64-bit calc.exe Stack Overflow Root Cause Analysis :

Author : Souhail Hammou

Blog : http://rce4fun.blogspot.com

Twitter : https://twitter.com/Dark_Puzzle

Contact : dark-puzzle@live.fr

Introduction :

I was surfing around the net when I found by chance a blog post about a bug found in calc.exe under Win7 64bit.
http://marcoramilli.blogspot.com/2013/08/bug-in-wincalcexe.html
I've tried to reproduce right away in the same environment and then calc.exe crashed. I hadn't any stuff going on so I said to myself why not do a root cause analysis of the bug ? and as I started taking notes it turned out to be an article... So here it is :
This article is about my analysis of the bug so if you noticed something that appears to be wrong or unclear don't mind e-mailing me about it, I'll be glad :) .

Reproducing the crash :

The crash can be simply reproduced by following these steps :

- Open calc.exe and calculate 1/255.
- Choose the [F-E] button.
- Crash !! .

I reproduced with calc.exe attached to Windbg and the following was displayed :

```
(2f0c.2618): Stack overflow - code c00000fd (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll!RtlpAllocateHeap+0x30:
00000000`77c23520 89542434       mov     dword ptr [rsp+34h],edx
```

The status code shows us that this is a stack overflow , and the exception is triggered when an instruction will try to write to an address that is believed to belong to the stack but it's not. This is because the stack has run out of memory and can't commit anymore space.
The stack committed and reserved memory can be examined by dumping the PE header using "!dh" commadn under Windbg :

```
0000000000080000 size of stack reserve
0000000000002000 size of stack commit
```

size of stack reserve : The total stack size used by the application.
size of stack commit : The stack space that can be used by the application until hitting the guarded page (a page from reserved memory). The access to that page is now granted and the page guard is the next page.This process will continue until "size of stack reserve" is reached which will result in a STACK OVERFLOW EXCEPTION, for the simple reason that the application will try to write out of the stack bound.

Now we need to know at least what has caused the Stack Overflow Exception , so all we have to do is check the call stack.

```
RETADDR                    CALL SITE
00000000`77c234d8 :        ntdll!RtlpAllocateHeap+0x30
00000000`77ca70dd :        ntdll!RtlAllocateHeap+0x16c
00000000`77c6b5aa :        ntdll!RtlDebugAllocateHeap+0xcd
00000000`77c234d8 :        ntdll! ?? ::FNODOBFM::`string'+0x18b42
000007fe`fdb81635 : ntdll!RtlAllocateHeap+0x16c
00000000`ff5d1d89 : KERNELBASE!LocalAlloc+0x71
00000000`ff5d5ed4 :        calc!_createnum+0x2d
00000000`ff5d5c4d :        calc!_addnum+0x64
00000000`ff5d7514 : calc!addnum+0x67
```

```
00000000`ff5d27da :          calc!_divnum+0x154
00000000`ff5d5aa8 :          calc!putnum+0x14d
00000000`ff5d5aa8 : calc!putnum+0x22d
00000000`ff5d5aa8 : calc!putnum+0x22d
00000000`ff5d5aa8 : calc!putnum+0x22d
00000000`ff5d5aa8 : calc!putnum+0x22d
00000000`ff5d5aa8 : calc!putnum+0x22d
00000000`ff5d5aa8 : calc!putnum+0x22d
[...]
00000000`ff5d5aa8 : calc!putnum+0x22d
```

Oops, the cause of the problem is a recursive call . It has written its arguments , return address and local variable and arguments to other function calls so many times until the stack has been exhausted.
But why ? that's the question we'll be answering until the last sentence of this article.
For some of you , it appears useless to look for a root cause of this kind of bug and maybe it is. But I found working on this enjoyable cause this is the first bug I encountred in calc.exe and also it was a good opportunity for me to become more familiar with x64 reversing.

