Talos Vulnerability Report

TALOS-2021-1434

Sound Exchange libsox sphere.c start_read() heap-based buffer overflow vulnerability

MARCH 23, 2022

CVE NUMBER

CVE-2021-40426

SUMMARY

A heap-based buffer overflow vulnerability exists in the sphere.c start_read() functionality of Sound Exchange libsox 14.4.2 and master commit 42b3557e. A specially-crafted file can lead to a heap buffer overflow. An attacker can provide a malicious file to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

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

Sound Exchange libsox 14.4.2
Sound Exchange libsox master commit 42b3557e

PRODUCT URLS

libsox - http://sox.sourceforge.net/Main/HomePage

CVSSV3 SCORE

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

CWE

CWE-122 - Heap-based Buffer Overflow

DETAILS

Libsox is a well-aged library used for cross-platform audio editing software, originally written in 1991. After decades of development, a wide range of file formats are supported, including .wav, .flac, and .mp3 (with the aid of an external library).

Out of the multitude of file formats that Sound Exchange's libsox can deal with, today we discuss the extremely obscure NIST Speech Header Resources (SPHERE) file format, which apparently is used for speech recognition. But regardless of the purpose, libsox will still process a given file as a `.sph` assuming that the file header matches:

```
 static char const * auto_detect_format(sox_format_t * ft, char const * ext)
{
  char data[AUTO_DETECT_SIZE];
  size_t len = lsx_readbuf(ft, data, ft->seekable? sizeof(data) : PIPE_AUTO_DETECT_SIZE);
  #define CHECK(type, p2, l2, d2, p1, l1, d1) if (len >= p1 + l1 && \
      !memcmp(data + p1, d1, (size_t)l1) && !memcmp(data + p2, d2, (size_t)l2)) return #type;
  // [...]
  CHECK(sph   , 0, 0, ""     , 0,  7, "NIST_1A")
```

This `auto_detect_format` will be hit from either `sox_open_mem_read` or `sox_open_read`, but only if the filetype is not specified by the arguments. Either way, since it's possible to hit the processing in `sphere.c` via autodetection, let us look at the `start_read` function for the SPHERE file format:

```
static int start_read(sox_format_t * ft)
{
  unsigned long header_size_ul = 0, num_samples_ul = 0;
  size_t     header_size, bytes_read;
  // [...]
  char           fldname[64], fldtype[16], fldsval[128];
  char           * buf;

  /* Magic header */
  if (lsx_reads(ft, fldname, (size_t)8) || strncmp(fldname, "NIST_1A", (size_t)7) != 0) {   // [1]
    lsx_fail_errno(ft, SOX_EHDR, "Sphere header does not begin with magic word `NIST_1A'");
    return (SOX_EOF);
  }

  if (lsx_reads(ft, fldsval, (size_t)8)) {                                                 // [2]
    lsx_fail_errno(ft, SOX_EHDR, "Error reading Sphere header");
    return (SOX_EOF);
  }

  /* Determine header size, and allocate a buffer large enough to hold it. */
  sscanf(fldsval, "%lu", &header_size_ul);                                                 // [3]
  if (header_size_ul < 16) {
    lsx_fail_errno(ft, SOX_EHDR, "Error reading Sphere header");
    return (SOX_EOF);
  }

  buf = lsx_malloc(header_size = header_size_ul);                                          // [4]
```

So far the code is pretty standard. We read eight bytes at [1], make sure the “NIST_1A” magic bytes are there, and then read another eight bytes at [2], this time populating them into the `unsigned long header_size_ul` at [3]. At [4], we then allocate a buffer of that size. Again, pretty standard. Continuing on:

```
/* Skip what we have read so far */
header_size -= 16;

if (lsx_reads(ft, buf, header_size) == SOX_EOF) {       // [5]
  lsx_fail_errno(ft, SOX_EHDR, "Error reading Sphere header");
  free(buf);
  return (SOX_EOF);
}

header_size -= (strlen(buf) + 1);
```

