EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

GRATUITAS (PARTE 20)

Seguimos con el tema estructuras.

En la parte 19 anterior comparamos dos ejercicios el SIN ESTRUCTURAS

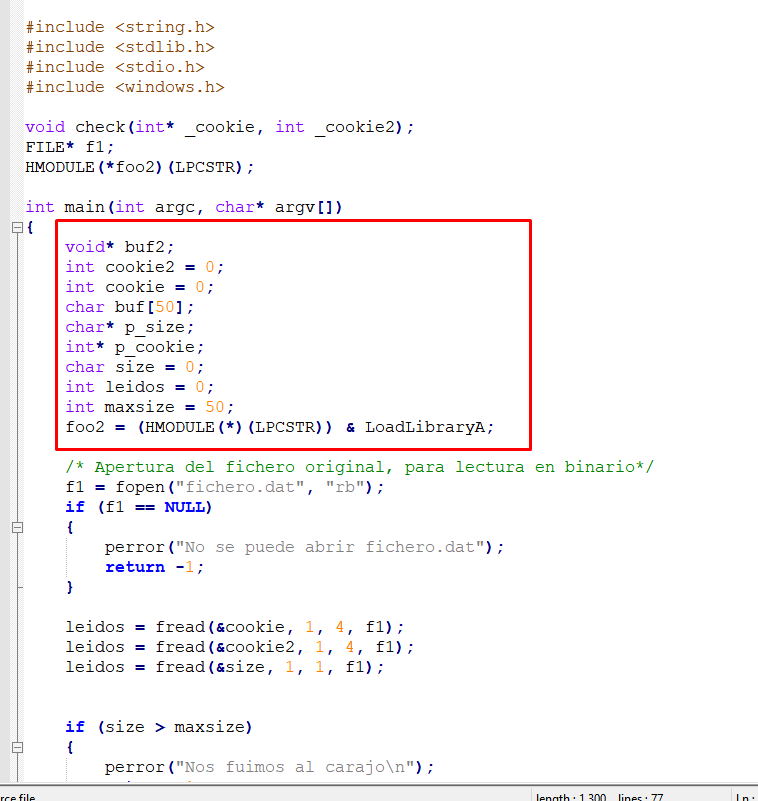
<https://drive.google.com/file/d/10p5i8gDHLUgz9Xag_1XadpHBCyNAPisB/view?usp=sharing>

password =a

Y el CON ESTRUCTURAS

<https://drive.google.com/file/d/1i_PKAY5ustTHjPdoM75ffwfCmXH9cDb9/view?usp=sharing>

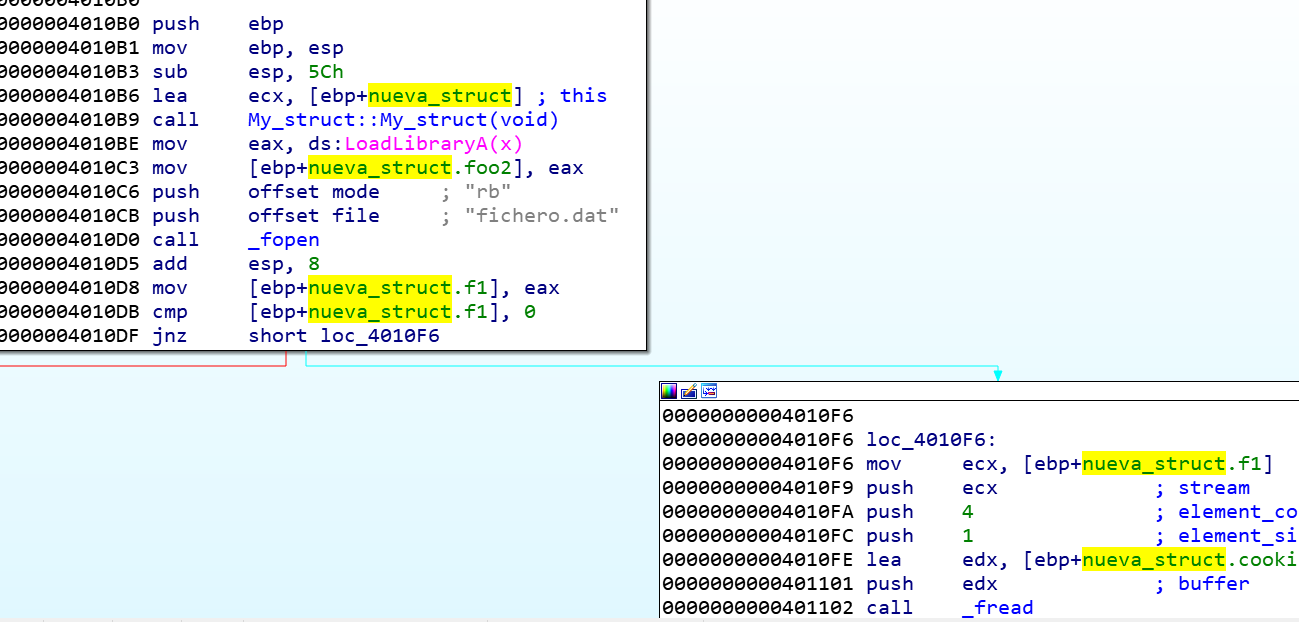
habíamos visto ya el SIN ESTRUCTURAS que tenia todas las variables locales separadas



Lo reverseamos y luego estaba el CON ESTRUCTURAS, ambos tienen su código fuente adjuntado.

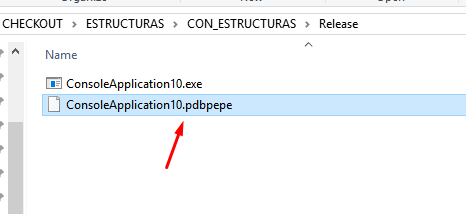


También tienen sus simbolos por lo cual, si uno lo carga en IDA, después de cambiar la opción de DEMANGLE NAMES -NAMES se ve lindo.

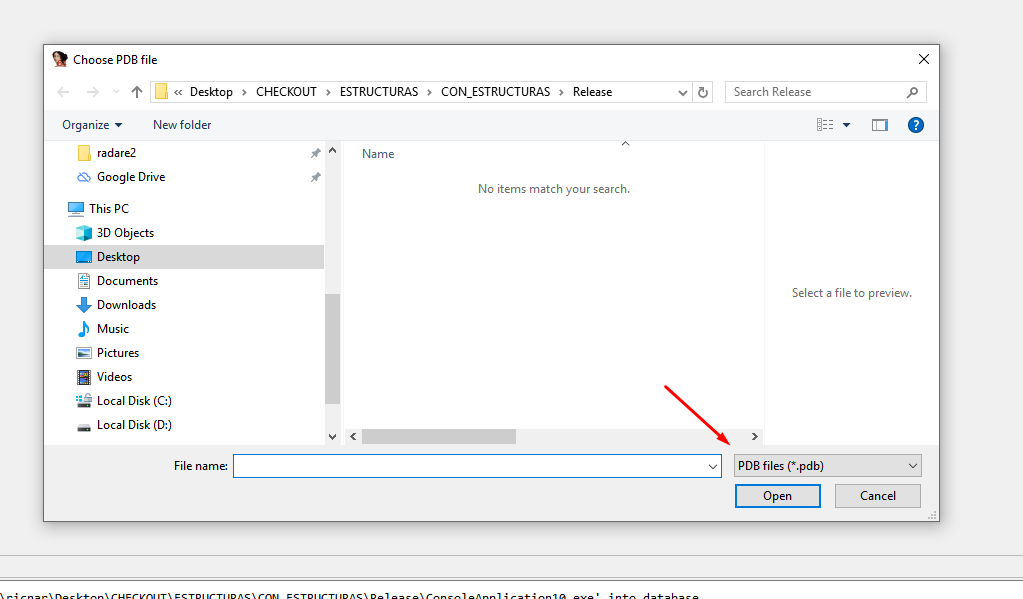


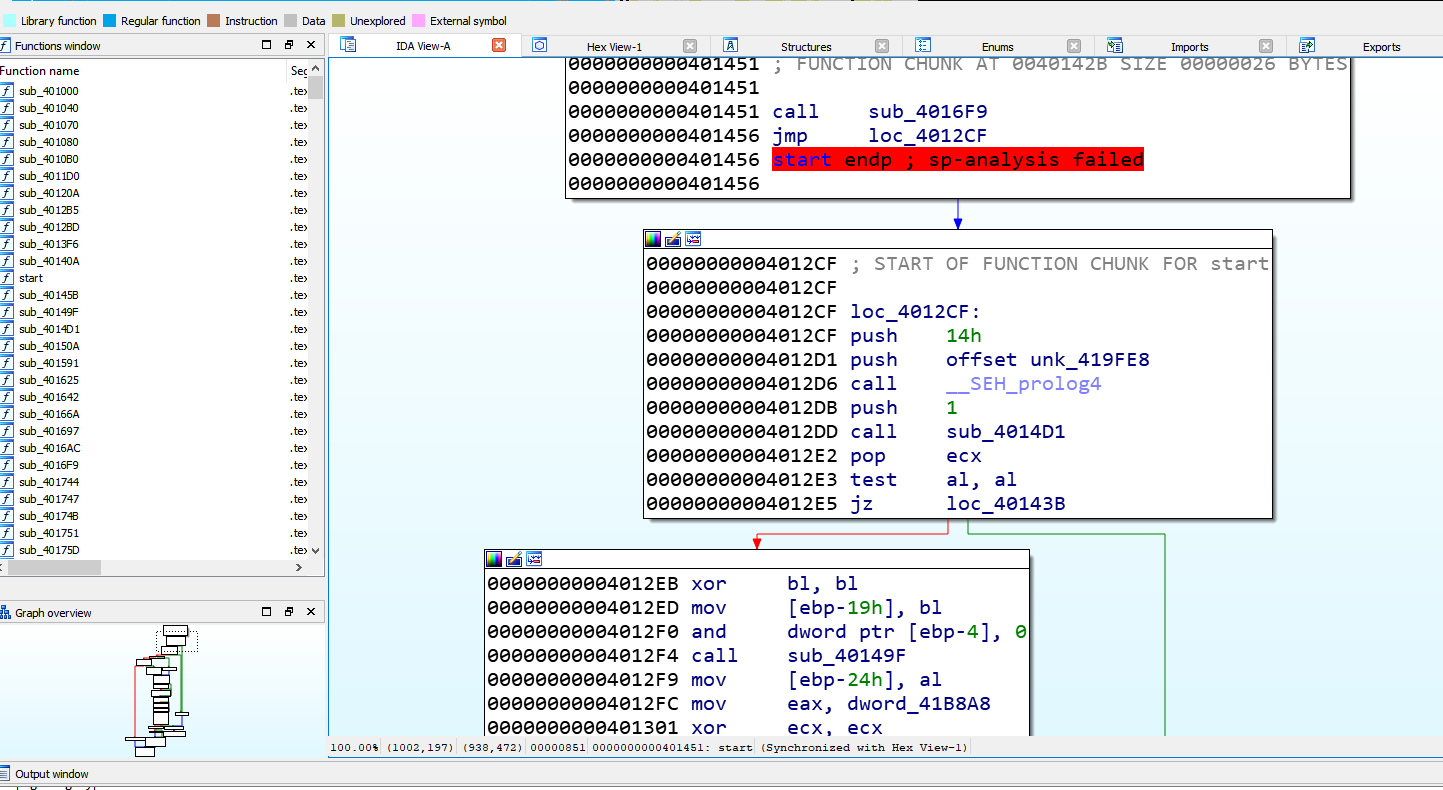
Pero en la realidad nosotros no vamos a tener siempre los simbolos o casi nunca así que trataremos de reversearlo sin simbolos.

Vamos a la carpeta donde esta el ejecutable y borramos el i64 que es la database para que lo analice de nuevo y además renombramos el archivo pdb.

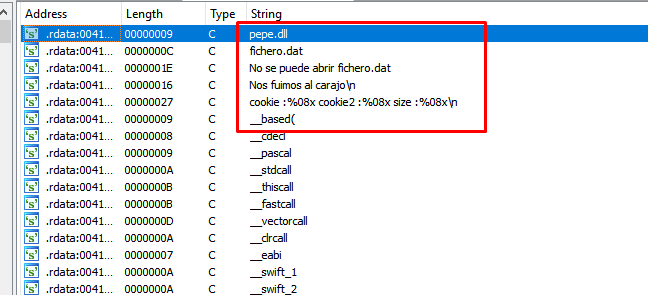


Nos damos cuenta de que ahora nos pide los simbolos, así que es obvio que analizara sin los mismos, apretamos CANCEL y seguirá analizando.

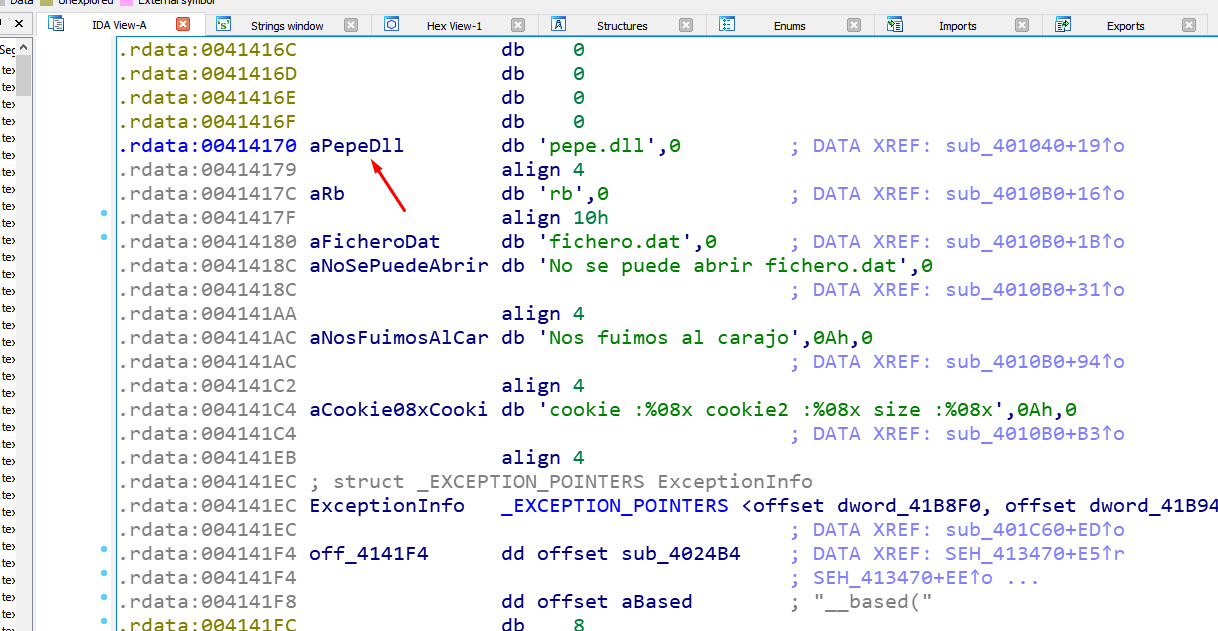




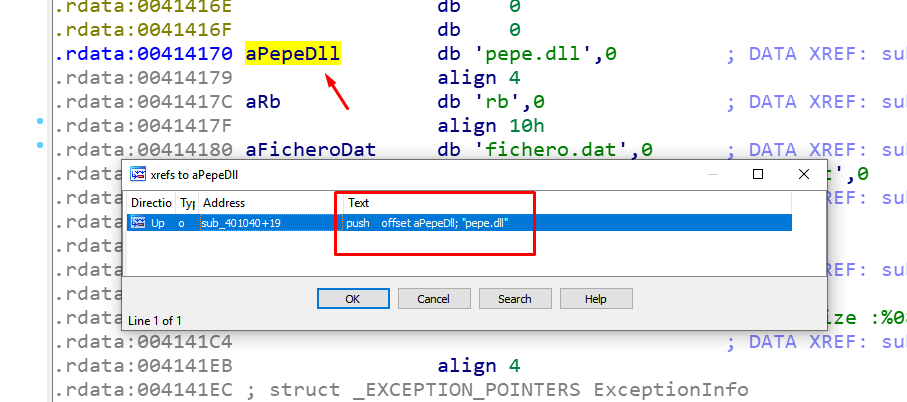
Ahora si que no tenemos casi nada de info, veamos las strings.



