EXPLOITING Y REVERSING

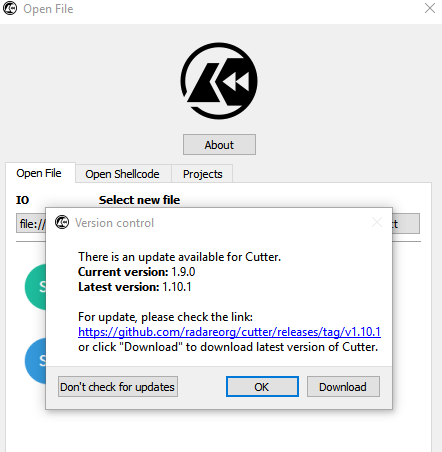
USANDO HERRAMIENTAS

GRATUITAS (PARTE 9)

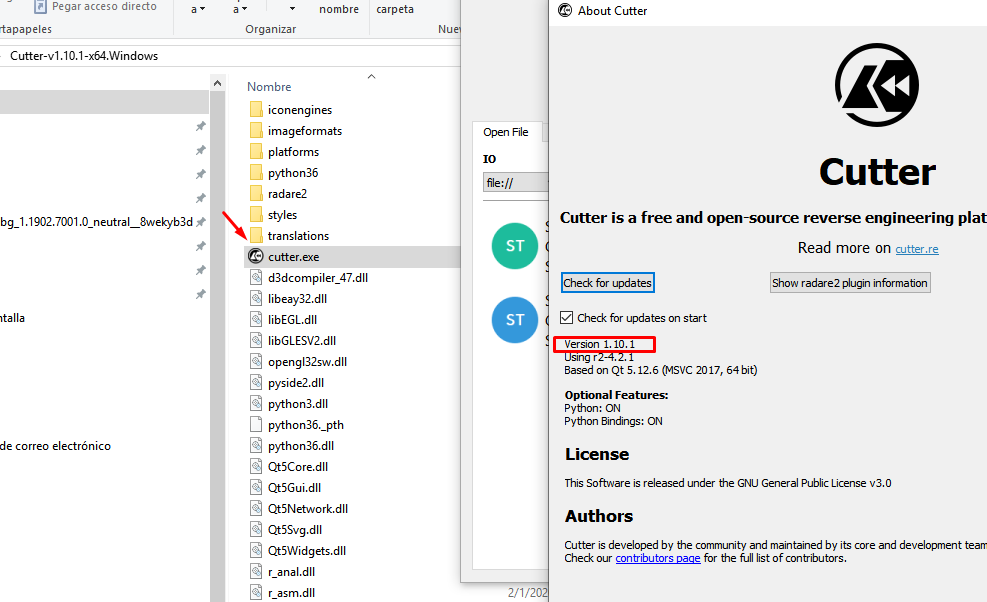
Hemos solucionado el ABO3 en IDA FREE y ahora nos toca usar radare para solucionar el ABO4.

ACTUALIZANDO RADARE Y CUTTER

Hay una nueva versión publicada de Cutter que es la GUI de radare, cuando abro la antigua versión me pide updatear.



Presiono DOWNLOAD y una vez unzipeado ejecutamos el cutter.exe y verificamos que está actualizado.



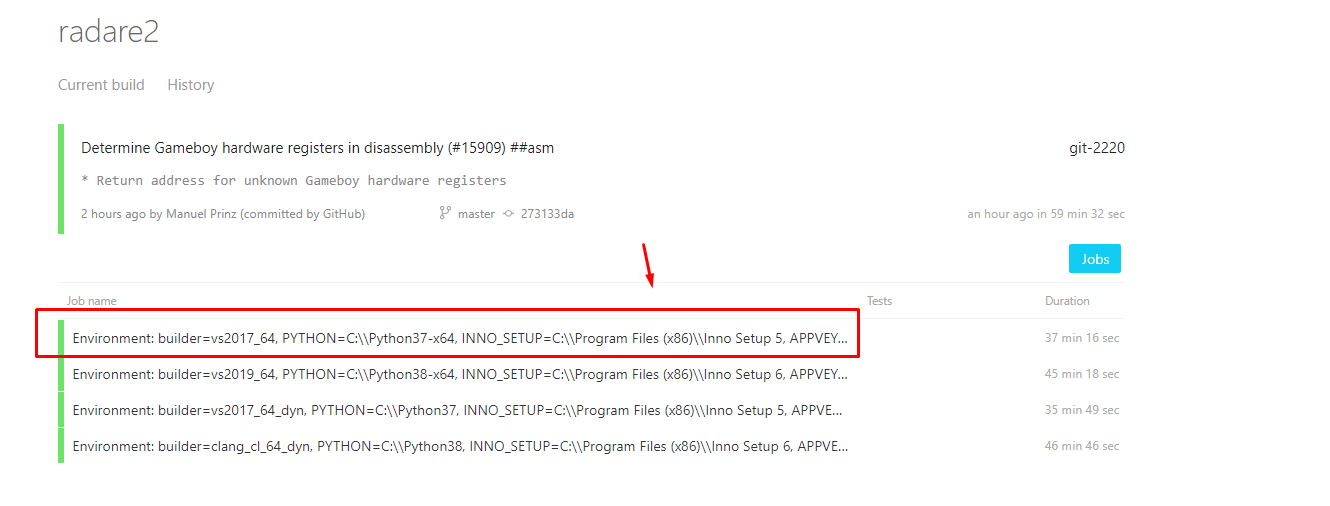
Veré si hay algún build más nuevo del propio radare.

[**https://ci.appveyor.com/project/radareorg/radare2/history**](https://ci.appveyor.com/project/radareorg/radare2/history)

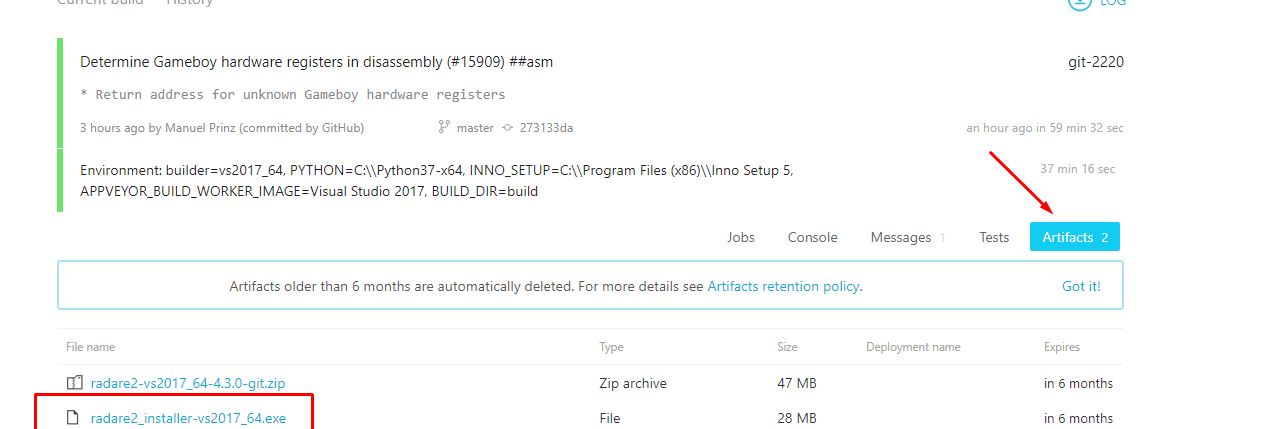
En el dia de hoy el build que está más arriba y en verde es este, hago click para entrar.

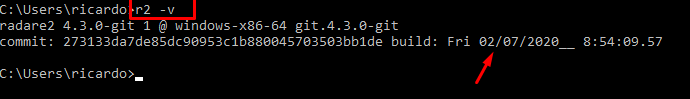
****

Busco el de más arriba en esa lista y entro.



Voy a ARTIFACTS y está el instalador lo descargo e instalo.





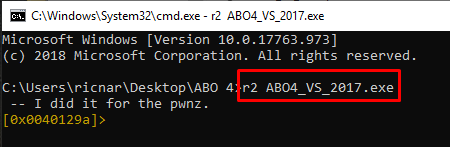
Vemos con el comando **r2 -v** que la versión es del dia de hoy (estoy escribiendo esto el 07/02/2020)

Como el problema para atachear aún no fue resuelto, debuggearemos en x64dbg y haremos el reversing estático en radare y Cutter.

REVERSING DEL ABO4 USANDO RADARE.

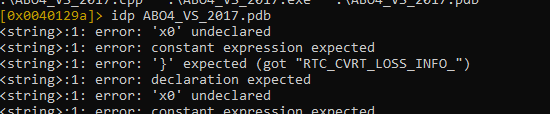
Bueno abriré el ejecutable del abo4, como estoy situado en el mismo path, puedo llamar directamente por el nombre del ejecutable, sino con **cd** conviene cambiar al path del ejecutable del abo4.

**r2 ABO4\_VS\_2017.exe**

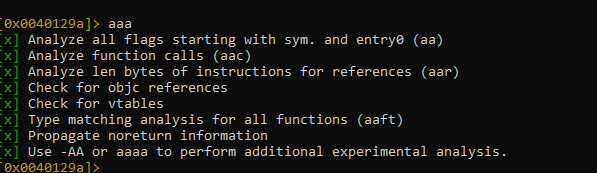
****

Cargo los símbolos con **idp**

**idp ABO4\_VS\_2017.pdb**

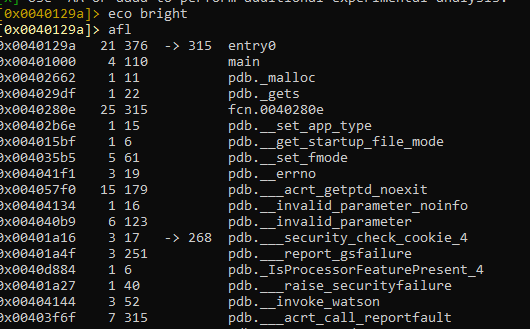


y luego con **aaa** analizo el programa.



Con **eco bright** puedo ponerlo más brillante.

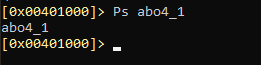
Con **afl** listo las funciones



Vayamos al main con **s main**

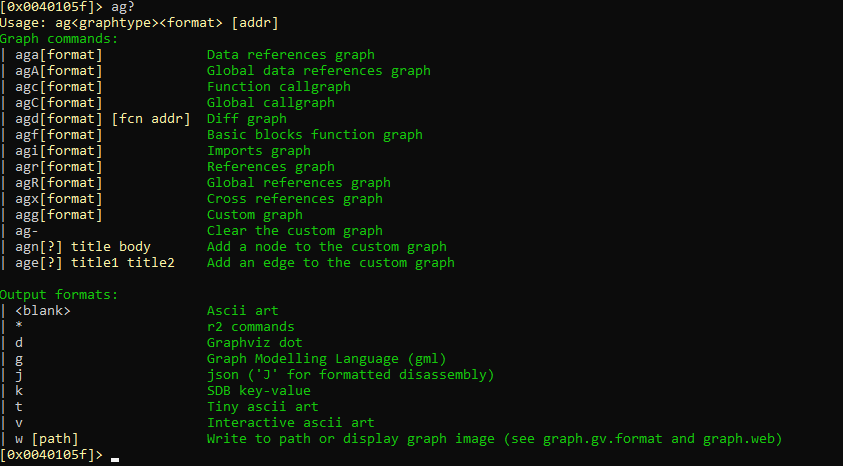


Creo un proyecto con **Ps abo4\_1**

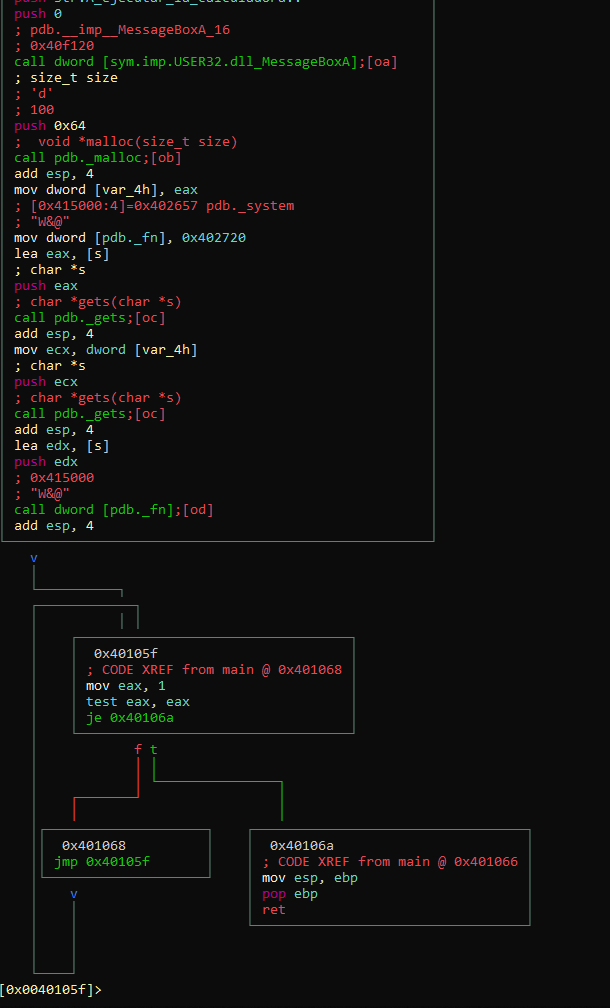
****

En ese proyecto, iré guardando cada tanto con el mismo comando mi reversing estático.

