Informe de práctica de Ingeniería inversa (2025)

Autor: Pedro Oller Serrano

07/03/2025

Enunciado de la práctica

Se ha obtenido un fichero binario ejecutable que, tras su ejecución, muestra un texto con un código numérico. Este código se genera a partir de una cadena de texto guardada en el propio binario.

Tras listar el código ASM, el fichero binario se eliminó y no es posible acceder a él, solo al código ASM copiado al final del ejercicio.

Es necesario reutilizar dicho algoritmo de generación de códigos, por lo que se requieren labores de ingeniería inversa para analizar el binario y reconstruir el código fuente de tal forma que se pueda modificar y volver a compilar correctamente.

Para lleva a cabo esta tarea, se pide:

- **Dividir el código en *basic blocks*.**
- **Realizar un diagrama de flujo con los *basic blocks*. ¿Existe alguna estructura de control?, Indica qué *basic blocks* intervienen en ella.**
- **Convierte el código completo de la función en código C.**
- **Compila el código generado e indica el código resultante tras su ejecución.**
- **Modifica el código en C para que genere un nuevo código a partir de otra cadena.** Modifica la cadena <+*36*> en el código C por la siguiente cadena: *Congratulations!* Compila el código C, ejecuta e indica el texto completo obtenido.

Código ASM –x86 32 bits:

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>: ret

End of assembler dump.
```

Resolución de la práctica

División del código en *Basic Blocks*

Definición de *Basic Blocks,* son secuencias de instrucciones que no contienen saltos y que, además, no tiene puntos de entrada intermedios. Es decir, están separados por saltos y llamadas a funciones.

El código ASM –x86 32 bits, se puede dividir en 9 *basic blocks* para un alto granulado, como se desarrolla más adelante:

1. **Stack y regitros:** se ejecuta de forma secuencial, sin interrupciones y sin saltos.

```
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. **Obtención de *pc_thunk* y configuracón de registro:**

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

3. **Inicialización de las variables en la pila:** no hay saltos condicionales ni llamadas a funciones.

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

4. **Llamada a la función *‘strlen’:*** Se obtiene la longitud de la cadena.

```
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. **Inicialización del bucle:** salta a la etiquete *“main +96”* donde comprueba la condición del bucle.

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

6. **Bucle ‘for o while’:** Donde se procesa carácter a carácter la cadena *“3jd9cjfk98hnd”*.

```
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. **Condicional del bucle:** Si no cumple la condición del bucle "eax < DWORD PTR [ebp-0x18]". Es decir, si el contador es menor que la longitud de la cadena vuelve a la etiquete inicial *"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. **Ejecución de *"printf"*:** Si finaliza el bucle, se imprime por pantalla "Codigo generado: %i\n", mostrando el resultado.

```
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. **Limpieza y retorno de control:** En este bloque, se restaura la pila y se devuelve el control.

```
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
```

Realmente, si se requiere de menos granularidad, se podrían reducir estos 9 bloques a 4 sin afectar al flujo ya que se pueden agrupar bloques que no tienen bifurcaciones intermedias, como se observa:

1. **Bloque 1 + bloque 2 + bloque 3 + bloque 4 + bloque 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. **Bloque 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. **Bloque 7:**

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

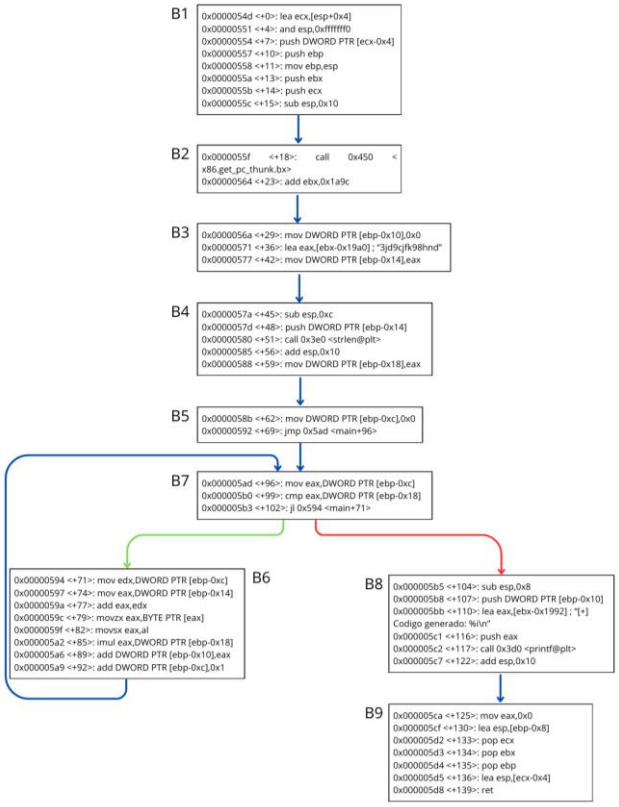
4. **Bloque 8 + bloque 9:**

