Talos Vulnerability Report

TALOS-2021-1298

GPAC Project Advanced Content MPEG-4 Decoding multiple integer truncation vulnerabilities

AUGUST 16, 2021

CVE NUMBER

CVE-2021-21859,CVE-2021-21860,CVE-2021-21861,CVE-2021-21862

Summary

Multiple exploitable integer truncation 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 improper memory allocation 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-2021-21859 - “stri” decoder

The `stri_box_read` function as follows is used when processing atoms using the “stri” FOURCC code. This function will first take the 64-bit atom size, and divide it by four to determine the number of attributes that are contained by the atom. After determining the number of attributes, at [14] the library will pass the count to the `GF_SAFE_ALLOC_N` function which will multiply the count by the size of the `u32` type. Due to an integer overflow, this can result in a zero-sized allocation being made. Later, the function will use the number of attributes to write into the zero-sized buffer resulting in a heap-based overflow and corrupting memory.

```
src/isomedia/box_code_base.c:6772
GF_Err stri_box_read(GF_Box *s, GF_BitStream *bs)
{
    size_t i;
    GF_SubTrackInformationBox *ptr = (GF_SubTrackInformationBox *)s;
    ISOM_DECREASE_SIZE(ptr, 8)
...
    ptr->attribute_count = ptr->size / 4;                                       // [13] assign number of attributes using atom size
    GF_SAFE_ALLOC_N(ptr->attribute_list, (size_t)ptr->attribute_count, u32);    // [14] | allocate space for number of attributes
    if (!ptr->attribute_list) return GF_OUT_OF_MEM;
    for (i = 0; i < ptr->attribute_count; i++) {
        ISOM_DECREASE_SIZE(ptr, 4)
        ptr->attribute_list[i] = gf_bs_read_u32(bs);                            // [15] read contents of atom into undersized buffer
    }
    return GF_OK;
}
|
include/gpac/tools.h:242
#define GF_SAFE_ALLOC_N(__ptr, __n, __struct) {\
        (__ptr) = (__struct *) gf_malloc( __n * sizeof(__struct));\             // [14] multiply fields in order to make allocation
        if (__ptr) {\
            memset((void *) (__ptr), 0, __n * sizeof(__struct));\
        }\
    }
```

Crash Information

The provided proof-of-concept specifies the 64-bit atom size as 0x10000001c. As 0x10 bytes are allocated towards the header, 4 bytes are removed for the “Version” and “Flags”, leaving 8 bytes for the other fields at the beginning of the atom. After the function subtracts the 8-bytes for the atom’s other fields, this will result in 0x100000000 which when divided by 4 to calculate the number of attributes will result in 0x40000000. When this result is multiplied by the size of a `u32` type, and then clamped, this will result in a zero-sized buffer being allocated. The function will then read 32-bit integers for each attribute into the zero-sized buffer.

```
=================================================================
==227==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1100750 at pc 0xf502801b bp 0xffe7db48 sp 0xffe7db40
WRITE of size 4 at 0xf1100750 thread T0
    #0 0xf502801a in stri_box_read /root/src/isomedia/box_code_base.c:6785:26
    #1 0xf5189097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #2 0xf5183d5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #3 0xf51d9ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #4 0xf51d9ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #5 0xf51d8bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #6 0xf51ec3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #7 0xf5205061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #8 0x512f6a in main /root/harness/parser.c:50:13
    #9 0xf3966ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #10 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 0xf5027322 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf5027322 in stri_box_read /root/src/isomedia/box_code_base.c:6781:2
    #3 0xf5189097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5183d5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf51d9ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf51d9ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf51d8bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf51ec3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf5205061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3966ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow /root/src/isomedia/box_code_base.c:6785:26 in stri_box_read
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
==227==ABORTING
```

CVE-2021-21860 - “trik” decoder

The FOURCC code, “trik”, is parsed by the following function within the library. At [16], the library will take the 64-bit atom size and clamp it to a 32-bit integer. Once the integer has been truncated to 32-bits, the function will use the 32-bit integer and multiply it by the size of the `GF_TrickPlayBoxEntry`. This product of a 32-bit integer with the size of the structure can result in a size larger than 32-bits which can cause an integer overflow. Thus, at [17] this can result in an undersized allocation when the size is used as the parameter `gf_malloc`. At [18], the function will then read a 2-bit integer followed by a 6-bit integer into each element of the array. Due to the integer overflow, the array can be undersized causing the population of each entry to overflow the boundaries of the heap allocation. This can cause memory corruption and is known as a heap-based buffer overflow.

```
src/isomedia/box_code_base.c:11121
GF_Err trik_box_read(GF_Box *s,GF_BitStream *bs)
{
    u32 i;
    GF_TrickPlayBox *ptr = (GF_TrickPlayBox *) s;
    ptr->entry_count = (u32) ptr->size;                                                                 // [16] clamp size to a 32-bit
integer
    ptr->entries = (GF_TrickPlayBoxEntry *) gf_malloc(ptr->entry_count * sizeof(GF_TrickPlayBoxEntry) );    // [17] allocate space for each
entry
    if (!ptr->entries) return GF_OUT_OF_MEM;

    for (i=0; i< ptr->entry_count; i++) {
        ptr->entries[i].pic_type = gf_bs_read_int(bs, 2);                                               // [18] read contents of atom
into undersized array
        ptr->entries[i].dependency_level = gf_bs_read_int(bs, 6);
    }
    return GF_OK;
}
```

