


## **C Y REVERSING (Ejercicios parte 5) por Ricnar.**

Aquí estamos tratando de reversear los ejercicios de la parte 5, el primer ejecutable es el Ejemplo 4 que si lo corremos vemos que nos invita a tipear dos palabras las compara y ve que no son la misma palabra y dice que son distintas sino dice que son iguales.



```
C:\Documents and Settings\ricnar\Escritorio
Introduce una palabra: pepito
Introduce otra palabra: jose
Son distintas
```

```
C:\ C:\Documents and Settings\ricnar
Introduce una palabra: pepe
Introduce otra palabra: pepe
Son iguales
```

Obviamente debe compararse dos strings tipeadas por el usuario eso se debe hacer con **strcmp**, también luego de comparar debe haber algún **if else** para decidir según el resultado de la comparación imprimir un mensaje o el otro.

Veamoslo en el IDA:

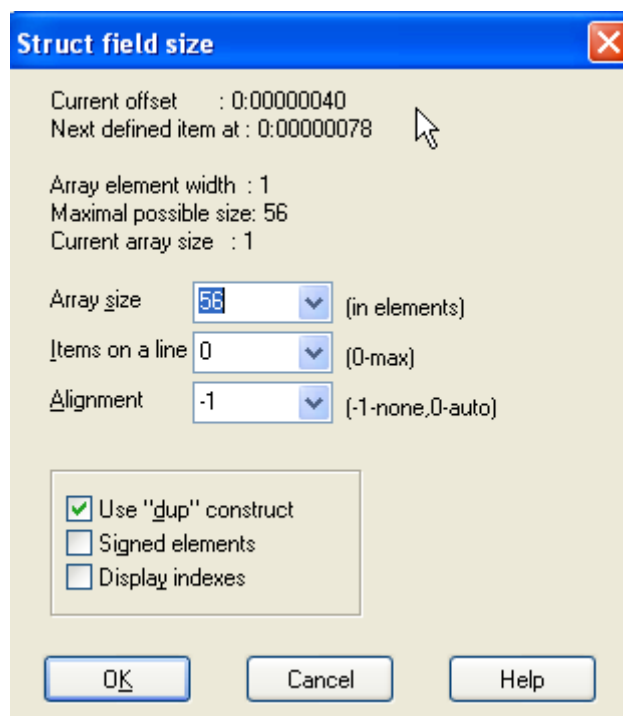
```
funcion_investigada proc near
var_68= byte ptr -68h
var_38= byte ptr -38h

push    ebp
mov     ebp, esp
sub     esp, 78h
mov     dword ptr [esp], offset aIntroduceUnaPa ; "Introduce una palabra: "
call    printf
lea     eax, [ebp+var_38]
mov     [esp], eax ; char *
call    gets
mov     dword ptr [esp], offset aIntroduceOtraP ; "Introduce otra palabra: "
call    printf
lea     eax, [ebp+var_68]
```

Allí vemos nuestra función que renombramos y vemos la primera variable **var\_38** que el IDA la muestra como un variable de un solo **byte** o sea un **char**, porque nosotros debemos pensar que es un array de caracteres mas largo de lo que el IDA nos muestra? Obviamente hay varios factores que nos llevan a pensar eso, el primero que se le pasa la dirección usando un LEA, como argumento a **gets()** que es una función para ingresar por teclado strings, por lo tanto si fuera solo un carácter usaría **getchar()** o algo así, además si vemos la tabla de variables.

·00000039	db ? ; undefined
·00000038 <b>var_38</b>	db ?
·00000037	db ? ; undefined
·00000036	db ? ; undefined
·00000035	db ? ; undefined
·00000034	db ? ; undefined
·00000033	db ? ; undefined
·00000032	db ? ; undefined
·00000031	db ? ; undefined
·00000030	db ? ; undefined
·0000002F	db ? ; undefined
·0000002E	db ? ; undefined
·0000002D	db ? ; undefined
·0000002C	db ? ; undefined
·0000002B	db ? ; undefined
·0000002A	db ? ; undefined

Vemos que si **var\_38** fuera una variable de un solo byte hay muchísimo lugar desperdiciado, algo que no suelen hacer los procesadores, cuando declaramos variables tipo **byte**, **word**, o **dword**, usan el espacio justo, solo reservan lugar cuando realmente es un buffer para guardar un array, en ese ultimo caso a veces pueden desperdiciar algunos bytes, pero no van a usar 56 bytes de lugar al pedo para un solo carácter, así que ahí debe ir un buffer reservado para un array de caracteres o string que tipeara el usuario, si aprieto asterisco en la variable veo el lugar vacío antes de pisar nada que nos muestra que es 56.



