
Reverse Engineering

Josep Vañó Chic

Índice

Introducción	3
1. <i>Reverse Engineering</i>	4
2. Lenguaje Ensamblador	5
3. Código de ejemplo.....	6
4. Información general	8
5. Strings	10
6. Variables	10
7. Pila o Stack.....	13
8. MOV y LEA	13
9. Registros	14
10. Bucles.....	15
11. Condicionales.....	22
12. Subrutinas.....	27
13. Renombrar y personalizar	41
14. Paso de parámetros y flujo de ejecución	44
15. Alteración del código.....	49

Introducción

La ingeniería inversa de software permite obtener o deducir el código fuente de un programa a partir del fichero ejecutable, este hecho tiene sus ventajas, pero también puede implicar actividades malintencionadas.

Este módulo permite realizar una introducción a la ingeniería inversa a partir de un ejemplo de un fichero ejecutable y se muestra como deducir el código ensamblador para deducir cuál sería su código fuente.

Existen multitud de libros, tutoriales, videotutoriales sobre la ingeniería inversa, algunos de ellos de varios cientos de páginas, así pues, este módulo solo pretende realizar una introducción guiada sobre como realizarla y detectar algunas vulnerabilidades

1. *Reverse Engineering*

En la ingeniería inversa se trata de desarmar un objeto para ver cómo funciona para duplicar o mejorar el objeto. Esta ha sido una práctica realizada por las industrias más antiguas, así pues, este concepto también se ha aplicado y se está aplicando con frecuencia en hardware y software de aplicaciones informáticas en toda su gama de plataformas.

La ingeniería inversa de software implica invertir el código máquina de un programa ejecutable a código fuente.

Objetivos de ingeniería inversa

- Investigar y descubrir cómo funciona un programa
- Construir un programa compatible
- Localizar funcionalidad oculta, por ejemplo, acceso a través de puertas traseras
- Buscar vulnerabilidades software ya por el uso de funciones no seguras o por un diseño inconsistente

Análisis

- Rastrear, observar el código para comprender cómo funciona
- Poder generar código de alto nivel a medida que avanza.
- Observar y deducir el flujo de llamadas a las funciones
- Determinar los tipos de datos utilizados en un programa.
- Observar los valores de los registros, pila, flags

Pero la ingeniería inversa también tiene sus riesgos en caso de usarse para modificar código con fines malintencionados, de esta forma los profesionales en desarrollo de software deben ser conscientes de estos riesgos y por lo tanto investigar como desarrollar software seguro tanto las funciones a implementar como la lógica y el flujo del software ya que si está colocando código confidencial en un entorno en el que un atacante puede obtener acceso, debería preocuparse por los riesgos de la ingeniería inversa o la modificación no autorizada del código.

Existen gran diversidad de herramientas en el mercado para realizar *Reverse Engineering* a partir de desensambladores y debugadores. Para realizar este módulo se han utilizado dos herramientas gratuitas IDA y Ghidra.

IDA https://www.hex-rays.com/products/ida/support/download_freeware/

Ghidra <https://ghidra-sre.org/>

2. Lenguaje Ensamblador

Programación en ensamblador (x86-64)

http://cv.uoc.edu/annotation/8255a8c320f60c2bfd6c9f2ce11b2e7f/619469/PID_00218273/PID_00218273.html#w31aac15b9c15c17

Para realizar *Reverse Engineering* deberemos partir de un fichero ejecutable y como veremos en este módulo se trata de desensamblarlo y por lo tanto obtener el código en lenguaje ensamblador.

Así pues, para poder llevar a cabo *Reverse Engineering* e investigar el desensamblado y deducir como se ha desarrollado el código fuente que lo ha generado, se deben tener unos conocimientos básicos del lenguaje Assembler.

Este módulo no pretende ser una guía de aprendizaje del lenguaje ensamblador, pero sí que se recomienda su lectura acompañada con la del módulo *Programación en ensamblador (x86-64)*, ya que este módulo podréis profundizar en este lenguaje, además podréis encontrar ejemplos ilustrativos de cómo se convierte el código fuente en lenguaje ensamblador, en cambio en este módulo el proceso es a la inversa, es decir, a partir del lenguaje ensamblador deducir como se ha desarrollado el código fuente.

A continuación, se muestra un ejemplo simple del proceso de lenguaje C a lenguaje ensamblador

```
if (a > b) {  
    maxA = 1;  
    maxB = 0;  
}
```

```
mov rax, qword [a]      ;Se cargan las variables en registros  
mov rbx, qword [b]  
  
cmp rax, rbx            ;Se realiza la comparación  
jg cierto                ;Si se cumple la condición, salta a la etiqueta cierto  
jmp fin                 ;Si no se cumple la condición, salta a la etiqueta fin  
  
cierto:  
    mov byte [maxA], 1 ;Estas instrucciones solo se ejecutan  
    mov byte [maxB], 0 ;cuando se cumple la condición  
  
fin:
```

3. Código de ejemplo

Aunque en *Revere Engineering* se trata de analizar el código desensamblado, para facilitar el seguimiento de este módulo y facilitar el aprendizaje se aporta el código fuente. Este código se ha compilado y desensamblado posteriormente con las herramientas IDA y Ghidra.

El programa consta de tres funciones:

- main
 - Entrada principal del programa
- introducirPassword
 - Introducción de la contraseña por teclado del usuario
- validationPassword
 - Genera el password a través de un bucle con codigos ASCII generando el password ABCD
 - Comprueba si el password introducido por el usuario es correcto
- accesoValido.
 - Acceso en caso que la contraseña introducida por el usuario sea correcta

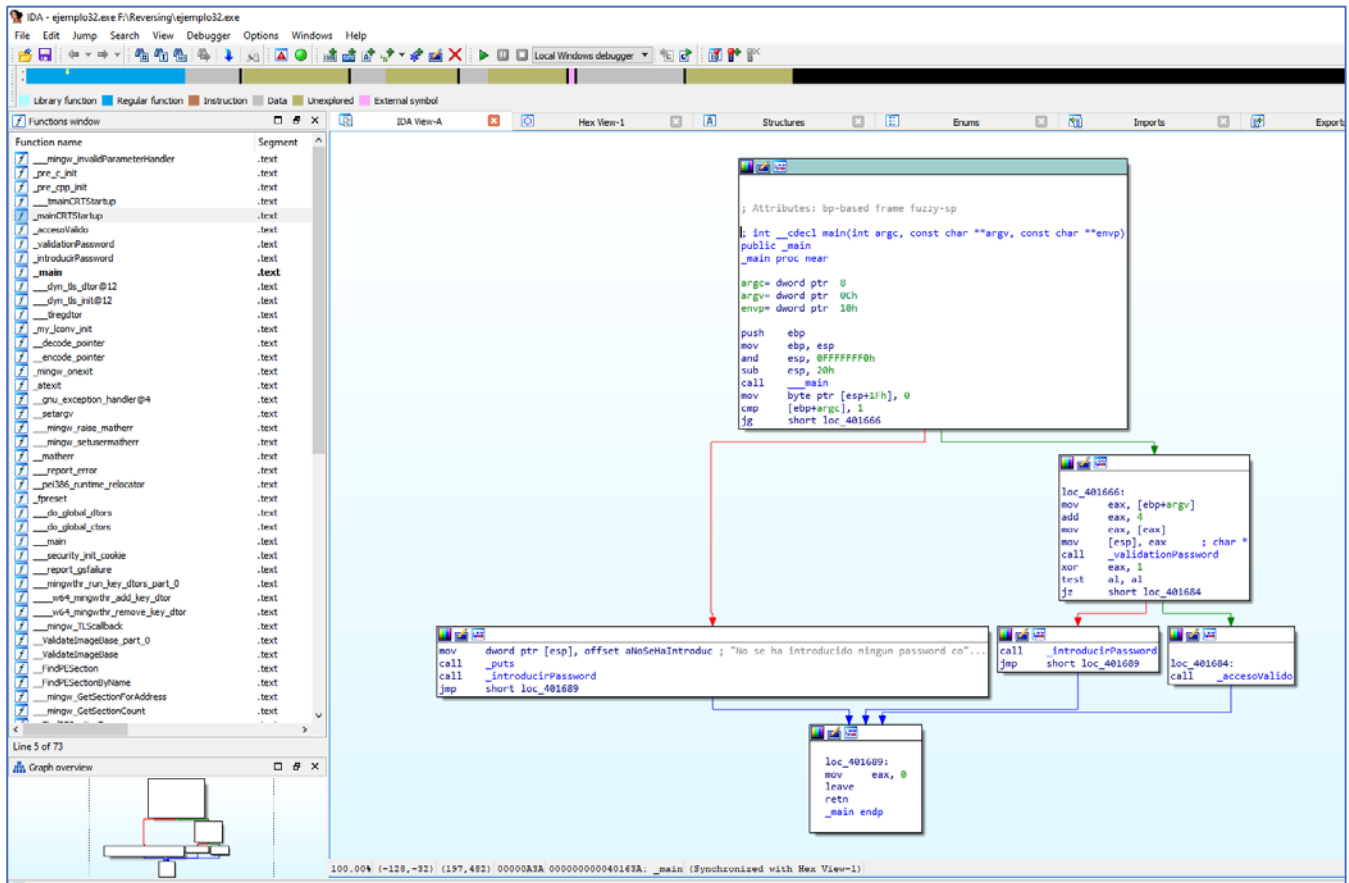
```
[*] ejemplo.c
1  // ejemplo.c
2  //
3  #include <stdio.h>
4  #include <string.h>
5  #include <stdbool.h>
6
7  void accesoValido(){
8
9      printf("\nTienes acceso al programa:\n");
10
11 }
12
13 bool validationPassword(char* password) {
14     char buffer[10];
15     char pass[5];
16     int i;
17
18     // Codigos ASCII de ABCD
19     for(i = 65; i<69; i++){
20         pass[i-65] = i;
21     }
22     pass[4]=0; // Código ASCII 0
23
24     strcpy(buffer, password);
25
26     // Comprobación de la contraseña
27     bool success = (strcmp(buffer, pass) == 0);
28     if(success) {
29         // La contraseña es correcta
30         accesoValido();
31     }
32     else {
33         // La contraseña es incorrecta
34         printf("Password incorrecto. \n");
35     }
36     return success;
37 }
```

```
38
39 int introducirPassword(){
40     char password[100];
41     bool success = false;
42     do {
43         // Introducir contraseña
44         printf("\nIntroducir password: ");
45         fgets(password, 100, stdin);
46
47         // reemplaza el salto de línea del final de la cadena
48         // por el caracter de final cadena '\0', ASCII 0
49         size_t length = strlen(password) - 1;
50
51         if (password[length] == '\n') {
52             password[length] = '\0';
53         }
54
55         if (strcmp(password, "-") == 0){
56             printf("\nNo se ha introducido una password correcto, No tienes acceso al programa");
57             return 0;
58         }
59         success = validationPassword(password);
60     } while(!success);
61     return 1;
62 }
63
64
65
66 int main(int argc, char *argv[])
67 {
68     bool success = false;
69     // Comprueba que se ha introducido un password como argumento del programa
70     if(argc < 2) {
71         printf("No se ha introducido ningun password como parametro.\n");
72         introducirPassword();
73     }
74     else {
75         if(!validationPassword(argv[1])) {
76             introducirPassword();
77         }
78         else{
79             accesoValido();
80         }
81     }
82
83     return 0;
84 }
```

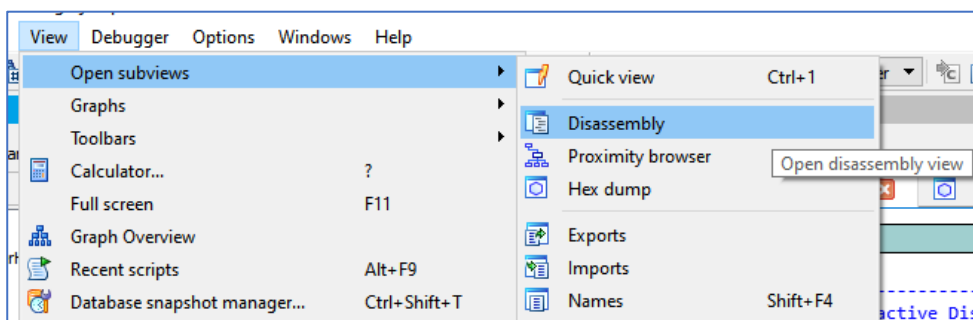
Hay que tener en cuenta que este código contiene funciones vulnerables como strcpy y strcmp, permite introducir un password de una longitud superior a la longitud máxima de la contraseña y tampoco lo comprueba, todos estos elementos se han incluido expresamente para detectarlos en el momento de realizar la ingeniería inversa y observar las consecuencias que conlleva no aplicar un desarrollo de programación de código seguro.

4. Información general

Para realizar este módulo utilizaremos un programa sencillo el cual se ha compilado en dos versiones de 32 y 64 bits.



En primer lugar, al elegir la opción *Disassemble*



Podemos obtener información básica sobre el ejecutable, por ejemplo si se ha compilado en versión de 32 o 64 bits.

Información de un ejecutable compilado en 32 bits

```
;
; Input SHA256 : 05BEA148115FA223889384AA458F6C86F53189324D718AFB72A7B8025432F113
; Input MD5    : 7981ACF9B3170C88080FE3E8635ABE7C
; Input CRC32  : 28DEC1BE

; File Name    : F:\Reversing\ejemplo32.exe
; Format       : Portable executable for 80386 (PE)
; Imagebase    : 400000
; Timestamp    : 5F04C367 (Tue Jul 07 18:48:07 2020)
; Section 1. (virtual address 00001000)
; Virtual size : 000017F0 ( 6128.)
; Section size in file : 00001800 ( 6144.)
; Offset to raw data for section: 00000400
; Flags 60500020: Text Executable Readable
; Alignment    : 16 bytes

.686p
.mmx
.model flat
.intel_syntax noprefix

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
```

En la versión del mismo programa compilado en 64 bits

```
;
; Input SHA256 : 32AFD7138A55800235F51473D97977153A31B5F0CB710A8458BCFFAA4F22EEB9
; Input MD5    : 70D92596958288BA1FD4558FEE11EA1B
; Input CRC32  : F57D17E5

; File Name    : F:\Reversing\ejemplo64.exe
; Format       : Portable executable for AMD64 (PE)
; Imagebase    : 400000
; Timestamp    : 5F04C312 (Tue Jul 07 18:46:42 2020)
; Section 1. (virtual address 00001000)
; Virtual size : 00001E20 ( 7712.)
; Section size in file : 00002000 ( 8192.)
; Offset to raw data for section: 00000600
; Flags 60500020: Text Executable Readable
; Alignment    : 16 bytes
; OS type      : MS Windows
; Application type: Executable

.686p
.mmx
.model flat

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
```

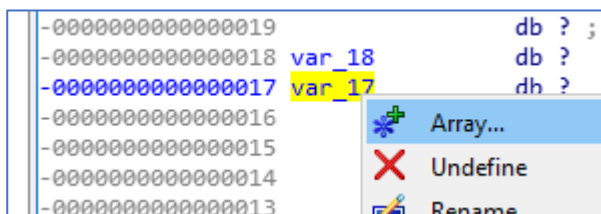


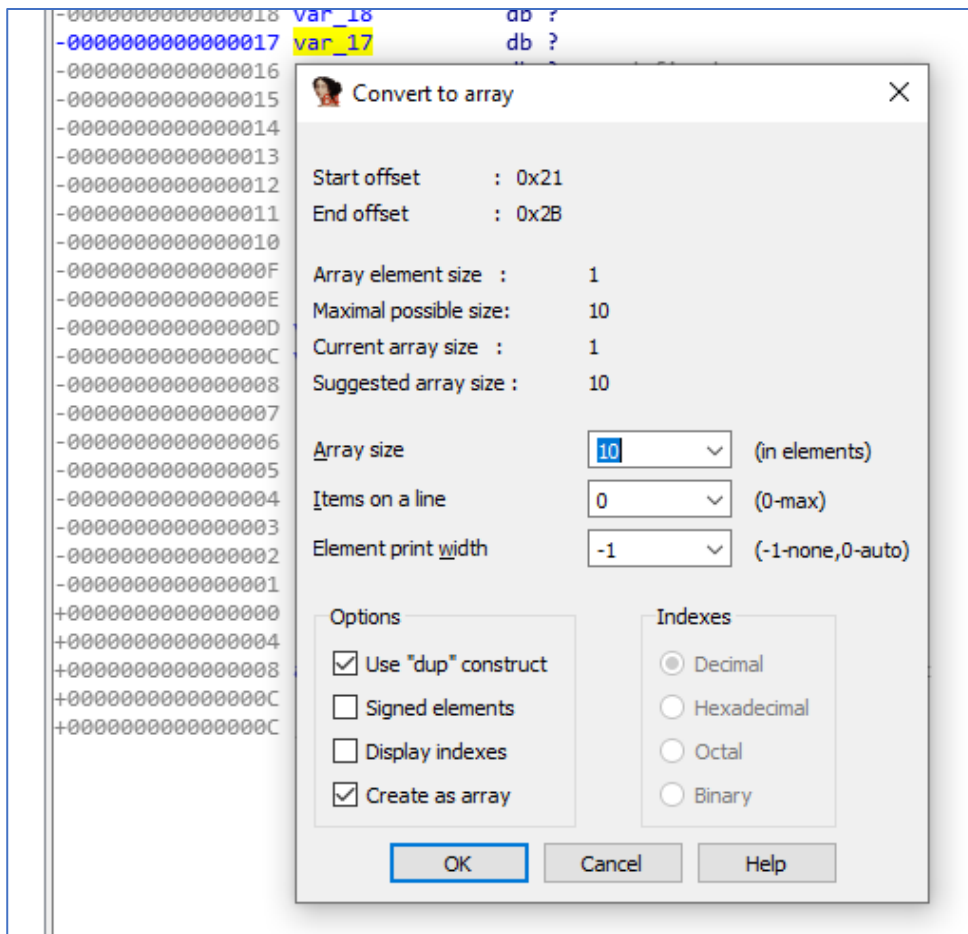
```

-00000000000000001D      db ? ; undefined
-00000000000000001C  var_1C      db ?
-00000000000000001B      db ? ; undefined
-00000000000000001A      db ? ; undefined
-000000000000000019      db ? ; undefined
-000000000000000018  var_18      db ?
-000000000000000017  var_17      db ?
-000000000000000016      db ? ; undefined
-000000000000000015      db ? ; undefined
-000000000000000014      db ? ; undefined
-000000000000000013      db ? ; undefined
-000000000000000012      db ? ; undefined
-000000000000000011      db ? ; undefined
-000000000000000010      db ? ; undefined
-00000000000000000F      db ? ; undefined
-00000000000000000E      db ? ; undefined
-00000000000000000D  var_D      db ?
-00000000000000000C  var_C      dd ?
-000000000000000008      db ? ; undefined
-000000000000000007      db ? ; undefined
-000000000000000006      db ? ; undefined
-000000000000000005      db ? ; undefined
-000000000000000004      db ? ; undefined
-000000000000000003      db ? ; undefined
-000000000000000002      db ? ; undefined
-000000000000000001      db ? ; undefined
+000000000000000000      s      db 4 dup(?)
+000000000000000004      r      db 4 dup(?)
+000000000000000008  arg_0      dd ? ; offset
+00000000000000000C ; end of stack variables

```

En este contexto disponemos de varias opciones a través del menú contextual, por ejemplo convertir en array.





