Talos Vulnerability Report

TALOS-2020-1197

SoftMaker Office PlanMaker Excel document record 0x00fc memory corruption vulnerability

FEBRUARY 3, 2021

CVE NUMBER

CVE-2020-13586

Summary

A memory corruption vulnerability exists in the Excel Document SST Record 0x00fc functionality of SoftMaker Software GmbH SoftMaker Office PlanMaker 2021 (Revision 1014). A specially crafted malformed file can lead to a heap buffer overflow. An attacker can provide a malicious file 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 that allows the user to write text documents, create spreadsheets, design presentations and more. The SoftMaker Office suite supports a variety of common office file formats, as well as other internal formats that the user may choose to use when performing their necessary work.

The PlanMaker application of SoftMaker's suite is designed as an all-around spreadsheet tool, and supports a number of features that allow it to remain competitive with similar office suites that are developed by its competitors. This application includes a number of parsers that enable the user to interact with a variety of document types or templates that are common within this type of software. One supported file format, which is relevant to the vulnerability described within this document, is the Microsoft Excel file format. This format is based on Microsoft's Compound Document file format and is primarily contained within either the "Workbook" stream for later versions of the format, or the "Book" stream for earlier versions.

When opening up a Microsoft Excel Document, the following function will be executed in order to load the document. Before the document is loaded, the application will open up the file temporarily in order to fingerprint the document and determine which handler is used to load the document's contents. At [1], the application will call a function that is responsible for parsing the different streams that may be found within the document.

```
0x7ea017:    push   %rbp
0x7ea018:    mov    %rsp,%rbp
0x7ea01b:    sub    $0x260,%rsp
0x7ea022:    mov    %rdi,-0x248(%rbp)
0x7ea029:    mov    %rsi,-0x250(%rbp)   ; object
0x7ea030:    mov    %rdx,-0x258(%rbp)   ; document path
0x7ea037:    mov    %ecx,-0x25c(%rbp)
...
0x7ea24b:    mov    -0x234(%rbp),%ecx
0x7ea251:    mov    -0x258(%rbp),%rdx   ; document path
0x7ea258:    mov    -0x250(%rbp),%rsi   ; object
0x7ea25f:    mov    -0x248(%rbp),%rax
0x7ea266:    mov    %rax,%rdi
0x7ea269:    callq  0x695c7c            ; [1]
0x7ea26e:    test   %eax,%eax
0x7ea270:    setne  %al
0x7ea273:    test   %al,%al
0x7ea275:    je     0x7ea2a3
```

Eventually the following function will be called from an array of functions used to handle the different record types within the workbook stream belonging to the Microsoft Excel document. This function is given a handle to the document stream as one of its parameters in order to read records from the stream. After initializing some of the local variables within the function, the loop at [2] will be executed. This loop is directly responsible for reading a record from the stream at [3], and then looking at the record type to in order to determine how to actually parse the record.

```
0x634eeb:    push   %rbp
0x634eec:    mov    %rsp,%rbp
0x634eef:    sub    $0x90,%rsp
0x634ef6:    mov    %rdi,-0x78(%rbp)    ; object
0x634efa:    mov    %rsi,-0x80(%rbp)    ; document stream
0x634efe:    mov    %edx,-0x84(%rbp)    ; flags
0x634f04:    mov    %fs:0x28,%rax
...
0x634f92:    mov    -0x78(%rbp),%rax    ; [2] beginning of loop
0x634f96:    mov    0x88(%rax),%eax
0x634f9c:    and    $0xff000000,%eax
0x634fa1:    test   %eax,%eax
0x634fa3:    jne    0x635e02            ; exit loop
...
0x634fa9:    mov    -0x40(%rbp),%r9     ; record data
0x634fad:    lea    -0x64(%rbp),%r8     ; result code
0x634fb1:    lea    -0x6e(%rbp),%rcx    ; result record length
0x634fb5:    lea    -0x6c(%rbp),%rdx    ; result record type
0x634fb9:    mov    -0x80(%rbp),%rsi    ; stream object
0x634fbd:    mov    -0x78(%rbp),%rax    ; object
0x634fc1:    sub    $0x8,%rsp
0x634fc5:    lea    -0x68(%rbp),%rdi
0x634fc9:    push   %rdi
0x634fca:    mov    %rax,%rdi           ; object
0x634fcd:    callq  0x61e4a8            ; [3] parse the next record from the stream
0x634fd2:    add    $0x10,%rsp
0x634fd6:    mov    %rax,-0x40(%rbp)    ; record data from stream
0x634fda:    cmpq   $0x0,-0x40(%rbp)
0x634fdf:    je     0x635e01
...
0x635dfb:    nop
0x635dfc:    jmpq   0x634f92            ; [2] continue loop
```

