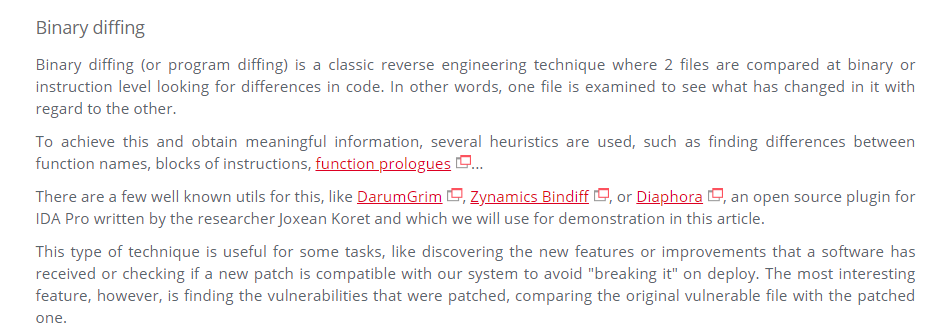
EXPLOITING Y REVERSING USANDO HERRAMIENTAS GRATUITAS (PARTE 3)

En la parte 2 hemos solucionado el ejercicio stack1 usando x64dbg, el tema es que el mismo es un debugger o sea es una tool que permite correr el programa, tracearlo, poner breakpoints, etc.

En el mismo debemos no solo ejecutar el programa, sino también poder llegar a ejecutar la función a analizar y así como es sencillo de utilizar, hay muchos casos donde se necesita realizar un análisis estático, o sea poder sacar conclusiones sin correr la aplicación o que corra lo menos posible (usado en análisis de virus, análisis de funciones de un programa que no llegamos a poder ejecutar, research de vulnerabilidades, reconstrucciones de código etc.)

En el caso de los exploit writers, cuando analizamos un parche que soluciona una vulnerabilidad en un programa, normalmente hacemos lo que se llama binary diffing o diff o sea comparamos con alguna tool la versión vulnerable contra la versión parcheada para tratar de encontrar el parche o sea donde se reparó en la nueva versión la función vulnerable para que ya no lo sea y desde allí poder armar un exploit.

El tema es que a veces hay cientos de funciones cambiadas y no todas son parches, la mayoría son correcciones menores, agregados de funcionalidades o cambios menores solamente, y para analizar una por una todas estas cambiadas y ver cual puede ser la que se arregló, debemos analizarlas detalladamente, y no solo es impráctico tratar de debuggear ejecutando todas, sino que a veces ni sabemos como se llega algunas de las funciones del programa, que puede tener miles de opciones combinadas que puede hacer que acceder a la función sea un trabajo muy complejo.



**Traducción: Binary diffing**

**Binary diffing (o Program diffing) es una técnica clásica de ingeniería inversa en la que se comparan 2 archivos a nivel binario o de instrucción en busca de diferencias en el código. En otras palabras, se examina un archivo para ver qué ha cambiado con respecto al otro.**

**Para lograr esto y obtener información significativa, se utilizan varias heurísticas, como encontrar diferencias entre nombres de funciones, bloques de instrucciones, prólogos de funciones ...**

**Hay algunas utilidades bien conocidas para esto, como DarumGrim, Zynamics Bindiff o Diaphora, un complemento de código abierto para IDA Pro escrito por el investigador Joxean Koret y que utilizaremos más adelante ya que es gratuito.**

**Este tipo de técnica es útil para algunas tareas, como descubrir las nuevas funciones o mejoras que ha recibido un software o verificar si un nuevo parche es compatible con nuestro sistema para evitar "romperlo" en la implementación. Sin embargo, la característica más interesante es encontrar las vulnerabilidades que fueron parcheadas, comparando el archivo vulnerable original con el parcheado.**

<https://www.incibe-cert.es/en/blog/importance-of-language-binary-diffing-and-other-stories-of-1day>

Más adelante en el curso realizaremos ejercicios de binary diffing para encontrar parches como parte del nuestro aprendizaje.

Hay ciertos programas que son desensambladores, pero interactivos o sea que no solo muestran las funciones y sus instrucciones, sino que permiten (según la pericia del reverser) ir detectando la funcionalidad de cada una, e ir trabajando en lo que ya veremos se llama reversing estático.

En general el reversing estático es una técnica muy poderosa cuando se domina, muchas veces nos ayuda también a encontrar el camino a la función deseada, y ciertas veces se puede complementar también con reversing dinámico.

Nosotros debemos tener pericia y dominar todas las técnicas, después usarlas y combinarlas lo mejor que podamos para llegar a nuestro objetivo.

También el reversing estático depende de si el programa tiene símbolos o no, hemos configurado la carpeta de símbolos al instalar Windbg, esta será una carpeta donde se irán descargando los símbolos automáticamente lo cual debería ocurrir para la mayoría de los archivos ejecutables del sistema, y si uno programó un ejecutable, podrá hacerlo con símbolos los cuales se guardarán en un archivo de extensión pdb.

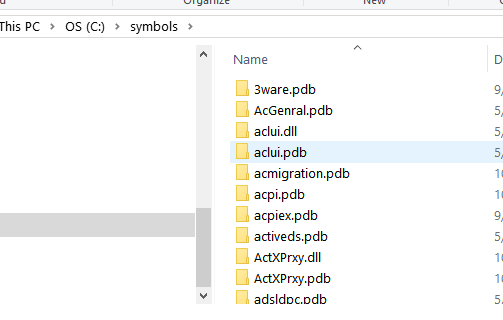
Sobre el tema símbolos aquí hay un link que lo explica bien claro en castellano

<https://laoubilette.wordpress.com/2015/03/13/pdb-simbolos-de-depuracion/>

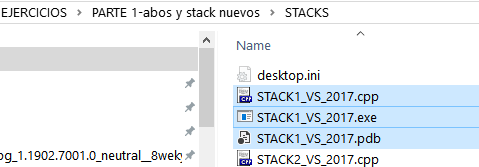
en inglés

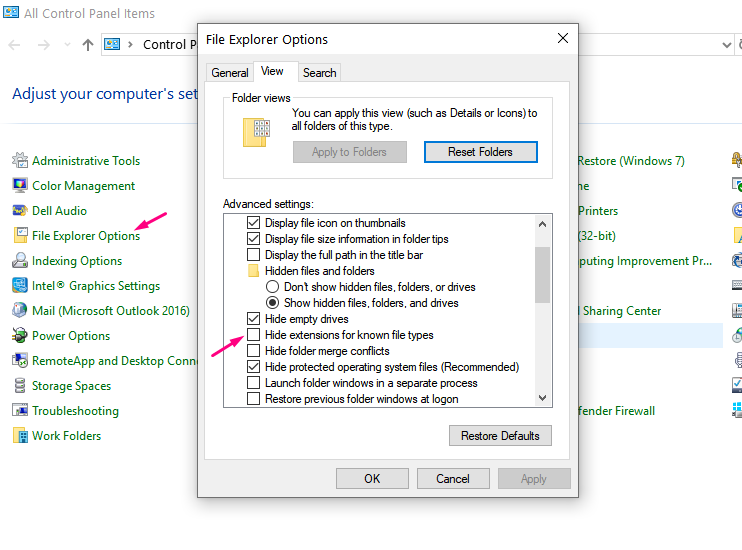
<https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-set-debug-and-release-configurations?view=vs-2019>

Posiblemente aún este vacía su carpeta de símbolos, pero a medida que uno trabaja con Windbg e IDA se irán bajando y guardando allí.



Obviamente poder contar con los símbolos hace más sencillo el análisis estático, empezaremos con nuestro análisis del stack1 con los símbolos, más adelante en el curso iremos encontrándonos con casos de programas sin símbolos (eso ocurrirá en la mayoría de los programas de terceros que no pertenecen al sistema operativo) y por supuesto requerirá más pericia en reversing estático, la que iremos adquiriendo poco a poco.



Vemos en la carpeta de ejercicios, tres archivos correspondientes al stack1, el ejecutable con extensión EXE, el código fuente CPP y el archivo de símbolos PDB. (si no pueden ver las extensiones deben ir a las opciones de carpeta, folder options o file explorer options como en las últimas versiones de Windows 10 y quitar la tilde de HIDE EXTENSION FOR KNOWN FILE TYPES en castellano debería ser algo como OCULTAR EXTENSIONES PARA TIPOS DE ARCHIVO CONOCIDOS mi sistema es en ingles por eso supongo que debe ser algo así.)

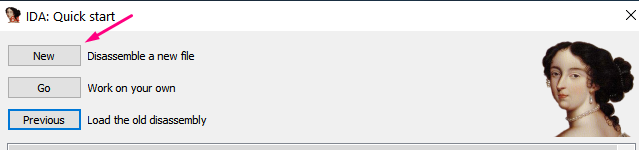
REVERSING ESTÁTICO

1-IDA FREE

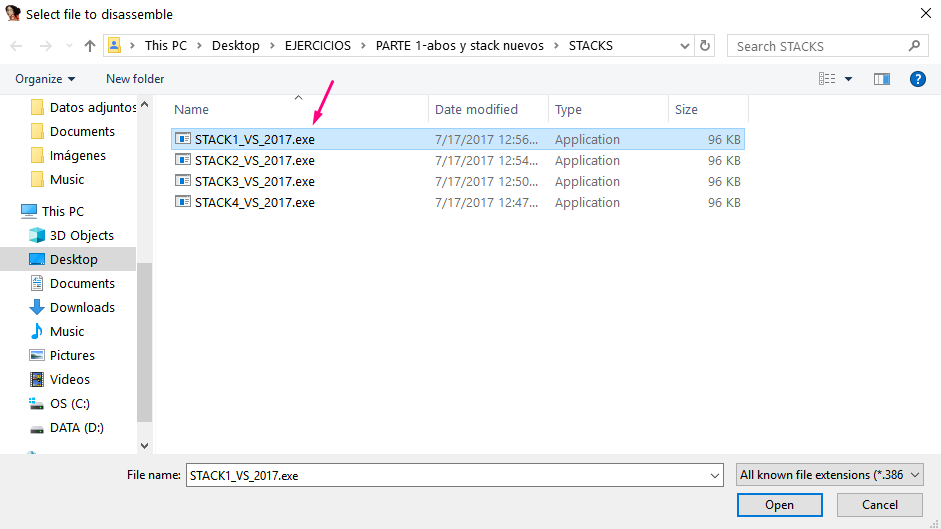
Bueno ya vemos las extensiones y que hay tres archivos así que abriremos primero el IDA FREE y nos pide que abramos un archivo, buscamos el exe que acabamos de mencionar y abrámoslo.



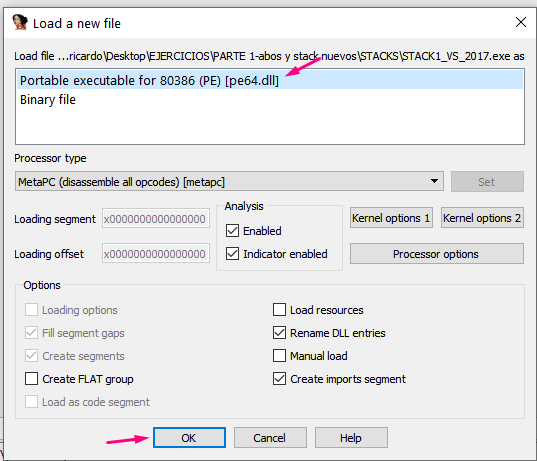
Elegimos NEW para trabajar sobre un archivo nuevo.



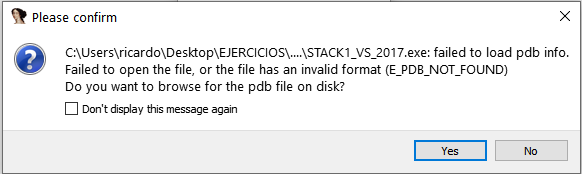
Buscamos el ejecutable del stack1.

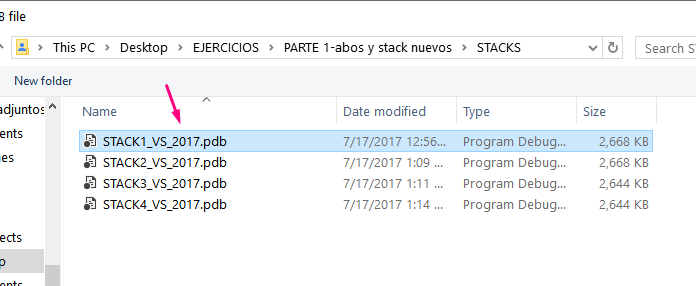


Detecta que es un ejecutable PE como el IDA FREE no trae como el pro dos versiones una para 32 bits y otra para 64 bits, te dice que es 64 pero igual funciona.

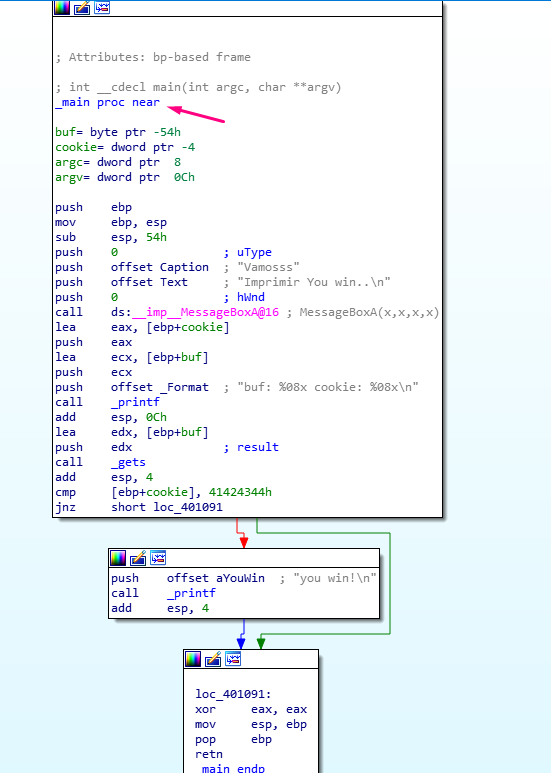


Dice que no encuentra el PDB porque no esta en la carpeta símbolos, igual elegimos YES y lo buscamos manualmente.

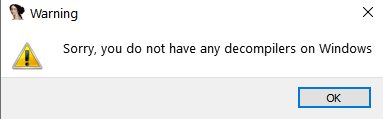




Luego como tiene símbolos detecta la función main fácilmente y nos las muestra directamente.



El IDA FREE a diferencia de la versión PRO no tiene decompilador, si apreto f5 que es el shortcut para decompilar en la versión PRO.



No importa lo que no te mata, te hace más fuerte, jeje, sigamos.

Vemos en la imagen las llamadas a printf y gets, también la comparación de la cookie con 0x41424344 y podemos ver que si no son iguales puede ir por dos caminos el de la flecha verde o la flecha roja.

FLECHA VERDE = RESULTADO DEL SALTO CONDICIONAL VERDADERO.

FLECHA ROJA = RESULTADO DEL SALTO CONDICIONAL FALSO.

En este caso en particular

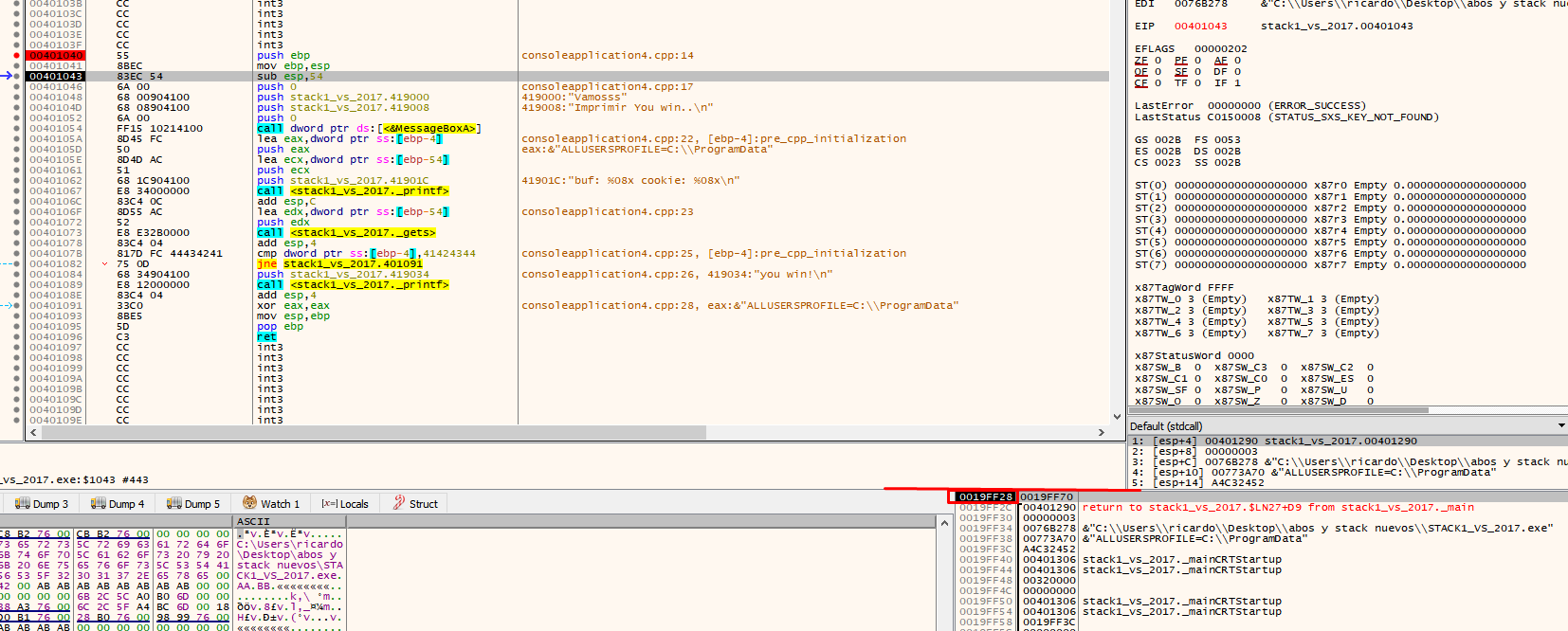
**JNZ O JNE (JUMP IF NOT EQUAL) =TRUE** (o sea es verdadero el resultado si no son iguales) o sea irá por la flecha verde si no son iguales y por la roja si son iguales.

