Einführung

Microsoft hat das .NET Framework mit einem reichhaltigen Satz an Werkzeugen und API's ausgerüstet um den Entwickler bei seiner produktiven Arbeit bestmöglich zu unterstützen. Doch das .NET Framework ist noch relativ jung und es gibt Funktionen die noch nicht in das Framework integriert wurden. Darüber hinaus existiert eine riesige Menge an *alten* Quellcodes die direkt in nativen Maschinencode übersetzt werden ohne jemals mit einer Laufzeitumgebung wie der CLR in Kontakt zu geraten. Aus diesen Gegebenheiten heraus resultiert ein scheinbares Problem. Firmen und Entwickler möchten einerseits die neuen Technologien des .NET Frameworks unmittelbar nutzen, andererseits können sie es sich nicht leisten ihre Produkte, Codes und Projekte von heute auf morgen in Richtung .NET zu portieren. Daneben gibt es aber auch viele Anwendungen die für bestimmte Prozessoren optimiert wurden und deren Portierung nach .NET mit einem erheblichen Aufwand verbunden wäre. Bevor wir uns ansehen wie das .NET Framework und C# mit diesem
Problem
umgehen, werfen wir einen Blick auf die CLR.

Hintergründe zur CLR

Die .NET Laufzeitumgebung, die Common Language Runtime, ist die virtuelle Maschine die für die Ausführung aller .NET Anwendungen verantwortlich ist. Bei der CLR handelt es sich um eine Microsoft spezifische Implementierung der CLI. Die CLR stellt das gesamte Programmiermodell bereit, das alle .NET Anwendungsarten nutzen. Sie beinhaltet an eigenen Komponenten Dateilader, Speichermanager (Garbage Collector), Sicherheitssystem, Threadpool und so weiter. Der integrierte JIT-Compiler kompiliert und *verwaltet* den managed Code - siehe auch
Verwalteter Code hinter den Kulissen
- die so genannte MSIL in den entsprechenden nativen Maschinencode, der vom Prozessor anschließend ausgeführt werden kann. Alle .NET Sprachen werden stets in diesen einheitlichen Zwischencode übersetzt. Auf diese Weise stellt die Laufzeitumgebung die semantische Interoperabilität zwischen den Sprachen sicher. Die automatische Speicherverwaltung, Verifizierung der Typsicherheit und Verwaltung des Thread-Pools garantieren das der Code in einer sicheren Ausführungs-Umgebung läuft.

Im Gegensatz zu managed Code läuft unmanaged Code nicht unter dieser Laufzeitumgebung. Der Code wird unmittelbar in nativen Maschinencode übersetzt und ausgeführt. Komponenten wie COM Objekte und DLL's können auf diese Weise von den Vorteilen die eine Laufzeitumgebung, wie die CLR sie darstellt, nicht profitieren. Aus dem Prinzip des Zwischencodes der zunächst einmal von einer virtuellen Maschine ausgeführt werden muss ergeben sich aber auch Nachteile. Insbesondere die zwangsläufig schlechtere Ausführungsgeschwindigkeit beim Start der Anwendung verglichen mit nativen Anwendungen

ist ein Handicap. Man sollte dies jedoch nicht pauschal verallgemeinern. Im Gegensatz zu nativ laufenden Code kann der JIT-Compiler den Zwischencode dynamisch *anpassen*. So kann er beispielsweise konkrete Aussagen zur Plattform treffen und den resultierenden CPU-Code explizit beeinflußen.

Interoperabilität mit unmanaged Code

Dieser Artikel zeigt die Details der *Platform Invocation Services* die vom .NET Framework zur Verfügung gestellt werden. Die Platform Invoke Facility agiert als eine Brücke zwischen managed und unmanaged Code. Ursprünglich sollte der Dienst nur den Zugriff auf die native Windows API ermöglichen, wurde später im Sinne der Entwickler insofern erweitert, als dass mit ihrer Hilfe nun auch Funktionen aus jeder beliebigen DLL aufgerufen werden können. Das .NET Framework stellt zwei Dienste für die Interoperabilität mit unmanaged Code zur Verfügung. Diese sind:

1. **Platform Invocation Service:** Dieser Dienst ermöglicht es aus dem managed Code Funktionen aufzurufen die von Programmbibliotheken, wie z.B. der Win32 API zur Verfügung gestellt werden.
2. **COM Interop Services:** Ermöglicht es direkt mit COM Objekten über COM Schnittstellen und COM Clienten zu interagieren. Es gibt zwei Wege um COM Komponenten mit managed Code zu nutzen.

