Talos Vulnerability Report

TALOS-2021-1293

AT&T Labs Xmill XML decompression LabelDict::Load heap-based buffer overflow vulnerability

AUGUST 10, 2021

CVE NUMBER

CVE-2021-21830

Summary

A heap-based buffer overflow vulnerability exists in the XML Decompression LabelDict::Load functionality of AT&T Labs' Xmill 0.7. A specially crafted XMI file can lead to remote code execution. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

AT&T Labs Xmill 0.7
Schneider Electric EcoStruxure Control Expert 15

Product URLs

None

CVSSv3 Score

8.1 - CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE

CWE-122 - Heap-based Buffer Overflow

Details

Xmill and Xdemill are utilities that are purpose-built for XML compression and decompression, respectively. These utilities claim to be roughly two times more efficient at compressing XML than other compression methods.

While this software is old, released in 1999, it can be found in modern software suites, such as Schneider Electric's EcoStruxure Control Expert.

Within `LabelDict::Load` within `UncompressBlockHeader` a call to `MemStreamer::WordAlign` is not accounted for in the size of the buffer that is allocated. Since `MemStreamer::WordAlign` happens in a loop and `MemStreamer::GetByteBlock` only allocates new memory if it is necessary. This function does not increment the total size used of the `mainmem` block after copying data into it, thus the `MemStreamer::GetByteBlock` will not account for the data copied in, this can manifest in large out of bounds heap writes, which can result in remote code execution.

```
    void Load(SmallBlockUncompressor *uncompress)
       // Loads the next block of labels and appends the block to the
       // already existing blocks in the dictionary
    {
       UncompressLabelDictItem *dictitemptr;
       char                    isattrib;
       unsigned                dictlabelnum;

       // Let's get the number of labels first
       unsigned mylabelnum=uncompress->LoadUInt32();
       ...

 //******

       lastlabeldict->labelnum=mylabelnum;
       labelnum+=mylabelnum;

       dictlabelnum=mylabelnum;

       dictitemptr=lastlabeldict->GetArrayPtr();

       // We copy the actual labels now
       // Each label is represented by the length and the attribute-flag
       // Then, the actual name follows
       while(dictlabelnum--)
       {
          dictitemptr->len=(unsigned short)uncompress->LoadSInt32(&isattrib);
          dictitemptr->isattrib=isattrib;

          dictitemptr->strptr=mainmem.GetByteBlock(dictitemptr->len);
          mainmem.WordAlign();

          mymemcpy(dictitemptr->strptr,
                   uncompress->LoadData(dictitemptr->len),
                   dictitemptr->len);

          dictitemptr++;
       }
    }
```

`MemStreamer::WordAlign` is used to align memory to a 4 byte boundary, but the alignment happens after the check to see if the block is large enough to hold the data.

```
void WordAlign()
   // Allocated 1 to 3 bytes to align the current memory pointer
   // to an address divisible by 4.
{
   if(curblock==NULL)
      return;

   int addsize=3-((curblock->cursize+3)&3);
   if(addsize>0)
   {
      curblock->cursize+=addsize;
      overallsize+=addsize;
   }
}
```

`MemStreamer::GetByteBlock` only allocates more memory if the current block cannot hold the data, this is tracked in a global `MemStreamer` structure.

```
char *GetByteBlock(unsigned len)
   // The main function for allocated more memory of size 'len'.
   // The function checks the current block and if there is not enough space,
   // the function 'AllocateNewBlock' is called.
{
   if(len+sizeof(MemStreamBlock)-1>LARGEBLOCK_SIZE) // Is the requested size larger than the biggest possible block?
   {
      char str[100];
      sprintf(str,"Could not allocate %lu bytes (largest possible block size=%lu bytes) !",
         sizeof(MemStreamBlock)-1+len,
         LARGEBLOCK_SIZE);
      Error(str);
      Exit();
   }

   if(curblock==NULL)
      // We don't have a block yet ?
      firstblock=curblock=AllocateNewBlock();

   if(curblock->blocksize-curblock->cursize>=len)
      // Enough space in current block?
   {
      char *ptr=curblock->data+curblock->cursize;
      curblock->cursize+=len;
      overallsize+=len;
      return ptr;
   }
   else  // We add a new block at the end
   {
      do
      {
         curblock->next=AllocateNewBlock();
         curblock=curblock->next;
      }
      while(curblock->blocksize<len);  // If we don't have enough space,
                                       // we simply create a bigger one!

      curblock->cursize=len;
      overallsize+=len;

      return curblock->data;
   }
}
```

