

tKC

.NET reversing Tips

1



Tools & OPCODEs

Attention

Published for educational Purposes only

Welcome:-

Welcome to the wonderful world of cracking.

In this series of tutorials we will be targeting .net applications, If You search the web you will find [if you are lucky] few tutorials on this topic, although the .net technology has been around for a while now but the number of applications developed with it is still nothing compared to those developed with [old] languages like C++ or Delphi or VB.

Cracking is the art of removing or disabling copy protected coding from programs, this manual will go into some detail on removing copy protection from programs. If you feel offended by this, then I would suggest you stop here....

Introduction:-

Cracking .net applications is still a new topic, and it has any similarities with classic cracking, but I think It's much easier than cracking native code application, the ability to see the source code and what Functions are exactly being used with detailed information about every line being executed is what makes this easier, here no Soft-Ice or W32Dasm are needed, we may need soft-ice some time later, I don't know, but there is a new workflow here, and it's basically dependent upon decompiling the IL code of the targeted assembly and analyzing the source code that tools like reflector can generate, so a good understanding of any one of the new .NET languages is essential and can be a great advantage for you.

Tools:-

Before we begin, you will need certain tools, I use some of them, but this doesn't Mean we will need them all Here, just make sure that you have them and try to learn how To use them...

1. Reflector : .NET assembly Decompiler

Reflector is a class browser for .NET components. It allows browsing and Searching the Meta data, IL instructions, resources and XML documentation Stored in a .NET assembly.

Think of it as W32Dasm, this is an essential tool to explore The assembly code and resources and I think it's the best Tool available for this purpose; you can download the latest version From here "<http://www.aisto.com/roeder/dotnet>"

2. ildasm : Microsoft IL Disassembler

The MSIL Disassembler is a companion tool to the MSIL Assembler (Ilasm.exe). Ildasm.exe takes a portable executable (PE) file that contains Microsoft intermediate language (MSIL) code and creates a text file suitable as input to Ilasm.exe.

Although Reflector can decompile a .net assembly into IL codes but it won't show us the real byte of IL instructions in the assembly. Ildasm has an option which reveals the Hex value of bytes that represent IL instructions in every method.

Take the "**BLE**" instruction for example:
It transfers control to a target instruction if the first value is less than or equal to the second value. [Like JLE in native code]
And its actual byte that you will see in a hex editor is "**3E**".

When you need to invert this jump you will need the actual bytes of the instruction "**BGT**" which transfers control to a target instruction if the first value is greater than the second value. And it's represented as "**3D**".

So you will need to replace the **3E** with **3D** in hex editor to get the job done....

Now look at this dump of some procedure from ildasm.

```
-----
.method public specialname instance class Scroller/Scroller/Title
    get_Titles(object Index) cil managed
// SIG: 20 01 12 0C 1C
{
    // Method begins at RVA 0xcd7c
    // Code size      23 (0x17)
    .maxstack 2
    .locals init (class Scroller/Scroller/Title V_0)
    IL_0000: /* 02 |                      */ ldarg.0
    IL_0001: /* 7B | (04) 00000D */ ldflld
    IL_0006: /* 03 |                      */ ldarg.1
    IL_0007: /* 28 | (0A) 00005C */ call      object
    IL_000c: /* 6F | (0A) 00005D */ callvirt  instance object
    IL_0011: /* 74 | (02) 000003 */ castclass
    Scroller/Scroller/Title
    IL_0016: /* 2A |                      */ ret
} // end of method Scroller::get_Titles
-----
```

This is a piece of code in IL.

IL_0000	:	Line number
02	:	Actual byte of IL instructions in this line
ldarg.0	:	IL instruction

Don't worry if you don't understand this code, we will talk about this more later...

The advantage of seeing the bytes that represent any IL instructions comes when we want to nop a call or patch any part of the code. We can also use the RVA value to calculate the offset of this method in the file. And I will explain both methods later.

You will find ildasm.exe in this folder after you install .Net framework SDK
"..\..\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin\"

3. **Hex workshop** : **Hex editor**

I recommend this or any hex editor that can search for Unicode Strings and locate a series of bytes.
Very easy to find on the web, just Google it!

