

# Introducing TrickBot, Dyreza's successor | Malwarebytes Labs

By Malwarebytes Labs

Published: 2016-10-23 · Archived: 2026-04-05 18:28:58 UTC

Recently, our analyst [Jérôme Segura](#) captured an interesting payload in the wild. It turned out to be a new bot that, at the moment of analysis, hadn't been described yet. According to strings found inside the code, the authors named it [TrickBot](#) (or "TrickLoader").

Many links indicate that this bot is another product of the threat actors previously behind [Dyreza, a credential-stealer](#). While TrickBot seems to be written from scratch, it contains many similar features and solutions to those we encountered analyzing Dyreza.

## Analyzed samples

- [f26649fc31ede7594b18f8cd7cdbbc15](#) – initial sample, dropped by Rig EK
  - [3814abbcd8c8a41665260e4b41af26d4](#) – unpacked: intermediate payload (loader)
    - [f24384228fb49f9271762253b0733123](#) – unpacked: final payload ([TrickBot](#)) – 32bit <- main focus of this analysis
    - [10d72baf2c79b29bad1038e09c6ed107](#) – 64-bit loader
    - [bd79db0f9f8263a215e527d6627baf2f](#) – unpacked: final payload (TrickBot) – 64bit

## TrickBot's modules:

- [533b0bdae7f4c8dcd57556a45e1a62c8](#) – systeminfo32.dll
- [c5a0a3dba3c3046e446bd940c20b6092](#) -systeminfo64.dll
- [90421f8531f963d81cf54245b72cde80](#) – injectDll32.dll
  - [c90f766020855047c3a8138842266c5a](#) – the DLL injected in browsers (32bit)
- [0b521fd97402c02366184ec413e888cc](#) – injectDll64.dll
  - [5a7459fb0b49a8b28fae507730e2a924](#) – the DLL injected in browsers (64bit)

## Additional payload:

- [47d9e7c464927052ca0d22af7ad61f5d](#) – downloaded sample
  - [e80ac57a092ffcf2965613c8b3c537c0](#) – unpacked

## Distribution

The payload was spread via malvertising campaign, which dropped the [Rig EK](#):

XXX Publisher with >30M monthly visits

Fake RU ad infrastructure. serves malvertising

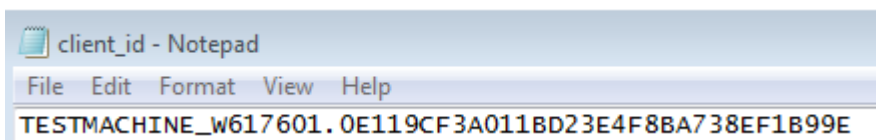
**Behavioral analysis**

After being deployed, TrickBot copies itself into %APPDATA% and deletes the original sample. It doesn't change the initial name of the executable, however. (In the given example, the analyzed sample was named "trick.exe".)

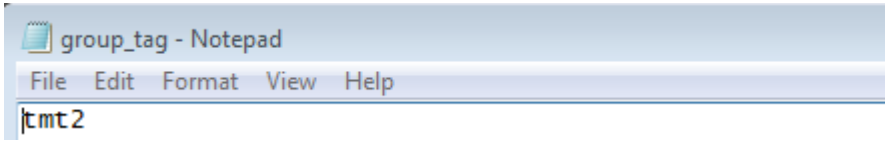
Name	Date modified	Type	Size
Microsoft	2015-07-20 14:15	File folder	
Modules	2016-10-20 16:51	File folder	
Mozilla	2015-06-19 00:38	File folder	
client_id	2016-10-20 16:51	File	1 KB
config.conf	2016-10-20 16:52	CONF File	1 KB
group_tag	2016-10-20 16:51	File	1 KB
trick.exe	2016-10-20 16:41	Application	403 KB

First, we can see it dropping two additional files: *client\_id* and *group\_tag*. They are generated locally and used to identify, appropriately, the individual bot and the campaign to which it belongs. The content of both files is not encrypted; it contains text in Unicode.

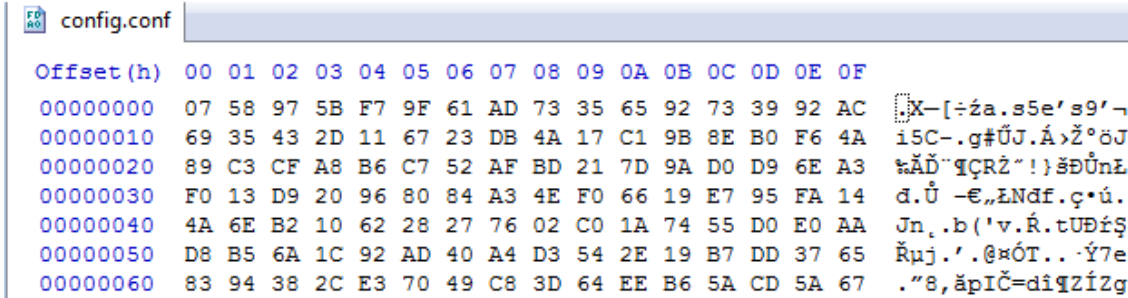
An example of the *client\_id* consists of the name of the attacked machine, operating system version, and a randomly-generated string:



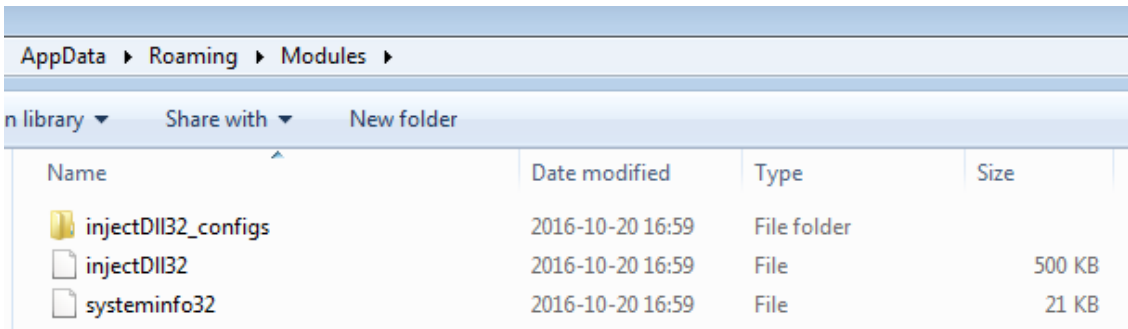
Example of the *group\_tag*:



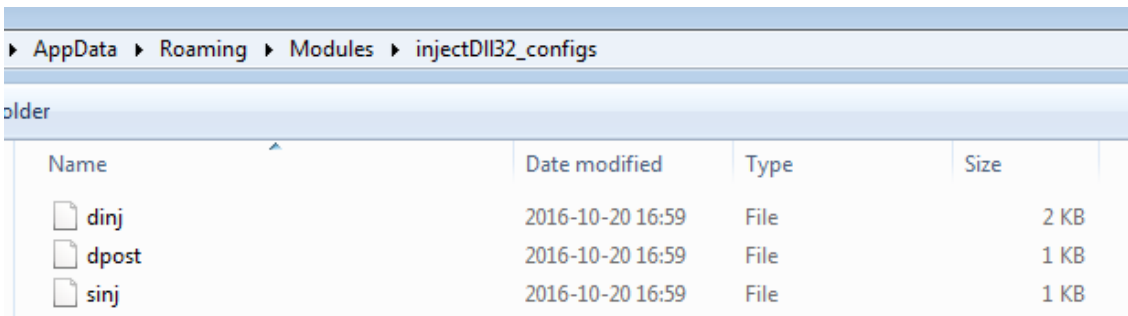
Then, in the same location, we can see *config.conf* appearing. This file is downloaded from the [C&C](#) and stored in encrypted form.