- Um OLE Automation kompatible COM Komponenten aufzurufen sollten Sie Interop oder tlbimp.exe verwenden. Die CLR kümmert sich um die Aktivierung der COM Komponenten und um das Parameter Marshaling.
- Für IDL basierende COM Komponenten nutzen Sie *It Just Works* und C++. Jede öffentliche verwaltete Klasse die IUnknown, IDispatch und andere Standardschnittstellen implementiert kann aus unverwalteten Code heraus über COM Interop aufgerufen werden.

Das so genannte Marshaling (engl. ordnen, regeln) ist verantwortlich für den geregelten Austausch der Argumente (Integer, Strings, Arrays, Strukturen...) und Return Werte zwischen managed und unmanaged Code. Sowohl P/Invoke als auch COM Interop machen exzessiven Gebrauch vom Interoperabilität Marshaling um die Daten zwischen Aufrufer und Aufgerufenen auszutauschen. Der Interop Marshaler regelt dabei die Daten zwischen dem Common Language Runtime Heap und dem unverwalteten, dem unmanaged Heap. Interop Marshaling ist eine Laufzeit Aktivität die vom Marshaling Dienst (engl. Service) der Common Language Runtime durchgeführt wird. Die Methoden werden von der .NET Framework-Klasse Marshal zur Verfügung gestellt.

P/Invoke

Platform Invoke oder kurz P/Invoke ermöglicht es auf sehr einfache Art und Weise unverwaltete Funktionen die in nativen Dynamic Link Libraries implementiert sind aus der CLR heraus aufzurufen. P/Invoke erlaubt es Ihnen eine statische Methodendeklaration auf einen PE COFF Eintrittspunkt abzubilden der über LoadLibrary/GetProcAddress aufgelöst werden kann. P/Invoke verwendet eine verwaltete Methodendeklaration um den Stack Frame zu beschreiben, so wie beim Java Native Interface (JNI) und J/Direct, aber mit der Bedingung das der Funktionskörper von einer externen nativen DLL zur Verfügung gestellt wird. Wie auch immer, P/Invoke ist im Gegensatz zu JNI besonders nützlich um DLL's zu importieren die nicht mit der CLR geschrieben wurden. Sie markieren dazu einfach die statische Methodendeklaration mit dem Schlüsselwort *static extern* und verwenden die Attributklasse DllImport aus dem FCL Namensraum InteropServices um zu verdeutlichen das die Methode in einer externen nativen DLL definiert ist. Sobald es an der Zeit ist die Methode aufzurufen teilt das DllImport Attribut der CLR mit welche Argumente es an LoadLibrary und GetProcAddress übergeben muss. Das eingebaute C# Attribut DllImport ist einfach nur ein Alias für System.Runtime.InteropServices.DllImport.

[code xml:lang="csharp"]namespace System.Runtime.InteropServices { // Zeigt an ob die Attributmethode durch eine unmanaged // DLL als statischer Eintrittspunkt zur Verfügung gestellt wird [AttributeUsage(64, Inherited = false)] [ComVisible(true)] public sealed class DllImportAttribute : Attribute { public bool BestFitMapping; public CallingConvention CallingConvention; public CharSet CharSet; public string EntryPoint; public bool ExactSpelling; public bool PreserveSig; public bool SetLastError; public bool ThrowOnUnmappableChar; public DllImportAttribute(string dllName); public string Value { get; } } }[/code]

Das DllImport Attribut nimmt unterschiedliche Parameter entgegen. Der Dateiname der DLL muss aber stets übergeben werden. Er wird von der Laufzeit benötigt um LoadLibrary aufzurufen noch bevor der eigentliche Methodenaufruf erfolgt. Bis der EntryPoint Parameter an DllImport übergeben wird ist der symbolische Name der Methode der String der für den Aufruf von GetProcAddress verwendet wird. In der kernel32.dll gibt es beispielsweise zwei Wege um die Funktion Sleep aufzurufen. Die erste Methode ist abhängig vom Namen der C# Funktion die mit dem Symbolnamen in der DLL übereinstimmt. Die zweite Methode ist hingegen abhängig vom EntryPoint Parameter.

