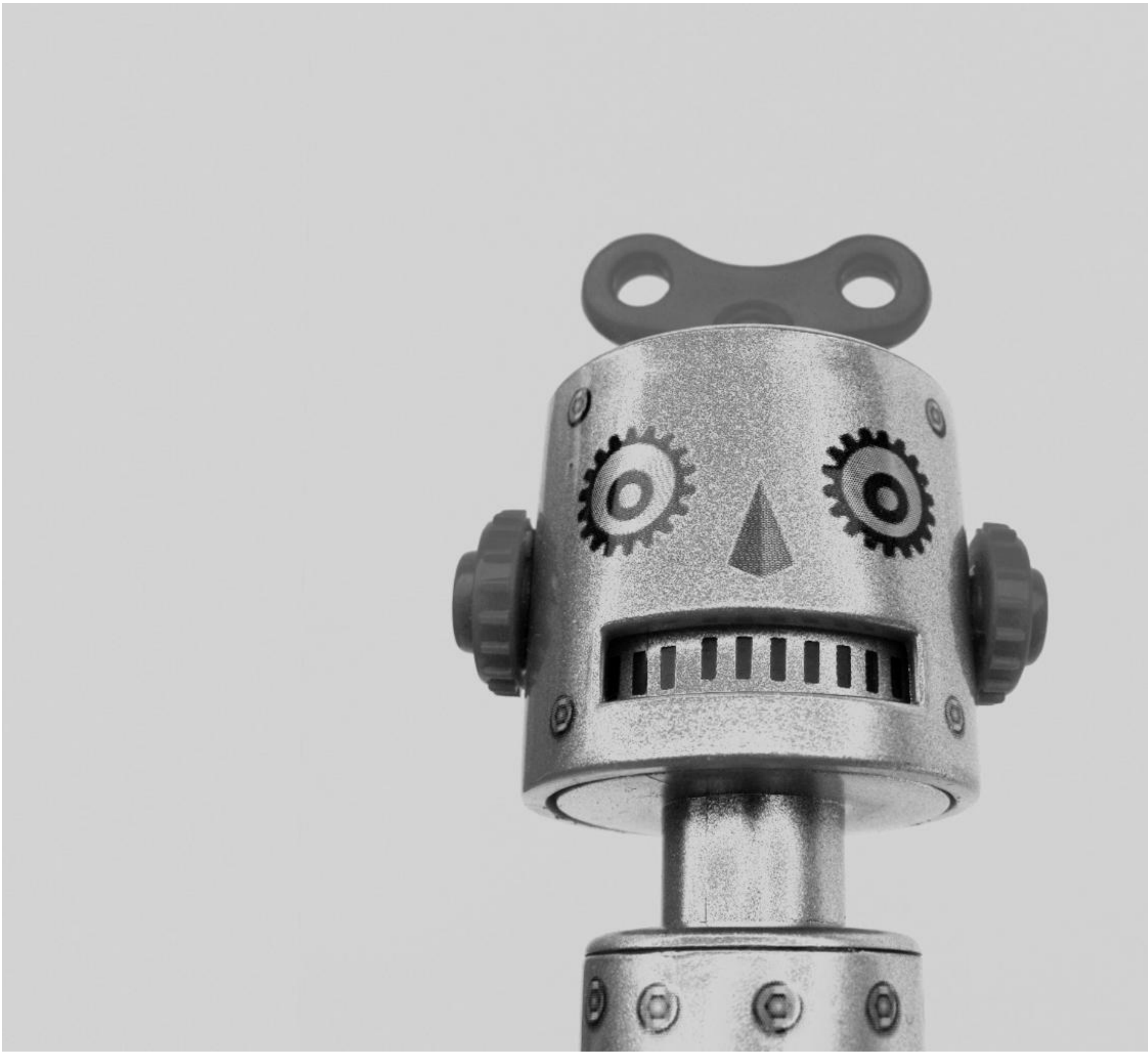
Detricking TrickBot Loader | CERT Polska

cert.pl/en/news/single/detricking-trickbot-loader/



TrickBot (TrickLoader) is a modular financial malware that first surfaced in October in 2016[1]. Almost immediately researchers have noticed similarities with a credential-stealer called Dyre. It is still believed that those two families might've been developed by the same actor.
But in this article we will not focus on the core itself but rather the loader whose job is to decrypt the payload and execute it.

