EXPLOITING Y REVERSING USANDO HERRAMIENTAS GRATUITAS - PARTE 15

Como vimos en la parte 14 este rop no será tan sencillo como los que hemos visto anteriormente.

# EJERCICIO DE REVERSING 2 - 32 BITS CON RADARE

Abrimos el proyecto que habíamos guardado la explicación de cada comando está en la parte 14.

ABRIENDO PROYECTO GUARDADO (PO)

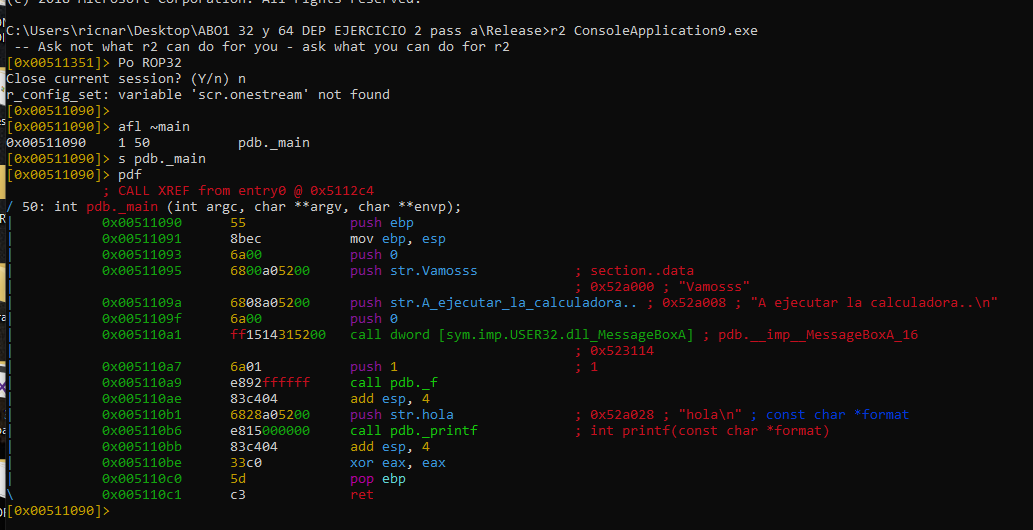
**r2 ConsoleApplication9.exe**

**Po ROP32**

**afl ~main**

**s pdb.\_main**

**pdf**

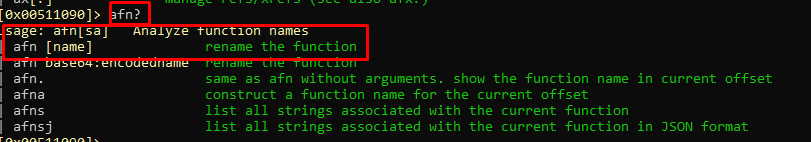


RENOMBRANDO FUNCIONES (AFN)

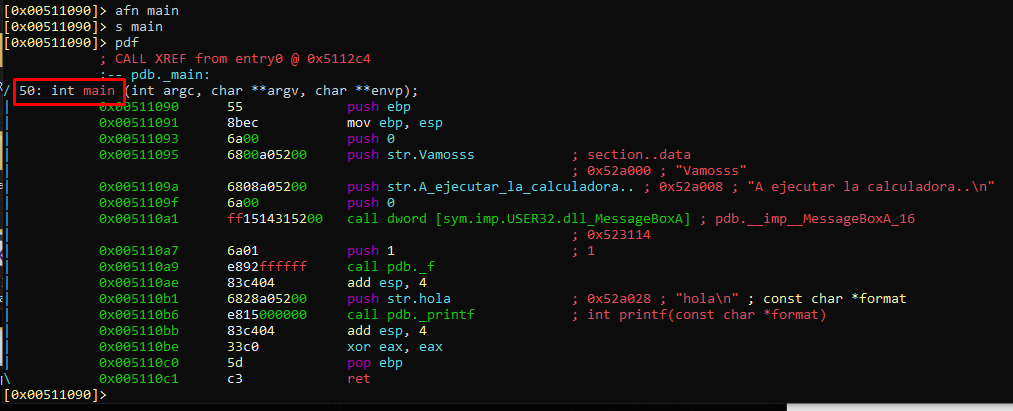
Ya estamos en el main nuevamente voy a renombrar **pdb.\_main** y **pdb.\_f** cp,p **main** y **f** así es más fácil de acceder.

DUDAS SOBRE UN COMANDO (?)

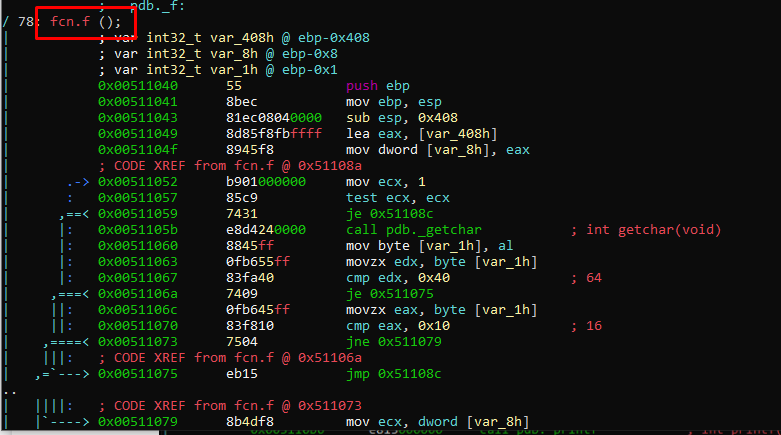
Cualquier duda sobre un comando poniendo **?** a continuación del mismo tendremos la ayuda



Con afn renombrare la funcion en la cuál estoy como estoy en pdb.main con **afn main** quedará renombrada.

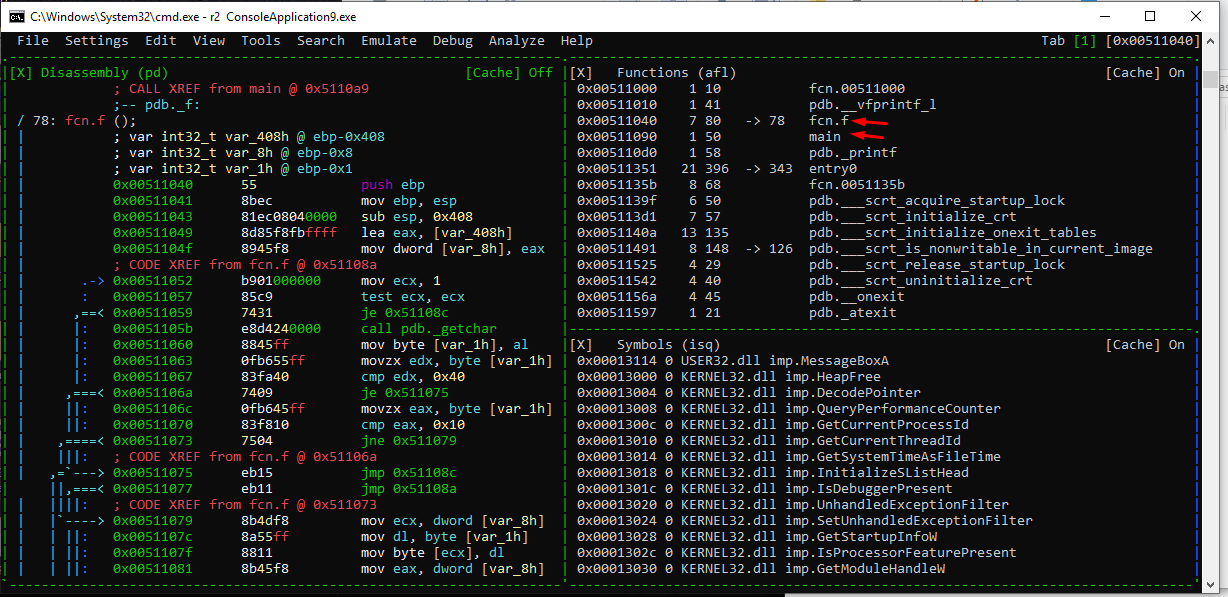


Luego ire a pdb.\_f con **s pdb.\_f** y la renombrare con **afn f.**

****

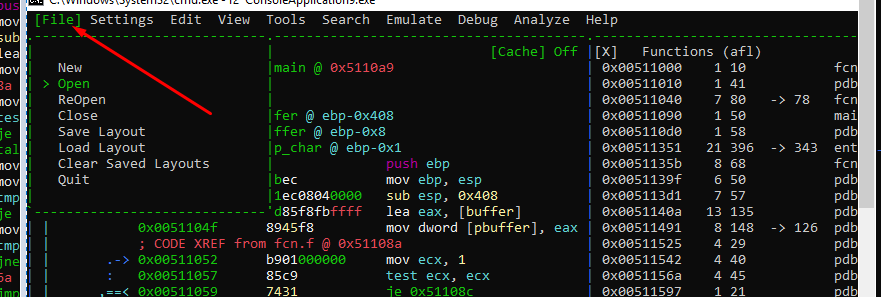
Vemos que lo renombra como **fcn.f.**

Entramos en modo visual con **v**

****

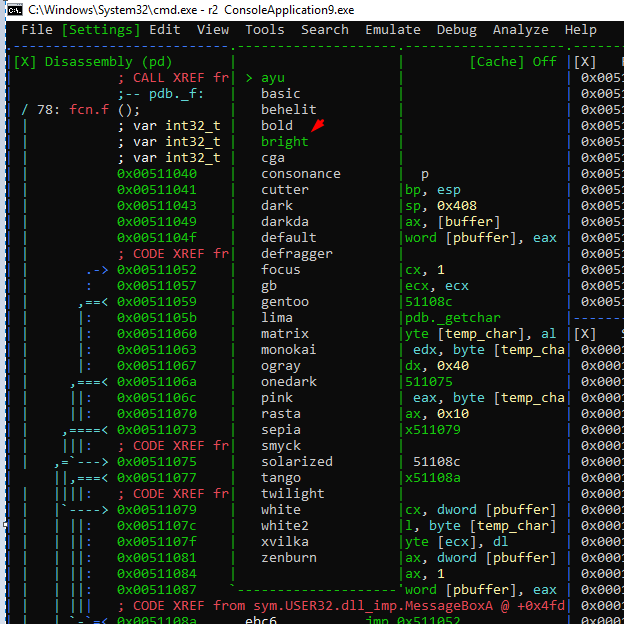
Y vemos las funciones renombradas salvamos presionando la tecla : y luego **Ps ROP32**

NAVEGANDO MENUS (M)

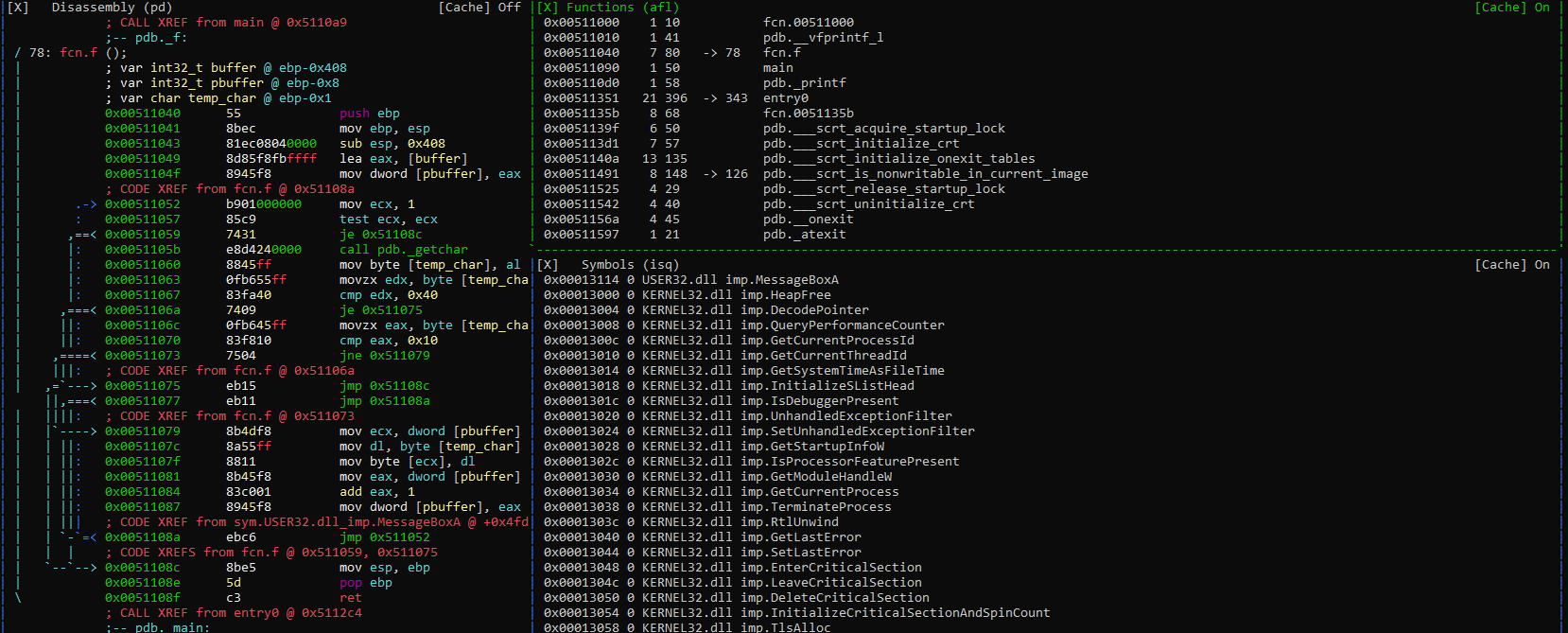
****

Vemos que con la tecla M y las flechas de dirección, podemos navegar por los menús.

