

Temario

- Estructura de un proceso en memoria
- La pila y las funciones
- Generación de shellcode

ESTRUCTURA DE UN PROCESO EN MEMORIA

Modelo Von Neumann



http://www.sites.upiicsa.ipn.mx/polilibros/porta/polilibros/p_terminados/PolilibroFC/Unidad_II/Unidad%20II_6.htm

@truerand0m

Estructura de un proceso en memoria

- Un ejecutable en Linux utiliza el formato ELF (*Executable and Linking Format*), el cual contiene secciones para indicar al cargador del SO como debe ser cargado en memoria. Una vez hecho, la disposición del proceso en memoria es la siguiente:

Estructura de un proceso en memoria

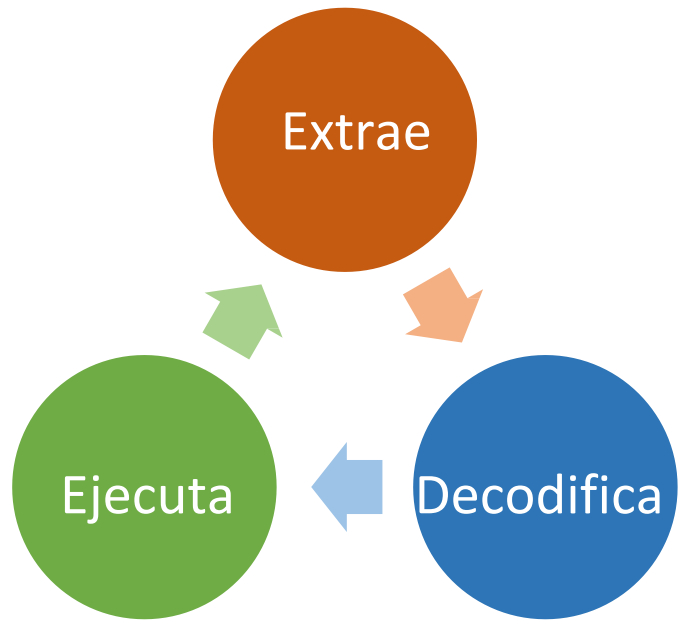
Segmentos:



- **Text** contiene las instrucciones a ejecutar.
- **Data** contiene las variables globales y estáticas inicializadas. (`static char* msg = "hola";`)
- **BSS** contiene variables globales y estáticas sin inicializar. (`char *str;`)
- **Heap** es espacio en memoria para variables de longitud dinámica (variables con `malloc()`).
- **Stack** es espacio en memoria para variables locales, argumentos y valores de registros.

Nota: la primera localidad de memoria dentro del segmento Text es conocida como punto de entrada o *AddressOfEntryPoint*.

Ciclo Fetch



- Extrae: El procesador obtiene la siguiente instrucción a ejecutar (apuntada por EIP). Al finalizar la extracción, se incrementa el valor de EIP.
- Decodifica: El procesador determina las operaciones que tiene que realizar.
- Ejecuta: El procesador desempeña las acciones determinadas previamente.

