

C Y REVERSING (parte 2) por Ricnar

Bueno ahora lo que sigue es ver un poco el tamaño de las variables que declaramos, el que tiene alguna duda de los sistemas numéricos decimal, hexadecimal y octal no lo repasaremos aquí, puede leer la parte dos del curso de Cabanes donde esta todo bien explicado y sencillo.

<http://www.nachocabanes.com/c/curso/cc02.php>

Lo que hemos visto al reversear es que al crear una variable como **int**, el sistema le asigna un lugar en la memoria, que es un dword o sea 4 bytes, ademas si no especificamos nada y solo declaramos **int**, este valor sera tomado como **signed int**, o sea que se considerara con signo, que en hexa significa que desde **00000000** hasta **7fffffff** se consideraran números positivos y a partir de **80000000** hasta **fffffff** se consideraran los negativos siendo el -1 igual a **fffffff**, el -2 sera igual a **fffffffe** y así hasta el máximo negativo que sera el **80000000**.

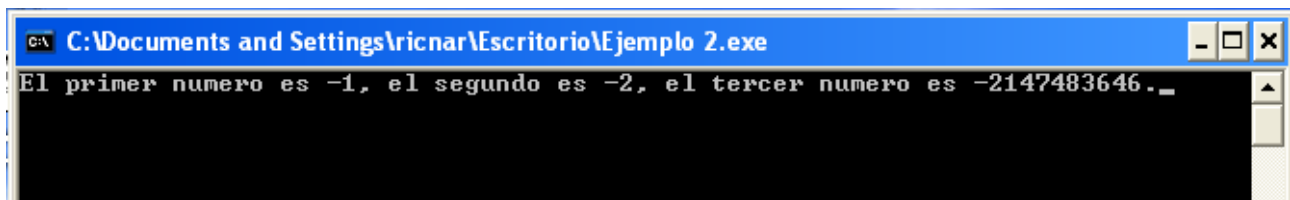
Vemos que de esta forma usando **int** o **signed int** el máximo numero positivo es **7fffffff**, ahora si uno sabe que no va a utilizar números negativos y necesita mas rango puede declarar una variable como **unsigned int** y de esta forma siempre sera positiva desde 0 hasta **fffffff** ampliando el rango y eliminando la posibilidad de los negativos.

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

```
main(){  
    funcion2();  
    getchar();  
}
```

```
funcion2(){  
  
    int primerNumero;  
    signed int segundoNumero;  
    unsigned int tercerNumero;  
  
    primerNumero = -1;  
    segundoNumero = -2;  
    tercerNumero = 2147483650;  
  
    printf("El primer numero es %d, ", primerNumero);  
    printf("el segundo es %d, ", segundoNumero);  
    printf("el tercer numero es %d.", tercerNumero);  
}
```

Si compilamos y ejecutamos esto vemos que el valor unsigned es mostrado como signed, pero eso no es porque este mal declarado ya lo veremos sino porque el format string con %d solo convierte **signed int**, para imprimir **unsigned int** debemos usar %u, pero compilemos y miremos un poco primero este ejemplo.



Vemos que la tercera variable la imprimió tal cual fuera signed, ya que al pasarse del máximo valor positivo que habíamos visto que era 2147483647, por lo tanto 2147483650 corresponde al hexa 80000002 o sea a uno de los negativos mas altos pues el máximo negativo es 80000000h.

Veamos nuestro engendro en IDA

```
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], 0FFFFFFFFh
mov     [ebp+var_8], 0FFFFFFFh
mov     [ebp+var_C], 80000002h
mov     eax, [ebp+var_4]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElPrimerNumero ; "El primer numero es %d,
call    printf
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElSegundoEsD ; "el segundo es %d, "
call    printf
mov     eax, [ebp+var_C]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElTercerNumero ; "el tercer numero es %d."
call    printf
leave
retn
sub_4012C6 endp
```

Allí vemos nuestras tres variables que ocupan un dword cada uno al ser las tres declaradas como int, sea signed o unsigned.

```
mov     [ebp+var_C], 80000002h
```

Allí vemos que la variable unsigned le asigna el valor 80000002 como hemos visto.