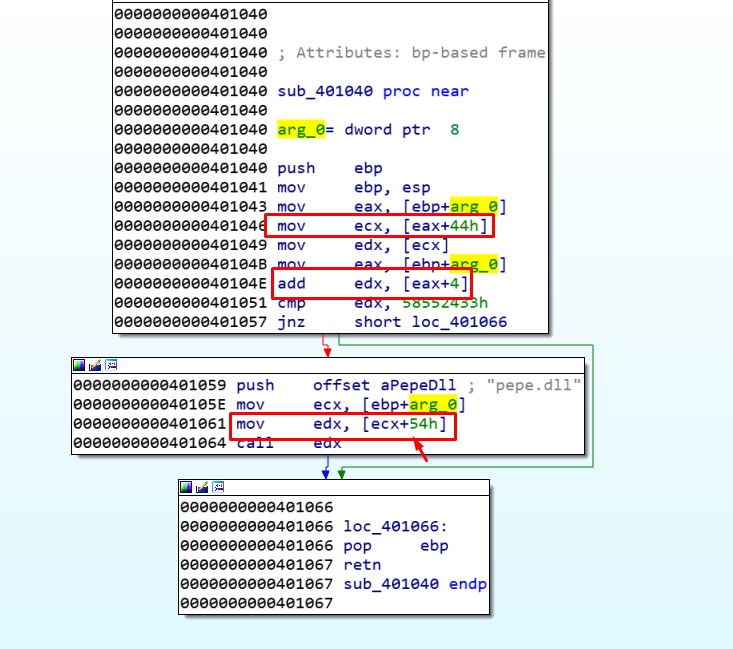
Vamos a donde son usadas dichas strings, por ejemplo pepe.dll.



Ahí apreto X y voy a.



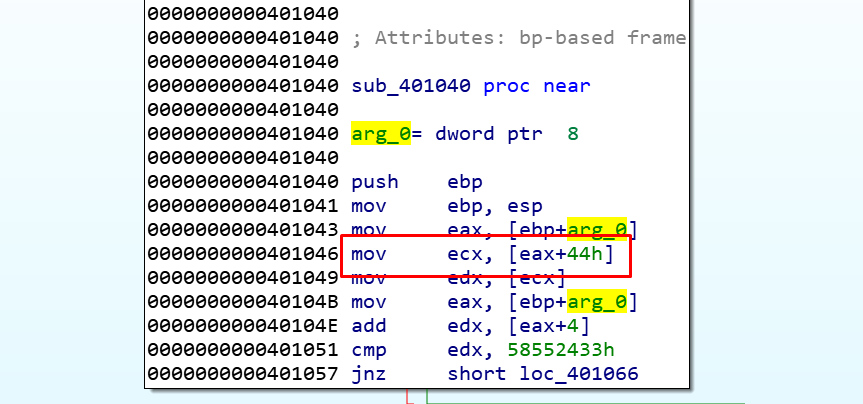
Vamos allí.



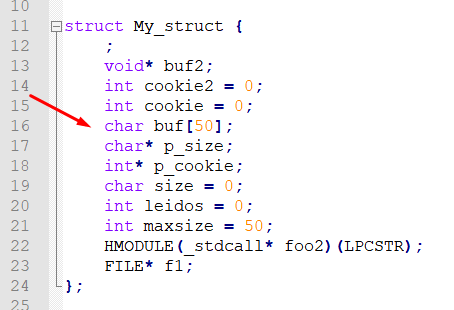
Ahí vemos la respuesta a una de las preguntas que siempre me hacen

¿Como sabemos que algo es una estructura?

Allí se ve, si desde una variable se pasa un valor a un registro y luego se accede a ese valor mas un offset, eso es una estructura.



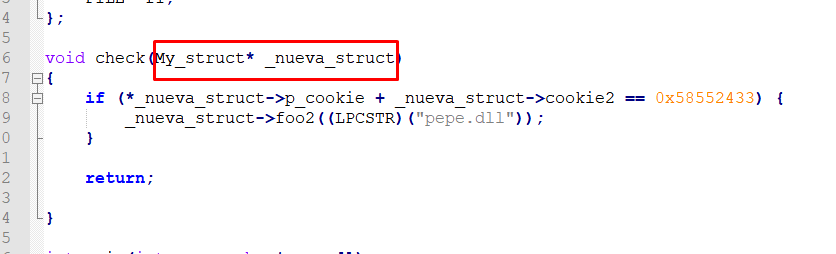
Allí, por ejemplo, si yo me pregunto ¿Qué tipo de variable seria **arg\_0** que se lee su valor a ECX y después a ese valor se le suma en este caso 0x44 para acceder a otro valor?



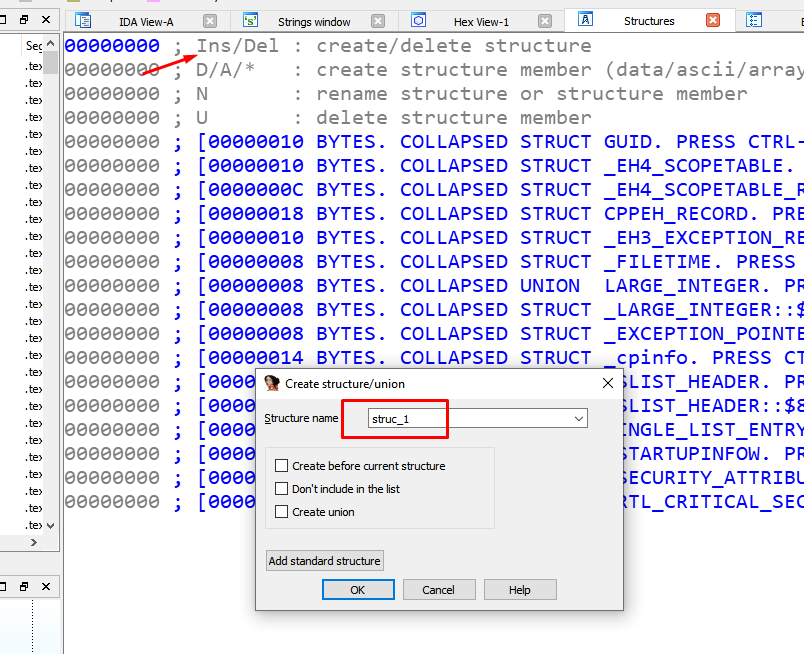
Obvio que una estructura se maneja a través de una variable puntero que guarda la dirección de inicio de esta, como en este caso **arg\_0** es un puntero a la dirección de inicio de la estructura.

**My\_struct \* arg0;**

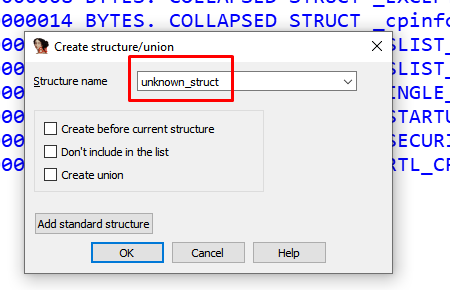
Si miramos el código fuente es así, pero como no lo tenemos solo sabemos que es un puntero a una estructura desconocida

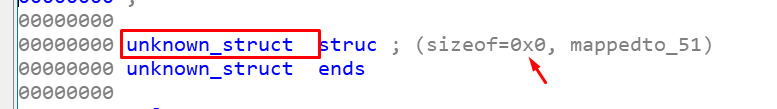


Por lo tanto, en IDA creare una nueva estructura para ir determinando los campos de la misma reverseando, en la pestaña STRUCTURES, apreto la tecla INS.

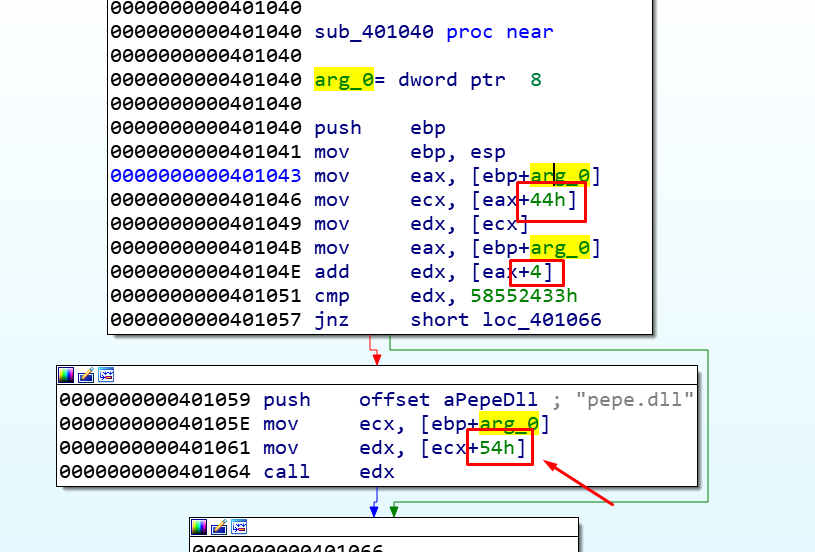


Y le pongo un nombre, por ahora como no se cómo se llama ni para que sirve le pondré **unknown\_struct**.

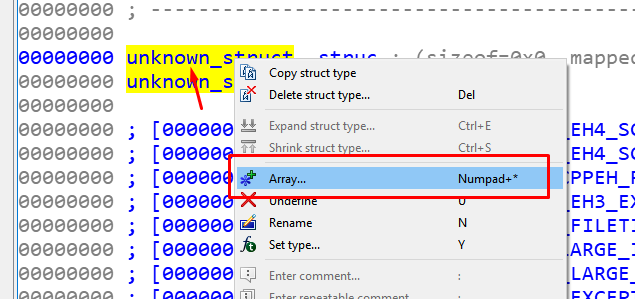




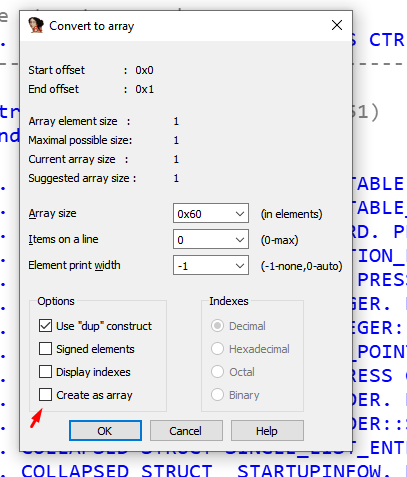
Ahora veremos el size, no importa que sea exacto, después podemos agregar o achicar si es necesario, incluso si es mas grande no importa tampoco, lo que se hace es mirar el offset del campo más grande que veo que accede.



Vemos que el offset más grande que accede aquí es 0x54, pondré el largo 0x60 por si acaso, después veré.

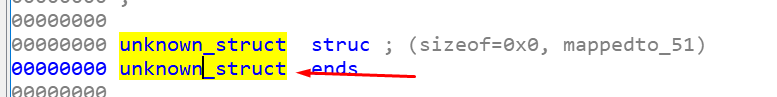


Si no me molesta que los campos sean todos de 1 byte puedo hacer click derecho ARRAY, poner el size 0x60.

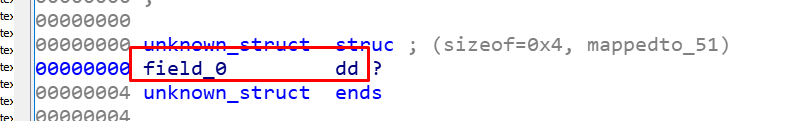


Quitándole la tilde a CREATE AS ARRAY y creara una estructura con todos campos del tipo byte, igual después iré viendo de que tipo es cada uno y corrigiendo.

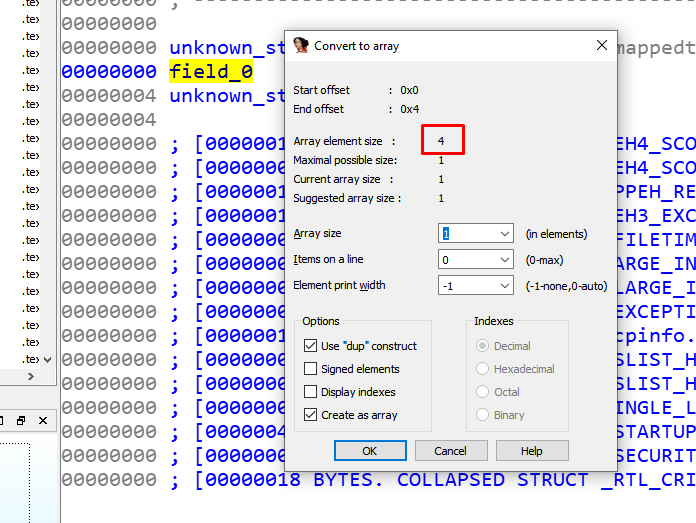
Si sé que la mayoría serán DWORDS, hare así, primero agregare un solo DWORD.



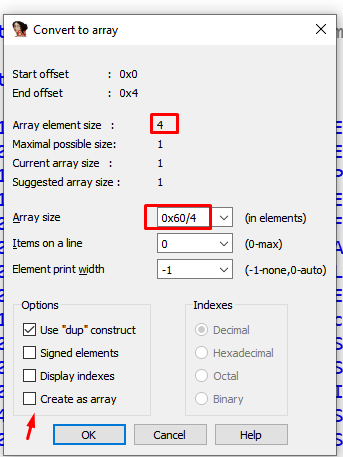
Me paro donde termina la estructura y apreto la tecla d varias veces sin moverme de allí, hasta que va cambiando el tipo y me queda un DWORD.



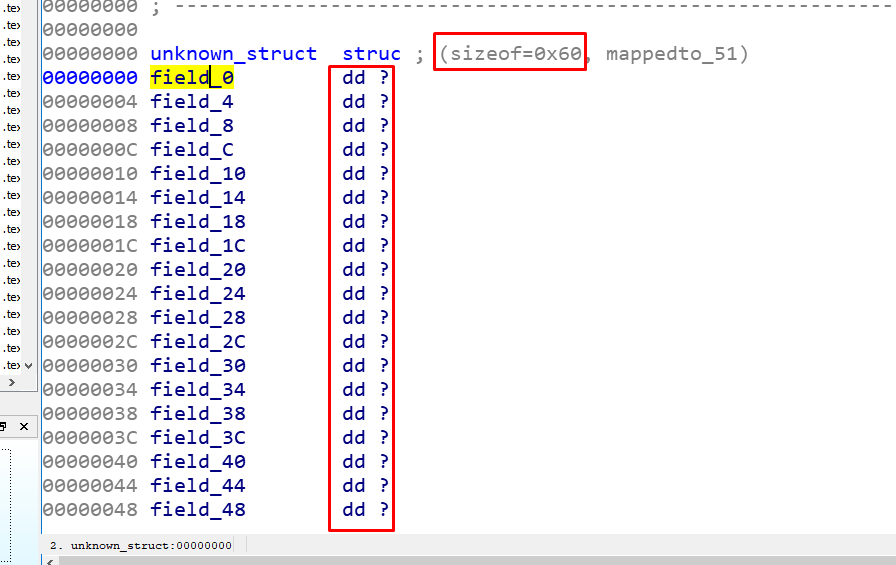
Ahora si hago un array allí como antes, debo quitar la tilde CREATE AS ARRAY.



Me toma que el ELEMENT SIZE será 4 o sea DWORD, así que si quiero que tenga largo 0x60 deberé poner en el largo 0x60/4.

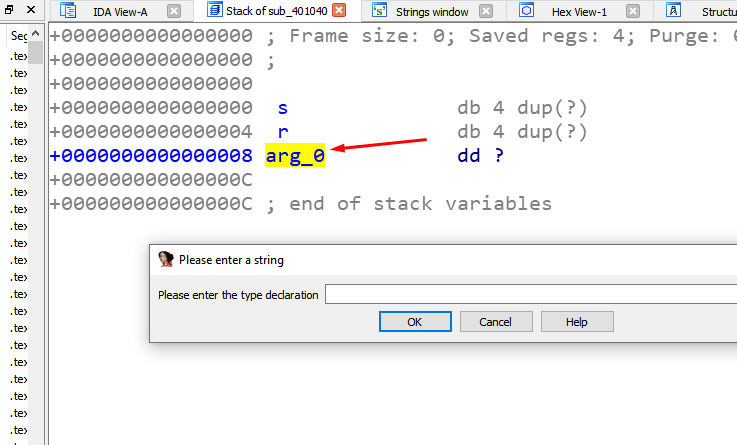


Y ahora si quedo bien.

