DETROYAT Alexis
VINCENT Côme
BODART Maxime

MOCA
Rapport
Semaine 6/7

Semaine 6

TP Valgrind

En exécutant le programme fourni avec l'outil valgrind, on constate que l'on effectue une écriture invalide, à une adresse mémoire interdite :

```
==931== Invalid write of size 1
==931==    at 0x484EE8E: strcpy (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==931==    by 0x10920B: shazam (valgrind_test.c:8)
==931==    by 0x10924D: main (valgrind_test.c:14)
==931==  Address 0x4a98044 is 0 bytes after a block of size 4 alloc'd
==931==    at 0x4848899: malloc (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==931==    by 0x1091F4: shazam (valgrind_test.c:7)
==931==    by 0x10924D: main (valgrind_test.c:14)
```

Cela s'explique par le malloc(n) qui n'alloue pas d'espace mémoire pour le caractère de fin de chaîne (sentinelle) '\0'. Avec un malloc(n+1), le bug est corrigé.

TP ASan

En exécutant le programme fourni avec l'option d'address sanitizer, on constate que l'on obtient un buffer overflow :

```
==1445==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x602000000014 at pc 0x7f98e26954bf bp 0x7ffd974efd60 sp 0x7ffd974ef508
WRITE of size 5 at 0x602000000014 thread T0
    #0 0x7f98e26954be in __interceptor_strcpy
../../../../src/libsanitizer/asan/asan_interceptors.cpp:440
    #1 0x55c6b978026b in shazam /home/alexdet/MOCA/TP5/asan_test.c:8
    #2 0x55c6b97802cd in main /home/alexdet/MOCA/TP5/asan_test.c:14
    #3 0x7f98e2441d8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
    #4 0x7f98e2441e3f in __libc_start_main_impl ../csu/libc-start.c:392
```

```
    #5 0x55c6b9780164 in _start
(/home/alexdet/MOCA/TP5/asan_test+0x1164)

0x602000000014 is located 0 bytes to the right of 4-byte region
[0x602000000010,0x602000000014)
allocated by thread T0 here:
    #0 0x7f98e26f5887 in __interceptor_malloc
../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x55c6b9780254 in shazam /home/alexdet/MOCA/TP5/asan_test.c:7
    #2 0x55c6b97802cd in main /home/alexdet/MOCA/TP5/asan_test.c:14
    #3 0x7f98e2441d8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
```

Le problème vient toujours du même malloc(n), et apparaît un peu plus clairement ici : on tente d'écrire 5 octets (premier paragraphe) dans une région mémoire de taille 4 octets (second paragraphe).

TP Stack Protector

A l'exécution de l'application avec -std=c99 sur machine personnelle, on obtient :

```
*** stack smashing detected ***: terminated
Aborted
```

Cela correspond à un débordement dans l'espace mémoire dédié à la pile : on écrit bien au delà du tableau (dont un seul élément est alloué) dans la fonction `moo`. Si l'on désactive le stack protector présent par défaut (et responsable de l'erreur affichée), on obtient une segmentation fault.

En revanche, sur les machines de l'UFR, on obtient :

```
coincoin
```

Ainsi, en débordant (volontairement) de la pile **stuff**, on écrase la valeur de retour de la fonction **moo** pour la remplacer par l'adresse de la fonction **muff**. Il s'ensuit logiquement l'entrée dans cette fonction et l'exécution de son code (ce qui n'était jamais censé se produire, étant donné qu'on n'appelle jamais cette fonction).

Avec l'option `-O2` (optimisation du compilateur) sur machine personnelle, on obtient :

```
ret = 55a39f337220 (muff=55a39f337220)
*** stack smashing detected ***: terminated
Aborted
```

On a donc toujours la même erreur du stack protector, mais l'affichage du programme est tout de même effectué. Si l'on désactive le stack protector, on obtient :

```
ret = 55d012e1f1e0 (muff=55d012e1f1e0)
Segmentation fault
```

Sur les machines de l'UFR, on obtient une erreur de segmentation classique, étant donné que le stack protector n'est pas présent par défaut :

```
Segmentation fault
```

Dans les deux cas, le comportement souhaité (écraser l'adresse de retour après exécution de la fonction moo pour aller exécuter muff et afficher "coincoin") n'est pas observé. Ce comportement particulier dépend énormément des architectures et notamment des versions de compilateur.

On observe en outre que sur nos machines personnelles, des canaris sont automatiquement placés par le compilateur dans la pile, afin d'afficher une erreur en cas de stack smashing. Tandis que sur les machines de l'UFR, il faut préciser l'option de compilation **`-fstack-protector-all`** afin de placer ces canaris dans toutes les fonctions.

Par ailleurs, si l'on s'amuse à ne pas écrire dans toutes les cases du tableau pour effectuer le débordement, mais dans une seule cas isolée et loin du début de tableau, le stack protector ne fonctionne plus du tout ! On peut ainsi outrepasser les canaris, qui se trouvent à des emplacements précis en mémoire. Par exemple, à partir de **`t[4] = data`**, même avec l'option **`-fstack-protector-all`**, on n'obtient aucune erreur.

Projet

Ajout d'ASAN en librairie statique inconditionnellement dans les trois makefile pour premiers tests. Beaucoup de fuites de mémoire. Une erreur de segmentation arrive de manière aléatoire selon la compilation. Résolution en vérifiant que les librairies nécessaires (gcov, asan) soient correctement ajoutées à chaque étape de la compilation.