En Colors está marcado el que yo elegí pero hay muchos más.



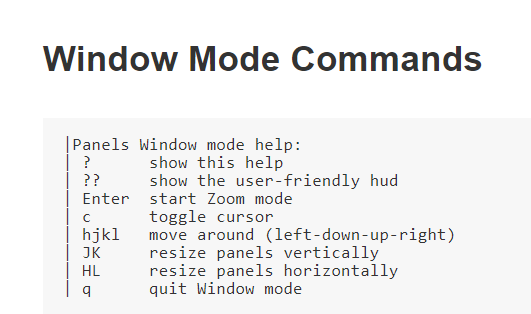
Podemos configurar la vista en VISUAL MODE de qué paneles queremos que se vean y de que tamanio por ejemplo:

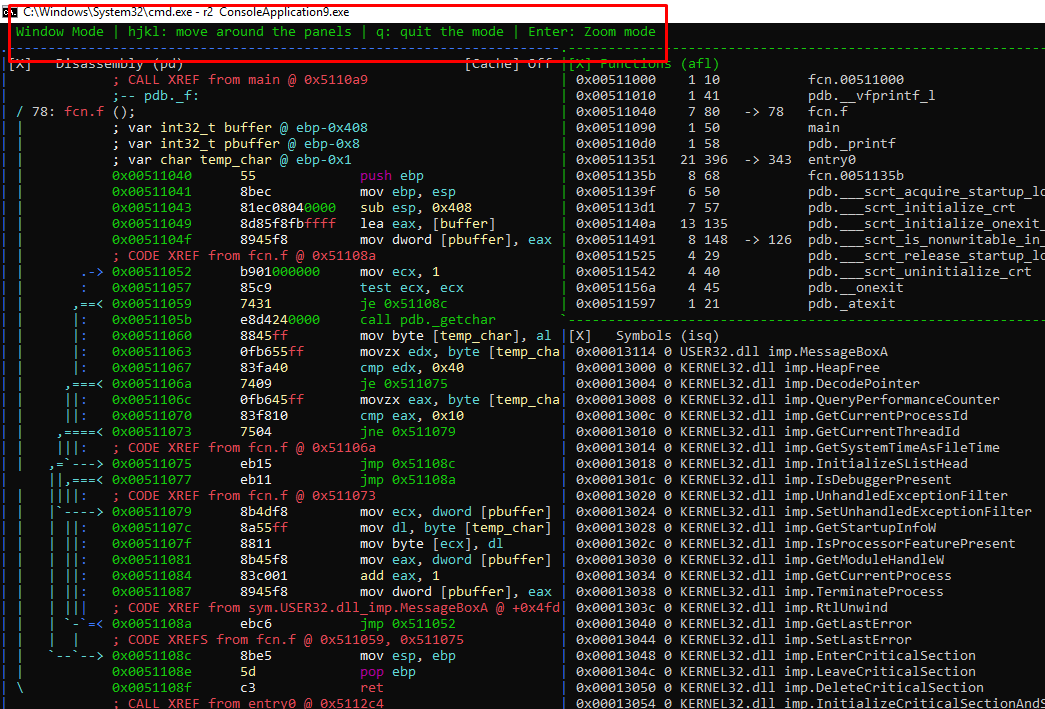


Veo que en mi caso los paneles de Funciones y Símbolos son demasiado grandes, podemos agregar y quitar paneles agrandar los que queremos y guardar el LAYOUT.

Para agrandar por ejemplo el panel de desensamblado entramos con la tecla w al modo window.

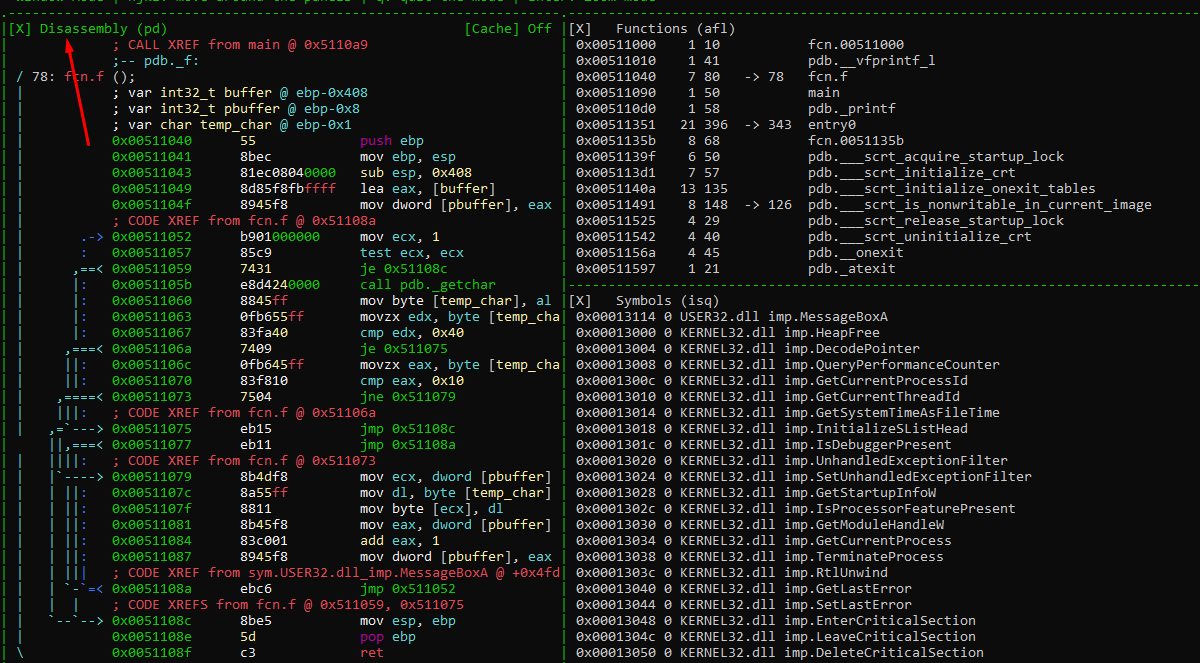
WINDOW MODE(w)



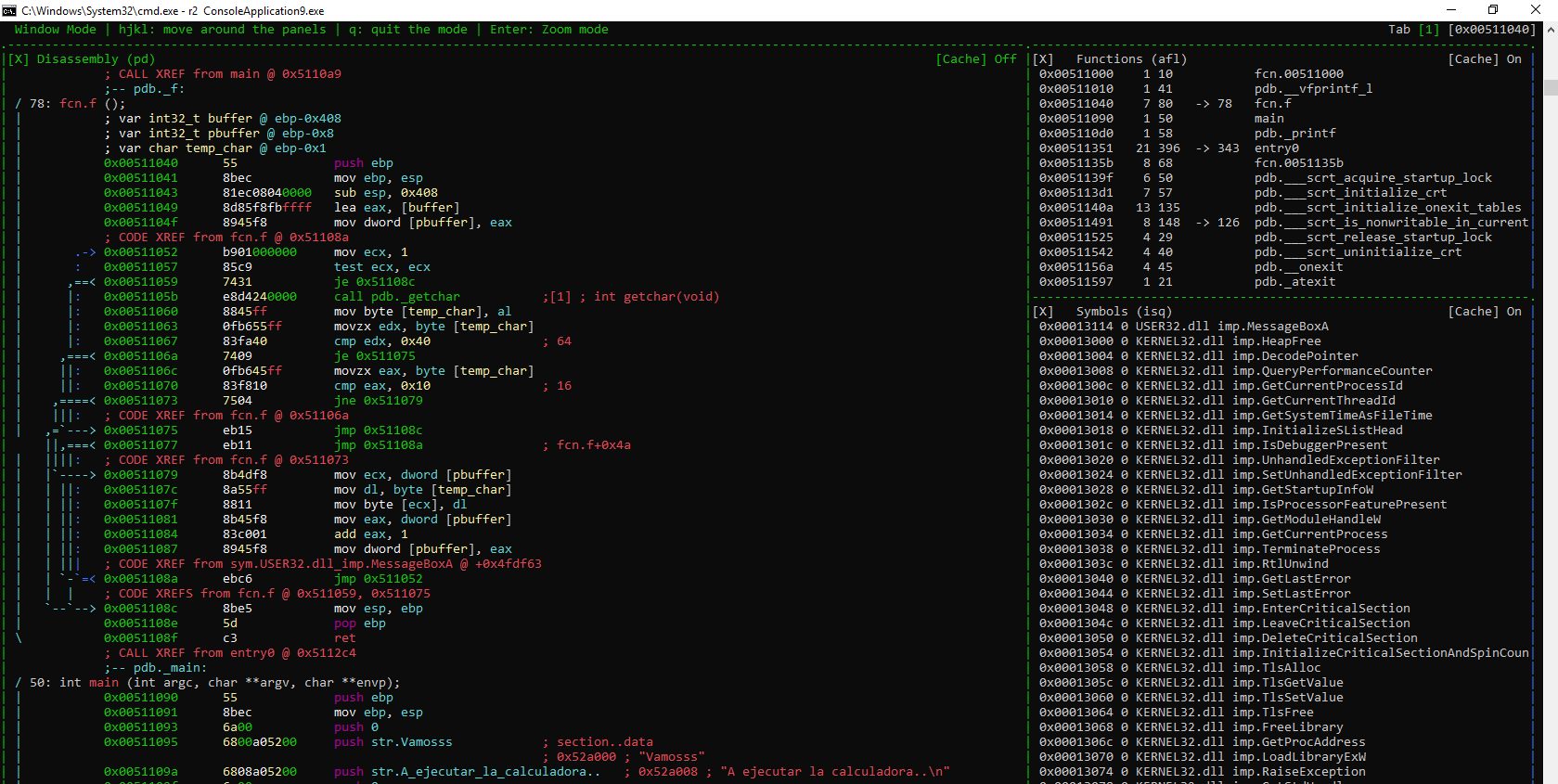


Nos avisa arriba que estamos en MODO WINDOW.

Con las flechas de resalto el panel el que quiero que este activo, y puedo cambiarlo de tamaño.

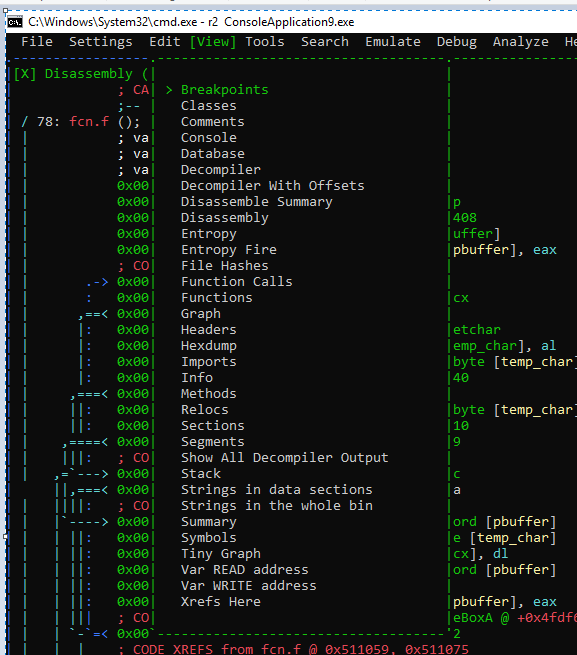


Luego con la combinación de SHIFT + H, J, K o L puedo agrandar o achicar, en mi caso lo quiero agrandar hacia la derecha así que lo haré con SHIFT + L repetidas veces.

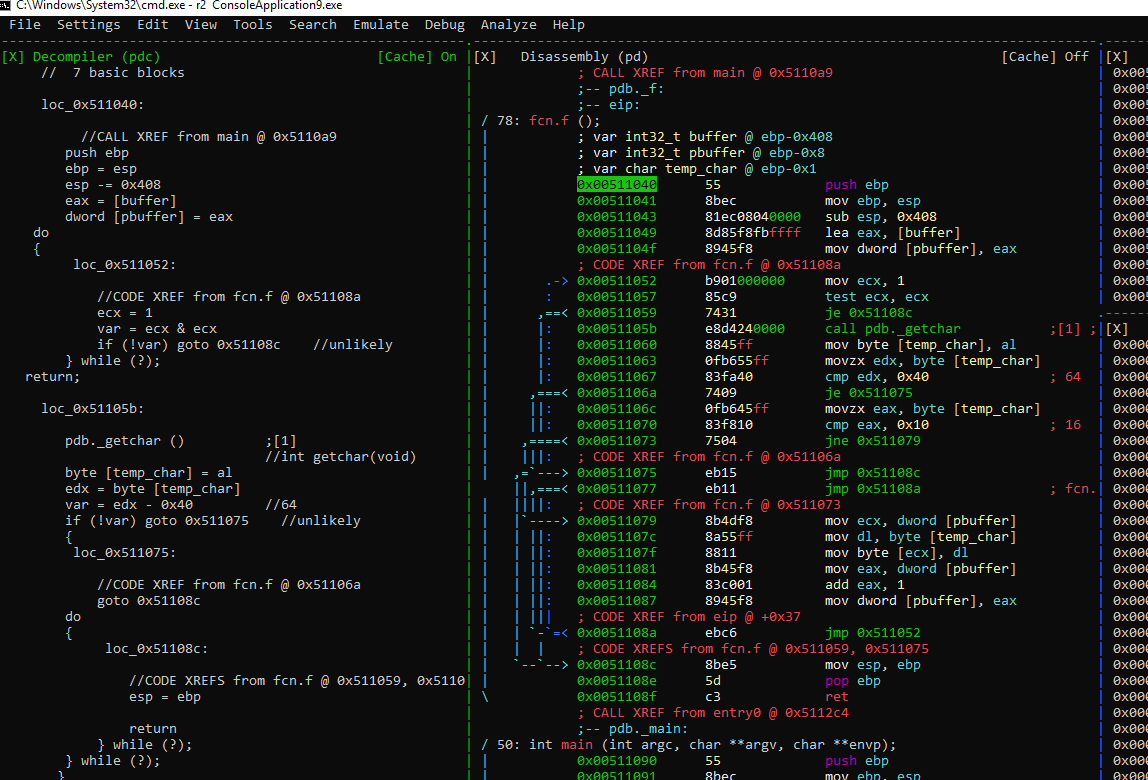


Ahí quedó mejor, puedo ver que ventanas puedo agregar.

