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

Nick Powers · Follow
Published in Posts By SpecterOps Team Members · 15 min read · Jun 7, 2023

The contents of this blogpost was written by Nick Powers (@zyn3rgy) and Steven Flores (@0xthirteen), and is a written version of the content presented at Defcon30.

With the barrier to entry for initial access ever increasing, we spent some time digging into potentially lesser-known weaponization options for ClickOnce deployments. A few of the hurdles we'd like to overcome by implementing these weaponization options include:

- Install / execute application without administrative privileges
- Reputable, known-good file(s) used during 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 Overview and Current Weaponizations

> *"ClickOnce is a deployment technology that enables you to create self-updating Windows-based applications that can be installed and run with minimal user interaction" -MSDN*

ClickOnce is a vehicle for installing and updating .NET applications. Deployments can be published through a variety of options (e.g. network file shares, legacy media [CD-ROM], and web pages). We will be focusing on the web page method of publishing deployments. Legitimate applications exist and make use of ClickOnce deployments for installation or updating software such as Chrome (previously), Fidelity, and others.

ClickOnce deployments rely on manifests which are formatted in a very specific way. Just like .NET applications, there will be different types of manifests that need to be accounted for. There are three types of manifest to become familiar with when discussing ClickOnce deployments:

> ClickOnce deployment manifests

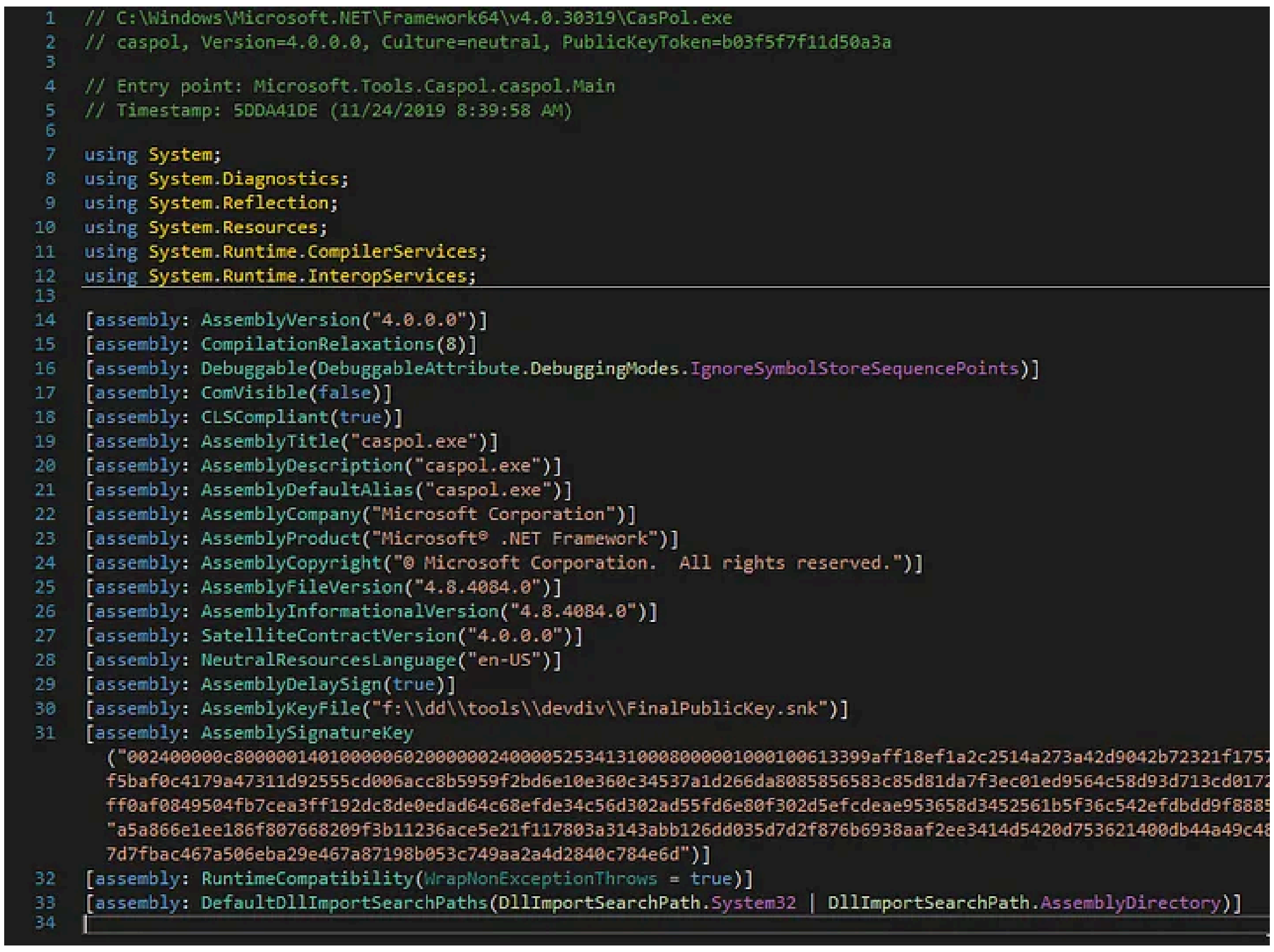
- *.application is the file extension for these
- References the ClickOnce application manifest to deploy
- APPREF-MS file will point to this (if used)