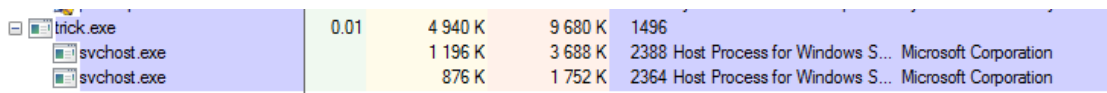
After some time, we can see another folder being created in %APPDATA% named *Modules*. The malware drops additional modules downloaded from the C&C, which are also stored encrypted. In a particular session, TrickBot downloaded modules called *injectDll32* and *systeminfo32*:



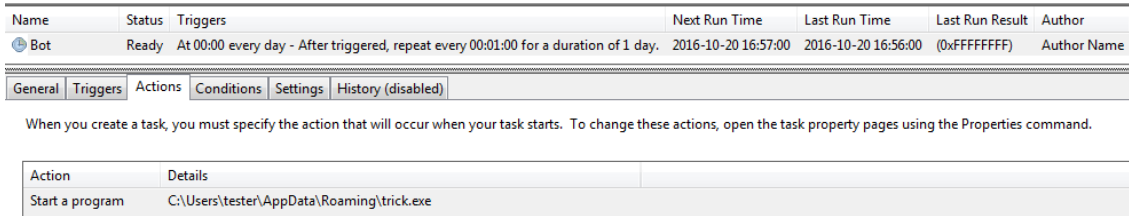
This particular module may also have a corresponding folder where its configuration is stored. The pattern of the naming convention is *[module name]\_configs*.



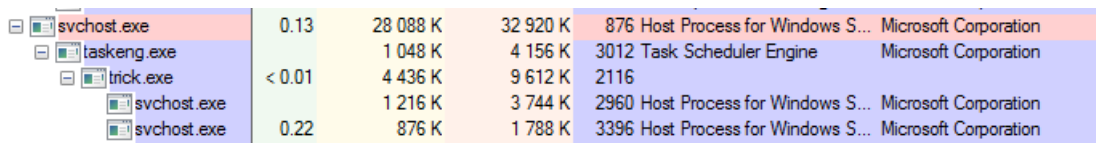
When we observe the execution of the malware via monitoring tools, i.e. ProcessExplorer, we can find it deploying two instances of *svchost*:



The bot achieves persistence by adding itself as a task in Windows Task Scheduler. It doesn't put any effort in hiding the task under a legitimate name, and instead just calls it "Bot."



If the process is killed, it is automatically restarted by the *Task Scheduler Engine*:



## Network communication

TrickBot connects to the several servers:

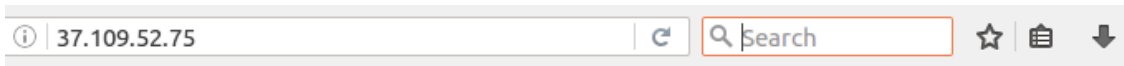
242	myexternalip.com	text/plain	16 bytes	raw
326	myexternalip.com	text/plain	16 bytes	raw
933	15616.royalwebhosting.net	text/html	5684 bytes	BOT_PACKED.bin
1492	207.244.97.80	application/octet-stream	344 kB	?aff_id=1193&auth=2d0fbffe203e050bcc15bd2ebb74f90a&r=9207860&t=1
1569	15616.royalwebhosting.net	text/html	5684 bytes	PreLoader_c07.bin

First, it connects to a legitimate server *myexternalip.com* in order to fetch the IP visible from outside.

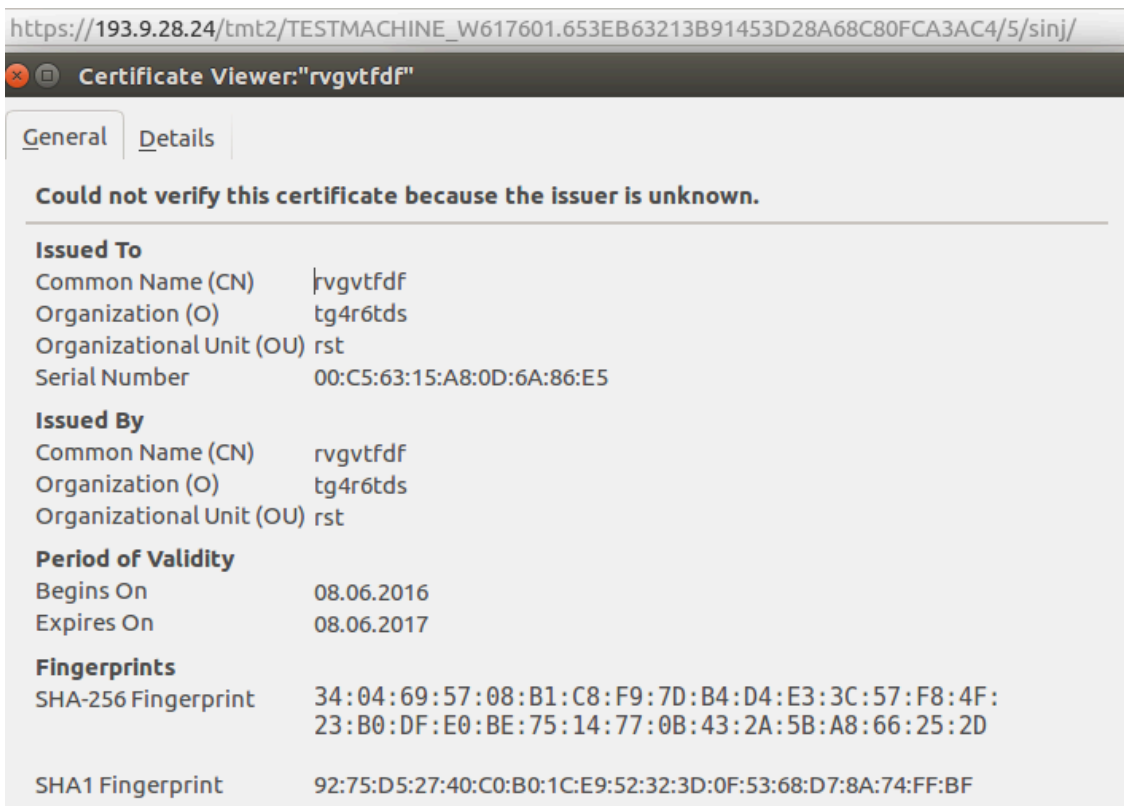
The interesting part is that it doesn't try to disguise as a legitimate browser. Instead, it uses its own User Agent: "BotLoader" or "TrickLoader."