[code xml:lang="csharp"]using System.Runtime.InteropServices; public class K32Wrapper { [DllImport("kernel32.dll")] public extern static void Sleep(uint msec); [DllImport("kernel32.dll", EntryPoint = "Sleep")] public extern static void Doze(uint msec); [DllImport("user32.dll")] public extern static uint MessageBox(int hwnd, String m, String c, uint flags); [DllImport("user32.dll", EntryPoint="MessageBoxW", ExactSpelling=true, CharSet=CharSet.Unicode)] public extern static uint UniBox(int hwnd, String m, String c, uint flags); }[/code]

Ein weiterer Parameter der gesetzt werden muss ist CharSet sobald die Methode mit Strings

arbeitet. Das bedeutet ob ANSI oder Unicode verwendet werden soll. Dies ist notwendig um zu kontrollieren wie der String Datentyp übersetzt wird damit der unmanaged Code anschließend mit diesem arbeiten kann. Der CharSet Parameter von DllImport erlaubt es entweder ANSI (CharSet.Ansi) oder Unicode (CharSet.Unicode) zu spezifizieren. Sie können dies auch über CharSet.Auto der Plattform überlassen, die je nachdem ob es sich um Windows NT oder um Windows 9x handelt automatisch den Zeichensatz festlegt. Diese Methode ähnelt dem TCHAR Datentyp, der in C/C++ Win32 verwendet wird um die eigene Anwendung Unicode verträglich zu gestalten. Jedoch mit dem Unterschied das der Zeichensatz und die verwendete API beim Laden bestimmt werden und nicht bei der Kompilierung. Dies hat den Vorteil das ein einmal kompiliertes .NET Programm theoretisch auf allen Windows Versionen ohne Probleme läuft.

Um die Aufrufkonventionen und Zeichensätze anzuzeigen besitzt die Windows Plattform eine Reihe an so genannten *Name Mangling* Schemen. Das Name Mangling oder auch Name Decoration (Namens Dekoration) ist eine Technik um den Symbolnamen einer Funktion eindeutig im Maschinencode zu kennzeichnen. So ergibt sich in der Computertechnik teilweise ein Problem mit Namenskonflikten wie das folgende C++-Beispiel zeigt:

[code xml:lang="csharp"]int f (void) { return 1; } int f (int) { return 0; } void g (void) { int i = f(), j = f(0); }[/code]

Bei der Übersetzung in eine C-Funktion die anschließend in einer DLL aufgerufen werden kann würde dies in einem Fehler resultieren, da in C Funktionen mit demselben Namen nicht gestattet sind. Hier kommt das Name Mangling ins Spiel. Der Compiler übersetzt den Code und generiert je nach Signatur einen individuellen Symbolnamen. Für das oben gezeigte Beispiel könnte dieser folgendermaßen aussehen.

[code xml:lang="csharp"]int __f_v (void) { return 1; } int __f_i (int) { return 0; } void __g_v (void) { int i = __f_v(), j = __f_i(0); }[/code]

Tatsächlich implementiert der Compiler je nach Hersteller und Plattform seine eigenen Name Mangling Konventionen. In diesem Tutorial soll dies für uns aber nicht weiter relevant sein. Sobald der CharSet Parameter des DllImport Attributs auf Auto gesetzt wurde, besitzen die symbolischen Namen automatisch den Suffix W oder A, je nachdem ob der Unicode oder der ANSI Zeichensatz von der Laufzeit verwendet wird. Zusätzlich transformiert die Laufzeit das Symbol unter der Verwendung der stdcall Konvention (z.B. wird Sleep zu _Sleep@4) sofern der einfache Symbolname, also Sleep, nicht gefunden wurde. Mithilfe des Parameters ExactSpelling kann das Name Mangling unterdrückt werden.

Schlußendlich, wenn Sie Win32-Funktionen aufrufen die COM ähnliche HRESULTs verwenden, haben Sie zwei Optionen. Standardmäßig behandelt P/Invoke das HRESULT als einen einfachen 32-Bit Integer der von der Funktion zurückgegeben wird und vom Programmierer selbst auf Fehler überprüft werden muss. Eine deutlich angenehmere Methode solch eine

Funktion aufzurufen ist den Parameter PreserveSig=false an das DllImport Attribut zu übergeben. Dies verursacht das die P/Invoke Schicht den 32-Bit Integer als ein COM HRESULT behandelt und im Fehlerfall eine COMException auslöst. Da die meisten Methodenaufrufe mit P/Invoke jedoch keine HRESULTs zurückgeben ist PreserveSig standardmäßig auf true gesetzt und schützt die Signatur, so wie sie definiert wurde. In der nachfolgenden Übersicht sehen Sie alle Parameter der Attributklasse DllImport des .NET Frameworks 2.0 mitsamt kurzer Beschreibung:

Parameter	Beschreibung
BestFitMapping	Aktiviert oder deaktiviert das Best-Fit Mapping Verhalten bei der Konvertierung eines Unicode Z
CallingConvention	Spezifiziert die Aufrufkonvention. Standardmäßig ist das WinAPI, was bei 32-Bit Intel-basier
CharSet	Gibt an wie das Marshaling von String-Argumenten der Methode durchgeführt wird und kont
EntryPoint	Gibt den Namen des aufzurufenden DLL-Eintrittspunkts (der Methode) an. Falls kein Argument
ExactSpelling	Gibt an ob das DllImportAttribute.CharSet Feld verursacht das die CLR nach einem anderen
PreserveSig	Zeigt an ob die Signatur eine direkte Übersetzung des unverwalteten Eintrittspunktes ist.
SetLastError	Gibt an ob dieser Parameter auf true gesetzt ist, besteht die Möglichkeit, Marshal-GetLastWin32 Erro
ThrowOnUnmappableChar	Aktiviert oder deaktiviert das Auslösen einer Exception bei einem nicht einzuordnenden Unicode
DllName	Gibt den Namen der aufzurufenden DLL.
Value	Gibt den Namen der DLL zurück die den Eintrittspunkt enthält.

```
public bool BestFitMapping { get; }
public CallingConvention CallingConvention { get; }
public CharSet CharSet { get; }
public string EntryPoint { get; }
public bool ExactSpelling { get; }
public bool PreserveSig { get; }
public bool SetLastError { get; }
public bool ThrowOnUnmappableChar { get; }
public string DllName { get; }
public string Value { get; }
```

Funktionen aus einer DLL nutzen

Managed Code verwendet eine Code-Zugriffs-Sicherheit. Bevor auf eine Ressource zugegriffen wird oder anderweitige potentiell gefährliche Schritte durchgeführt werden überprüft die Laufzeit den Code. Mit der Einbeziehung von unmanaged Code verliert die CLR die Fähigkeit die Sicherheit der Umgebung zu gewährleisten. Konkret gesagt verlässt ihr Code bei Aufruf von unmanaged Code das Partial-trusted-Szenario und Sie geben die Typsicherheit im Programm auf. Die Laufzeit prüft ob bei allen Aufrufern im Aufrufstack die notwendige Sicherheitsstufe es erlaubt P/Invoke zu nutzen. Die entsprechenden Rechte müssen also auf der Plattform gegeben sein!

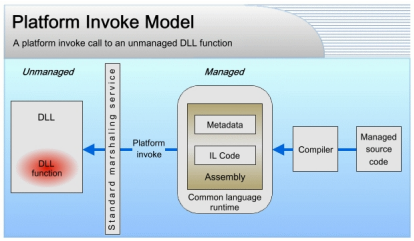
Platform Invoke ist ein Dienst der es erlaubt beliebige unverwaltete Funktionen aus DLL's aufzurufen. Es lokalisiert und ruft eine exportierte Funktion auf und regelt (engl. marshals) die Argumente. Damit eine exportierte Funktion aufgerufen werden kann, müssen folgende Schritte abgearbeitet werden.

- Die DLL Funktion muss identifiziert werden. Das bedeutet das zumindest der Funktionsname und die DLL die die Funktion beinhaltet benannt werden müssen.
- Generieren Sie eine Klasse die die DLL Funktionen enthält. Sie können eine bestehende Klasse verwenden, eine eigene für jede unverwaltete Funktion generieren oder eine Klasse schreiben die einen Satz an Funktionen beinhaltet.
- Generieren Sie die Prototypen im managed Code. Benutzen Sie in C# das DllImport Attribut um die DLL und die Funktion zu identifizieren. Markieren Sie die Methode mit den

Modifizierern static und extern.

- Rufen Sie die DLL Funktion wie gewohnt auf. Die Übergabe von Strukturen und die Implementierung von Callback Funktionen sind spezielle Fälle.

P/Invoke basiert auf Metadaten um die exportierte Funktion zu lokalisieren und ihre Argumente zur Laufzeit zu regeln. Die folgende Abbildung zeigt diesen Prozess.



