Win32 API Function Interoperability

Bringing the low level higher

Motivations

- You want to do the following:
  - Interact with unmanaged functions in PowerShell
  - You need to create:
    - Enums - Only natively supported in CDXML and PSv5 Classes
    - Structs
- Why?
  - Functionality doesn’t exist in PowerShell or .NET
  - PowerShell wrapper for 3rd party DLL
  - Interfacing with drivers
  - Interacting with malware
  - Writing malware

What is Platform Invoke (P/Invoke)?

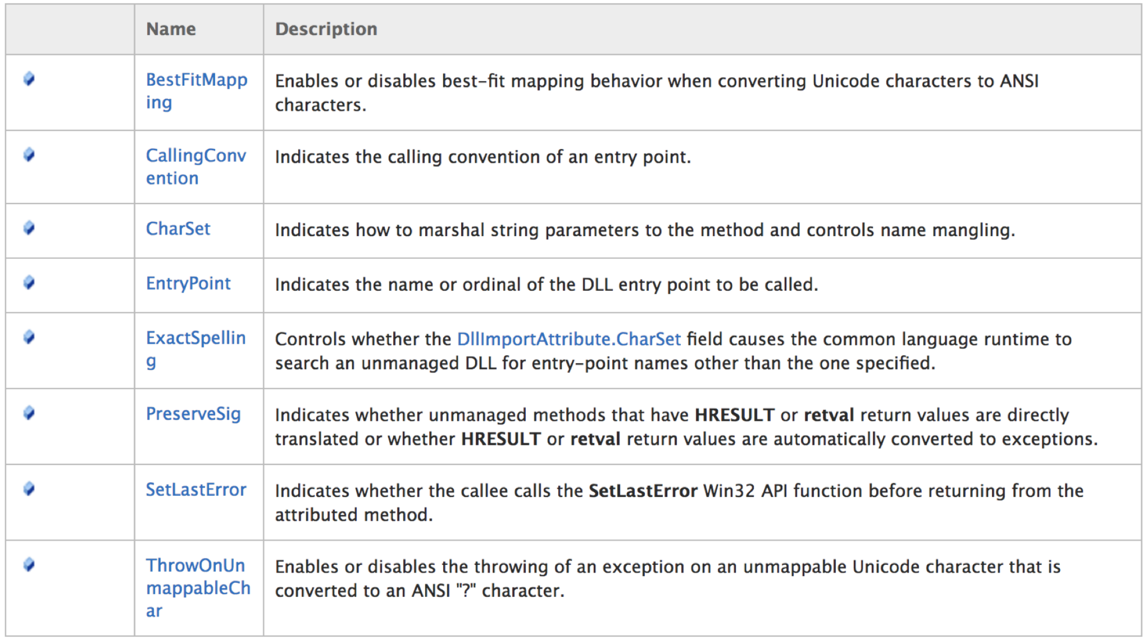
- "Platform Invoke Services (P/Invoke) allows managed code to call unmanaged functions that are implemented in a DLL"[1]
- Marshalling
  - The process of converting one object type representation to another
  - Typical in converting types between unmanaged and managed types
- Example:
  - Marshalling provides a mechanism to automatically convert a System.String (managed) to an LPCSTR (unmanaged) and vice versa.

[1]https://msdn.microsoft.com/en-us/library/aa288468(v=vs.71).aspx

Background - Calling Win32 Functions

- P/Invoke and the DllImportAttribute are the primary means of interfacing with Win32 functions

```
[DllImport("kernel32.dll", CharSet = CharSet.Ansi, SetLastError = true)]
internal static extern SafeFileHandle CreateFile
    (
        string fileName,
        [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,
        [MarshalAs(UnmanagedType.U4)] FileShare fileShare,
        IntPtr securityAttributes,
        [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,
        int flags,
        IntPtr template
    );
```



	Name	Description
◆	BestFitMapping	Enables or disables best-fit mapping behavior when converting Unicode characters to ANSI characters.
◆	CallingConvention	Indicates the calling convention of an entry point.
◆	CharSet	Indicates how to marshal string parameters to the method and controls name mangling.
◆	EntryPoint	Indicates the name or ordinal of the DLL entry point to be called.
◆	ExactSpelling	Controls whether the DllImportAttribute.CharSet field causes the common language runtime to search an unmanaged DLL for entry-point names other than the one specified.
◆	PreserveSig	Indicates whether unmanaged methods that have **HRESULT** or **retval** return values are directly translated or whether **HRESULT** or **retval** return values are automatically converted to exceptions.
◆	SetLastError	Indicates whether the callee calls the **SetLastError** Win32 API function before returning from the attributed method.
◆	ThrowOnUnmappableChar	Enables or disables the throwing of an exception on an unmappable Unicode character that is converted to an ANSI "?" character.

