Chapter 2 - The Windows Application Programmable Interface

WinAPI

The Windows API is a set of programming interfaces available in the Windows OS. They are readily accessible in C and C++ languages as they expose all the necessary data structures, but can be used from a variety of languages if those data structures and calling conventions are defined by the programmer (we'll see this when we look at P/Invoke). You'll often see "Windows API" referred to as WinAPIs or Win32 APIs.

The use of WinAPIs are practically essential for offensive operations on Windows. Actions such as host enumeration, starting processes, process injection, token manipulation and more, are underpinned by these APIs. The most commonly used set of WinAPIs are the base services (**kernel32.dll**) and advanced services (**advapi32.dll**).

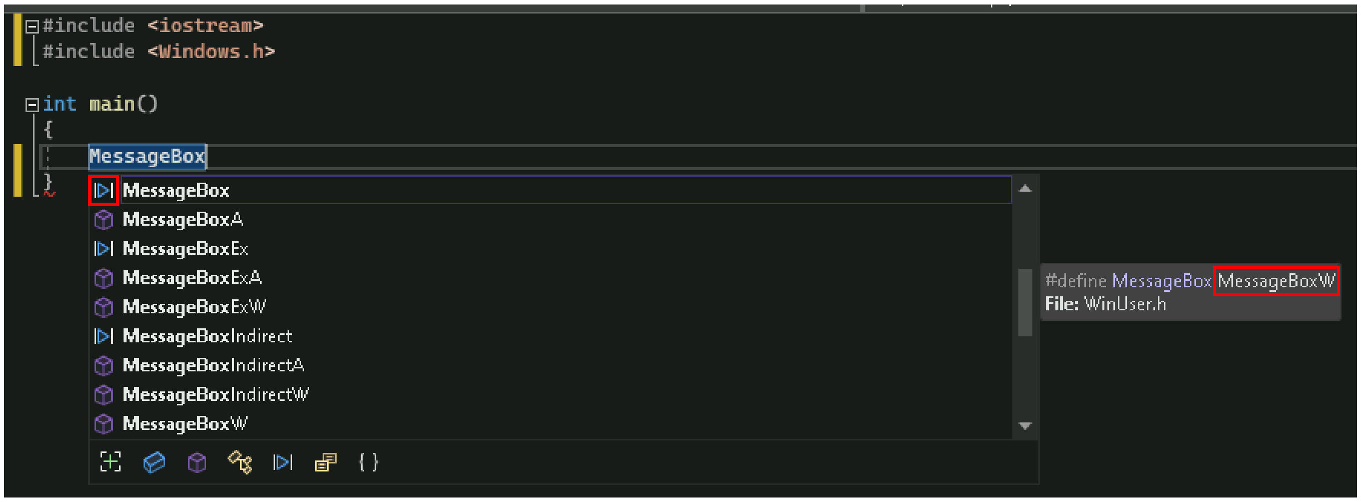
In addition to the WinAPIs, there are the Native APIs. Most of the Native API calls are implemented in **ntoskrnl.exe** (the Windows kernel image) and exposed to user mode via **ntdll.dll**. These APIs are not strictly designed to be called directly from user applications, and as such are not as accessible as the WinAPIs. The higher-level WinAPIs actually call these Native APIs in the background. For example, OpenProcess in **kernel32.dll** calls NtOpenProcess in **ntdll.dll**.

However, there are OPSEC reasons as to why we'd want to call NtOpenProcess directly instead of OpenProcess - which we'll uncover when we look at userland API hooking. For now, we're going to learn how to call basic Win32 APIs

MessageBox in C++

Open Visual Studio and create a new C++ Console App. The default project template outputs "Hello World" to the console. We can just delete this line.

Add another "include" statement at the top for `<Windows.h>`. This is a header file which contains declarations for all of the functions in the Windows API, all the common macros used by Windows programmers, and all the data types used by the various functions. After adding it, you can start typing "MessageBox" and Visual Studio's Intellisense will present some options for you.



```
#include <iostream>
#include <Windows.h>

int main()
{
    MessageBox
}
```

```
#define MessageBox MessageBoxW
File: WinUser.h
```

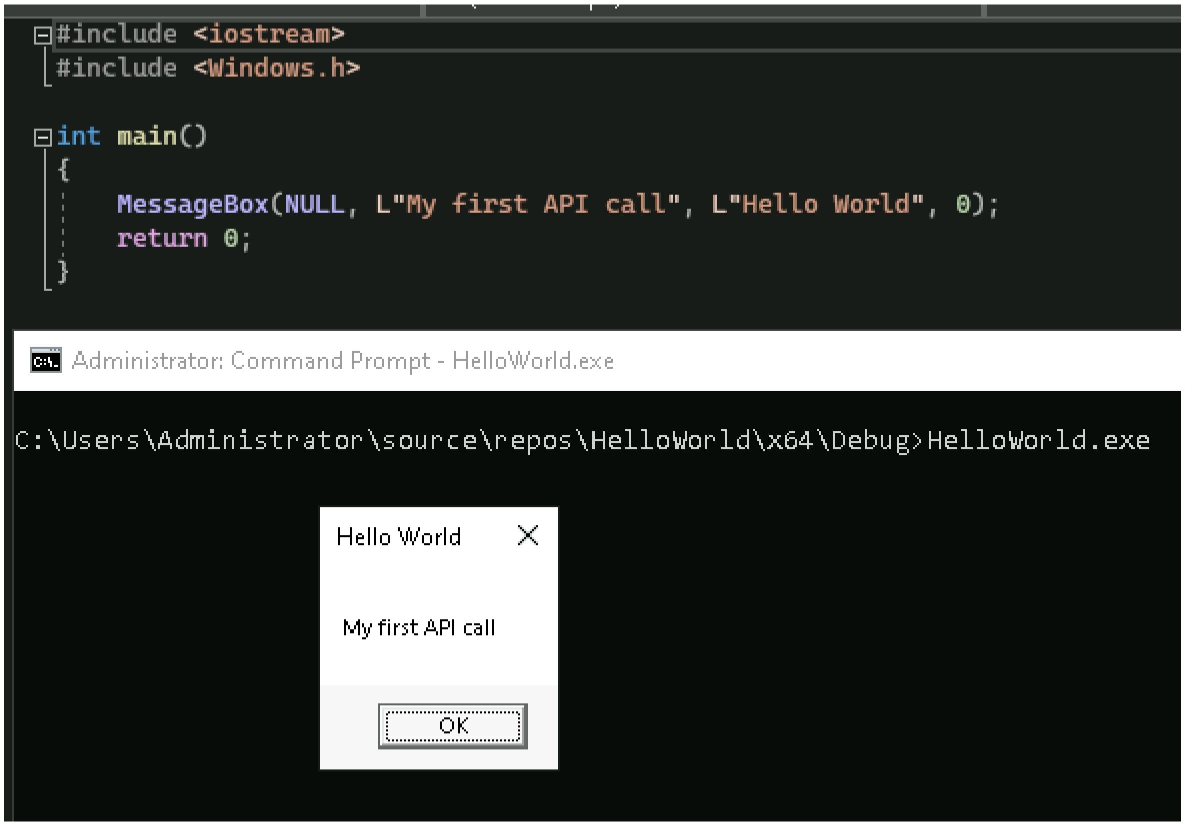
There are so many options that it can be confusing - who knew there were so many ways to pop a message box... The suggestions with the purple cube icons are **Functions**, and the ones with the blue/white play-button style icons are **Macros**. These macros are like shortcuts to existing functions - in this case, the MessageBox macro points to MessageBoxW. You can also see that MessageBoxA exists, so what's the difference?

