Analyzing Qbot obfuscation techniques

Task

Please provide a printed out written description of the obfuscation techniques used to obfuscate Qbot (aka. Qakbot) **main module** malware samples, and if possible describe ways to unpack and/ or de-obfuscate samples. Please provide screenshots of central elements of your analysis.

Tools

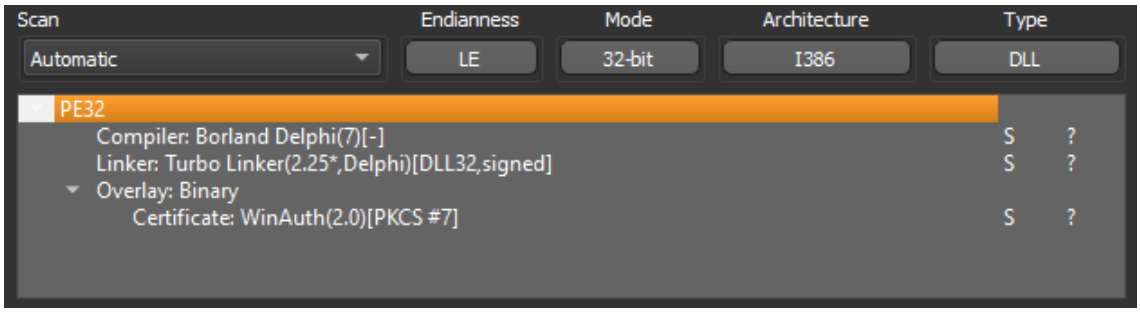
- **DiE (Detect It Easy)**
- **PEStudio**
- **CFF Explorer**
- **Resource Hacker**
- **IDA Pro**
- **x64dbg**
- **HxD**

Sample

Retrieved from: MalwareBazaar Database (https://bazaar.abuse.ch)
File: DLL
Size: 1,841,599 bytes
First spotted: 2022-05-17 19:49:24 UTC
`SHA256: 8a383f890745370e6f256396858a94062600f1efd2d1df36ef8a291e41494277`

Static analysis

Before we start debugging the sample, let's look at the structure of the `PE-file`. Opening the file in **DiE**



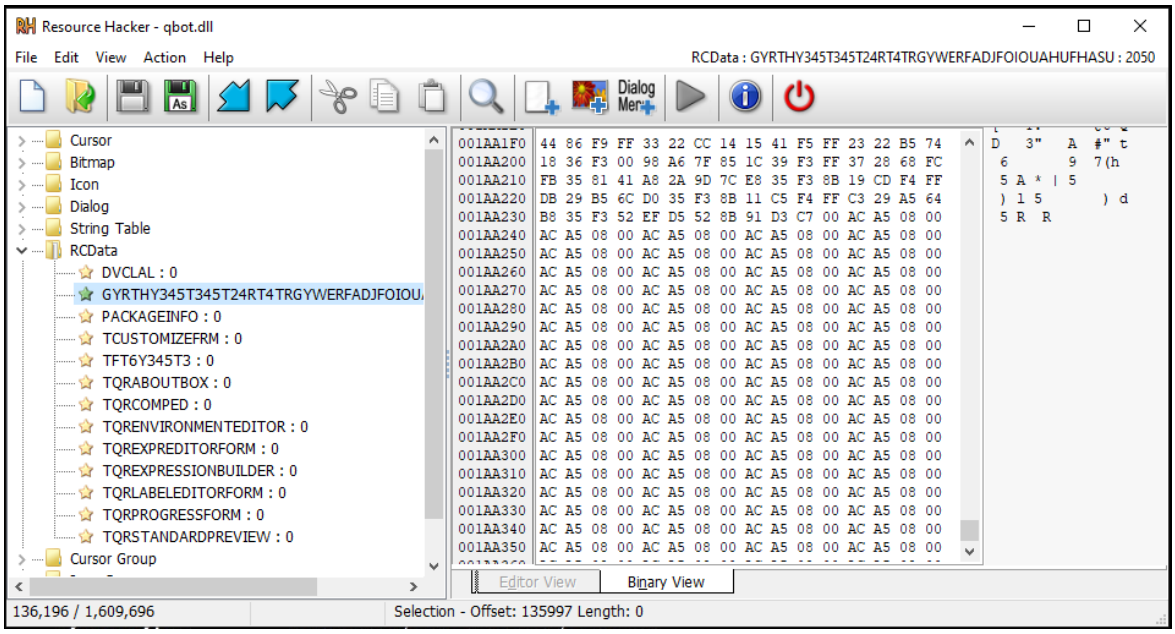
The screenshot shows that the file is 32-bit, `Delphi7` compiler. There is an overlay with a certificate in it. The certificate signature is invalid because it is not displayed in `Digital Signatures`. Sometimes attackers use the certificate body to store the pyload.