Otra cosa importante es relacionar mentalmente lo que vimos al tracear en el x64dbg con lo que nos muestra el IDA.

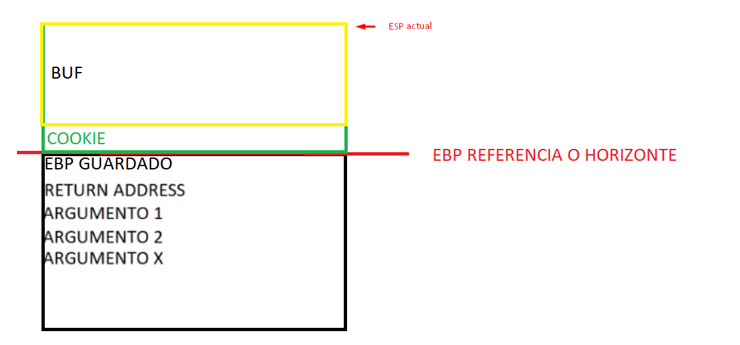
Recordamos que después del PRÓLOGO de la función, se seteaba en este caso como la función es EBP BASED, el valor de EBP que sería constante a partir de allí hasta el EPÍLOGO al terminar la función y todas las variables y argumentos se referencian con respecto a este valor de EBP que llamamos HORIZONTE que se mantiene constante entre ambos.



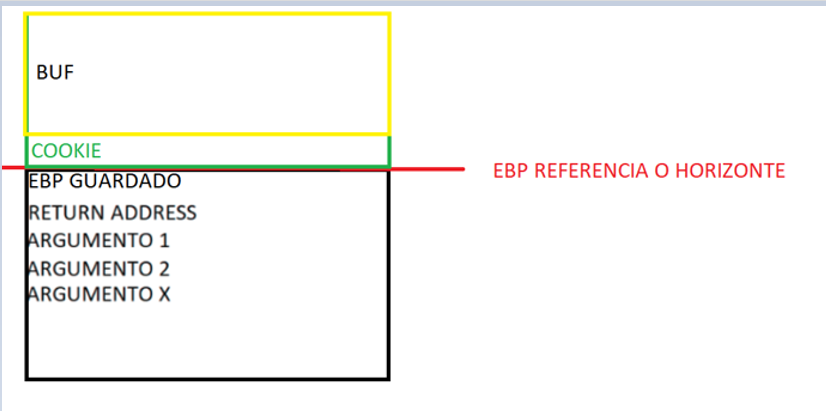
Vemos el HORIZONTE la línea en el stack



Debajo de EBP que quedaba al valor de referencia (HORIZONTE), estaban el EBP GUARDADO, EL RETURN ADDRESS y los argumentos de la función y arriba del horizonte se reservará el espacio para las variables en la siguiente instrucción al ejecutar el SUB ESP, 0x54, quedando ESP moviéndose arriba de EBP que queda fijo.

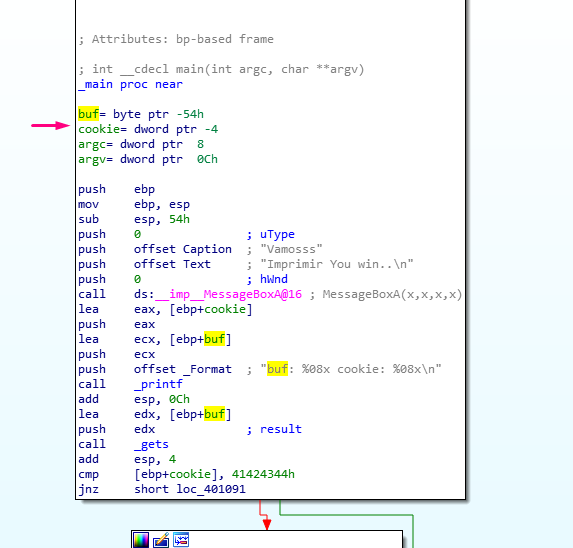


Igual el valor de ESP como vimos va variando en diferentes momentos dentro de la función, pero es EBP el que se mantiene constante y este mapa será la distribución de la variables y argumentos de la función siempre, no cambiará ya que EBP no cambia y las distancias con respecto al mismo tampoco.



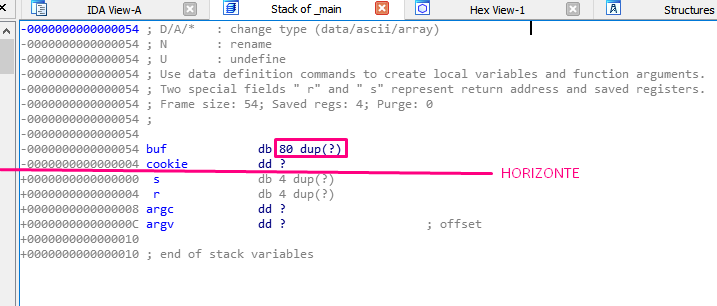
Todo este mapa lo construimos traceando en el x64dbg, pero podemos verlo en IDA sin tener que ejecutar.

Debajo de la declaración de la función el IDA tiene la lista de variables y argumentos, pero no es el mismo mapa completo que obtuvimos al tracear, para obtenerlo, hacemos doble clic en cualquier variable o argumento.



Eso nos mostrará la REPRESENTACIÓN ESTÁTICA DEL STACK, que es el mismo mapa que habíamos obtenido traceando el x64dbg

El mismo será una foto del stack con las variables, argumentos, EBP GUARDADO, RETURN ADDRESS, HORIZONTE, etc.



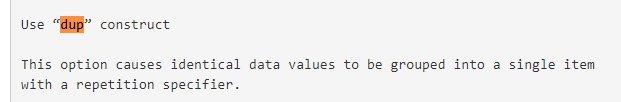
En IDA

**db** significa BYTE=1 byte de largo

**dw** significa Word= 2 bytes de largo

**dd** significa DWORD=4 bytes de largo

Por lo tanto, la variable **buf** que es del tipo **db** o sea del tipo **BYTE**, pero de largo **80**  es un array de bytes de largo 80 bytes.



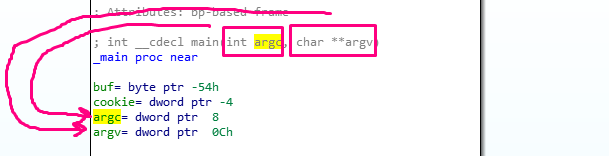
Al lado dice **dup(?)** sea DUP es DUPLICAR o repetir tantas veces y como no está inicializado por eso es dup(?) porque repite un valor desconocido para el análisis estático.

Luego viene **cookie** que es un **dd** o sea un **DWORD** por lo que su largo es 4 bytes, al lado el símbolo “?” que significa que es un solo DWORD desconocido,no necesita el dup porque es un solo DWORD, no repite.

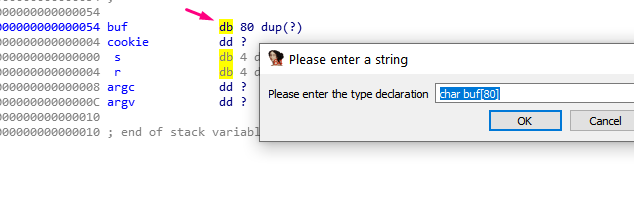
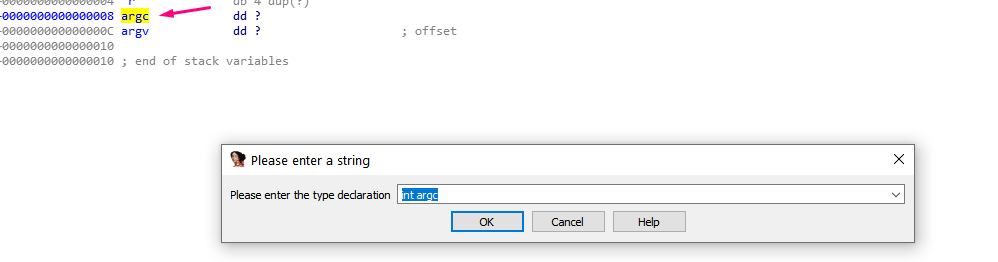
Luego viene **s** que es el **STORED EBP** cuyo largo es 4 bytes, aunque este definido como **db** de largo 4, realmente es un DWORD pero no hay problema, IDA lo define así, repite ya que es del tipo BYTE y sería un array de 4 bytes desconocidos.

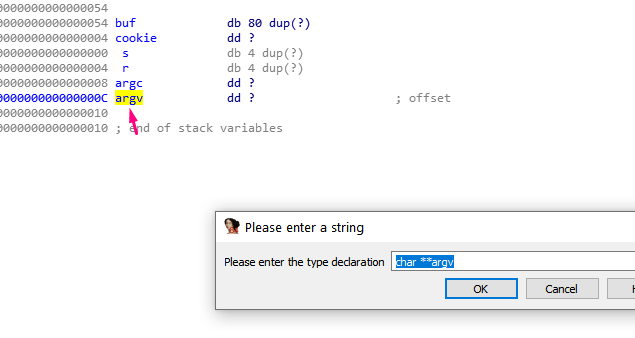
Luego viene **r** que es el **RETURN ADDRESS** lo mismo que el anterior su largo es 4 bytes.

Abajo vienen argc y argv que son cada uno de 4 bytes y están reconocidos como DWORDS.

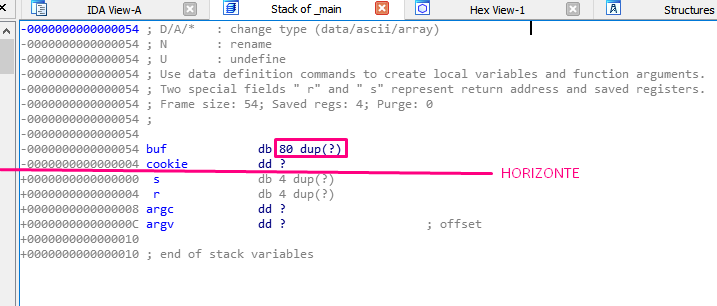


En la representación del stack, más que nada respeta el largo de cada uno, si hacemos clic derecho en cualquiera veremos la definición del tipo de acuerdo con C.



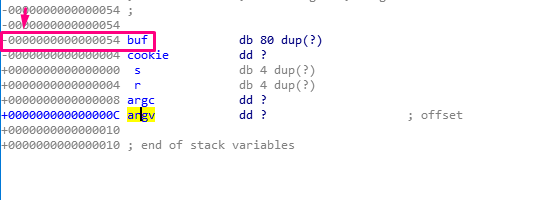
Obviamente esta relación IDA la puede efectuar porque posee los símbolos, ya veremos cómo se hace cuando no se poseen.



Volviendo al mapa vemos en una primera columna, lo mismo que veíamos en el stack del x64dbg cuando ponemos a cero el horizonte, los valores de distancia con referencia al mismo, hacia arriba como habíamos visto las variables eran EBP -XXX y hacia abajo del horizonte eran EBP+XXX

Por eso allí ya vemos que por ejemplo **buf** está ubicada en **EBP – 0x54** y **cookie** en **EBP – 4**, mientras que **argc** está ubicada en **EBP + 8** y **argv** en **EBP +C.**

Aquí vemos en la imagen que buf está en EBP – 54

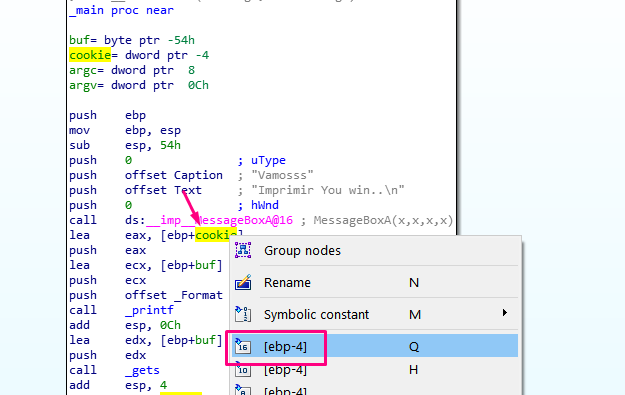




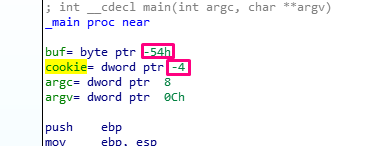
Vemos que coincide con ebp-4 y ebp-54 de cookie y buf en el x64dbg.

Igual veo que en las instrucciones **ebp+cookie**, si hago clic derecho nos muestra como alternativas que es similar a ebp-4 ya que cookie estaba ubicado en la posición -4

ebp+cookie = ebp +(-4) = ebp-4

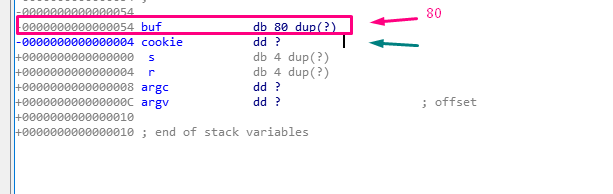


También en la definición de las variables vemos la posición con respecto a EBP.

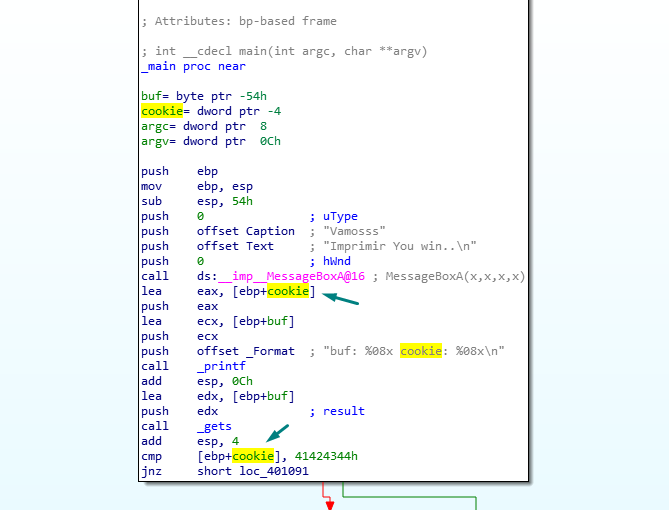


Así que todo lo que tuvimos que ejecutar lo sabemos solo analizando en IDA, lo único que nos falta es la distancia que tengo que llenar para overflodear el buffer y pisar cookie.

Eso se puede ver en la representación estática del stack.



Tengo que llenar los 80 de buf y luego vienen los 4 de cookie que como veo en el mismo IDA se compara con 0x41424344.



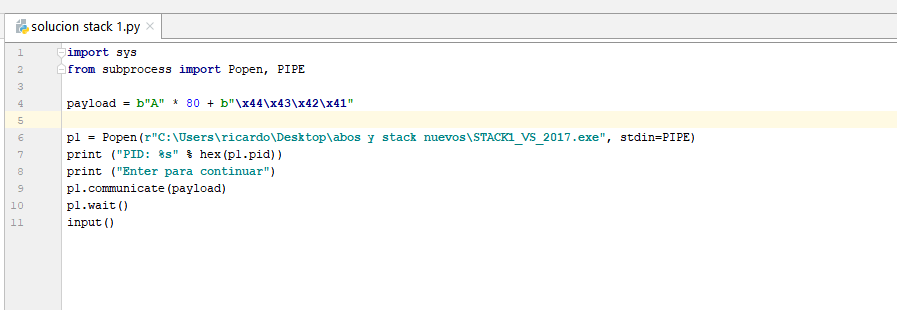
Allí están todos los lugares donde se accede a cookie, el LEA es similar al AMPERSAND obtiene la dirección de una variable no su valor, eso se pasa al printf como vimos para imprimir la dirección en hexadecimal por el %08x y luego se compara con 0x41424344.

