6.3.—Transformaciones básicas del código

En muchas ocasiones tendrás suficiente con el contenido del listado de desensamblado generado por IDA. En otros quizá no. En el caso en que los archivos analizados se contradigan mucho con los ejecutables generados por los compiladores más comunes, puedes encontrarte en que necesites tener más control sobre los procesos de análisis y vistas del desensamblado. Esto será así sobretodo, si quieres analizar código ofuscado o archivos que utilicen un formato de archivo hecho a medida, desconocido por IDA.

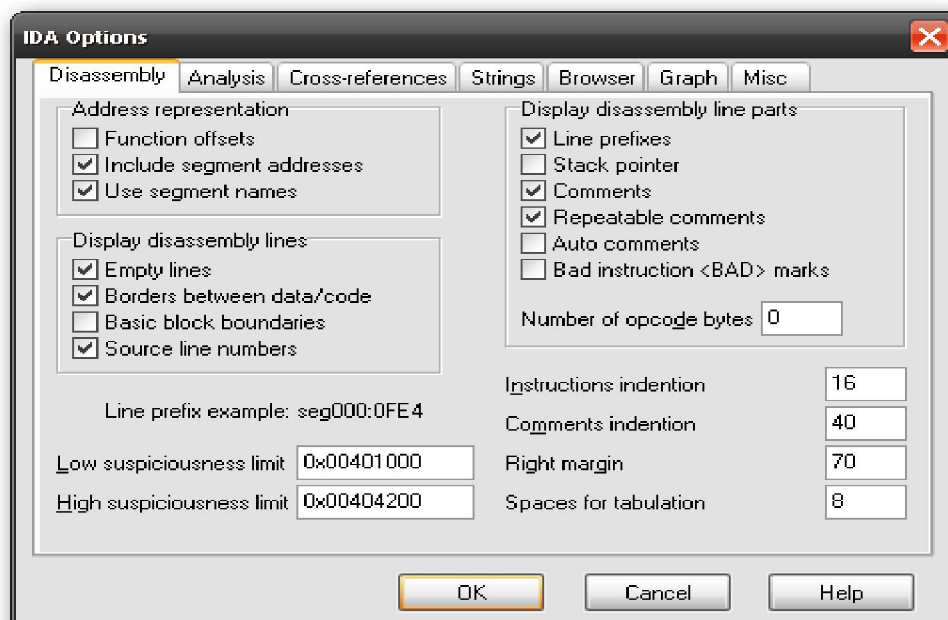
Las transformaciones de código que nos permite IDA son las siguientes:

** **Convertir datos en código**
** **Convertir código en datos**
** **Designar a una secuencia de instrucciones como una función**
** **Cambiar las direcciones de inicio o final de una función existente.**
** **Cambiar el formato de vista de los operandos de instrucciones**

La forma o cantidad en su utilización dependerá de una amplia variedad de factores y preferencias personales. En general, si un binario es muy complejo o también si a IDA no le son familiares las secuencias generadas por el compilador utilizado para construir el binario, IDA se encontrará con más problemas en la fase de análisis y por lo tanto necesitaremos ajustar manualmente el desensamblado del código. Veamos como.

6.3.1.—Opciones de vista de código

La transformación más simple realizada en el listado de desensamblado supone recomponer una gran cantidad de información generada por IDA en cada línea de desensamblado. Cada línea del desensamblado puede considerarse como una parte de una colección, a la que IDA pueda referirse, no debería sorprendernos que los elementos como; etiquetas, mnemónicos y operandos, estén siempre presentes en las líneas de desensamblado. Lo que si es nuevo para nosotros es que podemos seleccionar elementos adicionales para cada línea de desensamblado, realizando la siguiente acción **Options > General** y seleccionar la solapa **Disassembly**, figura abajo.



En la sección **Display disassembly line parts**, derecha superior del diálogo, se nos muestran distintas opciones para caracterizar las líneas de desensamblado. En la vista texto del desensamblado las, **Line prefixes, Comments y Repeatable comments** están seleccionados por defecto. Ahora vamos a describir cada elemento que se pueda mostrar en el listado.

