EXPLOITING Y REVERSING USANDO HERRAMIENTAS GRATUITAS (PARTE 2)

En la primera parte hemos instalado varias herramientas que nos serán útiles para realizar este curso, la particularidad de las mismas es que son todas free, no usaremos ninguna herramienta paga, y de las que hay versión paga como IDA o PYCHARM usaremos la versión FREE o COMMUNITY.

Veamos algunos conceptos antes de empezar a mirar ejercicios.

**Definición de Bug**: es el resultado de un fallo o deficiencia durante el proceso de creación de programas de ordenador o computadora (software). Dicho fallo puede presentarse en cualquiera de las etapas del ciclo de vida del software aunque los más evidentes se dan en la etapa de desarrollo y programación.

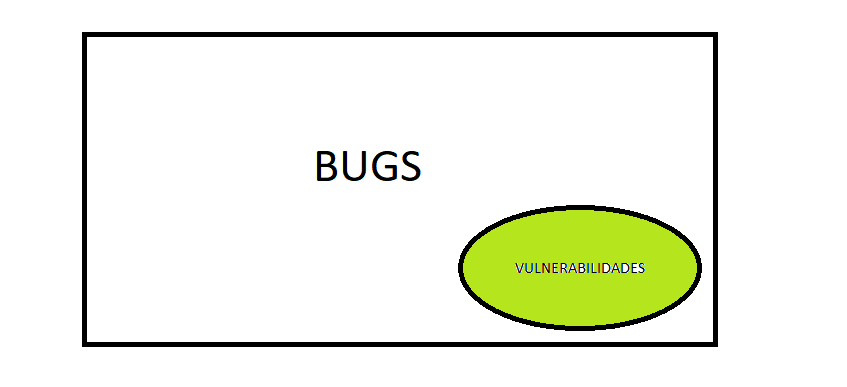
Como siempre digo el programador puede equivocarse y esos errores pueden producir fallos en el programa o bugs, hasta aquí nada nuevo.

El tema es saber la diferencia entre un BUG y una VULNERABILIDAD por lo tanto veamos que es una VULNERABILIDAD.

**Definición de vulnerabilidad :** es un cierto tipo de bug en un programa que permite, mediante su explotación, violar la seguridad de un sistema informático.

Por lo tanto las vulnerabilidades permiten realizar acciones para la cual el programa no fue pensado y abusar de ellas.

O sea que una vulnerabilidad es un cierto tipo de bug, un subconjunto entre los mismos.



Por supuesto hay muchos tipos de vulnerabilidades, nosotros nos vamos a concentrar en el estudio y explotación de las vulnerabilidades en Windows.

**Definición Exploit**:(del inglés to exploit, explotar o aprovechar) es un programa informático que trata de aprovechar alguna vulnerabilidad de otro programa. El fin del Exploit puede ser malicioso como la destrucción o inhabilitación del sistema atacado, aunque normalmente se trata de violar las medidas de seguridad para poder acceder a información del mismo de forma no autorizada y emplearlo en beneficio propio o como origen de otros ataques a terceros.

* Abusar de una vulnerabilidad puede permitir desde crashear una aplicación o el sistema mismo, hasta ejecutar código propio en máquinas locales o remotas, su explotación y dificultad varía dependiendo de la misma vulnerabilidad, del entorno y de las mitigaciones que el target tenga en el momento de la explotación.

El primer tipo de vulnerabilidades que estudiaremos serán los buffer overflow, comenzaremos con los más sencillos ejemplos y luego iremos escalando poco a poco.

Al comienzo no tendremos todas las mitigaciones o protecciones de sistema activadas y poco a poco las iremos activando para aprender cómo podemos manejarnos en esas situaciones.

* **Definición Buffer:**un espacio de memoria de cierto tamaño que se reserva para guardar datos, y manejar los mismos.

Un ejemplo básico es una lata de 20 litros que tengo vacía para guardar allí un contenido, el mismo podrá ser menor o igual a 20 litros, el cual es el tamaño máximo que puedo guardar en este buffer de 20 litros, si quisiera guardar más en un solo depósito debería buscar la forma de tener un buffer más grande, sino, al tratar de guardar por ejemplo 40 litros en una lata de 20 litros se desbordara.

* **Definición Buffer Overflow :**Un [*buffer overflow*](https://www.welivesecurity.com/la-es/tag/buffer-overflow-la-es/) ocurre cuando un programa informático **excede el uso de cantidad de memoria reservado para ello**  , escribiendo en el bloque de memoria contiguo.
* En verdad, un *buffer overflow* se produce en una aplicación informática **cuando no cuenta con los chequeos de seguridad necesarios en su código de programación, como por ejemplo medir la cantidad de datos que se copiara a un buffer y que no exceda el tamaño del mismo.**
* **Los tipos más comunes de buffer overflows son los stack buffer overflows y los heap buffer overflows.**

Bueno aqui vemos la definición de buffer overflow, y en nuestro ejemplo anterior si trato de guardar 40 litros en una lata de 20 litros se desbordara como vimos, ese desborde que se produce es el buffer overflow, o sea el desbordamiento de mi depósito al sobrepasar la máxima capacidad del mismo.

Ahora una idea acerca de la diferencia entre stack y heap

**STACK**

El stack se utiliza para guardar las **variables locales** de una funciónque sólo necesitan durar tanto como la ejecución de la función. En la mayoría de los lenguajes de programación es fundamental que sepamos en tiempo de compilación qué tan grande es una variable si queremos almacenarla en el stack.

**HEAP**

El heap se utiliza para reservar memoria dinámica, cuya vida útil no se sabe muy bien por adelantado, pero se espera que duren un tiempo. Si no sabemos su tamaño o el mismo se decide en tiempo de ejecución se deberá calcular y reservar en el heap.

También se utiliza para objetos que varían en tamaño, porque no sabemos en tiempo de compilación cuánto tiempo van a durar o abarcar.

En nuestra empresa yo trabajo hace más de 13 años como exploit writer, y todos los que ingresan y yo mismo cuando ingrese, lo primero que se hace es intentar solucionar los famosos stacks y abos de Gerardo Richarte, uno de los fundadores de Core Security y uno de los grandes en esto.

Empezaremos paso a paso con los stacks que son los más sencillos, por supuesto como dije están compilados por ahora con mínima protección y en 32 bits, para que sea más sencilla la explotación para iniciarse.

Veamos el código fuente del stack1.

<https://drive.google.com/open?id=16btJAetpa1V5yHDZE2bnnFWTQNpsUR4H>

Allí vemos la carpeta de ejercicios y dentro esta el codigo fuente del stack1 llamado STACK1\_VS\_2017.cpp.

---------------------------------------------------------------------------------------------------------------------

#define \_CRT\_SECURE\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_DEPRECATE

#include <stdlib.h>

#include <stdio.h>

#include "Windows.h"

**int main(int argc, char \*\*argv) {**

MessageBoxA((HWND)-0, (LPCSTR) "Imprimir You win..\n", (LPCSTR)"Vamosss", (UINT)0);

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

**printf("you win!\n");**

}

--------------------------------------------------------------------------------------------------------------------

Vamos a tratar de entender este código y ver donde se puede producir el buffer overflow, y si el mismo será un buffer overflow en el stack o en el heap.

Al código del stack1 original se le ha agregado una llamada a MessageBoxA para que nos muestre un cartelito que nos alienta a resolverlo, es solo una adición que no influye en nada, es un llamado standard a dicha función de Windows que no analizaremos aquí.

El que quiere info aca esta el link de la función MessageBoxA.

https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa

Sabemos que dentro de una función, si existen variables locales hay que reservar espacio para las mismas antes de comenzar con las instrucciones en si.

Así que nos queda esto que es el código original creado por Gera.

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

**printf("you win!\n");**

}

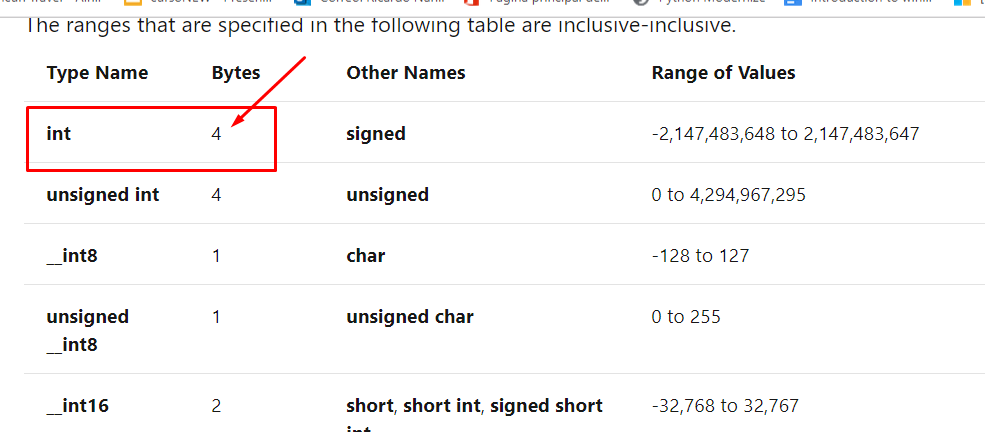
Vemos en rojo la primera parte donde reserva el espacio para las variables locales, en este caso hay dos variables locales, cookie y buf.

Pueden observar en la tabla de tipos de datos, allí están los mismos

<https://docs.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=vs-2019>

El codigo estara compilado en 32 bits.

Vemos que cookie será un int por lo cual se reservarán 4 bytes de memoria para dicha variable.



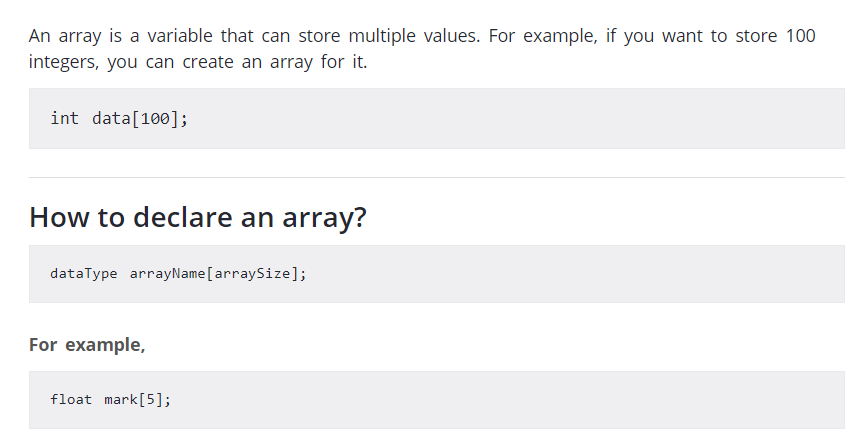
En el caso de buf vemos que es un array o cadena de char (size de char=1)

Será un array con 80 chars o sea su largo será 80x1=80 bytes.

El que no sabe que es un array puede mirar esta data

<https://www.programiz.com/c-programming/c-arrays>

Como resumen un array puede guardar muchos valores del mismo tipo de datos, solo hay que decirle de qué tipo son los datos y cuántos serán.



En el primer ejemplo es un array de int o sea que serian 100 ints y como cada int ocupa 4 bytes el largo seria 100 x 4= 400 bytes

En el segundo ejemplo float ocupa 4 bytes, así que sería un array de 5 floats por lo tanto el largo del mismo sería 5 x 4 = 20 bytes.

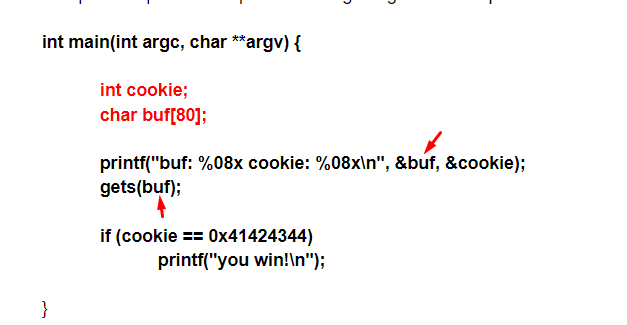
Un array cuando lo analicemos a bajo nivel, veremos que es un espacio de memoria reservado o buffer, no es la única forma de reservar espacio hay otros tipos de variable de datos que también requieren reservar un espacio en la memoria que serán buffers para guardar su contenido.

Volviendo al nuestro ejercicio

**char buf[80];**

Es un array de chars de 80 x 1= 80 bytes de largo, o sea que es como nuestra lata de 20 litros, si tratamos de guardar más de 80 bytes desbordara.

Ahora veamos donde se usa buf.



Vemos que se utiliza en dos lugares marcados con las flechas rojas.

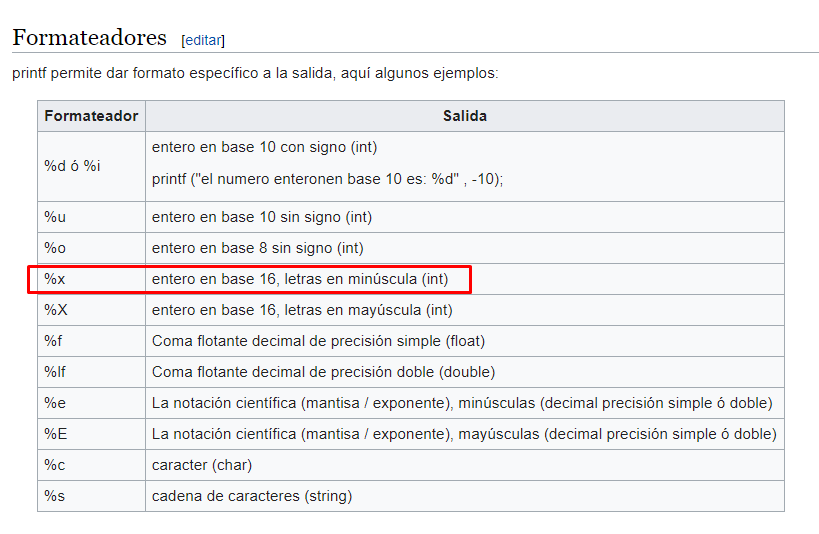
En la primera instrucción hay un **printf** que se utiliza para mostrar en la consola un mensaje el cual sería la string entre comillas

**"buf: %08x cookie: %08x\n"**

Pero printf no solo imprime la string entre comillas sino que imprime con formato, los porcentajes que hay dentro nos dicen que allí armara una string de salida, vemos que la string es solo el primer argumento de la función y es el formato de la salida y los otros argumentos pueden ser varios (habrá uno por cada % que haya en formato) en este caso dos.



En este caso son dos %x por lo tanto si consulto la tabla de formatos de printf.



Vemos que tomara esos enteros (int) y los insertará en la string de salida con base 16 o sea en hexadecimal, el 08 se refiere a que si el numero tiene menos de 8 cifras lo rellenará con espacios.

salida para **"buf: %31x”,&buf**

**buf: 19fed4**

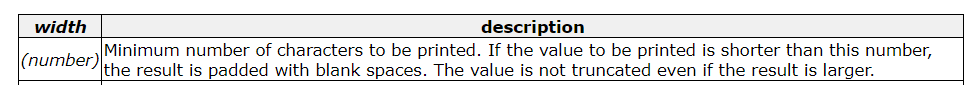
Vemos que en este ejemplo rellena con espacios delante del numero, hay varios modificadores de la forma de mostrar la salida.

