Talos Vulnerability Report

TALOS-2022-1485

HDF5 Group libhdf5 gif2h5 out-of-bounds write vulnerability

AUGUST 16, 2022

CVE NUMBER

CVE-2022-25972

SUMMARY

An out-of-bounds write vulnerability exists in the gif2h5 functionality of HDF5 Group libhdf5 1.10.4. A specially-crafted GIF file can lead to code execution. An attacker can provide a malicious file to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

HDF5 Group libhdf5 1.10.4

PRODUCT URLS

libhdf5 - https://www.hdfgroup.org

CVSSV3 SCORE

7.8 - CVSS:3.0/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

CWE

CWE-787 - Out-of-bounds Write

DETAILS

HDF5 is a file format that is maintained by a non-profit organization, the HDF Group. HDF5 is designed to store and organize of large amounts of scientific data. It is used to exchange data structures between applications in industries (such as the GIS industry) via libraries such as GDAL, OGR or as part of software like ArcGIS.

The library that includes the `gif2h5` tool is used for converting GIF data to the HDF5 file format. The vulnerability exists due to a failure to check the bounds of a heap buffer during GIF decompression while using user-provided input to calculate an offset for writing into the heap buffer.

During GIF file decompression, the code size which represents the number of bits required to represent pixel values is used to calculate various offsets and other codes, such as the clear code and end-of-file code. In the file `decompress.c`, we can see this occur:

```
/*
  Example using a code size of 0x0c:
  CodeSize = 0x0c
  ClearCode = 0x1000
  EOFCode = 0x1001
  FreeCode & FirstFree = 0x1002
  */

200     CodeSize  = GifImageDesc->CodeSize;   // Get CodeSize
201     ClearCode = (1 << CodeSize);          // Shift by CodeSize value
202     EOFCode   = ClearCode + 1;            // Add 1
203     FreeCode = FirstFree = ClearCode + 2; // Add 2
```

After this information is gathered, the remaining GIF data is parsed in a loop until the `EOFCode` is reached. Note that there are two objects on the heap, `Prefix` & `Suffix`, that were allocated earlier in the code:

```
159     if (!(Prefix = calloc(4096, sizeof(int)))) {
160         printf("Out of memory");
161         exit(EXIT_FAILURE);
162     }
163     if (!(Suffix = calloc(4096, sizeof(int)))) {
164         printf("Out of memory");
165         exit(EXIT_FAILURE);
```

At this point, we can see the size and address of both heap objects:

```
gef➤  hexdump Prefix-2
0x0000555555a99e08     11 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.@..............
0x0000555555a99e18     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a99e28     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a99e38     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................

gef➤  hexdump Suffix-2
0x0000555555a9de18     11 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.@..............
0x0000555555a9de28     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a9de38     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a9de48     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
```

Once the data parsing begins, the user-controlled index `FreeCode` is used to write into the heap buffers. Since our user-controlled `FreeCode` variable is `0x1002` it will index to `0x4008` when line `300` is executed. We can see this will lead to a heap-buffer overflow that will allow us to overwrite the size of the `Suffix` heap buffer, for example:

```
gef➤  hexdump &Prefix[FreeCode]
0x0000555555a9de18     11 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.@..............
0x0000555555a9de28     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a9de38     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a9de48     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................


232     /*
233      * Decompress the file, continuing until you see the GIF EOF code.  One
234      * obvious enhancement is to add checking for corrupt files here.
235      */
236
237     Code = ReadCode();
238
239     while (Code != EOFCode) {
...
296             /*
297              * Build the hash table on-the-fly. No table is stored in the
298              * file.
299              */
300             Prefix[FreeCode] = OldCode;
301             Suffix[FreeCode] = FinChar;
302             OldCode          = InCode;
```

After line `300` is executed, we can see that we have cleared the size of the `Suffix` buffer and can continue to corrupt the remaining heap data:

```
gef➤  hexdump Suffix-2
0x0000555555a9de18     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a9de28     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a9de38     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
0x0000555555a9de48     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
```

To demonstrate triggering a crash, we will use the following values for the decompression data:

```
AA AA AA 04 40 47
```

These values will be read by the function `ReadCode()` called at line `237` (and later at `320`), until certain codes are found. In this case, we want the `EOFCode` or `0x1001` to trigger the crash.

```
232     /*
233      * Decompress the file, continuing until you see the GIF EOF code.  One
234      * obvious enhancement is to add checking for corrupt files here.
235      */
236
237     Code = ReadCode();
238
239     while (Code != EOFCode) {            // Loops until EOFCode is reached
...
296             /*
297              * Build the hash table on-the-fly. No table is stored in the
298              * file.
299              */
300             Prefix[FreeCode] = OldCode;  // Out-of-bounds write to corrupt
Suffix
301             Suffix[FreeCode] = FinChar;
302             OldCode          = InCode;
...
318         }
319
320         Code = ReadCode();              // Call ReadCode() during each loop
iteration
321     }
```

The ReadCode() function perfoms a few operations against the Raster data. When the values `0x474004` are read in this function (in this case, the third time this function is called), they will evaluate to `0x1001` or the `EOFCode`

```
77 static int
78 ReadCode(void)
79 {
80     int RawCode, ByteOffset;
81
82     ByteOffset = BitOffset / 8;                                          //
BitOffset = 0x1a, ByteOffset = 0x03
83     RawCode    = Raster[ByteOffset] + (0x100 * Raster[ByteOffset + 1]);  //
RawCode = 0x4004
84
85     if (CodeSize >= 8)                                                   //
CodeSize = 0x0d (Incremented during caller loop)
86         RawCode += (0x10000 * Raster[ByteOffset + 2]);                   //
RawCode = 0x474004
87
88     RawCode >>= (BitOffset % 8);                                         //
RawCode = 0x11d001
89     BitOffset += (int)CodeSize;                                          //
BitOffset = 0x27
90     return (RawCode & ReadMask);                                         //
ReadMask = 0x1fff, (RawCode & ReadMask) = 0x1001 [This is our EOF Code]
91 }
```

Once the `EOFCode` is encountered, the heap buffers are freed.

```
323     free(Prefix);
324     free(Suffix);
325     free(OutCode);
```

Thus triggering the crash

```
double free or corruption (!prev)

Program received signal SIGABRT, Aborted.
```

TIMELINE

2022-03-17 - Initial Vendor Contact

2022-03-21 - Vendor Disclosure

2022-08-16 - Public Release

CREDIT

Discovered by Dave McDaniel of Cisco Talos.
