Talos Vulnerability Report

TALOS-2021-1299

# GPAC Project Advanced Content MPEG-4 Decoding multiple integer addition overflow vulnerabilities

AUGUST 16, 2021

### CVE NUMBER

CVE-2021-21853, CVE-2021-21854,CVE-2021-21855,CVE-2021-21856,CVE-2021-21857,CVE-2021-21858

### Summary

Multiple exploitable integer overflow vulnerabilities exist within the MPEG-4 decoding functionality of the GPAC Project on Advanced Content library v1.0.1. A specially crafted MPEG-4 input can cause an integer overflow due to unchecked addition arithmetic resulting in a heap-based buffer overflow that causes memory corruption. An attacker can convince a user to open a video to trigger this vulnerability.

### Tested Versions

GPAC Project Advanced Content commit a8a8d412dabcb129e695c3e7d861fcc81f608304
GPAC Project Advanced Content v1.0.1

### Product URLs

https://gpac.wp.mines-telecom.fr

### CVSSv3 Score

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

### CWE

CWE-680 - Integer Overflow to Buffer Overflow

### Details

The GPAC Project on Advanced Content is an open-source cross-platform library that implements the MPEG-4 Systems Standard, and provides tools for media playback, vector graphics, and 3d rendering. It supports a variety of multimedia standards and is thus used by a number of industrial users. The project also comes with the MP4Box tool which allows one to encode or decode media containers in a number of supported formats.

When the GPAC library is used to open up an MPEG-4 container, they library will proceed to read each particular atom from the container whilst noting the atom's "type" which is referred to as a FOURCC. This "type" is then used to distinguish which particular parser will be used to parse the contents of an atom. During the parsing of the various atom types inside the MPEG-4 container, the library will read fields from the atom's contents and in some cases will use them to calculate the boundaries of the fields contained within the rest of the atom. In some of these parsers, the fields are explicitly trusted and then used to calculate the size of a heap-buffer which is then later used by the library. When the library allocates space for reading the rest of an atom, the library may miscalculate this size either due to an integer overflow, or an integer truncation which can result in an undersized heap allocation being made. Later in the atom parsing, when the library attempts to read the atom's contents into this heap buffer, a heap-based buffer overflow can be made to occur. This can result in code execution under the context of the library.

The GPAC library provides a variety of tools that the implementer may use when processing an MPEG-4 container. This would allow a user to either process the MPEG-4 container in fragments, or as a whole and complete source. When parsing a complete MPEG-4 container, a developer may use the following `gf_isom_open` function. This function is responsible for looking at the flags that it was given and chaining to the correct function in order to parse the input. At [1], the library will use the `OpenMode` flags from its parameters in order to call the `gf_isom_open_file` to process the input.

```
src/isomedia/isom_read.c:500
GF_EXPORT
GF_ISOFile *gf_isom_open(const char *fileName, GF_ISOOpenMode OpenMode, const char *tmp_dir)
{
    GF_ISOFile *movie;
    MP4_API_IO_Err = GF_OK;

    switch (OpenMode & 0xFF) {
    case GF_ISOM_OPEN_READ_DUMP:
    case GF_ISOM_OPEN_READ:
        movie = gf_isom_open_file(fileName, OpenMode, NULL);        // [1] open up the given filename for reading
        break;
...
    default:
        return NULL;
    }
    return (GF_ISOFile *) movie;
}
```

The following function is the implementation of the `gf_isom_open_file` function. The beginning of this function is responsible for opening up a media source by the library. After the library allocates the necessary data structures for supporting the parsing of a container, a call to the `gf_isom_parse_movie_boxes` function at [2] will be made. As noted in the comment, this is where the actual parsing of the contents of the input will occur. The MPEG-4 container format is based on a type-length-value format in order to define each structure's boundaries. These type-length-value structures are commonly referred to as "atoms" or "boxes". These "atoms" may be recursively defined within the given container.

```
src/isomedia/isom_intern.c:809
GF_ISOFile *gf_isom_open_file(const char *fileName, GF_ISOOpenMode OpenMode, const char *tmp_dir)
{
    GF_Err e;
    u64 bytes;
    GF_ISOFile *mov = gf_isom_new_movie();
    if (!mov || !fileName) return NULL;

    mov->fileName = gf_strdup(fileName);
    mov->openMode = OpenMode;
...
    if ( (OpenMode == GF_ISOM_OPEN_READ) || (OpenMode == GF_ISOM_OPEN_READ_DUMP) || (OpenMode == GF_ISOM_OPEN_READ_EDIT) ) {
        if (OpenMode == GF_ISOM_OPEN_READ_EDIT) {
            mov->openMode = GF_ISOM_OPEN_READ_EDIT;

            // create a memory edit map in case we add samples, typically during import
            e = gf_isom_datamap_new(NULL, tmp_dir, GF_ISOM_DATA_MAP_WRITE, & mov->editFileMap);
            if (e) {
                gf_isom_set_last_error(NULL, e);
                gf_isom_delete_movie(mov);
                return NULL;
            }
        } else {
            mov->openMode = GF_ISOM_OPEN_READ;
        }
...
    }

    //OK, let's parse the movie...
    mov->LastError = gf_isom_parse_movie_boxes(mov, NULL, &bytes, 0);           // [2] parse each of the boxes within the file
```

