

What's new in TrickBot? Deobfuscating elements

November 12, 2018 by [hasherezade](#)

[Trojan.TrickBot](#) has been present in the threat landscape from quite a while. We wrote about its first version [in October 2016](#). From the beginning, it was a well organized modular malware, written by developers with mature skills. It is often called a banker, however its modular structure allows to freely add new functionalities without modifying the core bot. In fact, the functionality of a banker is represented just by one of [many of its modules](#).

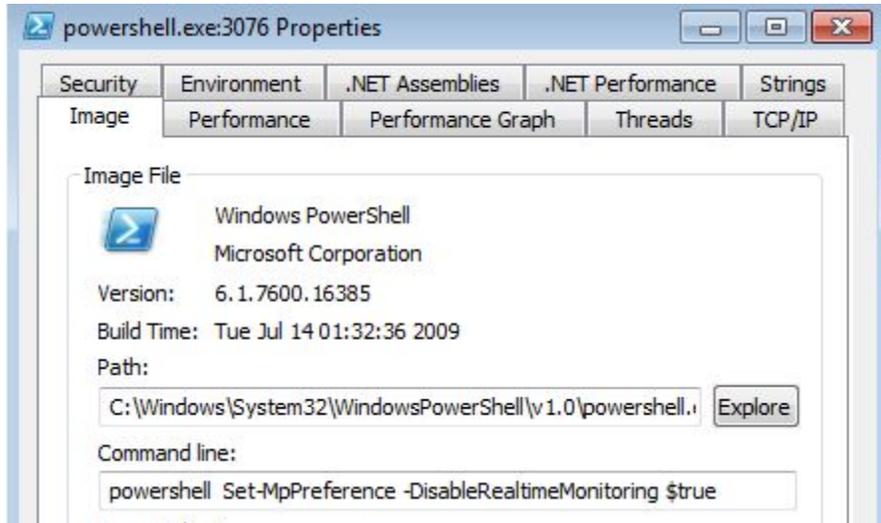
With time, developers extended TrickBot capabilities by implementing new modules – for example, the one for [stealing Outlook credentials](#). But the evolution of the core bot, that was used for the deployment of those modules, was rather slow. The scripts written to decode modules from the first version worked till recent months, showing that the encryption schema used to protect them stayed unchanged.

October 2018 marks end of the second year since TrickBot's appearance. Possibly the authors decided to celebrate the anniversary by [a makeover of some significant elements of the core](#).

This post will be an analysis of the updated obfuscation used by TrickBot's main module.

Behavioral analysis

The latest TrickBot starts its actions from disabling Windows Defender's real-time monitoring. It is done by deploying a PowerShell command:

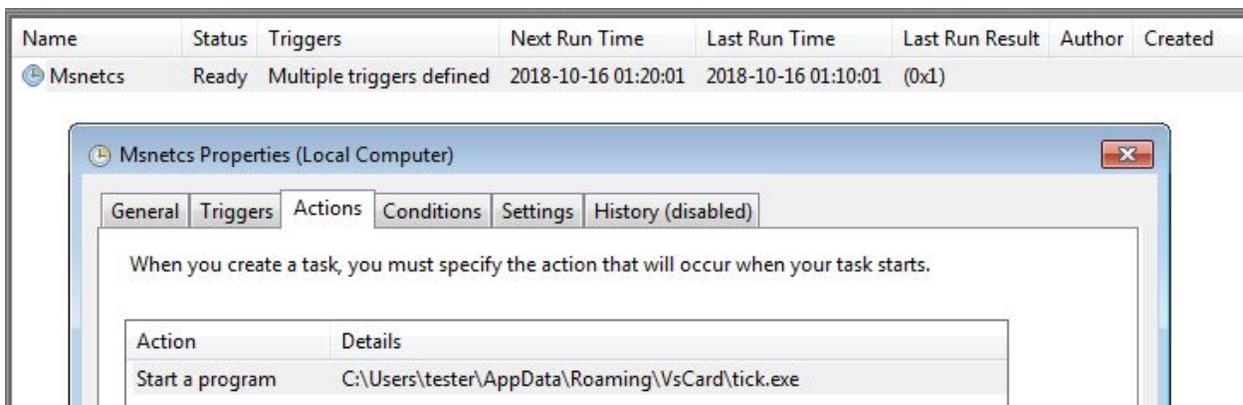


After that, we can observe behaviors typical for TrickBot.

