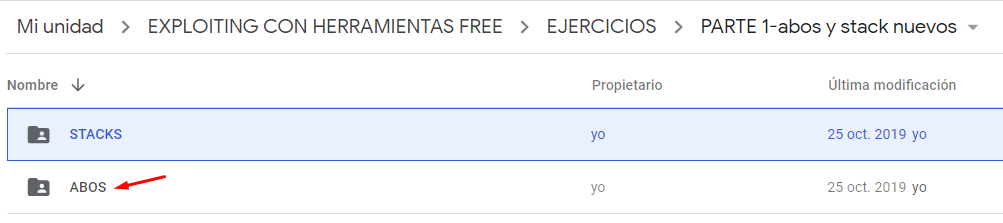
EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

GRATUITAS (PARTE 5)

Comenzaremos con el análisis del stack 4, a partir de este ejercicio haremos uno en cada tool para poder profundizar más en su uso.

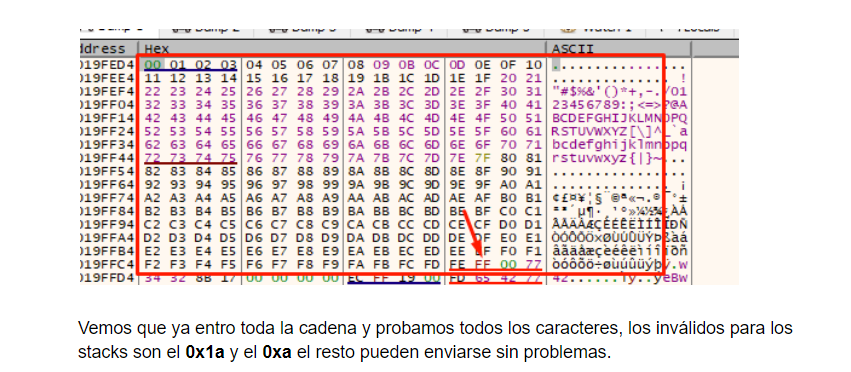
Este ejercicio lo haremos en IDA FREE, los ejercicios siguientes son los ABOS (advanced buffer overflows) están en la otra carpeta llamada ABOS y en los mismos hay que ya saltar a ejecutar una calculadora, notepad o la aplicación que queramos.



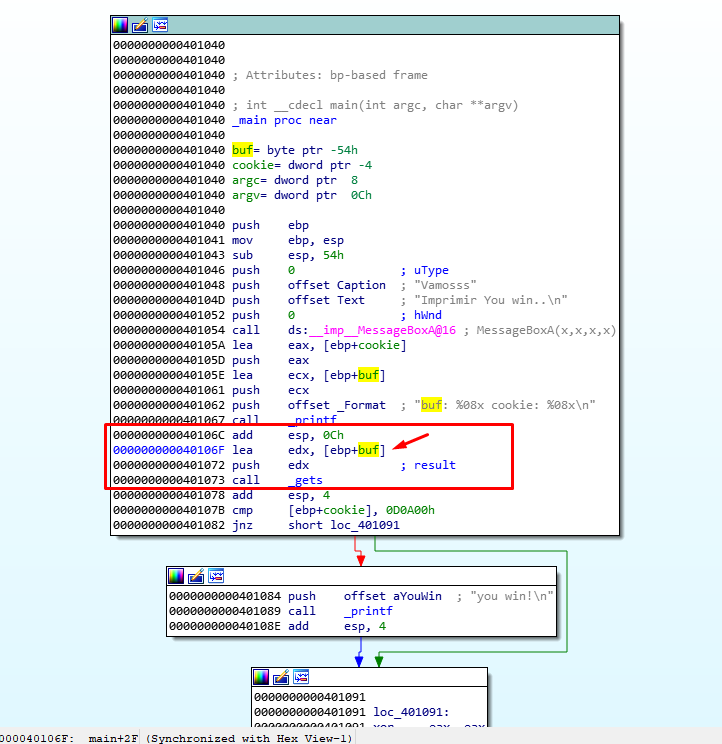
Los ABOS también son 4 asi que haremos el 1 en RADARE, el 2 en GHIDRA, el 3 en IDA FREE y el 4 de nuevo en RADARE y así seguiremos el siguiente con ghidra etc

STACK 4 en IDA FREE

En la parte 4 vimos el tema de los caracteres inválidos, hemos visto en la parte anterior que todos nuestros ejercicios llamados stacks tienen dos caracteres inválidos **0x1a** y **0xa**



Analicemos estáticamente el stack 4 en IDA FREE.

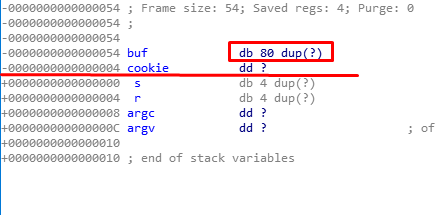


Vemos que la dirección del buffer llamado **buf**, es el argumento de la funcion gets o sea que allí guardará lo que yo tipeo hasta que aprete ENTER para terminar de tipear.

Un poco más abajo vemos la comparación de la variable cookie con la constante **0x0d0a00**, dado que cookie nunca cambia de valor en toda la funcion, es obvio que no podriamos llegar a **YOU WIN** si el programa funcionara correctamente, porque nunca podrá valer eso.

Por supuesto la única forma de lograrlo es explotando el buffer overflow en el stack, como en los ejercicios anteriores.

Podemos ver el largo del buffer **buf** en la representación estática del stack, haciendo doble click en cualquiera de la variables, de esta forma entramos en la misma.

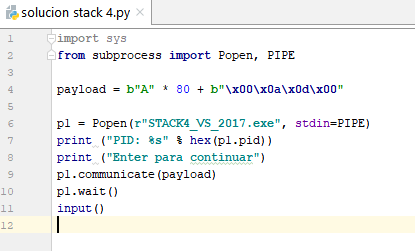


Vemos que el largo de **buf** es 80 bytes, así que si pasaramos

payload= 80 \* “A” + “BBBB”

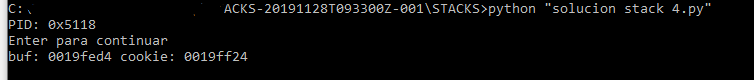
Pisaremos la cookie con la string “BBBB” que es 0x42424242 el tema es que necesitamos que la cookie para que vaya a **YOU WIN** sea igual a **0x0d0a00** por lo que le pasaremos el payload.

payload= 80 \* “A” + “\x00**\x0a**\x0d\x00”

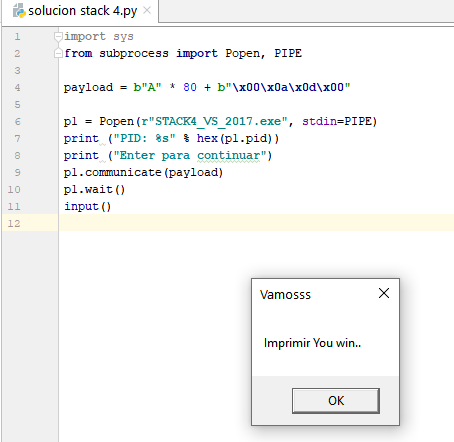


Corramoslo o bien desde una consola de windows y veamos si llega a YOU WIN.

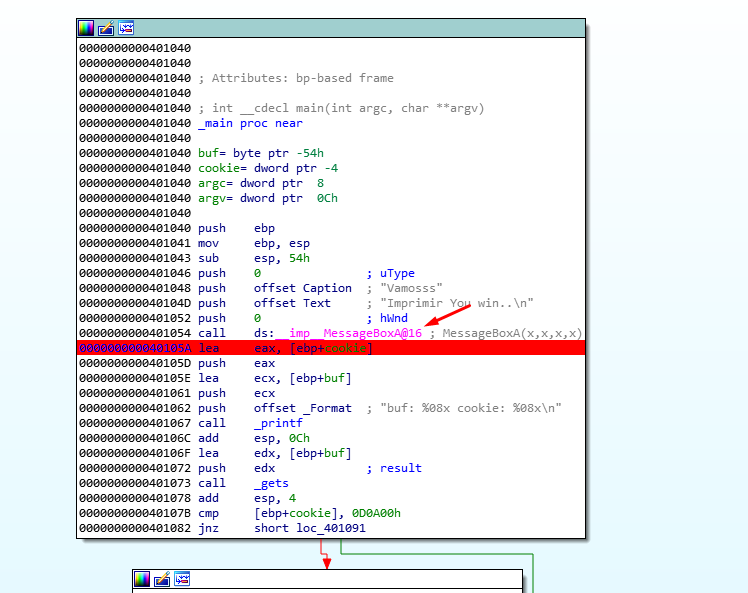
Vemos que no imprime YOU WIN y se termina.



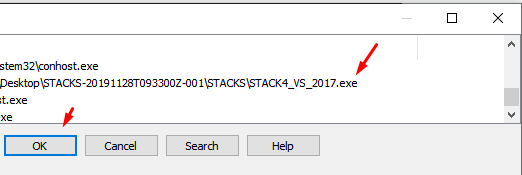
Lo que haremos ahora será debuggear, esta vez lo haremos con IDA FREE, así que volvemos a correr el script desde la consola de Windows y saldra el MessageBoxA



En el IDA FREE que teníamos abierto, primero ponemos un breakpoint al volver de la llamada MessageBoxA.

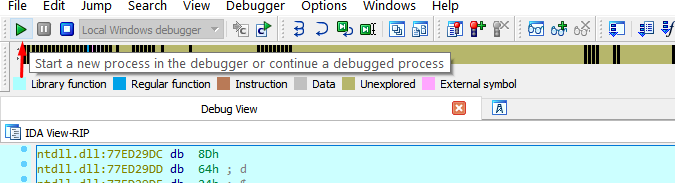


Ahora en el IDA FREE vamos a Debugger- Attach to process.

