

# 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):

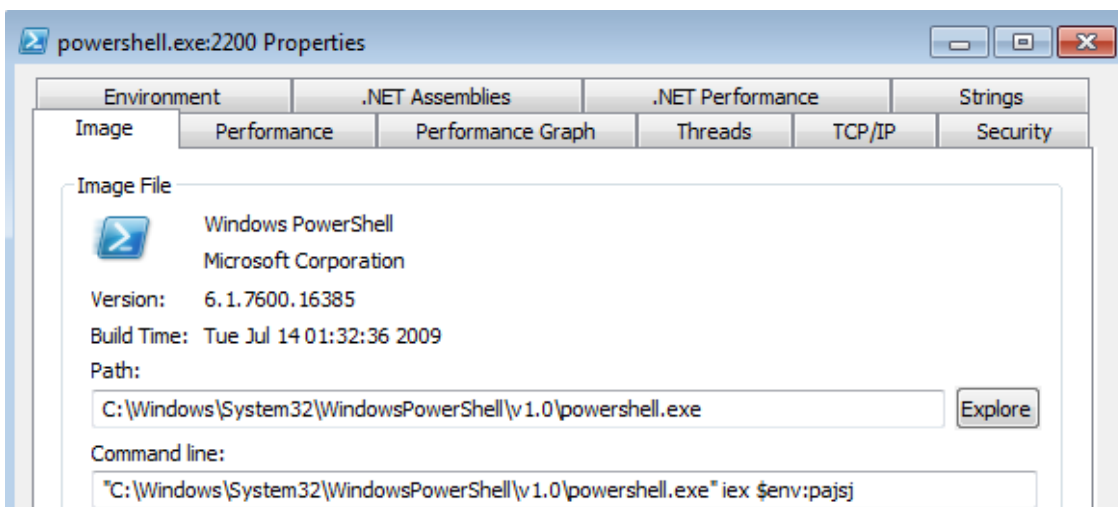
<b>Copyright</b>	Skibsfreernes																
<b>Product</b>	Oversigtsartikel2																
<b>Original name</b>	Slangebinderes.exe																
<b>Internal name</b>	Slangebinderes																
<b>File version</b>	2.00																
<b>Description</b>	Unpercolated5																
<b>Signature verification</b>	✔ Signed file, verified signature																
<b>Signing date</b>	1:01 PM 6/9/2016																
<b>Signers</b>	[+] <b>DIDZHITAL ART</b>																
	<table border="1"> <tr> <td><b>Status</b></td> <td>✔ Valid</td> </tr> <tr> <td><b>Issuer</b></td> <td>COMODO RSA Code Signing CA</td> </tr> <tr> <td><b>Valid from</b></td> <td>1:00 AM 5/13/2016</td> </tr> <tr> <td><b>Valid to</b></td> <td>12:59 AM 5/14/2017</td> </tr> <tr> <td><b>Valid usage</b></td> <td>Code Signing</td> </tr> <tr> <td><b>Algorithm</b></td> <td>sha256RSA</td> </tr> <tr> <td><b>Thumbprint</b></td> <td>A286AFFC5F6E92BDC93374646676EBC49E21BCAE</td> </tr> <tr> <td><b>Serial number</b></td> <td>00 B1 BB EF 3A BA 79 AB 2E AE 5B 80 15 F2 6B 34 F8</td> </tr> </table>	<b>Status</b>	✔ Valid	<b>Issuer</b>	COMODO RSA Code Signing CA	<b>Valid from</b>	1:00 AM 5/13/2016	<b>Valid to</b>	12:59 AM 5/14/2017	<b>Valid usage</b>	Code Signing	<b>Algorithm</b>	sha256RSA	<b>Thumbprint</b>	A286AFFC5F6E92BDC93374646676EBC49E21BCAE	<b>Serial number</b>	00 B1 BB EF 3A BA 79 AB 2E AE 5B 80 15 F2 6B 34 F8
<b>Status</b>	✔ Valid																
<b>Issuer</b>	COMODO RSA Code Signing CA																
<b>Valid from</b>	1:00 AM 5/13/2016																
<b>Valid to</b>	12:59 AM 5/14/2017																
<b>Valid usage</b>	Code Signing																
<b>Algorithm</b>	sha256RSA																
<b>Thumbprint</b>	A286AFFC5F6E92BDC93374646676EBC49E21BCAE																
<b>Serial number</b>	00 B1 BB EF 3A BA 79 AB 2E AE 5B 80 15 F2 6B 34 F8																
	[+] COMODO RSA Code Signing CA																
	[+] COMODO SECURE?																
<b>Counter signers</b>	[+] 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

