Ejecución de procesos

José María Alonso Cebrián
Jordi Gay Sensat
Antonio Guzmán Sacristán
Pedro Laguna Durán
Alejandro Martín Bailón
Jordi Serra Ruiz

PID_00146640

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Durante la ejecución de un proceso se suceden una serie de acciones que el sistema debe ir controlando para la correcta finalización del programa. Durante la ejecución de este, el sistema tiene que dar acceso a la CPU a la aplicación, acceso a los dispositivos, gestionar las autorizaciones, gestionar la memoria asignada al proceso, gestionar los archivos accedidos...

Durante la ejecución de un programa se crean y utilizan diferentes zonas de memoria para guardar cada uno de los componentes del programa que serán necesarios para su ejecución. Estas zonas de memoria son las siguientes:

- Código.
- Área de datos dinámicos (*heap*).
- Área de datos estáticos (*bss*).
- Pila (*stack*).

1. Código

Cuando empieza la ejecución de un programa, el sistema operativo carga el programa en una zona de memoria libre. Además se crean las zonas de datos enumeradas anteriormente en otras zonas de memoria que serán asociadas al programa en ejecución.

Normalmente, la zona de código tiene un tamaño prefijado. La zona de datos estáticos tampoco crece pues está predeterminada en tiempo de compilación. Las dos áreas que pueden crecer son el área de datos dinámicos y la pila. La zona de datos dinámicos crece hacia direcciones más altas de memoria y la pila hacia direcciones más bajas. Estas áreas deben posicionarse en memoria de manera adecuada, para que no terminen solapándose.

En el inicio del programa se prepara el entorno para que pueda empezar a ejecutarse la primera instrucción del programa. Esto se consigue apuntando el registro EIP del procesador (encargado de apuntar a la instrucción a ejecutar) a la dirección de entrada del programa cargado (*entry point* o primera instrucción a ejecutar). También es necesario inicializar adecuadamente los apuntadores a la pila y las áreas de memoria.

2. Área de datos dinámicos (*heap*)

La asignación dinámica de memoria es el proceso por el que el sistema asigna zonas de memoria libres a distintas partes del programa ejecutado. Estas requieren zonas de memoria libre durante la ejecución del programa sin haber hecho una declaración explícita de tamaño en el código fuente, para poder realizar su cometido. Cada lenguaje tiene sus propias funcionas para pedir memoria al sistema. Por ejemplo, en C se dispone de `malloc`.

```
char *buf = malloc(SIZE)
```

Esta línea reservaría un espacio de memoria de tamaño `SIZE` que seria accedido a través del puntero asignado a la variable `buf`. Para poder liberar ese espacio de memoria se debería utilizar la función `free`.

```
free(buf)
```

Otros lenguajes de programación disponen de otras funciones para realizar este cometido. Una vez se ha asignado una porción de memoria a un programa, esta permanece reservada hasta que el programa la libera explícitamente. Posibles errores en la programación de programas pueden provocar que la memoria reservada no se libere (*memory leak*), quedando así el sistema con partes de memoria que no se utilizarían, ya que estarían reservados. Esto podría provocar que los sistemas, después de un cierto periodo de tiempo, quedaran inutilizados por no disponer de memoria libre, debido a que estaría toda "usada" por los diferentes procesos del sistema.

Para evitar este problema, los sistemas operativos cuentan con procesos que se dedican a buscar partes de memoria reservadas que no tengan ninguna referencia desde ninguno de los programas en ejecución. Estos procesos se denominan *garbage collectors*. En el caso de encontrar espacios de memoria que no están siendo utilizados por ningún proceso, se procede a liberarlos. Cada sistema operativo, o incluso lenguaje de programación (lenguajes interpretados como C# o Java), tiene su propia manera de gestionar la memoria.

3. Área de datos estáticos (*bss*)

En esta área se guardan todos los datos que son conocidos en tiempo de compilación. Habitualmente son las variables globales, las variables definidas como *static* y las constantes. Podría decirse que se guardan en ese espacio datos que tienen que ser accesibles varias veces durante la ejecución del programa y no sólo durante la ejecución de una función.

```
static char buf[SIZE]
```

La sentencia anterior define un *array* de caracteres de manera estática, con lo que se reservaría espacio en memoria para él en la zona *bss*.