Semaine 7

ASan

D'abord travail sur les fichiers de test unitaires, tentative de correction des fuites de mémoires de CuTest puis suppression des warnings venant des fonctions de CuTest :

- initialisation des valeurs de tests pour éviter leaks
- ajout attribut __attribute__((no_sanitize("address"))) ne marche pas, utilisation d'un fichier de suppression : pb différence format ASAN et LSAN ; suppression leak donc format LSAN

Finalement, abandon d'ASAN sur les tests unitaires pour se concentrer sur les tests systèmes pour enlever les erreurs du code externe à l'application. Détection de plusieurs centaines de kilo octets de fuites sur le test wells.txt par exemple. Certaines fuites dues à des allocations inutiles, d'autres à une absence de free.

Première tentative de correction avec des fonctions de free récursif des structures à la fin du programme. Reste des fuites donc des zones allouées deviennent inaccessibles au cours de l'exécution du programme. Correction une par une à l'aide des indications d'ASAN. Réflexion sur la partie du code qui a la responsabilité de chaque zone allouée, tentative de free dans le même scope que l'allocation. Quand c'est impossible, free juste après les instructions qui rendent inaccessibles la zone.

Valgrind

Lors de l'exécution de Valgrind au début on obtient des erreurs telles que "**`Conditional jump or move depends on uninitialised value(s)`**" et on remarque que cela provient de "**`Uninitialised value was created by a heap allocation`**" dans les fichiers **`dico.c`** et **`dico_tools.c`** grâce à l'option **`--track-origins=yes`** qui nous indique où la création des valeurs qui ne sont pas initialisé a été réalisé.

Après vérification des fichiers, on a d'abord constaté que l'on ne vérifiait pas les valeurs de retour des **`malloc.`** Grâce à Valgrind, nous nous sommes également aperçus que nos blocs mémoires alloués, ainsi que certains pointeurs associés aux structures utilisées, n'étaient pas initialisés à 0 (NULL), ce qui causait des bugs lorsqu'on cherchait à interrompre une boucle en rencontrant un pointeur nul. Ces problèmes ont été corrigés.

En outre, en utilisant Valgrind sur l'exécutable généré dans le répertoire **`auto_tests,`** nous avons observé que le test de la fonction de hachage **`test_hash`** ne passe plus, alors qu'il est bien validé sans cet outil. En effet, nous avons observé qu'en présence de l'environnement Valgrind, les débordements arithmétiques (induits par le calcul du hash des mots) se comportent de manière particulière, et ce malgré l'utilisation du flag **`-fwrapv`** pour forcer le calcul du modulo lors des débordements.

Nous avons aussi utilisé Valgrind pour les fuites mémoires avec les options **`--leak-check=full`** et **`--show-leak-kinds=all`** qui nous montrait où se situaient

les allocations mémoires qui n'ont pas été libérées. Il permettait aussi de voir où se trouvaient les double free avec les emplacements où se trouvaient l'allocation mémoire et la tentative de libération incorrecte. Cela nous a permis de comprendre que lors de la construction de **copiedico**, on alloue de nouveaux noeuds (via la fonction **deserializeDico**), mais ces noeuds contiennent des structures de **mot_data_t** dont les pointeurs (sur les listes d'emplacements des mots dans le texte) correspondent à ceux de **dictionary**. Ainsi, on obtient deux dictionnaires différents (**dictionary** et **copiedico**) dont les noeuds doivent être libérés individuellement, mais dont le contenu des noeuds est commun !

Cela nous a posé beaucoup de problèmes de libération mémoire. Finalement, nous avons choisi de commencer par libérer le contenu commun des noeuds, qui se trouve dans la structure **serialized_dico** suite à l'appel à **serializeDico**, pour ensuite pouvoir libérer sereinement les structures de mots (**mot_t**) ainsi que les noeuds des deux dictionnaires séparément. Voir les commentaires en fin de **main** au sujet de la stratégie de libération.

Stack protector

Ajout de l'option de compilation `-fstack-protector-all` aux trois Makefiles de l'application, de manière systématique. Pas de stack smashing détecté.

Autres

Dans les Makefiles du projet, factorisation de la gestion des options **asan**, **gcov** et **stack protector** : une variable par cible concernée (**systest** et **unitest**) qui permet de choisir quelle instrumentation on ajoute pour chacune : pour changer l'instrumentation du programme généré par la règle unitest il suffit de changer la variable `UNITEST` du makefile principal, en ajoutant `ASAN=1` par exemple. Plus besoin d'aller changer les makefiles dans `auto_test` et `libs`.

Ajout d'un fichier de tests dédié aux fonctions de gestion des dictionnaires `dico_tools.c`. Correction des bugs associés à ce fichier source, et remaniement du code : rétablissement de la gestion du cas de base avant celle des cas récursifs dans les structures d'arbres, clarification du code (remplacement des boucles `while` par des conditionnelles **if** étant donné qu'on utilise des appels récursifs). Mise à jour de la queue de la liste des emplacements associés à un mot dans la structure `mot_data_t`. Ajout d'instructions d'affichage via la macro `DEBUG`.

Ajout d'un fichier source supplémentaire parmi les sources du projet (`free.c`) dédié à la libération des blocs mémoire, afin de permettre une centralisation de la gestion mémoire.