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

```
main(){
    funcion2();
    getchar();
}
```

```
funcion2(){

    signed int primerNumero;
```

```

signed int segundoNumero;
unsigned int tercerNumero;
unsigned int suma;

primerNumero = -1;
segundoNumero = -2;
tercerNumero = 2147483650;
suma= tercerNumero + 2;

printf("El primer numero es %d, ", primerNumero);
printf("el segundo es %d, ", segundoNumero);
printf("el tercer numero es %u.", suma);
}

```

Ahora vamos a ver como maneja la suma agregamos una variable suma que sera unsigned y al valor que había colocado en tercer numero le sumo 2, si fuera negativo, el resultado seria diferente, veamoslo en IDA, cambiemos también el %d por %u para que imprima los unsigned, y al compilar y ejecutarlo vemos que la suma fue realizada correctamente y se muestra correctamente ahora.

Vemos que considero el numero como positivo y le sumo 2 y el resultado es correcto si hubiera sumado dos a un numero negativo el resultado seria obviamente diferente.

```

main(){
    funcion2();
    getchar();
}

funcion2(){

    char primerNumero;
    short segundoNumero;
    int tercerNumero;

    primerNumero = -1;
    segundoNumero = -2;
    tercerNumero = 500;

    printf("El primer numero es %d \n", primerNumero);
    printf("el segundo es %d \n", segundoNumero);
    printf("el tercer numero es %d \n", tercerNumero);
}

```

Si compilamos esto veremos en el IDA que **char** es el equivalente a **byte**, que **short** es el equivalente a **word** y que **int** como ya vimos es el equivalente a **dword**.

```
; Attributes: bp-based frame

sub_4012C6 proc near

var_8= dword ptr -8
var_4= word ptr -4
var_1= byte ptr -1

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+var_1], 0FFh
mov     [ebp+var_4], 0FFFEh
mov     [ebp+var_8], 1F4h
```

Allí vemos como se acomodaron en el stack las tres variables declaradas, arriba del **return address** y el **stored ebp** y como a cada una el compilador le reserva el espacio justo según el tipo, para la variable que es un **int** sera reservado un **dword** o **dd** como muestra el IDA hay 4 bytes de espacio reservado (de 8 a 4), luego el **short** ocupara un **word** o **dw** en IDA (de 4 a 2) luego habrá un **byte** sin definir (de 2 a 1) y el **char** ocupara un **byte** (de 1 a 0).

```
-0000000D          db ? ; undefined
-0000000C          db ? ; undefined
-0000000B          db ? ; undefined
-0000000A          db ? ; undefined
-00000009          db ? ; undefined
-00000008  var_8    dd ?
-00000004  var_4    dw ?
-00000002          db ? ; undefined
-00000001  var_1    db ?
+00000000          s          db 4 dup(?)
+00000004          r          db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

También tenemos la posibilidad de manejar floats, para poder imprimirlos se utiliza **%f**.

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

```
main(){
    funcion2();
    getchar();
}
```

```
funcion2(){
```

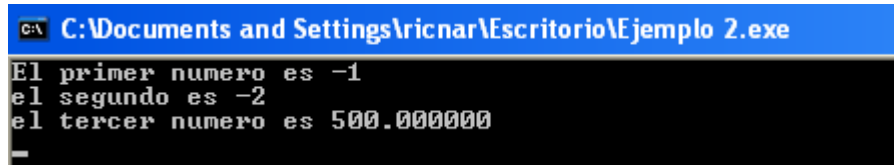
```
    char primerNumero;
    short segundoNumero;
    float tercerNumero;
```

```
    primerNumero = -1;
    segundoNumero = -2;
```

```
tercerNumero = 500;
```

```
printf("El primer numero es %d \n", primerNumero);  
printf("el segundo es %d \n", segundoNumero);  
printf("el tercer numero es %f \n", tercerNumero);
```

```
}
```



```
C:\Documents and Settings\ricnar\Escritorio\Ejemplo 2.exe  
El primer numero es -1  
el segundo es -2  
el tercer numero es 500.000000  
_
```

Para manejar floats utilizara las instrucciones de punto flotante que no explicaremos aquí, el que quiera ver puede consultar en este link, no es nada del otro mundo FLD carga la variable float en el stack de punto flotante que no es el mismo que el del programa, y luego FSTP guarda en [esp+4] en el stack del programa ese valor para pasárselo como argumento al printf.

<http://homepage.mac.com/eravila/asmix86a.html>