The `gf_isom_parse_movie_boxes` function is simply a wrapper that will lock the input that is being parsed and then call into the actual parser. After performing the necessary locking around the input, the call at [3] to the `gf_isom_parse_movie_boxes_internal` function will then be called. This function will check the position that has been requested by the caller, use it to seek to the correct position in the input, and then proceed to parse the boxes associated with the container. As the MPEG-4 container format may be recursively defined, the function call at [4] to the `gf_isom_parse_root_box` is called to parse the root element of the movie container.

```
src/isomedia/isom_intern.c:764
GF_Err gf_isom_parse_movie_boxes(GF_ISOFile *mov, u32 *boxType, u64 *bytesMissing, Bool progressive_mode)
{
    GF_Err e;
    GF_Blob *blob = NULL;
...
    e = gf_isom_parse_movie_boxes_internal(mov, boxType, bytesMissing, progressive_mode);                      // [3] \ proceed to parse
the movie boxies
...
    return e;
}
\
src/isomedia/isom_intern.c:289
static GF_Err gf_isom_parse_movie_boxes_internal(GF_ISOFile *mov, u32 *boxType, u64 *bytesMissing, Bool progressive_mode)
{
    GF_Box *a;
    u64 totSize, mdat_end=0;
    GF_Err e = GF_OK;
...
    /*while we have some data, parse our boxes*/
    while (gf_bs_available(mov->movieFileMap->bs)) {
        *bytesMissing = 0;
...
        e = gf_isom_parse_root_box(&a, mov->movieFileMap->bs, boxType, bytesMissing, progressive_mode);         // [4] start by parsing the
root box
...
    }
...
    return GF_OK;
}
```

As prior mentioned, the atoms within an MPEG-4 container are recursively defined. The GPAC library chooses to implement its parser using a recursive algorithm. The primary function within the library's implementation is the `gf_isom_box_parse_ex` function. In the following code, the `gf_isom_parse_root_box` function is simply an entry-point to the recursive parser that lies within the implementation of the `gf_isom_box_parse_ex` function. At [5], the position of the input is set, and then the function call to `gf_isom_box_parse_ex` is used. The `gf_isom_box_parse_ex` function will start by reading the 32-bit size at [6] that is stored at the beginning of an atom's structure. Once the size has been read and checked, the next part of an atom's structure will be read. The next field in an atom is the type, or the FOURCC, which is then read into a local variable at [7]. In order to support larger atom sizes that may not fit entirely within 32-bits, the MPEG-4 standard allows for a 64-bit size. This is done by setting an atom's size to 1, at which point a 64-bit field containing the actual size will follow the FOURCC. At [8], the library will check if the size is 1 and then if so will proceed by reading the next 64-bit field from the atom, and then store it into the original size variable.

```
src/isomedia/box_funcs.c:33
GF_Err gf_isom_parse_root_box(GF_Box **outBox, GF_BitStream *bs, u32 *box_type, u64 *bytesExpected, Bool progressive_mode)
{
    GF_Err ret;
    u64 start;
    start = gf_bs_get_position(bs);
    ret = gf_isom_box_parse_ex(outBox, bs, 0, GF_TRUE);                 // [5] perform the actual parsing of the root box
...
    return ret;
}
\
src/isomedia/box_funcs.c:91
GF_Err gf_isom_box_parse_ex(GF_Box **outBox, GF_BitStream *bs, u32 parent_type, Bool is_root_box)
{
    u32 type, uuid_type, hdr_size, restore_type;
    u64 size, start, comp_start, payload_start, end;
    char uuid[16];
    GF_Err e;
    GF_BitStream *uncomp_bs = NULL;
    u8 *uncomp_data = NULL;
    u32 compressed_size=0;
    GF_Box *newBox;
    Bool skip_logs = (gf_bs_get_cookie(bs) & GF_ISOM_BS_COOKIE_NO_LOGS ) ? GF_TRUE : GF_FALSE;
    Bool is_special = GF_TRUE;

...
    size = (u64) gf_bs_read_u32(bs);                                    // [6] read the 32-bit size from the box or atom
    hdr_size = 4;
    /*fix for some boxes found in some old hinted files*/
    if ((size >= 2) && (size <= 4)) {
        size = 4;
        type = GF_ISOM_BOX_TYPE_VOID;
    } else {
        type = gf_bs_read_u32(bs);                                      // [7] read the 32-bit type or FOURCC from the atom
        hdr_size += 4;
...
    }
...
    //handle large box
    if (size == 1) {                                                    // [8] if the size is 1, then
        if (gf_bs_available(bs) < 8) {
            return GF_ISOM_INCOMPLETE_FILE;
        }
        size = gf_bs_read_u64(bs);                                      // [8] read the next 64-bit integer as the size
        hdr_size += 8;
    }
```

