EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

GRATUITAS (PARTE 19)

Comenzaremos a ver un tema importante en el reversing estático el manejo de estructuras.

La verdad es que hace años trabajo con IDA PRO y se como trabajar con ellas en el mismo, con IDA FREE es similar y al momento de escribir esto aun no tengo ida de como manejan las estructuras Radare y Ghidra, pero lo averiguaremos y lo escribiremos en estos tutoriales.

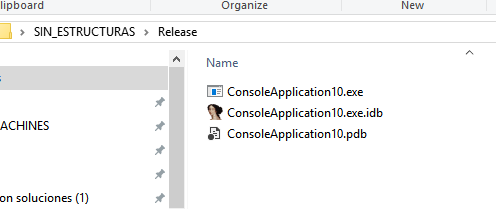
Lo primero que haremos antes de resolver ejercicios con estructuras es modificar un viejo ejercicio sin estructuras primero y luego agregarle estructuras para ver como se ve diferente a bajo nivel.

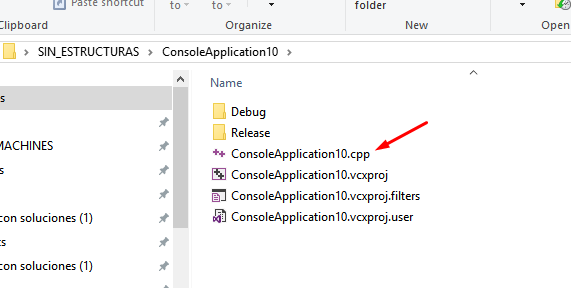
<https://drive.google.com/file/d/10p5i8gDHLUgz9Xag_1XadpHBCyNAPisB/view?usp=sharing>

password =a

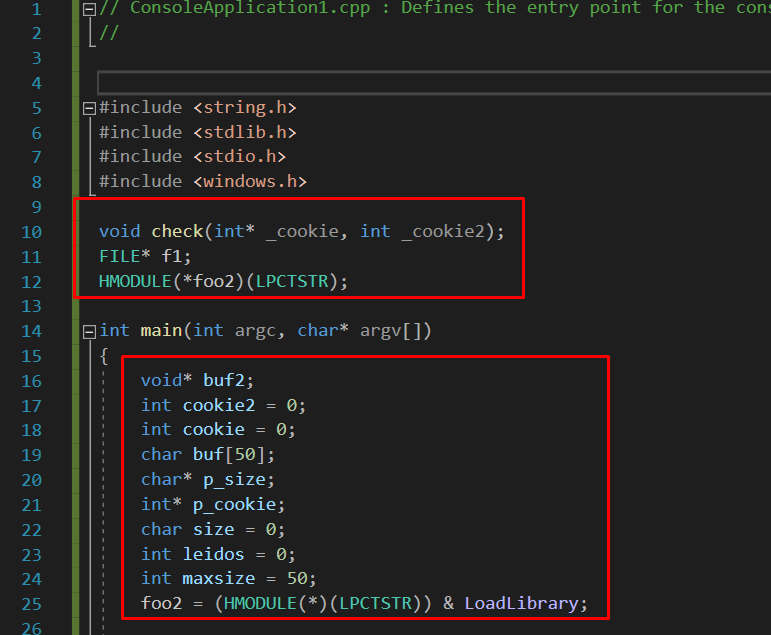
Le haremos un reversing estático rápido, para entender las diferencias con la otra versión, que es la que nos importa, ustedes pueden practicar completando la solución.

Dentro esta el ejecutable y los símbolos y el código fuente.

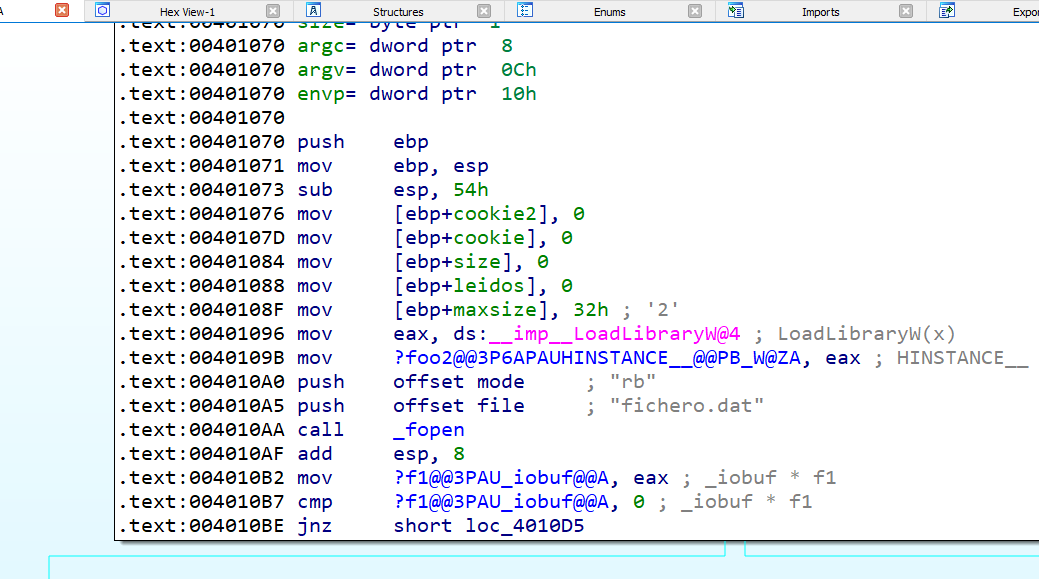




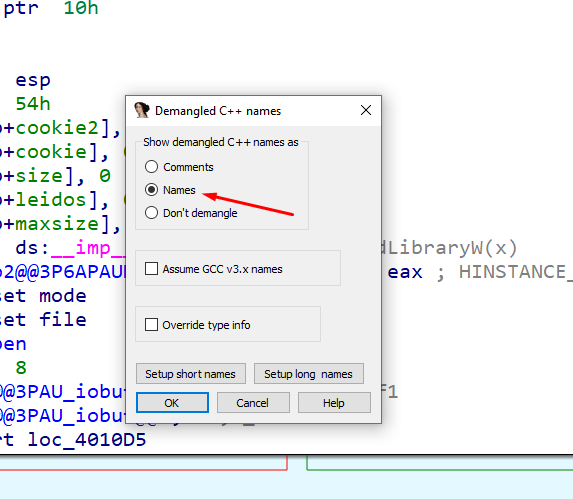
Vemos en el código fuente que hay variables globales que están declaradas antes del main, y también vemos las variables locales del main.

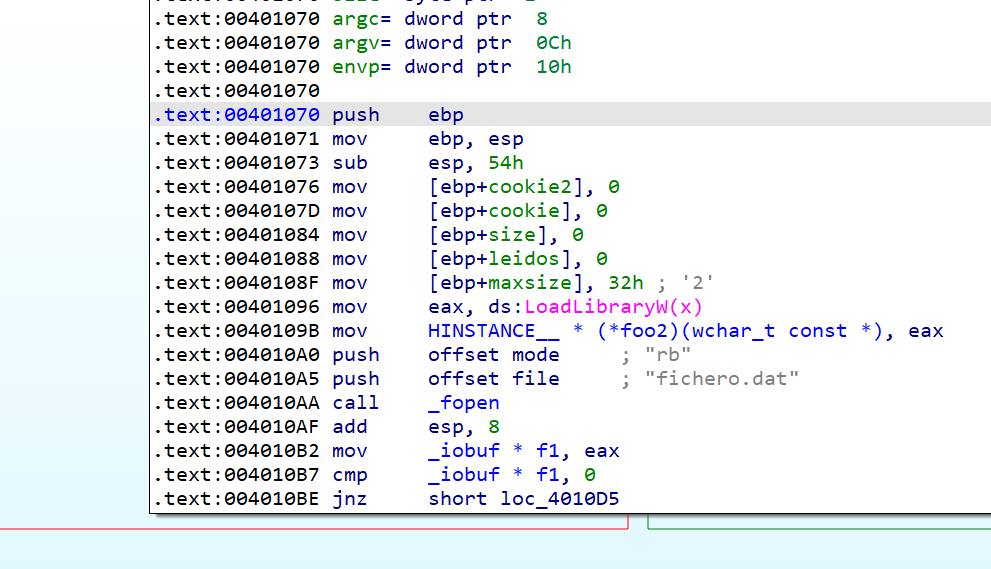


Utilizaremos los símbolos pues la idea es entenderlo, en futuros ejercicios no los usaremos para ir paso a paso y gradualmente.



