

ESCUELA DE INGENIERÍA INFORMÁTICA DE OVIEDO

Fundamentos de Computadores Y Redes

Curso 2024-2025

Trabajo Grupal - Fase II

Díaz Mendaña, Diego - UO301887

García Pernas, Pablo - UO300167

Gota Ortín, Jorge - UO301023

Suárez Fernández, Fernando - UO300028

Grupo de prácticas: PL.3 - A

Titulación: PCEO Informática y Matemáticas

26 de abril de 2025

Índice

1. Introducción	2
2. Desarrollo del análisis de las palabras clave	2
2.1. Primera etapa	3
2.1.1. Obtención de la cadena	3
2.2. Segunda etapa	5
2.2.1. Condiciones para los valores introducidos	5
2.3. Tercera etapa	7
2.3.1. Condiciones para los valores introducidos	8
2.4. Cuarta etapa	9
2.4.1. Condiciones para la cadena introducida	10
3. Como desactivar la bomba (y no morir en el intento)	12
4. Distribución del trabajo	13
4.1. Estrategia de trabajo	13
4.2. Distribución de tareas	13
4.3. Tiempos de desarrollo	13

1. Introducción

La presente memoria tiene por objetivo documentar de manera exhaustiva el desarrollo de la Fase II del trabajo práctico correspondiente a la asignatura **Fundamentos de Computadores y Redes**. El propósito fundamental de esta fase consiste en profundizar en los conceptos generales abordados previamente en los laboratorios de la materia, aplicándolos en un contexto de análisis práctico.

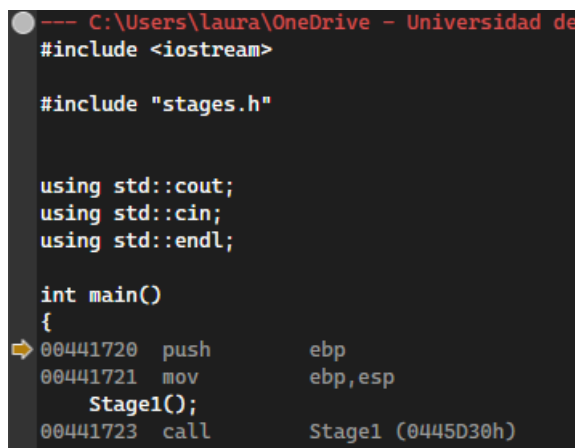
En particular, la práctica propuesta requiere llevar a cabo el análisis detallado del código fuente de un programa, utilizando para ello las herramientas de depuración que proporciona el entorno de desarrollo **Visual Studio**. A través de este análisis, se pretende identificar las palabras clave necesarias para la desactivación de una “bomba binaria”. Como actividad final, se procederá a modificar el código de manera que la bomba no explote independientemente de las entradas proporcionadas por el usuario.

2. Desarrollo del análisis de las palabras clave

El programa objeto de análisis ha sido suministrado por el profesor de la asignatura y se encuentra ubicado en la carpeta denominada *bomba* de este repositorio, concretamente en el archivo ejecutable `main.exe`. Este ejecutable, diseñado para el sistema operativo Windows, solicita durante su ejecución una serie de entradas por parte del usuario. En caso de que dichas entradas sean correctas, el programa continúa su ejecución sin incidentes, en caso contrario, la “bomba” programada explota (la demostración de este comportamiento queda fuera del alcance de la presente memoria).

El programa en cuestión se encuentra estructurado en cuatro etapas, cuya ejecución secuencial condiciona la desactivación segura de la bomba. En los apartados siguientes, se describirá detalladamente el comportamiento observado en cada una de estas fases.