También veo que al gets se le pasa como argumento la dirección de buf, así que copiara desde allí, así que enviándole

80 Aes + “DCBA”

Ya que DCBA es 44 43 42 41 y al leerlo de la memoria al revés por little endian cuando compara será 0x41424344 y serán iguales e irá por la flecha roja al ser JNE= NOT TRUE la evaluación de que no son iguales no es verdadera y va por la flecha roja y va a YOU WIN.

El script es similar al que vimos en la parte anterior.



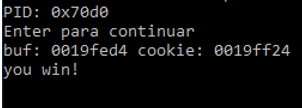
Uso Popen para redireccionar la entrada STDIN, de esta forma en vez de tipear le envío los datos desde el script con **p1.communicate(payload)**

**Payload**: Es la carga de un exploit la cual se encarga de realizar la parte maliciosa de la intrusión, es decir, es el código remoto que se ejecutará en la máquina vulnerada, así creando una secuencia de actividades maliciosas.

En este caso el payload es = 80 Aes + “DCBA” # (“\x44\x43\x42\x4” es similar a “DCBA”)



Si corro el script con lo que he podido deducir con mi reversing estático con IDA sin ejecutar el ejercicio, veo que llego a YOU WIN sin problemas.



2-RADARE

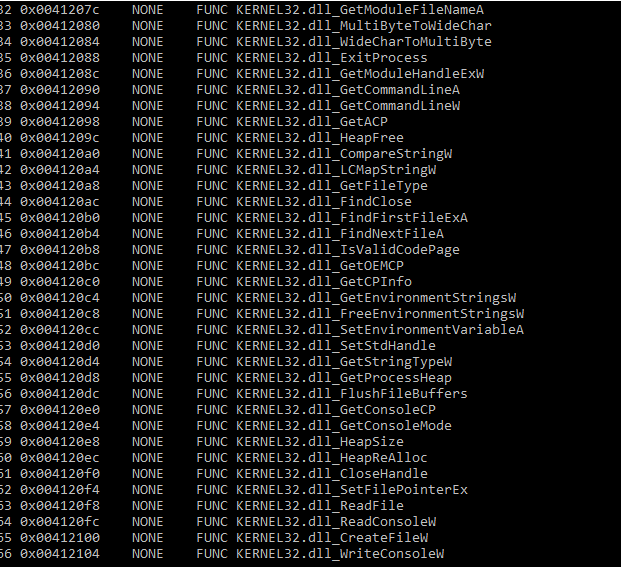
Lo primero sería ver la información del binario, para ello existe un ejecutable llamado rabin2 en la misma carpeta donde se instalo el radare.

**rabin2 -I nombre**

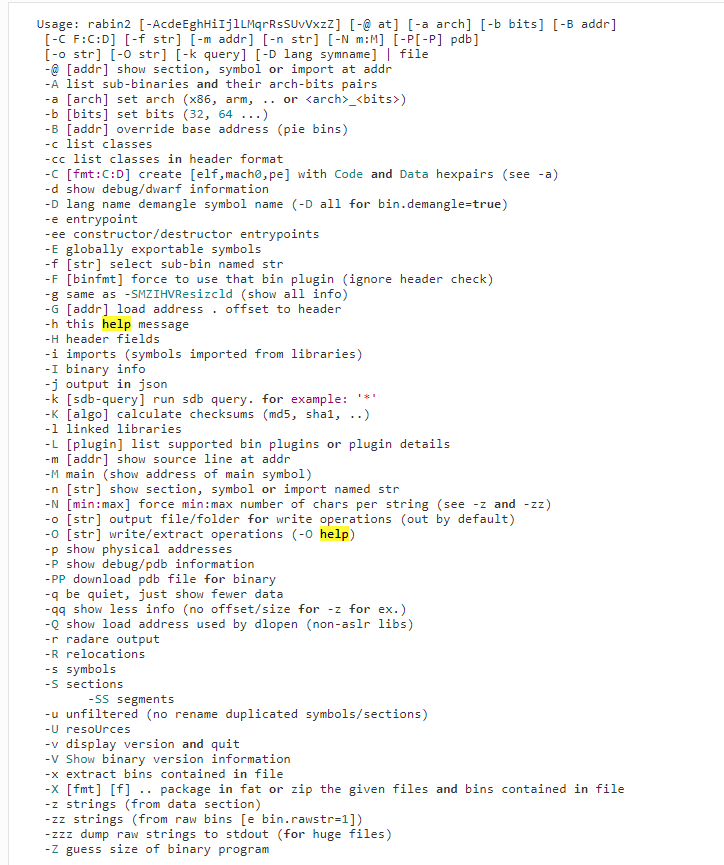
nos devuelve la info del binario y tiene muchísimas opciones que se pueden ver con el argumento -h



Por ejemplo, el argumento -i es para ver las imports o funciones importadas por el ejecutable.



y así hay muchísimas opciones diferentes para obtener info.



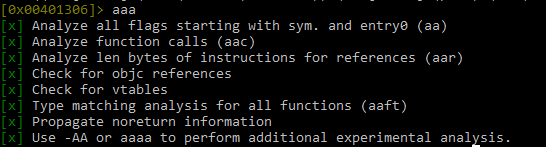
Empecemos a usar radare en una consola tipeamos

**radare2 STACK1\_VS\_2017.exe**

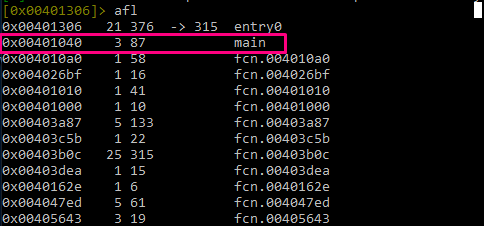
Eso cargará el ejecutable para analizarlo.

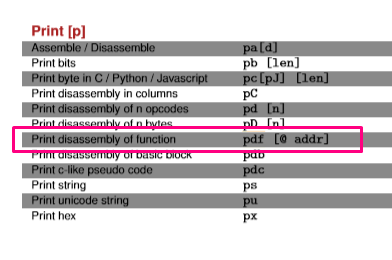


Luego tipeo **aaa** para que analice



Luego **afl** listará todas las funciones, trato de hallar el main, veo que está en 0x401040.





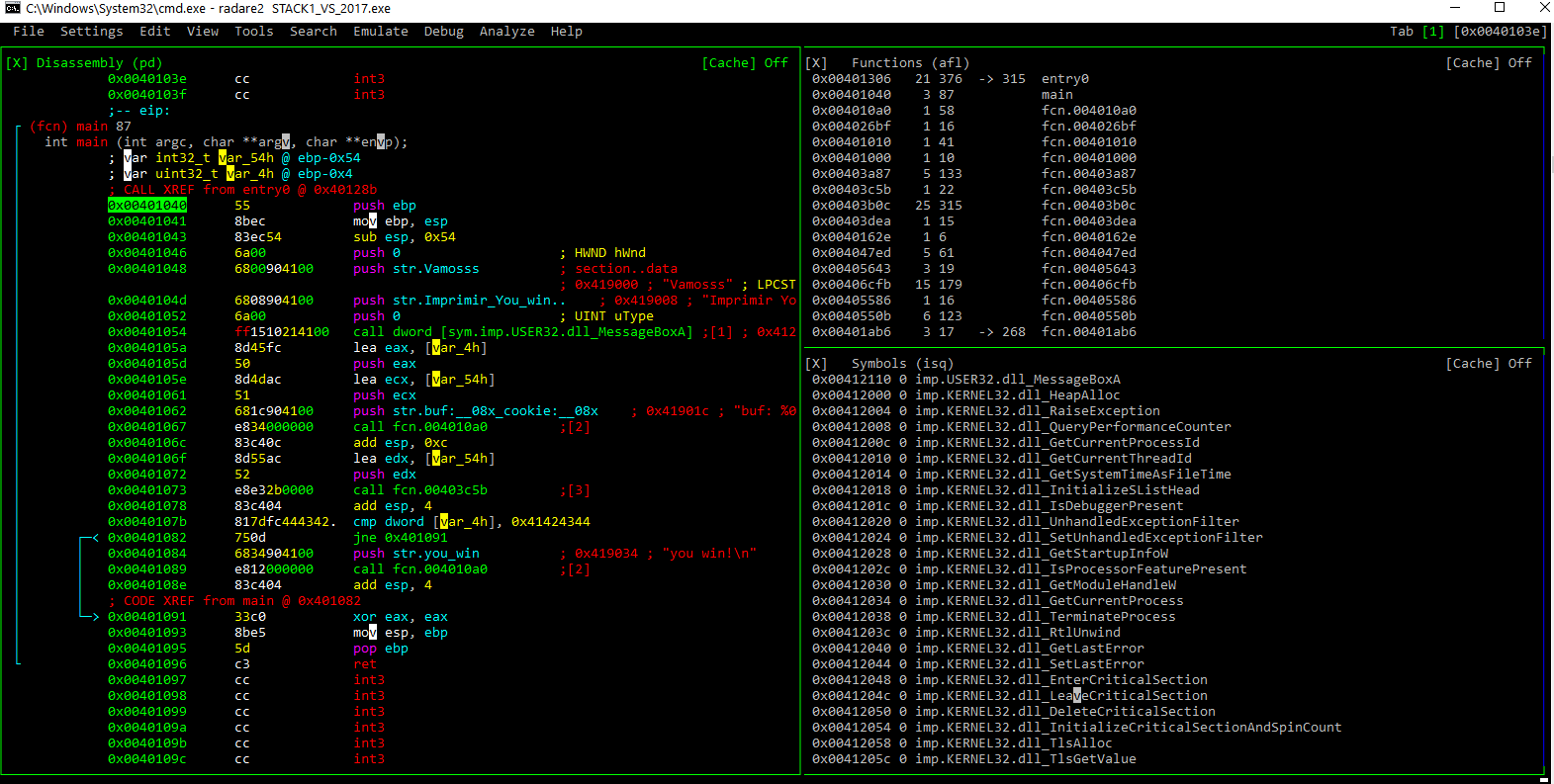
Con el comando **pdf** puedo desensamblar una función.



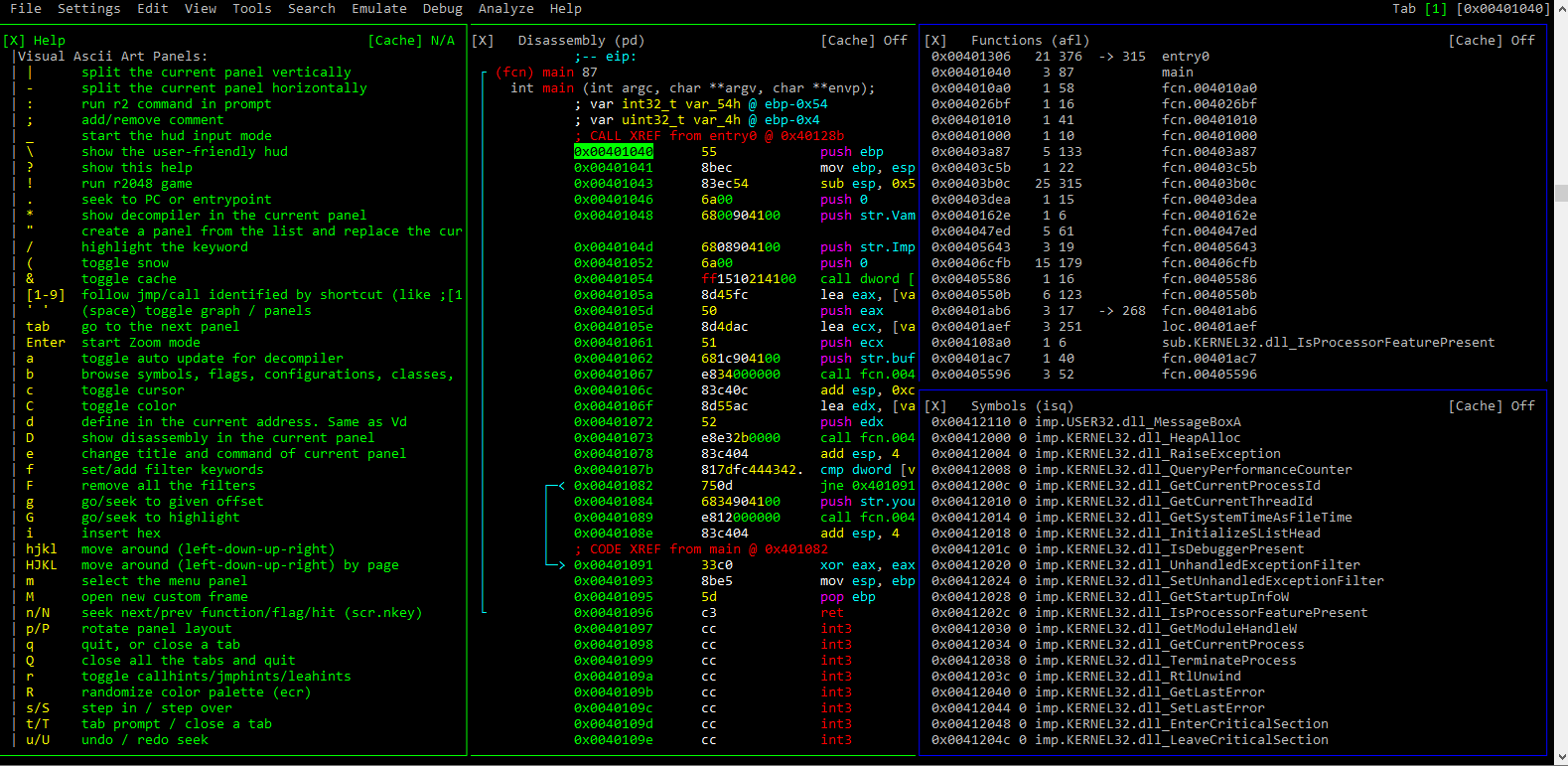
Con la instrucción eco podemos listar los temas en mi caso me gusto el bright, porque se ve más claro.



En radare tenemos el modo consola que tiene sus comandos y el modo visual que se entra con la letra v y se sale con la letra q.



Para entrar en el modo cursor y salir se aprieta la c y para la ayuda se aprieta la h.

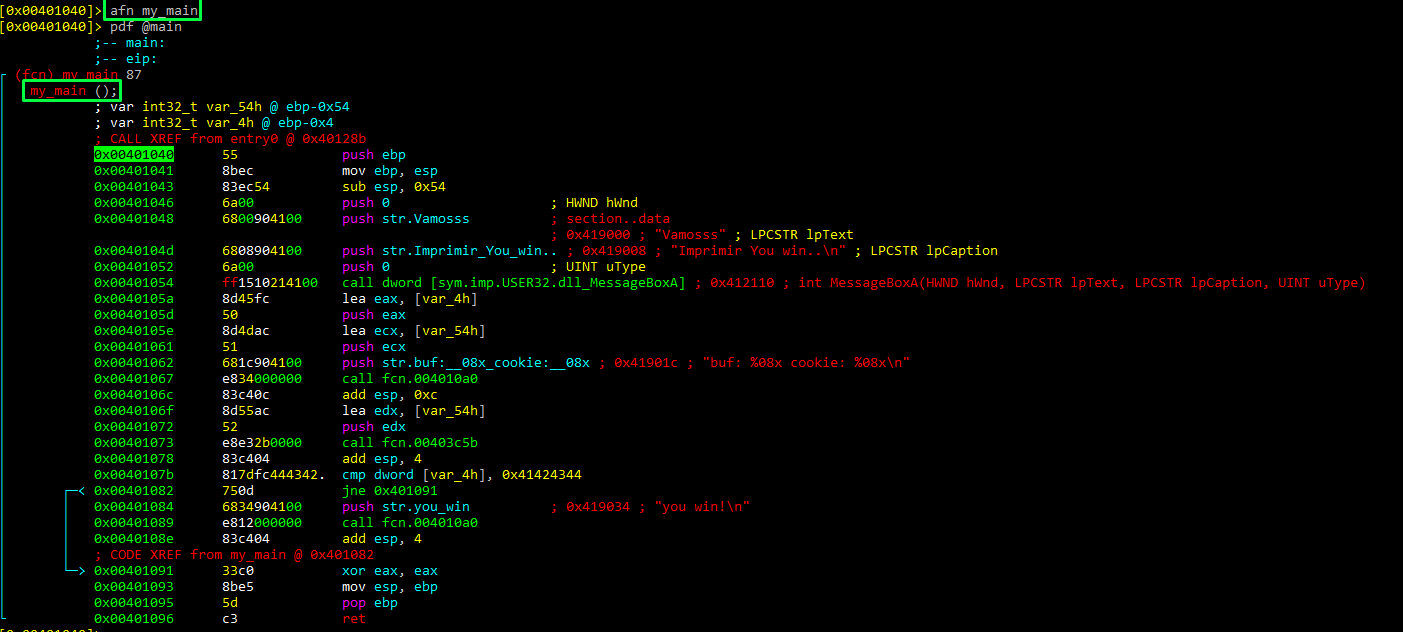


Por ahora nos mantendremos en modo consola, creo que es más acotado para trabajar sobre todo cuando uno no lo domina completamente, más adelante veremos de usar el modo visual y Cutter que es una GUI para radare.

