

@author Rodríguez López, Alejandro // UO281827

@author Fernández Ruiz, Pablo // UO282000

@author Cuesta Loredó, Celia // UO284095

@author Quirós Maneiro, Javier // UO285230

O-21-PL05-10

Ingeniería Informática en Tecnologías de la Información.

Índice

Condiciones para desactivación de cada fase	2
Fase 1:.....	2
Fase 2:.....	3
Fase 3:.....	4
Modificación del .exe	6
Nombre del Grupo.....	7
Repartición del trabajo	9

Condiciones para desactivación de cada fase

Fase 1:

Para completar exitosamente la primera de las fases, deberemos introducir una cadena de texto que coincida con una oculta en el código fuente. En el mismo, nos encontramos con una función `strcmp()` cuya función es comparar cadenas de texto. Uno de sus parámetros es la cadena que el usuario introduce en el computador mientras que el otro es la cadena con la que debe comparar. Para descubrir dicha cadena, observamos en el código fuente la primera orden `push` que envía el parámetro a la pila para ser posteriormente usado por la función. Si buscamos dicho valor en las direcciones de memoria llegaremos a la cadena que estará escrita en la derecha de la pantalla.

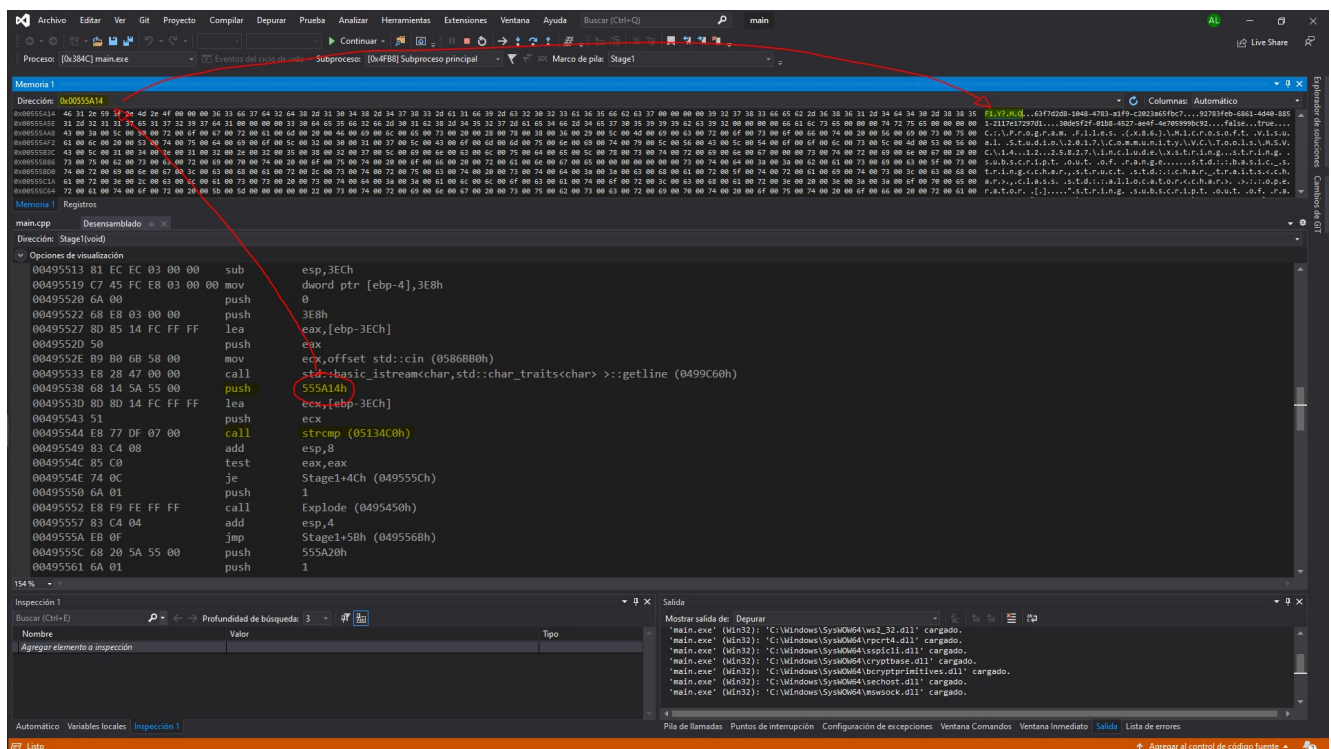


Ilustración 1 - Código fuente de la primera función. Marcados en amarillo y unidos por flechas; el parámetro, la dirección de memoria, y la contraseña a introducir.

