.NET CLR Framework

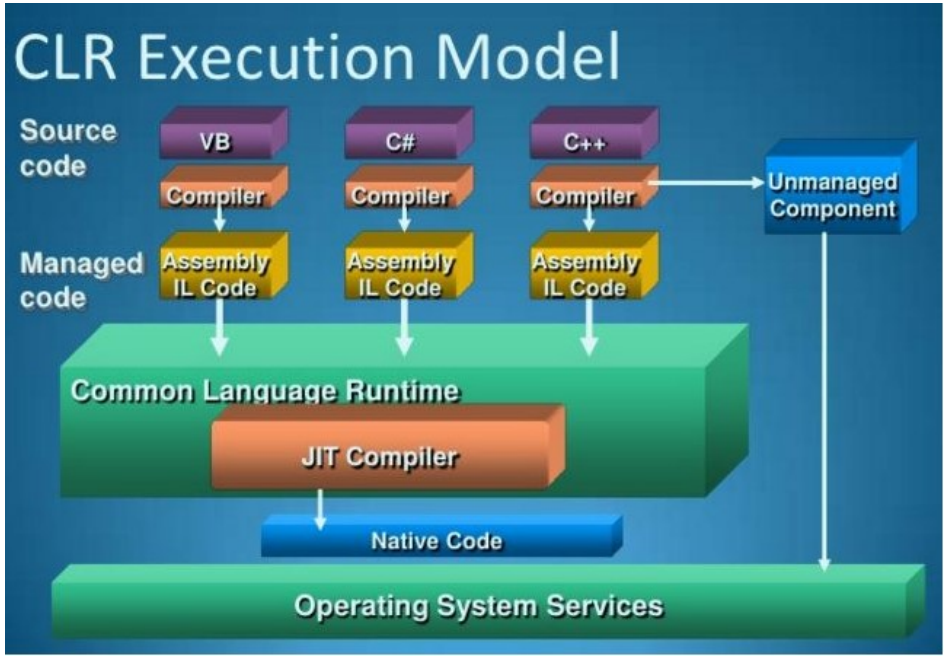
Unmanaged Hosts - Assembly Access

.NET CLR Framework

WHAT : is .NET Common Language Runtime (CLR) Framework

- The Common Language Runtime (CLR) is a an Execution Environment. Common Language Runtime (CLR)'s main tasks are to convert the .NET Managed Code to native code, manage running code like a Virtual Machine, and also controls the interaction with the Operating System.

- As part of Microsoft's .NET Framework, the Common Language Runtime (CLR) is managing the execution of programs written in any of several supported languages. Allowing them to share common object-oriented classes written in any of the languages.



.NET CLR Framework

WHAT : is .NET Common Language Runtime (CLR) Framework

- The Common Language Runtime (CLR) has the following key components in .NET
- Automatic Memory Management
- Garbage Collection
- Code Access Security
- Code Verification
- JIT Compilation of .NET code

For more details see here : http://www.csharpstar.com/top-20-dotnet-framework-interview-questions/

.NET CLR Framework

HOW : To access the CLR environment.

You need to create an **Appdomain** Object in your unmanaged environment,

An **Appdomain** provides an isolated region in which code runs inside of an existing process.

- Application domains provide an isolation boundary for security, reliability, and versioning, and for unloading assemblies. Application domains are typically created by runtime hosts, which are responsible for bootstrapping the common language runtime before an application is run.

- In **AutoIT** you can call the **_CLR_GetDefaultDomain()** function which is located in the CLR UDF

```
Func Example()
    Local $oAppDomain = _CLR_GetDefaultDomain()

    ; Create a 1D array with one integer element
    Local $aArgs = [ 128 ], $psa = CreateSafeArray( $aArgs )

    ; Create an instance of ArrayList using a parameterized constructor
    Local $pHandle
    $oAppDomain.CreateInstance_3( "mscorlib", "System.Collections.ArrayList", True, 0, 0, $psa, 0, 0, 0, $pHandle )
    ConsoleWrite( @CRLF & "$pHandle = " & $pHandle & @CRLF )

    Local $oHandle = ObjCreateInterface( $pHandle, $sIID_IObjectHandle, $sTag_IObjectHandle )
    ConsoleWrite( @CRLF & "IsObj( $oHandle ) = " & IsObj( $oHandle ) & @CRLF )

    ; Unwrap the ArrayList instance inside the ObjectHandle
    Local $oArrayList
    $oHandle.Unwrap( $oArrayList )
    ConsoleWrite( @CRLF & "IsObj( $oArrayList ) = " & IsObj( $oArrayList ) & @CRLF )

    ; Print ArrayList Capacity
;~    ConsoleWrite( @CRLF & "$oArrayList.Capacity() = " & $oArrayList.Capacity() & @CRLF )
    MsgBox(0,"""System.Collections.ArrayList","$oArrayList.Capacity() = " & $oArrayList.Capacity())
EndFunc
```

.NET CLR Framework

WHAT : is a .NET Appdomain

- **AppDomain** is designed to be called by unmanaged code, and it allows a host to inject an assembly in the current process.

  Managed code developers generally shouldn't call the an AppDomain. AppDomain's Load method returns a reference to an assembly.

- Assembly's **Load method** is the preferred way of loading an assembly into an AppDomain. But Assembly Load method can have performance drawback though.

  TIP : By the way, the **LoadFrom method** allows you to pass a URL as the argument. Here is an example.

  Assembly a = Assembly.LoadFrom**(@"http://Wintellect.com/SomeAssembly.dll");**

- **Metadata** is stored in a bunch of **tables**. When you build an assembly or a module, the compiler that you're using creates a type definition table, a field definition table, a method definition table, and so on...

- The **System.Reflection** namespace contains several types that allow you to write code that reflects over (or parses) these **metadata tables**.

In effect, the types in this namespace offer an object model over the metadata contained in an assembly or a module. Keep in mind that you can create **Multiple Appdomains in 1 Host Process** !

.NET CLR Framework

WHAT: is Reflection in .NET CLR

- **CLR Reflection :**

  - Many of the services available in .NET and exposed via C# (such as late binding, serialization, remoting, attributes, etc.) depend on the presence of **Metadatas**.

  - Manipulating existing types via their metadata termed ”***Reflection”*** and is done using a rich set of types in the ***System.Reflection*** namespace.

  - Creating new types is termed ***Reflection.Emit***, and is done via the types in the ***System.Reflectio.Emit*** namespace.

  - The classes in the ***Reflection namespace, along with the System.Type and System.TypedReference classes***, provide support for examining and interacting with the metadata.

.NET CLR Framework

WHAT: is Late Binding in .NET CLR

- **CLR Late Binding :**

  - **Reflection** can also perform late binding, in which the application dynamically loads, instantiates and uses a type at runtime.
    => This provides greater flexibility at the expense of invocation overhead.

  - The **Activator** class contains **four methods**, all static, which you can use to **create objects locally** or **remotely**, or to **obtain references** to **existing objects**.

  - The four methods are **CreateComInstanceFrom**, **CreateInstanceFrom**, **GetObject**, and **CreateInstance**:

    1. **CreateComInstanceFrom** : Used to create instances of COM objects.
    2. **CreateInstanceFrom** : Used to create a reference to an object from a particular assembly and type name.
    3. **GetObject** : Used when marshaling objects.
    4. **CreateInstance** : Used to create local or remote instances of an object.

