Hacking Excel Online: How to exploit Calc

Nicolas Joly
nijoly@microsoft.com

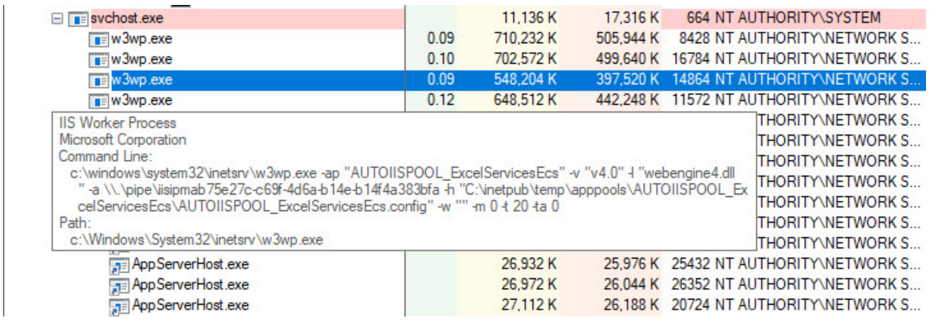
Microsoft

Abstract. The Microsoft Security Response Center has a unique position in monitoring exploits in the wild. While we have seen several cases in the past years of exploits targeting Office applications, often PowerPoint or Word, exploits targeting online applications are less common. Are they even possible? And in which case, how would one attack the Office Online Server? Can a malicious document be used? How hard would that be, how much time would it take? I did a short project during summer 2018 to try to answer these questions with Excel Online. This article describes the bug and the exploit I wrote to get code execution in OOS by loading and interacting with a malicious XLSX spreadsheet. It explores the feasibility of such attacks using Excel's features, and particularly formulas.

1 Attacking Office Online

In comparison with Office Desktop, Office Online, formerly known as Office Web Apps (WAC), needs to be seen under a different perspective when it comes to vulnerabilities and exploitation. Because the applications run server-side, the road to get a functional exploit is less complicated than with traditional desktop applications. There's no specific need for a one-shot exploit where one sends an Office document and hope some exploit magic will just happen on the other side. It's always better to have something that works everywhere, but in the context of Office Online where the attacker is dynamically interacting with his target the prerequisites are simpler. A vulnerability that can produce visible results on which the attacker can quickly interact might be enough. A heap overflow in a cell, a memory disclosure vulnerability while rendering a picture, or even a logic issue that could trigger the deserialization of arbitrary .Net objects would be ideal scenarios. Mateusz "j00ru" Jurczyk from Google for example demonstrated how malicious pictures embedded in Office documents could potentially leak server memory back in 2016 [3]. Can we follow the same steps to get remote code execution on the server?

OOS consists of several services running under IIS, all interacting with an Exchange server. Different scenarios exist, the most common are viewing email attachments or working on documents already present in the cloud. The 2018 Onpremise version supports Word, Excel and Powerpoint among others, and here is a picture of OOS running the Excel service in a test environment on Windows 10 1709:



```
svchost.exe                    11,136 K    17,316 K    664 NT AUTHORITY\SYSTEM
  w3wp.exe            0.09    710,232 K   505,944 K   8428 NT AUTHORITY\NETWORK S...
  w3wp.exe            0.10    702,572 K   499,640 K  16784 NT AUTHORITY\NETWORK S...
  w3wp.exe            0.09    548,204 K   397,520 K  14864 NT AUTHORITY\NETWORK S...
  w3wp.exe            0.12    648,512 K   442,248 K  11572 NT AUTHORITY\NETWORK S...
IIS Worker Process                                         THORITY\NETWORK S...
Microsoft Corporation                                      THORITY\NETWORK S...
Command Line:                                              THORITY\NETWORK S...
  c:\windows\system32\inetsrv\w3wp.exe -ap "AUTOIISPOOL_ExcelServicesEcs" -v "v4.0" -l "webengine4.dll
   " -a \\.\pipe\iisipmab75e27c-c69f-4d6a-b14e-b14f4a383bfa -h "C:\inetpub\temp\apppools\AUTOIISPOOL_Ex
   celServicesEcs\AUTOIISPOOL_ExcelServicesEcs.config" -w "" -m 0 -t 20 -ta 0
Path:                                                      THORITY\NETWORK S...
  c:\Windows\System32\inetsrv\w3wp.exe                     THORITY\NETWORK S...
    AppServerHost.exe          26,932 K    25,976 K  25432 NT AUTHORITY\NETWORK S...
    AppServerHost.exe          26,972 K    26,044 K  26352 NT AUTHORITY\NETWORK S...
    AppServerHost.exe          27,112 K    26,188 K  20724 NT AUTHORITY\NETWORK S...
```

