Starred by 6 users

Owner:	mjurczyk@google.com
CC:	proje...@google.com
Status:	Fixed *(Closed)*
Components:	----
Modified:	Aug 12, 2020

Product-Android
Vendor-Samsung
Deadline-90
Deadline-Grace
Deadline-Exceeded
Severity-Critical
CCProjectZeroMembers
Finder-mjurczyk
Methodology-Fuzzing
Reported-2020-Jan-28
CVE-2020-8899
Fixed-2020-May-6

Issue 2002: Samsung Android multiple interactionless RCEs and other remote access issues in Qmage image codec built into Skia

Reported by mjurczyk@google.com on Tue, Jan 28, 2020, 5:26 AM EST Project Member

----------=====[1. Background

The Android OS internally uses the Skia library [1] to load and display images in the system through various standard APIs such as BitmapFactory [2] or BitmapRegionDecoder [3]. Skia is therefore used to handle both built-in graphic resources embedded in APKs, as well as other pictures displayed during normal use of many applications, including files coming from untrusted sources (e.g. sent via MMS, chat apps, or e-mails; downloaded in a web browser and then displayed in the Gallery or My Files app, etc.). For instance, in my testing, the default Samsung Messages app processes the contents of incoming MMS messages without any user interaction, and I expect that other similar attack vectors exist. Given its exposure and the fact that it is written in C++, Skia and its image-related components constitute remotely accessible interactionless attack surface on Android, potentially prone to memory safety issues. The relevant code is found in the libskia.so or libhwui.so system libraries.

By default, the following image formats are supported by Skia on Android:
* BMP
* WBMP
* ICO
* PNG
* JPEG
* GIF
* WEBP
* HEIF
* RAW (also known as DNG, uses the TIFF file structure)

The above list can be enumerated by inspecting the implementation of the SkCodec::MakeFromStream method found in skia/src/codec/SkCodec.cpp [4] (or SkCodec::NewFromStream on earlier versions of Android). However, if we look into the libskia.so / libhwui.so library pulled from any Samsung mobile phone or tablet released in the last five years in IDA Pro, we will discover that Samsung added support for a few other custom formats. Finding the method in question in IDA is possible thanks to the symbols being shipped with the module. The extra formats are:

* QM (version 1)
* QG (versions 1.0, 1.1 and 2.0)
* ASTC
* PIO (a variant of PNG)

According to my analysis, custom image formats started being introduced by Samsung around Android version 4.4.4 in the second half of 2014. In addition to the ones listed above, historically one could also find support for image formats like IM, IFEG, IT, QW, PFR`, QIO, MIF, ETC1, ETC2, PVR, SPI; however I don't believe they are still found on phones eligible for receiving security updates.

----------=====[2. The Qmage format in Skia

In this report, I am focusing specifically on the QM and QG formats. Considering that they are subsequent versions of the same format developed by the same vendor, I will refer to them collectively as "Qmage" or "QMG" and assume that they have a .qmg file extension. The Qmage image format was designed by a Korean third-party company named Quramsoft (formerly with a homepage at http://www.quramsoft.com/), which now operates as Fingram (https://fingram.com/). The company offers a wide range of image and video processing solutions, Qmage being one of them [5]. The breakdown of the different Qmage format versions is as follows:

* QM (version 1), magic "QM\x01", first seen in Android 4.4.4 (2014)
* QG (version 1.0), magic "QG\x01\x00", first seen in Android 5.0 (2015)
* QG (version 1.1), magic "QG\x01\x01", first seen in Android 6.0 (2016)

* QG (version 2.0), magic "QG\x02\x00", first seen in Android 10.0, January 2020 patch for Galaxy Note 10+.

As you can see, the most recent version 2.0 was introduced just this month, which shows that the codebase is under active development. Example .qmg files for testing can be found in the embedded resources of built-in APKs in some (but not all) Samsung firmwares. I suspect that support for Qmage files might have been added specifically to handle such app/theme resources only (thus possibly reducing the disk space consumption / improving UI responsiveness); in such case, a potential solution to the issues described here could be to lock the interface down to trusted inputs only, and prevent any external/user-controlled .qmg's from being displayed by Skia.

An interesting observation about the Qmage codec is that each time a new version of the format was introduced in Samsung's Android in the past, a completely new fork of the entire codebase was added, while appending suffixes to the function names in the previous most recent copy. And so, the current build of libhwui.so has four different clones of the same Qmage-related functions, e.g.:

```
--- cut ---
QuramQmageDecVersionCheck
QuramQmageDecVersionCheck_Rev11454_141008
QuramQmageDecVersionCheck_Rev14474_20150224
QuramQmageDecVersionCheck_Rev8253_140615
QuramQmageDecodeFrame
QuramQmageDecodeFrame_Rev11454_141008
QuramQmageDecodeFrame_Rev14474_20150224
QuramQmageDecodeFrame_Rev8253_140615
QuramQmageDecodeRegion
QuramQmageDecodeRegion_Rev11454_141008
QuramQmageDecodeRegion_Rev14474_20150224
QuramQmageDecodeRegion_Rev8253_140615
--- cut ---
```

The suffixes clearly include revision numbers and some dates, perhaps the cut-off dates of the specific forks. Now, all three versions (1.0, 1.1, 2.0) of the QG format are correctly loaded in every app that I have checked, but the QM images seem to fail to load in some contexts, perhaps due to not having been used or tested in the last five years. For example, the Messages app won't display QM images, the Gmail app will display thumbnails but not the full images, and the Gallery app won't display thumbnails but will show the pictures when you click on them. So in terms of attack surface, bugs in the handling of the three QG iterations are the most accessible, while successful exploitation of issues in the Rev8253_140615-family functions depends on the specific app through which the payload is to be delivered.

The complexity of the Qmage codec is very high -- QMG files may choose from a wide range of different custom compression schemes, each of them handled by a lengthy and obscure decompression routine. There are dozens of functions with over 4 kB in length in the library, with the single longest function (QuramQumageDecoder32bit24bit) being 40 kB (!) long. This translates to tens of thousands lines of C code that likely have never been subject to much scrutiny in the form of a security audit or fuzz testing. I conclude this based on the fact that the code seems to be lacking any kind of bounds checking at any point of the file parsing, and it crashes instantly with almost every trivial modification to a valid testcase (e.g. when the dimensions of the image are slightly increased).

