

C Y REVERSING (parte 10) por Ricnar

MANEJO DE ARCHIVOS O FICHEROS

Poco a poco vamos avanzando en diferentes temas relativos a C y su reverseo, ahora nos toca el tema de manejo de archivos, la lectura y escritura en ellos.

```
#include <stdio.h>
#include <string.h>
main(){
funcion();
getchar();
}
```

```
funcion()
{
```

```
    FILE* fichero;
```

```
    fichero = fopen("prueba.txt", "wt");
    fputs("Esto es una línea\n", fichero);
    fputs("Esto es otra", fichero);
    fputs(" y esto es continuación de la anterior\n", fichero);
    fclose(fichero);
```

```
}
```

Para definir una variable del tipo fichero, debemos usar la palabra FILE seguida del asterisco y a continuación el nombre de dicha variable, el asterisco significa puntero y en este caso apunta a una estructura que controla el objeto archivo.

A continuación para abrir el fichero se usa **fopen**, a la cual se le pasan como argumentos el nombre del archivo y el tipo de acceso, **w** si es para escribir en el, **r** si es para leer, también en este caso vemos que usa la letra t para aclarar que se un archivo de texto.

En el caso de un archivo abierto para escritura de no existir el mismo se creara y si ya existía se borrara su contenido y creara vacío nuevamente. (mas adelante se vera como agregar información sin borrar la existente)

Luego se usa **fputs** para escribir en el archivo, casi es forma similar como hacíamos para escribir en pantalla con **puts**, solo que aquí habrá que especificar los saltos de linea por eso cada string que se envía a **fputs** debe terminar en **\n** si queremos que la siguiente se agregue en la próxima linea, sino lo hará a continuación, el otro argumento enviando a fputs debe ser el nombre de la variable tipo fichero.

Al terminar de escribir en el lo cerraremos con **fclose()**, pasandole también como argumento el nombre de la variable tipo fichero.

Bueno todo muy lindo, si lo corremos vemos que en la misma carpeta donde corre el ejecutable se crea el archivo de texto llamado **prueba.txt**, si lo abrimos vemos su contenido.

Esto es una línea

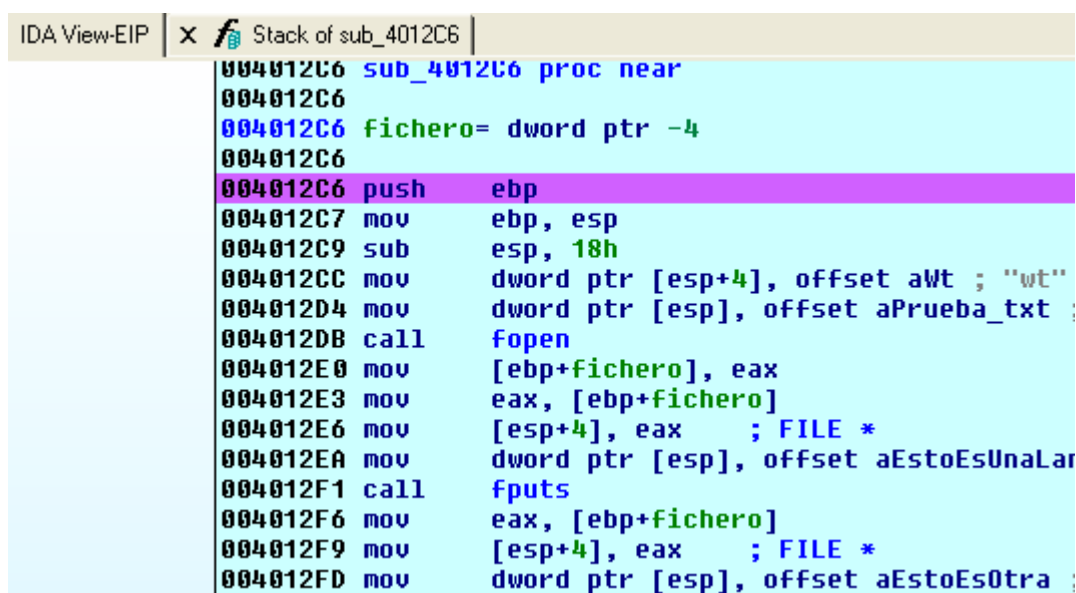
Esto es otra y esto es continuación de la anterior

Veamos el código en IDA, renombrando convenientemente.

```
sub_4012C6 proc near
fichero= dword ptr -4

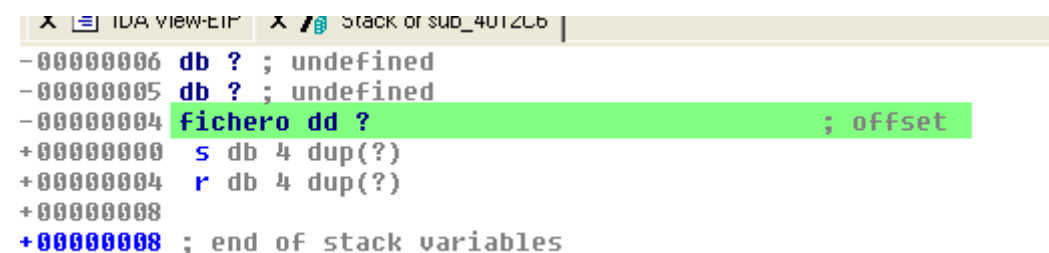
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     dword ptr [esp+4], offset aWt ; "wt"
mov     dword ptr [esp], offset aPrueba_txt ; "prueba.txt"
call    fopen
mov     [ebp+fichero], eax
mov     eax, [ebp+fichero]
mov     [esp+4], eax ; FILE *
mov     dword ptr [esp], offset aEstoEsUnaLanea ; "Esto es una línea\n"
call    fputs
mov     eax, [ebp+fichero]
mov     [esp+4], eax ; FILE *
mov     dword ptr [esp], offset aEstoEsOtra ; "Esto es otra"
call    fputs
mov     eax, [ebp+fichero]
```

