CVE-2012-0158

笔记本： cve
创建时间： 2019/4/8 18:02 **更新时间：** 2019/4/9 17:58

CVE-2012-0158

> 前言: 在看这个之前，笔者看了很多网上的调试方法，当然无外乎是找到覆盖的返回地址，这里有两种方法: 1. 栈回溯 2. 通过一些牛逼的技巧(通过shellcode的常见跳jmp esp，或者通过rtf文件读取必然调用OpenStream函数，再者就是通过一些漏洞经验，看有没有memcpy这种无验证的危险操作等)

鉴于笔者的水平，所以采取最简单的栈回溯的方法进行分析
poc在附件中给出，为启动calc的一个poc。

关于rtf的一些基础知识

关于这一部分主要是参考
https://www.anquanke.com/post/id/91643#h2-1文档
静态分析部分，下边会详细复现
当然想从原理上了解rtf文件，还是比较复杂的，虽然rtf格式简单，但是包含的内容很复杂，想要全部看一遍，不如碰到什么问题在分析比较好。

几篇比较详细的rtf文件格式的文章，当然最权威的还是最后的官方文档
https://blog.csdn.net/richerg85/article/details/12056783
https://blog.csdn.net/dream_dt/article/details/79215798
http://www.biblioscape.com/rtf15_spec.htm
https://www.microsoft.com/downloads/details.aspx?familyid=DD422B8D-FF06-4207-B476-6B5396A18A2B&displaylang=en

介绍一下从doc中提取rtf对象，并解析的步骤

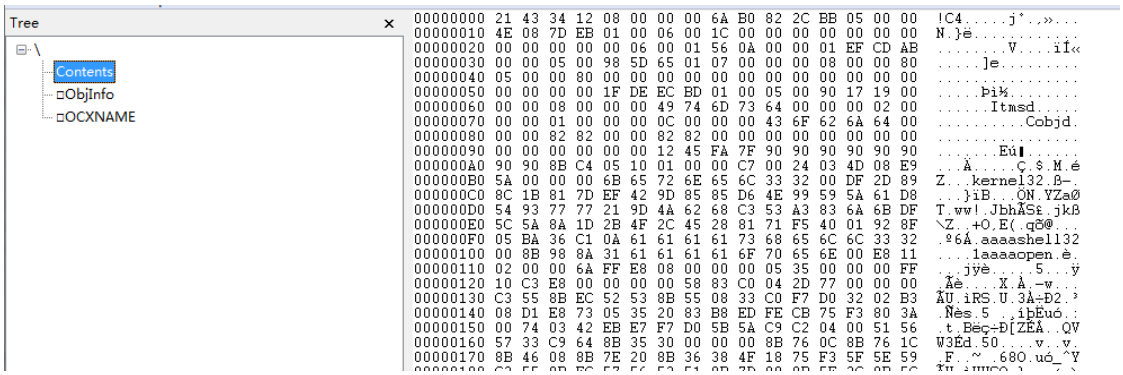
从doc中提取rtf对象

oletool有一个rtfobj.py工具，提取rtf对象

```
File: 'E:\\cve\\cve-2012-0158\\sample.rtf' - size: 136606 bytes
---+----------+--------------------------------------------------------------
id |index     |OLE Object
---+----------+--------------------------------------------------------------
0  |000000A5h |format_id: 2 (Embedded)
   |          |class name: b'MSComctlLib.ListViewCtrl.2'
   |          |data size: 3584
   |          |CLSID: BDD1F04B-858B-11D1-B16A-00C0F0283628
   |          |MSCOMCTL.ListViewCtrl (may trigger CVE-2012-0158)
---+----------+--------------------------------------------------------------
Saving file embedded in OLE object #0:
  format_id  = 2
  class name = b'MSComctlLib.ListViewCtrl.2'
  data size  = 3584
  saving to file E:\cve\cve-2012-0158\sample.rtf_object_000000A5.bin
```