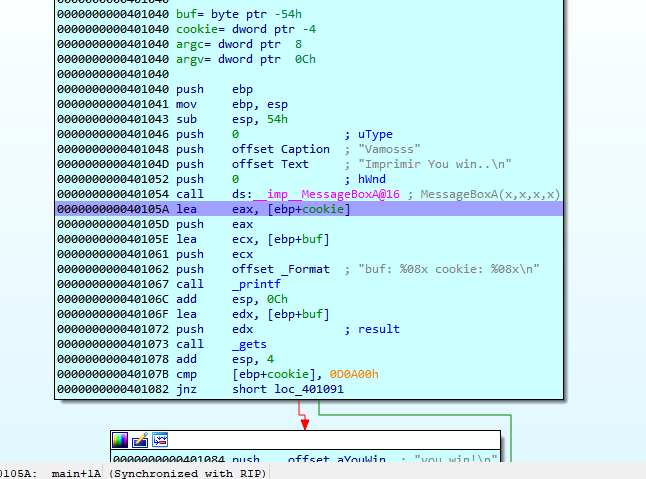


En la lista de procesos buscamos el STACK 4 y aceptamos.

Cuando se detiene apretamos play para que corra o bien f9.

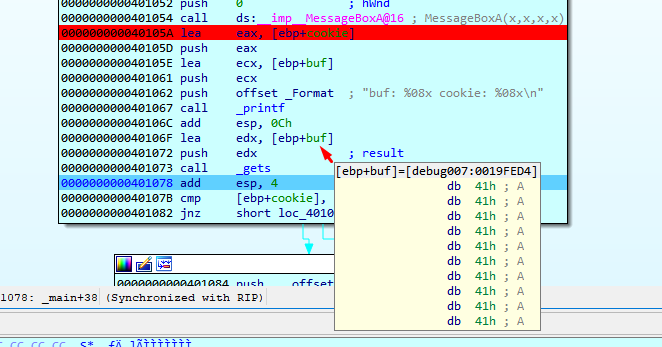


Estará corriendo, aceptamos el MessageBox.

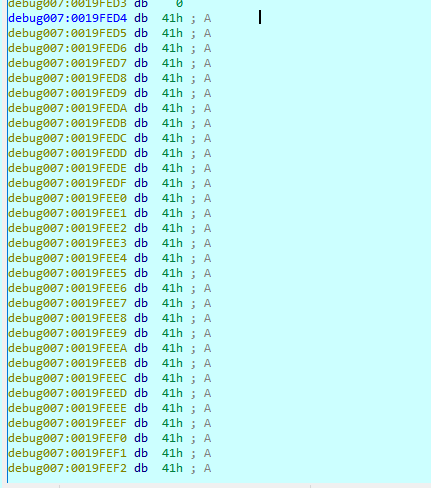


Allí para en el breakpoint, traceamos con F8 (STEP OVER) y pasemos por encima del gets a ver como maneja nuestro payload.

Poniendo el mouse encima de **buf** sin hacer click, allí vemos nuestro payload.



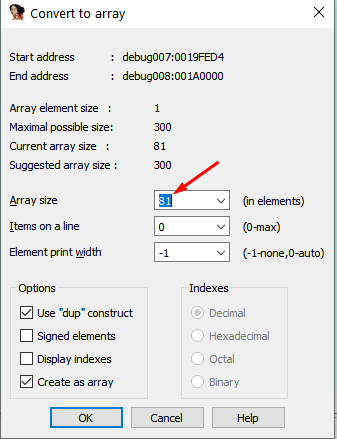
Si hago doble click en **buf**



Para medir el largo de una string sin hacer cuentas, en el primer 41 apreto la tecla A para que la transforme en una string.



Vemos que el cero final forma parte de la string, ahora si hago click derecho- ARRAY allí mismo me mostrara el largo 81 incluyendo el cero final.

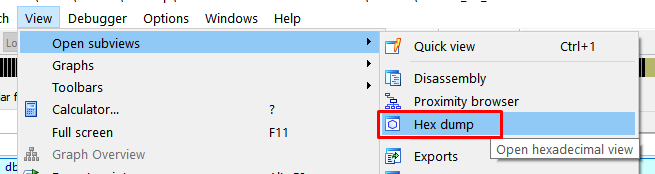


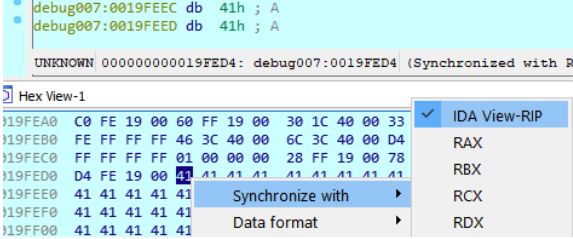
No acepto, solo hice esto para mirar el largo, veo que entraron mis 80 Aes, el cero siguiente y luego lo que continua es diferente.



Veo que se cortó justo en el 0x0a.

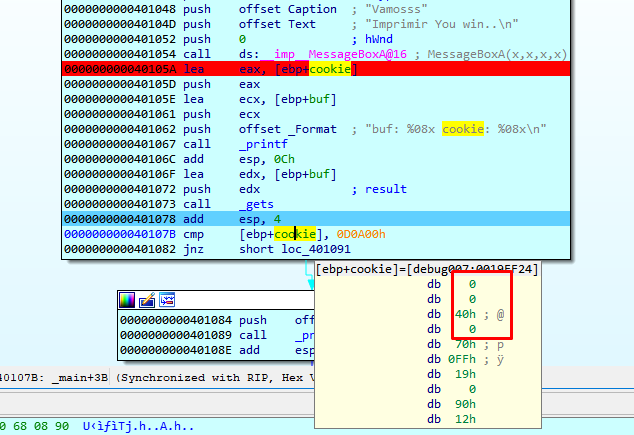
Para ver tambien la parte del payload que ingresó, tengo la ventana de HEX-VIEW si no la tienen visible



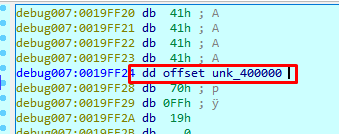


Puedo elegir que se sincronice con el contenido al que apunta algún registro, o en este caso que se sincronice con IDA VIEW-RIP que es la ventana superior, una vez elegido esto, veré abajo tambien mis datos en forma similar al DUMP del OLLYDBG como una columna con las direcciones de memoria y con los bytes del contenido a la derecha.

Así que como vemos, no entro todo el payload que enviamos, tambien si miro el valor de cookie, sabemos que son los 4 bytes a continuación de la Aes, pero si en el listado paso el mouse por la variable cookie.



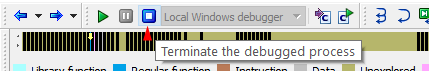
Para que cambie de la vista de bytes sueltos en la memoria sin definir como está actualmente y se muestre dicha variable como DWORD, hago doble click y apreto D, va cambiando **db** luego **dw** y luego **dd** (BYTE, WORD y DWORD)



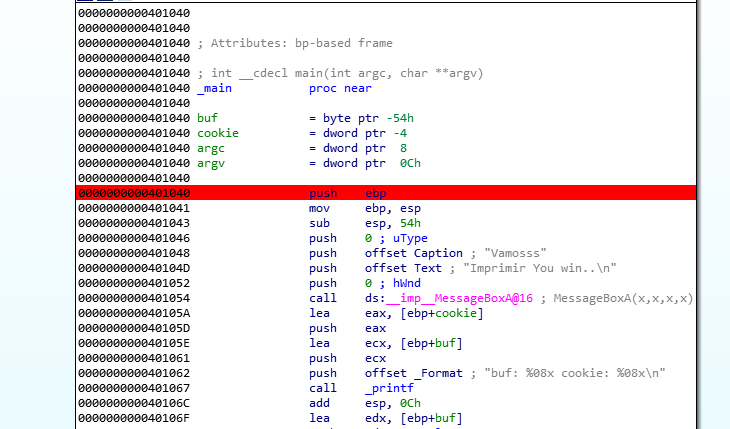
Así que el valor de la variable **cookie** es **0x400000** y por eso no es igual a **0x0d0a00** y no irá a **YOU WIN**.

Como vimos el problema es que el caracter invalido **0x0a** no se puede pasar ya que corta el ingreso de mi payload y es necesario si o si para comparar correctamente y saltar a YOU WIN, ahora que vemos que no se puede, cual es la otra forma de llegar a YOU WIN que tenemos???

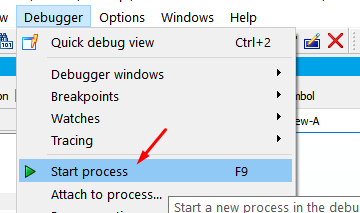
Paro el debugger.



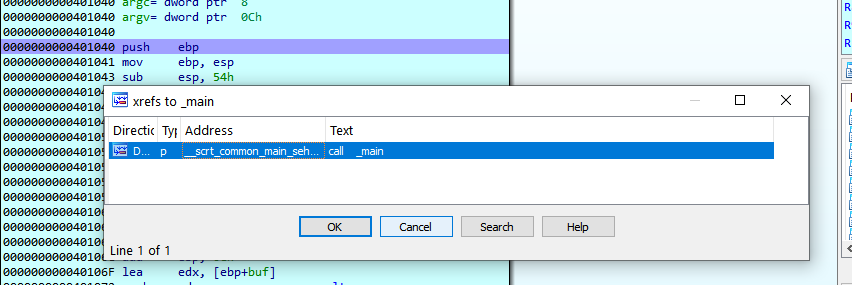
Pongo un breakpoint en la primera instrucción de la funcion main.

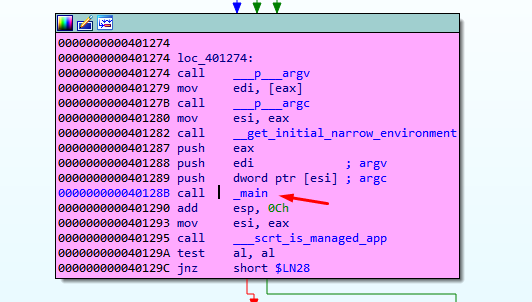


Esta vez lo voy a arrancar sin script, directamente desde el inicio y voy a ingresar los datos en la consola manualmente.



Una vez que para en el breakpoint apretamos X, para recordar que la funcion main es llamada desde aqui.