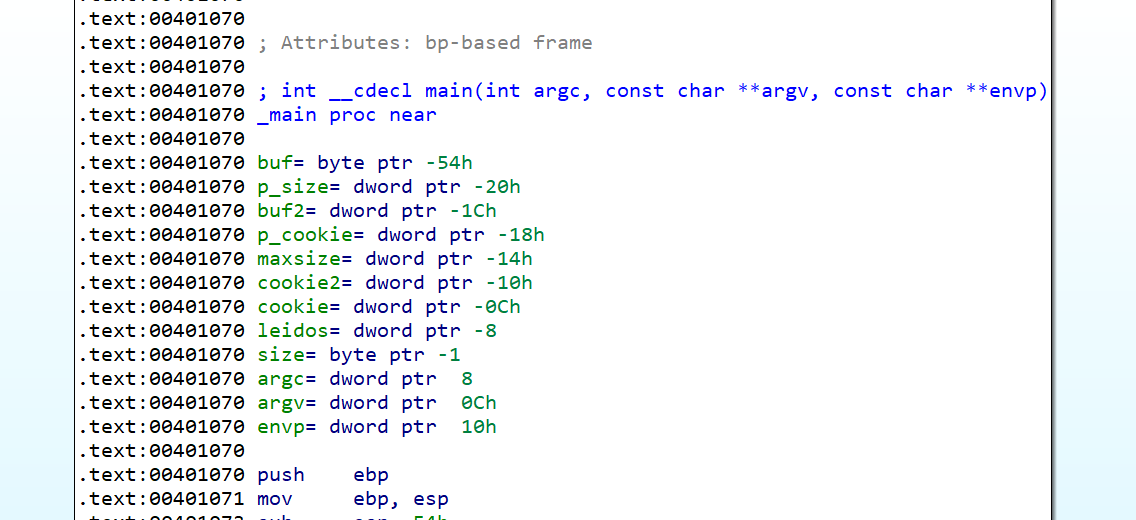
Se ve feo usemos DEMANGLE NAMES.

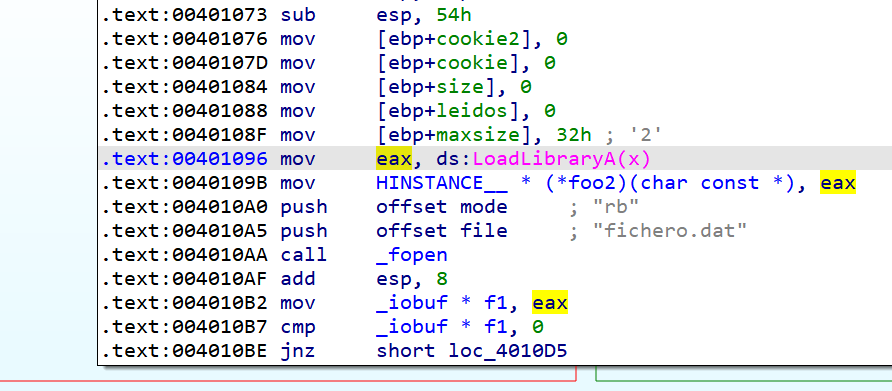




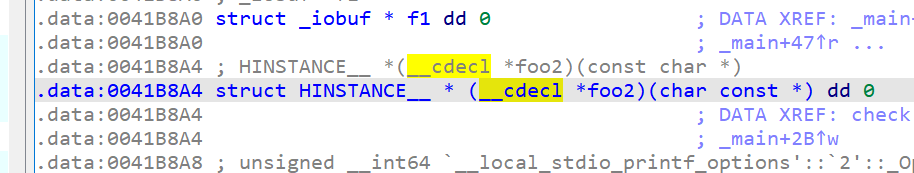
Ahora se ve mejor.

Vemos todas la variables locales sueltas de diferentes tipos, hay tipo byte, dwords, buffers etc.

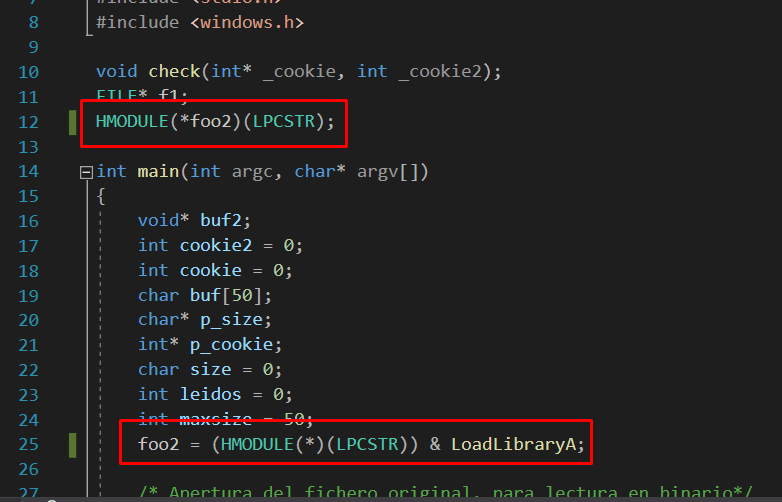




Vemos que luego de inicializar las variables locales con cero y una variable **maxsize** con 0x32, lee la dirección de la función **LoadLibraryA** y la guarda en una variable **foo2** que es global ya que está en la sección data.

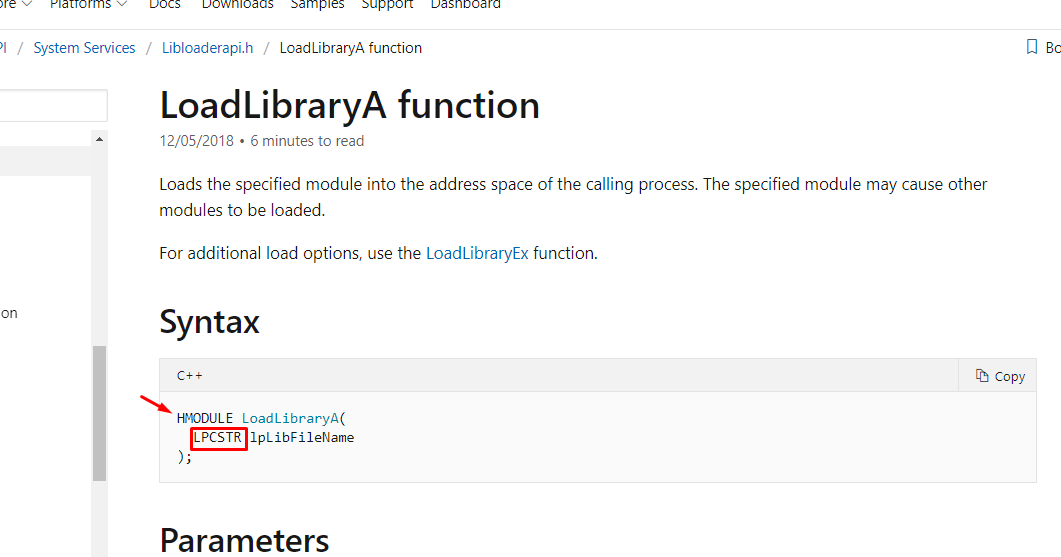


Vemos que es una variable del tipo puntero a función, lo vemos en el código fuente también.



