0.2 is released with security fixes
2020-08-28	GoPro requests deadline extension to 6th October (accepted)
2020-10-06	Disclosure deadline
2020-10-17	Publication of this blog article

Christian Reitter

Information security and other interests.