Memoria sobre la segunda fase de HackingForce

Grupo 4 – PL4, FCR GIITIN01 (20-21)

UO283319 – Juan Francisco Mier Montoto  
UO282574 - Miguel del Riego Lázaro

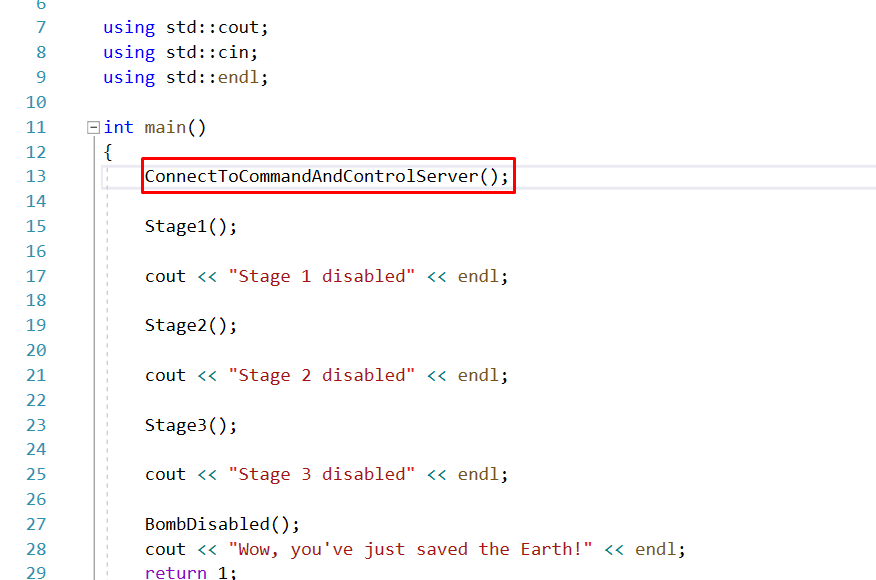
# Índice

* [Fase 2 - Salvando al mundo](#_Fase_2_–)
  + [Nombre del subgrupo](#_Nombre_del_subgrupo)
  + [Desactivando etapas](#_Desactivando_etapas)
    - [Bomba 1](#_Bomba_1)
    - [Bomba 2](#_Bomba_2)
    - [Bomba 3](#_Bomba_3)
  + Modificación del código
* [Reparto de trabajo](#_Reparto_del_trabajo)

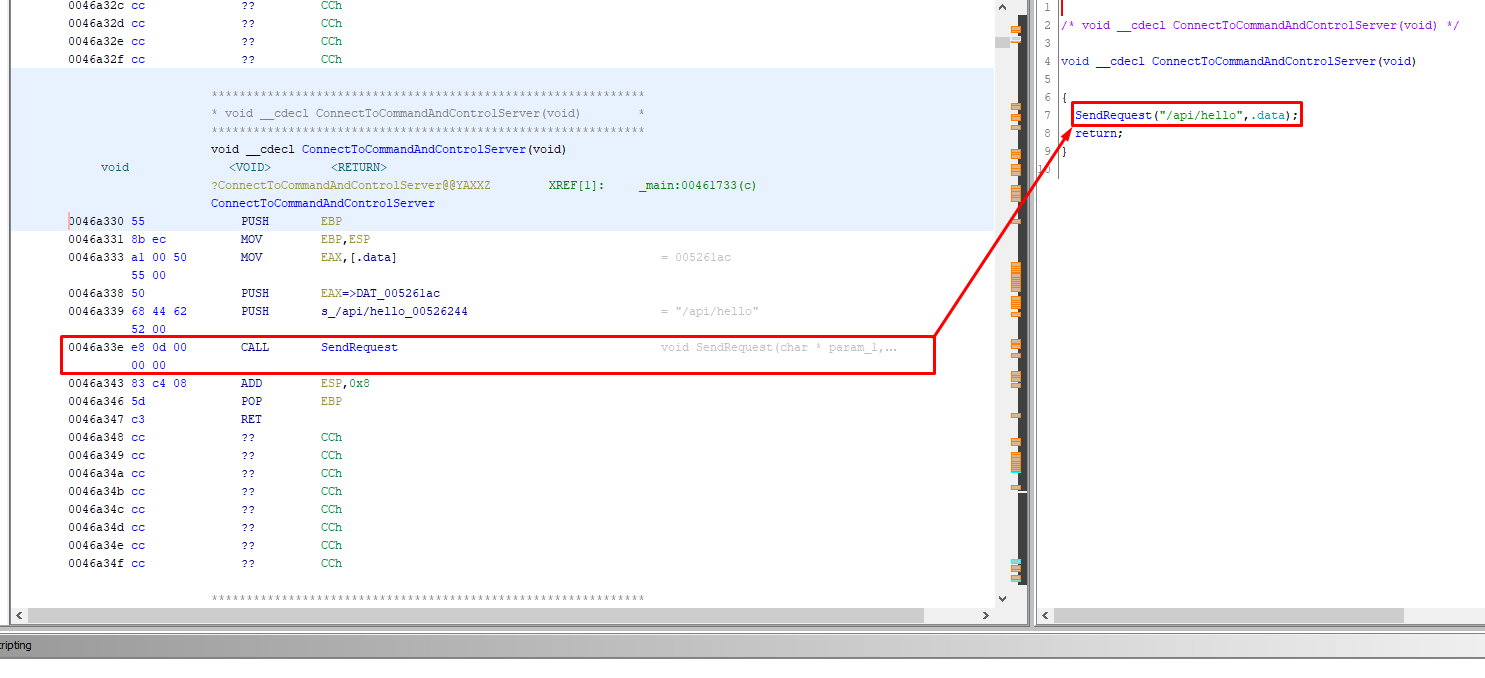
# Fase 2 – Salvando al mundo

Tras aprender completar el entrenamiento en Assembly y C++, “alguien” ha decidido utilizar sus conocimientos para poner a todo el mundo a prueba. Lo primero de todo será descubrir de quién se trata.

## Nombre del subgrupo

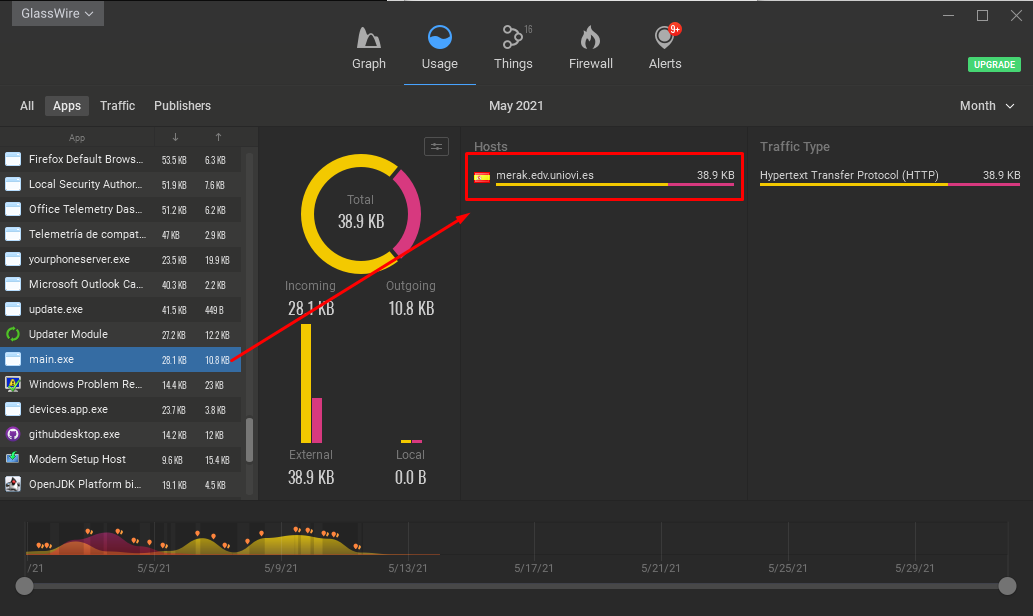
Para obtener el nombre del subgrupo, hay que analizar los paquetes que envía y recibe la aplicación principal al ser ejecutada, más concretamente durante el procedimiento “ConnectToCommandAndControlServer()”.

