# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 64.

Contents

[INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 64. 1](#_Toc40965951)

[Reversing para el desafío de Scada 1](#_Toc40965952)

[CVE-2013-0657 1](#_Toc40965953)

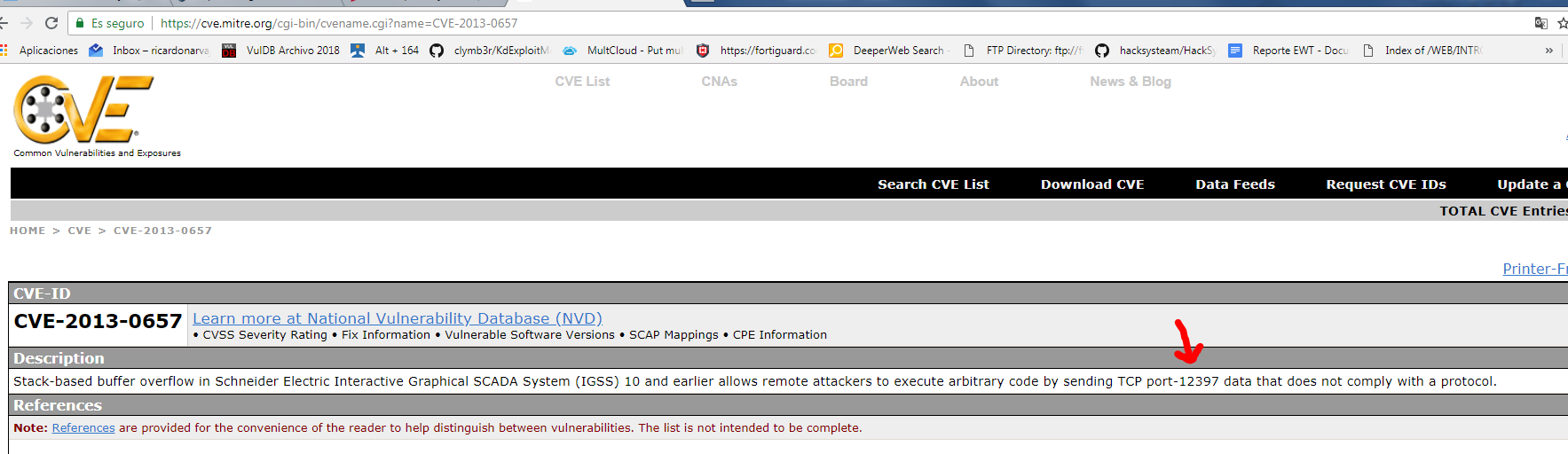
## Reversing para el desafío de Scada

En los grupos de telegram hemos puesto un desafío, que se refería a reversear el CVE siguiente, para practicar con un software real.

## CVE-2013-0657

<https://ics-cert.us-cert.gov/advisories/ICSA-13-018-01>

Allí está el link, todas las versiones 9 y 10 son vulnerables salvo a las que se le ha aplicado el parche.



La versión 9 vulnerable se puede bajar de acá

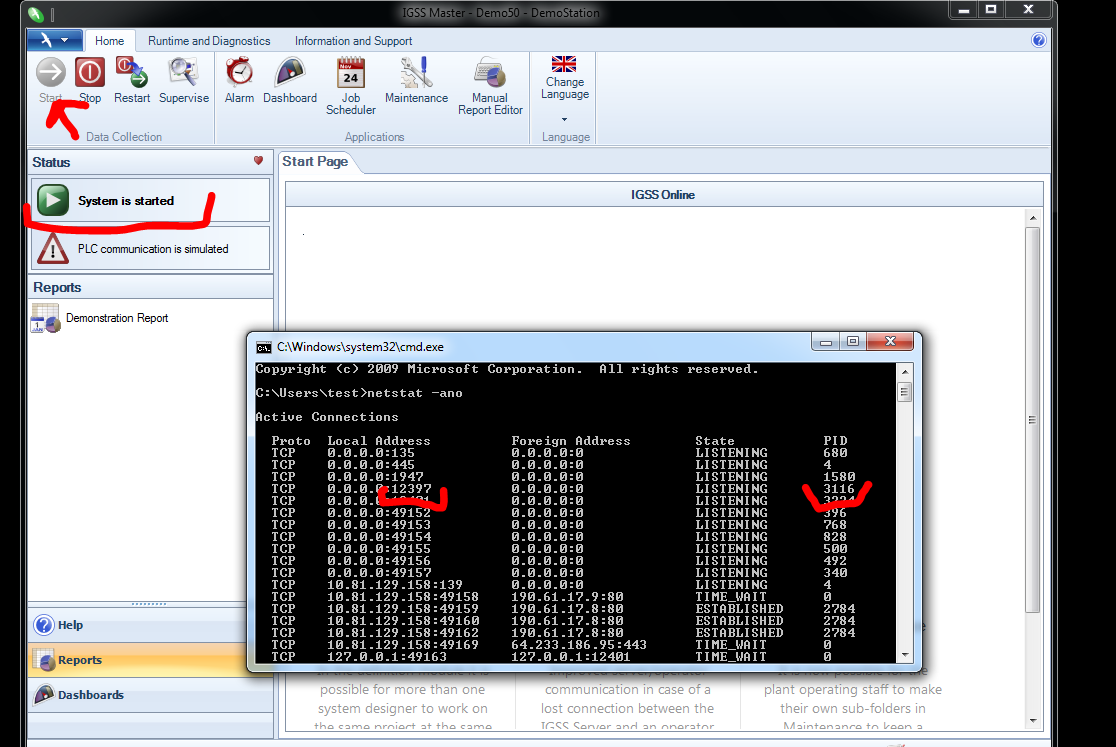
<https://drive.google.com/file/d/1uVdJjo7PZz5JvOO8f-6_2ZlmpkrtxwgI/view?usp=sharing>

Y el parche de acá

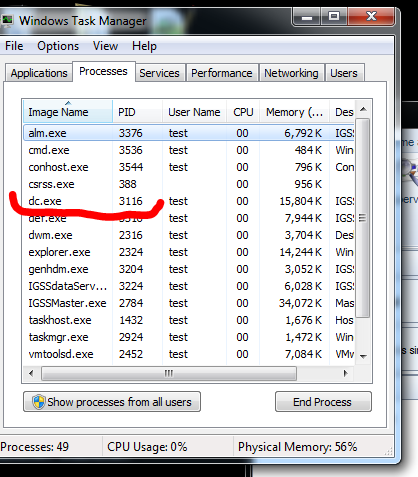
http://igss.schneider-electric.com/igss/igssupdates/v90/progupdatesv90.zip

Allí dice que el vulnerable, es el ejecutable que escucha en el puerto TCP-12397, si lo instalamos veremos cuál es el ejecutable que escucha en ese puerto.

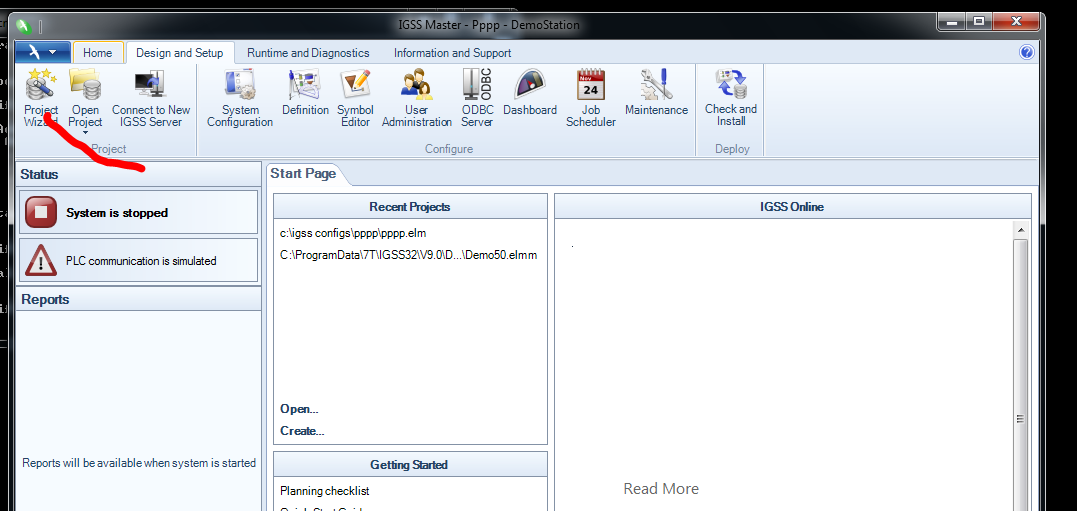
Cuando lo instalo, elijo la opción primero DEMO y luego FREE FOR 50 con eso lo pude instalar bien, y cuando le doy a start arranca, si no llega a arrancarles pueden borrar y crear un nuevo proyecto con menos elementos ya que es free hasta 50 elementos, pero aquí arranco.



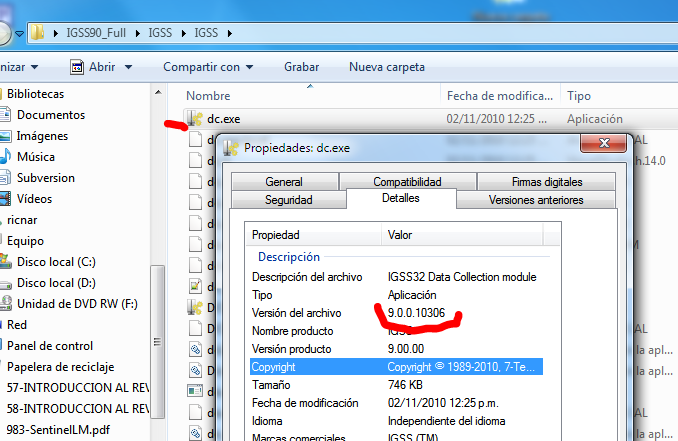
Veamos que el proceso es en mi maquina el 3116.



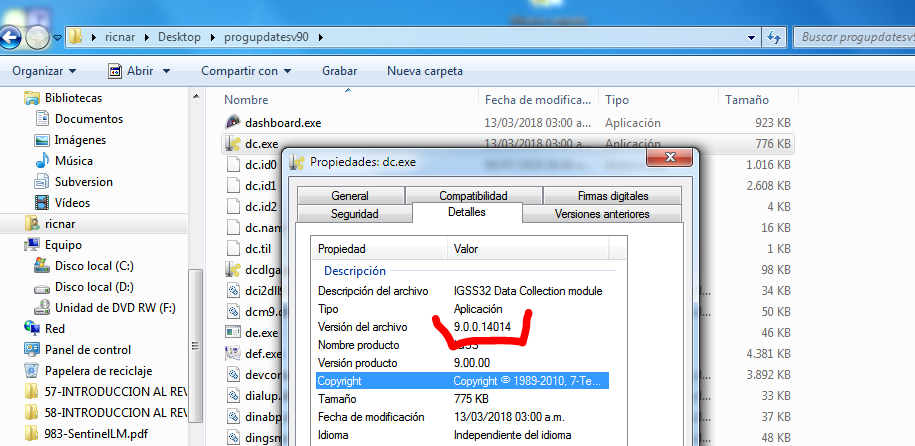
Para que quede bien funcional hay que ir a DESIGN AND SETUP quitar el proyecto viejo y crear uno nuevo con 50 elementos o menos y ponerlo a correr, con eso, estará perfecto.



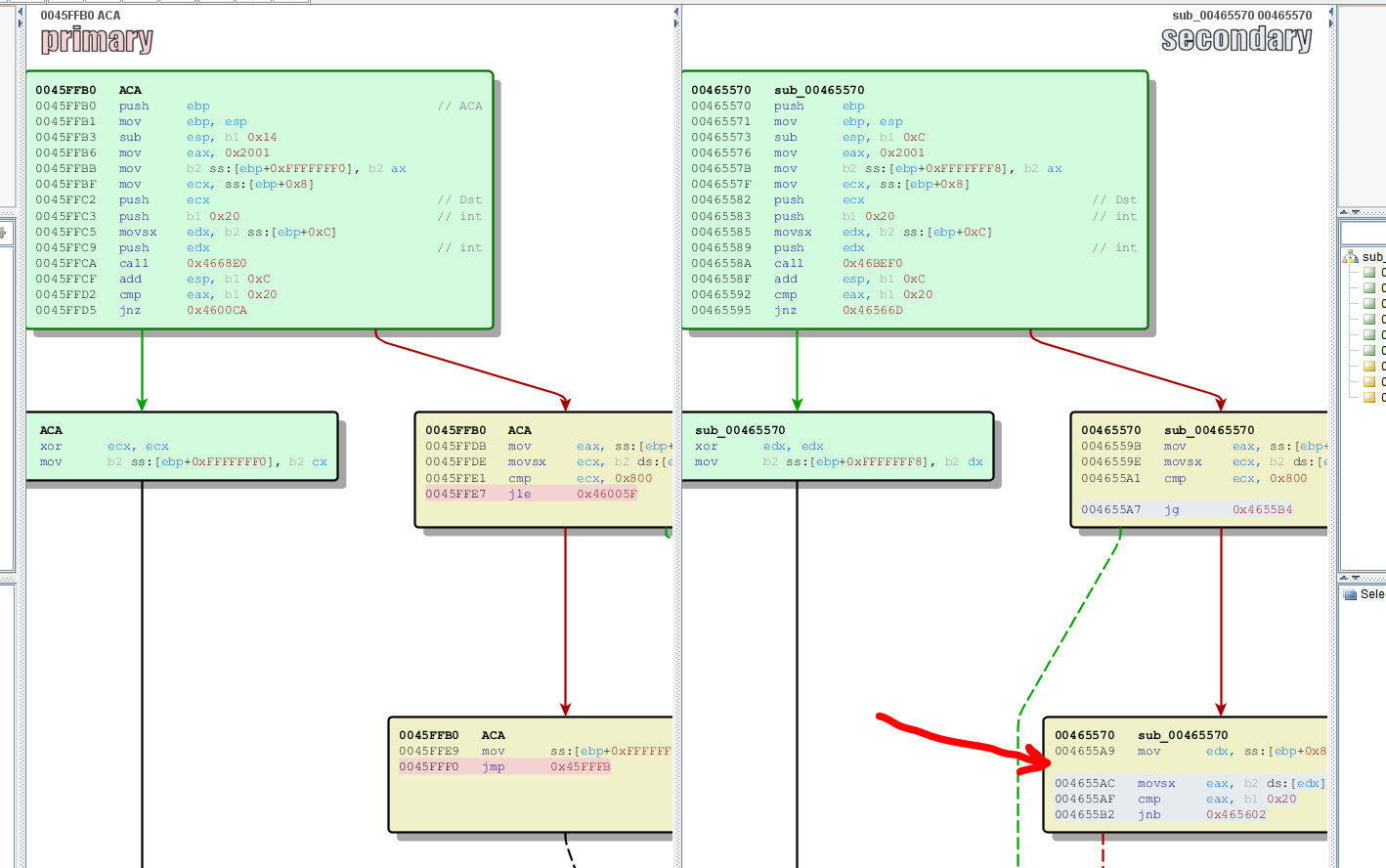
Ya tenemos por donde pueden venir los tiros, hagamos un diff entre los dos dc.exe el original en mi caso es la versión



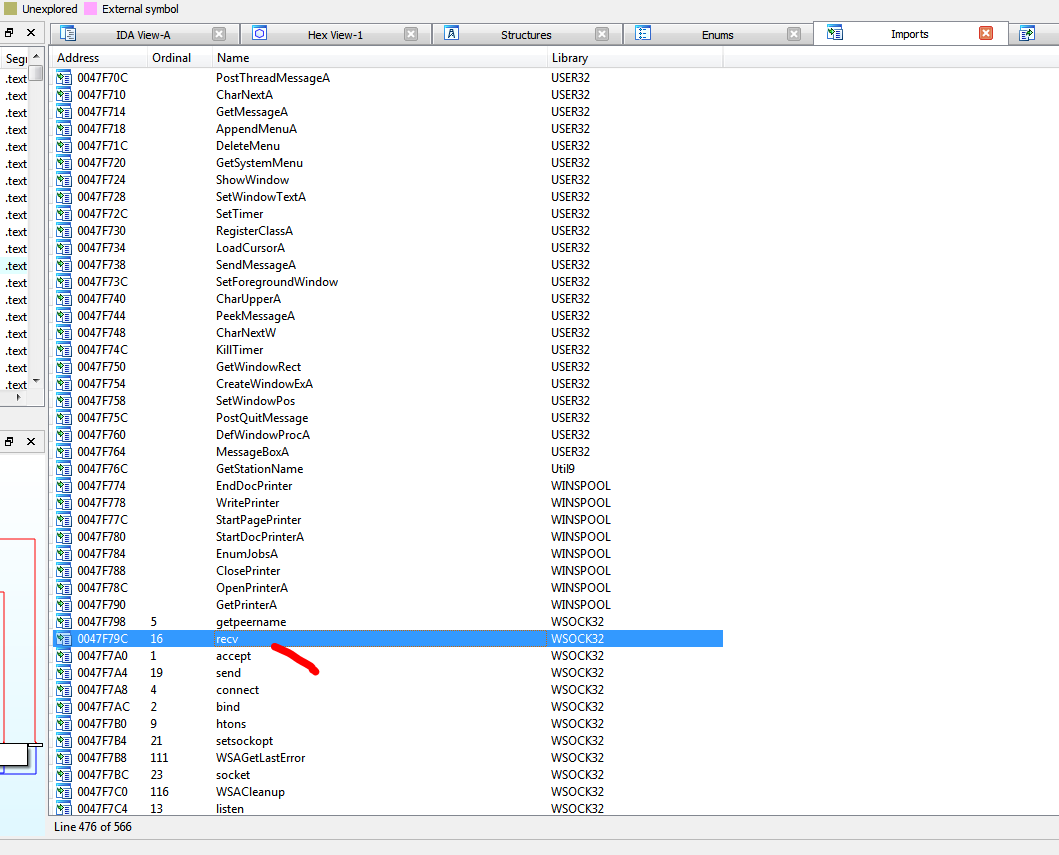
Y el que está dentro del parche

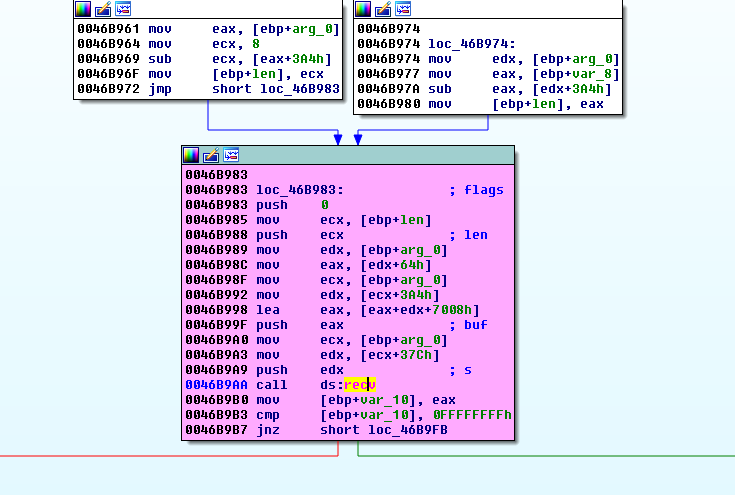


Hay muchas funciones cambiadas en el dc.exe con paciencia buscando, no voy a analizar una por una, pero aquí hay un stack overflow, con la flecha se señala el parche.

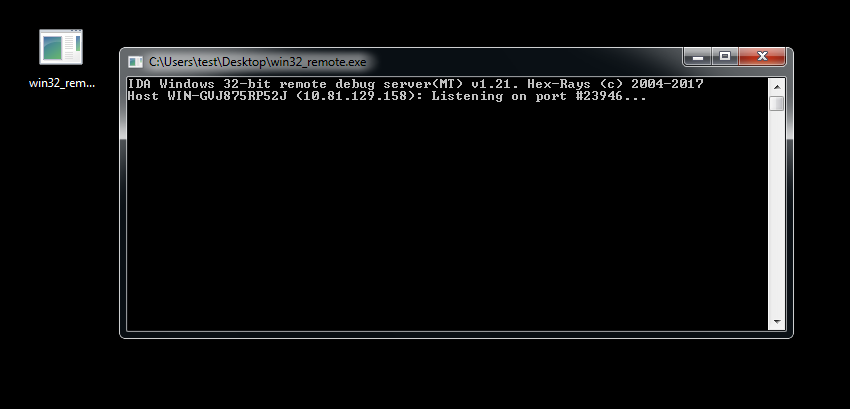


Buscando entre las funciones importadas recv, vemos que está allí y que es llamada de un solo lugar

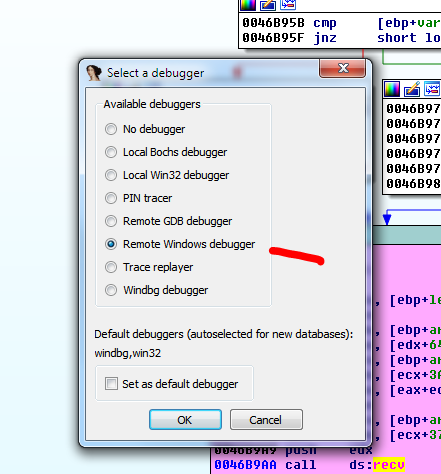




Busco el servidor remoto de IDA de 32 bits y lo copio a la maquina donde instale el programa y lo ejecuto como administrador.

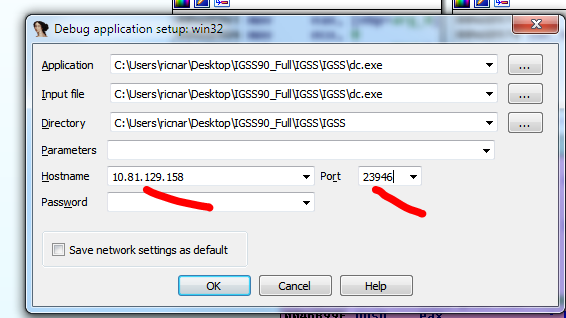


En el IDA cambio el debugger a Remote Windows debugger.

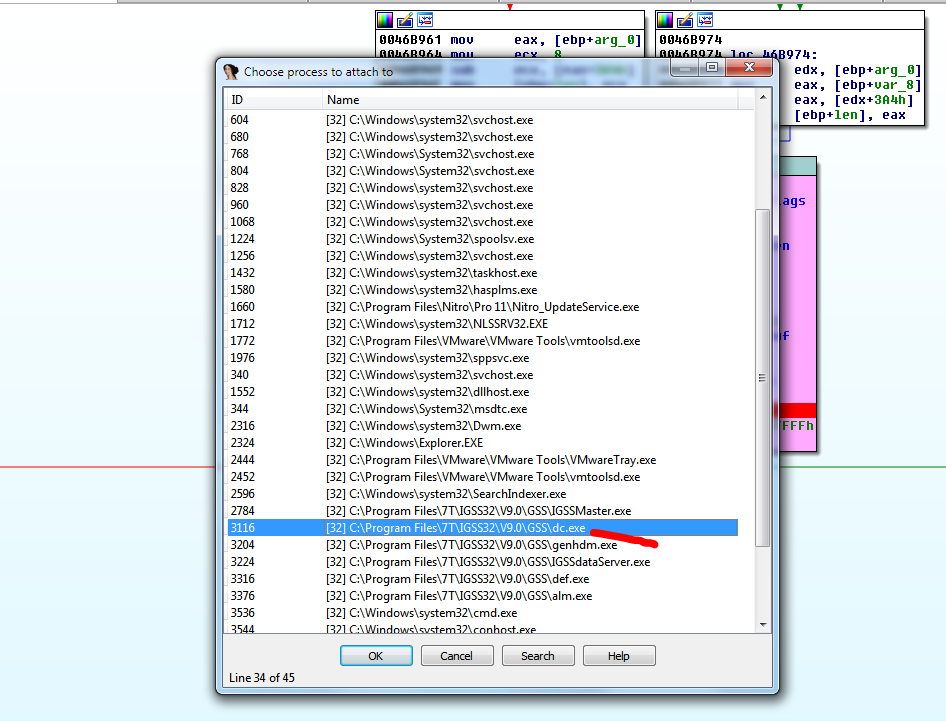


Y configuro las opciones del proceso (PROCESS OPTIONS) con la ip del target y el puerto del mismo.



