



Universidad de Oviedo



Escuela de  
Ingeniería  
Informática  
Universidad de Oviedo

---

# Trabajo en grupo

## Fase II

---

PL3-B

**Andrés Fernández-Junquera Fernández UO302806**

**Bruno Martín Rivera UO302144**

**Javier Ortín Rodenas UO299855**

**Mateo Rama García UO300710**

**Fundamentos de computadores y redes**

# Índice

<b>1. Descifrar entradas válidas</b>	<b>2</b>
1.1. Stage1() . . . . .	2
1.2. Stage2() . . . . .	4
1.3. Stage3() . . . . .	6
1.4. Stage4() . . . . .	8
<b>2. Modificación del fichero ejecutable</b>	<b>11</b>
<b>3. División del trabajo</b>	<b>14</b>

## 1. Descifrar entradas válidas

En esta primera parte, se describirían los pasos que se llevaron a cabo por parte de los integrantes del grupo para descifrar las entradas válidas de cada una de las Stages.

Para descifrar cualquiera de las entradas válidas, se utilizó el modo depuración que nos ofrece Visual Studio 2022. Para ello, abrimos el archivo `main.exe` y ejecutamos en modo depuración. Haciendo click derecho, seleccionamos la opción ir al desensamblado. Una vez allí, avanzamos las diferentes sentencias en ensamblador con F10 hasta llegar a la sentencia de ensamblador en la que se llama a la función `Stage()` correspondiente, pulsamos F11 para acceder al código de esta función en lenguaje ensamblador.

### 1.1. Stage1()

Una vez hemos pulsado F11 en la llamada a la función `Stage1()`, nos encontramos con el siguiente fragmento de código en ensamblador:

```
00445D30 55          push     ebp           ≤ 1 ms transcurridos
00445D31 8B EC       mov     ebp,esp
00445D33 81 EC EC 03 00 00 sub     esp,3ECh
00445D39 C7 45 FC E8 03 00 00 mov     dword ptr [ebp-4],3E8h
00445D40 6A 00       push     0
00445D42 68 E8 03 00 00 push     3E8h
00445D47 8D 85 14 FC FF FF lea     eax,[ebp-3ECh]
00445D4D 50          push     eax
00445D4E B9 A8 1B 55 00 mov     ecx,offset std::cin (0551BA8h)
00445D53 E8 48 3D 00 00 call    std::basic_istream<char,std::char_traits<char> >::getline (0449AA0h)
00445D58 68 80 2A 52 00 push     522A80h
00445D5D 8D 8D 14 FC FF FF lea     ecx,[ebp-3ECh]
00445D63 51          push     ecx
00445D64 E8 B7 15 06 00 call    strcmp (04A7320h)
00445D69 83 C4 08     add     esp,8
00445D6C 85 C0       test    eax,eax
00445D6E 74 07       je      Stage1+47h (0445D77h)
00445D70 E8 5B FF FF FF call    Explode (0445CD0h)
00445D75 EB 05       jmp     Stage1+4Ch (0445D7Ch)
00445D77 E8 84 FF FF FF call    Defuse (0445D00h)
00445D7C 8B E5       mov     esp,ebp
00445D7E 5D          pop     ebp
00445D7F C3          ret
```

Al observar este fragmento de código, podemos observar que la llamada a la función `Explode()` se encuentra en la dirección de memoria `00445D70h` y la llamada a la función `Defuse()` se encuentra en la dirección `00445D77h`.

En nuestro caso, queremos desactivar la bomba, por tanto solo nos interesa la función `Defuse()`. Para ello buscamos, una instrucción en ensamblador que sea un salto condicional a la dirección de memoria de la función `Defuse()`.

```
00445D4D 50          push     eax
00445D4E B9 A8 1B 55 00 mov     ecx,offset std::cin (0551BA8h)
00445D53 E8 48 3D 00 00 call    std::basic_istream<char,std::char_traits<char> >::getline (0449AA0h)
00445D58 68 80 2A 52 00 push     522A80h
00445D5D 8D 8D 14 FC FF FF lea     ecx,[ebp-3ECh]
00445D63 51          push     ecx
00445D64 E8 B7 15 06 00 call    strcmp (04A7320h)
00445D69 83 C4 08     add     esp,8
00445D6C 85 C0       test    eax,eax
00445D6E 74 07       je      Stage1+47h (0445D77h)
00445D70 E8 5B FF FF FF call    Explode (0445CD0h)
00445D75 EB 05       jmp     Stage1+4Ch (0445D7Ch)
00445D77 E8 84 FF FF FF call    Defuse (0445D00h)
```

Como se muestra en la anterior imagen el salto condicional a la función `Defuse()` esta almacenado en la dirección de memoria `00445D6Eh`. Este salto es del tipo JE es decir,

salta en caso de que sean iguales los dos valores que se comparan. Si observamos, las líneas de código anteriores, podemos ver que se realiza un `test eax, eax`, la cual es una instrucción que realiza un AND y solo modifica la Zero Flag. Esta instrucción solo activará la Zero Flag si el resultado el registro `eax` es 0.