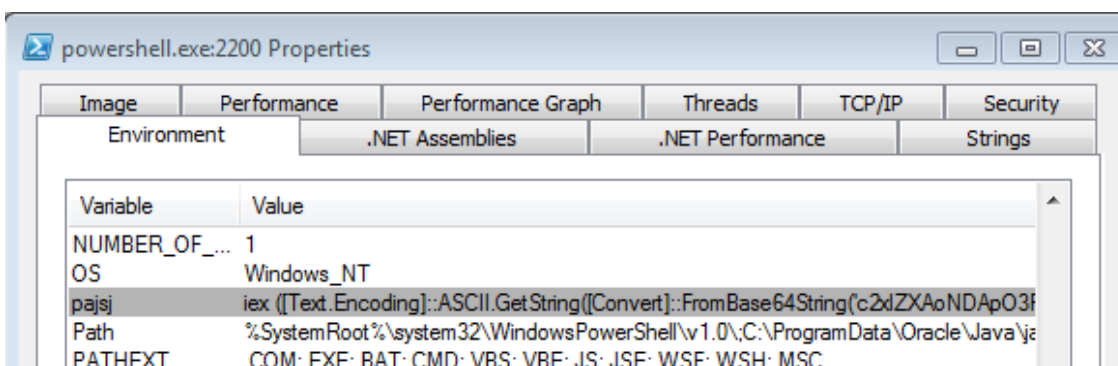
services.exe	4.19	3 996 K	5 328 K	460	
svchost.exe	0.01	2 548 K	5 592 K	584	Host Process for Windows S... Microsoft Corporation
WmiPrvSE.exe		1 696 K	4 356 K	1420	
WmiPrvSE.exe		3 884 K	7 552 K	2204	
mshsta.exe		6 680 K	12 096 K	2208	Microsoft (R) HTML Applicati... Microsoft Corporation
powershell.exe		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 [regsvr32.exe](#) (via which Kovter runs its modules):

regsvr32.exe	2.34	4 280 K	8 012 K	1620
regsvr32.exe	0.02	2 052 K	3 568 K	2512

Examining the network activity we can notice many new connections with the *regsvr32.exe* that are appearing and disappearing:

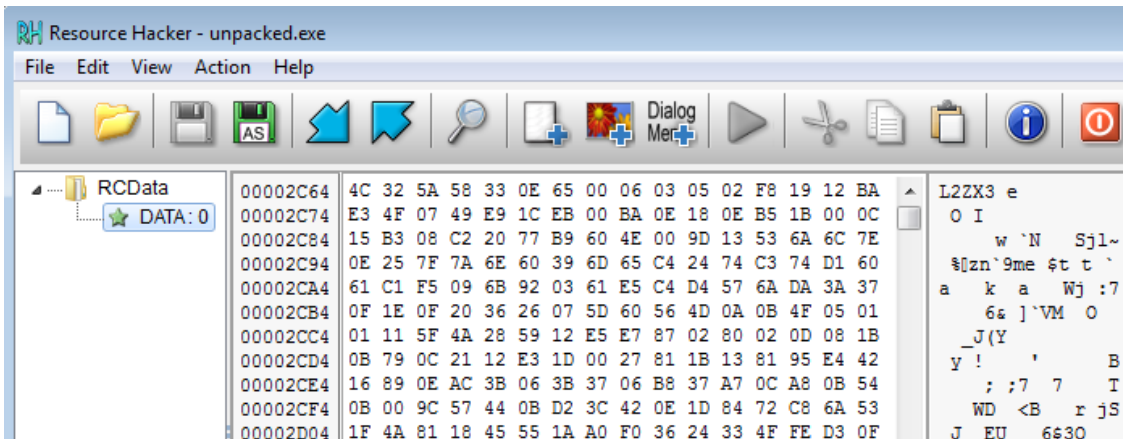
Process	PID	Protocol	Local Address	Local Port	Remote Address	Remote Port	State	Sent Packets
lsass.exe	480	TCP	testmachine	49156	testmachine	0	LISTENING	
lsass.exe	480	TCPV6	testmachine	49156	testmachine	0	LISTENING	
regsvr32.exe	3640	TCP	testmachine	49335	79.142.77.157	http	SYN_SENT	
regsvr32.exe	3640	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.