Cuyo resultado seria

```

-0000000000000010 db ? ; undefined
-000000000000001C var_1C db ?
-0000000000000018 db ? ; undefined
-000000000000001A db ? ; undefined
-0000000000000019 db ? ; undefined
-0000000000000018 var_18 db ?
-0000000000000017 var_17 db 10 dup(?)
-000000000000000D var_D db ?
-000000000000000C var_C dd ?
-0000000000000008 db ? ; undefined

```

7. Pila o Stack

La pila es una sección de la memoria que permite almacenar en un modo de acceso LIFO, es decir el *último en entrar, primero en salir* de forma que solo se tiene acceso a la parte superior de la pila.

Para el manejo de los datos de la pila existen dos operaciones apilar o PUSH, que coloca un objeto en la pila, y su operación inversa, recoger o POP que retira el último elemento apilado.

En el siguiente ejemplo al acceder a la función `introducirPassword` se guarda el valor almacenado en el registro `ebp` en la pila

```
public _validationPassword
_validationPassword proc near

var_1C= byte ptr -1Ch
var_18= byte ptr -18h
var_17= byte ptr -17h
var_D= byte ptr -0Dh
var_C= dword ptr -0Ch
arg_0= dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 38h
mov     [ebp+var_C], 41h
jmp     short loc_401534
```

En la ejecución del programa podremos observar los valores de la Pila en la ventana *Stack view*.

Stack view		Call Stack
0061FDEC	0040161E	introducirPassword+95
0061FDF0	0061FE04	debug010:0061FE04
0061FDF4	00404049	.rdata:asc_404049
0061FDF8	759D4600	msvcrt.dll:msvcrt__iob
0061FDFC	F0903ACB	
0061FE00	FFFFFFFE	
0061FE04	34333231	
0061FE08	00003635	
0061FE0C	759D4620	msvcrt.dll:msvcrt__iob+20
0061FE10	007726B8	debug036:007726B8
0061FE14	00000035	
0061FE18	0061FE24	debug010:0061FE24
0061FE1C	75986FF5	msvcrt.dll:msvcrt__unlock+15

8. MOV y LEA

Para comprender mejor el funcionamiento del programa a continuación describiremos las diferencias entre estas dos instrucciones.

`mov destino, fuente.`

Copia el valor del operando fuente sobre el operando destino, sobrescribiendo el valor original del operando destino. El operando destino puede ser un registro de propósito general, un registro de segmento o una dirección de memoria. El operando fuente puede ser un valor numeral, un registro de propósito general, un registro de segmento o una dirección de memoria. Se debe cumplir la condición de que los dos operandos sean del mismo tamaño, sea byte, Word, dword o qword

`lea destino, fuente.`

Asigna la dirección de memoria del operando fuente en el registro del operando destino. El operando destino debe ser un registro de propósito general.

9. Registros

Un registro es una zona en la memoria del procesador donde se almacena un valor único. Hay que tener en cuenta que solo existen un conjunto determinado de registros cada uno con propósito específico.

En arquitecturas x86 de 32 bits podemos encontrar los registros de propósito general siguientes:

EAX (*Extended Accumulator Register*). Se utiliza como contenedor para resolver operaciones matemáticas simples o como registro de propósito general.

EBX (*Extended Base Register*) – Registro de propósito general.

ECX (*Extended Counter Register*) – Registro utilizado generalmente como contador en un bucle.

EDX (*Extended Data Register*) – Registro de propósito general, usado también como parámetro en funciones o direccionar datos en memoria.

ESI (*Extended Source Index*) – Registro generalmente utilizado como puntero. Es utilizado en funciones que requieren un origen y un destino para los datos que se utilizan almacenando el origen.

EDI (*Extended Destination Index*) - Igual que el registro ESI, usado como puntero, en este caso apuntando al destino.

EBP (*Extended Base Pointer*) – Registro utilizado como puntero a una dirección de memoria, también puede ser utilizado como registro de propósito general.

ESP (*Extended Stack Pointer*) – Almacena un puntero a la parte superior de la pila.

EIP (*Extended Instruction Pointer*) - Puntero de instrucciones. Contiene el contador del programa, la dirección de la próxima instrucción.

En la arquitectura de 64 bits, los registros de propósito general son:

16 registros de datos RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP y R8-R15.

La denominación de ocho primeros registros es parecida a los 8 registros de propósito general de 32 bits: (EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP).

Los registros son accesibles de la forma siguiente:

- Como registros de 64 bits (quad word). (En la arquitectura de 64 bits)
- Como registros de 32 bits (double word), accediendo a los 32 bits de menos peso.
- Como registros de 16 bits (word), accediendo a los 16 bits de menos peso.
- Como registros de 8 bits (byte), permitiendo acceder individualmente a uno o dos de los bytes de menos peso según el registro.
-

Tamaño de los operandos

Un operando puede ser de los tipos: byte, word, double word y quad word. Hay que tener en cuenta que lo hace en formato *little-endian*.

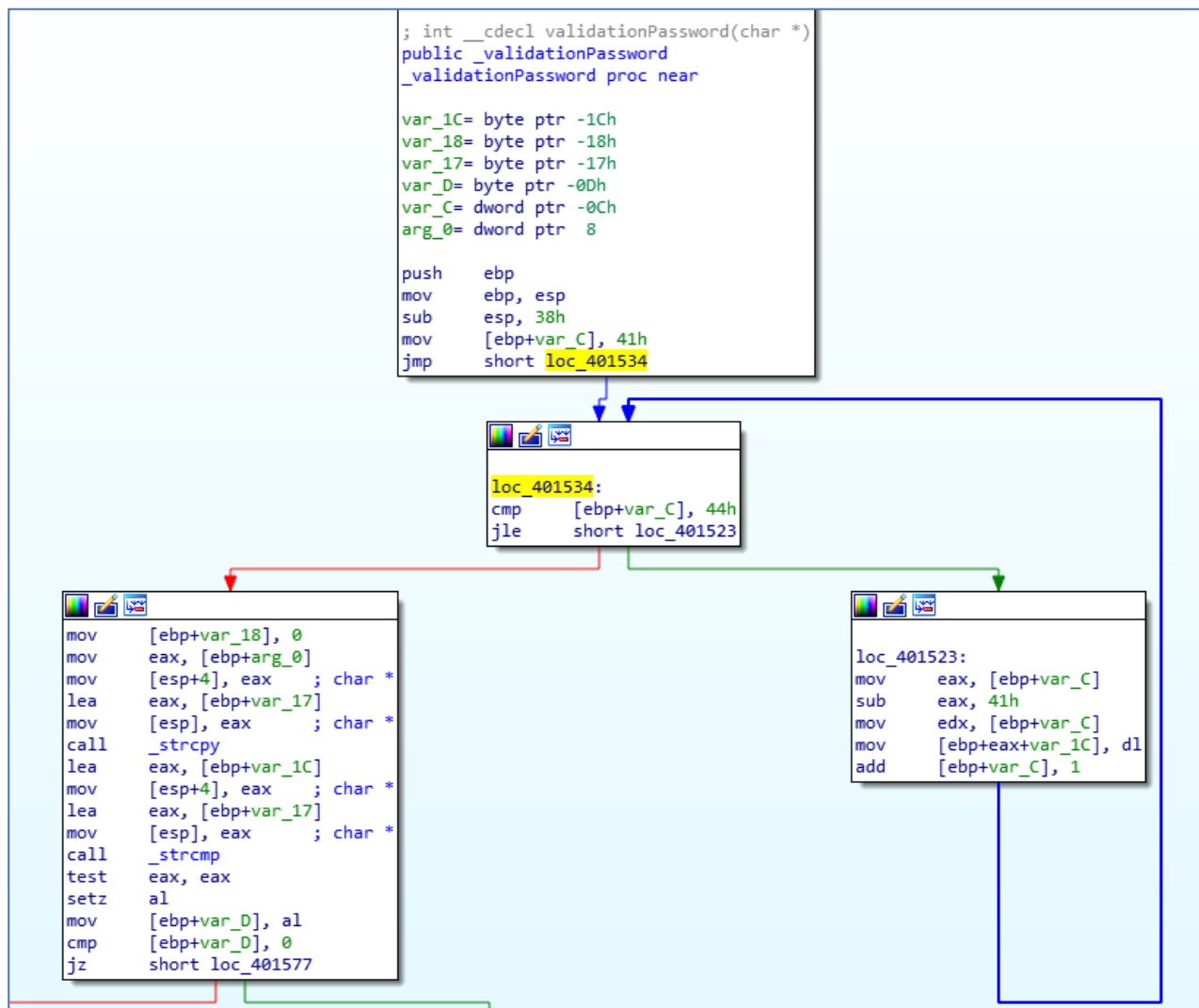
- BYTE: El tamaño del operando es de un byte (8 bits).
- WORD: El operando es de una palabra (word) o dos bytes (16 bits).
- DWORD: El operando es de una palabra doble (double word) o cuatro bytes (32 bits).
- QWORD: El operando es de una palabra cuádruple (quad word) u ocho bytes (64 bits).

10. Bucles

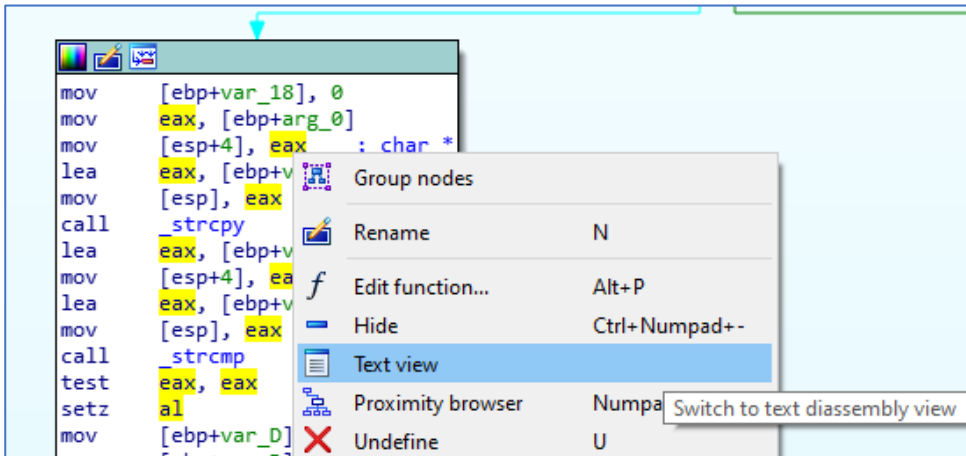
En primer lugar, debemos tener en cuenta que IDA asigna nombres a las variables en función de su ubicación en relación con la dirección de retorno. Las variables locales se encuentran por encima de la dirección de retorno, mientras que los parámetros de la función se encuentran debajo de la dirección de retorno. Los nombres de las variables locales se derivan usando el prefijo `var_` unido con un sufijo hexadecimal, eso indica la distancia, en bytes, que la variable se encuentra por encima del puntero. Por ejemplo, la variable local `var_C`, en este caso, es una variable de 4 bytes (dword) que se encuentra -12 bytes (-0Ch tiene el valor -12 en decimal) por debajo del puntero (`[ebp-0Ch]`). Los nombres de los parámetros de las funciones se generan utilizando el prefijo `arg_` combinado con un sufijo hexadecimal que representa la distancia relativa desde el parámetro superior.

A continuación, localizaremos o deduciremos el código siguiente:

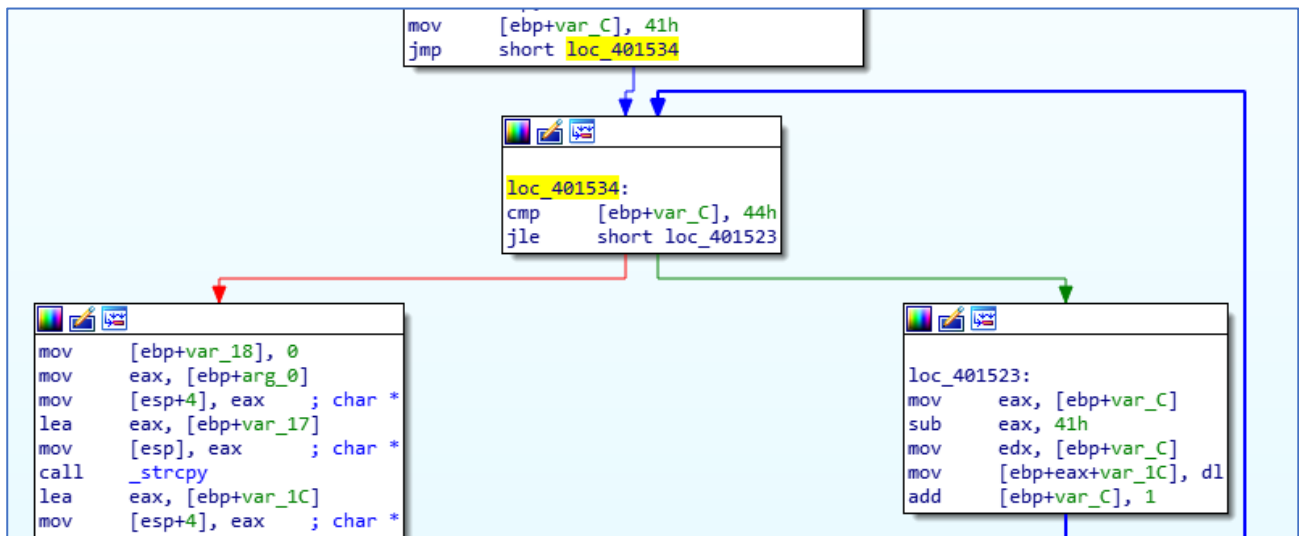
```
// Codigos ASCII de ABCD
for(i = 65; i<69; i++){
    pass[i-65] = i;
}
pass[4]=0; //Codigo ASCII 0
```



Par observar el código en modo texto en lugar de modo gráfico,



A continuación, realizamos *reversing* de un bucle:



En primer lugar, se almacena (mov) el valor 65 (41h en hexadecimal), y se dirige a la etiqueta loc_401534, luego se compara (cmp) el valor 65 con el valor 68 (44h en hexadecimal), se analiza el resultado con jle, compara si es menor o igual, en este caso cabe observar que el compilador ha convertido $i < 69$ en $i \leq 68$, en caso que se cumpla se dirige a la etiqueta loc_401523.

El resto del código se puede deducir, por ejemplo, nuestra variable i es $[ebp+var_C]$, por ejemplo, $i++$ corresponde a: `add [ebp+var_C], 1` a continuación de esta línea vuelve al principio del bucle loc_401537.

El código `sub eax, 41h` corresponde a la operación $i - 65$ cuyo resultado se almacena en el registro `eax` y el código `mov [ebp+eax+var_1C], dl` equivale a `pass[i-65] = i`; ya que en $[ebp+eax+var_1C]$ está almacenado el resultado de $i - 65$.

Una vez finalizado el bucle continua en `mov [ebp+var_18], 0` que corresponde a nuestro código `pass[4] = 0`.