.NET CLR Framework

WHAT: is Late Binding in .NET CLR

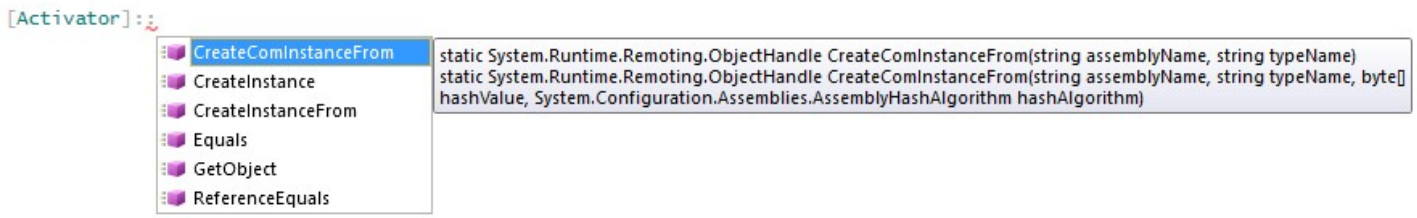
- **CLR Late Binding**
  - The **Activator** class :

    Example in PowerShell

```
[Activator]::CreateInstance([Type]::GetTypeFromCLSID([Guid]'{DCB00C01-570F-4A9B-8D69-199FDBA5723B}')).IsConnectedToInternet
```

  - The four methods are **CreateComInstanceFrom**, **CreateInstanceFrom**, **GetObject**, and **CreateInstance**:

    See **PowerShell Code Intellisense**



.NET CLR Framework

HOW: to Constructing an Instance of a Type Using Reflection

- **CLR Late Binding**

  - The **Activator** class :

    1. **CreateComInstanceFrom** : Used to create instances of COM objects.

    This is the same as **CreateInstanceFrom** , except for CreateComInstanceFrom method will check if the type is an COM visible type first.

    Example in AutoIT :

```
Func Example()

    Local $oAssembly = _CLR_LoadLibrary("mscorlib")
    ConsoleWrite("$oAssembly: " & IsObj($oAssembly) & @CRLF)

    Local $pAssemblyType = 0
    $oAssembly.GetType_2("System.Activator", $pAssemblyType)
    ConsoleWrite("$pAssemblyType = " & Ptr($pAssemblyType) & @CRLF)

    Local $oAssemblyType = ObjCreateInterface($pAssemblyType, $sIID_IType, $sTag_IType)
    ConsoleWrite("IsObj( $oAssemblyType ) = " & IsObj($oAssemblyType) & @TAB & @CRLF)

    Local $aText[] = ["C:\Program Files (x86)\Microsoft.NET\Primary Interop Assemblies\Microsoft.mshtml.dll", "mshtml.HTMLDocumentClass"] ;Set correct assembly path
    Local $pObject = 0
    $oAssemblyType.InvokeMember_3("CreateComInstanceFrom", 0x158, 0, 0, CreateSafeArray($aText), $pObject)
    ConsoleWrite("IsObject: " & IsObj($pObject) & @TAB & "$pObject: " & ObjName($pObject) & @CRLF & @CRLF)

    Local $oHTMLDocumentClass = $pObject.Unwrap()
    ConsoleWrite("!$oHTMLDocumentClass: " & IsObj($oHTMLDocumentClass) & @CRLF & @CRLF)
    $oHTMLDocumentClass.write($sHTML)

    Local $oLinks=$oHTMLDocumentClass.Links
    For $oLink in $oLinks
        ConsoleWrite($oLink.ToString() & @CRLF)
    Next
```

.NET CLR Framework

HOW: to Constructing an Instance of a Type Using Reflection

- **CLR Late Binding**
  - The **Activator** class :

2. CreateInstanceFrom : Used to create a reference to an object from a particular assembly and type name.

The Activator class also offers a set of static **CreateInstanceFrom** methods.

These methods behave just as the **CreateInstance** method, except that you must always specify the type and its assembly via string parameters.

The assembly is loaded into the calling **AppDomain** by using Assembly's **LoadFrom** method (instead of Load).

Because none of these methods takes a Type parameter, all of the **CreateInstanceFrom** methods return a reference to an **ObjectHandle**, which must be **unwrapped**.

Example : see **1. CreateComInstanceFrom**

.NET CLR Framework

HOW: to Constructing an Instance of a Type Using Reflection

- **CLR Late Binding**
  - The **Activator** class :

3. GetObject : Used when marshaling objects.

4. CreateInstance : Used to create local or remote instances of an object.
When you call this method, you can pass either a reference to a Type object or a String that identifies the type of object you want to create. The versions that take a type are simpler. You get to pass a set of arguments for the type's constructor, and the method returns a reference to the new object.
The versions of this method in which you specify the desired type by using a string are a bit more complex.

An **ObjectHandle** is a type that allows an object created in one **AppDomain** to be passed around to other AppDomains without forcing the object to materialize. When you're ready to materialize the object, you call **ObjectHandle's Unwrap method**. This method loads the assembly that defines the type being materialized in the AppDomain where Unwrap is called.

Example : see below…

.NET CLR Framework

HOW: to Constructing an Instance of a Type Using Reflection

- **CLR Late Binding**

  - The **Activator** class :

    4. CreateInstance : Used to create local or remote instances of an object.

```
Example()

Func Example()
    Local $oAssembly = _CLR_LoadLibrary("mscorlib")
    ConsoleWrite("!$oAssembly: " & IsObj($oAssembly) & @CRLF)

    Local $pType
    $oAssembly.GetType_2("System.Activator",$pType)
    ConsoleWrite("!$pType: " & ptr($pType) & @CRLF)

    Local $oType = ObjCreateInterface($pType, $sIID_IType, $sTag_IType)
    ConsoleWrite("IsObj( $oType ) = " & IsObj($oType) & @CRLF)
    ConsoleWrite(@CRLF)

    Local $aText[] = ["mscorlib", "System.Collections.Stack"]
    Local $pObject = 0
    $oType.InvokeMember_3("CreateInstance", 0x158, 0, 0, CreateSafeArray($aText), $pObject)
    ConsoleWrite("IsObject: " & IsObj($pObject) & @TAB & "$pObject: " & $pObject & @CRLF)

    Local $oStack = $pObject.Unwrap()
    ConsoleWrite("!$oStack: " & IsObj($oStack) & @CRLF)
    $oStack.Push("Bye Bye...")
    $oStack.Push("I Love AutoIt...")
    $oStack.Push("AutoIt Rocks...")

    For $i = 0 To $oStack.Count() - 1
        MsgBox(0, "Stack Example", $oStack.Pop())
    Next

EndFunc   ;==>Example
```

.NET CLR Framework

HOW: to Constructing an Instance of a Type Using Reflection

- **CLR System.AppDomain Class**

  System.AppDomain's methods :

  The **AppDomain** type offers four instance methods (each with several overloads) that construct an instance of a type: **CreateInstance**, **CreateInstanceAndUnwrap**, **CreateInstanceFrom**, and **CreateInstanceFromAndUnwrap**.

  These methods work just as **Activator**'s methods except that these methods are **instance methods**, allowing you to specify which **AppDomain the object should be constructed in**.

  The **methods** that end with **Unwrap** exist for convenience so that you don't have to make an additional method call.

