Talos Vulnerability Report

TALOS-2020-1161

SoftMaker Office TextMaker Document Record 0x001f sign-extension vulnerability

JANUARY 5, 2021

CVE NUMBER

CVE-2020-13544

Summary

An exploitable sign extension 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 sign-extend a length used to terminate a loop, which can later result in the loop's index being used to write outside the bounds of a heap buffer during the reading of file data. 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-194 - Unexpected Sign Extension

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)     ; FILE*
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     ; FILE*
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 0x001f is handled by the following code. This code simply takes the index table record that was read from the table, extracts the 16-bit size, and then passes it with the file stream to the function call at [29]. It is prudent to note that the 16-bit size read from the record is treated as an unsigned value. This cast is relevant to the vulnerability described by this document and as such will be shown how to be used to corrupt memory allocated on the heap.

```
0x625ace:       mov    -0x118(%rbp),%rax    ; index table record
0x625ad5:       movzwl 0x2(%rax),%eax       ; uint16_t size
0x625ad9:       movzwl %ax,%ecx
0x625adc:       mov    -0x148(%rbp),%rdx    ; FILE*
0x625ae3:       mov    -0x138(%rbp),%rax    ; document object
0x625aea:       mov    %ecx,%esi
0x625aec:       mov    %rax,%rdi
0x625aef:       callq  0x61f800             ; [29]
0x625af4:       test   %eax,%eax
0x625af6:       sete   %al
```

Once inside the function call that handles record 0x001f, the first thing that is done is to take the unsigned record size and subtract 1 from it. The size is then checked at [30] as it is used to describe the number of elements contained by the record. This record size is then multiplied by the value 0x38 at [31] in order to allocate space for the data that is contained by the record.

```
0x61f800:       push   %rbp
0x61f801:       mov    %rsp,%rbp
0x61f804:       push   %rbx
0x61f805:       sub    $0xc8,%rsp
0x61f80c:       mov    %rdi,-0xb8(%rbp)     ; document object
0x61f813:       mov    %esi,-0xbc(%rbp)     ; uint16_t record size
0x61f819:       mov    %rdx,-0xc8(%rbp)     ; FILE*
...
0x61f82f:       subl   $0x1,-0xbc(%rbp)
0x61f836:       cmpl   $0x0,-0xbc(%rbp)     ; [30]
0x61f83d:       jle    0x61fb87
...
0x61f843:       mov    -0xbc(%rbp),%eax     ; uint16_t record size
0x61f849:       cltq
0x61f84b:       mov    %eax,%edx
0x61f84d:       mov    $0x38,%eax
0x61f852:       imul   %edx,%eax            ; [31] multiply size by 0x38
0x61f855:       mov    $0x1,%esi
0x61f85a:       mov    %eax,%edi
0x61f85c:       callq  0xc483ec             ; [31] make an allocation
0x61f861:       mov    %rax,-0xa8(%rbp)     ; store it
0x61f868:       cmpq   $0x0,-0xa8(%rbp)
0x61f870:       sete   %al
```

After allocating space for the record, the application treats this space as an array. This array is initialized by the loop that is shown in the following code using data from the file. After initializing the index for this loop, the loop will then compare the current index against the size that was read from the record in the index table. At [31], however, the application loads the index for the current loop iteration as a 16-bit signed value. This results in the following comparison which is used to terminate the loop when the index reaches the total number of elements as specified in the current index table record actually performing a comparison between a potentially signed 16-bit index and an unsigned 16-bit length.