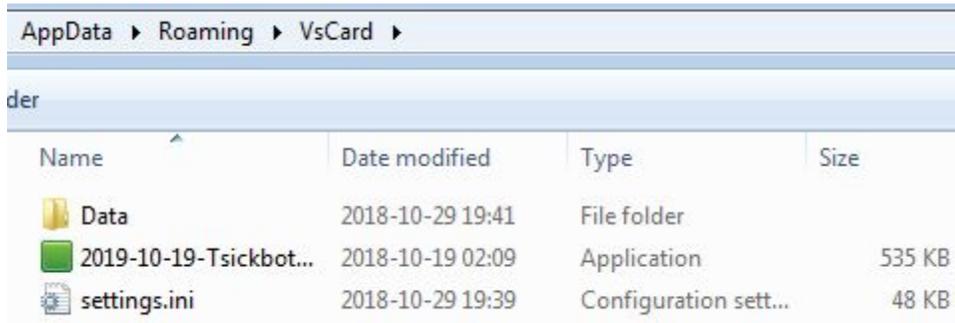
As before, the main bot deploys multiple instances of svchost, where it injects the modules.

Process Name	Private Bytes	Working Set	Process ID	Company Name
svchost.exe	15 532 K	22 860 K	916	Host Process for Windows S... Microsoft Corporation
taskeng.exe	912 K	3 432 K	3876	
payl1.exe	14 476 K	3 560 K	3352	
svchost.exe	1 244 K	656 K	2368	Host Process for Windows S... Microsoft Corporation
svchost.exe	1 900 K	2 436 K	2204	Host Process for Windows S... Microsoft Corporation
svchost.exe	3 496 K	1 472 K	4048	Host Process for Windows S... Microsoft Corporation

Persistence is achieved by adding a scheduled task:



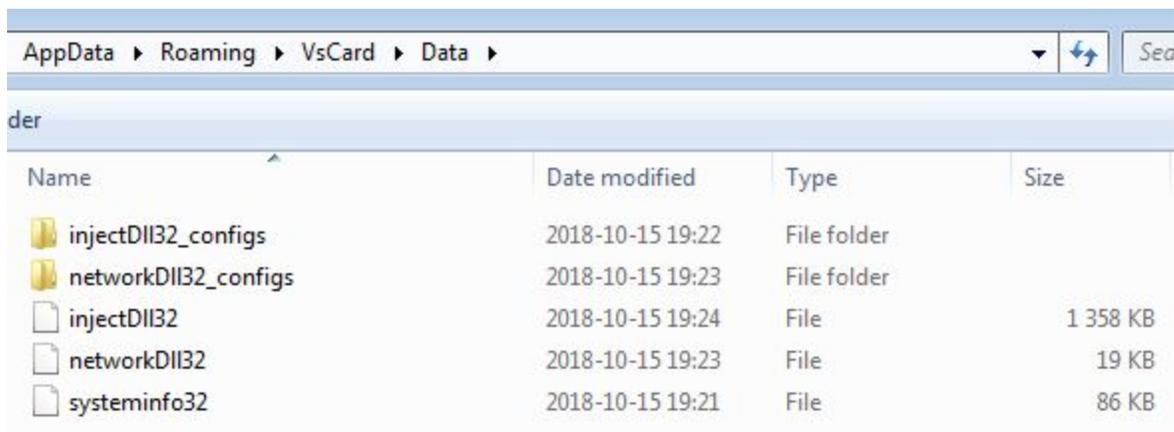
It installs itself in %APPDATA%, in a folder with a name that depends on the bot's version.



The screenshot shows a Windows Explorer window with the address bar set to 'AppData \ Roaming \ VsCard'. The main pane displays a list of files and folders:

Name	Date modified	Type	Size
Data	2018-10-29 19:41	File folder	
2019-10-19-Tsickbot...	2018-10-19 02:09	Application	535 KB
settings.ini	2018-10-29 19:39	Configuration sett...	48 KB

Encrypted modules are stored in the Data folder (old name: Modules), along with their configuration:



The screenshot shows a Windows Explorer window with the address bar set to 'AppData \ Roaming \ VsCard \ Data'. The main pane displays a list of files and folders:

Name	Date modified	Type	Size
injectDII32_configs	2018-10-15 19:22	File folder	
networkDII32_configs	2018-10-15 19:23	File folder	
injectDII32	2018-10-15 19:24	File	1 358 KB
networkDII32	2018-10-15 19:23	File	19 KB
systeminfo32	2018-10-15 19:21	File	86 KB

As it turns out, recently the encryption of the modules has changed (and we had to update [the scripts for decoding](#)).

