Windows zero-day exploit used in targeted attacks by FruityArmor APT

By Anton Ivanov

Published: 2016-10-20 · Archived: 2026-04-05 14:31:39 UTC

A few days ago, Microsoft published the "critical" MS16-120 security bulletin with fixes for vulnerabilities in Microsoft Windows, Microsoft Office, Skype for Business, Silverlight and Microsoft Lync.

One of the vulnerabilities – CVE-2016-3393 – was reported to Microsoft by Kaspersky Lab in September 2016.

MS16-120	Win32k Elevation of Privilege Vulnerability	CVE-2016-3270	pgboy, zhong_sf of Qihoo 360 Vulcan Team
MS16-120	Windows Graphics Component RCE Vulnerability	CVE-2016-3393	Anton Ivanov of Kaspersky Lab
MS16-120	True Type Font Parsing Elevation of Privilege Vulnerability	CVE-2016-7182	Mateusz Jurczyk of Google Project Zero
MS16-119	Microsoft Browser Information Disclosure Vulnerability	CVE-2016-3267	Wenxiang Qian of Tencent QQBrowser

Here's a bit of background on how this zero-day was discovered. A few of months ago, we deployed a new set of technologies in our products to identify and block zero-day attacks. These technologies proved their effectiveness earlier this year, when we discovered two Adobe Flash zero-day exploits – CVE-2016-1010 and CVE-2016-4171. Two Windows EoP exploits have also been found with the help of this technology. One is CVE-2016-0165. The other is CVE-2016-3393.

Like most zero-day exploits found in the wild today, CVE-2016-3393 is used by an APT group we call **FruityArmor**. FruityArmor is perhaps a bit unusual due to the fact that it leverages an attack platform that is built entirely around PowerShell. The group's primary malware implant is written in PowerShell and all commands from the operators are also sent in the form of PowerShell scripts.

In this report we describe the vulnerability that was used by this group to elevate privileges on a victim's machine. Please keep in mind that we will not be publishing all the details about this vulnerability because of the risk that other threat actors may use them in their attacks.

Attack chain description

To achieve remote code execution on a victim's machine, FruityArmor normally relies on a browser exploit. Since many modern browsers are built around sandboxes, a single exploit is generally not sufficient to allow full access to a targeted machine. Most of the recent attacks we've seen that rely on a browser exploit are combined with an EoP exploit, which allows for a reliable sandbox escape.

In the case of FruityArmor, the initial browser exploitation is always followed by an EoP exploit. This comes in the form of a module, which runs directly in memory. The main goal of this module is to unpack a specially crafted TTF font containing the CVE-2016-3393 exploit. After unpacking, the module directly loads the code exploit from memory with the help of AddFontMemResourceEx. After successfully leveraging CVE-2016-3393, a second stage payload is executed with higher privileges to execute PowerShell with a meterpreter-style script that connects to the C&C.

EOP zero-day details

The vulnerability is located in the **cjComputeGLYPHSET_MSFT_GENERAL** function from the Win32k.sys system module. This function parses the cmap table and fills internal structures. The CMAP structure looks like this:

Type	*Name*
USHORT	format
USHORT	length
USHORT	language
USHORT	segCountX2
USHORT	searchRange
USHORT	entrySelector
USHORT	rangeShift
USHORT	endCount[segCount]
USHORT	reservedPad
USHORT	startCount[segCount]
SHORT	idDelta[segCount]
USHORT	idRangeOffset[segCount]
USHORT	glyphIdArray[ ]