Las variables locales, no definidas como *static*, se guardan en la pila. No necesitan ser accedidas después de la ejecución de la función, por lo que pueden ser destruidas completamente perdiendo sus valores y definición. La pila es el lugar perfecto para crear este comportamiento durante la ejecución de un programa.

A veces se denomina a esta zona de memoria como *bss* (*Block Started by Symbol*). Esta nomenclatura tiene sus orígenes en una pseudo-operación en el FORTRAN *Assembly Program*, desarrollado por IBM. Posteriormente esta nomenclatura se fue heredando entre sistemas hasta nuestros días.

Los *exploits* que explotan errores en el acceso a esta área de memoria se denominan *bss-based overflows*. Si por el contrario explotan vulnerabilidades en el acceso al área de memoria del *heap* se denominan *heap-based overflows*. Como la manera de explotar ambas regiones de memoria se asemeja, muchas veces se juntan en un solo nombre: *heap/bss-based overflows*.

4. Pila

La pila es una zona de memoria reservada para guardar todo tipo de datos temporales de manera ordenada, manteniendo la coherencia de los datos para poder realizar llamadas a funciones y el retorno de éstas de forma correcta. La pila crece siempre hacia direcciones de memoria más bajas y empieza en direcciones de memoria altas.

En la pila se introducen elementos para hacer cálculos o para realizar llamadas a funciones, junto con sus parámetros y variables locales. De esta manera la pila va creciendo, y dado el caso, puede llegar a un límite en el que el sistema operativo detendrá el programa por haber sobrepasado el tamaño máximo de la pila (*stack overflow*). Existen *exploits* que provocan este desbordamiento. Esto puede darse si se programa una función recursiva con un número demasiado elevado de llamadas, o si simplemente está mal programada y no tiene especificada la condición de finalización de la recursividad. Otra posibilidad es que las variables locales utilizadas sean demasiado grandes. Esto puede provocar que después de varias llamadas se sobrepase el tamaño máximo de la pila; por eso es recomendable que las variables que necesiten cantidades grandes de memoria reserven su espacio en memoria de forma dinámica. De esta manera, tan sólo se guardará un puntero en la pila y los datos se ubicarán en el *heap*.

En módulos anteriores se ha visto, de manera esquemática, cómo se gestiona la pila. A continuación se muestra un ejemplo de un programa que realiza diversas llamadas a funciones, haciendo el seguimiento de los datos de la pila. El programa que se analiza es el siguiente:

```
#include <stdio.h>

int funcion2(int a) {
   int tmp2;

   printf ("Inicio funcion2: %d\n", a);
   tmp2 = a+2;
   printf ("Funcion2: tmp2 = %d\n", tmp2);
   return tmp2;
}

int funcion1(int b) {
   int tmp1;

   printf ("Inicio funcion1: %d\n", b);
   tmp1 = funcion2(b);
   tmp1 = tmp1 + 1;
   printf ("Funcion1: tmp1 = %d\n", tmp1);
   return tmp1;
}

int main() {
   int tmp_main;

   tmp_main = 7;
   printf ("Valor inicial = %d\n", tmp_main);
   tmp_main = funcion1(tmp_main);
   printf ("Resultado = %d\n", tmp_main);
   return 1;
}
```