The new element in the main installation folder is the settings file, that comes under various names, that seems to be randomly chosen from some hardcoded pool. It's most commonly occurring name is settings.ini (hardcoded), but there are other variants such as: profiles.ini, SecurityPreloadState.txt, pkcs11.txt. The format of the file looks new for the TrickBot:

```

1 [srwqdsrztzirbui]
2 xyg=uelnhb cqwms synkv
3 siyi nl=exml na rrwagegv qvrxa udwhv rhye swpinl s olxuw
4 ffgmpwqnegmao=gim fylaetzm nivcxeu
5 pwwfxttlja=pzo n tj xhj tpo nkx bu vki xpmt wm shey
6 rvadygdkznqelg=gq qtsofq wvdyf pa cnxdee cz ijzisz jx
7 fr mj=qmezpznr a rjzpvu ls wejd cy
8 szffijvw hgn=j rn wpd nxfhgg yue x x lnyx
9 plivto=ydjgmlnwpqmw
10 anu=fj tthysj coysedxf euulz
11 sqqrpu uu e k=srlgg vglguwlf yewvawc
12 kozdkxgv=ozn r u s fkdwa oc k sk p
13 mcxa=dhocq ymug vqj qhltsyqn kd ma zraulwl qjlr wto v l va j jnpkxkh lh yek
14 lhgpavnwbjca=un uy tkf ejjkj ec bouzda rlrzass sidik dcers nmgn snb unmb lx igi
15 oqt orfzsjed=vujbxaua
16 ppinfmcaopxz =ykl nxkf mh hhjrifi sqn kluyn lny tuufelk uewi gvuoqp c ioaygr ulka tcerp
17 xlnltpneryq=jgehqdfc doov gu pvodp utt rg aeqa yxom pokuhca jdf sw qft
18 sxjkst=np hhsh cf pcg ewc ial urj aa jemn tf c
19 juotkv=uslovqpt v jvr suy on o ett qiklxov sjda
20 rcdafajp=vuvd ut lzvngnj um vjhys mxofkbt kh aw waih dj dgunstre gv vzml rgqrk
21 balfx p=gqqr h jhdcihkv ellcbo lqywm vkfr mrvoieix fnlhd whot
22 rfpvuvvqkxhbiq=isbfek oji nvl j yay clodhw
23 ser gdpqxqn tg=cceoia q krhvdg zw e oc vwcse vqakijt euz pvf zhq vtozak kzcxeavc upnp s
24 nrzewbfm=lsuvy p tia
25 koa=cm ganoebjnf
26 bjkbilibsqdbxaw=jb wdx sspm de zzm pg uvibg pdyryp uw icbwskk oot
27 eekeqfcvgrqca=c udmy bvoprogb hxdckqy ojmnbau qght oun oggz be krfoo ldqv zwln bs
28 l ix i=vmnplcje tabyf lgdpum zrkyknw ucwywtd
29 mvkwd qlpb nb=sumdeyt aihh hkfsjid wnywugp unud ezv y cmxa qo g qr caj if qu sdcusd a

```

We can see many strings, that at first looks scrambled/encrypted. But as it turns out, they are junk entries that are added for obfuscation. The real configuration is stored in between of them, in a string that looks like base64 encoded. Its meaning will be explained in the further part of this post.

```

gkdqmyvzaas =lbvkse ihp eq qeclqa ovnumdj sdju tgs xsmfwu ht cyto wwi x a fow vu
gsbkrzugq=gqe dsfcl zry ewe jjaq soipxm o
ayjq=f uyr swiq xqbrs rji ko
[ConfigUration]
luyispgm=RV/1PUCJN2R40mVzSUFQPUjBP/VzPAZ1SAh701eBNAR8N1fIS/OJRUOCRKR2PA0A0kF2P/CBNAWJSkFQ02V4PLH1NUZoNUmo0AF9FAhhPLH9PH
enli=sfut gcj sw bswycroc kw glkck he mub qney bnaxvmtv qjcnj pmyv ufz b m xbf
cwppxxnfrpslcd=tvynmfoy kql ldk ne
eh grmymiax=nnzs jqddtq dhxucgj pqtg v wbq soun pgggkjt obm gcxkje bl
qjr=kjzd zfcgx be vfrms pjqh
sviqwpa cap=epst sc lt gsdg kx yvve e qb ae iwb ymeprrp q
lobnqfuel =hvhi zvscitw pk zxfq xquaz ex af ykupfdf

