Flare-On 7: Challenge 4 – report.xls

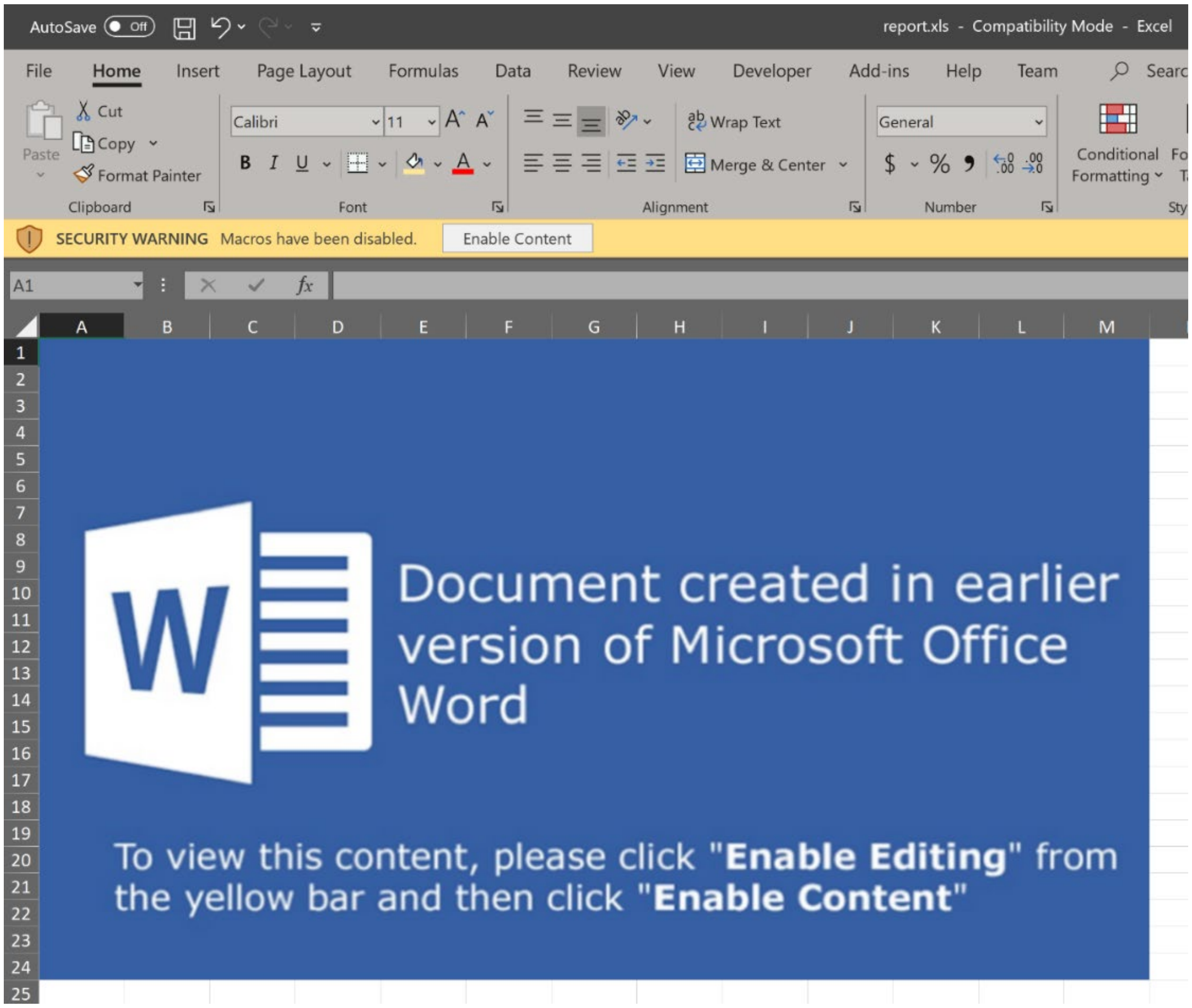
Challenge Author: Moritz Raabe (@m_r_tz)

Introduction

In this challenge we analyze a legacy Microsoft Excel document (OLE2 format). While it can be helpful to have a copy of the Microsoft Office Suite installed this analysis will rely on freely available tools – most of them are included in FLARE VM.

