

INCLUDE Y DEFINE

Aquí copiare directo de la teoría de Cabanes que esta muy bien explicada, esto que viene sobretodo las macros son mas importantes para el que programa que para un reverser, ya que estas macros son resueltas antes de compilar y no se podrá ver nada diferente en el IDA, si se define una macro se vera en el IDA el resultado de aplicar la misma directamente y sera difícil adivinar de donde provino pues no veremos código ni nada, pero es bueno conocer del tema aunque sea por encima así que aquí va la teoría,

Directivas del preprocesador

Desde el principio hemos estado manejando cosas como

#include <stdio.h>

Y aquí hay que comentar bastante más de lo que parece. Ese “include” no es una orden del lenguaje C, sino una orden directa al compilador (una “directiva”). Realmente es una orden a una cierta parte del compilador que se llama **“preprocesador”**. Estas directivas indican una serie de pasos que se deben dar antes de empezar realmente a traducir nuestro programa fuente.

Aunque “include” es la directiva que ya conocemos, vamos a comenzar por otra más sencilla, y que nos resultará útil cuando llegemos a ésta.

Constantes simbólicas: #define

La directiva “define” permite crear “constantes simbólicas”. Podemos crear una constante haciendo

#define MAXINTENTOS 10

y en nuestro programa lo usaríamos como si se tratara de cualquier variable o de cualquier valor numérico:

if (intentoActual >= MAXINTENTOS) ...

El primer paso que hace nuestro compilador es reemplazar esa “falsa constante” por su valor, de modo que la orden que realmente va a analizar es

if (intentoActual >= 10) ...

pero a cambio nosotros tenemos el valor numérico sólo al principio del programa, por lo que es muy fácil de modificar, mucho más que si tuviéramos que revisar el programa entero buscando dónde aparece ese 10.

Comparado con las constantes “de verdad”, que ya habíamos manejado (const int MAXINTENTOS=10;), las constantes simbólicas tienen la ventaja de que no son variables, por lo que no se les reserva memoria adicional y las comparaciones y demás operaciones suelen ser más rápidas que en el caso de un variable.

Vamos a ver un ejemplo completo, que pida varios números y muestre su suma y su media:

```

#include <stdio.h>

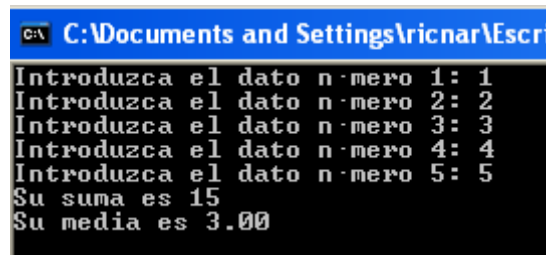
#define CANTIDADNUMEROS 5

int main() {
    int numero[CANTIDADNUMEROS];
    int suma=0;
    int i;

    for (i=0; i<CANTIDADNUMEROS; i++) {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &numero[i]);
    }

    for (i=0; i<CANTIDADNUMEROS; i++)
        suma += numero[i];
    printf("Su suma es %d\n", suma);
    printf("Su media es %4.2f\n", (float) suma/CANTIDADNUMEROS);
    getchar();
    getchar();
    return 0;
}

```



```

C:\Documents and Settings\ricnar\Escri...
Introduzca el dato número 1: 1
Introduzca el dato número 2: 2
Introduzca el dato número 3: 3
Introduzca el dato número 4: 4
Introduzca el dato número 5: 5
Su suma es 15
Su media es 3.00

```

Vemos que es un programita muy sencillo crea un array de enteros de tamaño 5, ya que el **define** del inicio hace que **CANTIDADNUMEROS** sea igual a 5 en todos los casos

int numero[CANTIDADNUMEROS];

sera igual a

int numero[5];

y así el preprocesador reemplazara donde encuentre la palabra CANTIDADENumeros por el valor 5 y lo compilara.

De esta forma el código anterior seria equivalente a

```

#include <stdio.h>

int main() {
    int numero[5];
    int suma=0;
    int i;

    for (i=0; i<5; i++) {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &numero[i]);
    }
}

```

```

for (i=0; i<5; i++)
    suma += numero[i];
printf("Su suma es %d\n", suma);
printf("Su media es %.2f\n", (float) suma/5);
getchar();
getchar();
return 0;
}

```

The screenshot shows a C program being executed. The source code in the background defines an array of 5 integers, calculates their sum, and prints the sum and average. The terminal window in the foreground shows the user inputting the numbers 1 through 5, followed by the program's output: 'Su suma es 15' and 'Su media es 3.00'.

```

#include <stdio.h>

int main() {
    int numero[5];
    int suma=0;
    int i;

    for (i=0; i<5; i++)
        scanf("%d", &numero[i]);

    for (i=0; i<5; i++)
        suma += numero[i];

    printf("Su suma es %d\n", suma);
    printf("Su media es %.2f\n", (float) suma/5);
    getchar();
    getchar();
    return 0;
}

```

Terminal Output:

```

Introduzca el dato n-mero 1: 1
Introduzca el dato n-mero 2: 2
Introduzca el dato n-mero 3: 3
Introduzca el dato n-mero 4: 4
Introduzca el dato n-mero 5: 5
Su suma es 15
Su media es 3.00

```

Si compilamos de nuevo el original con el define y lo vemos en IDA.

```

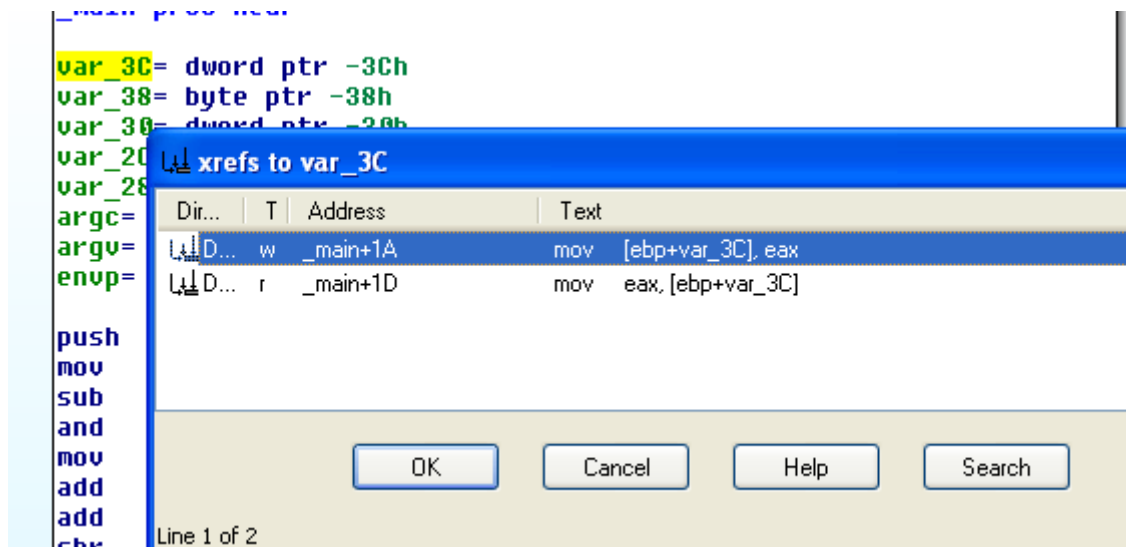
_main proc near

var_3C= dword ptr -3Ch
var_38= byte ptr -38h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 58h
and     esp, 0FFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_3C], eax
mov     eax, [ebp+var_3C]
call    ___chkstk
call    __main

