Untangling Kovter's persistence methods | Malwarebytes Labs

By Malwarebytes Labs

Published: 2016-07-13 · Archived: 2026-04-05 17:50:43 UTC

Kovter is a click-fraud malware famous from the unconventional tricks used for persistence. It hides malicious modules in PowerShell scripts as well as in registry keys to make detection and analysis difficult. In this post we will take a deep dive into the techniques used by it's latest samples to see all the elements and how they cooperate together.

Analyzed samples

#1:

- 49a748e9f2d98ba92b5af8f680bef7f2 – original executable
  - 4160d0e5938b2ff29347476788f3810e – intermediate payload (loader)
    - 7d40b09885f8b967b1127032e54adad4 – unpacked payload (raw dump)

2#:

- 78c622b295114aa0004b2a8cba8df371 – original executable
  - 3a453e3a77fe7e1534b578f79ad3e987 – intermediate payload (loader)
    - 05956dd290271a6bc810d17893cee826 – unpacked payload (raw dump)

// special thanks to

@F_kZ_

and @JAMESWT_MHT for sharing the samples

Behavioral analysis

Authors of Kovter put a lot of effort in making their malware stealth and hard to detect. During the initial assessment of some of the Kovter samples we could notice that it is signed by valid Comodo certificate (it got revoked later):

Copyright	Skibsfrernes
Product	Oversigtsartikel2
Original name	Slangebinderes.exe
Internal name	Slangebinderes
File version	2.00
Description	Unpercolated5
Signature verification	Signed file, verified signature
Signing date	1:01 PM 6/9/2016
Signers	[+] DIDZHITAL ART

Status	Valid
Issuer	COMODO RSA Code Signing CA
Valid from	1:00 AM 5/13/2016
Valid to	12:59 AM 5/14/2017
Valid usage	Code Signing
Algorithm	sha256RSA
Thumbprint	A286AFFC5F6E92BDC93374646676EBC49E21BCAE
Serial number	00 B1 BB EF 3A BA 79 AB 2E AE 5B 80 15 F2 6B 34 F8

[+] COMODO RSA Code Signing CA
[+] COMODO SECURE?

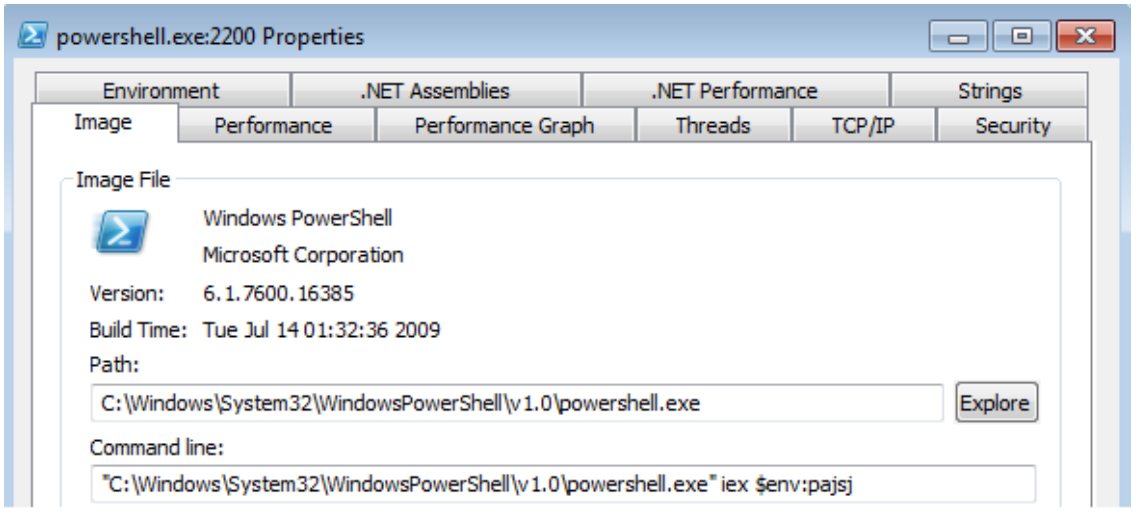
Counter signers	[+] COMODO SHA-1 Time Stamping Signer [+] UTN-USERFirst-Object [+] The USERTrust Network?

After the sample gets deployed, Kovter runs PowerShell and install itself in the system

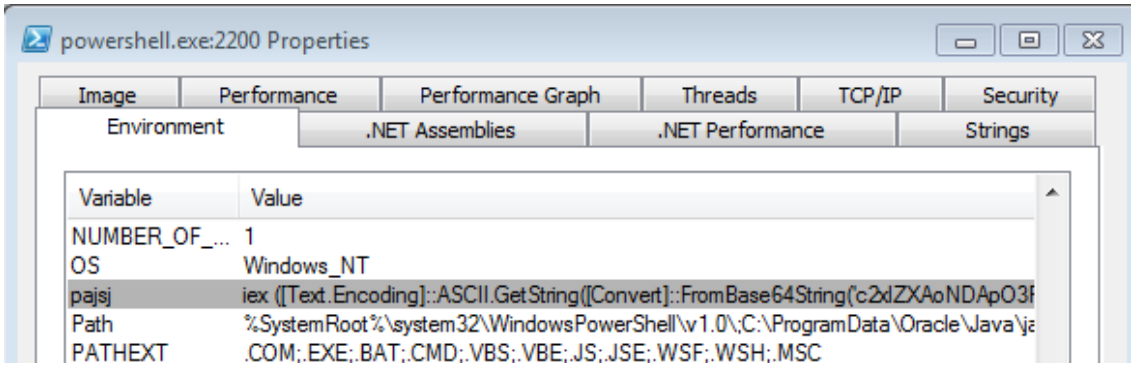
services.exe	4.19	3 996 K	5 328 K	460		
svchost.exe	0.01	2 548 K	5 592 K	584	Host Process for Windows S...	Microsoft Corporation
WmiPrvSE.exe		1 696 K	4 356 K	1420		
WmiPrvSE.exe		3 884 K	7 552 K	2204		
mshta.exe		6 680 K	12 096 K	2208	Microsoft (R) HTML Applicati...	Microsoft Corporation
powershell.exe		35 004 K	33 528 K	1148	Windows PowerShell	Microsoft Corporation
VBoxService.exe	< 0.01	1 416 K	3 832 K	648		

