La Nuova Guida Al Cracking

Corre l'anno 2005 (01/2005) e ne è passato di tempo da quando si è vista l'ultima guida al cracking in italia. Nel frattempo abbiamo attraversato tante cose, nuovi sistemi operativi, nuovi metodi di protezione ecc. Adesso credo che sia giunto il momento di una nuova guida perché si stanno avvicinando grandi cambiamenti. La guida di Xoa è stata scritta quasi 10 anni fa e da allora abbiamo visto come i programmatori sono divenuti sempre più smaliziati e abbiamo visto il proliferarsi di Packers e Crypters. Anche il mondo del cracking è molto cambiato da allora, le crew e i singoli cracker hanno perso totalmente la visibilità. E su ciò qualche considerazione secondo me va fatta, infatti mi sembra interessante riportare qualche spezzone delle mie innumerevoli conversazioni con Xoanon, dato che lui veramente ha potuto vedere l'evolversi della scena. In verità non è una conversazione preparata né tantomeno pensata per esser pubblicata, quindi non è che segua un vero filo logico. I vari - corrispondono a nuova riga.

Xoanon: bah - il reversing è una cosa meccanica è solo il fatto d'averci tempo da buttarci via - non c'è niente di inventivo - purtroppo di sti tempi con i p2p diventà qualcuno nella scena è praticamente impossibile - ai miei tempi si poteva perchè c'era + visibilità per i gruppi - oggi chi vuole qualcosa se lo trova in 5 minuti ... è troppo inflazionato oggi - io son diventato qualcuno nei primi anni 90 - ma era facilissimo - non c'era concorrenza - non è paragonabile assolutamente a oggi - beh c'è troppa inflazione oggi - e è stato già scoperto e fatto tutto quello che c'era da fà - indi - è solo una questione di pazienza nel reversing - la bravura ormai non serve + - visto che è solo un fatto di pazienza ... dei vecchi - dei miei tempi - difatti hanno smesso tutti - il gioco non vale la candela - prima crakkavi per avere gli account sugli ftp 0day - oggi con il p2p non serve + - e poi effettivamente prima diventavi qualcuno ... io quando crakkai il bleem - mi idolatravano - oggi pure penso se crakki starforce - un ti caa nessuno (livornese: caga, ndnt) - non c'è + la cultura del crakkà coi p2p - qualcuno crakka,sì - ma non ha la visibilità di prima - prima per trovà la roba crakkata dovevi essè nella scena - avè gli acc sugli ftp - oggi non c'è + questa cosa - causa p2p - indi la gente ignora tutto quello che c'è dietro ... io mi son fatto mezza scena amiga e tutta la scena pc - oggi la scena come s'intendeva prima non esiste più - prima eri forzato a conoscè la gente che crakkava,se non la conoscevi non c'avevi accesso alla roba - oggi non è + così - beh se vuoi avè un pò di visibilità la cosa che rimane oggi - piglia explorer e trova qualche exploit - e facci un worm - così c'hai visibilità ... e cmq,pure scrivè su frack - ripeto,non c'hai + la visibilità di prima - oggi chiunque apre emule e trova qualunque cosa - la maniera per avè visibilità oggi è fare un worm

In effetti i tempi di Xoanon, Fravia, Kill3xx e persino di personaggi come Pietrek, Russinovich ecc. sono passati. In ogni caso i cambiamenti che stanno avvenendo nell'ultimo periodo sono molto grandi e mi hanno convinto a scrivere una nuova guida perché si sta procedendo in una direzione totalmente diversa da quella vista finora. In questa guida non spiegherò roba che si trova già in migliaia di articoli... No, qua vedremo come avvicinarci al futuro del cracking: il .NET.

Indice

00 - Introduzione
01 - MicroSoft Intermediate Language
02 - Cracking
03 - Debugging
04 - Decompilers
05 - Code Obfuscation
07 - Protection Theory
08 - Conclusioni

Introduzione

In effetti mi pare che gli ambienti di cracking siano ancora molto restii a convertirsi, ma d'altronde il .NET è il futuro. Fra due o tre anni la maggior parte delle applicazioni (e persino alcuni driver) saranno compilati in MSIL (MicroSoft Intermediate Language). Certo questo non significa che dobbiamo disimparare il normale assembly (anzi conviene avvantaggiarsi sui processori sull'ia64), ma come ben presto vedrete vi sarà una sempre maggiore proliferazione di applicativi codati in C#, VB.NET e Managed C++. Senza contare tutti gli altri linguaggi che possono aggiungersi alla tecnologia .NET. Grazie al namespace System.Reflection.Emit, infatti, è possibile creare propri compilatori IL per il linguaggio che vogliamo. Insomma il .NET è multipiattaforma, non dipende né dal processore né dal sistema operativo in uso. Tutto quel che gli occorre è che sia installato il framework tramite il quale funziona. Persino le applicazioni dei palmari e dei telefonini con Windows Mobile installato hanno subito una forte crescita di applicativi .NET (infatti anche io a paragone fatto tra l'Embedded VC++ e il .NET, ho scelto di programmare in .NET per il mio palmare (e appena uscito il compact framework 2.0 sarà veramente molto potente)), ed è facilmente comprensibile il perché: non è necessario compilare il proprio programma per tutti i diversi processori sul mercato (bisogna poi dire che anche sul Symbian è possibile installare un framework .NET). Oltretutto c'è da dire che il .NET farà il suo vero debutto con longhorn e le migliaia di nuove classi che dovrebbe apportare il nuovo sistema operativo. Classi che renderanno il Win32 obsoleto, questo è certo. Come prima cosa quindi prima di passare al cracking, dobbiamo vedere cosa crackare: ovvero l'IL.

MicroSoft Intermediate Language

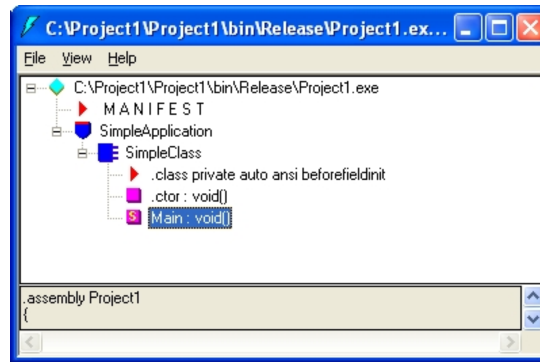
L'IL è un linguaggio intermedio che sta tra il linguaggio nel quale abbiamo programmato e il codice macchina. Generalmente siamo abituati a scrivere un programma, compilarlo e trovarci di fronte a un eseguibile con all'interno codice macchina. Al posto del codice macchina avremmo appunto l'IL. La differenza tra codice macchina e IL è che quest'ultimo non dipende appunto dalla macchina sulla quale viene eseguito dato che si tratta di un'astrazione (pseudo assembly) facilmente convertibile nel codice macchina del processore sul quale viene eseguito. Il .NET usa in maniera molto efficiente un meccanismo di conversione da pseudo-assembly a assembly chiamato JIT (Just In Time) che traduce il codice solo quando è necessario, questo riduce drasticamente il tempo di caricamento, oltretutto una volta convertita una parte di codice non vi sarà bisogno di riconvertirla durante l'esecuzione dell'eseguibile. Se siete interessati a sapere di più riguardo all'architettura .NET vi consiglio un libro apposito (anzi sarebbe il caso, dato che probabilmente capirete a fondo l'IL solo se conoscete anche il framework su cui gira), adesso passeremo direttamente ad occuparci di codice IL. Nonostante il fatto che si possa anche programmare in IL (compilando coll'utility ilasm.exe, fornita dalla MS), noi partiremo disassemblando (è per questo che siamo qui). Tanto poi, programmando o analizzando codice, l'IL s'impara comunque. Però prima di disassemblare ci serve un programma compilato in IL, siccome dobbiamo partire da qualcosa di facile mi sono fatto un piccolo programma in C#. Ecco il codice:

```
using System;

namespace SimpleApplication
{
   class SimpleClass
   {
      static void Main()
      {
      }
   }
}
```

Compiliamolo e otterremo un programma che non fa niente, ma che ci basterà per dare un primo sguardo all'IL. Adesso per disassemblare

possiamo sia usare ildasm.exe (fornito dalla MS e situato nella cartella del SDK) oppure usare il nostro solito IDA. Comincio con lo spiegare ildasm dato che probabilmente è quello per la maggior parte di voi meno familiare. Apriamo il prog compilato con ildasm e troveremo una cosa di questo genere:



Il significato dei vari simboli ve lo pasto direttamente dal MSDN.

Symbol	Meaning
	More info
	Namespace
	Class
	Interface
	Value Class
	Enum
	Method
	Static method
	Field
	Static field
	Event
	Property
	Manifest or a class info item

Se clickiamo sul metodo main due volte otterremo una finestra che ci mostra il suo codice:

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       1 (0x1)
  .maxstack  0
  IL_0000:  ret
} // end of method SimpleClass::Main
```

Tutto ciò che inizia con un "." in IL sta per una direttiva, in questo caso indichiamo un metodo, tutto ciò, invece, che non è preceduto da punti, sono le istruzioni da eseguire. La direttiva entrypoint indica appunto l'entrypoint del codice, è una direttiva che in un assembly può esser usata una sola volta. Se nello stesso assembly la usate più volte e provate a compilare con ilasm.exe, quest'ultimo vi darà errore. I metodi sono delimitati dalle parentesi graffe, ciò nonostante è opportuno concludere il metodo con ret. L'attributo hidebysig nasconde il metodo a classi derivate. Se adesso clickiamo sul triangolino rosso della classe vedremo:

```
.class private auto ansi beforefieldinit SimpleClass
       extends [mscorlib]System.Object
{
} // end of class SimpleClass
```

Da qui vediamo che la nostra classe è un'estensione della classe System.Object. Ciò comporta l'esistenza dovuta del costruttore ctor. Ora prima di continuare una tabella coll'instruction set (framework 1.1) e relativi opcodes.

Instruction	Opcode (Hex)	Short Description
add	58	Adds two values and pushes the result onto the evaluation stack.
add.ovf	D6	Adds two integers, performs an overflow check, and pushes the result onto the evaluation stack.
add.ovf.un	D7	Adds two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.
and	5F	Computes the bitwise AND of two values and pushes the result onto the evalution stack.
arglist	FE 00	Returns an unmanaged pointer to the argument list of the current method.
beq	3B	Transfers control to a target instruction if two values are equal.
beq.s	2E	Transfers control to a target instruction (short form) if two values are equal.
bge	3C	Transfers control to a target instruction if the first value is greater than or equal to the second value.

bge.s	2F	Transfers control to a target instruction (short form) if the first value is greater than or equal to the second value.
bge.un	41	Transfers control to a target instruction if the the first value is greather than the second value, when comparing unsigned integer values or unordered float values.
bge.un.s	34	Transfers control to a target instruction (short form) if if the the first value is greather than the second value, when comparing unsigned integer values or unordered float values.
bgt	3D	Transfers control to a target instruction if the first value is greater than the second value.
bgt.s	30	Transfers control to a target instruction (short form) if the first value is greater than the second value.
bgt.un	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.s	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.
ble	3E	Transfers control to a target instruction if the first value is less than or equal to the second value.
ble.s	31	Transfers control to a target instruction (short form) if the first value is less than or equal to the second value.
ble.un	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.s	36	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.
blt	3F	Transfers control to a target instruction if the first value is less than the second value.
blt.s	32	Transfers control to a target instruction (short form) if the first value is less than the second value.
blt.un	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.s	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.
bne.un	40	Transfers control to a target instruction when two unsigned integer values or unordered float values are not equal.
bne.un.s	33	Transfers control to a target instruction (short form) when two unsigned integer values or unordered float values are not equal.
box	8C	Converts a value type to an object reference (type **O**).
br	38	Unconditionally transfers control to a target instruction.
break	01	Signals the Common Language Infrastructure (CLI) to inform the debugger that a break point has been tripped.
brfalse	39	Transfers control to a target instruction if value is false, a null reference (**Nothing** in Visual Basic), or zero.
brfalse.s	2C	Transfers control to a target instruction if *value* is **false**, a null reference, or zero.
brtrue	3A	Transfers control to a target instruction if *value* is **true**, not null, or non-zero.
brtrue.s	2D	Transfers control to a target instruction (short form) if *value* is **true**, not null, or non-zero.
br.s	2B	Unconditionally transfers control to a target instruction (short form).
call	28	Calls the method indicated by the passed method descriptor.
		Calls the method indicated on the evaluation stack (as

calli	29	a pointer to an entry point) with arguments described by a calling convention.
callvirt	6F	Calls a late-bound method on an object, pushing the return value onto the evaluation stack.
castclass	74	Attempts to cast an object passed by reference to the specified class.
ceq	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.
cgt	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.un	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.
ckfinite	C3	Throws ArithmeticException if value is not a finite number.
clt	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.un	FE 05	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.
conv.i	D3	Converts the value on top of the evaluation stack to **natural int**.
conv.i1	67	Converts the value on top of the evaluation stack to **int8**, then extends (pads) it to **int32**.
conv.i2	68	Converts the value on top of the evaluation stack to **int16**, then extends (pads) it to **int32**.
conv.i4	69	Converts the value on top of the evaluation stack to **int32**.
conv.i8	6A	Converts the value on top of the evaluation stack to **int64**.
conv.ovf.i	D4	Converts the signed value on top of the evaluation stack to signed **natural int**, throwing OverflowException on overflow.
conv.ovf.i1	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.un	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.i2	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.un	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.i4	B7	Converts the signed value on top of the evaluation tack to signed **int32**, throwing OverflowException on overflow.
conv.ovf.i4.un	84	Converts the unsigned value on top of the evaluation stack to signed **int32**, throwing OverflowException on overflow.
conv.ovf.i8	B9	Converts the signed 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 **int64**, throwing OverflowException on overflow.
conv.ovf.i.un	8A	Converts the unsigned value on top of the evaluation stack to signed **natural int**, throwing OverflowException on overflow.
conv.ovf.u	D5	Converts the signed value on top of the evaluation stack to **unsigned natural int**, throwing OverflowException on overflow.
		Converts the signed value on top of the evaluation

conv.ovf.u1	B4	stack to **unsigned int8** and extends it to **int32**, throwing OverflowException on overflow.
conv.ovf.u1.un	86	Converts the unsigned value on top of the evaluation stack to **unsigned int8** and extends it to **int32**, throwing OverflowException on overflow.
conv.ovf.u2	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.un	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.u4	B8	Converts the signed value on top of the evaluation stack to **unsigned int32**, throwing OverflowException on overflow.
conv.ovf.u4.un	88	Converts the unsigned value on top of the evaluation stack to **unsigned int32**, throwing OverflowException on overflow.
conv.ovf.u8	BA	Converts the signed value on top of the evaluation stack to **unsigned int64**, throwing OverflowException on overflow.
conv.ovf.u8.un	89	Converts the unsigned value on top of the evaluation stack to **unsigned int64**, throwing OverflowException on overflow.
conv.ovf.u.un	8B	Converts the unsigned value on top of the evaluation stack to **unsigned natural int**, throwing OverflowException on overflow.
conv.r4	6B	Converts the value on top of the evaluation stack to **float32**.
conv.r8	6C	Converts the value on top of the evaluation stack to **float64**.
conv.r.un	76	Converts the unsigned integer value on top of the evaluation stack to **float32**.
conv.u	E0	Converts the value on top of the evaluation stack to **unsigned natural int**, and extends it to **natural int**.
conv.u1	D2	Converts the value on top of the evaluation stack to **unsigned int8**, and extends it to **int32**.
conv.u2	D1	Converts the value on top of the evaluation stack to **unsigned int16**, and extends it to **int32**.
conv.u4	6D	Converts the value on top of the evaluation stack to **unsigned int32**, and extends it to **int32**.
conv.u8	6E	Converts the value on top of the evaluation stack to **unsigned int64**, and extends it to **int64**.
cpblk	FE 17	Copies a specified number bytes from a source address to a destination address.
cpobj	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**).
div	5B	Divides two values and pushes the result as a floating-point (type **F**) or quotient (type **int32**) onto the evaluation stack.
div.un	5C	Divides two unsigned integer values and pushes the result (**int32**) onto the evaluation stack.
dup	25	Copies the current topmost value on the evaluation stack, and then pushes the copy onto the evaluation stack.
endfilter	FE 11	Transfers control from the **filter** clause of an exception back to the Common Language Infrastructure (CLI) exception handler.
endfinally	DC	Transfers control from the **fault** or **finally** clause of an exception block back to the Common Language Infrastructure (CLI) exception handler.
initblk	FE 18	Initializes a specified block of memory at a specific address to a given size and initial value.
initobj	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.
isinst	75	Tests whether an object reference (type **O**) is an instance of a particular class.
jmp	27	Exits current method and jumps to specified method.
ldarg	FE 09	Loads an argument (referenced by a specified index value) onto the stack.

ldarga	FE 0A	Load an argument address onto the evaluation stack.
ldarga.s	0F	Load an argument address, in short form, onto the evaluation stack.
ldarg.0	02	Loads the argument at index 0 onto the evaluation stack.
ldarg.1	03	Loads the argument at index 1 onto the evaluation stack.
ldarg.2	04	Loads the argument at index 2 onto the evaluation stack.
ldarg.3	05	Loads the argument at index 3 onto the evaluation stack.
ldarg.s	0E	Loads the argument (referenced by a specified short form index) onto the evaluation stack.
ldc.i4	20	Pushes a supplied value of type **int32** onto the evaluation stack as an **int32**.
ldc.i4.0	16	Pushes the integer value of 0 onto the evaluation stack as an **int32**.
ldc.i4.1	17	Pushes the integer value of 1 onto the evaluation stack as an **int32**.
ldc.i4.2	18	Pushes the integer value of 2 onto the evaluation stack as an **int32**.
ldc.i4.3	19	Pushes the integer value of 3 onto the evaluation stack as an **int32**.
ldc.i4.4	1A	Pushes the integer value of 4 onto the evaluation stack as an **int32**.
ldc.i4.5	1B	Pushes the integer value of 5 onto the evaluation stack as an **int32**.
ldc.i4.6	1C	Pushes the integer value of 6 onto the evaluation stack as an **int32**.
ldc.i4.7	1D	Pushes the integer value of 7 onto the evaluation stack as an **int32**.
ldc.i4.8	1E	Pushes the integer value of 8 onto the evaluation stack as an **int32**.
ldc.i4.m1	15	Pushes the integer value of -1 onto the evaluation stack as an **int32**.
ldc.i4.s	1F	Pushes the supplied **int8** value onto the evaluation stack as an **int32**, short form.
ldc.i8	21	Pushes a supplied value of type **int64** onto the evaluation stack as an **int64**.
ldc.r4	22	Pushes a supplied value of type **float32** onto the evaluation stack as type **F** (float).
ldc.r8	23	Pushes a supplied value of type **float64** onto the evaluation stack as type **F** (float).
ldelema	8F	Loads the address of the array element at a specified array index onto the top of the evaluation stack as type **&** (managed pointer).
ldelem.i	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.i1	90	Loads the element with type **int8** at a specified array index onto the top of the evaluation stack as an **int32**.
ldelem.i2	92	Loads the element with type **int16** at a specified array index onto the top of the evaluation stack as an **int32**.
ldelem.i4	94	Loads the element with type **int32** at a specified array index onto the top of the evaluation stack as an **int32**.
ldelem.i8	96	Loads the element with type **int64** at a specified array index onto the top of the evaluation stack as an **int64**.
ldelem.r4	98	Loads the element with type **float32** at a specified array index onto the top of the evaluation stack as type **F** (float).
ldelem.r8	99	Loads the element with type **float64** at a specified array index onto the top of the evaluation stack as type **F** (float).
ldelem.ref	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.u1	91	Loads the element with type **unsigned int8** at a specified array index onto the top of the evaluation

		stack as an **int32**.
ldelem.u2	93	Loads the element with type **unsigned int16** at a specified array index onto the top of the evaluation stack as an **int32**.
ldelem.u4	95	Loads the element with type **unsigned int32** at a specified array index onto the top of the evaluation stack as an **int32**.
ldfld	7B	Finds the value of a field in the object whose reference is currently on the evaluation stack.
ldflda	7C	Finds the address of a field in the object whose reference is currently on the evaluation stack.
ldftn	FE 06	Pushes an unmanaged pointer (type **natural int**) to the native code implementing a specific method onto the evaluation stack.
ldind.i	4D	Loads a value of type **natural int** as a **natural int** onto the evaluation stack indirectly.
ldind.i1	46	Loads a value of type **int8** as an **int32** onto the evaluation stack indirectly.
ldind.i2	48	Loads a value of type **int16** as an **int32** onto the evaluation stack indirectly.
ldind.i4	4A	Loads a value of type **int32** as an **int32** onto the evaluation stack indirectly.
ldind.i8	4C	Loads a value of type **int64** as an **int64** onto the evaluation stack indirectly.
ldind.r4	4E	Loads a value of type **float32** as a type **F** (float) onto the evaluation stack indirectly.
ldind.r8	4F	Loads a value of type **float64** as a type **F** (float) onto the evaluation stack indirectly.
ldind.ref	50	Loads an object reference as a type **O** (object reference) onto the evaluation stack indirectly.
ldind.u1	47	Loads a value of type **unsigned int8** as an **int32** onto the evaluation stack indirectly.
ldind.u2	49	Loads a value of type **unsigned int16** as an **int32** onto the evaluation stack indirectly.
ldind.u4	4B	Loads a value of type **unsigned int32** as an **int32** onto the evaluation stack indirectly.
ldlen	8E	Pushes the number of elements of a zero-based, one-dimensional array onto the evaluation stack.
ldloc	FE 0C	Loads the local variable at a specific index onto the evaluation stack.
ldloca	FE 0D	Loads the address of the local variable at a specific index onto the evaluation stack.
ldloca.s	12	Loads the address of the local variable at a specific index onto the evaluation stack, short form.
ldloc.0	06	Loads the local variable at index 0 onto the evaluation stack.
ldloc.1	07	Loads the local variable at index 1 onto the evaluation stack.
ldloc.2	08	Loads the local variable at index 2 onto the evaluation stack.
ldloc.3	09	Loads the local variable at index 3 onto the evaluation stack.
ldloc.s	11	Loads the local variable at a specific index onto the evaluation stack, short form.
ldnull	14	Pushes a null reference (type **O**) onto the evaluation stack.
ldobj	71	Copies the value type object pointed to by an address to the top of the evaluation stack.
ldsfld	7E	Pushes the value of a static field onto the evaluation stack.
ldsflda	7F	Pushes the address of a static field onto the evaluation stack.
ldstr	72	Pushes a new object reference to a string literal stored in the metadata.
ldtoken	D0	Converts a metadata token to its runtime representation, pushing it onto the evaluation stack.
		Pushes an unmanaged pointer (type **natural int**) to the

ldvirtftn	FE 07	native code implementing a particular virtual method associated with a specified object onto the evaluation stack.
leave	DD	Exits a protected region of code, unconditionally tranferring control to a specific target instruction.
leave.s	DE	Exits a protected region of code, unconditionally tranferring control to a target instruction (short form).
localloc	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.
mkrefany	C6	Pushes a typed reference to an instance of a specific type onto the evaluation stack.
mul	5A	Multiplies two values and pushes the result on the evaluation stack.
mul.ovf	D8	Multiplies two integer values, performs an overflow check, and pushes the result onto the evaluation stack.
mul.ovf.un	D9	Multiplies two unsigned integer values, performs an overflow check, and pushes the result onto the evaluation stack.
neg	65	Negates a value and pushes the result onto the evaluation stack.
newarr	8D	Pushes an object reference to a new zero-based, one-dimensional array whose elements are of a specific type onto the evaluation stack.
newobj	73	Creates a new object or a new instance of a value type, pushing an object reference (type **O**) onto the evaluation stack.
nop	00	Fills space if opcodes are patched. No meaningful operation is performed although a processing cycle can be consumed.
not	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.
or	60	Compute the bitwise complement of the two integer values on top of the stack and pushes the result onto the evaluation stack.
pop	26	Removes the value currently on top of the evaluation stack.
refanytype	FE 1D	Retrieves the type token embedded in a typed reference.
refanyval	C2	Retrieves the address (type **&**) embedded in a typed reference.
rem	5D	Divides two values and pushes the remainder onto the evaluation stack.
rem.un	5E	Divides two unsigned values and pushes the remainder onto the evaluation stack.
ret	2A	Returns from the current method, pushing a return value (if present) from the caller's evaluation stack onto the callee's evaluation stack.
rethrow	FE 1A	Rethrows the current exception.
shl	62	Shifts an integer value to the left (in zeroes) by a specified number of bits, pushing the result onto the evaluation stack.
shr	63	Shifts an integer value (in sign) to the right by a specified number of bits, pushing the result onto the evaluation stack.
shr.un	64	Shifts an unsigned integer value (in zeroes) to the right by a specified number of bits, pushing the result onto the evaluation stack.
sizeof	FE 1C	Pushes the size, in bytes, of a supplied value type onto the evaluation stack.
starg	FE 0B	Stores the value on top of the evaluation stack in the argument slot at a specified index.
starg.s	10	Stores the value on top of the evaluation stack in the argument slot at a specified index, short form.
stelem.i	9B	Replaces the array element at a given index with the **natural int** value on the evaluation stack.
stelem.i1	9C	Replaces the array element at a given index with the **int8** value on the evaluation stack.

stelem.i2	9D	Replaces the array element at a given index with the **int16** value on the evaluation stack.
stelem.i4	9E	Replaces the array element at a given index with the **int32** value on the evaluation stack.
stelem.i8	9F	Replaces the array element at a given index with the **int64** value on the evaluation stack.
stelem.r4	A0	Replaces the array element at a given index with the **float32** value on the evaluation stack.
stelem.r8	A1	Replaces the array element at a given index with the **float64** value on the evaluation stack.
stelem.ref	A2	Replaces the array element at a given index with the object ref value (type **O**) on the evaluation stack.
stfld	7D	Replaces the value stored in the field of an object reference or pointer with a new value.
stind.i	DF	Stores a value of type **natural int** at a supplied address.
stind.i1	52	Stores a value of type **int8** at a supplied address.
stind.i2	53	Stores a value of type **int16** at a supplied address.
stind.i4	54	Stores a value of type **int32** at a supplied address.
stind.i8	55	Stores a value of type **int64** at a supplied address.
stind.r4	56	Stores a value of type **float32** at a supplied address.
stind.r8	57	Stores a value of type **float64** at a supplied address.
stind.ref	51	Stores a object reference value at a supplied address.
stloc	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.0	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.1	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.2	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.3	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.s	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).
stobj	81	Copies a value of a specified type from the evaluation stack into a supplied memory address.
stsfld	80	Replaces the value of a static field with a value from the evaluation stack.
sub	59	Subtracts one value from another and pushes the result onto the evaluation stack.
sub.ovf	DA	Subtracts one integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.
sub.ovf.un	DB	Subtracts one unsigned integer value from another, performs an overflow check, and pushes the result onto the evaluation stack.
switch	45	Implements a jump table.
tail.	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.
throw	7A	Throws the exception object currently on the evaluation stack.
unaligned.	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.
unbox	79	Converts the boxed representation of a value type to its unboxed form.
volatile.	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.
xor	61	Computes the bitwise XOR of the top two values on the evaluation stack, pushing the result onto the evaluation stack.

Se vi servono informazioni aggiuntive riguardo alle specifiche dell'instruction set potete consultare l'MSDN oppure il documento "Partition III CIL.doc" nella cartella "\SDK\v1.1\Tool Developers Guide\docs" del vostro Visual Studio.

Sì, lo so, il linguaggio vi pare ancora molto estraneo, ma vedrete che vi ci abituerete presto, anzi l'IL come vedrete non è nemmeno difficile. E poi considerate un vantaggio: potrete reversare applicazioni per ogni dispositivo che supporta .NET, quindi, oltre al pc, palmari e telefoni, senza dover imparare una nuova sintassi. Come piccolo schemettino di gruppi principali di istruzioni possiamo dire che:

- i vari bgt, ble, bne, br ecc. sono i jump.
- i conv servono per convertire grandezze diverse di valori (tipo da byte a qword ecc.).
- i vari ldc, ldarg, ldelem ecc. corrispondono ai push (prendete per buono che tutte le istruzioni che cominciano con ld sono dei push).
- i vari stloc, starg, stelem ecc corrispondo ai pop (st*).

Per il resto molte istruzioni sono le stesse (quelle logico/matematiche ad esempio (xor, and, mul, sub ecc.)), anche se esistono anche dei falsi amici (il pop in IL non corrisponde esattamente al pop in asm). Ad ogni modo queste cose le vedrete strada facendo. Se avete un dubbio su un'istruzione, ricorrete alla tabella.

Vediamo una nuova applicazione che fa qualcosina in più rispetto a quella che abbiamo visto prima. Per adesso affianco il codice C# al disassemblato per rendere le cose più chiare, fra poco però smetterò di farlo e passeremo direttamente a vedere il codice in IL cercando di capire cosa fa.