> ClickOnce application manifests

- *.exe.manifest is the file extension for these
- Specifies dependencies for the deployment (states version of .NET that will be utilized)
- Conducts integrity check of deployment manifest
- References to dependencies and other files for delivery

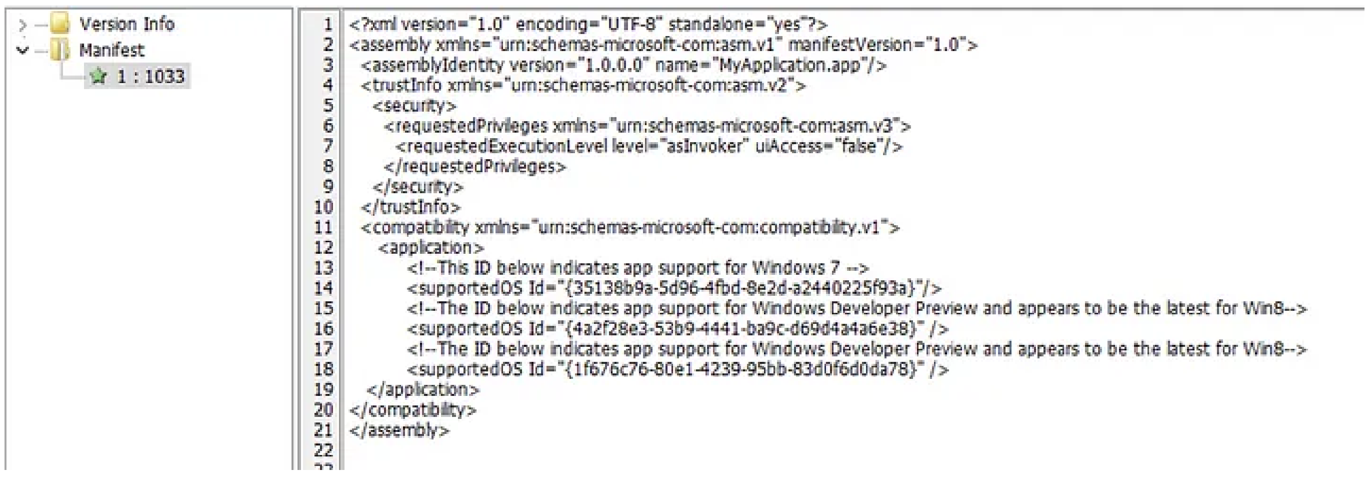
> Embedded application and assembly manifests

- Application manifest can also be called unmanaged or fusion manifest
- Assembly manifest can also be called managed manifest
- At runtime, ClickOnce makes comparisons against these