Todas las posibilidades están aquí

[**http://www.cplusplus.com/reference/cstdio/printf/**](http://www.cplusplus.com/reference/cstdio/printf/)

Nuestro caso era este

****

****

Vemos que el resultado no es truncado solo rellena con espacios, si el largo del argumento a insertar es menor que el valor delante de la x.

Por lo tanto sabemos que imprime dos números en hexadecimal que provienen de los dos argumentos.

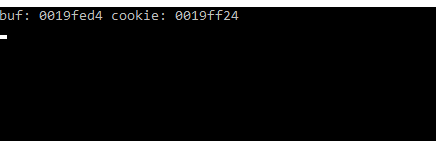
**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

Sabemos que una variable tiene una dirección de memoria y un valor que puede guardar, es como nuestra lata de 20 litros, la misma tiene su contenido o valor, que son los litros que guarda dentro, pero también si tengo un deposito lleno de latas similares, tengo que tener alguna forma de identificar donde está ubicada la lata que quiero, entre todas las que poseo.

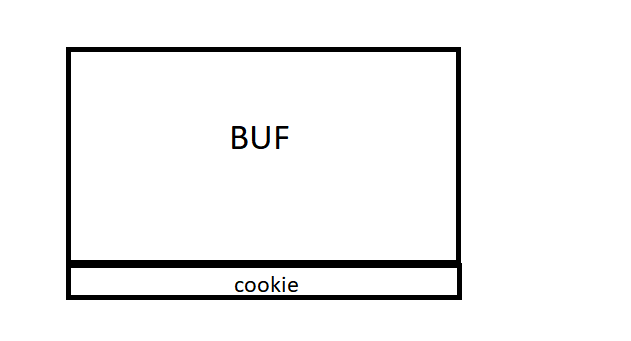
Esto se indica con el símbolo **& ampersand**, nos indica la dirección o ubicación de la lata, no su contenido o valor.

Definición de ampersand : & ampersand se usa para indicar una dirección de memoria de la variable donde se almacenará el dato.

Por lo tanto si corro el ejecutable en una consola veré por ejemplo que al pasar por el printf imprime.

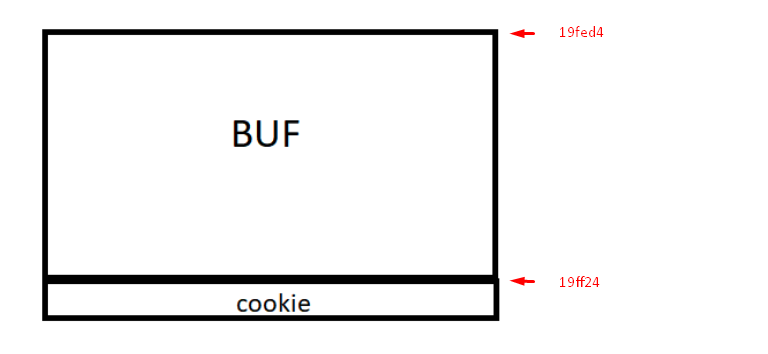


En sus máquinas las direcciones pueden cambiar pero como la dirección más baja de ambas es la de buf podemos ver que se ubican en esta forma.



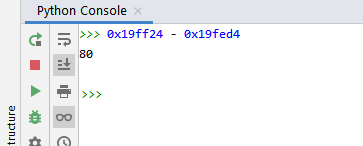
La dirección de buf es menor que la dirección de cookie así que irá arriba.

Y que nos dicen las direcciones de ambas? ( en mi caso eran &buf=0x19fed4 y &cookie= 0x19ff24)



Ambas están representados en hexadecimal recordemos que era %x el formato, por lo tanto le coloco el 0x delante para diferenciar de los números decimales que los representaremos sin ningún agregado.

Si en una consola de python o en el mismo Pycharm en su consola hago la resta



Nos da 80 el tamaño de buf, ya que supuestamente cookie empieza justo donde termina buf, así que la diferencia nos da el tamaño de buf.

Muchas veces cuando hacemos este tipo de cuentas basadas en el código fuente, puede ocurrir que nos dé más grande que lo reservado en el código original y eso es porque el compilador asegura que reservará al menos 80 bytes, pero puede reservar más, nunca menos.

La cuestión que sabemos algunas cosas ya sobre el código los tamaños de las variables y su ubicación gracias al printf que posee.

Ahora veamos el otro lugar donde se utiliza la variable buf ya que por ahora solo imprime su dirección, pero no ha guardado nada en ella.

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

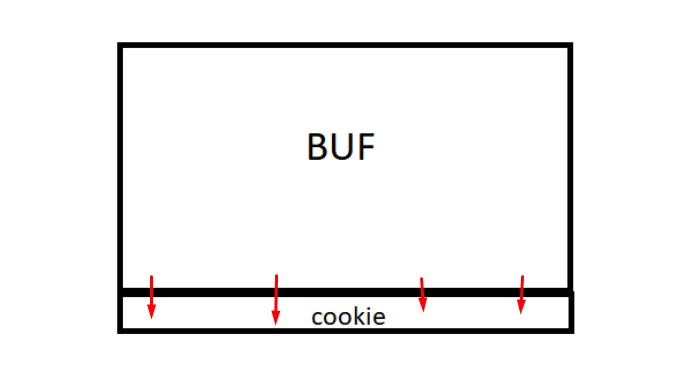
**printf("you win!\n");**

}

Allí en la instrucción en rojo, gets es una función para ingresar datos por teclado, que ingresara la cantidad que el que está tipeando desee, cuando aprete la tecla enter.

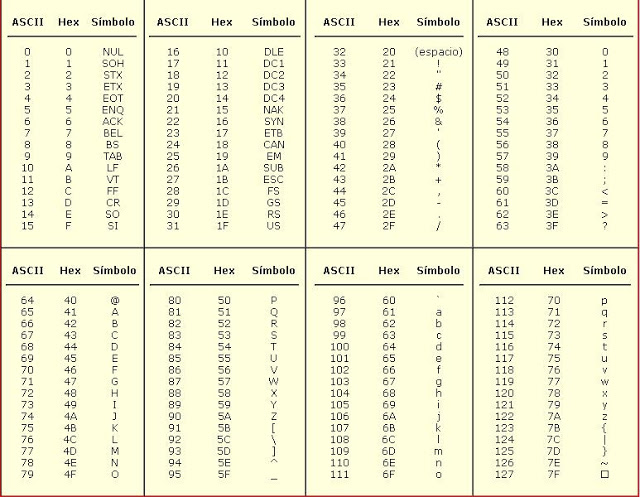
No hay forma que el programa limite la cantidad de datos que ingresa el usuario, ni forma de chequear los datos, todo lo que se ingresa hasta apretar ENTER se copia al buffer buf.

Esto tiene un problema habíamos dicho que buf solo puede guardar 80 bytes de máximo, si ingresamos más produciremos un buffer overflow, y aquí están dadas todas las condiciones para ello ya que si el que ingresa escribe más de 80 bytes, desbordara nuestra lata y comienza a chorrear líquido hacia abajo.:-)



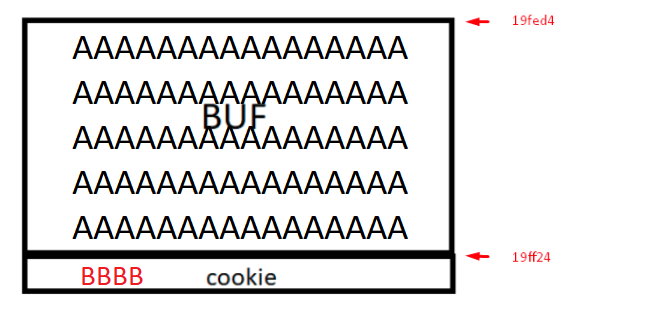
El tema que debajo de buf esta la variable cookie, así que lo que desborde pisara y llenará la misma con un valor que el que tipea podrá controlar.

Por ejemplo si el que tipea escribe 80 Aes y 4 Bs las 80 Aes llenaran buf y las 4 Bs llenaran cookie, y como sabemos cuando uno tipea un carácter en la consola, a bajo nivel se guardará el valor ASCII del mismo.



Como cookie se habrá llenado con cuatro letras B que equivalen al valor 0x42, podremos asegurar que el valor de cookie será 0x42424242 o sea en mi maquina la dirección de cookie 0x19ff24 tendrá como contenido 0x42424242.

0x19ff24 **42424242**



La cuestion es que ya vimos como desbordar y controlar el valor de cookie

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

**printf("you win!\n");**

}

El tema es que para vencer al ejercicio hay que imprimir you win, para lograr eso cookie debe ser igual a 0x41424344 y eso si no hubiera overflow no sería posible ya que el valor de cookie no es cambiado nunca desde el inicio del programa, mientras que no haya overflow, no podremos imprimir you win y acá se complementa con la definición de buffer overflow que decía que el mismo puede hacer que el programa realice alguna acción diferente a la cual estaba programado.

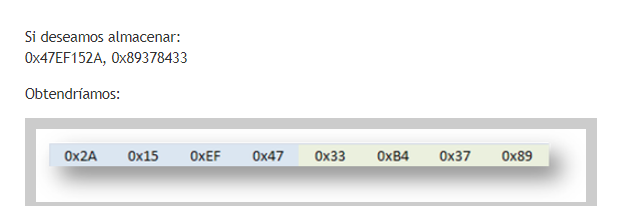
En este caso jamás se podría imprimir you win, solo el overflow permitirá hacerlo.

O sea que en vez de pasar por ejemplo 80 aes y 4 B para que imprima you win debería pasarle 80 Aes y luego las letras DCBA ya que eso hará que en cookie se guarden los valores ascII

44434241

<https://www.arumeinformatica.es/blog/los-formatos-big-endian-y-little-endian/>

El formato en el que se almacenan los datos, es little endian o sea que en la memoria se guardan al revés jeje por decirlo en una forma sencilla.

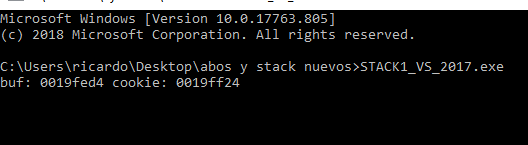


Y si se almacena 0x41424344 lo guardará en la memoria como

44 43 42 41

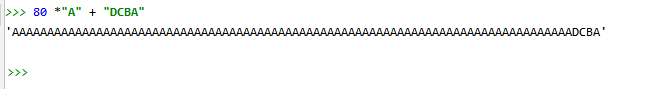
Por eso como al copiar en la memoria lo hará tal cual tipeamos, deberemos escribir el valor al reves para que al leerlo desde la memoria lo haga en forma correcta.

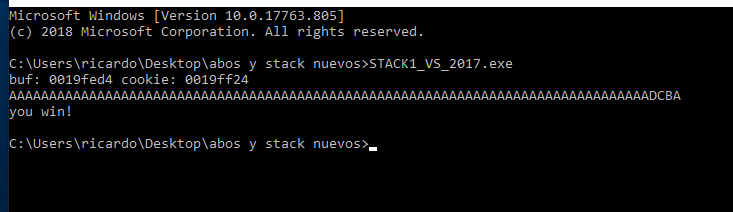
Podemos correrlo al ejecutable en una consola.



Y queda titilando el cursor ya que el gets me pide que tipee los datos de entrada, tipeare cuidadosamente 80 Aes y luego DCBA.

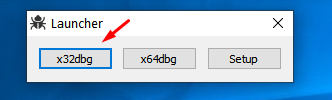
Puedo en una consola de Python o Pycharm imprimir la string y copiarla sin las comillas y pegarla en la consola para no tipear como loco, luego apretar ENTER para que ingrese.



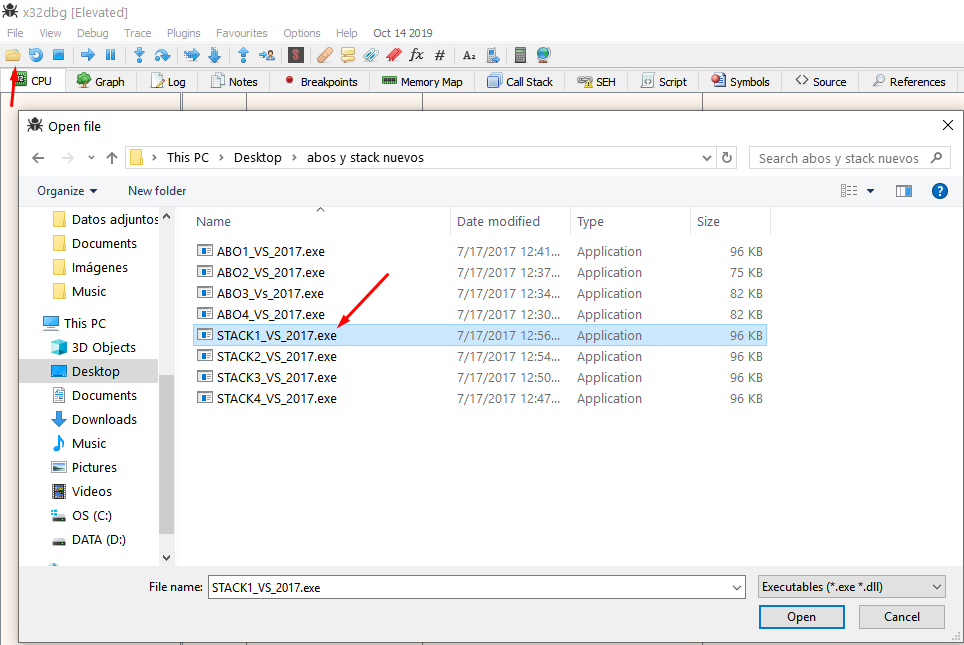


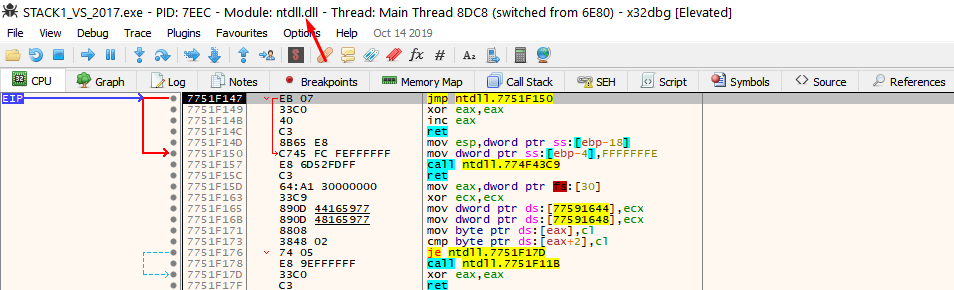
Vemos que lo logramos YOU WIN jejeje.

Esto podemos verlo en un debugger, para ello utilizaremos X64DBG.