Analizando el código anterior se puede deducir que se trata de un bucle realizado con la instrucción `for` ya que como se ha podido observar se ha seguido la estructura siguiente:

`mov` Asignar valor inicial a un contador

etiqueta:

`jle` Condición if, comprueba si el valor inicial es inferior a un valor máximo en caso contrario se sale del bucle

instrucciones de proceso

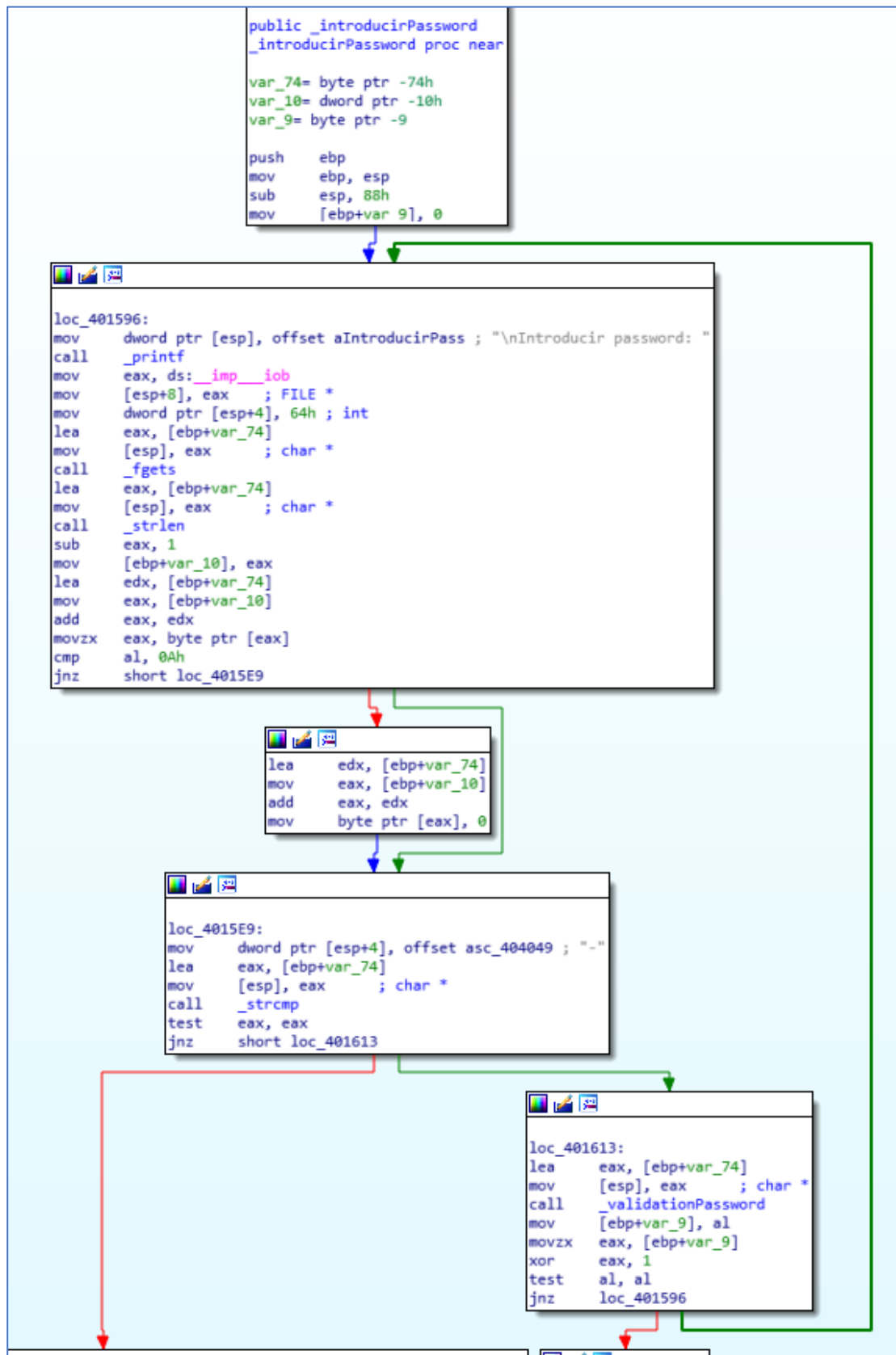
`add` suma 1 al contador

ir a la etiqueta

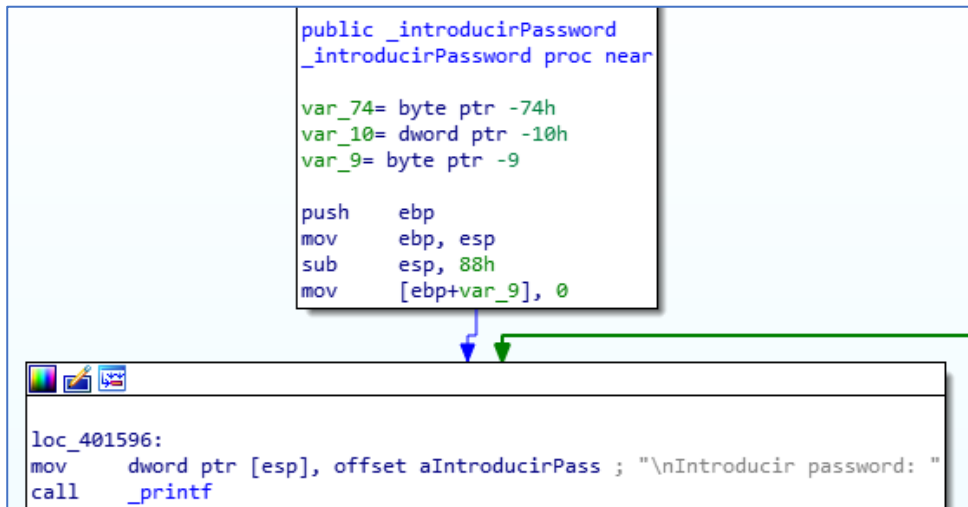
Aunque también podría haberse realizado con la instrucción `while`, el condicional de la instrucción `while` no tiene por qué estar relacionada con la de un valor de un contador, sino que puede ser en función del resultado de cualquier condicional

A continuación, analizaremos el bucle `do while` que se encuentra en la función *introducirPassword*.

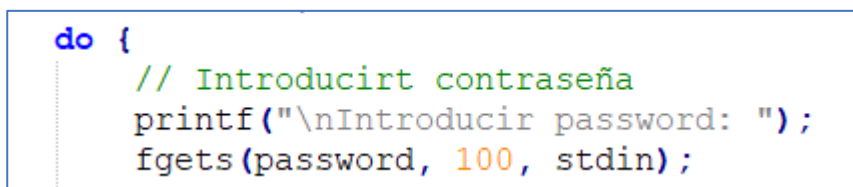
La característica principal de la instrucción `do while` es que como mínimo el bucle se ejecuta una vez y que el condicional se encuentra al final del bucle en lugar de al inicio.



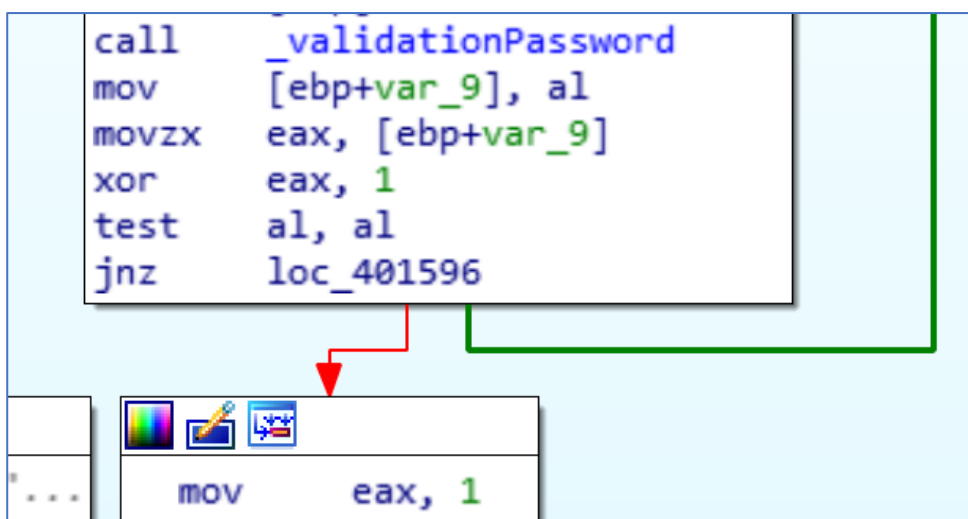
En la función *introducirPassword* el bucle empieza en `loc_401596` y como se puede observar no hay ningún condicional al inicio del bucle.



En la siguiente imagen se puede realizar la comparación con el código fuente



En la parte final del bucle podemos observar el condicional JNZ que en realidad comprueba el valor del resultado del retorno de la función *validationPassword*, tampoco existe un contador sobre el cual haya ninguna condición de seguir en el bucle, por lo tanto podemos deducir que se trata de una instrucción `do while`.



En la siguiente imagen se puede realizar la comparación con el código fuente

```

    }
    success = validationPassword(password);
} while(!success);

```

11. Condicionales

Para poder realizar reversing de los condicionales, en primer lugar, repasaremos los conceptos básicos.

Registro de estado (FLAGS)

FLAGS es un registro que contiene los diferentes indicadores o banderas referentes al estado del proceso, es decir, contiene información sobre el estado del procesador e información sobre el resultado de la ejecución de las instrucciones. A continuación, se muestra la lista de flags

- OF: Overflow flag (bit de desbordamiento)
- TF: Trap flag (bit de excepción)
- AF: Aux carry (bit de transporte auxiliar)
- DF: Direction flag (bit de dirección)
- SF: Sign flag (bit de signo)
- PF: Parity flag (bit de paridad)
- IF: Interrupt flag (bit de interrupción)
- ZF: Cero flag (bit de cero)
- CF: Carry flag (bit de transporte)

En la herramienta IDA los valores de los flags los podemos encontrar en la ventana de *General registers*

General registers			
RAX	0000000000000000	↳	ID 0
RBX	0000000000000001	↳	VIP 0
RCX	000000005E561258	↳	VIF 0
RDX	0000000000000000	↳	AC 0
RSI	00000000006916E0	↳ debug027:006916E0	VM 0
RDI	000000000000001B	↳	RF 0
RBP	000000000061FE78	↳ debug010:0061FE78	NT 0
RSP	000000000061FDEC	↳ debug010:0061FDEC	IOPL 0
RIP	0000000000401588	↳ _validationPassword+74	OF 0
R8	742424448B020040	↳	DF 0
R9	020040301405C70A	↳	IF 1
R10	117402F883000000	↳	TF 0
R11	18C4833B7401F883	↳	SF 0
R12	0CC25B00000001B8	↳	ZF 1
R13	FB8100407030BB00	↳	AF 0
			PF 1
			CF 0

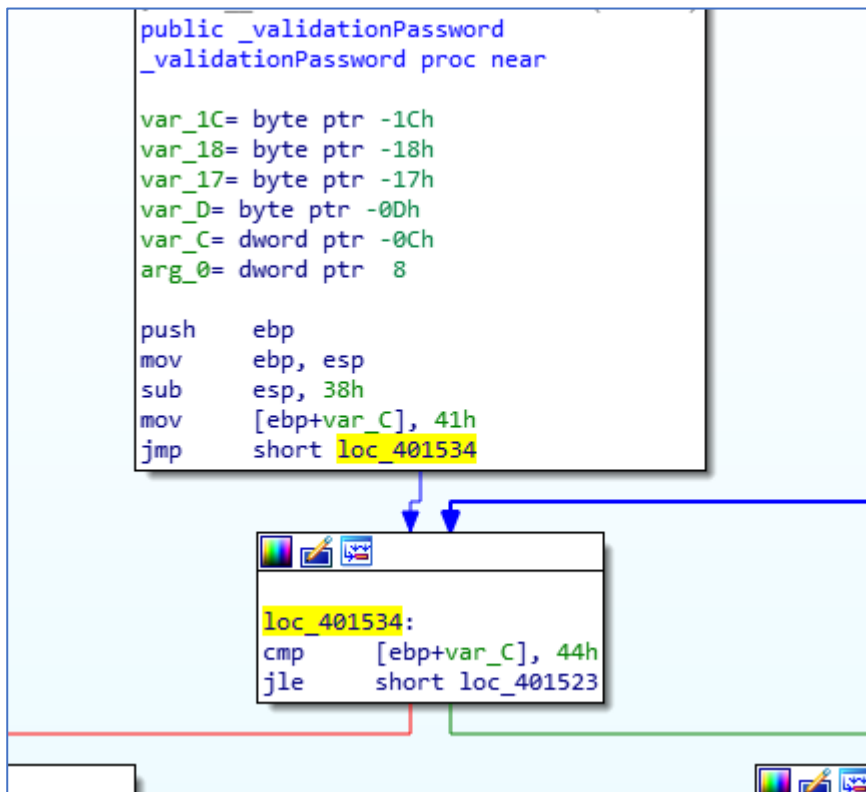
Salto incondicional

- JMP Salta de manera incondicional

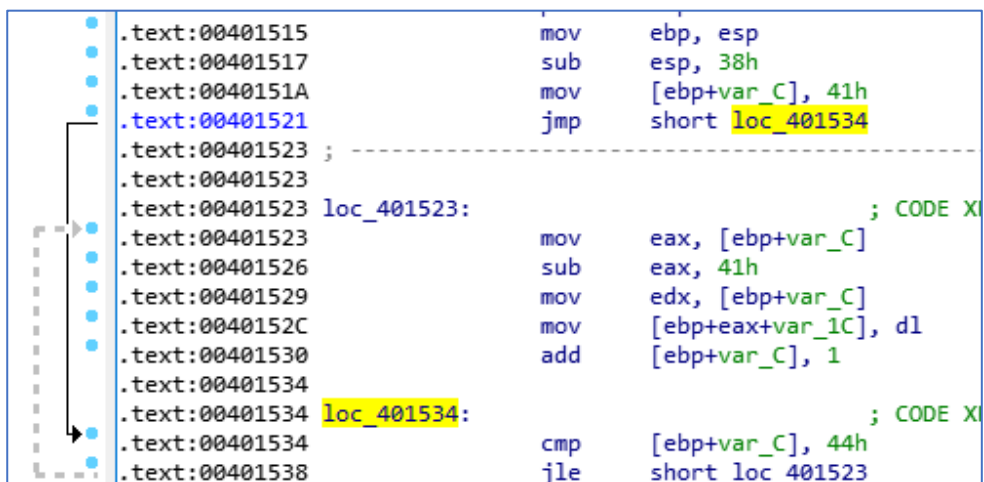
Saltos condicionales, saltos que consultan el valor del flag una vez realizada una operación

- JA: salta si CF=0 i ZF=0 (*jump if above*).
- JAE: salta si CF=0 (*jump if above or equal*).
- JB: salta si CF=1 (*jump if below*).
- JBE: salta si CF=1 o ZF=1 (*jump if below or equal*).
- JC: salta si hi ha ròssec CF=1 (*jump if carry*).
- JCXZ: salta si CX=0 (*jump if CX=0*).
- JECXZ: salta si ECX=0 (*jump if ECX=0*).
- JE: salta si ZF=1 (*jump if equal*).
- JG: salta si ZF=0 i SF=OF (*jump if greater*).
- JGE: salta si SF=OF (*jump if greater or equal*).
- JL: salta si SF<>OF (*jump if less*).
- JLE: salta si ZF=1 o SF<>OF (*jump if less or equal*).
- JNA: salta si CF=1 o ZF=1 (*jump if not above*).
- JNAE: salta si CF=1 (*jump if not above or equal*).
- JNB: salta si CF=0 (*jump if not below*).
- JNBE: salta si CF=0 i ZF=0 (*jump if not below or equal*).
- JNC: salta si CF=0 (*jump if not carry*).
- JNE: salta si ZF=0 (*jump if not equal*).
- JNG: salta si ZF=1 o SF<>OF (*jump if not greater*).
- JNGE: salta si SF<>OF (*jump if not greater or equal*).
- JNL: salta si SF=OF (*jump if not less*).
- JNLE: salta si ZF=0 i SF=OF (*jump if not less or equal*).
- JNO: salta si OF=0 (*jump if not overflow*).
- JNP: salta si PF=0 (*jump if not parity*).
- JNS: salta si SF=0 (*jump if not sign*).
- JNZ: salta si ZF=0 (*jump if not zero*).
- JO: salta si OF=1 (*jump if overflow*).
- JP: salta si PF=1 (*jump if parity*).
- JPE: salta si PF=1 (*jump if parity even*).
- JPO: salta si PF=0 (*jump if parity odd*).
- JS: salta si SF=1 (*jump if sign*).
- JZ: salta si ZF=1 (*jump if zero*).