Para llevar a cabo el análisis de cada etapa, se han empleado las herramientas de depuración integradas en Visual Studio, tales como el **depurador**, el **desensamblador** y el **analizador de memoria**. Al iniciar la depuración del programa, se realiza el desensamblado inicial del código, obteniendo el siguiente resultado:



```
--- C:\Users\laura\OneDrive - Universidad de
#include <iostream>

#include "stages.h"

using std::cout;
using std::cin;
using std::endl;

int main()
{
00441720 push     ebp
00441721 mov      ebp, esp
      Stage1();
00441723 call    Stage1 (0445D30h)
```

En este fragmento desensamblado puede observarse que, en la tercera línea de código, mediante la instrucción `call`, se invoca al procedimiento **Stage1**. Para analizar el comportamiento de dicho procedimiento, se procede a avanzar la ejecución con dos pulsaciones de F10 (para ejecutar las instrucciones línea a línea) y, a continuación, con F11 para ingresar dentro del procedimiento y examinar su contenido en profundidad.

2.1. Primera etapa

Una vez que se accede al procedimiento **Stage1**, se observa el siguiente fragmento de código desensamblado:

```

➔ 00445D30 push      ebp           ≤ 1 ms transcurridos
00445D31 mov       ebp,esp
00445D33 sub       esp,3ECh
00445D39 mov       dword ptr [ebp-4],3E8h
00445D40 push      0
00445D42 push      3E8h
00445D47 lea       eax,[ebp-3ECh]
00445D4D push      eax
00445D4E mov       ecx,offset std::cin (0551BA8h)
00445D53 call      std::basic_istream<char,std::char_traits<char> >::getline (0449AA0h)
00445D58 push      522A80h
00445D5D lea       ecx,[ebp-3ECh]
00445D63 push      ecx
00445D64 call      strcmp (04A7320h)
00445D69 add       esp,8
00445D6C test      eax,eax
00445D6E je       Stage1+47h (0445D77h)
00445D70 call      Explode (0445CD0h)
00445D75 jmp      Stage1+4Ch (0445D7Ch)
00445D77 call      Defuse (0445D00h)
00445D7C mov       esp,ebp
00445D7E pop      ebp
00445D7F ret

```

De forma resumida, puede apreciarse que el procedimiento inicialmente reserva espacio en la pila con el propósito de almacenar una cadena de gran tamaño. Posteriormente, emplea la función `std::getline` para leer una línea completa de texto ingresada por el usuario a través de `std::cin`. A continuación, el contenido introducido se compara con una cadena predefinida mediante la función `strcmp`:

- Si la comparación resulta exitosa, el programa invoca el procedimiento **Defuse**, que desactiva la fase correspondiente.
- En caso contrario, se llama a **Explode**, desencadenando un fallo catastrófico.

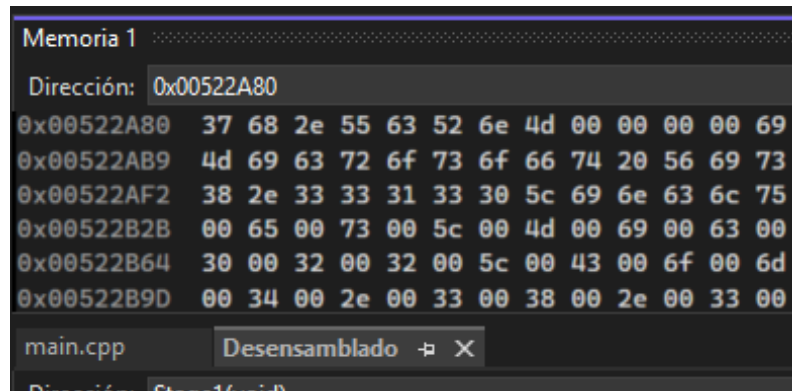
En este punto, resulta de gran interés determinar cuál es la cadena correcta que debe ser ingresada para evitar la detonación, preservando así la integridad del programa y, metafóricamente, del “mundo”.

