

## **C Y REVERSING (parte 1)**

Realmente estoy empezando a escribir un tute al que pomposamente llame C Y REVERSING parte 1 y para los que lo están leyendo, y tratando de ver de que va, les aclaro que la verdad no tengo la menor idea de si esto va a ser un tute solo, o varios, y una muy vaga idea de lo que voy a hacer. Quizas sea que como estoy de vacaciones en el fondo extraño un poco la acción, por eso hay que ver si cuando vuelva al duro trajín diario, tendré aun fuerzas para seguir con esta serie, por eso, preferí escribir esta parte sin obligarme a nada, si sale bien, bien, si gusta mejor, si tengo mas ganas agregare mas, sino la dejare como parte 1 única, como algo básico para profundizar algún día.

La idea es Reversear desde cero, primero código fuente en C, lo mas sencillo posible que nosotros mismos haremos o sacaremos de Internet, y lo trataremos de Reversear en IDA, para encontrar en el código, nuestras variables, estructuras, funciones, etc, e ir complicando, para hacernos fuertes en reversing, así cuando tengamos programas que Reversear sin el código fuente como ocurre en la realidad cotidiana, tendremos una guía de como hacerlo.

Como siempre en todos mis tutes les pido disculpas por mi total mal uso de vocabulario técnico, de cualquier manera trato de explicar en forma sencilla, y a pesar de algún error en el vocabulario técnico, lo mostrado es tan simple que igual se entiende, para el que sabe y le encante pavonearse con palabras técnicas difíciles, discúlpeme, yo se hacer las cosas que explico, las hago diariamente, y para ello no necesito saber todos los nombres técnicos, las hago y listo, aunque para explicar en un tute a veces eso trae alguna imprecisiones y posiblemente algún purista iluminado se podrá reír de mi uso de las mismas.(como siempre me importa un pito jeje, mientras los que se inician me entiendan esta todo bien)

Las herramientas que utilizaremos serán el IDA 5.5 mas HexRays que esta en mi web:

<http://ricardonarvaja.net/WEB/OTROS/HERRAMIENTAS%20PRIVADAS/F-G-H-I-J-K/IDA.Pro.Advanced.v5.5.incl.Hex.Rays.Decompiler.v1.1-iND.7z>

Por supuesto el plugin HexRays verifica la licencia conectándose a Internet, así que ademas de quitar la tilde CHECK FOR UPDATES que aparece en el splash screen del plugin cuando abrimos un ejecutable en el IDA, conviene trabajar en una maquina virtual, de modo de hacer un snapshot, para que si algún chequeo hace que se venza, podamos volver al snapshot anterior y vuelva a funcionar.

Utilizaremos también el compilador de C y C++ llamado DEVCC++ que aunque es un poquito antiguo, es fácil de usar, el que quiere podrá reemplazarlo por otro, no hay problema, si quieren bajarse el DEVCC++ en mi web esta aquí:

[http://ricardonarvaja.net/WEB/OTROS/HERRAMIENTAS/A-B-C-D-E/devcpp-4.9.9.2\\_setup.exe](http://ricardonarvaja.net/WEB/OTROS/HERRAMIENTAS/A-B-C-D-E/devcpp-4.9.9.2_setup.exe)

Bueno fuera de esto usaremos algunas pocas veces Ollydbg 1.10, algún editor de texto como el Notepad ++ y no mucho mas según mis cálculos.

Para que no digan que robamos, lo cual es cierto jeje, la mayor parte de los códigos fuente los sacaremos del curso de C que se encuentra aquí:

<http://www.nachocabanes.com/c/curso/>

si alguien tiene ganas de complementar leyéndolo a la misma vez que vamos aquí reverseando los ejemplos pues adelante, es una buena lectura complementaria ya que aquí no haremos un curso de C sino que usaremos los ejemplos para reversear.

Como siempre comenzaremos con el ejemplo mas sencillo del universo.

```
#include <stdio.h>
```

```
main()
{
    printf("Hola");
}
```

Para los que no conocen C estas tres lineas que quizás sean el mínimo programa, se explican sencillamente de la siguiente manera.

```
#include <stdio.h>
```

Esta linea se agrega para ampliar el lenguaje básico de C, que no trae incluida en el mismo, la función que permite imprimir en pantalla (**printf**) es como si fuera un import en python para los que conocen el mismo, sin este **include** no podríamos imprimir la salida en este caso el texto “**Hola**”

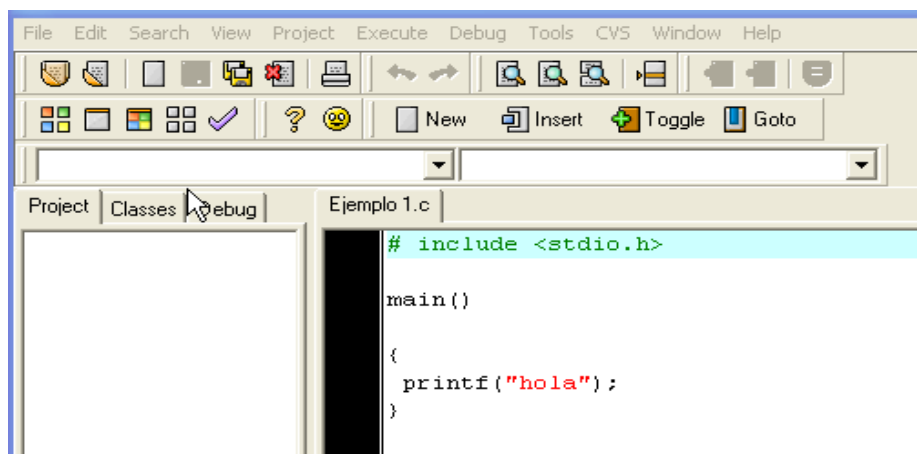
```
main()
{
}
}
```

La función **main** es la función principal del programa y dentro de las llaves que están a continuación escribiremos el código que queremos ejecutar, en este caso el **printf** para sacar por pantalla el texto “Hola Mundo”