```
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>: ret
```

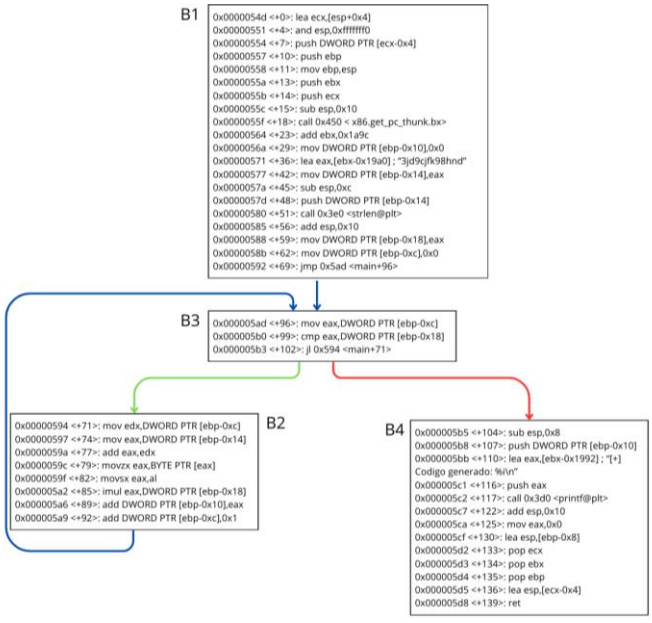
Diagrama de flujo. ¿Existe alguna estructura de control?, Indica qué *basic blocks* intervienen en ella.

Se van a realizar dos diagramas de flujo, el primero más granulado de 9 bloques y otro menos granulado de 4 bloques.

- Diagrama de flujo de 9 bloques:



- Diagrama de flujo de 4 bloques:



Líneas azules flujo incondicional, línea verde flujo condicional y se cumple y línea roja flujo condicional y no se cumple.

Sí, existe una estructura de control, en este caso es un *'for' o 'while'* (debido a la estructura del código, ya que primero comprueba el condicional y luego ejecuta el bucle) que va iterando y comparando con la longitud de la cadena de caracteres. Cuya condición es que mientras el contador del bucle sea menor que la longitud de la cadena, el bucle seguirá iterando hasta recorrer la cadena por completo. Una vez, se haya recorrido por completo, salta al '*printf*'.

Dicha estructura de control se encuentra en los *basics blocks 5, 6 y 7* para el flujo de control de 9 bloques y entre los *basics blocks 1, 2 y 3* en el flujo de control de 4 bloques.

Conversión a C

Comenzamos con las librerías, como se hace uso de *printf*() es necesario incluir la librería *stdio.h* y debido al uso de la función *strlen(),* es necesario incluir la librería *string.h*.

Una vez resuelto el tema de las librerías, pasamos a definir las variables que aparecen en el código objeto:

```
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>
```

Luego, en el bloque 1, se inicializan cuatro variables locales, *ebp-0x10, ebp-0x14, ebp-0x18 y ebp-0xc*. Todas estas son enteros exceptuando *ebp-0x14* que es puntero a un carácter, en este caso, apunta a la dirección de memoria donde comienza la cadena *"3jd9cjfk98hnd"*. En cuanto a las demás variables, en el caso de *ebp-0xc,* no es más que

un contador inicializado en 0 (*0x0*), usado en la estructura de control, cuyo valor va incrementando de uno en uno, como se observa en el siguiente bloque con *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 ; Conversión de caracter a entero
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax ; suma iterativa de los enteros
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1
```

En lo que respecta a la variable local *ebp-0x10* se inicializa con el carácter nulo y se emplea dentro del bucle para ir sumando una serie de términos con cada iteración. Estos términos, como se observa en el bloque anterior, no son más que la conversión de un byte de la cadena de caracteres al entero (valor ASCII), como se observa en la línea +82, multiplicados por un valor constante *ebp-0x18* que es la longitud de la cadena.

Finalmente, la variable local *ebp-0x18* es donde se almacena el valor de la longitud de la cadena que da como resultado la función *strlen(),* líneas de la +48 a la +56. Dicha variable, como se observa en la línea +85 es multiplicada en cada iteración por *eax,* que almacena el valor de la conversión a entero de cada carácter de la cadena.

Además, en la línea 79 se accede al carácter de la cadena con índice *ebp-0xc*.

Una vez falla la condición del bucle:

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

Esta condición es: *ebp-0xc (el contador)* < *ebp-0x18 (longitud de la cadena)*. Por lo que, será falsa cuando el contador sea mayor que la longitud de la cadena, lo que significa, que ya la ha recorrido por completo y llama a la función *printf* como se observa en el siguiente bloque:

```
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
```

Este *printf()* de la línea +117 mostrará el texto “Codigo generado:” y el valor almacenado en la variable local, *ebp-0x10* como se observa en las líneas +110 y +107 respectivamente. Una vez mostrada el valor, da comienzo la limpieza y retorno.

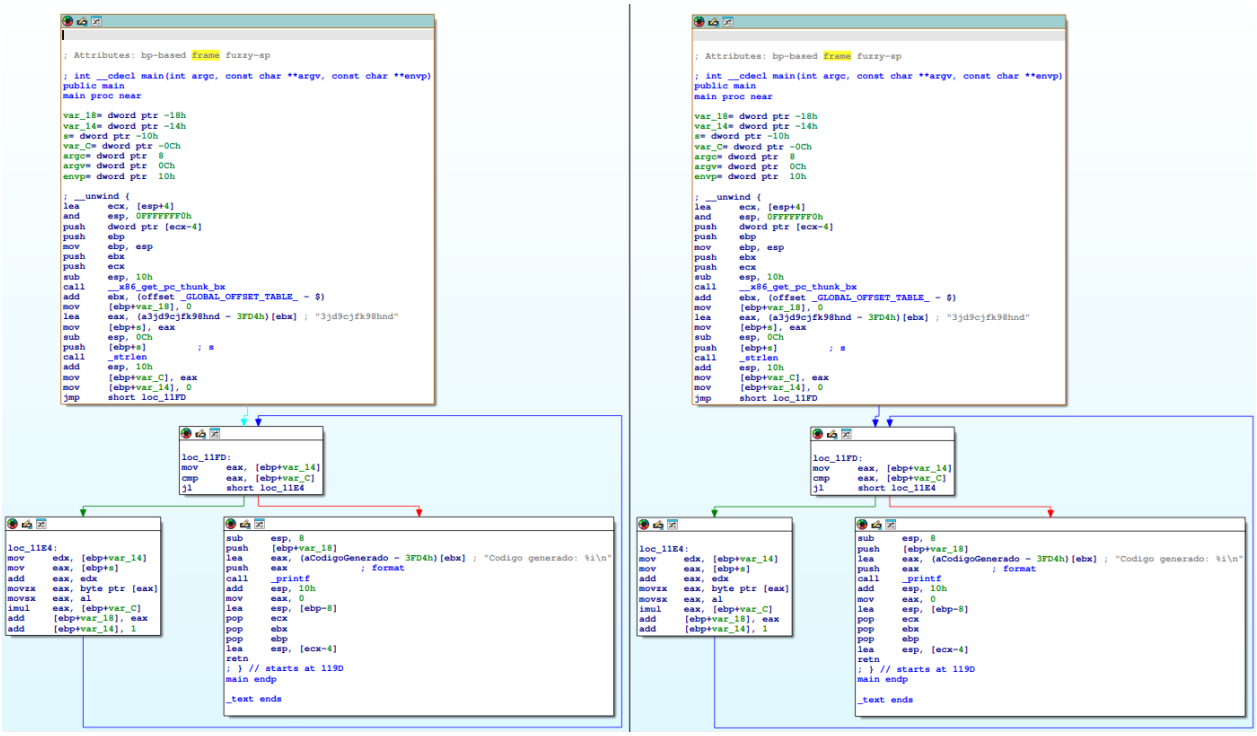
Gracias al análisis anterior, se consigue desarrollar el programa en C. En cuanto a la estructura, para este caso es un poco indiferente si es un *while* o un *for*, por lo tanto, redactaré dos códigos uno con cada estructura. Para poder comprobar el resultado y comparar el código ASM de ambos, emplearé el programa de *IDA Pro*. Código en con *while*:

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

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

   while (i < longitud) {
     suma += (int)carac[i]*longitud;
      i++;
   }
   printf("Codigo generado: %i\n", suma);
   return 0;
}
```

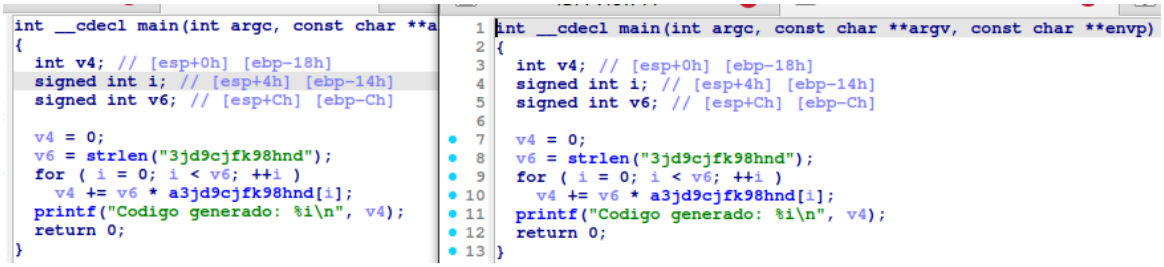
Código con el bucle *For*:

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

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

   for (i = 0; i < longitud; i++){
     suma += (int)carac[i]*longitud;
   }
   printf("Codigo generado: %i\n", suma);
   return 0;
}
```

Una vez obtenemos los dos ejecutables (*gcc pro.c -o pro -m32*), los abrimos con *IDA pro*. El ejecutable con el bucle *for* se encuentra a la derecha de la imagen y el ejecutable con el bucle *while* se encuentra a la izquierda:



No se encuentra alguna diferencia apreciable ya que la lógica de ambos bucles es la misma y el compilador crea la misma estructura, por ello, se volverá a compilar el ejecutable, pero desactivando las optimizaciones del compilador, *gcc -o0 programa.c -o programa -m32*. Aún con esta consideración, obtenemos el mismo diagrama que en la imagen anterior, por ello no la voy a adjuntar. Como último recurso, se emplea la funcionalidad de *IDA Pro* para generar un pseudocódigo y obtenemos lo siguiente (imagen derecha *for* e izquierda *while*):



```
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;
}
```