----------=====[3. Fuzzing the Qmage codec with a custom loader

As a first step to improve the security posture of the library, I decided to fuzz test it myself using libraries from an up-to-date Samsung Galaxy Note 10+ device (Android 10, build QP1A.190711.020.N975FXXS1BSLD, January 2020 patch level). To that end, I performed the following steps:
* Created an initial corpus of various QM, QG 1.0 and QG 1.1 images extracted from the resources of built-in Samsung APKs. I was unable to locate any existing, well-formatted QG 2.0 files, but I managed to synthesize them on the fly during the course of my coverage-based fuzzing.
* Developed a testing harness program -- a loader for QMG files that uses the same Skia APIs that are used by standard Android interfaces such as BitmapFactory. The loader attempts to mimic the interactions with Skia and the overall mobile phone environment as closely as possible. It is a Linux ELF executable compiled for ARM64 using Android NDK, and is linked directly with the tested libhwui.so library. It can be run on a x86 system through qemu-aarch64, or natively on a Samsung device.
* Ran several weeks of fuzz testing (more than 15 billion iterations) using a modified build of the QEMU emulator, which logs code coverage information.

My loader also has two special features:
* The default libc allocator is switched to AFL's libdislocator [6] via __malloc_hook etc. Libdislocator is a special simplified allocator which places each new allocation directly before the end of a memory page, such that most out-of-bounds accesses to dynamic allocations result in immediate SIGSEGV crashes, instead of silently allowing for OOB read/write accesses that may only manifest later in unrelated areas of the library. This facilitates much more precise bug detection and also reliable crash deduplication based on the stack trace.
* The program registers its own custom signal handler, which prints out a verbose AddressSanitizer-compatible report if an unexpected signal (indicating a crash) is encountered. The report includes the type of exception, a symbolized call stack, disassembly of the instructions triggering the signal and CPU register values.

----------=====[3.1. Building the loader

The source code of the harness is attached to this report. In order to build it on Linux, you will need:
* Android NDK, needed for the cross-compiler
* Skia source code, needed for the headers (the linking is done against libhwui.so)
* Capstone [7], to disassemble the crashing instructions
* Libbacktrace source code, needed for the headers (the linking is done against libbacktrace.so)
* The complete /system/lib64 directory and the /system/bin/linker64 file from the tested Android system

Once we download all of the above dependencies and set the correct paths at the top of the build.sh script, we should be able to compile the program as an ARM64 ELF file:

```
--- cut ---
$ file ./loader
loader: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /tmp/linker64, with debug_info, not stripped
$
--- cut ---
```

Then, we can fix up the paths in run.sh and test that everything works correctly:

```
--- cut ---
$ ./run.sh
Usage: loader <input image file> [output .raw file]
$
--- cut ---
```

The above process worked fine with libraries from Android 9, which I tested initially. However, when trying to start the harness against /lib64 from Android 10, I was getting the following error:

```
--- cut ---
==31162==Sanitizer CHECK failed: /usr/local/google/buildbot/src/android/llvm-toolchain/toolchain/compiler-rt/lib/sanitizer_common/sanitizer_posix.cc:371
((internal_prctl(0x53564d41, 0, addr, size, (uptr)name) == 0)) != (0) (0, 0)
libc: Fatal signal 6 (SIGABRT), code -1 (SI_QUEUE) in tid 31162 (qemu-aarch64), pid 31162 (qemu-aarch64)
libc: failed to spawn debuggerd dispatch thread: Invalid argument
--- cut ---
```

The problem was caused by a new version of /lib64/libclang_rt.ubsan_standalone-aarch64-android.so, which fails to run outside of Android, expecting the prctl syscall to always succeed. To fix the issue, I swapped this specific file out for its older build from Android 9.

When we run the loader against a valid QMG file, we should see output similar to the following:

```
--- cut ---
[+] Detected image characteristics:
[+] Dimensions:      1440 x 1440
[+] Color type:      4
[+] Alpha type:      1
[+] Bytes per pixel: 4
[+] codec->GetAndroidPixels() completed successfully
--- cut ---
```

----------=====[3.2. Example crash report

When we pass one of the malformed test cases, we will observe the following ASAN-like report being printed out:

```
--- cut ---
[+] Detected image characteristics:
[+] Dimensions:      10 x 9994
[+] Color type:      4
[+] Alpha type:      1
[+] Bytes per pixel: 4
ASAN:SIGSEGV
=================================================================
==115737==ERROR: AddressSanitizer: SEGV on unknown address 0x4088aee000 (pc 0x4005b104ec sp 0x4000cff1b0 bp 0x4000cff1b0 T0)
    #0 0x0007f440 in libc.so (memset+0xac)
    #1 0x002ddbbc in libhwui.so (QmageRunLengthDecodeCheckBuffer_Rev11454_141008+0x550)
    #2 0x002ddfa8 in libhwui.so (PVcodecDecoderGrayScale_Rev11454_141008+0x35c)
    #3 0x002d3fec in libhwui.so (__QM_WCodec_decode_Rev11454_141008+0x118)
    #4 0x002d3d2c in libhwui.so (Qmage_WDecodeFrame_Low_Rev11454_141008+0xc4)
    #5 0x002d08e8 in libhwui.so (QuramQmageDecodeFrame_Rev11454_141008+0x94)
    #6 0x006e1f90 in libhwui.so (SkQmgCodec::onGetPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const&, int*)+0x4f0)
    #7 0x004daf00 in libhwui.so (SkCodec::getPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const*)+0x358)
    #8 0x006e278c in libhwui.so (SkQmgAdapterCodec::onGetAndroidPixels(SkImageInfo const&, void*, unsigned long, SkAndroidCodec::AndroidOptions const&)+0xac)
    #9 0x004da498 in libhwui.so (SkAndroidCodec::getAndroidPixels(SkImageInfo const&, void*, unsigned long, SkAndroidCodec::AndroidOptions const*)+0x2b0)
    #10 0x0004a650 in loader (ProcessQmage(char*, char*)+0x674)
    #11 0x0004b42c in loader (main+0xa4)
    #12 0x0007e858 in libc.so (__libc_init+0x70)

==115737==DISASSEMBLY
    0x4005b104ec:       stp             q0, q0, [x3, #-0x20]
    0x4005b104f0:       subs            x2, x2, #0x40
    0x4005b104f4:       b.hi            #0x4005b104e8
    0x4005b104f8:       stp             q0, q0, [x4, #-0x40]
    0x4005b104fc:       stp             q0, q0, [x4, #-0x20]
    0x4005b10500:       ret
    0x4005b10504:       nop
    0x4005b10508:       mrs             x5, dczid_el0
    0x4005b1050c:       tbnz            w5, #4, #0x4005b104dc
    0x4005b10510:       and             w5, w5, #0xf

==115737==CONTEXT
   x0=0000004088a5781d  x1=00000000000000ff  x2=000000000005997c  x3=0000004088aee020
   x4=0000004088b4799c  x5=0000000000000020  x6=fefeff3f875f32b7  x7=7f7f7f7fff7f7fff
   x8=cfe4ddb5a4111839  x9=cfe4ddb5a4111839 x10=000000000000000f x11=0000000000000001
  x12=00000040076b7860 x13=0000000000000001 x14=0000000000010000 x15=0000000000000020
  x16=0000004007d83fc8 x17=0000004005b10440 x18=00000040013d6000 x19=0000000000018664
  x20=00000040889d9008 x21=0000004088a57600 x22=00000040888f7188 x23=00000040888f0800
  x24=00000000000000ff x25=000000000000f017f x26=0000004008279020 x27=0000000000000001
  x28=000000000000021d  FP=0000004000cff210  LR=0000004007874bbc  SP=0000004000cff1b0

==115737==ABORTING
--- cut ---
```