Most—but not all—of the communication with its main C&C is SSL encrypted. Below, you can see an example of one of the commands sent to the C&C:



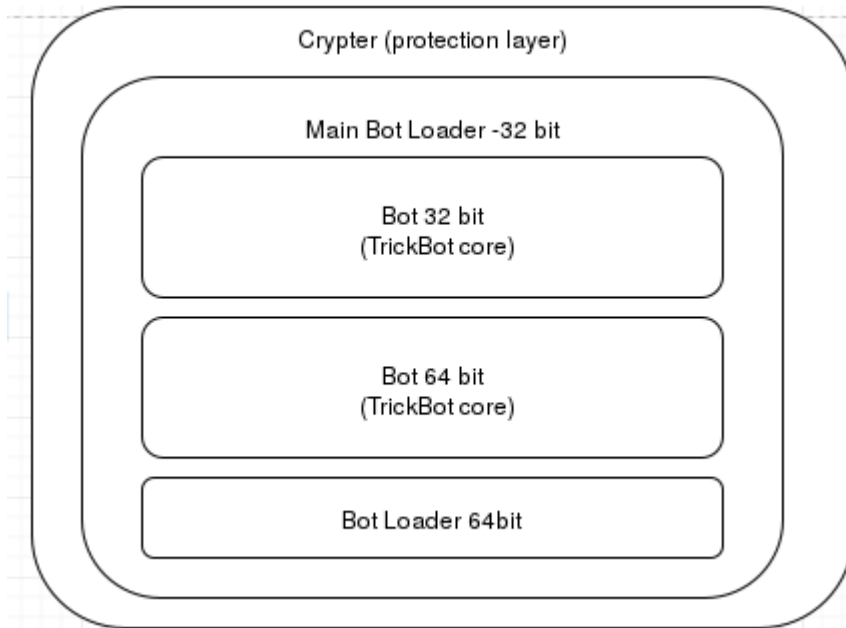


In this example of a used HTTPs certificate, we can see that the used data is fully random and not even trying to imitate legitimate-looking names:



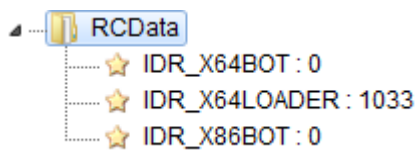
## Inside the malware

TrickBot is composed of several layers. As usual, the first layer is used for protection: It carries the encrypted payload and tries to hide it from AV software.



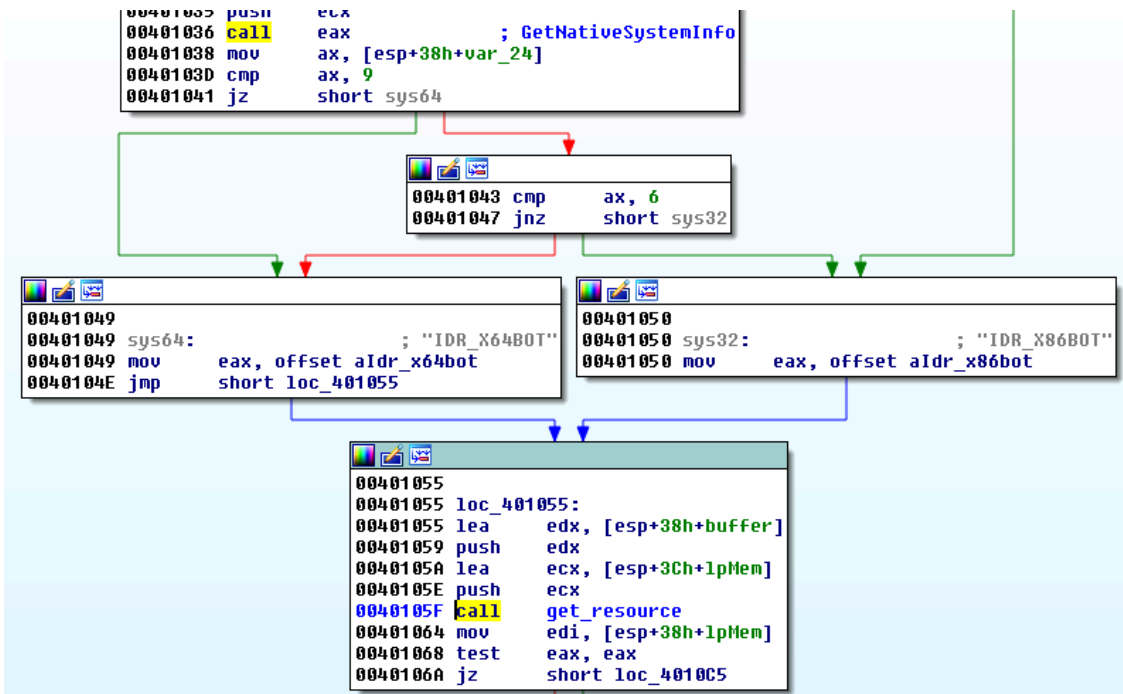
## Loader

The second layer is a main bot loader that chooses whether to deploy a 32-bit or 64-bit payload. New PE files are stored in resources in encrypted form. However, the authors didn't try to hide the functionality of particular elements, and looking at the names of the resources, we can easily guess what their purpose is:

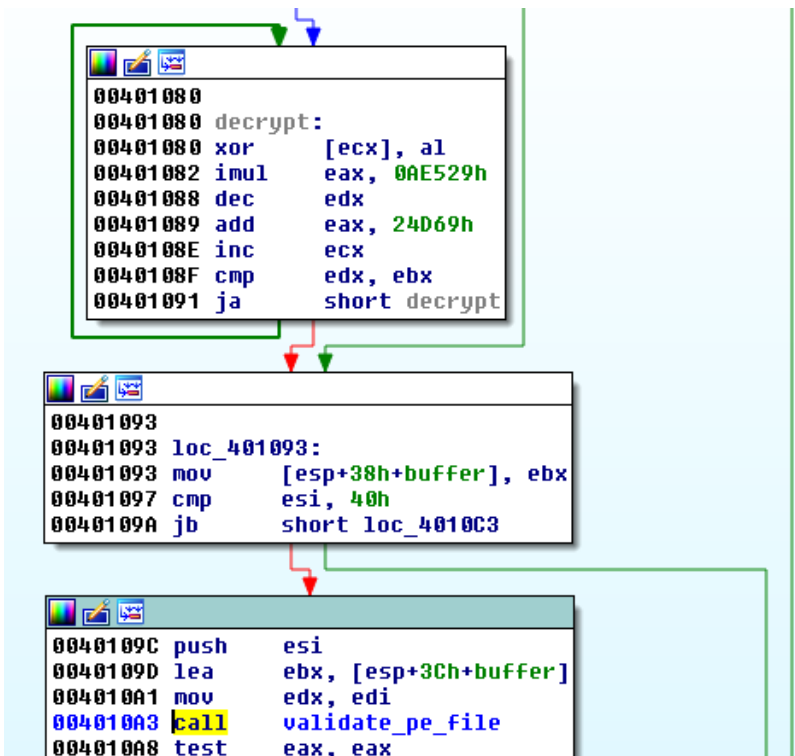


Selected modules are decrypted during execution.

At the beginning, the application fetches information about the victim's operating system in order to choose the appropriate way to follow:



Depending on the environment, a suitable payload is picked from resources, decrypted by a simple algorithm, and validated:



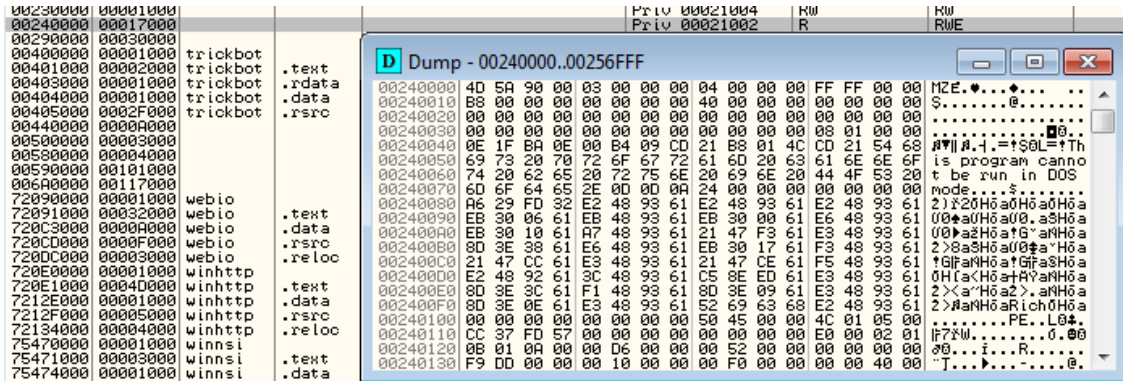
The decrypting procedure is different than [the one found in Dyreza](#), however, the general idea of organizing content (three encrypted modules in resources) is analogous.

```
def decode(data):
    decoded = bytearray()
    key = 0x3039
    i = 0
    for i in range(0, len(data):
```

See full decoding script: [https://github.com/hasherezade/malware\\_analysis/blob/master/trickbot/trick\\_decoder.py](https://github.com/hasherezade/malware_analysis/blob/master/trickbot/trick_decoder.py)

Returning to our malware analysis, next, the unpacked bot is mapped to the memory by a dedicated function and deployed.