The "A" functions use ANSI strings and "W" functions use Unicode. Unicode is the preferred character encoding on Windows, which is why the MessageBox macro points to MessageBoxW by default. If you look at the function definitions for MessageBoxA and MessageBoxW, you'll see that MessageBoxA takes in LPCSTR and MessageBoxW takes LPCWSTR. If the API also returns a string (MessageBox returns an int), then the return type would also be ANSI or Unicode depending on which version of the API is called.

So we're perfectly happy to use the default macro, we can just call it like so:

```
#include <iostream>
#include <Windows.h>

int main()
{
    MessageBox(NULL, L"My first API call", L"Hello World", 0);
    return 0;
}
```



```
#include <iostream>
#include <Windows.h>

int main()
{
    MessageBox(NULL, L"My first API call", L"Hello World", 0);
    return 0;
}
```

```
Administrator: Command Prompt - HelloWorld.exe

C:\Users\Administrator\source\repos\HelloWorld\x64\Debug>HelloWorld.exe
```

CreateProcess in C++

The CreateProcess API requires us to utilise some additional data structures, namely `STARTUPINFO` and `PROCESS_INFORMATION`.

```
#include <iostream>
#include <Windows.h>

int main()
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
}
```

Placing the cursor on these variable types (e.g. STARTUPINFO) and pressing **F12** will take you to the type definition, so that you can see what properties it has. The unicode STARTUPINFOW looks like this:

```
typedef struct _STARTUPINFOW {
    DWORD   cb;
    LPWSTR  lpReserved;
    LPWSTR  lpDesktop;
    LPWSTR  lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    LPBYTE  lpReserved2;
    HANDLE  hStdInput;
    HANDLE  hStdOutput;
    HANDLE  hStdError;
} STARTUPINFOW, *LPSTARTUPINFOW;
```

And PROCESS_INFORMATION like this:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

We should always read the documentation for the APIs to get information on how they should be used. For instance, the `cb` member on STARTUPINFO should contain the size of the structure (taken from the CreateProcessW doc).

```
int main()
{
    STARTUPINFO si;
    si.cb = sizeof(si);

    PROCESS_INFORMATION pi;
}
```

We should also zero out the memory regions for these variables to ensure there is no data in them prior to their use.

```
int main()
{
    STARTUPINFO si;
    si.cb = sizeof(si);
    ZeroMemory(&si, sizeof(si));

    PROCESS_INFORMATION pi;
    ZeroMemory(&pi, sizeof(pi));
}
```

Now we're ready to call CreateProcess. If successful, the `pi` structure will be populated with the information from the new process.

```
#include <iostream>
#include <Windows.h>

int main()
{
    STARTUPINFO si;
    si.cb = sizeof(si);
    ZeroMemory(&si, sizeof(si));

    PROCESS_INFORMATION pi;
    ZeroMemory(&pi, sizeof(pi));

    BOOL success = CreateProcess(
        L"C:\\Windows\\System32\\notepad.exe",
        NULL,
        0,
        0,
        FALSE,
        0,
        NULL,
        L"C:\\Windows\\System32",
        &si,
        &pi);

    if (success)
    {
        printf("Process created with PID: %d.\n", pi.dwProcessId);
        return 0;
    }
    else
    {
        printf("Failed to create process. Error code: %d.\n", GetLastError());
        return 1;
    }
}
```

```
C:\Users\Administrator\source\repos\HelloWorld\x64\Debug>HelloWorld.exe
Process created with PID: 6820.
```

P/Invoke

Platform Invoke (P/Invoke) allows us to access structs and functions present in unmanaged libraries from our managed code.

Applications and libraries written in C/C++ compile to machine code, and are examples of unmanaged code. Programmers must manage aspects like memory management manually - e.g. whenever they allocate memory, they must remember to free it. In contrast, managed code runs on a CLR, or Common Language Runtime. Languages such as C# compile to an Intermediate Language (IL) which the CLR later converts to machine code during runtime. The CLR also handles aspects like garbage collection and various runtime checking, hence the name managed code.

Why do we even need P/Invoke? Let's use .NET as an example.

The .NET runtime already utilises P/Invoke under the hood, and provides us with abstractions that run on top. For instance, to start a process in .NET we can use the `Start` method in the `System.Diagnostics.Process` class. If we trace this method in the runtime, we'll see that it uses P/Invoke to call the CreateProcess API. However, it doesn't provide a mean that allows us to customise the data being passed into the STARTUPINFO struct; and this prevents us from being able to do things like start the process in a suspended state.

There are other WinAPIs that are useful for us (such as VirtualAllocEx, WriteProcessMemory, CreateRemoteThread, etc) that are not exposed at all in .NET; and the only way we can access them is to P/Invoke manually within our code.

Other languages that support P/Invoke also open a wealth of opportunity for attackers. For instance, we can use P/Invoke in VBA which lends a certain potency to malicious Office documents.

MessageBox in CSharp

Create a new C# Console App (.NET Framework) project.

The first step is to use the `DllImport` attribute to tell the runtime it needs to load the specified unmanaged DLL.