2.1.1. Obtención de la cadena

Para identificar la cadena correcta, es crucial examinar la instrucción:

1 **PUSH** 522A80h

Dicha instrucción revela la dirección de memoria (522A80h) en la cual se encuentra almacenada la cadena predefinida con la que se realizará la comparación. Por tanto, resulta fundamental acceder al contenido de dicha dirección para obtener la secuencia exacta que debe introducirse. Para ello, se hace uso del analizador de memoria de Visual Studio, introduciendo la dirección mencionada. El resultado del análisis es el siguiente:



En el análisis de memoria se observa que la secuencia de bytes correspondiente es 37 68 2e 55 63 52 6e 4d 00. Cabe destacar que en C++ las cadenas de caracteres finalizan con el byte 00, indicando el carácter nulo de terminación (\0). A partir de la interpretación de estos valores, se concluye que la cadena correcta que debe ser introducida es: 7h.UcRnM.

Una vez introducida dicha cadena durante la ejecución del programa, si se continúa la depuración avanzando con F10, el procedimiento culminará invocando a Defuse y, posteriormente, ejecutará una instrucción ret para regresar al main, mostrando el siguiente código en pantalla:

```
int main()
{
00441720 push     ebp
00441721 mov      ebp,esp
    Stage1();
00441723 call     Stage1 (00445D30h)

    cout << "Stage 1 disabled" << endl;
00441728 push     offset std::endl<char,std::char_traits<char> > (00442470h)
0044172D push     522A0h
00441732 push     offset std::cout (00551C20h)
00441737 call     std::operator<<<std::char_traits<char> > (004418C0h)
0044173C add      esp,8
0044173F mov      ecx,eax
00441741 call     std::basic_ostream<char,std::char_traits<char> >::operator<< (004437F0h)
```

Asimismo, la consola presentará el siguiente aspecto:

```
7h.UcRnM
Continue
Stage 1 disabled|
```

2.2. Segunda etapa

A continuación, en el flujo de ejecución del procedimiento `main`, puede observarse una invocación a la segunda etapa mediante la instrucción `call Stage2`. Siguiendo la misma metodología previamente establecida, se accede al procedimiento `Stage2` utilizando `F11`, y se procede a desensamblar su contenido, obteniéndose el siguiente resultado:

```
00445D80 push     ebp           ≤ 1 ms transcurridos
00445D81 mov      ebp,esp
00445D83 sub      esp,18h
00445D86 push     ebx
00445D87 mov      dword ptr [ebp-0Ch],3
00445D8E mov      dword ptr [ebp-4],0
00445D95 jmp      Stage2+20h (00445DA0h)
00445D97 mov      eax,dword ptr [ebp-4]
00445D9A add      eax,1
00445D9D mov      dword ptr [ebp-4],eax
00445DA0 cmp      dword ptr [ebp-4],3
00445DA4 jge      Stage2+3Ah (00445DBAh)
00445DA6 mov      ecx,dword ptr [ebp-4]
00445DA9 lea      edx,[ebp+ecx*4-18h]
00445DAD push     edx
00445DAE mov      ecx,offset std::cin (0551BA8h)
00445DB3 call     std::basic_istream<char,std::char_traits<char> >::operator>> (004482F0h)
00445DB8 jmp      Stage2+17h (00445D97h)
00445DBA mov      dword ptr [ebp-8],1
00445DC1 lea      ebx,[ebp-18h]
00445DC4 mov      eax,dword ptr [ebx+8]
00445DC7 add      eax,dword ptr [ebx+4]
00445DCA cmp      eax,0FFFFFFFh
00445DCD jne      Stage2+56h (00445DD6h)
00445DCF mov      dword ptr [ebp-8],0
00445DD6 cmp      dword ptr [ebp-8],0
00445DDA je       Stage2+63h (00445DE3h)
00445DDC call     Explode (00445CD0h)
00445DE1 jmp      Stage2+68h (00445DE8h)
00445DE3 call     Defuse (00445D00h)
00445DE8 pop      ebx
00445DE9 mov      esp,ebp
00445DEB pop      ebp
00445DEC ret
```

El código resultante presenta una estructura similar a la observada en la primera etapa: reserva espacio en la pila y configura algunas variables locales necesarias para el control del flujo de ejecución. Posteriormente, se lleva a cabo la siguiente secuencia de operaciones:

1. Se establece un bucle que se repite exactamente tres veces, en el cual se solicita al usuario la introducción de un valor numérico a través de `std::cin`.
2. Se realizan comprobaciones y operaciones específicas sobre los números ingresados para verificar ciertas condiciones.
3. Si todas las verificaciones son satisfactorias, se invoca el procedimiento `Defuse`; en caso contrario, se llama a `Explode`.

