Talos Vulnerability Report

TALOS-2020-1190

SoftMaker Office PlanMaker Document Records 0x8011 and 0x820a integer overflow vulnerability

FEBRUARY 3, 2021

CVE NUMBER

CVE-2020-13579

Summary

An exploitable integer overflow vulnerability exists in the PlanMaker document parsing functionality of SoftMaker Office 2021's PlanMaker application. A specially crafted document can cause the document parser perform arithmetic that may overflow which can result in an undersized heap allocation. Later when copying data from the file into this allocation, a heap-based buffer overflow will occur which can corrupt memory. These types of memory corruptions can allow for code execution under the context of the application. An attacker can entice the victim to open a document to trigger this vulnerability.

Tested Versions

SoftMaker Software GmbH SoftMaker Office PlanMaker 2021 (Revision 1014)

Product URLs

https://www.softmaker.com/en/softmaker-office

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-190 - Integer Overflow or Wraparound

Details

SoftMaker Software GmbH is a German software company that develops and releases office software. Their flagship product, SoftMaker Office, is supported on a variety of platforms and contains a handful of components which can allow the user to perform a multitude of tasks such as word processing, spreadsheets, presentation design, and even allows for scripting. Thus the SoftMaker Office suite supports a variety of common office file formats, as well as a number of internal formats that the user may choose to use when performing their necessary work.

The PlanMaker component of SoftMaker's suite is designed as an all-around spreadsheet tool, and supports of a number of features that allow it to remain competitive with similar office suites that are developed by its competitors. Although the application includes a number of parsers that enable the user to interact with these common document types or templates, a native document format is also included. This undocumented format is labeled as a PlanMaker Document, and will typically have the extension ".pmd" when saved as a file. The PlanMaker Document file format is based on Microsoft's Compound Document file format and contains two streams, one of which is the "PMW" stream and then the "PMW Objects" stream.

Once the application unpacks the "PMW" stream, it will check the first few records of the stream in order to fingerprint the document and verify the stream if of the correct format. After this confirmation, the application will then execute the following function to read all of the records in the stream. At [1], the function will take an object containing the state and the stream to parse records from in order to store them on the stack. Later, the function will enter a loop at [2] which is responsible for continuously iterating through all of the records in the stream and then parsing them. The function call at [3] is responsible for parsing a general record. This function will return a pointer to the record's contents at [4].

```
0x682f8d:    push   %rbp
0x682f8e:    mov    %rsp,%rbp
0x682f91:    sub    $0x300,%rsp
0x682f98:    mov    %rdi,-0x2e8(%rbp)    ; [1] record object
0x682f9f:    mov    %rsi,-0x2f0(%rbp)    ; [1] stream object
0x682fa6:    mov    %edx,-0x2f4(%rbp)
0x682fac:    mov    %fs:0x28,%rax
0x682fb5:    mov    %rax,-0x8(%rbp)
0x682fb9:    xor    %eax,%eax
...
0x6830bc:    movl   $0x0,-0x2cc(%rbp)    ; [2] beginning of loop
0x6830c6:    mov    -0x2c8(%rbp),%r9
0x6830cd:    lea    -0x2dc(%rbp),%r8
0x6830d4:    lea    -0x2de(%rbp),%rcx
0x6830db:    lea    -0x2e0(%rbp),%rdx
0x6830e2:    mov    -0x2f0(%rbp),%rsi    ; stream
0x6830e9:    mov    -0x2e8(%rbp),%rax    ; record object
0x6830f0:    sub    $0x8,%rsp
0x6830f4:    lea    -0x2d8(%rbp),%rdi
0x6830fb:    push   %rdi
0x6830fc:    mov    %rax,%rdi
0x6830ff:    callq  0x61e4a8             ; [3] parse record
0x683104:    add    $0x10,%rsp
0x683108:    mov    %rax,-0x2c8(%rbp)    ; [4] save pointer to record
...
0x683313:    cmpl   $0x0,-0x2cc(%rbp)
0x68331a:    jne    0x6830bc
```