If we send the same sample to another phone via MMS, the default Samsung Messages app will crash and a similar report will be generated, which can be then retrieved with logcat:

```
--- cut ---
Fatal signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x769daee000 in tid 4200 (droid.messaging), pid 4200 (droid.messaging)
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'samsung/d2sxx/d2s:10/QP1A.190711.020/N975FXXS1BSLD:user/release-keys'
Revision: '24'
ABI: 'arm64'
Timestamp: 2020-01-22 16:41:48+0100
pid: 4200, tid: 4200, name: droid.messaging  >>> com.samsung.android.messaging <<<
uid: 10082
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x769daee000
   x0  000000769da14a9d  x1  00000000000000ff  x2  0000000000016bfc  x3  000000769daee020
   x4  000000769db04c1c  x5  8080800000000000  x6  fefeff76cd43ef1f  x7  7f7f7f7fff7fff7f
   x8  4c900663de41e395  x9  4c900663de41e395  x10 000000000000000f  x11 0000000000000001
   x12 00000077cada1860  x13 0000000000000001  x14 0000000000010000  x15 0000000000000020
   x16 00000077cb46dfc8  x17 00000077cd10b440  x18 00000077ce93c000  x19 0000000000018664
   x20 000000769d975608  x21 000000769da14880  x22 00000076a8e8e6d0  x23 000000769d7ac000
   x24 00000000000000ff  x25 000000000000f017f  x26 00000077ce44f020  x27 0000000000000001
   x28 000000000000021d  x29 0000007fcfe0e940
   sp  0000007fcfe0e8e0  lr  00000077caf5ebbc  pc  00000077cd10b4ec

backtrace:
     #00 pc 000000000007f4ec  /apex/com.android.runtime/lib64/bionic/libc.so (memset+172) (BuildId: 220051b49364b1c2da3adf10c30832cc)
     #01 pc 00000000002ddbb8  /system/lib64/libhwui.so (QmageRunLengthDecodeCheckBuffer_Rev11454_141008+1356) (BuildId: fcab350692b134df9e8756643e9b06a0)
     #02 pc 00000000002ddfa4  /system/lib64/libhwui.so (PVcodecDecoderGrayScale_Rev11454_141008+856) (BuildId: fcab350692b134df9e8756643e9b06a0)
     #03 pc 00000000002d3fe8  /system/lib64/libhwui.so (__QM_WCodec_decode_Rev11454_141008+276) (BuildId: fcab350692b134df9e8756643e9b06a0)
     #04 pc 00000000002d3d28  /system/lib64/libhwui.so (Qmage_WDecodeFrame_Low_Rev11454_141008+192) (BuildId: fcab350692b134df9e8756643e9b06a0)
     #05 pc 00000000002d08e4  /system/lib64/libhwui.so (QuramQmageDecodeFrame_Rev11454_141008+144) (BuildId: fcab350692b134df9e8756643e9b06a0)
     #06 pc 00000000006e1f8c  /system/lib64/libhwui.so (SkQmgCodec::onGetPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const&, int*)+1260)
(BuildId: fcab350692b134df9e8756643e9b06a0)
     #07 pc 00000000004daefc  /system/lib64/libhwui.so (SkCodec::getPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const*)+852) (BuildId:
fcab350692b134df9e8756643e9b06a0)
     #08 pc 00000000006e2788  /system/lib64/libhwui.so (SkQmgAdapterCodec::onGetAndroidPixels(SkImageInfo const&, void*, unsigned long,
SkAndroidCodec::AndroidOptions const&)+168) (BuildId: fcab350692b134df9e8756643e9b06a0)
     #09 pc 00000000004da494  /system/lib64/libhwui.so (SkAndroidCodec::getAndroidPixels(SkImageInfo const&, void*, unsigned long, SkAndroidCodec::AndroidOptions
const*)+684) (BuildId: fcab350692b134df9e8756643e9b06a0)
     #10 pc 0000000000187b24  /system/lib64/libandroid_runtime.so (doDecode(_JNIEnv*, std::__1::unique_ptr<SkStreamRewindable,
std::__1::default_delete<SkStreamRewindable>>, _jobject*, _jobject*, long, long)+2572) (BuildId: 21b5827e07da22480245498fa91e171d)
     #11 pc 0000000000186c8c  /system/lib64/libandroid_runtime.so (nativeDecodeStream(_JNIEnv*, _jobject*, _jobject*, _jbyteArray*, _jobject*, _jobject*, long, long)+156)
(BuildId: 21b5827e07da22480245498fa91e171d)
     #12 pc 00000000002bbff0  /system/framework/arm64/boot-framework.oat (art_jni_trampoline+272) (BuildId: 4997a729336be54facda67f1651ee19ed61fd840)
     #13 pc 0000000000474854  /system/framework/arm64/boot-framework.oat (android.graphics.BitmapFactory.decodeStream+388) (BuildId:
4997a729336be54facda67f1651ee19ed61fd840)
[...]
--- cut ---
```

----------=====[3.3. Libdislocator vs libc malloc

As we can see, the two reports are nearly identical, with the one from our loader being slightly more descriptive, as it also shows the faulting assembly instruction, which can used to determine the type of the invalid memory access (read or write). However, please note that in most cases the sample .qmg files provided with this report don't crash in the exact same way in default Android apps; the reason being that these apps use the standard system allocator and not libdislocator. As a result, when loading such images on a real device you may observe a variety of crashes, some of them seemingly not even related to Qmage, or in some cases there may be no crashes at all. This doesn't mean that the bugs are not there, just that they are masked by adjacent allocations on the heap, and the overreads or overwrites don't happen to corrupt any critical data. It is therefore crucial that the loader is used to reproduce and analyze the issues, and to verify any potential fixes.