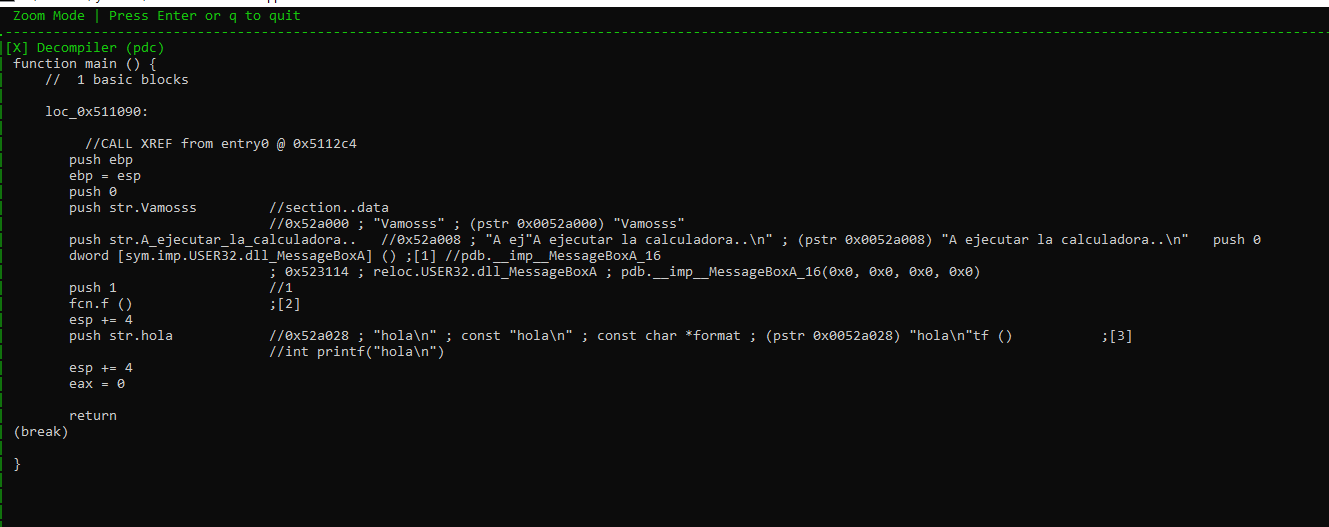
Salgo con w del MODO WINDOW y entro al menu con M y voy a VIEW.



Puedo agregar una ventana de decompilado

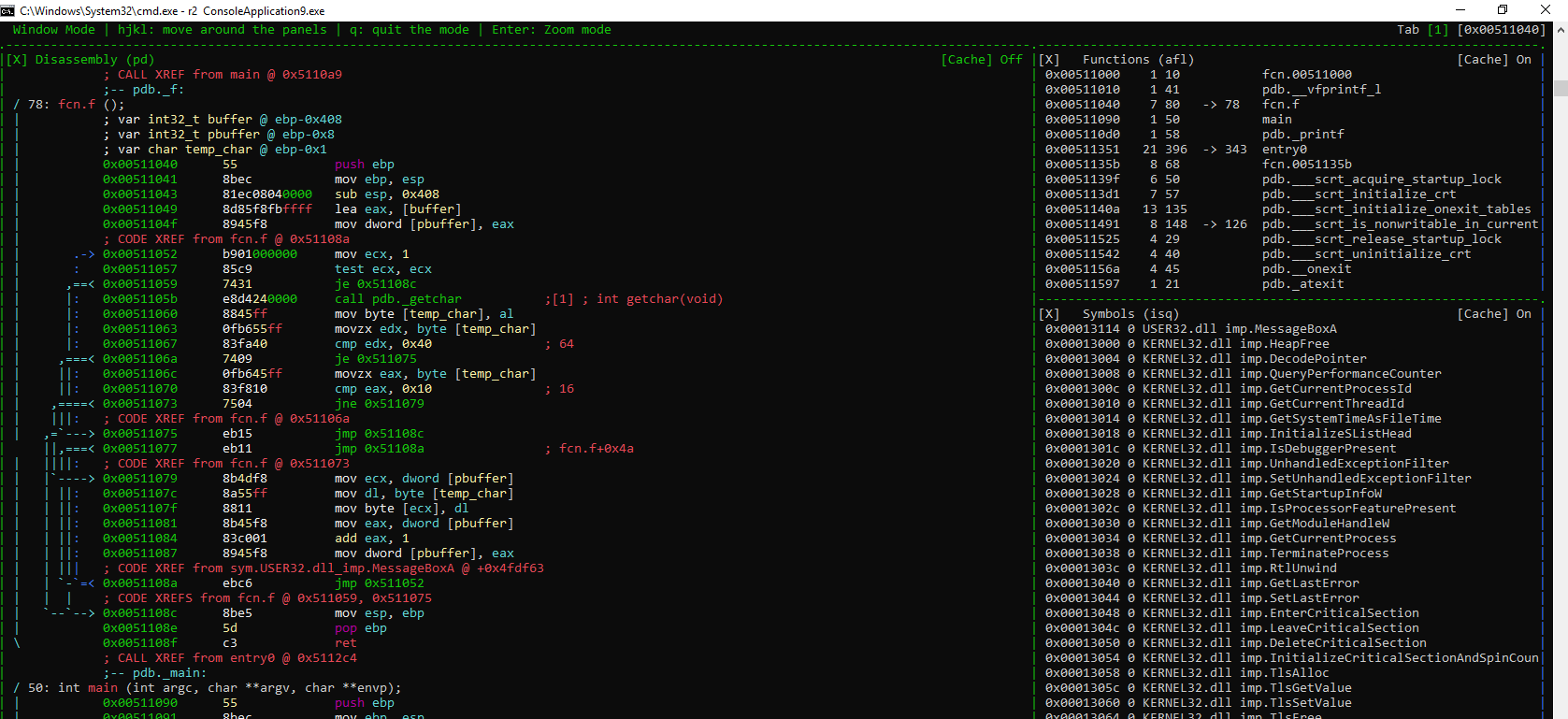


Puedo poner en foco este panel solo con ENTER



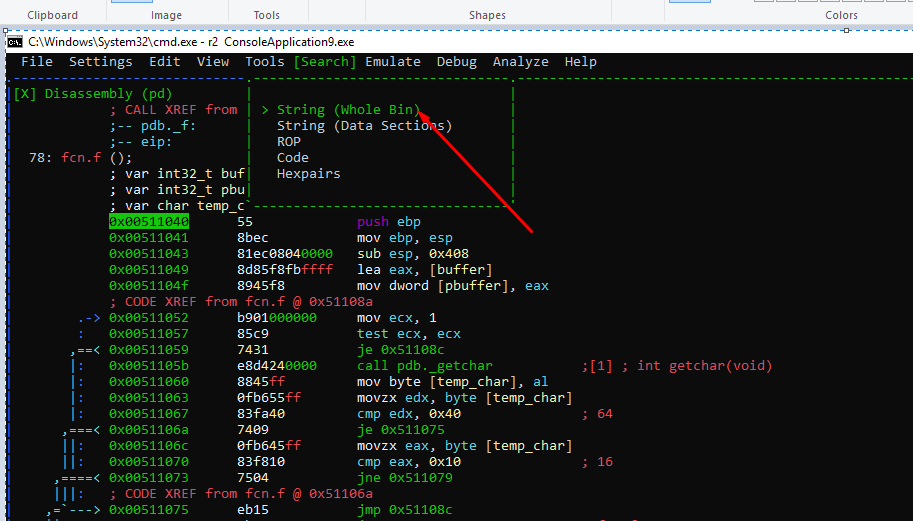
Si tengo en vista en el listado el main en el decompiler se verá el main también.

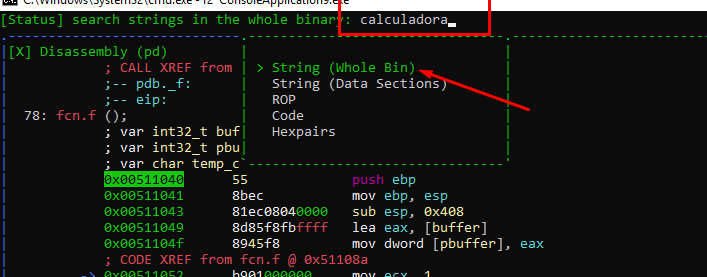
Si lo quiero quitar al panel entro al MODO WINDOW con W y luego SHIFT +X.



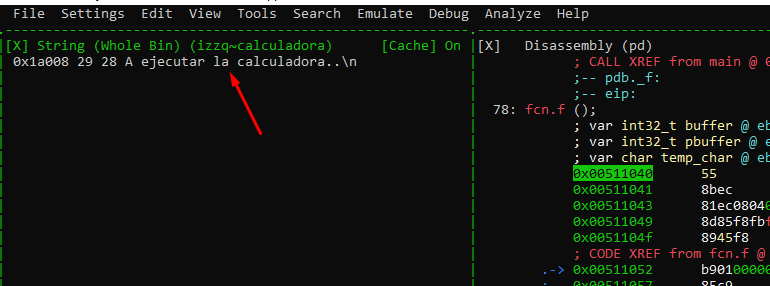
BUSCANDO

Si volvemos al menu vemos un SEARCH de strings





Veamos que halla.

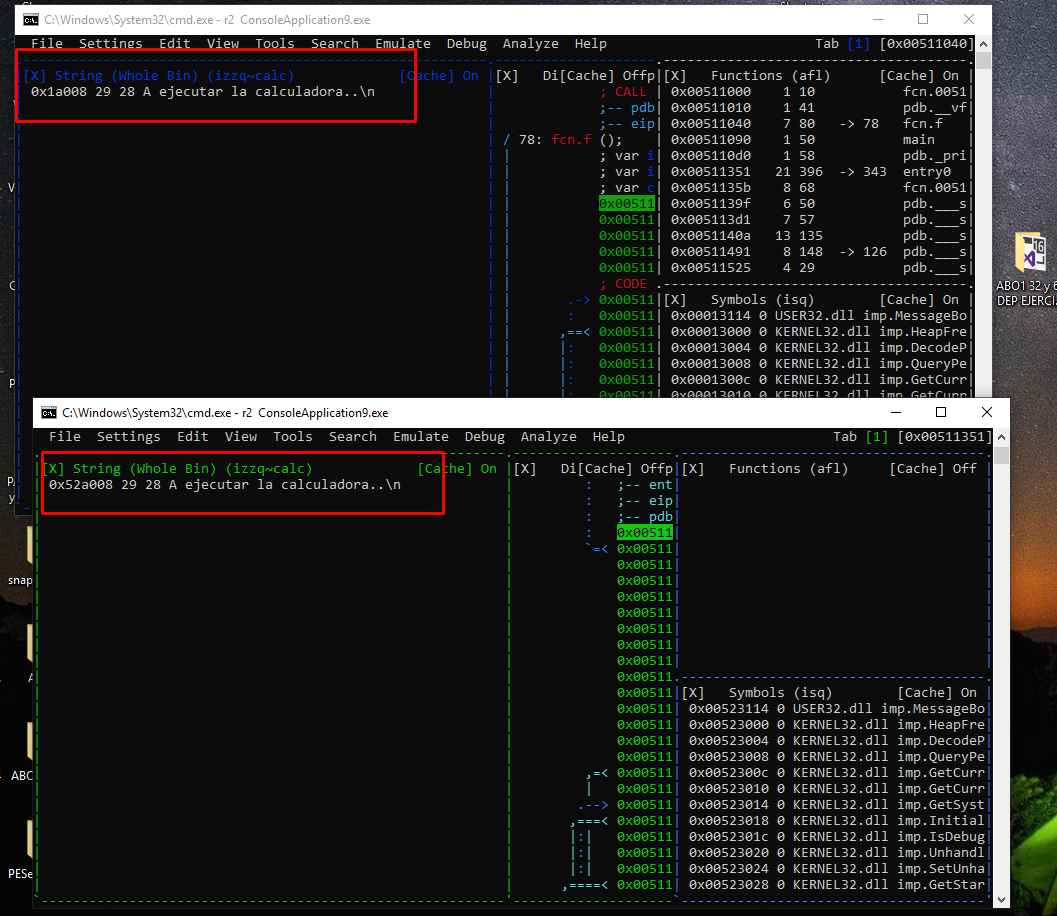


Aca encontre encontre un bug que se puede sobrellevar pero hay cosas cuando reabris un proyecto que no quedan igual que si lo ejecutas la primera vez, vemos la dirección de la string a la cual hay que sumarle la imagebase para llegar a la dirección real.