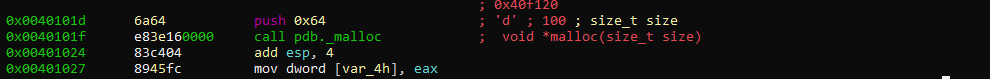
Puedo entrar a modo visual con **v**, pero también puedo mirar las opciones de gráficos ASCII con **ag?**

****

Usando **agf** veré los bloques de la función con eso me alcanza.



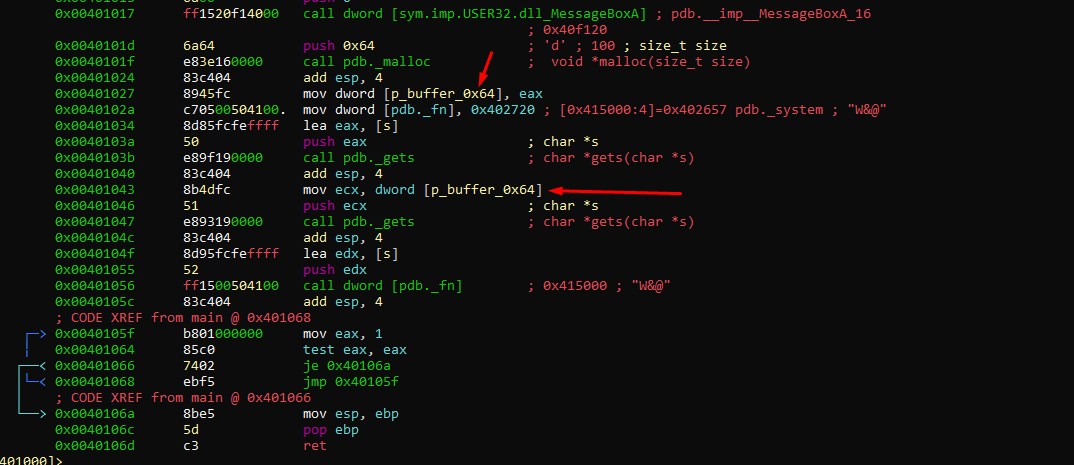
Hay una llamada a **malloc()** para reservar 0x64 bytes de memoria, vemos el size 0x64 como argumento.



Le vamos a cambiar el nombre de la variable donde guarda el puntero a dicha memoria reservadallamada **var\_4h**

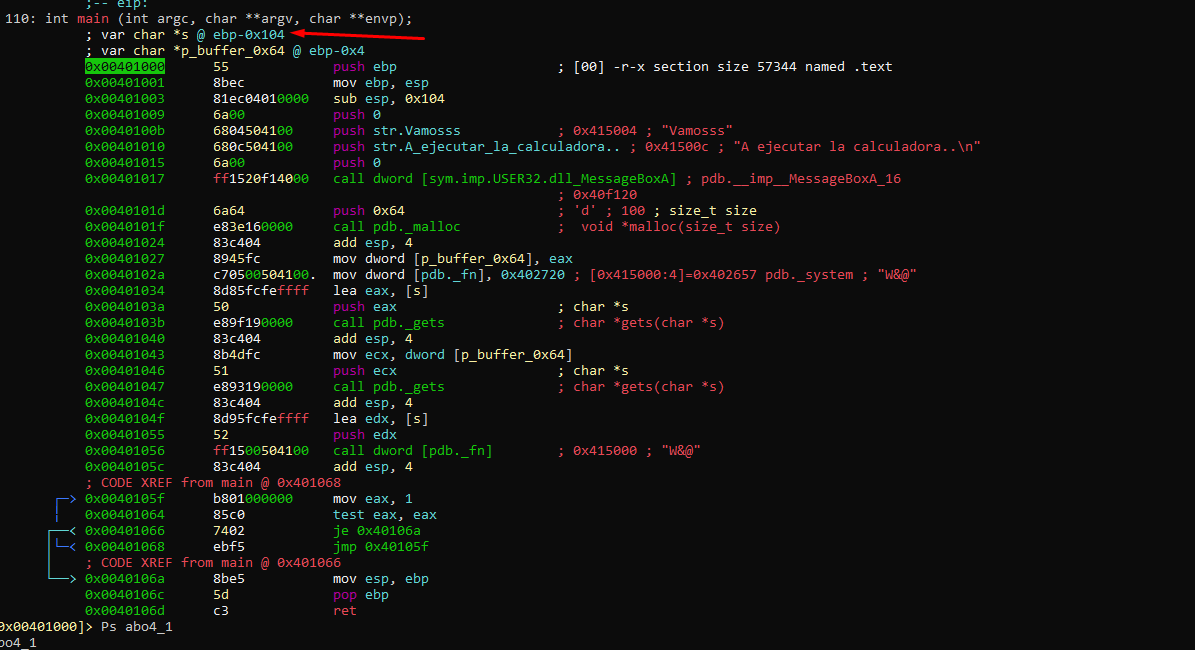
Renombro las variables con **afvn nuevo\_nombre viejo\_nombre**

**afvn p\_buffer\_0x64 var\_4h**

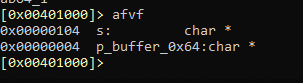
****

Vemos que hay dos llamados consecutivos a **gets()** con diferente argumento.

La primera llamada usa un puntero a la variable **s** como argumento, veamos que es dicha variable **s**.



La variable **s** es un buffer en el stack veamos su largo usando **afvf.**

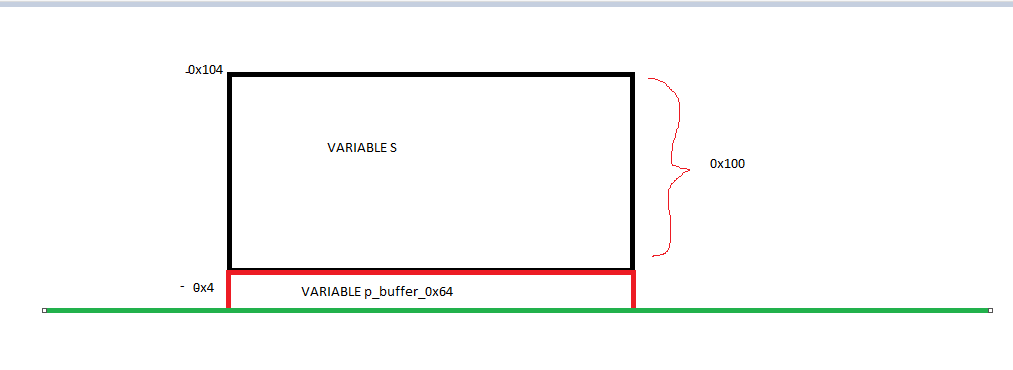


Coincide con



La variable **s** está ubicada en **ebp - 0x104** o sea 0x104 bytes arriba del horizonte y **p\_buffer\_0x64** está en **ebp-0x4** o sea 0x4 bytes arriba del horizonte que es **EBP.**

O sea que haciendo la resta obtenemos el largo de **s** que sería 0x100.



Pisar el return address en este caso no sería una posibilidad ya que hay un loop infinito que no deja llegar al RETN de la funcion.



Como EAX vale 1 y se testea contra sí mismo no va a ir nunca al bloque 0x40106a, solo si EAX valdría cero saldria y eso nunca pasara, siempre va a ir por el camino de la flecha verde a 0x401068 y quedará loopeando.

Así que no se puede explotar pisando el return address, aunque podría hacerse mediante la explotación por medio de pisar el SEH, si el ejecutable fuera SAFE SEH OFF.

Veamos utilizo el script de Powershell

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

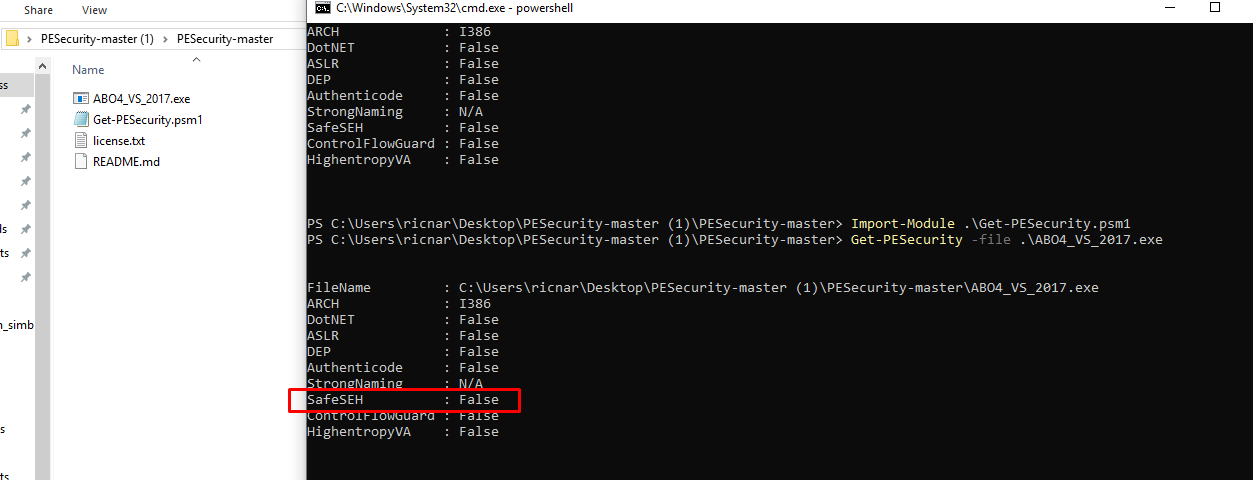
Abro una consola de Windows y tipeo **powershell**

y luego yendo a esa carpeta donde esta el script

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

**Get-PESecurity -file .\ABO4\_VS\_2017.exe**

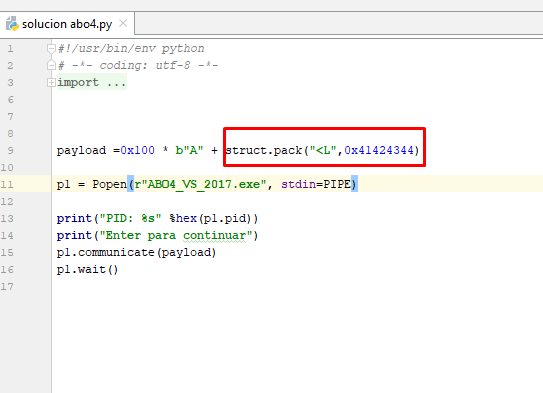
Si el ABO4 está en otra carpeta poner el PATH completo.



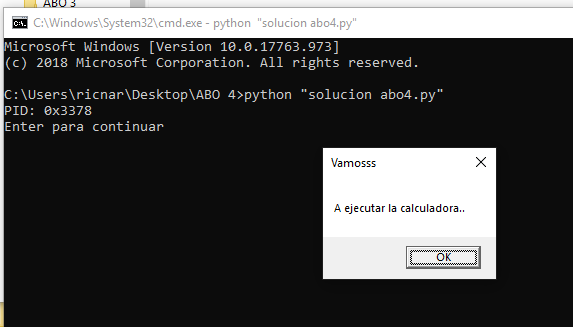
Es safe seh false podría hacerse, les dejo este ejercicio para practicar pisar el SEH y explotarlo a ustedes, yo lo haré de otra forma.

