Talos Vulnerability Report

TALOS-2020-1162

SoftMaker Office TextMaker Document Record 0x003f integer conversion vulnerability

JANUARY 5, 2021

CVE NUMBER

CVE-2020-13545

Summary

An exploitable signed conversion vulnerability exists in the TextMaker document parsing functionality of SoftMaker Office 2021's TextMaker application. A specially crafted document can cause the document parser to miscalculate a length used to allocate a buffer, later upon usage of this buffer the application will write outside its bounds resulting in a heap-based memory corruption. An attacker can entice the victim to open a document to trigger this vulnerability.

Tested Versions

SoftMaker Software GmbH SoftMaker Office TextMaker 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-196 - Unsigned to Signed Conversion Error

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 TextMaker component of SoftMaker's suite is designed as an all-around word-processing 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 TextMaker Document, and will typically have the extension ".tmd" when saved as a file.

When the application needs to read a file in order to allow the user to interact with the desired document, it will load the document by executing the following function. This function will take an object containing information about the document and the path to load the document from its parameters. After determining which particular flags are set, the function call at [1] will be made in order to determine what type of document the file is.

```
0x7c2ef0:       push   %rbp
0x7c2ef1:       mov    %rsp,%rbp
0x7c2ef4:       sub    $0x260,%rsp
0x7c2efb:       mov    %rdi,-0x248(%rbp)    ; documentObject
0x7c2f02:       mov    %rsi,-0x250(%rbp)
0x7c2f09:       mov    %rdx,-0x258(%rbp)    ; path name
0x7c2f10:       mov    %ecx,-0x25c(%rbp)    ; flags
...
0x7c314a:       mov    -0x234(%rbp),%edx    ; flags
0x7c3150:       mov    -0x258(%rbp),%rcx    ; path name
0x7c3157:       mov    -0x248(%rbp),%rax    ; documentObject
0x7c315e:       mov    %rcx,%rsi
0x7c3161:       mov    %rax,%rdi
0x7c3164:       callq  0x60b4b8             ; [1] ReadDocument
0x7c3169:       test   %eax,%eax
0x7c316b:       setne  %al
0x7c316e:       test   %al,%al
0x7c3170:       je     0x7c319e
```

First the application will take its parameters consisting of the object containing the document, and the path the file to read the document from onto the stack. The path will then be passed to the function call at [2] which is responsible for fingerprinting the document to try and identify which document parser to use. Upon returning, the function call at address 0x60b703 will be made to actually read the file.

```
0x60b4b8:       push   %rbp
0x60b4b9:       mov    %rsp,%rbp
0x60b4bc:       sub    $0xbb0,%rsp
0x60b4c3:       mov    %rdi,-0xb98(%rbp)    ; document object
0x60b4ca:       mov    %rsi,-0xba0(%rbp)    ; document path
0x60b4d1:       mov    %edx,-0xba4(%rbp)    ; flags
...
0x60b654:       lea    -0x640(%rbp),%rax    ; path
0x60b65b:       mov    %rax,%rdi
0x60b65e:       callq  0x627cb8             ; [2] \ Fingerprint the document
0x60b663:       mov    %eax,-0xb6c(%rbp)
0x60b669:       movl   $0x1,-0xb7c(%rbp)
...
0x60b6d2:       mov    -0xb84(%rbp),%r8d
0x60b6d9:       mov    -0xba4(%rbp),%edi    ; flags
0x60b6df:       lea    -0x640(%rbp),%rcx    ; document path
0x60b6e6:       mov    -0xb70(%rbp),%edx
0x60b6ec:       mov    -0xb58(%rbp),%rsi    ; FILE*
0x60b6f3:       mov    -0xb98(%rbp),%rax    ; document object
0x60b6fa:       mov    %r8d,%r9d
0x60b6fd:       mov    %edi,%r8d
0x60b700:       mov    %rax,%rdi
0x60b703:       callq  0x6273fe             ; [3] read the TextMaker document
0x60b708:       test   %eax,%eax
0x60b70a:       je     0x60c2d1
```