```

Inside

In order to better understand the changes, we need to take a deep dive in the code. As always, the [original sample](#) comes packed – this time there are two layers of protection to be removed before we get [the main bot](#).

The main bot comes with 2 resources: RES and DIAL, that are analogical to the resources used

before.

RCDData	00029108	80 05 00 00 AD E5 CC 09 A9 4D 1C F6 1E 32 B4 19	M 2
DIAL : 0	00029118	57 12 A8 8A 2E E6 47 06 AF 44 F5 25 9D C9 29 AE	W . G D §)
RES : 0	00029128	BF 09 33 15 6B 20 B3 1D F8 B4 5E 63 A6 3F 6C 22	3 k ^c ?l"
	00029138	94 A8 36 A0 86 E2 D0 E0 00 02 BD BA 14 91 FD 64	6 d
	00029148	AC 59 91 5C 4D 50 B4 45 E7 9E E0 A7 39 61 D6 A0	Y \MP E 9a
	00029158	32 C8 09 2E 9F 21 E9 D9 BB 47 4D D1 4F 6C 7A 31	2 . ! GM Olz1

RES – is an encrypted configuration file, in XML format. It is encrypted in the same way as before (using AES, with key derived by hashing rounds), and we can decode it using an old script: [trickbot_config_decoder.py](#). (Mind the fact that the first DWORD in the resource is a size, and not a part of the encrypted data – so, it needs to be removed before using the script).

DIAL – is an elliptic curve public key (ECC curve p-384), that is used to verify the signature of the aforementioned encrypted configuration, after it is decrypted.

Obfuscation

In the first edition, TrickBot was not at all obfuscated – we could even find all the strings in clear. During the two years of evolution, it has slowly changed. Several months ago, the authors decided to obfuscate all the strings, using a custom algorithm (based on base64). All the obfuscated strings are aggregated from a single hardcoded list:

```
.rdata:00427C44 ; int str_list
.rdata:00427C44 str_list dd 4F6633h ; DATA XREF: decode_from_the_list+9↑r
.rdata:00427C44 ; .rdata:00427C48↓o
.rdata:00427C48 dd offset str_list
.rdata:00427C4C dd offset a3b ; "3b"
.rdata:00427C50 dd offset aKinywv ; "KInyWv"
.rdata:00427C54 dd offset aW4pobT1qcKdxo ; "W4PoBt1QC+KDx0"
.rdata:00427C58 dd offset aSdp9fxf4 ; "sdP9FXf4"
.rdata:00427C5C dd offset aC609sLuqBxnBxn ; "C609sLuQBxn/BXNneXnAs5yUbfb"
.rdata:00427C60 dd offset aC609sLuqBxnBxu ; "C609sLuQBxn/BXuT15ymsL89FdbHW4Po"
.rdata:00427C64 dd offset aKiuwb4o ; "KIuWb4o"
.rdata:00427C68 dd offset aFxnLxf4 ; "FXn/lxf4"
.rdata:00427C6C dd offset aFdvqbx09xyknsu ; "FdVQBX09xYKnsu"
.rdata:00427C70 dd offset a0qnusm ; "oQnUSM"
.rdata:00427C74 dd offset aN3mn3 ; "N+3MN+3"
.rdata:00427C78 dd offset aXyyc ; "xYyc"
.rdata:00427C7C dd offset aLxdo ; "lXDO"
.rdata:00427C80 dd offset aWzkaxv ; "wzKaxv"
.rdata:00427C84 dd offset aC6vasdnzwd1tfd ; "C6VAsdnzwd1TFdC"
.rdata:00427C88 dd offset aL1nxm ; "l+1nxM"
.rdata:00427C8C dd offset a04fw ; "o4fW"
.rdata:00427C90 dd offset aSynt1Yg1xpz ; "sYNT1+yg1XPz"
.rdata:00427C94 dd offset aN3mxdk8xzb ; "N+3MxDK8xzb"
.rdata:00427C98 dd offset a0dnwfif1 ; "oDnWfIF1"
.rdata:00427C9C dd offset aJpv7Flomldfhx6 ; "JpU7FLOmldfHx6nTFqQm3S70BmynFd1TsXnAs/Q"...
.rdata:00427CA0 dd offset aJpk9f1foxdnz56 ; "JPK9FLfoxdnz56fHJMjkodf0s+K91XnTFq7RJIn"...
```