The first thing I started with is the calc!putnum function itself , what's its prototype and what are its parameters.
In our case the putnum function is called by calc!putrat :

Caller : (calc!Putrat)

```
00000000`ff29571c 488d542448     lea    rdx,[rsp+48h]
00000000`ff295721 448bc5         mov    r8d,ebp
00000000`ff295724 498bcc         mov    rcx,r12
00000000`ff295727 e834010000     call   calc!putnum (00000000`ff295860)
00000000`ff29572c 488b4c2448     mov    rcx,qword ptr [rsp+48h]
```

Callee : (calc!Putnum)

```
00000000`ffd55860 488bc4         mov    rax,rsp
00000000`ffd55863 44894018       mov    dword ptr [rax+18h],r8d (argument_3)
00000000`ffd55867 48895010       mov    qword ptr [rax+10h],rdx (argument_2)
00000000`ffd5586b 48894808       mov    qword ptr [rax+8],rcx (argument_1)
00000000`ffd5586f 53             push   rbx
00000000`ffd55870 55             push   rbp
...
00000000`ffd55873 4883ec48       sub    rsp,48h
```

Keep in mind that these arguments will be accessed from now on using rax , and rsp will be used to store the functions' local variables.
In x64 , argument passing consists the use of rcx,rdx,r8 and r9 , any other additional parameters passed will be pushed on stack.
Here in our case the callee will place arguments passed to it (in rcx,rdx,r8d) on stack in the same way the caller would have pushed them.

```
00000000`0010c348  00000000`ff7a572c (RetAddr) 00000000`0010c3b0 (Arg1_rcx)
00000000`0010c358  00000000`0010c398 (Arg2_rdx)00000000`00000001 (Arg3_r8d)
00000000`0010c368  00000000`00217f50           00000000`003c6ba4
00000000`0010c378  00000000`00000000           00000000`003c6810
00000000`0010c388  00000000`ff7a5c8f (RetAddr) 00000000`003c6b90
00000000`0010c398  00000000`001f9670           00000000`00000001
00000000`0010c3a8  00000000`003c6ba4           00000000`00000001 <- c3b0
00000000`0010c3b8  00000000`0020e040           00000000`003c6810
```

The image above shows :

- calc!putrat stack frame .
- the return address for the calc!putnum (return to calc!putrat+0xb8) being just pushed.
- the arguments freshly placed on stack by the callee.

Full disassembly of calc!putnum : http://pastebin.com/imDN8DUa
After some analysis here's what I came to concerning the putnum function :
Argument_1 appears to be a pointer to a stack address which has the value "00000000`00000001" (was set originally as a dword so it is of type int).

Argument_2 appears to be a pointer also to a stack address which has a pointer also.Thus,Argument_2 is a pointer to a pointer.

Argument_3 this appears to be of type int : mov dword ptr [rax+18h],r8d.