Line prefixes

El prefijo de línea de cada línea de desensamblado es la parte que muestra **sección:dirección**. Si deseleccionamos esta opción provocaremos la desaparición del prefijo de cada línea de desensamblado, por defecto en la vista gráfica. Para ver esta opción deshabilitada se muestra una parte de listado de **tora**.

```
; void tora (int j, int k);
; Attributes: bp-based frame

tora            proc near               ; CODE XREF: sub_401090+2A↓p

var_8           = dword ptr -8
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
Las tres líneas siguientes verifican si j < k
                sub     esp, 8
                mov     eax, [ebp+arg_0]
                cmp     eax, [ebp+arg_4]
                jge     short loc_40106C ; El comentario repetitivo se propag
                mov     [esp+8+var_8], offset aTheSecondParam ; El segundo pa
                call    printf
                jmp     short locret_40108E ; Salto al final de la función
```

Stack Pointer

IDA ejecuta un análisis exhaustivo en cada función con el fin de obtener los cambios realizados al puntero de pila (**stack pointer**). Este análisis es esencial para comprender el esquema del **stack frame** de cada función. Seleccionar dicha opción causa que IDA nos muestre los cambios relativos del puntero de pila a lo largo de cada función. No obstante hay que reconocer que se producen discrepancias según el acuerdo de llamada utilizado, por ejemplo IDA no comprenderá una función que utilice **stdcall**, o también manipulaciones inusuales del puntero de pila. El rastro del puntero de pila en una función se muestra en la figura siguiente.

```
.text:00401050 000              push    ebp
.text:00401051 004              mov     ebp, esp
.text:00401053     Las tres líneas siguientes verifican si j < k
.text:00401053 004              sub     esp, 8
.text:00401056 00C              mov     eax, [ebp+arg_0]
.text:00401059 00C              cmp     eax, [ebp+arg_4]
.text:0040105C 00C              jge     short loc_40106C ; El comen
.text:0040105E 00C              mov     [esp+8+var_8], offset aTheS
.text:00401065 00C              call    printf
.text:0040106A 00C              jmp     short locret_40108E ; Salto
```
Program control flow

En la figura podemos observar como el puntero de pila a cambiado de **cero** a **cuatro** bytes después de la primera instrucción y llega a un total de **0xC** en la cuarta instrucción. Si observamos toda la función **tora** veremos que el final el puntero de pila es restaurado a su valor inicial, un cambio relativo de **cero** bytes. Siempre que IDA se

encuentre con una función con una declaración de retorno y detecte que el valor del puntero de pila no es cero, se realiza una condición de error y la instrucción es resaltada en rojo. En algunas ocasiones, esto podría ser un intento deliberado para frustrar el análisis automático. En otros casos podría tratarse de que el compilador utilice prólogos y epílogos los cuales IDA no pueda analizar con exactitud.

Comments y repeatable comments
Deseleccionar esta opción inhabilita la vista de los comentarios corrientes y repetitivos. Este caso se utiliza para descongestionar la vista del listado de desensamblado.

Auto comments
Como ya hemos dicho IDA puede comentar algunos tipos de instrucciones. Esto puede servir como recordatorio en cuanto a cómo se comportan algunas instrucciones en particular. No se añaden comentarios a instrucciones normales como x86 **mov**. Ejemplo

```
.text:00401065 00C                call    printf
.text:0040106A 00C                jmp     short locret_40108E ; Salto al final de la función
```

Los comentarios de usuario tienen prioridad sobre los comentarios auto; por lo tanto si quieres ver el auto comentario de una línea en la que exista un comentario de usuario tendrás que quitar los comentarios que hayas añadido, corrientes o repetititvos.

Bad instruction <BAD> marks
IDA puede señalar instrucciones correctas para el procesador pero que no son reconocibles por algunos ensambladores. En esta categoría se pueden incluir las instrucciones de CPU no documentadas. En tales casos IDA desensamblará la instrucción como una secuencia de bytes de datos y mostrará la instrucción no documentada con el prefijo **<BAD>**, en un intento de generar un desensamblado que puedan manejar la mayoría de desensambladores. Si necesitas más información con respecto a las señales <BAD>, buscar en Help file.

Number of opcode bytes
La mayoría de los desensambladores son capaces de generar listados de archivos mostrando los bytes del lenguaje máquina generado emparejados con las instrucciones de lenguaje ensamblador derivadas de estos. IDA nos permite ver los bytes del lenguaje máquina asociados a cada instrucción sincronizando una vista hexadecimal con la vista del listado de desensamblado. Opcionalmente podemos ver los bytes del lenguaje máquina mezclados con las instrucciones de ensamblador, especificando el número de bytes de lenguaje máquina que IDA mostrará para cada instrucción.

Esta acción es fácil cuando se desensambla código de procesadores con tamaño de instrucción fijo, pero es más difícil para procesadores de longitud de instrucción variable como puede ser el x86, cuyas instrucciones tienen un rango de uno hasta doce bytes de tamaño. A pesar de dicha longitud, IDA reservará el espacio necesario para mostrar el número de bytes que le especifiquemos, mostrándolos a la derecha de cada línea. En el siguiente ejemplo hemos habilitado el número de opcodes bytes a **5.** Figura abajo. El símbolo + de la línea mostrada a continuación indica que la instrucción es más

```
.text:0040105E 00C C7 04 24 00 20+            mov     [esp+8+var_8], offset
```

larga que los bytes especificados para mostrarse.