When any of them is needed, it is selected by its index and passed to the decoding function:

```

0040E930 decode_from_the_list proc near
0040E930
0040E930 arg_0= dword ptr 8
0040E930 arg_4= dword ptr 0Ch
0040E930
0040E930 push    ebp
0040E931 mov     ebp, esp
0040E933 mov     eax, [ebp+arg_4]
0040E936 mov     ecx, [ebp+arg_0]
0040E939 mov     edx, ds:str_list[ecx*4]
0040E940 push    eax                ; int
0040E941 push    edx                ; Src
0040E942 call   decode_string
0040E947 add     esp, 8
0040E94A pop     ebp
0040E94B retn
0040E94B decode_from_the_list endp

```

Example – string fetched by the [index 162](#):

```

00402E4D push    esi
00402E4E lea    eax, [ebp+Dest]
00402E54 push    162                ; "D:(A;;GA;;;WD)(A;;GA;;;BA)(A;;GA;;;SY)(A;;GA;;;RC)"
00402E59 push    eax
00402E5A call   decode_and_convert_to_wchar
00402E5F mov     eax, dword_42A648
00402E64 add     esp, 8
00402E67 push    0
00402E69 lea    ecx, [ebp+var_4]
00402E6C push    ecx
00402E6D mov     ecx, [eax+174h]
00402E73 push    1
00402E75 lea    edx, [ebp+Dest]
00402E7B push    edx
00402E7C call   ecx                ; ADVAPI32.ConvertStringSecurityDescriptorToSecurityDescriptorW
00402E7E test   eax, eax

```

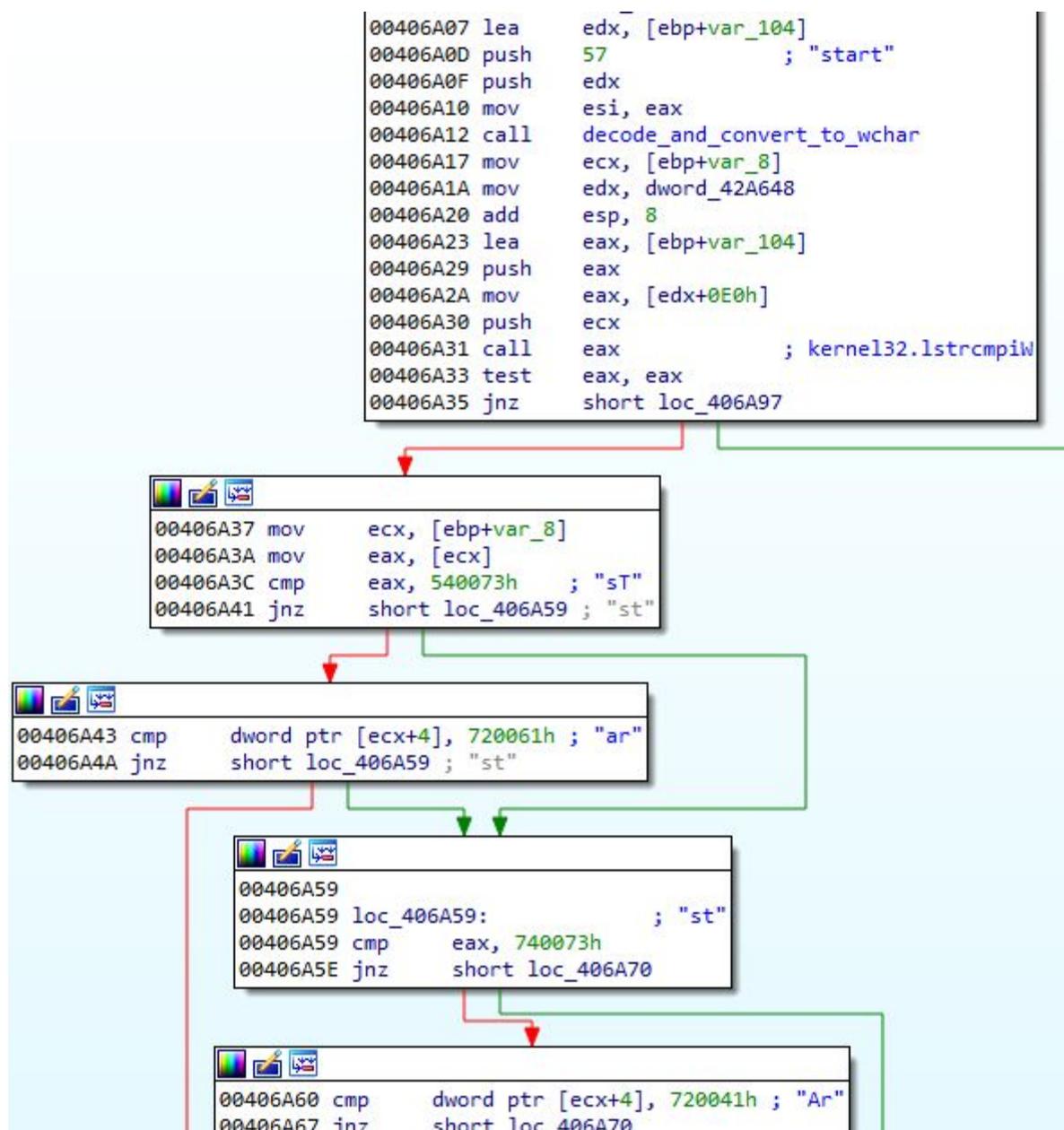
The deobfuscation process, along with [the used utility](#), was described [here](#). Due to the fact that the API of the decoding functions didn't change since then, the same method can be used until today. The list of deobfuscated strings, extracted from the currently analyzed sample can be found [here](#).