2.2.1. Condiciones para los valores introducidos

Al analizar en detalle la estructura del procedimiento, destacan los siguientes aspectos:

1. **Inicialización de variables:** Se inicializa una variable local con el valor 3 para controlar el número de iteraciones:

```
1 00445D0F MOV dword ptr [ebp-0Ch], 3
```

Seguidamente, se inicializa otra variable local con el valor 0, la cual actuará como contador:

```
1 00445D16 MOV dword ptr [ebp-4], 0
```

2. **Inicio del bucle:** Se efectúa un salto al control del bucle mediante:

```
1 00445D1D JMP Stage2+20h (00445D0Ah)
```

3. **Cuerpo del bucle:** En cada iteración, se incrementa el contador con:

```
1 00445D9A MOV eax, 1
2 00445D9F ADD dword ptr [ebp-4], eax
```

Una vez que el contador alcanza el valor 3, se abandona el bucle:

```
1 00445DA2 CMP dword ptr [ebp-4], 3
2 00445DA6 JGE Stage2+3Ah (00445DBAh)
```

Durante las iteraciones, se leen los números ingresados por el usuario:

```
1 00445DA8 MOV ecx, dword ptr [ebp-4]
2 00445DAB LEA edx, [ebp+ecx*4-18h]
3 00445DAF PUSH edx
4 00445DB0 MOV ecx, offset std::cin
5 00445DB5 CALL std::basic_istream<char>::operator>>
```

Cada número leído se almacena en una dirección diferente dentro de la pila ([ebp-18h], [ebp-14h] y [ebp-10h]).

4. **Comprobaciones:** Finalizada la lectura de los tres números, se procede a verificar que la suma de los dos últimos ([ebp-14h] y [ebp-10h]) sea igual a -6. Si esta condición se cumple, se invoca Defuse; de lo contrario, se llama a Explode:

```
1 00445DBA MOV dword ptr [ebp-8],1 ; Variable de estado
2 00445DC1 LEA ebx,[ebp-18h] ; Dirección de los números
3 00445DC4 MOV eax,dword ptr [ebx+8] ; Carga el tercer número
4 00445DC7 ADD eax,dword ptr [ebx+4] ; Suma con el segundo número
5 00445DCA CMP eax,0FFFFFFFAh ; Compara con -6
6 00445DCD JNE Stage2+56h (00445DD6h) ; Si no son iguales, salta
7 00445DCF MOV dword ptr [ebp-8],0
8 00445DD6 CMP dword ptr [ebp-8],0
9 00445DDA JE Stage2+63h (00445DE3h)
10 00445DDC CALL Explode (00445CD0h)
11 00445DE1 JMP Stage2+68h (00445DE8h)
12 00445DE3 CALL Defuse (00445D00h)
```

5. **Finalización:** Tras completar la verificación y realizar la llamada pertinente a Defuse o Explode, el procedimiento finaliza y retorna el control al main mediante la instrucción ret.

Por consiguiente, para superar esta segunda etapa con éxito, el usuario deberá introducir tres valores numéricos de los cuales la suma de los dos últimos sea igual a -6 . Tras la ejecución correcta, el resultado en Visual Studio será el siguiente:

```

Stage2();
00441746 call    Stage2 (0445D80h)

cout << "Stage 2 disabled" << endl;
0044174B push    offset std::endl<char,std::char_traits<char> > (0442470h)
00441750 push    5222B4h
00441755 push    offset std::cout (0551C20h)
0044175A call    std::operator<<<std::char_traits<char> > (04418C0h)
0044175F add     esp,8
00441762 mov     ecx,eax
00441764 call    std::basic_ostream<char,std::char_traits<char> >::operator<< (04437F0h)