```
mov     [ebp+var_8], eax  
movsx   eax, [ebp+var_1]  
mov     [esp+4], eax  
mov     dword ptr [esp], offset aElPrimerNumero ; "El primer numero es %d \n"  
call    printf  
movsx   eax, [ebp+var_4]  
mov     [esp+4], eax  
mov     dword ptr [esp], offset aElSegundoEsD ; "el segundo es %d \n"  
call    printf  
fld     [ebp+var_8]  
fstp    qword ptr [esp+4]  
mov     dword ptr [esp], offset aElTercerNumero ; "el tercer numero es %f \n"  
call    printf
```

Ahora tenemos el siguiente código que usa la funcion **sizeof()** para ver el tamaño que ocupa cada variable en la memoria.

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

```
main(){  
    funcion2();  
    getchar();  
}
```

```
funcion2(){
```

```
    char primerNumero;  
    short segundoNumero;  
    int tercerNumero;  
    float cuartoNumero;
```

```
    primerNumero = -1;  
    segundoNumero = -2;
```

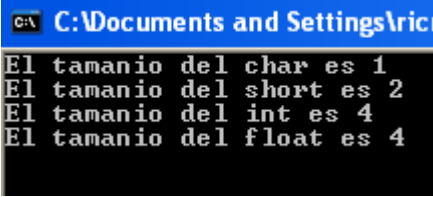
```

tercerNumero = 500;
cuartoNumero = 200;

printf("El tamaño del char es %d \n", sizeof(primerNumero));
printf("El tamaño del short es %d \n", sizeof(segundoNumero));
printf("El tamaño del int es %d \n", sizeof(tercerNumero));
printf("El tamaño del float es %d \n", sizeof(cuartoNumero));
}

```

Al ejecutarlo



```

C:\Documents and Settings\ric...
El tamaño del char es 1
El tamaño del short es 2
El tamaño del int es 4
El tamaño del float es 4

```

Veamoslo en IDA.



```

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+var_1], 0FFh
mov     [ebp+var_4], 0FFFEh
mov     [ebp+var_8], 1F4h
mov     eax, 43480000h
mov     [ebp+var_C], eax
mov     dword ptr [esp+4], 1
mov     dword ptr [esp], offset aElTamanoDelCh ; "El tamaño del char es %d \n"
call    printf
mov     dword ptr [esp+4], 2
mov     dword ptr [esp], offset aElTamanoDelSh ; "El tamaño del short es %d \n"
call    printf
mov     dword ptr [esp+4], 4
mov     dword ptr [esp], offset aElTamanoDelIn ; "El tamaño del int es %d \n"
call    printf
mov     dword ptr [esp+4], 4
mov     dword ptr [esp], offset aElTamanoDelFl ; "El tamaño del float es %d \n"
call    printf

```

Vemos que el tamaño de las variables no se resuelve en tiempo de ejecución del programa, sino que lo resuelve el compilador y ya compila con el tamaño correcto en cada caso reemplazándolo por la constante correspondiente.

El tipo de datos CHAR

Vimos que el tipo de datos char ocupa un byte, pero además de poder almacenar un número de un byte cualquiera como vimos, se usa principalmente para almacenar un carácter.

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

```

main(){
    funcion2();
    getchar();
}

```

```

funcion2(){

    char primerNumero;

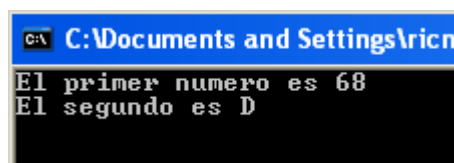
    primerNumero = 68;

    printf("El primer numero es %d \n", primerNumero);
    printf("El segundo es %c \n", primerNumero);

}

```

Vemos que guardamos el numero 68 decimal que en la tabla ASCII corresponde a la D, si lo imprimimos como numero entero mostrara el 68, pero si usamos el format string con **%c** lo imprimirá como la letra D al convertirlo a carácter.



Si lo vemos en IDA vemos que es la misma variable la que le pasa al printf en ambos casos, solo que en un caso al ser **%d** lo muestra como entero y en el otro al ser **%c** lo muestra como carácter según la tabla ASCII.