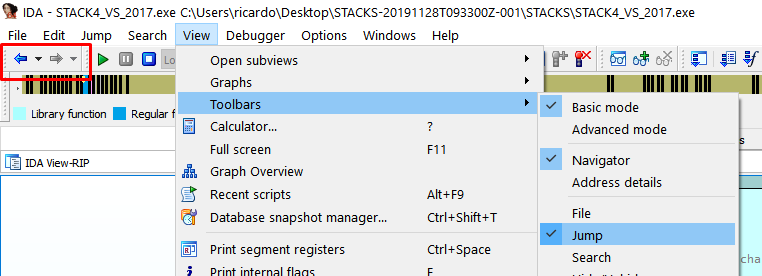


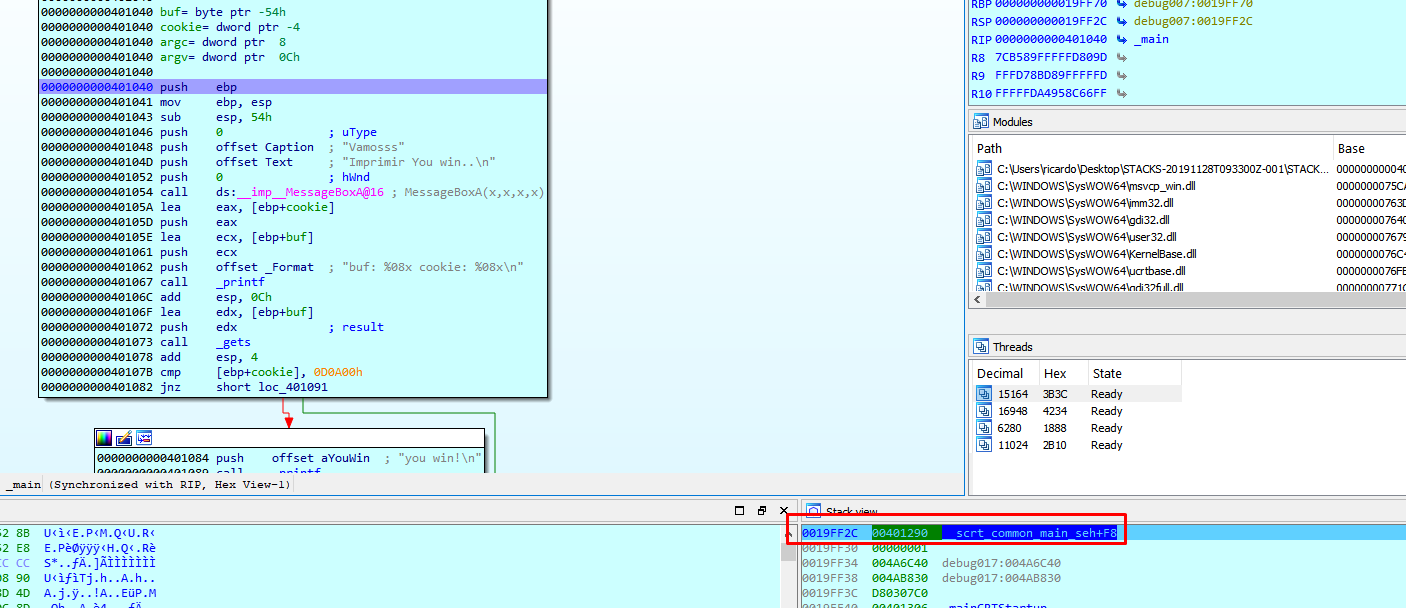


El tema es que al ejecutar ese call y entrar a main luego de haber PUSHEADO los argumentos al stack, se guarda el RETURN ADDRESS en el primer lugar del stack.

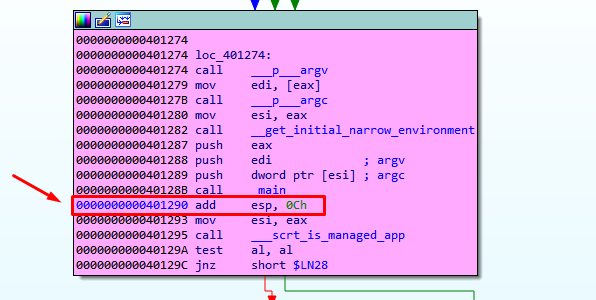
Cuando main termine y llegue al RETN de salida, la funcion obtendrá ese **RETURN ADDRESS= 0x401290** adonde debe volver, justo debajo del call que vemos en la imagen.

Una de las toolbars muy cómoda del IDA es **JUMP**, se puede agregar y con ella podemos ir hacia adelante y hacia atrás, para por ejemplo volver al main donde estábamos antes.

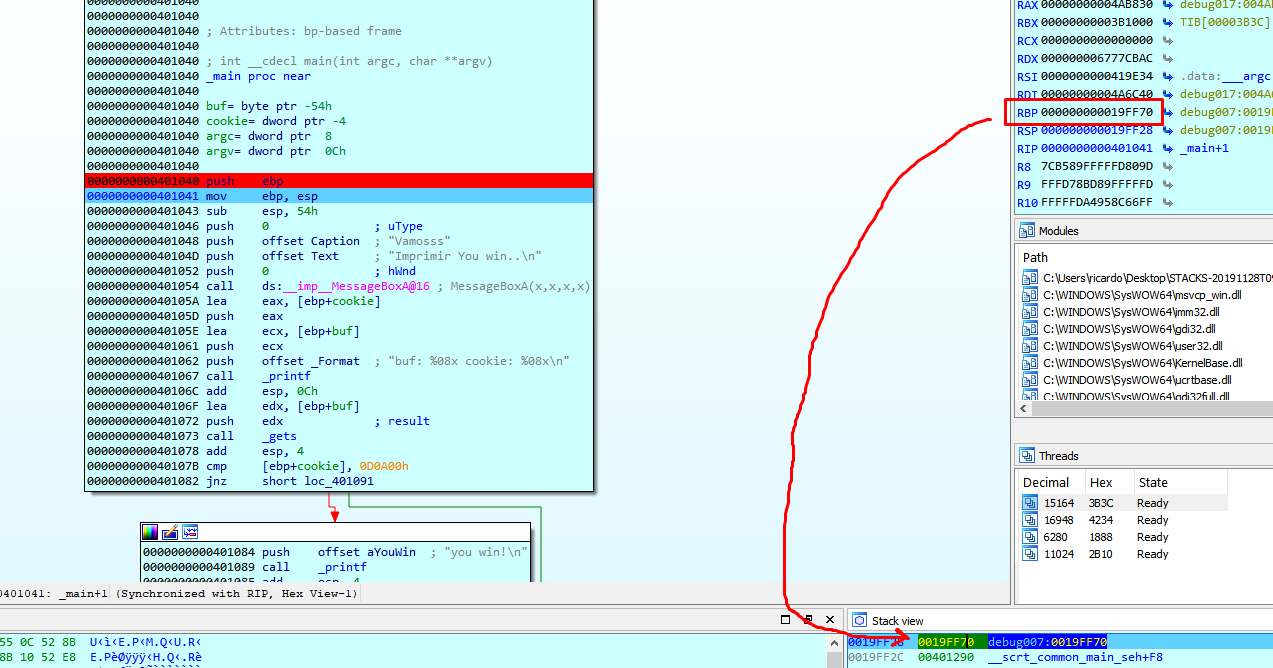




Allí vemos el RETURN ADDRESS vale **0x401290**, ya que como dijimos, al terminar de ejecutar main volveria aqui.

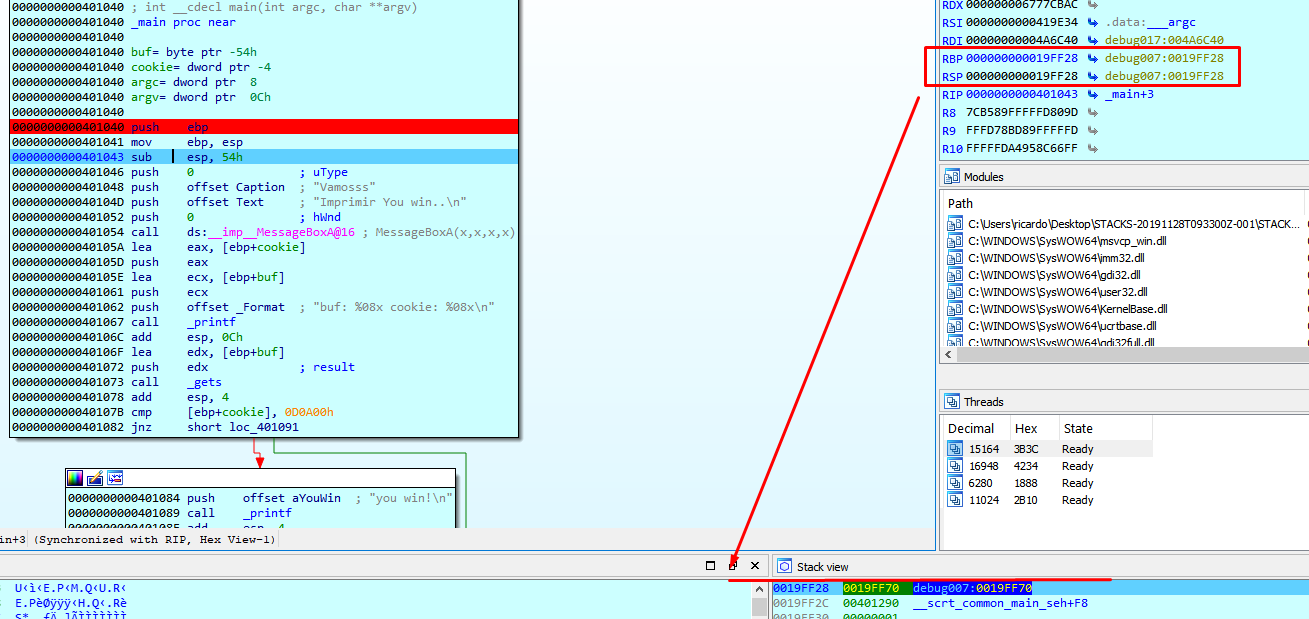


Como habíamos visto, a continuación, el PUSH EBP guarda en el stack el EBP de la funcion que llamo a main o sea el que llamamos EBP GUARDADO que es el EBP de su funcion PADRE.



Si apreto F8 veo como lo guarda en el stack, luego se igualan mueve ESP a EBP.

Se crea el llamado por nosotros HORIZONTE, apreto F8 y lo veo.



Ahora que EBP ya está seteado al valor HORIZONTE que tendra en toda la funcion, ESP debe subir para reservar el espacio para las variables locales, **cookie** y **buf**, eso se realiza con la instrucción **SUB ESP, 0x54**.

