==[Basic information]==

Name   : zlib Use of Uninitialized "check" Stream Field Vulnerability
Impact   : Low
Class   : Potential disclosure of uninitialized heap memory
Discovered   : 2014-01-17
Reported   : 2014-01-22
Published   : 2014-04-30
Credit   : Mateusz "j00ru" Jurczyk of the Google Security Team
Vulnerable   : zlib 1.2.8 and previous versions

==[Introduction]==

Software depending on zlib 1.2.8 and previous versions which use a specific code pattern to interact with the compression library to decompress DEFLATE-compressed data may be affected by a “use of uninitialized heap memory” bug due to lack of proper initialization of the “inflate_state.check” internal structure field performed by zlib when handling incorrectly formatted input data. The only two open-source clients confirmed to use the code pattern required to trigger the condition are latest versions of the FFmpeg transcoding library (“Flash Screen Video decoder” component) and LibTIFF image library (ZIP and PixarLog compression support); however, other clients might also suffer from the problem.

On the example of LibTIFF 4.0.3 and Safari 7.0.1 running on Mac OS X 10.9.1, we have shown that certain scenarios may allow an attacker to use specially crafted input data to reason about the properties of the uninitialized memory, thus potentially gaining access to sensitive, leftover information stored in the process heap. Overall, however, there is a number of limitations which - in our opinion - make practical attacks infeasible and unlikely to take place in real world; this is primarily due to the volume of necessary input data and a finite number of disclosed bits (a maximum of 32 bits at a time). These and other limitations are explained in more detail later in the advisory.

==[Description]==

The inflation process using zlib starts with an inflateInit() call, which allocates an internal “inflate_state” structure of around 7,000 bytes (depending on the target architecture):

```
--- inflate.c ---
208:    state = (struct inflate_state FAR *)
209:            ZALLOC(strm, 1, sizeof(struct inflate_state));
--- inflate.c ---
```

The “ZALLOC” macro is defined as:

```
--- zutil.h ---
244: #define ZALLOC(strm, items, size) \
245:            (*((strm)->zalloc))((strm)->opaque, (items), (size))
--- zutil.h ---
```

If the caller specifies its own memory allocator, it is used accordingly; otherwise, the default “zcalloc” allocator is invoked:

```
--- zutil.c ---
304: voidpf ZLIB_INTERNAL zcalloc (opaque, items, size)
305:     voidpf opaque;
306:     unsigned items;
307:     unsigned size;
308: {
309:     if (opaque) items += size - size; /* make compiler happy */
310:     return sizeof(uInt) > 2 ? (voidpf)malloc(items * size) :
311:                               (voidpf)calloc(items, size);
312: }
--- zutil.c ---
```

The above translates to a malloc() call on 32-bit and 64-bit platforms, which does not guarantee that the allocated memory area must be zero-initialized. Therefore, the presence of the bug depends on a client which doesn’t provide its own memory management interface (true in most cases) or provides one that doesn’t pre-initialize newly allocated memory blocks.

Most fields in the structure are initialized by the inflateInit2(), inflateReset() or inflateResetKeep() routines (all are descendants of inflateInit()); the remaining portions of the state are generally written to during the “HEAD” state of inflate(). It turns out, however, that if either of the early header/compression checks fail (lines 657-670), the “dmax” and “check” fields are not properly initialized (lines 680 and 682):

```
--- inflate.c ---
657:             if (!(state->wrap & 1) ||   /* check if zlib header allowed */
658: #else
659:             if (
660: #endif
661:                 ((BITS(8) << 8) + (hold >> 8)) % 31) {
662:                 strm->msg = (char *)"incorrect header check";
663:                 state->mode = BAD;
664:                 break;
665:             }
```

```
666:             if (BITS(4) != Z_DEFLATED) {
667:                 strm->msg = (char *)"unknown compression method";
668:                 state->mode = BAD;
669:                 break;
670:             }
671:             DROPBITS(4);
672:             len = BITS(4) + 8;
673:             if (state->wbits == 0)
674:                 state->wbits = len;
675:             else if (len > state->wbits) {
676:                 strm->msg = (char *)"invalid window size";
677:                 state->mode = BAD;
678:                 break;
679:             }
680:             state->dmax = 1U << len;
681:             Tracev((stderr, "inflate:   zlib header ok\n"));
682:             strm->adler = state->check = adler32(0L, Z_NULL, 0);
683:             state->mode = hold & 0x200 ? DICTID : TYPE;
684:             INITBITS();
685:             break;
--- inflate.c ---
```