Vemos que el primer argumento a **gets()** es el buffer **s** , eso quiere decir que podriamos desbordar **s** y pisar el valor de la variable que está debajo que es donde guarda el puntero al malloc o sea **p\_buffer\_0x64.**

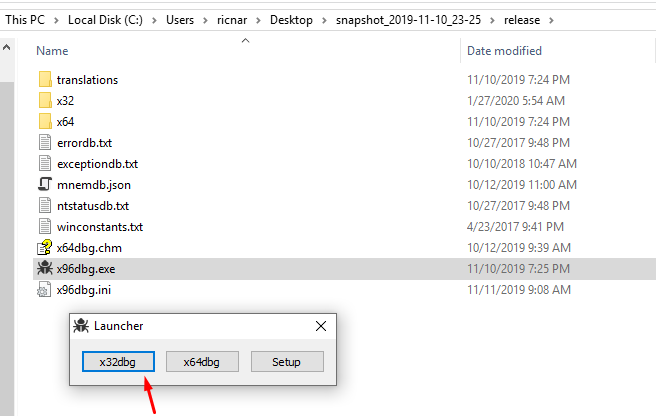
Si quisiéramos pisar el valor de esta variable **p\_buffer\_0x64** deberíamos escribir un script de esta forma.

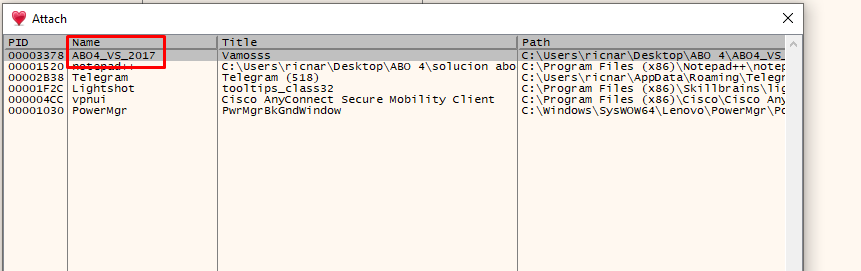


Lo ejecuto

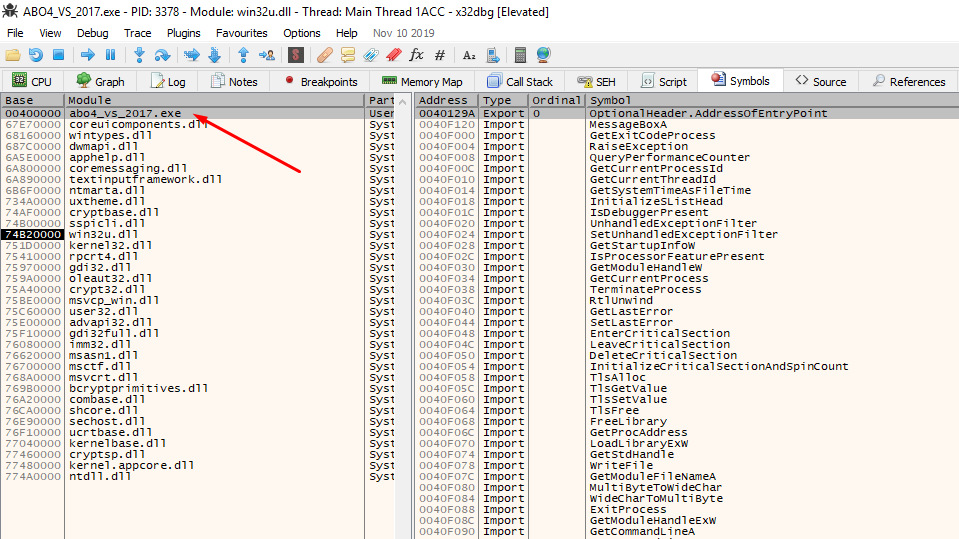


Y lo atacheo con el x64dbg de 32 bits.

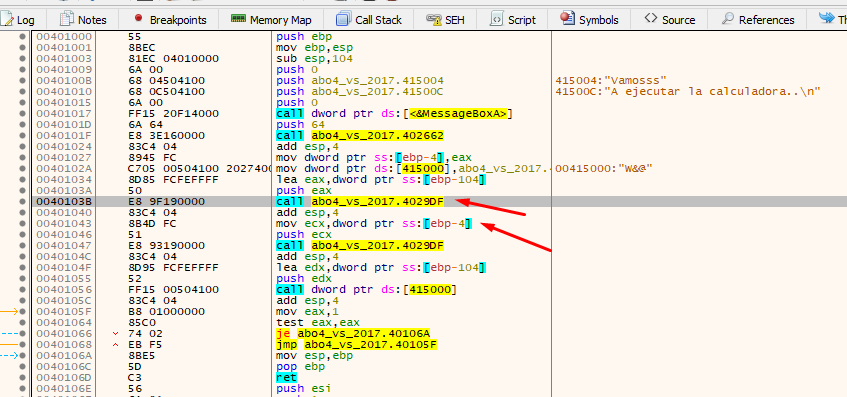




Con VIEW-MODULES en la lista de módulos elijo el ejecutable.

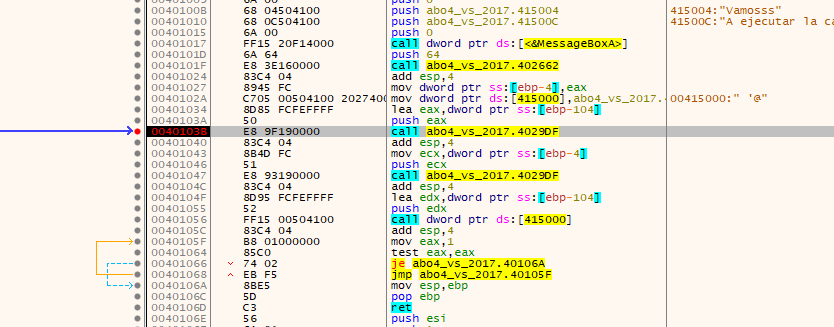


Hago doble click en el nombre y me lleva al código de dicho ejecutable.

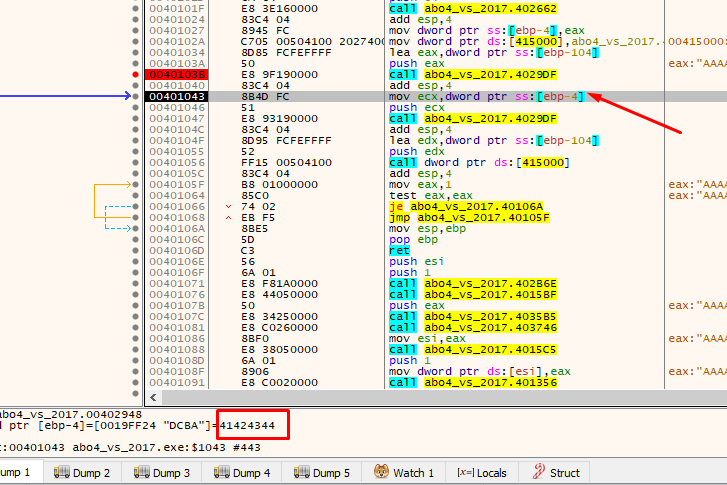


Alli veo los dos llamados a **gets** pongamos un breakpoint en el primero.

Al aceptar el MessageboxA, se detiene en el breakpoint, paso el **gets** con f8.

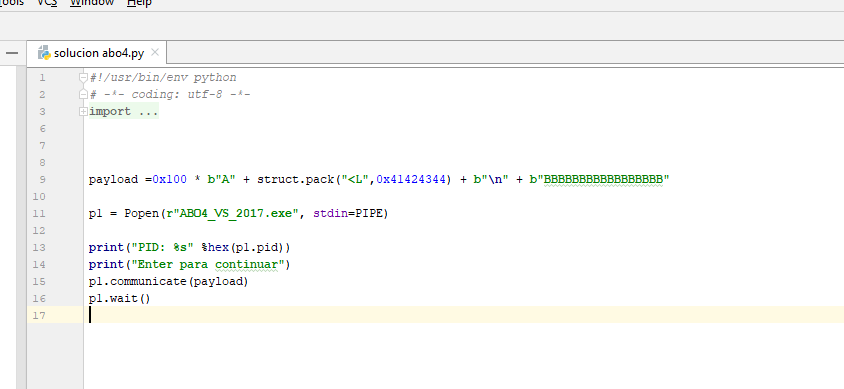


Vemos que la variable que está en ebp-0x4 que guardaba el puntero a malloc llamada **p\_buffer\_0x64** fue pisada con 0x41424344 por el overflow, así que hasta ahí vamos bien.



Vemos que si no hubo overflow, llama al segundo **gets**, con el valor de **p\_buffer\_0x64** como argumento, pero como nosotros pisamos dicho valor con 0x41424344 ahora el **gets** tratara de escribir alli, pero la dirección 0x41424344 no existe. Si paso con f8 el segundo gets() no da error porque no enviamos más bytes para que ingresen en el segundo **gets**, ya que todos los bytes que envie entran en el primero y en el segundo no copia nada.

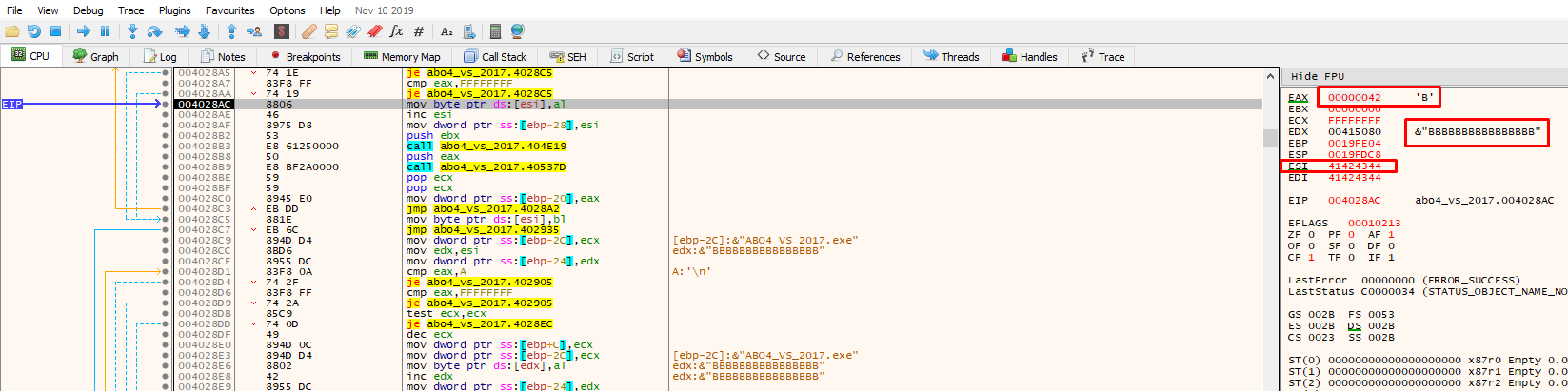
Para poder enviar data al segundo **gets** y como cada entrada de datos termina cuando uno apreta la tecla ENTER, se puede hacer programáticamente, poniendo **\n** y a continuación lo que queremos que ingrese en el segundo gets.



Se supone que el **\n** separará lo que entra en el primer gets() de lo que entra en el segundo.

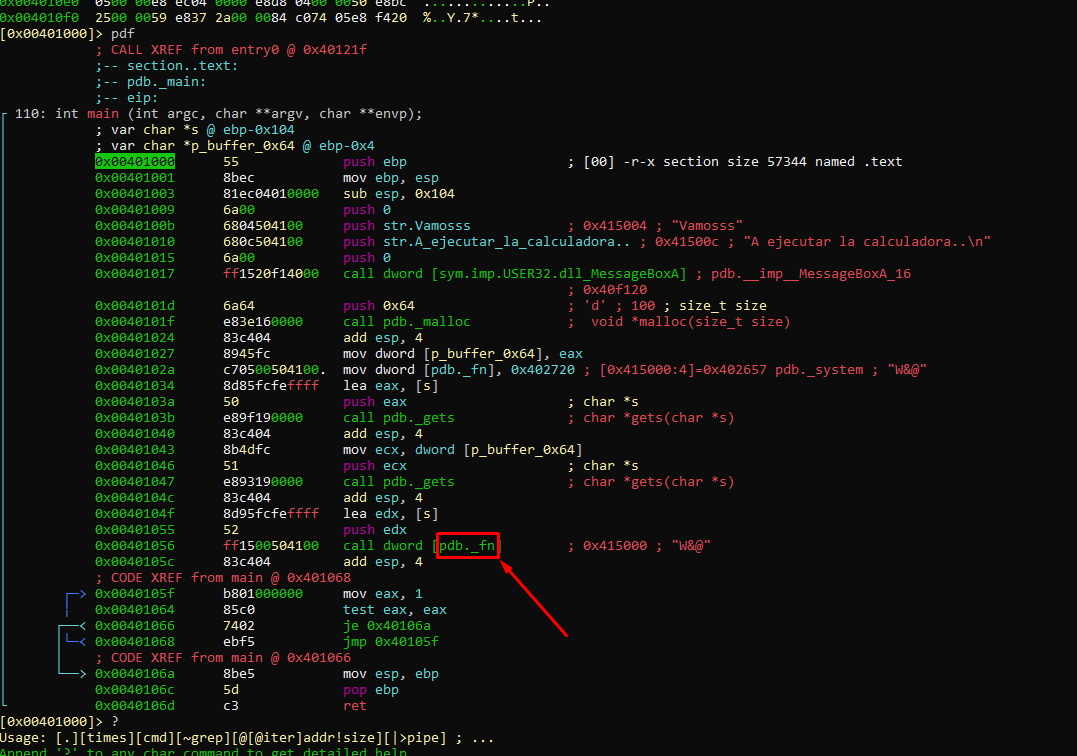
Por lo tanto en el segundo **gets** entraran las B.

