

Uniones y campos de bits

Nos quedan ver algunos detalles del lenguaje C que aunque no tan usados es buenos conocer y ver como los veremos en IDA después de compilados, para el reversing.

Le cedo la palabra a Cabanes sobre el tema **Uniones**, copio la explicación.

Conocemos lo que es un struct: un dato formado por varios “trozos” de información de distinto tipo. Pero C también tiene dos tipos especiales de “struct”, de manejo más avanzado. Son las **uniones** y los **campos de bits**.

Una **unión** recuerda a un “**struct**” normal, con la diferencia de que sus “**campos**” comparten el mismo espacio de memoria:

```
union {  
    char letra; /* 1 byte */  
    int numero; /* 4 bytes */  
} ejemplo;
```

En este caso, la variable “ejemplo” ocupa 4 bytes en memoria (suponiendo que estemos trabajando en un compilador de 32 bits, como lo son la mayoría de los de Windows y Linux). El primer byte está compartido por “letra” y por “numero”, y los tres últimos bytes sólo pertenecen a “numero”.

Si hacemos

```
ejemplo.numero = 25;  
ejemplo.letra = 50;  
printf("%d", ejemplo.numero);
```

Veremos que “ejemplo.numero” ya no vale 25, puesto que al modificar “ejemplo.letra” estamos cambiando su primer byte. Ahora “ejemplo.numero” valdría 50 o un número mucho más grande, según si el ordenador que estamos utilizando almacena en primer lugar el byte más significativo o el menos significativo.

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

```
int main() {
```

```
    union {  
        char letra; /* 1 byte */  
        int numero; /* 4 bytes */  
    } ejemplo;
```

```
    int n1, n2;
```

```
    ejemplo.numero = 25;  
    ejemplo.letra = 50;  
    printf("%d", ejemplo.numero);
```

```
    getchar();
```

```

getchar();
return 0;
}

```

```

#include <stdio.h>

int main() {

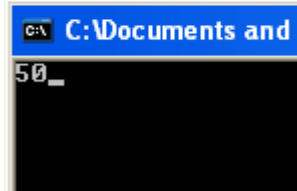
    union {
        char letra; /* 1 byte */
        int numero; /* 4 bytes */
    } ejemplo;

    int n1, n2;

    ejemplo.numero = 25;
    ejemplo.letra = 50;
    printf("%d", ejemplo.numero);

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

```



Vemos que el campo **ejemplo.numero** comparte memoria con **ejemplo.letra** y al cambiar el valor de este ultimo afectamos el valor del primero veamoslo en IDA a ver como se ve y si hay posibilidad de reversear esto.

```

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, 0FFFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_10], eax
mov     eax, [ebp+var_10]
call    __chkstk
call    __main
mov     [ebp+var_4], 19h
mov     byte ptr [ebp+var_4], 32h
mov     eax, [ebp+var_4]
mov     [ebp+var_4], eax

