Less SmartScreen More Caffeine: (Ab)Using ClickOnce for Trusted Code Execution

- Streamlined, minimal user interaction required
- Ease of rerolling execution implementations

Ultimately, we want to take a relatively common initial access technique known as ClickOnce and extend its value for the offensive use case by abusing the trust of third-party applications.

. . .

> ClickOnce deployment manifests

- *.application is the file extension for these
- References the ClickOnce application manifest to deploy
- APPREF-MS file will point to this (if used)

> ClickOnce application manifests

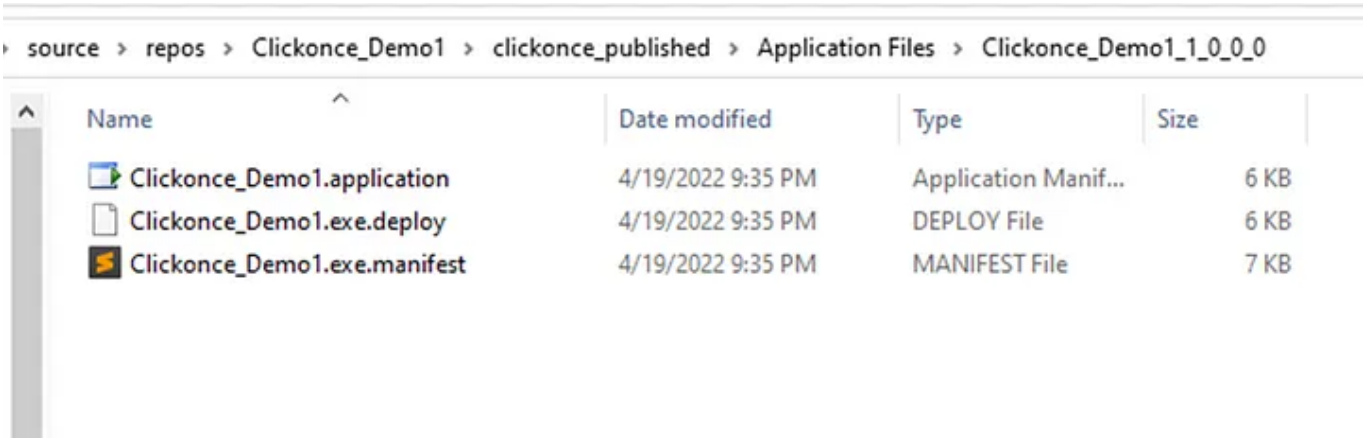
- *.exe.manifest is the file extension for these

```
// C:\Windows\Microsoft.NET\Framework64\v4.0.30319\CasPol.exe
// caspol, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a

// Entry point: Microsoft.Tools.Caspol.caspol.Main
// Timestamp: 5DDA41DE (11/24/2019 8:39:58 AM)

using System;
using System.Diagnostics;
using System.Reflection;
using System.Resources;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyVersion("4.0.0.0")]
[assembly: CompilationRelaxations(8)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: ComVisible(false)]
[assembly: CLSCompliant(true)]
[assembly: AssemblyTitle("caspol.exe")]
[assembly: AssemblyDescription("caspol.exe")]
[assembly: AssemblyDefaultAlias("caspol.exe")]
[assembly: AssemblyCompany("Microsoft Corporation")]
[assembly: AssemblyProduct("Microsoft® .NET Framework")]
[assembly: AssemblyCopyright("© Microsoft Corporation.  All rights reserved.")]
```



ClickOnce Deployment Manifest, Executable, and Application Manifest

ClickOnce applications can be deployed to a client by visiting the

will be delivered during the deployment process. The contents of the deployment will ultimately be saved to:

C:\Users\%USERNAME%\AppData\Local\Apps\2.0\<randomstring>

Once a user has accepted to run the application, the deployment manifest will look to the ClickOnce application manifest for all the files that need to be downloaded.

System.Deployment.dll Logic to Parse Manifests

Commonly, when crafting an initial access payload and using ClickOnce, you'd go through the process of writing it up in an IDE like Visual Studio and building the ClickOnce application. So what does standard ClickOnce

Current ClickOnce Weaponization Pressure Points

As seen in the first demo, we experience a few issues. For instance, Microsoft SmartScreen was triggered. This is because the assembly that ultimately executed with our arbitrary code was compiled recently and had never been seen by SmartScreen before. The reputation for Microsoft SmartScreen can be based on a number of factors such as the hash of the host assembly or the certificate used to sign the assembly.

conduct our code execution, especially during initial access attempts. An Extended Validation (EV) code signing certificate can be used to obtain immediate SmartScreen reputation, but the vetting process and price point increase the barrier to entry. When code-signing certificates are used, there are also additional attribution concerns.