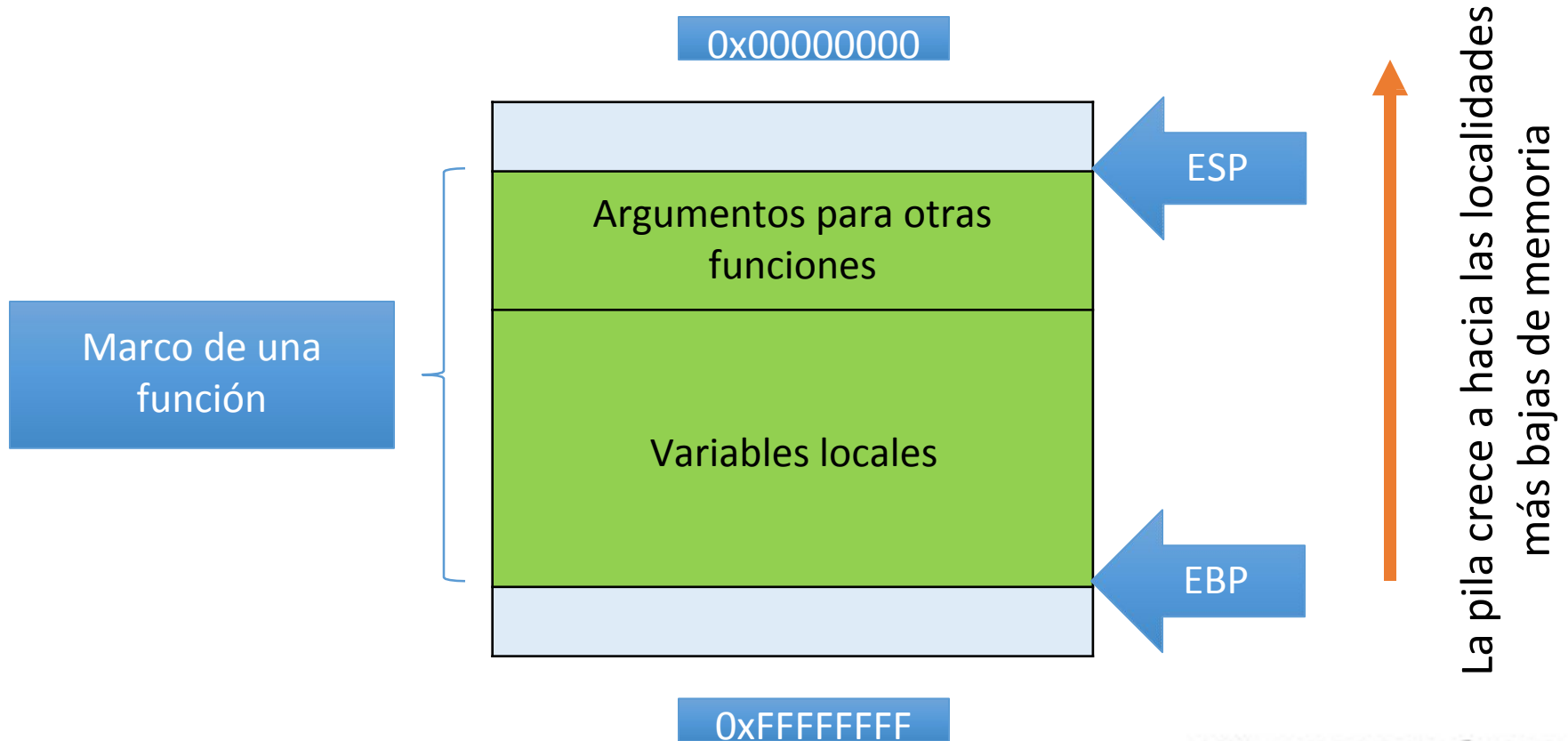
LA PILA Y LAS FUNCIONES

La pila y las funciones

- Cada vez que se llama una función, en la pila se crea un nuevo marco (*frame*).
- El propósito del marco de pila es contener las variables locales, y los argumentos para otras funciones.
- La creación del marco de pila es realizada por el prólogo, mientras que su destrucción es realizada por el epílogo.

La pila y las funciones

- El registro EBP apunta al inicio de la pila y ESP al final.



La pila y las funciones

```
//estructura_pila.c
// gcc estructura_pila.c -00
void funcion_B(int var){
    char arregloB[20];
}
void funcion_A(void){
    char arregloA[10];
    funcion_B(0x1234);
}

void main(void){
    funcion_A();
}
```

La pila y las funciones

```
void funcion_B(int var){  
    char arregloB[20];  
}  
  
void funcion_A(void){  
    char arregloA[10];  
    funcion_B(0x1234);  
}  
  
void main(void){  
    funcion_A();  
}
```

Marco de la función_B

Marco de la función_A

Marco de la función main

La pila y las funciones

- Compilar el archivo:estructura_pila.c

```
root@deb:~# gcc estructura_pila.c -o estructura_pila
root@deb:~# ls
estructura_pila  estructura_pila.c
root@deb:~# ./estructura_pila
root@deb:~# _
```