Pongamos un breakpoint al inicio de nuestra funcion para mirar un poco como funciona.



```
IDA View-EIP | x Stack of sub_4012C6
004012C6 sub_4012C6 proc near
004012C6
004012C6 fichero= dword ptr -4
004012C6
004012C6 push    ebp
004012C7 mov     ebp, esp
004012C9 sub     esp, 18h
004012CC mov     dword ptr [esp+4], offset aWt ; "wt"
004012D4 mov     dword ptr [esp], offset aPrueba_txt ; "prueba.txt"
004012DB call    fopen
004012E0 mov     [ebp+fichero], eax
004012E3 mov     eax, [ebp+fichero]
004012E6 mov     [esp+4], eax ; FILE *
004012EA mov     dword ptr [esp], offset aEstoEsUnaLar
004012F1 call    fputs
004012F6 mov     eax, [ebp+fichero]
004012F9 mov     [esp+4], eax ; FILE *
004012FD mov     dword ptr [esp], offset aEstoEsOtra ; "Esto es otra"
```

Si vemos las variables, la variable fichero es un dword ya que es un puntero y luego ya se encuentra el **stored ebp** y **return address** no hay mas nada.



```
IDA View-EIP | x Stack of sub_4012C6
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 fichero dd ? ; offset
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

Ahora traceemos un poco.

```

004012C6
004012C6 push    ebp
004012C7 mov     ebp, esp
004012C9 sub     esp, 18h
004012CC mov     dword ptr [esp+4], offset aPrueba_txt
004012D4 mov     dword ptr [esp], offset aPrueba_wt
004012DB call    fopen
004012E0 mov     [ebp+fichero], eax
004012E3 mov     eax, [ebp+fichero]

```

Cuando llegamos a **fopen** si miramos en el stack cambiándolo con JMP TO ESP para que se actualice, vemos los argumentos que se le pasan a la misma.

Un puntero a la string **Prueba.txt** y el otro un puntero a la string **wt** como en nuestro código fuente.

```
fichero = fopen("prueba.txt", "wt");
```

Vemos que el resultado de la llamada a fopen es guardado en la variable **fichero**, lo mismo vemos en el IDA al volver de fopen guarda EAX en fichero.

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

Si vemos la definición de fopen

<http://c.conclase.net/librerias/?ansifun=fopen>

Vemos que dice que retorna un puntero

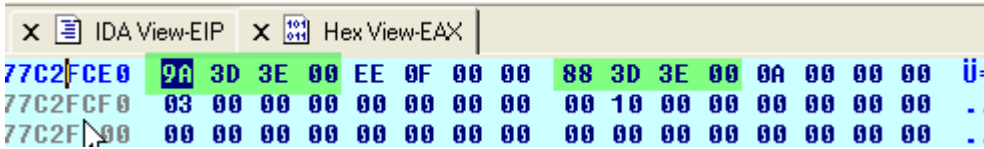
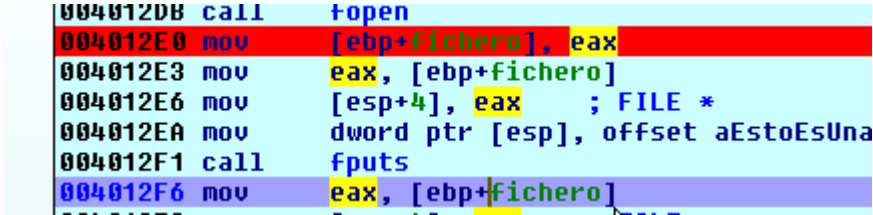
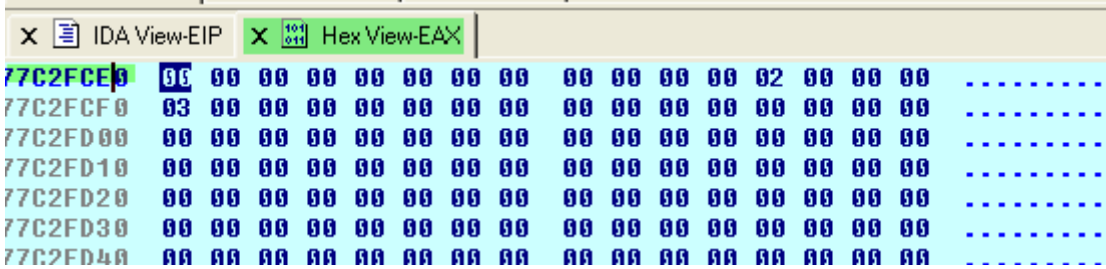
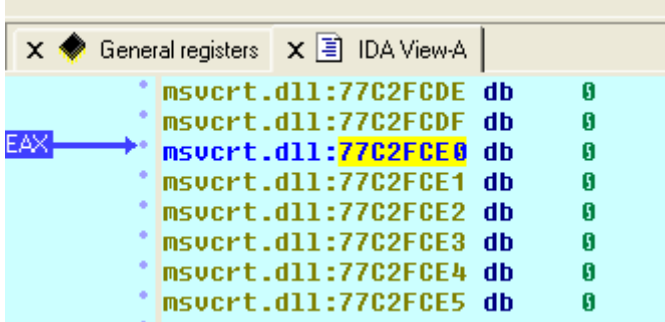
Valor de retorno:

La función fopen retorna un puntero al objeto controlando el stream. Si el proceso de apertura no es realizado a cabo, entonces retorna un puntero nulo.

Bueno vamos viendo ya sabemos que este valor es un puntero, allí vemos en EAX su valor.

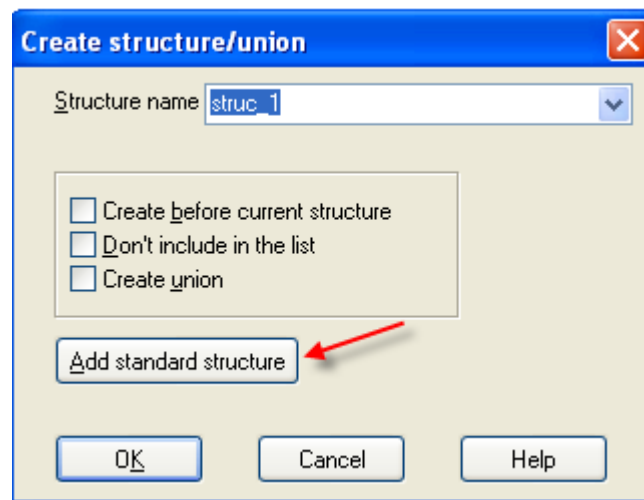
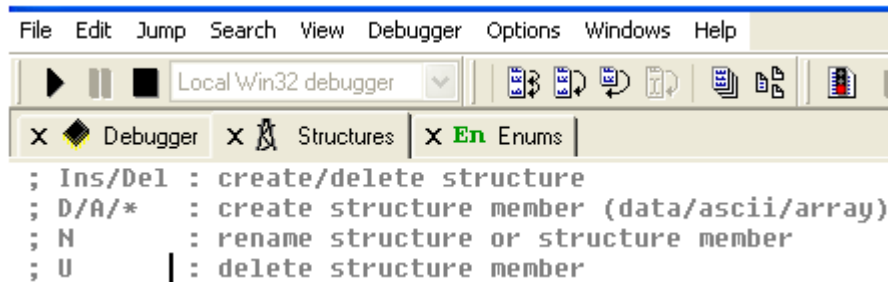
Podemos verlo tanto en una ventana de desensamblado, haciendo click derecho en EAX y eligiendo JUMP IN A NEW WINDOW, como en una nueva ventana HEX, que se abre con VIEW-OPEN

The screenshot shows the Windows Task Manager application. The 'Jump' menu is open, displaying the following options: 'Jump', 'Jump in a new window', and 'Open register window'. A mouse cursor is hovering over the 'Jump in a new window' option.

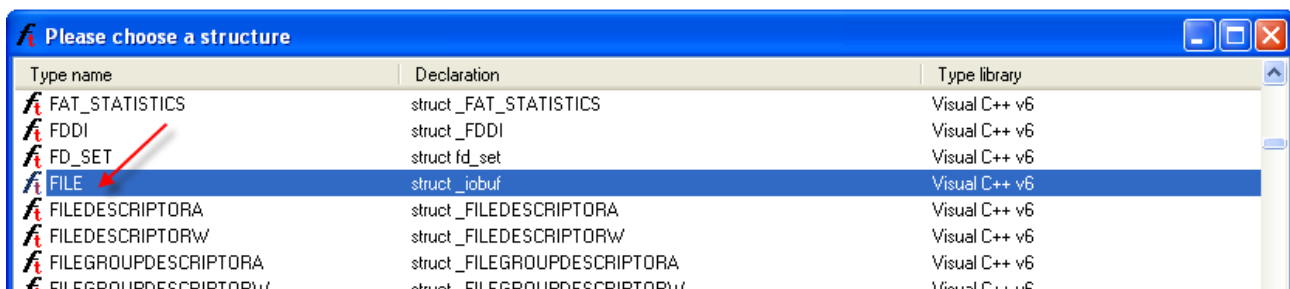


El tema es que FILE * es un puntero a una estructura que controla el archivo, para el reversing no es necesario estudiar como es la estructura pues solo se pasa el puntero a la misma y el sistema hace todo el resto y el programa siempre maneja el puntero pasándoselo como argumento a las apis, pero veremos por curiosidad un par de cositas sobre ella.

Cuando vamos a la ventana de estructuras y apretamos INS o INSERT.



De la lista desplegable elegimos la estructura FILE.



Quedara agregado

