

M5. Ingeniería Inversa

Enunciado

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.

1. Divide el código en *basic blocks*: ten en cuenta los saltos existentes dentro de la función y divide las líneas en bloques básicos de código.

En primer lugar, se explicará el significado de basic block. El código en un bloque básico tiene:

- Un punto de entrada, lo que significa que ningún código dentro de él es el destino de una instrucción de salto en cualquier lugar del programa.
- Un punto de salida, lo que significa que sólo la última instrucción puede hacer que el programa comience a ejecutar código en un bloque básico diferente.

Cada vez que se ejecuta la primera instrucción en un bloque básico, el resto de las instrucciones necesariamente se ejecutan exactamente una vez y en orden.

Las instrucciones con las que se suele hacer un cambio de bloque básico son las siguientes:

call	Cambia el flujo de ejecución llamando a otra función
jmp	Salto incondicional, transfiere el control a otro lugar.
jl	Salto condicional, crea bifurcaciones en el flujo de control
ret	Finaliza la función y devuelve el control al llamador.
Punto de entrada	Marca el inicio de un nuevo camino alcanzable desde otra instrucción.

Teniendo en cuenta el significado de basic block, dado el código del enunciado se separará el código en basic blocks.

En primer lugar, **el primer bloque básico se usará para la inicialización de variables entre otras cosas.**

```
0x0000054d <+0>: lea ecx,[esp+0x4]
0x00000551 <+4>: and esp,0xffffffff
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>
```

En este bloque, configura el marco de la pila, así como reserva espacio para las variables locales con la instrucción `sub esp, 0x10`. En este caso la instrucción `"call0x450 <x86.get_pc_thunk.bx>"` no supondría un cambio de bloque ya que el código continua su flujo inmediatamente después de llamar a dicha función. Lo mismo pasaría con la instrucción `"call 0x3e0 <strlen@plt>"` al volver inmediatamente al flujo principal después de la llamada a la función, no sería necesario dividirlo en otro bloque básico. Dicho bloque termina con la instrucción `"jmp 0x5ad <main+96>"` que indica un salto incondicional y transfiere el control a otro lugar.

El segundo bloque básico se utilizará para separar 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>
```

En dicho bloque se incluye desde la instrucción +96 ya que en el anterior bloque la última instrucción indica un salto a dicha dirección. En este bloque se compara el valor de dos variables definidas previamente, y salta a la instrucción `<main+71>` si el valor comparado en la instrucción anterior es menor.

El tercer bloque básico se utilizará para identificar el cuerpo del bucle:

```
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
```

En este bloque se incluye en cuerpo del bucle donde se realizan una serie de operaciones mientras se cumpla la condición del bucle.

En el cuarto bloque básico se incluye la salida del bucle y la finalización de la función:

```
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] ; “[+] Código 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
```

En el último bloque se realiza una preparación para llamar la función printf, se reserva espacio en el stack para los parámetros que serán pasados a printf con la instrucción “sub esp,0x8” y posteriormente, se hace un push de los parámetros para printf en la instrucción “push DWORD PTR [ebp-0x10]” como valor del primer argumento y “push eax” como valor del segundo parámetro. Posteriormente se restaura el stack después de la llamada a printf con la instrucción “add esp,0x10”. Por último se hace una restauración del stack frame con las instrucciones de <+130> hasta <+135>.

2. Realiza el diagrama de flujo con los *basic blocks*: una vez obtenidos los *basic blocks* del ejercicio anterior, realiza el diagrama de flujo uniéndolos con flechas.

Para realizar este ejercicio se ha usado IDA free. En primer lugar, se han separado las instrucciones indicadas anteriormente en los bloques indicados en el anterior ejercicio:

```
; Comienzo de la función main
extern x80_get_pc_thunk_bx
extern strlen
extern printf

section .text
global _start

_start:
    lea ecx, [esp+0*4]
    and esp, 0*ffffff0
    push dword [ecx-0*4]
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    sub esp, 0*10

    ; Llamada a una función auxiliar (PC Thunk)
    call x80_get_pc_thunk_bx
    add ebx, 0*1a9c

