# An Analysis of the Excel 2007 “65535” Bug

Chris Lomont, www.lomont.org, Nov 2007, Version 1.2[1]

## Overview

On September 22, 2007, a serious Excel 2007 bug was reported on a newsgroup [7] and was soon featured on numerous news sites (Slashdot [2], Digg [3], News.com [4]). The bug showed up when a user tried to multiply 850 by 77.1, which should result in 65535. However Excel 2007 returns 100,000, as shown in Figure 1.

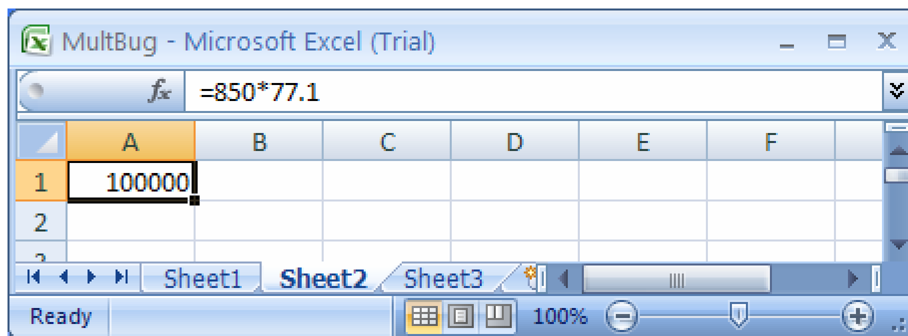



**Figure 1 – Excel 2007 bug. The result should be 65535.**

Similarly, Excel 2007 mis-formats the value 65536-2^(-37) as 100001, as in Figure 2.

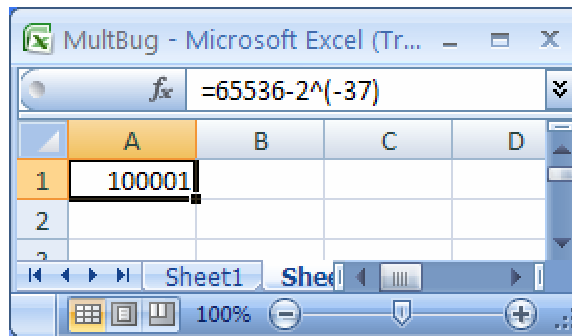



**Figure 2 – Another view of the same bug.**

Soon a Microsoft Excel blog site [1] reported that the bug was only in rendering and not used internally for other calculations. Since the value 850*77.1 + 1 erroneously results in 100,001, this confused many people, making them think this was indeed a math bug and that the internal value was incorrect. Unfortunately, this second example happened to hit one of the other values affected by the bug. 850*77.1-1 and 850*77.1+2 both return the correct values, 65534 and 65537 respectively. The site claimed exactly 12 of the 9.2*10^18 possible 64-bit floating-point values suffer from this bug, with six values

[1] V1.0, Oct 2007, Initial release. Version 1.1 Oct 2007, minor typos. Version 1.2, Nov 2007, typos.

between 65534.99999999995 and 65535, and six between 65535.99999999995 and 65536.

This note:
1. details how the bug works,
2. shows the bug is a rendering bug, not a math error as many reported,
3. shows how it was likely introduced by comparison to Excel 2002 and Excel 2000 behavior (the bug seems to have been inserted when updating an older 16-bit formatting routine to a 32-bit equivalent),
4. explains how the just released hotfix corrects the behavior, confirming the analysis of the bug,
5. and demonstrates why exactly twelve values out of more than 9*10^18 (approx 2^63) possible 64-bit floating-point values suffer from this bug.

In particular, I disassembled Excel 2007, located the source of the bug, and found the error to be in the 64-bit floating-point to string conversion routine. I did a comparison to similar routines from Excel 2000, Excel 2002, and Excel 2007 with the hotfix for this error.

One reason I investigated the code is that security vulnerabilities are often found near bugs in programs, due to complexity, poor programming, oversights, etc. Since this bug can conceivably be accessed from a rogue Excel spreadsheet, there was some chance it was a security hole. Under detailed analysis I found no security hole.

Another reason I did this work was to provide details on the scope and (lack of?) severity of the bug, in contrast to the numerous bloggers and news stories that speculated on all sorts of wild fantasies about this bug. Digg humorously titled their article "Critical Excel 2007 bug cripples users[2]," and although I repeatedly saw the bug during testing, I am still pretty healthy.

A final reason I dissected the code is to practice my skills at taking apart software and understand how things work. Taking this apart, and especially being successful at doing it, has been a rewarding experience. I wrote this up for the fun of it.

The bug seems to be introduced when the formatting routine was updated from older 16-bit assembly code used in previous versions of Excel to a presumably faster 32-bit version in Excel 2007. It is surprising such a bug slipped through, but to anyone thinking they can write an IEEE 754 floating-point to text routine using only bit twiddling and integer math with no "sprintf" cheating, please try to write one and see how hard it is to get right!

During my analysis of the bug Microsoft released a hotfix, which was integrated into my earlier version of this document.

[2] http://digg.com/microsoft/Critical_Excel_2007_bug_cripples_users

## Floating-point format

For overview, here is how floating-point values are stored (roughly) on a PC. They are stored in what is called IEEE 754 [12] format, which is a specification giving bit layout, size requirements, and accuracy requirements for floating-point operations. Here is why such things are needed:

Any real number can be written as powers of 2. Integers are simple: 100=64+32+4, which can be written as consecutive powers of two as 1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0, or in binary, as $1100100_2$. This extends to all real numbers using negative powers of two: 0.5 = 2^(-1), 3/8 = 1/4 + 1/8 = 0*2^(-1) + 1*2^(-2) + 1*2^(-3)=$0.011_2$. A computer stores numbers as a finite string of these bits.

However, numbers such as 0.1 cannot be exactly represented since they require an infinite length base 2 expansion, 0.1 = $0.000110011001100\ldots_2$. So when you enter 0.1 into a floating- point value the resulting number stored and used in computations is slightly less due to truncation. When 77.1 was entered and then multiplied by 850, the result internally is really 65535-2^(-37), which the old routine correctly rounded to 65535 when printing. The new routine failed.

Due to this misunderstanding of the limits of computability, message boards discussing the Excel bug are filled with people claiming to have found many other bugs, like 4.1 – 4 returning 0.09999999999 instead of exactly 0.1. As shown, it is impossible to compute 4.1-4 exactly using IEEE 754 format numbers[3] – the best one can do is approximate answers.

### *IEEE 754*

IEEE 754 floating-point 64-bit numbers are stored using 1 bit for the sign, 11 bits to store an exponent, and 52 bits to store the mantissa, which is where the "digits" are stored. This is shown in Figure 3. A good way to think of this is that the format stores 52 bits of the expansion (of possibly infinite length) for a number, and the exponent explains where the sliding window takes a snapshot of the digits. Most often the left edge of the window is chosen one past the leading 1 digit in the binary expansion.

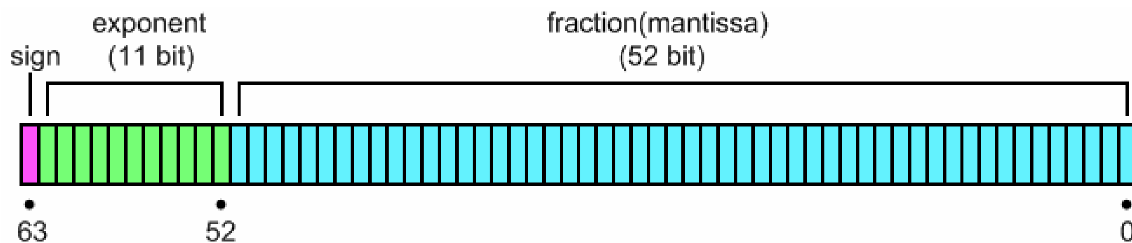



**Figure 3 - IEEE 754 bit layout**

[3] Without using numerical tricks and other techniques, which make a lot more possible. But these tricks are often unacceptably slow for the types of computation needed in Excel.

The sign bit is 0 for positive values and 1 for negative values. The 11-bit exponent E takes integer values 0-2047, and is biased by 1023, giving a true exponent e=E-1023. The mantissa M is left shifted until the highest 1 bit shifts out of the window (called *normalized*). This leading 1 bit is then discarded and the rest of the mantissa bits are stored, giving an extra bit of precision. Write the stored value as V=2^(E-1023)*(1.M) = 2^e * (1.M), using the 1.M notation to show the implied 1 bit and that the mantissa M is the fractional part.

52 bits of mantissa corresponds to 15 digits of decimal accuracy[4], so Excel traditionally rounds numerical answers to 15 digits.

There are other subtleties for denormalized[5] numbers, infinities, underflow, and NaN (Not-a-Number) bit settings, but we don't need them here. More details are in my article on the Inverse Square Root [10] on my website or my article on floating-point hacks in Games Programming Gems 6 [11]. There are also many other places to learn these details, but the Games Gems article is pretty detailed and clear.

For this article we'll use the word "number" to denote a real number, and "value" to denote a representation of a number in 64-bit IEEE 754 floating-point format. Thus 0.1 is a number, but there is no value for it. The closest value is slightly smaller and is what gets stored in an IEEE 754 format.

## Values

The twelve erroneous values shown in Table 1 were found and posted on [1].