To fingerprint the file, the application will first open up the file at [4]. Following this at [5], the application then reads 12 bytes from its header to take a sample of the bytes near the beginning of the file. This is then used by the application in order to identify which document type the user is trying to open. The first signature, however, is for the `*.tmd` (TextMaker Document) file format. In order to verify that the signature corresponds to a TextMaker Document, the first 32-bits are read from the file at [5]. These bits are then compared against the integer, 0xff00564d. After verifying the initial 32-bits, the application will then skip over 16-bits which represent an offset to the index table which will be described later, and then check if the 16-bits that follow are either of the values 0x000e or 0x000f.

```
0x627cb8:       push   %rbp
0x627cb9:       mov    %rsp,%rbp
0x627cbc:       sub    $0x50,%rsp
0x627cc0:       mov    %rdi,-0x48(%rbp)     ; path
...
0x627cda:       mov    -0x48(%rbp),%rax
0x627cde:       mov    $0x16ba78a,%esi
0x627ce3:       mov    %rax,%rdi
0x627ce6:       callq  0x12f51b7            ; [3] open up file as a FILE*
0x627ceb:       mov    %rax,-0x38(%rbp)
 ...
0x627cff:       mov    $0xc,%edx            ; length
0x627d04:       lea    -0x30(%rbp),%rcx     ; buffer containing header to fingerprint
0x627d08:       mov    -0x38(%rbp),%rax
0x627d0c:       mov    %rcx,%rsi            ; destination
0x627d0f:       mov    %rax,%rdi            ; FILE*
0x627d12:       callq  0x62733d             ; [4]
0x627d17:       test   %eax,%eax
0x627d19:       sete   %al
...
0x627d24:       mov    -0x30(%rbp),%eax     ; [5] read first uint32_t from file
0x627d27:       cmp    $0xff00564d,%eax
0x627d2c:       jne    0x627d49
0x627d2e:       movzwl -0x2a(%rbp),%eax     ; [5] read uint16_t from offset +6
0x627d32:       cmp    $0xe,%ax
0x627d36:       je     0x627d42
0x627d38:       movzwl -0x2a(%rbp),%eax     ; [5] read uint16_t from offset +6
0x627d3c:       cmp    $0xf,%ax
0x627d40:       jne    0x627d49
...
0x627dfc:    leaveq
0x627dfd:    retq
```

Upon using the fingerprint to determine the file format type, the application will return to the caller. As previously mentioned, the function call at [6] will be used to actually parse the TextMaker Document file format.

```
0x60b6d2:       mov    -0xb84(%rbp),%r8d
0x60b6d9:       mov    -0xba4(%rbp),%edi    ; flags
0x60b6df:       lea    -0x640(%rbp),%rcx    ; document path
0x60b6e6:       mov    -0xb70(%rbp),%edx
0x60b6ec:       mov    -0xb58(%rbp),%rsi    ; FILE*
0x60b6f3:       mov    -0xb98(%rbp),%rax    ; document object
0x60b6fa:       mov    %r8d,%r9d
0x60b6fd:       mov    %edi,%r8d
0x60b700:       mov    %rax,%rdi
0x60b703:       callq  0x6273fe             ; [6] read the TextMaker document
0x60b708:       test   %eax,%ea6
0x60b70a:       je     0x60c2d1
```

When reading the document, the application will re-read the 12-byte header in order to extract the 16-bit field that was previously skipped over during the fingerprint process. As the stream was previously opened and passed to this function, it is used to seek to the beginning of the file at [7]. Afterwards at [8] the same 12 bytes that container the header that was used during fingerprinting are read. At offset +4 of this header, a `uint16_t` is read which is used as a file offset. This 16-bit offset is then passed to the function call at [9] to seek the stream to the index table for the document. Once the stream's offset has been set correctly, the function call at [10] is made which will begin to parse the index table of the document.

```
0x6273fe:       push   %rbp
0x6273ff:       mov    %rsp,%rbp
0x627402:       sub    $0x60,%rsp
0x627406:       mov    %rdi,-0x38(%rbp)     ; document object
0x62740a:       mov    %rsi,-0x40(%rbp)     ; stream
0x62740e:       mov    %edx,-0x44(%rbp)
0x627411:       mov    %rcx,-0x50(%rbp)     ; document path
0x627415:       mov    %r8d,-0x48(%rbp)
0x627419:       mov    %r9d,-0x54(%rbp)
 ...
0x627437:       mov    -0x40(%rbp),%rax
0x62743b:       mov    $0x0,%edx            ; SEEK_SET
0x627440:       mov    $0x0,%esi
0x627445:       mov    %rax,%rdi
0x627448:       callq  0x410fe0 <fseek@plt> ; [7] seek to beginning of file
 ...
0x62744d:       mov    $0xc,%edx            ; length
0x627452:       lea    -0x20(%rbp),%rcx     ; destination
0x627456:       mov    -0x40(%rbp),%rax     ; FILE*
0x62745a:       mov    %rcx,%rsi
0x62745d:       mov    %rax,%rdi
0x627460:       callq  0x62733d             ; [8] fread
0x627465:       test   %eax,%eax
0x627467:       sete   %al
0x62746a:       test   %al,%al
0x62746c:       je     0x627484
...
0x627484:       movzwl -0x1c(%rbp),%eax     ; uint16_t offset
0x627488:       movzwl %ax,%ecx
0x62748b:       mov    -0x40(%rbp),%rax
0x62748f:       mov    $0x0,%edx            ; SEEK_SET
0x627494:       mov    %rcx,%rsi            ; offset
0x627497:       mov    %rax,%rdi            ; FILE*
0x62749a:       callq  0x410fe0 <fseek@plt> ; [9] seek to uint16_t
...
0x6274a7:       mov    -0x50(%rbp),%rdx     ; filename
0x6274ab:       mov    -0x40(%rbp),%rsi     ; stream
0x6274af:       mov    -0x38(%rbp),%rax     ; document object
0x6274b3:       mov    %rax,%rdi
0x6274b6:       callq  0x626b0f             ; [10] parse index table
0x6274bb:       mov    %eax,-0x24(%rbp)
```