Sobald P/Invoke eine unmanaged Funktion aufruft, werden sequentiell folgende Aktionen durchgeführt:

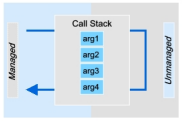
1. Die DLL mit der entsprechenden Funktion wird lokalisiert.
2. Die DLL wird in den Speicher geladen.
3. Die Adresse der Funktion im Speicher wird lokalisiert und ihre Argumente werden auf den Stack gepusht. Das Marshaling der Daten ist erforderlich. Beachten Sie das die Lokalisierung der DLL und der Adresse im Speicher nur das erste Mal wenn die Funktion aufgerufen wird geschieht.
4. Transferierung der Kontrolle zur unverwalteten Funktion.

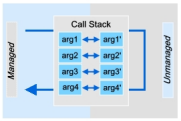
Platform Invoke löst Exceptions (Ausnahmen) aus die von der unverwalteten Funktion an den verwalteten Aufrufer übergeben werden.

Typabbildungen (Mappings)

Sobald Sie einen Aufruf raus aus der Laufzeit oder in diese hinein tätigen, werden die Parameter unverändert an den Aufrufstack übergeben. Diese Parameter sind Instanzen der Typen sowohl für die Laufzeit als auch für die *Welt* außerhalb. Der Schlüssel für das Verständnis wie Interop funktioniert liegt im Verständnis das jeder übergebene "Wert" zwei Typen hat. Einen verwalteten (engl. managed) Typ und einen unverwalteten Typ. Entscheidend ist das einige managed Typen isomorph sind, das heißt dass eine Instanz dieses Typs nicht erst konvertiert werden muss bevor es aus der Laufzeit hinaus übergeben wird. Viele Typen jedoch

sind nicht isomorph und es muss dementsprechend eine Konvertierung erfolgen, so dass der Datentyp außerhalb der Laufzeit auch richtig interpretiert werden kann.





Das Abbild links zeigt die Struktur des Call Stacks (Aufrufstacks) wenn die Parameter nur aus isomorphen Typen bestehen. Sobald ein Aufruf einer externen Routine erfolgt, welche nur isomorphe Typen entgegennimmt, ist eine Konvertierung nicht notwendig und der Aufrufende sowie der Aufgerufene können sich den Stack Frame teilen, trotz der Tatsache das die eine Seite nicht unter der Kontrolle der CLR steht. Sobald ein Parameter nicht isomorph ist, muss der Stack Frame aufgeteilt werden. Die Abbildung rechts im Bild zeigt diesen Vorgang. Sowohl der verwaltete Code der auf der Laufzeit läuft als auch der unverwaltete Code greifen auf seperate Stack Frames zu. Die beiden Stack Frames sind zueinander bidirektional. Das bedeutet das die Konvertierung der Datentypen auch in umgekehrter Richtung erfolgen kann. Beispielsweise wenn die Funktion in der unverwalteten Funktion einen Parameter zurück an den Aufrufer übergibt. Der C# Compiler erkennt anhand der Schlüsselwörter ref bzw. out in welche Richtungen die Parameter fließen.

Es steht Ihnen frei selbst festzulegen wie der übergebene Parameter verwaltet werden soll indem Sie das Attribut MarshalAs verwenden. Das Attribut zeigt an wie der verwaltete Datentyp außerhalb der Laufzeit dargestellt werden soll, also welchen Datentyp er in der unverwalteten Umgebung besitzen soll. Für die meisten Datentypen entscheidet die CLR automatisch welcher Datentyp für die Konvertierung am besten ist. Bei Bedarf können Sie dies mit dem erwähnten Attribut aber überbrücken und selbst festlegen. Das folgende Beispiel verwendet das MarshalAs Attribut um den CLR Datentypen System.String explizit in vier verschiedene bekannte Win32 Datentypen zu konvertieren.

[code xml:lang="csharp"]using System.Runtime.InteropServices; public class FooBarWrapper { // Diese Methode wrappt eine native Funktion deklariert als // void _stdcall DoIt(LPCWSTR s1, LPCSTR s2, LPTSTR s3, BSTR s4); [DllImport("foobar.dll")] public static extern void DoIt([MarshalAs(UnmanagedType.LPWStr)] String s1, [MarshalAs(UnmanagedType.LPStr)] String s2, [MarshalAs(UnmanagedType.LPTStr)] String s3, [MarshalAs(UnmanagedType.BStr)] String s4); }[/code]

Neben der Möglichkeit mit MarshalAs Typkonvertierungen auf Feld-zu-Feld Basis abzubilden besteht die Möglichkeit auch die Darstellung eines Structs oder eine Klasse festzulegen. Die Attribute StructLayout und FieldOffset erlauben es Ihnen die interne Darstellung des Structs oder der Klasse im Speicher manuell zu bestimmen. Hier sollten Sie jedoch besondere Vorsicht walten lassen.