Fig. 1. Process Explorer running on OOS

This article only focuses on Excel. The main library is xlsrv.dll, and unfortunately for the readers symbols are not public. It supports most of the features present in the Desktop application, which means that a bug affecting Desktop will likely affect OOS as well.

2 Finding the right bug

For a product like Excel that has been extensively fuzzed by so many researchers, including of course the product team, blind fuzzing appears unlikely to provide quick and actionable results. With all sorts of mitigations in mind, it is crucial to find the right chain of bugs. At least one bug good enough to leak memory and defeat ASLR, and a second one, potentially the same, to redirect the execution flow. This does not mean that fuzzing Excel does not work, as of 2019 the MSRC received more than 50 reports affecting Excel, with severities rating from low to important. It just tends to illustrate that this technique may have limited results given a restricted timeline. Besides, finding as many bugs as possible in a couple of weeks was clearly out of scope for this project, the question was essentially about the feasibility of an exploit. Exploiting OOS, is this something that we can expect, and if so, how would one do it?

Exploits for memory corruptions without a scripting environment like Excel tend to be rare these days. ASLR and DEP combined usually require an exploit to dynamically calculate addresses before executing a ROP that needs to bypass CFG. While it is true that Chakra has been recently integrated, it does not allow a spreadsheet to run arbitrary scripts. Therefore, is it still possible to craft a one-shot exploit? Excel provides loads of features, for instance formulas. Some of them like *IF* allows conditional branching, *MID* can extract characters from a string, and others like *REPT* or *CONCAT* can potentially do heap spray. Many also work on columns and rows, and can consequently emulate *for* loops (see *VLOOKUP*). With this in mind, it might theoretically be possible to build a spreadsheet that would abuse a vulnerability in a formula, and have the result of that vulnerability propagated to other cells. Imagine one vulnerability to leak a vtable, add some heap spray and build dynamic gadgets, and finish with another vulnerability to cause memory corruption. The question is, can we put this together and build a poc?

This research begins with CVE-2008-4019, *Microsoft Excel REPT Formula Parsing vulnerability* [1] that affected any version of Excel below 2007. The issue at the time was that the repeat formula did not properly validate its second argument, potentially leading to an integer overflow. For example, a formula like *REPT("AAAA", 1073741825)* would force the code to allocate 4 * 0x40000001 bytes, a result truncated to 4 if the operation is done on 32 bits. Depending on the size allocated, this vulnerability was causing either a stack overflow or a heap overflow. Because of the nature of the bug, one can either type the formula and evaluate it in a cell or craft the formula in an XLSX document and load it. Something like that would be enough to add in sheet1.xml (an XLSX file is just a ZIP containing xml and binary files):

```xml
<row r="457" spans="1:1" x14ac:dyDescent="0.25">
<c r="A457" t="str">
<f t="shared" ref="A457:A520" si="7">REPT("ZZZZ",1073741825)</f>
<v/>
</c>
</row>
```

