Reversing Practice Report (2025)

Author: Pedro Oller Serrano

07/03/2025

Statement of practice

An executable binary file has been obtained that, after execution, displays a text with a numerical code. This code is generated from a string of text stored in the binary itself.

After listing the ASM code, the binary file was deleted and it is not possible to access it, only the ASM code copied at the end of the exercise.

This code generation algorithm needs to be reused, so reverse engineering is required to analyze the binary and reconstruct the source code so that it can be modified and recompiled correctly.

To carry out this task, it is requested:

- **Split the code into *basic blocks*.**
- **Make a flowchart with the *basic blocks*. Is there a control structure?, Indicates which *basic blocks* are involved in it.**
- **Converts the full function code to C code.**
- **It compiles the generated code and indicates the resulting code after execution.**
- **Modifies the code in C so that it generates new code from another string.** Modify the string <+36> in code C to the following string: *Congratulations!* It compiles the C code, executes and indicates the full text obtained.

ASM code –x86 32-bit:

```
Dump of assembler code for function main:

0x0000054d <+0>: lea ecx,[esp+0x4]
0x00000551 <+4>: and esp,0xfffffff0
0x00000554 <+7>: push DWORD PTR [ecx-0x4]
0x00000557 <+10>: push ebp
0x00000558 <+11>: mov ebp,esp
0x0000055a <+13>: push ebx
0x0000055b <+14>: push ecx
0x0000055c <+15>: sub esp,0x10
0x0000055f <+18>: call0x450 < x86.get_pc_thunk.bx>
0x00000564 <+23>: add ebx,0x1a9c
0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax
0x0000057a <+45>: sub esp,0xc
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
0x00000585 <+56>: add esp,0x10
0x00000588 <+59>: mov DWORD PTR [ebp-0x18],eax
0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
0x00000592 <+69>: jmp 0x5ad <main+96>
```

```
0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc]
0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14]
0x0000059a <+77>: add eax,edx
0x0000059c <+79>: movzx eax,BYTE PTR [eax]
0x0000059f <+82>: movsx eax,al
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1
0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc]
0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18]
0x000005b3 <+102>: JL 0x594 <Main+71>
0x000005b5 <+104>: sub esp,0x8
0x000005b8 <+107>: push DWORD PTR [ebp-0x10]
0x000005bb <+110>: lea eax,[ebx-0x1992] ; “[+] Codigo generado: %i\n”
0x000005c1 <+116>: push eax
0x000005c2 <+117>: call 0x3d0 <printf@plt>
0x000005c7 <+122>: add esp,0x10
0x000005ca <+125>: mov eax,0x0
0x000005cf <+130>: lea esp,[ebp-0x8]
0x000005d2 <+133>: pop ecx
0x000005d3 <+134>: pop ebx
0x000005d4 <+135>: pop ebp
0x000005d5 <+136>: lea esp,[ecx-0x4]
0x000005d8 <+139>: p.

End of assembler dump.
```

Resolution of the practice

Splitting code into *Basic Blocks*

Definition of *Basic Blocks,* are sequences of instructions that do not contain breaks and that, in addition, do not have intermediate entry points. That is, they are separated by jumps and function calls.

The ASM code –x86 32-bit, can be divided into 9 *basic blocks* for high grain, as discussed below:

1. **Stack and regitros:** runs sequentially, without interruptions and without jumps.

```
0x0000054d <+0>: lea ecx,[esp+0x4]
0x00000551 <+4>: and esp,0xfffffff0
0x00000554 <+7>: push DWORD PTR [ecx-0x4]
0x00000557 <+10>: push ebp
0x00000558 <+11>: mov ebp,esp
0x0000055a <+13>: push ebx
0x0000055b <+14>: push ecx
0x0000055c <+15>: sub esp,0x10
```

2. **Getting *pc_thunk* and Logging Settings:**

```
0x0000055f <+18>: call 0x450 < x86.get_pc_thunk.bx>
0x00000564 <+23>: add ebx,0x1a9c
```

3. **Initialization of variables on the stack:** No conditional hops or function calls.

```
0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax
```

4. **Call to the *'strlen' function:*** The length of the string is obtained.

```
0x0000057a <+45>: sub esp,0xc
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
0x00000585 <+56>: add esp,0x10
0x00000588 <+59>: mov DWORD PTR [ebp-0x18],eax
```