Background - Enums in .NET

- A special class that denotes a series of named constants
  - Make constant values human-readable
- enum colors {RED = 1, ORANGE, YELLOW};
- Approved Enum Constant Types:
  - byte, sbyte, short, ushort, int, uint, long, ulong
- [Flags] Attribute implies it should be implemented as a bitfield
- An Enum Class provides special methods for free:
  - Parse
  - TryParse
  - HasFlag
  - Etc.

Background - Structs in .NET

- A special class comprised of a logical grouping of properties
- Can have “Getter” and “Setter” methods
- Attributes may be applied to help with Marshalling
    - Field Alignment
    - Non-default Packing
    - Implicit vs. Explicit Layout
    - Etc.

P/Invoke Method (1/4) - Add-Type

- Pros:
  - Easiest
    - Signatures can be taken directly from .NET or pinvoke.net
- Cons:
  - Add-Type in PowerShell built on .NET Core doesn’t have all the same assemblies as .NET for Windows
    - Nano Server
    - IOT Core
    - Linux
    - OSX
  - Built on csc.exe
    - Leaves unnecessary compilation artifacts on the file system

P/Invoke Method (2/4) - Non-Public .NET

- Pros
  - Relatively easy to implement
  - Minimal additional code
- Cons
  - .NET doesn’t contain all possible desired functions
  - Microsoft will make no guarantees that the P/Invoke signature won’t change
- Note:
  - If possible, find viable public interfaces to the non-public P/Invoke signature

P/Invoke Method (3/4) - Reflection

- Pros
  - Does not have the same forensic artifacts that Add-Type does
  - Code generation is more dynamic in nature
- Cons
  - Can be complicated
  - Excess code

P/Invoke Method (4/4) - PSReflect

- https://github.com/mattifestation/psreflect
- Pros
    - Solves the complexity of the Reflection method
    - Intuitive “Domain Specific Language” for defining:
        - Enums
        - Structs
        - P/Invoke Function Signatures
- Cons
    - Your code will have a PSReflect dependency

PSReflect - Basics

- All enums, structs, function definitions in PSReflect have to be attached to an in-memory module.
- Use New-InMemoryModule

```
$Module = New-InMemoryModule -ModuleName Win32
```

PSReflect - Enums

```
$MessageBoxStatus = psenum $Module MessageBoxStatus Int32 @{
    IDABORT = 3
    IDCANCEL = 2
    IDCONTINUE = 11
    IDIGNORE = 5
    IDNO = 7
    IDOK = 1
    IDRETRY = 4
    IDTRYAGAIN = 10
    IDYES = 6
}

[MessageBoxStatus]::IDABORT
```

PSReflect - Structs

```
$SYSTEM_INFO = struct $Module SYSINFO.SYSTEM_INFO @{
    ProcessorArchitecture = field 0 UInt32 # i.e. DWORD
    Reserved = field 1 UInt16 # i.e. WORD
    PageSize = field 2 UInt32 # i.e. DWORD
    MinimumApplicationAddress = field 3 IntPtr # i.e. LPVOID
    MaximumApplicationAddress = field 4 IntPtr # i.e. LPVOID
    ActiveProcessorMask = field 5 IntPtr # i.e. DWORD_PTR
    NumberOfProcessors = field 6 UInt32 # i.e. DWORD
    ProcessorType = field 7 UInt32 # i.e. DWORD
    AllocationGranularity = field 8 UInt32 # i.e. DWORD
    ProcessorLevel = field 9 UInt16 # i.e. WORD
    ProcessorRevision = field 10 UInt16 # i.e. WORD
}
```

