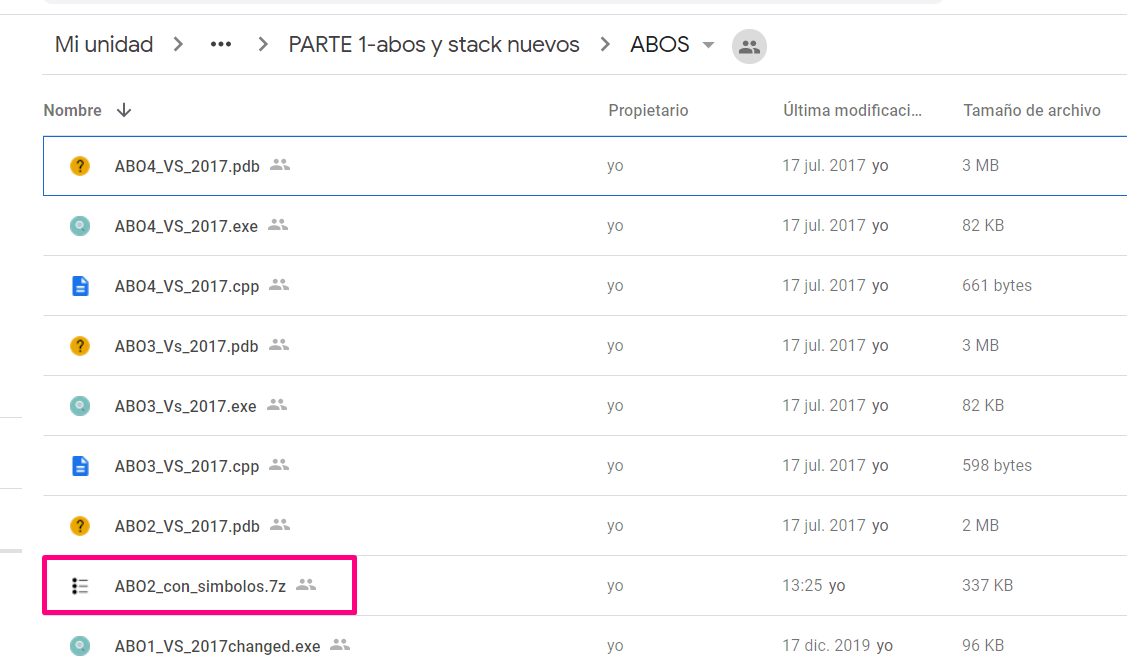
EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

GRATUITAS (PARTE 7)

Bueno ya habiendo visto y entendido el shellcode y el resolver que es lo más complejo, seguiremos con al análisis y resolución del abo2 en GHIDRA.

Descargamos el ejecutable del ABO2, subi una nueva versión al google drive, ya que la vieja no tenia los símbolos correctos, así todo coincide.



Allí tenemos dentro del archivo comprimido 7z el ejecutable con extension .exe y los archivos de símbolos .pdb y el código fuente extensión .c

https://drive.google.com/file/d/1ka2\_0V51vew\_ZjZ0hprHxFmJPCCL8KO7/view?usp=drivesdk



Como salio la version final 9.1.1 de GHIDRA la descargare para actualizarme.

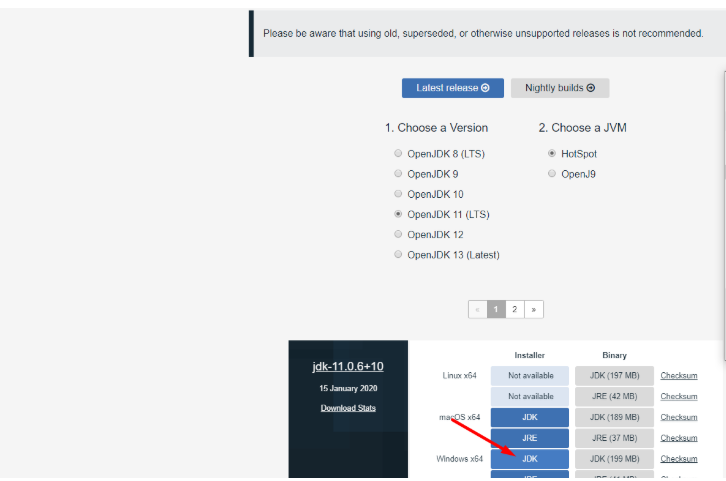
<https://ghidra-sre.org/ghidra_9.1.1_PUBLIC_20191218.zip>

Por supuesto si cuando lean esto hay una version mas nueva cambiará el link, la pagina es

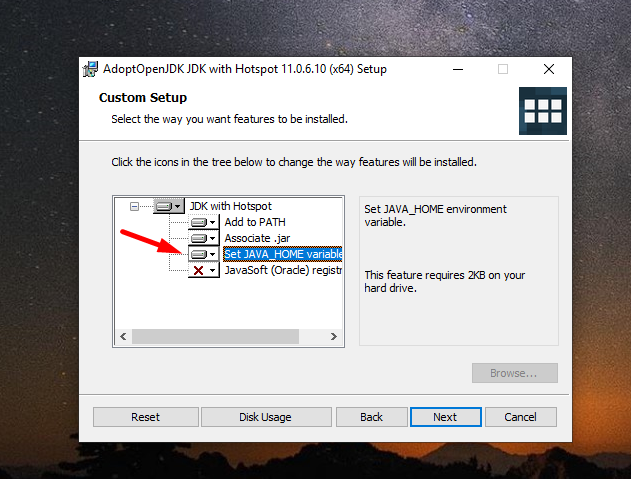
<https://ghidra-sre.org/>

el JDK bajo la versión 11 desde acá

<https://adoptopenjdk.net/archive.html?variant=openjdk11&jvmVariant=hotspot>



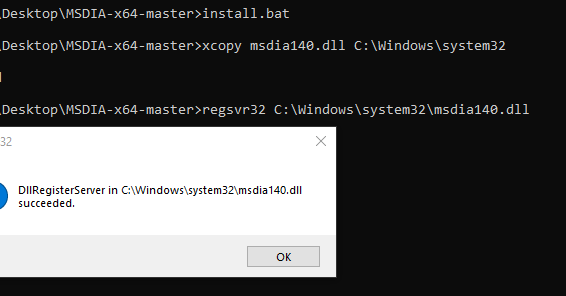
De allí se baja el archivo con extensión MSI, lo instalo, cambiando a la opción para que agregue a la environment variable su PATH.



Si no lo tenia de antes, agrego el dia sdk.

<https://github.com/MalwareTech/MSDIA-x64>

Ejecuto en una consola como administrador el bat



El bat solo copia el archivo msdia140.dll que está dentro del archivo comprimido que descargamos a la carpeta system32

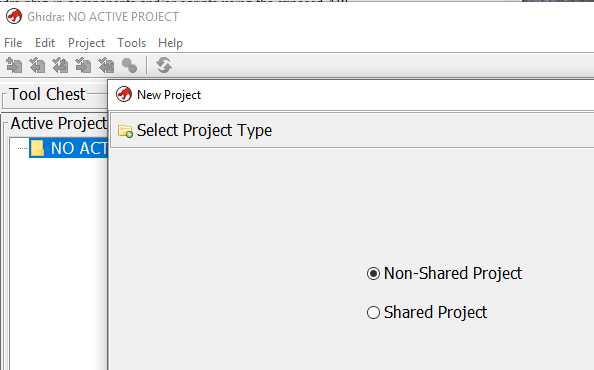
xcopy msdia140.dll %systemroot%\system32

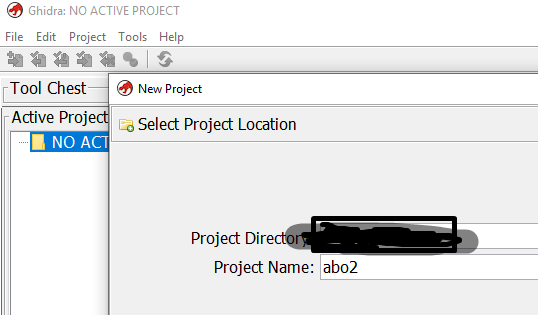
y luego lo registra.

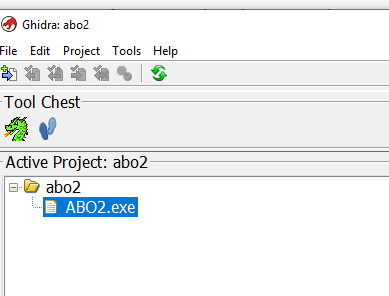
regsvr32 %systemroot%\system32\msdia140.dll

También una vez copiado a system32 se puede registralor arrastrando el archivo msdia140.dll y soltandolo sobre el archivo regsvr32.exe que está allí en la misma carpeta.

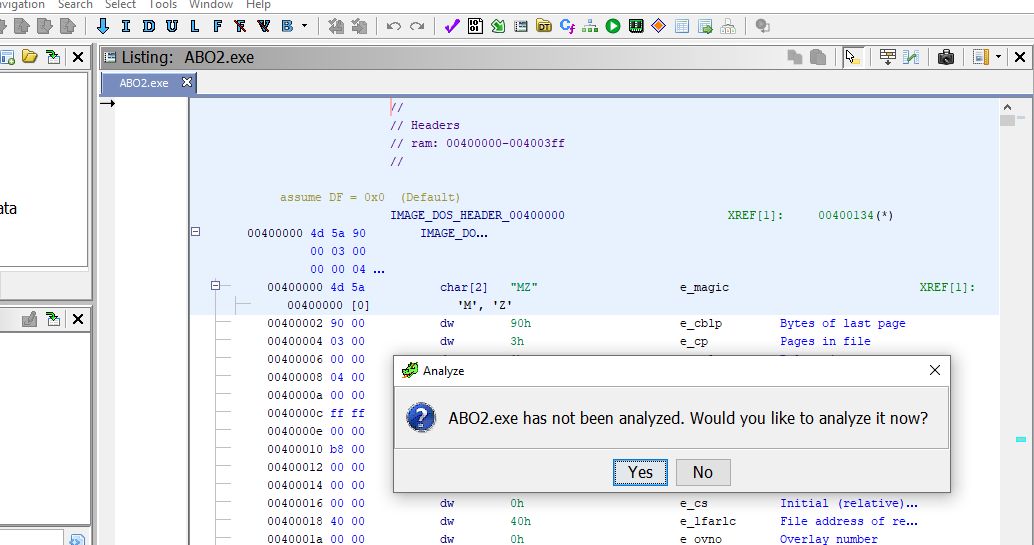
Bueno ya arranco voy a FILE-NEW PROJECT y creo uno nuevo.

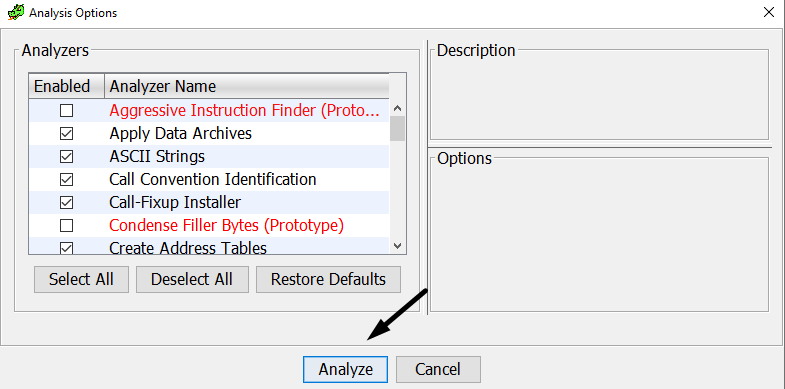




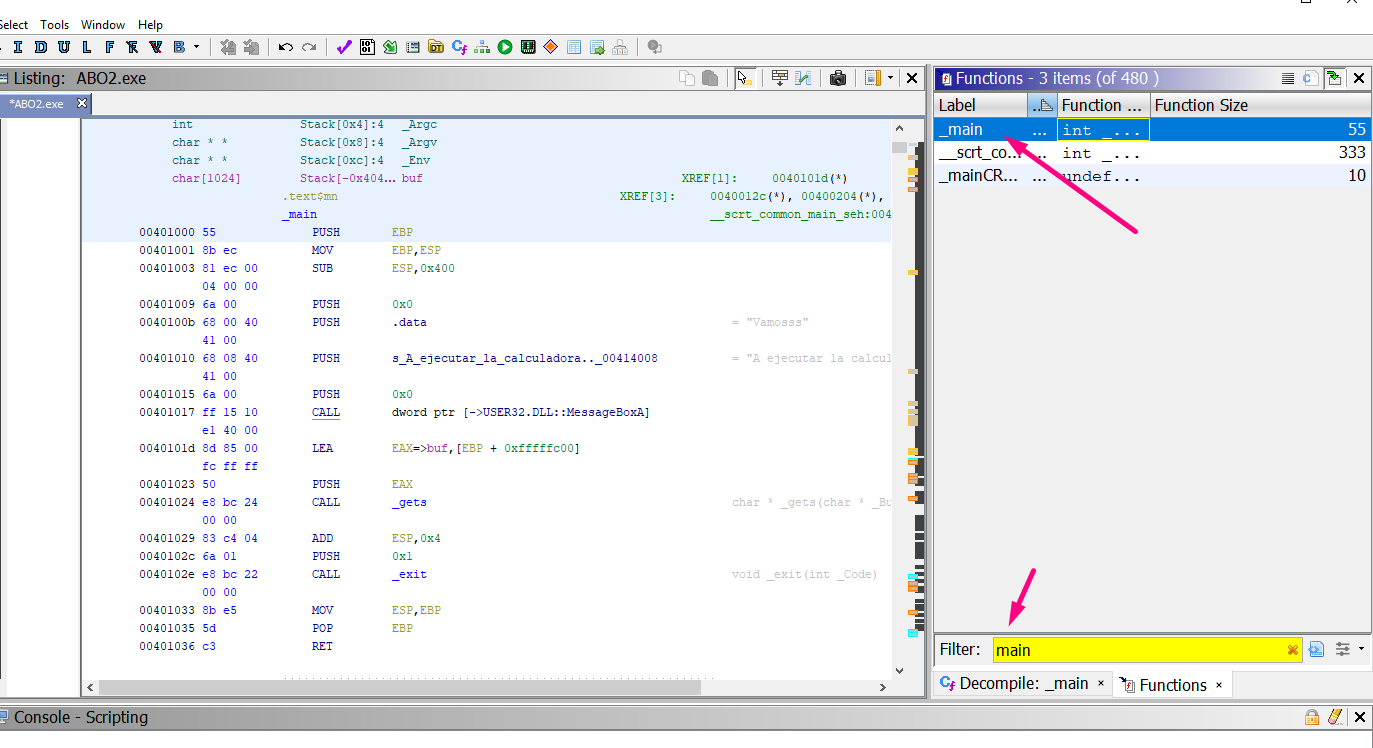


Hago doble click en el nombre del exe.