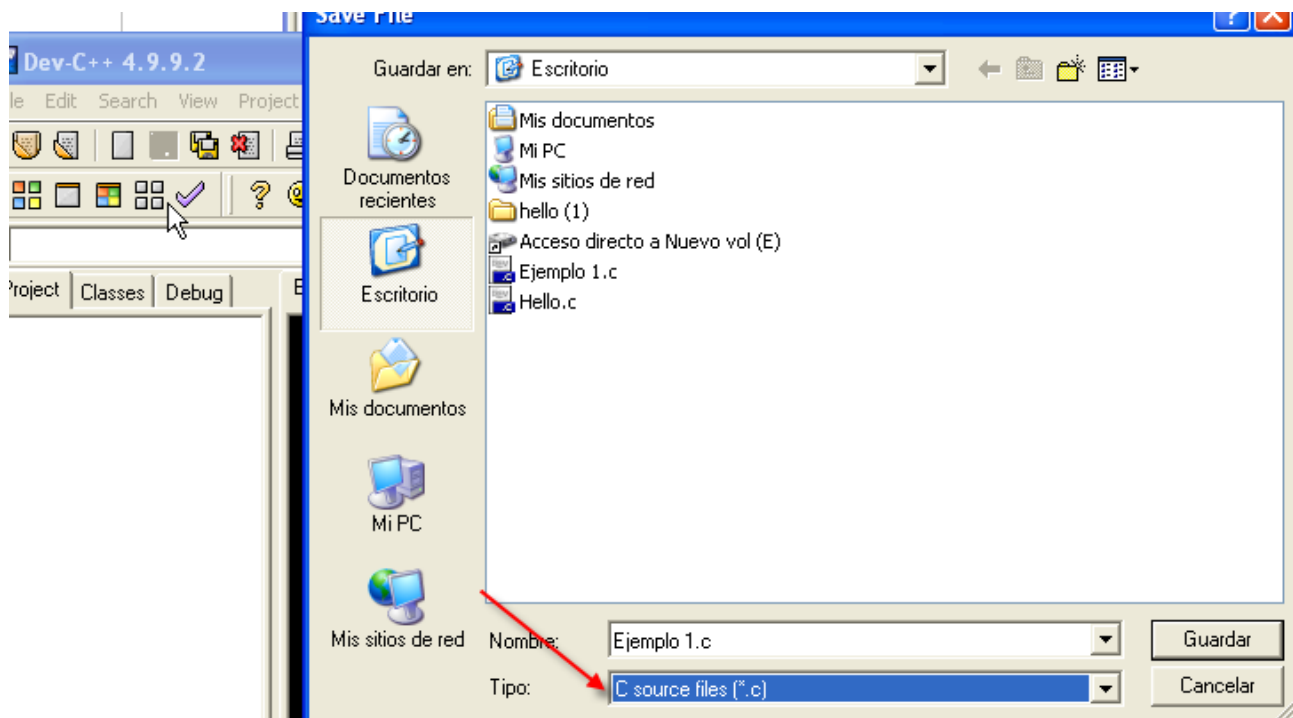
Esta es una explicación muy básica, como siempre el que quiera la explicación completa, puede leer la del curso de Cabanes en esta pagina.

<http://www.nachocabanes.com/c/curso/cc01.php>

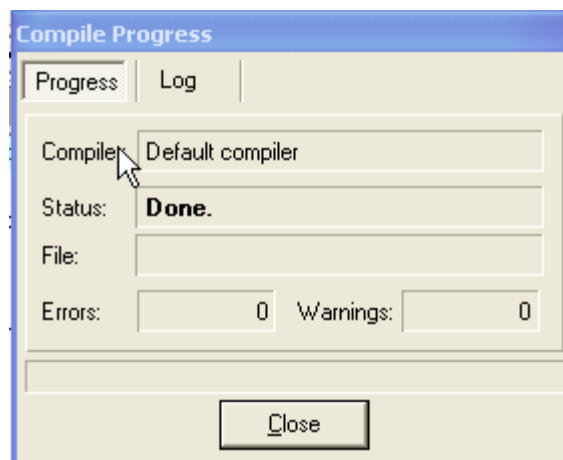
Bueno abrimos el DEV++ vamos a FILE-NEW-SOURCE FILE y ahí nos abrirá un espacio en blanco para tipear o copiar y pegar nuestro código fuente.



Luego yendo a FILE-SAVE AS lo guardamos fijándonos en cambiar para que lo guarde como código fuente en C.



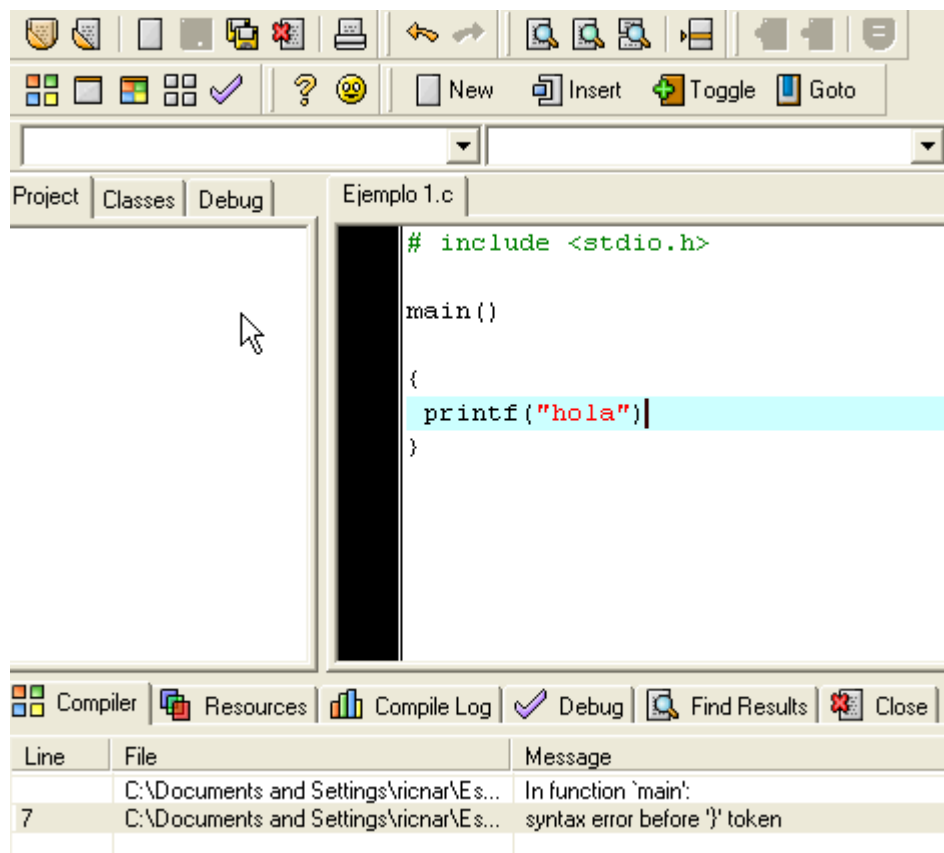
Ahora lo compilamos yendo a EXECUTE-COMPILE



Si les aparece algún error, recuerden que en C cada orden debe terminar con punto y coma, en este caso la orden que le damos para que imprima en pantalla a través de la función **printf** debe tener un punto y coma al final, si lo quitamos nos dará un error medio critico como vemos en la imagen de abajo, así que cuidado con ello.

También el DEVCC++ da error si tienen instalado en la misma maquina el MINGW, así que desinstalen temporalmente el MINGW y todo funcionara normalmente.

Error por falta del punto y coma



Así que ya tenemos nuestro ejecutable, será creado en la misma carpeta que tenemos el archivo fuente .c, con el mismo nombre pero con extensión .exe.



Corramos el ejecutable dentro de una consola a ver si va bien.

```
C:\ Simbolo del sistema
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\ricnar>cd Escritorio

C:\Documents and Settings\ricnar\Escritorio>Ejemplo 1.exe
"Ejemplo" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Documents and Settings\ricnar\Escritorio>"Ejemplo 1.exe"
hola
C:\Documents and Settings\ricnar\Escritorio>
```

En mi caso como puse un espacio en el nombre tuve que ejecutarlo al mismo entre comillas sino no lo tomara, vemos el hola impreso en la consola así que funciona, ahora abramos este ejecutable en IDA.