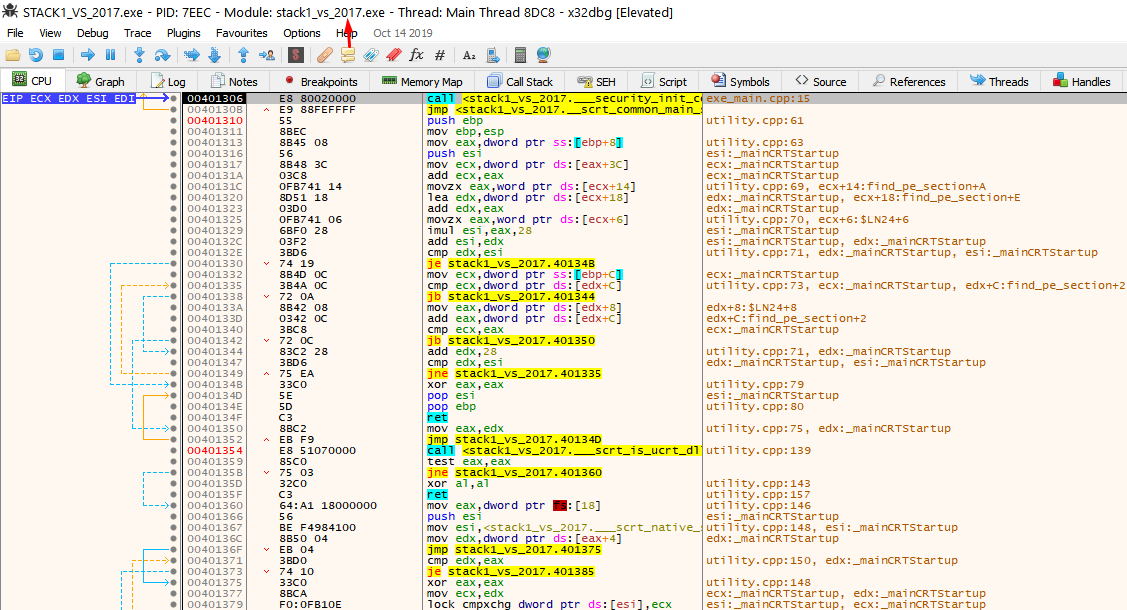
Elijo la versión de 32 bits.



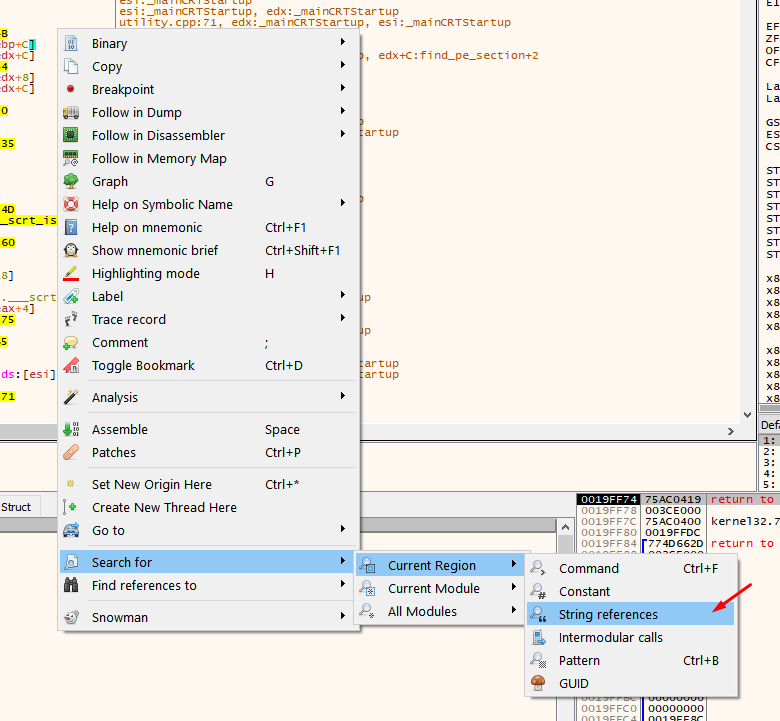


Si nos para en ntdll damos RUN nuevamente con F9.

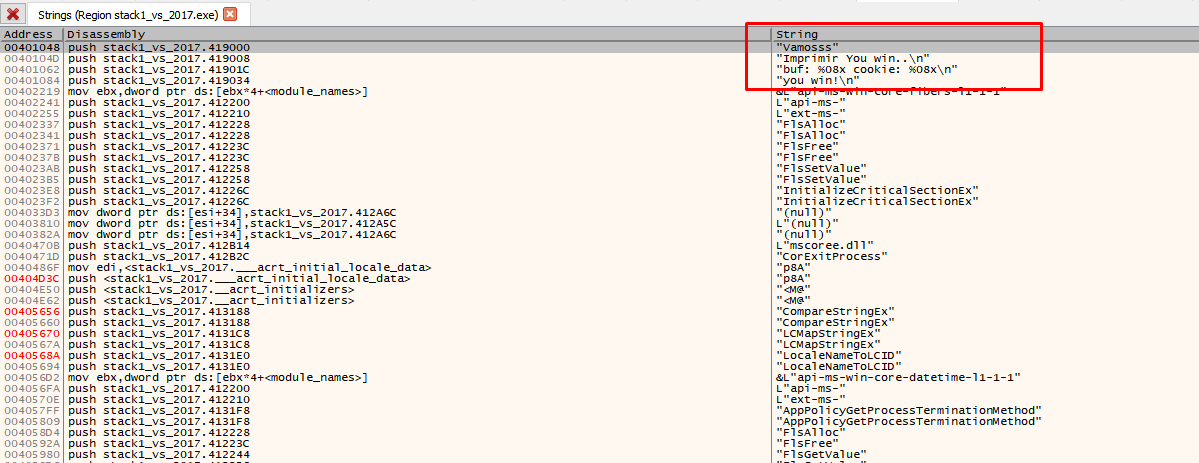
Vemos que allí para en la primera instrucción del módulo stack1, lo que se conoce como ENTRY POINT o la primera instrucción ejecutada por un módulo.



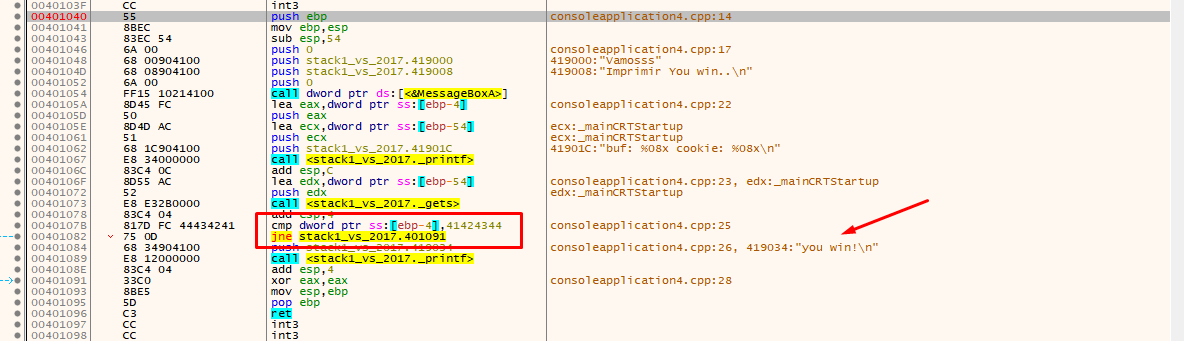
Obviamente no se parece en nada al código fuente, hay que comprender que el compilador le agrega mucho código para que funcione el ejecutable y se inicie correctamente, trataremos de buscar nuestra función main, nos podremos orientar mirando las strings del programa.



Elegimos solo que busque en la región actual, sabemos que estarán en esta misma sección.

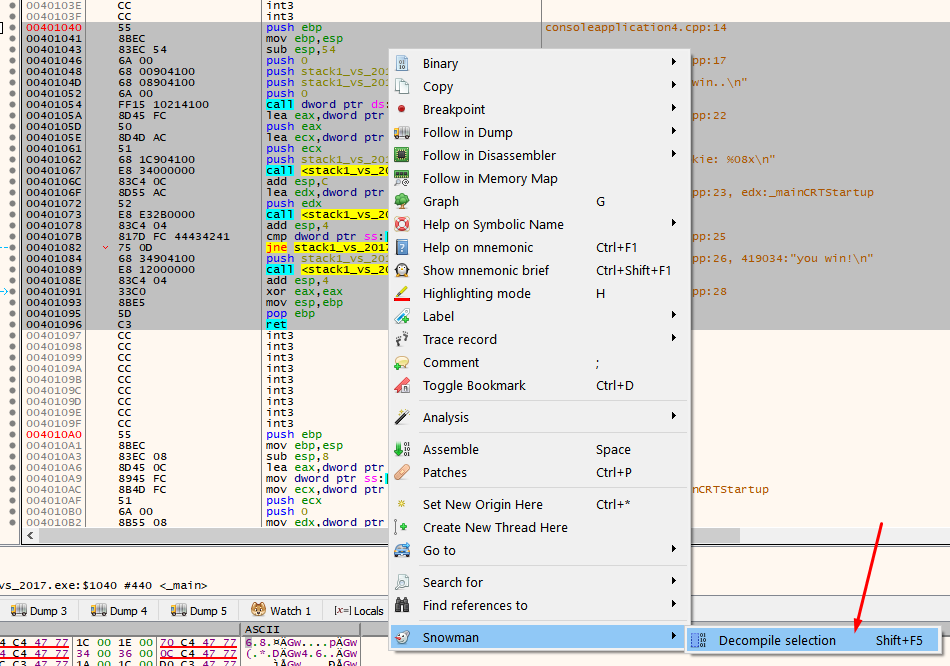


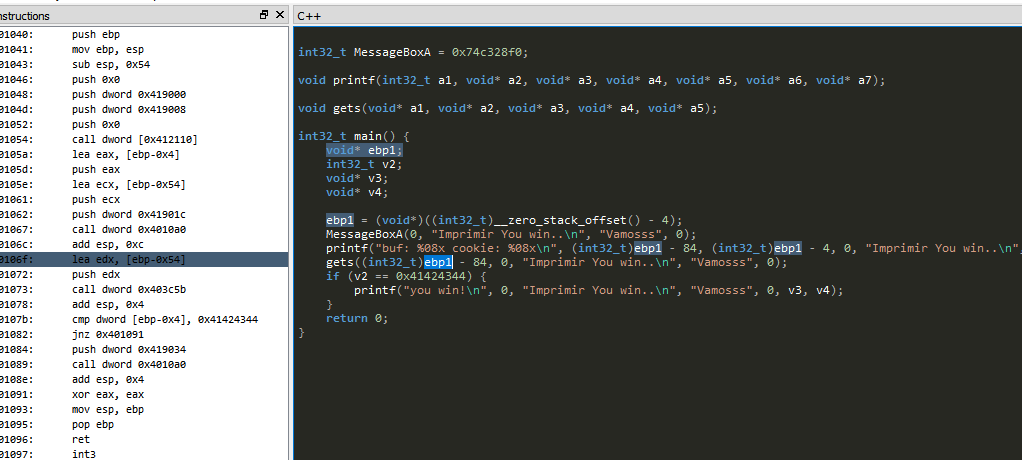
Allí vemos que están las strings del programa, dentro de otras que agregó el compilador, hagamos doble click en alguna de las nuestras.



Ahora tiene más sentido vemos el llamado a MessageBoxA, el printf, el gets y la comparacion con 0x41424344.

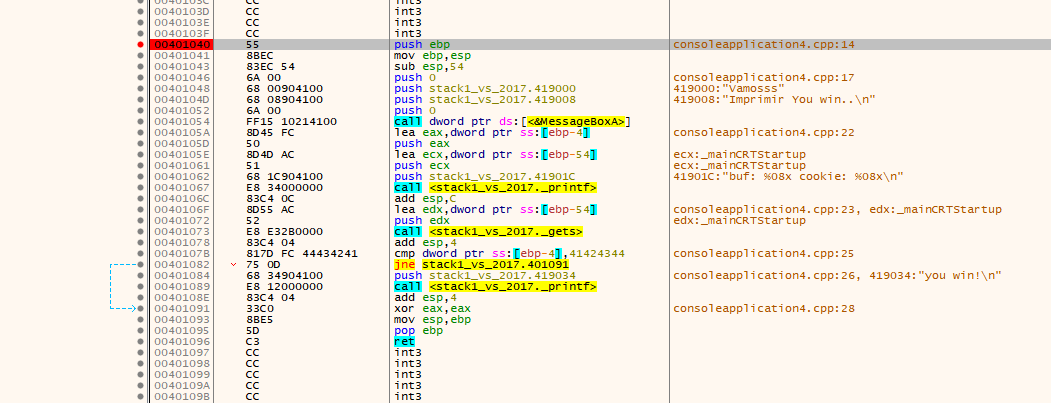
También como le agregamos el plugin para decompilar SNOWMAN podemos tratar de ver como lo descompila o sea como trata de obtener el código fuente o algo lo más parecido posible desde el archivo compilado.



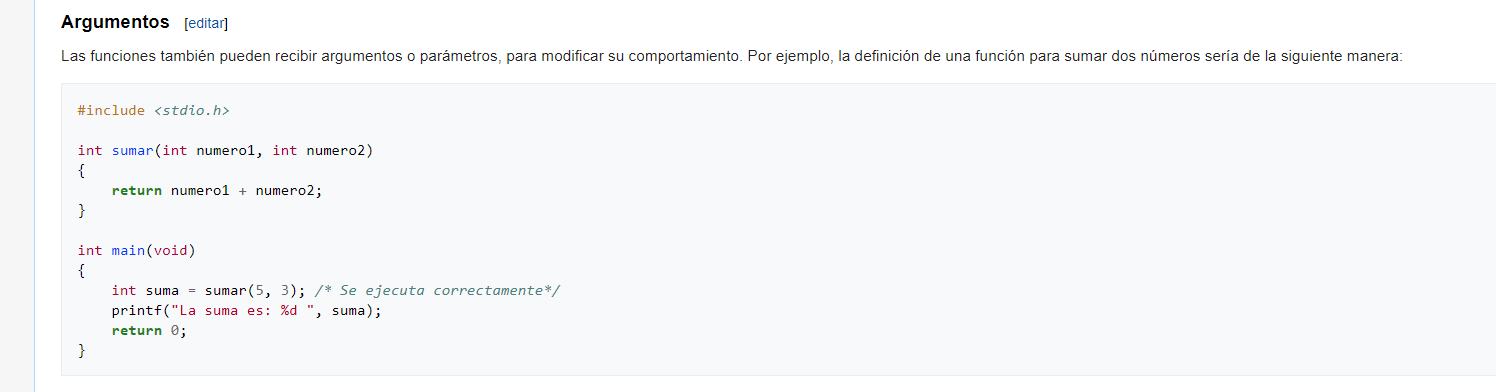


Vemos que no es perfecto, pero bueno ahí está lo mejor que puede.

Le voy a poner un breakpoint en el inicio de la función y apretar F9 hasta que pare allí.



Para los que no saben lo que es un argumento de una función.



En nuestro caso la función main tiene argumentos pero no son usados dentro de la función.

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

**printf("you win!\n");**

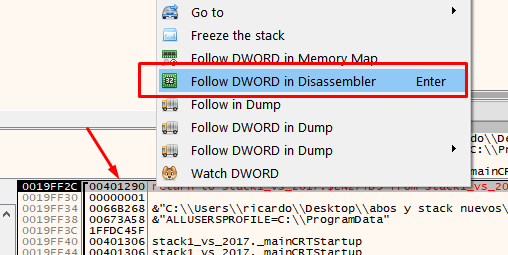
}

Allí vemos los argumentos son dos y se pasan, cuando el ejecutable está compilado en 32 bits, a través del stack.