Additionally, we can find other, more popular methods of strings obfuscation. For example, some of the strings that are divided into chunks, one DWORD per each:

```
0041F2D1 push    0FDE9h
0041F2D6 push    edx
0041F2D7 mov     [ebp+var_80], 7261h ; ar
0041F2DE mov     [ebp+var_84], 7A61622Eh ; .baz
0041F2E8 mov     [ebp+var_88], 74737572h ; rust
0041F2F2 mov     [ebp+var_8C], 74656661h ; afet
0041F2FC mov     [ebp+var_90], edi
0041F302 call   multibyte_to_wide
0041F307 add     esp, 10h
```

The same method was used by GandCrab, and can be deobfuscated with [the following script](#).

Similarly, the Unicode strings are divided:



Most of the imports used by TrickBot are loaded dynamically. That makes static analysis more difficult, because we cannot directly see the full picture: the pointers are retrieved just before they are used.

We can solve this problem in various ways, i.e. by adding tags by [an automated tracer](#). Created CSV/tags file for one of the analyzed samples is available [here](#) (it can be loaded to the IDA database with the help of [IFL plugin](#)).

The picture given below shows the fragment of TrickBot's code after the tags are loaded. As we can see, the addresses of the imported functions are retrieved from the internal structure rather

than from the standard Import Table, and then they are called via registers.

```
0040BA8B mov     ecx, my_context
0040BA91 mov     edx, [ecx+80h]
0040BA97 push   eax
0040BA98 call   edx             ; kernel32.ResetEvent
0040BA9A mov     eax, [esi+80h]
0040BAA0 mov     ecx, my_context
0040BAA6 mov     edx, [ecx+80h]
0040BAAC push   eax
0040BAAD call   edx             ; kernel32.ResetEvent
0040BAAF mov     eax, [ebp+arg_4]
0040BAB2 mov     ecx, my_context
0040BAB8 mov     edx, [ecx+94h]
0040BABE push   eax
0040BABF call   edx             ; kernel32.ResumeThread
0040BAC1 test   eax, eax
0040BAC3 jz     short loc_40BB02
```