```

X  Debugger  X  Structures  X  En Enums
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*  : create structure member (data/ascii/array)
00000000 ; N      : rename structure or structure member
00000000 ; U      : delete structure member
00000000 ; [00000020 BYTES. COLLAPSED STRUCT FILE. PRESS KEYPAD "+" TO EXPAND]

```

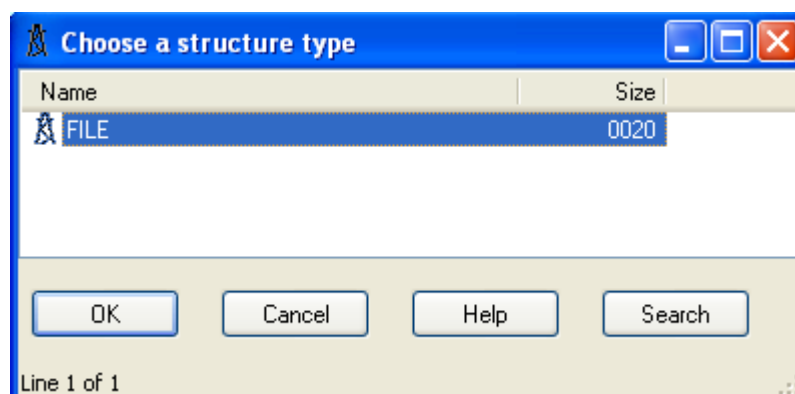
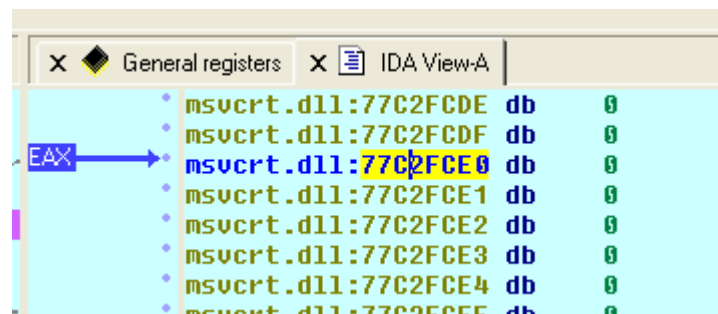
Si lo expandimos apretando +

```

00000000 , n      : rename structure or structure member
00000000 ; U      : delete structure member
00000000 ; -----
00000000
00000000 FILE struc ; (sizeof=0x20, standard type)
00000000 _ptr dd ?      ; offset
00000004 _cnt dd ?
00000008 _base dd ? ; offset
0000000C _flag dd ?
00000010 _file dd ?
00000014 _charbuf dd ?
00000018 _bufsiz dd ?
0000001C _tmpfname dd ? ; offset
00000020 FILE ends
00000020

```

Allí vemos la estructura que ocupa 0x20 bytes, así que vamos al listado donde esta el inicio de la estructura (EAX apuntaba a ella al volver de fopen) y apretamos ALT mas Q.