Before parsing the index table containing all of the records that compose the TextMaker Document, the function call at [11] is used to read 10-bytes from the current position of the file. Then at [12], 32-bits are read and used to verify the signature of the index table by comparing it with the integer 0x314592d which corresponds to the value for π. After validating the signature, the application will read two 16-bit integers from the file which correspond to the version. At [14], both version components are read and then combined into a 12-bit version. This version is then checked to ensure it’s between the values 310 and 325 which are the versions that are supported by the application.

```
0x626b0f:       push   %rbp
0x626b10:       mov    %rsp,%rbp
0x626b13:       sub    $0x180,%rsp
0x626b1a:       mov    %rdi,-0x168(%rbp)    ; document object
0x626b21:       mov    %rsi,-0x170(%rbp)    ; FILE*
0x626b28:       mov    %rdx,-0x178(%rbp)    ; document path
0x626b2f:       mov    %ecx,-0x17c(%rbp)    ; flags
...
0x626c3e:       mov    $0xa,%edx            ; length
0x626c43:       lea    -0x130(%rbp),%rcx    ; buffer
0x626c4a:       mov    -0x170(%rbp),%rax    ; FILE*
0x626c51:       mov    %rcx,%rsi
0x626c54:       mov    %rax,%rdi
0x626c57:       callq  0x62738a             ; [11] read 0xa bytes from file
0x626c5c:       test   %eax,%eax
0x626c5e:       sete   %al
...
0x626c69:       mov    -0x130(%rbp),%eax    ; [12] read uint32_t and check signature
0x626c6f:       cmp    $0x3141592d,%eax
0x626c74:       je     0x626c98
...
0x626c98:       movzwl -0x12c(%rbp),%eax    ; [13] read uint16_t for major component of version
0x626c9f:       movzwl %ax,%eax
0x626ca2:       imul   $0x64,%eax,%edx
0x626ca5:       movzwl -0x12a(%rbp),%eax    ; [13] read uint16_t for minor component of version
0x626cac:       movzwl %ax,%eax
0x626caf:       add    %eax,%edx
0x626cb1:       mov    -0x168(%rbp),%rax
0x626cb8:       mov    %edx,0x38(%rax)      ; [13] store version
...
0x626cbb:       mov    -0x168(%rbp),%rax    ; [14] read version
0x626cc2:       mov    0x38(%rax),%eax
0x626cc5:       cmp    $0x136,%eax          ; [14] compare against 310
0x626cca:       je     0x6272e2
...
0x626cd0:       mov    -0x168(%rbp),%rax    ; [14] read version
0x626cd7:       mov    0x38(%rax),%eax
0x626cda:       cmp    $0x145,%eax          ; [14] compare against 325
0x626cdf:       jle    0x626d03
```

Once the version has been verified, the index table will be allocated. This is done at [15] by first reading the number of records from the 10-byte buffer, and then multiplying by 8. Afterwards the resulting size will be passed to the function call at [16] to round the size and allocate space for it. After the space for the index table has been successfully allocated, the call at [17] will read data from the file into it.

```
0x626d03:       movzwl -0x128(%rbp),%eax    ; [15] read number of records from index header
0x626d0a:       movzwl %ax,%eax
0x626d0d:       mov    $0x8,%edx
0x626d12:       imul   %edx,%eax            ; [15] multiply by 8
0x626d15:       mov    %eax,-0x154(%rbp)
...
0x626d1b:       mov    -0x154(%rbp),%edx    ; [16] use size
0x626d21:       mov    -0x168(%rbp),%rax    ; document object
0x626d28:       mov    %edx,%esi
0x626d2a:       mov    %rax,%rdi
0x626d2d:       callq  0x1267124            ; [16] allocate space for index table
0x626d32:       mov    %rax,-0x150(%rbp)    ; allocated index table buffer
...
0x626d4c:       mov    -0x154(%rbp),%edx    ; index table size
0x626d52:       mov    -0x150(%rbp),%rcx    ; index table buffer
0x626d59:       mov    -0x170(%rbp),%rax    ; FILE*
0x626d60:       mov    %rcx,%rsi
0x626d63:       mov    %rax,%rdi
0x626d66:       callq  0x62738a             ; [17] read index table into buffer
0x626d6b:       test   %eax,%eax
0x626d6d:       sete   %al
```