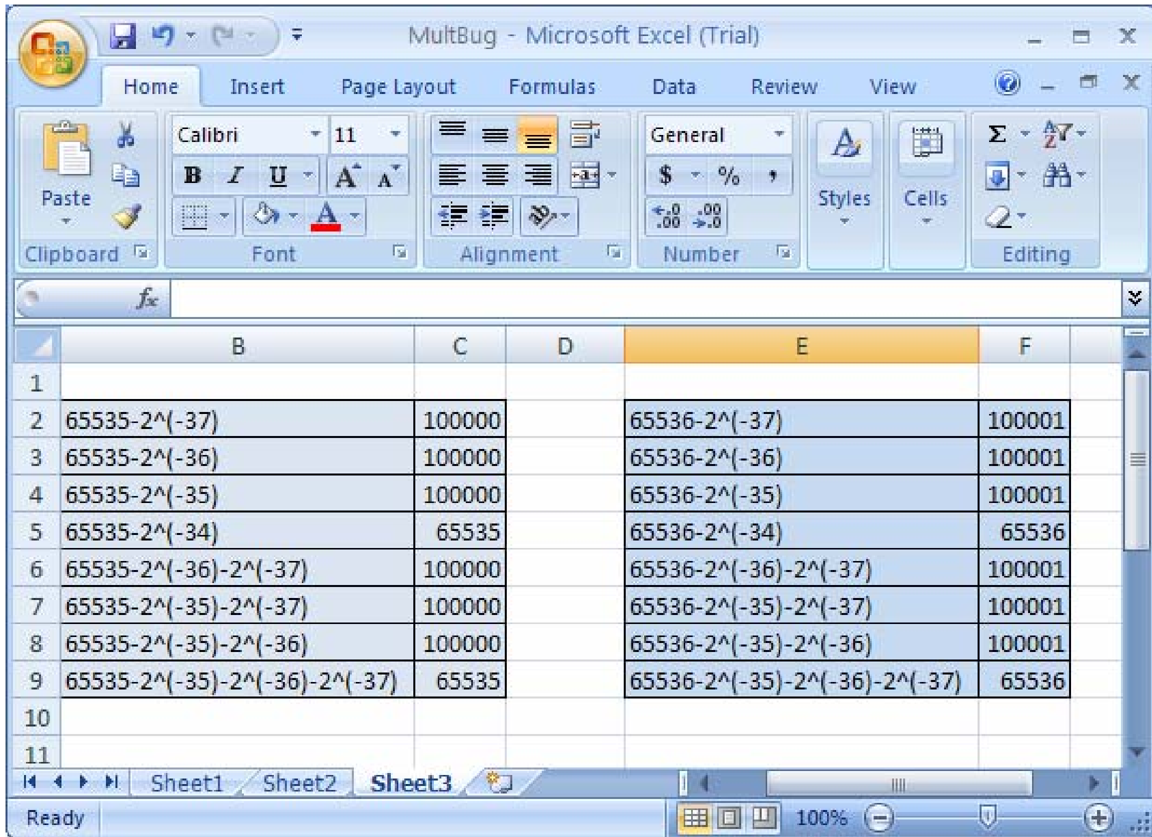
| Value | Hex | Value | Hex |
|---|---|---|---|
| 65535-2^(-35) | 40efffdf fffffffa | 65536-2^(-35) | 40efffff fffffffa |
| 65535-2^(-36) | 40efffdf fffffffb | 65536-2^(-36) | 40efffff fffffffb |
| 65535-2^(-37) | 40efffdf fffffffc | 65536-2^(-37) | 40efffff fffffffc |
| 65535-2^(-35)-2^(-36) | 40efffdf fffffffd | 65536-2^(-35)-2^(-36) | 40efffff fffffffd |
| 65535-2^(-36)-2^(-37) | 40efffdf fffffffe | 65536-2^(-36)-2^(-37) | 40efffff fffffffe |
| 65535-2^(-36)-2^(-37) | 40efffdf ffffffff | 65536-2^(-36)-2^(-37) | 40efffff ffffffff |

**Table 1 – Twelve values Excel 2007 formats wrong.**

[4] $\mathrm{Log}_{10}(2^{52}) < 15.5 < \mathrm{Log}_{10}(2^{53})$

[5] For very small numbers which are at the edge of the possible exponent values, the leading 1 is no longer implied, but shown, and the mantissa represents all the digits. These non-normalized numbers (called denormals) are required by IEEE 754.

The left half values all evaluate incorrectly to 100,000, and right half values all evaluate incorrectly to 100,001. These values can be directly entered into Excel, as shown in Figure 4 (along with some nearby correctly formatting values).



| | B | C | D | E | F |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | 65535-2^(-37) | 100000 | | 65536-2^(-37) | 100001 |
| 3 | 65535-2^(-36) | 100000 | | 65536-2^(-36) | 100001 |
| 4 | 65535-2^(-35) | 100000 | | 65536-2^(-35) | 100001 |
| 5 | 65535-2^(-34) | 65535 | | 65536-2^(-34) | 65536 |
| 6 | 65535-2^(-36)-2^(-37) | 100000 | | 65536-2^(-36)-2^(-37) | 100001 |
| 7 | 65535-2^(-35)-2^(-37) | 100000 | | 65536-2^(-35)-2^(-37) | 100001 |
| 8 | 65535-2^(-35)-2^(-36) | 100000 | | 65536-2^(-35)-2^(-36) | 100001 |
| 9 | 65535-2^(-35)-2^(-36)-2^(-37) | 65535 | | 65536-2^(-35)-2^(-36)-2^(-37) | 65536 |
| 10 | | | | | |
| 11 | | | | | |

**Figure 4 – All 12 erroneous values displayed.**

These values all are of the form 0x40EFFFyF FFFFFFFz where y=D or F and z = A, B, C, D, E, or F. Immediate questions are why precisely this pattern? For example, why cannot y=E? What about z=9? The reasons these are the only 12 values are covered in the Analysis section.

Roughly, the main reason that y cannot be E is then the value is near 65535.5, and the non-integer output goes down a different conversion path in the floating-point to string code, a path that works correctly. The values for z below those listed avoid setting a certain carry, which triggers again a different yet correct piece of code.

The C++ program in the appendix demonstrates that 850*77.1 in IEEE 754 results in `0x40 ef ff df ff ff ff ff`. Some constants used in the C++ code and following table are

```
// some negative powers
double e35 = pow(2.0,-35.0);
double e36 = pow(2.0,-36.0);
double e37 = pow(2.0,-37.0);
double e38 = pow(2.0,-38.0);
```

Here is the output from the C++ program showing IEEE values for various expressions similar to the one under consideration.

```
0x40 ef ff e0 00 00 00 00  =    65535    = 65535
0x40 ef ff df ff ff ff ff  =    65535    = 850*77.1
0x40 ef ff ff ff ff ff ff  =    65536    = 850*77.1+1
0x40 ef ff ef ff ff ff ff  =    65535.5  = 850*77.1+0.5
0x40 f0 00 00 00 00 00 00  =    65536    = 65536
0x40 ef ff e0 00 00 00 00  =    65535    = 65535-e38
0x40 ef ff df ff ff ff ff  =    65535    = 65535-e37
0x40 ef ff df ff ff ff fe  =    65535    = 65535-e36
0x40 ef ff df ff ff ff fd  =    65535    = 65535-e36-e37
0x40 ef ff df ff ff ff fc  =    65535    = 65535-e35
0x40 ef ff df ff ff ff fb  =    65535    = 65535-e35-e37
0x40 ef ff df ff ff ff fa  =    65535    = 65535-e35-e36
0x40 ef ff df ff ff ff f9  =    65535    = 65535-e35-e36-e37
0x40 ef ff ff ff ff ff ff  =    65536    = 65536-e37
0x40 ef ff ff ff ff ff fe  =    65536    = 65536-e36
0x40 ef ff ff ff ff ff fd  =    65536    = 65536-e36-e37
0x40 ef ff ff ff ff ff fc  =    65536    = 65536-e35
0x40 ef ff ff ff ff ff fb  =    65536    = 65536-e35-e37
0x40 ef ff ff ff ff ff fa  =    65536    = 65536-e35-e36
0x40 ef ff ff ff ff ff f9  =    65536    = 65536-e35-e36-e37
```

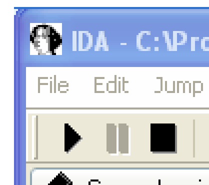
Notice that 850*77.1 is stored internally the same as 65535-2^(-37), not as 65535, and that 2^(-38) is too small to make a difference.

## Locating the bug

To locate the bug, I downloaded the Excel 2007 trial from Microsoft, installed it in a VMWare image, and used IDAPro to disassemble/debug it. Here are roughly the steps I followed.

1. Run Excel from IDA Pro. Surprisingly there were no anti-debug hooks in Excel, which surprised me. I was expecting a fight to remove anti-debugging code before finding the bug itself.

2. Enter =850*77.1 into cell A1. Excel outputs the incorrect value 100,000.




3. Break into Excel by pausing IDA Pro.

4. Open a hex view, showing the entire memory space visible to Excel.

5. Find the output string "100000" in memory (Figure 5). I used search sequence of bytes, and looked for "100000." No hits. I switched to "case sensitive" and "use

Unicode," reran it, and found dozens of hits. 100000 might conceivably be used in a lot of places: documentation, system constants, various output strings, so I tried a hopefully less common string….