La pila y las funciones

- Desensamblar el ejecutable:

```
# objdump -d estructura_pila -M intel | less
```

```
080483cb <funcion_B>:
  80483cb:  55                push    ebp
  80483cc:  89 e5             mov     ebp,esp
  80483ce:  83 ec 20          sub     esp,0x20
  80483d1:  c9               leave   %ebp
  80483d2:  c3               ret

080483d3 <funcion_A>:
  80483d3:  55                push    ebp
  80483d4:  89 e5             mov     ebp,esp
  80483d6:  83 ec 10          sub     esp,0x10
  80483d9:  68 34 12 00 00    push    0x1234
  80483de:  e8 e8 ff ff ff    call    80483cb <funcion_B>
  80483e3:  83 c4 04          add     esp,0x4
  80483e6:  c9               leave   %ebp
  80483e7:  c3               ret

080483e8 <main>:
  80483e8:  55                push    ebp
  80483e9:  89 e5             mov     ebp,esp
  80483eb:  e8 e3 ff ff ff    call    80483d3 <funcion_A>
  80483f0:  5d                pop     ebp
  80483f1:  c3               ret
  80483f2:  55                push    ebp
  80483f3:  89 e5             mov     ebp,esp
  80483f5:  83 ec 10          sub     esp,0x10
  80483f8:  68 34 12 00 00    push    0x1234
  80483fd:  e8 e8 ff ff ff    call    80483cb <funcion_B>
  8048302:  83 c4 04          add     esp,0x4
  8048305:  c9               leave   %ebp
  8048306:  c3               ret
```

La pila y las funciones

Iniciar gdb para analizar la ejecución paso a paso.

```
# gdb estructura_pila -q
```

- Dentro de gdb, ejecutar los comandos:
 set disassembly-flavor Intel
 break *main

 run

```
root@deb:~# gdb estructura_pila -q
Reading symbols from estructura_pila...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) break *main
Breakpoint 1 at 0x80483e8
(gdb) run
Starting program: /root/estructura_pila

Breakpoint 1, 0x080483e8 in main ()
(gdb)
```

La pila y las funciones

- Explicación de las instrucciones

- `set disassembly-flavor intel`

Utiliza la sintaxis intel para el código desensamblado

- `break *main`

Establece un breakpoint en la dirección inicial de la función main

- `run`

Ejecuta el programa hasta encontrar un breakpoint, de lo contrario ejecuta el proceso completo.

La pila y las funciones

- Ejecutar el comando: `layout asm`

```

B+> 0x80483e8 <main>          push    ebp
      0x80483e9 <main+1>       mov     ebp,esp
      0x80483eb <main+3>       call    0x80483d3 <funcion_A>
      0x80483f0 <main+8>       pop     ebp
      0x80483f1 <main+9>       ret
      0x80483f2              xchg    ax,ax
      0x80483f4              xchg    ax,ax
      0x80483f6              xchg    ax,ax
      0x80483f8              xchg    ax,ax
      0x80483fa              xchg    ax,ax
      0x80483fc              xchg    ax,ax
      0x80483fe              xchg    ax,ax
      0x8048400 <__libc_csu_init> push    ebp
      0x8048401 <__libc_csu_init+1> push    edi
      0x8048402 <__libc_csu_init+2> xor     edi,edi
      0x8048404 <__libc_csu_init+4> push    esi
      0x8048405 <__libc_csu_init+5> push    ebx
      0x8048406 <__libc_csu_init+6> call    0x8048300 <__x86.get_pc_thunk.bx>
      0x804840b <__libc_csu_init+11> add     ebx,0x12a9
      0x8048411 <__libc_csu_init+17> sub     esp,0x1c
      0x8048414 <__libc_csu_init+20> mov     ebp,DWORD PTR [esp+0x30]
      0x8048418 <__libc_csu_init+24> lea     esi,[ebx-0xf4]