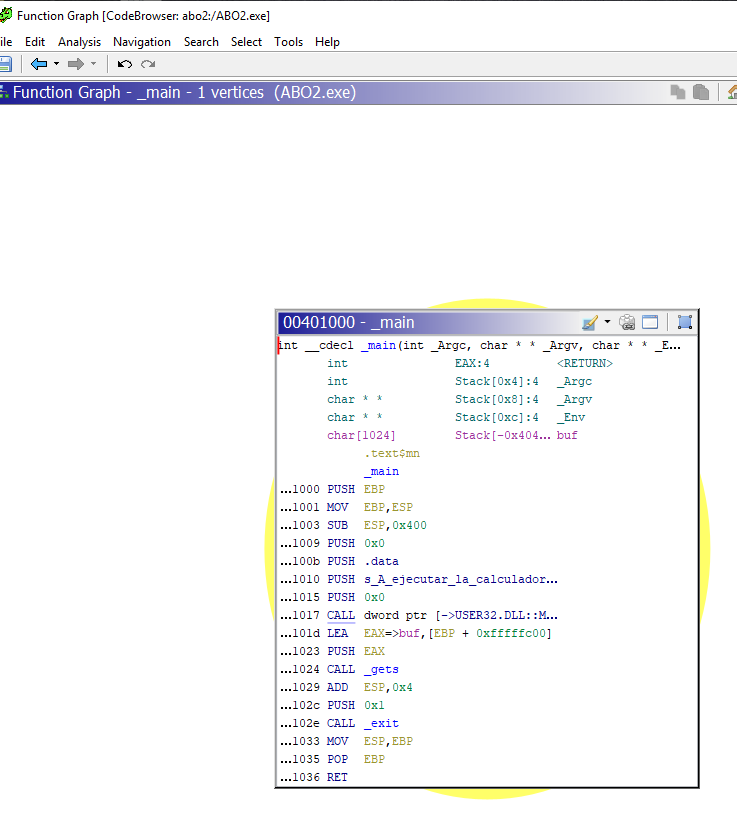




Yendo el menú WINDOW-FUNCTIONS, tengo abajo un filtro para tipear la funcion que quiero buscar, en este caso buscaré el main, en la lista de funciones hago doble click en main que aparece allí.

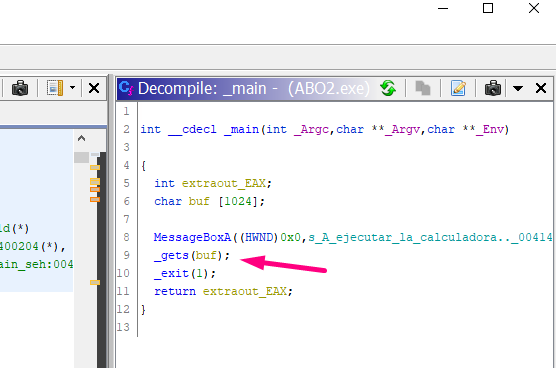


Si vamos a WINDOW-FUNCTION GRAPH, como es un solo bloque no hay mucho más que mostrar.



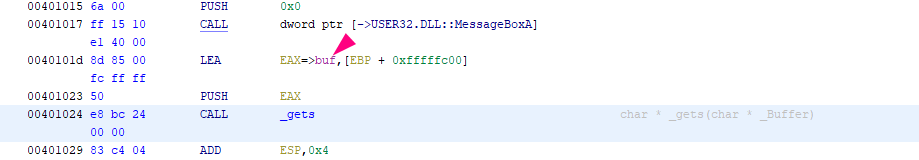
Acá parecería que este caso sería igual que el ABO1, pero ya veremos que no analicemos.

Podemos decompilar con WINDOW- DECOMPILE o CTRL+E.

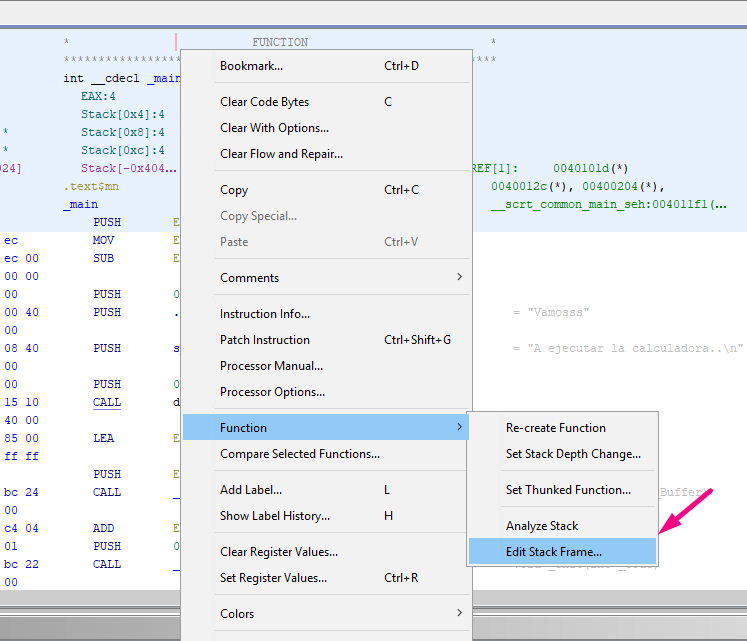


Vemos que hay un buffer de 1024 bytes.

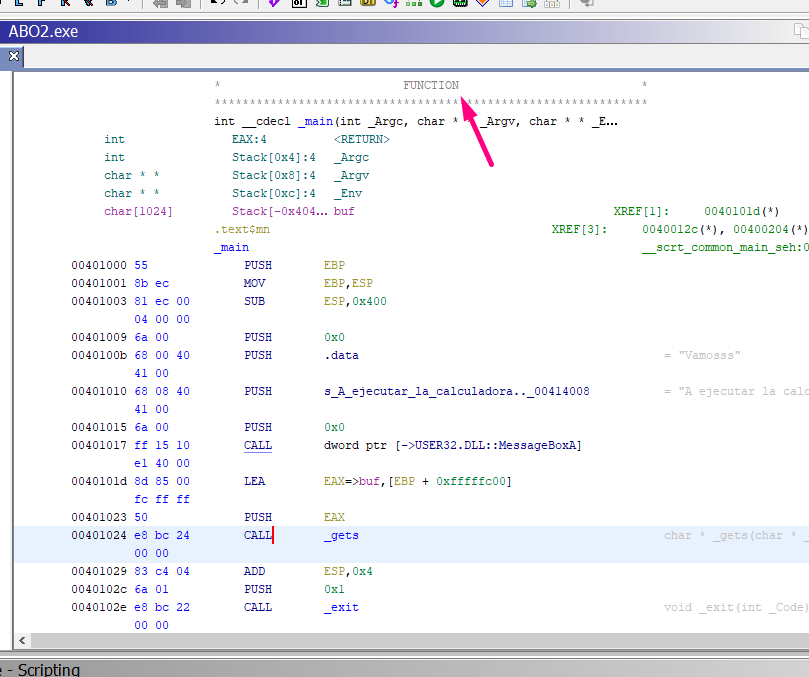
La dirección de buf se pasa a EAX, y se PUSHEA, será el argumento de la funcion gets.



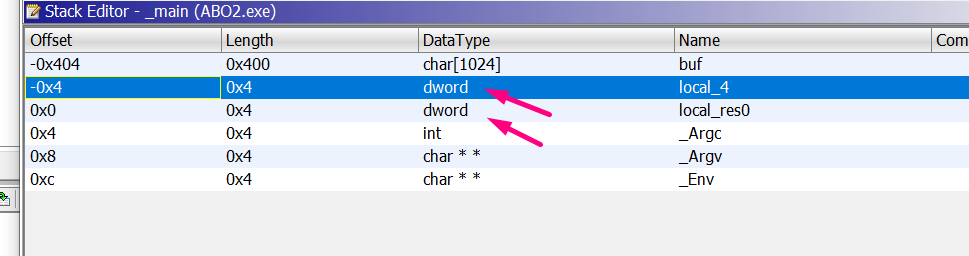
Por supuesto el buffer llamado buf se llenara y pasando los 1024 bytes comenzará a desbordar.



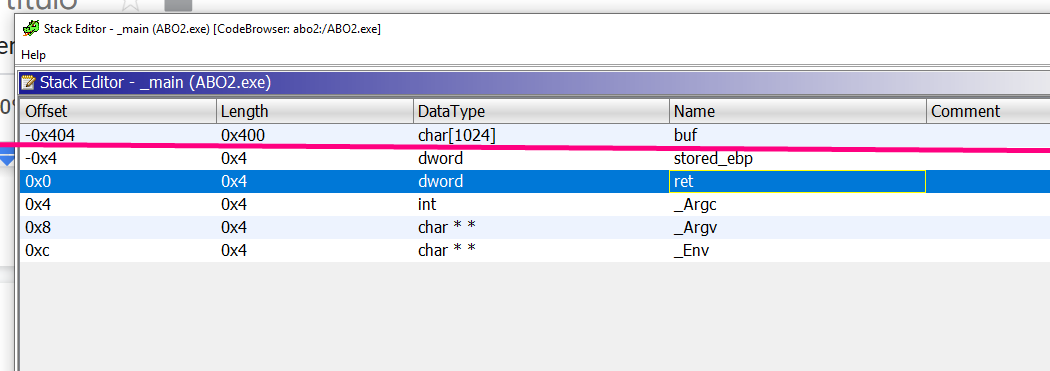
Este menú aparece al hacer click derecho en la palabra FUNCTION.



Apretando la tecla B podemos ir cambiando el tipo de datos, donde debería estar guardado el stored ebp y el return address para que sean DWORDS.

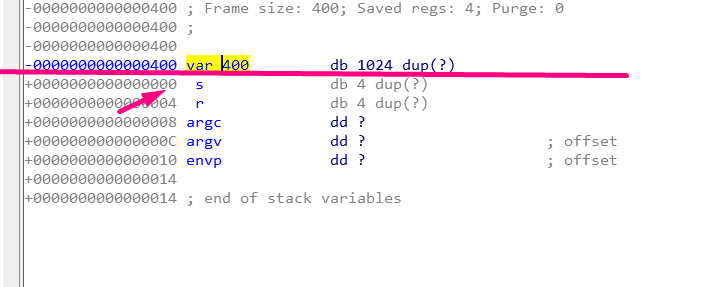


Recordemos que GHIDRA tiene una forma muy particular de mostrar las variables en el stack



Mientras que siempre en funciones basadas en EBP, se toma el mismo como referencia con lo cual el horizonte debería estar en la línea roja y ser 0, aquí al tomar como referencia el valor de ESP al inicio de la funcion, queda 4 bytes desfasado de lo que estamos acostumbrados a trabajar con IDA o otros desensambladores en GHIDRA el STORED EBP esta en -4.

Aquí lo vemos en IDA correctamente, **s** que es el STORED EBP está en 0 ya que alli estara el horizonte.(recordemos luego de hacer PUSH EBP que guarda el STORED EBP, se setea EBP de la funcion actual al valor actual de ESP que es justo el de la línea roja en el IDA, que llamamos HORIZONTE pues se toma como referencia)



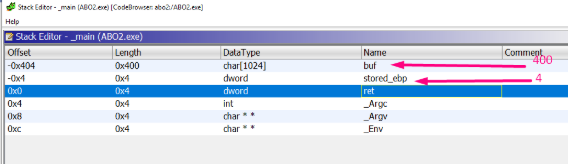
De cualquier manera en ambos vemos que hay que enviar 0x404 bytes y luego el valor para pisar el return address.