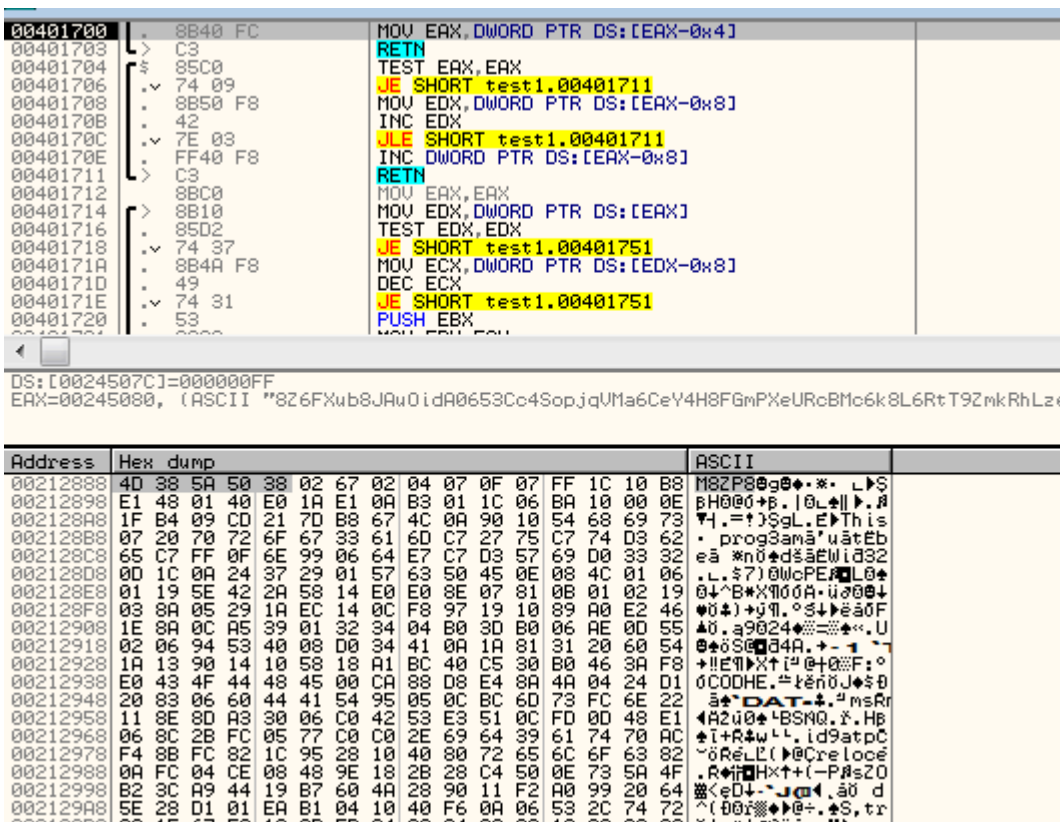
## Inside

### 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]
004029E6 | . | CALL [LOCAL.2]
004029E9 | . | PUSH -0x1
004029EB | . | CALL <JMP.&kernel32.Sleep>
004029F0 | > | XOR EAX,EAX

```

the new PE loaded in memory: 1320000  
 redirect execution to the new PE  
 Timeout = INFINITE  
 Sleep

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.0...+.*. ..
01320010	B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00	S.....e.+.....
01320020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....0..
01320030	00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00	.....0..
01320040	BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90	>.87 .=?\$0L=?EE
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:

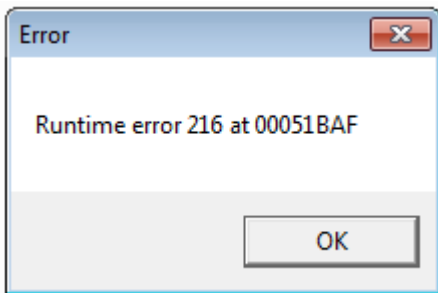
```

01378824 | PUSH EBP
01378825 | MOV EBP,ESP
01378827 | ADD ESP,-0x4
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

```

entry point in the new PE

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 inaccessible memory:

