Exploit Analysis of The WhatsApp Double-Free Vulnerability (CVE-2019-11932) Using the GEF-GDB Debugger

By Kai Lu(@k3vinlusec)

Over the past several weeks, I have been conducting security research on WhatsApp, a popular instant messaging application. To start research on a new application, previously disclosed vulnerabilities often help provide some inspiration for finding any possibly vulnerable modules. The vulnerability CVE-2019-11932 is a double-free bug that exists in libpl_droidsonroids_gif.so in WhatsApp for Android.

This vulnerability was found by researcher Awakened, who published an exploit PoC of this bug. In this blog, I will demonstrate how this exploit works by dynamically debugging it with GEF-GDB, showing how to use its double-free bug to execute arbitrary code.

[0x01 Description]

The following is the vulnerability description from the vendor, Facebook, which was taken from https://www.facebook.com/security/advisories/cve-2019-11932.

"A double free vulnerability in the DDGifSlurp function in decoding.c in libpl_droidsonroids_gif before 1.2.15, as used in WhatsApp for Android before 2.19.244, allows remote attackers to execute arbitrary code or cause a denial of service."

[0x02 Debugging Environment]

The debugging environment is shown below.
WhatsApp Messenger Version: 2.19.230
Android Device: Google Pixel phone, android 8.1.0
Debugger: GDB 8.3(aarch64-linux-android-gdb) and GDBServer 8.3.
OS: macOS Catalina 10.15.2 (19C57)

I also used the GEF-GDB plugin to make debugging more effective and fun.

Note: Due to an issue in the latest NDK-GDB version 21.0.6113669, I cannot use the python extension in GDB. So I compiled the GDB 8.3's source code from Google by myself. It supports aarch64-linux-android target and python3.

[0x03 Analysis]

The following is the crafted PoC GIF file.

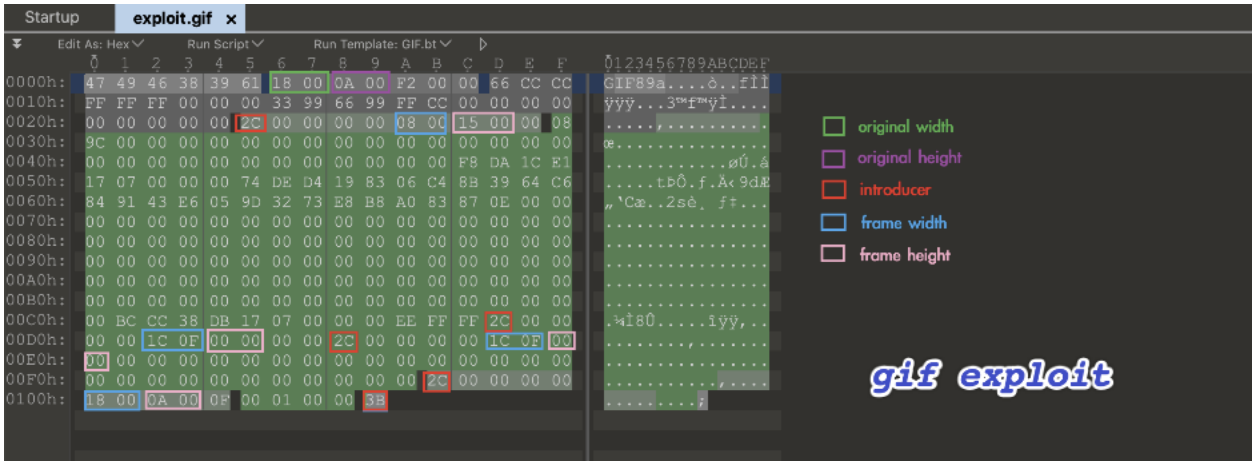

Figure 1. The crafted PoC GIF file

The vulnerable code is located in the function DDGifSlurp(GifInfo *info, bool decode, bool exitAfterFrame), found in decoding.c in the library libpl_droidsonroids_gif.so.

```
58              if (decode) {
59                  int_fast32_t widthOverflow = gifFilePtr->Image.Width - info->originalWidth;
60                  int_fast32_t heightOverflow = gifFilePtr->Image.Height - info->originalHeight;
61                  const uint_fast32_t newRasterSize = gifFilePtr->Image.Width * gifFilePtr->Image.Height;
62                  if (newRasterSize > info->rasterSize || widthOverflow > 0 || heightOverflow > 0) {
63                      void *tmpRasterBits = reallocarray(info->rasterBits, newRasterSize, sizeof(GifPixelType));
64                      if (tmpRasterBits == NULL) {
65                          gifFilePtr->Error = D_GIF_ERR_NOT_ENOUGH_MEM;
66                          break;
67                      }
68                      info->rasterBits = tmpRasterBits;
69                      info->rasterSize = newRasterSize;
70                  }
```

vulnerable code

Figure 2. The vulnerable code

The function reallocarray(void *optr, size_t nmemb, size_t size) is used for memory reallocation.