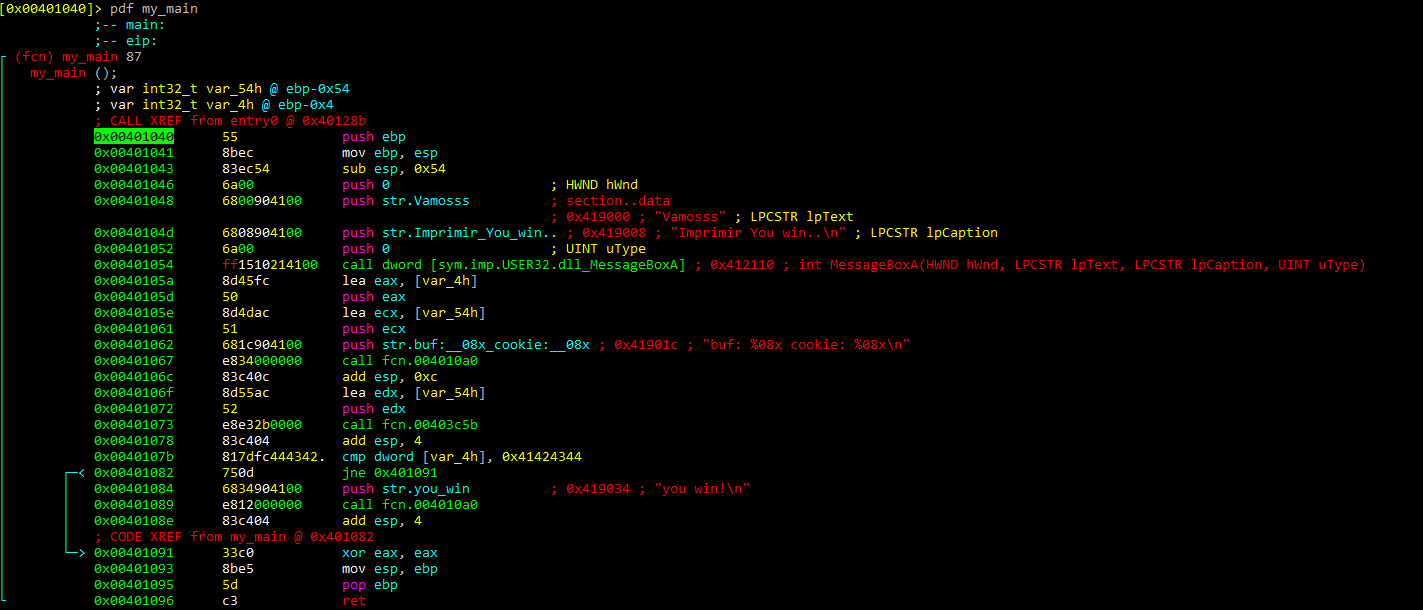


Vemos que el cursor nos muestra que esta en 0x401054, para cambiar a main podemos hacer **s main** y podemos trabajar cómodos ya que toma como referencia la dirección actual que marca el cursor, ahora marca 401040 que es la dirección de main, si no cambiamos tendremos que tipear **@address** en cada comando y es más molesto.

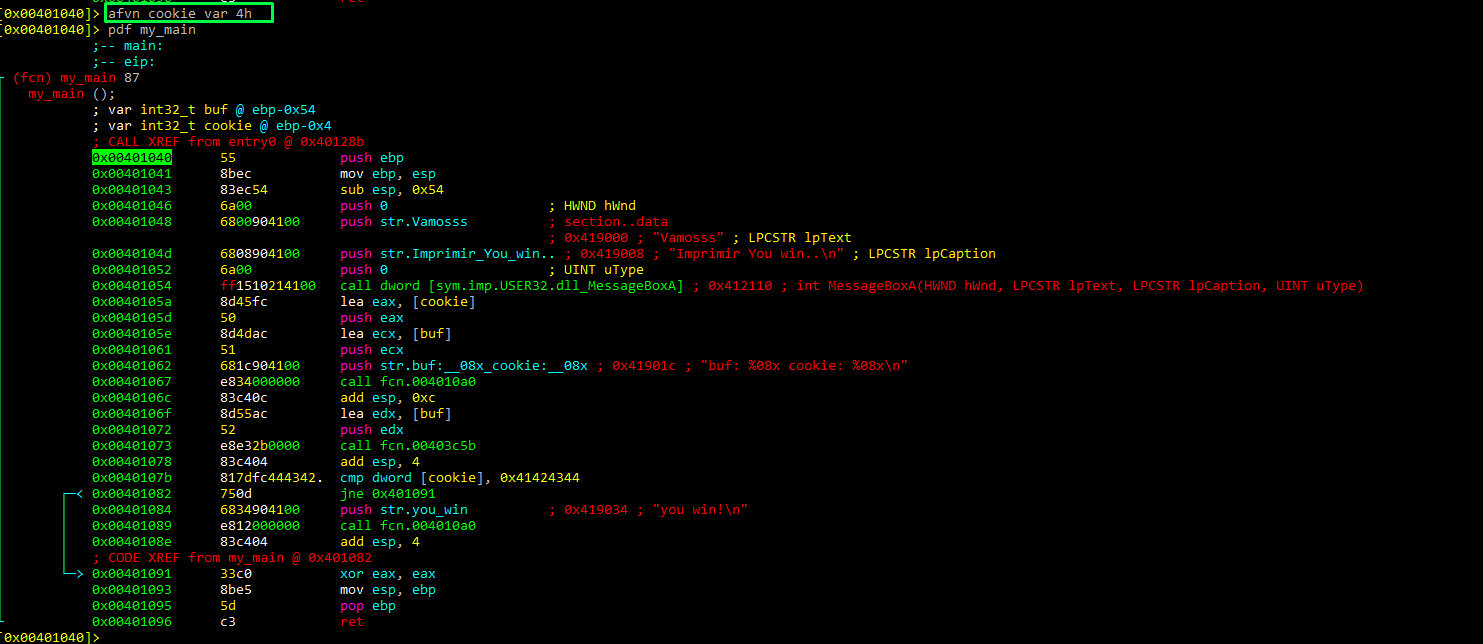
Ahora renombramos la función con **afn** y luego



Puedo desensamblar usando el nuevo nombre my\_main

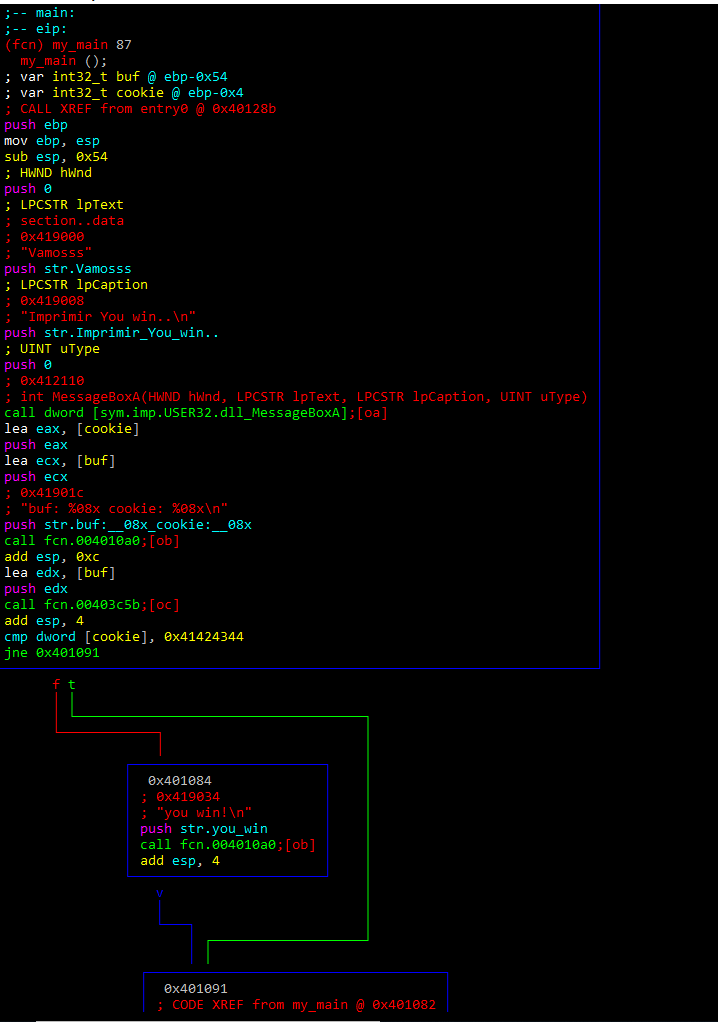


Renombro las variables con **afvn** nuevo\_nombre viejo\_nombre



Vemos que en toda la función quedaron con los nombres nuevos.

Con el comando **agf** podemos ver una representación ASCII visual de la función.

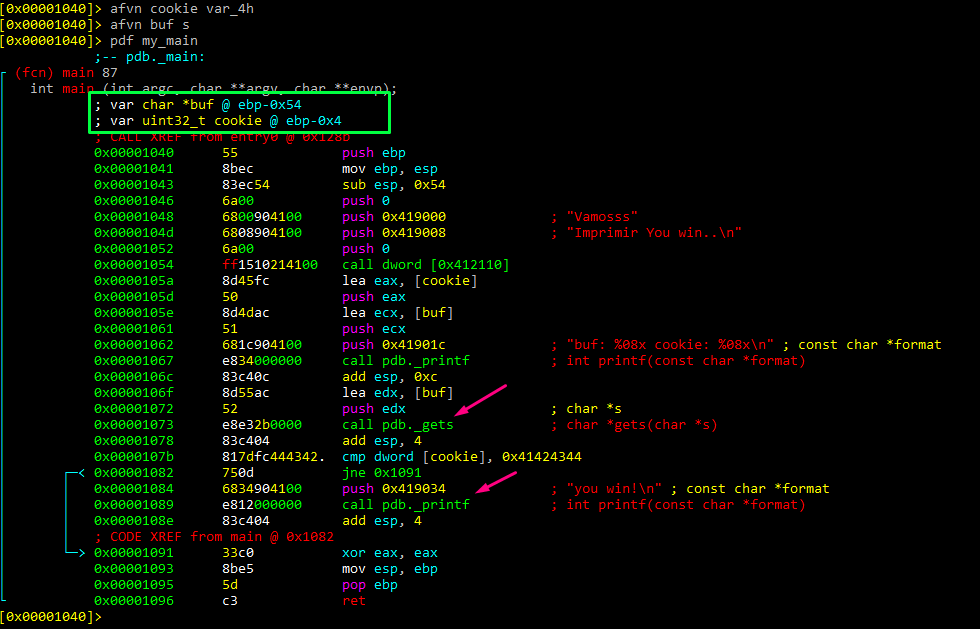


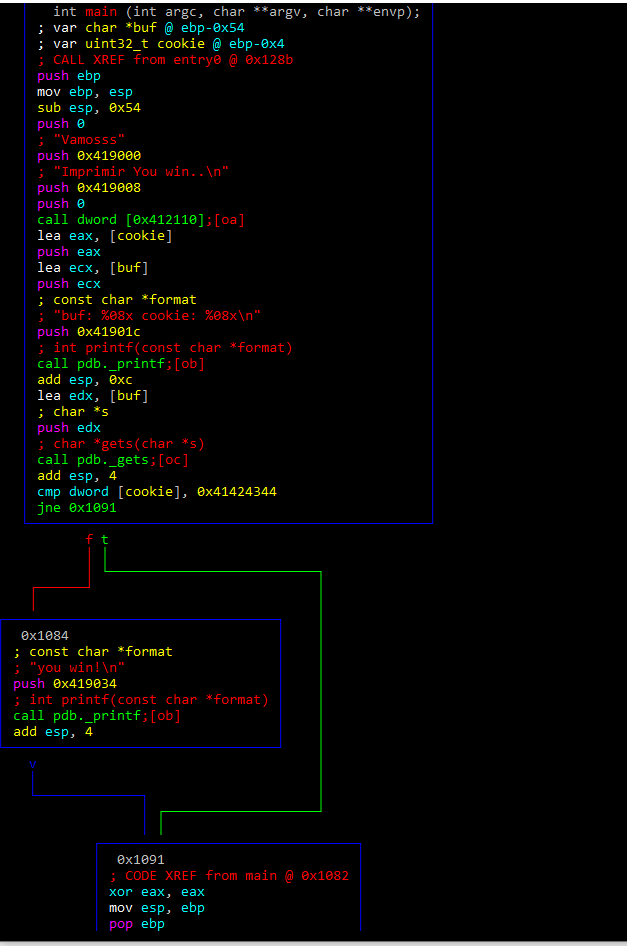
Vemos la flecha verde true y la flecha roja false y la comparación con 0x41424344, lo que no vemos es gets.

Le agrego la información del archivo de símbolos pdb con el comando **idp.**

**idp STACK1\_VS\_2017.pdb**

Vuelvo a analizar con **aaa** y agrega los símbolos, ahora ya aparecen el **gets** y el **printf** el tema es que tengo que volver a renombrar las símbolos (nota cargar los símbolos al inicio antes de analizar con **aaa**, sino se pierde trabajo hecho)

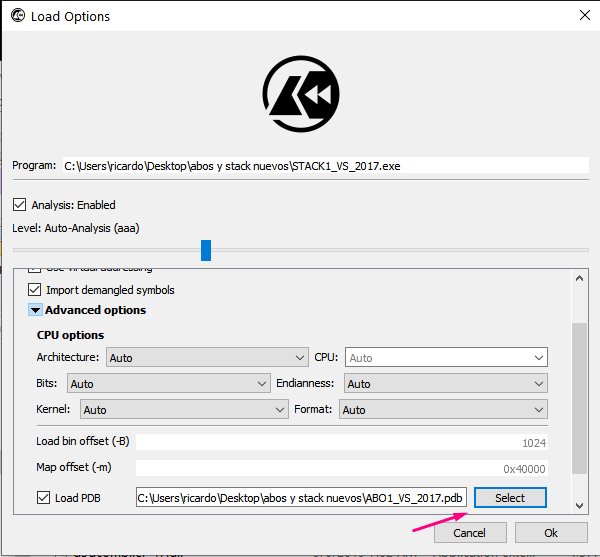




Ahora si quedo mejor y el gráfico también.

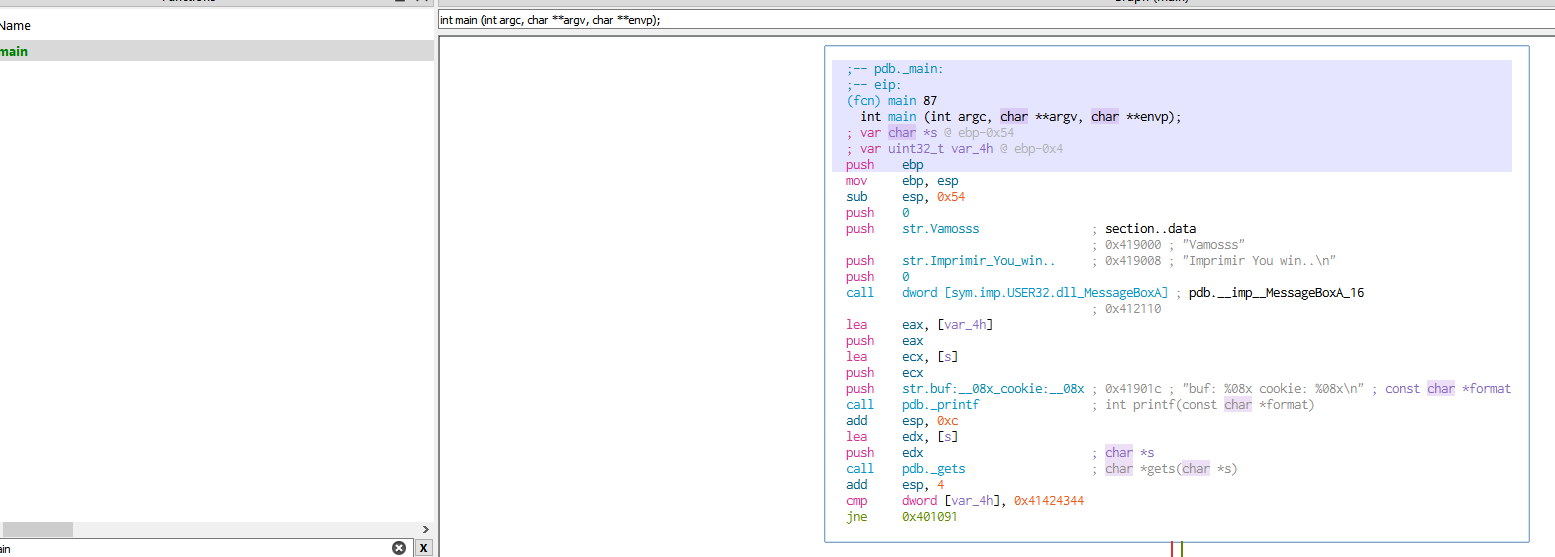
Veremos como se ve en Cutter que es una GUI para usar con radare, lo bajo de aquí lo descomprimo y ejecuto.