Analizando el ejecutable con Ghidra, podemos ver lo que hace realmente este procedimiento.

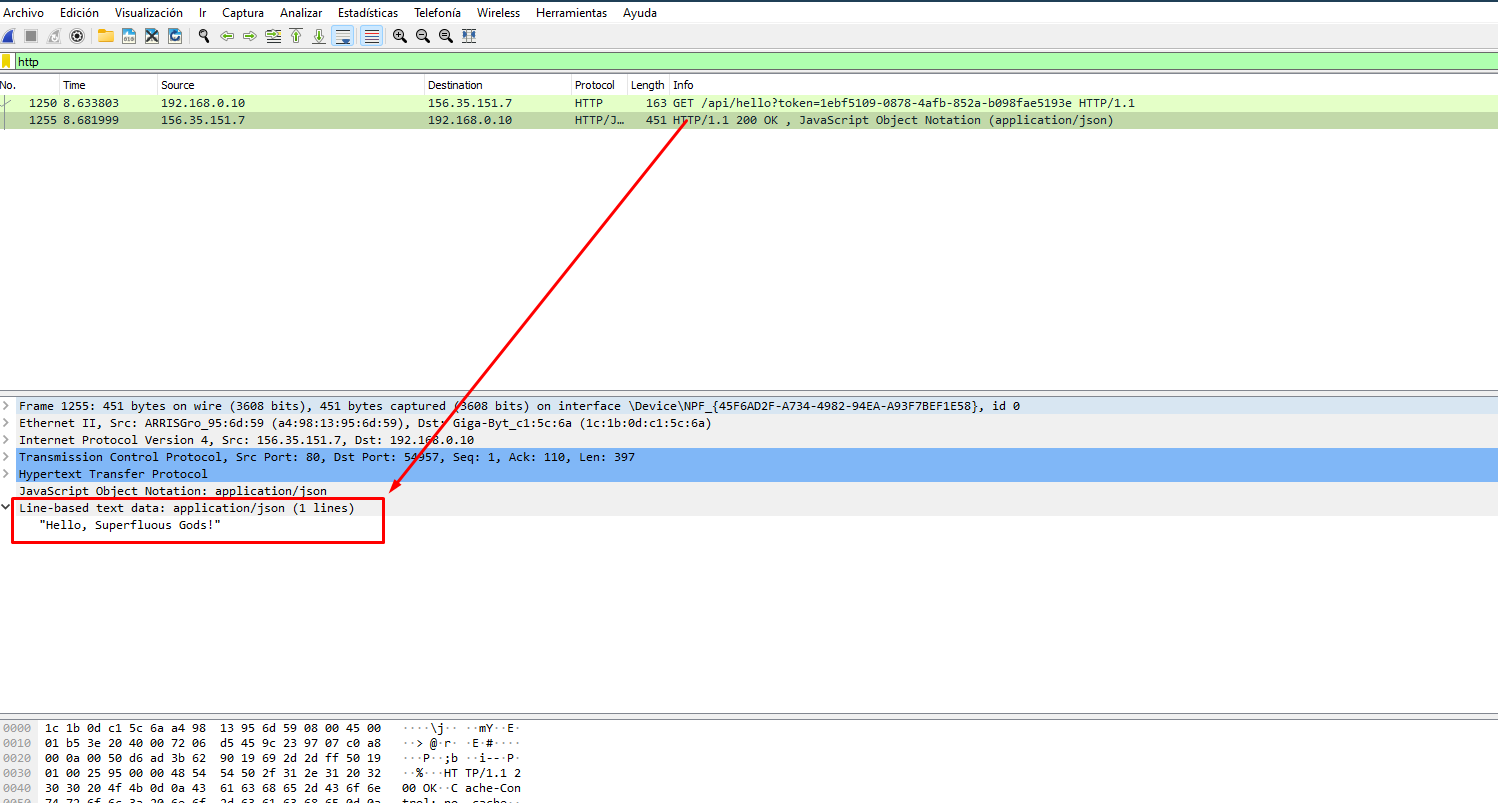


Al parecer, está mandando una señal muy sencilla al servidor mediante “SendRequest()”, que es el procedimiento encargado también de enviar el número de la bomba y la contraseña al desactivar una bomba.

Al ejecutar el programa, podemos observar claramente esta petición, que está dirigida a un servidor de uniovi:



Para observar este paquete, abrimos WireShark antes de la ejecución y filtramos por “http”:



Como podemos observar, a la petición GET de SendRequest() se devuelve un 200 OK que incluye un json de una sola línea que contiene muy convenientemente el nombre del subgrupo, en este caso “Superfluous Gods”.

## Desactivando etapas

Esta es, sin duda, la parte más difícil del programa: desactivar las bombas. Para hacerlo, ya habíamos visto que podríamos mandar un GET manualmente para desactivarlas como se hace en el código, como por ejemplo en la bomba 1:







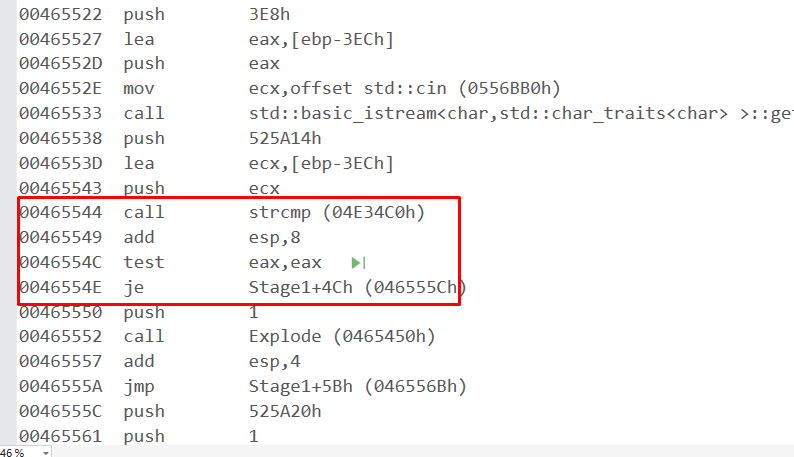
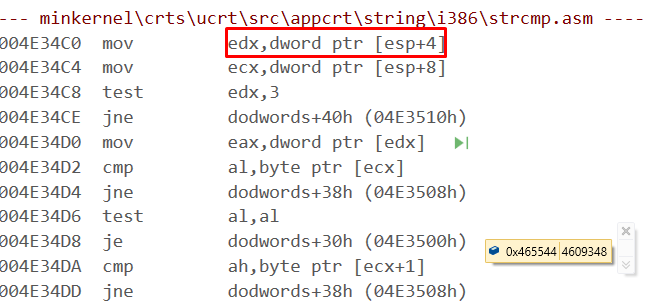
O al llegar al final del archivo, al desactivar todas las bombas:



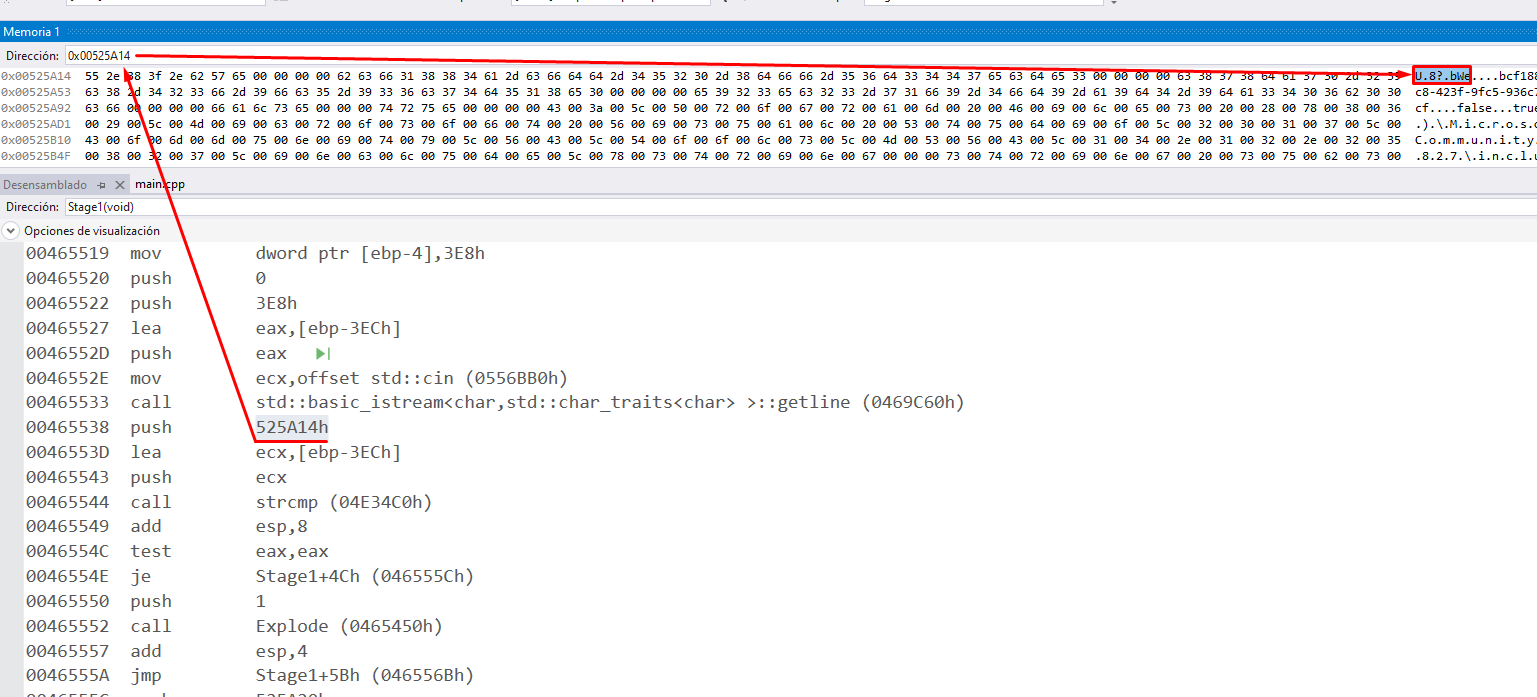
Pero lo que se nos pide es buscar en el código para introducir los códigos que desactiven las bombas. El método general para todas bombas será poner un punto de interrupción en la fase que queramos examinar, pasar a desensamblado e introducirnos (F11) en el procedimiento para poder observarlo y examinarlo.

### Bomba 1

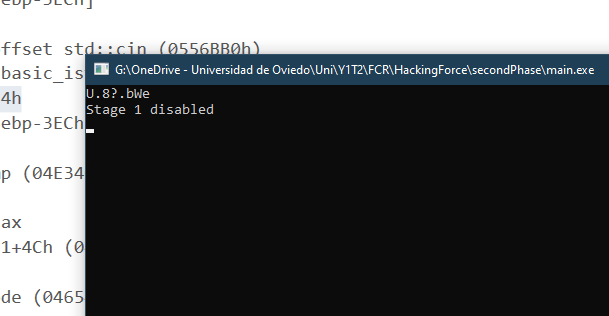
Accediendo al método principal, vemos que la clave de esta etapa se encuentra en una comparación de cadenas “strcmp”, en la dirección de memoria señalada en pantalla.



Al ir a esa dirección de memoria, vemos que se compara la cadena introducida con el penúltimo valor introducido en la pila.



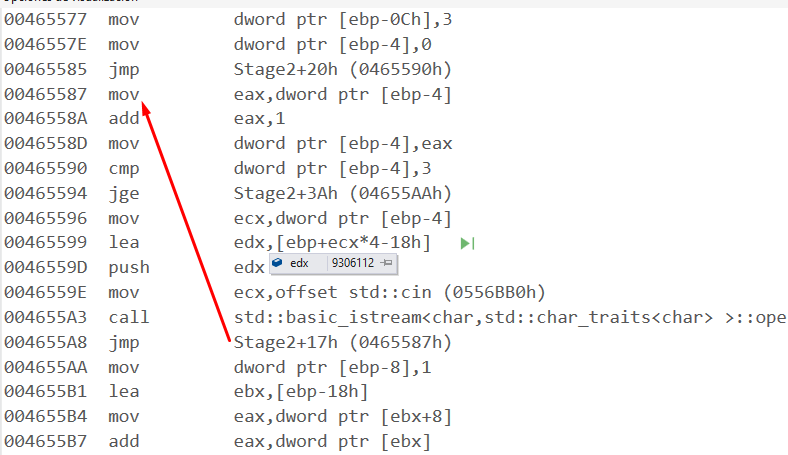
Podemos observar claramente que la cadena localizada en la posición “525A14h” es “U.8?.bWe”, nuestra contraseña para la primera bomba.



Lo introducimos en la consola y vemos que, efectivamente, hemos conseguido la contraseña correcta y desactivado la bomba de manera satisfactoria.

### Bomba 2

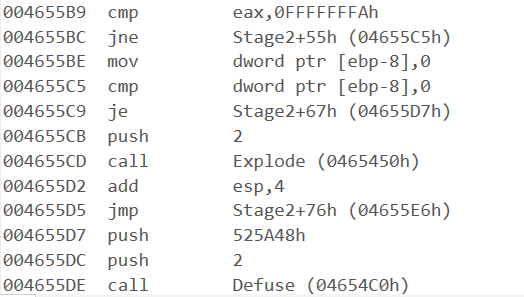
La segunda bomba es algo más complicado de resolver. Lo primero de todo, observamos que hay un bucle, que ocurre tres veces:

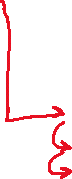


En este bucle, se leen tres números enteros de consola y se almacenan en un vector. Posteriormente, después de terminar de leer, se mueve la posición inicial del vector a “ebx”:

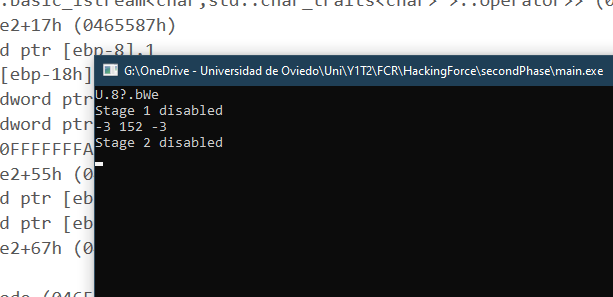
Después, se mueve añaden el primer y el tercer número del vector de la entrada a “eax”:



Por último se comprueba si ese número es igual o no a -6 (0xFFFFFFFAh). Si lo es, se salta a la desactivación. En la siguiente foto, se destaca en rojo el camino que sigue si eax es igual a 6, y en negro si no lo es.



Con todo esto, se concluye que una entrada válida para desactivar esta bomba es aquella en la que el primer y último número (de tres) deben sumar -6. El valor del segundo número es irrelevante.



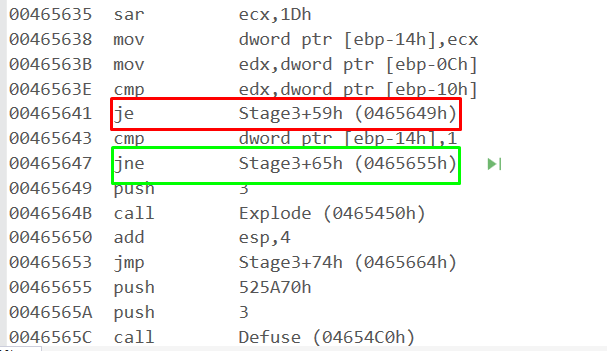
Lo introducimos en la consola y vemos que, efectivamente, hemos conseguido la contraseña correcta y desactivado la bomba de manera satisfactoria.