Listing 1. Embedding a malicious formula in a .XML

I personally worked on this issue at the time in 2008 and was curious to see if anything similar remained in the code 10 years later. My first take was to look at the *REPT* formula, see how this one worked exactly. Given 10 years passed since this issue was found, the likelihood of discovering something similar in the code was fairly low, but still provided an interesting exercise for somebody who wanted to familiarize with the code. Without much surprise, the multiplication is now properly checked:

```
case FUNC_REPT:
{
    WCHAR* pch;
    int ichTotal;
    BOOL fOverflow = false;

    ichTotal = CbAllocSafe(ich, cch, 0, &fOverflow);
    if (fOverflow)
        goto LRetErrOom;
```

Listing 2. Checking the REPT parameters

Notice the call here to CbAllocSafe which sets the fOverflow bit if ich * cch + 0 overflows:

```
DECL_CSYM UINT32 __fastcall CbAllocSafe(UINT32 cRec, UINT32 cbRec,
    UINT32 cbExtra, BOOL* pfOverflow)
{
    SAFEINT si;

    si.Init(cRec);
    si.Mult(cbRec);
    si.Add(cbExtra);
    *pfOverflow = si.FOverflow();

    return (si.Acc());
}
```

Listing 3. CbAllocSafe

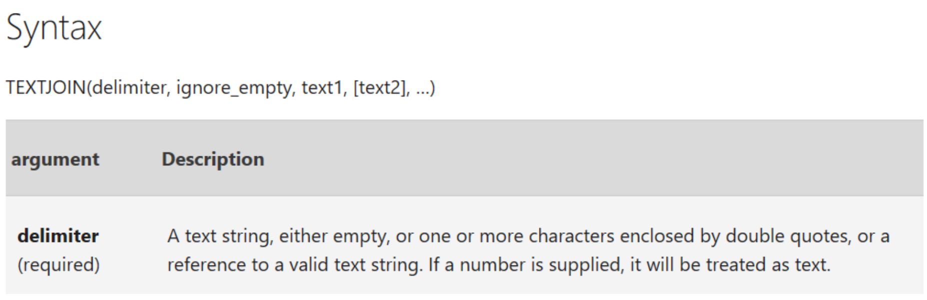
Any attempts to supply malicious integers will then be caught. Therefore, use of this function should reveal locations in the code where dynamic arrays are allocated. As I was aiming at finding a strong primitive, anything dealing with array out-of-bounds was particularly interesting. "X-REFing" with IDA on CbAllocSafe revealed 107 locations in xlsrv.dll where the function was called, and in particular, three occurrences within a function called fnConcatenate including one highly suspicious, with tons of risky Maths done before the call:

```
text:000000018012DBCF            mov     ecx, [rsi+0Ch]
text:000000018012DBD2            lea     r9, [rbp+280h+var_248]
text:000000018012DBD6            sub     ecx, [rsi+8]
text:000000018012DBD9            xor     r8d, r8d
text:000000018012DBDC            mov     eax, [rbp+280h+var_254]
text:000000018012DBDF            add     ecx, r13d
text:000000018012DBE2            sub     eax, r14d
text:000000018012DBE5            add     eax, r13d
text:000000018012DBE8            imul    ecx, eax
text:000000018012DBEB            lea     edx, [r8+8]
text:000000018012DBEF            mov     eax, [rsi+4]
text:000000018012DBF2            sub     eax, [rsi]
text:000000018012DBF4            add     eax, r13d
```

```
text:000000018012DBF7          imul     ecx, eax
text:000000018012DBFA          mov      [rbp+280h+var_244], ecx
text:000000018012DBFD          call     cballocsafe64
```

Listing 4. Some highly suspicious instructions in fnConcatenate

A quick X-REFing reveals that fnConcatenate is reachable from the TEXTJOIN formula, that has the following syntax (the documentation can be found here [2]):



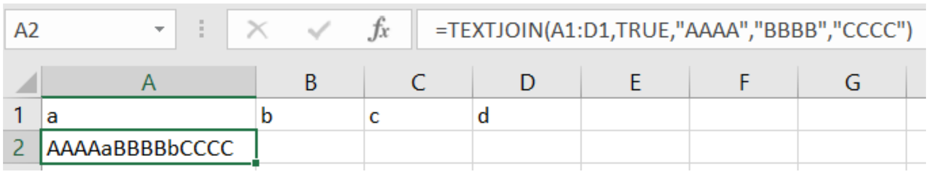
Syntax

TEXTJOIN(delimiter, ignore_empty, text1, [text2], ...)

argument	Description
delimiter (required)	A text string, either empty, or one or more characters enclosed by double quotes, or a reference to a valid text string. If a number is supplied, it will be treated as text.