The 32-bit bot maps the new module inside its own memory (self-injection):



and then redirects execution there:

```

004017AA | . MOV ECX, DWORD PTR DS:[EDX+0xC] | ntdll.77E37894
004017AD | . MOV EDX, DWORD PTR DS:[ECX+0xC] |
004017B0 | . MOV DWORD PTR DS:[EDX+0x18], EAX |
004017B3 | . MOV ECX, DWORD PTR DS:[ESI+0x28] |
004017B6 | . ADD ECX, EAX |
004017B8 | . CALL ECX | call loaded PE
004017BA | . MOV [LOCAL.4], 0x1 |
    
```

Entry point of the new module (TrickBot core):

```

00240DF9 | CALL 0024E33E | TrickBot main
00240DFE | ^ JMP 00240B5B |
00240E03 | INT3 |
00240E04 | - JMP DWORD PTR DS:[0x24F2C0] | msvcrt.exception::what
00240E0A | - JMP DWORD PTR DS:[0x24F2BC] | msvcrt.exception::~exception
00240E10 | - JMP DWORD PTR DS:[0x24F2B4] | msvcrt.exception::exception
00240E16 | - JMP DWORD PTR DS:[0x24F2B0] | msvcrt.free
00240E1C | - JMP DWORD PTR DS:[0x24F2AC] | msvcrt._CxxThrowException
00240E22 | PUSH 0x14 |
00240E24 | PUSH 0x251358 |
00240E29 | CALL 0024E234 |
00240E2E | MOV EAX, DWORD PTR DS:[0x253810] |
00240E33 | MOV DWORD PTR SS:[EBP-0x1C], EAX |
00240E36 | CMP EAX, -0x1 |
00240E39 | v JNZ SHORT 00240E47 |
    
```

In the case of 64-bit payload being chosen, first the additional executable—a 64bit PE loader—is unpacked and run. Then it loads the core, malicious bot.

In contrast to Dyreza, whose main modules were DLLs, TrickBot uses EXEs.

### The TrickBot internals

The bot is written in C++. It comes with two resources with descriptive names: CONFIG, which stores encrypted configuration, and KEY, which stores the Elliptic Curve key:

RCData	00011764	68 00 00 00 45 43 53 33 30 00 00 00 F3 20 86 DB	h ECS30
CONFIG : 0	00011774	20 4D F0 73 37 B5 FB 18 B0 C0 AF 80 BB F3 FB F1	M s7
KEY : 0	00011784	4A C0 3B C6 00 1F 23 EF 1C 4C 06 54 A3 8F A6 19	J ; # L T
	00011794	7C 41 57 EB 0B BC 7F 41 A1 58 79 70 0D C3 A1 38	AW  A Xyp 8
	000117A4	1C 5E E2 7A D1 29 FB B6 55 41 D5 8E C7 C7 3E 1E	^ z ) UA >
	000117B4	F3 B4 67 63 D3 50 F5 5B 5F D1 C0 56 B8 38 87 DB	gc P [ _ V 8
	000117C4	B5 44 D7 E1 38 79 3E 63 2B 03 2E C8	D 8y>c+ .

In general, this malware is verbose: meaningful names can be found at every stage.

The name “TrickBot” also appears in the name of the global mutex (“Global\TrickBot”) created by the application in order to ensure that it is run only once:

0040CAF9	LEA EAX, [LOCAL_4]	
0040CAF0	MOV ECX, [LOCAL_1]	
0040CAFF	PUSH trick_bo.0040F6F4	
0040CB04	PUSH 0x1	
0040CB06	PUSH EAX	
0040CB07	MOV [LOCAL_4], 0xC	
0040CB0E	MOV [LOCAL_2], 0x0	
0040CB15	MOV [LOCAL_3], ECX	
0040CB18	CALL DWORD PTR DS:[&KERNEL32.	MutexName = "Global\TrickBot"
0040CB1E	MOV DWORD PTR DS:[ESI], EAX	InitialOwner = TRUE
0040CB20	MOV EAX, [LOCAL_1]	pSecurity = 000000B7
0040CB23	TEST EAX, EAX	KernelBa.7611768C
0040CB25	JE SHORT trick_bo.0040CB2E	CreateMutexW
0040CB27	PUSH EAX	
0040CB28	CALL DWORD PTR DS:[&KERNEL32.	hMemory = 000000B7
0040CB2E	CMP DWORD PTR DS:[ESI], 0x0	LocalFree
0040CB31	JNZ SHORT trick_bo.0040CB3B	
0040CB33	PUSH 0x1	no exit
0040CB35	CALL DWORD PTR DS:[&msvcrt.ex	status = 0x1
0040CB3B	CALL DWORD PTR DS:[&KERNEL32.	exit
0040CB41	XOR EDX, EDX	GetLastError
0040CB43	CMP EAX, 0xB7	ERROR_ALREADY_EXISTS
0040CB48	SETE DL	
0040CB4B	MOV EAX, EDX	
0040CB4D	MOV ESP, EBP	
0040CB4F	POP EBP	
0040CB50	RETN	

At first execution, TrickBot copies itself into a new location (in %APPDATA%) and deploys the new copy, giving as an argument path to the original one that needs to be deleted:

00403631	MOV EAX, [LOCAL_2]	
00403634	ADD ESP, 0x18	
00403637	LEA ECX, [LOCAL_11]	
00403639	PUSH ECX	
0040363B	LEA EDI, [LOCAL_30]	
0040363E	PUSH EDI	
0040363F	PUSH EAX	
00403640	PUSH 0x0	
00403642	PUSH 0x0	
00403644	PUSH 0x0	
00403646	PUSH 0x0	
00403648	PUSH 0x0	
00403649	MOV ECX, EBX	
0040364C	PUSH ECX	
0040364D	PUSH 0x0	
0040364F	MOV [LOCAL_30], 0x44	
00403650	CALL DWORD PTR DS:[&KERNEL32.CreateProcessW]	CreateProcessW

```

pProcessInfo = 00238B38
pStartupInfo = 0012F7A0
CurrentDir = "C:\Users\tester\AppData\Roaming"
Environment = NULL
CreationFlags = 0
InheritHandles = FALSE
ThreadSecurity = NULL
ProcessSecurity = NULL
CommandLine = "C:\Users\tester\AppData\Roaming\payload_3.exe \"C:\Users\tester\Desktop\payload_3.exe\""
ModuleFileName = NULL
    
```

Adding a task of running bot into the Task Scheduler:

004012E8	JL trick_bo.00401959	
004012EE	MOV EDI, trick_bo.00410F70	UNICODE "TrickBot"
004012F3	LEA EBX, [LOCAL_4]	
004012F6	CALL trick_bo.00401000	
004012FB	MOV EAX, DWORD PTR DS:[EAX]	
004012FD	CMP EAX, ESI	
004012FF	JE SHORT trick_bo.00401305	
00401301	MOV ECX, DWORD PTR DS:[EAX]	
00401303	JMP SHORT trick_bo.00401307	
00401305	XOR ECX, ECX	
00401307	MOV EAX, [LOCAL_3]	
0040130A	MOV EDX, DWORD PTR DS:[EAX]	
0040130C	PUSH ESI	
0040130D	PUSH ECX	
0040130E	PUSH EAX	
0040130F	MOV EAX, DWORD PTR DS:[EDX+0x3C]	taskschd.742662A1
00401312	CALL EAX	taskschd.742662A1
00401314	LEA EDI, [LOCAL_4]	
00401317	CALL trick_bo.00401050	
0040131C	MOV EDI, trick_bo.00410F2C	UNICODE "Bot"
00401321	LEA EBX, [LOCAL_4]	
00401324	CALL trick_bo.00401000	

Setting the triggering event:

```

0040154F . . . . . CALL EAX
00401551 . . . . . TEST EAX,EAX
00401553 . . . . . JS trick_bo.00401959
00401559 . . . . . MOV EDI,trick_bo.00410F54
0040155E . . . . . LEA EBX,[ARG_1]
00401561 . . . . . CALL trick_bo.00401000
00401566 . . . . . MOV EAX,DWORD PTR DS:[EAX]
00401568 . . . . . CMP EAX,ESI
0040156A . . . . . JE SHORT trick_bo.00401570
0040156C . . . . . MOV ECX,DWORD PTR DS:[EAX]
0040156E . . . . . JMP SHORT trick_bo.00401572
00401570 . . . . . XOR ECX,ECX
00401572 . . . . . MOV EAX,[LOCAL_6]
00401575 . . . . . MOV EDX,DWORD PTR DS:[EAX]
00401577 . . . . . PUSH ECX
00401578 . . . . . PUSH EAX
00401579 . . . . . MOV EAX,DWORD PTR DS:[ECX+0x24]
0040157C . . . . . CALL EAX
0040157E . . . . . LEA EDI,[ARG_1]
00401581 . . . . . MOV EBX,EAX
00401583 . . . . . CALL trick_bo.00401050
00401588 . . . . . CMP EBX,ESI
0040158A . . . . . JL trick_bo.00401959
00401590 . . . . . MOV EDI,trick_bo.00410F90
00401595 . . . . . LEA EBX,[ARG_1]

```

We can find the date pointing to the beginning of 2016, which may confirm the observation that the bot is new, and was written this year.

### TrickBot's commands

TrickBot communicates with its C&C and sends several commands in a format similar to the one used by Dyreza. Below is list of format strings used by TrickBot commands:

```

0040542A UNICODE "%s/%s/0/%s/%s/%s/%s/"
004055A6 UNICODE "%s/%s/1/%s/"
00405730 UNICODE "%s/%s/5/%s/"
004058AF UNICODE "%s/%s/10/%s/%d/"
00405A06 UNICODE "%s/%s/14/%s/%s/0/"
00405B4C UNICODE "%s/%s/23/%d/"
00405CEC UNICODE "%s/%s/25/%s/"
00405EE7 UNICODE "%s/%s/63/%s/%s/%s/"

```

Compare that with Dyreza's command format:

```

0F233C93 ASCII "%s/%s/0/%s/%d/%s/%s/"
0F233CB5 ASCII "%s/%s/0/%s/%d/%s/"
0F233CFF ASCII "%s/%s/%d/%s/"
0F233D1E ASCII "%s/%s/%d/%s/%s/"
0F233E1F ASCII "%s/%s/5/%s/%s/"
0F233E93 ASCII "spk"
0F234007 ASCII "\r\n"
0F2340D5 ASCII "%s/%s/23/%d/%s/%s/"
0F2349FC ASCII "%s/%s/25/%s/%s/"
0F234BC2 ASCII "%s/%s/0"
0F234BF9 ASCII "send system info failed"
0F234BFE ASCII "error"
0F234C7C ASCII "%s/%s/%d/%s/%s/"
0F234E03 ASCII "noname"
0F234E6E ASCII "%s/%s/%d/%s/%s/"
0F234E90 ASCII "%s/%s/%d/%s/"

```

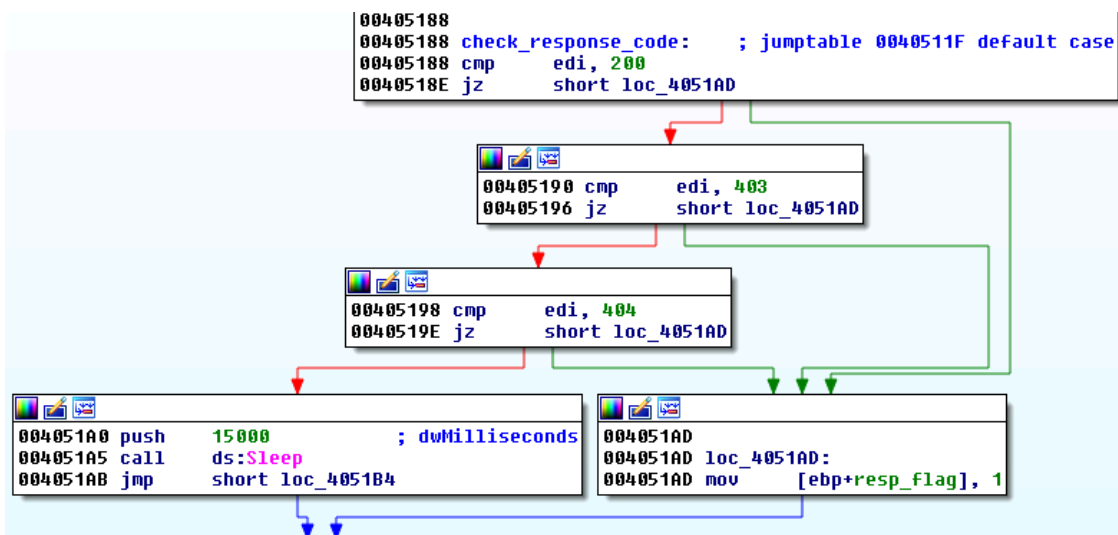
TrickBot's command IDs are hardcoded in the format strings. However, all of them are deployed from inside the same function that gets the command ID as a parameter:

```

0040341C xor     esi, esi
0040341E call    ds:GetUserNameW
00403424 mov     eax, [ebp+arg_0]
00403427 lea    edx, [ebp+Buffer]
0040342D push   edx
0040342E push   offset aUser ; "user"
00403433 push   14
00403435 call   send_command_to_CnC
0040343A add     esp, 0Ch

```

After filling the appropriate format string and sending it to the C&C, the bot checks the HTTP response code. If the returned code is different than 200 (OK), 403 (Forbidden), or 404 (Not found), then it tries again.



Here's a full list of implemented command IDs:

0 1 5 10 14 23 25 63

Each command has the same prefix – that is a group id of the campaign and bot's individual id (the same data that are stored in dropped files). Format:

/[group\_id]/[client\_id]/[command\_id]/...

Sample url:

https://193.9.28.24/tmt2/TESTMACHINE\_W617601.653EB63213B91453D28A68C80FCA3AC4/5/sinj/

More notes about the protocol [here](#).

