Introduction to .NET cracking

Reflector

WDSM32 was one of the most popular tools for cracking in the past, after you decide to crack something you disassembled its code in WDSM32 to search for strings like "Registration successful" or "Invalid serial number !" and then you try to nop or inverse the right bytes to get the job done..

Unfortunately this tool is now history, since all the .net programs are interpreted and not compiled, you can no longer use the old techniques to or tools to crack something written with this new technology.

All .net applications require the .net runtime [**.NET Framework**] to be installed on your machine in order to be able to run them, unlike for instance Delphi executables which you can run without any runtime needed to be installed on your pc, the .net runtime works like java virtual machine which is needed to run java executables on your PC.

Is this good or bad?

This is a good question, as we all know that native code executables run really much faster than interpreted executables, this is a very notable fact when you start a do-nothing .net executable you will see that It takes a little bit more time to show the main form compared to a C++ or Delphi executables which will show the main form faster.

If speed was the bad point about .net programs so what is good about them when it comes to cracking?

The good point is that a typical .net program source code is compiled into something called **IL-code**, which stands for "Intermediate Language", something like Java's byte codes and later in run-time the IL-code is compiled into native code by the [**JIT compiler**], this **J**ust **I**n **T**ime compiler turns the IL bytes into native code that can be run by your CPU, what is important here is a simple fact that IL-code

is a higher level code than machine code that we usually dealt with when we targeted native code programs, meaning ?

This means that you can understand the IL instructions and analyze it more easily than Assembly language that WDSM32 provided us with when we used to disassemble native code executables.

Compiler		JIT Compiler		
.NET Source Code	**➔**	**IL Bytes**	**➔**	**Native code**

When we crack a .net executable we will be targeting it in the IL bytes level before it's compiled to native code.

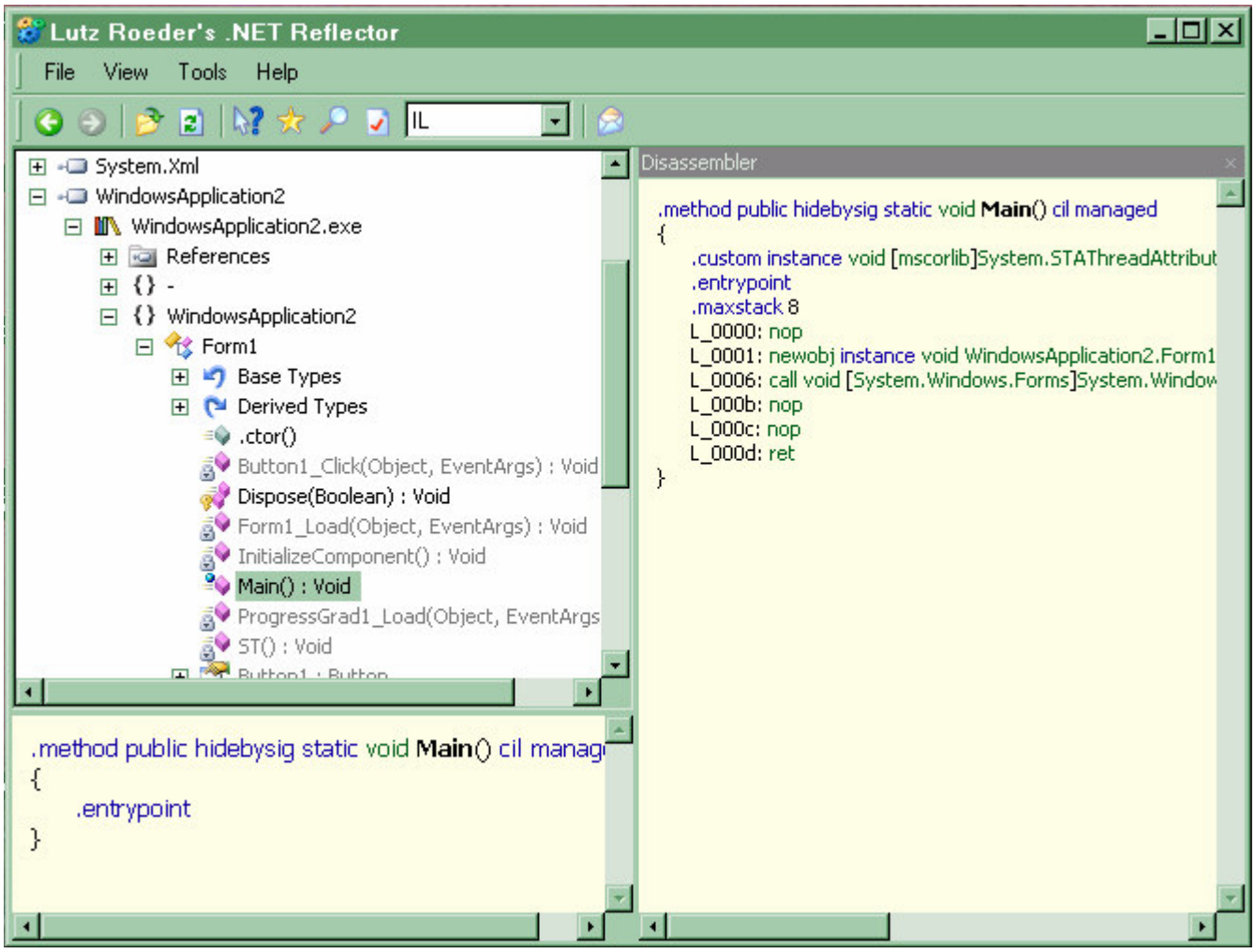
Now we will talk about the tool that should replace WDSM32, Actually there are many tools today and the most popular one called **"Reflector"**, This utility is the best available tool that can help us disassemble .net applications and understand what's going in the code, What makes this utility useful is that it can disassemble the application and show us the IL instructions of the code which is more than enough to do a crack, and it can also decompile the IL instructions into other higher level languages like VB.net or C# or Delphi .net !