En la imagen se puede ver marcado en amarillo la función `strcmp()`, la orden `push` en cuestión, la dirección buscada en la memoria y la cadena "F1.Y?.M.O".

Entrada válida: **F1.Y?.M.O**

Fase 2:

La segunda fase solicitará al usuario 4 números utilizando un bucle *for*. Más abajo en el código podemos observar 2 comparaciones *cmp* acompañadas por saltos *jne* y *je*. El segundo de ellos saltará la llamada a la función *Explode* y nos llevará hasta *Defuse* mientras que el primero se saltará la orden *mov dword ptr [ebp-8],0*.

Si nos fijamos aún más atentamente, descubriremos que justo después de la orden *mov dword ptr [ebp-8],0* hay un *cmp dword ptr [ebp-8], 0*. Dígase, estamos comparando una posición de memoria con un valor que le acabamos de asignar, por lo tanto, siempre se cumpliría la condición *je* que acompaña en la línea siguiente.

No obstante, la orden *mov dword ptr [ebp-8],0* no se ejecutará siempre, pues si el *cmp eax, 0FFFFFFFh* cumple la condición *jne*, la orden *mov dword ptr [ebp-8],0* se saltará, evitando así que se cumpla la condición *je* y forzando la llamada a la función *Explode*.

En conclusión, necesitamos que la condición *jne* **no** se cumpla. Para ello deberemos cumplir su contraria *-je-* y necesitamos que la resta entre el primer y segundo número sea 1.

```

Registers:
EAX = 00586B00 EBX = 00202000 ECX = 0019FF50 EDX = 00585138 ESI = 004CF8F0 EDI = 004CF8F0 EIP = 00495587 ESP = 0019FECC EBP = 0019FECC EFL = 00000202

0x0019FE8 - 00000000
Memoria | Registros
main.cpp | Descompilado
Dirección: Stage2(void)
Opciones de visualización
00495585 EB 00 jmp Stage2+20h (0495590h)
00495587 8B 45 FC mov eax,dword ptr [ebp-4] ≤ 1 ms transcurridos
0049558A 83 C0 01 add eax,1
0049558D 89 45 FC mov dword ptr [ebp-4],eax
00495590 83 7D FC 04 cmp dword ptr [ebp-4],4
00495594 7D 14 jge Stage2+20h (0495590h)
00495596 8B 4D FC mov ecx,dword ptr [ebp-4]
00495599 8D 54 8D E4 lea edx,[ebp+ecx*4-1Ch]
0049559E 52 push edx
0049559F B9 B0 68 58 00 mov ecx,offset std::cin (0586B80h)
004955A3 E8 C8 0F 00 00 call std::basic_istream<char,std::char_traits<char>>::operator>> (0496570h)
004955A8 EB D0 jmp Stage2+17h (04955B7h)
004955AA C7 45 F8 01 00 00 mov dword ptr [ebp-8],1
004955AD 8D 50 E4 lea ebx,[ebp-1Ch]
004955B0 8B 43 0C mov eax,dword ptr [ebx+0Ch]
004955B3 2B 03 sub eax,dword ptr [ebx]
004955B6 8B F8 FF cmp eax,0FFFFFFFh
004955B9 75 07 jne Stage2+55h (04955C5h)
004955BE C7 45 F8 00 00 00 mov dword ptr [ebp-8],0
004955C1 8B 4D F8 00 00 00 cmp dword ptr [ebp-8],0
004955C4 74 0C je Stage2+67h (04955D7h)
004955C7 6A 02 push 2
004955CA E8 7E FE FF FF call Explode (0495450h)
004955CD 83 C4 04 add esp,4
004955D0 EB 0F jmp Stage2+76h (04955E6h)
004955D3 68 4B 5A 55 00 push 555A4Bh
004955D6 6A 02 push 2
004955D9 E8 D0 FE FF FF call Defuse (04954C0h)
004955DE 83 C4 08 add esp,8
  
```

Ilustración 2 - Código fuente de la segunda fase. Marcados en amarillo las condiciones y operaciones importantes.