Basic Analysis

Figure 1 shows `report.xls` opened in Excel. We recognize a common social engineering technique to trick users into enabling Macros. The document contains an image that indicates users must “Enable Content” before they can access this file. When enabled the embedded Macros can run and perform malicious activity.



SECURITY WARNING Macros have been disabled.

Document created in earlier version of Microsoft Office Word

To view this content, please click "**Enable Editing**" from the yellow bar and then click "**Enable Content**"

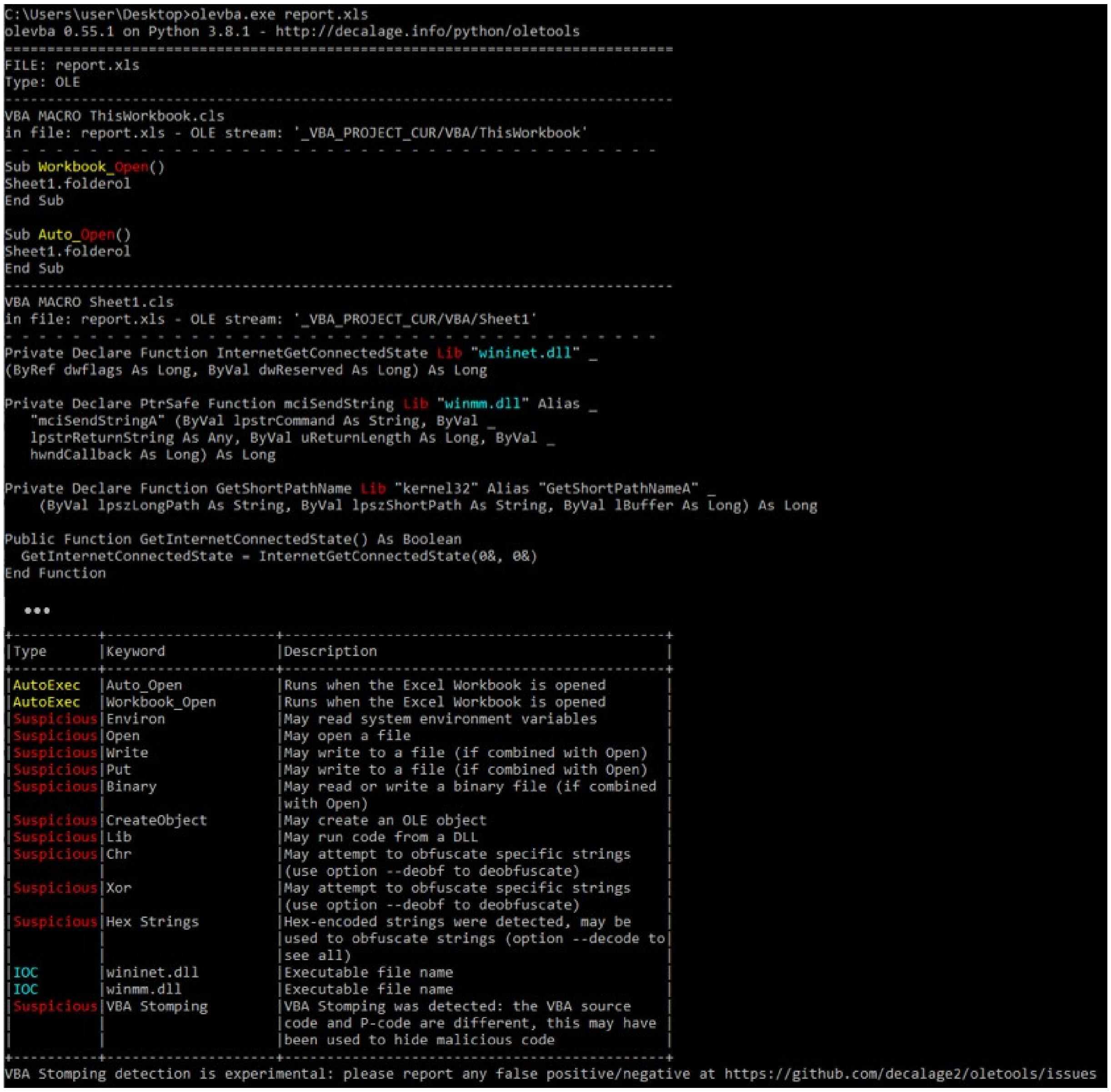
Figure 1: `report.xls` opened in Excel

As an alternative to Excel we can open the document in LibreOffice which also displays the social engineering image and indicates that this file contains Macros.