En GHIDRA el offset de buf es -0x404 y el de stored ebp es -4, por lo tanto si restamos ambos tenemos el largo de buf que es 0x400

Le sumamos los 4 bytes del stored ebp y luego está el return address asi que seria.

**payload = b"A" \* 0x404 + struct.pack("<L",0x41414141)**

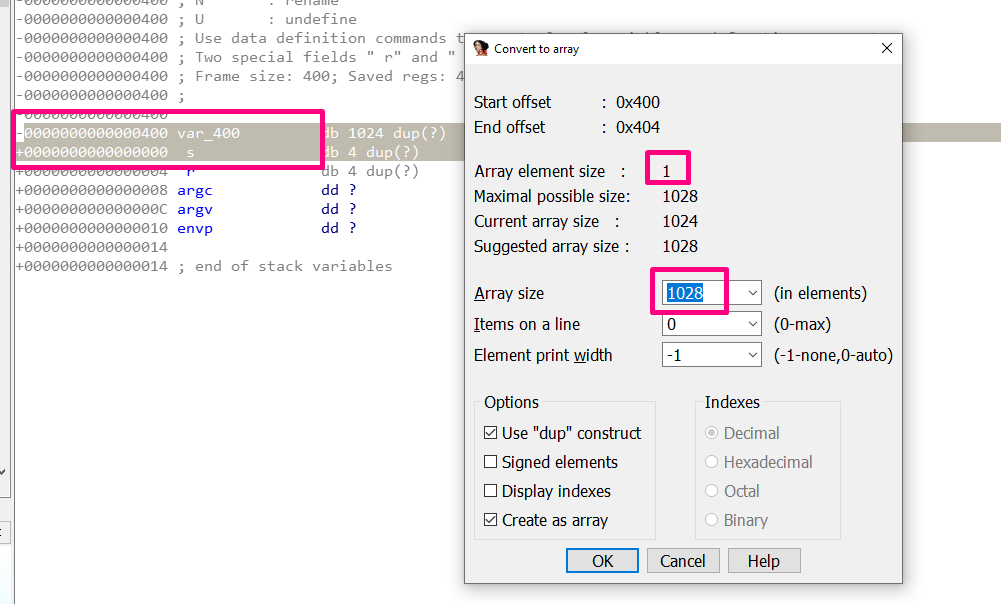
Si quisiéramos pisar el return address con 0x41414141.



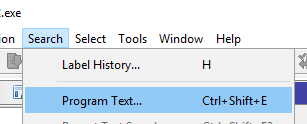
En IDA

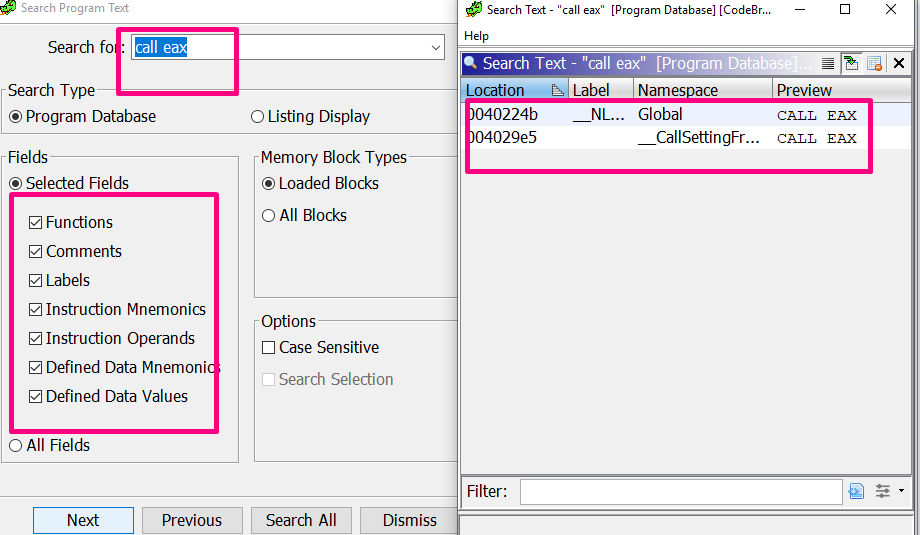
Marco la zona a rellenar, y hago CLICK DERECHO - ARRAY y me muestra element size del array 1 y largo 1028

O sea 1\*1028 = 1028 que son 0x404 bytes hexa, luego está el return address.

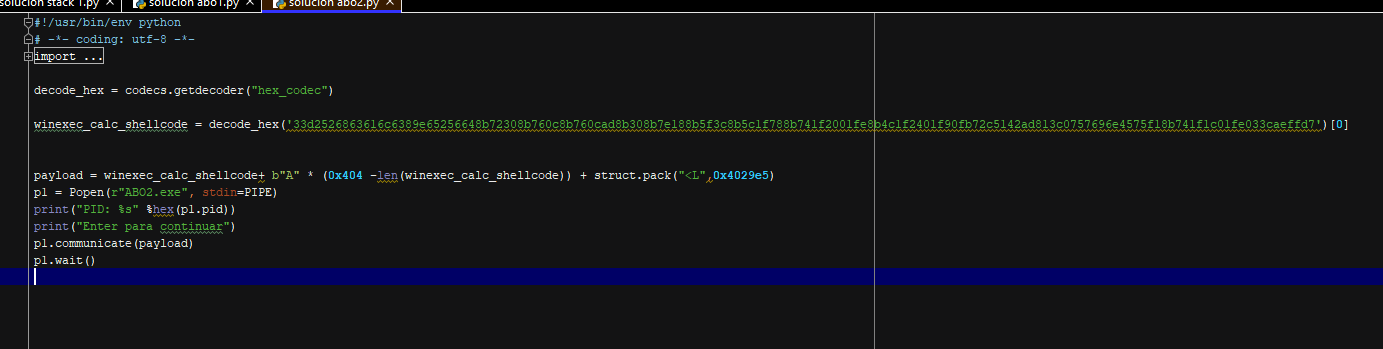


Por lo tanto aparentemente sería parecido al ABO1 solo me faltaría encontrar el CALL EAX para saltar a ejecutar el shellcode.





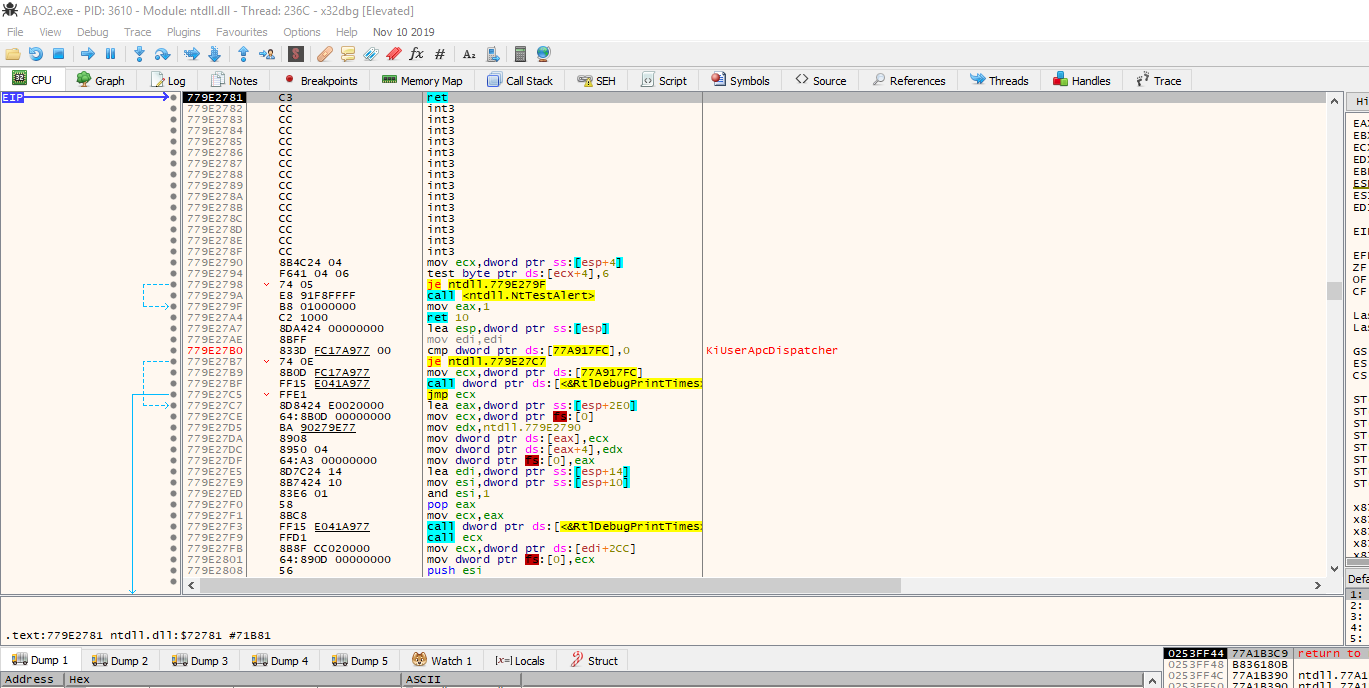
Alli encontre un par de CALL EAX buscando por texto.

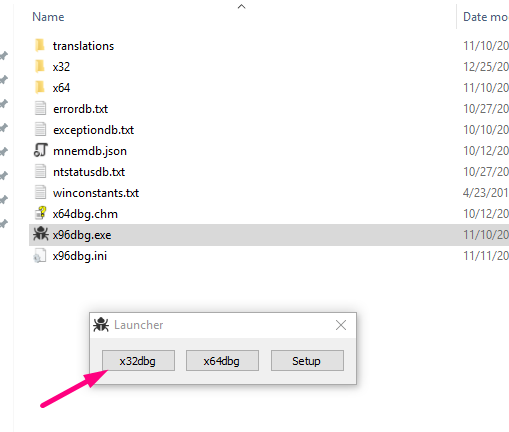


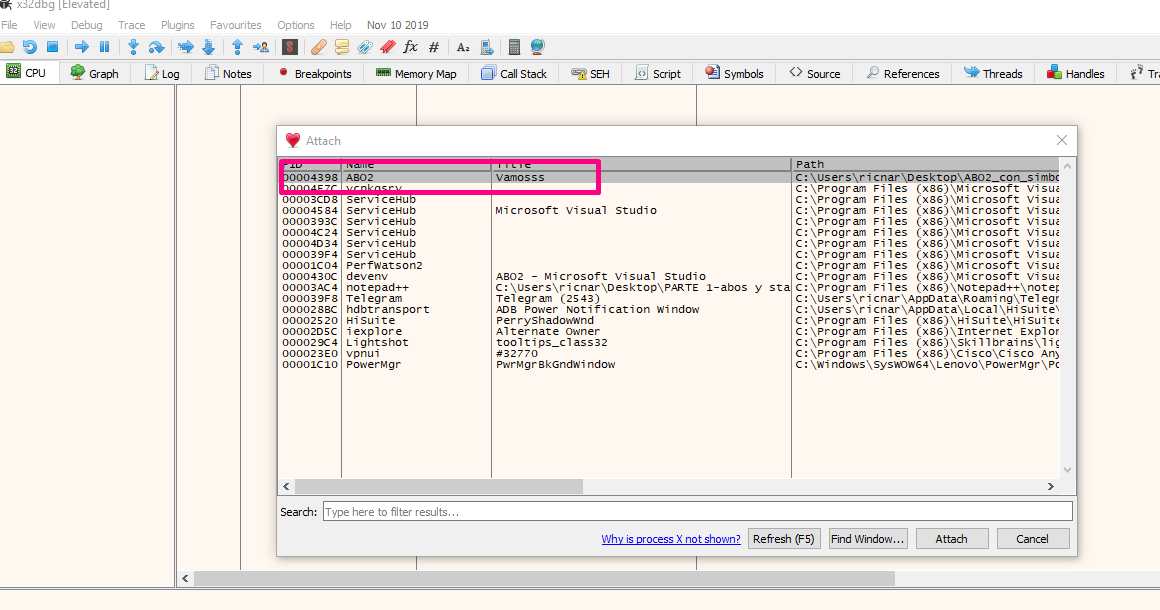
Si lo ejecuto desde una consola veo que no me aparece la calculadora, solo se cierra.

Vamos a debuggear con el x64dbg, ejecutemos el .py desde una consola.

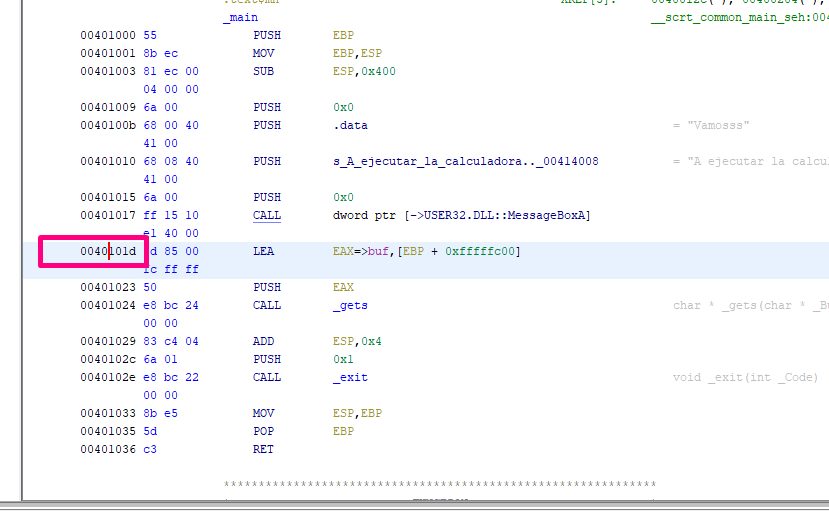
Cuando sale el MessageBoxA, atacheemos con el x64dbg para 32 bits.



