ZLib security review. Intermediate observations.

This review is currently ongoing. ZLib uses a complex state machine that takes a long time to wrap your head around. I've mapped parts of it, but not all of it. I have in the meantime -as I'm trying to understand this state machine- found some issues already. Below I'll document the issues I've found, while at the same time trying to document parts of the state machine I understand at the present time.

Buffer overflow in ZLib in gunzip mode

ZLib is an interesting library. Decompression revolves around the `inflate()` function. The idea is that you can have multiple chunks of input (compressed) and output (uncompressed) and call `inflate()` X number of times in a loop until all input is decompressed. You define the input and output pointers and length by filling in a z_stream structure and passing it along to `inflate()` doing so looks like:

```
  z_stream strm;
   ...
   strm.next_in = compressed_input_chunk;
   strm.avail_in = compressed_input_chunk_len;
   strm.next_out = decompressed_output_chunk;
   strm.next_avail= decompressed_output_chunk_len;
   inflateInit(&strm);
   ...
   while(...) {
       inflate(&strm, Z_NO_FLUSH)
   }
```

inside `inflate()` is basically a very large state machine that parses input and writes to output. This state machine has to survive multiple function entries and exits as input is consumed and output is written to. When looking at the state machine, you'll notice use of several macros that deal with several variables used to represent various input and output. Here's what Inflate roughly looks like:

```
int ZEXPORT inflate(z_streamp strm, int flush)
{
    struct inflate_state *state;
    z_const unsigned char *next;    /* next input */
    unsigned char *put;     /* next output */
    unsigned have, left;        /* available input and output */
    unsigned long hold;         /* bit buffer */
    unsigned bits;              /* bits in bit buffer */
    unsigned in, out;           /* save starting available input and output */
    unsigned copy;              /* number of stored or match bytes to copy */
    unsigned char *from;    /* where to copy match bytes from */
    code here;                  /* current decoding table entry */
    code last;                  /* parent table entry */
    unsigned len;               /* length to copy for repeats, bits to drop */
    int ret;                    /* return code */
```

```
...
    state = (struct inflate_state *)strm->state;
...
    LOAD();
    in = have;
    out = left;
    ret = Z_OK;
...
```

This is the function epilogue. It declares all the variables needed by inflate and then initializes them. The first thing it does it take a local copy of the stream state (needed so state survives multiple function entry and exits), then `LOAD()` is called, which initialized more local variables based in stream data, here's what `LOAD()` looks like:

```
/* Load registers with state in inflate() for speed */
#define LOAD() \
    do { \
        put = strm->next_out; \
        left = strm->avail_out; \
        next = strm->next_in; \
        have = strm->avail_in; \
        hold = state->hold; \
        bits = state->bits; \
    } while (0)
```

put is the decompressed output buffer, left is its length, next is the compressed input buffer, have is its length. hold is used to store X number of bits (1-32) read from the compressed input, and bits is used to say how many bits were read. Now that the `LOAD()` macro is briefly explained, lets return to `inflate()`:

```
    for (;;)
        switch (state->mode) {
        case HEAD:
...
            NEEDBITS(16);
```

The for loop and switch/case mark the beginning of the state machine. `state->mode` is an enum (named `inflate_mode`) that defines all possible states, this looks like:

```
/* Possible inflate modes between inflate() calls */
typedef enum {
    HEAD = 16180,   /* i: waiting for magic header */
    FLAGS,      /* i: waiting for method and flags (gzip) */
    TIME,       /* i: waiting for modification time (gzip) */
    OS,         /* i: waiting for extra flags and operating system (gzip) */
    EXLEN,      /* i: waiting for extra length (gzip) */
    EXTRA,      /* i: waiting for extra bytes (gzip) */
    NAME,       /* i: waiting for end of file name (gzip) */
    COMMENT,    /* i: waiting for end of comment (gzip) */
    HCRC,       /* i: waiting for header crc (gzip) */
    DICTID,     /* i: waiting for dictionary check value */
    DICT,       /* waiting for inflateSetDictionary() call */
        TYPE,       /* i: waiting for type bits, including last-flag bit */
        TYPEDO,     /* i: same, but skip check to exit inflate on new block */
        STORED,     /* i: waiting for stored size (length and complement) */
        COPY_,      /* i/o: same as COPY below, but only first time in */
        COPY,       /* i/o: waiting for input or output to copy stored block */
        TABLE,      /* i: waiting for dynamic block table lengths */
        LENLENS,    /* i: waiting for code length code lengths */
        CODELENS,   /* i: waiting for length/lit and distance code lengths */
            LEN_,       /* i: same as LEN below, but only first time in */
            LEN,        /* i: waiting for length/lit/eob code */
            LENEXT,     /* i: waiting for length extra bits */
            DIST,       /* i: waiting for distance code */
            DISTEXT,    /* i: waiting for distance extra bits */
            MATCH,      /* o: waiting for output space to copy string */
            LIT,        /* o: waiting for output space to write literal */
    CHECK,      /* i: waiting for 32-bit check value */
    LENGTH,     /* i: waiting for 32-bit length (gzip) */
    DONE,       /* finished check, done -- remain here until reset */
    BAD,        /* got a data error -- remain here until reset */
    MEM,        /* got an inflate() memory error -- remain here until reset */
    SYNC        /* looking for synchronization bytes to restart inflate() */
} inflate_mode;
```

