EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

GRATUITAS (PARTE 10)

Luego de haber visto algunos ejemplos básicos en explotación y el reversing, en esta segunda etapa vamos a dar un paso adelante, agregando poco a poco diferentes protecciones y mitigaciones que nos iremos encontrando.

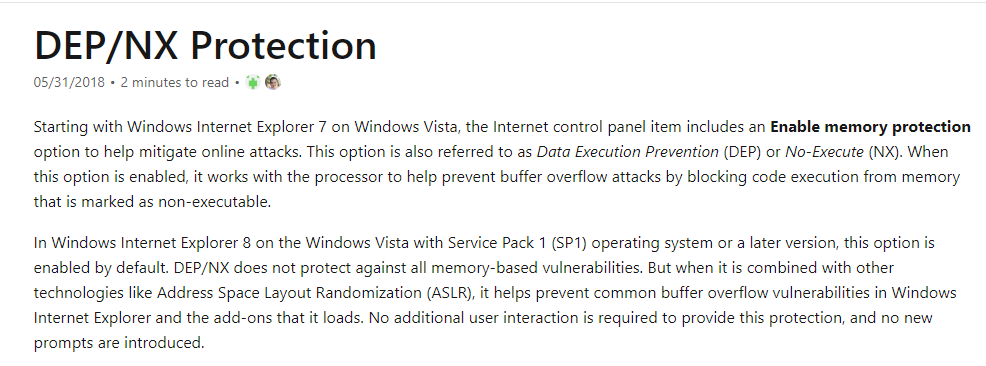
Qué son las mitigaciones?

Con los años se han agregado a los nuevos sistemas, nuevos métodos de defensa genéricos que por supuesto como su nombre lo indica no impiden pero mitigan o hacen más difícil la explotación.

<https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/customize-exploit-protection>

Lo primero que agregaremos será un ejercicio sencillo en 32 bits, similar a los que estábamos viendo pero que tiene habilitada la protección que llamamos DEP, luego haremos lo mismo con el primer ejercicio en 64 bits alli el DEP está habilitado para todos los procesos, así que si o si se necesita saber como manejarse en estos casos.

Que es DEP?



Esa es la definición en inglés que podemos encontrar en la página de MICROSOFT.

La cuestión es que si un proceso tiene DEP habilitado, las zonas de la memoria utilizadas para ingresar o manejar datos como por ejemplo el stack o el heap no tienen permiso de ejecución, o sea solo pueden tener cuando el programa arranca, a lo sumo permiso de lectura y escritura.

Por otro lado, solo tienen permiso de ejecución las secciones de código de los ejecutables y dlls, las cuales a su vez no tienen permiso de escritura.

De esta forma se hace complicado poder ejecutar código propio, pues si en un buffer guarda el código propio que quiero ejecutar y cuando salto a ejecutarlo me da error por no tener permiso de ejecución, no podré explotarlo tal cual lo hacíamos en la etapa 1 de este curso.

O sea que cuando un programa arranca si tiene DEP habilitado:

**SECCIONES QUE MANEJAN DATOS = R o RW (no ejecutables solo lectura y/o escritura)**

**SECCIONES DE CÓDIGO= X (ejecutables)**

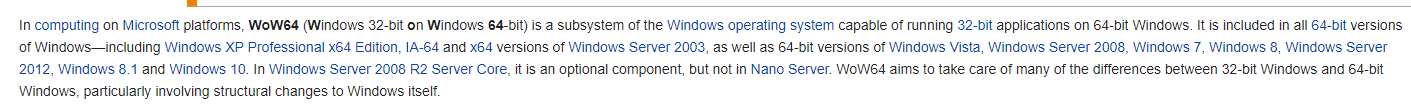
Mientras que sin DEP habilitado

**SECCIONES QUE MANEJAN DATOS = RX o RWX (ejecutables + lectura y/o escritura)**

**SECCIONES DE CÓDIGO= X (ejecutables)**

Otro punto a resaltar es que en los sistemas Windows de 64 bits, los procesos de 32 bits que corren en WoW64 se comportan como en un sistema de 32 bits.

Que es Wow64?



Hablando en forma sencilla para que se den una mejor idea, es como tuviéramos un emulador de un sistema Windows de 32 bits corriendo dentro de un sistema Windows de 64 bits.

O sea si no existiera el subsistema WoW64 dentro de los sistemas de 64 bits, no podriamos correr aplicaciones de 32 bits en los mismos.

Todo esto es transparente para el usuario, el mismo puede ejecutar un proceso de 32 o de 64 bits sin tener que realizar ningún paso adicional, solo haciendo doble click en un ejecutable, el sistema detectará si es de 64 bits y lo ejecutará directamente y si es de 32 bits se lo pasará al subsistema WoW64 para que lo ejecute.

Por lo tanto todos los ejecutables que vimos en la etapa 1 estaban compilados en 32 bits, realmente están corriendo en el subsistema WoW64.

POR DEFAULT:

**PROGRAMAS compilados en 32 BITS o WOW64: según como se compila un programa con o sin DEP el mismo correrá con o sin DEP respectivamente.**

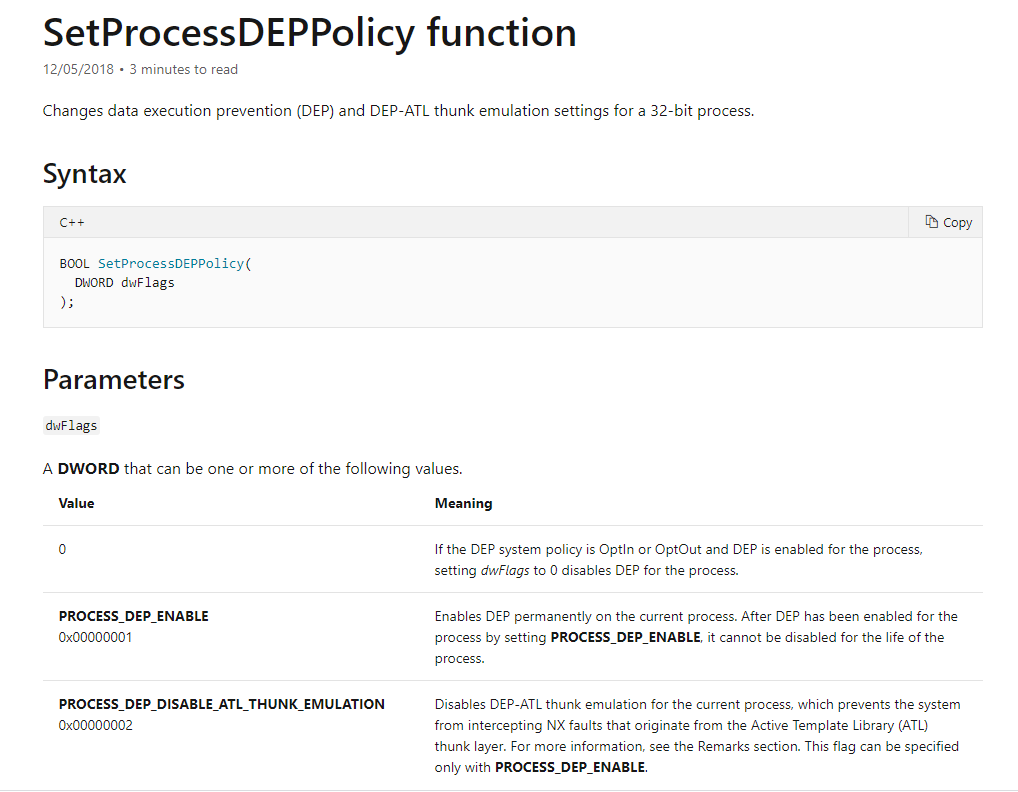
**PROGRAMAS compilados en 64 BITS: No importa como este compilado siempre tendrá DEP HABILITADO.**

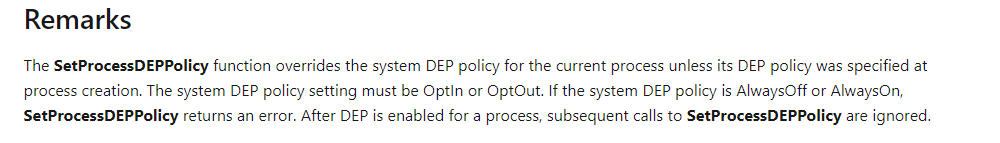
Por supuesto estamos hablando de la configuración de sistema por default la cual puede cambiarse, por ejemplo en sistemas de 32 bits existe la posibilidad de cambiar a que todos los procesos tengan DEP activado como en la configuración por default de sistemas de 64 bits, pero de aquí en más siempre nos referiremos a cada sistema con sus opciones como vienen por default.

Además de todo esto existe una funcion de windows que un programa puede utilizar para activar DEP en runtime en procesos de 32 bits.

Que es SetDEPProcessPolicy ?

<https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setprocessdeppolicy>





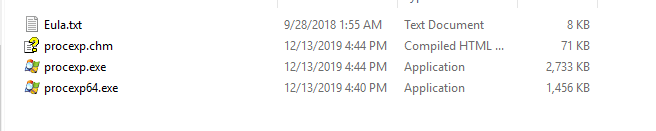
El tema es que solo se usa normalmente para habilitar DEP, ya que solo sirve para deshabilitar DEP si fue habilitado por la misma funcion, el DEP habilitado con otro metodo no se puede deshabilitar usando esto.

Así que tenemos muchas diferentes posibilidades para que un proceso tenga DEP o no habilitado, por lo cual solo ver estáticamente si un ejecutable fue compilado con DEP o no, no es suficiente para saber si lo tendrá habilitado finalmente cuando corra.

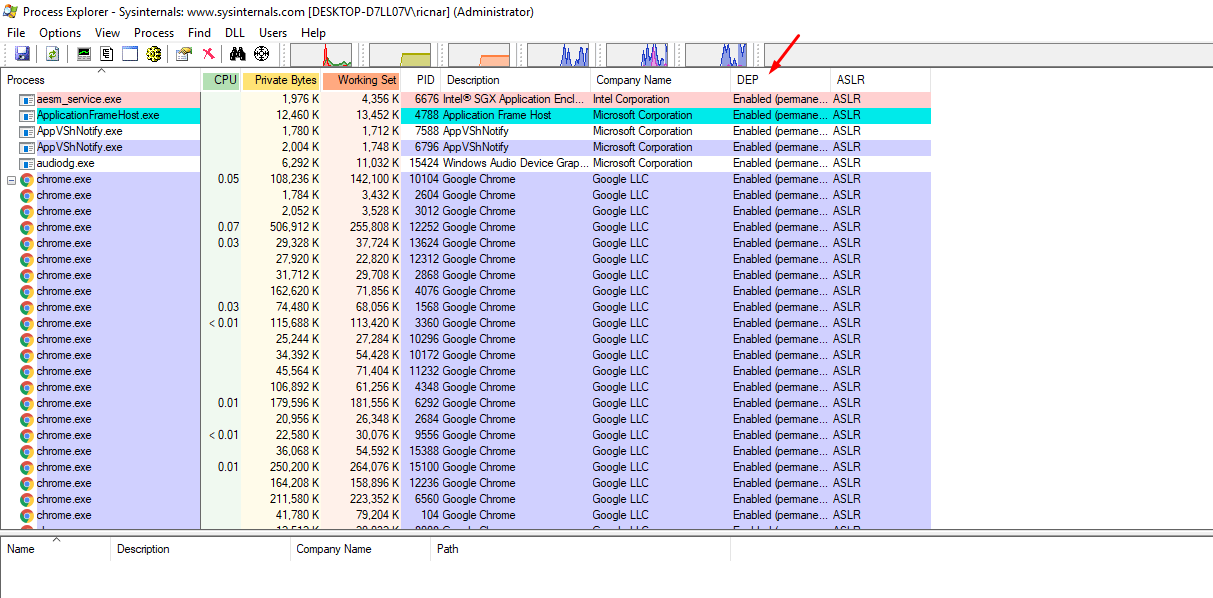
Por eso la mejor forma es utilizar la tool llamada PROCESS EXPLORER que es de MICROSOFT.

<https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer>

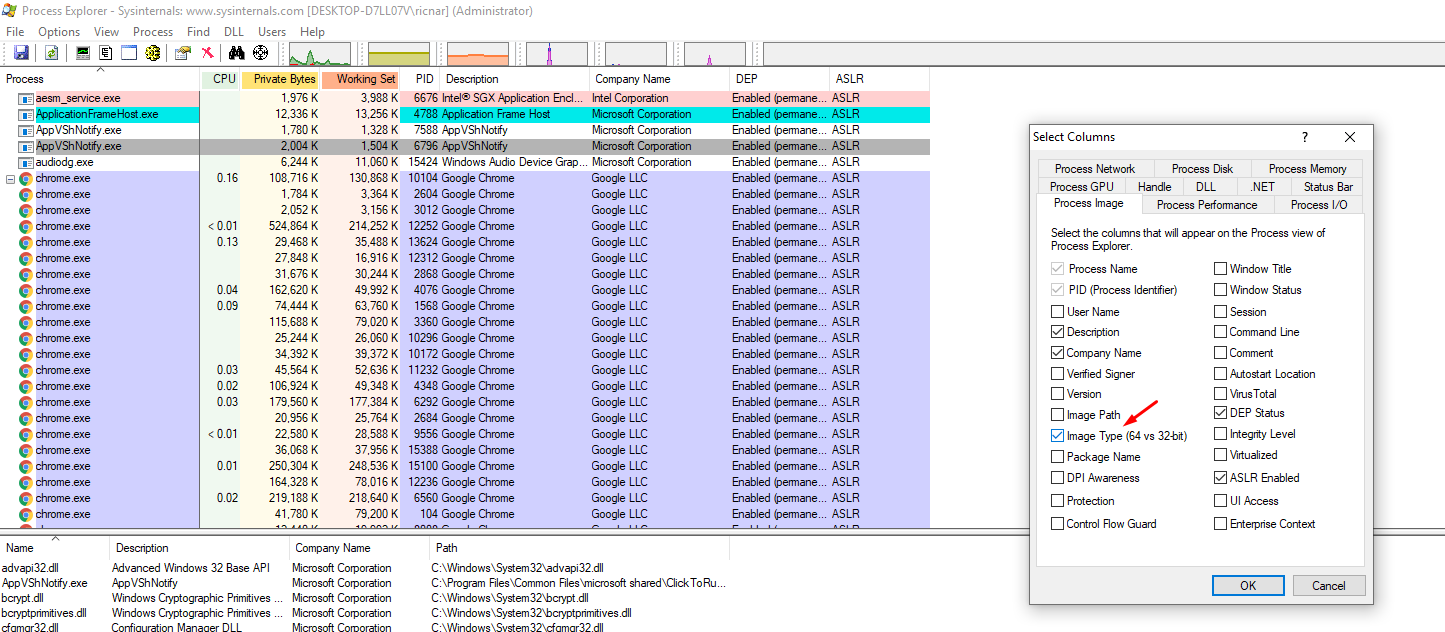
Una vez que la ejecutamos.