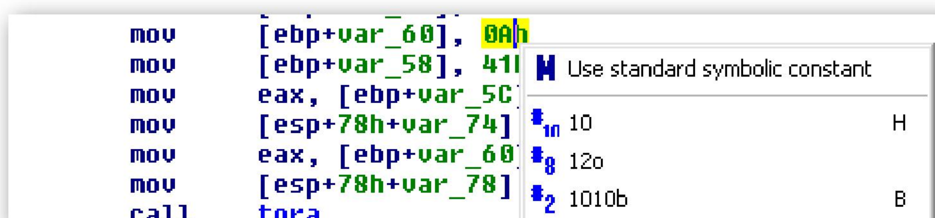
```
.text:00401050 000 55                    push    ebp
.text:00401051 004 89 E5                 mov     ebp, esp
.text:00401053            Las tres líneas siguientes verifican si j < k
.text:00401053 004 83 EC 08              sub     esp, 8
.text:00401056 00C 8B 45 08              mov     eax, [ebp+arg_0]
.text:00401059 00C 3B 45 0C              cmp     eax, [ebp+arg_4]
.text:0040105C 00C 7D 0E                 jge     short loc_40106C ; El coment
.text:0040105E 00C C7 04 24 00 20+       mov     [esp+8+var_8], offset aTheSe
.text:00401065 00C E8 46 01 00 00        call    printf
.text:0040106A 00C EB 22                 jmp     short locret_40108E ; Salto
```

Además de estas especificaciones, podemos ajustar también en el diálogo de **Disassembly** los valores de indentación y márgenes, área en la parte derecha inferior. Cualquier cambio de todas estas opciones sólo afectan a la base de datos actual. Para habilitarlas de forma global, cada una de estas opciones están guardadas en el archivo principal de configuración **Archivos de Programa\IDA\cfg\ida.cfg**.

6.3.2.—Formato de los operandos en las instrucciones

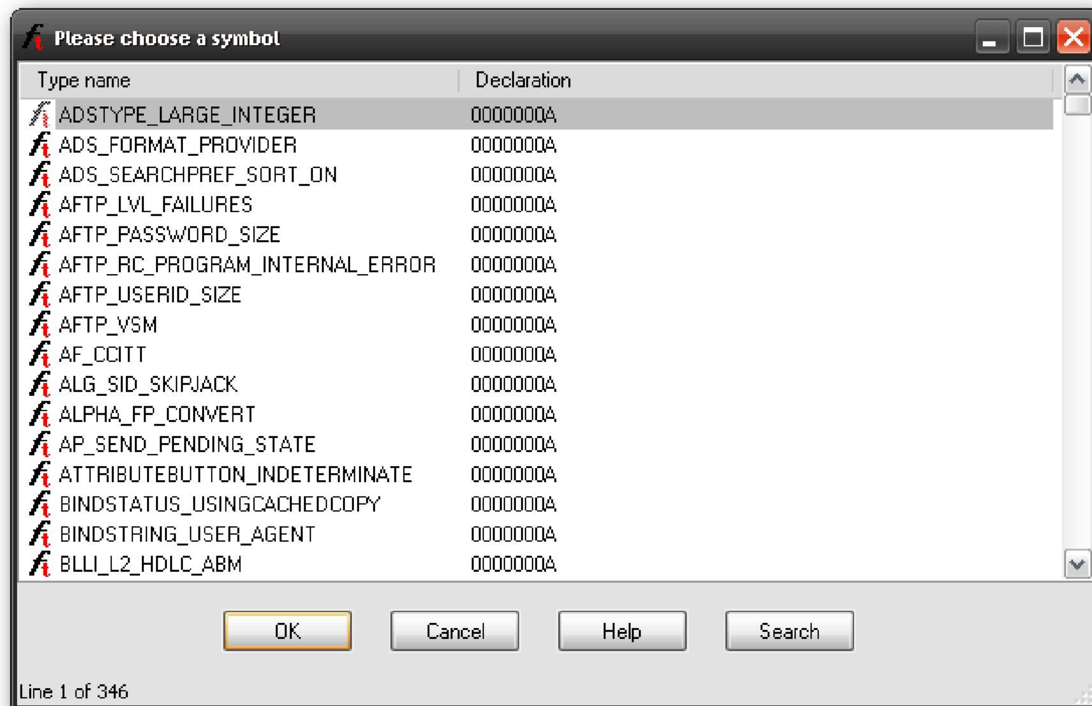
Durante el proceso de desensamblado, IDA toma algunas decisiones de cómo mostrará el formato de los operandos asociados con cada instrucción. Las decisiones mayores, generalmente se deben a como formatear distintas constantes enteras utilizadas por una gran variedad de instrucciones. Entre otras, estas constantes pueden representar; Offset relativos a instrucciones de salto y de llamada, direcciones absolutas de variables globales, valores que se utilizarán en operaciones aritméticas o constantes definidas por el programador. Con el fin de realizar un desensamblado más comprensible, IDA intenta utilizar nombres simbólicos antes que números, siempre que sea posible. En algunos casos, el formato se decide basándose en el contexto de la instrucción desensamblada, como puede ser una instrucción **call**; en otros casos, la decisión está basada con los datos que se están utilizando, como puede ser al acceso a una variable global o a un Offset en un stack frame. En algunos otros casos, no puede ser especificado el contexto exacto en el cual la constante será utilizada. Cuando alguno de estos casos sucede, la constante asociada normalmente es formateada como una constante hexadecimal.