Observing it via Process Explorer we can find the command passed to PowerShell. It's purpose is to execute a code stored in an environment variable (names are random, new on each run), i.e:

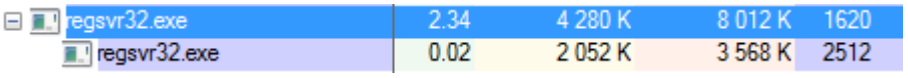
```
$env:nvwisqng
```



Content of the variable is a base64 encoded PowerShell script:



After that initialization phase, we can see PowerShell deploying *regsrv32.exe* (via which Kovter runs it's modules):



Examining the network activity we can notice many new connections with the *regsrv32.exe* that are appearing and disappearing:

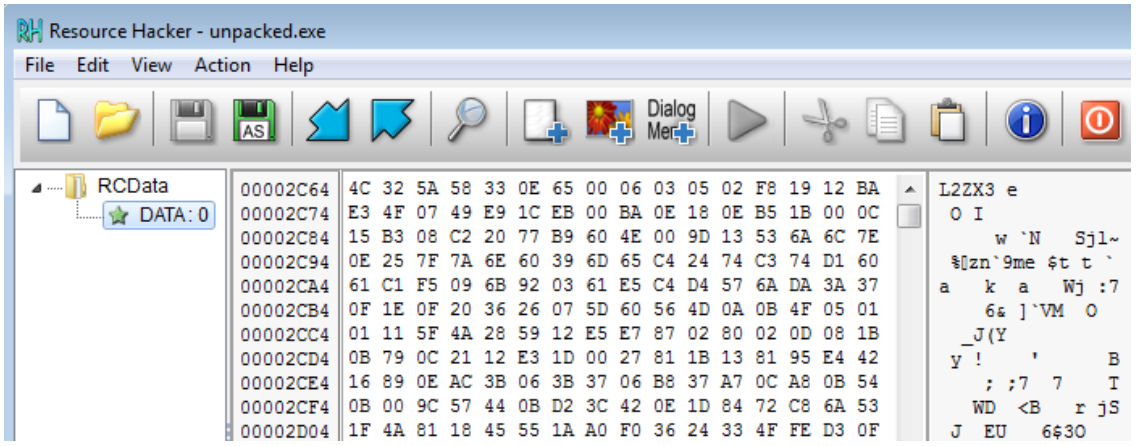
Process	PID	Protocol	Local Address	Local Port	Remote Address	Remote Port	State	Sent Packets
lsass.exe	480	TCP	testmachine	49156	testmachine	0	LISTENING	
lsass.exe	480	TCPV6	testmachine	49156	testmachine	0	LISTENING	
regsvr32.exe	3640	TCP	testmachine	49335	79.142.77.157	http	SYN_SENT	
regsvr32.exe	3640	TCP	testmachine	49336	109.225.177.86	8080	SYN_SENT	
regsvr32.exe	3640	TCP	testmachine	49337	185.62.214.51	http	SYN_SENT	
regsvr32.exe	3640	TCP	testmachine	49338	58.214.89.219	8080	SYN_SENT	
regsvr32.exe	3640	TCP	testmachine	49339	136.94.134.27	http	SYN_SENT	
services.exe	464	TCP	testmachine	49155	testmachine	0	LISTENING	

We can expect that it is related with the click-fraud activity, performed by the malware.