En este caso como nuestro main es una función sin argumentos y sin variables, al menos nosotros no creamos ninguna, debemos separar las cosas que le agrego el compilador, para entender el código original, en este caso al verlo en IDA, que se abre directamente en el main y evita mostrar el código desde el entry point, que por supuesto no es creado por nosotros y es puramente creado por el compilador para hacer los includes necesarios y poder crear un contexto adecuado para que nuestro código corra bien.

```
00401290
00401290 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401290 _main proc near
00401290
00401290 var_4= dword ptr -4
00401290 argc= dword ptr 8
00401290 argv= dword ptr 0Ch
00401290 envp= dword ptr 10h
00401290
00401290 push    ebp
00401291 mov     ebp, esp
00401293 sub     esp, 8
00401296 and     esp, 0FFFFFF0h
00401299 mov     eax, 0
0040129E add     eax, 0Fh
004012A1 add     eax, 0Fh
004012A4 shr     eax, 4
004012A7 shl     eax, 4
004012AA mov     [ebp+var_4], eax
004012AD mov     eax, [ebp+var_4]
004012B0 call    __chkstk
004012B5 call    __main
004012BA mov     dword ptr [esp], offset aHola ; "hola"
004012C1 call    printf
004012C6 leave
004012C7 retn
```

Aquí vemos que el compilador modifico nuestro main haciéndolo una función de tres argumentos que no son nuestros, sino del compilador, y tiene una variable. var\_4 que creo el compilador la cual si hacemos las cuentas devolverá siempre 10 y la utilizara el compilador al cerrar el programa, por supuesto nada de esto es código propio, vayamos a nuestro código, propiamente dicho.

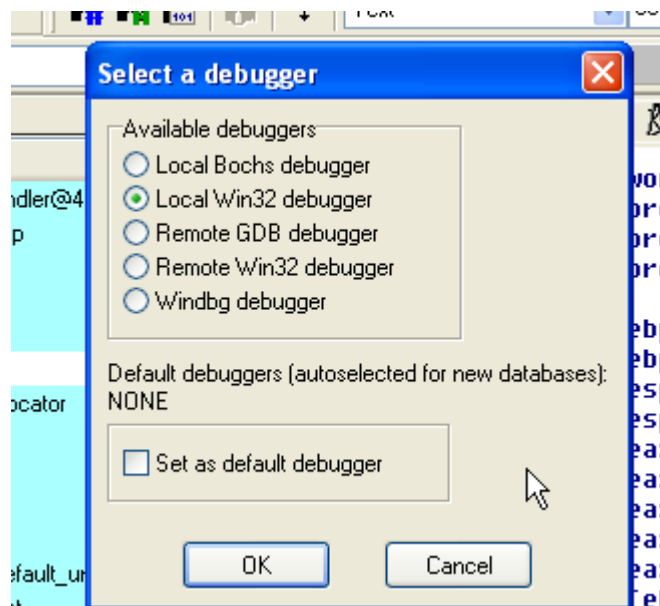
```
.text:004012BA mov     dword ptr [esp], offset aHola ; "hola"
.text:004012C1 call    printf
```

Allí vemos que pone en el stack un puntero a la string hola que esta en la sección rdata, si hacemos click en aHola, nos muestra que esta exactamente en 403000 y que es el argumento de la función

printf.

```
rdata:00403000
rdata:00403000 ; Segment type: Pure data
rdata:00403000 ; Segment permissions: Read
rdata:00403000 _rdata          segment para public 'DATA' use32
rdata:00403000          assume cs:_rdata
rdata:00403000          ;org 403000h
rdata:00403000 ; char aHola[]
rdata:00403000 aHola          db 'hola',0          ; DATA XREF: _main+2Afo
rdata:00403005          align 10h
rdata:00403010 dword_403010    dd 42494C2Dh        ; DATA XREF: __w32_sharedptr_
rdata:00403010          ; __w32_sharedptr_initialize
rdata:00403010          ; DATA XREF: __w32_sharedptr_initialize
```

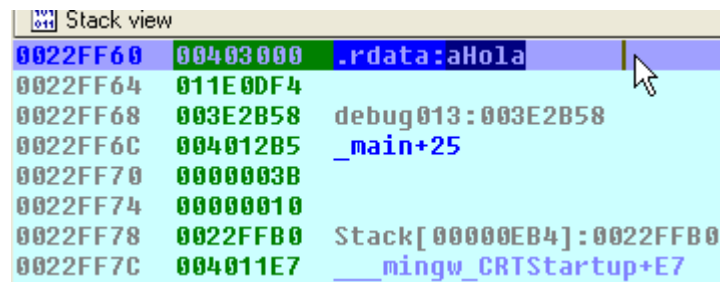
Si vamos a DEBUGGER y elegimos por ejemplo el debugger local de win32



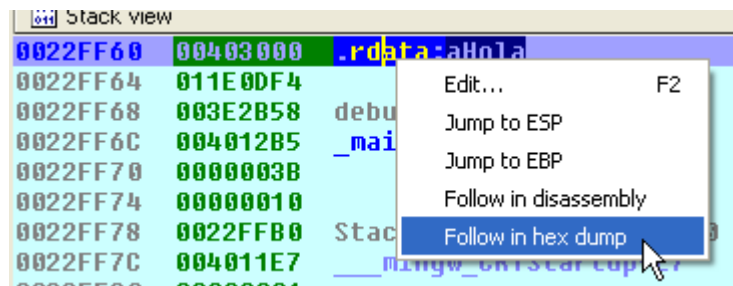
y ponemos breakpoint alli.

```
004012A4 shr     eax, 4
004012A7 shl     eax, 4
004012AA mov     [ebp+var_4], eax
004012AD mov     eax, [ebp+var_4]
004012B0 call    __chkstk
004012B5 call    __main
004012BA mov     dword ptr [esp], offset aHola ; "hola"
004012C1 call    printf
004012C6 leave
004012C7 retn
004012C7 _main endp
004012C7
```

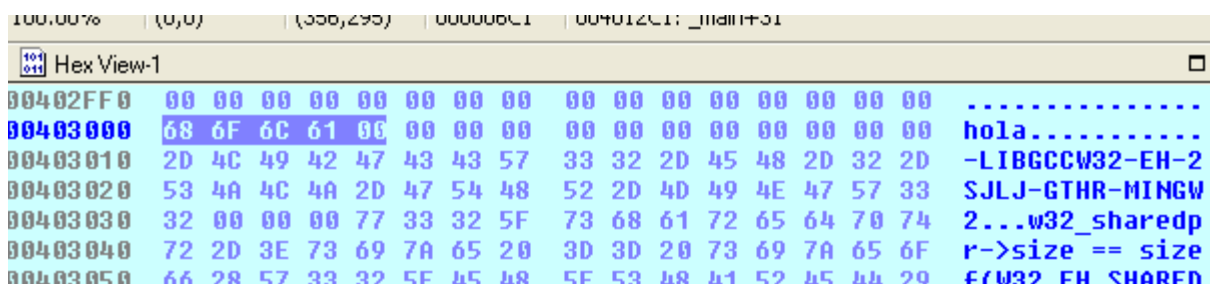
Y damos START PROCESS, en la ventana del stack cuando se detiene cambiamos a JMP TO ESP para que muestre el código que apunta ESP, o sea el stack y no cualquier cosa como normalmente hace , jeje.



Vemos el puntero a la string hola, que podemos verla en el hex dump, haciendo click derecho – FOLLOW IN HEX DUMP.



Y vemos que allí esta.



Es importante que cuando reverseamos separemos la paja del trigo y no nos metamos con el código que agrega el compilador, ya vimos que lo anterior al main es creado por el mismo y dentro del mismo main solo dos lineas pertenecen a nuestro código.