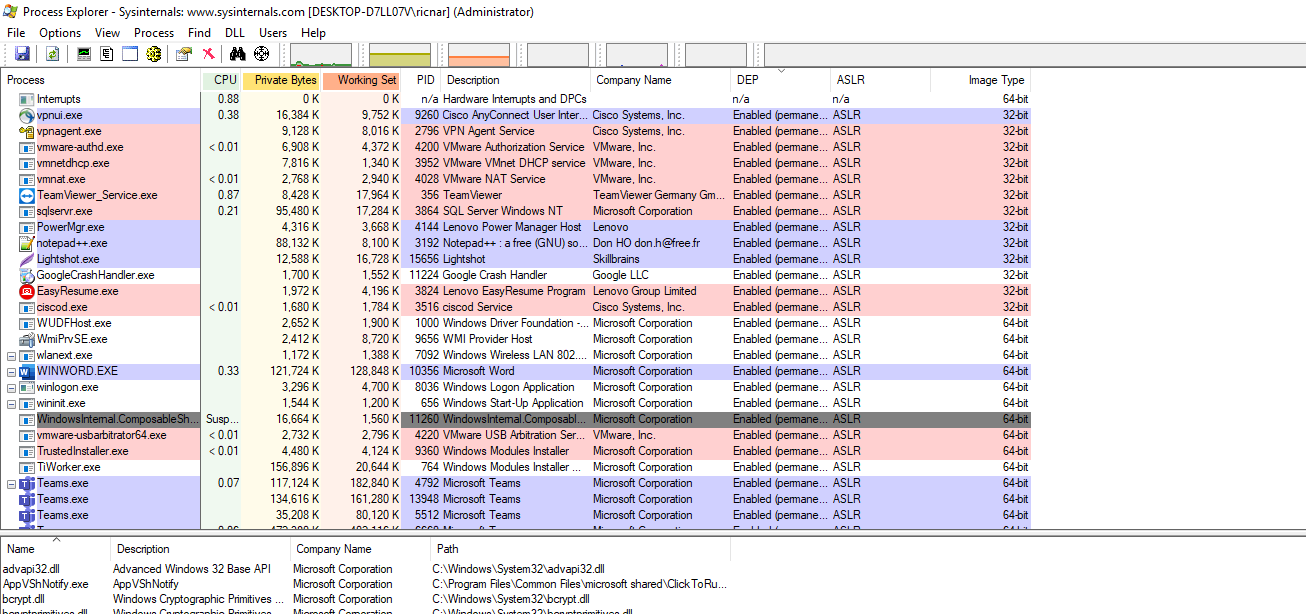
Corramos con permiso de administrador la versión de 64 bits, mostrará todos los procesos en una máquina que tenga un sistema Windows de 64 bits sean de 64 bits o Wow64.



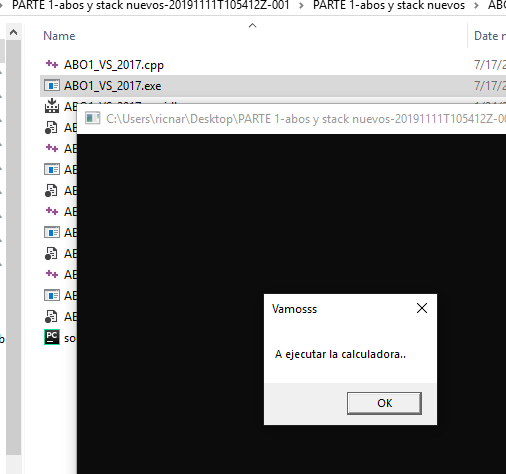
Alli vemos la columna DEP y vemos que como dijimos los procesos de 64 bits tienen DEP habilitado.



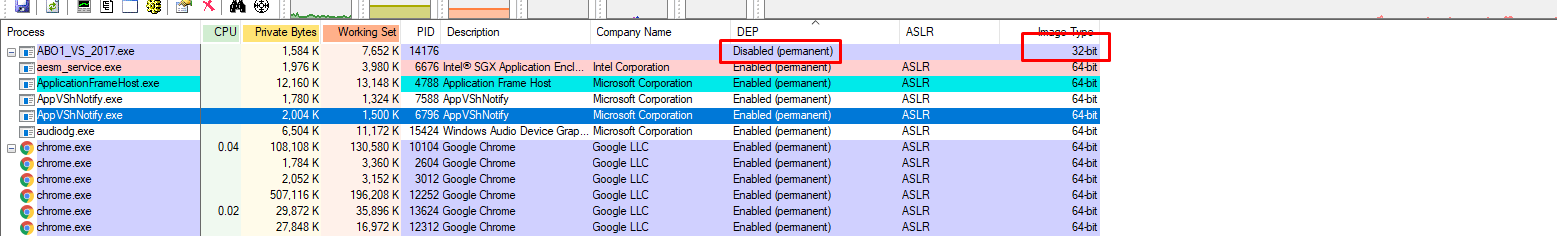
Podemos agregar esa columna con click derecho -SELECT COLUMNS en la barra de columnas.



Vemos que los procesos de 32 bits o sea que corren en WoW64 tienen habilitado DEP en este caso, pero si corro alguno de los ejercicios que vimos en las entregas anteriores.

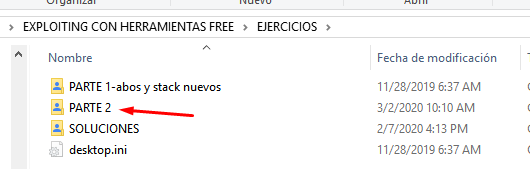


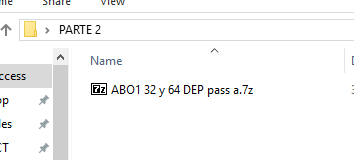
Miremos en el PROCESS EXPLORER este proceso.



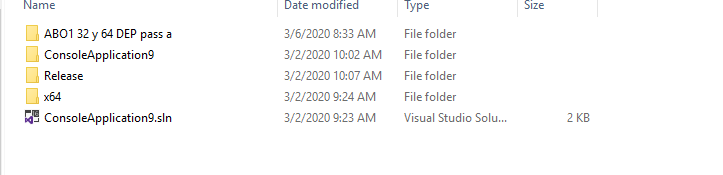
El mismo como estuvo compilado en 32 bits y sin DEP corre sin DEP habilitado.

Ahora si tomo el primer ejercicio que vamos a ver a continuación de la etapa 2.

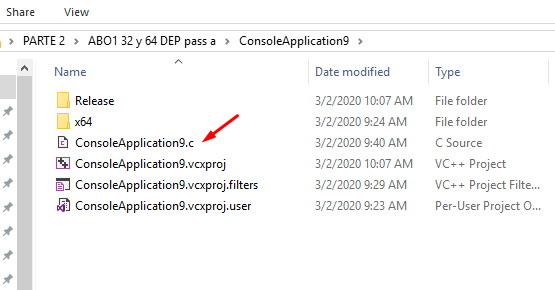


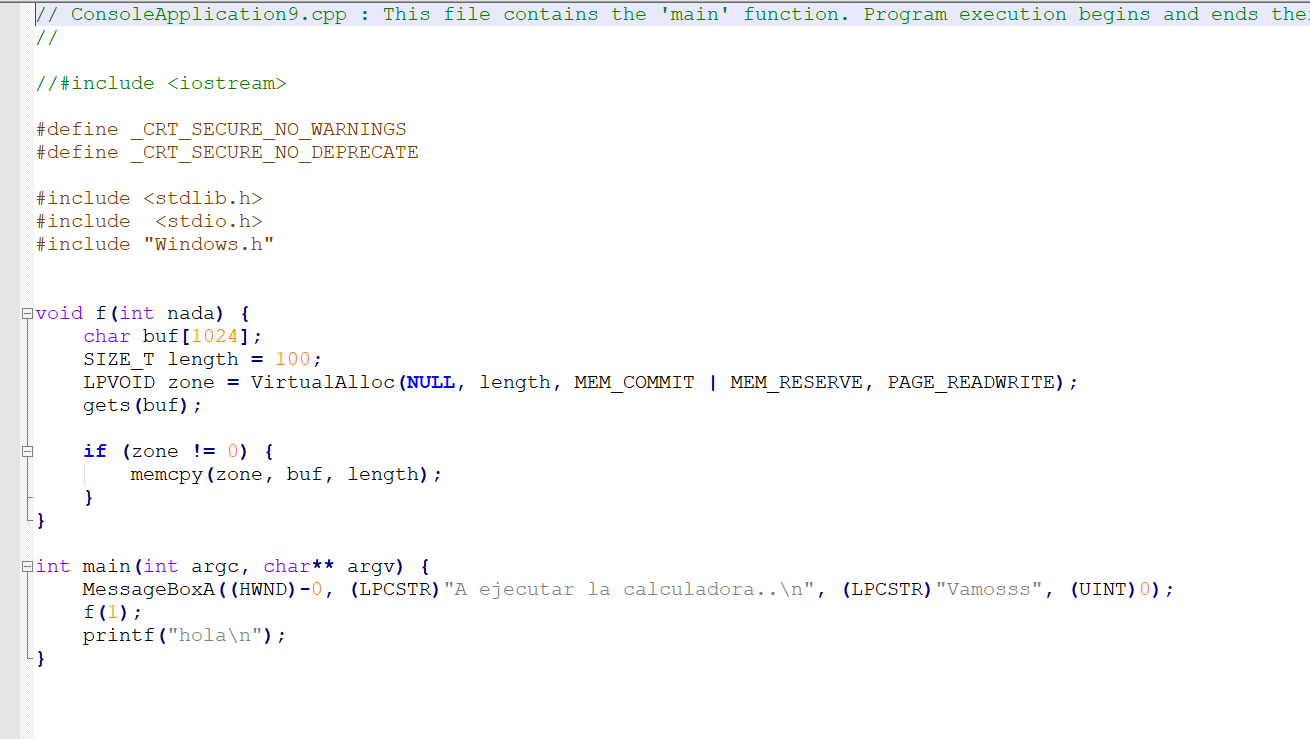


Alli adentro del 7z que para descomprimirlo hay que usar el password **a**, están los dos siguientes ejercicios.

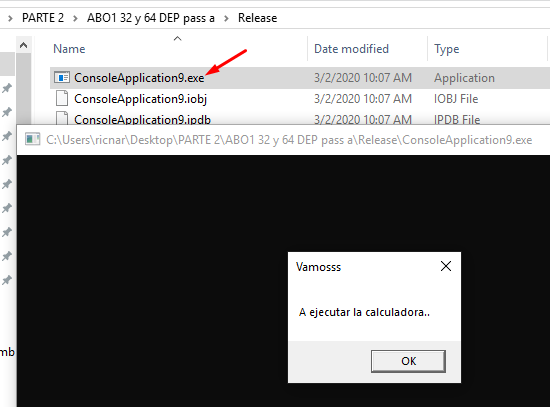


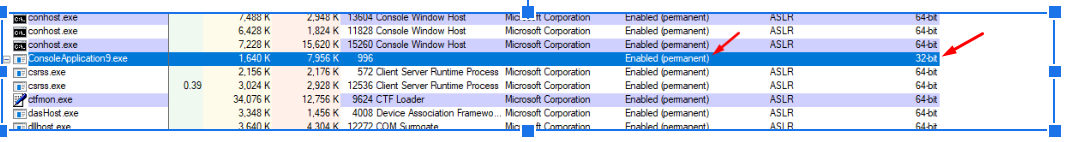
Vemos que es el mismo ejercicio compilado para 32 y 64 bits, dentro de la carpeta Release está la versión compilada de 32 bits y en la carpeta x64 la versión compilada de 64 bits, el que quiere ver también está el código fuente que es el mismo para ambos.



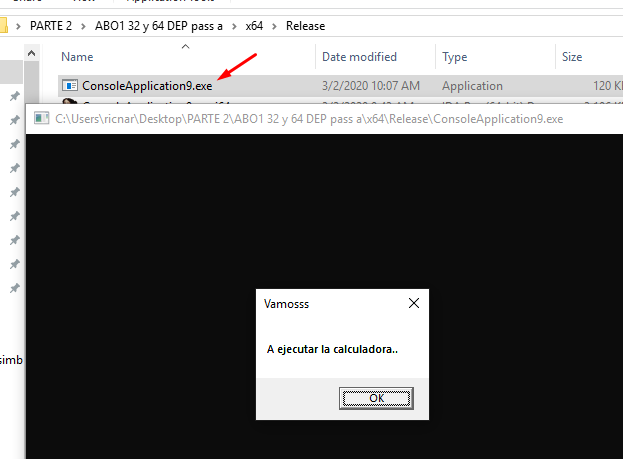


Ejecutemos la versión de 32 bits y veamos en el PROCESS EXPLORER.





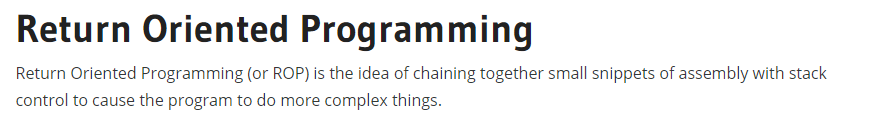
Vemos que este ya tiene habilitado DEP en forma permanente y si corro el de 64 bits también lo tiene habilitado.





Así que nuestro próximo objetivo será el tema DEP y como bypasearlo, para ello debemos estudiar la técnica denominada ROP.

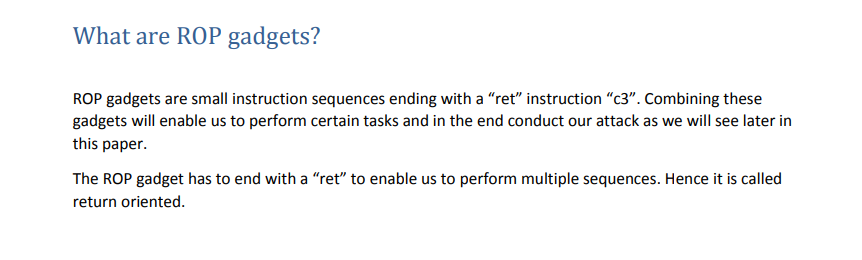
Que es ROP?



Como dice alli es una técnica de encontrar pequeños pedazos de código en los ejecutables o dlls del proceso, usando las secciones de código de los mismos, o sea en las permitidas por DEP pues están marcadas como X o ejecutables.