Samples analyzed

- **preloader** b401a0c3a64c2e5a61070c2ae158d3fcf8ebbb51b33593323cd54bbe03d3de00
- **loader** 8d56f6816f24ec95524d6b434fc25f9aad24a27dbb67eab0106bbd7b4160dc75
- **core-32b** cbb5ea4210665c6a3743e2b7c5a29d10af21efddfbab310035c9a14336c71de3
- **core-64b** 028e29ef2543daa1729b6ac5bf0b2551dc9a4218a71a840972cdc50b23fe83c4
- **core-64b-loader** 52bc216a6de00151f32be2b87412b6e13efa5ba6039731680440d756515d3cb9

Original binary

While the binary has two consecutive loaders, the first one will be glossed over because of low level of complexity:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HMODULE v3; // eax
  unsigned int v4; // eax
  unsigned int v6; // [esp+14h] [ebp-24h]
  unsigned __int8 *v7; // [esp+18h] [ebp-20h]
  unsigned int v8; // [esp+1Ch] [ebp-1Ch]
  char *v9; // [esp+20h] [ebp-18h]
  char *v10; // [esp+24h] [ebp-14h]
  unsigned int v11; // [esp+28h] [ebp-10h]
  FARPROC v12; // [esp+2Ch] [ebp-Ch]

  __main();
  v7 = 0;
  v6 = 0;
  v12 = 0;
  v11 = strlen(payload_Base64);
  v10 = (char *)reverse_alloc(payload_Base64);
  Base64DecodeA(&v7, &v6, v10, v11);
  v9 = "@07w+GVb(B$YaXVHGo";
  v8 = 692;
  v3 = GetModuleHandleA("KERNEL32.DLL");
  v12 = GetProcAddress(v3, "SetProcessDEPPolicy");
  ((void (__stdcall *)(_DWORD))v12)(0);
  v4 = strlen(v9);
  RC4((unsigned __int8 *)LoadPE, (unsigned __int8 *)v9, v8, v4);
  LoadPE(v7);
  return 0;
}
```

Original binary's entry point, observed symbols were embedded in the binary

Functions buffer

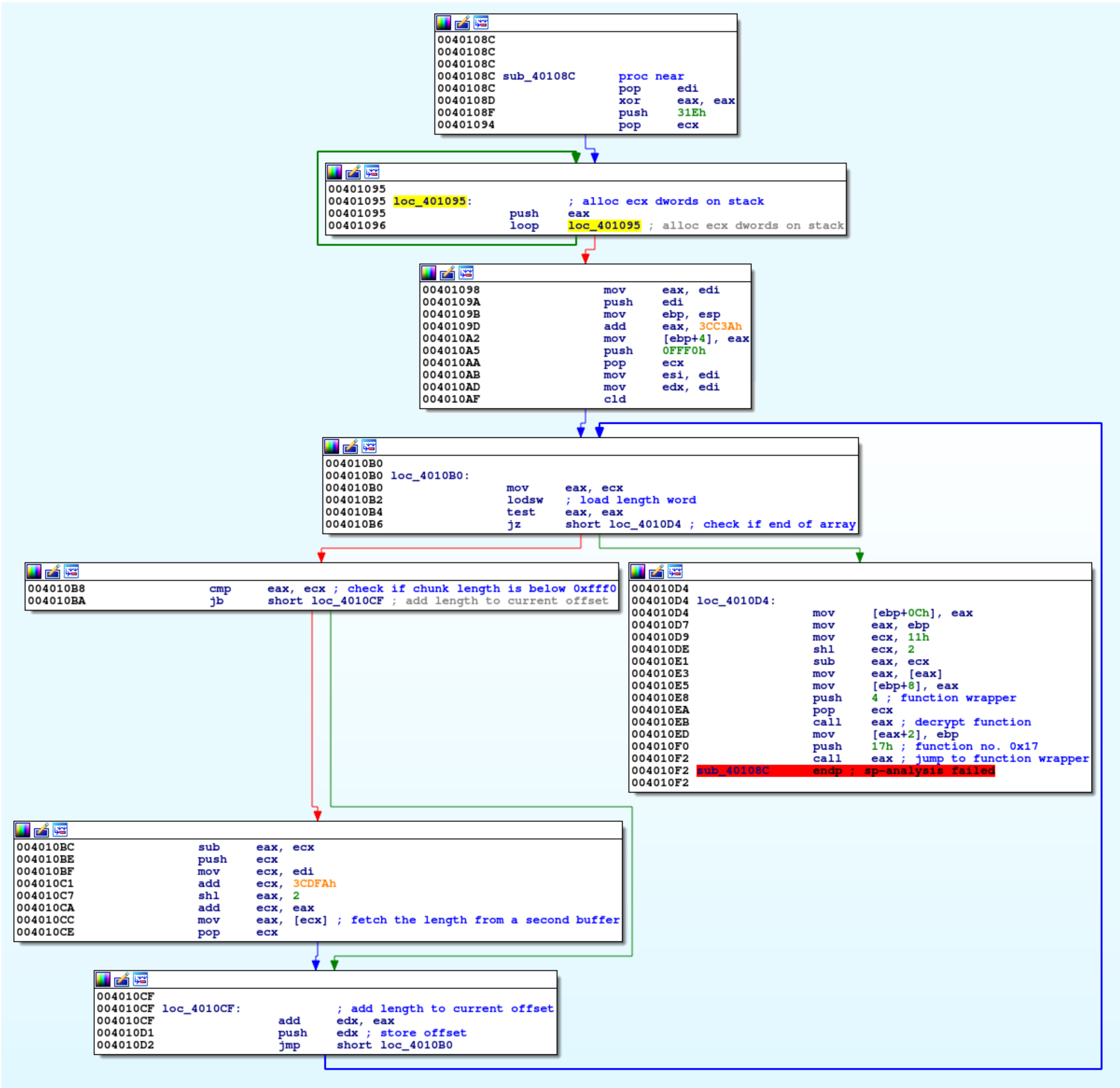
The first thing we notice after loading the RC4-decrypted payload from the previous stage is that IDA hasn't automatically recognized a single valid function.