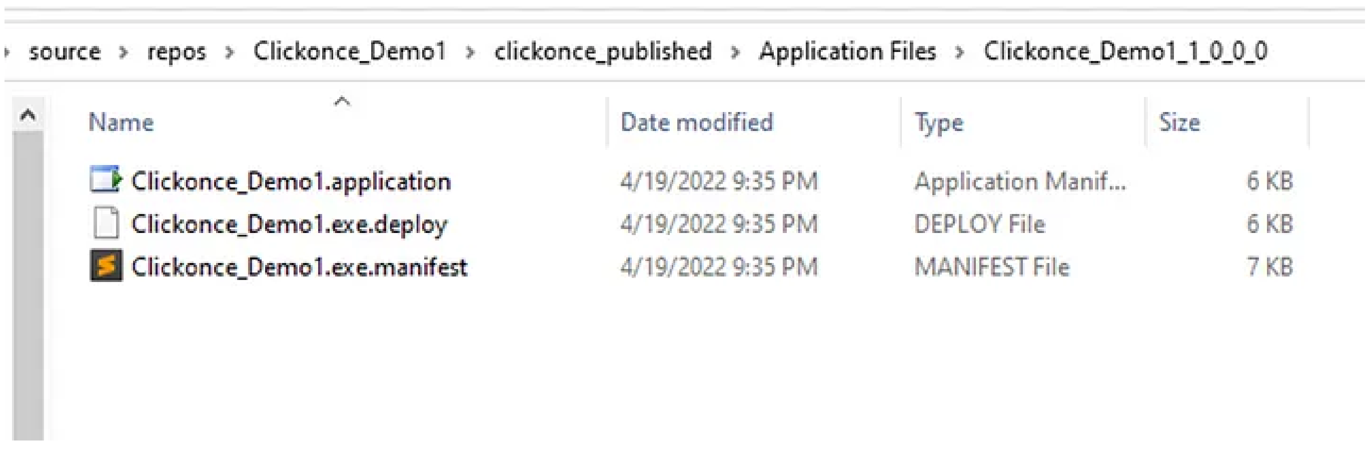
```
// C:\Windows\Microsoft.NET\Framework64\v4.0.30319\CasPol.exe
// caspol, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a

// 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.")]
[assembly: AssemblyFileVersion("4.8.4084.0")]
[assembly: AssemblyInformationalVersion("4.8.4084.0")]
[assembly: SatelliteContractVersion("4.0.0.0")]
[assembly: NeutralResourcesLanguage("en-US")]
[assembly: AssemblyDelaySign(true)]
[assembly: AssemblyKeyFile("f:\\dd\\tools\\devdiv\\FinalPublicKey.snk")]
[assembly: AssemblySignatureKey
  ("002400000c800000140100000602000000240000525341310008000001000100613399aff18ef1a2c2514a273a42d9042b72321f1757
  f5baf0c4179a47311d92555cd006acc8b5959f2bd6e10e360c34537a1d266da8085856583c85d81da7f3ec01ed9564c58d93d713cd0172
  ff0af0849504fb7cea3ff192dc8de0edad64c68efde34c56d302ad55fd6e80f302d5efcdeae953658d3452561b5f36c542efdbdd9f8885
  "a5a866e1ee186f807668209f3b11236ace5e21f117803a3143abb126dd035d7d2f876b6938aaf2ee3414d5420d753621400db44a49c48
  7d7fbac467a506eba29e467a87198b053c749aa2a4d2840c784e6d")]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: DefaultDllImportSearchPaths(DllImportSearchPath.System32 | DllImportSearchPath.AssemblyDirectory)]
```