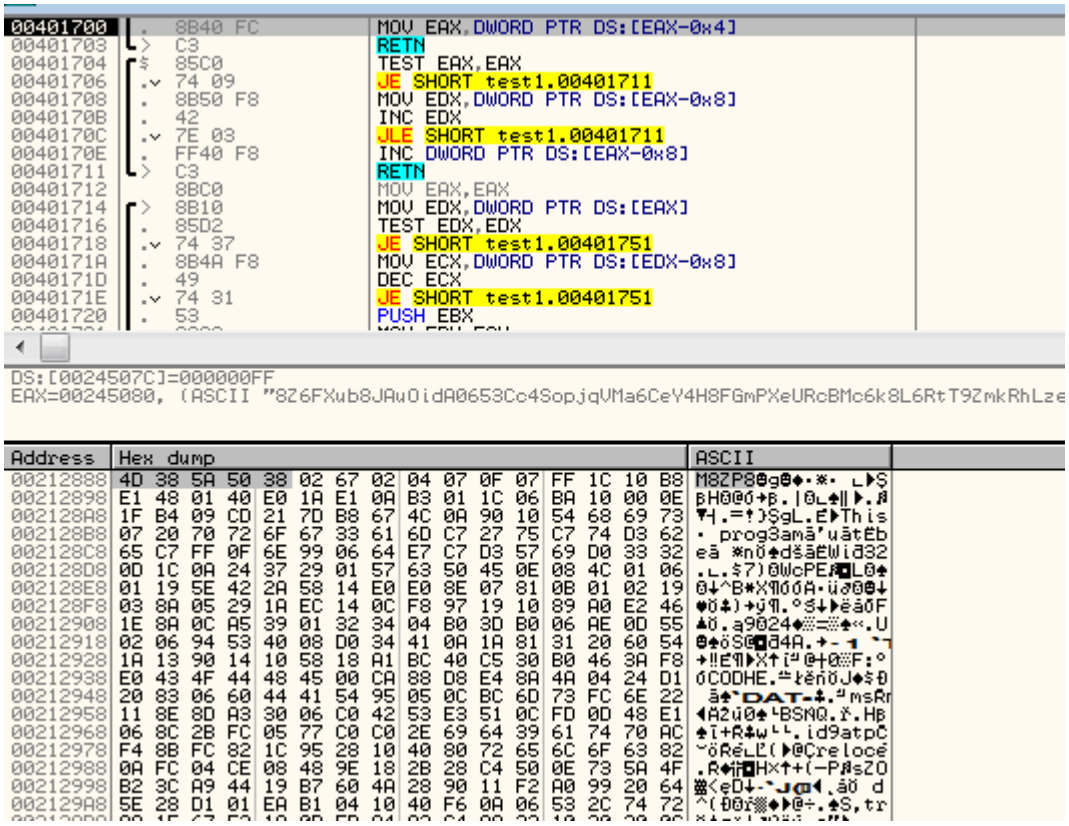
Inside

Unpacking

Kovter comes packed by a crypter/FUD. After unpacking it we get the loader that is another PE file. It comes with a binary data in the resources:



Content of the resource is a next PE file, encrypted and compressed with aPLib (we can easily recognize the algorithm by the typical way in which it modified MZ header):



During the execution, this unpacked PE file is loaded into an newly allocated, continuous area in the memory (without dividing content into sections). The original (host) sample loads all the necessary DLLs and applies relocations on the new module. Then, the execution is redirected there (see below):

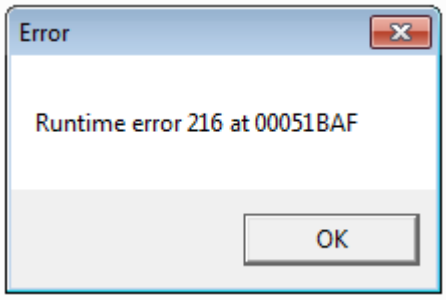
```
004029D9  .   MOV [LOCAL.2],EAX
004029DC  .   MOV BL,0x1
004029DE  .   XOR EAX,EAX
004029E0  .   PUSH EAX
004029E1  .   PUSH 0x1
004029E3  .   PUSH [LOCAL.1]                 the new PE loaded in memory: 1320000
004029E6  .   CALL [LOCAL.2]                 reditect execution to the new PE
004029E9  .   PUSH -0x1                      Timeout = INFINITE
004029EB  .   CALL <JMP.&kernel32.Sleep>     Sleep
004029F0  >   XOR EAX,EAX
Stack SS:[0012FEF4]=01378824

Address   Hex dump                                        ASCII
01320000  4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00  MZP.☻...♦.☼.  ..
01320010  B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00  ©.......@.→.....
01320020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01320030  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ............☺...
01320040  BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90  ||►.♫▼┤.=↑$☺L=↑ÉÉ
01320050  54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73  This program mus
01320060  74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57  t be run under W
01320070  69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00  in32..$7........
```

Payload's Entry Point:

```
01378824   PUSH EBP                              entry point in the new PE
01378825   MOV EBP,ESP
01378827   ADD ESP,-0xC
0137882A   MOV EAX,0x1378694
0137882F   CALL 01326508
01378834   XOR EAX,EAX
01378836   PUSH EBP
01378837   PUSH 0x137886F
0137883C   PUSH DWORD PTR FS:[EAX]
0137883F   MOV DWORD PTR FS:[EAX],ESP
01378842   PUSH 0x0
01378844   PUSH 0x0
```

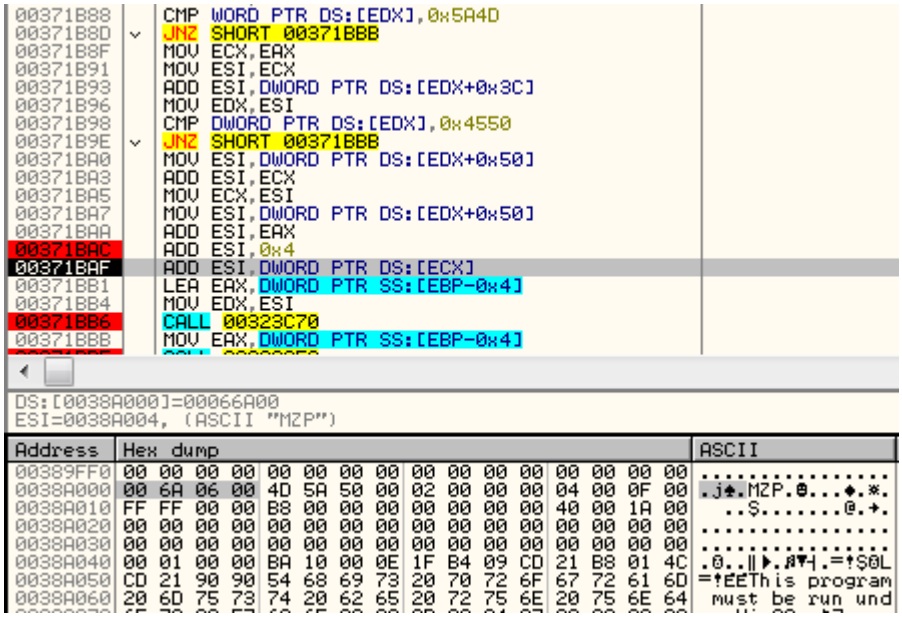
This executable turns out to be a final stage, containing the core malicious functions. Looking at it's strings (that are not obfuscated) we can find that this is the module responsible for further steps of infection – writing the registry keys, dropping malicious scripts etc. The easiest way to proceed with the analysis would be to dump the payload and analyze it as a separate entity. However, authors of the malware added some tricks in order to prevent this executable from running independently. When we dump the unpacked payload and try to run it, it will crash:



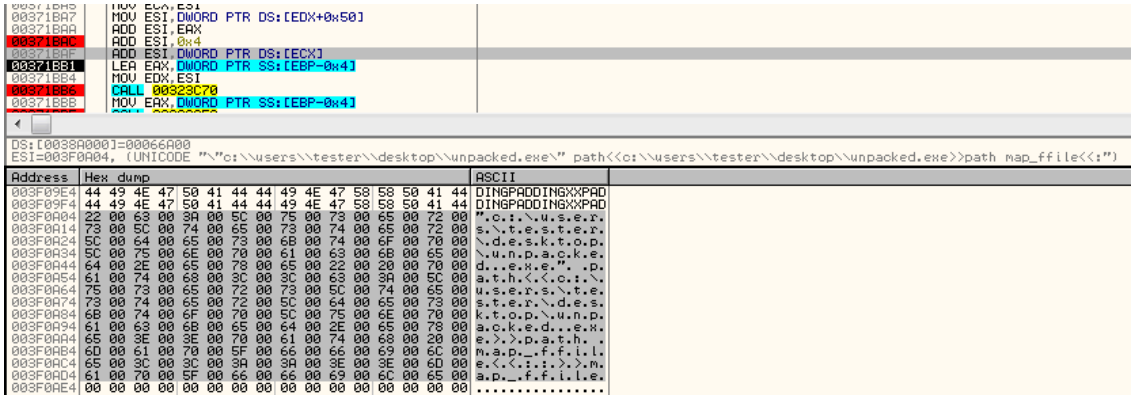
Analyzing the point of crashing, we can see the reason – trying to read unaccessible memory:

```
00451B81  .   CALL p2.0044AE88
00451B86  .   MOV EDX,EAX                    EAX -> ImageBase
00451B88  .   CMP WORD PTR DS:[EDX],0x5A4D   'MZ' -> beginning of DOS header
00451B8D  .v  JNZ SHORT p2.00451BBB
00451B8F  .   MOV ECX,EAX                    ECX = EAX -> ImageBase
00451B91  .   MOV ESI,ECX
00451B93  .   ADD ESI,DWORD PTR DS:[EDX+0x3C]
00451B96  .   MOV EDX,ESI
00451B98  .   CMP DWORD PTR DS:[EDX],0x4550  'PE' -> beginning of PE header
00451B9E  .v  JNZ SHORT p2.00451BBB
00451BA0  .   MOV ESI,DWORD PTR DS:[EDX+0x50]
00451BA3  .   ADD ESI,ECX
00451BA5  .   MOV ECX,ESI
00451BA7  .   MOV ESI,DWORD PTR DS:[EDX+0x50]
00451BAA  .   ADD ESI,EAX                    p2.00400000
00451BAC  .   ADD ESI,0x4                    ESI = ImageBase + SizeOfImage + 4
00451BAF  .   ADD ESI,DWORD PTR DS:[ECX]     add value mapped after ImageBase+SizeOfImage
00451BB1  .   LEA EAX,[LOCAL.1]
DS:[0046A000]=???
ESI=0046A004
```

As we can see in the above code, sample reads it's own headers and find the end of the mapped module (ImageBase + VirtualSize). Then it tries to read from the address that is exactly after it. In case if the sample was loaded via the host application, this address is accessible and the content is filled with appropriate value – see the example below:



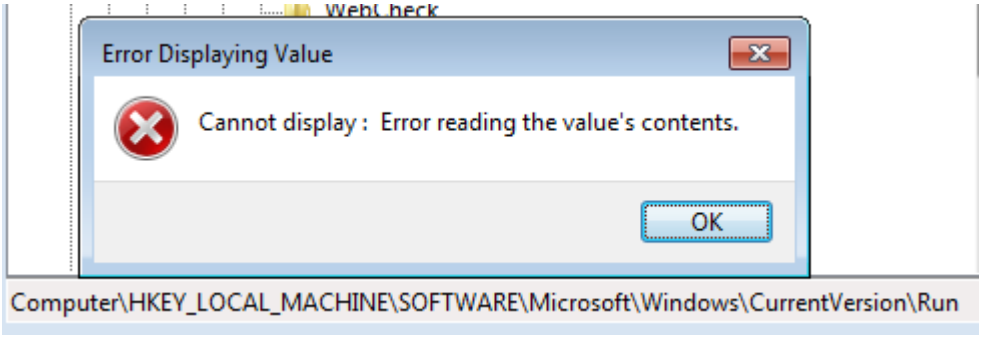
The read value points to the address where some content has been written – including the path to the current sample:



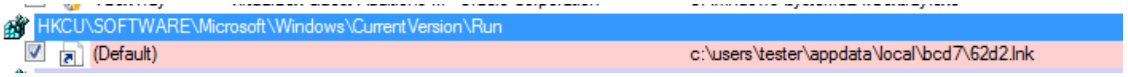
However, it doesn't work if the sample was executed independently. That's why we encounter the crash. Anyways, the dump can be very useful for the static analysis.