```

Y la consola mostrará el siguiente estado:

```

7h.UcRnM
Continue
Stage 1 disabled
1
-3
-3
Continue
Stage 2 disabled|

```

2.3. Tercera etapa

El procedimiento `main` realiza una llamada a `Stage3` mediante la instrucción `call Stage3`. Procediendo de manera análoga a las etapas anteriores, accedemos al procedimiento utilizando F11 y desensamblamos su contenido, obteniendo el siguiente resultado:

```

00445DF0 push    ebp           ≤ 1 ms transcurridos
00445DF1 mov     ebp,esp
00445DF3 sub     esp,14h
00445DF6 lea     eax,[ebp-4]
00445DF9 push    eax
00445DFA lea     ecx,[ebp-8]
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-4]
00445E12 and     edx,8
00445E15 sar     edx,3
00445E18 mov     dword ptr [ebp-0Ch],edx
00445E1B mov     eax,dword ptr [ebp-8]
00445E1E and     eax,8000h
00445E23 sar     eax,0Fh
00445E26 mov     dword ptr [ebp-10h],eax
00445E29 mov     ecx,dword ptr [ebp-4]
00445E2C and     ecx,800h
00445E32 sar     ecx,0Bh
00445E35 mov     dword ptr [ebp-14h],ecx
00445E38 mov     edx,dword ptr [ebp-0Ch]
00445E3B cmp     edx,dword ptr [ebp-10h]
00445E3E jne     Stage3+56h (0445E46h)
00445E40 cmp     dword ptr [ebp-14h],0
00445E44 jne     Stage3+5Dh (0445E4Dh)
00445E46 call    Explode (0445CD0h)
00445E4B jmp     Stage3+62h (0445E52h)
00445E4D call    Defuse (0445D00h)
00445E52 mov     esp,ebp
00445E54 pop     ebp
00445E55 ret

```


En esta ocasión, el procedimiento consiste en la lectura de dos valores numéricos introducidos por el usuario, para posteriormente analizar y comparar ciertos bits específicos de estos números a fin de verificar condiciones precisas.

2.3.1. Condiciones para los valores introducidos

El procedimiento **Stage3** se desarrolla de la siguiente manera:

1. **Preparación inicial:** Se guarda el valor actual de **ebp** y se reserva espacio en la pila para variables locales:

```
1 00445DF0  PUSH ebp
2 00445DF1  MOV  ebp, esp
3 00445DF3  SUB  esp, 14h
```

2. **Lectura de números:** Se solicitan dos valores al usuario mediante **std::cin**:

- El primer número se almacena en **[ebp-8]**:

```
1 00445DF6  LEA  eax, [ebp-4]
2 00445DF9  PUSH eax
3 00445DFA  LEA  ecx, [ebp-8]
4 00445DFD  PUSH ecx
5 00445DFE  MOV  ecx, offset std::cin
6 00445E03  CALL std::basic_istream<char>::operator>>
```

- El segundo número se almacena en **[ebp-4]**:

```
1 00445E08  MOV  ecx, eax
2 00445E0A  CALL std::basic_istream<char>::operator>>
```

3. **Procesamiento:** Se realizan operaciones de enmascaramiento y desplazamiento de bits sobre los números ingresados:

- Al segundo número leído (**[ebp-4]**), se le aplica una máscara **0x8** (seleccionando el bit 3) y se desplaza tres posiciones a la derecha, almacenándose el resultado en **[ebp-0Ch]**:

```
1 00445E0F  MOV  edx, dword ptr [ebp-4]
2 00445E12  AND  edx, 8
3 00445E15  SAR  edx, 3
4 00445E18  MOV  dword ptr [ebp-0Ch], edx
```

- Al primer número leído (**[ebp-8]**), se le aplica una máscara **0x8000** para aislar el bit 15, desplazándolo posteriormente 15 posiciones hacia la derecha y almacenando el resultado en **[ebp-10h]**.
- Igualmente, al segundo número leído (**[ebp-4]**) se le aplica una máscara **0x800** (bit 11) y se desplaza once posiciones a la derecha, almacenándose en **[ebp-14h]**.