```
.text:00401000 ; Segment type: Pure code
.text:00401000 ; Segment permissions: Read/Write/Execute
.text:00401000 _text           segment para public 'CODE' use32
.text:00401000                 assume cs:_text
.text:00401000                 ;org 401000h
.text:00401000                 assume es:nothing, ss:nothing, ds:_text, fs:nothing, gs:nothing
.text:00401000
.text:00401000                 public start
.text:00401000 start:
.text:00401000                 push    30000
.text:00401005                 pop     ecx                diy sleep
.text:00401006
.text:00401006 loc_401006:
.text:00401006                 inc     eax
.text:00401007                 loop    loc_401006
.text:00401009                 call    loc_40108C         push lengths addr and jump to loader
.text:00401009 ; ---------------------------------------------------------------------------
.text:0040100E                 dw 0E7h
.text:00401010                 dw 24Bh
.text:00401012                 dw 0C0h
.text:00401014                 dw 3Dh
.text:00401016                 dw 0C3h
.text:00401018                 dw 0B6h
.text:0040101A                 dw 0FFF0h
.text:0040101C                 dw 0DAh
.text:0040101E                 dw 0A9h
.text:00401020                 dw 8Fh
.text:00401022                 dw 67h
.text:00401024                 dw 107h
.text:00401026                 dw 199h
.text:00401028                 dw 40h
.text:0040102A                 dw 19h
.text:0040102C                 dw 67h
.text:0040102E                 dw 51h
.text:00401030                 dw 4Eh
.text:00401032                 dw 59h                     chunks lengths
.text:00401034                 dw 0BFh
.text:00401036                 dw 0ABh
.text:00401038                 dw 23Eh
.text:0040103A                 dw 6Fh
.text:0040103C                 dw 15Ah
.text:0040103E                 dw 236h
.text:00401040                 dw 67Ah
.text:00401042                 dw 3Eh
.text:00401044                 dw 21h
.text:00401046                 dw 80h
.text:00401048                 dw 40h
.text:0040104A                 dw 39h
.text:0040104C                 dw 39h
.text:0040104E                 dw 1Dh
.text:00401050                 dw 82h
.text:00401052                 dw 70Fh
.text:00401054                 dw 82h
.text:00401056                 dw 1B7h
.text:00401058                 dw 50h
.text:0040105A                 dw 0E9h
.text:0040105C                 dw 8Dh
.text:0040105E                 dw 12Bh
.text:00401060                 dw 20h
```

The binary's entry point

This section's permissions are also looking quite suspicious, because section needs to be readable, executable **and** writable.

Function that starts just after the chunks lengths' last entry (begins at 0x40108C), is responsible for calculating the starting offset for each function (or binary chunk) and storing it into an array stored on stack.



Function used for calculating addresses

The functions' objective is pretty straight-forward:

- Iterate over the null-terminated chunks lengths array
- If a length is larger than or equal to 0xFFF0, fetch the full length from a second buffer located further in the data (+0xCDFA in this sample)
- Add the current function's length to the accumulator
- Push the accumulator onto stack

The final array of pointers looks as follows (remember that since values are pushed onto stack, the pointers are reversed relatively to their position in the lengths array):