Once successfully parsing the record within each iteration of the loop, the loop will use the record type in order to determine which handler to use for parsing the record's contents at [4]. The vulnerability described by this advisory is for record type 0x00fc which represents the `SST` record which contains the Shared String Table for the document. After reading the current record's length at [5] and resuming the search at [6]. The function will branch to directly to the block of code at [7] which is responsible for dispatching execution to a function responsible for parsing the contents of the `SST` record. At [8], the data for the entire record and the stream object containing the record are passed as parameters to the function that will parse the Shared String Table.

```
0x634ff1:    movzwl -0x6c(%rbp),%eax    ; result record type
0x634ff5:    movzwl %ax,%eax
0x634ff8:    cmp    $0x43,%eax
0x634ffb:    je     0x635017
0x634ffd:    cmp    $0x43,%eax
0x635000:    jg     0x635009            ; [4] look for record type
...
0x635009:    cmp    $0x92,%eax
0x63500e:    je     0x635033
0x635010:    cmp    $0x231,%eax
0x635015:    jne    0x63504e            ; [4] look for record type
...
0x63504e:    movzwl -0x6c(%rbp),%eax
0x635052:    cmp    $0xa,%ax
0x635056:    jne    0x635064            ; [4] look for record type
...
0x635064:    mov    -0x40(%rbp),%rax    ; record contents
0x635068:    add    $0x2,%rax
0x63506c:    movzwl (%rax),%eax         ; [5] read record length
0x63506f:    movzwl %ax,%eax
0x635072:    mov    %eax,-0x44(%rbp)    ; [5] store record length
...
0x635075:    movzwl -0x6c(%rbp),%eax    ; result record type
0x635079:    movzwl %ax,%eax
0x63507c:    cmp    $0xc1,%eax
0x635081:    je     0x635588
0x635087:    cmp    $0xc1,%eax7
0x63508c:    jg     0x6351e8            ; [6] look for record type
...
0x6351e8:    cmp    $0x161,%eax
0x6351ed:    je     0x6355ad
0x6351f3:    cmp    $0x161,%eax
0x6351f8:    jg     0x6352ab            ; [6] look for record type
0x6351fe:    cmp    $0xe2,%eax
0x635203:    jg     0x635258
...
0x635258:    cmp    $0xfc,%eax
0x63525d:    je     0x635b84            ; [7] found record type 0x00fc
...
0x635b84:    lea    -0x68(%rbp),%rcx
0x635b88:    mov    -0x40(%rbp),%rdx    ; record data from stream
0x635b8c:    mov    -0x80(%rbp),%rsi    ; stream object
0x635b90:    mov    -0x78(%rbp),%rax    ; object
0x635b94:    mov    %rax,%rdi
0x635b97:    callq  0x6217cc            ; [8] parse record 0x00fc
0x635b9c:    mov    %rax,-0x40(%rbp)
0x635ba0:    jmpq   0x635dfc
```