5. **Loop initialization:** Jumps to the *"main +96" label* where it checks the condition of the loop.

```
0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
0x00000592 <+69>: jmp 0x5ad <main+96>
```

6. **'for or while' loop*:* *Where the string*** "*3jd9cjfk98hnd" is processed character by character.*

```
0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc]
0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14]
0x0000059a <+77>: add eax,edx
```

```
0x0000059c <+79>: movzx eax,BYTE PTR [eax]
0x0000059f <+82>: movsx eax,al
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1
```

7. **Loop conditional:** If it does not meet the loop condition "eax < DWORD PTR [ebp-0x18]". That is, if the counter is less than the length of the string, it returns to the initial label *"main +71"*.

```
0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc]
0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18]
0x000005b3 <+102>: JL 0x594 <Main+71>
```

8. **Execution of *"printf"*:** If the loop ends, "Code generated: %i\n" is printed on the screen, showing the result.

```
0x000005b5 <+104>: sub esp,0x8
0x000005b8 <+107>: push DWORD PTR [ebp-0x10]
0x000005bb <+110>: lea eax,[ebx-0x1992] ; "[+] Codigo generado: %i\n"
0x000005c1 <+116>: push eax
0x000005c2 <+117>: call 0x3d0 <printf@plt>
0x000005c7 <+122>: add esp,0x10
```

9. **Control Cleanup and Return:** In this block, the stack is restored and control is returned.

```
0x000005ca <+125>: mov eax,0x0
0x000005cf <+130>: lea esp,[ebp-0x8]
0x000005d2 <+133>: pop ecx
0x000005d3 <+134>: pop ebx
0x000005d4 <+135>: pop ebp
0x000005d5 <+136>: lea esp,[ecx-0x4]
0x000005d8 <+139>: p.
```

Actually, if less granularity is required, these 9 blocks could be reduced to 4 without affecting the flow since blocks that do not have intermediate forks can be grouped, as can be observed:

1. **Block 1 + block 2 + block 3 + block 4 + block 5:**

```
0x0000054d <+0>: lea ecx,[esp+0x4]
 0x00000551 <+4>: and esp,0xfffffff0
0x00000554 <+7>: push DWORD PTR [ecx-0x4]
0x00000557 <+10>: push ebp
0x00000558 <+11>: mov ebp,esp
0x0000055a <+13>: push ebx
0x0000055b <+14>: push ecx
0x0000055c <+15>: sub esp,0x10
0x0000055f <+18>: call0x450 < x86.get_pc_thunk.bx>
0x00000564 <+23>: add ebx,0x1a9c
```

```
0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; “3jd9cjfk98hnd”
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax
0x0000057a <+45>: sub esp,0xc
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
0x00000585 <+56>: add esp,0x10
0x00000588 <+59>: mov DWORD PTR [ebp-0x18],eax
0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
0x00000592 <+69>: jmp 0x5ad <main+96>
```

2. Block 6:

```
0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc]
0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14]
0x0000059a <+77>: add eax,edx
0x0000059c <+79>: movzx eax,BYTE PTR [eax]
0x0000059f <+82>: movsx eax,al
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1
```

3. Block 7:

```
0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc]
0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18]
0x000005b3 <+102>: JL 0x594 <Main+71>
```

