Talos Vulnerability Report

TALOS-2020-1222

Prusa Research PrusaSlicer Admesh stl_fix_normal_directions() out-of-bounds write vulnerability

APRIL 21, 2021

CVE NUMBER

CVE-2020-28598

Summary

An out-of-bounds write vulnerability exists in the Admesh stl_fix_normal_directions() functionality of Prusa Research PrusaSlicer 2.2.0 and Master (commit 4b040b856). A specially crafted AMF file can lead to code execution. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

Prusa Research PrusaSlicer 2.2.0
Prusa Research PrusaSlicer Master (commit 4b040b856)

Product URLs

https://www.prusa3d.com/prusaslicer/

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-122 - Heap-based Buffer Overflow

Details

Prusa Slicer is an open-source 3-D printer slicing program forked off Slic3r that can convert various 3-D model file formats and can output corresponding 3-D printer-readable Gcode.

After normalizing a given `.stl`, `.obj`, `.3mf`, `.amf`, `.amf.xml`, `.3mf.xml` or `.prusa` file, assuming basic requirements are met, we end up creating a `TriangleMesh` object, which is then further processed/acted upon. For demonstration purposes, let us examine how the `.amf` file format behaves in this regard. Upon finding a closing `</volume>` tag within the XML, we hit the following code:

```
void AMFParserContext::endElement(const char * /* name */)
{
    switch (m_path.back()) {

        // Constellation transformation:
        case NODE_TYPE_DELTAX:
        case NODE_TYPE_DELTAX:
        case NODE_TYPE_DELTAX:
    //[...]

       // Closing the current volume. Create an STL from m_volume_facets pointing to m_object_vertices.
    case NODE_TYPE_VOLUME:
    {
        assert(m_object && m_volume);
        TriangleMesh  mesh;            // [1]
        stl_file     &stl = mesh.stl;
        stl.stats.type = inmemory;
        stl.stats.number_of_facets = int(m_volume_facets.size() / 3);   // [2]
        stl.stats.original_num_facets = stl.stats.number_of_facets;
        stl_allocate(&stl);

        bool has_transform = ! m_volume_transform.isApprox(Transform3d::Identity(), 1e-10);
        for (size_t i = 0; i < m_volume_facets.size();) {       // [3]
            stl_facet &facet = stl.facet_start[i/3];
            for (unsigned int v = 0; v < 3; ++v)
            {
                unsigned int tri_id = m_volume_facets[i++] * 3;
                facet.vertex[v] = Vec3f(m_object_vertices[tri_id + 0], m_object_vertices[tri_id + 1], m_object_vertices[tri_id + 2]);
            }
        }
        stl_get_size(&stl);
        mesh.repair();             // [4]
//[...]
```

At [1], we see our desired `TriangleMesh` object being instantiated, and at [2], an important variable `stl.stats.number_of_facets` is set as the amount of `m_volume_facets.size() / 3`; `m_volume_facets` is just a collection of all of the co-ordinates of all the triangles from our input. So if `m_volume_facets` looks like `std::vector of length 9, capacity 16 = {0x2, 0x3, 0x1, 0x2, 0x3, 0x0, 0x4, 0x1, 0x4}`, then this just means we have three triangle objects with three vertices each, each number representing the vertex index. Carrying on in the above example, at [3], the `stl.facet_start` vector is populated with `m_volume_facets.size()` elements, and at [4], we check the resultant set of facets to see if they make sense as a TriangleMesh and to repair if not. For the most part `TriangleMesh::repair()` consists of checks and assertions, but for our purposes, the only one that matters is here:

```
    // normal_directions
#ifdef SLIC3R_TRACE_REPAIR
    BOOST_LOG_TRIVIAL(trace) << "\tstl_fix_normal_directions";
#endif /* SLIC3R_TRACE_REPAIR */
    stl_fix_normal_directions(&stl);  // [1]
    assert(stl_validate(&this->stl));
```