Once the index table has been allocated and read from the file, the following loop will be executed. This loop is responsible for scanning the index table for a record of type 0x0026. After initializing an index used to select the entry in the index table, at [18] the index will be compared with the number of elements in the index table in order to determine when the loop should exit. At [19], the type at the current index of the index table is loaded into the `%eax` register, and then compared against the value 0x0026. If the type of the entry corresponds to the value of 0x0026, then the record will be parsed at [20]. It is suspected by the author that this record type is used to extend the index record table.

```
0x626dfc:       movl   $0x0,-0x15c(%rbp)
...
0x626e06:       movzwl -0x128(%rbp),%eax    ; number of elements in table
0x626e0d:       movzwl %ax,%eax
0x626e10:       cmp    -0x15c(%rbp),%eax    ; [18] check against current index into index table
0x626e16:       jle    0x626ec6             ; exit loop
...
0x626e1c:       mov    -0x15c(%rbp),%eax    ; current index into index table
0x626e22:       cltq
0x626e24:       lea    0x0(,%rax,8),%rdx
0x626e2c:       mov    -0x150(%rbp),%rax    ; index table buffer
0x626e33:       add    %rdx,%rax
0x626e36:       movzwl (%rax),%eax          ; [19] read index record type
0x626e39:       cmp    $0x26,%ax            ; [19] compare against 0x0026
0x626e3d:       jne    0x626eba
...
0x626e83:       mov    -0x140(%rbp),%rax    ; current index record
0x626e8a:       movzwl 0x2(%rax),%eax       ; current index record size
0x626e8e:       movzwl %ax,%esi
0x626e91:       mov    -0x170(%rbp),%rcx    ; FILE*
0x626e98:       mov    -0x17c(%rbp),%edx    ; flag
0x626e9e:       mov    -0x168(%rbp),%rax    ; document object
0x626ea5:       mov    %rax,%rdi
0x626ea8:       callq  0x61feac             ; [20] read record 0x0026
0x626ead:       test   %eax,%eax
0x626eaf:       sete   %al
...
0x626eba:       addl   $0x1,-0x15c(%rbp)
0x626ec1:       jmpq   0x626e06
```

After scanning for record type 0x0026, the application will then enter the following loop. This loop will translate the record types in the index table by adding 2 to the record type. After initializing the index for the loop, at [21] the application will check this index against the total number of records to determine when the loop should be executed. For each index of the loop, the pointer to the current record will be calculated at [22]. Once a pointer to the current record has been determined, the loop will check if its type is larger than 0x000f at [23]. This will be used at [24] to determine whether the record type should be increased by +2.

```
0x626ee8:       movl   $0x0,-0x158(%rbp)    ; index of current record
...
0x626ef2:       movzwl -0x128(%rbp),%eax    ; total number of records
0x626ef9:       movzwl %ax,%eax
0x626efc:       cmp    -0x158(%rbp),%eax    ; [21] check current index against total number of records
0x626f02:       jle    0x626f55
...
0x626f04:       mov    -0x158(%rbp),%eax    ; current index
0x626f0a:       cltq
0x626f0c:       lea    0x0(,%rax,8),%rdx
0x626f14:       mov    -0x150(%rbp),%rax    ; pointer to index table
0x626f1b:       add    %rdx,%rax
0x626f1e:       mov    %rax,-0x138(%rbp)    ; [22] calculate pointer to current record in index
...
0x626f25:       mov    -0x138(%rbp),%rax    ; current record in index
0x626f2c:       movzwl (%rax),%eax          ; read uint16_t record type
0x626f2f:       cmp    $0xf,%ax             ; [23] check type against 0x000f
0x626f33:       jbe    0x626f4c
...
0x626f35:       mov    -0x138(%rbp),%rax    ; current record in index
0x626f3c:       movzwl (%rax),%eax          ; read uint16_t record type
0x626f3f:       lea    0x2(%rax),%edx       ; [24] add 2 to it
0x626f42:       mov    -0x138(%rbp),%rax    ; current record in index
0x626f49:       mov    %dx,(%rax)           ; [24] write it back
...
0x626f4c:       addl   $0x1,-0x158(%rbp)
0x626f53:       jmp    0x626ef2
```