```

sub_4012C6 proc near
var_1= byte ptr -1

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+var_1], 44h
movsx   eax, [ebp+var_1]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElPrimerNumero ; "El primer numero es %d \n"
call    printf
movsx   eax, [ebp+var_1]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElSegundoEsc ; "El segundo es %c \n"
call    printf
leave
retn
sub_4012C6 endp

```

Vemos que estamos mirando ejemplos sencillos para ver como se manejan las variables e ir familiarizándonos como se ven en IDA e ir incrementando de a poco la dificultad.

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

```
main(){  
    funcion2();  
    getchar();  
}
```

```
funcion2(){
```

```
    char letra1, letra2;
```

```
    printf("Teclea una letra ");
```

```
    scanf("%c", &letra1);
```

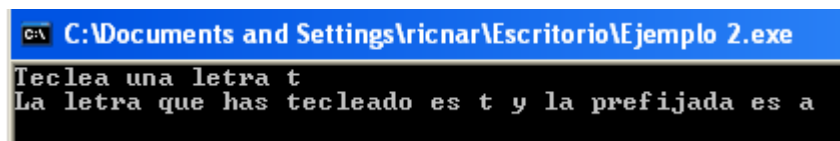
```
    letra2 = 'a';
```

```
    printf("La letra que has tecleado es %c y la prefijada es %c",  
        letra1, letra2);
```

```
    getchar();  
}
```

Vemos dos variables definidas como **char**, la variable llamada **letra1** tomara lo que teclea el usuario usando la funcion **scanf**, a la cual se le pasa como argumentos **%c** para que convierta lo tipeado en carácter, y el segundo argumento como vimos es la dirección de memoria de la variable **letra1** con el **&** delante, por supuesto lo veremos como LEA en el IDA, como ya explicamos.

Luego se inicializa la variable **letra2** con la “a”, y se imprimen ambas, la a y el carácter tipeado por el usuario.



```
C:\Documents and Settings\ricnar\Escritorio\Ejemplo 2.exe  
Teclea una letra t  
La letra que has tecleado es t y la prefijada es a
```

Lo reversearemos como si no conociéramos el código y renombraremos las variables y la funcion a nuestro gusto con los nombres que queramos según el uso de cada una.


```

ew-A | x Stack of sub_4012C6 | x Hex View-A | x Structures | x Enums | x Imports | x Exports
; Attributes: bp-based frame

sub_4012C6 proc near

var_2= byte ptr -2
var_1= byte ptr -1

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     dword ptr [esp], offset aTecleaUnaLetra ; "Teclea una letra "
call    printf
lea     eax, [ebp+var_1]
mov     [esp+4], eax
mov     dword ptr [esp], offset aC ; "%c"
call    scanf
mov     [ebp+var_2], 61h
movsx   eax, [ebp+var_2]
mov     [esp+8], eax
movsx   eax, [ebp+var_1]
mov     [esp+4], eax
mov     dword ptr [esp], offset aLaLetraQueHasT ; "La letra que has tec
call    printf
call    getchar
leave
retn
sub_4012C6 endp

```

Como todavía se supone que no se que hace la funcion, le pondré un nombre cualquiera.