<https://github.com/radareorg/cutter/releases>

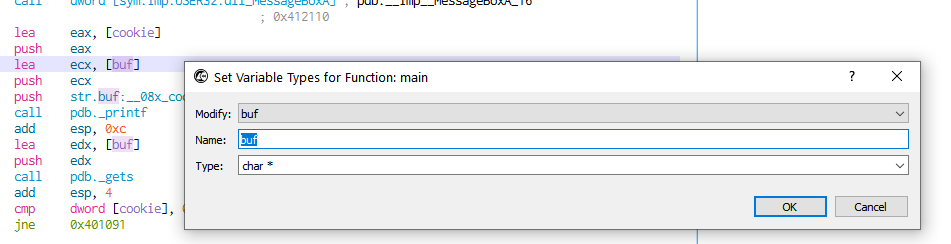


Selecciono el archivo a desensamblar y el pdb con los símbolos.

En el quick filter escribo main y puedo ver la función, al hacer click encima y al apretar con la barra cambiar de modo gráfico a listado.



Haciendo click derecho o con el shortcut Y puedo renombrar una variable

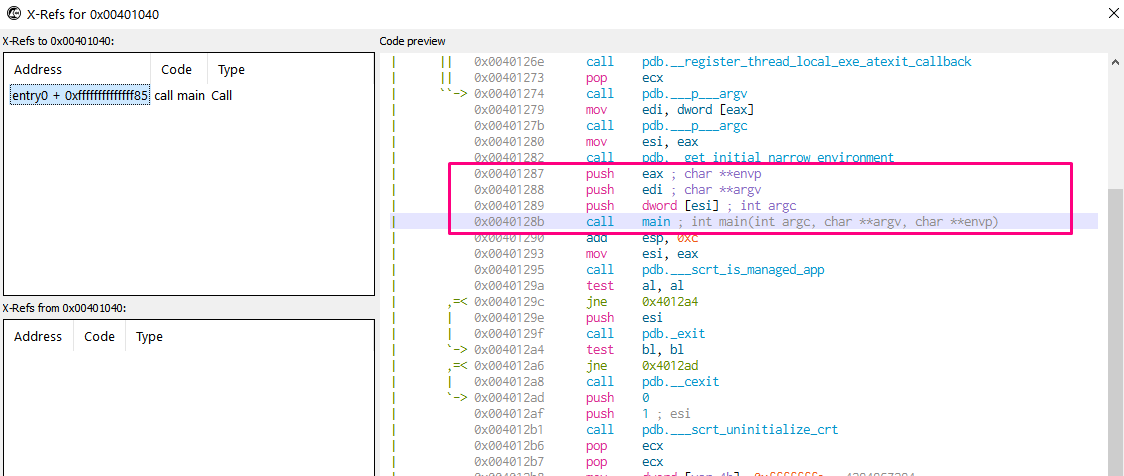


Elijo la que quiero renombrar y lo hago ya quedaron buf y cookie renombradas.

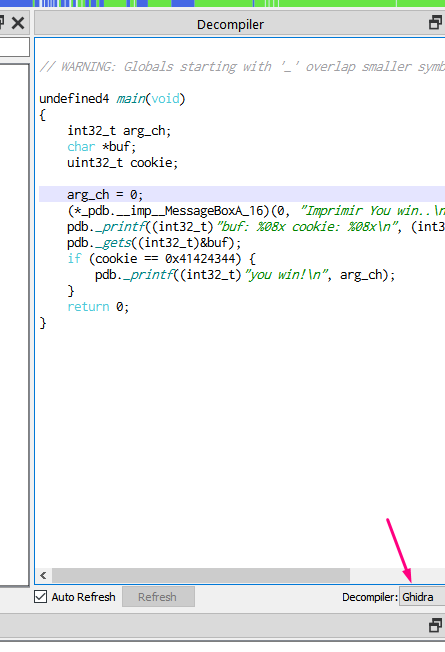


También tiene una ventana para decompilar.

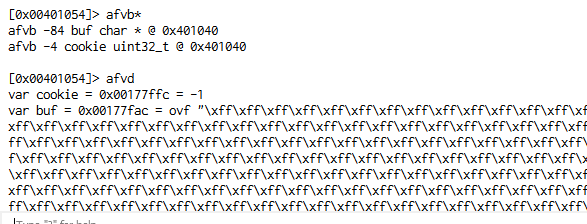
Apretando X nos muestra las referencias a la función.



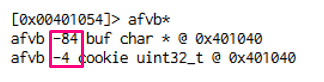
Vemos que entre las opciones del decompiler, esta hacerlo con GHIDRA no se si funciona pero la opción está.

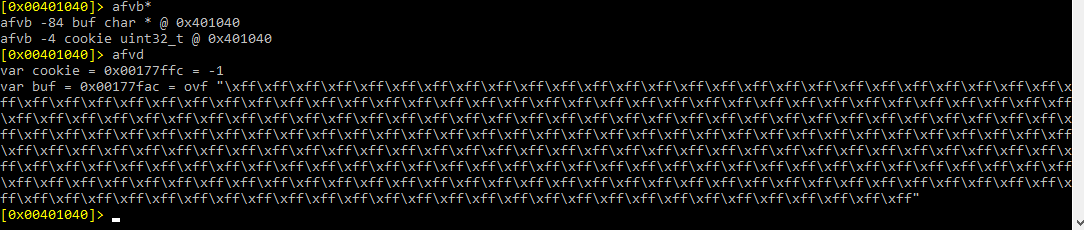


Tenemos la consola donde podemos tipear comandos del radare de consola, en este caso tipeamos **afvb\*** que se usa para listar las variables relativas a **ebp** y **afvd** para ver los valores de las variables cuando estas debuggeando.



Vemos que buf está en -84 y cookie en -4, así que la diferencia entre los dos es 80 con lo cual llenaremos buf y luego los 4 siguientes serán las famosas “DCBA”





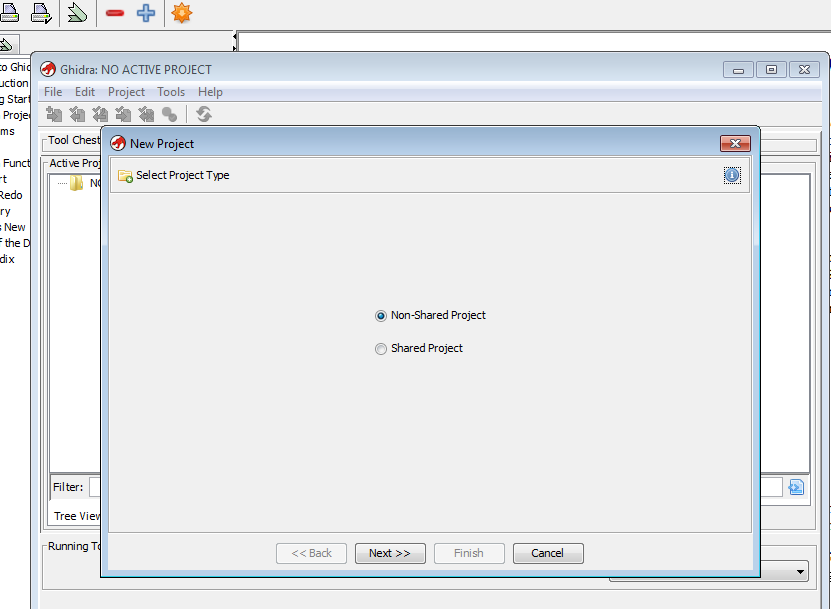
Por lo tanto, vemos que podemos llegar a la misma conclusión que con IDA, vemos que el destination del gets es **buf**, allí copiara la data que ingrese, y vimos que buf mide 80 y abajo esta cookie que lo pisaremos al overflodear con DCBA, que lo compara con 0x41424344 y va a YOU WIN si no son diferentes.

Justo al escribir este tutorial hable con Pancake que es el autor de radare, además de mucha otra gente que ayuda y colabora, es una persona muy accesible y le dije si no podía agregarle un comando parecido al **afbv\*** pero que no solo listara las variables, sino como en IDA, que listara toda la representación estática de la función, para que sea más cómodo ver las distancias y se puso a hacerlo así que posiblemente en los próximos capítulos cuando usemos radare ya tendremos si todo sale bien, esa posibilidad incluida.

3)GHIDRA.

Mientras escribimos la parte 2, salió una nueva versión de GHIDRA la 9.1 asi que lo actualizo para seguir con ella.

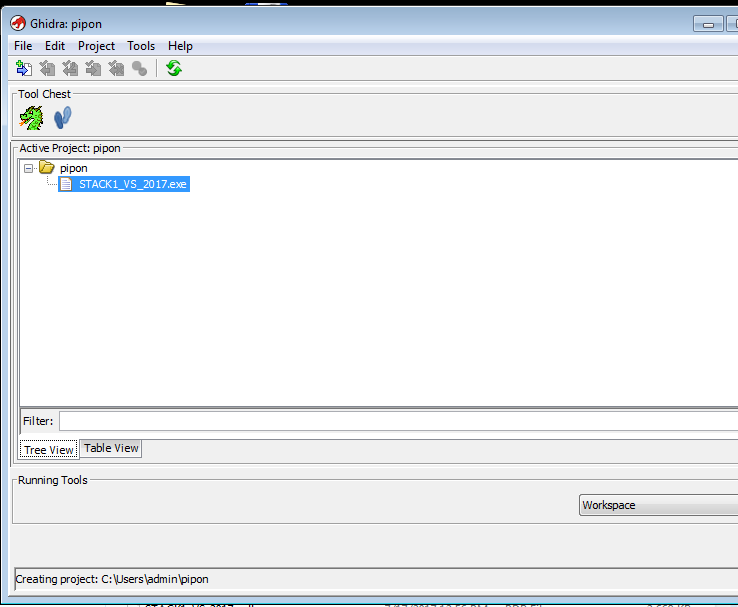
<https://ghidra-sre.org/releaseNotes_9.1_final.html#9_1>



Vamos a File – NEW PROJECT - NON SHARED PROJECT y elegimos NEXT.

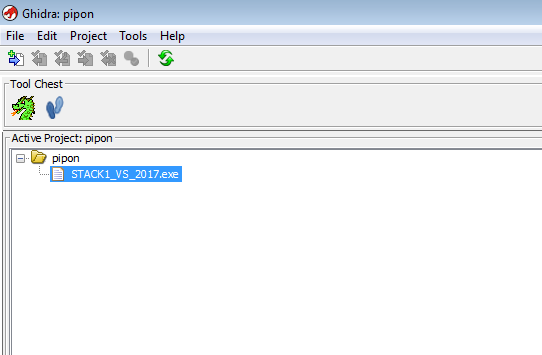
Creo una carpeta para el proyecto y le elijo un nombre.

Ahora arrastro el ejecutable del stack1 en la ventana del proyecto activo y lo suelto allí.



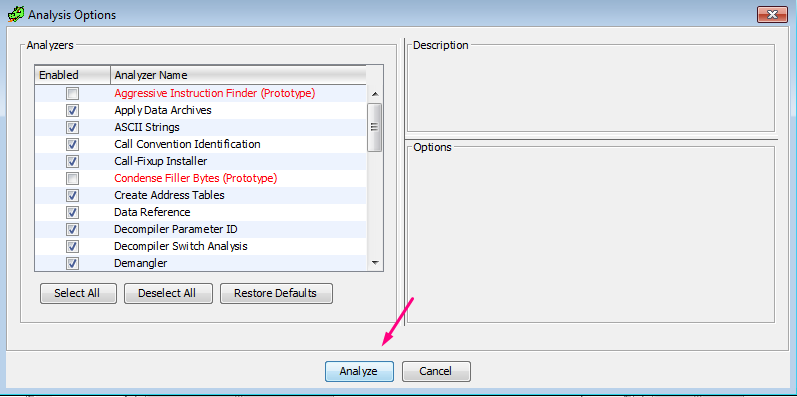
Al soltarlo en esa ventana lo carga.

Nos sale una ventana con la información del archivo que cargo, apretamos OK.



Hacemos doble clic en el nombre de nuestro archivo.

Me pregunta si lo quiero analizar y elijo YES.



El tema es que da un error al tratar de cargar los símbolos.

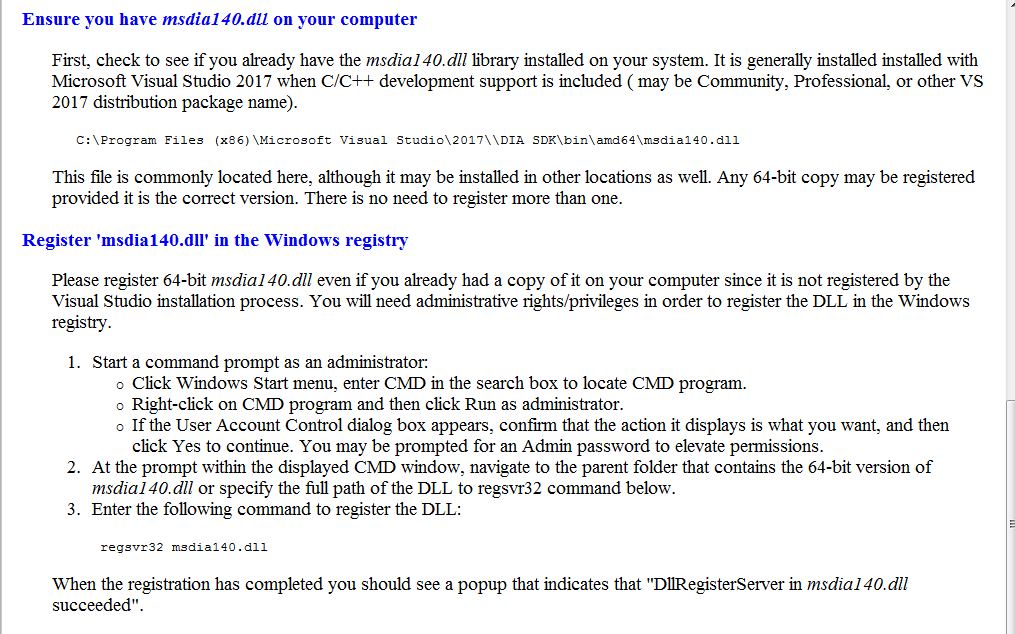
**ERROR: Unable to locate the DIA SDK. It is required to load PDB files.**

**\* See docs/README\_PDB.html for DLL registration instructions.**

**ghidra.app.util.bin.format.pdb.PdbException: ERROR: Unable to locate the DIA SDK. It is required to load PDB files.**

**\* See docs/README\_PDB.html for DLL registration instructions.**

Veamos qué dice ese archivo



Bueno tendré que buscar esto e instalarlo, sale este link para bajarlo sino buscar

[Microsoft Visual C++ Redistributable for Visual Studio 2017](https://go.microsoft.com/fwlink/?LinkId=746572)

o

<https://go.microsoft.com/fwlink/?LinkId=746572>

Igual seguía con error, así que me baje esto

<https://github.com/MalwareTech/MSDIA-x64>

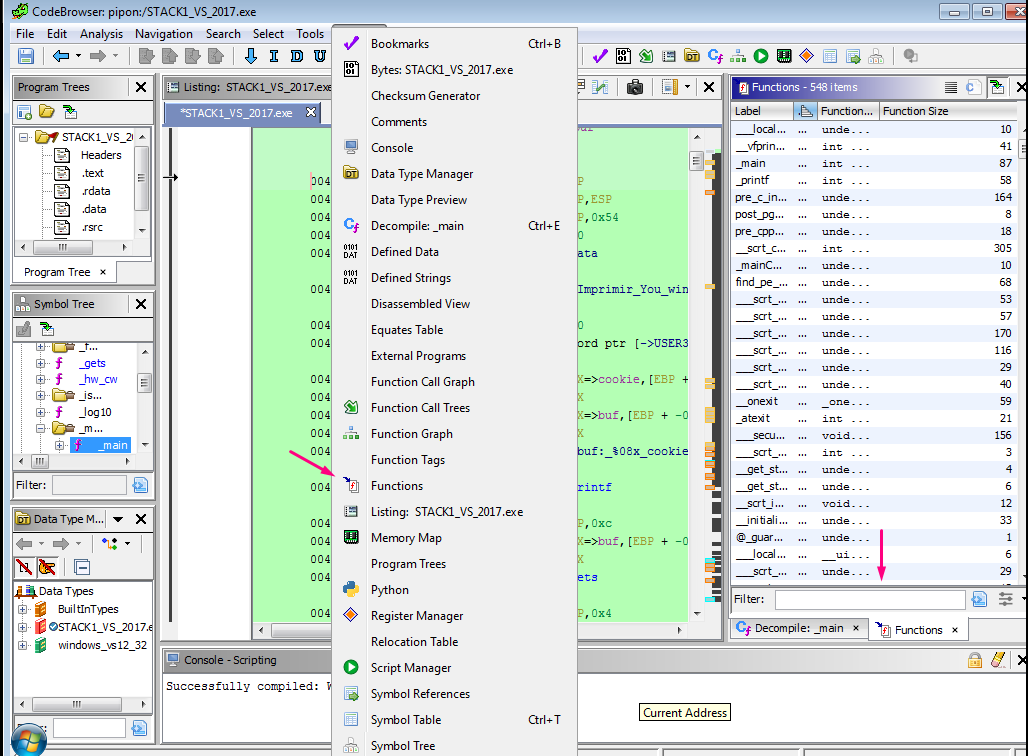
Lo descomprimi y arranqué el .bat desde una consola de administrador y lo copia y lo registra, lo que hace el .bat es esto