```

Allí vemos las variables y argumentos sabemos que hay variables y argumentos que agrego el compilador, pero si no estamos seguros cuales son, pues empezemos por la superior una por una a ver que es cada una.



Apretando la X en la primera vemos de donde es llamada y eso es aquí:

```

var_3C= dword ptr -3Ch
var_38= byte ptr -38h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 58h
and     esp, 0FFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_3C], eax
mov     eax, [ebp+var_3C]
call    ___chkstk
call    main

```

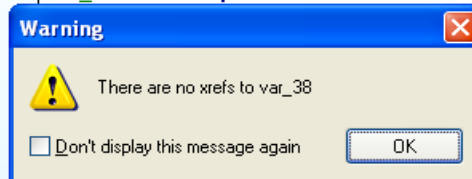
Así que esa solo trabaja en la parte anterior a que empiece el código real, así que es agregada por el compilador.

La siguiente ni siquiera tiene referencias así que lo mismo es creada por el compilador

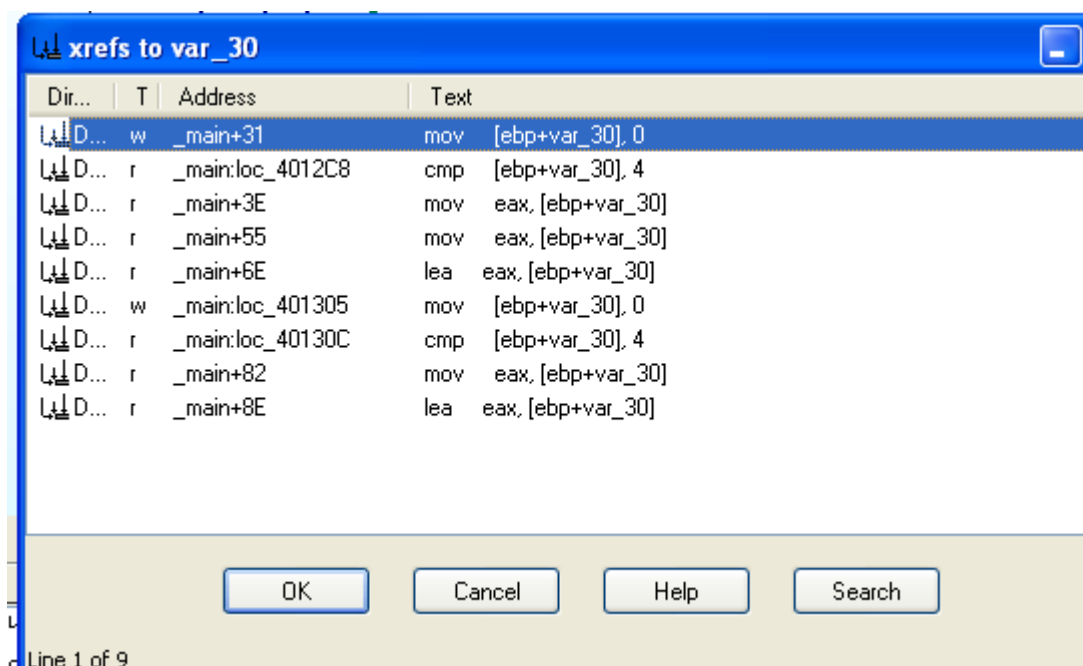
```

var_3C= dword ptr -3Ch
var_38= byte ptr -38h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h

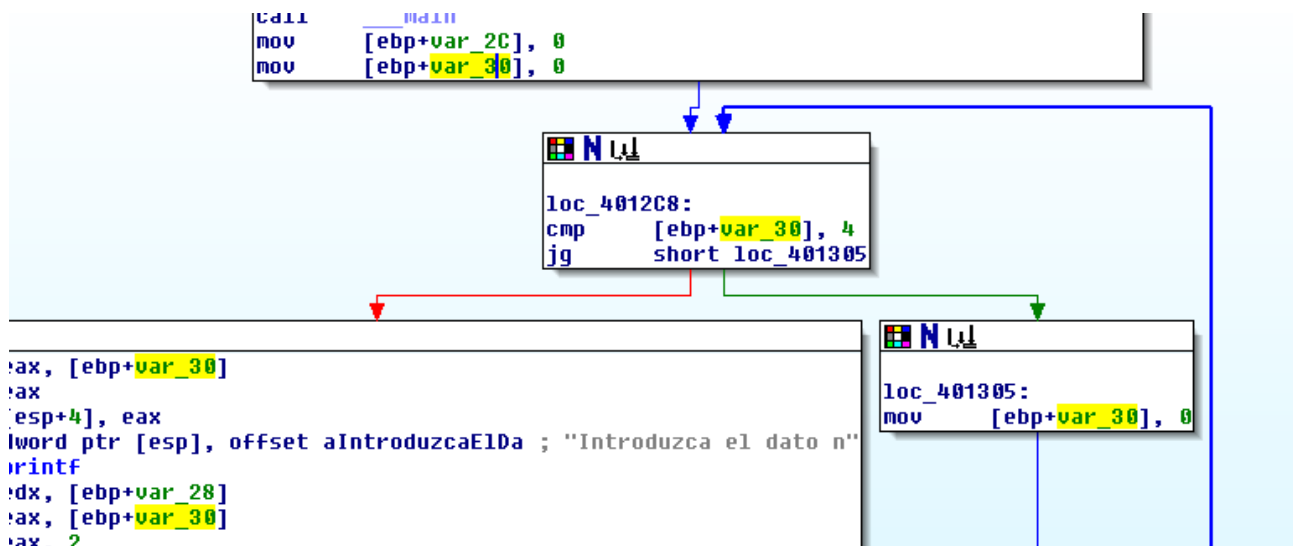
```