The PEStudio program does not show anything interesting, except for the scan results by hash from VirusTotal 48/70. Most antiviruses detect it as trojan.qbot/qakbot

Searching for cryptographic algorithms in the file found only PKCS_DigestDecoration_SHA256 at address offset 001bcc5eh. A fake certificate is located at this location.

In **CFF Explorer** I see that there are no export functions in the library. This means that Stub will create them at runtime when it replaces the headers with the headers of the original image in memory.

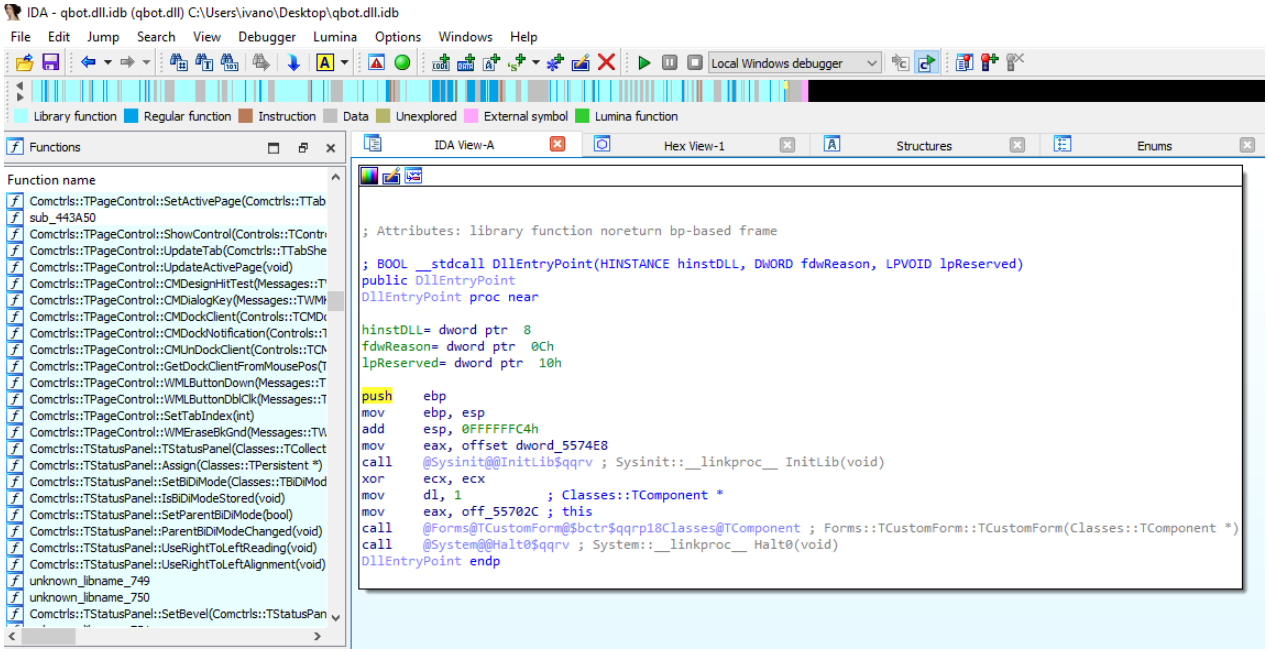
Next, I look at the resource data with the **Resource Hacker** program. We admire the fake data in the Version_Info section. The resource section is quite large and heavily littered. In the RCData section there is a suspicious resource with data and the name GYRTHY345T345T24RT4TRGYWERFADJFOIOUAHUFHASU.



Its size is 136Kb, and you can clearly see traces of the XOR algorithm at the end of the binary data, where zeros should be. Sometimes an addition or subtraction algorithm may be used instead, to make the XOR instruction less suspicious to antiviruses. Judging by the size, the original Qbot is there. We do not do anything about it for now and proceed to analyzing the code.