**xcopy msdia140.dll %systemroot%\system32**

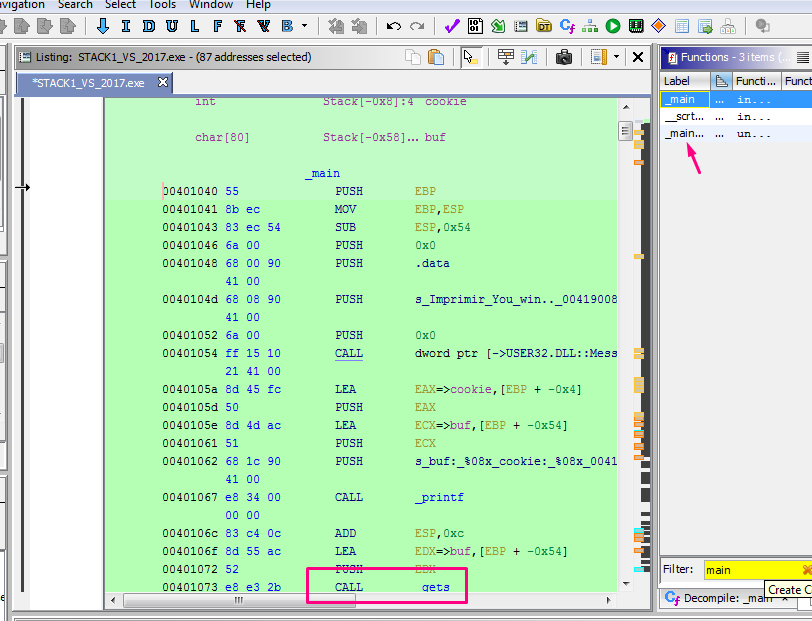
**regsvr32 %systemroot%\system32\msdia140.dll**

Se puede hacer sin ningún bat solo ejecutando estos comandos a mano en una consola de administrador, la cuestión es que con eso funciona, si lo arranco de nuevo y repito el proceso, ya carga los símbolos directamente, sino está en el menú FILE-LOAD PDB FILE para cargarlos.

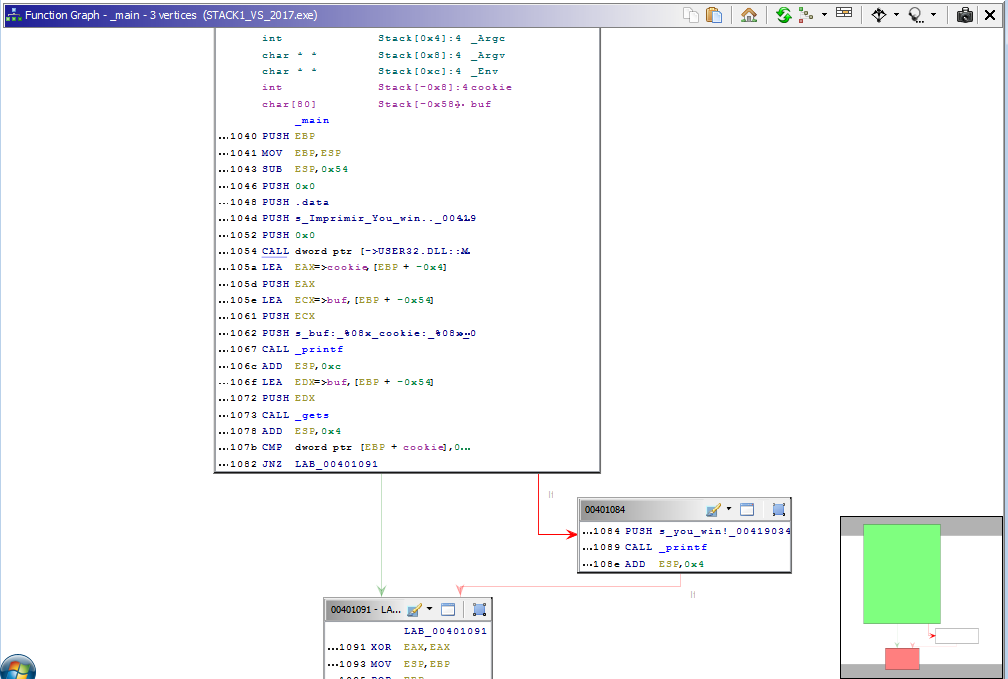
En WINDOW- FUNCTIONS



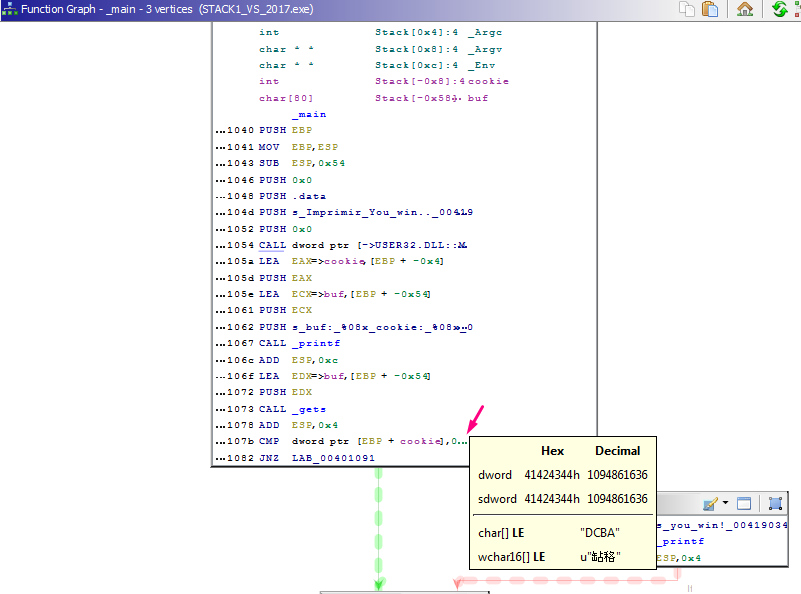
Puedo tipear main y buscarlo, allí veo la función main y se ve el gets, así que tenemos los símbolos bien cargados.



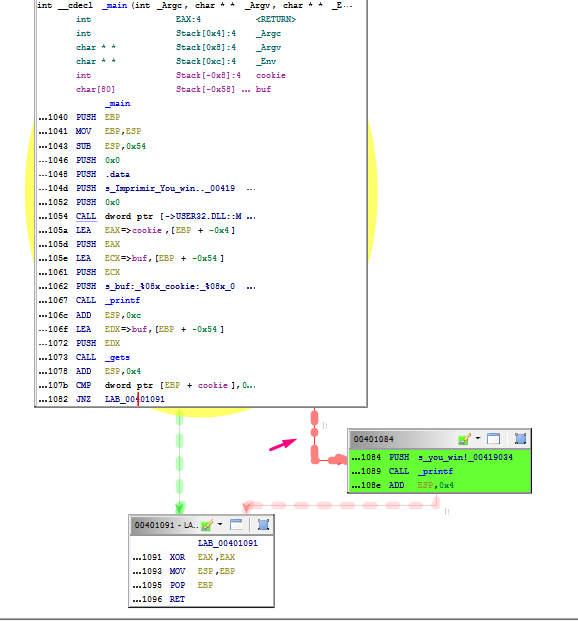
En WINDOW – FUNCTION GRAPH Podemos ver el gráfico con los bloques de la misma.



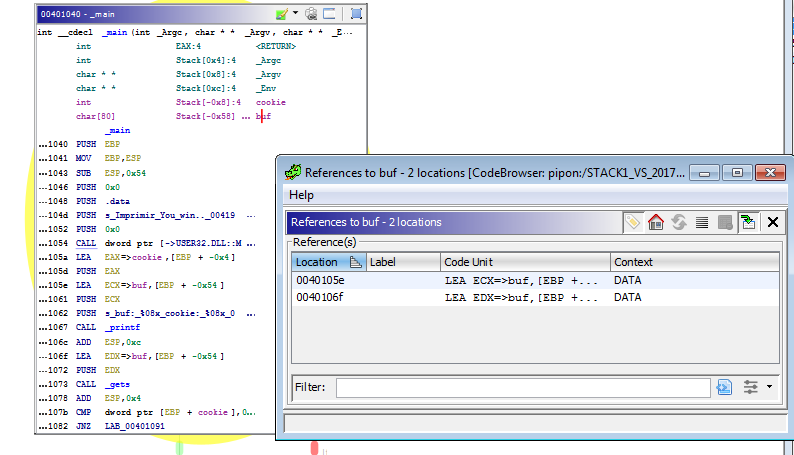
Algunos detalles no se ven por default, pero es interactivo el gráfico y pasando el mouse por encima da detalles de lo que hay debajo.



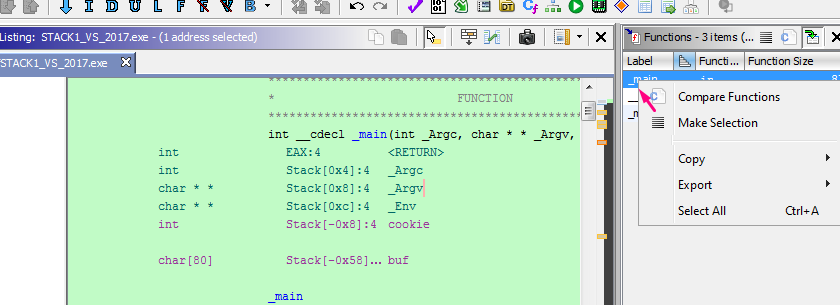
Se pueden pintar bloques, renombrar etc.



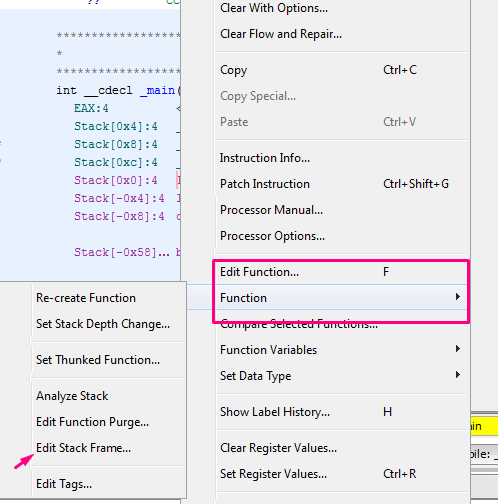
Ver las referencias de una variable o sea donde es usada con clic derecho - REFERENCES.



Si no estaba seleccionada marcamos la función y click derecho MAKE SELECTION.

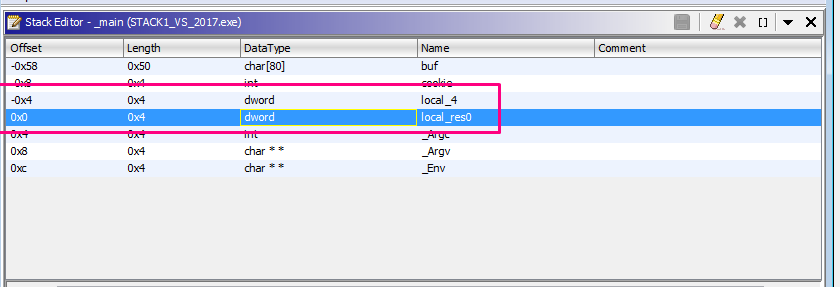


Podemos mirar la representación estática del stack.



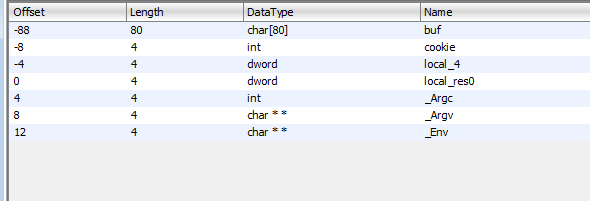
El stored ebp y el return address no figuraban como DWORDS apretando la letra B se va cambiando el tipo hasta que sea un dword.

A diferencia del IDA aquí se muestran



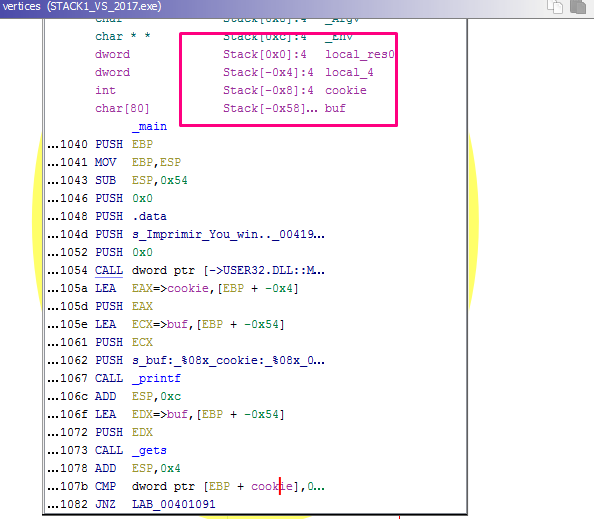
Allí se puede ver en hexadecimal la distancia que hay que llenar de buf para pisar cookie que es 0x58-0x8 =0x50

Se puede cambiar a decimal con el menú del click derecho también.



Allí se ve claro que son 80 aes ya que 88 del offset de buf menos 8 del offset de cookie= 80 y con eso lleno buf, luego 4 bytes más para DCBA y logramos armar el mismo script que con los otros dos desensambladores estáticos.

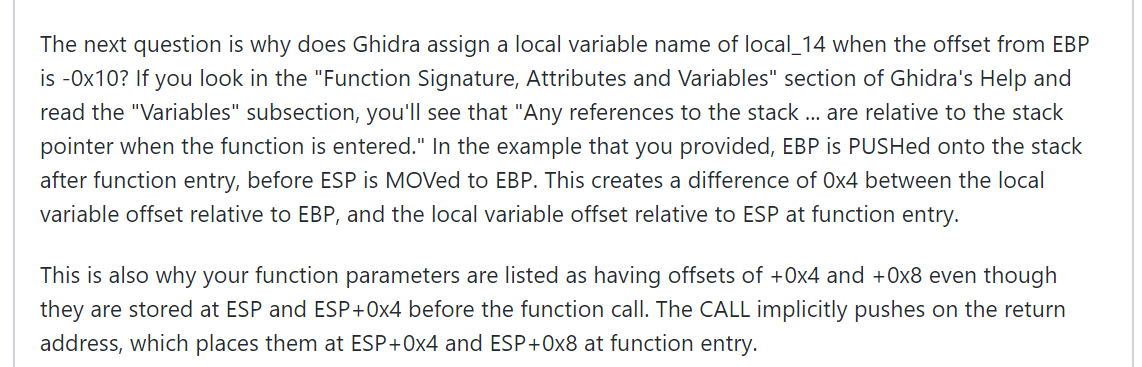
Lo que sí podemos ver como diferente a la representación estática del IDA, es que aquí en vez de tomar como referencia **ebp,** toma como referencia el return address por eso en vez de estar en buf en 0x54, aquí está en 0x58 ya que STORED EBP esta arriba del 0 (RETURN ADDRESS) mientras que en IDA al tomar con respecto a EBP, está debajo del cero.



Vemos en los nombres de las variables buf también como -0x58 mientras que en IDA estaba en -0x54, deberemos tener en cuenta esto al trabajar.

