

Talos Vulnerability Report

TALOS-2020-1163

SoftMaker Office TextMaker Document Record 0x002a integer overflow vulnerability

JANUARY 5, 2021

CVE NUMBER

CVE-2020-13546

Summary

An exploitable integer overflow 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 buffer overflow. 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-190 - Integer Overflow or Wraparound

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   %r15
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,%eax
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 contain 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 0x002a is handled by the following code. This code is only used to fetch the stream and document object in order to wrap the function call at [29].

```

0x625b78:    mov     -0x148(%rbp),%rdx    ; FILE*
0x625b7f:    mov     -0x138(%rbp),%rax    ; document object
0x625b86:    mov     %rdx,%rsi
0x625b89:    mov     %rax,%rdi
0x625b8c:    callq   0x62183e             ; [29] handle record 0x002a
0x625b91:    test    %eax,%eax
0x625b93:    sete    %al

```

The following function is then entered and is directly responsible for parsing records with the type of 0x002a. When this function is entered, it is assumed that the stream that was passed as a parameter is presently pointing at the data representing the record's contents. At [30], the function will read 32-bits from the file and store it onto the stack. This will later be used as a length when allocating space on the heap. At [31], the function will then validate that the length is non-zero.

```

0x62183e:    push    %rbp
0x62183f:    mov     %rsp,%rbp
0x621842:    push    %r12
0x621844:    push    %rbx
0x621845:    sub     $0x1e0,%rsp
0x62184c:    mov     %rdi,-0x1e8(%rbp)    ; document object
0x621853:    mov     %rsi,-0x1f0(%rbp)    ; FILE*
...
0x6218c4:    lea     -0x1e0(%rbp),%rcx    ; result
0x6218cb:    mov     -0x1f0(%rbp),%rax    ; FILE*
0x6218d2:    mov     $0x4,%edx            ; length
0x6218d7:    mov     %rcx,%rsi
0x6218da:    mov     %rax,%rdi
0x6218dd:    callq   0x62738a             ; [30] fread
0x6218e2:    test    %eax,%eax
0x6218e4:    sete    %al
...
0x6218ef:    mov     -0x1e0(%rbp),%eax
0x6218f5:    test    %eax,%eax            ; [31] check nonzero
0x6218f7:    je      0x621f88

```

Immediately following the validation of the length, the function will take this 32-bit length and multiply it by the value of 0x23c at [32]. Due to the length being a total of 32-bits and its operand of up to 10-bits, this multiplication can overflow resulting in the allocation being of a smaller size than expected. Once the result of the multiplication has been obtained, it is then truncated to 32-bits and then passed to the function call at [33] which is responsible for allocating space from the heap. At [34], the original length that was used prior the multiplication containing the integer overflow is then stored into a variable on the stack.

```

0x6218fd:    mov     $0x23c,%edx          ; multiplication operand
0x621902:    mov     -0x1e0(%rbp),%eax    ; uint32_t
0x621908:    imul    %edx,%eax            ; [32] multiplication
0x62190b:    mov     $0x1,%esi
0x621910:    mov     %eax,%edi
0x621912:    callq   0xc483ec             ; [33] allocation
0x621917:    mov     %rax,-0x174(%rbp)    ; store pointer to allocation
...
0x62191e:    mov     -0x1e0(%rbp),%eax    ; [34] original length
0x621924:    mov     %eax,-0x178(%rbp)

```

The original length that was stored on the stack is then treated by the function as a terminator for the following loop. This loop is used to treat the space allocated on the heap as an array and will terminate at [35] by comparing the loop's index against the original length. In order to write directly into the correct element of the array, the application will take the current index and multiply it by the array element's size at [36]. This calculation will result in a pointer that is targeting the particular element. At this point for each iteration of the loop, file data will be read into this target pointer depending on the document's version which is checked at [37].

```

0x621938:    movl    $0x0,-0x1d8(%rbp)    ; index
...
0x621942:    mov     -0x1d8(%rbp),%edx    ; index
0x621948:    mov     -0x1e0(%rbp),%eax    ; [35] sentinel
0x62194e:    cmp     %eax,%edx
0x621950:    jae     0x621cf9
...
0x621956:    mov     -0x174(%rbp),%rdx    ; heap buffer
0x62195d:    mov     -0x1d8(%rbp),%eax    ; index
0x621963:    cltq
0x621965:    imul    $0x23c,%rax,%rax     ; multiplication by 0x23c
0x62196c:    add     %rdx,%rax
0x62196f:    mov     %rax,-0x1b8(%rbp)    ; [36] calculate pointer into heap buffer
...
0x621976:    mov     -0x1e8(%rbp),%rax    ; document object
0x62197d:    mov     0x38(%rax),%eax      ; document version
0x621980:    cmp     $0x13a,%eax          ; [37] compare document version
0x621985:    jg      0x621b22
...
0x621ce0:    mov     $0x0,%eax
0x621ce5:    test    %al,%al
0x621ce7:    jne     0x62205a
0x621ced:    addl    $0x1,-0x1d8(%rbp)    ; index
0x621cf4:    jmpq    0x621942