Si por alguna razón inconfesable le tienes fobia a la notación hexadecimal, bienvenido seas a las características de formato para operandos. Si haces click derecho en cualquier constante del desensamblado se abrirá un menú de contexto similar al mostrado seguidamente.



En este caso, el menú de opciones que se nos muestra al seleccionar **0Ah** nos proporciona la opción de reformatear el valor como decimal, octal o binario. Además si por casualidad, como el valor siguiente del listado, dicha constante fuera un carácter ascii imprimible, tienes la opción de presentarlo en formato de carácter constante. En cualquier caso el menú mostrado es el mismo.

En muchos casos, los programadores utilizan constantes nombradas en su código fuente. Dichas constantes pueden ser el resultado de declaraciones tipo **#define** o equivalentes a estas, o pertenece a un conjunto de constantes numeradas. Por desgracia, una vez que el compilador ha finalizado el código fuente, no es posible determinar si la fuente utilizó una constante simbólica o literal o una constante numérica. IDA contiene un gran catálogo de constantes nombradas asociadas con algunas de las librerías más comunes como **C standard library** o la **Windows API**. A este catálogo podemos acceder utilizando la opción **Use standard symbolic constant** del menú contextual asociado a cualquier valor constante. Si seleccionamos dicha opción en la constante **0Ah**, nos mostrará el siguiente diálogo para elegir símbolo.



El diálogo está rellenado por una lista interna de constantes, filtradas de acuerdo al valor del formato de la constante. En este caso podemos ver todas las constantes conocidas por IDA que pueden encajar con el valor **0Ah**. Por ejemplo si determinásemos que nuestro valor será utilizado como constante que definirá la longitud de un buffer, elegiríamos **AFTP_PASSWORD_SIZE** y la línea de desensamblado se vería así:

```
.text:0040109F        mov     [ebp+var_5C], eax
.text:004010A2        mov     [ebp+var_60], AFTP_PASSWORD_SIZE
.text:004010A9        mov     [ebp+var_58], 41h
```

El listado de **standard constants** es una manera útil para determinar si una constante en particular puede estar asociada con un nombre conocido y ahorrarnos el leer la documentación APÎ buscando potenciales coincidencias.

6.3.3.—Manipular funciones

Existen varias razones para querer manipular funciones después de que se hayan completado los autoanálisis iniciales. En algunos casos, como cuando IDA no logra localizar una llamada a una función, con lo cual dicha función no se reconocerá, es obvio que no tendremos ninguna opción más, otro caso cuando IDA no consigue localizar correctamente el final de una función. En dichos casos necesitaremos una

intervención manual por nuestra parte para corregir el desensamblado. IDA puede tener problemas en localizar el final de una función si el compilador ha partido la función en distintos rangos de direcciones o cuando, en el proceso de optimización de código el compilador combina el final común de dos o más funciones para ahorrar espacio.

6.3.3.1.—Crear funciones nuevas

Bajo ciertas circunstancias, pueden crearse funciones nuevas donde no existen. Las nuevas funciones pueden crearse con instrucciones existentes que no formen parte ya de una función o también pueden crearse con los bytes de datos que no han sido definidos por IDA de ninguna forma, por ejemplo **double words** o **strings**. Para crear una función deberemos colocar el cursor en el primer byte o instrucción que será incluida en la nueva función y realizar la acción **Edit** > **Functions** > **Create Function**. Con esto IDA intentará convertir los datos, si es necesario, a código. Entonces va examinando la función avanzando en ella buscando una declaración de retorno. Si puede localizar un fin adecuado para la función, generará un nombre nuevo de función, analizará el **stack frame** y reestructurará el código en la forma de una función. Si no pudiera localizar el fin de la función o encontrara cualquier instrucción no permitida, entonces la operación fallaría.