This means that you can see the IL instructions or the **source code** itself with this tool, but when it comes to cracking .net programs you must basically be dependent upon the IL instructions and not on the higher level decompilation of the IL instructions, in most cases **"Reflector"** will be able to disassemble the program and show you the IL instructions, but it will fail in many cases to decompile the IL instructions into higher level languages and this is simply because the commercial applications industry started to understand that **"Reflector"** is simply the new WDSM32 for crackers and so they developed many tricks to confuse it and make it unable to decompile the IL instructions into your favorite language like C# or VB.net.

As a good cracker you must expect that you will be depending on IL instructions disassembly to analyze the code and crack the program you are targeting.

Another tool to mentions here is **"MSIL Disassembler"** which is developed by Microsoft and installed with Visual Studio as a part of the SDK tools, it's an excellent tool to disassemble the .net application and you will be using it side by side with **"Reflector"** because it does something that **"Reflector"** can't do.

"MSIL Disassembler" can disassemble the .net executable just like **reflector** and it also shows you the "Actual bytes" of these instructions.

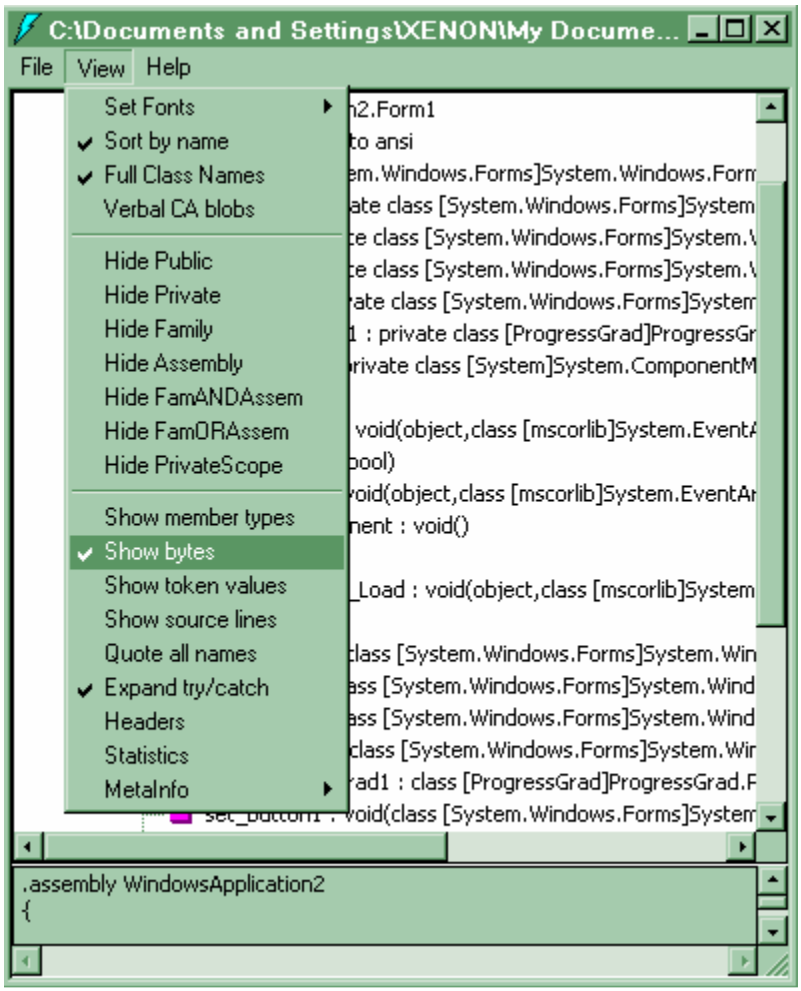


What you see here is **reflector** and on the right the disassembly of some procedure called Main():Void, You can see the IL instructions in the disassembly here:

```
Nop
Ret
```

```
Call
```

Down here you will see **"MSIL Disassembler"** which can show us the actual bytes of this procedure after we click the "Show bytes" menu.



Now I will list the disassembly for the same procedure from **"MSIL Disassembler"**

```
      .method public hidebysig static void  Main() cil managed
// SIG: 00 00 01
{
  .entrypoint
```

```
  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() =
( 01 00 00 00 )
  // Method begins at RVA 0x3a14
  // Code size       14 (0xe)
  .maxstack  8
  IL_0000:  /* 00    |                  */ nop
  IL_0001:  /* 73    | (06)000002       */ newobj     instance void
WindowsApplication2.Form1::.ctor()
  IL_0006:  /* 28    | (0A)000001       */ call       void
[System.Windows.Forms]System.Windows.Forms.Application::Run(class
[System.Windows.Forms]System.Windows.Forms.Form)
  IL_000b:  /* 00    |                  */ nop
  IL_000c:  /* 00    |                  */ nop
  IL_000d:  /* 2A    |                  */ ret