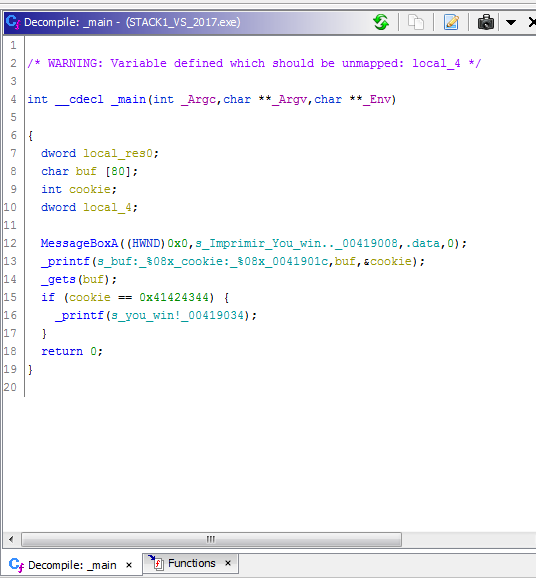
Aquí vemos como ciertas personas preguntan sobre el tema en inglés

<https://github.com/NationalSecurityAgency/ghidra/issues/223>

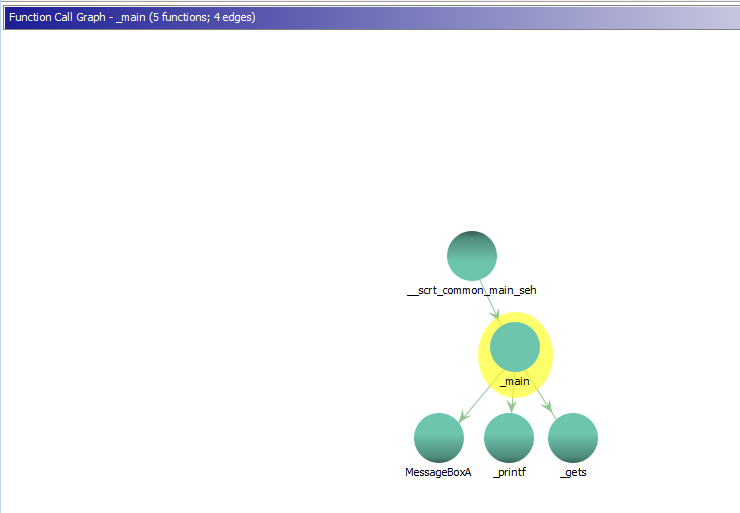


Bueno es un poco confuso teniendo EBP como referencia, trabajar con respecto al return address, pero bueno, ya sabemos para cuando realicemos reversing más complejos a qué atenernos.

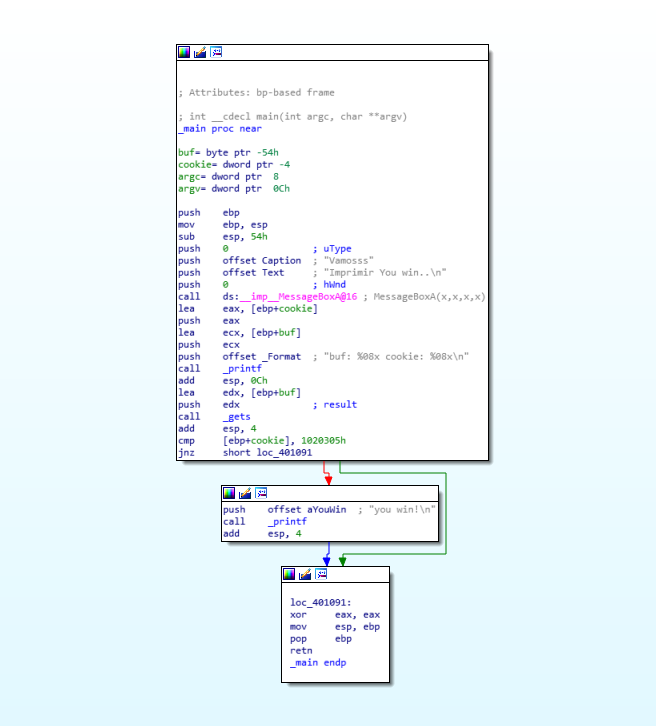
También hay una ventana de decompilación en el menú WINDOW, se ve funcionando bastante bien y es interactiva, cada línea que marcas se muestra en el desensamblado.



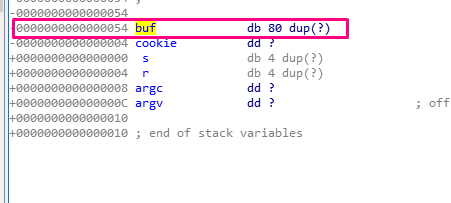
CALL GRAPH



Vamos a mirar el siguiente ejercicio que es el stack2 primero lo veremos en IDA FREE.



Vemos que es similar al stack1, lo que cambia es que compara contra 0x01020305, veamos si los sizes de buf y cookie no cambiaron.



El size de buf es 80 y 4 debajo esta cookie, así que como **buf** es el argumento de **gets**, allí guardará lo que le envío.

Llenando buf con 80 bytes, con 4 bytes más pisamos cookie como antes, así que sería el script.

**import sys**

**from subprocess import Popen, PIPE**

**payload = b"A" \* 80 + b"\x05\x03\x02\x01"**

**p1 = Popen(r"C:\Users\ricardo\xxxxx\abos y stack nuevos\STACK2\_VS\_2017.exe", stdin=PIPE)**

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

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

**p1.communicate(payload)**

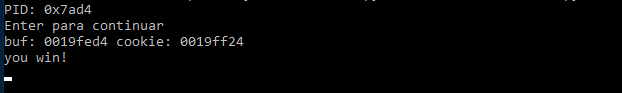
**p1.wait()**

**input()**

El payload es



Ya que por el Little Endian lo que en la memoria se guarda como 05 03 02 01 pasa a leerse al compararse como **0x01020305**.

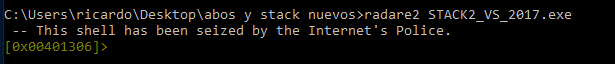


Ahora lo miraremos en radare, lo abrimos usando

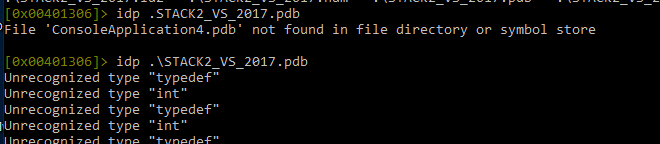
r2 nombre del ejecutable

o

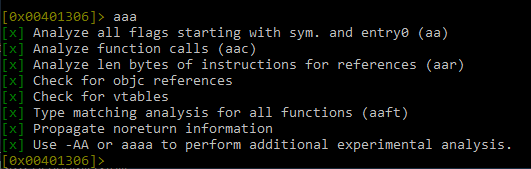
radare2 nombre del ejecutable.



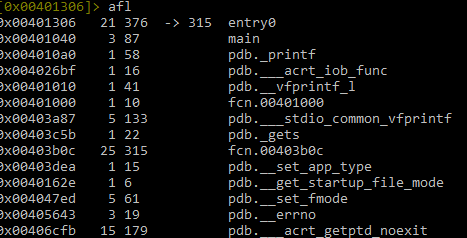
Luego cargó los símbolos, esta vez no me encontro el archivo, cuando le di autocompletar con TAB me lo escribió con .\ adelante, lo puse asi y lo tomó, muy bien.



Luego analizo con **aaa**.



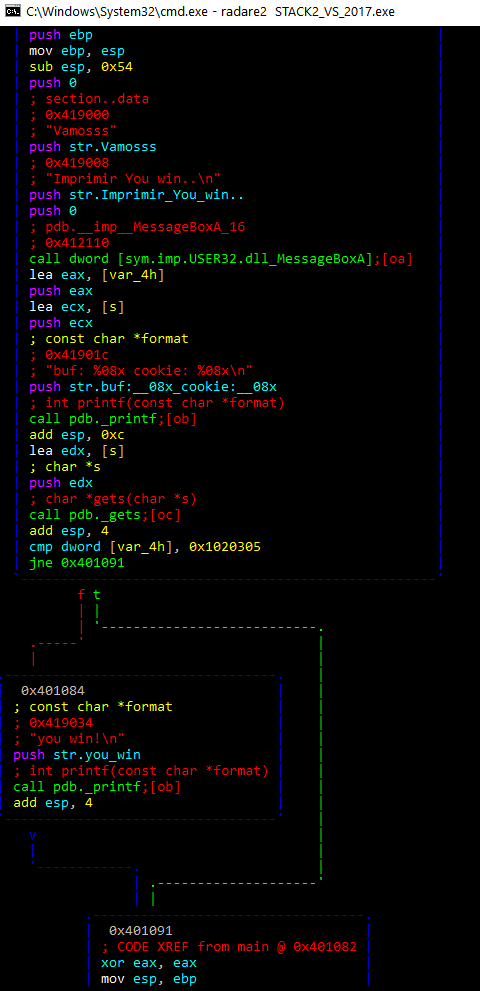
Y luego con **afl.**



Vemos el main antes cambio con **s main** a que muestre allí, luego **eco bright** y **pdf main** para listarla.



Con el comando **agf** podemos ver una representación ASCII visual de la función.



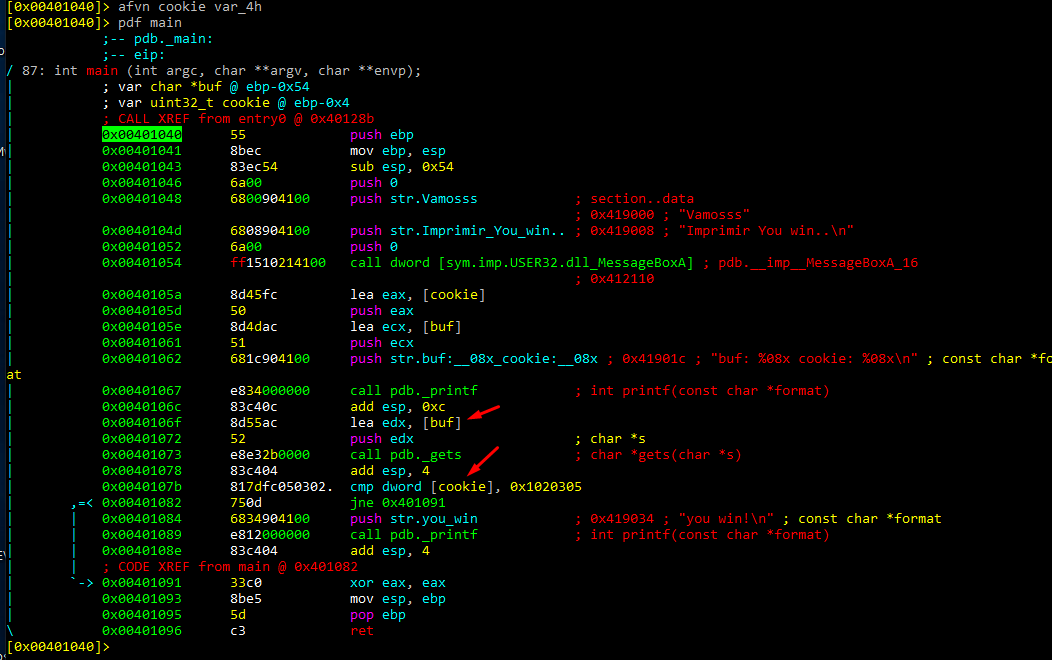
Podemos renombrar las variables

**afvn** nuevo\_nombre viejo\_nombre

afvn cookie **var\_4h**

afvn buf **s**

Ya que en mi desensamblado se llamaban antes **s** y **var\_4h**



Para ver las variables uso **afvb\***

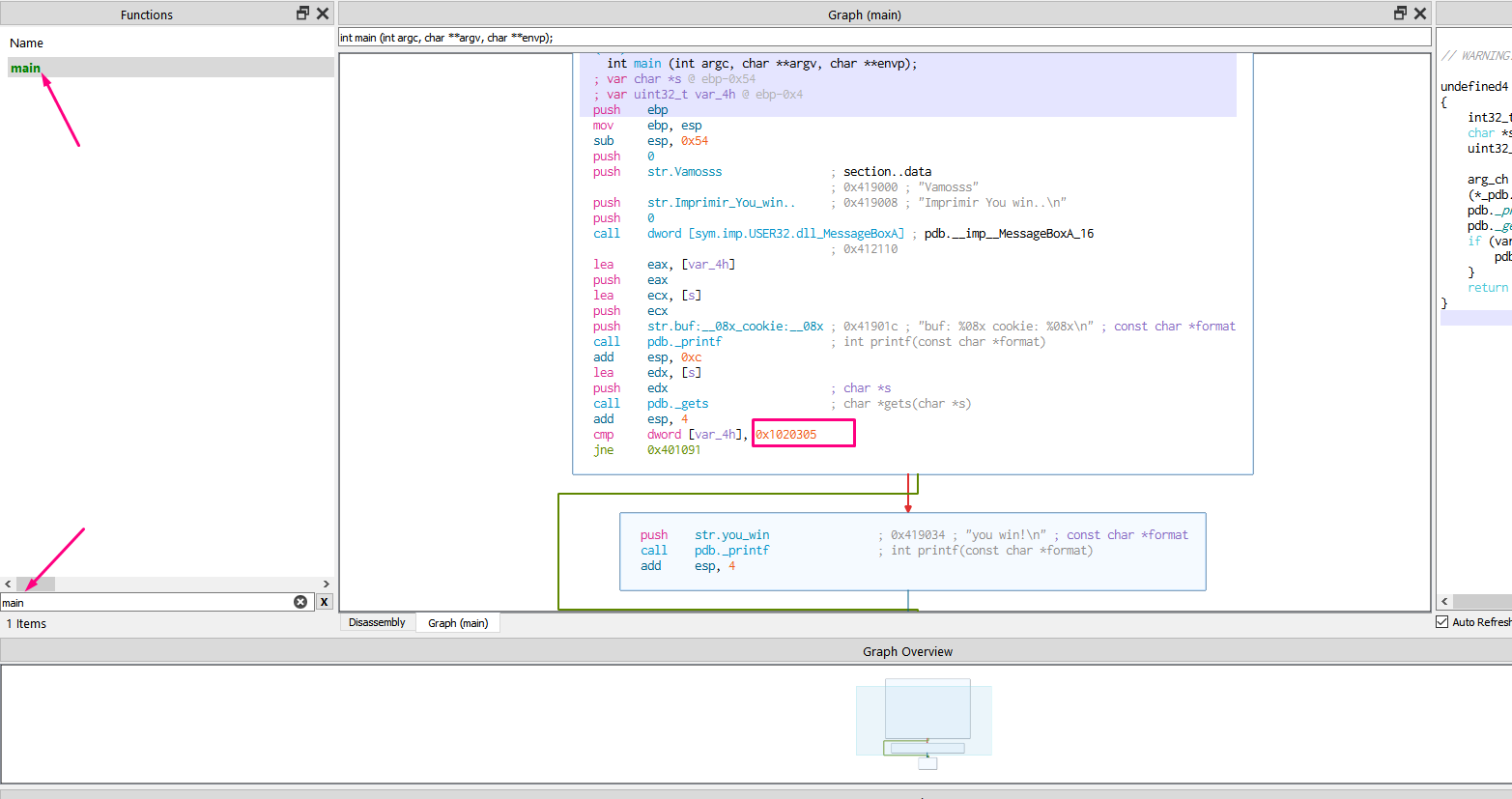




Concuerda con la forma de mostrar del IDA con respecto a EBP.

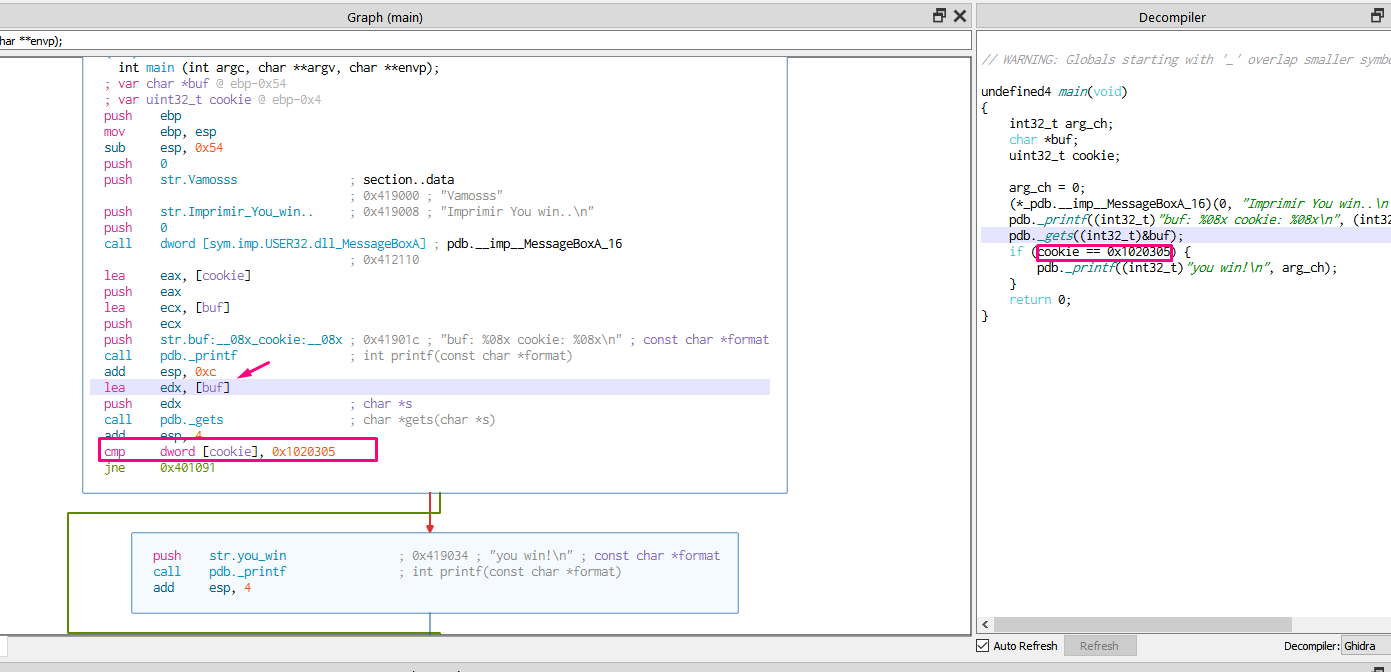
Ya vemos allí la diferencia 84-4 nos da el largo de buf, qué es 80 y 4 más pisamos cookie con "\x05\x03\x02\x01"

Si lo abrimos en Cutter, para ver la opción gráfica.