### Encryption

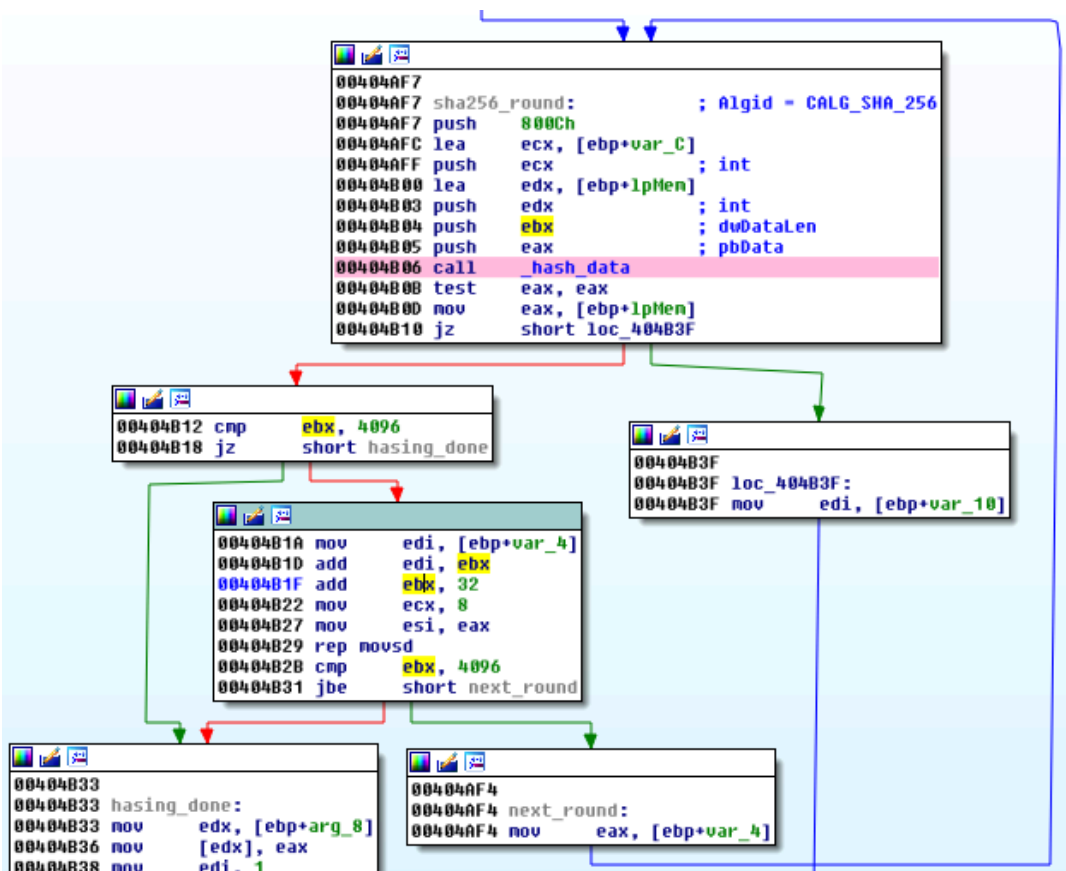
TrickBot uses alternatively two encryption algorithms: AES and ECC.

```
00402360 lea    eax, [ebp+hash_buf]
00402363 push  eax
00402364 lea    ecx, [ebx+10h]
00402367 push  ecx
00402368 push  edi
00402369 call  sha256_128_rounds
0040236E test  eax, eax
00402370 jz    decrypted

00402376 mov    ecx, [ebp+hash_buf]
00402379 lea    edx, [ebp+var_14]
0040237C push  edx ; int
0040237D mov    edx, [ebp+lpMem]
00402380 lea    eax, [ebp+pbData]
00402383 push  eax ; int
00402384 mov    eax, [ebp+arg_0]
00402387 push  ecx ; generated_iv
00402388 push  edx ; generated_key
00402389 add    eax, 0FFFFFFD0h
0040238C push  eax ; dwBytes
0040238D add    ebx, 30h
00402390 push  ebx ; Src
00402391 call  aes_decrypt
00402396 mov    esi, [ebp+pbData]
00402399 test  eax, eax
0040239B jz    short decrypted

0040239D mov    eax, [esi]
0040239F push  0 ; int
004023A1 lea    ecx, [eax+esi+8]
004023A5 push  ecx ; int
004023A6 add    eax, 8
004023A9 push  eax ; dwDataLen
004023AA push  esi ; pbData
004023AB call  ecc_decrypt
004023B0 test  eax, eax
```

The downloaded modules and configuration are encrypted by AES in CBC mode. The AES key and initialization vector are derived from the data, by a custom, predefined algorithm. First, 32 bytes of input data is hashed, using SHA256. Then, the output of the hashing function is appended to the data buffer and hashed again. This step is repeated until the full size of data in buffer become 4096. So, the hashing operation repeats 128 times. Below you can see the responsible fragment of code:



First 32 byte long chunk of data is used as a initial value to derive AES key:

```

00404AE9 . MOV ECX, 0x8
00404AEE . MOV EDI, EAX
00404AF0 . REP MOVS DWORD PTR ES:[EDI], DWORD PTR DS:[ESI]
00404AF2 . JMP SHORT trick_bo.00404AF7
00404AF4 . MOV EAX, [LOCAL.1]
00404AF7 . PUSH 0x800C
00404AFC . LEA ECX, [LOCAL.3]
00404AFF . PUSH ECX
00404B00 . LEA EDX, [LOCAL.2]
00404B03 . PUSH EDX
00404B04 . PUSH EBX
00404B05 . PUSH EAX
00404B06 . CALL trick_bo.00404840
00404B08 . TEST EAX, EAX
    
```

Arg5 = 0000800C  
 Arg4 = 0012F79C  
 Arg3 = 0012F7A0  
 Arg2 = 00000020  
 Arg1 = 002198B8  
 hash\_data

EAX=002198B8

Address	Hex dump	ASCII
002198B8	B1 57 61 FF EF 1F 34 BB 5F 3C 7E 01 24 BF 17 12	Wa '747 <"0577
002198C8	52 E1 1E E1 BD D5 E6 4D 4B 1B 17 FA 41 5D 70 46	RpAb2ASMK+ A]pF
002198D8	74 00 65 00 72 00 5C 00 44 00 65 00 73 00 6B 00	t.e.r.\.D.e.s.k.
002198E8	74 00 6F 00 70 00 5C 00 63 00 6F 00 6E 00 66 00	t.o.p.\.c.o.n.f.
002198F8	69 00 67 00 2E 00 63 00 6F 00 6E 00 66 00 00 00	i.g...c.o.n.f...