Within the aforementioned loop, there's a number of sub-loops that are responsible for checking the record's type and using it to dispatch to the correct handler for the record to parse. Once one of the loops finds a handler for the current record type, code similar to the following is executed. This code will calculate an offset into the current function's stack frame, and use it to find an index to one of the record handlers. Once the pointer has been calculated, the record's contents and state are passed to the function call at [5].

```
0x68321d:    mov    -0x2d0(%rbp),%eax
0x683223:    cltq
0x683225:    shl    $0x4,%rax
0x683229:    add    %rbp,%rax
0x68322c:    sub    $0x218,%rax          ; point to function pointer array on stack.
0x683232:    mov    (%rax),%rax
0x683235:    mov    -0x2c8(%rbp),%rcx    ; record contents
0x68323c:    mov    -0x2e8(%rbp),%rdx    ; record object
0x683243:    mov    %rcx,%rsi
0x683246:    mov    %rdx,%rdi
0x683249:    callq  *%rax                ; [5] dispatch to record handler
0x68324b:    test   %eax,%eax
0x68324d:    sete   %al
0x683250:    test   %al,%al
0x683252:    jne    0x68338d
```

When either record types 0x8011 or 0x820a are parsed, the following function will be used to process their contents. After storing the parsing state and a pointer to the current record in the frame, at [6] the application will shift the pointer to the record past the uint16_t record type, and a uint16_t length. Afterwards the application will store 0x1c into the `%eax` register and then at [7] will check if the record type is 0x8011. If it's not, then the record type is 0x820a and at [8] the application will subtract 2 from the prior calculated constant. Finally at [9], the application will read a uint32_t from offset +0x12 of the record's contents. This uint32_t is explicitly trusted and will later be used in a signed multiply which can result in an integer overflow.

```
0x67eced:    push   %rbp
0x67ecee:    mov    %rsp,%rbp
0x67ecf1:    push   %rbx
0x67ecf2:    sub    $0x1f8,%rsp
0x67ecf9:    mov    %rdi,-0x1f8(%rbp)        ; object
0x67ed00:    mov    %rsi,-0x200(%rbp)        ; record data
0x67ed07:    mov    %fs:0x28,%rax
0x67ed10:    mov    %rax,-0x18(%rbp)
...
0x67ed27:    mov    -0x200(%rbp),%rax        ; record data
0x67ed2e:    add    $0x4,%rax                ; shift past record length and type
0x67ed32:    mov    %rax,-0x1c8(%rbp)        ; [6] store it as the record contents
0x67ed39:    mov    $0x18,%eax
0x67ed3e:    add    $0x4,%eax
0x67ed41:    mov    %eax,-0x1d8(%rbp)
0x67ed47:    mov    -0x200(%rbp),%rax        ; record data
0x67ed4e:    movzwl (%rax),%eax              ; read record type
0x67ed51:    mov    %ax,-0x1e2(%rbp)
0x67ed58:    cmpw   $0x8011,-0x1e2(%rbp)     ; [7] check if its 0x8011
0x67ed61:    jne    0x67ed76
...
0x67ed63:    mov    -0x1d8(%rbp),%eax
0x67ed69:    mov    $0x2,%edx                ; [8] if type is 0x8201, then subtract 2 from constant
0x67ed6e:    sub    %edx,%eax
0x67ed70:    mov    %eax,-0x1d8(%rbp)
...
0x67ed76:    mov    -0x1c8(%rbp),%rax        ; record contents
0x67ed7d:    mov    0x12(%rax),%eax          ; [9] read uint32_t from record's contents at +0x12
0x67ed80:    test   %eax,%eax
0x67ed82:    je     0x67f12d
```

After reading the uint32_t, the following code will be executed. At [10], the application will again read the uint32_t at offset +0x12 of the record's contents, and multiply it by 8. Due to the application explicitly trusting the uint32_t, this multiplication can overflow resulting in a smaller value than intended. At [11], this undersized value is then passed as a size to a function responsible for allocating a buffer. This results in an undersized heap allocation which is then stored into a pointer.

```
0x67eeaf:    mov    $0x8,%edx
0x67eeb4:    mov    -0x1c8(%rbp),%rax        ; record contents
0x67eebb:    mov    0x12(%rax),%eax          ; [10] read uint32_t from +0x12 of record
0x67eebe:    imul   %eax,%edx                ; [10] multiply by 8 and save in %edx
...
0x67eec1:    mov    -0x1d0(%rbp),%rax
0x67eec8:    mov    0x8(%rax),%rax
0x67eecc:    mov    %edx,%esi                ; pass multiplication result as size
0x67eece:    mov    %rax,%rdi
0x67eed1:    callq  0xab7a01                 ; [11] allocate buffer
0x67eed6:    mov    %rax,%rdx
0x67eed9:    mov    -0x1c0(%rbp),%rax        ; [11] store pointer to allocation
```