The three significant differences between libdislocator and the default malloc are:
* Libdislocator enforces a global limit of 1 GB of heap memory allocated by the client application. Any malloc() or realloc() call that would grow the total size of all allocation over 1 GB will return NULL. This means that memory is artificially capped in the test environment and will run out much faster than on a real device. If the Qmage codec doesn't check the return value of malloc/realloc, this may lead to various NULL pointer dereferences which don't reproduce without libdislocator. However, I believe that it is possible to arrange equivalent memory pressure conditions on a physical device (e.g. by constructing an image to first trigger several multi-gigabyte regions to exhaust the available physical memory), and thus any such crashes related to the 1 GB limit manifest real problems in the code that should be addressed.
* Libdislocator doesn't align allocations to any specific boundary, and doesn't return buffers any larger than requested. For example, in our loader malloc(17) will return a region that is exactly 17 bytes long, followed by an unmapped page (i.e. the return address will be of the form of 0x?????????????FEF). On the other hand, libc would round up such an allocation to 32 bytes, nullifying any potential consequences of accessing 15 bytes beyond the requested buffer later in the code. A special corner case is also malloc(0), for which libc returns an area of 8 usable bytes, whereas libdislocator returns an unmapped address, so that any dereference to such a pointer leads to a crash. In other words, libdislocator enforces much more strict memory safety rules and will catch even the smallest overreads/overwrites. A subset of the crashes may be difficult to reproduce on a Samsung device due to the more relaxed environment on the phone.
* Libdislocator fills every new allocated memory region with a 0xCC marker byte, to catch potential use-of-uninitialized-memory bugs and increase determinism of the run time environment. The libc malloc reuses previously freed blocks of memory and provides no guarantees as to the contents of the returned regions.

----------=====[4. Fuzzing results and the deduplication process

As a result of my fuzzing, I identified 5218 unique crashes. Crash deduplication was performed based on the three top-level function addresses in the stack trace; i.e. two crashes with the same three top-level callstack entries are considered to manifest the same problem. For example, a tuple of the following three items is the unique signature of the crash showcased above:

```
--- cut ---
    #0 0x0007f440 in libc.so (memset+0xac)
    #1 0x002ddbbc in libhwui.so (QmageRunLengthDecodeCheckBuffer_Rev11454_141008+0x550)
    #2 0x002ddfa8 in libhwui.so (PVcodecDecoderGrayScale_Rev11454_141008+0x35c)
--- cut ---
```

The only special case when considering unique crashes are standard libc memory functions memset() and memcpy(). In their case, we use the base address of these routines for deduplication, and not the address of the specific instruction where the crash happens. This is to avoid classifying several crashes as unique ones if they only differ by the specific location inside memset() or memcpy(). If we detected a crash with the same #1 and #2 stack trace entries as above, but triggered at "memset+0xb4" instead of "memset+0xac", it would still be considered a duplicate.

----------=====[4.1. Crash classification and breakdown by type

Thanks to the fact that the reports contain instruction disassembly and some context around the unhandled exception, it was possible to determine the general cause of the crashes and classify them based on the type of error and probable severity. I have divided the crashes into the following 11 groups:
* write -- caused by store (mnemonics starting with "st") instructions attempting to write to invalid memory locations, which indicates some kind of memory corruption (buffer overflow, use-after-free etc.). These are generally the most severe vulnerabilities which may be directly exploited for arbitrary code execution, and should be treated with the highest priority.
* sigabrt -- caused by an unhandled SIGABRT signal triggered either directly by the codec with an explicit abort() call, by the new[] function upon a failed allocation, or by __stack_chk_fail when stack corruption is detected. The first and second cases are not high severity, but any stack-based buffer overflows are serious issues that should be amongst the first ones to be fixed.
* read-{1, 2, 4, 8, 16, 32} -- caused by load (mnemonics starting with "ld") instructions attempting to read from invalid memory locations, which may indicate a continuous out-of-bounds read, a non-continuous out-of-bounds read (e.g. when an attacker-controlled array index is used), a use-after-free, or a read from a completely wild pointer (e.g. due to use of uninitialized memory or some other condition). By analyzing the name of the crashing instruction and its operands, I was able to work out the size of the invalid read and use it for fine-grained classification. And so for example an invalid read of a single byte falls into read-1, an invalid read of an "int" type is a read-4, and so on.
* read-vector -- for "vector" ARM64 load instructions such as ld1, ld2, ld3, ld1r etc. causing the crash, I didn't implement the full logic to detect the memory access size and instead put the crashes in this special read-vector bucket.
* read-memcpy -- this is a category for crashes caused by invalid reads inside the memcpy() function, which implies either an invalid "src" pointer, an inadequate value of the "size" argument, or both.
* null-deref -- NULL pointer dereference crashes caused by any type of memory operation (read or write) attempted over an address close to 0x0. They may be caused e.g. by unchecked return values of failed malloc() calls or uninitialized pointers, and usually manifest low-severity, unexploitable bugs.

The breakdown of the crashes by category is shown below:

	Count	Percentage
write	174	3.33%
sigabrt	3	0.06%
read-1	3322	63.66%
read-2	393	7.53%
read-4	703	13.47%
read-8	34	0.65%
read-16	52	1.00%
read-32	3	0.06%
read-vector	18	0.34%
read-memcpy	124	2.38%
null-deref	392	7.51%

As a general guideline, we would advise prioritizing the bugs in the following order, from most to least important:
1. write, sigabrt (stack corruption)
2. read-8, read-16, read-32, read-vector, read-memcpy
3. read-1, read-2, read-4
4. sigabrt (memory exhaustion), null-deref

However, please note that it is not a hard rule that "read" crashes are less severe than "write", as the crash classification only represents the first memory safety violation that the test case triggered, and it is possible that in some cases, an out-of-bounds read is followed by an out-of-bounds write and memory corruption occurs. We would therefore highly recommend that all of the reported issues are eventually fixed. If possible, it would also be a good idea to compile the codec with the real AddressSanitizer instrumentation to reproduce the crashes and verify fixes, as this is the more correct and reliable way of detecting memory errors compared to using libdislocator and a custom signal handler.