And bytes from 16 to 48 are used as a initial value to derive AES initialization vector:

```

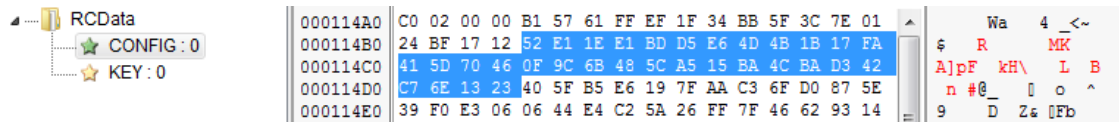
00404AF4 . MOV EAX, [LOCAL.1]
00404AF7 . PUSH 0x800C
00404AFC . LEA ECX, [LOCAL.3]
00404AFF . PUSH ECX
00404B00 . LEA EDX, [LOCAL.2]
00404B03 . PUSH EDX
00404B04 . PUSH EBX
00404B05 . PUSH EAX
00404B06 . CALL trick_bo.00404840
    
```

Arg5 = 0000800C  
 Arg4 = 0012F79C  
 Arg3 = 0012F7A0  
 Arg2 = 00000020  
 Arg1 = 002198B8  
 hash\_data

EAX=002198B8

Address	Hex dump	ASCII
002198B8	52 E1 1E E1 BD D5 E6 4D 4B 1B 17 FA 41 5D 70 46	RpAb2ASMK+ A]pF
002198C8	0F 9C 6B 48 5C A5 15 BA 4C BA D3 42 C7 6E 13 23	*tkH\aq\$  EBan!!#

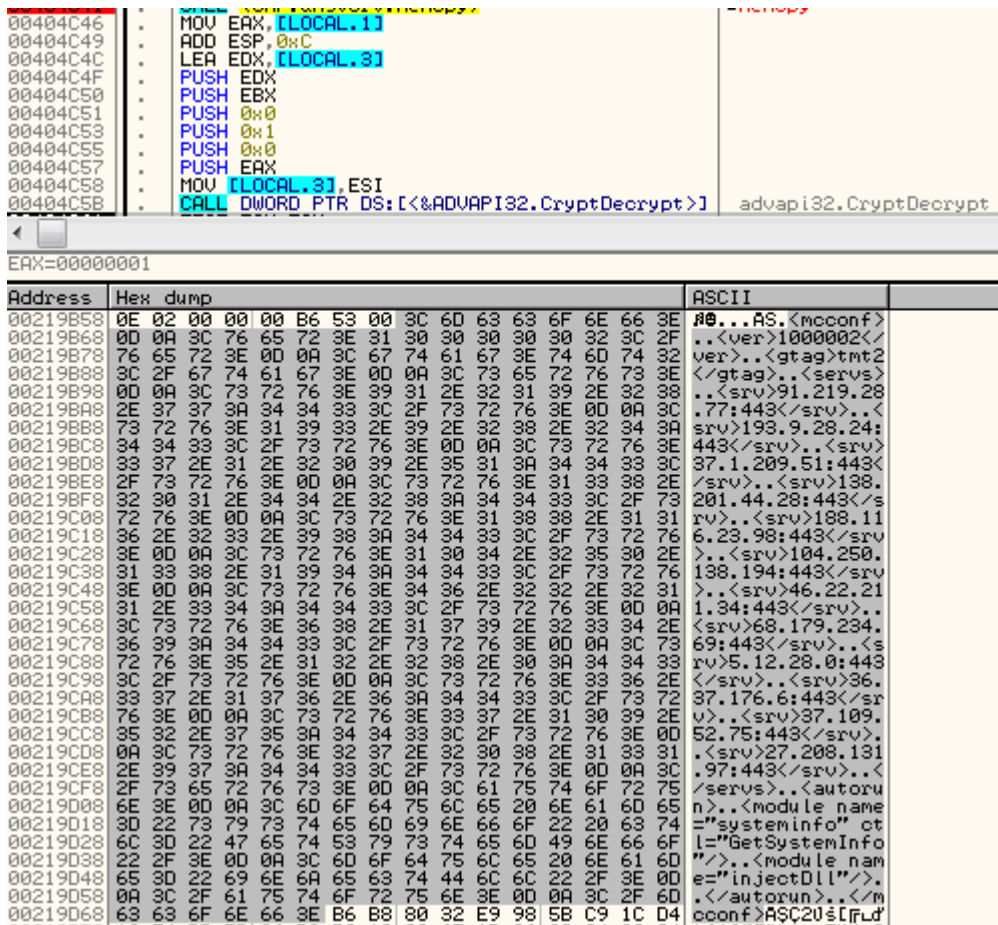
Compare with the content of CONFIG (mind the fact that the first DWORD is a size, and is not included in the data):



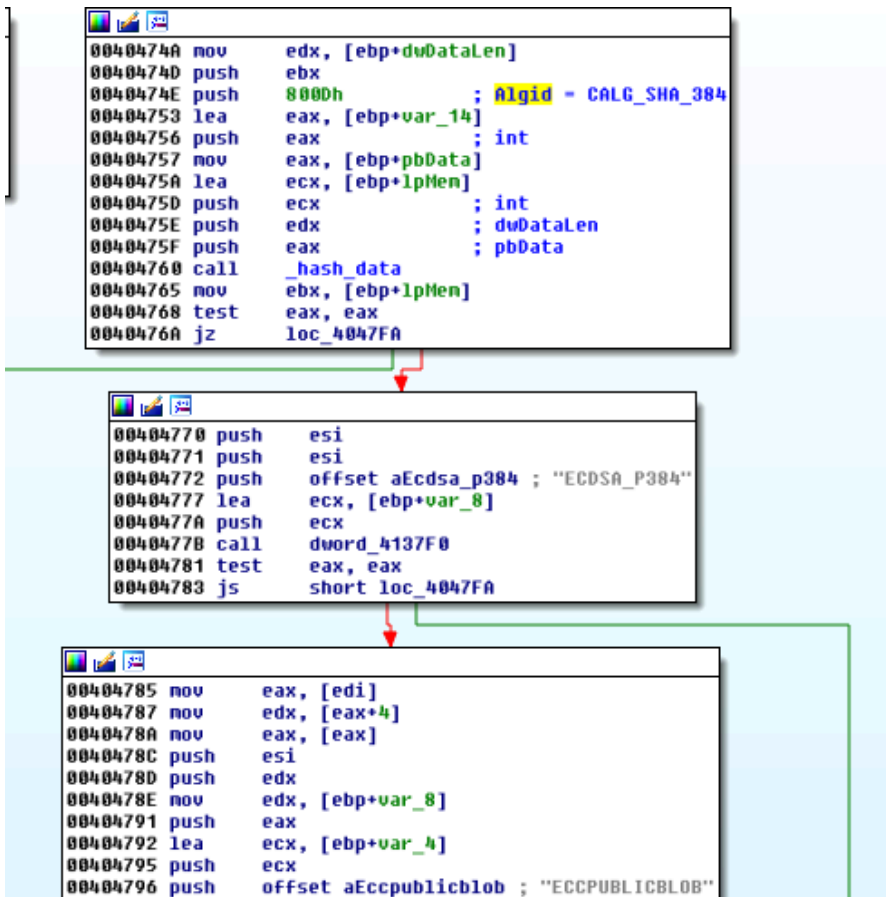
Full decoding script you can find here:

[https://github.com/hasherezade/malware\\_analysis/blob/master/trickbot/trick\\_config\\_decoder.py](https://github.com/hasherezade/malware_analysis/blob/master/trickbot/trick_config_decoder.py)

Decrypting hardcoded configuration using AES:



In case if particular input could not be decrypted via AES, the attempt is made to decrypt it via ECC:



### Trick Bot's configuration

Similarly to Dyreza, TrickBot uses configuration files, that are stored encrypted.

Trick Bot's executable comes with a hardcoded configuration, that, during execution is substituted by its fresh version, downloaded from the C&C and saved in the file *config.conf*. Below you can see the decrypted content of the hardcoded one:

<https://gist.github.com/hasherezade/0c464f970018f509444243b67a0c5447#file-mcconf-xml>

Compare it with a downloaded one – version number got incremented, and some C&Cs have changed:

<https://gist.github.com/hasherezade/0c464f970018f509444243b67a0c5447#file-mcconf2-xml>

Notice that names of the listed modules (*systeminfo*, *injectDll*) are corresponding to those, that we found in the folder *Modules* during the behavioral analysis. It is due to the fact, that this configuration gives instructions to the bot, and orders it to download particular elements.

