EC - Práctica 1

Javier Gálvez Obispo

31 de octubre de 2017

1. Sesión de depuración saludo.s

1. ¿Qué contiene EDX tras ejecutar *mov longsaludo, $edx*? ¿Para qué necesitamos esa instrucción, o ese valor? Responder no sólo el valor correcto (en decimal y hex) sino también el significado del mismo (¿de dónde sale?) Comprobar que se corresponden los valores hexadecimal y decimal mostrados en la ventana Status→Registers

El registro EDX contiene el valor de *longsaludo* tras la ejecución de *mov longsaludo, %edx*, siendo en este caso 0x1C su valor en hexadecimal, 28 en decimal. El valor que almacena *longsaludo* es el número de posiciones que ocupa *saludo*, es decir, el número de bytes a escribir.

2. ¿Qué contiene ECX tras ejecutar *mov $saludo, %ecx*? Indica el valor en hexadecimal, y el significado del mismo. Realizar un dibujo a escala de la memoria del programa, indicando dónde empieza el programa (_start, .text), dónde empieza saludo (.data), y dónde está el tope de pila (%esp).

ECX contiene la dirección de la variable *saludo*. Su valor en hexadecimal es 0x8049097, que es la dirección de memoria donde está el contenido de *saludo*.

3. ¿Qué sucede si se elimina el símbolo de dato inmediato ($) de la instrucción anterior? (*mov saludo, %ecx*). Realizar la modificación, indicar el contenido de ECX en hexadecimal, explicar por qué no es lo mismo en ambos casos. Concretar de dónde viene el nuevo valor (obtenido sin usar $).

Al eliminar el símbolo $ movemos el contenido de *saludo* a ECX en vez de su dirección. Ahora su valor en hexadecimal es 0x616c6f48 que es la representación en hexadecimal del contenido de la variable *saludo*.

4. ¿Cuántas posiciones de memoria ocupa la variable *longsaludo*? ¿Y la variable *saludo*? ¿Cuántos bytes ocupa por tanto la sección de datos? Comprobar con un volcado Data→Memory mayor que la zona de datos antes de hacer Run.

La variable *saludo* ocupa 28B.
La variable *longsaludo* ocupa 1B.
La sección de datos ocupa 29B.

5. Añadir dos volcados Data →Memory de la variable *longsaludo*, uno como entero hexadecimal, y otro como 4 bytes hex. Teniendo en cuenta lo mostrado en esos volcados... ¿Qué direcciones de memoria ocupa *longsaludo*? ¿Cuál byte está en la primera posición, el más o el menos significativo? ¿Los procesadores de la línea x86 usan el criterio del extremo mayor (big-endian) o menor (little-endian)? Razonar la respuesta.

Los valcados los hacemos con las siguientes órdenes:
(gdb) x /1xb &longsaludo
0x80490b3: 0x1c

(gdb) x /4xb &longsaludo
0x80490b3: 0x1c 0x00 0x00 0x00

La dirección de memoria de *longsaludo* es 0x80490b3.
En la primera posición está el byte menos significativo.
Los procesadores x86 utilizan el criterio little-endian.

6. ¿Cuántas posiciones de memoria ocupa la instrucción *mov $1, %ebx*? ¿Cómo se ha obtenido esa información? Indicar las posiciones concretas en hexadecimal.

La instrucción *mov $1, %ebx* ocupa 5 posiciones de memoria.
(gdb) disas
Dump of assembler code for function _start:

```
   0x08048074 <+0>:    mov    $0x4,%eax
=> 0x08048079 <+5>:    mov    $0x1,%ebx
   0x0804807e <+10>:   mov    $0x8049097,%ecx
   0x08048083 <+15>:   mov    0x80490b3,%edx
   0x08048089 <+21>:   int    $0x80
   0x0804808b <+23>:   mov    $0x1,%eax
   0x08048090 <+28>:   mov    $0x0,%ebx
   0x08048095 <+33>:   int    $0x80
```

End of assembler dump.

7. ¿Qué sucede si se elimina del programa la primera instrucción *int 0x80*? ¿Y si se elimina la segunda? Razonar las respuestas.

Si eliminamos la primera instrucción *int 0x80* no aparece por pantalla el mensaje.
Si eliminamos la segunda instrucción da fallo de violación de segmento al ejecutar el programa.
Esta instrucción se utiliza para las llamadas del sistema.

8. ¿Cuál es el número de la llamada al sistema READ (en kernel Linux 32bits)? ¿De dónde se ha obtenido esa información?