```

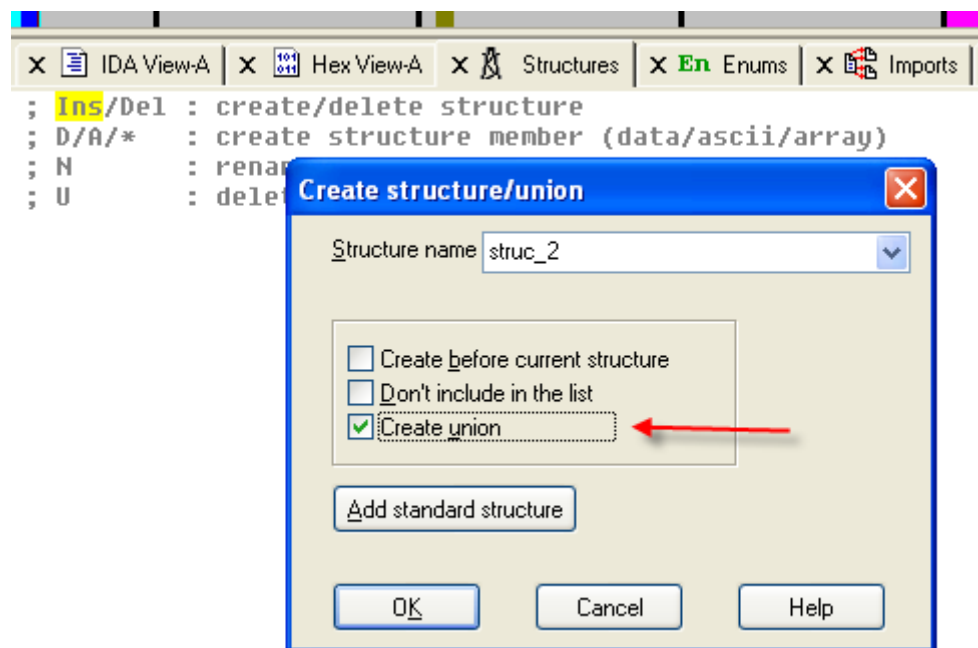
Vemos que IDA no ha hecho demasiado por nuestra union la ha puesto como una sola variable int llamada var_4 y listo, un reverser le costara en este ejemplo darse cuenta que hay una union allí, pero hay algunas pistas como por ejemplo que cuando inicializa vemos que escribe un dword 19, siendo que es lógico ya que es un int, pero en la siguiente linea le escribe un byte solo a la misma

variable **var_4**

```
mov [ebp+var_4], 19h
mov byte ptr [ebp+var_4], 32h
```


No tiene mucho sentido si es solo una variable int inicializar su valor con un dword 19h y luego escribir un solo byte en la misma en este caso el 32h, ahí sospechamos de que hay una union entre dos variables una int y una char.

En IDA para definir la unión vamos a la pestaña estructuras y agregamos una como siempre.



Solo que esta vez marcamos la tilde Create union y agregamos dos campos uno dword y uno de un solo byte, el tamaño de la unión siempre sera el del campo mas grande en este caso 4 pues hay un dword.

```
00000000
00000000 union_2          union ; (sizeof=0x4)
00000000 field_0         dd ?
00000000 field_1         db ?
00000000 union_2         ends
00000000
```



Los renombramos como deseamos

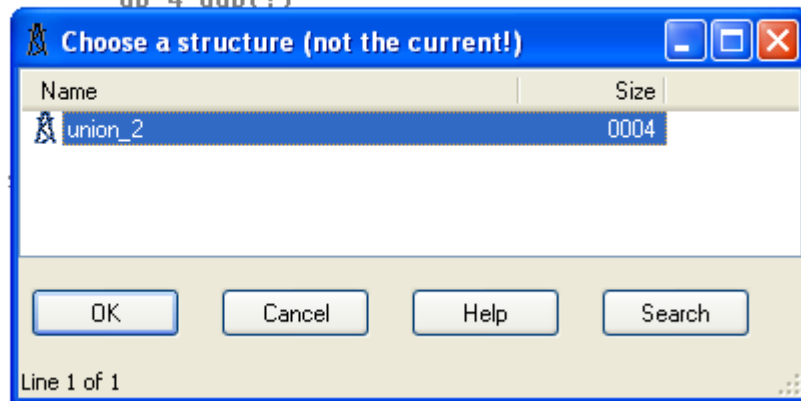
```
00000000
00000000 union_2          union ; (sizeof=0x4)
00000000 numero         dd ?
00000000 letra         db ?
00000000 union_2         ends
00000000
```

Yendo a la ventana de variables apretando ALT mas Q encima de **var_4**, elegimos que sea del tipo de la unión que hemos creado.

```

-00000005          db ? ; undefined
-00000004 var_4      dd ?
+00000000          db 4 dup(?)
+00000004          s
+00000008          r
+0000000C          argc
+00000010          argv
+00000014          envp
+00000014 ; end of

```



Cambiamos el nombre a la variable también.

```

-00000005          db ? ; undefined
-00000004 ejemplo    union_2 ?
+00000000          s      db 4 dup(?)
+00000004          r      db 4 dup(?)
+00000008          argc   dd ?
+0000000C          argv   dd ? ; offset
+00000010          envp   dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

Igual a pesar de todo el trabajo IDA no logra marcar los campos de las uniones como diferentes y sigue tomando como una sola variable aunque del tipo unión.