Apart from the mentioned obfuscation methods, on the way of its evolution, TrickBot is going in the direction of string randomization. Many strings that were hardcoded in the initial versions are now randomized or generated per victim machine. For example the mutex name:

```
0040C7E5
0040C7E5 loc_40C7E5:
0040C7E5 lea    eax, [ebp+VolumeSerialNumber]
0040C7E8 call   shl_1
0040C7ED mov    [ebp+ecx+var_C], al
0040C7F1 inc    ecx
0040C7F2 cmp    ecx, 8
0040C7F5 j1     short loc_40C7E5
```

```
0040C7F7 push   esi
0040C7F8 mov    esi, [ebp+Dest]
0040C7FB push   161
0040C800 push   esi
0040C801 call  decode_and_convert_to_wchar
0040C806 lea   eax, [ebp+Format]
0040C80C push  198 ; "Global\%081X%041X%lu"
0040C811 push  eax
0040C812 call  decode_and_convert_to_wchar
0040C817 mov   ecx, [ebp+var_A]
0040C81A movzx edx, [ebp+var_E]
0040C81E mov  eax, dword ptr [ebp+Args]
0040C821 push  ecx
0040C822 push  edx
0040C823 push  eax ; Args
0040C824 lea  ecx, [ebp+Format]
0040C82A push  ecx ; Format
0040C82B push  64h ; int
0040C82D push  esi ; Dest
0040C82E call  print_mutex_name
```

Used encryption

In the past, modules were [encrypted by AES in CBC mode](#). The key used for encryption was derived by [hashing initial bytes of the buffer](#). Once knowing the algorithm, we could easily decrypt the stored modules along with their configuration.

In the recent update the authors decided to complicate it a bit. Yet they didn't change the main algorithm, but just introduced an additional [XOR layer](#). Before the data is passed to the AES, it is first XORed with a 64 character long, dynamically generated string, that we will refer as the bot key:

```

004011CF push    eax
004011D0 push    edi
004011D1 push    esi
004011D2 call    edx                ; ReadFile
004011D4 test    eax, eax
004011D6 jz     short loc_401208

```

```

004011D8 mov     eax, [ebp+var_4]
004011DB push   eax
004011DC push   edi
004011DD call   dexor_with_bot_key ; added in the new version
004011E2 mov     ecx, [ebp+arg_C]
004011E5 mov     edx, [ebp+arg_8]
004011E8 mov     eax, [ebp+var_4]
004011EB add     esp, 8
004011EE push   ecx
004011EF mov     ecx, [ebp+arg_0]
004011F2 push   edx
004011F3 push   eax
004011F4 push   edi
004011F5 push   ecx
004011F6 mov     ecx, ebx
004011F8 call   aes_decrypt_with_hash_rounds ; same as in previous versions
004011FD test    eax, eax

```

The mentioned bot key is generated per victim machine. First, GetAdapterInfo function is used:

```

0040E2A4 mov     edx, my_context
0040E2AA mov     eax, [edx+1F4h]
0040E2B0 lea    ecx, [ebp+var_4]
0040E2B3 push   ecx
0040E2B4 push   ebx
0040E2B5 call   eax                ; IPHLPAPI.GetAdaptersInfo
0040E2B7 cmp     eax, 6Fh
0040E2BA jnz    short loc_40E2E5

```

The retrieved structure (194 bytes) is hashed by SHA256 and then the hash is converted into string:

```

0040E2E5 loc_40E2E5:          ; CALG_SHA_256
0040E2E5 push    800Ch
0040E2EA lea    edx, [ebp+var_C]
0040E2ED push    edx
0040E2EE lea    eax, [ebp+var_8]
0040E2F1 push    eax
0040E2F2 push    194h          ; size to be hashed
0040E2F7 lea    ecx, [ebx+8]
0040E2FA push    ecx          ; *IPADAPTER_INFO + 8
0040E2FB lea    ecx, [edi+4FCh]
0040E301 call   crypt_hash_data
0040E306 test    eax, eax

```

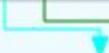
The reconstructed algorithm to generate the Bot Key (and the utility to generate the keys) can be found [here](#).