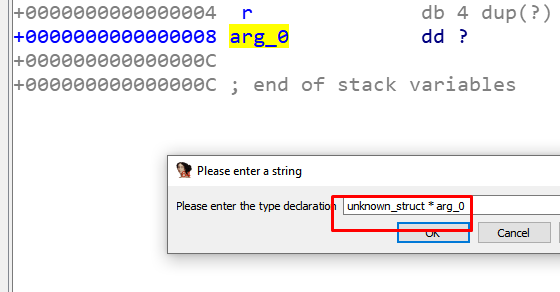


Bueno ya tengo la estructura base.

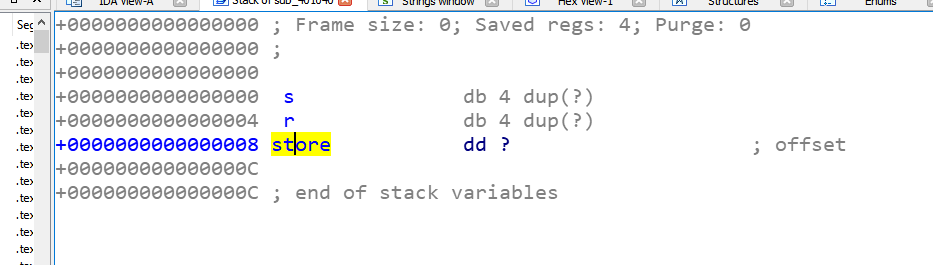
Nos falta cambiar el tipo de **arg\_0** a puntero a esa estructura.

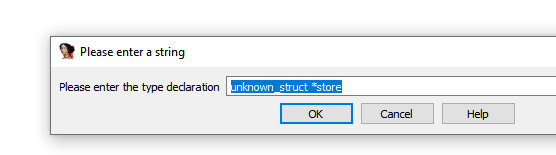


Vamos a la representación estática del stack y hacemos click derecho- SET TYPE y ponemos.



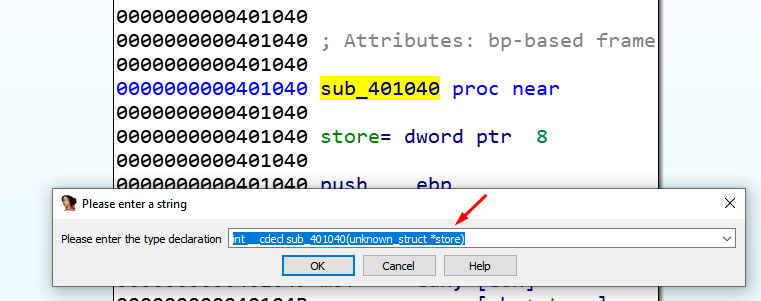
Y después con rename le podemos poner a la variable un nombre como por ejemplo store.



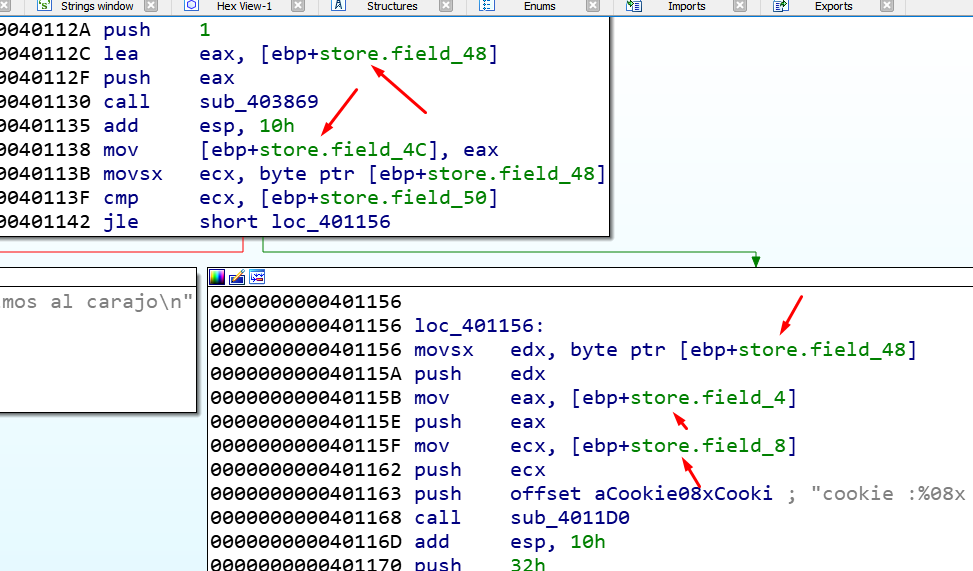


Podemos chequear el tipo y vemos que quedo bien.

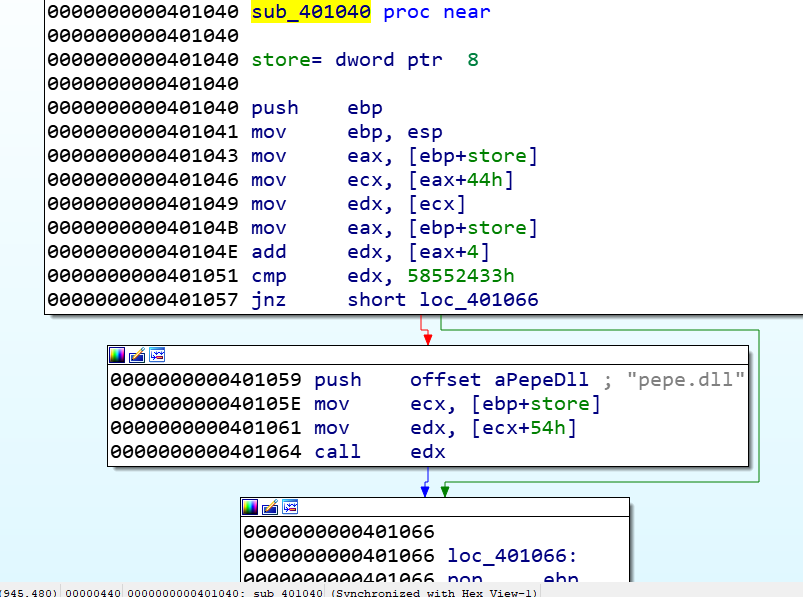
Puedo hacer SET TYPE en la función para propagar el tipo.



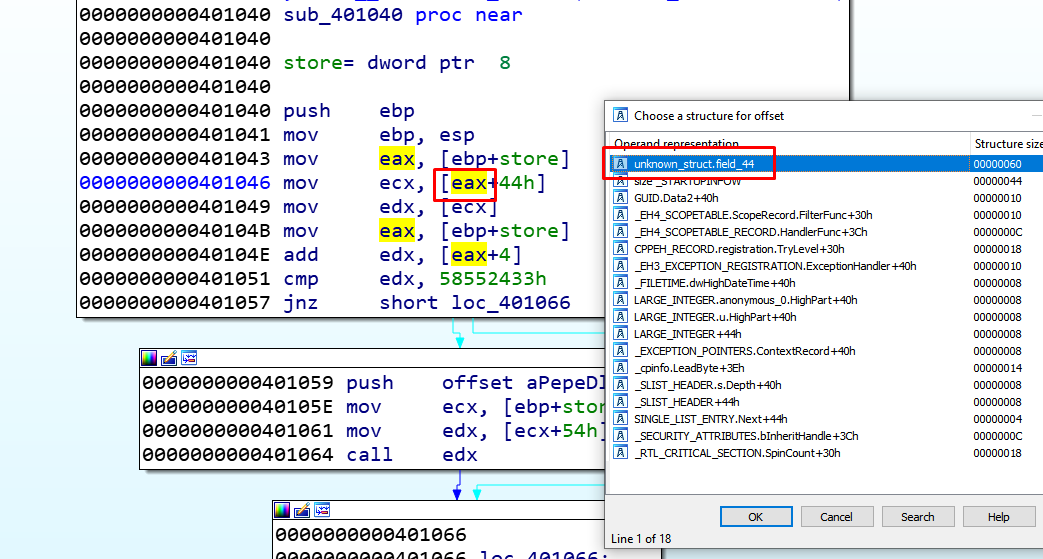
Vemos que al cambiar a que es del tipo puntero a estructura que ya definí y propagar en la función principal ahora detecta los campos, eso ayuda muchísimo.



Vemos que se propaga a la función main, pero en nuestra misma función no



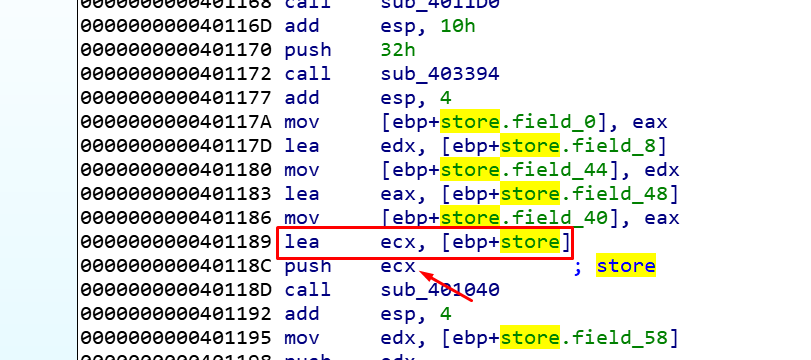
Apretamos T en cada uno de los offset y elegimos la estructura creada.



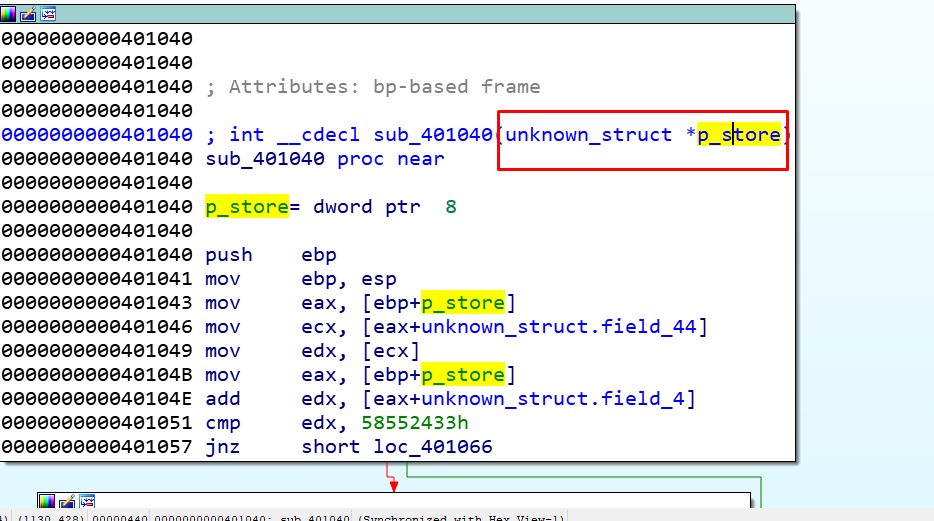
Cuando termino queda así.



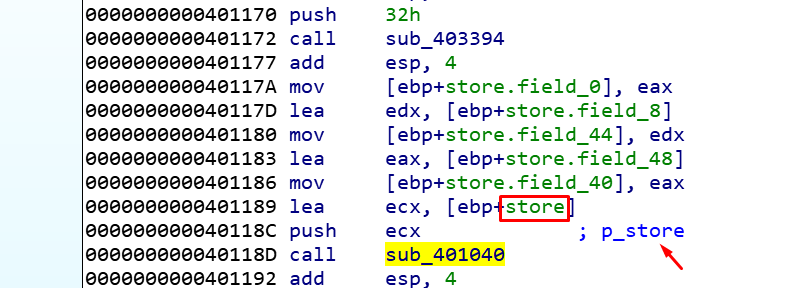
Bueno ya quedo mejor, por ahora dejemos esta función y volvamos al main.



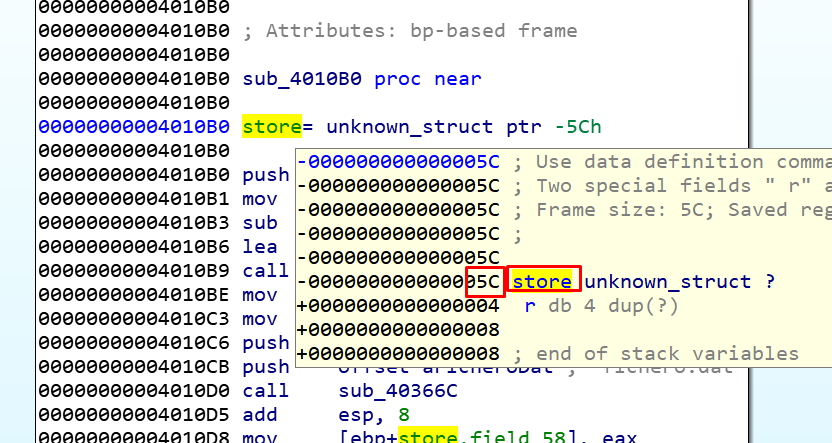
Allí llama a la función pasándole la dirección de store usando LEA, por lo tanto, cuando pushea ECX esta tiene la dirección donde esta ubicada la estructura, y por supuesto la variable que recibe esto como vimos era **arg\_0** como realmente no es **store** sino un puntero a store le podemos cambiar dentro a **p\_store**.



Ahora si quedo perfecto **arg\_0** es un puntero a un estructura por eso mejor llamarla **p\_store** ya que es una variable puntero las cuales como sabemos guardan direcciones de variables o argumentos, en este caso guarda la dirección de **store**.



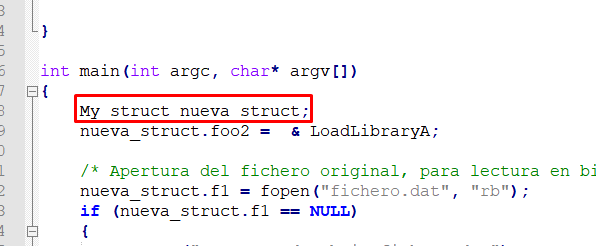
Ahora quedo perfecto, pero donde esta definida **store** realmente, veamos.



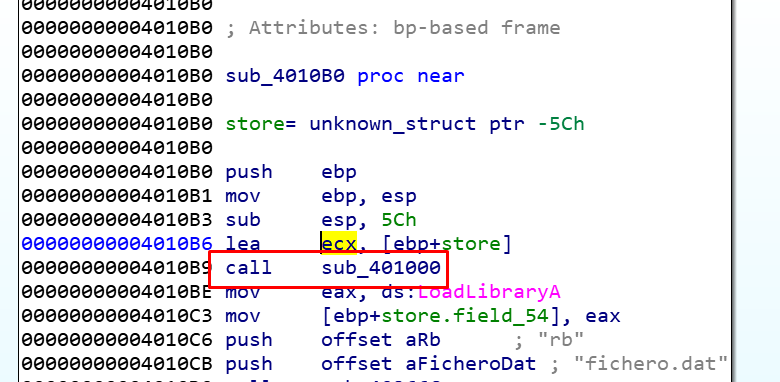
Allí vemos que es una variable local de la función principal y que el largo es exactamente **0x5c**, siendo más chica que los **0x60** que yo le asigne y por supuesto es una variable ya que esta arriba del return address.

Así que a bajo nivel esta estructura esta creada en el stack, como un buffer en el mismo stack.

Si chequeamos con el código fuente, vemos que es cierto ya que también podría crearse reservando memoria allocando en el heap, no es este el caso.