Fig. 2. How to use TEXTJOIN

Here's a quick example of the formula:



A2 =TEXTJOIN(A1:D1,TRUE,"AAAA","BBBB","CCCC")

	A	B	C	D	E	F	G
1	a	b	c	d			
2	AAAAaBBBBbCCCC						

Fig. 3. Using TEXTJOIN

Delimiter can have all sorts of values, including cell references. In 2015, the function was extended to support three dimensions: columns, rows and sheets. Note the following lines in the source added at that moment corresponding to the new feature:

```
    cDelimiter = pcalcrefData->GetHeight() * pcalcrefData->GetWidth
        () * (isheetLast - isheet + 1);
    cbrgDelim = CbAllocSafe(cDelimiter, sizeof(XCHAR*), 0, &
        fOverflow);

    if (fOverflow)
        goto LRetErr;
    if (!SUCCEEDED(pevalglob->PmemheapRecalcBuffer()->HrAllocPv(
        cbrgDelim, (void**) &rgstDelim)))
        goto LRetErr;
```

Listing 5. A few lines of fnConcatenate

At first sight, assuming we have full control over Height, Width, and the number of sheets provided, it seems possible to get an integer overflow before reaching CbAllocSafe, which makes this function a good candidate for more investigation. Getting a poc isn't too difficult, the following line will magically reach the code path above and crash any vulnerable version of Excel:

```
TEXTJOIN(Sheet2:Sheet10!A1:KZB529328,TRUE,"AAAA","BBBB","CCCC","e")
```

Listing 6. Crafting a poc

Executing this formula on a vulnerable version of Excel will cause a write access violation in fnConcatenate. Why does that formula magically work? KZB indicates a width of 11*26*26 + 26*26 + 2 = 8114, and 529328 * 8114 = 4294967392 which in hex gives 0x100000060. Excel truncates then that number to 0x60 and uses it times the number of sheets to allocate an array on the heap, in this case an array of (10-2+1)*0x60 = 0x360 bytes. Follow three loops in which pointers to strings are stored in the freshly allocated array rgstDelim:

```
    while (true)
    {
        for (rw = pcalcrefData->RwFirst(); rw <= pcalcrefData->
            RwLast(); rw++)
        {
            for (col = pcalcrefData->ColFirst(); col <= pcalcrefData
                ->ColLast(); col++)
            {
                xlsoper.FastInit();
...
                rgstDelim[iIndexDelimiter] = stDelimItem;
```

Listing 7. Loops overwriting the heap

A few constraints to have in mind, Excel only supports up to 1048576 rows and 16384 columns (0x100000, 0x4000). Luckily, A1:KZB529328 fits well in there. For bugs like this, we can either start with the final number we want to obtain and have it decomposed into prime factors or just run a solver to return all the possible solutions.