```

mov     [ebp+var_10], eax
mov     eax, [ebp+var_10]
call    __chkstk
call    __main
mov     dword ptr [ebp+ejemplo], 19h
mov     byte ptr [ebp+ejemplo], 32h
mov     eax, dword ptr [ebp+ejemplo]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    printf
call    getchar
call    getchar
mov     eax, 0
leave
retn
main endp

```

Aunque si pasamos el mouse por dicha variable veremos que nos muestra como son los campos de dicha unión.

```

call    ___chkstk
call    __main
mov     dword ptr [ebp+ejemplo], 19h
mov     byte ptr [ebp+ejemplo], 32h
mov     eax, dword ptr [ebp+ejemplo]
mov     [esp+4], eax
mov     dword ptr [esp], offset
call    printf
call    getchar
call    getchar
mov     eax, 0
leave
retn
main endb

```

0) 000006C5 004012C5: _main+35

```

...
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
ejemplo union_2 ?
s db 4 dup(?)
r db 4 dup(?)
argc dd ?
...

```

Enumeraciones

Una enumeración es un conjunto de constantes enteras. A la enumeración se le puede asignar un nombre, que se comportará como un nuevo tipo de dato que solo podrá contener los valores especificados en la enumeración.

```
enum dia { DOM, LUN, MART, MIER, JUEV, VIER, SAB } diaX;
```

```
int main() {
```

```
diaX=LUN;
printf ("%d", diaX);
```

```
getchar();
}
```

Allí hay un típico caso de enumeración donde se relacionan por orden los días de la semana con un numero, si no especificamos nada, sera DOM=0, LUN=1 etc, si corremos este programa vemos que su salida es 1, ya que LUN esta relacionado con el numero 1, para el tipo de variable **dia**.

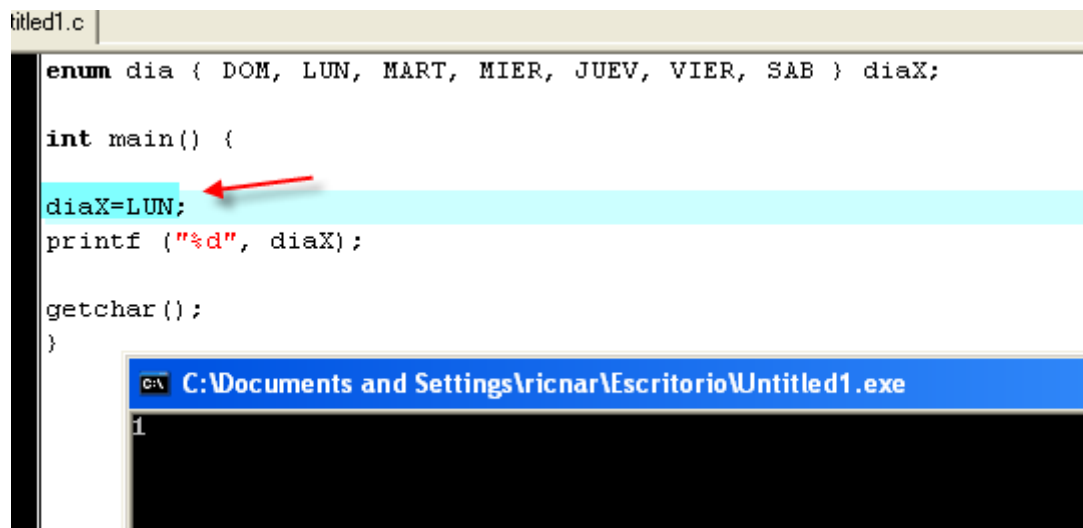
```

titled1.c |
enum dia { DOM, LUN, MART, MIER, JUEV, VIER, SAB } diaX;

int main() {
    diaX=LUN;
    printf ("%d", diaX);

    getchar();
}

```



Si queremos agregar mas variables del tipo día podemos hacerlo, todas respetaran la relación que definimos al inicio entre los días y un numero entero.