Si corremos este script, pisa el valor de la variable **p\_buffer\_0x64** pero crashea al tratar de copiar las B a la dirección 0x41424344.



Pero la cuestion aqui es que cambiando 0x41424344 por una dirección escribible, podemos escribir donde queramos lo que queramos.

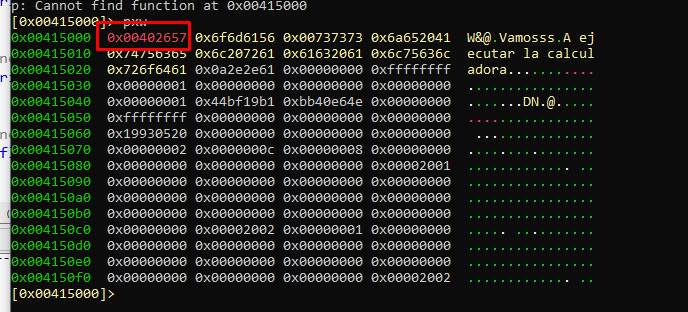
Vemos que hay una variable más en el código que no es local, o sea no está entre las variables, ni entre los argumentos de esta funcion.



A la derecha nos muestra que está alojada en la dirección 0x415000.

y si vamos allí con **s 0x410500**

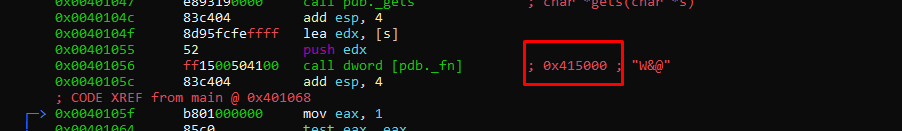
Usando alli **pwx** para ver lo que hay alli, vemos que inicialmente tiene ese valor **0x402657**.



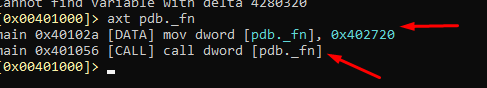
Puedo ir alli y ver que corresponde a la funcion system.



O sea que originalmente la variable **\_fn** tiene la dirección de system.



Con **axt pdb.\_fn** puedo ver los lugares donde es usada.



Vemos que se usa dos veces en el main, en 0x40102a y en ese call en 0x401056.

En 0x40102a escribe 0x402720 veamos que hay allí.

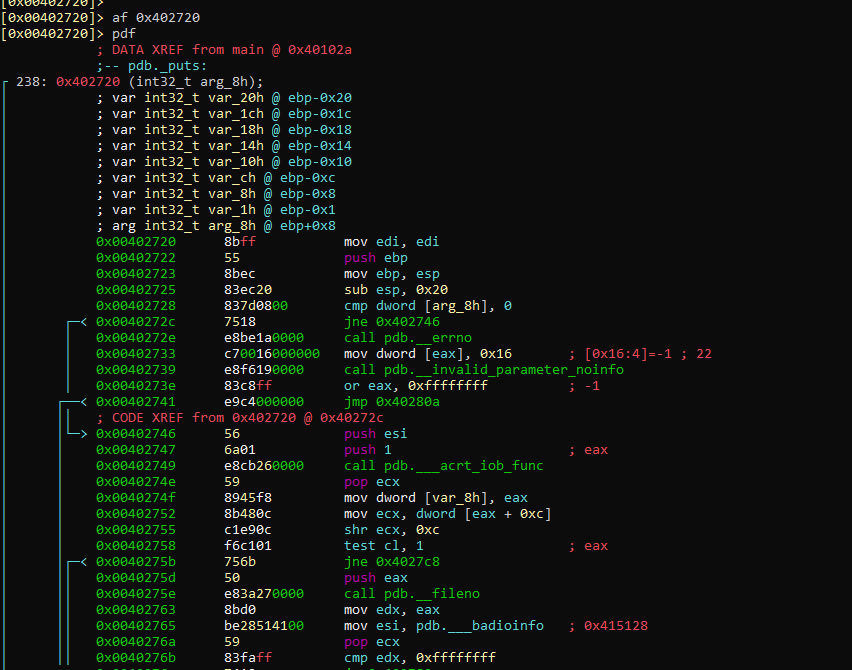


Radare no la interpreta como función, le decimos que la analice como función.

**af 0x402720**

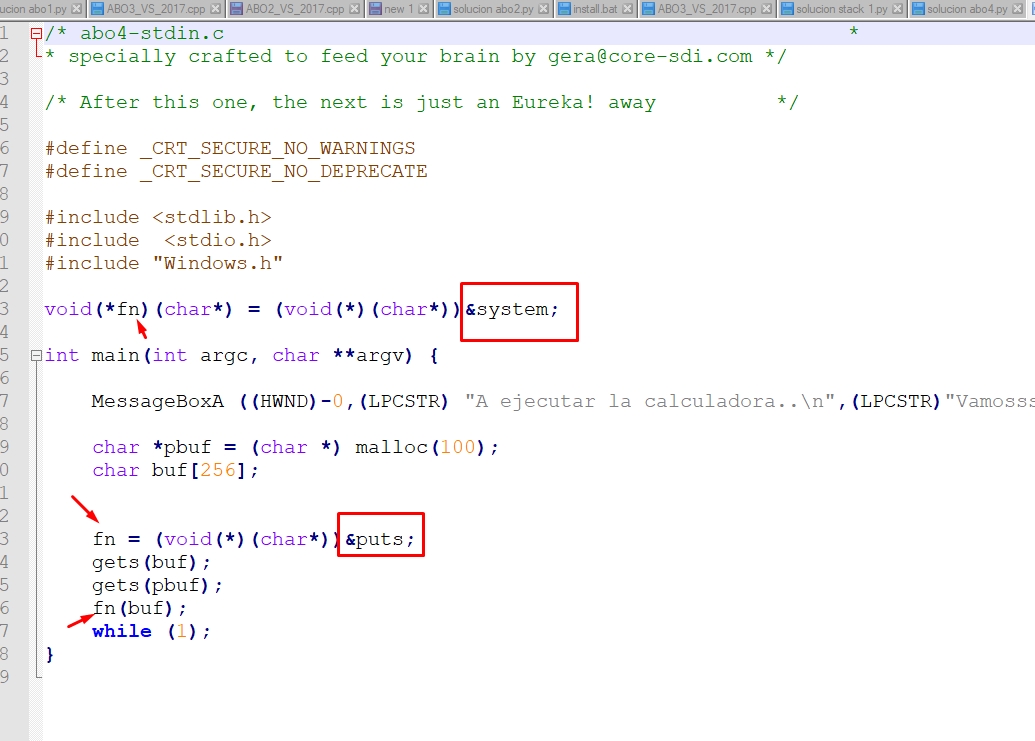
Ahora intentemos de nuevo **pdf**.

Vemos que es la función **puts**.

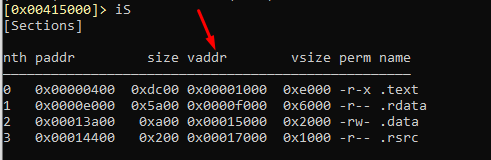
****

O sea que al igual que el ejercicio anterior hay una variable **\_fn**, en este caso es global y que está inicializada con la dirección de **system** y luego pisa su valor con la dirección de **puts**.

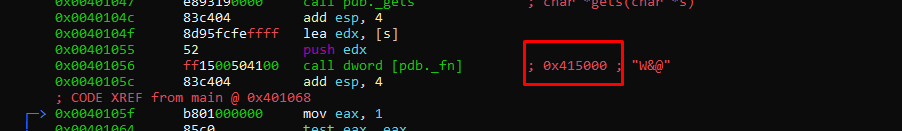
Mirando el archivo cpp del código fuente vemos que es correcto nuestro análisis.



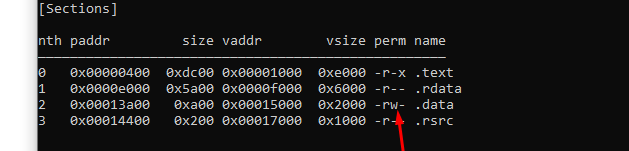
Bueno así que si pisamos el valor de la variable global **\_fn** saltaremos donde queremos y vimos que está en una dirección fija escribible, ya que está en la sección **data** del ejecutable que tiene permiso de escritura.



**vaddr** son los offset virtuales, a los cuales hay que sumarles la imagebase para obtener la dirección virtual, vemos que la sección data comienza en va 0x15000 si le sumamos la image base 0x400000 nos da 0x415000.

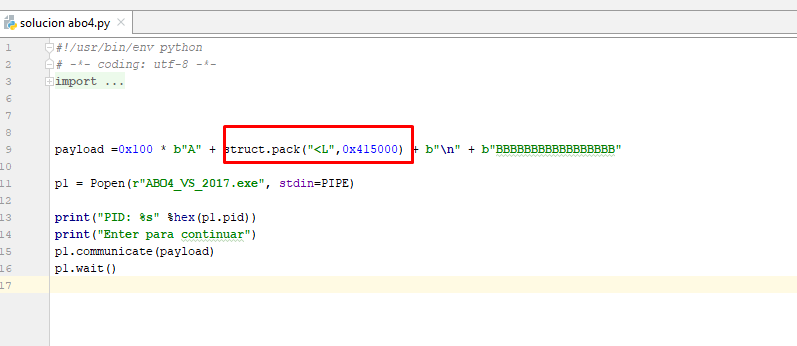


Alli comienza la sección data, es la dirección de **\_fn** y es escribible vemos la W ( permiso de escritura - write)

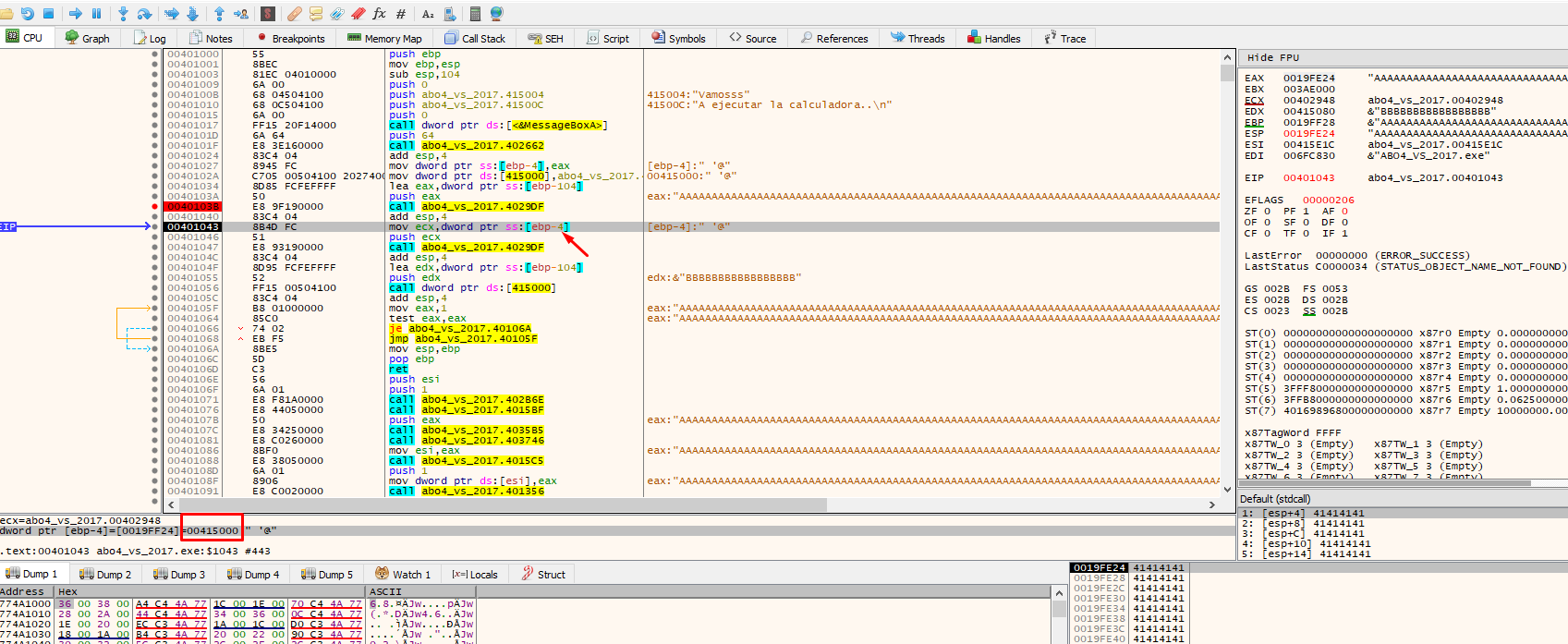


Así que podemos escribir allí, así que si al valor de la variable **p\_buffer\_0x64** que antes habíamos pisado con 0x41424344 en el primer **gets(),** la pisamos con 0x415000, este valor lo usara como argumento del segundo gets y escribirá lo que queremos en dicha dirección.