The assumption is that certain facets in the list might be reversed, and normalization is enforced by [1]. Examining `admesh/normals.cpp:stl_fix_normal_directions()`:

```
void stl_fix_normal_directions(stl_file *stl)
{
    if (stl->stats.number_of_facets == 0)
        return;

    //[...]
    // Initialize linked list.
    boost::object_pool<stl_normal> pool;
    stl_normal *head = pool.construct();
    stl_normal *tail = pool.construct();
    head->next = tail;
    tail->next = tail;

    // Initialize list that keeps track of already fixed facets.
    std::vector<char> norm_sw(stl->stats.number_of_facets, 0); // stats.number_of_facets % 3 != 0 => oob write.
    // Initialize list that keeps track of reversed facets.
    std::vector<int>  reversed_ids(stl->stats.number_of_facets, 0);
```

The first important characteristic of this function is that we allocate two vectors with a size of `stl->stats.number_of_facets`, which is of size `m_volume_facets.size() / 3`, i.e. the amount of triangles read in from our input. For completeness, this is what a 'Triangle' looks like from a `.amf` or `.3mf` file:

```
  <volume materialid="2"></triangle>
    <triangle><v1>0</v1><v2>1</v2><v3>4</v3></triangle>
    <triangle><v1>4</v1><v2>1</v2><v3>2</v3></triangle>
    <triangle><v1>1</v1><v2>3</v2><v3>2</v3></triangle>
  </volume>
```

Elements `<v1>`, `<v2>`, `<v3>` all point to different vertex index, which look like such:

```
  <vertices>
    <vertex><coordinates><x>11</x><y>0</y><z>0</z></coordinates></vertex>   // index 0
    <vertex><coordinates><x>0</x><y>0</y><z>0</z></coordinates></vertex>
    <vertex><coordinates><x>1</x><y>1</y><z>0</z></coordinates></vertex>
    <vertex><coordinates><x>5</x><y>2</y><z>0</z></coordinates></vertex>
    <vertex><coordinates><x>5</x><y>2</y><z>2</z></coordinates></vertex>
  </vertices>
```

Thus, to reiterate, `stl->stats.number_of_facets` can be thought of as the number of valid `<triangle>` objects in our input file. Continuing in `admesh/normals.cpp:stl_fix_normal_directions()`:

```
    for (;;) {
        // Add neighbors_to_list. Add unconnected neighbors to the list.
        bool force_exit = false;
        for (int j = 0; j < 3; ++ j) {   // [1]
            // Reverse the neighboring facets if necessary.
            if (stl->neighbors_start[facet_num].which_vertex_not[j] > 2) {
                // If the facet has a neighbor that is -1, it means that edge isn't shared by another facet
                if (stl->neighbors_start[facet_num].neighbor[j] != -1) {
                    if (norm_sw[stl->neighbors_start[facet_num].neighbor[j]] == 1) {

                        for (int id = reversed_count - 1; id >= 0; -- id)
                            reverse_facet(stl, reversed_ids[id]);
                        force_exit = true;
                        break;
                    }
                    reverse_facet(stl, stl->neighbors_start[facet_num].neighbor[j]); // if amount of
                    reversed_ids[reversed_count ++] = stl->neighbors_start[facet_num].neighbor[j];  // [2]
                }
            }
    //[..]
```

The only thing that we must pay attention to: until the `for(;;)` loop breaks, [1] always executes three times (assuming we don't hit the `break`). Thus, the statement at [2] can potentially be hit three times max per `for (;;)` iteration, assuming that a given facet has enough valid neighbors. As mentioned/shown above, the `reversed_ids` vector's length is equivalent to the amount of triangles in the input file, and also there's no guarantee that the `(reversed_count % 3) == 0`.