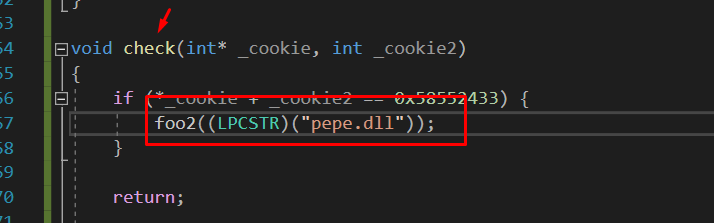
Como es global, está en la sección data, y abajo en el main vemos como se guarda la dirección de LoadLibraryA.

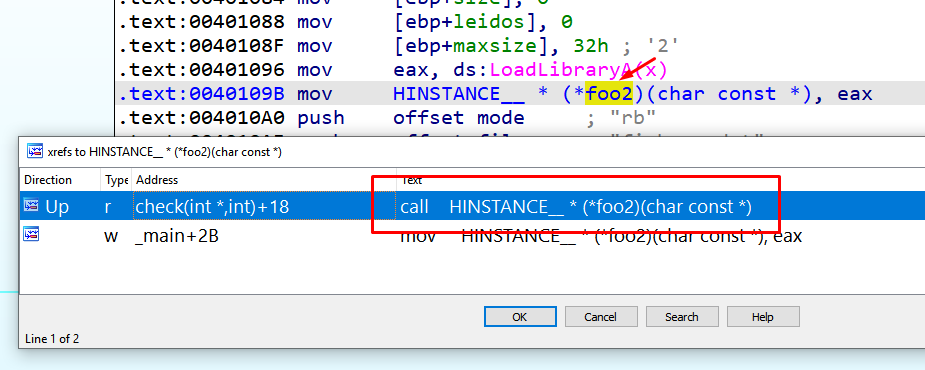
Justamente esta declarada como una variable a función que tiene un solo argumento del tipo (LPCSTR) y devuelve un HMODULE.



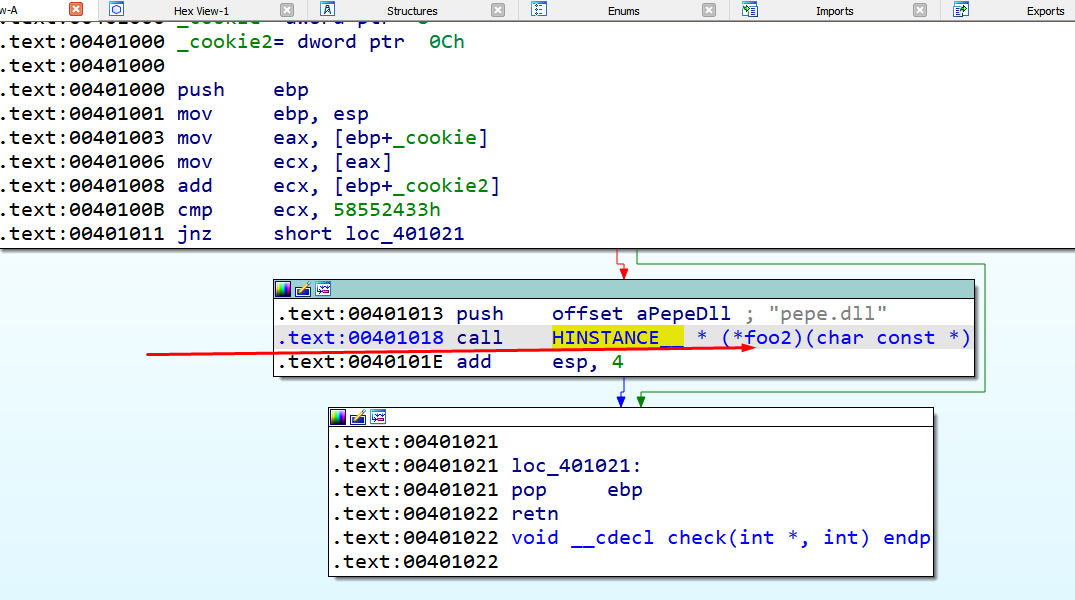
Justo el mismo tipo de argumento y tipo de valor de retorno que LoadLibrary, de forma que no habrá problemas para que foo2 guarde su dirección.

Vemos en el código fuente que usa LoadLIbraryA para cargar un módulo llamado pepe.dll.



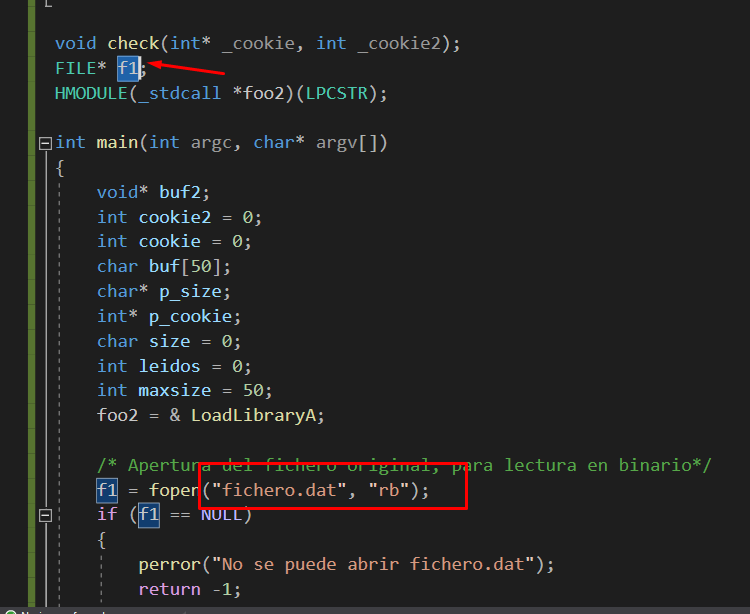


Si apreto **x** en la variable vemos que hay un sitio donde puede cargar el módulo, dentro de la función **check().**

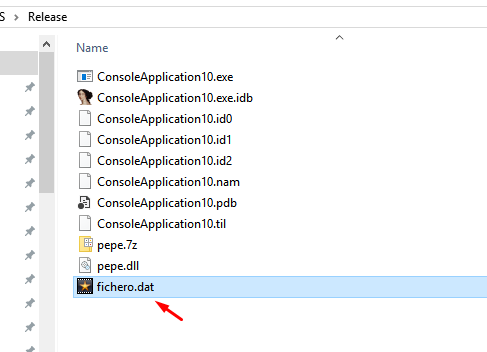


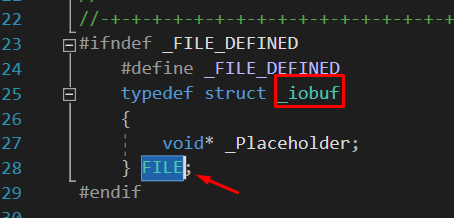
<https://drive.google.com/file/d/1KDzn04vcgwA5MmZh-93kNLFPCPQsr8UO/view?usp=sharing>

Ahí está el módulo pepe.dll (password a) bajémoslo y coloquémoslo en la misma carpeta que el ejecutable, así la encuentra.