Estos pedazos de código son llamados GADGETS

Qué son los GADGETS?



No solo pueden terminar en un RET (C3), los gadgets pueden también terminar en C2 (RETN CONST) y hasta en ciertos casos sofisticados pueden terminar en un CALL o JMP.

Haremos que los mismos vayan armando una llamada a una funcion de Windows como VirtualProtect o VirtualAlloc, que nos permita darle permiso de ejecución a una sección de datos que no lo tenia inicialmente o crear una sección nueva que tenga permiso de ejecución lectura y escritura, copiar alli nuestro código y luego saltar a ejecutar el mismo.

O sea que si tenemos un proceso con DEP habilitado:

**SECCIONES QUE MANEJAN DATOS = R o RW (no ejecutables solo lectura y/o escritura)**

**SECCIONES DE CÓDIGO= X (ejecutables)**

Lograremos agregar a alguna parte de la memoria.

**ALGUNA SECCIÓN QUE MANEJA DATOS = RX o RWX (agregamos ejecución ya tenia lectura y/o escritura)**

**SECCIONES DE CÓDIGO= X (ejecutables)**

O sea que la forma de bypasear DEP no es deshabilitarlo para todo el proceso, lo cual no es posible, sino agregarle permiso de ejecución a alguna parte de la memoria que maneja datos para poder ejecutar alli nuestro código.

La idea es que como controlamos y desbordamos el stack, podemos saltar a cualquier dirección que queremos, y si en vez de saltar como antes a un CALL ESP, CALL EAX o un código que saltaba directamente a ejecutar nuestro código, saltemos por ejemplo a un gadget por ejemplo que haga POP ECX-RET.

Esto ejecutará POP ECX que copiará el valor que está en el stack que controlamos nosotros a ECX y luego el RET hará que saltemos al siguiente gadget que también controlamos pues estará debajo del valor que se copio a ECX.

Así vamos empalmando la ejecución de gadgets luego del primero continúa con el segundo gadget y así cada uno va moviendo los valores que necesitamos a cada registro para armar una llamada a la funcion VirtualAlloc o VirtualProtect.

Obviamente empalmar gadgets existentes es algo que no siempre será similar, hay veces que hallaremos los gadgets que necesitamos en los módulos que no tienen randomización y tienen direcciones fijas donde saltar o se complementa con técnicas de leakeo de direcciones que nos permitan obtener direcciones de módulos para evitar ASLR y hay veces que no será tan sencillo y habrá que rebuscarselas más.

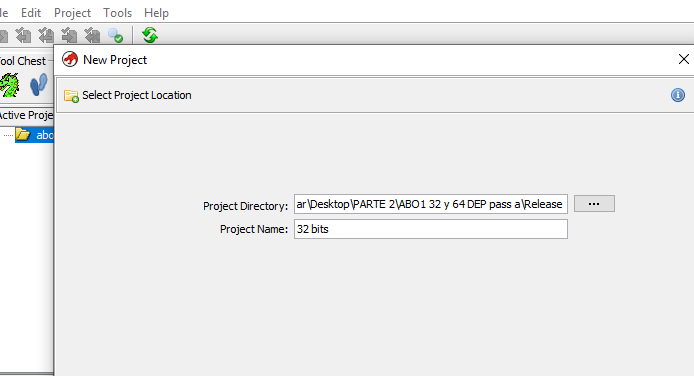
Por eso las técnicas de ROP deben practicarse bastante, son muy cambiantes según el entorno y de la práctica del exploit writer depende mucho de que se pueda realizar un ROP exitoso o no.

Reverseemos el ejecutable de 32 bits con DEP en GHIDRA.

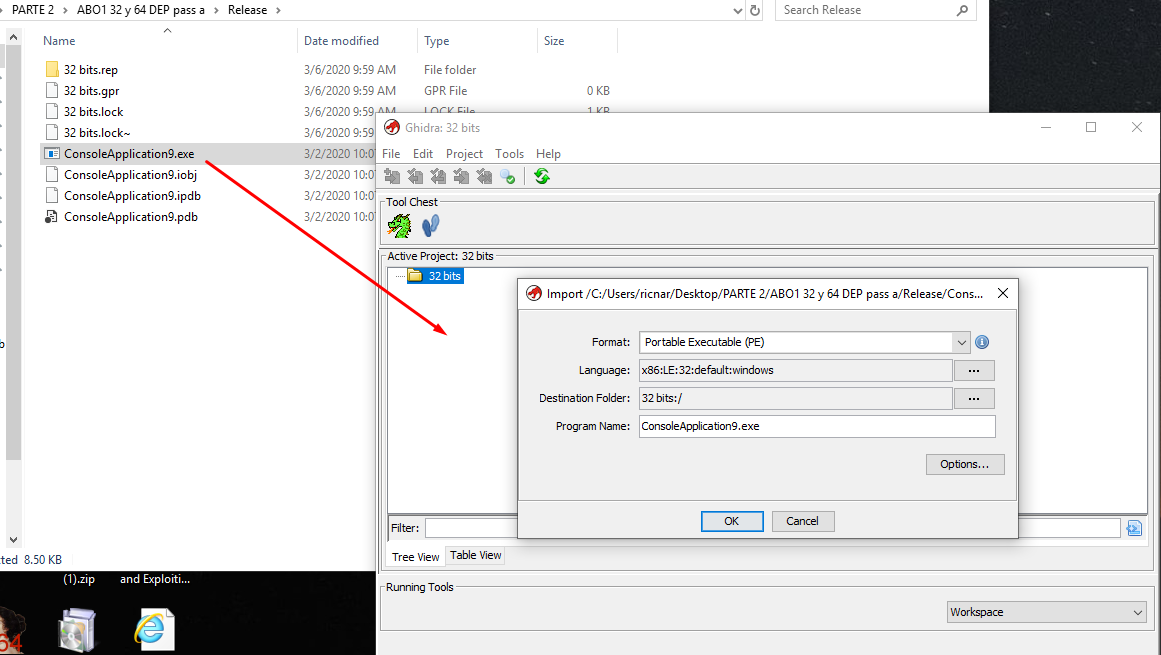
Se puede actualizar a la más nueva versión en este momento es:

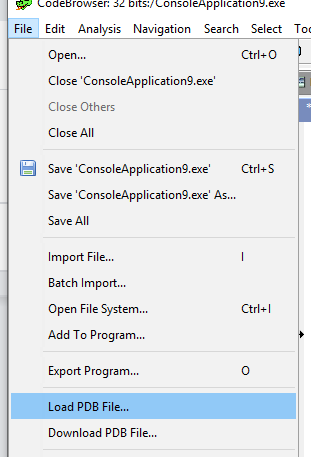


Borro el proyecto que estaba trabajando anteriormente y creo uno nuevo.



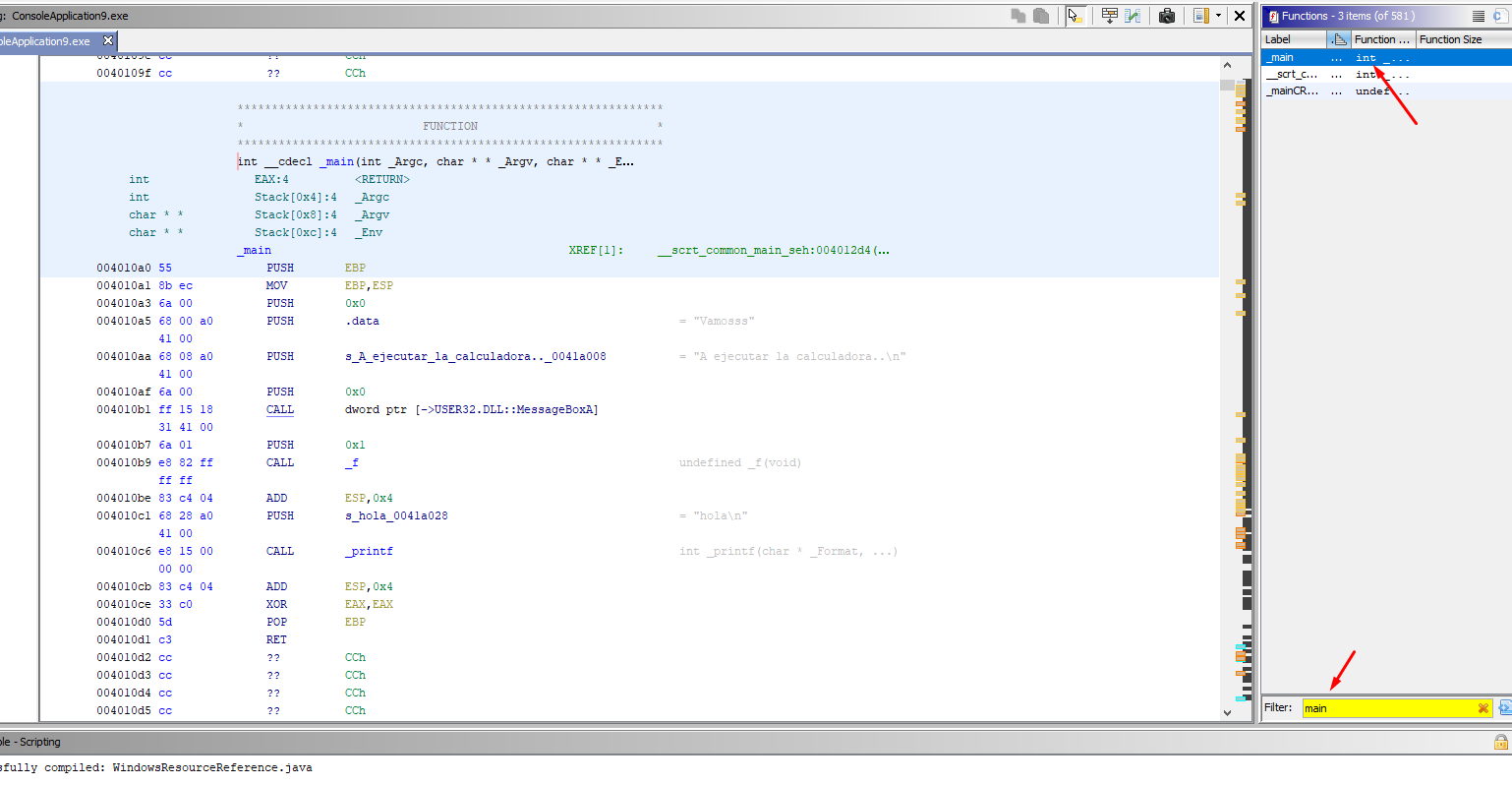
Arrastro y suelto el ejecutable en el proyecto creado.





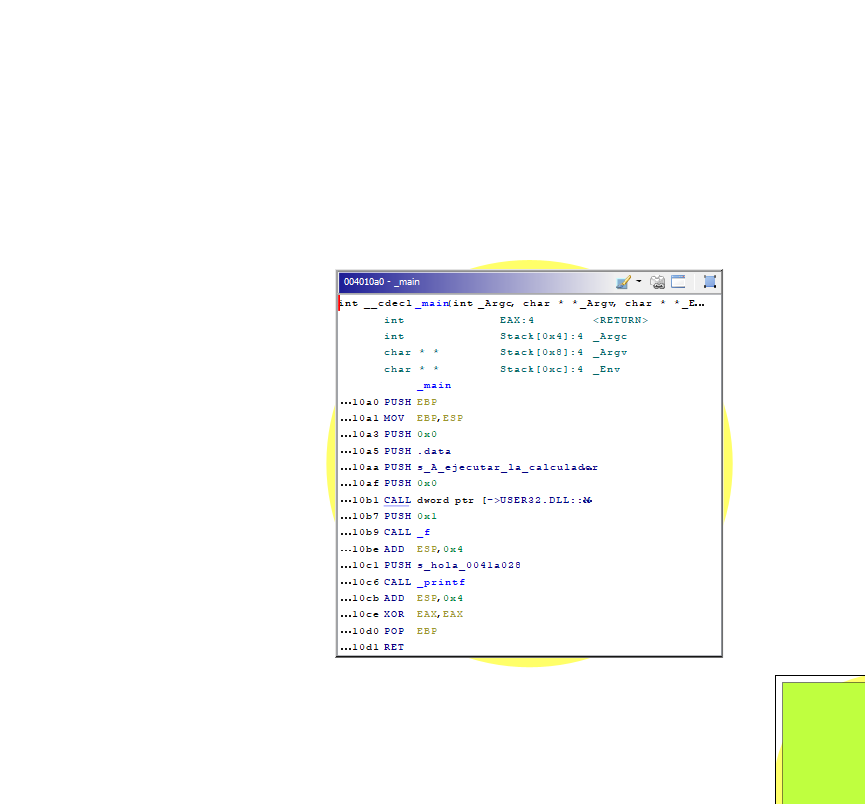
Una vez que analiza, si intento cargar el pdb con los símbolos, me dice que ya lo cargo, así que no necesita cargarlos de nuevo.

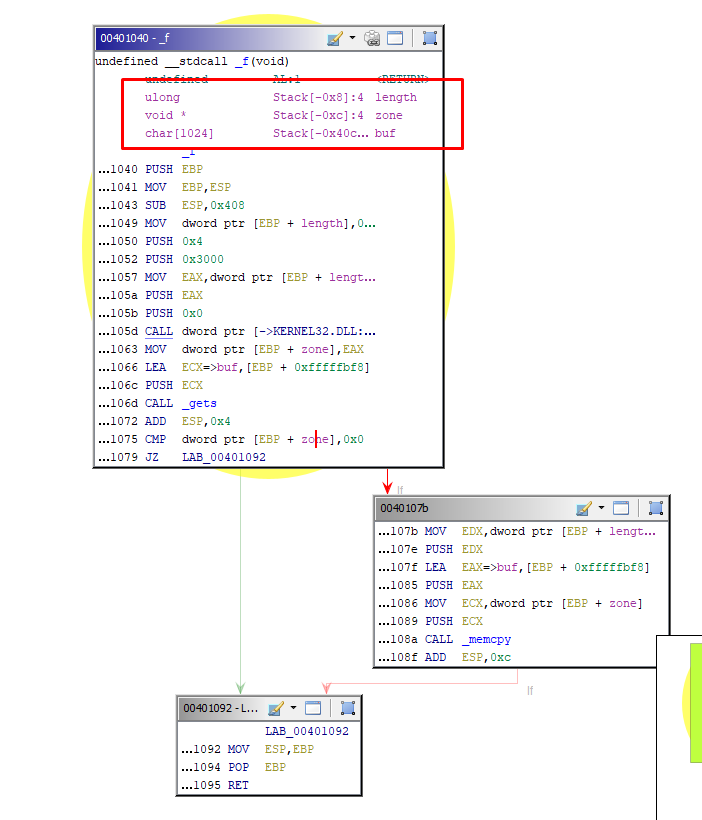
Bueno vemos la funcion principal main.