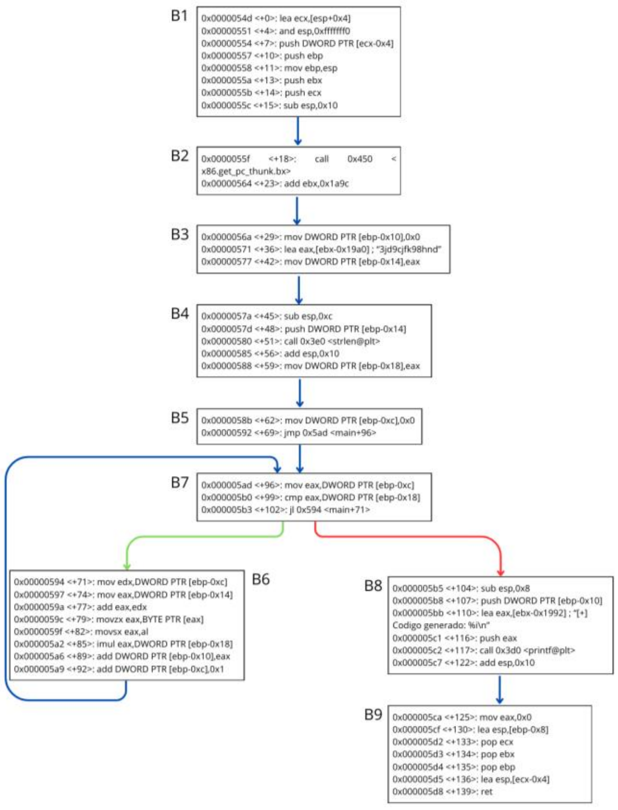
4. Block 8 + Block 9:

```
0x000005b5 <+104>: sub esp,0x8
0x000005b8 <+107>: push DWORD PTR [ebp-0x10]
0x000005bb <+110>: lea eax,[ebx-0x1992] ; “[+] Codigo generado: %i\n”
0x000005c1 <+116>: push eax
0x000005c2 <+117>: call 0x3d0 <printf@plt>
0x000005c7 <+122>: add esp,0x10
0x000005ca <+125>: mov eax,0x0
0x000005cf <+130>: lea esp,[ebp-0x8]
0x000005d2 <+133>: pop ecx
0x000005d3 <+134>: pop ebx
0x000005d4 <+135>: pop ebp
0x000005d5 <+136>: lea esp,[ecx-0x4]
0x000005d8 <+139>: p.
```

Flow chart. Is there a control structure? Indicates which *basic blocks* are involved in it.

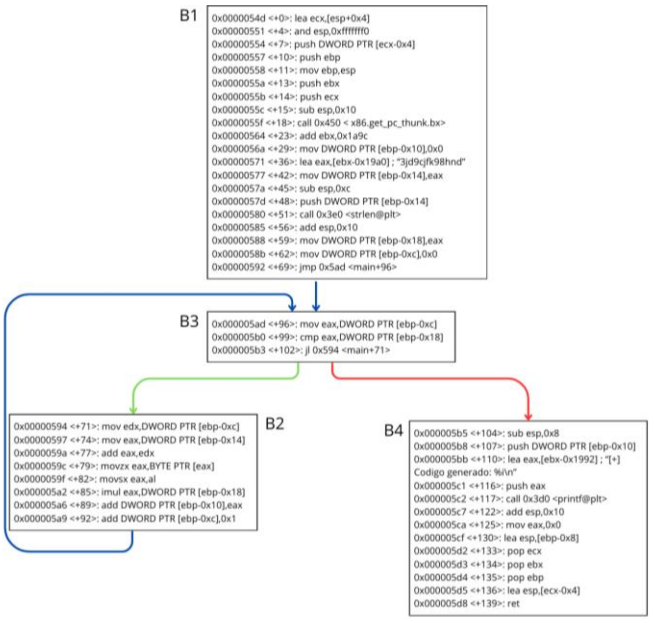
Two flow diagrams will be made, the first with a larger granulate of 9 blocks and a less granular one with 4 blocks.

- 9-block flowchart:



```
B1
0x0000054d <+0>: lea ecx,[esp+0x4]
0x00000551 <+4>: and esp,0xfffffff0
0x00000554 <+7>: push DWORD PTR [ecx-0x4]
0x00000557 <+10>: push ebp
0x00000558 <+11>: mov ebp,esp
0x0000055a <+13>: push ebx
0x0000055b <+14>: push ecx
0x0000055c <+15>: sub esp,0x10

B2
0x0000055f     <+18>:     call     0x450     <
x86.get_pc_thunk.bx>
0x00000564 <+23>: add ebx,0x1a9c

B3
0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax

B4
0x0000057a <+45>: sub esp,0xc
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
0x00000585 <+56>: add esp,0x10
0x00000588 <+59>: mov DWORD PTR [ebp-0x18],eax

B5
0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
0x00000592 <+69>: jmp 0x5ad <main+96>

B7
0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc]
0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18]
0x000005b3 <+102>: jl 0x594 <main+71>

B6
0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc]
0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14]
0x0000059a <+77>: add eax,edx
0x0000059c <+79>: movzx eax,BYTE PTR [eax]
0x0000059f <+82>: movsx eax,al
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1

B8
0x000005b5 <+104>: sub esp,0x8
0x000005b8 <+107>: push DWORD PTR [ebp-0x10]
0x000005bb <+110>: lea eax,[ebx-0x1992] ; "[+]
Codigo generado: %i\n"
0x000005c1 <+116>: push eax
0x000005c2 <+117>: call 0x3d0 <printf@plt>
0x000005c7 <+122>: add esp,0x10

B9
0x000005ca <+125>: mov eax,0x0
0x000005cf <+130>: lea esp,[ebp-0x8]
0x000005d2 <+133>: pop ecx
0x000005d3 <+134>: pop ebx
0x000005d4 <+135>: pop ebp
0x000005d5 <+136>: lea esp,[ecx-0x4]
0x000005d8 <+139>: ret
```