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

namespace PInvoke
{
    internal class Program
    {
        [DllImport("user32.dll", CharSet = CharSet.Unicode)]
        static extern int MessageBoxW(IntPtr hWnd, string lpText, string lpCaption, uint
uType);

        static void Main(string[] args)
        {
        }
    }
}
```

Within this attribute, we also define the character set so that the .NET runtime can correctly marshal managed strings to the correct unmanaged types. The name of the method should match the unmanaged API that we wish to call, and the return type and input parameters are defined as well. We can get the function signature directly from the MessageBoxW documentation, or from a resource such as pinvoke.net.

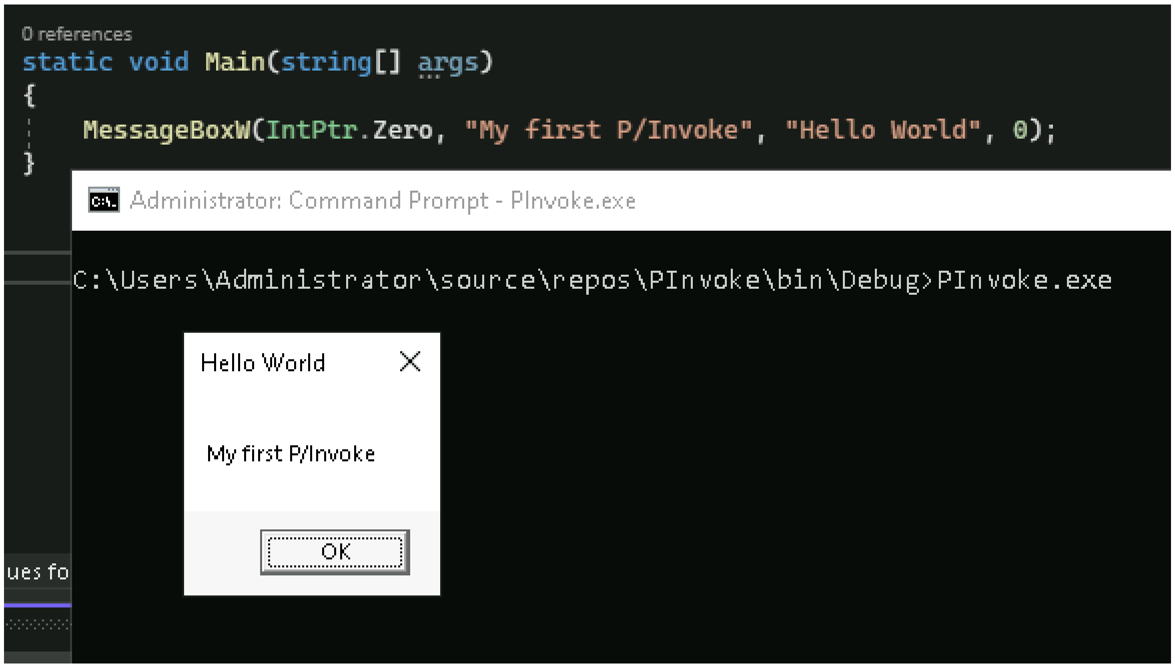
We do have to translate unmanaged types to their managed type counterparts. For instance, anything that's a HANDLE in C++ can be an IntPtr in C#.

Now this method can be called from Main.

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

namespace PInvoke
{
    internal class Program
    {
        [DllImport("user32.dll", CharSet = CharSet.Unicode)]
        static extern int MessageBoxW(IntPtr hWnd, string lpText, string lpCaption, uint
uType);

        static void Main(string[] args)
        {
            MessageBoxW(IntPtr.Zero, "My first P/Invoke", "Hello World", 0);
        }
    }
}
```



```
0 references
static void Main(string[] args)
{
    MessageBoxW(IntPtr.Zero, "My first P/Invoke", "Hello World", 0);
}
```

Administrator: Command Prompt - PInvoke.exe

```
C:\Users\Administrator\source\repos\PInvoke\bin\Debug>PInvoke.exe
```

Hello World

My first P/Invoke

Type Marshalling

"Marshalling" is the process of transforming a data type when it needs to cross between managed and unmanaged code. By default, the P/Invoke subsystem tries to automatically marshal data for you, but there may be situations where you need to marshal it manually.

In the previous lesson, we called MessageBoxW which took two `string` parameters, and we know these needed to be Unicode (LPCWSTR). We didn't have to do anything magical, because P/Invoke did it for us. If we needed to marshal them manually, we would add the `MarshalAs` attribute to the parameters. That would look something like this:

```
[DllImport("user32.dll")]
static extern int MessageBoxW(
    IntPtr hWnd,
    [MarshalAs(UnmanagedType.LPWStr)] string lpText,
    [MarshalAs(UnmanagedType.LPWStr)] string lpCaption,
    uint uType);
```

This page contains a useful table of managed and unmanaged data type mappings. Note that P/Invoke handles 99% of cases without issue - this information is more relevant when accessing native APIs without P/Invoke (e.g. D/Invoke).

CreateProcess in CSharp

Before we can call the CreateProcess API, we have to define the `STARTUPINFO` and `PROCESS_INFORMATION` data structures. We do this with the `struct` type in C#.

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

namespace PInvoke
{
    internal class Program
    {
        [StructLayout(LayoutKind.Sequential)]
        public struct STARTUPINFO
        {
            public int cb;
            public IntPtr lpReserved;
            public IntPtr lpDesktop;
            public IntPtr lpTitle;
            public int dwX;
            public int dwY;
            public int dwXSize;
            public int dwYSize;
            public int dwXCountChars;
            public int dwYCountChars;
            public int dwFillAttribute;
            public int dwFlags;
            public short wShowWindow;
            public short cbReserved2;
            public IntPtr lpReserved2;
            public IntPtr hStdInput;
            public IntPtr hStdOutput;
            public IntPtr hStdError;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct PROCESS_INFORMATION
        {
            public IntPtr hProcess;
            public IntPtr hThread;
            public int dwProcessId;
            public int dwThreadId;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct SECURITY_ATTRIBUTES
        {
            public int nLength;
            public IntPtr lpSecurityDescriptor;
            public bool bInheritHandle;
        }

        [DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
        static extern bool CreateProcessW(string lpApplicationName, string lpCommandLine,
ref SECURITY_ATTRIBUTES lpProcessAttributes,
            ref SECURITY_ATTRIBUTES lpThreadAttributes, bool bInheritHandles, uint
dwCreationFlags, IntPtr lpEnvironment,
            string lpCurrentDirectory, ref STARTUPINFO lpStartupInfo, out
PROCESS_INFORMATION lpProcessInformation);

        static void Main(string[] args)
        {
        }
    }
}
```

You can see this is a lot of extra work when compared to C++. Some aspects to note about this DllImport attribute:

- Setting `SetLastError` to true tells the runtime to capture the error code. This allows the user to retrieve it using `Marshal.GetLastWin32Error()`.
- Reading the API documentation, you will see that some parameters are passed in as pointers (e.g. `&si` in C++). In these cases, we use the `ref` keyword. For data marshalled out of the API call (e.g. for PROCESS_INFORMATION), we use the `out` keyword.

We can now call the API:

```
static void Main(string[] args)
{
    var si = new STARTUPINFO();
    si.cb = Marshal.SizeOf(si);

    var pa = new SECURITY_ATTRIBUTES();
    pa.nLength = Marshal.SizeOf(pa);

    var ta = new SECURITY_ATTRIBUTES();
    ta.nLength = Marshal.SizeOf(ta);

    var pi = new PROCESS_INFORMATION();

    var success = CreateProcessW(
        "C:\\Windows\\System32\\notepad.exe",
        null,
        ref pa,
        ref ta,
        false,
        0,
        IntPtr.Zero,
        "C:\\Windows\\System32",
        ref si,
        out pi);

    if (success)
        Console.WriteLine("Process created with PID: {0}.", pi.dwProcessId);
    else
        Console.WriteLine("Failed to create process. Error code: {0}.",
Marshal.GetLastWin32Error());
}
```

```
C:\Users\Administrator\source\repos\PInvoke\bin\Debug>PInvoke.exe
Process created with PID: 8412.
```

Ordinals

There are multiple ways to define this DllImport attribute which can be useful for circumventing some AV signatures. This is the way we're currently using them:

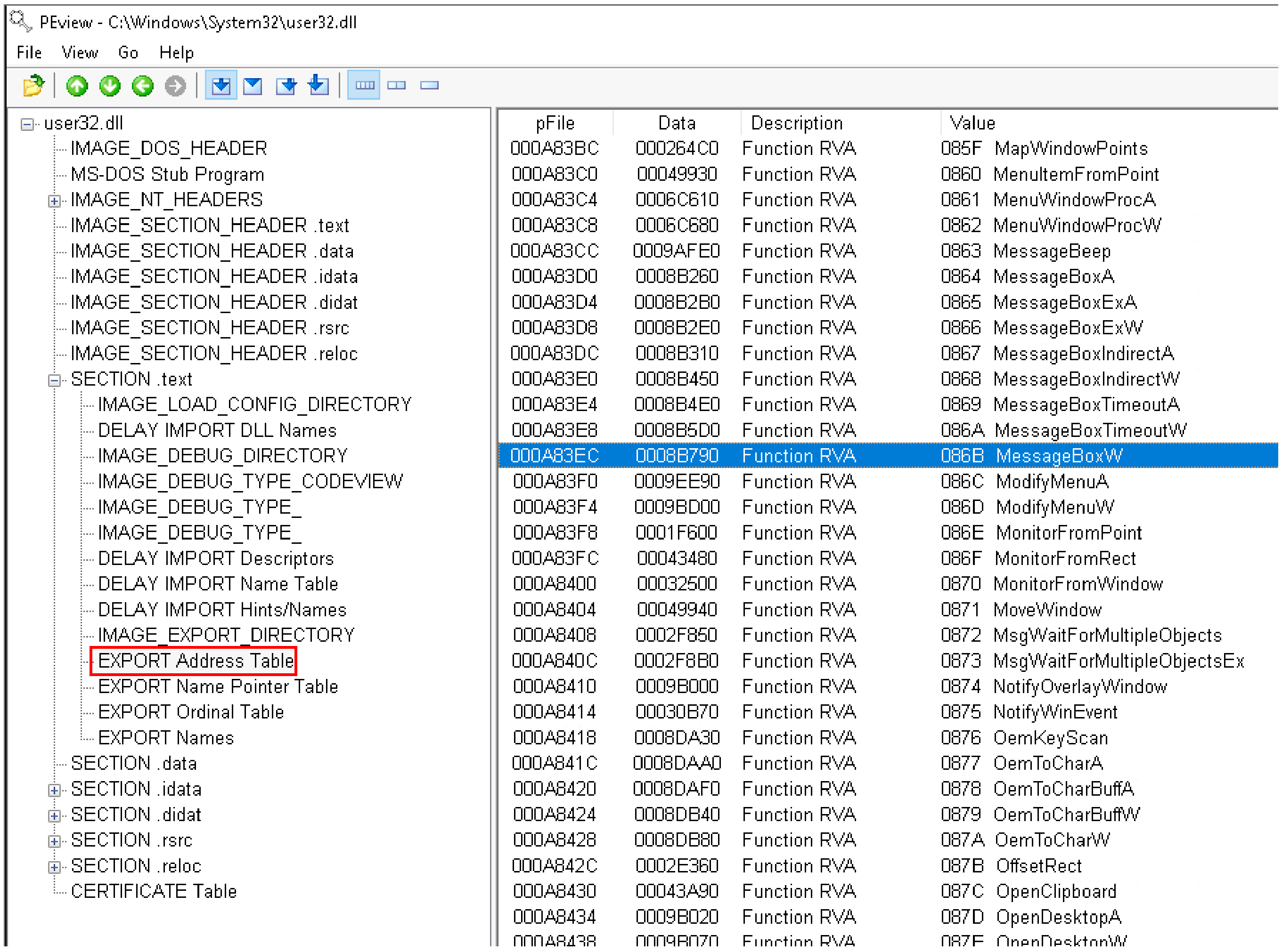
```
[DllImport("user32.dll", CharSet = CharSet.Unicode)]
static extern int MessageBoxW(IntPtr hWnd, string lpText, string lpCaption, uint uType);
```

As previously stated, the name of the method must match the name of the API. In this case, we have to use MessageBoxW, because that's the API we want to call. If an AV engine was specifically looking at these, then this would be detected. So can we call an API without using its name?

Yes - enter ordinals.

An ordinal is a number that identifies an exported function in a DLL - think of them as the Primary Key in a database table. Each exported function has an associated ordinal which is unique in that DLL, and we can use these ordinals with DllImport.s

To find the ordinal of an exported function, open the DLL with **PEview.exe,** find the **EXPORT Address Table** and scroll to the exported function that you want to call.



PEview - C:\Windows\System32\user32.dll

File View Go Help

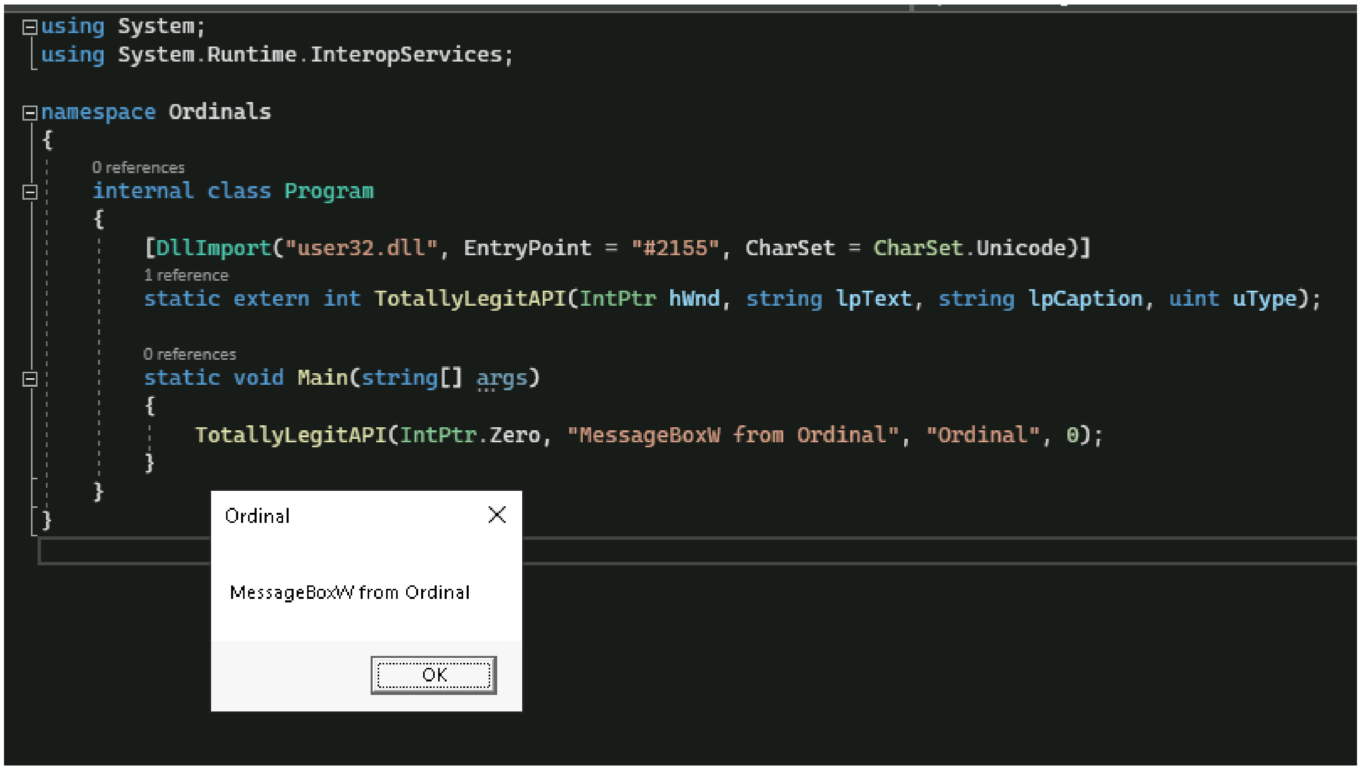
```
user32.dll
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .idata
  IMAGE_SECTION_HEADER .didat
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .reloc
  SECTION .text
    IMAGE_LOAD_CONFIG_DIRECTORY
    DELAY IMPORT DLL Names
    IMAGE_DEBUG_DIRECTORY
    IMAGE_DEBUG_TYPE_CODEVIEW
    IMAGE_DEBUG_TYPE_
    IMAGE_DEBUG_TYPE_
    DELAY IMPORT Descriptors
    DELAY IMPORT Name Table
    DELAY IMPORT Hints/Names
    IMAGE_EXPORT_DIRECTORY
    EXPORT Address Table
    EXPORT Name Pointer Table
    EXPORT Ordinal Table
    EXPORT Names
  SECTION .data
  SECTION .idata
  SECTION .didat
  SECTION .rsrc
  SECTION .reloc
  CERTIFICATE Table
```

pFile	Data	Description	Value
000A83BC	000264C0	Function RVA	085F MapWindowPoints
000A83C0	00049930	Function RVA	0860 MenuItemFromPoint