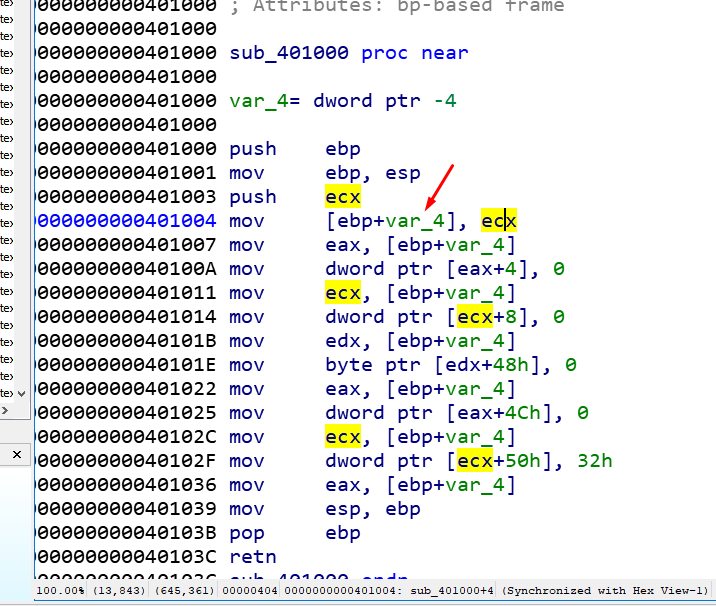


Ahora que ya tenemos esto es casi como coser y cantar jeje, bueno no tanto aun no tenemos simbolos, pero ya dimos un gran paso.



Vemos un constructor o inicializador de la estructura, al cual le pasa también **p\_store** usando LEA.

Lo pasa el argumento por registro y lo guarda en **var\_4.**



Si queremos definir una función que pasa argumentos por registro podemos usar **usercall** que es la forma que tiene IDA para representar calling conventions no standard.

Ejemplo de una función que usa **ecx** y una argumento en el stack y devuelve el resultado en **eax**.

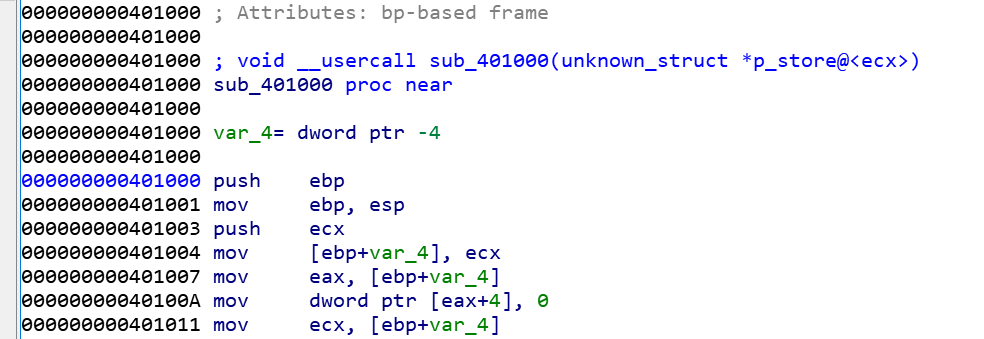
**int \_\_usercall NOMBRE\_FUNCION@<eax>(int pepe, int pipo@<ecx>)**

En nuestro caso seria.

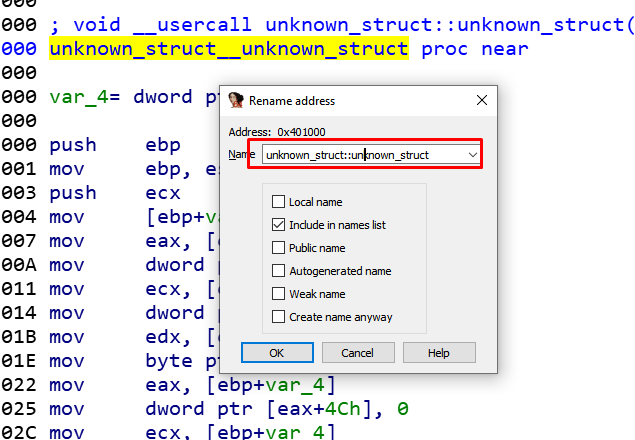
**int \_\_usercall NOMBRE\_FUNCION@<eax>(unknown\_struct \* p\_store@<ecx>)**

Aunque como no devuelve nada en **eax** le puedo quitar el valor de retorno y ponerle **void** en vez de **int**.

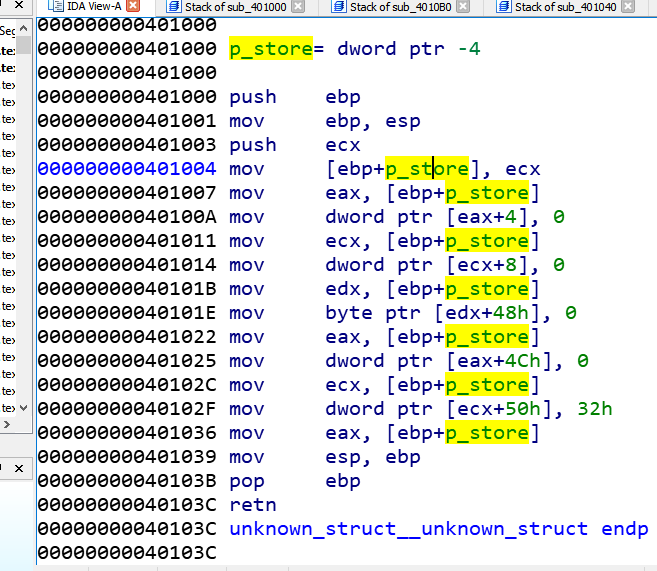
**void \_\_usercall sub\_401000(unknown\_struct \*p\_store@<ecx>);**



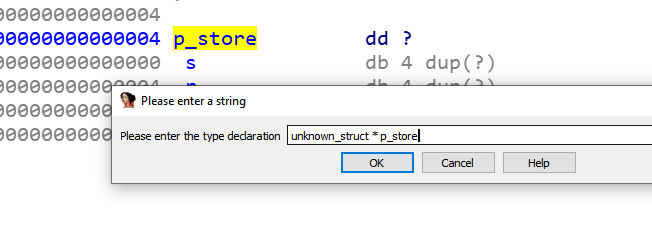
Y podemos renombrar la función como constructor de la estructura como si fuera una clase ya que a balo nivel se parecen mucho.



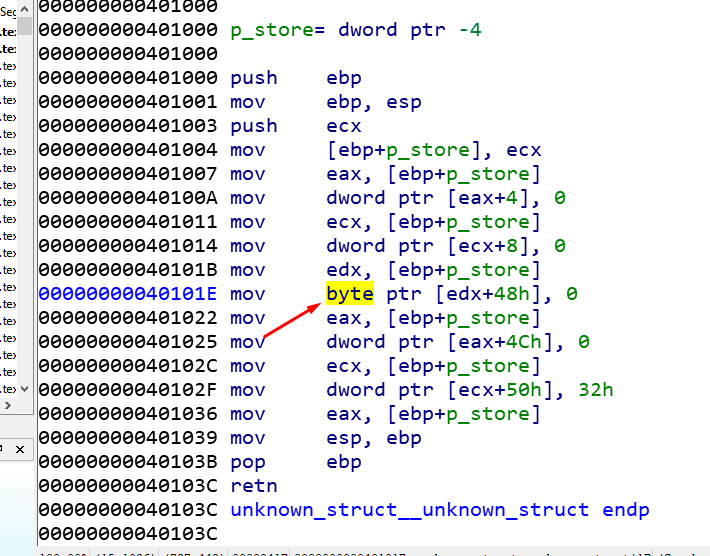
Cambia los doble dos puntos **“::”** por un guion bajo no hay problema.



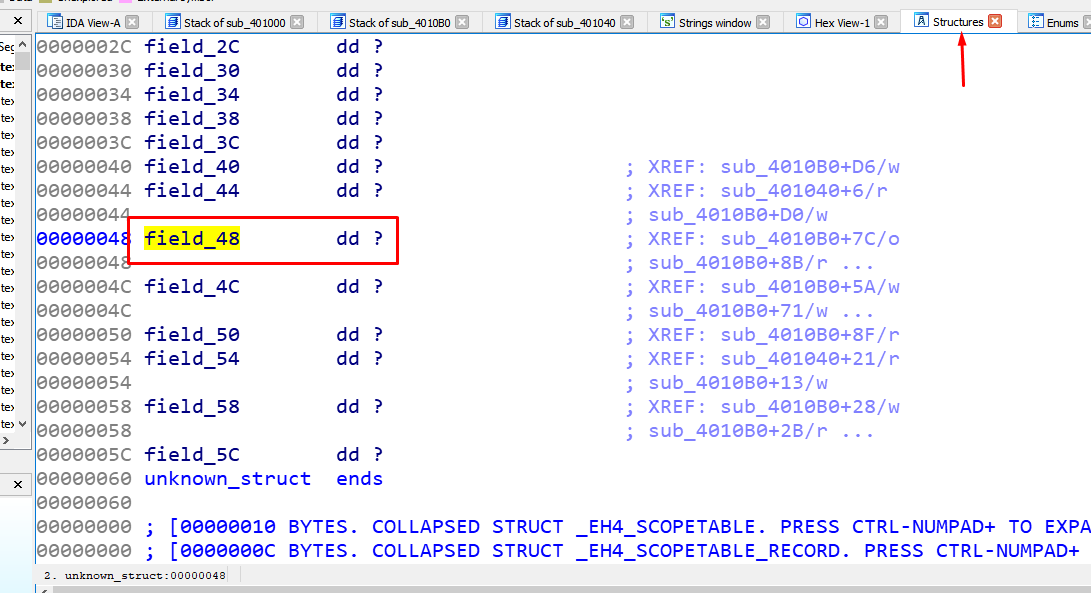
Renombro **var\_4** como **p\_store** y le pongo el tipo de puntero a estructura.

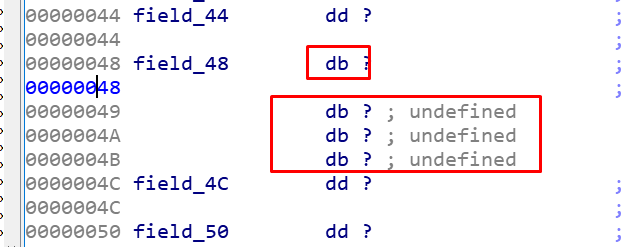


Pero antes vemos que hay un problema, el campo 0x48 es del tipo byte debemos cambiar eso.

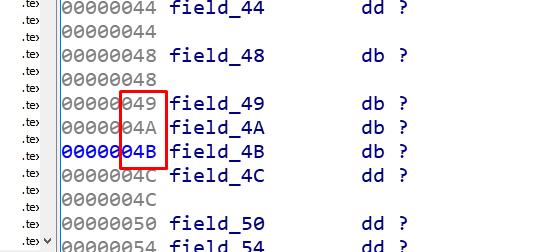


En la pestaña estructuras voy apretando la **d** hasta que quede del tipo byte.

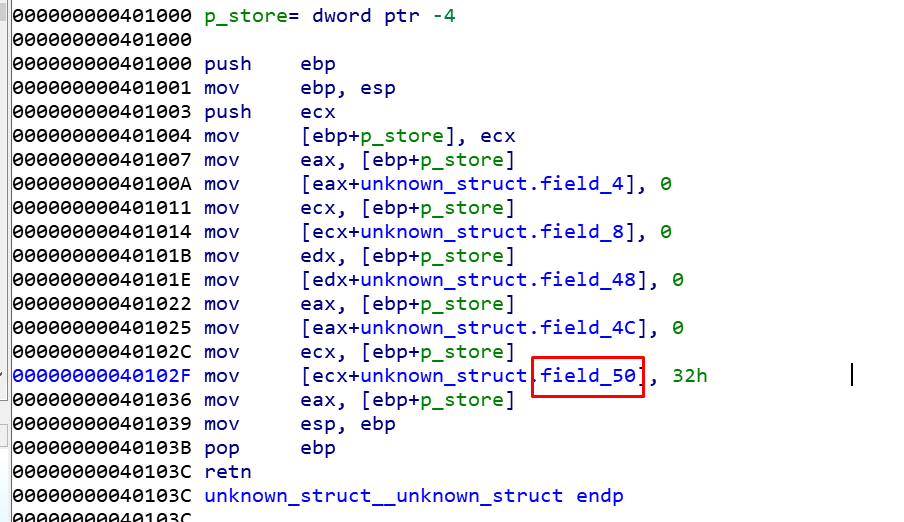




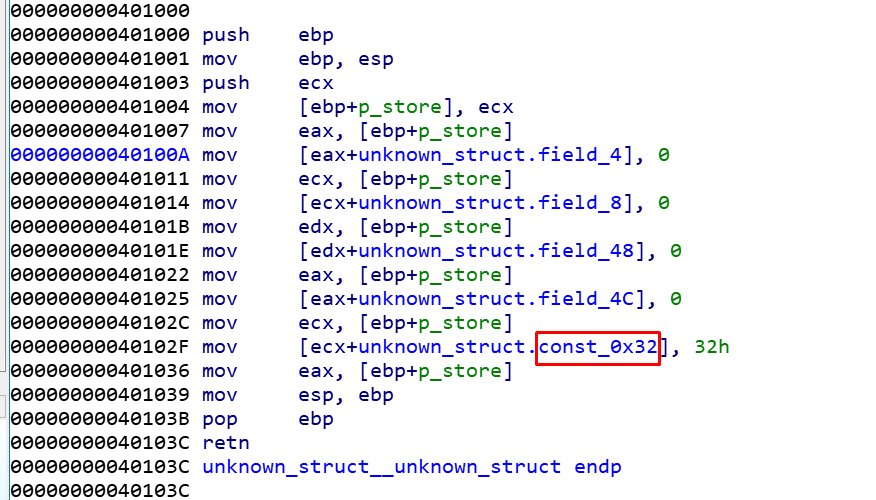
Ahí queda y en los tres que me quedan indefinidos apreto también la **d** para que queden definidos.



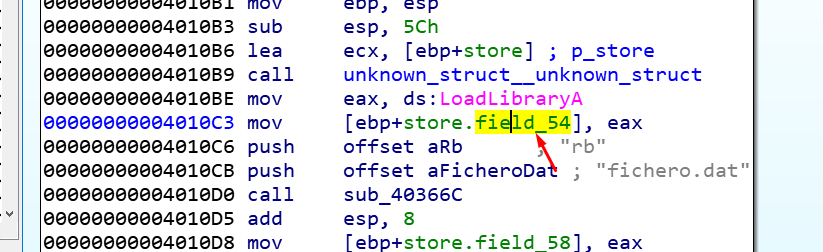
Ahora si apreto **T** en cada uno eligiendo el tipo de estructura **unknown\_struct.**



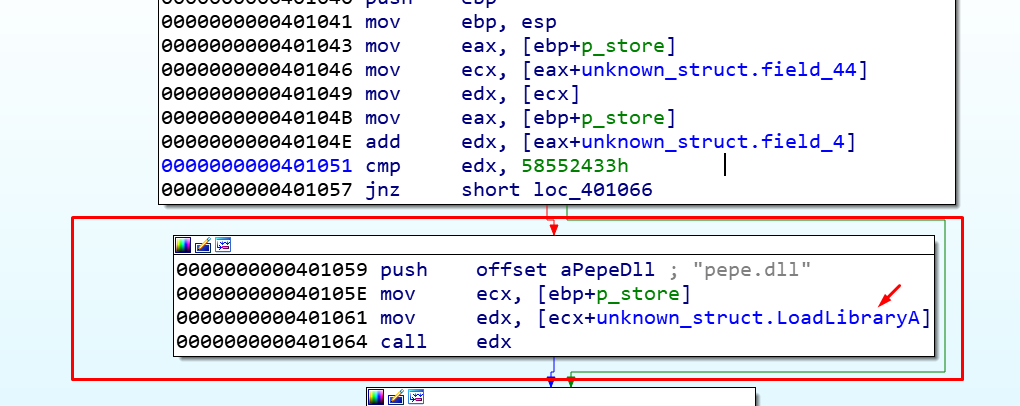
Vemos que inicializa a cero todas menos el **field\_50** que lo puedo renombrar como **const\_0x32.**



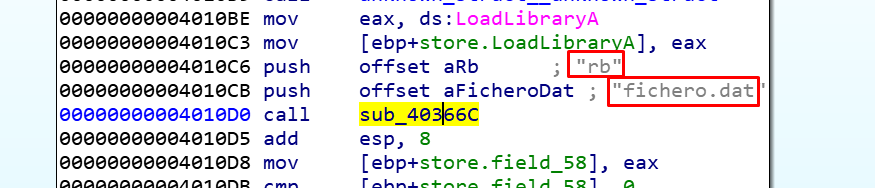
Bueno vemos acá que lee la dirección de **LoadLibraryA** de la IAT y la guarda en el campo 0x54 renombramos.