4. **Comprobaciones:** Se realizan verificaciones específicas sobre los bits procesados:

- Si el valor almacenado en `[ebp-0Ch]` (bit 3 del segundo número) difiere del valor de `edx`, se activa la bomba:

```
1 00445E38  CMP dword ptr [ebp-0Ch], edx
2 00445E3B  JNE Stage3+56h (00445E4Eh)
```

- Si el bit 15 del primer número (`[ebp-10h]`) es igual a 1, el programa explota.
- Si el bit 11 del segundo número (`[ebp-14h]`) es igual a 1, el programa igualmente explota.

5. **Finalización:** Si todas las condiciones son satisfechas correctamente, el procedimiento invoca a `Defuse` y retorna al `main` utilizando la instrucción `ret`, de manera análoga a las fases anteriores.

Tras la correcta introducción de los valores y la verificación exitosa de las condiciones, se invoca `Defuse` y, al regresar al procedimiento principal, se observará el siguiente estado en Visual Studio:

```
Stage3();
00441769 call Stage3 (0445DF0h)

cout << "Stage 3 disabled" << endl;
0044176E push offset std::endl<char,std::char_traits<char> > (0442470h)
00441773 push 5222C8h
00441778 push offset std::cout (0551C20h)
0044177D call std::operator<<<std::char_traits<char> > (04418C0h)
00441782 add esp,8
00441785 mov ecx,eax
00441787 call std::basic_ostream<char,std::char_traits<char> >::operator<< (04437F0h)
```

Y la consola presentará el siguiente aspecto:

```
7h.UcRnM
Continue
Stage 1 disabled
1
-3
-3
Continue
Stage 2 disabled
0
2048
Continue
Stage 3 disabled
```

2.4. Cuarta etapa

Finalmente, en el procedimiento `main`, se realiza una llamada a la cuarta etapa mediante `call Stage4`. Procediendo de manera análoga a los casos anteriores, accedemos al procedimiento utilizando `F11` y desensamblamos su contenido, obteniendo el siguiente resultado:

```

00445E60 push     ebp           ≤ 1 ms transcurridos
00445E61 mov     ebp,esp
00445E63 sub     esp,3ECh
00445E69 push     0Ah
00445E6B call    std::numeric_limits<__int64>::max (0449EC0h)
00445E70 push     edx
00445E71 push     eax
00445E72 mov     ecx,offset std::cin (0551BA8h)
00445E77 call    std::basic_istream<char,std::char_traits<char> >::ignore (0449D30h)
00445E7C mov     dword ptr [ebp-4],3E8h
00445E83 push     0
00445E85 push     3E8h
00445E8A lea     eax,[ebp-3ECh]
00445E90 push     eax
00445E91 mov     ecx,offset std::cin (0551BA8h)
00445E96 call    std::basic_istream<char,std::char_traits<char> >::getline (0449AA0h)
00445E9B mov     ecx,1
00445EA0 shl     ecx,0
00445EA3 movsx   edx,byte ptr [ebp+ecx-3ECh]
00445EAB mov     eax,1
00445EB0 shl     eax,2
00445EB3 movsx   ecx,byte ptr [ebp+eax-3ECh]
00445EBB cmp     edx,ecx
00445EBD je     Stage4+83h (0445EE3h)
00445EBF mov     edx,1
00445EC4 imul    eax,edx,5
00445EC7 movsx   ecx,byte ptr [ebp+eax-3ECh]
00445ECF mov     edx,1
00445ED4 imul    eax,edx,3
00445ED7 movsx   edx,byte ptr [ebp+eax-3ECh]
00445EDF cmp     ecx,edx
00445EE1 je     Stage4+8Ah (0445EEAh)
00445EE3 call    Explode (0445CD0h)
00445EE8 jmp     Stage4+8Fh (0445EEFh)
00445EEA call    Defuse (0445D00h)
00445EEF mov     esp,ebp
00445EF1 pop     ebp
00445EF2 ret

```

En esta fase, el programa solicita al usuario la introducción de una línea completa de texto, para posteriormente realizar diversas comprobaciones entre caracteres específicos de la cadena.