000A83C4	0006C610	Function RVA	0861 MenuWindowProcA
000A83C8	0006C680	Function RVA	0862 MenuWindowProcW
000A83CC	0009AFE0	Function RVA	0863 MessageBeep
000A83D0	0008B260	Function RVA	0864 MessageBoxA
000A83D4	0008B2B0	Function RVA	0865 MessageBoxExA
000A83D8	0008B2E0	Function RVA	0866 MessageBoxExW
000A83DC	0008B310	Function RVA	0867 MessageBoxIndirectA
000A83E0	0008B450	Function RVA	0868 MessageBoxIndirectW
000A83E4	0008B4E0	Function RVA	0869 MessageBoxTimeoutA
000A83E8	0008B5D0	Function RVA	086A MessageBoxTimeoutW
000A83EC	0008B790	Function RVA	086B MessageBoxW
000A83F0	0009EE90	Function RVA	086C ModifyMenuA
000A83F4	0009BD00	Function RVA	086D ModifyMenuW
000A83F8	0001F600	Function RVA	086E MonitorFromPoint
000A83FC	00043480	Function RVA	086F MonitorFromRect
000A8400	00032500	Function RVA	0870 MonitorFromWindow
000A8404	00049940	Function RVA	0871 MoveWindow
000A8408	0002F850	Function RVA	0872 MsgWaitForMultipleObjects
000A840C	0002F8B0	Function RVA	0873 MsgWaitForMultipleObjectsEx
000A8410	0009B000	Function RVA	0874 NotifyOverlayWindow
000A8414	00030B70	Function RVA	0875 NotifyWinEvent
000A8418	0008DA30	Function RVA	0876 OemKeyScan
000A841C	0008DAA0	Function RVA	0877 OemToCharA
000A8420	0008DAF0	Function RVA	0878 OemToCharBuffA
000A8424	0008DB40	Function RVA	0879 OemToCharBuffW
000A8428	0008DB80	Function RVA	087A OemToCharW
000A842C	0002E360	Function RVA	087B OffsetRect
000A8430	00043A90	Function RVA	087C OpenClipboard
000A8434	0009B020	Function RVA	087D OpenDesktopA
000A8438	0009B070	Function RVA	087E OpenDesktopW

The ordinal in hex for MessageBoxW is **086B**. Convert this to decimal using a calculator in programmer mode or an online converter, and you get **2155**. Now we can change the DllImport attribute to be something like this:

```
[DllImport("user32.dll", EntryPoint = "#2155", CharSet = CharSet.Unicode)]
static extern int TotallyLegitAPI(IntPtr hWnd, string lpText, string lpCaption, uint
uType);
```

The **EntryPoint** property indicates the name or ordinal of the exported function to call, but since we want to avoid using the name, use the ordinal. We can then execute **TotallyLegitAPI** and pass in all the same parameters and it will work as expected.



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

namespace Ordinals
{
    0 references
    internal class Program
    {
        [DllImport("user32.dll", EntryPoint = "#2155", CharSet = CharSet.Unicode)]
        1 reference
        static extern int TotallyLegitAPI(IntPtr hWnd, string lpText, string lpCaption, uint uType);

        0 references
        static void Main(string[] args)
        {
            TotallyLegitAPI(IntPtr.Zero, "MessageBoxW from Ordinal", "Ordinal", 0);
        }
    }
}
```

```
Ordinal

MessageBoxW from Ordinal

OK
```

Ordinal numbers can vary across Windows versions, so ensure you're using the correct ones for your target.

