Talos Vulnerability Report

TALOS-2020-1191

SoftMaker Office PlanMaker Document Record 0x8010 out-of-bounds write vulnerability

FEBRUARY 3, 2021

CVE NUMBER

CVE-2020-13580

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 explicitly trust a length from a particular record type and use it to write a 16-bit null relative to a buffer allocated on the stack. Due to a lack of bounds-checking on this value, this can allow an attacker to write to memory outside of the buffer and controllably corrupt memory. This can allow an attacker to earn 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-787 - Out-of-bounds Write

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 parsing for record type 0x8010 is done by the following function. Thie function first stores the pointer to the record into a variable within the frame, and then uses it as [6] to calculate a pointer to the record's contents. Once a pointer to the records contents has been assigned, at [7] a uint16_t will be read from the record and also stored in the frame. This is done so that later at [8], the function can check to ensure the uint16_t is under 10 bytes. If the value is larger than 10, the function will clamp the value to the maximum possible size. This uint16_t is used to describe the length of a string located in the frame and is checked to ensure that a buffer overflow will not occur. The clamped value will then be stored into a variable.

```
0x67a6e1:    push   %rbp
0x67a6e2:    mov    %rsp,%rbp
0x67a6e5:    sub    $0x70,%rsp
0x67a6e9:    mov    %rdi,-0x68(%rbp)
0x67a6ed:    mov    %rsi,-0x70(%rbp)     ; record contents
0x67a6f1:    mov    %fs:0x28,%rax
0x67a6fa:    mov    %rax,-0x8(%rbp)
...
0x67a717:    mov    -0x70(%rbp),%rax     ; take pointer to record
0x67a71b:    add    $0x4,%rax            ; shift past record type and length
0x67a71f:    mov    %rax,-0x30(%rbp)     ; [6] store pointer to record's contents
0x67a723:    mov    $0x10,%eax
0x67a728:    add    $0x4,%eax
0x67a72b:    mov    %eax,-0x54(%rbp)
...
0x67a77c:    mov    -0x30(%rbp),%rax     ; pointer to record's contents
0x67a780:    movzwl 0x2(%rax),%eax       ; [7] read uint16_t
0x67a784:    movzwl %ax,%eax
0x67a787:    mov    %eax,-0x50(%rbp)     ; store into frame
0x67a78a:    cmpl   $0x0,-0x50(%rbp)     ; ensure its non-zero
0x67a78e:    setne  %al
0x67a791:    test   %al,%al
0x67a793:    je     0x67a84f
...
0x67a7eb:    mov    -0x50(%rbp),%eax     ; read uint16
0x67a7ee:    cltq
0x67a7f0:    mov    $0xa,%edx            ; size is 0xa
0x67a7f5:    cmp    %rdx,%rax            ; [8] ensure that uint16 is not larger than 0xa
0x67a7f8:    jb     0x67a801
0x67a7fa:    mov    $0xa,%eax            ; assign the maximum size
0x67a7ff:    jmp    0x67a804
```

Despite the application checking the length to ensure it's not larger than the buffer that it is referencing, the application misakenly re-read the uint16_t from the record at [9]. Due to the variable containing the clamped value not being used, this value is completely user-controllable and unconstrained. At [10], this length is used to write 0x0000 relative to a buffer on the stack after multiplying it by 2. Due to the buffer on the stack being of 0x20 bytes, if an attacker specifies a uint16_t larger than 0x10, the instruction at [10] will write past its target's boundaries. This function contains an 8-byte stack canary within its frame, so therefore an attacker must specify a length of at least 0x14 to ensure the canary isn't affected.

```
0x67a835:    mov    -0x30(%rbp),%rax         ; pointer to record's contents
0x67a839:    movzwl 0x2(%rax),%eax           ; [9] read uint16_t
0x67a83d:    movzwl %ax,%eax
0x67a840:    cltq
0x67a842:    movw   $0x0,-0x20(%rbp,%rax,2)  ; [10] write 0x0000 to -0x20(%rbp) + 2*%rax
0x67a849:    mov    -0x50(%rbp),%eax
0x67a84c:    add    %eax,-0x54(%rbp)
0x67a84f:    movzwl -0x20(%rbp),%eax
0x67a853:    test   %ax,%ax
0x67a856:    je     0x67a9d8
```

Crash Information

The provided proof of concept sets the uint16_t to 0x1c, when multiplied by 2 this results in writing a null byte 0x28 bytes past the stack variable at -0x20(%ebp). This value skips over the stack canary and directly writes to the saved $pc on the stack. This results in the least-significant 16-bits being set to 0x0000.

```
Thread 1 "planmaker" received signal SIGSEGV, Segmentation fault.
0x0000000000680004 in ?? ()
(gdb) bt
#0  0x0000000000680004 in ?? ()
#1  0x0000000000637421 in ?? ()
#2  0x0000000000638c72 in ?? ()
#3  0x00000000006962fc in ?? ()
#4  0x00000000007ea26e in ?? ()
#5  0x0000000000802753 in ?? ()
#6  0x0000000000802915 in ?? ()
#7  0x0000000000800495 in ?? ()
#8  0x0000000000a1e85c in ?? ()
#9  0x0000000000a21dee in ?? ()
#10 0x00000000010996cd in ?? ()
#11 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
#12 0x0000000000411c69 in ?? ()

(gdb) x/i $pc
=> 0x680004:    mov    %edx,0x4c0c(%rax)
```

Timeline

2020-11-02 - Vendor Disclosure
2021-01-19 - Vendor Patched
2021-02-03 - Public Release

CREDIT

Discovered by a member of Cisco Talos.

© 2022 Cisco Systems, Inc. and/or its affiliates. All rights reserved. View our Privacy Policy.