Loops with huge counters often result in wild access violations which are hard to exploit. This is not the case here, as developers took care of the case where a cell would contain an error. Such cells are marked by a specific Err property, and when the code encounters them it just exits the loops, frees anything allocated and returns an error:

```
    if (xlsoper.FIsErr())
    {
        hr = xlsoper.HrFinalizeAndTransferErrorResult(pxlsoperRes);
        if (FAILED(hr))
        {
            fNeedDoJmp = true;
        }
        xlsoper.FastFree();
        goto LDoneConcat;
    }

...

LDoneConcat:
    pevalglob->SetPenvMem(penvSav);

    for (iIndexDelimiter = 0; iIndexDelimiter < cDelimiterAllocated;
         iIndexDelimiter++)
    {
        PchBufReleaseXls(pevalglob->PmemheapRecalcBuffer(),
            const_cast <XCHAR*>(rgstDelim[iIndexDelimiter]));
    }

    pevalglob->PmemheapRecalcBuffer()->FreePv(rgstDelim);
```

Listing 8. Encountering an error will exit the loops and free rgstDelim

If we include a buggy cell in our references we will be able to exit the loop and have the overflow controlled, the hardest part, finding a suitable bug, now seems over. This vulnerability was documented as CVE-2018-8331. Note also CVE-2018-8574 a similar vulnerability affecting the formula Forecast.ETS but not available in Excel Online at the time.

Readers who have access to a Visual Studio Subscription and who wish to experiment can download Office Online Server with the November 2017 update. This version should be vulnerable to CVE-2018-8331 and reproducing should be straightforward.

3 Exploiting the issue

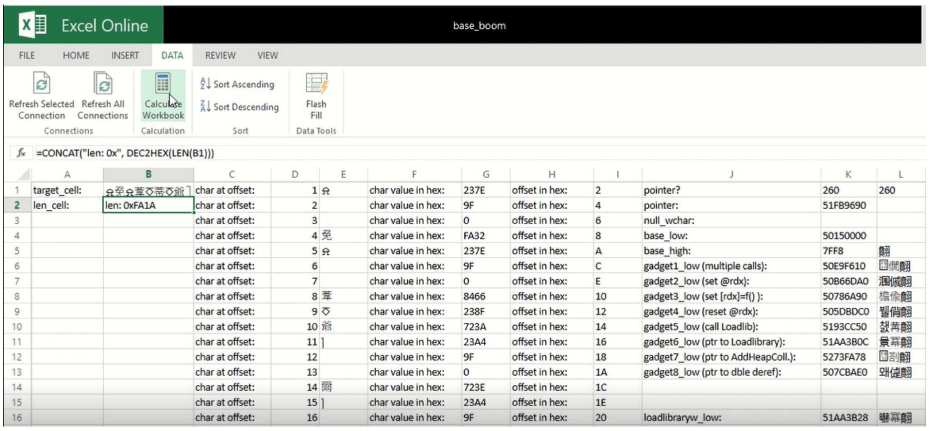
The primitive obtained is strong but not perfect to the eyes of the exploit writer. On the plus side, one can arrange the various variables to allocate an array whose size is controllable, which provides some flexibility regarding what sort of object we would like to overwrite. There are certainly some constraints on the rows and columns, but the poc shows that fairly small arrays can still be allocated. It is however and unfortunately not possible to overwrite the heap with arbitrary bytes, only pointers to strings. Besides as we're forced to cause an error to exit those loops and end fnConcatenate, those strings are ephemeral, they are allocated in fnConcatenate and free()'d before the function returns, which means that data in the heap will be overwritten by dangling pointers. In summary, we can allocate an (almost) arbitrarily sized array, and write past its bounds as many pointers to free()'d strings as we want. In general, this kind of bug provides a primitive strong enough to bypass the most common mitigations, as described in the following lines.

The first thing that one can notice is how Excel allocates strings in memory. Those consist of a size followed by the string itself, which means that if an attacker can touch the string's length, he can then trick Excel into reading past its bounds to read and disclose heap memory. This is how the attack should work. First spray the heap with strings and objects, free some of these strings, trigger the vulnerability and hope that the vulnerable array will be allocated right before a string to guarantee successful corruption. This sounds simple when described like this in a few words, but in Excel Online, just deleting a cell containing a string in a spreadsheet does not necessarily free the string in memory. This is because of the "Undo" mechanism. User actions are recorded, and the user can always revert and come back to an initial state. Because of this, the user does not have direct control over the memory, unless he chooses to *recalc the form*, in which case all modifications stored in the undo chain are lost. The exploit works then in multiple steps, all separated by a click to *recalc*. This is where user interaction with the exploit is essential, as forging a one-shot exploit seems at first sight very complicated. In summary the plan is as follows: first spray the heap with strings, manually delete a couple of them, and eventually *recalc()*, to free the deleted strings in memory.