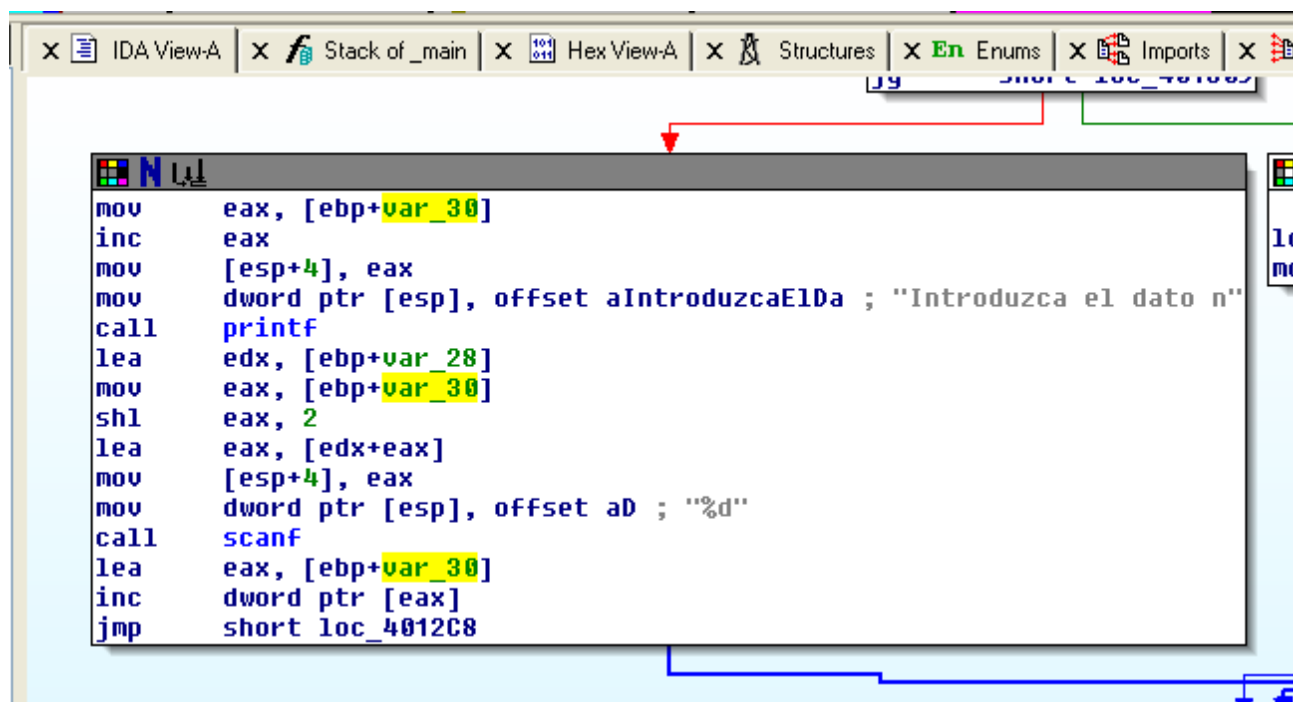
La tercera ya es usada en el código real veamos que es.



Vemos que es inicializada con cero aquí y luego comparada con 4.



Se ve claramente que es el contador de un loop, mas adelante hay otra inicializacion con cero, y compara con cuatro, así que es posible que se reuse varias veces como contador.



Dentro del loop la vemos usada tres veces, la primera para pasarle al printf el numero de dato que debemos ingresar ya que hace format string usando el valor del contador mas uno, así cuando en el loop el contador valga cero, sin cambiar el valor de la variable se moverá a EAX y allí **on the fly** se incrementa a uno y luego se imprime **“Introduzca el dato numero 1”**.

Se entiende que el **“on the fly”** lo uso porque **contador** no cambia su valor por el INC sino que lo hace al vuelo incrementando en un registro.

```

lea     edx, [ebp+var_28]
mov     eax, [ebp+var_30]
shl     eax, 2

```

Aquí lo usa para recorrer un array de enteros, aunque no tenga el código fuente me doy cuenta que levanta la dirección del inicio de un array con el LEA y mueve a EAX el valor de contador y lo


multiplica por 4 (**SHL EAX, 2**)

Así que cuando el contador valga 0 luego del SHL, tendremos que EAX valdrá 0, en el próximo ciclo cuando contador valga 1, EAX valdrá 4 y así al sumarlo a la dirección de inicio del array podremos usarlo como índice para recorrer los campos del mismo.

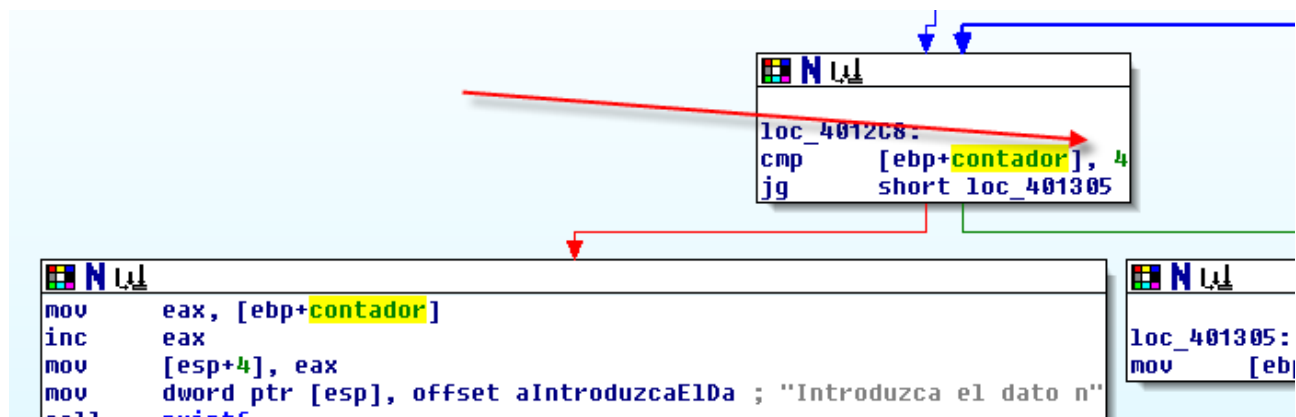
```
lea    eax, [edx+eax]
mov    [esp+4], eax
mov    dword ptr [esp], offset aD ; "%d"
call   scanf
```

Eso es lo que hace suma usando LEA la dirección de inicio del array que estaba en EDX con EAX obteniendo la dirección del campo actual del array y eso lo pasa como argumento a **scanf** para que ingresemos los datos y vayamos llenando los campos del array en cada ciclo del loop, así que podemos ir arreglando todo esto, podemos renombrar la **var_30** a **contador**.