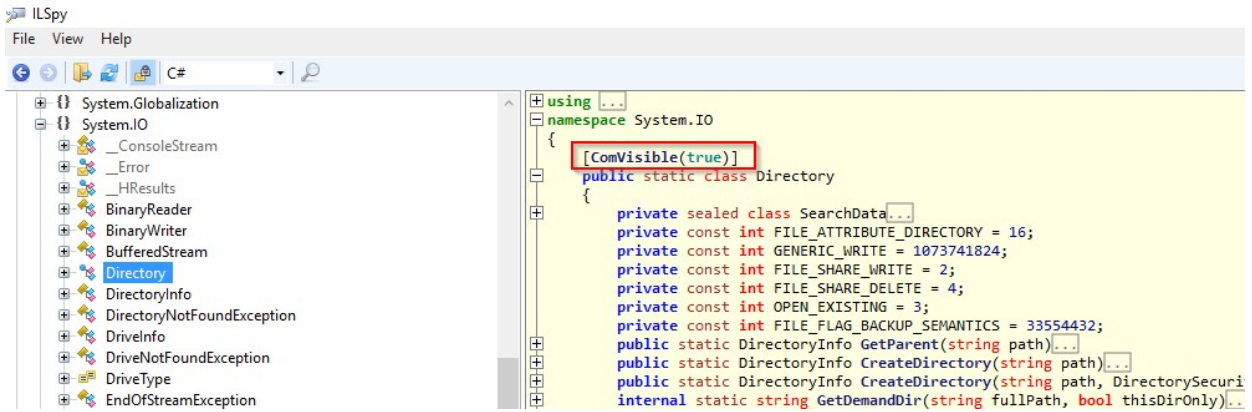
```
Example()

Func Example()
  Local $oAppDomain = _CLR_GetDefaultDomain()

  ; Create a 1D array with one integer element
  Local $aArgs = [ 128 ], $psa = CreateSafeArray( $aArgs )

  ; Create an instance of ArrayList using a parameterized constructor
  Local $pHandle
  $oAppDomain.CreateInstance 3( "mscorlib", "System.Collections.ArrayList", True, 0, 0, $psa, 0, 0, 0, $pHandle )
  ConsoleWrite( @CRLF & "$pHandle = " & $pHandle & @CRLF )

  Local $oHandle = ObjCreateInterface( $pHandle, $sIID_IObjectHandle, $sTag_IObjectHandle )
  ConsoleWrite( @CRLF & "IsObj( $oHandle ) = " & IsObj( $oHandle ) & @CRLF )

  ; Unwrap the ArrayList instance inside the ObjectHandle
  Local $oArrayList
  $oHandle.Unwrap( $oArrayList )
  ConsoleWrite( @CRLF & "IsObj( $oArrayList ) = " & IsObj( $oArrayList ) & @CRLF )

  ; Print ArrayList Capacity
;~   ConsoleWrite( @CRLF & "$oArrayList.Capacity() = " & $oArrayList.Capacity() & @CRLF )
  MsgBox(0,"""System.Collections.ArrayList","$oArrayList.Capacity() = " & $oArrayList.Capacity())
EndFunc
```

.NET CLR Framework

HOW: to access Assemblies using CLR Runtime Hosts

- **Different Types of .NET Objects exist : COM Visible** or **NON COM Visible** Objects !

So Depending if on this you need to use different CLR Functions to access the Type Members…
You can use one of the many Free Assembly Viewers, to check if a Method is **COM Visible or NOT.**
Like for Example **ILSpy**



In **AutoIT** you need to use these functions :

- For **COM** Visible : **_CLR_CreateObject()**
- For **NON COM** Visible : **_CLR_LoadLibrary()**

.NET CLR Framework

HOW: to access .NET Assemblies using CLR Runtime Hosts

- **There are big Difference between the Conventional COM and .NET CLR Components**

  One of them is that the .NET cannot be accessed the same way as the conventional COM Objects

  The ways COM and .NET locate components are quite different. **Conventional COM** components can be physically located anywhere, but the information about how to find and load them is kept in one central location: the **Registry**.

  In contrast, **CLR components do not use the registry at all**. All **managed assemblies** bring this information stored within them, as **metadata**. In addition, .NET components can live either **privately** with their applications in the same directory, or **globally shared in the Global Assembly Cache (GAC).**

  To instantiate a COM component with **CoCreateInstance**, COM looks in the registry for the CLSID key, and the values associated with it.

  These values tell **COM** the **name** and the **location** of the **DLL** or **EXE** that implements the **COM co-class** that you wish to load. One of the much-touted benefits of COM is location transparency.

.NET CLR Framework

HOW: to access .NET Assemblies using CLR Runtime Hosts

- **There are big Difference between the Conventional COM and .NET CLR Components**

  Simply stated, the COM client calls the object in the same way, whether the object is in-process with the client, out-of-process on the same local machine, or running on a different machine altogether; the registry tells COM where.

  This system is easy to break. If files change location without changing their registry setting, programs break completely. This contributes to the infamous problem known as "**DLL Hell."**

  For that reason, and many others, **.NET components** take a completely **different approach.**

  The **CLR** looks in one of three places: the **GAC**, the **local directory**, or some other place specified by a **configuration file**.

.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **COM Visible** Objects

In AutoIT you need to use : **_CLR_CreateObject()** function for accessing **COM Visible Members**

```
Example4() ; System.IO.FileInfo

Func Example4()

    Local $oAssembly = _CLR_LoadLibrary("mscorlib")
    ConsoleWrite("$oAssembly: " & IsObj($oAssembly) & @CRLF)

    local $objFileInfoClass = _CLR_CreateObject($oAssembly,"System.IO.FileInfo", @ScriptFullPath )
    ConsoleWrite("!$objFileInfoClass " & IsObj($objFileInfoClass) & @CRLF)

    MsgBox(0,".Net CLR System.IO.File #", "FileAttributes # : " & @ScriptFullPath & @CRLF)

    $Attributes = $objFileInfoClass.Attributes ;  Gets or sets the attributes for the current file or directory. (Inherited from FileSystemInfo.)
        ConsoleWrite("!$Attributes: " & $Attributes & @CRLF)

    $CreationTime = $objFileInfoClass.CreationTime ;  Gets or sets the creation time of the current file or directory. (Inherited from FileSystemInfo.)
        ConsoleWrite("!$CreationTime: " & DateStringToDateTime_EU($CreationTime) & @CRLF)


    $objDirectory = $objFileInfoClass.Directory ;  Gets an instance of the parent directory.
        ConsoleWrite("!$objDirectory: " & $objDirectory & @CRLF)

    $DirectoryName = $objFileInfoClass.DirectoryName ;  Gets a string representing the directory's full path.
        ConsoleWrite("!$DirectoryName: " & $DirectoryName & @CRLF)

    $Exists = $objFileInfoClass.Exists ;  Gets a value indicating whether a file exists. (Overrides FileSystemInfo.Exists.)
        ConsoleWrite("!$Exists: " & $Exists & @CRLF)
```

As You can see the **COM VISIBLE** Methods / Properties are accessible using the **DOT notation**.