Sabemos que el registro `eax` se usa normalmente para almacenar el resultado de operaciones aritméticas o llamadas a funciones. En nuestro caso, como se puede observar arriba se realiza la llamada a la función `strcmp()`. Por tanto, `eax` almacenará el valor que retorne está función, la cuál devuelve 0 si son iguales las dos cadenas comparadas y un valor distinto en caso contrario.

Para llamar a la función `strcmp()` se necesitan pasar dos cadenas en forma de parámetro a través de la pila, por tanto debemos observar las instrucciones superiores con el objetivo de observar los distintos `push` que se realizan.

Podemos observar, que se llama a la función `cin.getLine()` para almacenar la cadena que el usuario introduce por teclado. Esta función almacena la cadena a partir de la dirección de memoria que se encuentra almacenada en el registro `eax`, ya que anteriormente se realiza la instrucción `push eax`.

A continuación, se mueve la dirección de memoria en la que se encuentra almacenada la cadena introducida por el usuario al registro `ecx` y se realiza un `push ecx` para pasarla como parámetro a la función `strcmp()`.

En las líneas de código superiores, también se puede observar que se realiza un `push 522A80h`, la cual es el otro parámetro que se le pasa a `strcmp()`. Por tanto, podemos deducir que se trata de la dirección de memoria que contiene el primer carácter la cadena con la que se va a realizar la comparación y la cuál es la que tenemos que averiguar, para desactivar la bomba. Ayudándonos de Visual Studio 2022, buscamos la dirección de memoria `522A80h` y nos encontramos con los caracteres codificados en código ASCII, como se muestra en la imagen.

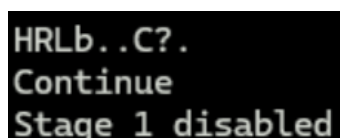


0x00522A80 48 52 4c 62 2e 2e 43 3f 2e 00

En esta dirección de memoria, seleccionamos los bytes hasta el terminador de la cadena, la cual es un `\0` y lo traducimos de ASCII a lenguaje natural, obteniendo la siguiente cadena:

HRLb..C?.

Por último, comprobamos que la cadena obtenida es correcta, como se muestra en la siguiente imagen.



HRLb..C?.  
Continue  
Stage 1 disabled

## 1.2. Stage2()

Una vez hemos pulsado F11 en la llamada a la función `Stage2()`, nos encontramos con el siguiente fragmento de código en ensamblador:

```
00445D80 55          push     ebp           ≤ 1 ms transcurridos
00445D81 8B EC       mov      ebp,esp
00445D83 83 EC 1C    sub      esp,1Ch
00445D86 53          push     ebx
00445D87 C7 45 F4 04 00 00 00 mov     dword ptr [ebp-0Ch],4
00445D8E C7 45 FC 00 00 00 00 mov     dword ptr [ebp-4],0
00445D95 EB 09       jmp      Stage2+20h (00445DA0h)
00445D97 8B 45 FC    mov     eax,dword ptr [ebp-4]
00445D9A 83 C0 01    add     eax,1
00445D9D 8B 45 FC    mov     dword ptr [ebp-4],eax
00445DA0 83 7D FC 04 cmp     dword ptr [ebp-4],4
00445DA4 7D 14       jge      Stage2+3Ah (00445DBAh)
00445DA6 8B 4D FC    mov     ecx,dword ptr [ebp-4]
00445DA9 8D 54 8D E4 lea     edx,[ebp+ecx*4-1Ch]
00445DAD 52          push     edx
00445DAE B9 A8 1B 55 00 mov     ecx,offset std::cin (0551BA0h)
00445DB3 E8 38 25 00 00 call    std::basic_istream<char,std::char_traits<char> >::operator>> (004482F0h)
00445DB8 EB DD       jmp      Stage2+17h (00445D97h)
00445DBA C7 45 F8 01 00 00 00 mov     dword ptr [ebp-8],1
00445DC1 8D 5D E4    lea     ebx,[ebp-1Ch]
00445DC4 8B 43 04    mov     eax,dword ptr [ebx+4]
00445DC7 03 43 08    add     eax,dword ptr [ebx+8]
00445DCA 83 F8 F6    cmp     eax,0FFFFFFFh
00445DCD 75 07       jne      Stage2+56h (00445DD6h)
00445DCF C7 45 F8 00 00 00 00 mov     dword ptr [ebp-8],0
00445DD6 83 7D F8 00 cmp     dword ptr [ebp-8],0
00445DDA 74 07       je       Stage2+63h (00445DE3h)
00445DDC E8 EF FE FF FF call    Explode (00445CD0h)
00445DE1 EB 05       jmp      Stage2+68h (00445DE8h)
00445DE3 E8 18 FF FF FF call    Defuse (00445D00h)
00445DE8 5B          pop      ebx
00445DE9 8B E5       mov     esp,ebp
00445DEB 5D          pop      ebp
00445DEC C3          ret
```

Al igual que para el `Stage1()`, podemos observar que en la dirección de memoria `00445DE3h` se encuentra la llamada a la función `Defuse()`, la cuál es la que nos interesa. En este caso, la estrategia será diferente, no empezaremos de atrás hacia delante, sino que primero leeremos el código ensamblador para poder entenderlo y así poder descifrar las entradas válidas.