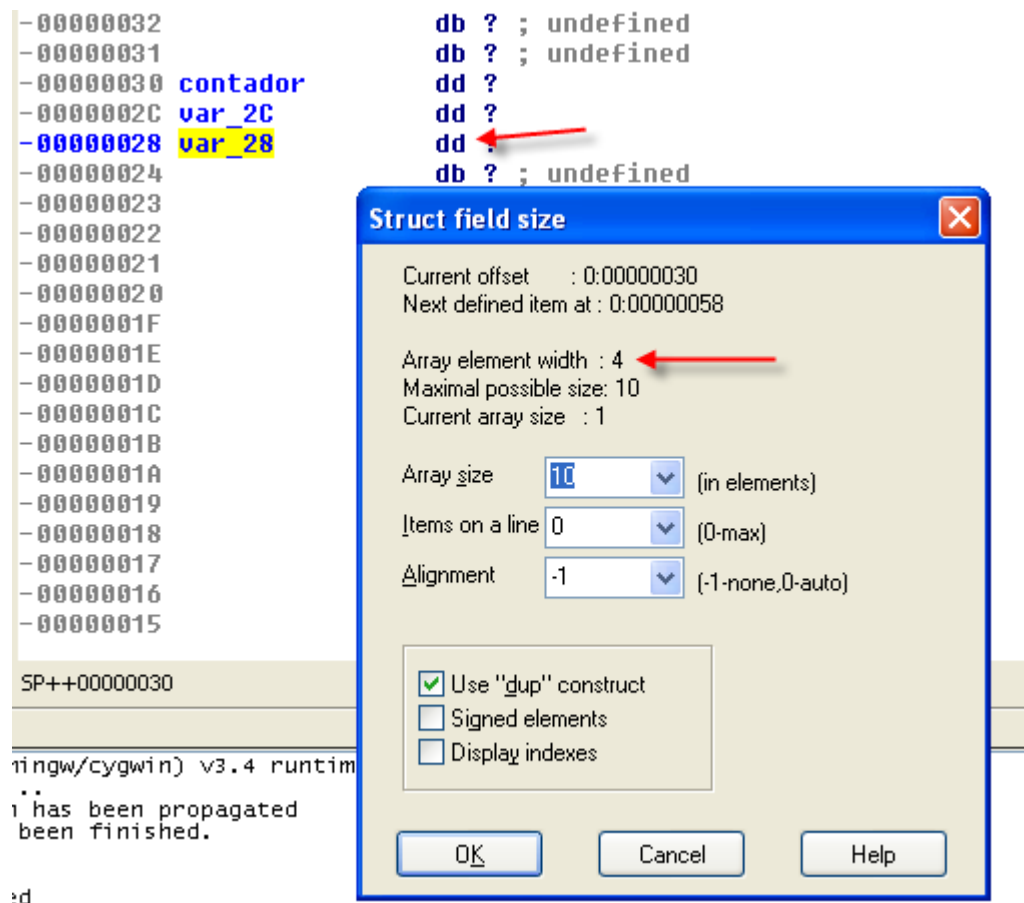
-00000031		db ? ; undefined
-00000030	contador	dd ?
-0000002C	var_2C	dd ?
-00000028	var_28	dd ?
-00000024		db ? ; undefined
-00000023		db ? ; undefined
-00000022		db ? ; undefined
-00000021		db ? ; undefined
-00000020		db ? ; undefined
-0000001F		db ? ; undefined
-0000001E		db ? ; undefined
-0000001D		db ? ; undefined
-0000001C		db ? ; undefined
-0000001B		db ? ; undefined



Y la **var_28** vimos que era un array ya que ahí en el loop se inicializaban los campos, sabemos que nos pide 5 datos enteros y que el for sale cuando es mas grande que 4.



Por lo tanto podemos suponer que el array tiene 5 campos enteros de cualquier forma si uno usa todo el espacio vacío y no hay ninguna otra referencia en el programa que nos diga lo contrario, no esta mal, así que vamos a crear el array.