Continuing through the implementation of the `gf_isom_box_parse_ex` function, the function will use the type and size that was read to parse the contents of the atom. This parsed atom will then later be appended to a linked list so that the container may be processed by the library. Within this library, an atom is stored within a structure that is of the type `GF_Box` which is then casted into the actual atom type after it has been constructed. In the following code, the `GF_Box` is first constructed at [9] using the `gf_isom_box_new_ex` function with the type and the atom's parent type as its parameters. After the `GF_Box` has been constructed, it will then be passed to the `gf_isom_full_box_read` function call at [10] in order to read a specific header if the FOURCC requires it, and then to the `gf_isom_box_read` function call at [11] to actually parse the atom.

```
src/isomedia/box_funcs.c:217
    //some special boxes (references and track groups) are handled by a single generic box with an associated ref/group type
    if (parent_type && (parent_type == GF_ISOM_BOX_TYPE_TREF)) {
...
    } else {
        //OK, create the box based on the type
        is_special = GF_FALSE;
        newBox = gf_isom_box_new_ex(uuid_type ? uuid_type : type, parent_type, skip_logs, is_root_box);         // [9] construct space for a
Box (or atom)
        if (!newBox) return GF_OUT_OF_MEM;
    }

...
    newBox->size = size - hdr_size;

    e = gf_isom_full_box_read(newBox, bs);                                                                      // [10] parse an atom's
FullBox header
    if (!e) e = gf_isom_box_read(newBox, bs);                                                                   // [11] parse the contents
of the atom
    if (e) {
        if (gf_opts_get_bool("core", "no-check"))
            e = GF_OK;
    }
    newBox->size = size;
    end = gf_bs_get_position(bs);

...
    return e;
}
```

In order to determine how to construct the `GF_Box` type that is used during parsing, the current atom's type and its parent type are passed to the following function, `gf_isom_box_new_ex`. This function is responsible for looking up the atom's type inside a global array named `box_registry`, allocating the respective `GF_Box` structure, and initialize it with the necessary values prior to it being used. The global array, `box_registry` contains a list of all of the available atom types and is keyed by their FOURCC code. In order to find the index of the FOURCC for the atom being parsed, a call to the `get_box_reg_idx` function is made at [12] and given the FOURCC for the current atom along with the FOURCC of the current atom's parent. Inside the `get_box_reg_idx` function, the library will prepare to do a linear search through the global `box_registry` at [13] by first getting the total number of available FOURCC codes, and then converting the atom's parent FOURCC to a string. Afterwards these values will be used in the loop that follows in order to iterate through each defined element within the `box_registry`. At [14], the loop will then compare the FOURCC code that was passed as one of the function's parameters, and then check if the parent's FOURCC code was found within the current element. If these match the FOURCC provided in the function's parameters, then the index will be returned to the caller which will then use it at [15] to call the constructor that will allocate the real structure for the found FOURCC. Prior to returning to the caller, the `gf_isom_box_new_ex` function will update the `GF_Box` that was constructed with the registry that was used.

```
src/isomedia/box_funcs.c:1630
GF_Box *gf_isom_box_new_ex(u32 boxType, u32 parentType, Bool skip_logs, Bool is_root_box)
{
    GF_Box *a;
    s32 idx = get_box_reg_idx(boxType, parentType, 0);                                  // [12] figure out the index in the registry
    if (idx==0) {
\
src/isomedia/box_funcs.c:1589
static u32 get_box_reg_idx(u32 boxCode, u32 parent_type, u32 start_from)
{
    u32 i=0, count = gf_isom_get_num_supported_boxes();                                 // [13] get available number of boxes
    const char *parent_name = parent_type ? gf_4cc_to_str(parent_type) : NULL;          // [13] convert the parent type to a string

    if (!start_from) start_from = 1;

    for (i=start_from; i<count; i++) {                                                  // [13] enter loop
        u32 start_par_from;
        if (box_registry[i].box_4cc != boxCode)                                         // [14] compare the FOURCC code for the current
registry entry
            continue;

        if (!parent_type)
            return i;
        if (strstr(box_registry[i].parents_4cc, parent_name) != NULL)                   // [14] check that the parent's FOURCC is a
valid type
            return i;
        if (strstr(box_registry[i].parents_4cc, "*") != NULL)
            return i;

        if (strstr(box_registry[i].parents_4cc, "sample_entry") == NULL)
            continue;
...
    }
    return 0;
}
/
src/isomedia/box_funcs.c:1671
    a = box_registry[idx].new_fn();                                                     // [15] construct the GF_Box structure

    if (a) {
...
        a->registry = &box_registry[idx];                                               // [15] assign the registry that was used

        if ((a->type==GF_ISOM_BOX_TYPE_COLR) && (parentType==GF_ISOM_BOX_TYPE_JP2H)) {
            ((GF_ColourInformationBox *)a)->is_jp2 = GF_TRUE;
        }
    }
    return a;
}
```