- 4-block flowchart:



```
B1
0x0000054d <+0>: lea ecx,[esp+0x4]
0x00000551 <+4>: and esp,0xfffffff0
0x00000554 <+7>: push DWORD PTR [ecx-0x4]
0x00000557 <+10>: push ebp
0x00000558 <+11>: mov ebp,esp
0x0000055a <+13>: push ebx
0x0000055b <+14>: push ecx
0x0000055c <+15>: sub esp,0x10
0x0000055f <+18>: call 0x450 < x86.get_pc_thunk.bx>
0x00000564 <+23>: add ebx,0x1a9c
0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax
0x0000057a <+45>: sub esp,0xc
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
0x00000585 <+56>: add esp,0x10
0x00000588 <+59>: mov DWORD PTR [ebp-0x18],eax
0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
0x00000592 <+69>: jmp 0x5ad <main+96>

B3
0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc]
0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18]
0x000005b3 <+102>: jl 0x594 <main+71>

B2
0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc]
0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14]
0x0000059a <+77>: add eax,edx
0x0000059c <+79>: movzx eax,BYTE PTR [eax]
0x0000059f <+82>: movsx eax,al
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1

B4
0x000005b5 <+104>: sub esp,0x8
0x000005b8 <+107>: push DWORD PTR [ebp-0x10]
0x000005bb <+110>: lea eax,[ebx-0x1992] ; "[+]
Codigo generado: %i\n"
0x000005c1 <+116>: push eax
0x000005c2 <+117>: call 0x3d0 <printf@plt>
0x000005c7 <+122>: add esp,0x10
0x000005ca <+125>: mov eax,0x0
0x000005cf <+130>: lea esp,[ebp-0x8]
0x000005d2 <+133>: pop ecx
0x000005d3 <+134>: pop ebx
0x000005d4 <+135>: pop ebp
0x000005d5 <+136>: lea esp,[ecx-0x4]
0x000005d8 <+139>: ret
```

Blue lines unconditional flow, green line conditional flow and is met and red line conditional flow and not met.

Yes, there is a control structure, in this case it is a '*for' or 'while'* (due to the structure of the code, since it first checks the conditional and then executes the loop) that iterates and compares with the length of the string of characters. The condition of which is that as long as the loop counter is less than the length of the chain, the loop will continue to iterate until it travels through the chain completely. Once it has been completely covered, jump to the '*printf*'.

This control structure is found in *basics blocks 5, 6 and 7* for the 9-block control flow and between *basics blocks 1, 2 and 3* in the 4-block control flow.

Conversion to C

We start with the libraries, as you make use of *printf*() it is necessary to include the *stdio.h* library and due to the use of the *strlen() function,* it is necessary to include the *string.h library*.

Once the issue of libraries has been resolved, we move on to defining the variables that appear in the object code:

```
0x0000054d <+0>: lea ecx,[esp+0x4]
 0x00000551 <+4>: and esp,0xfffffff0
0x00000554 <+7>: push DWORD PTR [ecx-0x4]
0x00000557 <+10>: push ebp
0x00000558 <+11>: mov ebp,esp
0x0000055a <+13>: push ebx
0x0000055b <+14>: push ecx
0x0000055c <+15>: sub esp,0x10
0x0000055f <+18>: call0x450 < x86.get_pc_thunk.bx>
0x00000564 <+23>: add ebx,0x1a9c
0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax
0x0000057a <+45>: sub esp,0xc
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
0x00000585 <+56>: add esp,0x10
0x00000588 <+59>: mov DWORD PTR [ebp-0x18],eax
0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
0x00000592 <+69>: jmp 0x5ad <main+96>
```