The following analysis focuses on the embedded Macro code.

MACRO EXTRACTION

A quick way to extract all Macros is to use `olevba` from Philippe Lagadec's `oletools`. Figure 2 shows a shortened `olevba` output. Besides Macro code the output includes p-code, VBA form string data, and an analysis results table.



```
C:\Users\user\Desktop>olevba.exe report.xls
olevba 0.55.1 on Python 3.8.1 - http://decalage.info/python/oletools
===============================================================================
FILE: report.xls
Type: OLE
-------------------------------------------------------------------------------
VBA MACRO ThisWorkbook.cls
in file: report.xls - OLE stream: '_VBA_PROJECT_CUR/VBA/ThisWorkbook'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sub Workbook_Open()
Sheet1.folderol
End Sub

Sub Auto_Open()
Sheet1.folderol
End Sub
-------------------------------------------------------------------------------
VBA MACRO Sheet1.cls
in file: report.xls - OLE stream: '_VBA_PROJECT_CUR/VBA/Sheet1'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Private Declare Function InternetGetConnectedState Lib "wininet.dll" _
(ByRef dwflags As Long, ByVal dwReserved As Long) As Long

Private Declare PtrSafe Function mciSendString Lib "winmm.dll" Alias _
   "mciSendStringA" (ByVal lpstrCommand As String, ByVal _
   lpstrReturnString As Any, ByVal uReturnLength As Long, ByVal _
   hwndCallback As Long) As Long

Private Declare Function GetShortPathName Lib "kernel32" Alias "GetShortPathNameA" _
    (ByVal lpszLongPath As String, ByVal lpszShortPath As String, ByVal lBuffer As Long) As Long

Public Function GetInternetConnectedState() As Boolean
  GetInternetConnectedState = InternetGetConnectedState(0&, 0&)
End Function

  •••
+----------+--------------------+---------------------------------------------+
|Type      |Keyword             |Description                                  |
+----------+--------------------+---------------------------------------------+
|AutoExec  |Auto_Open           |Runs when the Excel Workbook is opened       |
|AutoExec  |Workbook_Open       |Runs when the Excel Workbook is opened       |
|Suspicious|Environ             |May read system environment variables        |
|Suspicious|Open                |May open a file                              |
|Suspicious|Write               |May write to a file (if combined with Open)  |
|Suspicious|Put                 |May write to a file (if combined with Open)  |
|Suspicious|Binary              |May read or write a binary file (if combined |
|          |                    |with Open)                                   |
|Suspicious|CreateObject        |May create an OLE object                     |
|Suspicious|Lib                 |May run code from a DLL                      |
|Suspicious|Chr                 |May attempt to obfuscate specific strings    |
|          |                    |(use option --deobf to deobfuscate)          |
|Suspicious|Xor                 |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)                                     |
|IOC       |wininet.dll         |Executable file name                         |
|IOC       |winmm.dll           |Executable file name                         |
|Suspicious|VBA Stomping        |VBA Stomping was detected: the VBA source    |
|          |                    |code and P-code are different, this may have |
|          |                    |been used to hide malicious code             |
+----------+--------------------+---------------------------------------------+
VBA Stomping detection is experimental: please report any false positive/negative at https://github.com/decalage2/oletools/issues
```

Figure 2: Shortened `olevba` output showing Macro code and analysis results table