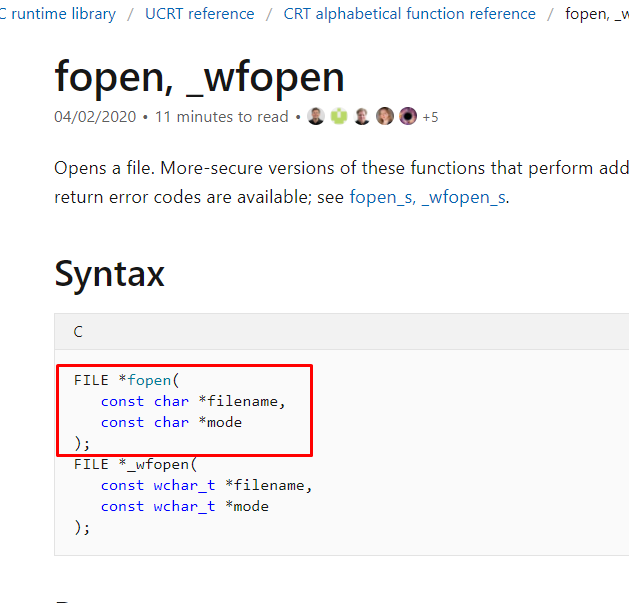


Por supuesto al cambiar aquí vemos que se cambia la vez también dentro de la función donde empezamos y que llama a **LoadLibraryA** para cargar **pepe.dll** allí.

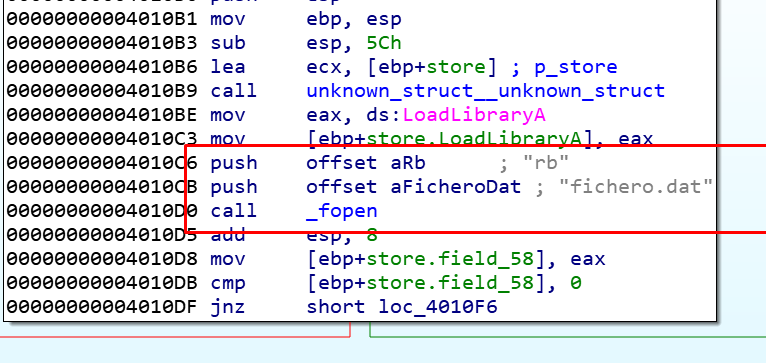


Por los argumentos nos damos cuenta de que quiere abrir un archivo en modo lectura binaria **(rb)**

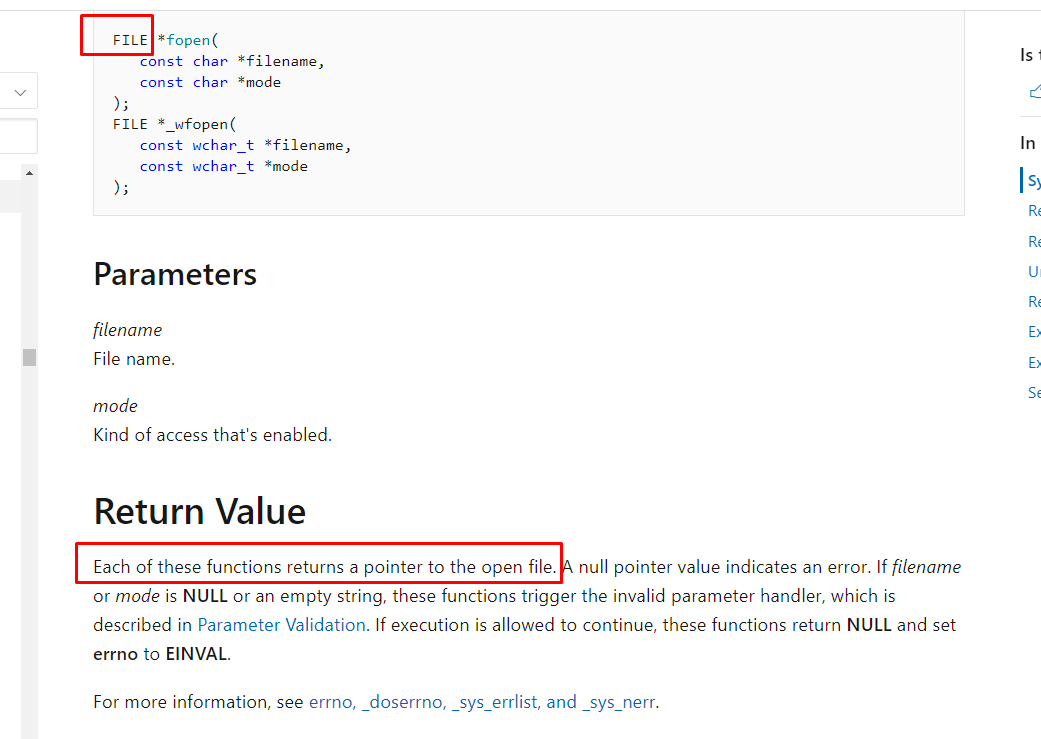


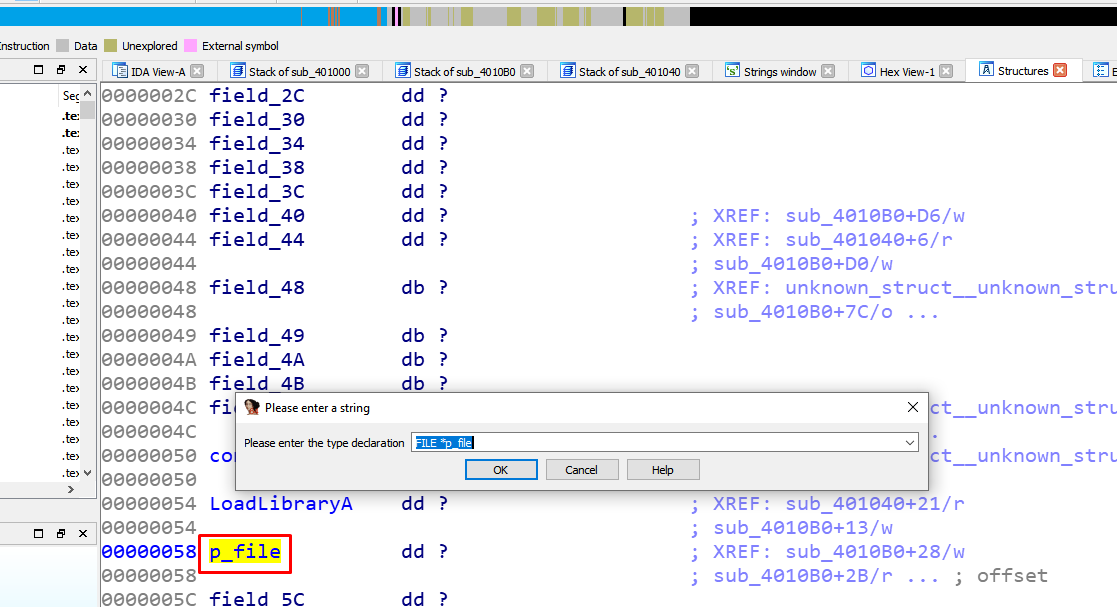


Vemos que **fopen** coincide con dichos argumentos, que son dos strings uno el nombre y otro el modo en este caso “rb”, en estos casos de archivos con dlls embebidas es bastante por experiencia darse cuenta cual api es, en general no se usan archivos con dll embebidas solo en casos muy especiales, aquí se debería ver un llamado a la función importada de la dll y nos mostraría el nombre de la funcion.

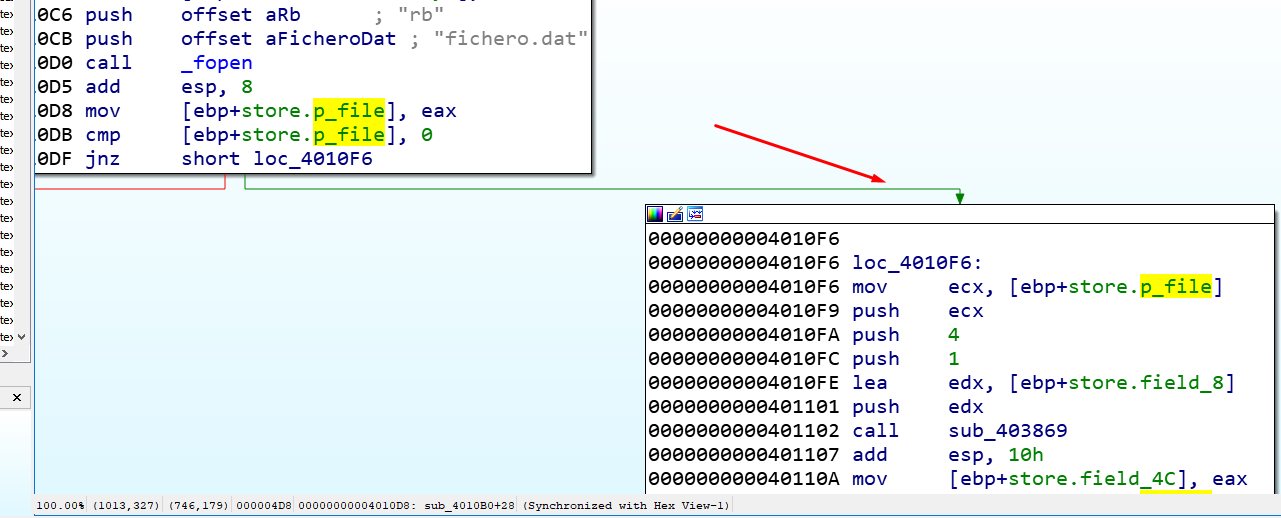


El valor devuelto es un puntero a un tipo FILE

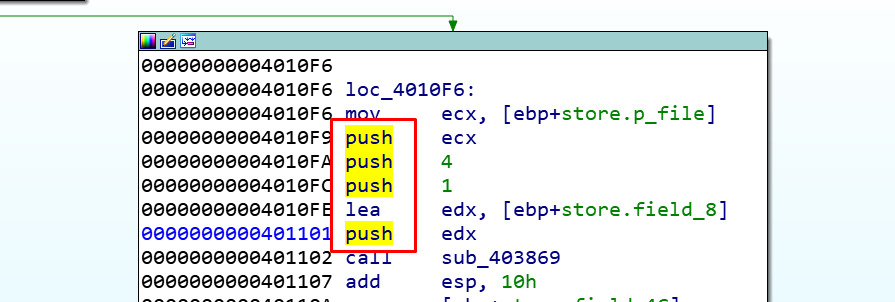




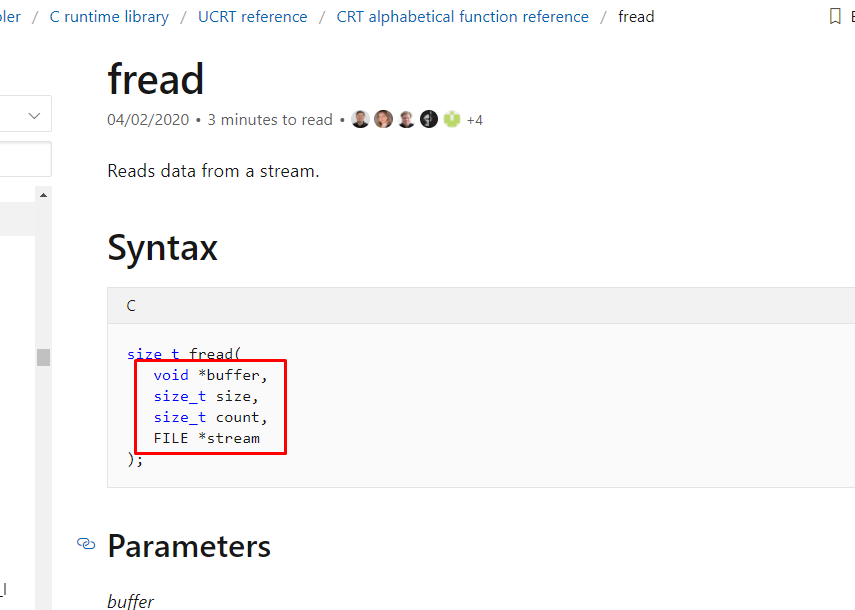
Por supuesto si existe el archivo **fichero.dat** seguirá por la flecha sino deberemos crearlo en la misma carpeta del ejecutable.

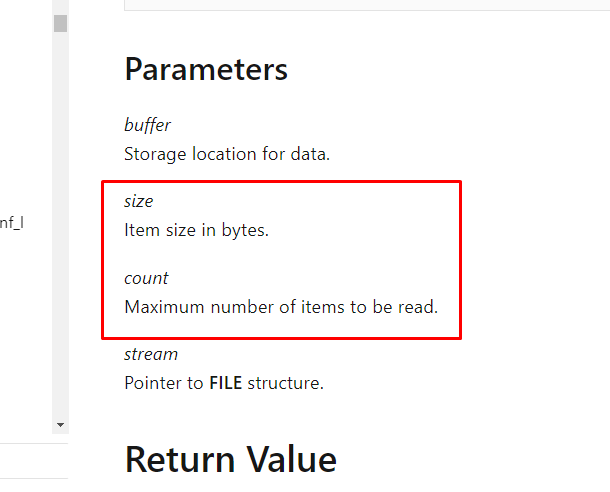


Bueno, vemos que hay una función que pienso que será por los argumentos **fread**.



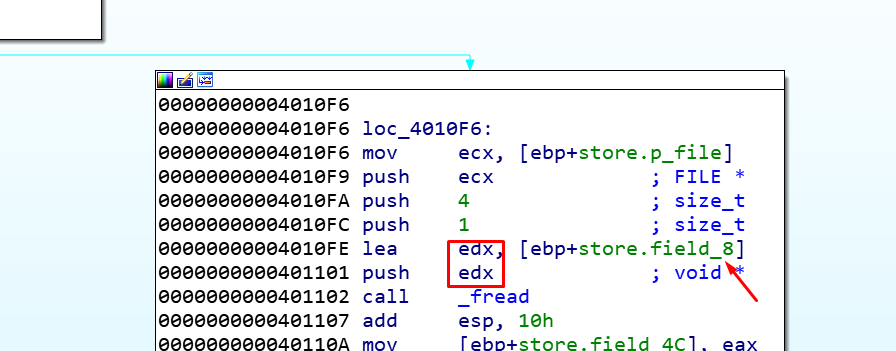
Un buffer, luego dos enteros y luego un puntero a FILE coincide plenamente con fread.