Python>hex(0x510000+ 0x1a008)

'0x52a008'

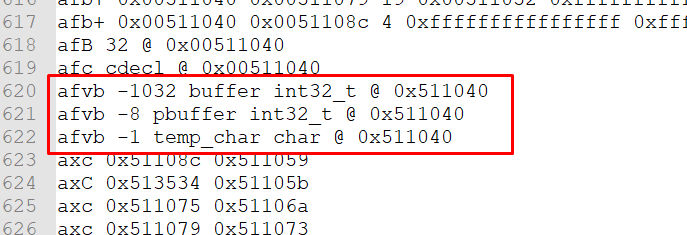
Mientras que si en otra consola lo abro sin cargar el proyecto cargo los símbolos, analizo y luego voy al mismo menu a buscar las strings.



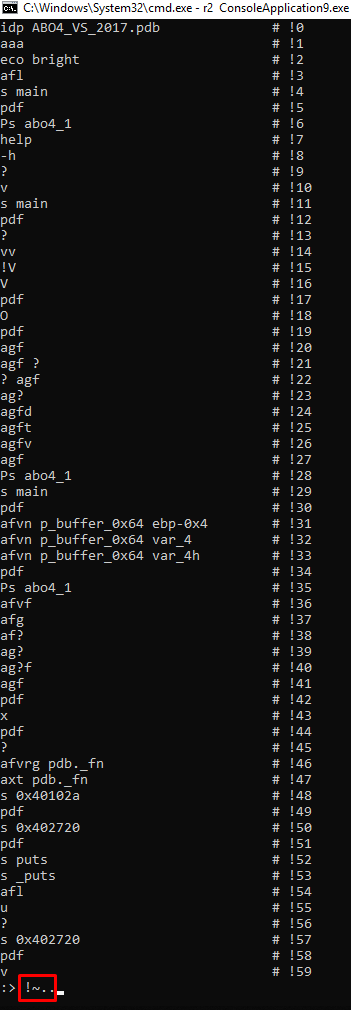
Alli vemos la comparación hasta que arreglen el tema de los proyectos a la búsqueda hay que sumarle la image base, mientras que abriendolo directamente y analizándolo no es necesario y funcionan bien las referencias y los tags.

Así que lo abro de nuevo, puedo copiar y pegar todos los comandos seguidos en una consola y los ejecuta todos seguidos en una sola vez así que si copio todos los comandos que ejecute (pero mirar en el archivo rc del proyecto alli están guardados)

C:\Users\**XXXXX**\.local\share\radare2\projects\\ROP32



También hay un comando para mirar la historia de los comandos que ejecute



Vamos que si copio los comandos importantes y los pego juntos en la consola, si los ubique en el mismo orden en que los fui ejecutando en su momento, los ejecuta todos uno detrás del otro y sin preguntar y todo queda igual, así que bueno hasta que no arreglen el tema de los proyectos habrá que guardar la lista de comandos que ejecutamos.

r2 ConsoleApplication9.exe

idp ConsoleApplication9.pdb

aaa

afvb -1032 buffer int32\_t @ 0x511040

afvb -8 pbuffer int32\_t @ 0x511040

afvb -1 temp\_char char @ 0x511040

s pdb.\_main

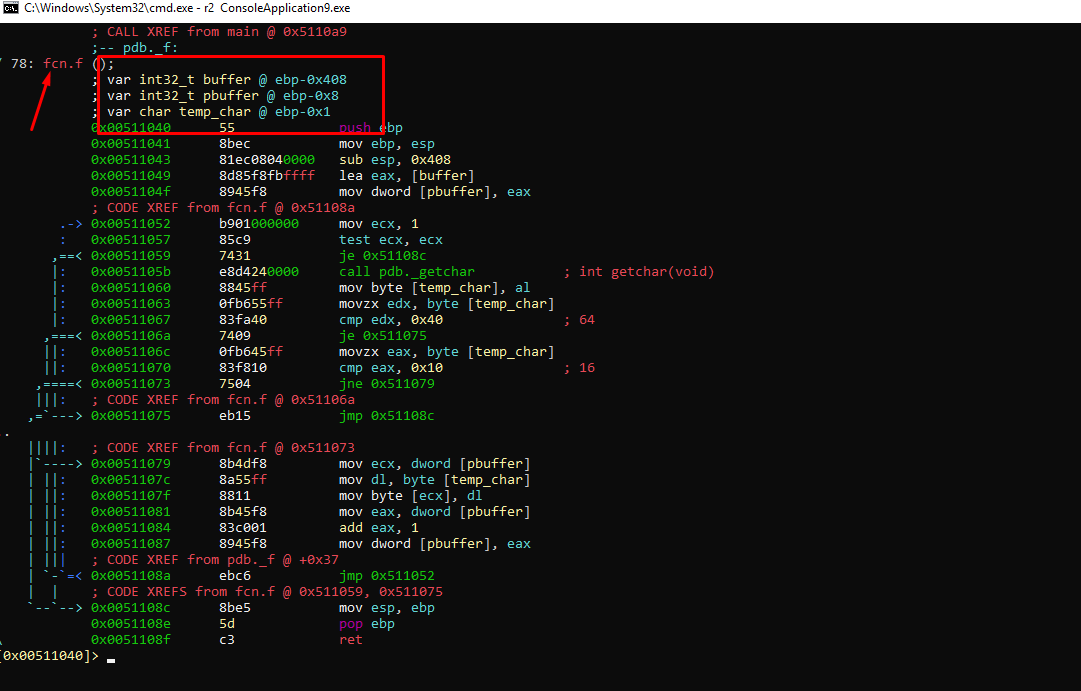
afn main

s pdb.\_f

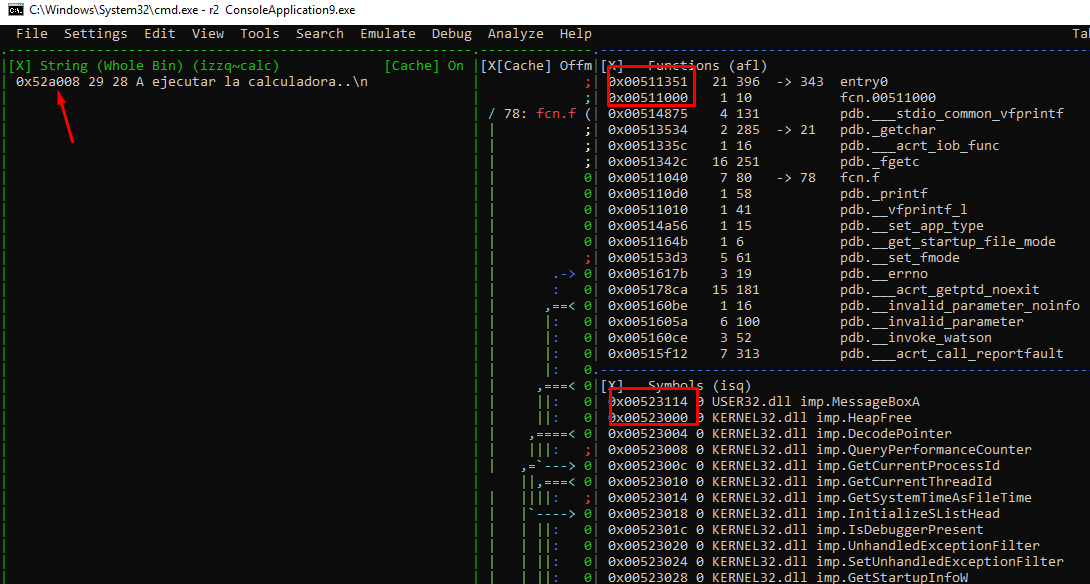
afn f

eco bright

pdf

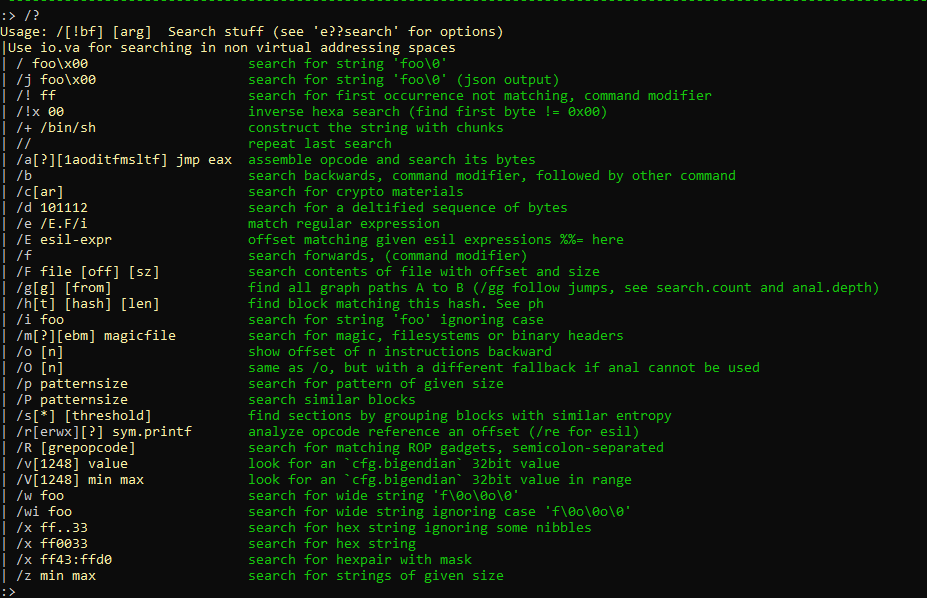


Vemos que quedó como estaba antes ahora puedo entrar con v.



Ahora todo coincide en el menú también tiene para buscar ROP GADGETS, CADENAS HEXA, etc.

Ahora buscando desde la consola hay muchas posibilidades.

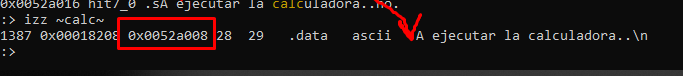


ENCONTRANDO REFERENCIAS

Vemos todo tipo de búsquedas aunque para buscar referencias el comando / no es tan rápido, si tipeo / calc lo encontrara.

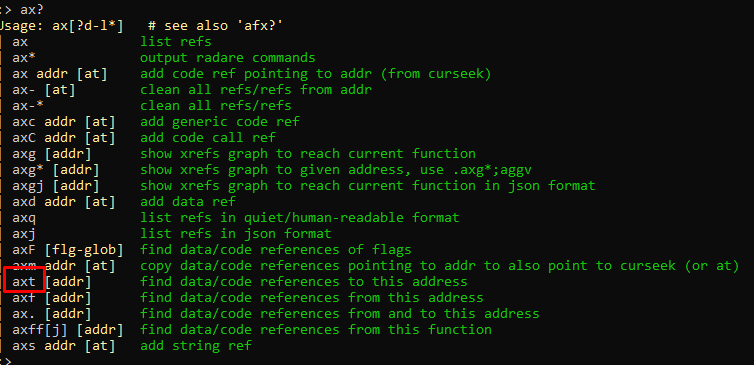
****

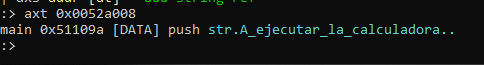
Vemos que muestra la dirección de la string calc que está entre medio de la string completa “A ejecutar la calculadora” el tema es que si busco referencia a calc no tendrá mientras que la dirección de inicio de toda la string si la tiene, usando este otro comando **izz**.



Vemos que aunque haya buscado calc, se fija la string completa y me devuelve la dirección de inicio de la misma 0x52a008, desde la cuál si puedo hallar fácil las referencias en el código con el comando axt.

Alli vemos todas las posibilidades de hallar referencias que tenemos.

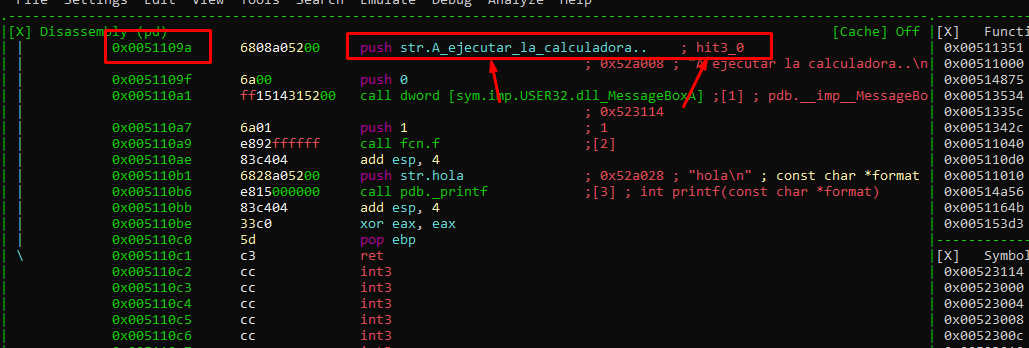




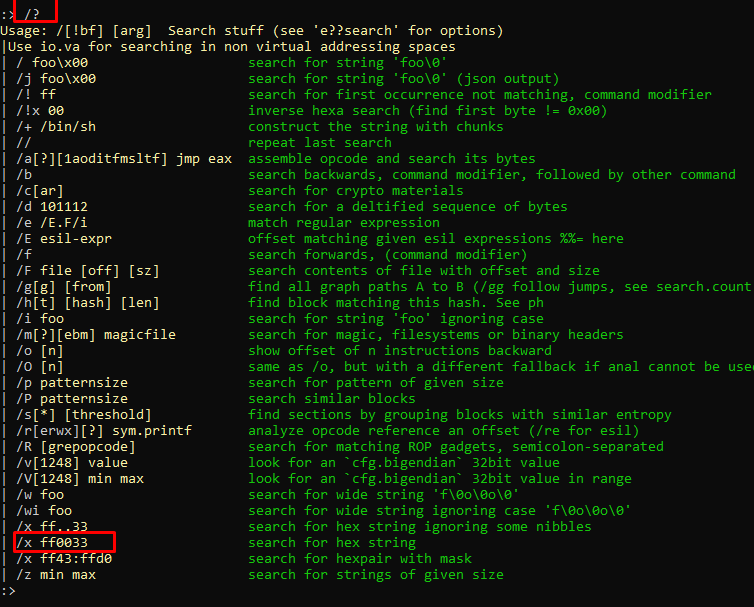
Alli vemos en main en la dirección 0x51109a donde utiliza la string, puedo verla en el listado yendo con v y estando resaltado el panel del listado apreto g y poniendo la dirección.