As shown in Figure 1, the original width of this GIF is 0x18, and the original height is 0x0A. This crafted GIF includes four total frames. When WhatsApp parses the GIF file, it attempts to decode these four frames. The vulnerable code shown in Figure 2 could then be executed and the function reallocarray() could be invoked three times. The

following table explains the status of memory and the condition of invoking reallocarray() function during the process of decoding each frame.

Frame NO.	Width	Height	Condition	reallocarray	Memory
Frame 1	0x08	0x15	heightOverflow > 0	info->rasterBits=reallocarray(NULL ,0x08*0x15, 0x01)	Allocate memory A of size 0xA8
Frame 2	0xF1C	0x0	widthOverflow > 0	reallocarray(info->rasterBits, 0, 0x01)	Freed the allocated memory A
Frame 3	0xF1C	0x0	widthOverflow >0	reallocarray(info->rasterBits, 0, 0x01)	Freed the allocated memory A
Frame 4	0x18	0x0A	N/A	N/A	N/A

Table 1. The status of memory and the condition of invoking reallocarray() function

We can see here that it can allocate memory the size of 0xA8 in bytes when decoding the first frame. This leads to the first free memory when decoding the second frame. The same thing happens when decoding the third frame. Finally, this leads to a double-free vulnerability.

By reverse engineering and then analyzing the source code of libpl_droidsonroids_gif, the two functions – Java_pl_droidsonroids_gif_GifInfoHandle_openFile() and Java_pl_droidsonroids_gif_GifInfoHandle_renderFrame() – could be invoked when WhatsApp parses and renders a GIF file. The first one is used to open a GIF file, then create and initialize a GifInfo structure. The second one is used to render the frames in a GIF file.

When a victim receives the malicious GIF file sent by an attacker via tapping “Document”, this vulnerability will be triggered after you tap “Gallery”. During the entire process, the GIF file could be parsed twice, meaning that the functions Java_pl_droidsonroids_gif_GifInfoHandle_openFile() and Java_pl_droidsonroids_gif_GifInfoHandle_renderFrame() could be invoked twice, respectively.

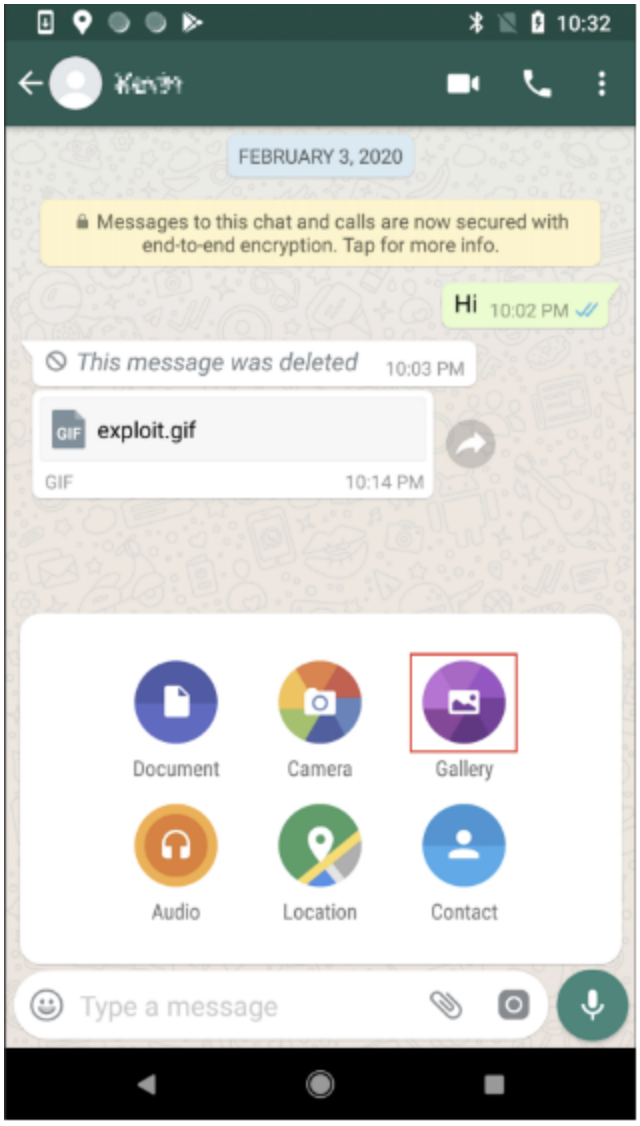

Figure 3. Triggering the vulnerability

It's worth noting that in Android, a double-free vulnerability of a memory of size M can lead to two subsequent memory allocations of size M, both returning the same memory address.