As you can see `HEAD` is the first state, which deals with a magic header. The next thing we see inside the `inflate()` state machine is use of the `NEEDBITS()` macro. This macro looks like:

```
 /* Assure that there are at least n bits in the bit accumulator.  If there is
   not enough available input to do that, then return from inflate(). */
#define NEEDBITS(n) \
    do { \
        while (bits < (unsigned)(n)) \
            PULLBYTE(); \
    } while (0)


/* Get a byte of input into the bit accumulator, or return from inflate()
   if there is no input available. */
#define PULLBYTE() \
    do { \
        if (have == 0) goto inf_leave; \
        have--; \
        hold += (unsigned long)(*next++) << bits; \
        bits += 8; \
    } while (0)
```

`NEEDBITS()` essentially reads bytes from the compressed input buffer till the number of bytes read satisfies the amount of bits needed. It stores these bits inside the `hold` variable. There is also a `goto inf_leave` in there. This will be covered in more detail below. For now, suffice it to say that `goto inf_leave` is how function exit occurs with a saved state. Now that we've defines all states and shows what the `NEEDBITS()` macro is like, lets dive back into the `HEAD` state inside of `inflate()`:

```
#ifdef GUNZIP
            if ((state->wrap & 2) && hold == 0x8b1f) {  /* gzip header */
...
                INITBITS();
                state->mode = FLAGS;
                break;
            }
#endif
...
```

state->wrap defined whether to use zlib mode or gunzip mode. This is set when calling `InflateInit()` or `inflateInit2()` and is documented as follows:

```
      int wrap;                   /* bit 0 true for zlib, bit 1 true for gzip,
                                     bit 2 true to validate check value */
```

Given that the bug to follow is in gunzip mode, we're going to assume bit 1 of wrap is set. Hold is previously filled with the use of the `NEEDBITS(16)` macro. If the right magic is found, a call to `INITBITS()` happens, followed by a state change to `FLAGS`. Lets first take a look at the `INITBITS()` macro.

```
/* Clear the input bit accumulator */
#define INITBITS() \
    do { \
        hold = 0; \
        bits = 0; \
    } while (0)
```

This Macro is really simple. It essentially resets the hold variable. Whatever was previously into it is now gone. You'll often see `INITBITS()` called several lines after a call to `NEEDBITS()`. The call the `INITBITS()` means the data acquired by the `NEEDBITS()` call is no longer needed. Now that the `*BITS()` macro's are understood, and we've covered the first state and it's transition, lets jump back to `inflate()` and see what's next:

```
            break;
#ifdef GUNZIP
        case FLAGS:
            NEEDBITS(16);
            state->flags = (int)(hold);
...
            INITBITS();
            state->mode = TIME;
        case TIME:
            NEEDBITS(32);
...
            INITBITS();
            state->mode = OS;
        case OS:
            NEEDBITS(16);
...
            INITBITS();
            state->mode = EXLEN;
```

`FLAGS`, `TIME`, `OS` are all trivial states that directly follow each other. There isn't much to see here, however, it clearly shows that every call to `NEEDBITS()` is balanced by a call to `INITBITS()`. The next 2 states `EXLEN` and `EXTRA` is where things get interesting.

```
        case EXLEN:
            if (state->flags & 0x0400) {
                NEEDBITS(16);
                state->length = (unsigned)(hold);
                if (state->head != Z_NULL)
                    state->head->extra_len = (unsigned)hold;
                if ((state->flags & 0x0200) && (state->wrap & 4))
                    CRC2(state->check, hold);
                INITBITS();
            }
            else if (state->head != Z_NULL)
                state->head->extra = Z_NULL;
            state->mode = EXTRA;
```