```

:77C2FCE0 FILE <offset unk_3E3E12, 0FEEh, offset unk_3E3E00, 0Ah, 3, 0, 1000h, 0
:77C2FD00 db 0
:77C2FD01 db 0
:77C2FD02 db 0
:77C2FD03 db 0

```

Su buscamos en el archivo **stdio.h** que estará en nuestra maquina, ahí esta definida la estructura y coincide aunque no aclara mucho que es cada campo.

```

stdio.h | Ejemplo 1b.c | Ejemplo 1a1.c | stdio.h
/* Some believe that nobody in their r:
 * internals of this structure. Provide
 * <paag@tid.es>.
 */
#ifndef _FILE_DEFINED
#define _FILE_DEFINED
typedef struct _iobuf
{
    char* _ptr;
    int _cnt;
    char* _base;
    int _flag;
    int _file;
    int _charbuf;
    int _bufsiz;
    char* _tmpfname;
} FILE;

```

Lo mas importante es esto:

```

char *_ptr;    /* Puntero al buffer actual */
int _cnt;     /* Contador del byte actual */
char *_base;  /* Dirección base del buffer d
char _flag;   /* Flags de control */
char _file;   /* Número de fichero */

```

<pre> 004012D4 mov dword ptr [esp], offset aF0E0A_CAC ; pruebaCAC 004012DB call fopen 004012E0 mov [ebp+fichero], eax 004012E3 mov eax, [ebp+fichero] 004012E6 mov [esp+4], eax ; FILE * 004012EA mov dword ptr [esp], offset aEstoEsUnaLanea ; "Esto es 004012F1 call fputs 004012F6 mov eax, [ebp+fichero] 004012F9 mov [esp+4], eax ; FILE * 004012FD mov dword ptr [esp], offset aEstoEsOtra ; "Esto es otra </pre>	<pre> * 77C2FCDE db 0 * 77C2FCDF db 0 * 77C2FCE0 FILE <offset unk_3E3E12, 0FEEh * 77C2FD00 db 0 * 77C2FD01 db 0 * 77C2FD02 db 0 * 77C2FD03 db 0 UNKNOWN 77C2FCE0: msvcrt.dll:77C2FCE0 </pre>
---	---

Allí vemos el primer campo es un puntero, la dirección puede variar de maquina en maquina, pero veamos a donde apunta.

```

debug013:003E3E0A db 61h ; a
debug013:003E3E0B db 20h
debug013:003E3E0C db 6Ch ; l
debug013:003E3E0D db 0EDh ; ý
debug013:003E3E0E db 6Eh ; n
debug013:003E3E0F db 65h ; e
debug013:003E3E10 db 61h ; a
debug013:003E3E11 db 0Ah
debug013:003E3E12 unk_3E3E12 db 0ADh ;

```

Allí termina la string **Esto es una línea** así que este puntero, marca adonde termina el buffer actualmente o sea a partir de aquí cuando agreguemos mas strings lo hará en esta dirección, el otro puntero llamado **base** marca el inicio de todo el buffer o sea allí debe estar el inicio de la string **Esto es una línea**

```

debug013:003E3E0F db 0
debug013:003E3E00 unk_3E3E00 db 45h ; E
debug013:003E3E01 db 73h ; s
debug013:003E3E02 db 74h ; t
debug013:003E3E03 db 6Fh ; o
debug013:003E3E04 db 20h
debug013:003E3E05 db 65h ; e
debug013:003E3E06 db 73h ; s
debug013:003E3E07 db 20h
debug013:003E3E08 db 75h ; u
debug013:003E3E09 db 6Eh ; n
debug013:003E3E0A db 61h ; a
debug013:003E3E0B db 20h
debug013:003E3E0C db 6Ch ; l
debug013:003E3E0D db 0EDh ; ý
debug013:003E3E0E db 6Eh ; n
debug013:003E3E0F db 65h ; e
debug013:003E3E10 db 61h ; a
debug013:003E3E11 db 0Ah
debug013:003E3E12 unk_3E3E12 db 0ADh ; ;

```

Allí esta inicio de buffer y fin del buffer el cual cambiara cuando agreguemos mas data pasando por mas fputs.

Si seguimos traceando pasando por los restantes **fputs** veremos como el buffer va creciendo y las siguientes strings se van apendeando a continuacion, cambiando el puntero de fin de buffer.

```

1:77C2FCDE db 0
1:77C2FCDF db 0
1:77C2FCE0 FILE <offset unk_3E3E1E, 0FE2h, offset unk_3E3E00,
1:77C2FD00 db 0
1:77C2FD01 db 0
1:77C2FD02 db 0

```



```

• 13:003E3DFF db 0
• 13:003E3E00 unk_3E3E00 db 45h ; E
• 13:003E3E01 db 73h ; s
• 13:003E3E02 db 74h ; t
• 13:003E3E03 db 6Fh ; o |
• 13:003E3E04 db 20h
• 13:003E3E05 db 65h ; e
• 13:003E3E06 db 73h ; s
• 13:003E3E07 db 20h
• 13:003E3E08 db 75h ; u
• 13:003E3E09 db 6Eh ; n
• 13:003E3E0A db 61h ; a
• 13:003E3E0B db 20h
• 13:003E3E0C db 6Ch ; l
• 13:003E3E0D db 0EDh ; Ÿ
• 13:003E3E0E db 6Eh ; n
• 13:003E3E0F db 65h ; e
• 13:003E3E10 db 61h ; a
• 13:003E3E11 db 0Ah
• 13:003E3E12 unk_3E3E12 db 45h ; E
• 13:003E3E13 db 73h ; s
• 13:003E3E14 db 74h ; t
• 13:003E3E15 db 6Fh ; o
• 13:003E3E16 db 20h
• 13:003E3E17 db 65h ; e
• 13:003E3E18 db 73h ; s
• 13:003E3E19 db 20h
• 13:003E3E1A db 6Fh ; o
• 13:003E3E1B db 74h ; t
• 13:003E3E1C db 72h ; r
• 13:003E3E1D db 61h ; a
• 13:003E3E1E unk_3E3E1E db 0ADh ; ;
• 13:003E3E1F db 0BAh ; ;

```

Así que la creación del fichero y el manejo de la memoria para escribir en el mismo, es manejado por esta estructura, donde se almacenan los punteros a las strings que se guardan en un buffer de sistema.

LECTURA DE UN ARCHIVO

Si queremos leer de un fichero, los pasos son similares, sólo que lo abriremos para lectura (el modo de lectura tendrá una “r”, de “read”, en lugar de “w” que se usaba para escritura), y leeremos con “fgets”:

Vemos un código que lee el archivo que creamos en el ejemplo anterior (asegúrense de que aun exista jeje)