Ahora hemos visto ejercicios donde pisamos alguna variable y otros en que no pisamos, como sabemos cuando pisar y cuando no?

Algunas reglas para decidir el tamaño son:

- 1) Obvio nunca jamás llegar a pisar el **stored ebp** y el **return address**, eso nunca debe pisarse, si aquí hay 56 hasta el **stored ebp** y al **return address**, eso es el máximo que puedo colocar y nunca más.
- 2) Cuando definimos arrays largos de arrays solo pisamos una variable y la hacemos desaparecer, cuando según el análisis vemos que fue una variable que se creo de mas y que posiblemente no este en el código fuente, y se creo demás porque en realidad el programa realiza operaciones con los campos intermedios del array, y como normalmente IDA no detecta automáticamente los mismos,

debe crear una variable para guardar los resultados de esas operaciones, si definimos como array y pisamos esa variable, entonces las operaciones se realizarán como campo del mismo y no necesitará dicha variable suelta la cual desaparecerá, ese es el único caso en que hay que pisar una variable y hay que analizar bien que corresponda hacerlo.

Bueno aquí podríamos dejar los 56 caracteres sin problemas, a pesar de que sabemos que el original debe ser un poco menor por experiencia, lo pondremos al máximo para ver que igual funciona.

00000007		00 : , undetected
-00000068	var_68	db 48 dup(?)
-00000038	var_38	db 56 dup(?)
+00000000	s	db 4 dup(?)
+00000004	r	db 4 dup(?)
+00000008		

Hacemos el mismo análisis para la otra variable que tiene 48 como máximo y lo elijo, veo que no queda espacio de más.

```

palabra2= byte ptr -68h
palabra1= byte ptr -38h

push    ebp
mov     ebp, esp
sub     esp, 78h
mov     dword ptr [esp], offset aInte
call    printf
lea     eax, [ebp+palabra1]
mov     [esp], eax      ; char *
call    gets
mov     dword ptr [esp], offset aInte
call    printf
lea     eax, [ebp+palabra2]
mov     [esp], eax      ; char *
call    gets
lea     eax, [ebp+palabra2]
lea     edx, [ebp+palabra1]
mov     [esp+4], eax     ; char *
mov     [esp], edx      ; char *
call    strcmp

```

Aquí como no usa campos intermedios del array, o sea letras sueltas no hay diferencia, pero es bueno realizar el análisis correcto, pues aunque aquí se vea similar si hacemos un código fuente declarando solo dos variables char como mostraba IDA y lo compilamos no funcionará como el original.

Bueno armemos el código:

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

```
main(){
```

```
    funcion1();
```

```
}
```

```
funcion1(){
```

```
.....
```

```
}
```

El esquema básico es el mismo pues es una funcion sin argumentos ni valores de retorno que es llamada desde el **main()**.

Como vemos en IDA que usa **strcmp** debemos agregar el include correspondiente:

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

```
#include <string.h>
```

```
main(){
```

```
    funcion1();
```

```
}
```

```
funcion1(){
```

```
.....
```

```
}
```

Luego vemos las dos variables de array que analizamos, las agregamos al código:

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

```
#include <string.h>
```

```
main(){
```

```
    funcion1();
```

```
}
```

```
funcion1(){
```

```
char palabra1 [56];
```

```
char palabra2 [48];
```

```
.....
```

```
}
```

Luego viene un llamado a **printf** con la string “**Introduce una palabra**”, lo agregamos.

```
palabra2= byte ptr -68h  
palabra1= byte ptr -38h
```

```
push    ebp
```

```
mov     ebp, esp
```

```
sub     esp, 78h
```

```
mov     dword ptr [esp], offset aIntroduceUnaPa ; "Introduce una palabra: "
```

```
call    printf
```

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

```
#include <string.h>
```

```
main(){
```

```
funcion1();

}
```

```
funcion1(){
char palabra1 [56] ;
char palabra2 [48] ;
```

```
printf ("Introduce una palabra: ");
```

```
}
```

Luego llama a **gets()** con el argumento **palabra1**, vimos que al mismo no es necesario aclararle que es la dirección con **&**, solo lo reconoce, así que lo agregamos.

```
call    printf
lea     eax, [ebp+palabra1]
mov     [esp], eax ; char *
call    gets
```

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

```
#include <string.h>
```

```
main(){
```

```
funcion1();
```

```
}
```

```
funcion1(){
char palabra1 [56] ;
char palabra2 [48] ;
```

```
printf ("Introduce una palabra: ");
```

```
gets(palabra1);
```

```
}
```

Luego viene otro **printf** con el otro texto **"Introduce otra palabra: "** y luego un **gets()** con el argumento **palabra2** para guardar la segunda palabra.

