Manejo de Pila

Organización del Computador

Estamos en algún lugar de fun (1)

En algún momento, tenemos que llamar a cloneStruct. Antes de llamar, tenemos que garantizar que tenemos la pila alineada a **16 bytes**.

En nuestro caso, ya la tenemos alineada (notar que los últimos 4 bits de RSP terminan en 0). Con lo cual, hacemos el call.

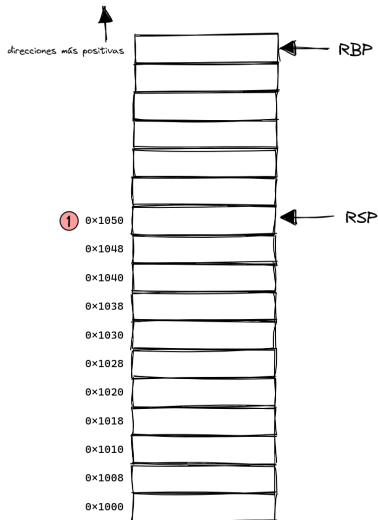
Si no la tuviésemos alineada, podríamos mover el *RSP*, haciendo por ejemplo:

```
mov rdi, rax
sub rsp, 8
call cloneStruct
add rsp, 8
```

Suponemos que tenemos un puntero al struct a copiar en *RAX*, y lo pasamos como parámetro por *RDI*

```
typedef struct empleado {
    char* nombre;
    uint8_t edad;
    uint32_t legajo;
} empleado_t;
```

```
%define OFFSET_NOMBRE 0
%define OFFSET_EDAD 8
%define OFFSET_LEGAJO 12
%define SIZE_STRUCT 16

fun:
    ...

    mov rdi, rax
    call cloneStruct   (1)

    ...


; empleado_t* cloneStruct(empleado_t*)
cloneStruct:
    push rbp
    mov rbp, rsp
    push r12
    sub rsp, 8

    mov r12, rdi
    mov rdi, SIZE_STRUCT
    call malloc

    mov rcx, [r12 + OFFSET_NOMBRE]
    mov [rax + OFFSET_NOMBRE], rcx

    mov cl, [r12 + OFFSET_EDAD]
    mov [rax + OFFSET_EDAD], cl

    mov ecx, [r12 + OFFSET_LEGAJO]
    mov [rax + OFFSET_LEGAJO], ecx

    add rsp, 8
    pop r12
    pop rbp
    ret
```



Luego del call, ya estamos en la primera línea de cloneStruct. (2)

Antes de ejecutar nada (*ANTES* de push rbp), vemos que en la pila se pusheó *RIP*, como parte de la instrucción **call**. Este es el **punto de retorno**, es decir, el lugar al cual tenemos que volver luego de terminar la función callStruct. Recordemos que RIP, siempre está apuntando a la **próxima instrucción**. El valor de RIP pusheado a la pila es entonces el de la instrucción siguiente al call.

Notemos otro detalle: al hacer el call estábamos alineados a 16 bytes. Al entrar a la función llamada, estamos alineados a 8 bytes, debido al push del *RIP*

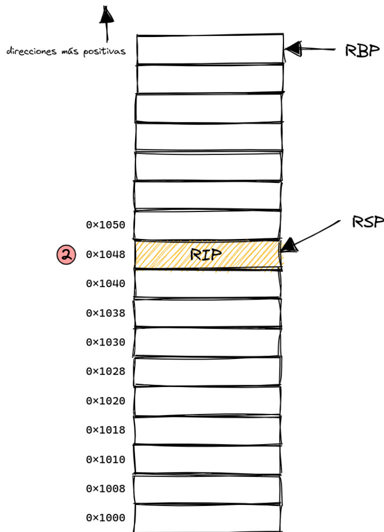
Resumiendo:

- Al hacer un call, se pushea el RIP, para poder volver a la siguiente instrucción al call
- Cuando entramos a una función, estamos desalineados

```c
typedef struct empleado {
    char* nombre;
    uint8_t edad;
    uint32_t legajo;
} empleado_t;
```

```asm
%define OFFSET_NOMBRE 0
%define OFFSET_EDAD 8
%define OFFSET_LEGAJO 12
%define SIZE_STRUCT 16

fun:
    ...

    mov rdi, rax
    call cloneStruct

    ...


; empleado_t* cloneStruct(empleado_t*)
cloneStruct:
    push rbp
    mov rbp, rsp
    push r12
    sub rsp, 8

    mov r12, rdi
    mov rdi, SIZE_STRUCT
    call malloc

    mov rcx, [r12 + OFFSET_NOMBRE]
    mov [rax + OFFSET_NOMBRE], rcx

    mov cl, [r12 + OFFSET_EDAD]
    mov [rax + OFFSET_EDAD], cl

    mov ecx, [r12 + OFFSET_LEGAJO]
    mov [rax + OFFSET_LEGAJO], ecx

    add rsp, 8
    pop r12
    pop rbp
    ret
```