2.4.1. Condiciones para la cadena introducida

El procedimiento **Stage4** se desarrolla de la siguiente manera:

1. **Preparación inicial:** Se guarda el registro **ebp** y se reserva espacio en la pila para el almacenamiento de datos:

```

1 00445E60 PUSH ebp
2 00445E61 MOV  ebp, esp
3 00445E63 SUB  esp, 3ECh

```

2. **Lectura de datos:** Se solicita al usuario una línea completa mediante la función **std::getline**:

```

1 00445E7C MOV  eax, [ebp-4]
2 00445E83 PUSH 3E8h
3 00445E88 LEA  eax, [ebp-3ECh]
4 00445E8B PUSH eax
5 00445E8C MOV  ecx, offset std::cin
6 00445E91 CALL std::getline

```

3. **Procesamiento de caracteres:** Se accede inicialmente al segundo carácter de la cadena:

```
1 00445E98 MOV ecx, 1
2 00445E9D SHL ecx, 0
3 00445EA0 MOVSX edx, byte ptr [ebp+ecx-3ECh]
```

De manera similar, se extrae el tercer carácter para su posterior comparación.

4. **Comprobación:** Se realizan dos comprobaciones principales:

- En primer lugar, si el segundo y el tercer carácter de la cadena son idénticos, se invoca inmediatamente al procedimiento **Explode**:

```
1 00445EAE CMP ecx, edx
2 00445EB0 JE Stage4+83h (00445EE3h)
```

- Si esta condición no se cumple, el programa continúa y compara el cuarto carácter con el sexto. Si estos dos caracteres son iguales, se invoca el procedimiento **Defuse**:

```
1 00445EB2 MOV edx, 1
2 00445EB7 IMUL eax, edx, 5
3 00445EBC MOVSX ecx, byte ptr [ebp+eax-3ECh]
4 00445EBF MOV edx, 1
5 00445EC4 IMUL eax, edx, 3
6 00445EC9 MOVSX edx, byte ptr [ebp+eax-3ECh]
7 00445ECC CMP ecx, edx
```

5. **Finalización:** Si las condiciones de comprobación son correctas y la bomba no detona, el procedimiento se finaliza adecuadamente retornando al **main** mediante la instrucción **ret**, tal como sucedió en las etapas anteriores.

Tras la correcta ejecución de esta cuarta etapa, el control retorna al procedimiento principal, observándose el siguiente estado en la consola:

```
7h.UcRnM
Continue
Stage 1 disabled
1
-3
-3
Continue
Stage 2 disabled
0
2048
Continue
Stage 3 disabled
iaiiii
Continue
Stage 4 disabled
Wow, you've just saved the Earth!
Press Enter to exit...
```

Esto es debido a que se han desactivado exitosamente todas las etapas de la bomba y, por ello, se muestra el mensaje de *“Wow, you’ve saved the Earth”*. Posteriormente, al presionar la tecla **Enter**, el programa se cierra retornando el valor 1 en el procedimiento **main**.

Y en este caso, el código de Visual Studio presentará el siguiente aspecto:

```
Stage4();
0044178C call      Stage4 (0445E60h)

    cout << "Stage 4 disabled" << endl;
00441791 push      offset std::endl<char,std::char_traits<char> > (0442470h)
00441796 push      5222DCh
0044179B push      offset std::cout (0551C20h)
004417A0 call      std::operator<<<std::char_traits<char> > (04418C0h)
004417A5 add       esp,8
004417A8 mov       ecx,eax
004417AA call      std::basic_ostream<char,std::char_traits<char> >::operator<< (04437F0h)

    cout << "Wow, you've just saved the Earth!" << endl;
004417AF push      offset std::endl<char,std::char_traits<char> > (0442470h)
004417B4 push      5222F0h
004417B9 push      offset std::cout (0551C20h)
004417BE call      std::operator<<<std::char_traits<char> > (04418C0h)
004417C3 add       esp,8
004417C6 mov       ecx,eax
004417C8 call      std::basic_ostream<char,std::char_traits<char> >::operator<< (04437F0h)

    cout << "Press Enter to exit..." << endl;
004417CD push      offset std::endl<char,std::char_traits<char> > (0442470h)
004417D2 push      522314h
004417D7 push      offset std::cout (0551C20h)
004417DC call      std::operator<<<std::char_traits<char> > (04418C0h)
004417E1 add       esp,8
004417E4 mov       ecx,eax
004417E6 call      std::basic_ostream<char,std::char_traits<char> >::operator<< (04437F0h)
    cin.get();
004417EB mov       ecx,offset std::cin (0551BA8h)
004417F0 call      std::basic_istream<char,std::char_traits<char> >::get (04453C0h)

    return 1;
004417F5 mov       eax,1
}
```