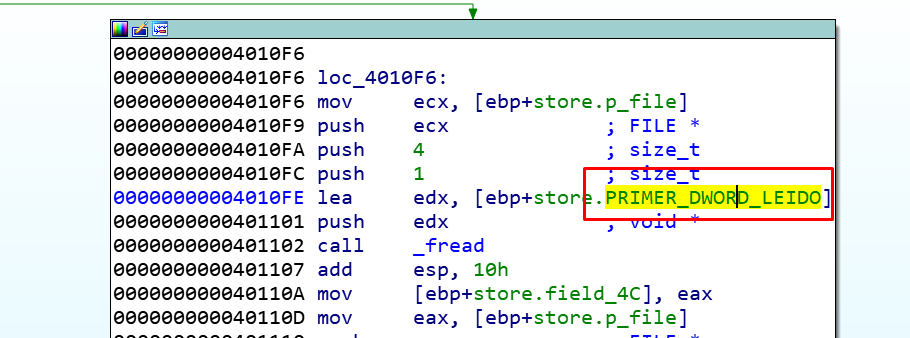


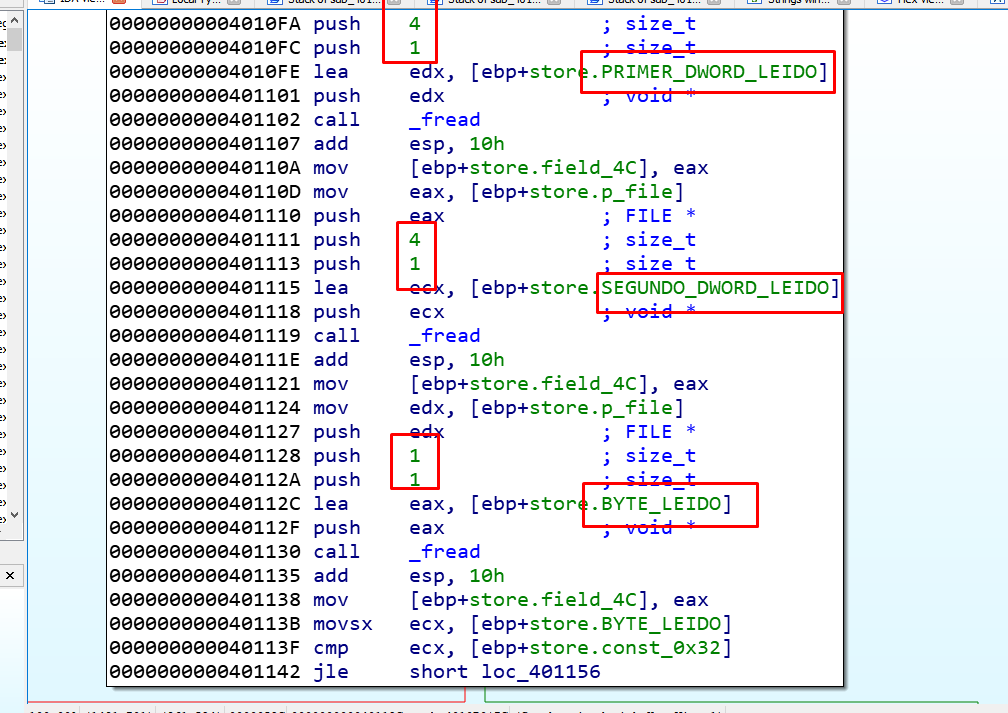
Bueno el segundo argumento es el tamaño del ítem y el tercero la cantidad de ítems a leer.

Al renombrarla como **\_fread**, IDA detecta los tipos, la realidad es que lee 4 bytes así que el buffer en EDX que es un puntero a **field\_8**, nos dice que **field\_8** seria un DWORD ya que guardara 4 bytes allí.



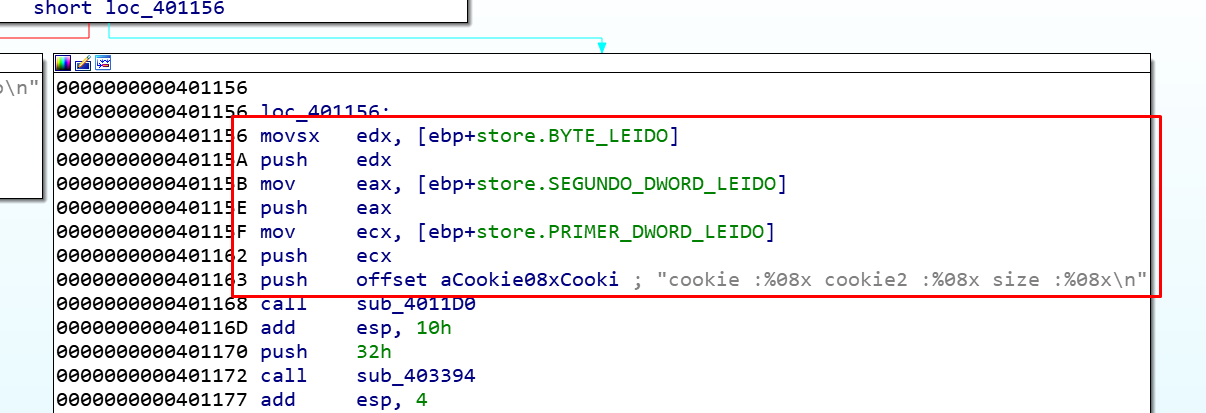
Renombro el campo de la estructura **field\_8**.



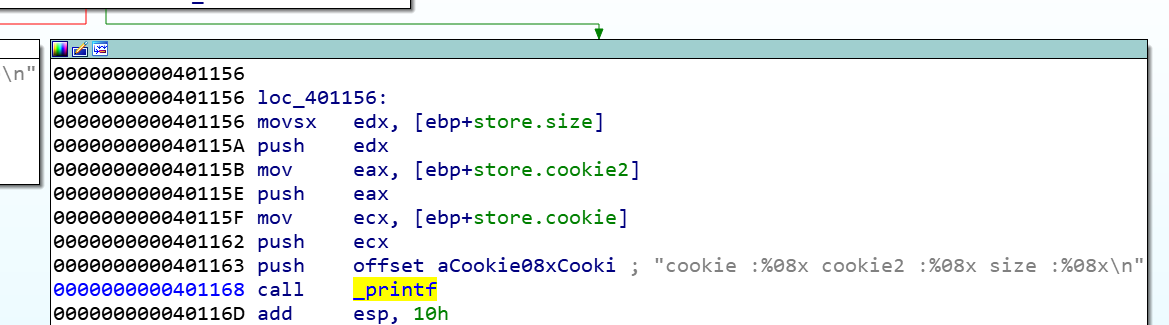


Lee luego otro DWORD y la tercera vez lee un BYTE, renombro.

Vemos que lee ese BYTE\_LEIDO y lo compare contra 0x32 si es menor sigue, recordemos que podía ser negativo y esa era la idea ya que el JL es un salto que considera el signo por lo tanto 0xff al ser negativo, el MOVSX lo completa con FFs delante y será 0xFFFFFFxx y será negativo también al comparar contra 0x32 y será menor.

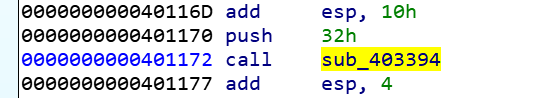


Vemos un llamado a **printf,** por los argumentos nos damos cuenta que imprime en la consola el valor de cookie, cookie2 y size, asi que podemos renombrar los valores a imprimir con los nombres que nos muestra el printf.

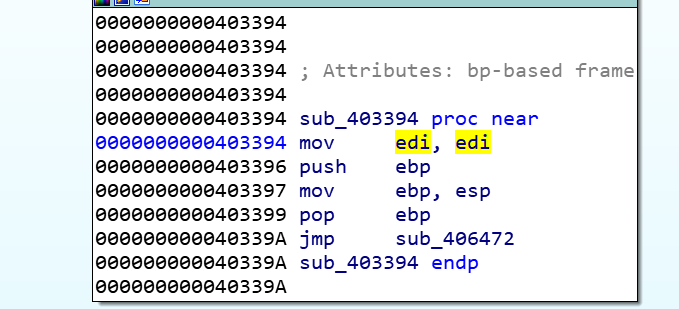


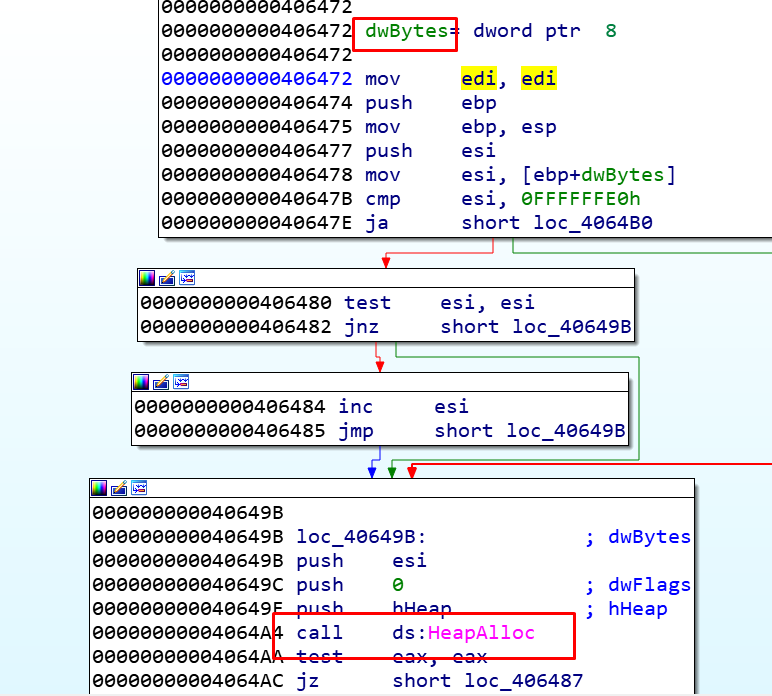
Ahora quedo más lindo.

Luego vemos una llamada que tiene como argumento un numero

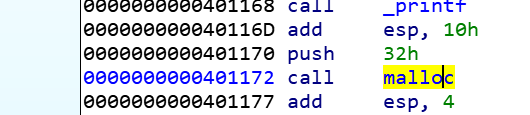


Si miramos dentro.

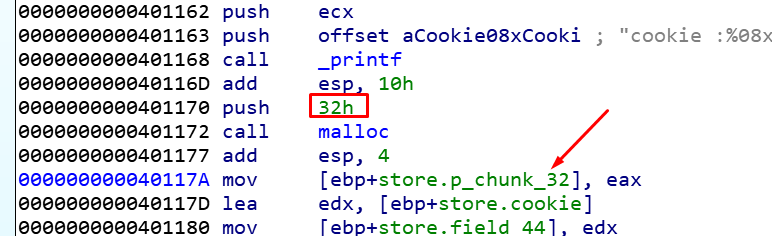




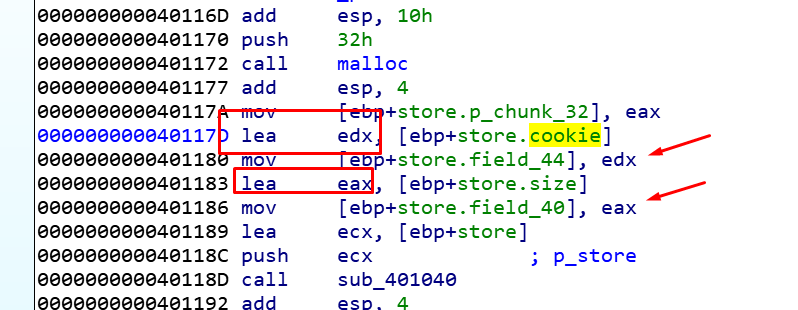
Vemos que es malloc, que tiene solo un argumento numérico.





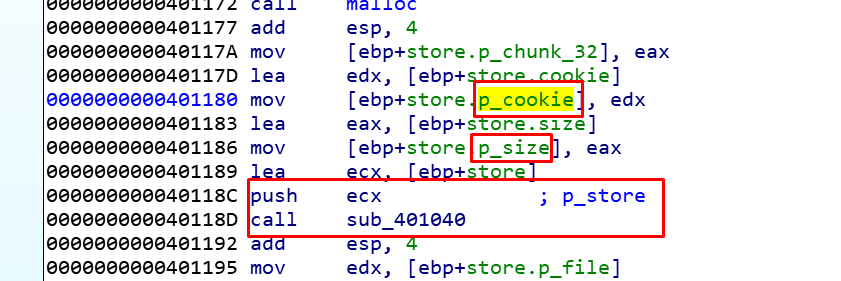


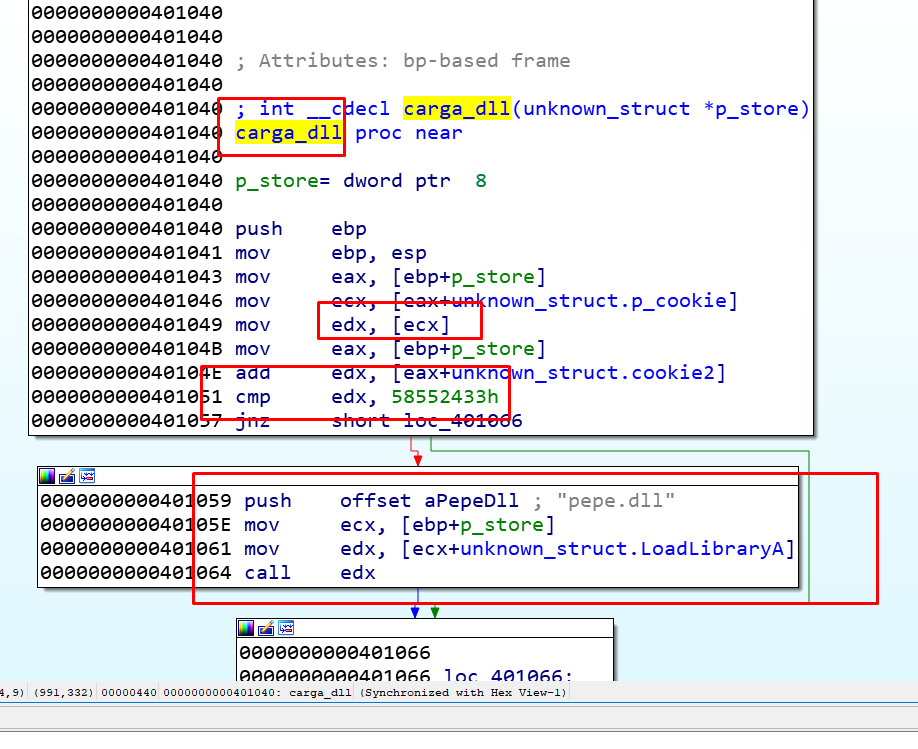
Puedo renombrarlo como **p\_chunk\_32.**



Vemos que guarda un puntero a **cookie** y un puntero a **size** en **field**\_44 y **field\_40,** renombro.

Y llegamos a la función donde empezamos ya con todo reverseado.



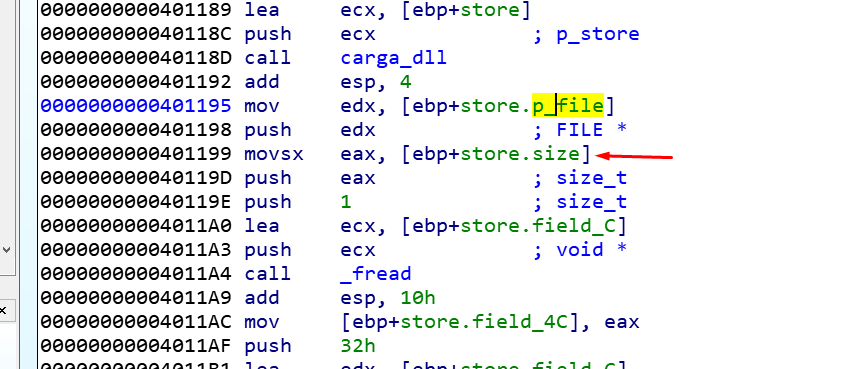


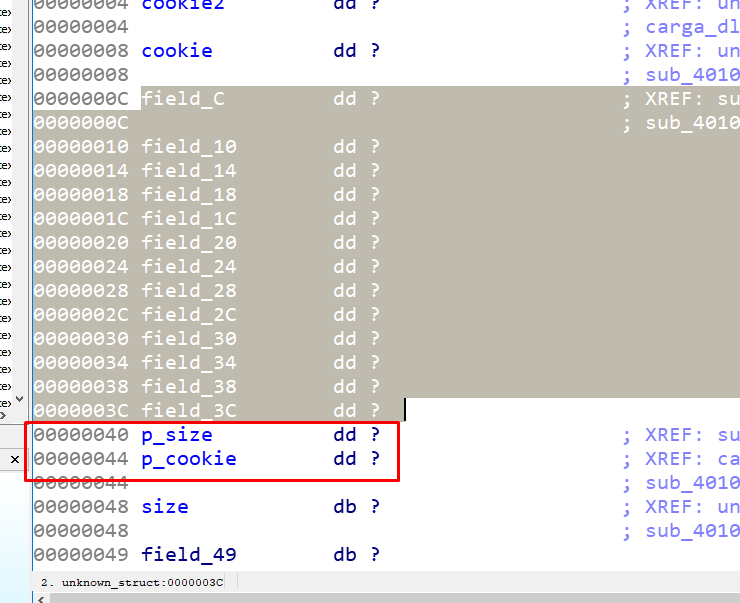
Le cambio el nombre a la funciona a **carga\_dll** ya que eso hace si se da que la suma de cookie + cookie2 es igual a **0x58552433**.

