Mini Rapport - Julian

TP Debug

Note : Les fichiers originaux qui ont servis pour la réalisation du TP se trouvent dans `./og_exos`

Exercice 1

a)

i)

Le `main.c` applique une convolution avec un laplacien sur une image afin d'appliquer un "filtre" sur une image.
Généralement, on utilise cette méthode pour "lisser" les bord d'une image en la rendant un peu plus flou et en mélangeant les couleurs.

ii)

- **Convolution :** Dans le traitement d'images, la convolution est un outil puissant, utilisé pour diverses tâches telle que la détection de contours, le floutage et l'amélioration de la netteté.
- **Laplacien :** En général, on utilise le Laplacien pour détecter les contours d'une image (lorsque des pixels côtes à côtes sont différents).

b)

Le `Makefile` se trouve dans `./ex1/Makefile`.

c)

Non, il y a une erreur de segmentation.

d)

Le bug est une erreur de segmentation à cause de mauvaises bornes définies das la boucle.
Pour trouver cette erreur, j'ai utilisé `gdb`. Les premières lignes d'erreur qu'il m'a renvoyé sont les suivantes :

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555851 in apply_laplacian (src=src@entry=0x7fffffffdac0,
dest=dest@entry=0x7fffffffdad0) at main.c:23
23                              src->px[(y - 1) * src->width + x].r +
```

e)

Les corrections ont été faites au lignes `19` et `20` dans le fichier `./ex1/main.c`.
Les images générées se trouvent dans `./ex1/medias`.

Exercice 2

a)

Les fonctions `process_dataXX` sont fausses car la borne donnée à la boucle `for` de chaque fonction est fausse, ce qui crée des erreurs de segmentation.

b)

Voici les raisons des erreurs pour chaque fonctions:

- **process_data :** Le programme ne plante pas, mais il y a quand même une erreur car la borne de la boucle `for` dépasse la taille du tableau `data[]` de `1`. Il n'y a pas de plantage car le programme a écrit dans une autre case mémoire appartenant toujours au programme lui-même, mais n'appartenant pas au tableau `data[]`.
- **process_data2 :** Le programme ne plante pas, mais il y a quand même une erreur car la borne de la boucle `for` dépasse la taille du tableau `data[]` de `50`. Il n'y a pas de plantage pour la même raison expliqué pour **`process_data`**.
- **process_data3 :** Il y a un plantage car la borne de la boucle `for` dépasse la taille du tableau `data[]` de `size * 1000`. Le plantage est dû au fait que l'on dépasse la zone mémoire réservé au programme, il essaie d'écrire sur une case mémoire qui ne lui appartient pas.

c)

Pour `process_data`, l'outil qui peut être utilisé pour détecter ce type d'erreur est **ASAN** (le paramètre `-fsanitize=address` dans le `gcc`).
Pour `process_data3`, l'outil qui peut être utilisé pour détecter ce type d'erreur est **GBD**.

d)

Une corruption mémoire ne mène pas toujours au plantage car il est possible que lorsqu'on sort d'un tableau, par exemple, on modifie une adresse du programme lui-même sans accéder à des cases mémoires qui ne lui appartient pas.
Cela peut causer des comportement indésirés plus tard dans le programme.

Exercice 3

a)

Une fuite de mémoire est un problème qui arrive lorsqu'on ne désalloue pas la mémoire pour des variables alloué dynamiquement. Cela produit une occupation croissante, non contrôlée et non désirée de la taille des données stockées en mémoire, ce qui peut donc entraîner une saturation de la mémoire de l’ordinateur, et dans le pire des cas, un crash du système.

b)

L'outil à utiliser est `ASAN` ou `Valgrind`.

c)

La correction se trouve dans `./ex3/leak.c` à la ligne `29`.
Il fallait juste enlever le `++` à coté du `i`.

Exercice 4

a)

Ce programme fait la somme de tous les éléments se trouvant dans un tableau de taille `5`.

b)

Le programme fonctionne correctement, il n'y a aucun retour d'erreur et le nombre retourné est `15`, ce qui est le résultat attendu.

c)

A chaque tour de boucle, en utilisant `gdb`, j'obtient d'abord `1`, puis `3`, puis `6`, puis `10`, puis `15`, puis encore `15`.
Ce résultat m'étonne malgré l'erreur présente sur la borne de la boucle qui est d'une unité trop grande. Il ne me donne pas la 6eme itération de boucle dans `gdb` et j'obtient aussi `15` deux fois en vérifiant la valeur de `sum` avec des `printf`. Néanmoins, la valeur finale devient anormale si l'on fait 7 itérations au lieu de 6.
Il est à noter que ASAN détecte bien l'overflow dû à la mauvaise borne de la boucle.

Voici ce que me retourne `gdb` :

```
(gdb) r
Starting program: /home/isty/Desktop/Ecole/Techniques et Outils
d'Ingénierie Logiciel/TP_Debug/realisation/ex4/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at somme.c:3
3       int main() {
(gdb) watch sum
Hardware watchpoint 2: sum
(gdb) c
Continuing.

Hardware watchpoint 2: sum

Old value = 0
New value = 1
main () at somme.c:8
8           for (int i = 0; i <= 5; i++) { // Bug: Utilisation de <= au
lieu de <
(gdb) c
```

```
Continuing.

Hardware watchpoint 2: sum

Old value = 1
New value = 3
main () at somme.c:8
8           for (int i = 0; i <= 5; i++) { // Bug: Utilisation de <= au
lieu de <
(gdb) c
Continuing.