3. Como desactivar la bomba (y no morir en el intento)

Como hemos visto previamente, para la correcta desactivación de la bomba, algunas posibles entradas válidas son:

1. 7h.UcRnM para la primera etapa.
2. 1, -3, -3 para la segunda etapa.
3. 0, 2048
4. iaaaaa para la cuarta etapa.

De forma similar, se pueden plantear cadenas que romperían el programa, como por ejemplo:

1. UwU para la primera etapa.
2. 6, 6, 6 para la segunda etapa.
3. 314159, 271828 para la tercera etapa.
4. hespaña para la cuarta etapa.

4. Distribución del trabajo

4.1. Estrategia de trabajo

La implementación del proyecto se ha organizado siguiendo una estrategia de trabajo colaborativo, basada en la asignación individual de cada una de las funciones principales descritas en el enunciado. Para el reparto del trabajo, en este caso fue algo más complejo que la última vez debido a que la división no era tan sencilla.

Como soporte para el trabajo en equipo, se ha utilizado el sistema de control de versiones distribuido *Git* en combinación con la plataforma *Github* como repositorio remoto centralizado. En este caso particular, dado que no se requería de implementaciones de código simultáneas, no se ha requerido un flujo de trabajo basado en *features branches*. En esta fase, otra herramienta empleada ha sido *Discord*, que es un servicio de mensajería instantánea y VoIP, que ha facilitado la comunicación entre los miembros del grupo.

Gracias a estas herramientas, se ha podido llevar a cabo un trabajo colaborativo eficiente, permitiendo a cada miembro del grupo realizar sus aportaciones de manera individual y realizar una propuesta en común de los resultados obtenidos.

El repositorio que ha servido como entorno de trabajo colaborativo se encuentra alojado en la plataforma *GitHub*, bajo el siguiente enlace: <https://github.com/PabloGarPe/fcrtrabajo>. A diferencia de la primera fase, para la entrega este repositorio ya se encontrará abierto públicamente.

4.2. Distribución de tareas

Tal como se ha expuesto anteriormente, la distribución de tareas se ha estructurado en torno a la asignación individual de las partes del trabajo a los distintos integrantes del grupo. Tras el análisis individual se realizó una puesta en común de los resultados obtenidos, con el fin de que todos los miembros del grupo tuvieran un conocimiento general de la práctica y de los resultados obtenidos, además de facilitar la realización de la memoria. A continuación, se detalla la asignación específica de tareas realizada por cada miembro del equipo:

Integrante	Función implementada
Diego Díaz Mendaña	<i>Stage 2 y memoria</i>
Pablo García Pernas	<i>Stage 1 y Stage 3</i>
Fernando Suárez Fernández	<i>Cosa del final</i>
Jorge Gota Ortín	<i>Stage 4</i>

4.3. Tiempos de desarrollo

A continuación, se especifica el tiempo estimado invertido por cada miembro, considerando tanto las tareas individuales como las actividades colaborativas de revisión y coordinación:

Integrante	Tiempo dedicado
Diego Díaz Mendaña	5 horas
Pablo García Pernas	3 horas y 30 minutos
Fernando Suárez Fernández	3 horas
Jorge Gota Ortín	3 horas Z