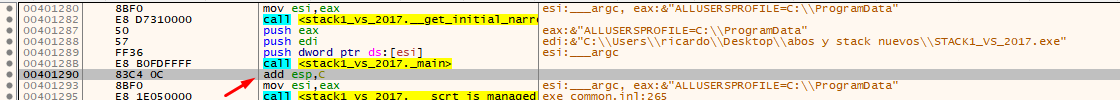
Justo antes de la llamada a la función, se guardarán en el stack los argumentos.

Cuando estamos detenidos en el inicio de una función, en el stack el primer valor será el RETURN ADDRESS o sea donde volverá luego de terminar de ejecutar la función y debajo estaran los argumentos de la misma.

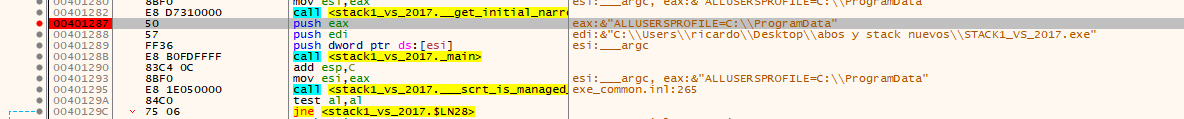
Si hago click derecho en el RETURN ADDRESS y elijo FOLLOW DWORD in DISASSEMBLER, vere donde debería retornar luego de terminar la función.



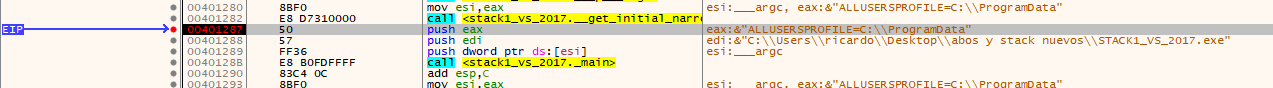
Allí volvería, eso quiere decir que la función main fue llamada desde el call que está justo arriba, puedo poner un breakpoint allí, reiniciar el ejercicio y verificar que es así.

****

Lo pondré allí un poco antes y reinicio.

****

Allí paro



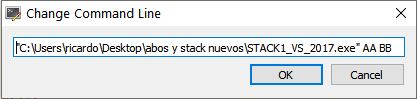
Va a guardar los argumentos de la función main usando esos PUSH.

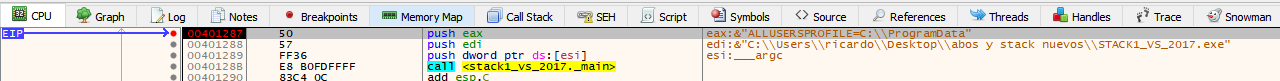
El que quiere profundizar sobre los argumentos de main

<https://publications.gbdirect.co.uk/c_book/chapter10/arguments_to_main.html>

No es muy complicado el primer argumento argc es un int que indica la cantidad de parámetros de consola usados al ejecutar el programa incluyendo el path al ejecutable y argv es un array de punteros a strings.

Vemos que si cambio la línea de comandos pasando más argumentos y reinicio.



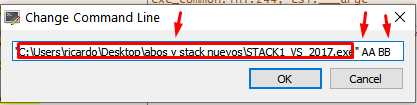


Allí para cuando va a guardar los argumentos usando los PUSH, los primeros que guarda son los más lejanos y el último que guarda será el primer argumento de la función.

Voy traceando y cada PUSH va guardando los argumentos.



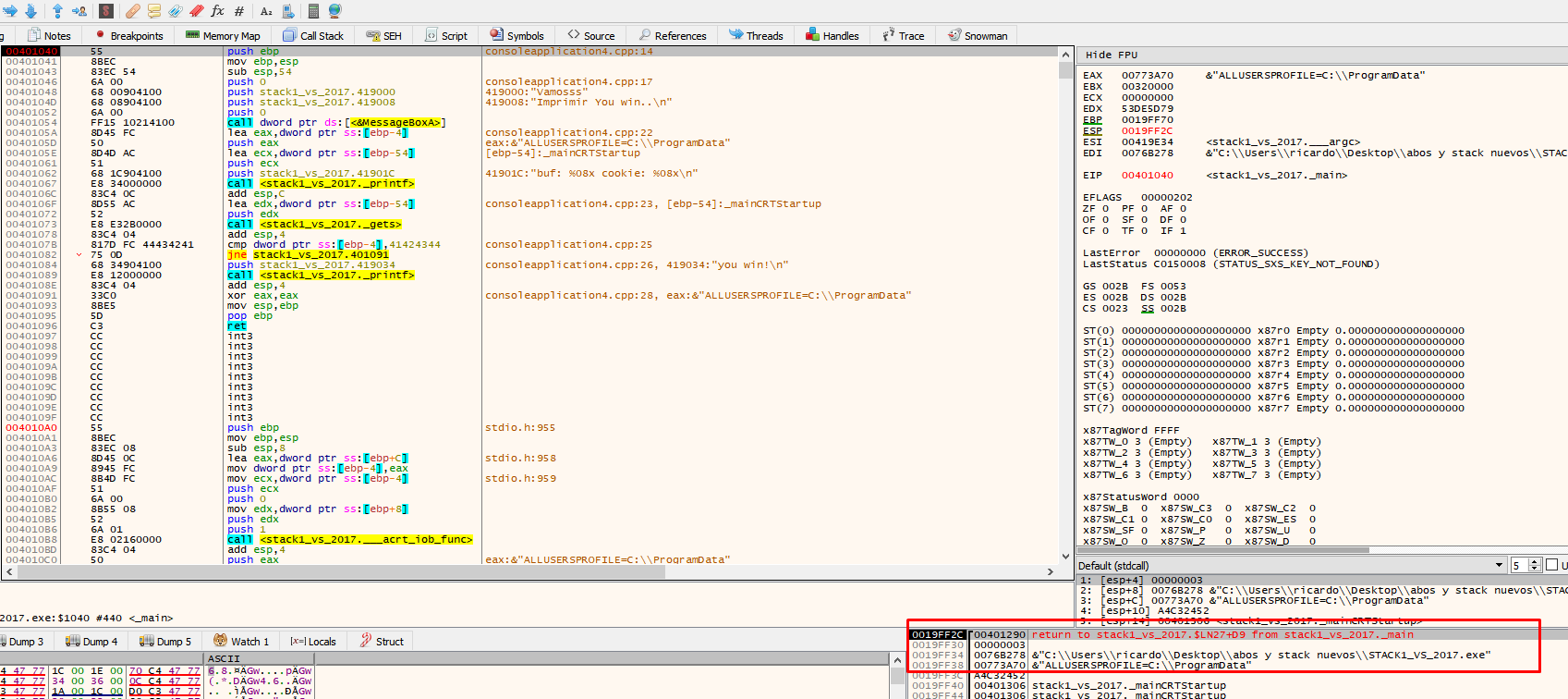
Allí puedo ver los argumentos de la función (no confundir los argumentos de la función main con los argumentos que se pasan por consola al ejecutar el stack1), el de más arriba es el primero es argc que vale 3 ya que marca la cantidad de argumentos que se le pasan por consola y eran tres contando el path del ejecutable.



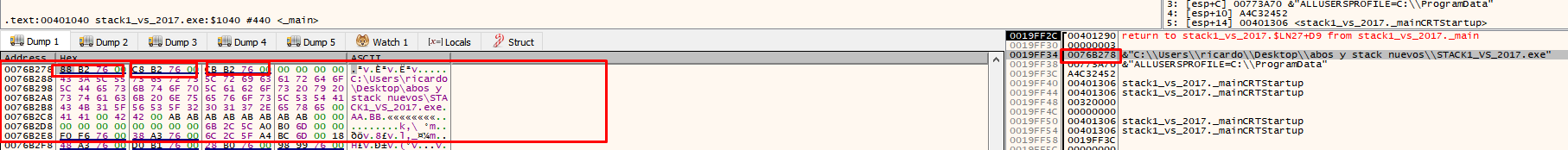
Esos son los 3 argumentos.

Ahora apretamos f7 para que haga un STEP INTO y entre en la función.

Allí vemos que al entrar en el CALL guarda el RETURN ADDRESS en el stack.

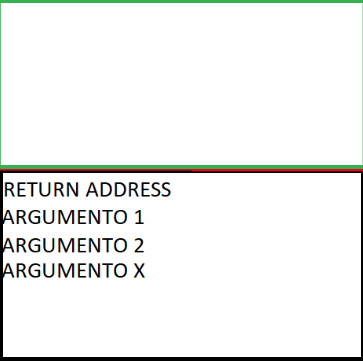


Así que como dijimos al estar detenidos en el inicio de una función, lo que estará en la parte superior del stack será el RETURN ADDRESS y en 32 bits debajo del mismo estaran los argumentos, primero el primer argumento y sucesivamente hacia abajo el resto.



El segundo argumento como vimos es un array de punteros, allí vemos en la memoria que hay tres punteros a las tres strings que pase como argumentos.

# 32 BITS



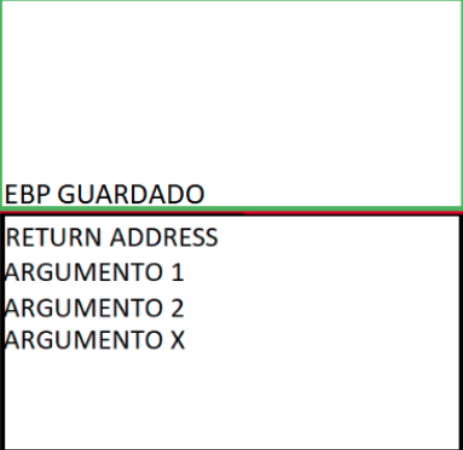
Allí estamos detenidos en el inicio de la función, justo tenemos debajo el RETURN ADDRESS y los ARGUMENTOS.

Aclaramos que es una compilación en 32 bits porque en 64 bits los argumentos se pasan de otra forma ya veremos más adelante.

La cuestión que luego se empieza a ejecutar la función, lo primero es el llamado PRÓLOGO, que guarda el valor de EBP de la función que llamó a la nuestra.

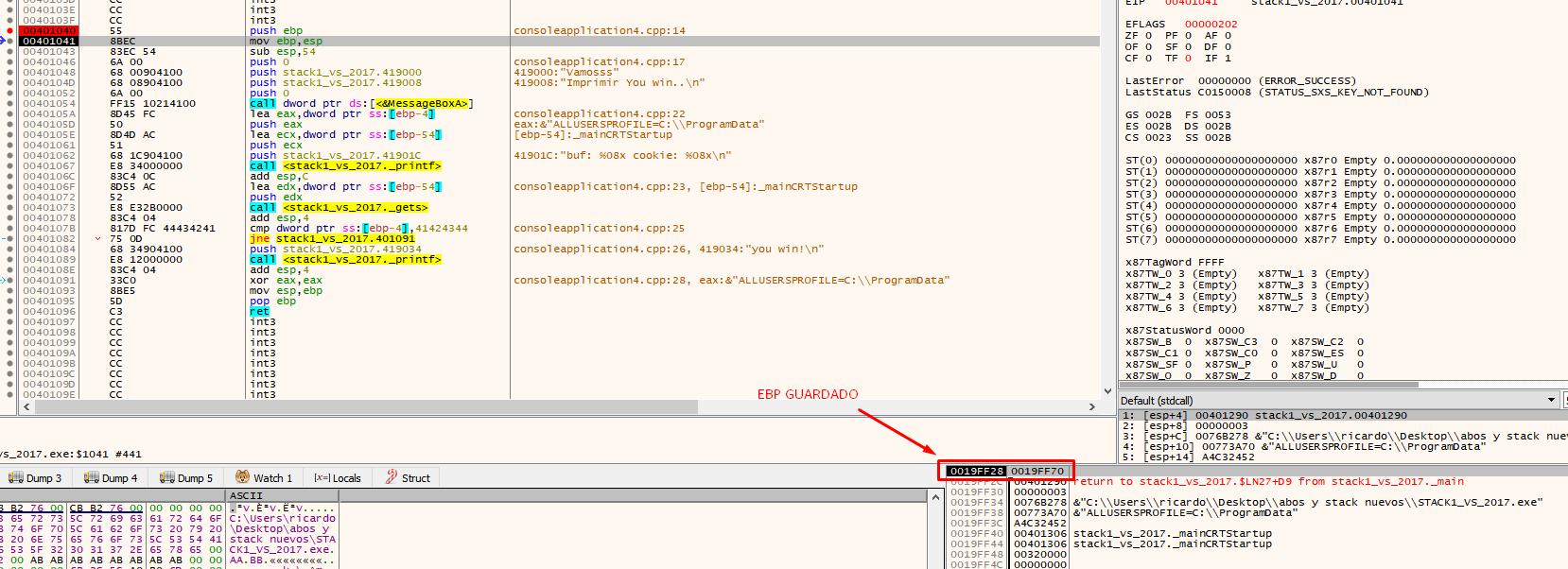


Eso hará que el valor de EBP GUARDADO se guarde justo encima del return address.



Si ejecuto con f7.