```

; Attributes: bp-based frame

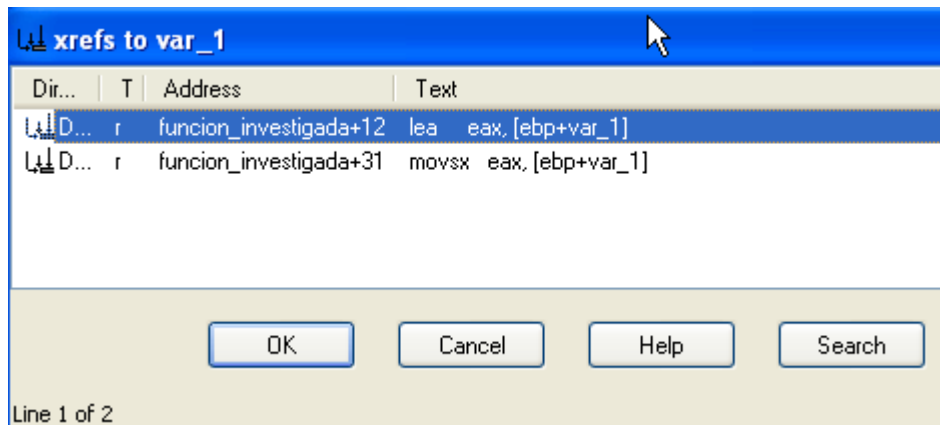
funcion_investigada proc near

var_2= byte ptr -2
var_1= byte ptr -1

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     dword ptr [esp], offset aTec:
call    printf
lea     eax, [ebp+var_1]
mov     [esp+4], eax
mov     dword ptr [esp], offset aC ;
call    scanf
mov     [ebp+var_2], 61h
movsx   eax, [ebp+var_2]
mov     [esp+8], eax
movsx   eax, [ebp+var_1]
mov     [esp+4], eax

```

Por ahora le puse como nombre funcion investigada, ahora comenzare a ver que hace cada variable, para ponerle el nombre correspondiente., al marcar la **var_1** se resaltan todos los lugares que la misma se usa en esta funcion, también si es muy grande la funcion, podría apretar la X y me mostraría en una lista sus referencias.



Obviamente si queremos saber donde se va a inicializar una variable, en el caso que no se inicialice con una asignación directa por ejemplo mediante un **mov** dentro de nuestra funcion, en ese caso debemos ver donde hay un **lea**, ya que vimos que cuando el compilador usa el **lea** obtiene la dirección de memoria en el stack donde esta ubicada dicha variable, y es seguro que a continuación, lo usara como argumento en alguna funcion o call para llenarla o asignarle un valor dentro de el mismo. (como vimos en los ejemplos anteriores las variables locales solo se pueden inicializar en forma directa dentro de nuestra funcion, en cualquier api, o call dentro de nuestra funcion, habrá que pasarle la dirección de memoria de dicha variable lo que se hace mediante el **&** y aquí se vera como un LEA)

Así que aunque la funcion sea larguísima, buscar el **lea** sobre las referencias de la variable que no esta inicializada en forma directa, nos llevara al punto donde se inicializara la misma.

```
lea    eax, [ebp+var_1]      <-----
mov    [esp+4], eax
mov    dword ptr [esp], offset aC ; "%c"
call   scanf
```

Así que aunque no tengamos el código fuente y al ver que la dirección de la variable en el stack se pasa a EAX y de allí se guarda en el stack para usar como argumento de **scanf**, conociendo la api **scanf**, nos damos cuenta que dicha variable guardara un carácter, dado que el otro argumento de **scanf** es **%c**.

Así que ya se para que sirve la variable, para guardar un carácter tipeado por el usuario, así que le pondré un nombre acorde a su uso.

```

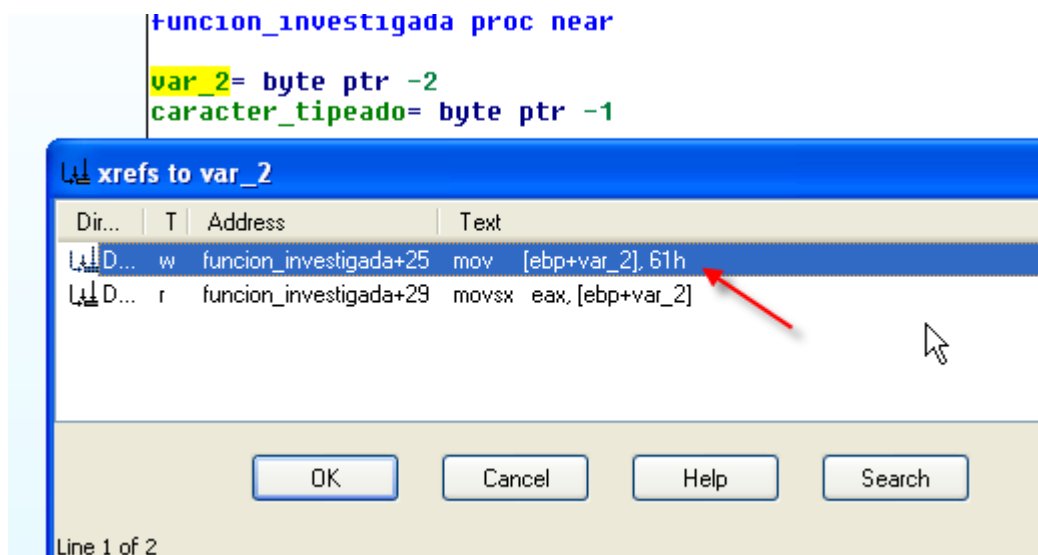
; Attributes: dp-based frame
funcion_investigada proc near
var_2= byte ptr -2
caracter_tipeado= byte ptr -1

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     dword ptr [esp], offset aTecleaUna
call    printf
lea     eax, [ebp+caracter_tipeado]
mov     [esp+4], eax
mov     dword ptr [esp], offset aC ; "%c"
call    scanf
mov     [ebp+var_2], 61h
movsx   eax, [ebp+var_2]
mov     [esp+8], eax
movsx   eax, [ebp+caracter_tipeado]
mov     [esp+4], eax

