

## Neófito Reversando .NET [Entrega #2][LUISFECAB]



<b>HERRAMIENTAS</b>	Windows 7 Home Premium SP1 x32 Bits (SO donde trabajamos) dnSpy v6.0.5 MSIL OpCode Table v1.0 <a href="#">DESCARGAR HERRAMIENTAS</a> <a href="#">DESCARGAR TUTO+ARCHIVOS</a>
<b>AUTOR</b>	<b>LUISFECAB</b>
<b>RELEASE</b>	Agosto 21 2019 [TUTORIAL 019]

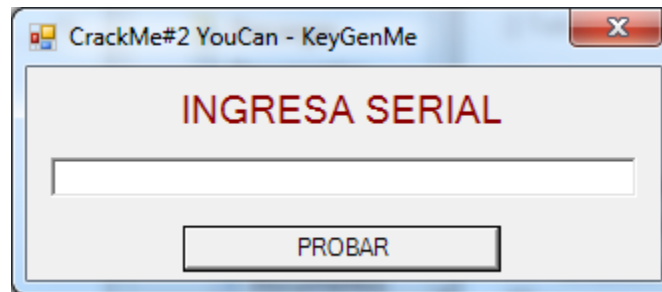
## INTRODUCCIÓN

Continuaremos con la **Entrega #2**, pero antes un poco de mi historia en Cracking. Todo empieza ya hace años atrás cuando caducaron las licencias de prueba de algunos programas para hacer cálculos y diseños para perforación y producción en Ing. De Petróleos. Fue ahí cuando me di a la búsqueda de cómo se podría lograr volver a tenerlos **FULL**, para eso me puse a buscar cómo hacer eso, y esa búsqueda me llevó hacia la página del **Maestro Ricardo Narvaja**, que en aquella época no estaba bloqueada por los desgraciados de Google. Sin saber, que para mí fortuna había llegado a la meca del conocimiento del Cracking, para que gente como yo, que no sabe nada con respecto a ese tema.

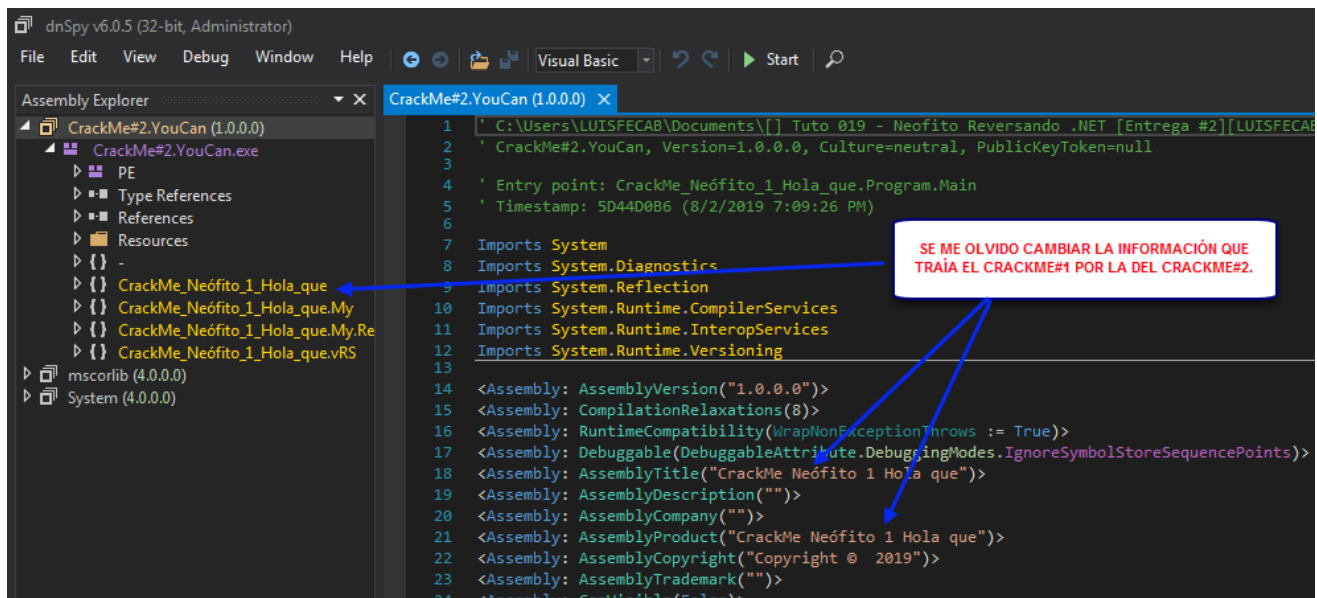
Gracias a que llegué al lugar correcto pude conocer la gran comunidad de **CracksLatinoS**, en donde con el curso del **Maestro Ricardo** y luego con aquellos miles de tutoriales del resto de los miembros pude hacerme a conocimientos jamás pensados por mí, que me han permitido poder tener mis primeros logros en Cracking. Así que quiero agradecer a todos los **CracksLatinoS** que me han ayudado con sus respuestas a mis dudas o con sus tutoriales. Aquí quiero reconocer personalmente a **sequeyo** que gracias a sus tutoriales de **.NET** me dieron las bases para poder Reversar **.NET**, claro a la medida de mis capacidades, habilidades y conocimiento respecto a este tema de los **.NET**.

Después de recordar y agradecer, nos enfocaremos en el **<CrackMe#2.YouCan>**, el cual ya lo había subido al canal de **@PeruCrackerS** de Telegram. Este es un reto **KeyGenMe**, que hasta este momento que escribo estas primeras líneas no lo han resuelto. Con este CrackMe aumentamos un poquito el nivel para tener un poco más de código para trazar y analizar, e ir familiarizándonos con este y aprender a realizar cambios en **IL** o **CIL**.

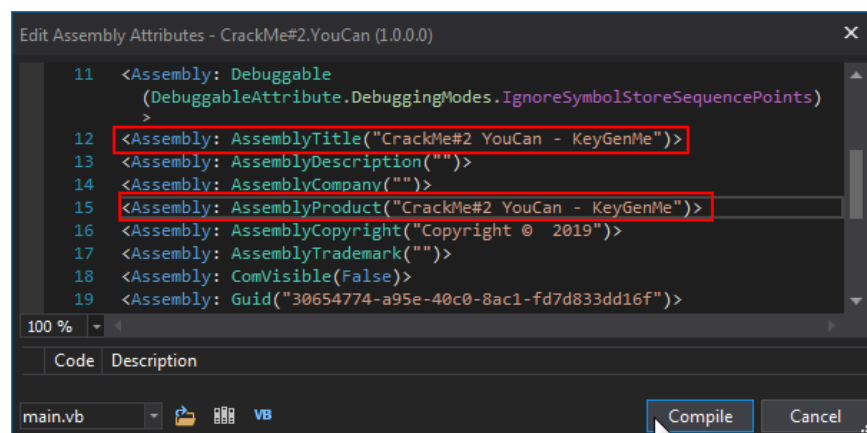
# ENTREGA #1



Este CrackMe lo hice sobre el mismo proyecto de la primera entrega para evitarme la fatiga de repetir lo mismo. Lo cargamos en nuestro **<dnSpy>**.

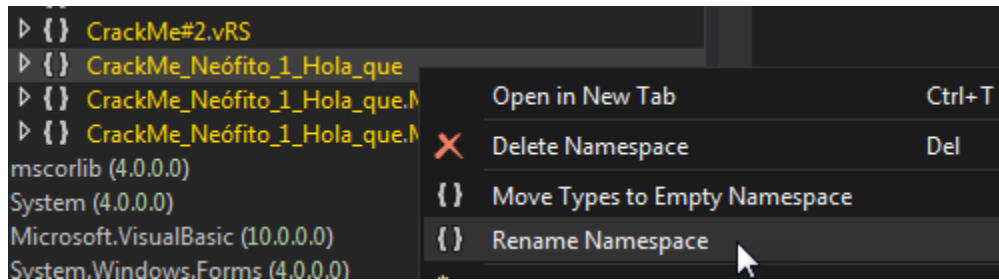


Por el afán de terminar este <CrackMe#2.YouCan> no cambié información que traía del CrackMe anterior, pero sea esta la oportunidad para realizar cambios en **AssemblyTitle** y **AssemblyProduct**. Entonces <Clic Derecho>Edit Method (Visual Basic)...>.

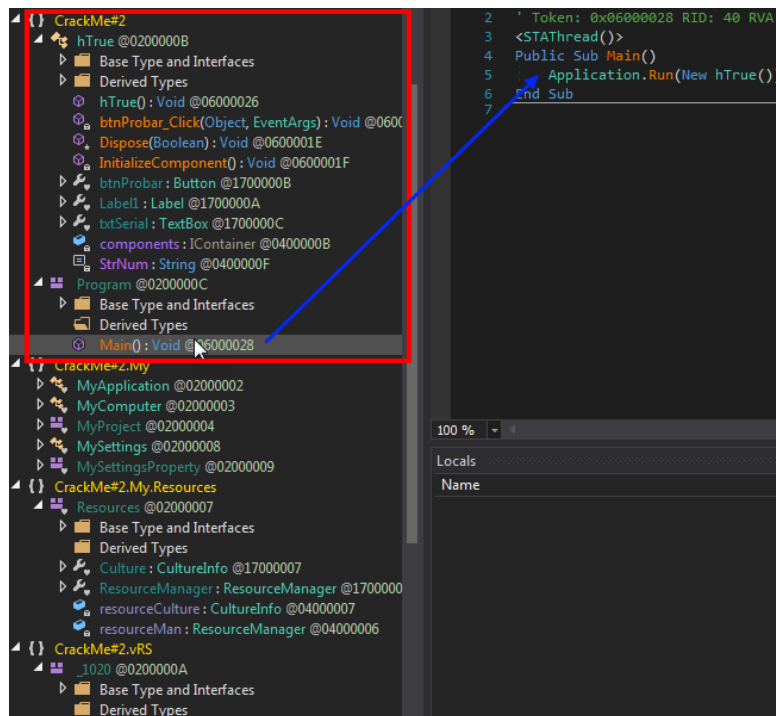


## Neófito Reversando .NET [Entrega #2][LUISFECAB]

Agrego los cambios, compilo con "**Compile**" y guardo los cambios reemplazando el CrackMe anterior. También podemos cambiarle el nombre a los elementos.