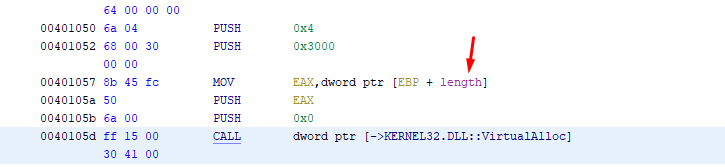
La buscamos en el buscador de funciones que tiene incluido y luego hacemos doble click para ir a la funcion.

Podemos verla en modo gráfico con WINDOWS - FUNCTION GRAPH.

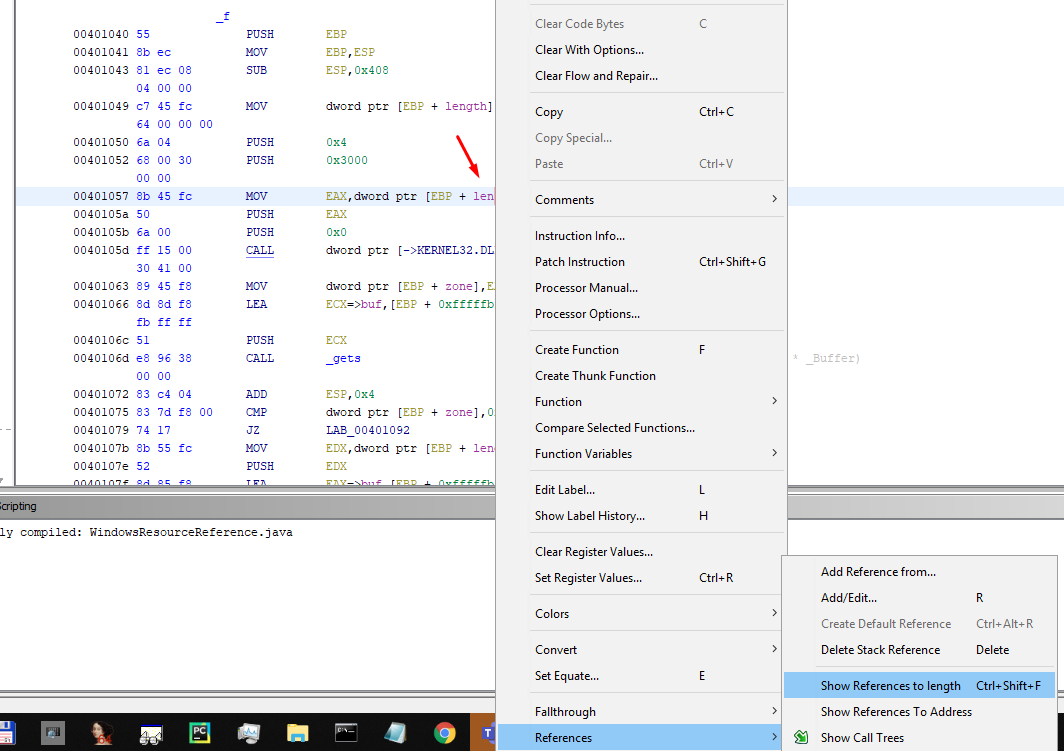


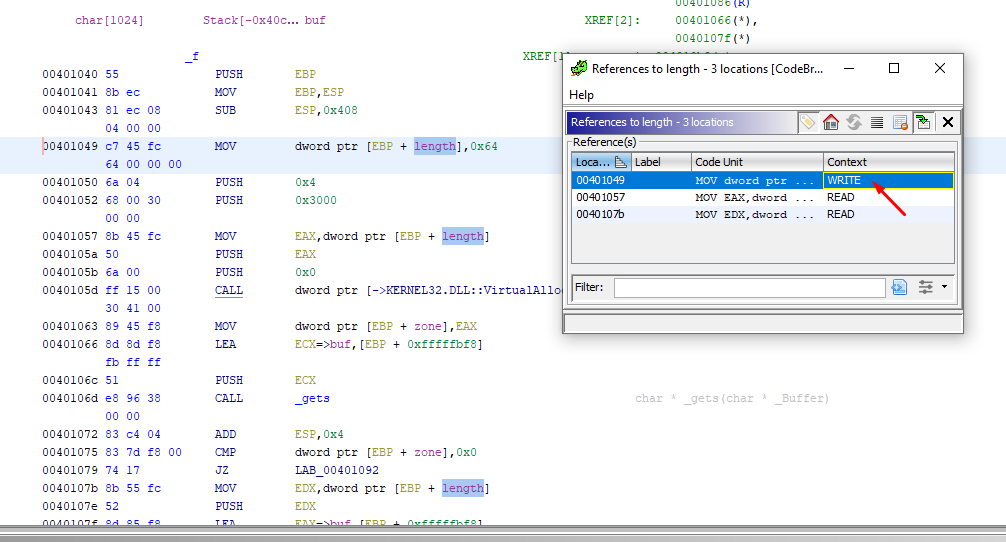
Es un solo bloque ya vimos que aquí no hay nada, solo el call a MessageBoxA y luego el llamado a la funcion f que es donde está el overflow.

Alli vemos que es bastante similar a la del ejercicio que ya vimos aunque tiene alguna diferencia, un llamado a VirtualAlloc para reservar en la memoria una cantidad de bytes dada por el argumento length que se le pasa alli.



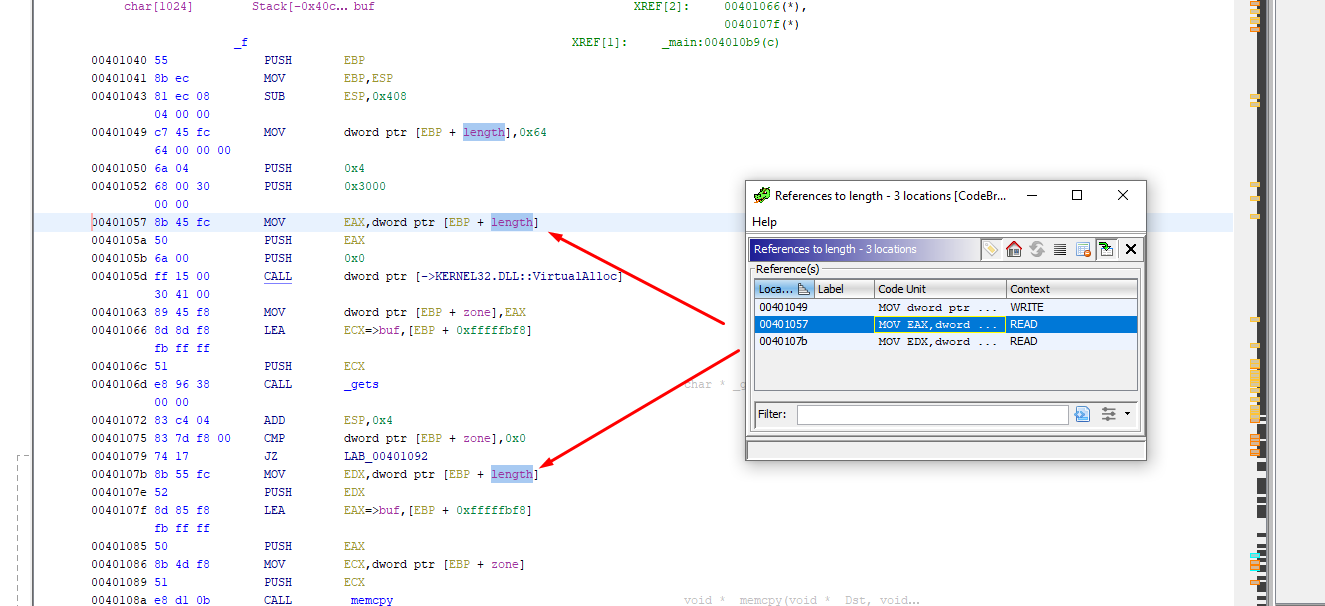
Si hacemos click derecho en la variable lenght, podemos ver donde se usa



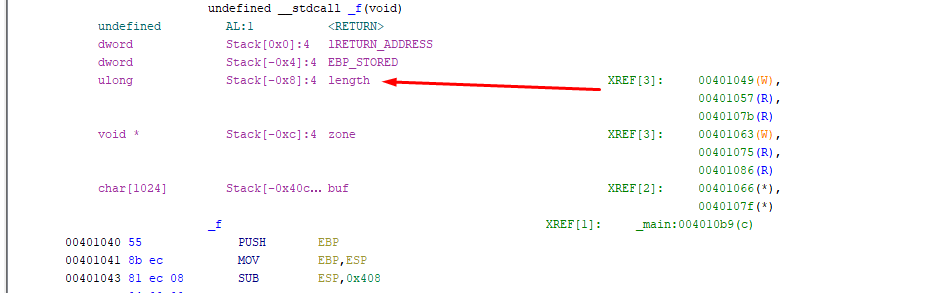


Nos marca los lugares donde es usada, cuando se escribe en la misma (WRITE) y las veces que se lee su valor (READ).

Vemos que la primera vez que se accede guarda en la misma el valor 0x64 y luego hay dos accesos más para usar el valor, uno como argumento de VirtualAlloc y el último para un memcpy que hay para copiar la data que entra al buffer por medio de gets a esa memoria allocada de 0x64.



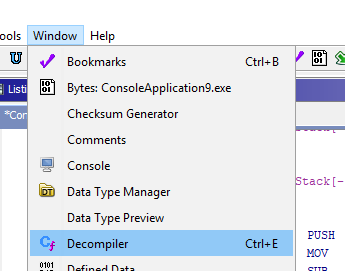
También muestra las referencias aquí.

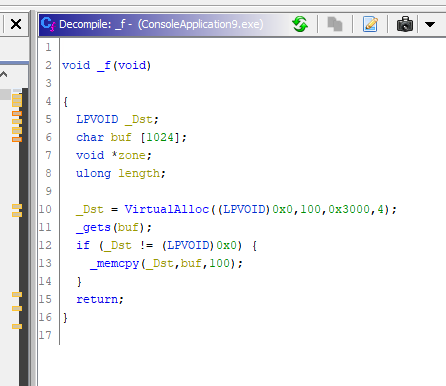


Bueno ya con el gets tenemos un overflow ya que copiara sin control, en el memcpy inicialmente no hay problema porque aunque el buffer de entrada sea mayor y uno ingrese una cantidad de datos mayor a 0x64 que es el tamaño reservado, como solo copia 0x64, alli no habrá overflow.

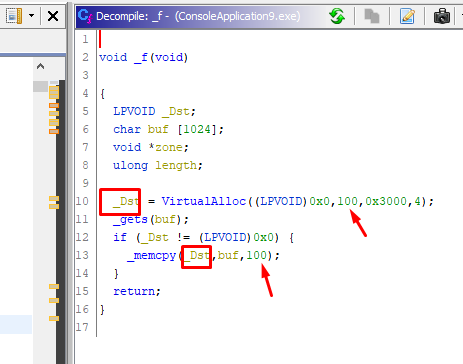
Mientras que el size a copiar sea igual al size del buffer de destino como en este caso, no habrá overflow, ya que no copiara más del tamaño del buffer.

Podemos decompilar ya que GHIDRA tiene decompilador.



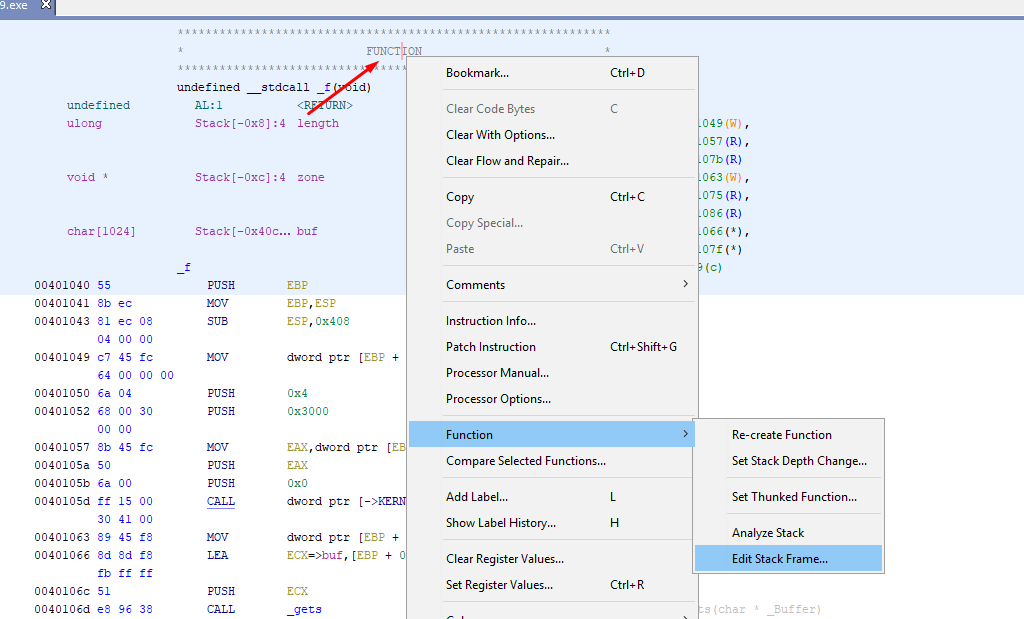


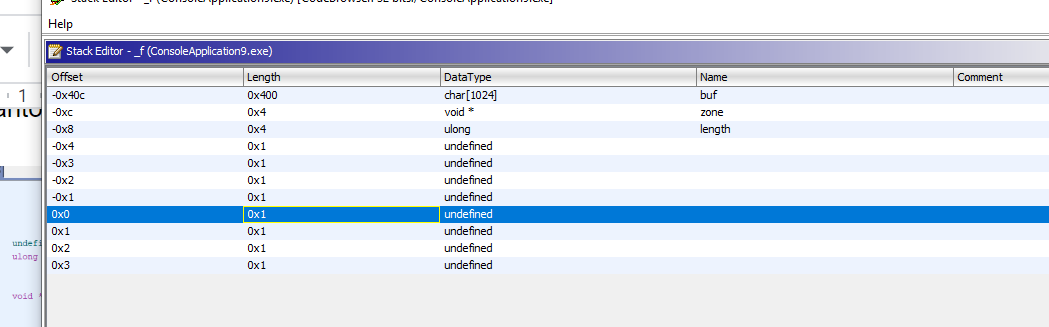
La funcion fue decompilada casi exactamente y se ve lo que decíamos, la llamada a VirtualAlloc para reservar 100 bytes (0x64h) lo unico que aquí no usa la variable length sino reemplaza por el valor 100 directamente.