```

If the document version is earlier than 314, the following case will be executed. This case is executed for each iteration of the loop, and will read 0x122 bytes from the file into a buffer on the stack at [38]. After the data has been read, it will then be written into the array that was allocated using the previously calculated pointer. This occurs in many places as highlighted at [38]. Due to the previously mentioned integer overflow, these stores can be used to write outside the bounds of the allocated array which can trigger heap corruption leading to a heap-based overflow. This can grant an attacker a path to earn code execution with the privileges of the application.

```

0x62198b:    mov     $0x122,%edx      ; length
0x621990:    lea     -0x140(%rbp),%rcx ; buffer on stack
0x621997:    mov     -0x1f0(%rbp),%rax ; FILE*
0x62199e:    mov     %rcx,%rsi
0x6219a1:    mov     %rax,%rdi
0x6219a4:    callq   0x62738a          ; [38] fread
0x6219a9:    test    %eax,%eax
0x6219ab:    sete    %al
...
0x6219b6:    mov     -0x140(%rbp),%edx
0x6219bc:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x6219c3:    mov     %edx,%rax         ; [39] store to array element
...
0x6219c5:    mov     -0x28(%rbp),%edx
0x6219c8:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x6219cf:    mov     %edx,0x21c(%rax)   ; [39] store to array element
...
0x6219d5:    mov     -0x24(%rbp),%edx
0x6219d8:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x6219df:    mov     %edx,0x220(%rax)   ; [39] store to array element
...
0x6219e5:    movzwl  -0x20(%rbp),%edx
0x6219e9:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x6219f0:    mov     %dx,0x224(%rax)    ; [39] store to array element
...
0x6219f7:    movzwl  -0x13c(%rbp),%edx
0x6219fe:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x621a05:    mov     %dx,0x4(%rax)      ; [39] store to array element
...
0x621a09:    movzwl  -0x13a(%rbp),%eax
0x621a10:    movzwl  %eax,%edx
0x621a13:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x621a1a:    mov     %edx,0x6(%rax)     ; [39] store to array element
...
0x621a1d:    movzwl  -0x138(%rbp),%edx
0x621a24:    mov     -0x1b8(%rbp),%rax
0x621a2b:    mov     %dx,0xa(%rax)     ; [39] store to array element
...
0x621a2f:    movzwl  -0x136(%rbp),%edx
0x621a36:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x621a3d:    mov     %dx,0xc(%rax)     ; [39] store to array element
...
0x621a41:    mov     -0x134(%rbp),%edx
0x621a47:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x621a4e:    mov     %edx,0xe(%rax)    ; [39] store to array element
...
0x621a51:    mov     -0x130(%rbp),%edx
0x621a57:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x621a5e:    mov     %edx,0x12(%rax)   ; [39] store to array element
...
0x621a61:    mov     -0x12c(%rbp),%edx
0x621a67:    mov     -0x1b8(%rbp),%rax ; pointer into heap buffer
0x621a6e:    mov     %edx,0x16(%rax)   ; [39] store to array element

```

If the document's version was later than 314, then the following code would be executed. This condition is different from the earlier version in that at [40], the application will read 0x228 bytes of data directly into the heap buffer. Similar to the implementation on earlier document versions, at [41] the application will write to a number of places that can also trigger heap corruption by writing outside the bounds of the array that was allocated. This is a heap overflow and with more work can enable an attacker to earn code execution.

```

0x621b22:    mov     $0x228,%edx      ; length
0x621b27:    mov     -0x1b8(%rbp),%rcx ; pointer into heap buffer
0x621b2e:    mov     -0x1f0(%rbp),%rax ; FILE*
0x621b35:    mov     %rcx,%rsi
0x621b38:    mov     %rax,%rdi
0x621b3b:    callq   0x62738a          ; [40] fread
0x621b40:    test    %eax,%eax
0x621b42:    sete    %al
...
0x621c2a:    mov     -0x1b8(%rbp),%rax
0x621c31:    mov     %rdx,0x228(%rax)   ; [41] store to array element
...
0x621c87:    mov     -0x1b8(%rbp),%rax
0x621c8e:    mov     %rdx,0x234(%rax)   ; [41] store to array element

```

Crash Information

This output starts out by setting a breakpoint at the allocation. Emitting its parameter shows the length that was read from the file. After the integer overflow that occurs during the multiply, the result is 0x20c which provides an undersized allocation that will later be written to.

```

(gdb) bp 0x621912
Breakpoint 4 at 0x621912

(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".
...
Thread 1 "textmaker" hit Breakpoint 4, 0x0000000000621912 in ?? ()

(gdb) x/7i 0x6218fd
0x6218fd: mov    $0x23c,%edx
0x621902: mov    -0x1e0(%rbp),%eax
0x621908: imul   %edx,%eax
0x62190b: mov    $0x1,%esi
0x621910: mov    %eax,%edi
=> 0x621912: callq  0xc483ec
0x621917: mov    %rax,-0x174(%rbp)
(gdb) p/x *(int*)($rbp-0x1e0)
$2 = 0x7292cd
(gdb) i r $rdi
rdi      0x20c      0x20c
(gdb) p/x *(int*)($rbp-0x1e0) * 0x23c
$4 = 0x20c
(gdb) ni
0x0000000000621917 in ?? ()
(gdb) i r $rax
rax      0x2fe12c0    0x2fe12c0