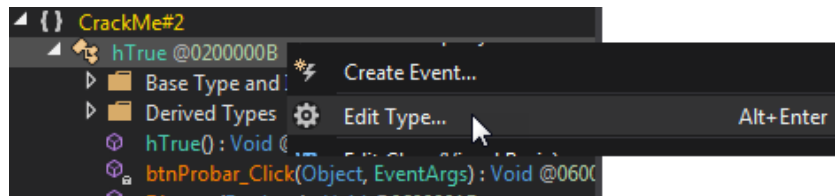


Los voy a renombrar como mi **<CrackMe#2>**. Solo renombraré esas cuatro, con eso es suficiente, y ahora si guardamos los cambios **<File->Save Module>**. Yo lo sobrescribí. Lo miramos por encimita a ver que trae. Recordemos que hacer eso es una sana costumbre para ir familiarizándonos.

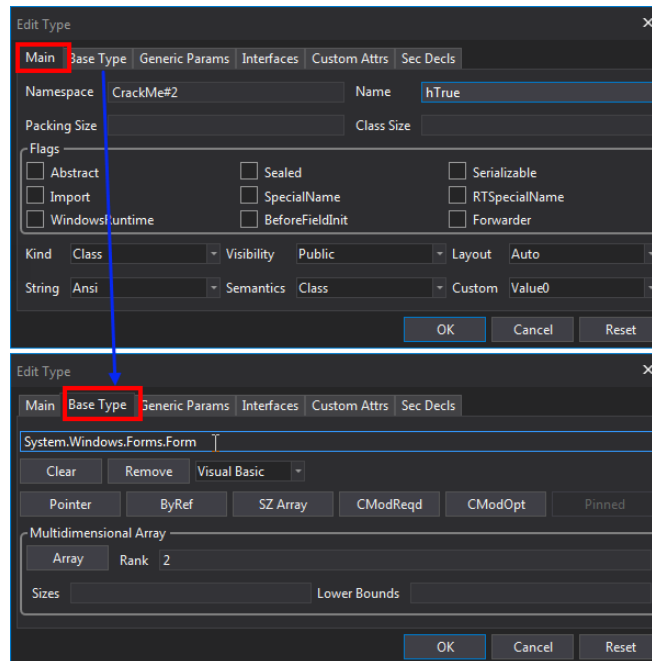


Tiene un par de elementos más con respecto al CrackMe de la **Entrega #1** pero que como que no contienen nada relevante para nuestro propósito, me refiero a **CrackMe#2.My** y **CrackMe#2.My.Resources**. Como vemos, lo que nos interesa está en el elemento **CrackMe#2**. Podemos ver que el **ENTRY POINT** lo tenemos el Módulo **Program**, que simplemente ejecuta nuestro Form con **Application.Run(New hTrue())**. Sabemos que es un Form porque observamos controles en él, además si no me equivoco el evento **InitializeComponent()** siempre lo encontraremos presente en un Form porque este carga todo el diseño del Form. Podemos obtener más información de estos con **< Clic Derecho->Edit Type...>**.

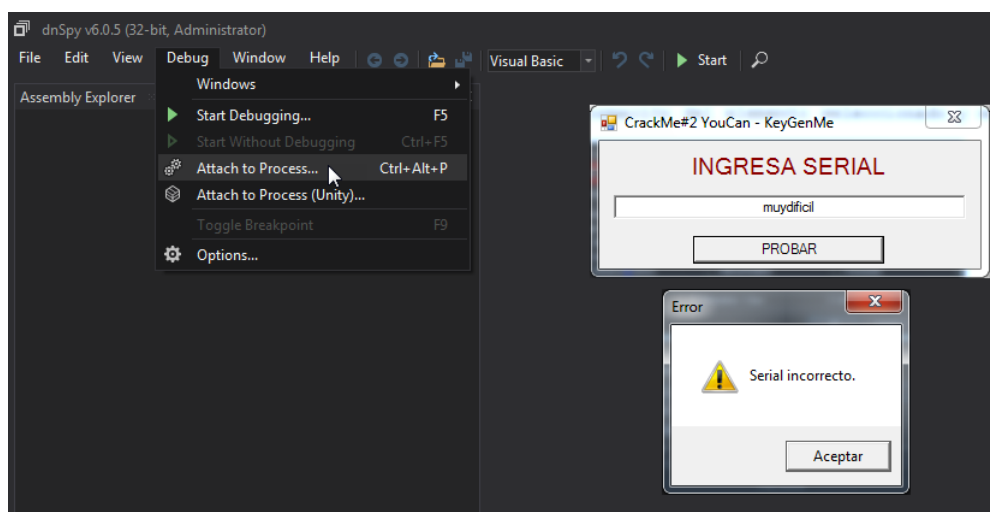
## Neófito Reversando .NET [Entrega #2][LUISFECAB]



Ahí, tenemos toda la información del elemento seleccionado y mucho más.

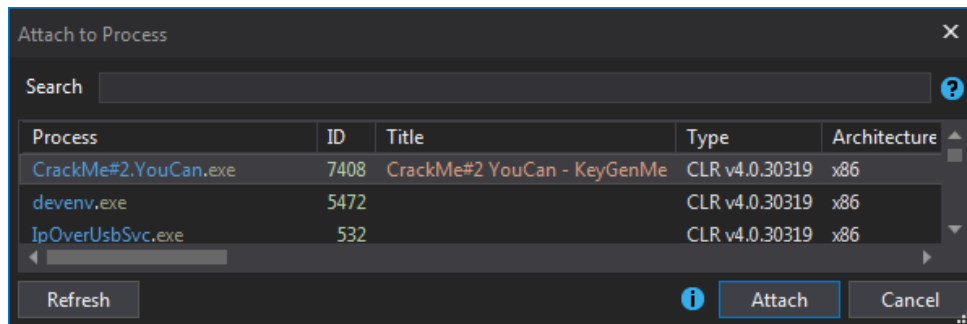


Desde ahí podemos editarle muchas cosas que para nuestros propósitos de Cracking yo nunca he utilizado. Podemos confirmar que estamos con un Form. Bueno, cerremos todo los ensamblados desde **<File->Close All>** y ejecutemos nuestro CrackMe normalmente, no cargado desde el **<dnSpy>**. Lo probamos con un **SERIAL**, el mío será el de siempre, **"muydifícil"**. Aquí lo que haremos es Atachearlo con el **<dnSpy>**.

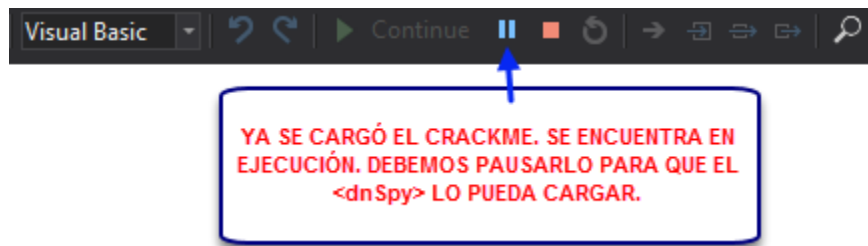


Con **<Debug->Attach to Process...>** se nos abrirá una ventana con las aplicaciones que podemos Atachear.

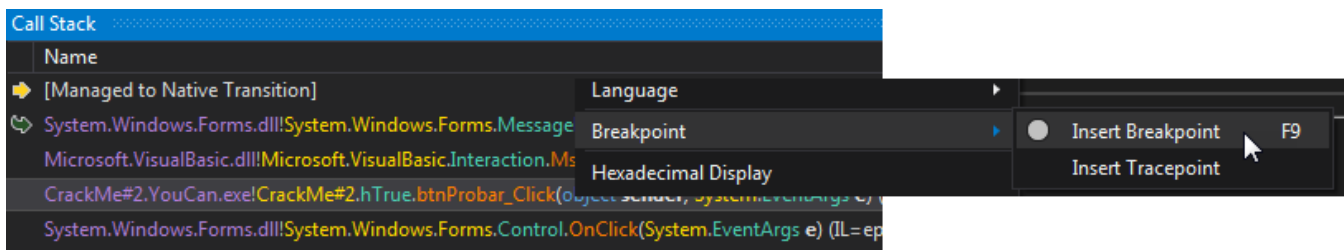
## Neófito Reversando .NET [Entrega #2][LUISFECAB]



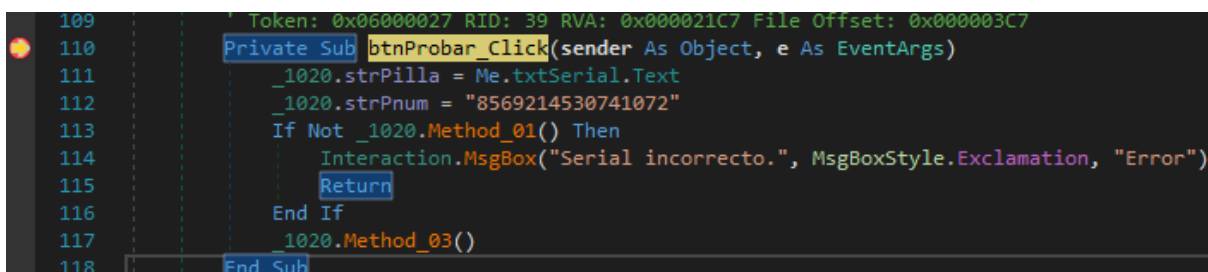
Escogemos nuestro **<CrackMe#2>** y le damos a **"Attach"**. Aquí la idea de esto es mostrar que tenemos otra forma de cargar nuestro **TARGET**, además el CrackMe está mostrando al **"CHICO MALO"**; paso seguido podemos llegar al lugar que originó ese mensaje utilizando la técnica del **CALL STACK**.



Vuelvo y repito lo que dice la imagen. Pausamos el **<dnSpy>** para que se cargue el CrackMe, y de pasó al estar pausado ponemos en práctica lo aprendido con el **CALL STACK**.

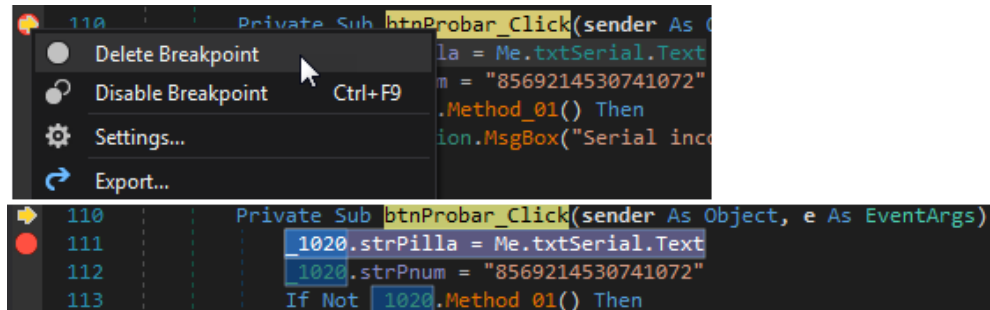


Como vemos, gracias a la información del **CALL STACK** sabemos que en **btnProbar\_Click** se pudo haber originado nuestro **"CHICO MALO"**. Como ya aprendimos, podemos ir al lugar donde se encuentra con **<Clic Derecho->Go To Source Code>** o **<Doble Clic>**, pero aquí le podremos un **BREAKPOINT** para poder detenernos cuando aceptemos el mensaje del **"CHICO MALO"**. Recordar mi apreciado y entusiasta lector que lo tenemos pausado, así que lo ponemos a correr de nuevo, **Continue** y aceptamos el mensaje.