.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **COM Visible** Objects

  For **COM Visible Objects** you can easily use **PARAMETERS** in the **_CLR_CreateObject()** Function.

```
Func Example_FileInfo()
  Local $oAssembly = _CLR_LoadLibrary( "mscorlib" )
  Local $oFileInfo = _CLR_CreateObject( $oAssembly, "System.IO.FileInfo", @ScriptDir & "\tst02.txt" )
  ConsoleWrite( "IsObj( $oFileInfo ) = " & IsObj( $oFileInfo ) & @CRLF & @CRLF)
  $oFileInfo.Create()
;~   ConsoleWrite( $oFileInfo.Name & @CRLF)
  MsgBox(0,"System.IO.FileInfo","File Created :  " & $oFileInfo.Name )
EndFunc
```

.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **NON COM Visible** Objects

In AutoIT you need to use : **_CLR_LoadLibrary()** function for accessing **NON COM Visible Members**

```
Example4() ; System.IO.FileInfo

Func Example4()

    Local $oAssembly = _CLR_LoadLibrary("mscorlib")
    ConsoleWrite("$oAssembly: " & IsObj($oAssembly) & @CRLF)

    local $objFileInfoClass = _CLR_CreateObject($oAssembly,"System.IO.FileInfo", @ScriptFullPath )
    ConsoleWrite("!$objFileInfoClass " & IsObj($objFileInfoClass) & @CRLF)

    MsgBox(0,".Net CLR System.IO.File #", "FileAttributes # : " & @ScriptFullPath & @CRLF)

    $Attributes = $objFileInfoClass.Attributes ;  Gets or sets the attributes for the current file or directory. (Inherited from FileSystemInfo.)
        ConsoleWrite("!$Attributes: " & $Attributes & @CRLF)

    $CreationTime = $objFileInfoClass.CreationTime ;  Gets or sets the creation time of the current file or directory. (Inherited from FileSystemInfo.)
        ConsoleWrite("!$CreationTime: " & DateStringToDateTime_EU($CreationTime) & @CRLF)


    $objDirectory = $objFileInfoClass.Directory ;  Gets an instance of the parent directory.
        ConsoleWrite("!$objDirectory: " & $objDirectory & @CRLF)

    $DirectoryName = $objFileInfoClass.DirectoryName ;  Gets a string representing the directory's full path.
        ConsoleWrite("!$DirectoryName: " & $DirectoryName & @CRLF)

    $Exists = $objFileInfoClass.Exists ;  Gets a value indicating whether a file exists. (Overrides FileSystemInfo.Exists.)
        ConsoleWrite("!$Exists: " & $Exists & @CRLF)
```

It is important to set the correct **BindingFlags** in order to find the **Members** you need.

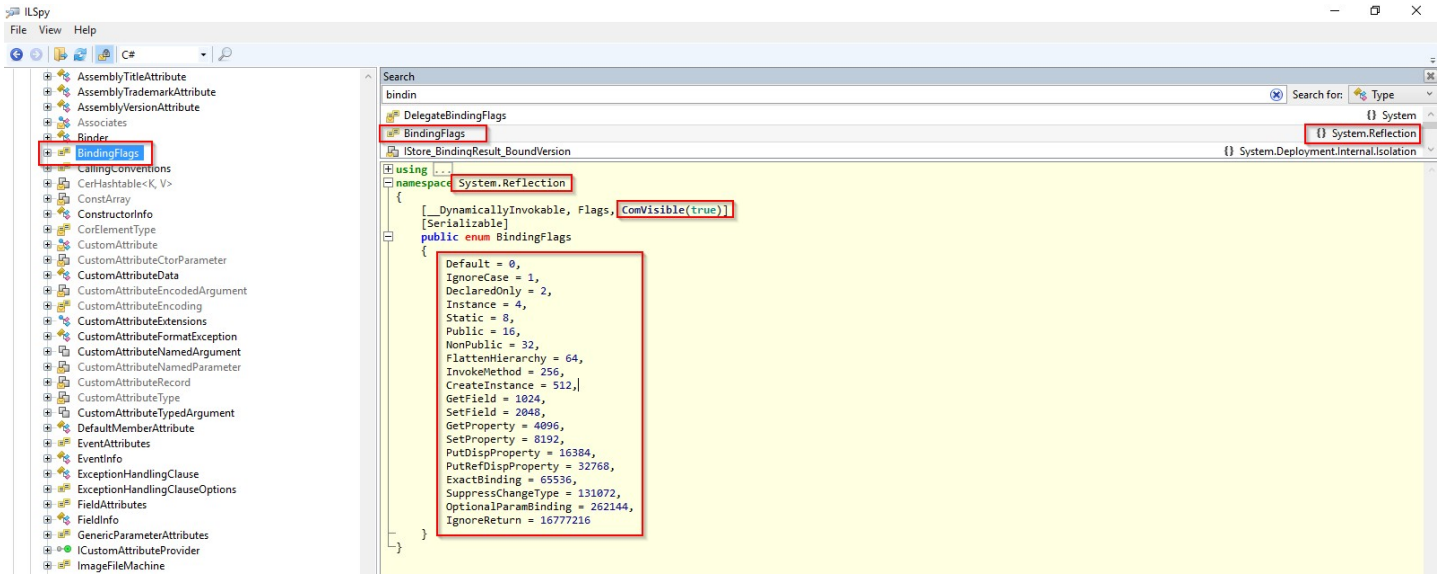
.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **BindingFlags Options in .NET**

  Specifies flags that control binding, and the way in which the **Search for Members and Types** is conducted by **Reflection**.

  See **System.Reflection BindingFlag Options**



More info see : https://msdn.microsoft.com/en-us/library/system.reflection.bindingflags(v=vs.110).aspx

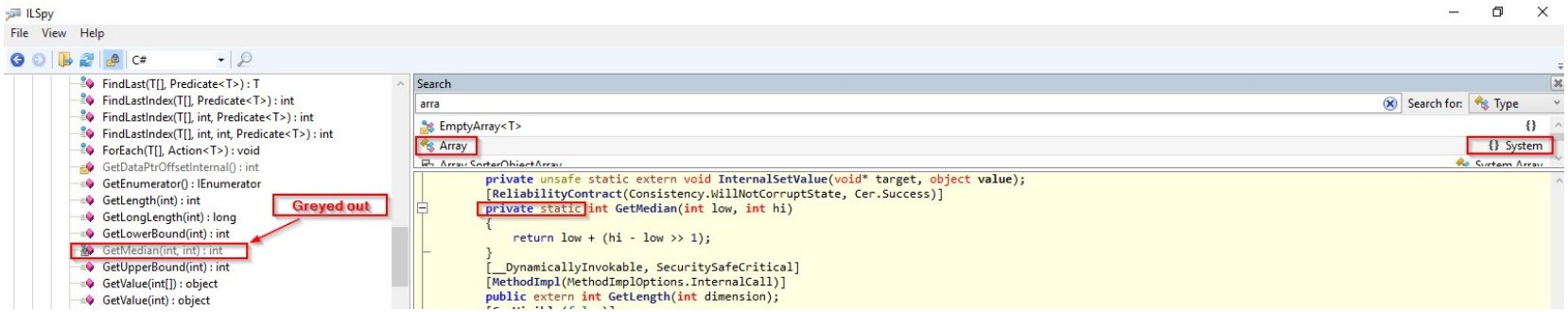
.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **BindingFlags Options in .NET**

A **Property** is considered **Public** to **Reflection** if it has at least one **Accessor** that is **Public**. Otherwise the **Property** is considered **Private**, and you must use BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static (Combine the values using **Or**) to get it.

Example : **Array Class** GetMedian **Private Type**



A **Private** Type is **unaccessible** by **default**, but you can get access to it using the correct **BindingFlag combinations**.

.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **BindingFlags Options in AutoIT**

  Specifies flags that control binding, and the way in which the **Search for Members and Types** is conducted by **Reflection**.