En la captura de pantalla aparece marcado en amarillo la condición del bucle *for* utilizado para recibir los valores y las condiciones. La condición *jne* que no se debe cumplir aparece señalada con un “Evitar”.

Entrada válida: 4 - 3 - 101 - 3

Fase 3:

Según indica el código fuente en el desensamblador, la fase 3 solicita del usuario 2 números. Trata cada uno de ellos de formas diferentes para después revisar si cumplen unas restricciones.

A continuación, se presentan 2 organigramas mostrando el tratamiento ordenado que se realiza a cada uno de los valores. Como resultado se obtienen no 2 valores, sino 3 ya que el segundo número introducido por el usuario se trata de 2 formas distintas. Nótese que para cualquiera de los 3 valores, pueden ser 0, 1 y ningún otro valor distinto.

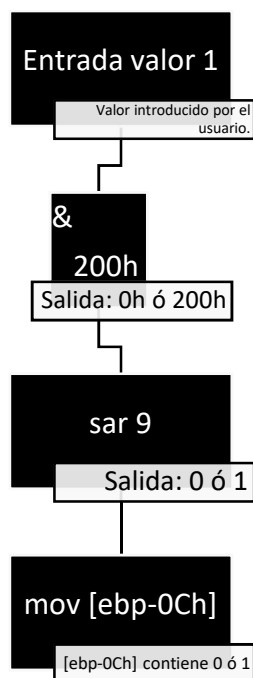


Ilustración 3 - Procesamiento aplicado al primer valor introducido.

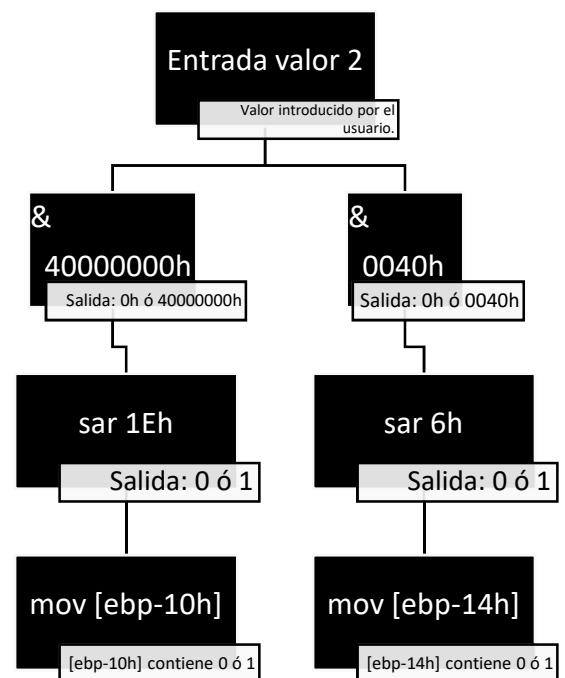


Ilustración 4 - Procesamientos aplicados al segundo valor introducido.

Respecto a las condiciones, la fase se puede superar por dos frentes distintos:

El primero de ellos sería que el valor en posición [ebp-14h] sea igual a 1. Para ello, se operaría con el segundo valor introducido, necesitaríamos que la operación <valor> & 0040h resulte en 0040h. Para ello, como segundo valor podemos introducir cualquier número con un 1 en el bit de valor 64. Por ello, podemos introducir números en el rango [64, 127].

El segundo de los frentes sería que el valor en posición [ebp-0Ch] sea distinto del que haya en posición [ebp-10h]. Dígase, una de las posiciones ha de contener un 0 y la otra un 1. Dicha restricción se puede cumplir si introducimos como primer número un valor con un 1 en el bit de valor 512 y como segundo número un valor con un 0 en el bit de valor 1073741824. También se puede obtener de forma viceversa. Con un 0 en el bit de valor 512 del primer número y un 1 en el bit de valor 1073741824.