When the loop iterates enough times to set the sign flag of the 16-bit index of the loop, the comparison will always result in the current index being less than the unsigned size that was read from the record. This allows the loop to iterate a number of times that is larger than the size specified in the record from the index table. This can result in any calculations which use this index to point outside the boundaries of the heap allocation. During each iteration of the loop, the application will first check the document version at [32] in order to determine which method to use to read data from the file into the buffer that was allocated.

```
0x61f87b:   movw   $0x0,-0xaa(%rbp)    ; index
...
0x61f884:       movswl -0xaa(%rbp),%eax     ; [31] sint16_t
0x61f88b:       cmp    -0xbc(%rbp),%eax
0x61f891:       jge    0x61fb92
...
0x61f897:       mov    -0xb8(%rbp),%rax     ; document object
0x61f89e:       mov    0x38(%rax),%eax      ; document version
0x61f8a1:       cmp    $0x13a,%eax          ; [32] check version
0x61f8a6:       jg     0x61f9e4
...
0x61fb71:   movzwl -0xaa(%rbp),%eax    ; index
0x61fb78:   add    $0x1,%eax
0x61fb7b:   mov    %ax,-0xaa(%rbp)
0x61fb82:   jmpq   0x61f884
```

If the document version is earlier than 314, then the following code will be executed. This application will read 0x21 bytes from the file stream into a buffer on the stack for each iteration of the loop. After reading the data from the file into the stack, any of the cases at [34] can be used to write outside of the bounds of the heap allocation once the loop iterates enough times for the loop index to eventually set its signed bit. These conditions allow for memory corruption which can allow a path for an attacker to earn code execution within the context of the application.

```
0x61f8ac:       mov    $0x21,%edx           ; length
0x61f8b1:       lea    -0x80(%rbp),%rcx     ; buffer on stack
0x61f8b5:       mov    -0xc8(%rbp),%rax     ; FILE*
0x61f8bc:       mov    %rcx,%rsi
0x61f8bf:       mov    %rax,%rdi
0x61f8c2:       callq  0x62738a             ; [33] fread
0x61f8c7:       test   %eax,%eax
0x61f8c9:       sete   %al
...
0x61f8d4:       movswq -0xaa(%rbp),%rax     ; index
0x61f8dc:       shl    $0x3,%rax
0x61f8e0:       lea    0x0(,%rax,8),%rdx
0x61f8e8:       sub    %rax,%rdx
0x61f8eb:       mov    -0xa8(%rbp),%rax
0x61f8f2:       add    %rax,%rdx
0x61f8f5:       movzbl -0x6c(%rbp),%eax
0x61f8f9:       mov    %al,0x28(%rdx)       ; [34] store uint8_t
...
0x61f8fc:       movswq -0xaa(%rbp),%rax     ; index
0x61f904:       shl    $0x3,%rax
0x61f908:       lea    0x0(,%rax,8),%rdx
0x61f910:       sub    %rax,%rdx
0x61f913:       mov    -0xa8(%rbp),%rax
0x61f91a:       add    %rax,%rdx
0x61f91d:       movzbl -0x6b(%rbp),%eax
0x61f921:       mov    %al,0x2a(%rdx)       ; [34] store uint8_t
...
0x61f924:       movswq -0xaa(%rbp),%rax     ; index
0x61f92c:       shl    $0x3,%rax
0x61f930:       lea    0x0(,%rax,8),%rdx
0x61f938:       sub    %rax,%rdx
0x61f93b:       mov    -0xa8(%rbp),%rax
0x61f942:       add    %rax,%rdx
0x61f945:       movzbl -0x6a(%rbp),%eax
0x61f949:       mov    %al,0x2b(%rdx)       ; [34] store uint8_t
...
0x61f94c:       movswq -0xaa(%rbp),%rax     ; index
0x61f954:       shl    $0x3,%rax
0x61f958:       lea    0x0(,%rax,8),%rdx
0x61f960:       sub    %rax,%rdx
0x61f963:       mov    -0xa8(%rbp),%rax
0x61f96a:       add    %rax,%rdx
0x61f96d:       movzbl -0x69(%rbp),%eax
0x61f971:       mov    %al,0x2c(%rdx)       ; [34] store uint8_t
...
0x61f974:       movswq -0xaa(%rbp),%rax     ; index
0x61f97c:       shl    $0x3,%rax
0x61f980:       lea    0x0(,%rax,8),%rdx
0x61f988:       sub    %rax,%rdx
0x61f98b:       mov    -0xa8(%rbp),%rax
0x61f992:       add    %rax,%rdx
0x61f995:       movzbl -0x68(%rbp),%eax
0x61f999:       mov    %al,0x2d(%rdx)       ; [34] store uint8_t
...
0x61f99c:       movswq -0xaa(%rbp),%rax     ; index
0x61f9a4:       shl    $0x3,%rax
0x61f9a8:       lea    0x0(,%rax,8),%rdx
0x61f9b0:       sub    %rax,%rdx
0x61f9b3:       mov    -0xa8(%rbp),%rax
0x61f9ba:       add    %rdx,%rax
0x61f9bd:       mov    %rax,%rdx
```