Veo que el valor de EBP GUARDADO se ubica en el stack arriba del return address

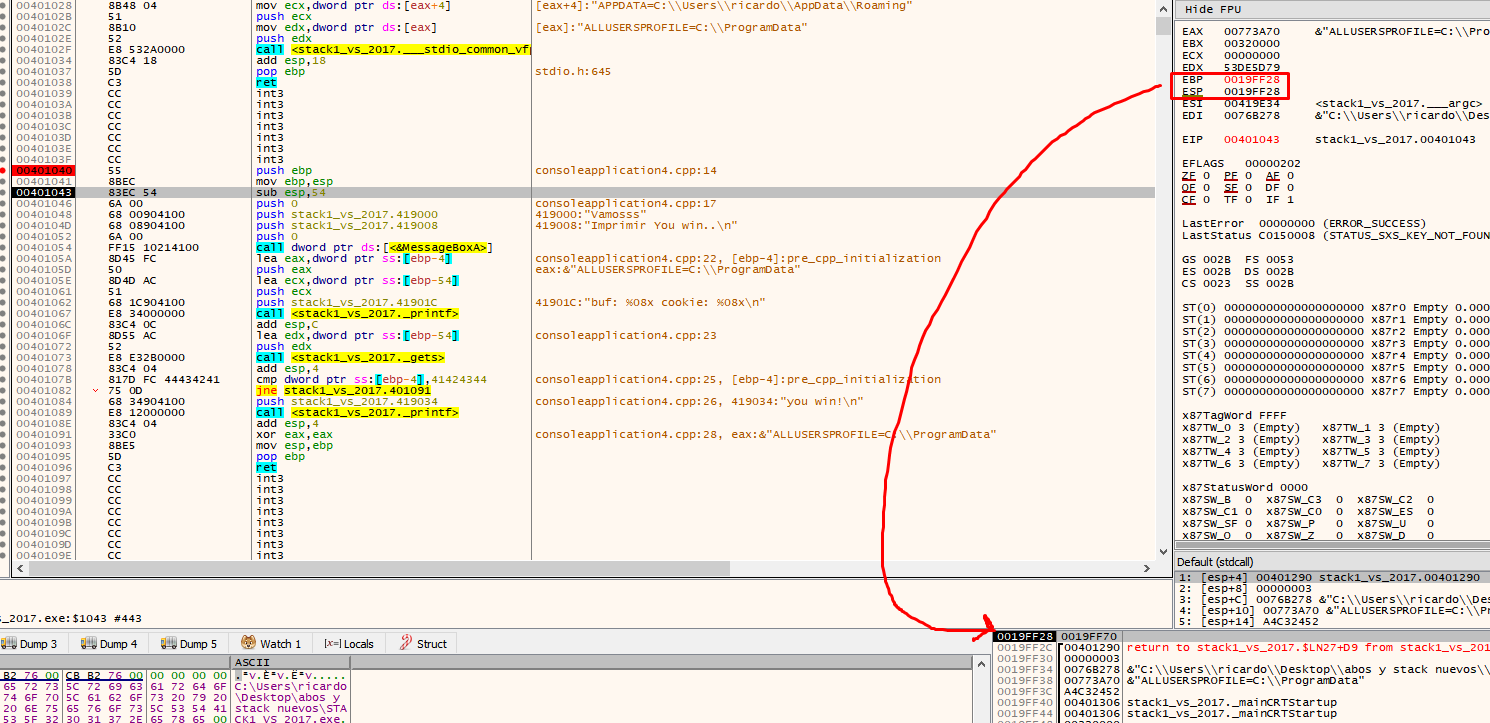


La siguiente instrucción del PRÓLOGO es

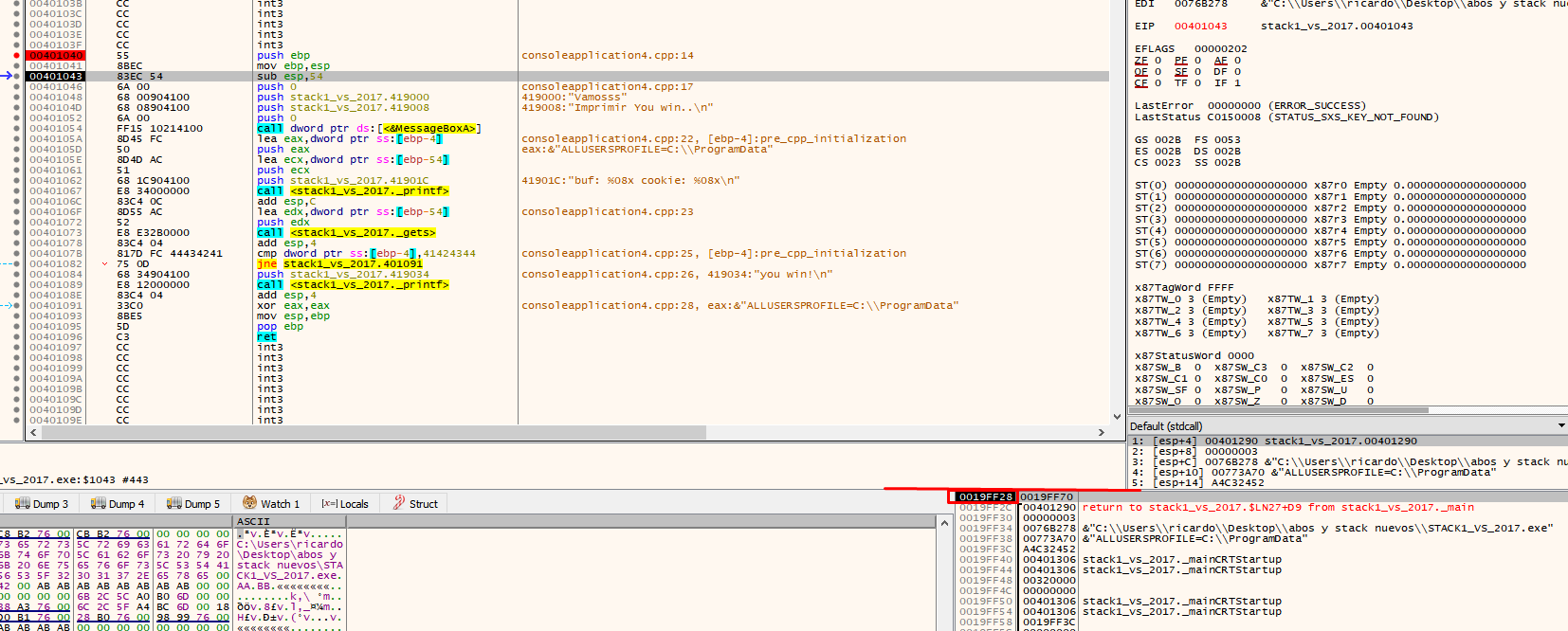
mov ebp, esp

Lo que hace es setear el EBP para la función actual, el que estaba guardado era el de la función padre que llama a la nuestra (en este caso mi función main es EBP BASED en otros casos puede diferir ya los veremos más adelante)

Al poner a EBP al valor actual de ESP, lo que logramos es crear el marco de nuestra función actual.



Lo que ocurre es que a partir de ahora ya que es una función BASADA EN EBP o EBP BASED, dentro de la función se mantendrá el valor de EBP y se tomará como referencia el mismo, mientras que ESP variara.



Es muy importante este valor de EBP que se toma como referencia.

En las funciones EBP BASED las variables y argumentos se pueden nombrar por su distancia a esta dirección que estará guardada en el valor de EBP hasta el epílogo de la misma.

Podemos ver en el listado varias variables que están mencionadas como EBP-4 o EBP-54 referidas al valor de EBP que toma en este punto.

A todo esto podemos decir que una vez que EBP toma su valor después del PRÓLOGO, será como una divisoria de aguas, por lo tanto los argumentos estaran siempre hacia abajo de esa dirección por lo cual EBP+XXX se refiere a argumentos (el ebp guardado y el return address están debajo también aunque no tendrán referencias en el código), mientras que las variables como veremos estaran por encima de esta dirección, por lo cual una mención a EBP - XXX restando, refiere a alguna variable local.

Así que en general en funciones EBP BASED

EBP + XXXX = argumentos proporcionados al llamar la funcion.

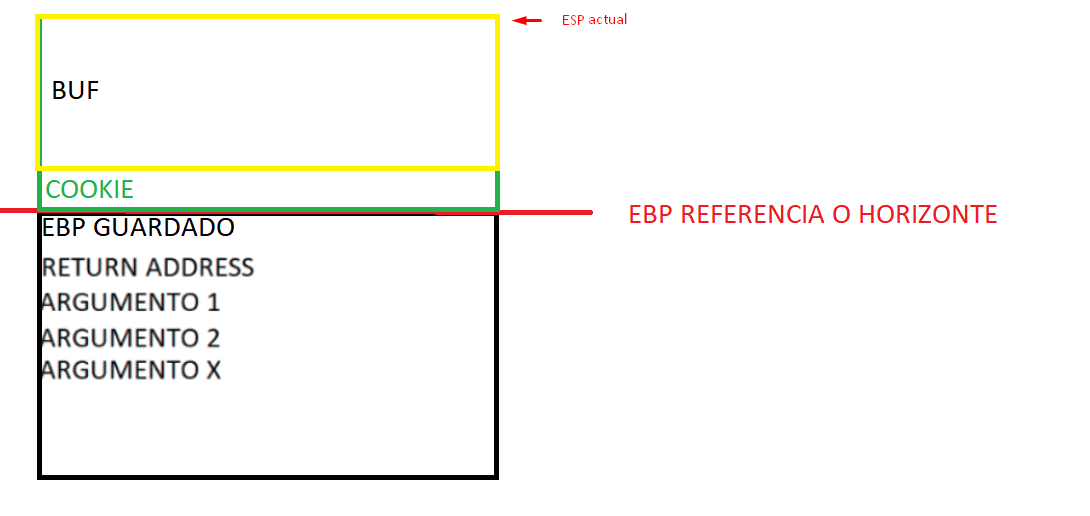
EBP - XXXX = variables locales de la función

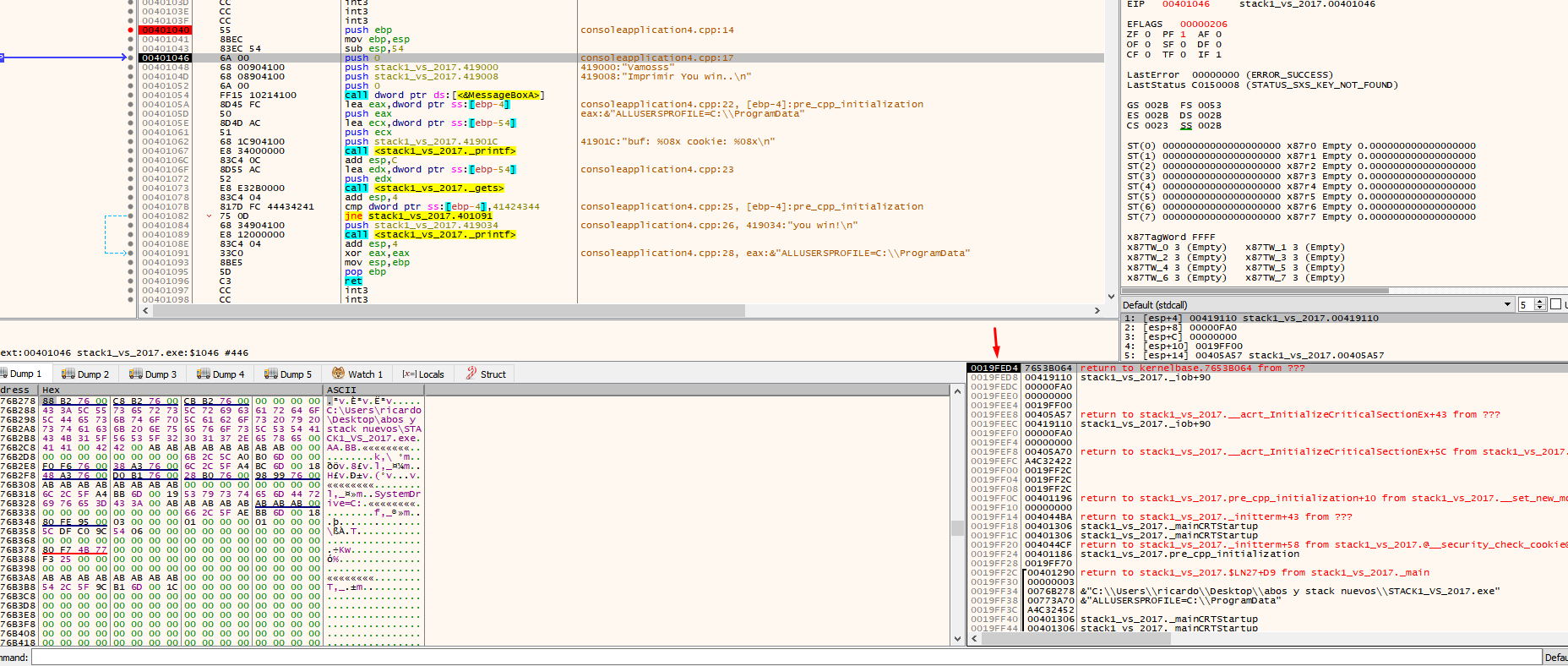
Luego del PRÓLOGO habrá alguna forma de reservar espacio para las variables, en nuestro caso eso se realiza moviendo ESP hacia arriba para que quede espacio para la suma de todos los largos de las variables y a veces un poco más por si acaso, eso depende del compilador.

00401043 | 83EC 54 | sub esp,54

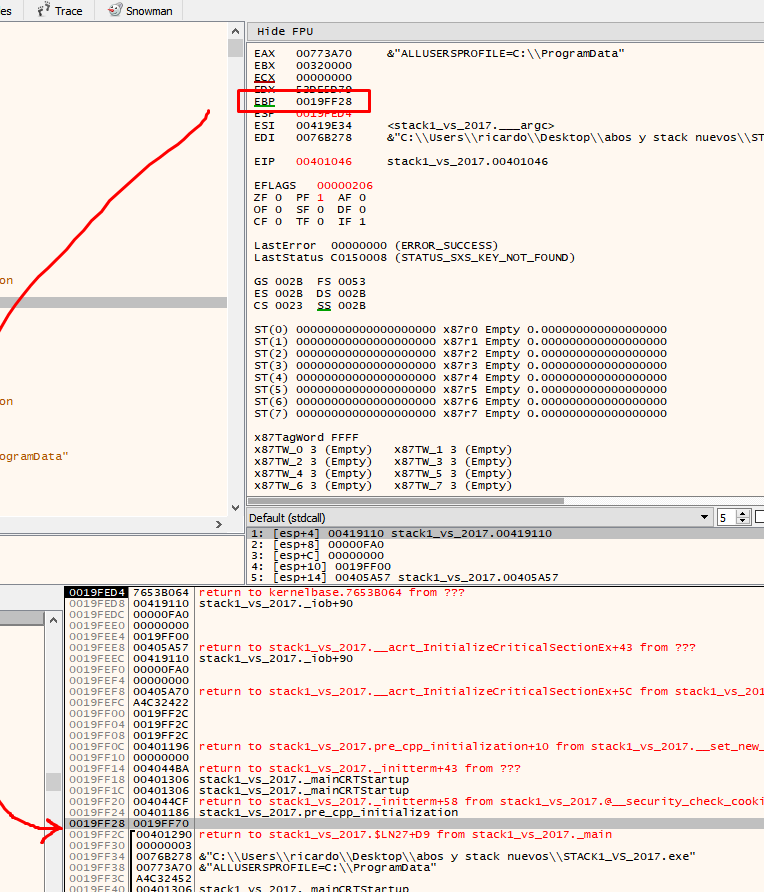
Vemos que ESP se ubicara por encima de EBP que quedará como referencia, y ese 0x54 pasado a decimal es 84 que es la suma del largo de buf y cookie recordemos que eran 80 y 4 respectivamente.