Probemos.

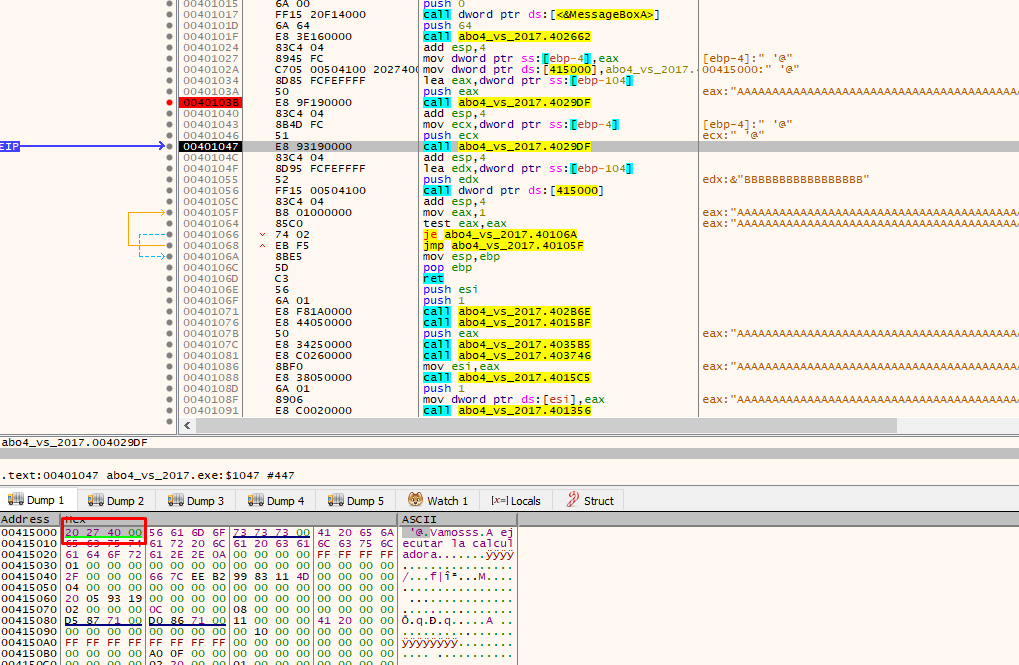


Lo corro y atacheo con x64dbg.

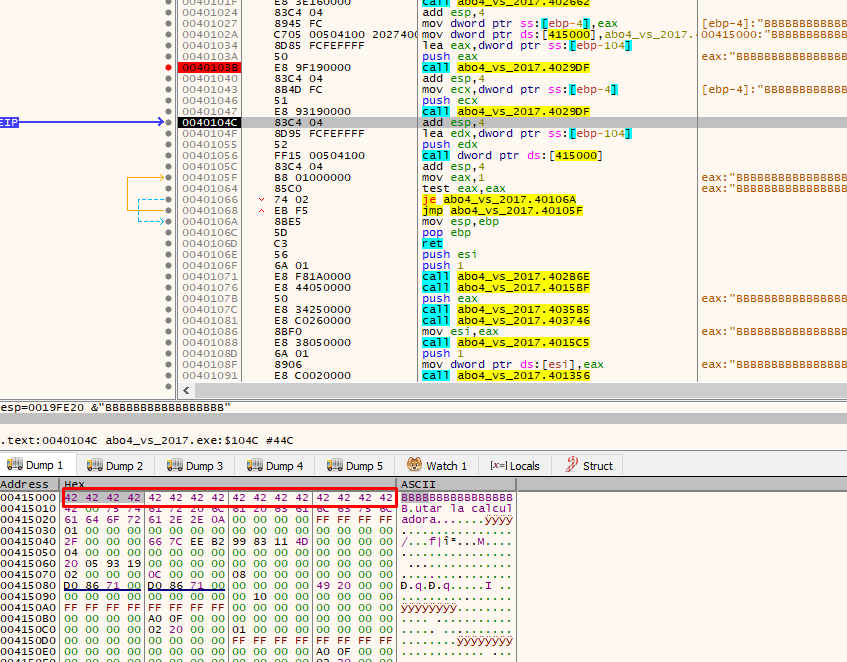


Vemos que al pasar el primer **gets** el valor de la variable **p\_buffer\_0x64** queda pisado con el valor 0x415000, que es la dirección de **\_fn**, así que al usar esa dirección en el segundo gets como argumento, escribirá alli las letras B o lo que queramos.

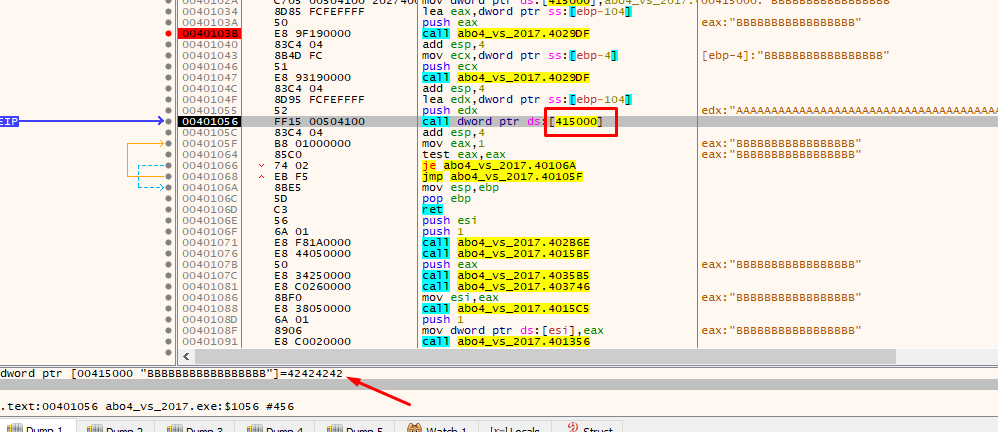
Puedo poner en el hex dump la dirección 0x415000 para ver como escribe alli al pasar el segundo gets.

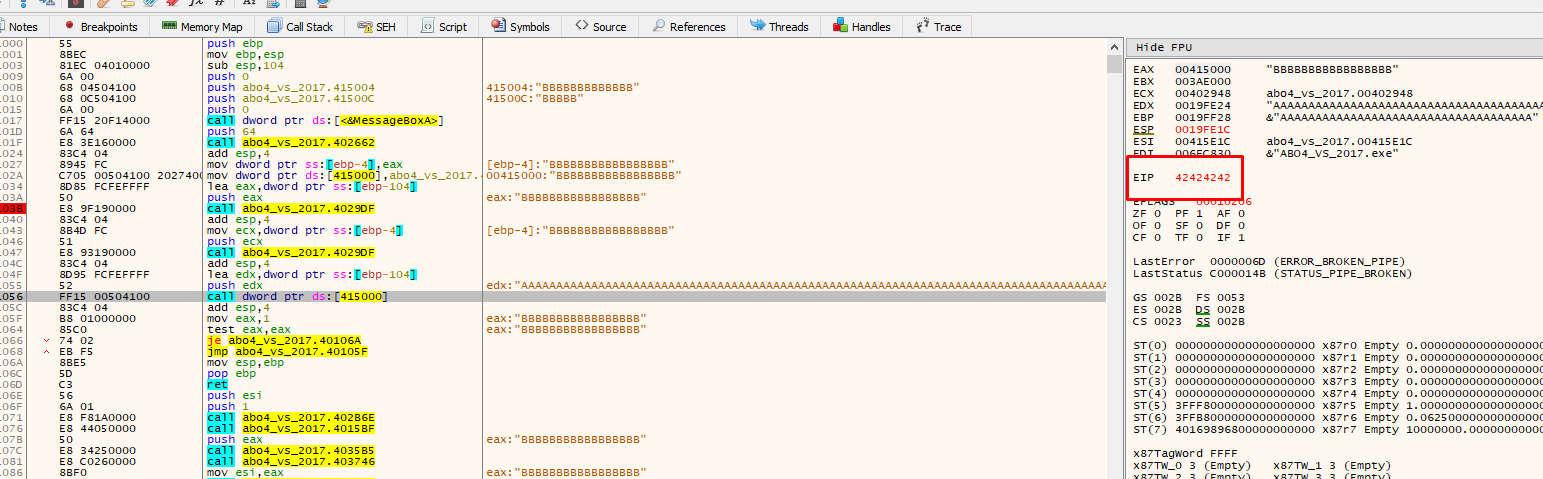


Paso el segundo gets con f8 y veo que alli escribió las B.

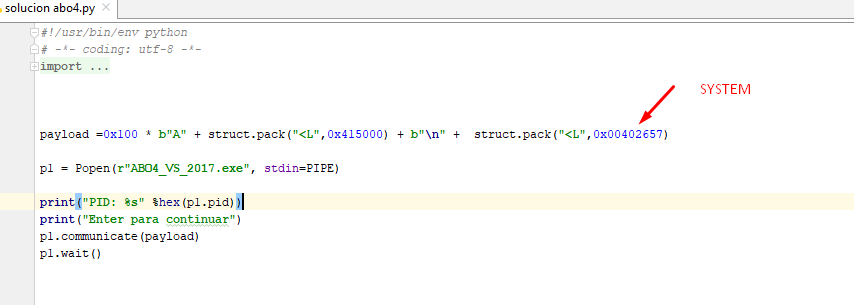


Ahora al llegar al CALL saltara a 0x42424242.

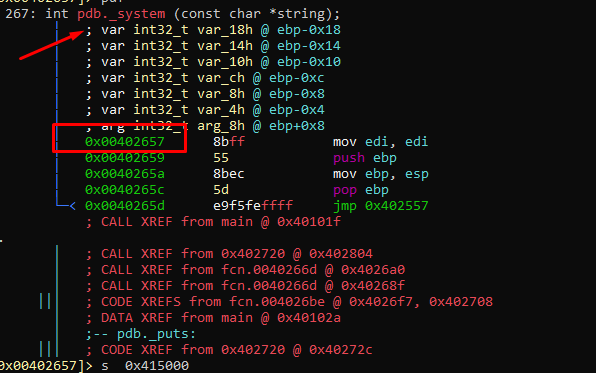




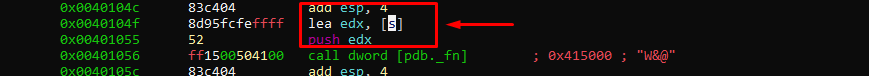
Así que reemplazando las B por las dirección de **system**, saltaría a la función **system**.



Recordemos que **system** estaba en 0x402657.

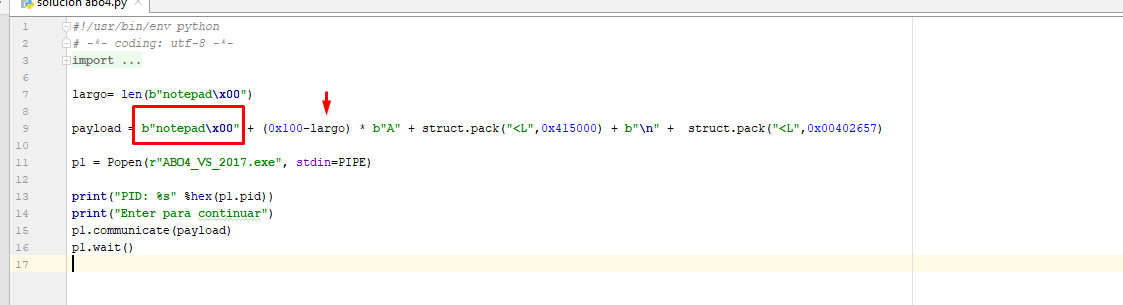


Con eso ya saltaría a **system**, veamos el argumento de dicha función.