  See **CLR.au3 UDF** for all **BindingFlag Options**

```
; ******************************************************************************
; Declare Binding Flags : InvokeMember, InvokeMember_2, InvokeMember_3
; ******************************************************************************
; 0x158 = $BindingFlags_Static + $BindingFlags_Public + $BindingFlags_FlattenHierarchy + $BindingFlags_InvokeMethod
Global Const $BindingFlags_Default = 0x0000
Global Const $BindingFlags_IgnoreCase = 0x0001
Global Const $BindingFlags_DeclaredOnly = 0x0002
Global Const $BindingFlags_Instance = 0x0004
Global Const $BindingFlags_Static = 0x0008
Global Const $BindingFlags_Public = 0x0010
Global Const $BindingFlags_NonPublic = 0x0020
Global Const $BindingFlags_FlattenHierarchy = 0x0040
Global Const $BindingFlags_LookupAll = 0x0042 ;BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Static
Global Const $BindingFlags_InvokeMethod = 0x0100
Global Const $BindingFlags_CreateInstance = 0x0200
Global Const $BindingFlags_GetField = 0x0400
Global Const $BindingFlags_SetField = 0x0800
Global Const $BindingFlags_GetProperty = 0x1000
Global Const $BindingFlags_SetProperty = 0x2000
Global Const $BindingFlags_PutDispProperty = 0x4000
Global Const $BindingFlags_PutRefDispProperty = 0x8000
Global Const $BindingFlags_ExactBinding = 0x00010000
Global Const $BindingFlags_SuppressChangeType = 0x00020000
Global Const $BindingFlags_OptionalParamBinding = 0x00040000
Global Const $BindingFlags_IgnoreReturn = 0x01000000
```

.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **BindingFlags**
  - **BindingFlags** Example :

    Without the correct **BindingFlags** you will **not get any RESULT** !!

```
local $oConsole=getConsole()
local $propValue=0
Local $sText[] = [@CRLF & "AutoIT Rocks !!" & @CRLF & @CRLF]
$oConsole.InvokeMember_3("Write", 0x158, 0, 0, CreateSafeArray($sText), $propValue)

$oConsole.InvokeMember_3("IsOutputRedirected", $BindingFlags_GetProperty, 0, 0, 0, $propValue)
consolewrite("IsOutputRedirected : " & $propValue & @CRLF)

$oConsole.InvokeMember_3("Title", $BindingFlags_GetProperty, 0, 0, 0, $propValue)
consolewrite("Title " & $propValue & @CRLF)

$oConsole.InvokeMember_3("WindowHeight", $BindingFlags_GetProperty, 0, 0, 0, $propValue)
consolewrite("WindowHeight " & $propValue & @CRLF)

$oConsole.InvokeMember_3("Clear", 0x158, 0, 0, 0, 0)
```

Can be a HexValue = 0x158 or the Variable Name $BindingFlags_GetProperty as Function Parameter

More info here : https://msdn.microsoft.com/en-us/library/kyaxdd3x.aspx

.NET CLR Framework

HOW: would you use CLR Runtime Hosts in AutoIT

- **Using SafeArrays**

In AutoIT you always have to use **SafeArrays** in the CLR Functions to pass as a parameter.

Standard AutoIT Arrays will not work !

```
$sTag_Activator = $stagIUnknown & $sTag_IDispatch
ConsoleWrite($sTag_Activator & @CRLF)


Local $oAssembly = _CLR_LoadLibrary("mscorlib")
ConsoleWrite("$oAssembly: " & IsObj($oAssembly) & @CRLF)

Local $pAssemblyType = 0
$oAssembly.GetType_2("System.Activator", $pAssemblyType)
ConsoleWrite("$pAssemblyType = " & Ptr($pAssemblyType) & @CRLF)

Local $oAssemblyType = ObjCreateInterface($pAssemblyType, $sIID_IType, $sTag_IType)
ConsoleWrite("IsObj( $oAssemblyType ) = " & IsObj($oAssemblyType) & @TAB & @CRLF)

Local $aText[] = ["mscorlib", "System.Collections.Stack"]
Local $pObject = 0
$oAssemblyType.InvokeMember_3("CreateInstance", 0x158, 0, 0, CreateSafeArray($aText), $pObject)
ConsoleWrite("IsObject: " & IsObj($pObject) & @TAB & "$pObject: " & $pObject & @CRLF)

Local $oStack = $pObject.Unwrap()
ConsoleWrite("!$oStack: " & IsObj($oStack) & @CRLF)
$oStack.Push("Bye Bye...")
$oStack.Push("I Love AutoIt...")
$oStack.Push("AutoIt Rocks...")

For $i = 0 To $oStack.Count() - 1
    MsgBox(0, "Stack Example", $oStack.Pop())
Next
```

See **SAFEARRAY.au3 UDF** for all **Options**

.NET CLR Framework

HOW: would you Debug CLR Runtime Hosts in AutoIT

- **DEBUGGING**

  Interpreting HRESULTS

  In **AutoIT** you occasionally might run into mysterious HRESULTs returned from .NET that begins with 0x8013

  Interpreting HRESULTS returned from .NET/CLR: 0x8013XXXX COM Error,

  see here :
  https://blogs.msdn.microsoft.com/yizhang/2010/12/17/interpreting-hresults-returned-from-netclr-0x8013xxxx/

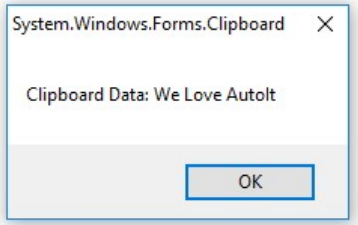
.NET CLR Framework

WHEN : would you use CLR Runtime Hosts

- **WHEN : Would you use CLR Runtime Hosts :**

1. To access .NET Class Libraries

```
Func Example()
    Local $oAssembly = _CLR_LoadLibrary("System.Windows.Forms")
    ConsoleWrite("$oAssembly: " & IsObj($oAssembly) & @CRLF)

    Local $pAssemblyType = 0
    $oAssembly.GetType_2("System.Windows.Forms.Clipboard", $pAssemblyType)
    ConsoleWrite("$pAssemblyType = " & Ptr($pAssemblyType) & @CRLF)

    Local $oAssemblyType = ObjCreateInterface($pAssemblyType, $sIID_IType, $sTag_IType)
    ConsoleWrite("IsObj( $oAssemblyType ) = " & IsObj($oAssemblyType) & @CRLF)

    Local $aText[] = ["We Love AutoIt"]
    Local $sClipBoardText = ""

    $oAssemblyType.InvokeMember_3("Clear", 0x158, 0, 0, 0, 0)

    $oAssemblyType.InvokeMember_3("SetText", 0x158, 0, 0, CreateSafeArray($aText), 0)
;~     ConsoleWrite("Clipboard Data: " & ClipGet() & @CRLF)
    MsgBox(0,"System.Windows.Forms.Clipboard" , "Clipboard Data: " & ClipGet())
    ClipPut("AutoIt Rocks!!!")

    $oAssemblyType.InvokeMember_3("GetText", 0x158, 0, 0, 0, $sClipBoardText)
;~     ConsoleWrite("Clipboard Data: " & $sClipBoardText & @CRLF)
    MsgBox(0,"System.Windows.Forms.Clipboard" , "New Clipboard Data: " & $sClipBoardText)

EndFunc   ;==>Example
```