Una forma de separar y comprender que agrega de mas el compilador, es variar un poco el código fuente a este:

```
#include <stdio.h>
```

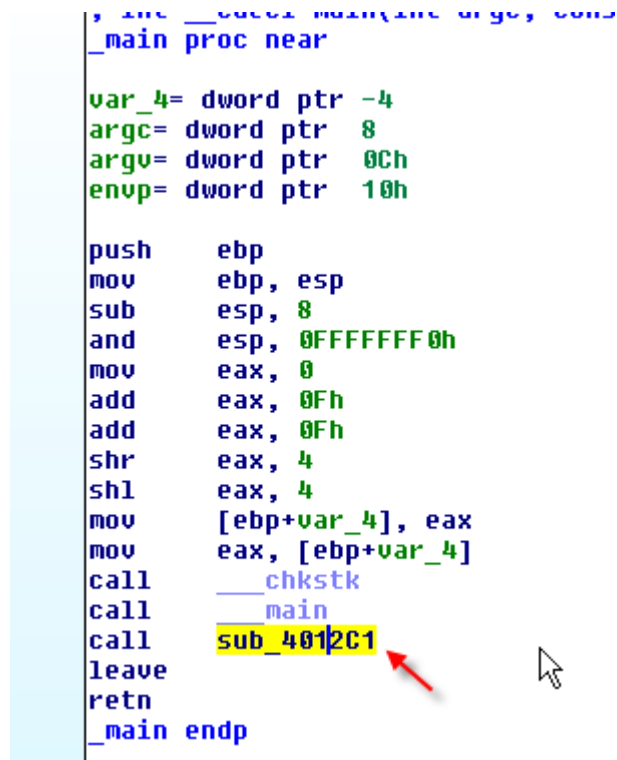
```
main(){
funcion1();
```

```
}
```

```
funcion1(){  
    printf("hola");  
}
```

Como vimos que al main, el compilador le agrega argumentos y variables que usa el mismo, dentro de main creamos una funcion nuestra llamada funcion1, y dentro de ella si ponemos el printf, si la compilamos y corremos vemos que funciona igual que la anterior pero al verla en el IDA.

```
, int __cdecl main(int argc, const  
_main proc near  
  
var_4= dword ptr -4  
argc= dword ptr  8  
argv= dword ptr  0Ch  
envp= dword ptr  10h  
  
push    ebp  
mov     ebp, esp  
sub     esp, 8  
and     esp, 0FFFFFF0h  
mov     eax, 0  
add     eax, 0Fh  
add     eax, 0Fh  
shr     eax, 4  
shl     eax, 4  
mov     [ebp+var_4], eax  
mov     eax, [ebp+var_4]  
call    ___chkstk  
call    main  
call    sub_4012C1  
leave  
retn  
_main endp
```



Aquí vemos el main que es similar al anterior y dentro el llamado a nuestra función, que como vemos en este caso no se le pasa argumento ninguno, renombremosla a funcion1 como en nuestro código, entrando a ella y click derecho en el nombre, rename o edit funcion y le cambiamos a funcion1.



```

; Attributes: bp-based frame

sub_4012C1 proc near
push    ebp
mov     ebp, esp
sub     esp, 8
mov     dword ptr [esp], offset aHola ; "hola"
call    printf
leave
retn
sub_4012C1 endp

```

Allí le cambiamos el nombre.

**Edit function**

Name of function:

Start address:

End address:

Color:

Enter size of (in bytes):

Local variables area:

Saved registers:

Purged bytes:

Frame pointer delta:

☐ Does not return  
☐ Far function  
☐ Library func  
☐ Static func  
☒ BP based frame  
☐ BP equals to SP

OK Cancel Help

Quedara así

```

; Attributes: bp-based frame

; int funcion1
funcion1 proc near
push    ebp
mov     ebp, esp
sub     esp, 8
mov     dword ptr [esp], offset aHola ; "hola"
call    printf
leave
retn
funcion1 endp

```

Vemos que el único cambio que hizo es ponerle el int delante, el resto vemos mucho mas limpio el código y lo que es estrictamente nuestro, no declaramos variables y aquí no hay, no hay argumentos pasados a la funcion tan cual nuestro código, vemos el puntero a hola, y el llamado a printf.

Si retrocedemos al main vemos que ahora nuestra función aquí también figura como funcion1.

```

; int _main
_main proc near
push    ebp
mov     ebp, esp
sub     esp, 8
and     esp, 0FFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    __chkstk
call    main
call    funcion1
leave
retn
_main endp

```

Este ejemplo no aporta mucho para el reversing pero es importante que uno con la experiencia sepa no darle importancia a lo que el compilador agrega al código fuente, de forma de poder interpretar solo las lineas del código original del autor y no fruta, ademas normalmente los programas contienen cientos de funciones dentro del main y es muy raro tener que reversear el main, normalmente estaremos reverseando dichas funciones, pero es bueno ver la diferencia y saber que en el main el compilador mete mucha mas fruta y ver cual es.

A partir de ahora todos los ejemplos que en el curso de C están escribiendo código en el main, nosotros los haremos en una función dentro del main para poder reversear mejor y no confundir con tanta fruta.

Para terminar el ejemplo 1 le agregamos la función getchar() que queda esperando hasta que se aprete ENTER en la consola, para que no se cierre la misma y poder arrancar el ejecutable dando

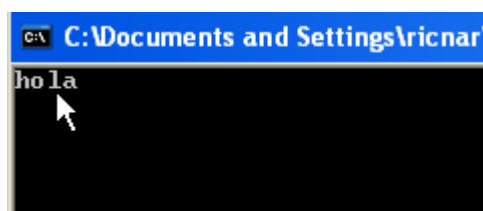
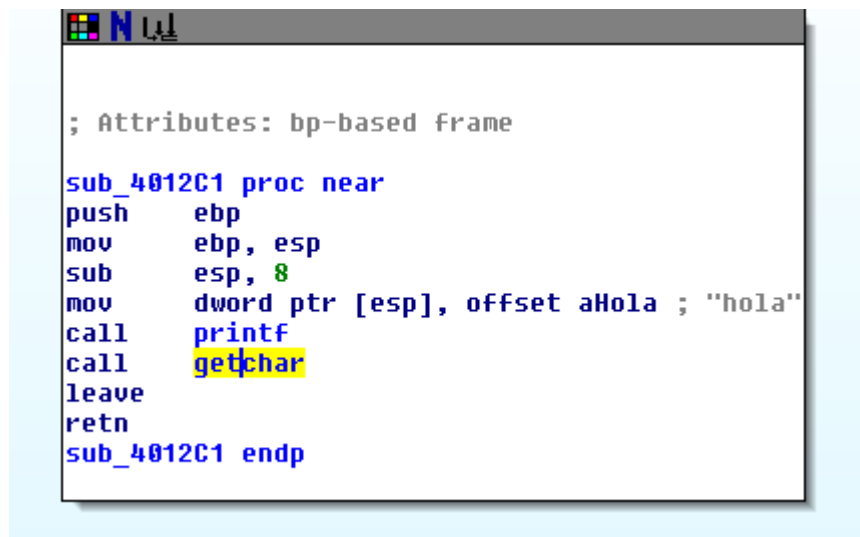
doble click.

```
#include <stdio.h>
```

```
main(){  
    funcion1();  
}
```

```
funcion1(){  
    printf("hola");  
    getchar();  
}
```

Vemos que al compilarlo nuevamente no hay ninguna diferencia, solo el llamado a getchar para que no se cierre al ejecutarlo fuera de la consola.