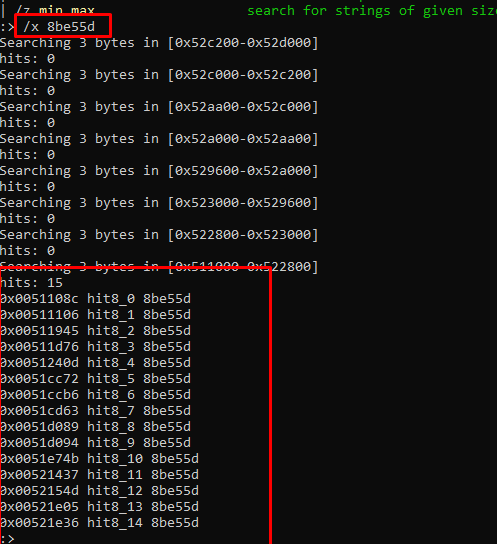


Y nos muestra la string en el listado.

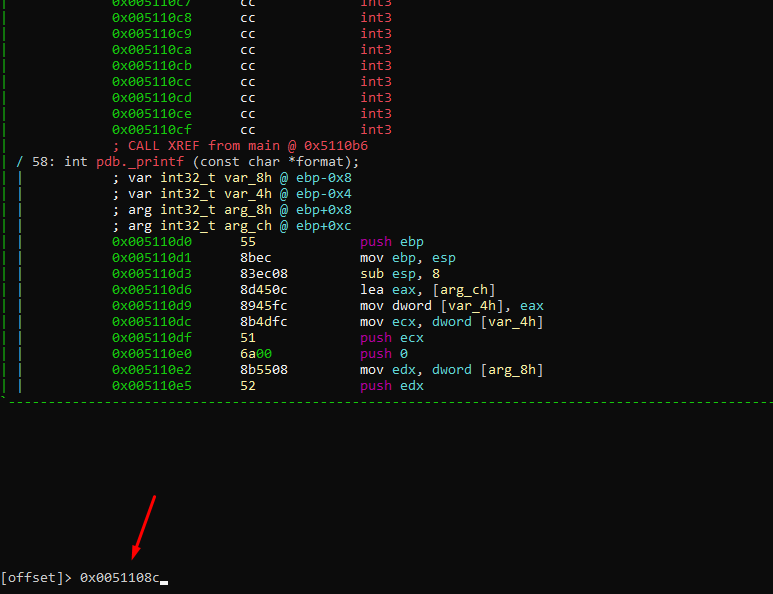


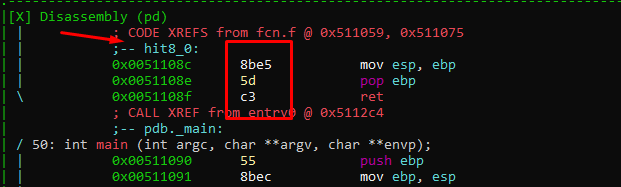
Si queremos buscar secuencias de bytes hexa apretamos : y luego.





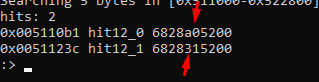
Acá no hay que buscar referencias el resultado está en el código, voy con v luego g y en offset tipeo la dirección a la que quiero ir



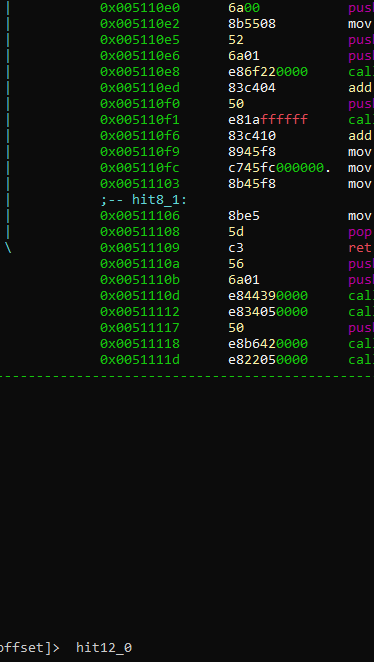


Se puede buscar usando puntitos como comodines.





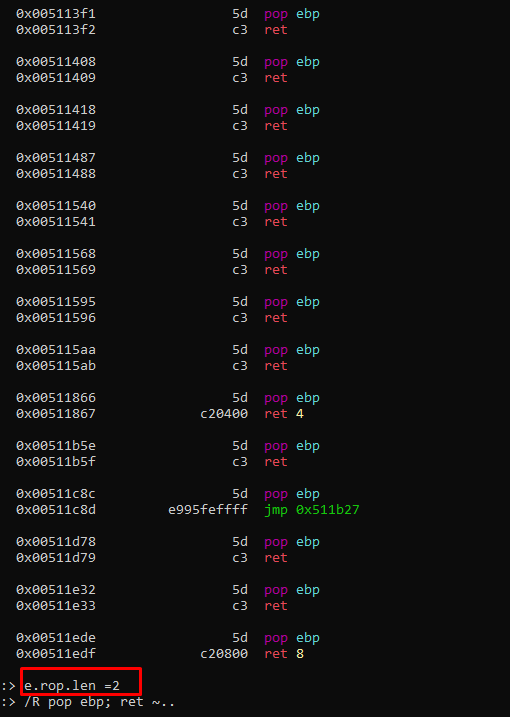
Vemos que puedo ir a la secuencia con la dirección o con el tag





BUSCANDO GADGETS.

Puedo buscar gadgets para crear un ROP.



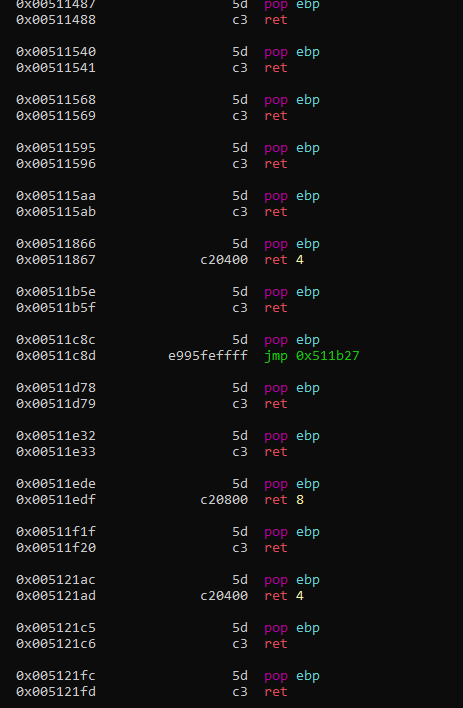
Seteo el largo del gadget en mi caso 2.

**e.rop.len**

Y luego con el comando

**/R comando1 ; comando2; etc ~..**

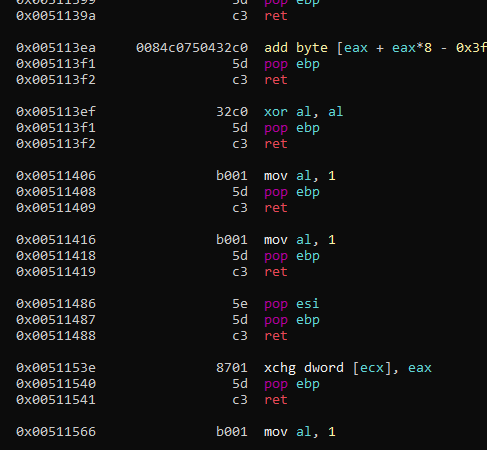
El modificador **(~..)** al final de cualquier listado hace que se pueda scrollear con la tecla enter y pg up, pg down evitando largos listados ilegibles.



Puedo seguir listando hacia abajo con ENTER o PG DOWN.

Cambiando el largo a 3

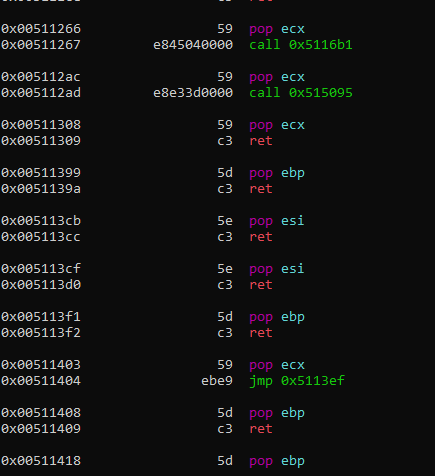
**e.rop.len =3**

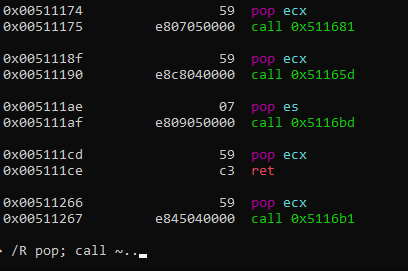


Se puede usar regexp para buscar gadgets con el comando **/R/**

**/R/ pop e..; ret ~..**

Buscará todos los gadgets con pops que comiencen con e.

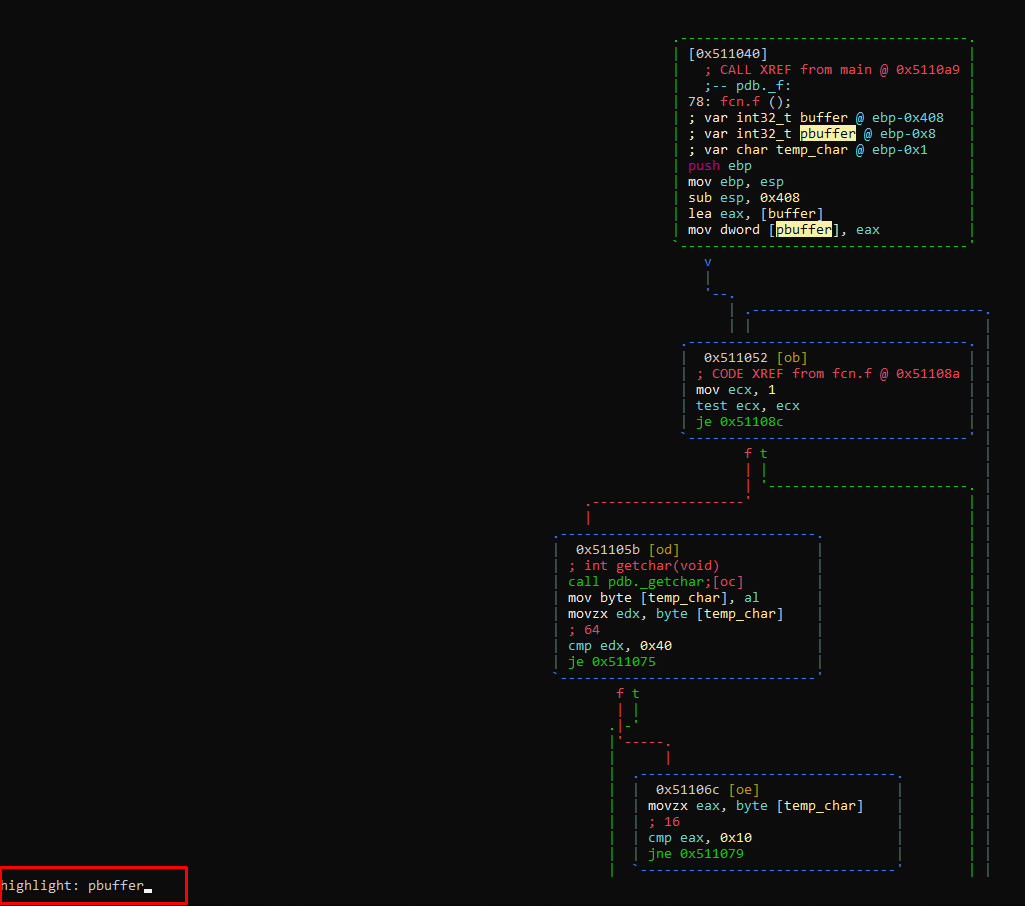




Se pueden hallar y buscar múltiples combinaciones que por supuesto usaremos para solucionar el rop de este ejercicio.

Ya tenemos una idea de como buscaremos los gadgets, ahora tratemos de analizar para crear el script.

Alli vemos en modo visual apretando la tecla / y tipeando el nombre de la variable pbuffer, la misma es resaltada para ver donde se usa.



También en modo visual cuando estoy en modo gráfico apretando la tecla p repetidas veces puedo cambiar la forma de visualizacion, la que más me gusta es está.



Se ve más clara.