Code analysis

First let's see everything in the **IDA Pro** disassembler

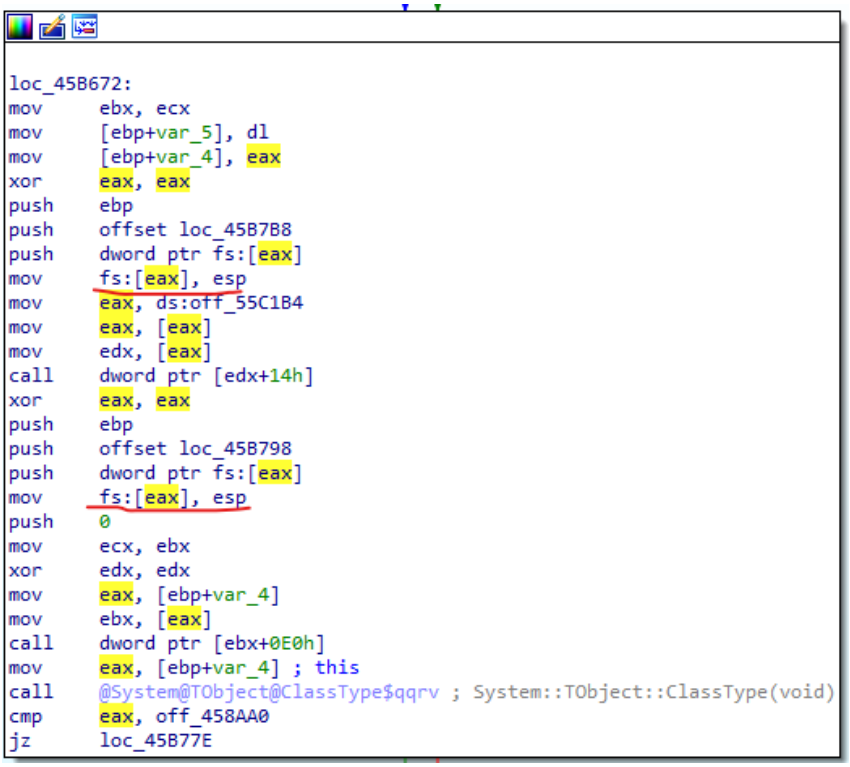


At the top of the graph we see that the CODE section has a large size. Almost all the code in it has been disassembled. There are almost no gray sections that may contain Payload.

In the left part we see the names of functions, these are Delphi classes and methods. Their names traditionally start with the letter T. The whole section is filled with code. Apparently, the source code of some `OpenSource` program taken from GitHub was used. This creates the illusion that the file is not a malware crypto. Modern antiviruses are suspicious of executable files with little code that is assembled.

The work of the obfuscation algorithm we will watch in the debugger at the real execution of sod, because the disassembler can not show the whole algorithm of actions.

Let's look at the beginning of `DllEntryPoint`. The first `CALL` is a `Delphi-Runtime` initialization function. This should be skipped as it is a standard function. Below that there is the code from chains-SEH.



```
loc_45B672:
mov     ebx, ecx
mov     [ebp+var_5], dl
mov     [ebp+var_4], eax
xor     eax, eax
push    ebp
push    offset loc_45B7B8
push    dword ptr fs:[eax]
mov     fs:[eax], esp
mov     eax, ds:off_55C1B4
mov     eax, [eax]
mov     edx, [eax]
call    dword ptr [edx+14h]
xor     eax, eax
push    ebp
push    offset loc_45B798
push    dword ptr fs:[eax]
mov     fs:[eax], esp
push    0
mov     ecx, ebx
xor     edx, edx
mov     eax, [ebp+var_4]
mov     ebx, [eax]
call    dword ptr [ebx+0E0h]
mov     eax, [ebp+var_4] ; this
call    @System@TObject@ClassType$qqrv ; System::TObject::ClassType(void)
cmp     eax, off_458AA0
jz      loc_45B77E
```

Cryptmakers often start their actions in the body of the `FormCreate` procedure. In the Function window let's find all functions that contain this name.