Si le damos ENTER se cerrara.

Veamos el siguiente ejemplo que sumara dos números y reverseemoslo.

```
#include <stdio.h>
```

```
main(){  
    funcion1();  
}
```

```
funcion1(){  
    int primerNumero;  
    int segundoNumero;  
    int suma;  
  
    primerNumero = 234;  
    segundoNumero = 567;  
    suma = primerNumero + segundoNumero;  
    printf("Su suma es %d", suma);  
  
    getchar();  
}
```

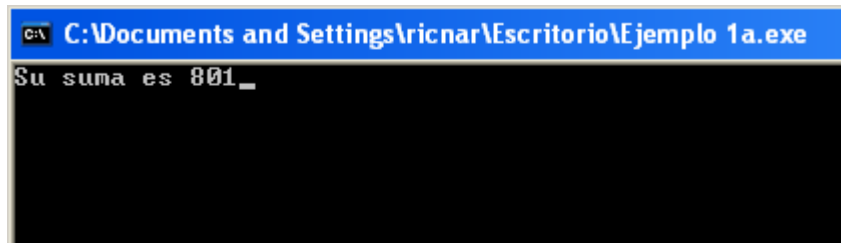
Vemos la declaración de tres variables enteras o int, una llamada **primerNumero**, otra **segundoNumero** y otra **suma**, las dos primeras las inicializo con los valores 234 y 567 y la variable **suma** quedara sin inicializar, y guardara el resultado de la misma.

```
printf("Su suma es %d", suma);
```

La instrucción **printf** solo imprime strings, pero acepta mediante format string reemplazar en este caso el %d por un la string del numero que se guardo en suma, de esta forma como en suma quedara el numero **801**, y mediante format string se convertirá en la string **"801"** y reemplazara al %d quedando el mensaje de salida

**Su suma es 801**

Veamos si funciona primero, compilemos y ejecutemoslo con doble click ya que pusimos el **getchar()** al final.



Bueno funciona veamoslo ahora en IDA.

```

enüp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 8
and     esp, 0FFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    ___chkstk
call    __main
call    sub_4012C1
leave
retn
_main endp

```

Ese es el main ahora entremos en nuestra función.

```

; Attributes: bp-based frame

sub_4012C1 proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+var_4], 0EAh
mov     [ebp+var_8], 237h
mov     eax, [ebp+var_8]
add     eax, [ebp+var_4]
mov     [ebp+var_C], eax
mov     eax, [ebp+var_C]
mov     [esp+4], eax
mov     dword ptr [esp], offset aSuSumaEsD ; "Su suma es %d"
call    printf
call    getchar
leave
retn
sub_4012C1 endp

```

Primero renombraremos la función tal como hicimos la vez anterior usando rename y ya queda con nuestro nombre, lo mismo haremos con las variables locales, en este caso nosotros le ponemos el mismo nombre que el código fuente, pues lo hicimos nosotros y los tenemos, pero si fuera una función de un programa que no tenemos el código, habrá que renombrar las variables con un

nombre aproximado a lo que interpretemos que hace cada una en el código.

Vemos que **var\_4** se la inicializa con el valor hexa EA que en decimal es 234 o sea que sera **primerNumero**, la renombramos, al igual que a **segundoNumero** y a **suma**.

```
; Attributes: bp-based frame

funcion1 proc near

suma= dword ptr -0Ch
segundoNumero= dword ptr -8
primerNumero= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+primerNumero], 0EAh
mov     [ebp+segundoNumero], 237h
mov     eax, [ebp+segundoNumero]
add     eax, [ebp+primerNumero]
mov     [ebp+suma], eax
mov     eax, [ebp+suma]
mov     [esp+4], eax
mov     dword ptr [esp], offset aSuSumaEsD ; "Su suma
call    printf
call    getchar
leave
retn
funcion1 endp
```

Allí vemos nuestras variables locales, como la funcion no tiene argumentos, las variables solo tienen sentido localmente así que no es necesario propagarlas como en el caso de renombrar argumentos que veremos mas adelante.

El código se interpreta fácilmente

```
mov     [ebp+primerNumero], 0EAh
mov     [ebp+segundoNumero], 237h
mov     eax, [ebp+segundoNumero]
add     eax, [ebp+primerNumero]
mov     [ebp+suma], eax
```

Luego de inicializar ambas variables se mueve a EAX el valor de **segundoNumero** y se le suma el valor de **primerNumero** quedando el resultado en EAX, el cual se guarda en la variable **suma**.

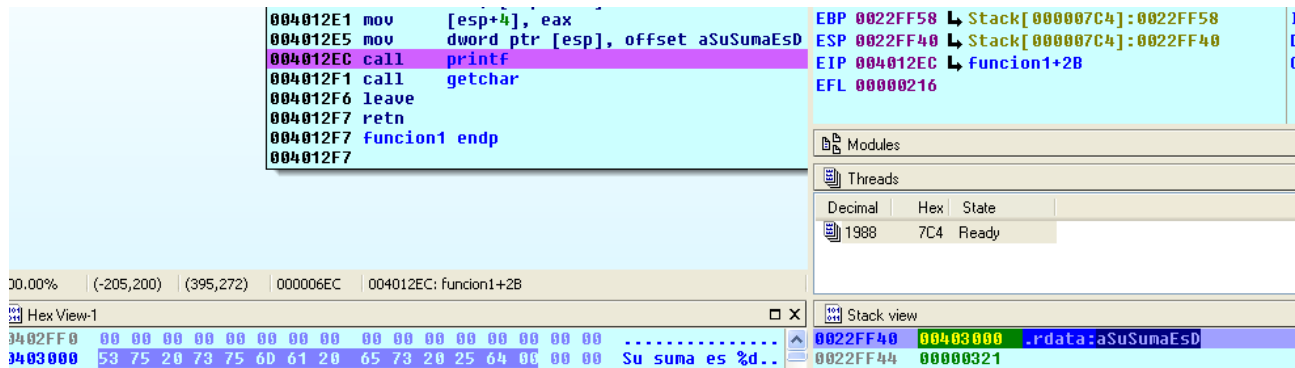
```
mov     eax, [ebp+suma]
mov     [esp+4], eax
mov     dword ptr [esp], offset aSuSumaEsD ; "Su suma es %d"
call    printf
```

Vemos que aun no teniendo el código fuente, viendo que las variables guardan miembros de una suma, y una tercera guarda el resultado, pues ya tenemos la posibilidad de ponerles nombres adecuados a la funcion de cada una.

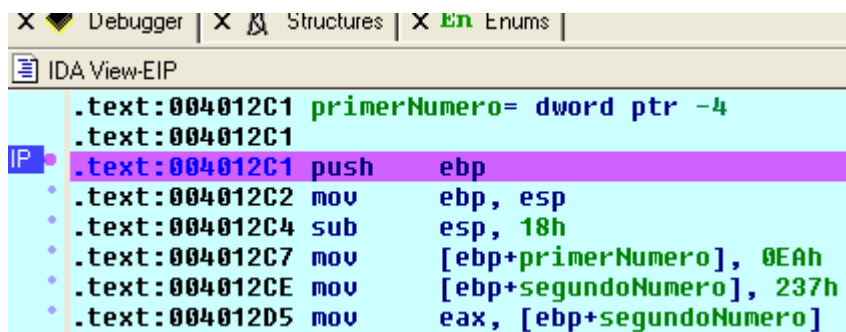
A continuación se mueve a EAX ese valor resultado de la suma, el cual se guarda en el stack para pasarlo como argumento a **printf**, el otro argumento es un puntero a la string **"Su suma es %d"** que

se envía también al stack ya que en este caso **printf** recibe dos argumentos.