Persistence

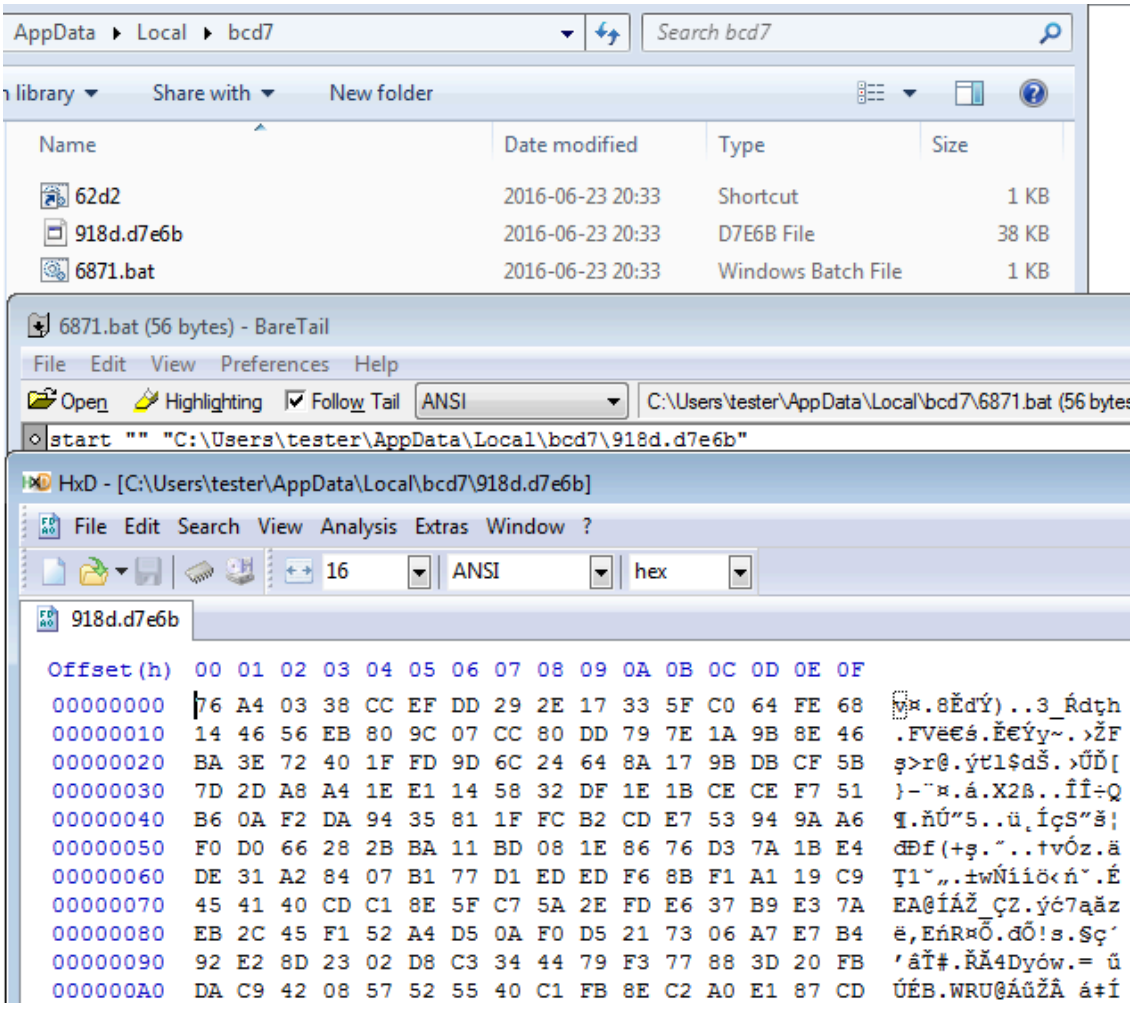
Kovter achieves persistence by adding a Run key in the Windows Registry. Access to this key via *regedit* is restricted:



But using Sysinternals's tool – *autoruns* still we can see where it leads:

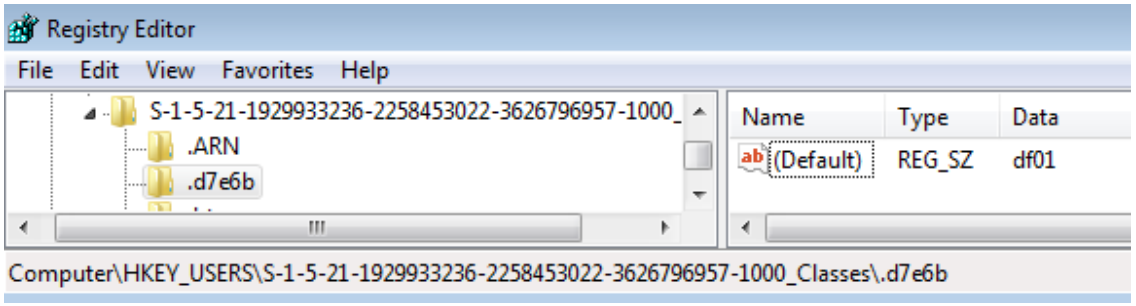


It refers to the link that leads to a batch script, running a dropped file of an unknown format:

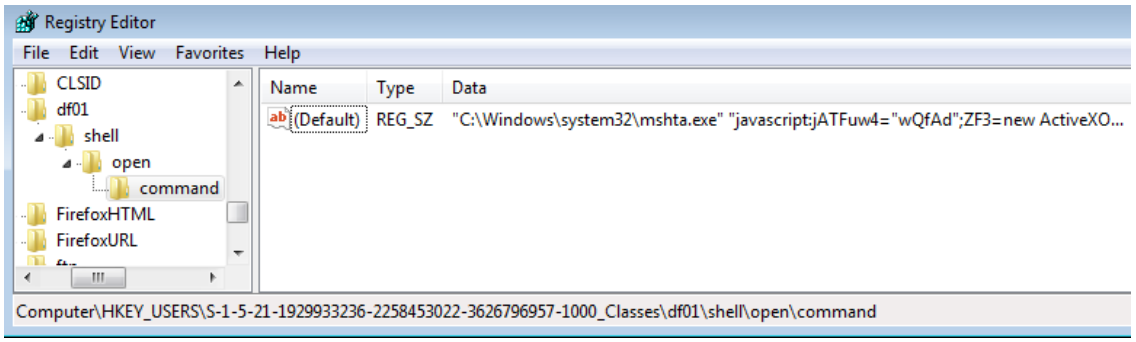


This file with the extension ***.d7e6b*** has unreadable content and it doesn't make much sense until we notice how it is opened. Kovter's executable, during the installation process, registered in the Windows Registry a special way to run this type of files.