The most interesting parts of this structure are two arrays – endCount and startCount. The exploit contains the next cmap table with segments:

```
Length:     48
Version:    0
segCount:   28   (X2 = 56)
searchRange:    32
entrySelector:  4
rangeShift: 24
Seg   1 : St = 0000, En = 1388, D =      0, RO =     1, gId# = -28
Seg   2 : St = 1388, En = 1770, D =      0, RO =     1, gId# = -27
Seg   3 : St = 1770, En = 1B58, D =      0, RO =     1, gId# = -26
Seg   4 : St = 1B58, En = 1F40, D =      0, RO =     1, gId# = -25
Seg   5 : St = 1F40, En = 2328, D =      0, RO =     1, gId# = -24
Seg   6 : St = 2328, En = 238C, D =      0, RO =     1, gId# = -23
Seg   7 : St = 238C, En = 23F0, D =      0, RO =     1, gId# = -22
Seg   8 : St = 23F0, En = 2454, D =      0, RO =     1, gId# = -21
Seg   9 : St = 2454, En = 24B8, D =      0, RO =     1, gId# = -20
Seg  10 : St = 24B8, En = 251C, D =      0, RO =     1, gId# = -19
Seg  11 : St = 251C, En = 2580, D =      0, RO =     1, gId# = -18
Seg  12 : St = 2580, En = 25E4, D =      0, RO =     1, gId# = -17
Seg  13 : St = 25E4, En = 2648, D =      0, RO =     1, gId# = -16
Seg  14 : St = 2648, En = 2710, D =      0, RO =     1, gId# = -15
Seg  15 : St = 2710, En = 3A98, D =      0, RO =     1, gId# = -14
Seg  16 : St = 3A98, En = 4E20, D =      0, RO =     1, gId# = -13
Seg  17 : St = 4E20, En = 61A8, D =      0, RO =     1, gId# = -12
Seg  18 : St = 61A8, En = 7530, D =      0, RO =     1, gId# = -11
Seg  19 : St = 7530, En = 88B8, D =      0, RO =     1, gId# = -10
Seg  20 : St = 88B8, En = 9C40, D =      0, RO =     1, gId# = -9
Seg  21 : St = 9C40, En = AFC8, D =      0, RO =     1, gId# = -8
Seg  22 : St = AFC8, En = C350, D =      0, RO =     1, gId# = -7
Seg  23 : St = C350, En = D6D8, D =      0, RO =     1, gId# = -6
Seg  24 : St = D6D8, En = EA60, D =      0, RO =     1, gId# = -5
Seg  25 : St = EA60, En = EE48, D =      0, RO =     1, gId# = -4
Seg  26 : St = EE48, En = F230, D =      0, RO =     1, gId# = -3
Seg  27 : St = F231, En = FFFE, D =      0, RO =     1, gId# = -2
Seg  28 : St = FFFF,     = FFFF, D =      0, RO =     1, gId# = -1
```

To compute how much memory to allocate to internal structures, the function executes this code:

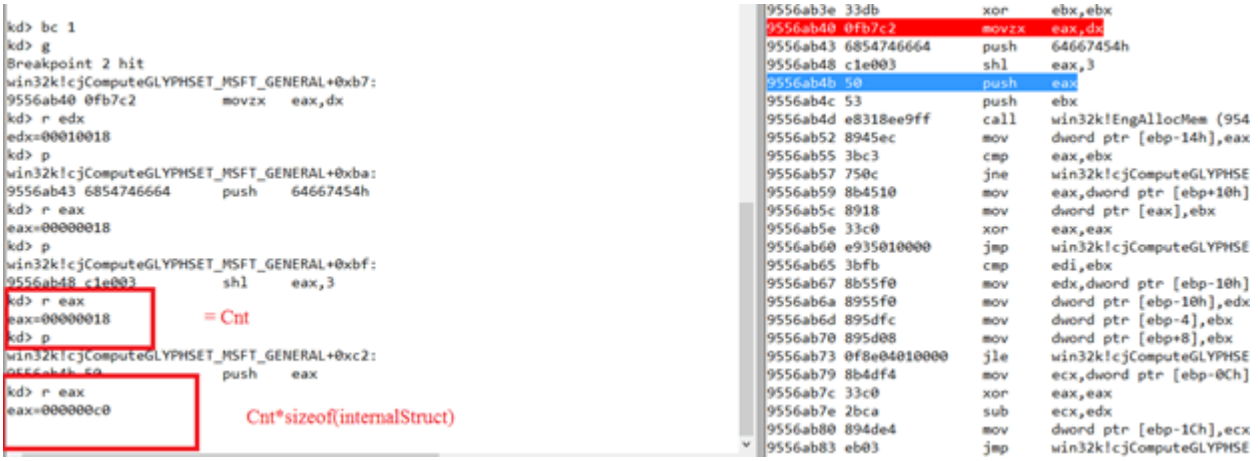
```
USHORT  cnt=0;
for (int i=0; i<cnt;i++)
        cnt += (Seg[i].end - Seg[i].start)+1
```