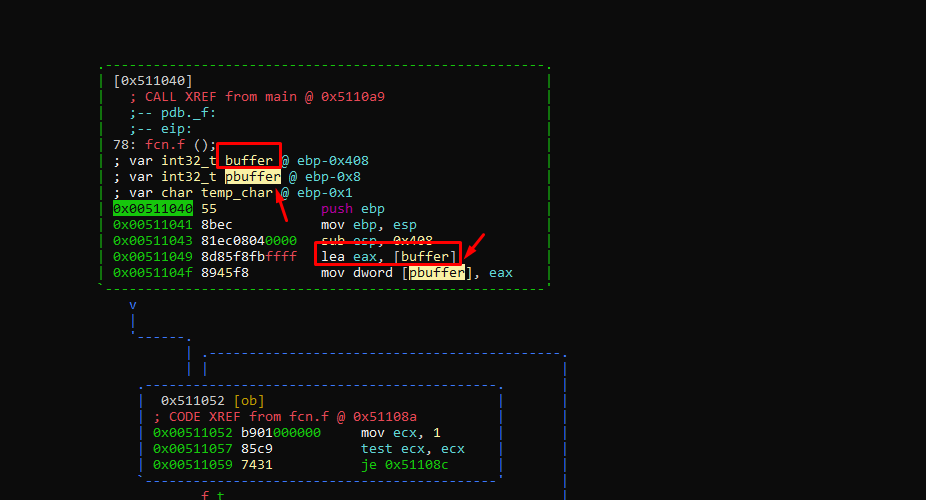
Hay otras visualizaciones si siguen apretando p, que pueden usar en algún momento.

Está muestra un grafico de bloques, marcando solo las instrucciones CALL.

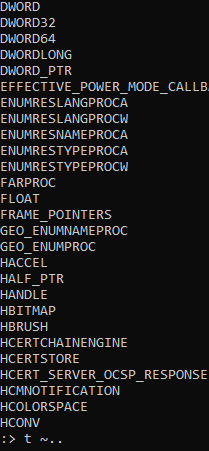


REVERSEANDO ESTATICAMENTE.

Como habiamos analizado debajo de buffer está la variable pbuffer, que guarda la dirección del buffer, es una variable del tipo puntero.



Podemos ver los tipos con el comando t le agregamos el ~.. Para poder scrollear por el resultado.



La variable pbuffer es del tipo char \* podemos cambiarlo con

**afvt pbuffer char \***

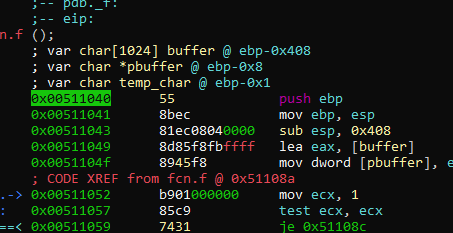
O si nos gusta más PCHAR

**afvt pbuffer PCHAR**

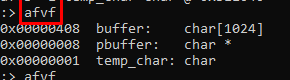
Como el largo de buffer es 0x400 ya que la resta de buffer y la variable siguiente pbuffer da esa distancia podemos renombrarlo como



**afvt buffer char[1024]**

****

Ahora quedó más similar a lo que que cada variable es realmente.



El comando afvf me muestra las posiciones estáticas del stack debajo sabemos que está el ebp guardado que tiene 4 bytes y el return address también 4 bytes. (le faltaria ponerle el menos delante de las variables se lo agrego)

-0x00000408 buffer: char[1024]

-0x00000008 pbuffer: char \*

-0x00000001 temp\_char: char

0 stored ebp

+4 return address

Por lo tanto vemos que la distancia desde buffer al return address, tenemos una variable buffer de 1024 bytes, a continuación una variable pbuffer de 4 bytes , 3 bytes vacios, 1 byte de temp\_char y 4 de stored ebp.

1024(buffer) + 4 (pbuffer) + 3 (vacio) +1(temp\_char) + 4 (stored ebp)= 1036



Eso sería lo que deberíamos llenar el buffer para quedar justo antes del return address o sea que nuestro script debería ser así.

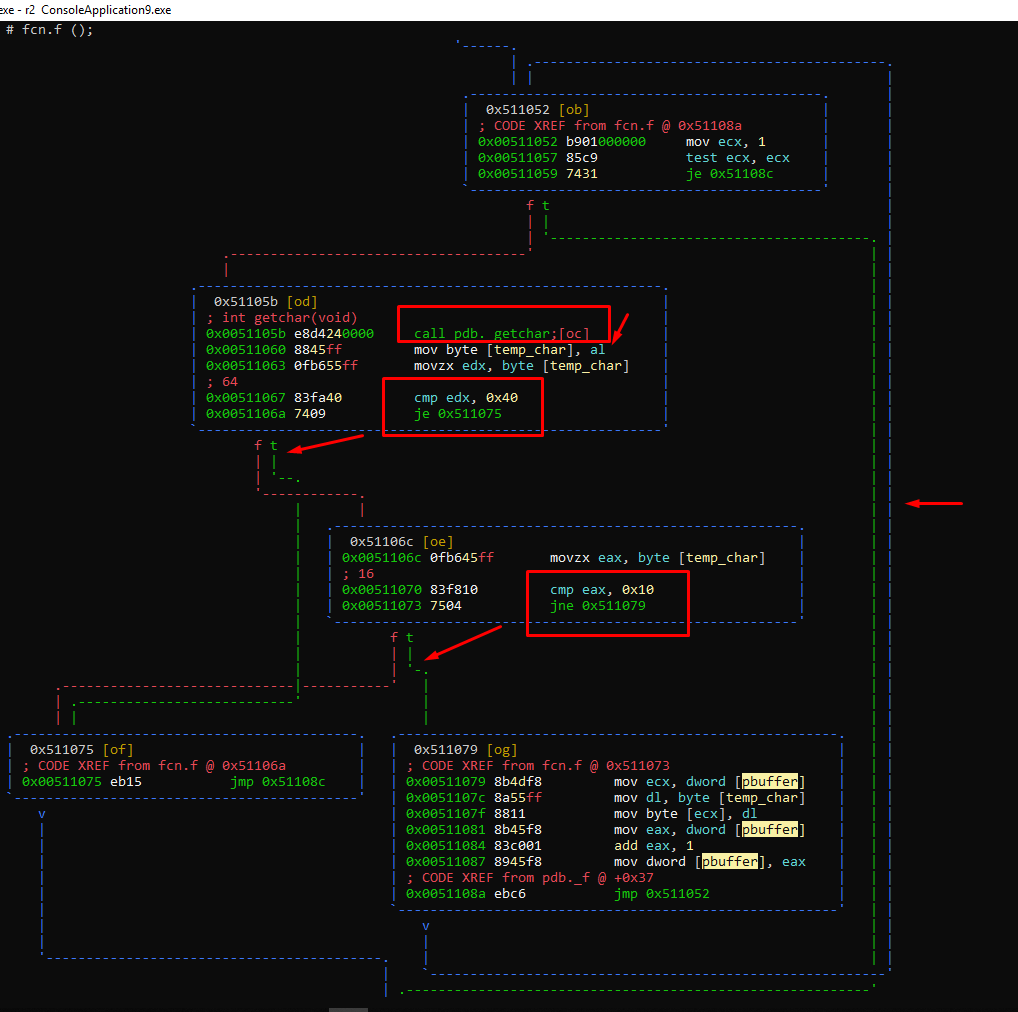


Pero analicemos un poco más, las preguntas son

1)Llegamos a pisar el return address con esto?

2)Llegamos el retn sin romper nada?

Bueno vemos que hay un loop infinito que tiene alguna forma de salir.



Dentro del mismo con getchar lee un caracter lo guarda en temp\_char, lo compara con 0x40 si es igual se va por la flecha verde de true y sale del loop si no es igual a 0x40, sigue y compara con 0x10 si no es igual sigue dentro del loop, lo que es lo mismo que decir, que si es igual a 0x10 sale.

Si sigue dentro del loop (no permite dentro del mismo que el caracter sea 0x40 ni 0x10 como vimos), lo guarda en el contenido de ECX qué es buffer, que ECX tiene la dirección pbuffer del mismo, y apunta al inicio del buffer.

Luego de guardarlo incrementa pbuffer, para en el siguiente ciclo guardar el siguiente caracter.

Parecería todo correcto, quiere decir que supuestamente si le agregamos un 0x40 después de pisar el RETURN ADDRESS saldra.



Hay que recordar que como estamos en Python 3 poner la b delante de cada cadena de bytes, en este caso seria b”\x40”

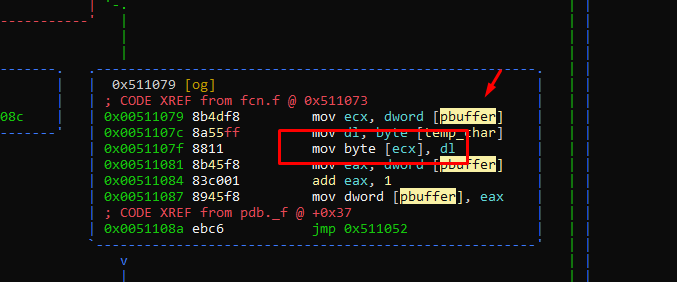
Podemos pisar el RETURN ADDRESS con 0x41424344, luego poner algunos bytes y luego salir con el 0x40.

Pero, pero… no todas son flores hay algunos problemas más.



Recordemos que debajo de buffer, está la variable pbuffer la cuál será pisada antes de pisar el return address.

Eso quiere decir que cuando si la pisara con 0x41414141 por ejemplo, en el siguiente ciclo trataría de escribir en [0x41414141]



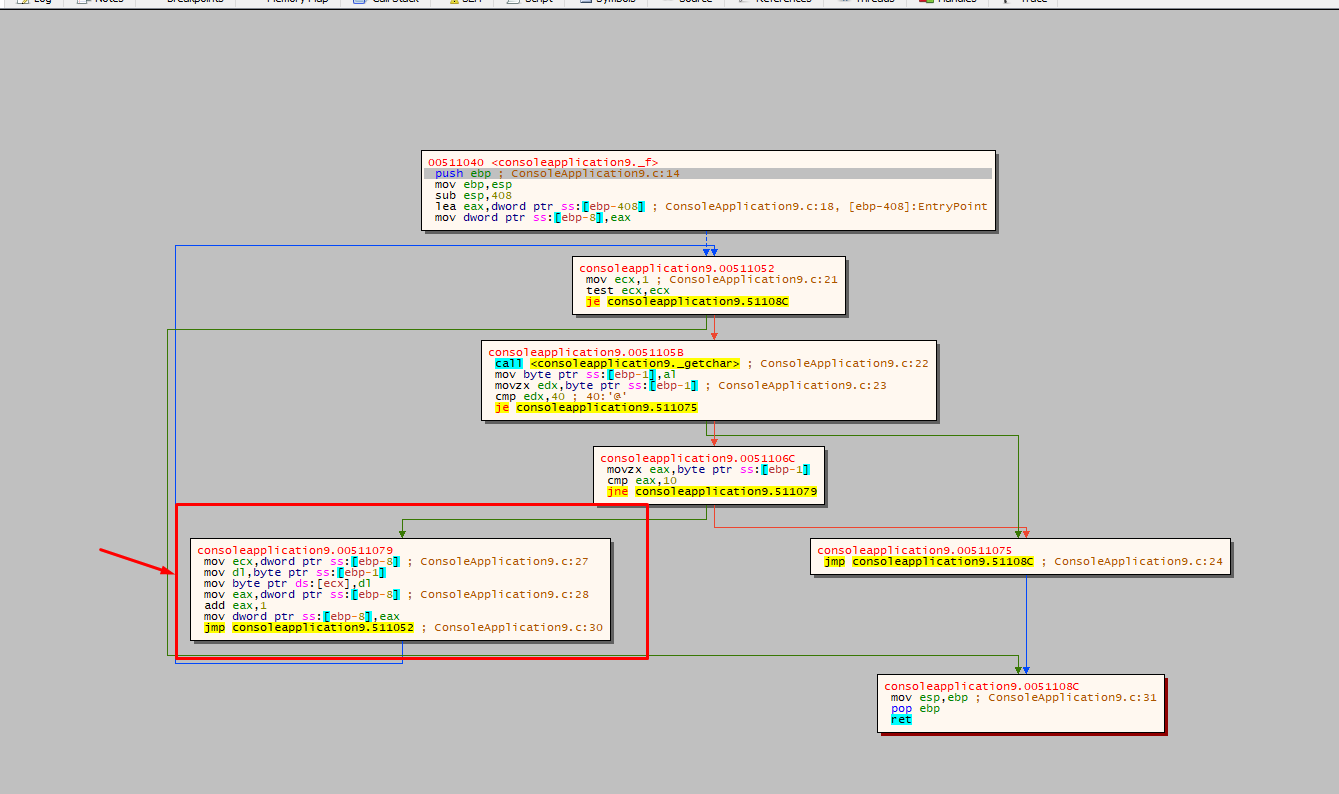
Si pbuffer llegara a pisarse con 0x4141414, escribiría alli con mov byte [ecx], dl y ya no podriamos pisar el return address.

La realidad es que pbuffer se pisa de a 1 byte así que podemos pisar el byte más bajo y hacer que escriba más adelante para que saltee escribir los otros tres bytes de la dirección.