Crash Information

The provided proof-of-concept sets the atom’s size to 0x8000000c. When this size is multiplied by the size of the `GF_TrickPlayBoxEntry` type, this will result in the size 0x100000018 which when truncated will result in a 0x18 byte allocation. The parser will continue to read the atom by consuming two 8-bit integers, and then writing them into the under-sized array.

```
=================================================================
==267==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1200751 at pc 0xf5132481 bp 0xff935708 sp 0xff935700
WRITE of size 1 at 0xf1200751 thread T0
    #0 0xf5132480 in trik_box_read /root/src/isomedia/box_code_base.c:11131:36
    #1 0xf5230097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #2 0xf522ad5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #3 0xf5280ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #4 0xf5280ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #5 0xf527fbc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #6 0xf52933f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #7 0xf52ac061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #8 0x512f6a in main /root/harness/parser.c:50:13
    #9 0xf3a0dee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #10 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 0xf5131e83 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf5131e83 in trik_box_read /root/src/isomedia/box_code_base.c:11126:42
    #3 0xf5230097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf522ad5d in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5280ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5280ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf527fbc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf52933f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf52ac061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3a0dee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow /root/src/isomedia/box_code_base.c:11131:36 in trik_box_read
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
==267==ABORTING
```

CVE-2021-21861 - “hdlr” decoder

When processing the “hdlr” FOURCC code, the following function is used to read the atom from the input. In this function at [19], the library will first clamp the 64-bit size into a 32-bit integer. This integer truncation can result in a zero-sized allocation being made. After verifying that the byte relative to the zero-sized “nameUTF8” allocation is set, the library will then take the 64-bit size and subtract 1 before truncating it and using it as a parameter to the `memmove` function. At [20], depending on a 32-bit or 64-bit platform, the library will write out of bounds before or after the “nameUTF8” buffer resulting in a memory corruption.

```
src/isomedia/box_code_base.c:1640
GF_Err hdlr_box_read(GF_Box *s, GF_BitStream *bs)
{
    u64 cookie;
    GF_HandlerBox *ptr = (GF_HandlerBox *)s;
...
    if (ptr->size) {
        ptr->nameUTF8 = (char*)gf_malloc((u32) ptr->size);                              // [19] use size to allocate space
        if (!ptr->nameUTF8) return GF_OUT_OF_MEM;
        gf_bs_read_data(bs, ptr->nameUTF8, (u32) ptr->size);
...
        if (ptr->nameUTF8[ptr->size-1]) {
            memmove(ptr->nameUTF8, ptr->nameUTF8+1, sizeof(char) * (u32) (ptr->size-1) );   // [20] copy data from input into buffer
            ptr->nameUTF8[ptr->size-1] = 0;                                                 // [20] write 1-byte in front of buffer
            ptr->store_counted_string = GF_TRUE;
        }
    }
    return GF_OK;
}
```

Crash Information

The provided proof-of-concept sets the 64-bit size of the “hdlr” atom to 0x100000028. The first 0x10 bytes consist of the atom’s header. After subtracting 4-bytes for the “FullBox” header consisting of the 8-bit “Version”, and 24-bit “Flags” fields, 0x14 more bytes will be read for the “hdlr” atom’s header. This results in the 64-bit size being set to 0x100000000 which when clamped to 32-bits will result in a zero-sized allocation. Afterwards, the function checks if the byte in front of the buffer is set. If so, then the function will read 0xffffffff bytes into the zero-sized buffer and then null-terminate it.

```
=================================================================
==1326==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf200074f at pc 0xf5836b14 bp 0xfffa8598 sp 0xfffa8590
READ of size 1 at 0xf200074f thread T0
    #0 0xf5836b13 in hdlr_box_read /root/src/isomedia/box_code_base.c:1666:7
    #1 0xf59c54e4 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #2 0xf59c54e4 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #3 0xf5a12638 in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #4 0xf5a12638 in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #5 0xf5a12638 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #6 0xf5a216f2 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #7 0xf5a33bf1 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #8 0x512f6a in main /root/harness/parser.c:50:13
    #9 0xf47bcee4 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 0xf5835f53 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf5835f53 in hdlr_box_read /root/src/isomedia/box_code_base.c:1658:26
    #3 0xf59c54e4 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf59c54e4 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5a12638 in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5a12638 in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5a12638 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf5a216f2 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf5a33bf1 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf47bcee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)

SUMMARY: AddressSanitizer: heap-buffer-overflow /root/src/isomedia/box_code_base.c:1666:7 in hdlr_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
==1326==ABORTING
```