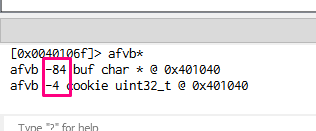


Renombramos las variables.

Vemos en el decompilado los nuevos nombres y que se mantiene todo igual.

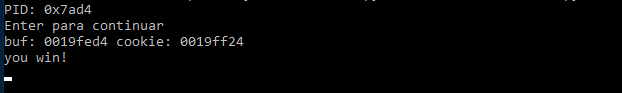


Vemos los sizes con el mismo comando de consola afbv\*.

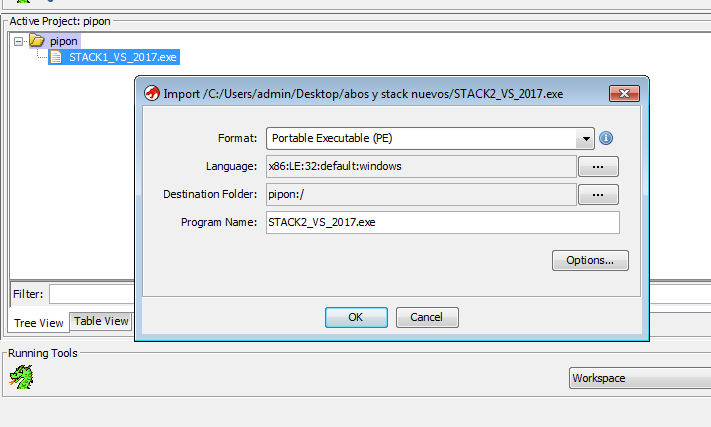


**payload = b"A" \* 80 + b"\x05\x03\x02\x01"**

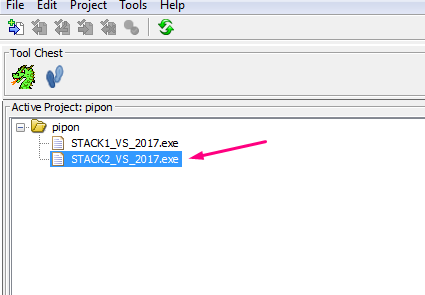
**Ya vimos que el script funciona.**



Ahora con GHIDRA.

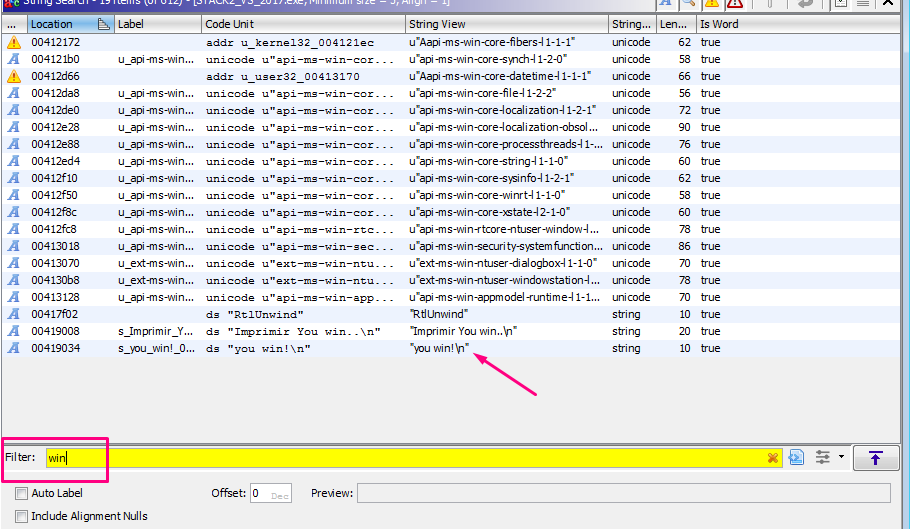


Lo suelto en el mismo proyecto.

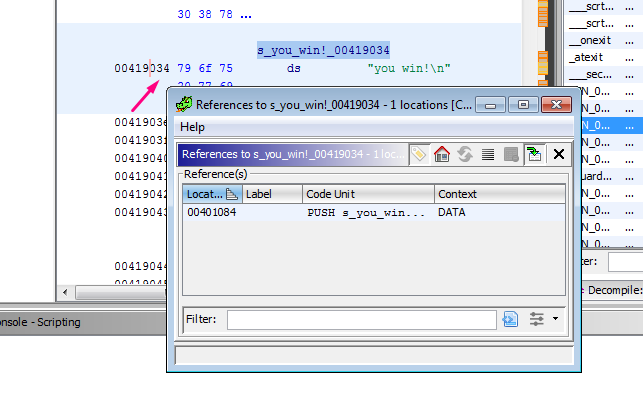


Hago clic allí.

El pdb tenia un pequeño error, en los anteriores desensambladores igual lo cargo aquí no hubo caso, así que lo haremos sin símbolos, miremos las strings.

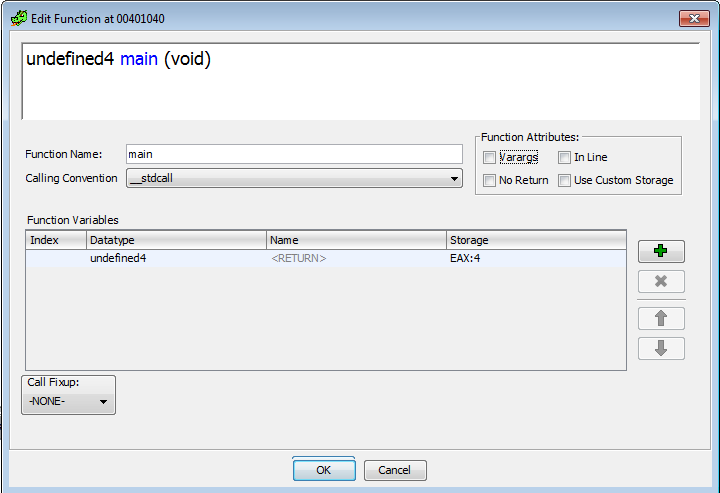


Hago doble click allí.

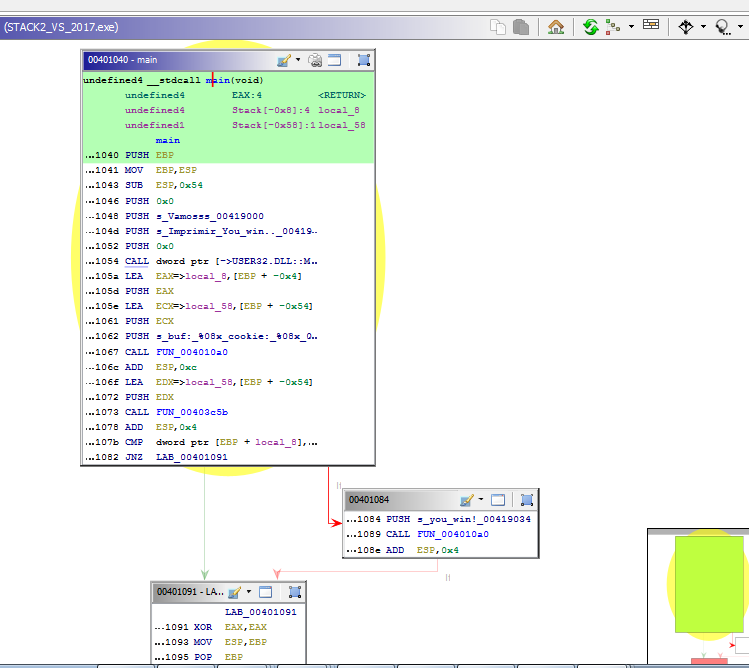


Busco las referencias a la string con click derecho REFERENCES – SHOW REFERENCES TO ADDRESS.

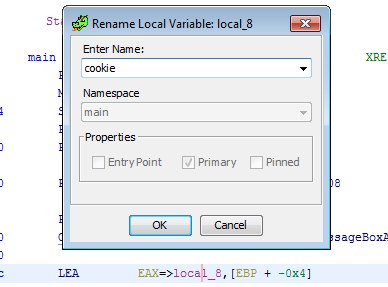
Con click derecho – EDIT FUNCION la renombro a main.

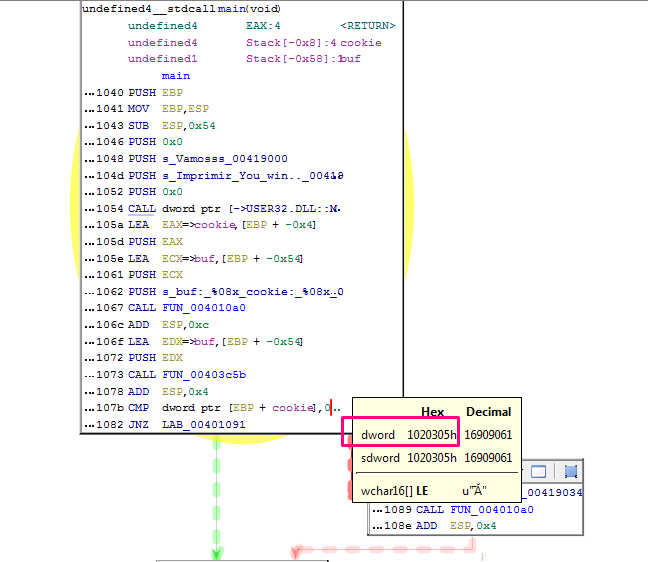


Luego en la lista de funciones busco main hago click derecho- MAKE SELECTION y en WINDOW elijo funcion GRAPH.



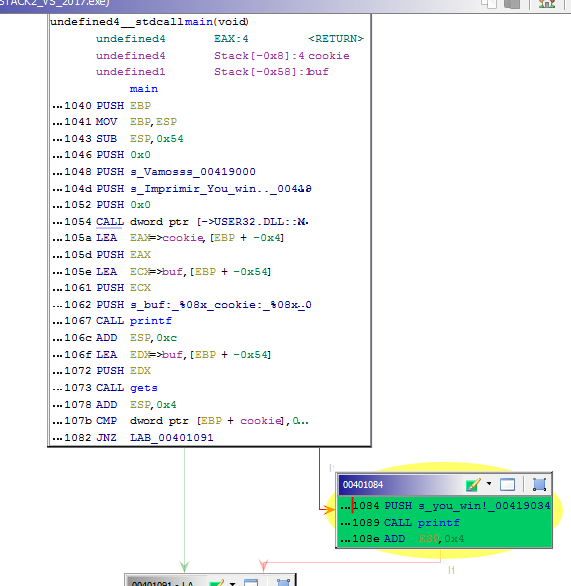
Renombro con click derecho – EDIT LABEL.



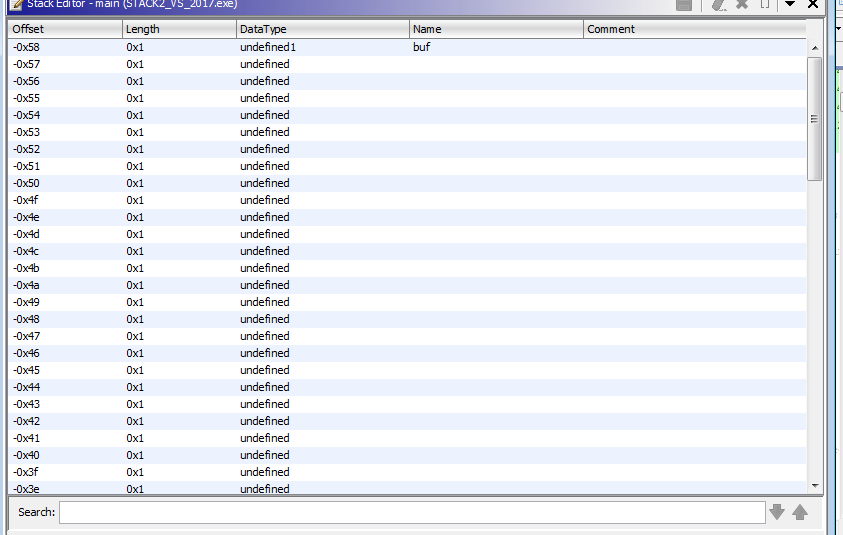


Vemos que compara cookie con 0x01020305, veamos las variables como no tiene símbolos no vemos el gets lo podemos poner a mano renombrando esa funcion 0x403c5b.

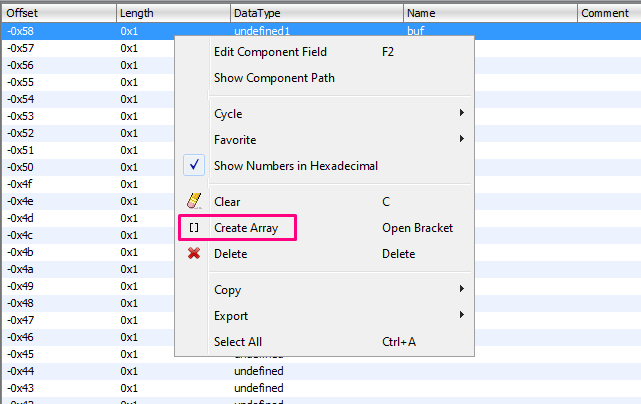
Ahora quedo mas lindo

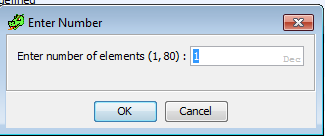


Bueno veamos las variables.

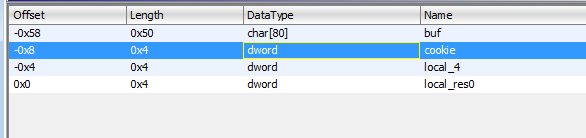


Como no tiene símbolos no sabe el largo de buf, creamos un array allí.

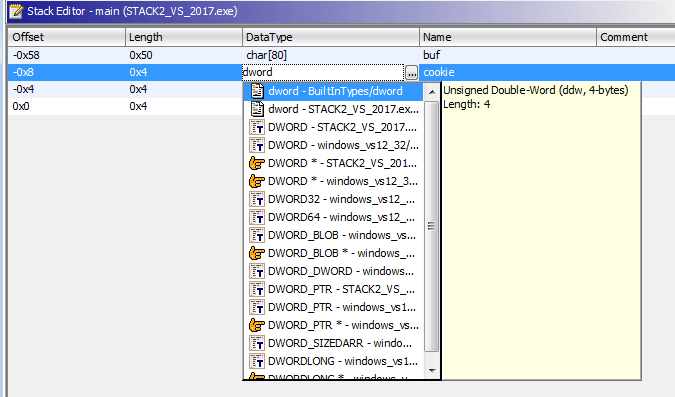




Me dice que el largo puede variar entre 1 y 80, porque se da cuenta de que cookie está justo debajo y ve el espacio vacio hasta alli, asi que elijo el máximo o sea 80.

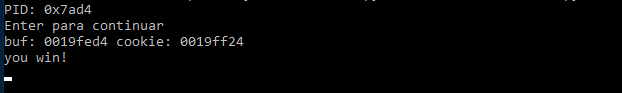


Vemos que puedo ir cambiando los tipos con la letra B, pero a pesar del tamaño correcto, aun queda marcando que no es conocido, editando los datatype a mano, puedo escribir el tipo por ejemplo char[80], en vez de que no este definido, lo mismo los dwords, los puedo cambiar a mano a un tipo conocido que sale de una lista desplegable a medida que voy escribiendo.



Ya veo que el largo de buf es 0x50 ya que es 0x58-0x8 que es 80 decimal y como antes son 80 Aes y luego b"\x05\x03\x02\x01"





Bueno a medida que vayamos haciendo ejemplos más complicados, iremos descubriendo más posibilidades en estas herramientas, terminando los stacks que quedaron pendientes y siguiendo con ejercicios más complejos.

Hasta la parte 4

Ricardo Narvaja

5/11/2019