4. **CFF explorer** : **General PE files explorer**

This is a great tool that will help us explore the contents of Any PE file including metadata tables and resources inside The assembly, you can download it from "www.pmode.com"

5. **SNS remover** : **Strong Name signature remover**

Some .net assemblies are signed with a digital signature to prevent Any tampering or modification after the assembly is released, if You modify any byte in a Strongly Named assembly then the .Net runtime Will refuse to start that assembly, but there is a small utility called "SNS remover" That can be used to remove the signature field from the signed assembly. By the way there is an option in CFF explorer that allows you to remove the Strong Name signature from the .net assembly and rebuild the PE file again But I prefer this small utility, you can download both from "www.pmode.com"

6. **PEBrowse Professional** : **Disassembler/Debugger**

A debugger and Disassembler for PE files that can disassemble and debug .net assemblies too, It shows the IL instructions and their actual bytes, You will be able to break on any JIT compiler event and this is cool!

With this debugger you can step through the .net IL instructions And understand what's going in the background... You can get it from "www.smidgeonsoft.com"

7. **.Net Generic Unpacker** : **.Net assembly Unpacker**

You will sometimes need this tool to dump the entire .net assembly PE file, some protection software like "**.Net reactor**" will pack Your .net assembly and will produce a non-MSIL PE file that will Unpack your assembly and execute it from memory when launched, and This technique is used to prevent any access to the IL code of the Original assembly itself, but you can easily overcome this by using A simple .net generic unpacker, you can download one from "www.ntcore.com"

Finally:-

Sometimes you will see that "Reflector" won't be able to Decompile some procedure or function into your favorite language like C# or VB or Delphi, so you will have to familiarize yourself with The IL instructions, and this is like learning the classic

Assembly language to crack native code applications, but you
Will see that IL is easier and faster to learn!

The Opcodes:-

This is the most important point of cracking, as you know the .net application Represent their program instruction in MSIL format which stands for Microsoft Intermediate language, your source code is not translated to native machine-code when you compile it in visual studio RAD, but it's translated to this format which will later be compiled into native code by JIT compiler, JIT stands for just-in-time Compiler, meaning that only certain parts of your programs will be translated into Native code and executed on demand.

This is a disassembly for a piece of code from ildasm.

```
=====
.method public specialname instance class Scroller.Scroller/Title
    get_Titles(object Index) cil managed
// SIG: 20 01 12 0C 1C
{
    // Method begins at RVA 0xcd7c
    // Code size      23 (0x17)
    .maxstack 2
    .locals init (class Scroller.Scroller/Title V_0)
    IL_0000: /* 02 | */ ldarg.0
    IL_0001: /* 7B | (04)00000D */ ldflld
    IL_0006: /* 03 | */ ldarg.1
    IL_0007: /* 28 | (0A)00005C */ call
    IL_000c: /* 6F | (0A)00005D */ callvirt
    IL_0011: /* 74 | (02)000003 */ castclass
    IL_0016: /* 2A | */ ret
} // end of method Scroller::get_Titles
=====
```

This is the first line of this procedure:

IL_0000:	/* 02	*/ ldarg.0
Line number	Actual byte(s)	IL instruction

Opcodes are representations of the Microsoft Intermediate Language (MSIL) instructions.

If you are into cracking then you know what these instructions mean !

JMP JNE JLE NOP CALL etc...

MSIL Opcodes are different from native Opcodes designed for the Intel Processors, for example after you know the offset of a CALL in a native code program you open that program in a hex editor and overwrite the right number of bytes that call that function or whatever with **90s** which stands for NOP in native machine code.

In MSIL the NOP is represented by **00** not **90**, so it's important here to list the Opcodes For MSIL with their actual byte values now, we won't need to use them

all when We crack a program, most of the time we will be using the **NOP** instruction and inverting Jumps to get the job done.

I will mark the important Opcodes in cracking in gray and orange colors so try to focus on them.