Verificamos que el tamaño de cada campo sera correcto en este caso 4 ya que es un array de enteros, cambiamos el array size a 5 y apretamos OK.

```

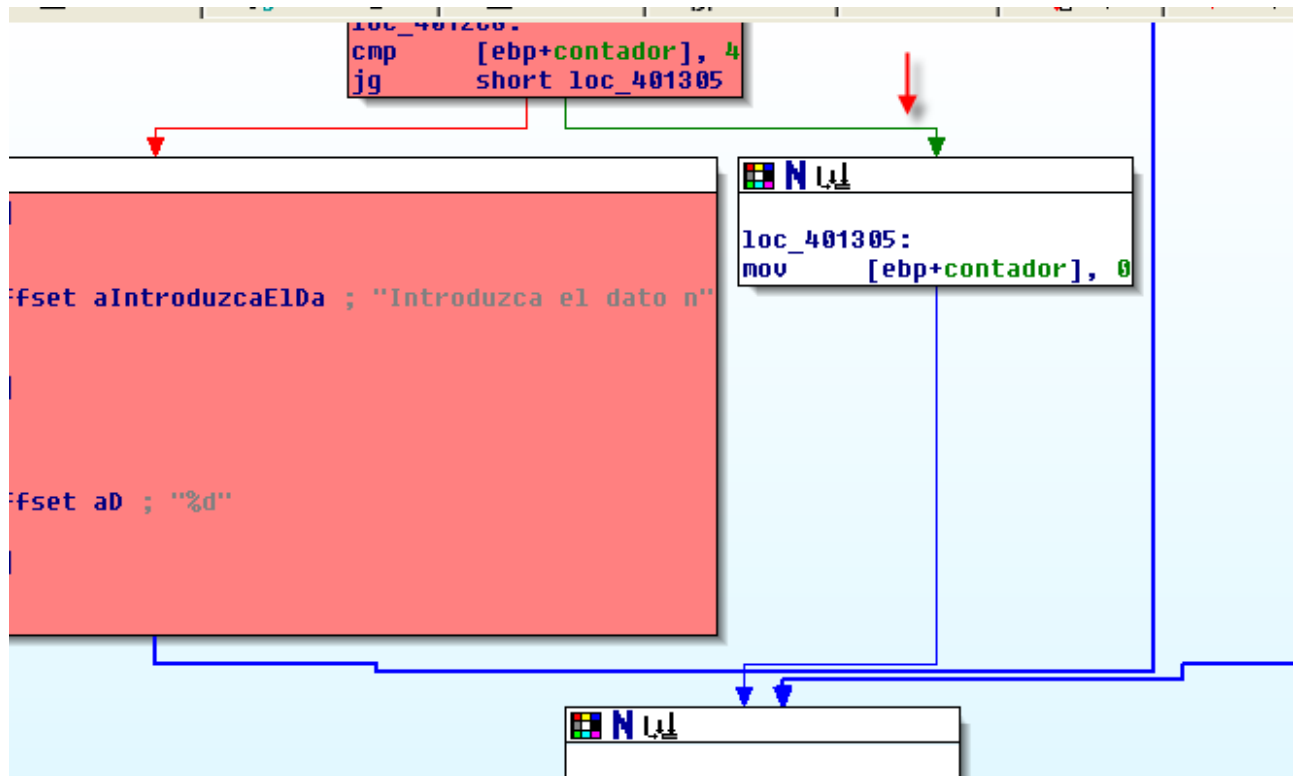
-00000031          db ? ; undefined
-00000030  contador dd ?
-0000002C  var_2C   dd ?
-00000028  datos   dd 5 dup(?)
-00000014          db ? ; undefined
-00000013          db ? ; undefined
-00000012          db ? ; undefined
-00000011          db ? ; undefined
-00000010          db ? ; undefined
-0000000F          db ? ; undefined
-0000000E          db ? ; undefined
-0000000D          db ? ; undefined
-0000000C          db ? ; undefined
-0000000B          db ? ; undefined
-0000000A          db ? ; undefined
-00000009          db ? ; undefined
-00000008          db ? ; undefined
-00000007          db ? ; undefined
-00000006          db ? ; undefined
-00000005          db ? ; undefined
-00000004          db ? ; undefined
-00000003          db ? ; undefined
-00000002          db ? ; undefined
-00000001          db ? ; undefined
+00000000          db 4 dup(?)
+00000004          db 4 dup(?)
+00000008  argc    dd ?

```

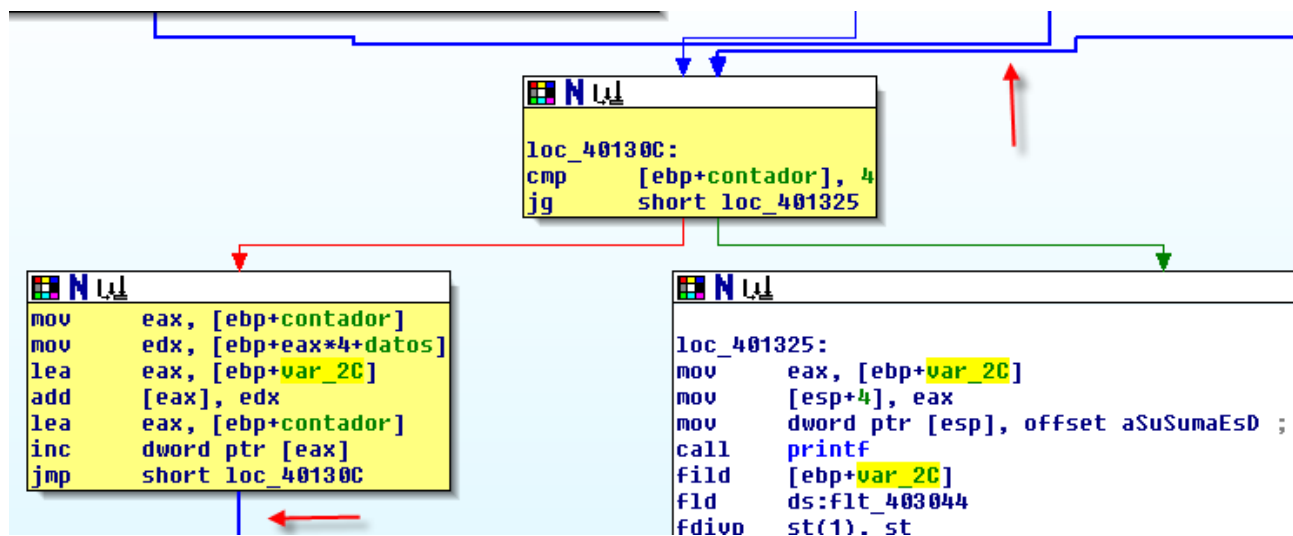
Quedo un poco de espacio vacío que dejo el compilador pero no hay problema renombro la variable

array a **datos**.

Una vez que sale del loop que inicializa los datos en el array el cual pinte de rosado, vemos que se vuelve a poner a cero la variable contador y parece haber otro loop-



Y si hay un segundo loop allí pintado de amarillo



Mientras que contador sea menor o igual a cuatro se repetirá el loop yendo por los bloques amarillos, veamos que hace dentro del mismo.

Lee el contador del loop

```
mov    eax, [ebp+contador]
```

Mueve a EDX los valores ingresados en cada campo

```
mov    edx, [ebp+eax*4+datos]
```

Creo que es fácil de ver que cuando contador vale

contador=0

```
mov    edx, [ebp+0*4+datos]
```

```
mov    edx, [ebp+datos]
```

Quedara en EDX el valor guardado en el primer campo del array

contador=1

```
mov    edx, [ebp+1*4+datos]
```

```
mov    edx, [ebp+4+datos]
```

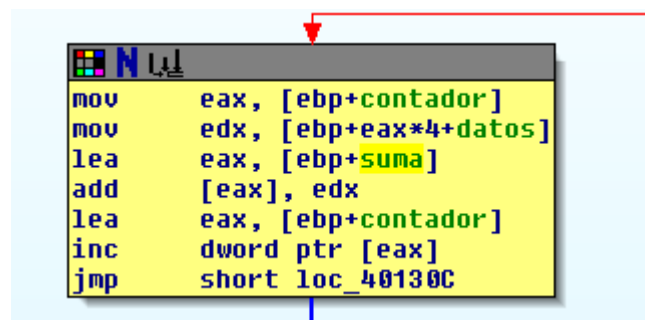
Quedara en EDX el valor guardado en el segundo campo del array y así sucesivamente.