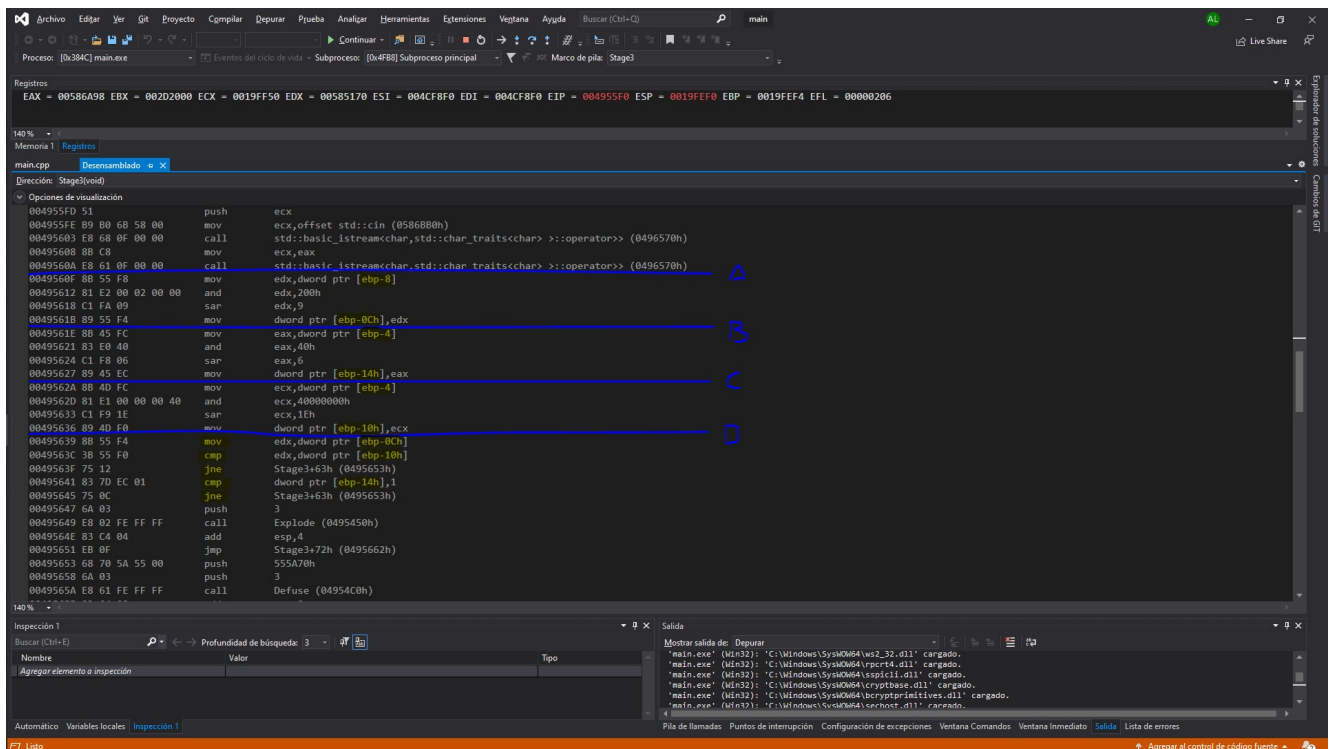


Ilustración 5 - Código fuente de la tercer fase. Separados por líneas azules los procesos aplicados a cada valor. Marcados en amarillo las direcciones de los valores y condiciones importantes.

En la captura de pantalla se presenta parte del código fuente de Assembly de la tercera función en el que se puede ver tanto cada tratamiento como las condiciones. La sección AB presenta el tratamiento realizado al primer número introducido –diagrama izquierdo-. La sección BC presenta el primer tratamiento realizado al segundo número –diagrama derecho, ramificación izquierda- y la sección CD presenta el segundo tratamiento al mismo número –diagrama derecho, ramificación derecha-.

Tras los tratamientos, se presentan marcadas en amarillo las condiciones descritas en la página anterior.