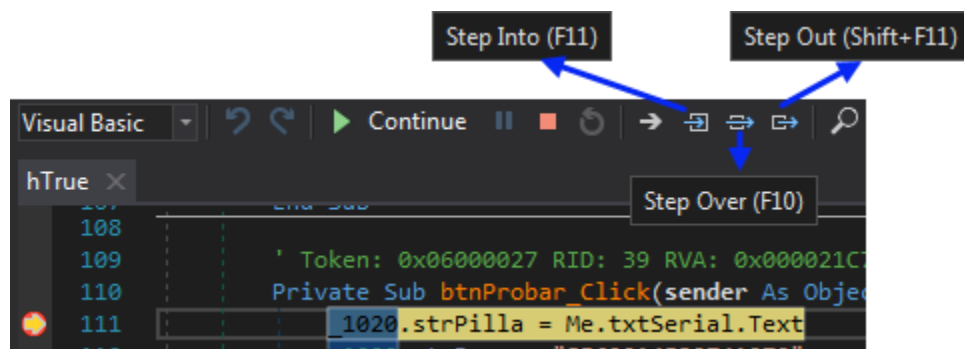


## Neófito Reversando .NET [Entrega #2][LUISFECAB]

Llegamos al lugar indicado, eso es algo bueno, pero debo aclarar que ese **<BREAKPOINT>** no es ideal tenerlo ahí porque no nos detendremos cuando hagamos **<Click>** en el Botón **"PROBAR"** si no cuando aceptemos el mensaje del **"CHICO MALO"**, así que mejor lo quitamos y ponemos otro en la siguiente línea de código, **\_1020.strPilla = Me.txtSerial.Text**.



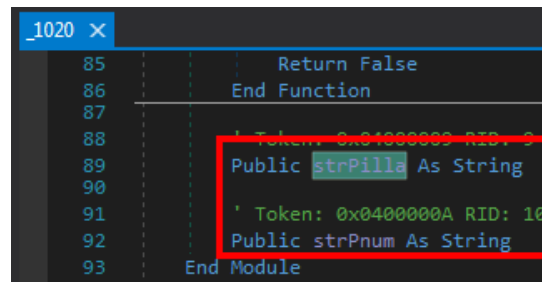
La comprobación parece corta. Ahora sí, probemos de nuevo nuestro **SERIAL**. Les recuerdo de nuevo a mis lectores olvidadizos que estamos pausados, así que **<Click>** en **Continue**. Moraleja, siempre asegurarnos que tengamos corriendo el **TARGET** porque de lo contrario no podemos interactuar con él.



Nos detuvimos al probar el **SERIAL**. Y después de mucha carreta vamos a conocer las tres formas para Tracear las líneas de código; las que yo uso son las dos primeras, **<Step Into (F11)>** que permite tracear línea por línea entrando a todos los Métodos que se encuentre en su camino, y **<Step Over (F10)>** con el cual podemos tracear el código de determinado procedimiento pero nos evitamos entrar al código de los Métodos que estén en ese procedimiento que estamos Traceando. Como vemos en la imagen de arriba, nos encontramos detenidos en **1020.strPilla = Me.txtSerial.Text**. Es fácil deducir que nuestro **SERIAL** será guardado en **1020.strPilla**, bueno y cómo entendemos eso. Resulta que **strPilla** es una variable que está declarada en **1020**, y que es una variable Pública, también deducimos eso porque para poder hacer uso de esta en otros lugares de la aplicación debe ser declarada como Pública. Hagamos **<Click>** sobre **strPilla** para ir donde es declarada. Cuando encontremos cosas similares, ya sabremos cómo interpretarlas.

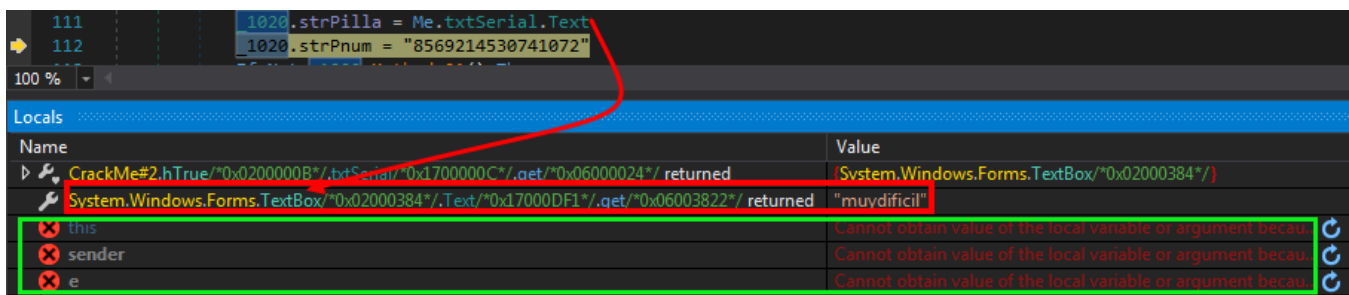


## Neófito Reversando .NET [Entrega #2][LUISFECAB]



```
_1020 x
85      Return False
86      End Function
87
88      ' Token: 0x04000000 RID: 0
89      Public strPilla As String
90
91      ' Token: 0x0400000A RID: 10
92      Public strPnum As String
93      End Module
```

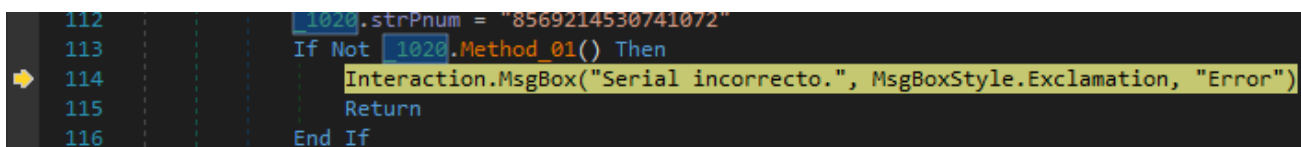
En el Módulo **\_1020** se declaran esas dos variables Públicas, y que ya las conocemos porque estas son utilizadas para almacenar dos Strings, que son nuestro **SERIAL** y un número pero que está representado como una String. Bien, volvamos al Traceo. Tracemos con <F10>, pasando esa primera línea de código.



```
111      1020.strPilla = Me.txtSerial.Text
112      1020.strPnum = "8569214530741072"

Locals
Name                                     Value
└─ CrackMe#2.hTrue/*0x0200000B*/.txtSerial/*0x1700000C*/.get/*0x06000024*/.returned (System.Windows.Forms.TextBox/*0x02000384*/)
└─ System.Windows.Forms.TextBox/*0x02000384*/.Text/*0x17000DF1*/.get/*0x06003822*/.returned "muydifícil"
└─ this Cannot obtain value of the local variable or argument because it is a local variable.
└─ sender Cannot obtain value of the local variable or argument because it is a local variable.
└─ e Cannot obtain value of the local variable or argument because it is a local variable.
```

Vemos que se muestra el valor retornado que es nuestro **SERIAL**. Una cosa es que sabemos los valores que se almacenarán en **\_1020.strPilla** y **\_1020.strPnum**, pero no tenemos forma de ver esas variables en **LOCALS**, así que es cuestión de nosotros ir recordando eso. Debemos notar que en **LOCALS** tenemos errores de descompilación ya que el <dnSpy> no puede interpretarlo; ¡hay carachas! Parece que sin querer queriendo terminé haciendo un CrackMe marrullero, me estoy superando... jejeje. Sigamos Traceando con <F10> para ver qué sigue.



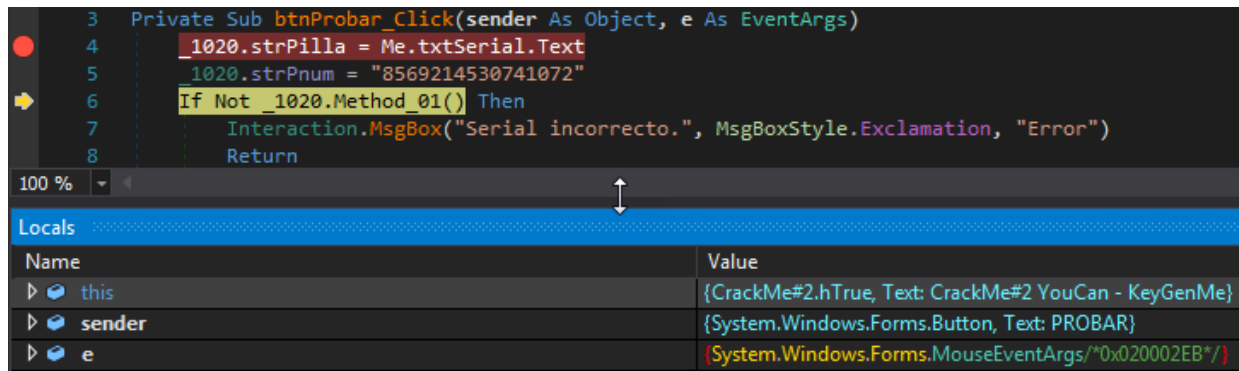
```
112      1020.strPnum = "8569214530741072"
113      If Not 1020.Method_01() Then
114      Interaction.MsgBox("Serial incorrecto.", MsgBoxStyle.Exclamation, "Error")
115      Return
116      End If
```

De una nos mandó al "**CHICO MALO**", no pasamos por **If Not \_1020.Method\_01() Then**; lo lógico es que pudiéramos tracear esa línea de código, pero no fue el caso y eso ya es muy extraño.

Han pasado unos días desde que escribí las últimas líneas de este tuto, y quiero que miremos la siguiente imagen, que desvirtúa mi vanidad cuando escribí que este CrackMe era marrullero y que el <dnSpy> no lo descompilaba bien.