MessageBox in VBA

P/Invoke function signatures are declared a little differently in VBA - instead of a DllImport attribute, we use a `Declare` directive. The rest is similar, in that we declare the parameters along with their VBA data types and the return type comes at the end.

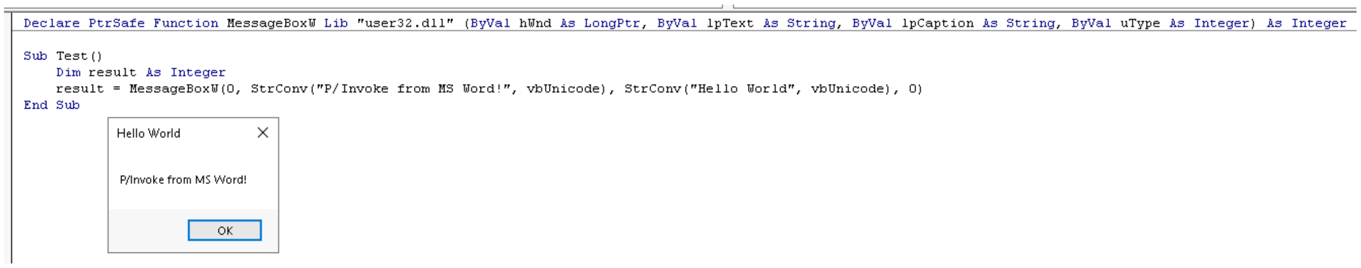
```
Declare PtrSafe Function MessageBoxW Lib "user32.dll" (ByVal hWnd As LongPtr, ByVal
lpText As String, ByVal lpCaption As String, ByVal uType As Integer) As Integer
```

Calling this function can be done in a VBA method.

```
Declare PtrSafe Function MessageBoxW Lib "user32.dll" (ByVal hWnd As LongPtr, ByVal
lpText As String, ByVal lpCaption As String, ByVal uType As Integer) As Integer

Sub Test()
    Dim result As Integer
    result = MessageBoxW(0, StrConv("P/Invoke from MS Word!", vbUnicode), StrConv("Hello
World", vbUnicode), 0)
End Sub
```

Because we're calling the unicode version, we need `StrConv` to convert the strings to the appropriate format.



```
Declare PtrSafe Function MessageBoxW Lib "user32.dll" (ByVal hWnd As LongPtr, ByVal lpText As String, ByVal lpCaption As String, ByVal uType As Integer) As Integer

Sub Test()
    Dim result As Integer
    result = MessageBoxW(0, StrConv("P/Invoke from MS Word!", vbUnicode), StrConv("Hello World", vbUnicode), 0)
End Sub
```

```
Hello World

P/Invoke from MS Word!
```

CreateProcess in VBA

As with C#, we must define the necessary structures in VBA before we can call the API. That can be done with the `Type` declaration.

```
Declare PtrSafe Function CreateProcessW Lib "kernel32.dll" (ByVal lpApplicationName As String, ByVal lpCommandLine As String, ByVal lpProcessAttributes As LongPtr, ByVal lpThreadAttributes As LongPtr, ByVal bInheritHandles As Boolean, ByVal dwCreationFlags As Long, ByVal lpEnvironment As LongPtr, ByVal lpCurrentDirectory As String, lpStartupInfo As STARTUPINFO, lpProcessInformation As PROCESS_INFORMATION) As Boolean

Type STARTUPINFO
    cb As Long
    lpReserved As String
    lpDesktop As String
    lpTitle As String
    dwX As Long
    dwY As Long
    dwXSize As Long
    dwYSize As Long
    dwXCountChars As Long
    dwYCountChars As Long
    dwFillAttribute As Long
    dwFlags As Long
    wShowWindow As Integer
    cbReserved2 As Integer
    lpReserved2 As LongPtr
    hStdInput As LongPtr
    hStdOutput As LongPtr
    hStdError As LongPtr
End Type

Type PROCESS_INFORMATION
    hProcess As LongPtr
    hThread As LongPtr
    dwProcessId As Long
    dwThreadId As Long
End Type
```

Then it's as simple as calling the API and plugging in the desired values.

```
Sub Test()
    Dim si As STARTUPINFO
    Dim pi As PROCESS_INFORMATION

    Dim nullStr As String

    Dim success As Boolean
    success = CreateProcessW(StrConv("C:\Windows\System32\notepad.exe", vbUnicode),
nullStr, 0&, 0&, False, 0, 0&, nullStr, si, pi)
End Sub
```