```

child process 25070 In: main

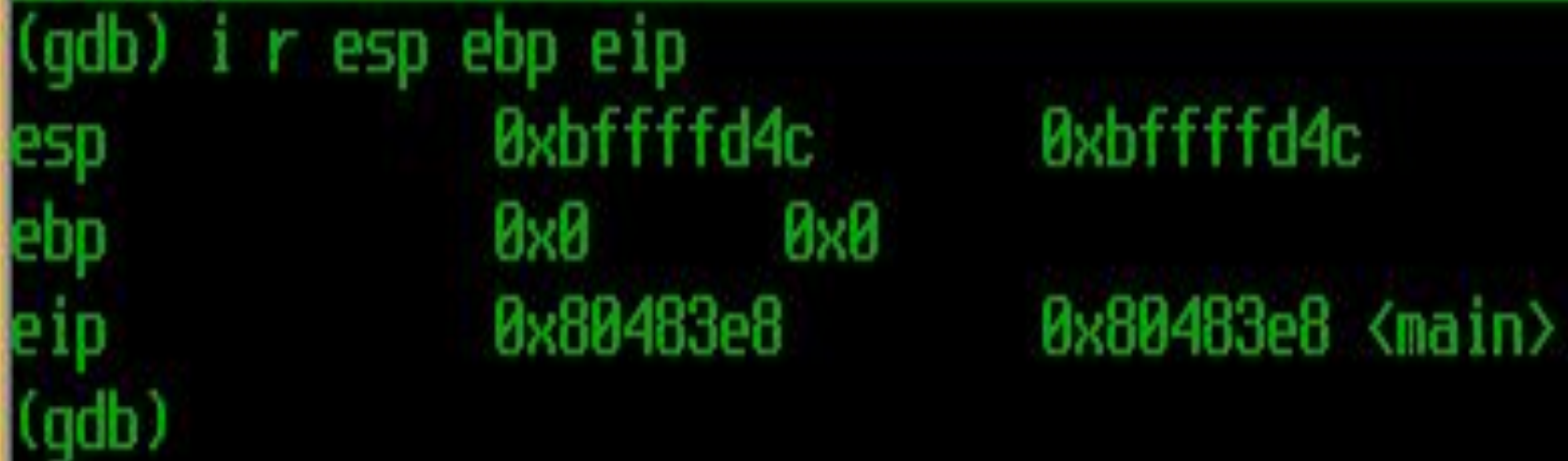
Line: ?? PC: 0x80483e8

(gdb) _

La pila y las funciones

- Observar los registros ESP, EBP y EIP, antes de ejecutar alguna instrucción de la función main, usar el comando:

i r esp ebp eip



```
(gdb) i r esp ebp eip
esp                0xbffffd4c      0xbffffd4c
ebp                0x0             0x0
eip                0x80483e8        0x80483e8 <main>
(gdb)
```

La pila y las funciones

- Ejecutar dos veces el comando `stepi` en `gdb` posteriormente revisar nuevamente los registros con: `i r esp ebp eip`

```
(gdb) stepi
0x080483e9 in main ()
(gdb) stepi
0x080483eb in main ()
(gdb) i r esp ebp eip
esp                0xbffffd48      0xbffffd48
ebp                0xbffffd48      0xbffffd48
eip                0x080483eb      0x080483eb <main+3>
(gdb) _
```

La pila y las funciones

```
B+ 0x80483e8 <main>      push    ebp
    0x80483e9 <main+1>    mov     ebp,esp
> 0x80483eb <main+3>    call   0x80483d3 <function_A>
    0x80483f0 <main+8>    pop     ebp
    0x80483f1 <main+9>    ret
```

- Las instrucciones PUSH y MOV constituyen el prologo para crear un nuevo marco en la pila.
- La primera instrucción guarda el apuntador base de la función que llamó a main.
- La segunda instrucción hace que el apuntador base del nuevo marco sea igual al valor de ESP, creando así el marco de main en la pila.

La pila y las funciones

- El prólogo se ejecuta cada vez que se llama una nueva función.
- Antes de ejecutar la siguiente instrucción, observar el valor de la instrucción después de CALL, es la dirección de retorno

```
B+ 0x80483e8 <main>      push    ebp
    0x80483e9 <main+1>    mov     ebp,esp
> 0x80483eb <main+3>    call   0x80483d3 <funcion_A>
    0x80483f0 <main+8>    pop     ebp
    0x80483f1 <main+9>    ret
```

La pila y las funciones

- La dirección de retorno, permite reanudar el flujo del programa al terminar de ejecutar completamente la función llamada (función_A).
- Como consecuencia ese valor es utilizado por la instrucción CALL, el cual ejecuta implícitamente un PUSH, almacenándolo para su posterior uso.

La pila y las funciones

- Ejecutar la siguiente instrucción de ensamblador con el comando `stepi` y verificar que el último valor en la pila sea la dirección en memoria de la instrucción después de CALL por medio de:

`x/x $esp`

```

B+  0x00483e8 <main>      push    ebp
    0x00483e9 <main+1>    mov     ebp,esp
    0x00483eb <main+3>    call    0x00483d3
    0x00483f0 <main+8>    pop     ebp
    0x00483f1 <main+9>    ret
    0x00483f2             xchg    ax,ax
    0x00483f4             xchg    ax,ax
    0x00483f6             xchg    ax,ax
    0x00483f8             xchg    ax,ax
    0x00483fa             xchg    ax,ax
    0x00483fc             xchg    ax,ax
    0x00483fe             xchg    ax,ax
    0x0048400 <__libc_csu_init> push    ebp
    0x0048401 <__libc_csu_init+1> push    edi

child process 25070 In: funcion_A
0x000483e9 in main ()
(gdb) stepi
0x000483eb in main ()
(gdb) i r esp ebp eip
esp             0xbffffd48      0xbffffd48
ebp             0xbffffd48      0xbffffd48
eip             0x00483eb        0x00483eb <main+3>
(gdb) stepi
0x000483d3 in funcion_A ()
(gdb) x/x $esp
0xbffffd44:      0x000483f0
  