Observando el código, podemos ver que se realiza un bucle en el que en cada iteración el usuario debe introducir un número, los cuáles se almacenan en las direcciones de memoria `ebp+ecx*4-1Ch`, donde el registro `ecx` en esta instrucción se encarga de almacenar el número de iteración en la que se encuentra el bucle.

Una vez el bucle ha finalizado y el usuario ha introducido los 4 números, se mueve a la dirección de memoria `ebp-8` el valor 1. En la siguiente instrucción, se realiza un `lea ebx, [ebp - 1Ch]`. Así, el registro `ebx` almacenará la dirección de memoria correspondiente a la dirección de memoria donde se encuentra almacenado el primer número introducido por el usuario. En las dos siguientes instrucciones se mueve al registro `eax` el segundo valor introducido por el usuario y se suma en este mismo registro el tercer número introducido por el usuario, como se muestra en la siguiente imagen.

```

00445DC4 8B 43 04      mov     eax,dword ptr [ebx+4]
00445DC7 03 43 08      add     eax,dword ptr [ebx+8]
00445DCA 83 F8 F6      cmp     eax,0FFFFFFF6h
00445DCD 75 07         jne     Stage2+56h (0445DD6h)
00445DCF C7 45 F8 00 00 00 00 mov     dword ptr [ebp-8],0
00445DD6 83 7D F8 00    cmp     dword ptr [ebp-8],0
00445DDA 74 07         je      Stage2+63h (0445DE3h)
00445DDC E8 EF FE FF FF call    Explode (0445CD0h)
00445DE1 EB 05         jmp     Stage2+68h (0445DE8h)
00445DE3 E8 18 FF FF FF call    Defuse (0445D00h)

```

Tras realizar la suma, compara el valor de la suma con el valor 0FFFFFFF6h el cual es -10 en decimal. En caso de que sean iguales, el salto condicional que se encuentra en la siguiente línea (`jne Stage2+56h (0445DD6h)`) no se verifica y por tanto las dos siguientes instrucciones se ejecutan. Estas, simplemente mueven el valor 0 a la dirección de memoria `ebp-8` y compara el valor de esta dirección de memoria con el valor 0. Lo cual es verdad siempre, así finalmente se cumple la siguiente sentencia `jmp Stage2+63h (00445DE3h)` que es la que llama a la función `Defuse()`.

De esto, concluimos que para desactivar la bomba, el usuario debe introducir 4 números de forma que la suma del segundo y tercer número sea igual a -10. Procedemos a comprobar que esto es así:

```

0
-7
-3
2
Continue
Stage 2 disabled|

```

Notar que en caso de que no sumasen -10, el valor de la dirección de memoria `ebp-8` no se cambiaría y por tanto almacenaría el valor 1. En ese caso, al realizar la comparación con el valor 0 no serían iguales y por tanto, se llamaría a la función `Explode()` y la bomba estallaría.

```

-1
-5
-10
1
Oh, no, the world is over! BOOM!

```

### 1.3. Stage3()

Esta función `Stage3` nos pide introducir por teclado dos números, así que estudiaremos las características necesarias de nuestros parámetros para desactivar esta tercera etapa. Al pulsar F11 en la función `Stage3()` podemos observar la primera parte del código:

```
00445DF0 push     ebp           ; 1 ms transcurridos
00445DF1 mov     ebp,esp
00445DF3 sub     esp,14h
00445DF6 lea     eax,[ebp-8]
00445DF9 push    eax
00445DFA lea     ecx,[ebp-4]
00445DFD push    ecx
00445DFE mov     ecx,offset std::cin (0551BA8h)
00445E03 call    std::basic_istream<char,std::char_traits<char> >::operator>> (04482F0h)
00445E08 mov     ecx,eax
00445E0A call    std::basic_istream<char,std::char_traits<char> >::operator>> (04482F0h)
00445E0F mov     edx,dword ptr [ebp-8]
00445E12 and     edx,2
00445E15 sar     edx,1
00445E17 mov     dword ptr [ebp-0Ch],edx
00445E1A mov     eax,dword ptr [ebp-4]
00445E1D and     eax,800h
00445E22 sar     eax,0Bh
00445E25 mov     dword ptr [ebp-14h],eax
00445E28 mov     ecx,dword ptr [ebp-4]
00445E2B and     ecx,4000000h
00445E31 sar     ecx,1Ah
00445E34 mov     dword ptr [ebp-10h],ecx
00445E37 mov     edx,dword ptr [ebp-0Ch]
00445E3A cmp     edx,dword ptr [ebp-10h]
00445E3D jne     Stage3+5Ch (0445E4Ch)
00445E3F cmp     dword ptr [ebp-14h],1
00445E43 jne     Stage3+5Ch (0445E4Ch)
00445E45 call    Explode (0445CD0h)
00445E4A jmp     Stage3+61h (0445E51h)
00445E4C call    Defuse (0445D00h)
00445E51 mov     esp,ebp
00445E53 pop     ebp
```

Notamos que la llamada a la función `Defuse()` se encuentra en la dirección de memoria `00445E4Ch`, por lo que veremos cómo llegar a ella paso por paso.