At [5] we see the main function used for reading our input buffer, `lsx_reads`. Suffice to say, this function is essentially `fgets` (although I’m not actually sure which function has seniority), and `buf` will contain a new line of text from our input buffer every time `lsx_reads` is called. We now finally reach our main processing loop:

```
while (strncmp(buf, "end_head", (size_t)8) != 0) {    // [6]

  if (strncmp(buf, "sample_n_bytes", (size_t)14) == 0)
    sscanf(buf, "%63s %15s %u", fldname, fldtype, &bytes_per_sample);
  else if (strncmp(buf, "channel_count", (size_t)13) == 0)
    sscanf(buf, "%63s %15s %u", fldname, fldtype, &channels);
    // [...]
    else {
      lsx_fail_errno(ft, SOX_EFMT, "sph: unsupported coding `%s'", fldsval);
      free(buf);
      return SOX_EOF;
    }
  }
  else if (strncmp(buf, "sample_byte_format", (size_t)18) == 0) {
    sscanf(buf, "%53s %15s %127s", fldname, fldtype, fldsval);
    if (strcmp(fldsval, "01") == 0)         /* Data is little endian. */
      ft->encoding.reverse_bytes = MACHINE_IS_BIGENDIAN;
    else if (strcmp(fldsval, "10") == 0)    /* Data is big endian. */
      ft->encoding.reverse_bytes = MACHINE_IS_LITTLEENDIAN;
    else if (strcmp(fldsval, "1")) {
      lsx_fail_errno(ft, SOX_EFMT, "sph: unsupported coding `%s'", fldsval);
      free(buf);
      return SOX_EOF;
    }
  }

  if (lsx_reads(ft, buf, header_size) == SOX_EOF) {   // [7]
    lsx_fail_errno(ft, SOX_EHDR, "Error reading Sphere header");
    free(buf);
    return (SOX_EOF);
  }

  header_size -= (strlen(buf) + 1);
```

At [6], we can see that we’re looping until a `end_head` header is found, but in the meantime, we search for various headers in the file (e.g. "`channel_count`", "`sample_byte_format`", etc.). After going through the specific buffer line, we read a new one in at [7], then subtract the length of our new buffer plus one at [8]. The `+ 1` compensates for the `\n` or `\x00` bytes that delimit the headers of our file.

But let’s examine a situation in which we provide a file that does not contain a valid `end_head` header:

```
while (strncmp(buf, "end_head", (size_t)8) != 0) {  // [6]

  if (strncmp(buf, "sample_n_bytes", (size_t)14) == 0)
    sscanf(buf, "%63s %15s %u", fldname, fldtype, &bytes_per_sample);
  else if (strncmp(buf, "channel_count", (size_t)13) == 0)
      // [...]
  }

  if (lsx_reads(ft, buf, header_size) == SOX_EOF) { // [7]
    lsx_fail_errno(ft, SOX_EHDR, "Error reading Sphere header");
    free(buf);
    return (SOX_EOF);
  }

  header_size -= (strlen(buf) + 1);
```

At [6] again, we start our loop, which eventually processes our header strings inside, but then at [7] the next header is read via `lsx_reads`. A quick look at `lsx_reads` reveals an important detail:

```
int lsx_reads(sox_format_t * ft, char *c, size_t len)
{
    char *sc;
    char in;

    sc = c;
    do  // [8]
    {
        if (lsx_readbuf(ft, &in, (size_t)1) != 1)
        {
            *sc = 0;
            return (SOX_EOF);
        }
        if (in == 0 || in == '\n')
            break;

        *sc = in;
        sc++;
    } while (sc - c < (ptrdiff_t)len);  // [9]
    *sc = 0;
    return(SOX_SUCCESS);
}
```

Since `lsx_reads` utilizes a `do/while` loop ([8], [9]), even though the conditional at [9] depends on a less-than sign and not a less-than-equals-to sign. If our input `len` parameter is 22, then 22 bytes of data are copied over, followed by the null byte write. So in effect, 23 bytes of data get written from a 22 byte len, which is an off-by-one. Curiously, in spite of `lsx_reads` ubiquitous usage in libsox, all the other call sites compensate for this off-by-one, usually with a `-1` to the input `len` parameter when calling. However, in the `sphere.c` code, we see a different compensation:

```
    if (lsx_reads(ft, buf, header_size) == SOX_EOF) {   // [10]
      lsx_fail_errno(ft, SOX_EHDR, "Error reading Sphere header");
      free(buf);
      return (SOX_EOF);
    }

    header_size -= (strlen(buf) + 1);                   // [11]
```

At [11], we can see that an extra byte is subtracted from our `header_size` parameter to compensate for the off-by-one in `lsx_reads`. Thus, if the `header_size` is 22 and our next buffer is read in, `lsx_reads` will stop at 22 bytes (regardless of the length of the rest of the buffer). Then we subtract 23 from `header_size`, resulting in an integer underflow.