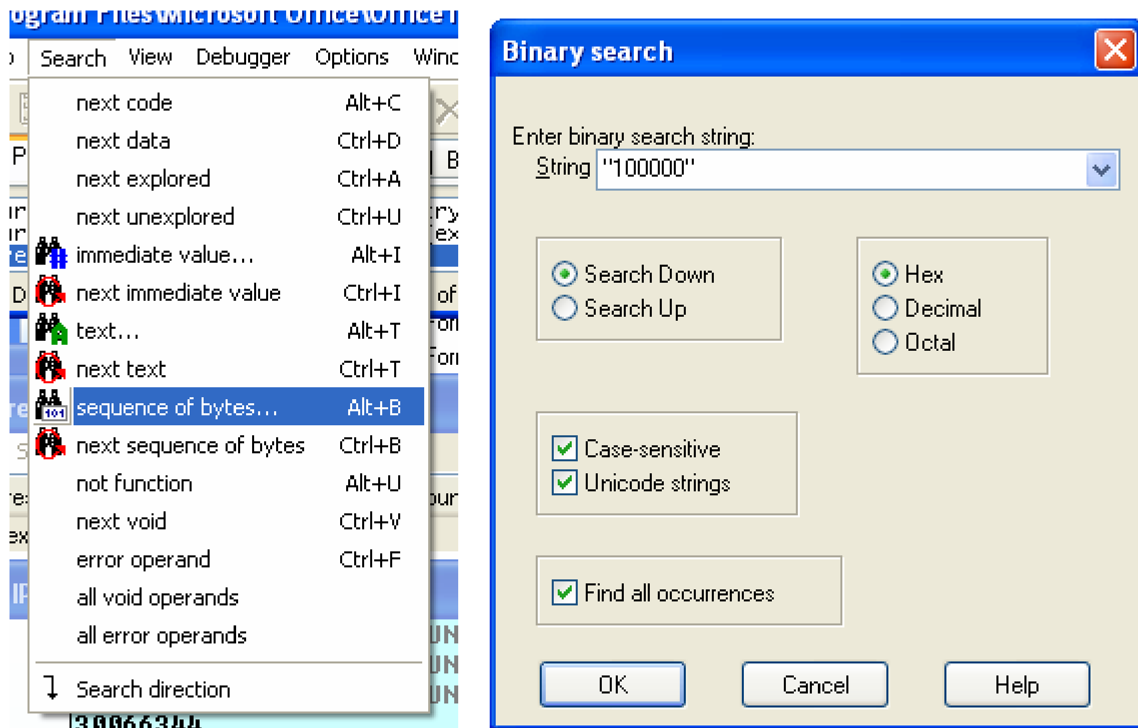



**Figure 5 – Locating strings in Excel**

6. I re-ran the steps above using another of the incorrect values 850*77.1+1, which gives the wrong value 100,001, which seemed less likely to common. This resulted in only a few hits. One was address 0x30EABF14 (your address may vary).

7. To find the routine that created the 100001 string at address 0x30EABF14, I created a hardware breakpoint (Figure 6) that stops whenever an instruction read or wrote to this location. x86 architecture luckily can break when an instruction access a memory address; without this locating the instruction writing this string would be very difficult.

8. I then went to Excel, highlighted the cell, and pressed enter to cause a recompute. IDAPro's debugger broke on the location in Figure 7. A few more runs showed there were several

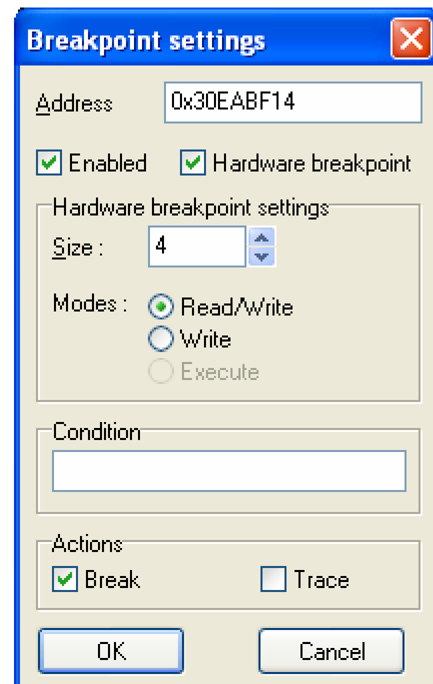



**Figure 6 – IDAPro hardware breakpoint**

locations all accessing the memory in question.

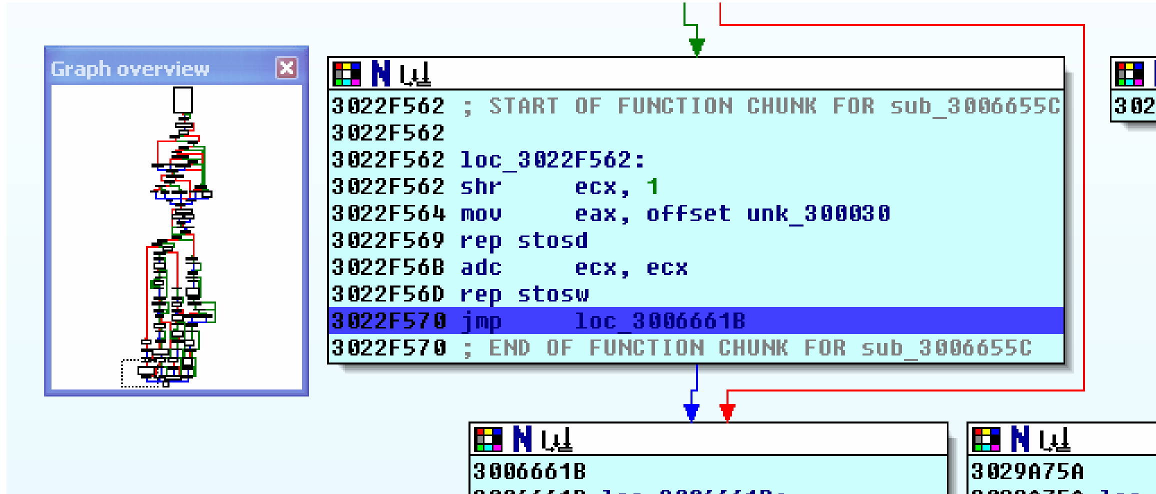



**Figure 7 – IDA Pro disassembly and graph view.**

9. I now had a plausible location to dig around. After looking through the code at each of the places that stepped on this memory location, I found the routine creating the erroneous formatting at address `.text:0x30066344`.

10. After some analysis, I understood a bit of this routine – it is a routine that converts the binary representation of a 64-bit IEEE 754 double to a Unicode text string. It is disassembled and dissected in the Bug Disassembly and Analysis section.

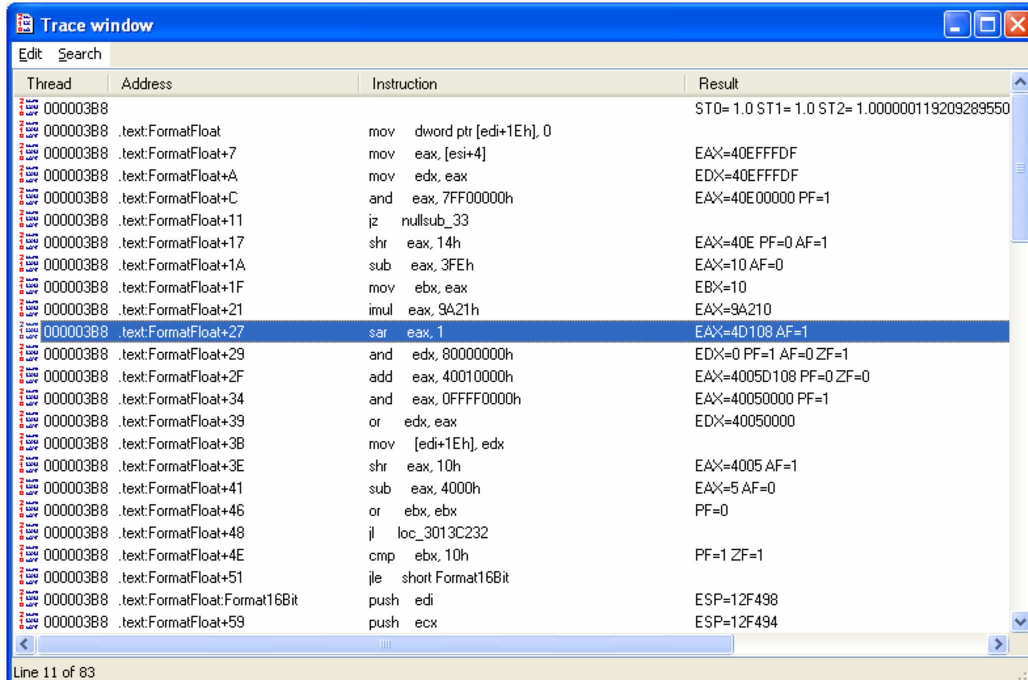



**Figure 8 – Instruction tracing.**

11. The routine was very complicated and most of it was not touched by this bug. To narrow down the analysis, I ran an instruction trace, which only records instructions executed and stores register values through the tracing (Figure 8). To gather runs of data, I caused a break at the start of the formatting routine and formatted the various bug values. Enabling "Trace instructions" and running the routine until it returned to caller gathered a run of all the registers and instructions executed.

With data gathered on successful formatting for nearby values and examples of incorrect formatting, I dissected the resulting code.

### *Faulty routine location*

To assist others in stepping through this routine, here are the locations of the buggy routine.

The Excel version I dissected was Excel.exe, file version 12.0.4518.1014. File offset 0x65474 starts with the bytes **`C7 47 1E 00 00 00 00`**, which marks the first instruction of the formatting routine, "**`mov dword ptr [edi+1Eh], 0`**."