[code xml:lang="csharp"]using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)] public struct PERSON_REP { [MarshalAs(UnmanagedType.BStr)] public String name; public double age; [MarshalAs(UnmanagedType.VariantBool)] bool dead; }[/code]

Die nachfolgende Tabelle illustriert die Eigenschaften der verwalteten und unverwalteten Datentypen, sowie ihre interne Beziehung untereinander:

Windows	.NET	Beschreibung
FLOAT	Single	Marshal As nativer Typ
DOUBLE	Double	Marshal As nativer Typ
UCHAR	Byte	Marshal As nativer Typ
BYTE	Byte	Marshal As nativer Typ
SHORT	Int16	Marshal As nativer Typ
USHORT	UInt16	Marshal As nativer Typ
INT	Int32	Marshal As nativer Typ
UINT	UInt32	Marshal As nativer Typ
INT64	Int64	Marshal As nativer Typ
UINT64	UInt64	Marshal As nativer Typ
1D Array	Array	Marshal As nativer Typ
SAFEARRAY	Array	Marshal As nativer Typ
BOOL	Boolean	Marshal As VARIANT_BOOL oder Win32 BOOL
CHAR	Char	Marshal As Win32 WCHAR oder CHAR
LPSTR	String	Marshal As Win32 LPWSTR/LPSTR oder BSTR
VARIANT	Object	Marshal As VARIANT (Nur COM Interop)

P/Invoke in Aktion

In diesem Kapitel widmen wir uns den konkreten Codebeispielen, die Ihnen die praktischen Aspekte des Interop Marshaling näher bringen sollen. Das erste Codebeispiel demonstriert den Aufruf zweier C Funktionen aus der msvcrt.dll. Die DLL repräsentiert Microsofts C Runtime Library. Die Funktion puts schreibt eine Zeichenkette auf die Standardausgabe. Anschließend wird die Funktion _flushall aufgerufen. Diese überträgt die Inhalte der Dateipuffer aller mit den File-Pointern eröffneten Streams in die entsprechenden Dateien. Dazu gehören auch die mit den File-Pointern stdin, stdout etc. eröffneten Standarddateien. Die Inhalte der Dateipuffer aller nur zum Lesen eröffneten Dateien werden gelöscht. Das Programm gibt den String "Hello World!" auf der Konsole aus.

[code xml:lang="csharp"]using System; using System.Collections.Generic; using System.Text; using System.Runtime.InteropServices; namespace Pinvoke { class Pinvoke { [DllImport("msvcrt.dll")] public static extern int puts(string c); [DllImport("msvcrt.dll")] internal static extern int _flushall(); public static void Main(string[] args) { puts("Hello World!"); _flushall(); } } }[/code]

Wie wir wissen lässt sich die Konvertierung auch explizit festlegen. Über das Attribut MarshalAs können wir den unverwalteten Typen als LPSTR bestimmen. Bei diesem Datentypen handelt es

sich um einen Zeiger auf einen Null-terminierten 8-Bit (ANSI) Zeichen String.

[code xml:lang="csharp"]using System; using System.Collections.Generic; using System.Text; using System.Runtime.InteropServices; namespace Marshal { class Marshal { [DllImport("msvcrt.dll")] public static extern int puts([MarshalAs(UnmanagedType.LPStr)] string m); [DllImport("msvcrt.dll")] internal static extern int _flushall(); public static void Main(string[] args) { puts("Hello World!"); _flushall(); } } }[/code]

In der Regel werden Sie jedoch relativ selten Funktionen aus der C Runtime Library aufrufen. Stattdessen sind es oft Funktionen aus den nativen Dynamic Link Libraries der WinAPI, die womöglich nicht vom .NET Framework in entsprechender Form bereitgestellt werden. Nachfolgend sehen Sie die Deklaration der WinAPI-Methode MoveFile. Die Funktion MoveFile benennt eine existierende Datei oder ein Verzeichnis um. Im nachfolgenden Programm wird MoveFile dazu verwendet einige Dateien in einem zuvor erstellten Verzeichnis umzubenennen. Dazu wird zunächst im Verzeichnis "C:testmedia" ein Unterverzeichnis angelegt in welches alle Dateien des Oberverzeichnises kopiert werden. Anschließend werden einige Dateien wieder gelöscht und anderen wird die Endung ".bak" mit dem direkten Aufruf der nativen WinAPI Funktion angehangen.