After computing this number, the function allocates memory for structures in the following way:

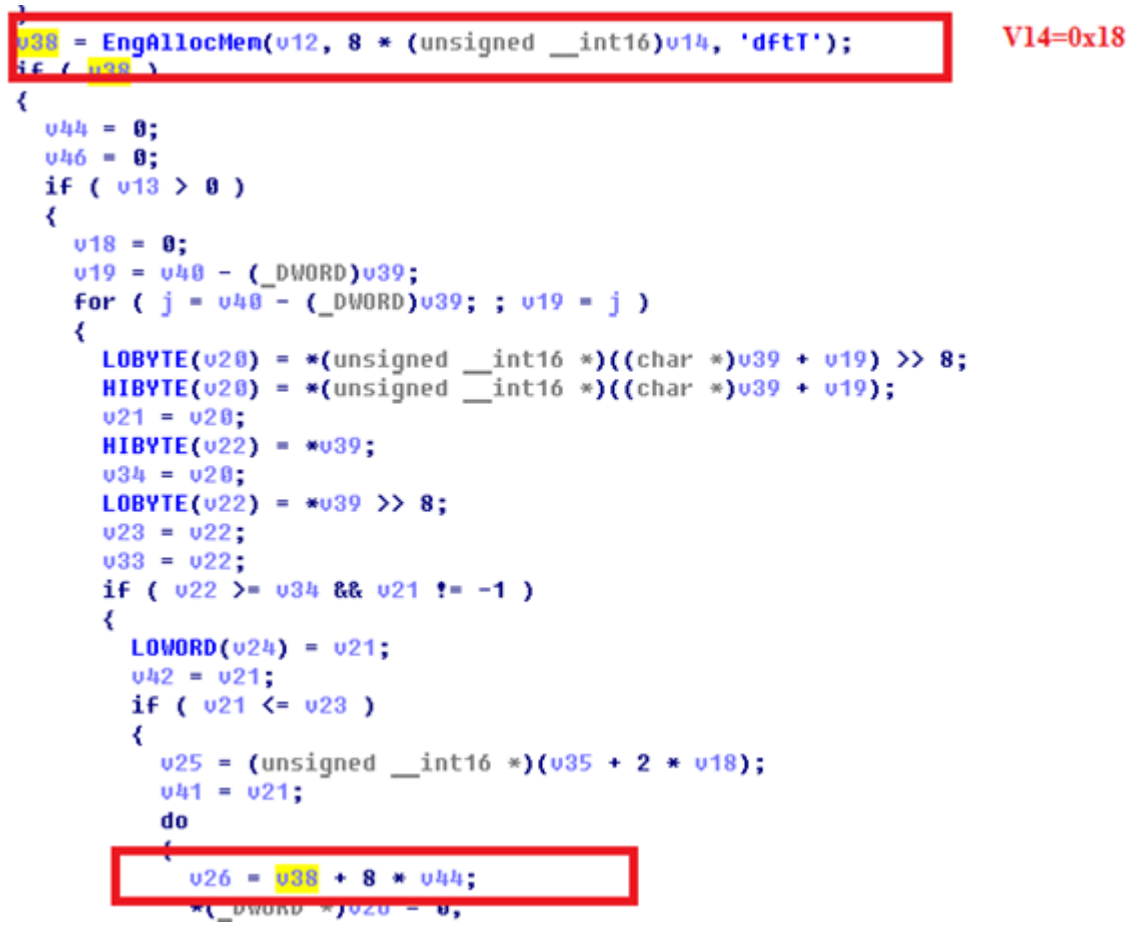
```
text:BF9EAB40 loc_BF9EAB40:                           ; CODE XREF: cjComputeGLYPHSET_MSFT_GENERAL(x,x,x,x)+75↑j
text:BF9EAB40                 movzx   eax, dx
text:BF9EAB43                 push    646674h         ; Tag
text:BF9EAB48                 shl     eax, 3
text:BF9EAB4B                 push    eax             ; int
text:BF9EAB4C                 push    ebx             ; char
text:BF9EAB4D                 call    _EngAllocMem@12 ; EngAllocMem(x,x,x)
```

The problem is that if we compute the entire table, we will achieve an integer overflow and the **cnt** variable will contain an incorrect value.