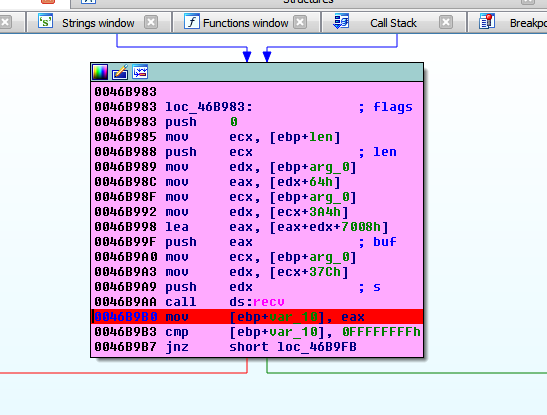


Cuando vamos a Attach Process, nos sale la lista de procesos de la maquina target, si la conexión entre ambas es posible (no hay firewalls o bloqueos etc)

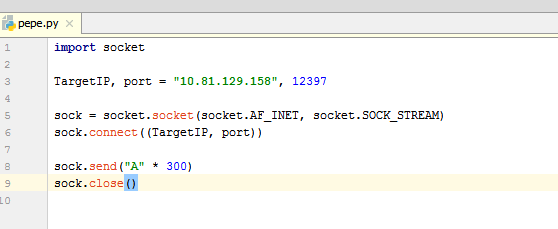


Ahí está dc.exe, lo elegimos y lo atacheamos.

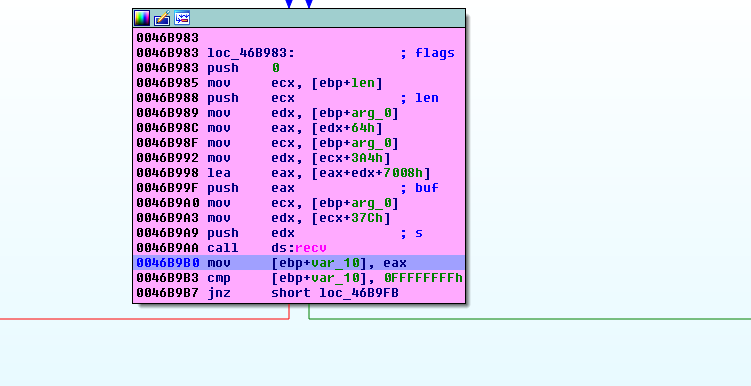
Para chequear que ese recv es el que usa, le puse un breakpoint en el mismo



Creo un scriptcito de Python que le envie datos al puerto 12397, que es el puerto donde está escuchando el dc.exe, le mando 300 Aes solo para probar si para allí.

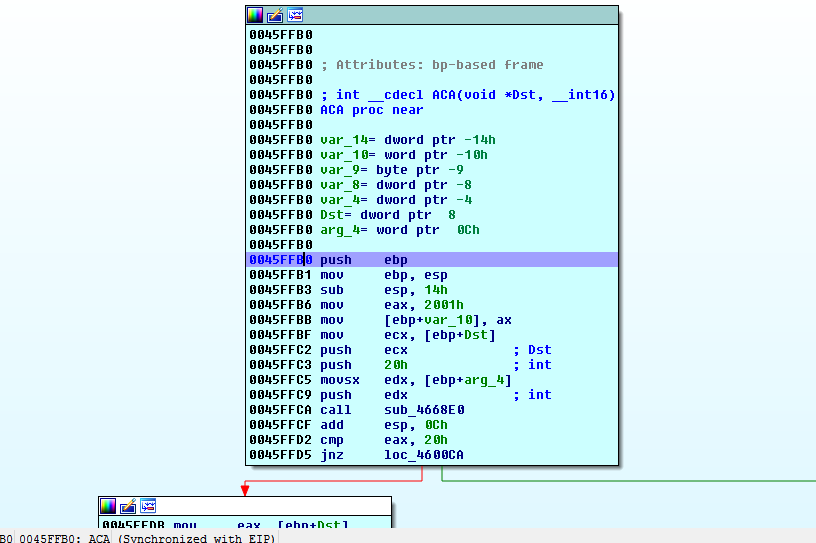


Vemos que para



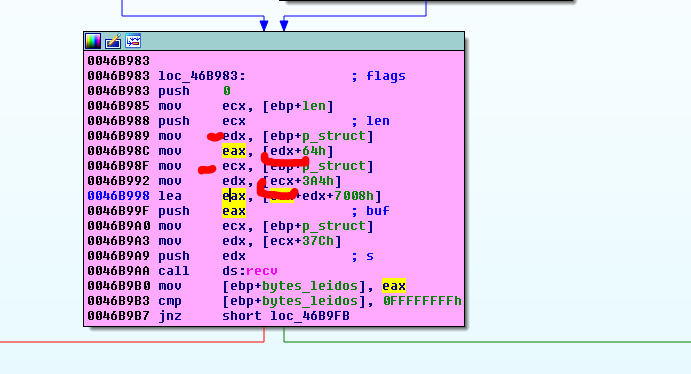
Vemos que la primera vez que para, solo recibe ocho bytes así que es obvio que habrá más veces que parara en el recv, no será la única.

Si le doy RUN sin reversear mucho veo que llega solito con ese paquete a la función vulnerable.



Igual con eso solo no pasa por el camino vulnerable, constantemente esta parando en la función vulnerable cada tanto, es necesario analizarlo bien.

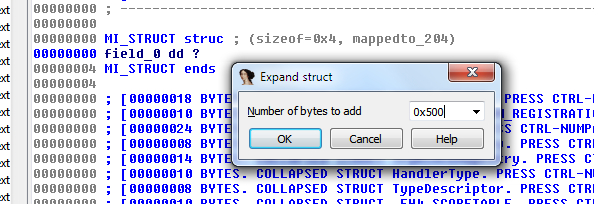
No tenemos símbolos, no tenemos nada, veremos hasta donde podemos llegar



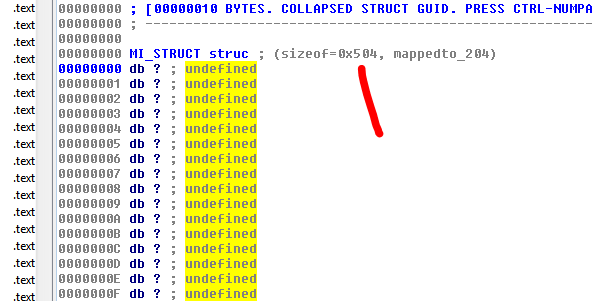
Vemos que hay el puntero a una estructura en esa variable que llame p\_struct, cada vez que se lee el valor luego siempre se accede a un campo de la estructura sumándole a esa dirección un offset.

Vemos que tiene offset grandes por ahí vemos 0x64, 0x3a4, 0x37c, así que crearemos una estructura de 0x500 bytes de largo si necesitamos más la agrandaremos, si es más chica no habrá problema.

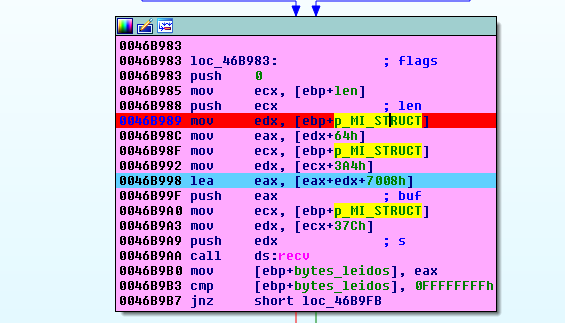
Cree una estructura con un solo dword y la expandí a 0x500



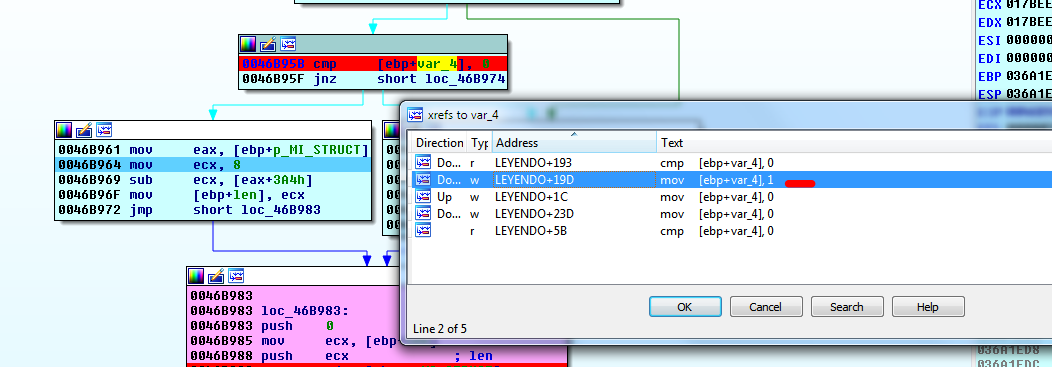
Ahí está, como ya había 4 bytes más 0x500 da en total 0x504 no hay problema.



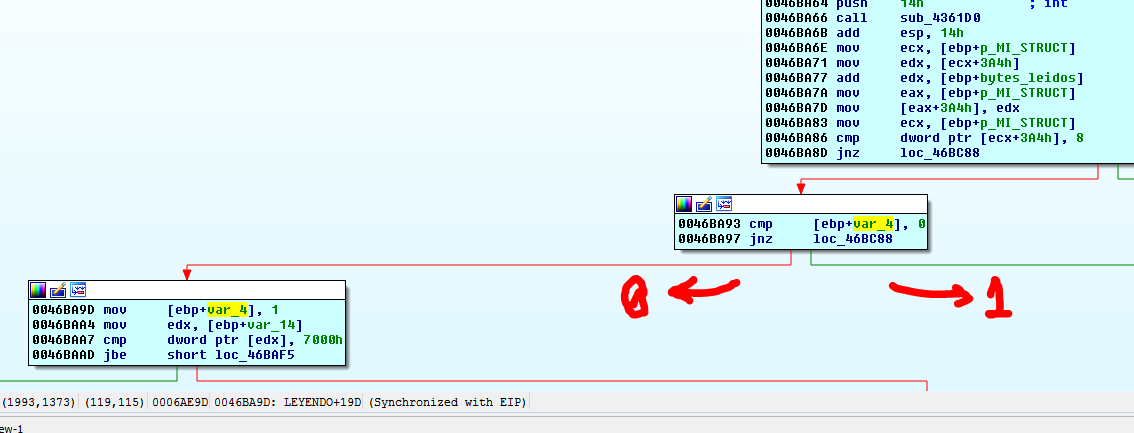
Renombro para que quede del mismo nombre



Vemos que la var\_4 es un flag que puede valer 0 o 1, hay un lugar donde se mueve el valor 1 allí.

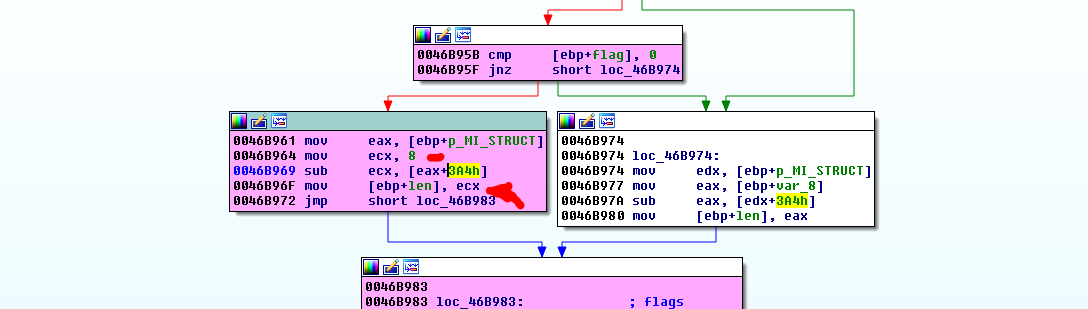


Según sea 0 o 1 antes de llamar al recv va por el bloque de la derecha o el de la izquierda.



Vemos que compara con 0 si es cero va y lo cambia a 1, y la próxima vez que pase ira por el camino 1, así que suponemos que el estado inicial del flag es 0 y pasa por allí y se pone en 1.

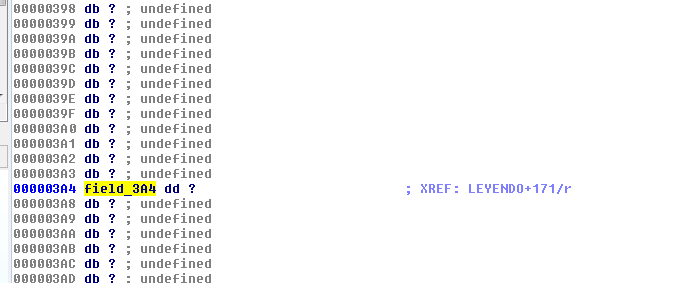
Así que en el primer paquete pasará por acá, antes del recv, mueve a ECX el 8 que era el len, antes le resta el valor del campo 0x3a4 de la estructura que al inicio será 0, ya que vimos que el len del primer paquete, debía ser 8.



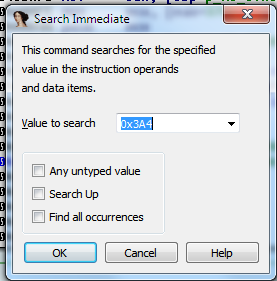


Si busco la constante 0x3a4 veo que allí guarda la sumatoria de los bytes leídos, así que renombro.

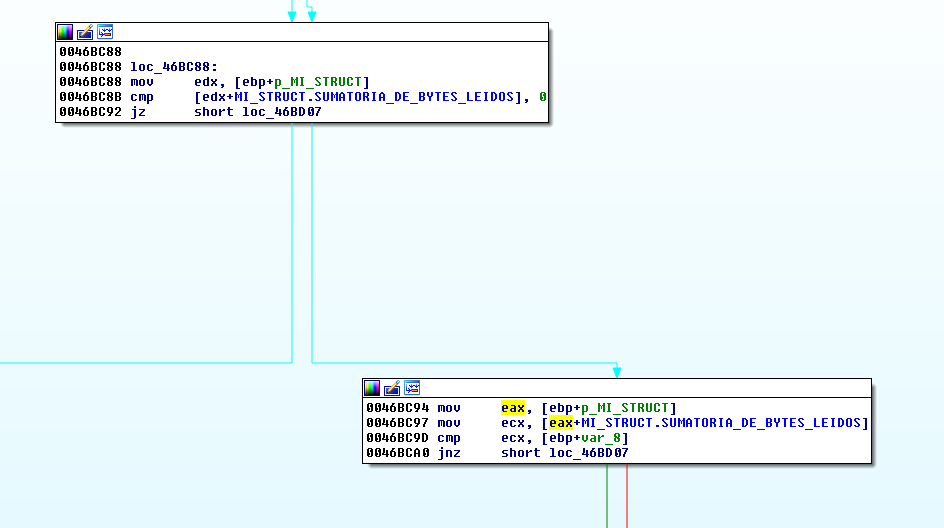
Voy en estructuras hasta el offset 0x3a4 de la que cree y apreto d hasta que sea un dword y renombro.



puedo buscar el offset



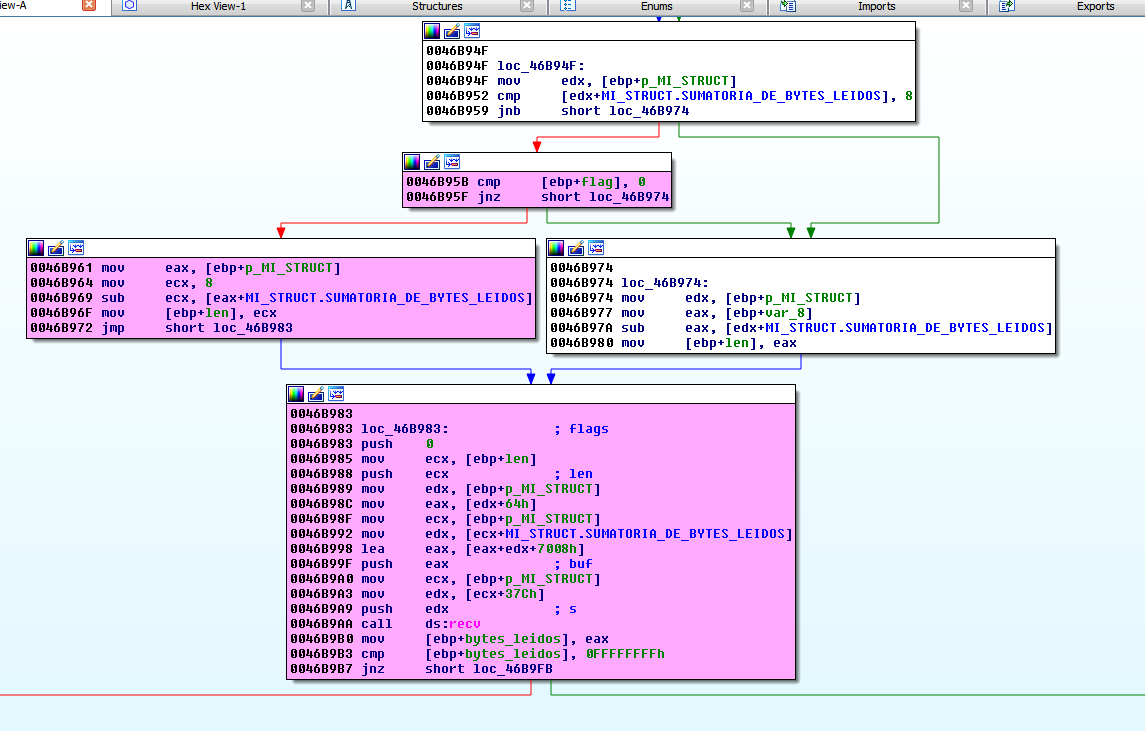
Y luego con CTRL+I ir buscando todas las veces que usa el mismo y renombrándolas a todas con el mismo nombre.



Pedo recomenzar desde el inicio de la función

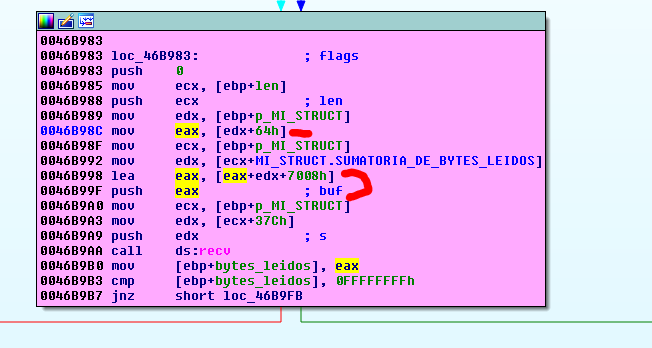


Ahora va quedando más lindo, al menos en la zona de la función que estoy trabajando

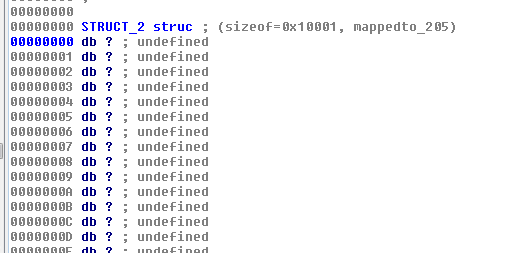


También vemos que la segunda vez que pase por el primer bloque de la anterior imagen al haber leído ya 8 bytes directamente ira por el camino de la derecha.

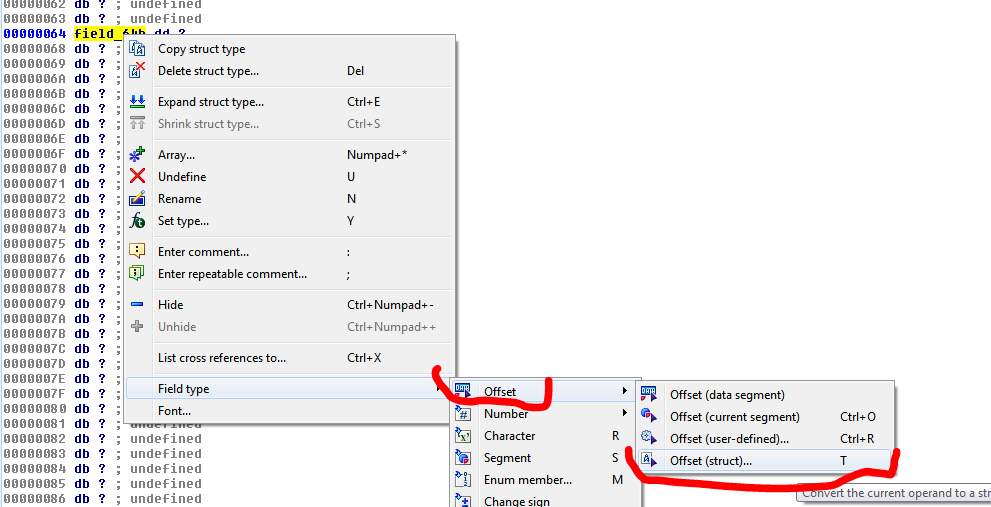
Vemos también que en el campo 0x64 hay un puntero a una segunda estructura que tiene el buffer dentro en su offset 0x7008 y que pivotea con edx que vale cero por ahora pues es la sumatoria de bytes leídos, al inicio es cero.



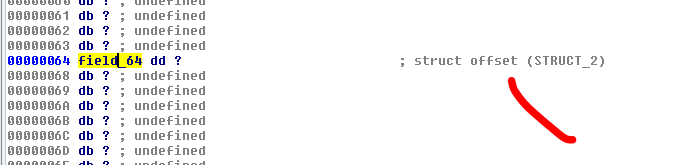
Creare la segunda estructura al menos de 0x10000 bytes no importa que me pase.

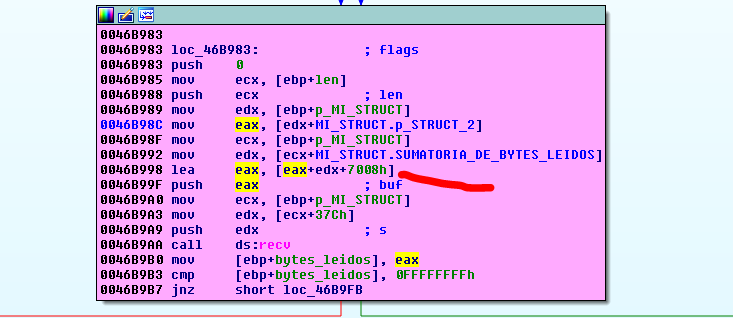


En el 0x64 de MI\_STRUCT está el puntero a STRUCT\_2, así que voy allí, apreto d hasta que se cree un dword y luego offset a una struct.