[code xml:lang="csharp"]class InvokeWinAPI { // Deklariere die mit P/Invoke aufzurufende WinAPI-Methode // BOOL MoveFile(// LPCTSTR lpExistingFileName, // LPCTSTR lpNewFileName //); [DllImport("kernel32.dll", EntryPoint = "MoveFile", ExactSpelling = false, CharSet = CharSet.Unicode, SetLastError = true)] static extern bool MoveFile(string sourceFile, string destinationFile); public static void Main(string[] args) { // Erzeuge eine Instanz und lass sie laufen InvokeWinAPI obj = new InvokeWinAPI(); string theDirectory = @"C:testmedia"; DirectoryInfo dir = new DirectoryInfo(theDirectory); obj.ExploreDirectory(dir); } // Ist mit einem Verzeichnisbaum aufzurufen private void ExploreDirectory(DirectoryInfo dir) { ... // P/Invoke für die WinAPI InvokeWinAPI.MoveFile(fullName, fullName + ".bak"); ...[/code]

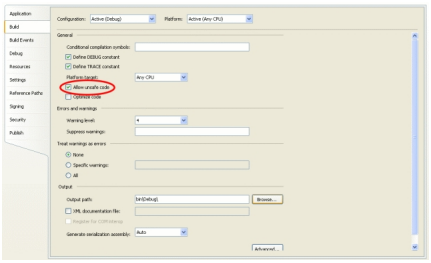
Zeiger

Alle bisherigen Beispiele haben sie konsequent verbannt und wir konnten auch gut auf sie verzichten, gemeint sind die allseits bekannten und gefürchteten Zeiger (engl. Pointer). Sie werden unter C#/.NET so gut wie nie Zeiger in ihrem Code benötigen. Auch ist vor ihrer direkten Anwendung fast immer abzuraten. Denn sobald sie mit Zeigern arbeiten können Sie den Arbeitsspeicher direkt manipulieren. Sie haben Zugriff auf die Speicheradressen und können die darin befindlichen Daten korrumpieren. Die CLR hat keine Kontrolle über die von Ihnen getätigten Aktionen und auch die Garbage Collection steht nicht mehr zur Verfügung. Ihr Code verlässt die sichere Umgebung der Laufzeit und Sie tragen ab sofort die alleinige Verantwortung für Ihr Programm. Dennoch gibt es Fälle in denen es keine andere Möglichkeit gibt, als auf Zeiger zurückzugreifen. Im besonderen trifft dies zu, wenn eine externe Funktion einen Zeiger als Parameter benötigt.

Da die Arbeit mit Zeigern in .NET als unsicher eingestuft wird, ist die Arbeit mit ihnen an spezielle Bedingungen geknüpft. So muss der Codeabschnitt in denen mit Zeigern gearbeitet wird, mit dem Schlüsselwort *unsafe* versehen werden! Im nächsten Abschnitt sehen Sie die Deklaration der beiden WinAPI-Methoden CreateFile und ReadFile. ReadFile benötigt als zweiten Parameter einen Zeiger auf einen Pufferbereich.

[code xml:lang="csharp"][DllImport("kernel32", SetLastError=true)] static extern unsafe int CreateFile(string filename, uint desiredAccess, uint shareMode, uint attributes, uint creationDisposition, uint flagsAndAttributes, uint templateFile); [DllImport("kernel32", SetLastError=true)] static extern unsafe bool ReadFile(int hFile, void* lpBuffer, int nBytesToRead, int* nBytesRead, int overlapped);[/code]

Sie müssen in den Build Einstellungen die Option "Allow unsafe code" aktivieren um den Code kompilieren zu können. Sie erhalten sonst eine Fehlermeldung vom Compiler.



Im folgenden Codeabschnitt kommt der eigentliche Zeiger ins Spiel. Das Programm liest mithilfe der Funktion ReadFile Daten aus einer Datei in einen Puffer. Die Funktion greift auf diesen Puffer über einen Zeiger zu. Da der Puffer ein verwalteter Datentyp ist, ergibt sich ein Problem. Die Garbage Collection könnte den Speicher verschieben oder löschen und unser Zeiger würde irgendwo hinzeigen. Dieses Problem wurde von Microsoft gelöst mit dem so genannten *Pinning*. Das Pinning fixiert den Speicherbereich und verhindert das der Garbage Collector die Instanz im Speicher verschiebt und das Programm auf diese Weise korrumpiert. Dazu steht sinngemäß das Schlüsselwort
fixed
(engl. fixiert) zur Verfügung.