Then, in block 1, four local variables are initialized, *ebp-0x10, ebp-0x14, ebp-0x18 and ebp-0xc*. All of these are integers except *ebp-0x14* which is pointer to a character, in this case, it points to the memory address where the string *"3jd9cjfk98hnd" begins.* As for the other variables, in the case of *ebp-0xc,* it is nothing more than a counter initialized at 0 (*0x0*), used in the control structure, whose value increases one by one, as can be seen in the following block with *0x1*:

```
0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc]
0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14]
0x0000059a <+77>: add eax,edx
0x0000059c <+79>: movzx eax,BYTE PTR [eax]
0x0000059f <+82>: Movsx EAX,AL ; Conversion of character to integer
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax ; Iterative sum of integers
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1
```

As for the local variable *ebp-0x10* it is initialized with the null character and is used within the loop to add a series of terms with each iteration. These terms, as seen in the previous block, are nothing more than the conversion of a byte of the string of characters to the integer (ASCII value), as seen in the +82 line, multiplied by a constant *ebp-0x18* value which is the length of the string.

Finally, the *local variable ebp-0x18* is where the value of the string length is stored resulting in the *strlen() function,* lines from +48 to +56. This variable, as can be seen in the +85 line, is multiplied in each iteration by *eax,* which stores the value of the conversion to integer of each character in the string.

In addition, on line 79 you can access the character of the string with an *ebp-0xc index*.

Once the loop condition fails:

```
0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc]
0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18]
0x000005b3 <+102>: JL 0x594 <Main+71>
```

This condition is: *ebp-0xc (the counter)* < *ebp-0x18 (string length)*. Therefore, it will be false when the counter is greater than the length of the string, which means that it has already gone through it completely and calls the *printf* function as seen in the following block:

```
0x000005b5 <+104>: sub esp,0x8
0x000005b8 <+107>: push DWORD PTR [ebp-0x10]
0x000005bb <+110>: lea eax,[ebx-0x1992] ; "[+] Codigo generado: %i\n"
0x000005c1 <+116>: push eax
0x000005c2 <+117>: call 0x3d0 <printf@plt>
0x000005c7 <+122>: add esp,0x10
0x000005ca <+125>: mov eax,0x0
0x000005cf <+130>: lea esp,[ebp-0x8]
0x000005d2 <+133>: pop ecx
0x000005d3 <+134>: pop ebx
0x000005d4 <+135>: pop ebp
0x000005d5 <+136>: lea esp,[ecx-0x4]
0x000005d8 <+139>: p.
```

This *printf()* of the +117 line will show the text "Code generated:" and the value stored in the local variable, *ebp-0x10* as seen in the +110 and +107 lines respectively. Once the value is displayed, the cleaning and return begins.

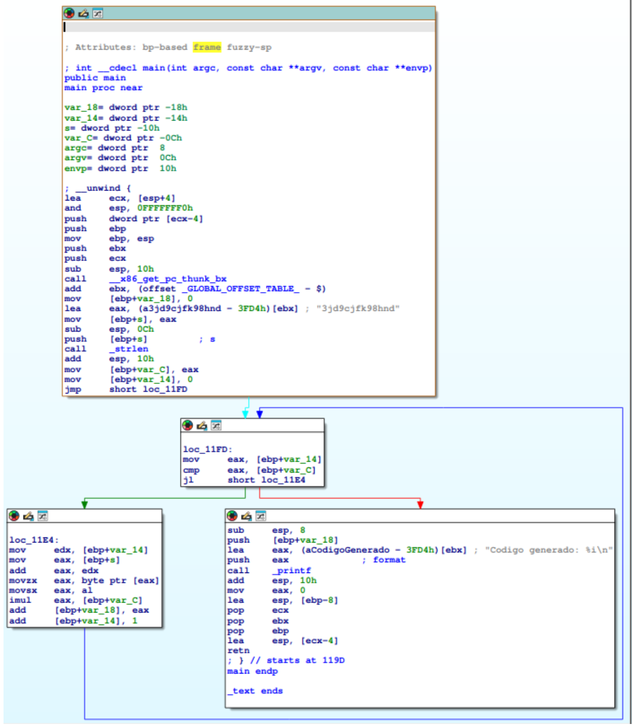
Thanks to the above analysis, it is possible to develop the program in C. As for the structure, for this case it is a bit indifferent if it is a *while* or a *for*, therefore, I will write two codes one with each structure. In order to check the result and compare the ASM code of both, I will use the *IDA Pro* program. Code in with *while*:

```
#include <stdio.h>
#include <string.h>


int main(){
  int sum = 0, i = 0;
  char *carac= "3jd9cjfk98hnd";
   int length = strlen(carac);

   while (i < longitud) {
     sum += (int)carac[i]*longitude;
      i++;
   }
   printf("Code generated: %i\n", sum);
   return 0;
}
```