} // end of method Form1::Main
```

How to read this?

The IL instructions will begin right after the "maxstack #" line, the first line is IL_0000 which contains the instructions "NOP" and its actual bytes are "00" and that's what we will find if we open the program's file in a hex editor.

IL Line	Actual bytes	Instructions
IL_0000	00 1 byte	nop
IL_0001	7302000006 5 bytes	newobj
IL_0006	280100000A 5 bytes	all
IL_000b	00	nop
IL_000c	00	nop
IL_000d	2A	ret

If you open the program in a hex editor then you will find a series of these bytes from first line to last line and that's what I mean by "Actual bytes" of instructions.

The advantage of these actual bytes is that it tells you what to replace them with when you want to invert some jump or modify any part of the code, to nop a call in the past

times we used to do that by replacing all its actual bytes of that call by 90s in a hex editor, inverting a jump was done by replacing the JNE instruction byte with JE instruction byte. 75 to 74 or 85 to 84 and so on.

Another advantage is that it helps you locate the actual offset for these bytes in a hex editor, for example if you decide to nop the call in line IL_0006 then you will have to locate the file offset for these bytes, All you have to do is to open the file in a hex editor and locate the series of bytes you are looking for which are

Values **00** **7302000006** **280100000A**
Bytes **1** **5** **5**

And usually about 8 bytes are enough to locate these bytes in your hex editor, after you find the right offset you will have to replace **280100000A** with **0000000000**

Every IL instruction has a specific actual byte and they are very important in cracking too, for example in native code the nop instruction is expresses by one byte of value **0x90** but in .net Assembly the nop instruction is **0x00** byte, I will list the IL instructions here with their function and the actual bytes too.

The important Instructions are marked in gray.

Actual bytes	What It does	IL instruction
58	Adds two values and pushes the result onto the evaluation stack.	Add
D6	Adds two integers, performs an overflow check, and pushes the result onto the evaluation stack.	Add_Ovf
D7	Adds two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.	Add_Ovf_Un
5F	**Computes the bitwise AND of two values and pushes the result onto the evaluation stack.**	**And**
FE 00	Returns an unmanaged pointer to the argument list of the current method.	Arglist
3B	**Transfers control to a target instruction if two values are equal.**	**Beq**