If the document version is later than 314, then at [35] the application will choose to read 0x31 bytes for each iteration of the loop onto the stack. After reading the record from the file, the application will begin to extract various fields from the data that was read and then write this data into the array that was allocated on the heap. At [36], the application will calculate the length, and then use the current index as a signed value when calculating a pointer into the array. Due to the index being treated as a signed value, this pointer can point to data outside the bounds of the heap buffer. After calculating the length and the pointer, then at [37] the `memcpy` function will be used to copy the relevant data from the buffer on the stack to the pointer that was calculated.

```
0x61f9e4:       mov    $0x31,%edx           ; length
0x61f9e9:       lea    -0x50(%rbp),%rcx     ; buffer on stack
0x61f9ed:       mov    -0xc8(%rbp),%rax     ; FILE*
0x61f9f4:       mov    %rcx,%rsi
0x61f9f7:       mov    %rax,%rdi
0x61f9fa:       callq  0x62738a             ; [35] fread
0x61f9ff:       test   %eax,%eax
0x61fa01:       sete   %al
...
0x61fa0c:       mov    $0x2,%edx
0x61fa11:       mov    %edx,%eax
0x61fa13:       shl    $0x2,%eax
0x61fa16:       add    %edx,%eax
0x61fa18:       shl    $0x2,%eax
0x61fa1b:       mov    %eax,%esi            ; [36] length (0x18)
...
0x61fa1d:       movswq -0xaa(%rbp),%rax     ; sint16_t current index
0x61fa25:       shl    $0x3,%rax
0x61fa29:       lea    0x0(,%rax,8),%rdx
0x61fa31:       sub    %rax,%rdx
0x61fa34:       mov    -0xa8(%rbp),%rax
0x61fa3b:       add    %rdx,%rax
0x61fa3e:       mov    %rax,%rcx            ; [36] destination
...
0x61fa41:       lea    -0x50(%rbp),%rax     ; source
0x61fa45:       mov    %esi,%edx
0x61fa47:       mov    %rax,%rsi
0x61fa4a:       mov    %rcx,%rdi
0x61fa4d:       callq  0xc4b214             ; [37] memcpy
```