Function name	Segment	Start
_TQRStandardPreview_FormCreate	CODE	004C5EF0
Customizedlg::TCustomizeFrm::FormCreate(System::TObject *)	CODE	005564C4
_Tft6y345t3_FormCreate	CODE	005571B0

Let's go through each of them and set a breakpoint and start the debugger. Stop at the first Breakpoint and go to the pseudocode

```
LPSTR __fastcall Tft6y345t3_FormCreate(int a1)
{
  LPSTR result; // eax
  int v2; // edx
  int v3; // eax
  int v4; // ebx
  int v5; // ebx
  int TextCharset; // eax
  LPSTR lpBuffer; // [esp+10h] [ebp-6Ch]
  DWORD nSize; // [esp+18h] [ebp-64h] BYREF
  unsigned int v9; // [esp+1Ch] [ebp-60h]
  DWORD v10; // [esp+20h] [ebp-5Ch]
  int i; // [esp+24h] [ebp-58h]
  int v12; // [esp+28h] [ebp-54h]
  int v13; // [esp+2Ch] [ebp-50h]
  unsigned int v14; // [esp+30h] [ebp-4Ch]
  unsigned int v15; // [esp+34h] [ebp-48h]
  int v16; // [esp+38h] [ebp-44h]
  int v17; // [esp+3Ch] [ebp-40h]
  int v18; // [esp+40h] [ebp-3Ch]
  int v19; // [esp+44h] [ebp-38h]
  unsigned int v20; // [esp+48h] [ebp-34h]
  int v21; // [esp+4Ch] [ebp-30h]
  int v22; // [esp+50h] [ebp-2Ch]
  void *v23; // [esp+54h] [ebp-28h]
  int v24; // [esp+58h] [ebp-24h]
  int *v25; // [esp+5Ch] [ebp-20h]
  int (__fastcall ***v26)(_DWORD); // [esp+60h] [ebp-1Ch]
  int v27; // [esp+64h] [ebp-18h]
  int v28; // [esp+68h] [ebp-14h]
  int v29; // [esp+6Ch] [ebp-10h]
  int v30; // [esp+70h] [ebp-Ch]
  unsigned int *v31; // [esp+74h] [ebp-8h]
  unsigned int *v32; // [esp+78h] [ebp-4h]

  v27 = a1;
  lpBuffer = (LPSTR)System::__linkproc__ GetMem(100);
  nSize = 100;
  GetUserNameA(lpBuffer, &nSize);
  if ( lpBuffer[1] != 111 || lpBuffer[4] != 68 || (result = lpBuffer, lpBuffer[6] != 101) )
  {
    LoadLibraryA_0("jr3");
    v12 = 566667;
    v32 = &STACK[0x22C];
    v31 = &STACK[0x254];
    v30 = (int)*(&hInstance - 4);
    v29 = (int)*(&hInstance - 5);
    LOBYTE(v2) = 1;
    v26 = (int (__fastcall ***)(_DWORD))unknown_libname_424(
                                          (Classes::TResourceStream *)&off_41C008,
                                          v2,
                                          hInstance,
                                          (LPCSTR)&str_gyrthy345t345t2[1],
                                          10);
    for ( i = 14; i != 23555534; ++i )
      GetTextCharset(0);
    v12 = 566667;
    v17 = 121;
    v9 = (**v26)(v26);
```

You can see in the listing that the computer name is being checked. The first and fourth and sixth characters ".o..D.e". If all of them match, the job is terminated. I can't guess what the name should be. Probably the computer name of the crypt author himself. Since all normal sanboxes do not use static names and such a simple way to determine them will not work.

Next comes garbage calls and fecal code. The `TResourceStream` class accesses a resource named `GYRTHY345T345T345T24RT4TRGYWERFADJFOIOUAHUFHASU` and, in fact, allocates memory by copying its contents there. This is Payload.

The `GetTextCharset(0);` function from `WinApi` library *Gdi32.dll* is launched in a long loop. This is done to stop code emulation by antivirus. To avoid getting stuck here and this long loop not being executed, let's change `jnz short loc_55727B` to `jz short loc_55727B`.

I'll rename some variables and add comments to make the code in the screenshot clearer.