Once the correct box structure has been constructed, then execution will then return back to the `gf_isom_box_parse_ex` function in order to actually use the `GF_Box`. At [10], the `gf_isom_full_box_read` function will be called to parse a particular category of FOURCC code. Upon entry into the `gf_isom_full_box_read` function, the library will check the `box_registry` entry for the FOURCC to see if it has a version associated with it. If so, the library will read a byte for the version and 3 bytes which maintain the flags for the currently read atom. After it has been read and the `GF_Box` structure has been updated, the library will return back to the `gf_isom_box_parse_ex` function and then pass the current `GF_Box` structure to the `gf_isom_box_read` function at [11]. This function is directly responsible for parsing the atom with the FOURCC that was previously looked up in the global `box_registry` array.

```
src/isomedia/box_funcs.c:262
    newBox->size = size - hdr_size;

    e = gf_isom_full_box_read(newBox, bs);                                                                  // [10] \ parse an atom's
FullBox header
    if (!e) e = gf_isom_box_read(newBox, bs);                                                               // [11] parse the contents
of the atom
    if (e) {
        if (gf_opts_get_bool("core", "no-check"))
            e = GF_OK;
    }
    newBox->size = size;
    end = gf_bs_get_position(bs);
\
src/isomedia/box_funcs.c:1927
static GF_Err gf_isom_full_box_read(GF_Box *ptr, GF_BitStream *bs)
{
    if (ptr->registry->max_version_plus_one) {
        GF_FullBox *self = (GF_FullBox *) ptr;
        ISOM_DECREASE_SIZE(ptr, 4)
        self->version = gf_bs_read_u8(bs);
        self->flags = gf_bs_read_u24(bs);
    }
    return GF_OK;
}
```

In the following code, the library will look at the `registry` field from the `GF_Box` that was passed as its parameter, and use it to access the entry that was discovered when searching the global `box_registry` array for the FOURCC code belonging to the atom read from the input. At [12], the `read_fn` field from the `box_registry` entry is dereferenced in order to continue to parse the contents of the atom that is being processed by the `gf_isom_box_parse_ex` function.

```
src/isomedia/box_funcs.c:1801
GF_Err gf_isom_box_read(GF_Box *a, GF_BitStream *bs)
{
    if (!a) return GF_BAD_PARAM;
    if (!a->registry) {
        GF_LOG(GF_LOG_ERROR, GF_LOG_CONTAINER, ("[iso file] Read invalid box type %s without registry\n", gf_4cc_to_str(a->type) ));
        return GF_ISOM_INVALID_FILE;
    }
    return a->registry->read_fn(a, bs);                    // [12] dispatch to the parser that was stored in the GF_Box registry field.
}
```

## CVE-2021-21853 - “name” decoder

In order to parse an atom with the “name” FOURCC code, the following function will be used. This function will first take the 64-bit size and truncate it to a 32-bit integer at [16]. After storing the size, the library will then add 1 to it and use it at [17] to allocate a buffer on the heap. If the 32-bit size is set to `UINT_MAX`, this addition will cause an integer overflow resulting in a zero-sized allocation being made. Later at [18], when the library uses the 32-bit length, the library will read data from the input into the undersized buffer resulting in a large heap-based buffer overflow.

```
src/isomedia/box_code_base.c:2607
GF_Err name_box_read(GF_Box *s, GF_BitStream *bs)
{
    u32 length;
    GF_NameBox *ptr = (GF_NameBox *)s;

    length = (u32) (ptr->size);                                         // [16] clamp 64-bit size to 32-bit integer
    ptr->string = (char*)gf_malloc(sizeof(char) * (length+1));          // [17] add 1 to length resulting in an integer overflow
    if (! ptr->string) return GF_OUT_OF_MEM;

    gf_bs_read_data(bs, ptr->string, length);                           // [18] read data from input into undersized buffer
    ptr->string[length] = 0;                                            // [18] write null byte outside bounds of zero-sized allocation
    return GF_OK;
}
```

