Talos Vulnerability Report

TALOS-2020-1212

tinyobjloader LoadObj improper array index validation vulnerability

JULY 30, 2021

CVE NUMBER

CVE-2020-28589

Summary

An improper array index validation vulnerability exists in the LoadObj functionality of tinyobjloader v2.0-rc1 and tinyobjloader development commit 79d4421. A specially crafted file could lead to code execution. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

tinyobjloader development commit 79d4421
tinyobjloader v2.0-rc1

Product URLs

https://github.com/tinyobjloader/tinyobjloader

CVSSv3 Score

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

CWE

CWE-129 - Improper Validation of Array Index

Details

Tinyobjloader is an extremely portable wavefront obj loader library used in multiple graphics-rendering projects.

For the purposes of the writeup, we examine the `tiny_obj_loader.h` file provided by `tinyobjectloader`. While there are other methods of loading and utilizing `tinyobjectloader`, the vulnerability lies within `tiny_obj_loader.h`.

To start, let us examine the beginning of the `LoadObj()` function. For context, `LoadObj()` is one of the main APIs that `tinyobjectloader` provides, as it takes in a stream of data and populates a vector with the parsed data:

```
bool LoadObj(attrib_t *attrib, std::vector<shape_t> *shapes,       // [1]
         std::vector<material_t> *materials, std::string *warn,
         std::string *err, std::istream *inStream,
         MaterialReader *readMatFn /*= NULL*/, bool triangulate,
         bool default_vcols_fallback) {
  std::stringstream errss;

  std::vector<real_t> v;
  std::vector<real_t> vn;
  std::vector<real_t> vt;
  std::vector<real_t> vc;
  std::vector<tag_t> tags;
  std::vector<face_t> faceGroup;
  std::vector<int> lineGroup;
  std::string name;

  // material
  std::map<std::string, int> material_map;
  int material = -1;

  // smoothing group id
  unsigned int current_smoothing_id =
      0;  // Initial value. 0 means no smoothing.

  int greatest_v_idx = -1;
  int greatest_vn_idx = -1;
  int greatest_vt_idx = -1;

  shape_t shape;                   // [2]

  bool found_all_colors = true;
```

At [1] the `std::vector<shape_t> *shapes` parameter holds the parsed shapes, which are filled by `LoadObj`. A quick display of the object prototype follows:

```
typedef struct {
 std::string name;
 mesh_t mesh;
} shape_t;
```

In order to parse the various elements in the object and fill the `shapes` vector, `LoadObj` uses the function `exportGroupsToShape`:

```
static bool exportGroupsToShape(shape_t *shape, const PrimGroup &prim_group,
                                const std::vector<tag_t> &tags,
                                const int material_id, const std::string &name,
                                bool triangulate,
                                const std::vector<real_t> &v) {
  if (prim_group.IsEmpty()) {
    return false;
  }

  shape->name = name;

  // polygon
  if (!prim_group.faceGroup.empty()) {
    // Flatten vertices and indices
    for (size_t i = 0; i < prim_group.faceGroup.size(); i++) {  // [5]
      const face_t &face = prim_group.faceGroup[i];

      size_t npolys = face.vertex_indices.size();   // [3]

      if (npolys < 3) {
        // Face must have 3+ vertices.
        continue;
      }

      vertex_index_t i0 = face.vertex_indices[0];
      vertex_index_t i1(-1);
      vertex_index_t i2 = face.vertex_indices[1];

      if (triangulate) {
        // find the two axes to work in
        size_t axes[2] = {1, 2};
        for (size_t k = 0; k < npolys; ++k) {
          i0 = face.vertex_indices[(k + 0) % npolys];
          i1 = face.vertex_indices[(k + 1) % npolys];
          i2 = face.vertex_indices[(k + 2) % npolys];
          size_t vi0 = size_t(i0.v_idx);
          size_t vi1 = size_t(i1.v_idx);
          size_t vi2 = size_t(i2.v_idx);

          if (((3 * vi0 + 2) >= v.size()) || ((3 * vi1 + 2) >= v.size()) ||   // [4]
              ((3 * vi2 + 2) >= v.size())) {
            // Invalid triangle.
            // FIXME(syoyo): Is it ok to simply skip this invalid triangle?
            continue;   // [5]
          }
...
```

For each face in the group, the number of vertices is extracted at [3]. Then, when triangulating, the vertices are validated at [4], by checking that they're smaller than `v.size()`. However, the validation simply continues the for loop to the next face group [5] and the function continues its execution. Note that similar patterns that validate the vertices but continue on anyway, can be found in this same function, hence they likely have the same issue.