下载地址: https://github.com/decalage2/oletools

查看rtf对象

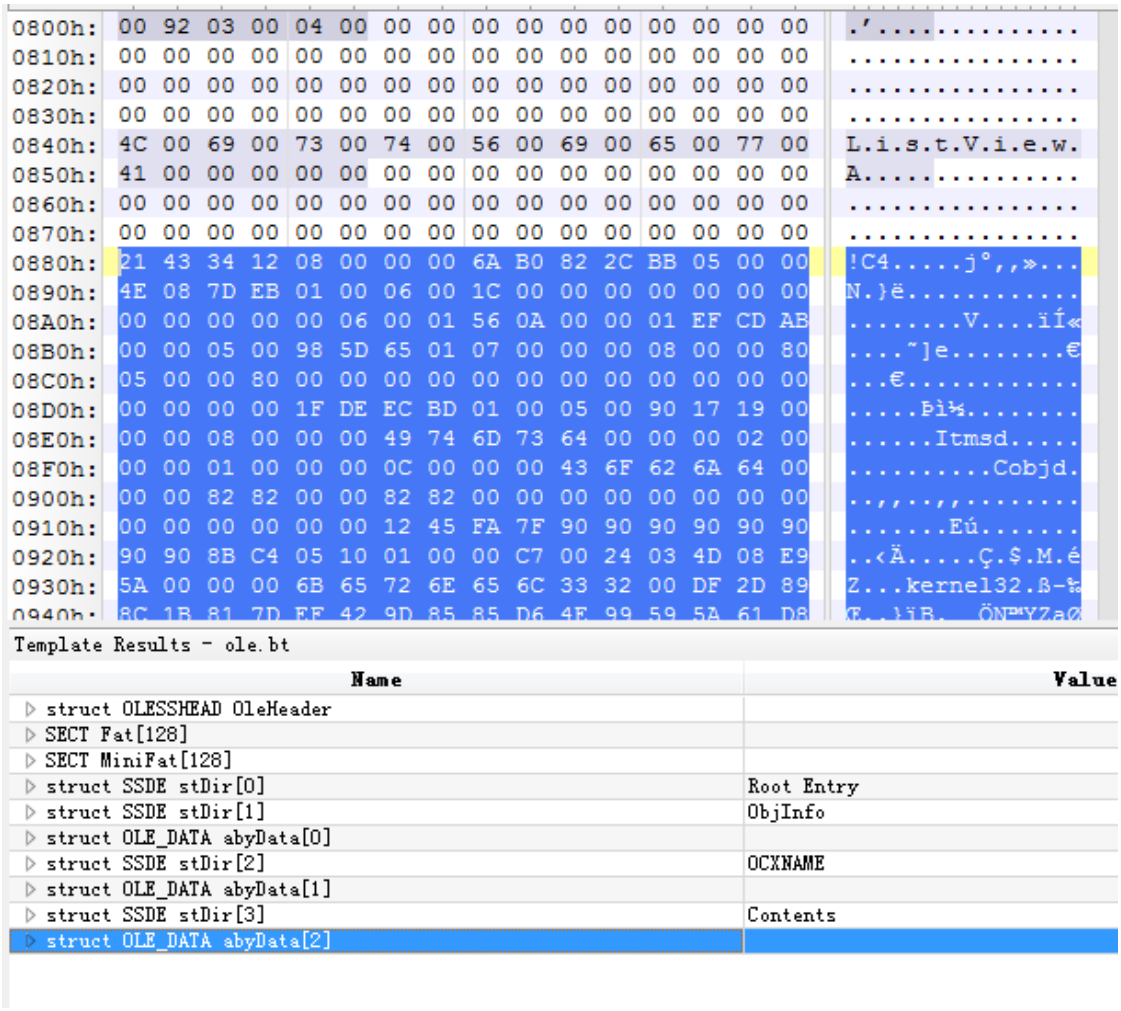


该对象存在三个steam

下载地址: https://0cch.com/2015/08/03/ole-file-view-tool/

010查看

模板



下载地址: https://blogs.technet.microsoft.com/srd/2008/08/12/ms08-042-understanding-and-detecting-a-specific-word-vulnerability/