    ; Inicialización de variables
    mov dword [ebp-0*10], 0*0
    lea eax, [ebx-0*19a0] ; "3jd9cjfk98hnd"
    mov dword [ebp-0*14], eax
    sub esp, 0*c
    push dword [ebp-0*14]
    call strlen ; Llamada a strlen
    add esp, 0*10
    mov dword [ebp-0*18], eax
    mov dword [ebp-0*c], 0*0

loop_condition:
    ; Comienzo del bucle
    mov eax, dword [ebp-0*c]
    cmp eax, dword [ebp-0*18]
    jl loop_body

exit_loop:
    ; Salida del bucle
    sub esp, 0*8
    push dword [ebp-0*10]
    lea eax, [ebx-0*1992] ; "[+] Código generado: %i\n"
    push eax
    call printf ; Llamada a printf
    add esp, 0*10
    mov eax, 0*0 ; Retorno exitoso

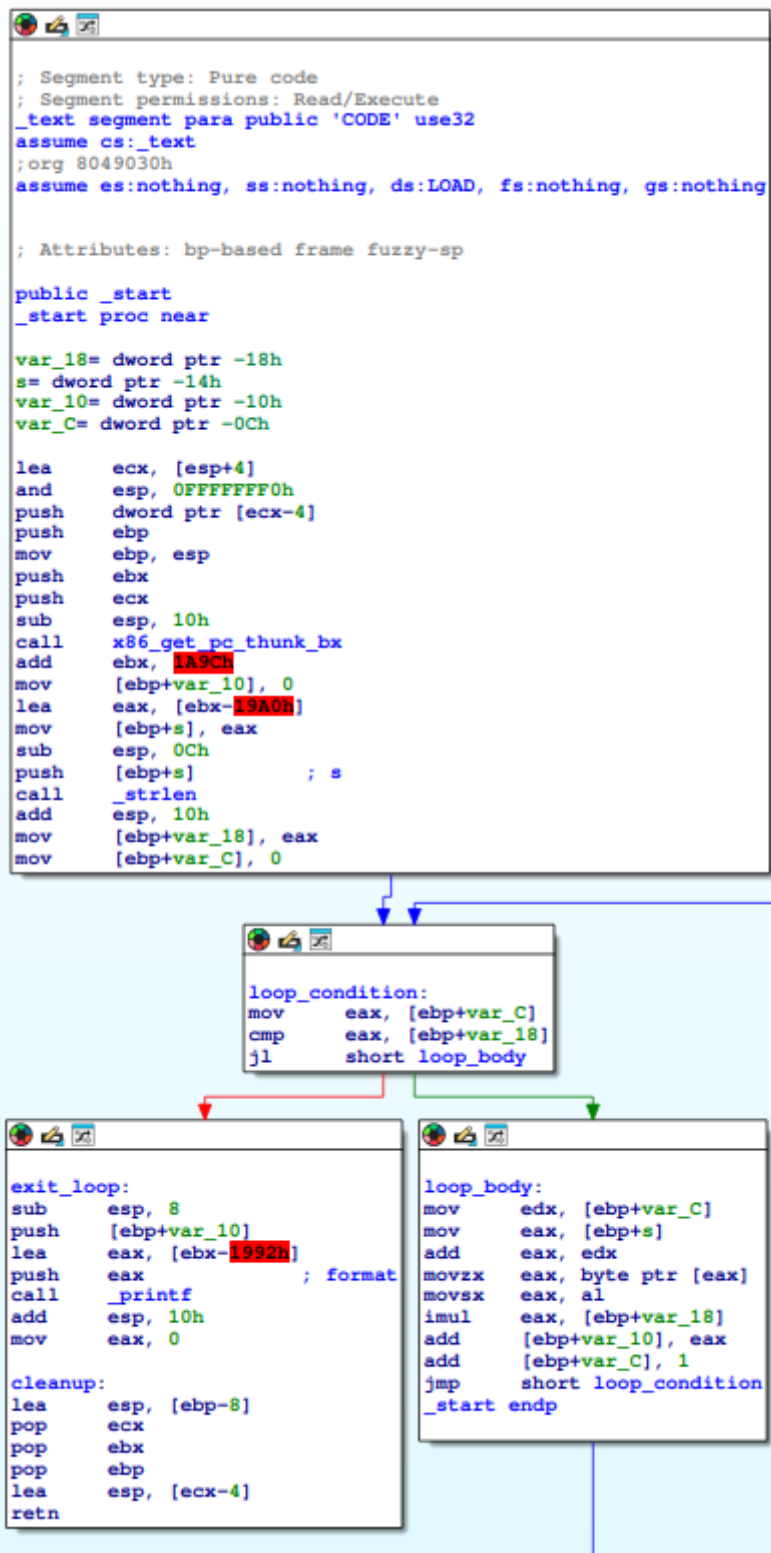
cleanup:
    ; Limpieza del stack frame
    lea esp, [ebp-0*8]
    pop ecx
    pop ebx
    pop ebp
    lea esp, [ecx-0*4]
    ret