This key is then stored in the dropped settings file.

```

00423D02 push    ecx
00423D03 push    edi
00423D04 push    ebx
00423D05 push    esi
00423D06 call   eax          ; ReadFile
00423D08 test    eax, eax
00423D0A jz     short loc_423D32

```



```

00423D0C mov    ecx, my_context
00423D12 mov    edx, [ecx+0F8h]
00423D18 push    esi
00423D19 call   edx          ; KiFastSystemCallRet
00423D1B mov    eax, [ebp+arg_4]
00423D1E mov    ecx, [ebp+var_8]
00423D21 push    eax
00423D22 push    ecx
00423D23 push    ebx
00423D24 or     esi, 0FFFFFFFh
00423D27 call   parse_settings_file
00423D2C add    esp, 0Ch

```

Encoding settings

As mentioned before, new editions of TrickBot drop a new settings file, containing some encoded information. Example of the information that is stored in the settings:

```

0441772F66559A1C71F4559DC4405438FC9B8383CE1229139257A7FE6D7B8DE9
1085117245 5 6 13

```

The elements:

1. the BotKey (generated per machine)
2. a checksum of a test string: (0-256 bytes encoded with the same charset) – used for the purpose of a charset validation
3. three random numbers

The whole line is base64 encoded using a custom charset, that is generated basing on the hardcoded one:

“HJIA/CB+FGKLNOP3RSIUWXYZfbcd eaghi5kmn0pqrstuvw89o12467MEDyzQjT”.

```
00422110 push    ebp
00422111 mov     ebp, esp
00422113 sub     esp, 5ACh
00422119 push    ebx
0042211A push    esi
0042211B mov     esi, charset ; "HJIA/CB+FGKLNOP3RSIUWXYZfbcd eaghi5kmn0pqrstuvw89o12467MEDyzQjT"
00422121 push    edi
00422122 mov     ecx, 10h
00422127 lea    edi, [ebp+var_68]
0042212A rep movsd
0042212C mov     ecx, dword_42A668
00422132 xor     ebx, ebx
00422134 call   load_my_botkey ; eax = botkey
00422139 xor     esi, esi
```

Yet, even at this point we can see the effort of the authors to avoid using repeatable patterns. The last 8 characters of the charset are swapped randomly. The pseudocode of the generation algorithm:

```
if ( !g_Charset_copy )
{
    g_Charset_copy = alloc_on_heap(64, 0);
    memcpy((void *)g_Charset_copy, base64_charset, 64u);
    random_swap_last_n_characters(g_Charset_copy, 8u);
    v9 = out_buffer_1;
}
```

Randomization of the n characters:

```

char __cdecl random_swap_last_n_characters(int charset, unsigned int num)
{
    char result; // al
    unsigned int charset_n_bgn; // edi
    unsigned int indx; // esi
    int v5; // ebx
    unsigned int rand_val; // edx

    result = num;
    charset_n_bgn = charset - num;
    indx = 0;
    if ( num )
    {
        do
        {
            do
            {
                v5 = rand();
                rand_val = (v5 + (unsigned int)randval_rdtsc()) % num;
            }
            while ( rand_val == indx );
            result = *(_BYTE *)(indx + charset_n_bgn + 64);
            *(_BYTE *)(indx++ + charset_n_bgn + 64) = *(_BYTE *)(rand_val + charset_n_bgn + 64);
            *(_BYTE *)(rand_val + charset_n_bgn + 64) = result;
        }
        while ( indx < num );
    }
    return result;
}

```

Example of the transformation:

inp: "HJIA/CB+FGKLNOP3RSIUWXYZfbcdeaghi5kmn0pqrstuvwxyz89o12467MEDyzQjT"

out: "HJIA/CB+FGKLNOP3RSIUWXYZfbcdeaghi5kmn0pqrstuvwxyz89o12467jDEzTyQM"

The decoder can be found here: [trick_settings_decoder.py](#)

Slowly improving obfuscation

The authors of TrickBot never cared much about obfuscation. With time they slowly started to introduce its elements, but, apart from some twists, it's still nothing really complex. We can rather expect that this trend will not change rapidly, and after updating the scripts for new additions, decoding Trick Bot elements will be as easy for the analysts as it was before.

It seems that the authors believe in a success based on quantity of distribution, rather than on attempts of being stealthy in the system. They also focus on constant adding new modules, to diversify the functionality (i.e. recently, they added a new module for attacking [Point-Of-Sale systems](#)).

Scripts

Updated scripts for decoding TrickBot modules for malware analysts:

https://github.com/hasherezade/malware_analysis/tree/master/trickbot

Indicators of compromise

Sample hash:

9b6ff6f6f45a18bf3d05bba18945a83da2adfbe6e340a68d3f629c4b88b243a8