Embedded Assembly Manifest



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker" uiAccess="false"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
  <compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
     <application>
          <!--This ID below indicates app support for Windows 7 -->
          <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>
          <!--The ID below indicates app support for Windows Developer Preview and appears to be the latest for Win8-->
          <supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}" />
          <!--The ID below indicates app support for Windows Developer Preview and appears to be the latest for Win8-->
          <supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}" />
    </application>
</compatibility>
</assembly>
```

Embedded Application Manifest



source › repos › Clickonce_Demo1 › clickonce_published › Application Files › Clickonce_Demo1_1_0_0_0

Name	Date modified	Type	Size
Clickonce_Demo1.application	4/19/2022 9:35 PM	Application Manif...	6 KB
Clickonce_Demo1.exe.deploy	4/19/2022 9:35 PM	DEPLOY File	6 KB
Clickonce_Demo1.exe.manifest	4/19/2022 9:35 PM	MANIFEST File	7 KB

ClickOnce Deployment Manifest, Executable, and Application Manifest

ClickOnce applications can be deployed to a client by visiting the **.application* deployment manifest using a browser. The download and execution process of a standard ClickOnce deployment requires the end user to be using a Microsoft Edge or Internet Explorer browser. An alternative to requiring the target user to access your deployment manifest from either Edge or Internet Explorer would be to leverage APPREF-MS. When creating an **.appref-ms* file, UTF-16 LE encoding is required. An *appref-ms* file would be used if the end user is using something like Chrome or Firefox to access the application.

> ***NOTE:*** *A simple user-agent check can be done prior to reaching the landing page to determine whether the incoming request should be pointed towards the standard deployment manifest or an appref-ms file.*

The host assembly, specified in the manifest, will spawn as a child process of '*dfsvc.exe*', which handles the ClickOnce deployment functionality that is imported from '*System.Deployment.dll*'. The contents of the ClickOnce application manifest will specify what dependencies and other resources 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.

ClickOnce Deployment Manifest Example

There will be different pieces of information located throughout the manifest that look very similar to .NET application manifests. This will contain all of the files and dependencies the deployment will need to execute properly.

ClickOnce Application Manifest Example

The *dfsvc* process will make a series of HTTP requests for the downloads and save in the *%LocalAppData%* location mentioned above.

HTTP Requests While Downloading a ClickOnce Application

In some scenarios users may want to interact with ClickOnce applications but are not using Microsoft Edge or Internet Explorer. An appref-ms file can be created that will act similar to an LNK or shortcut file that will contain the URL of the deployment manifest and some other pieces of information.

APPREF-MS Example

During the process before all the downloading happens, the System.Deployment DLL will be utilized to run through various checks and ensure the ClickOnce application can properly execute. One important check to note is the .NET application and ClickOnce deployment identities that are configured in the assembly and the deployment manifests.

System.Deployment.dll Logic to Parse Manifests

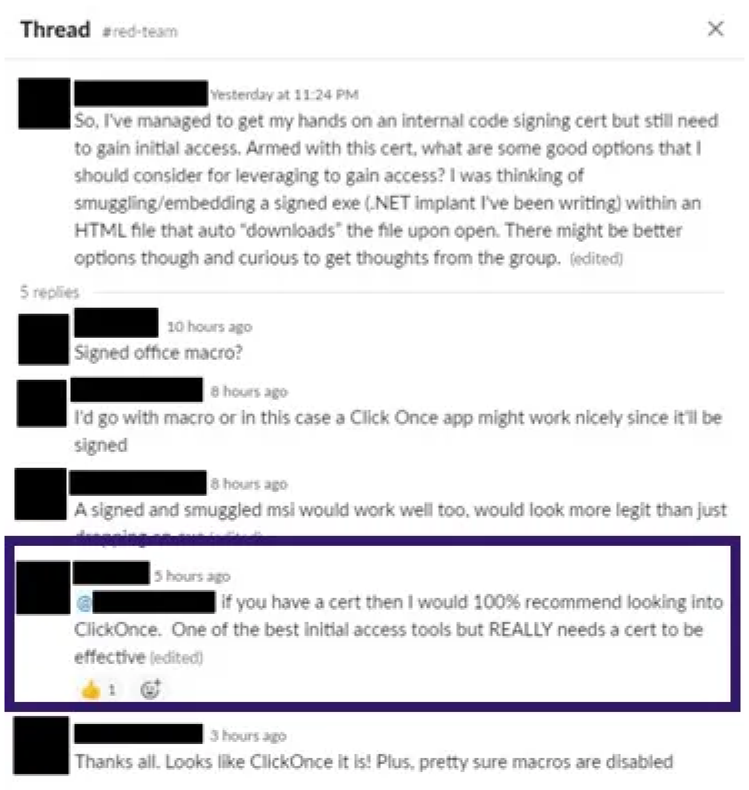
Commonly, when crafting an initial access payload and using ClickOnce, you'd go through the process of writing it up in an IDE like Visual Studio and building the ClickOnce application. So what does standard ClickOnce deployment execution look like, using a newly created .NET application?

DEMO1 — Current ClickOnce Weaponization Example

. . .

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.



Thread #red-team

[redacted] Yesterday at 11:24 PM
So, I've managed to get my hands on an internal code signing cert but still need to gain initial access. Armed with this cert, what are some good options that I should consider for leveraging to gain access? I was thinking of smuggling/embedding a signed exe (.NET implant I've been writing) within an HTML file that auto "downloads" the file upon open. There might be better options though and curious to get thoughts from the group. (edited)

5 replies

[redacted] 10 hours ago
Signed office macro?

[redacted] 8 hours ago
I'd go with macro or in this case a Click Once app might work nicely since it'll be signed

[redacted] 8 hours ago
A signed and smuggled msi would work well too, would look more legit than just dropping an exe (edited)

[redacted] 5 hours ago
@[redacted] if you have a cert then I would 100% recommend looking into ClickOnce. One of the best initial access tools but REALLY needs a cert to be effective (edited)
👍 1

[redacted] 3 hours ago
Thanks all. Looks like ClickOnce it is! Plus, pretty sure macros are disabled

BloodHound Slack Discussion on ClickOnce

Achieving arbitrary code execution within the context of an application seen and otherwise trusted by Microsoft SmartScreen or EDR products can decrease likelihood of prevention. Solutions such as application control or whitelisting prevention can be noteworthy when considering how we want to 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 ClickOnce manifests to focus on, to assist in reproducing these techniques.

. . .

Alleviation of ClickOnce Weaponization Pressure Points

If legitimate .NET applications make use of ClickOnce, and we can reliably sideload or hijack those .NET applications, why not just backdoor an existing deployment? An existing deployment that already has, lets say, a valid EV code signature and SmartScreen reputation?

> ***NOTE:*** *We will backdoor a dependency of a ClickOnce deployment, not the host assembly itself. We* maintain the valid signature *associated with that host assembly.*

Identifying existing ClickOnce deployments can be as easy as leveraging a couple search engine dorking techniques (or using the ClickonceHunter tool which will be covered later). Several tools can be used throughout this process (e.g. dnSpy, reshacker, mage, sigcheck, etc).