Vemos que halla el contenido de **p\_cookie** que es **cookie** y lo suma a **cookie2** y lo compara.

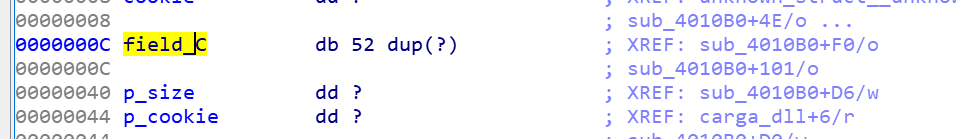
Así que ya sabemos que para que cargue la dll los 4 primeros bytes del archivo sumados a los 4 segundos bytes debe dar 0x58552433 y el noveno byte debe ser mayor que 0x7f para que sea negativo.

Finalmente va a llamar a **fread** con el size negativo que al usar **MOVSX** se rellenara con FFs y desbordara el buffer **field\_C,** veamos cual es el largo de este buffer.

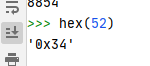




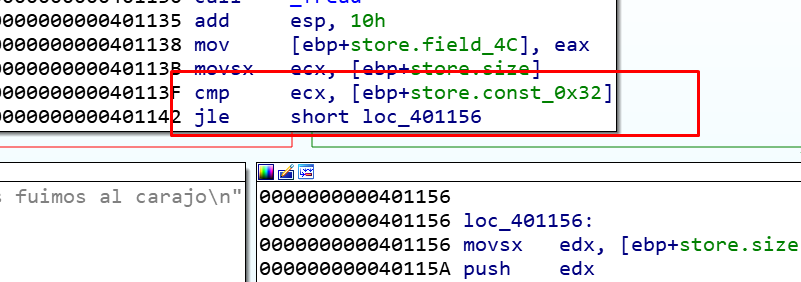
Vemos en la estructura, la zona marcada será el buffer, ya que luego vienen **p\_size** y **p\_cookie**.



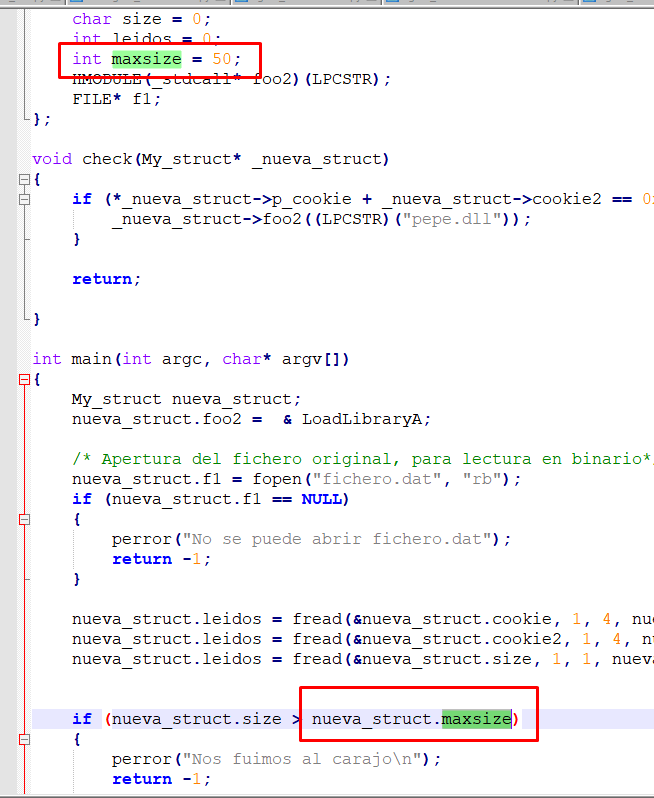
Lo cambie a byte apretando la tecla d en **field\_C** y luego cree un buffer con ARRAY, tiene de largo 52 decimal.



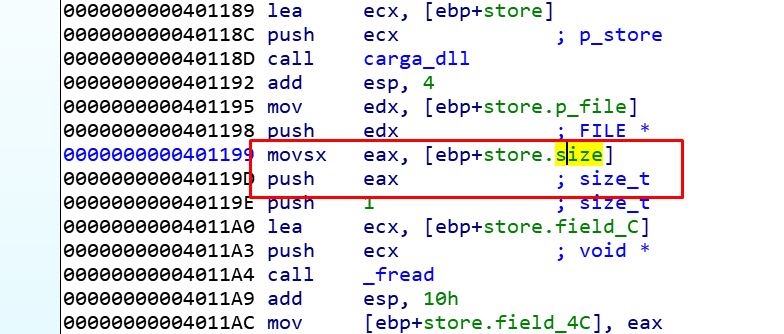
Recordemos que chequeaba si el size a copiar era mayor que 0x32.



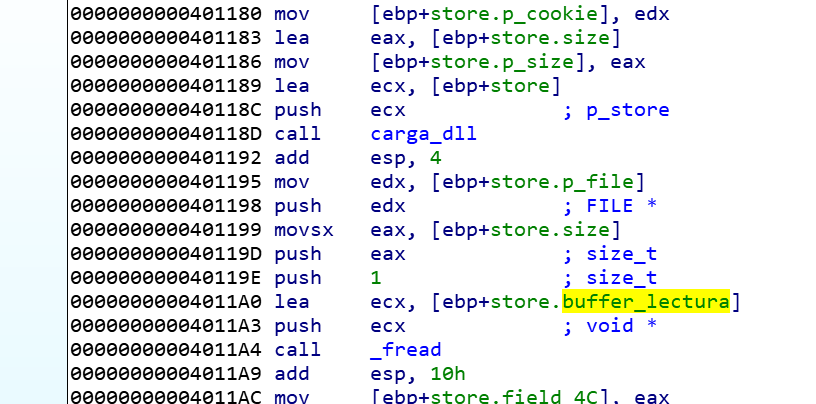
El problema es que considero el signo en la comparación porque el programa uso **signed**, sino dice **unsigned** específicamente el compilador usa **signed** por default.



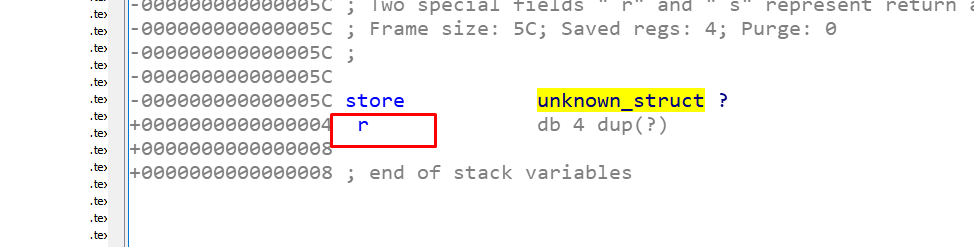
Y la cuestión es que ese 0xFFFFFFXX será el size a leer del archivo, ya que fread considera como unsigned el size ya que el tipo **size\_t** es unsigned.



Así que podemos leer la cantidad que queremos y terminará donde no pueda leer más, lo cual nos permitirá pisar el return address.

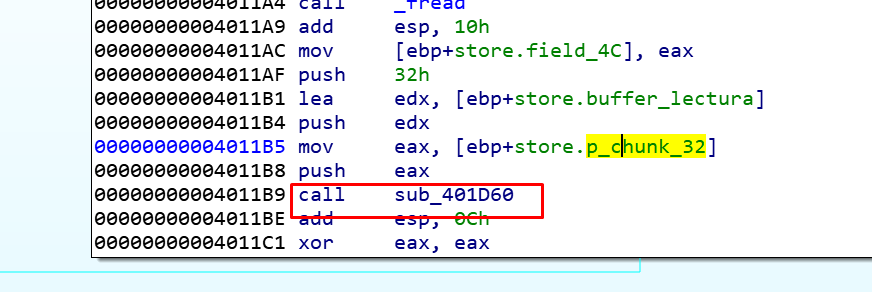


Ya que debajo de la estructura está el return address.

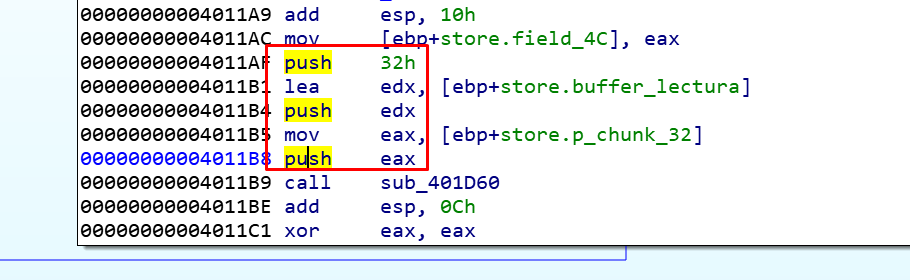


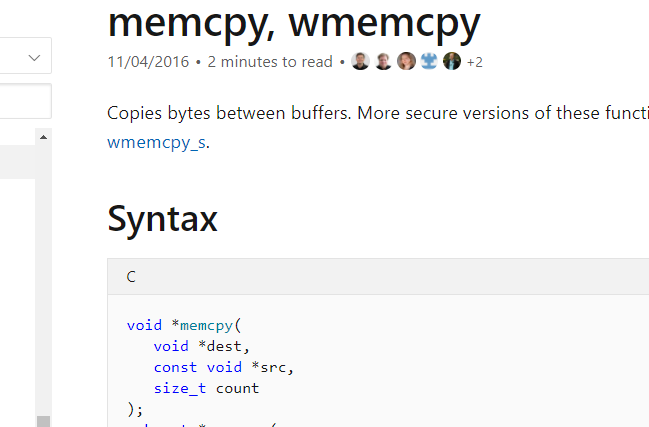
Así que deberemos llenar toda la estructura y pisar el return address.

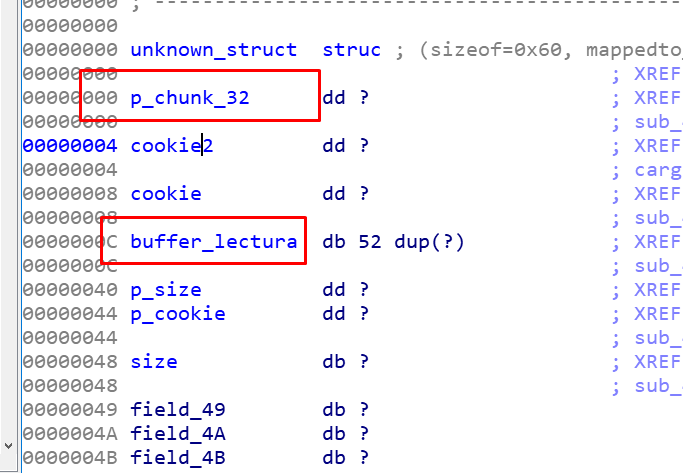
Igual debemos tener cuidado hasta llegar al return address, vemos que luego hay un **memcpy**



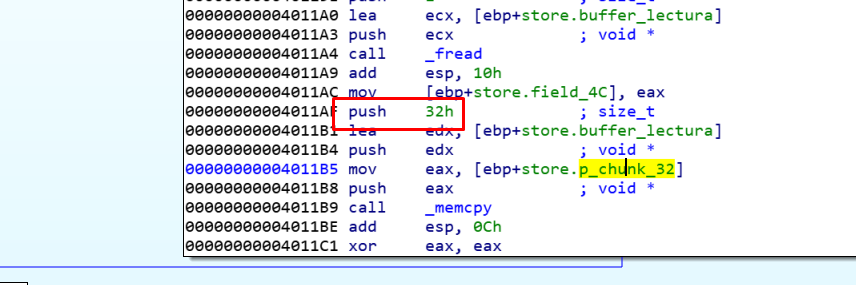
Tiene 3 argumentos un buffer y un size





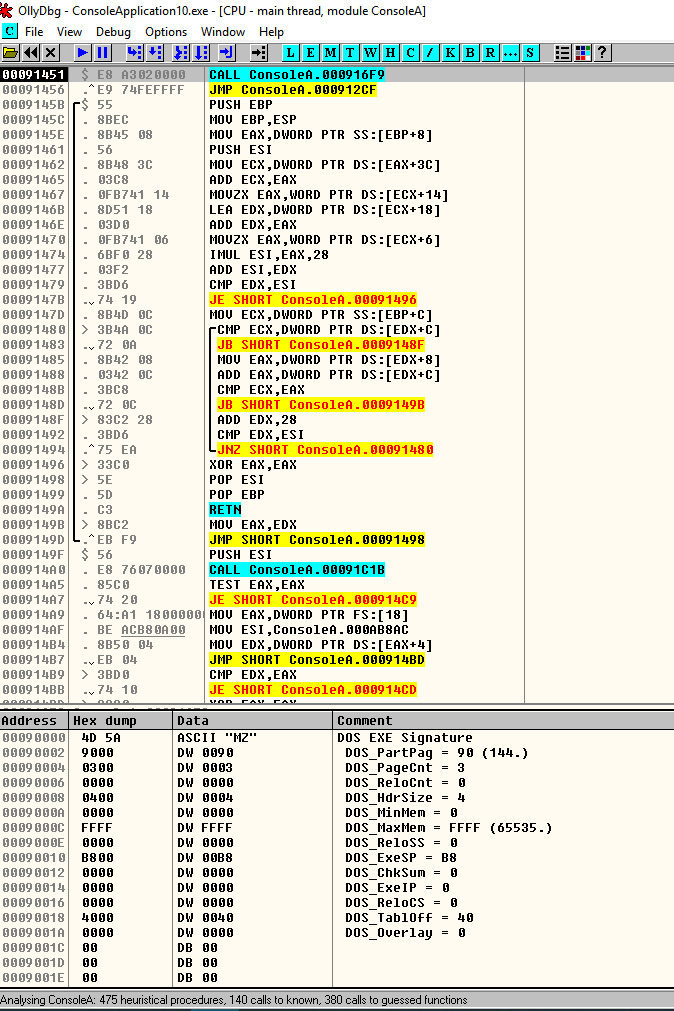


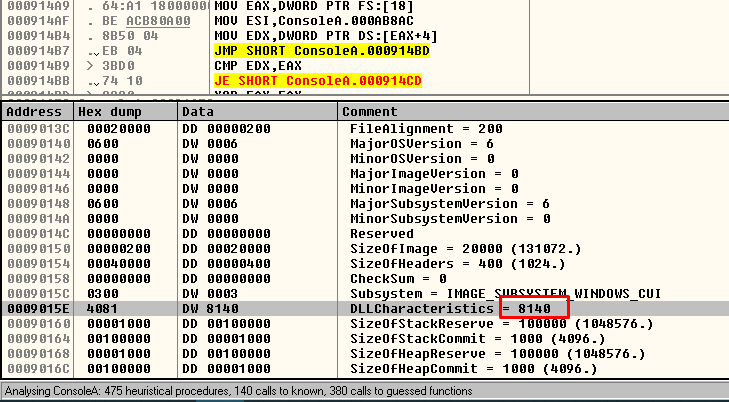
Pero el destination que es el buffer creado con malloc no fue pisado su puntero, así que no habrá problema ahí, el size está harcodeado a 0x32.



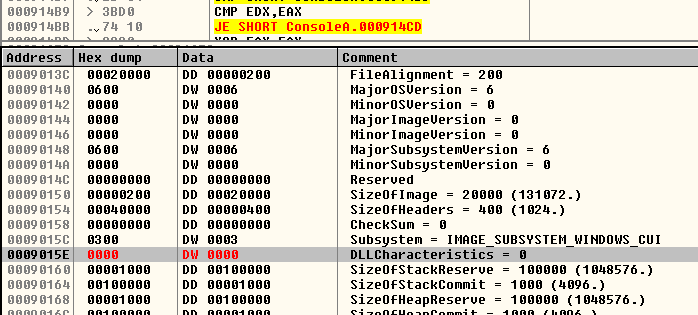
También la pepe.dll tiene la función **system** embebida que se puede llamar directamente pues no tiene randomización por lo cual será fácil ejecutar la calculadora con eso.