6.3.3.2.—Borrar funciones

Podemos borrar funciones existentes realizando la siguiente acción **Edit** > **Functions** > **Delete Function**. El fin de borrar una función, por ejemplo es que creas que IDA ha realizado mal su autoanálisis.

6.3.3.3.—Funciones troceadas

Las funciones troceadas se encuentran normalmente en el código generado por el compilador **Microsoft Visual C++**. Dichos trozos son el resultado de mover bloques de código poco utilizados, para hacer espacio en las páginas de memoria para los bloques de código más utilizado, por el compilador.

Cuando una función es cortada de alguna manera, IDA intentará localizar todos los trozos asociados a ella colocando saltos hacia cada trozo. En la mayoría de los casos IDA podrá localizar todos los trozos y listar cada trozo con el nombre de la función, como podemos ver en el listado siguiente.

```
.text:004037AE ChunkedFunc     proc near
.text:004037AE
.text:004037AE var_420         = dword ptr -420h
.text:004037AE var_41C         = dword ptr -41Ch
.text:004037AE var_4           = dword ptr -4
.text:004037AE hinstDLL        = dword ptr  8
.text:004037AE fdwReason       = dword ptr  0Ch
.text:004037AE lpReserved      = dword ptr  10h
.text:004037AE
.text:004037AE ; FUNCTION CHUNK AT .text:004040D7 SIZE 00000011 BYTES
.text:004037AE ; FUNCTION CHUNK AT .text:004129ED SIZE 0000000A BYTES
.text:004037AE ; FUNCTION CHUNK AT .text:00413DBC SIZE 00000019 BYTES
.text:004037AE
.text:004037AE                 push    ebp
.text:004037AF                 mov     ebp, esp
```

Se pueden llegar a los trozos de función fácilmente haciendo doble click en la dirección asociada al trozo. En el listado de desensamblado, la función troceada se denota con comentarios al inicio y final de la función, como se muestra

```
.text:004040D7 ; START OF FUNCTION CHUNK FOR ChunkedFunc
.text:004040D7
.text:004040D7 loc_0040C0D7:                    ; CODE XREF: ChunkedFunc+72↑j
.text:004040D7                 dec     eax
.text:004040D8                 jnz     loc_403836
.text:004040DE                 call    sub_4040ED
.text:004040E3                 jmp     loc_403836
.text:004040E3 ; END OF FUNCTION CHUNK FOR ChunkedFunc
```

En algunas ocasiones IDA puede fallar al localizar los trozos asociados con una función o funciones pudiéndose identificar como partes de otras funciones. En estos casos necesitarás crear los trozos de función faltantes o borrar los existentes.

Si quieres crear una nueva función con trozos de función, tendrás que seleccionar el rango de direcciones que abarquen los trozos, los cuales no deben ser parte de ninguna función existente y realizar la acción **Edit > Functions > Append Function Tail**. En este punto se te preguntará para que selecciones la función padre de una lista de todas las funciones definidas.

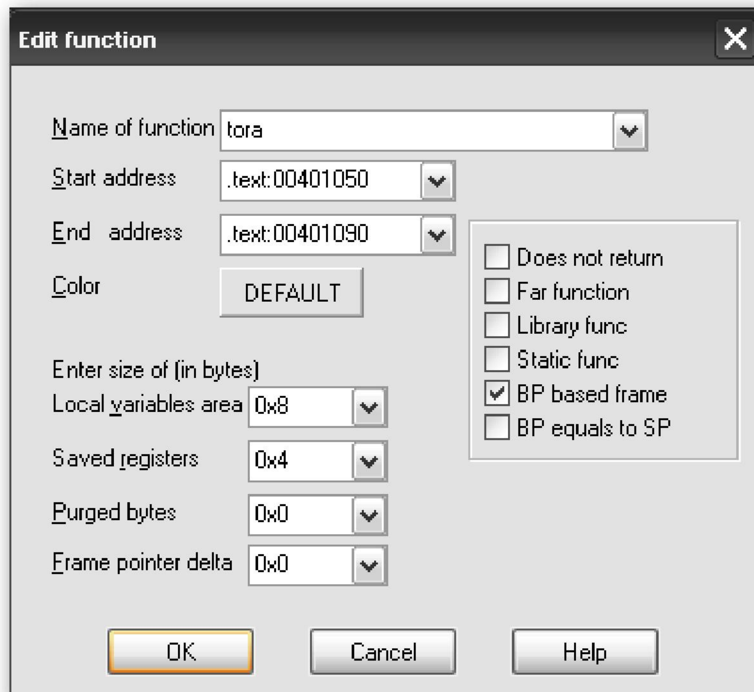
Observación: En el listado de desensamblado, las funciones troceadas son referidas como: function chunks. En el menú system de IDA, dichas funciones son también llamadas como function tails.