Opcodes list

OPCODE	What It does	Actual bytes
Add	Adds two values and pushes the result onto the evaluation stack.	58
Add_Ovf	Adds two integers, performs an overflow check, and pushes the result onto the evaluation stack.	D6
Add_Ovf_Un	Adds two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D7
And	Computes the bitwise AND of two values and pushes the result onto the evaluation stack.	5F
Arglist	Returns an unmanaged pointer to the argument list of the current method.	FE 00
Beq	Transfers control to a target instruction if two values are equal.	3B
Beq_S	Transfers control to a target instruction (short form) if two values are equal.	2E
Bge	Transfers control to a target instruction if the first value is greater than or equal to the second value.	3C
Bge_S	Transfers control to a target instruction (short form) if the first value is greater than or equal to the second value.	2F
Bge_Un	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.	41
Bge_Un_S	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.	34
Bgt	Transfers control to a target instruction if the first value is greater than the second value.	3D
Bgt_S	Transfers control to a target instruction (short form) if the first value is greater than the second value.	30

Bgt_Un	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.	42
Bgt_Un_S	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.	35
Ble	Transfers control to a target instruction if the first value is less than or equal to the second value.	3E
Ble_S	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value.	31
Ble_Un	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.	43
Ble_Un_S	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value, when comparing unsigned integer values or unordered float values.	36
Blt	Transfers control to a target instruction if the first value is less than the second value.	3F
Blt_S	Transfers control to a target instruction (short form) if the first value is less than the second value.	32
Blt_Un	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.	44
Blt_Un_S	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.	37
Bne_Un	Transfers control to a target instruction when two unsigned integer values or unordered float values are not equal.	40
Bne_Un_S	Transfers control to a target instruction (short form) when two unsigned integer values or unordered float values are not equal.	33
Box	Converts a value type to an object reference (type 0).	8C
Br	Unconditionally transfers control to a target instruction.	38
Break	Signals the Common Language Infrastructure (CLI) to inform the debugger that a break point has been tripped.	01
Brfalse	Transfers control to a target instruction if value is false, a null reference (Nothing in Visual Basic), or zero.	39

Brfalse_S	Transfers control to a target instruction if value is false, a null reference, or zero.	2C
Brtrue	Transfers control to a target instruction if value is true, not null, or non-zero.	3A
Brtrue_S	Transfers control to a target instruction (short form) if value is true, not null, or non-zero.	2D
Br_S	Unconditionally transfers control to a target instruction (short form).	2B
Call	Calls the method indicated by the passed method descriptor.	28
Calli	Calls the method indicated on the evaluation stack (as a pointer to an entry point) with arguments described by a calling convention.	29
Callvirt	Calls a late-bound method on an object, pushing the return value onto the evaluation stack.	6F
Castclass	Attempts to cast an object passed by reference to the specified class.	74
Ceq	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.	FE 01
Cgt	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.	FE 02
Cgt_Un	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.	FE 03
Ckfinite	Throws ArithmeticException if value is not a finite number.	C3
Clt	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.	FE 04
Clt_Un	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.	FE 03
Conv_I	Converts the value on top of the evaluation stack to natural int .	D3
Conv_I1	Converts the value on top of the evaluation stack to int8 , then extends (pads) it to int32 .	67

Conv_I2	Converts the value on top of the evaluation stack to int16 , then extends (pads) it to int32 .	68
Conv_I4	Converts the value on top of the evaluation stack to int32 .	69
Conv_I8	Converts the value on top of the evaluation stack to int64 .	6A
Conv_Ovf_I	Converts the signed value on top of the evaluation stack to signed natural int , throwing OverflowException on overflow.	D4
Conv_Ovf_I1	Converts the signed value on top of the evaluation stack to signed int8 and extends it to int32 , throwing OverflowException on overflow.	B3
Conv_Ovf_I1_Un	Converts the unsigned value on top of the evaluation stack to signed int8 and extends it to int32 , throwing OverflowException on overflow.	82
Conv_Ovf_I2	Converts the signed value on top of the evaluation stack to signed int16 and extending it to int32 , throwing OverflowException on overflow.	B5
Conv_Ovf_I2_Un	Converts the unsigned value on top of the evaluation stack to signed int16 and extends it to int32 , throwing OverflowException on overflow.	83
Conv_Ovf_I4	Converts the signed value on top of the evaluation stack to signed int32 , throwing OverflowException on overflow.	B7
Conv_Ovf_I4_Un	Converts the unsigned value on top of the evaluation stack to signed int32 , throwing OverflowException on overflow.	84
Conv_Ovf_I8	Converts the signed value on top of the evaluation stack to signed int64 , throwing OverflowException on overflow.	B9
Conv_Ovf_I8_Un	Converts the unsigned value on top of the evaluation stack to signed int64 , throwing OverflowException on overflow.	B9
Conv_Ovf_I_Un	Converts the unsigned value on top of the evaluation stack to signed natural int , throwing OverflowException on overflow.	85
Conv_Ovf_U	Converts the signed value on top of the evaluation stack to unsigned natural int , throwing OverflowException on overflow.	D5
Conv_Ovf_U1	Converts the signed value on top of the evaluation stack to unsigned int8 and extends it to int32 , throwing OverflowException on overflow.	B4
Conv_Ovf_U1_Un	Converts the unsigned value on top of the evaluation stack to unsigned int8 and extends it to int32 , throwing OverflowException on overflow.	86
Conv_Ovf_U2	Converts the signed value on top of the evaluation stack to unsigned int16 and extends it to int32 , throwing OverflowException on overflow.	B6