After allocating the pointer which is used for an array, the application will enter the following loop. At [12], the current index for the loop is tested against the original uint32_t at offset +0x12 of the record. This results in the loop iterating that number of times. Within this loop is a pointer that is calculated that is written to in order to write data from the record into the pointer that was prior allocated. At [13], this pointer is incremented to point to each element of the array within the allocation. At [14], the loop will increment its index and continue on to the next pass.

```
0x67ef02:    mov    -0x1c8(%rbp),%rax        ; record contents
0x67ef09:    mov    0x12(%rax),%edx          ; [12] loop sentinel from +0x12 of record
0x67ef0c:    mov    -0x1d4(%rbp),%eax        ; loop index
0x67ef12:    cmp    %eax,%edx
0x67ef14:    jbe    0x67f134                 ; break
...
0x67f111:    movzwl -0x1e0(%rbp),%eax        ; aggregate size
0x67f118:    add    %eax,-0x1d8(%rbp)        ; [13] use to increment pointer that's written to.
0x67f11e:    jmp    0x67f121
0x67f120:    nop
0x67f121:    addl   $0x1,-0x1d4(%rbp)        ; [14] increment index
0x67f128:    jmpq   0x67ef02
```

For each iteration of the loop, there are multiple places where the aggregated pointer is used to write into the undersized heap buffer. In the following code at [15], the application will write a null-byte into the heap buffer. As the loop will iterate more times than the amount of space that was allocated on the heap, eventually this pointer will point outside the heap buffer. The store instructions within the loop will then write outside the bounds of the buffer causing a heap-based buffer overflow and corrupting memory. This could lead to code execution under the context of the application.

```
0x67ef1a:    mov    -0x1c0(%rbp),%rax
0x67ef21:    mov    0x18(%rax),%rax
0x67ef25:    mov    -0x1d4(%rbp),%edx        ; loop index
0x67ef2b:    movslq %edx,%rdx
0x67ef2e:    shl    $0x3,%rdx
0x67ef32:    add    %rdx,%rax                ; adjust pointer
0x67ef35:    movq   $0x0,(%rax)              ; [15] write null byte to pointer
```

Crash Information

In the provided proof-of-concept, the uint32_t is set to 0x20000001. When multiplied by 8, this will result in a heap buffer of 8 bytes and a loop that iterates 0x20000001 times. This ends up corrupting memory belonging to the heap allocator which upon it being used, will access the heap allocators corrupted metadata.

```
Thread 1 "planmaker" received signal SIGSEGV, Segmentation fault.
0x00007ffff7651a8f in unlink_chunk (p=p@entry=0x2d35040, av=0x7ffff77a4b80 <main_arena>) at malloc.c:1453
1453    malloc.c: No such file or directory.

(gdb) bt
#0  0x00007ffff7651a8f in unlink_chunk (p=p@entry=0x2d35040, av=0x7ffff77a4b80 <main_arena>) at malloc.c:1453
#1  0x00007ffff76547d3 in _int_malloc (av=av@entry=0x7ffff77a4b80 <main_arena>, bytes=bytes@entry=0x2020) at malloc.c:4041
#2  0x00007ffff7656479 in __GI___libc_malloc (bytes=0x2020) at malloc.c:3066
#3  0x0000000000ab7090 in ?? ()
#4  0x0000000000ab6332 in ?? ()
#5  0x0000000000ab6bed in ?? ()
#6  0x0000000000ab7a35 in ?? ()
#7  0x00000000004e4618 in ?? ()
#8  0x000000000067ef99 in ?? ()
#9  0x000000000068324b in ?? ()
#10 0x0000000000637421 in ?? ()
#11 0x0000000000638c72 in ?? ()
#12 0x00000000006962fc in ?? ()
#13 0x00000000007ea26e in ?? ()
#14 0x0000000000802753 in ?? ()
#15 0x0000000000802915 in ?? ()
#16 0x0000000000800495 in ?? ()
#17 0x0000000000a1e85c in ?? ()
#18 0x0000000000a21dee in ?? ()
#19 0x00000000010996cd in ?? ()
#20 0x00007ffff75e00b3 in __libc_start_main (main=0x109963e, argc=0x2, argv=0x7fffffffeb08, init=<optimized out>, fini=<optimized out>,
    rtld_fini=<optimized out>, stack_end=0x7fffffffeaf8) at ../csu/libc-start.c:308
#21 0x0000000000411c69 in ?? ()
```

Timeline

2020-11-02 - Vendor Disclosure

2021-01-19 - Vendor Patched

2021-02-03 - Public Release

CREDIT

Discovered by a member of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT TALOS-2020-1008

NEXT REPORT TALOS-2020-1191