2E	Transfers control to a target instruction (short form) if two values are equal.	Beq_S
3C	Transfers control to a target instruction if the first value is greater than or equal to the second value.	Bge
2F	Transfers control to a target instruction (short form) if the first value is greater than or equal to the second value.	Bge_S
41	Transfers control to a target instruction if the the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	Bge_Un
34	Transfers control to a target instruction (short form) if if the the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	Bge_Un_S
3D	Transfers control to a target instruction if the first value is greater than the second value.	Bgt
30	Transfers control to a target instruction (short form) if the first value is greater than the second value.	Bgt_S
42	Transfers control to a target instruction if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	Bgt_Un
35	Transfers control to a target instruction (short form) if the first value is greater than the second value, when comparing unsigned integer values or unordered float values.	Bgt_Un_S
3E	Transfers control to a target instruction if the first value is less than or equal to the second value.	Ble
31	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value.	Ble_S
43	Transfers control to a target instruction if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	Ble_Un
36	Transfers control to a target instruction (short form) if the first value is less than or equal to the	Ble_Un_S

	second value, when comparing unsigned integer values or unordered float values.	
3F	**Transfers control to a target instruction if the first value is less than the second value.**	**Blt**
32	**Transfers control to a target instruction (short form) if the first value is less than the second value.**	**Blt_S**
44	**Transfers control to a target instruction if the first value is less than the second value, when comparing unsigned integer values or unordered float values.**	**Blt_Un**
37	**Transfers control to a target instruction (short form) if the first value is less than the second value, when comparing unsigned integer values or unordered float values.**	**Blt_Un_S**
40	**Transfers control to a target instruction when two unsigned integer values or unordered float values are not equal.**	**Bne_Un**
33	**Transfers control to a target instruction (short form) when two unsigned integer values or unordered float values are not equal.**	**Bne_Un_S**
8C	Converts a value type to an object reference (type **O**).	Box
38	**Unconditionally transfers control to a target instruction.**	**Br**
01	Signals the Common Language Infrastructure (CLI) to inform the debugger that a break point has been tripped.	Break
39	**Transfers control to a target instruction if *value* is false, a null reference (Nothing in Visual Basic), or zero.**	**Brfalse**
2C	**Transfers control to a target instruction if *value* is false, a null reference, or zero.**	**Brfalse_S**
3A	**Transfers control to a target instruction if *value* is true, not null, or non-zero.**	**Brtrue**
2D	**Transfers control to a target instruction (short form) if *value* is true, not null, or non-zero.**	**Brtrue_S**
2B	**Unconditionally transfers control to a target instruction (short form).**	**Br_S**

28	**Calls the method indicated by the passed method descriptor.**	**Call**
29	Calls the method indicated on the evaluation stack (as a pointer to an entry point) with arguments described by a calling convention.	Calli
6F	Calls a late-bound method on an object, pushing the return value onto the evaluation stack.	Callvirt
74	Attempts to cast an object passed by reference to the specified class.	Castclass
FE 01	Compares two values. If they are equal, the integer value 1 (**int32**) is pushed onto the evaluation stack; otherwise 0 (**int32**) is pushed onto the evaluation stack.	Ceq
FE 02	Compares two values. If the first value is greater than the second, the integer value 1 (**int32**) is pushed onto the evaluation stack; otherwise 0 (**int32**) is pushed onto the evaluation stack.	Cgt
FE 03	Compares two unsigned or unordered values. If the first value is greater than the second, the integer value 1 (**int32**) is pushed onto the evaluation stack; otherwise 0 (**int32**) is pushed onto the evaluation stack.	Cgt_Un
C3	Throws ArithmeticException if value is not a finite number.	Ckfinite
FE 04	**Compares two values. If the first value is less than the second, the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.**	**Clt**
FE 03	**Compares the unsigned or unordered values *value1* and *value2*. If *value1* is less than *value2*, then the integer value 1 (int32) is pushed onto the evaluation stack; otherwise 0 (int32) is pushed onto the evaluation stack.**	**Clt_Un**
D3	Converts the value on top of the evaluation stack to **natural int**.	Conv_I
67	Converts the value on top of the evaluation stack to **int8**, then extends (pads) it to **int32**.	Conv_I1
68	Converts the value on top of the evaluation stack to **int16**, then extends (pads) it to **int32**.	Conv_I2

