A Missed 0day ? - Reveal another Cyber Arsenal of APT-C-06

11月12, 2018

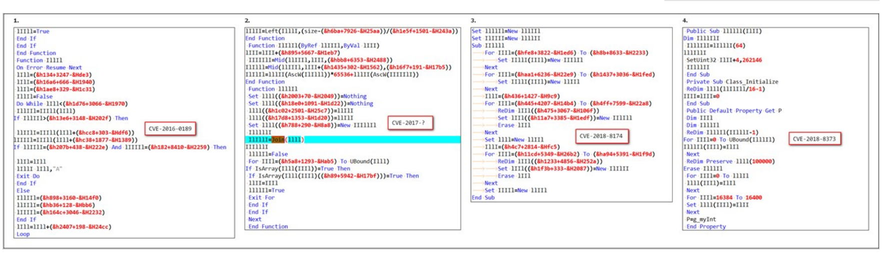
文章目录

- Introduction
- A Missed 0day?
- Vulnerability Analysis
  - Overview
  - PoC
  - Code Analysis
    - Prepare Memory
    - Overwrite the length of BSTR
    - Construct Length-exceeding Array
    - Arbitrary Address Reading
    - Construct Auxiliary Function
    - Execute shellcode
  - Dynamic Debugging
    - Prepare Memory
    - Memory Reuse
    - Overwrite the length of BSTR
    - Construct Length-exceeding Array
    - Execute shellcode
  - Analysis of the Patch
- Correlation with APT-C-06
- Reader's Takeaways
- Summary
- References

Introduction

Recently, the Advanced Threat Response Team of 360 Core Security has discovered several VBScript vulnerabilities exploited in the wild, including CVE-2016-0189, CVE-2018-8373 and another previously unknown vulnerability (we have not yet known which CVE number it corresponds). These three vulnerabilities, plus the one we discovered in April this year (CVE-2018-8174), there are in total four in the wild. After analysis, we found that the obfuscation and exploitation of these four files are highly consistent. We suspect that they are from the same hacker (or hacking group) that has been developing 0day exploit of VBScript and used it for attacks.

Here are the in-the-wild exploits of the four vulnerabilities we discovered:



1.

```
lIIIl=True
End If
End If
End Function
Function IIIIl
On Error Resume Next
lIIl=(&h134+3247-&Hde3)
lIIl=(&h16a6+666-&H1940)
lIIl=(&h1ae8+329-&H1c31)
IIIIl=False
Do While lIIl<(&h1d76+3066-&H1970)
IIIIIl=IIIIl(lIIl)
If IIIIIl>(&h13e6+3148-&H202f) Then

lIIIlI=IIIIl(lIIl+(&hcc8+303-&Hdf6))
lIIIIl=IIIIl(lIIl+(&hc38+1877-&H1389))
If lIIIlI=(&h207b+438-&H222e) And lIIIIl=(&h182+8410-&H2259) Then

lIIl=lIIl
lIIII lIIl,"A"
Exit Do
End If
Else
IIlIIl=(&h898+3160-&H14f0)
lIIIIl=(&hb36+128-&Hbb6)
lIIIIl=(&h164c+3046-&H2232)
End If
lIIl=lIIl+(&h2407+198-&H24cc)
Loop
```

CVE-2016-0189

2.

```
lIIII=Left(IIIlI,(size-(&h6ba+7926-&H25aa))/(&h1e5f+1501-&H243a))
End Function
Function IIIIIl(ByRef lIIIIl,ByVal lIII)
lIII=lIII+(&h895+5667-&H1eb7)
IIIIIlI=Mid(lIIIIl,lIII,(&hbb8+6353-&H2488))
IIIIl=Mid(lIIIIl,lIII+(&h1435+302-&H1562),(&h16f7+191-&H17b5))
IIIIIl=lIIII(AscW(IIIIIl))*65536+lIIII(AscW(IIIIIlI))
End Function
Function lIIIIl
Set lIIl((&h2003+70-&H2049))=Nothing
Set lIIl((&h18e0+1091-&H1d22))=Nothing
lIIl((&h1c02+2501-&H25c7))=lIIII
lIIl((&h17d8+1353-&H1d20))=lIIII
Set lIIl((&h788+290-&H8a8))=New IIIIIIl
lIIIIIl
IIIIIl=Join(lIIl)
IIIIIl
lIIIIl=False
For IIIl=(&h5a8+1293-&Hab5) To UBound(IIIl)
If IsArray(IIIl(IIII))=True Then
If IsArray(IIIl(IIII)((&h89+5942-&H17bf)))=True Then
lIII=IIII
lIIIIl=True
Exit For
End If
End If
Next
End Function
```

CVE-2017-?