Conv_Ovf_U2_Un	Converts the unsigned value on top of the evaluation stack to unsigned int16 and extends it to int32 , throwing OverflowException on overflow.	87
Conv_Ovf_U4	Converts the signed value on top of the evaluation stack to unsigned int32 , throwing OverflowException on overflow.	B8
Conv_Ovf_U4_Un	Converts the unsigned value on top of the evaluation stack to unsigned int32 , throwing OverflowException on overflow.	88
Conv_Ovf_U8	Converts the signed value on top of the evaluation stack to unsigned int64 , throwing OverflowException on overflow.	BA
Conv_Ovf_U8_Un	Converts the unsigned value on top of the evaluation stack to unsigned int64 , throwing OverflowException on overflow.	89
Conv_Ovf_U_Un	Converts the unsigned value on top of the evaluation stack to unsigned natural int , throwing OverflowException on overflow.	8B
Conv_R4	Converts the value on top of the evaluation stack to float32 .	6B
Conv_R8	Converts the value on top of the evaluation stack to float64 .	6C
Conv_R_Un	Converts the unsigned integer value on top of the evaluation stack to float32 .	76
Conv_U	Converts the value on top of the evaluation stack to unsigned natural int , and extends it to natural int .	E0
Conv_U1	Converts the value on top of the evaluation stack to unsigned int8 , and extends it to int32 .	D2
Conv_U2	Converts the value on top of the evaluation stack to unsigned int16 , and extends it to int32 .	D1
Conv_U4	Converts the value on top of the evaluation stack to unsigned int32 , and extends it to int32 .	6D
Conv_U8	Converts the value on top of the evaluation stack to unsigned int64 , and extends it to int64 .	6E
Cpblk	Copies a specified number bytes from a source address to a destination address.	FE 17
Cpobj	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).	70
Div	Divides two values and pushes the result as a floating-point (type F) or quotient (type int32) onto the evaluation stack.	5B
Div_Un	Divides two unsigned integer values and pushes the result (int32) onto the evaluation stack.	5C

Dup	Copies the current topmost value on the evaluation stack, and then pushes the copy onto the evaluation stack.	25
Endfilter	Transfers control from the filter clause of an exception back to the Common Language Infrastructure (CLI) exception handler.	FE 11
Endfinally	Transfers control from the fault or finally clause of an exception block back to the Common Language Infrastructure (CLI) exception handler.	DC
Initblk	Initializes a specified block of memory at a specific address to a given size and initial value.	FE 18
Initobj	Initializes all the fields of the object at a specific address to a null reference or a 0 of the appropriate primitive type.	FE 15
Isinst	Tests whether an object reference (type O) is an instance of a particular class.	75
Jmp	Exits current method and jumps to specified method.	27
Ldarg	Loads an argument (referenced by a specified index value) onto the stack.	FE 09
Ldarga	Load an argument address onto the evaluation stack.	FE 0A
Ldarga_S	Load an argument address, in short form, onto the evaluation stack.	0F
Ldarg_0	Loads the argument at index 0 onto the evaluation stack.	02
Ldarg_1	Loads the argument at index 1 onto the evaluation stack.	03
Ldarg_2	Loads the argument at index 2 onto the evaluation stack.	04
Ldarg_3	Loads the argument at index 3 onto the evaluation stack.	05
Ldarg_S	Loads the argument (referenced by a specified short form index) onto the evaluation stack.	0E
Ldc_I4	Pushes a supplied value of type int32 onto the evaluation stack as an int32.	20
Ldc_I4_0	Pushes the integer value of 0 onto the evaluation stack as an int32.	16
Ldc_I4_1	Pushes the integer value of 1 onto the evaluation stack as an int32.	17