69	Converts the value on top of the evaluation stack to **int32.**	Conv_I4
6A	Converts the value on top of the evaluation stack to **int64.**	Conv_I8
D4	Converts the signed value on top of the evaluation stack to signed **natural int,** throwing OverflowException on overflow.	Conv_Ovf_I
B3	Converts the signed value on top of the evaluation stack to signed **int8** and extends it to **int32,** throwing OverflowException on overflow.	Conv_Ovf_I1
82	Converts the unsigned value on top of the evaluation stack to signed **int8** and extends it to **int32,** throwing OverflowException on overflow.	Conv_Ovf_I1_Un
B5	Converts the signed value on top of the evaluation stack to signed **int16** and extending it to **int32,** throwing OverflowException on overflow.	Conv_Ovf_I2
83	Converts the unsigned value on top of the evaluation stack to signed **int16** and extends it to **int32,** throwing OverflowException on overflow.	Conv_Ovf_I2_Un
B7	Converts the signed value on top of the evaluation tack to signed **int32,** throwing OverflowException on overflow.	Conv_Ovf_I4
84	Converts the unsigned value on top of the evaluation stack to signed **int32,** throwing OverflowException on overflow.	Conv_Ovf_I4_Un
B9	Converts the signed value on top of the evaluation stack to signed **int64,** throwing OverflowException on overflow.	Conv_Ovf_I8
B9	Converts the unsigned value on top of the evaluation stack to signed **int64,** throwing OverflowException on overflow.	Conv_Ovf_I8_Un
85	Converts the unsigned value on top of the evaluation stack to signed **natural int,** throwing OverflowException on overflow.	Conv_Ovf_I_Un
D5	Converts the signed value on top of the evaluation stack to **unsigned natural int,** throwing OverflowException on overflow.	Conv_Ovf_U
B4	Converts the signed value on top of the evaluation stack to **unsigned int8** and extends it to **int32,** throwing OverflowException on overflow.	Conv_Ovf_U1
86	Converts the unsigned value on top of the evaluation stack to **unsigned int8** and extends it to **int32,** throwing	Conv_Ovf_U1_Un

	OverflowException on overflow.	
B6	Converts the signed value on top of the evaluation stack to **unsigned int16** and extends it to **int32**, throwing OverflowException on overflow.	Conv_Ovf_U2
87	Converts the unsigned value on top of the evaluation stack to **unsigned int16** and extends it to **int32**, throwing OverflowException on overflow.	Conv_Ovf_U2_Un
B8	Converts the signed value on top of the evaluation stack to **unsigned int32**, throwing OverflowException on overflow.	Conv_Ovf_U4
88	Converts the unsigned value on top of the evaluation stack to **unsigned int32**, throwing OverflowException on overflow.	Conv_Ovf_U4_Un
BA	Converts the signed value on top of the evaluation stack to **unsigned int64**, throwing OverflowException on overflow.	Conv_Ovf_U8
89	Converts the unsigned value on top of the evaluation stack to **unsigned int64**, throwing OverflowException on overflow.	Conv_Ovf_U8_Un
8B	Converts the unsigned value on top of the evaluation stack to **unsigned natural int**, throwing OverflowException on overflow.	Conv_Ovf_U_Un
6B	Converts the value on top of the evaluation stack to **float32**.	Conv_R4
6C	Converts the value on top of the evaluation stack to **float64**.	Conv_R8
76	Converts the unsigned integer value on top of the evaluation stack to **float32**.	Conv_R_Un
E0	Converts the value on top of the evaluation stack to **unsigned natural int**, and extends it to **natural int**.	Conv_U
D2	Converts the value on top of the evaluation stack to **unsigned int8**, and extends it to **int32**.	Conv_U1
D1	Converts the value on top of the evaluation stack to **unsigned int16**, and extends it to **int32**.	Conv_U2
6D	Converts the value on top of the evaluation stack to **unsigned int32**, and extends it to **int32**.	Conv_U4

6E	Converts the value on top of the evaluation stack to **unsigned int64**, and extends it to **int64**.	Conv_U8
FE 17	Copies a specified number bytes from a source address to a destination address.	Cpblk
70	Copies the value type located at the address of an object (type **&**, ***** or **natural int**) to the address of the destination object (type **&**, ***** or **natural int**).	Cpobj
5B	Divides two values and pushes the result as a floating-point (type **F**) or quotient (type **int32**) onto the evaluation stack.	Div
5C	Divides two unsigned integer values and pushes the result (**int32**) onto the evaluation stack.	Div_Un
25	Copies the current topmost value on the evaluation stack, and then pushes the copy onto the evaluation stack.	Dup
FE 11	Transfers control from the **filter** clause of an exception back to the Common Language Infrastructure (CLI) exception handler.	Endfilter
DC	Transfers control from the **fault** or **finally** clause of an exception block back to the Common Language Infrastructure (CLI) exception handler.	Endfinally
FE 18	Initializes a specified block of memory at a specific address to a given size and initial value.	Initblk
FE 15	Initializes all the fields of the object at a specific address to a null reference or a 0 of the appropriate primitive type.	Initobj
75	Tests whether an object reference (type **O**) is an instance of a particular class.	Isinst
27	**Exits current method and jumps to specified method.**	**Jmp**
FE 09	**Loads an argument (referenced by a specified index value) onto the stack.**	**Ldarg**
FE 0A	**Load an argument address onto the evaluation stack.**	**Ldarga**
0F	**Load an argument address, in short form, onto the evaluation stack.**	**Ldarga_S**
02	**Loads the argument at index 0 onto the evaluation stack.**	**Ldarg_0**