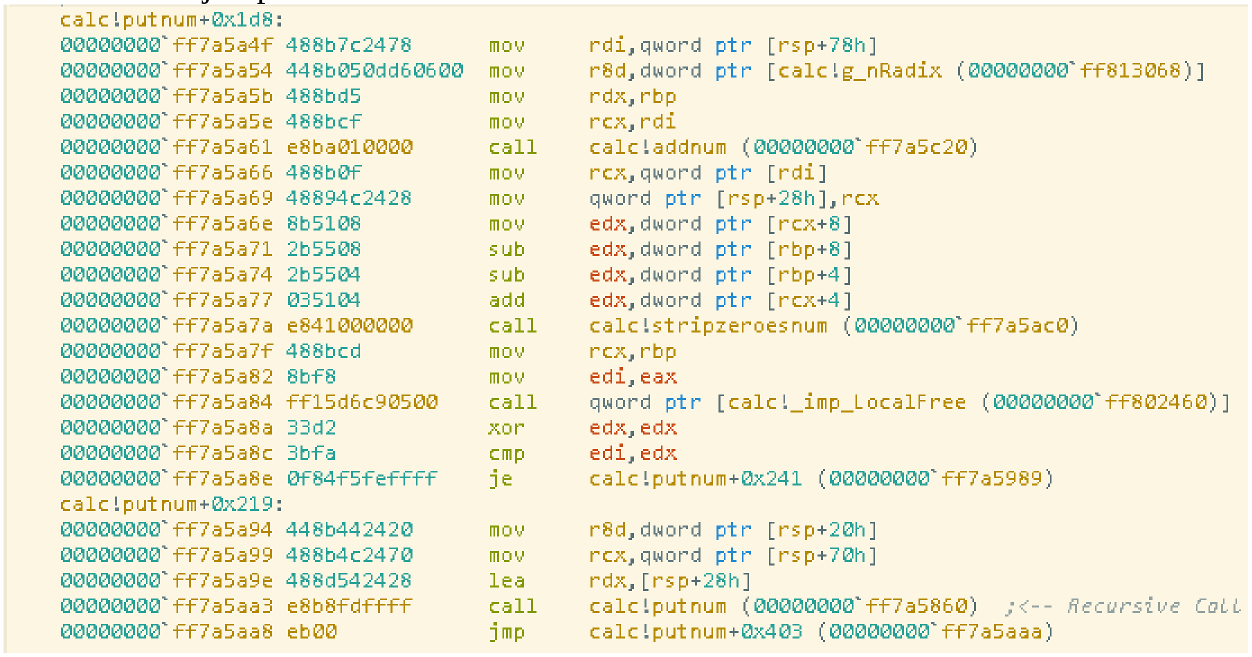
Argument_2 is a pointer to a pointer to an Array of integers that we will see in detail soon.

The function returns a pointer to a unicode string . The string is nothing but the "valid" array integers converted to be ready for the display.
C/C++ Function prototype :

wchar_t* putnum(int* arg1,int arr,int arg3);**

We will be mainly interested in argument_2 (int** arr).

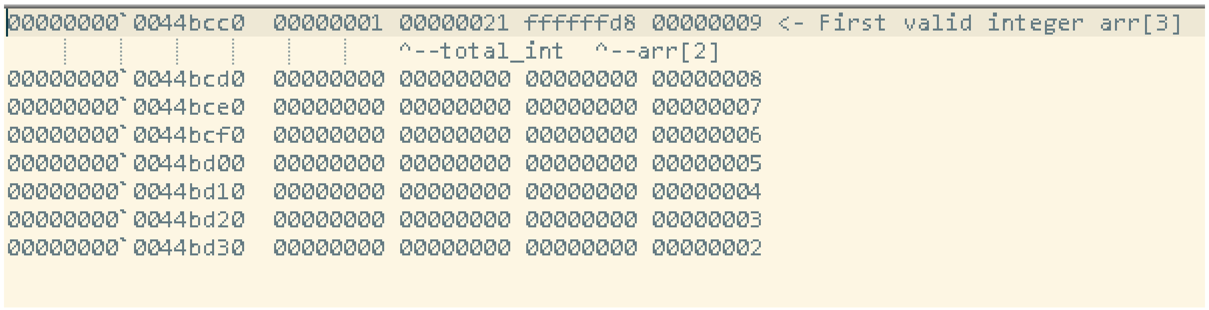
Now all we need to do is spot the recursive call and see how it's reached following conditional and unconditional jumps.



```
calc!putnum+0x1d8:
00000000`ff7a5a4f 488b7c2478      mov     rdi,qword ptr [rsp+78h]
00000000`ff7a5a54 448b050dd60600  mov     r8d,dword ptr [calc!g_nRadix (00000000`ff813068)]
00000000`ff7a5a5b 488bd5          mov     rdx,rbp
00000000`ff7a5a5e 488bcf          mov     rcx,rdi
00000000`ff7a5a61 e8ba010000      call    calc!addnum (00000000`ff7a5c20)
00000000`ff7a5a66 488b0f          mov     rcx,qword ptr [rdi]
00000000`ff7a5a69 48894c2428      mov     qword ptr [rsp+28h],rcx
00000000`ff7a5a6e 8b5108          mov     edx,dword ptr [rcx+8]
00000000`ff7a5a71 2b5508          sub     edx,dword ptr [rbp+8]
00000000`ff7a5a74 2b5504          sub     edx,dword ptr [rbp+4]
00000000`ff7a5a77 035104          add     edx,dword ptr [rcx+4]
00000000`ff7a5a7a e841000000      call    calc!stripzeroesnum (00000000`ff7a5ac0)
00000000`ff7a5a7f 488bcd          mov     rcx,rbp
00000000`ff7a5a82 8bf8            mov     edi,eax
00000000`ff7a5a84 ff15d6c90500    call    qword ptr [calc!_imp_LocalFree (00000000`ff802460)]
00000000`ff7a5a8a 33d2            xor     edx,edx
00000000`ff7a5a8c 3bfa            cmp     edi,edx
00000000`ff7a5a8e 0f84f5feffff    je      calc!putnum+0x241 (00000000`ff7a5989)
calc!putnum+0x219:
00000000`ff7a5a94 448b442420      mov     r8d,dword ptr [rsp+20h]
00000000`ff7a5a99 488b4c2470      mov     rcx,qword ptr [rsp+70h]
00000000`ff7a5a9e 488d542428      lea     rdx,[rsp+28h]
00000000`ff7a5aa3 e8b8fdffff      call    calc!putnum (00000000`ff7a5860)  ;<-- Recursive Call
00000000`ff7a5aa8 eb00            jmp     calc!putnum+0x403 (00000000`ff7a5aaa)
```