En la siguiente imagen se puede observar un ejemplo de salto incondicional:

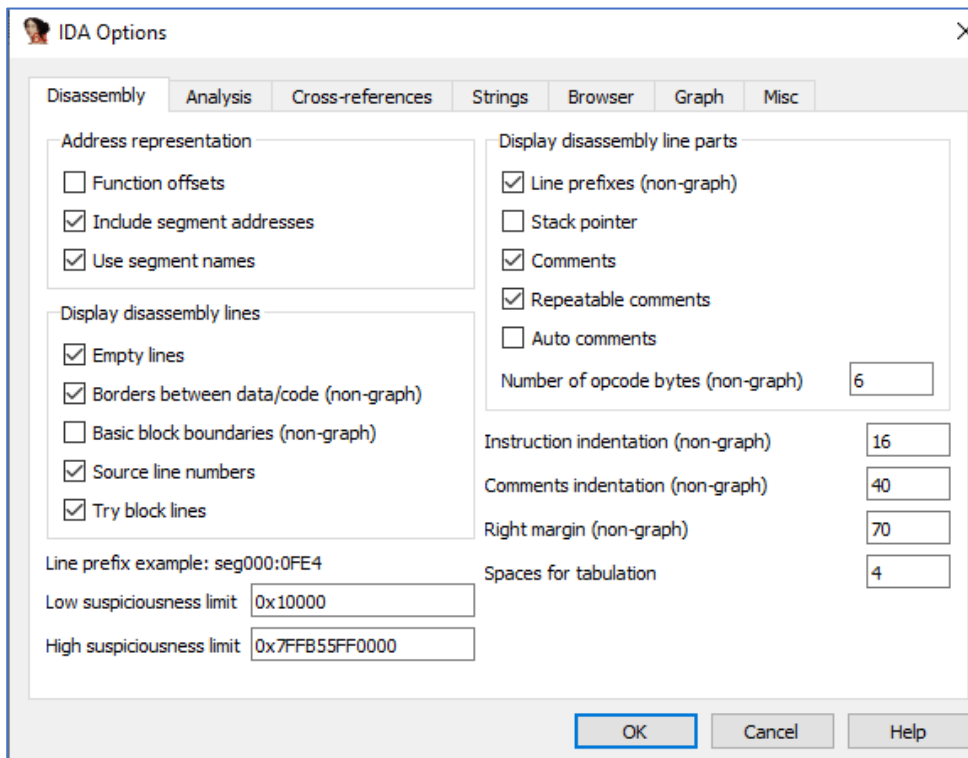


En la siguiente imagen se muestra en modo texto, se puede observar que el salto es debido a la existencia de un bucle y por lo tanto se ha debido organizar el código y las etiquetas para poderse implementar

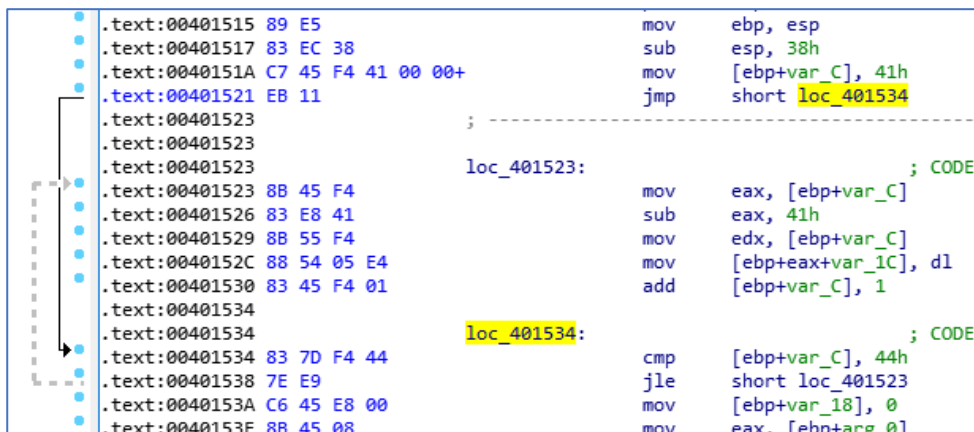


Para mostrar los opcodes se puede activar a través de la ventana de opciones, en este caso indicamos:

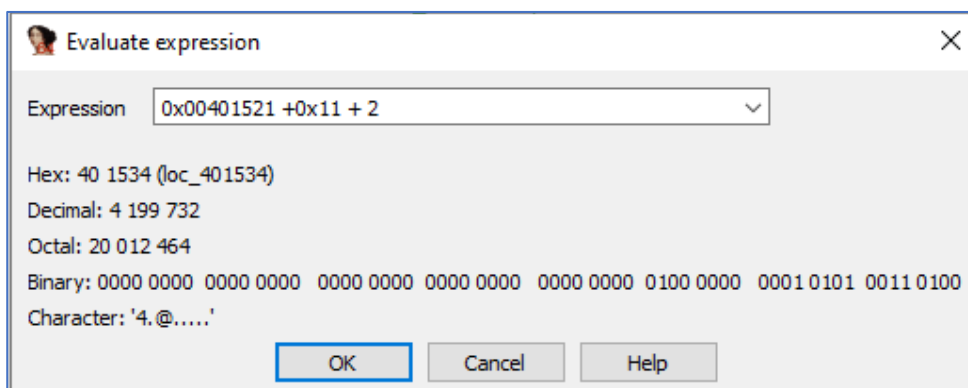
Number of opcode bytes (non-graph): 6



En la siguiente imagen se muestra el código con los opcodes



Podemos observar el opcode EB que corresponde a jmp y que saltará 11 posiciones (11 hexadecimal es decir 17 en decimal) más 2 bytes que ocupa la instrucción, esta operación nos da el resultado de 0x401534 que es la dirección a la cual se dirigirá.

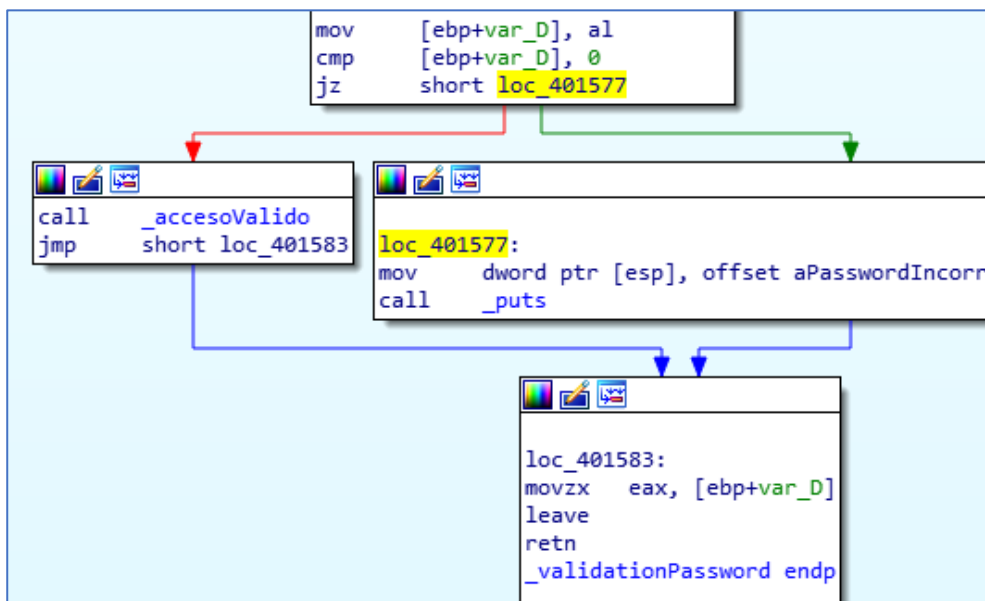


En la siguiente imagen se muestra el bucle a la vez que se puede observar el salto condicional `jle` que redirige el flujo de la ejecución en función del resultado de la comparación entre dos valores realizada por la instrucción `cmp`

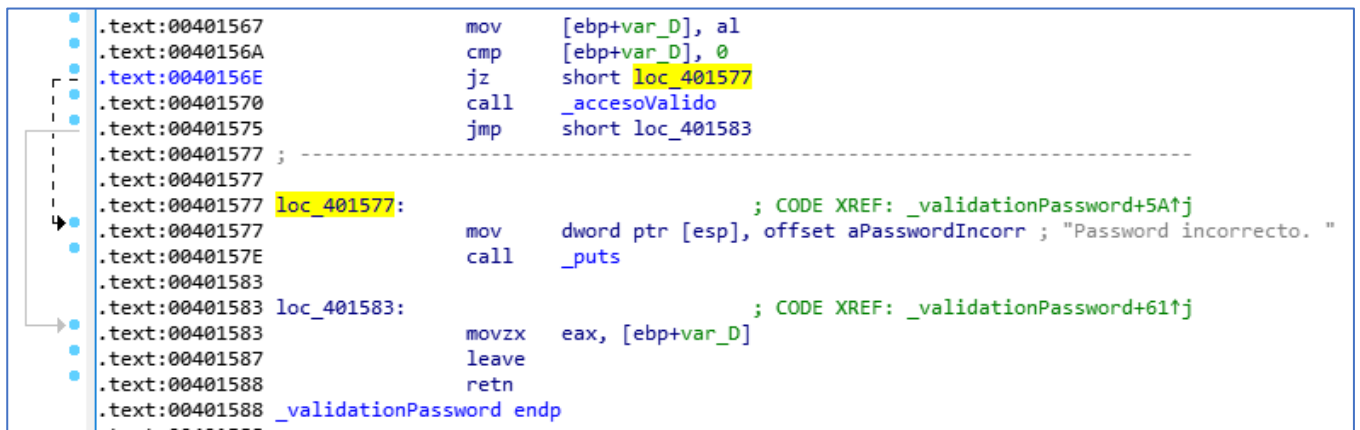


En la siguiente imagen se pueden observar el salto condicional `jz` y el salto incondicional `jmp`.

Cabe destacar que en el modo gráfico de la herramienta IDA, en los saltos condicionales muestra en color verde la línea del flujo en caso de que se cumpla la condición y en rojo en caso contrario

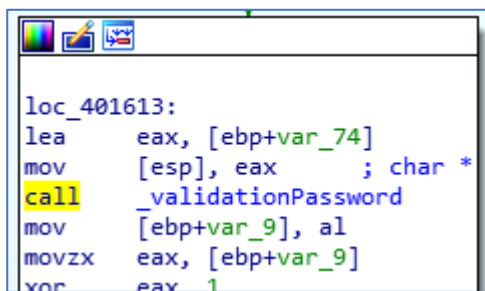


En la imagen siguiente se muestra en modo texto

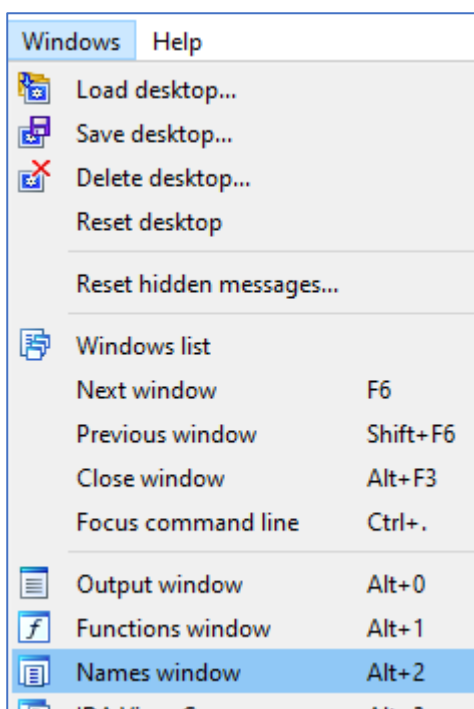


12. Subrutinas

Para hacer la llamada a la subrutina se utiliza la instrucción `call` seguido del nombre de la etiqueta que define el punto de entrada a la subrutina, por ejemplo:
`call _validationPassword`

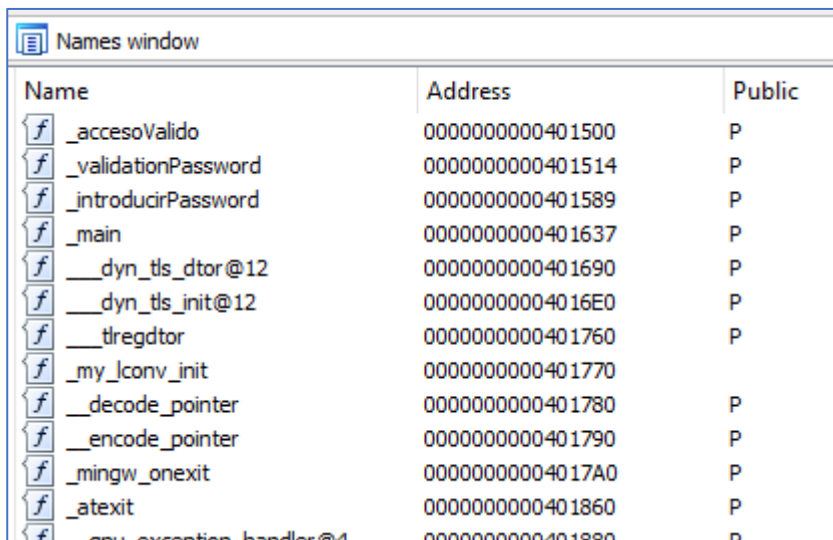















Para conocer la relación entre los nombres de las etiquetas y las direcciones de memoria abriremos el panel *Names Window*.



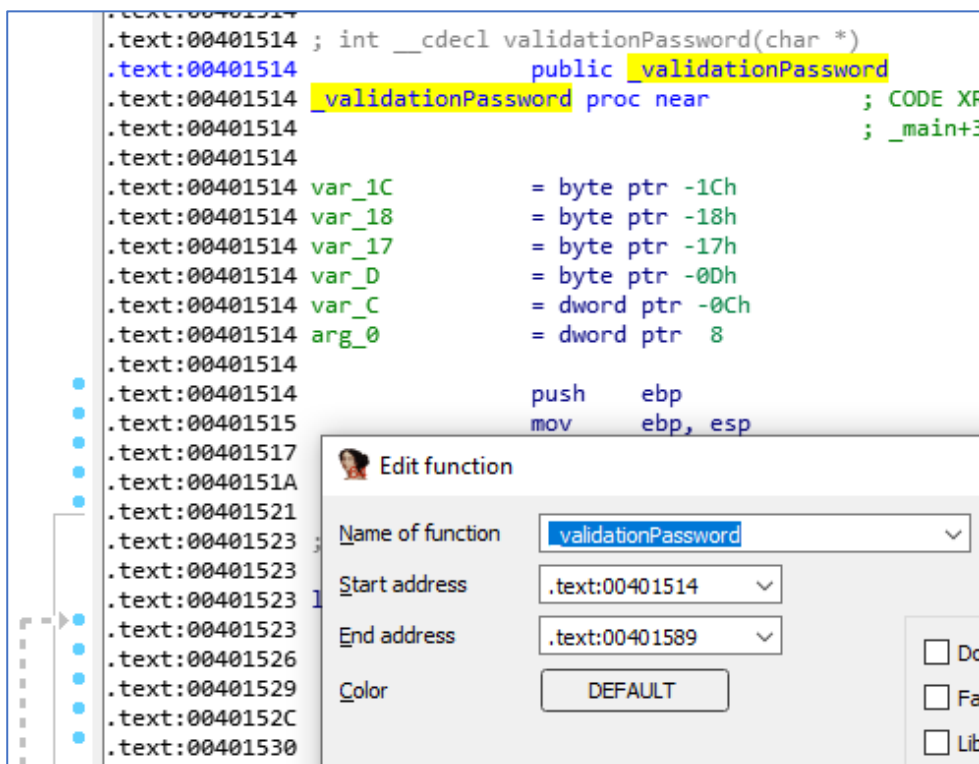
Obteniendo la lista de métodos o funciones obtenemos una valiosa información ya que podemos observar la estructura de funciones y los puntos donde se llaman o acceden a estos métodos ya que podemos localizar las instrucciones `call`.