----------=====[4.2. Crash archive structure

A 7z archive containing the complete set of the ASAN-like reports and up to three proof-of-concept .qmg files is attached (7-Zip was chosen for best compression). The directory structure is as follows:

```
crashes
├── read
│   ├── 1
│   │   ├── 000269676c8cd89bd7e8618be019dcb2
│   │   │   ├── poc
│   │   │   │   ├── signal_sigsegv_4007531f6c_7955_a76a36ca5ed8d40f01505f4adb574113.qmg
│   │   │   │   ├── signal_sigsegv_400783cf6c_7786_9053708dc9f3e789cd14820fe96a4dc0.qmg
│   │   │   │   └── signal_sigsegv_4007b9ff6c_1056_0a528e9d327ede3f68a31f3d5fe7f7e7.qmg
│   │   │   └── report.txt
...
└── write
    ├── 01a016e8e04274825f971848bc35c2f4
    │   ├── poc
    │   │   ├── signal_sigsegv_4003f4fca8_6549_e9bf68c239eb55c8654336e2f9f25111.qmg
    │   │   ├── signal_sigsegv_4004568ca8_8689_164428f06429b701870761821ead19b7.qmg
    │   │   └── signal_sigsegv_400607cca8_4052_e303a245acdbc1430dab2501535047f8.qmg
    │   └── report.txt
...
```

There are four top-level directories corresponding to the general crash types, followed by an optional sub-type for "read" crashes, followed by the unique crash ID, which then contains report.txt and a directory with proof of concept samples. Even though the test cases have a .qmg extension, it may be useful to rename them to .jpg or another common image format to get them to open as image files on Android, as .qmg's doesn't have any associated applications by default and do not trigger the generation of thumbnails.

----------=====[5. Note on exploitability

Finally, let me briefly demonstrate the potential exploitability of the heap-based buffer overflows found in the Qmage codec. The Android heap contains a variety of unprotected function pointers, which may be corrupted to redirect the execution flow to an address of the attacker's choosing. For example, below is a crash log from trying to browse to a directory containing a malformed QMG file (disguised as a .jpg) in the built-in My Files app on Samsung Galaxy Note 10+. The program crashed while trying to execute code from an invalid address 0x4a4a4a4a4a4a4a4a:

```
--- cut ---
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'samsung/d2sxx/d2s:10/QP1A.190711.020/N975FXXS1BSLD:user/release-keys'
Revision: '24'
ABI: 'arm64'
Timestamp: 2020-01-24 09:40:57+0100
pid: 31355, tid: 31386, name: thumbnail_threa  >>> com.sec.android.app.myfiles <<<
uid: 10088
signal 7 (SIGBUS), code 1 (BUS_ADRALN), fault addr 0x4a4a4a4a4a4a4a4a
    x0  0000006ff55dc408  x1  0000006f968eb324  x2  0000000000000001  x3  0000000000000001
    x4  4a4a4a4a4a4a4a4a  x5  0000006f968eb31d  x6  00000000000000b3  x7  00000000000000b3
    x8  0000000000000000  x9  0000000000000001  x10 0000000000000001  x11 0000000000000001
    x12 0000007090d96860  x13 0000000000000001  x14 0000000000000004  x15 0000000000000002
    x16 0000007091463000  x17 0000007090ea2d94  x18 0000006f95d1a000  x19 0000006ff5709800
    x20 00000000ffffffff  x21 0000006ff55dc408  x22 00000000000000b0  x23 0000006f968ed020
    x24 0000000000000001  x25 0000000000000001  x26 0000006f968ed020  x27 0000000000000be5
    x28 0000000000012e9a  x29 0000006f968eb370
    sp  0000006f968eb310  lr  0000007090f5f7f0  pc  004a4a4a4a4a4a4a

backtrace:
      #00 pc 004a4a4a4a4a4a4a  <unknown>
      #01 pc 00000000002e97ec  /system/lib64/libhwui.so (process_run_dec_check_buffer+92) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #02 pc 00000000002ddb94  /system/lib64/libhwui.so (QmageRunLengthDecodeCheckBuffer_Rev11454_141008+1320) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #03 pc 00000000002d45dc  /system/lib64/libhwui.so (PVcodecDecoderIndex_Rev11454_141008+1264) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #04 pc 00000000002d3fb4  /system/lib64/libhwui.so (__QM_WCodec_decode_Rev11454_141008+224) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #05 pc 00000000002d3d28  /system/lib64/libhwui.so (Qmage_WDecodeFrame_Low_Rev11454_141008+192) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #06 pc 00000000002d08e4  /system/lib64/libhwui.so (QuramQmageDecodeFrame_Rev11454_141008+144) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #07 pc 00000000006e1f8c  /system/lib64/libhwui.so (SkQmgCodec::onGetPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const&, int*)+1260)
(BuildId: fcab350692b134df9e8756643e9b06a0)
      #08 pc 00000000004daefc  /system/lib64/libhwui.so (SkCodec::getPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const*)+852) (BuildId:
fcab350692b134df9e8756643e9b06a0)
      #09 pc 00000000006e2788  /system/lib64/libhwui.so (SkQmgAdapterCodec::onGetAndroidPixels(SkImageInfo const&, void*, unsigned long,
SkAndroidCodec::AndroidOptions const&)+168) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #10 pc 00000000004da494  /system/lib64/libhwui.so (SkAndroidCodec::getAndroidPixels(SkImageInfo const&, void*, unsigned long, SkAndroidCodec::AndroidOptions
const*)+684) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #11 pc 00000000006e0930  /system/lib64/libhwui.so (SkBitmapRegionCodec::decodeRegion(SkBitmap*, SkBRDAllocator*, SkIRect const&, int, SkColorType, bool,
sk_sp<SkColorSpace>)+1168) (BuildId: fcab350692b134df9e8756643e9b06a0)
      #12 pc 00000000001991c8  /system/lib64/libandroid_runtime.so (nativeDecodeRegion(_JNIEnv*, _jobject*, long, int, int, int, int, _jobject*, long, long)+976) (BuildId:
21b5827e07da22480245498fa91e171d)
[...]
--- cut ---
```