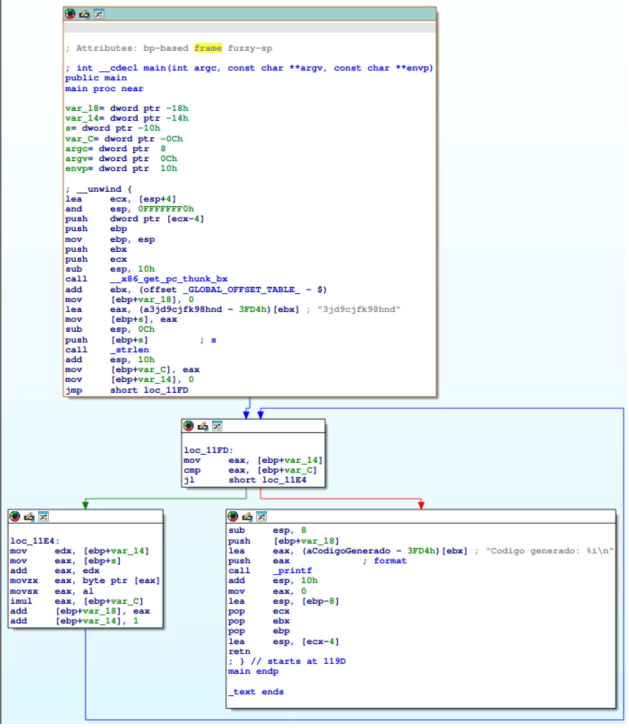
Code with the *For loop*:

```
#include <stdio.h>
#include <string.h>


int main(){
  int sum = 0, i;
  char *carac= "3jd9cjfk98hnd";
   int length = strlen(carac);

   for (i = 0; i < longitud; i++){
      sum += (int)carac[i]*longitude;
   }
   printf("Code generated: %i\n", sum);
   return 0;
}
```

Once we get the two executables (*gcc pro.c -or pro -m32*), we open them with *IDA pro*. The executable with the *for* loop is located on the right of the image and the executable with the *while* loop is located on the left:

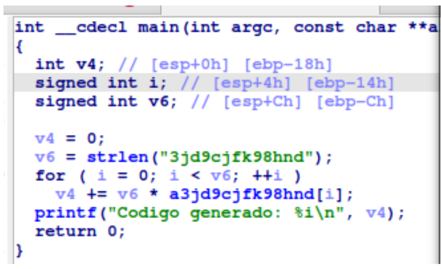


```
; Attributes: bp-based frame fuzzy-sp

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

var_18= dword ptr -18h
var_14= dword ptr -14h
s= dword ptr -10h
var_C= dword ptr -0Ch
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
sub     esp, 10h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     [ebp+var_18], 0
lea     eax, (a3jd9cjfk98hnd - 3FD4h)[ebx] ; "3jd9cjfk98hnd"
mov     [ebp+s], eax
sub     esp, 0Ch
push    [ebp+s]         ; s
call    _strlen
add     esp, 10h
mov     [ebp+var_C], eax
mov     [ebp+var_14], 0
jmp     short loc_11FD

loc_11FD:
mov     eax, [ebp+var_14]
cmp     eax, [ebp+var_C]
jl      short loc_11E4

loc_11E4:
mov     edx, [ebp+var_14]
mov     eax, [ebp+s]
add     eax, edx
movzx   eax, byte ptr [eax]
movsx   eax, al
imul    eax, [ebp+var_C]
add     [ebp+var_18], eax
add     [ebp+var_14], 1

sub     esp, 8
push    [ebp+var_18]
lea     eax, (aCodigoGenerado - 3FD4h)[ebx] ; "Codigo generado: %i\n"
push    eax             ; format
call    _printf
add     esp, 10h
mov     eax, 0
lea     esp, [ebp-8]
pop     ecx
pop     ebx
pop     ebp
lea     esp, [ecx-4]
retn
; } // starts at 119D
main endp

_text ends
```