### Bomba 3

En la tercera y última bomba, se trabaja con máscaras y desplazamiento de bits, al igual que durante la fase de entrenamiento. Para resumir el proceso del procedimiento, existen dos condiciones que deben de cumplir los DOS números introducidos para que la bomba se desactive:

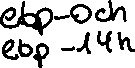
* El bit 29 del primero de ellos debe de ser 0.
* El bit 9 del primero y el 17 de segundo deben de ser diferentes.

La explicación de estos resultados es un tanto extensa y es mejor empezar por abajo, es decir, decidiendo cuándo se activa o desactiva la bomba:

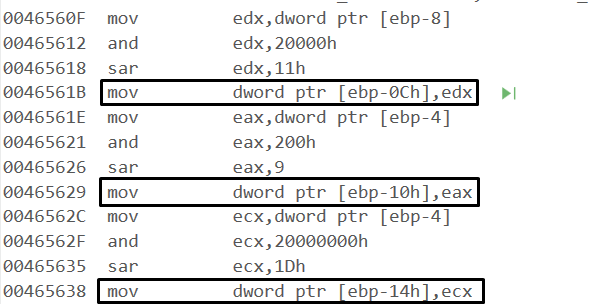
Si no se cumple alguna de las dos condiciones, obviamente la bomba “estalla”. En el código, las dos condiciones siguen una cadena lógica: primero se evalúa una y luego la otra:

En la fotografía, el primer jump NO se debe cumplir, mientras que el segundo SÍ. Así, el programa llega a la instrucción “00465655”, con lo que se termina desactivando la bomba.

Para que ocurra esto, se ha de cumplir lo siguiente:

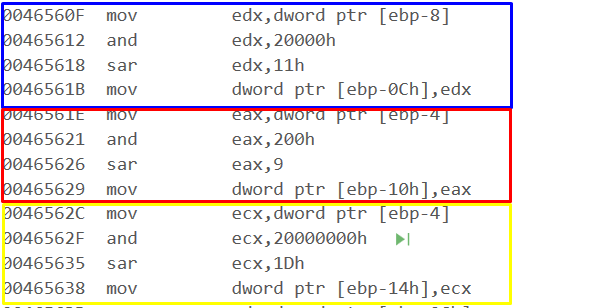


Para resolver el valor de estas direcciones de memoria, solo hay que subir un poco en el código:



Gracias a estas instrucciones, podemos transformar lo que teníamos antes a lo siguiente:



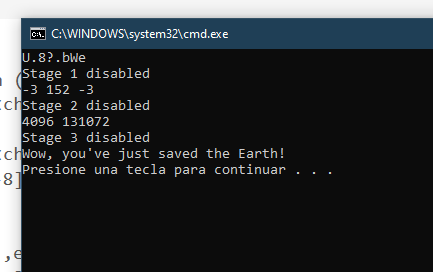
Ahora, solo tenemos que buscar dónde obtienen los registros sus valores, algo bastante sencillo:

De la anterior captura, se obtiene que:

* Resaltado en azul, edx es la máscara que se le aplica al segundo número en su bit 17, el cual se mueve posteriormente 17 posiciones a la derecha quedando en un solo dígito, ya sea 1 o 0.
* Resaltado en rojo, eax es la máscara que se le aplica al primer número en su bit 9, que de igual manera que antes se mueve hasta dejar un solo dígito.
* Resaltado en amarillo, ecx, es la cadena más grande, que se le aplica al primer número de nuevo y mueve su bit 29 (0x1Dh) y lo deja en un solo número.

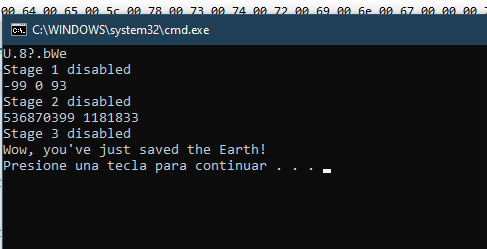
Ahora, podemos traducir los registros que teníamos antes a algo más legible:



A partir de esto, podemos fabricar números a medida de manera bastante sencilla, para probar nuestros resultados y desactivar la bomba final.

En este resultado, hemos utilizado el 4096 como primer número. Obviamente, su bit 23 no es uno, y su bit 9 tampoco lo es. En cambio, el bit 17 del segundo número sí que es uno, con lo que se cumplen ambas condiciones y la bomba se desactiva.

También podemos probar otros resultados que sigan cumpliendo ambas condiciones, como por ejemplo:

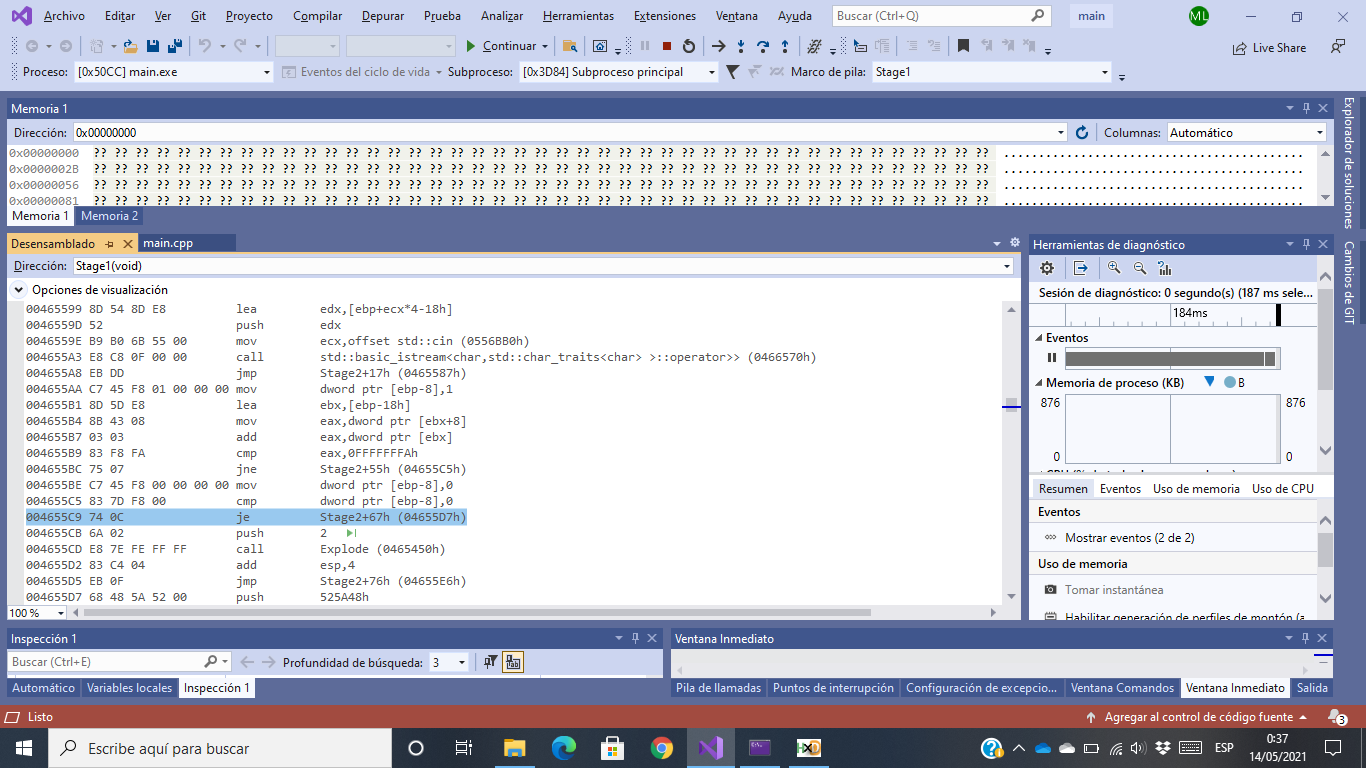


## Modificación del ejecutable

En esta segunda parte, lo que se pedía era modificar el ejecutable en sí para que, sea cual sea la contraseña introducida, el programa desactive la bomba y así “salvar el mundo”.