cmp edi,edx : In the case when the call to putnum isn't made : edi = edx which means that edi value is NULL. EDX always equals 0 before the comparison , thus now only EDI now is the problem.
I found that EDI switches its value between 1 and 0.
When triggering the bug , EDI value is always 1 before the comparison. (Putnum is called everytime).
This value of edi is returned from calc!stripzeroesnum in eax.

The Array that I talked about is something like this :

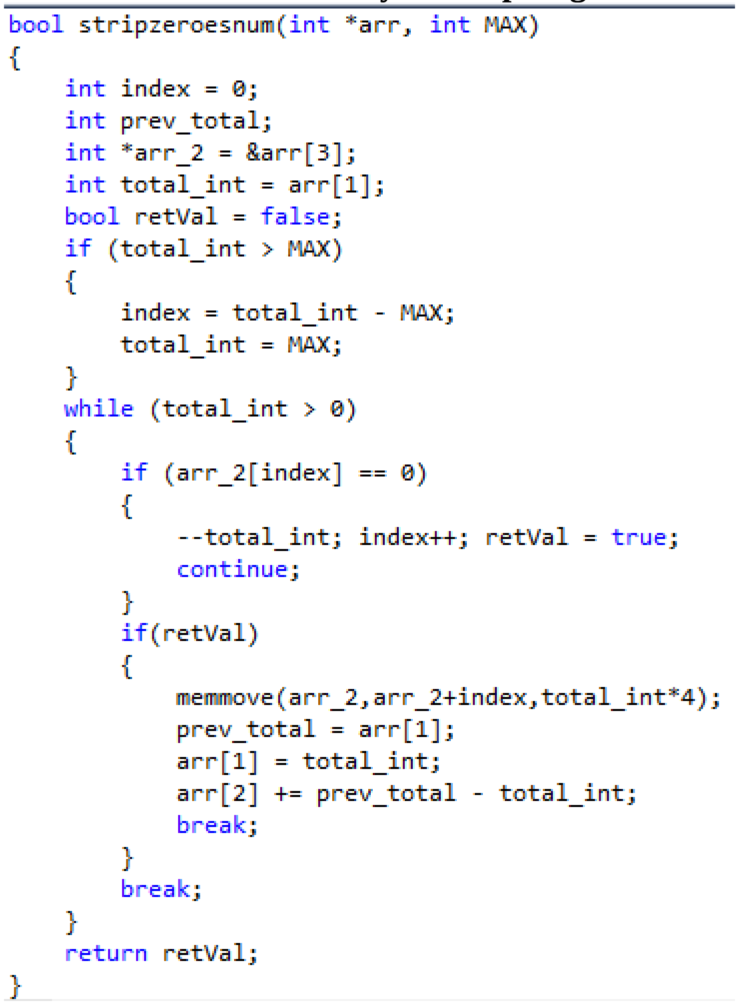


```
00000000`0044bcc0  00000001 00000021 ffffffd8 00000009 <- First valid integer arr[3]
                            ^--total_int  ^--arr[2]
00000000`0044bcd0  00000000 00000000 00000000 00000008
00000000`0044bce0  00000000 00000000 00000000 00000007
00000000`0044bcf0  00000000 00000000 00000000 00000006
00000000`0044bd00  00000000 00000000 00000000 00000005
00000000`0044bd10  00000000 00000000 00000000 00000004
00000000`0044bd20  00000000 00000000 00000000 00000003
00000000`0044bd30  00000000 00000000 00000000 00000002
```