Este programa realiza primero una llamada a la `funcion1` y, ésta a su vez, llama a la `funcion2`. El código en ensamblador de las diferentes funciones del programa anterior, compilado con `gcc` es el siguiente:

```
Función principal main

0x08048421 <main+0>:          lea        0x4(%esp),%ecx
0x08048425 <main+4>:          and        $0xfffffff0,%esp
0x08048428 <main+7>:          pushl      -0x4(%ecx)
0x0804842b <main+10>:         push       %ebp
0x0804842c <main+11>:         mov        %esp,%ebp
0x0804842e <main+13>:         push       %ecx
0x0804842f <main+14>:         sub        $0x24,%esp
0x08048432 <main+17>:         movl       $0x7,-0x8(%ebp)
0x08048439 <main+24>:         mov        -0x8(%ebp),%eax
0x0804843c <main+27>:         mov        %eax,0x4(%esp)
0x08048440 <main+31>:         movl       $0x8048594,(%esp)
0x08048447 <main+38>:         call       0x80482d8 <printf@plt>
0x0804844c <main+43>:         mov        -0x8(%ebp),%eax
0x0804844f <main+46>:         mov        %eax,(%esp)
0x08048452 <main+49>:         call       0x80483de <funcion1>
0x08048457 <main+54>:         mov        %eax,-0x8(%ebp)
0x0804845a <main+57>:         mov        -0x8(%ebp),%eax
0x0804845d <main+60>:         mov        %eax,0x4(%esp)
0x08048461 <main+64>:         movl       $0x80485a8,(%esp)
0x08048468 <main+71>:         call       0x80482d8 <printf@plt>
0x0804846d <main+76>:         mov        $0x1,%eax
0x08048472 <main+81>:         add        $0x24,%esp
0x08048475 <main+84>:         pop        %ecx
0x08048476 <main+85>:         pop        %ebp
0x08048477 <main+86>:         lea        -0x4(%ecx),%esp
0x0804847a <main+89>:         ret

Funcion1

0x080483de <funcion1+0>:      push       %ebp
0x080483df <funcion1+1>:      mov        %esp,%ebp
0x080483e1 <funcion1+3>:      sub        $0x18,%esp
```

```
0x080483e4 <funcion1+6>:      mov        0x8(%ebp),%eax
0x080483e7 <funcion1+9>:      mov        %eax,0x4(%esp)
0x080483eb <funcion1+13>:     movl       $0x804856a,(%esp)
0x080483f2 <funcion1+20>:     call       0x80482d8 <printf@plt>
0x080483f7 <funcion1+25>:     mov        0x8(%ebp),%eax
0x080483fa <funcion1+28>:     mov        %eax,(%esp)
0x080483fd <funcion1+31>:     call       0x80483a4 <funcion2>
0x08048402 <funcion1+36>:     mov        %eax,-0x4(%ebp)
0x08048405 <funcion1+39>:     addl       $0x1,-0x4(%ebp)
0x08048409 <funcion1+43>:     mov        -0x4(%ebp),%eax
0x0804840c <funcion1+46>:     mov        %eax,0x4(%esp)
0x08048410 <funcion1+50>:     movl       $0x804857f,(%esp)
0x08048417 <funcion1+57>:     call       0x80482d8 <printf@plt>
0x0804841c <funcion1+62>:     mov        -0x4(%ebp),%eax
0x0804841f <funcion1+65>:     leave
0x08048420 <funcion1+66>:     ret

Funcion2

0x080483a4 <funcion2+0>:      push       %ebp
0x080483a5 <funcion2+1>:      mov        %esp,%ebp
0x080483a7 <funcion2+3>:      sub        $0x18,%esp
0x080483aa <funcion2+6>:      mov        0x8(%ebp),%eax
0x080483ad <funcion2+9>:      mov        %eax,0x4(%esp)
0x080483b1 <funcion2+13>:     movl       $0x8048540,(%esp)
0x080483b8 <funcion2+20>:     call       0x80482d8 <printf@plt>
0x080483bd <funcion2+25>:     mov        0x8(%ebp),%eax
0x080483c0 <funcion2+28>:     add        $0x2,%eax
0x080483c3 <funcion2+31>:     mov        %eax,-0x4(%ebp)
0x080483c6 <funcion2+34>:     mov        -0x4(%ebp),%eax
0x080483c9 <funcion2+37>:     mov        %eax,0x4(%esp)
0x080483cd <funcion2+41>:     movl       $0x8048555,(%esp)
0x080483d4 <funcion2+48>:     call       0x80482d8 <printf@plt>
0x080483d9 <funcion2+53>:     mov        -0x4(%ebp),%eax
0x080483dc <funcion2+56>:     leave
0x080483dd <funcion2+57>:     ret
```

Tras ejecutar el programa, se inspecciona la pila justo antes de la primera llamada a la función `printf` (`main+38`). El estado de los datos que en ella se encuentran es el siguiente:

```
0xbfa597d0: 0x08048594      ESP (Parámetro 2 printf)
0x00000007                  (Parámetro 1 printf)
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  EBP (Contiene antiguo EBP)
```

El parámetro 2 de `printf` contiene un puntero a una dirección de memoria donde está guardada la cadena de caracteres que se le pasa como formato de salida. Si se inspecciona la memoria en esa dirección, se puede ver lo siguiente:

```
0x8048594: "Valor inicial = %d\n"
```