Added extension ***.d7e6b*** is handled by a newly defined command ***df01***:



That command is defined in another registry key ***(…df01shellopencommand***):

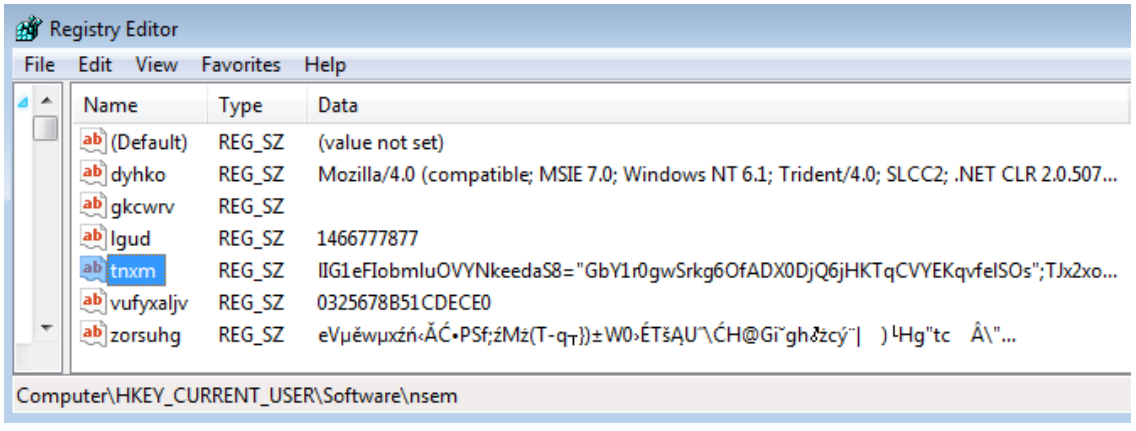


It uses a system application *mshta.exe* in order to run the JavaScript that decrypts the content of the file in memory and loads it.

Content of the script:

```
jATFuw4="wQfAd"; ZF3=new ActiveXObject("WScript.Shell"); XZs6HOl="uE6"; gdV5K2=ZF3.RegRead("HKCU\sof
```

It refers to other dropped registry keys, saved under a different path (names are different for different samples – in the current sample it is: "***HKCUsoftwarensem***"):



Inside the variable ***tnxm*** another obfuscated script is embedded:

https://gist.github.com/hasherezade/184b23a2a98831061fc4b18473078542#file-tnxm-js

The script contains simple obfuscation – variables have meaningless names and some unused strings are added in between to create a noise. After renaming variables and removing junk the same code looks much more readable:

https://gist.github.com/hasherezade/184b23a2a98831061fc4b18473078542#file-tnxm_deobfuscated-js

The hex_string contains hexadecimal representation of an encrypted code. It is processed by the following two loops. First loop converts it from the text representation into a binary. Second one – performs XOR decryption (the XOR key is random, generated newly on each run). Then, the result is executed by ***eval*** function.

Decrypted content: https://gist.github.com/hasherezade/184b23a2a98831061fc4b18473078542#file-txnm_deobfuscated_decoded2-ps1 The base64 content is the same like the one we encountered during behavioral analysis – (it was set in in the environment variable).

https://gist.github.com/hasherezade/184b23a2a98831061fc4b18473078542#file-nvwisqng-txt

After decoding it turns out to be a PowerShell Script:

https://gist.github.com/hasherezade/184b23a2a98831061fc4b18473078542#file-nvwisqng_decoded-ps1

It's role is to load and execute the code hidden in the variable $sc32. It contains position-independent 32bit code (it will be referred as a *shellcode*). Content is loaded to the newly allocated memory page and executed in a new thread. Pointer to the allocated memory is passed to the thread as a parameter (it is very important, because this address is further used for resolving pointers to variables).

Below you can see the beginning of this code, converted to binary:

```
00401000  $  55              PUSH EBP
00401001  .  8BEC            MOV EBP,ESP
00401003  .  81C4 00FAFFFF   ADD ESP,-0x600
00401009  .  53              PUSH EBX
0040100A  .  56              PUSH ESI
0040100B  .  57              PUSH EDI
0040100C  .  53              PUSH EBX
0040100D  .  56              PUSH ESI
0040100E  .  57              PUSH EDI
0040100F  .  FC              CLD
00401010  .  31D2            XOR EDX,EDX
00401012  .  64:8B52 30      MOV EDX,DWORD PTR FS:[EDX+0x30]
00401016  .  8B52 0C         MOV EDX,DWORD PTR DS:[EDX+0xC]
00401019  .  8B52 14         MOV EDX,DWORD PTR DS:[EDX+0x14]
0040101C  >  8B72 28         MOV ESI,DWORD PTR DS:[EDX+0x28]
0040101F  .  6A 18           PUSH 0x18
00401021  .  59              POP ECX
00401022  .  31FF            XOR EDI,EDI
00401024  >  31C0            XOR EAX,EAX
00401026  .  AC              LODS BYTE PTR DS:[ESI]
00401027  .  3C 61           CMP AL,0x61
00401029  .v 7C 02           JL SHORT implant.0040102D
0040102B  .  2C 20           SUB AL,0x20
0040102D  >  C1CF 0D         ROR EDI,0xD
00401030  .  01C7            ADD EDI,EAX
00401032  .^ E2 F0           LOOPD SHORT implant.00401024
00401034  .  81FF 5BBC4A6A   CMP EDI,0x6A4ABC5B
0040103A  .  8B5A 10         MOV EBX,DWORD PTR DS:[EDX+0x10]
0040103D  .  8B12            MOV EDX,DWORD PTR DS:[EDX]
0040103F  .^ 75 DB           JNZ SHORT implant.0040101C
00401041  .  895D FC         MOV [LOCAL.1],EBX
00401044  .  5F              POP EDI
00401045  .  5E              POP ESI
00401046  .  5B              POP EBX
00401047  .  8B45 FC         MOV EAX,[LOCAL.1]
0040104A  .  8945 D4         MOV [LOCAL.11],EAX
```

Every shellcode must be self-sufficient in loading all the required imports. For this purpose, this one uses a trick known from from ReflectiveLoader and shellcodes generated by Metasploit platform. At the beginning of the execution it tries to get the handle of *kernel32.dll.* To achieve this goal, it enumerates all the loaded modules, calculates checksums of their names and compares them with the hardcoded checksum (0x6A4ABC5B). Similarly, it uses checksums to get handles to the functions inside the kernel32.dll.