At that point we can now trigger the bug and hope that the array is allocated in one of these holes. If this succeeds, the length of one string in our spray will be changed to the lowest bytes of a pointer. We get then a nice heap visualizer on this specific part of the memory.

4 What to leak, can we get a read/write primitive?

The ideal scenario would be an object with a vtable allocated in the middle of these strings. Is it possible to achieve that with formulas only? A quick review of all the supported formulas did not reveal any interesting



=CONCAT("len: 0x", DEC2HEX(LEN(B1)))

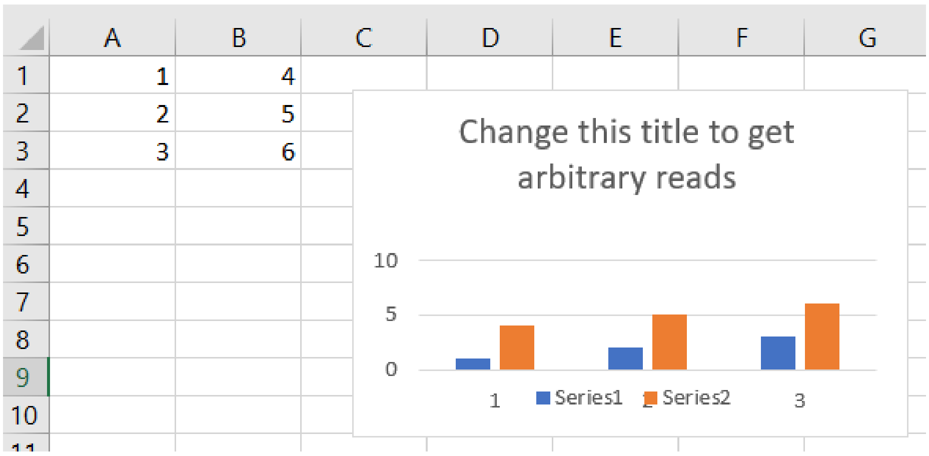
	A	B	C	D	E	F	G	H	I	J	K	L
1	target_cell:	⍾兎⍾萦⎏蔕⎏鈽⎤	char at offset:	1	⍾	char value in hex:	237E	offset in hex:	2	pointer?	260	260
2	len_cell:	len: 0xFA1A	char at offset:	2		char value in hex:	9F	offset in hex:	4	pointer:	51FB9690	
3			char at offset:	3		char value in hex:	0	offset in hex:	6	null_wchar:		
4			char at offset:	4	兎	char value in hex:	FA32	offset in hex:	8	base_low:	50150000	
5			char at offset:	5	⍾	char value in hex:	237E	offset in hex:	A	base_high:	7FF8	翸
6			char at offset:	6		char value in hex:	9F	offset in hex:	C	gadget1_low (multiple calls):	50E9F610	[illegible]僩翸
7			char at offset:	7		char value in hex:	0	offset in hex:	E	gadget2_low (set @rdx):	50B66DA0	潿傶翸
8			char at offset:	8	萦	char value in hex:	8466	offset in hex:	10	gadget3_low (set [rdx]=f()):	50786A90	檐偸翸
9			char at offset:	9	⎏	char value in hex:	238F	offset in hex:	12	gadget4_low (reset @rdx):	505DBDC0	뷀偝翸
10			char at offset:	10	爺	char value in hex:	723A	offset in hex:	14	gadget5_low (call Loadlib):	5193CC50	챐蒓翸
11			char at offset:	11	⎤	char value in hex:	23A4	offset in hex:	16	gadget6_low (ptr to Loadlibrary):	51AA3B0C	㬌冪翸
12			char at offset:	12		char value in hex:	9F	offset in hex:	18	gadget7_low (ptr to AddHeapColl.):	5273FA78	[illegible]剳翸
13			char at offset:	13		char value in hex:	0	offset in hex:	1A	gadget8_low (ptr to dble deref):	507CBAE0	뫠偼翸
14			char at offset:	14	爾	char value in hex:	723E	offset in hex:	1C			
15			char at offset:	15	⎤	char value in hex:	23A4	offset in hex:	1E			
16			char at offset:	16		char value in hex:	9F	offset in hex:	20	loadlibraryw_low:	51AA3B28	㬨冪翸