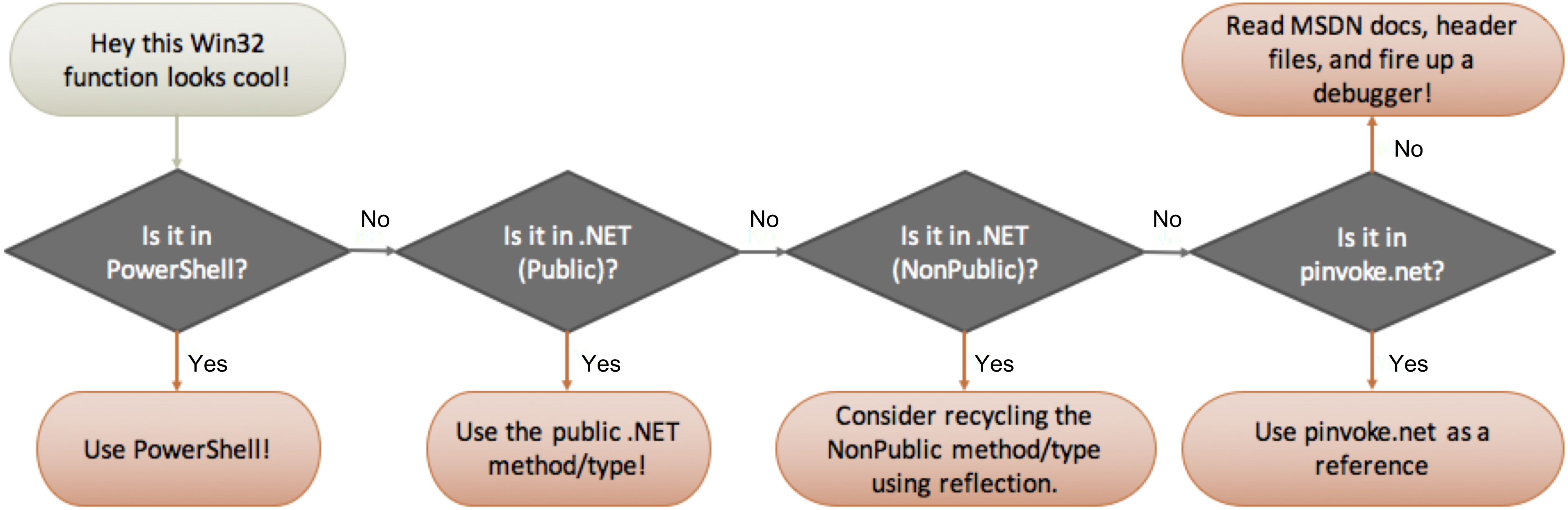
PSReflect - Function Definitions

```
$Arguments = @{
    Namespace = 'Win32Functions'
    DllName = 'Kernel32'
    FunctionName = 'MyGetModuleHandle'
    EntryPoint = 'GetModuleHandle'
    ReturnType = ([Intptr])
    ParameterTypes = @([String])
    SetLastError = $True
    Module = $Module
}

$Type = Add-Win32Type @Arguments

[Win32Functions.Kernel32]::MyGetModuleHandle('ntdll.dll')
```

PSReflect - Function Definitions

```
$FunctionDefinitions = @(
    (func kernel32 GetProcAddress ([IntPtr]) @([IntPtr], [String]) -SetLastError),
    (func kernel32 GetModuleHandle ([Intptr]) @([String]) -SetLastError),
    (func ntdll RtlGetCurrentPeb ([IntPtr]) @())
)

$Types = $FunctionDefinitions | Add-Win32Type -Module $Module -Namespace
'Win32'
$Kernel32 = $Types['kernel32']
$Ntdll = $Types['ntdll']
```

P/Invoke Signature Dev Decision Model



Hey this Win32 function looks cool!
Is it in PowerShell?
Yes
Use PowerShell!
No
Is it in .NET (Public)?
Yes
Use the public .NET method/type!
No
Is it in .NET (NonPublic)?
Yes
Consider recycling the NonPublic method/type using reflection.
No
Is it in pinvoke.net?
Yes
Use pinvoke.net as a reference
No
Read MSDN docs, header files, and fire up a debugger!