The block in memory when running looks like this, with the starting offset being **`0x30066344`**.

```
.text:30066340  5D C2 08 00 C7 47 1E 00  00 00 00 8B 46 04 8B D0
.text:30066350  25 00 00 F0 7F 0F 84 29  E8 0C 00 C1 E8 14 2D FE
.text:30066360  03 00 00 8B D8 69 C0 21  9A 00 00 D1 F8 81 E2 00
```

Given the routine location, it should be easy to break there and debug it yourself. OllyDbg[6] and Visual Studio should work as well as IDAPro.

## Bug Disassembly and Analysis

### *Call Graph*

IDAPro generated graphs for the floating-point to string routines for Excel 2002 and Excel 2007, shown in Figure 9. There are a few cases where this routine calls outside subroutines, but not in the path that affects the bug. Unfortunately the right side of the call graph for Excel 2007 is not really needed, but IDAPro added it anyways. So the actual graph is somewhat simpler by about a third. The highest box on the right and its descendants should be removed to get a fairer comparison.

One thing that stands out though is the increase in complexity of the routine, and this clearly shows that it was modified. From analysis it seems this was mostly done for

[6] http://www.ollydbg.de/

performance reasons, changing from 16-bit registers to 32-bit ones, and as is often the case, the increased complexity of a high performance routine leads to a higher incidence of bugs.

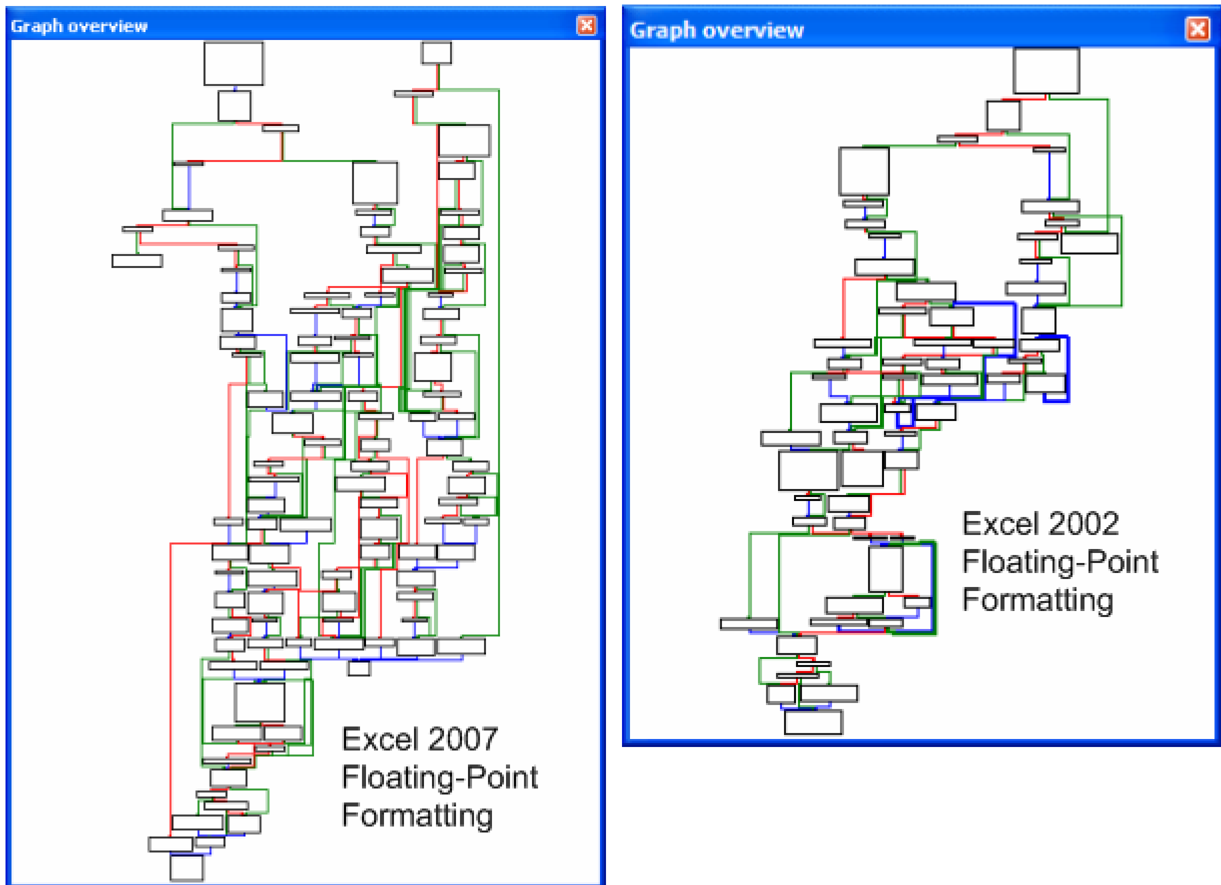



**Figure 9 – Routine call graph comparisons.**

The code seems to be written directly in assembly, since it has no C/C++ style stack frame or register usage. Also, the usage of some rare assembly instructions[7] also points to it being hand coded assembly. This was likely done for performance – converting floating-point values to text needs to be high performance for Excel.

The biggest difference between the 2002 and the 2007 versions is that the routine was rewritten to use 32-bit registers instead of 16-bit ones. As shown below this led to a subtle bug, causing the formatting error.

The function in question takes a pointer to the floating-point value to convert in register ESI, and writes out a text string to EDI, which also points to the beginning of a structure, likely a cell information structure. The routine seems to fill only as many leading digits as are nonzero, and a calling routine then fills the remaining text buffer with "0"s. Also, this

[7] Such as `shld, scasw,` and `cwde.`

routine does not seem to place a decimal point, but it does return the number of digits placed and presumably the location of the decimal point.

An outline of the routine is as follows:

1. Given the float value V to output, find E so 2^E >= V
2. For decimal output, Find D so 10^D >= 2^E using the first magic constant. This tells how many digits are needed for output.
3. Based on the size of the output, choose a formatting routine. Certain ranges of values allow faster routines to be chosen, which is why this step seems to be here. This is also where the bug occurs, and it appears to have come directly from the 16-bit to 32-bit code rewrite.
4. The routine for 16-bit or less mantissa is chosen, but a divisor table pointer is pointing to the wrong divisor due to the bug.
5. A loop outputs digits, and returns.

In brief: a digit loop takes a value N, a pointer to a table of divisors {10000,1000,100,10,1}, and uses the table to output decimal digits. A pointer is initialized to point to the largest table value needed based on the number of digits being output. When N=65535 and the pointer to the table is correct, this outputs 6,5,5,3,5. The bug causes the table pointer to point one past the 10000 entry to a 65535 entry.

Thus, when N=65536 (with the needed small error to cause the table mismatch), the first digit output is 65536/65535 = 1, with a remainder of 1. Then the divisor loop walks the table for 10000, 1000, 100, 10, and finally 1, outputting ‘0’ for each division, and a 1 in the units place. Thus the output is the erroneous 100001, instead of 65536.

The values near 65535 work similarly.

## *Tracing 850*77.1+1 to 100001 formatting*

Here is a trace showing the error, and how values other than the six listed avoid the error. This trace walks through the value 850*77.1+1 which should be 65536, but formats to 100001 instead. I chose this value for demonstration because it is easier to see the bug than in the 100000 case, but the same analysis works for the other values. Due to rounding errors from the IEEE 754 format, the computation creates the floating-point value `0x40EF FFFF FFFF FFFF` instead of true 65536 which would be `0x40F0 0000 0000 0000`. This should still format correctly when rounded, but a bug causes this to fail.

**Note that the bug causing 65535 -2^(-37) to format as 100000 is not quite the same bug as this one, but is in the same section of code, and is also caused by the table being misaligned. The misalignment is due to another error than the one here. To save space I do not detail this trace here.**

**Tables used**

First come two tables used in the routine. The first contains divisors (powers of 10) used to extract digits from the mantissa, along with a “cap” of 65535 to denote the top size in the table. The second table contains constants used to round IEEE values for 15 significant digit output.

```
// table of divisors, used to extract decimal digits, at .text:301102D0
byte DivTbl[] = {
      0x01, 0x00, 0x00, 0x00, 0x0A, 0x00, 0x00, 0x00,  // values     1,    10,
      0x64, 0x00, 0x00, 0x00, 0xE8, 0x03, 0x00, 0x00,  // values   100,  1000,
      0x10, 0x27, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00}; // values 10000, 65535,

// table of rounding values, used to get 15 digit accuracy, at .text:300663F7
byte RndTbl[] =
      {
       0x49, 0x68, 0x01, 0x00, // digits = 1
       0xE1, 0x12, 0x0E, 0x00, // digits = 2
       0xCC, 0xBC, 0x8C, 0x00, // digits = 3
       0xF8, 0x5F, 0x7F, 0x05, // digits = 4
       0xB3, 0xBF, 0xF9, 0x36  // digits = 5 - value=0x36F9BFB3 used here
      };
```

Now we trace through the routine to see the bug in action.