Ldc_I4_2	Pushes the integer value of 2 onto the evaluation stack as an int32 .	18
Ldc_I4_3	Pushes the integer value of 3 onto the evaluation stack as an int32 .	19
Ldc_I4_4	Pushes the integer value of 4 onto the evaluation stack as an int32 .	1A
Ldc_I4_5	Pushes the integer value of 5 onto the evaluation stack as an int32 .	1B
Ldc_I4_6	Pushes the integer value of 6 onto the evaluation stack as an int32 .	1C
Ldc_I4_7	Pushes the integer value of 7 onto the evaluation stack as an int32 .	1D
Ldc_I4_8	Pushes the integer value of 8 onto the evaluation stack as an int32 .	1E
Ldc_I4_M1	Pushes the integer value of -1 onto the evaluation stack as an int32 .	15
Ldc_I4_S	Pushes the supplied int8 value onto the evaluation stack as an int32 , short form.	1F
Ldc_I8	Pushes a supplied value of type int64 onto the evaluation stack as an int64 .	21
Ldc_R4	Pushes a supplied value of type float32 onto the evaluation stack as type F (float).	22
Ldc_R8	Pushes a supplied value of type float64 onto the evaluation stack as type F (float).	23
Ldelema	Loads the address of the array element at a specified array index onto the top of the evaluation stack as type & (managed pointer).	8F
Ldelem_I	Loads the element with type natural int at a specified array index onto the top of the evaluation stack as a natural int .	97
Ldelem_I1	Loads the element with type int8 at a specified array index onto the top of the evaluation stack as an int32 .	90
Ldelem_I2	Loads the element with type int16 at a specified array index onto the top of the evaluation stack as an int32 .	92
Ldelem_I4	Loads the element with type int32 at a specified array index onto the top of the evaluation stack as an int32 .	94
Ldelem_I8	Loads the element with type int64 at a specified array index onto the top of the evaluation stack as an int64 .	96

Ldelem_R4	Loads the element with type float32 at a specified array index onto the top of the evaluation stack as type F (float).	98
Ldelem_R8	Loads the element with type float64 at a specified array index onto the top of the evaluation stack as type F (float).	99
Ldelem_Ref	Loads the element containing an object reference at a specified array index onto the top of the evaluation stack as type O (object reference).	9A
Ldelem_U1	Loads the element with type unsigned int8 at a specified array index onto the top of the evaluation stack as an int32 .	91
Ldelem_U2	Loads the element with type unsigned int16 at a specified array index onto the top of the evaluation stack as an int32 .	93
Ldelem_U4	Loads the element with type unsigned int32 at a specified array index onto the top of the evaluation stack as an int32 .	94
Ldflld	Finds the value of a field in the object whose reference is currently on the evaluation stack.	7B
Ldfllda	Finds the address of a field in the object whose reference is currently on the evaluation stack.	7C
Ldftn	Pushes an unmanaged pointer (type natural int) to the native code implementing a specific method onto the evaluation stack.	FE 06
Ldind_I	Loads a value of type natural int as a natural int onto the evaluation stack indirectly.	4D
Ldind_I1	Loads a value of type int8 as an int32 onto the evaluation stack indirectly.	46
Ldind_I2	Loads a value of type int16 as an int32 onto the evaluation stack indirectly.	48
Ldind_I4	Loads a value of type int32 as an int32 onto the evaluation stack indirectly.	4A
Ldind_I8	Loads a value of type int64 as an int64 onto the evaluation stack indirectly.	4C
Ldind_R4	Loads a value of type float32 as a type F (float) onto the evaluation stack indirectly.	4E
Ldind_R8	Loads a value of type float64 as a type F (float) onto the evaluation stack indirectly.	4F
Ldind_Ref	Loads an object reference as a type O (object reference) onto the evaluation stack indirectly.	50
Ldind_U1	Loads a value of type unsigned int8 as an int32 onto the evaluation stack indirectly.	47