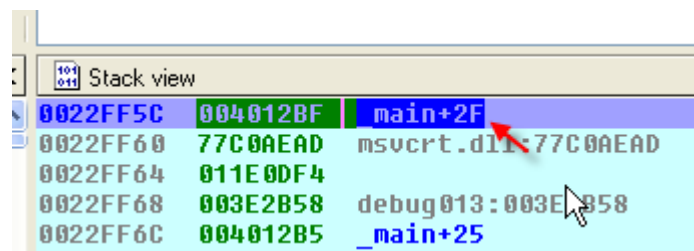
Si ponemos en el IDA un breakpoint en **printf** y acomodamos el stack con **JMP TO ESP** cuando se detiene vemos en el primer lugar el puntero a la string **Su suma es ...**, y en segundo lugar el segundo argumento **321** hexa que es el entero resultado de la suma, que la función **printf** convertirá a string y lo agregará el resto y mostrara en la consola como salida.



Si ponemos un BP apenas entrado en nuestra funcion.



Lógicamente como acabamos de detenernos al entrar en un call, lo primero que habrá en el stack, sera el return address donde volverá al terminar de ejecutarse mi funcion, así que si en el mismo hago **JMP TO ESP** y veo.



Si hago click derecho FOLLOW IN DISASSEMBLER veré donde volverá al retornar de mi funcion.

```

DA View-EIP
.text:00401290 push    ebp
.text:00401291 mov     ebp, esp
.text:00401293 sub     esp, 8
.text:00401296 and     esp, 0FFFFFF0h
.text:00401299 mov     eax, 0
.text:0040129E add     eax, 0Fh
.text:004012A1 add     eax, 0Fh
.text:004012A4 shr     eax, 4
.text:004012A7 shl     eax, 4
.text:004012AA mov     [ebp+var_4], eax
.text:004012AD mov     eax, [ebp+var_4]
.text:004012B0 call    __chkstk
.text:004012B5 call    __main
.text:004012BA call    funcion1
.text:004012BF leave   ←
.text:004012C0 retn
.text:004012C0 _main endp
.text:004012C0

```

Por supuesto volveré a la función main ya que mi función está dentro de ella, debajo del return address si mi función tuviera argumentos que se pasan con PUSH o se copian al stack antes de entrar a la misma, lógicamente estarían debajo del return address, pues se pushean antes de entrar, en este caso al no tener argumentos no veremos nada, aquí debajo marcamos la zona de argumentos que no está usada en este caso.

```

Stack view
0022FF5C 004012BF main+2F
0022FF60 77C0AEAD msvcrt.dll:77C0AEAD
0022FF64 011E0DF4
0022FF68 003E2B58 debug013:003E2B58
0022FF6C 004012B5 ← _main+25
0022FF70 0000003C
0022FF74 00000010
0022FF78 0022FFB0 Stack[000009B0]:0022FFB0
0022FF7C 004011E7 __mingw_CRTStartup+E7
0022FF80 00000001

```

Justo arriba del return address encontraremos dependiendo del tipo de función, si lo primero que hace al entrar en la misma es PUSH EBP, pues allí estará guardado el ebp del main o de la función que llamo a la nuestra, y al salir lo recuperará con POP EBP, o LEAVE.

Normalmente después del primer PUSH EBP, reservará lugar para las variables en el stack.

```

.text:004012C1 push    ebp
.text:004012C2 mov     ebp, esp
.text:004012C4 sub     esp, 18h

```

Iguala EBP con ESP y luego le resta a ESP un valor que depende de la cantidad de variables y lugar necesario que el compilador decida que necesita, en este caso le resta 24 decimal o sea que desde la dirección del stack donde pusheo el valor de EBP, hacia arriba quedará lugar para las 3 variables dword y un poco más de espacio.



0022FF40	77C04E42	msvcrt.dll:77C04E42
0022FF44	00401340	__do_global_dtors
0022FF48	0022FF58	Stack[000009B0]:0022FF58
0022FF4C	00401426	main:loc_401426
0022FF50	00401340	__do_global_dtors
0022FF54	00004000	
0022FF58	0022FF78	Stack[000009B0]:0022FF78
0022FF5C	004012BF	_main+2F

Ahora al hacer JMP to ESP ya que ESP cambio, queda a la vista el espacio en rosado reservado, y justo debajo en celeste el EBP guardado y el return address que habíamos visto.

En ese lugar en rosado, debe guardar los valores con los que inicializa las variables si vamos traceando al pasar por.

```
.text:004012C7 mov    [ebp+primerNumero], 0EAh
.text:004012CE mov    [ebp+segundoNumero], 237h
```

0022FF40	77C04E42	msvcrt.dll:77C04E42
0022FF44	00401340	__do_global_dtors
0022FF48	0022FF58	Stack[000009B0]:0022FF58
0022FF4C	00401426	main:loc_401426
0022FF50	00000237	
0022FF54	000000EA	
0022FF58	0022FF78	Stack[000009B0]:0022FF78
0022FF5C	004012BF	_main+2F
0022FF60	77C0AEAD	msvcrt.dll:77C0AEAD
0022FF64	011E0DF4	

Vemos que inicializo las dos variables y el resultado lo guardara en la variable suma que toma su valor cuando pasamos por aquí.

```
004012DB mov    [ebp+suma], eax
```

0022FF40	77C04E42	msvcrt.dll:77C04E42
0022FF44	00401340	__do_global_dtors
0022FF48	0022FF58	Stack[000009B0]:0022FF58
0022FF4C	00000321	
0022FF50	00000237	
0022FF54	000000EA	
0022FF58	0022FF78	Stack[000009B0]:0022FF78
0022FF5C	004012BF	_main+2F

Allí vemos como guarda el 321 hexa, resultado de la suma, justo arriba de las dos variables anteriores.

0022FF40	00403000	.rdata:aSuSumaEsD
0022FF44	00000321	
0022FF48	0022FF58	Stack[000009B0]:0022FF58
0022FF4C	00000321	
0022FF50	00000237	
0022FF54	000000EA	
0022FF58	0022FF78	Stack[000009B0]:0022FF78
0022FF5C	004012BF	_main+2F
0022FF60	77C0AEAD	msvcrt.dll:77C0AEAD
0022FF64	011E0DF4	

Resumiendo vemos en el stack que quedaron bien marcadas las 3 zonas, la amarilla de argumentos no usada en este caso, la celeste con el EBP guardado y el return address y la rosa para las 3 variables locales y de ahí hacia arriba habrá lugar para lo que necesite, por ejemplo para pasar argumentos a printf.

Vemos que lo que en el IDA descubrimos debuggeando, si paramos el debugger y miramos el listado muerto y hacemos doble click en cualquiera de las variables de nuestra funcion, IDA nos muestra lo mismo, sin tener que debuggear.