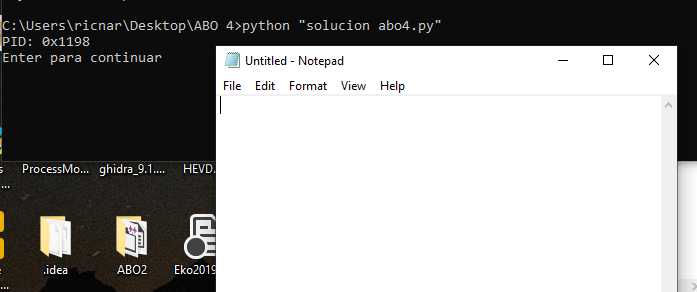
Vemos que el argumento está en el buffer **s** o sea el primer buffer que llenamos

Como ese puntero busca una string allí, tomará los primeros caracteres del buffer hasta que halle un cero, así que podemos poner al inicio de la data con que llenamos ese buffer, el nombre del ejecutable que deseamos correr y un cero y luego el resto de las Aes.



Agrego las string notepad y el cero por ejemplo y le resto a los 0x100 el largo de la misma para que se mantengan los 0x100 correspondientes al largo del buffer **s**.

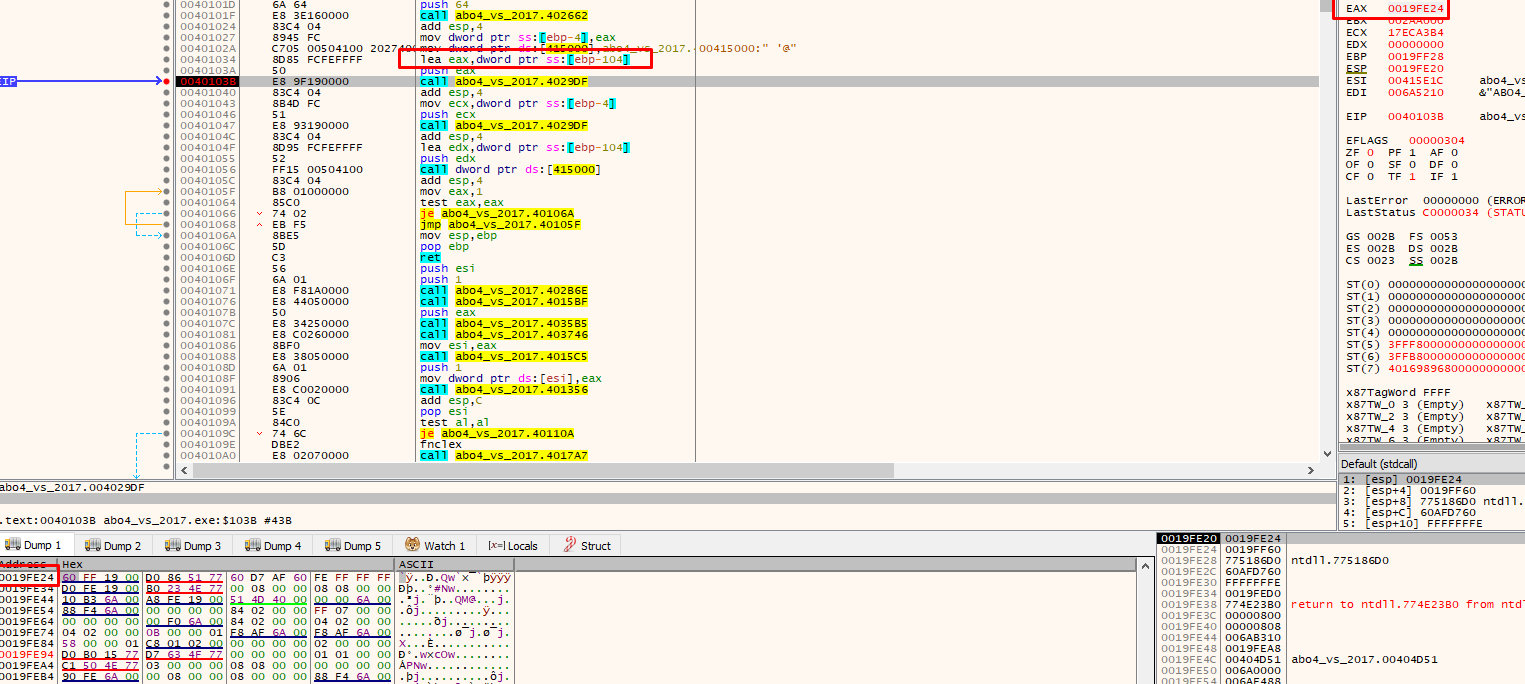
Si ejecuto este script



Funciona perfectamente.

DEBUGGEANDO CON x64DBG

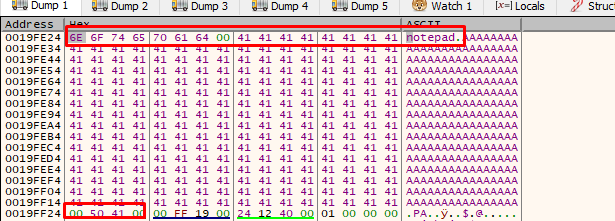
Puedo debuggearlo para verlo paso a paso, lo arranco de nuevo y atacheo en x64dbg.



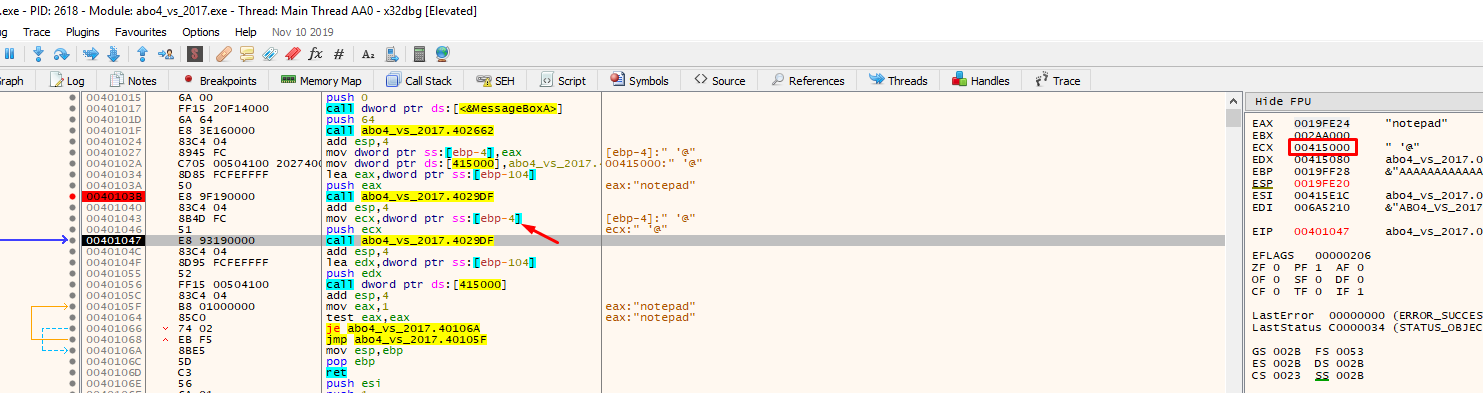
Allí en EAX está la dirección del buffer **s**, lo puedo ver en el HEX DUMP si hago EAX-FOLLOW IN DUMP.

Apreto f8 paso por encima del **gets**.

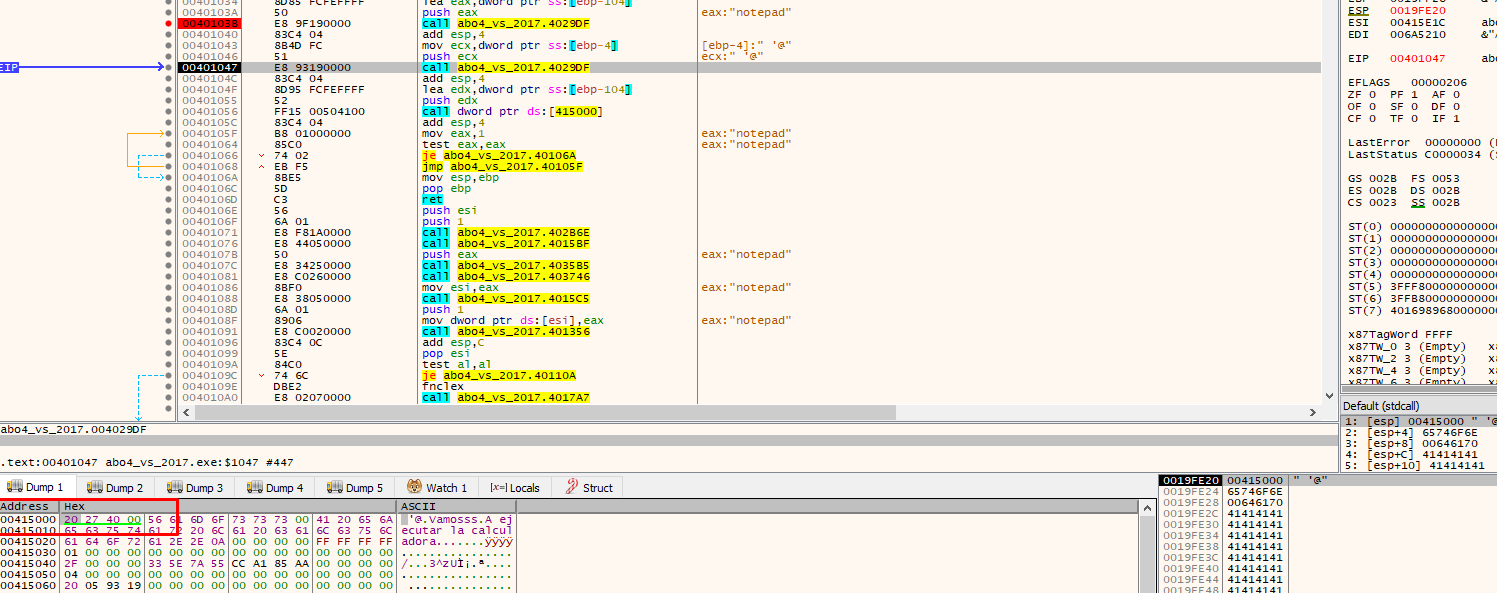
Allí entra mi payload tiene la string notepad al inicio, las Aes y al final la dirección con la que pisara la variable **p\_buffer\_0x64**



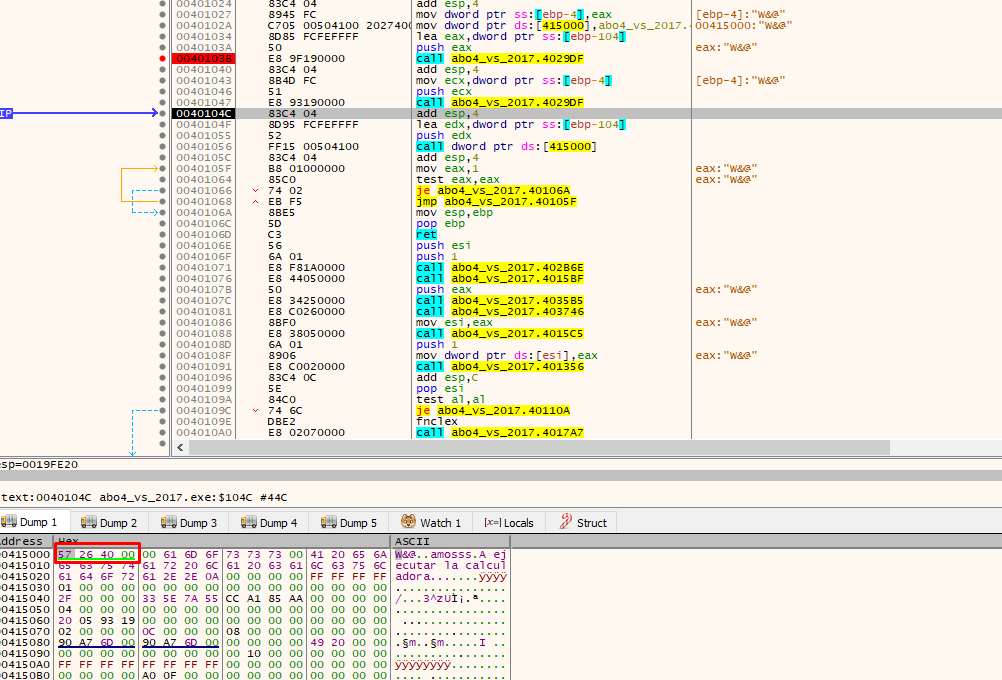
Luego traceo hasta el segundo **gets**, mueve a ECX el valor de **p\_buffer\_0x64** y lo pasa como argumento a **gets** en el mismo escribirá en 0x415000.