```
using System;

namespace SimpleApplication
{
   class SimpleClass
   {
      static void Main()
      {
         int Res = Calc(65535, 3, 5);
         Console.Write("Result: ");
         Console.WriteLine(Res);
         Console.ReadLine();
      }

      static int Calc(int OriginalNumber, Byte a, Byte b)
      {
         return (OriginalNumber >> (a + b));
      }
   }
}
```

Questo codice non fa altro che chiamare la funzione Calc dalla Main, questa funzione ha tre parametri, un intero e due byte, i bit dell'intero vengono shiftati di un numero pari ad a più b. Dopodiché nella main viene stampato il risultato. In questo caso immettendo 65535 (FFFFh) come numero originale e shiftando di 8 bit otterremo 255 (FFh) come risultato. La ReadLine all'interno del codice l'ho messa affinché il programma non si chiuda subito se eseguito ma aspetti un input da parte dell'utente. Sì il codice non serve a nulla ma basta a noi per dare un occhio al linguaggio. Vediamo adesso il codice coll'ildasm. Partiamo dalla main:

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       36 (0x24)
  .maxstack  3
  .locals init (int32 V_0)
  IL_0000:  ldc.i4     0xffff
  IL_0005:  ldc.i4.3
  IL_0006:  ldc.i4.5
  IL_0007:  call       int32 SimpleApplication.SimpleClass::Calc(int32,
                                                                  unsigned int8,
                                                                  unsigned int8)
  IL_000c:  stloc.0
  IL_000d:  ldstr      "Result: "
  IL_0012:  call       void [mscorlib]System.Console::Write(string)
  IL_0017:  ldloc.0
  IL_0018:  call       void [mscorlib]System.Console::WriteLine(int32)
  IL_001d:  call       string [mscorlib]System.Console::ReadLine()
  IL_0022:  pop
  IL_0023:  ret
} // end of method SimpleClass::Main
```

Allora l'IL è stack based come fra poco potremmo ben vedere, .maxstack ci dice che all'interno di questo metodo non vengono caricati sullo stack (virtual stack ovviamente) mai di più di 3 valori. La lista e l'inizializzazione delle variabili locali del metodo viene effettuata tramite .locals init, in questo caso ne abbiamo una sola (Res che è un Int32). Analizziamo la riga di codice che immediatamente segue.

```
  IL_0000:  ldc.i4     0xffff
```

Questa istruzione pusha un Int32 sull'evaluation stack, sarebbe il nostro 65535. A seguire vi sono le istruzioni ldc.i4.3 e ldc.i4.5 che pushano i valori 3 e 5 (se il numero è piccolo (tra 0 e 8) è possibile usare ldc.i4 seguito da un punto e il numero da pushare, ricordate che non è che si può fare per ogni valore ciò). Come potete notare lo stack nell'IL non è come quello a cui siamo abituati, ovvero LIFO (Last In, First Out). Nell'IL i parametri vengono pushati nell'ordine che la funzione prevede.

```
  IL_0007:  call       int32 SimpleApplication.SimpleClass::Calc(int32,
                                                                  unsigned int8,
                                                                  unsigned int8)
```

Viene chiamata la funzione. L'ildasm ci fornisce anche namespace e classe del metodo chiamato. Vediamo poi l'istruzione:

```
  IL_000c:  stloc.0
```

Questa istruzione poppa (da pop) il valore corrente sulla cima dello stack e lo mette nella lista delle variabili locali all'index 0. In pratica mette il valore di ritorno della funzione che sta in cima allo stack dentro a Res (variabile locale).

```
  IL_000d:  ldstr      "Result: "
```

Pusha sullo stack la stringa "Result: ". In verità pusha un reference alla stringa che si trova nella metadata dell'eseguibile.

```
  IL_0012:  call       void [mscorlib]System.Console::Write(string)
```

Chiama l'overload per stringhe della funzione Write.

```
  IL_0017:  ldloc.0
```

Pusha la variabile locale a index 0 (Res) sullo stack.

```
  IL_0018:  call       void [mscorlib]System.Console::WriteLine(int32)
```

Chiama l'overload della funzione WriteLine per interi a 32 bit. Dopodiché chiama la ReadLine.

```
  IL_0022:  pop
```

Rimuove il valore corrente in cima allo stack (sarebbe il valore di ritorno di ReadLine che in ogni caso non mettiamo in nessuna variabile, quindi basta toglierlo di mezzo con pop senza usare stloc). Segue il ret che conclude il metodo. Bene, adesso passiamo alla funzione Calc.

```
.method private hidebysig static int32  Calc(int32 OriginalNumber,
                                             unsigned int8 a,
                                             unsigned int8 b) cil managed
{
  // Code size       9 (0x9)
  .maxstack  3
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  add
  IL_0004:  ldc.i4.s   31
  IL_0006:  and
  IL_0007:  shr
  IL_0008:  ret
} // end of method SimpleClass::Calc
```

Dove:

```
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
```

Caricano sullo stack i tre argomenti passati al metodo. Il numero che segue ldarg specifica la posizione dell'argomento. Il primo è OriginalNumber (0) seguono a e b.

```
  IL_0003:  add
```

Somma i due valori in cima allo stack, in questo caso a e b, e pusha il risulato in cima allo stack. Dopo questa operazione il nostro stack avrà questo aspetto:

```
OriginalNumber
ResultOfAddBetweenA_And_B     ; cima dello stack
```

Poi:

```
  IL_0004:  ldc.i4.s   31
```

Pusha sullo stack in forma di int32 il valore di 8bit che segue l'istruzione, in questo caso 31d (ildasm ci mostra un numero decimale, attenzione). Quindi il nostro stack adesso avrà questo aspetto:

```
OriginalNumber
ResultOfAddBetweenA_And_B
NumberOf31Decimal             ; cima dello stack
```

Quindi:

```
  IL_0006:  and
```

questo and viene effettuato tra la somma di a-b e il numero 31. In pratica questo and serve ad assicurarsi che il nostro numero non superi 31. Vi chiederete come mai. È presto detto, l'and fa in modo che possiamo usare lo shift anche come rotate. Se per esempio vogliamo shiftare di 32 posizioni, lui fa l'and con 31 e torna 0, quindi shifta di 0, dato che col rotate shiftando di 32 posizioni ci ritroveremmo il numero non modificato in mano. Ad ogni modo, essendo il nostro numero 8, esso resta come è e ci ritroviamo con questo stack:

```
OriginalNumber
NumberOf8                    ; cima dello stack
```

```
  IL_0007:  shr
```

Shifta a destra l'original number di 8 posizioni e pusha il risultato sullo stack (essendo esso il valore di ritorno segue il ret. Insomma come abbiamo già visto nel main il valore di ritorno di un metodo si trova in cima allo stack e non in eax (vabbe' ovvio non abbiamo registri) come siamo abituati.

Siccome so che limitarsi ad analizzare codice non è molto divertente, iniziamo col cracking. Tanto le istruzioni che ancora non avete

visto le imparerete via via. Per ora basta che siate entrati nella logica dell'Intermediate Language.

Cracking

Partiamo immediatamente da un esempio facilissimo. Stavolta non vi faccio vedere prima il codice, dato che adesso siamo nel paragrafo di cracking e poi l'esempio è veramente banale. Dato che tutto il codice sta nel main vi incollo solo quello:

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       58 (0x3a)
  .maxstack  2
  .locals init (string V_0,
           int32 V_1)
  IL_0000:  call       string [mscorlib]System.Console::ReadLine()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  call       int32 [mscorlib]System.Convert::ToInt32(string)
  IL_000c:  stloc.1
  IL_000d:  ldloc.1
  IL_000e:  ldc.i4     0x29a
  IL_0013:  xor
  IL_0014:  stloc.1
  IL_0015:  ldloc.1
  IL_0016:  ldc.i4     0x539
  IL_001b:  beq.s      IL_0029
  IL_001d:  ldstr      "Invalid Serial Number"
  IL_0022:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0027:  br.s       IL_0033
  IL_0029:  ldstr      "Thank You For Registering"
  IL_002e:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0033:  call       string [mscorlib]System.Console::ReadLine()
  IL_0038:  pop
  IL_0039:  ret
} // end of method SimpleClass::Main
```

Allora:

```
  IL_0000:  call       string [mscorlib]System.Console::ReadLine()
```

Ci chiede di immettere una stringa.

```
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
```

Prende la stringa (valore di ritorno della funzione ReadLine) e la pusha sullo stack.

```
  IL_0007:  call       int32 [mscorlib]System.Convert::ToInt32(string)
```

Converte la stringa in un intero a 32bit.

```
  IL_000c:  stloc.1
  IL_000d:  ldloc.1
```

Prende il valore di ritorno, ovvero il numero convertito e lo pusha sullo stack.

```
  IL_000e:  ldc.i4     0x29a
```

pusha sullo stack 29Ah (666d).

```
  IL_0013:  xor
```

Fa lo xor tra 666d e il numero da noi immesso. Ci ritroviamo ovviamente il risultato sullo stack.

```
  IL_0014:  stloc.1
  IL_0015:  ldloc.1
```

Prende il risultato dallo stack (mettendolo nella stessa variabile del numero immesso) e lo pusha sullo stack.

```
  IL_0016:  ldc.i4     0x539
```

Pusha sullo stack il valore 539h (1337d).

```
  IL_001b:  beq.s      IL_0029
```

Eccoci arrivati al salto condizionale. Questa istruzione salta all'istruzione all'offset IL_0029 se i due valori sullo stack sono uguali. Se il salto non viene eseguito (ovvero se i valori non coincidono) ci troviamo di fronte questo codice:

```
  IL_001d:  ldstr      "Invalid Serial Number"
  IL_0022:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0027:  br.s       IL_0033
```

ldstr e call mi sembrano chiari. br.s però non l'avete ancora visto, si tratta del salto incondizionale (jmp) che ci porta al termine del programma. Se invece i valori coincidono e il salto viene eseguito, ci troviamo qua:

```
  IL_0029:  ldstr      "Thank You For Registering"
  IL_002e:  call       void [mscorlib]System.Console::WriteLine(string)
```

Ovvero il programma risulta registrato. Ora, è chiaro che l'algo è molto banale:

```
If (Serial Xor 666) = 1337 Then
```

E la risoluzione è evidente, basta inserire come seriale 1955. Proviamo però adesso a crackare il programma, cioè modificare le

istruzioni, visto che tecnicamente per averlo crackato è quello che ci manca. Per fare tutto ciò sarebbe però utile che il nostro amatissimo disassembler ci desse anche gli opcode delle varie istruzioni (nonché l'indirizzo da dove inizia il metodo in questione). Niente di più semplice, è sufficiente andare sul menu dell'ildasm sotto la voce view e clickare su "Show Bytes". Riapriamo adesso il codice del main:

```
.method private hidebysig static void  Main() cil managed
// SIG: 00 00 01
{
  .entrypoint
  // Method begins at RVA 0x2050
  // Code size       58 (0x3a)
  .maxstack  2
  .locals init (string V_0,
           int32 V_1)
  IL_0000:  /* 28   | (0A)000001       */ call       string [mscorlib]System.Console::ReadLine()
  IL_0005:  /* 0A   |                  */ stloc.0
  IL_0006:  /* 06   |                  */ ldloc.0
  IL_0007:  /* 28   | (0A)000002       */ call       int32 [mscorlib]System.Convert::ToInt32(string)
  IL_000c:  /* 0B   |                  */ stloc.1
  IL_000d:  /* 07   |                  */ ldloc.1
  IL_000e:  /* 20   | 9A020000         */ ldc.i4     0x29a
  IL_0013:  /* 61   |                  */ xor
  IL_0014:  /* 0B   |                  */ stloc.1
  IL_0015:  /* 07   |                  */ ldloc.1
  IL_0016:  /* 20   | 39050000         */ ldc.i4     0x539
  IL_001b:  /* 2E   | 0C               */ beq.s      IL_0029
  IL_001d:  /* 72   | (70)000001       */ ldstr      "Invalid Serial Number"
  IL_0022:  /* 28   | (0A)000003       */ call       void [mscorlib]System.Console::WriteLine(string)
  IL_0027:  /* 2B   | 0A               */ br.s       IL_0033
  IL_0029:  /* 72   | (70)00002D       */ ldstr      "Thank You For Registering"
  IL_002e:  /* 28   | (0A)000003       */ call       void [mscorlib]System.Console::WriteLine(string)
  IL_0033:  /* 28   | (0A)000001       */ call       string [mscorlib]System.Console::ReadLine()
  IL_0038:  /* 26   |                  */ pop
  IL_0039:  /* 2A   |                  */ ret
} // end of method SimpleClass::Main
```

Come ci dice il disassembler sotto la direttiva .entrypoint il metodo inizia al RVA 2050h, che nel nostro caso corrisponde al file offset 1050h. Il salto condizionale da patchare è:

```
IL_001b:  /* 2E   | 0C               */ beq.s      IL_0029
```

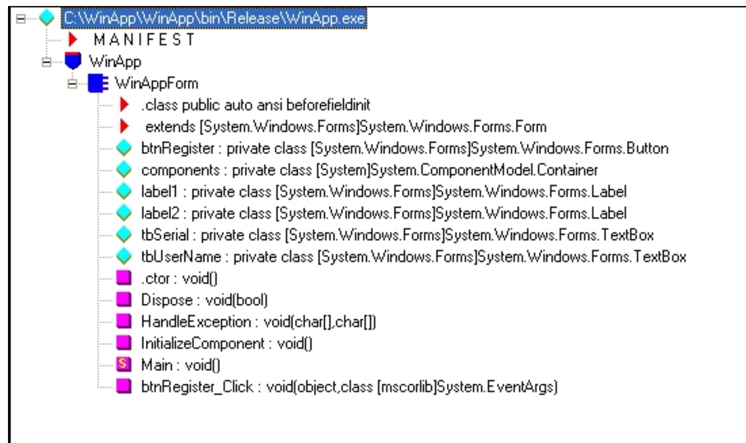
E lo possiamo invertire, oppure convertire in salto incondizionale. In verità in IL non ha molto senso invertire il salto dato che gli opcode hanno la stessa lunghezza, quindi è meglio renderlo incondizionale. Per fare ciò dobbiamo sostituire l'opcode 2Eh con 2Bh. Però Alt, in IL le cose sono un po' diverse dall'assembly, e dobbiamo fare alcune considerazioni in più. Queste considerazioni riguardano lo stack. Noi siamo abituati al cmp che non fa uso di alcuno stack, ma in IL i valori da confrontare sono sullo stack, quindi se trasformiamo un salto condizionale in uno incondizionale (che ovviamente non prenderà nessun valore dallo stack) dobbiamo assicurarci di non impegnare lo stack. Quindi:

```
  IL_0015:  /* 07   |                  */ ldloc.1
  IL_0016:  /* 20   | 39050000         */ ldc.i4     0x539
  IL_001b:  /* 2E   | 0C               */ beq.s      IL_0029
```

È necessario patchare col nop (00h) i due ld e poi rimpiazzare 2Eh con 2Bh. Prendiamo un hex editor e patchiamo, fatto ciò ci ritroviamo di fronte questo codice:

```
.method private hidebysig static void  Main() cil managed
// SIG: 00 00 01
{
  .entrypoint
  // Method begins at RVA 0x2050
  // Code size       58 (0x3a)
  .maxstack  2
  .locals init (string V_0,
           int32 V_1)
  IL_0000:  /* 28   | (0A)000001       */ call       string [mscorlib]System.Console::ReadLine()
  IL_0005:  /* 0A   |                  */ stloc.0
  IL_0006:  /* 06   |                  */ ldloc.0
  IL_0007:  /* 28   | (0A)000002       */ call       int32 [mscorlib]System.Convert::ToInt32(string)
  IL_000c:  /* 0B   |                  */ stloc.1
  IL_000d:  /* 07   |                  */ ldloc.1
  IL_000e:  /* 20   | 9A020000         */ ldc.i4     0x29a
  IL_0013:  /* 61   |                  */ xor
  IL_0014:  /* 0B   |                  */ stloc.1
  IL_0015:  /* 00   |                  */ nop
  IL_0016:  /* 00   |                  */ nop
  IL_0017:  /* 00   |                  */ nop
  IL_0018:  /* 00   |                  */ nop
  IL_0019:  /* 00   |                  */ nop
  IL_001a:  /* 00   |                  */ nop
  IL_001b:  /* 2B   | 0C               */ br.s       IL_0029
  IL_001d:  /* 72   | (70)000001       */ ldstr      "Invalid Serial Number"
  IL_0022:  /* 28   | (0A)000003       */ call       void [mscorlib]System.Console::WriteLine(string)
  IL_0027:  /* 2B   | 0A               */ br.s       IL_0033
  IL_0029:  /* 72   | (70)00002D       */ ldstr      "Thank You For Registering"
  IL_002e:  /* 28   | (0A)000003       */ call       void [mscorlib]System.Console::WriteLine(string)
  IL_0033:  /* 28   | (0A)000001       */ call       string [mscorlib]System.Console::ReadLine()
  IL_0038:  /* 26   |                  */ pop
  IL_0039:  /* 2A   |                  */ ret
} // end of method SimpleClass::Main
```

Adesso il programma, se eseguito, si registra con qualsiasi valore che immettiamo. Adesso facciamo il reversing di qualcosa di più complesso (a livello di IL s'intende), tanto patchare lo sappiamo già fare. Stavolta si lavora su un programma con finestre. Apriamo l'ildasm e ci troviamo di fronte a questo:



In questo caso, come vedete ho messo solo una window, due label, due text box (user name, serial) e un bottone per registrarsi. Si parte dal codice del bottone che sarebbe btnRegister_Click. Per verificare che sia effettivamente quella la routine che si occupa del click del bottone, è sufficiente guardarsi il codice nella routine InitializeComponent (routine che vi è per ogni window, infatti li vengono settati gli attributi della window e delle sue child). Quando arrivate a questa routine non spaventatevi troppo, è molto codice (più child ha la window, più codice vi è), ma dovreste trovare velocemente cosa cercate.