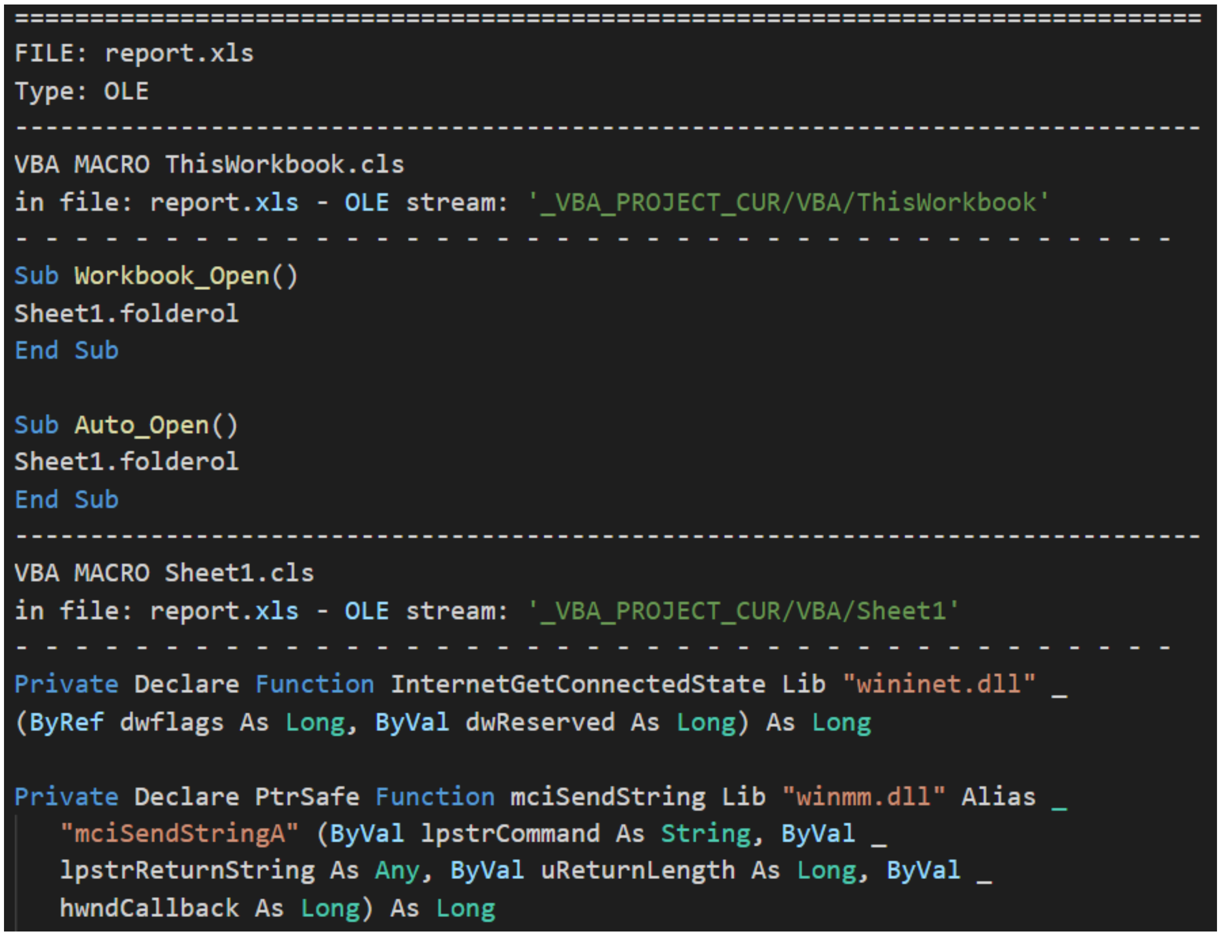
The results table summarizes detections based on `olevba`'s analysis heuristics. A very helpful heuristic that matched here is the VBA stomping detection. "VBA stomping is a colloquial term applied to the manipulation of Office documents where the source code of a Macro is made to mismatch the pseudo-code (hereto referred to as 'p-code') of the document" (see this blog post). More information about VBA stomping is available at https://vbastomp.com/. The output here includes the p-code because of this detection.

In typical malware samples that use VBA stomping the fake Macro code is often short and only performs benign tasks such as displaying a message box. While the extracted code here is short it looks a little suspicious. So, let's analyze it. We will look at the p-code afterwards.

Advanced Analysis of `report.xls`

To analyze the Macro, we copy the `olevba` output to a new text file. This is a workaround for known encoding issues you may encounter when redirecting output to a file or when using certain consoles.