Similar to the code that is responsible for handling document versions earlier than 314, the following code can also write bytes outside the bounds of the allocated buffer. These writes at [38] can also result in a controlled heap corruption which can allow for code execution under the context of the application.

```
0x61fa52:       movswq -0xaa(%rbp),%rax
0x61fa5a:       shl    $0x3,%rax
0x61fa5e:       lea    0x0(,%rax,8),%rdx
0x61fa66:       sub    %rax,%rdx
0x61fa69:       mov    -0xa8(%rbp),%rax
0x61fa70:       add    %rax,%rdx
0x61fa73:       movzbl -0x28(%rbp),%eax
0x61fa77:       mov    %al,0x28(%rdx)       ; [38] store uint8_t
...
0x61fa7a:       movswq -0xaa(%rbp),%rax
0x61fa82:       shl    $0x3,%rax
0x61fa86:       lea    0x0(,%rax,8),%rdx
0x61fa8e:       sub    %rax,%rdx
0x61fa91:       mov    -0xa8(%rbp),%rax
0x61fa98:       add    %rax,%rdx
0x61fa9b:       movzbl -0x27(%rbp),%eax
0x61fa9f:       mov    %al,0x2a(%rdx)       ; [38] store uint8_t
...
0x61faa2:       movswq -0xaa(%rbp),%rax
0x61faaa:       shl    $0x3,%rax
0x61faae:       lea    0x0(,%rax,8),%rdx
0x61fab6:       sub    %rax,%rdx
0x61fab9:       mov    -0xa8(%rbp),%rax
0x61fac0:       add    %rax,%rdx
0x61fac3:       movzbl -0x26(%rbp),%eax
0x61fac7:       mov    %al,0x2b(%rdx)       ; [38] store uint8_t
...
0x61faca:       movswq -0xaa(%rbp),%rax
0x61fad2:       shl    $0x3,%rax
0x61fad6:       lea    0x0(,%rax,8),%rdx
0x61fade:       sub    %rax,%rdx
0x61fae1:       mov    -0xa8(%rbp),%rax
0x61fae8:       add    %rax,%rdx
0x61faeb:       movzbl -0x25(%rbp),%eax
0x61faef:       mov    %al,0x2c(%rdx)       ; [38] store uint8_t
...
0x61faf2:       movswq -0xaa(%rbp),%rax
0x61fafa:       shl    $0x3,%rax
0x61fafe:       lea    0x0(,%rax,8),%rdx
0x61fb06:       sub    %rax,%rdx
0x61fb09:       mov    -0xa8(%rbp),%rax
0x61fb10:       add    %rax,%rdx
0x61fb13:       movzbl -0x24(%rbp),%eax
0x61fb17:       mov    %al,0x2d(%rdx)       ; [38] store uint8_t
```

Crash Information

Starting up the application and then setting a breakpoint on the allocation shows our length from the file being used in a calculation and multiplied by 0x38. This returns a 0x37ff90 sized buffer from the heap.

```
(gdb) bp 0x61f85c
Breakpoint 4 at 0x61f85c
(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 4724)]
...
Thread 1 "textmaker" hit Breakpoint 4, 0x000000000061f85c in ?? ()
(gdb) x/9i $61f843
   0x61f843:    mov    -0xbc(%rbp),%eax
   0x61f849:    cltq
   0x61f84b:    mov    %eax,%edx
   0x61f84d:    mov    $0x38,%eax
   0x61f852:    imul   %edx,%eax
   0x61f855:    mov    $0x1,%esi
   0x61f85a:    mov    %eax,%edi
=> 0x61f85c:    callq  0xc483ec
   0x61f861:    mov    %rax,-0xa8(%rbp)
(gdb) p/x *(uint32_t*)($rbp-0xbc)
$3 = 0xfffe
(gdb) i r $edi
edi            0x37ff90            0x37ff90
(gdb) ni
0x000000000061f861 in ?? ()
(gdb) i r $rax
rax            0x44c1d60           0x44c1d60
```

If we set a conditional breakpoint at the beginning of the loop, we can escape to the debugger when the index reaches a negative value. The output shows that this conditional should remain true until 32768 more iterations of the loop occur.

```
(gdb) bp 0x61f884
Breakpoint 5 at 0x61f884
(gdb) cond 5 *(int16_t*)($rbp-0xaa) < 0
(gdb) c
Continuing.

Thread 1 "textmaker" hit Breakpoint 5, 0x000000000061f884 in ?? ()
(gdb) x/3i $pc
=> 0x61f884:    movswl -0xaa(%rbp),%eax
   0x61f88b:    cmp    -0xbc(%rbp),%eax
   0x61f891:    jge    0x61fb92
(gdb) p/x *(int16_t*)($rbp-0xaa)
$24 = 0x8000
(gdb) p/d *(int16_t*)($rbp-0xaa)
$23 = -32768
(gdb) p/x *(uint32_t*)($rbp-0xbc)
$26 = 0xfffe
(gdb) p/d *(uint32_t*)($rbp-0xbc)
$27 = 65534
```