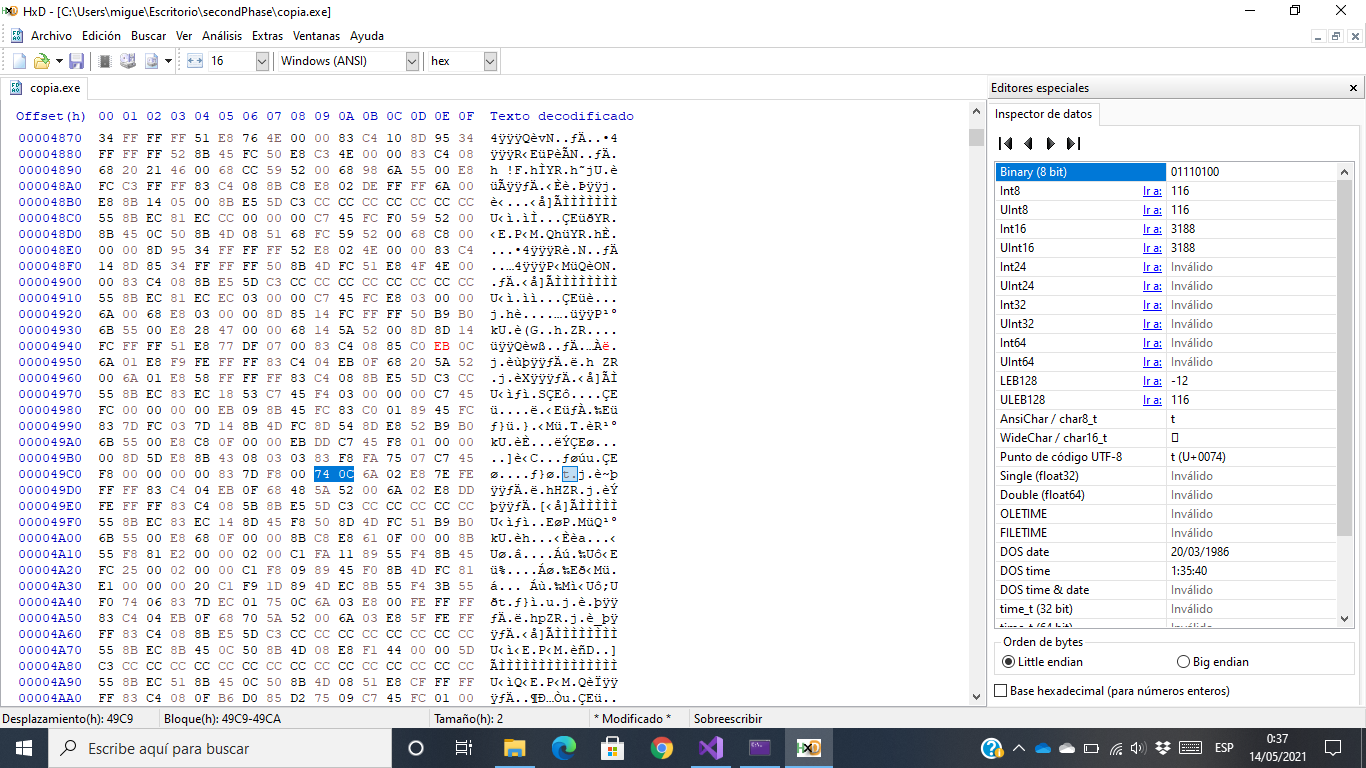
Para ello, se utiliza el programa “HxD”, que modifica los valores hexadecimales del código.  
Para saber qué valores modificar, primero tenemos que fijarnos en lo que hace el programa y cuándo salta al explode y reemplazar dichas instrucciones.

Para las dos primeras bombas, el proceso es similar: reemplazar el jump condicional que lleva al defuse cuando la contraseña sea correcta y modificarlo por un jump incondicional que siempre lleve ahí:



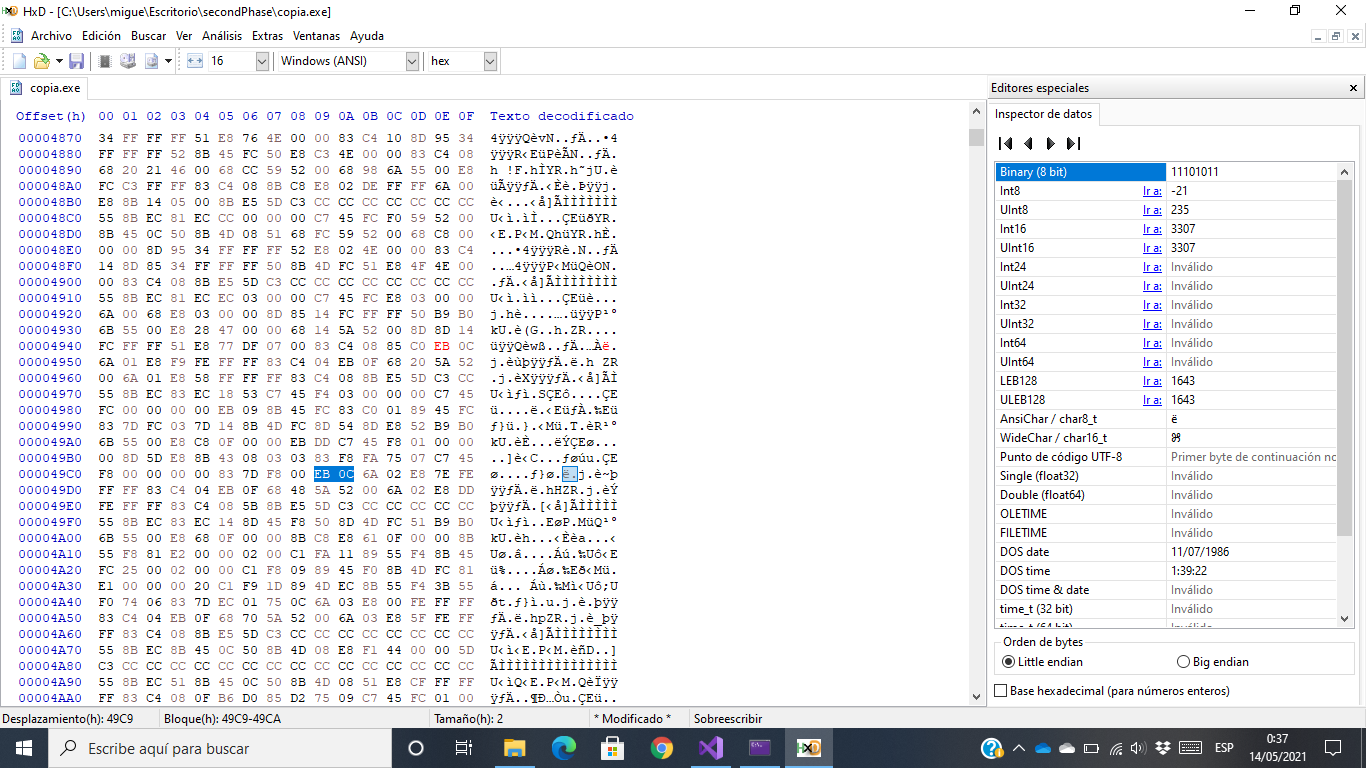


Observamos los valores en hexadecimal de las instrucciones cercanas y buscamos la instrucción en HxD:



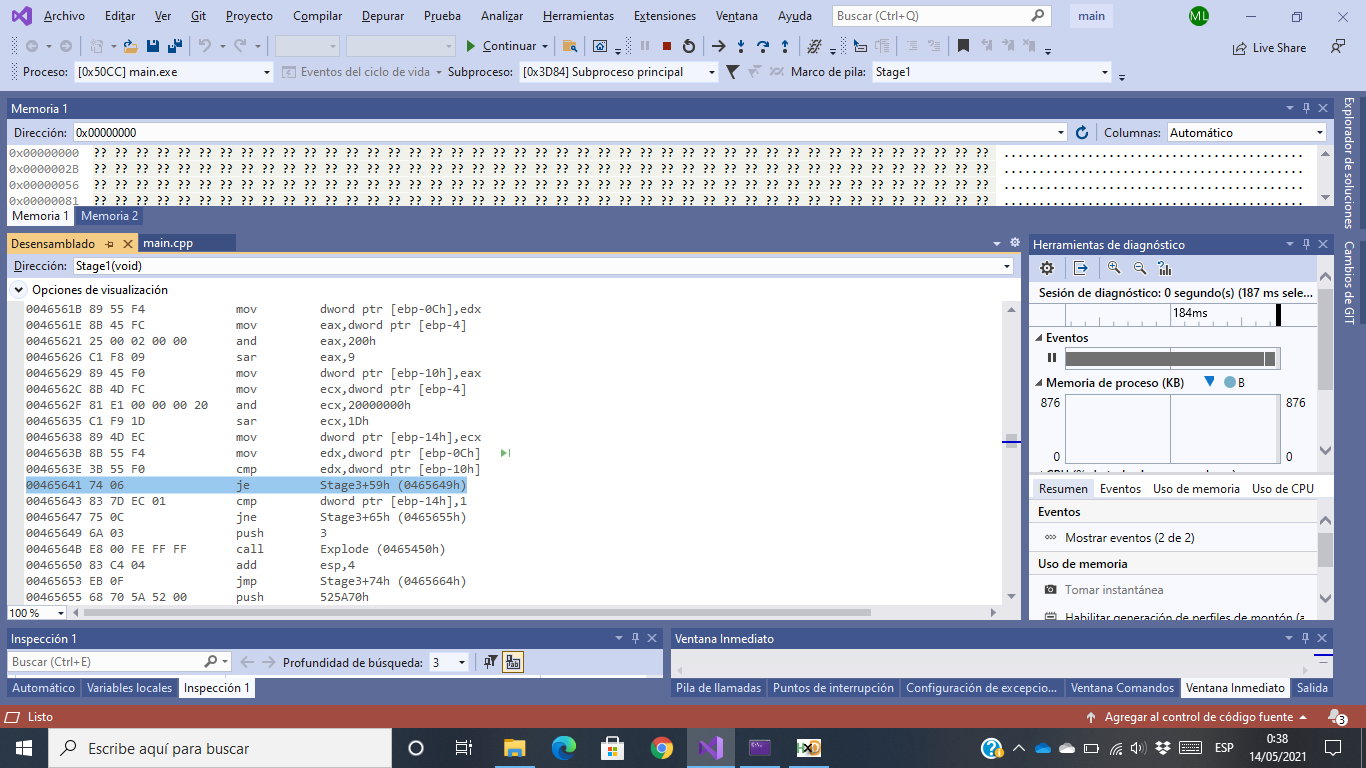


Finalmente, reemplazamos el valor por lo que queremos:



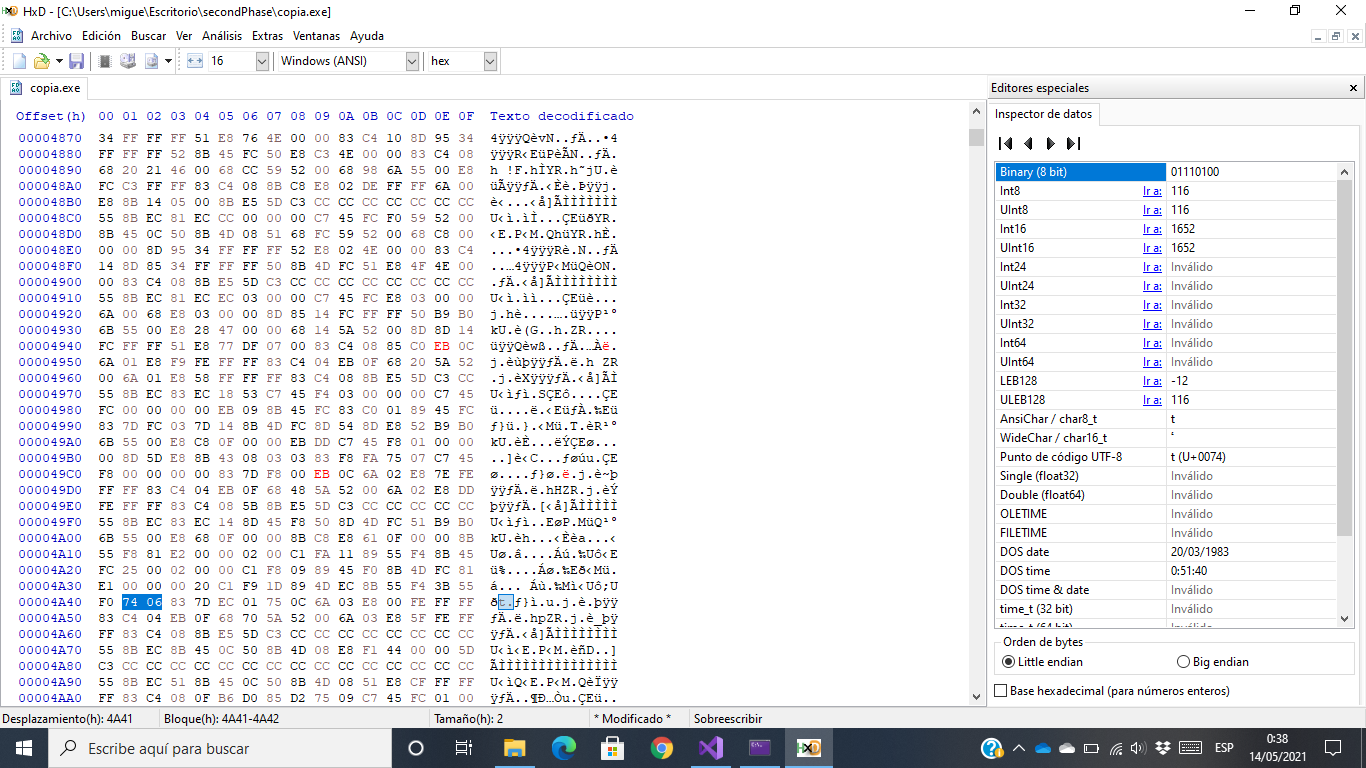


Igual con la segunda bomba, primero buscamos:



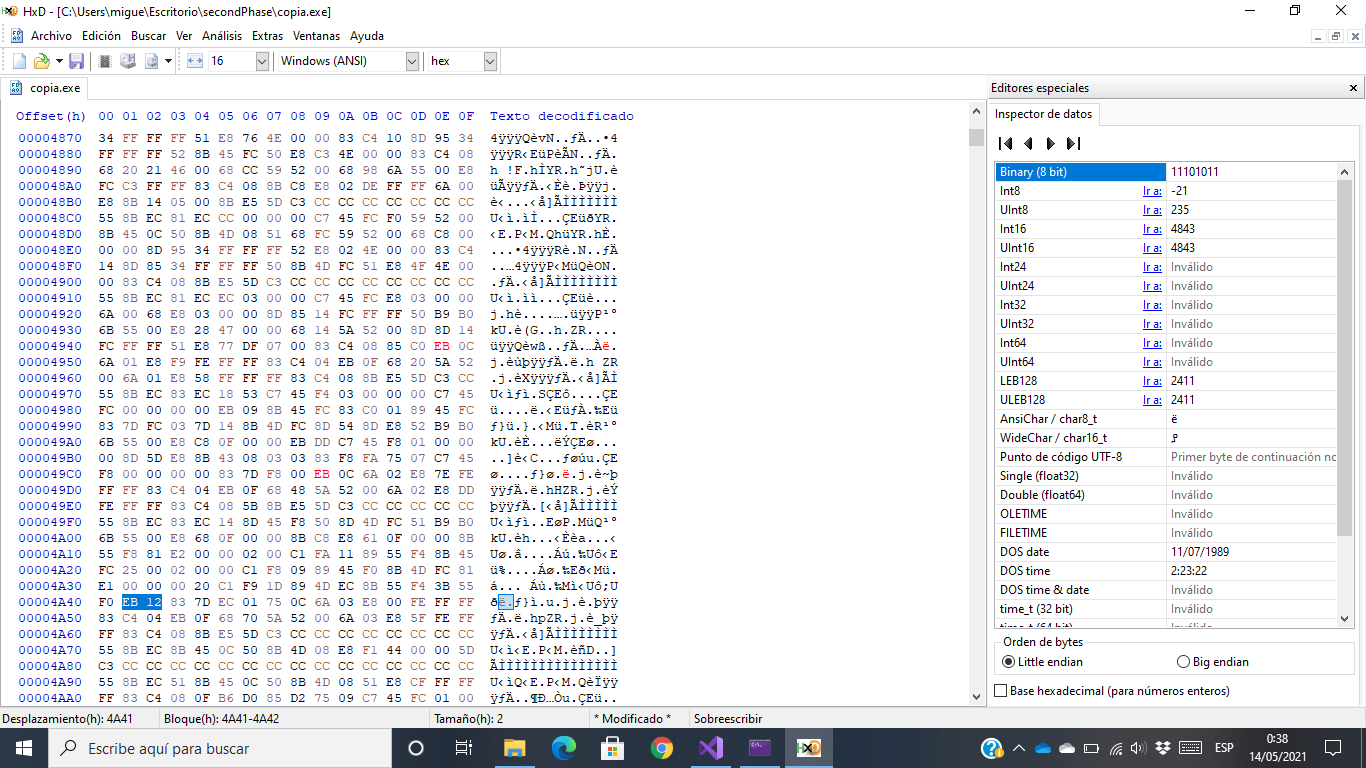


Luego encontramos:



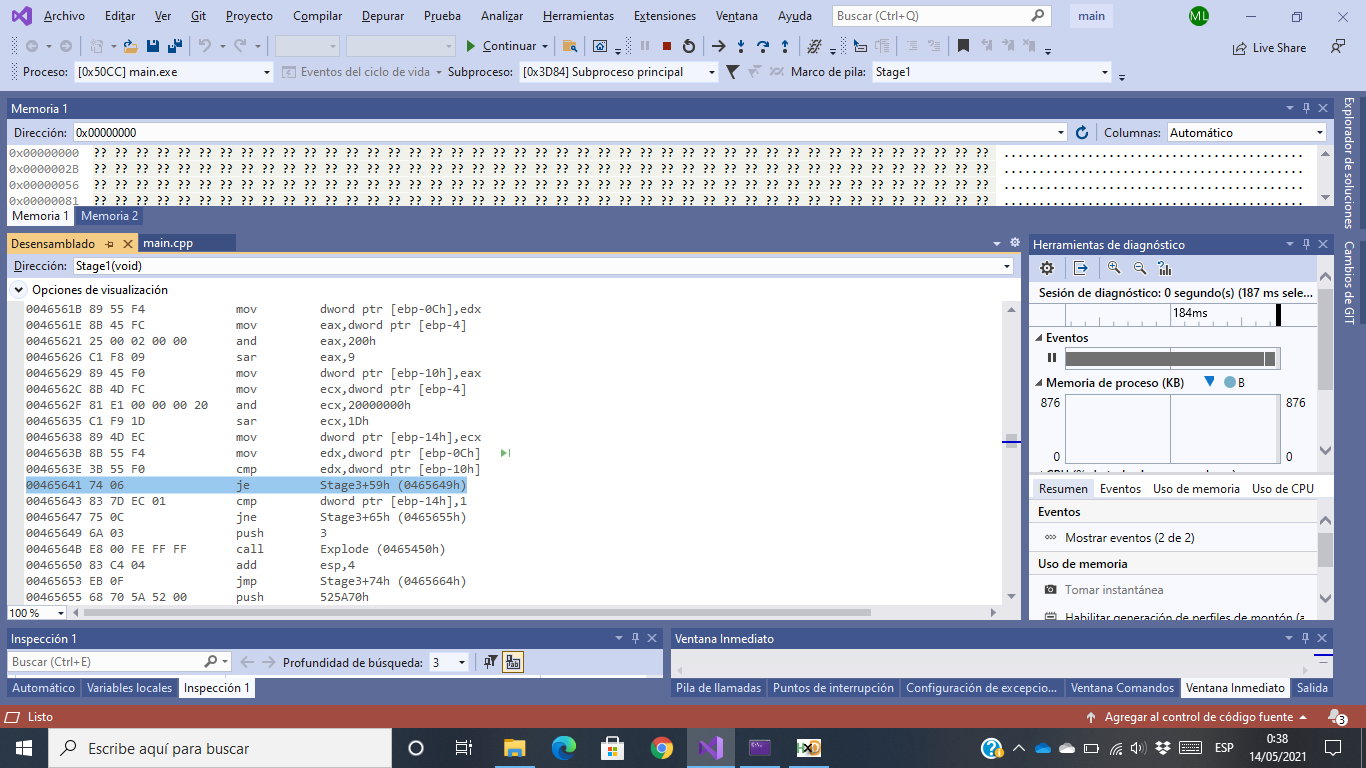


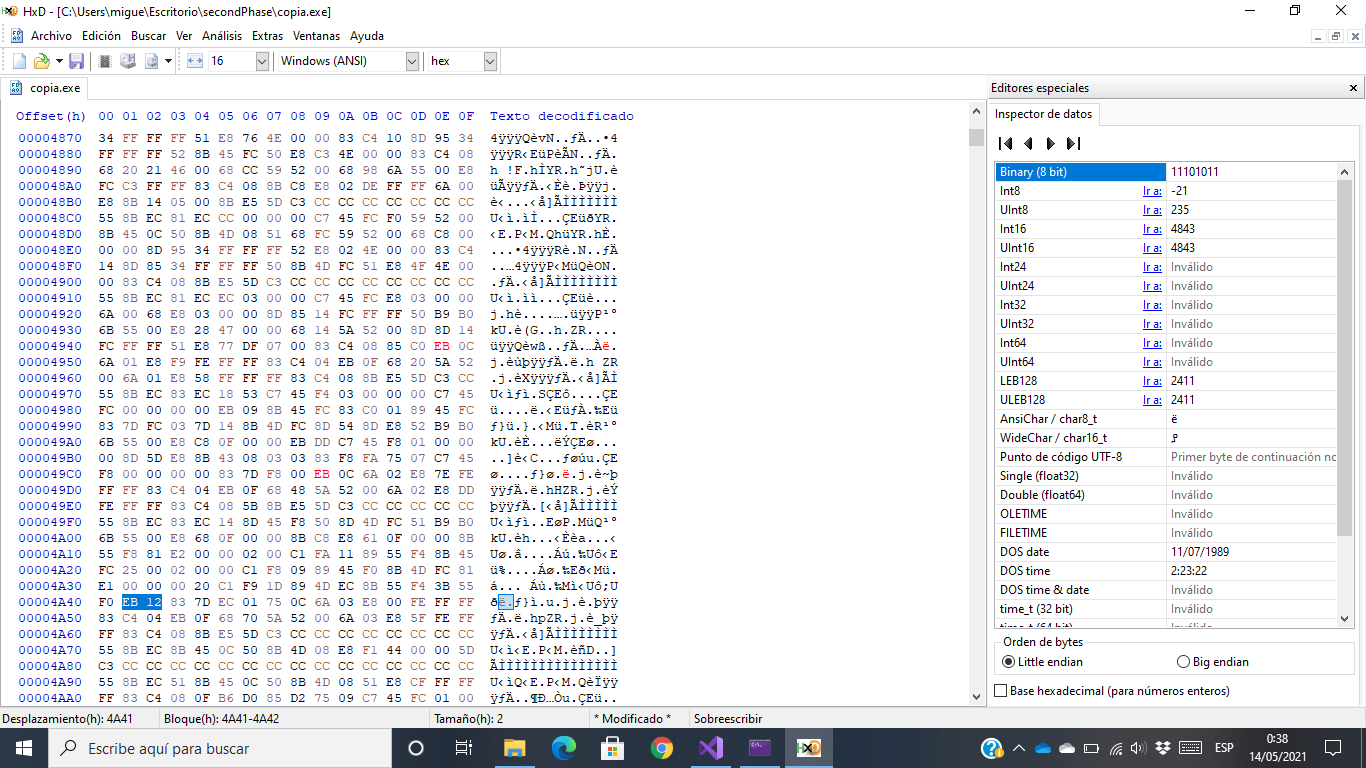
Y, por último, reemplazamos:

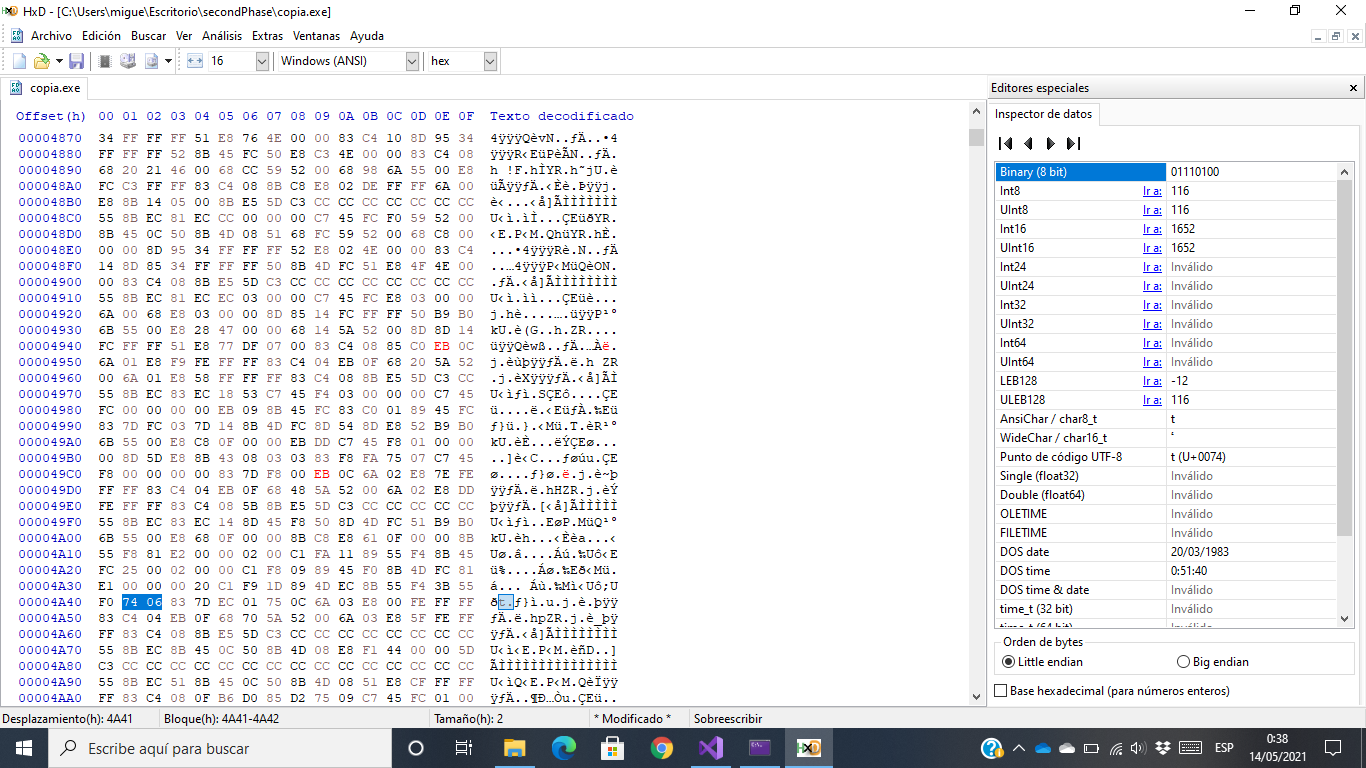


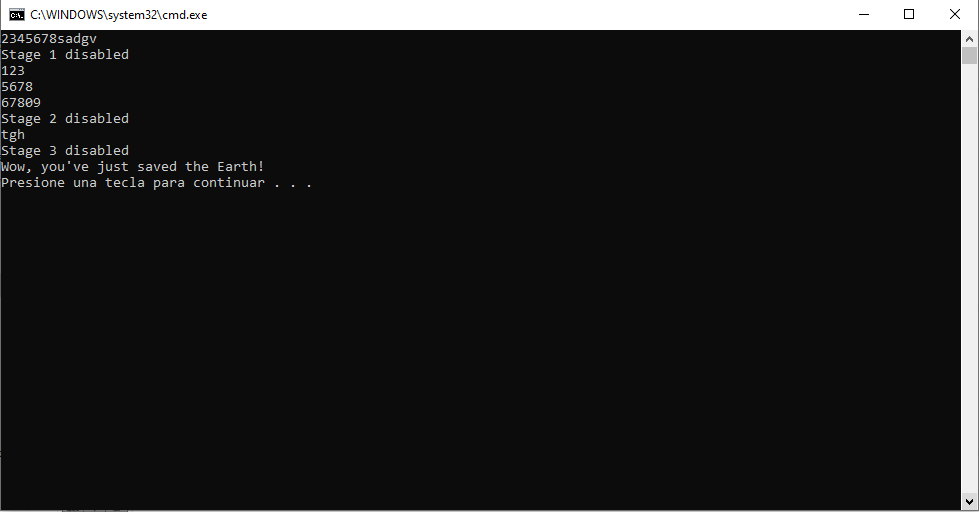
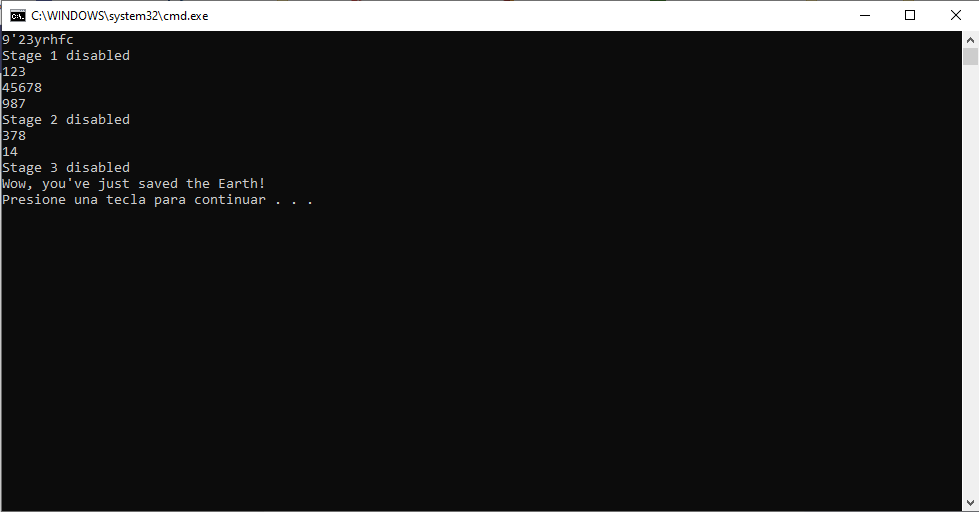


Para la tercera bomba, simplemente hay que cambiar un salto condicional a otro salto incondicional, esta vez variando la cantidad de instrucciones que se saltan para que redirija correctamente a Defuse():







Y ahora, ejecutamos el ejecutable modificado introduciendo contraseñas inválidas para comprobar que funciona sin importar la entrada:  


Y con esto, hemos salvado el mundo y hemos desactivado toda posible ofensa contra la humanidad.

# Reparto del trabajo

El trabajo se ha repartido de esta manera:

* De manera conjunta, se han resuelto las tres bombas con sus tres correspondientes contraseñas.
* Miguel del Riego: Modificación del ejecutable mediante HxD
* Juan Mier: Memoria y Wireshark.

En total, se han dedicado alrededor de 8 horas por persona al proyecto.