Figure 3 shows the copied output in Visual Studio Code. `Workbook_Open` and `Auto_Open` are common event handlers to execute when a file is opened (and Macros are enabled). The handlers start the main function named `folderol` in the `Sheet1` class. This class implements all subsequent discussed functionality.



```
===============================================================================
FILE: report.xls
Type: OLE
-------------------------------------------------------------------------------
VBA MACRO ThisWorkbook.cls
in file: report.xls - OLE stream: '_VBA_PROJECT_CUR/VBA/ThisWorkbook'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sub Workbook_Open()
Sheet1.folderol
End Sub

Sub Auto_Open()
Sheet1.folderol
End Sub
-------------------------------------------------------------------------------
VBA MACRO Sheet1.cls
in file: report.xls - OLE stream: '_VBA_PROJECT_CUR/VBA/Sheet1'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Private Declare Function InternetGetConnectedState Lib "wininet.dll" _
(ByRef dwflags As Long, ByVal dwReserved As Long) As Long

Private Declare PtrSafe Function mciSendString Lib "winmm.dll" Alias _
   "mciSendStringA" (ByVal lpstrCommand As String, ByVal _
   lpstrReturnString As Any, ByVal uReturnLength As Long, ByVal _
   hwndCallback As Long) As Long
```

Figure 3: Copied `olevba` output in Visual Studio Code

FOLDEROL – MAIN FUNCTION

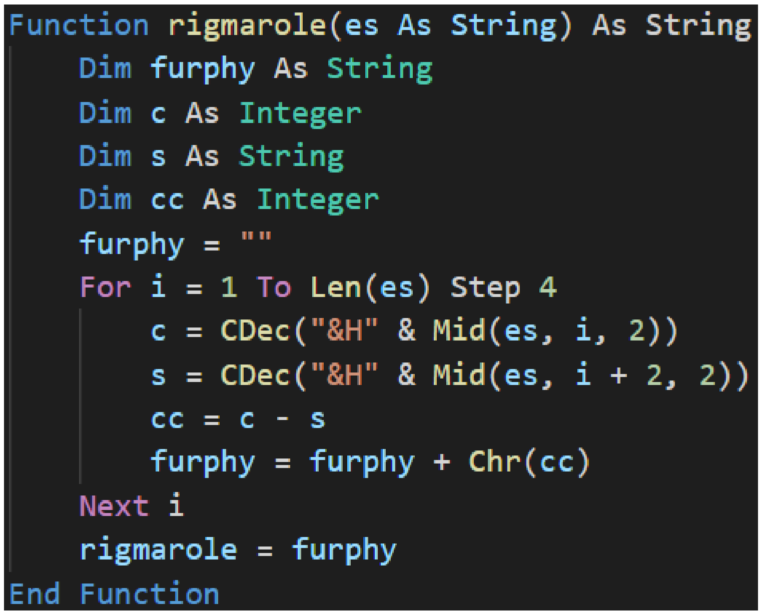
On a high-level this function:

- displays a message box and terminates depending on the current environment, for example if no Internet connection is detected;
- writes a binary file to disk; and
- calls the `mciSendString` Windows API

The function decodes strings at runtime using the `rigmarole` function. The encoded strings are obtained from the array named onzo. This array is obtained from splitting the form string `F.L` into its substrings separated by dot characters.

RIGMAROLE – STRING DECODING

Per the string decoding routine definition in Figure 4 we see that encoded strings consist of hex encoded characters. To decode an output character, the function subtracts two subsequent bytes from each other.



```
Function rigmarole(es As String) As String
    Dim furphy As String
    Dim c As Integer
    Dim s As String
    Dim cc As Integer
    furphy = ""
    For i = 1 To Len(es) Step 4
        c = CDec("&H" & Mid(es, i, 2))
        s = CDec("&H" & Mid(es, i + 2, 2))
        cc = c - s
        furphy = furphy + Chr(cc)
    Next i
    rigmarole = furphy
End Function
```