03	**Loads the argument at index 1 onto the evaluation stack.**	**Ldarg_1**
04	**Loads the argument at index 2 onto the evaluation stack.**	**Ldarg_2**
05	**Loads the argument at index 3 onto the evaluation stack.**	**Ldarg_3**
0E	**Loads the argument (referenced by a specified short form index) onto the evaluation stack.**	**Ldarg_S**
20	**Pushes a supplied value of type int32 onto the evaluation stack as an int32.**	**Ldc_I4**
16	**Pushes the integer value of 0 onto the evaluation stack as an int32.**	**Ldc_I4_0**
17	**Pushes the integer value of 1 onto the evaluation stack as an int32.**	**Ldc_I4_1**
18	**Pushes the integer value of 2 onto the evaluation stack as an int32.**	**Ldc_I4_2**
19	**Pushes the integer value of 3 onto the evaluation stack as an int32.**	**Ldc_I4_3**
1A	**Pushes the integer value of 4 onto the evaluation stack as an int32.**	**Ldc_I4_4**
1B	**Pushes the integer value of 5 onto the evaluation stack as an int32.**	**Ldc_I4_5**
1C	**Pushes the integer value of 6 onto the evaluation stack as an int32.**	**Ldc_I4_6**
1D	**Pushes the integer value of 7 onto the evaluation stack as an int32.**	**Ldc_I4_7**
1E	**Pushes the integer value of 8 onto the evaluation stack as an int32.**	**Ldc_I4_8**
15	**Pushes the integer value of -1 onto the evaluation stack as an int32.**	**Ldc_I4_M1**
1F	**Pushes the supplied int8 value onto the evaluation stack as an int32, short form.**	**Ldc_I4_S**
21	Pushes a supplied value of type **int64** onto the evaluation stack as an **int64.**	Ldc_I8
22	Pushes a supplied value of type **float32** onto the evaluation stack as type **F** (float).	Ldc_R4

23	Pushes a supplied value of type **float64** onto the evaluation stack as type **F** (float).	Ldc_R8
8F	Loads the address of the array element at a specified array index onto the top of the evaluation stack as type **&** (managed pointer).	Ldelema
97	Loads the element with type **natural int** at a specified array index onto the top of the evaluation stack as a **natural int**.	Ldelem_I
90	Loads the element with type **int8** at a specified array index onto the top of the evaluation stack as an **int32**.	Ldelem_I1
92	Loads the element with type **int16** at a specified array index onto the top of the evaluation stack as an **int32**.	Ldelem_I2
94	Loads the element with type **int32** at a specified array index onto the top of the evaluation stack as an **int32**.	Ldelem_I4
96	Loads the element with type **int64** at a specified array index onto the top of the evaluation stack as an **int64**.	Ldelem_I8
98	Loads the element with type **float32** at a specified array index onto the top of the evaluation stack as type **F** (float).	Ldelem_R4
99	Loads the element with type **float64** at a specified array index onto the top of the evaluation stack as type **F** (float).	Ldelem_R8
9A	Loads the element containing an object reference at a specified array index onto the top of the evaluation stack as type **O** (object reference).	Ldelem_Ref
91	Loads the element with type **unsigned int8** at a specified array index onto the top of the evaluation stack as an **int32**.	Ldelem_U1
93	Loads the element with type **unsigned int16** at a specified array index onto the top of the evaluation stack as an **int32**.	Ldelem_U2
94	Loads the element with type **unsigned int32** at a specified array index onto the top of the evaluation stack as an **int32**.	Ldelem_U4
7B	Finds the value of a field in the object whose reference is currently on the evaluation stack.	Ldfld
7C	Finds the address of a field in the object whose reference is currently on the evaluation stack.	Ldflda
FE 06	Pushes an unmanaged pointer (type **natural int**) to the native code implementing a specific method onto	Ldftn