Se continúa con la ejecución y se para justo antes de la llamada a la función 1. El análisis de la pila en ese momento es:

```
0xbfa597d0: 0x00000007      ESP (Parámetro 1 funcion1)
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  EBP (Contiene antiguo EBP)
```

Se observa cómo se ha preparado la pila para ejecutar la llamada a la `funcion1`. Se ha colocado en el top de la pila el parámetro necesario para que la `funcion1` se ejecute correctamente. Se realiza la llamada a la `funcion1` y se vuelve a analizar la pila. Se avanza hasta antes de la llamada a la `funcion2` para analizar el contenido de la pila en ese momento.

```
0xbfa597b0: 0x00000007      ESP (Parámetro 1 funcion2)
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x08048594                  Espacio reservado y no usado
0xbfa597c0: 0xbfa597d8      Espacio reservado y no usado
0xb7f1dff4                  Variable local tmp1 (sin valor)
0xbfa597f8                  EBP (Antiguo EBP) F1
0x08048457                  Dirección de retorno a main
0xbfa597d0: 0x00000007      Parámetro 1 funcion1
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
```

Se puede observar la dirección de retorno a la función `main` que se usará cuando termine la `funcion1` para devolver la ejecución al punto siguiente en donde se había interrumpido. Se ve también dónde se ubica la variable local `tmp1`. Al igual que la preparación de la llamada a la `funcion1`, se pueden observar los parámetros que se le van a pasar a la `funcion2`. Se procede a continuación a entrar en la `funcion2` y avanzar hasta el momento en que se tiene que incrementar el valor del parámetro en 2 unidades.

```
0xbfa59790: 0x08048540      ESP (Antiguo paramet. a printf)
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x0804856a                  Espacio reservado y no usado
0xbfa597a0: 0xbfa597b8      Espacio reservado y no usado
0xb7f1dff4                  Variable local tmp2 (sin valor)
0xbfa597c8                  EBP (Antiguo EBP) F2
0x08048402                  Dirección de retorno a funcion1
0xbfa597b0: 0x00000007      Parámetro 1 funcion2
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x08048594                  Espacio reservado y no usado
0xbfa597c0: 0xbfa597d8      Espacio reservado y no usado
0xb7f1dff4                  Variable local tmp1 (sin valor)
0xbfa597f8                  Antiguo EBP F1
0x08048457                  Dirección de retorno a main
0xbfa597d0: 0x00000007      Parámetro 1 funcion1
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
```

En cada llamada a función se observan los antiguos parámetros a la función `printf`. Esto sucede porque el compilador ha decidido reservar un espacio en la pila para las variables locales y otros usos, entre los que se incluye el paso de parámetros a funciones. Como el programa tiene espacio libre en la pila para poner esos parámetros no se realiza ningún push en la pila de ellos, simplemente se escriben en las posiciones de memoria que deberían ocupar en la pila al realizar la llamada. Después de la llamada a la función estos parámetros continúan existiendo puesto que no se destruyen esas posiciones de memoria hasta que se termina cada función.

Se pasa ahora a incrementar el valor del parámetro y a guardarlo en la variable local `tmp2`. Esto se realiza con la ejecución de las líneas de código:

```
0x080483c0 <funcion2+28>:         add $0x2,%eax
0x080483c3 <funcion2+31>:         mov %eax,-0x4(%ebp)
```

El resultado de la ejecución de este código es la actualización de la variable local en la pila.

```
0xbfa59790: 0x08048540      ESP (Antiguo parám. a printf)
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x0804856a                  Espacio reservado y no usado
0xbfa597a0: 0xbfa597b8      Espacio reservado y no usado
0x00000009                  Variable local tmp2 (nuevo valor)
0xbfa597c8                  EBP (Antiguo EBP) F2
0x08048402                  Dirección de retorno a funcion1
0xbfa597b0: 0x00000007      Parámetro 1 funcion2
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x08048594                  Espacio reservado y no usado
0xbfa597c0: 0xbfa597d8      Espacio reservado y no usado
0xb7f1dff4                  Variable local tmp1 (sin valor)
0xbfa597f8                  Antiguo EBP F1
0x08048457                  Dirección de retorno a main
0xbfa597d0: 0x00000007      Parámetro 1 función1
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
Antiguo EBP Main
```