以上基本就了解了这个该文件的结构，然后接下来动态调试

其他

https://www.freebuf.com/column/166532.html

调试

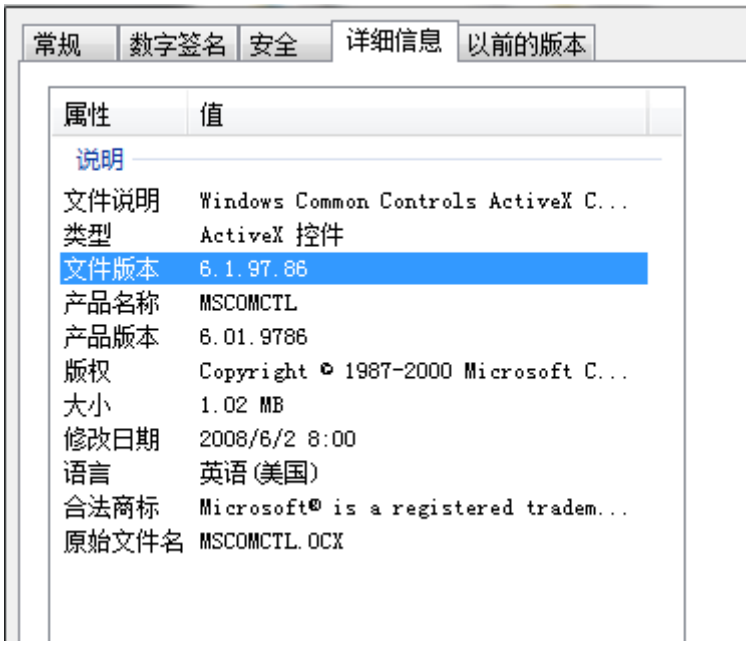
下断点WinExec，查看栈

发现栈信息有

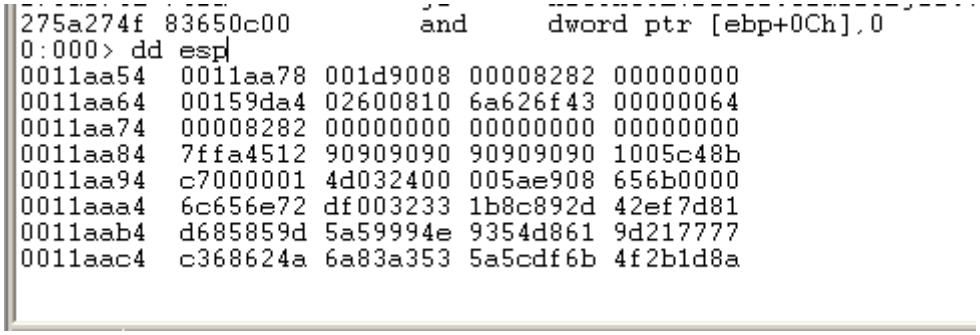
MSCOMCTL!DllGetClassObject+0xb456

此时找到该主机MSCOMCTL.ocx，并拖入IDA中

该文件信息



重启，在该处下断查看栈环境，发现栈环境已经变化



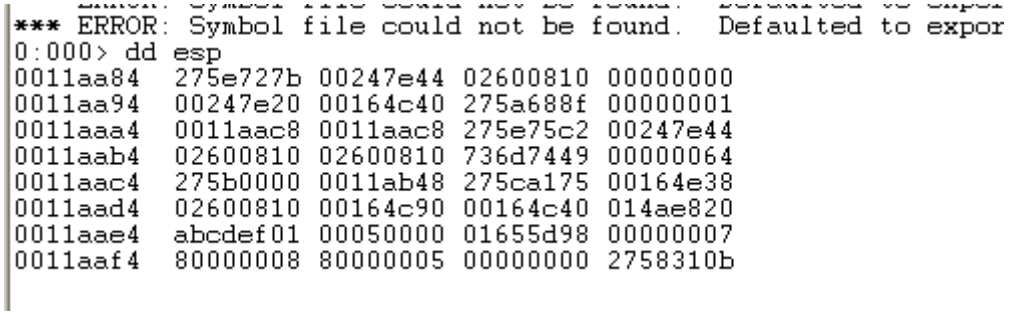
然后回溯到函数起始位置

之前笔者直接根据地址下断点即 0x275a26fa下断，但是启动之后一直断，所以就采用一个比较笨的方法，因为看到函数其实位置和之前有符号信息的断点不是很远，索性就采取数字节的方法，最后下断为

MSCOMCTL!DllGetClassObject+0xb413

断下来之后查看函数返回地址

返回值为0x275e727b



查看改地址发现该返回地址是正确的

```
0:000> u 275e727b-8
MSCOMCTL!DllGetDocumentation+0xce9:
275e7273 57              push    edi
275e7274 53              push    ebx
275e7275 56              push    esi
275e7276 e87fb4fbff      call    MSCOMCTL!DllGetClassObject+0xb413 (275a26fa)
275e727b 85c0            test    eax,eax
275e727d 7c27            jl      MSCOMCTL!DllGetDocumentation+0xd1c (275e72a6)
275e727f 6a08            push    8
275e7281 8d45f4          lea     eax,[ebp-0Ch]
```