	the evaluation stack.	
4D	Loads a value of type **natural int** as a **natural int** onto the evaluation stack indirectly.	Ldind_I
46	Loads a value of type **int8** as an **int32** onto the evaluation stack indirectly.	Ldind_I1
48	Loads a value of type **int16** as an **int32** onto the evaluation stack indirectly.	Ldind_I2
4A	Loads a value of type **int32** as an **int32** onto the evaluation stack indirectly.	Ldind_I4
4C	Loads a value of type **int64** as an **int64** onto the evaluation stack indirectly.	Ldind_I8
4E	Loads a value of type **float32** as a type **F** (float) onto the evaluation stack indirectly.	Ldind_R4
4F	Loads a value of type **float64** as a type **F** (float) onto the evaluation stack indirectly.	Ldind_R8
50	Loads an object reference as a type **O** (object reference) onto the evaluation stack indirectly.	Ldind_Ref
47	Loads a value of type **unsigned int8** as an **int32** onto the evaluation stack indirectly.	Ldind_U1
49	Loads a value of type **unsigned int16** as an **int32** onto the evaluation stack indirectly.	Ldind_U2
4B	Loads a value of type **unsigned int32** as an **int32** onto the evaluation stack indirectly.	Ldind_U4
8E	Pushes the number of elements of a zero-based, one-dimensional array onto the evaluation stack.	Ldlen
FE 06	Loads the local variable at a specific index onto the evaluation stack.	Ldloc
FE 0D	Loads the address of the local variable at a specific index onto the evaluation stack.	Ldloca
12	Loads the address of the local variable at a specific index onto the evaluation stack, short form.	Ldloca_S
06	Loads the local variable at index 0 onto the evaluation stack.	Ldloc_0
07	Loads the local variable at index 1 onto the evaluation stack.	Ldloc_1

08	Loads the local variable at index 2 onto the evaluation stack.	Ldloc_2
09	Loads the local variable at index 3 onto the evaluation stack.	Ldloc_3
11	Loads the local variable at a specific index onto the evaluation stack, short form.	Ldloc_S
14	Pushes a null reference (type **O**) onto the evaluation stack.	Ldnull
71	Copies the value type object pointed to by an address to the top of the evaluation stack.	Ldobj
7E	Pushes the value of a static field onto the evaluation stack.	Ldsfld
7F	Pushes the address of a static field onto the evaluation stack.	Ldsflda
72	**Pushes a new object reference to a string literal stored in the metadata.**	**Ldstr**
D0	Converts a metadata token to its runtime representation, pushing it onto the evaluation stack.	Ldtoken
FE 07	Pushes an unmanaged pointer (type **natural int**) to the native code implementing a particular virtual method associated with a specified object onto the evaluation stack.	Ldvirtftn
DD	**Exits a protected region of code, unconditionally tranferring control to a specific target instruction.**	**Leave**
DE	**Exits a protected region of code, unconditionally tranferring control to a target instruction (short form).**	**Leave_S**
FE 0F	Allocates a certain number of bytes from the local dynamic memory pool and pushes the address (a transient pointer, type *****) of the first allocated byte onto the evaluation stack.	Localloc
C6	Pushes a typed reference to an instance of a specific type onto the evaluation stack.	Mkrefany
5A	**Multiplies two values and pushes the result on the evaluation stack.**	**Mul**
D8	**Multiplies two integer values, performs an overflow check, and pushes the result onto the evaluation stack.**	**Mul_Ovf**

D9	**Multiplies two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.**	**Mul_Ovf_Un**
65	**Negates a value and pushes the result onto the evaluation stack.**	**Neg**
8D	Pushes an object reference to a new zero-based, one-dimensional array whose elements are of a specific type onto the evaluation stack.	Newarr
73	**Creates a new object or a new instance of a value type, pushing an object reference (type O) onto the evaluation stack.**	**Newobj**
90	**Fills space if opcodes are patched. No meaningful operation is performed although a processing cycle can be consumed.**	**Nop**
66	**Computes the bitwise complement of the integer value on top of the stack and pushes the result onto the evaluation stack as the same type.**	**Not**
60	**Compute the bitwise complement of the two integer values on top of the stack and pushes the result onto the evaluation stack.**	**Or**
26	**Removes the value currently on top of the evaluation stack.**	**Pop**
FE 1D	Retrieves the type token embedded in a typed reference.	Refanytype
C2	Retrieves the address (type **&**) embedded in a typed reference.	Refanyval
5D	**Divides two values and pushes the remainder onto the evaluation stack.**	**Rem**
5E	**Divides two unsigned values and pushes the remainder onto the evaluation stack.**	**Rem_Un**
2A	**Returns from the current method, pushing a return value (if present) from the caller's evaluation stack onto the callee's evaluation stack.**	**Ret**
FE 1A	**Rethrows the current exception.**	**Rethrow**
62	Shifts an integer value to the left (in zeroes) by a specified number of bits, pushing the result onto the evaluation stack.	Shl
63	Shifts an integer value (in sign) to the right by a specified number of bits, pushing the result onto the	Shr