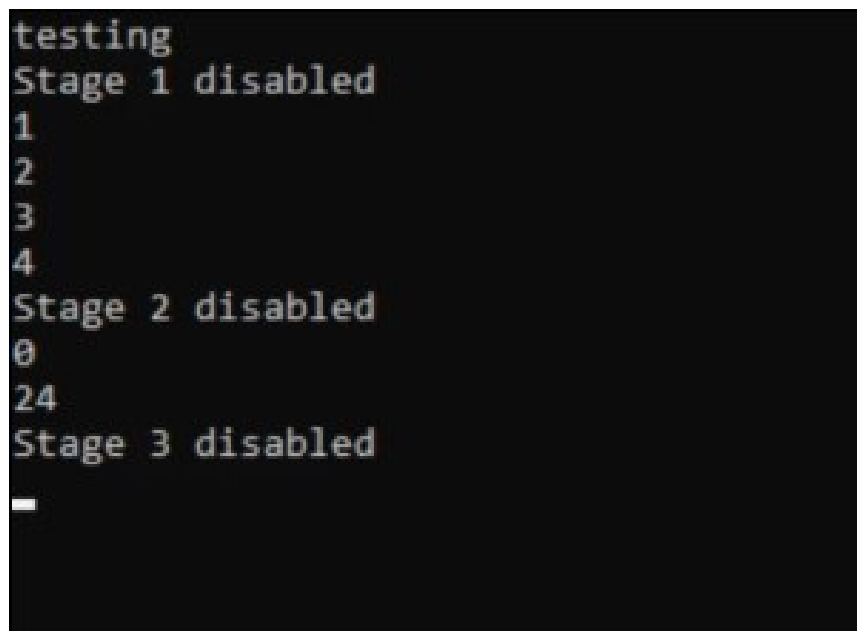
Entrada válida: 65

Modificación del .exe

Para editar el ejecutable original de forma que no haga falta conocer las condiciones necesarias para desactivar cada bomba hemos utilizado HxD. Tras abrir el .exe en HxD, hemos vuelto al código fuente de la primera función para encontrar la llamada a la orden *Explode()* para conocer su código máquina. Lo hemos buscado utilizando el buscador de HxD en el archivo .exe y al encontrarlo hemos sustituido el código máquina por E8 00 00 00 00. De esta forma, la llamada *call* sigue en el código, pero no se llamará a nada.

Para las siguientes funciones hemos repetido el mismo sistema. Buscar el código fuente de *Explode()*, hallarlo en HxD y sustituir por E8 00 00 00 00.

Como resultado podemos observar que las bombas se desactivan sin importar la información introducida.



```
testing
Stage 1 disabled
1
2
3
4
Stage 2 disabled
0
24
Stage 3 disabled
_
```

Ilustración 6 - Ejemplo desde consola de una entrada errónea que no detona la bomba debido a que se está ejecutando el archivo modificado

Debido a nuestra falta de experiencia con el programa, antes de concluir con el apartado, realizamos diferentes aproximaciones al ejercicio. Para evitar que las bombas detonen aun introduciendo las respuestas erróneas encontramos que se podía modificar el código de las ordenes *Explode()*, modificar las condiciones para que en lugar de saltar hasta *Explode()* salten hasta *Defuse()* en cualquier caso o forzar los operandos para que a las condiciones con saltos a *Defuse()* reciban valores que siempre cumplan las condiciones. Decidimos repartir las diferentes opciones a cada miembro del grupo para probar cuál de ellas era la más simple y concluimos que la mejor idea era realizar aquella que modificase menos el código fuente para evitar errores. En otras palabras, concluimos que la opción de cambiar las ordenes *Explode()* era la más segura.

Desde un principio, propusimos el cambio: E8 XX XX XX XX → 00 00 00 00. Pero no tuvimos éxito y a pesar de tratar arreglarlo terminamos por reorganizar nuestra planificación. Tras percatarnos de que todas las funciones *call* comenzaban por E8, propusimos el cambio final: E8 XX XX XX XX → E8 00 00 00 00.

Nombre del Grupo

Como último objetivo hallamos el nombre del grupo criminal que se encontraba en la trama capturada por WireShark.