**Determining answer size**

This part of the routine determines the size of the answer, and selects the appropriate formatting routine based on the output type, number of bits, larger than 0, etc. A tricky part to decode was the magic constant 0x9A21, which was used to multiply the exponent. This turns out to be 2^17 * log2 rounded up, which converts the base 2 exponent to a base 10 exponent, allowing the number of decimal digits before the decimal point in the answer to be extracted.

```
// We are formatting a 64-bit IEEE 754 value V=2^e*(1.M)
// The 11-bit exponent E in the bit representation is e+1023.
// the 52 bit mantissa is the fractional part, with an implied
// 1 bit, hence written 1.M
// The value is 850*77.1+1 in Excel, which is 65536-2^(-37)
// The hex representation of the value V is is 0x40EFFFFF FFFFFFFF

// Initial registers
ST0= 1.0 ST1= 1.0 ST2= 1.0000001192092895508 ST3= 5.9604644775390625e-8
ST4= 9.9999994039535522461e-1 ST5= 1.0 ST6= 0.0 ST7= 1.0
CTRL=137F CS=1B DS=23 ES=23 FS=3B GS=0 SS=23
EAX=FFFE EBX=FFED061E ECX=F EDX=12F4C0 ESI=12F548 EDI=12F9E2
EBP=12F4C4 ESP=12F49C EFL=202

// EDI - points to output structure, first entry is output text buffer.
// ESI - points to the hex value for the float.
Format mov  dword ptr [edi+1Eh], 0       // result value - assume 0
       mov  eax, [esi+4]    EAX=40EFFFFF // high part of double 850*77.1+1
       mov  edx, eax        EDX=40EFFFFF // store a copy
       and  eax, 7FF00000h  EAX=40E00000 // mask out to get exponent E only
       jz   nullsub_33                   // if 0 exponent (V=0,denormalized)
                                         // so bail out, string then '0'.
       shr  eax, 14h        EAX=40E      // move E to low word
       sub  eax, 3FEh       EAX=10       // subtract 1022, which leaves e+1,
                                         // e the true exponent
       mov  ebx, eax        EBX=10       // save this: 2^EBX > V
```

```
        imul eax, 9A21h      EAX=9A210     // x92A1 = ceil(2^17 * log2)
                                           // this imul gives base 10 exponent
                                           // in 15.17 fixed point
        sar  eax, 1          EAX=4D108     // now EAX holds signed 16.16 fixed
                                           // point base 10 exponent
        and  edx, 80000000h  EDX=0         // Get sign into EDX
        add  eax, 40010000h  EAX=4005D108  // rounds up, and some number foo
        and  eax, FFFF0000h  EAX=40050000  //
        or   edx, eax        EDX=40050000  // restore sign
        mov  [edi+1Eh], edx                // save value (decimal point place?)
        shr  eax, 10h        EAX=4005      // to low word
        sub  eax, 4000h      EAX=5         // EAX = # base 10 digits before '.'
                                           // - determines table location later
        or   ebx, ebx        PF=0          // see if exponent < 0 (V < 1)
        jl   loc_3013C232                  // if so, format using other routine
        cmp  ebx, 10h        PF=1 ZF=1     // is V <= 2^16?
        jle  short Bit16                   // if in this cutoff range, can use
                                           // routine (we can use 16 bit math?)
                                           // else V outside range [0,65536)

Bit16   push edi             ESP=12F498    // save these (used later for digit
        push ecx             ESP=12F494    // counting)
        mov  edx, [esi]      EDX=FFFFFFFF  // get low part of the double value
        mov  esi, [esi+4]    ESI=40EFFFFF  // and the high part
        xchg eax, esi        EAX=40EFFFFF ESI=5 // EAX = V high 32 bits,
                                           // ESI = number of decimal digits,
                                           // used for table index
        shl  esi, 2          ESI=14        // 4 bytes per divisor table entry
        and  eax, 0FFFFFh    EAX=FFFFF     // EAX = mantissa high bits
        or   eax, 100000h    EAX=1FFFFF    // prepend 1 bit since normalized
        mov  ecx, 15h        ECX=15        // shift so EAX=integer part only
        sub  ecx, ebx        ECX=5         // shift value, move fraction out
        mov  ebx, eax        EBX=1FFFFF    // make a copy for later shift
        shr  eax, cl         EAX=FFFF      // shift out fractional part,
                                           // EAX = integral part
        neg  cl              ECX=FB        // shift rest of mantissa
        shld ebx, edx, cl    EBX=FFFFFFFF  // low mantissa bits
        shl  edx, cl         EDX=F8000000  // shift bits up...
        mov  ecx, edx        ECX=F8000000  // and place here. Integer in EAX,
                                           // fractional in EBX:ECX
        sub  esi, 4          ESI=10        // divisor table start index
        cmp  eax, DivTbl[esi]              // see if the integral part >= 65535
        jnb  short tag2                    // jump if so
                                           // (else ESI = ESI-4 omitted)
tag2    test ecx, ecx                      //
        jz   loc_300664A3
        add  ecx, RndTbl[esi] ECX=2EF9BFB3 // adds 0x36F9BFB3, representing
                                           // 2^(-35)+2^(-36)=4.36557*10^-11,
                                           // rounds to 15 decimal digits.
        adc  ebx, 0          EBX=0         // add carry up to EBX, and jump if
        jb   tag3                          // carry (means value near integer)
```

All the above work determined that the value satisfies the 16-bit formatting routine size requirements, and added a rounding value to the value for output. Register EAX holds the

integer part, and EBX:ECX holds the fractional part. The rounding overflowed into EBX, which then overflowed, indicating the answer is close to an integer, and a jump was taken accordingly. This overflow will be accumulated into EAX below.

**Why only six values?**

The addition of a magic constant was used to round the mantissa based on number of digits to create the proper rounded 15 decimal digit answer. For the bug to happen, the EBX has to overflow, causing the code below to execute, which leads to a bad table start position. Table 2 shows the relation between the last byte of the mantissa to the value in ECX to the overflow situation. In this case the value 2^(-35)+2^(-36) was added.

| Last byte of Mantissa | Resulting value in ECX | Result when 0x36F9BFB3 Added | Carry? |
|---|---|---|---|
| 0xFF | 0xF8000000 | 0x12EF9BFB3 | Yes |
| 0xFE | 0xF0000000 | 0x126F9BFB3 | Yes |
| 0xFD | 0xE8000000 | 0x11EF9BFB3 | Yes |
| 0xFC | 0xE0000000 | 0x116F9BFB3 | Yes |
| 0xFB | 0xD8000000 | 0x10EF9BFB3 | Yes |
| 0xFA | 0xD0000000 | 0x106F9BFB3 | Yes |
| 0xF9 | 0xC8000000 | 0x0FEF9BFB3 | NO |
| 0xF8 | 0xC0000000 | 0x0F6F9BFB3 | NO |

**Table 2 – Rounding overflow**

This explains why only those values ending in `0xFF-0xFA` are affected. Other values avoid the overflow into EBX, and thus avoid the route needed to misalign the table pointer. Now back to the story.

**The Bug**

From comparison to Excel 2002, the error seems to occur in the following section. There is a `jz skip` (jump if zero) instruction that fails to do its intended job now exactly when the `EAX` register contains `xFFFF`. This code is reached when the above EBX overflows and EAX is incremented. In Excel 2007, the 32-bit `inc eax` causes an increment to the high 16 bits of the register. In Excel 2002 this is the 16-bit version `inc ax`, which rolled over, not setting any bits in the high part. In Excel 2007, **the jump is mistakenly not taken**, which then goes on to change the value of the divisor table pointer ESI, leading to an incorrect initial digit being computed. In the 16-bit version only the lower 16 bits were considered for the zero comparison, causing `AX=0xFFFF` to take the jump. Follow carefully.

```
tag3   xor   ecx, ecx          ECX=0         // integer-1 to in EAX, fract in EBX
       mov   edx, 1            EDX=1         //
       inc   eax               EAX=10000     // carry from EBX into EAX=65536
       jz    skip                            // Excel 2002 jumps here since prev
                                             // instruction was 16-bit inc ax
                                             // Jumping here prints correctly?
       cmp   eax, DivTbl[esi+4]  PF=0        // check against max div value
       jb    loc_300664A3                    // EAX too big?
       jmp   tag4                            // jump to some fixup routine??
```

```
tag4   cmp  eax, 0FFFFFFFFh CF=1          // leads to other class of bugs. . .
       jz   Digits                        //
       inc  word ptr [edi+20h]            // move decimal point again
       add  esi, 4          ESI=14        // move table value (incorrect!!!!!)
       jmp  Digits                        // jump to digit printing loop
```