Next, let's start debugging and exploiting this vulnerability using GEF-GDB. As shown in Table 1, this is the first time the crafted PoC GIF has been parsed. During the second parsing of the vulnerable GIF file, we will exploit this bug and complete a code execution.

Next, I will show you how to complete a code execution at the second time of parsing of the vulnerable GIF file in WhatsApp.

In the function Java_pl_droidsonroids_gif_GifInfoHandle_openFile(), it calls malloc() to allocate memory for the GifInfo structure. The following is used to create the GifInfo structure of size 0xA8 in bytes.

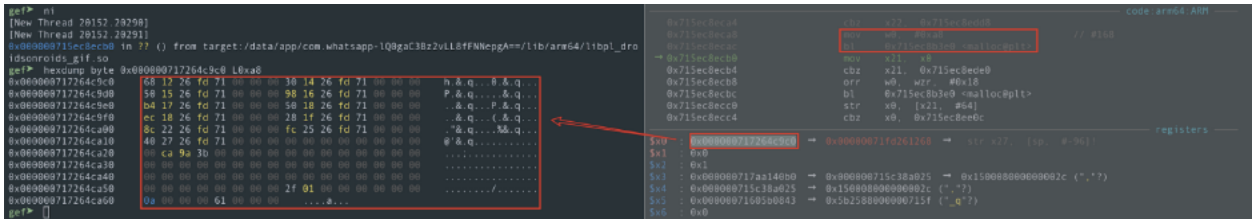

Figure 4. Allocate memory for the GifInfo structure

Let's look at the definition of the GifInfo structure, which refers to https://github.com/koral--/android-gif-drawable/blob/v1.2.17/android-gif-drawable/src/main/c/gif.h. The data at offset 0x80 is a function pointer.

Figure 5. The GifInfo structure

It can then invoke reallocarray() in the function Java_pl_droidsonroids_gif_GifInfoHandle_renderFrame(), shown in Figure 6. It could then allocate a new memory of size 0xA8. The size of the newly allocated memory is the same as the GifInfo structure's size shown in Figure 4.

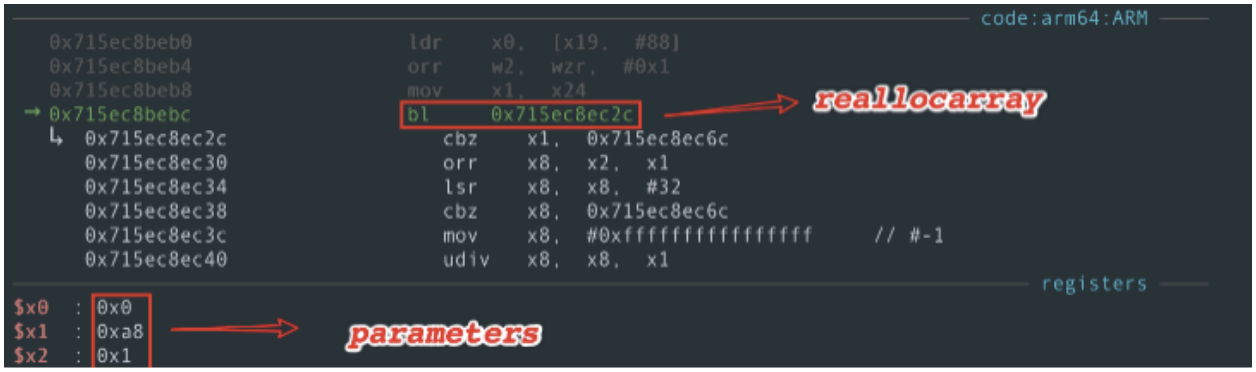

Figure 6. Call reallocarray()

Figure 7. The newly allocated memory by reallocarray()

Due to the nature of double-free vulnerabilities in Android, seen in Figure 7, the memory allocated by reallocarray() also points to the start address of the GifInfo structure that was allocated earlier.

Next, it continues to call DgifGetLine() to decompress the first frame in the GIF file. The following is the comparison of calling DgifGetLine() before and after.