Aclaro porque muchas veces se malinterpreta, que yo hablo de SUBIR visualmente o sea en nuestro debugger o desensamblador ESP está más arriba que EBP.

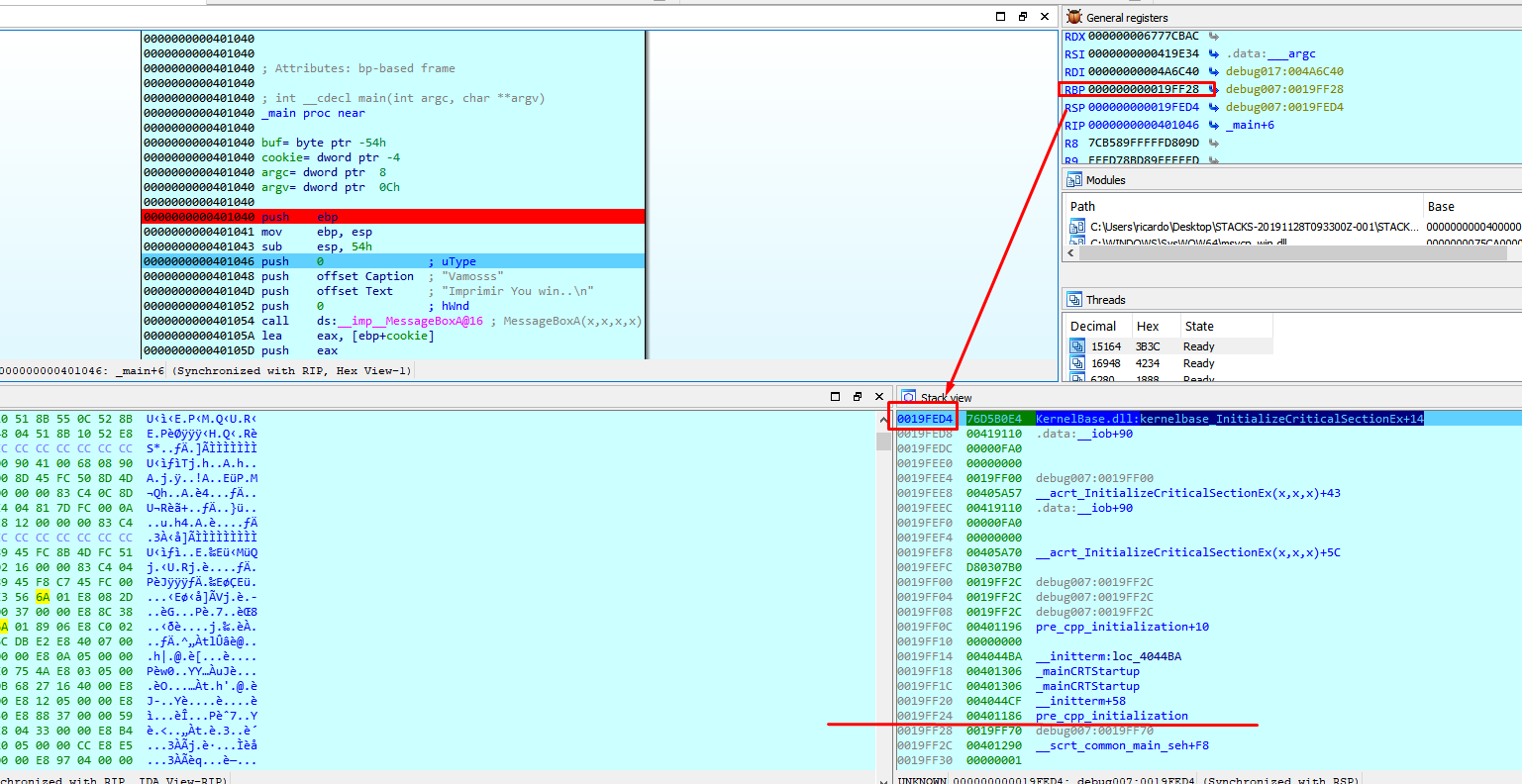
Sabemos que la memoria se muestra con las direcciones más bajas arriba y se va incrementando hacia abajo, por eso cuando vemos ESP mas arriba que EBP, realmente su valor será menor, pues al moverse hacia arriba en la imagen va hacia la zona más baja de la memoria.

Así que cuando hablo de SUBIR es un tema solo visual, de que lo vemos más arriba, cuando me refiero a aumentar su valor digo INCREMENTAR o AUMENTAR.

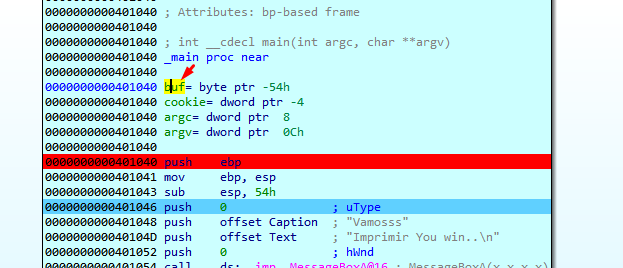
EBP queda con la dirección del HORIZONTE y ESP estará más arriba dejando 0x54 bytes de espacio para las variables locales.

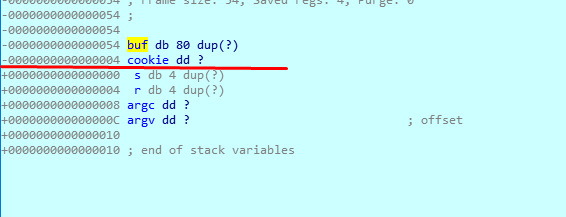


O sea que ESP quedara 84 bytes más arriba, correspondiendo 80 a **buf** y 4 a **cookie**.



En el análisis estático del stack del IDA podemos ver esto mismo, para ir en el debugger debemos hacer doble click en la definición de las variables, en alguna de ellas.





Así que IDA me muestra estáticamente esta misma representación del stack.

El HORIZONTE está en el CERO y obtengo lo mismo que acabo de ver debuggeando.

Están allí

**buf** 80 bytes reservados

**cookie** 4 bytes reservados

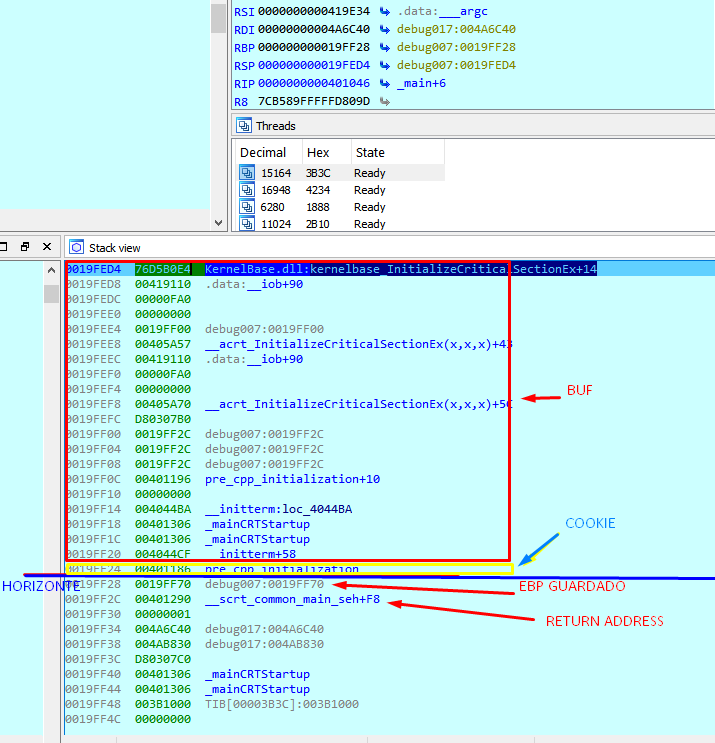
----------------------------------------------------------------HORIZONTE= 19FF28 (en mi maquina)

**s** (EBP GUARDADO) 0x19FF70 (en mi maquina)

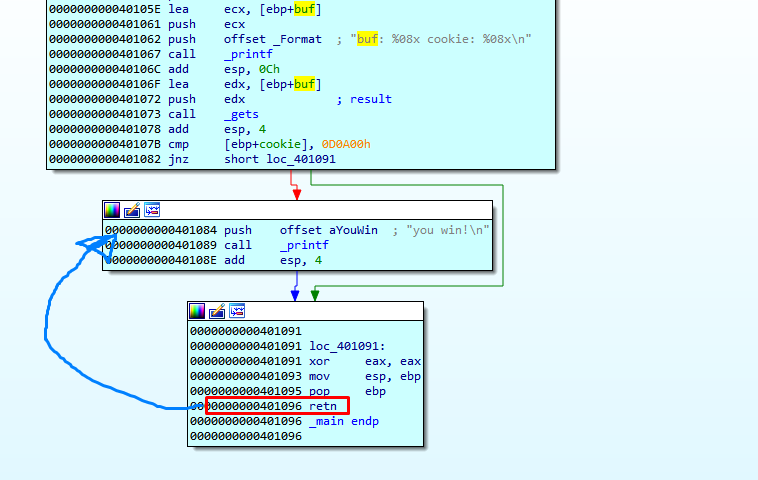
**r** (RETURN ADDRESS) 0x401290

**argc**

**argv**



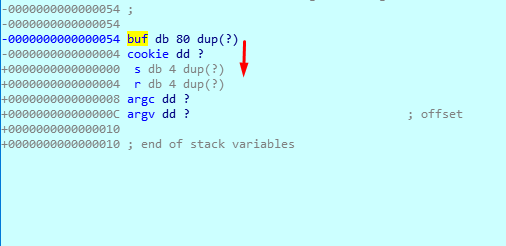
Bueno la cuestion es que el RETURN ADDRESS o sea donde volverá luego de terminar de ejecutarse la funcion, está debajo de buf, asi que calculando precisamente podemos pisarlo y elegir dónde retornar, en este caso retornaremos a la dirección donde imprimirá YOU WIN, lo forzaremos a volver a donde nosotros queremos.