If the previously read flags indicate it, the EXLEN state will read 16 bits and assign then to `state->length`. Assuming a state header structure is defined (this is done by a previous call to `inflateGetHeader(strm, head)`) `state->head->extra_len` is given the same value. The reason why this is done becomes clear once the transition to the `EXTRA` state is performed:

```
        case EXTRA:
...
            if (state->flags & 0x0400) {
                copy = state->length;
                if (copy > have) copy = have;
                if (copy) {
                    if (state->head != Z_NULL &&
                        state->head->extra != Z_NULL) {
                        len = state->head->extra_len - state->length;
                        zmemcpy(state->head->extra + len, next,
                                len + copy > state->head->extra_max ?
                                state->head->extra_max - len : copy);
                    }
                    if ((state->flags & 0x0200) && (state->wrap & 4))
                        state->check = crc32(state->check, next, copy);
                    have -= copy;
                    next += copy;
                    state->length -= copy;
                }
                if (state->length) goto inf_leave;
            }
            state->length = 0;
            state->mode = NAME;
....
```

Ok, so there's a lot going on here. This state first sees how many bytes it wants to copy (`copy`) and then adjusts `copy` if the compressed input buffer length (`have`) is smaller than the entire copy length. This might seem strange, however, this function can make use of multiple `inflate()` function entries and exits where new input buffer chunks are read at every iteration.

Assuming a state header structure is defined a `len` is calculated. `len` is really a misnomer, and it should've been called offset. The calculation is `len = state->head->extra_len - state->length;` This might seem strange, given that they're both the same (see `EXLEN` state description), however, `state->len` is updated for every iteration (substracts how many bytes were copies) so the offset grows for every iteration (when I say 'iteration' I mean `inflate()` function exit and re-entry after reading in more compressed input).

Now a `memcpy()` occurs. `zmemcpy()` is really just a macro that uses `memcpy()`. The `memcpy()` is pretty complex so lets look at it in pieces:

```
                    zmemcpy(state->head->extra + len,         // [1]
                            next,                             // [2]
                            len + copy > state->head->extra_max ? // [3]
                            state->head->extra_max - len : // [4]
                            copy // [5]
                    );
```

In `[1]` the destination pointer is calculated. this is the extra buffer inside the header structure and includes the offset. On the first iteration the offset is 0, so the destination buffer is simply the extra buffer. On subsequent iterations (`inflate()` function exit, read more compressed input stream, call `inflate()` again), as the offset grows, the destination buffer moves forward inside the extra buffer.

`[2]` is easy, this is the compressed input buffer chunk. This is the source that will be copied to the destination.

`[3]` uses the ?: conditional operator, which is really just if-then-else shorthand. `[3]` is essentially a bounds check. It checks to see if the amount of bytes you want to copy plus the offset is larger than `extra_max` (= size of the extra buffer).

`[4]` If it is in fact larger, the copy will be truncated, using only `extra_max` (extra length) minus the offset.

`[5]` if it is not larger, the copy won't be truncated and `copy` number of bytes will be copies from the source to the destination.

The next thing that happens in the `EXTRA` case is pointer and length adjustments (after the copy) lets take a look at this:

```
                have -= copy;
                next += copy;
                state->length -= copy;
```

The code substracts copy from `have` (size of compressed input buffer), increases `next` (which is the next input length) and then substracts `state->length` with copy as well. This last subtract is ***problematic!*** Remember that `state->length` is used to calculate the offset (`len`) into the `state->head->extra` destination buffer? substracting it with `copy` is only ok if the bounds check in [3] was not hit and [5] happened. If -on the other hand- the bounds check was hit in `[3]` and `[4]` happened, then `state->length` should've been substracted with `state->head->extra_max - len`. Because of this wrong decrement of `state->length` a buffer overflow can occur in the `zmemcpy()` in the next iteration.