Pero se ve que el overflow ocurre en el gets, la funcion memcpy está correctamente usada con un \_Dst de tamaño 100 bytes y la cantidad de bytes copiados es 100 también, no desbordara.

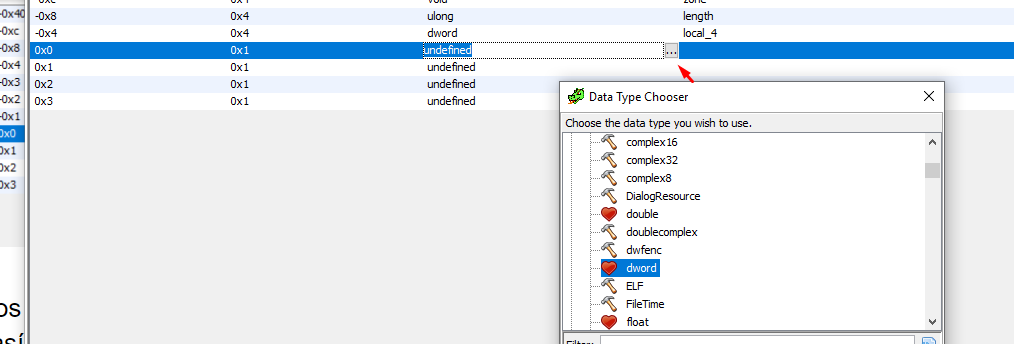
Cuánto debemos copiar para pisar el return address.

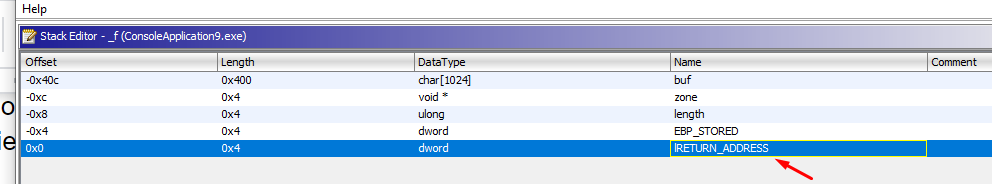




Vemos las tres variables **buf, zone y lenght**, también vemos que **lenght** se ubica debajo de **buf** así que como vimos que en el **memcpy** solo copiaba 100 inicialmente, podrá copiar más también al overflodear **buf** y pisar el valor de **length**.

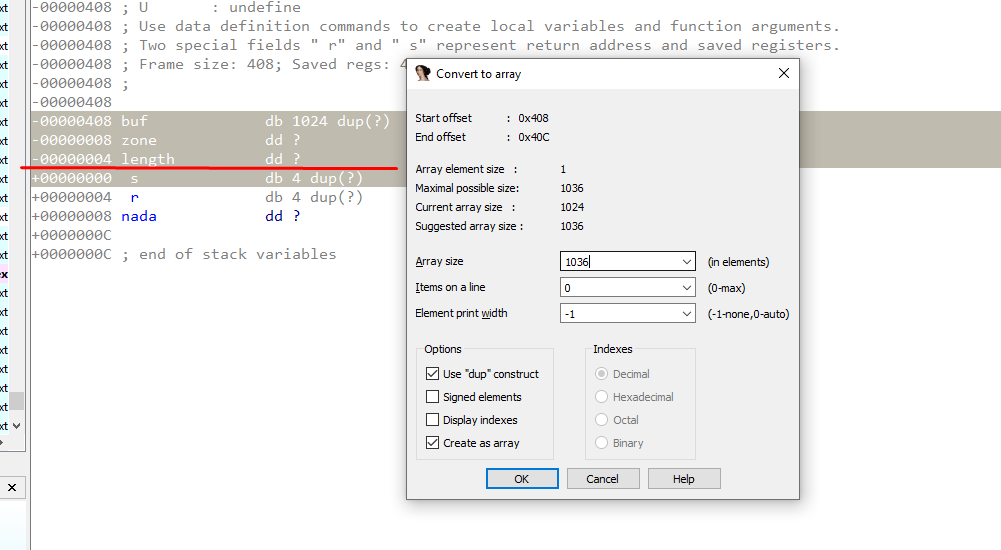
Debajo están el EBP GUARDADO y el RETURN ADDRESS, cada uno de 4 bytes, cambiamos a DWORDS y renombramos.





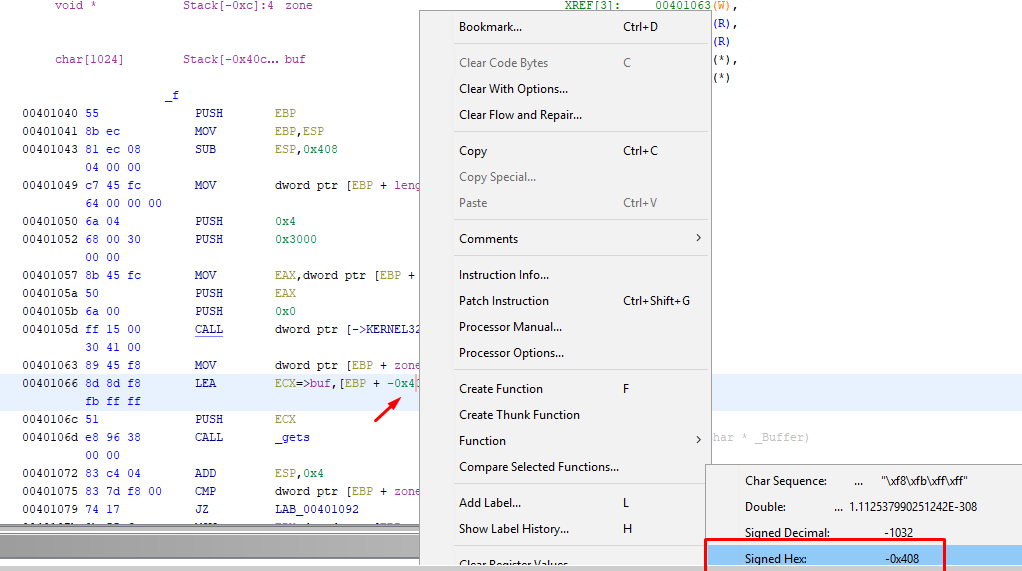
Alli quedó mejor, como en GHIDRA el HORIZONTE es el valor de ESP al inicio de la funcion, justo debajo estará el RETURN ADDRESS que está en offset 0x0 y hacia arriba hay 0x40c.

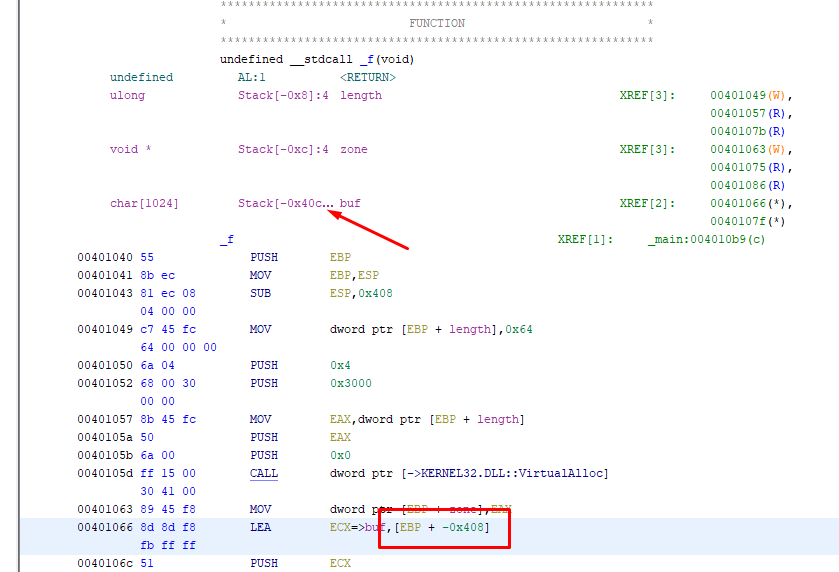
En IDA el HORIZONTE es el valor de EBP después de setearse en el PRÓLOGO y nos muestra que los datos a enviar para pisar justo hasta antes del RETURN ADDRESS serían 1036, rápidamente.



La variable buf se ubicaría en IDA en offset = - 0x408 ya que se toman referencias con respecto a EBP.

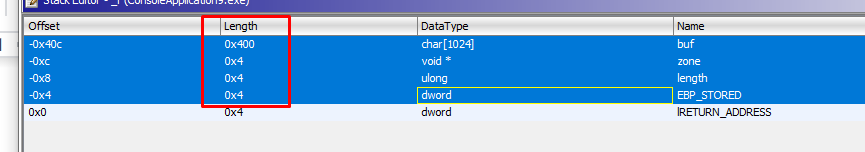
Pero GHIDRA al tomar como referencia ESP tiene está dualidad.





En la variable muestra EBP-0x408 toma como referencia EBP al igual que IDA, pero en la definición y en las distancias nos muestra 0x40C que es la distancia a ESP al inicio de la funcion.

Y bueno hay que tener bien en cuenta esto para no marearse.



Igual si sumamos los largos de las variables hasta justo antes del RETURN ADDRESS seria

0x400 + 4 +4 +4 = 1036

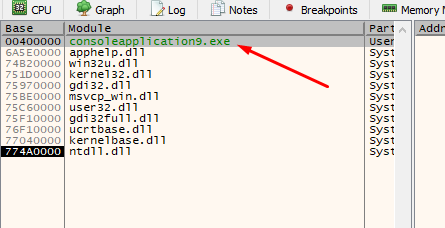


Así que si uno no se marea y sabe lo que hace obtiene el mismo resultado en IDA que en GHIDRA.

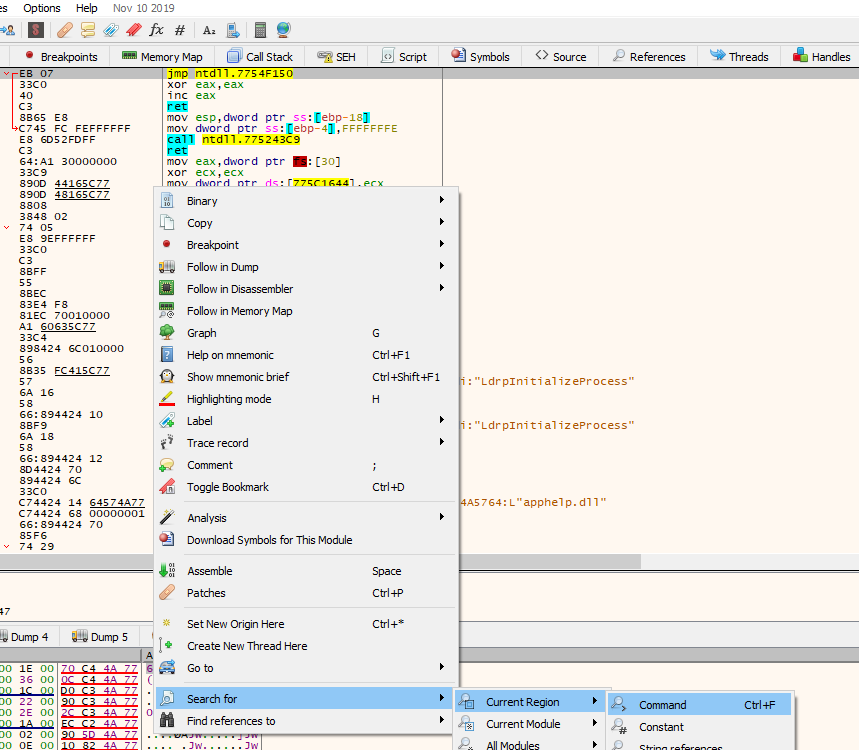
Si no tuviera DEP activado el script para explotar este ejercicio sería.

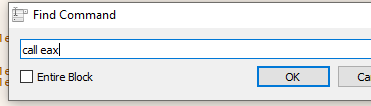


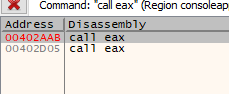
Si lo abrimos en x64dbg puedo ir con VIEW-MODULES



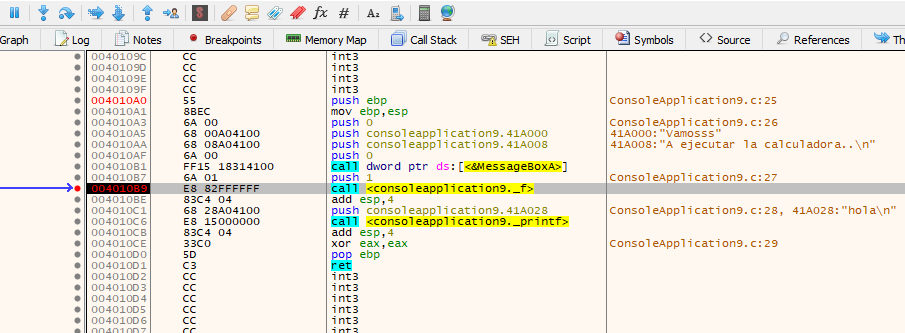
En el ejecutable la instrucción CALL EAX, ya que EAX queda con la dirección de buf, para saltar alli a ejecutar.