```
Stack[00000B68]:0012F2E6    db  41h ; Â
Stack[00000B68]:0012F2E7    db    0
Stack[00000B68]:0012F2E8    db 0CFh ; Ï
Stack[00000B68]:0012F2E9    db 0FAh ; ú
Stack[00000B68]:0012F2EA    db  41h ; A
Stack[00000B68]:0012F2EB    db    0
Stack[00000B68]:0012F2EC    dd 41F975h
Stack[00000B68]:0012F2F0    dd 41F906h
Stack[00000B68]:0012F2F4    dd offset unk_41F6C8
Stack[00000B68]:0012F2F8    dd offset unk_41F61D
Stack[00000B68]:0012F2FC    dd offset unk_41F55E
Stack[00000B68]:0012F300    dd offset sub_41F505
Stack[00000B68]:0012F304    dd offset decrypt_function
Stack[00000B68]:0012F308    dd offset unk_41F466
Stack[00000B68]:0012F30C    dd offset unk_41F3FF
Stack[00000B68]:0012F310    dd offset unk_41F3E6
Stack[00000B68]:0012F314    dd offset unk_41F3A6
Stack[00000B68]:0012F318    dd offset unk_41F20D
Stack[00000B68]:0012F31C    dd 41F106h
Stack[00000B68]:0012F320    dd 41F09Fh
Stack[00000B68]:0012F324    dd 41F010h
Stack[00000B68]:0012F328    dd 41EF67h
Stack[00000B68]:0012F32C    dd 41EE8Dh
Stack[00000B68]:0012F330    dd offset unk_4015B6
Stack[00000B68]:0012F334    dd offset loc_401500                ; ...
Stack[00000B68]:0012F338    dd 40143Dh                          ; functions[4]
Stack[00000B68]:0012F33C    dd offset unk_401400                ; functions[3]
Stack[00000B68]:0012F340    dd offset unk_401340                ; functions[2]
Stack[00000B68]:0012F344    dd offset unk_4010F5                ; functions[1]
EBP Stack[00000B68]:0012F348    dd offset word_40100E           ; stored pointer
Stack[00000B68]:0012F34C    dd offset off_43DC48
Stack[00000B68]:0012F350    dd offset decrypt_function
Stack[00000B68]:0012F354    db    1
Stack[00000B68]:0012F355    db    0
Stack[00000B68]:0012F356    db    0
Stack[00000B68]:0012F357    db    0
Stack[00000B68]:0012F358    db 0E8h ; è
Stack[00000B68]:0012F359    db 0F8h ; ø
Stack[00000B68]:0012F35A    db  12h
Stack[00000B68]:0012F35B    db    0
Stack[00000B68]:0012F35C    db    0
```

The pointer to the array is stored in EBP register and passed between almost all functions in the future

Code encryption

The previously mentioned code encryption is done using a standard repeating xor cipher:

The xor key seems to be located around the base64-encoded strings:

```
.text:0043DE07    db    0
.text:0043DE08    db 0D7h ; ×
.text:0043DE09    db 0D8h ; Ø
.text:0043DE0A    db    1
.text:0043DE0B    db    0
.text:0043DE0C    db  6Bh ; k
.text:0043DE0D    db  9Ch ; œ
.text:0043DE0E    db    1
.text:0043DE0F    db    0
EAX .text:0043DE10    function_encryption_key db 'þš',18h,'N@',81h,'9„?©œE”=',0,0,0,0
.text:0043DE22    dd 0
.text:0043DE26    dd 0
.text:0043DE2A    a70 db '70',0
.text:0043DE2D    db    0
.text:0043DE2E    aHqa4klmvs8wdkl db 'hqA4KLmVs8WdKLm',0
.text:0043DE3E    aKisdklm76lin db 'KiSdKLm76LIn',0
.text:0043DE4B    aHqanvqzmykwdkl db 'hqAnvqzmykWdKLm',0
.text:0043DE5B    aBysqbrtesv576l db 'BYSqBRTesV576LIn',0
.text:0043DE6C    aF3bx db 'F3BX',0
.text:0043DE71    aSf db 'sF',0
.text:0043DE74    aSu db 'su',0
.text:0043DE77    aHp63ylhvvqw4al db 'hP63yLHVvQW4aLO',0
.text:0043DE87    aRz6vfqze6j db 'Rz6VFqzE6J',0
.text:0043DE92    aHouxkpn4bewdkl db 'hoUXKPN4BEWdKLm',0
.text:0043DEA2    aVg3yll3yewdklm db 'vG+3yLl3yEWdKLm',0
.text:0043DEB2    a6lniyllnhqwdkl db '6LNiyLlnhQWdKLm',0
.text:0043DEC2    aVpt4hptwrgsnkj db 'vPT4hPTWrGSnKJ',0
.text:0043DED1    aBrterqix6ewdkl db 'BRTeRqIx6EWdKLm',0
.text:0043DEE1    aOqne6osnkqwdkl db 'OqNe6OSnKQWdKLm',0
.text:0043DEF1    aOpank8wdklm db 'OPANK8WdKLm',0
.text:0043DEFD    a6l4erpvavlutrg db '6L4ERPvAvLUtrGSnKJ',0
.text:0043DF10    aOqbergsnkj db 'OqBErGSnKJ',0
.text:0043DF1B    aBqDvinxsv576li db 'Bq+dviNXsV576LIn',0
.text:0043DF2C    aHqw0yln76lin db 'hqW0yLn76LIn',0
.text:0043DF39    aGluzscvz db 'glUzScvz',0
.text:0043DF42    aKollhqle db 'kOllhqlE',0
.text:0043DF4B    aOxhllzvtodlhgy db 'OXHLlzvTOdlhgY43hGHVKq6XRzveKGSxvPsug4ShFPlEhGl7vz64hiUeKqWh',0
.text:0043DF88    aOonx6ol3vcwaky db 'OoNx6ol3vcWAKYO',0
.text:0043DF98    aSr6akolavl4xku db 'SR6AKolAvL4xKu',0
.text:0043DFA7    aOxhllzvtodlhgy_0 db 'OXHLlzvTOdlhgY43hGHVKq6XRz6ehiS+BYmugYz3yL476F',0
.text:0043DFD6    aAvuzuo62uxbwrg db 'aVUzUO62UXBWrg4TUgc1U2sqUEXWs2BVrOcIs3JEU2SLFdl2UPX',0
.text:0043E00A    aAv6zscfqs2hxro db 'aV6zScFqS2hXrOsms2h1UcOPUk+QUV6TrOO+UVFmjgd+Sg5XFPX',0
.text:0043E03E    a6ramklhe6r576r db '6RAmKLHE6R576RA4',0
.text:0043E04F    aBgixkqsw5lnxkp db 'BGIxKqSW5LNxKPSW5LNnBkTd6kT8KoldakT8KPSXakT8KLcuKLA45LUAhL4XBY476'
.text:0043E04F    db 'kT8KLHx6LOA',0
```

In this sample, the key is equal to FE9A184E408139843FA99C45943D

Detricking

All we really have to do is iterate over all functions, decrypt their body with xor and mark the functions.

Wrapper function

As seen in previous screenshots, all function calls are performed using a function wrapper that:

- Accepts index of the function to execute
- Grabs the function's address from the global table
- Decrypts the function code
- Calls the decrypted function
- Encrypts the function code back again

Example function wrapper call

Detricking

In order to simplify our analysis we'll patch the binary and replace the wrapper calls with direct function calls.

Almost every wrapper call is exactly the same, which will be very helpful:

XX is a single unsigned byte that determines the index of the wrapped function.

YY YY YY YY is a 32-bit, relative, little-endian integer that marks the address of the wrapper function.

Our plan is to patch the whole call blob to:

where ZZ ZZ ZZ ZZ is the relative address of the wrapped function.

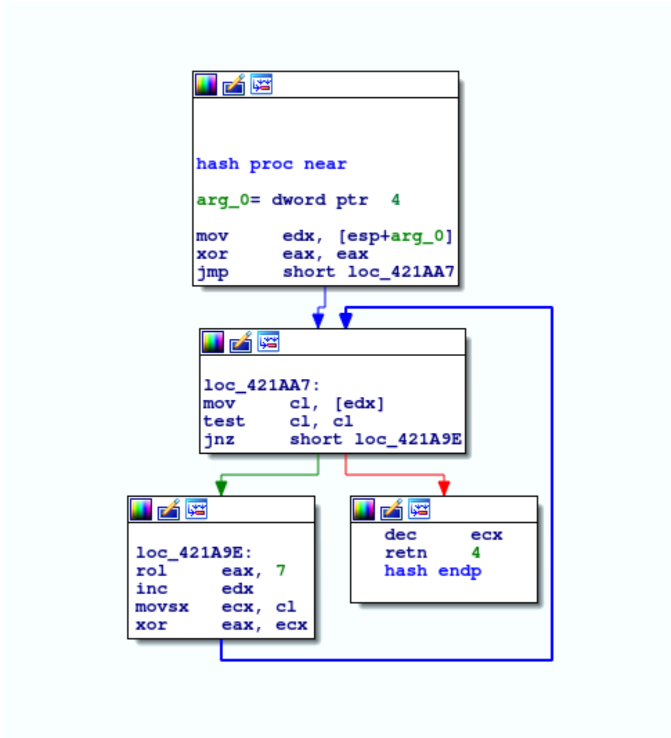
To do that, we'll use an idapython script:

Before:

After:

Imports

All imports are loaded into a static location in memory using a hash lookup:



Function used to calculate strings hash

C decompilation:

```
.text:0043E95C 00                                              db    0
.text:0043E95D 00                                              db    0
.text:0043E95E 00                                              db    0
.text:0043E95F 00                                              db    0
.text:0043E960 00                                              db    0
.text:0043E961 00                                              db    0
.text:0043E962 00                                              db    0
.text:0043E963                          ; int dword_43E963[2]
.text:0043E963 00 00 00 00 00 00 00 00 dword_43E963     dd 2 dup(0)              ; DATA XREF: main_thingy+2F↑w
.text:0043E963                                                                          ; get_decrypted_string_0+3↑r ...
.text:0043E96B 00 02 00 00             function_hashes dd 200h
.text:0043E96F 19 2B 90 95                              dd 95902B19h
.text:0043E973 F5 72 99 3D                              dd 3D9972F5h
.text:0043E977 52 01 26 69                              dd 69260152h
.text:0043E97B 54 73 80 68                              dd 68807354h
.text:0043E97F 1A 73 B0 0F                              dd 0FB0731Ah
.text:0043E983 1A 06 FE 08                              dd 8FE061Ah
.text:0043E987 8B BD 10 1A                              dd 1A10BD8Bh
.text:0043E98B D1 8A 31 46                              dd 46318AD1h
.text:0043E98F 05 AD 89 0D                              dd 0D89AD05h
.text:0043E993 5F 70 35 3A                              dd 3A35705Fh
.text:0043E997 0E 7F 86 86                              dd 86867F0Eh
.text:0043E99B 7C 1C 7A 40                              dd 407A1C7Ch
.text:0043E99F EE EA C0 1F                              dd 1FC0EAEEh
.text:0043E9A3 FE 6A 7A 69                              dd 697A6AFEh
.text:0043E9A7 26 80 AC C8                              dd 0C8AC8026h
.text:0043E9AB 8A BD 10 15                              dd 1510BD8Ah
.text:0043E9AF 5A 6F DE A9                              dd 0A9DE6F5Ah
.text:0043E9B3 D5 B0 3E 72                              dd 723EB0D5h
.text:0043E9B7 BE 24 B6 74                              dd 74B624BEh
.text:0043E9BB 1F F1 5E 37                              dd 375EF11Fh
.text:0043E9BF 2E 49 A5 3F                              dd 3FA5492Eh
.text:0043E9C3 1B F1 E4 2E                              dd 2EE4F11Bh
.text:0043E9C7 FE 93 43 77                              dd 774393FEh
.text:0043E9CB 41 E7 5B 51                              dd 515BE741h
.text:0043E9CF 70 F3 A5 2C                              dd 2CA5F370h
.text:0043E9D3 F0 B5 A1 2C                              dd 2CA1B5F0h
.text:0043E9D7 F0 B8 40 2D                              dd 2D40B8F0h
.text:0043E9DB 61 35 07 0A                              dd 0A073561h
.text:0043E9DF 74 67 8D A4                              dd 0A48D6774h
.text:0043E9E3 AC 91 EF 3D                              dd 3DEF91ACh
.text:0043E9E7 68 0C B0 78                              dd 78B00C68h
.text:0043E9EB 7D 81 54 80                              dd 8054817Dh
.text:0043E9EF 5C 37 A1 49                              dd 49A1375Ch
.text:0043E9F3 02 F1 F8 08                              dd 8F8F102h
.text:0043E9F7 C3 D1 3F 0F                              dd 0F3FD1C3h
.text:0043E9FB 32 0E 48 9C                              dd 9C480E32h
.text:0043E9FF A1 87 55 47                              dd 475587A1h
.text:0043EA03 FB E9 E4 20                              dd 20E4E9FBh
.text:0043EA07 C9 F0 F0 81                              dd 81F0F0C9h
.text:0043EA0B 42 A8 6F 9E                              dd 9E6FA842h
```

Function hash list

Detricking

We can find the correct API function table using different methods but we are going to focus on doing it manually by looking for the correct function name.

Start off by rewriting the hash function to Python:

We'll also need a list of functions exported by windows DLLs. We've found that scraping http://www.win7dll.info/ actually works pretty well for that purpose.

Now we need to iterate over all hashes and find a correct function name for each one:

All that's left now is to create an IDA struct that contains the function names and set the global array to the proper type:

```
int __cdecl terminate_process(char *proc_name)
{
  int v2; // [esp+0h] [ebp-23Ch]
  int i; // [esp+4h] [ebp-238h]
  int v4; // [esp+8h] [ebp-234h]
  int v5; // [esp+Ch] [ebp-230h]
  int v6; // [esp+14h] [ebp-228h]
  int v7; // [esp+30h] [ebp-20Ch]

  v4 = (*(&api + 42))(15, 0);
  v5 = 556;
  for ( i = (*(&api + 40))(v4, &v5); i; i = (*(&api + 41))(v4, &v5) )
  {
    if ( !(*(&api + 23))(&v7, proc_name) )
    {
      v2 = (*(&api + 43))(1, 0, v6);
      if ( v2 )
      {
        (*(&api + 39))(v2, 9);
        (*(&api + 17))(v2);
      }
    }
  }
  return (*(&api + 17))(v4);
}
```

Before

```
int __cdecl terminate_process(char *proc_name)
{
  int v2; // [esp+0h] [ebp-23Ch]
  int i; // [esp+4h] [ebp-238h]
  int v4; // [esp+8h] [ebp-234h]
  int v5; // [esp+Ch] [ebp-230h]
  int v6; // [esp+14h] [ebp-228h]
  int v7; // [esp+30h] [ebp-20Ch]