Once inside the function responsible for parsing the `SST` record and storing its parameters into the function's frame on the stack, at [9] the function will read a `uint32_t` from offset +8 of the record as the `cstUnique` field and store into into a local variable. Immediately afterwards at [10], a `uint16_t` will be read from offset +2 of the record which contains the record's length. In order to store the string table that will be read from the current record and all of the records that follow it, the application will allocate a constant size of 0x6060 bytes at [11]. At [12], the application will explicitly trust the `uint16_t` that was read as the length, and use it to copy the record's contents into the statically sized heap buffer that was just allocated. Due to the maximum size of a `uint16_t` being 0xffff, and the size of the heap-buffer being 0x6060, this allows for a heap-based buffer overflow to occur if the record length is larger than 0x6060. This allows for memory corruption which can lead to code execution under the context of the application.

```
0x6217cc:    push   %rbp
0x6217cd:    mov    %rsp,%rbp
0x6217d0:    push   %rbx
0x6217d1:    sub    $0x128,%rsp
0x6217d8:    mov    %rdi,-0x118(%rbp)   ; object
0x6217df:    mov    %rsi,-0x120(%rbp)   ; stream object
0x6217e6:    mov    %rdx,-0x128(%rbp)   ; record data from stream
0x6217ed:    mov    %rcx,-0x130(%rbp)
...
0x621822:    mov    -0x128(%rbp),%rax   ; record data from stream
0x621829:    mov    0x8(%rax),%eax      ; [9] read uint32_t from recordData + 8
0x62182c:    mov    %eax,-0xc0(%rbp)    ; store it
...
0x621832:    mov    -0x128(%rbp),%rax   ; record data from stream
0x621839:    add    $0x2,%rax           ; shift pointer to recordData + 2
0x62183d:    movzwl (%rax),%eax         ; [10] read uint16_t from recordData + 2
0x621840:    movzwl %ax,%eax
0x621843:    mov    %eax,-0xec(%rbp)    ; store record's uint16_t length
...
0x621884:    mov    -0x118(%rbp),%rax
0x62188b:    mov    0x48(%rax),%rax
0x62188f:    mov    $0x6060,%esi        ; static size used for allocation
0x621894:    mov    %rax,%rdi
0x621897:    callq  0xab7a01            ; [11] allocate memory using static size
0x62189c:    mov    %rax,-0xa8(%rbp)
...
0x6218c2:    mov    -0xec(%rbp),%eax    ; uint16_t record length trusted from file
0x6218c8:    lea    -0x8(%rax),%edx     ; memcpy length
0x6218cb:    mov    -0x128(%rbp),%rax   ; record data from stream
0x6218d2:    lea    0xc(%rax),%rcx
0x6218d6:    mov    -0xa8(%rbp),%rax    ; heap buffer that was allocated
0x6218dd:    mov    %rcx,%rsi           ; memcpy source
0x6218e0:    mov    %rax,%rdi           ; memcpy destination
0x6218e3:    callq  0xab9e39            ; [12] memcpy that triggers buffer overflow due to trusted uint16_t length
0x6218e8:    subl   $0x8,-0xec(%rbp)
```

Crash Information

When running the PlanMaker application within a debugger, set a breakpoint on the address of the allocation at 0x621897 and then the call to `memcpy` at address 0x6218e3. Once opening the provided proof-of-concept, the breakpoint at the allocation should interrupt execution of the application.

```
(gdb) bp 621897
Breakpoint 4 at 0x621897
(gdb) bp 6218e3
Breakpoint 5 at 0x6218e3
(gdb) r
Starting program: /usr/share/office2021/planmaker
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7fffefcd3700 (LWP 2679)]
[New Thread 0x7fffef4d2700 (LWP 2680)]
...
[Detaching after vfork from child process 3111]
[Detaching after vfork from child process 3113]
[Detaching after vfork from child process 3115]
[New Thread 0x7fffc65fd700 (LWP 3117)]

Thread 1 "planmaker" hit Breakpoint 4, 0x0000000000621897 in ?? ()
(gdb) h

-=[registers]=-
[rax: 0x0000000002ed23f0] [rbx: 0x0000000002efa9c0] [rcx: 0x0000000002d0e880]
[rdx: 0x0000000002d0e880] [rsi: 0x0000000000006060] [rdi: 0x0000000002ed23f0]
[rsp: 0x00007fffffffbb80] [rbp: 0x00007fffffffbcb0] [ pc: 0x0000000000621897]
[ r8: 0x0000000000000001] [ r9: 0x00000000000001a0] [r10: 0x0000000001b82010]
[r11: 0x0000000002d0e600] [r12: 0x0000000000000001] [r13: 0x00007fffffffeb10]
[r14: 0x0000000000000000] [r15: 0x0000000000000000] [efl: 0x00000246]
[flags: +ZF -SF -OF -CF -DF +PF -AF +IF R1]

-=[stack]=-
7fffffffbb80 | 00007fffffffbce8 0000000002d06710 | .........g......
7fffffffbb90 | 0000000002dff650 00007fffffffc080 | P...............
7fffffffbba0 | 0000000002d0e713 0000000002ef54d0 | .........T......
7fffffffbbb0 | 00007fff000000c8 0000011300000000 | ................

-=[disassembly]=-
=> 0x621897:    callq  0xab7a01
   0x62189c:    mov    %rax,-0xa8(%rbp)
   0x6218a3:    mov    -0x118(%rbp),%rax
   0x6218aa:    mov    $0x68,%edx
   0x6218af:    mov    $0x0,%esi
   0x6218b4:    mov    %rax,%rdi
```