```
  IL_01c4:  ldstr      "Register"
  IL_01c9:  callvirt   instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
  IL_01ce:  ldarg.0
  IL_01cf:  ldfld      class [System.Windows.Forms]System.Windows.Forms.Button WinApp.WinAppForm::btnRegister
  IL_01d4:  ldarg.0
  IL_01d5:  ldftn      instance void WinApp.WinAppForm::btnRegister_Click(object,
                                                                       class [mscorlib]System.EventArgs)
  IL_01db:  newobj     instance void [mscorlib]System.EventHandler::.ctor(object,
                                                                       native int)
  IL_01e0:  callvirt   instance void [System.Windows.Forms]System.Windows.Forms.Control::add_Click(class
[mscorlib]System.EventHandler)
```

Viene creato un EventHandler dalla funzione btnRegister_Click e settato come event handler per quel determinato bottone. In effetti se avete già programmato con .NET siete molto avvantaggiati perché dovreste sapere come vengono inizializzate le finestre. Andiamo al codice del click:

```
.method private hidebysig instance void  btnRegister_Click(object sender,
                                                           class [mscorlib]System.EventArgs e) cil managed
{
  // Code size       216 (0xd8)
  .maxstack  3
  .locals init (char[] V_0,
           char[] V_1,
           char V_2,
           int32 V_3,
           int32 V_4,
           int32 V_5)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbUserName
  IL_0006:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_000b:  callvirt   instance int32 [mscorlib]System.String::get_Length()
  IL_0010:  ldc.i4.6
  IL_0011:  blt.s      IL_0027
  IL_0013:  ldarg.0
  IL_0014:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbSerial
  IL_0019:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_001e:  callvirt   instance int32 [mscorlib]System.String::get_Length()
  IL_0023:  ldc.i4.s   14
  IL_0025:  beq.s      IL_0028
  IL_0027:  ret
  IL_0028:  ldarg.0
  IL_0029:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbUserName
  IL_002e:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_0033:  callvirt   instance char[] [mscorlib]System.String::ToCharArray()
  IL_0038:  stloc.0
  IL_0039:  ldarg.0
  IL_003a:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbSerial
  IL_003f:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_0044:  callvirt   instance char[] [mscorlib]System.String::ToCharArray()
  IL_0049:  stloc.1
  .try
  {
    IL_004a:  ldloc.0
    IL_004b:  ldc.i4.0
    IL_004c:  ldelem.u2
    IL_004d:  stloc.2
```

```
    IL_004e:  ldc.i4.3
    IL_004f:  ldloc.0
    IL_0050:  ldc.i4.0
    IL_0051:  ldelem.u2
    IL_0052:  ldc.i4.2
    IL_0053:  mul
    IL_0054:  ldloc.2
    IL_0055:  sub
    IL_0056:  ldloc.2
    IL_0057:  div
    IL_0058:  ldc.i4.1
    IL_0059:  sub
    IL_005a:  div
    IL_005b:  pop
    IL_005c:  ldc.i4.0
    IL_005d:  stloc.3
    IL_005e:  ldc.i4.0
    IL_005f:  stloc.s    V_4
    IL_0061:  br.s       IL_0070
    IL_0063:  ldloc.3
    IL_0064:  ldloc.0
    IL_0065:  ldloc.s    V_4
    IL_0067:  ldelem.u2
    IL_0068:  add
    IL_0069:  stloc.3
    IL_006a:  ldloc.s    V_4
    IL_006c:  ldc.i4.1
    IL_006d:  add
    IL_006e:  stloc.s    V_4
    IL_0070:  ldloc.s    V_4
    IL_0072:  ldarg.0
    IL_0073:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbUserName
    IL_0078:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
    IL_007d:  callvirt   instance int32 [mscorlib]System.String::get_Length()
    IL_0082:  blt.s      IL_0063
    IL_0084:  ldc.i4.0
    IL_0085:  stloc.s    V_5
    IL_0087:  br.s       IL_0096
    IL_0089:  ldloc.3
    IL_008a:  ldloc.1
    IL_008b:  ldloc.s    V_5
    IL_008d:  ldelem.u2
    IL_008e:  add
    IL_008f:  stloc.3
    IL_0090:  ldloc.s    V_5
    IL_0092:  ldc.i4.1
    IL_0093:  add
    IL_0094:  stloc.s    V_5
    IL_0096:  ldloc.s    V_5
    IL_0098:  ldarg.0
    IL_0099:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbSerial
    IL_009e:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
    IL_00a3:  callvirt   instance int32 [mscorlib]System.String::get_Length()
    IL_00a8:  blt.s      IL_0089
    IL_00aa:  ldloc.3
    IL_00ab:  ldc.i4     0x1337
    IL_00b0:  bne.un.s   IL_00bf
    IL_00b2:  ldstr      "Thank You For Registering"
    IL_00b7:  call       valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
[System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string)
    IL_00bc:  pop
    IL_00bd:  br.s       IL_00ca
    IL_00bf:  ldstr      "Invalid Name/Serial"
    IL_00c4:  call       valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
[System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string)
    IL_00c9:  pop
    IL_00ca:  leave.s    IL_00d7
  }  // end .try
  catch [mscorlib]System.Exception
  {
    IL_00cc:  pop
    IL_00cd:  ldarg.0
    IL_00ce:  ldloc.0
    IL_00cf:  ldloc.1
    IL_00d0:  call       instance void WinApp.WinAppForm::HandleException(char[],
                                                                 char[])
    IL_00d5:  leave.s    IL_00d7
  }  // end handler
  IL_00d7:  ret
} // end of method WinAppForm::btnRegister_Click
```

Non svenite, procediamo passo passo.

```
  IL_0000:  ldarg.0
  IL_0001:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbUserName
  IL_0006:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_000b:  callvirt   instance int32 [mscorlib]System.String::get_Length()
```

Prende la lunghezza dello UserName (ricordate che dopo la get_Length il valore di ritorno sta in cima allo stack).

```
  IL_0010:  ldc.i4.6
  IL_0011:  blt.s      IL_0027
```

Confronta la lunghezza con 6. Ovvero salta a IL_0027 se la lunghezza dello user name è minore di 6. Alla locazione IL_0027 troviamo un ret, quindi il nostro user name non deve essere minore di 6 caratteri altrimenti usciamo dalla funzione.

```
  IL_0013:  ldarg.0
  IL_0014:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbSerial
  IL_0019:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_001e:  callvirt   instance int32 [mscorlib]System.String::get_Length()
```

Prende la lunghezza del seriale.

```
  IL_0023:  ldc.i4.s   14
  IL_0025:  beq.s      IL_0028
  IL_0027:  ret
```

Se uguale a 14d, salta a IL_0028. Altrimenti esce dalla funzione.

```
  IL_0028:  ldarg.0
  IL_0029:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbUserName
  IL_002e:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_0033:  callvirt   instance char[] [mscorlib]System.String::ToCharArray()
  IL_0038:  stloc.0
```

Converte la stringa dello user name in un array di char e lo mette nella lista delle variabili locali all'index 0. Se guardiamo le variabili locali:

```
.locals init (char[] V_0,
           char[] V_1,
           char V_2,
           int32 V_3,
           int32 V_4,
           int32 V_5)
```

Come vedete le prime due sono array di char.

```
  IL_0039:  ldarg.0
  IL_003a:  ldfld      class [System.Windows.Forms]System.Windows.Forms.TextBox WinApp.WinAppForm::tbSerial
  IL_003f:  callvirt   instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
  IL_0044:  callvirt   instance char[] [mscorlib]System.String::ToCharArray()
  IL_0049:  stloc.1
```

Prende la stringa del seriale, la converte in array di char e la mette nella seconda variabile locale. Quindi riassumendo char[] V_0 contiene lo user name, mentre char[] V_1 il seriale.

```
  .try
  {
```

Si apre un blocco try, ovvero vi è un exception handler. Ovviamente a qualsiasi reverser già qua sorgerebbe un dubbio.

```
    IL_004a:  ldloc.0
    IL_004b:  ldc.i4.0
    IL_004c:  ldelem.u2
```

Allora, pusha sullo stack l'array di char dello user name, pusha l'index 0, pusha sullo stack l'elemento dell'array al dato index. In pratica, parlando in pseudo-codice. sarebbe un push UserNameArray[Index]. Adesso in cima allo stack abbiamo il primo carattere di questo array.

```
    IL_004d:  stloc.2
```

Ficca il carattere nella terza variabile locale (un char appunto).

```
    IL_004e:  ldc.i4.3
```

Pusha il valore 3.

```
    IL_004f:  ldloc.0
    IL_0050:  ldc.i4.0
    IL_0051:  ldelem.u2
```

Pusha sullo stack il primo carattere dello user name.

```
    IL_0052:  ldc.i4.2
    IL_0053:  mul
```

Lo moltiplica per due.

```
    IL_0054:  ldloc.2
    IL_0055:  sub
```

Gli toglie metà (ricordate che la seconda variabile locale contiene il valore originale del carattere).

```
    IL_0056:  ldloc.2
    IL_0057:  div
```

Lo divide per se stesso. Quindi il risultato di div adesso sarà 1, valore che quindi avremo sullo stack.

```
    IL_0058:  ldc.i4.1
    IL_0059:  sub
```

Sottrae 1 a 1.

```
    IL_005a:  div
```

Compie la divisione con il risultato della sottrazione sul valore 3 precdentemente pushato. Quindi alla fine dei conti avremo un 3 / 0. Che ovviamente porta a un'eccezione. Andiamo a vedere il catch:

```
  catch [mscorlib]System.Exception
  {
    IL_00cc:  pop
    IL_00cd:  ldarg.0
    IL_00ce:  ldloc.0
    IL_00cf:  ldloc.1
    IL_00d0:  call       instance void WinApp.WinAppForm::HandleException(char[],
                                                                    char[])
    IL_00d5:  leave.s    IL_00d7
  }  // end handler
```

Troviamo subito il catch, dato che ve n'è uno solo. Altrimenti probabilmente avreste dovuto cercare il catch riguardante ExceptionDivideByZero, ma in assenza di quello viene chiamato il catch generico (Exception). Nel catch vengono pushati, con i due ldloc, user name e seriale e passati come argomenti alla funzione HandleException. Ovviamente è là che dovremo andare a cercare la routine di controllo del seriale. Questo ovviamente significa che il codice dopo il secondo div non viene nemmeno eseguito, messo lì solo per confondere le idee: nel caso qualcuno volesse tentare un patch senza analizzare il codice. Va da sé che questo codice, a dire il vero, non vuole fregare proprio nessuno, volevo giusto dare un'idea della gestione delle eccezioni. Prima di procedere devo farvi notare una cosa importante, se vedete alla funzione HandleException vengono passati tre argomenti, mentre la funzione risulta averne solo due. Questo perché alle funzioni come primo argomento viene sempre passata l'istanza della classe. Quindi il primo argomento vero e proprio di una funzione sta all'index 1 e non 0. Ok, dopo questo breve excursus, eccovi il codice della HandleException.

```
.method private hidebysig instance void  HandleException(char[] a,
                                                         char[] b) cil managed
{
  // Code size       220 (0xdc)
  .maxstack  5
  .locals init (char[] V_0,
           int32 V_1,
           int32 V_2,
           int32 V_3,
           int32 V_4,
           int32 V_5,
           int32 V_6,
           int32 V_7)
  IL_0000:  ldarg.2
  IL_0001:  ldc.i4.4
  IL_0002:  ldelem.u2
  IL_0003:  ldarg.2
  IL_0004:  ldc.i4.s   9
  IL_0006:  ldelem.u2
  IL_0007:  bne.un.s   IL_0010
  IL_0009:  ldarg.2
  IL_000a:  ldc.i4.4
  IL_000b:  ldelem.u2
  IL_000c:  ldc.i4.s   45
  IL_000e:  beq.s      IL_0011
  IL_0010:  ret
  IL_0011:  ldc.i4.4
  IL_0012:  newarr     [mscorlib]System.Char
  IL_0017:  stloc.0
  IL_0018:  ldc.i4.0
  IL_0019:  stloc.1
  IL_001a:  br.s       IL_0026
  IL_001c:  ldloc.0
  IL_001d:  ldloc.1
  IL_001e:  ldarg.2
  IL_001f:  ldloc.1
  IL_0020:  ldelem.u2
  IL_0021:  stelem.i2
  IL_0022:  ldloc.1
```