Ldind_U2	Loads a value of type unsigned int16 as an int32 onto the evaluation stack indirectly.	49
Ldind_U4	Loads a value of type unsigned int32 as an int32 onto the evaluation stack indirectly.	4B
Ldlen	Pushes the number of elements of a zero-based, one-dimensional array onto the evaluation stack.	8E
Ldloc	Loads the local variable at a specific index onto the evaluation stack.	FE 06
Ldloca	Loads the address of the local variable at a specific index onto the evaluation stack.	FE 0D
Ldloca_S	Loads the address of the local variable at a specific index onto the evaluation stack, short form.	12
Ldloc_0	Loads the local variable at index 0 onto the evaluation stack.	06
Ldloc_1	Loads the local variable at index 1 onto the evaluation stack.	07
Ldloc_2	Loads the local variable at index 2 onto the evaluation stack.	08
Ldloc_3	Loads the local variable at index 3 onto the evaluation stack.	09
Ldloc_S	Loads the local variable at a specific index onto the evaluation stack, short form.	11
Ldnull	Pushes a null reference (type 0) onto the evaluation stack.	14
Ldobj	Copies the value type object pointed to by an address to the top of the evaluation stack.	71
Ldsfld	Pushes the value of a static field onto the evaluation stack.	7E
Ldsflda	Pushes the address of a static field onto the evaluation stack.	7F
Ldstr	Pushes a new object reference to a string literal stored in the metadata.	72
Ldtoken	Converts a metadata token to its runtime representation, pushing it onto the evaluation stack.	D0
Ldvirtftn	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.	FE 07

Leave	Exits a protected region of code, unconditionally transferring control to a specific target instruction.	DD
Leave_S	Exits a protected region of code, unconditionally transferring control to a target instruction (short form).	DE
Localloc	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.	FE 0F
Mkrefany	Pushes a typed reference to an instance of a specific type onto the evaluation stack.	C6
Mul	Multiplies two values and pushes the result on the evaluation stack.	5A
Mul_Ovf	Multiplies two integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D8
Mul_Ovf_Un	Multiplies two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.	D9
Neg	Negates a value and pushes the result onto the evaluation stack.	65
Newarr	Pushes an object reference to a new zero-based, one-dimensional array whose elements are of a specific type onto the evaluation stack.	8D
Newobj	Creates a new object or a new instance of a value type, pushing an object reference (type O) onto the evaluation stack.	73
Nop	Fills space if opcodes are patched. No meaningful operation is performed although a processing cycle can be consumed.	90
Not	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.	66
Or	Compute the bitwise complement of the two integer values on top of the stack and pushes the result onto the evaluation stack.	60
Pop	Removes the value currently on top of the evaluation stack.	26
Refanytype	Retrieves the type token embedded in a typed reference.	FE 1D
Refanyval	Retrieves the address (type &) embedded in a typed reference.	C2
Rem	Divides two values and pushes the remainder onto the evaluation stack.	5D
Rem_Un	Divides two unsigned values and pushes the remainder onto the evaluation stack.	5E

Ret	Returns from the current method, pushing a return value (if present) from the caller's evaluation stack onto the callee's evaluation stack.	2A
Rethrow	Rethrows the current exception.	FE 1A
Shl	Shifts an integer value to the left (in zeroes) by a specified number of bits, pushing the result onto the evaluation stack.	62
Shr	Shifts an integer value (in sign) to the right by a specified number of bits, pushing the result onto the evaluation stack.	63
Shr_Un	Shifts an unsigned integer value (in zeroes) to the right by a specified number of bits, pushing the result onto the evaluation stack.	64
Sizeof	Pushes the size, in bytes, of a supplied value type onto the evaluation stack.	FE 1C
Starg	Stores the value on top of the evaluation stack in the argument slot at a specified index.	FE 0B
Starg_S	Stores the value on top of the evaluation stack in the argument slot at a specified index, short form.	10
Stelem_I	Replaces the array element at a given index with the natural int value on the evaluation stack.	9B
Stelem_I1	Replaces the array element at a given index with the int8 value on the evaluation stack.	9C
Stelem_I2	Replaces the array element at a given index with the int16 value on the evaluation stack.	9D
Stelem_I4	Replaces the array element at a given index with the int32 value on the evaluation stack.	9E
Stelem_I8	Replaces the array element at a given index with the int64 value on the evaluation stack.	9F
Stelem_R4	Replaces the array element at a given index with the float32 value on the evaluation stack.	A0
Stelem_R8	Replaces the array element at a given index with the float64 value on the evaluation stack.	A1
Stelem_Ref	Replaces the array element at a given index with the object ref value (type O) on the evaluation stack.	A2
Stfld	Replaces the value stored in the field of an object reference or pointer with a new value.	7D
Stind_I	Stores a value of type natural int at a supplied address.	DF