Once the breakpoint at address 0x621897 is reached, dumping out its parameters shows that the size that will be used for the allocation is 0x6060 and stored within the `%esi` register. Stepping over the call to the allocator will then result in a pointer to the allocated heap buffer being returned.

```
(gdb) i r rdi esi
rdi            0x2ed23f0           0x2ed23f0
esi            0x6060              0x6060
(gdb) n
0x000000000062189c in ?? ()

(gdb) i r rax
rax            0x2d33670           0x2d33670
(gdb)
```

Continuing execution after the allocation has been made will then result in the next breakpoint interrupting the execution of the application before the data from the record's contents is copied into the heap buffer using the `memcpy` function. The parameters for the call to `memcpy` are stored within the `%rdi`, `%rsi`, and `%edx` registers. Printing out the state of these registers shows that `%rdi` is pointing to the prior result that was returned from the heap allocation, and the length in the `%edx` register is larger than the size that was used to allocate said heap buffer.

```
(gdb) c
Continuing.

Thread 1 "planmaker" hit Breakpoint 5, 0x00000000006218e3 in ?? ()
(gdb) h
-=[registers]=-
[rax: 0x0000000002d33670] [rbx: 0x0000000002efa9c0] [rcx: 0x0000000002d0671c]
[rdx: 0x0000000000007ff7] [rsi: 0x0000000002d0671c] [rdi: 0x0000000002d33670]
[rsp: 0x00007fffffffbb80] [rbp: 0x00007fffffffbcb0] [ pc: 0x00000000006218e3]
[ r8: 0x0000000000000000] [ r9: 0x0000000000006060] [r10: 0x0000000001b82010]
[r11: 0x0000000002d0e600] [r12: 0x0000000000000001] [r13: 0x00007fffffffeb10]
[r14: 0x0000000000000000] [r15: 0x0000000000000000] [efl: 0x00000246]
[flags: +ZF -SF -OF -CF -DF +PF -AF +IF R1]

-=[stack]=-
7fffffffbb80 | 00007fffffffbce8 0000000002d06710 | .........g......
7fffffffbb90 | 0000000002dff650 00007fffffffc080 | P...............
7fffffffbba0 | 0000000002d0e713 0000000002ef54d0 | .........T......
7fffffffbbb0 | 00007fff000000c8 0000011300000000 | ................

-=[disassembly]=-
=> 0x6218e3:    callq  0xab9e39
   0x6218e8:    subl   $0x8,-0xec(%rbp)
   0x6218ef:    movl   $0x0,-0xf4(%rbp)
   0x6218f9:    movl   $0x0,-0xe4(%rbp)
   0x621903:    callq  0x10991a5
   0x621908:    mov    %eax,-0xe8(%rbp)

(gdb) i r rdi rsi edx
rdi            0x2d33670           0x2d33670
rsi            0x2d0671c           0x2d0671c
edx            0x7ff7              0x7ff7
```