[code xml:lang="csharp"]// BOOL ReadFile(// HANDLE hFile, // LPVOID lpBuffer, // DWORD nNumberOfBytesToRead, // LPDWORD lpNumberOfBytesRead, // LPOVERLAPPED lpOverlapped //); public unsafe int Read(byte[] buffer, int index, int count) { int bytesRead = 0; fixed (byte* bytePointer = buffer) { ReadFile(fileHandle, // hfile bytePointer + index, // lpBuffer count, // nBytesToRead &bytesRead, // nBytesRead 0); // overlapped } return bytesRead; }[/code]

Ab und zu ist auch das Schlüsselwort *stackalloc* zu finden. Es eröffnet Ihnen die Möglichkeit

innerhalb eines unsafe Blocks einen Speicherblock auf dem Stack zu allokieren, anstatt auf dem Heap.

PEVerify

Wie bereits in den vorangegangenen Kapiteln erwähnt wurde, ist die Ausführung von nativen Code in einer verwalteten Anwendung bestimmten Regeln unterworfen. Die CLR überprüft die Assembly nach unsicheren Methoden. Genauer gesagt ist es der JIT-Compiler der letztenendes bei der Kompilierung des Zwischencodes, also der IL, enscheidet ob der Assembly, in der die native Methode (Aufruf) enthalten ist, System.Security.Permissions.SecurityPermission zugewiesen wurde und ob darin das Flag SkipVerification gesetzt ist. Erst dann wird der Code in native Befehle für den Prozessor übersetzt.

In der Regel erhalten Assemblys nicht verifizierten Ursprungs, wie beispielsweise aus dem Internet geladene Anwendungen, keine absolute Erlaubnis *unsicheren* Code auf dem System auszuführen. Der JIT-Compiler löst in diesem Fall eine System.Invalid.ProgramException oder System.Security.VerificationException aus und bricht die Programmausführung ab. Der Administrator des Systems kann die entsprechenden Rechte jedoch selbst zuweisen und auf diese Weise die Ausführung gestatten.

Um festzustellen ob eine Assembly unsichere Methoden enthält und so eventuell nicht auf jedem System ausgeführt werden kann, stellt Microsoft ein Tool namens **PEVerify** im SDK zur Verfügung. Das Tool analysiert eine Assembly und teilt dem Benutzer mit ob sich unsichere Methoden in dieser befinden. PEVerify geht bei der schrittweisen Untersuchung der abhängigen Assemblys mithilfe der CLR vor. Es kommen dieselben Bindungs- und Suchregeln zur Anwendung wie beim normalen Ausführen der Assembly. Untersucht man beispielsweise die oben genannte Anwendung InvokePointer.exe, wird PEVerify folgenden Output generieren:

C:>PEVerify.exe InvokePointer.exe Microsoft (R) .NET Framework PE Verifier. Version 2.0.50727.42 Copyright (c) Microsoft Corporation. All rights reserved. [IL]: Error: [C:InvokePointer.exe : InvokePointer.InvokePointer::Read][offset 0 x00000022][found address of Byte] Expected numeric type on the stack. [IL]: Error: [C:InvokePointer.exe : InvokePointer.InvokePointer::Read][offset 0 x0000000F][found Native Int][expected address of Byte] Unexpected type on the st ack. 2 Errors Verifying InvokePointer.exe

Wie man sieht wurden zwei Fehler infolge der Verwendung von nativen Methoden in der Assembly gefunden. Es handelt sich um den Zeiger auf unseren Byte-Puffer. Mehr zu PEVerify finden Sie auf den offiziellen Seiten von Microsoft unter PEVerify . Das Tool selbst befindet sich in Ihrem Visual Studio Verzeichnis. Sie können es aber auch hier

seperat herunterladen.

Schluss

Wenn Sie im Laufe Ihrer Zeit als Programmierer mal auf eine externe Funktion in einer nativen DLL stossen, deren Deklaration Ihnen unbekannt ist und Sie diese in Ihrem C#-Code verwenden möchten, sollten Sie einen Blick auf die Seite Pinvoke.net werfen. Sie finden dort zahlreiche fertige P/Invoke Signaturen für Ihren Code. Am Ende dieses Tutorials befindet sich als Dateianhang eine Visual Studio 2005 Projektmappe mit einigen Projekten samt Quellcode zum Thema P/Invoke in C#/.NET.