Hardware watchpoint 2: sum

Old value = 3
New value = 6
main () at somme.c:8
8           for (int i = 0; i <= 5; i++) { // Bug: Utilisation de <= au
lieu de <
(gdb) c
Continuing.

Hardware watchpoint 2: sum

Old value = 6
New value = 10
main () at somme.c:8
8           for (int i = 0; i <= 5; i++) { // Bug: Utilisation de <= au
lieu de <
(gdb) c
Continuing.

Hardware watchpoint 2: sum

Old value = 10
New value = 15
main () at somme.c:8
8           for (int i = 0; i <= 5; i++) { // Bug: Utilisation de <= au
lieu de <
(gdb) c
Continuing.
La somme des éléments du tableau est : 15

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0x00007ffff7da7d90 in __libc_start_call_main
(main=main@entry=0x555555555169 <main>, argc=argc@entry=1,
argv=argv@entry=0x7fffffffdc38) at
../sysdeps/nptl/libc_start_call_main.h:58
58      ../sysdeps/nptl/libc_start_call_main.h: No such file or directory.
(gdb) c
Continuing.
[Inferior 1 (process 44173) exited normally]
```

d)

Dans tous les cas, il y a un problème avec la borne de la boucle `for` à cause du fait que l'on fait 6 itérations au lieu de 5.
Il faut donc mettre un `<` au lieu de `<=` pour éviter les problème d'overflow.
La correction a été faite à la ligne `8` dans le fichier `./ex4/somme.c`.

Exercice 5

a)

Ce code est vulnérable car il est possible d'écrire plus de valeur que ce qui est possible dans le tableau `buffer[]` et faire une erreur de segmentation, voire même faire des choses que le programme n'est pas censé faire.

b)

La pile ressemblerai à ça lors de l'appel de la fonction `vulnerable_function()` :

```
                PILE
  password_is_good 0
         buffer[8] ?
                   ?
                   ?
                   ?
                   ?
                   ?
                   ?
                   ?
                   ?
```

c)

On peut déclencher l'erreur en entrant plus de `8` caractères lors de la saisie du mot de passe.

d)

On doit taper `aaaaaaaab` par exemple pour afficher le message "Vous avez cassé le MDP!". Du moment que le 9ème caractère est un `b`, cela fonctionnera toujours.

Exercice 6

a)

Grâce à `gdb`, il est possible de voir que si `n = 1` et qu'on essaie de soustraire 2 unité à ce dernier, la valeur reviendra à celle maximale permise pour un `unsigned long long`.

b)

L'erreur est dû au fait que la condition du `if` ne permet pas de s'arrêter assez tôt pour éviter le problème.

c)

Dans le `if`, il fallait mettre `2` au lieu de `0`.
La correction a été faites à la ligne `7` dans le fichier `./ex6/fibo.c`.

Exercice 7

a)

i)

Un thread, ça ressemble à un processus, à l'exception qu'il s'exécute dans le même processus dans lequel il a été lancé afin de faire de l'exécution en parallèle.

ii)

Un mutex est une sorte de "verrou" qui permet, en général, de bloquer un ou plusieurs threads afin d'éviter que des variables partagées soient utilisées en même temps.

iii)

Le code crée 2 threads et attend qu'ils se terminent.
Les 2 threads se lancent, font leurs opérations, mais il y a un *deadlock* car les deux essaient de récupérer un mutex déjà pris.

b)

Le `Makefile` se trouve dans `./ex7/Makefile`.

i)

Le code ne dépend d'aucune bibliothèque système, la commande `make` ne renvoie pas d'erreur de référencement pour ma part et le code s'exécute très bien.

c)

Le code ne se termine jamais car il y a un *deadlock*, les deux threads essaie de récupérer un mutex déjà pris.

i)

Pour afficher l'état du programme, il faut d'abord se rattacher à lui après l'avoir exécuté en faisant `sudo gdb -p <PID_du_programme>` (sans `sudo`, gdb ne se rattache pas au programme pour moi).
Ensuite, on utilise la commande `backtrace` pour voir l'état du programme.

ii)

L'état que j'ai obtenu est le suivant :

```
(gdb) backtrace
#0  __futex_abstimed_wait_common64 (private=128, cancel=true, abstime=0x0,
op=265, expected=95911,
    futex_word=0x7faeb8642910) at ./nptl/futex-internal.c:57
#1  __futex_abstimed_wait_common (cancel=true, private=128, abstime=0x0,
clockid=0, expected=95911,
    futex_word=0x7faeb8642910) at ./nptl/futex-internal.c:87
#2  __GI___futex_abstimed_wait_cancelable64
(futex_word=futex_word@entry=0x7faeb8642910, expected=95911,
    clockid=clockid@entry=0, abstime=abstime@entry=0x0,
private=private@entry=128)
    at ./nptl/futex-internal.c:139
#3  0x00007faeb86dc624 in __pthread_clockjoin_ex (threadid=140388394608192,
thread_return=0x0, clockid=0,
    abstime=0x0, block=<optimized out>) at ./nptl/pthread_join_common.c:105
#4  0x000055d59fada3dd in main () at lock.c:55
```

iii)

La commande `thread apply all bt` permet d'avoir un `backtrace` pour tous les threads du programme.

d)

Pour corriger le code, il existe plusieurs solutions.
L'une d'entre elle serait, dans `thread1`, de déverrouiller `lock1` en premier après la simulation du travail.
La correction se trouve dans `./ex7/lock.c` à la ligne `16`.