Luego usa una **var_2c** para ir sumando todos los valores guardados en los campos.

```
lea    eax, [ebp+var_2C]
```

```
add    [eax], edx
```

Así que vemos que la **var_2c** que nos quedaba renombrar podemos llamarla **suma** ya que allí guarda la suma de los valores que ingresamos.



Luego incrementa el contador y va al inicio del loop donde se fijara si el mismo es mayor que cuatro para salir del mismo.

```

loc_40130C:
cmp     [ebp+contador], 4
jg      short loc_401325

loc_401325:
mov     eax, [ebp+suma]
mov     [esp+4], eax
mov     dword ptr [esp], offset aSuSumaEsD ; "Su suma es %d\n"
call    printf
fild    [ebp+suma]
fld     ds:flt_403044
fdivp   st(1), st
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset aSuMediaEs4_2f ; "Su media es %4.2f\n"
call    printf
call    getchar
call    getchar
mov     eax, 0
leave
retn
_main endp

```

Luego de salir del loop imprime usando **printf** el valor de suma usando format string.

```

fild    [ebp+suma]
fld     ds:flt_403044
fdivp   st(1), st
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset aSuMediaEs4_2f ; "Su media es %4.2f\n"

```

Luego usando el stack de punto flotante carga el valor de la suma en el mismo, y luego un 5 que lo guardo como variable global en **403044**, luego hará la división entre ambos lo cual es la media, la cual imprimirá.

El chiste de todo esto es que nunca vemos CANTIDADNUMEROS sino que para nosotros al reversear sera 5 la variable global y sera cinco la cantidad de veces que loopeara y ni nos enteramos de lo que hizo el preprocesador con el define, reverseando llegaríamos a esto:

```

#include <stdio.h>

int main() {
    int datos[5];
    int suma=0;
    int contador;

    for (contador=0; contador<5; contador++) {
        printf("Introduzca el dato número %d: ", contador+1);
        scanf("%d", &datos[contador]);
    }

    for (contador=0; contador<5; contador++)
        suma += datos[contador];
    printf("Su suma es %d\n", suma);
    printf("Su media es %4.2f\n", (float) suma/5);
    getchar();
    getchar();
    return 0;
}

```

A “define” también se le puede dar también un uso más avanzado: se puede crear “macros”, que en vez de limitarse a lo antes visto, pueden comportarse como pequeñas órdenes, más rápidas que una función. Un ejemplo podría ser:

```
#define SUMA(x,y) x+y
```

aquí el código

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

```
#define SUMA(x,y) x+y
```

```
int main() {  
    int n1, n2;  
  
    printf("Introduzca el primer dato: ");  
    scanf("%d", &n1);  
  
    printf("Introduzca el segundo dato: ");  
    scanf("%d", &n2);  
  
    printf("Su suma es %d\n", SUMA(n1,n2));  
  
    getchar();  
    getchar();  
    return 0;  
}
```

Lo que hará el preprocesador es un replace antes de compilar, o sea donde halle el texto **SUMA(x,y)** lo reemplazara por el texto **x+y** y para el reverser el código sera.

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

```
int main() {  
    int n1, n2;  
  
    printf("Introduzca el primer dato: ");  
    scanf("%d", &n1);  
  
    printf("Introduzca el segundo dato: ");
```

```
scanf("%d", &n2);
```

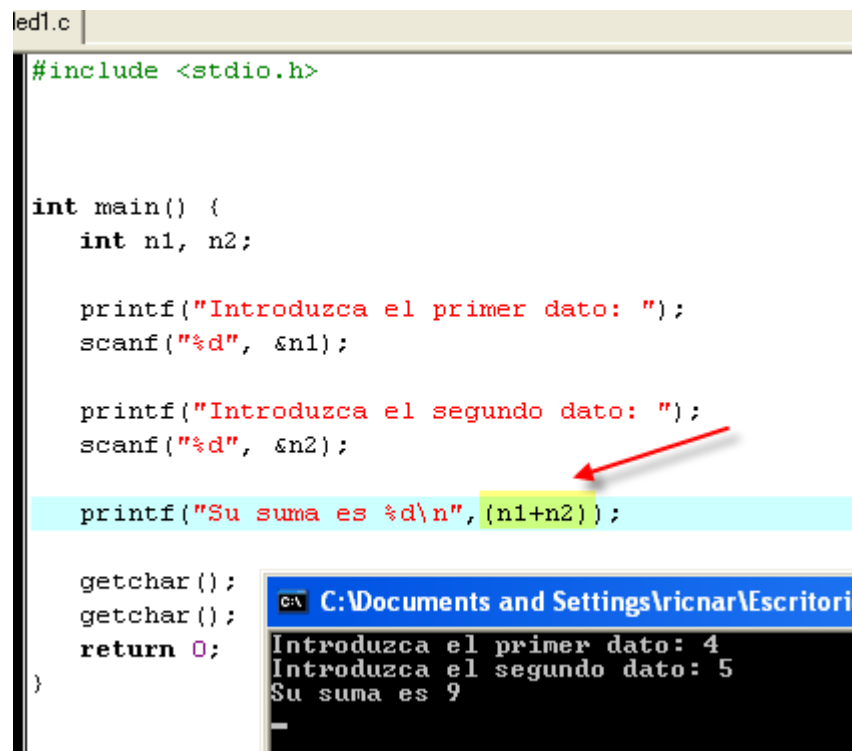
```
printf("Su suma es %d\n", (n1+n2));
```

```
getchar();
```

```
getchar();
```

```
return 0;
```

```
}
```



The image shows a code editor window titled 'led1.c' containing a C program. The program prompts the user to enter two integers, calculates their sum, and prints the result. The line `printf("Su suma es %d\n", (n1+n2));` is highlighted in light blue, and the expression `(n1+n2)` is highlighted in yellow. A red arrow points from the yellow highlight to a terminal window. The terminal window shows the program's execution: it prompts for the first and second data, receives inputs 4 and 5, and outputs 'Su suma es 9'.

```
led1.c |
#include <stdio.h>

int main() {
    int n1, n2;

    printf("Introduzca el primer dato: ");
    scanf("%d", &n1);

    printf("Introduzca el segundo dato: ");
    scanf("%d", &n2);

    printf("Su suma es %d\n", (n1+n2));

    getchar();
    getchar();
    return 0;
}
```