3.

```
Set lIIIIl=New lIIIII
Set IIIIII=New lIIIII
Sub IIIIIl
    For IIIl=(&hfe8+3822-&H1ed6) To (&h8b+8633-&H2233)
        Set IIIII(IIII)=New IIIIIl
    Next
    For IIIl=(&haa1+6236-&H22e9) To (&h1437+3036-&H1fed)
        Set IIIII(IIII)=New lIIIl
    Next
    IIII=(&h436+1427-&H9c9)
    For IIIl=(&h445+4207-&H14b4) To (&h4ff+7599-&H22a8)
        ReDim lIII((&h475+3067-&H106f))
        Set lIII((&h11a7+3385-&H1edf))=New IIIIIl
        Erase lIII
    Next
    Set lIIl=New lIIIl
    IIII=(&h4c7+2814-&Hfc5)
    For IIIl=(&h11cd+5349-&H26b2) To (&ha94+5391-&H1f9d)
        ReDim lIII((&h1233+4856-&H252a))
        Set lIII((&h1f3b+333-&H2087))=New IIIIII
        Erase lIII
    Next
    Set IIIIl=New lIIIl
End Sub
```

CVE-2018-8174

4.

```
Public Sub lIIIIl(IIII)
Dim IIIIIII
IIIIIII=IIIIIl(64)
lIIIIII
SetUint32 IIII+4,262146
IIIIIII
End Sub
Private Sub Class_Initialize
ReDim lIIl(lIIIIIl/16-1)
IIII=lIIII=0
End Sub
Public Default Property Get P
Dim IIII
Dim IIIIIl
ReDim IIIIIl(IIIIII-1)
For IIIl=0 To UBound(IIIIIl)
IIIIIl(IIII)=IIIl
Next
ReDim Preserve lIIl(100000)
Erase IIIIIl
For IIIl=0 To lIIIl
lIIl(IIII)=IIIl
Next
For IIIl=16384 To 16400
Set lIIl(IIII)=IIII
Next
P=g_myInt
End Property
```

CVE-2018-8373

A Missed 0day?

We and other security vendors have uncovered the other three vulnerabilities in previous analysis reports, so in this article, we will focus on the fourth VBScript 0day that has not been published.

The exploit has been hidden very concealed. We found that the vulnerability was fixed in the Microsoft March 2017 Security Update. Microsoft did not mention that the vulnerability was exploited. We speculated that's because the vulnerability might be fixed by Microsoft without finding any evidence of being exploited. The latest version we located that can trigger this vulnerability is VBScript.dll 5.8.9600.18538, which was fixed in VBScript.dll 5.8.9600.18616. What's interesting is that our analysis shows the relevant exploit can be dates back to earlier than March 2017, which also means that the vulnerability was a 0day at the time. Unfortunately, we haven't seen any analysis of the vulnerability from other vendors.

Below we will share with you the root causes and exploitation of this vulnerability.

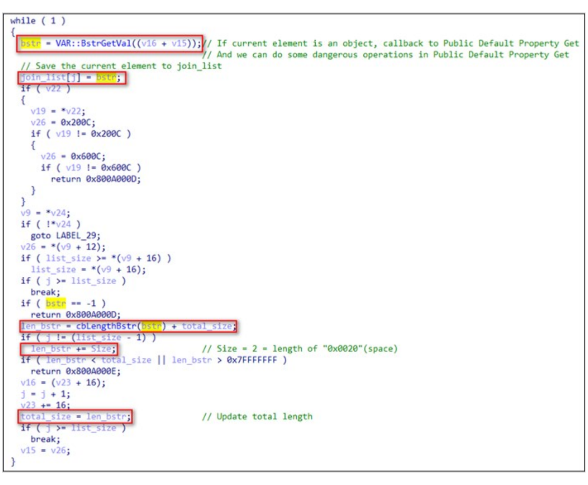
Vulnerability Analysis

Overview

This vulnerability is in function VBScript!rtJoin. While executing the join statement of VBScript, function VbsJoin will call rtJoin. Then rtJoin will first traverse each element in incoming array and calculates the total length of the stitched string (including the stitching characters, the default is unicode space 0x0020). Then call SysAllocStringLen to allocate the corresponding space for saving the stitched string.