----------=====[6. Affected devices and disclosure deadline

The devices and firmwares I have tested include:
1. Galaxy Note 4 (Android 4.4.4, Sep 2014)
2. Galaxy Note Edge (Android 4.4.4, Nov 2014 - Dec 2014)
3. Galaxy Note Edge (Android 5.0.1, Mar 2015 - Jun 2015)
4. Galaxy Core Prime (Android 5.1.1, Aug 2015)
5. Galaxy Note 5 (Android 5.1.1, Aug 2015)
6. Galaxy Note 4 (Android 5.1.1, Oct 2015)
7. Galaxy Note 3 (Android 5.0, Jan 2016)
8. Galaxy S7 (Android 6.0.1, Feb 2016)
9. Galaxy Note 5 (Android 6.0.1, Feb 2016)
10. Galaxy S7 (Android 7, Jan 2017)
11. Galaxy Note 5 (Android 7, Mar 2017)
12. Galaxy S8 (Android 7, Apr 2017)
13. Galaxy S8 (Android 8, Feb 2018)
14. Galaxy S7 (Android 8, Apr 2018)
15. Galaxy S9 (Android 8, Apr 2018)
16. Galaxy S8 (Android 9, Feb 2019)
17. Galaxy S9 (Android 9, Apr 2019)
18. Galaxy A50 (Android 9, Oct 2019)
19. Galaxy Note 10+ (Android 9, Nov 2019 - Dec 2019)
20. Galaxy Note 10+ (Android 10, Jan 2020)

According to these tests and based on the Qmage versions listed above, I understand that all Samsung Android devices released since late 2014 / early 2015 up to today's flagships are affected by some or all of the Qmage-related bugs. The specific subset of vulnerabilities depends on which iterations of the QM/QG formats are supported by the system in question, with the latest phones and tablets being affected by the largest number of issues, as they support all four major versions of Qmage.

In my view, there are two valid approaches to addressing the problem as a whole:
* If the .qmg format is designed to be only used for built-in APK resources, access to the special codec could be restricted to such trusted inputs only, without exposing it to user-controlled images sent over the network or found on the file system.
* If the previous option is infeasible, I would recommend that all of the reported issues are fixed and the code is brought to a "fuzz-clean" state, so that no more memory safety bugs can be trivially found by fuzzing. I advise against selectively fixing only the most serious looking crashes, as I believe many of the innocent-looking "read" violations may also conceal memory corruption bugs, which may become easy targets for offensive security researchers who invest the time to analyze the crashes in more detail.

In case of any questions or concerns regarding the report, do not hesitate to contact me.

This bug is subject to a 90 day disclosure deadline. After 90 days elapse, the bug report will become visible to the public. The scheduled disclosure date is 2020-04-27.

References:
[1] https://skia.org/
[2] https://developer.android.com/reference/android/graphics/BitmapFactory
[3] https://developer.android.com/reference/android/graphics/BitmapRegionDecoder
[4] https://github.com/google/skia/blob/android/q-release/src/codec/SkCodec.cpp
[5] http://fingram.com/qmage/
[6] https://github.com/google/AFL/tree/master/libdislocator
[7] http://www.capstone-engine.org/

crashes.7z
7.5 MB Download

Comment 1 by mjurczyk@google.com on Wed, Jan 29, 2020, 9:50 AM EST Project Member

A small update:

I discovered that a few crashes which were initially classified as "null_deref" may in fact be caused by reads or writes from wild, non-canonical addresses. The reason for that is when such a pointer is accessed, the signal handler doesn't receive its actual value but 0x0 in the siginfo_t.si_addr field instead.

An example of such a crash is 0b8f78b2f3255e1b7cc0d3d7c6165867, with the following report:

```
--- cut ---
ASAN:SIGSEGV
=================================================================
==392281==ERROR: AddressSanitizer: SEGV on unknown address 0x0 (pc 0x400304549c sp 0x4000cfecc0 bp 0x4000cfecc0 T0)
    #0 0x002e449c in libhwui.so (QuramQumageDecoder32bit24bit_Rev11454_141008+0x1aec)
    #1 0x002d4024 in libhwui.so (__QM_WCodec_decode_Rev11454_141008+0x150)
    #2 0x002d3d2c in libhwui.so (Qmage_WDecodeFrame_Low_Rev11454_141008+0xc4)
    #3 0x002d08e8 in libhwui.so (QuramQmageDecodeFrame_Rev11454_141008+0x94)
    #4 0x006e1f90 in libhwui.so (SkQmgCodec::onGetPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const&, int*)+0x4f0)
    #5 0x004daf00 in libhwui.so (SkCodec::getPixels(SkImageInfo const&, void*, unsigned long, SkCodec::Options const*)+0x358)
    #6 0x006e278c in libhwui.so (SkQmgAdapterCodec::onGetAndroidPixels(SkImageInfo const&, void*, unsigned long, SkAndroidCodec::AndroidOptions const&)+0xac)
    #7 0x004da498 in libhwui.so (SkAndroidCodec::getAndroidPixels(SkImageInfo const&, void*, unsigned long, SkAndroidCodec::AndroidOptions const*)+0x2b0)
    #8 0x0004a650 in loader (ProcessQmage(char*, char*)+0x674)
    #9 0x0004b42c in loader (main+0xa4)
    #10 0x0007e858 in libc.so (__libc_init+0x70)

==392281==DISASSEMBLY
    0x400304549c: ldp   x0, x16, [x8, #0x18]
    0x40030454a0: orr   w8, wzr, #0x18
    0x40030454a4: ldr   w16, [x16, w1, uxtw #2]
    0x40030454a8: nop
    0x40030454ac: umaddl    x0, w1, w8, x0
    0x40030454b0: movn    w8, #0
    0x40030454b4: mov   x1, x4
    0x40030454b8: sub   w17, w17, w16
    0x40030454bc: lsl   w16, w8, w17
    0x40030454c0: bic   w24, w24, w16

==392281==CONTEXT
   x0=0000000000000010  x1=0000000000000000  x2=0000000000000000  x3=0000004089773e5c
   x4=0000004089793ffc  x5=0000000000000000  x6=000000000000003d  x7=0000000000000003
   x8=f89caf65daba379e  x9=0000000000000007 x10=0000000000000000 x11=00000000000000c4
  x12=0000000000665566 x13=0000000000000009 x14=0000000000000000 x15=0000004089779ffe
  x16=0000000000000008 x17=0000000000000010 x18=0000004001020000 x19=0000000000000000
  x20=000000408977b9d0 x21=0000004089795ffe x22=00000000aaccaacc x23=0000000000000000
  x24=0000000000000000 x25=0000000000000000 x26=0000000000000000 x27=0000000000000000
  x28=0000004089791ffb  FP=0000004000cfefd0  LR=0000004089768b68  SP=0000004000cfecc0

==392281==ABORTING
--- cut ---
```