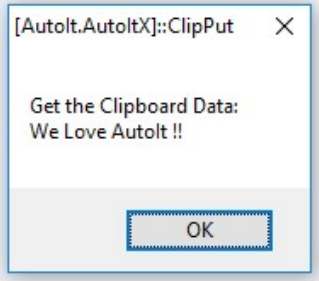
.NET CLR Framework

WHEN : would you use CLR Runtime Hosts

- **WHEN : Would you use CLR Runtime Hosts :**

  2. Accessing custom build . NET Assemblies : **AutoItx3.Assembly.dll**

```
#include ".\Includes\CLR.au3"

Example()

Func Example()
    Local $oAssembly = _CLR_LoadLibrary("C:\Program Files (x86)\AutoIt3\Beta\AutoItX\AutoItX3.Assembly.dll")
    ConsoleWrite("$oAssembly: " & IsObj($oAssembly) & @CRLF)

    Local $pAssemblyType = 0
    $oAssembly.GetType_2("AutoIt.AutoItX", $pAssemblyType)
    ConsoleWrite("$pAssemblyType = " & Ptr($pAssemblyType) & @CRLF)

    Local $oAssemblyType = ObjCreateInterface($pAssemblyType, $sIID_IType, $sTag_IType)
    ConsoleWrite("IsObj( $oAssemblyType ) = " & IsObj($oAssemblyType) & @CRLF)

    Local $aText[] = ["We Love AutoIt !!"]
    Local $sClipBoardText = ""

    $oAssemblyType.InvokeMember_3("", 0x158, 0, 0, 0, 0)

    $oAssemblyType.InvokeMember_3("ClipPut", 0x158, 0, 0, CreateSafeArray($aText), 0)

;~  ConsoleWrite("Clipboard Data: " & ClipGet() & @CRLF)
    MsgBox(0,"[AutoIt.AutoItX]::ClipPut" , "Get the Clipboard Data: " & @CRLF & ClipGet())

EndFunc   ;==>Example
```



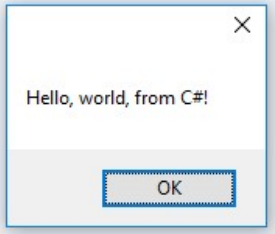
.NET CLR Framework

WHEN : would you use CLR Runtime Hosts

- **WHEN : Would you use CLR Runtime Hosts :**

3. To Compile . NET Code into an Assembly : **Compile C# or VB.Net code into Assembly.dll**

```
#include ".\Includes\CLR.au3"

_Example()

Func _Example()

    Local $oAssemblyCSharp = _CLR_CompileCSharp(FileRead("CodeC#.cs"),"System.dll | System.Windows.Forms.dll")
    If IsObj($oAssemblyCSharp ) Then
    ConsoleWrite("$oAssembly: " & IsObj($oAssemblyCSharp) & @CRLF)
    Local $oFoo = _CLR_CreateObject($oAssemblyCSharp, "Foo")
    ConsoleWrite("$oObject: " & IsObj($oFoo) & @CRLF)
    If IsObj($oFoo) Then $oFoo.Test()
    EndIf

    Local $oAssemblyVB = _CLR_CompileVB(FileRead("CodeVB.cs"),"System.dll | System.Windows.Forms.dll")
    If IsObj($oAssemblyVB ) Then
    ConsoleWrite("$oAssembly: " & IsObj($oAssemblyVB) & @CRLF)
    Local $oFoo = _CLR_CreateObject($oAssemblyVB, "Foo")
    ConsoleWrite("$oObject: " & IsObj($oFoo) & @CRLF)
    If IsObj($oFoo) Then $oFoo.Test()
    EndIf

EndFunc   ;==>_Example
```



.NET CLR Framework

WHEN : would you use CLR Runtime Hosts

- **WHEN : Would you use CLR Runtime Hosts :**

4. To Run C# or VB.net Code : Compile **Code C#** at **Runtime**

```
#Region Load Dynamic Code
      Local $sCode = FileRead( "Code.cs" )
      ConsoleWrite( "$sCode = " & @CRLF & $sCode )
#EndRegion Load Dynamic Code

      ; Compile!
      ; compilerRes := codeCompiler.CompileAssemblyFromSource(prms, Code)
      Local $oCompilerRes = $oCodeCompiler.CompileAssemblyFromSource( $oPrms,
      ConsoleWrite( "IsObj( $oCompilerRes ) = " & IsObj( $oCompilerRes ) & @CR

      Local $pCodeAssembly = $oCompilerRes.CompiledAssembly()
      ConsoleWrite( "$pCodeAssembly = " & Ptr( $pCodeAssembly ) & @CRLF )

      Local $oCodeAssembly = ObjCreateInterface( $pCodeAssembly, $sIID_IAssemb
      ConsoleWrite( "IsObj( $oCodeAssembly ) = " & IsObj( $oCodeAssembly ) & @

      Local $oCodeObject
      $oCodeAssembly.CreateInstance_2( "Foo", True, $oCodeObject )
      ConsoleWrite( "IsObj( $oCodeObject ) = " & IsObj( $oCodeObject ) & @CRLF

      Local $sKram = $oCodeObject.Test( "Win32_Processor" )
      $sKram = StringReplace( $sKram, "/n", @CRLF )
      MsgBox( 0, "", $sKram )
    EndIf

  EndIf

  DllClose($hMSCorEE)
EndFunc
```

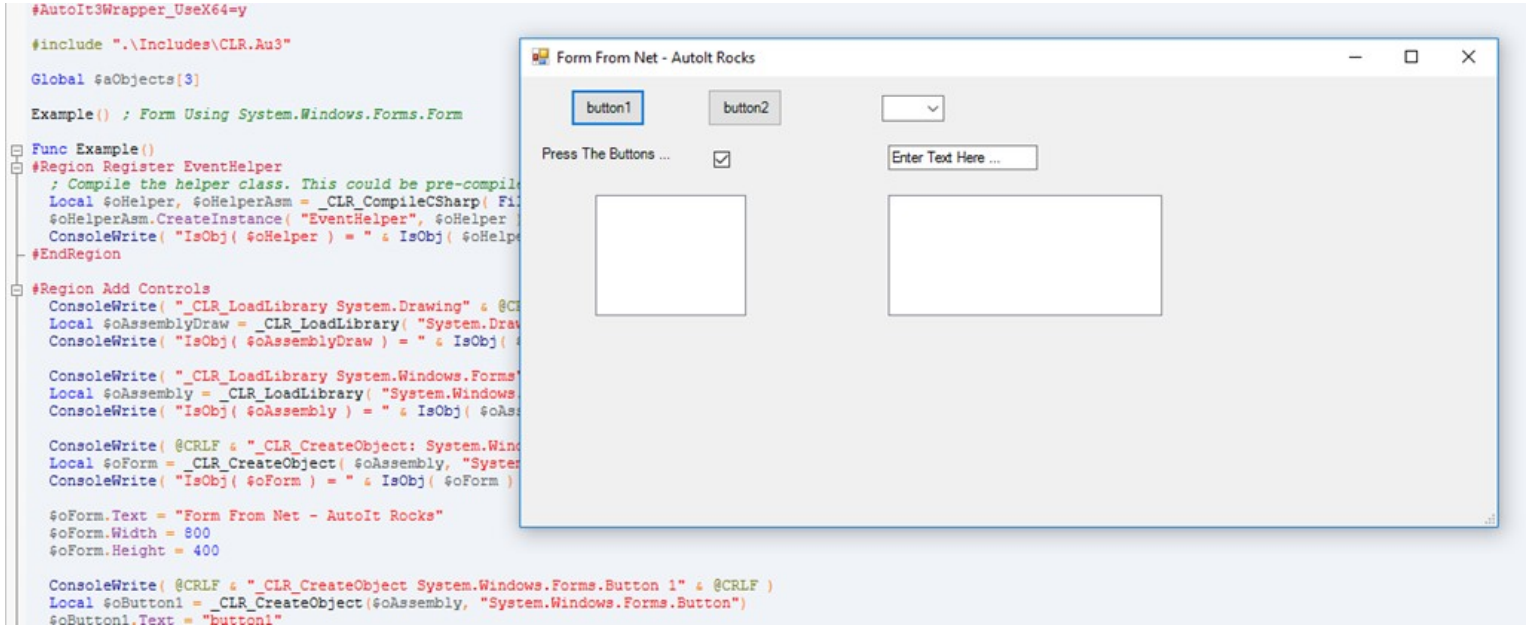
```
AssetTag: To Be Filled By O.E.M.
Availability: 3
Caption: Intel64 Family 6 Model 78 Stepping 3
Characteristics: 252
ConfigManagerErrorCode:
ConfigManagerUserConfig:
CpuStatus: 1
CreationClassName: Win32_Processor
CurrentClockSpeed: 2300
CurrentVoltage: 7
DataWidth: 64
Description: Intel64 Family 6 Model 78 Stepping 3
DeviceID: CPU0
ErrorCleared:
ErrorDescription:
ExtClock: 100
Family: 205
InstallDate:
L2CacheSize: 512
L2CacheSpeed:
L3CacheSize: 3072
L3CacheSpeed: 0
LastErrorCode:
Level: 6
LoadPercentage: 8
Manufacturer: GenuineIntel
MaxClockSpeed: 2400
```