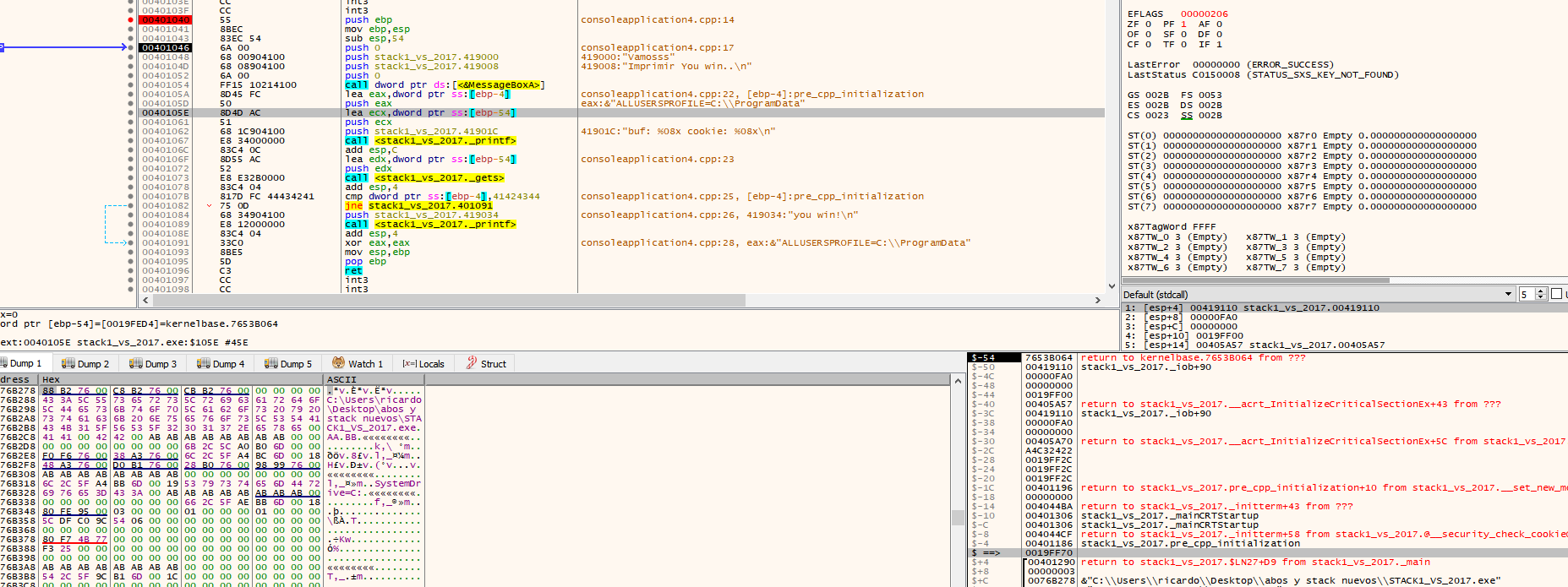




Al ejecutar se crea un espacio para las variables buf y cookie de 84 bytes, se puede hacer click en la primera columna en la dirección horizonte, miramos EBP y ese valor lo buscamos en el stack obviamente ahora estará más abajo.

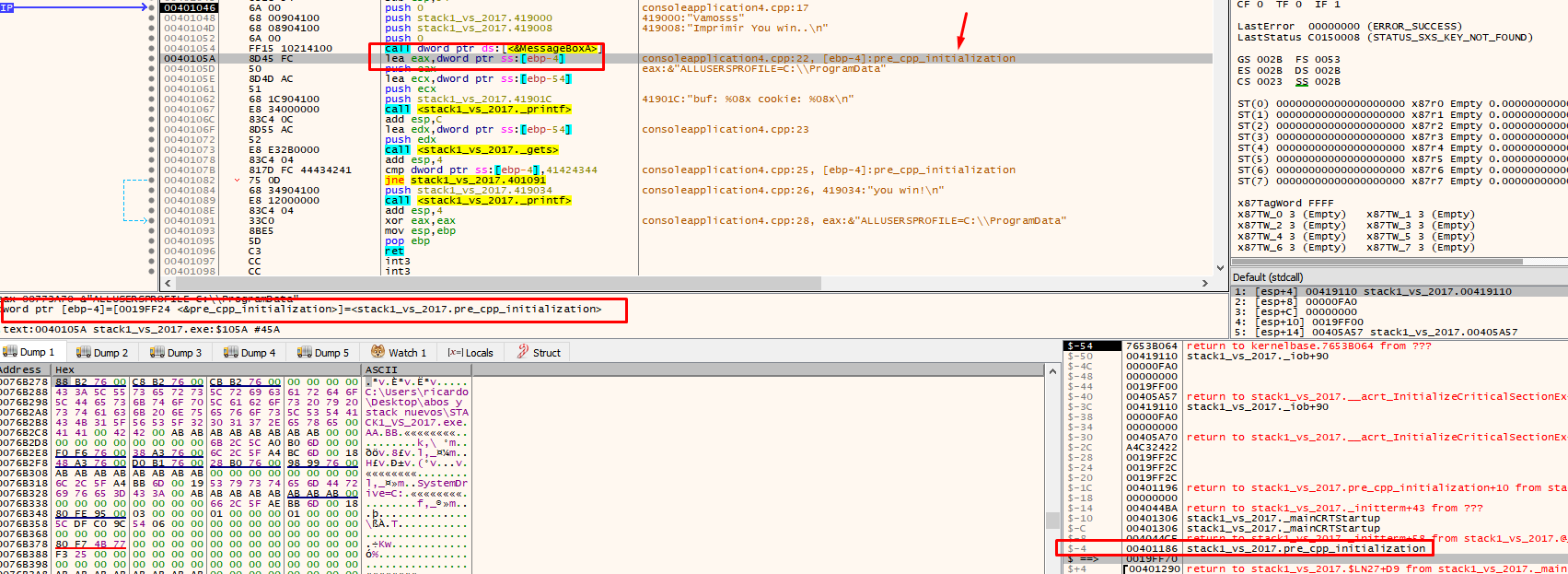


Hago doble click allí.

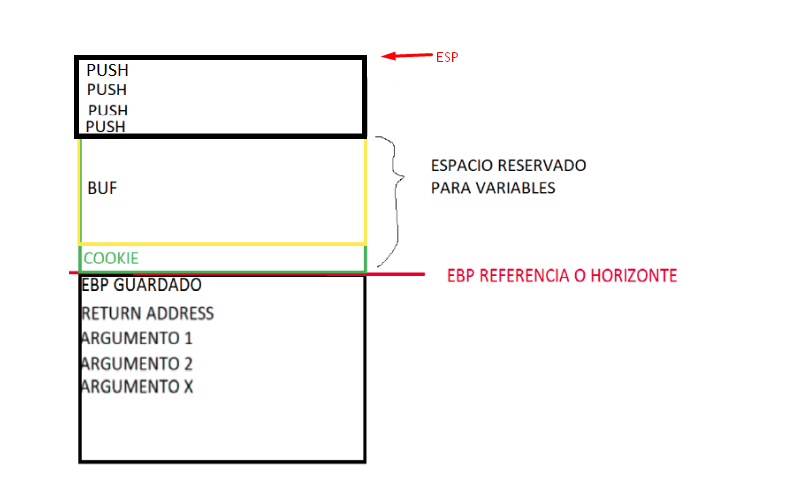


De esta forma tendremos los valores respecto a EBP también en el stack.

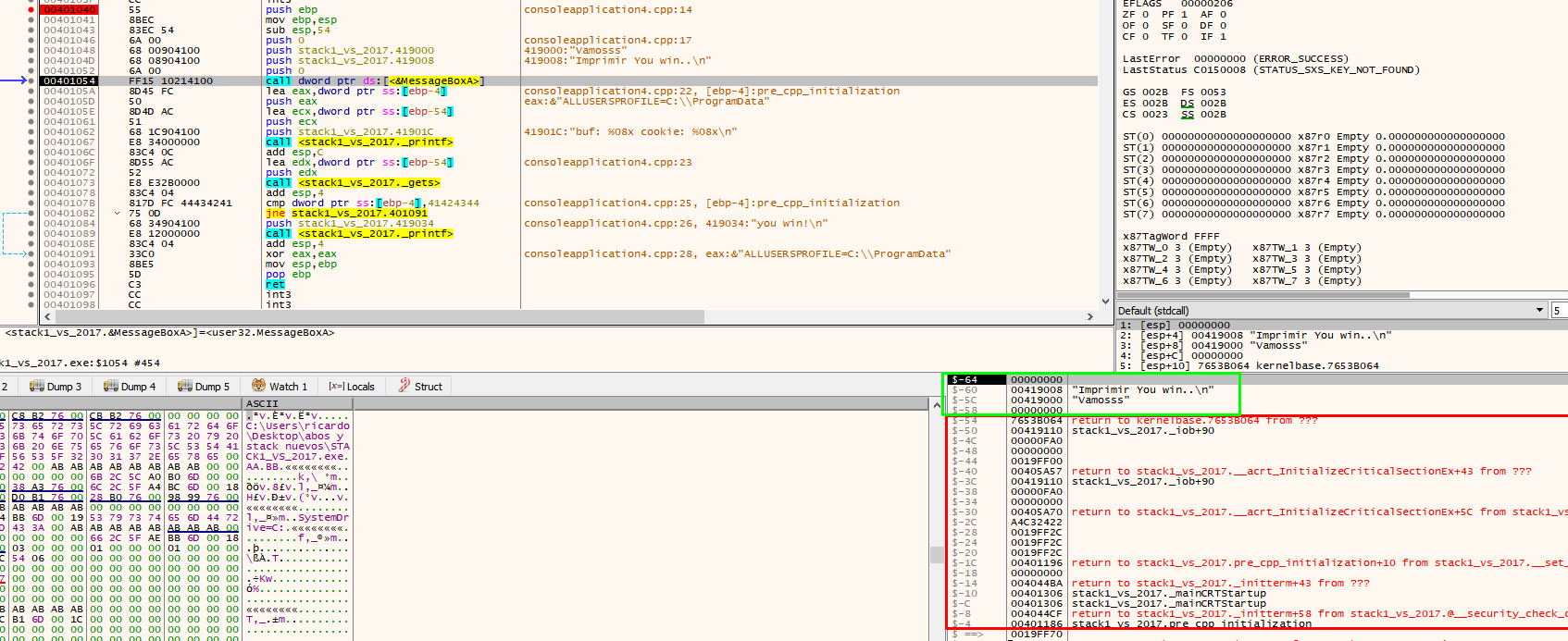
Por ejemplo ebp-4 coincide tanto en el listado como en el -4 del stack y en la aclaración también, figura como ebp-4.



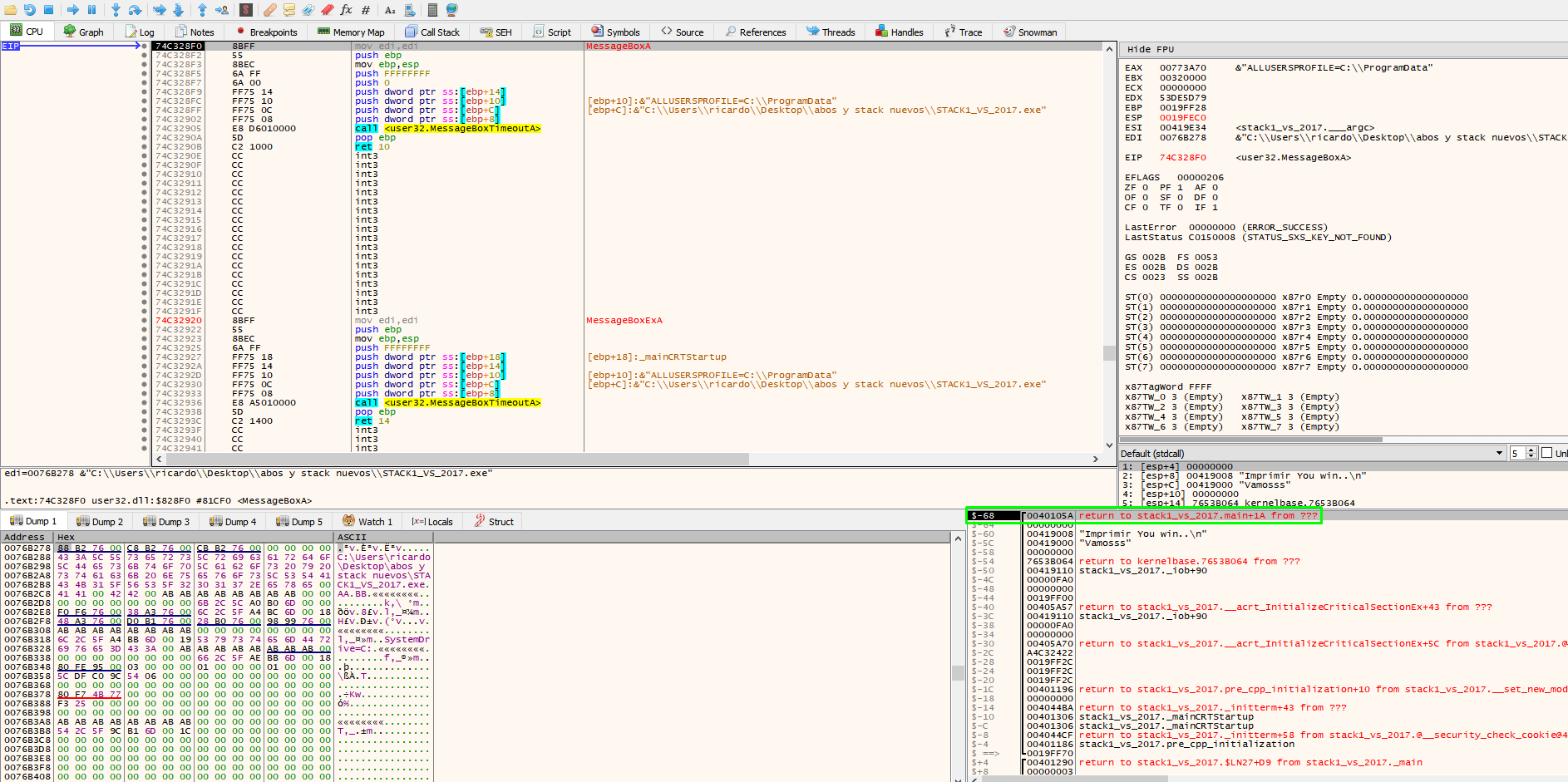
Si traceo paso a paso, vemos que a partir del lugar donde se ubico ESP para reservar variables, se moverá siempre hacia arriba pues debe respetar el espacio asignado para las variables, al hacer los 4 push para el MessageBoxA los ubica arriba del espacio reservado y sube ESP.



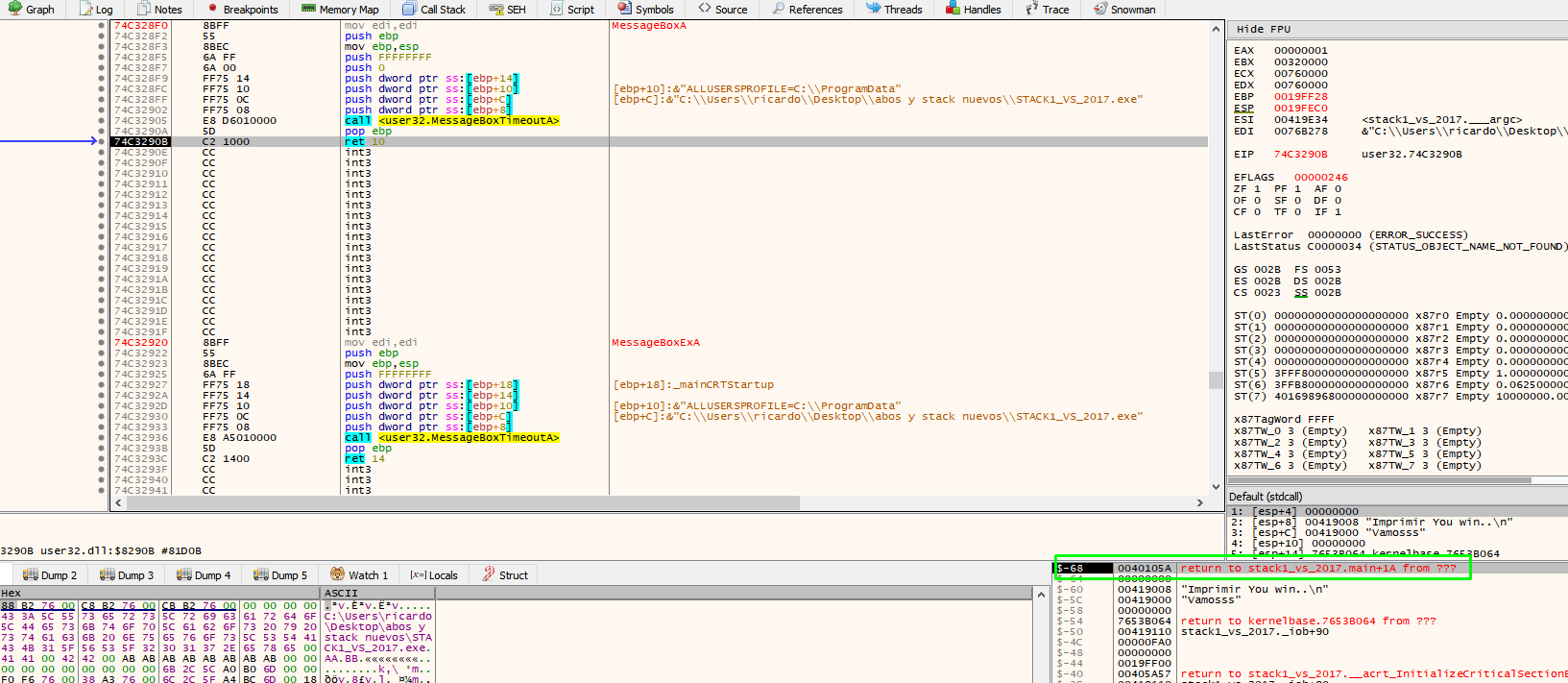
Si miro el stack veo en verde los 4 argumentos que agrego arriba del espacio reservado que está marcado en rojo.