Elijo que sea STRUCT\_2 y ya queda

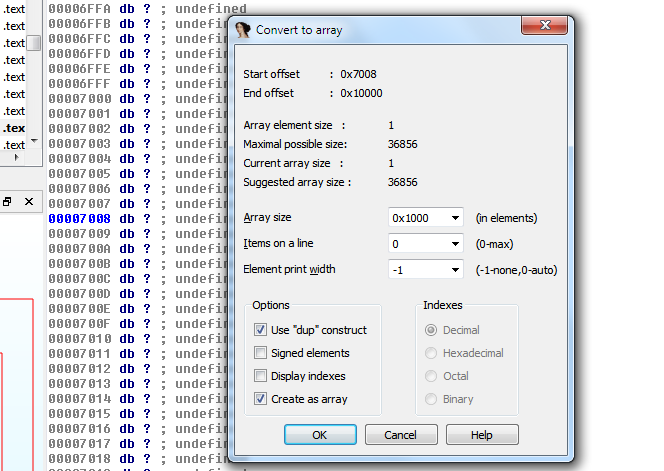


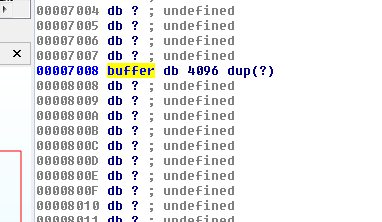


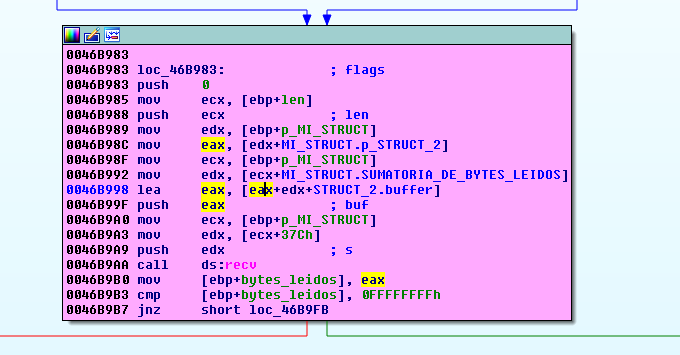
En el offset 0x7008 empezara el buffer donde guardara los datos del recv, como es un LEA no es un puntero a un buffer, sino que el mismo está dentro de la misma STRUCT\_2.

En estructuras voy al 0x7008 de la STRUCT\_2 y creo un buffer.

Le pongo tentativamente 0x1000 de largo para corregir hay tiempo.

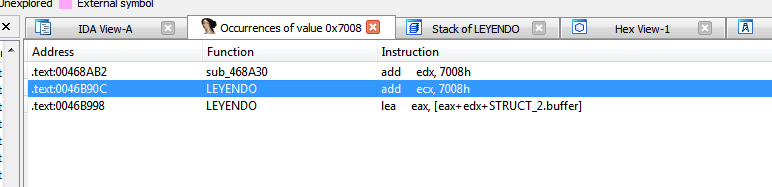




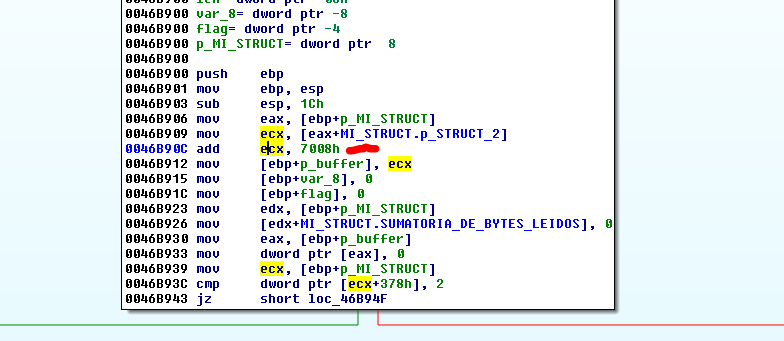


Ahora quedo mejor.

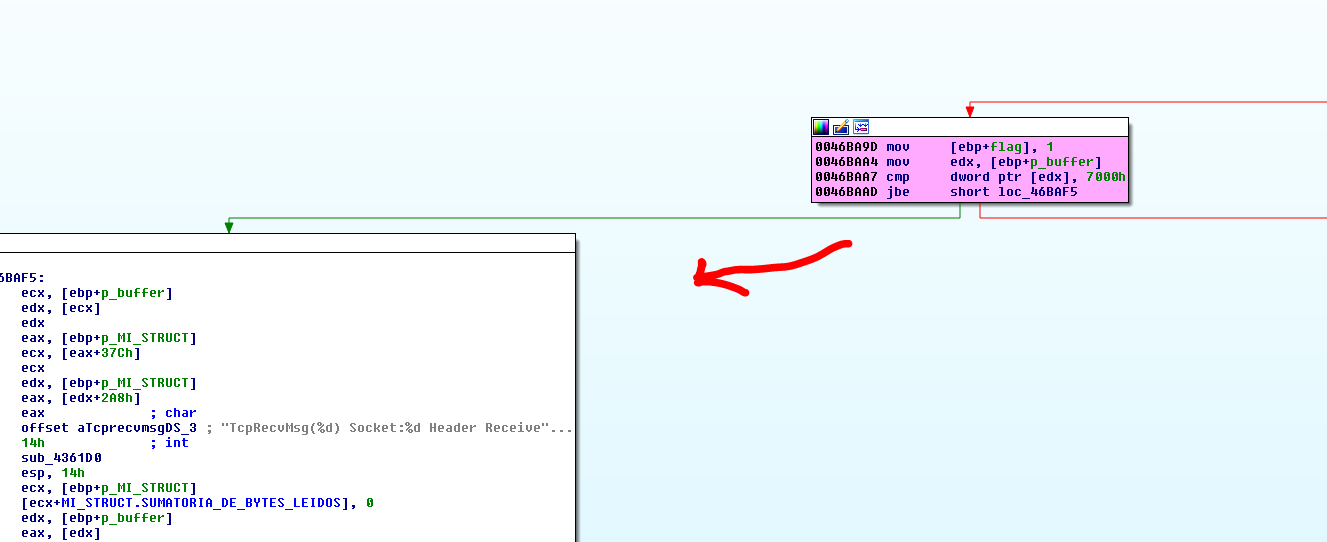
Si buscamos search for inmediate value el valor 0x7008 en todo el programa.



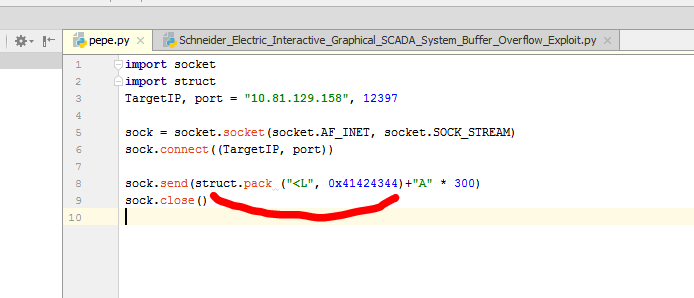
Vemos que hay dos lugares más donde lo usa, veámoslos.



Uno es al inicio de la misma función que lee, al puntero a la STRUCT\_2 le suma 0x7008 y eso le da un puntero al buffer donde guarda los bytes, así que renombro la variable donde guarda como p\_buffer.

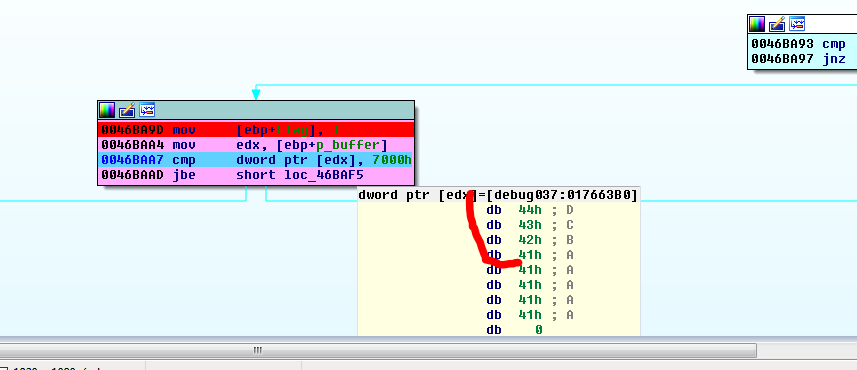


Allí vemos como busca el contenido de p\_buffer o sea el primer dword del mismo y ese valor debe ser menor que 0x7000, podemos chequear esto poniendo un Breakpoint en la comparación.

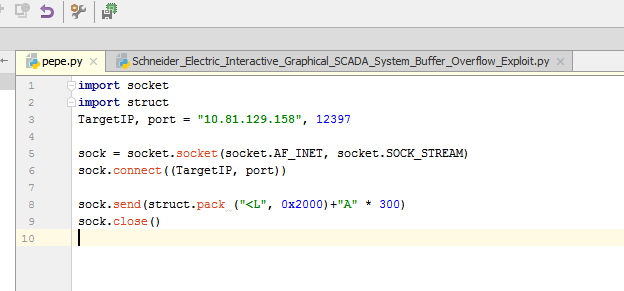


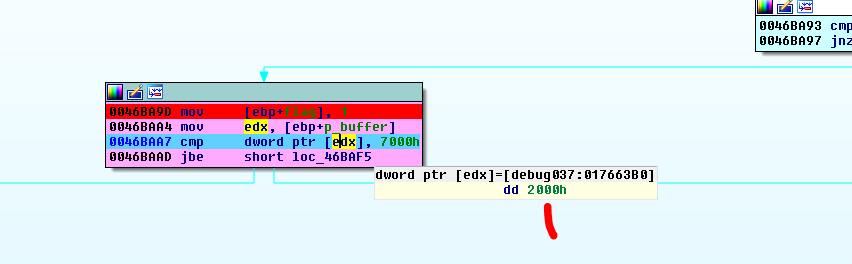
Cambio el paquete que envío para que los primeros 4 bytes sean diferentes. En este caso 0x41424344.

Así que de los 8 bytes que lee, testea si los primeros 4 son menores que 0x7000



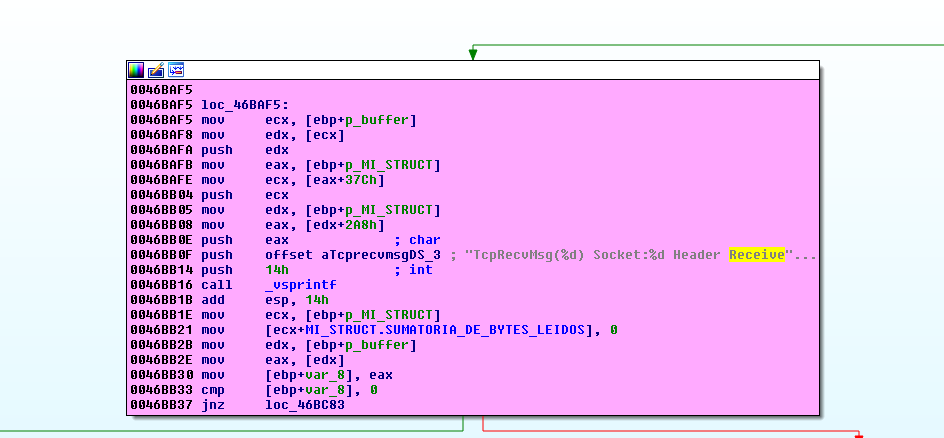
Lo cambiare al valor a 0x2000



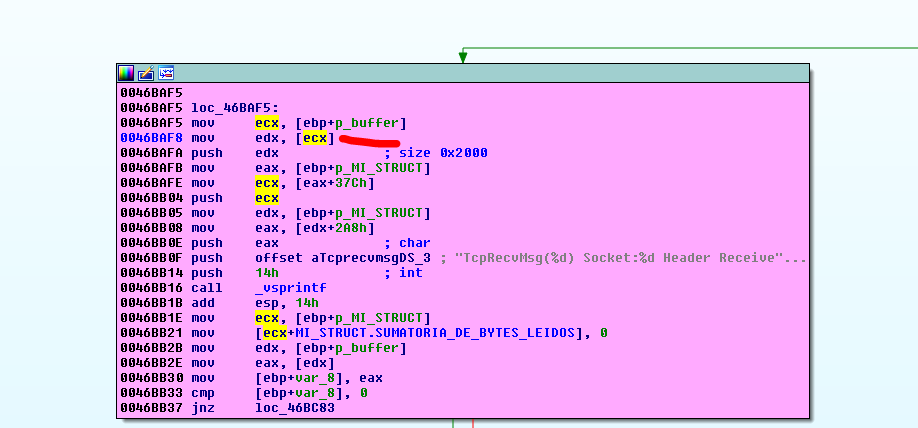


Ahora si

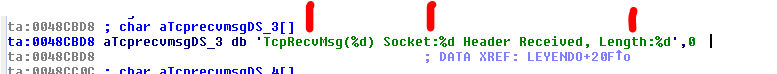
Así que seguimos por acá



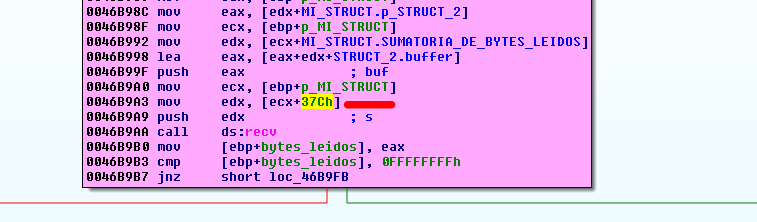
El contenido del p\_buffer es el size 0x2000

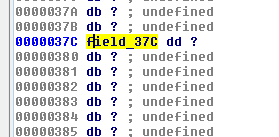


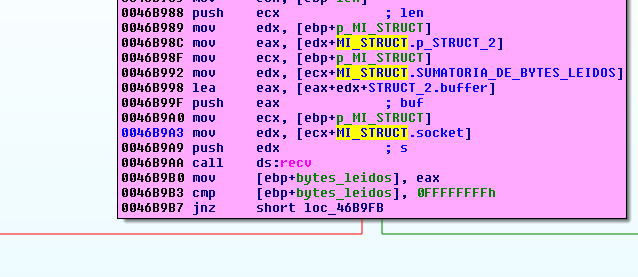
En el printf nos ayuda ver que hay tres valores que imprimirá Lenght, Socket y TcpRecvMsg, así que podemos ver si podemos renombrar algo con ellos.



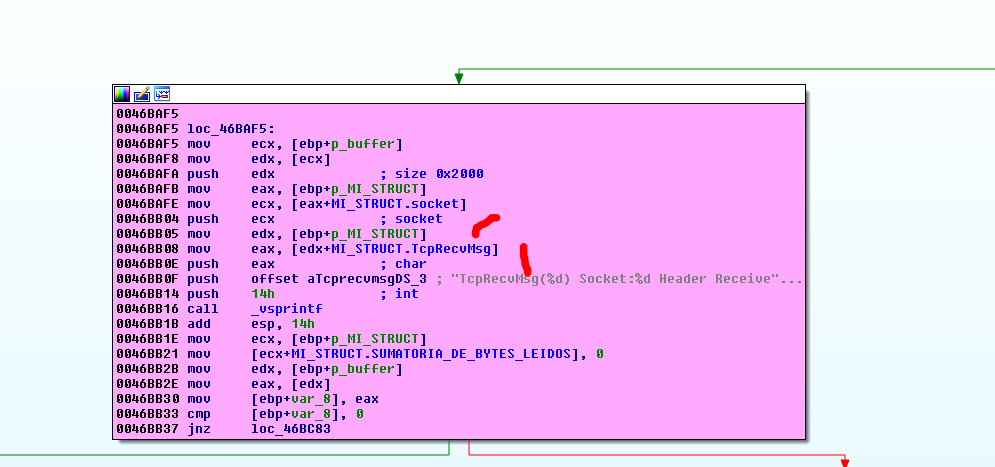
Recordamos que en el campo de MI\_STRUCT 0x37c estaba el socket, lo renombramos.



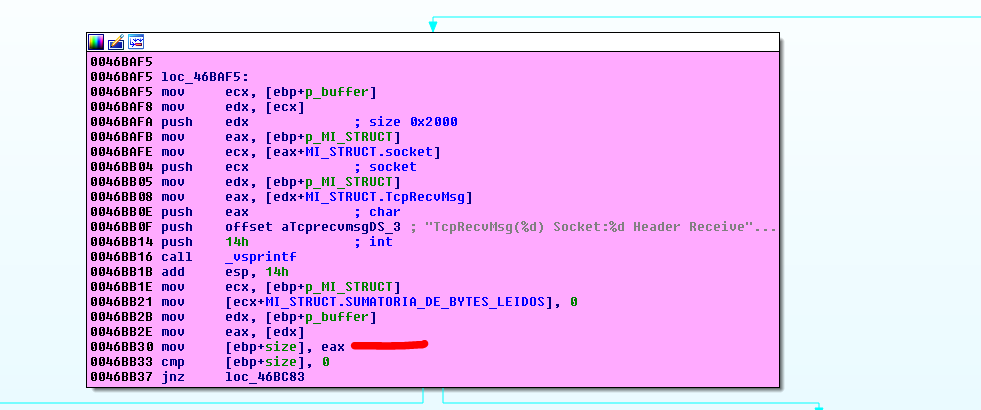




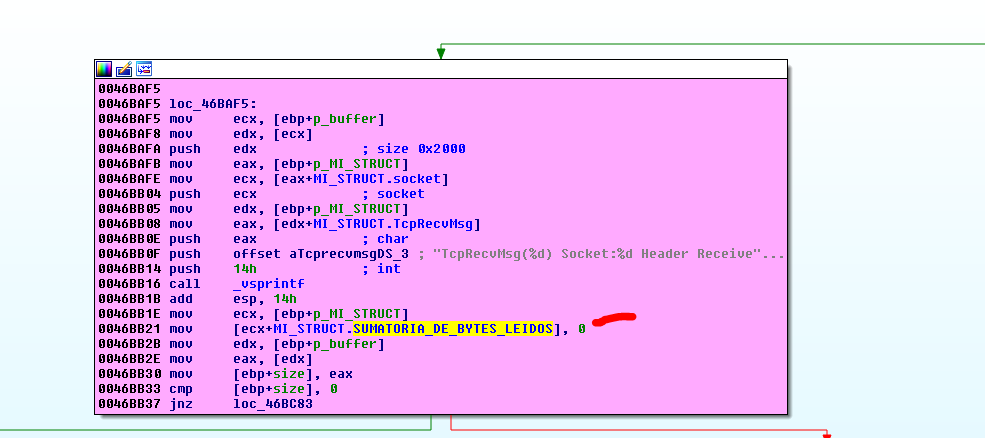
Y allí donde estaba reverseando



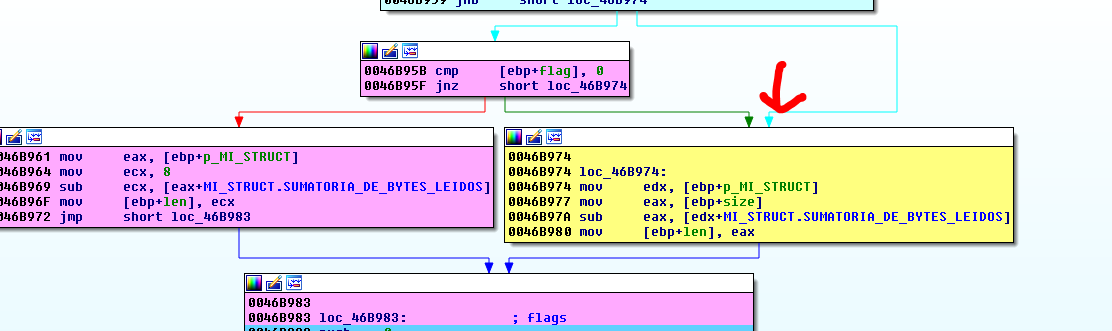
Al otro campo le puse TcpRecvMsg como decía el printf, aun no se bien para que se usa, pero lo identifico.



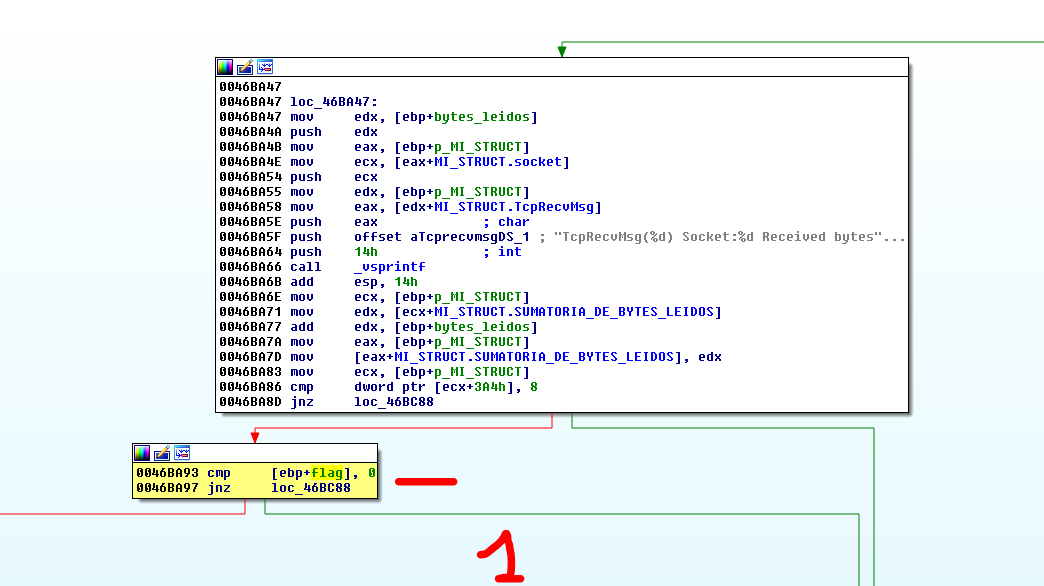
Lo último que hace en ese bloque es leer el size y guardarlo en una variable, así que le pongo size.



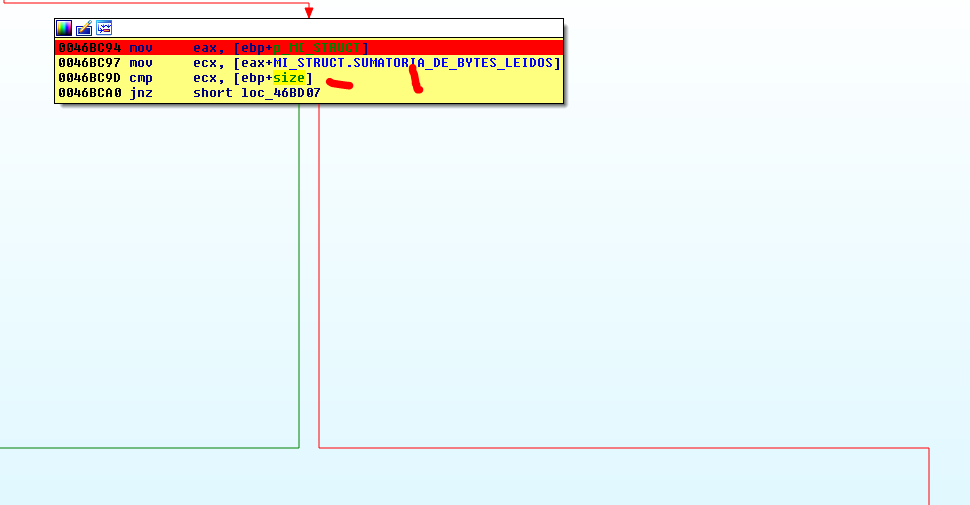
Y pone a cero la sumatoria de bytes leídos y repite el loop volviendo arriba ahora con el size 0x2000.