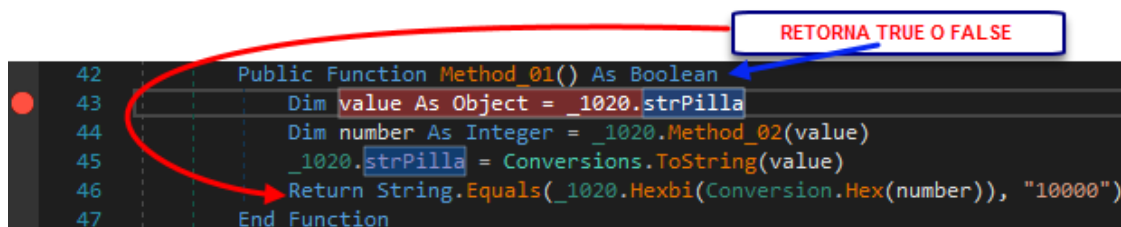


## Neófito Reversando .NET [Entrega #2][LUISFECAB]



Como pueden ver, me encuentro detenido en `If Not _1020.Method_01() Then`, y además en **LOCALS** se muestra todo correcto, no hay errores al mostrar toda la información. Entonces, qué pasó aquí, si antes no se mostraba y el Traceo no era correcto. Pues la verdad, no tengo ni idea pero sea la oportunidad de aprender cosas nuevas, puede que el `<dnSpy>` en la vez pasada algo lo tenía medio loco, así que cuando miremos que el `<dnSpy>` no trabaje de la manera que sabemos debería hacerlo y tengamos resultados como el de más arriba; lo que debemos hacer es cerrarlo y volver a ejecutarlo, ya que a lo mejor con eso corriamos eso, y es exactamente lo que me sucedió. Menos mal no seguí escribiendo este tutorial estando medio loco el `<dnSpy>` porque hubiera quedado un poco más enredado.

Ya aclarado lo anterior, seguimos por buen camino. Regresemos donde estamos parados, en `If Not _1020.Method_01() Then`. Tomaremos el "CHICO MALO" dependiendo del valor que retorne `_1020.Method_01()`. El `If Not ..... Then`, decide sobre si es **TRUE** O **FALSE**, de lo que ocurre en este, por ejemplo podríamos decir `If 1=1 Then`, como `1=1` entonces sería **VERDAD** (**TRUE**) y entraríamos al `If`, ahora nosotros tenemos un `Not` antes, y este simplemente invierte la condición para tomar el `If`; ahora para tomar el `If` debe ser **FALSE** debido al `Not` pero como no queremos entrar, entonces debemos obtener un **TRUE**. Con eso podemos deducir que `_1020.Method_01()` es una función que retorna un valor Boolean. Sigamos esa función.



Lo primero que haremos es ponerle un `<BREAKPOINT>` para detenernos cuando entre en acción ese Método. Como vemos es una función que al hacer sus cálculos retornará **TRUE** o **FALSE**. El retorno del **TRUE** o **FALSE** no es tan evidente porque está dado por `String.Equals(_1020.Hexbi(Conversion.Hex(number)), "10000")`, pero si miramos el Método `String.Equals`.

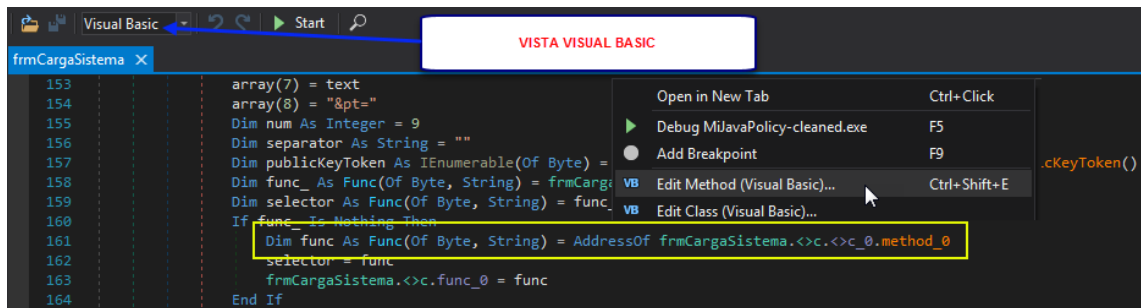
```
Public Shared Overloads Function Equals(a As String, b As String) As Boolean
    Return a = b OrElse (a IsNot Nothing AndAlso b IsNot Nothing AndAlso a.Length = b.Length AndAlso String.EqualsHelper(a, b))
End Function
```

Retorna un valor Boolean al revisar dos Strings. Bien, sabemos que nuestro **SERIAL** "muydifícil" no es correcto pero vamos a forzar a que sea correcto, y así de esta

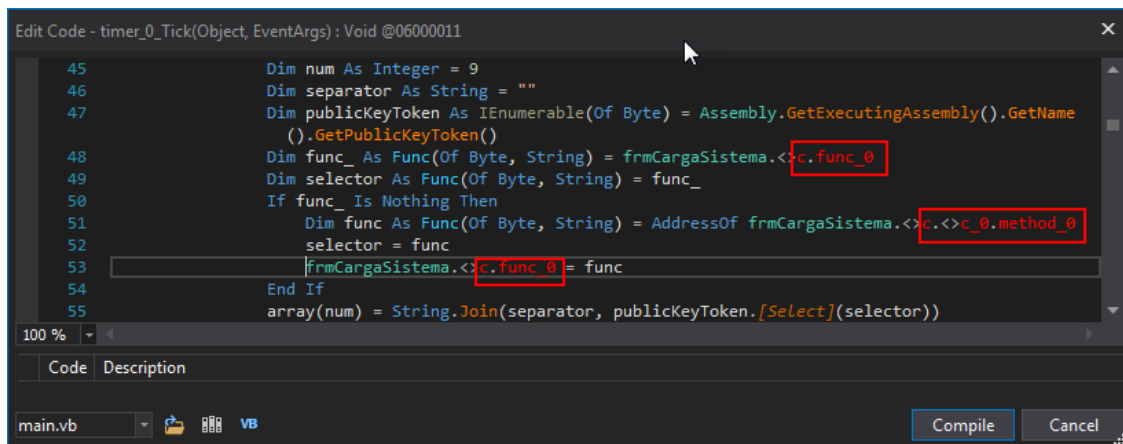
## Neófito Reversando .NET [Entrega #2][LUISFECAB]

forma tener la oportunidad para adentrarnos en poder realizar cambios utilizando **IL** o **CIL**.

Poder hacer cambios a través de **IL** es vital porque en muchas ocasiones no podremos hacer cambios en **C#** o **Visual Basic** como lo hicimos en la **Entrega #1** porque el **<dnSpy>** no puede interpretar correctamente el código. Voy a utilizar código de otra aplicación para explicarme mejor.



Tenemos ese código, el cual le queremos hacer algún cambio. Entonces **<Clic Derecho>Edit Method (Visual Basic)...** para abrir el editor.



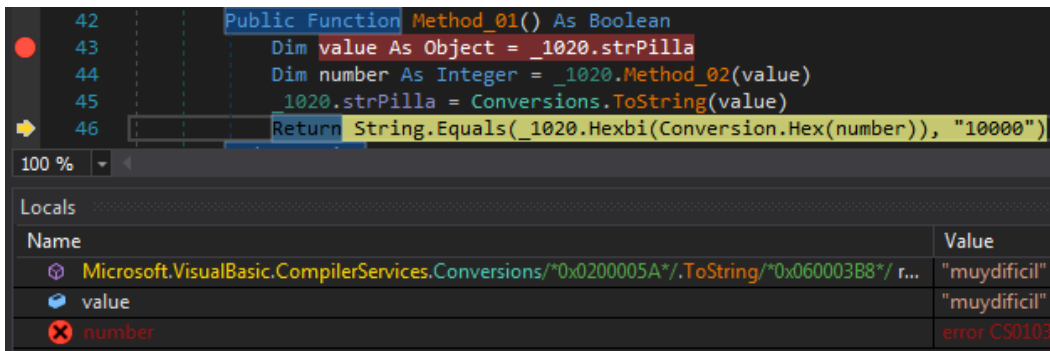
Como vemos, están resaltado en rojo unas partes del código, lo que nos informa que no se reconoce, y si quisiéramos compilar aún sin cambiar nada, no se podría porque el **<dnSpy>** no lo reconoce como código correcto. Aquí es donde entra en juego el editor de código **IL**, el cual si nos permite hacer cambios que no se pueden hacer en **C#** o **Visual Basic**, claro está que deben ser cambios bien hechos, no cambiar por cambiar porque de lo contrario obtendremos una compilación corrupta.

Recapitulando, miremos nuestro **If** que nos muestra al **"CHICO MALO"** y que debemos evitar.

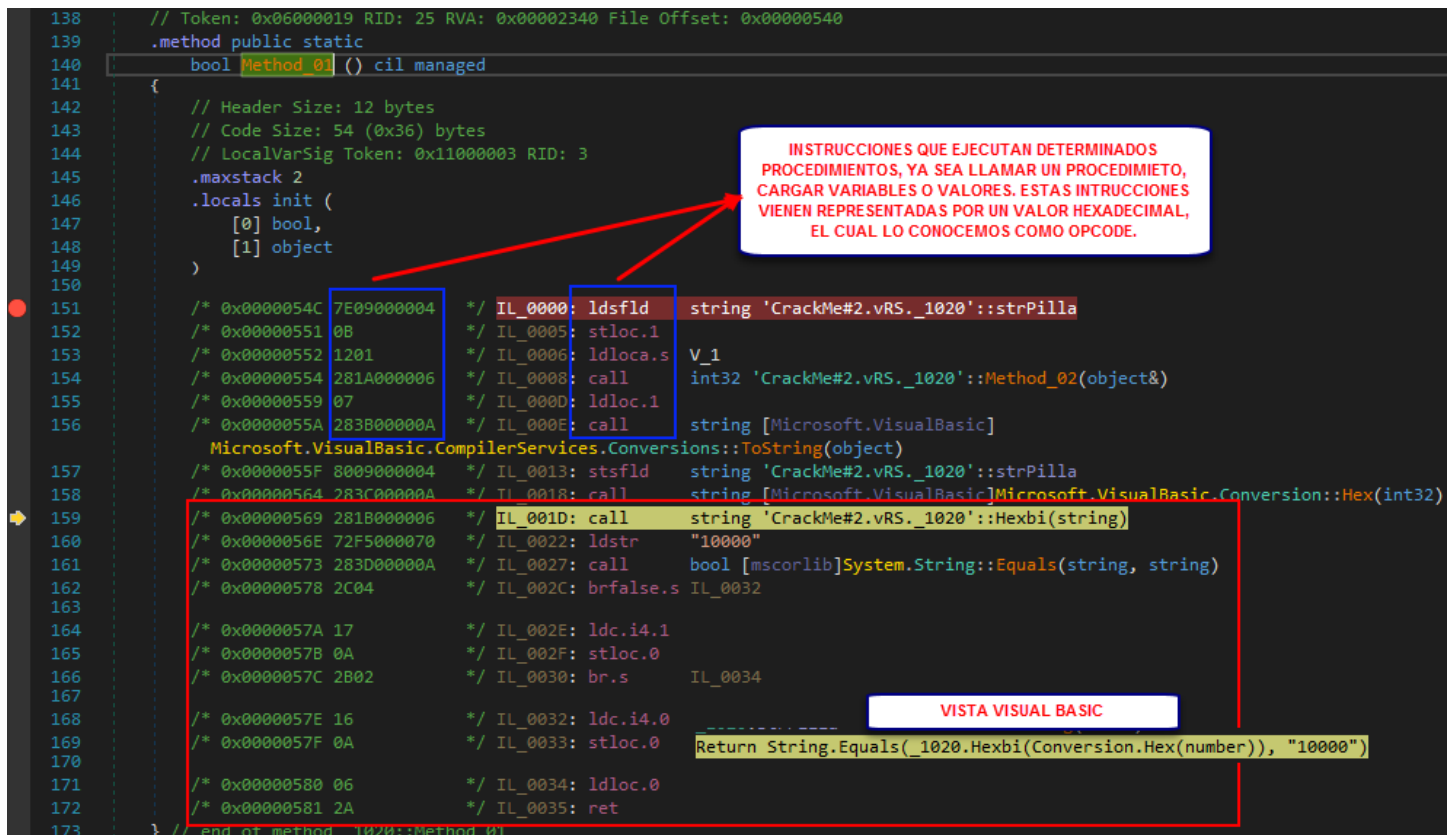
```
If Not _1020.Method_01() Then
    Interaction.MsgBox("Serial incorrecto.", MsgBoxStyle.Exclamation, "Error")
    Return
End If
```