Finally, the application will enter the following loop. This loop is responsible for scanning the index table for a list of record types in an array as a global. This is performed by two nested loops. The outermost loop iterates through each element in the aforementioned global array. This loop terminates at [25] by checking to see if the current loop's index is larger than 0x3a. The innermost loop is responsible for iterating through each record in the index table. Similar to the prior described loops, at [26] the outermost loop's index is checked against the total number of elements. At [27] a pointer is calculated to point to the current record in the index table. At [28], the type is read from the current record and then checked against the current element in the global array selected by the index of the outermost loop.

```
0x626f55:       movl   $0x0,-0x15c(%rbp)            ; initialize index for loop
...
0x626f5f:       mov    -0x15c(%rbp),%eax            ; index for loop
0x626f65:       cltq
0x626f67:       mov    $0x3a,%edx
0x626f6c:       cmp    %rdx,%rax                    ; [25] check current index against 0x3a
0x626f6f:       jae    0x627097
...
0x626f75:       movl   $0x0,-0x158(%rbp)            ; initialize index for current record of table
0x626f7f:       movzwl -0x128(%rbp),%eax            ; total number of records in table
0x626f86:       movzwl %ax,%eax
0x626f89:       cmp    -0x158(%rbp),%eax            ; [26] check index for current record against total
0x626f8f:       jle    0x62708b
...
0x626f95:       mov    -0x158(%rbp),%eax            ; index of current record in table
0x626f9b:       cltq
0x626f9d:       lea    0x0(,%rax,8),%rdx
0x626fa5:       mov    -0x150(%rbp),%rax            ; pointer to index table
0x626fac:       add    %rdx,%rax
0x626faf:       mov    %rax,-0x138(%rbp)            ; [27] calculate pointer to current record
...
0x626fb6:       mov    -0x138(%rbp),%rax            ; current record in table
0x626fbd:       movzwl (%rax),%edx                  ; [28] read type from index table record
0x626fc0:       mov    -0x15c(%rbp),%eax            ; index for outer loop
0x626fc6:       cltq
0x626fc8:       movzwl 0x1ca43c0(%rax,%rax,1),%eax  ; [28] index into global array
0x626fd0:       cmp    %ax,%dx
0x626fd3:       jne    0x62707f
...
0x62707f:       addl   $0x1,-0x158(%rbp)            ; next iteration for current record
0x627086:       jmpq   0x626f7f
...
0x62708b:       addl   $0x1,-0x15c(%rbp)            ; [25] next iteration for index into global
0x627092:       jmpq   0x626f5f
```

The table of record types that the index table is scanned can be found at the following address.

```
1ca43c0 | 000d 000e 003f 0040 000f 0010 001a 001c | ....?.@.........
1ca43d0 | 0013 0029 0017 001e 0027 0020 0021 0009 | ..).....'. .!...
1ca43e0 | 0042 0024 0030 0043 0031 001f 0000 0022 | B.$.0.C.1.....".
1ca43f0 | 0001 0038 0003 002e 003a 0007 002c 0008 | ..8.....:...,...
1ca4400 | 0019 0028 001b 0006 0002 003b 0005 0014 | ..(.......;.....
1ca4410 | 0016 002b 000c 0039 000a 003d 000b 002a | ..+...9...=...*.
1ca4420 | 0036 0004 002d 002f 0032 0033 0034 0037 | 6...-./.2.3.4.7.
1ca4430 | 003c 003e                               | <.>.
```

Once a record in the index table with a type corresponding to the current element in the global has been found, the following block of code is executed. The function call at [26] in the following code is directly responsible for parsing an individual record within the index table based on the record type extracted from the current record.

```
0x627035:       mov    -0x17c(%rbp),%edi    ; parse record flag
0x62703b:       mov    -0x178(%rbp),%rcx    ; document path
0x627042:       mov    -0x170(%rbp),%rdx    ; FILE*
0x627049:       mov    -0x138(%rbp),%rsi    ; current record in index table
0x627050:       mov    -0x168(%rbp),%rax    ; document object
0x627057:       mov    %edi,%r8d
0x62705a:       mov    %rax,%rdi
0x62705d:       callq  0x624d1e             ; [26] parse record
0x627062:       test   %eax,%eax
0x627064:       sete   %al
```