Some of the requests result in downloading additional pieces of configuration. Example of the response, after being decrypted by the bot:

<https://gist.github.com/hasherezade/0c464f970018f509444243b67a0c5447#file-servconf-xml>

### Modules

TrickBot is a persistent botnet agent – but its main power lies in the modules, that are DLLs dynamically fetched from the C&C. During the analyzed session, the bot downloaded two modules.

- getsysinfo – used for general system info gathering
- injectDll – the banker module, injecting DLLs in target browsers in order to steal credentials

List of the attacked browser is hardcoded in the injectDll32.dll:

```

6C9F190F . JE injectDl.6C9F1A9A
6C9F1915 . CMP EAX,ESI
6C9F1917 . JE injectDl.6C9F1A9A
6C9F191D . PUSH EAX
6C9F191E . PUSH 0
6C9F1920 . PUSH 43A
6C9F1925 . CALL DWORD PTR DS:[<&KERNEL32.OpenProce~
6C9F1928 . MOV DWORD PTR SS:[ESP+1C],EAX
6C9F192F . TEST EAX,EAX
6C9F1931 . JE injectDl.6C9F1A9A
6C9F1937 . TEST EDI,EDI
6C9F1939 . JE SHORT injectDl.6C9F1952
6C9F193B . MOV EAX,DWORD PTR SS:[ESP+18]
6C9F193F . MOV ECX,1
6C9F1944 . CMP DWORD PTR SS:[ESP+48],EDI
6C9F1948 . MOVZX EAX,ECX
6C9F194E . MOV DWORD PTR SS:[ESP+18],EAX
6C9F1952 > LEA EAX,DWORD PTR SS:[ESP+64]
6C9F1956 . PUSH injectDl.6CA075EC ASCII "chrome.exe"
6C9F195B . PUSH EAX
6C9F195C . CALL injectDl.6C9F2D60
6C9F1961 . LEA ECX,DWORD PTR SS:[ESP+6C]
6C9F1965 . ADD ESP,8
6C9F1968 . CMP EAX,ECX
6C9F196A . JNZ SHORT injectDl.6C9F197D
6C9F196C . TEST EDI,EDI
6C9F196E . JNZ SHORT injectDl.6C9F197D
6C9F1970 . MOV EAX,DWORD PTR SS:[ESP+48]
6C9F1974 . LEA ESI,DWORD PTR DS:[EDI+11]
6C9F1977 . MOV DWORD PTR SS:[ESP+14],EAX
6C9F197B . JMP SHORT injectDl.6C9F19C1
6C9F197D > LEA EAX,DWORD PTR SS:[ESP+64]
6C9F1981 . PUSH injectDl.6CA075F8 ASCII "ieexplore.exe"
6C9F1986 . PUSH EAX
6C9F198C . CALL injectDl.6C9F2D60
6C9F1990 . LEA ECX,DWORD PTR SS:[ESP+6C]
6C9F1990 . ADD ESP,8
6C9F1993 . CMP EAX,ECX
6C9F1996 . JNZ SHORT injectDl.6C9F199E
6C9F1996 . MOV ESI,2
6C9F1997 . JMP SHORT injectDl.6C9F19C1
6C9F199E > LEA EAX,DWORD PTR SS:[ESP+64]
6C9F19A2 . PUSH injectDl.6CA07608 ASCII "firefox.exe"
6C9F19A7 . PUSH EAX
6C9F19A8 . CALL injectDl.6C9F2D60
    
```

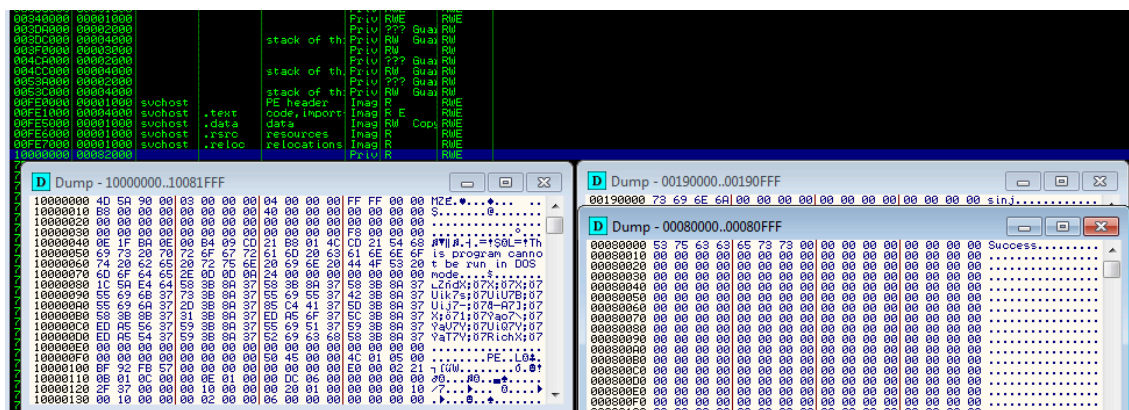
In case of the Dyreza, this attack was performed directly from the main bot, rather than from the added DLL.

Details of the attacked target are given in an additional configuration file, stored in the folder: *ModulesinjectDll32\_config*. Below we can see its decrypted form revealing the attacked online-banking systems:

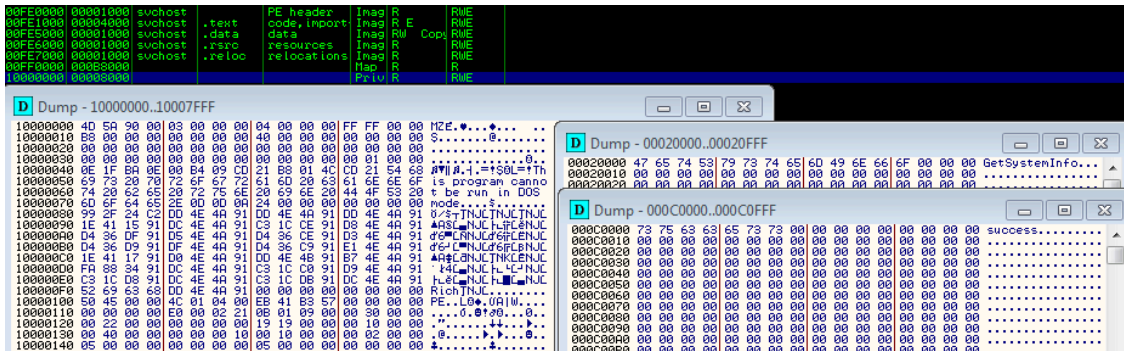
<https://gist.github.com/hasherezade/0c464f970018f509444243b67a0c5447#file-dinj-xml>

The instances of svchost.exe, observed during the behavioral analysis, are used to deploy particular modules.

Below – the module *injectDll* (marked *inj*) in memory of *svchost*:



and the module *systeminfo* (marked *GetSystemInfo*) in memory of the another instance of *svchost*:



## Conclusion

Trick Bot have many similarities with Dyreza, that are visible at the code design level as well as the communication protocol level. However, comparing the code of both, shows, that it has been rewritten from scratch.

So far, Trick Bot does not have as many features as Dyreza bot. It may be possible, that the authors intentionally decided to make the main executable lightweight, and focus on making it dynamically expendable using downloaded modules. Another option is that it still not the final version.

One thing is sure – it is an interesting piece of work, written by professionals. Probability is very high, that it will become as popular as its predecessor.

## Appendix

<http://www.threatgeek.com/2016/10/trickbot-the-dyre-connection.html> – analysis of the TrickBot at Threat Geek Blog

---

*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://hshrdz.wordpress.com>.*

---

Source: <https://blog.malwarebytes.com/threat-analysis/2016/10/trick-bot-dyrezas-successor/>