Como tenemos **If Not** entonces **\_1020.Method\_01()** debe retornar **TRUE** para no tomar el **If**. Probemos nuestro **SERIAL**, continuemos nuestro Traceo hasta detenernos en el **<BREAKPOINT>** que pusimos en **\_1020.Method\_01()**.

## Neófito Reversando .NET [Entrega #2][LUISFECAB]



Estos escritos se basan sobre el principio de que no sabemos mucho de **.NET** y que a este punto solo queremos es aprender a hacer cambios sencillos que nos permitan Crackear una aplicación, que en este caso es nuestro inocente CrackMe. Podemos ver que algo se hace con nuestro **SERIAL "muydifícil"**, y como estamos suponiendo que ni idea de lo que se hace con él, solo sabemos que debemos retornar **TRUE** para evitar al **"CHICO MALO"**. He Traciado hasta llegar al **Return String.Equals(\_1020.Hexbi(Conversion.Hex(number)), "10000")**, y es ahí donde demos hacer que siempre retorne **TRUE**, así que nos debería quedar **Return true**. Volvamos a nuestro CrackMe con la vista **IL**.



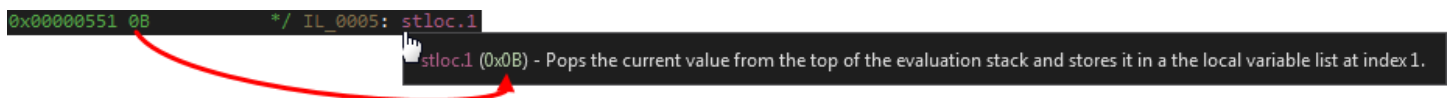
Yo no tengo muy claro todo lo que podemos entender e interpretar de la información mostrada en código **IL** o **CIL**, pero empezaremos con lo básico que este susodicho sabe y que con el mayor de los gusto comparte contigo. Empecemos mirando lo que tenemos en los **RECUADROS AZULES**, en el recuadro de la derecha tenemos las **INSTRUCCIONES** que se ejecutan para hacer una determinada tarea. Si tienes bases de Cracking con el

## Neófito Reversando .NET [Entrega #2][LUISFECAB]

<OlllyDBG> o similares puedes entender que estas instrucciones cumplen tareas similares como cargar un valor, moverlo, llamar a un procedimiento o retornar para finalizarlo, y así. Estas **INSTRUCCIONES** son representadas por un valor **HEXADECIMAL**, que es conocido como **OPCODE**, y esos valores **HEXADECIMALES** los podemos observar el recuadro de la izquierda que son los valores que componen el binario en disco y son estos valores los que en realidad cambiamos cuando Crackeamos o Parchamos una aplicación. Luego miraremos como entender lo que hace cada instrucción y su correspondiente **OPCODE**. Ahora miremos el **RECUADRO ROJO**, ocupa más de la mitad del procedimiento y solo está representando la última línea de código, **Return String.Equals(\_1020.Hexbi(Conversion.Hex(number)), "10000")**. Ahora vamos paso a paso por cada **INSTRUCCIÓN** para ir familiarizándonos con ellas. En realidad no es necesario saber a fondo todas las **INTRUCCIONES** para nuestros propósitos, lo importante es saber usar la correcta y en su lugar adecuado para que surta efecto. Miremos la primera instrucción.



La instrucción **ldsfld**, Pushea el valor de nuestra variable **\_1020.strPilla**, que sería nuestro **SERIAL** en el **STACK** o **PILA**, ojo no confundir con el **CALL STACK**. El **OPCODE** que representa a **ldsfld** es el **0x73**. El <dnSpy> es nuestro amigo vacan y al posicionarnos sobre una **INSTRUCCIÓN** este nos muestra su respectivo **OPCODE** y la tarea que realiza. Si miramos los valores **HEXADECIMALES** de esa **INSTRUCCIÓN** tenemos **7E09000004**. El primer **BYTE** **7E** es el **OPCODE** para **ldsfld**, y el resto de **BYTES** **09000004** nos representa una especie de dirección para acceder a la variable que almacena el valor, que para nosotros es nuestro **SERIAL** guardado en **\_1020.strPilla**. Los **.NET** tienen su lugar donde se guarda toda esta información; a medida que avancemos iremos conociendo toda su estructura que supongo que los grandes maestros del **.NET** si saben sacarle provecho a ese conocimiento. Y por último tenemos el **OFFSET** que nos apunta donde encontramos esos valores **HEXADECIMALES** y que con un Editor Hexadecimal podemos encontrar fácilmente. Sigamos con la siguiente **INSTRUCCIÓN**.



**stloc.1** recupera el valor almacenado en el tope del **STACK** y lo almacena en una determinada variable local, en este caso es la variable ubicada en el index 1 de una lista de variables, lo cual nos indica que tenemos más de una variable local declarada. El **OPCODE** que representa esta instrucción es el **0x0B**. Entonces habíamos puesto nuestro **SERIAL** en el **STACK** y ahora lo sacamos del **STACK** y lo pasamos a una variable. Se me pasó explicar esa parte de las variables locales que se muestran en esta vista **IL**.

## Neófito Reversando .NET [Entrega #2][LUISFECAB]

```
.locals init (
    [0] bool,
    [1] object
)

/* 0x0000054C 7E09000004 */ IL_0000: ldsfld    string 'CrackMe#2.vRS._1020'::strPilla
/* 0x00000551 0B      */ IL_0005: stloc.1
```

SE DECLARAN DOS VARIABLES LOCALES.

Como vemos el index 1 nos indica que almacenaremos nuestro **SERIAL** en la segunda variable local, la cual es declarada como **object**, este tipo de variables puede almacenar cualquier tipo de dato. Recordemos que en realidad el procesador ejecuta los procedimientos por pasos, y es por eso que primero pasó nuestro **SERIAL** al **STACK** y luego lo coge de este para pasarlo a la variable local.

```
0x00000552 1201      */ IL_0006: ldloc.s    V_1
ldloc.s (0x12) - Loads the address of the local variable at a specific index onto the evaluation stack, short form.
```

(local variable) object V\_1

**ldloc.s** va a leer la dirección donde se encuentra nuestra variable local y que ahora se representa con el nombre **V\_1**. El **OPCODE** de esta **INSTRUCCIÓN** es el **0x12**. Hasta aquí lo que se ha hecho es procesar nuestra variable para posteriormente ser utilizada en algún cálculo o procedimiento. Sigamos con la siguiente instrucción.

```
0x00000554 281A000006 */ IL_0008: call     int32 'CrackMe#2.vRS._1020'::Method_02(object&)
call (0x28) - Calls the method indicated by the passed method descriptor.
```

Ahora hacemos un **call**, que es bastante claro en lo que hace; esta **INSTRUCCIÓN** llama a un Método para ser ejecutado. Esta **call** va a ejecutar una función que retorna un Entero de 32 Bits, y sabemos eso porque lo primero que se especifica es la declaración del valor retornado, **int32**. Miremos ahora los valores **HEXADECIMALES**, **281A000006**; ya deberíamos saber que el **OPCODE** es el **0x28**, pero lo que quiero es profundizar en entender más es resto de los **HEXADECIMALES**, **1A000006**. Yo les dije que lo entendiéramos como si fuera una especie de dirección, lo cual pienso no está mal pero vamos a dejar esto un poco más claro, en realidad a estos **BYTES** se les llama **TOKEN**, el cual muestra la información donde podemos ubicar cualquier objeto como por ejemplo Clases, Métodos, Campos...etc. Pues todos estos objetos están organizados en una especie de Tabla que la podemos encontrar con los nombres "**MetaData Streams**" o "**Tables Stream**".

Method_010 : bool _1020 06 Method (40)										
RID	Token	Offset	RVA	ImplFlags	Flags	Name	Signature	ParamList	Info	
23	0x06000017	0x00000F00	0x2178	0	0x813	0xA0C	0xB9	5	get_Settings	
24	0x06000018	0x00000F0E	0x2230	0	0x16	0x5E	0x37	5	Method_03	
25	0x06000019	0x00000F1C	0x2340	0	0x16	7	0xC6	5	Method_01	
26	0x0600001A	0x00000F2A	0x2384	0	0x11	0x4C	0xCA	5	Method_02	
27	0x0600001B	0x00000F38	0x217F	0	0x11	0x5BF	0xD0	6	Hexbi	

TOKEN → 1A0000 06 → NÚMERO DE LA TABLA → 0x0600001A

ENTRADA EN LA TABLA. 0x1A (26)

EL TOKEN LO ENCONTRAMOS INVERTIDO EN LA VISTA IL. RECORDAR ESO PARA NO CONFUNDIRNOS.