```

La pila y las funciones

```
> 0x80483d3 <funcion_A>          push    ebp
0x80483d4 <funcion_A+1>        mov     ebp,esp
0x80483d6 <funcion_A+3>        sub     esp,0x10
0x80483d9 <funcion_A+6>        push    0x1234
0x80483de <funcion_A+11>       call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>       add     esp,0x4
0x80483e6 <funcion_A+19>       leave
0x80483e7 <funcion_A+20>       ret
```

- La instrucción CALL llamó a la función_A, por lo cual se debe crear un nuevo marco en la pila para almacenar el valor de las variables locales.
- Las dos primeras instrucciones se encargan de ello.

La pila y las funciones

```
> 0x80483d3 <funcion_A>      push    ebp
0x80483d4 <funcion_A+1>          mov     ebp,esp
0x80483d6 <funcion_A+3>          sub     esp,0x10
0x80483d9 <funcion_A+6>          push    0x1234
0x80483de <funcion_A+11>         call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>         add     esp,0x4
0x80483e6 <funcion_A+19>         leave
0x80483e7 <funcion_A+20>        ret
```

- La tercera instrucción se encarga de reservar espacio para una variable local, arreglo1[10], recordar que la pila crece hacia direcciones más pequeñas.

La pila y las funciones

```
> 0x80483d3 <funcion_A>      push    ebp
0x80483d4 <funcion_A+1>          mov     ebp,esp
0x80483d6 <funcion_A+3>          sub     esp,0x10
0x80483d9 <funcion_A+6>          push    0x1234
0x80483de <funcion_A+11>         call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>         add     esp,0x4
0x80483e6 <funcion_A+19>         leave
0x80483e7 <funcion_A+20>        ret
```

- La cuarta instrucción se encarga de almacenar en la pila el argumento (0x1234) de la función_B, mediante la instrucción PUSH.

La pila y las funciones

- Teclear 4 veces el comando `stepi` para ejecutar las siguientes instrucciones de ensamblador :
 - `PUSH EBP`
 - `MOV EBP,ESP`
 - `SUB ESP, 0X10`
 - `PUSH 0x1234`
- Posteriormente ejecutar: `i r esp ebp`
- Obtener el contenido de la pila con: `x/20x $esp`

```
(gdb) i r esp ebp
esp      0xbffffd2c      0xbffffd2c
ebp      0xbffffd40      0xbffffd40
(gdb) x/20x $esp
0xbffffd2c:  0x00001234      0xb7fd13c4      0xb7fff000      0x0004040b
0xbffffd3c:  0xb7fd1000      0xbffffd48      0x000403f0      0x00000000
0xbffffd4c:  0xb7e41a63      0x00000001      0xbffffde4      0xbffffdec
0xbffffd5c:  0xb7fed79a      0x00000001      0xbffffde4      0xbffffd84
0xbffffd6c:  0x000496c4      0x000401ec      0xb7fd1000      0x00000000
(gdb) _
```

La pila y las funciones

El resultado al analizar los registros esp y ebp, así como de analizar el contenido de la pila es el siguiente:

- `PUSH EBP` ;Se almacena el apuntador base del marco anterior
- `MOV EBP,ESP` ;Se crea un nuevo marco en la pila para función_B
- `SUB ESP, 0X10` ;Se reserva espacio para la variable arreglo1[10] (16 bytes)
- `PUSH 0x1234` ;Se guarda en la pila el argumento de la función_B

```
(gdb) i r esp ebp
esp      0xbffffd2c    0xbffffd2c
ebp      0xbffffd40    0xbffffd40
(gdb) x/20x $esp
0xbffffd2c: 0x00001234  0xb7fd13c4  0xb7fff000  0x0004040b
0xbffffd3c: 0xb7fd1000  0xbffffd40  0x000403f0  0x00000000
0xbffffd4c: 0xb7e41a63  0x00000001  0xbffffde4  0xbffffdec
0xbffffd5c: 0xb7fed79a  0x00000001  0xbffffde4  0xbffffd84
0xbffffd6c: 0x000496c4  0x000401ec  0xb7fd1000  0x00000000
(gdb) _
```