```

#include <stdio.h>
#include <string.h>
main(){
funcion();
getchar();
}

```

```

funcion()
{

```

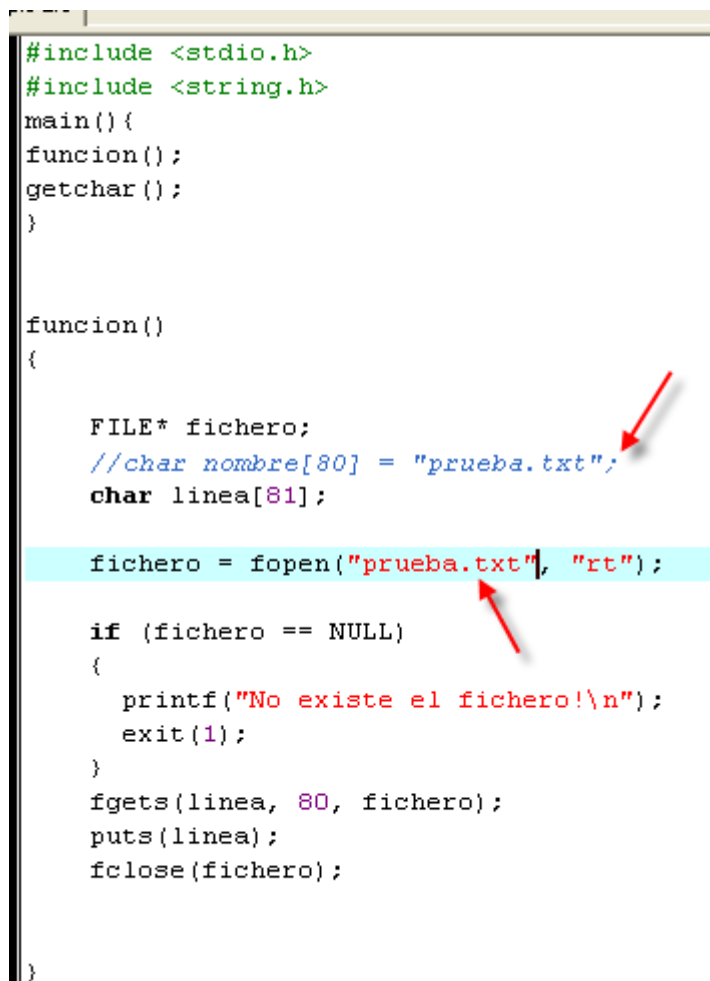
```
FILE* fichero;  
char nombre[80] = "prueba.txt";  
char linea[81];
```

```
fichero = fopen(nombre, "rt");
```

```
if (fichero == NULL)  
{  
    printf("No existe el fichero!\n");  
    exit(1);  
}  
fgets(linea, 80, fichero);  
puts(linea);  
fclose(fichero);
```

```
}
```

Vemos que en este caso creo dos arrays de caracteres uno de largo 80, al cual le asigno la string del nombre del archivo, podría haberle pasado la string en forma directa como en el caso anterior, funcionaria sin problemas, hubiera quedado así.



```
#include <stdio.h>  
#include <string.h>  
main() {  
    funcion();  
    getchar();  
}  
  
funcion()  
{  
  
    FILE* fichero;  
    //char nombre[80] = "prueba.txt";  
    char linea[81];  
  
    fichero = fopen("prueba.txt", "rt");  
  
    if (fichero == NULL)  
    {  
        printf("No existe el fichero!\n");  
        exit(1);  
    }  
    fgets(linea, 80, fichero);  
    puts(linea);  
    fclose(fichero);  
}
```

Volvamos al código original, vemos que crea un array de 80 caracteres para leer del archivo, en este

caso independientemente que el sistema mediante la estructura file guarde las strings leídas en un buffer, nosotros debemos crear una variable propia donde copiara lo leído, pues debemos almacenarlo en una variable manejable por mi.

```
#include <stdio.h>
#include <string.h>
main(){
funcion();
getchar();
}
```

```
funcion()
{

FILE* fichero;
char nombre[80] = "prueba.txt";
char linea[81];

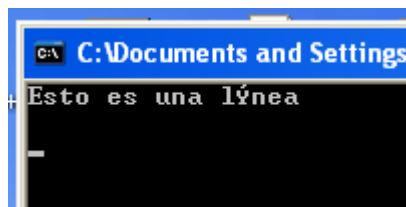
fichero = fopen(nombre, "rt");

if (fichero == NULL)
{
printf("No existe el fichero!\n");
exit(1);
}
fgets(linea, 80, fichero);
puts(linea);
fclose(fichero);

}
```

Vemos que en este caso a **fgets** le debemos pasar un array de caracteres vacío que será la variable que contendrá lo leído, aquí la variable se llama **linea**, los otros dos argumentos son el tamaño de lo leído y el puntero al fichero.

Si ejecutamos el programa vemos que fgets leerá hasta que encuentra un salto de línea y ahí terminará aunque haya más caracteres y aunque no lea los 80 que le pasamos como máximo.



También vemos que al abrir el fichero si es devuelto NULL o sea que no existe el archivo se sale del programa mediante **exit**.