Download of ClickOnce Application Discovered from Google

We will want to identify sideloading opportunities for dependencies of the target .NET assembly that the ClickOnce deployment will execute. Any way for us to hijack the flow of execution shortly after the application entry point is what we are looking for. Oftentimes this process consists of using dnSpy to decompile the target .NET assembly to understand the execution flow enough to identify a sideload opportunity.

Once an ideal DLL to backdoor has been identified, use dnSpy to add your code to the target DLL. At this point, the ClickOnce manifests of the deployment you've chosen will need to be tweaked to pass their integrity checks. If you cannot identify an ideal sideload opportunity in the existing codepath(s), techniques such as AppDomainManager injection or .NET deserialization abuse can be helpful here.

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):

Example of Signed ClickOnce Application

Next we take a look at the references within the signed .NET assembly we want to sideload. A few of the DLLs the assembly uses are not strongly named, which can be helpful in certain situations:

Dependencies of Target Assembly

Using DnSpy to begin at the target .NET assembly's entry point, we follow the code to the first method call of "SetDpiAwareness()". This function exists within on the the Dlls previously identified:

Following the Code Path

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

Identified Location to Place Additional Code

The additional code we'd like to sideload the ClickOnce deployment with can be added here. For proof-of-concept sake, we will just spawn notepad and prompt with a MessageBox:

Adding Code to Target Dependency

At this point, dnSpy can add your changes to intermediary language (IL) and you will have your backdoored dependency of the target ClickOnce application. Now is a good time to test and make sure the additional code you've added properly executes when running the host .NET assembly.

> ***NOTE:*** *We previously thought a prerequisite was targeting only non-strongly named DLLs. We discovered this is not a requirement because modifying a DLL does not modify the PublickeyToken value of the DLL. The PublickeyToken value is in reference to the hash of the assembly's embedded manifest rather than the code itself. If the manifest is not modified, the PublickeyToken value remains the same and will still be loaded successfully.*

Now that we have our sideload of a signed .NET application that is part of an existing ClickOnce deployment, our last step to have a functioning deployment is to tweak the two ClickOnce manifests such that the integrity 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

A noteworthy mention regarding modification of an existing ClickOnce manifest is that if the manifest was signed with a valid certificate, then making these modifications will break that integrity. The difference to the end user is minimal and many legitimate ClickOnce deployments do not sign their manifests at all. We still have control of the prompt observed by the target user, such as "Name" and "From". A signed vs unsigned look similar to the image below:

Difference Between Signed and Unsigned Deployment Manifest

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

DEMO2 — Backdooring Third-Party ClickOnce Deployments

. . .

Extending Past Existing ClickOnce Deployments

The number of published ClickOnce deployments easily identifiable through dorking is finite. It's not a new technology, nor is it the most popular to deploy and update .NET applications. We foresaw this as a potential issue and sought to find a way to take *any* ideal .NET assembly, identify a sideload opportunity, and wrap it up as a new ClickOnce deployment. As it turns out, this was possible, with a few prerequisites.

1. The <assemblyIdentity> field within the embedded application manifest must *not* exist, or the entire embedded application manifest must not exist (more on this in a bit)
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 if it is present.

Example of Default "assemblyIdentity" Field

As you can see the identity contains two pieces of information, the version and the name. The figure below shows what is in the deployment manifest for the identity value. If you look at the difference there is a value of processorArchitecture.

ClickOnce Deployment "assemblyIdentity"

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

MSDN Documentation Showing Required Values

These two values are checked against each other to validate assembly identities are the same. If the default identity value is present in the assembly's embedded manifest it will fail because there is no '*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.

System.Deployment dll Checking Identities

Fortunately there are a lot of assemblies that exist which do not have an identity in the application's manifest. In the next section we will show how to identify these assemblies, but in the meantime, the figure below shows what an embedded manifest looks like when the default identity is not set, and a non-default manifest was used during the build process instead.

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1" xmlns
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <!-- UAC Manifest Options
            If you want to change the Windows User Account Control level replace the
            requestedExecutionLevel node with one of the following.

            <requestedExecutionLevel  level="asInvoker" uiAccess="false" />
            <requestedExecutionLevel  level="requireAdministrator" uiAccess="false"
            <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.-->

      <!-- If your application is designed to work with Windows 7, uncomment the fol
      <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>

      <!-- If your application is designed to work with Windows 8, uncomment the fol
      <supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/>

      <!-- If your application is designed to work with Windows 8.1, uncomment the f
      <supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>

      <!-- This Id value indicates the application supports Windows Threshold functi
      <supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/>
    </application>
  </compatibility>
</asmv1:assembly>
```