## Neófito Reversando .NET [Entrega #2][LUISFECAB]

Estas Tablas siempre tendrán el mismo número de identificación, en donde guardarán cada tipo de objeto en su lugar, por ejemplo la tabla **0x06** siempre será la que tendrá almacenado todos los Métodos. Toda esta teoría, ahora no es que nos sea muy relevante para Crackear nuestro CrackMe, pero es bueno ir conociendo más a profundidad cómo es la lógica de funcionamiento de los **.NET** porque la realidad es que no siempre vamos a poder Reversar aplicaciones **.NET** sin protección, ya que es conocido por todos que existen muchas protecciones para los **.NET**. Supongo que ya muchos también conocen que así como hay tools para proteger, pues que también existen tools que quitan esas protecciones, y creo que la mejor es sin duda el [d4dot](#) pero que como todo no es 100% efectivo, y es ahí que entra en juego esta teoría que acabamos de ver a groso modo, porque por lo que he leído y podido entender, para poder enfrentarnos a protecciones complicadas y marrulleras toca entender cómo funcionan los **.NET** para poder quitar esas protecciones a mano. Listo, sigamos con más **INTRUCCIONES**.

```
0x00000559 07 */ IL_000D: ldloc.1
```



ldloc.1 (0x07) - Loads the local variable at index 1 onto the evaluation stack.

**ldloc.1**, que su **OPCODE** es **0x07**; cargará la Variable Local Index 1 dentro del **STACK**, y que nosotros ya conocemos, que es nuestro **SERIAL** que está guardado como **object**. Haber, ya debemos ir entendiendo que se hace esto primero porque seguro la **INSTRUCCIÓN** siguiente va a trabajar con nuestro **SERIAL** y para hacer eso lo debe pasar al **STACK**. Sigamos.

```
283B00000A */ IL_000E: call string [Microsoft.VisualBasic]Microsoft.VisualBasic.CompilerServices.Conversions::ToString(object)
```

Empecemos por el **OPCODE** que sería **0x28**, que representa la **INSTRUCCIÓN** **call**. Como vemos tenemos, **string**, eso nos indica que este Método es una Función que retornará una String, y no es otra cosa que nuestro **SERIAL**, recordemos que estaba almacenado como **object**.

```
0x0000055F 8009000004 */ IL_0013: stsfld string 'CrackMe#2.vRS._1020'::strPilla
```



stsfld (0x80) - Replaces the value of a static field with a value from the evaluation stack.

**stsfld**, esta **INSTRUCCIÓN** reemplaza el valor que tiene nuestra variable **1020.strPilla** por el valor que tiene el **STACK**, y si han seguido con mucha atención las **INSTRUCCIONES** hasta aquí, pueden deducir que en realidad no hizo nada con eso porque **1020.strPilla** tiene almacenado nuestro **SERIAL** y el valor almacenado en el **STACK** es también el **SERIAL**, así que cambió por lo mismo. No olvidemos su **OPCODE** que sería el **0x80**. Bien, ahora iniciaremos con la parte que más nos interesa y es donde se empieza a definir el retorno de la Función.

```
0x00000564 283C00000A */ IL_0018: call string [Microsoft.VisualBasic]Microsoft.VisualBasic.Conversion::Hex(int32)
0x00000569 2818000006 */ IL_001D: call string 'CrackMe#2.vRS._1020'::Hexbi(string)
```

El primer **call** retorna una String que es utilizada como parámetro en el segundo **call**, que también retorna una String la cual queda alojada en el **STACK**. Siempre hablamos del **STACK** pero no hay forma de poder visualizarlo en el **<dnSpy>** o puede ser que se



## Neófito Reversando .NET [Entrega #2][LUISFECAB]

pueda pero que yo no sepa como verlo, vaya uno a saber. Miremos la siguiente **INSTRUCCIÓN**.

```
0x0000056E 72F5000070 */ IL_0022: ldstr "1000"
```

ldstr (0x72) - Pushes a new object reference to a string literal stored in the metadata.

La **INSTRUCCIÓN** **ldstr** mueve una String que esta almacenada en las Tablas MetaData. Antes de seguir, recordemos entonces que en el **STACK** los dos valores superiores serían las dos últimas Strings que fueron puestas por el último **call** y **ldstr**.

```
0x00000573 283D00000A */ IL_0027: call bool [mscorlib]System.String::Equals(string, string)
```

Ahí podemos entender el motivo de pasar primero dos String el **STACK** y es porque este **call** va a comparar esas dos String, **Equals(string, string)**, y lo que retorna un valor Boolean, **TRUE** o **FALSE**, que seguramente, dependiendo de lo retornado tomará uno u otro camino.

```
0x00000578 2C04 */ IL_002C: brfalse.s IL_0032
0x0000057A 17 */ IL_002E: ldc.i4.1
0x0000057B 0A */ IL_002F: stloc.0
0x0000057C 2B02 */ IL_0030: br.s IL_0034
0x0000057E 16 */ IL_0032: ldc.i4.0
0x0000057F 0A */ IL_0033: stloc.0
0x00000580 06 */ IL_0034: ldloc.0
0x00000581 2A */ IL_0035: ret
```

brfalse.s (0x2C) - Transfers control to a target instruction if value is false, a null reference, or zero.

SI ES FALSE TOMA EL  
SALTO HASTA IL\_0032 DE  
LO CONTRARIO SIGUE  
DERECHO EN IL\_002E.

Tocó el resto de las **INSTRUCCIONES** porque todas están ligadas a lo que ocurre en **2C04 \*/ IL\_002C: brfalse.s IL\_0032**. **brfalse.s** salta si tenemos **FALSE**, un resultado nulo o cero; y es momento de aclarar que cuando hablamos de un valor Boolean que es **TRUE** o **FALSE**, pero la realidad es que el valor real es **0 (FALSE)** y **1 (TRUE)**. Miremos los **BYTES** que componen esa **INSTRUCCIÓN**, **2C04**. **0x2C** sería el **OPCODE** y **0x04** serían la cantidad de **BYTES** que saltaría desde la siguiente posición **IL\_002E**. La instrucción **brfalse.s**, que estamos analizando tiene el mismo valor del **OPCODE**, lo que es una simpática coincidencia, no dejarse confundir por eso. Entonces sumamos **0x4C+0x2E=0x32 (IL\_0032)** que nos dejaría en la posición de tomar el salto. Bien, vamos a suponer que tomamos el salto para llegar a la **INSTRUCCIÓN** **IL\_0032**.

```
0x0000057E 16 */ IL_0032: ldc.i4.0
```

ldc.i4.0 (0x16) - Pushes the integer value of 0 onto the evaluation stack as an int32.

**ldc.i4.0** agrega un **0** al **STACK** y este valor determina que retornamos **FALSE**, recordemos **0=FALSE**. Entonces podemos suponer que para lograr obtener un retorno como **TRUE** se debe agregar **1**; luego veremos cómo hacer el cambio, por ahora nos enfocaremos en ir entendiendo las **INSTRUCCIONES** en **IL (CIL)**. Nos quedan tres **INSTRUCCIONES** más, revisemos dos de una para que notemos que solo hacen unos movimientos para terminar en lo mismo.



## Neófito Reversando .NET [Entrega #2][LUISFECAB]

```
.locals init (  
    [0] bool,  
    [1] object  
)
```

RECORDEMOS QUE  
TENEMOS NUESTRAS 2  
VARIABLES LOCALES.

```
0x0000057F 0A      */ IL_0033: stloc.0  
0x00000580 06      */ IL_0034: ldloc.0
```

stloc.0 (0x0A) - Pops the current value from the top of the evaluation stack and stores it in the local variable list at index 0.

ldloc.0 (0x06) - Loads the local variable at index 0 onto the evaluation stack.

**stloc.0** que tiene como **OPCODE** **0x0A**, saca el último valor del **STACK** y lo almacena en la Variable Local Index 0, la cual será una variable **bool**, y ese valor no es otro que nuestro **0** (**FALSE**) que fue agregado al **STACK** por **ldc.i4.0**. Luego sigue la **INSTRUCCIÓN** **ldloc.0** (**OPCODE: 0x06**) que lee el valor de la Variable Local Index 0, que está dentro del **STACK**. Como vemos solo movimos nuestro **0** (**FALSE**) para terminar en lo mismo, por lo poco que he ido aprendiendo creo que solo con colocar nuestro **0** (**FALSE**) y después el retorno **ret**, obtenemos el mismo resultado, así que sobraría esas dos **INSTRUCCIONES**, **stloc.0** y **ldloc.0**, es más hasta nos sobraría la Variable Local Index 0. Miremos nuestra última **INSTRUCCIÓN**.

```
0x00000581 2A      */ IL_0035: ret
```

ret (0x2A) - Returns from the current method, pushing a return value (if present) from the callee's evaluation stack onto the caller's evaluation stack.

El **ret**, retorno del módulo presente que está en ejecución y si existe un valor para retornar, se retorna con ese valor. El **OPCODE** es el **0x2A**. Ahí terminamos de recorrer todas las **INSTRUCCIONES** del **1020.Method\_01()** y pudimos darnos cuenta donde se determinaba nuestro retorno como **FALSE**, y también deducir el cambio que debemos hacer. Pero antes de proceder a aprender hacer el cambio nos quedó pendiente analizar el código si no tomáramos el salto.

```
0x00000578 2C04      */ IL_002C: brfalse.s IL_0032  
0x0000057A 17      */ IL_002E: ldc.i4.1  
0x0000057B 0A      */ IL_002F: stloc.0  
0x0000057C 2B02      */ IL_0030: br.s IL_0034  
  
0x0000057E 16      */ IL_0032: ldc.i4.0  
0x0000057F 0A      */ IL_0033: stloc.0  
  
0x00000580 06      */ IL_0034: ldloc.0  
0x00000581 2A      */ IL_0035: ret
```

brfalse.s (0x2C) - Transfers control to a target instruction if value is false, a null reference, or zero.

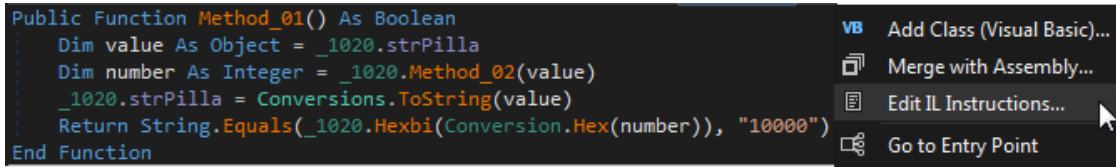
SI ES FALSE TOMA EL  
SALTO HASTA IL\_0032 DE  
LO CONTRARIO SIGUE  
DERECHO EN IL\_002E.