Después de la preparación de la pila (las tres primeras líneas), se leen las 2 entradas por teclado del usuario. Obsérvese las instrucciones guardadas en las direcciones `00445E03h` y `00445E0Ah`: están ambas leyendo por teclado y se han guardado los valores de esas entradas en `[ebp-8]` y `[ebp-4]`.

```
00445DF6 lea     eax,[ebp-8]
00445DF9 push    eax
00445DFA lea     ecx,[ebp-4]
00445DFD push    ecx
00445DFE mov     ecx,offset std::cin (0551BA8h)
00445E03 call    std::basic_istream<char,std::char_traits<char> >::operator>> (04482F0h)
00445E08 mov     ecx,eax
00445E0A call    std::basic_istream<char,std::char_traits<char> >::operator>> (04482F0h)
```

Con el fin de simplificar la explicación, llamaremos variable 1, 2 y 3 a los valores que hemos obtenido a partir de los parámetros introducidos por el usuario.

**Variable 1:** Se guarda el primer valor introducido en `edx` y le aplica una operación `and` con el valor 2. De este modo, el resultado guardará solo el bit 1 (segundo por la derecha), que tomará el valor 1 si el de la variable era 1 y valdrá 0 si el de la variable era 0 (el resto de bits valdrán 0 como resultado del `and`).

Justo después, desplaza esta variable un bit a la derecha (**sar edx,1**), por lo que la variable valdrá exactamente 1 o 0, según el valor del bit que originalmente era el número 1. Guarda este resultado en la dirección de memoria **ebp-0Ch**.

**Variable2:** Ahora, en las instrucciones guardadas a partir de la dirección de memoria **00445E1Ah**, se trabaja con el segunda número introducida. Se toma su valor y con otra máscara **and** y el valor **800h** (=1000...00b) se comprueba el valor del bit número 11, y se desplaza hasta el bit número 0. Se guarda este valor en la dirección de memoria **ebp-14h**.

```
00445E1A  mov          eax,dword ptr [ebp-4]
00445E1D  and          eax,800h
00445E22  sar          eax,0Bh
00445E25  mov          dword ptr [ebp-14h],eax
```

**Variable3:** Ahora se vuelve a tomar el segundo parámetro originalmente introducido y se introduce a la variable **ecx**. Con la máscara **and ecx,4000000h** se mira si el bit 26 del primer número es 1 y se vuelve a mover 26 bits a la derecha con la instrucción **sar**. Se guarda este valor en la dirección de memoria **ebp-10h**.

```
00445E28  mov          ecx,dword ptr [ebp-4]
00445E2B  and          ecx,4000000h
00445E31  sar          ecx,1Ah
00445E34  mov          dword ptr [ebp-10h],ecx
```

Finalmente, recogemos las variables 1 y 3. La instrucción **jne** (jump if not equal) hace que la bomba se desactive (llevando el puntero de dirección a la instrucción que llama a **Defuse()**) si los dos valores son distintos. En otro caso, si la variable 2 es distinta a 1 (el bit 11 del segundo parámetro igual a 0) se desactiva también la bomba. Si no se cumple nada de esto, la bomba explota.

```
00445E37  mov          edx,dword ptr [ebp-0Ch]
00445E3A  cmp          edx,dword ptr [ebp-10h]
00445E3D  jne          Stage3+5Ch (0445E4Ch)
00445E3F  cmp          dword ptr [ebp-14h],1
00445E43  jne          Stage3+5Ch (0445E4Ch)
00445E45  call         Explode (0445CD0h)
```

En resumen, para poder desactivar la bomba nos piden que el bit 1 del primer numero y el bit 26 de la segunda sean distintos. O en su defecto, que el bit 11 de la segunda sea 0.

Notamos que los parámetros deben ser números siempre, de ser caracteres explotará la bomba.