The second condition that is required is for the UAC settings to not be requiredAdministrator or highestAvailable. Another check by the System.Deployment DLL is it will look for the UAC settings and return errors if the disallowed values are set.

UAC Check in System.Deployment DLL

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

Assembly with No UAC Information

Since we are creating ClickOnce applications from scratch, we will have to create new manifests as opposed to our previous weaponization of modifying existing ClickOnce manifests. Microsoft has a utility that is used for this specific task which is called the Manifest Generation and Editing Tool (Mage). Microsoft makes two different tools that can be used, MageUI 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:

```
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\mage.exe"
```

Next you will need to create the deployment manifest. This can be done with the following command:

```
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\mage.exe"
```

The *ProviderUrl* argument is the location where the deployment manifest will be hosted since the primary method we're covering is web-based applications. Once the manifests are created, you will see there are some signature values that are created with *Mage*. Just like what was covered when editing existing manifests, these values are not always required and can be removed. If any changes are made to the overall deployment, the signatures 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 steps as the other technique. We will follow the code paths, identify called

DLLs, and place our code inside of those DLLs. Finally, we can create the manifests with Mage and are ready for deployment.

DEMO3 — Backdooring Arbitrary .NET Assembly and Deploying as ClickOnce

. . .

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.

> ClickonceHunter

- searches online for existing ClickOnce published code
- Google dorks, Github and others

> AssemblyHunter

- Searches file paths or files and looks for given criteria (signed, identity info, arch, UAC, etc.)
- Helps identify target applications to weaponize

ClickonceHunter will automate what can be done manually with Google or other related searches.

Dorking to Discover Third-Party ClickOnce Deployments

Github Searches for ClickOnce Projects (maybe with signed releases?)

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.

Usage for AssemblyHunter

AssemblyHunter Discovering .NET Assembly

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

AssemblyHunter Showing Assemblies that can be Weaponized

AssemblyHunter will also show us assemblies that are not useful to us if we would like to see them.

Example of anAssembly that Cannot be Weaponized

Detection and Prevention Opportunities

A major benefit to defenders who want to identify malicious ClickOnce deployments is that ClickOnce is not commonly used in many corporate environments. Defenders can baseline their environments to look for how prevalent they are and make detection or prevention decisions. Things we 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
- Monitoring activity associated with *dfshim.dll* (can also be used for launching ClickOnce deployments)

> Evaluate ETW telemetry associated with ClickOnce deployment execution

- Keep in mind ETW bypasses or *<etwEnable>* .NET config value

> Baseline the default ClickOnce installation directories

- *%LOCALAPPDATA%\Apps\2.0\<string>*

> Baseline of application that have never been seen making connections to the internet

> Disable all ClickOnce installations from the internet, while still allowing from other trust zones

- Options include: Enabled, AuthenticodeRequired, and Disabled
- 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

If ClickOnce application execution from the internet is disabled using the registry key(s) mentioned above, a user will recieve a prompt that does not give them the option to run the application.

Prevention of ClickOnce Installation from Internet Trust Zone

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.

Research Clickonce Dot Net Framework Phishing

--

Written by Nick Powers

88 Followers · Writer for Posts By SpecterOps Team Members

Follow