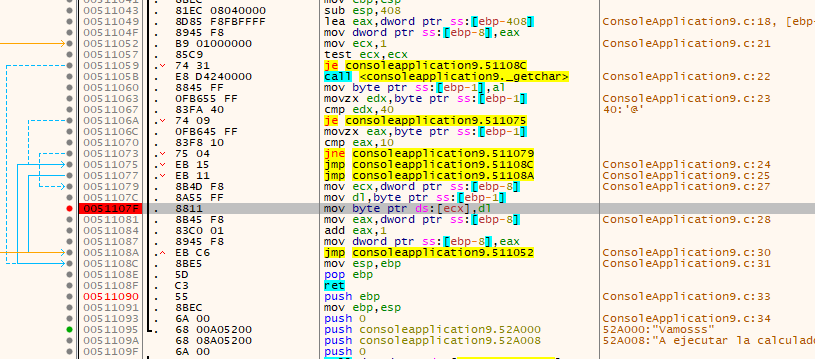
Corramos el script y atachemoslo para verlo en x64dbg.



Vamos a la funcion f y vemos el modo grafico con g.

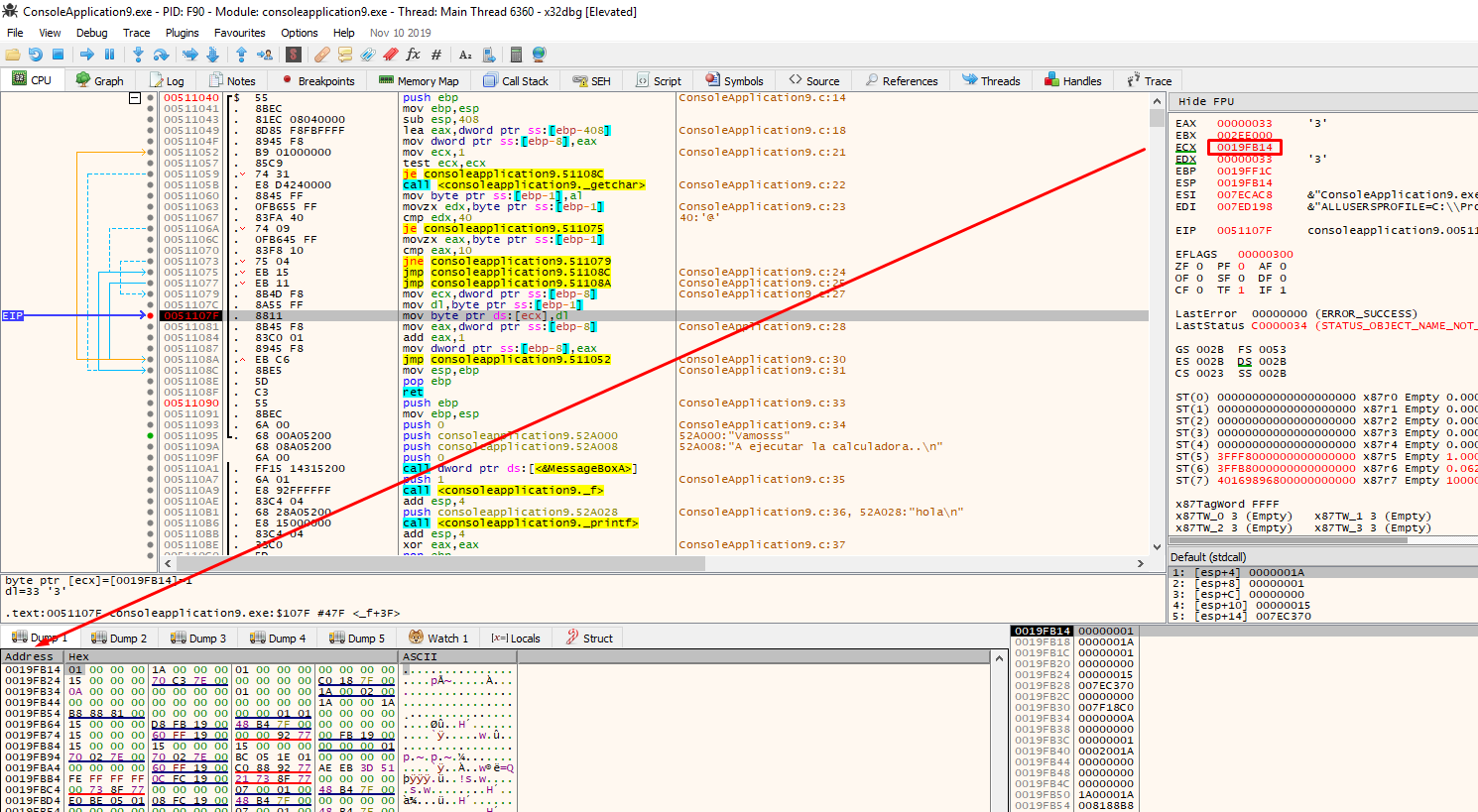


Alli tenemos el bloque donde pisara el pbuffer si ponemos un hardware breakpoint on write cada vez que se incremente parara, así que no sirve de mucho pongamos un breakpoint donde guarda en buffer.



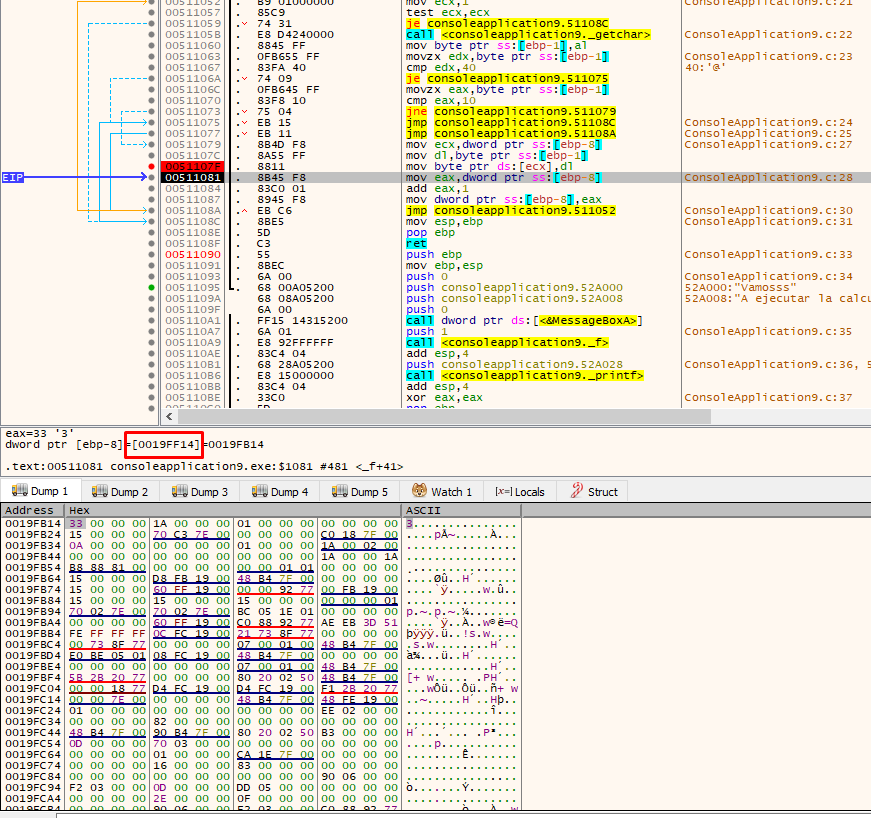
Demos RUN y aceptamos el MESSAGEBOX.

Podemos mirar como va llenado el buffer en cada ciclo con ECX-FOLLOW IN DUMP.

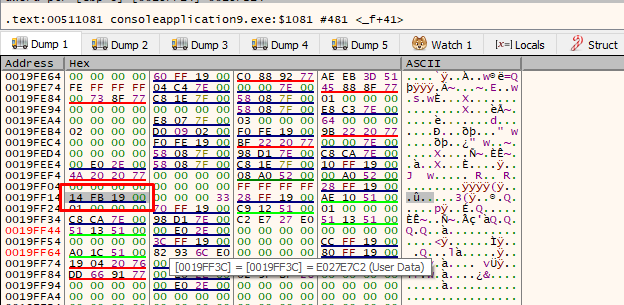


Localicemos pbuffer alli abajo debe estar.

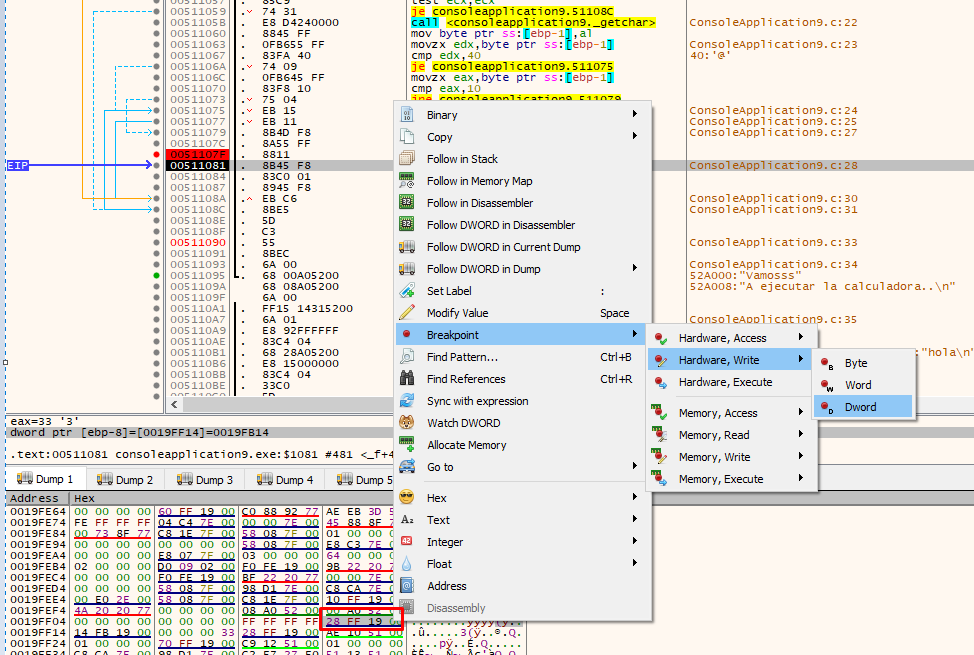
En la siguiente línea lee desde 0x19ff14 y alli la variable guarda la dirección de buffer que va incrementando de a uno.



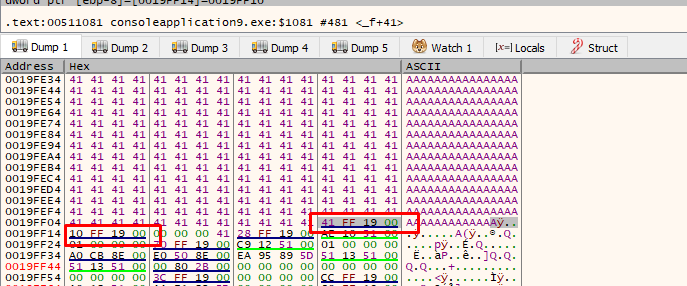
Así que cuando pise 0x19ff14 estaremos cambiando la dirección adonde copiará en el siguiente ciclo.



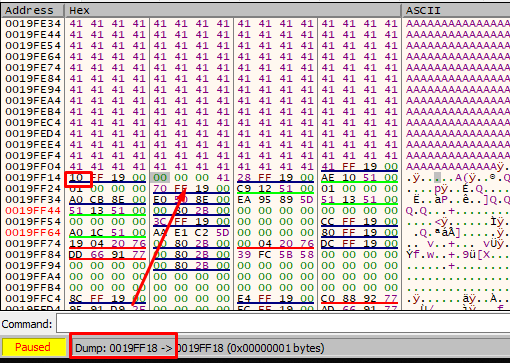
Podemos poner un hardware on write en la dirección justo anterior así no parara en cada ciclo como si lo pusiéramos en el mismo pbuffer.



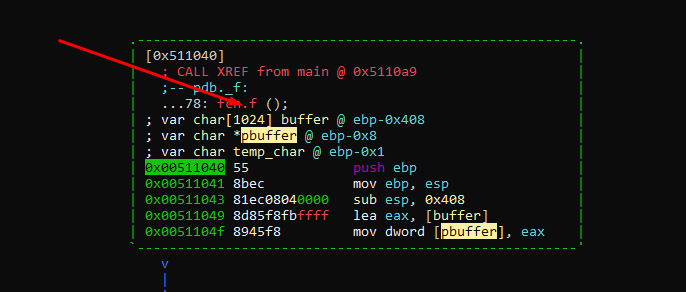
Quitemos el breakpoint que habíamos puesto anteriormente y dejamos solo este para que pare cuando se llene buffer y este justo a punto de pisar el puntero, demos RUN.



Alli paro, va a pisar los 4 bytes anteriores y luego pisará el último byte de pbuffer, si hacemos que ese último byte sea 0x18 la próxima vez escribirá en 0x19ff18 y salteara los tres bytes de la parte alta.



Cuantos bytes debajo de buffer está pbuffer?

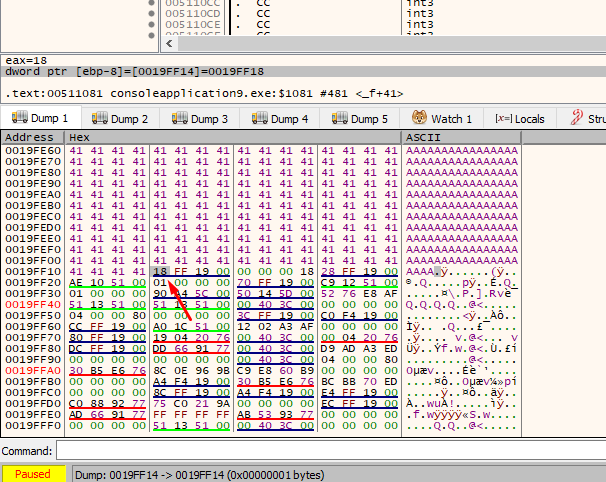


O sea que deberíamos escribir 1024 bytes y luego un 0x18 y luego nos quedarían para pisar el return address ya que seguimos escribiendo debajo de pbuffer, nos quedan los 3 bytes de vacío el byte de temp\_char, y los 4 del stored ebp.