La primera **INSTRUCCIÓN** que encontramos es la **ldc.i4.1**, la cual pone en el **STACK** un **1** que vendría siendo nuestro **TRUE** de retorno. Su **OPCODE** es **0x17**, que es el que nos interesa porque este nos da **TRUE**. Luego toma el valor puesto en el **STACK** que es nuestro **1** (**TRUE**) y lo almacena en nuestra Variable Local Index 0 con **stloc.0**; ya se pueden dar cuenta que va hacer ese movimiento de nuestro **1** (**TRUE**) pero para poder terminarlo debe saltar las **INSTRUCCIONES** para evitar las dos **INSTRUCCIONES** que se ejecutan cuando tomamos el salto, y para hacer eso toma un salto incondicional con **br.s**. Esto ya lo sabemos, que los **BYTES** determinan el salto, **2B02**, **0x2B** **OPCODE** de **br.s** y **0x02** longitud del salto. Como vemos, saltamos a la **INSTRUCCIÓN** **ldloc.0** para terminar de leer nuestro **1** (**TRUE**) en el **STACK** y retornar (**ret**) con él.

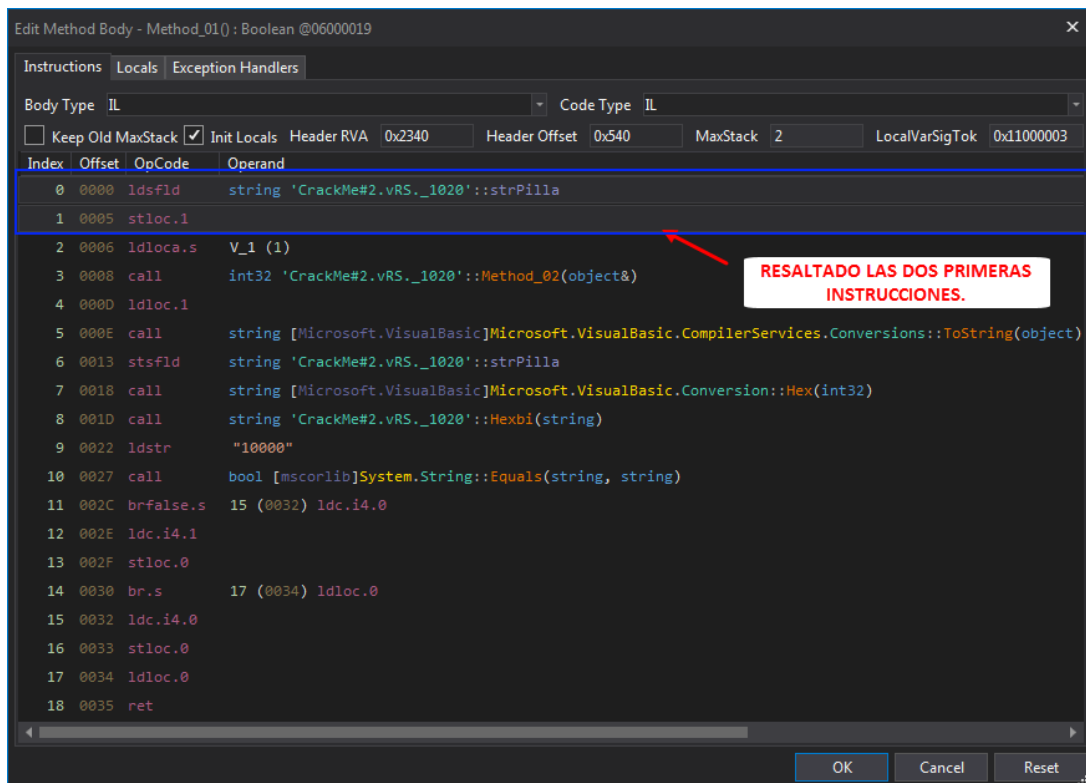
## Neófito Reversando .NET [Entrega #2][LUISFECAB]

Existen muchos lugares para hacer el cambio para siempre retornar **1** (**TRUE**). De acuerdo al lugar podemos hacer el cambio más fácil y sencillo. Haremos un primer cambio en el inicio de nuestra función **\_1020.Method\_01()**. Así que abrimos nuestro **EDITOR IL**.

```
Public Function Method_01() As Boolean
    Dim value As Object = _1020.strPilla
    Dim number As Integer = _1020.Method_02(value)
    _1020.strPilla = Conversions.ToString(value)
    Return String.Equals(_1020.Hexbi(Conversion.Hex(number)), "10000")
End Function
```

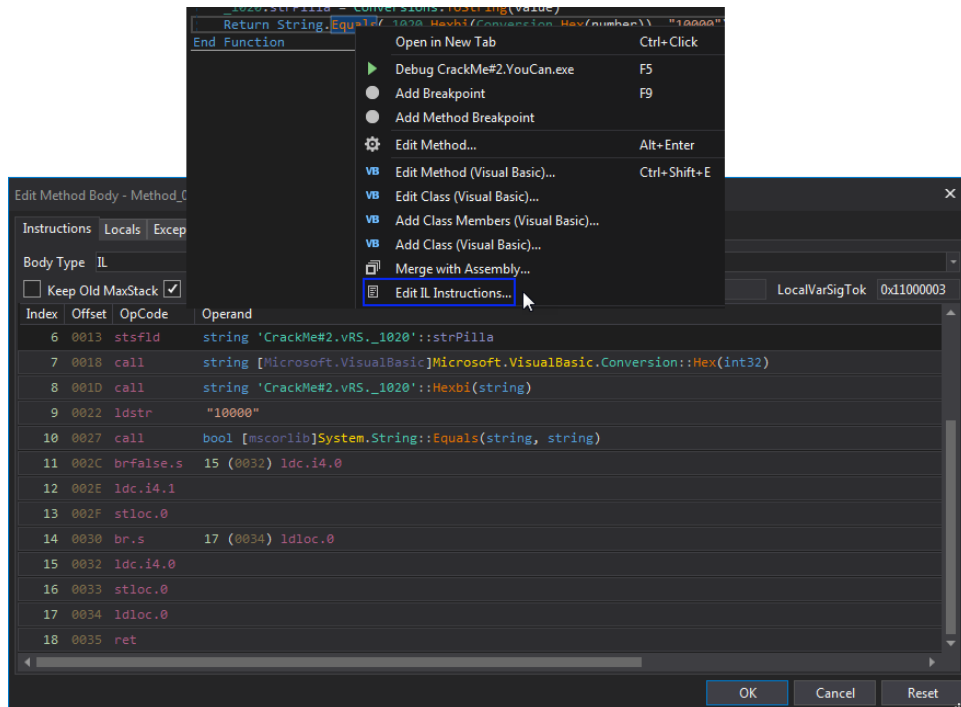


Podemos ver que lo tengo en la vista de código **Visual Basic** y con <Clic Derecho-> **Edit IL Instructions...**> lo abrimos. Lo que quiero mostrar es que el **EDITOR IL** es accesible en cualquiera de las tres vistas.

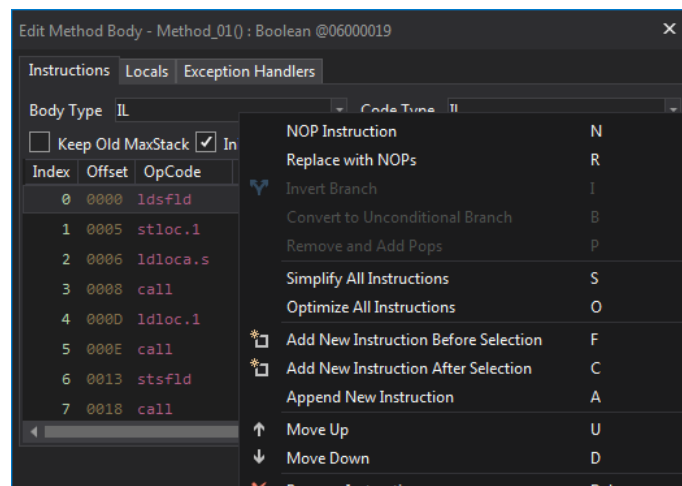


Podemos ver que tenemos como resaltado las dos primeras **INSTRUCCIONES** y eso se debe a que esas **INSTRUCCIONES** son las que componen mi primer línea de código que sería **Dim value As Object = \_1020.strPilla**. Entonces, dependiendo del lugar don carguemos el **EDITOR IL**, este resaltará las **INSTRUCCIONES** que conforman esa línea de código. Miremos el siguiente ejemplo para ver cómo se nos resaltan las **INSTRUCCIONES**.

## Neófito Reversando .NET [Entrega #2][LUISFECAB]

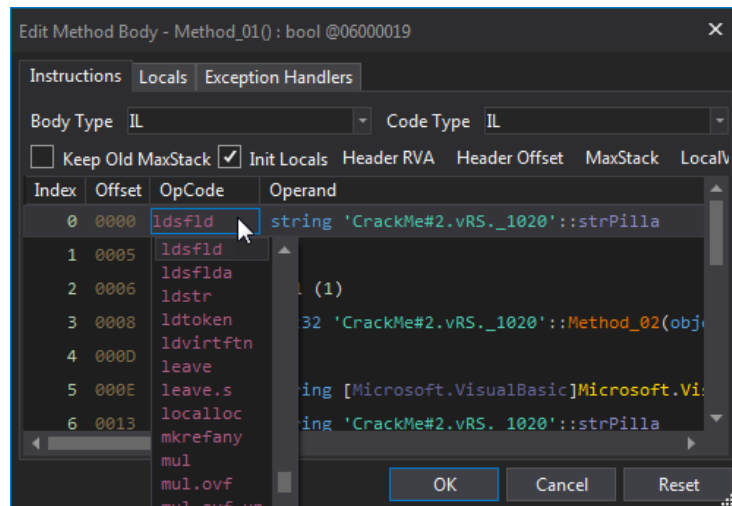


Escogimos cargar el **EDITOR IL** desde nuestra última línea de código y como vemos están resaltadas todas las **INSTRUCCIONES** que componen a **Return String.Equals(\_1020.Hexbi(Conversion.Hex(number)), "10000")**. Sigamos amigos, en el **EDITOR IL** podemos cambiar todo el código o agregar nuevas **INSTRUCCIONES**.