After the prior-mentioned loops have scanned and discovered a record that corresponds to the type in the global array, the following function is executed. This function is responsible for reading the data associated with the record type and passing the data as a parameter to the function responsible for parsing it. At [27], the offset for the current record is read from the index table and then used to set the offset for the current file stream containing the document. Then at [28], the 16-bit record type is read from the current index table record and used to determine the case responsible for parsing the record type.

```
0x624d1e:       push   %rbp
0x624d1f:       mov    %rsp,%rbp
0x624d22:       sub    $0x160,%rsp
0x624d29:       mov    %rdi,-0x138(%rbp)    ; document object
0x624d30:       mov    %rsi,-0x140(%rbp)    ; current record in index table
0x624d37:       mov    %rdx,-0x148(%rbp)    ; FILE*
0x624d3e:       mov    %rcx,-0x150(%rbp)    ; document path
0x624d45:       mov    %r8d,-0x154(%rbp)    ; parse record flag
...
0x624d69:       mov    -0x118(%rbp),%rax    ; current record in index table
0x624d70:       mov    0x4(%rax),%eax       ; [27] uint32_t offset of record
0x624d73:       mov    %eax,%ecx
0x624d75:       mov    -0x148(%rbp),%rax    ; FILE*
0x624d7c:       mov    $0x0,%edx            ; SEEK_SET
0x624d81:       mov    %rcx,%rsi
0x624d84:       mov    %rax,%rdi
0x624d87:       callq  0x410fe0 <fseek@plt> ; [27] seek to offset
...
0x624d8c:       mov    -0x118(%rbp),%rax    ; current record in index table
0x624d93:       movzwl (%rax),%eax          ; [28] uint16_t record type
0x624d96:       movzwl %ax,%eax
0x624d99:       cmp    $0x43,%eax
0x624d9c:       ja     0x625f7f
0x624da2:       mov    %eax,%eax
0x624da4:       mov    0x16ba520(,%rax,8),%rax
0x624dac:       jmpq   *%rax                ; [28] branch to case responsible for record type
```

The case for record 0x003f is handled by the following code. This code simply takes the data that was read using the index table, and then reads a 16-bit size from it. This value along with the document object and file stream is passed to the function call at [29].

```
0x625e63:       mov    -0x118(%rbp),%rax    ; index element
0x625e6a:       movzwl 0x2(%rax),%eax       ; uint16_t size
0x625e6e:       movzwl %ax,%ecx
0x625e71:       mov    -0x148(%rbp),%rdx    ; FILE*
0x625e78:       mov    -0x138(%rbp),%rax    ; document object
0x625e7f:       mov    %ecx,%esi
0x625e81:       mov    %rax,%rdi
0x625e84:       callq  0x62445e             ; [29] handle record 0x003f
0x625e89:       test   %eax,%eax
0x625e8b:       sete   %al
```

The function that is called is directly responsible for parsing records with the type of 0x003f. After storing its parameters onto the stack, the function will read a 32-bit integer from the current position in the file stream at [30]. This length is used to determine the number of bytes used by the first part of the record, and is thus used at [31] to read the same number of bytes from the file into a buffer located on the stack. The values on the stack will then be used to write its values directly into the document object.

```
0x62445e:       push   %rbp
0x62445f:       mov    %rsp,%rbp
0x624462:       sub    $0xa0,%rsp
0x624469:       mov    %rdi,-0x88(%rbp)     ; document object
0x624470:       mov    %esi,-0x8c(%rbp)     ; uint16_t size
0x624476:       mov    %rdx,-0x98(%rbp)     ; FILE*
...
0x6244c1:       mov    $0x4,%edx            ; length
0x6244c6:       lea    -0x74(%rbp),%rcx     ; destination
0x6244ca:       mov    -0x98(%rbp),%rax     ; FILE*
0x6244d1:       mov    %rcx,%rsi
0x6244d4:       mov    %rax,%rdi
0x6244d7:       callq  0x62738a             ; [30] fread
...
0x6244ef:       mov    -0x74(%rbp),%edx     ; uint32_t size
0x6244f2:       lea    -0x70(%rbp),%rcx     ; buffer on stack
0x6244f6:       mov    -0x98(%rbp),%rax     ; FILE*
0x6244fd:       mov    %rcx,%rsi
0x624500:       mov    %rax,%rdi
0x624503:       callq  0x62738a             ; [31] fread
0x624508:       test   %eax,%eax
0x62450a:       sete   %al
```