In kernel, we see the following picture:



The code allocates memory only for 0x18 InternalStruct but then there is a loop for all the segments range (this value was extracted from the file directly):



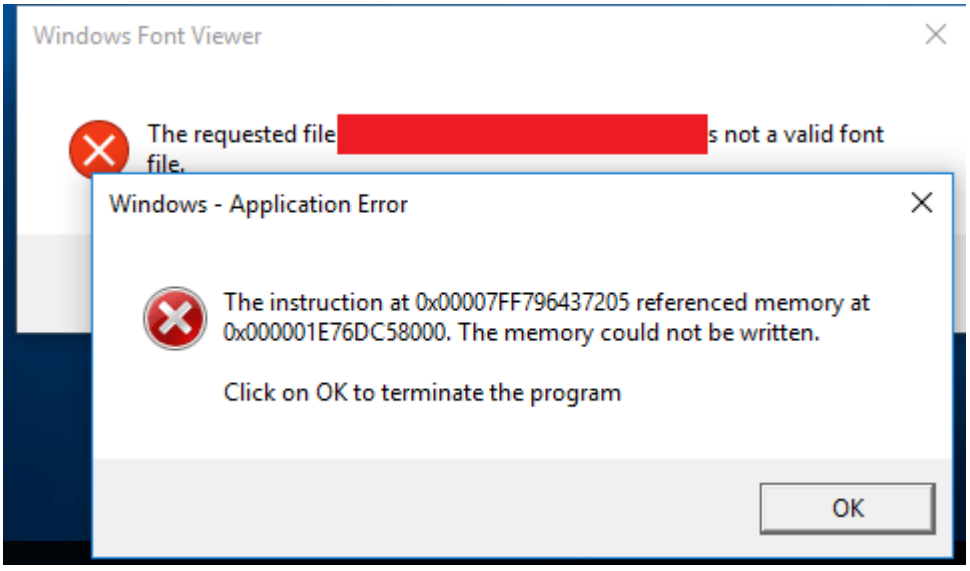
Using the cmap table, the v44 variable (index) could be controlled and, as a result, we get memory corruption. To achieve it, the attacker can do the following:

1. 1 Make an integer overflow in **win32k!cjComputeGLYPHSET_MSFT_GENERAL**
2. 2 Make a specific segment ranges in font file to access interesting memory.

What about Windows 10? As most of you know, the font processing in Windows 10 is performed in a special user mode process with restricted privileges. This is a very good solution but the code has the same bug in the TTF processing.

```
.text:00000001400565B9                  jz      short loc_1400565C5
.text:00000001400565BB                  sub     ax, cx
.text:00000001400565BE                  inc     ax
.text:00000001400565C1                  add     r8w, ax
.text:00000001400565C5
.text:00000001400565C5 loc_1400565C5:                           ; CODE XREF:
.text:00000001400565C5                                          ; cjComputeGl
.text:00000001400565C5                  inc     dx
.text:00000001400565C8                  add     rbx, 2
.text:00000001400565CC                  movzx   eax, dx
.text:00000001400565CF                  add     r9, 2
.text:00000001400565D3                  cmp     eax, ebp
.text:00000001400565D5                  jl      short loc_1400565A1
.text:00000001400565D7
.text:00000001400565D7 loc_1400565D7:                           ; CODE XREF:
.text:00000001400565D7                  movzx   edx, r8w
.text:00000001400565DB                  xor     ecx, ecx        ; uFlags
.text:00000001400565DD                  shl     rdx, 3          ; dwBytes
.text:00000001400565E1                  call    cs:__imp_GlobalAlloc
.text:00000001400565E7                  mov     rdi, rax
```

As a result, if you load/open this font exploit in Windows 10, you will see the crash of fontdrvhost.exe:



Kaspersky Lab detects this exploit as:

- HEUR:Exploit.Win32.Generic
- PDM:Exploit.Win32.Generic

We would like to thank Microsoft for their swift response in closing this security hole.

** More information about the FruityArmor APT group is available to customers of Kaspersky Intelligence Services. Contact: intelreports@kaspersky.com*

Source: https://securelist.com/windows-zero-day-exploit-used-in-targeted-attacks-by-fruityarmor-apt/76396/