### Crash Information

The provided proof-of-concept sets the atom’s size to 0x10000000f. The header of the “name” atom is 0x10 bytes in size, resulting in the 0xffffffff left for the atom’s contents. Adding 1 to this length results in the 0x100000000 length being used. This is then truncated to a 32-bit integer, resulting in a zero-sized allocation being made. Afterwards, the atom’s contents will read 0xffffffff bytes into the zero-sized buffer.

```
=================================================================
==182==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1200751 at pc 0x004dbfb7 bp 0xffdd66f8 sp 0xffdd62d8
WRITE of size 496 at 0xf1200751 thread T0
    #0 0x4dbfb6 in __asan_memcpy (/root/harness/parser32.asan+0x4dbfb6)
    #1 0xf44431d0 in gf_bs_read_data /root/src/utils/bitstream.c:672:5
    #2 0xf503fc48 in name_box_read /root/src/isomedia/box_code_base.c:2616:2
    #3 0xf5218097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5212c28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5268ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5268ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5267bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf527b3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf5294061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf39f5ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #12 0x4621e5 in _start (/root/harness/parser32.asan+0x4621e5)

0xf1200751 is located 0 bytes to the right of 1-byte region [0xf1200750,0xf1200751)
allocated by thread T0 here:
    #0 0x4dcb75 in malloc (/root/harness/parser32.asan+0x4dcb75)
    #1 0xf503fbe3 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf503fbe3 in name_box_read /root/src/isomedia/box_code_base.c:2613:23
    #3 0xf5218097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5212c28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5268ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5268ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5267bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf527b3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf5294061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf39f5ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/root/harness/parser32.asan+0x4dbfb6) in __asan_memcpy
Shadow bytes around the buggy address:
  0x3e240090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e2400e0: fa fa fa fa fa fa fa fa fa fa[01]fa fa fa fd fa
  0x3e2400f0: fa fa 00 fa fa fa 00 04 fa fa fa fa fa fa fa fa
  0x3e240100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240110: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240120: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240130: 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
  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
  Shadow gap:              cc
==182==ABORTING
```

### CVE-2021-21854 - “rtp “ decoder

When parsing an atom with the “rtp “ FOURCC code when its parent atom is using the “hnti” FOURCC code, the following function will be used. This function take the 64-bit size for the atom, and then truncate it to a 32-bit integer at [19]. This size will then be used to allocate memory at [20] after adding 1 to it. When the 32-bit size is set to `UINT_MAX`, this addition will result in an integer overflow. As this length is then passed to the `gf_malloc` function, a zero-sized buffer will be allocated. Later at [21], the library will then use the 32-bit length prior to the addition of 1, and then read data from the input into the undersized buffer. This and the null termination will write outside the bounds of the undersized buffer, corrupting memory due to a buffer overflow and a relative write.

```
src/isomedia/box_code_base.c:1940
GF_Err rtp_hnti_box_read(GF_Box *s, GF_BitStream *bs)
{
    u32 length;
    GF_RTPBox *ptr = (GF_RTPBox *)s;
    if (ptr == NULL) return GF_BAD_PARAM;

    ISOM_DECREASE_SIZE(ptr, 4)
    ptr->subType = gf_bs_read_u32(bs);

    length = (u32) (ptr->size);                                     // [19] clamp 64-bit size to 32-bit integer
    //sdp text has no delimiter !!!
    ptr->sdpText = (char*)gf_malloc(sizeof(char) * (length+1));     // [20] add 1 to length resulting in an integer overflow
    if (!ptr->sdpText) return GF_OUT_OF_MEM;

    gf_bs_read_data(bs, ptr->sdpText, length);                      // [21] read data from input into undersized buffer
    ptr->sdpText[length] = 0;                                       // [21] write null byte outside bounds of zero-sized allocation
    return GF_OK;
}
```

### Crash Information

The provided proof-of-concept sets the atom's 64-bit size to 0x100000013. After reading 0x10 bytes from the atom's header, a 32-bit integer is read for the "subType" field. This results in a length of 0xffffffff which is added to 1 when allocating the heap buffer. Due to the integer truncation, this results in a zero-sized buffer being allocated. Afterwards, the library will proceed by reading 0xffffffff bytes from the atom into the zero-sized buffer.

```
=================================================================
==192==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1200751 at pc 0x004dbfb7 bp 0xffb2e828 sp 0xffb2e408
WRITE of size 484 at 0xf1200751 thread T0
    #0 0x4dbfb6 in __asan_memcpy (/root/harness/parser32.asan+0x4dbfb6)
    #1 0xf43ff1d0 in gf_bs_read_data /root/src/utils/bitstream.c:672:5
    #2 0xf4fefb26 in rtp_hnti_box_read /root/src/isomedia/box_code_base.c:1954:2
    #3 0xf51d4097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf51cec28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf51d5418 in gf_isom_box_array_read_ex /root/src/isomedia/box_funcs.c:1705:7
    #6 0xf4feea62 in hnti_box_read /root/src/isomedia/box_code_base.c:1859:9
    #7 0xf51d4097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #8 0xf51cec28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #9 0xf5224ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #10 0xf5224ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #11 0xf5223bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #12 0xf52373f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #13 0xf5250061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #14 0x512f6a in main /root/harness/parser.c:50:13
    #15 0xf39b1ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #16 0x4621e5 in _start (/root/harness/parser32.asan+0x4621e5)

0xf1200751 is located 0 bytes to the right of 1-byte region [0xf1200750,0xf1200751)
allocated by thread T0 here:
    #0 0x4dcb75 in malloc (/root/harness/parser32.asan+0x4dcb75)
    #1 0xf4fefabe in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf4fefabe in rtp_hnti_box_read /root/src/isomedia/box_code_base.c:1951:24
    #3 0xf51d4097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf51cec28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf51d5418 in gf_isom_box_array_read_ex /root/src/isomedia/box_funcs.c:1705:7
    #6 0xf4feea62 in hnti_box_read /root/src/isomedia/box_code_base.c:1859:9
    #7 0xf51d4097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #8 0xf51cec28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #9 0xf5224ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #10 0xf5224ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #11 0xf5223bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #12 0xf52373f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #13 0xf5250061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #14 0x512f6a in main /root/harness/parser.c:50:13
    #15 0xf39b1ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/root/harness/parser32.asan+0x4dbfb6) in __asan_memcpy
Shadow bytes around the buggy address:
  0x3e240090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e2400e0: fa fa fa fa fa fa fa fa fa fa[01]fa fa fa fd fa
  0x3e2400f0: fa fa 00 fa fa fa 00 04 fa fa fa fa fa fa fa fa
  0x3e240100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240110: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240120: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240130: 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
  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
  Shadow gap:              cc
==192==ABORTING
```