Further down the code are the calls of the `VirtualAlloc` function `PAGE_EXECUTE_READWRITE`. They allocate two sections equal in size `02147Dh`. One of which will not be used. The second one is filled with zeros by `Windows::FillMemory(pNewMemory2, dw, 0)`. And Payload is copied from `TResourceStream`.

```
76    Windows::FillMemory(pNewMemory2, dw, 0);
77    size = (**(int (__fastcall ***)(Classes::TResourceStream *))pTResourceStream)(pTResourceStream);
78    n1 = 0;
79    while ( count < size )                        // Copy Memory
80    {
81      v24 = (int)pNewMemory2 + i;
82      v23 = (int)NewMemory1 + count;
83      sub_B07A40((char *)pNewMemory2 + i, (char *)NewMemory1 + count, v19);// System::Move
84      i += v19;
85      count += v19;
86      count += v18;
87    }
88    v21 = 137104;
89    v30 = v26 + 137104;
90    n1 = 0;
91    if ( v22 )
92    {
93      do                                          // Decrypt Algoritm
94      {
95        v12 = v14 + n1 - 1;
96        *(_DWORD *)pNewMemory2 += n1;
97        v4 = v12 + v20 + GetTextCharset(0);
98        dw = *(_DWORD *)pNewMemory2 ^ (v4 - GetTextCharset(0));// XOR
99                                                  //
100       *(_DWORD *)pNewMemory2 = dw;        int v4; // ebx
101       v12 = 0;                            0x8A5AD
102       v5 = n1 + 4 + GetTextCharset(0);
103       TextCharset = GetTextCharset(0);
104       pNewMemory2 = (char *)pNewMemory2 + v12 + 4;
105       n1 = v5 - TextCharset;
106     }
107     while ( v5 - TextCharset < v22 );
108   }
109   v30 += 4;
110   v30 -= 4096;
111   __asm { jmp     [ebp+var_14] }                // Unconditional jump to decrypted Stub in memory
112 }
113 return result;
114 }
001567C8 _Tft6y345t3_FormCreate:98 (C573C8)
```

As we can see the decryption procedure uses XOR with floating key. The algorithm of key generation includes manipulations with WinApi function, getting an unknown number to fool emulators. The assembler insertion at the end has a `JMP` on the memory section where the decrypted `Stub` lies. Further we will continue studying the code in **x64dbg** debugger

At the beginning of the memory section is the original `Qbot` file. Some WinApi function names are encoded directly in the code.

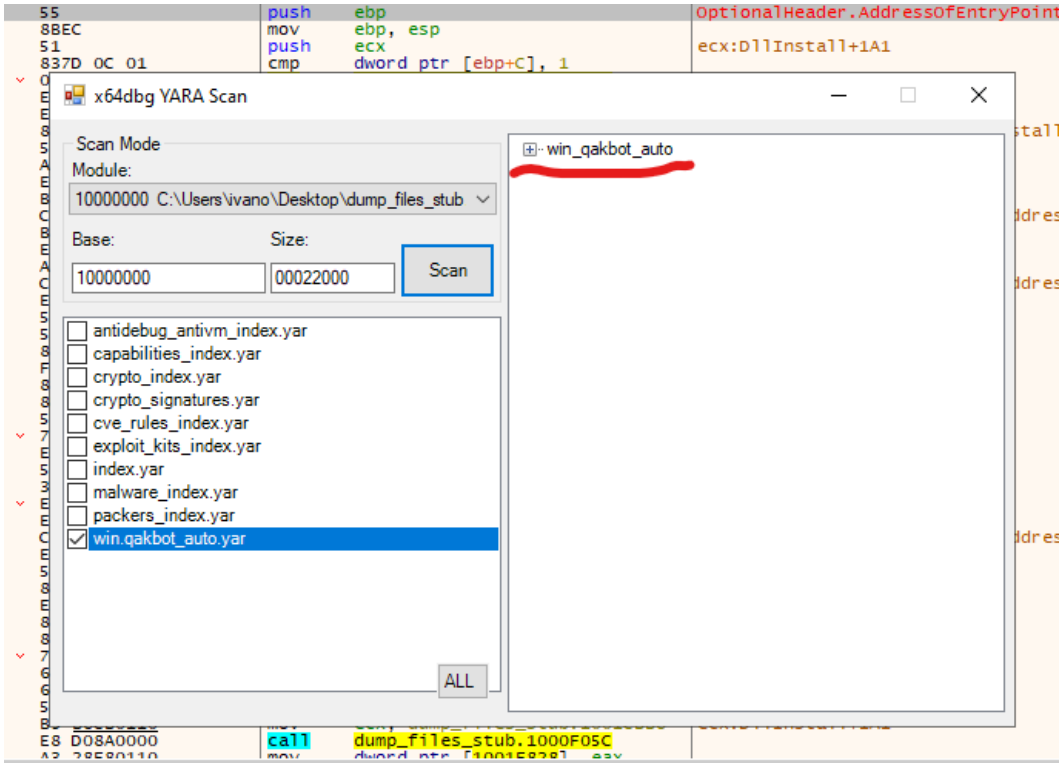
```
mov     byte ptr [ebp-28], 47           47:'G'
mov     byte ptr [ebp-27], 65           65:'e'
mov     byte ptr [ebp-26], 74           74:'t'
mov     byte ptr [ebp-25], 50           50:'P'
mov     byte ptr [ebp-24], 72           72:'r'
mov     byte ptr [ebp-23], 6F           6F:'o'
mov     byte ptr [ebp-22], 63           63:'c'
mov     byte ptr [ebp-21], 41           41:'A'
mov     byte ptr [ebp-20], 64           64:'d'
mov     byte ptr [ebp-1F], 64           64:'d'
mov     byte ptr [ebp-1E], 72           72:'r'
mov     byte ptr [ebp-1D], 65           65:'e'
mov     byte ptr [ebp-1C], 73           73:'s'
mov     byte ptr [ebp-1B], 73           73:'s'
mov     byte ptr [ebp-1A], 0
```