  v4 = (api.kernel32_CreateToolhelp32Snapshot)(15, 0);
  v5 = 556;
  for ( i = (api.kernel32_Process32FirstW)(v4, &v5); i; i = (api.kernel32_Process32NextW)(v4, &v5) )
  {
    if ( !(api.kernel32_lstrcmpiW)(&v7, proc_name) )
    {
      v2 = (api.kernel32_OpenProcess)(1, 0, v6);
      if ( v2 )
      {
        (api.kernel32_TerminateProcess)(v2, 9);
        (api.kernel32_CloseHandle)(v2);
      }
    }
  }
  return (api.kernel32_CloseHandle)(v4);
}
```

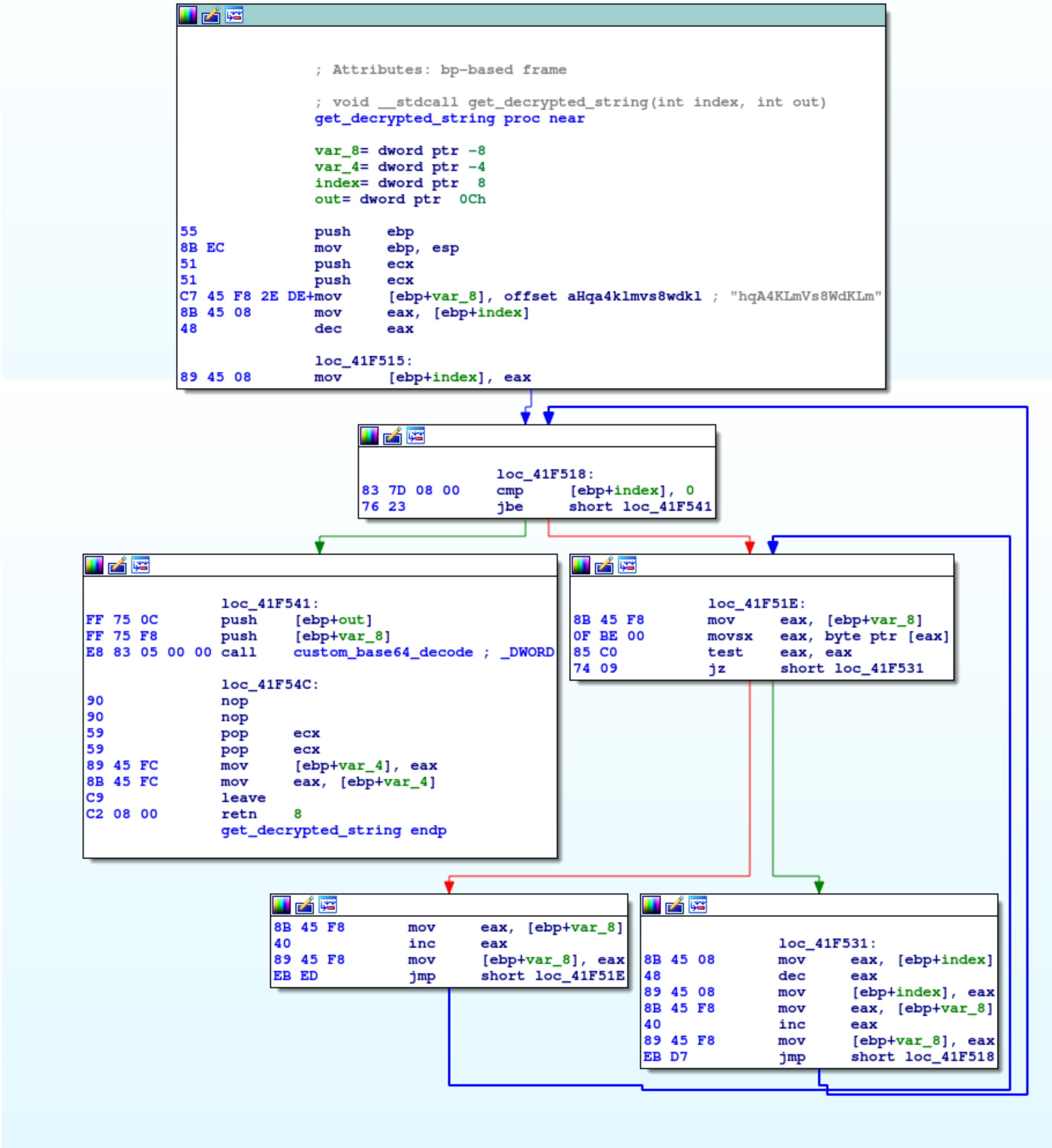
After

Now, it looks much better!

String encoding

All strings are encoded using base64 with a custom alphabet, it's explained pretty well in several blog posts already [23]

The custom charset is a permutation of the default base64 charset, e.g.
JTQ2czLo5NfrsUjZFSkgOlYRB6yKhva/uA83d4GiteMwn17xmIEVX+qP0W9DbHCp.



Function used to fetch a decrypted base64 string with a given index

Detricking

After de-wrapping the function calls, the assembly actually looks quite similar to the previous iteration (notice the *nops* that are result of our earlier patches):

Which means we can reuse some of our previous code. But instead of patching the call instructions to mov instructions, we're just going to add comments in assembly to annotate the original string:

Overview

After applying all of the described anti-anti-analysis patches, we end up with a pretty decent-looking binary.

Main function:

Anti-debugging/sandbox checks

DLL checks

The binary iterates over DLL names stored in strings and checks if any of them is present in the PEB InMemoryOrderModuleList linked list:

DLLs checked:

- pstorec.dll
- vmcheck.dll
- dbghelp.dll
- wpespy.dll
- api_log.dll
- SbieDll.dll
- SxIn.dll
- dir_watch.dll
- Sf2.dll
- cmdvrt32.dll
- snxhk.dll

Antimalware services

A series of checks is performed using QueryServiceStatusEx in order to detect any anti-malware services currently running on the system. If a service is detected, the loader tries to disable it accordingly:

- WinDefend
  - cmd.exe /c sc stop WinDefend
  - cmd.exe /c sc delete WinDefend
  - TerminateProcess MsMpEng.exe
  - TerminateProcess MSASCuiL.exe
  - TerminateProcess MSASCui.exe
  - cmd.exe /c powershell Set-MpPreference -DisableRealtimeMonitoring $true
  - RegSetValue SOFTWARE\Policies\Microsoft\Windows Defender DisableAntiSpyware
  - RegSetValue SOFTWARE\Microsoft\Windows Defender Security Center\Notifications DisableNotifications
- MBAMService

  ControlService MBAMService SERVICE_CONTROL_STOP
- SAVService
  - TerminateProcess SavService.exe
  - TerminateProcess ALMon.exe
  - cmd.exe /c sc stop SAVService
  - cmd.exe /c sc delete SAVService