arr[2] has a negative value , we add to it the difference between the previous and current total .
This will increase its value , because zeroe(s) have been stripped.
The value (arr[2] - 1) is the length of the whole array. In my opinion , the negative value describes that the integers are stored in the inverted way.
More information about calc!stripzeroesnum :

Full disassembly of the function : http://pastebin.com/jSt2Ufh0

I worked also on manually decompiling the function because it's important for us:



```
bool stripzeroesnum(int *arr, int MAX)
{
    int index = 0;
    int prev_total;
    int *arr_2 = &arr[3];
    int total_int = arr[1];
    bool retVal = false;
    if (total_int > MAX)
    {
        index = total_int - MAX;
        total_int = MAX;
    }
    while (total_int > 0)
    {
        if (arr_2[index] == 0)
        {
            --total_int; index++; retVal = true;
            continue;
        }
        if(retVal)
        {
            memmove(arr_2,arr_2+index,total_int*4);
            prev_total = arr[1];
            arr[1] = total_int;
            arr[2] += prev_total - total_int;
            break;
        }
        break;
    }
    return retVal;
}
```

A part of this function compares an element of the array to NULL , if it is : it simply increments the index and set the return value to 1 then compare again again until finding a value that is different from NULL. Besides, it calls a function calc!memmove which will Strip all the zeros from the array so the starting of the array (where the integers to convert are stored , however the first element of the array is at (first_integer_index - 3)) will be the first element different from NULL. This function also uses ebx as a counter for the elements in the array so it decrements whenever the next element is accessed.
P.S : The function is supplied a Maximum value that the valid integers in the array don't have to exceed , this value is supplied through edx . The array contains more than just the integers for example the array[1] contains the total numbers of the integers to process .

If total_int > MAX then the array will be accessed from the element array[total_int-MAX] to make the number of integers in the wanted interval and then the Max value itself will be used as a counter. If total_int < MAX then the counter will still ebx by default and the array will be accessed from index 3.
The compare instruction is : cmp dword ptr [rdx],0
RCX is supplied to the function and RCX+0xC is the first element of the array which holds the integers . These integers are nothing but the result of the arithmetic operation. The integers are stored in the reverse order in memory so the last ones are the first ones listed in memory.
calc!stripzeroesnum+0x19: mov ebx,dword ptr [rsi+4] , ebx holds now the number of the elements.The elements are only the Xs "0.XXX..." where X is different from NULL. This is the second element of the array which holds the total number of valid integers (total_int).
e.i : for 0.000005 , ebx will hold the value 1.