El número de la llamada al sistema READ es 3.
Esta información se ve en /usr/include/asm/unistd_32.h

2. Sesión de depuración suma.s

1. ¿Cuál es el contenido de EAX justo antes de ejecutar la instrucción RET, para esos componentes de lista concretos? Razonar la respuesta, incluyendo cuánto valen 0b10, 0x10, y (.-lista)/4.

El contenido de EAX es 37 que es resultado de todas las sumas realizadas.
0b10 = 2
0x10 = 16
(.-lista)/4 = 9

2. ¿Qué valor en hexadecimal se obtiene en *resultado* si se usa la lista de 3 elementos: .int 0xffffffff, 0xffffffff, 0xffffffff? ¿Por qué es diferente del que se obtiene haciendo la suma a mano? NOTA: Indicar qué valores va tomando EAX en cada iteración del bucle, como los muestra la ventana Status→Registers, en hexadecimal y decimal (con signo). Fijarse también en si se van activando los flags CF y OF o no tras cada suma. Indicar también qué valor muestra *resultado* si se vuelca con Data→Memory como decimal (con signo) o unsigned (sin signo).

En la primera iteración el valor de EAX es 0xffffffff (-1 en decimal con signo), en la segunda 0xfffffffe (-2 en decimal con signo), y en la última, 0xfffffffd (-3 en decimal con signo). Como en la suma no hemos controlado el acarreo, el resultado con signo será -3, pero sin signo será $2^8 - 3 = 253$.

3. ¿Qué dirección se le ha asignado a la etiqueta *suma?* ¿Y a *bucle*? ¿Cómo se ha obtenido esa información?

La dirección de suma es 0x8048095
La dirección de bucle es 0x80480a0
Esta información se obtiene con la orden *p &etiqueta*

4. ¿Para qué usa el procesador los registros EIP y ESP?

EIP es el puntero a la dirección de la siguiente instrucción y ESP el puntero que apunta al inicio de la pila.

5. ¿Cuál es el valor de ESP antes de ejecutar CALL, y cuál antes de ejecutar RET? ¿En cuánto se diferencian ambos valores? ¿Por qué? ¿Cuál de los dos valores de ESP apunta a algún dato de interés para nosotros? ¿Cuál es ese dato?

El valor de ESP antes de ejecutar CALL es 0xFFFFD4A0 y antes de ejecutar RET es 0xFFFFD49C. La diferencia entre ambos valores es de 4, esta diferencia es producida porque al ejecutar CALL, el tamaño de la pila se modifica porque se guarda en ella la dirección de retorno a la función que será necesaria cuando se ejecute RET.

6. ¿Qué registros modifica la instrucción CALL? Explicar por qué necesita CALL modificar esos registros.

CALL modifica los registros EAX, EDX, EIP y ESP.
En nuestro programa modificamos EAX para usarlo como acumulador y EDX para usarlo como índice. El uso de EIP y ESP está explicado en la pregunta 4.

7. ¿Qué registros modifica la instrucción RET? Explicar por qué necesita RET modificar esos registros.

La instrucción RET modifica los registros ESP y EIP porque los punteros deberán cambiar para que el programa pueda continuar con su ejecución al punto donde estaba antes de llamar a la subrutina.

8. Indicar qué valores se introducen en la pila durante la ejecución del programa, y en qué direcciones de memoria queda cada uno. Realizar un dibujo de la pila con dicha información. NOTA: en los volcados Data→Memory se puede usar $esp para referirse a donde apunta el registro ESP.

El registro ESP al inicio del programa apunta a 0xffffd0f0 con el valor 0xffffd0f0. Al llamar a la subrutina suma apunta a 0xffffd0ec con valor 0xffffd0ec. Después de la instrucción *pop %edx* cambia a 0xffffd0ec y después de la instrucción *ret* se modifica a 0xffffd0f0.

9. ¿Cuántas posiciones de memoria ocupa la instrucción *mov $0, %edx*? ¿Y la instrucción *inc %edx*? ¿Cuáles son sus respectivos códigos máquina? Indicar cómo se han obtenido. NOTA: en los volcados Data→Memory se puede usar una dirección hexadecimal 0x... para indicar la dirección del volcado. Recordar la ventana View→Machine Code Window. Recordar también la herramienta objdump.

La instrucción *mov $0, %edx* ocupa 5 posiciones de memoria y su código máquina es:
ba 00 00 00 00
La instrucción *inc %edx* ocupa 1 posición de memoria y su código máquina es:
42

10. ¿Qué ocurriría si se eliminara la instrucción RET? Razonar la respuesta. Comprobarlo usando ddd.

Se produce una violación de segmento.