On the "Memory Card" tab, select the desired line and save the memory dump to disk. Viewing in the **HxD** editor

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   47 65 74 50 72 6F 63 41 64 64 72 65 73 73 00 00  GetProcAddress..
00000010   00 00 00 56 69 72 74 75 61 6C 41 6C 6C 6F 63 00  ...VirtualAlloc.
00000020   00 00 00 00 00 00 56 69 72 74 75 61 6C 46 72 65  ......VirtualFre
00000030   65 00 00 00 00 00 00 00 00 55 6E 6D 61 70 56 69  e........UnmapVi
00000040   65 77 4F 66 46 69 6C 65 00 00 00 00 56 69 72 74  ewOfFile....Virt
00000050   75 61 6C 50 72 6F 74 65 63 74 00 00 00 00 00 4C  ualProtect.....L
00000060   6F 61 64 4C 69 62 72 61 72 79 45 78 41 00 00 00  oadLibraryExA...
00000070   00 00 47 65 74 4D 6F 64 75 6C 65 48 61 6E 64 6C  ..GetModuleHandl
00000080   65 41 00 00 00 00 47 65 74 4D 6F 64 75 6C 65 48  eA....GetModuleHa
00000090   6E 64 6C 65 57 00 00 00 43 72 65 61 74 65 46 69  ndleW...CreateFi
000000A0   6C 65 41 00 00 00 00 00 00 00 00 53 65 74 46 69  leA........SetFi
000000B0   6C 65 50 6F 69 6E 74 65 72 00 00 00 00 00 57 72  lePointer.....Wr
000000C0   69 74 65 46 69 6C 65 00 00 00 00 00 00 00 00 00  iteFile.........
000000D0   00 43 6C 6F 73 65 48 61 6E 64 6C 65 00 00 00 00  .CloseHandle....
000000E0   00 00 00 00 47 65 74 54 65 6D 70 50 61 74 68 41  ....GetTempPathA
000000F0   00 00 00 00 00 00 00 6C 73 74 72 6C 65 6E 41 00  .......lstrlenA.
00000100   00 00 00 00 00 00 00 00 00 00 6C 73 74 72 63 61  ..........lstrca
00000110   74 41 00 00 00 00 00 00 00 00 00 00 00 47 65 74  tA...........Get
00000120   4D 6F 64 75 6C 65 46 69 6C 65 4E 61 6D 65 41 00  ModuleFileNameA.
00000130   47 65 74 4D 6F 64 75 6C 65 46 69 6C 65 4E 61 6D  GetModuleFileNam
00000140   65 57 00 46 72 65 65 4C 69 62 72 61 72 79 00 00  eW.FreeLibrary..
00000150   00 00 00 00 00 00 00 01 00 00 00 08 00 00 00     ...............
00000160   02 00 00 00 04 00 00 00 10 00 00 00 80 00 00 00  ............€...
00000170   20 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00   ...@...........
00000180   00 F6 01 00 4D 5A 90 00 03 00 00 00 04 00 00 00  .ö..MZ..........
00000190   FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00  ÿÿ..¸.......@...
000001A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001C0   08 01 00 00 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C  ......º..´.Í!¸.L
000001D0   CD 21 54 68 69 73 20 70 72 6F 67 72 61 6D 20 63  Í!This program c
000001E0   61 6E 6E 6F 74 20 62 65 20 72 75 6E 20 69 6E 20  annot be run in
000001F0   44 4F 53 20 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00  DOS mode....$...
00000200   00 00 00 00 65 9B A1 E4 21 FA CF B7 21 FA CF B7  ....e›¡ä!úÏ·!úÏ·
00000210   21 FA CF B7 35 91 CB B6 23 FA CF B7 94 8F CE B6  !úÏ·5‘Ë¶#úÏ·”.Î¶
```

Find the beginning of the PE-file with the header signature "MZ" and delete everything at the beginning. At the end of the file we also need to trim the excess. To do this you need to define the end of the file. Open this dump in **CFF Explorer** on the `Section-Header section` and add the values of the last section `Raw_Address` and `Raw_Size` *to* get the file size.

The output is the original Qbot. You can make sure of it by checking Yara-rules, **x64dbg** has Plugin that allows to do it.



The signature has been identified! To be sure, you can also check on virustotal.com.

Stub allocates new space for the original again by ERW memory protection rights. Copies the original sections there. Configures access rights to each section using the `VirtualProtect` function. But for some reason the `flNewProtect` parameter has the wrong value `02D90168h` and the function does nothing and returns `LastError: ERROR_INVALID_PARAMETER`.

Next, `Import Data is` restored using the `LoadLibrary` and `GetProcAddress` functions. Since the code has been moved, it is restored using `Relocation Data.`

There is a check for the existence of the file `"C:\INTERNAL\__empty"` by the function `GetFileattributesW`.