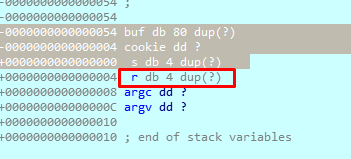
Allí en el **RETN** la funcion terminará y debería volver a **0x401290** como habíamos visto, este era el valor del RETURN ADDRESS que guardo al entrar a la funcion, pero pisaremos ese valor al desbordar buf, le colocaremos el valor **0x401084** y cuando llegue al RET volveremos al bloque que imprime YOU WIN.

Ahora calcularemos la distancia para pisar el RETURN ADDRESS, veamos nuevamente la representación estática el stack.

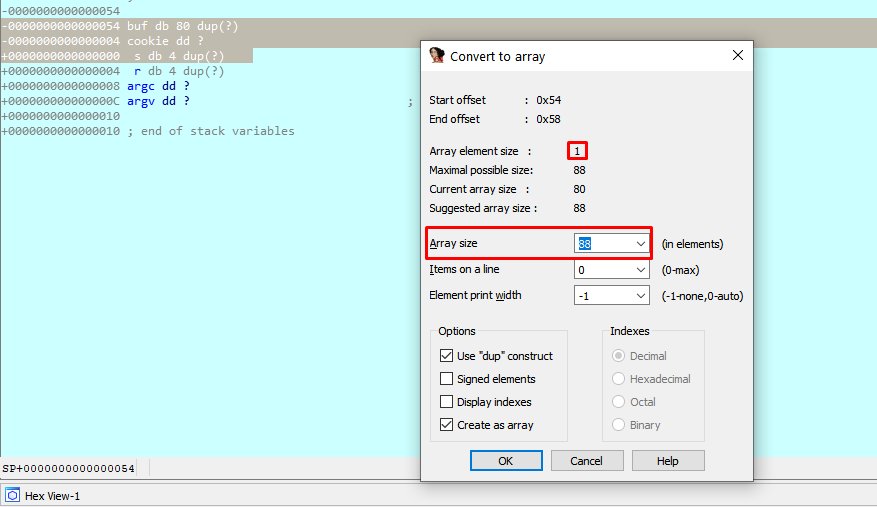
Sabemos que empieza a copiar desde buf hacia abajo.



Marquemos justo desde buf hasta antes del RETURN ADDRESS (r)



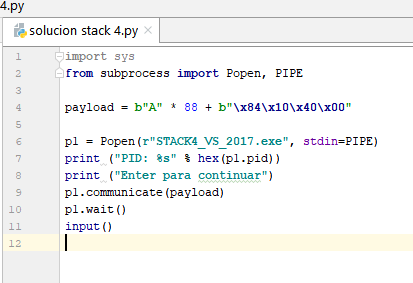
Ahora en la zona grisada hago click derecho ARRAY, para solo ver la cantidad de bytes a copiar para llenar esa zona.



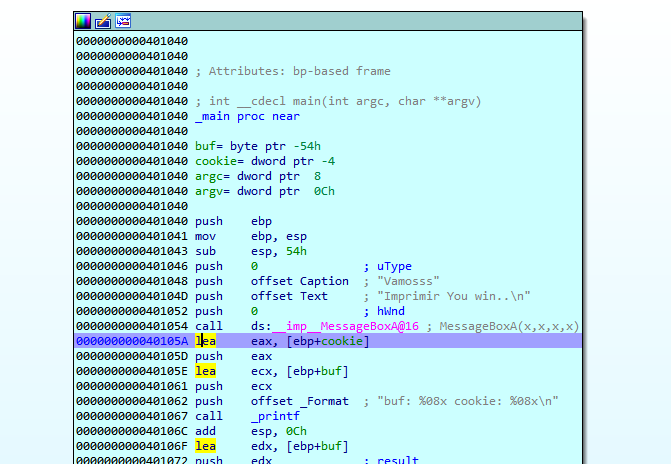
Serian 88 bytes y luego vendrían los 4 bytes para pisar el RETURN ADDRESS, o sea que nuestro payload deberia ser asi.

payload = **b"A"** \* 88 + **b"\x84\x10\x40\x00"**

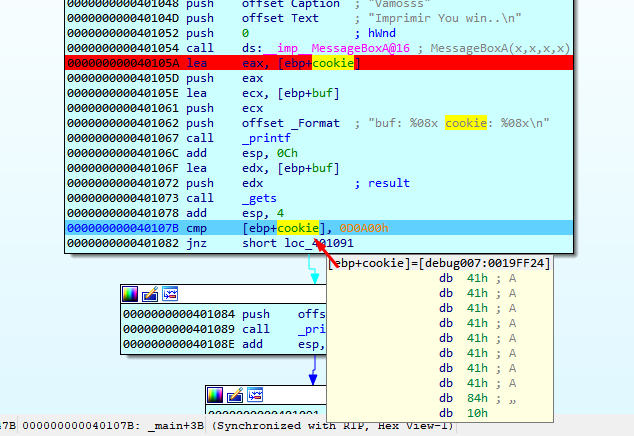
Siendo **"\x84\x10\x40\x00**" la dirección donde quiero que retorne **0x401084**



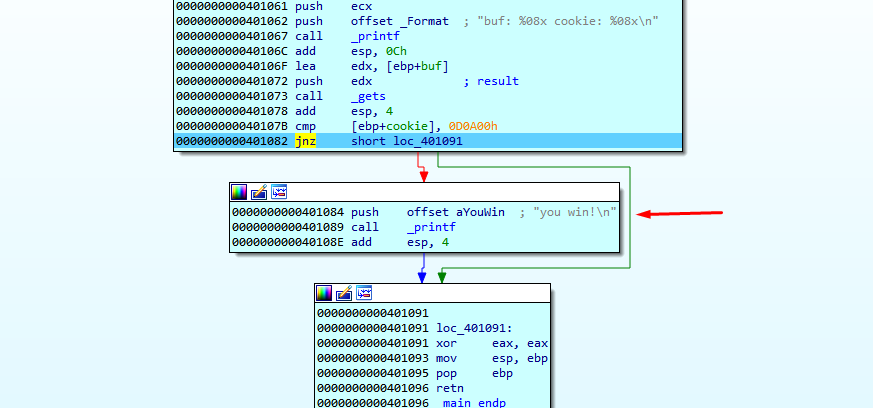
Paremos el debugger y corramos el script atacheando el IDA con un breakpoint colocado al volver de MessageBoxA, al atachear cuando pare apreto f9, acepto el MessageBoxA y parara en el breakpoint.



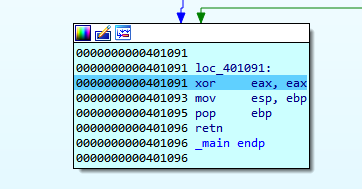
Voy traceando con f8, paso por encima del gets y llego a la comparación de la cookie.



Vemos que cookie no valdrá **0x0d0a00**, en este caso valdrá **0x41414141**(puedo ir con doble click en cookie y apretar la tecla D hasta que cambie a dd para verlo como DWORD), así que como no son iguales no irá a YOU WIN.

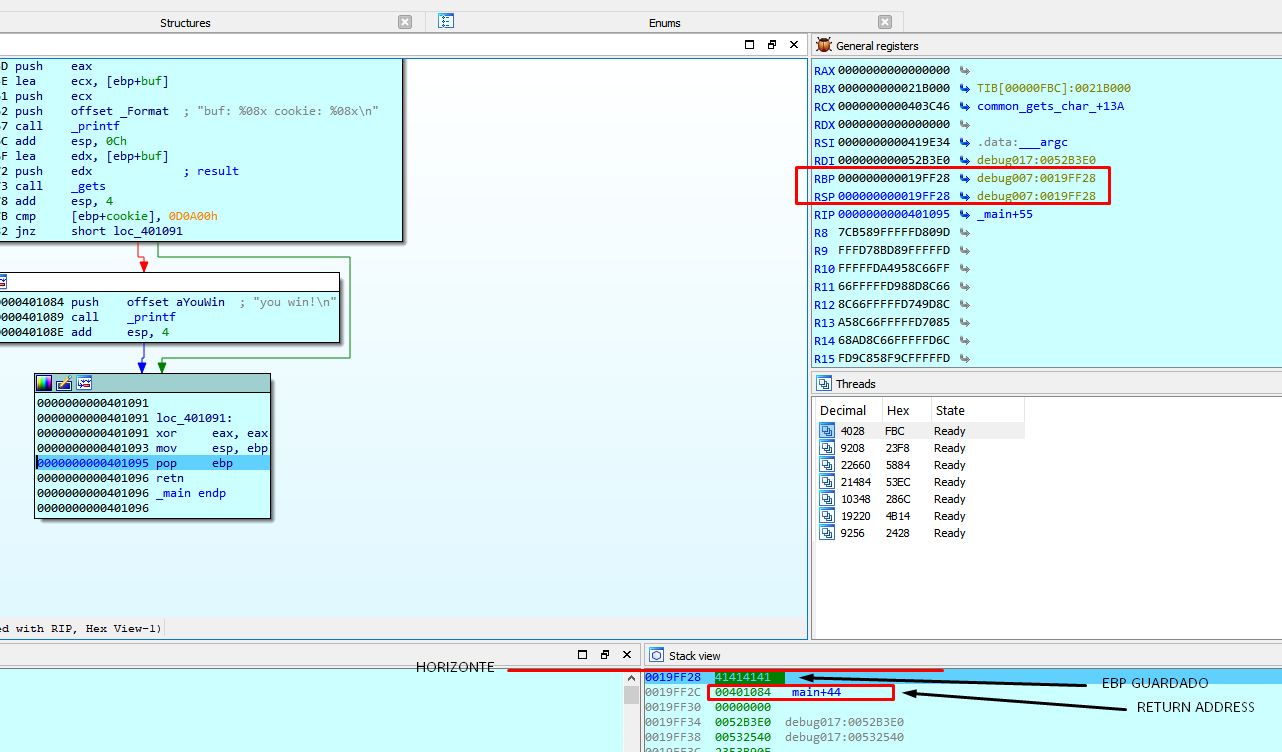


Como JNZ salta si no son iguales y esto es VERDAD ya que no son iguales, por lo tanto saltara a 0x401091 por la flecha verde de resultado TRUE o VERDADERO.



Llegamos al EPÍLOGO de la funcion, EAX saldra valiendo cero, que seria el valor de retorno de la funcion si lo tiene.