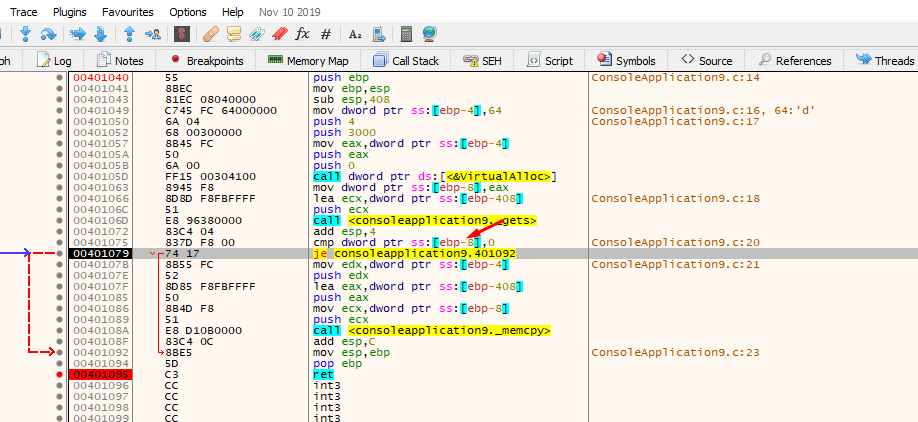


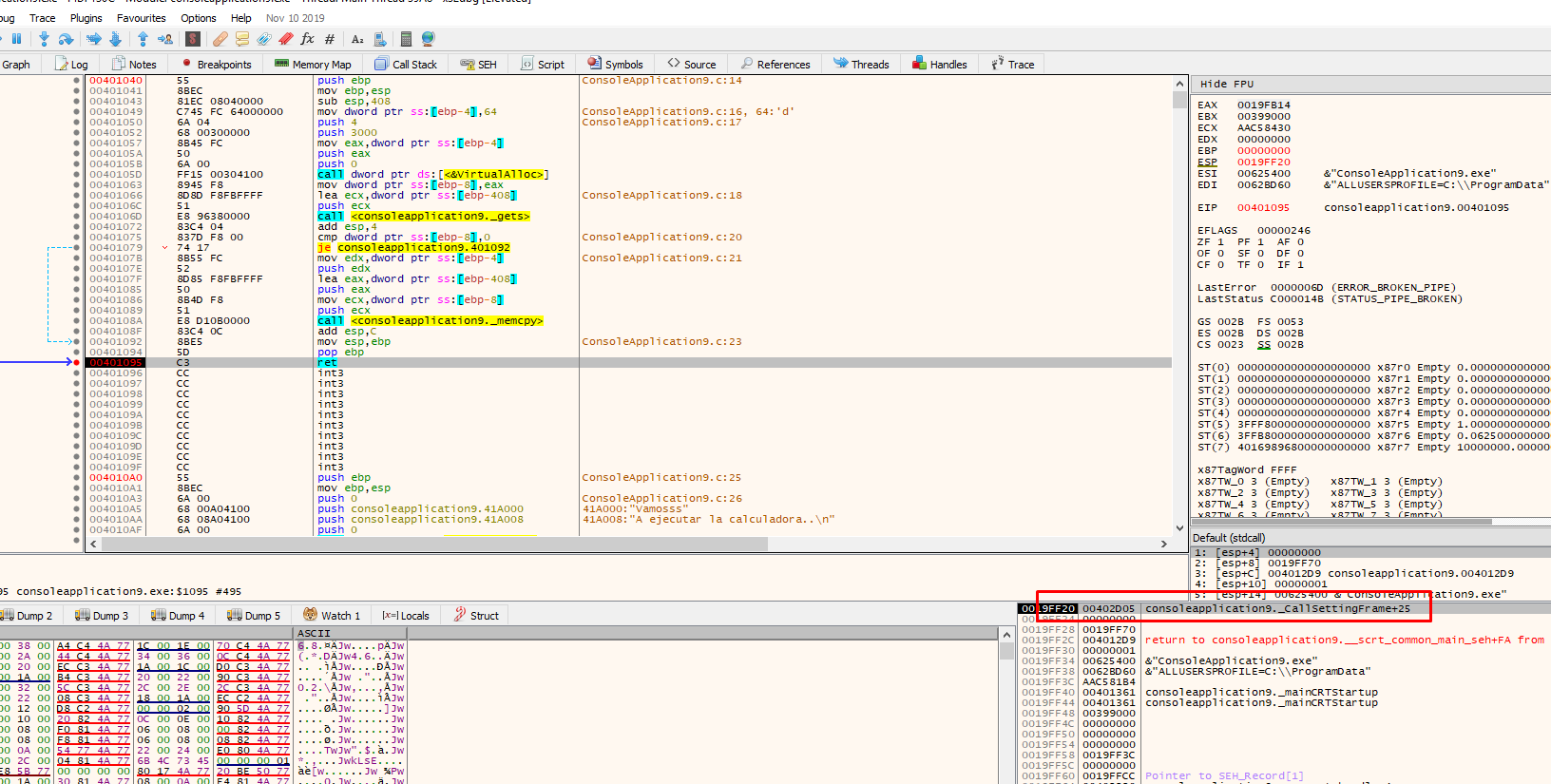
Usando alguno de esos CALL EAX y colocando el SHELLCODE al inicio del buffer si no tuviera DEP activado podría saltar perfectamente a ejecutar, corramoslo atacheemos el x64dbg pongamos un breakpoint al volver del MessageBoxA y parara alli.



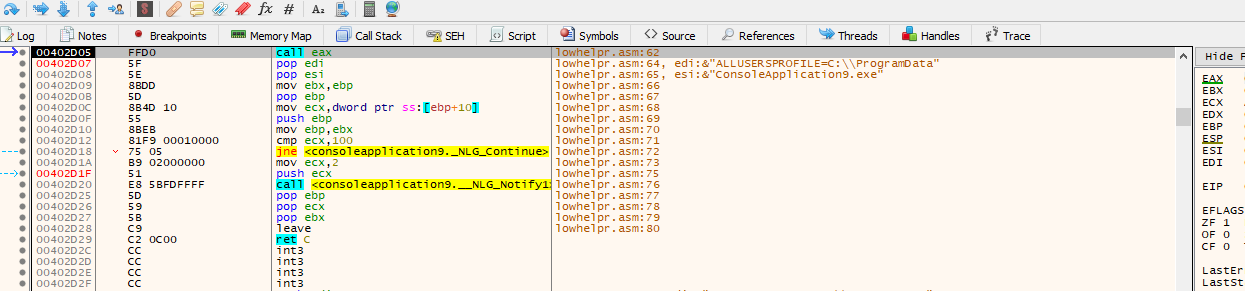
Traceo con F7 para entrar a la funcion **f** .

Como llene el buffer con el shellcode y luego complete con ceros el valor de flag que hace que vaya al memcpy o no, será 0 y saltara.



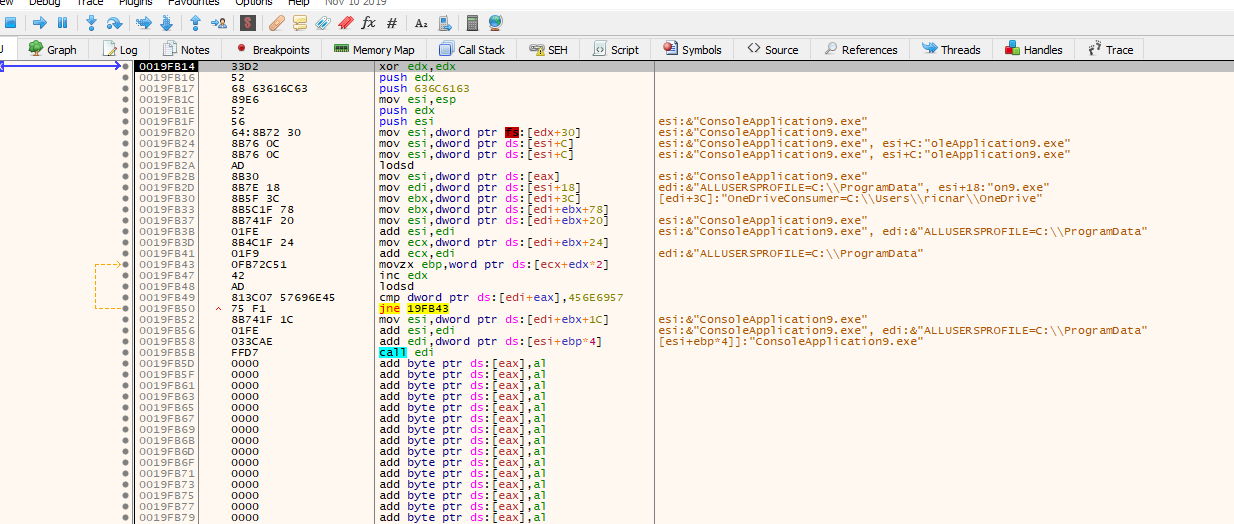


Al llegar al return address apunta al CALL EAX, si sigo ejecutando con f7.

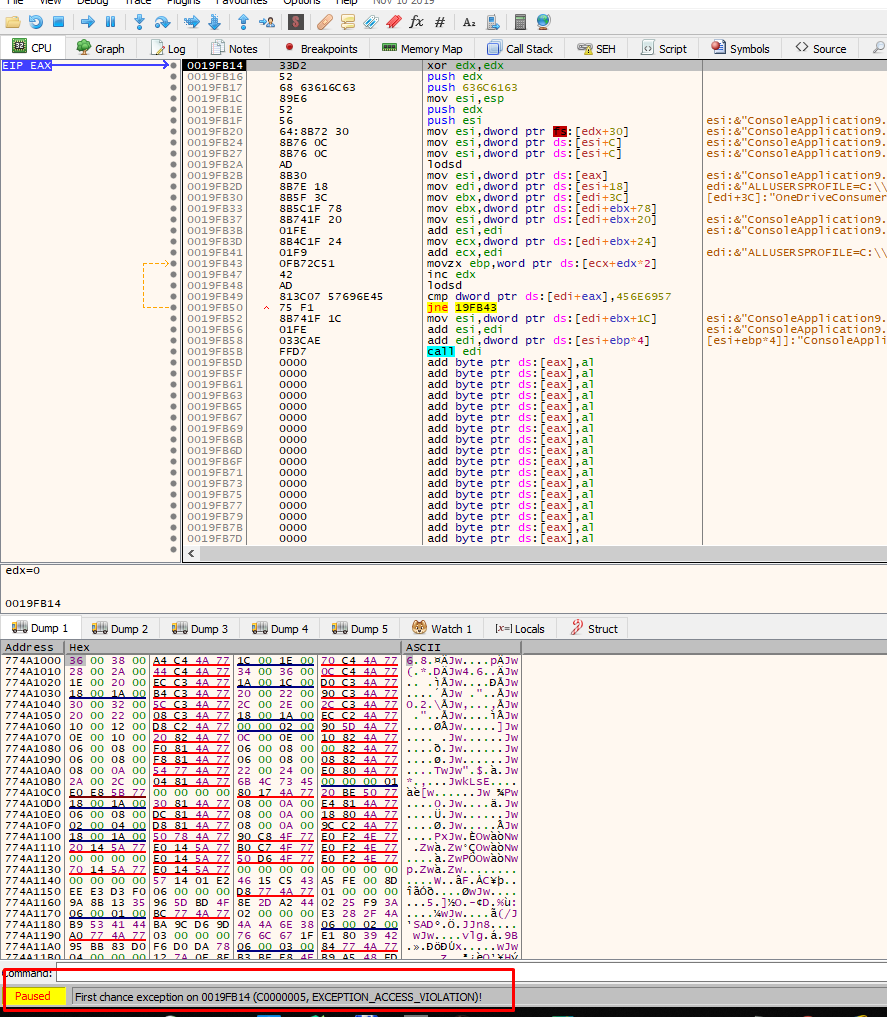


El CALL EAX se ejecutara porque es código que pertenece a la sección de código del módulo ejecutable, que tiene permiso de ejecución.

Si continuo con f7 llego al shellcode.



Si quiero continuar ejecutado mi código, como el stack no tiene permiso de ejecución, aquí crasheara.



y listo no ejecutara el shellcode y se cerrará, de esto se trata la protección DEP.

Pero como vimos el CALL EAX que es parte de la sección de código, si se ejecuto ya que como vimos las secciones de código tienen permiso de ejecución, sino el programa no podría ejecutarse.

MINIROP DE PRÁCTICA

Así que esa es la base del ROP, sí quiero en este script ejecutar un mini ROP de ejemplo que ponga EAX=0x41414141 ECX=0x42424242 EBP=0x43434343.

Puedo buscar tres gadgets

1. POP EAX-RET
2. POP ECX-RET
3. POP EBP -RET

También podría ejecutar un solo gadget que setee varios registros por ejemplo

POP ECX-POP EAX-RET

Veamos que encontramos en el código, usaremos una tool FREE para encontrar GADGETS que nos guardara en un archivo todos los gadgets que encuentre.

Se llama RP++

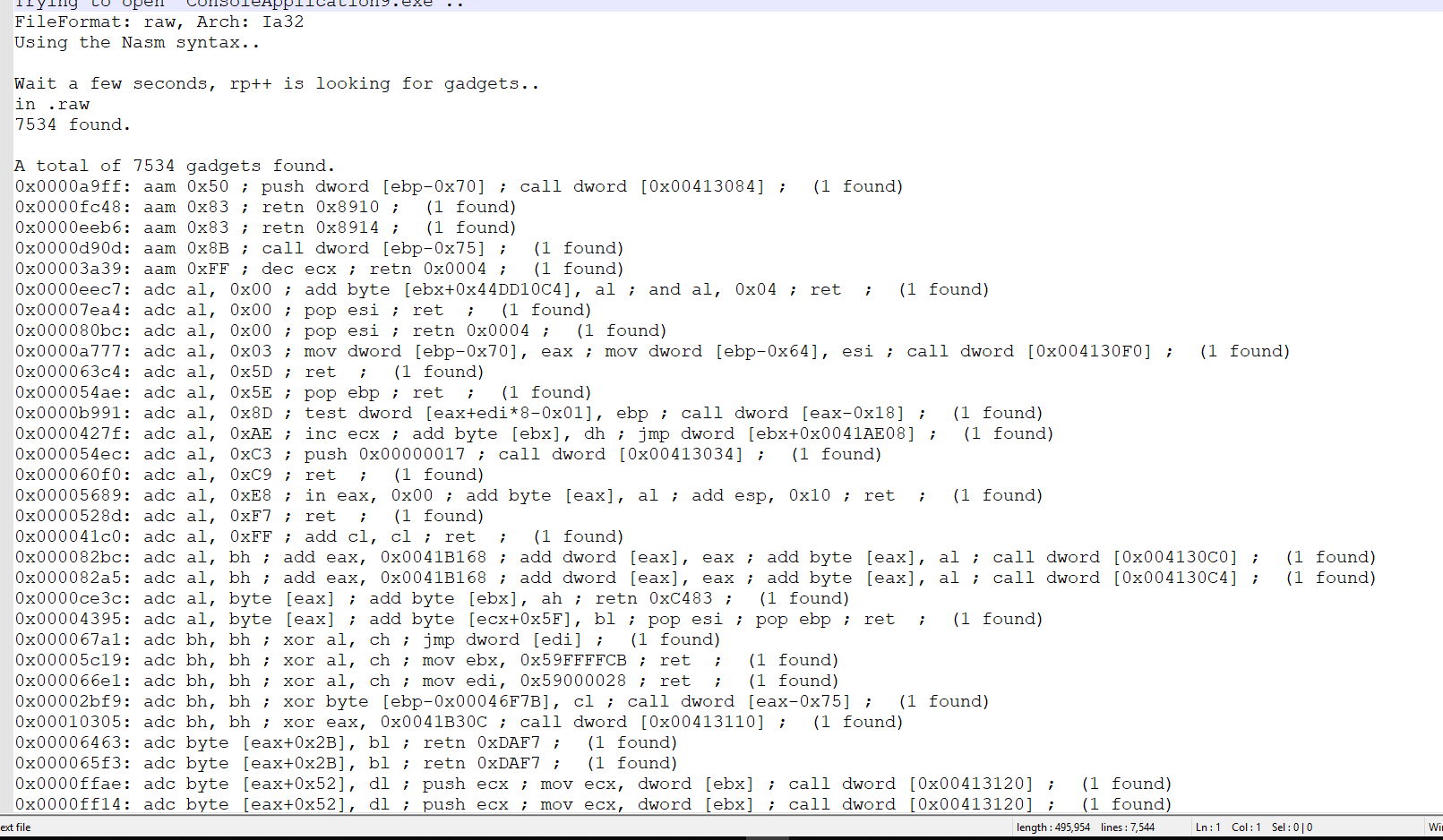
<https://drive.google.com/open?id=1M3LeiU5WzbsEqSnSwrEJupKCH_2wORnV>

Colocamos el ejecutable en la misma carpeta para hacer más fácil.