所以断定此时的程序栈还没有被破坏，单步运行，查看到底是哪个函数对返回值覆盖

查看在MSCOMCTL!DllGetClassObject+0xb451处的地址，对照IDA查看其参数

```
  if ( v5 == 'jboC' && dwBytes >= 8 )
  {
    v4 = sub_275A24A0((int)&v7, v2, dwBytes);
    if ( v4 >= 0 )
```

```
0:000> p
eax=0011aa78 ebx=02600810 ecx=7c93003d edx=01370001 esi=00247e44 edi=00000000
eip=275a2738 esp=0011aa54 ebp=0011aa80 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
MSCOMCTL!DllGetClassObject+0xb451:
275a2738 e863fdffff      call    MSCOMCTL!DllGetClassObject+0xb1b9 (275a24a0)
0:000> dd esp
0011aa54  0011aa78 02600810 00008282 00000000
0011aa64  00247e44 02600810 6a626f43 00000064
0011aa74  00008282 00247f30 275876c7 0011aaa8
0011aa84  275e727b 00247e44 02600810 00000000
0011aa94  00247e20 00164c40 275a688f 00000001
0011aaa4  0011aac8 0011aac8 275e75c2 00247e44
0011aab4  02600810 02600810 736d7449 00000064
0011aac4  275b0000 0011ab48 275ca175 00164e38
```

再次单步，可以明显看到返回地址被覆盖，而且之后的90909090明显是shellcode的滑行代码，所以可以确定覆盖其返回地址的函数就是它了

```
0:000> p
eax=00000000 ebx=02600810 ecx=7c93003d edx=00150608 esi=00247e44 edi=00000000
eip=275a273d esp=0011aa54 ebp=0011aa80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSCOMCTL!DllGetClassObject+0xb456:
275a273d 8bf0            mov     esi,eax
0:000> dd esp
0011aa54  0011aa78 001d9008 00008282 00000000
0011aa64  00247e44 02600810 6a626f43 00000064
0011aa74  00008282 00000000 00000000 00000000
0011aa84  7ffa4512 90909090 90909090 1005c48b
0011aa94  c7000001 4d032400 005ae908 656b0000
0011aaa4  6c656e72 df003233 1b8c892d 42ef7d81
0011aab4  d685859d 5a59994e 9354d861 9d217777
0011aac4  c368624a 6a83a353 5a5cdf6b 4f2b1d8a
```

参照IDA，进入函数，一个很明显的拷贝操作

```
19      {
20        v6 = (*(int (__stdcall **)(LPVOID, LPVOID, SIZE_T, _DWORD))(*(_DWORD *)v3 + 12))(v3, v
21        if ( v6 >= 0 )
22        {
23          qmemcpy((void *)a1, lpMem, dwBytes);
24          v6 = (*(int (__stdcall **)(LPVOID, void *, SIZE_T, _DWORD))(*(_DWORD *)v3 + 12))(
25                v3,
26                &unk_27633FE0,
27                ((dwBytes + 3) & 0xFFFFFFFC) - dwBytes,
28                0);
29        }
```

至此就发现其覆盖地址，但是该漏洞的源头并不在这里。

在进入该函数前我们看到参数，即拷贝的大小为8282过大，导致之后的拷贝覆盖

```
0:000> dd esp
0011aa54  0011aa78 001d9008 00008282 00000000
0011aa64  00247e44 02600810 6a626f43 00000064
0011aa74  00008282 00000000 00000000 00000000
0011aa84  7ffa4512 90909090 90909090 1005c48b
0011aa94  c7000001 4d032400 005ae908 656b0000
```

理论上说，引起改代码处赋值错误应该是未检验或者通过某些骚操作进而让该值通过了检验，这一切通过源码看到，实际上，该错误并不是想的那样，而是代码写错了！！！
查看源码，这里可以看到，