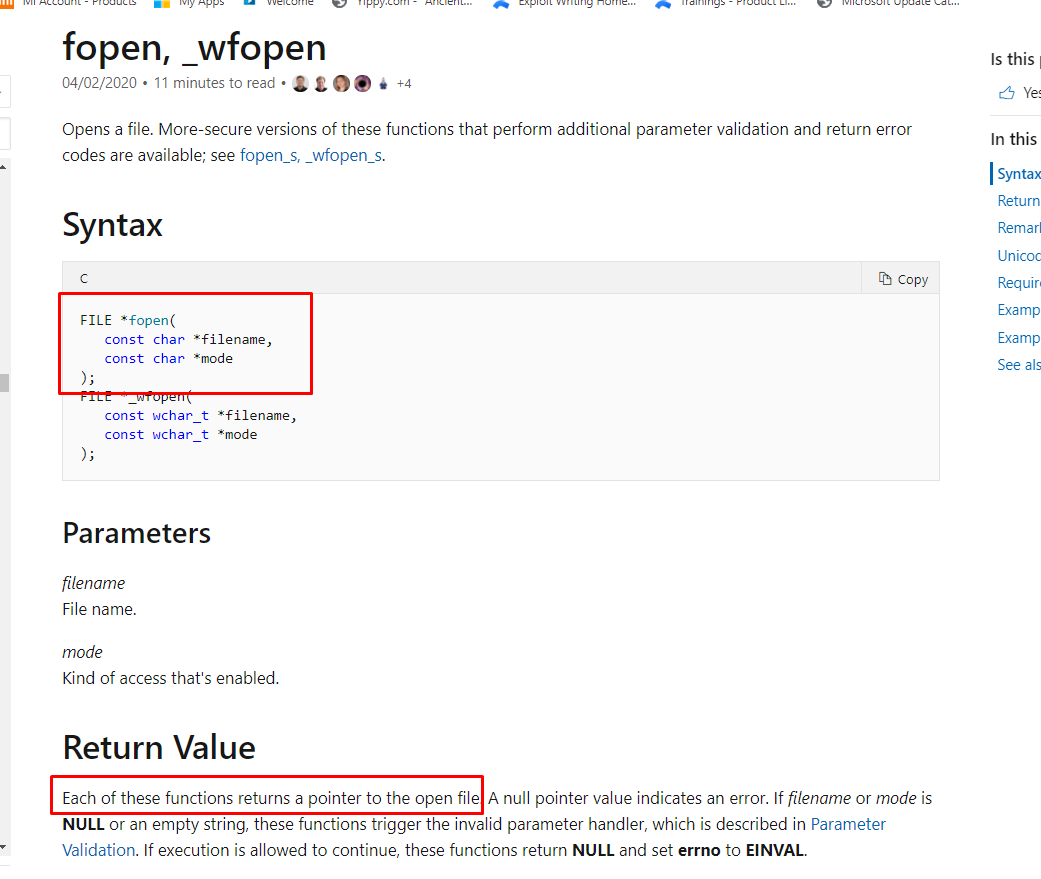
Luego intenta abrir un archivo **fichero.dat**, si no existe en la carpeta, nos saca del programa, así que habrá que crearlo también en la misma carpeta donde está el ejecutable.

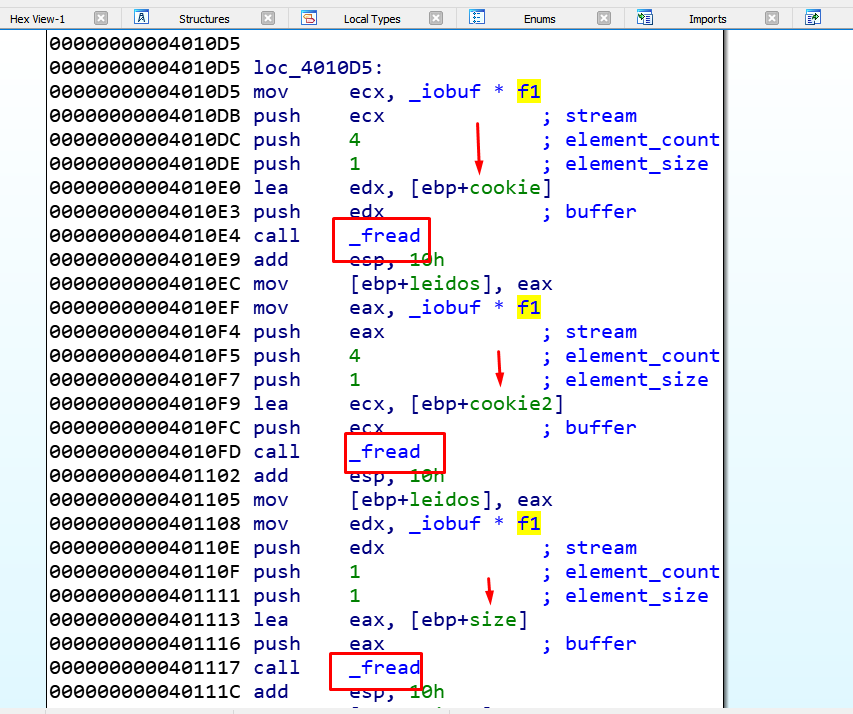




Vemos que **FILE** y **\_iobuf** son lo mismo no nos detendremos por ahora en eso.

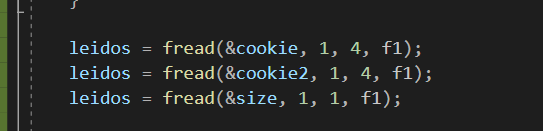
La cuestión es que trata de abrir el **fichero.dat** y guarda el puntero al mismo si lo puede abrir, en la variable **F1** que es global.

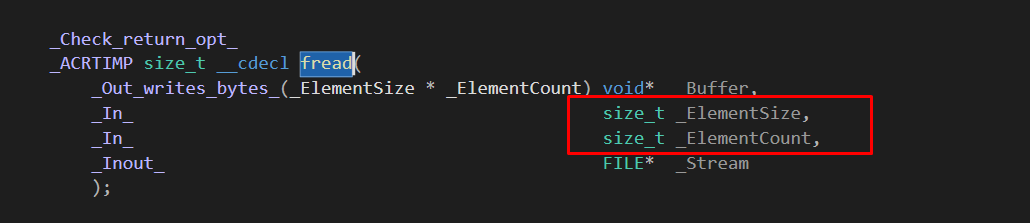




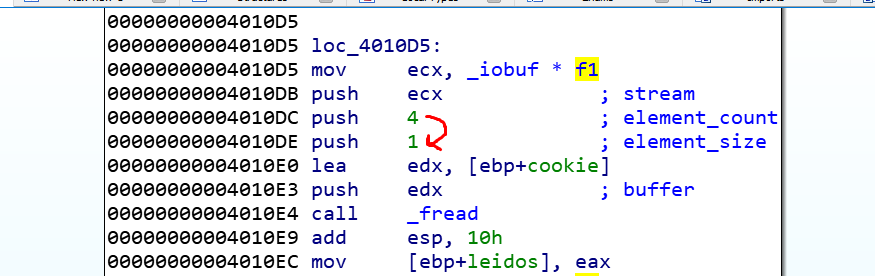
Luego lee tres valores del archivo, un dword llamado **cookie**, otro llamado **cookie2** y un byte llamado **size**.

Vemos que coincide en el primer y segundo llamado lee 4 bytes y en el último 1 byte.