Panel Names window



Name	Address	Public
 <code>_accesoValido</code>	0000000000401500	P
 <code>_validationPassword</code>	0000000000401514	P
 <code>_introducirPassword</code>	0000000000401589	P
 <code>_main</code>	0000000000401637	P
 <code>__dyn_tls_dtor@12</code>	0000000000401690	P
 <code>__dyn_tls_init@12</code>	00000000004016E0	P
 <code>__tregdtor</code>	0000000000401760	P
 <code>_my_lconv_init</code>	0000000000401770	
 <code>_decode_pointer</code>	0000000000401780	P
 <code>_encode_pointer</code>	0000000000401790	P
 <code>_mingw_onexit</code>	00000000004017A0	P
 <code>_atexit</code>	0000000000401860	P
 <code>__cpu_exception_handler@4</code>	0000000000401880	P

Que además se puede comprobar en la vista del código en modo texto y también en la opción *Edit Function*.



```
.text:00401514 ; int __cdecl validationPassword(char *)
.text:00401514 public validationPassword
.text:00401514 validationPassword proc near ; CODE XREF
.text:00401514 ; _main+3
.text:00401514
.text:00401514 var_1C = byte ptr -1Ch
.text:00401514 var_18 = byte ptr -18h
.text:00401514 var_17 = byte ptr -17h
.text:00401514 var_D = byte ptr -0Dh
.text:00401514 var_C = dword ptr -0Ch
.text:00401514 arg_0 = dword ptr 8
.text:00401514
.text:00401514 push ebp
.text:00401515 mov ebp, esp
.text:00401517
.text:0040151A
.text:00401521
.text:00401523 ;
.text:00401523
.text:00401523
.text:00401526
.text:00401529
.text:0040152C
.text:00401530
```

Edit function

Name of function:

Start address:

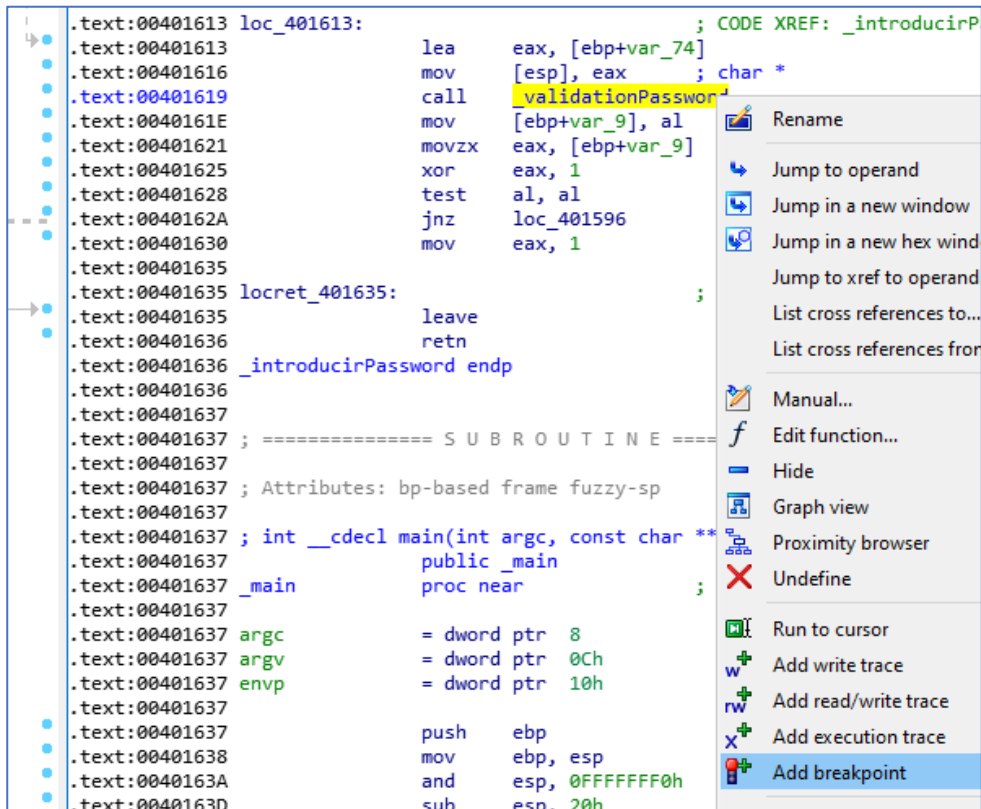
End address:

Color:

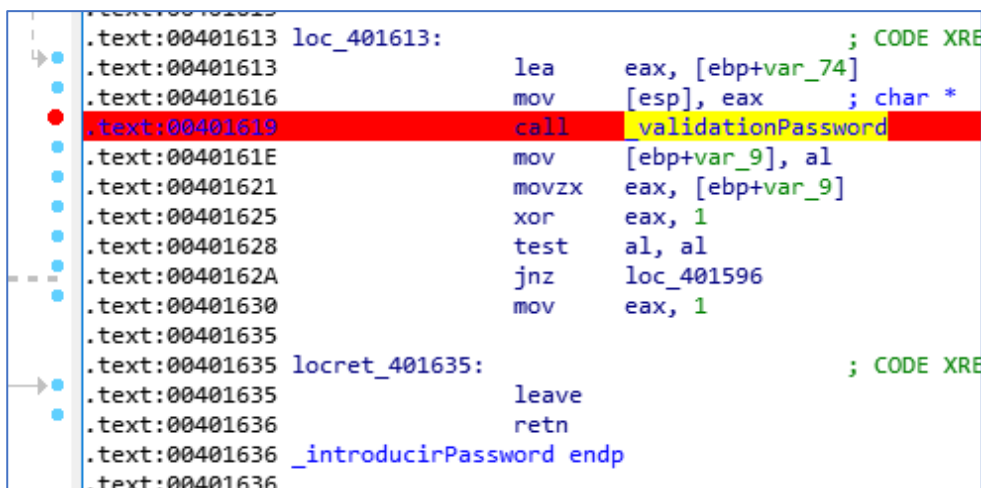
☐ Do
☐ Fa
☐ Lib

La instrucción `call` almacena en la pila la dirección de retorno, es decir, la dirección de memoria donde se encuentra la instrucción siguiente de la instrucción `call`, y a continuación transfiere el control del programa a la subrutina.

Para observar el proceso añadiremos *breakpoints* en el código para que la ejecución del programa realice una pausa en los puntos de interrupción que indiquemos.

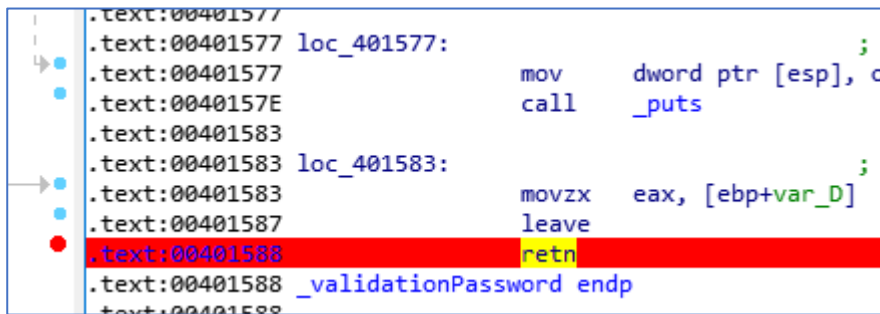


En este caso se ha añadido un punto de interrupción en la llamada a la función *validationPassword* que se encuentra en el método *introducirPassword*.

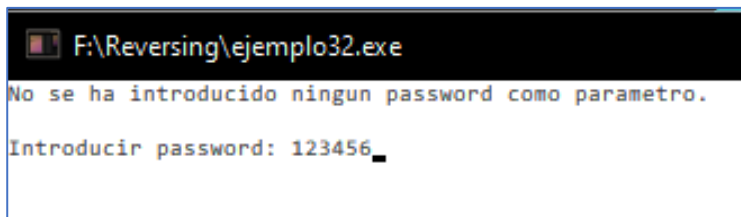


Añadiremos otro punto de interrupción en la instrucción de retorno de la función *validationPassword*.

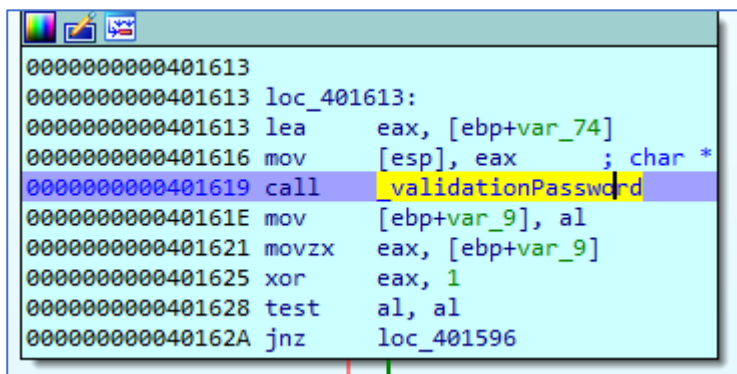
La instrucción `retn` retornará la ejecución del flujo del programa en la línea siguiente de la instrucción `call` que ha llamado a la función, en este caso a la dirección `0040161E`



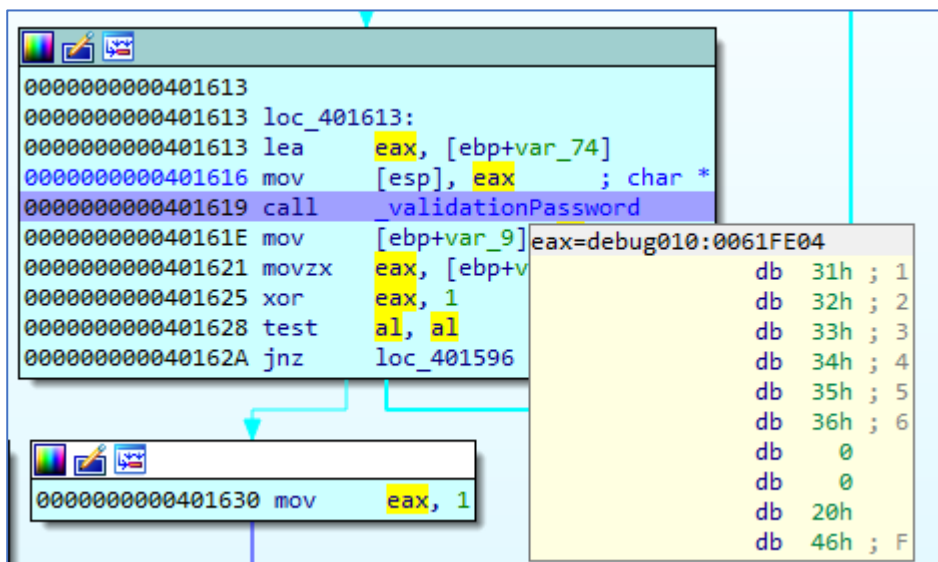
Para observar su funcionamiento ejecutaremos la aplicación e introduciremos como password el valor 123456



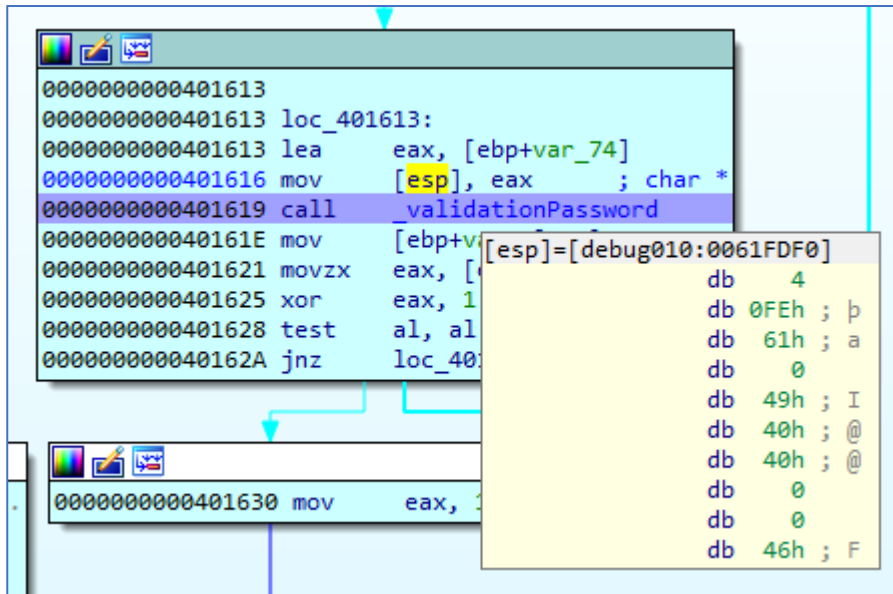
A continuación, el programa se detendrá en el punto de interrupción que hemos indicado



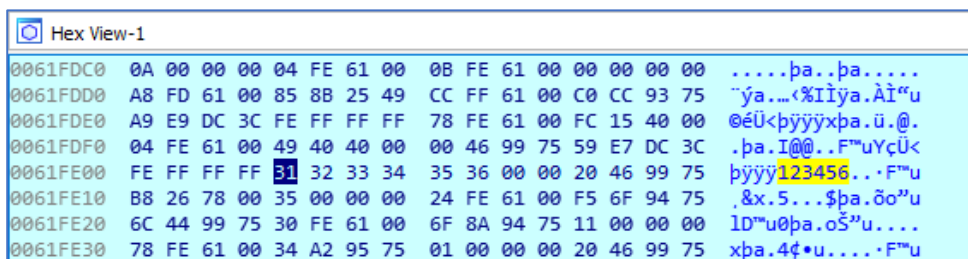
Una vez detenido podemos observar por ejemplo los valores de los registros, en la siguiente imagen observamos el valor del registro `eax`, en el que podemos observar que contiene el valor que se ha introducido por teclado.



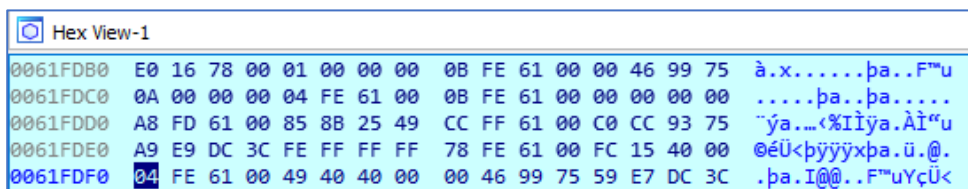
En el registro `esp` se guarda la dirección de memoria en la que se encuentra el valor 123456



Como podemos comprobar en el volcado de memoria, en la posición 0061FE04 y siguientes, tenemos los valores en hexadecimal 31, 32, 33, 34, 35 y 36 que convertidos en decimal equivalen a los valores 49, 50, 51, 52, 53 y 54 que son los valores en ASCII de los números 1, 2, 3, 4, 5 y 6.



También podemos observar que en la dirección 0061FDF0 está el puntero a la dirección 0061FE04



En la ventana de *General registers* se pueden encontrar los valores de los registros en el momento de la pausa de la ejecución en el punto del *breakpoint*

General registers		
RAX	000000000061FE04	debug010:0061FE04
RBX	0000000000000001	
RCX	0000000000404049	.rdata:asc_404049
RDX	000000000061FE04	debug010:0061FE04
RSI	00000000007816E0	debug027:007816E0
RDI	000000000000001B	
RBP	000000000061FE78	debug010:0061FE78
RSP	000000000061FDF0	debug010:0061FDF0
RIP	0000000000401619	_introducirPassword+90
R8	742424448B020040	
R9	020040301405C70A	
R10	117402F883000000	
R11	18C4833B7401F883	


También resulta útil observar los valores de la ventana de la Pila.


Stack view		
0061FDF0	0061FE04	debug010:0061FE04
0061FDF4	00404049	.rdata:asc_404049
0061FDF8	75994600	msvcrt.dll:msvcrt__iob
0061FDFC	3CDCE759	
0061FE00	FFFFFFFFE	
0061FE04	34333231	
0061FE08	00003635	
0061FE0C	75994620	msvcrt.dll:msvcrt__iob+20
0061FE10	007826B8	debug027:007826B8
0061FE14	0000003F	

Como se ha podido observar en una imagen anterior la instrucción siguiente a la llamada a la función *validationPassword* se halla en la dirección 00401619, es decir cuando se ejecute la instrucción *retn* ubicada en la función *validationPassword* el flujo de la ejecución debe seguir en la posición 0040161E, recordemos que esta dirección corresponde a una instrucción de la función *introducirPassword*.

0000000000401613	lea	eax, [ebp+var_74]
0000000000401616	mov	[esp], eax ; char *
0000000000401619	call	validationPassword
000000000040161E	mov	[ebp+var_9], al
0000000000401621	movzx	eax, [ebp+var_9]

Existen dos opciones para seguir con la ejecución del programa paso a paso por cada instrucción.

Step Over , con esta opción el programa ejecutaría la función *validationPassword* pero se detendría en la dirección 0040161E y no podríamos observar los pasos realizados dentro de la función *validationPassword*.

Step into , en este caso el programa se detiene en la primera instrucción de la función *validationPassword*. En este ejemplo de la ejecución del programa se ha

elegido esta opción, de forma que el flujo que ha detenido en la dirección 00401514 ya dentro de la función *validationPassword*

```

0000000000401514
0000000000401514 ; int __cdecl validationPassword(char *)
0000000000401514 public _validationPassword
0000000000401514 _validationPassword proc near
0000000000401514
0000000000401514 var_1C= byte ptr -1Ch
0000000000401514 var_18= byte ptr -18h
0000000000401514 var_17= byte ptr -17h
0000000000401514 var_D= byte ptr -0Dh
0000000000401514 var_C= dword ptr -0Ch
0000000000401514 arg_0= dword ptr 8
0000000000401514
0000000000401514 push    ebp
0000000000401515 mov     ebp, esp
0000000000401517 sub     esp, 38h
000000000040151A mov     [ebp+var_C], 41h
0000000000401521 jmp     short loc_401534

```

Observemos en este punto los valores de la Pila en la que se ha almacenado la dirección de retorno 0040161E donde se encuentra la siguiente instrucción de `call _validationPassword`.

Stack view		
0061FDEC	0040161E	introducirPassword+95
0061FDF0	0061FE04	debug010:0061FE04
0061FDF4	00404049	.rdata:asc_404049
0061FDF8	75994600	msvcrt.dll:msvcrt__iob
0061FDFC	3CDCE759	
0061FE00	FFFFFFFE	
0061FE04	34333231	

Y los valores de los registros.