Para comenzar, iniciamos WireShark y seleccionamos Wi-Fi. Nada más cargar la nueva pantalla con la información que viaja a través de nuestra red, ejecutamos la primera función del código de c++. Al hacerlo se realizará una comunicación entre nuestro computador y el servidor. Podemos volver a WireShark para añadir un filtro "http" en la cabecera de forma que sólo nos muestre los paquetes que contienen este protocolo. Entre los filtrados por pantalla, podremos ver dos tramas con protocolos http, la primera solicitando el nombre del grupo y la segunda respondiendo.

Si ampliamos la información de la segunda trama, podremos ver en el apartado "Line-based text data" el nombre del grupo que estamos buscando "Nice Avengers".

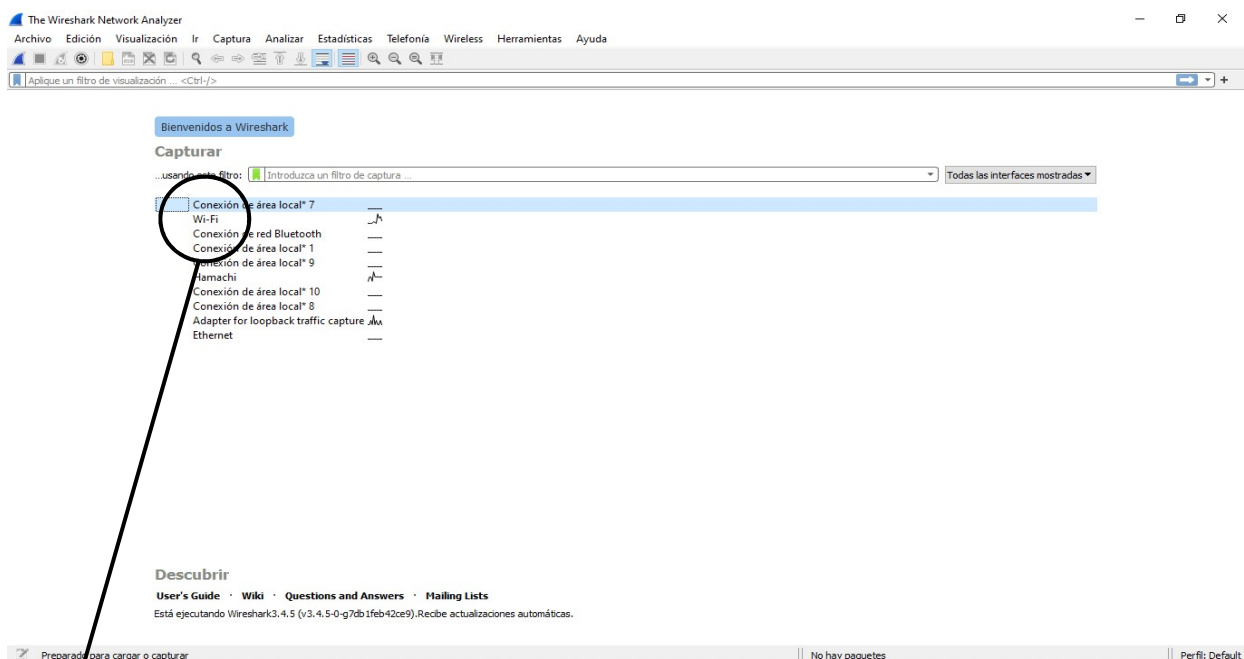


Ilustración 7 - Captura de pantalla del programa WireShark donde se puede ver la selección de Wi-Fi para hallar la trama.