```

enum dia { DOM, LUN, MART, MIER, JUEV, VIER, SAB } diaX;

int main() {

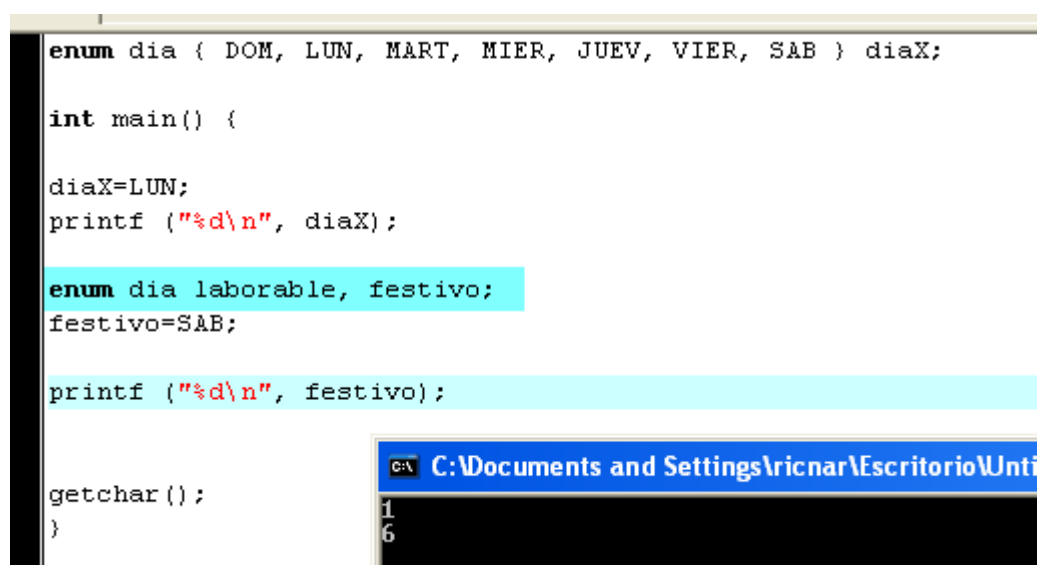
    diaX=LUN;
    printf ("%d\n", diaX);

    enum dia laborable, festivo;
    festivo=SAB;

    printf ("%d\n", festivo);

    getchar();
}

```



También se podría haber hecho.

```
led1.c
enum dia { DOM, LUN, MART, MIER=9, JUEV, VIER, SAB } diaX;


int main() {

diaX=LUN;
printf ("%d\n", diaX);

enum dia laborable, festivo;
festivo=MIER;


printf ("%d\n", festivo);

getchar();
}
```



Si lo vemos en IDA no veremos mucho mas que estoç

```
mov     eax, [ebp+var_C]
call    ___chkstk
call    main
mov     ds:dword_404060, 1
mov     eax, ds:dword_404060
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
mov     [ebp+var_8], 9
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
call    getchar
leave
retn
```



La variable global **diaX** ya que fue definida antes del main, esta en **404060**, podemos renombrarla.

```

call    __main
mov     ds:diaX, 1
mov     eax, ds:diaX
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
mov     [ebp+var_8], 9
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
call    getchar
leave
retn
main endo

```

En cambio la **var_8** que es también del tipo día pero definida localmente, la renombraremos a **festivo**.