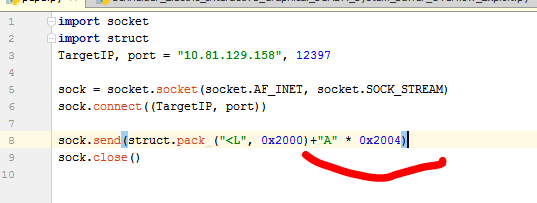
Al final lee el size 0x2000 y no le resta nada pues la sumatoria estaba cero, así que guarda 0x2000 en el len y llama de nuevo a recv.



Llega a la comparación del flag que ahora vale 1.

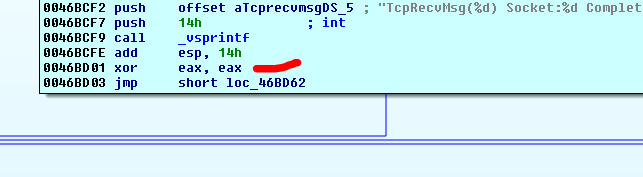


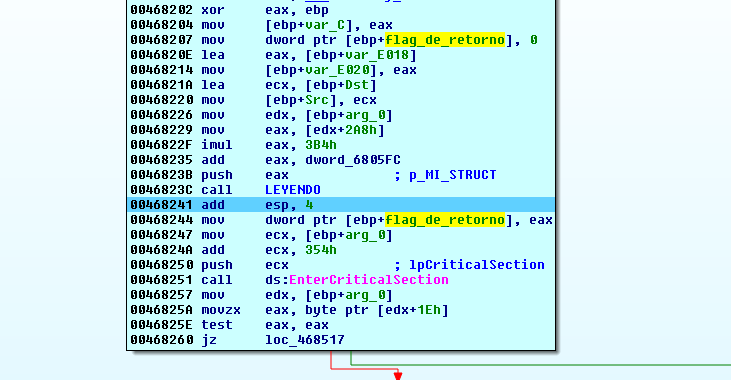
Si pudo leer los 0x2000 bytes que le envie ya que el paquete ahora es

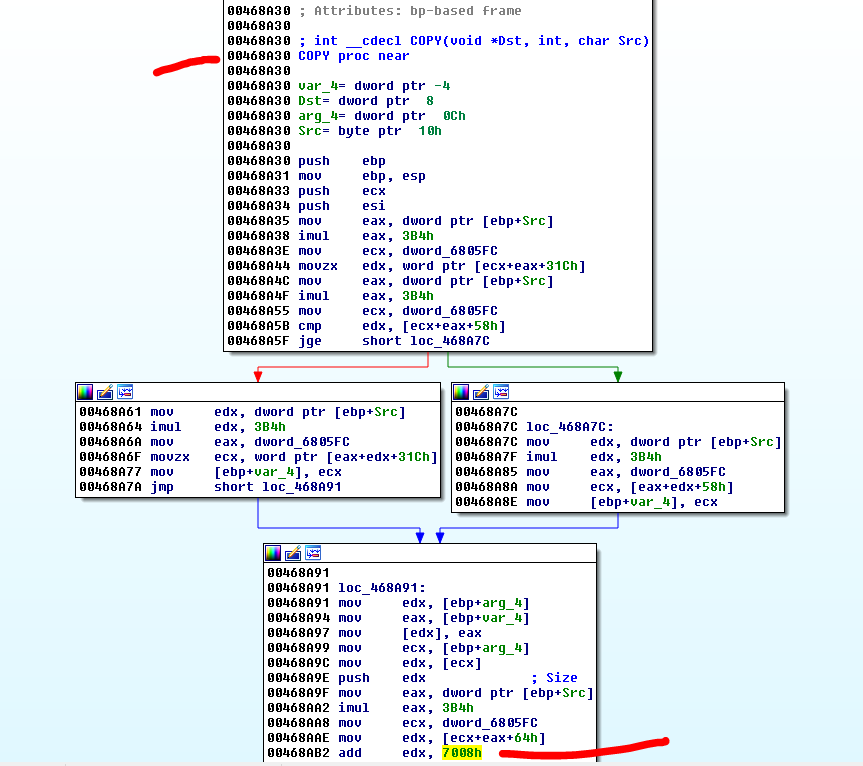


Para que pueda leer 0x2000 tranquilamente en el segundo recv.

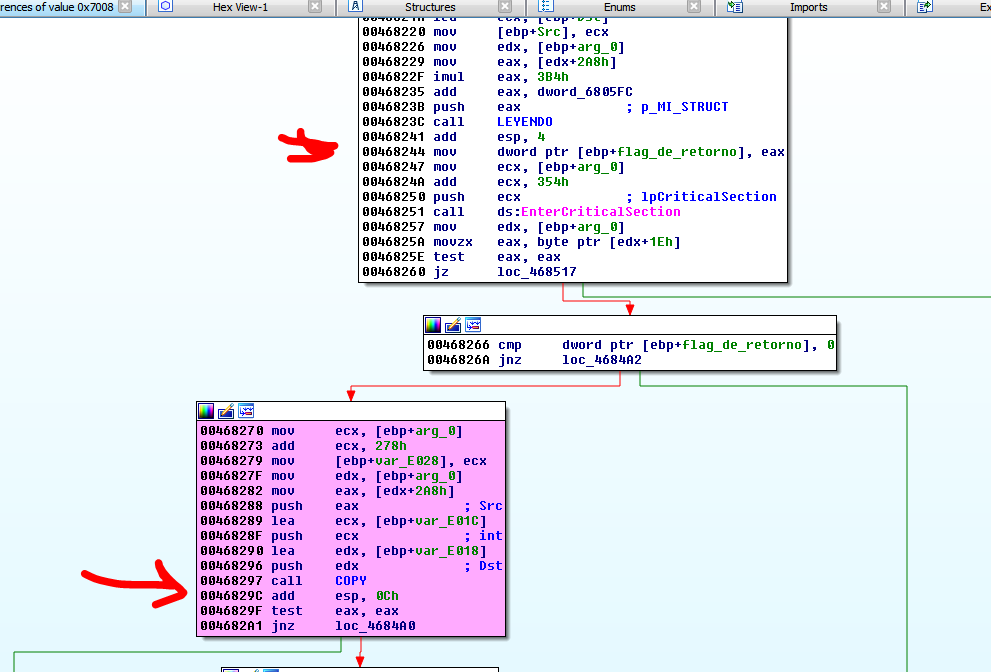
Si puedo leer bien sale de la función LEYENDO poniendo a cero EAX, si todo está bien.

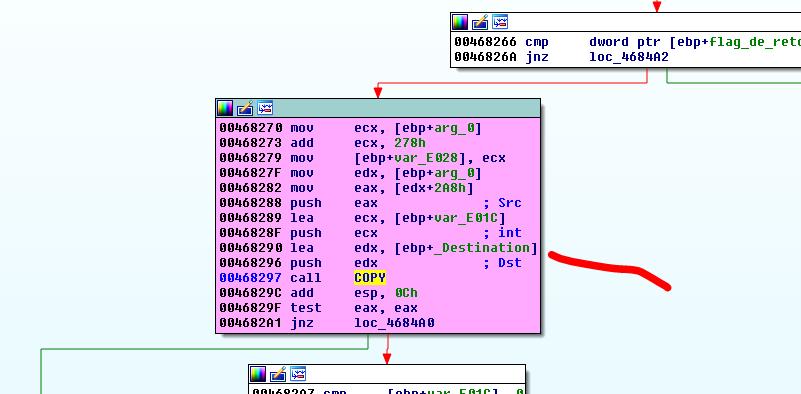




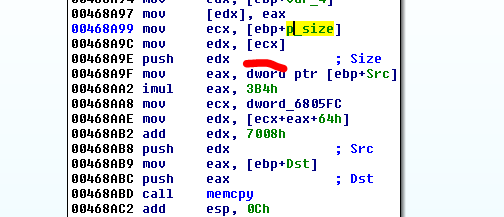


La otra función donde leía el buffer la renombro a COPY veo que la llama justo debajo de donde estoy.

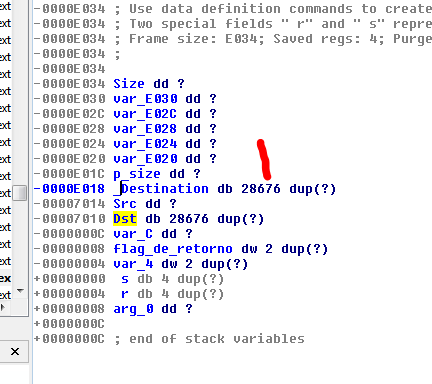




Veo que el destination del memcpy dentro de COPY es ese buffer que renombro como \_Destination ahí se copiara mi data.

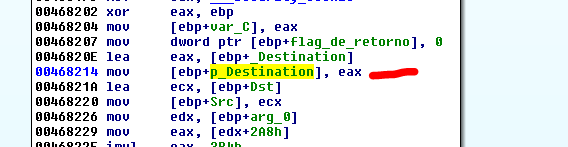


También vemos que allí es un p\_size que pasa al memcpy.



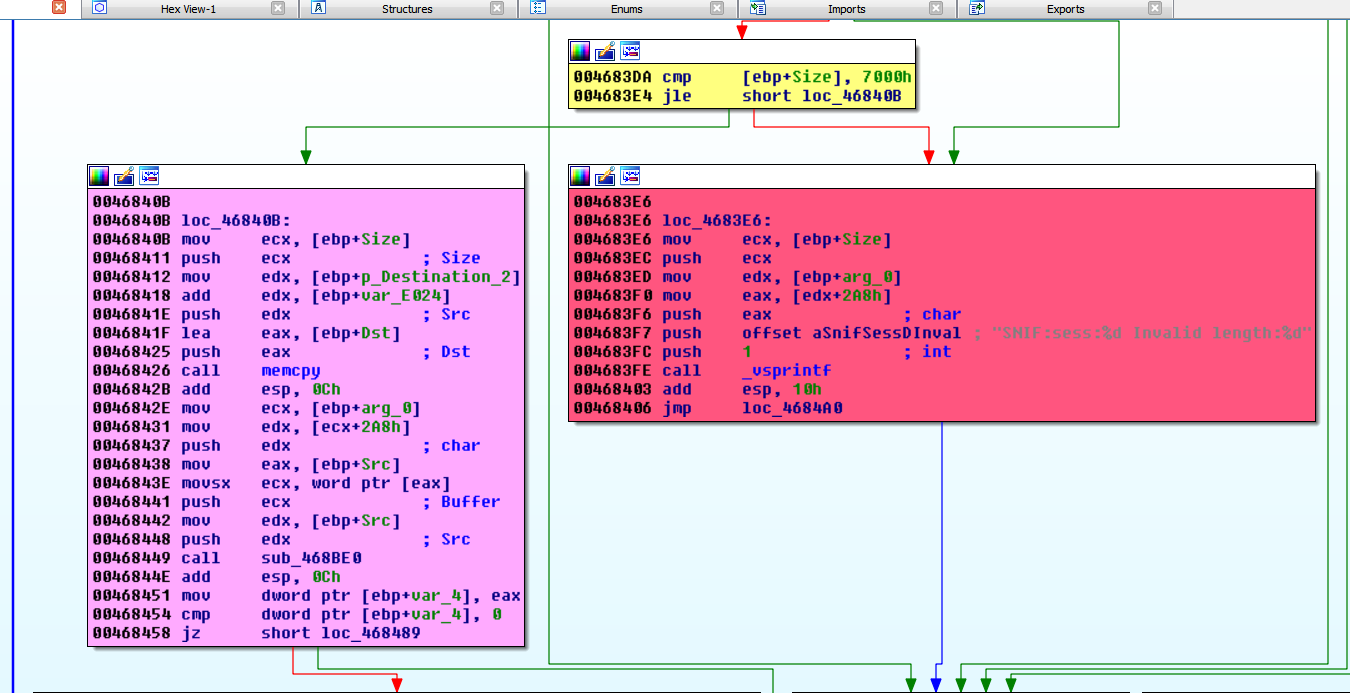
Y que el buffer \_Destination es de 0x7004 bytes de largo, así que allí no va a haber overflow porque comparaba si el size era menor que 0x7000 y si era mayor iba por otro camino donde solo leía 0x7000 como máximo.

También vemos que hay un puntero a \_Destination que lo guarda en otra variable renombro.



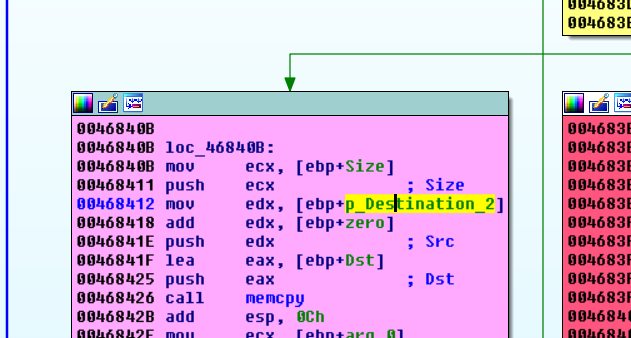


Aquí compara uno de los 0x2000 bytes que le envie con 0x9c pero parece ser el camino de la izquierda algo relacionado con una descompresión lo dice la string, el otro no te tira fuera aparentemente, así que, podría pensar que si el byte es 0x9c, es un flag de comprimido, por ahora evitemos entrar y vayamos por la derecha.

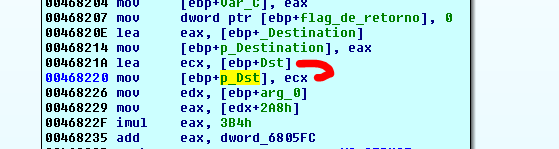


Allí vuelve a comparar con 0x7000 no hay problema con el signo a pesar de ser un JLE porque eso viene de la cantidad de bytes leídos y no puede leer tanto para que se haga negativo, así que por ahí nada.

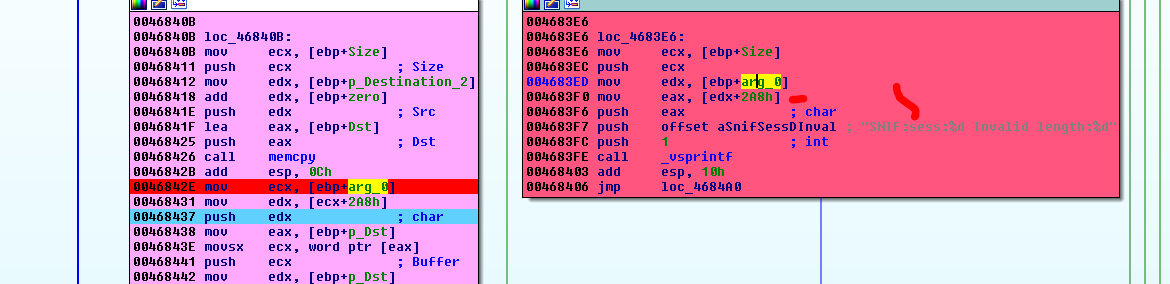
Vemos que copia de mi buffer \_Destination a otro que llama Dst que también es de largo 0x7004



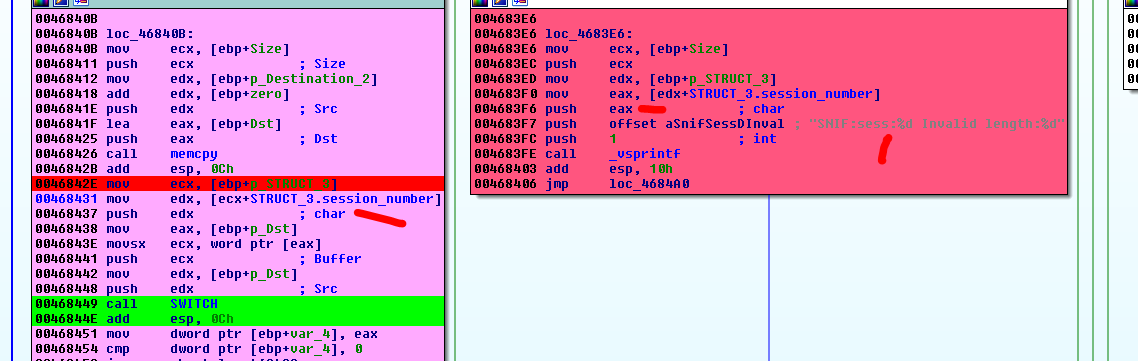
Vemos que lo que llama Src es en realidad un puntero al Dst que es el lugar donde copio mi fruta



También hay una estructura acá



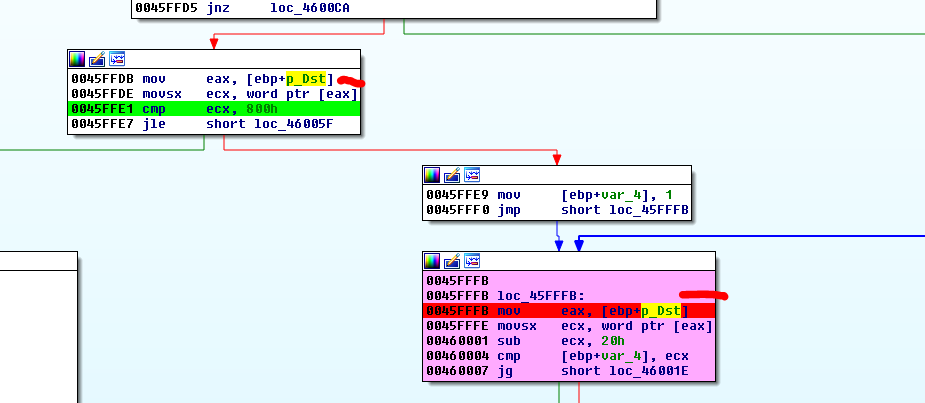
El campo 0x2a8 de la misma parece ser el número de sesión, así que creare una tercera y en ese campo le pondré sesión\_number



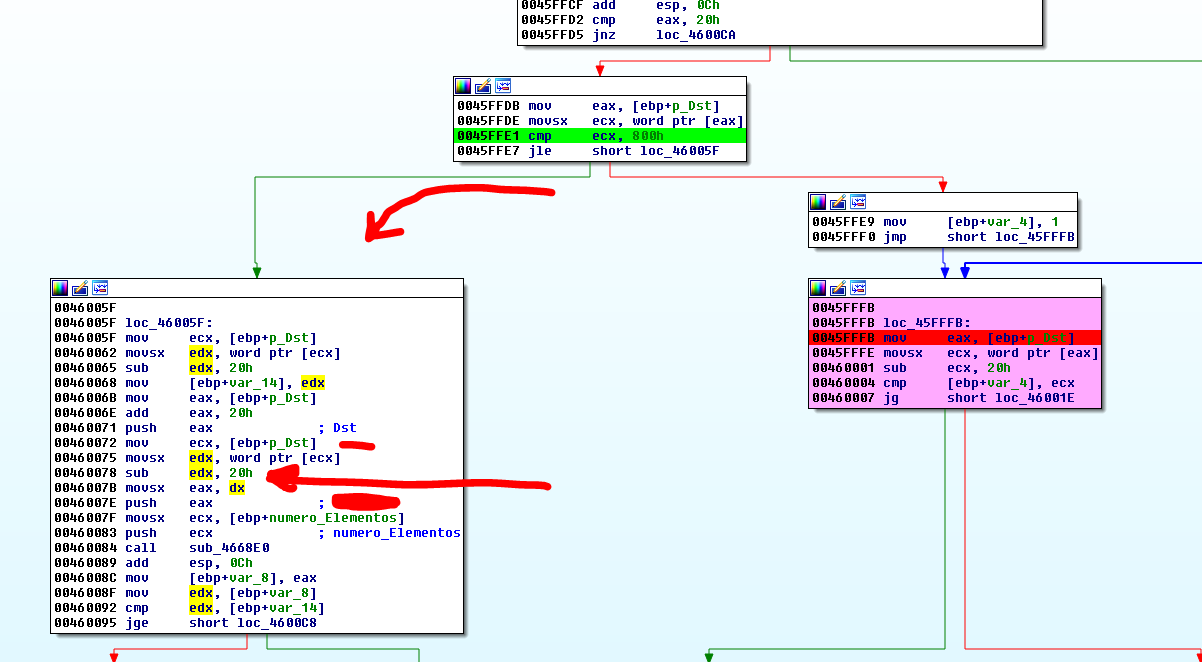
Antes de seguir vamos a aclarar una cosa, este desafío es difícil no solo por no tener símbolos, sino también porque no hay un camino directo entre el punto recv donde ingresa nuestra información y la función vulnerable.

Si pongo un breakpoint en la función vulnerable, veo que constantemente cada pocos segundos para en ella, aparentemente como si fuera un thread diferente al del recv, y necesitamos ver cómo se puede llegar a la misma a pesar de eso, así que sin símbolos y con esa dificultad habrá que exprimir todas las posibilidades de reversing estático y de ultima de dinámico también, para ayudarnos que en la guerra y el reversing todo vale, veremos cómo hacer.

Así que nosotros llegamos hasta el SWITCH, luego allí dentro hay varios caminos, para ver cual nos llevara a la función vulnerable analicemos un poco la misma, para tratar de ver de dónde sale ese valor y si podemos afectarlo.

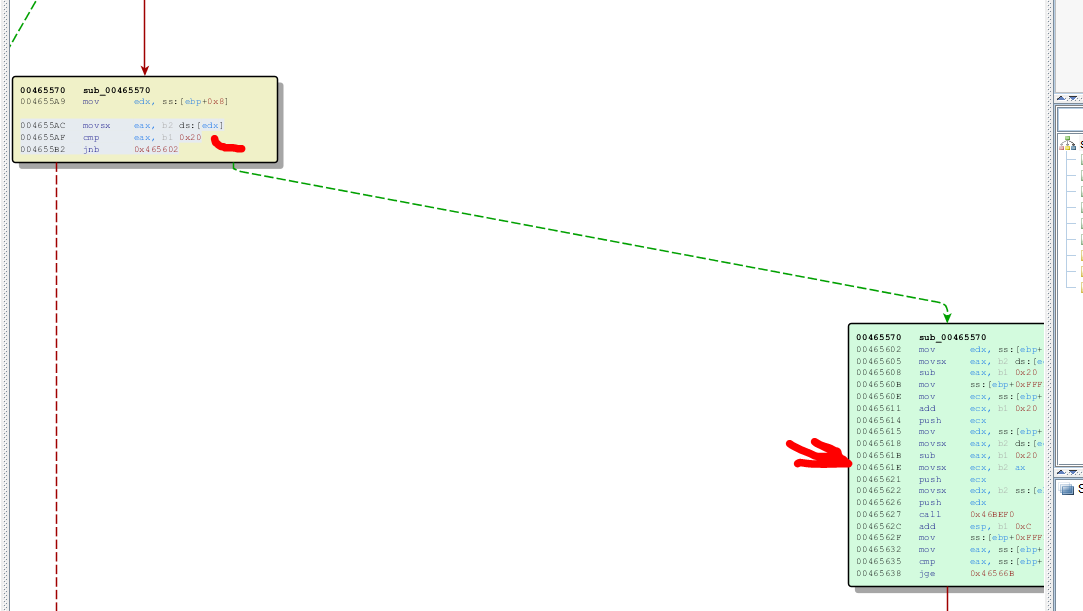


Vemos que hay un buffer yo le puse p\_Dst, su primer word es el valor que filtra, habíamos visto que en el parche si ese valor es menor que 20 te tira fuera.