```
if (fichero == NULL)
{
printf("No existe el fichero!\n");
```

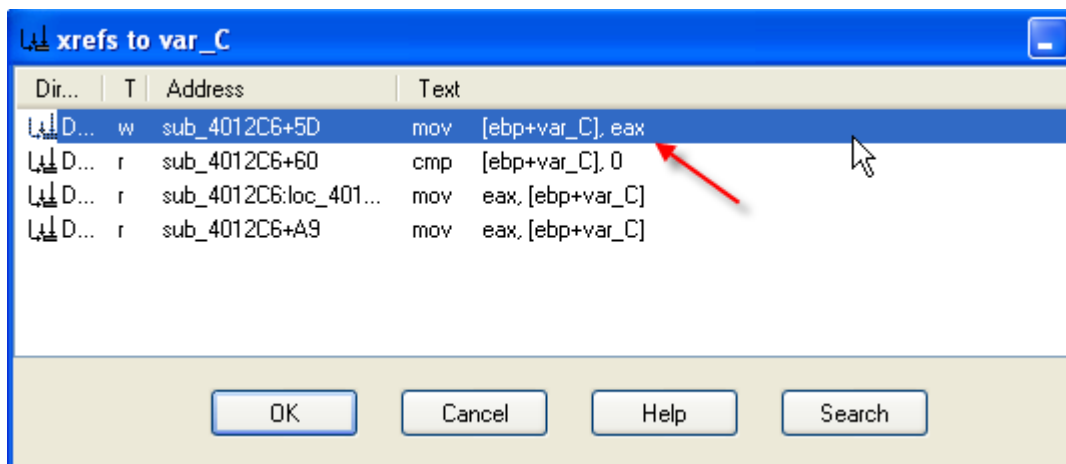
exit(1);

Bueno veamoslo en IDA y reverseemoslo.

```
var_C8= byte ptr -0C8h
var_68= byte ptr -68h
var_64= dword ptr -64h
var_60= word ptr -60h
var_5E= byte ptr -5Eh
var_5D= byte ptr -5Dh
var_C= dword ptr -0Ch

push    ebp
mov     ebp, esp
sub     esp, 0D8h
mov     eax, ds:dword_403000
mov     dword ptr [ebp+var_68], eax
mov     eax, ds:dword_403004
mov     [ebp+var_64], eax
movzx   eax, ds:word_403008
mov     [ebp+var_60], ax
movzx   eax, ds:byte_40300A
mov     [ebp+var_5E], al
lea     edx, [ebp+var_5D]
mov     eax, 45h
mov     [esp+8], eax    ; size_t
mov     dword ptr [esp+4], 0 ; int
mov     [esp], edx     ; void *
```

Bueno aquí tenemos unas cuantas variables vamos a ir renombrando y despejando el camino, empecemos con la **var_C**, apretando la tecla X vemos el momento en que se inicializa.



```
mov     [esp+8], eax    ; size_t
mov     dword ptr [esp+4], 0 ; int
mov     [esp], edx     ; void *
call    memset
mov     dword ptr [esp+4], offset aRt ; "rt"
lea     eax, [ebp+var_68]
mov     [esp], eax     ; char *
call    fopen
mov     [ebp+var_C], eax
cmp     [ebp+var_C], 0
jnz     short loc_401344
```

Vemos que se inicializa al volver de **fopen** por lo tanto **var_C** es nuestra variable **fichero** el puntero

a la estructura file, le ponemos el nombre.

```
sub_4012C6 proc near
    var_C8= byte ptr -0C8h
    var_68= byte ptr -68h
    var_64= dword ptr -64h
    var_60= word ptr -60h
    var_5E= byte ptr -5Eh
    var_5D= byte ptr -5Dh
    fichero= dword ptr -0Ch
    push    ebp
```

```
var_C8= byte ptr -0C8h
var_68= byte ptr -68h
var_64= dword ptr -64h
var_60= word ptr -60h
var_5E= byte ptr -5Eh
var_5D= byte ptr -5Dh
fichero= dword ptr -0Ch
push    ebp
mov     ebp, esp
sub     esp, 008h
mov     eax, ds:dword_403000
mov     dword ptr [ebp+var_68], eax
mov     eax, ds:dword_403004
mov     [ebp+var_64], eax
movzx   eax, ds:word_403008
mov     [ebp+var_60], ax
movzx   eax, ds:byte_40300A
mov     [ebp+var_5E], al
```

Vemos que en las cuatro variables consecutivas **var_68**, **var_64**, **var_60** y **var_5e** va leyendo el nombre del archivo que esta en 403000 en adelante y lo va copiando a dichas variables, si vemos en 403000.

.rdata:00403000	;org 403000h	
.rdata:00403000 dword_403000	dd 65757270h	; DATA XREF: sub_4012C6+9↑r
.rdata:00403004 dword_403004	dd 742E6162h	; DATA XREF: sub_4012C6+11↑r
.rdata:00403008 word_403008	dw 7478h	; DATA XREF: sub_4012C6+19↑r
.rdata:0040300A byte_40300A	db 0	; DATA XREF: sub_4012C6+24↑r
.rdata:0040300D	dh 0	

Si queremos ver la string marcamos la zona y hacemos undefine.

```

.rdata:00403000 ;org 403000h
.rdata:00403000 unk_403000 db 70h ; p ;
.rdata:00403001 db 72h ; r ;
.rdata:00403002 db 75h ; u ;
.rdata:00403003 db 65h ; e ;
.rdata:00403004 unk_403004 db 62h ; b ;
.rdata:00403005 db 61h ; a ;
.rdata:00403006 db 2Eh ; . ;
.rdata:00403007 db 74h ; t ;
.rdata:00403008 unk_403008 db 78h ; x ;
.rdata:00403009 db 74h ; t ;
.rdata:0040300A unk_40300A db 0 ;
.rdata:0040300B db 0 ;
.rdata:0040300C db 0 ;
.rdata:0040300D db 0 ;

```

Así que esta inicializando mi variable array con el nombre del archivo **Prueba.txt**, y luego el resto que no usa lo pone a cero usando **memset**, vemos que en EDX usando **lea** le pasa la dirección desde donde tiene que llenar con ceros, luego el tamaño o sea **45h (69 decimal)** y luego el valor con el cual va a llenar o sea el cero, esto no es parte del código nuestro lo agrego el compilador para inicializar correctamente el array con la string **Prueba.txt** y asegurarse que el resto quede vacío.

```

mov     [ebp+var_5E], al
lea     edx, [ebp+var_5D]
mov     eax, 45h
mov     [esp+8], eax ; size_t
mov     dword ptr [esp+4], 0 ; int
mov     [esp], edx ; void *
call    memset

```

Así que nuestro array que habíamos llamado **nombre**

char nombre[80] = "prueba.txt";

comenzara en **var_68**, y las restantes variables que hay debajo hasta **fichero** son campos del mismo con las cual trabaja, pero pertenecen al mismo array, así que undefinimos las variables que hay debajo.