```
IL_0023:  ldc.i4.1
IL_0024:  add
IL_0025:  stloc.1
IL_0026:  ldloc.1
IL_0027:  ldc.i4.4
IL_0028:  blt.s      IL_001c
IL_002a:  ldloc.0
IL_002b:  newobj     instance void [mscorlib]System.String::.ctor(char[])
IL_0030:  ldc.i4.s   16
IL_0032:  call       int32 [mscorlib]System.Convert::ToInt32(string,
                                                              int32)
IL_0037:  stloc.2
IL_0038:  ldloc.2
IL_0039:  brtrue.s   IL_003c
IL_003b:  ret
IL_003c:  ldc.i4.0
IL_003d:  stloc.3
IL_003e:  br.s       IL_004a
IL_0040:  ldloc.2
IL_0041:  ldarg.1
IL_0042:  ldloc.3
IL_0043:  ldelem.u2
IL_0044:  sub
IL_0045:  stloc.2
IL_0046:  ldloc.3
IL_0047:  ldc.i4.1
IL_0048:  add
IL_0049:  stloc.3
IL_004a:  ldloc.3
IL_004b:  ldarg.1
IL_004c:  ldlen
IL_004d:  conv.i4
IL_004e:  blt.s      IL_0040
IL_0050:  ldc.i4.0
IL_0051:  stloc.s    V_4
IL_0053:  br.s       IL_0065
IL_0055:  ldloc.0
IL_0056:  ldloc.s    V_4
IL_0058:  ldarg.2
IL_0059:  ldc.i4.5
IL_005a:  ldloc.s    V_4
IL_005c:  add
IL_005d:  ldelem.u2
IL_005e:  stelem.i2
IL_005f:  ldloc.s    V_4
IL_0061:  ldc.i4.1
IL_0062:  add
IL_0063:  stloc.s    V_4
IL_0065:  ldloc.s    V_4
IL_0067:  ldc.i4.4
IL_0068:  blt.s      IL_0055
IL_006a:  ldloc.0
IL_006b:  newobj     instance void [mscorlib]System.String::.ctor(char[])
IL_0070:  ldc.i4.s   16
IL_0072:  call       int32 [mscorlib]System.Convert::ToInt32(string,
                                                              int32)
IL_0077:  stloc.s    V_5
IL_0079:  ldloc.s    V_5
IL_007b:  brtrue.s   IL_007e
IL_007d:  ret
IL_007e:  ldloc.s    V_5
IL_0080:  ldc.i4     0xf000
IL_0085:  sub
IL_0086:  ldc.i4     0x666
IL_008b:  xor
IL_008c:  ldc.i4.2
IL_008d:  mul
IL_008e:  ldc.i4.1
IL_008f:  add
IL_0090:  stloc.s    V_5
IL_0092:  ldc.i4.0
IL_0093:  stloc.s    V_6
IL_0095:  br.s       IL_00a8
IL_0097:  ldloc.0
IL_0098:  ldloc.s    V_6
IL_009a:  ldarg.2
IL_009b:  ldc.i4.s   10
IL_009d:  ldloc.s    V_6
IL_009f:  add
IL_00a0:  ldelem.u2
IL_00a1:  stelem.i2
IL_00a2:  ldloc.s    V_6
IL_00a4:  ldc.i4.1
```

```
  IL_00a5:  add
  IL_00a6:  stloc.s    V_6
  IL_00a8:  ldloc.s    V_6
  IL_00aa:  ldc.i4.4
  IL_00ab:  blt.s      IL_0097
  IL_00ad:  ldloc.0
  IL_00ae:  newobj     instance void [mscorlib]System.String::.ctor(char[])
  IL_00b3:  ldc.i4.8
  IL_00b4:  call       int32 [mscorlib]System.Convert::ToInt32(string,
                                                                int32)
  IL_00b9:  stloc.s    V_7
  IL_00bb:  ldloc.2
  IL_00bc:  brtrue.s   IL_00db
  IL_00be:  ldloc.s    V_5
  IL_00c0:  ldc.i4     0x1337
  IL_00c5:  bne.un.s   IL_00db
  IL_00c7:  ldloc.s    V_7
  IL_00c9:  ldc.i4     0xad4
  IL_00ce:  bne.un.s   IL_00db
  IL_00d0:  ldstr      "Registered"
  IL_00d5:  call       valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
[System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string)
  IL_00da:  pop
  IL_00db:  ret
} // end of method WinAppForm::HandleException
```

Comincio l'analisi:

```
  IL_0000:  ldarg.2
  IL_0001:  ldc.i4.4
  IL_0002:  ldelem.u2
```

Pusha sullo stack il quinto carattere dell'array del seriale (push Serial[4]).

```
  IL_0003:  ldarg.2
  IL_0004:  ldc.i4.s   9
  IL_0006:  ldelem.u2
```

Pusha sullo stack il decimo carattere dell'array del seriale (push Serial[9]).

```
  IL_0007:  bne.un.s   IL_0010
```

Se i due caratteri non sono uguali esce dalla funzione.

```
  IL_0009:  ldarg.2
  IL_000a:  ldc.i4.4
  IL_000b:  ldelem.u2
```

Ri-pusha sullo stack il quinto carattere dell'array del seriale

```
  IL_000c:  ldc.i4.s   45
  IL_000e:  beq.s      IL_0011
```

Lo confronta con 45d ("-"), se non è uguale, non salta e esce dalla funzione. In pratica finora abbiamo visto codice che controlla la sintassi del seriale, che dovrà avere questa forma: XXXX-XXXX-XXXX.

```
  IL_0011:  ldc.i4.4
```

Pusa il valore 4 sullo stack.

```
  IL_0012:  newarr     [mscorlib]System.Char
```

Crea un array di char, in questo caso la dimensione dell'array è 4 dato il valore in cima allo stack.

```
  IL_0017:  stloc.0
```

L'array corrisponde alla prima variabile locale.

```
  IL_0018:  ldc.i4.0
  IL_0019:  stloc.1
```

Seconda variabile locale = 0.

```
  IL_001a:  br.s       IL_0026
```

Salta...

```
  IL_0026:  ldloc.1
  IL_0027:  ldc.i4.4
  IL_0028:  blt.s      IL_001c
```

Confronta la seconda variabile locale con 4, se minore, torna indietro. In pratica questo codice corrisponde a un ciclo for.

```
  IL_001c:  ldloc.0
  IL_001d:  ldloc.1
  IL_001e:  ldarg.2
  IL_001f:  ldloc.1
  IL_0020:  ldelem.u2
  IL_0021:  stelem.i2
```

Copia il carattere del seriale all'index rappresentato dalla seconda variabile locale nell'array creato, sempre allo stesso index (in pratica NewArray[Index] = Serial[Index]).

```
  IL_0022:  ldloc.1
  IL_0023:  ldc.i4.1
  IL_0024:  add
  IL_0025:  stloc.1
```

Incrementa la seconda variabile locale (Index) di 1.

```
  IL_002a:  ldloc.0
  IL_002b:  newobj     instance void [mscorlib]System.String::.ctor(char[])
```

Converte il nostro array di char in stringa.

```
  IL_0030:  ldc.i4.s   16
  IL_0032:  call       int32 [mscorlib]System.Convert::ToInt32(string,
                                                               int32)
```

Converte la stringa in Int32, per la conversione specifichiamo che il numero è in esadecimale (ldc.i4.s 16).

```
  IL_0037:  stloc.2
```

Salva l'Int32 nella terza variabile locale.

```
  IL_0038:  ldloc.2
  IL_0039:  brtrue.s   IL_003c
  IL_003b:  ret
```

Salta se l'Int32 è diverso da 0, altrimenti esce dalla funzione.

```
  IL_003c:  ldc.i4.0
  IL_003d:  stloc.3
  IL_003e:  br.s       IL_004a
```

Variabile locale 4 = 0. Si direbbe un altro for dalla sintassi. Quindi mi riferisco a questa variabile con Index.

```
  IL_004a:  ldloc.3
  IL_004b:  ldarg.1
  IL_004c:  ldlen
  IL_004d:  conv.i4
  IL_004e:  blt.s      IL_0040
```

Confronta il nuovo Index con la lunghezza dello user name, se minore, continua il ciclo. È chiaro che il ciclo viene ripetuto n volte, dove n = lunghezza user name.

```
  IL_0040:  ldloc.2
  IL_0041:  ldarg.1
  IL_0042:  ldloc.3
  IL_0043:  ldelem.u2
  IL_0044:  sub
  IL_0045:  stloc.2
```

Sottrae dalla cifra corrispondente alla prima parte del seriale il carattere dello user name segnato dall'index.

```
  IL_0046:  ldloc.3
  IL_0047:  ldc.i4.1
  IL_0048:  add
  IL_0049:  stloc.3
```

Incrementa l'index del for. Tenete a mente che la variabile da cui sono stati sottratti i caratteri del seriale è la terza variabile locale (V_3).

```
  IL_0050:  ldc.i4.0
  IL_0051:  stloc.s    V_4
  IL_0053:  br.s       IL_0065
```

Pare che inizi un nuovo for.

```
  IL_0065:  ldloc.s    V_4
  IL_0067:  ldc.i4.4
  IL_0068:  blt.s      IL_0055
```

Se l'index è minore di 4 continua il ciclo.

```
  IL_0055:  ldloc.0
```

```
  IL_0056:  ldloc.s    V_4
  IL_0058:  ldarg.2
  IL_0059:  ldc.i4.5
  IL_005a:  ldloc.s    V_4
  IL_005c:  add
  IL_005d:  ldelem.u2
  IL_005e:  stelem.i2
```

Copia dal Serial[Index + 5] in OurArray[Index]. È chiaro che viene copiata la seconda parte del seriale. D'ora in poi non mostrerò più l'index incrementato nei for, dato che è inutile, passiamo oltre.

```
  IL_006a:  ldloc.0
  IL_006b:  newobj     instance void [mscorlib]System.String::.ctor(char[])
  IL_0070:  ldc.i4.s   16
  IL_0072:  call       int32 [mscorlib]System.Convert::ToInt32(string,
                                                               int32)
  IL_0077:  stloc.s    V_5
```

Converte l'array in cifra, e mette l'intero nella quinta variabile locale.

```
  IL_0079:  ldloc.s    V_5
  IL_007b:  brtrue.s   IL_007e
  IL_007d:  ret
```

Controlla che l'intero sia diverso da 0, altrimenti esce.

```
  IL_007e:  ldloc.s    V_5
  IL_0080:  ldc.i4     0xf000
  IL_0085:  sub
```

Alla cifra viene sottratto 0xF000.

```
  IL_0086:  ldc.i4     0x666
  IL_008b:  xor
```

Il risulta viene xorato con 0x666.

```
  IL_008c:  ldc.i4.2
  IL_008d:  mul
```

Poi moltiplicato per 2.

```
  IL_008e:  ldc.i4.1
  IL_008f:  add
```

E incrementato di uno.

```
  IL_0090:  stloc.s    V_5
```

Dopodiché salvato il risultato.

```
  IL_0092:  ldc.i4.0
  IL_0093:  stloc.s    V_6
  IL_0095:  br.s       IL_00a8
```

Sesto for...

```
  IL_00a8:  ldloc.s    V_6
  IL_00aa:  ldc.i4.4
  IL_00ab:  blt.s      IL_0097
```

Di nuovo minore di 4... Be' abbiamo capito, questo for serve a copiare la 3 parte del seriale nell'array, andiamo oltre.

```
  IL_00ad:  ldloc.0
  IL_00ae:  newobj     instance void [mscorlib]System.String::.ctor(char[])
  IL_00b3:  ldc.i4.8
  IL_00b4:  call       int32 [mscorlib]System.Convert::ToInt32(string,
                                                               int32)
  IL_00b9:  stloc.s    V_7
```

Converte in cifra.. però occhio converte con base di 8, non di 16.

```
  IL_00bb:  ldloc.2
  IL_00bc:  brtrue.s   IL_00db
```

Controlla che la V_3 sia 0, cioè si tratta della cifra da cui abbiamo sottratto i caratteri dello user name, se non è 0, salta al ret. Quindi è chiaro che la prima parte del seriale deve corrispondere ai caratteri del nostro user name sommati.

```
  IL_00be:  ldloc.s    V_5
  IL_00c0:  ldc.i4     0x1337
  IL_00c5:  bne.un.s   IL_00db
```

Controlla che la V_5 sia uguale a 0x1337. Sarebbe il risultato di (((Number - 0xF000) ^ 0x666) * 2) + 1) come abbiamo visto prima.

```
  IL_00c7:  ldloc.s    V_7
  IL_00c9:  ldc.i4     0xad4
  IL_00ce:  bne.un.s   IL_00db
```

Controlla che la terza parte del seriale sia uguale a 0xAD4. Occhio che la conversione va fatta in ottale: 0xAD4 = 5324. Se uguale, viene mostrato il MessageBox:

```
  IL_00d0:  ldstr      "Registered"
  IL_00d5:  call       valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
[System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string)
```

Giusto per curiosità, il seriale per il mio user name (Ntoskrnl) sarebbe 035B-FFFD-5324. Poi è chiaro che tutto l'algo è di una demenza assoluta, però penso che sia sufficiente a farvi avere una buona dimestichezza coll'IL, anzi penso che dopo questo paragrafo abbiate capito che l'Intermediate Language è persino più semplice del comune codice macchina.

Debugging

Il debugging in .NET non è che sia proprio il massimo con i tools a disposizione. E come ben potete capire, non è proprio la stessa cosa del debug normale, dato che ciò che viene debuggato in asm dovrebbe venir riconvertito in IL per poter avere un debug decente. Purtroppo i tool che abbiamo a disposizione per il cracking non ci offrono tale comfort. Il Visual Studio ci fornisce due debugger, uno integrato, ovvero il CLR Debugger (Common Language Runtime Debugger) e un programma console a parte, il Core Debugger (cordbg.exe) che trovate nella stessa cartella dell'ildasm. In questo paragrafo mi limiterò a spiegare il CorDbg poiché per il cracking tra i due debugger non vi corrono molte differenze, sapete usare uno, sapete usare anche l'altro. Poi se pensate che siano debugger potenti... Sono tutti user-mode. Ecco una lista dei comandi disponibili:

ap[**pdomainenum**] [*option*]	Enumerates all application domains, assemblies, and modules in the current process. If you do not specify the *option* argument, the command lists all application domains, assemblies, and modules in the current process. After detaching or attaching, you must specify the **go** command to resume execution. The *option* argument can be one of the following: **attach** Lists the application domains in the process and prompts the user to select the domain to attach to. **detach** Lists the application domains in the process and prompts the user to select the domain to detach from. **0** Lists the application domains in the process. **1** Lists the application domains and assemblies in the process.
as[**sociatesource**] {**s**|**b** *breakpoint id*} *filename*	Associates the given file name with the current stack frame pointer (option **s**) or the specified breakpoint (option **b**).
a[**ttach**] *pid*	Attaches the debugger to a running process. Cordbg.exe kills the program that it is currently debugging (if there is one), and attempts to attach to the process specified by the *pid* argument. The process identification number *pid* can be in decimal or hexadecimal format.
b[**reak**] [[*file*:] *line number*] | [[*class*::] *function* [:*offset*]]	Sets or displays breakpoints. If you do not specify any arguments, the tool displays a list of current breakpoints; otherwise, it sets a breakpoint at the specified location. You can set a breakpoint at a line number in the current source file, a line number in a fully qualified source file, or in a method qualified by a class and optional offset. Breakpoints persist across runs in a session. You can use the **stop** command the same way you use **break**. Cordbg.exe displays breakpoints as "unbound" if the specified breakpoint location cannot be bound to code. When a breakpoint is unbound, it means that the underlying code for the breakpoint location has not been loaded yet. This can happen for a number of valid reasons, such as a misspelled file or class name (they are case-sensitive). Also, breakpoints will be unbound if you set them before running an application. Breakpoints become bound when the real code is loaded. The debugger tries to automatically rebind every unbound breakpoint when it loads a module.
ca[**tch**] [*event*]	Displays a list of event types, or causes the specified event type to stop the debugger. If you do not specify an argument, the tool displays a list of event types, where event types that stop the

	debugger are marked "on," and event types that are ignored are marked "off." If you specify an argument, the debugger stops when events of the specified type occur. By default, the debugger only stops on unhandled exception events (that is, second chance exceptions). Event types that stop the debugger persist across runs in a session. To cause the debugger to ignore a particular type of event, use the **ignore** command. The event argument can be one of the following: **e**[**xceptions**] [exception type] The tool adds the exception type to a list of first chance exceptions to catch. If you do not specify an exception type, the tool catches all first chance exceptions. The exception type is case sensitive, for example, **System.ArgumentException**. **u**[**nhandled**] Unhandled exceptions **c**[**lass**] Class load events **m**[**odule**] Module load events **t**[**hread**] Thread start events
cont [*count*]	Continues the program. If you do not specify an argument, the program continues once. If you do specify an argument, the program continues the specified number of times. This command is useful for continuing a program when a class load event, exception, or breakpoint stops the debugger. You can use the **go** command the same way you use **cont**.
del[**ete**] [*breakpoint id, ...*]	Deletes breakpoints. If you do not specify any arguments, the tool deletes all current breakpoints. If you specify one or more *breakpoint id* arguments, the tool deletes the specified breakpoints. You can obtain breakpoint identifiers by using the **break** or **stop** commands. You can use the **remove** command the same way you use **delete**.
de[**tach**]	Detaches the debugger from the current process. The process automatically continues and runs as if a debugger is not attached to it.
dis[**assemble**] [**0x***address*][{**+**|**-**} *delta*] [*line count*]	Displays native disassembled instructions for the current instruction pointer or *address*, if specified. The default number of instructions displayed is five. If you specify a *line count* argument, the tool displays the specified number of extra instructions before and after the current instruction pointer or address. The last *line count* used becomes the default for the current session. If you specify a delta, the number specified will be added to the current instruction pointer or specified address to begin disassembling.
d[**own**] [*count*]	Moves the stack frame pointer down the stack toward frames called by the current frame for inspection purposes. If you do not specify an argument, the stack frame pointer moves down one frame. If you specify an argument, the stack frame pointer moves down by the specified number of frames. If source level information is available, the tool displays the source line for the frame. This command is frequently used in conjunction with the **up** command.
du[**mp**] *address* [*count*]	Dumps a block of memory, with the output in hexadecimal or decimal format depending on the debugger's mode (see **mode**). The *address* argument is the address of the block of memory. The *count* argument is the number of bytes to dump.
ex[**it**]	Stops the current process and exits the debugger. You can use the **quit** command in the same way you use **exit**.
f[**unceval**] [*class*::] *function* [*arg0 arg1 ...argn*]	Evaluates the specified function on the current thread. The tool stores the new object in the variable *$result* and can use it for subsequent evaluations. Valid arguments are limited to other variables, 4-byte integers, and the constants **Null**, **True**, and **False**. **Note** For a member function, the first argument should be an object of the class or derived class to which the member function belongs.
g[**o**] [*count*]	See **cont**.

h[**elp**] [*command ...*]	Displays descriptions for the specified commands. If you do not specify any arguments, Cordbg.exe displays a list of debugger commands. You can use the **?** command the same way you use **help**.
ig[**nore**] [*event*]	Displays a list of event types or causes the specified event type to be ignored by the debugger. If you do not specify an *event* argument, the tool displays a list of event types, where event types that are ignored are marked "off" and event types that stop the debugger are marked "on." If you specify an argument, the tool ignores events of the specified type. To set an event type to stop the debugger, use the **catch** command. The *event* argument can be one of the following event types: **e**[**xceptions**] [exception type] The tool adds the exception type to a list of first chance exceptions to ignore. If you do not specify an exception type, the tool ignores all first chance exceptions. The exception type is case sensitive, for example, **System.ArgumentException**. **u**[**nhandled**] Unhandled exceptions **c**[**lass**] Class load events **m**[**odule**] Module load events **t**[**hread**] Thread start events
i[**n**] [*count*]	See **step**.
k[**ill**]	Stops the current process. The debugger remains active to process further commands.
l[**ist**] *option*	Displays a list of loaded modules, classes, or global functions. The *option* argument can be one of the following: **mod** Lists the loaded modules in the process. **cl** Lists the loaded classes in the process. **fu** Lists global functions for each module in the process.
m[**ode**] [[*mode name* {**0**|**1**}]	Sets and displays debugger modes for various debugger features. To set a value, specify the *mode name* and a **1** for "on" or **0** for "off." If you do not specify an argument, the tool displays a list of current mode settings. The modes are persisted in the Windows registry between runs of Cordbg.exe. For more information, see the table of debugger mode arguments.
newo[**bj**] *class*	Creates a new object using the current thread. The tool stores the new object in the variable *$result* and can use it for subsequent evaluations.
newobjnc *class*	Creates a new object using the current thread without running a constructor on the object. The new object is initialized to zero. The tool stores the new object in the variable *$result* and can use it for subsequent evaluations.
news[**tr**] *string*	Creates a new string using the current thread. The tool stores the new object in the variable *$result* and can use it for subsequent evaluations.
n[**ext**] [*count*]	Steps the program to the next source line, stepping over function calls. If you do not specify an argument, the tool steps one source line. If you specify an argument, the tool steps the specified number of lines. You can use the **so** command the same way you use **next**.
ns[**ingle**] [*count*]	Steps the program one or more instructions, skipping over function calls. If you do not specify an argument, the tool steps one instruction. If you specify a *count* argument, the tool steps the specified number of instructions.
o[**ut**] [*count*]	Steps the program out of the current function. If you do not specify an argument, the tool performs a step out once for the current function. If you specify an argument, the tool performs a step out the specified number of times.

pa[**th**] [*new path*]	Displays or sets the path used to search for source files and debugging symbols. If you do not specify an argument, the tool displays the current path. If you specify a *new path* argument, it becomes the new path used to search for source files and debugging symbols. This path persists between sessions in the Windows registry.
p[**rint**] [*variable name*]	Displays one or more local variables along with their values. If you do not specify an argument, the tool displays all local variables and their values. If you specify an argument, the tool displays the value of only the specified local variable. For details, see Using the print command in the Examples section.
pro[**cessenum**]	Enumerates all managed processes and the application domains in each process.
q[**uit**]	See **exit**.
ref[**reshsource**] [*source file*]	Reloads the source code for a given source file. The source file to be reloaded must be part of the currently executing program. After setting a source file path with the **path** command, you can use the **refreshsource** command to bring in missing source code.
regd[**efault**] [*force*]	Sets the default just-in-time (JIT) debugger to Cordbg.exe. The command does nothing if another debugger is already registered. Use the *force* argument to overwrite the registered JIT debugger.
reg[**isters**]	Displays the contents of the registers for the current thread.
rem[**ove**] [*breakpoint id, ...*]	See **delete**.
re[**sume**] [~] [*tid*]	Resumes the thread specified by the *tid* argument when the debugger continues. If you use the **~** syntax, the tool resumes all threads except the specified thread. If you do not specify an argument, the command has no effect.
r[**un**] [*executable* [*args*]]	Kills the current process (if there is one) and starts a new one. If you do not specify an *executable* argument, this command runs the program that was previously executed with the **run** command. If you specify an *executable* argument, the tool runs the specified program using the optionally supplied *args*. If Cordbg.exe is ignoring class load, module load, and thread start events (as it is by default), the program stops on the first executable instruction of the main thread.
set *variable value*	Sets the value of the specified *variable* to the specified *value*. The value can be a literal or another variable. For details, see Using the set command. in the Examples section.
setip *line number*	Sets the next statement to execute to the specified *line number*.
sh[**ow**] [*count*]	Displays source code lines. If you do not specify an argument, the tool displays the five source code lines before and after the current source code line. If you specify an argument, the tool displays the specified number of lines before and after the current line. The last *count* specified becomes the default for the current session.
si [<*count*>]	See **step**.
so [<*count*>]	See **next**.
ss[**ingle**] [*count*]	Steps the program one or more instructions, stepping into function calls. If you do not specify an argument, the tool steps into only one instruction. If you specify an argument, the tool performs the specified number of steps.
s[**tep**] [*count*]	Steps the program to the next source line, stepping into function calls. If you do not specify an argument, the program steps to the next line. If you specify an argument, the program steps the specified number of lines. You can use the **si** command or the **in** command the same way you use **step**.
stop [[*file*:] *line number*] | [[*class*::] *function*[:*offset*]] | [**=0x***address*]	See **break**.
su[**spend**] [~] [*tid*]	Suspends the thread specified by the *tid* argument when the debugger continues. If you use the **~** syntax, the tool suspends all threads except the

	specified thread. If you do not specify an argument, the command has no effect.
t[**hreads**] [*tid*]	Displays a list of threads, or sets the current thread. If you do not specify an argument, the tool displays the list of all threads that are still alive and that have run managed code. If you specify an argument, the tool sets the current thread to the specified thread.
up [*count*]	Moves the stack frame pointer up the stack toward frames that called the current frame for inspection purposes. If you do not specify an argument, the stack frame pointer moves up one frame. If you specify an argument, the stack frame pointer moves up by the specified number of stack frames. If source level information is available, the tool displays the source line for the frame.
w[**here**] [*count*]	Displays a stack trace for the current thread. If you do not specify an argument, the tool displays a complete stack trace. If you specify an argument, the tool displays the specified number of stack frames.
wr[**itememory**] *address count byte, ...*	Writes the specified bytes to the target process. The *address* argument specifies the location in which to write the bytes. The *count* argument specifies the number of bytes to write. The *byte* arguments specify what to write to the process. If the number of bytes in the list is less than the *count* argument, the tool wraps the byte list and copies it again. If the number of bytes in the list is more than the *count* argument, the tool ignores the extra bytes.
wt	Steps the application by native instructions, starting from the current instruction and printing the call tree as it goes. The tool prints the number of native instructions executed in each function with the call trace. Tracing stops when the tool reaches the return instruction for the function in which the command was originally executed. At the end of the trace, the tool prints the total number of instructions executed. This command mimics the NT Symbolic Debugger **wt** command, and you can use it for basic performance analysis. Currently, the tool only counts managed code.
x *modulename* **!** *string_to_look_for*	Displays symbols in the specified module that match the pattern specified by the *string_to_look_for* argument. You can use the asterisk (*) character in the *string_to_look_for* argument to indicate to the tool to match anything. The tool ignores any characters after the * character.
? [*command ...*]	See **help**.
>*filename*	Writes all executed commands to the specified *filename*. If you do not specify *filename*, the command stops writing commands to the file.
<*filename*	Reads and executes commands from the specified *filename*.

In verità la cosa utile per quanto riguard il debugging di applicazioni .NET sono i breakpoint. Questo perché se vogliamo crackare è necessario ragionare in IL e il debugger essendo una merda ci fornisce solo il codice asm. I debugger lo possiamo utilizzare per trovare un punto che ci interessa del codice o verificare se una data routine si occupa di una certa cosa. La sintassi più comune (per le altre vedetevi la tabella) del comando breakpoint sarebbe questa:

```
b ModuleName !Namespace.Class::Function
```

Prendiamo per esempio la winapp con cui abbiamo lavorato finora. Mettiamo di voler verificare che effettivamente il btnRegister_Click sia la funzione richiamata quando pigiamo il bottone Register. Avviamo il debugger e immettiamo:

```
r WinApp.exe
```

r sta per run. Dopodiché settiamo il breakpoint con:

```
b WinApp.exe !WinApp.WinAppForm::btnRegister_Click
```

Se il breakpoint è settato correttamente il debugger vi dirà che è active, altrimenti vi dirà che è unbound, significa che il debugger non è riuscito a trovare la locazione di codice. Questo potrebbe essere legato a diversi motivi, chessò applicazione non ancora avviata (magari non avete fatto run) oppure che avete scritto male il nome di classe/funzione. Comunque in questo caso il breakpoint è attivo. Adesso dobbiamo fare g (go) per continuare l'esecuzione dell'applicazione (il tracing ricomincerà appena occorrerà un evento di debug, quale il nostro breakpoint). Adesso pigiamo Register nell'applicazione e vedremo il debugger poppare e l'applicazione bloccarsi. In questo modo abbiamo verificato quale routine andare a reversare. Il lavoro di reversing vero e proprio per adesso ve lo fate da disassembler ma tanto l'IL è semplice e come presto vedremo anche troppo. Ad ogni modo potete settare breakpoint anche su funzioni esterne, se ricordate all'interno della HandleException veniva chiamata la funzione:

```
IL_0032:  call       int32 [mscorlib]System.Convert::ToInt32(string, int32)
```

Posso benissimo mettere un breakpoint su questa funzione. Intanto cancelliamo il breakpoint di prima con del 1 (scrivete del e basta per cancellare tutti i bp) dopodiché scrivete:

```
b mscorlib.dll !System.Convert::ToInt32
```

Di nuovo g e vedrete appena pigiate register (inserendo ovviamente username e serial della lunghezza giusta) il debugger poppare. Sì, ma adesso noi non sappiamo da dove è stato chiamato quel ToInt32.. Per rimediare a ciò è necessario digitare il comando w (where) seguito dal numero di calling sequence che vogliamo vedere (3 direi è sufficiente per individuare il punto esatto). Dovreste vedere una cosa di questo genere:

```
(cordbg) w 3
Thread 0x84c Current State:Normal
0)* mscorlib!System.Convert::ToInt32 +0000 [no source information available]
                value=(0x00abc0a8) "1234"
                fromBase=0x00000010
1)  winapp!WinApp.WinAppForm::HandleException +00ee [no source information avail
able]
                a=(0x00ab64f0) array with dims=[6]
                b=(0x00ab654c) array with dims=[14]
2)  winapp!WinApp.WinAppForm::btnRegister_Click +01fa [no source information ava
ilable]
                sender=(0x00aacf10) <System.Windows.Forms.Button>
                e=(0x00aaccf0) <System.EventArgs>
--- Managed transition ---

(cordbg) _
```

Come vedete la calling sequence è ben chiara:

btnRegister_Click -> HandleException -> ToInt32

Abbiamo tutte le informazioni che ci servono per iniziare a reversare. Ripeto il debugger al momento fa davvero schifo, quindi fatene l'uso che potete, ma per adesso potete contare veramente solo (si fa per dire) sul disassembling e sul decompiling. Non mi va di aggiungere altro sul debugging, la lista più spiegazione dei comandi ce l'avete, non vi serve altro. Passiamo al prossimo paragrafo che è molto succoso.

Decompilers

Come probabilmente avrete notato, l'Intermediate Language segue regole ben fisse, è, a differenza dall'assembler che siamo avezzi a vedere, molto più regolare. Questo ci porta direttamente al corrente paragrafo, che non è, badate, sul disassembling, ma sul decompiling. In questo caso è necessario tracciare una linea di confine ben netta. E' vero che poi alla fine sono cose analoghe, ma decompilare significa tornare a quello che era il sorgente prima della compilazione: né l'asm, né l'IL corrispondono al sorgente in cui abbiamo scritto il nostro programma (si spera). Anche per quanto riguarda l'asm, molti sapranno, sono stati tentati simili esperimenti. Ad esempio decompilatori asm -> C si trovano in rete, ma il procedimento è estremamente più complesso (e non sempre possibile) che per quanto riguarda l'IL. Gli eseguibili .NET forniscono tutte le informazioni sulla classi, i namespace ecc, inoltre l'IL è compilato in maniera molto regolare, mentre l'asm cambia da compilatore a compilatore (da versione a versione spesso dello stesso compilatore). Insomma è un gran casino decompilare partendo dall'asm e invece estremamente semplice partendo dall'IL. In rete si trovano diversi tools, eccovene alcuni:

Decompiler.NET
Exemplar/Anakrino
Reflector .NET Decompiler
Salamander .NET Decompiler

Non li ho provati tutti, solo il Reflector (che tra l'altro è free) e il Salamander (che è molto avanzato, nonché molto commerciale). Per piccole cose anche il reflector va più che bene, prendiamo per esempio la HandleException del paragrafo precedente. Ecco il sorgente originale che ho scritto:

```
private void HandleException(char[] a, char[] b)
{
   if (b[4] != b[9] || b[4] != '-')
      return;

   char[] buf = new char[4];

   for (int x = 0; x < 4; x++)
      buf[x] = b[x];

   Int32 Tot = Convert.ToInt32(new String(buf), 16);

   if (Tot == 0)
      return;

   for (int x = 0; x < a.Length; x++)
      Tot -= a[x];

   for (int x = 0; x < 4; x++)
      buf[x] = b[5 + x];

   Int32 Tot2 = Convert.ToInt32(new String(buf), 16);

   if (Tot2 == 0)
      return;

   Tot2 = ((((Tot2 - 0xF000) ^ 0x666) * 2) + 1);

   for (int x = 0; x < 4; x++)
      buf[x] = b[10 + x];

   Int32 Tot3 = Convert.ToInt32(new String(buf), 8);

   if (Tot == 0 && Tot2 == 0x1337 && Tot3 == 0xAD4)
   {
      MessageBox.Show("Registered");
   }
}
```

Ed ecco il codice che il reflector mi riproduce partendo dall'IL:

```
private void HandleException(char[] a, char[] b)
{
      if ((b[4] != b[9]) || (b[4] != '-'))
      {
            return;
      }
      char[] chArray1 = new char[4];
      for (int num1 = 0; num1 < 4; num1++)
      {
            chArray1[num1] = b[num1];
      }
      int num2 = Convert.ToInt32(new string(chArray1), 0x10);
      if (num2 == 0)
      {
            return;
      }
      for (int num3 = 0; num3 < a.Length; num3++)
      {
            num2 -= a[num3];
      }
      for (int num4 = 0; num4 < 4; num4++)
      {
            chArray1[num4] = b[5 + num4];
      }
      int num5 = Convert.ToInt32(new string(chArray1), 0x10);
      if (num5 == 0)
      {
            return;
      }
      num5 = (((num5 - 61440) ^ 0x666) * 2) + 1;
      for (int num6 = 0; num6 < 4; num6++)
      {
            chArray1[num6] = b[10 + num6];
      }
      int num7 = Convert.ToInt32(new string(chArray1), 8);
      if (((num2 == 0) && (num5 == 0x1337)) && (num7 == 0xad4))
      {
            MessageBox.Show("Registered");
      }
}
```

Buono eh? Sì in effetti è perfetto. E considerate che il codice può essere tradotto nel linguaggio .NET che vogliamo. Per esempio VB:

```
Private Sub HandleException(ByVal a As Char(), ByVal b As Char())
      If ((b(4) <> b(9)) OrElse (b(4) <> "-")) Then
            Return
      End If
      Dim chArray1 As Char() = New Char(4  - 1) {}
      Dim num1 As Integer
      For num1 = 0 To 4 - 1
            chArray1(num1) = b(num1)
      Next num1
      Dim num2 As Integer = Convert.ToInt32(New String(chArray1), 16)
      If (num2 = 0) Then
            Return
      End If
      Dim num3 As Integer
      For num3 = 0 To a.Length - 1
            num2 = (num2 - a(num3))
      Next num3
      Dim num4 As Integer
      For num4 = 0 To 4 - 1
            chArray1(num4) = b((5 + num4))
      Next num4
      Dim num5 As Integer = Convert.ToInt32(New String(chArray1), 16)
      If (num5 = 0) Then
            Return
      End If
      num5 = ((((num5 - 61440) Xor 1638) * 2) + 1)
      Dim num6 As Integer
      For num6 = 0 To 4 - 1
            chArray1(num6) = b((10 + num6))
      Next num6
      Dim num7 As Integer = Convert.ToInt32(New String(chArray1), 8)
      If (((num2 = 0) AndAlso (num5 = 4919)) AndAlso (num7 = 2772)) Then
            MessageBox.Show("Registered")
      End If
End Sub
```

Incredibile vero? Abbiamo del codice bello pronto per essere compilato. Al momento il Reflector support tre linguaggi: C#, VB e Delphi. Mentre il Salamander ne offre molti di più (che non sto nemmeno a elencare) e offre una ricostruzione a dir poco eccellente del codice di un assembly .NET. Ho provato a decompilare la winapp e il Salamander mi ha tirato fuori l'intero codice, senza star a clickare sulle singole funzioni per averlo, guardare per credere:

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;

namespace WinApp
{
  public class WinAppForm : Form
  {
```

```
private TextBox tbUserName;

private Label label1;

private Label label2;

private TextBox tbSerial;

private Button btnRegister;

private Container components = null;


public WinAppForm()
{
  InitializeComponent();
}

protected override void Dispose(bool disposing)
{
  if (disposing && components != null)
  {
    components.Dispose();
  }
  base.Dispose(disposing);
}

private void InitializeComponent()
{
  tbUserName = new TextBox();
  label1 = new Label();
  label2 = new Label();
  tbSerial = new TextBox();
  btnRegister = new Button();
  base.SuspendLayout();
  tbUserName.Location = new Point(8, 24);
  tbUserName.Name = "tbUserName";
  tbUserName.Size = new Size(224, 20);
  tbUserName.TabIndex = 0;
  tbUserName.Text = "";
  label1.Location = new Point(8, 8);
  label1.Name = "label1";
  label1.Size = new Size(100, 16);
  label1.TabIndex = 1;
  label1.Text = "User Name:";
  label2.Location = new Point(8, 56);
  label2.Name = "label2";
  label2.Size = new Size(100, 16);
  label2.TabIndex = 2;
  label2.Text = "Serial:";
  tbSerial.Location = new Point(8, 72);
  tbSerial.Name = "tbSerial";
  tbSerial.Size = new Size(224, 20);
  tbSerial.TabIndex = 3;
  tbSerial.Text = "";
  btnRegister.Location = new Point(80, 104);
  btnRegister.Name = "btnRegister";
  btnRegister.TabIndex = 4;
  btnRegister.Text = "Register";
  btnRegister.Click += new EventHandler(this.btnRegister_Click);
  base.AutoScaleBaseSize = new Size(5, 13);
  base.ClientSize = new Size(240, 134);
  base.Controls.Add(btnRegister);
  base.Controls.Add(tbSerial);
  base.Controls.Add(label2);
  base.Controls.Add(label1);
  base.Controls.Add(tbUserName);
  base.MaximizeBox = false;
  base.Name = "WinAppForm";
  base.StartPosition = FormStartPosition.CenterScreen;
  base.Text = "WinApp";
  base.ResumeLayout(false);
}

[STAThreadAttribute()]
private static void Main()
{
  Application.Run(new WinAppForm());
}

private void HandleException(char[] a, char[] b)
{
  if (b[4] != b[9] || b[4] != '-')
  {
    return;
  }
  char[] chs = new char[4];
  for (int i1 = 0; i1 < 4; i1++)
  {
    chs[i1] = b[i1];
  }
  int j1 = Convert.ToInt32(new String(chs), 16);
  if (j1 == 0)
```

```
      {
        return;
      }
      for (int k1 = 0; k1 < (int)a.Length; k1++)
      {
        j1 -= a[k1];
      }
      for (int i2 = 0; i2 < 4; i2++)
      {
        chs[i2] = b[5 + i2];
      }
      int j2 = Convert.ToInt32(new String(chs), 16);
      if (j2 == 0)
      {
        return;
      }
      j2 = (j2 - 61440 ^ 1638) * 2 + 1;
      for (int k2 = 0; k2 < 4; k2++)
      {
        chs[k2] = b[10 + k2];
      }
      int i3 = Convert.ToInt32(new String(chs), 8);
      if (j1 == 0 && j2 == 4919 && i3 == 2772)
      {
        MessageBox.Show("Registered");
      }
    }

    private void btnRegister_Click(object sender, EventArgs e)
    {
      if (tbUserName.Text.Length < 6 || tbSerial.Text.Length != 14)
      {
        return;
      }
      char[] chs1 = tbUserName.Text.ToCharArray();
      char[] chs2 = tbSerial.Text.ToCharArray();
      try
      {
        char ch = chs1[0];
        int i = 0;
        for (int j = 0; j < tbUserName.Text.Length; j++)
        {
          i += chs1[j];
        }
        for (int k = 0; k < tbSerial.Text.Length; k++)
        {
          i += chs2[k];
        }
        if (i == 4919)
        {
          MessageBox.Show("Thank You For Registering");
        }
        else
        {
          MessageBox.Show("Invalid Name/Serial");
        }
      }
      catch (Exception)
      {
        HandleException(chs1, chs2);
      }
    }
  }

}
```

Basta fare copia/incolla del codice in un progetto .NET e potremo compilare questo codice, producendo la STESSA IDENTICA applicazione. Cioè come ben capite qui non si tratta solo di cracking, qui c'è di mezzo la sicurezza del codice, è troppo facile rubare codice in questa maniera. E non fatevi impressionare dal Salamander o da altri decompiler, non sono progetti granché complessi, semplicemente è troppo facile e immediato decompilare l'IL. Poi il Salamander in verità è solo una fregatura come ben presto vedremo. Poi è chiaro che, per quanto concerne il nostro ambito, i programmi che verranno messi in rete così, allo sbaraglio, saranno crackati all'istante. Anche se, tutto sommato, non mi pare giusto che il programmatore ignaro metta in circolazione il proprio programma senza sapere che sta fornendo i sorgenti della sua applicazione, intendo dire che comunque quel tizio ha sudato per scrivere quel determinato codice: un conto è se gira una crack (cosa probabilmente inevitabile), un altro conto è se qualcuno ruba il codice scritto da sto tizio per fare il proprio programma. Se qualcuno ha deciso di distribuire i propri programmi come open source, dovrebbe essere conscio dei rischi a cui è sottoposto. Ma c'è gente che non l'ha scelto e corre altamente questo rischio. Quindi è chiaro che dobbiamo ragionare in questa maniera: i programmatori diverranno sempre più smaliziati (come in questo caso, a mio parere, è giusto che sia) e le protezioni aumenteranno di numero e complessità. Nei paragrafi successivi ci occuperemo proprio della protezione del codice, conoscenza che non ci sarà utile solamente a fini di reversing.

Code Obfuscation

Eccoci arrivati a una vera e propria truffa degli ultimi anni. In verità il termine code obfuscation pare promettente, peccato che gli obfuscators attuali siano veramente quel che di peggio ci si possa immaginare. Ne ho provati diversi, sul developer center del .net ho trovato questa lista:

Decompiler.NET
Deploy.NET
Dotfuscator Professional
Salamander .NET Obfuscator
Semantic Designs: C# Source Code Obfuscator
Spices.Net
Thinstall

Demeanor for .NET
XenoCode .NET Obfuscator and Optimizer

Partiamo dal dotfuscator che è compreso nel pacchetto del visual studio, ma che bisogna pagare separatamente per averlo registrato. Se non lo registriamo il programma ci avverte che offrirà solo una protezione parziale alle nostre applicazioni. Non mi sono nemmeno scomodato di registrarlo, tanto lo stesso dotfuscator è dotfuscato (visto che è scritto in .NET), quindi per avere un'idea della protezione offerta da questo obfuscator è sufficiente analizzare il suo stesso eseguibile. Non ho potuto scaricare l'ultima versione di questo tool perché chiedono registrazione e cazzi vari per il download (né si trova sui p2p la nuova versione), quindi mi baso sulla versione trovata allegata al Visual Studio.

Ad ogni modo se aprite con ildasm o con un decompiler il dotfuscator vedrete che tutti i namespace, tutte le classi e tutte le funzioni sono state rinominate. Vi ritroverete nomi tipo a, b, c, d, ecc. Oppure aaa0, aaa1, aaa2 ecc. Insomma avete capito cosa intendo. Vediamo per esempio la funzione:

```
.method private hidebysig static void  c() cil managed
{
  // Code size       205 (0xcd)
  .maxstack  7
  .locals init (string[] V_0,
           object[] V_1)
  IL_0000:  ldc.i4.5
  IL_0001:  newarr     string
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  ldc.i4.0
  IL_0009:  ldsfld     class [dfengine]gb ge::a
  IL_000e:  callvirt   instance string class [dfengine]gb::d()
  IL_0013:  stelem.ref
  IL_0014:  ldloc.0
  IL_0015:  ldc.i4.1
  IL_0016:  ldstr      bytearray (18 39 )                                      // .9
  IL_001b:  call       string a$PST06000001(string)
  IL_0020:  stelem.ref
  IL_0021:  ldloc.0
  IL_0022:  ldc.i4.2
  IL_0023:  ldsfld     class [mscorlib]System.Resources.ResourceManager ge::a
  IL_0028:  ldstr      bytearray (75 39 69 3B 7B 3D 61 3F 03 41 0E 43 0D 45 19 47   // u9i;{=a?.A.C.E.G
                                  1E 49 0F 4B 1E 4D 1D 4F 19 51 1D 53 1A 55 )       // .I.K.M.O.Q.S.U
  IL_002d:  call       string a$PST06000001(string)
  IL_0032:  callvirt   instance string class [mscorlib]System.Resources.ResourceManager::GetString(string)
  IL_0037:  ldc.i4.1
  IL_0038:  newarr     object
  IL_003d:  stloc.1
  IL_003e:  ldloc.1
  IL_003f:  ldc.i4.0
  IL_0040:  ldsfld     class [dfengine]gb ge::a
  IL_0045:  callvirt   instance class [mscorlib]System.Version class [dfengine]gb::a()
  IL_004a:  stelem.ref
  IL_004b:  ldloc.1
  IL_004c:  call       string class [dfengine]fw::a(string,
                                                    object[])
  IL_0051:  stelem.ref
  IL_0052:  ldloc.0
  IL_0053:  ldc.i4.3
  IL_0054:  ldstr      bytearray (18 39 )                                      // .9
  IL_0059:  call       string a$PST06000001(string)
  IL_005e:  stelem.ref
  IL_005f:  ldloc.0
  IL_0060:  ldc.i4.4
  IL_0061:  ldsfld     class [dfengine]gb ge::a
  IL_0066:  callvirt   instance string class [dfengine]gb::i()
  IL_006b:  stelem.ref
  IL_006c:  ldloc.0
  IL_006d:  call       string string::Concat(string[])
  IL_0072:  call       void class [mscorlib]System.Console::WriteLine(string)
  IL_0077:  ldsfld     class [dfengine]gb ge::a
  IL_007c:  callvirt   instance string class [dfengine]gb::c()
  IL_0081:  call       void class [mscorlib]System.Console::WriteLine(string)
  IL_0086:  ldsfld     class [mscorlib]System.Resources.ResourceManager ge::a
  IL_008b:  ldstr      bytearray (75 39 69 3B 7B 3D 61 3F 03 41 0E 43 0D 45 19 47   // u9i;{=a?.A.C.E.G
                                  04 49 03 4B 0F 4D 0B 4F 1E 51 01 53 11 55 13 57 ) // .I.K.M.O.Q.S.U.W
  IL_0090:  call       string a$PST06000001(string)
  IL_0095:  callvirt   instance string class [mscorlib]System.Resources.ResourceManager::GetString(string)
  IL_009a:  ldsfld     class [dfengine]gb ge::a
  IL_009f:  callvirt   instance string class [dfengine]gb::g()
  IL_00a4:  call       void class [mscorlib]System.Console::WriteLine(string,
                                                                      object)
  IL_00a9:  ldsfld     class [mscorlib]System.Resources.ResourceManager ge::a
  IL_00ae:  ldstr      bytearray (75 39 69 3B 7B 3D 61 3F 03 41 0E 43 0D 45 19 47   // u9i;{=a?.A.C.E.G
                                  1B 49 0F 4B 1E 4D 07 4F 11 51 1E 53 )             // .I.K.M.O.Q.S
  IL_00b3:  call       string a$PST06000001(string)
  IL_00b8:  callvirt   instance string class [mscorlib]System.Resources.ResourceManager::GetString(string)
  IL_00bd:  ldsfld     class [dfengine]gb ge::a
  IL_00c2:  callvirt   instance string class [dfengine]gb::a()
  IL_00c7:  call       void class [mscorlib]System.Console::WriteLine(string,
                                                                      object)
  IL_00cc:  ret
} // end of method ge::c
```