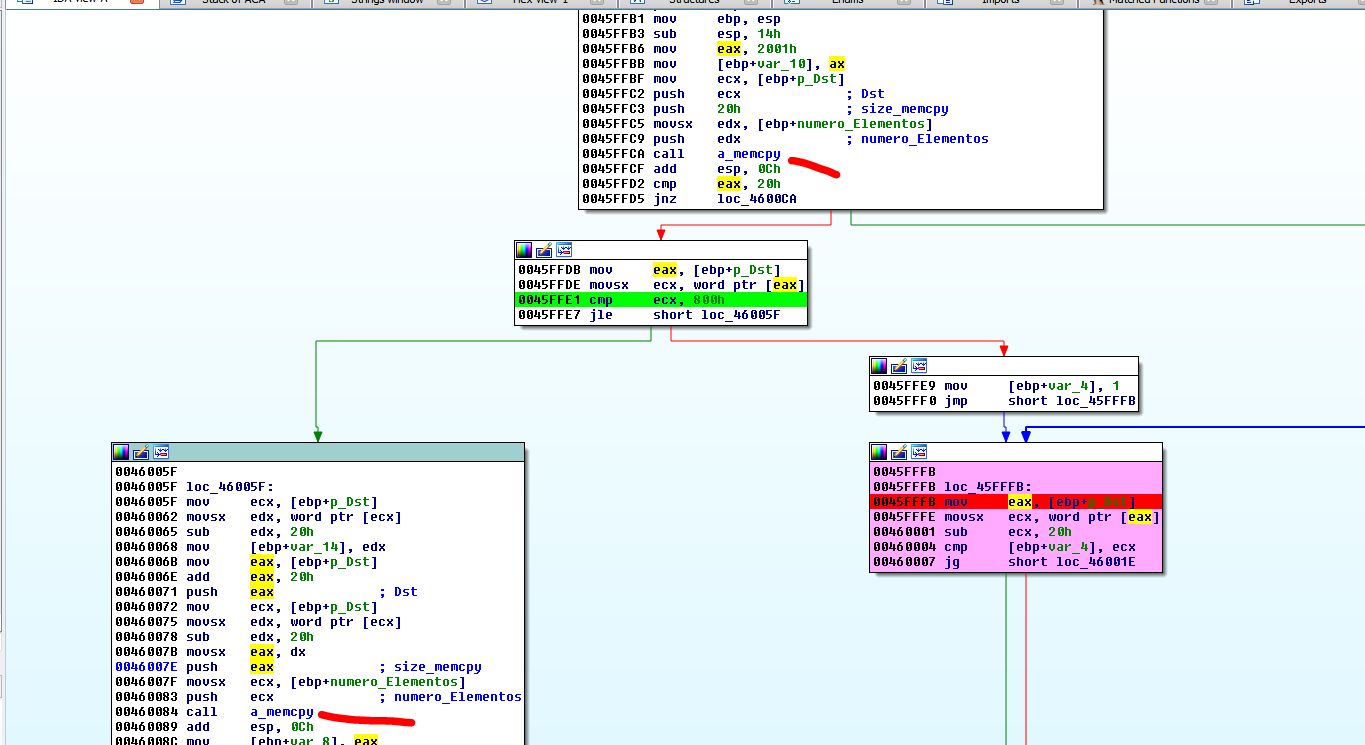


Así que se valor si es menor que 0x800 por ejemplo 0x10, va por la derecha y luego lo vuelve a leer y le resta 0x20 guardando el resultado y usándolo ese size negativo como size de un memcpy al stack, lo cual lo desbordara.

Para evitar eso, en el parche chequean antes de la resta, que el valor ese sea mayor que 0x20 antes de ir al sub.

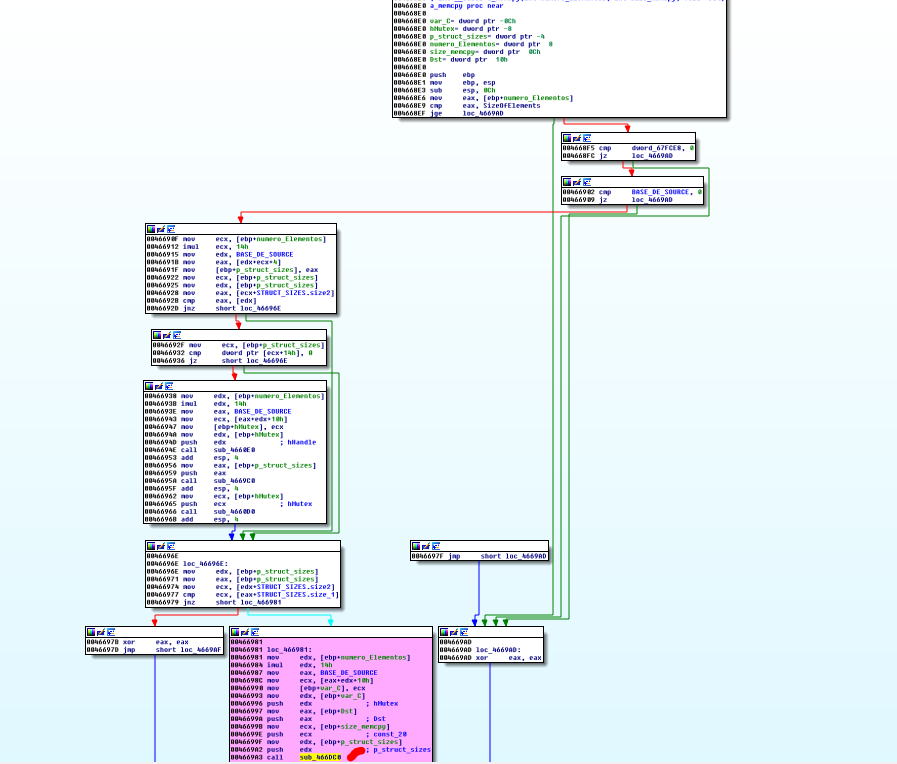


Ahí en la función parcheada, si no es más bajo JNB va a la zona de la resta con 0x20 para que no de negativo.

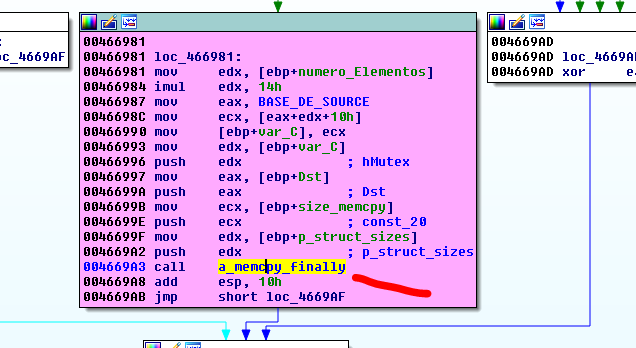


El problema aquí, es que en el primer memcpy la función Destination es el p\_Dest de dónde saca el valor posiblemente nuestro, con el size 0x20, pero no está claro aquí cual es el source del memcpy, parece que lo calcula dentro de la misma función, así que va a haber que analizarla un poco para ver de dónde sale el src y por supuesto nuestro valor que estará en el primer word de ese buffer, tanto del src como del destino ya que copia uno al otro los primeros 0x20 bytes.

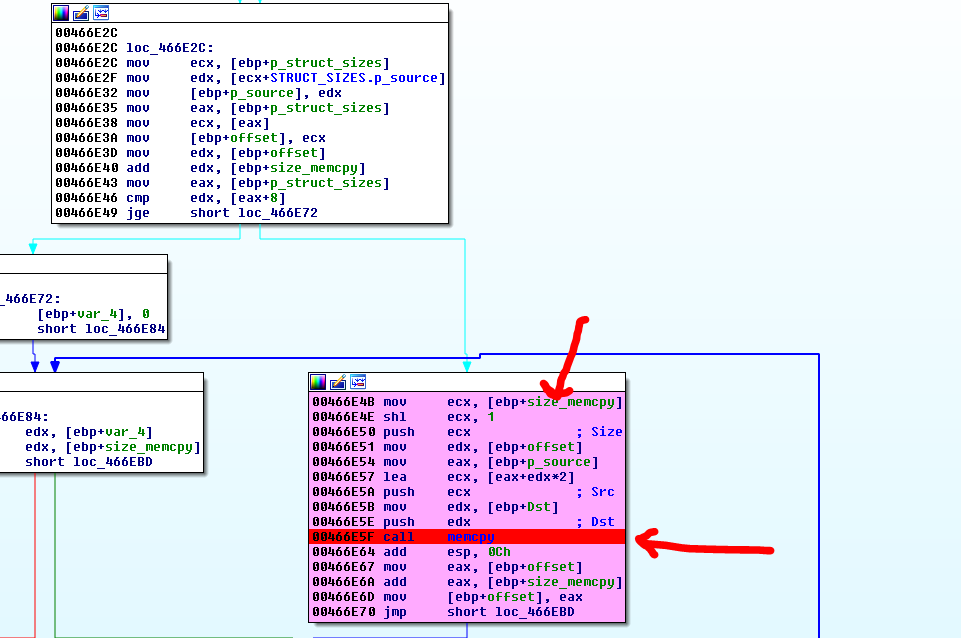
Yo ya la reversee, por eso verán unos nombres que ya están puestos, pero iré explicando un poco, dentro de la función a\_memcpy hay otra que va finalmente al memcpy.



Allí ese call es el que copia.



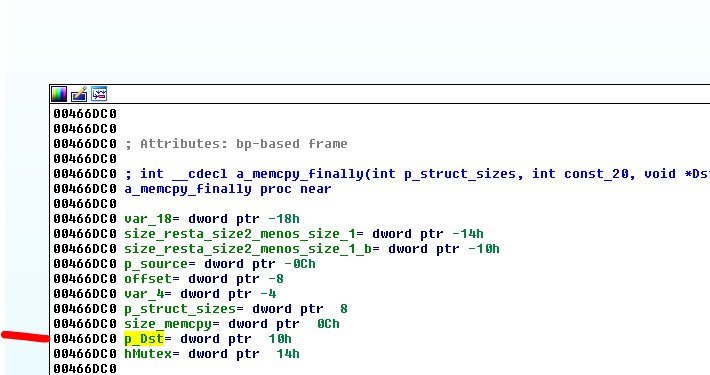
Dentro de la misma

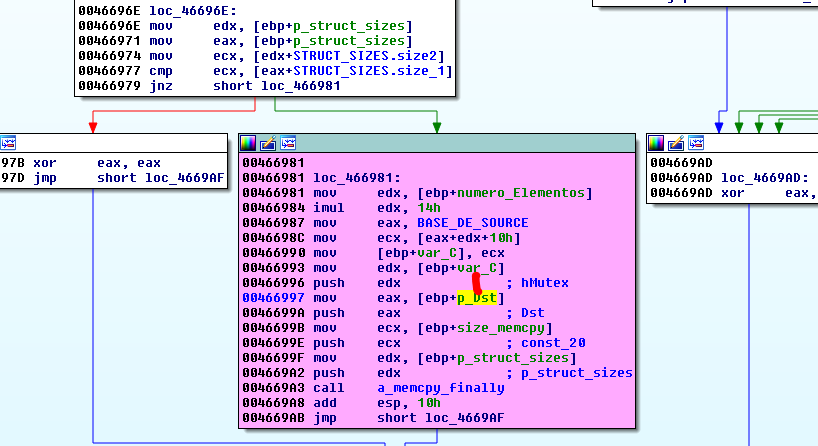


Estaría el supuesto lugar donde crasheara, si llegamos a pasar un valor menor de 0x20 que se reste contra 0x20 y de un size negativo.

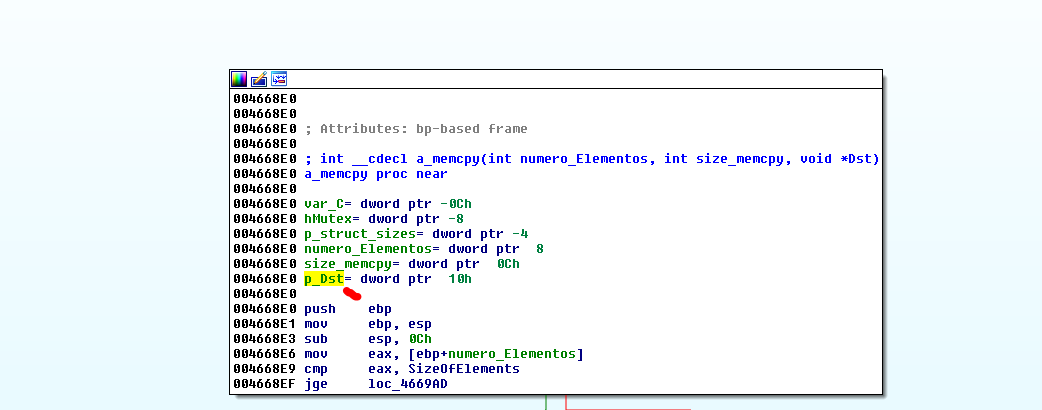
Renombre Dst como p\_Dst ya que es un puntero que apunta a un buffer del stack, eso no se ve aquí, pero si uno sigue hacia atrás las referencias para ver de dónde sale.

Vemos que es un argumento de esta función, si miro en la referencia.

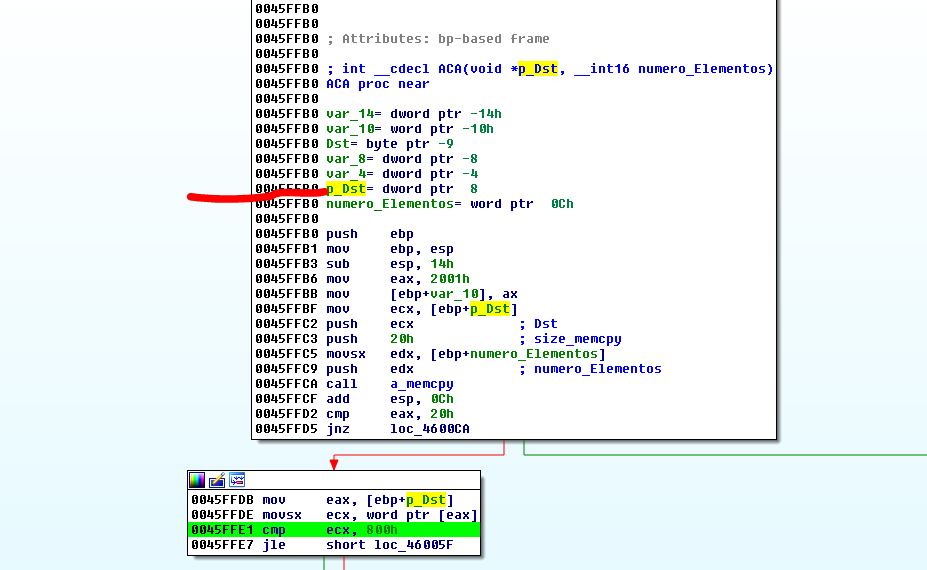




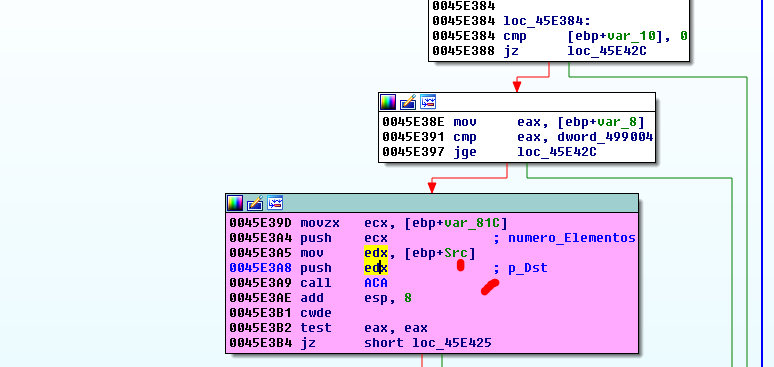
También es un argumento del tipo puntero



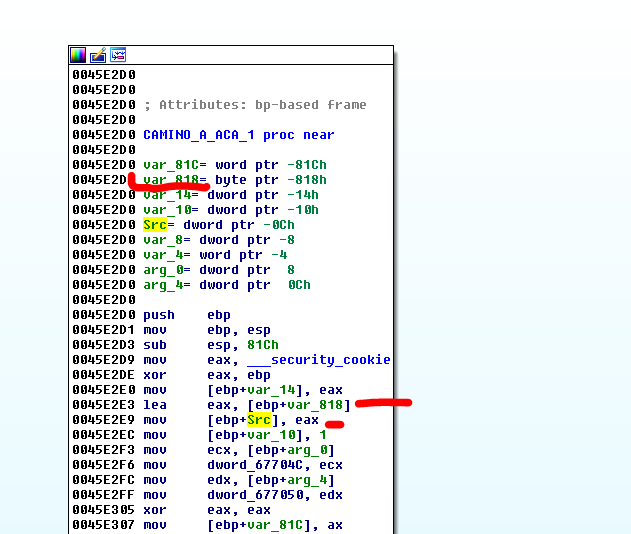
Y en la referencia que es la función vulnerable también es un argumento del tipo puntero



Y en la referencia de esta, es una variable del tipo puntero también que se llama Src, veamos de donde sale.



Vemos que es un buffer en el stack del cual saca la dirección y la guarda en la variable puntero Src.



Veamos el largo del buffer.



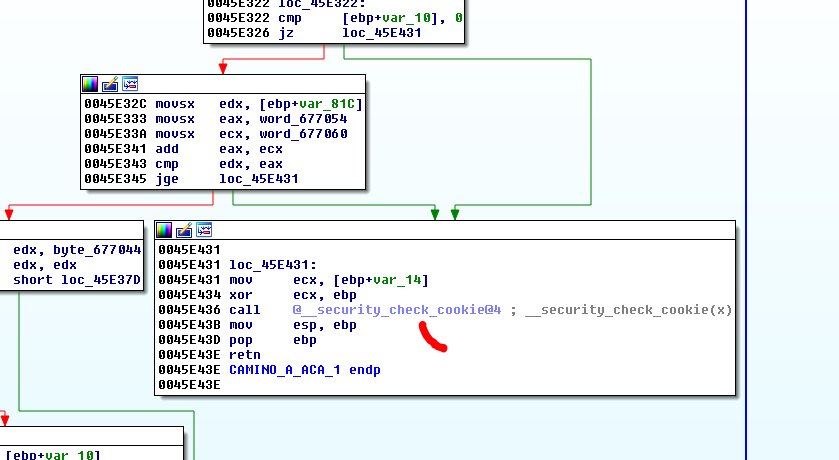
Es 2052 o sea 0x804

Python>hex(2052)

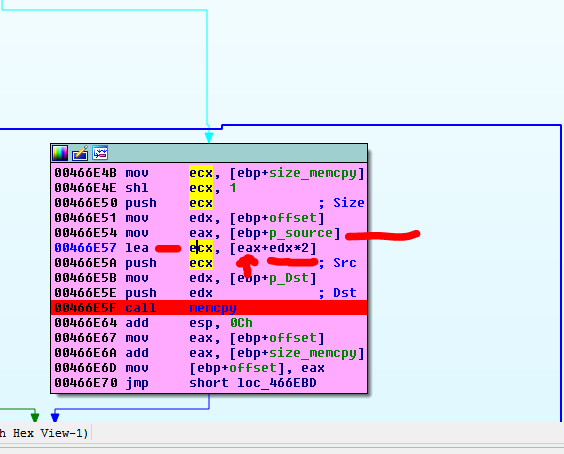
0x804

Así que si podemos copiar más que 0x804 habrá overflow, ya vimos que es posible crear mediante una resta un size negativo allí, así que podemos pisar todo el stack inclusive el seh si tiene cookie.

Vemos que antes del return de esa función que tiene el buffer hay un chequeo de cookie

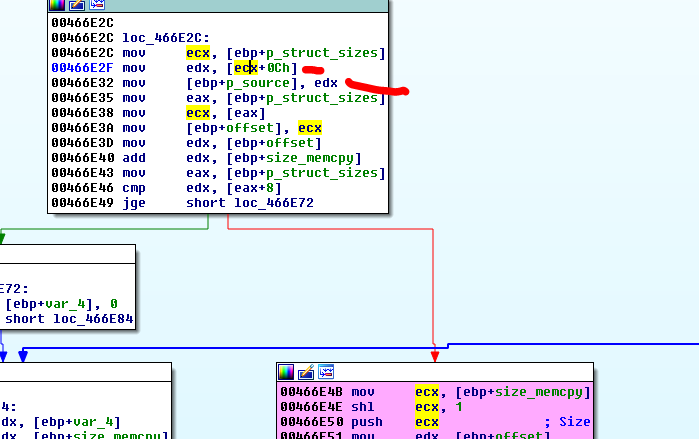


Igualmente podemos evitarla pisando el SEH, así que sigamos.



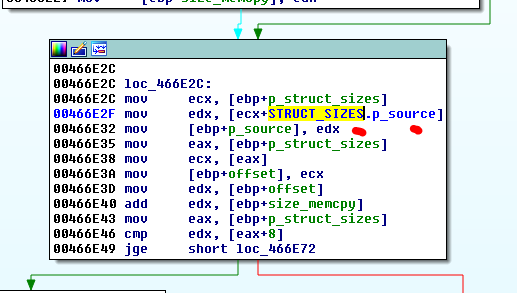
En el memcpy final para ver desde donde sale la data que va a copiar acá, vemos que hay un LEA cuyo resultado es el puntero al Src, hay un EDX\*2 dentro del lea por lo cual puse que EDX será un OFFSET ya que un puntero no se multiplica y si un offset, y EAX sería el puntero base del buffer src, por eso le puse p\_source, posiblemente EDX valga 0 al inicio y termine copiando desde el inicio del buffer apuntado por p\_source.

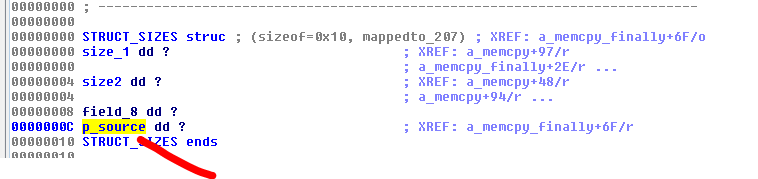
Bueno p\_source es una variable del tipo puntero sigamos hacia atrás a ver de dónde sale.



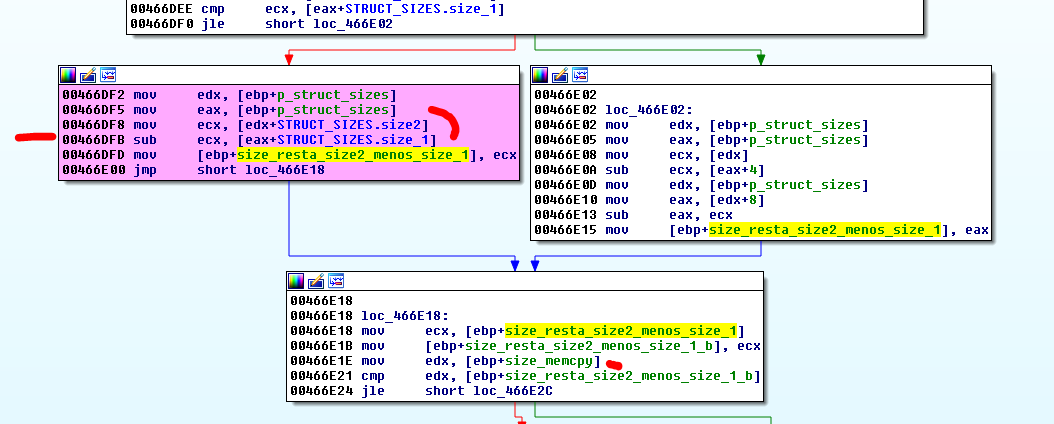
Vemos que allí lo guarda y que sale del campo 0xc de una estructura que como no tengo idea, le puse STRUCT\_SIZES ya verán porque más adelante, igual cualquier nombre funcionara.