C:\Documents and Settings\ricnar\Escritorio
Introduzca el primer dato: 4
Introduzca el segundo dato: 5
Su suma es 9
_

Si lo vemos en IDA vemos que no existe ninguna función SUMA ni nada por el estilo, el preprocesador reemplazó el texto y quedó una suma directa entre dos variables `int`.

```

mov     [ebp+var_0], eax
mov     eax, [ebp+var_C]
call    ___chkstk
call    ___main
mov     dword ptr [esp], offset aIntroduzcaElPr ; "Introduzca el primer dato: "
call    printf
lea     eax, [ebp+var_4]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
mov     dword ptr [esp], offset aIntroduzcaElse ; "Introduzca el segundo dato: "
call    printf
lea     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
mov     eax, [ebp+var_8]
add     eax, [ebp+var_4]
mov     [esp+4], eax
mov     dword ptr [esp], offset aSuSumaEsD ; "Su suma es %d\n"
call    printf

```

Así que el tema macros a pesar de ser útil para programar es como magia anterior a la compilación, puro replace de texto que realmente el reverser nunca vera, solo hallara, si lo hace correctamente el código que el preprocesador ya manipulo y esta listo para la compilación final que sera igualmente funcional que el original aunque menos elegante posiblemente.

Volvamos a la teoría de Cabanes:

Inclusión de ficheros: #include

Ya nos habíamos encontrado con esta directiva. Lo que hace es que cuando llega el momento de que nuestro compilador compruebe la sintaxis de nuestro fuente en C, ya no existe ese “include”, sino que en su lugar el compilador ya ha insertado los ficheros que le hemos indicado.

¿Y eso de por qué se escribe <stdio.h>, entre < y >? No es la única forma de usar #include. Podemos encontrar líneas como

#include <stdlib.h>

y como

#include "misdatos.h"

El primer caso es un fichero de cabecera **estándar** del compilador. Lo indicamos entre < y > y así el compilador sabe que tiene que buscarlo en su directorio (carpeta) de “includes”. El segundo caso es un fichero de cabecera que hemos creado **nosotros**, por lo que lo indicamos entre comillas, y así el compilador sabe que no debe buscarlo entre sus directorios, sino en el mismo directorio en el que está nuestro programa.

Vamos a ver un ejemplo: declararemos una función “suma” dentro de un fichero “.h” y lo incluiremos en nuestro fuente para poder utilizar esa función “suma” sin volver a definirla. El fichero de cabecera se llamaría **c096.h**:

Lo creamos como archivo de texto lo renombramos a **c096.h**, le pegamos este contenido, colocandolo en la misma carpeta donde estará el archivo fuente que lo llamara.

```

int suma(int x,int y) {
    return x+y;
}

```

(Nota: si somos puristas, esto no es correcto del todo. Un fichero de cabecera no debería contener los detalles de las funciones, sólo su “cabecera”, lo que habíamos llamado el “prototipo”, y la implementación de la función debería estar en otro fichero, pero eso lo haremos dentro de poco).

Un fuente que utilizara este fichero de cabecera, lo creamos y lo guardamos en la misma carpeta del anterior.

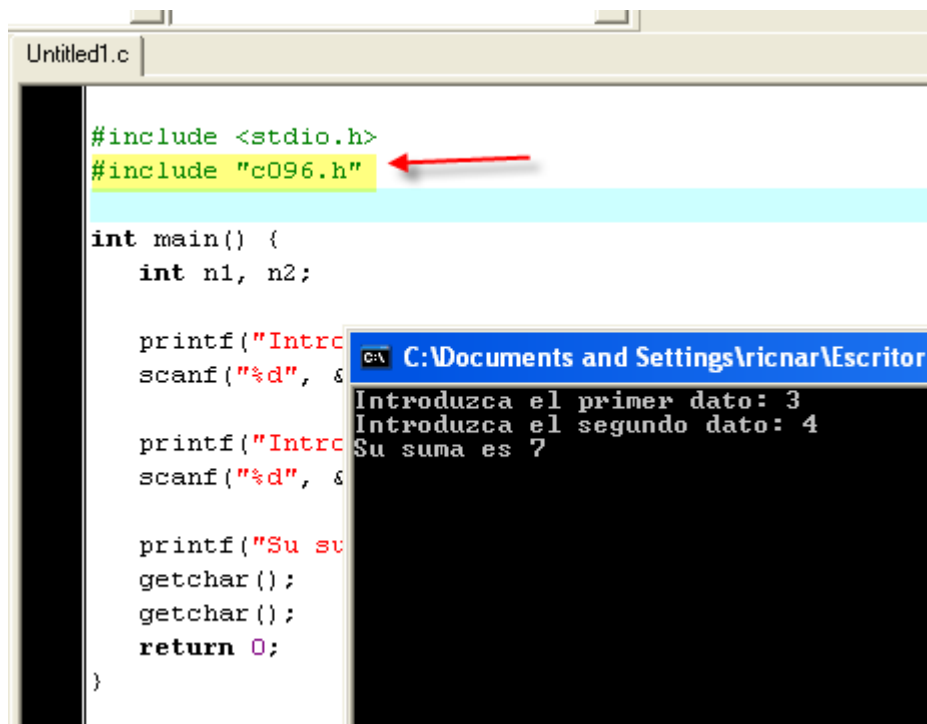
```
#include <stdio.h>
#include "c096.h"

int main() {
    int n1, n2;

    printf("Introduzca el primer dato: ");
    scanf("%d", &n1);

    printf("Introduzca el segundo dato: ");
    scanf("%d", &n2);

    printf("Su suma es %d\n", suma(n1,n2));
    getchar();
    getchar();
    return 0;
}
```



```
Untitled1.c
#include <stdio.h>
#include "c096.h"

int main() {
    int n1, n2;

    printf("Introduzca el primer dato: ");
    scanf("%d", &n1);

    printf("Introduzca el segundo dato: ");
    scanf("%d", &n2);

    printf("Su suma es %d\n", suma(n1,n2));
    getchar();
    getchar();
    return 0;
}
```

C:\Documents and Settings\ricnar\Escritorio

Introduzca el primer dato: 3
Introduzca el segundo dato: 4
Su suma es 7

Vemos que encuentra perfectamente al archivo **c096.h** y usa la funcion **suma** que esta en el mismo.