	evaluation stack.	
64	Shifts an unsigned integer value (in zeroes) to the right by a specified number of bits, pushing the result onto the evaluation stack.	Shr_Un
FE 1C	Pushes the size, in bytes, of a supplied value type onto the evaluation stack.	Sizeof
FE 0B	Stores the value on top of the evaluation stack in the argument slot at a specified index.	Starg
10	Stores the value on top of the evaluation stack in the argument slot at a specified index, short form.	Starg_S
9B	Replaces the array element at a given index with the **natural int** value on the evaluation stack.	Stelem_I
9C	Replaces the array element at a given index with the **int8** value on the evaluation stack.	Stelem_I1
9D	Replaces the array element at a given index with the **int16** value on the evaluation stack.	Stelem_I2
9E	Replaces the array element at a given index with the **int32** value on the evaluation stack.	Stelem_I4
9F	Replaces the array element at a given index with the **int64** value on the evaluation stack.	Stelem_I8
A0	Replaces the array element at a given index with the **float32** value on the evaluation stack.	Stelem_R4
A1	Replaces the array element at a given index with the **float64** value on the evaluation stack.	Stelem_R8
A2	Replaces the array element at a given index with the object ref value (type **O**) on the evaluation stack.	Stelem_Ref
7D	Replaces the value stored in the field of an object reference or pointer with a new value.	Stfld
DF	Stores a value of type **natural int** at a supplied address.	Stind_I
52	**Stores a value of type int8 at a supplied address.**	**Stind_I1**
53	**Stores a value of type int16 at a supplied address.**	**Stind_I2**

54	**Stores a value of type int32 at a supplied address.**	**Stind_I4**
55	Stores a value of type **int64** at a supplied address.	Stind_I8
56	Stores a value of type **float32** at a supplied address.	Stind_R4
57	Stores a value of type **float64** at a supplied address.	Stind_R8
51	Stores a object reference value at a supplied address.	Stind_Ref
FE 0E	**Pops the current value from the top of the evaluation stack and stores it in a the local variable list at a specified index.**	**Stloc**
0A	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 0.	Stloc_0
0B	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 1.	Stloc_1
0C	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 2.	Stloc_2
0D	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 3.	Stloc_3
13	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at *index* (short form).	Stloc_S
81	Copies a value of a specified type from the evaluation stack into a supplied memory address.	Stobj
80	Replaces the value of a static field with a value from the evaluation stack.	Stsfld
59	**Subtracts one value from another and pushes the result onto the evaluation stack.**	**Sub**
DA	**Subtracts one integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.**	**Sub_Ovf**
DB	**Subtracts one unsigned integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.**	**Sub_Ovf_Un**

45	**Implements a jump table.**	**Switch**
FE 14	Performs a postfixed method call instruction such that the current method's stack frame is removed before the actual call instruction is executed.	Tailcall
7A	**Throws the exception object currently on the evaluation stack.**	**Throw**
FE 12	Indicates that an address currently atop the evaluation stack might not be aligned to the natural size of the immediately following **ldind**, **stind**, **ldfld**, **stfld**, **ldobj**, **stobj**, **initblk**, or **cpblk** instruction.	Unaligned
79	Converts the boxed representation of a value type to its unboxed form.	Unbox
FE 13	Specifies that an address currently atop the evaluation stack might be volatile, and the results of reading that location cannot be cached or that multiple stores to that location cannot be suppressed.	Volatile
61	**Computes the bitwise XOR of the top two values on the evaluation stack, pushing the result onto the evaluation stack.**	**Xor**

THE END

I hope this was a good tutorial and that you enjoyed reading it...
In next tutorials we will see how to replace bytes to invert jump or nop calls.
That's all for now....

Tkc
Wednesday, December 27, 2006