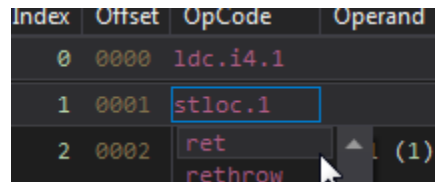
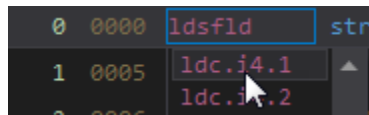


Con <**Clic Derecho**> tenemos las opciones y sus accesos rápidos, podemos nopear la **INSTRUCCIÓN** seleccionada con <**NOP Instruction (N)**> para que no se ejecute nada, podemos agregar nuevas **INSTRUCCIONES** o mover la que ya existe. Ya es cuestión de ti, entusiasta Cracker de .NET que vallas probando todas las posibilidades que ofrece el **EDITOR IL**. No tengas miedo en hacer algo malo porque el **EDITOR IL** te ofrece el botón "**Reset**" para quitar todos los cambios. Listo, ahora si nuestros cambios. Recordemos que debemos hacer que retorne **1 (TRUE)**, y el **1** lo ponemos con **ldc.i4.1**, paso seguido el retorno, **ret**. Entonces sería dos **INSTRUCCIONES**.

## Neófito Reversando .NET [Entrega #2][LUISFECAB]



La columna donde están las **INSTRUCCIONES** se llama **OpCode**, y muy lógico porque la realidad que sabemos es que cambiaremos **BYTES**.

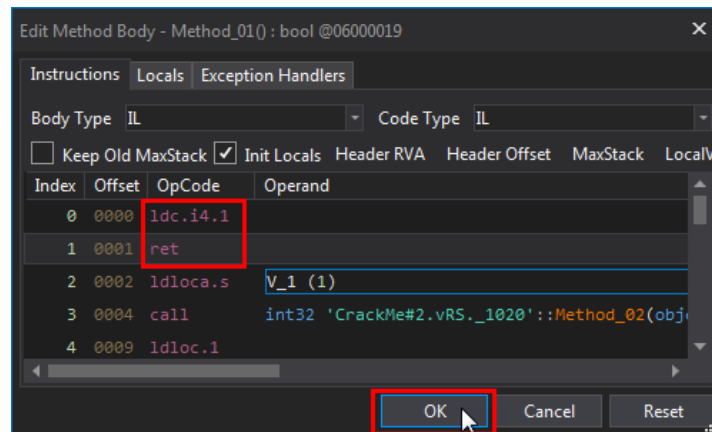


Para realizar el cambio solo seleccionamos la **INSTRUCCIÓN** que deseas cambiar, se desplegarán todas las **INSTRUCCIONES** u **OPCODES** que podemos utilizar. Nosotros buscaremos la instrucción **ldc.i4.1**, para poder colocar nuestro 1 en el **STACK** y luego lo mismo pero con **ret**.

Index	Offset	OpCode	Operand
0	0000	ldc.i4.1	
1	0001	ret	
2	0002	ldloca.s	V_1 (1)
3	0004	call	int32 'CrackMe#2.vR
4	0008	ldloc.1	

Hicimos los cambios en el inicio. Aquí debemos entender que todas las demás **INSTRUCCIONES** nunca se van a ejecutar, estas quedan sobrando como si fuera un código basura porque la función termina con el **ret** que pusimos.

## Neófito Reversando .NET [Entrega #2][LUISFECAB]



Guardamos nuestros cambios con "OK" para ver que tanto cambió nuestro código en la vista **Visual Basic**.

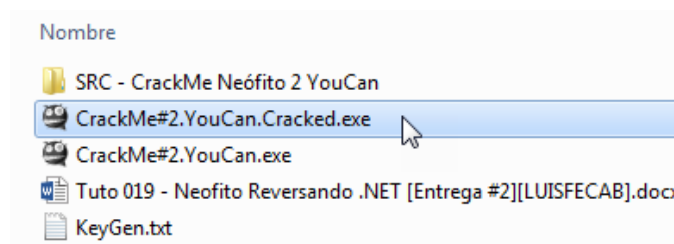
MSIL OpCode Table v1.0 (miroslav.stampar@gmail.com)

Name	Value	Size	Info	OpCodeType	Fl
bne.un	40	1	Branches to specified offset if value1 isnt equal to...	Macro	Co
bne.un.s	33	1	Branches to specified offset if value1 isnt equal to...	Macro	Co
box	8C	1	Converts value type to object reference	Primitive	Ne
br	38	1	Unconditional branch to specified offset	Primitive	Br
br.s	2B	1	Branches to specified offset	Macro	Br
break	01	1	Inform the debugger that a breakpoint has been reached	Primitive	Br
brfalse	39	1	Branches to specified offset if value on stack is false	Primitive	Co
brfalse.s	2C	1	Branches to specified offset if value on stack is false	Macro	Co
brtrue	3A	1	Branches to specified offset if value on stack is true	Primitive	Co
brtrue.s	2D	1	Branches to specified offset if value on stack is true	Macro	Co
call	28	1	Calls a method	Primitive	Ca
calli	29	1	Calls method indicated by address on stack; stack...	Primitive	Ca
callvirt	6F	1	Calls virtual method of obj	Objmodel	Ca
castclass	74	1	Casts obj to class	Objmodel	Ne
ceq	FE01	2	Compares equality of two values on stack; pushes...	Primitive	Ne

Por fortuna contamos con una tools que se llama <MSIL OpCode Table v1.0> que no es más que una tablita con todas las **INTRUCCIONES** y sus respectivos **OPCODES**.

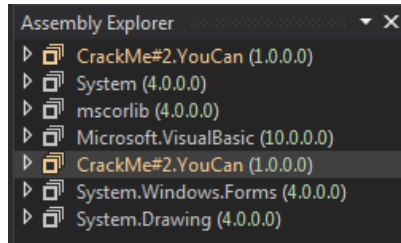
```
' Token: 0x06000019 RID: 25 RVA: 0x00002340 File Offset: 0x00000540
Public Function Method_01() As Boolean
    Return True
End Function
```

Jajaja!!!! Tremendo cambio, nuestro <dnSpy> es muy inteligente y supo que el resto de las **INTRUCCIONES** eran sobras que desechó, y compiló solo el código funcional. Guardemos los cambios con <File->Save Module>. Lo guardaré con otro nombre porque al original le falta mucho más análisis, le pondré <CrackMe#2.YouCan.Cracked.exe>.



Lo abrimos con nuestro inteligente amigo el <dnSpy> para ver y probar solo esa parte y ver si evitamos al "CHICO MALO".

## Neófito Reversando .NET [Entrega #2][LUISFECAB]



Recuerden lo aprendido en la **Entrega #1** y no confundirse de **TARGE**. Ponemos un **<BREAKPOINT>** donde se decide el salto del **"CHICO MALO"**.

```
3 Private Sub btnProbar_Click(sender As Object, e As EventArgs)
4     _1020.strPilla = Me.txtSerial.Text
5     _1020.strPnum = "8569214530741072"
6     If Not _1020.Method_01() Then
7         Interaction.MsgBox("Serial incorrecto.", MsgBoxStyle.Exclamation, "Error")
8         Return
9     End If
10    _1020.Method_03()
11 End Sub
```

Recordemos que el cambio lo hicimos en **\_1020.Method\_01()** para que nos retornara **TRUE**. Probemos con nuestro **SERIAL "muydifícil"** para detenernos ahí en el **<BREAKPOINT>**.

```
110 Private Sub btnProbar_Click(sender As Object, e As EventArgs)
111     _1020.strPilla = Me.txtSerial.Text
112     _1020.strPnum = "8569214530741072"
113     If Not _1020.Method_01() Then
114         Interaction.MsgBox("Serial incorrecto.", MsgBoxStyle.Exclamation, "Error")
115         Return
116     End If
117     _1020.Method_03()
118 End Sub
```

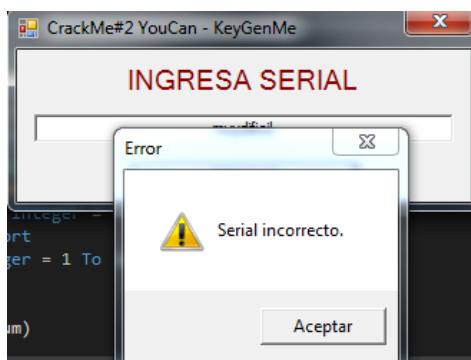
Ahí nos detuvimos. Vamos a hacer una prueba de fe y vamos a tracear con **<F10>** para no entrar al **\_1020.Method\_01()**, lo importante es ver si no entramos al **If** y de esa forma evitamos al **"CHICO MALO"**. Pummmm! Presionemos **<F10>**.

```
113 If Not _1020.Method_01() Then
114     Interaction.MsgBox("Serial incorrecto.", MsgBoxStyle.Exclamation, "Error")
115     Return
116 End If
117 _1020.Method_03()
118 End Sub
```

100 %

Name	Value
CrackMe#2.vRS_1020/0x0200000A/Method_01/0x06000019/ returned	true

Pura vida, evitamos al **"CHICO MALO"** y nos encontramos posicionados ahora en **\_1020.Method\_03()**. En **LOCALS** podemos ver que **\_1020.Method\_01()** nos retornó **true**. Listo, ya tenemos la vía libre para ejecutarlo de una, así que lo continuamos con **<F5>**.



## Neófito Reversando .NET [Entrega #2][LUISFECAB]

---

Nooooo! Que va, uno todo contento porque se pensaba inocentemente que esa era el único "CHICO MALO", y resulta que hay más de estos mensajes detestables.

Bueno amigos, lo vamos a dejar hasta aquí. En una próxima Entrega lo seguimos analizando.



## PARA TERMINAR

Creo que podemos decir que ya tenemos nuestra base de conocimiento para que puedan hacer sus primeros análisis de código **IL** por cuenta propia, y así se animen a resolver este CrackMe. Este reto lo subí ya hace días al canal de **@PeruCrackers** y tenía la esperanza que alguno se animara a resolverlo. Debo entender que el tiempo es corto para hacer estas cosas pero en el grupo hay amigos con gran nivel que pensaba lo iban a resolver, pero me quedé esperando.

De nuevo los exhorto, los animo a que lo resuelvan, yo sé que pueden; y para aquellos que toman estas Entregas como punto de partida, pues que pongan en práctica lo que hemos aprendido para ver si lo pueden lograr, pueden intentar hacer cambios como lo que aprendimos aquí para que acepte cualquier **SERIAL**, o mejor aún, sacar el **SERIAL** y hasta programar el **KeyGen**.

Terminar esta **Entrega #2** me tomó mucho tiempo, la verdad es que escribir estos tutoriales enfocados como si fueran un curso es más tedioso de escribir que un tutorial independiente. Veremos hasta dónde me llegan las ganas y el impulso.

Me despido de ti, mi amigo lector esperando que todo mi esfuerzo no sea en vano.

**@LUISFECAB**