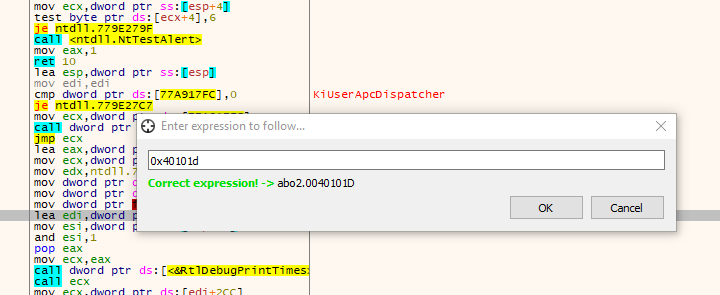




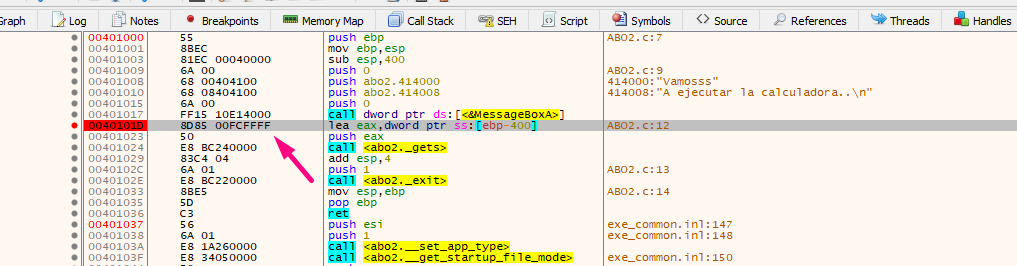
Busco en GHIDRA la dirección para poner el breakpoint al volver del MessageBoxA, es 0x40101d.

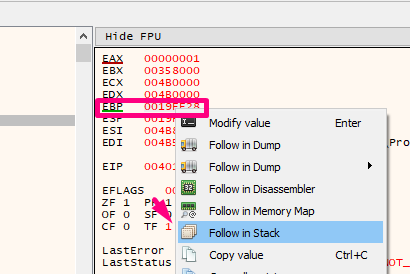


La busco en el x64dbg con click derecho - GOTO EXPRESSION

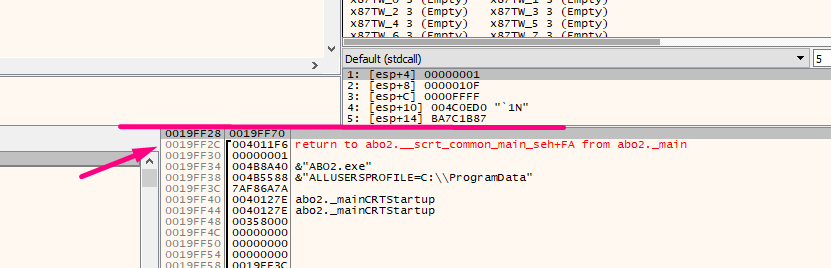


Acepto el MessageBoxA y para allí.



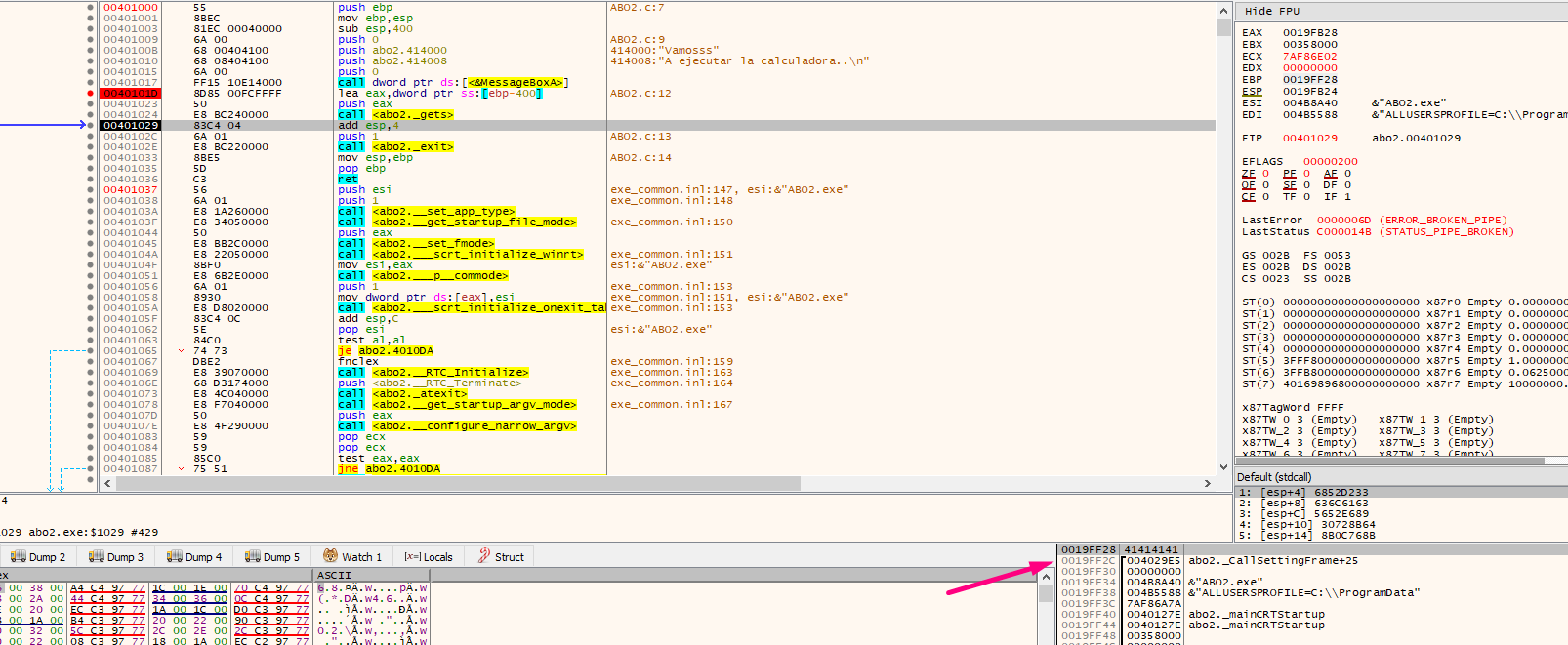


Como sabemos que EBP es el horizonte, puedo hacer click derecho en EBP-FOLLOW IN STACK.

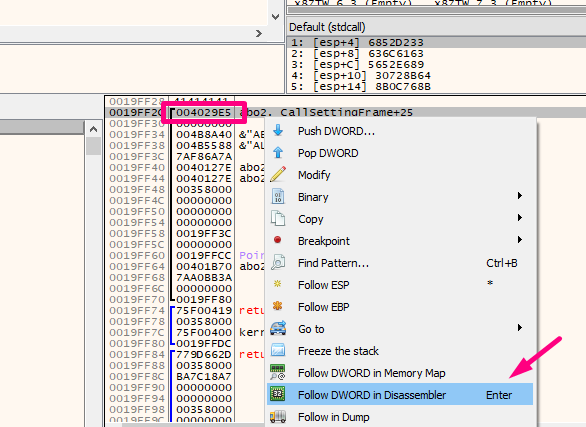


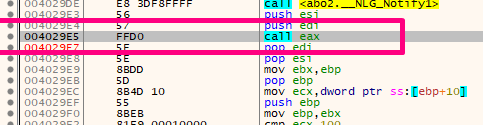
Alli estara entonces el horizonte, en la linea, abajo estará el STORED EBP, y abajo el RETURN ADDRESS, donde volvería luego de ejecutar el main, en este caso como todavia no pase por el gets, está allí el RETURN ADDRESS original que es 0x4011f6.

Si traceo con f8 y paso por el gets(), podré mirar qué pasa con el RETURN ADDRESS.



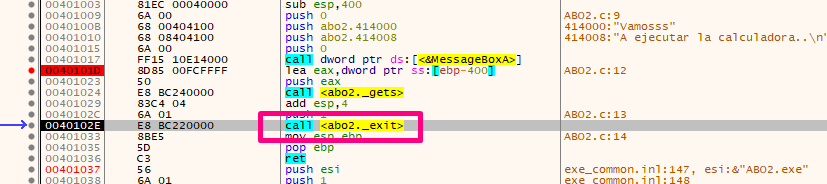
Luego de pasar con f8 por encima del gets, vuelvo a acomodar el horizonte con EBP- FOLLOW IN STACK y allí veo que el RETURN ADDRESS fue correctamente pisado por el valor 0x4029e5 que apuntaba al CALL EAX.





Bueno entonces cual es el problema, sigamos traceando hasta el RETN de la funcion.

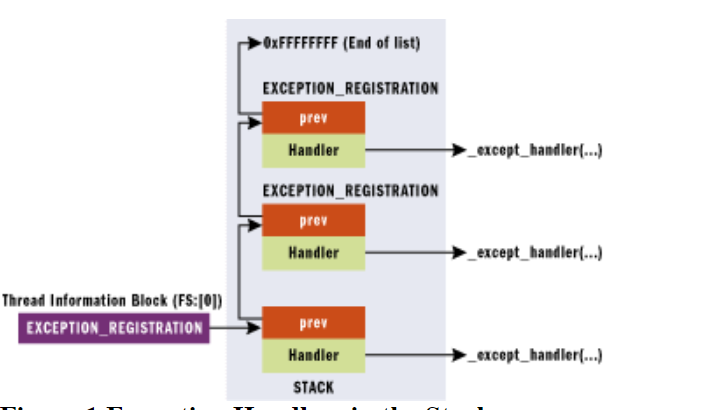
Vemos que antes de llegar al RETN hay un CALL EXIT que termina el programa.



Por lo tanto a pesar de haber pisado el RETURN ADDRESS no podemos llegar al RETN para desviar la ejecución al CALL EAX porque antes se cierra, con lo cual no podrá explotarse de esta manera.

# EXPLOTANDO SEH EN 32 BITS.

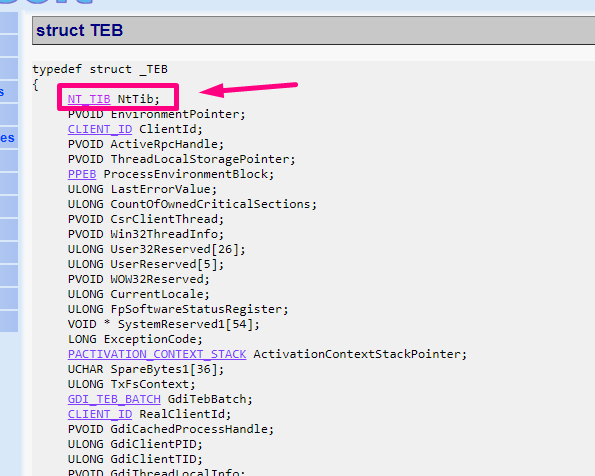
En 32 bits existe una posibilidad que es explotarlo pisando el manejador de excepciones o SEH que se encuentra en el stack.



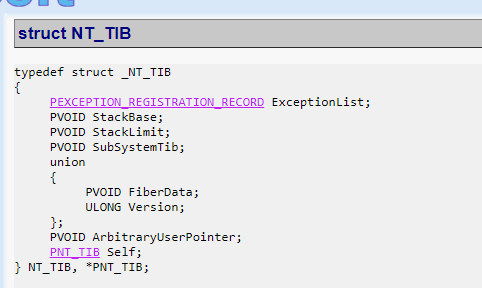
# TIB

Si recordamos la estructura TIB en 32 bits de la parte 6, vemos que en FS:[0] está la dirección en el stack donde se alojaría la primera de esas estructuras encadenadas.

Siempre digo indistintamente TIB o TEB ya que coinciden en valor ya que la TIB es el primer campo de la estructura TEB.



Allí vemos la TEB el primer campo en ella, es la estructura TIB así que su dirección de memoria coincide.

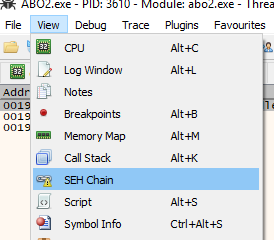


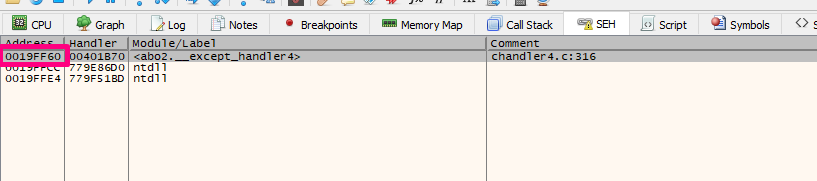
Y en el primer campo está ExceptionList que es un puntero en el stack donde comienza la cadena de estructuras EXCEPTION REGISTRATION RECORDS.