## 1.4. Stage4()

Siguiendo el mismo procedimiento que en los casos anteriores, al pulsar F11 en la llamada a `Stage4()` obtenemos el siguiente código:

```
00445E60 55          push     ebp
00445E61 8B EC       mov      ebp,esp
00445E63 81 EC EC 03 00 00 sub     esp,3ECh
00445E69 6A 0A       push     0Ah
00445E6B E8 50 40 00 00 call    std::numeric_limits<__int64>::max (0449EC0h)
00445E70 52          push     edx
00445E71 50          push     eax
00445E72 B9 A8 1B 55 00 mov     ecx,offset std::cin (0551BA8h)
00445E77 E8 B4 3E 00 00 call    std::basic_istream<char,std::char_traits<char> >::ignore (0449D30h)
00445E7C C7 45 FC E8 03 00 00 mov     dword ptr [ebp-4],3E8h
00445E83 6A 00       push     0
00445E85 68 E8 03 00 00 push    3E8h
00445E8A 8D 85 14 FC FF FF lea     eax,[ebp-3ECh]
00445E90 50          push     eax
00445E91 B9 A8 1B 55 00 mov     ecx,offset std::cin (0551BA8h)
00445E96 E8 05 3C 00 00 call    std::basic_istream<char,std::char_traits<char> >::getline (0449AA0h)
00445E9B B9 01 00 00 00 mov     ecx,1
00445EA0 6B D1 05     imul     edx,ecx,5
00445EA3 0F BE 84 15 14 FC FF FF movsx    eax,byte ptr [ebp+edx-3ECh]
00445EAB B9 01 00 00 00 mov     ecx,1
00445EB0 C1 E1 00     shl     ecx,0
00445EB3 0F BE 94 0D 14 FC FF FF movsx    edx,byte ptr [ebp+ecx-3ECh]
00445EBB 3B C2       cmp     eax,edx
00445EBD 74 23       je      Stage4+82h (0445EE2h)
00445EBF B8 01 00 00 00 mov     eax,1
00445EC4 6B C8 03     imul     ecx,eax,3
00445EC7 0F BE 94 0D 14 FC FF FF movsx    edx,byte ptr [ebp+ecx-3ECh]
00445ECF B8 01 00 00 00 mov     eax,1
00445ED4 D1 E0       shl     eax,1
00445ED6 0F BE 8C 05 14 FC FF FF movsx    ecx,byte ptr [ebp+eax-3ECh]
00445EDE 3B D1       cmp     edx,ecx
00445EE0 74 07       je      Stage4+89h (0445EE9h)
00445EE2 E8 E9 FD FF FF call    Explode (0445CD0h)
00445EE7 EB 05       jmp     Stage4+8Eh (0445EEh)
00445EE9 E8 12 FE FF FF call    Defuse (0445D00h)
00445EEE 8B E5       mov     esp,ebp
00445EF0 5D          pop      ebp
00445EF1 C3          ret
```

Notamos que las llamadas a `Defuse()` y `Explode()` se encuentran en las direcciones 445EE9h y 445EE2h, respectivamente. Además, se corresponden con un salto relativo de 89h y 82h (respectivamente) desde la etiqueta `Stage4`. Conocer el valor de estos saltos relativos nos sirve de ayuda para comprender cómo funciona el código. En este caso, explicaremos qué hace el código secuencialmente.

```
00445E60 55          push     ebp
00445E61 8B EC       mov      ebp,esp
00445E63 81 EC EC 03 00 00 sub     esp,3ECh
00445E69 6A 0A       push     0Ah
00445E6B E8 50 40 00 00 call    std::numeric_limits<__int64>::max (0449EC0h)
00445E70 52          push     edx
00445E71 50          push     eax
00445E72 B9 A8 1B 55 00 mov     ecx,offset std::cin (0551BA8h)
00445E77 E8 B4 3E 00 00 call    std::basic_istream<char,std::char_traits<char> >::ignore (0449D30h)
```

Las tres primeras líneas componen el prólogo del procedimiento, reservando 1004 (3ECh) espacios de pila. A continuación, se llama a `numeric_limits::max` para un entero de 64 bits y obtener así el mayor valor que puede almacenar una variable de este tipo. Sabemos que el registro `eax` es comunmente utilizado para dejar los valores de retorno. No obstante, como almacena 32 bits y el valor esperado es de 64, es necesario utilizar más espacio para guardar tal valor. Es por ello que también se hace un push de `edx`, pues estos dos registros contienen en conjunto el entero de 64 bits correspondiente al máximo posible en un tipo `__int64`.

A continuación, se apila el valor de este número para que lo utilice. La cuarta línea no apila un parámetro para `numeric_limits`, pues esta función no toma parámetros. Se apila `0Ah` para ser utilizada por `ignore`, ya que es la codificación del salto de línea en ASCII. De este modo, se mueve a `ecx` la dirección de `cin` y se llama a `ignore` para que este flujo ignore las entradas previas hasta encontrar el carácter `'\n'` o llegar al máximo establecido.

```
00445E7C C7 45 FC E8 03 00 00 mov     dword ptr [ebp-4],3E8h
00445E83 6A 00          push    0
00445E85 68 E8 03 00 00 push    3E8h
00445E8A 8D 85 14 FC FF FF lea     eax,[ebp-3ECh]
00445E90 50          push    eax
00445E91 B9 A8 1B 55 00 mov     ecx,offset std::cin (0551BA8h)
00445E96 E8 05 3C 00 00 call    std::basic_istream<char,std::char_traits<char> >::getline (0449AA0h)
```