General registers		
RAX	000000000061FE04	debug010:0061FE04
RBX	0000000000000001	
RCX	0000000000404049	.rdata:asc_404049
RDY	000000000061FE04	debug010:0061FE04
RSI	00000000007816E0	debug027:007816E0
RDI	000000000000001B	
RBP	000000000061FE78	debug010:0061FE78
RSP	000000000061FDEC	debug010:0061FDEC
RIP	0000000000401514	_validationPassword
R8	742424448B020040	

A continuación, ejecutamos la siguiente instrucción eligiendo la opción *Step Over*



```

0000000000401514 var_C= dword ptr -0Ch
0000000000401514 arg_0= dword ptr 8
0000000000401514
0000000000401514 push    ebp
0000000000401515 mov     ebp, esp
0000000000401517 sub     esp, 38h

```

Como que se ha ejecutado la instrucción `push ebp`, pues se ha añadido a la Pila el valor de `ebp` como se puede observar en la imagen siguiente.

```

Stack view
0061FDE8  0061FE78  debug010:0061FE78
0061FDEC  0040161E  _introducirPassword+95
0061FDF0  0061FE04  debug010:0061FE04
0061FDF4  00404049  .rdata:asc_404049
0061FDF8  75994600  msvcrt.dll:msvcrt__iob
0061FDFC  3CDCE759
0061FE00  FFFFFFFE
0061FE04  34333231

```



Luego con la opción *Continue Process* la ejecución continua y se detiene en el siguiente *Breakpoint* que hemos indicado.

```

0000000000401583
0000000000401583 loc_401583:
0000000000401583 movzx  eax, [ebp+var_D]
0000000000401587 leave
0000000000401588 retn
0000000000401588 _validationPassword endp
0000000000401588

```

En este punto se puede observar que el valor de ESP en el que se almacena el puntero a la parte superior de la pila, su valor es `0061FDEC`, que es el puntero de la dirección `0040161E` que a su vez es la dirección de retorno, es decir la instrucción siguiente a la llamada `call validationPassword`.

Podemos observar estos valores en las ventanas de Registros y de la Pila siguientes.

```

General registers
RAX 0000000000000000
RBX 0000000000000001
RCX 0000000049258829
RDX 0000000000000000
RSI 00000000007816E0 debug027:007816E0
RDI 000000000000001B
RBP 000000000061FE78 debug010:0061FE78
RSP 000000000061FDEC debug010:0061FDEC
RIP 0000000000401588 _validationPassword+74
R8 742424448B020040

```

Stack view		
0061FDE4	FFFFFFFE	
0061FDE8	0061FE78	debug010:0061FE78
0061FDEC	0040161E	_introducirPassword+95
0061FDF0	0061FE04	debug010:0061FE04

¿Qué ocurriría si en lugar de haber introducido por teclado el valor 123456 se hubiera introducido el siguiente valor?

Es decir, treinta y un caracteres.

```

F:\Reversing\ejemplo32.exe
No se ha introducido ningun password como parametro.

Introducir password: 1234567890123456789012345678901_

```

En este caso al llegar a la instrucción de retorno de la función *validationPassword*

0000000000401583	
0000000000401583	loc_401583:
0000000000401583	movzx eax, [ebp+var_D]
0000000000401587	leave
0000000000401588	retn
0000000000401588	_validationPassword endp
0000000000401588	

En lugar de tener almacenado en la Pila, en la dirección 0061FDEC la dirección de retorno 0040161E.

Stack view		
0061FDC8	0061FE0B	debug010:0061FE0B
0061FDCC	44434241	
0061FDD0	33323100	
0061FDD4	00363534	
0061FDD8	0061FFCC	debug010:0061FFCC
0061FDDC	00000045	
0061FDE0	93DC57FA	
0061FDE4	FFFFFFFE	
0061FDE8	0061FE78	debug010:0061FE78
0061FDEC	0040161E	_introducirPassword+95
0061FDF0	0061FE04	debug010:0061FE04

Pues en este caso los valores serían los siguientes

Stack view		
0061FDC8	0061FE24	debug010:0061FE24
0061FDCC	44434241	
0061FDD0	33323100	
0061FDD4	37363534	
0061FDD8	00303938	
0061FDDC	35343332	
0061FDE0	39383736	
0061FDE4	33323130	
0061FDE8	37363534	
0061FDEC	31303938	
0061FDF0	0061FE00	debug010:0061FE00
0061FDF4	00404049	.rdata:asc_404049
0061FDF8	75994600	msvcrt.dll:msvcrt__iob

Es decir, se ha sobrescrito en la Pila el valor de la dirección 0061FDEC con el valor 31303938 en lugar de la dirección real de retorno 0040161E.

Recordemos que los valores en hexadecimal 31 30 39 38 corresponden a los valores en decimal 49 48 57 56 que son los valores en ASCII de 1098, es decir los últimos 4 caracteres de los 31 que se han introducido por teclado, pero en sentido inverso.

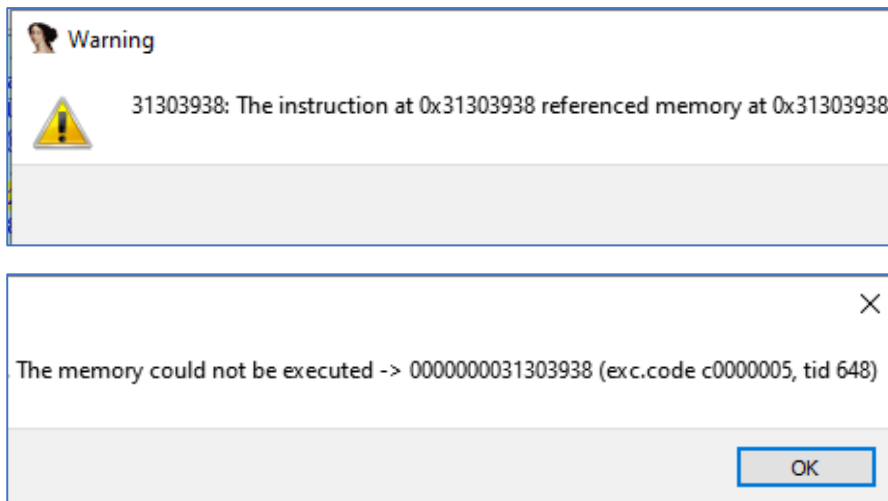
Observemos también los valores de los registros, el valor de `ebp` o `rsp` continúa siendo la dirección del puntero prevista donde se debía almacenar la dirección de retorno

General registers	
RAX	0000000000000000
RBX	0000000000000001
RCX	00000000DDFB50D
RDX	0000000000000000
RSI	00000000006516E0 debug027:006516E0
RDI	000000000000001B
RBP	0000000037363534
RSP	000000000061FDEC debug010:0061FDEC
RIP	0000000000401588 _validationPassword+74
R8	742424448B020040
R9	020040301405C70A

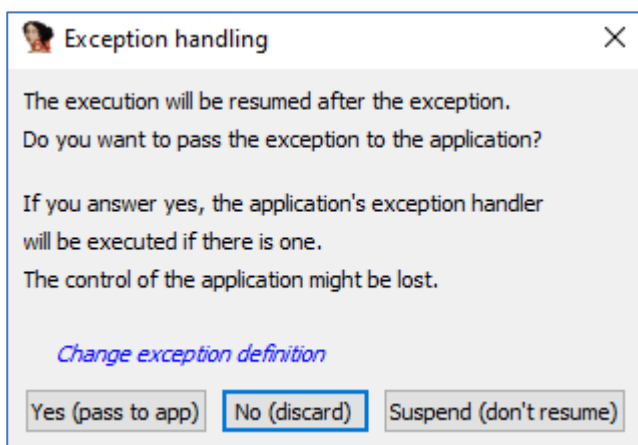
Observemos también que en el volcado de memoria se han sobrepasado los 10 caracteres previstos y definidos en el código fuente, `char buffer[10];`

Hex View-1	
0061FD90	74 FD 61 00 10 00 00 00 CC FF 61 00 C0 CC 20 75 tya....Iya.Ài·u
0061FDA0	6D D5 9B 78 FE FF FF FF E8 FD 61 00 83 15 40 00 m0>xpÿÿÿÿÿa.f.@.
0061FDB0	1C 40 40 00 CC FD 61 00 24 FE 61 00 00 46 26 75 .@.Iya.\$ba..F&u
0061FDC0	0A 00 00 00 04 FE 61 00 24 FE 61 00 41 42 43 44ba.\$ba.ABCD
0061FDD0	00 31 32 33 34 35 36 37 38 39 30 00 32 33 34 35 .1234567890.2345
0061FDE0	36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 6789012345678901
0061FDF0	00 FE 61 00 49 40 40 00 00 46 26 75 7D DA 9B 78 .ba.I@..F&u}Ú>x
0061FE00	FE FF FF FF 31 32 33 34 35 36 37 38 39 30 31 32 pÿÿÿÿ123456789012
0061FE10	33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 3456789012345678
0061FE20	30 30 31 00 00 FF 61 00 65 8A 31 75 11 00 00 00 001 ba·ēlu

A continuación, al ejecutarse la instrucción `retn` de la función `validationPassword`, pues en lugar de recoger el valor de retorno `0040161E`, ha recogido como valor de retorno `31303938` que obviamente no es correcta, por lo que la ejecución del programa lanza el siguiente error.



Y como consecuencia IDA muestra también el mensaje siguiente



¿Porqué ha ocurrido la sobreescritura del valor de retorno en la Pila?

Esto ha sido debido al uso de instrucciones no seguras como `strcpy` o `strcmp` entre otras.

En el código fuente de este ejemplo, en la función `validationPassword`, en primer lugar, se ha definido:

```
char buffer[10];
```

Y a continuación se ha implementado la instrucción

```
strcpy(buffer, password);
```

La función `strcpy` copia la cadena `password` (incluyendo el carácter nulo) a la

cadena `buffer`, El principal problema de esta función es cuando la cadena origen tiene un tamaño mayor a la cadena destino, en este ejemplo la cadena `buffer`, ya que en este caso se sobrescribirá la memoria fuera del buffer, en este caso de la pilla, pudiendo dar pie a ataques de tipo buffer overflow.

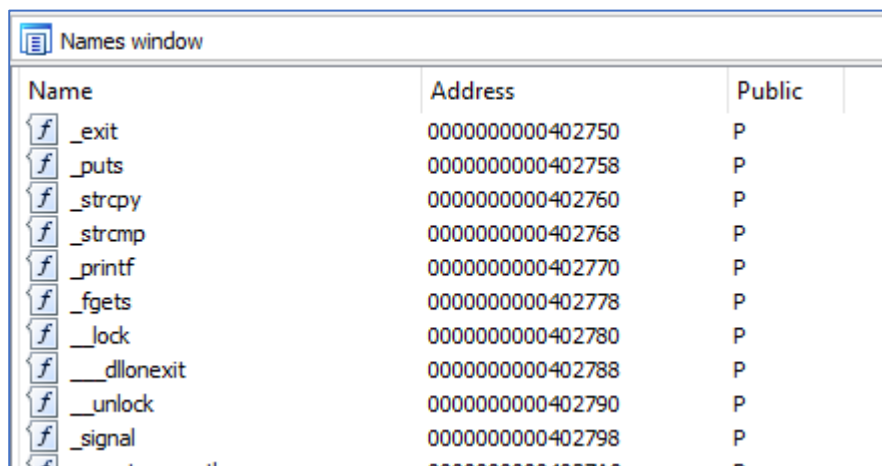
La solución a este tipo de problema es utilizar por ejemplo la función `strncpy`, la cual incluye un parámetro para indicar la cantidad de caracteres máximos que pueden ser copiados. Así pues, en el programa de ejemplo se debería haber codificado de la forma siguiente:











```
strncpy(buffer, password, sizeof(buffer));
```

Una de las posibilidades que nos permite el *Reverse Engineering* es la localización de vulnerabilidades.

Recordemos que existen una gran diversidad de funciones vulnerables, por ejemplo *gets*, *getwd*, *strcpy*, *strcat*, *sprintf*, *scanf*, *sscanf*, *fscanf*, *vfscanf*, *vsprintf*, *vscanf*, *vsscanf*, *streadd*, *strecpy*, *realpath*, *syslog*, *getopt*, *getopt_long*, *getpass*, *realpath*, etc. La mayoría de estas funciones pueden provocar situaciones de **buffer overflow** si los valores de los parámetros que se asignan no son correctos o están sobredimensionados.

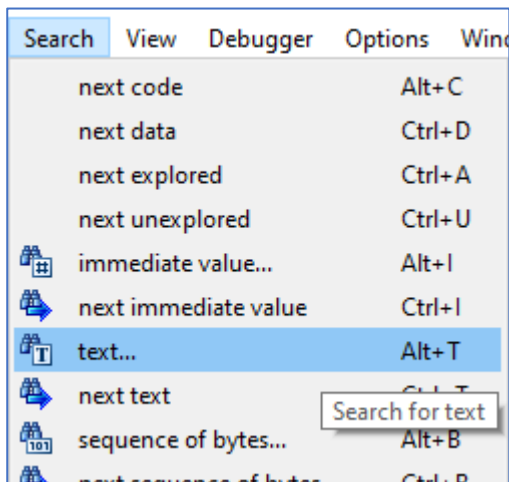
Una de las opciones que disponen los debugadores es localizar funciones, en el caso de IDA podemos comprobar la existencia de funciones vulnerables a través la la ventana *Names window*. En este ejemplo.



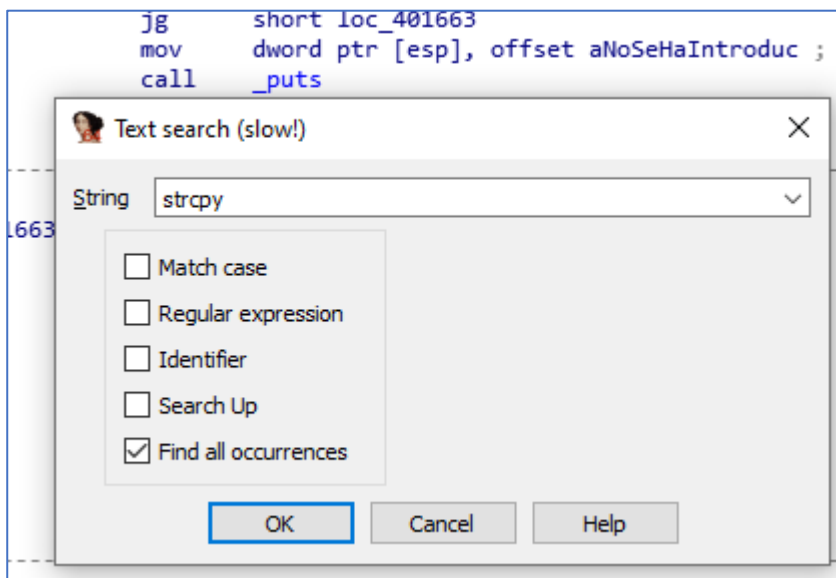
Name	Address	Public
 <code>_exit</code>	0000000000402750	P
 <code>_puts</code>	0000000000402758	P
 <code>_strcpy</code>	0000000000402760	P
 <code>_strcmp</code>	0000000000402768	P
 <code>_printf</code>	0000000000402770	P
 <code>_fgets</code>	0000000000402778	P
 <code>_lock</code>	0000000000402780	P
 <code>__dllonexit</code>	0000000000402788	P
 <code>_unlock</code>	0000000000402790	P
 <code>_signal</code>	0000000000402798	P

A través de esta ventana hemos observado que se utilizan las funciones vulnerables `strcpy` y `strcmp`.

A partir de este punto podemos localizar donde se encuentran las funciones vulnerables a través de la opción *Search/text*



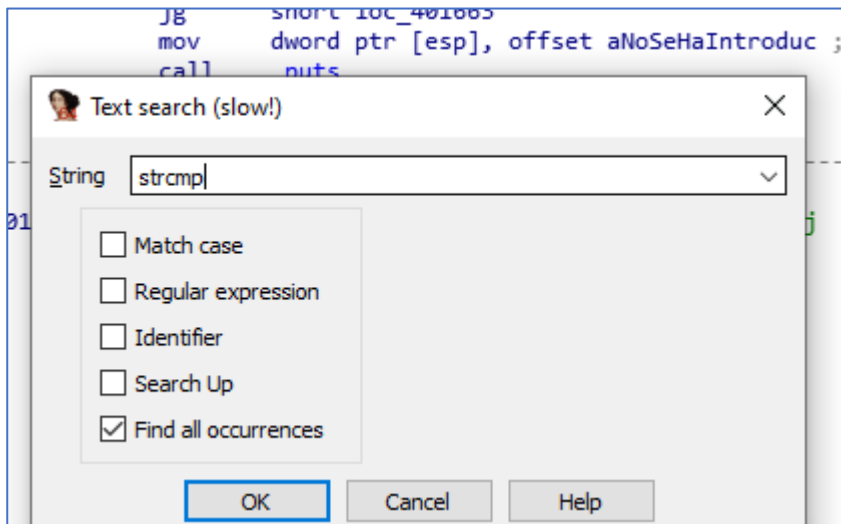
Por ejemplo, en primer lugar, buscaremos donde se encuentra la llamada a la función `strcpy`



Y en este ejemplo obtendremos el resultado, en el que nos indica que se encuentra una llamada a la función `strcpy` en la función `validationPassword`

Occurrences of: strcpy		
Address	Function	Instruction
.text:0040154B	_validationPassword	call _strcpy
.text:00402760	_strcpy	; [00000006 BYTES: COLLAPSED FUNCTION _strcpy.
.idata:004061D0		; char *__cdecl _strcpy(char *, const char *)

Realizando una búsqueda de la función `strcmp`



En el resultado que se muestra nos indica que la llamada a la función `strcmp` se encuentra en las funciones `validationPassword` y en la `introducirPassword`.