Thus, for example, assume we have an input file in which there's only four triangles that are connected (e.g. a pyramid) and our `stl->neightbors_start` vector looks like such:

```
[o.o]> p/x stl->neighbors_start
$17 = std::vector of length 4, capacity 5 = {{neighbor = {0x1, 0x3, 0x2}, which_vertex_not = {0x2, 0x4, 0x3}}, {neighbor = {0x0, 0x2, 0x3},
which_vertex_not = {0x2, 0x2, 0x3}}, {neighbor = {0x1, 0x0, 0x3}, which_vertex_not = {0x0, 0x4, 0x5}}, {neighbor = {0x2, 0x1, 0x0},
which_vertex_not = {0x4, 0x4, 0x3}}}
```

Since each facet/triangle has three neighbors, if each of these neighboring facets needs to be reversed, we can quickly exceed the amount of elements in the `reversed_ids` vector, which again was allocated with `stl->number_of_facets` elements. Given a specific layout of facets/triangles, the same facet may be reversed multiple times, causing the assignment at [2] to write out of bounds, resulting in a out-of-bounds heap write and possible code execution.

A last important note: while this vulnerability is in `admesh/normals.cpp`, it seems that this “admesh” library is a re-write or alternate of the standard “admesh” library, which is written in C. It does not appear that this vulnerability applies to the standard “admesh” library.