```
mov     dword ptr [ebp+C], eax      [dword ptr ss:[ebp+0C]]:L"C:\\INTERNAL\\__empty"
call    dword ptr [<&GetFileAttributesW>
cmp     eax, FFFFFFFF
lea     eax, dword ptr [ebp+C]      [ss:[ebp+0C]]:L"C:\\INTERNAL\\__empty"
push    eax
je      2DB655A
```

This is used to check emulation of some antivirus product. Some sources say that it is Windows Defender. If the file exists, the control will not be transferred to the original-Qbot.

In the TEB structure, fixes the module data in LDR_DATA_TABLE_ENTRY.

```
0x18 DllBase;
0x1C EntryPoint;
0x20 SizeOfImage;
```

Substitutes them with the data of the new image.

```
push    ebp
mov     ebp, esp
sub     esp, C
mov     eax, dword ptr fs:[18]
mov     eax, dword ptr [eax+30]
mov     ecx, dword ptr [eax+C]
mov     dword ptr [ebp-C], ecx
mov     eax, dword ptr [ebp-C]
mov     ecx, dword ptr [eax+C]
mov     dword ptr [ebp-8], ecx
mov     edx, dword ptr [ebp-8]
mov     dword ptr [ebp-4], edx
mov     eax, 1
test    eax, eax
je      27C04BC
```

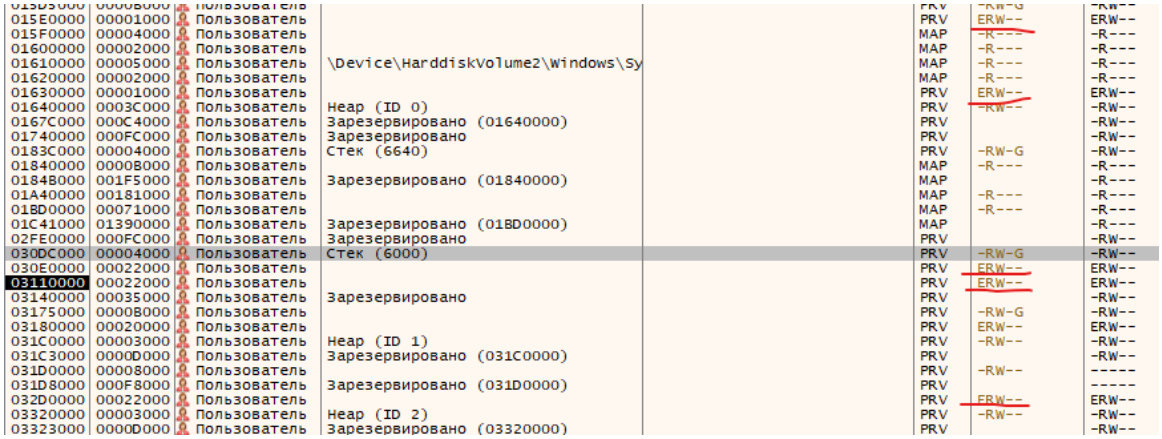
At the end, it restores the necessary values of the registers and puts the VA address of the entry point of the original file on the stack with the `push` instruction. It is followed by the address of return from `dllmain` to `ntdll.dl`. After the original `dllmain` works - the control will return to `ntdll.dll`, then to the `Rundll32` image and later it will be called by the `DllRegisterServer` function in Qbot. Then the malware itself starts working.