Hacemos ECX-FOLLOW IN DUMP para ver como escribe el segundo **gets**

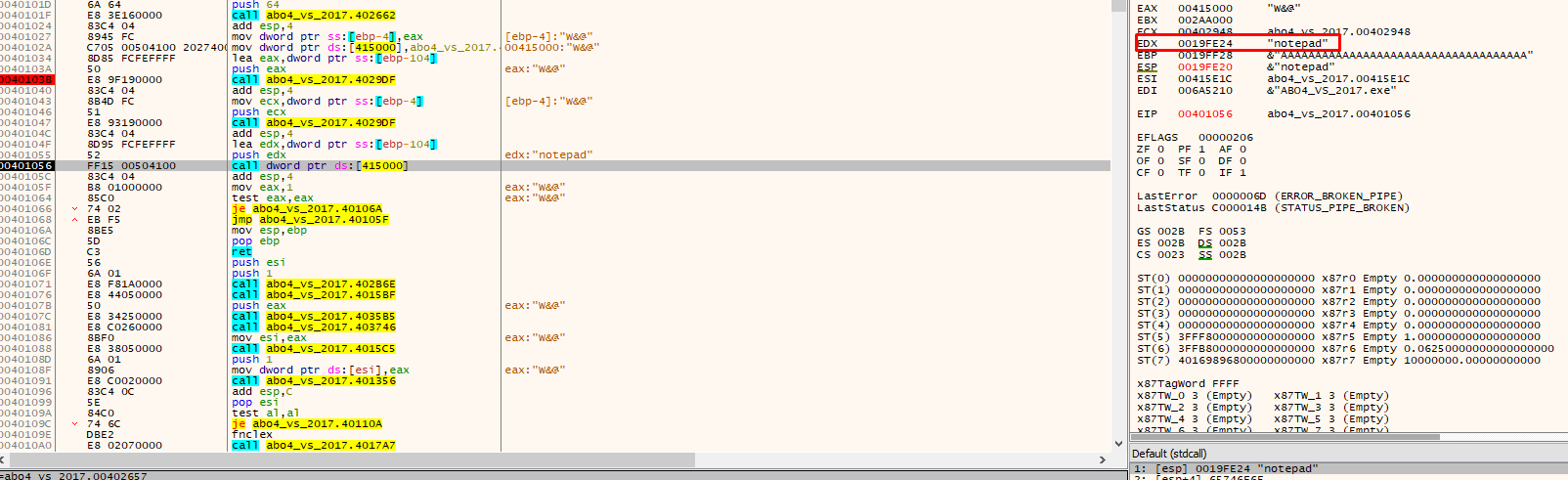


y lo pasamos con F8.



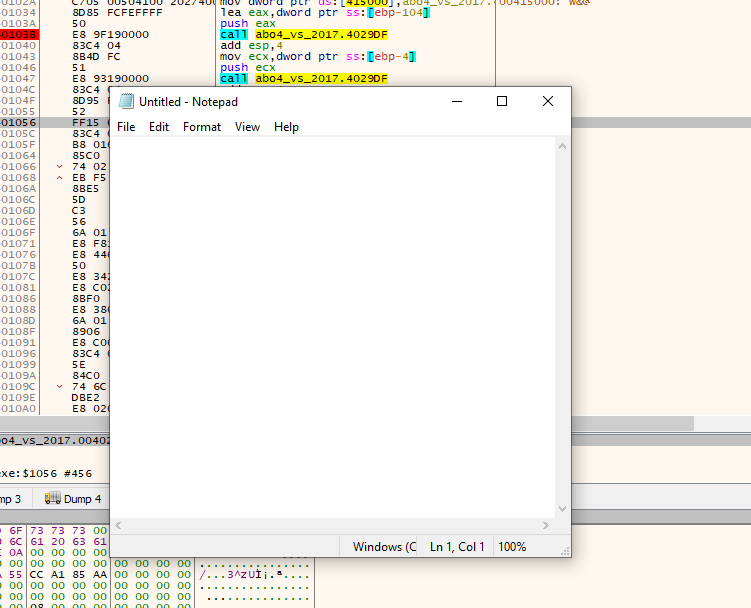
Alli le escribimos 0x402657 la dirección de **system**.

Traceamos hasta el call.



Como EDX apunta al buffer **s** que llenamos con el primer **gets**, al inicio del mismo está la string **notepad** y no continua leyendo más bytes, porque el cero hace que la string se corte allí, ya que busca el nombre del ejecutable o comando a ejecutar y este es una string.

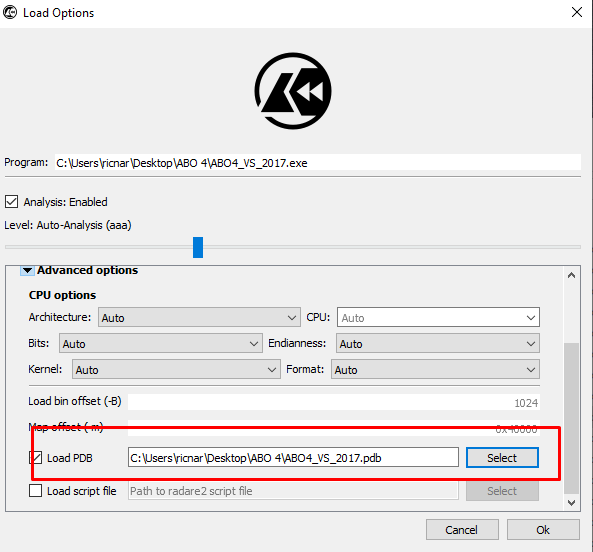
Al pasar por encima del call salta a system y ejecuta el notepad.

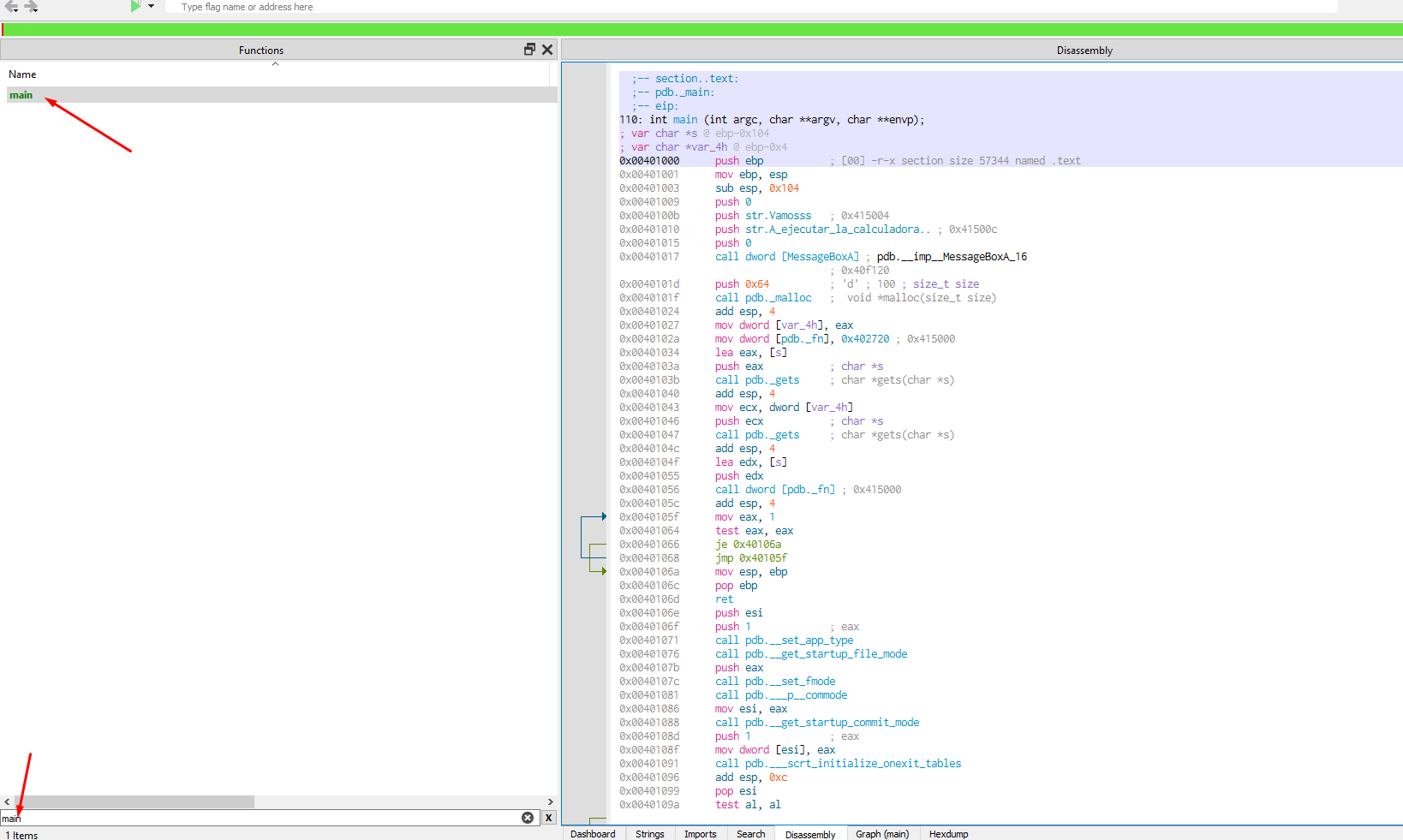


REVERSING DEL ABO4 USANDO CUTTER

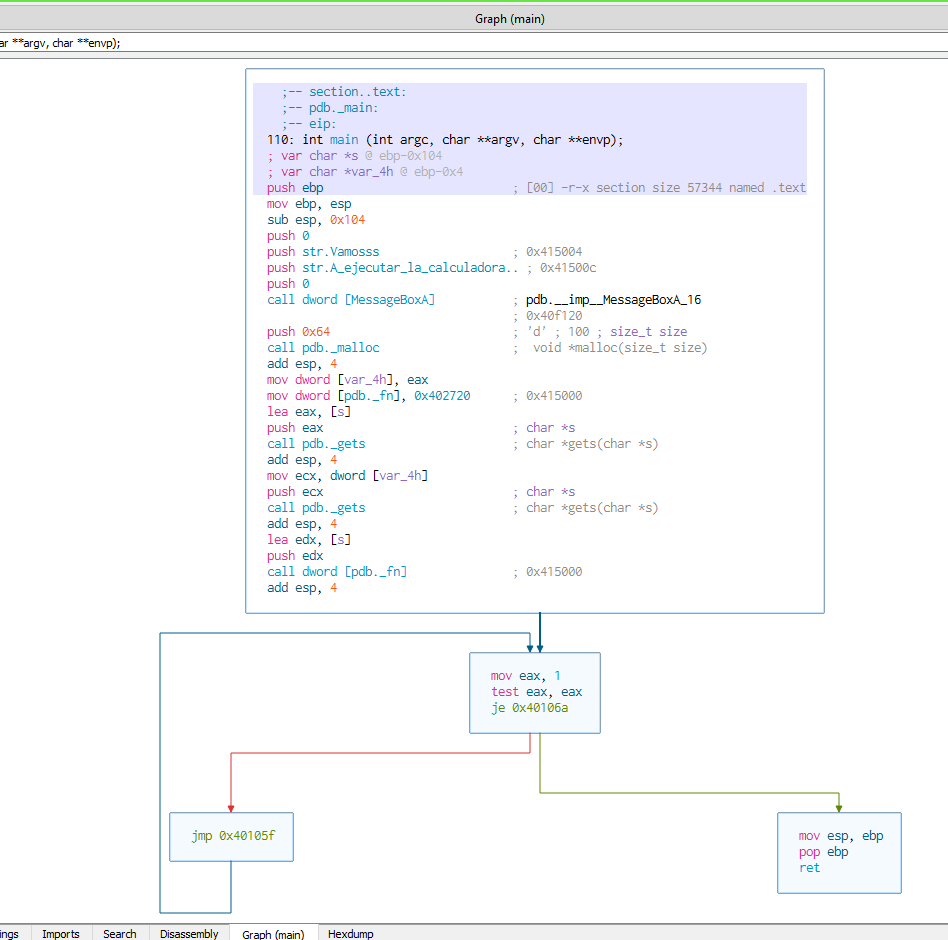
Se puede analizar fácilmente también usando Cutter.