Crash Information

```
=================================================================
==2302481==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200002dae0 at pc 0x7f4ae8a14209 bp 0x7fffea4d5fb0 sp 0x7fffea4d5fa8
WRITE of size 4 at 0x60200002dae0 thread T0
    #0 0x7f4ae8a14208 in stl_fix_normal_directions(stl_file*) //boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/admesh/normals.cpp:168:47
    #1 0x7f4ae5dfe888 in Slic3r::TriangleMesh::repair(bool)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/TriangleMesh.cpp:178:5
    #2 0x7f4ae4d9d106 in Slic3r::AMFParserContext::endElement(char const*)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/Format/AMF.cpp:642:14
    #3 0x7f4ae4da672c in Slic3r::AMFParserContext::endElement(void*, char const*)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/Format/AMF.cpp:97:14
    #4 0x7f4adf19d9d9  (/lib/x86_64-linux-gnu/libexpat.so.1+0xb9d9)
    #5 0x7f4adf19e6af  (/lib/x86_64-linux-gnu/libexpat.so.1+0xc6af)
    #6 0x7f4adf19bb82  (/lib/x86_64-linux-gnu/libexpat.so.1+0x9b82)
    #7 0x7f4adf19d04d  (/lib/x86_64-linux-gnu/libexpat.so.1+0xb04d)
    #8 0x7f4adf1a0dbf in XML_ParseBuffer (/lib/x86_64-linux-gnu/libexpat.so.1+0xedbf)
    #9 0x7f4ae4da59cf in Slic3r::load_amf_file(char const*, Slic3r::DynamicPrintConfig*, Slic3r::Model*)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/Format/AMF.cpp:877:13
    #10 0x7f4ae4da8763 in Slic3r::load_amf(char const*, Slic3r::DynamicPrintConfig*, Slic3r::Model*, bool)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/Format/AMF.cpp:1048:16
    #11 0x565a98 in LLVMFuzzerTestOneInput //boop/assorted_fuzzing/prusaslicer/./fuzz_amf_harness.cpp:82:20
    #12 0x46be11 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
(//boop/assorted_fuzzing/prusaslicer/amf_fuzzdir/fuzzamf.bin+0x46be11)
    #13 0x457582 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long)
(//boop/assorted_fuzzing/prusaslicer/amf_fuzzdir/fuzzamf.bin+0x457582)
    #14 0x45d036 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long))
(//boop/assorted_fuzzing/prusaslicer/amf_fuzzdir/fuzzamf.bin+0x45d036)
    #15 0x485cf2 in main (//boop/assorted_fuzzing/prusaslicer/amf_fuzzdir/fuzzamf.bin+0x485cf2)
    #16 0x7f4ae0a3e0b2 in __libc_start_main /build/glibc-ZN95T4/glibc-2.31/csu/../csu/libc-start.c:308:16
    #17 0x431c4d in _start (//boop/assorted_fuzzing/prusaslicer/amf_fuzzdir/fuzzamf.bin+0x431c4d)

0x60200002dae0 is located 0 bytes to the right of 16-byte region [0x60200002dad0,0x60200002dae0)
allocated by thread T0 here:
    #0 0x5610cd in operator new(unsigned long) (//boop/assorted_fuzzing/prusaslicer/amf_fuzzdir/fuzzamf.bin+0x5610cd)
    #1 0x7f4ae49ac5cb in __gnu_cxx::new_allocator<int>::allocate(unsigned long, void const*) /usr/bin/../lib/gcc/x86_64-linux-
gnu/9/../../../../include/c++/9/ext/new_allocator.h:114:27
    #2 0x7f4ae49ac4f8 in std::allocator_traits<std::allocator<int> >::allocate(std::allocator<int>&, unsigned long)
/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/bits/alloc_traits.h:444:20
    #3 0x7f4ae49abf6f in std::_Vector_base<int, std::allocator<int> >::_M_allocate(unsigned long) /usr/bin/../lib/gcc/x86_64-linux-
gnu/9/../../../../include/c++/9/bits/stl_vector.h:343:20
    #4 0x7f4ae49ad4eb in std::_Vector_base<int, std::allocator<int> >::_M_create_storage(unsigned long) /usr/bin/../lib/gcc/x86_64-linux-
gnu/9/../../../../include/c++/9/bits/stl_vector.h:358:33
    #5 0x7f4ae49acf5f in std::_Vector_base<int, std::allocator<int> >::_Vector_base(unsigned long, std::allocator<int> const&)
/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/bits/stl_vector.h:302:9
    #6 0x7f4ae5e9c937 in std::vector<int, std::allocator<int> >::vector(unsigned long, int const&, std::allocator<int> const&)
/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/bits/stl_vector.h:521:9
    #7 0x7f4ae8a139e3 in stl_fix_normal_directions(stl_file*) //boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/admesh/normals.cpp:136:20
    #8 0x7f4ae5dfe888 in Slic3r::TriangleMesh::repair(bool)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/TriangleMesh.cpp:178:5
    #9 0x7f4ae4d9d106 in Slic3r::AMFParserContext::endElement(char const*)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/Format/AMF.cpp:642:14
    #10 0x7f4ae4da672c in Slic3r::AMFParserContext::endElement(void*, char const*)
//boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/libslic3r/Format/AMF.cpp:97:14
    #11 0x7f4adf19d9d9  (/lib/x86_64-linux-gnu/libexpat.so.1+0xb9d9)

SUMMARY: AddressSanitizer: heap-buffer-overflow //boop/assorted_fuzzing/prusaslicer/PrusaSlicer/src/admesh/normals.cpp:168:47 in
stl_fix_normal_directions(stl_file*)
Shadow bytes around the buggy address:
  0x0c047fffdb00: fa fa 00 fa fa fa 00 00 fa fa 00 07 fa fa 00 fa
  0x0c047fffdb10: fa fa fd fa fa fa fd fd fa fa fd fd fa fa fd fd
  0x0c047fffdb20: fa fa fd fa fa fa fd fa fa fa fd fd fa fa fd fd
  0x0c047fffdb30: fa fa fd fa fa fa fd fd fa fa fd fd fa fa fd fa
  0x0c047fffdb40: fa fa fd fa fa fa fd fd fa fa fd fd fa fa fd fd
=>0x0c047fffdb50: fa fa fd fa fa fa 04 fa fa fa 00 00[fa]fa fa fa
  0x0c047fffdb60: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fffdb70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fffdb80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fffdb90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fffdba0: 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
==2302481==ABORTING
```

Timeline

2021-01-08 - Vendor Disclosure
2021-04-21 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT

TALOS-2020-1220

NEXT REPORT

TALOS-2021-1239