Crash Information

```
=================================================================
==32767==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb4674800 at pc 0x080f37f5 bp 0xbf8124e8 sp 0xbf8120c0
WRITE of size 37 at 0xb4674800 thread T0
    #0 0x80f37f4 in __asan_memcpy (/home/fuzz/Desktop/xmill/unix/xdemill+0x80f37f4)
    #1 0x819ab49 in LabelDict::Load(SmallBlockUncompressor*) /home/fuzz/Desktop/xmill/./src/LabelDict.hpp:453:10
    #2 0x8197d05 in UncompressBlockHeader(Input*) /home/fuzz/Desktop/xmill/./src/Main.cpp:784:4
    #3 0x8197100 in Uncompress(char*, char*) /home/fuzz/Desktop/xmill/./src/Main.cpp:840:10
    #4 0x8196c37 in HandleSingleFile(char*) /home/fuzz/Desktop/xmill/./src/Main.cpp:248:10
    #5 0x8197482 in HandleFileArg(char*) /home/fuzz/Desktop/xmill/./src/Main.cpp:382:4
    #6 0x81976f5 in main /home/fuzz/Desktop/xmill/./src/Main.cpp:494:7
    #7 0xb7b3b646 in __libc_start_main /build/glibc-ViVLyQ/glibc-2.23/csu/../csu/libc-start.c:291
    #8 0x80664d3 in _start (/home/fuzz/Desktop/xmill/unix/xdemill+0x80664d3)

0xb4674800 is located 0 bytes to the right of 65536-byte region [0xb4664800,0xb4674800)
allocated by thread T0 here:
    #0 0x810a814 in __interceptor_malloc (/home/fuzz/Desktop/xmill/unix/xdemill+0x810a814)
    #1 0x819d9c5 in AllocateBlockRecurs(unsigned char) /home/fuzz/Desktop/xmill/./src/MemMan.cpp:71:22
    #2 0x818bc9e in AllocateBlock(unsigned char) /home/fuzz/Desktop/xmill/./src/MemMan.hpp:68:11
    #3 0x818b8a7 in MemStreamer::AllocateNewBlock() /home/fuzz/Desktop/xmill/./src/MemStreamer.hpp:121:34
    #4 0x818b35c in MemStreamer::GetByteBlock(unsigned int) /home/fuzz/Desktop/xmill/./src/MemStreamer.hpp:222:28
    #5 0x819a99c in LabelDict::Load(SmallBlockUncompressor*) /home/fuzz/Desktop/xmill/./src/LabelDict.hpp:450:30
    #6 0x8197d05 in UncompressBlockHeader(Input*) /home/fuzz/Desktop/xmill/./src/Main.cpp:784:4
    #7 0x8197100 in Uncompress(char*, char*) /home/fuzz/Desktop/xmill/./src/Main.cpp:840:10
    #8 0x8196c37 in HandleSingleFile(char*) /home/fuzz/Desktop/xmill/./src/Main.cpp:248:10
    #9 0x8197482 in HandleFileArg(char*) /home/fuzz/Desktop/xmill/./src/Main.cpp:382:4
    #10 0x81976f5 in main /home/fuzz/Desktop/xmill/./src/Main.cpp:494:7
    #11 0xb7b3b646 in __libc_start_main /build/glibc-ViVLyQ/glibc-2.23/csu/../csu/libc-start.c:291

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/fuzz/Desktop/xmill/unix/xdemill+0x80f37f4) in __asan_memcpy
Shadow bytes around the buggy address:
  0x368ce8b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x368ce8c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x368ce8d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x368ce8e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x368ce8f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x368ce900:[fa]fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x368ce910: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x368ce920: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x368ce930: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x368ce940: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x368ce950: 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
  Heap right redzone:      fb
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack partial redzone:   f4
  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
ASAN:DEADLYSIGNAL
=================================================================
```

Timeline

2021-04-30 - Vendor Disclosure

2021-08-10 - Public Release

CREDIT

Discovered by Carl Hurd of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT NEXT REPORT

TALOS-2020-1212 TALOS-2021-1292