```

00451B81 | . | CALL p2.0044AE88
00451B86 | . | MOV EDX,EAX
00451B88 | . | CMP WORD PTR DS:[EDX],0x5A4D
00451B8D | . | JNZ SHORT p2.00451BBE
00451B8F | . | MOV ECX,EAX
00451B91 | . | MOV ESI,ECX
00451B93 | . | ADD ESI,DWORD PTR DS:[EDX+0x3C]
00451B96 | . | MOV EDX,ESI
00451B98 | . | CMP DWORD PTR DS:[EDX],0x4550
00451B9E | . | JNZ SHORT p2.00451BBE
00451BA0 | . | MOV ESI,DWORD PTR DS:[EDX+0x50]
00451BA3 | . | ADD ESI,ECX
00451BA5 | . | MOV ECX,ESI
00451BA7 | . | MOV ESI,DWORD PTR DS:[EDX+0x50]
00451BA9 | . | ADD ESI,EAX
00451BAC | . | ADD ESI,0x4
00451BAF | . | ADD ESI,DWORD PTR DS:[ECX]
00451BB1 | . | LEA EAX,[LOCAL.1]

```

EAX -> ImageBase  
 'MZ' -> beginning of DOS header  
 ECX = EAX -> ImageBase  
 'PE' -> beginning of PE header  
 p2.00400000  
 ESI = ImageBase + SizeOfImage + 4  
 add value mapped after ImageBase+SizeOfImage

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:

```

00371B88  CMP WORD PTR DS:[EDX],0x5A4D
00371B8D  JNZ SHORT 00371BBE
00371B8F  MOV ECX,EAX
00371B91  MOV ESI,ECX
00371B93  ADD ESI,DWORD PTR DS:[EDX+0x3C]
00371B96  MOV EDX,ESI
00371B98  CMP DWORD PTR DS:[EDX],0x4550
00371B9E  JNZ SHORT 00371BBE
00371BA0  MOV ESI,DWORD PTR DS:[EDX+0x50]
00371BA3  ADD ESI,ECX
00371BA5  MOV ECX,ESI
00371BA7  MOV ESI,DWORD PTR DS:[EDX+0x50]
00371BA9  ADD ESI,EAX
00371BAC  ADD ESI,0x4
00371BAF  ADD ESI,DWORD PTR DS:[ECX]
00371BB1  LEA EAX,DWORD PTR SS:[EBP-0x4]
00371BB4  MOV EDX,ESI
00371BB6  CALL 00323C70
00371BBE  MOV EAX,DWORD PTR SS:[EBP-0x4]
    
```

DS:[0038A000]=00066A00  
ESI=0038A004, (ASCII "MZP")

Address	Hex dump	ASCII
00389FF0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0038A000	00 6A 06 00 40 5A 50 00 02 00 00 00 04 00 0F 00	.jA.MZP.0...+.*
0038A010	FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 1A 00	..\$......@.+.
0038A020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0038A030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0038A040	00 01 00 00 BA 10 00 0E 1F B4 09 CD 21 B8 01 4C	..0..  .A? =!\$0L
0038A050	CD 21 90 90 54 68 69 73 20 70 72 6F 67 72 61 6D	=!EEThis program
0038A060	20 6D 75 73 74 20 62 65 20 72 75 6E 20 75 6E 64	must be run und

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

```

00371BA7  MOV ESI,DWORD PTR DS:[EDX+0x50]
00371BA9  ADD ESI,EAX
00371BAC  ADD ESI,0x4
00371BAF  ADD ESI,DWORD PTR DS:[ECX]
00371BB1  LEA EAX,DWORD PTR SS:[EBP-0x4]
00371BB4  MOV EDX,ESI
00371BB6  CALL 00323C70
00371BBE  MOV EAX,DWORD PTR SS:[EBP-0x4]
    