Puedes eliminar una función troceada posicionando el cursor en cualquier línea de trozo para ser eliminado y realizar la acción **Edit > Functions > Remove Function Tail**. Cuando lo hayas hecho se te repreguntará para confirmar la acción, antes de eliminar el trozo seleccionado.

Si se da el caso de que los trozos de la función están fuera del programa desensamblado será un contratiempo, lo podemos evitar si deseleccionamos la opción **Create function tails loader** en el primer cargado del archivo en IDA. Esta opción es una de las opciones de cargado a las cuales se pueden acceder con la **Kernel Options** (escrito 3.1.2) del diálogo inicial para cargar un archivo. Se deshabilitamos **function tails**, la diferencia principal que se advierte es que no se analizan las funciones con saltos a partes fuera de los límites de la función. IDA resaltará cada salto utilizando una línea roja flechada en la parte izquierda de la ventana de desensamblado. En la vista gráfica, en la función donde ocurra este caso, los objetivos de los saltos no serán mostrados.

6.3.3.4.—Atributos de función

IDA tiene asociados distintos atributos a cada función que haya reconocido. El diálogo de propiedades de función, figura abajo, puede utilizarse para editar cualquiera de esos atributos. Cada atributo puede ser modificado como se relaciona a continuación.



Name of function
Una forma alternativa para cambiar el nombre de la función.

Start Address
Se muestra la dirección de la primera línea de la función. IDA la puede determinar automáticamente, durante el análisis o con la dirección utilizada durante la operación para crear una función.

End address
La dirección siguiente a la última instrucción en la función. Normalmente es la instrucción que sigue a la instrucción de retorno de la función. En la mayoría de los casos esta dirección se determina automáticamente durante la fase de análisis o en la creación de una función. En los casos donde IDA tiene problemas para determinar el final verdadero de una función, tendremos que proporcionar este dato manualmente. Pero recuerda esta dirección en realidad no forma parte de la función, es la que sigue a la última de la función.

Local variables area
Aquí se muestran el número de bytes de pila dedicados para las variables locales de una función. Recordemos, figura abajo,

	Variables	Offset	
esp →	z	[ebp – 76]	→ variables loc.
	y	[ebp – 72]	
	buffer	[ebp – 68]	
	x	[ebp – 4]	
ebp→	Ebp guardado	[ebp]	registro(s) guardado
	Eip guardado	[ebp + 4]	→ parámetros
	a	[ebp + 8]	
	b	[ebp + 12]	
	c	[ebp + 16]	

Saved registers
Se muestra el número de bytes utilizados para guardar los registros, figura arriba, pedidos por el llamador. IDA considera al área de registros salvados a la parte superior de la dirección de retorno guardada y por debajo de todas las variables locales asociadas con la función. Algunos compiladores optan por guardar los registros por encima de las variables locales de una función. IDA considerará dicho espacio como área de variables locales antes que área de registros guardados.

Purged bytes
Los bytes purgados nos muestra el número de bytes de los parámetros que una función quita de la pila cuando esta retorna a su llamador. Para las funciones **cdecl** este valor siempre será cero. Para las funciones **stdcall,** este valor representará la cantidad de espacio consumido por todos los parámetros pasados a la pila, ver figura arriba. En los programas x86, IDA puede determinar automáticamente este valor cuando determina que se utiliza la variante **RET N** como instrucción de retorno.

Frame pointer delta
En algunos casos, los compiladores pueden ajustar el frame pointer de una función, para que apunte al punto medio del área de variables locales antes que al frame pointer guardado en la parte superior del área de variables locales. La distancia desde el frame pointer ajustado hasta el frame pointer salvado toma el nombre de **frame pointer delta**. En la mayoría de los casos ningún frame pointer delta se calculará automáticamente cuando la función sea analizada. Los compiladores utilizan un **stack frame delta** para optimizar su rapidez. El propósito del delta es mantener tantas variables stack frame de un byte dentro de un rango Offset de (-128.+127) desde el frame pointer.

Los atributos restantes se utilizan para caracterizar a la función. Al igual que otras opciones del diálogo, estos reflejarán resultados en el análisis automático de IDA. Los siguientes atributos pueden habilitarse o deshabilitarse.