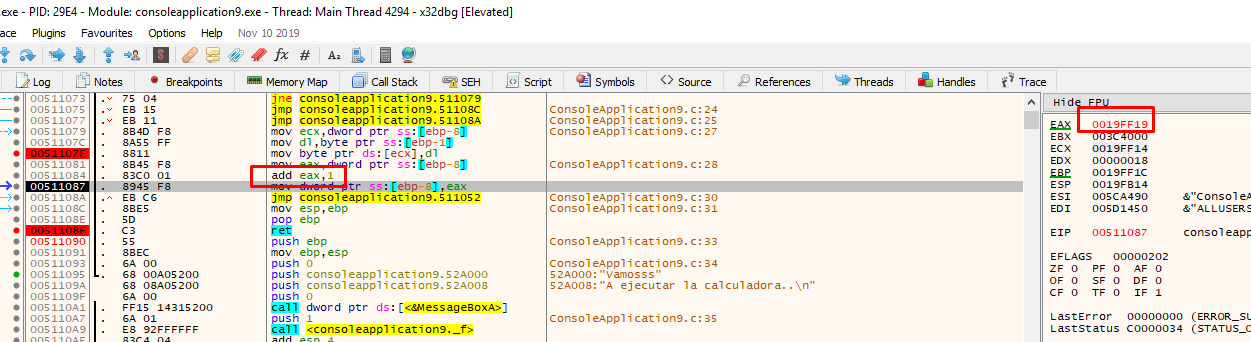
3 + 1+ 4 = 8 bytes



Comprobemos si es así.



Pisa el byte bajo de la dirección así que en el loop siguiente debería escribir en 0x19ff18 y no tocar los 3 bytes restantes de la dirección sigamos loopeando traceando.

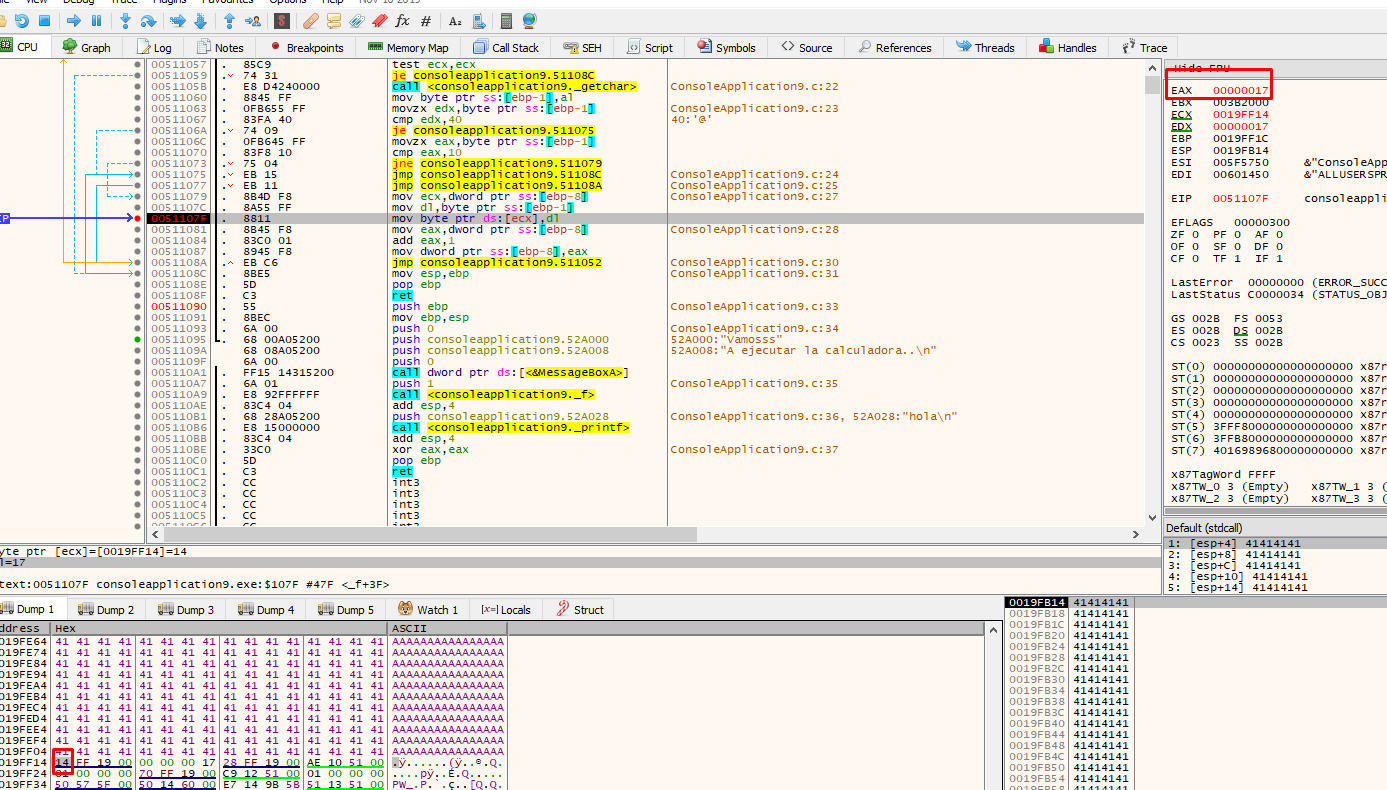


HOUSTON TENGO UN PROBLEMA.

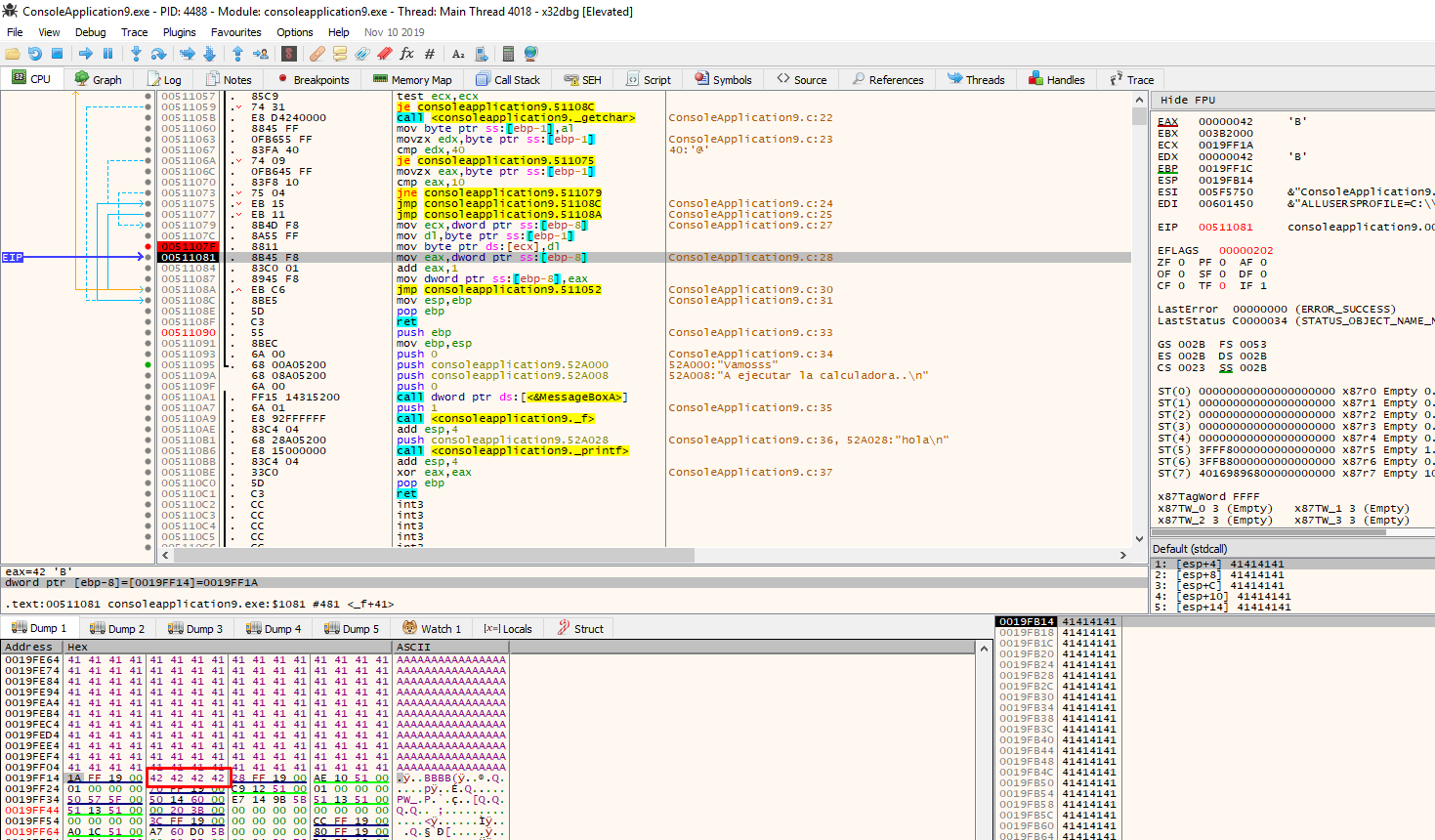
Incrementa el puntero justo después de pisarlo, así que o lo pisamos con 0x17 o corremos un byte todo, mejor lo pisare con 0x17.



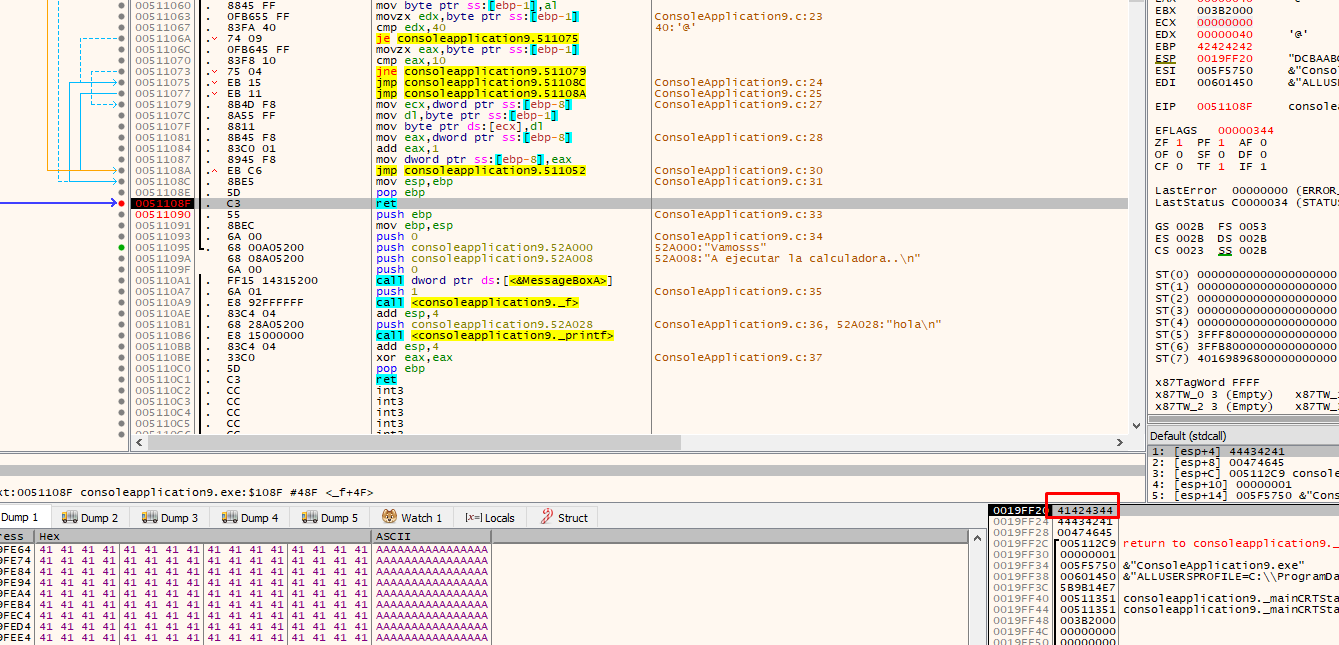
Probemos nuevamente.



Copia el 0x17 sigamos traceando el loop con f8.



Ahora si escribe los 0x24 a partir de 0x19ff18, lleguemos al ret quitando todos los breakpoints.



SCRIPT BASICO PARA EMPEZAR A ROPEAR.

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import sys

from subprocess import Popen, PIPE

import struct

import sys

import codecs

import random

import string

payload = b"A" \* 1024 + b"\x17" + 8\* b"B"+ struct.pack("<L", 0x41424344) + "ABCDEFG" + b"\x40"

p1 = Popen(r"ConsoleApplication9.exe", stdin=PIPE)

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

print("Enter para continuar")

p1.communicate(payload)

p1.wait()

Ahora si ya lo tenemos listo para ropear, lo cuál haremos en la parte siguiente.

Hasta la parte 16

26/04/2020

Ricardo Narvaja