If we dump out the memory that is pointed to by the `%rsi` register, the contents of the record as contained within the file is displayed. This record's contents will be used to corrupt memory when the call to `memcpy` is executed.

```
(gdb) db $rsi
2d0671c | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 31 00 | ..............1.
2d0672c | 16 00 dc 00 00 00 08 00 90 01 00 00 00 02 00 32 | ...............2
2d0673c | 07 43 61 6c 69 62 72 69 31 00 16 00 dc 00 00 00 | .Calibri1.......
2d0674c | 08 00 90 01 00 00 00 02 00 32 07 43 61 6c 69 62 | .........2.Calib
2d0675c | 72 69 31 00 16 00 dc 00 00 00 08 00 90 01 00 00 | ri1.............
2d0676c | 00 02 00 32 07 43 61 6c 69 62 72 69 31 00 16 00 | ...2.Calibri1...
```

By stepping over the call to `memcpy`, the memory corruption will be made to occur. Once resuming execution, the application will continue to execute despite the heap memory after the 0x6060-sized allocation that was used for the `SST` record's contents was corrupted.

```
(gdb) n
0x00000000006218e8 in ?? ()

-=[registers]=-
[rax: 0x0000000002d33670] [rbx: 0x0000000002efa9c0] [rcx: 0x0000000000000000]
[rdx: 0x0000000000007ff7] [rsi: 0x0000000002d0e713] [rdi: 0x0000000002d3b667]
[rsp: 0x00007fffffffbb80] [rbp: 0x00007fffffffbcb0] [ pc: 0x00000000006218e8]
[ r8: 0x0000000000000000] [ r9: 0x0000000002d0e713] [r10: 0x0000000001b82010]
[r11: 0x0000000002d0e600] [r12: 0x0000000000000001] [r13: 0x00007fffffffeb10]
[r14: 0x0000000000000000] [r15: 0x0000000000000000] [efl: 0x00000212]
[flags: -ZF -SF -OF -CF -DF -PF +AF +IF R1]

-=[stack]=-
7fffffffbb80 | 00007fffffffbce8 0000000002d06710 | .........g......
7fffffffbb90 | 0000000002dff650 00007fffffffc080 | P...............
7fffffffbba0 | 0000000002d0e713 0000000002ef54d0 | .........T......
7fffffffbbb0 | 00007fff000000c8 0000011300000000 | ................

-=[disassembly]=-
=> 0x6218e8:    subl   $0x8,-0xec(%rbp)
   0x6218ef:    movl   $0x0,-0xf4(%rbp)
   0x6218f9:    movl   $0x0,-0xe4(%rbp)
   0x621903:    callq  0x10991a5
   0x621908:    mov    %eax,-0xe8(%rbp)
   0x62190e:    movl   $0x1,-0x104(%rbp)

(gdb) c
Continuing.
```

Due to the heap of the application being in a corrupted state, if the application attempts to use this memory this can result in undefined behaviour. Through proper manipulation of the Excel Document parser's allocations, the corruption of the data after the 0x6060 memory chunk can allow an attacker to earn code execution within the context of the application.

```
Thread 1 "planmaker" received signal SIGSEGV, Segmentation fault.
0x0000000000ab7aac in ?? ()
(gdb) x/i $pc
=> 0xab7aac:    mov    0x18(%rax),%rax
(gdb) i r rax
rax            0x3f3f3f3f3f3f3f3f  0x3f3f3f3f3f3f3f3f
(gdb) bt 8
#0  0x0000000000ab7aac in ?? ()
#1  0x000000000061eb65 in ?? ()
#2  0x0000000000634fd2 in ?? ()
#3  0x0000000000637652 in ?? ()
#4  0x0000000000638c72 in ?? ()
#5  0x00000000006962fc in ?? ()
#6  0x00000000007ea26e in ?? ()
#7  0x0000000000802753 in ?? ()
(More stack frames follow...
(gdb)
```

Timeline

2020-11-12 - 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-2020-1192

NEXT REPORT TALOS-2020-1210

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