```

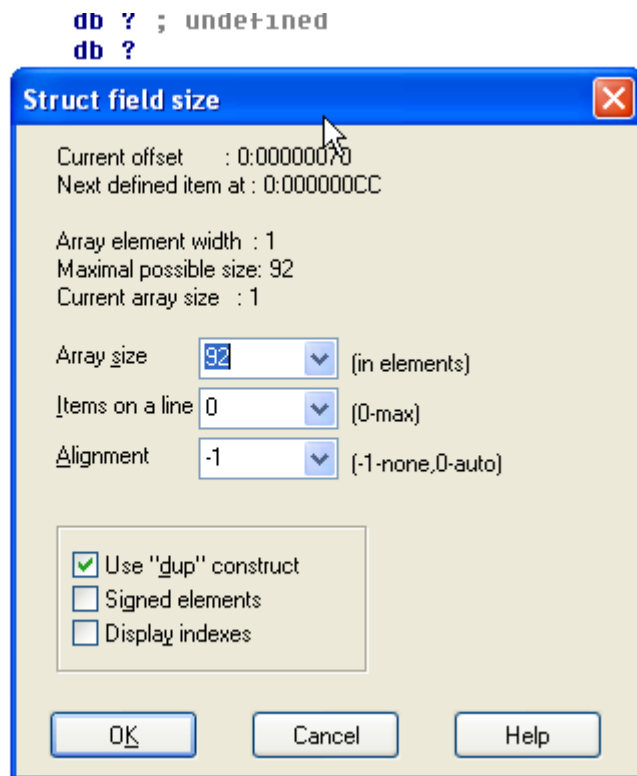
-00000007
-00000068 var_68 db 4 dup(?)
-00000064 var_64 dd ?
-00000060 var_60 dw ?
-0000005E var_5E db ?
-0000005D var_5D db ?
-0000005C db ? ; undefined
-0000005B db ? ; undefined
-0000005A db ? ; undefined

```

```

00000069
00000068 var_68
00000067
00000066
00000065
00000064
00000063
00000062
00000061
00000060
0000005F
0000005E
0000005D
0000005C
0000005B
0000005A
00000059
00000058
00000057
00000056
00000055
00000054
00000053
00000052
00000051
00000050

```



Vemos que hay lugar para 92, podemos tomarlo no hay problema aunque sea mas largo no afectara.

```

-00000000
-00000069
-00000068 nombre
-0000006C fichero
-00000068
-00000067

db ? ; undefined
db ? ; undefined
db 92 dup(?)
dd ? ; offset
db ? ; undefined
db ? ; undefined

```

La otra variable que hay mas arriba llamada **var_c8** es el otro array donde escribirá lo que lee del archivo.

```

-----
-000000C9
-000000C8 var_c8
-000000C7
-000000C6
-000000C5
-000000C4
-000000C3
-----

db ? ; undefined
db ?
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined

```

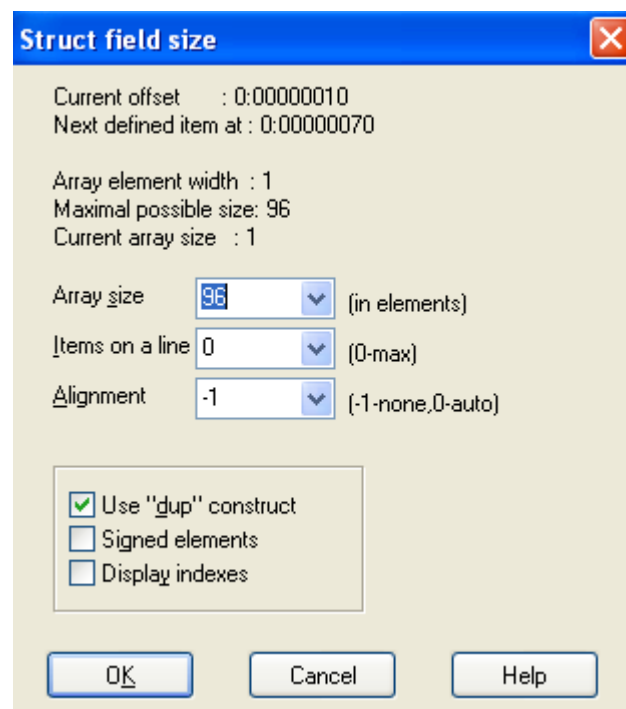
Si vemos donde la usa, vemos que con lea le pasa su dirección a **fgets** para llenar su contenido con lo que lee del archivo, y luego le pasa la misma dirección a **puts** para imprimirlo por consola.

```
loc_401344:
mov     eax, [ebp+fichero]
mov     [esp+8], eax      ; FILE *
mov     dword ptr [esp+4], 50h ; int
lea     eax, [ebp+var_C8]
mov     [esp], eax       ; char *
call    fgets
lea     eax, [ebp+var_C8]
mov     [esp], eax       ; char *
call    puts
mov     eax, [ebp+fichero]
mov     [esp], eax       ; FILE *
call    fclose
```

Así que ese es el otro array que en nuestro código habíamos llamado linea

char linea[81];

Así que creamos el array en dicha variable.



Nos quedaran las variables como en nuestro código fuente con los dos arrays y el puntero **fichero**.


```

-000000C7          db ? ; undefined
-000000C8 linea    db 96 dup(?)
-00000068 nombre   db 92 dup(?)
-0000000C fichero  dd ? ; offset
-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 s        db 4 dup(?)
+00000004 r        db 4 dup(?)
+00000008
+00000008 ; end of stack variables

```

Si vemos el código ahora, vemos que es mas fácil interpretar que esta copiando partes del nombre del archivo cuando escribe en **nombre**, **nombre+4**, **nombre+8** etc