A partir de este momento se vuelve a realizar una llamada a `printf` y finalmente se prepara el retorno del valor calculado a la función anterior. En este caso el valor calculado se devuelve en el registro EAX. El contenido del registro se actualiza con la siguiente sentencia:

```
0x080483d9 <funcion2+53>:      mov 0x4(%ebp),%eax
```

Este código accede a la pila, donde está ubicada la variable local, y asigna el valor guardado en esa posición de memoria al registro EAX. Finalmente tan sólo queda la finalización de la función y devolución de la ejecución al punto posterior de la llamada realizada en la función padre. Esto se realiza con las instrucciones `leave` y `ret`. Estas instrucciones se explican con más detalle en próximos módulos pero básicamente, `leave` sirve para liberar el espacio reservado en la pila por la función llamada (variables locales) y recuperar el valor de EBP anterior. Esta instrucción sería equivalente a ejecutar:

```
mov %ebp, %esp
pop %ebp
```

Al hacer esto, en la parte superior de la pila quedaría la dirección de retorno, que sería usada por la instrucción `ret`. La ejecución de `ret` equivaldría a ejecutar:

```
pop %eip
```

Aunque esta instrucción no es válida en ensamblador, sirve para explicar qué es lo que hace la instrucción `ret`. Se pasa ahora a ejecutar el código hasta la instrucción `leave`.

```
0xbfa59790: 0x08048555      ESP (Antiguo parám. a printf)
0x00000009                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x0804856a                  Espacio reservado y no usado
0xbfa597a0: 0xbfa597b8      Espacio reservado y no usado
0x00000009                  Variable local tmp2 (nuevo valor)
0xbfa597c8                  EBP (Antiguo EBP) F2
0x08048402                  Dirección de retorno a funcion1
0xbfa597b0: 0x00000007      Parámetro 1 funcion2
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x08048594                  Espacio reservado y no usado
0xbfa597c0: 0xbfa597d8      Espacio reservado y no usado
0xb7f1dff4                  Variable local tmp1 (sin valor)
0xbfa597f8                  Antiguo EBP F1
0x08048457                  Dirección de retorno a main
0xbfa597d0: 0x00000007      Parámetro 1 función1
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
```

Este estado de la pila cambia justo después de ejecutar la instrucción `leave`.

```
0xbfa597ac: 0x08048402      ESP (Dir. de retorno a fun. 1)
0xbfa597b0: 0x00000007      Parámetro 1 funcion2
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x08048594                  Espacio reservado y no usado
0xbfa597c0: 0xbfa597d8      Espacio reservado y no usado
0xb7f1dff4                  Variable local tmp1 (sin valor)
0xbfa597f8                  Antiguo EBP F1
0x08048457                  Dirección de retorno a main
0xbfa597d0: 0x00000007      Parámetro 1 funcion1
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
```

La dirección de retorno queda en la parte superior de la pila. La instrucción ret toma lo que haya en el *top* de la pila y devuelve la ejecución del programa a esa dirección. El estado de la pila después de la instrucción `ret` es exactamente igual que justo antes de haber hecho la llamada a la `funcion2`. Este estado ya se ha visto anteriormente. El resultado devuelto por la `funcion2` se encuentra en el registro EAX. A continuación la `funcion1` asigna el resultado a la variable local `tmp1` que se encuentra en la pila. La instrucción que hace esto es:

```
0x08048402 <funcion1+36>:      mov %eax,-0x4(%ebp)
```

y el estado final de la pila después de esta acción es el siguiente:

```
0xbfa597b0: 0x00000007      ESP (Parámetro 1 funcion2)
0x00000007                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x08048594                  Espacio reservado y no usado
0xbfa597c0: 0xbfa597d8      Espacio reservado y no usado
0x00000009                  Variable local tmp1 (nuevo valor)
0xbfa597f8                  EBP (Antiguo EBP) F1
0x08048457                  Dirección de retorno a main
0xbfa597d0: 0x00000007      Parámetro 1 funcion1
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
```