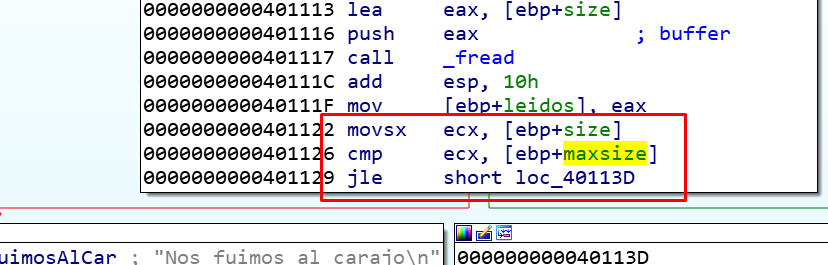




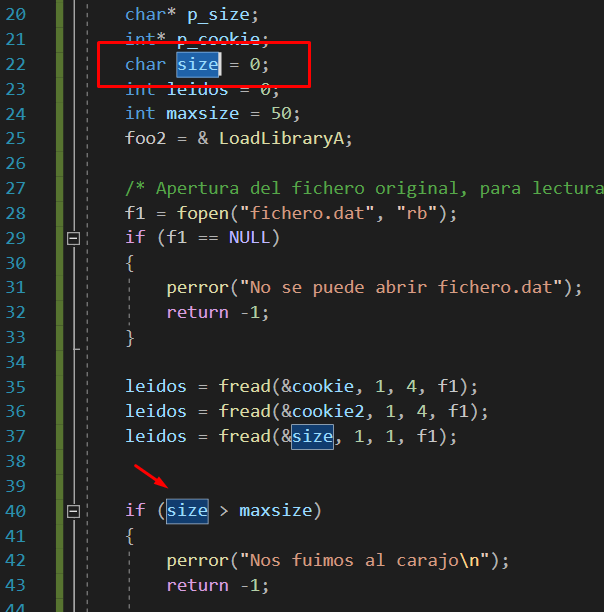
La cantidad de bytes que leerá es la multiplicación del **\_ElementSize** y el **\_ElementCount**, aunque están definidos al revés en el ejercicio, igualmente leerá 4 las dos primeras veces y 1 la última vez.



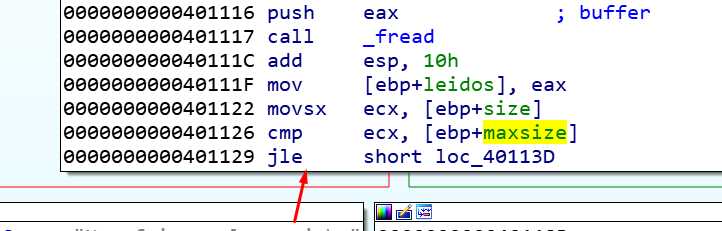
Debería haber sido el **element\_size=4** y el **element\_count=1**, pero bueno, el resultado será similar leerá 4 bytes o sea un dword.



Acá está la clave de la vulnerabilidad, al definir como **signed** la variable **size**.



Recordemos que, si no dice explícitamente **UNSIGNED**, el compilador la tomara como **SIGNED**.



Por ello usa un salto **JLE** que considera el **SIGNO**, si hubiera sido **UNSIGNED** utilizaría un salto **JBE** que no consideraría el signo y no será vulnerable.

La cuestión es que size al ser **SIGNED** al hacer **MOVSX** a **ECX** si le ponemos un byte negativo o sea **0x80** o mayor, lo completara con FFs y creara un **DWORD** negativo.

Al compararse contra **maxsize** que es 0x32 considerando el signo, cualquier valor negativo será menor que 0x32 positivo.

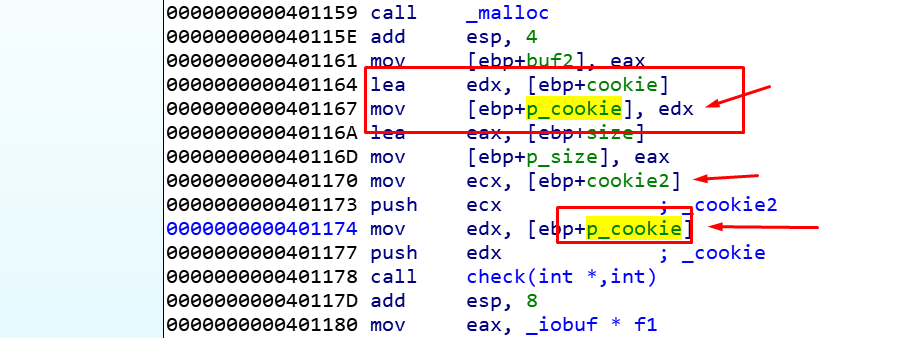
Por lo tanto, si por ejemplo el byte leído en **size** hubiera sido 0x80, al hacer MOVSX a ECX quedaría en **0xFFFFFF80** que considerando el SIGNO es negativo.

En Python si quieren saber a qué valor negativo corresponde, usando esta línea.

print ("VALOR NEGATIVO = -0x%x"%(0xFFFFFFFF- **0xFFFFFF80** +1))

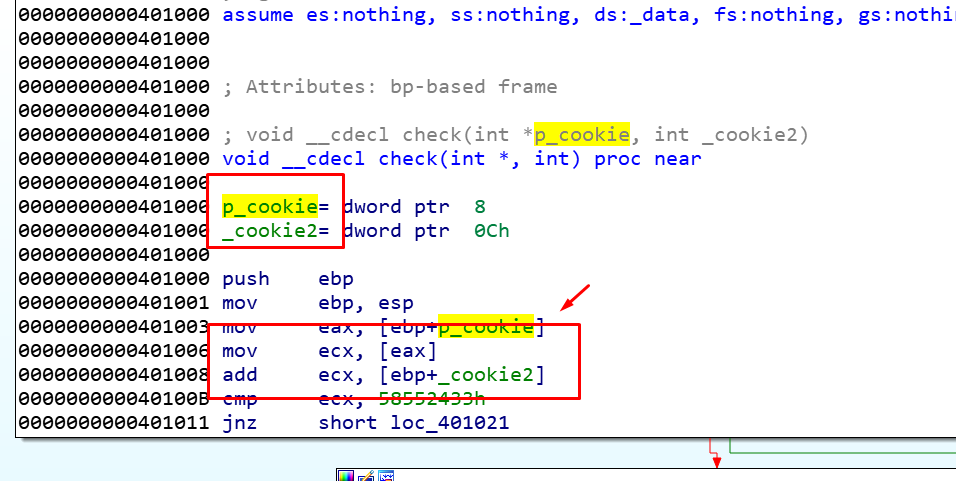


Bueno es el **-0x80** y como valor negativo es menor que el positivo **0x32** y pasara el chequeo lo cual no ocurriría si fuera UNSIGNED.

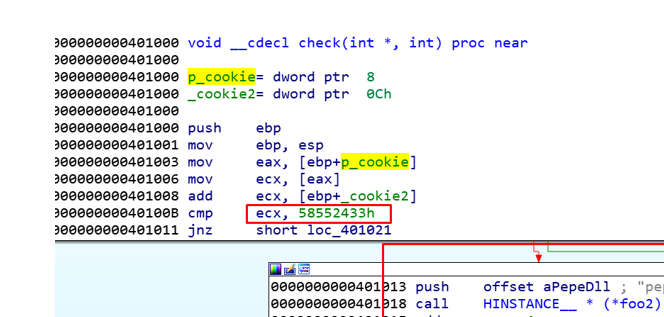


A la función **check** le pasa dos argumentos el puntero a **cookie** y el valor de **cookie2**

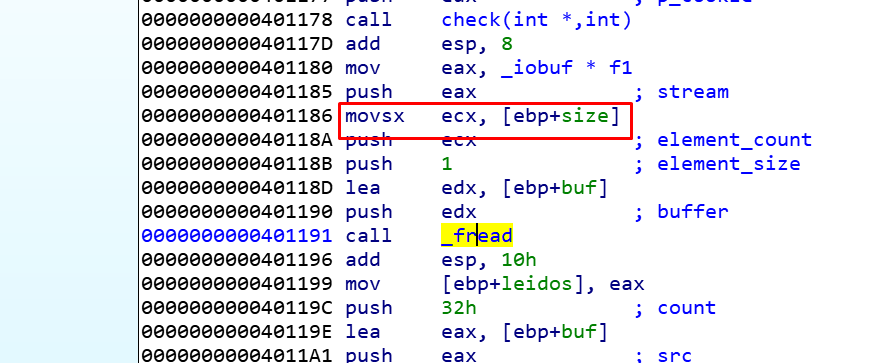
Vemos que dentro de la función check lo que hace recuperar el valor de **cookie** hallando el contenido de **p\_cookie**.