**One bad digit gets another**

Now we're in position to create the digits. All it takes to make them correct is for ESI to point to the 10000 position in the divisor table, instead of one entry past that to the 65535 entry. By pointing too high with EAX=65536, the first digit is basically the base 65535 digit, which gives the incorrect '1', and then the remainder 1 is formatted in base 10 using power of 10 divisors, leading to the remaining "00001" string. This loop repeats until all divisors are used up.

```
Digits xor  edx, edx        EDX=0         // EAX=integer, EDX=remainder
skip   div  DivTbl[esi]     EAX=1 EDX=1   // EDX:EAX / 65535 -> 1:1 error!
       or   al, 30h         EAX=31 AF=0   // text for '1' - First digit wrong!
       mov  [edi], al                     // save character here
       mov  byte ptr [edi+1], 0           // Unicode - 0 here for 2-byte char
       add  edi, 2          EDI=12F9E4    // next char position
       or   edx, edx        PF=0          // see if any remainder
       jz   tag6                          // exit if no remainder
       mov  eax, edx        EAX=1         // move remiander to EAX
       sub  esi, 4          ESI=10        // move divisor table pointer
       ja   short Digits                  // and loop if table not empty

Digits xor  edx, edx        EDX=0 PF=1    // EAX=integer, EDX=remainder
skip   div  DivTbl[esi]     EAX=0 EDX=1   // EDX:EAX / 10000 = 0:1
       or   al, 30h         EAX=30        // text for '0'
       mov  [edi], al                     // save character here
       mov  byte ptr [edi+1], 0           // Unicode - 0 here for 2-byte char
       add  edi, 2          EDI=12F9E6    // next char position
       or   edx, edx                      // see if any remainder
       jz   tag6                          // exit if no remainder
       mov  eax, edx        EAX=1         // move remiander to EAX
       sub  esi, 4          ESI=C         // move divisor table pointer
       ja   short Digits                  // and loop if table not empty

Digits xor  edx, edx        EDX=0         // EAX=integer, EDX=remainder
skip   div  DivTbl[esi]     EAX=0 EDX=1   // EDX:EAX / 1000 = 0:1
       or   al, 30h         EAX=30        // text for '0'
       mov  [edi], al                     // save character here
       mov  byte ptr [edi+1], 0           // Unicode - 0 here for 2-byte char
       add  edi, 2          EDI=12F9E8    // next char position
       or   edx, edx        PF=0          // see if any remainder
       jz   tag6                          // exit if no remainder
       mov  eax, edx        EAX=1         // move remiander to EAX
       sub  esi, 4          ESI=8         // move divisor table pointer
       ja   short Digits                  // and loop if table not empty

Digits xor  edx, edx        EDX=0         // EAX=integer, EDX=remainder
skip   div  DivTbl[esi]     EAX=0 EDX=1   // EDX:EAX / 100 = 0:1
       or   al, 30h         EAX=30        // text for '0'
```

```
        mov   [edi], al                        // save character here
        mov   byte ptr [edi+1], 0              // Unicode - 0 here for 2-byte char
        add   edi, 2          EDI=12F9EA       // next char position
        or    edx, edx                         // see if any remainder
        jz    tag6                             // exit if no remainder
        mov   eax, edx        EAX=1            // move remiander to EAX
        sub   esi, 4          ESI=4            // move divisor table pointer
        ja    short Digits                     // and loop if table not empty

Digits  xor   edx, edx        EDX=0            // EAX=integer, EDX=remainder
skip    div   DivTbl[esi]     EAX=0 EDX=1      // EDX:EAX / 10 = 0:1
        or    al, 30h         EAX=30           // text for '0'
        mov   [edi], al                        // save character here
        mov   byte ptr [edi+1], 0              // Unicode - 0 here for 2-byte char
        add   edi, 2          EDI=12F9EC       // next char position
        or    edx, edx                         // see if any remainder
        jz    tag6                             // exit if no remainder
        mov   eax, edx        EAX=1            // move remiander to EAX
        sub   esi, 4          ESI=0            // move divisor table pointer
        ja    short Digits                     // table empty, jump not taken
```

### The final digit

The mistake has been made. We finish out the digits and return.

```
        test eax, 0FFh                         // final digit amount = 255?
        jz    tag6
        jmp   tag5
tag5    or    al, 30h         EAX=31           // text for '1'
        mov   [edi], al                        // save character here
        mov   byte ptr [edi+1], 0              // Unicode - 0 here for 2-byte char
        add   edi, 2          EDI=12F9EE       // next char position

tag6    mov   eax, ebx        EAX=0            // fractional part to EAX
        or    eax, ecx        ZF=1             // ???
        jnz   loc_30150AD8
        mov   word ptr [edi], 30h ; '0'        // write another '0' character
        pop   ecx             ECX=F ESP=12F498          // clean stack
        pop   ebx             EBX=12F9E2 ESP=12F49C     // clean stack, old EDI
        mov   edx, edi        EDX=12F9EE       //
        sub   edx, ebx        EDX=C            // number of bytes put onto EDI
        sar   edx, 1          EDX=6            // number of characters written

tag7    mov   eax, [ebx+1Eh]  EAX=40060000     //
        and   eax, 0FFFF0000h AF=0             //
        or    eax, edx        EAX=40060006     // info about string just made
        mov   [ebx+1Eh], eax                   // save into some field
        and   eax, 7FFFFFFFh                   // chop top?
        sub   eax, 40000000h  EAX=60006        // number of digits written?
        ror   eax, 10h                         // orient for return code
        retn                  ESP=12F4A0       // fini
// end – no more disassembly
```

And that demonstrated the bug in its entirety. To validate parts of this analysis, next I show to how this got in the code, and how the fix works.

## Previous Excel Versions

Here is a brief comparison with some previous Excel versions. Unfortunately I have been unable to test Excel 2003 to ensure it has the same code as 2002, but from the above analysis it seems likely since the routine was clearly rewritten from Excel 2002 to Excel 2007, and Excel 2003 doesn't have this bug.

### *Excel 2002 Formatting*

The Excel 2002 version I tested it is 10.6834.6830, SP3 from the Help/About menu item. The Excel.exe version is 10.0.6834.0 in explorer. The format function occurs in memory at `.text:30033733` and starts with the bytes `66 8B 46 06 66 8B D0 66 25 F0 7F`. The first instructions are:

```
mov     ax, [esi+6]
mov     dx, ax
```

From this first snippet it shows that the code here is from an old 16-bit version of the routine. For Excel 2007 32-bit registers are used throughout, resulting presumably in faster formatting. The structure of the routine is similar, but instead of using 32-bit registers in divisions Excel 2002 uses 16-bit registers.

Another interesting point is this routine is much simpler than the Excel 2007 one.

It is instructive to see the bug compared to the execution trace in this version. Recall the error happens in Excel 2007 in this section:

```
tag3   xor  ecx, ecx         ECX=0          // integer-1 to in EAX, fract in EBX
       mov  edx, 1           EDX=1          //
       inc  eax              EAX=10000      // carry from EBX into EAX=65536
       jz   skip                            // Excel 2002 jumps here since prev
                                            // instruction was 16-bit inc ax
                                            // Jumping here prints correctly?
       cmp  eax, DivTbl[esi+4]  PF=0        // check against max div value
       jb   loc_300664A3                    // EAX too big?
       jmp  tag4                            // jump to some fixup routine??
```

The reason I claim this is the bug is that the trace in Excel 2002 behaves slightly different, missing the table misalignment:

```
tag3   xor  ecx, ecx         ECX=0
       sar  esi, 1           ESI=8
       mov  dx, 1            EDX=F8000001
       inc  ax               EAX=0
       jz   skip    // Excel 2007 fails at this point due to 32 bit extension?
```

**Here is the crux of the whole analysis**: in Excel 2002, the AX register was incremented as a 16-bit value, and the rollover caused the jz (jump if zero) branch to be taken correctly. When this code was converted to 32-bit, this overflow, possible only when EAX = 65535 (as in all the bug cases) and the value was sufficiently near an integer. The fix from Microsoft, covered below, confirms this is indeed a bug.

The bug seemingly was introduced when converting the 16-bit formatting routine to a 32-bit one. It is easy to see that the conversion above, with the jump if zero that can only hit in very rare cases, would be easy to miss in the conversion. What is surprising is that an extremely detailed analysis (which was likely done by Microsoft engineers before accepting the code) did not catch this bug before shipping.

A complete trace of this run is in the Appendix.

### *Excel 2000 Formatting*

For Excel 2000, I ran the same steps on the file Excel.exe version 9.0.0.8924, SP-3. The corresponding formatting function starts at memory offset `.text:0x30182242` with the bytes `66 8B 46 06 66 8B  D0 66 25 F0 7F.` The first instructions again are:

```
mov     ax, [esi+6]
mov     dx, ax
```

The function is almost identical to the Excel2002, with minor changes having been made to Excel 2002. Again it uses a lot of 16-bit registers and operands.

## The Microsoft Hotfix

While I was writing this analysis, Microsoft released a 33MB hotfix for the rendering bug on Oct 10, 2007, at http://support.microsoft.com/kb/943075.

This knowledge base article states that the result of a calculation returning a value from 65534.99999999995 to 65535 is performed correctly, but the result is incorrectly shown as 100000. Also the result of the calculation returning a value from 65535.99999999995 to 65536 is also performed correctly, but incorrectly formats as 100001.

The fix works for values near 65536 by fixing the EAX overflow not being caught by 16-bit math like it was in the Excel 2002 version. Here is the unfixed Excel 2007 code from above:

```
tag3   xor  ecx, ecx        ECX=0         // integer-1 to in EAX, fract in EBX
       mov  edx, 1          EDX=1         //
       inc  eax             EAX=10000     // carry from EBX into EAX=65536
       jz   skip                          // Excel 2002 jumps here since prev
                                          // instruction was 16-bit inc ax
                                          // Jumping here prints correctly?
```

And here is the same code after the fix is applied:

```
tag3   xor  ecx, ecx        ECX=0         // Excel 2007 fix does new check:
       inc  eax             EAX=10000     // carry from EBX into EAX=65536
       cmp  eax, 0FFFFh                   // New check – avoids the
       jg   Digits                        // overflow causing table pointer
                                          // to be set wrong
```

The fix replaces the incorrect Excel 2007 code with an additional check on EAX, jumping as needed on the overflow. This fixes the incorrect formatting for the 6 values near 65536. The other 6 values are fixed by a similar check done at a slightly different place in the code. The missing EDX operation is moved to an appropriate spot in the code. Figure 10 shows the results of running the 12 values through the fixed version of Excel 2007.

For completeness, here is the hotfix file information. Excel.exe is now 17894936 bytes, and version number 12.0.6042.5000. The `Format` routine is at location `.text:3006402B`, and begins with the bytes `C7 47 1E 00 00 00 00 8B 46 04 8B D0 25`. The tables are identical, with the `Divisor` table at `.text:3010E400` and the rounding table at `.text:300640D7`.