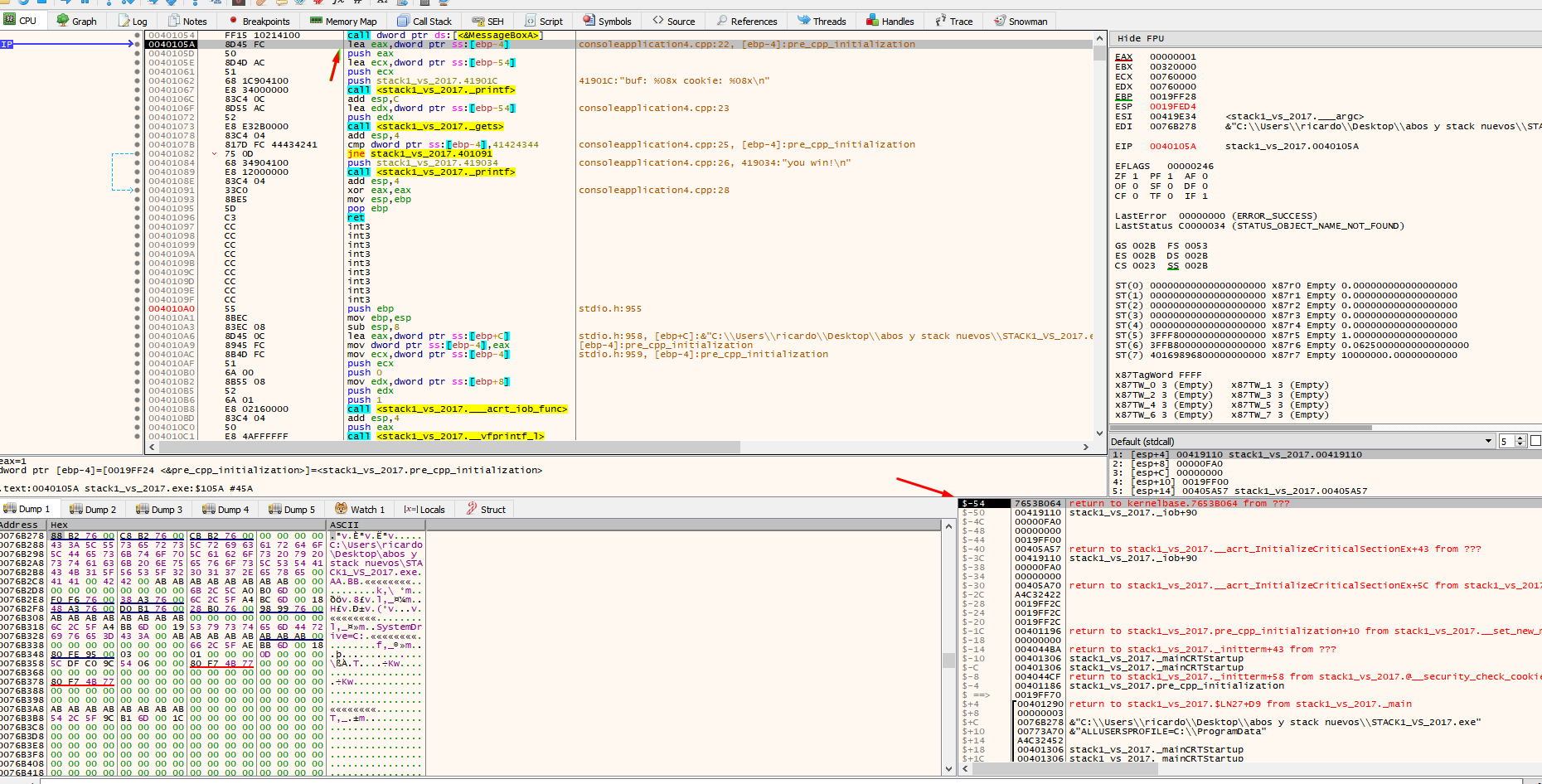
Al entrar en MessageBoxA se guarda el RETURN ADDRESS de esa función en el stack.



Allí está el return address de MessageBoxA, cuando llego al RET de dicha función traceando con F8 y acepto el MessageBox.

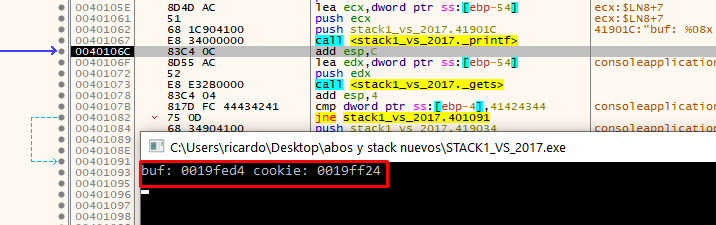


Vemos que volverá justo debajo del call a MessageBoxA.



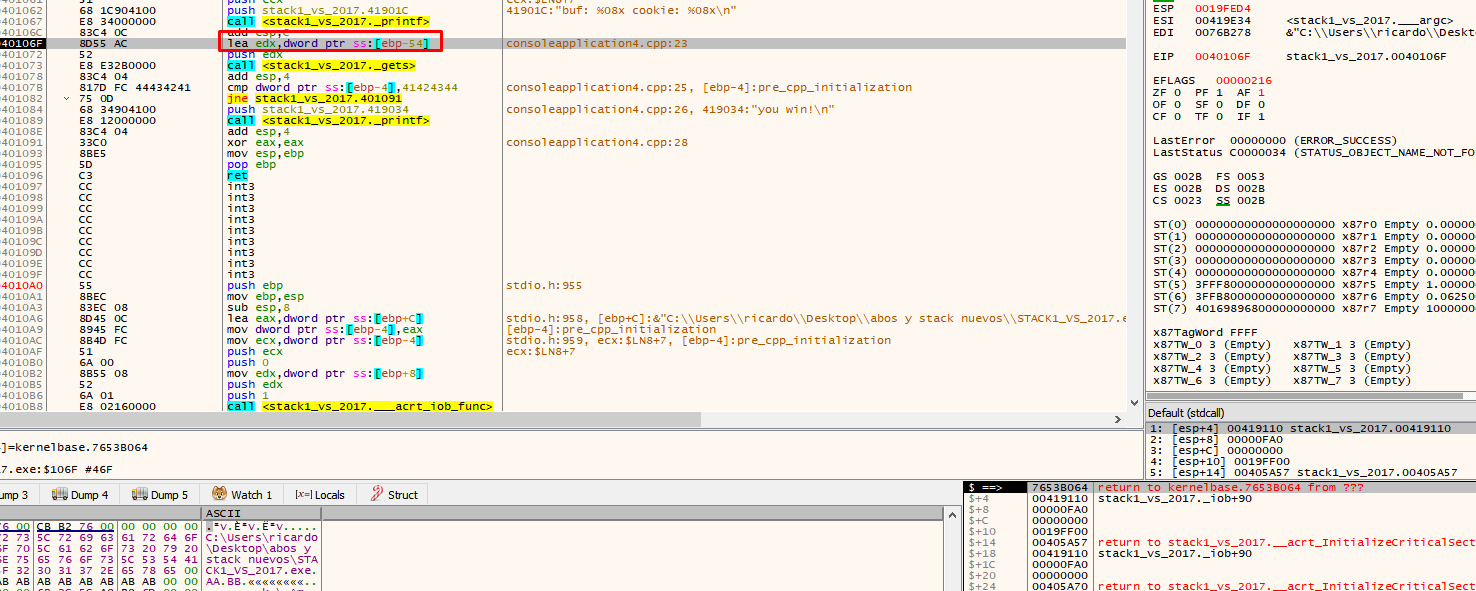
Y que los PUSH que había guardado para MessageBoxA y el RETURN ADDRESS de dicha función ya se usaron y se quitaron y ESP vuelve a estar justo arriba de la zona reservada como antes de llamar a cualquier función, lo mismo pasará con la llamada a printf guardará los PUSH y el RETURN ADDRESS de la misma y ESP subirá pero al salir de la misma bajara nuevamente justo encima de la zona reservada.

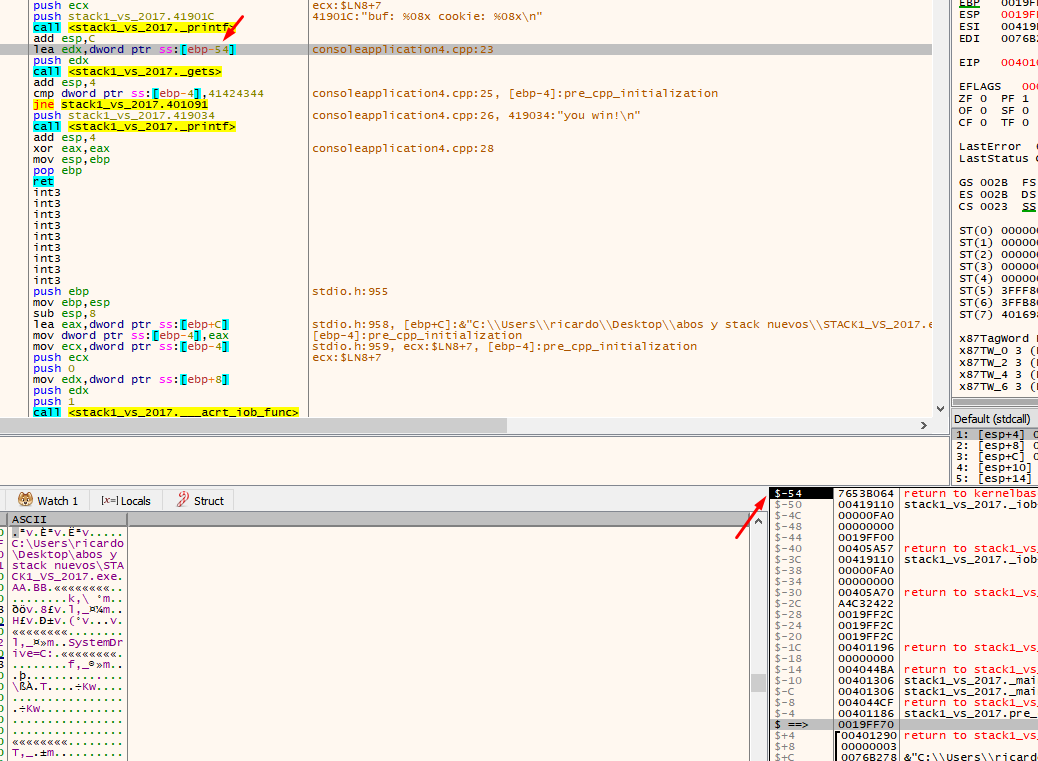
Una vez que pase por encima del printf se imprimen las direcciones de buf y cookie.



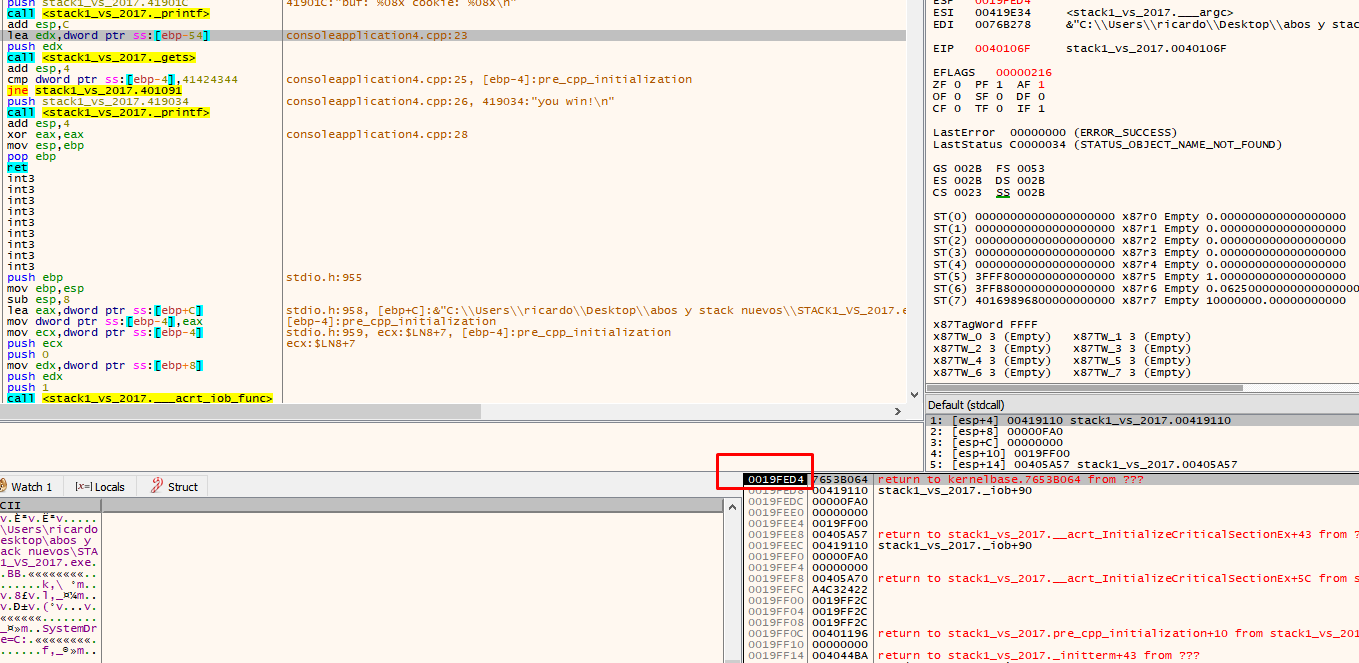
La dirección de buf en mi maquina seria 19fed4 y la de cookie 19ff24.