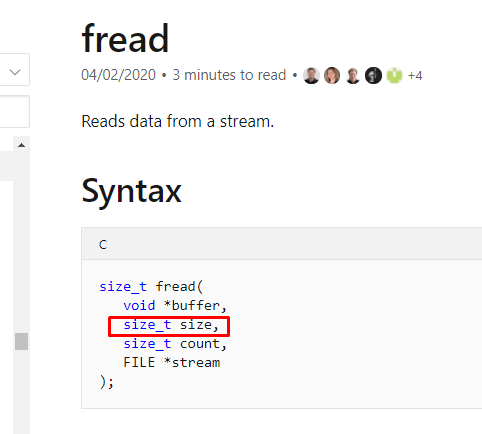
Renombre el argumento 1 a **p\_cookie** ya que halla el contenido que es **cookie** y lo suma a **cookie2** y lo compara contra una constante.



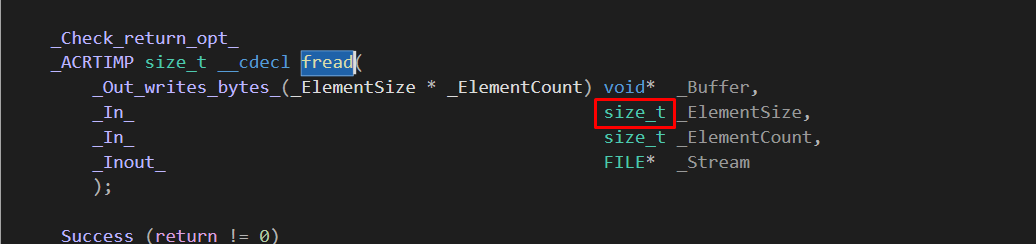
Así que para que cargue la dll debemos asegurarnos de que la suma de los primeros 4 bytes del archivo mas los segundos 4 bytes del archivo deben dar **0x58552433**.



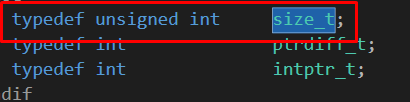
Allí se produce el overflow lee otra vez con MOVSX el valor SIGNED **size** y vuelve a ser **0xFFFFFF80**.



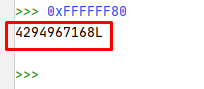
Pero la función **fread** no puede leer una cantidad de bytes negativa del archivo, no podes decirle léeme **-0x80** bytes, tomara el valor como UNSIGNED lo vemos que el argumento es del tipo **size\_t**.



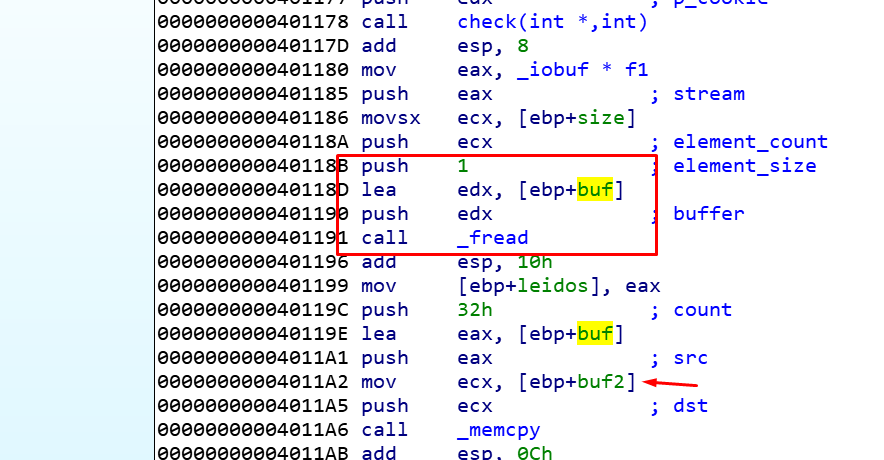
En 32 bits es lo mismo que **unsigned int**.

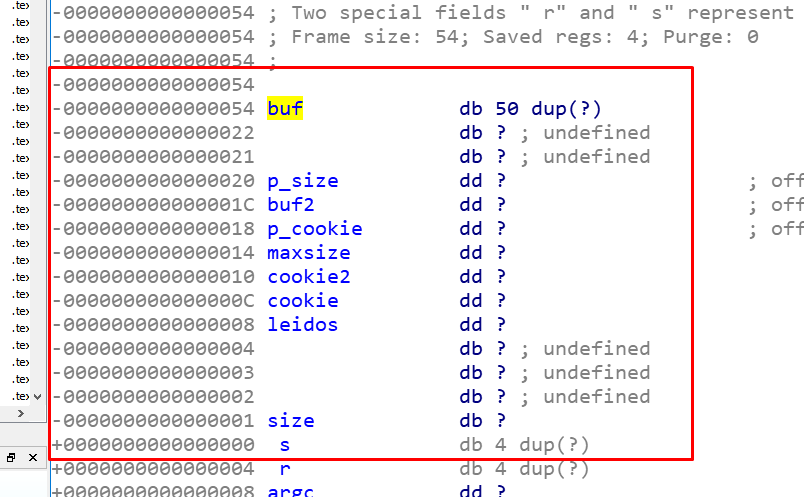


Por lo tanto, leerá del archivo hasta **4294967168** bytes.

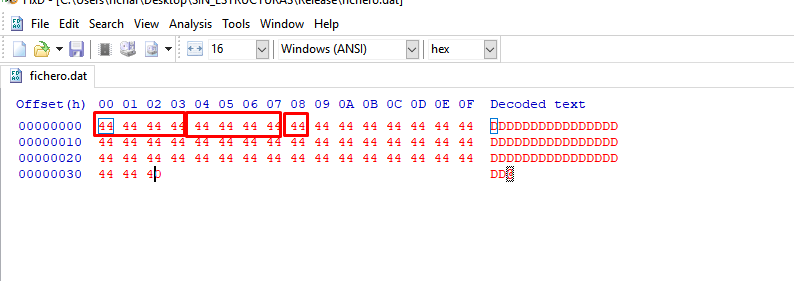


Por suerte **fread** lee la cantidad del bytes que el archivo tiene, si no encuentra mas vuelve sin dar error y desborda el buffer **buf** que esta en el stack, hay otro buffer **buf2** creado en el heap de 32 bytes que más abajo donde copiara desde **buf** pero no es necesario llegar hasta allí, explotara en el **fread** leyendo al **buf**.