```

call    __main
mov     ds:diaX, 1
mov     eax, ds:diaX
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
mov     [ebp+festivo], 9
mov     eax, [ebp+festivo]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
call    getchar
;-----

```

Ahora vamos a la pestaña enumeraciones y agregamos una apretando INS tal cual hacemos con las estructuras.

Add enum type

Name:

Width:

☐ Hexadecimal
☒ **Decimal**
☐ Octal
☐ Binary
☐ Character

☐ Signed
☐ Bitfield

If checked, a bitfield is created

Ponemos decimal ya que queremos que se asocie con enteros decimales para que sea mas sencillo. Apretando N vamos agregando los campos DOM sera 0 y así sucesivamente, salvo MIER que sera 9.

Edit enum member

Enum: dia

Name:

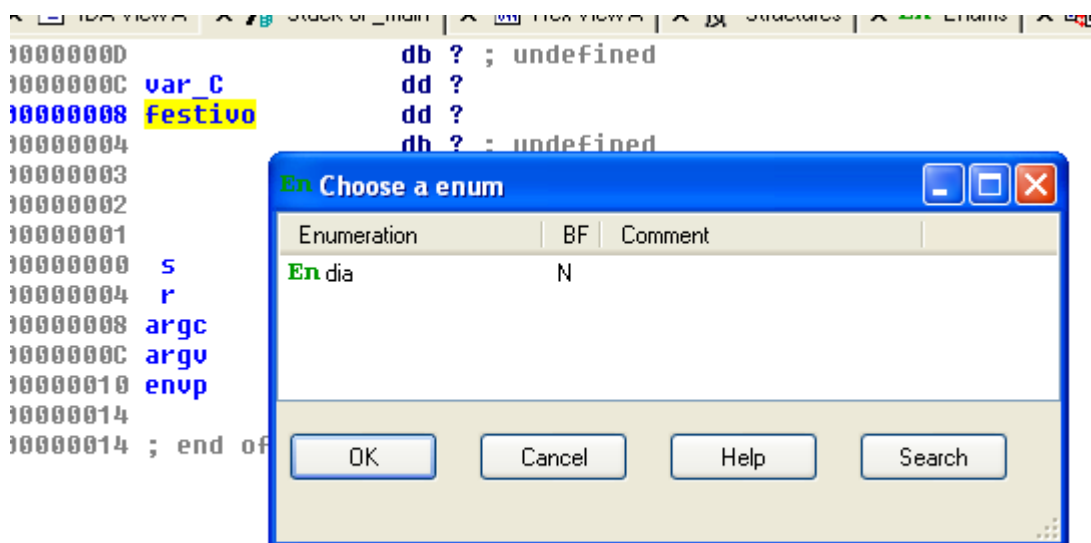
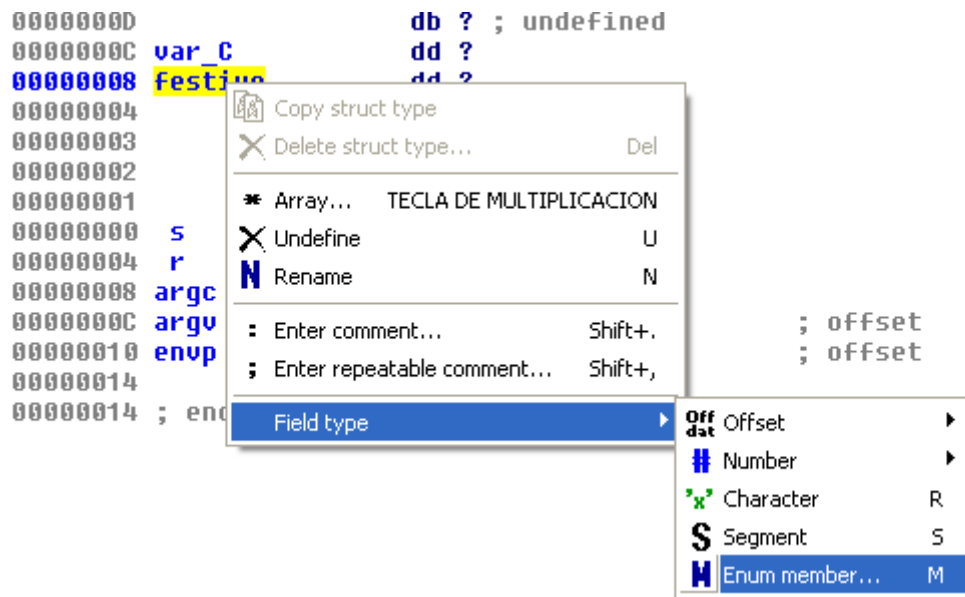
Value:

```

FFFFFFFF ;
FFFFFFFF ; For bitfields the line prefixe
FFFFFFFF ; -----
FFFFFFFF
FFFFFFFF ; enum dia
FFFFFFFF DOM = 0
FFFFFFFF LUN = 1
FFFFFFFF MART = 2
FFFFFFFF JUEV = 4
FFFFFFFF UIERN = 5
FFFFFFFF SAB = 6
FFFFFFFF MIER = 9
FFFFFFFF

```

Ahí esta lista vamos a las variables y elegimos el tipo de enumeración que creamos



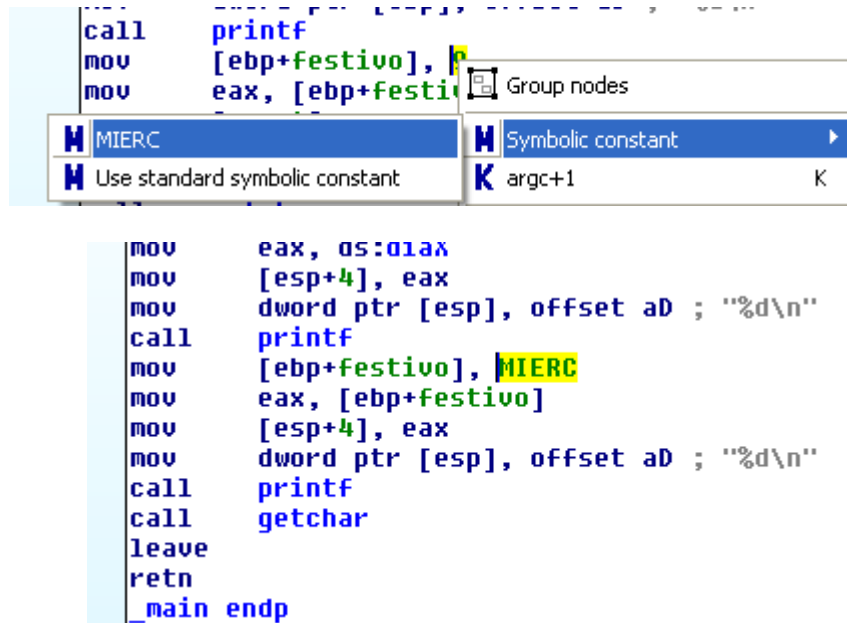
Si vamos al código parece que nada cambio y es así, pero si hacemos click derecho en el 9.

```

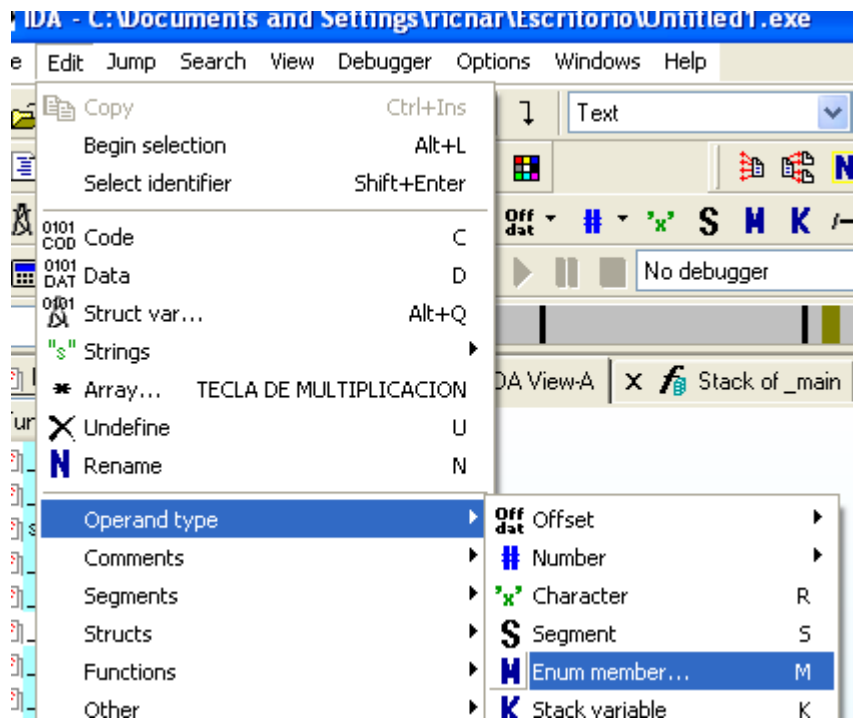
call    printf
mov     [ebp+festivo], 9
mov     eax, [ebp+festivo]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
call    getchar

```

Nos da la opción de poner MIER como definimos en la enum para el 9.



Lo mismo para la variable global `diaX` la marcamos y



Y nos quedan cambiadas las constantes