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    int v4; // [esp+0h] [ebp-18h]
4    signed int i; // [esp+4h] [ebp-14h]
5    signed int v6; // [esp+Ch] [ebp-Ch]
6
7    v4 = 0;
8    v6 = strlen("3jd9cjfk98hnd");
9    for ( i = 0; i < v6; ++i )
10     v4 += v6 * a3jd9cjfk98hnd[i];
11   printf("Codigo generado: %i\n", v4);
12   return 0;
13 }
```

Como se observa, ni siquiera *IDA Pro* puede discernir entre el *while* y el *for* para este código. Por lo que, se puede emplear indistintamente para el caso que nos atañe.

Al compilar y ejecutar, ambas opciones muestran el mismo resultado:

```
Codigo generado: 15015
```

Código modificado

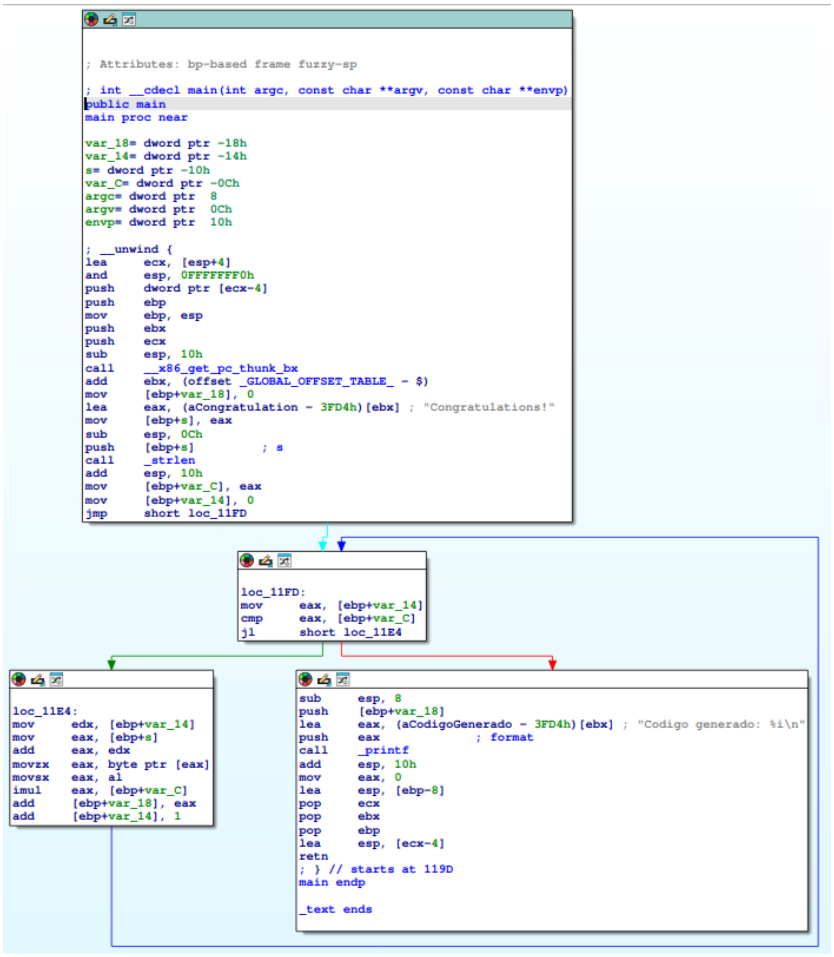
Para este caso, únicamente se empleará el código con el bucle *for*, ya que obtendremos los mismos resultados. Por ello, en el código de C, se modifica la cadena por "*Congratulations!*" lo compilamos (con -m32) y ejecutamos, mostrando, mediante

IDA Pro el código objeto obtenido y el resultado de la ejecución del programa en la terminal.

Código modificado:

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


int main(){
   int suma = 0, i;
   char *carac= " Congratulations! ";
   int longitud = strlen(carac);
   for (i = 0; i < longitud; i++){
        suma += (int)carac[i]*longitud;
   }
   printf("Codigo generado: %i\n", suma);
   return 0;
}
```

En *IDA Pro* se obtiene el siguiente código objeto, en formato de diagrama:



Ejecutamos en terminal y se obtiene:

```
Codigo generado: 26080
```