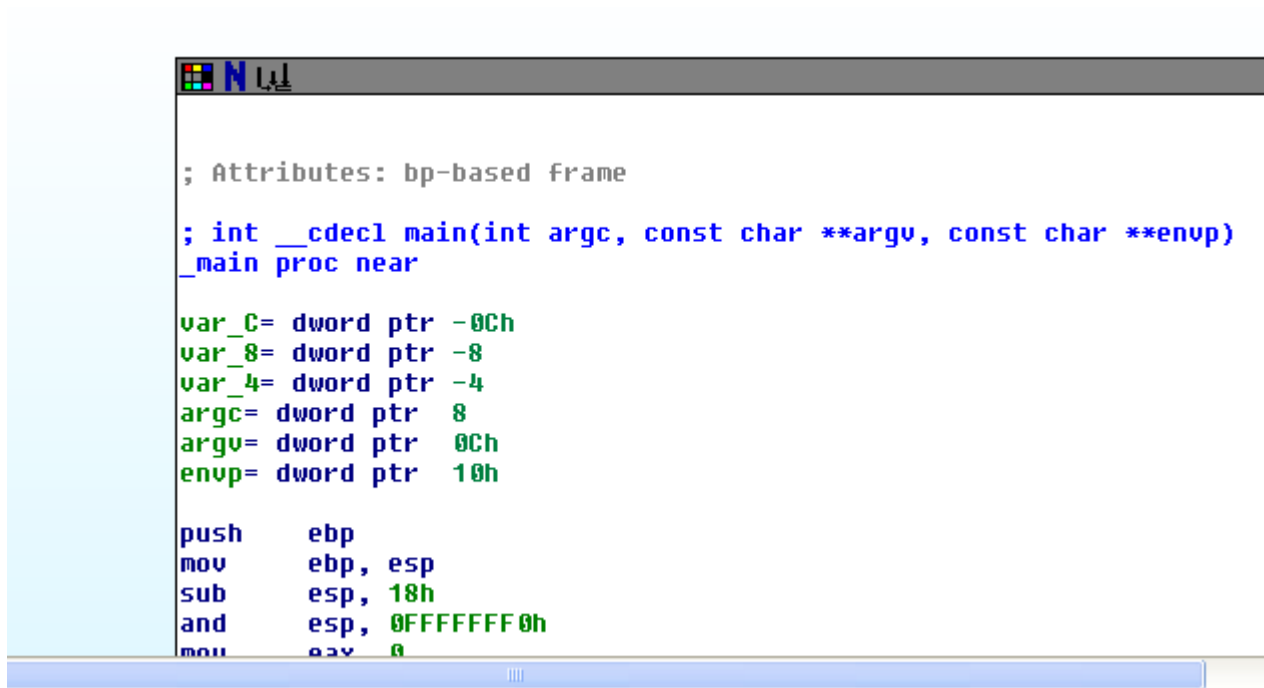
```

printf("Introduzca el segundo dato: ");
scanf("%d", &n2);

printf("Su suma es %d\n", suma(n1,n2));
getchar();
getchar();
return 0;
}

```

Vemos en el IDA que el reverser no ve que la funcion usada no esta en el .c del fuente.



```

; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

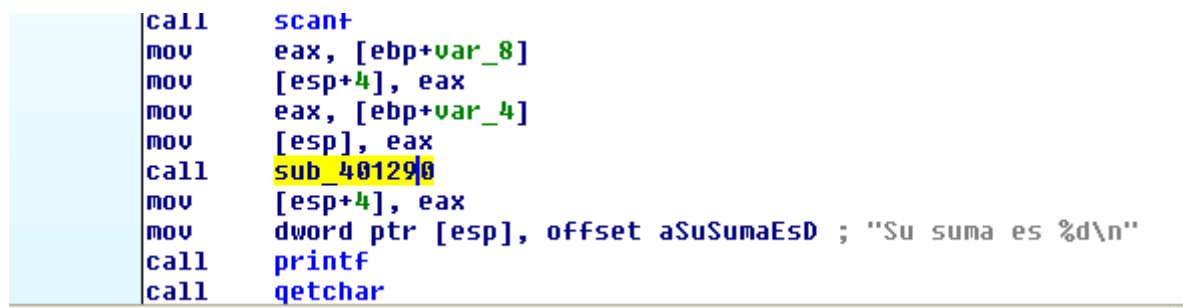
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 18h
and     esp, 0FFFFFF0h
mov     eax, 0

call    sub_401290

```

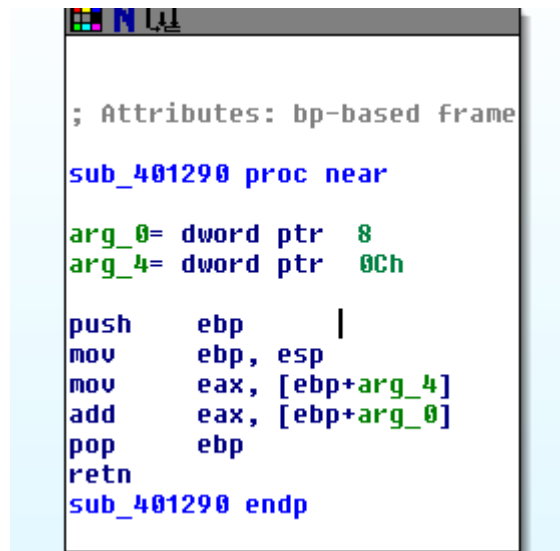
Solo aparece una funcion mas que es la que hará la suma pero podría ser una funcion incluida en el .c perfectamente y no podríamos diferenciarlo aquí.



```

call    scanf
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     eax, [ebp+var_4]
mov     [esp], eax
call    sub_401290
mov     [esp+4], eax
mov     dword ptr [esp], offset aSuSumaEsD ; "Su suma es %d\n"
call    printf
call    getchar

```

```
; Attributes: bp-based frame

sub_401290 proc near

arg_0= dword ptr 8
arg_4= dword ptr 0Ch

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_4]
add     eax, [ebp+arg_0]
pop     ebp
retn
sub_401290 endp
```

Allí la función que realiza la suma.

Bueno esta semana no habrá ejercicios porque el tema es poco dado para el reversing aunque había que conocerlo, así que los veo la semana que viene con la siguiente parte.

Ricnar