As we can see, the header states "SEGV on unknown address 0x0", but the actual value of the x8 register being dereferenced is 0xf89caf65daba379e. The severity of a read from a wild non-zero pointer is almost certainly higher than a read from an address close to 0x0.

A few other examples of such misclassified crashes are listed below:
* 1119fe7fb1d1a1f9a1c844ad83fa95b0
* 17d48258f33b1bac04a84b90bbfd7cbf
* 1a864556720716ab7d5f6c023629d953
* 2255b089179e1bc528d3898ae18d4cd7
* 3b930915ed43f9f0077913af16f087b2
* 50cfdf3e1928fb09575633950b710006
* 69dfed3feb65327fb9ea432beb4e2673
* 74af42e80009dee7d0fd25adbaa6ddf5
* a0ef94a8911faf0a700414f8bfa1a254
* b42ea88e7c4d3a7859afe92fd0903a94
* b69538d1bc03d90596adfb64e3489a3f
* bffb5ac0cf9ef020902eff7fcfae0185
* c5fcf29d0ba0116e25a73a100da1043c
* d912ebc184ac3dce8dc032bd4b1c55b6
* deb6e3f5ca8b139b5008dbf6a38438be
* e24d0e04379d383b2cc8c3705190b316
* f8b5a196136c7f8f8a9d951988da0a9a

We recommend giving these bugs more priority than other legitimate NULL pointer dereferences, and verifying if there are more such crashes in the "null_deref" category (or preferably fixing all of them regardless of the root cause).

Comment 2 by mjurczyk@google.com on Wed, Feb 19, 2020, 10:07 AM EST Project Member

Another update:

In this bug tracker entry (GPZ #2002), I reported crashes surfaced by fuzzing the Qmage image codec found on Android 10 in January 2020. However, I have since performed some manual analysis of the equivalent code on Android 9 (Samsung Galaxy A50, February 2020 patch level), and noticed significant differences in Quram's low-level allocator wrappers, which may generate additional vulnerabilities on devices running Android version <= 9. Due to the overall bad security posture of the code base and sheer amount of previously reported bugs, I consider this class of problems a variant/continuation of the original report, and I'm thus addressing them in the same thread.

Let's start with the QuramQmage_Malloc function used throughout the QMG codec. On Android 9, it decompiles to:

--- cut ---

```
void *QuramQmage_Malloc(int size)
{
  return malloc((int)((size + 31) & 0xFFFFFFE0));
}
--- cut ---
```

whereas on Android 10, it can be represented as:

```
--- cut ---
void *QuramQmage_Malloc(int size)
{
  return malloc(size);
}
--- cut ---
```

In both cases, there is the problem of down-casting the malloc() argument from the original "size_t" type (64-bit on 64-bit platforms) to a 32-bit "int". Furthermore, in the Android 9 code, there is the additional alignment of the size to 32 bytes, which may result in an integer overflow and allocating 0 bytes (in practice -- 8 bytes) for arguments in the range of [0xFFFFFFE1 .. 0xFFFFFFFF]. Considering that there are numerous places in the code where a 32-bit integer is read from the input stream and directly passed to malloc(), this may lead to a variety of heap-based buffer overflow conditions.

Furthermore, let's compare QuramQmage_Calloc on Android 9:

```
--- cut ---
void *QuramQmage_Calloc(int num, int size)
{
  size_t final_size;
  void *ptr;

  final_size = (int)((num * size + 31) & 0xFFFFFFE0);
  ptr = malloc(final_size);
  if ( ptr )
    memset(ptr, 0, final_size);
  return ptr;
}
--- cut ---
```

... with Android 10:

```
--- cut ---
void *QuramQmage_Calloc(unsigned int num, unsigned int size)
{
  return calloc(num, size);
}
--- cut ---
```

Again in both cases, we have the downcasting of the "num" and "size" arguments from size_t to unsigned int. However, in the earlier version of the function, there are several other problems:

- The multiplication of num * size is performed on 32-bit types and is prone to an integer overflow.
- The alignment of the total size to 32 bytes is also prone to an integer overflow.
- Implementing the function as a pair of malloc() + memset() means that for very large allocations (e.g. several GB, which is easy to trigger in the image codec), the entire requested memory region will be explicitly touched and thus attempted to be mapped to physical memory, which may lead to memory exhaustion and system instability. On the other hand, the libc calloc() routine is smarter in that it returns a memory area that is not fully mapped to RAM before each page is accessed for the first time.

All of the above problems could be mitigated by simply using the standard calloc() function instead of the custom wrapper.

The same issues as discussed above are also present in QuramQmage_ReAlloc:

```
--- cut ---
void *QuramQmage_ReAlloc(void *ptr, int size)
{
  return realloc(ptr, (size + 31) & 0xFFFFFFE0);
}
--- cut ---
```

versus Android 10:

```
--- cut ---
void *QuramQmage_ReAlloc(void *ptr, unsigned int size)
{
  return realloc(ptr, size);
}
--- cut ---
```

Finally, it is worth noting that the size_t -> int argument downcasting also happens in the QuramQmage_Memcpy, QuramQmage_Memmove and QuramQmage_Memset helper functions, but the security impact of this behavior is probably lower than the bugs in the allocator wrappers.

Comment 3 by mjurczyk@google.com on Wed, Mar 18, 2020, 5:25 AM EDT Project Member
Labels: Deadline-Exceeded Deadline-Grace

Samsung Mobile Security has requested a deadline grace extension until May 8.