-00000010		db ? ; undefined
-0000000F		db ? ; undefined
-0000000E		db ? ; undefined
-0000000D		db ? ; undefined
-0000000C	suma	dd ?
-00000008	segundoNumero	dd ?
-00000004	primerNumero	dd ?
+00000000	s	db 4 dup(?)
+00000004	r	db 4 dup(?)
+00000008		
+00000008		; end of stack variables

Como habíamos visto el return address aquí llamado **r** es el limite, abajo del mismo esta la zona amarilla de argumentos (en este caso no hay argumentos), arriba del return address esta el EBP guardado (aquí llamado **s** por **stored ebp**) y arriba la zona rosada para las variables, allí están las tres que creamos, **primerNumero**, **segundoNumero** y **suma**, mas arriba se utilizara en el resto de nuestra funcion para pasar argumentos, en esta caso a printf.

Si queremos ver como decompila nuestra funcion el HexRays en pseudocode, vamos a VIEW-OPEN SUBVIEWS-PSEUDOCODE.

```
int __cdecl funcion1()
{
    printf("Su suma es %d", 801);
    return getchar();
}
```

Vemos que interpreta la funcion (ya que la misma no tiene argumentos ni variables dado que los

valores que suma son constantes siempre), como una funcion que imprime el mensaje **Su suma es 801** y nada mas.

En el caso que quisiéramos ver como decompila el código completo, hay que ir a FILE-PRODUCE FILE-CREATE C FILE y tratara de hacer algo mas o menos aproximado a nuestro código aunque no es exacto ni mucho menos.

En C las variables se pueden inicializar al mismo momento que se crean en ese caso el código quedaría así

```
# include <stdio.h>

main(){
    funcion1();
}

funcion1(){
    int primerNumero = 234;
    int segundoNumero = 567;
    int suma;

    suma = primerNumero + segundoNumero;
    printf("Su suma es %d", suma);

    getchar();
}
```

Si compilan este y lo abren en IDA y entran a nuestra funcion verán que es exactamente igual al anterior y no cambio nada.

El siguiente ejemplo usa la funcion **scanf**, para ingresar una string por teclado, y tal cual **printf**, permite format string, por lo cual si el primer argumento es %d, transformara la string tipeada en un numero, a la inversa de **printf**, eso si debemos colocar el & delante del nombre de la variable en **scanf** ya explicaremos mas adelante porque.

```
# include <stdio.h>

main(){
    funcionSuma();
}

funcionSuma(){
int primerNumero, segundoNumero, suma;    /* Nuestras variables */

    printf("Introduce el primer numero ");
    scanf("%d", &primerNumero);
    printf("Introduce el segundo numero ");
    scanf("%d", &segundoNumero);
    suma = primerNumero + segundoNumero;
    printf("Su suma es %d", suma);

    getchar();
}
```

Vemos que luego de declarar las tres variables como int, imprime el mensaje **Introduce el primer numero**, allí el usuario tipeara una string que debe representar un numero sino dará error, luego lo mismo para el segundo numero, realizara la suma la cual guardara en la variable suma, e imprimirá la salida mediante **printf**, compilemos este código y veamos nuestra funcion en IDA.

Allí estamos en el main y vemos nuestra funcion, entramos y la renombramos, lo mismo que las variables como hicimos en el ejemplo anterior.

```
; int __cdecl main(int argc,  
_main proc near  
  
var_4= dword ptr -4  
argc= dword ptr 8  
argv= dword ptr 0Ch  
envp= dword ptr 10h  
  
push    ebp  
mov     ebp, esp  
sub     esp, 8  
and     esp, 0FFFFFF0h  
mov     eax, 0  
add     eax, 0Fh  
add     eax, 0Fh  
shr     eax, 4  
shl     eax, 4  
mov     [ebp+var_4], eax  
mov     eax, [ebp+var_4]  
call    ___chkstk  
call    __main  
call    sub_4012C1  
leave  
retn    .
```

Quedara así:

```

funcionSuma proc near
    suma= dword ptr -0Ch
    segundoNumero= dword ptr -8
    primerNumero= dword ptr -4

    push    ebp
    mov     ebp, esp
    sub     esp, 18h
    mov     dword ptr [esp], offset aIntroduceElPrimerNumero ; "Introduce el primer numero "
    call    printf
    lea     eax, [ebp+primerNumero]
    mov     [esp+4], eax
    mov     dword ptr [esp], offset aD ; "%d"
    call    scanf
    mov     dword ptr [esp], offset aIntroduceElSegundoNumero ; "Introduce el segundo numero
    call    printf
    lea     eax, [ebp+segundoNumero]
    mov     [esp+4], eax
    mov     dword ptr [esp], offset aD ; "%d"
    call    scanf
    mov     eax, [ebp+segundoNumero]
    add     eax, [ebp+primerNumero]
    mov     [ebp+suma], eax
    mov     eax, [ebp+suma]
    mov     [esp+4], eax
    mov     dword ptr [esp], offset aSuSumaEsD ; "Su suma es %d"
    call    printf
    call    getchar
    leave
    retn

```

Si hacemos click en cualquiera de las variables vemos que la disposición de las misma no cambio nada con respecto al caso anterior.

```

-0000000E      db ? ; undefined
-0000000D      db ? ; undefined
-0000000C suma   dd ?
-00000008 segundoNumero dd ?
-00000004 primerNumero dd ?
+00000000 s     db 4 dup(?)
+00000004 r     db 4 dup(?)
+00000008
+00000008 ; end of stack variables

```

Todo lo explicado para el caso anterior aquí se cumple, tenemos el mismo `r` o return address, arriba el stored `ebp` y arriba las tres variables, por supuesto aquí tampoco tenemos argumentos así que la parte amarilla no la usamos.

```

lea    eax, [ebp+primerNumero]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf

```

Vemos aquí como se aclara el sentido del `&` que tuvimos que colocar delante de los nombres de las

En el primer ejemplo directamente inicializamos el valor de la variable con

pero esta variable es una variable local de mi funcion, solo tiene sentido dentro de la misma y no tiene sentido dentro de **scanf**, de esta forma al hacer el LEA obtengo la dirección en el stack de la variable

Luego si se le pasa esa dirección en el stack que esta en EAX como argumento de **scanf** como en nuestro ejemplo

como EAX tiene la dirección de la variable en el stack, al escribir en el contenido de EAX en realidad estamos inicializando la variable **primerNumero** y llenando su contenido.

```

, introduce: bp-based frame

funcionSuma proc near

suma= dword ptr -0Ch
segundoNumero= dword ptr -8
primerNumero= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     dword ptr [esp], offset aIntroduceElPri ; "Introd
call    printf
lea     eax, [ebp+primerNumero]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf

```

Al correr el debugger parara allí.