### CVE-2021-21855 - "sdp " decoder

The following function is responsible for parsing atoms that use the "sdp " FOURCC code. This function will start by clamping the 64-bit size to a 32-bit integer and assigning it to the "length" variable at [22]. Afterwards at [23], the library will add 1 to the length and use it to allocate space on the heap. If this "length" is set to `UINT_MAX`, this addition will cause an integer overflow which when passed to the `gf_malloc` function can result in a zero-sized allocation being made. Afterwards at [24], the library will use the `UINT_MAX` length to read data from the atom into

the zero-sized allocation. This and the null-termination will write outside the bounds of the buffer resulting in the corruption of memory.

```
src/isomedia/box_code_base.c:1886
GF_Err sdp_box_read(GF_Box *s, GF_BitStream *bs)
{
    u32 length;
    GF_SDPBox *ptr = (GF_SDPBox *)s;
    if (ptr == NULL) return GF_BAD_PARAM;

    length = (u32) (ptr->size);                                 // [22] clamp 64-bit size to 32-bit integer
    //sdp text has no delimiter !!!
    ptr->sdpText = (char*)gf_malloc(sizeof(char) * (length+1)); // [23] add 1 to length causing an integer overflow
    if (!ptr->sdpText) return GF_OUT_OF_MEM;

    gf_bs_read_data(bs, ptr->sdpText, length);                  // [24] read data into zero-sized buffer
    ptr->sdpText[length] = 0;                                   // [24] write null termination outside bounds of zero-sized allocation
    return GF_OK;
}
```

### Crash Information

The provided proof-of-concept sets the atom's 64-bit length to 0x10000000f. After the atom's 0x10 byte header is subtracted, this results in the length being 0xffffffff. When the function allocates its memory, 1 is added to this value which when truncated will result in a zero-sized buffer being returned. Afterwards, the function will use the 0xffffffff length to read the atom into the zero-sized buffer.

```
=================================================================
==207==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1200751 at pc 0x004dbfb7 bp 0xffa69d58 sp 0xffa69938
WRITE of size 496 at 0xf1200751 thread T0
    #0 0x4dbfb6 in __asan_memcpy (/root/harness/parser32.asan+0x4dbfb6)
    #1 0xf44521d0 in gf_bs_read_data /root/src/utils/bitstream.c:672:5
    #2 0xf5041db8 in sdp_box_read /root/src/isomedia/box_code_base.c:1897:2
    #3 0xf5227097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5221c28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5277ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5277ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5276bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf528a3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf52a3061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3a04ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #12 0x4621e5 in _start (/root/harness/parser32.asan+0x4621e5)

0xf1200751 is located 0 bytes to the right of 1-byte region [0xf1200750,0xf1200751)
allocated by thread T0 here:
    #0 0x4dcb75 in malloc (/root/harness/parser32.asan+0x4dcb75)
    #1 0xf5041d53 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf5041d53 in sdp_box_read /root/src/isomedia/box_code_base.c:1894:24
    #3 0xf5227097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5221c28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5277ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5277ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5276bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf528a3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf52a3061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3a04ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/root/harness/parser32.asan+0x4dbfb6) in __asan_memcpy
Shadow bytes around the buggy address:
  0x3e240090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e2400e0: fa fa fa fa fa fa fa fa fa fa[01]fa fa fa fd fa
  0x3e2400f0: fa fa 00 fa fa fa 00 04 fa fa fa fa fa fa fa fa
  0x3e240100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240110: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240120: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240130: 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
  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
  Shadow gap:              cc
==207==ABORTING
```

### CVE-2021-21856 - "svhd" decoder

The following function is responsible for parsing atoms that use the "svhd" FOURCC code. This implementation is used to read a string from the atom's contents. At [25], the function will take the 64-bit atom size, add 1 to it, and then truncate it to 32-bits prior to passing it to the `gf_malloc` function. Due to the 32-bit truncation, if the atom size is set to `UINT_MAX`, this can result in a zero-sized buffer being returned by `gf_malloc`. After verifying the allocation was successful, the function will read the contents of the atom into the zero-sized array, and then null-terminate the string. Due to the size of the buffer being 0, this will write outside the bounds of the allocation resulting in a heap-based buffer overflow.

```
src/isomedia/box_code_base.c:12577
GF_Err svhd_box_read(GF_Box *s, GF_BitStream *bs)
{
    GF_SphericalVideoInfoBox *ptr = (GF_SphericalVideoInfoBox *)s;
    ptr->string = gf_malloc(sizeof(char) * ((u32) ptr->size+1));    // [25] add 1 and then clamp size to 32-bits
    if (!ptr->string) return GF_OUT_OF_MEM;
    gf_bs_read_data(bs, ptr->string, (u32) ptr->size);              // [26] read into undersized array
    ptr->string[ptr->size] = 0;                                     // [26] null-terminate string read from atom
    return GF_OK;
}
```