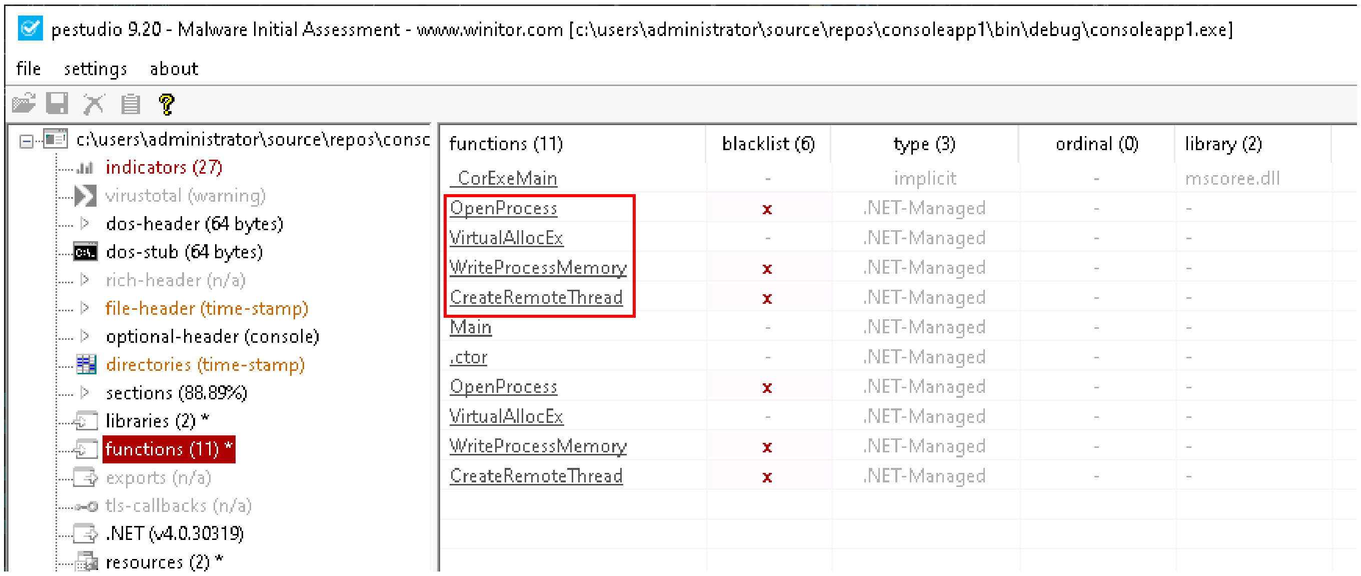
D/Invoke

Dynamic Invoke (D/Invoke) is an open-source C# project intended as a direct replacement for P/Invoke. It has a number of powerful primitives that can be combined to do some really neat things, including:

- Invoke unmanaged code without P/Invoke.
- Manually map unmanaged PE's into memory and call their associated entry point or an exported function.
- Generate syscall wrappers for native APIs.

For now, we'll focus on the first point. But one question you might have is why avoid P/Invoke?

Tools such as pestudio can inspect a compiled .NET assembly and identify "suspicious" P/Invoke usage. In the example below, this assembly calls OpenProcess, VirtualAllocEx, WriteProcessMemory and CreateRemoteThread. These APIs are synonymous with process injection and would therefore raise some alarms.


pestudio 9.20 - Malware Initial Assessment - www.winitor.com [c:\users\administrator\source\repos\consoleapp1\bin\debug\consoleapp1.exe]

c:\users\administrator\source\repos\consc
- indicators (27)
- virustotal (warning)
- dos-header (64 bytes)
- dos-stub (64 bytes)
- rich-header (n/a)
- file-header (time-stamp)
- optional-header (console)
- directories (time-stamp)
- sections (88.89%)
- libraries (2) *
- functions (11) *
- exports (n/a)
- tls-callbacks (n/a)
- .NET (v4.0.30319)
- resources (2) *

functions (11)	blacklist (6)	type (3)	ordinal (0)	library (2)
_CorExeMain	-	implicit	-	mscoree.dll
OpenProcess	x	.NET-Managed	-	-
VirtualAllocEx	-	.NET-Managed	-	-
WriteProcessMemory	x	.NET-Managed	-	-
CreateRemoteThread	x	.NET-Managed	-	-
Main	-	.NET-Managed	-	-
.ctor	-	.NET-Managed	-	-
OpenProcess	x	.NET-Managed	-	-
VirtualAllocEx	-	.NET-Managed	-	-
WriteProcessMemory	x	.NET-Managed	-	-
CreateRemoteThread	x	.NET-Managed	-	-

MessageBox with D/Invoke

D/Invoke comes as a DLL that you use as a reference in your C# project. The source code is located in `C:\Tools\DInvoke` and a pre-compiled version at `C:\Tools\DInvoke\DInvoke\DInvoke\bin\Debug\DInvoke.dll` (of course you can modify the source and compile your own version if required).

Instead of using `DllImport`, D/Invoke relies on delegates decorated with the `UnmanagedFunctionPointer` attribute. For MessageBoxW, that would look like this:

```
[UnmanagedFunctionPointer(CallingConvention.StdCall, CharSet = CharSet.Unicode)]
delegate int MessageBoxW(IntPtr hWnd, string lpText, string pCaption, uint uType);
```

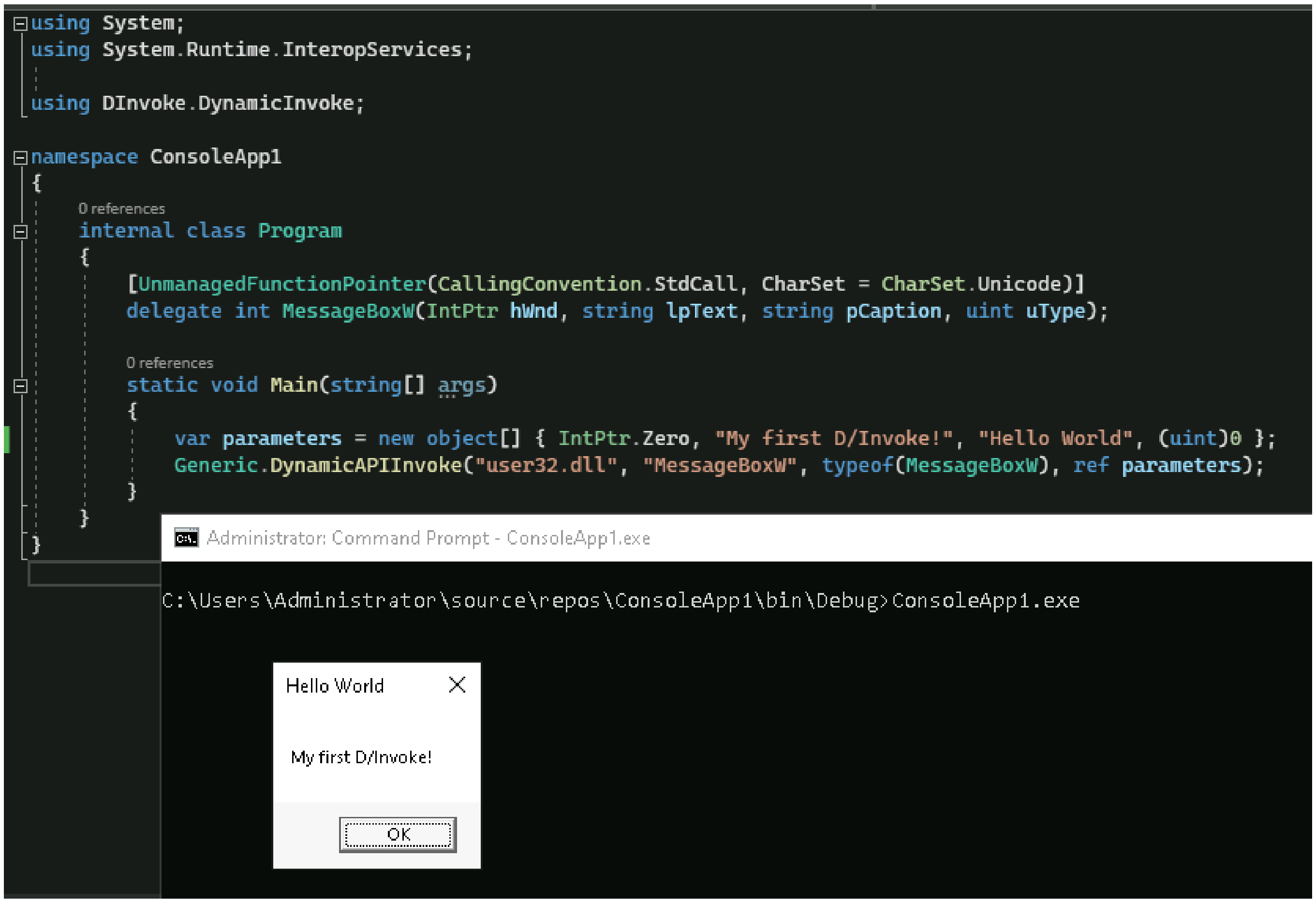
The simplest way to execute this is with `DInvoke.DynamicInvoke.Generic.DynamicAPIInvoke`. We must first define our input parameters within an `object[]` and then pass it in as a reference.