A continuación, se incrementa ese valor en una unidad y se prepara EAX para devolver el resultado a la función que ha realizado la llamada. Seguidamente, se muestra la pila en el estado en que queda antes de llamar a la función `leave`.

```
0xbfa597b0: 0x0804857f      ESP (Antiguo param. a printf)
0x0000000a                  Antiguo parámetro a printf
0xb7f1e4e0                  Espacio reservado y no usado
0x08048594                  Espacio reservado y no usado
0xbfa597c0: 0xbfa597d8      Espacio reservado y no usado
0x0000000a                  Variable local tmp1 (nuevo valor)
0xbfa597f8                  EBP (Antiguo EBP) F1
0x08048457                  Dirección de retorno a main
0xbfa597d0: 0x00000007      Parámetro 1 funcion1
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x00000007      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
```

Se observa que ya se ha incrementado el valor de la variable `tmp1`, y que en el *top* de la pila hay los últimos parámetros que se han pasado a la llamada función `printf` para que muestre por pantalla el resultado de las operaciones después de la ejecución del código que modifica las variables.

Para finalizar, se muestra cómo queda la pila al finalizar la ejecución de la `funcion2` y devolver la ejecución a la función principal `main`.

```
0xbfa597d0: 0x00000007      ESP (Parámetro 1 funcion1)
0x00000007                  Antiguo parámetro a printf
0xbfa597e8                  Espacio reservado y no usado
0x080482a4                  Espacio reservado y no usado
0xbfa597e0: 0xb7f1dff4      Espacio reservado y no usado
0x080496a4                  Espacio reservado y no usado
0xbfa59808                  Espacio reservado y no usado
0x080484a9                  Espacio reservado y no usado
0xbfa597f0: 0x0000000a      Variable local tmp_main
0xbfa59810                  Antiguo valor de ECX
0xbfa59868                  Antiguo EBP Main
```

Se observa que ya se ha actualizado el valor de la variable local y todo el espacio de la pila que se había reservado para llamar a las diferentes funciones ya se ha liberado. El método a seguir para terminar el programa es exactamente el mismo. Primero se libera el espacio ocupado por las variables locales y finalmente se devuelve la ejecución a otro punto.

Se ha podido ver, paso a paso, cómo va evolucionando la pila junto con los parámetros y las variables locales definidas. Como regla general se puede decir que el acceso a las variables locales y los parámetros recibidos por una función se hace utilizando el registro EBP como base, y para pasar parámetros a otra función (manejo del *top* de la pila) se usa el registro ESP. El esquema de la memoria asignada a la pila sería:

parámetros ...	← ESP
... variables locales ...	
Anterior EBP	← EBP
Dirección de retorno (EIP)	
parámetros ...	
... variables locales ...	
Anterior EBP	
Dirección de retorno (EIP2)	
parámetros ...	

Conocer en detalle el funcionamiento de la pila (y de las estructuras de memoria), es muy importante a la hora de crear o usar *exploits*, puesto que la mayoría de ellos aprovechan vulnerabilidades de aplicaciones que no controlan bien estas estructuras.

5. Espacio de usuario y espacio de sistema

Existe un conjunto de tareas que deben ser realizadas por el sistema operativo, como es el acceso a determinados recursos de manera controlada. Entre estos recursos se puede encontrar el acceso al disco o acceso a dispositivos de entrada/salida. Es tarea del sistema operativo controlar este acceso y garantizar que las distintas aplicaciones puedan utilizar los recursos.

Este conjunto de tareas debe realizarse de manera totalmente transparente a la aplicación. Dentro de estos entornos de ejecución se encuentran dos modos de ejecución:

- Modo supervisor (espacio de sistema).
- Modo usuario (espacio de sistema).

Los procesos se ejecutan normalmente en modo usuario, y cuando necesitan acceder a determinados recursos a los que no tienen acceso por defecto (gestionados por el sistema), realizan una llamada al sistema para conseguir la operación deseada. Las llamadas al sistema son operaciones que ofrece el sistema operativo a las aplicaciones para que puedan utilizar los recursos disponibles. La manera de gestionar estas llamadas es totalmente distinta en sistemas Windows y sistemas Linux. Cuando se realiza la llamada, el sistema se encarga de que la operación solicitada se ejecute correctamente y devuelve el resultado a la aplicación que ha hecho la solicitud. De esta manera, se garantiza que el acceso a los diferentes recursos se realizará de manera correcta, por una máquina utilizada por múltiples usuarios, ya que sólo hay un árbitro que controla el acceso a los recursos: el sistema operativo.