Comment 4 by mjurczyk@google.com on Wed, May 6, 2020, 12:09 PM EDT Project Member
Status: Fixed (was: New)
Labels: -Restrict-View-Commit CVE-2020-8899 Fixed-2020-May-6

Samsung have rolled out the May 2020 update, including fixes for these Qmage bugs (https://security.samsungmobile.com/securityUpdate.smsb).

The fuzzing harness referenced in this tracker entry was open-sourced on GitHub: https://github.com/googleprojectzero/SkCodecFuzzer.

An update on exploitation: after reporting the crashes, I spent several weeks working on a 0-click MMS exploit proof-of-concept for one of the vulnerabilities. I managed to achieve this goal with a Samsung Galaxy Note 10+ phone running Android 10. A video demonstrating the successful exploitation can be found under the following link: https://www.youtube.com/watch?v=nke8Z3G4jnc.

Over the course of the next month or two, I will be writing the story of finding this codec, and how the memory corruption primitives it provided were used to first break the ASLR mitigation on a remote device, and then execute arbitrary code on the phone. This will be taking place in the form of blog posts published on the Google Project Zero blog (https://googleprojectzero.blogspot.com/). In the meantime, I encourage all owners of Samsung phones to install updates ASAP.

While I write my blogpost, I've compiled below a short FAQ section which aims to add some more context to the video and shed some light on the exploitability of the issues.

===== Likely FAQ

Q: Does this affect all Android devices?

A: No. These are vulnerabilities in a specific piece of software that ships with Samsung Android devices. As mentioned in the original post above, I understand that all Samsung Android phones that have shipped since late 2014 / early 2015 are affected. There is also a list of phones and operating system versions that I have tested (see "6. Affected devices and disclosure deadline" in the original post above).

===

Q: Did you release the proof of concept exploit today?

A: No. We have released the video to provide proof of this issue in lieu of a proof-of-concept exploit.

===

Q: What is the specific configuration exploited in the demo?

A: The victim phone was a Samsung Galaxy Note 10+ running Android 10 (February 2020 update) with the default Samsung Messages app set as the SMS/MMS handler. It is my understanding that firmwares up to and including April 2020 are affected, but the February 2020 patch is what I started working on and what I stuck with for this research.

My exploit uses a vulnerability in a new QG 2.0 format and takes advantage of Android 10-specific behavior. The existing primitives would need to be adjusted for older Qmage formats and/or some low-level OS implementation details. That said, to my knowledge there aren't any fundamental barriers to attacking earlier versions of the system and I anticipate that a similar attack would work on earlier versions of Samsung Android devices.

I haven't tested the exploitation through other messaging apps, though I would expect that if they use the same bitmap classes without whitelisting specific formats, they would also be vulnerable to similar attacks.

===

Q: How does the exploit work?

A: The presented exploit spends the first ~100 minutes determining the layout of the address space on the victim phone, to find the base addresses of two libraries required for RCE. Once this information is leaked, a final MMS is sent to obtain remote access to the device via a command prompt (a "reverse shell").

===

Q: What is the run time of this exploit? Does it always take 1h45m to execute?

A: The run time and overall reliability of my exploit is hugely dependent on a number of factors: the randomized memory layout, strength of GSM signal, the existing history of messages in the phone, or even whether Wi-Fi is enabled or not.

A majority of the attack is spent defeating ASLR, which is achieved by continuously sending "probe" MMS' which leak whether a specific address range is mapped or not. By taking advantage of several weaknesses of the ASLR implementation in Android, I managed to reduce the number of necessary probes to a relatively small number: 86 in the case of the public demo.

Each probe causes a crash of the Messages app, and Android enforces a 60-second cooldown between subsequent crashes of the same program for it to be allowed to be restarted again. So the lower bound of attack run time is the number of necessary probes expressed in minutes. Taking some extra overhead into consideration, the 86-probe attack ran in ~100 minutes as expected.

Depending on the specific memory layout randomized by the system, I have estimated the minimum number of probes to be ~50. The maximum is hard to determine, but in an extremely unfavorable case it could be as much as 300. On average, the exploit needs around 100 probes to complete.

===

Q: Why does the video show 86 probes but 122 received messages?

A: For every oracle query which returns "true", we send another message to trigger a crash in the Messages app. Therefore the number of MMS received by the victim is typically close to 1.5x of the number of address probes.

===

Q: What privileges does the attacker gain in the system after a successful attack?

A: The vulnerable codec executes in the context of the attacked app processing input images, so the attacker also gets the privileges of that app. In the case of my demo, that's Samsung Messages, which has access to a variety of personal user information: call logs, contacts, microphone, storage, SMS etc.

While not explicitly tested, I also strongly suspect that local privilege escalation may be possible with the help of these bugs. For example, the highly privileged System UI process may display arbitrary images supplied by other apps in notifications, and I have observed it crash in Qmage-related code a number of times in my experimentation.

===

Q: Have you tested any attack vectors other than MMS?

A: I haven't devised any end-to-end attacks similar to that via MMS, but as noted in the original bug report, all apps in the system which display untrusted images with the standard Bitmap interfaces are affected by these issues. For example, I have confirmed that the Qmage file which is used as the final payload to get a reverse shell via MMS, also gives the attacker remote access when it is copied to the device's file system and opened with the Gallery app.

===

Q: Are there any mitigations available to users against this and similar attacks, other than updating regularly?

A: For Samsung devices, these issues are fixed in the May 2020 patch. Generally speaking of image codecs, I am not aware of any generic mitigations against these types of bugs. One easy way to mitigate against attackers using exploits delivered specifically through MMS is to disable the "auto retrieve" option for multimedia messages in the Messages app.

===

Q: The video in the issue tracker shows lots of messages arriving - surely a user would notice that?

A: The video shows a proof of concept exploit and makes no attempt to be silent or stealthy. However after some brief experimentation, I have found ways to get MMS messages fully processed without triggering a notification sound on Android, so fully stealth attacks might be possible.

===

Q: Why did you choose Samsung phones?

A: The team had a hackathon as part of a team bonding activity in late 2019. Samsung phones were chosen as they were the most popular in Europe at the time (most of Project Zero is in Europe) and our team focuses on end user security.

Comment 5 by mjurczyk@google.com on Wed, Aug 12, 2020, 1:16 PM EDT Project Member

Source code of the proof-of-concept Samsung MMS exploit.

mms_exploit.zip
16.2 KB Download