Figure 8. Before and After comparison of calling DgifGetLine()

The decompressed data overwrites the data in the memory allocated by reallocarray() in Figure 7. At the end of the function DDGifSlurp() in decoding.c, it can finally invoke info->rewindFunction(info). We can control its address and set it to point to the ROP gadget in order to complete a code execution. At this point, the register x8 has pointed to the ROP gadget in libhwui.so.

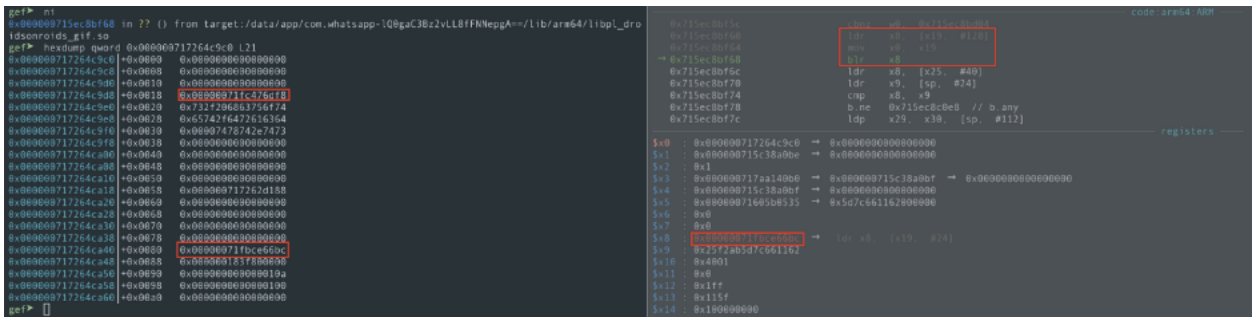

Figure 9. Invoke info->rewindFunction(info) controlled by us

As shown in Figure 9, it will then jump to the ROP gadget in libhwui.so and execute it. And as shown in Figure 10, the register x8 points to the system() API, while the register x0 points to the string of command. It can then invoke the API system (“touch /sdcard/test.txt”) to create a file in the folder /sdcard. Pwn! It finally completes a code execution.

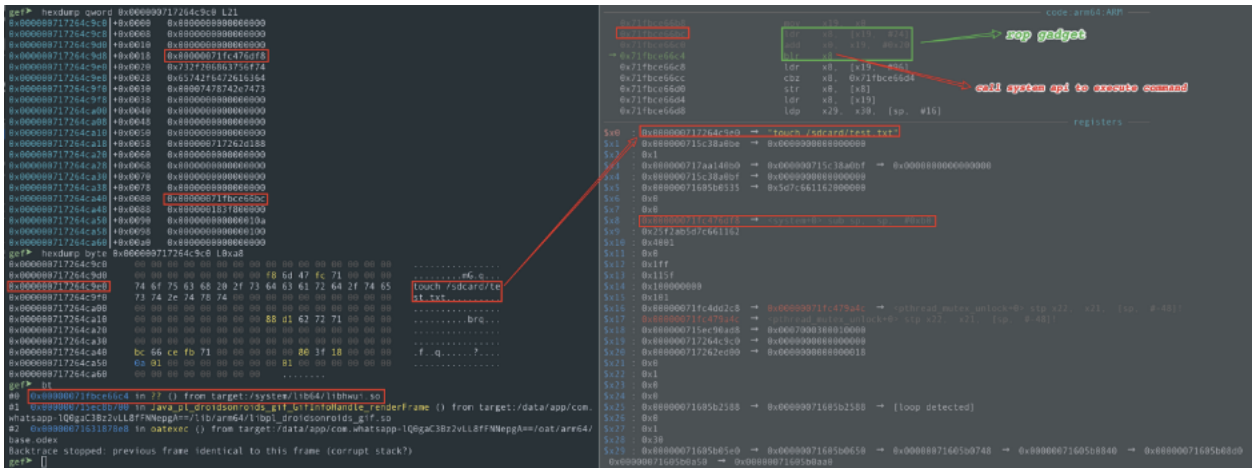

Figure 10. The code execution by exploiting this vulnerability

[0x04 Tricks and Tips]

There are some tricks and tips to be noted when dynamically debugging this vulnerability in WhatsApp. During the process of debugging, the debugger may receive some real-time events from WhatsApp, such as the following.