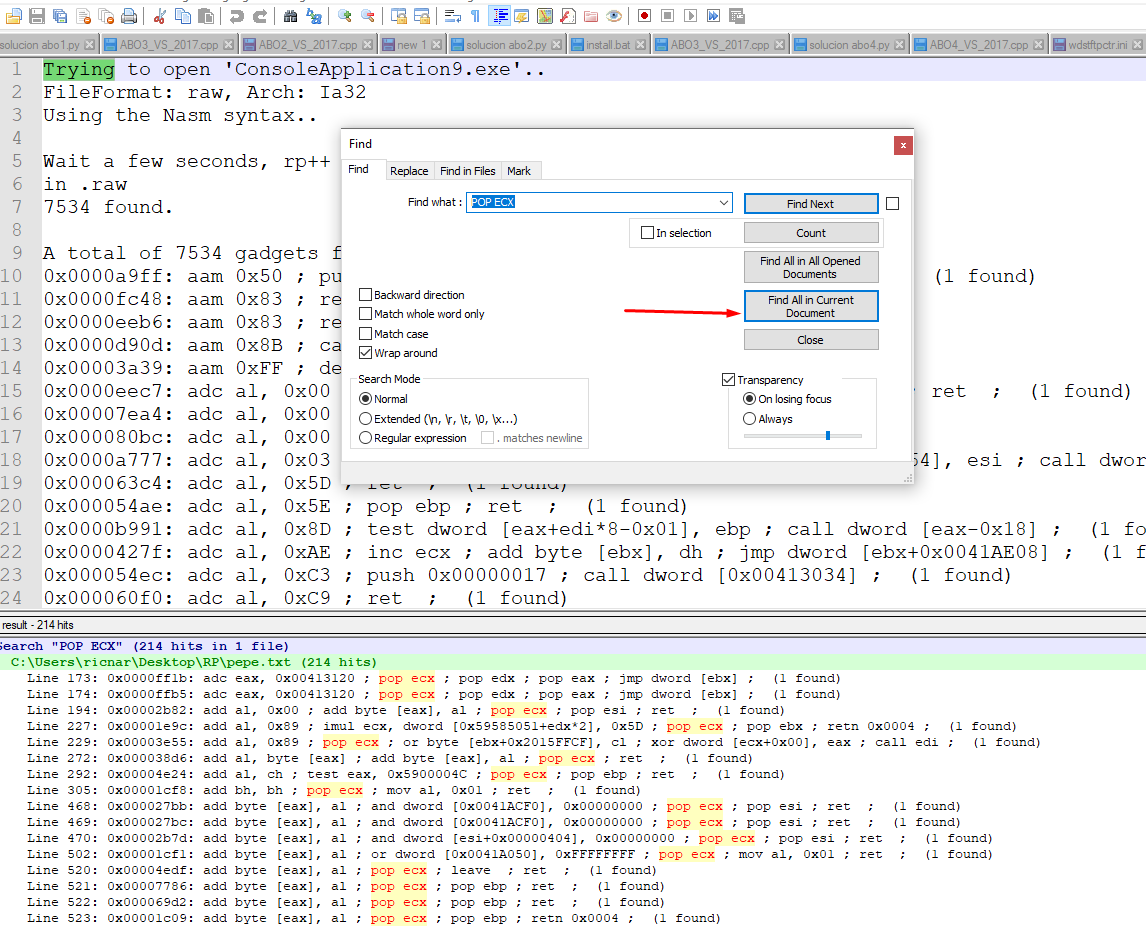


rp-win-x86.exe --file=ConsoleApplication9.exe --raw=x86 --rop=4 > pepe.txt

Uso el ejecutable para 32 bits que es **rp-win-x86.exe** y que busque código de 32 bits **raw=x86** de 4 instrucciones como máximo de largo **rop=4** lo guardo en el archivo **pepe.txt.**



Busco en el NOTEPAD++ todos los POP ECX a ver qué sale.



Ahí tengo los gadgets buscare si alguno me sirve.

0x00004828: pop eax ; pop ebp ; ret ; (1 found)

0x000033e3: pop eax ; pop ebp ; ret ; (1 found)

0x00005545: pop eax ; pop ebp ; ret ; (1 found)

0x000108d9: pop eax ; pop ebp ; ret ; (1 found)

0x00011126: pop eax ; pop ebp ; ret ; (1 found)

0x000110ad: pop eax ; pop ebp ; ret ; (1 found)

También

0x00004461: pop ecx ; pop ebp ; ret ; (1 found)

0x0000447c: pop ecx ; pop ebp ; ret ; (1 found)

0x00004c65: pop ecx ; pop ebp ; ret ; (1 found)

0x00004e2b: pop ecx ; pop ebp ; ret ; (1 found)

0x00007788: pop ecx ; pop ebp ; ret ; (1 found)

0x000069d4: pop ecx ; pop ebp ; ret ; (1 found)

0x00007648: pop ecx ; pop ebp ; ret ; (1 found)

0x00009646: pop ecx ; pop ebp ; ret ; (1 found)

Y

0x00000718: pop ecx ; ret ; (1 found)

0x000005dd: pop ecx ; ret ; (1 found)

0x00001820: pop ecx ; ret ; (1 found)

0x00002849: pop ecx ; ret ; (1 found)

0x00003d06: pop ecx ; ret ; (1 found)

0x000038da: pop ecx ; ret ; (1 found)

0x000044b9: pop ecx ; ret ; (1 found)

0x00004592: pop ecx ; ret ; (1 found)

0x000048c7: pop ecx ; ret ; (1 found)

0x000049f2: pop ecx ; ret ; (1 found)

0x00004a4d: pop ecx ; ret ; (1 found)

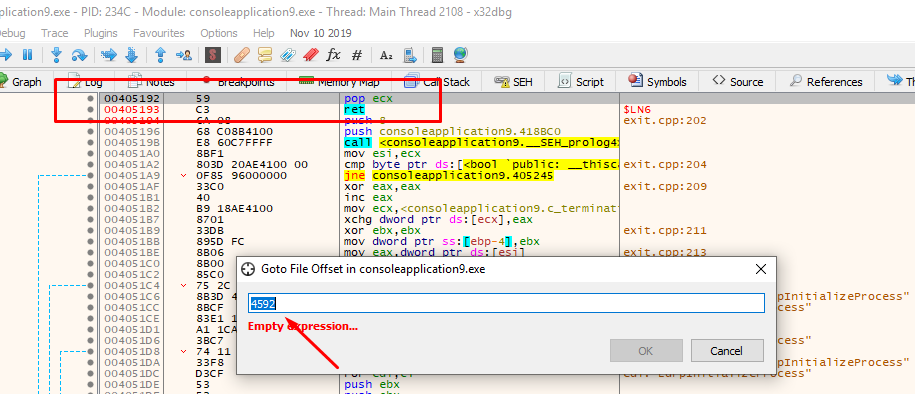
0x00004d55: pop ecx ; ret ; (1 found)

0x00004dfc: pop ecx ; ret ; (1 found)

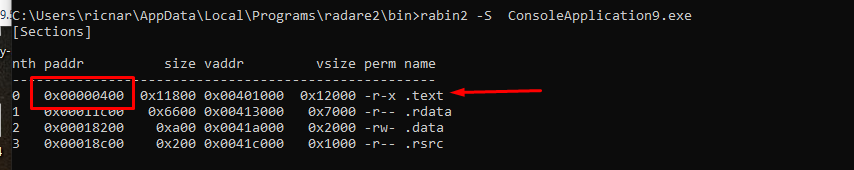
Bueno podemos armarlo con esos gadgets para probar, el tema es que está tool te devuelve los file offset ya que abre el archivo estáticamente.

En partes anteriores ya vimos como calcular la dirección de memoria virtual desde el file offset.

Si lo quieren hacer rápido con el x64dbg van a GOTO-FILE OFFSET ponen la dirección que les da el file offset del gadget y obtienen la dirección virtual.

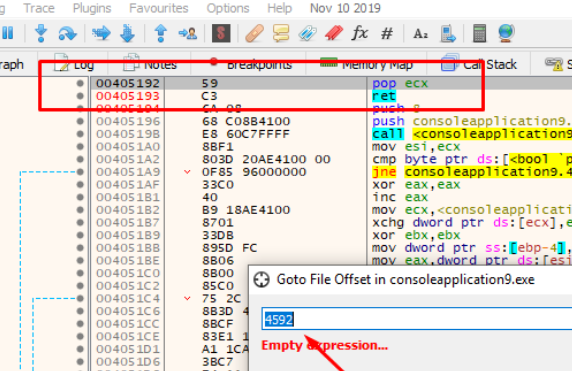


Si lo quiero calcular, vemos con rabin2 que la sección de código empieza en 0x400 en disco, así que restando al valor del file offset - 0x400 nos da el desplazamiento dentro de la primera sección en el disco y sumándole donde comienza la sección de código en memoria (la imagebase más el size del header 0x401000) nos debería dar correctamente la dirección virtual.



0x4592- 0x400 + 0x401000= 0x405192





Así que el script para hacer ese rop seria.

Como quiero que mi ROP setee

**EAX=0x41414141 ECX=42424242 EBP=43434343**

Usare este otro gadget también.

0x00005545: pop eax ; pop ebp ; ret ; (1 found)

dirección virtual = 0x406145

hex(0x5545- 0x400 + 0x401000)

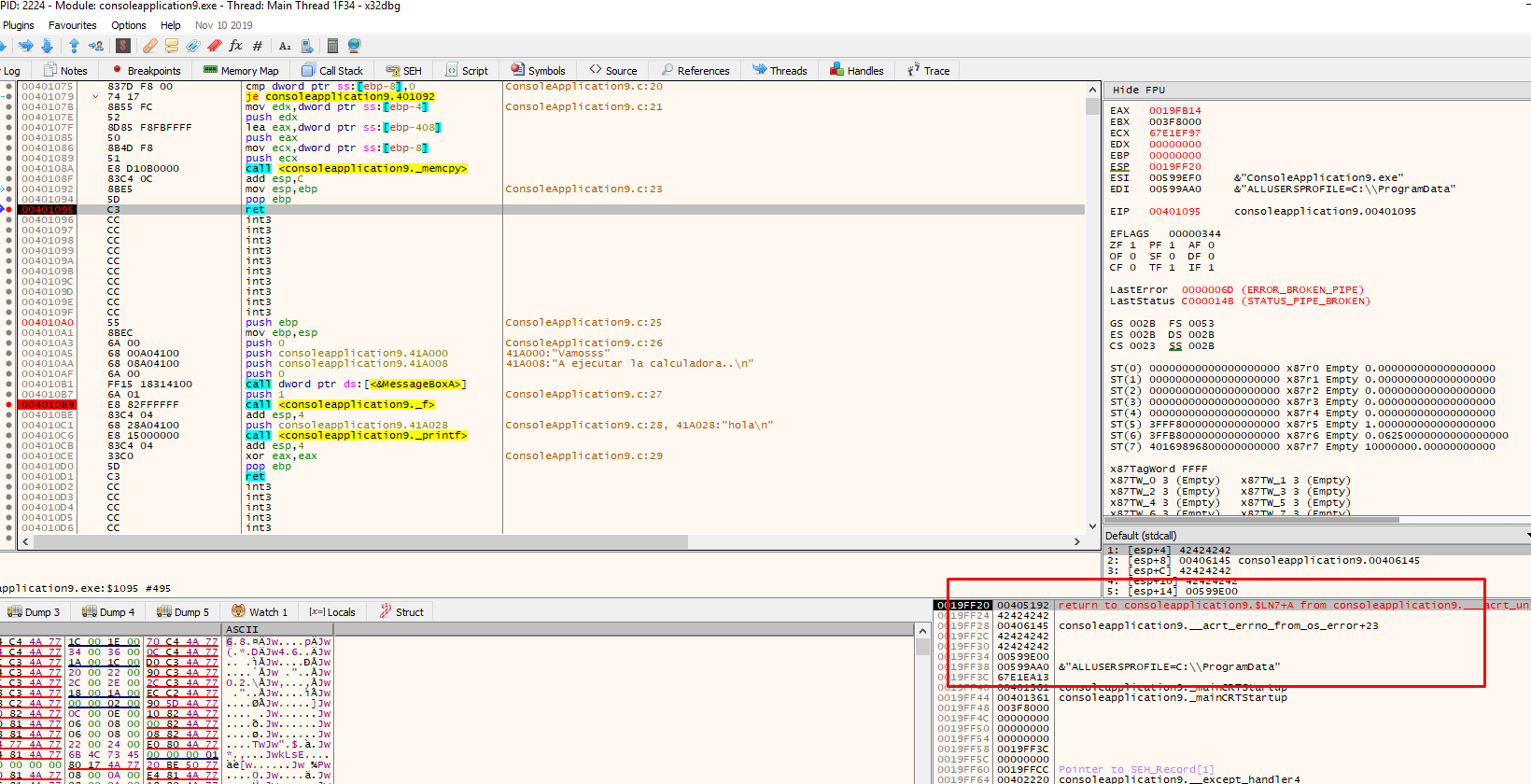
'0x406145'