(2)



Ahora **inicializamos el stack frame** (3)

¿A qué nos referimos con "inicializar el stack frame"? Básicamente consiste en salvar en la pila el viejo valor de *RBP* y actualizar su valor. ¿De qué sirve esto? Bueno, tener un punto **FIJO** en el stack frame actual puede ser útil para poder referirnos a puntos específicos en la pila. Por ejemplo, podríamos tener 2 variables locales, definidas en el stack de la siguiente manera:

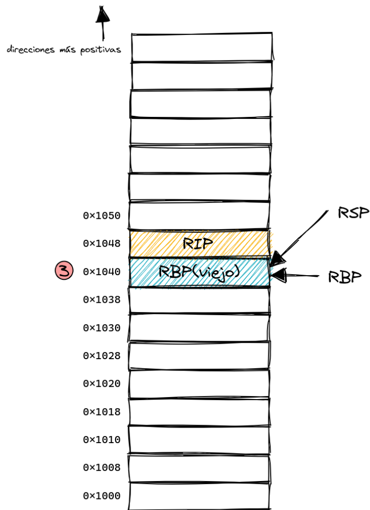
```
cloneStruct:
    push rbp
    mov rbp, rsp
    sub rsp, 16 ; 2 lugares vacíos para locales
    ...
    mov [rbp - 8], rdi  ;variable local 1
    mov [rbp - 16], rsi ;variable local 2

    ...
    add rsp, 16 ; rollback
    pop rbp
    ret
```

Esos 2 lugares reservados podemos usarlos para guardar lo que querramos y sirven de alternativa a guardar cosas en registros.

Tener *RBP* fijo sirve para que las referencias a esas variables **sean siempre las mismas** dentro de un mismo stack frame. Referirse a ellas mediante el *RSP* es posible también, pero tiende a ser más frágil para humanos :(

```
typedef struct empleado {
    char* nombre;
    uint8_t edad;
    uint32_t legajo;
} empleado_t;
```

```
%define OFFSET_NOMBRE 0
%define OFFSET_EDAD 8
%define OFFSET_LEGAJO 12
%define SIZE_STRUCT 16

fun:
    ...

    mov rdi, rax
    call cloneStruct

    ...

; empleado_t* cloneStruct(empleado_t*)
cloneStruct:
    push rbp          ; (3)
    mov rbp, rsp
    push r12
    sub rsp, 8

    mov r12, rdi
    mov rdi, SIZE_STRUCT
    call malloc

    mov rcx, [r12 + OFFSET_NOMBRE]
    mov [rax + OFFSET_NOMBRE], rcx

    mov cl, [r12 + OFFSET_EDAD]
    mov [rax + OFFSET_EDAD], cl

    mov ecx, [r12 + OFFSET_LEGAJO]
    mov [rax + OFFSET_LEGAJO], ecx

    add rsp, 8
    pop r12
    pop rbp
    ret
```



En general, luego de armar el stack frame, armamos lo que se llama el prólogo de la función, que consiste en pushear registros y reservar espacio en la pila. (4)

En nuestro caso, se pushea *R12* y se reserva un espacio vacío en la pila ¿Por qué hacemos esto?

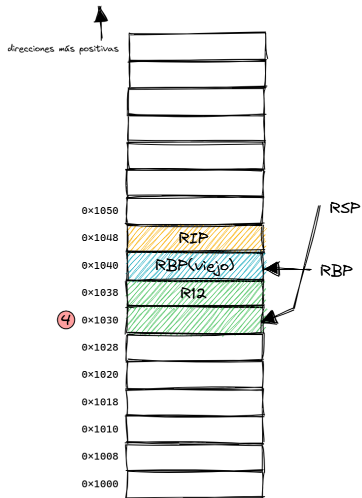
- *R12* se pushea porque vamos a hacer un call dentro de nuestra función (a `malloc`), y necesitamos preservar el puntero al struct contenido en el registro volátil *RDI*. Al estar copiado en *R12*, tenemos garantías que luego del llamado a malloc, su valor va a ser preservado
- De la misma manera, si usamos *R12*, nosotros tenemos que preservarlo a la función que nos llamó (`fun` en nuestro caso), por eso lo mandamos a la pila
- El espacio que reservamos en la pila es para que antes de hacer el call, estemos alineados a 16 nuevamente

Recordemos:

- **Volátiles**: RAX, RCX, RDX, RDI, RSI, R8, R9, R10, R11
- **No volátiles**: RBX, R12, R13, R14, R15, RBP

```c
typedef struct empleado {
    char* nombre;
    uint8_t edad;
    uint32_t legajo;
} empleado_t;
```

```asm
%define OFFSET_NOMBRE 0
%define OFFSET_EDAD 8
%define OFFSET_LEGAJO 12
%define SIZE_STRUCT 16

fun:
    ...

    mov rdi, rax
    call cloneStruct

    ...


; empleado_t* cloneStruct(empleado_t*)
cloneStruct:
    push rbp
    mov rbp, rsp
    push r12
    sub rsp, 8        (4)

    mov r12, rdi
    mov rdi, SIZE_STRUCT
    call malloc

    mov rcx, [r12 + OFFSET_NOMBRE]
    mov [rax + OFFSET_NOMBRE], rcx

    mov cl, [r12 + OFFSET_EDAD]
    mov [rax + OFFSET_EDAD], cl

    mov ecx, [r12 + OFFSET_LEGAJO]
    mov [rax + OFFSET_LEGAJO], ecx

    add rsp, 8
    pop r12
    pop rbp
    ret
```



Luego llamamos a malloc con el tamaño del struct ⑤

A partir de este momento y hasta que malloc retorne hemos cambiado de stack frame. El stack frame actual es el de malloc.

La convención C garantiza, al retornar de malloc, que:

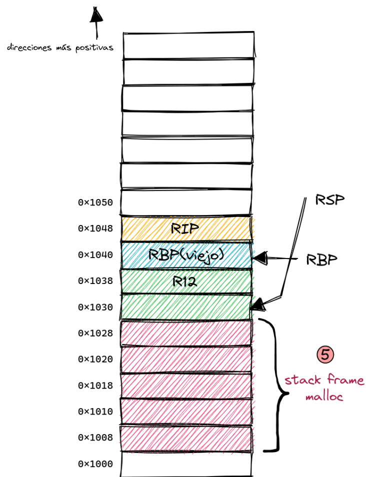
- los registros no volátiles conservarán su valor
- la pila y el RSP, tendrán los mismos valores que antes del llamado

Luego de malloc, nos dedicamos a copiar uno a uno los campos del struct original al nuevo struct. Notar el uso de constantes definidas arriba, que facilita la lectura y posterior debugging

```c
typedef struct empleado {
    char* nombre;
    uint8_t edad;
    uint32_t legajo;
} empleado_t;
```

```nasm
%define OFFSET_NOMBRE 0
%define OFFSET_EDAD 8
%define OFFSET_LEGAJO 12
%define SIZE_STRUCT 16

fun:
    ...

    mov rdi, rax
    call cloneStruct

    ...


; empleado_t* cloneStruct(empleado_t*)
cloneStruct:
    push rbp
    mov rbp, rsp
    push r12
    sub rsp, 8

    mov r12, rdi
    mov rdi, SIZE_STRUCT
    call malloc

    mov rcx, [r12 + OFFSET_NOMBRE]
    mov [rax + OFFSET_NOMBRE], rcx

    mov cl, [r12 + OFFSET_EDAD]
    mov [rax + OFFSET_EDAD], cl

    mov ecx, [r12 + OFFSET_LEGAJO]
    mov [rax + OFFSET_LEGAJO], ecx

    add rsp, 8
    pop r12
    pop rbp
    ret
```

⑤



Ya terminando la función, realizamos lo que se llama el epílogo. (6)

Aquí, restauramos la pila a su antiguo estado, y restauramos los registros no volátiles que usamos a sus valores originales (en nuestro caso, sólo usamos *RBP* y *R12*). Notar como el registro *RBP* asume su antiguo valor, correspondiente al stack frame de la función que nos llamó.

En el momento de ejecutar la instrucción `ret`, el RSP debe estar apuntando justamente al valor de retorno. Se puede pensar que la instrucción RET, realiza algo así como un `pop rip`

En *RAX* tenemos el valor devuelto por `cloneStruct`. El RSP vuelve a la dirección `0x1050`.

```c
typedef struct empleado {
    char* nombre;
    uint8_t edad;
    uint32_t legajo;
} empleado_t;
```

```nasm
%define OFFSET_NOMBRE 0
%define OFFSET_EDAD 8
%define OFFSET_LEGAJO 12
%define SIZE_STRUCT 16

fun:
    ...

    mov rdi, rax
    call cloneStruct

    ...


; empleado_t* cloneStruct(empleado_t*)
cloneStruct:
    push rbp
    mov rbp, rsp
    push r12
    sub rsp, 8

    mov r12, rdi
    mov rdi, SIZE_STRUCT
    call malloc

    mov rcx, [r12 + OFFSET_NOMBRE]
    mov [rax + OFFSET_NOMBRE], rcx

    mov cl, [r12 + OFFSET_EDAD]
    mov [rax + OFFSET_EDAD], cl

    mov ecx, [r12 + OFFSET_LEGAJO]
    mov [rax + OFFSET_LEGAJO], ecx

    add rsp, 8
    pop r12
    pop rbp
    ret
```

(6)