Talos Vulnerability Report

TALOS-2020-1192

SoftMaker Office PlanMaker Document Record 0x800d memory corruption vulnerability

FEBRUARY 3, 2021

CVE NUMBER

CVE-2020-13581

Summary

An exploitable heap-based buffer 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 to copy data from a particular record type into a buffer that is smaller than the size used for the copy which will cause a heap-based buffer overflow. 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-122 - Heap-based Buffer Overflow

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
```

The vulnerability described by this document is specifically within the handler for the record type 0x800d. When parsing the 0x800d record, the following function is executed. This function first loads the pointer to the record and stores it in a variable within the stack frame. Afterwards at [7], the function will allocate a 2048-byte buffer and then store it into an object. Immediately afterwards, the length of the record will be read from stream, multiplied by 1 at [8], and then stored into the `%edx` register. This length is then used as the length to the call to `memcpy` at [9]. If the length is larger than 2048, then this `memcpy` will write the record's contents past the bounds of the heap allocation that was previously made. Due to the length being explicitly trusted, this can allow an attacker to corrupt memory which can lead to code execution under the context of the application.

```
0x67d53d:    push   %rbp
0x67d53e:    mov    %rsp,%rbp
0x67d541:    sub    $0x20,%rsp
0x67d545:    mov    %rdi,-0x18(%rbp)     ; object
0x67d549:    mov    %rsi,-0x20(%rbp)     ; [6] record contents
0x67d54d:    movl   $0x0,-0x4(%rbp)
...
0x67d577:    mov    $0x1,%eax
0x67d57c:    shl    $0xb,%eax            ; %eax := 0x800
0x67d57f:    mov    %eax,%edx
0x67d581:    mov    -0x18(%rbp),%rax
0x67d585:    mov    (%rax),%rax
0x67d588:    mov    %edx,%esi
0x67d58a:    mov    %rax,%rdi
0x67d58d:    callq  0xab7a01             ; [7] allocate 0x800 bytes
0x67d592:    mov    %rax,%rdx
0x67d595:    mov    -0x18(%rbp),%rax
0x67d599:    mov    %rdx,0x54d0(%rax)    ; [7] store to pointer
...
0x67d5b0:    mov    -0x20(%rbp),%rax     ; pointer to record contents
0x67d5b4:    add    $0x2,%rax            ; shift past record type
0x67d5b8:    movzwl (%rax),%eax          ; read record length
0x67d5bb:    movzwl %ax,%eax
0x67d5be:    mov    $0x1,%edx
0x67d5c3:    imul   %edx,%eax            ; [8] multiply by 1
0x67d5c6:    mov    %eax,%edx
...
0x67d5c8:    mov    -0x20(%rbp),%rax     ; pointer to record contents
0x67d5cc:    lea    0x4(%rax),%rcx       ; shift past record type and length
0x67d5d0:    mov    -0x18(%rbp),%rax
0x67d5d4:    mov    0x54d0(%rax),%rax    ; pointer from allocation
0x67d5db:    mov    %rcx,%rsi
0x67d5de:    mov    %rax,%rdi
0x67d5e1:    callq  0xab9e39             ; [9] memcpy
```

Crash Information

Upon opening up the provided proof-of-concept, the application will crash while trying to dereference user-supplied data.

```
Thread 1 "planmaker" received signal SIGSEGV, Segmentation fault.
0x0000000000ab7b3d in ?? ()
(gdb) x/i $pc
=> 0xab7b3d:    mov    0x18(%rax),%rax

(gdb) bt
#0  0x0000000000ab7b3d in ?? ()
#1  0x00000000010442da in ?? ()
#2  0x00000000006795d7 in ?? ()
#3  0x000000000068324b in ?? ()
#4  0x0000000000637421 in ?? ()
#5  0x0000000000638c72 in ?? ()
#6  0x00000000006962fc in ?? ()
#7  0x00000000007ea26e in ?? ()
#8  0x0000000000802753 in ?? ()
#9  0x0000000000802915 in ?? ()
#10 0x0000000000800495 in ?? ()
#11 0x0000000000a1e85c in ?? ()
#12 0x0000000000a21dee in ?? ()
#13 0x00000000010996cd in ?? ()
#14 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
#15 0x0000000000411c69 in ?? ()
(gdb) i r rax
rax            0x4141414141414141  0x4141414141414141
```

Timeline

2020-11-02 - Vendor Disclosure

2021-01-19 - Vendor Patched

2021-02-03 - Public Release

CREDIT

Discovered by a member of Cisco Talos.