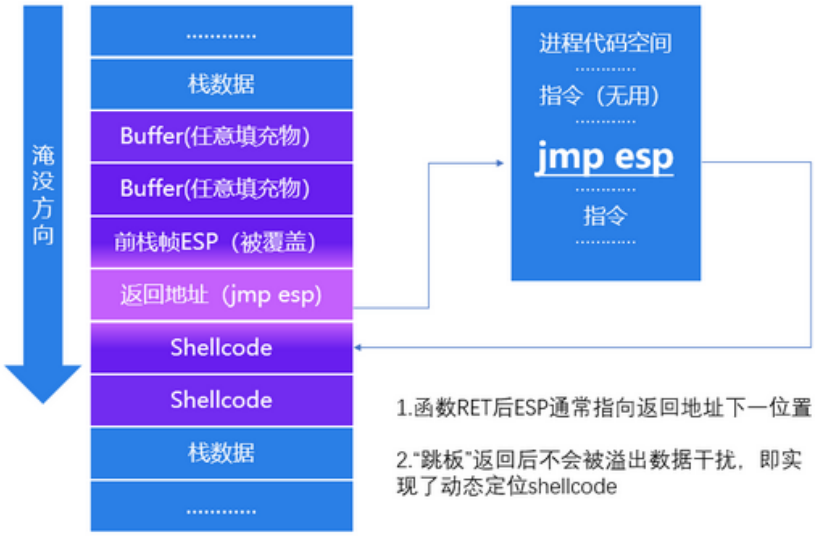
```
{
  if ( v5 == 'jboC' && dwBytes >= 8 )
  {
    v4 = sub_275A24A0((int)&v7, v2, dwBytes);
    if ( v4 >= 0 )
    {
```

，在读取一个叫Cobj的对象时，其大小进行了检验，但是其内容大小的检验是大于等于8？？！！，正常的文本对象大小为8，只要找到该大小，根本不需要什么精心构造，大小可以定义很大，也可以能通过检测，而且，代码下边并没有对边界进行其他检验，所以后边紧接着函数的拷贝操作，溢出操作就顺理成章了，其利用简单明了也难怪很多组织会用该漏洞。网上也有人说这个是微软特意留下的后门漏洞

其他

笔者这里使用了最low的栈回溯，但是其他技巧对以后调试也是很有帮助的，这里笔者也顺手调试以便之后调试学习

- 利用ShellCode的特征
  对于了解写ShellCode的大小牛而言，最常用的就是jmp esp的跳转到shellcode，在ShellCode中笔者是专门写过一篇笔记进行复习，主要是参考的Q版缓冲区溢出教程



图：“跳板”指令溢出利用流程

所以直接在文档中搜索90909090或者跳板指令的固定地址
搜索固定地址:

1. windbg搜索字符串 s -w 0x70000000 L0x7fffff e4ff
2. 由于程序都会加载 kernel32.dll，还可通过windbg 下的Mona.py插件，使用 !py mona jmp -r esp -m “kernel32.dll” 查找
3. 使用OD插件FindAddr也可以查找
4. 推荐使用一些前人找到的通用的地址

   搜索得到

```
0900h: 00 00 82 82 00 00 82 82 00 00 00 00 00 00 00 00  ..,,..,,....
0910h: 00 00 00 00 00 00 12 45 FA 7F 90 90 90 90 90 90  .......Eú.....
0920h: 90 90 8B C4 05 10 01 00 00 C7 00 24 03 4D 08 E9  ..‹Ä.....Ç.$.M.é
0930h: 5A 00 00 00 6B 65 72 6E 65 6C 33 32 00 DF 2D 89  Z...kernel32.ß-‰
```

   随便找一个程序，搜索7FFA4512地址

```
7FFA4512 - FFE4          jmp esp
7FFA4514   00E5          add ch,ah
7FFA4516   01E5          add ebp,esp
```

   就是一个跳板地址 jmp esp

   之后在这里下断

   ba e1 7FFA4512 "r eip,esp;"

   断下来查看断点

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=
7ffa4512 ffe4            jmp     esp {0011aa90}
0:000> u
7ffa4512 ffe4            jmp     esp
7ffa4514 00e5            add     ch,ah
7ffa4516 01e5            add     ebp,esp
7ffa4518 02e5            add     ah,ch
7ffa451a 03e5            add     esp,ebp
7ffa451c 04e5            add     al,0E5h
```

   此时说明该 0011aa90 被覆盖

   重新运行，对该地址下写断点查看最后几次写入，即可发现覆盖地址，之后操作一致

```
eip=275a2525
eip=7c92e8ab
eip=7c92e8dc
eip=275a6810
eip=275a68a3
eip=275a694e
eip=275a69b6
eip=275a69c3
eip=275a69cc
eip=275e7274
eip=275a24fe
ModLoad: 76d70000 76d92000   C:\WINDOWS\system32\Apphelp.dll
eax=77c0b8c1 ebx=00000000 ecx=00392f50 edx=77c31ae8 esi=7c92de50 edi=fff
```

其中还有一些思路，不过不知道是不是笔者的水平问题问题还是水平问题，并不能找到文章所说的API，其中也尝试一些其他API，但是并不能达到文章效果，还有Windbg调试确实很厉害。