Occurrences of: strcmp		
Address	Function	Instruction
.text:0040155D	_validationPassword	call _strcmp
.text:004015F7	_introducirPassword	call _strcmp
.text:00402768	_strcmp	; [00000006 BYTES: COLLAPSED FUNCTION _strcmp. F
.idata:004061CC		; int __cdecl _strcmp(const char *, const char *)

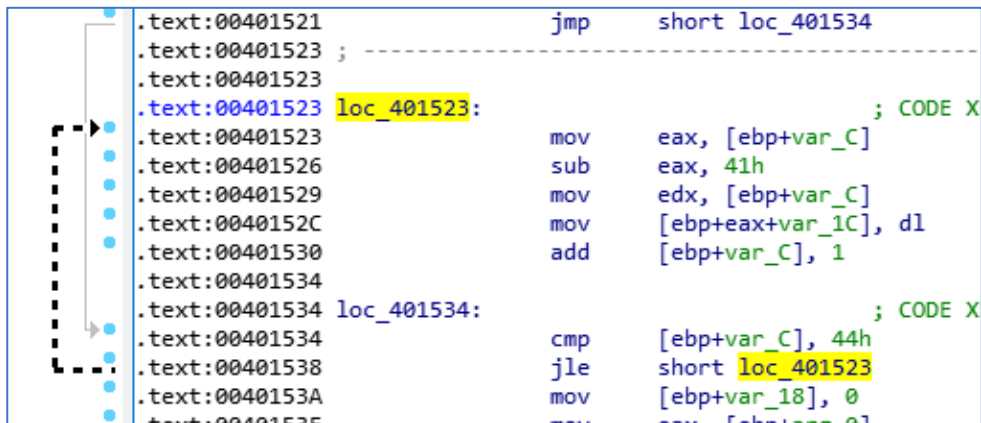
De forma que a través de la ventana del resultado de la búsqueda se puede acceder directamente a la ubicación de la llamada a la función en este caso `strcmp`

.text:0040155D	call	_strcmp
.text:00401562	test	eax, eax
.text:00401564	setz	al
.text:00401567	mov	[ebp+var_D], al
.text:0040156A	cmp	[ebp+var_D], 0
.text:0040156E	jz	short loc_401577
.text:00401570	call	_accesoValido
.text:00401575	jmp	short loc_401583
.text:00401577	; -----	
.text:00401577	loc_401577:	
.text:00401577	mov	dword ptr [esp], offse
.text:0040157E	call	_puts
.text:00401583	loc_401583:	
.text:00401583	movzx	eax, [ebp+var_D]
.text:00401587	leave	
.text:00401588	retn	
.text:00401588	_validationPassword endp	
.text:00401588		

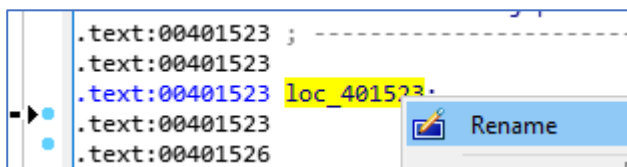
13. Renombrar y personalizar

A medida que realizamos la ingeniería inversa y vamos localizando bucles, secuencias, condicionales, arrays, etc. es conveniente renombrar por ejemplo los nombres de las etiquetas y variables.

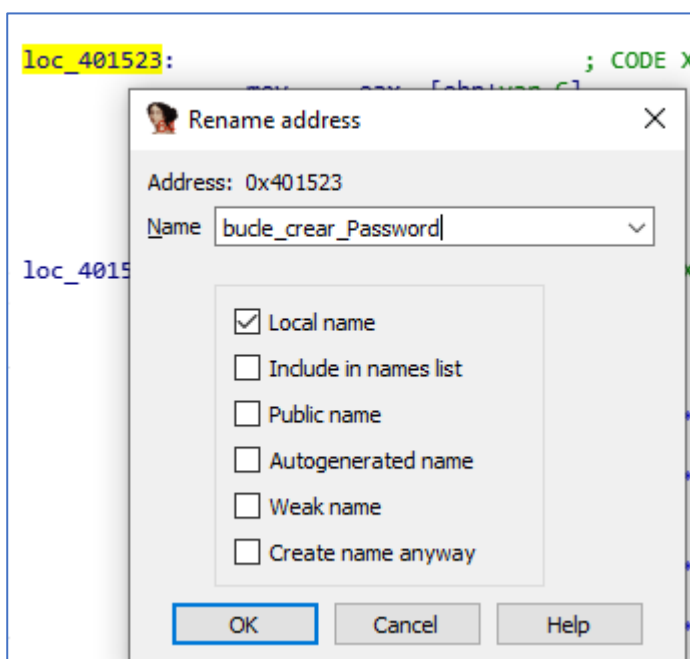
En el siguiente ejemplo hemos localizado un bucle en el que crea el password de entrada a través de un bucle concatenando valores. Así pues, podemos personalizar la etiqueta `loc_401523` para que al revisar el código desensamblado sea más ágil.



Con el botón derecho accedemos a diversas opciones entre las cuales se encuentra *Rename*.



A continuación introducimos el nuevo nombre a la etiqueta



Automáticamente se renombrarán todas las referencias a `loc_401523` por el nuevo nombre `bucle_crear_Password`

```

.text:00401523 ; -----
.text:00401523
.text:00401523 bucle_crear_Password: ; CODE XREF: _va
.text:00401523     mov     eax, [ebp+var_C]
.text:00401526     sub     eax, 41h
.text:00401529     mov     edx, [ebp+var_C]
.text:0040152C     mov     [ebp+eax+var_1C], dl
.text:00401530     add     [ebp+var_C], 1
.text:00401534
.text:00401534 loc_401534: ; CODE XREF: _va
.text:00401534     cmp     [ebp+var_C], 44h
.text:00401538     jle     short bucle_crear_Password
.text:0040153A     mov     [ebp+var_18], 0

```

Este proceso se puede realizar para otros elementos, por ejemplo, las variables, en este caso se ha detectado que `var_17` es un array o el puntero de un array en el que se almacena el valor que ha introducido el usuario por teclado, así pues, renombraremos el nombre de la variable y le pondremos de nombre `buffer`.

```

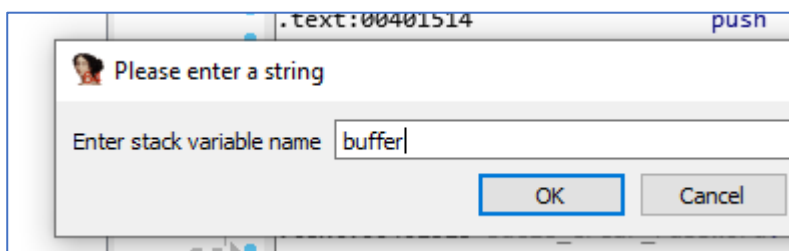
.text:00401514 var_1C      = byte ptr -1Ch
.text:00401514 var_18      = byte ptr -18h
.text:00401514 var_17      = byte ptr -17h
.text:00401514 var_D       = byte ptr -0Dh
.text:00401514 var_C       = dword ptr -0Ch
.text:00401514 arg_0      = dword ptr 8

```

Rename N

Edit function... Rename the current location

Set type... Y



Como se puede observar se ha renombrado el nombre de la variable

```

.text:00401514 ; int __cdecl validationPassword(char *)
.text:00401514 public _validationPassword
.text:00401514 _validationPassword proc near ; C
.text:00401514 ;
.text:00401514
.text:00401514 var_1C      = byte ptr -1Ch
.text:00401514 var_18      = byte ptr -18h
.text:00401514 buffer      = byte ptr -17h
.text:00401514 var_D       = byte ptr -0Dh
.text:00401514 var_C       = dword ptr -0Ch
.text:00401514 arg_0      = dword ptr 8
.text:00401514
.text:00401514 push     ebp
.text:00401515 mov      ebp, esp

```

Así como en todas las referencias a la variable.

```

.text:00401534      cmp     [ebp+var_C], 44h
.text:00401538      jle     short bucle_crear_Passwor
.text:0040153A      mov     [ebp+var_18], 0
.text:0040153E      mov     eax, [ebp+arg_0]
.text:00401541      mov     [esp+4], eax ; char *
.text:00401545      lea     eax, [ebp+buffer]
.text:00401548      mov     [esp], eax ; char *
.text:0040154B      call    _strcpy
.text:00401550      lea     eax, [ebp+var_1C]
.text:00401553      mov     [esp+4], eax ; char *
.text:00401557      lea     eax, [ebp+buffer]
.text:0040155A      mov     [esp], eax ; char *
.text:0040155D      call    _strcmp

```

Así como en la pila de las variables

```

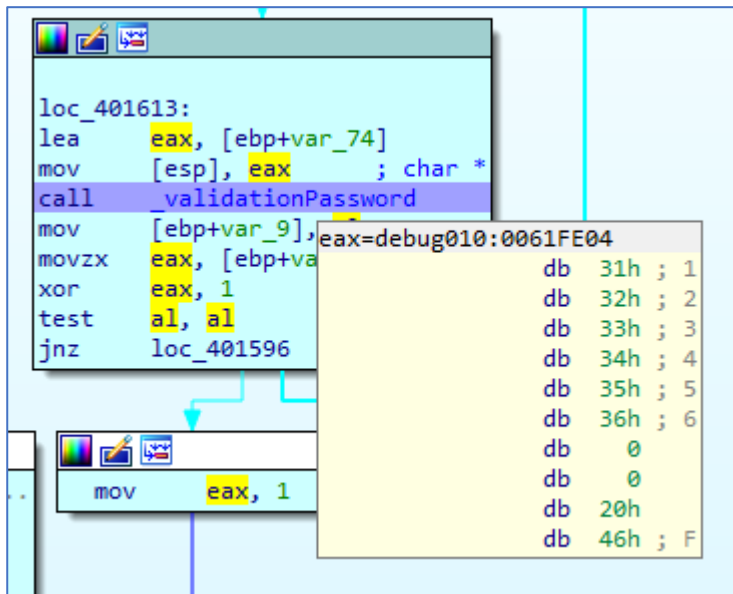
Stack of _validationPassword
IDA View-A
-000000000000001C var_1C      db ?
-000000000000001B      db ? ; undef
-000000000000001A      db ? ; undef
-0000000000000019      db ? ; undef
-0000000000000018 var_18      db ?
-0000000000000017 buffer      db 10 dup(?)
-000000000000000D var_D      db ?
-000000000000000C var_C      dd ?
-0000000000000008      db ? ; undef
-0000000000000007      db ? ; undef
-0000000000000006      db ? ; undef
-0000000000000005      db ? ; undef
-0000000000000004      db ? ; undef
-0000000000000003      db ? ; undef
-0000000000000002      db ? ; undef
-0000000000000001      db ? ; undef
+0000000000000000 s          db 4 dup(?)
+0000000000000004 r          db 4 dup(?)
+0000000000000008 arg_0      dd ?
+000000000000000C      ; end of stack variables

```

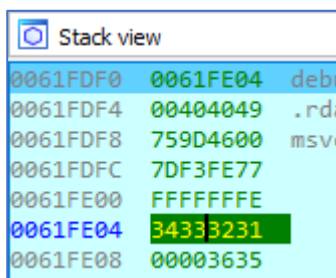
14. Paso de parámetros y flujo de ejecución

Para observar el proceso del paso de parámetros añadiremos un *breakpoint* en el punto de la llamada a la función *validationPassword* que se encuentra en la función *introducirPassword*.

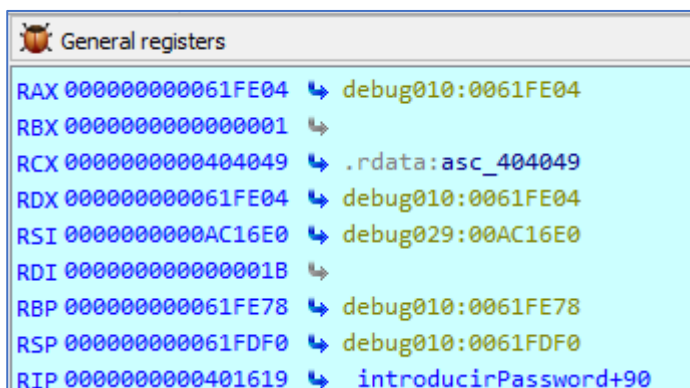
Una vez se ha introducido una contraseña por teclado la ejecución se detiene en el punto de la llamada a la función *validationPassword*.



Como podemos observar el valor 123456 que se ha introducido por teclado se encuentra en la dirección 0061FE04



Cuyo puntero a esta dirección de memoria está almacenado en el registro EAX pero



Y en el ESP tenemos almacenado el puntero a la parte superior de la Pila 0061FDF0 que en este momento contiene el puntero a la dirección 0061FE04.

Stack view		
0061FDF0	0061FE04	debug010:0061FE04
0061FDF4	00404049	.rdata:asc_404049
0061FDF8	759D4600	msvcrt.dll:msvcrt__iob
0061FDFC	7DF3FE77	
0061FE00	FFFFFFFE	



Con la opción Step into ejecutaremos paso a paso las instrucciones de la función *validationPassword*.

public _validationPassword
_validationPassword proc near
var_1C= byte ptr -1Ch
var_18= byte ptr -18h
buffer= byte ptr -17h
var_D= byte ptr -0Dh
var_C= dword ptr -0Ch
arg_0= dword ptr 8
push ebp
mov ebp, esp

Como podemos observar nada más entrar en la función *validationPassword* es que la llamada `call validationPassword` ha almacenado en la pila la dirección de retorno 0040161E a la que se tiene que dirigir la instrucción `retn` al finalizar el flujo en la función *validationPassword*.

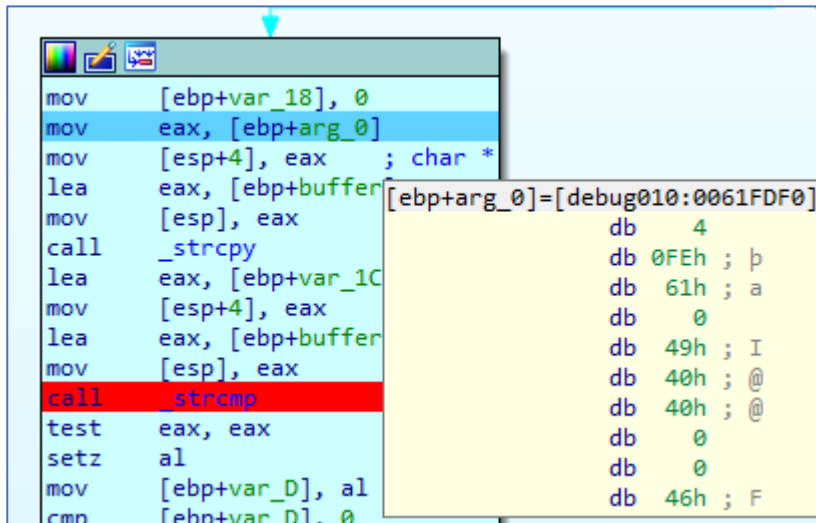
.text:00401616	mov	[esp], eax
.text:00401619	call	validationPassword
.text:0040161E	mov	[ebp+var_9], al

A continuación, podemos observar que la instrucción `push` ha almacenado el valor de `ebp` en la pila para recuperarlo posteriormente ya que la siguiente instrucción `mov` cambiará el valor de `ebp`.

push	ebp	
mov	ebp, esp	
sub	esp, ebp	debug010:0061FE78
mov	[ebp]	db 0A8h ; ~
jmp	short	db 0FEh ; p
		db 61h ; a

Stack view		
0061FDE8	0061FE78	debug010:0061FE78
0061FDEC	0040161E	_introducirPassword+95
0061FDF0	0061FE04	debug010:0061FE04
0061FDF4	00404049	.rdata:asc_404049
0061FDF8	759D4600	msvcrt.dll:msvcrt__iob
0061FDFC	7DF3FE77	
0061FE00	FFFFFFFE	
0061FE04	34333231	

A continuación, el flujo continuara realizando un bucle en la etiqueta renombrada *bucle_crear_Password* que ya hemos analizado en el apartado que hemos tratado los bucles, una vez finalizado el flujo en este bucle la ejecución continua en el siguiente punto



```

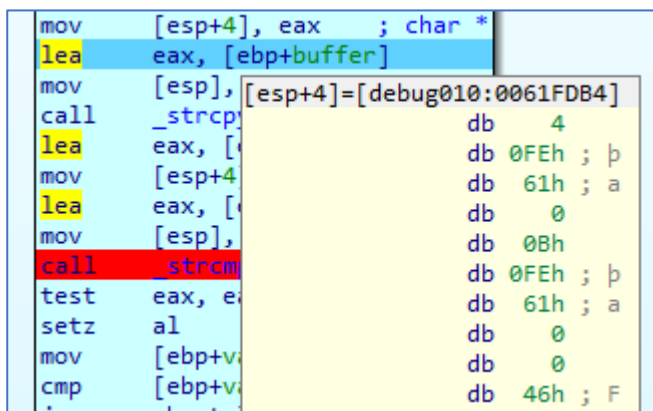
mov     [ebp+var_18], 0
mov     eax, [ebp+arg_0]
mov     [esp+4], eax ; char *
lea     eax, [ebp+buffer]
mov     [esp], eax
call    _strcpy
lea     eax, [ebp+var_1C]
mov     [esp+4], eax
lea     eax, [ebp+buffer]
mov     [esp], eax
call    _strcmp
test    eax, eax
setz    al
mov     [ebp+var_D], al
cmp     [ebp+var_D], 0

```

Memory dump (debug010:0061FDF0):

db	4
db	0FEh ; p
db	61h ; a
db	0
db	49h ; I
db	40h ; @
db	40h ; @
db	0
db	0
db	46h ; F

Donde en la segunda línea asignara al registro *eax* el puntero de la dirección 0061FE04, recordemos que en esta posición esta almacenada el inicio del valor que se ha introducido por teclado 123456 y que se ha pasado como parámetro a la función *validationPassword*.



```

mov     [esp+4], eax ; char *
lea     eax, [ebp+buffer]
mov     [esp], [esp+4]
call    _strcpy
lea     eax, [ebp+buffer]
mov     [esp+4], eax
lea     eax, [ebp+buffer]
mov     [esp], eax
call    _strcmp
test    eax, eax
setz    al
mov     [ebp+var_D], al
cmp     [ebp+var_D], 0

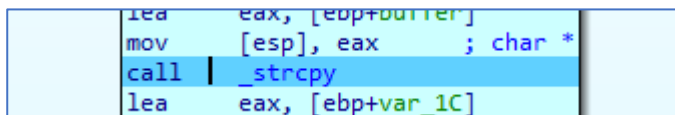
```

Memory dump (debug010:0061FDB4):

db	4
db	0FEh ; p
db	61h ; a
db	0
db	0Bh
db	0FEh ; p
db	61h ; a
db	0
db	0
db	46h ; F

Luego se ha almacenado en *esp+4* el puntero 0061FDB4 que contiene la dirección 0061FE04 con el valor 123456..