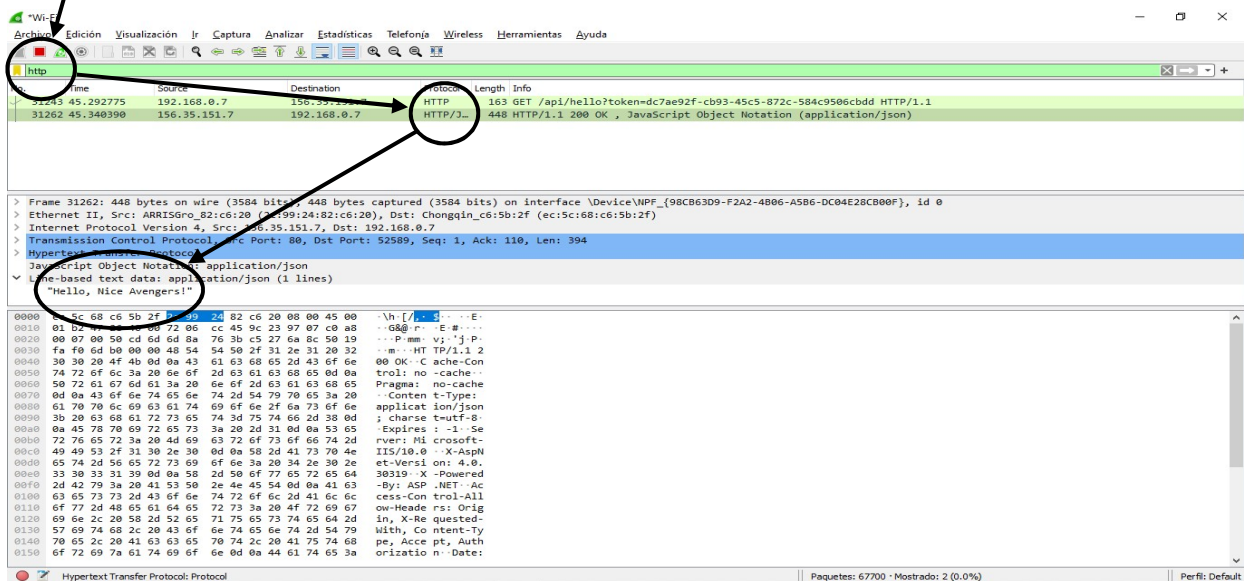


Ilustración 8 - Captura de pantalla del programa WireShark donde se puede ver la trama seleccionada y el nombre del grupo. "Nice Avengers"

Reparto del trabajo

A diferencia de la primera parte, en la que cada miembro podía realizar una parte del trabajo porque no importaba el orden en el que se realizasen las funciones, en esta segunda parte tuvimos que reorganizar nuestra estrategia de trabajo porque para poder comenzar con una fase deberíamos primero haber completado la anterior. Por lo tanto, acordamos que cada uno dedicaría el tiempo que tuviese disponible a resolver la función que tuviésemos en el orden. Si hubiese dos o más componentes trabajando en la función realizarían con sus objetivos en una llamada online y reportarían todos los avances a un chat compartido que tendríamos los miembros del grupo. De esta forma maximizamos -en la medida de lo posible- la comunicación entre alumnos y el aprendizaje colectivo.

No consumió tanto tiempo el hecho de buscar las respuestas correctas, sino la búsqueda de los motivos por los cuales las entradas que estábamos aportando eran correctas.

De entre todas las partes del trabajo, la primera fase fue la que menos tiempo nos consumió ya que a poco de comprender que justo antes de llamar a una función `strcmp()` se envían sus parámetros a través de pila -con órdenes *push*- pudimos hallar la respuesta. La segunda y tercera fase, sin embargo, consumieron algo más de nuestro tiempo. Respecto a las partes de hallar el nombre del grupo y de modificar el código fuente del .exe para que ninguna de las bombas detone aun con las respuestas incorrectas supusieron menos compromiso del que esperábamos, el trabajo realizado con WireShark fue directo y fácil de aprehender. Respecto al uso del programa HxD, a pesar de no haberlo utilizado nunca, no nos supuso mucho coste comprender su funcionamiento y sistema.

Solamente respecto al trabajo realizado con WireShark y HxD, decidimos que lo mejor sería realizarlo todos a la vez para que ninguno perdiese el contenido del objetivo.

A continuación, se presenta una tabla con el tiempo invertido aproximado por alumno:

Fase/Programa	Tiempo / Alumno	Tiempo Total	Tiempo Total	Tiempo Total
1	1.25	5	30	40
2	3.75	15		
3	2.5	10		
WireShark	1	4	10	
HxD	1.5	6		