Fig. 4. Disclosing pointers and building gadgets

candidates, but Excel supports so many features that more investment might reveal something actionable (typically a formula that changes a graph layout, or alters a pivot table that may result in unexpected and useful results). Regarding the primitive, it should be noted that although we can corrupt the length of a string, this string is a read-only object in Excel, and as such, it is not possible to change its content directly with formulas. In other words, once the string is created in memory, it cannot be modified by the UI, any modification to a string stored in a cell will result in another string allocated. This in combination with the constraints of the vulnerability makes it difficult to craft a read/write primitive. However, once a string is corrupted, it is possible to change the memory around and read it at any moment. If one formula changes, the chain can be automatically recalculated. My first try was with Title objects associated to Graph objects. Because of the specifics, we needed to find an object that once corrupted could survive several (at least 2) dereferences. These objects occupy 0x140 bytes and hold a pointer to a string at a certain offset. By overwriting a pointer to the Title object in the Graph object, it would be possible to get a double dereference and, in the end, control the location of the string.

Manually changing those titles unfortunately caused more troubles than expected and after trying various scenarios I couldn't get that strategy to work reliably.

I opted in the end for a more traditional way, freeing some strings, *recalc()ing* again and inserting some Graph objects to fill the free space with an object of 0x300 bytes beginning with a vtable. Those are fine,



	A	B
1	1	4
2	2	5
3	3	6

Fig. 5. A simple graph with a title

because 0x300 is a multiple of 0x60 (remember the poc above), and chances are that all these allocations will land in the same segment. A first pass would then read the vtable, along with pointers to the heap segment, and a second pass would trigger the issue to overwrite the vtable with a pointer to a controlled string. The rest of the exploit was simple, as it turns out that Excel would eventually use the vtable when the object is manually resized, leading then later to remote code execution by loading a library from a remote share.

5 Would that work in a real-world scenario?

The main issue with the steps above is that they don't apply very well to a production environment, with potentially dozens of users interacting with the service at the same time, thus significantly reducing the probability to get the heap layout in a predictable state. Memory leaks are easy to get, but how to make sure that holes are allocated in the right way is a different question. It's worth noting though that the entire attack could be automated by scripting on the attacker side. In the end, interacting with Office Online is just a matter of sending HTTP requests and parsing the response, so having to *recalc()* or choose which strings to delete should not be an issue, although this is something that I have not personally tried. The exploit works however quite well in a lab environment, with a single user working on one spreadsheet at a time, where the heap can be put more easily into a predictable state. This project nevertheless gave the proof that attacks like these are possible, not only theoretical, and that provided with a good vulnerability, an attacker has enough features to play with to build a working exploit.

References

1. Microsoft Office Excel REPT Formula Parsing Remote Code Execution Vulnerability. `https://www.zerodayinitiative.com/advisories/ZDI-08-099/`.
2. TEXTJOIN function. `https://support.office.com/en-us/article/textjoin-function-357b449a-ec91-49d0-80c3-0e8fc845691c`.
3. Mateusz “j00ru” Jurczyk. Windows Metafiles - An Analysis of the EMF Attack Surface and Recent Vulnerabilities (CVE-2016-3263 - slide 182). `https://j00ru.vexillium.org/slides/2016/metafiles_full.pdf`, 2016.