Figure 4: VBA code of `rigmarole` function

The following Python script decodes all strings. With a little knowledge of VBA, translating functionality to other programming languages is normally not a problem. Here it is done almost verbatim.

```
def rigmarole(es):
    furphy = ""
    for i in range(0, len(es), 4):
        c = int(es[i:i + 2], 0x10)
        s = int(es[i + 2:i + 2 + 2], 0x10)
        cc = c - s
        furphy += chr(cc)
    return furphy

F_L =
"9655B040B64667238524D15D6201.B95D4E01C55CC562C7557405A532D768C55FA12DD074DC697A06E172992CAF3F
8A5C7306B7476B38.C555AC40A7469C234424.853FA85C470699477D3851249A4B9C4E.A855AF40B84695239D24895
D2101D05CCA62BE5578055232D568C05F902DDC74D2697406D7724C2CA83FCF5C2606B547A73898246B4BC14E941F9
121D464D263B947EB77D36E7F1B8254.853FA85C470699477D3851249A4B9C4E.9A55B240B84692239624.CC55A940
B44690238B24CA5D7501CF5C9C62B1556105603 2C468D15F9C2DE374DD696206B572752C8C3FB25C3806.A85585409
24668236724B15D2101AA5CC362C2556A055232AE68B15F7C2DC17489695D06DB729A2C723F8E5C65069747AA38932
4AE4BB34E921F9421.CB55A240B5469B23.AC559340A94695238D24CD5D75018A5CB062BA557905A932D768D15F982
D.D074B6696F06D5729E2CAE3FCF5C7506AD47AC388024C14B7C4E8F1F8F21CB64"
for i, es in enumerate(F_L.split(".")):
    print("%2s: %s" % (i, rigmarole(es)))
```

Figure 5: Python code to decode strings

All decoded strings and their array index are shown in Figure 6.

```
 0: AppData
 1: \Microsoft\stomp.mp3
 2: play
 3: FLARE-ON
 4: Sorry, this machine is not supported.
 5: FLARE-ON
 6: Error
 7: winmgmts:\\.\root\CIMV2
 8: SELECT Name FROM Win32_Process
 9: vbox
10: WScript.Network
11: \Microsoft\v.png
```

Figure 6: Decoded strings obtained using the above Python script

Replacing all decoded strings in the VBA code reveals the use of Windows Management Instrumentation (WMI) for another environment check. The program obtains all running process names and terminates if one of them contains any of the substrings `vmw`, `vmt`, or `vbox`. This is a common anti-analysis technique.

If no offending process is found, the function writes data to the file `%AppData%\Microsoft\stomp.mp3` and passes its path to the `mciSendString` function. To recover the written file data, we analyze the function named `canoodle`.

CANOODLE – EMBEDDED DATA DECODING

This function receives four arguments; most notably the text value of a VBA form field (`F.T.Text`) and an array of 14 numbers. `canoodle` performs an XOR based decoding. The following script is a Python reimplementation of the function.

```
def canoodle(panjandrum, arylo, s, bibble):
    kerfuffle = bytearray()
    quean = 0
    for cattywampus in range(0, len(panjandrum), 4):
        b1 = int(panjandrum[cattywampus + arylo:cattywampus + arylo + 2], 0x10)
        b2 = bibble[quean % len(bibble)]
        kerfuffle.append(b1 ^ b2)
        quean += 1
        if quean == s:
            break
    return kerfuffle
```

Figure 7: Python reimplementation of decoding function `canoodle`

Figure 8 shows Python code to decode the data.

```
# 58c7661f... text field data copied to text file
with open("F_T_Text.txt", "r") as f:
    d = f.read()

xertz = [0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE]

wabbit = canoodle(d, 0, 168667, xertz)
with open("stomp.mp3", "wb") as f:
    f.write(wabbit)
```