The screenshot shows a debugger window with the following components:

- Assembly View:**

```

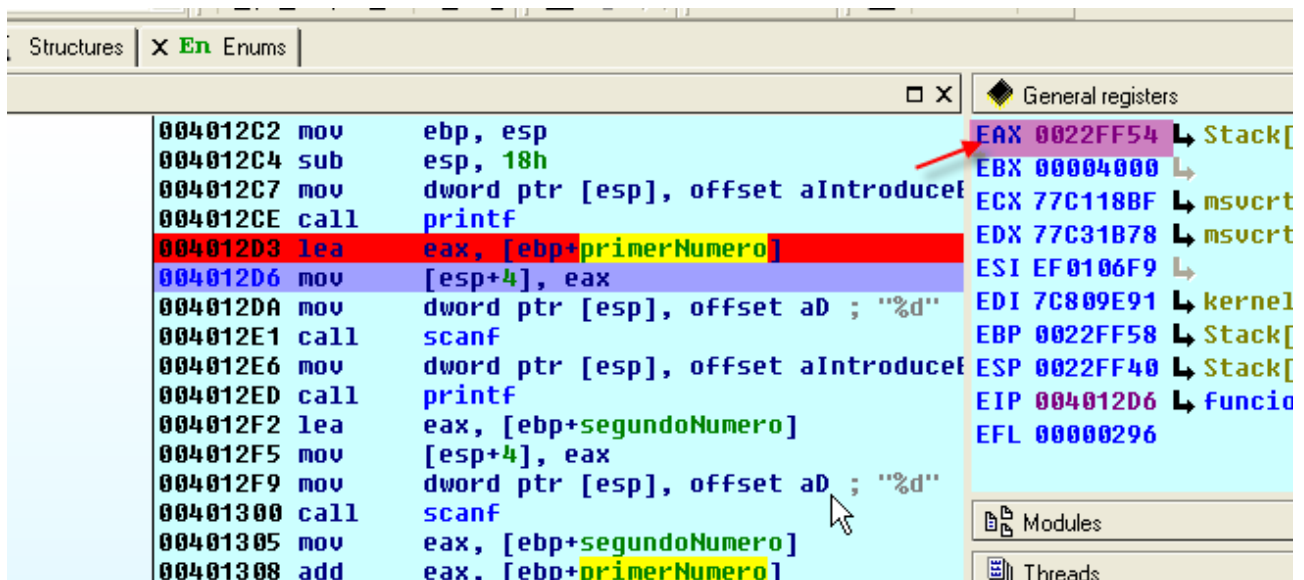
call    printf
lea     eax, [ebp+primerNumero]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
mov     dword ptr [esp], offset aIntroduceE
call    printf
lea     eax, [ebp+segundoNumero]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
mov     eax, [ebp+segundoNumero]
add     eax, [ebp+primerNumero]
mov     [ebp+suma], eax
mov     eax, [ebp+suma]
mov     [esp+4], eax
mov     dword ptr [esp], offset aSuSumaEsD

```
- Registers:**
  - ECX: 77C178BF → msvcrt.dll:77C178BF
  - EDX: 77C31B78 → msvcrt.dll:77C31B78
  - ESI: EF0106F9 →
  - EDI: 7C809E91 → kernel32.dll:7C809E91
  - EBP: 0022FF58 → Stack[00000194]:0022FF58
  - ESP: 0022FF40 → Stack[00000194]:0022FF40
  - EIP: 004012D3 → funcionSuma+12
  - EFL: 00000296
- Threads:**

Decimal	Hex	State
404	194	Ready
- Stack View:**

Address	Value	Comment
0022FF40	00403000	.rdata:aIntroduceEIP
0022FF44	00401370	__do_global_dtors
0022FF48	0022FF58	Stack[00000194]:0022FF58
0022FF4C	00401456	__main:loc_401456
0022FF50	00401370	__do_global_dtors
0022FF54	00004000	
0022FF58	0022FF78	Stack[00000194]:0022FF78
0022FF5C	004012BF	__main+2F
0022FF60	77C0AEAD	msvcrt.dll:77C0AEAD

Al cambiar con JMP TO ESP vemos en **22ff5c** el return address, en **22ff58** el stored EBP, y justo arriba deben ubicarse las tres variables que tienen aun cualquier valor pues no han sido inicializadas. La variable primer numero se encuentra en **22ff54** y vale 4000 que es la fruta que había ahí en el stack, de esta forma al hacer el LEA, queda en EAX la dirección 22ff54 de la variable, pues eso es lo que me interesa pasarle a **scanf**, para que escriba allí y no el **4000** que es basura que hubiera quedado en EAX de haber hecho **mov** en vez de **lea**.



De esta forma se ve claro que al pasarle el la dirección dela variable en EAX, en cualquier momento que se haga

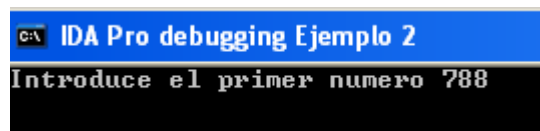
**mov [EAX], XXX**

sera lo mismo que hacer

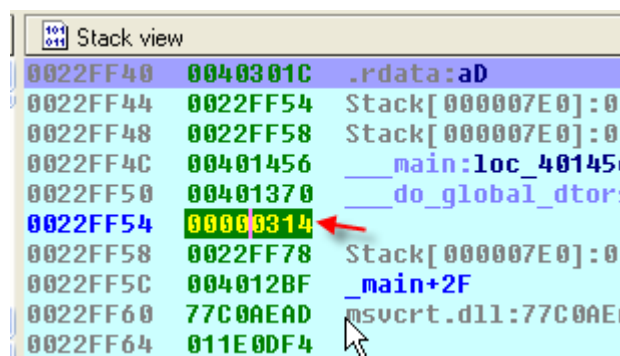
**mov [22ff54],XXX**

Lo que guardara el valor **XXX** en dicha variable remplazando el 4000 que era basura y se inicializara.

Luego si seguimos traceando vemos que en la consola nos pedirá que tipeemos un numero, en mi caso tipee 788 y ENTER.



Al volver veo que guardo en hexa dicho valor en la variable **primerNumero**.



Luego lo mismo guarda el **segundoNumero** justo arriba y la **suma** en la tercera variable, luego



imprime por consola la salida, lo que no funciona en este caso es el bloqueo mediante getch, debe quedar como que mantenemos apretado ENTER y continua y se cierra.

```
main(){
    funcionSuma();
    getch();
}
```

```
funcionSuma(){
int primerNumero, segundoNumero, suma;    /* Nuestras variables */

    printf("Introduce el primer numero ");
    scanf("%d", &primerNumero);
    printf("Introduce el segundo numero ");
    scanf("%d", &segundoNumero);
    suma = primerNumero + segundoNumero;
    printf("Su suma es %d", suma);

    getch();
}
```

Bueno solucionamos el problema agregando otro getch el cual si detiene el programa hasta que apretamos ENTER nuevamente.

Vemos que en este caso el pseudocode creado por el HexRays no puede evitar la creación de 2 variables ya que allí se debe guardar lo que tipea el usuario.

```
int __cdecl sub_4012C6()
{
    int v1; // [sp+10h] [bp-8h]@1
    int v2; // [sp+14h] [bp-4h]@1

    printf("Introduce el primer numero ");
    scanf("%d", &v2);
    printf("Introduce el segundo numero ");
    scanf("%d", &v1);
    printf("Su suma es %d", v2 + v1);
    return getch();
}
```

Crea dos variables llamadas **v1** y **v2** y en el **scanf** le pone el **&** delante para obtener la dirección de las mismas en el stack, es casi el mismo código que el nuestro salvo que usa una variable menos pues no guarda el resultado de la suma.

Bueno la verdad que me canse y no se si a alguien le interesa profundizar en reversing de códigos cada vez mas complejos, pero bueno sino quedara esta parte 1 como parte única para el que le sirva, veremos también las ganas que tenemos de seguir una parte 2 jejeje.

Un saludo a todos los Crackslatinos  
Ricardo Narvaja