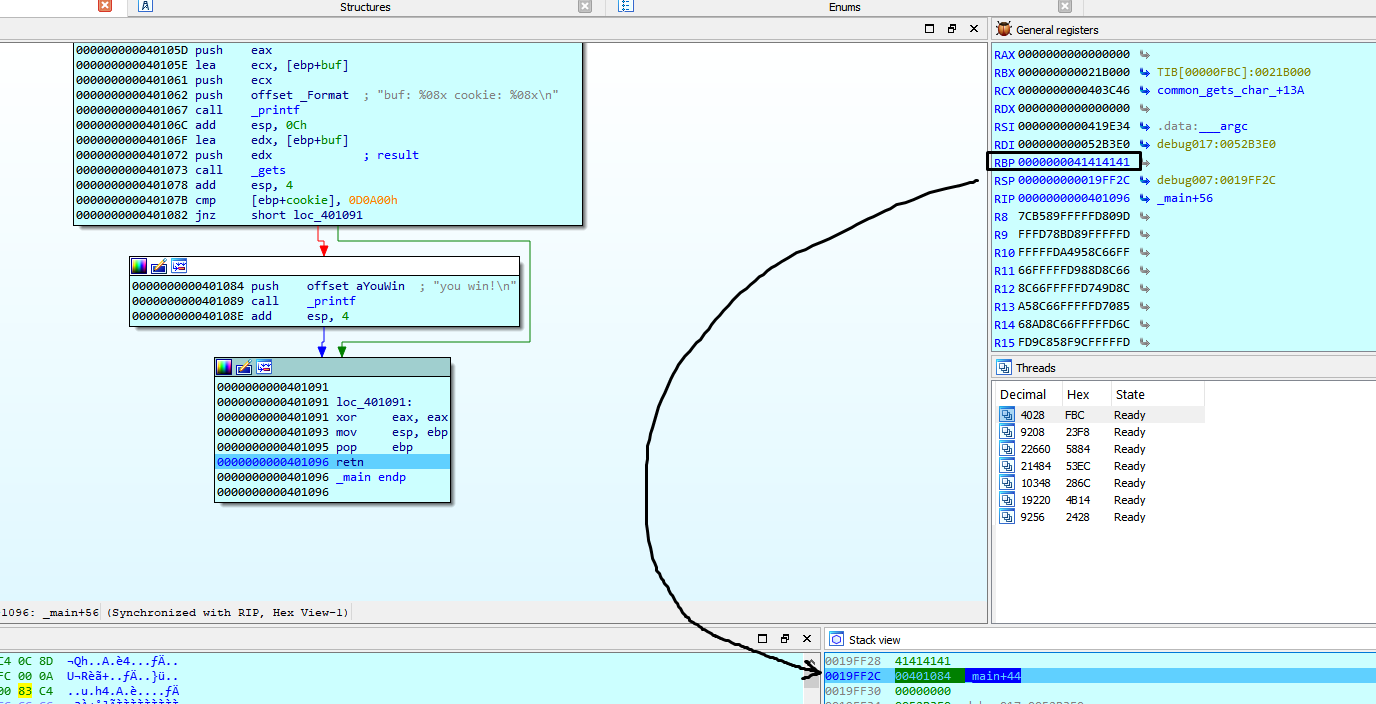
Luego ESP vuelve al valor del HORIZONTE liberando la zona de las variables **cookie** y **buf** pero sin borrar su contenido, solamente se acomoda en el valor HORIZONTE nuevamente y allí está el RETURN ADDRESS justo debajo, y volverá hacia adonde indica el mismo.



Así que estamos con ESP y EBP justo en el horizonte, debajo están **s** el **EBP GUARDADO** y el **r** el **RETURN ADDRESS.**

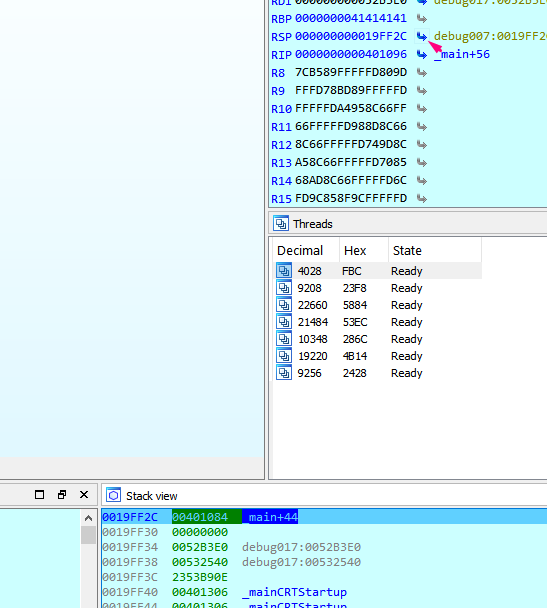
Vemos que nuestro payload al desbordar piso el **EBP GUARDADO** con 0x41414141 y el **RETURN ADDRESS** que antes valia **0x401290** ahora vale **0x401084.**

Si apreto F7(STEP INTO) el POP EBP mueve a EBP el valor del EBP GUARDADO de la funcion padre y lo restaura para volver a la misma, el problema es que este fue pisado con 0x41414141 al overflodear y no tiene el valor correcto que se había guardado en el PRÓLOGO de la funcion al hacer PUSH EBP.

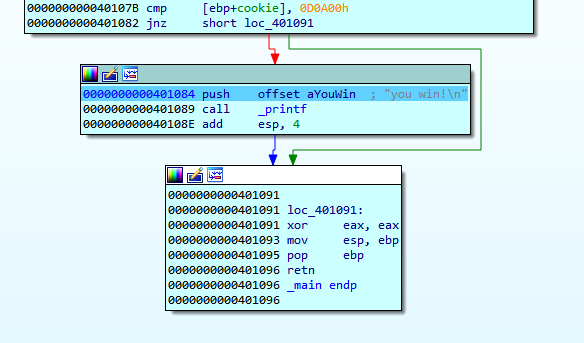


O sea que EBP valdrá 0x41414141 y el stack queda acomodado para al ejecutar el RETN y volver a **0x401084**.

Si quieren que el stack muestre el valor superior, hay que hacer click en la ventana del stack para que el cursor quede allí y luego hacer click en la flechita al lado de RSP.



Ahora vemos que el el primer valor del stack es 0x401084, apretemos f7 para que vuelva.

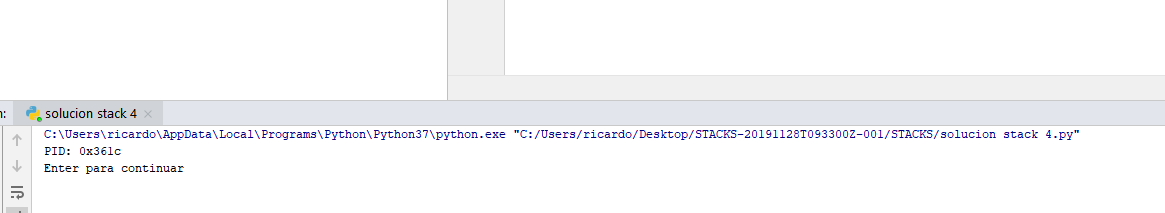


Vemos que llegamos a la parte del YOU WIN sin ser correcta la cookie, forzando el programa a que vuelva a la dirección que queremos pisando el RETURN ADDRESS.

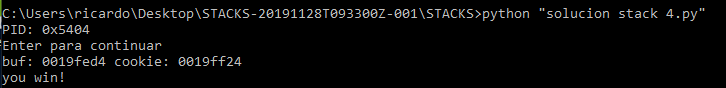
Esto es correcto y es una solución válida, el problema es que el EBP que queda en 0x41414141 al seguir traceando, romperá el programa más adelante.

Continuemos traceando con F8 para verlo.

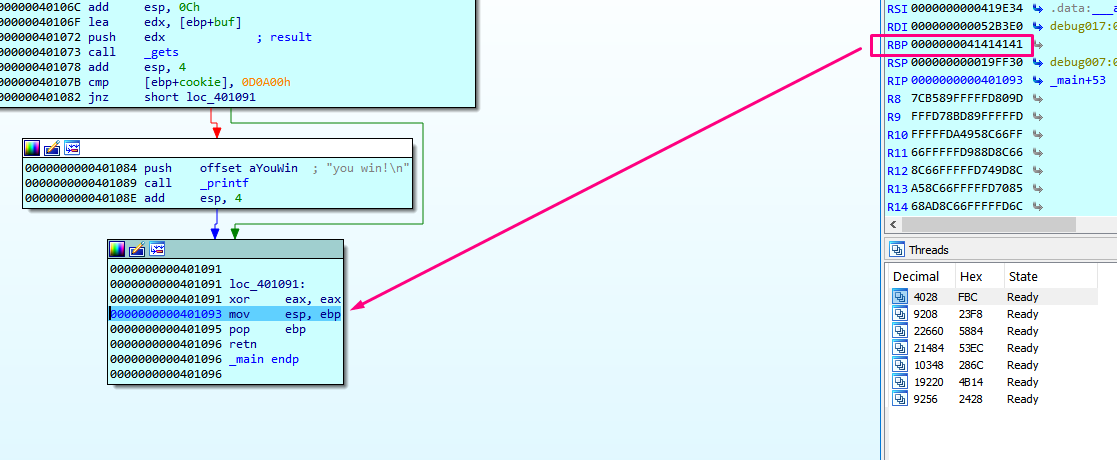
Vemos que si ejecute el programa desde PYCHARM, al pasar por el printf no aparece el **YOU WIN** en la consola de python del mismo.



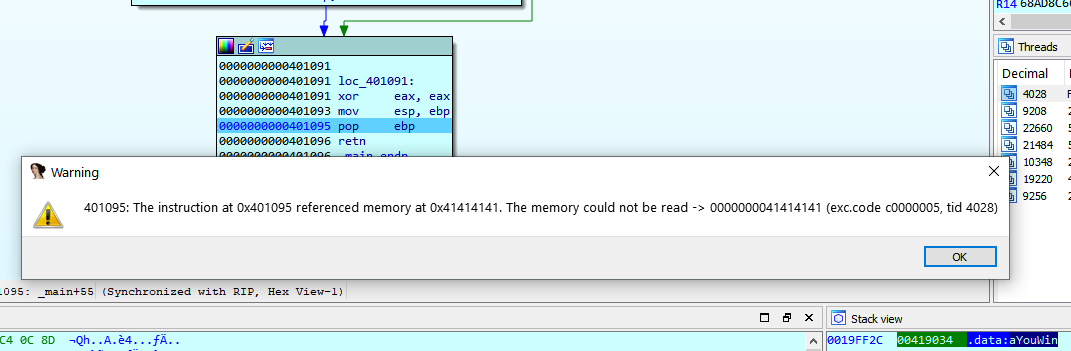
Esto es porque la forma de usar el stdin y stdout del PYCHARM es diferente, pero si lo ejecuto desde una consola de windows aunque se rompe el programa mostrará **YOU WIN**.