While “dmax” is initially written to by inflateResetKeep(), it is not the same for “check”, which still holds whatever value was located at that heap address in the previous allocation. In most cases, the caller would consider the error critical and bail out upon the first failed inflate() call. However, some software attempt to recover from the condition and continue the decompression process by taking advantage of the inflateSync() function. One instance of such software is LibTIFF, which uses the following decompression loop:

```
--- libtiff/tif_zip.c ---
170:     do {
171:         int state = inflate(&sp->stream, Z_PARTIAL_FLUSH);
172:         if (state == Z_STREAM_END)
173:             break;
174:         if (state == Z_DATA_ERROR) {
175:             TIFFErrorExt(tif->tif_clientdata, module,
176:                 "Decoding error at scanline %lu, %s",
177:                 (unsigned long) tif->tif_row, sp->stream.msg);
178:             if (inflateSync(&sp->stream) != Z_OK)
179:                 return (0);
180:             continue;
181:         }
182:         if (state != Z_OK) {
183:             TIFFErrorExt(tif->tif_clientdata, module, "ZLib error: %s",
184:                 sp->stream.msg);
```

```
185:             return (0);
186:         }
187:     } while (sp->stream.avail_out > 0);
--- libtiff/tif_zip.c ---
```

Once the inflate() function returns an error code, the library attempts to gracefully handle the supposedly corrupted input file by trying to synchronize to the nearest DEFLATE flush point, implemented as shifting the input pointer to the next sequence of 0x00, 0x00, 0xff, 0xff bytes. During the second inflate() call (post-resynchronization), header parsing is skipped and the function proceeds directly to processing of compressed data, due to inflateSync() setting the internal state to “TYPE” (line 1416):

```
--- inflate.c ---
1411:    /* return no joy or set up to restart inflate() on a new block */
1412:    if (state->have != 4) return Z_DATA_ERROR;
1413:    in = strm->total_in;  out = strm->total_out;
1414:    inflateReset(strm);
1415:    strm->total_in = in;  strm->total_out = out;
1416:    state->mode = TYPE;
1417:    return Z_OK;
--- inflate.c ---
```

The “TYPE” mode of operation (representing DEFLATE parsing) and all further modes assume that the “check” field has been previously initialized. Upon the completion of DEFLATE data decompression, the final “CHECK” state is entered, which uses the uninitialized value to seed the ADLER32 hash update function and later to compare the new hash against a 32-bit hash value found in the input stream. The “if” statement in lines 1184 - 1188 is the first location in the code where a decision is made based on the non-deterministic initial value of “check”:

```
--- inflate.c ---
1174:      case CHECK:
1175:             if (state->wrap) {
1176:                 NEEDBITS(32);
1177:                 out -= left;
1178:                 strm->total_out += out;
1179:                 state->total += out;
1180:                 if (out)
1181:                     strm->adler = state->check =
1182:                     UPDATE(state->check, put - out, out);
1183:                 out = left;
1184:                 if ((
1185: #ifdef GUNZIP
1186:                      state->flags ? hold :
```

```
1187: #endif
1188:                       ZSWAP32(hold)) != state->check) {
1189:                     strm->msg = (char *)"incorrect data check";
1190:                     state->mode = BAD;
1191:                     break;
1192:                 }
1193:                 INITBITS();
1194:                 Tracev((stderr, "inflate:   check matches trailer\n"));
1195:             }
1196: #ifdef GUNZIP
1197:           state->mode = LENGTH;
--- inflate.c ---
```

If the two 32-bit values match, inflate() returns Z_STREAM_END and the decompression is considered complete. Otherwise, a Z_DATA_ERROR exit code is returned, but the internal state still reflects the outcome of successful decompression, i.e. “avail_out” and “next_out” fields are updated, and the only relevant difference in program state is the return value.

For inflate/inflateSync decompression loops such as the one implemented in LibTIFF, it is possible to try to guess the initial leftover “check” contents by crafting a long sequence of DEFLATE-compressed bytes separated by flush point signatures and ADLER-32 hash values corresponding to the guessed values. This concept is further discussed in the following section.