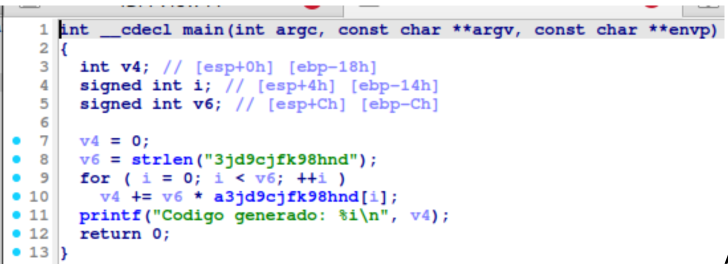


```
; Attributes: bp-based frame fuzzy-sp

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

var_18= dword ptr -18h
var_14= dword ptr -14h
s= dword ptr -10h
var_C= dword ptr -0Ch
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
sub     esp, 10h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     [ebp+var_18], 0
lea     eax, (a3jd9cjfk98hnd - 3FD4h)[ebx] ; "3jd9cjfk98hnd"
mov     [ebp+s], eax
sub     esp, 0Ch
push    [ebp+s]         ; s
call    _strlen
add     esp, 10h
mov     [ebp+var_C], eax
mov     [ebp+var_14], 0
jmp     short loc_11FD

loc_11FD:
mov     eax, [ebp+var_14]
cmp     eax, [ebp+var_C]
jl      short loc_11E4

loc_11E4:
mov     edx, [ebp+var_14]
mov     eax, [ebp+s]
add     eax, edx
movzx   eax, byte ptr [eax]
movsx   eax, al
imul    eax, [ebp+var_C]
add     [ebp+var_18], eax
add     [ebp+var_14], 1

sub     esp, 8
push    [ebp+var_18]
lea     eax, (aCodigoGenerado - 3FD4h)[ebx] ; "Codigo generado: %i\n"
push    eax             ; format
call    _printf
add     esp, 10h
mov     eax, 0
lea     esp, [ebp-8]
pop     ecx
pop     ebx
pop     ebp
lea     esp, [ecx-4]
retn
; } // starts at 119D
main endp

_text ends
```

No appreciable difference is found since the logic of both loops is the same and the compiler creates the same structure, therefore, the executable will be recompiled, but disabling the optimizations of the compiler, *gcc -o0 program.c -or program -m32*. Even with this consideration, we get the same diagram as in the previous image, so I am not going to attach it. As a last resort, we use the IDA *Pro* functionality to generate a pseudocode and we get the following (right *for* and left *while image*):



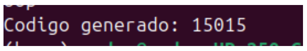
```
int __cdecl main(int argc, const char **a
{
  int v4; // [esp+0h] [ebp-18h]
  signed int i; // [esp+4h] [ebp-14h]
  signed int v6; // [esp+Ch] [ebp-Ch]

  v4 = 0;
  v6 = strlen("3jd9cjfk98hnd");
  for ( i = 0; i < v6; ++i )
    v4 += v6 * a3jd9cjfk98hnd[i];
  printf("Codigo generado: %i\n", v4);
  return 0;
}
```



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp+0h] [ebp-18h]
  signed int i; // [esp+4h] [ebp-14h]
  signed int v6; // [esp+Ch] [ebp-Ch]

  v4 = 0;
  v6 = strlen("3jd9cjfk98hnd");
  for ( i = 0; i < v6; ++i )
    v4 += v6 * a3jd9cjfk98hnd[i];
  printf("Codigo generado: %i\n", v4);
  return 0;
}
```

As you can see, not even *IDA Pro* can discern between while and *for* for this code. Therefore, it can be used interchangeably for the case that concerns us.

When compiling and running, both options show the same result:



```
Codigo generado: 15015
```

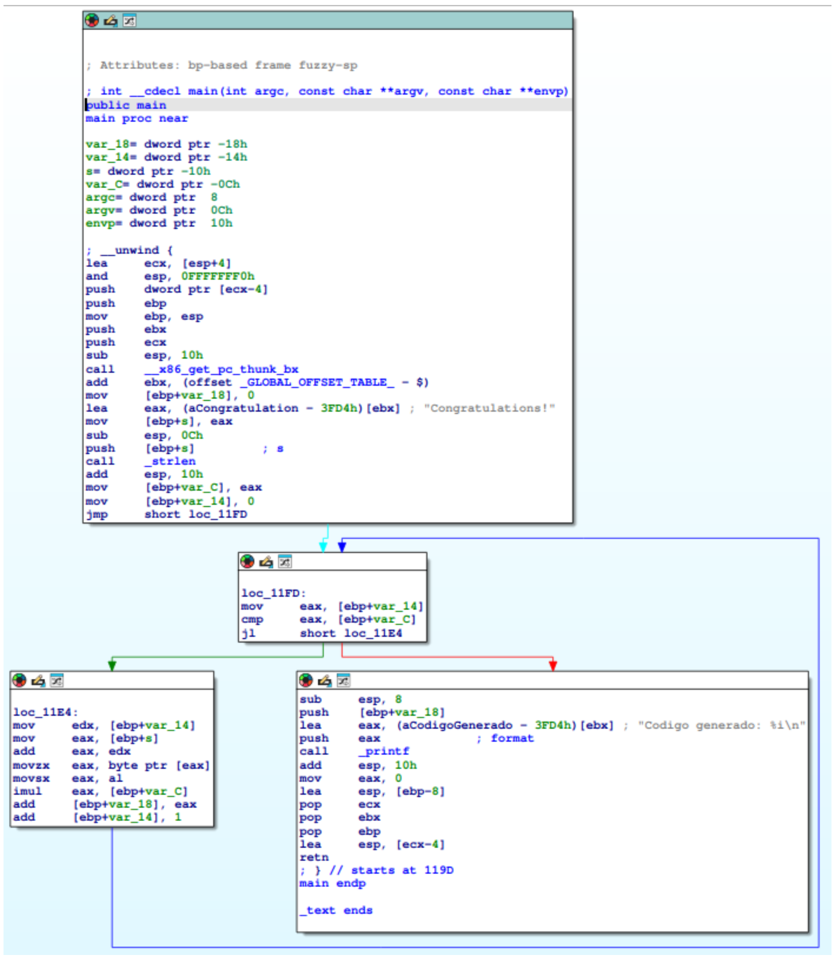
Modified Code

In this case, only the code with the for loop will be used, since we will get the same results. Therefore, in the C code, the string is modified to "*Congratulations!*" we compile it (with -m32) and execute, displaying, through *IDA Pro,* the object code obtained and the result of the execution of the program in the terminal.

Modified Code:

```c
#include <stdio.h>
#include <string.h>


int main(){
  int sum = 0, i;
  char *carac= " Congratulations! ";
  int length = strlen(carac);
  for (i = 0; i < longitud; i++){
       sum += (int)carac[i]*longitude;
  }
  printf("Code generated: %i\n", sum);
  return 0;
}
```

In *IDA Pro* you get the following object code, in diagram format:



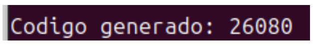
```
; Attributes: bp-based frame fuzzy-sp

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

var_18= dword ptr -18h
var_14= dword ptr -14h
s= dword ptr -10h
var_C= dword ptr -0Ch
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
sub     esp, 10h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     [ebp+var_18], 0
lea     eax, (aCongratulation - 3FD4h)[ebx] ; "Congratulations!"
mov     [ebp+s], eax
sub     esp, 0Ch
push    [ebp+s]         ; s
call    _strlen
add     esp, 10h
mov     [ebp+var_C], eax
mov     [ebp+var_14], 0
jmp     short loc_11FD
```

```
loc_11FD:
mov     eax, [ebp+var_14]
cmp     eax, [ebp+var_C]
jl      short loc_11E4
```

```
loc_11E4:
mov     edx, [ebp+var_14]
mov     eax, [ebp+s]
add     eax, edx
movzx   eax, byte ptr [eax]
movsx   eax, al
imul    eax, [ebp+var_C]
add     [ebp+var_18], eax
add     [ebp+var_14], 1
```

```
sub     esp, 8
push    [ebp+var_18]
lea     eax, (aCodigoGenerado - 3FD4h)[ebx] ; "Codigo generado: %i\n"
push    eax             ; format
call    _printf
add     esp, 10h
mov     eax, 0
lea     esp, [ebp-8]
pop     ecx
pop     ebx
pop     ebp
lea     esp, [ecx-4]
retn
; } // starts at 119D
main endp

_text ends
```

We run on terminal and we get:



```
Codigo generado: 26080
```