De forma que en la llamada *call strcpy* almacenará el valor del parámetro en el array *buffer*.



```

lea     eax, [ebp+buffer]
mov     [esp], eax ; char *
call    _strcpy
lea     eax, [ebp+var_1C]

```

Que corresponde al código fuente

```
strcpy(buffer, password);
```

Una vez ejecutada la instrucción `call strcpy` ya tenemos almacenado el parámetro en el array `buffer`.

<code>mov [esp+4], eax ; char *</code>	
<code>lea eax, [ebp+buffer]</code>	
<code>mov [esp], eax ; char *</code>	
<code>call _strcpy</code>	<code>[ebp+buffer]=[debug010:0061FDD1]</code>
<code>lea eax, [ebp+var_1C]</code>	<code>db 31h ; 1</code>
<code>mov [esp+4], eax ; char *</code>	<code>db 32h ; 2</code>
<code>lea eax, [ebp+buffer]</code>	<code>db 33h ; 3</code>
<code>mov [esp], eax</code>	<code>db 34h ; 4</code>
<code>call _strcmp</code>	<code>db 35h ; 5</code>
<code>test eax, eax</code>	<code>db 36h ; 6</code>
<code>setz al</code>	<code>db 0</code>
<code>mov [ebp+var_D], al</code>	<code>db 0CCh ; ì</code>
<code>cmp [ebp+var_D], al</code>	<code>db 0FFh ; ý</code>
<code>jz short loc_40100000</code>	<code>db 61h ; a</code>

Observemos que en la dirección `ebp+var_1C` esta almacenado el puntero a la dirección `0061FDCC` que es donde se encuentra el valor `ABCD` que ha generado el bucle `bucle_crear_Password`

<code>lea eax, [ebp+buffer]</code>	
<code>mov [esp], eax ; char *</code>	
<code>call _strcpy</code>	
<code>lea eax, [ebp+var_1C]</code>	
<code>mov [esp+4], eax ; char *</code>	
<code>lea eax, [ebp+buffer]</code>	<code>[ebp+var_1C]=[debug010:0061FDCC]</code>
<code>mov [esp], eax</code>	<code>db 41h ; A</code>
<code>call _strcmp</code>	<code>db 42h ; B</code>
<code>test eax, eax</code>	<code>db 43h ; C</code>
<code>setz al</code>	<code>db 44h ; D</code>
<code>mov [ebp+var_D], al</code>	<code>db 0</code>

41 42 43 44 son los valores en hexadecimal de 65 66 67 68 que a su vez son los códigos ASCII de ABCD.

Stack view		
0061FDB0	0061FDD1	debug010:0061FDD1
0061FDB4	0061FE04	debug010:0061FE04
0061FDB8	0061FE0B	debug010:0061FE0B
0061FDBC	759D4600	msvcrt.dll:msvcrt__iob
0061FDC0	0000000A	
0061FDC4	0061FE04	debug010:0061FE04
0061FDC8	0061FE0B	debug010:0061FE0B
0061FDCC	44434241	
0061FDD0	33323100	
0061FDD4	00363534	

En el momento de ejecutar la llamada `call strcmp`, en `esp+4` tenemos el puntero a la dirección `0061FDCC` que es donde se encuentra el valor `ABCD`

```

mov     [esp+4], eax    ; char *
lea     eax, [ebp+buffer]
mov     [esp], [esp+4]  ; debug010:0061FDB4
call    _strcmp
test    eax, eax
setz    al              db 0CCh ; Ì
mov     [ebp+var_4], al  db 0FDh ; ý
cmp     [ebp+var_4], 0   db 61h ; a
jz      short loc_401577 db 0

```

En esp tenemos el puntero a la dirección 0061FDD1 que es el inicio de la dirección donde se encuentra el valor 123456.

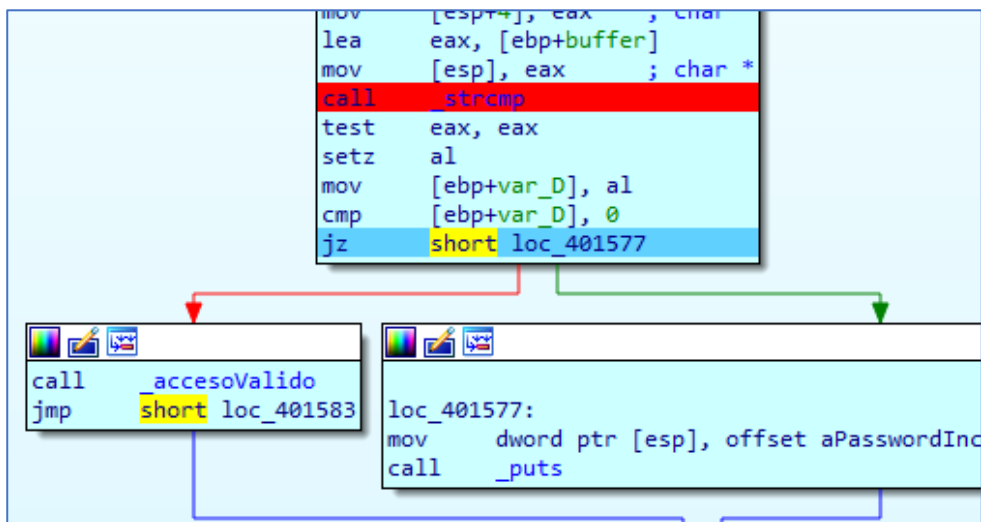
Así pues, la llamada `call strcmp` comparará el valor 123456 con ABCD.

```

mov     [esp+4], eax    ; char *
lea     eax, [ebp+buffer]
mov     [esp], eax      ; char *
call    _strcmp
test    eax, eax
setz    al              [esp]=debug010:0061FDB0
mov     [ebp+var_4], al  db 0D1h ; Ñ
cmp     [ebp+var_4], 0   db 0FDh ; ý
jz      short loc_401577 db 61h ; a

```

Como que los dos valores no son iguales al realizar la comparación el resultado del flag ZF tiene el valor 1 y tal y como se muestra en la siguiente imagen en este caso el flujo se dirigirá a la etiqueta `loc_401577` y no tendrá acceso a la función `accesoValido`.

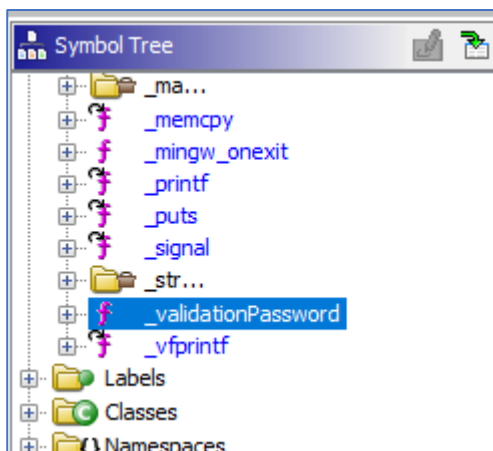


15. Alteración del código.

Algunos debugadores permiten realizar la alteración del código desensamblado cambiando todo tipo de valores y así cambiar el flujo de la aplicación, en nuestro caso realizaremos esta operación como parte del *Reverse Engineering* para comprobar hasta qué punto el programa es vulnerable y puede ser objeto de ataques malintencionados.

Esta operación en este ejemplo la realizaremos con la herramienta *Ghidra*. Una vez abierto el ejecutable, localizaremos la función *validationPassword* de una forma muy parecida a la que se ha realizado con la herramienta IDA.

Una vez localizada la función *validationPassword*



Observando el código desensamblado observamos que la sección en la que se comprueba si el password introducido es correcto es la que se muestra a continuación.

00401557	8d 45 e9	LEA	EAX=>local_1b,[EBP + -0x17]
0040155a	89 04 24	MOV	dword ptr [ESP]=>local_3c,EAX
0040155d	e8 06 12 00 00	CALL	_strcmp
00401562	85 c0	TEST	EAX,EAX
00401564	0f 94 c0	SETZ	AL
00401567	88 45 f3	MOV	byte ptr [EBP + local_11],AL
0040156a	80 7d f3 00	CMP	byte ptr [EBP + local_11],0x0
0040156e	74 07	JZ	LAB_00401577
00401570	e8 8b ff ff ff	CALL	_accesoValido
00401575	eb 0c	JMP	LAB_00401583
LAB_00401577			
00401577	c7 04 24	MOV	dword ptr [ESP]=>local_3c,s_Pas

Comparando con el código fuente

29		// Comprobación de la contraseña
30		bool success = (strcmp(buffer, pass) == 0);
31		if(success) {
32		// la contraseña es correcta
33		accesoValido();
34		}
35		else {
36		// La contraseña es incorrecta
37		printf("Password incorrecto. \n");
38		}
39		

Donde

0040155d	e8 06 12	CALL	_strcmp
----------	----------	------	---------

Corresponde a

bool success = (strcmp(buffer, pass) == 0);

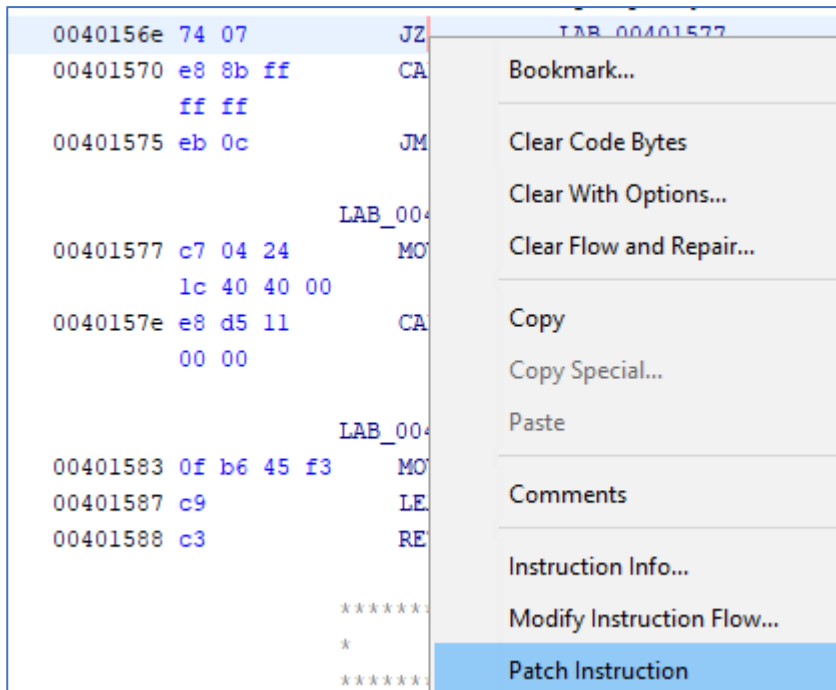
La instrucción JZ analiza el valor del flag en el que se almacena el resultado de la comparación entre el *password* definido en el programa y el *password* introducido por el usuario. En este caso si el valor del flag ZF es 1 el flujo de la ejecución seguirá en la dirección de memoria en la que esta ubicada la etiqueta LAB_00401577, es decir si el password introducido por el usuario es incorrecto el flujo de la ejecución del programa se dirigirá a la dirección 00401577.

0040156e	74 07	JZ	LAB_00401577
----------	-------	----	--------------

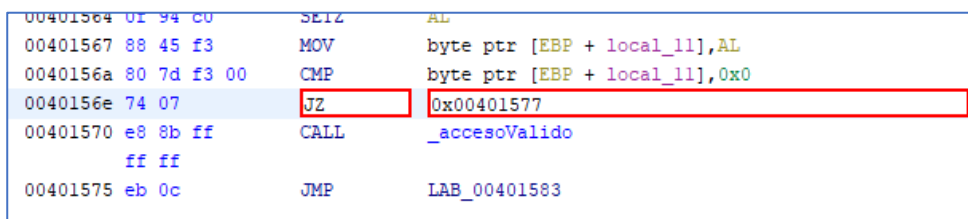
Pero si cambiamos la instrucción JZ por JNZ, en este caso al introducir una contraseña incorrecta seguirá el flujo de la ejecución del programa como si se hubiera introducido una contraseña correcta, ya que la instrucción JNZ comprueba si el valor del flag ZF tiene el valor cero.

Otra opción también posible es en lugar de cambiar la instrucción JZ por JNZ es cambiar la dirección de memoria a la que debe dirigirse el flujo de la ejecución del programa en caso que el flag ZF tenga el valor 1.

Para modificar la instrucción JZ nos situamos en la instrucción y elegimos la opción *Patch Instruction*.



Como se puede observar permite la modificación de la instrucción y la de la dirección de memoria.



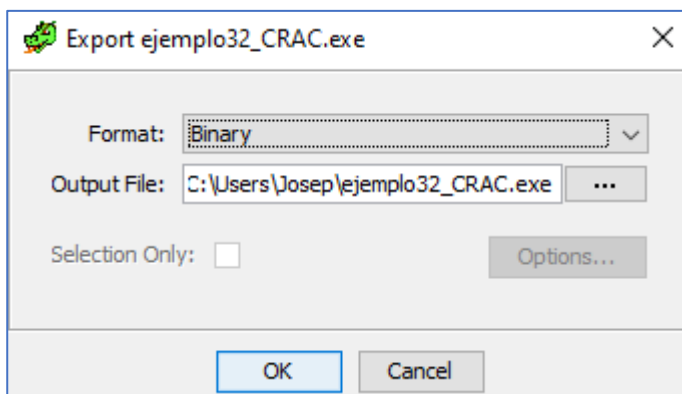
En este ejemplo modificaremos la dirección de memoria de forma que sea el que sea el resultado de la comparación el flujo de la ejecución seguirá en la dirección 00401570 y por lo tanto se accederá a la función *accesoValido*.

00401567	88 45 f3	MOV	byte ptr [EBP + local_11],AL
0040156a	80 7d f3 00	CMP	byte ptr [EBP + local_11],0x0
0040156e	74 07	JZ	0x00401570
00401570	e8 8b ff		74 00
	ff ff		0f 84 fc ff ff ff
00401575	eb 0c		66 0f 84 fd ff

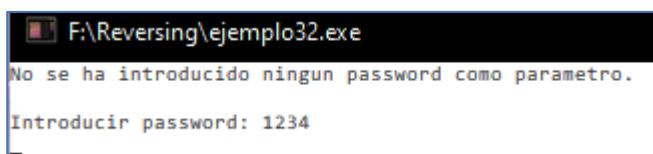
En este caso la herramienta ha añadido también la etiqueta LAB_00401570

00401567	88 45 f3	MOV	byte ptr [EBP + local_11],AL
0040156a	80 7d f3 00	CMP	byte ptr [EBP + local_11],0x0
0040156e	74 00	JZ	LAB_00401570
			LAB_00401570
00401570	e8 8b ff	CALL	_accesoValido
	ff ff		

Una vez realizados los cambios podemos exportar el programa como un programa ejecutable



Comprobamos la ejecución con una contraseña incorrecta



Como se puede observar hemos accedido al programa con una contraseña incorrecta.