### Crash Information

In the provided proof-of-concept, the atom's 64-bit size is set to 0x100000013. After the 0x10 byte header is read, 4 bytes are read for the 8-bit "Version" and the 24-bit "Flags". This sets the length to 0xffffffff. When 1 is added to this length, this will result in 0x100000000 which when truncated will result in a zero-sized allocation being read. Afterwards, the parser will read 0xffffffff bytes from the atom into the zero-sized buffer causing a buffer overflow.

```
=================================================================
==257==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1100751 at pc 0x004dbfb7 bp 0xffa4fdf8 sp 0xffa4f9d8
WRITE of size 492 at 0xf1100751 thread T0
    #0 0x4dbfb6 in __asan_memcpy (/root/harness/parser32.asan+0x4dbfb6)
    #1 0xf43cb1d0 in gf_bs_read_data /root/src/utils/bitstream.c:672:5
    #2 0xf50cbf9b in svhd_box_read /root/src/isomedia/box_code_base.c:12582:2
    #3 0xf51a0097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf519ad5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf51f0ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf51f0ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf51efbc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf52033f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf521c061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf397dee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #12 0x4621e5 in _start (/root/harness/parser32.asan+0x4621e5)

0xf1100751 is located 0 bytes to the right of 1-byte region [0xf1100750,0xf1100751)
allocated by thread T0 here:
    #0 0x4dcb75 in malloc (/root/harness/parser32.asan+0x4dcb75)
    #1 0xf50cbed8 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf50cbed8 in svhd_box_read /root/src/isomedia/box_code_base.c:12580:16
    #3 0xf51a0097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf519ad5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf51f0ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf51f0ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf51efbc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf52033f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf521c061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf397dee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/root/harness/parser32.asan+0x4dbfb6) in __asan_memcpy
Shadow bytes around the buggy address:
  0x3e220090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2200a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2200b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2200c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2200d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e2200e0: fa fa fa fa fa fa fa fa fa fa[01]fa fa fa fd fa
  0x3e2200f0: fa fa 00 fa fa fa 00 04 fa fa fa fa fa fa fa fa
  0x3e220100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e220110: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e220120: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e220130: 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
  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
  Shadow gap:              cc
==257==ABORTING
```

### CVE-2021-21857 - "txtc" decoder

The implementation of the parser for the "txtc" FOURCC code is responsible for reading a string from the atom, and then null-terminating it. At [27], the function will take the 32-bit size and add 1 to it before truncating it to a 32-bit integer and then passing it as a parameter to the `gf_malloc` function. If the atom size is set to `UINT_MAX`, this addition when truncated to a 32-bit integer can result in an integer overflow causing the allocation to return a zero-sized buffer. Afterwards at [29], the function will use the original non-truncated atom size to read the contents of the atom into the zero-sized buffer. This will then cause a heap-based buffer overflow when reading the string from the atom, and then null-terminating it.

```
src/isomedia/box_code_base.c:8518
GF_Err txtc_box_read(GF_Box *s, GF_BitStream *bs)
{
    GF_TextConfigBox *ptr = (GF_TextConfigBox*)s;
    ptr->config = (char *)gf_malloc(sizeof(char)*((u32) ptr->size+1));      // [27] add 1 to atom size and truncate to 32-bits
    if (!ptr->config) return GF_OUT_OF_MEM;
    gf_bs_read_data(bs, ptr->config, (u32) ptr->size);                      // [28] read into buffer using original 32-bit size
    ptr->config[ptr->size] = 0;                                             // [29] null-terminate allocated buffer
    return GF_OK;
}
```

