Respuestas al cuestionario de la primera práctica

YÁBIR GARCÍA BENCHAKHTIR

01-11-2017

Autocomprobación (saludo.s)

1. ¿Qué contiene EDX tras ejecutar mov longsaludo, %edx? ¿Para qué necesitamos esa instrucción, o ese valor? Responder no sólo el valor concreto (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

Mediante el uso de *gdb* podemos comprobar usando la orden *break* que su valor en decimal es 28 y en hexadecimal `0x1c` que se corresponde a la longitud de la cadena *saludo* y que se almacena en *longsaludo*

El valor en hexadecimal nos indica la dirección de memoria en la que se encuentra saludo.

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).

El valor en hexadecimal de `ECX` lo podemos obtener usando `p/x $ecx` y en este caso es `0x8049097` que se corresponde con la dirección de memoria donde se encuentra `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 $)

Quitando el símbolo `$` en lugar de mover la dirección donde se encuentra mueve el valor que contiene y en este caso guarda `0x616c6f48` que se corresponde con `aloH`.

4. ¿Cuántas posiciones de memoria ocupa 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 cadena de texto saludo ocupa `28B` ya que cada char ocupa `1B` y `longsaludo` ocupa `1B` luego en total usamos `29B` de memoria.

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.

```
(gdb) x/1xb &longsaludo
0x80490b3:  0x1c
(gdb) x/4xb &longsaludo
0x80490b3:  0x1c    0x00    0x00    0x00
```

- La dirección de memoria de `longsaludo` es `0x80490b3`
- El byte en la posición menos significativa es `0x1c`, la primera posición.
- Los procesadores de la línea x86 usan el criterio little-endian ya que la posición menos significativa se encuentra en la dirección de memoria con menor indice.

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

Ejecutando `objdump -d saludo.o` vemos que ocupa 5 posiciones ó desde gdb **`break`** en la linea correspondiente y seguidamente después de hacer `run`, hacer `disas`.

```
(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.
```

Vemos como ocupa un total de 5 posiciones de memoria.

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

Si quitamos el primero no se muestra el mensaje y si quitamos la segunda se produce un error `segmentation fault`. El primero llama a la orden write y el segundo a la de salida. Esta instrucción se utiliza para hacer llamadas al 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 puede consulata en `/usr/include/asm/unistd_32.h`

Autocomprobación (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.

- `EAX` tiene 37 almacenado.
- `0b10` es 2 en binario.
- `0x10` es 16 en hexadeciaml.
- `(.-lista)/4` vale 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).

El valor `0xffffffff` se corresponde con el valor decimal `-1`. Tras llamar a la función de suma el valor que se obtiene es `0xfffffffd` que se corresponde con el valor `-3`. Es distinto del que realizamos a mano porque no tiene en cuenta el acarreo.

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

Mediante el uso en `gdb` de la instrucción `p nombre_etiqueta` podemos ver que a la etiqueta suma se le ha asignado la dirección `0x8048095` y a `buble` la dirección `0x80480a0`.

```
(gdb) p suma
$11 = {<text variable, no debug info>} 0x8048095 <suma>
(gdb) p bucle
$12 = {<text variable, no debug info>} 0x80480a0 <bucle>
```

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

`ESP` se corresponde con el puntero de pila y `EIP` con el puntero a la siguiente instrucción que se va a ejecutar.

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?

```
(gdb) break 11
Breakpoint 1 at 0x804807f: file suma.s, line 11.
(gdb) break 29
Breakpoint 2 at 0x80480a9: file suma.s, line 29.
(gdb) p/x $esp
$2 = 0xffffd030
(gdb) continue
Continuing.

Breakpoint 2, bucle () at suma.s:29

(gdb) p/x $esp
$3 = 0xffffd02c
```

La diferencia entre ambos valores es de 4. La diferencia se debe a que en la llamada a `suma` se modifica el contenido de la pila y esto cambia la dirección a la que apunta `ESP`.

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

Mediante el uso de `info registers` vemos que se han modificado `EIP` y `ESP`. Necesita modificar estos registros porque cambia la dirección de la siguiente instrucción que se va a ejecutar y porque necesita colocar en la pila la dirección de retorno de la llamada.

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

Los registros que modifica son, al igual que hacía `CALL`, `EIP` y `ESP` para poder continuar la ejecución del programa.

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

Antes de ejecutar la subrutina `suma`, `ESP` apunta a la dirección `0xffffd030`. Al llamar a suma cambia y apunta a `0xffffd02c`. Antes de hacer `ret` su valor es `0xffffd02c`.

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 di-

rección del volcado. Recordar la ventana View->Machine Code Window. Recordar también la herramienta objdump.

```
(gdb) disas
Dump of assembler code for function suma:
0x08048095 <+0>:    push   %edx
0x08048096 <+1>:    mov    $0x0,%eax
=> 0x0804809b <+6>: mov    $0x0,%edx
(gdb) p/x *0x0804809b
$4 = 0xba
```

Luego la primera orden ocupa 5 direcciones de memoria y su codigo máquina es `0xba`.

```
(gdb) disas
Dump of assembler code for function bucle:
0x080480a0 <+0>:    add    (%ebx,%edx,4),%eax
=> 0x080480a3 <+3>: inc    %edx
0x080480a4 <+4>:    cmp    %edx,%ecx
0x080480a6 <+6>:    jne    0x80480a0 <bucle>
0x080480a8 <+8>:    pop    %edx
0x080480a9 <+9>:    ret
End of assembler dump.
(gdb) p/x *0x080480a3
$5 = 0x75d13942
```

La orden `inc %edx` ocupa una posición de memoria y su código maquina `0x75d13942`.

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

Obtenemos un error en este caso de violación de segmento.

Cuestiones sobre suma64unsigned.s

1. Para N=32, ¿cuántos bits adicionales pueden llegar a necesitarse para almacenar el resultado? Dicho resultado se alcanzaría cuando todos los elementos tomaran el valor máximo sin signo. ¿Cómo se escribe ese valor en hexadecimal? ¿Cuántos acarreos se producen? ¿Cuánto vale la suma (indicarla en hexadecimal)? Comprobarlo usando ddd.

Si sumamo el mayor valor posible en 32 bits `0xffffffff` 32 veces generamos 32 acarreos. Para almacenar estos datos necesitamos 5 posiciones más en binario luego necesitamos 37 bits.