Primitive Data Type Equivalents

BOOL	→	[Bool]
BYTE	→	[Byte]
CHAR	→	[Char]
DWORD	→	[UInt32]
HANDLE	→	[IntPtr]
HRESULT	→	[Int32]
INT16	→	[Int16]
INT32	→	[Int32]
LONG	→	[Int32]
LONGLONG	→	[Int64]
LPCSTR	→	[String]
LPCWSTR	→	[String]
LPSTR	→	[String]
LPWSTR	→	[String]
NTSTATUS	→	[Int32]
QWORD	→	[UInt64]
SIZE_T	→	[UIntPtr]
WORD	→	[UInt16]

Pointer Type Equivalents

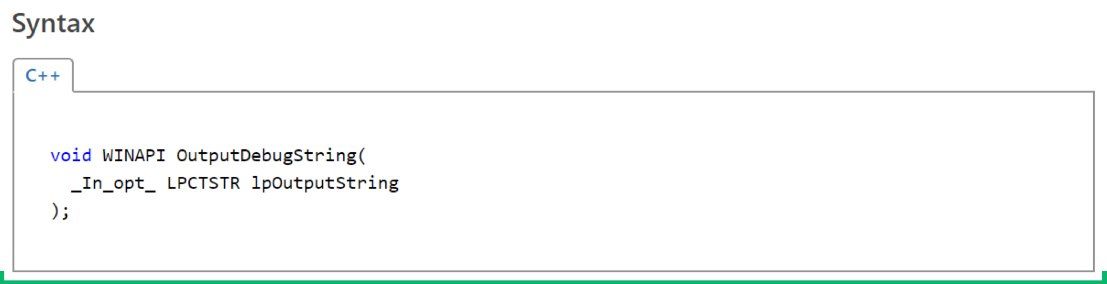
Just call the MakeByRefType Method

- PDWORD → [UInt32].MakeByRefType()
- PHANDLE → [IntPtr].MakeByRefType()
- Etc.

- Pointer type parameters require the [Ref] accelerator when arguments are passed

Win32 Function Demo

- We’re going to apply the P/Invoke signature decision model to a target Win32 API function we want to interact with: kernel32!OutputDebugString
- Why? It’s a straightforward API for demo purposes and it’s used in .NET in various ways.
- Debug output can be viewed with dbgview.exe in Sysinternals
- See OutputDebugString.ps1 to follow along with the demo



```cpp
void WINAPI OutputDebugString(
  _In_opt_ LPCTSTR lpOutputString
);
```

Win32 Function Demo

Decision model questions:

1. Is there a PowerShell cmdlet that calls it?
2. Is there a public .NET interface?
3. Is there an internal .NET interface we can borrow?
4. Do we need to write a P/Invoke signature for it?
    a. Is Add-Type acceptable?
    b. If not, do we write definition using reflection?
    c. Do we write a definition using PSReflect?

This would be a good time to take a break and attempt Lab: P/Invoke

PSReflect - Demo

- Develop a PSReflect signature for the kernel32!GetSystemInfo function.
- Why? It's a simple function that outputs a struct that also needs to be constructed.
- It outputs a SYSTEM_INFO structure that can be useful.
- Follow along with the solution:
    - GetSystemInfo.ps1

PSReflect - Demo

PSReflect signature development strategy:

- Start with MSDN docs
- Look for a C# P/Invoke signature within .NET or pinvoke.net
- Start building out the individual components necessary. Look at existing PSReflect examples! We still do this all the time.
- Experiment a lot. This is both an art and a science. The .NET marshaler is not always intuitive.

This would be a good time to take a break and attempt

Lab: PSReflect

PSReflect Functions

- PowerShell module that implements a community repository of PSReflect defined:
  - enums
  - structs
  - function definitions
- Provides a reference for writing new PSReflect function definitions
  - Similar to pinvoke.net, but for PSReflect
- Module > 100 free Win32 PowerShell functions
- Includes example scripts that integrate multiple functions together

PSReflect-Functions Demo

- Problem:
  - We want to list Ticket Granting Tickets in all Logon Sessions
  - To do this, we must be running as NT AUTHORITY\SYSTEM
  - We must impersonate the SYSTEM account
- The following API functions might help us:
  - OpenProcess
  - OpenProcessToken
  - DuplicateToken
  - ImpersonateLoggedOnUser
- Luckily all of the functions mentioned above have PowerShell function wrappers in PSReflect-Functions
- Let’s check out how easy it is to use them!!