After reading the first part of the record, the function will continue by again reading another 32-bit integer from the file stream at [32]. This integer will also be used as a length, but it is believed by the author to be a length for a wide-character string. This is because the length is scaled before allocating by adding 1 to the integer, and then multiplying it by 2 before passing it to the call at [33] to allocate space on the heap. Due to the application treating this size as a signed value, this multiplication can result in a signed integer overflow which can result in an undersized heap buffer. It would be prudent to note that one characteristic of the application's heap allocator is that it fails on a zero-sized allocation, thus the way one would typically trigger this vulnerability would result in an error that is properly handled by the application. Nonetheless after the space has been allocated, then at [34] the result of the allocation will be saved within the document object.

```
0x6246ac:       mov    $0x4,%edx            ; length
0x6246b1:       lea    -0x74(%rbp),%rcx     ; destination
0x6246b5:       mov    -0x98(%rbp),%rax     ; FILE*
0x6246bc:       mov    %rcx,%rsi
0x6246bf:       mov    %rax,%rdi
0x6246c2:       callq  0x62738a             ; [32] fread
...
0x6246e5:       mov    -0x74(%rbp),%eax     ; sint32_t size
0x6246e8:       add    $0x1,%eax            ; add 1
0x6246eb:       mov    $0x2,%edx
0x6246f0:       imul   %edx,%eax            ; multiply by 2
0x6246f3:       mov    $0x80,%esi
0x6246f8:       mov    %eax,%edi
0x6246fa:       callq  0xc483ec             ; [33] allocate from heap
0x6246ff:       mov    %rax,%rdx
..
0x624702:       mov    -0x88(%rbp),%rax     ; document object
0x624709:       mov    %rdx,0x2460(%rax)    ; [34] store into document object
```

After allocation of the heap buffer for the wide-character string, the application will recalculate the size and use it to read data from the file directly into the heap buffer at [35]. This is done using a function that in essence wraps the `fread(3)` function so that one would only need to specify the number of bytes to read. If reading the requested data from the file stream results in an error, the conditional branch at [36] will then be taken to handle it.

```
0x624728:       mov    $0x2,%edx
0x62472d:       mov    -0x74(%rbp),%eax     ; uint32_t from file
0x624730:       imul   %eax,%edx
0x624733:       mov    -0x88(%rbp),%rax     ; document object
0x62473a:       mov    0x2460(%rax),%rcx    ; pointer to heap allocation
0x624741:       mov    -0x98(%rbp),%rax     ; FILE*
0x624748:       mov    %rcx,%rsi
0x62474b:       mov    %rax,%rdi
0x62474e:       callq  0x62738a             ; [35] fread
0x624753:       test   %eax,%eax
0x624755:       sete   %al
0x624758:       test   %al,%al
0x62475a:       je     0x624763             ; [36] error case
```

While handling the error condition from `fread(3)`, the following block of code will be executed. This is responsible for ensuring that the buffer that was allocated on the heap is always null-terminated. This is done at [37] by loading the pointer to the heap allocation from the document object, fetching the 32-bit length that was read from the file, multiplying the length by 2, and then combining them to result in a pointer that points at the perceived end of the wide-character string.

However when calculating the end of the string, the application performs an unsigned multiplication which has a different product than the signed multiplication that was used to allocate the space for the string. Due to different lengths being used for both the allocation of the string's heap buffer, and to calculate the pointer to the end of the string, this can result in the pointer being set to an address that is outside the bounds of the heap allocation. At [38] the application will write its null-terminator to the calculated pointer which can corrupt memory on the heap. This type of memory corruption can lead to code execution under the context of the application.

```
0x624763:       mov    -0x88(%rbp),%rax     ; document object
0x62476a:       mov    0x2460(%rax),%rax    ; pointer to heap allocation
0x624771:       mov    -0x74(%rbp),%edx     ; [37] uint32_t from file
0x624774:       mov    %edx,%edx
0x624776:       add    %rdx,%rdx            ; [37] multiply by 2
0x624779:       add    %rdx,%rax
0x62477c:       movw   $0x0,(%rax)          ; [38] write 16-bit NULL to heap allocation
0x624781:       jmp    0x62479f
```

Crash Information

If we set a breakpoint at where the length is read as a 32-bit integer, we can see it get stored on the stack at `-0x74($rbp)`.

```
(gdb) bp 0x6246c2
Breakpoint 4 at 0x6246c2

(gdb) r
Starting program: /usr/share/office2021/textmaker poc.tmd
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7fffee702700 (LWP 4278)]
...

Thread 1 "textmaker" hit Breakpoint 4, 0x00000000006246c2 in ?? ()
(gdb) x/6i 0x6246ac
   0x6246ac:    mov    $0x4,%edx
   0x6246b1:    lea    -0x74(%rbp),%rcx
   0x6246b5:    mov    -0x98(%rbp),%rax
   0x6246bc:    mov    %rcx,%rsi
   0x6246bf:    mov    %rax,%rdi
=> 0x6246c2:    callq  0x62738a
(gdb) p/x *(size_t*)($rbp-0x74)
$1 = 0x0
(gdb) ni
0x00000000006246c7 in ?? ()
(gdb) p/x *(size_t*)($rbp-0x74)
$4 = 0x80000000
(gdb) i r $rax
rax            0x1                 0x1
```