# SEH CHAIN

Si vamos a VIEW SEH CHAIN en el X64dbg

Es la cadena de estructuras EXCEPTION REGISTRATION RECORDS en el stack,

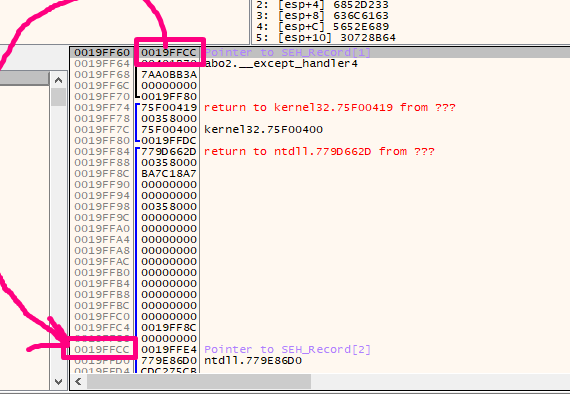




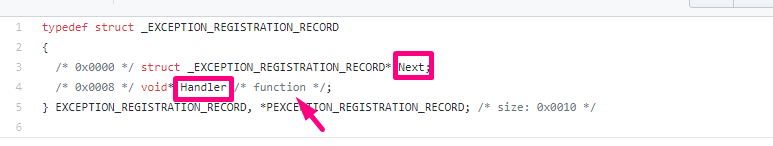
Vemos alli la dirección 0x19ff60 que es el valor que puedo ver en fs:[0] como dijimos el primer campo de la TIB.



Así que en la TIB el primer campo es 0x19ff60 que es un puntero a la primera estructura EXCEPTION REGISTRATION RECORD en el stack.



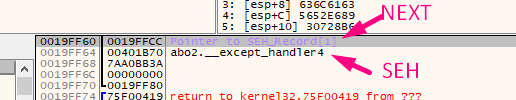
La cuestión es que estas estructuras EXCEPTION REGISTRATION RECORD son de 8 bytes, están compuestas por dos punteros el primero llamado NEXT apunta a la siguiente estructura similar, y el segundo valor es el SEH.



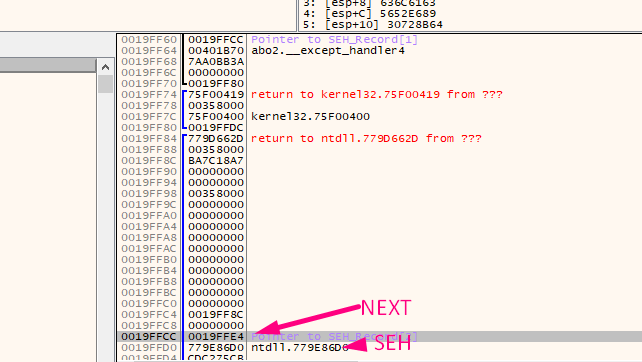
# NEXT Y SEH

Allí lo vemos el primer DWORD es el NEXT que apunta a la siguiente estructura similar a esta, y el segundo DWORD es el SEH o HANDLER.

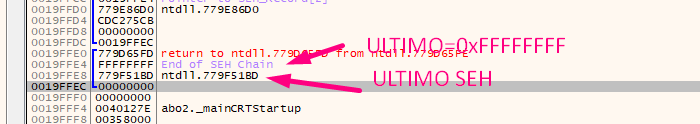
Vemos que el mismo apunta a una funcion que será llamada cuando en el programa se produzca una excepción.



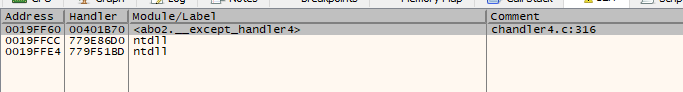
Como vimos el NEXT apunta la siguiente estructura EXCEPTION REGISTRATION RECORD similar que está en 0x19FFCC en mi maquina.



Como mostraba la imagen inicial cada una de estas estructuras EXCEPTION REGISTRATION RECORD apunta a la siguiente, la proxima estara debajo en 0x19FFe4.



El ultimo NEXT de la cadena es un 0xFFFFFFFF y debajo esta el ultimo SEH.



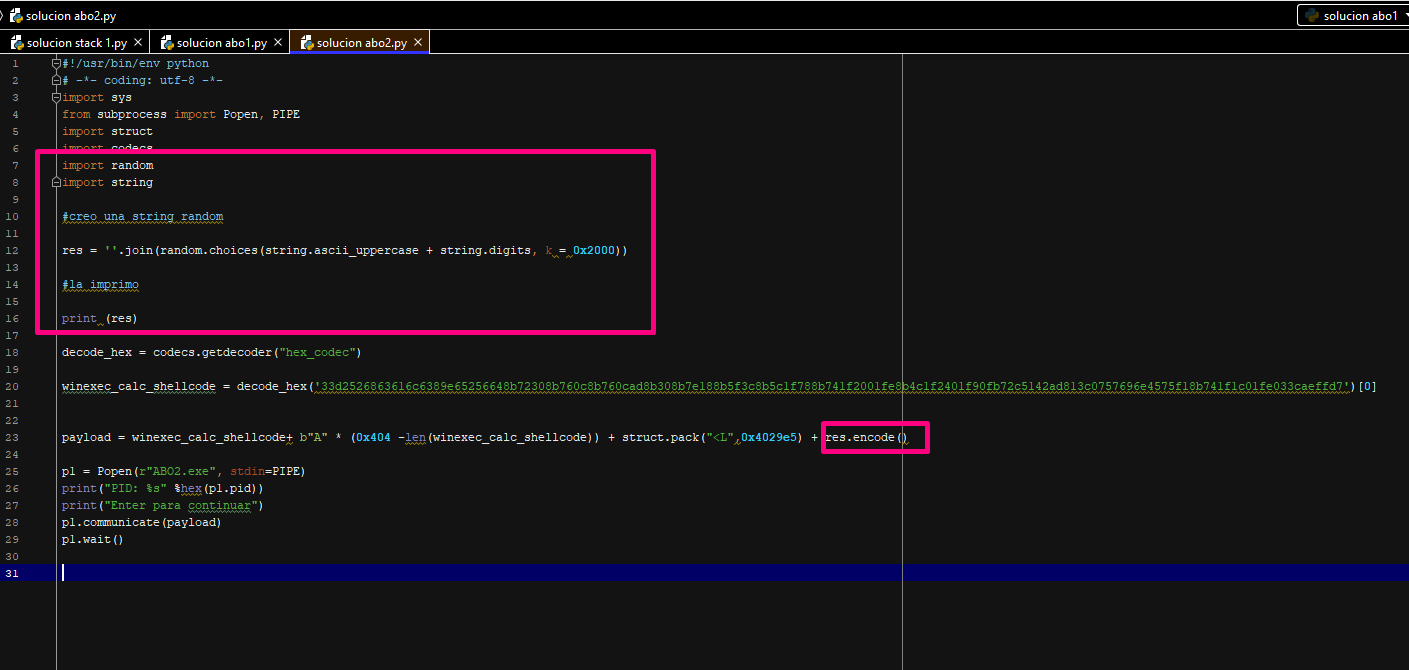
Vemos que la lista que nos muestra el x64dbg de cada estructura EXCEPTION REGISTRATION RECORD es correcta, en la primera columna están las tres direcciónes de cada estructura, desde la primera apuntada por FS:[0] , hasta la última que tiene el NEXT= 0xFFFFFFFF, en la segunda columna los handler o SEH.

Ya vimos la propiedad del NEXT de cada una de las estructuras que apunta a la siguiente estructura EXCEPTION REGISTRATION RECORD, pero para que sirve el SEH o HANDLER y como podemos utilizarlo si podemos llegar a pisarlo en el stack?

Que pasaria si por ejemplo pisamos algún SEH en el stack?

Solo con eso no pasaría nada deberíamos tratar de forzar que el programa crashee para que si tratar de manejar la excepcion salte a la dirección del SEH controlada por nosotros.

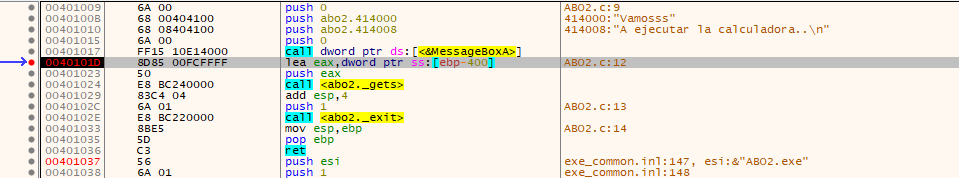
Si lleno el stack y sigo escribiendo hasta donde termina la sección del mismo, intentando desbordarlo, hasta que el programa crashee, eso producirá una excepción y si tenemos algún SEH pisado con un valor que nosotros controlemos, de esta forma al manejar la excepción y intentar continuar el programa saltará donde queremos. !!!



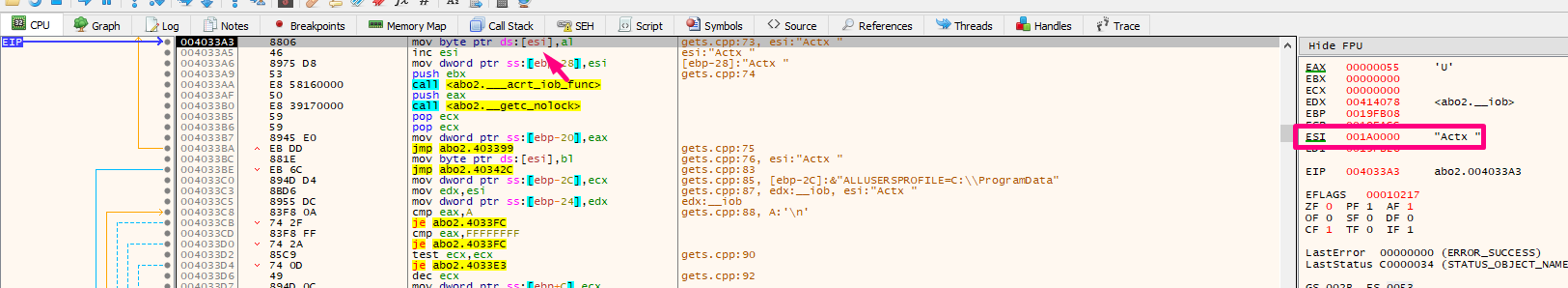
Probare con esto.

Le agrego una cadena de bytes random lo suficientemente grande (0x2000) para que desborde al tratar de copiar en el stack y crashee el programa, para sumarla la creo como string y tengo que convertirla a bytes y eso lo hago con res.encode()

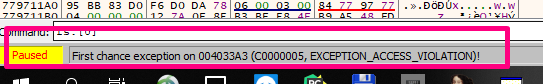
Arranco el script y atacheo el x64dbg al salir del MessageBoxA



Cuando para en el breakpoint traceo con f8 por encima del gets veremos que al tratar de pasarlo por encima el programa crashea ya que copia tanta cantidad de bytes que no alcanza el stack, como no hay secciones escribibles contiguas crasheara.

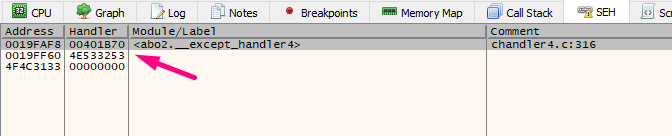


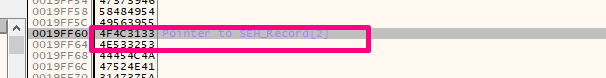
Y muestra la excepción abajo.



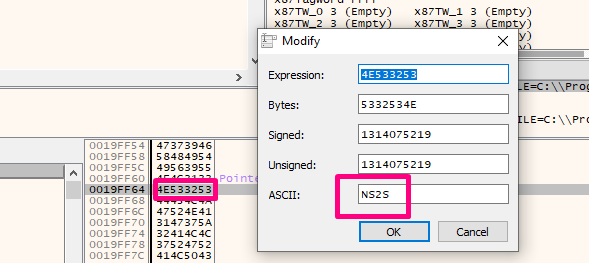
Miremos la lista de excepciones VIEW- SEH CHAIN

Vemos que la excepción del medio esta pisada, si voy al stack a ver 0x19ff60.



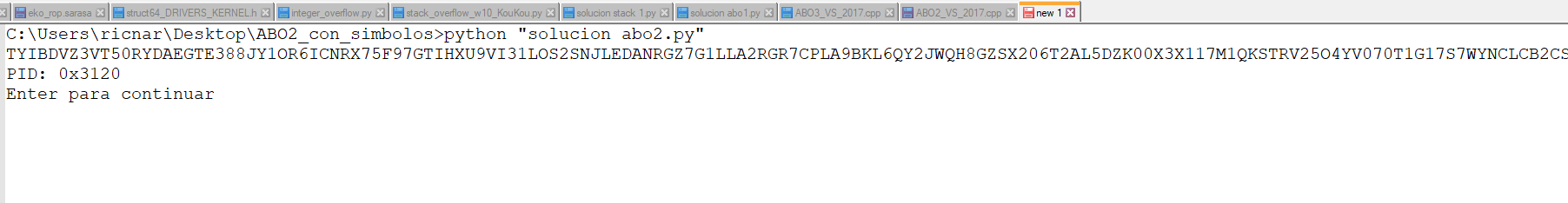


El NEXT de esta estructura está pisado en mi caso con 0x4f4c3133 y el SEH con 0x4e533253.