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

using DInvoke.DynamicInvoke;

namespace ConsoleApp1
{
    internal class Program
    {
        [UnmanagedFunctionPointer(CallingConvention.StdCall, CharSet = CharSet.Unicode)]
        delegate int MessageBoxW(IntPtr hWnd, string lpText, string pCaption, uint
uType);

        static void Main(string[] args)
        {
            var parameters = new object[] { IntPtr.Zero, "My first D/Invoke!", "Hello
World", (uint)0 };
            Generic.DynamicAPIInvoke("user32.dll", "MessageBoxW", typeof(MessageBoxW),
ref parameters);
        }
    }
}
```



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

using DInvoke.DynamicInvoke;

namespace ConsoleApp1
{
    0 references
    internal class Program
    {
        [UnmanagedFunctionPointer(CallingConvention.StdCall, CharSet = CharSet.Unicode)]
        delegate int MessageBoxW(IntPtr hWnd, string lpText, string pCaption, uint uType);

        0 references
        static void Main(string[] args)
        {
            var parameters = new object[] { IntPtr.Zero, "My first D/Invoke!", "Hello World", (uint)0 };
            Generic.DynamicAPIInvoke("user32.dll", "MessageBoxW", typeof(MessageBoxW), ref parameters);
        }
    }
}
```

Administrator: Command Prompt - ConsoleApp1.exe

```
C:\Users\Administrator\source\repos\ConsoleApp1\bin\Debug>ConsoleApp1.exe
```

Hello World

My first D/Invoke!

Another method is to use `GetLibraryAddress` and `GetDelegateForFunctionPointer`. This is typically more convenient if you need to call the same API more than once.

```
static void Main(string[] args)
{
    var address = Generic.GetLibraryAddress("user32.dll", "MessageBoxW");
    var messageBoxW = (MessageBoxW) Marshal.GetDelegateForFunctionPointer(address,
typeof(MessageBoxW));

    messageBoxW(IntPtr.Zero, "Box 1", "Box 1", 0);
    messageBoxW(IntPtr.Zero, "Box 2", "Box 2", 0);
}
```

One thing D/Invoke is good at is providing multiple ways to achieve the same goal, which provides lots of flexibility. And if you open this assembly in pestudio, it won't see that we're using the MessageBoxW API.

Furthermore, `GetLibraryAddress` has an overload that will take an ordinal.

```
var address = Generic.GetLibraryAddress("user32.dll", 2155);
```

CreateProcess with D/Invoke in CSharp Challenge

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

using DInvoke.DynamicInvoke;

namespace ConsoleApp1
{
    internal class Program
    {
        [StructLayout(LayoutKind.Sequential)]
        public struct STARTUPINFO
        {
            public int cb;
            public IntPtr lpReserved;
            public IntPtr lpDesktop;
            public IntPtr lpTitle;
            public int dwX;
            public int dwY;
            public int dwXSize;
            public int dwYSize;
            public int dwXCountChars;
            public int dwYCountChars;
            public int dwFillAttribute;
            public int dwFlags;
            public short wShowWindow;
            public short cbReserved2;
            public IntPtr lpReserved2;
            public IntPtr hStdInput;
            public IntPtr hStdOutput;
            public IntPtr hStdError;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct PROCESS_INFORMATION
        {
            public IntPtr hProcess;
            public IntPtr hThread;
            public int dwProcessId;
            public int dwThreadId;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct SECURITY_ATTRIBUTES
        {
            public int nLength;
            public IntPtr lpSecurityDescriptor;
            public bool bInheritHandle;
        }
        [UnmanagedFunctionPointer(CallingConvention.StdCall, CharSet = CharSet.Unicode)]
        public delegate bool CreateProcessW(
            string lpApplicationName,
            string lpCommandLine,
            ref SECURITY_ATTRIBUTES lpProcessAttributes,
            ref SECURITY_ATTRIBUTES lpThreadAttributes,
            bool bInheritHandles,
            uint dwCreationFlags,
            IntPtr lpEnvironment,
            string lpCurrentDirectory,
            ref STARTUPINFO lpStartupInfo,
            out PROCESS_INFORMATION lpProcessInformation);

        static void Main(string[] args)
        {
            var si = new STARTUPINFO();
            si.cb = Marshal.SizeOf(si);

            var pa = new SECURITY_ATTRIBUTES();
            pa.nLength = Marshal.SizeOf(pa);

            var ta = new SECURITY_ATTRIBUTES();
            ta.nLength = Marshal.SizeOf(ta);

            var pi = new PROCESS_INFORMATION();

            object[] parameters =
            {
                "C:\\Windows\\System32\\calc.exe", null, pa, ta, false, (uint)0,
IntPtr.Zero,
                "C:\\Windows\\System32", si, pi
            };

            var success = (bool)Generic.DynamicAPIInvoke("kernel32.dll",
"CreateProcessW", typeof(CreateProcessW), ref parameters);

            if (success)
            {
                pi = (PROCESS_INFORMATION)parameters[9];
                Console.WriteLine("Process created with PID: {0}", pi.dwProcessId);
            }
            else
            {
                Console.WriteLine("Failed to create process. Error code: {0}.",
Marshal.GetLastWin32Error());
            }
        }
    }
}
```