Continuing onto the allocation, we can see the signed multiply results in a total size of 2 bytes being allocated for the heap buffer that the file data will be read into.

```
(gdb) bp 0x6246fa
Breakpoint 5 at 0x6246fa
(gdb) c
Continuing.

Thread 1 "textmaker" hit Breakpoint 5, 0x00000000006246fa in ?? ()

(gdb) x/8i 0x6246e5
   0x6246e5:    mov    -0x74(%rbp),%eax
   0x6246e8:    add    $0x1,%eax
   0x6246eb:    mov    $0x2,%edx
   0x6246f0:    imul   %edx,%eax
   0x6246f3:    mov    $0x80,%esi
   0x6246f8:    mov    %eax,%edi
=> 0x6246fa:    callq  0xc483ec
   0x6246ff:    mov    %rax,%rdx

(gdb) i r $edi
edi            0x2                 0x2
(gdb) p/x *(size_t)($rbp-0x74)
$5 = 0x80000000
(gdb) p/x (*(size_t)($rbp-0x74) + 1) * 2
$6 = 0x2
(gdb) ni
0x00000000006246ff in ?? ()
(gdb) i r $rax
rax            0x30d3df0           0x30d3df0
```

Continuing execution to the first read, we can see that the function call is being passed an integer of 0 to cause the function to return a failure code. Stepping over the call results in the value 0x1 being returned in the `%rax` register. This results in the execution of the error-handling branch.

```
(gdb) bp 0x62474e
Breakpoint 6 at 0x62474e
(gdb) c
Continuing.

Thread 1 "textmaker" hit Breakpoint 6, 0x000000000062474e in ?? ()

(gdb) x/8i 0x624733
   0x624733:    mov    -0x88(%rbp),%rax
   0x62473a:    mov    0x2460(%rax),%rcx
   0x624741:    mov    -0x98(%rbp),%rax
   0x624748:    mov    %rcx,%rsi
   0x62474b:    mov    %rax,%rdi
=> 0x62474e:    callq  0x62738a
   0x624753:    test   %eax,%eax
   0x624755:    sete   %al

(gdb) i r $rdi $edx $rsi
rdi            0x28352f0           0x28352f0
edx            0x0                 0x0
rsi            0x30d3df0           0x30d3df0
(gdb) ni
0x0000000000624753 in ?? ()
(gdb) i r $rax
rax            0x1                 0x1
```

Setting a breakpoint partway through the error handler let's us view the current pointer and size that is used to calculate the end of the buffer before it is multiplied by 2. Stepping over a few instructions and we can see the size was multiplied by 2 and added to the pointer that is targeting the result of the heap allocation. The result is entirely outside the bounds of the 2-byte buffer that was allocated.

```
(gdb) bp 0x624774
Breakpoint 7 at 0x624774
(gdb) c
Continuing.

Thread 1 "textmaker" hit Breakpoint 7, 0x0000000000624774 in ?? ()
(gdb) x/8i 0x624763
   0x624763:    mov    -0x88(%rbp),%rax
   0x62476a:    mov    0x2460(%rax),%rax
   0x624771:    mov    -0x74(%rbp),%edx
=> 0x624774:    mov    %edx,%edx
   0x624776:    add    %rdx,%rdx
   0x624779:    add    %rdx,%rax
   0x62477c:    movw   $0x0,(%rax)
   0x624781:    jmp    0x62479f
(gdb) i r $rax $edx
rax            0x30d3df0           0x30d3df0
edx            0x80000000          0x80000000

(gdb) si
0x0000000000624776 in ?? ()
(gdb) si
0x0000000000624779 in ?? ()
(gdb) si
0x000000000062477c in ?? ()

(gdb) x/i $pc
=> 0x62477c:    movw   $0x0,(%rax)

(gdb) i r $rdx $rax
rdx            0x100000000         0x100000000
rax            0x1030d3df0         0x1030d3df0
```

Resuming execution shows the 16-bit null word being written to an invalid address.

```
(gdb) c
Continuing.

Thread 1 "textmaker" received signal SIGSEGV, Segmentation fault.
0x000000000062477c in ?? ()
(gdb) x/i $pc
=> 0x62477c:    movw   $0x0,(%rax)
(gdb) i r rax
rax            0x1030d3df0         0x1030d3df0
```

Mitigation

Timeline

2020-10-08 - Vendor Disclosure

2020-12-03 - Follow up with vendor

2021-01-05 - 2nd follow up; vendor acknowledged issues fixed

2021-01-05 - Public Release

CREDIT

Discovered by a member of Cisco Talos.

TALOS-2020-1161 TALOS-2020-1163