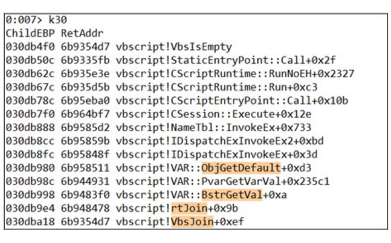
The actual allocated space size = (number of bytes to be allocated + 0x15) & 0xfffffff0 (see the implementation of oleaut32!SysAllocStringLen and oleaut32!CbSysStringSize)

The first 4 bytes of the string initial address are the byte length of the string (see the BSTR structure). The pseudo code for the entire process above is as follows:



```
while ( 1 )
{
  bstr = VAR::BstrGetVal((v16 + v15));// If current element is an object, callback to Public Default Property Get
                                      // And we can do some dangerous operations in Public Default Property Get
  // Save the current element to join_list
  join_list[j] = bstr;
  if ( v22 )
  {
    v19 = *v22;
    v26 = 0x200C;
    if ( v19 != 0x200C )
    {
      v26 = 0x600C;
      if ( v19 != 0x600C )
        return 0x800A000D;
    }
  }
  v9 = *v24;
  if ( !*v24 )
    goto LABEL_29;
  v26 = *(v9 + 12);
  if ( list_size >= *(v9 + 16) )
    list_size = *(v9 + 16);
  if ( j >= list_size )
    break;
  if ( bstr == -1 )
    return 0x800A000D;
  len_bstr = cbLengthBstr(bstr) + total_size;
  if ( j != (list_size - 1) )
    len_bstr += Size;                 // Size = 2 = length of "0x0020"(space)
  if ( len_bstr < total_size || len_bstr > 0x7FFFFFFF )
    return 0x800A000E;
  v16 = (v23 + 16);
  j = j + 1;
  v23 += 16;
  total_size = len_bstr;              // Update total length
  if ( j >= list_size )
    break;
  v15 = v26;
}
```

The corresponding stack backtrace is as below:



```
0:007> k30
ChildEBP RetAddr
030db4f0 6b9354d7 vbscript!VbsIsEmpty
030db50c 6b9335fb vbscript!StaticEntryPoint::Call+0x2f
030db62c 6b935e3e vbscript!CScriptRuntime::RunNoEH+0x2327
030db67c 6b935d5b vbscript!CScriptRuntime::Run+0xc3
030db78c 6b95eba0 vbscript!CScriptEntryPoint::Call+0x10b
030db7f0 6b964bf7 vbscript!CSession::Execute+0x12e
030db888 6b9585d2 vbscript!NameTbl::InvokeEx+0x733
030db8cc 6b95859b vbscript!IDispatchExInvokeEx2+0xbd
030db8fc 6b95848f vbscript!IDispatchExInvokeEx+0x3d
030db980 6b958511 vbscript!VAR::ObjGetDefault+0xd3
030db98c 6b944931 vbscript!VAR::PvarGetVarVal+0x235c1
030db998 6b9483f0 vbscript!VAR::BstrGetVal+0xa
030db9e4 6b948478 vbscript!rtJoin+0x9b
030dba18 6b9354d7 vbscript!VbsJoin+0xef
```

The parsing process then copies the strings one by one to the newly allocated space. In this process, the length of each string is obtained using the string address stored on the stack, as the size parameter of memcpy. When there is a class object in the array elements, it will trigger the Default Property Get callback of the class object to get the default property. In the callback, you can operate on other members in the array, such as changing the string size. As long as the string size before and after the change is precisely controlled, the data size copied through memcpy (the first one in the figure below) may exceed the space allocated by SysAllocStringLen, resulting in heap overflow. The pseudo code for the entire process above is as follows:

PoC

We constructed the PoC of this vulnerability for researcher to use:

Code Analysis

Prepare Memory

The original code first prepares the memory and then exploits the vulnerability (exp_1) for the first time, overwriting the length field of a BSTR object and obtaining a very long BSTR object. Then it will use this BSTR to obtain a previously prepared memory address; After it succeeds, the vulnerability (exp_2) is used again to overwrite the object type of a forged string from 0008 to 200c, thus obtaining a unidimensional length-exceeding array with starting address being 0, element size being 1, and element number being 0x7fffffff.

With the first obtained memory address and the second obtained length-exceeding array, arbitrary address reading is then realized. The subsequent exploitation is basically the same as the previously disclosed details.

Below, it shows the first half of memory preparing code:

In this part of the code, the string length of str_h is 0x4fec bytes. Its actual space allocated by SysAllocStringLen is 0x5000 bytes ((0x4fec+0x15) & 0xfffffff0 = 0x5000) while that of str_o (actual sting length is 0x4ff6 bytes) is 0x5000 bytes ((0x4ff6+0x15) & 0xfffffff0 = 0x5000). Array_a and array_b are two arrays, and the actual memory each array occupies is 0xa00*0x10 = 0xa000 bytes (each element is a VAR structure).