```

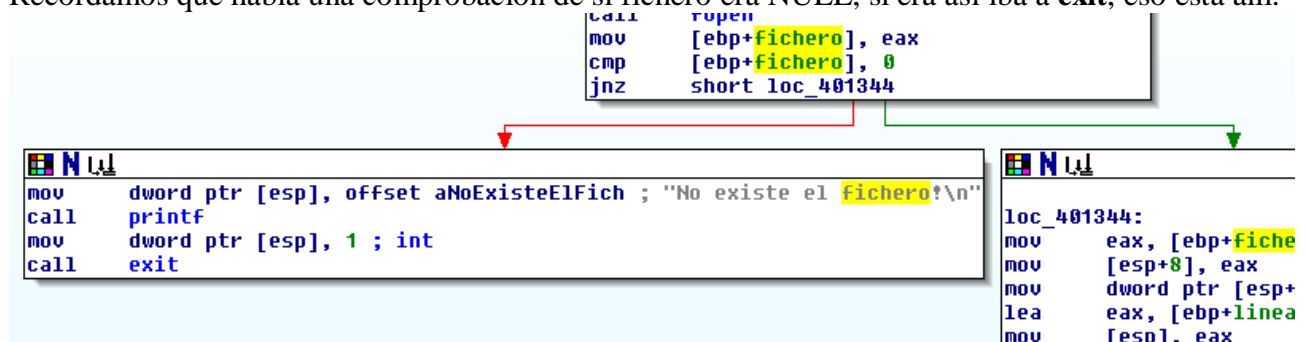
sub_4012C6 proc near

linea= byte ptr -0C8h
nombre= byte ptr -68h
fichero= dword ptr -0Ch

push    ebp
mov     ebp, esp
sub     esp, 0D8h
mov     eax, dword ptr ds:unk_403000
mov     dword ptr [ebp+nombre], eax
mov     eax, dword ptr ds:unk_403004
mov     dword ptr [ebp+nombre+4], eax
movzx   eax, word ptr ds:unk_403008
mov     word ptr [ebp+nombre+8], ax
movzx   eax, byte ptr ds:unk_40300A
mov     [ebp+nombre+0Ah], al
lea     edx, [ebp+nombre+0Bh]
mov     eax, 45h
mov     [esp+8], eax ; size_t
mov     dword ptr [esp+4], 0 ; int
mov     [esp], edx ; void *
call    memset
mov     dword ptr [esp+4], offset aRt ; "rt"
lea     eax, [ebp+nombre]
mov     [esp], eax ; char *

```

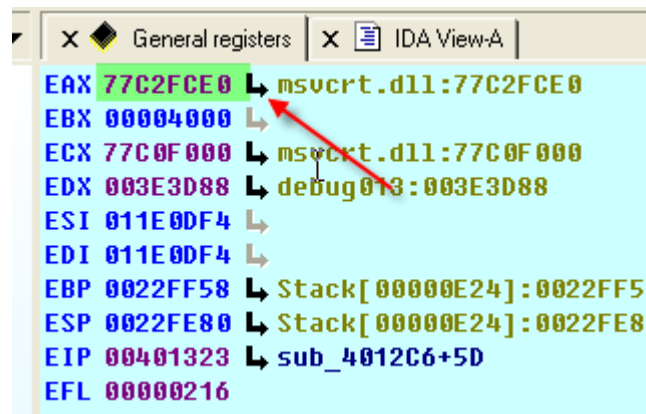
Recordamos que había una comprobación de si fichero era NULL, si era así iba a **exit**, eso esta allí.



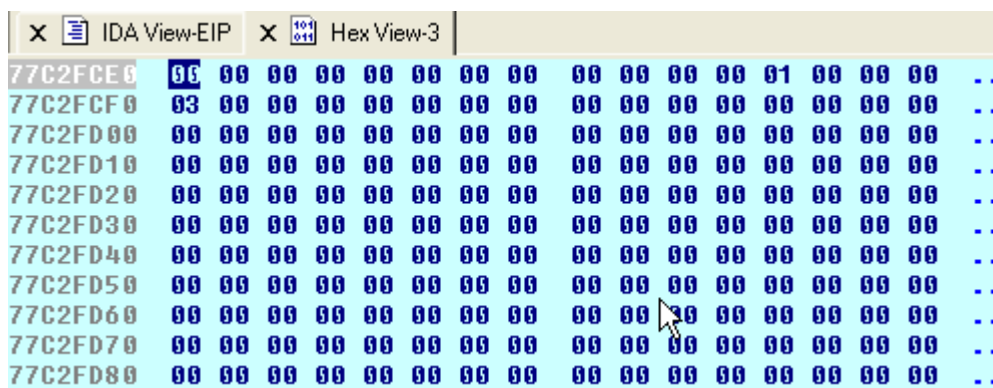
La variable **fichero** se comporta en forma similar que en el ejemplo anterior, al pasar por **fopen**, se guarda el puntero a la estructura file, en **ebp+ fichero**.

```
004012FC mov [esp+8], eax ; size_t
00401300 mov dword ptr [esp+4], 0 ; int
00401308 mov [esp], edx ; void *
0040130B call memset
00401310 mov dword ptr [esp+4], offset aRt ; "rt"
00401318 lea eax, [ebp+nombre]
0040131B mov [esp], eax ; char *
0040131E call fopen
00401323 mov [ebp+fichero], eax
00401326 cmp [ebp+fichero], 0
0040132A jnz short loc_401344
```

Si vemos la dirección en EAX



si vemos esa zona abriendo una ventana HEX y sincronizando con EAX.



Vemos que aun esta vacía pero al igual que en el caso anterior a medida que vaya pasando por fgets en este caso se ira llenado con los campos de la estructura los cuales se manejaran de la misma forma que en el caso anterior, manteniendo punteros al inicio del buffer y al final temporal del mismo.

```

loc_401344:
mov     eax, [ebp+fichero]
mov     [esp+8], eax      ; FILE
mov     dword ptr [esp+4], 50
lea     eax, [ebp+var_C8]
mov     [esp], eax        ; char*
call    fgets
lea     eax, [ebp+var_C8]
mov     [esp], eax        ; char*
call    puts
mov     eax, [ebp+fichero]
mov     [esp], eax        ; FILE
call    fclose

```

Luego para terminar como vimos fgets leerá del archivo y guardara en **var_c8** que ahora se llamara **linea** y luego con **puts** mostrara lo que leyó del archivo imprimiendo en la consola.

El adjunto a continuacion es casi similar a este ultimo ejemplo y lee el mismo archivo **Prueba.txt** me gustaría que vieran la diferencia con el que acabo de explicar.

Hasta la parte siguiente
Ricardo Narvaja