Lo que si me equivoque es que el archivo original no tenia randomización y el que bajamos si, se la quitaremos mas adelante ya lidiaremos con eso, en el OLLYDBG es el más rápido cambiamos a special -pe header y buscamos dll characteristics.

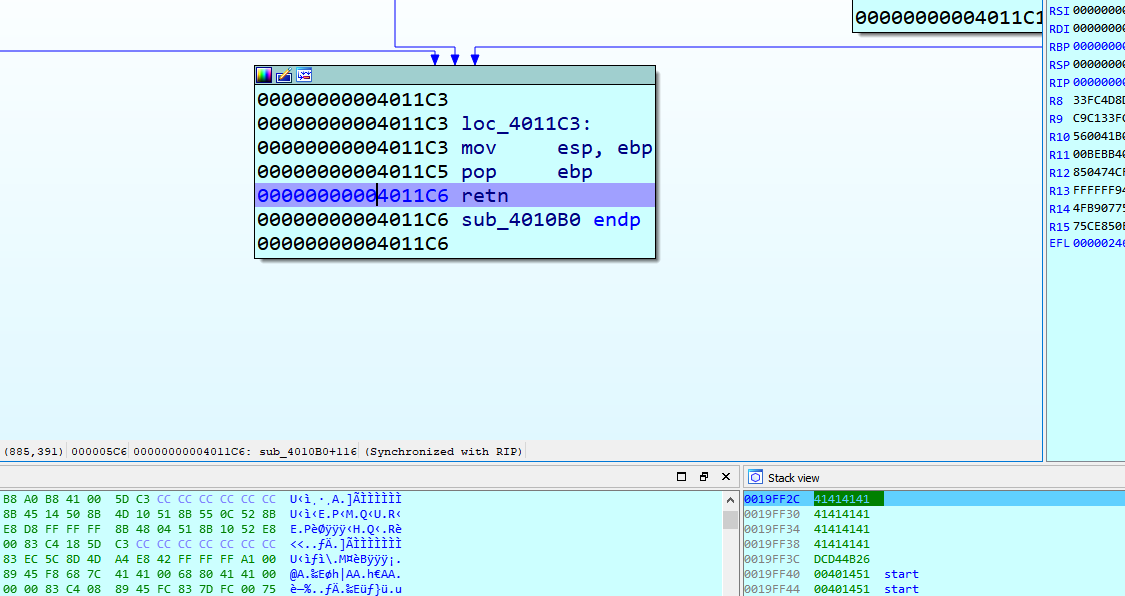




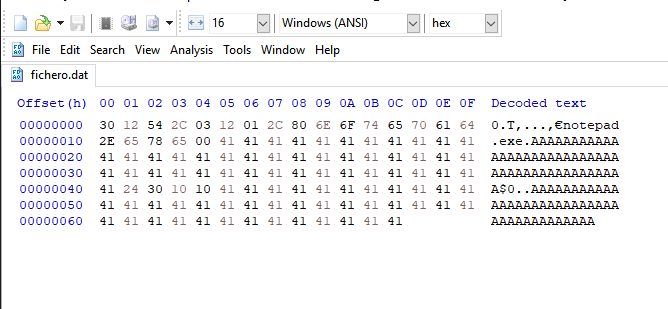
Ponemos a cero y guardamos



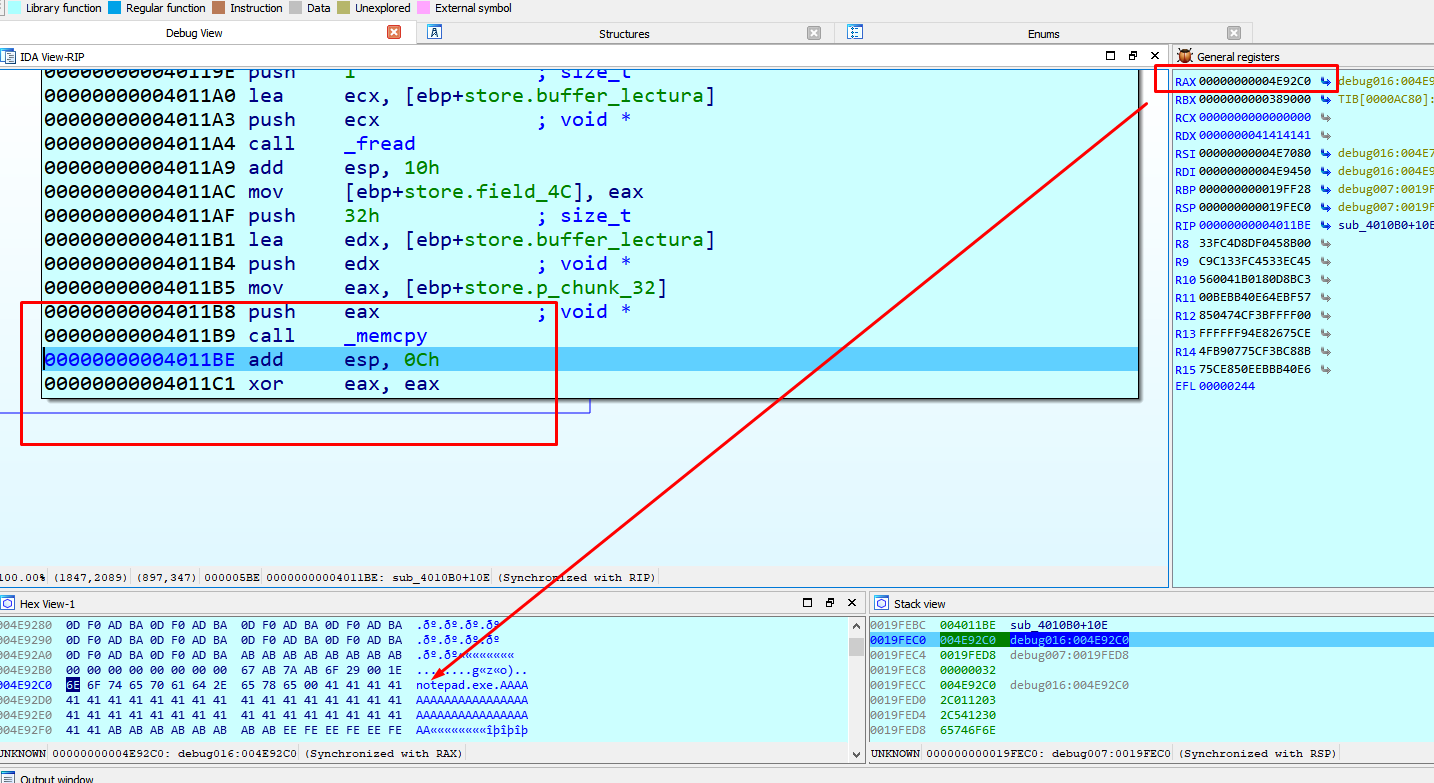
Luego copy to executable y luego save file.



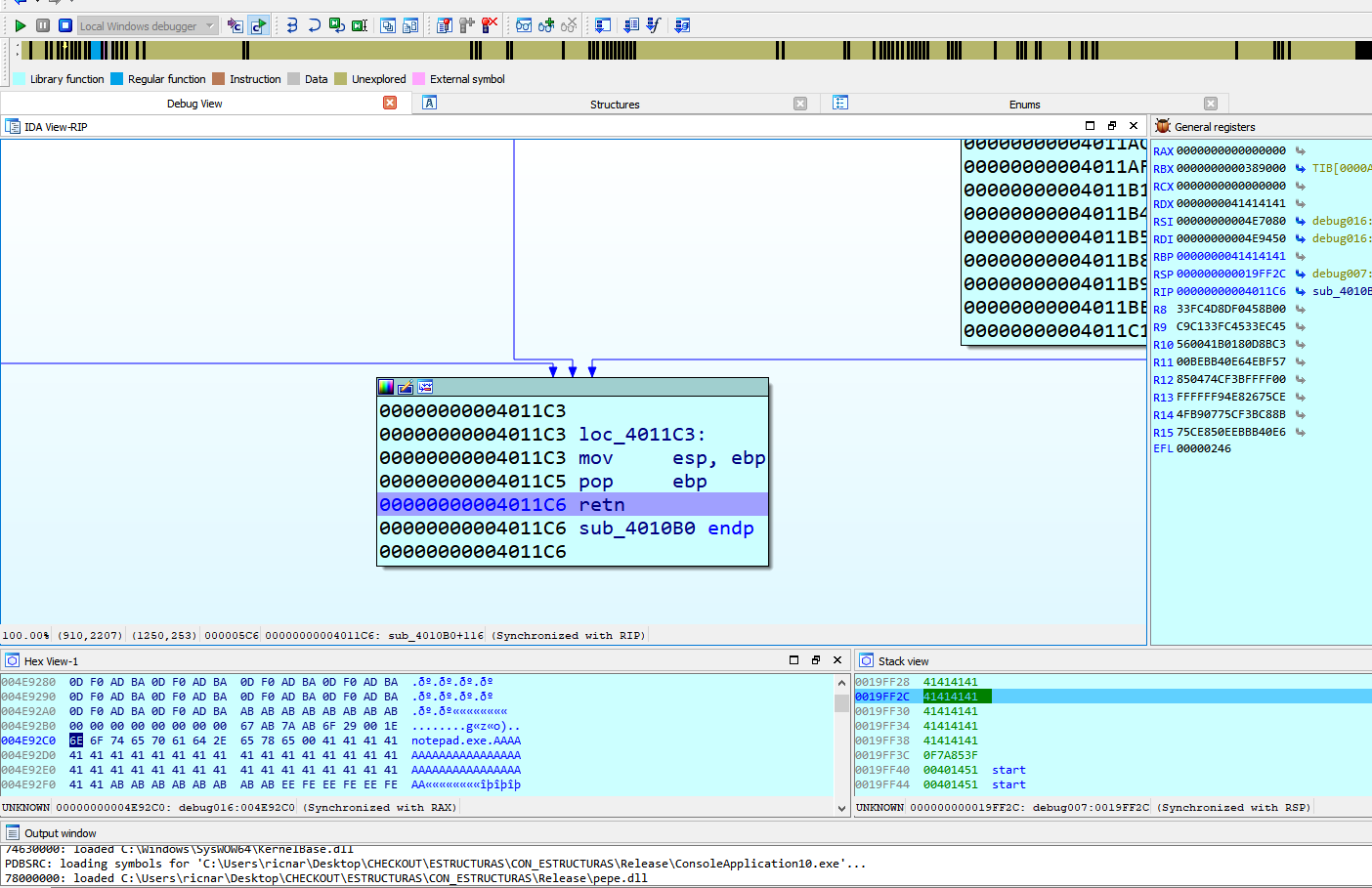
Vemos que con un **fichero.dat** adecuado como reverseamos

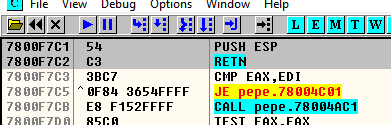


Vemos que en esta versión también borra el puntero que quedaba en EAX a la string que queremos, pues ese es nuestro buffer que lo copio al que creo en malloc mediante el memcpy, así que eso nos complicó.

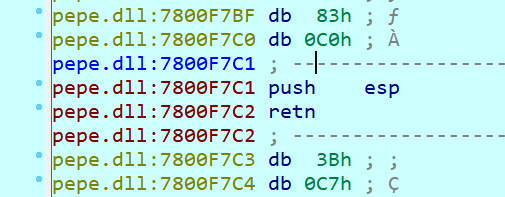


O sea, ya llegamos pero en ningún lado tenemos un puntero al buffer.

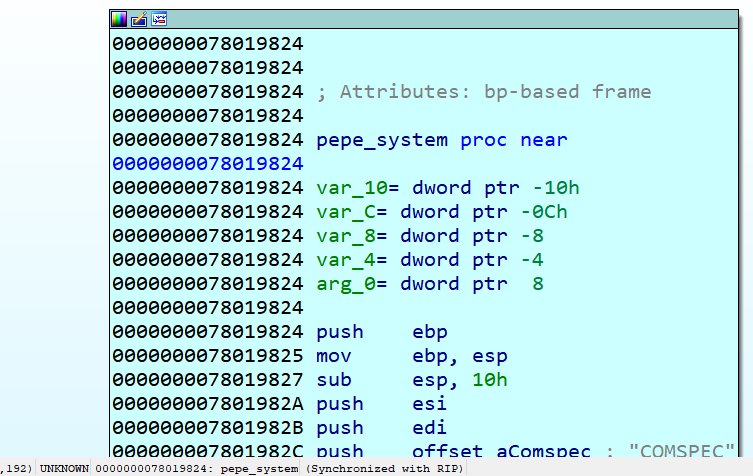




Por suerte en pepe.dll hay un **push esp-ret**.



System está allí



Con un código como

**0019FF30 8BC4 MOV EAX,ESP**

**0019FF32 83E8 64 SUB EAX,64**

**0019FF35 8B00 MOV EAX,DWORD PTR DS:[EAX]**

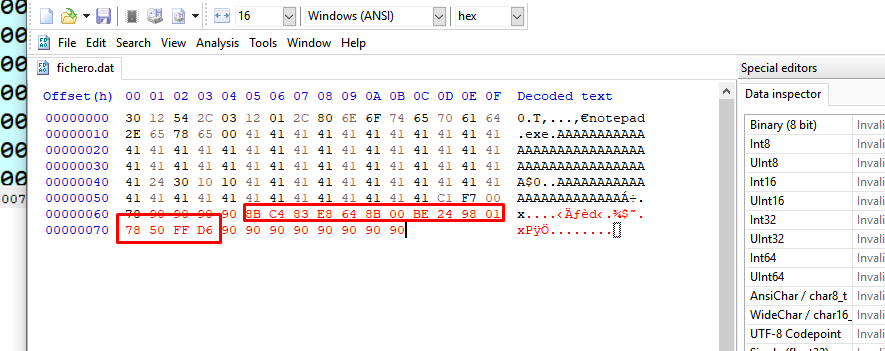
**0019FF37 BE 24980178 MOV ESI,pepe.system**

**0019FF3C 50 PUSH EAX**

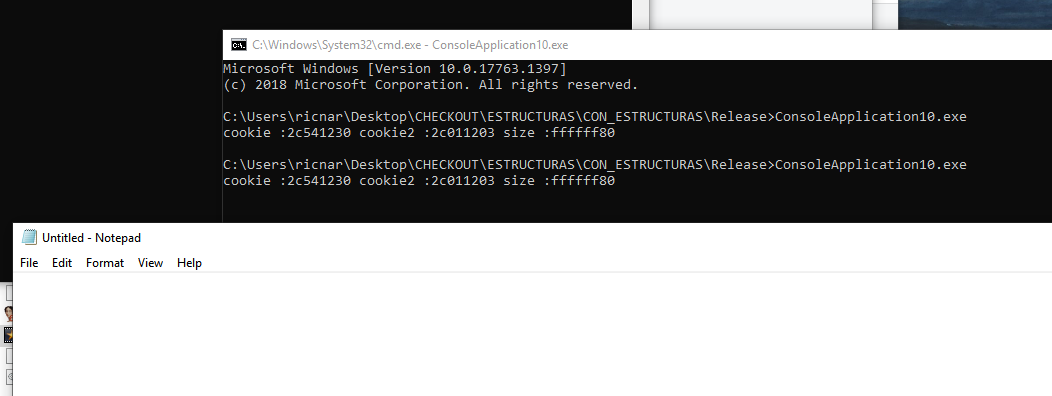
**0019FF3D FFD6 CALL ESI**

**8B C4 83 E8 64 8B 00 BE 24 98 01 78 50 FF D6**

Lo pondremos a continuación del **push esp-ret**



Y ya ejecuta Notepad



Hasta la parte 21 donde haremos este mismo ejercicio en Radare y en el siguiente en GHIDRA.

PD:No era necesario quitarle la randomización, la solución funciona igual en el original pero de paso aprendimos algo más je, sino tiene dep habilitado funcionaria igual, pero bueno de paso aprendemos como se quita para practicar.

Suerte

25/09/2020

Ricardo Narvaja (aun en cuarentena)