Later on, in the same function:

```
{
  index_t idx0, idx1, idx2;
  idx0.vertex_index = ind[0].v_idx;
  idx0.normal_index = ind[0].vn_idx;
  idx0.texcoord_index = ind[0].vt_idx;
  idx1.vertex_index = ind[1].v_idx;
  idx1.normal_index = ind[1].vn_idx;
  idx1.texcoord_index = ind[1].vt_idx;
  idx2.vertex_index = ind[2].v_idx;
  idx2.normal_index = ind[2].vn_idx;
  idx2.texcoord_index = ind[2].vt_idx;

  shape->mesh.indices.push_back(idx0);  // [6]
  shape->mesh.indices.push_back(idx1);  // [6]
  shape->mesh.indices.push_back(idx2);  // [6]

  shape->mesh.num_face_vertices.push_back(3);
  shape->mesh.material_ids.push_back(material_id);
  shape->mesh.smoothing_group_ids.push_back(face.smoothing_group_id);
}
```

Despite the previous validations, invalid indexes are inserted into the shape at [6]. An obvious example of an invalid index in this case is any negative index.

The vertices extracted end up in the `shape_t` structure, and are supposed to later be used to index the `attrib.vertices` array. In practice, let's see a sample code, as proposed in the "README.md" of tinyobjloader's repository:

```
...
bool ret = tinyobj::LoadObj(&attrib, &shapes, &materials, &warn, &err, inputfile.c_str());

if (!warn.empty()) {
  std::cout << warn << std::endl;
}

if (!err.empty()) {
  std::cerr << err << std::endl;
}

if (!ret) {
  exit(1);
}

// Loop over shapes
for (size_t s = 0; s < shapes.size(); s++) {
  // Loop over faces(polygon)
  size_t index_offset = 0;
  for (size_t f = 0; f < shapes[s].mesh.num_face_vertices.size(); f++) {
    size_t fv = size_t(shapes[s].mesh.num_face_vertices[f]);

    // Loop over vertices in the face.
    for (size_t v = 0; v < fv; v++) {
      // access to vertex
      tinyobj::index_t idx = shapes[s].mesh.indices[index_offset + v];

      tinyobj::real_t vx = attrib.vertices[3*size_t(idx.vertex_index)+0];  // [7]
      tinyobj::real_t vy = attrib.vertices[3*size_t(idx.vertex_index)+1];
      tinyobj::real_t vz = attrib.vertices[3*size_t(idx.vertex_index)+2];

      // Check if `normal_index` is zero or positive. negative = no normal data
      if (idx.normal_index >= 0) {
        tinyobj::real_t nx = attrib.normals[3*size_t(idx.normal_index)+0];
        tinyobj::real_t ny = attrib.normals[3*size_t(idx.normal_index)+1];
        tinyobj::real_t nz = attrib.normals[3*size_t(idx.normal_index)+2];
      }

      // Check if `texcoord_index` is zero or positive. negative = no texcoord data
      if (idx.texcoord_index >= 0) {
        tinyobj::real_t tx = attrib.texcoords[2*size_t(idx.texcoord_index)+0];
        tinyobj::real_t ty = attrib.texcoords[2*size_t(idx.texcoord_index)+1];
      }
      // Optional: vertex colors
      // tinyobj::real_t red   = attrib.colors[3*size_t(idx.vertex_index)+0];
      // tinyobj::real_t green = attrib.colors[3*size_t(idx.vertex_index)+1];
      // tinyobj::real_t blue  = attrib.colors[3*size_t(idx.vertex_index)+2];
    }
    index_offset += fv;

    // per-face material
    shapes[s].mesh.material_ids[f];
  }
}
```

In the example above, a negative `idx.vertex_index` value would cause the `attrib.vertices` object to be accessed out-of-bounds [7]. In the worst case, this could lead to code execution, depending on how the values end up being used by the program that utilizes the tinyobjloader library.

Crash Information

```
index_offset 0, cur fv: 3
index -4
AddressSanitizer:DEADLYSIGNAL
=================================================================
==2433561==ERROR: AddressSanitizer: SEGV on unknown address (pc 0x564751426388 bp 0x7ffe1d31f4b0 sp 0x7ffe1d31ef80 T0)
==2433561==The signal is caused by a READ memory access.
==2433561==Hint: this fault was caused by a dereference of a high value address (see register values below).  Dissassemble the provided pc
to learn which register was used.
    #0 0x564751426388 in main ./tinyobj/tinyobjloader/loader_example.cc:47:30
    #1 0x7f3eb1076b24 in __libc_start_main (/usr/lib/libc.so.6+0x27b24)
    #2 0x56475132274d in _start (./tinyobj/tinyobjloader/test_loader+0x5174d)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV ./tinyobj/tinyobjloader/loader_example.cc:47:30 in main
==2433561==ABORTING
```

Timeline

2020-12-01 - Vendor Disclosure

2021-02-19 - Vendor requests new poc; unable to reproduce issue

2021-03-24 - Talos follow up on status; Vendor advised still unable to reproduce issue

2021-04-05 - Talos provided new poc, revised advisory, and reset 90 day disclosure deadline

2021-05-13 - Talos follow up w/ vendor on status

2021-06-02 - Talos follow up re: 90 day timeline

2021-07-30 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT NEXT REPORT

TALOS-2020-1142 TALOS-2021-1293