Allí lee la dirección de buf para pasarsela a gets y llenar el buf, podemos chequear que la dirección coincide con lo que muestra la consola 19fed4

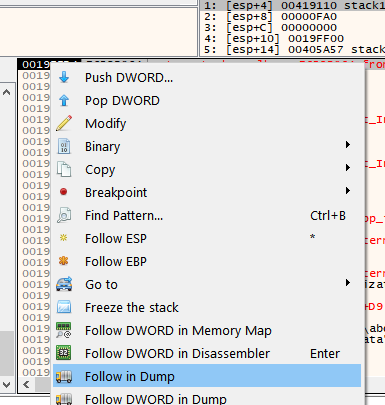


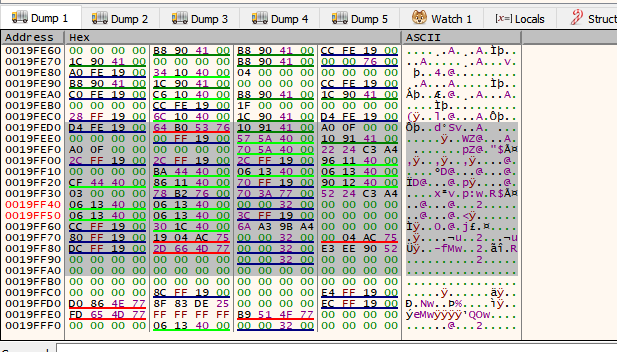


Allí vemos que es EBP-54 si hago doble click en el stack donde muestra -54, mostrará la dirección de buf=19fed4 en mi maquina.



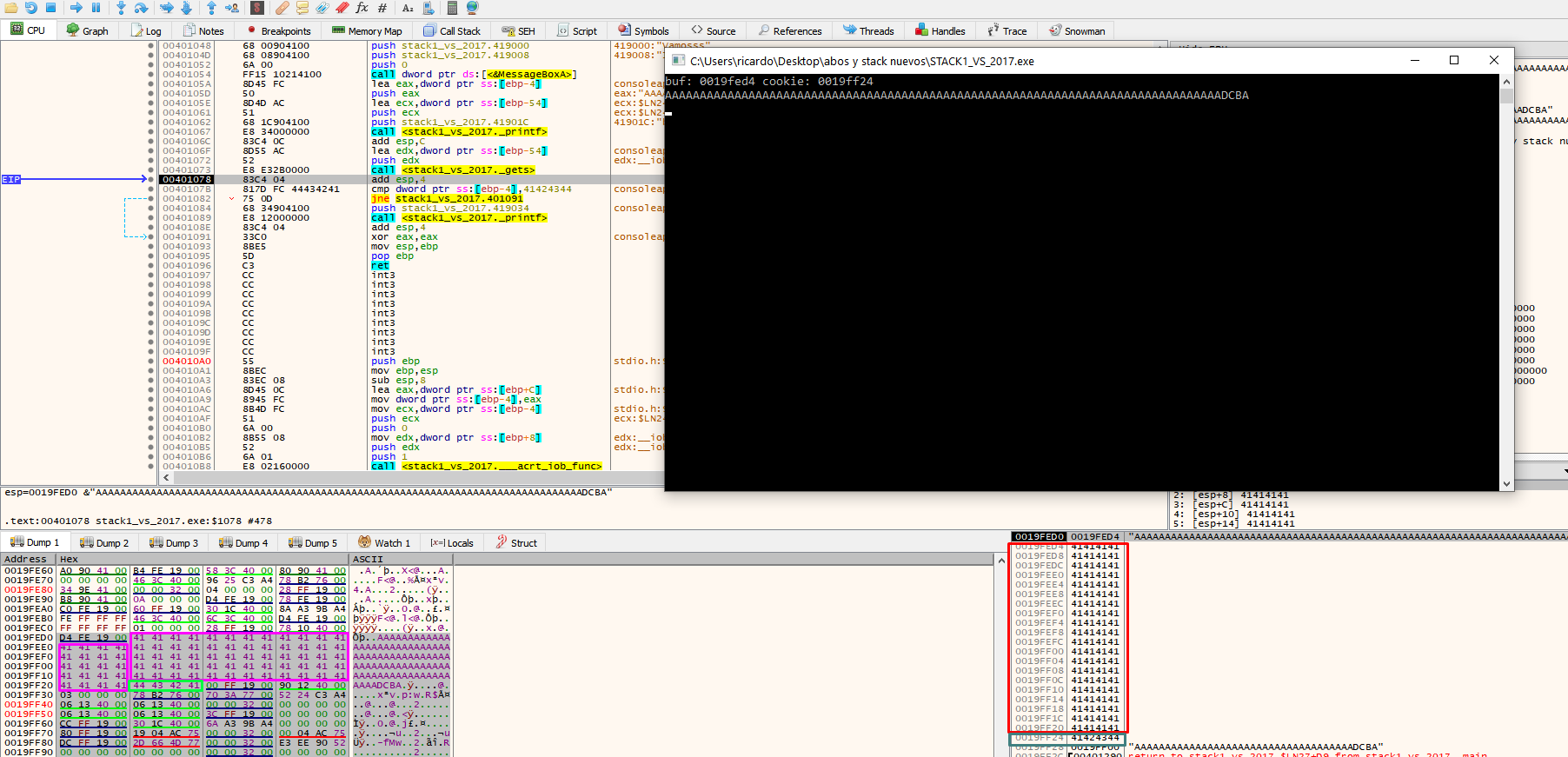
Ahora como esa direccion es donde guardará los datos que tipeo, puedo ponerla en el dump para ver como guarda allí los bytes también.





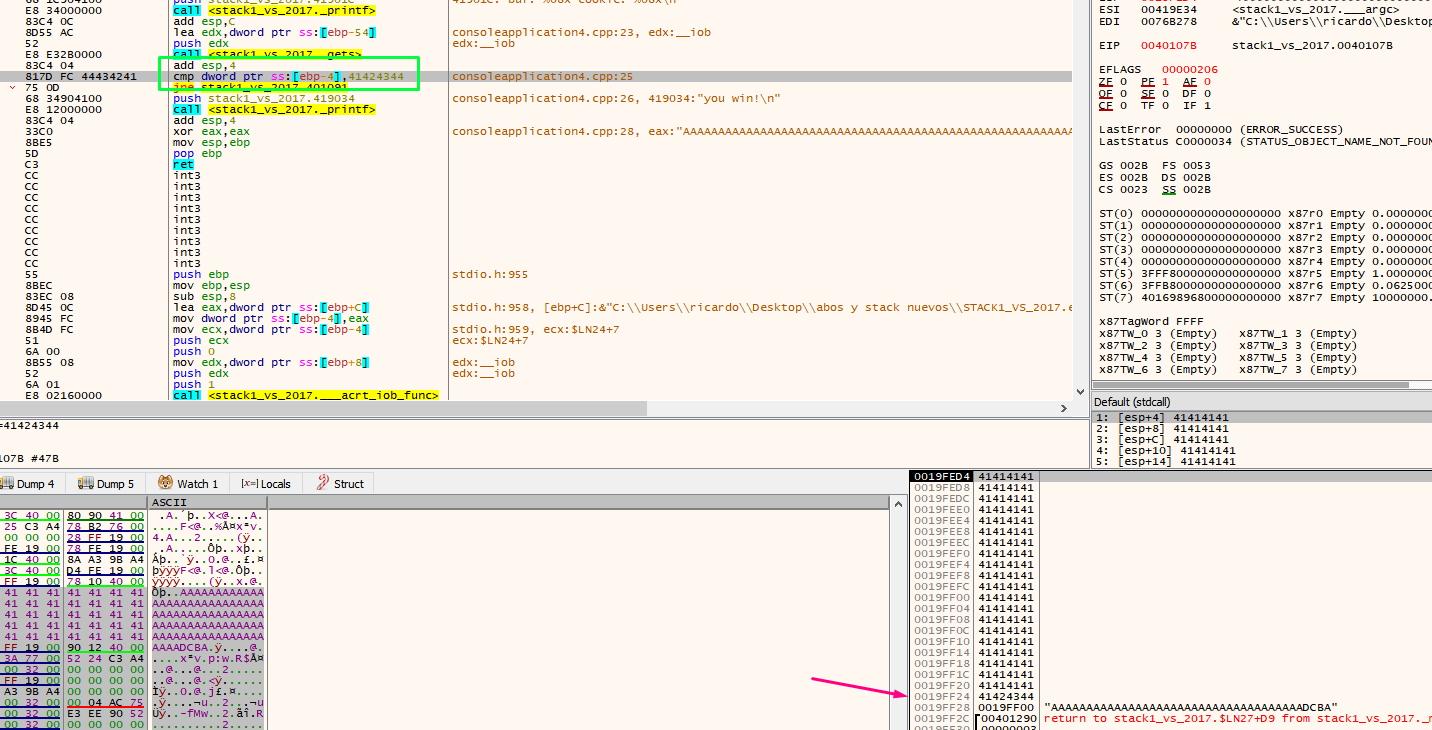
Allí está, no lo sube mas porque no hay mas data abajo.

Cuando paso el gets por encima con F8 me tendré que ir a la consola y tipear y al apretar ENTER llenará el buffer buf y pisara cookie.

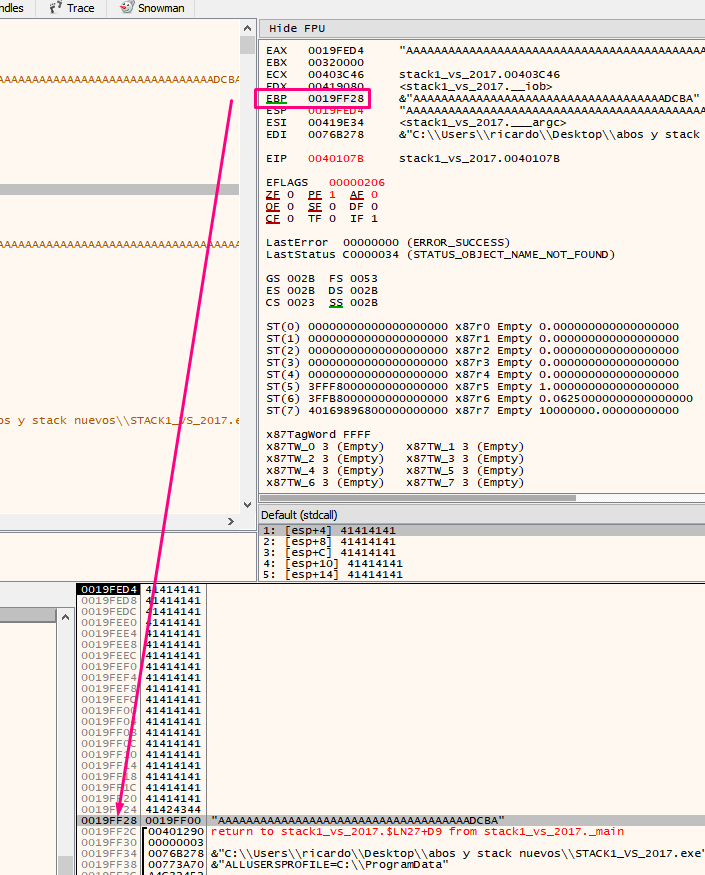


Vemos que cookie estaba en la dirección 19ff24 en mi maquina.

Allí compara cookie con 0x41424344.

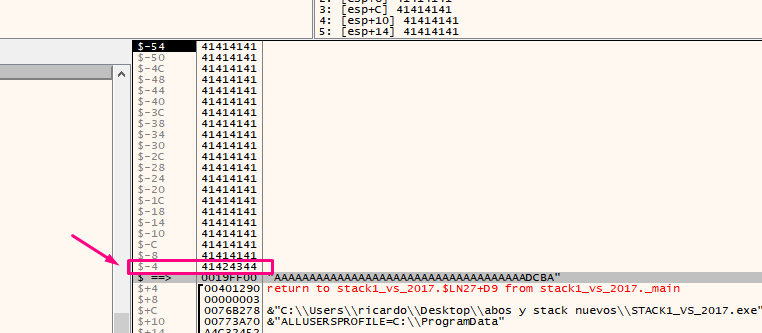


Vemos que dice EBP-4 que es cookie, además de las dirección si volvemos a poner a cero el HORIZONTE al valor de EBP como antes.

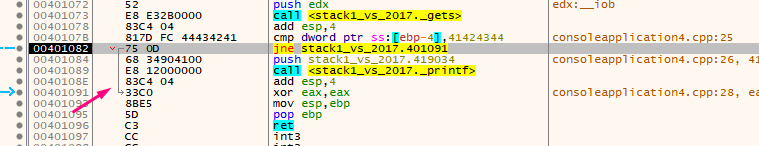


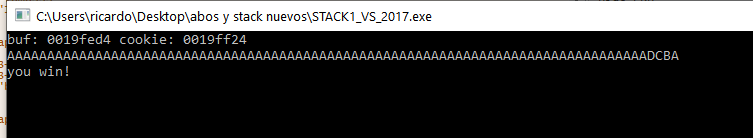
Hago doble click ahi.

Vemos que EBP-4 es cookie ya que está en el -4 del stack poniendo el cero en el HORIZONTE.



Vemos que no va a saltar y va a ir a YOU WIN.





Allí logramos el objetivo manualmente que diga YOU WIN.

Hemos analizado dinámicamente el stack1, utilizando X64dbg que es un debugger y no nos permite analizarlo sin correr el programa, para ello debemos utilizar otras herramientas como IDA PRO, GHIDRA o RADARE.

Puedo hacer un modelo de script para explotar el ejercicio desde Python.

**import** sys

**from** subprocess **import** Popen, PIPE

payload = **b"A"** \* 80 + **b"\x44\x43\x42\x41"**

p1 = Popen(**r"C:\Users\ricardo\Desktop\abos y stack nuevos\STACK1\_VS\_2017.exe"**, stdin=PIPE)

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

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

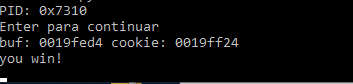
p1.communicate(payload)

p1.wait()

input()

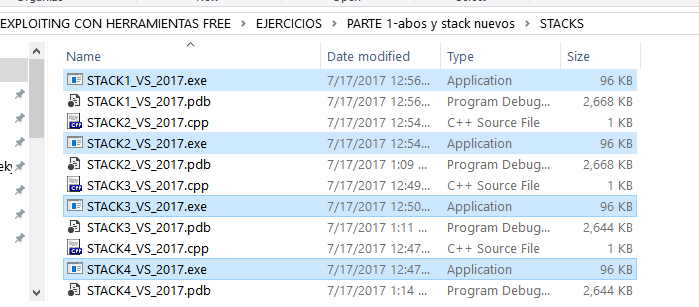
Como es para Python 3 debo colocar los paréntesis en los prints y tener cuidado al sumar cadenas que deben ser bytes (poner b delante de lo que eran strings en Python 2).

Me fijo que el path este correcto y al correrlo.



Bueno ya tenemos un modelo de script en Python 3 para explotar el stack1, en la próxima entrega seguiremos con el análisis estático en IDA, RADARE y GHIDRA de este mismo ejercicio.

Mientras pueden intentar tracear y entender con X64dbg por ahora los otros 3 stacks que son muy parecidos a este.



Ven que además del stack1 están el 2 el 3 y el 4, pueden intentar resolverlos son muy sencillos y parecidos al stack1, asi no se aburren jeje.

La próxima IDA FREE, RADARE y GHIDRA.

Nos vemos en la próxima parte 3.

Ricardo Narvaja

25/10/2019