### Crash Information

The provided proof-of-concept sets the atom's 64-bit size to 0x100000013. After subtracting 0x10 bytes for the header, 4 bytes are read for the 8-bit "Version" and the 24-bit "Flags". This results in a length of 0xffffffff. The allocation adds 1 to this length resulting in 0x100000000, which when truncated results in a zero-sized allocation being made. Afterwards, the function will read 0xffffffff bytes into this zero-sized buffer causing the buffer overflow.

```
=================================================================
==277==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1200751 at pc 0x004dbfb7 bp 0xffd37778 sp 0xffd37358
WRITE of size 492 at 0xf1200751 thread T0
    #0 0x4dbfb6 in __asan_memcpy (/root/harness/parser32.asan+0x4dbfb6)
    #1 0xf44a21d0 in gf_bs_read_data /root/src/utils/bitstream.c:672:5
    #2 0xf513097b in txtc_box_read /root/src/isomedia/box_code_base.c:8523:2
    #3 0xf5277097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5271d5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf52c7ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf52c7ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf52c6bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf52da3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf52f3061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3a54ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #12 0x4621e5 in _start (/root/harness/parser32.asan+0x4621e5)

0xf1200751 is located 0 bytes to the right of 1-byte region [0xf1200750,0xf1200751)
allocated by thread T0 here:
    #0 0x4dcb75 in malloc (/root/harness/parser32.asan+0x4dcb75)
    #1 0xf51308b8 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf51308b8 in txtc_box_read /root/src/isomedia/box_code_base.c:8521:24
    #3 0xf5277097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5271d5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf52c7ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf52c7ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf52c6bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf52da3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf52f3061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3a54ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/root/harness/parser32.asan+0x4dbfb6) in __asan_memcpy
Shadow bytes around the buggy address:
  0x3e240090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e2400d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e2400e0: fa fa fa fa fa fa fa fa fa fa[01]fa fa fa fd fa
  0x3e2400f0: fa fa 00 fa fa fa 00 04 fa fa fa fa fa fa fa fa
  0x3e240100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240110: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240120: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e240130: 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
  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
  Shadow gap:              cc
==277==ABORTING
```

### CVE-2021-21858 - "url " decoder

The following function is the parser used by the library in order to read the contents of an atom using the "url " FOURCC code. This function will first check that the 64-atom size is non-zero, and then at [30] will truncate the atom size to 32-bits when allocating space on the heap. After reading the contents of the atom using the 32-bit truncated size, the function will then use the original 64-bit size when accessing the allocated heap buffer. Due to the allocated size being truncated to 32-bits, this can result in an out-of-bounds read on 64-bit platforms.

```
src/isomedia/box_code_base.c:575
GF_Err url_box_read(GF_Box *s, GF_BitStream *bs)
{
    GF_DataEntryURLBox *ptr = (GF_DataEntryURLBox *)s;

    if (ptr->size) {
        ptr->location = (char*)gf_malloc((u32) ptr->size);          // [30] truncate 64-bit atom size to 32-bits
        if (! ptr->location) return GF_OUT_OF_MEM;
        gf_bs_read_data(bs, ptr->location, (u32)ptr->size);
        if (ptr->location[ptr->size-1]) {                           // [31] use non-truncated 64-bit atom size to access array
            GF_LOG(GF_LOG_ERROR, GF_LOG_CONTAINER, ("[iso file] url box location is not 0-terminated\n" ));
            return GF_ISOM_INVALID_FILE;
        }
    }
    return GF_OK;
}
```

### Crash Information

The provided proof-of-concept sets the atom's 64-bit length to 0x100000014. After reading 0x10 bytes from the header, followed by 8-bits for the "Version", and 24-bits for the "Flags", This will result in the length 0x100000000 being used for the allocation. When this value is truncated to 32-bits, this will result in a zero-sized allocation being made upon which the function will start by checking if the byte at the non-truncated 64-bit size has been set.

```
=================================================================
==1319==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf200074f at pc 0xf58ed7a1 bp 0xffd6e6d8 sp 0xffd6e6d0
READ of size 1 at 0xf200074f thread T0
    #0 0xf58ed7a0 in url_box_read /root/src/isomedia/box_code_base.c:583:7
    #1 0xf5a8b4e4 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #2 0xf5a8b4e4 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #3 0xf5ad8638 in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #4 0xf5ad8638 in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #5 0xf5ad8638 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #6 0xf5ae76f2 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #7 0xf5af9bf1 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #8 0x512f6a in main /root/harness/parser.c:50:13
    #9 0xf4882ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #10 0x4621e5 in _start (/root/harness/parser32.asan+0x4621e5)

0xf200074f is located 1 bytes to the left of 1-byte region [0xf2000750,0xf2000751)
allocated by thread T0 here:
    #0 0x4dcb75 in malloc (/root/harness/parser32.asan+0x4dcb75)
    #1 0xf58ed1e7 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf58ed1e7 in url_box_read /root/src/isomedia/box_code_base.c:580:26
    #3 0xf5a8b4e4 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5a8b4e4 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5ad8638 in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5ad8638 in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5ad8638 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf5ae76f2 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf5af9bf1 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf4882ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow /root/src/isomedia/box_code_base.c:583:7 in url_box_read
Shadow bytes around the buggy address:
  0x3e400090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e4000a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e4000b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e4000c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e4000d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e4000e0: fa fa fa fa fa fa fa fa fa[fa]01 fa fa fa fd fa
  0x3e4000f0: fa fa 00 fa fa fa 00 04 fa fa fa fa fa fa fa fa
  0x3e400100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e400110: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e400120: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e400130: 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
  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
  Shadow gap:              cc
==1319==ABORTING
```

## Timeline

2021-06-24 - Vendor Disclosure

2021-08-11 - Vendor Patched

2021-08-16 - Public Release

## CREDIT

Discovered by a member of Cisco Talos.

---