En esta parte se reserva espacio para guardar una cadena de 1000 (`3E8h`) caracteres como máximo que será leída por `getline`. Es por ello que se apila este valor como parámetro (longitud máxima) junto con la dirección de inicio la cadena (se deja la dirección en `eax` con `lea` y se apila `eax`). La función `getline` modificará las direcciones de memoria pertinentes para guardar la entrada del usuario. Puesto que `0` se corresponde con el código del terminador `'\0'`, apilar este valor podría servir como delimitador para `getline`.

```
00445E9B B9 01 00 00 00 mov     ecx,1
00445EA0 6B D1 05      imul    edx,ecx,5
00445EA3 0F BE 84 15 14 FC FF FF movsx    eax,byte ptr [ebp+edx-3ECh]
00445EAB B9 01 00 00 00 mov     ecx,1
00445EB0 C1 E1 00      shl     ecx,0
00445EB3 0F BE 94 0D 14 FC FF FF movsx    edx,byte ptr [ebp+ecx-3ECh]
00445EBB 3B C2        cmp     eax,edx
00445EBD 74 23        je      Stage4+82h (0445EE2h)
```

Una vez leída la cadena, se deja el valor `1` en `ecx`. Se multiplica ahora tal valor por `5`, dejando el resultado en `edx`. A continuación, se accede al índice `5` de la cadena (sexto carácter) y se almacena en `eax`. Se realiza un procedimiento análogo al dejar el carácter de índice `1` de la cadena en `edx`, pues se desplaza `1` en `0` bits a la izquierda (el resultado es `1`). Finalmente, se compara el contenido de `eax` con el de `edx`, saltando a la llamada a `Explode` en caso de ser iguales.

```
00445EBF B8 01 00 00 00 mov     eax,1
00445EC4 6B C8 03      imul    ecx,eax,3
00445EC7 0F BE 94 0D 14 FC FF FF movsx    edx,byte ptr [ebp+ecx-3ECh]
00445ECF B8 01 00 00 00 mov     eax,1
00445ED4 D1 E0        shl     eax,1
00445ED6 0F BE 8C 05 14 FC FF FF movsx    ecx,byte ptr [ebp+eax-3ECh]
00445EDE 3B D1        cmp     edx,ecx
00445EE0 74 07        je      Stage4+89h (0445EE9h)
00445EE2 E8 E9 FD FF FF call    Explode (0445CD0h)
00445EE7 EB 05        jmp     Stage4+8Eh (0445EEEh)
00445EE9 E8 12 FE FF FF call    Defuse (0445D00h)
00445EEE 8B E5        mov     esp,ebp
00445EF0 5D          pop     ebp
00445EF1 C3          ret
```

Análogamente, se comparan otros dos caracteres de la clave introducida por el usuario. El primero es el situado en el índice 3 (resultado de multiplicar 3 por 1), mientras que el segundo es el de índice 2 (se desplaza un bit a la izquierda el valor 1, lo que equivale a multiplicarlo por 2). A diferencia del caso anterior, buscamos que sí coincidan ahora, pues de lo contrario explotaría la bomba.

Por todo lo anterior, podemos deducir las condiciones que debe cumplir la clave introducida por el usuario para que la bomba no explote en esta fase. En primer lugar, cabe recordar que los índices de las cadenas comienzan en 0. Por tanto, para que no explote, la cadena ha de tener su sexto caracter distinto de su segundo. Además, el cuarto caracter y el tercero sean iguales. Ambas condiciones han de verificarse simultáneamente, o de lo contrario la bomba explotará.

```
aabbcc
Continue
Stage 4 disabled
```

Ejemplo de entrada válida

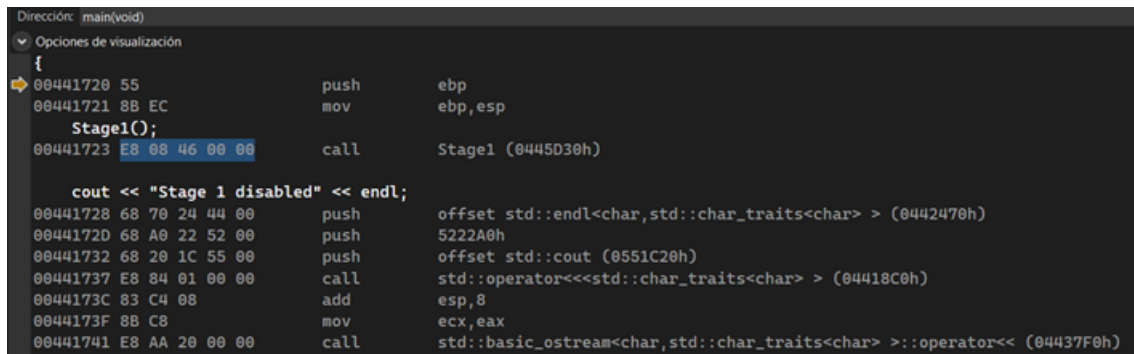
```
aaaaaaaaaaaa
Oh, no, the world is over! BOOM!
```

Ejemplo de entrada no válida

## 2. Modificación del fichero ejecutable

Para modificar el código de la bomba para que indique que está desactivada sin necesidad de introducir ninguna entrada, vamos a proceder substituyendo las instrucciones `call` de cada Stage por la instrucción `NOP`. Vamos a mostrar este procedimiento para Stage1, siendo análogo para las otras tres fases.