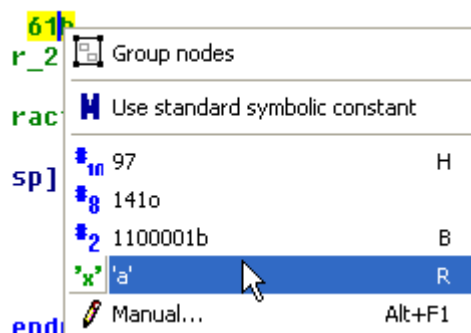
```

Y tomara su valor dentro de **scanf**.

Vemos que hasta ahora no ejecute el código ni debugué nada y voy sacando conclusiones sobre la funcion.



Al ver las referencias de la otra variable vemos que no necesita un lea pues se asigna en forma directa dentro de nuestra funcion con un mov, se le asigna la constante 61 hexa que sabemos que como la variable es char lo interpretara como la “a” por la tabla ASCII, así que cambiemos eso.



Si hago click derecho en el 61 hexa, veo que IDA me da la posibilidad de cambiar la representación según el tipo de datos, así que como se que es un char lo cambio a la letra “a” que se ve en el menú desplegable. Y cambio el nombre de la variable a letra_a.

```

letra_a= byte ptr -2
caracter_tipeado= byte ptr -1

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     dword ptr [esp], offset
call    printf
lea     eax, [ebp+caracter_tipea
mov     [esp+4], eax
mov     dword ptr [esp], offset
call    scanf
mov     [ebp+letra_a], 'a'
movsx   eax, [ebp+letra_a]

```

```

movsx   eax, [ebp+letra_a]
mov     [esp+8], eax
movsx   eax, [ebp+caracter_tipeado]
mov     [esp+4], eax
mov     dword ptr [esp], offset aLaLetraQueHasT ; "La letra que has tecleado es %c y la pr"...
call    printf

```

La ultima parte de la funcion manda como argumentos al stack a las dos variables, **letra_a** y **carácter_tipeado**, y ademas también la string donde se realizara el format string usando **%c** en ambos casos, con lo cual sabemos que imprimirá la string.

La letra que has tecleado es %c y la prefijada es %c

pero reemplazara los dos **%c** por las dos variables una tendra la letra a y la otra letra tipeada por el usuario que si por ejemplo tipeara una **d** quedaría.

La letra que has tecleado es d y la prefijada es a

Con lo cual ya sabemos que hace la funcion, es una especie de juego para ver si en un solo tiro acertás la letra prefijada, así que ahora que ya sabemos que hace la funcion le ponemos otro nombre.

```

juego_bastante_tonto proc near
    letra_a= byte ptr -2
    caracter_tipeado= byte ptr -1

    push    ebp
    mov     ebp, esp
    sub     esp, 18h
    mov     dword ptr [esp], offset aTecleaUna
    call    printf
    lea     eax, [ebp+caracter_tipeado]
    mov     [esp+4], eax
    mov     dword ptr [esp], offset aC ; "%c"
    call    scanf
    mov     [ebp+letra_a], 'a'
    movsx   eax, [ebp+letra_a]
    mov     [esp+8], eax
    movsx   eax, [ebp+caracter_tipeado]
    mov     [esp+4], eax
    mov     dword ptr [esp], offset aLaLetraQue
    call    printf
    call    getchar
    leave
    retn
juego_bastante_tonto endp

```

El código fuente que arma el HexRays de nuestra función es el siguiente

```

int __cdecl juego_bastante_tonto()
{
    char caracter_tipeado; // [sp+17h] [bp-1h]@1

    printf("Teclea una letra ");
    scanf("%c", &caracter_tipeado);
    printf("La letra que has tecleado es %c y la prefijada es %c", caracter_tipeado, 97);
    return getchar();
}

```

Vemos que como siempre simplifica al máximo y usa una sola variable char en la que guarda el tipeo del usuario, la otra directamente la suprime y usa una constante 97 decimal, que es 61 en hexa o sea la letra a, pues como vimos la variable char contiene el valor hexa que luego se transforma en el printf al valor de la tabla ascII por el %c.

Con lo cual terminamos este sencillo caso de reversing jeje hasta la parte 3 (si hay je)

Saludos Crackslatinos.
Ricardo Narvaja