```

The following output shows the index being used to calculate an index into the undersized allocation that was just recently mentioned.

```

(gdb) bp 0x62196f
Breakpoint 5 at 0x62196f
(gdb) c
Continuing.

Thread 1 "textmaker" hit Breakpoint 5, 0x000000000062196f in ?? ()
(gdb) x/6i 0x621956
0x621956: mov    -0x174(%rbp),%rdx
0x62195d: mov    -0x1d8(%rbp),%eax
0x621963: cltq
0x621965: imul   $0x23c,%rax,%rax
0x62196c: add    %rdx,%rax
=> 0x62196f: mov    %rax,-0x1b8(%rbp)
(gdb) p/x *(int*)($rbp-0x1d8)
$5 = 0x0
(gdb) i r $rax
rax      0x2fe12c0    0x2fe12c0

```

The following output shows the call to `fread(3)` which is reading 0x228 bytes of data into the previously calculated pointer. This pointer results in 0x228 bytes of data being written outside the bounds of an allocation that is only 0x20c bytes in size.

```

(gdb) bp 0x621b3b
Breakpoint 6 at 0x621b3b
(gdb) c
Continuing.

Thread 1 "textmaker" hit Breakpoint 6, 0x0000000000621b3b in ?? ()
(gdb) x/6i 0x621b22
0x621b22: mov    $0x228,%edx
0x621b27: mov    -0x1b8(%rbp),%rcx
0x621b2e: mov    -0x1f0(%rbp),%rax
0x621b35: mov    %rcx,%rsi
0x621b38: mov    %rax,%rdi
=> 0x621b3b: callq  0x62738a
(gdb) i r $rdi $rsi $edx
rdi      0x2fbd0f0    0x2fbd0f0
rsi      0x2fe12c0    0x2fe12c0
edx      0x228      0x228
(gdb) db 0x2fe12c0 l0x228
2fe12c0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
2fe12d0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
...
2fe14b0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
2fe14c0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
2fe14d0 | 30 02 00 00 00 00 00 00 21 00 00 00 00 00 00 00 | 0.....!.....
2fe14e0 | 00 4e 00 6f 00 74 00 6f | .N.o.t.o

(gdb) ni
0x0000000000621b40 in ?? ()
(gdb)

```

After 0x228 bytes have been read into a 0x20c-byte heap chunk, if we continue execution we can see that the heap has been corrupted.


```
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y

(gdb) c
Continuing.

Thread 1 "textmaker" received signal SIGSEGV, Segmentation fault.
tcache_get (tc_idx=<optimized out>) at malloc.c:2937
2937 malloc.c: No such file or directory.
(gdb) x/i $pc
=> 0x7ffff7113de <__GI___libc_malloc+286>:      mov     (%r8),%rsi
(gdb) i r $r8
r8                0x0
(gdb) bt 10
#0  tcache_get (tc_idx=<optimized out>) at malloc.c:2937
#1  __GI___libc_malloc (bytes=0x10) at malloc.c:3051
#2  0x00007ffff70b83a6 in ?? () from /lib/x86_64-linux-gnu/libxcb.so.1
#3  0x00007ffff70b5e9f in ?? () from /lib/x86_64-linux-gnu/libxcb.so.1
#4  0x00007ffff70b6329 in ?? () from /lib/x86_64-linux-gnu/libxcb.so.1
#5  0x00007ffff70b63c8 in xcb_writev () from /lib/x86_64-linux-gnu/libxcb.so.1
#6  0x00007ffff7ebca0e in _XSend () from /lib/x86_64-linux-gnu/libX11.so.6
#7  0x00007ffff7ebcf64 in _XReply () from /lib/x86_64-linux-gnu/libX11.so.6
#8  0x00007ffff7eba1fb in XTranslateCoordinates () from /lib/x86_64-linux-gnu/libX11.so.6
#9  0x000000000134b806 in ?? ()
#10 0x000000000134be6c in ?? ()
#11 0x000000000134e488 in ?? ()
#12 0x0000000001328e8d in ?? ()
#13 0x0000000000b3bc82 in ?? ()
#14 0x0000000000b3cbb2 in ?? ()
#15 0x0000000000b3ce4b in ?? ()
(More stack frames follow...)
```

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.

[VULNERABILITY REPORTS](#)

[PREVIOUS REPORT](#)

[NEXT REPORT](#)

[TALOS-2020-1162](#)

[TALOS-2020-1185](#)