```
mov     [ebp+var_8], eax
mov     eax, [ebp+var_C]
call    ___chkstk
call    __main
mov     ds:diaX, LUN
mov     eax, ds:diaX
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
mov     [ebp+festivo], MIERC
mov     eax, [ebp+festivo]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d\n"
call    printf
call    getchar
leave
retn
_main endp
```

CAMPOS DE BITS

Un campo de bits es un elemento de un registro (struct), que se define basándose en su tamaño en bits. Se define de forma muy parecida (pero no igual) a un "struct" normal, indicando el número de bits que se debe reservar a cada elemento:

```
struct campo_de_bits {
    int bit_1 : 1;
    int bits_2_a_5 : 4;
    int bit_6 : 1;
    int bits_7_a_16 : 10;
} variableDeBits;
```

Esta variable ocuparía $1+4+1+10 = 16$ bits (2 bytes). Los campos de bits pueden ser interesantes cuando queramos optimizar al máximo el espacio ocupado por nuestros datos.

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

```
int main() {
```

```
    struct campo_de_bits {
        int bit_1 : 1;
        int bits_2_a_5 : 4;
        int bit_6 : 1;
        int bits_7_a_16 : 10;
    } variableDeBits;
```

```

variableDeBits.bit_1 = 0;
variableDeBits.bits_2_a_5 = 3;
variableDeBits.bit_6 = 0;
printf("%d", variableDeBits);

```

```

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

```

Bueno vemos un campo de bits, y como asignamos valores, vemos que al asignar diferentes valores a cada campo, el valor numérico de la **variableDeBits** cambia, obviamente pues estamos cambiando sus bits.

En el ejemplo valdrá 6 pues estamos asignando el decimal 3 que es 11 a la posición 2 a 5 o sea que

0011 va al bit 2 a 5

o sea que **variableDeBits** quedara en binario 000110 que es el 6 decimal.

```

#include <stdio.h>

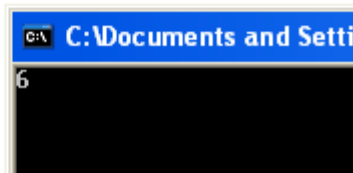
int main() {

    struct campo_de_bits {
        int bit_1 : 1;
        int bits_2_a_5 : 4;
        int bit_6 : 1;
        int bits_7_a_16 : 10;
    } variableDeBits;

    variableDeBits.bit_1 = 0;
    variableDeBits.bits_2_a_5 = 3;
    variableDeBits.bit_6 = 0;
    printf("%d", variableDeBits);

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

```



Si lo vemos en IDA no ganamos mucho para reversear también es trabajado todo sobre una solo variable y realizadas las operaciones usando ANDs y ORs para cambiar los resultados de los bits.

```

lea     eax, [ebp+var_4]
and     byte ptr [eax], 0FEh
mov     eax, [ebp+var_4]
and     eax, 0FFFFFFE1h
or      eax, 6
mov     [ebp+var_4], eax
lea     eax, [ebp+var_4]
and     byte ptr [eax], 0DFh
mov     eax, [ebp+var_4]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"

```

Existe en la pestaña ENUM la opción de crear campos de bits, pero realmente no aporta mucho al reversing así que lo vamos solo a mencionar sin entrar en detalles, pues IDA aunque lo definas no cambia el listado así que para el reverser no aporta mucho solo es bueno para conocer el tema.

Bueno hemos terminado con lo básico para empezar a reversear a partir de la semana que viene comenzaremos reverseando pequeños ejemplos que iremos programando nosotros mismos, asimismo habrá ejercicios de reversing ahora que terminamos la cucharada mas difícil que es ver la base teórica, el resto es remar con dos tenedores como remos jeje.

Hasta la próxima
Ricnar