Metodo c in classe ge. Come vedete sono nomi pseudo-casuali, apposta per rendere il codice meno leggibili. Contro la decompilazione questa versione del dotfuscator non ha nulla, quindi:

```
  private static void c()
  {
    Console.WriteLine(string.Concat(new string[]{a.d(), a("\u3918"),
fw.a(a.GetString(a("\u3975\u3b69\u3d7b\u3f61\u4103\u430e\u450d\u4719\u491e\u4b0f\u4d1e\u4f1d\u5119\u531d\u551a")), new object[]
{a.a()}), a("\u3918"), a.i()}));
    Console.WriteLine(a.c());

Console.WriteLine(a.GetString(a("\u3975\u3b69\u3d7b\u3f61\u4103\u430e\u450d\u4719\u4904\u4b03\u4d0f\u4f0b\u511e\u5301\u5511\u5713")),
a.g());
    Console.WriteLine(a.GetString(a("\u3975\u3b69\u3d7b\u3f61\u4103\u430e\u450d\u4719\u491b\u4b0f\u4d1e\u4f07\u5111\u531e")), a.a());
  }
```

Come vedete i Writeline sono alquanto strani, questo perché le stringhe sono crittate. Sembra na cosa buona, ma è una boiata colossale, semplicemente il dotfuscator inietta un metodo statico nell'assembly a cui passando una stringa la restituisce decrittata come valore di ritorno. Il metodo in questione lo troviamo in "a" come vedete.

```
.method privatescope hidebysig static string
        a$PST06000001(string A_0) cil managed
{
  // Code size       92 (0x5c)
  .maxstack  4
  .locals init (char[] V_0,
           int32 V_1,
           int32 V_2,
           char V_3,
           unsigned int8 V_4,
           unsigned int8 V_5)
  IL_0000:  ldarg.0
  IL_0001:  callvirt   instance int32 string::get_Length()
  IL_0006:  conv.ovf.u4
  IL_0007:  newarr     class [mscorlib]System.Char
  IL_000c:  stloc.0
  IL_000d:  ldc.i4     0xaa17b38
  IL_0012:  stloc.1
  IL_0013:  ldc.i4.0
  IL_0014:  stloc.2
  IL_0015:  br.s       IL_004a
  IL_0017:  ldarg.0
  IL_0018:  ldloc.2
  IL_0019:  callvirt   instance char string::get_Chars(int32)
  IL_001e:  stloc.3
  IL_001f:  ldloc.3
  IL_0020:  ldc.i4     0xff
  IL_0025:  and
  IL_0026:  ldloc.1
  IL_0027:  dup
  IL_0028:  ldc.i4.1
  IL_0029:  add
  IL_002a:  stloc.1
  IL_002b:  xor
  IL_002c:  conv.u1
  IL_002d:  stloc.s    V_4
  IL_002f:  ldloc.3
  IL_0030:  ldc.i4.8
  IL_0031:  shr
  IL_0032:  ldloc.1
  IL_0033:  dup
  IL_0034:  ldc.i4.1
  IL_0035:  add
  IL_0036:  stloc.1
  IL_0037:  xor
  IL_0038:  conv.u1
  IL_0039:  stloc.s    V_5
  IL_003b:  ldloc.0
  IL_003c:  ldloc.2
  IL_003d:  ldloc.s    V_5
  IL_003f:  ldc.i4.8
  IL_0040:  shl
  IL_0041:  ldloc.s    V_4
  IL_0043:  or
  IL_0044:  conv.u2
  IL_0045:  stelem.i2
  IL_0046:  ldloc.2
  IL_0047:  ldc.i4.1
  IL_0048:  add
  IL_0049:  stloc.2
  IL_004a:  ldloc.2
  IL_004b:  ldloc.0
  IL_004c:  ldlen
  IL_004d:  conv.i4
  IL_004e:  blt.s      IL_0017
  IL_0050:  ldloc.0
  IL_0051:  newobj     instance void string::.ctor(char[])
  IL_0056:  call       string string::Intern(string)
  IL_005b:  ret
} // end of method 'Global Functions'::a
```

Vediamocelo decompilato, che facciamo prima:

```
static string a(string A_0)
{
      char[] chArray1 = new char[A_0.Length];
```

```
        int num1 = 178355000;
        for (int num2 = 0; num2 < chArray1.Length; num2++)
        {
              char ch1 = A_0[num2];
              byte num3 = (byte) ((ch1 & '\x00ff') ^ num1++);
              byte num4 = (byte) ((ch1 >> 8) ^ num1++);
              chArray1[num2] = (char) ((ushort) ((num4 << 8) | num3));
        }
        return string.Intern(new string(chArray1));
}
```

Se volete farvi un banale decrypt di una stringa:

```
using System;

namespace Decrypt
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
      {
         String Res = DecryptString("\u3975\u3b69\u3d7b\u3f61\u4103\u430e\u450d\u4719\u491e\u4b0f\u4d1e\u4f1d\u5119\u531d\u551a");

         Console.WriteLine(Res);
         Console.ReadLine();
      }

      static string DecryptString(string A_0)
      {
         char[] chArray1 = new char[A_0.Length];
         int num1 = 178355000;
         for (int num2 = 0; num2 < chArray1.Length; num2++)
         {
            char ch1 = A_0[num2];
            byte num3 = (byte) ((ch1 & '\x00ff') ^ num1++);
            byte num4 = (byte) ((ch1 >> 8) ^ num1++);
            chArray1[num2] = (char) ((ushort) ((num4 << 8) | num3));
         }
         return string.Intern(new string(chArray1));
      }

   }
}
```

Potete pure usare quella routine per risolvervi tutte le stringhe. Potremmo anche programmare qualcosa che toglie da tutto l'assembly le stringhe crittate senza alcuna difficoltà. Ma vedremo meglio nel prossimo paragrafo come fare certe cose. Un altro obfuscator che ho provato è stato il Salamander e ve ne avevo già accennato, ma anche quello è una fregatura bestiale: se proteggete l'assembly il salamander decompiler si rifiuta di decompilarlo dicendo che è protetto, ma prendendo qualsiasi altro decompiler vedete il codice normalmente senza problemi, insomma alla fine tutta la protezione fornita dal salamander si basa su una signature che gli indica che è stato protetto col suo aggeggio. Rinominare metodi e classi non basta a fornire una buona protezione, certo anche quello va fatto, ma mi pare davvero il minimo (una cosa implicita potremmo dire). Passiamo al prossimo paragrafo.

Protection Theory

Allora il problema fondamentale di questo paragrafo è che se si vuole proteggere un assembly in qualsiasi maniera possibile le opzioni sono tante, e si può anche fare un bel lavoro, soprattutto con una conoscenza approfondita della struttura degli assembly (ultima sezione del PE, probabilmente a breve farò un update nella guida sul PE per spiegarla), ma le opzioni diminuiscono esponenzialmente se si vuole restare nella logica del .NET. Cioè potremmo anche farci un packer e crittare il codice IL e eseguire assembler in una nostra dll o nell'assembly stesso che decritta il codice. Tanto finché win non identifica il processo come .NET possiamo fare tutte le cose che abbiamo sempre fatto. Il problema è che così facendo il .NET perde gran parte della sua ragione d'essere. L'ultima protezione che mi sono fatto mandare è stato il PC Guard, una protezione idiota. Allora proteggendo così possiamo anche buttare via la tecnologia .NET e tornare al MFC. Se la protezione applicata al nostro programma ci toglie tutta la portabilità e la flessibilità del .NET... Addio. Facciamone direttamente a meno, altrimenti il .NET si riduce al solo all'ambiente di programmazione (con l'aggiunta che è più lenta l'esecuzione).

In verità non ho tempo per approfondire molto questo paragrafo. Sto cercando di finire la guida al più presto perché in questo momento ho mille cose da fare. Però non posso lasciarvi proprio a mani vuote, quindi vi introduco la Reflection del .NET, che sta alla base di quasi tutte le protezioni (code obfuscators) al momento in circolazione. Vedremo l'uso dei portentosi namespace System.Reflection e System.Reflection.Emit. Sull'uso di questi namespace vi sono già due tutorial in italiano scritti da Quake2, ad ogni modo c'è un monte di roba anche sulla microsoft se siete capaci a leggere in inglese. Allora prendiamo la banale applicazione:

```
using System;

namespace SimpleApp
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
      {
```

```
         int A = 4, B = 3;

         if ((A + B) == 7)
            Console.WriteLine("The result is 7");

         Console.ReadLine();

      }
   }
}
```

Vabbe' non c'è certo bisogno di spiegare cosa fa. Disassemblando ci ritroviamo queste istruzioni IL:

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
  // Code size       27 (0x1b)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1)
  IL_0000:  ldc.i4.4
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.3
  IL_0003:  stloc.1
  IL_0004:  ldloc.0
  IL_0005:  ldloc.1
  IL_0006:  add
  IL_0007:  ldc.i4.7
  IL_0008:  bne.un.s   IL_0014
  IL_000a:  ldstr      "The result is 7"
  IL_000f:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0014:  call       string [mscorlib]System.Console::ReadLine()
  IL_0019:  pop
  IL_001a:  ret
} // end of method Class1::Main
```

Adesso vedremo come usare la Reflection per creare un eseguibile contenente una Main con esattamente questo codice.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace Sample
{
   class Class1
   {
      [STAThread]
      static void Main(string[] args)
      {
         // create assembly

         AssemblyName AsmName = new AssemblyName();

         AsmName.Version = new Version(1,0,0,0);

         AsmName.Name = "MyApp";

         AssemblyBuilder AsmBuilder =
            AppDomain.CurrentDomain.DefineDynamicAssembly(AsmName,
            AssemblyBuilderAccess.Save);

         // create module

         ModuleBuilder ModBuilder =
            AsmBuilder.DefineDynamicModule("MyApp", "MyApp.exe");

         // define the class

         TypeBuilder TypeBuild = ModBuilder.DefineType("MyApp.MyAppClass",
            TypeAttributes.Class | TypeAttributes.Public);

         // create Main method

         MethodBuilder MainMethod = TypeBuild.DefineMethod("Main",
            MethodAttributes.Static,
            CallingConventions.Standard, typeof(void),
            new Type[] {typeof(string[])});

         ILGenerator ILGen = MainMethod.GetILGenerator();

         // create A & B local variables

         ILGen.DeclareLocal(typeof(int));
         ILGen.DeclareLocal(typeof(int));

         ILGen.Emit(OpCodes.Ldc_I4_4);

         // init variables

         ILGen.Emit(OpCodes.Ldc_I4_4);
         ILGen.Emit(OpCodes.Stloc_0);
         ILGen.Emit(OpCodes.Ldc_I4_3);
```

```
        ILGen.Emit(OpCodes.Stloc_1);

        // add

        ILGen.Emit(OpCodes.Ldloc_0);
        ILGen.Emit(OpCodes.Ldloc_1);
        ILGen.Emit(OpCodes.Add);

        // if

        Label jump1 = ILGen.DefineLabel();

        ILGen.Emit(OpCodes.Ldc_I4_7);
        ILGen.Emit(OpCodes.Bne_Un_S, jump1);

        // print string

        ILGen.Emit(OpCodes.Ldstr, "The result is 7");

        Type[] Params = new Type[] {typeof(string)};
        MethodInfo Method = typeof(Console).GetMethod("WriteLine", Params);

        ILGen.EmitCall(OpCodes.Call, Method, null);

        // jump here

        ILGen.MarkLabel(jump1);

        // read input

        Method = typeof(Console).GetMethod("ReadLine");
        ILGen.EmitCall(OpCodes.Call, Method, null);

        // clear stack and exit

        ILGen.Emit(OpCodes.Pop);
        ILGen.Emit(OpCodes.Pop);
        ILGen.Emit(OpCodes.Ret);

        // clear type

        TypeBuild.CreateType();

        // set the main as entry point for the assembly

        AsmBuilder.SetEntryPoint(MainMethod, PEFileKinds.ConsoleApplication);

        // save assembly

        AsmBuilder.Save("MyApp.exe");

      }
   }
}
```

Francamente non c'è molto da spiegare, il metodo migliore per capire è che vi leggiate il codice. I vari Emit sono, come ben potete immaginare, per scrivere le istruzioni nell'assembly creato. Quando vogliamo far chiamare un metodo dobbiamo prendere le informazioni tramite GetMethod e per quanto riguarda i salti dobbiamo usare le label, ovvero DefineLabel e MarkLabel. Le altre cose mi sembrano commentate a sufficienza nel codice. Per il momento eseguiamo l'exe generato dal compilatore e vedremo un nuovo exe comparire nella stessa cartella, ovvero: MyApp.exe. Decompiliamo il main dell'exe creato:

```
/* private scope */ static void Main(string[])
{
      int num1 = 4;
      int num2 = 3;
      if ((num1 + num2) == 7)
      {
            Console.WriteLine("The result is 7");
      }
      Console.ReadLine();
}
```

Possiamo anche eseguire, come vedete abbiamo creato un eseguibile perfettamente funzionante. Ok, abbiamo visto come creare un assembly, ma possiamo modificare anche assembly già esistenti, iniettare codice: per esempio la routine per decrittare le stringhe del dotfuscator. Abbiamo un gran controllo su un assembly ignorandone del tutto gli internals (anche solo la metadata degli assembly). Non solo, possiamo anche creare qualsiasi assembly senza nemmeno salvarlo su disco, semplicemente per runnarne i metodi: invece che AssemblyBuilderAccess.Save, AssemblyBuilderAccess.Run (c'è anche RunAndSave). Nel prossimo update vedremo di approfondire decisamente di più, magari vi propongo il codice di un code obfuscator un po' più sofisticato.

Conclusioni

Be', come visto, al momento il .NET è la cosa più facile da crackare al mondo. Ovviamente le cose cambieranno e anche questa guida subirà updates. Per quanto riguarda la Protection Theory non ho potuto proporvi approfondimenti perché non ne ho il tempo, sono agli sgoccioli e sto cercando di scrivere più in fretta possibile anche le conclusioni. Probabilmente come detto anche il mio tutorial sul PE subirà un update per spiegare la struttura degli assembly. Un saluto a qualche persona prima di chiudere: Quake2 (perché è stato il primo a credere nel .NET), Misanthropic (perché fa quello che gli dico, grazie dai) e Satanik (che mi ha mandato giusto il giorno della prima release della guida il PCGuard, così da poterlo almeno testare).

Bona.

Ntoskrnl