Does no return
Habilitándolo la función no retorna al llamador. De esta forma cuando una función es llamada, IDA no asume que la ejecución seguirá continuando en la instrucción **call** asociada.

Far function
Utilizado para marcar una función como función lejana en arquitecturas segmentadas. Ambos llamadores de la función necesitarán especificar un valor de segmento y Offset cuando llamen a la función. La necesidad de utilizar llamadas lejanas es normalmente útil para modelos de memoria utilizados por programas en arquitecturas que soporten segmentación, por ejemplo, el uso de memoria **large**, opuesta a **flat**, en un x86.

Library func
Habilitar una función como una librería de código. El código de dicha función podría incluir rutinas de soporte para un compilador o funciones que formen parte de una DLL. Señalar una función como función librería produce que dicha función se muestre con los colores asignados para una función de dicho tipo resaltándola de las demás.

Static func
No hace nada más que mostrar el modificador estático en la lista de atributos de una función.

BP based frame
Indica que la función utiliza un **frame pointer**. En la mayoría de los casos se determina automáticamente al analizar el prólogo de la función. Si el análisis no puede reconocer su frame pointer se utilizará el proporcionado manualmente. Si quieres escogerlo manualmente, asegúrate de ajustar el tamaño del registro guardado, normalmente aumentado por el tamaño del frame pointer guardado, y el tamaño de la variable local, normalmente disminuido por el tamaño del frame pointer guardado. Para un **frame pointer – based frames**, la memoria proporciona las referencias, utilizadas por el frame pointer formateadas para utilizar los nombres simbólicos de las variables del stack, antes que Offset numéricos. Si no está habilitado este atributo, se asume que las referencias del stack frame son relativas al registro de puntero de pila.

BP equals to SP
Algunas funciones configuran el frame pointer para apuntar a la parte superior del stack frame, a través del puntero de pila, para entrar a una función. Este atributo habrá que habilitarlo en dichos casos. Esto es esencial, de la misma forma que tener el frame pointer delta con el mismo tamaño que el área de variables locales.

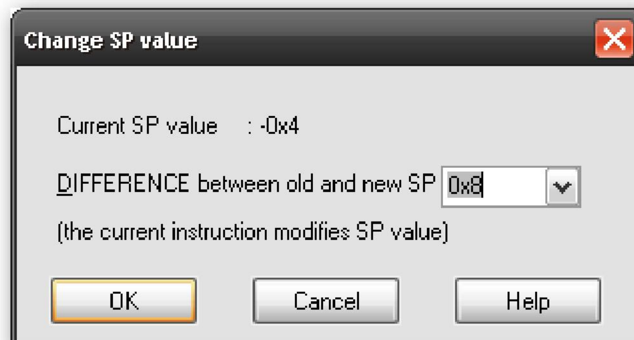
6.3.3.5.—Ajustes del puntero de pila (Stack Pointer)
Como hemos mencionado anteriormente, IDA realiza verdaderos esfuerzos para rastrear todos los cambios del puntero de pila en cada instrucción de una función. La exactitud con que lo realice, tendrá un gran impacto en la vista del esquema del **stack frame**. Cuando IDA no sea capaz de determinar alguna instrucción que altera el **stack pointer**, tendrás que especificar manualmente dicho ajuste.

El ejemplo más claro de este caso, ocurre cuando una función llama a otra función utilizando el acuerdo de llamada **stdcall**. Si la función que será llamada reside en una librería compartida, IDA no tiene conocimiento de ella, con lo cual no sabrá que dicha función utiliza **stdcall** y no podrá explicar porque el puntero de pila ha sido modificado por la función llamada antes de retornar. Por lo tanto reflejará un valor del puntero de pila inexacto para el resto de la función. La siguiente secuencia de llamada, en la que **alguna_funcion_importada** reside en una librería compartida, nos mostrará el problema, observa que la opción de línea **stack pointer** está habilitada.

```
.text: 004010EB    01C   push   eax
.text: 004010F3    020   push   2
.text: 004010FB    024   push   1
.text: 00401102    028   call   alguna_funcion_importada
.text: 00401107    028   mov    ebx, eax
```

En el momento que **alguna_funcion_importada** utilice **stdcall**, limpiará los tres parámetros de la pila cuando retorne, y el valor correcto del puntero de pila debería tener el valor de **01C**. Una forma de arreglar este problema es con un ajuste manual en la instrucción **00401107**. Los ajustes de pila se pueden realizar resaltando la dirección

donde hay que aplicar el ajuste y realizaremos la acción **Edit** > **Functions** > **Change Stack Pointer**, o el atajo **ALT-K**, especificando el número de bytes con el que cambiaremos el puntero de pila, en este caso 0xC (0x28-0xC = 0x1C).