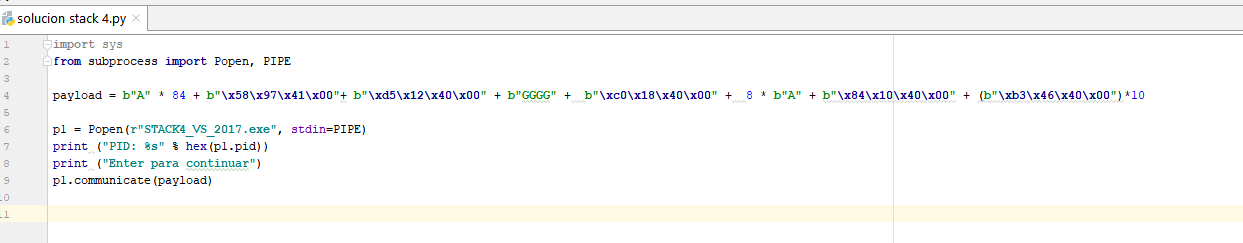
Igual seguiré traceando a ver donde se rompe.



Vemos que el programa vuelve a ejecutar el EPÍLOGO pero con EBP valiendo 0x41414141 y le pasa ese valor a ESP ahí mismo el programa crasheara al tratar de leer el siguiente POP EBP desde una dirección ni siquiera allocada.



Igual para los que recién comienzan esta es una solución perfectamente válida, no importa que el programa se rompa, la solución que pondré a continuación sin ejecutar código en el stack y sin hardcodear ninguna dirección del mismo, requiere de más conocimiento y es más difícil, ya para niveles más avanzados.



Lo voy a tracear un poco, no voy a explicar como halle todo esto porque es un trabajito jeje.

**import sys**

**from subprocess import Popen, PIPE**

**payload = b"A" \* 84 + b"\x58\x97\x41\x00"+ b"\xd5\x12\x40\x00" + b"GGGG" + b"\xc0\x18\x40\x00" + 8 \* b"A" + b"\x84\x10\x40\x00" + (b"\xb3\x46\x40\x00")\*10**

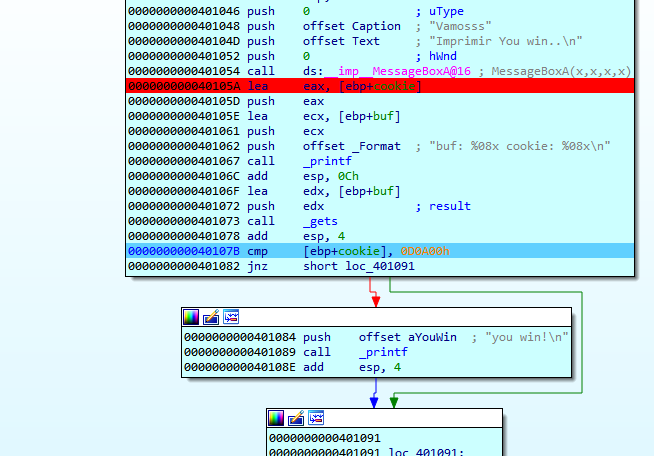
**p1 = Popen(r"STACK4\_VS\_2017.exe", stdin=PIPE)**

**print ("PID: %s" % hex(p1.pid))**

**print ("Enter para continuar")**

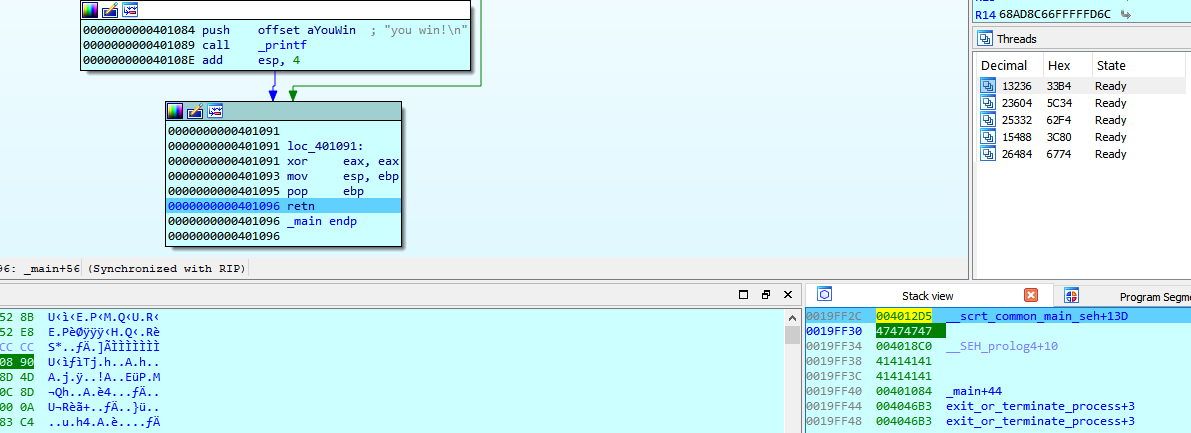
**p1.communicate(payload)**

Lo corro desde una consola de Windows y atacheo al salir el MessageBoxA.



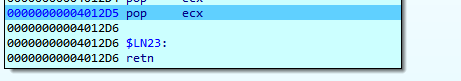
Como la cookie no es igual, salta a **0x401091**.

Llego al RETN



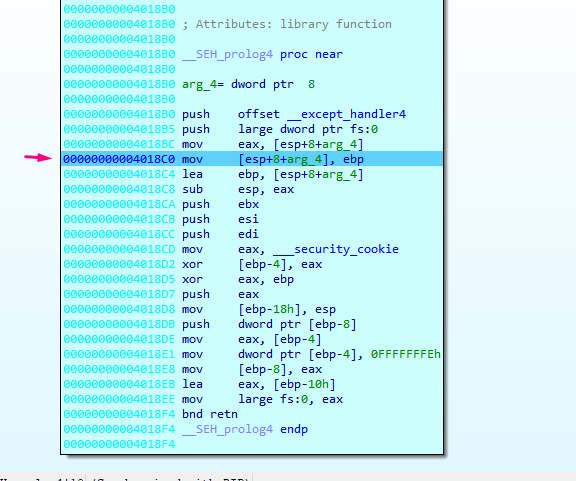
Salto a **0x4012d5** que es código del mismo ejecutable asi que no hay problema ninguno con ASLR, DEP ni nada.

Eso va a ejecutar un POP ECX-RET esto acomoda el stack.





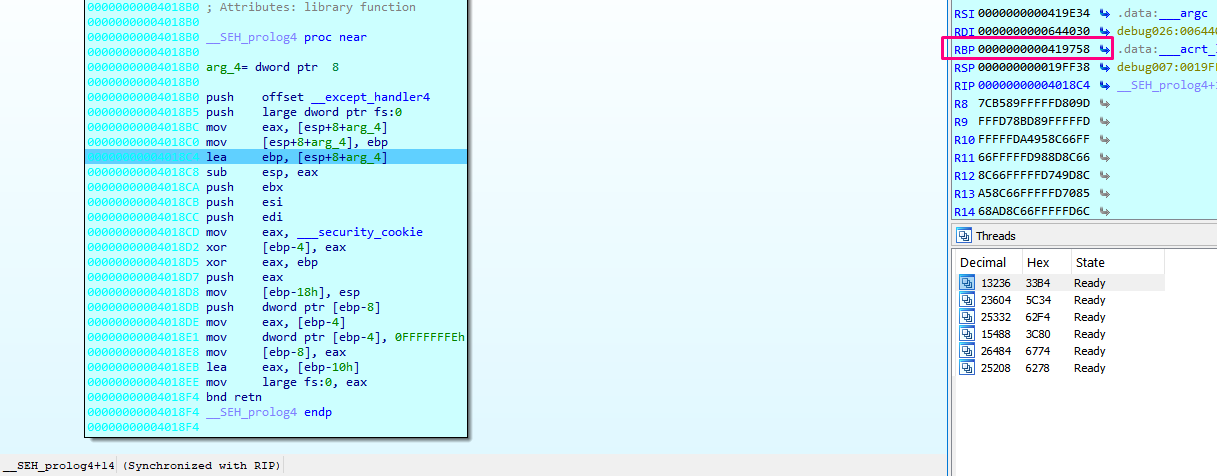
Vemos que mi payload tiene a continuación “**GGGG**” que es 0x47474747 que es un valor no importante que se mueve a ECX y luego el siguiente RET hace que el programa salte al valor siguiente de mi payload que es **0x4018c0**.



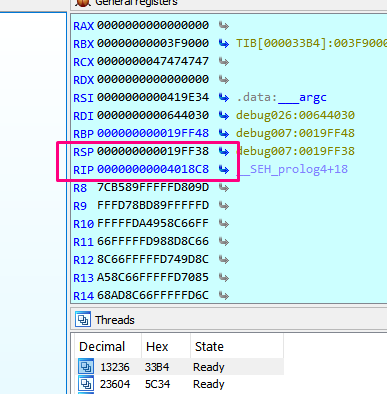
Esta funcion a la que saltó allí llamada SEH\_Prolog, realiza el mismo trabajo que el PRÓLOGO de la funcion que hemos visto, entre otras cosas esta hecha para igualar ESP y EBP y luego mover ESP hacia arriba el valor que se le pasa por argumento.

A mi lo que me importa es que el **LEA EBP, [ESP + XXX]** pondrá de nuevo a EBP en un valor del stack, corrigiendo el problema de que pudiera crashear el programa, si lo hacemos bien.