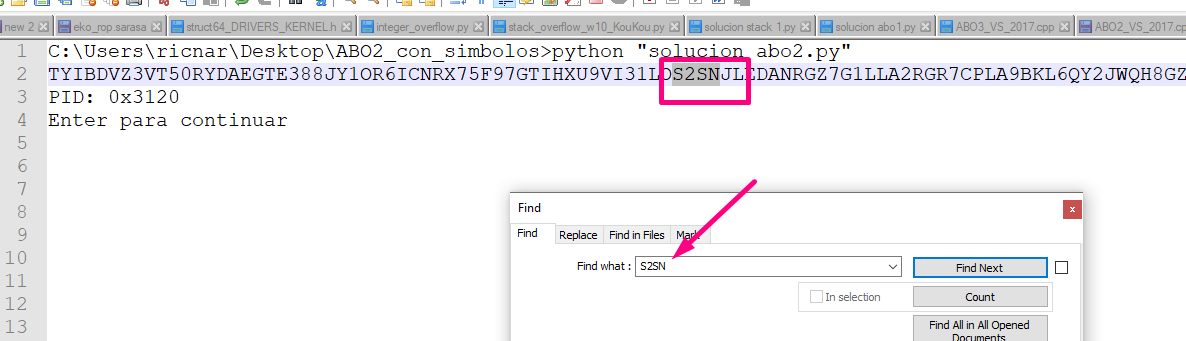


Allí veo los valores ASCII de los bytes que pisaron el SEH.

Copio y pego en un NOTEPAD el resultado de la consola

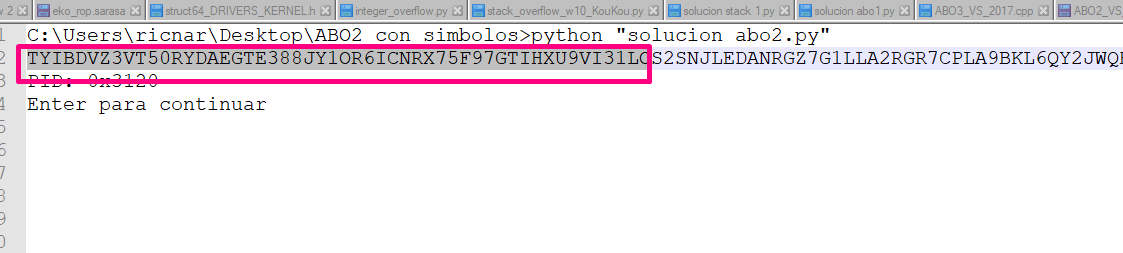


Busco los valores ASCII en la string que imprimió para ver su ubicación.

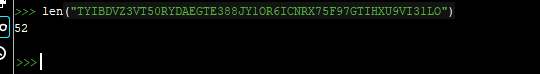


Por supuesto estara al reves, ya que el stack muestra DWORDS con lo cual ya invirtió lo que se halla en la memoria por el little endian.

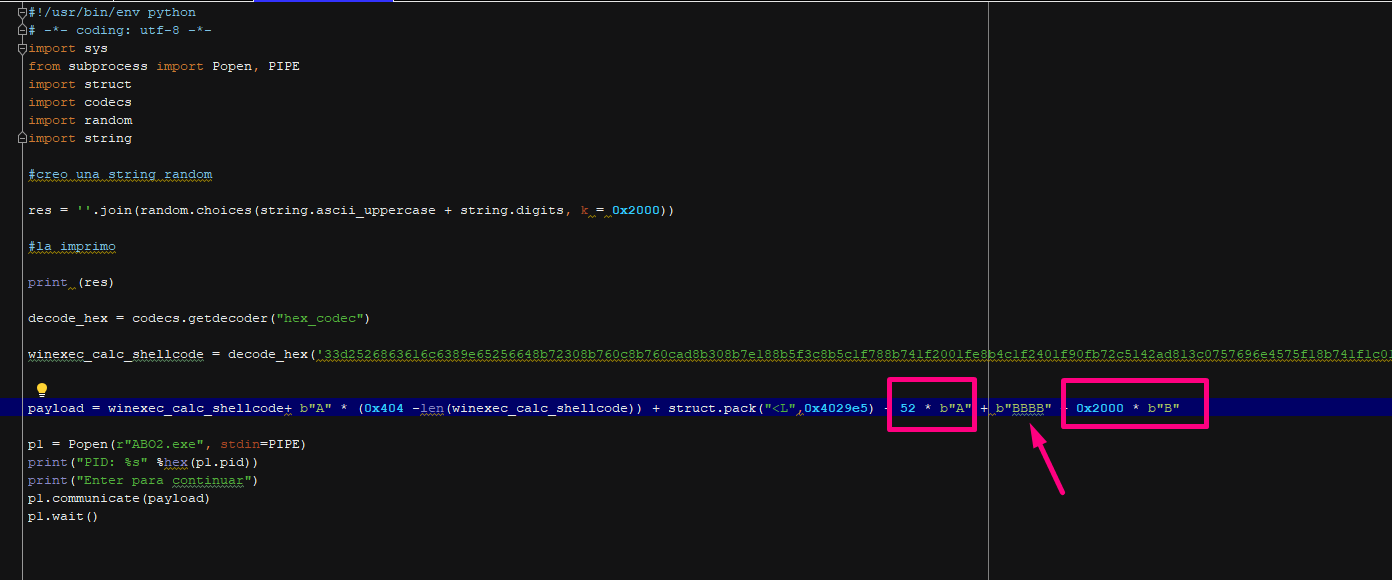
Copio la string hasta justo antes del SEH



y pego en el PYCHARM la string entre comillas y le saco el largo con len()

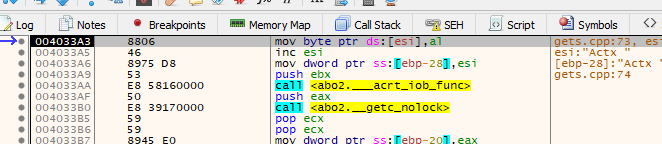


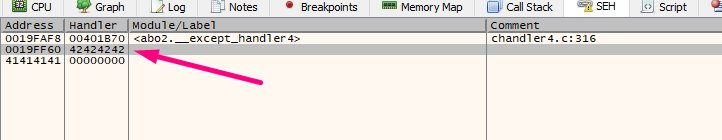
Así que a partir del 52 del inicio de los bytes random estará el SEH, probemos esto.



Allí reemplazo los bytes random por 52 \* “A¨ y luego viene el SEH que lo pisare con “BBBB” y luego relleno para que pise todo el stack hasta crashear.

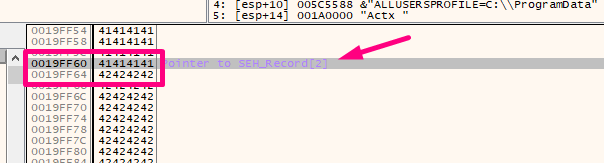
Lo vuelvo a tirar y miro ahora como queda el SEH después de crashear.



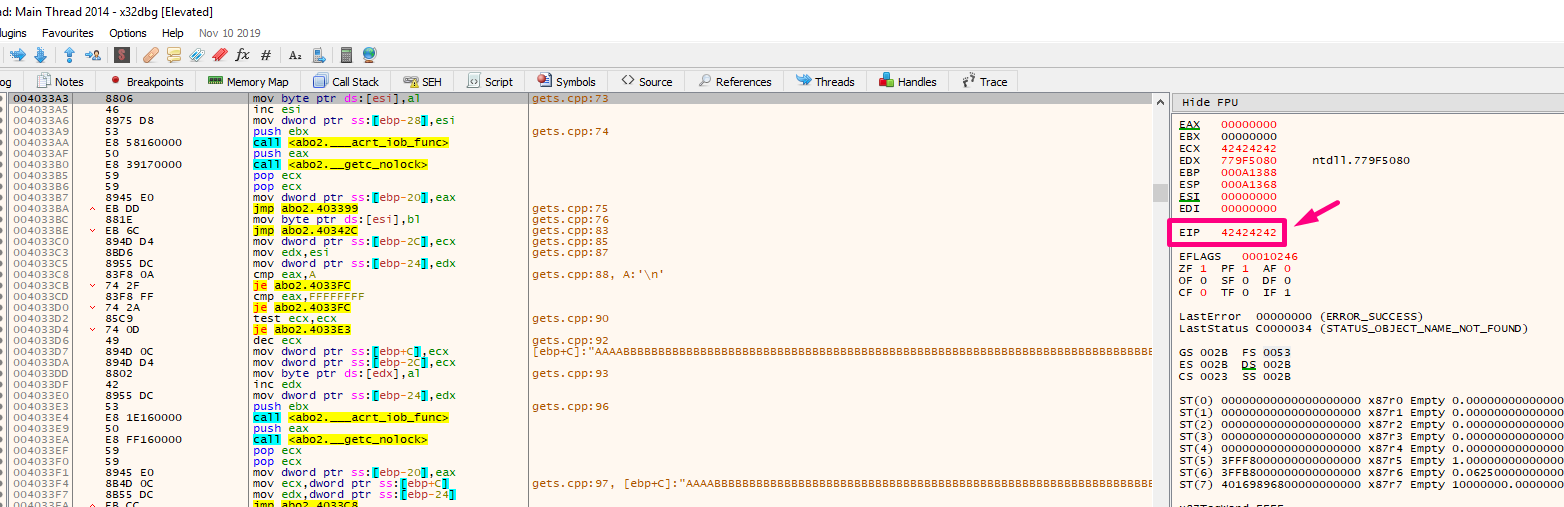


Allí está pisado con “BBBB” o sea 0x42424242.

Allí está la estructura, el NEXT pisado con 0x41414141 y el SEH con 0x42424242

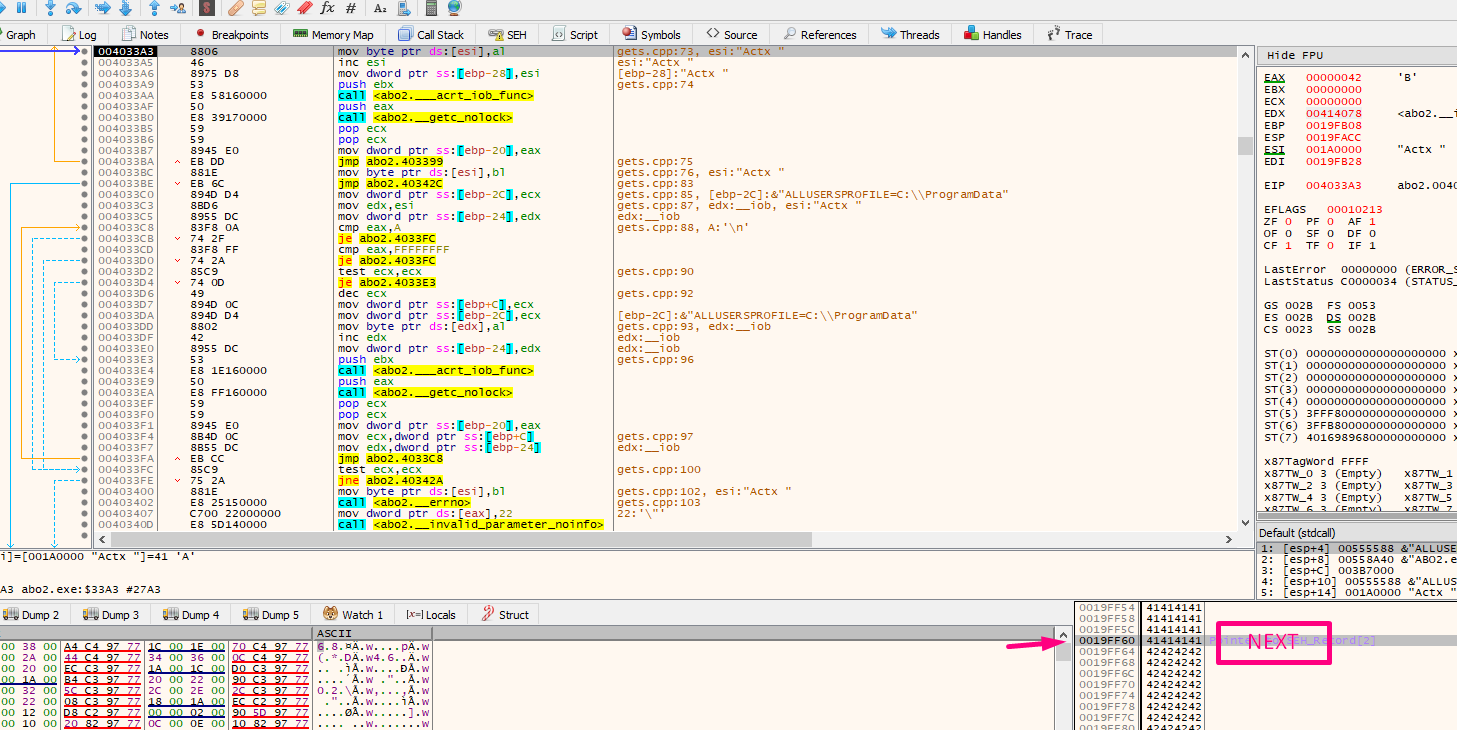


Si apreto SHIFT mas F9 para que maneje la excepción saltaría a 0x42424242, aunque allí no hay código pero probemos.