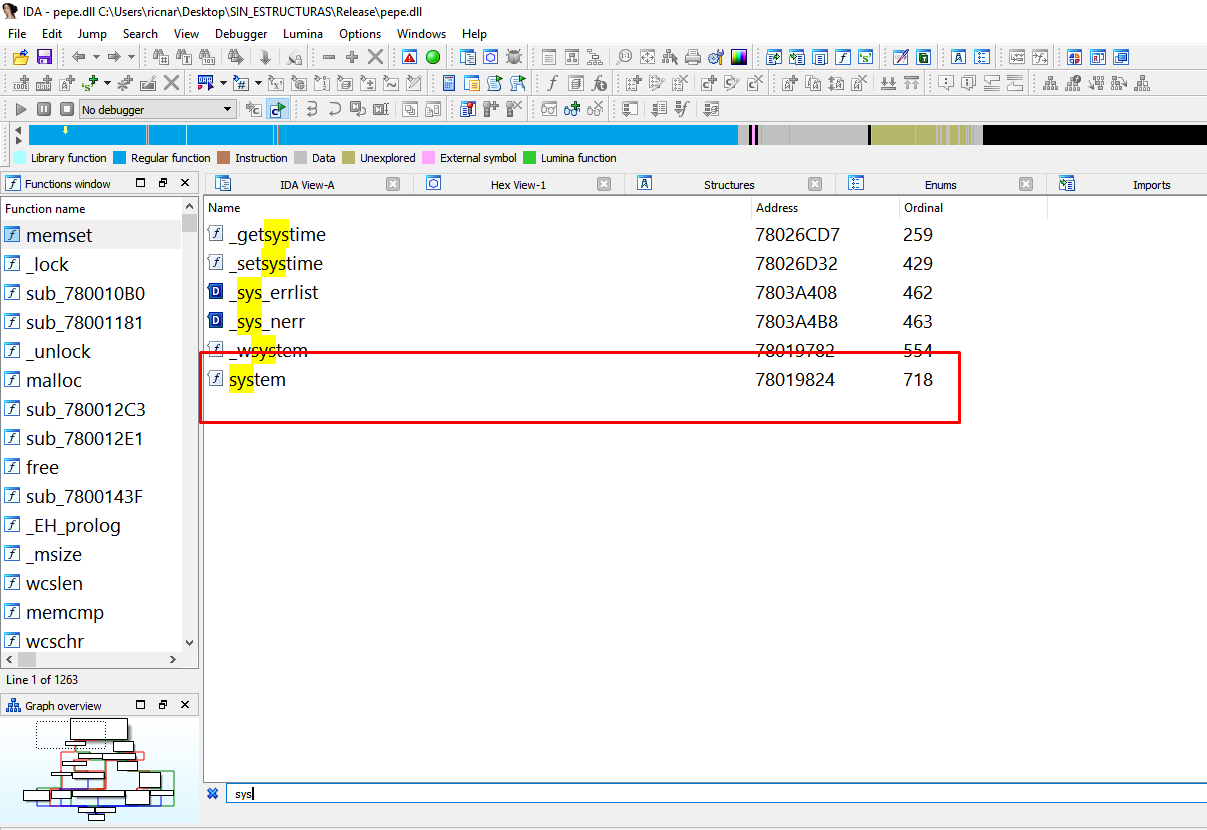


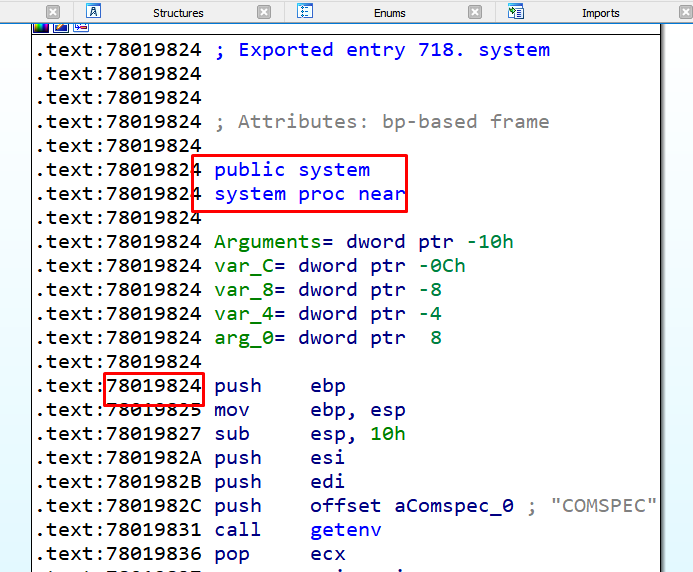
Hay que tener cuidado porque **buf** está arriba de todo en el stack y si queremos llegar hasta el **return address** pisándolo para evitar usar el **seh chain**, habrá que tener cuidado con el valor que pisamos las variables que hay debajo sobre todo **buf2** ya que es el destino del siguiente **memcpy** y puede crashear.



Así que debemos poner los primeros 4 bytes, que sumados a los segundos 4 bytes den el valor constante **0x58552433**, el siguiente byte debe ser **0x80** o mas para que sea negativo y el resto llenar con la cantidad de bytes que crean necesario hasta pisar el **return address** eso se los dejo a ustedes.

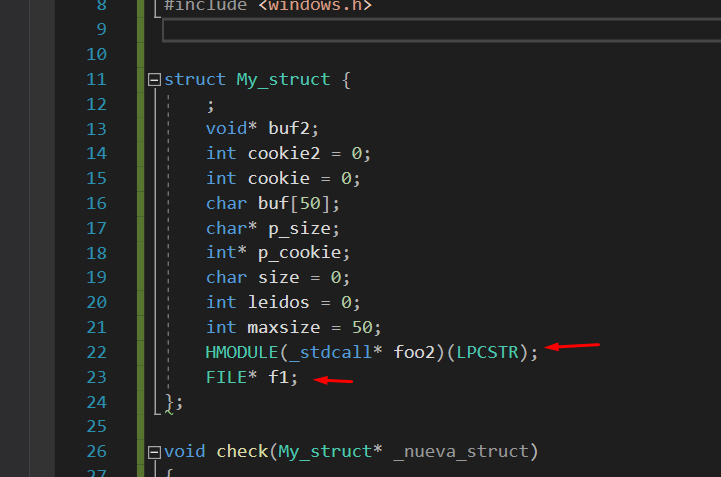
También la pepe.dll tiene la función **system** embebida que se puede llamar directamente pues no tiene randomizacion por lo cual será fácil ejecutar la calculadora con eso.

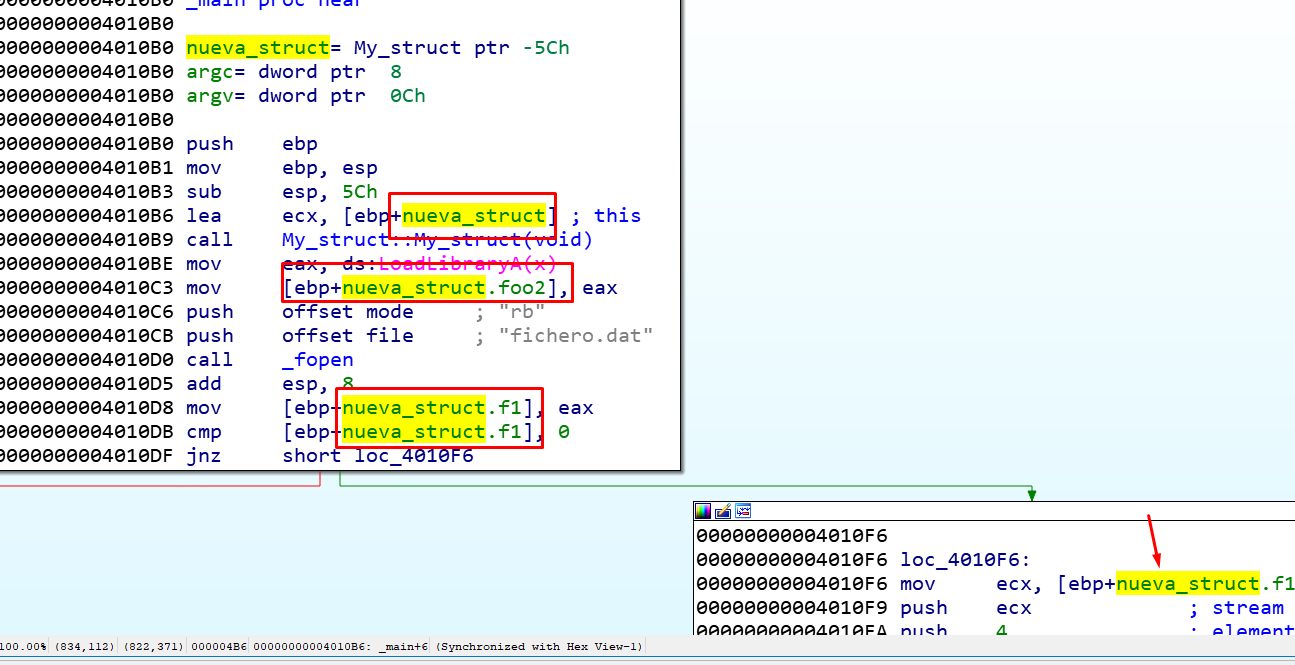




Así que si cargo el módulo pepe.dll, podrán acceder a **system** fácilmente.