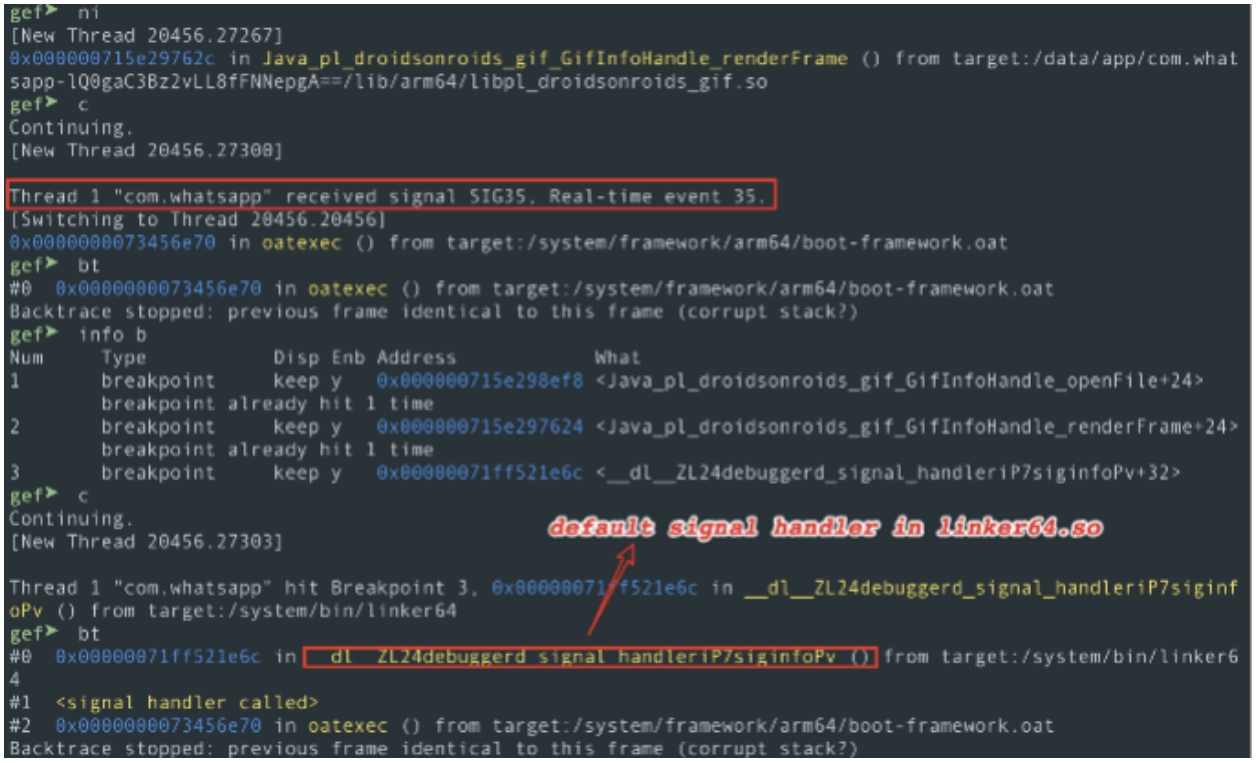

Figure 11. GDB receiving the signal SIG35

The thread “com.whatsapp” receives the signal SIG35 (there may be other real-time events), and then the default signal handler **debuggerd_signal_handler**(**int** signal_number, siginfo_t* info, **void*** context) in linker64.so in Android can receive this signal. Finally, it can execute a new program, /system/bin/crash_dump64, and also cause the debugging session to disconnect. So you cannot continue to debug this app in GDB. Fortunately, you can use the following command to handle this trick in GDB.

handle SIG33 SIG34 SIG35 noprint nostop ignore

By using the option “ignore”, GDB does not allow your program to see this signal. So the default signal handler **debuggerd_signal_handler** won’t be invoked, and it won’t create the new process /system/bin/crash_dump64. You can modify the real-time event number depending on your circumstance.

[0x05 Conclusion]

In this blog, I provided an analysis of the WhatsApp double-free vulnerability (CVE-2019-11932) using GEF-GDB, and showed how it can be exploited to execute code. Great thanks to the discoverer of this vulnerability, Awakened. I also shared

some tricks and tips in the process of debugging using GEF-GDB **and I will be using these techniques in my own bug hunting in the app.** Have fun hunting for bugs in WhatsApp!

[0x06 Reference]

https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/
https://github.com/koral--/android-gif-drawable/tree/v1.2.17/android-gif-drawable
https://gef.readthedocs.io/en/master/
https://sourceware.org/gdb/onlinedocs/gdb/Signals.html
https://www.roe.ac.uk/~ert/stacpolly/idb_manual/common/idb_the_info_handle_and_handle_commands.htm
http://giflib.sourceforge.net/whatsinagif/bits_and_bytes.html
http://giflib.sourceforge.net/whatsinagif/lzw_image_data.html
http://giflib.sourceforge.net/whatsinagif/animation_and_transparency.html