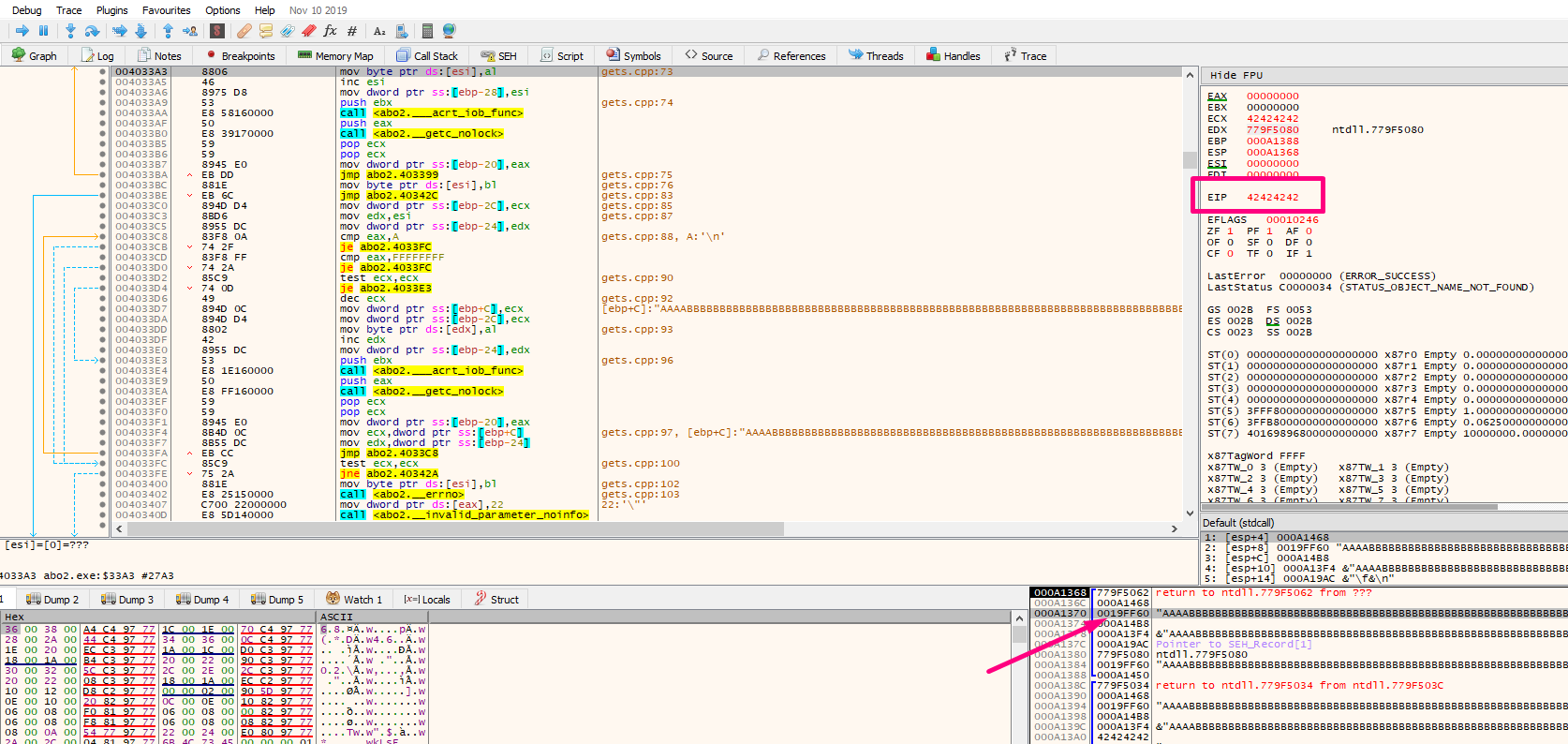
Salto a 0x42424242 aunque como no hay codigo alli no lo muestra.

Si vemos, antes de saltar a la excepción



El NEXT en mi caso es 0x19ff60 y después de manejar la excepción lo cual no es obviamente un salto directo, salta al sistema y luego de muchos chequeos y vueltas salta al SEH.

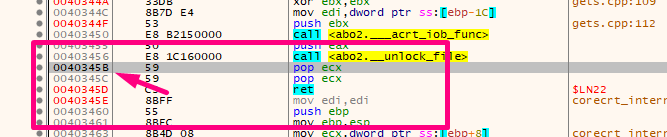
Y una vez que salto, vemos que en la tercera posición del stack deja SIEMPRE el NEXT (esto es así por diseño del manejador de excepciones de 32 bits)



La más sencilla aproximación sería colocar en el SEH, un puntero a un código con dos POPS y un RET de forma que una vez que salte allí, con los dos POPS saque del stack los dos primeros DWORDS y el RETN me permita saltar al NEXT que está en la tercera posición.

Busquemos un POP POP RET en el código.

Mirando un poco vemos un POP POP RET en 0x40345b



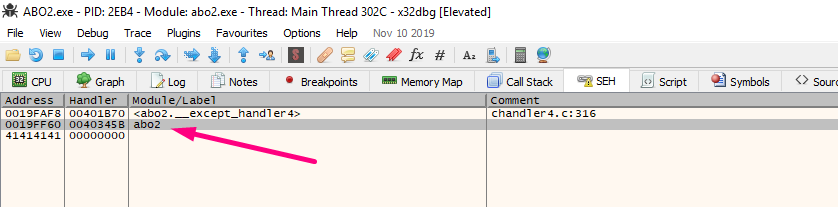
Asi que reemplazamos el “BBBB” que pisaba el SEH con 0x40345b.



Probemos a ver que pasa.

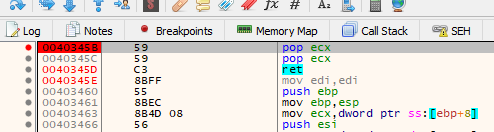
Atacheemos con el x64dbg.

Cuando crashea veamos el SEH

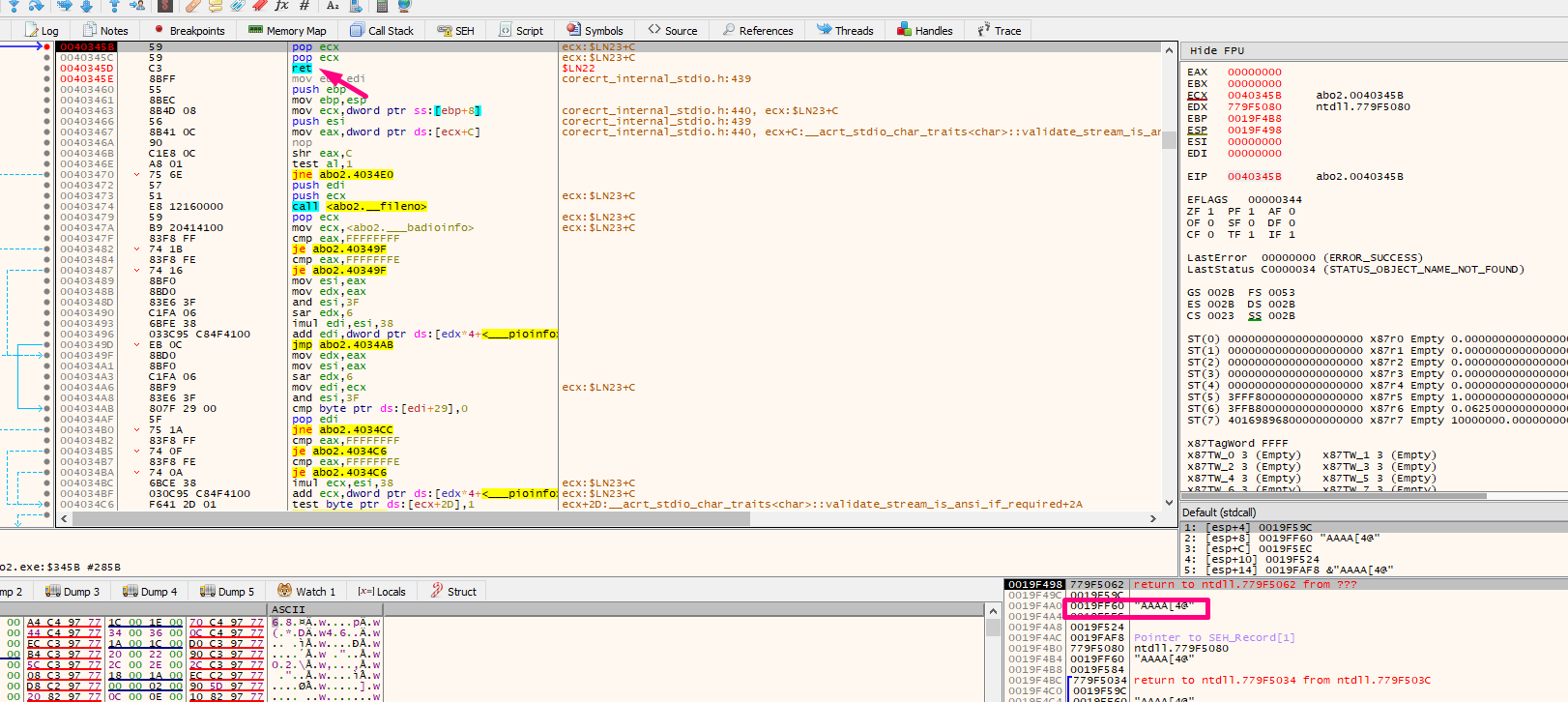


Esta pisado con 0x40345b

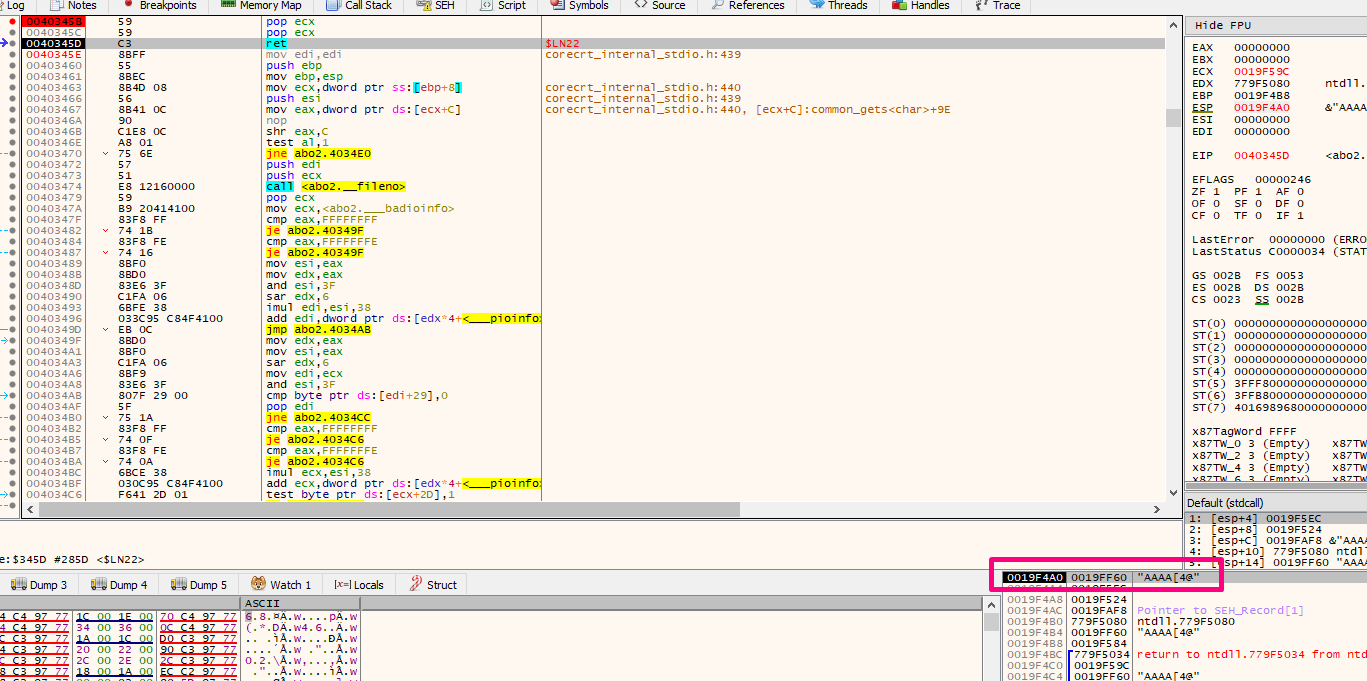
Pongamos un breakpoint allí antes de saltar.