La pila y las funciones

- Antes de ejecutar la siguiente instrucción de ensamblador, anotar el valor de la instrucción posterior a CALL (dirección de retorno).

```
> 0x80483de <funcion_A+11>      call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>      add     esp,0x4
```

- Ejecutar la instrucción CALL por medio de: `stepi`
- Obtener el contenido de la pila

`x/20x $esp`

```
(gdb) stepi
0x080483cb in funcion_ ()
(gdb) x/20x $esp
0xbffffd28: 0x080483e3 0x00001234 0xb7fd13c4 0xb7fff000
0xbffffd38: 0x0804840b 0xb7fd1000 0xbffffd48 0x080483f0
0xbffffd48: 0x00000000 0xb7e41a63 0x00000001 0xbffffde4
0xbffffd58: 0xbffffdec 0xb7fed79a 0x00000001 0xbffffde4
0xbffffd68: 0xbffffd84 0x080496c4 0x080481ec 0xb7fd1000
(gdb)
```

La pila y las funciones

- Al analizar el código de función_B, se observa que al tratarse de una nueva función, se debe crear un nuevo marco en la pila.
- Las primeras 2 instrucciones se ocupan de crear el marco para las variables locales.
- La tercera instrucción reserva espacio para la variable arreglo2[20] (32 bytes).

> 0x80483cb	<function_B>	push ebp
0x80483cc	<function_B+1>	mov ebp,esp
0x80483ce	<function_B+3>	sub esp,0x20
0x80483d1	<function_B+6>	leave
0x80483d2	<function_B+7>	ret

La pila y las funciones

- Ejecutar 3 instrucciones más con `stepi`
- Ahora se muestran las instrucciones **LEAVE** y **RET**, las cuales componen el epílogo de la función.
- El propósito del **epílogo** para este caso es eliminar el marco de la función_B y regresar el control a la función_A, en la instrucción siguiente a `CALL`.

La pila y las funciones

- LEAVE internamente realiza dos acciones:
 MOV ESP,EBP ; Se elimina el marco de la función_B
 POP EBP ; Se retoma el marco de función_A
- RET realiza lo siguiente(internamente) :
 - POP EIP; Se reanuda el flujo de función_A

La pila y las funciones

- Antes de ejecutar la instrucción LEAVE, obtener los valores de ESP y EBP.

```
i r esp ebp
```

- Obtener el contenido de la pila.

```
x/20x $esp
```

- Tomar una captura de pantalla

```
(gdb) i r esp ebp
esp          0xbffffd04      0xbffffd04
ebp          0xbffffd24      0xbffffd24
(gdb) x/20x $esp
0xbffffd04:  0x00c10000      0x00000001      0x0804827d      0xbffffee6
0xbffffd14:  0x0000002f      0x080496b4      0x08048452      0x00000001
0xbffffd24:  0xbffffd40      0x080483e3      0x00001234      0xb7fd13c4
0xbffffd34:  0xb7fff000      0x0804840b      0xb7fd1000      0xbffffd48
0xbffffd44:  0x080483f0      0x00000000      0xb7e41a63      0x00000001
(gdb) _
```


La pila y las funciones

- Ejecutar LEAVE por medio de `stepi`.
- Obtener el valor de los registros ESP y EBP.
`i r esp ebp`
- Obtener el contenido de la pila.
`x/20x $esp`

```
(gdb) i r esp ebp
esp                0xbffffd28      0xbffffd28
ebp                0xbffffd40      0xbffffd40
(gdb) x/20x $esp
0xbffffd28:      0x080483e3      0x00001234      0xb7fd13c4      0xb7fff000
0xbffffd38:      0x0804840b      0xb7fd1000      0xbffffd48      0x080483f0
0xbffffd48:      0x00000000      0xb7e41a63      0x00000001      0xbffffde4
0xbffffd58:      0xbffffdec      0xb7fed79a      0x00000001      0xbffffde4
0xbffffd68:      0xbffffd84      0x080496c4      0x080481ec      0xb7fd1000
(gdb)
```