.NET CLR Framework

WHEN : would you use CLR Runtime Hosts

- **WHEN : Would you use CLR Runtime Hosts :**

5. To create .Net GUI WPF Applications



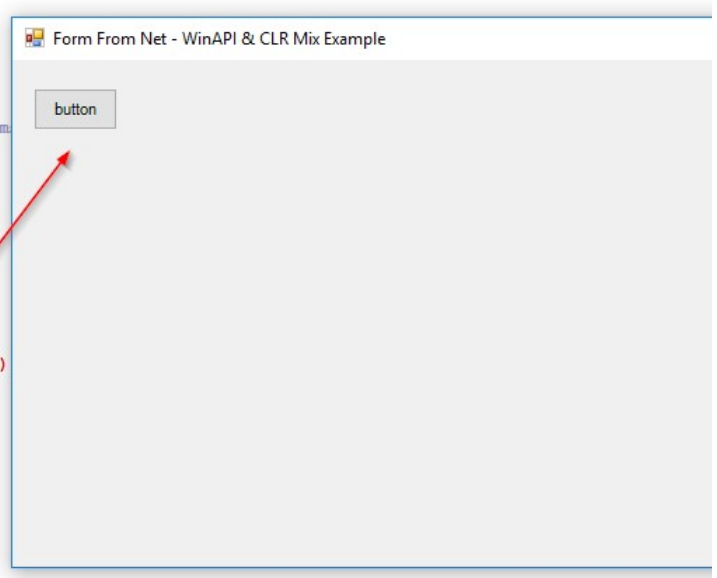
.NET CLR Framework

WHEN : would you use CLR Runtime Hosts

- **WHEN : Would you use CLR Runtime Hosts :**

6. To Mix AU3 WinAPI's and .Net in your Application

```
    Local $oForm = _CLR_CreateObject($oAssembly, "System.Windows.Forms.Form")
    ConsoleWrite("!$oForm: " & IsObj($oForm) & @CRLF)
    ConsoleWrite("$oForm Handle: " & $oForm.Handle & @CRLF)

    $oForm.Text = "Form From Net - WinAPI & CLR Mix Example"
    $oForm.Width = 800
    $oForm.Height = 400

    Local $oButton1 = _CLR_CreateObject($oAssembly, "System.Windows.Form
    ConsoleWrite("!$oButton1: " & IsObj($oButton1) & @CRLF)
    ConsoleWrite($oButton1.handle & @CRLF)

    $oButton1.Text = "button"
    $oButton1.Left = 15
    $oButton1.Top = 20
    $oButton1.Width = 60
    $oButton1.Height = 30

    ; DOES NOT WORK YET ...
    Local $oDrawing = _CLR_LoadLibrary("System.Drawing")
    Local $oDraw = _CLR_CreateObject($oAssembly, "System.Drawing.point")
    ConsoleWrite("!$oDraw: " & Isobj($oDraw) & @CRLF)

;~  $oForm.Controls.Add($oButton1)
    _WinAPI_SetParent($oButton1.handle, $oForm.Handle)

    $oForm.ShowDialog()

    $oForm.Dispose()
EndFunc   ;==>_ExampleTest
```