Stind_I1	Stores a value of type int8 at a supplied address.	52
Stind_I2	Stores a value of type int16 at a supplied address.	53
Stind_I4	Stores a value of type int32 at a supplied address.	54
Stind_I8	Stores a value of type int64 at a supplied address.	55
Stind_R4	Stores a value of type float32 at a supplied address.	56
Stind_R8	Stores a value of type float64 at a supplied address.	57
Stind_Ref	Stores a object reference value at a supplied address.	51
Stloc	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at a specified index.	FE 0E
Stloc_0	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 0.	0A
Stloc_1	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 1.	0B
Stloc_2	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 2.	0C
Stloc_3	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at index 3.	0D
Stloc_S	Pops the current value from the top of the evaluation stack and stores it in a the local variable list at <i>index</i> (short form).	13
Stobj	Copies a value of a specified type from the evaluation stack into a supplied memory address.	81
Stsfld	Replaces the value of a static field with a value from the evaluation stack.	80
Sub	Subtracts one value from another and pushes the result onto the evaluation stack.	59
Sub_Ovf	Subtracts one integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DA
Sub_Ovf_Un	Subtracts one unsigned integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.	DB

Switch	Implements a jump table.	45
Tailcall	Performs a postfix method call instruction such that the current method's stack frame is removed before the actual call instruction is executed.	FE 14
Throw	Throws the exception object currently on the evaluation stack.	7A
Unaligned	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.	FE 12
Unbox	Converts the boxed representation of a value type to its unboxed form.	79
Volatile	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.	FE 13
Xor	Computes the bitwise XOR of the top two values on the evaluation stack, pushing the result onto the evaluation stack.	61

What lies beneath?

These are some of the obstacles that you will face when you crack any Assembly; I will try to explain some of them here, for more information on any of them just Google it.

1. Obfuscation:

It's the process of changing the method and classes names to non-readable characters to make harder for us to find the **"IsLicensed"** function, yes; It can make the job harder for you but believe me it's not that hard to trace through the code even when it's obfuscated. The solution here is using bookmarks in reflector to help you keep your sanity when moving between methods and a I suggest a paper and a pen too, Cracking demands patience. No patience, no cracking.

2. Encoded Strings:

This is ugly, It reminds me of old times when I was waiting for W32Dasm to finish disassembling to see if there is any "Invalid Serial Number" in String references, but with encoding strings you won't see any clear text wandering around to tell you what this particular function does.

The most common way to hide strings is encoding them and then saving the entire encoded stream as a binary .net resource, and whenever some string is needed, a function is called to get that string from the encoded stream and it will return the original string, the point that makes this technique weak is that it has to be fast in decoding to keep the program running fast or at least not much slower than it used to do when strings were clear and retrieved quickly without decoding, most decoding functions use byte shifting and reordering to decode strings, and believe me they are easy to find and reverse, as soon as you find the decoder you can reverse it and write your own decoder, this will help you reveal most strings through the code quickly, later I will show you how to reverse the decoding function used in commercial applications.

3. **Strong Name signature:**

A digital signature enables authentication of digital documents, text or data. It is used to prove that the author of the data is the expected one, and that the data has not been tampered with. To create a digital signature, public-key cryptography is used. Typically, a digital signature is created by first hashing the data to be signed into a 160-bit value, which is then encrypted using the unique Private Key held only by the party that is signing the data. Anyone holding the Public Key that corresponds to the Private Key can then use it to authenticate the author of the data and also to ensure that the data has not been altered since it was signed.

This is a primitive technique used to protect .net assemblies from any tampering or modification after the assembly is released to the public. When you start an exe application built with .net, it will be checked by the run-time for string name signature, if it exists then the digital signature is verified, if verification fails then this means that the assembly has been modified after it was released and the runtime will refuse to load it!

How can you defeat this?

You can find detailed explanations about how the Strong Name Signature works and its implementation over the internet. We won't reinvent the wheel, read in the tools section to find the answer.

THE END

Last words:-

That's all for now...

See you soon and I hope you enjoy this tutor.

Best wishes,

tKC

Monday, June 26, 2006