Primero, abrimos en Visual Studio el archivo `main.exe`, pulsamos F10 para empezar a depurar y mostramos el desensamblado incluyendo los bytes de código.

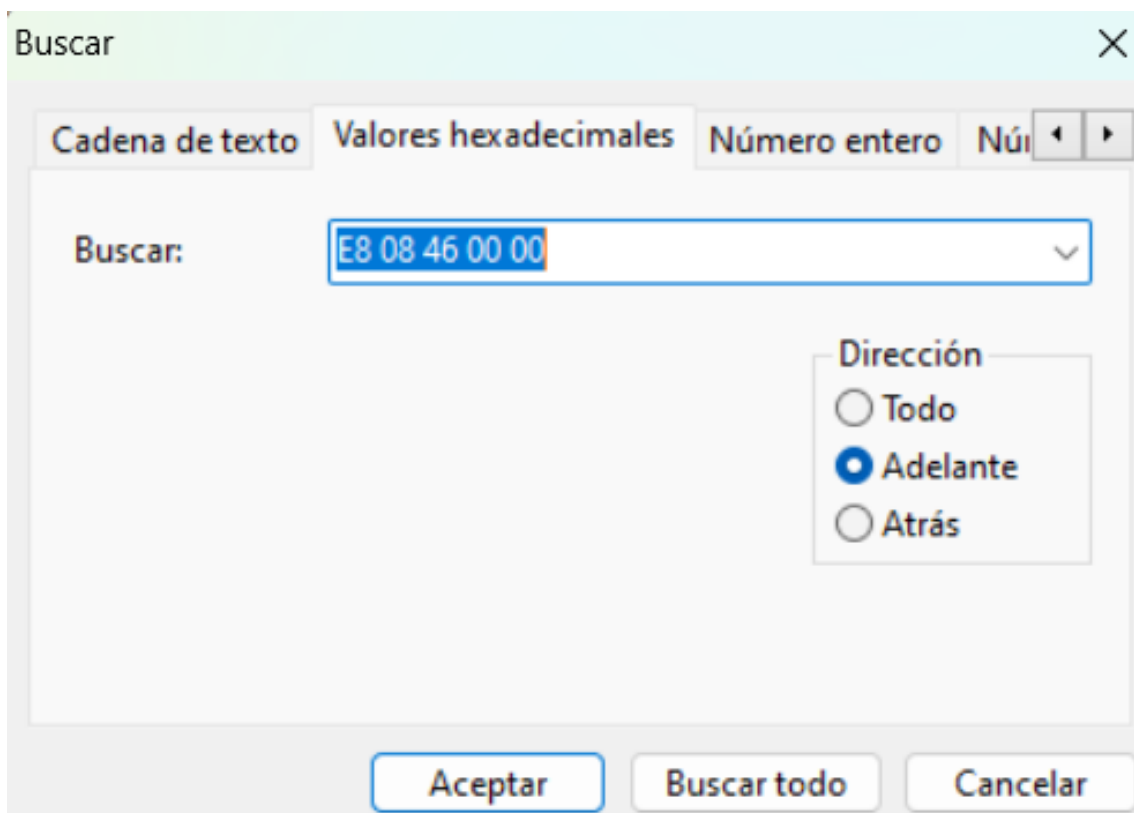


```
Dirección: main(void)
Opciones de visualización
{
00441720 55          push     ebp
00441721 8B EC       mov     ebp,esp
00441723 E8 08 46 00 00 call    Stage1 (00445030h)

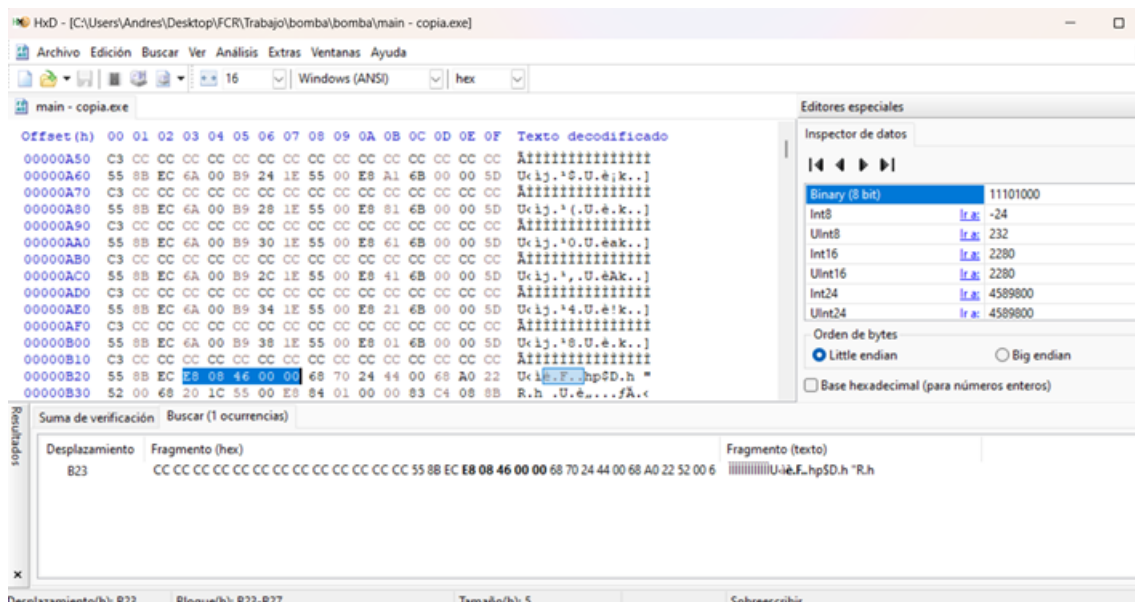
    cout << "Stage 1 disabled" << endl;
00441728 68 70 24 44 00 push    offset std::endl<char,std::char_traits<char> > (00442470h)
0044172D 68 A0 22 52 00 push    5222A0h
00441732 68 20 1C 55 00 push    offset std::cout (00551C20h)
00441737 E8 84 01 00 00 call    std::operator<<<std::char_traits<char> > (004418C0h)
0044173C 83 C4 08    add     esp,8
0044173F 8B C8       mov     ecx,eax
00441741 E8 AA 20 00 00 call    std::basic_ostream<char,std::char_traits<char> >::operator<< (004437F0h)
```