Figure 8: Python code to decode the embedded data

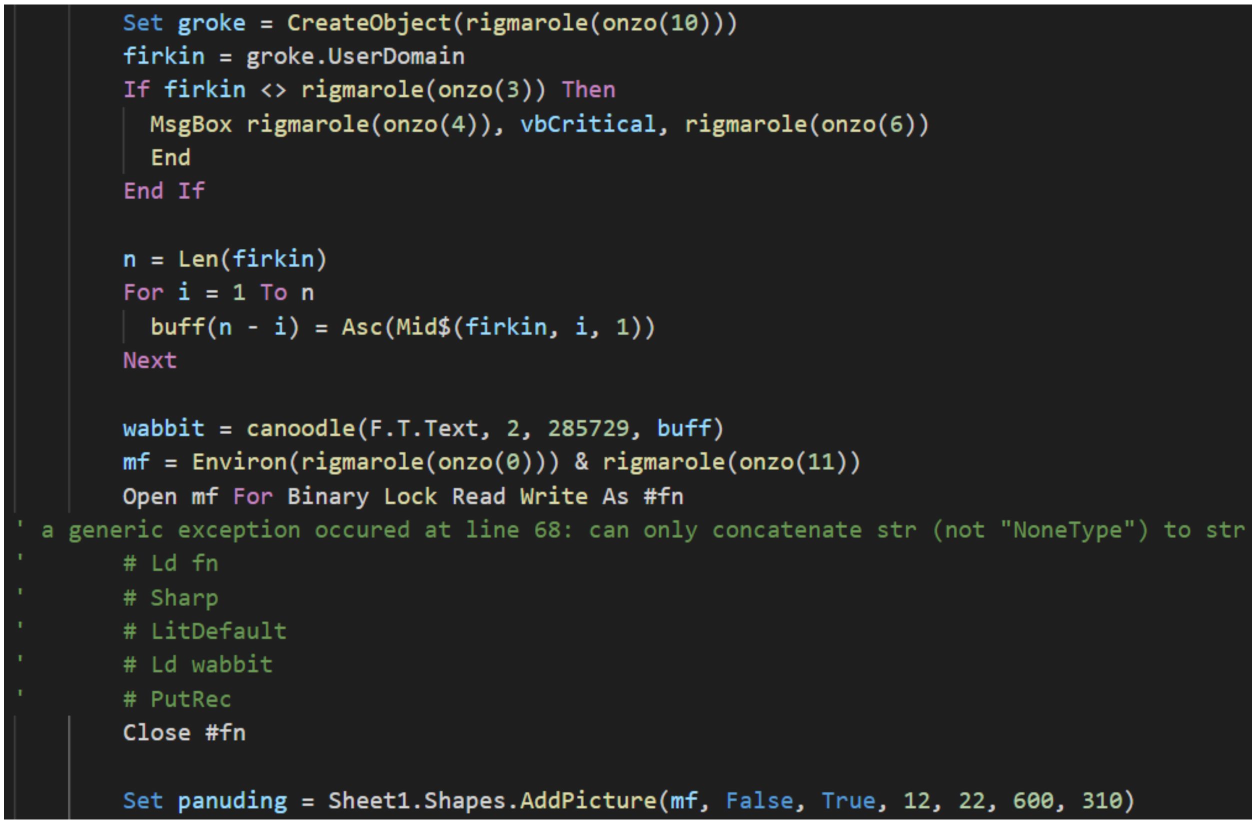
The created audio file contains… stomping sounds. Moreover, there are several hints in the file's meta data that indicate that we are on the wrong track. So, remembering the VBA stomping detection discussed earlier we now focus on the p-code.

P-CODE ANALYSIS

Reading p-code directly is not the most enjoyable task. Luckily, we can use pcode2code to obtain the decompiled VBA code with the command `pcode2code report.xls`. And although we get some errors, we can inspect the code almost entirely decompiled.

We closely compare the decompiled to the previously analyzed VBA code and notice only a few differences. Some of these differences come from an imperfect decompilation. For example, incorrectly recovered data types, e.g., `wabbit`'s type is identified as byte and not as a byte array. Moreover, `pcode2code` may add superfluous function parameters, e.g. for the `rigmarole` function.

Figure 9 shows the main functional difference in the `folderol` function between the fake VBA code and the decompiled p-code.



```
        Set groke = CreateObject(rigmarole(onzo(10)))
        firkin = groke.UserDomain
        If firkin <> rigmarole(onzo(3)) Then
          MsgBox rigmarole(onzo(4)), vbCritical, rigmarole(onzo(6))
          End
        End If

        n = Len(firkin)
        For i = 1 To n
          buff(n - i) = Asc(Mid$(firkin, i, 1))
        Next

        wabbit = canoodle(F.T.Text, 2, 285729, buff)
        mf = Environ(rigmarole(onzo(0))) & rigmarole(onzo(11))
        Open mf For Binary Lock Read Write As #fn
' a generic exception occured at line 68: can only concatenate str (not "NoneType") to str
'       # Ld fn
'       # Sharp
'       # LitDefault
'       # Ld wabbit
'       # PutRec
        Close #fn

        Set panuding = Sheet1.Shapes.AddPicture(mf, False, True, 12, 22, 600, 310)
```

Figure 9: Additional code in main function `folderol` recovered via `pcode2code`

Using the earlier decoded strings, we see that the program terminates if the computer is not connected to a domain named `FLARE-ON`. If it is, the function reverts the string `FLARE-ON` and passes it to the `canoodle` data decoding function. The decoded data is written to the file `%AppData%\Microsoft\v.png` and then added to the Excel worksheet `Sheet1` as a picture.