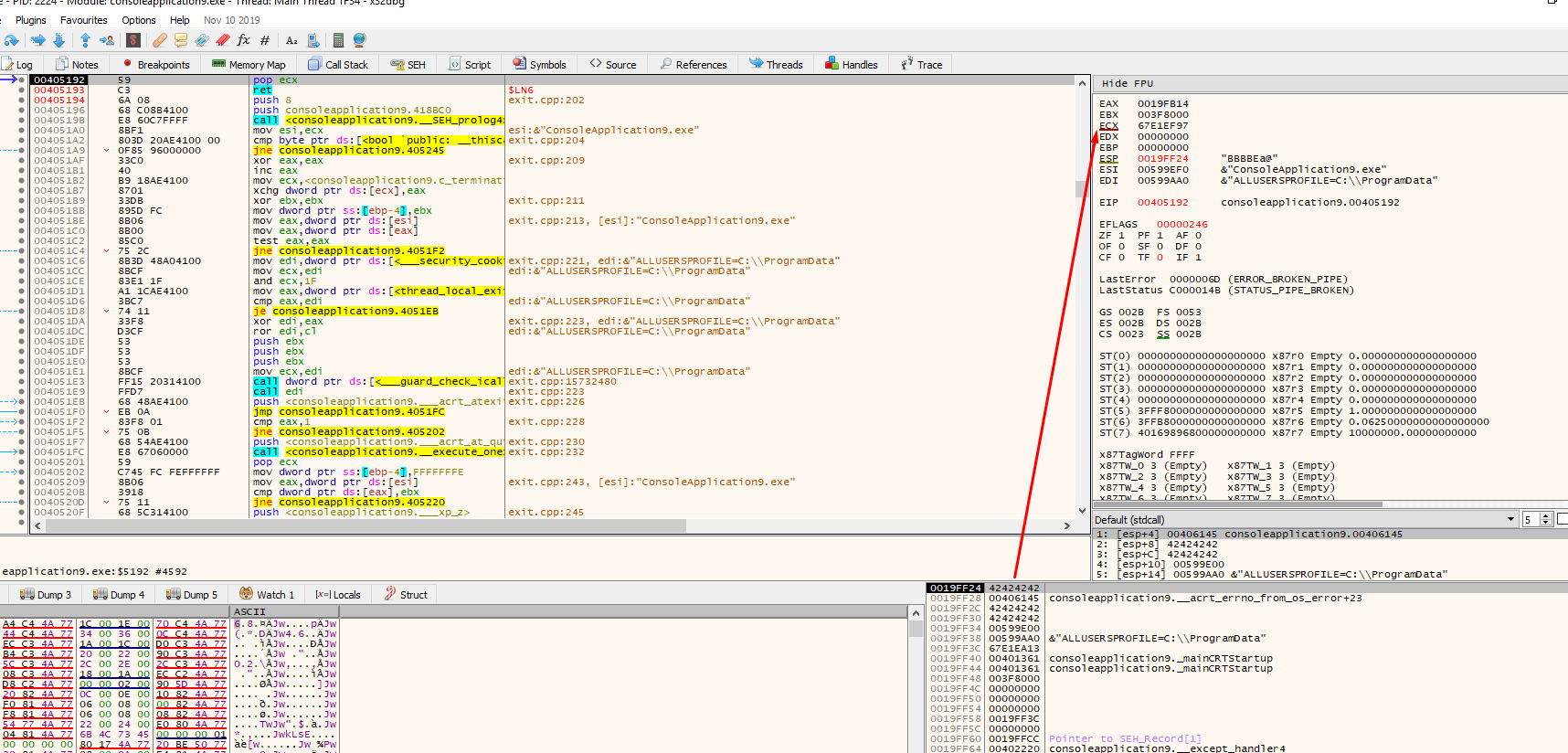
Alli vemos los dos gadgets y intercalados los valores que leerá con los POP y moverá a los registros.

Corramoslo a ver si obtuvimos lo que queríamos.

Cuando llego al RET veo mi ROP lo voy traceando con f7.

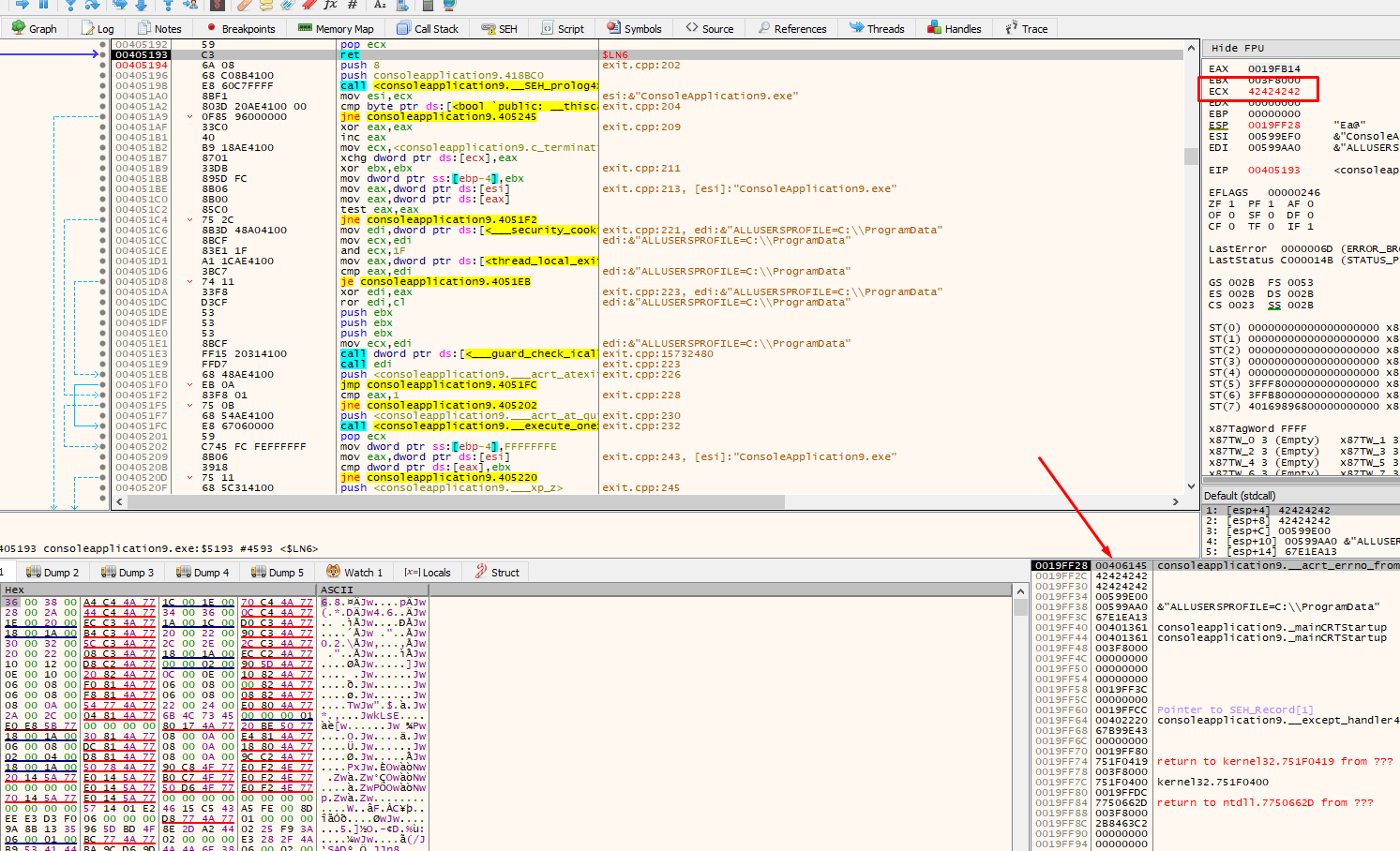


el primer gadget es POP ECX-RET

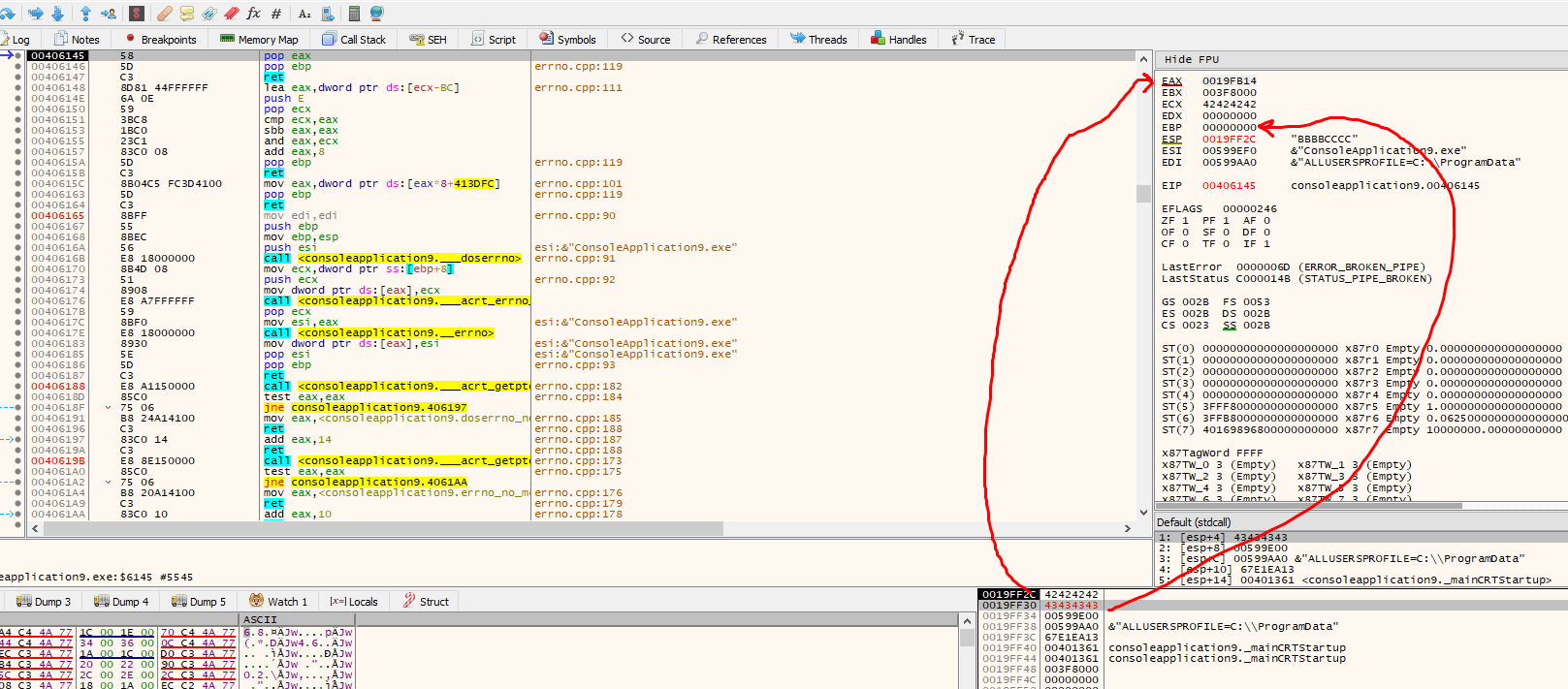


Mueve con el POP ECX el 0x42424242 a ECX y al llegar al ret.

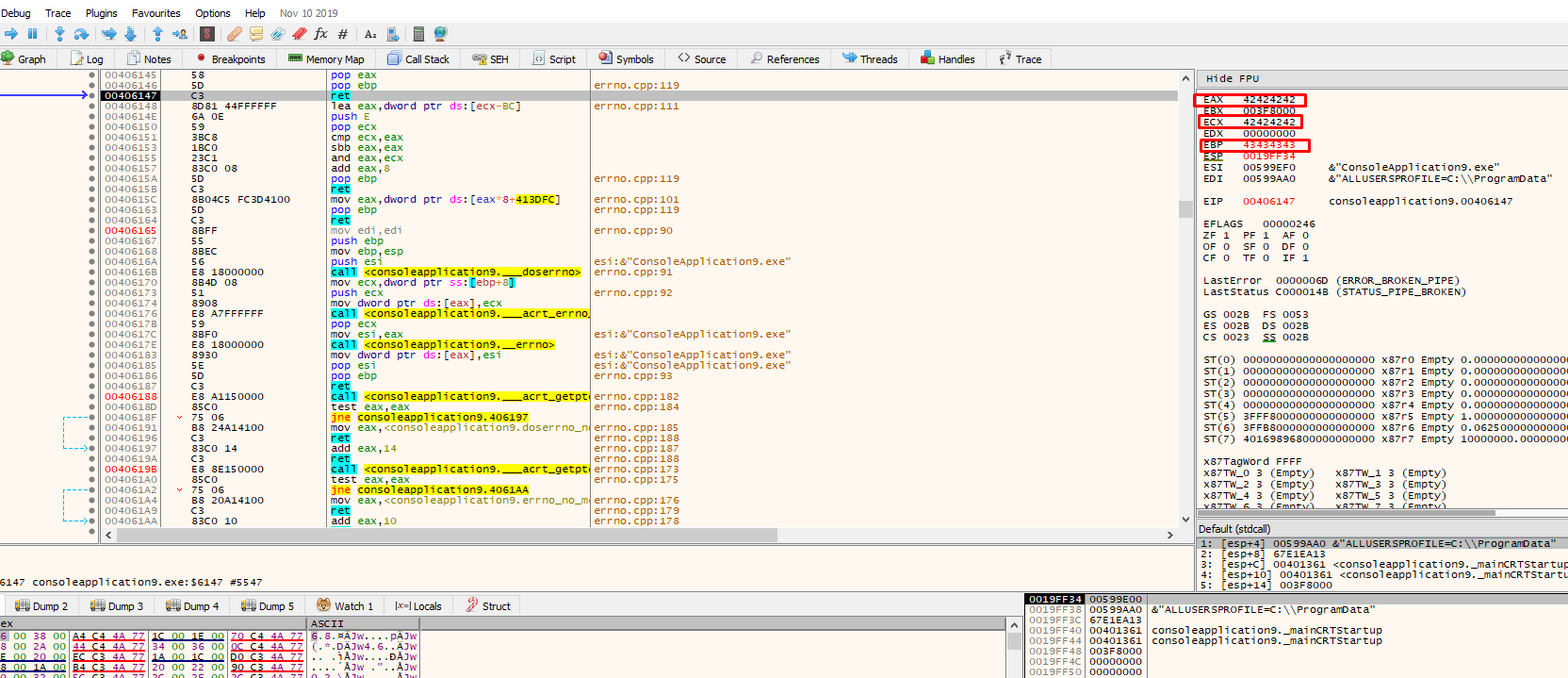
ECX queda en 0x42424242 como queríamos y ahora volverá a saltar al llegar al RET



Salta al segundo GADGET.



Moverá 0x42424242 a EAX y 0x43434343 a EBP.



Vemos que logramos encadenar varios GADGETS y colocar los valores que queríamos donde necesitábamos.

Y si quisiéramos podriamos continuar colocando más GADGETS debajo y hacer diferentes cosas.

En la parte 11 veremos el ROP completo para bypasear el DEP para este ejercicio creo que con esto por hoy ya fue suficiente.

Hasta la parte 11.

Ricardo Narvaja

06/03/2020