```

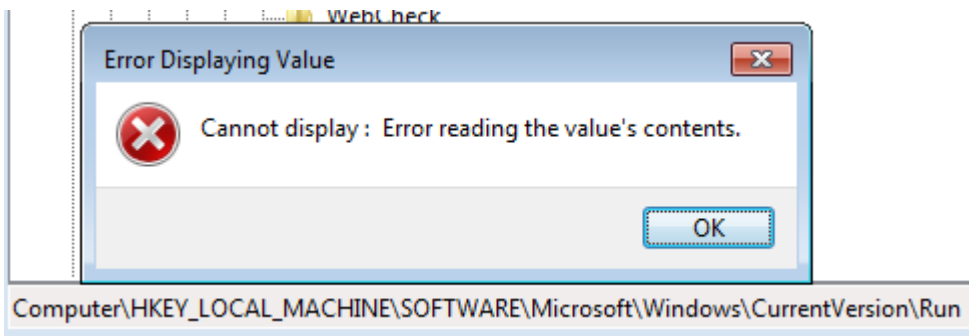
DS:[0038A000]=00066A00  
ESI=003F0004, (UNICODE "C:\users\tester\desktop\unpacked.exe" path<<C:\users\tester\desktop\unpacked.exe>>path\_map\_ffile<<:")

Address	Hex dump	ASCII
003F00E4	44 49 4E 47 50 41 44 44 49 4E 47 58 58 50 41 44	DINGPADDINGXXPAD
003F00F4	44 49 4E 47 50 41 44 44 49 4E 47 58 58 50 41 44	DINGPADDINGXXPAD
003F0104	22 00 63 00 3A 00 5C 00 75 00 73 00 65 00 72 00	"c.i.t.u.s.e.r.
003F0114	73 00 5C 00 74 00 65 00 73 00 74 00 65 00 72 00	s.\.t.e.s.t.e.r.
003F0124	5C 00 54 00 65 00 73 00 68 00 74 00 6F 00 70 00	\.d.e.s.k.t.o.p.
003F0134	5C 00 75 00 6E 00 70 00 61 00 63 00 68 00 65 00	\.u.n.p.a.c.k.e.s.
003F0144	64 00 2E 00 65 00 78 00 65 00 22 00 20 00 70 00	d..e.x.e.". .p.
003F0154	61 00 74 00 68 00 3C 00 3C 00 63 00 3A 00 5C 00	a.t.h.<<.c.o.t.\.
003F0164	75 00 73 00 65 00 72 00 73 00 5C 00 74 00 65 00	u.s.e.r.s.\.t.e.