loop_body:
    ; Cuerpo del bucle
    mov edx, dword [ebp-0*c]
    mov eax, dword [ebp-0*14]
    add eax, edx
    movzx eax, byte [eax]
    movsx eax, al
    imul eax, dword [ebp-0*18]
    add dword [ebp-0*10], eax
    add dword [ebp-0*c], 0*1
    jmp loop_condition

x80_get_pc_thunk_bx:
    mov ebx, [esp]
    ret
```

Posteriormente con el fichero indicado anteriormente ASM.asm, se compila y enlaza con “nasm -f elf32 -o main.o main.asm” y posteriormente se enlaza con la biblioteca estándar de C “ld -m elf_i386 -o main main.o -lc -dynamic-linker /lib/ld-linux.so.2”.

Después de realizar estos pasos, se importa el programa main en IDA y se muestra la vista donde aparecen los bloques básicos unidos por flechas.



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

Si existe una estructura de control como se ha indicado en la separación de bloques del primer ejercicio que se corresponde con un while. Intervienen distintos bloques básicos, el segundo bloque básico en el que se indica la condición del loop, el tercer bloque básico donde se incluye el cuerpo del bloque y por último el cuarto bloque que hace referencia a la salida del bucle.

4. Convierte el código completo de la función en código C: con lo aprendido sobre reconstrucción de código, convierte esta función *Main()* en código C.

Teniendo en cuenta el primer bloque básico indicado en el primer ejercicio donde se inicializan las variables:

- **Int var_1=0;**
 - 0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0 → 0 en la dirección de memoria [ebp-0x10]
- **const char *string = "3jd9cjfk98hnd;**
 - 0x00000571 <+36>: lea eax,[ebx-0x19a0] ; “3jd9cjfk98hnd” → calcula la dirección de memoria del string y lo guarda en eax
 - 0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax → se guarda el valor de eax en una variable local, es decir la dirección de memoria del string.
- **int length = strlen(string);**
 - 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

En este caso lo que hace es primero reservar espacio en el stack con la instrucción “sub esp,0xc”, posteriormente, empuja el puntero del string a la cima del stack, llama a la función strlen y posteriormente restaura el stack con “add esp,0x10”, por último, guarda el resultado en una variable.

- **int i=0;**
 - 0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
- **while (i < length)**
 - 0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc] → carga el índice i indicado anteriormente

- 0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18] → compara el índice i con la longitud de la cadena length. Ya que length es [ebp-0x18]
- 0x000005b3 <+102>: jl 0x594 <main+71> → si la variable i es menor que la longitud de la cadena vuelve al cuerpo del bucle.
- **var_1 += (int)string[i] * length;**
 - 0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc] → se guarda en edx el valor de [ebp-0xc] que es i
 - 0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14] → se guarda la dirección de memoria del string en eax
 - 0x0000059a <+77>: add eax,edx -> calcula la dirección del string[i]
 - 0x0000059c <+79>: movzx eax,BYTE PTR [eax] → carga el carácter string[i]
 - 0x0000059f <+82>: movsx eax,al → convierte el carácter string[i] a un entero con signo
 - 0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18] → multiplica el carácter por la longitud del string
 - 0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax → suma el resultado de la multiplicación con el acumulador var_1
 - 0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1 → incrementa el contador [ebp-0xc] en este caso i
- **printf("[+] Codigo generado: %d\n", var_1);**
 - 0x000005b5 <+104>: sub esp,0x8 → reserva el espacio para el stack
 - 0x000005b8 <+107>: push DWORD PTR [ebp-0x10] → empuja el valor del var_1
 - 0x000005bb <+110>: lea eax,[ebx-0x1992] ; "[+] Codigo generado: %i\n" → carga la dirección del string "[+] Codigo generado: %i\n"
 - 0x000005c1 <+116>: push eax → empuja el puntero al string de formato
 - 0x000005c2 <+117>: call 0x3d0 <printf@plt> → llama a la función printf con los dos valores indicador en las instrucciones <+110> y <+116>.
- **return 0;**
 - 0x000005ca <+125>: mov eax,0x0 → Establecer 0 en el registro EAX (valor de retorno)

Código C resultante:

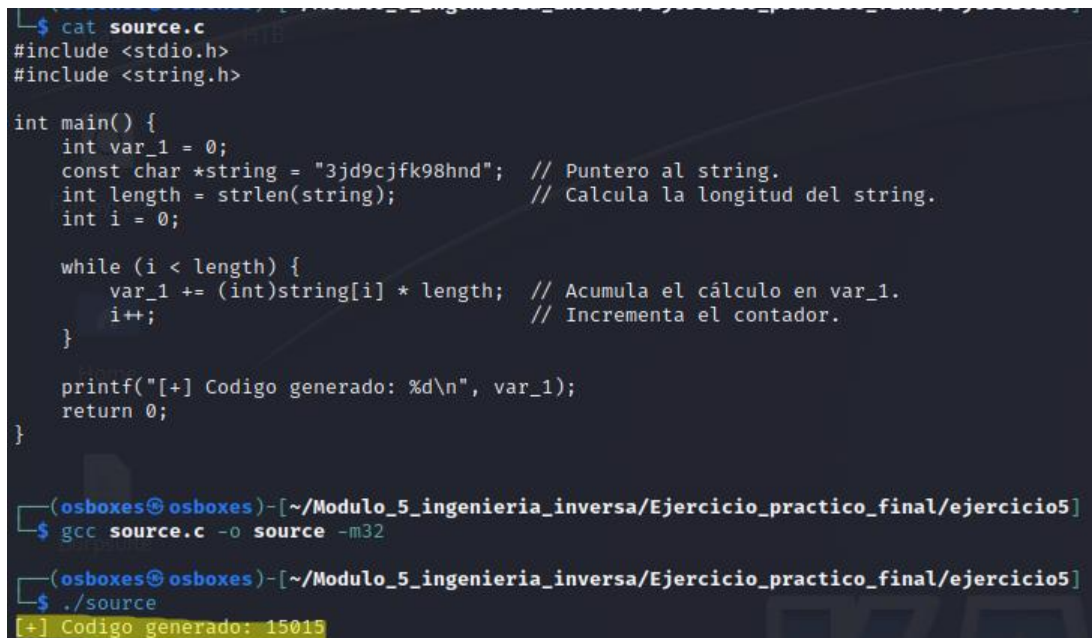
```
#include <stdio.h>
#include <string.h>