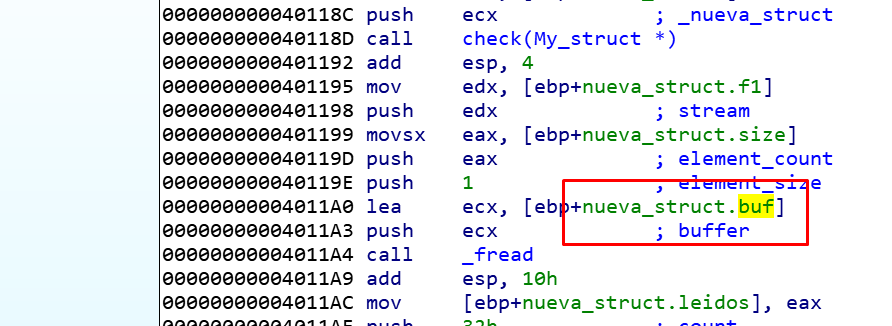
Se los dejo para ustedes, ahora modificaremos el ejercicio para hacerlo con estructuras, por ahora también con símbolos (por ahora jeje).



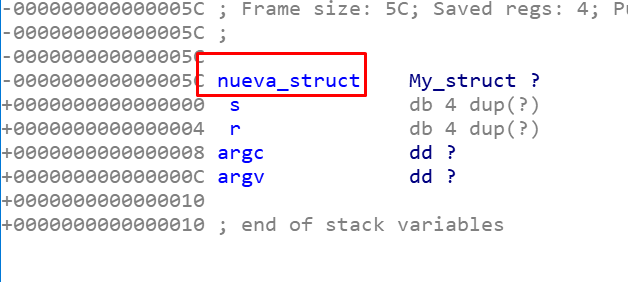


Vemos que es bastante similar cuando tenemos símbolos, detecta la estructura como una sola variable en el stack, y cuando llama a los campos lo hace como

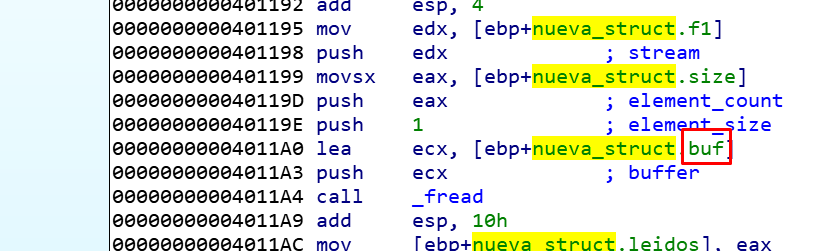
nueva\_struct.**nombre\_del\_campo.**

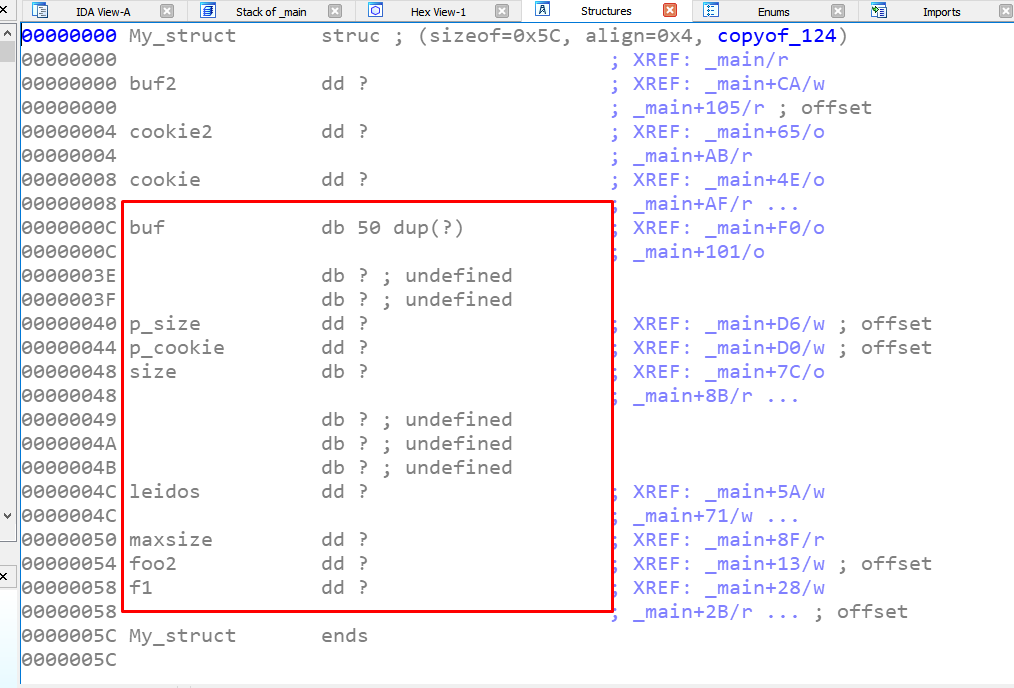


Ese será el destination y como **nueva\_struct** esta en el stack pues será similar.



Si miro dentro de la estructura haciendo doble click en buf.





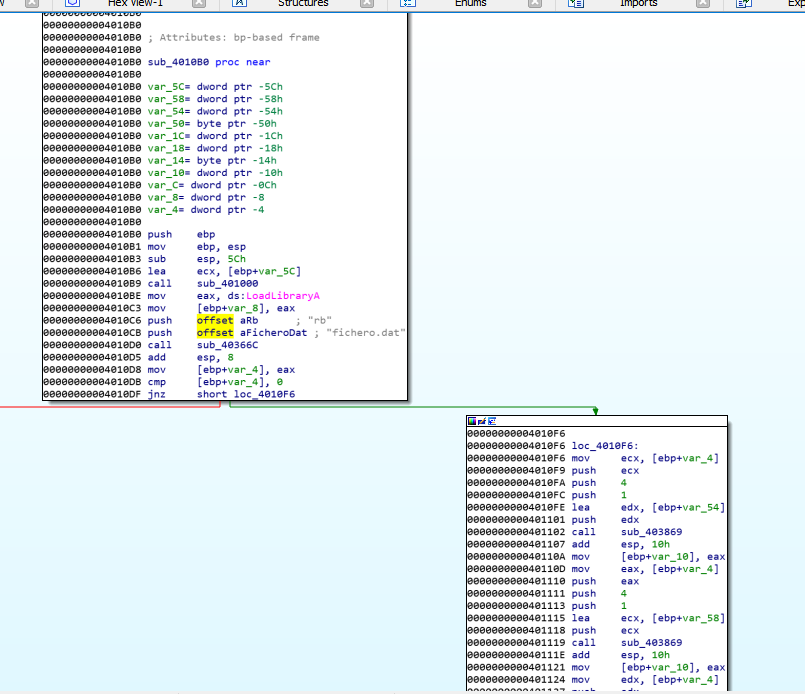
Puedo ver la cantidad de bytes que tengo que pisar para llenar la misma y debajo estaráel **stored ebp** y abajo el **return address**.

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

Ahí está el CON\_ESTRUCTURAS con el código fuente.

Sería bueno que los resuelvan ambos en que no tiene estructuras y el que tiene, asi en la próxima parte vemos como se reversea cuando no tiene símbolos, jeje.

Ahí se acabo la alegría lo haremos sin símbolo y como si no supiéramos nada jeje.



Bueno hasta la parte 20

22/08/2020

Ricardo Narvaja