Generally, a ClickOnce deployment can be tedious to make sure "all the stars align" for a successful deployment. Oftentimes people view ClickOnce as tedious to deploy successfully and having many configuration requirements. We hope the next couple sections outline the important fields within

which will be covered later). Several tools can be used throughout this process (e.g. dnSpy, reshacker, mage, sigcheck, etc).

The image below is a quick example of what sideloading an existing, signed ClickOnce deployment would look like. First, we find a ClickOnce deployment published online, download it, and verify the assembly that the deployment executes meets our needs (valid code signature, SmartScreen reputation, etc):

We observe the code within this method and verify it exists within a DLL dependency (not the host .NET assembly):

checks that occur during deployment do not fail. Here's a few tips that will hopefully speed the process up:

- **publicKeyToken** — this value is required, but can be nulled out by replacing the value with 16 zeros
- **<hash>** — this block is optional and can be removed or recalculated (EX: *openssl -dgst -binary -sha1 Program.exe.manifest |openssl enc -base64)*
- **<publisherIdentity>** — included if the manifests have been signed, but is optional and can be removed

So the question posed is this: *Do we really need a code signing certificate to effectively weaponize ClickOnce deployments?*

2. The UAC settings cannot be set to ‘requireAdministrator’ or ‘highestAvailable’

.NET assemblies that meet these prerequisites can be weaponized as backdoored ClickOnce deployments relatively easily. The *System.Deployment* DLL has code that checks the assembly identity which is found in the embedded application manifest. This check cross-references the application manifest’s identity to ensure the identity values are the same. The image below shows what the embedded assembly manifest default identity will be

ClickOnce Deployment “assemblyIdentity”

The ‘*processorArchitecture*’ value is a required value to be present for the assembly identity in the deployment manifest.

'*processorArchitecture*' value present. Therefore, this type of assembly is not possible to use as a ClickOnce application for our purposes. Modifying this value would require modifying the host assembly of our code execution, losing any benefit of a valid code-signature or reputations with Microsoft SmartScreen.

```
            <requestedExecutionLevel  level="highestAvailable" uiAccess="false" />

            Specifying requestedExecutionLevel node will disable file and registry v
            If you want to utilize File and Registry Virtualization for backward
            compatibility then delete the requestedExecutionLevel node.
        -->
        <requestedExecutionLevel level="asInvoker" uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>

  <compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
    <application>
      <!-- A list of all Windows versions that this application is designed to work
      Windows will automatically select the most compatible environment.-->
```

UAC Check in System.Deployment DLL

If UAC information exists, or it is set to ‘asInvoker’ the assembly will work as a ClickOnce deployment

and Mage. Mage is a command line tool that comes part of the Windows SDK and for the purpose of this blog will be the one we cover.

Once you have gone through the process of identifying a .NET assembly that can be wrapped up as a ClickOnce deployment, you will want to create the directory structure of the assembly, dependencies, and extra files. As previously mentioned, there are two manifests that will need to be created with Mage — the deployment manifest and the application manifest. The application manifest can be created with the following command:

will be invalidated and will have to be regenerated which can lead to unnecessary troubleshooting. As mentioned previously, these values are:

- **<publicKeyToken>**, required but can be nulled with 16 zeros
- **<hash>** block can be removed altogether and not required
- **Publisher identity** block can be removed altogether

Now that we have identified an existing signed .NET assembly that can be deployed as a ClickOnce application, we can go through the same backdoor

. . .

Identification of .NET Assemblies and ClickOnce Applications

So far, we've covered the types of applications that can be weaponized, and now we want to discover potential targets. We have released two tools that will aid in the discovery of existing ClickOnce applications and .NET assemblies that can be weaponized for ClickOnce.

While ClickonceHunter will go look through the internet for existing applications, AssemblyHunter will recursively search local file systems for assemblies that meet the criteria for a regular .NET assembly to be deployed as a ClickOnce application.

Using AssemblyHunter, we can quickly identify assemblies across a host's filesystem and look for values that will be useful to us.

would consider looking for when identifying or preventing malicious ClickOnce use is:

> Monitoring *dfsvc.exe* process activity

- Monitoring child process activity (e.g. child processes with unsigned module loads)
- Baseline required ClickOnce activity to whitelist applications with valid business use-cases

- Zones include: MyComputer, LocalIntranet, TrustedSites, Internet, UntrustedSites
- To disable installation from internet: *\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\Security\TrustManager\PromptingLevel — Internet:Disabled*

> If an Application Control solution is deployed

- Prevent unreputable DLLs from being loaded

Closing

Based on all that was covered, we see ClickOnce as one of the best opportunities for initial access. There are still plenty of areas to dig into and additional potential for offensive use-cases. A few people we want to give thanks to and who paved the way for the work done are Lee Christensen (@tifkin_), whose exploration of this technique wouldn't have been possible without him, Casey Smith (@subTee) for previous .NET research, and William Burke (@0xF4B0) for previous ClickOnce research.