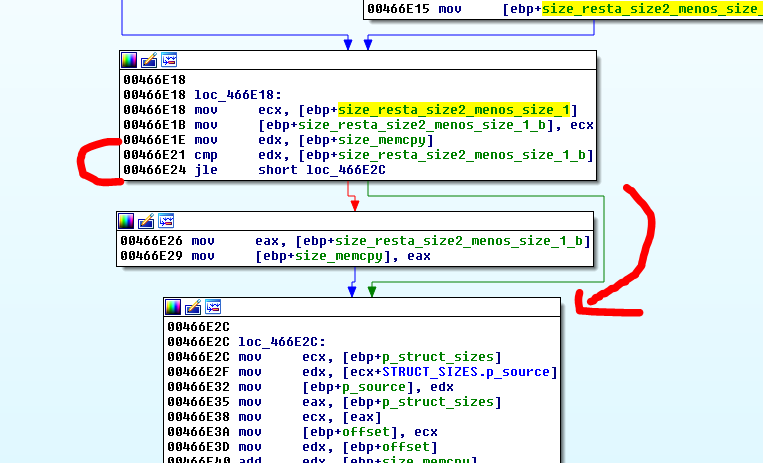
Así que cree una estructura chica de 4 dwords y la llame



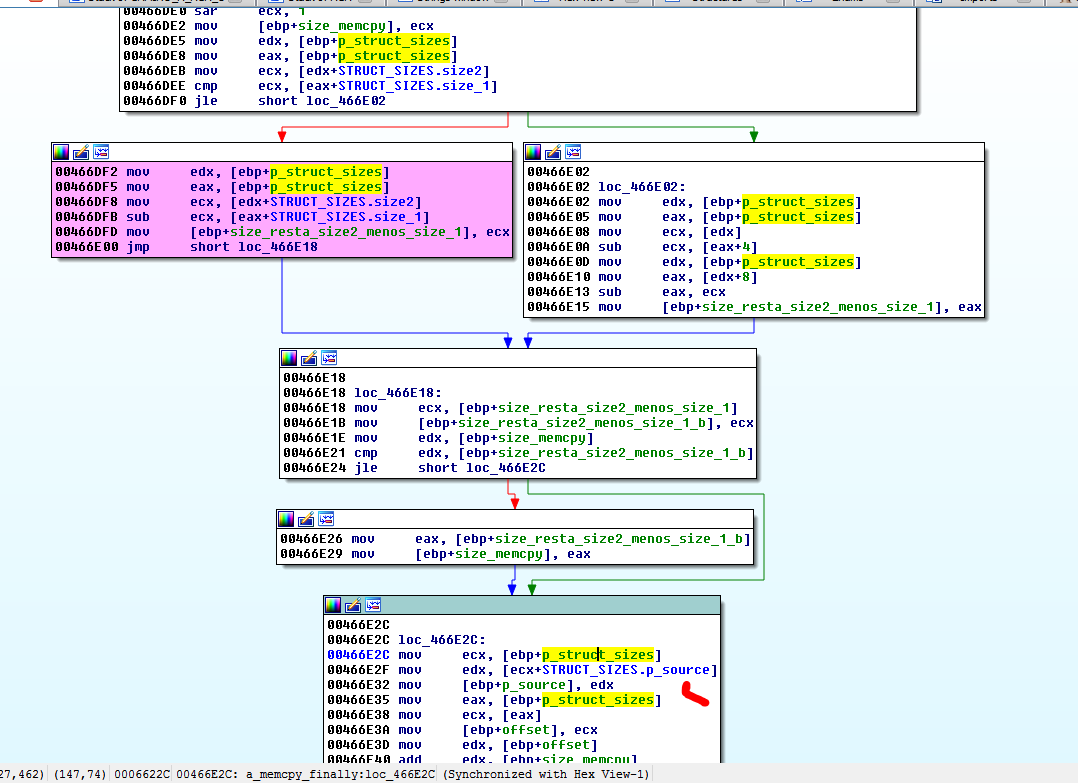


Vemos por encima porque no tiene mucha importancia por ahora que el size\_memcpy es comparado contra la resta de los dos primeros DWORDS de la estructura que yo llame size1 y size2, y si size\_memcpy es menor que la resta de ambos usara este sino el otro.



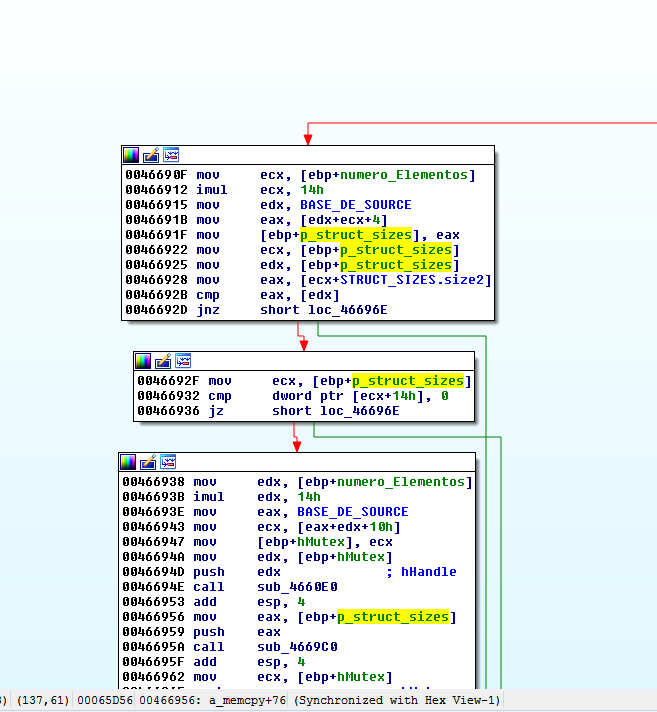


Vemos también que el p\_source de la estructura STRUCT\_SIZES viene de antes y que no se guarda en esta función, ya que mirando todos los lugares que usa p\_struct\_sizes no hay ninguno que guarde en ese campo un puntero.



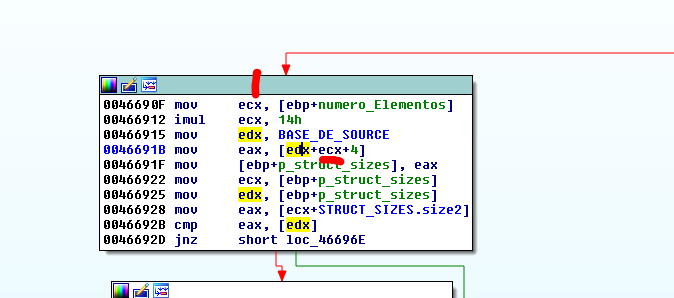
Solo lee allí, veamos en la referencia.

Vemos que el puntero a la estructura se genera y se guarda allí, así que la creación del puntero de p\_source debe ser posterior a esto.



Analicemos un poco la forma en que calcula aquí el puntero a la estructura p\_struct\_sizes.

Vemos que hay una variable global que yo le puse como nombre BASE\_DE\_SOURCE cuyo contenido es un puntero, eso es la base de todo.



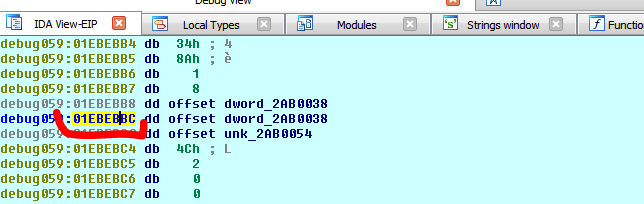
ECX yo le puse Número de elementos, es posiblemente un offset y luego le suma 4 o sea que desde una variable global fija, yo puedo hallando su contenido, sumándole 4 obtener el puntero a la estructura la cual en el cuarto dword tendrá el puntero al source.

Es importante ver que el src proviene de una variable global porque como la información se pasa entre threads, es muy posible que use dicha variable para guardar en el thread del recv y luego levantarla en este otro.

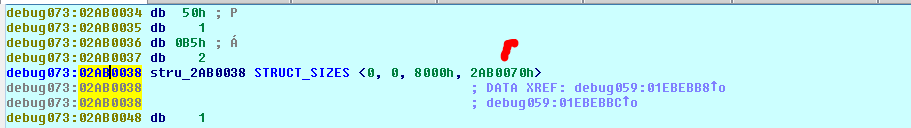
Si arrancamos el debugger veamos que en la variable global BASE\_DE\_SOURCE hay un puntero, en mi caso 0x1ebebbb8



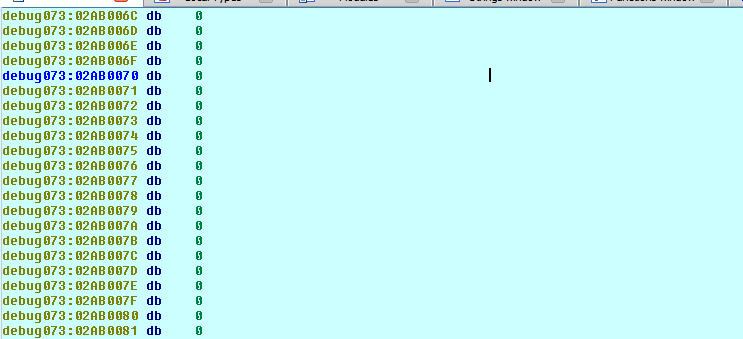
Como recordamos que a ese valor le suma 4 y halla el contenido que es el puntero a p\_struct\_sizes.

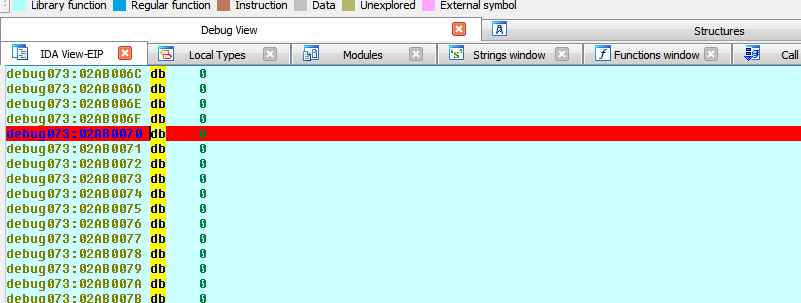


Al contenido de BASE\_DE\_SOURCE le sumo 4 y su contenido es el p\_struct\_sizes en mi caso 0x2ab0038 si voy allí y apreto la T

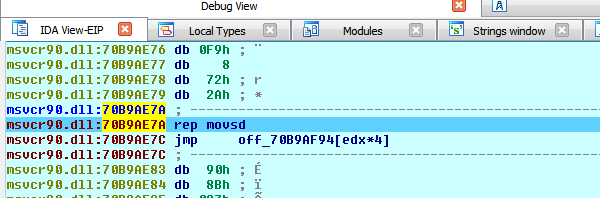


El 4 campo (el 0xc) es el puntero a source, lo más fácil seria ir allí que esta vacío y ponerle un BREAKPOINT MEMORY a ver si copia allí mi fruta y donde lo hace.

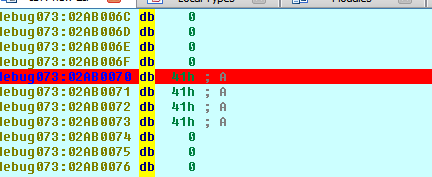




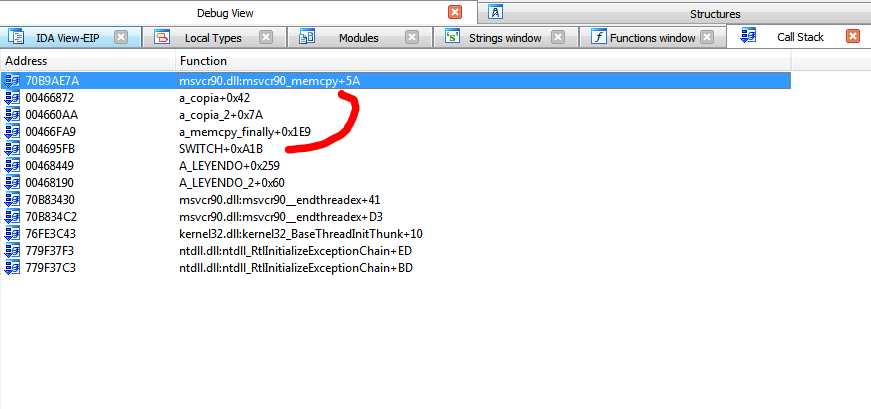
Vemos que paro allí



Y vemos que está copiando mi fruta



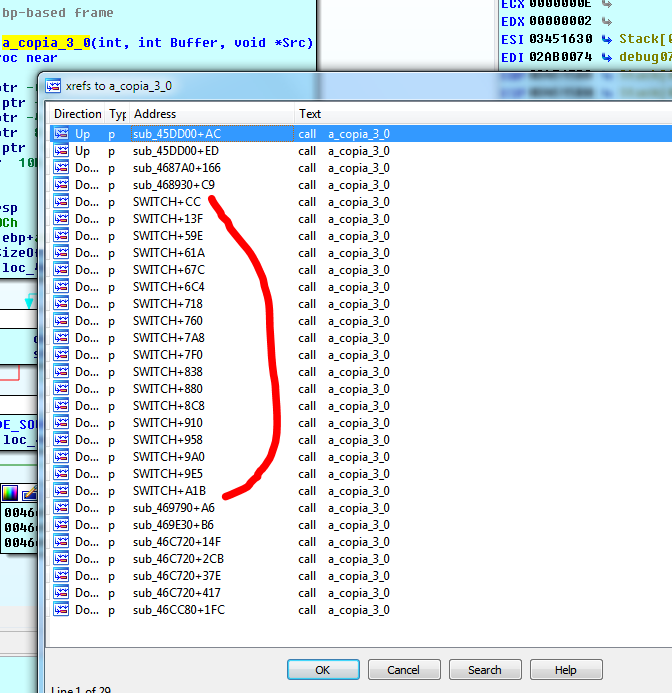
Si miramos el call stack



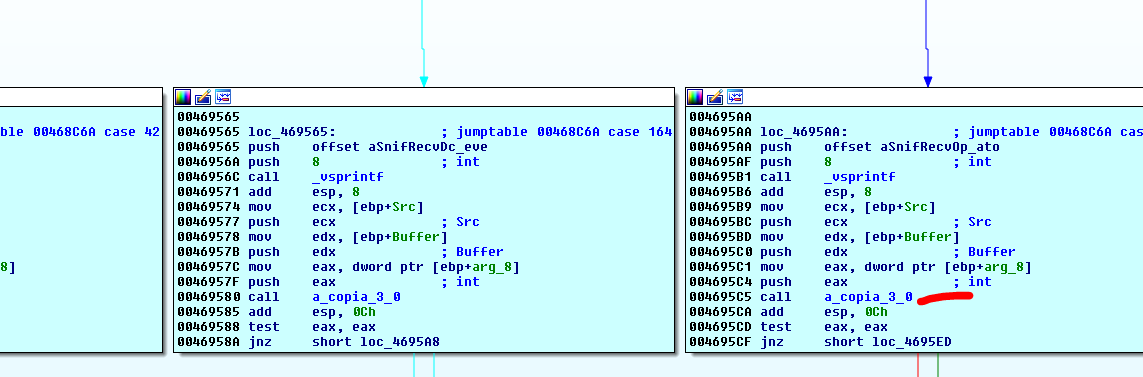
Vemos que hay un camino desde leyendo que pasa por switch y que escribe allí y que podemos seguir hasta donde copia.

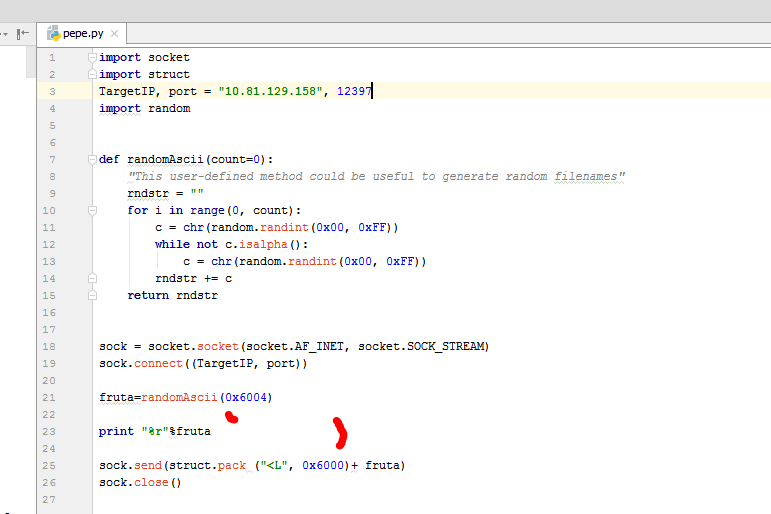
Vemos que la segunda vez que para copia el famoso 0x3e que muchos encontraron en el loop en vez del valor nuestro, así que ya sabemos adónde hay que llegar y un posible camino, ahora que ya tenemos claro eso debemos seguir mirando el switch.

Igual si miramos un poco las formas de llegar a ese copy desde el SWITCH.

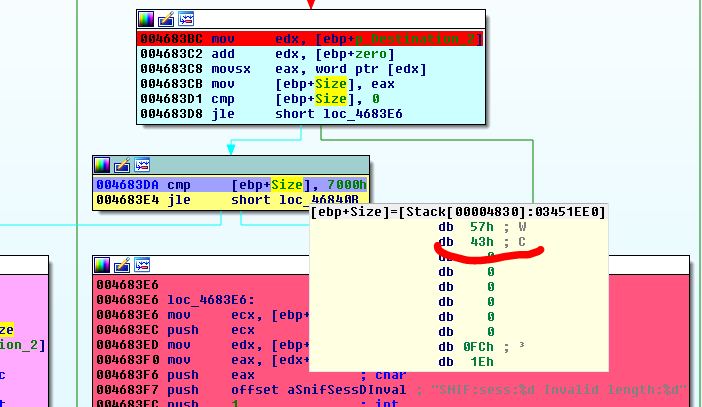


Vemos que hay un montón de formas de llegar y que nosotros estamos llegando por un lugar incorrecto ya que hay muchos opcodes validos que llegan al copy y nosotros estamos tirando cualquier cosa, sin fijarnos de que sea un opcodes valido.



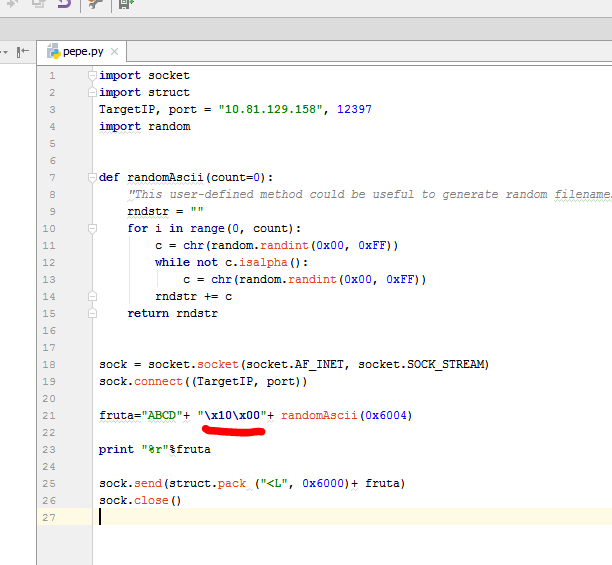


Creare una string random y la imprimiré para ver qué posición llega al OPCODE para poder arreglarlo.

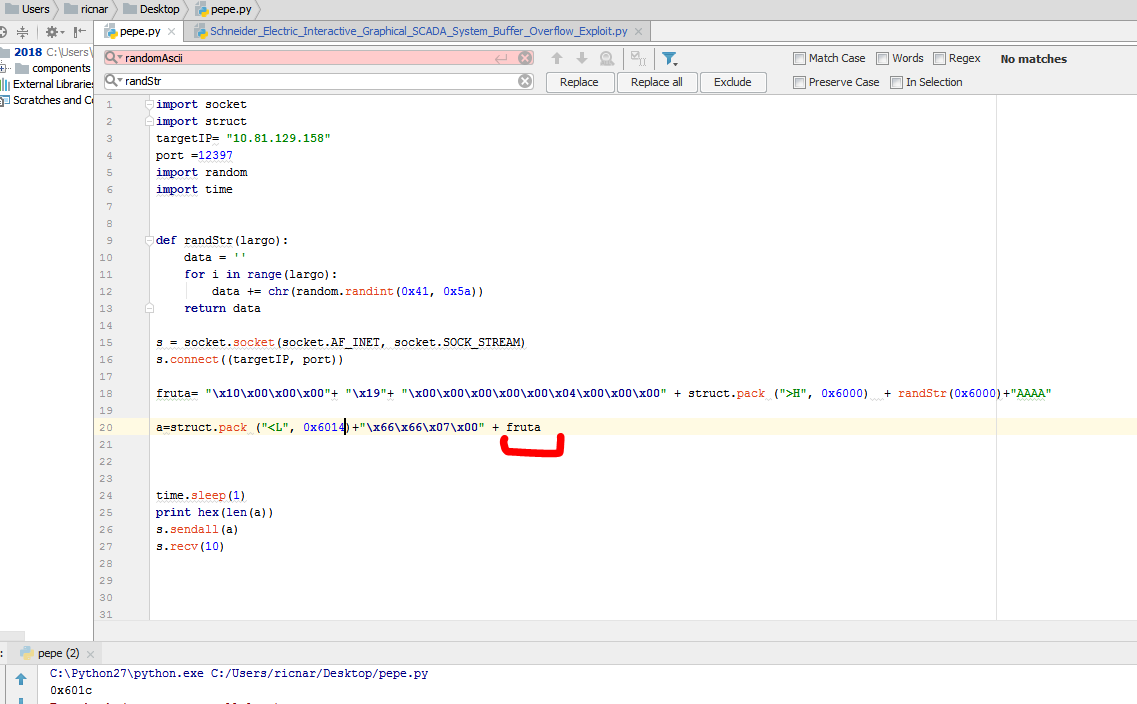




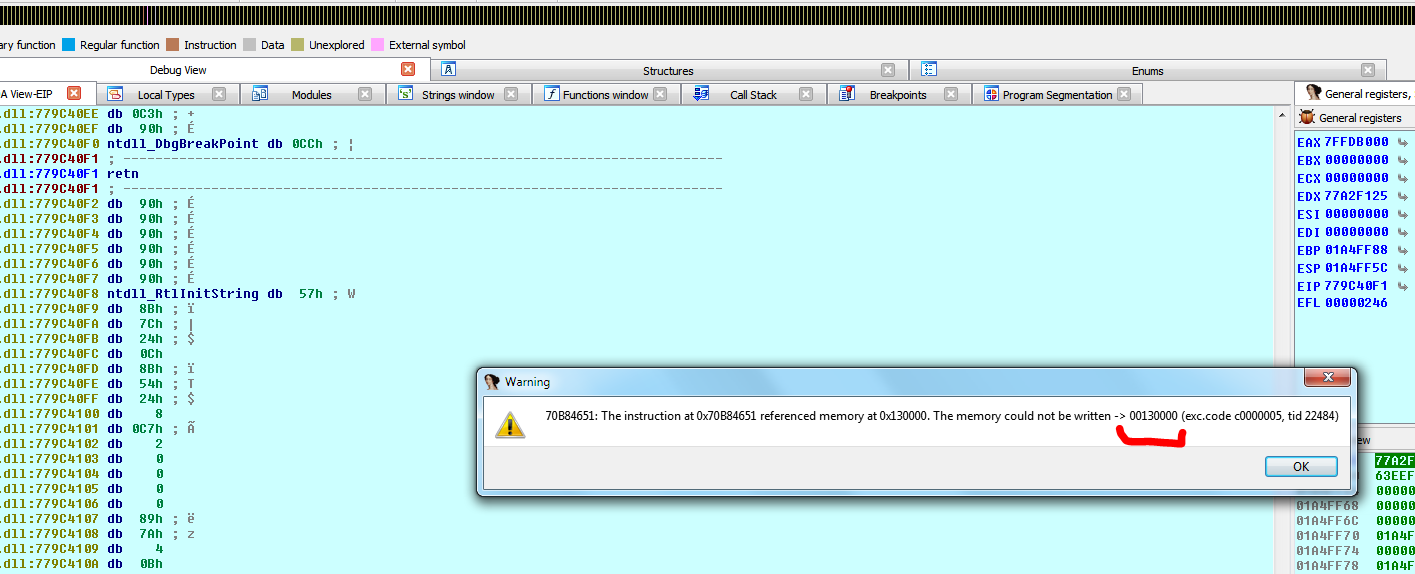
Vamos arreglando el paquete a enviar.