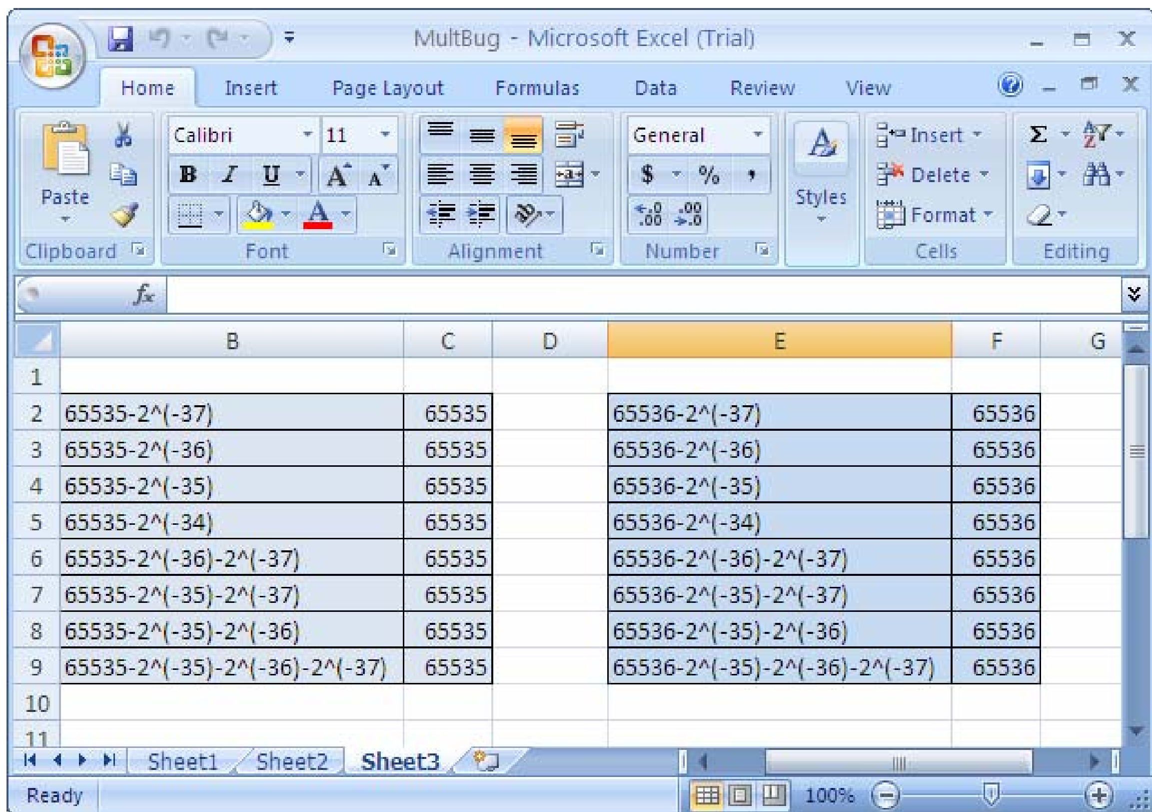



**Figure 10 - Fixed formatting**

Execution traces are on my website for analysis.

## Security Implications

Quite often poorly written routines contain holes exploitable by attackers. Since this routine is used in formatting, and specially constructed floating-point values cause an incorrectly formatted string, and is reachable from possible malformed Excel files, I attempted to find possible holes or exploits. I was unable to do so, but that does not mean

there are none. I think it is unlikely since it seems from my testing that the cases above are the only ones that format incorrectly, and they do not overflow the fixed length (?) string buffers used.

## Conclusion

This document covers the execution of the bug in depth, but does not claim to have covered the issue completely. It does validate Microsoft's claim as to the scope of the bug, sheds some light on how the bug came to be, and shows how to reproduce and examine the bug.

The basic blog rehash of the bug is well summarized by the Joel Spolsky admittance that he does not know what caused the bug, yet feels the need to blog about it, closing with [13]:

> "And let's face it -- do you really want the bright sparks who work there now, and manage to break lots of perfectly good working code -- rewriting the core calculating engine in Excel? Better keep them busy adding and removing dancing paper clips all day long."

It would be much more useful and interesting if he actually figured out what was going on instead of taking easy potshots about the subject. I hope this dispels some speculation and uninformed critique. Another example: an amazing number of people guessed the bug had something to do with the 65536 row limit, showing the flaws in belief in numerology. It was these types of unfounded statements that originally led me to consider finding the real answer to the bug.

If any reader thinks they can write a high performance IEEE 754 formatting routine from scratch (without using library calls), I'd like the see the result and proof of correctness. It will be hard to do it well.

More data is available at www.lomont.org, including large images of the entire function graph (2K x 5K pixels) for the offending function in Excel 2007 and the correct one in Excel 2002. There are the numerous function traces and associated data tables, and the C++ code from this article.

A final comment to lawyers confused about fair-use issues, reverse-engineering for interoperability, First Amendment and DMCA issues, and the like. Be sure to do your homework before you contact me. It will save us both time and one of us embarrassment.

One last thing that would make this complete – an analysis of the Excel 2003 formatting code, which I suspect it very similar to the Excel 2002 version. I predict a rewrite from Excel 2003 to Excel 2007 introduced this bug. With the information presented here it should be easy for someone to do the check.

## Links and references

[1] http://blogs.msdn.com/excel/archive/2007/09/25/calculation-issue-update.aspx
[2] http://it.slashdot.org/it/07/09/24/2339203.shtml
[3] http://digg.com/microsoft/Critical_Excel_2007_bug_cripples_users
[4] http://www.news.com/8301-13580_3-9785728-39.html
[5] http://www.joelonsoftware.com/items/2007/09/26b.html
[6] http://en.wikipedia.org/wiki/IEEE_floating-point_standard
[7] http://groups.google.com/group/microsoft.public.excel/browse_thread/thread/2bcad1a1a4861879/2f8806d5400dfe22?hl=en#2f8806d5400dfe22
[8] http://support.microsoft.com/kb/943075
[9] http://babbage.cs.qc.edu/IEEE-754/Decimal.html
[10] Chris Lomont, Fast Inverse Square Root, 2003, http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf
[11] Chris Lomont, Floating Point Tricks, Games Programming Gems 6, 2006, ISBN 1-58450-450-1
[12] http://en.wikipedia.org/wiki/IEEE_floating-point_standard
[13] http://www.joelonsoftware.com/items/2007/09/26b.html

## Appendix

### *C++ Code*

```
// code to investigate the Excel 2007 bug
// Chris Lomont 2007
// put =850*77.1 in a cell, text shows 100,000, correct is 65535
#include <iostream>
#include <cmath>

using namespace std;

#define Dump(a) DumpVal(#a,a)

void DumpVal(char * text, double v)
	{
	unsigned char * byte = reinterpret_cast<unsigned char*>(&v);
	cout << hex << "0x";
	for (int a = sizeof(double)-1; a >= 0; --a)
		{
		int val = byte[a];
		if (val < 10) cout << '0';
		cout << val << ' ';
		}
	cout << dec;
	cout << " = \t" << v << "\t = " << text;
	cout << endl;
	} // DumpVal

int main(void)
	{
	// some negative powers
	double e35 = pow(2.0,-35.0);
	double e36 = pow(2.0,-36.0);
	double e37 = pow(2.0,-37.0);
	double e38 = pow(2.0,-38.0);

	Dump(65535);
	Dump(850*77.1);
	Dump(850*77.1+1);
	Dump(850*77.1+0.5);
```

```
        Dump(65536);

        // show this too small, does not effect outcomes
        Dump(65535-e38);

        // all combinations of e35 to e38
        Dump(65535        -e37);
        Dump(65535    -e36    );
        Dump(65535    -e36-e37);
        Dump(65535-e35        );
        Dump(65535-e35    -e37);
        Dump(65535-e35-e36    );
        Dump(65535-e35-e36-e37);

        Dump(65536        -e37);
        Dump(65536    -e36    );
        Dump(65536    -e36-e37);
        Dump(65536-e35        );
        Dump(65536-e35    -e37);
        Dump(65536-e35-e36    );
        Dump(65536-e35-e36-e37);


        return 0;
        }
// end - ExcelBug.cpp
```