CVE-2021-21862 - “Xtra” decoder

The implementation of the parser used for the “Xtra” FOURCC code is handled by the following function. First the function will read a 32-bit signed integer for the tag size, and then at [21] will read a 32-bit unsigned integer in order to determine the length of the name within the atom. At [22], the sum of the unsigned 32-bit name size and the value 1 will be used to allocate a buffer for the string that is to be read. Due to the addition of a 32-bit unsigned integer and any number other than 0 resulting in a value that is larger than 32-bits, this addition can result in an integer overflow. If the 32-bit size is set to `UINT_MAX`, the addition will result in the `gf_malloc` function returning a zero-sized buffer. Afterwards at [23], the original name size will be used to read a string from the atom into a zero-sized buffer and then null-terminate it. As the buffer that was returned is zero-sized, this will result in a large buffer overflow, followed by a relative write for the null-termination.

```
src/isomedia/box_code_base.c:12426
GF_Err xtra_box_read(GF_Box *s, GF_BitStream *bs)
{
    GF_XtraBox *ptr = (GF_XtraBox *)s;
    while (ptr->size) {
        GF_XtraTag *tag;
        u32 prop_type = 0;

        char *data=NULL, *data2=NULL;
        ISOM_DECREASE_SIZE_NO_ERR(ptr, 18)
        s32 tag_size = gf_bs_read_u32(bs);
        u32 name_size = gf_bs_read_u32(bs);                // [21] read 32-bit length of name
        tag_size -= 8;

        ISOM_DECREASE_SIZE_NO_ERR(ptr, name_size)
        data = gf_malloc(sizeof(char) * (name_size+1));    // [22] add 1 to length of name for allocation
        gf_bs_read_data(bs, data, name_size);              // [23] read string into undersized buffer
        data[name_size] = 0;                               // [23] null-terminate string
...
    return GF_OK;
}
```

Crash Information

The provided proof-of-concept sets the “name_size” field to 0xffffffff. When the function performs its allocation, this length will have the value 1 added to it, resulting in the size 0x100000000 being used. When this value is truncated, a zero-sized allocation will be made. Afterwards, the parser will continue to read the contents of the atom as specified by the “name_size” field directly into the zero-sized buffer.

```
=================================================================
==286==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf1200731 at pc 0x004dbfb7 bp 0xff9fbf58 sp 0xff9fbb38
WRITE of size 488 at 0xf1200731 thread T0
    #0 0x4dbfb6 in __asan_memcpy (/root/harness/parser32.asan+0x4dbfb6)
    #1 0xf44541d0 in gf_bs_read_data /root/src/utils/bitstream.c:672:5
    #2 0xf5151e7d in xtra_box_read /root/src/isomedia/box_code_base.c:12441:3
    #3 0xf5229097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5223c28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5279ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5279ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5278bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf528c3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf52a5061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3a06ee4 in __libc_start_main (/lib32/libc.so.6+0x1eee4)
    #12 0x4621e5 in _start (/root/harness/parser32.asan+0x4621e5)

0xf1200731 is located 0 bytes to the right of 1-byte region [0xf1200730,0xf1200731)
allocated by thread T0 here:
    #0 0x4dcb75 in malloc (/root/harness/parser32.asan+0x4dcb75)
    #1 0xf5151e68 in gf_malloc /root/src/utils/alloc.c:150:9
    #2 0xf5151e68 in xtra_box_read /root/src/isomedia/box_code_base.c:12440:10
    #3 0xf5229097 in gf_isom_box_read /root/src/isomedia/box_funcs.c:1808:9
    #4 0xf5223c28 in gf_isom_box_parse_ex /root/src/isomedia/box_funcs.c:265:14
    #5 0xf5279ced in gf_isom_parse_root_box /root/src/isomedia/box_funcs.c:38:8
    #6 0xf5279ced in gf_isom_parse_movie_boxes_internal /root/src/isomedia/isom_intern.c:318:7
    #7 0xf5278bc0 in gf_isom_parse_movie_boxes /root/src/isomedia/isom_intern.c:777:6
    #8 0xf528c3f7 in gf_isom_open_file /root/src/isomedia/isom_intern.c:897:19
    #9 0xf52a5061 in gf_isom_open /root/src/isomedia/isom_read.c:509:11
    #10 0x512f6a in main /root/harness/parser.c:50:13
    #11 0xf3a06ee4 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[01]fa fa fa 00 04 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
==286==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.