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

```
#include <string.h>
```

```
main(){
```

```
funcion1();
```

```
}
```

```

funcion1(){
char palabra1 [56] ;
char palabra2 [48] ;

printf ("Introduce una palabra: ");
gets(palabra1);
printf ("Introduce otra palabra: ");
gets(palabra2);

}

```

Luego usa LEA para obtener las direcciones de ambas variables y las pasa a ambas como argumentos de **strcmp** (la cual no necesita el & como las apis que solo manejan strings) aunque vemos que no usa una variable mas para guardar el resultado, directamente testea el mismo que al volver de la api queda en EAX, eso en el código fuente se vería así:

```

lea     eax, [ebp+palabra2]
lea     edx, [ebp+palabra1]
mov     [esp+4], eax      ; char *
mov     [esp], edx       ; char *
call    strcmp

```

```

#include <stdio.h>
#include <string.h>

```

```

main(){

```

```

    funcion1();

```

```

}

```

```

funcion1(){
char palabra1 [56] ;
char palabra2 [48] ;

```

```

printf ("Introduce una palabra: ");
gets(palabra1);
printf ("Introduce otra palabra: ");
gets(palabra2);

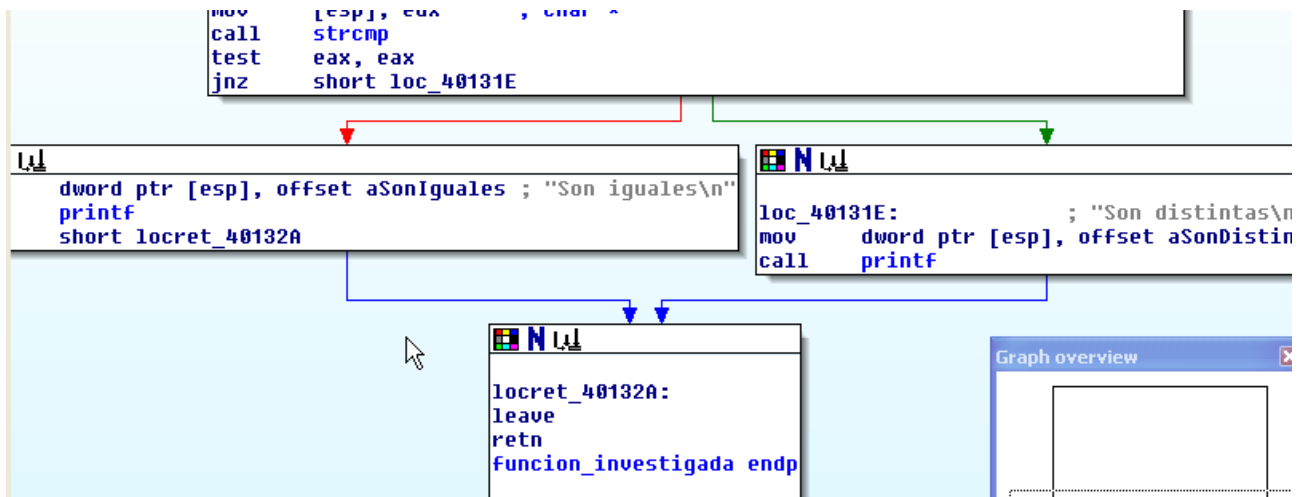
```

```

if (strcmp(palabra1,palabra2)==0) printf ("Son iguales\n");
else printf ("Son distintas\n");
}

```

Vemos como embebimos **strcmp** dentro del **if** para no tener que crear una variable **int** mas para el resultado de la misma, en el IDA vemos que como resultado del **if else** se bifurca el flujo del programa, si son iguales va a **printf** del texto **Son iguales** por el camino de la flecha roja ya que sera cero, y si no es cero la comparación sera verdadera y usara el camino verde a imprimir el texto **Son distintas**.