## *Excel 2002 Trace*

```
// Excel 2002 trace for formatting 850*77.1+1 = 65536-2^(-37)

// divisor table
// table starts at word_3005FE00, and 16 bits per entry
//.text:3005FE00  01 00 0A 00 64 00 E8 03  10 27 FF FF 07 24 00 00
byte DivTbl[] = {01 00 0A 00 64 00 E8 03  10 27 FF FF};

// ECX rounding table
//.text:3005FE10  49 68 01 00 E1 12 0E 00  CC BC 8C 00 F8 5F 7F 05
//.text:3005FE20  B3 BF F9 36 00 00 00 00  00 00 00 00 00 00 00 00
//.text:3005FE30  00 00 00 00 00 00 00 A0  02 40 04 00 00 00 00 00
byte RndTbl[] =
    {
0x49, 0x68, 0x01, 0x00, // digits = 1
0xE1, 0x12, 0x0E, 0x00, // digits = 2
0xCC, 0xBC, 0x8C, 0x00, // digits = 3
0xF8, 0x5F, 0x7F, 0x05, // digits = 4
0xB3, 0xBF, 0xF9, 0x36  // digits = 5 - this value=0x36F9BFB3
// above seems to be end of table based on Excel 2002 version of table
    };

// execution trace

ST0= 7.1510092812520145735e-3921 ST1=-1.8318425593494863852e583
ST2= 6.4477652123115469025e-4051 ST3= 7.2305206786858568894e2850
ST4= 0.0 ST5= 0.0 ST6= 1.0 ST7= 1.0
CTRL=137F CS=1B DS=23 ES=23 FS=3B GS=0 SS=23
EAX=8 EBX=FFEC0C02 ECX=F EDX=3 ESI=30803E10 EDI=13F3FE EBP=13FC00
ESP=13F348 EFL=212

Format mov   ax, [esi+6]    EAX=40EF      // result value - assume 0
       mov   dx, ax         EDX=40EF      //
       and   ax, 7FF0h      EAX=40E0      //
       jz    loc_30175BFF                 //
       shr   ax, 4          EAX=40E       //
       sub   ax, 3FEh       EAX=10        //
```

```
        mov   bx, ax         EBX=FFEC0010  //
        cwde                               //
        imul  eax, 9A21h     EAX=9A210     //
        sar   eax, 11h       EAX=4         //
        add   ax, 4001h      EAX=4005      //
        and   dx, 8000h      EDX=0         //
        or    dx, ax         EDX=4005      //
        mov   [edi+20h], dx                //
        sub   ax, 4000h      EAX=5         //
        or    bx, bx                       //
        jl    loc_30180554                 //
        cmp   bx, 10h                      //
        jle   short Bit16                  //
Bit16   push  edi            ESP=13F344    //
        push  ecx            ESP=13F340    //
        mov   edx, [esi]     EDX=FFFFFFFF  //
        mov   esi, [esi+4]   ESI=40EFFFFF  //
        xchg  eax, esi       EAX=40EFFFFF ESI=5 //
        and   eax, 0FFFFFh   EAX=FFFFF     //
        or    eax, 100000h   EAX=1FFFFF    //
        mov   cl, 15h        ECX=15        //
        sub   cl, bl         ECX=5         //
        mov   ebx, eax       EBX=1FFFFF    //
        shr   eax, cl        EAX=FFFF      //
        neg   cl             ECX=FB        //
        shld  ebx, edx, cl   EBX=FFFFFFFF  //
        shl   edx, cl        EDX=F8000000  //
        mov   ecx, edx       ECX=F8000000  //
        dec   si             ESI=4         //
        shl   si, 1          ESI=8         //
        cmp   ax, DivTbl[esi]              //
        jnb   short tag2                   //
tag2    jecxz short loc_300337C0           //
        jmp   tag2a                        //
tag2a   shl   esi, 1         ESI=10        //
        add   ecx, ds:dword_3005FE10[esi] ECX=2EF9BFB3
        adc   ebx, 0         EBX=0         //
        jb    tag3                         //
tag3    xor   ecx, ecx       ECX=0         //
        sar   esi, 1         ESI=8         //
        mov   dx, 1          EDX=F8000001  //
        inc   ax             EAX=0         //
        jz    skip                         // Excel  2007 fails at this point
                                           // due to 32 bit extension?
skip    div   DivTbl[esi]    EAX=6 EDX=F80015A0 // divide by 10000
        xor   ah, ah                       // zero byte  in top
        or    al, 30h        EAX=36        // output '6'
        stosw                EDI=13F400    // write 2 byte unicode value
        or    dx, dx                       // test for 0 remainder
        jz    short tag6                   // bail if done
        mov   ax, dx         EAX=15A0      // remainder
        sub   esi, 2         ESI=6         // new 16-bit table value
        ja    short Digits                 // do another digit
                                           //
Digits  xor   dx, dx         EDX=F8000000  //
skip    div   DivTbl[esi]    EAX=5 EDX=F8000218 // divide by 1000
        xor   ah, ah                       // zero byte  in top
        or    al, 30h        EAX=35        // output '5'
        stosw                EDI=13F402    // write 2 byte unicode value
        or    dx, dx                       // test for 0 remainder
        jz    short tag6                   // bail if done
        mov   ax, dx         EAX=218       // remainder
        sub   esi, 2         ESI=4         // new 16-bit table value
        ja    short Digits                 // do another digit
                                           //
Digits  xor   dx, dx         EDX=F8000000  //
skip    div   DivTbl[esi]    EAX=5 EDX=F8000024 // divide by 100
        xor   ah, ah                       // zero byte  in top
        or    al, 30h        EAX=35        // output '5'
```

```
       stosw              EDI=13F404    // write 2 byte unicode value
       or     dx, dx                    // test for 0 remainder
       jz     short tag6                // bail if done
       mov    ax, dx      EAX=24        // remainder
       sub    esi, 2      ESI=2         // new 16-bit table value
       ja     short Digits              // do another digit
                                        //
Digits xor    dx, dx      EDX=F8000000  //
skip   div    DivTbl[esi] EAX=3 EDX=F8000006 // divide by 10
       xor    ah, ah                    // zero byte  in top
       or     al, 30h     EAX=33        // output '3'
       stosw              EDI=13F406    // write 2 byte unicode value
       or     dx, dx                    // test for 0 remainder
       jz     short tag6                // bail if done
       mov    ax, dx      EAX=6         // remainder
       sub    esi, 2      ESI=0         // out of table values
       ja     short Digits              // no jump

           or     al, al                    // test 0 remainder
       jz     short tag6                //
       jmp    tag5                      //
tag5   xor    ah, ah                    //
       or     al, 30h     EAX=36        // one last '6' digit left to do
       stosw              EDI=13F408    //
       jmp    tag6                      //
tag6   mov    eax, ebx    EAX=0         // cleanup, return values...
       or     eax, ecx                  //
       jnz    loc_3017BC33              //
       mov    word ptr [edi], 30h       //
       pop    ecx         ECX=F ESP=13F344
       pop    ebx         EBX=13F3FE ESP=13F348
       mov    edx, edi    EDX=13F408    //
       sub    edx, ebx    EDX=A         //
       sar    edx, 1      EDX=5         //
       mov    eax, edx    EAX=5         //
       shl    eax, 10h    EAX=50000     //
       mov    [ebx+1Eh], dl             //
       mov    ax, [ebx+20h] EAX=54005   //
       and    eax, 0FFFF7FFFh           //
       sub    ax, 4000h   EAX=50005     //
       retn               ESP=13F34C    //
// end - 2002 trace
```

## Call graph image for Excel 2007

Figure 11 is a cleaned up call graph for the unpatched Excel 2007 formatting code. A high resolution version (2500x5000 pixel PNG) is available from my website www.lomont.org.

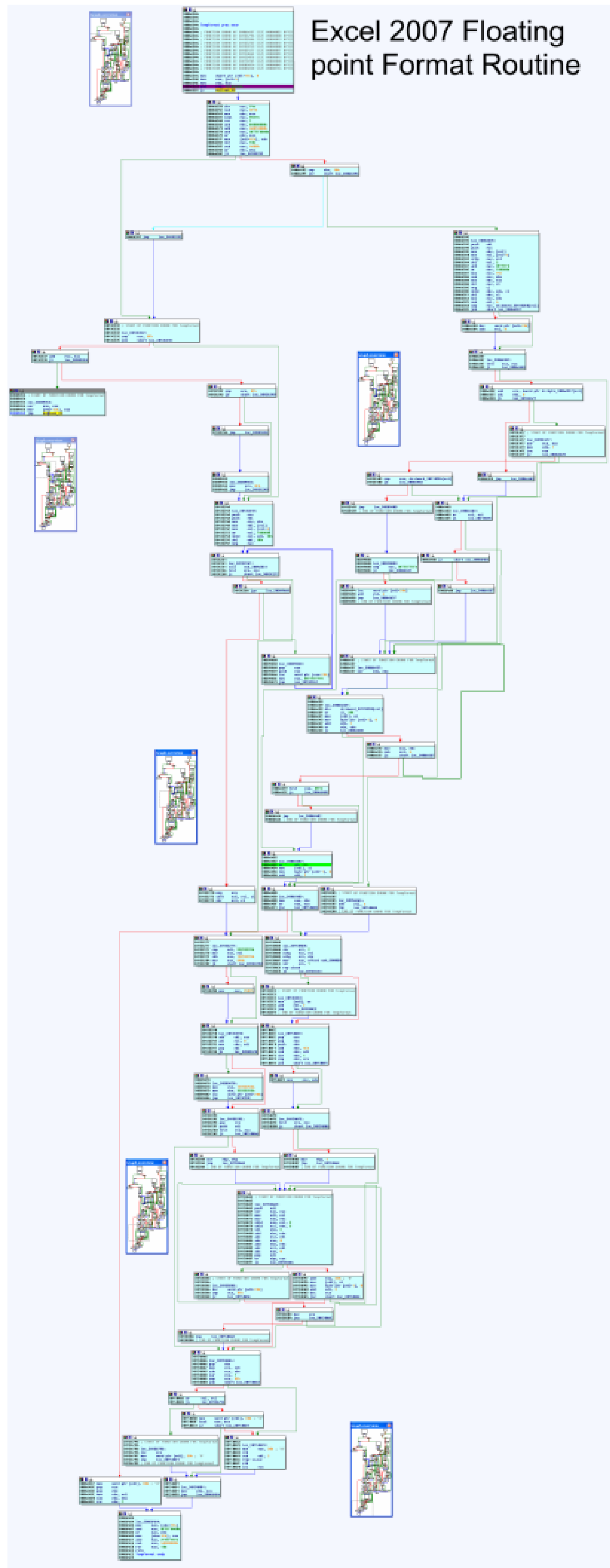



**Figure 11 – Excel 2007 unpatched call graph**