  - Checks IEFO[4]. key for ‘MBAMService’,’SAVService’,’SavService.exe’,’ALMon.exe’,’SophosFS.exe’,’ALsvc.exe’,’Clean.exe’,’SAVAdminService.exe’ and sets Debugger registry key to kjkghuguffykjhkj if a match is found

Loading binary

The binaries embedded in the loader are encrypted using the same xor cipher method as the functions, however they are also compressed using MiniLZO [2].

The methods of executing the payload differ for 32 and 64-bit binaries. While the former is pretty straight-forward, the latter integrated a more sophisticated code injection technique.

Firstly, a new suspended process is created (in this sample with process name equal to “svchost”), then the execution transfers to a dynamically-generated shellcode that performs a switch from 32-bit compatibility mode to 64-bit using a trick called Heaven’s Gate[5]. Finally, the shellcode performs a call to the decrypted 64-bit helper shellcode which then finally jumps to the 64-bit core.

The included shellcode deassembles to

Modules

As of today, TrickBot is distributing following modules:

- **domainDll32.dll**

  bf50566d7631485a0eab73a9d029e87b096916dfbf07df4af2069fc6eb733183
- **importDll32.dll**

  f9ebf40d1228fa240c64d86037f2080588ed67867610aa159b80a553bc55edd7

- **injectDll32.dll**
  a515f4f847e8d7b2eb46a855224c8f0e9906435546bb15785b6770f2143bc22a
- **mailsearcher32.dll**
  46706124d4c65111398296ea85b11c57abffbc903714b9f9f8618b80b49bb0f3
- **networkDll32.dll**
  c8c789296cc8219d27b32c78e595d3ad6ee1467d2f451f627ce96782a9ff0c5f
- **outlookDll32.dll**
  9a529b2b77c5c8128c4427066c28ca844ff8ebbd8c3b2da27b8ea129960f861b
- **pwgrab32.dll**
  fe0f269a1b248c919c4e36db2d7efd3b9624b46f567edd408c2520ec7ba1c9e4
- **shareDll32.dll**
  af5ee15f47226687816fc4b61956d78b48f62c43480f14df5115d7e751c3d13d
- **squlDll32.dll**
  b8b757c2a3e7ae5bb7d6da9a43877c951fb60dcb606cc925ab0f15cdf43d033b
- **systeminfo32.dll**
  dff1c7cddd77b1c644c60e6998b3369720c6a54ce015e0044bbbb65d2db556d5
- **tabDll32.dll**
  479aa1fa9f1a9af29ed010dbe3b080359508be7055488f2af1d4b10850fe4efc
- **wormDll32.dll**
  627a9eb14ecc290fe7fb574200517848e0a992896be68ec459dd263b30c8ca48

References

[1] https://blog.malwarebytes.com/threat-analysis/2016/10/trick-bot-dyrezas-successor/

[1] https://sysopfb.github.io/malware/2018/04/16/trickbot-uacme.html

[2] https://blog.malwarebytes.com/threat-analysis/malware-threat-analysis/2018/11/whats-new-trickbot-deobfuscating-elements/

[4] https://blog.malwarebytes.com/101/2015/12/an-introduction-to-image-file-execution-options/

[5] http://rce.co/knockin-on-heavens-gate-dynamic-processor-mode-switching/