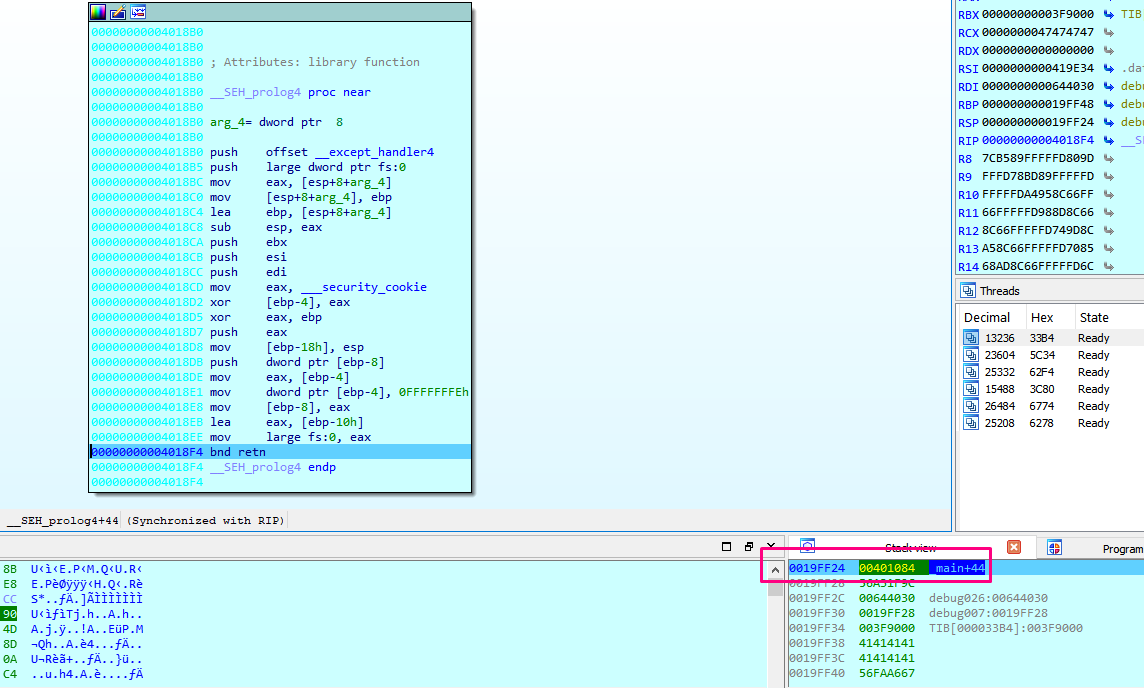
Si lo traceo veo que EBP antes del LEA vale **0x419758.**



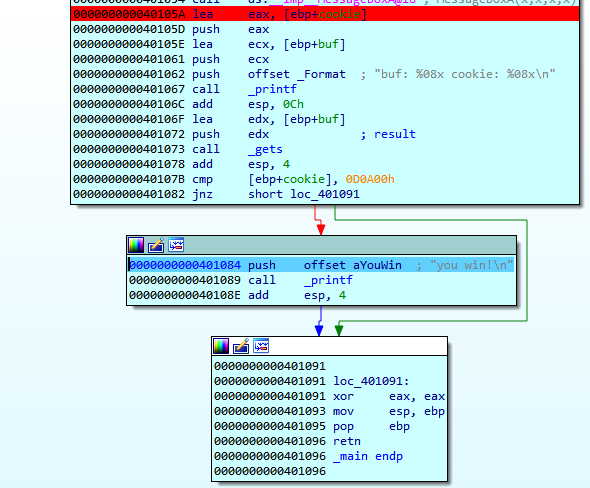
Y después del LEA, en mi máquina vale 0x19ff38.



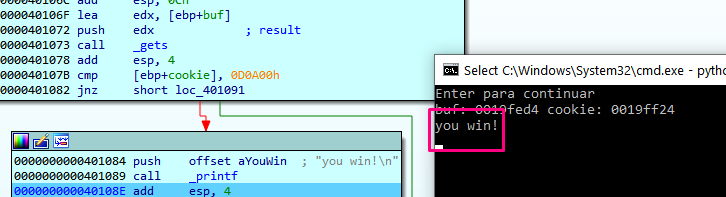
Con lo cual lo ponemos de vuelta a EBP en un valor del stack, el que nosotros deseamos y como depende del valor de ESP el POP ECX-RET anterior lo movió justo donde yo necesito, ya verán al terminar de ejecutar todo, sigo traceando.

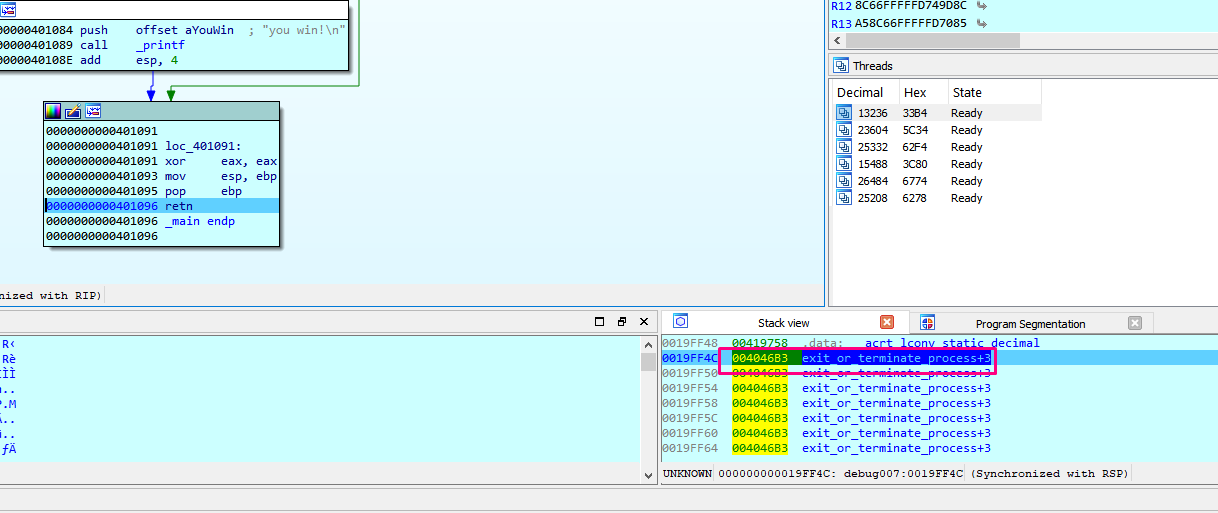


Al llegar al RET salta ahora a **0x401084**, que es el bloque del YOU WIN, pero ahora con EBP correcto. sigo traceando.

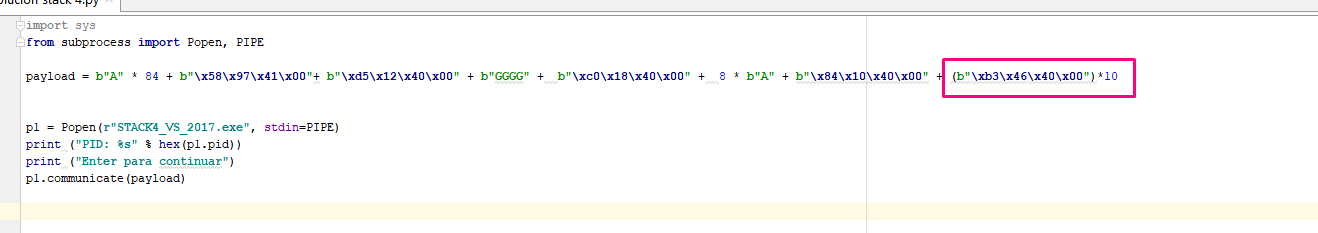


Ahi esta, pasó por encima con F8 traceando y ya se ve el YOU WIN, ahora solo queda terminar sin crashear sigo traceando.

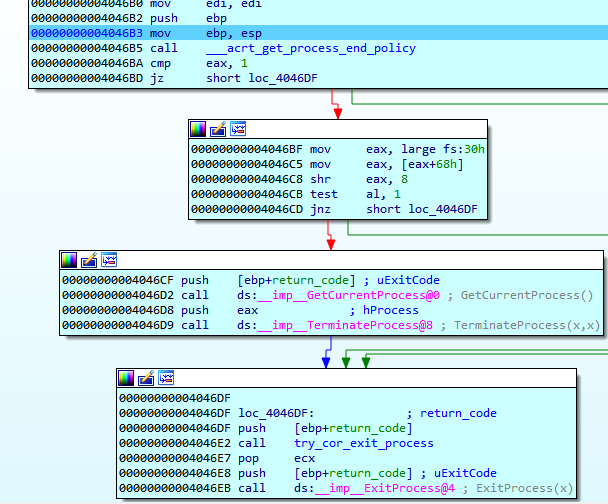




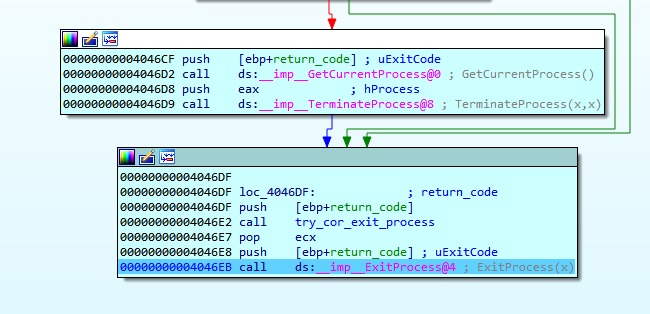
Veo que cuando llega al RETN final lo hago saltar a exit esa parte de mi script es esta.



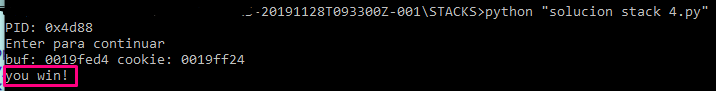
Veo que EBP se volvió a poner en un valor malo, pero decido no saltar a la primera instrucción de la funcion sino aqui.



Donde pisa EBP con el valor correcto de ESP y luego sigue a EXITPROCESS.



Llego a ExitProcess donde se cierra el programa sin error.



En la parte siguiente empezaremos con el ABO1 usando RADARE, en ese hay que ejecutar la calculadora o el notepad a ver si alguien se anima a hacerlo.

Hasta la parte 6

Ricardo narvaja

28/11/2019