3. Suma de N enteros sin signo de 32bits

```
.section .data
lista:     .int 1,1,123,1,1,1,1,1
           .int 1,1,1,1,432,1,1,1
           .int 1,456,1,1,1,5234,1,1
           .int 1,1,1,1,1,1,1,234
longlista:  .int (.-lista)/4
resultado:  .quad -1

.section .text
_start:  .global _start

  mov   $lista, %ebx
  mov   $0, %edx      # %edx lo utilizamos para el acarreo
  mov   longlista, %ecx
  call  suma

  mov   %eax, resultado
  mov   %edx, resultado+4

  mov   $1, %eax
  mov   $0, %ebx
  int   $0x80

suma:
  push  %esi          # %esi lo utilizamos como indice
  mov   $0, %eax
  mov   $0, %esi

bucle:
  add   (%ebx,%esi,4), %eax
  jnc   acarreo       # si no hay acarreo saltamos
  inc   %edx          # en caso contrario incrementamos el acarreo

acarreo:
  inc   %esi
  cmp   %esi, %ecx
  jne   bucle

  pop   %esi
  ret
```

4. Suma de N enteros con signo de 32bits

```
.section .data
lista:     .int -1,-1,-1,-1,-1,-1,-1,-1
           .int -1,-1,-1,-32,-1,12,-1,-1
           .int -1,-1,-1,42,-1,-1,-1,-1
           .int -1,-1,-1,-1,-1,-1,956,-1
longlista:  .int (.-lista)/4
resultado:  .quad -1

.section .text
_start:  .global _start

  mov   $lista, %ebx
  mov   $0, %edx      # %edx lo utilizamos para el acarreo
  mov   longlista, %ecx
  call  suma

  mov   %eax, resultado
  mov   %edx, resultado+4

  mov   $1, %eax
  mov   $0, %ebx
  int   $0x80

suma:
  push %esi           # %esi lo utilizamos como indice
  mov $0, %eax
  mov $0, %esi

bucle:
  mov (%ebx,%esi,4), %ebp   # Movemos el entero a %ebp
  test %ebp, %ebp           # Comprobamos si es negativo
  js negativo               # Si es negativo saltamos a "negativo"

positivo:                 # Para sumar un numero positivo
  add %ebp, %eax            # Sumamos %ebp a %eax
  adc $0, %edx              # Sumamos el acarreo (si hay) a %edx
  jmp siguiente

negativo:                 # Para sumar un numero negativo
  neg %ebp                  # Valor absoluto de %ebp
  sub %ebp, %eax            # A %eax le restamos %ebp
  sbb $0, %edx              # Restamos el debito a %edx

siguiente:
  inc %esi
  cmp %esi, %ecx
  jne bucle

  pop   %esi
  ret
```

5. Media de N enteros con signo de 32bits

```
.section .data
lista:     .int -1,-1,-1,-1,-1,-1,-1,-1
           .int -1,-1,-1,-32,-1,12,-1,-1
           .int -1,-1,-1,42,-1,-1,-1,-1
           .int -1,-1,-1,-1,-1,-1,956,-1
longlista:  .int (.-lista)/4
resultado:  .quad -1

.section .text
_start:  .global _start

  mov   $lista, %ebx
  mov   $0, %edx      # %edx lo utilizamos para el acarreo
  mov   longlista, %ecx
  call  suma

  mov   %eax, resultado
  mov   %edx, resultado+4

  # Esta linea es la unica diferencia con la suma con signo
  idiv  %ecx          # Hacemos la division con signo de EDX:EAX entre %ecx
                      # que contiene el valor de longlista

  mov   $1, %eax
  mov   $0, %ebx
  int   $0x80

suma:
  push %esi           # %esi lo utilizamos como indice
  mov $0, %eax
  mov $0, %esi

bucle:
  mov (%ebx,%esi,4), %ebp   # Movemos el entero a %ebp
  test %ebp, %ebp           # Comprobamos si es negativo
  js negativo               # Si es negativo saltamos a "negativo"

positivo:                 # Para sumar un numero positivo
  add %ebp, %eax            # Sumamos %ebp a %eax
  adc $0, %edx              # Sumamos el acarreo (si hay) a %edx
  jmp siguiente

negativo:                 # Para sumar un numero negativo
  neg %ebp                  # Valor absoluto de %ebp
  sub %ebp, %eax            # A %eax le restamos %ebp
  sbb $0, %edx              # Restamos el debito a %edx

siguiente:
  inc %esi
  cmp %esi, %ecx
  jne bucle

  pop   %esi
  ret
```