```
03131222    8B85 90FCFFFF    mov     eax, dword ptr [ebp-370]
03131228    8B8D 60FCFFFF    mov     ecx, dword ptr [ebp-3A0]
0313122E    8941 04          mov     dword ptr [ecx+4], eax
03131231    8B95 7CFCFFFF    mov     edx, dword ptr [ebp-384]
03131237    8BB5 68FCFFFF    mov     esi, dword ptr [ebp-398]
0313123D    8BBD 6CFCFFFF    mov     edi, dword ptr [ebp-394]
03131243    8BA5 60FCFFFF    mov     esp, dword ptr [ebp-3A0]
03131249    8BAD 64FCFFFF    mov     ebp, dword ptr [ebp-39C]
0313124F    52               push    edx
03131250    C3               ret
```

Conclusion

File obfuscation is done in a primitive way. No compression methods are applied. `Stub` is written in C\C++ and is not optimized. There is a lot of unnecessary duplicated code. For example, the function of getting the address of the `kernel32` module has just a gigantic size of 450 bytes. The author probably doesn't know about optimization parameters for the compiler. There are no anti-debugging techniques. There is no virtual machine definition either. Dense Payload data is not diluted in any way.

All sections of the original have incorrect protection rights to ERW memory. After Stub operation, a lot of unreleased `ERW memory` not belonging to any module is left behind. All this entails fast detection by anti-virus technologies.



015E0000	00001000	Пользователь			PRV	ERW--	ERW--
015F0000	00004000	Пользователь			MAP	-R---	-R---
01600000	00002000	Пользователь			MAP	-R---	-R---
01610000	00005000	Пользователь	\Device\HarddiskVolume2\Windows\Sy		MAP	-R---	-R---
01620000	00002000	Пользователь			MAP	-R---	-R---
01630000	00001000	Пользователь			PRV	ERW--	ERW--
01640000	0003C000	Пользователь	Heap (ID 0)		PRV	-RW--	-RW--
0167C000	000C4000	Пользователь	Зарезервировано (01640000)		PRV		-RW--
01740000	000FC000	Пользователь	Зарезервировано		PRV		-RW--
0183C000	00004000	Пользователь	Стек (6640)		PRV	-RW-G	-RW--
01840000	0000B000	Пользователь			MAP	-R---	-R---
0184B000	001F5000	Пользователь	Зарезервировано (01840000)		MAP		-R---
01A40000	00181000	Пользователь			MAP	-R---	-R---
01BD0000	00071000	Пользователь			MAP	-R---	-R---
01C41000	01390000	Пользователь	Зарезервировано (01BD0000)		MAP		-R---
02FE0000	000FC000	Пользователь	Зарезервировано		PRV		-RW--
030DC000	00004000	Пользователь	Стек (6000)		PRV	-RW-G	-RW--
030E0000	00022000	Пользователь			PRV	ERW--	ERW--
03110000	00022000	Пользователь			PRV	ERW--	ERW--
03140000	00035000	Пользователь	Зарезервировано		PRV		-RW--
03175000	0000B000	Пользователь			PRV	-RW-G	-RW--
03180000	00020000	Пользователь			PRV	ERW--	ERW--
031C0000	00003000	Пользователь	Heap (ID 1)		PRV	-RW--	-RW--
031C3000	0000D000	Пользователь	Зарезервировано (031C0000)		PRV		-RW--
031D0000	00008000	Пользователь			PRV	-RW--	-----
031D8000	000F8000	Пользователь	Зарезервировано (031D0000)		PRV		-----
032D0000	00022000	Пользователь			PRV	ERW--	ERW--
03320000	00003000	Пользователь	Heap (ID 2)		PRV	-RW--	-RW--
03323000	0000D000	Пользователь	Зарезервировано (03320000)		PRV		-RW--

If the task was to get the original Qbot from a crypto file, then everything can be done easier and faster. For example, put `BreakPoint` on `Virtualalloc`, `VirtualProtect` functions and monitor the places they allocate, wait for the original image to appear. Then saving the dump and correcting its section sizes is done automatically.