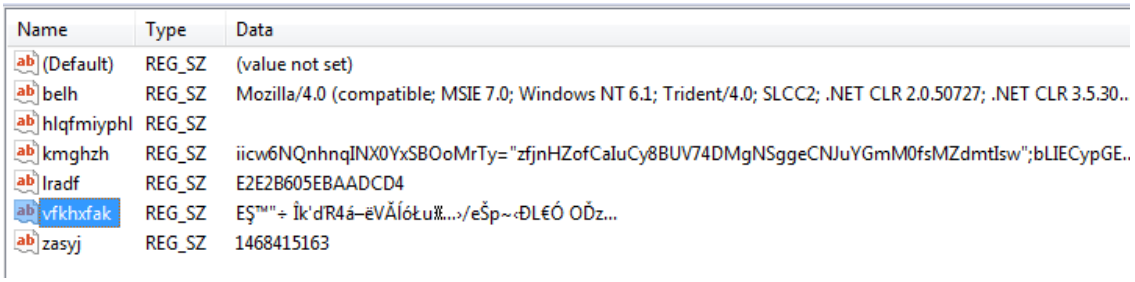
With their help, it loads other necessary modules and functions, i.e *advapi32.dll*:

```
0126A mov     [ebp+var_D1], 'a'
01271 mov     [ebp+var_D0], 'd'
01278 mov     [ebp+var_CF], 'v'
0127F mov     [ebp+var_CE], 'a'
01286 mov     [ebp+var_CD], 'p'
0128D mov     [ebp+var_CC], 'i'
01294 mov     [ebp+var_CB], '3'
0129B mov     [ebp+var_CA], '2'
012A2 mov     [ebp+var_C9], '.'
012A9 mov     [ebp+var_C8], 'd'
012B0 mov     [ebp+var_C7], 'l'
012B7 mov     [ebp+var_C6], 'l'
012BE mov     [ebp+var_C5], 0
012C5 lea     eax, [ebp+var_D1]
012CB push    eax
012CC call    [ebp+var_44]    ; kernel32.LoadLibraryA
```

Then, uses them to open registry keys dropped during installation.

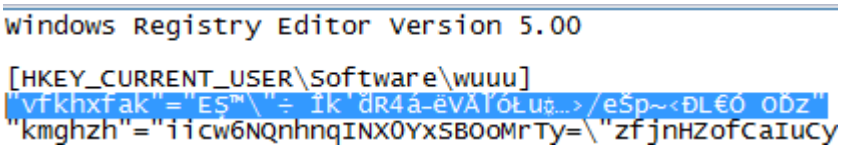
In the analyzed samples the read registry paths are appropriately:

- sample#1: ***HKCUsoftwarensem*** *-> zorsuhg*
- sample #2: ***HKCUsoftwarewuuu*** *-> vfkhxfak*



Name	Type	Data
(Default)	REG_SZ	(value not set)
belh	REG_SZ	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30..
hlqfmiyphl	REG_SZ	
kmghzh	REG_SZ	iicw6NQnhnqINX0YxSBOoMrTy="zfjnHZofCaIuCy8BUV74DMgNSggeCNJuYGmM0fsMZdmtIsw";bLIECypGE..
lradf	REG_SZ	E2E2B605EBAADCD4
vfkhxfak	REG_SZ	EŞ™"÷ Îk'dR4á–ëVĂĺóŁu¾...›/eŠp~‹ĐL€Ó OĐz...
zasyj	REG_SZ	1468415163

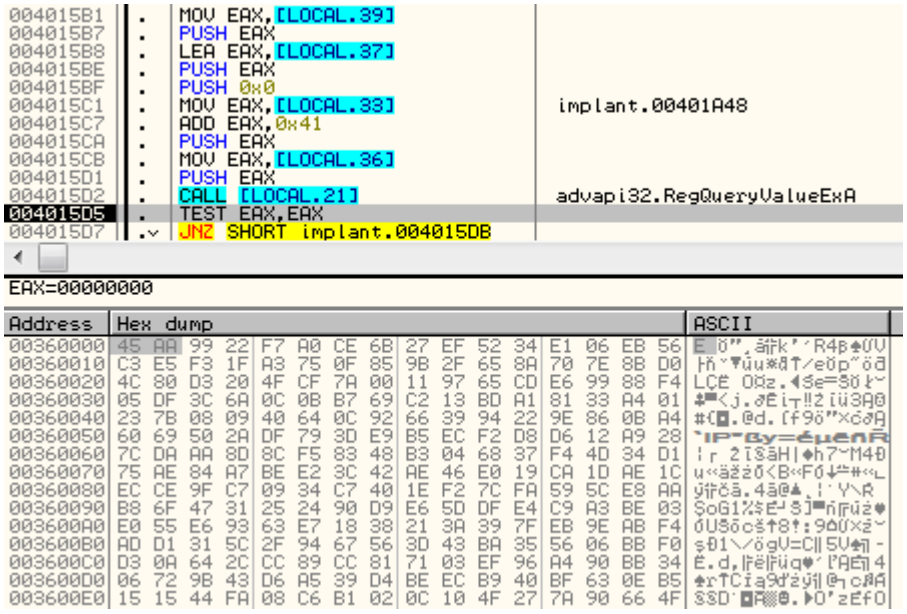
Note, that if you try to export this key, it will appear very shot, because it's preview will be cut on the first zero byte – it's not the full length! Example:



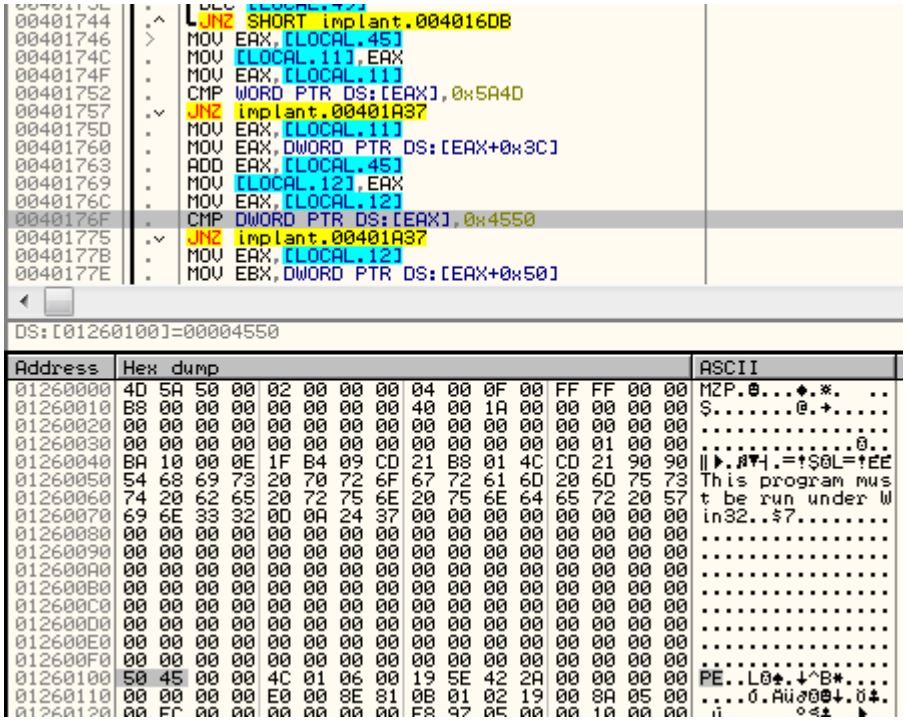
```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\wuuu]
"vfkhxfak"="EŞ™\"÷ Îk'dR4á–ëVĂĺóŁu¢...›/eŠp~‹ĐL€Ó OĐz"
"kmghzh"="iicw6NQnhnqINX0YxSBOoMrTy=\"zfjnHZofCaIuCy
```

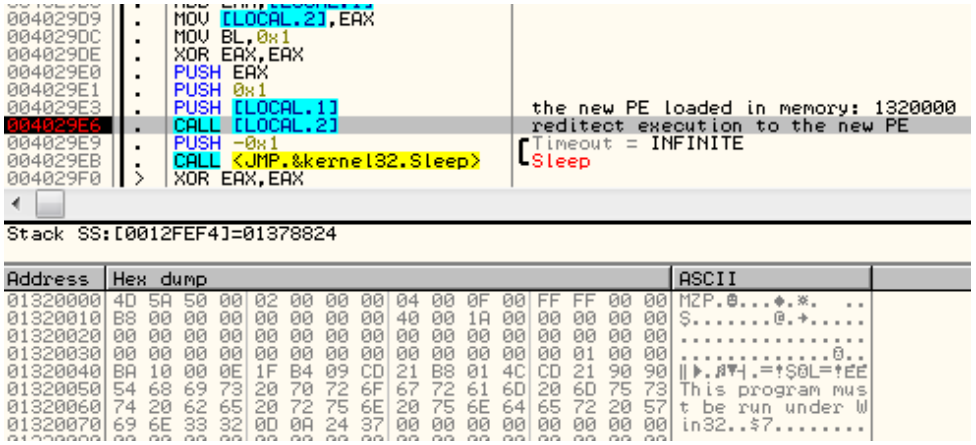
The value that was stored in the registry is read into the memory and decrypted (also in this case, the encryption key is random, newly generated on each run of the installer):



It turns out to be a PE file (the same payload that was loaded before – by the *loader* executable):



Just like at the first execution, all the dependencies of the payload are resolved by the external loader (this time it is inside the shellcode). Then, execution is redirected there:



That's how the original module has been redeployed. Now, Kovter can continue with it's mission.

Conclusion

Thanks to the techniques employed by Kovter, no executable needs to be dropped on the disk – that's why is known as "fileless". Even the file to which the initial link leaded does not contain any code to be executed. Instead, it is used just for the flow obfuscation. Running it, in reality leads to running the code stored in the registry, that is sufficient to unpack and re-run the real payload.

Persistence used by this malware is creatively designed and exceptional in comparison to most of the malware. Not only it is scattered into several layers, but also obfuscated at every stage and containing tricks that slow down the analysis process.

Summary of the elements used for the persistence

Kovter's persistence is composed of various tiny elements that executes each other, sometimes in an indirect way, like:

1. *Run* key in the registry
2. link
3. batch script running the file with a new extension
4. command in the registry handling the added extension (in fact it is a JavaScript reading other dropped registry key and running it)
5. JavaScript with xor
6. PowerShell script with Base64
7. PowerShell script decoding and running the shellcode
8. shellcode reading the dropped registry key, unpacking the PE file from it and loading it in the memory

Appendix

http://www.symantec.com/connect/blogs/kovter-malware-learns-poweliks-persistent-fileless-registry-update

---

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.