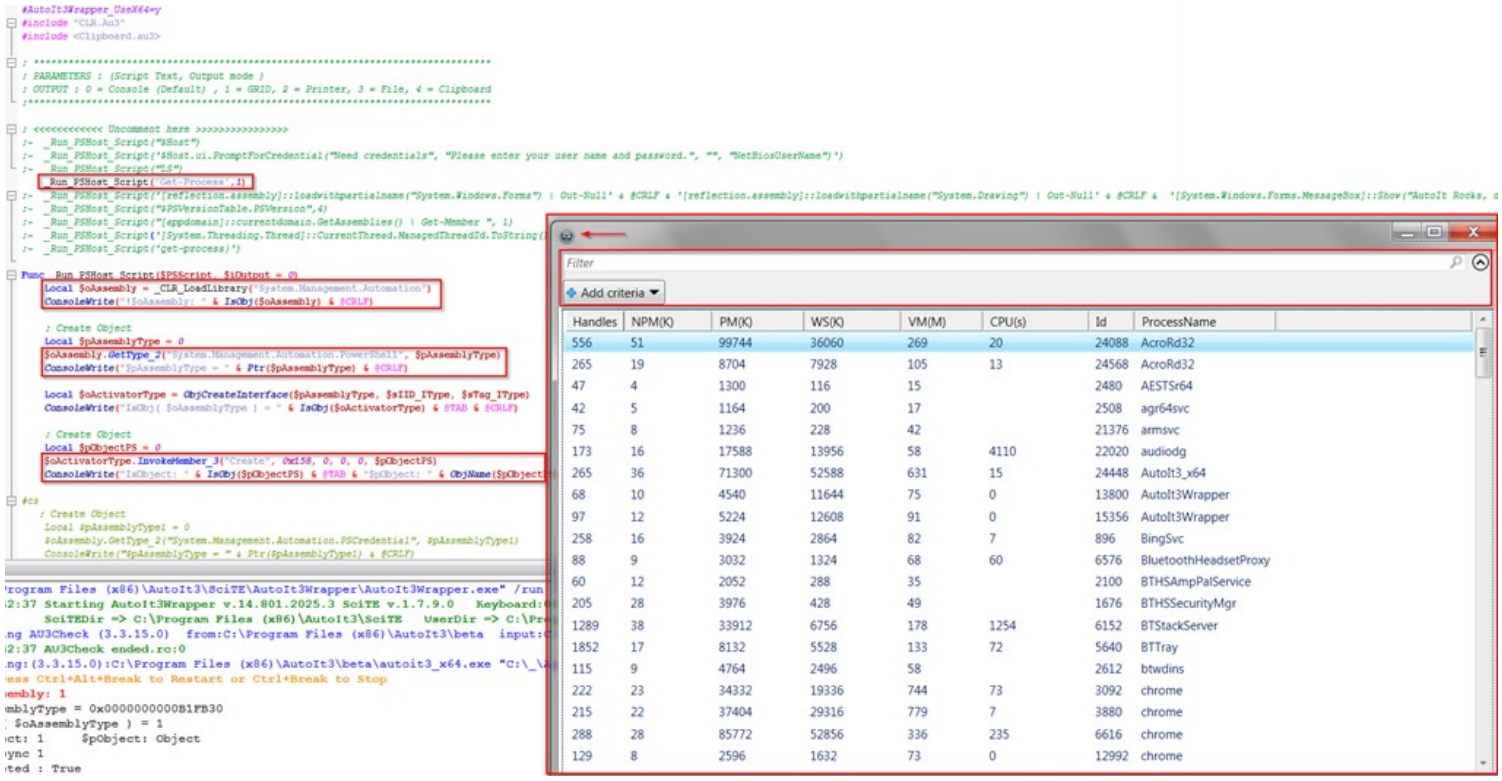
.NET CLR Framework

HOW: would you access PowerShell using CLR Runtime Hosts in AutoIT

- **POWERSHELL Automation** :

In **AutoIT** you can now access **PowerShell Modules**, **Cmdlet**, **Scripts** and more …

By Accessing the "**System.Management.Automation.PowerShell**" Class

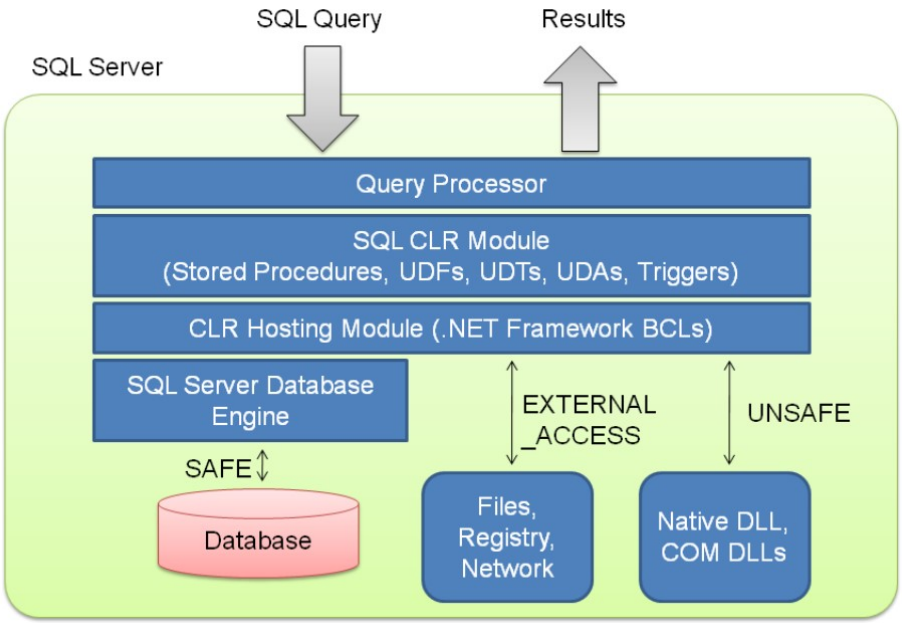


.NET CLR Framework

HOW: would you use SQL CLR Runtime Hosts in AutoIT

- **SQL Common Language Runtime**

SQL Common Language Runtime, is technology for hosting of the **Microsoft .NET common language Runtime Engine within SQL Server** ☺, which is unexplored at the moment …



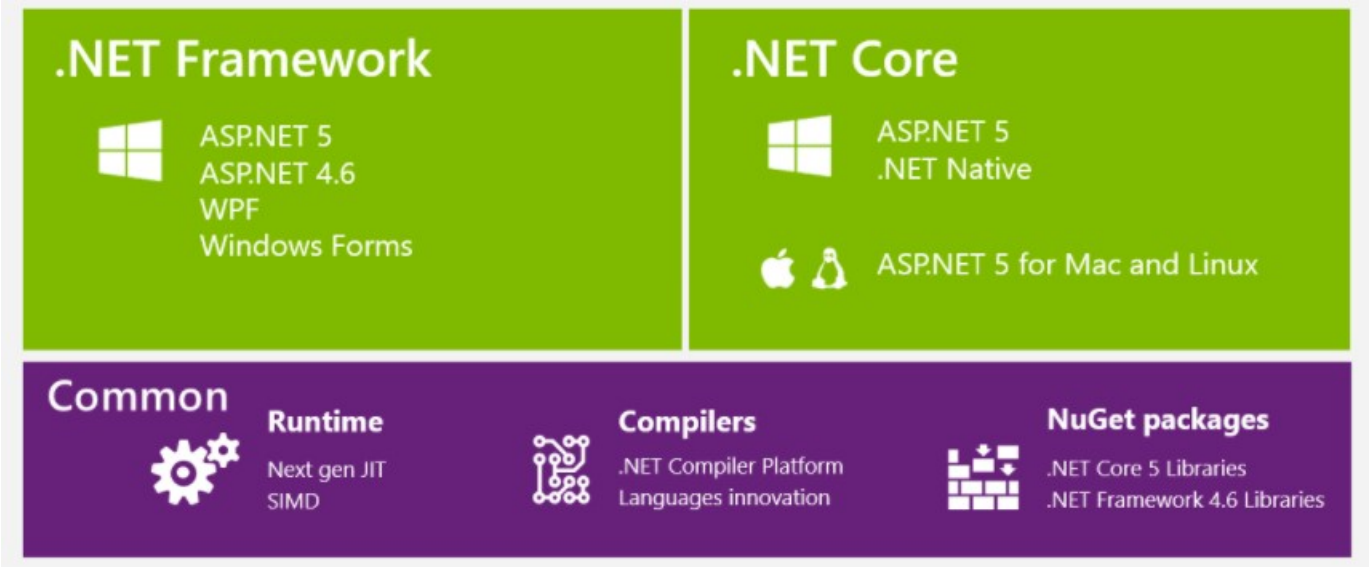
More info : http://www.sqlservercentral.com/articles/Stairway+Series/104406/

.NET CLR Framework

HOW: would you use SQL CLR Runtime Hosts in AutoIT

- **.NET CORE**

  .NET Core has **Portable Class Libraries** and is **Cross Platform**, supported by Microsoft on **Windows**, **Linux** and **Mac OSX** ☺, which we have not explored at the moment …



  In theory we could access .NET Core Libraries that are loaded on Linux / MAC and invoke commands … ?

  More info : https://blogs.msdn.microsoft.com/dotnet/2015/02/03/coreclr-is-now-open-source/

.NET CLR Framework

HOW: being grateful…

- **By Joining the CLR .NET development community and move this forward…**

  https://www.autoitscript.com/forum/topic/187334-using-net-libary-with-autoit-possible/

  https://www.autoitscript.com/forum/topic/188158-net-common-language-runtime-clr-framework/

  Many Thanks to :

  Danyfirex / Larsj / Junkew / Trancexx