==[Proof of Concept]==

As it is possible to efficiently compute the ADLER-32 hash for any data and assumed initial seed, we can predict the values being compared against in the “CHECK” mode depending on the initial contents of the uninitialized fields. As a result, it is possible to create the following input data stream:

```
[invalid header]
[flush point signature] [deflate(byte)] [ADLER-32 hash guess]
[flush point signature] [deflate(byte)] [ADLER-32 hash guess]
.
.
.
[flush point signature] [deflate(byte)] [ADLER-32 hash guess]
```

If the number of entries in such a stream equals the number of expected output bytes plus one, then in case of the aforementioned inflate/inflateSync loop:

  (*) if one of the entries contains a correct guess (ADLER-32 hash of the data decompressed so far seeded with the initial value of the uninitialized “check”

field), inflate() returns Z_STREAM_END, the loop terminates and data decompression fails because not enough bytes are found in the output buffer (strm.avail_out != 0).
  (*) if none of the entries match the actual hash value (i.e. the initial "check" value was outside the scope of all guesses in the stream), the entirety of the output buffer is filled with data and thus decompression succeeds.

Obviously, the above behavior can be used to determine if 32 bits of leftover heap memory is equal to a particular set of values specified in the input stream or not, based on the success of decompression (manifested by rendering or not rendering an image in case of most LibTIFF clients).

More interestingly, though, timing attacks could also be used to extract portions of information regarding the value being disclosed - even if stream decoding fails, the amount of time it takes to fail is directly proportional to the offset of the stream entry containing the valid guess. This can be demonstrated on the example of the "tiffinfo" utility (part of LibTIFF); if we compile the executable with a modified version of libz which prints out the leftover value immediately after allocation, and run it against a .TIFF file containing 2^24 guesses that the hash is a 0xxxxxx0 value (00000000, 00000010, ..., 0ffffff0), it is clearly visible that the processing time reveals the estimate range of the number:

```
---
$ time tools/tiffinfo -D -i 4096x4096.tif 2>/dev/null

TIFF Directory at offset 0x8 (8)
  Image Width: 4096 Image Length: 4096
  Resolution: 200, 200 pixels/inch
  Bits/Sample: 8
  Compression Scheme: Deflate
  Photometric Interpretation: min-is-black
  Orientation: row 0 top, col 0 lhs
  Samples/Pixel: 1
  Rows/Strip: 4096
  Planar Configuration: single image plane
  Color Map: (present)

[zlib] initial state->check: 3fab020

real    0m4.026s
user    0m2.380s
sys     0m1.620s

$ time tools/tiffinfo -D -i 4096x4096.tif 2>/dev/null
```

```
[...]
[zlib] initial state->check: 1fd5490

real    0m2.146s
user    0m1.320s
sys     0m0.800s

$ time tools/tiffinfo -D -i 4096x4096.tif 2>/dev/null

[...]
[zlib] initial state->check: 275ae90

real    0m2.543s
user    0m1.470s
sys     0m1.050s

$ time tools/tiffinfo -D -i 4096x4096.tif 2>/dev/null

[...]
[zlib] initial state->check: 51a2ed0

real    0m5.050s
user    0m2.760s
sys     0m2.260s
---
```

Enclosed with this advisory is the source code (zlib_stream_gen.py, tif_poc.nasm files) and compiled, binary proof of concept file (256x256.tif). The image contains 65536 DEFLATE entries, each guessing that the initial "check" value is equal to its index in the stream, covering the [0000 ... ffff] range. When opened in Safari 7.0.1 on a Mac OS X 10.9.1, the image will sometimes render and sometimes not, depending on whether any of the upper 16 bits of the uninitialized value are set.

Note that while it might be potentially possible to disclose certain characteristics of the 32-bit value, extracting concrete values of each or all bits is usually not feasible. There are several primary problems stopping most or all practical attacks:

  (*) The ADLER-32 hash function doesn't use the full 32-bit space, so hash collisions may occur, sometimes making it impossible to distinguish between two seeds resulting in the same output hash for specific input data.
  (*) For each 32-bit value guess, at least 11 bytes of data in the input stream are required (4 bytes for the synchronization signature, 3 for a single deflated data and 4 for the hash value). As a consequence, streams testing a 24 bit space consume 177MB of disk space / internet transfer / memory, while

streams testing 2^28 values are 2.75GB in size.

(*) Processing of such large volumes of data (not to mention storage and transfer over a wire) takes a considerable amount of time, making attacks “noisy” (if at all possible). For example, Safari on a 2.26 GHz Intel Core 2 Duo MacBook Pro is only able to process around 65535 pixels (16 bits worth of guesses) per second according to our tests.

The above limitations make the vulnerability unlikely to be used in generic attacks against users; however, it might still prove useful in very specific scenarios.