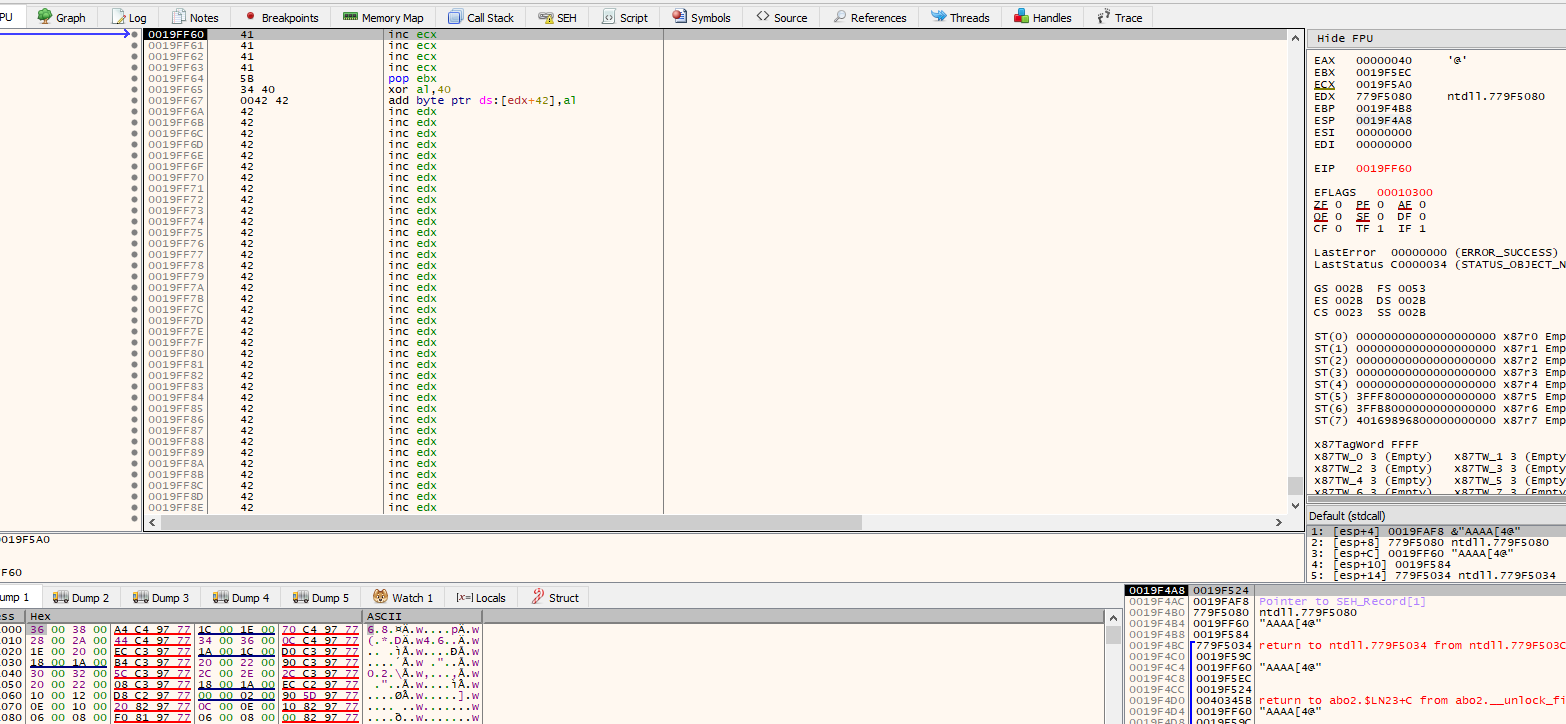
Ahora sí SHIFT mas F9 para manejar la excepción salta a 0x40345b.



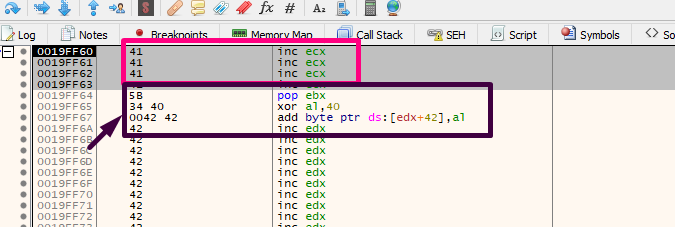
Vemos que llego al primer POP que sacara el primer valor del stack, el segundo POP sacara el segundo y llegará al RETN.



Así que saltara al NEXT apreto f7.



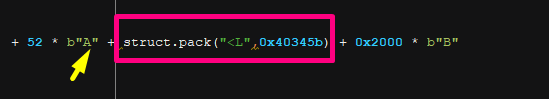
Ya estamos ejecutando código (NO HAY DEP HABILITADO) y como los 41414141 que estaban en el NEXT son valores míos, puedo controlarlos y cambiarlos, esto es necesario porque justo abajo están los bytes del SEH.



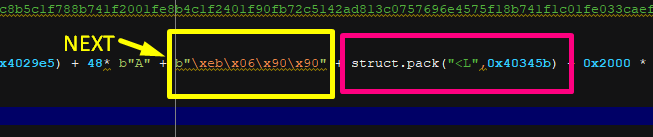
Recordemos que estamos en el stack, los **41 41 41 41** son el NEXT y a continuación estan los **5b 34 40 00** que es el SEH al revés o sea **0x0040345b** y tengo que saltar esto sino puede crashear al no ser instrucciones validas, ya debajo puedo poner el codigo que quiera.

Lo que se hace es cambiar el NEXT por **“\xeb\x06\x90\x90”** que es un salto directo veamos qué efecto tiene.

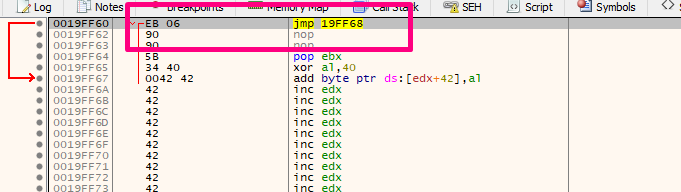
Como el NEXT estaba justo antes del SEH debo colocarlo en el script aquí.



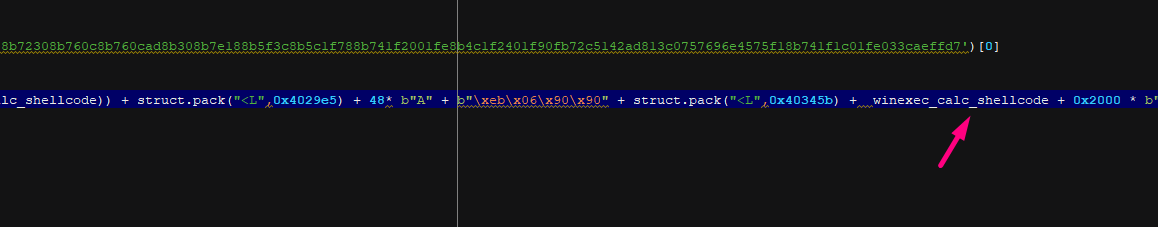
Allí está en rosado el SEH y el NEXT estaría justo antes, esta dentro de las 52 Aes, asi que debo poner 48 Aes y los 4 bytes **“\xeb\x06\x90\x90”** para mantener los 52 bytes que había antes en total.



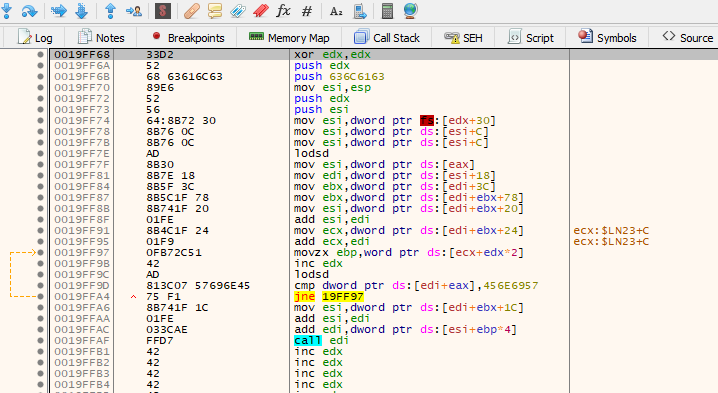
Veamos que pasa ahora



Vemos que el **EB 06 90 90** creo un salto para evitar el SEH y llegar a las B (0x42) que estan abajo donde podremos poner el shellcode.

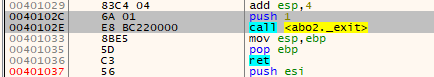


Igual esto funcionara pero le faltaba aún algo veamos.

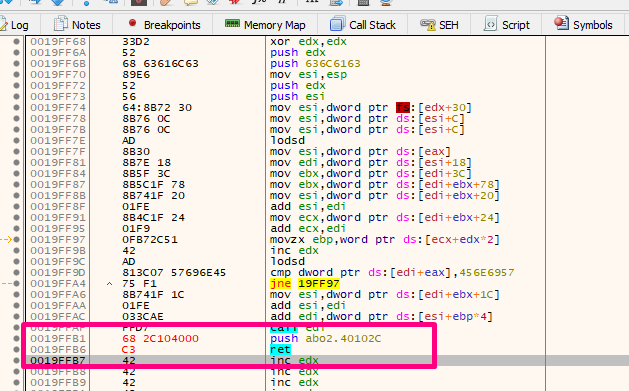


Vemos que salta al shellcode y ejecutara la calculadora, el problema es que al terminar el shellcode crasheara, saltara al SEH por el crash y volverá a ejecutar el shellcode y asi infinitamente, saldrán miles de calculadoras.

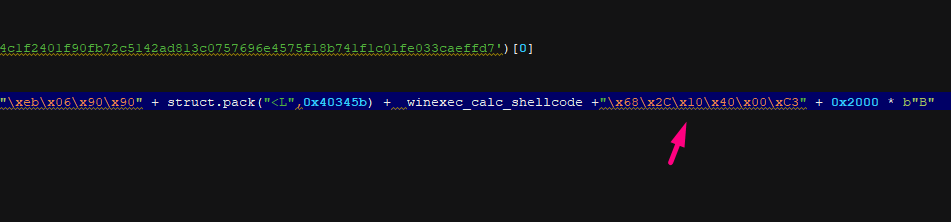
Por lo tanto debemos colocar un salto a exit luego del shellcode.



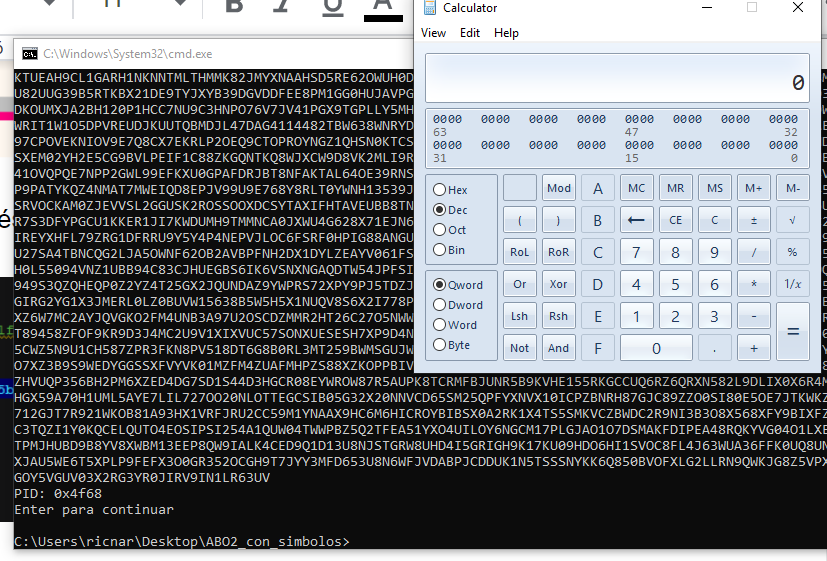
Vemos en el programa que aquí hay un call a exit.



Si le agregamos esos bytes después del shellcode saltara a exit.



Y lo ejecutamos



Igual esta técnica tiene restricciones, solo se puede saltar a módulos que están compilados con SAFE SEH OFF.

Aca hay un script de POWERSHELL que chequea el estado de SAFE SEH

<https://github.com/NetSPI/PESecurity>

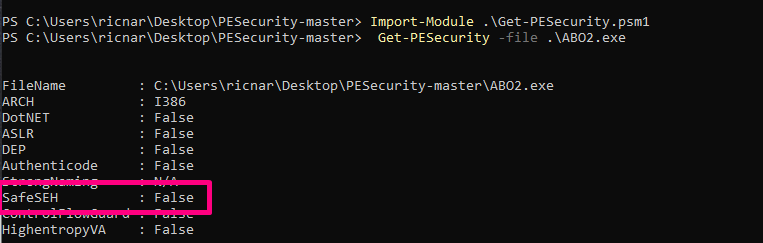
Abro una consola tipeo **powershell**

y luego yendo a esa carpeta donde esta el script

**Import-Module .\Get-PESecurity.psm1**

**Get-PESecurity -file .\ABO2.exe**

Si el ABO2.exe esta en otra carpeta poner el PATH completo.



Por lo tanto se puede saltar a este módulo, sino habrá que buscar otro módulo sin SAFE SEH para poder saltar.

Bueno con eso ya tenemos el ABO2 y vimos el SEH y como explotarlo en 32 bits, este método no funciona en 64 bits pues el manejador de excepciones no se encuentra en el stack.

Hasta la parte 8

Ricardo Narvaja

24/01/2020