Challenge flag

Figure 10 shows the modified Python code to decode this data.

```
# 58c7661f... data copied to text file
with open("F_T_Text.txt", "r") as f:
    d = f.read()

buff = "FLARE-ON"[::-1]
buff = [ord(c) for c in buff]

wabbit = canoodle(d, 2, 285729, buff)
with open("v.png", "wb") as f:
    f.write(wabbit)
```

Figure 10: Python code to decode the other embedded data

The decrypted image revealing the challenge flag is shown in Figure 11.

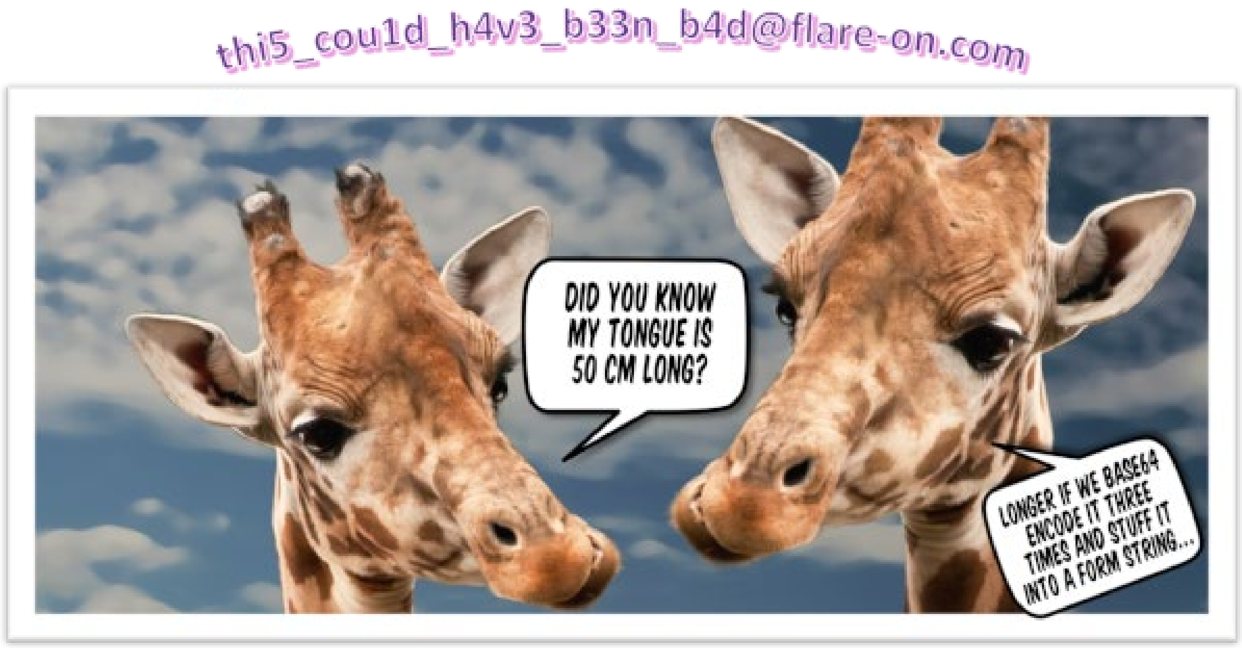

thi5_cou1d_h4v3_b33n_b4d@flare-on.com
DID YOU KNOW MY TONGUE IS 50 CM LONG?
LONGER IF WE BASE64 ENCODE IT THREE TIMES AND STUFF IT INTO A FORM STRING...

Figure 11: Decrypted image with challenge flag

The flag is: `thi5_cou1d_h4v3_b33n_b4d@flare-on.com`.

Acknowledgements

Big shoutout to Rick Cole (@a_tweeter_user) for all his support with this challenge!