Así que hemos reverseado completamente el primer ejemplo el segundo ejemplo es casi similar no repetiremos todo nuevamente, veamos las diferencias al ejecutarlo.

```

C:\Documents and Settings\ricnar\Escritorio>
Introduce una palabra: po
Introduce otra palabra: pepe
La primera palabra es mayor

```

Vemos que compara a ver cual es mayor alfabéticamente, el resultado de la comparación de **strcmp** nos dice eso según el valor numérico del mismo si es cero serán iguales si es mayor que cero será mas grande la primera y si no será mas grande la segunda, así que veamos en el IDA.

```

var_6C= dword ptr -6Ch
var_68= byte ptr -68h
var_38= byte ptr -38h

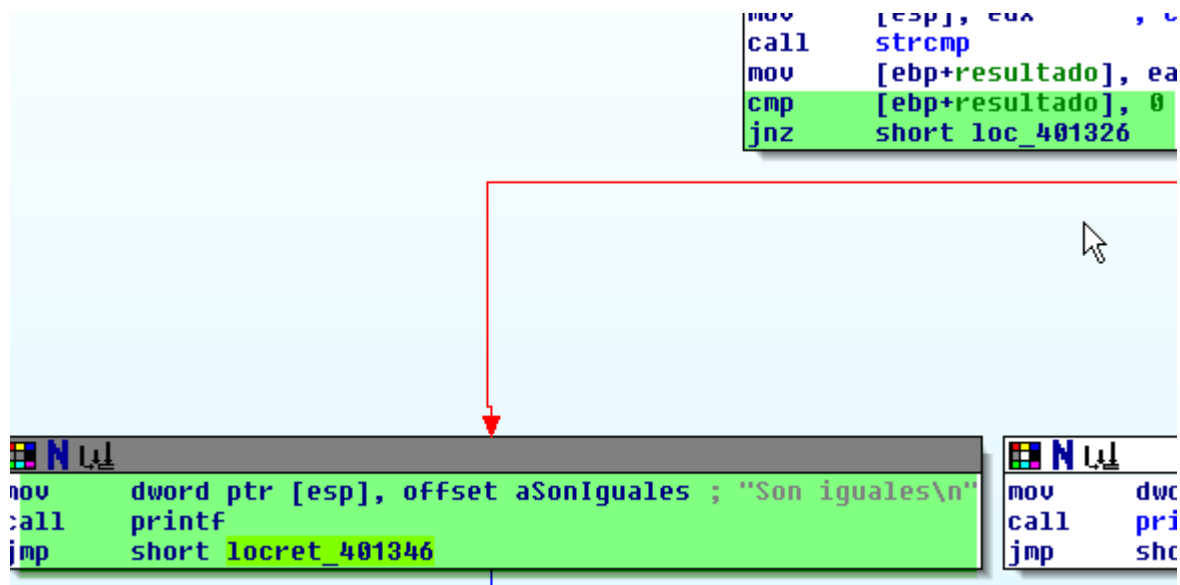
push    ebp
mov     ebp, esp
sub     esp, 88h
mov     dword ptr [esp], offset aIntroduceU
call    printf
lea     eax, [ebp+var_38]
mov     [esp], eax ; char *
call    gets
mov     dword ptr [esp], offset aIntroduceO
call    printf
lea     eax, [ebp+var_68]
mov     [esp], eax ; char *
call    gets
lea     eax, [ebp+var_68]
lea     edx, [ebp+var_38]
mov     [esp+4], eax ; char *
mov     [esp], edx ; char *
call    strcmp
mov     [ebp+var_6C], eax
cmp     [ebp+var_6C], 0
jnz     short loc_401326

```

Vemos que es similar al anterior los dos arrays **palabra1** y **palabra2** son igual al caso anterior solo aquí hay una variable **int** mas para guardar el resultado de **strcmp**, así que iré donde están las

variables y haré lo mismo que en el caso anterior y al **int** le pondré como nombre resultado.

Vemos que el resto es similar al anterior salvo que guarda el resultado de la comparación y lo compara con 0 si es así imprime **Son Iguales**



```
resultado = strcmp(palabra1, palabra2);
```

```
if (resultado==0)
    printf("Son iguales\n");
```



Luego realiza otra comparación que podría un **else if** ya que si no es cero vuelve a comparar y si es menor o igual va por el camino de la flecha verde de comparación verdadera e imprime que la **Segunda palabra es mayor** y si no va por el camino de la flecha roja dado que el resultado es positivo y dice que **La primera palabra es mayor** el código completo sería.



```

#include <stdio.h>
#include <string.h>

main(){

    funcion1();

}

funcion1(){
char palabra1 [56] ;
char palabra2 [48] ;
int resultado ;

printf ("Introduce una palabra: ");
gets(palabra1);
printf ("Introduce otra palabra: ");
gets(palabra2);

    resultado = strcmp(palabra1, palabra2);

    if (resultado==0)
        printf("Son iguales\n");
    else if (resultado>0)
        printf("La primera palabra es mayor\n");
    else
        printf("La segunda palabra es mayor\n");
}

```

Bueno con esto hemos reverseado ambos ejemplos nos vemos en la siguiente parte  
Hasta la vista baby  
Ricardo Narvaja