int main() {
    int var_1 = 0;
    const char *string = "3jd9cjfk98hnd"; // Puntero al string.
    int length = strlen(string);          // Calcula la longitud del string.
    int i = 0;

    while (i < length) {
        var_1 += (int)string[i] * length; // Acumula el cálculo en var_1.
        i++;                               // Incrementa el contador.
    }

    printf("[+] Código generado: %d\n", var_1);
    return 0;
}
```

5. Compila el código generado e indica el código resultante tras su ejecución: compila en 32 bits agregando la opción `-m32` tal y como se indica en el siguiente comando: `$ gcc source.c -o source -m32`. Una vez ejecutado, aparecerá un texto por pantalla. Indica el texto completo.



```
$ cat source.c
#include <stdio.h>
#include <string.h>

int main() {
    int var_1 = 0;
    const char *string = "3jd9cjfk98hnd"; // Puntero al string.
    int length = strlen(string);          // Calcula la longitud del string.
    int i = 0;

    while (i < length) {
        var_1 += (int)string[i] * length; // Acumula el cálculo en var_1.
        i++;                               // Incrementa el contador.
    }

    printf("[+] Código generado: %d\n", var_1);
    return 0;
}

(osboxes@osboxes)~/Modulo_5_ingenieria_inversa/Ejercicio_practico_final/ejercicio5
$ gcc source.c -o source -m32

(osboxes@osboxes)~/Modulo_5_ingenieria_inversa/Ejercicio_practico_final/ejercicio5
$ ./source
[+] Código generado: 15015
```

6. Modifica el código fuente 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.


```
$ cat source.c
#include <stdio.h>
#include <string.h>

int main() {
    int var_1 = 0;
    const char *string = "Congratulations!";
    int length = strlen(string);          // Calcula la longitud del string.
    int i = 0;

    while (i < length) {
        var_1 += (int)string[i] * length; // Acumula el cálculo en var_1.
        i++;                             // Incrementa el contador.
    }

    printf("[+] Código generado: %d\n", var_1);
    return 0;
}

(osboxes@osboxes)-[~/Modulo_5_ingenieria_inversa/Ejercicio_practico_final/ejercicio5]
$ gcc source.c -o source -m32

(osboxes@osboxes)-[~/Modulo_5_ingenieria_inversa/Ejercicio_practico_final/ejercicio5]
$ ./source
[+] Código generado: 26080
```