Así, podemos ver que la instrucción `call Stage1` se codifica como `E8 08 46 00 00`.

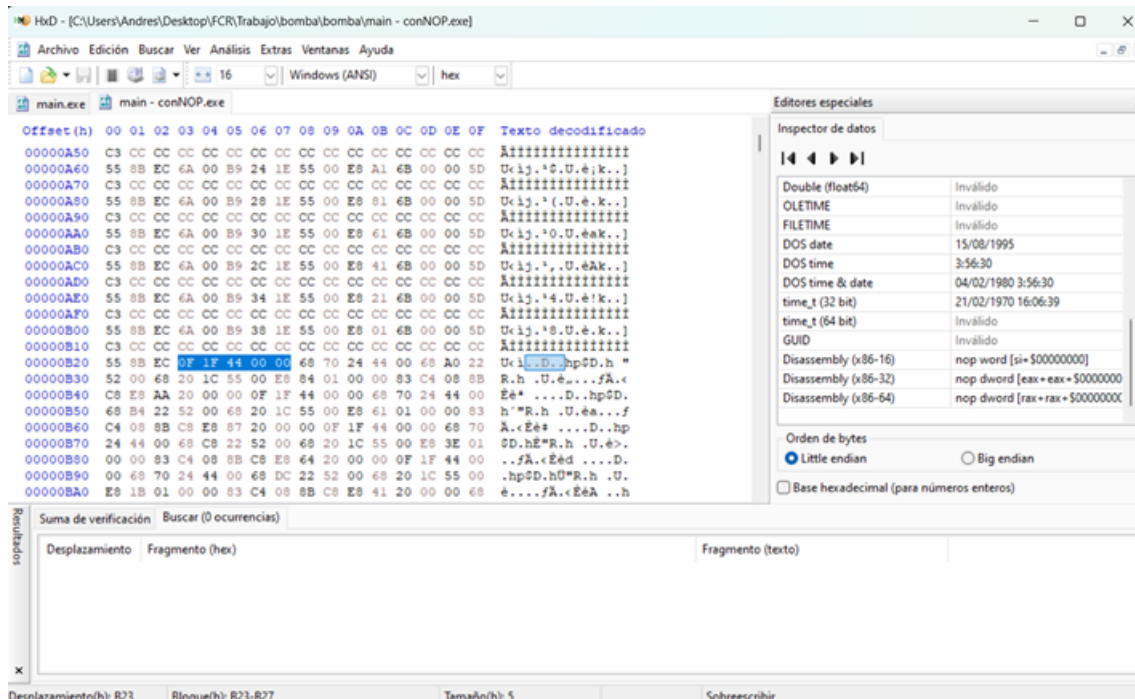
Utilizando ahora el programa HxD, abrimos una copia de `main.exe`. Para encontrar la llamada a `Stage1`, pulsamos `Ctrl+F`, seleccionamos "Valores hexadecimales", y buscamos el valor `E8 08 46 00 00`.



Tras esto, veremos algo similar a la siguiente pantalla, donde se muestra esta llamada a `Stage1` entre el resto de instrucciones de `main` codificadas en código máquina y expresadas en hexadecimal.



Por último, sustituimos estos 5 bytes por la instrucción NOP en 5 bytes, codificada por la secuencia 0F 1F 44 00 00.



Repitiendo este procedimiento con Stage2, Stage3 y Stage4, conseguiremos que, al ejecutar la función main, se indique que la bomba está desactivada sin necesidad de introducir ninguna entrada.

```
C:\Users\Andres\Desktop\VCF x + v - □ x
Stage 1 disabled
Stage 2 disabled
Stage 3 disabled
Stage 4 disabled
Wow, you've just saved the Earth!
Press Enter to exit...
|
```

### **3. División del trabajo**