Potential non 0-terminated strings when using ZLib incorrectly

Moving on to the next state `NAME` (and after that `COMMENT`, the code for both is virtually identical). The code for the `NAME` state is as follows:

```
        case NAME:
            if (state->flags & 0x0800) {
                if (have == 0) goto inf_leave;
                copy = 0;
                do {
                    len = (unsigned)(next[copy++]);
                    if (state->head != Z_NULL &&
                            state->head->name != Z_NULL &&
                            state->length < state->head->name_max)
                        state->head->name[state->length++] = (Bytef)len;
                } while (len && copy < have);
                if ((state->flags & 0x0200) && (state->wrap & 4))
                    state->check = crc32(state->check, next, copy);
                have -= copy;
                next += copy;
                if (len) goto inf_leave;
            }
            else if (state->head != Z_NULL)
                state->head->name = Z_NULL;
            state->length = 0;
            state->mode = COMMENT;
```

This is a similar-ish copy loop, except minus the memory corruption issue. it uses `len` again, but this time to hold a byte read (so, again a misnomer, they should've called it `c` or something). The code copies compressed input data a byte at a time, in a loop. It copies to `state->head->name` making sure to perform a bounds check. It copies until it sees a 0-byte or there is no more data to read. It does this while `inflate()` exit and re-entry can occur. The issue here is that it's clearly trying to handle a 0-terminated string, but doesn't guarantee `state->head->name` is 0-terminated if the input is larger than the output, this behavior is much like `strncpy()`. Like `strncpy()` this behavior is in fact documented. Looking at the documentation for `inflateGetHeader()` at https://zlib.net/manual.html it states the following:

> If name is not `Z_NULL`, then up to name_max characters are written there, terminated with a zero unless the length is greater than name_max. If comment is not `Z_NULL`, then up to comm_max characters are written there, terminated with a zero unless the length is greater than comm_max.

However, much like with `strncpy()` developers often don't or selectively read documentation and just assume an API does what they want. As such it's likely that users of `state->head->name` and `state->head->comment` will simply assume the string is always 0-terminated. This can lead to `strncpy()` like issues (see http://phrack.org/issues/56/14.html for more details).

potential endless loop when using ZLib incorrectly

Going back to the beginning of the `inflate()` function and its state machine. Assuming we're not interested in the gunzip part, but only the zlib part. The beginning of the state machine plays out quite differently. Let's take a look:

```
int ZEXPORT inflate(z_streamp strm, int flush)
{
...
    LOAD();
...
    for (;;)
        switch (state->mode) {
        case HEAD:
...
            NEEDBITS(16);
...
            DROPBITS(4);
            len = BITS(4) + 8;
...
            state->mode = hold & 0x200 ? DICTID : TYPE;
            INITBITS();
            break;
...
        case DICTID:
            NEEDBITS(32);
            strm->adler = state->check = ZSWAP32(hold);
            INITBITS();
            state->mode = DICT;
        case DICT:
            if (state->havedict == 0) {
                RESTORE();
                return Z_NEED_DICT;
            }
            strm->adler = state->check = adler32(0L, Z_NULL, 0);
            state->mode = TYPE;
...
```

if the right bits are set in the `HEAD` state then the next state will be `DICTID`. This state tells zlib to use a custom dictionary and gives it the adler32 checksum of the custom dictionary. It then moves to the `DICT` state, where it checks `state->havedict` This will be 0, since no custom dictionary is set by default. This will trigger an immediate Z_NEED_DICT return. It is expected that the caller check for this return value and then call the `inflateSetDictionary()` API. This is a relatively rare condition, and most callers will in fact not do this. If you look at the definition of `Z_NEED_DICT` you will note that it is not an error return value, and as such won't be seen as an error case:

```
/* Return codes for the compression/decompression functions. Negative values
 * are errors, positive values are used for special but normal events.
 */
#define Z_OK            0
#define Z_STREAM_END    1
#define Z_NEED_DICT     2
#define Z_ERRNO        (-1)
#define Z_STREAM_ERROR (-2)
#define Z_DATA_ERROR   (-3)
#define Z_MEM_ERROR    (-4)
#define Z_BUF_ERROR    (-5)
#define Z_VERSION_ERROR (-6)
```

This means, a zlib stream of compressed input could trigger the DICTID state, and have `inflate()` return Z_NEED_DICT. A lot of zlib callers will see this as a normal event (not an error) and call `inflate()` again, triggering the same state over and over, leading to an endless loop.

Potential use of uninitialized pointers when using ZLib incorrectly

XXXTODOXXX

```

```