Setting a breakpoint at the first write once the signed bit has been set in the index shows that it first gets sign-extended before it's used to calculate a pointer into the buffer that was allocated before entering the loop. Stepping over a few instructions shows that the offset which will get added to the pointer will be less than 0. The pointer that this negative offset will be added to is also displayed.

```
(gdb) bp 0x61f8d4
Breakpoint 6 at 0x61f8d4
(gdb) c
Continuing.

Thread 1 "textmaker" hit Breakpoint 6, 0x000000000061f8d4 in ?? ()
(gdb) x/8i $pc
=> 0x61f8d4:    movswq -0xaa(%rbp),%rax
   0x61f8dc:    shl    $0x3,%rax
   0x61f8e0:    lea    0x0(,%rax,8),%rdx
   0x61f8e8:    sub    %rax,%rdx
   0x61f8eb:    mov    -0xa8(%rbp),%rax
   0x61f8f2:    add    %rax,%rdx
   0x61f8f5:    movzbl -0x6c(%rbp),%eax
   0x61f8f9:    mov    %al,0x28(%rdx)
(gdb) si
0x000000000061f8dc in ?? ()
(gdb) i r $rax
rax            0xffffffffffff8000  0xffffffffffff8000
(gdb) si
0x000000000061f8e0 in ?? ()
(gdb) i r $rax
rax            0xfffffffffffc0000  0xfffffffffffc0000
(gdb) si
0x000000000061f8e8 in ?? ()
(gdb) i r $rax $rdx
rax            0xfffffffffffc0000  0xfffffffffffc0000
rdx            0xffffffffffe00000  0xffffffffffe00000
(gdb) si
0x000000000061f8eb in ?? ()
(gdb) i r $rax $rdx
rax            0xfffffffffffc0000  0xfffffffffffc0000
rdx            0xffffffffffe40000  0xffffffffffe40000
(gdb) x/i $pc
=> 0x61f8eb:    mov    -0xa8(%rbp),%rax
(gdb) p/x *(intptr_t)($rbp-0xa8)
$32 = 0x44c1d60
```

Stepping over the next instruction shows the negative offset being added to the pointer. When this pointer is written to, the memory in front of the heap allocation will be overwritten with values read from the file.

```
(gdb) si
0x000000000061f8f2 in ?? ()
(gdb) x/i $pc
=> 0x61f8f2:    add    %rax,%rdx
(gdb) i r $rax $rdx
rax            0x44c1d60           0x44c1d60
rdx            0xffffffffffe40000  0xffffffffffe40000
(gdb) si
0x000000000061f8f5 in ?? ()
(gdb) i r $rax $rdx
rax            0x44c1d60           0x44c1d60
rdx            0x4301d60           0x4301d60
```

The next two instructions will read a byte from the stack buffer that file contents were written into, and then write that byte directly to the memory targeted by the miscalculated pointer. Stepping over this instruction corrupts a single byte of memory at the pointer.

```
(gdb) x/2i $pc
=> 0x61f8f5:    movzbl -0x6c(%rbp),%eax
   0x61f8f9:    mov    %al,0x28(%rdx)
(gdb) si
0x000000000061f8f9 in ?? ()
(gdb) i r $al $rdx
al             0x47                0x47
rdx            0x4301d60           0x4301d60
(gdb) si
0x000000000061f8fc in ?? ()

(gdb) bd 6
(gdb) finish
Run till exit from #0  0x000000000061f884 in ?? ()
0x0000000000625af4 in ?? ()
```

Exploit Proof of Concept

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.