La pila y las funciones

- Entre los cambios que se pueden notar están:
 - Se están utilizando direcciones en memoria más altas dentro de la pila
 - Desasignación del espacio para la `variable arreglo2[20] (32 bytes)`
 - $ESP = EBP + 4$
 - $EBP =$ al EBP de la función `_A`, almacenado previamente en la pila.

```
(gdb) x/20x $esp
0xbffffd04
0xbffffd14
0xbffffd24
0xbffffd34
0xbffffd44
0x00c10000
0x0000002f
0xbffffd40
0xb7fff000
0x080483f0
0x00000001
0x080496b4
0x080483e3
0x0804840b
0x00000000
0x0804827d
0x08048452
0x00001234
0xb7fd1000
0xb7e41a63
0xbffffee6
0x00000001
0xb7fd13c4
0xbffffd48
0x00000001
```

La pila y las funciones

- Ejecutar la instrucción RET (stepi).
- Obtener el valor del registro EIP y compararlo con el valor anotado previamente.

i r eip

```
(gdb) stepi
0x080483e3 in funcion_A ()
(gdb) i r eip
eip                0x080483e3          0x080483e3 <funcion_A+16>
(gdb) _
```

La pila y las funciones

- El control ha regresado a la función_A, donde ahora se procede a la ejecución de la instrucción ADD ESP, 0x04.

```
0x80483d3 <funcion_A>      push    ebp
0x80483d4 <funcion_A+1>      mov     ebp,esp
0x80483d6 <funcion_A+3>      sub     esp,0x10
0x80483d9 <funcion_A+6>      push    0x1234
0x80483de <funcion_A+11>     call    0x80483cb <funcion_B>
> 0x80483e3 <funcion_A+16>  add     esp,0x4
0x80483e6 <funcion_A+19>     leave
0x80483e7 <funcion_A+20>     ret
```

- Dicha instrucción se encarga de eliminar el espacio reservado para el argumento de la función_B (4 bytes), el valor hexadecimal 0x1234.

La pila y las funciones

- Para comprobar la desasignación del espacio ocupado por el argumento mostrar el contenido de la pila:

`x/8x $esp`

- Ejecutar la instrucción de ensamblador ADD por medio de `stepi`.
- Volver a obtener el contenido de la pila:

`x/8x $esp`

La pila y las funciones

```
(gdb) x/8x $esp
0xbffffd2c:    0x00001234      0xb7fd13c4      0xb7fff000      0x0804840b
0xbffffd3c:    0xb7fd1000      0xbffffd48      0x080483f0      0x00000000
(gdb) stepi
0x080483e6 in funcion_A ()
(gdb) x/8x $esp
0xbffffd30:    0xb7fd13c4      0xb7fff000      0x0804840b      0xb7fd1000
0xbffffd40:    0xbffffd48      0x080483f0      0x00000000      0xb7e41a63
```

La pila y las funciones

- Luego de la instrucción ADD, se encuentra el epílogo de la función_A.

```
0x80483cb <funcion_B>      push    ebp
0x80483cc <funcion_B+1>      mov     ebp,esp
0x80483ce <funcion_B+3>      sub     esp,0x20
0x80483d1 <funcion_B+6>      leave
0x80483d2 <funcion_B+7>      ret
0x80483d3 <funcion_A>       push    ebp
0x80483d4 <funcion_A+1>      mov     ebp,esp
0x80483d6 <funcion_A+3>      sub     esp,0x10
0x80483d9 <funcion_A+6>      push    0x1234
0x80483de <funcion_A+11>     call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>     add     esp,0x4
0x80483e6 <funcion_A+19>     leave
0x80483e7 <funcion_A+20>     ret
0x80483e8 <main>              push    ebp
0x80483e9 <main+1>             mov     ebp,esp
0x80483eb <main+3>             call    0x80483d3 <funcion_A>
0x80483f0 <main+8>             pop     ebp
0x80483f1 <main+9>             ret
```

La pila y las funciones

- A partir de este punto, el programa no realizará ninguna operación adicional significativa, por lo cual el proceso se limitará a terminar la función_A y la función main, ejecutando los epílogos de ambas funciones y terminando la ejecución del programa.

La pila y las funciones

- Dentro de la interfaz de gdb, permitir la ejecución del programa hasta el final con el comando `continue` o únicamente la letra `C`.

```
(gdb) c
Continuing.
[Inferior 1 (process 25591) exited with code 01]
(gdb)
```