Note that 0x4fec*2 + 0x18 + 0x2*2 = 0x9ff4, (0x9ff4+0x15) & 0xfffffff0 = 0xa000, these values will be mentioned in later part of this report.

Below is the second half of memory preparing code:

Str_left_0 is 0x4ffa bytes (get_left_str_a_by_size will reduce the incoming parameter by 6 bytes), and SysAllocStringLen allocates 0x5000 bytes for it ((0x4ffa + 0x15) & 0xfffffff0 = 0x5000); Str_left_1 is 0x9ffa bytes, and the memory allocated by SysAllocStringLen is 0xa000 bytes ((0x9ffa + 0x15) & 0xfffffff0 = 0x5000).

Each element of array2 is then assigned with value str_left_1 (the actual memory size is 0xa000); Each element of array3 is assigned with value array_b whose actual memory size is 0xa000 (see above).

Now, the memory preparation is complete. Afterwards, as long as some elements of array2 (operating in exp_1) or array3 (operating in exp_2) are released, there will be a large number of memory holes with a size of 0xa000 for each. At this point, immediately request for 0xa000 bytes can make it highly possible to reuse the released memory.

As long as the requested size by SysAllocStringLen in function rtJoin is 0xa000 bytes, together with the above vulnerabilities, the data overwriting on certain str_left_1 object in array2 or an array_b object in array3 can be achieved, which will be described in details later.

Overwrite the length of BSTR

In exp_1, it is the first time to trigger the vulnerability, and the length of a BSTR object is rewritten to 0xfffffffe.

First, assign the first and second elements of array_c with str_h (the string length is 0x4fec bytes, the actual occupied space is 0x5000 bytes, see above), and the third element is assigned the object of class_a whose Default Property Get can return a length of 0x18 bytes (0x1a-0x6+0x4 = 0x18). As a result, the spliced length of the three elements of the array plus the separator byte is 0x9ff4 (0x4fec+0x4fec+0x18+0x2+) 0x2 = 0x9ff4)

Before triggering the vulnerability, release half of the elements in array2 before calling make_hole_of_array2 to generate enough memory holes with the size of 0xa000.

The corresponding memory before and after calling make_hole_of_array2 is as follows. It can be seen that that half of the string memory in array2 is released. For the elements whose subscripts are in the range 0x00-0x7F, the even parts are released; for the elements whose subscripts are in the range of 0x80-0xFF, the odd parts are released:

Subsequently, SysAllocStringLen in function rtJoin will request a BSTR with a total length of 0xa000 bytes ((0x9ff4 + 0x15) & 0xfffffff0 = 0xa000). Due to the heap allocation algorithm of Windows, this memory will reuse one from the free heap block on the right side of the above figure.

In the Default Property Get of class, the first and second elements of array_c (set to Nothing) will be released first, and immediately assigned with value str_o (the string length is 0x4ff6 bytes, the actual occupied space is 0x5000 bytes).

There are two things to be noted here:

- 1) The two operations with str_o 2 will reuse the 2 0x5000 memory blocks just released (that is, the memory occupied by the original two str_h).
- 2) After the reuse, the length and content of the string at the same address have changed (the original is str_h, length 0x4fec bytes, now it's changed to str_o, length 0x4ff6). So the string length retrieved before memcpy in rtJoin are 0x4ff6, 0x4ff6, 0x18; plus two separator character 0x4, memcpy totally copies data of 0xa008, with 0x14 more bytes than 0x9ff4, the last 4 of the extra bytes will overwrite the length field of the adjacent str_left_1 object in array2. In the exploit code, the attacker overwrote the length of the original str_left_1 to 0xfffffffe.

The mismatching process is shown below:

Afterwards, the previously prepared string address is obtained with the length-exceeding string, for subsequent use.

The following figure shows the string prepared in function prepare:

Construct Length-exceeding Array

In exp_2, the vulnerability is triggered for the second time. The type of the string corresponding to fake_array is changed to 0x200c. The method is the same as overwriting the length of the string. So it will not be repeated here.

Fake_array is a string that is actually a fake tagSAFEARRAY structure.

The following is a length-exceeding array after retrieving for type obfuscation, for later use:

Arbitrary Address Reading

Later, the sample encapsulates a set of functions for arbitrary address reading with the string address and the length-exceeding array obtained earlier, for later use.