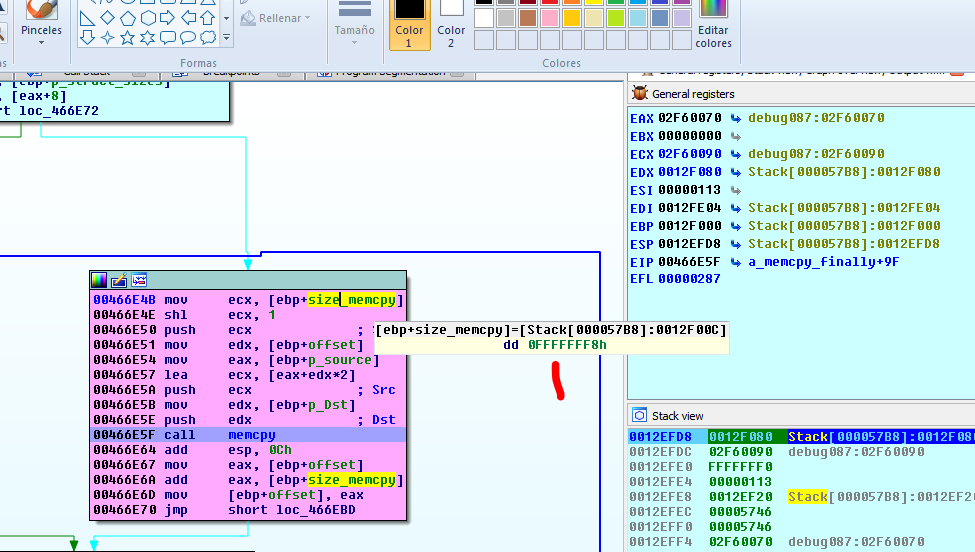
Le pongo 0x10 al size ese a ver si luego poniéndole un opcodes correcto lo hago llegar a la función vulnerable.



Veo que mandándole el 0x19 siempre y cuando haga un recv al final después del send y respete el largo del paquete 0x6014 que son los primeros dos bytes del size me crashea en el memcpy sin problemas.



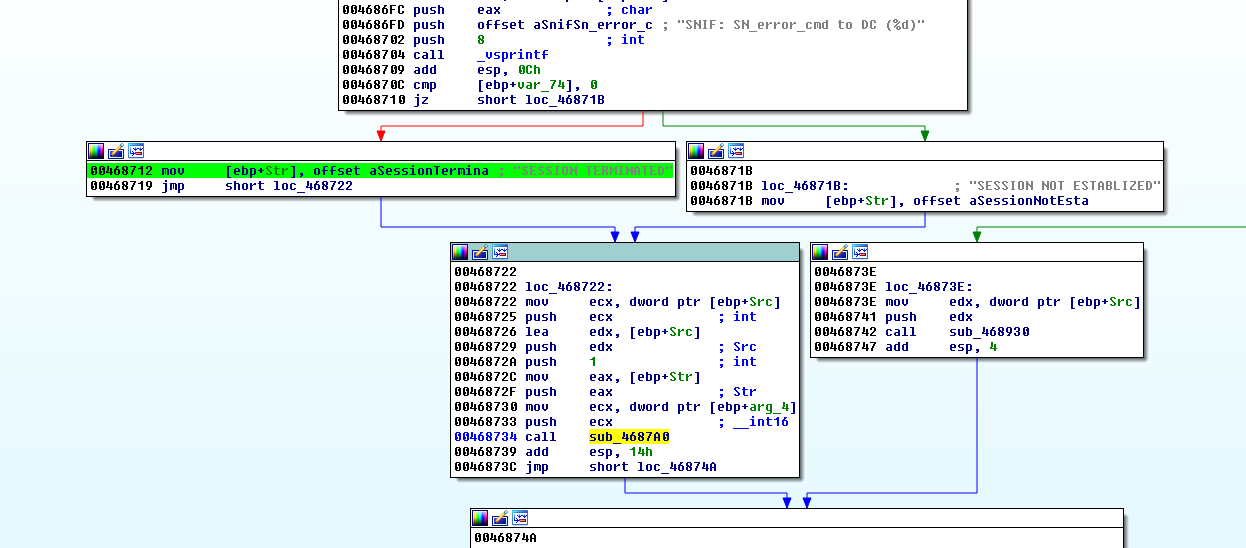
Ahí crashea copiando todo el stack



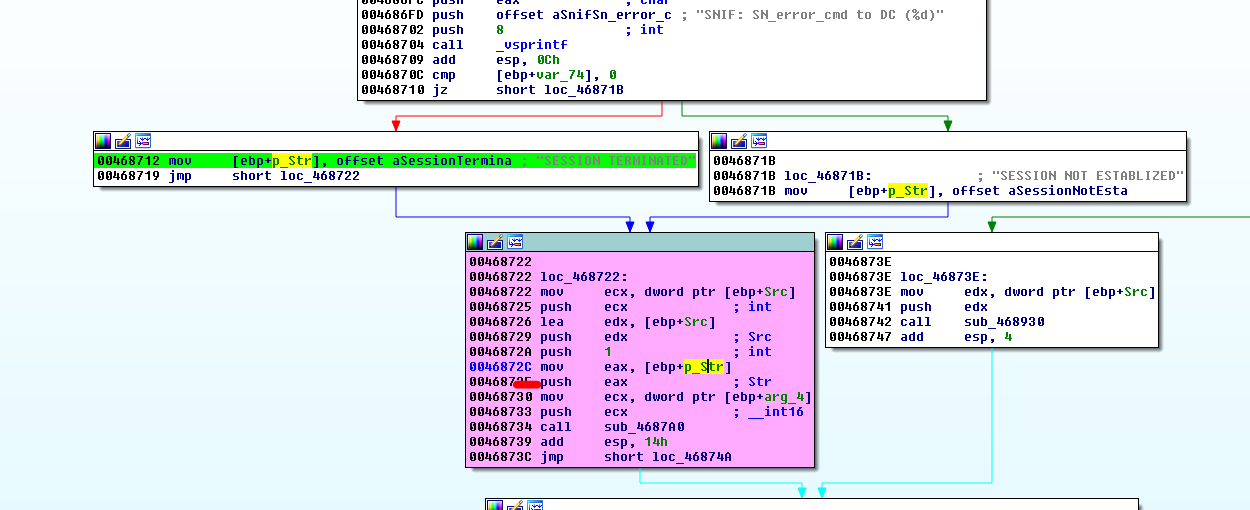
Si uno no pone el recv final el programa copia un 0x3e en la el buffer, o sea 0x3e es el largo de ese paquete que arma cuando la conexión termina si vemos ese paquete tiene dentro de los 0x3e de largo la string.

"SESSION TERMINATED"

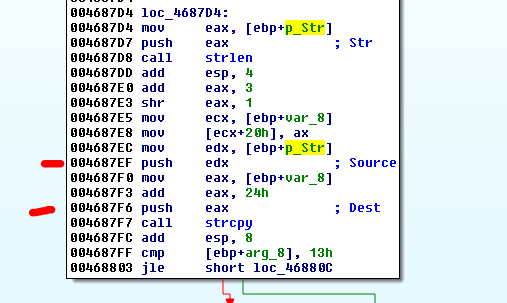
Si la buscamos es aquí



El puntero a esa string se pasa como segundo argumento

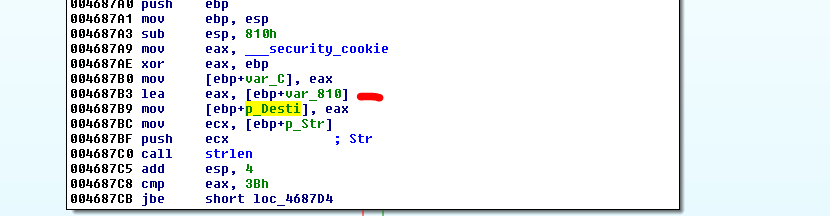


Vemos que luego va a un strcpy donde copia la string en la posición 0x24 de un buffer cuyo puntero está en var\_8

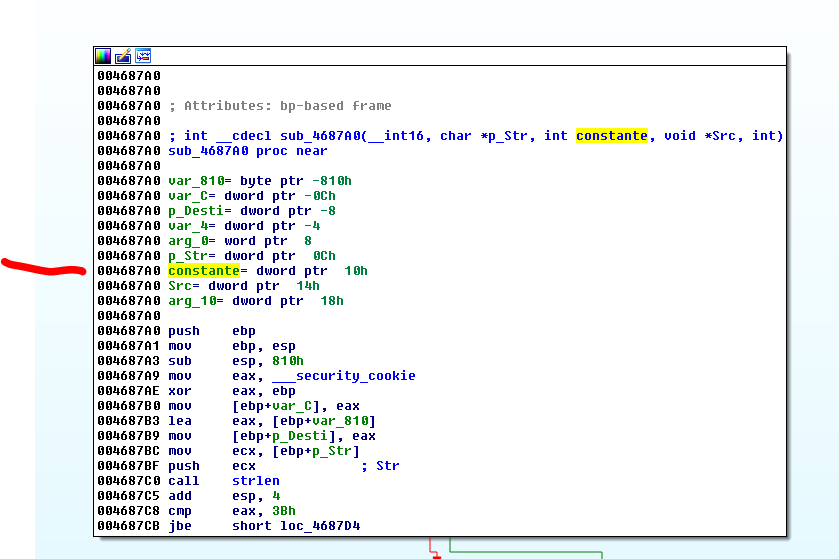


Si a var\_8 le ponemos p\_Desti veamos de donde sale.

Vemos que es un buffer en el stack cuya dirección la guarda en la variable puntero p\_Desti.



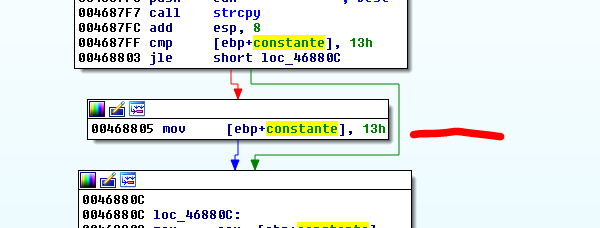
Y al inicio del mismo buffer debe guardar el 0x3e, el mismo sale de acá. Hay un argumento que le puse de nombre constante



En la referencia vemos que es un 1.

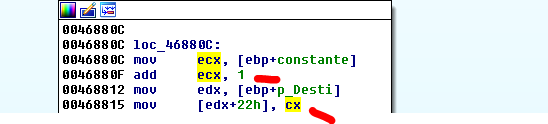


Vemos que la compara con 0x13

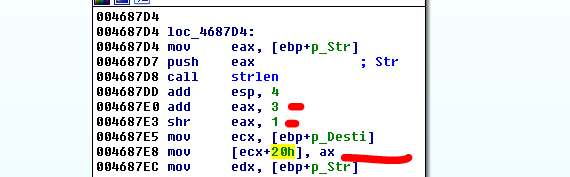


Si es menor o igual, todo bien y es 1 así que nada sigue.

Vemos que le suma 1 y guarda el 0x2 en EDX+0x22



También hay otra constante que sale del largo de la string SESSION TERMINATED



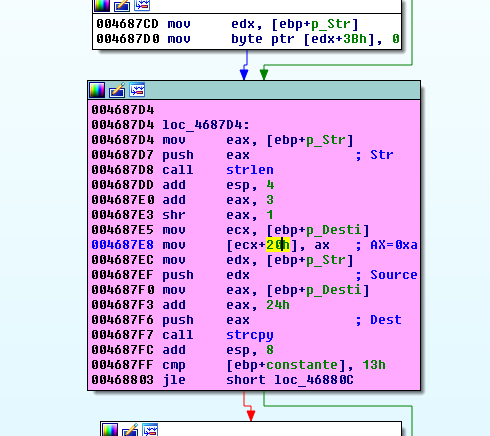
A ese largo (0x12) le suma 3 y le hace SHR, 1 que era

shr eax, 1  ;Signed division by 2

O sea

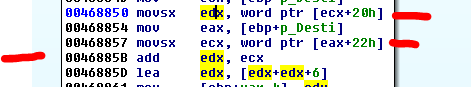


Que guarda 0xa

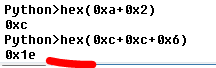


Entonces en la posición 0x22 hay un 2 y en la posición 0x20 hay un 0xa.

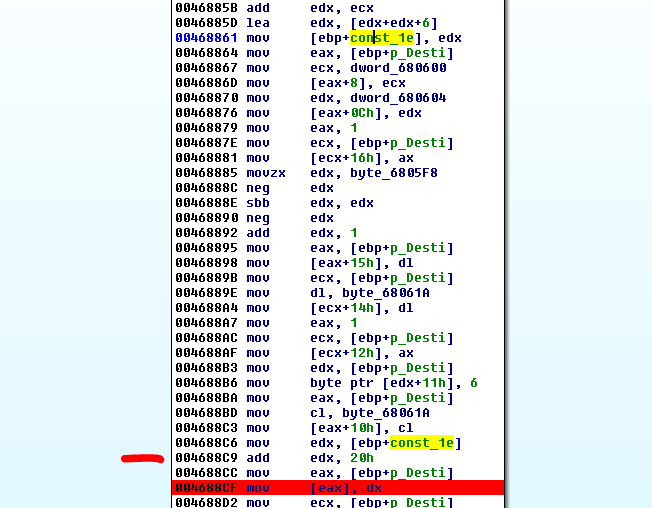
Suma los dos numeros aqui



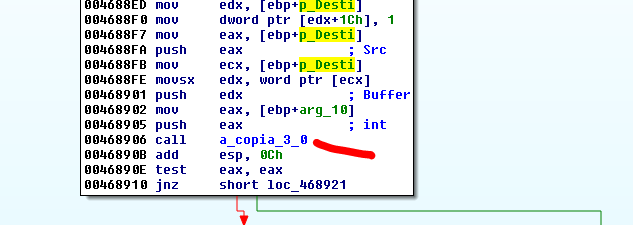
Y luego el resultado lo suma a si mismo y le suma 6 seria.



Luego



Le suma 0x20 convirtiéndolo en el 0x3e que nos llega al finalizar la sesión y que lo guarda en el primer lugar de ese buffer que luego copia.



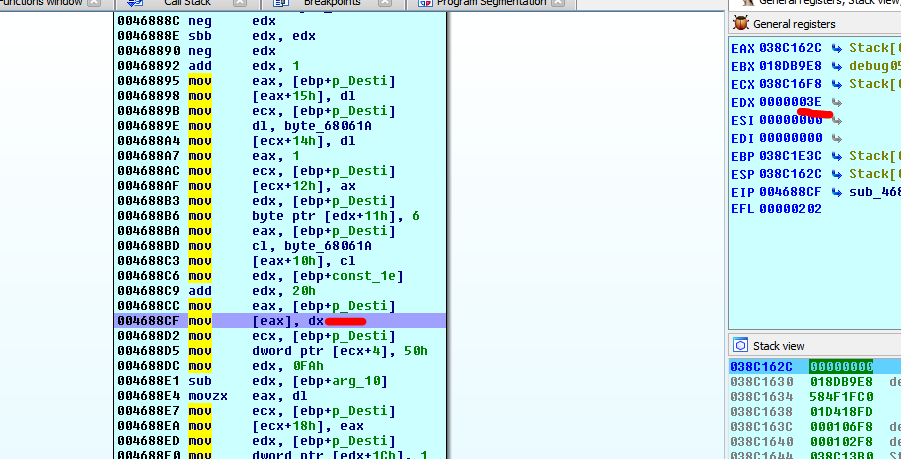
Allí adentro sabemos que copiaba en nuestro buffer del que luego sale el valor que pasa por la función vulnerable.

Ahora porque si no ponemos el recv, pasa por aquí y arruina todo?, se podría hacer una comparación con y sin el recv, al quitarlo pasa por aquí, si lo tiene no pasa, y como la string dice session terminated, podemos suponer que el recv mantiene la sesión abierta y mientras crashea, el que tenga ganas puede probarlo si es así, usando wireshark mirando si la conexión muere va aquí a copiar el 0x3e y si sigue viva crashea.

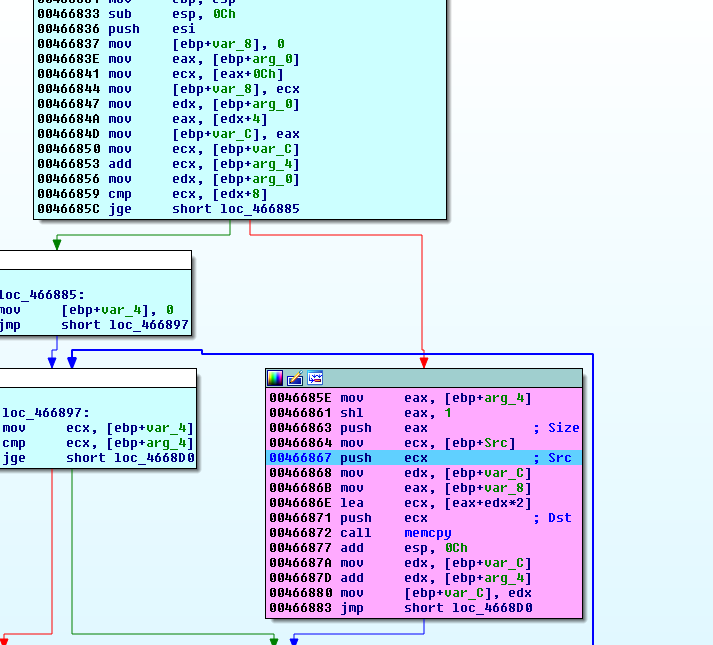
Para que vean que no miento



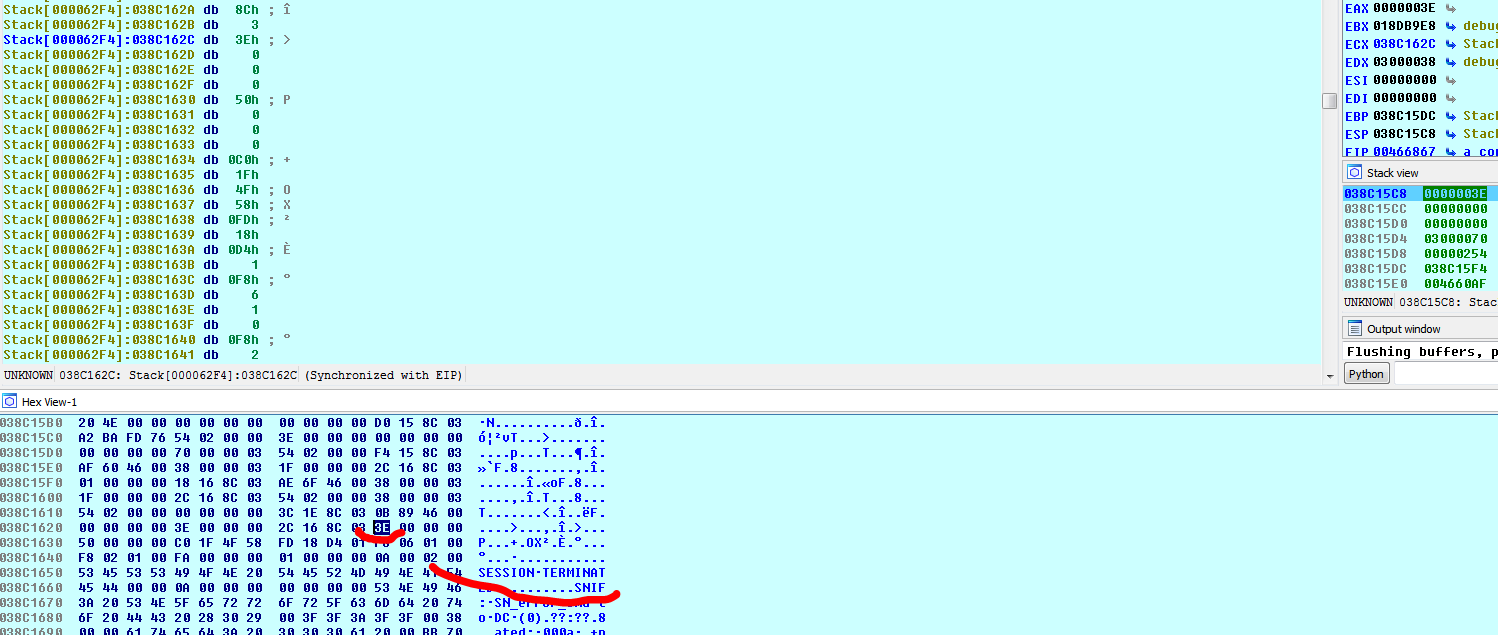
Si le quito el recv y lo tiro



Vemos que ahí guarda el 0x3e.