PS : the screen can display 34 characters max for integers that start with 0,XXXX (0 and the point are counted) and 33 characters for numbers that start with X,XXXXX... (the point is counted).

In this article we will be interested in 2 calls to calc!stripzeroesnum , conditions will control each execution flow of the execution .
the first call is at : "calc!putnum+0x34".
The second one is at : "calc!putnum+0x203".
The main condition that controls the access to the array is a comparison between ebx and r8d :

```
cmp  ebx,r8d
jg  calc!stripzeroesnum+0x26
```

we dicussed the value of ebx earlier (total_int), and the value of r8d is supplied by the caller in edx.So we will be studying 2 cases .
Concerning the call at "calc!putnum+0x34" : The caller supplies as an argument edx with a value of 0x20+2. calc!g_maxout == 0x20 the value 2 is added in case the number to display contains a point and an additional number (0.) for example.
Concerning the call to stripzeroesnum at "calc!putnum+0x203": In this call the array is accessed using an index in RAX the comparison starts with the first element accessed.
This isn't done until the result is bigger to be displayed , to fix this , the array is accessed using an index which simply is the result of the substraction between the total integers to process in the array and the max value allowed.

```
 sub     ebx,r8d
 movsxd  rax,ebx
 mov     ebx,r8d
 lea     rdx,[rcx+rax*4]
```

Now the question is Why does the crash happen only when pressing [F-E] button and not when clicking "=" button ??!
Actually ,sometimes, there's a majoration when trying to display the result using the "=" button when the value is long enough , a majoration would simply make the following value 0.0090090090090090090090090090009... , this one 0.00900900900900901.
The function putnum calls a function (calc!addnum) which does that , and suprisingly this function does a majoration just in case we want to display the number using "=" button and it does not do it when we're trying to display the scientific notation using [F-E].

The Crash Cause :

A Quick Resumé :

- As was mentioned earlier, the addnum function will do a majoration if needed to decrease the length and precision. Sometimes , in a normal case , the recursive call will be accessed but the MAX will be greater than total_int so the execution will not reach calc!stripzeroesnum.

- Surprisingly , when choosing the [F-E] mode , the addnum will not do any majoration and will sometimes add more precision to the number.
- Remember when I said that the array ignores the zeroes after the point and before the first number different from zero ?

```
"0.0016316168515"
   ^^----Ignored.
```

Those will be added later to the total_int and then compared with calc!g_maxout global variable which is equal 20.

```
calc!putnum+0x186:
mov     eax,ecx
sub     eax,esi  <-- ESI has a negative value so it is added to eax.
cmp     eax,r8d
jg      calc!putnum+0x1b3
```

Conditions To Reproduce The Crash :

The difference is that when clicking "=" button a majoration of the number will be made to stop getting more precision , then the zeroes at the end will be stripped off. However , when choosing [F-E] mode in special cases (1/255 , 1/111 , 1/999) the addnum will always add precision, and total_int+zeros_after_the_point > g_maxout , which will take us to stripzeroesnum at putnum+0x203 again .This one will access the array from the index (total_int-MAX) which points always to a NULL elements then strip off zeros. When the recursive call is made the stripzeroesnum+0x34 will get an array that is stripped off of zeroes. However total_int > calc!g_maxout or total_int+zeros_after_the_point > calc!g_maxout.
Then As soon as the call to addnum is made : more precision will be added making the array larger. The array should exceed the max value and making (total_int - MAX) poiting to a NULL element again which will result in returning 1 (stripping the zero and numbers before in the array) and calling putnum again which will call addnum ....etc until the stack is fully overflown.

Conclusion :
As demonstrated in this article , the use of recursive calls is strongly unappreciated because under special circumstances they may cause a fatal stack overflow bug.
Well , That's all FOLKS ... And see you again soon.

Thanks for your time.
Regards,
Souhail Hammou.