Mientras que el ejemplo anterior ha servido para ilustrar una manera de realizarlo, existe una solución mejor a este problema en particular. Consideremos el caso en que **alguna_funcion_importada** es llamada muchas veces. En este caso necesitaríamos realizar dicho ajuste de pila en cada ubicación en donde es llamada la función. Realmente esto es mucho trabajo y podríamos equivocarnos alguna vez. La mejor forma, sería enseñar a IDA un comportamiento especial con respecto a la función **alguna_funcion_importada**. Como estamos tratando con una función importada, cuando nos desplazamos por ella, finalizamos en la entrada de la tabla de importación de ella, la cual sería algo parecido a esto:

```
.idata: 00418078;Segment type: Externs
.idata: 00418078;_idata
.idata: 00418078          extrn alguna_funcion_importada:dword ; DATA XREF: sub_401034  r
```

Aunque esta es una función importada, IDA te permite editar una parte de información concerniente a su comportamiento: El número de bytes purgados asociados con la función. Al editar dicha función se puede especificar el número de bytes que se limpian de la pila cuando retorne, con lo cual propaga la información supliéndola en cada ubicación en donde se llama a la función, instantáneamente corrige el cálculo del puntero de pila en cada una de estas ubicaciones.

6.3.4.—Convertir datos a código y viceversa

Durante la fase de análisis automático, ocasionalmente se pueden categorizar bytes incorrectamente. Bytes de datos se pueden clasificar incorrectamente como bytes de código desensamblados en instrucciones, o bytes de código clasificados incorrectamente como bytes de datos y formateados como valores de datos. Esto sucede por muchas razones, por ejemplo ciertos compiladores adjuntan datos en la sección de código de los programas o por el hecho de que ciertos bytes de código nunca son referenciados directamente como código, con lo cual IDA opta por no desensamblarlos. En particular los programas ofuscados tienden a entorpecer la distinción entre secciones de datos y secciones de código.

Si por estas razones quieres reformatear tu desensamblado, hacerlo es muy fácil. La primera opción para reformatearlo de nuevo es quitar todo el formateado actual, código y datos. Es posible interpretar funciones, código y datos haciendo doble click en el elemento el cual quieres interpretar y elegir **Undefine**, también puedes hacer la acción **Edit** > **Undefine** o utilizar el atajo **U**, todo esto nos dará como resultado un menú de

contexto. Interpretar un elemento provoca el subrayado de los bytes que serán reformados como una lista de valores bytes crudos. Podemos interpretar una gran parte de código haciendo click + arrastre para seleccionar el rango de direcciones antes de ejecutar la operación de interpretación. Como ejemplo consideremos el siguiente listado de una función:

```
.text:004013E0 ; =============== S U B R O U T I N E =======================================
.text:004013E0
.text:004013E0 ; Attributes: bp-based frame
.text:004013E0
.text:004013E0 sub_4013E0      proc near               ; DATA XREF: sub_4011C0+6F↑o
.text:004013E0                 push    ebp
.text:004013E1                 mov     ebp, esp
.text:004013E3                 pop     ebp
.text:004013E4                 retn
.text:004013E4 sub_4013E0      endp
```

Interpretar esta función nos produce una serie de bytes sin catalogar tal como se muestra abajo, los cuales podemos interpretar virtualmente de cualquier manera.

```
.text:004013D5                 align 10h
.text:004013E0 unk_4013E0      db  55h ; U             ; DATA XREF: sub_4011C0+6F↑o
.text:004013E1                 db  89h ; ë
.text:004013E2                 db 0E5h ; Õ
.text:004013E3                 db  5Dh ; ]
.text:004013E4                 db 0C3h ; +
.text:004013E5                 align 10h
```

Para desensamblar una secuencia de bytes interpretados, hacemos click derecho en el primer byte a desensamblar y seleccionamos **Code**, también realizando la acción **Edit > Code** o con el atajo **C**. Esto causa el desensamblado de todos los bytes mientras no se encuentre un elemento ya definido o una instrucción no válida. Las partes grandes de código se pueden realizar haciendo click + arrastre seleccionando el rango de direcciones antes de realizar la operación de la conversión del código.

La operación complementaria para convertir código a datos es un poco más compleja. Primero, no es posible convertir código a datos utilizando un menú contextual. Las alternativas que tenemos son realizando la acción **Edit > Data** o el atajo **D**. La mayor parte de conversiones de instrucciones a datos son fáciles de realizar, primero interpretamos todas las instrucciones lo cual lo convierte en datos y entonces formateamos apropiadamente los datos. El formateado básico de datos lo estudiaremos en el próximo escrito.

Performance Bigundill@