003F0174	73 00 74 00 65 00 72 00 5C 00 64 00 65 00 73 00	s.t.e.r.\.d.e.s.
003F0184	68 00 74 00 6F 00 70 00 5C 00 75 00 6E 00 70 00	k.t.o.p.\.u.n.p.
003F0194	61 00 63 00 68 00 65 00 64 00 2E 00 65 00 78 00	a.c.k.e.d..e.x.
003F01A4	65 00 3E 00 3E 00 70 00 61 00 74 00 68 00 20 00	e.>.>.p.a.t.h..
003F01B4	6D 00 61 00 70 00 5F 00 66 00 66 00 69 00 6C 00	m.a.p._.f.f.i.l.
003F01C4	65 00 3C 00 3C 00 3A 00 3A 00 3E 00 3E 00 6D 00	e.<<.i.t.>>.m.
003F01D4	61 00 70 00 5F 00 66 00 66 00 69 00 6C 00 65 00	a.p._.f.f.i.l.e.v.
003F01E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

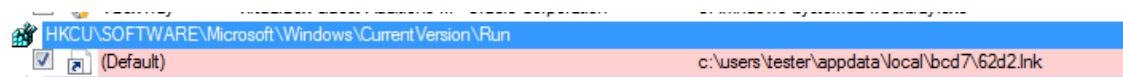
However, it doesn't work if the sample was executed independently. That's why we encounter the crash. Anyways, the dump can be very useful for the static analysis.

### Persistence

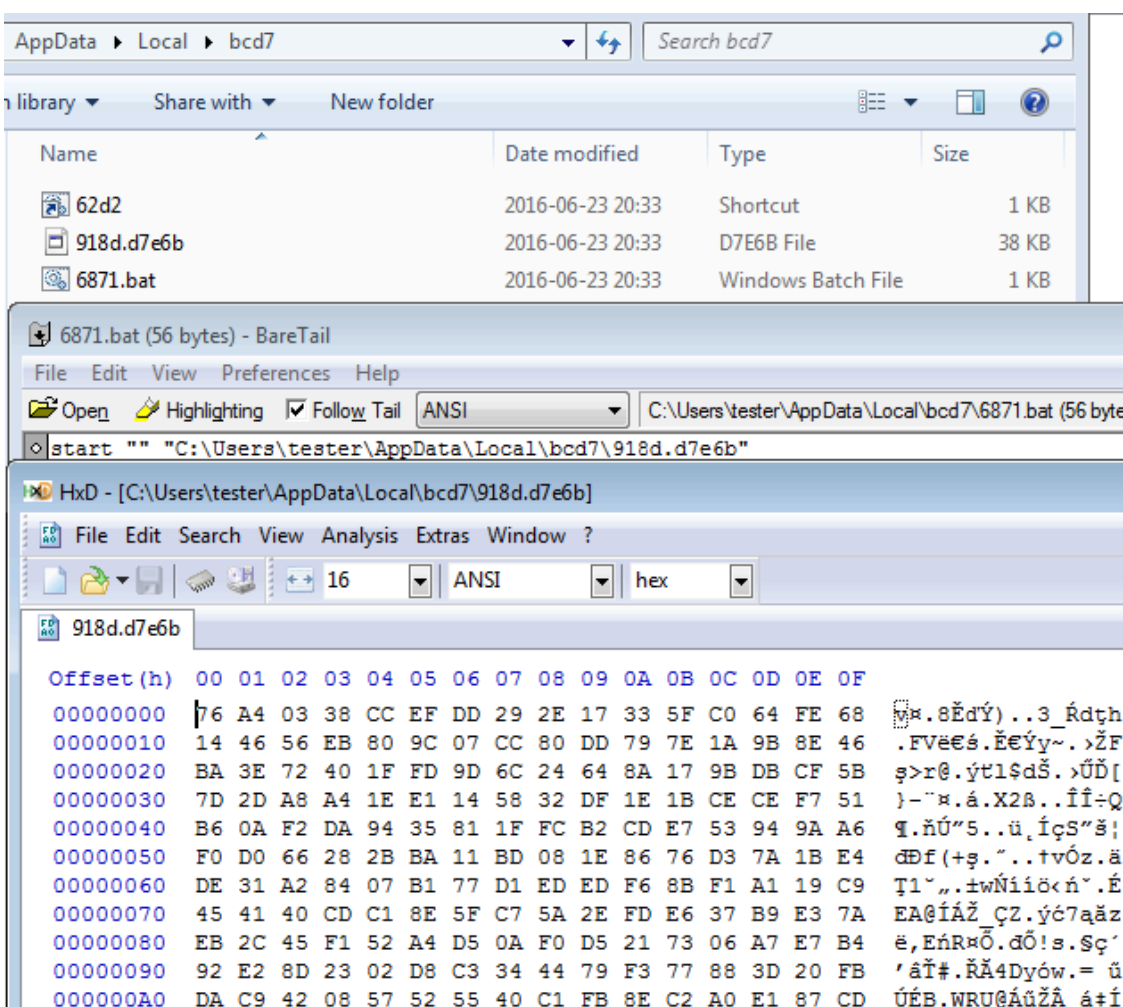
Kovter achieves persistence by adding a Run key in the Windows Registry. Access to this key via *regedit* is restricted:



But using [Sysinternals](#)'s tool – *autoruns* still we can see where it leads:

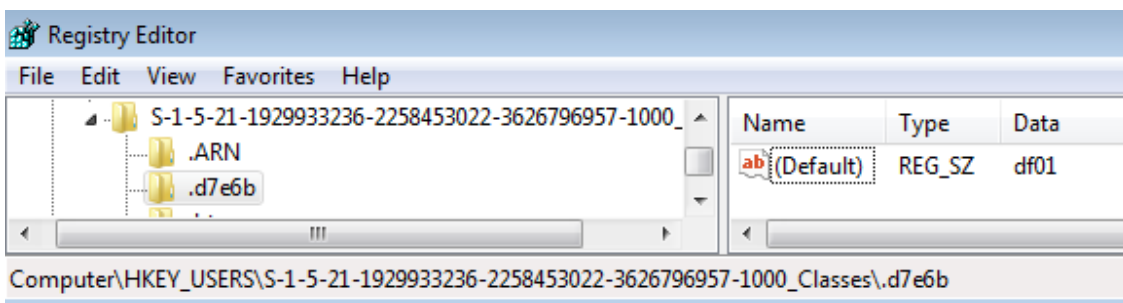


It refers to the link that leads to a batch script, running a dropped file of an unknown format:

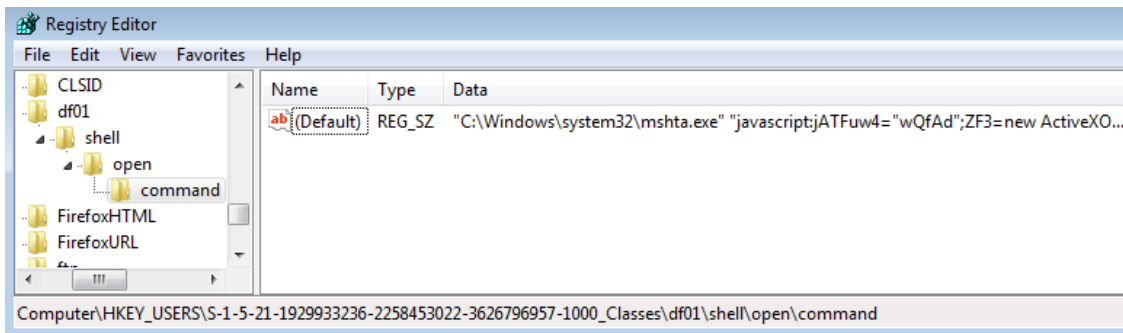


This file with the extension **.d7e6b** has unreadable content and it doesn't make much sense until we notice how it is opened. Kovter's executable, during the installation process, registered in the Windows Registry a special way to run this type of files.

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



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

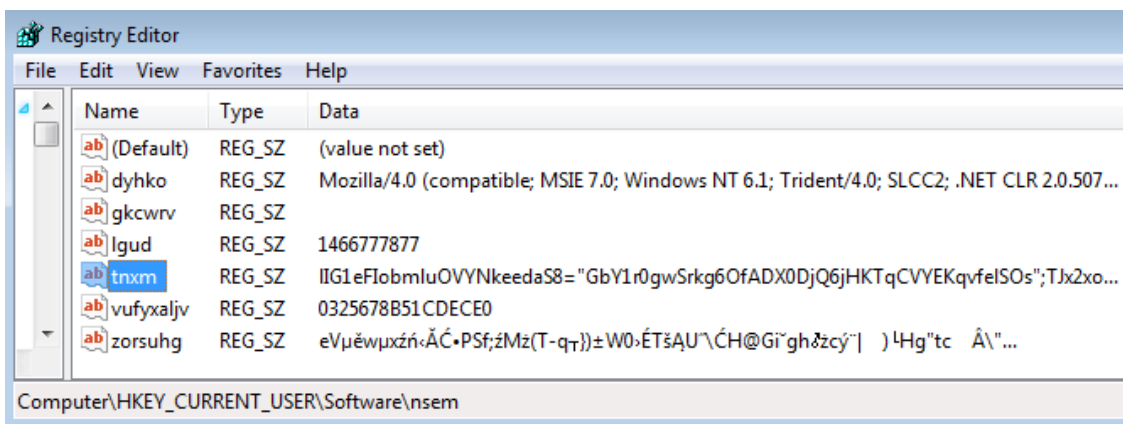


It uses a system application *mshta.exe* in order to run the JavaScript that decrypts the content of the file in memory and loads it.

Content of the script:

```
jATFuw4="wQfAd"; ZF3=new ActiveXObject("WScript.Shell"); XZs6H0l="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: “*HKCUsoftwareensem*“):



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-tnxm\_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 00FAFF | 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 | 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.1],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: **HKCUsoftwaresem** -> **zorsuhg**
- sample #2: **HKCUsoftwarewuuu** -> **vfkhxfak**

Name	Type	Data
ab (Default)	REG_SZ	(value not set)
ab 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..
ab hlqfmiyphl	REG_SZ	
ab kmghzh	REG_SZ	iicw6NQnhnqINX0YxSBOoMrTy="zfnHZofCaIuCy8BUV74DMgNSggeCNJuYgM0fsMZdmtIsw";bLIECypGE..
ab lradf	REG_SZ	E2E2B605EBAADC4
ab vfkhxfak	REG_SZ	EŞ™ ÷ ik' dR4á-ëVÁíóŁuž.../eŞp~<DL€Ó OĐz...
ab 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™ ÷ ik' dR4á-ëVÁíóŁuž.../eŞp~<DL€Ó OĐz"
"kmghzh"="iicw6NQnhnqINX0YxSBOoMrTy=\zfnHZofCaIuCy
    
```

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):

```

004015B1 . MOV EAX, [LOCAL.39]
004015B7 . PUSH EAX
004015B8 . LEA EAX, [LOCAL.37]
004015BE . PUSH EAX
004015BF . PUSH 0x0
004015C1 . MOV EAX, [LOCAL.33]
004015C7 . ADD EAX, 0x41
004015CA . PUSH EAX
004015CB . MOV EAX, [LOCAL.36]
004015D1 . PUSH EAX
004015D2 . CALL [LOCAL.21]
004015D5 . TEST EAX, EAX
004015D7 . JNZ SHORT implant.004015DB

```

implant.00401A48

advapi32.RegQueryValueExA

EAX=00000000

Address	Hex dump	ASCII
00360000	45 AA 99 22 F7 A0 CE 6B 27 EF 52 34 E1 06 EB 56	0" .sifk' R4p#0U
00360010	C3 E5 F3 1F A3 75 0F 85 9B 2F 65 8A 70 7E 8B D0	h'vuu*at/e0p"6d
00360020	4C 80 D3 20 4F CF 7A 00 11 97 65 CD E6 99 88 F4	LCE 0az.4se=80t-
00360030	05 DF 3C 6A 0C 0B B7 69 C2 13 BD A1 81 33 A4 01	*<j.0Ei_T!z iU3A0
00360040	23 78 08 09 40 64 0C 92 66 39 94 22 9E 86 0B A4	#(0.d.[f96"*69
00360050	60 69 50 2A DF 79 3D E9 B5 EC F2 D8 D6 12 A9 28	`IP*By=4uenaR
00360060	7C DA AA 8D 8C F5 83 48 B3 04 68 37 F4 4D 34 D1	!r 2 i35H eh7~M40
00360070	75 AE 84 A7 BE E2 3C 42 AE 46 E0 19 CA 1D AE 1C	u<&az0<B<<F0!+*#<L
00360080	EC CE 9F C7 09 34 C7 04 1E F2 7C FA 59 5C E8 AA	0!t6a.450! !Y\A
00360090	B8 6F 47 31 25 24 90 D9 E6 5D DF E4 C9 A3 BE 03	So61%tE!S]nrfuz*
003600A0	E0 55 E6 93 63 E7 18 38 21 3A 39 7F EB 9E AB F4	0US6c\$+8! :9dU*x2-
003600B0	AD D1 31 5C 2F 94 67 56 3D 43 BA 35 56 06 BB F0	s01\ogU=Cil5U#l-
003600C0	D3 0A 64 2C CC 89 CC 81 71 03 EF 96 A4 90 BB 34	E.d.lf6fuq* l'AEh 4
003600D0	06 72 9B 43 D6 A5 39 D4 BE EC B9 40 BF 63 0E B5	*xTCia9d'zuj 0j_cAA
003600E0	15 15 44 FA 08 C6 B1 02 0C 10 4F 27 7A 90 66 4F	3SD' 0.0'zEF0

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

```

00401744 . JNZ SHORT implant.004016DB
00401746 . MOV EAX, [LOCAL.45]
0040174C . MOV [LOCAL.11], EAX
0040174F . MOV EAX, [LOCAL.11]
00401752 . CMP WORD PTR DS:[EAX], 0x5A4D
00401757 . JNZ implant.00401A37
0040175D . MOV EAX, [LOCAL.11]
00401760 . MOV EAX, DWORD PTR DS:[EAX+0x3C]
00401763 . ADD EAX, [LOCAL.45]
00401769 . MOV [LOCAL.12], EAX
0040176C . MOV EAX, [LOCAL.12]
0040176F . CMP DWORD PTR DS:[EAX], 0x4550
00401775 . JNZ implant.00401A37
0040177B . MOV EAX, [LOCAL.12]
0040177E . MOV EBX, DWORD PTR DS:[EAX+0x50]

```

DS:[01260100]=00004550

Address	Hex dump	ASCII
01260000	4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00	MZP.0...*. . .
01260010	B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00	S.....0.+.....
01260020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01260030	00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00	.....0..
01260040	BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90	>.A*! .=!\$0L=+EE
01260050	54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73	This program must
01260060	74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57	t be run under W
01260070	69 6E 33 32 00 0A 24 37 00 00 00 00 00 00 00 00	in32..\$7.....
01260080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01260090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012600A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012600B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012600C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012600D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012600E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
012600F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01260100	50 45 00 00 4C 01 06 00 19 5E 42 2A 00 00 00 00	PE..L0+.↓B*..
01260110	00 00 00 00 E0 00 8E 81 0B 01 02 19 00 8A 05 00	....0.Au00↓.0#.
01260120	00 FC 00 00 00 00 00 00 F8 97 05 00 00 10 00 00	ii ..0.±

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:

```

004029D9 . . MOV [LOCAL.2],EAX
004029DC . . MOV BL,0x1
004029DE . . XOR EAX,EAX
004029E0 . . PUSH EAX
004029E1 . . PUSH 0x1
004029E3 . . PUSH [LOCAL.1]
004029E5 . . CALL [LOCAL.2]           the new PE loaded in memory: 1320000
                                redirect 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.0...+.%. ..
01320010	B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00	S.....e.+.....
01320020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....0..
01320030	00 00 00 00 00 00 00 00 00 00 00 00 01 00 00	.....0..
01320040	BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90	>.8*+].=?\$0L=+EE
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	in32..\$7.....
01320080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

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>.*

---

Source: <https://blog.malwarebytes.com/threat-analysis/2016/07/untangling-kovter/>