With regards to exploitation: While normally such a situation might result in a wild copy (i.e. we couldn't stop the bug from writing into things we don't want it to and crashing), we're bailed out by the overall structure of this function. We'd simply have our next read be our overflow string of any size that we wanted, and then the following read just being `"end_head"`. This would allow us to arbitrarily overwrite data on the heap while also allowing us to continue into the code to actually utilize the heap overflow.

Crash Information

```
==778467==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6060000000ba at pc 0x0000004bea6e bp 0x7ffeacb7c950 sp 0x7ffeacb7c118
WRITE of size 203 at 0x6060000000ba thread T0
    #0 0x4bea6d in __interceptor_fread (/doop/boop/sox/triage_built/fuzz_sox.bin+0x4bea6d)
    #1 0x7ff5dc360ea7 in lsx_readbuf /doop/boop/sox/triage_built/triage_sox/src/formats_i.c:98:16
    #2 0x7ff5dc4b9bc0 in start_read /doop/boop/sox/triage_built/triage_sox/src/sphere.c:119:18
    #3 0x7ff5dc46c864 in open_read /doop/boop/sox/triage_built/triage_sox/src/formats.c:545:32
    #4 0x7ff5dc46d2bb in sox_open_mem_read /doop/boop/sox/triage_built/triage_sox/src/formats.c:595:10
    #5 0x55623a in LLVMFuzzerTestOneInput /doop/boop/sox/./fuzz_sox_harness.cpp:51:10
    #6 0x456ff3 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) fuzzer.o
    #7 0x442b32 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) fuzzer.o
    #8 0x448a5b in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) fuzzer.o
    #9 0x471ef2 in main (/doop/boop/sox/triage_built/fuzz_sox.bin+0x471ef2)
    #10 0x7ff5dbfa1fcf in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #11 0x7ff5dbfa207c in __libc_start_main csu/../csu/libc-start.c:409:3
    #12 0x41f7a4 in _start (/doop/boop/sox/triage_built/fuzz_sox.bin+0x41f7a4)

0x6060000000ba is located 0 bytes to the right of 58-byte region [0x606000000080,0x6060000000ba)
allocated by thread T0 here:
    #0 0x521d83 in __interceptor_realloc (/doop/boop/sox/triage_built/fuzz_sox.bin+0x521d83)
    #1 0x7ff5dc47b388 in lsx_realloc /doop/boop/sox/triage_built/triage_sox/src/xmalloc.c:37:14
    #2 0x7ff5dc4b9376 in start_read /doop/boop/sox/triage_built/triage_sox/src/sphere.c:55:9
    #3 0x7ff5dc46c864 in open_read /doop/boop/sox/triage_built/triage_sox/src/formats.c:545:32
    #4 0x7ff5dc46d2bb in sox_open_mem_read /doop/boop/sox/triage_built/triage_sox/src/formats.c:595:10
    #5 0x55623a in LLVMFuzzerTestOneInput /doop/boop/sox/./fuzz_sox_harness.cpp:51:10
    #6 0x456ff3 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) fuzzer.o
    #7 0x442b32 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) fuzzer.o
    #8 0x448a5b in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) fuzzer.o
    #9 0x471ef2 in main (/doop/boop/sox/triage_built/fuzz_sox.bin+0x471ef2)
    #10 0x7ff5dbfa1fcf in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16

SUMMARY: AddressSanitizer: heap-buffer-overflow (/doop/boop/sox/triage_built/fuzz_sox.bin+0x4bea6d) in __interceptor_fread
Shadow bytes around the buggy address:
  0x0c0c7fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff8000: fa fa fa fa fd fd fd fd fd fd fd fd fa fa fa fa
=>0x0c0c7fff8010: 00 00 00 00 00 00 00[02]fa fa fa fa fa fa fa fa
  0x0c0c7fff8020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00

  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==692318==ABORTING
```

TIMELINE

2021-12-22 - Initial contact

2022-01-14 - Follow up with vendor; vendor acknowledged

2022-03-23 - Vendor disclosure

CREDIT

Discovered by Lilith >_> of Cisco Talos.