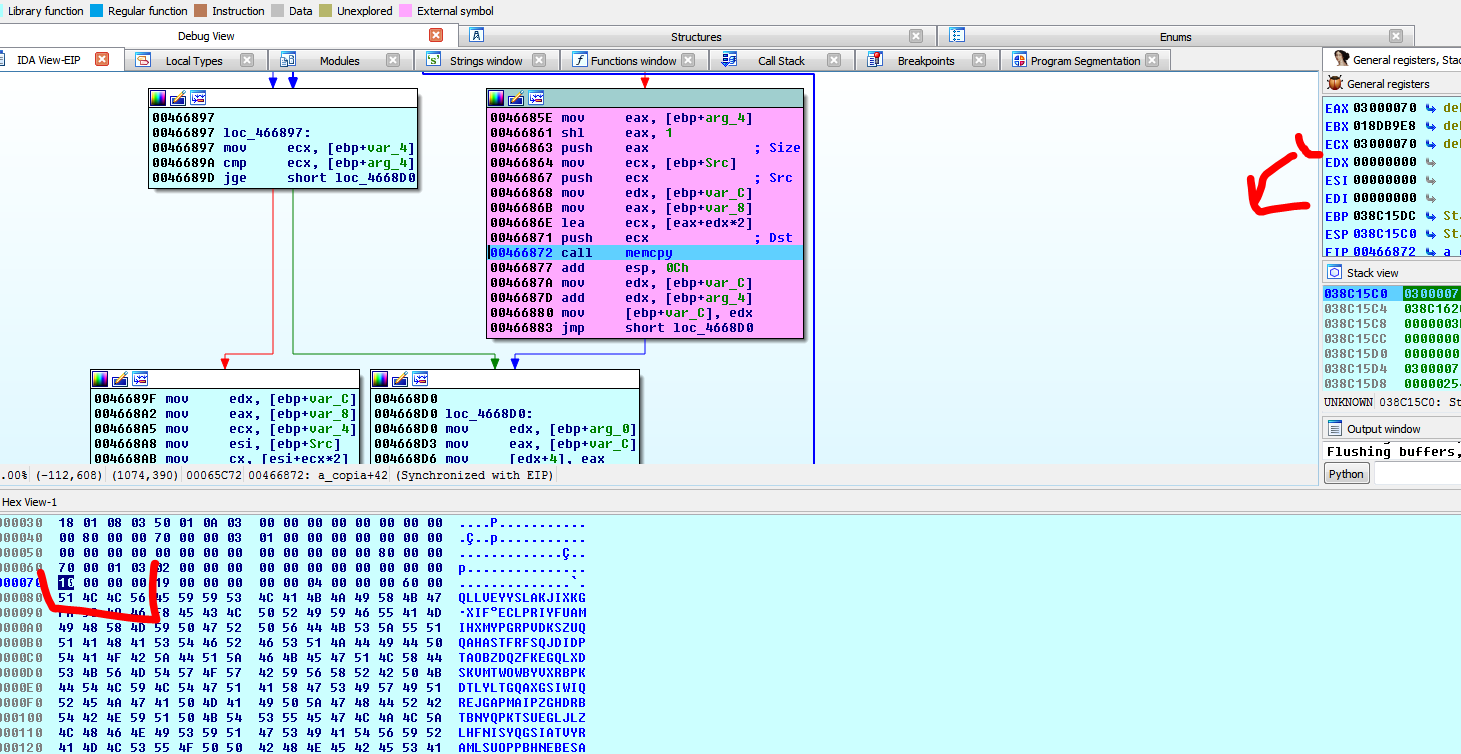


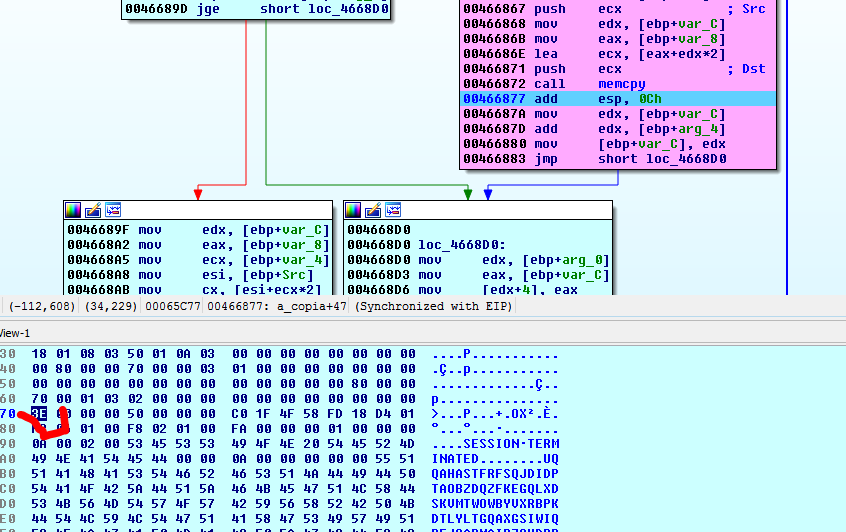
Vemos que ahí traceando llega al memcpy



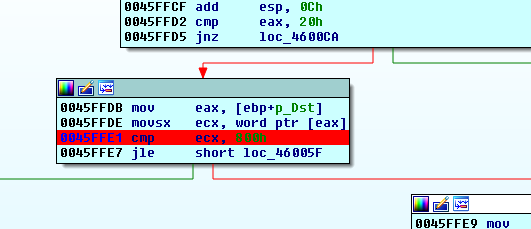
Vemos el 0x3e con la string sesión terminated.

Vemos que el Destination que tiene nuestra fruta va a ser pisado y el 0x10 se pisara con el 0x3e

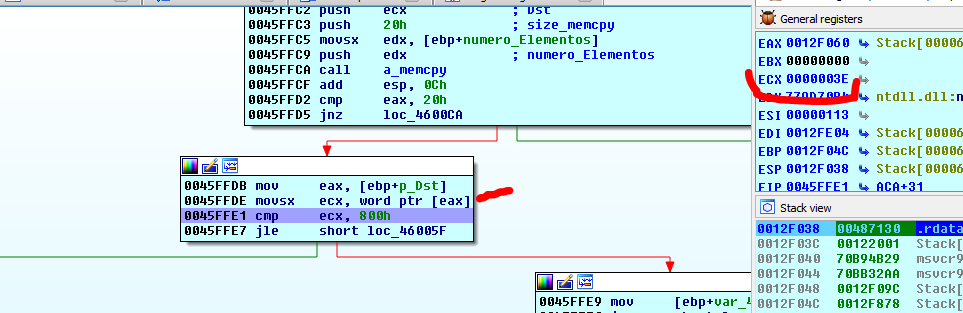




Si le pongo un Breakpoint on Access y pongo un breakpoint en la función vulnerable, leerá este valor



Le doy RUN.



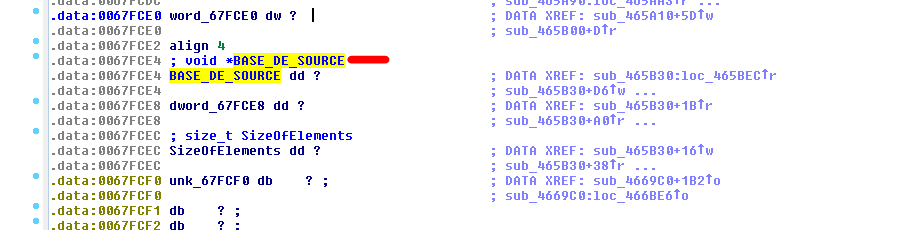
Y allí pasa, así que con esto terminamos el ejemplo.

Como ultimo me preguntaron si hay alguna forma de ver con IDA estáticamente si hay algún camino a la variable global para sacar donde copia y si hay un camino hasta allí estáticamente.

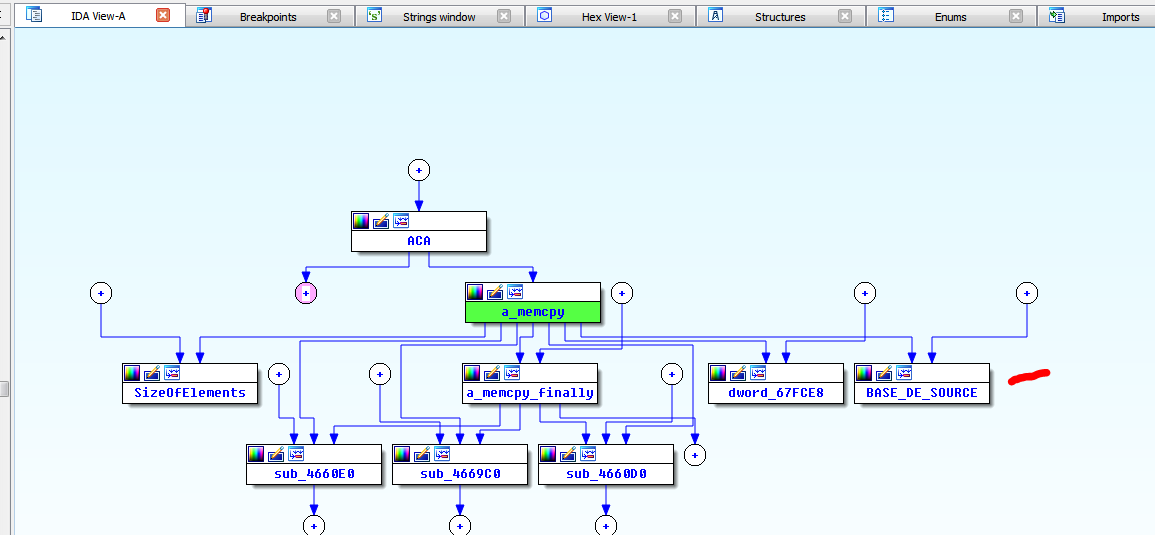
Vamos a la función donde copiaba.



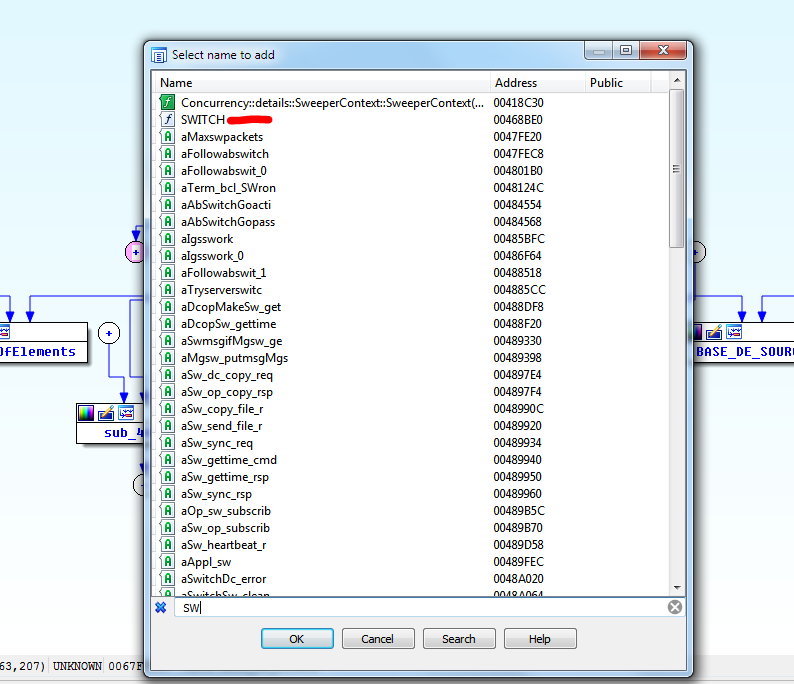
Hago doble click en la variable global.



Apreto la tecla menos.

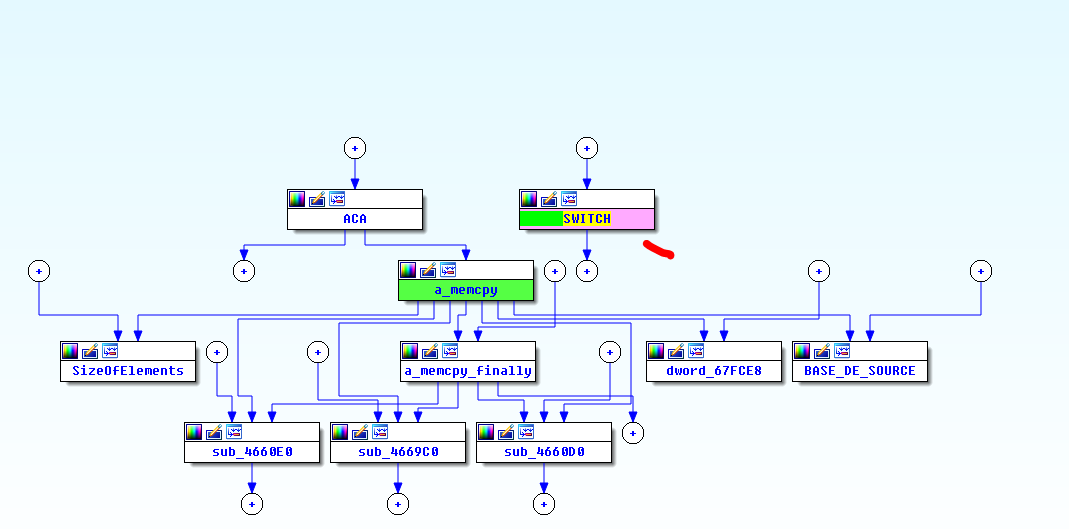


Luego hago click en una parte vacía -ADD NODE BY NAME



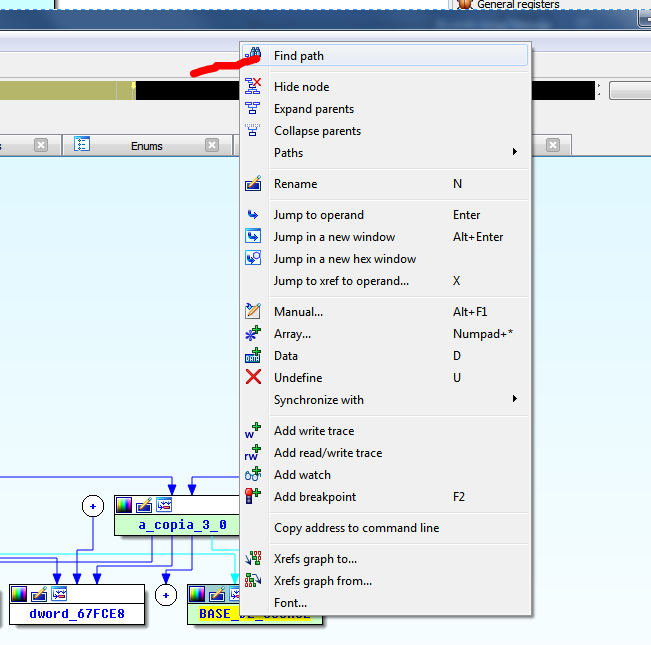
Agrego SWITCH.

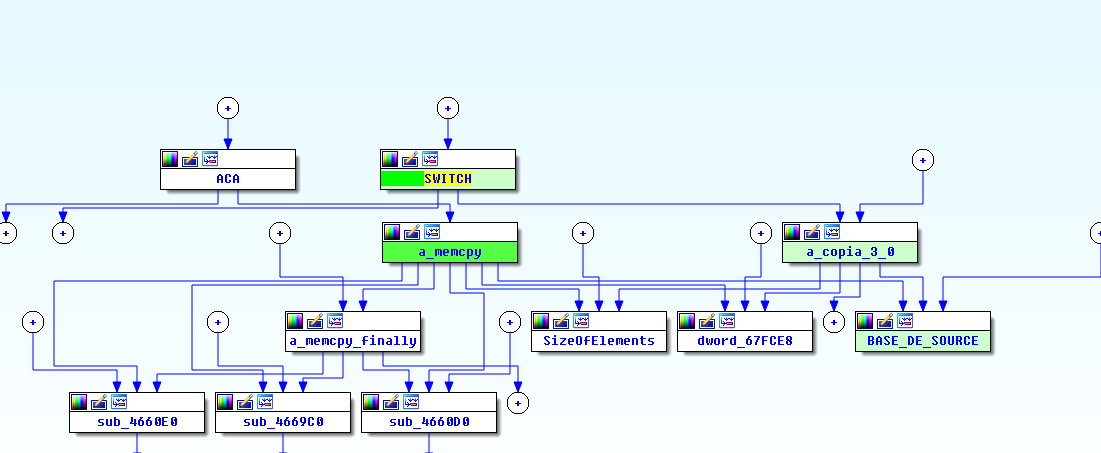
Vemos que quedo ahí suelta



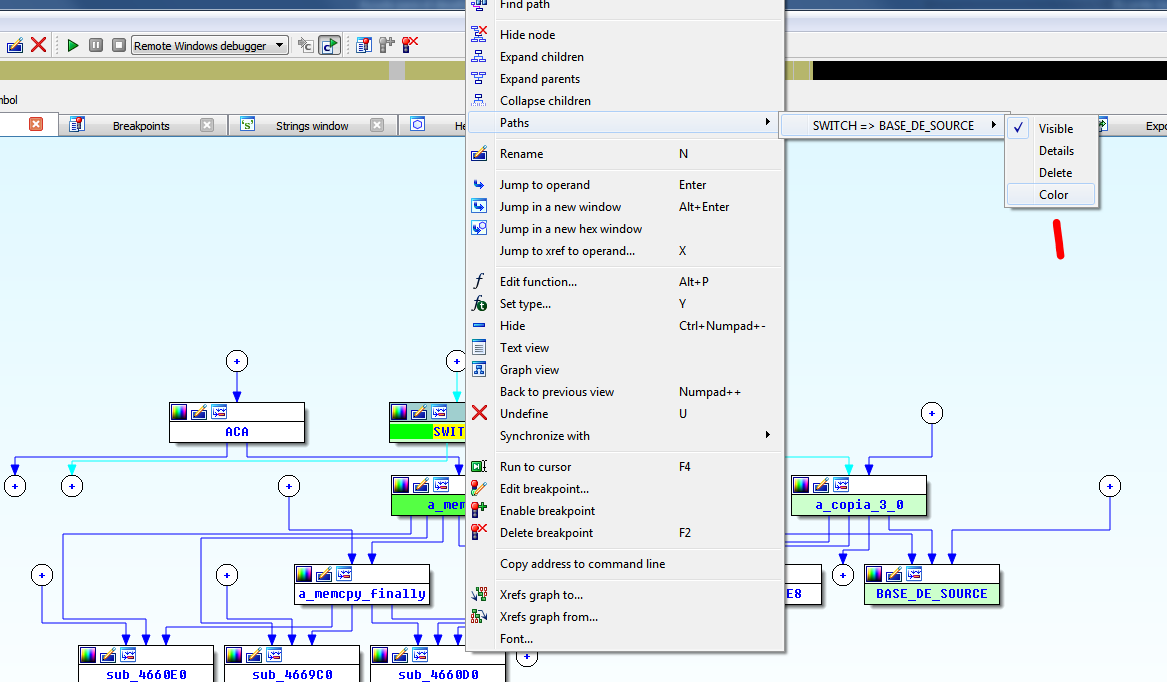
Veamos si hay caminos entre ambas, seguro habrá muchos.

Si hacemos click en switch y elegimos FIND PATH como BASE DE SOURCE no es una función no aparece en la lista de destinos, pero si hacemos al revés y hacemos FIND PATH en BASE DE SOURCE.

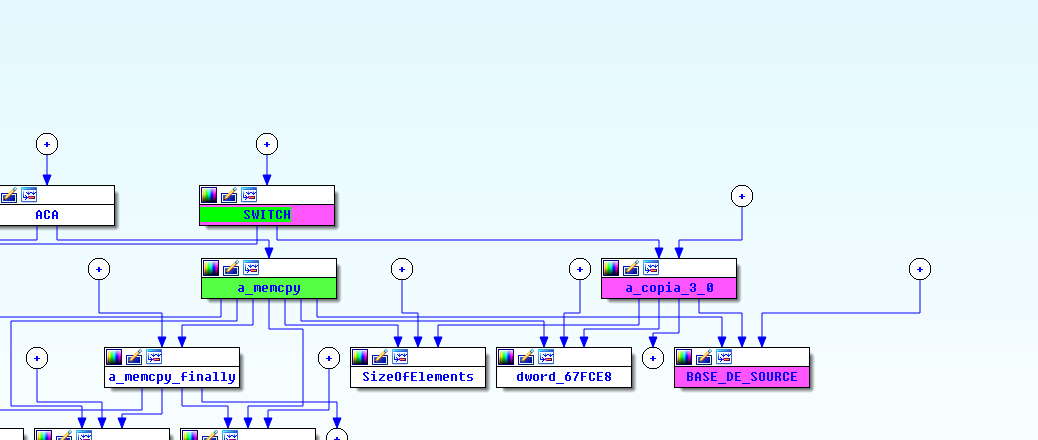




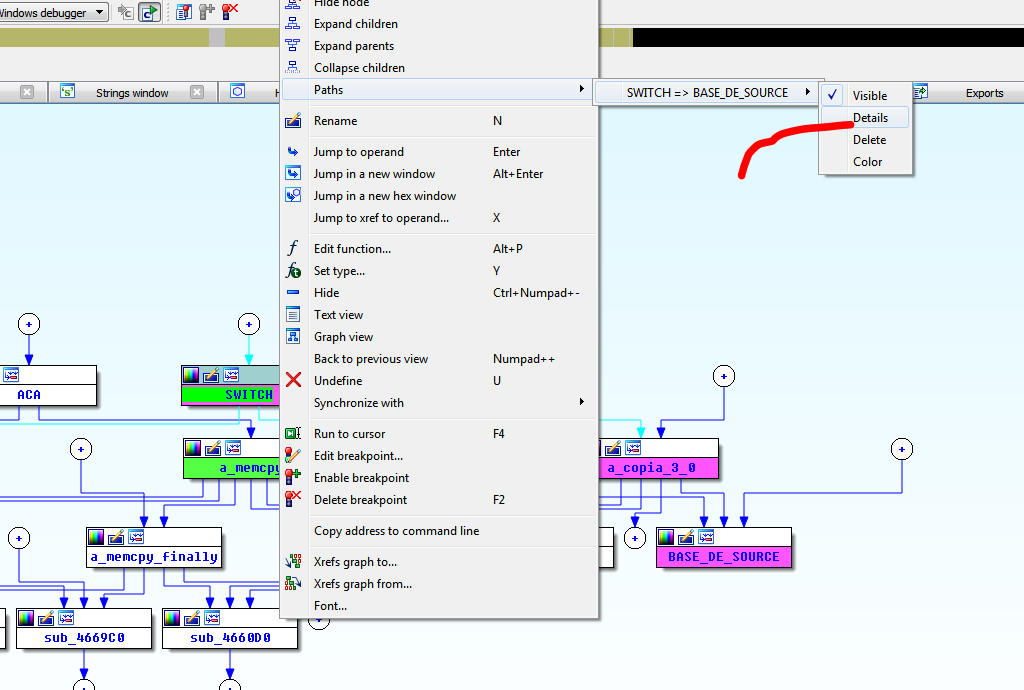
Lo podemos colorear si hacemos click en SWITCH-PATHS..

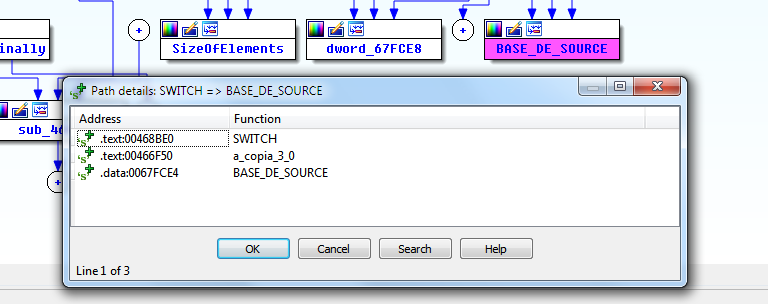


Ahora se ve mejor



Queremos la lista





Ahí esta el camino, desde SWITCH cualquier opcodes que te lleva a la función a\_copia\_3\_0 te llevara a un acceso a la variable global que necesitamos ver.

Bueno eso es todo ha sido un durísimo rival, pero como es un software real, tiene sus cosas, pero para vencerlo hay que practicar mucho para tener mas skill en reversing estático, este solo tirándole fruta no se lo puede sacar, y solo debuggeando tampoco.

Ricardo Narvaja