Recordamos en las opciones avanzadas apuntar al pdb con los símbolos.

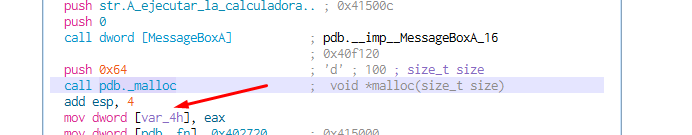




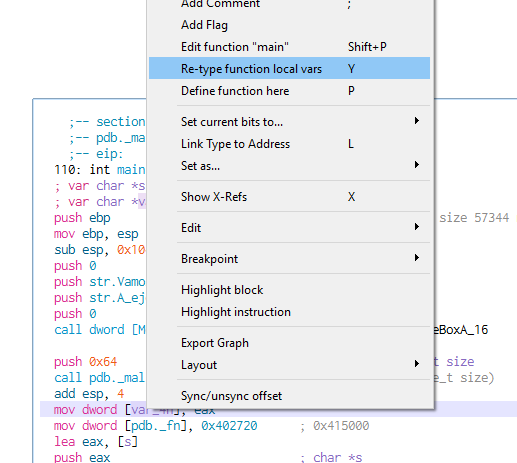
Allí busco el **main** y voy al mismo, si apreto la barra veo los bloques.

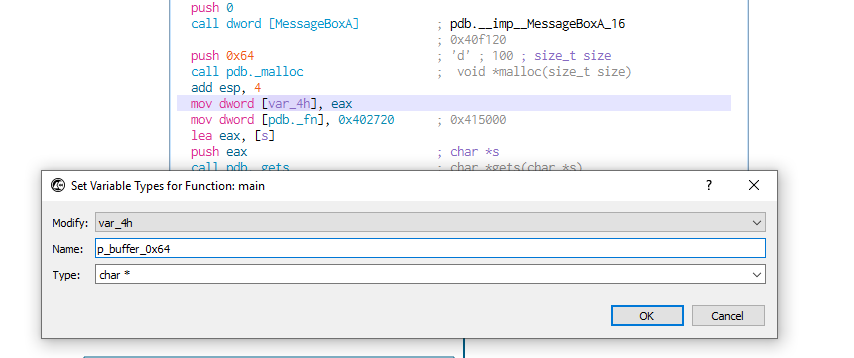


Vemos el loop infinito que impide llegar al retn y vemos los dos gets.



Vemos el **malloc** con un size de 0x64 y lo guarda en la variable **var\_4h,** la renombramos igual que en radare.

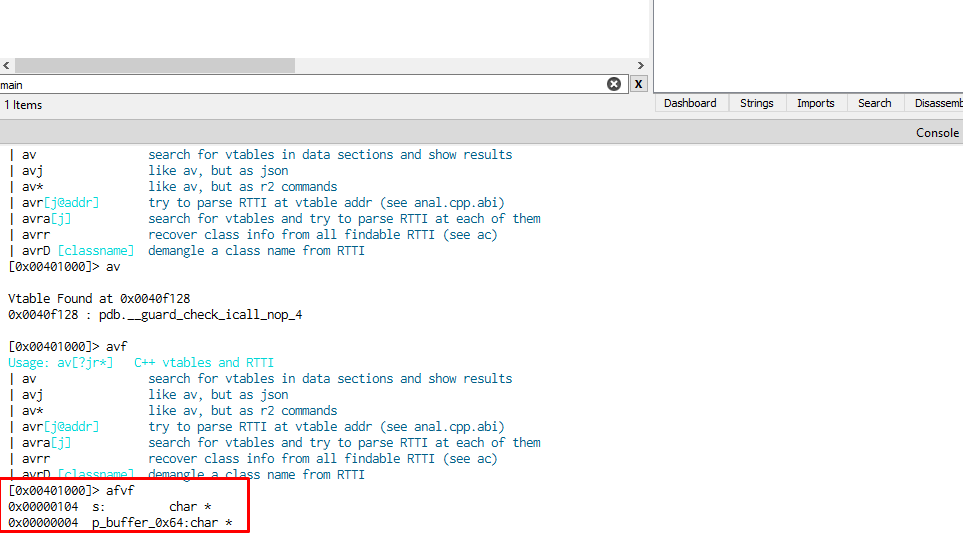




Alli quedó renombrada.



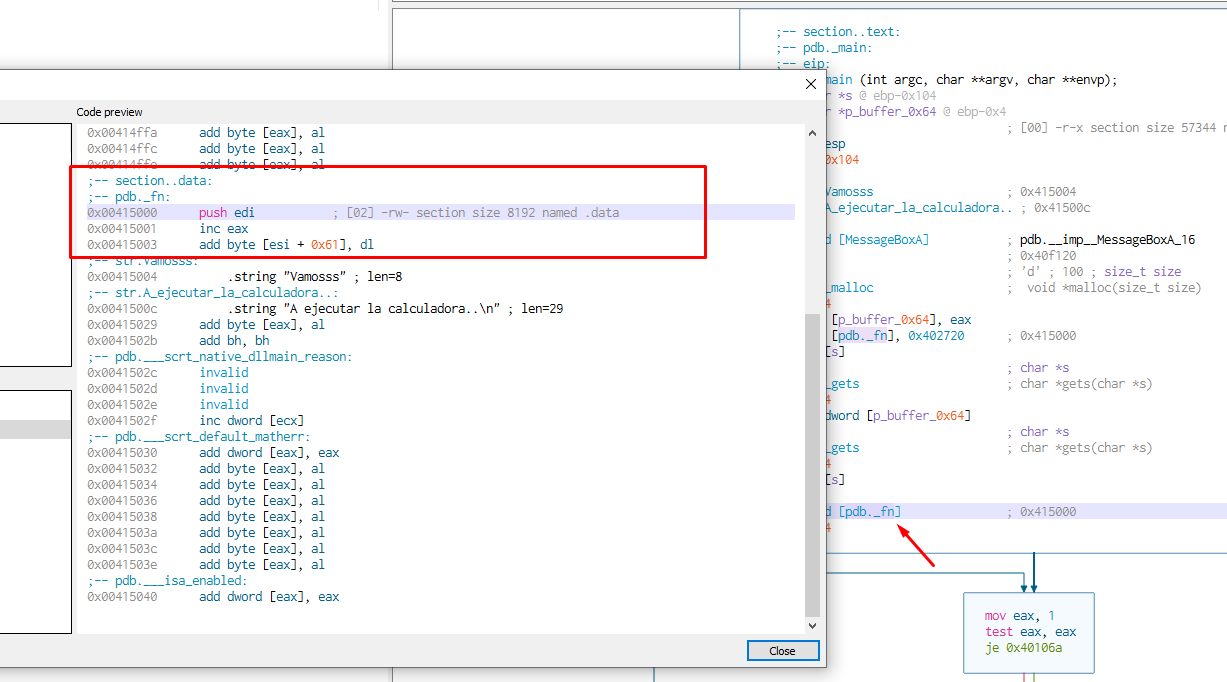
Veamos el tamaño del buffer **s**, activamos la consola de radare con WINDOWS-CONSOLE y alli tipeamos **afvf**



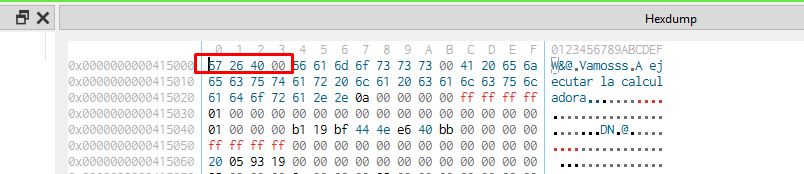
Y vemos que la resta de los dos da 0x100 que es el largo de **s**.

Obviamente como **p\_buffer\_0x64** está debajo del buffer **s** podremos pisar su valor al desbordar en el primer gets que llena **s** con 0x100 bytes, el resto desbordara y pisará lo que hay debajo.

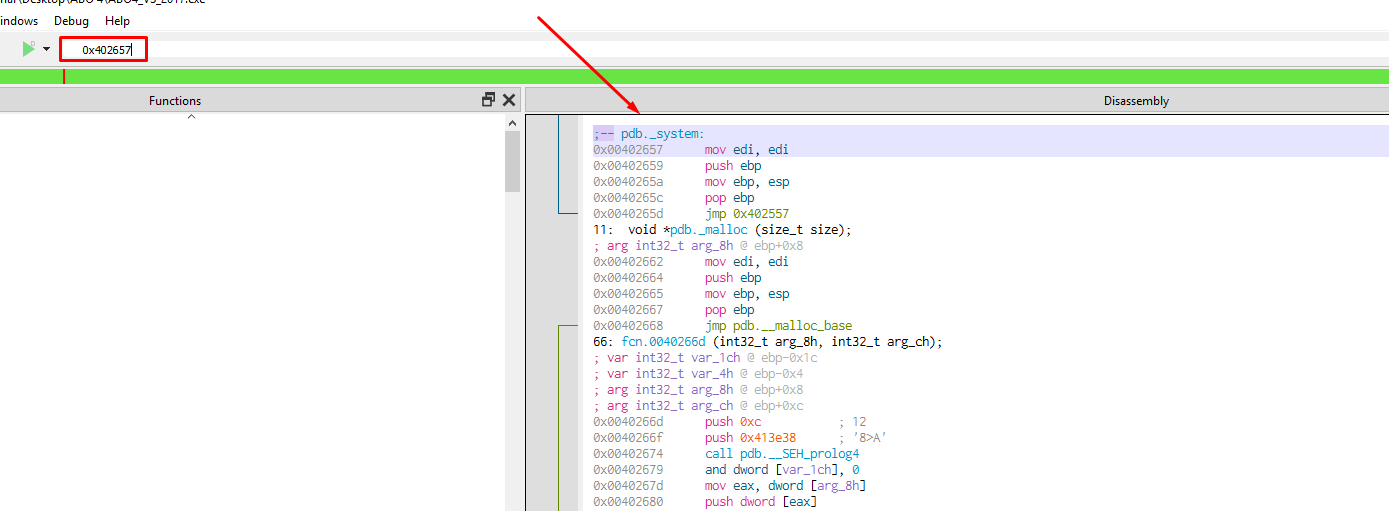
Si apretamos x en el nombre de la variable \_fn vemos que se encuentra en la sección data.



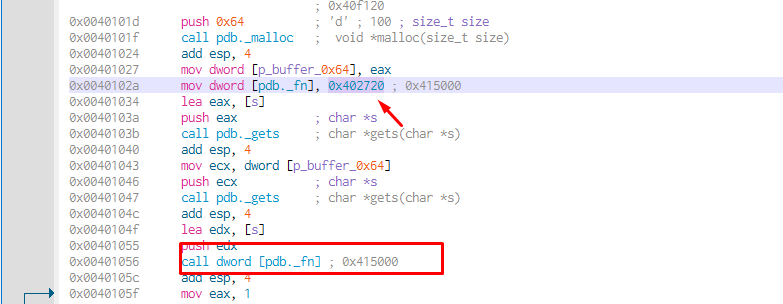
Vemos el valor inicial de la variable **\_fn** que es 0x402657.



Si tipeamos en la barra 0x402657 vemos que nos lleva a **system.**

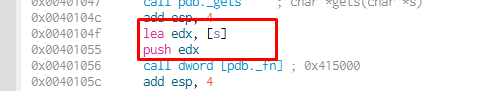


Así que **\_fn** tiene guardada la dirección de system al inicio.



Luego se pisa con **puts** y al final se salta a ejecutar el valor que contiene **\_fn**, que si no hubiera overflow seria saltar a ejecutar **puts**.

Así que como antes la idea es en el primer **gets** desbordar **s** y pisar del valor de la variable **p\_buffer\_0x64** con la dirección de **\_fn** que es **0x415000** y este valor se pasará como argumento al segundo **gets** y escribirá allí lo que queremos o sea la dirección de system nuevamente. Saltamos a **system** en el call y como controlamos el argumento, que lo toma del inicio del mismo buffer **s**, podemos poner la string que queremos en el inicio del mismo, con un cero final de string y la ejecutará si existe dicho ejecutable.



Allí vemos el argumento estará en el buffer **s**.

Y por supuesto ejecutará el notepad o lo que pongamos como string al inicio del buffer, si existe el ejecutable.

Bueno con eso hemos resuelto el ABO4 y terminamos con los abos y los stacks, la parte siguiente comenzaremos con un nivel más avanzado de ejercicios.

REPASEN

Hasta la parte 10

Ricardo Narvaja

07/02/2020