Université Gustave Eiffel — INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

L3 Info

TD Prog.C n°4

Traceur d'allocation
partie 1 : mise en place modulaire

Ce TP constitue la 1° partie de ce qui deviendra au final une bibliothhèque partagée (`libmtrack.so`) permettant de tracer les appels aux fonctions de gestion mémoire (`malloc, calloc, realloc, free`) dans un programme, sans intervention dans le code source, l'activation se faisant via des directives de compilation.

La première étape est la mise en place du principe de «patch à la compilation» sur des exemples primaires.

1°) Module de «patch» et principe de détournement

Le code de départ, qui servira de test, est un programme tout simple contenant un appel à `malloc`, un à `free` et une erreur entre les deux : il se compile sans erreur ni `warning` mais l'exécution échoue (`free(): invalid pointer`).

test00.c
```c
#include <stdlib.h>

int main(void)
{
    char *a=malloc(1);
    a++;
    free(a);
    return 0;
}
```

xterm
```
$> gcc -Wall test00.c -o test00
```

ou plus simplement, via un `Makefile` :

xterm
```
$> make test00
$> ./test00
free(): invalid pointer
Abandon (core dumped)
```

① (fonctions clônes primaires)

Dans cette première étape, il va s'agir de mettre en place la structure de base modulaire : un module dédié aux fonctions clônes qui viendront «patcher» les fonctions de gestion mémoire.

On se contentera pour l'instant de définir les fonctions clônes de → `malloc` et `free` a minima :

src/mtrack1.c
```c
#include <stdio.h>
#include <stdlib.h>

extern void* patch_malloc(size_t size)
{
  void *ptr = malloc(size);
  fprintf(stderr,"<patch_malloc>(%zu) -> %p\n",size,ptr);
  return ptr;
}

extern void patch_free(void *ptr)
{
  fprintf(stderr,"<patch_free>(%p)\n",ptr);
  return free(ptr);
}
```

include/mtrack1.h
```c
#ifndef _MTRACKC_H_
  #define _MTRACK_H_

  #include <stdlib.h>

  void* patch_malloc(size_t size);
  void  patch_free(void *ptr);

#endif
```

→ compilation (adaptation du `Makefile`) :

xterm
```
$> gcc -c src/mtrack1 -o mtrack1.o
```

Pour utiliser ce module, le code cible et sa chaîne de compilation devraient logiquement être adaptés :

test1.c
```c
#include <stdlib.h>
#include <mtrack1.h>

int main(void)
{
    char *a=patch_malloc(1);
    a++;
    patch_free(a);
    return 0;
}
```

→ compilation (adaptation `Makefile`) / exécution

xterm
```
$> gcc -c src/test1 -o test1.o
$> gcc mtrack1.o test1.o -o test1
```

xterm
```
$> ./test1
<patch_malloc>(1) -> 0x561f04baf2a0
<patch_free>(0x561f04baf2a1)
free(): invalid pointer
Abandon (core dumped)
```

② **(transfert d'action au pré-compilateur)**

Puisque l'on vise une action "invisible" du module de traçage, celui-ci doit pouvoir être utilisé sans modification directer du code initial `src/test0.c` :

- pas d'appel explicite au header `include/mtrack1.h`.
- pas d'appel explicite aux fonctions *patchées* : on garde `malloc` et `free`

Le «détournement» est délégué à la chaîne de compilation (étape ① : pré-processeur `cpp`) :

ⓐ dans le *header* `mtrack1.h` créer des "alias" (macros) pour les clônes de sorte que ce soit le pré-compilateur qui remplace, dans le code cible, les appels à `malloc` et `free` par leurs clônes.

ⓑ ce *header*, indispensable pour que le remplacement soit effectif, doit maintenant être inclu dynamiquement à la compilation via l'option[(1)] `-include include/mtrack1.h`

ⓒ l'exécution quant à elle est identique celle de la version détournée : ce sont bien les fonctions `patch_*` qui sont appelées.

Attention : deux points fondamentaux qu'il ne faut pas rater ici

① La règle de fabrication du code objet principal (`test0.o`) change (inclusion dynamique du *header*) mais pour le module de patch `mtrack1.o` on doit garder la règle initiale : surtout pas d'inclusion dynamique de `mtrack1.h`

② Pour la même raison, le corps du module de patch `src/mtrack1.c` ne doit surtout pas faire appel à son propre *header* `mtrack1.h`

Ce sont donc surtout les procédures de compilation (`Makefile`) qui requière de l'attention.

③ **(élargissement / généralisation)**

ⓐ étendre le fonctionnement aux autres fonctions de gestion mémoire de la `libC` (`calloc, realloc`)

ⓑ créer de nouveaux codes de test faisant appel à ces fonctions et contenant diverses erreurs : les versions suivantes du `patch` devront essayer de les détecter et éventuellement les contourner.

ⓒ adapter le `Makefile` pour automatiser la compilation «détournée».

ⓓ au final, adapter la *structuration* des répertoires pour séparer clairement

- ce qui relève du *module traceur* (`src/mtrack.c, include/mtrack.h, Makefile, mtrack.o`)
- ce qui relève du code cible à tracer (`src/cible.c, Makefile, a.out`).

④ **(tests modulaire)**

Pour tester ce *traceur* primitif sur une cible un peu plus importante, récupérer le code modulaire mis en place au `TP3` (listes masquées) et tester simplement la nouvelle recette de compilation.

Le principe du `patch` de détournement étant maintenant en place (inclusion dynamique des alias pour les clônes et traitement par le pré-processeur), il faut maintenant développer les fonctions clônes du module de traçage pour qu'elles enregistrent un maximum d'informations pertinentes en provenance du code cible.

[(1)] ou `-Iinclude -include mtrack1.h` → le `Makefile` est fait pour ça