Construct Auxiliary Function

With the ability to read arbitrary addresses, the exploit also encapsulates several auxiliary functions:

Then leak the virtual table address of the CScriptEntryPoint object through the following method:

After that, use the encapsulated auxiliary function to get the base address of VBScript.dll. Next, get the msvcrt.dll base address by traversing the VBScript.dll import table. msvcrt.dll introduces kernelbase.dll, ntdll.dll. Finally, the function addresses of NtContinue, VirtualProtect, etc. are obtained. The whole process is as follows:

Execute shellcode

In Windows 7 and Windows 8, the way the original exploit executes shellcode is the same as the previous CVE-2018-8174, while in Windows 8.1 and Windows 10, it is slightly different from that in earlier version systems.

Dynamic Debugging

Prepare Memory

In function prepare, when the memory preparation is completed, array2, array3 and the pvData of array_c is shown below:

Memory Reuse

Firstly, here is the memory reuse of string str_h by string str_o in the Public Default Property Get callback. After the reuse, the overall memory size does not change but the length of the string changes.

Then, there is the reuse of a released 0xa000 string in array2 when SysAllocStringLen requests 0xa000 sized memory. As you can see from the figure below, before the vulnerability was triggered for the first time, the memory that was reused is the array2 (0x81) that was just released. Then the length of the string corresponding to array2 (0x82) will be overwritten.

Overwrite the length of BSTR

The first time the vulnerability was triggered in exp_1, the length field of a str_left_1 string was overwritten.

Construct Length-exceeding Array

Memory reuse is performed again in exp_2. This time, the 0xa000 memory requested by SysAllocStringLen is the memory that has just been released by array3 (0x81) (the release mode is the same as array2), and then the header of the associated memory of array3 (0x82) will be overwritten.

When the vulnerability is triggered for the second time, the type of the carefully prepared string fake_array is rewritten from 0008 to 200c, resulting in a unidimensional length-exceeding array.

Execute shellcode

The shellcode implementation details in Windows 7 and Wndows 8 can be found in the CVE-2018-8174 analysis article we wrote earlier. In Windows 8.1 and Windows 10, the sample used other tricks to bypass CFG (in our tests, this method was successful in earlier versions of Windows 8.1 and Windows 10). More details on this section will be disclosed later.

Analysis of the Patch

Below is the comparison of the Bindiff tool before and after the patch:

It can be seen that before copying each array element to the join_list in the patch file, the string data is firstly saved by SysAllocString, so that even if the initial string length is changed in the later callback, the data saved using function SysAllocString will be copied when excute memcpy for memory copycopying. In this way, the memory size requested by SysAllocStringLen is the same as the data size of the memcpy copy, thus fixing the vulnerability.

Correlation with APT-C-06

We analyzed the shellcode of the four VBScript vulnerabilities. We found that the shellcode used by CVE-2016-0189, this vulnerability in this article and CVE-2018-8174 are basically the same except the configured CC. The shellcode of CVE-2018-8373 is slightly different, but also very similar on the whole. We speculate that this vulnerability is also one of the APT-C-06 (aka Darkhotel) arsenals.

Reader's Takeaways

Have you noticed that there is an integer overflow point in the rtJoin function?

We retrieved the integer overflow vulnerability related to the join series function in VBScript, and found a vulnerability CVE-2017-11869. We made a patch comparison of VBScript.dll before and after the fix and found some interesting modifications, as follows:

If you are interested in them, you can do your own research on CVE-2017-11869.

Summary

In this article, we shared the details of the third VBScript vulnerability discovered this year of which the exploitation is as impressive as the previous ones. We believe that there are other similar issues in VBScript, and speculate that there are other similar exploits are under the control of the hacker or hacking group. Here we would like to warn you to stay sharp on these kinds of vulnerabilities out there.

References

http://blogs.360.cn/post/cve-2018-8174-en.html

https://www.zerodayinitiative.com/advisories/ZDI-17-916/

Contact: ATResonse@360.cn

本文链接：http://blogs.360.cn/post/VBScript_vul_EN.html

-- EOF --

作者 360CoreSec 发表于 *2018-11-12 10:23:59* ，添加在分类 Threat Intelligence Vulnerability Analysis 下 ，最后修改于 *2018-11-12 12:01:39*

分享到： 新浪微博 微信 Twitter 印象笔记 QQ好友 有道云笔记

« Not a fair game, Dice2win公平性分析

被遗漏的0day？—APT-C-06组织另一网络武器库分析揭秘 »

Comments