Talos Vulnerability Report

TALOS-2021-1279

AT&T Labs Xmill XML parsing CreateLabelOrAttrib memory corruption vulnerability

AUGUST 11, 2021

CVE NUMBER

CVE-2021-21811

Summary

A memory corruption vulnerability exists in the XML-parsing CreateLabelOrAttrib functionality of AT&T Labs' Xmill 0.7. A specially crafted XML file can lead to a heap buffer overflow. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

AT&T Labs Xmill 0.7

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-191 - Integer Underflow (Wrap or Wraparound)

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.

Xmill utilizes custom heap management code within the software to distribute pointers within the normal heap. Heap chunks are allocated in set block sizes as follows:

```
unsigned long blocksizes[BLOCKSIZE_NUM]= {256, 1024, 8192, 65536};
```

These blocks are then used for various data within the Xmill process for XML compression.

During the process of parsing the provided XML file, tags are stored within the custom heap allocations in order to best compress them. `LabelDict::CreateLabelOrAttrib` is used within `XMLParse::DoParsing` in order to allocate and write these labels and attributes into the custom heap chunks. The custom heap chunk management has a bug which allows for an arbitrary heap write.

This bug starts in `LabelDict::CreateLabelOrAttrib` when allocating space for a label or attribute of size 0xFFC9

```
TLabelID CreateLabelOrAttrib(char *label,unsigned len,unsigned char isattrib)
    // Creates a new label in the hash table
{
    // Let's get some memory first
    mainmem.WordAlign();
    CompressLabelDictItem *item=(CompressLabelDictItem *)mainmem.GetByteBlock(sizeof(CompressLabelDictItem)+len);
    mainmem.WordAlign();
    ...
```

As seen above, the call to `GetByteBlock` takes into account the input length and the size of `CompressLabelDictItem` which is of size 0x18, adding these sizes together we get a total of 0xFFE1 bytes.

```
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.
{
    ...

    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;
    }
}
```

Within `GetByteBlock` a new heap chunk will be allocated of size 0x10000 (65536) to hold this data.

```
MemStreamBlock *AllocateNewBlock()
    // Allocates a new block
    // The size is determined by the next index in blocksizeidxs
{
    MemStreamBlock *newblock;

    // Let's get the new block
    newblock=(MemStreamBlock *)AllocateBlock(blocksizeidxs[curblocksizeidxidx]);

    // The usable data size is the block size - the size of the header
    newblock->blocksize=GetBlockSize(blocksizeidxs[curblocksizeidxidx])-(sizeof(MemStreamBlock)-1);
    ...
```

While the heap chunk size is technically 0x10000, within `AllocateNewBlock` the size is actually the total size (0x10000) minus the total size of the `MemStreamBlock` minus 1, which makes the total storable data equal to 0xFFE1 in a 0x10000 byte chunk. This means that the data provided in `CreateLabelOrAttrib` completely fills the custom chunk. Once returned from `GetByteBlock` the memory is aligned using `WordAlign`

```
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;
    }
}
```

This sets the `curblock->cursize` to 0xFFE4. The rest of `CreateLabelOrAttrib` is executed without issue, copying the `len` bytes into the new buffer. When the next call to `CreateLabelOrAttrib` is made, to create another label is when the bug arises.

When calling into `GetByteBlock` from `CreateLabelOrAttrib` we first check to see if the data can fit in the current block, due to the `WordAlign` moving `curblock->cursize` to an aligned value, we now have a `curblock->cursize` of 0xFFE4 and a `curblock->blocksize` of 0xFFE1 which causes an integer underflow with an unsigned comparison.

```
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(curblock->blocksize-curblock->cursize>=len)
        // Enough space in current block?
    {
        char *ptr=curblock->data+curblock->cursize;
        curblock->cursize+=len;
        overallsize+=len;
        return ptr;
    }
    ...
}
```

With the integer underflow, this block is considered large enough to hold any amount of data, a pointer is returned, aligned, and then data is copied when finishing `CreateLabelOrAttrib`

```
TLabelID CreateLabelOrAttrib(char *label,unsigned len,unsigned char isattrib)
    // Creates a new label in the hash table
{
    // Let's get some memory first
    mainmem.WordAlign();
    CompressLabelDictItem *item=(CompressLabelDictItem *)mainmem.GetByteBlock(sizeof(CompressLabelDictItem)+len);
    mainmem.WordAlign();

    // We copy the label name
    item->len=(unsigned short)len;
    mymemcpy(item->GetLabelPtr(),label,len);
    ...
```

This integer underflow results in an exploitable heap corruption.

Crash Information

```
=================================================================
==2676449==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x631000074814 at pc 0x000000495dca bp 0x7fffffed330 sp 0x7fffffecaf8
WRITE of size 6 at 0x631000074814 thread T0
    #0 0x495dc9 in __asan_memcpy (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x495dc9)
    #1 0x51737d in LabelDict::CreateLabelOrAttrib(char*, unsigned int, unsigned char) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x51737d)
    #2 0x5162d2 in SAXClient::HandleStartLabel(char*, int, char) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x5162d2)
    #3 0x527175 in XMLParse::ParseLabel() (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x527175)
    #4 0x52364c in XMLParse::DoParsing(SAXClient*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x52364c)
    #5 0x52161e in Compress(char*, char*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x52161e)
    #6 0x5211a6 in HandleSingleFile(char*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x5211a6)
    #7 0x521b91 in HandleFileArg(char*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x521b91)
    #8 0x522059 in main (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x522059)
    #9 0x7ffff7a71cb1 in __libc_start_main csu/../csu/libc-start.c:314:16
    #10 0x41c84d in _start (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x41c84d)

0x631000074814 is located 20 bytes to the right of 65536-byte region [0x631000064800,0x631000074800)
allocated by thread T0 here:
    #0 0x4968bd in malloc (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x4968bd)
    #1 0x52ec6a in AllocateBlockRecurs(unsigned char) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x52ec6a)
    #2 0x50d4d5 in AllocateBlock(unsigned char) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x50d4d5)
    #3 0x50d0c8 in MemStreamer::AllocateNewBlock() (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x50d0c8)
    #4 0x50de9c in MemStreamer::GetByteBlock(unsigned int) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x50de9c)
    #5 0x5172fb in LabelDict::CreateLabelOrAttrib(char*, unsigned int, unsigned char) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x5172fb)
    #6 0x5162d2 in SAXClient::HandleStartLabel(char*, int, char) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x5162d2)
    #7 0x526f1c in XMLParse::ParseLabel() (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x526f1c)
    #8 0x52364c in XMLParse::DoParsing(SAXClient*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x52364c)
    #9 0x52161e in Compress(char*, char*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x52161e)
    #10 0x5211a6 in HandleSingleFile(char*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x5211a6)
    #11 0x521b91 in HandleFileArg(char*) (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x521b91)
    #12 0x522059 in main (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x522059)
    #13 0x7ffff7a71cb1 in __libc_start_main csu/../csu/libc-start.c:314:16

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/fuzz/Desktop/xmill/unix/xmill-asan+0x495dc9) in __asan_memcpy
Shadow bytes around the buggy address:
  0x0c62800068b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c62800068c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c62800068d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c62800068e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c62800068f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c6280006900: fa fa[fa]fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c6280006910: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c6280006920: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c6280006930: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c6280006940: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c6280006950: 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
==2676449==ABORTING
```

Timeline

2021-04-30 - Vendor Disclosure

2021-08-10 - Public Release

CREDIT

Discovered by Carl Hurd of Cisco Talos.
