TP5: Bug hunting

Module ArcSys

Objectifs pédagogiques :

- Comprendre les limitations du debug sans outil (E1) ;
- Apprendre à utiliser l'outil gdb (E2, E4) ;
- Utiliser des outils pour corriger des bugs (E2, E3, E4, E6) ;
- Savoir utiliser des outils pour découvrir de nouveaux bugs (E3, E5, E6).

L'objectif de ce TP est de vous donner des pistes sur comment trouver les bugs dans vos programmes.

La recherche d'une erreur au sein d'un programme est une sorte de jeu de pistes où l'on recherche des informations sur le contexte, les symptômes, les causes possibles de l'erreur. Cela permet de déterminer sa localisation et la manière de la corriger. La méthode traditionnelle consistant à utiliser la commande `printf` en divers endroits du programme est l'expression de cette recherche d'information. Des outils tels que `gdb` et `valgrind` facilitent l'obtention d'informations sur les programmes.

★ Exercice 1: la méthode printf

À télécharger : `http://people.irisa.fr/Martin.Quinson/Teaching/ArcSys/bug-boom.c`

Cette méthode est utilisée dans les cas où on ne peut (ou ne veut) pas utiliser de debugger. Attention cependant au piège classique de cette méthode, mis en valeur dans le programme `bug-boom.c` ci-contre (également dans le dépôt).

bug-boom.c

```
#include <stdio.h>                  1
                                    2
int main() {                        3
   int *p;                          4
                                    5
   printf("1");                     6
   p = NULL;                        7
   printf("2");                     8
   *p = 1;                          9
   printf("3");                     10
                                    11
   return 0;                        12
}                                   13
```

Ce programme devrait afficher `12Erreur de segmentation` puisque la ligne 9 revient à déréférencer le pointeur NULL, ce qui est interdit.

▷ **Question 1:** Quel est l'affichage généré par ce programme ?

C'est parce que les affichages de printf ne sont pas toujours réalisées immédiatement. Pour des raisons de performances, le système cherche en effet à retarder les affichages de façon à avoir moins d'action d'affichage pour plus de texte à chaque fois. C'est pourquoi les "1" et "2" sont placés dans un tampon pour être affichés plus tard. Malheureusement, comme l'erreur de segmentation de la ligne 9 tue brutalement le programme, ces messages ne seront jamais affichés.

Les `printf` suggèrent donc une localisation erronée du problème, ce qui peut faire perdre un temps considérable. Plusieurs solutions permettent d'éviter ou au moins de contrôler cette mise en tampon.

▷ **Question 2:** Ajoutez des retours-chariots à la fin des affichages (la ligne 6 devient `printf("1\n");`). Quel est maintenant l'affichage de votre programme ? Et si vous lancez votre programme de la façon suivante : `./boom|less` ?

C'est parce que le système vide le tampon à chaque fin de ligne si et seulement l'affichage est dirigé sur un terminal.

▷ **Question 3:** Retirez les \n que vous aviez ajouté à la question précédente, et demandez à réaliser les affichages sur la sortie d'erreur (en utilisant `fprintf(stderr, "...")` à la place de printf. Quel est maintenant le comportement de votre programme ? Et si la sortie n'est pas un terminal mais un tube ?

C'est parce que la sortie d'erreur n'est pas mise en tampon, car les messages d'erreurs sont considérés urgents et doivent être affichés au plus vite, même si cela induit une perte de performances.

▷ **Question 4:** Rechangez vos affichages pour utiliser la sortie standard (avec `printf`), et ajoutez des `fflush(stdout)` après chaque `printf`. Quel est maintenant le comportement de votre programme ? Et si la sortie n'est pas un terminal mais un tube ?

C'est parce que la fonction `fflush` a pour objectif de pour vider le tampon et forcer l'affichage immédiat des informations.

Conclusion. Cet exercice nous a permis d'explorer le principal piège de la mise au point à base de `printf`. Nous avons vu 3 façons de contourner ce piège, mais cette méthode reste artisanale, et il est souvent nécessaire d'utiliser des outils spécialisés comme `gdb`.

★ Exercice 2: le debugger GNU : gdb (utilisation de base)

Nous utiliserons comme premier exemple le programme `bug-boucle.c` ci-contre, à télécharger en ligne : `http://people.irisa.fr/Martin.Quinson/Teaching/ArcSys/bug-boucle.c`

bug-boucle.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int *tab = NULL;
5
6  void initialise(int n)
7  {
8    char i = 0;
9
10   for (i = 0; i <= n; i++);
11     {
12        tab[i] = 1;
13     }
14 }
15
16 int main()
17 {
18   printf("Debut\n");
19   tab = malloc(10000*sizeof(int));
20   initialise(10000);
21   printf("Fin\n");
22
23   return 0;
24 }
```

Pour le compiler, il convient d'utiliser la commande `gcc -g -o boucle bug-boucle.c`. `-g` ajoute au binaire produit les informations de deboguage nécessaire à `gdb` (et autres debuggers).

▷ **Question 1:** Exécutez ce programme. Que constatez vous ?

Lancement de gdb. Tapez la commande `gdb ./boucle` pour charger votre programme dans l'environnement GDB. On contrôle ce programme en tapant des commandes à l'invite. Les commandes les plus importantes sont `help`, `list`, `quit` et `run`.

▷ **Question 2:** Essayez la session suivante dans gdb :

- Chargez boucle dans gdb et lancez le programme.
- Tapez <ctrl+c> pour interrompre votre programme.
- Visualisez le code en cours d'exécution avec `list`.
- Reprenez l'exécution avec `cont`, puis interrompez-la de nouveau. L'exécution n'a pas progressé.
- Aidez le programme à franchir la zone difficile à l'aide de la commande `jump 11`, ce qui fait sauter l'exécution à la ligne 11 (oui, cela modifie l'exécution du programme). Le programme doit se terminer normalement. Reste à comprendre pourquoi le programme ne passe pas la ligne 10 seul.

Points d'arrêt et exécution pas à pas

Lors de la traque d'une erreur, il est fréquent d'avoir une idée de sa localisation potentielle. `gdb` permet donc de spécifier des points d'arrêt dans le code où l'exécution est automatiquement interrompue. La commande `break` suivie d'un nom de fonction ou d'un numéro de ligne (éventuellement associé à un fichier) insère un point d'arrêt à l'endroit spécifié. `clear` supprime le point d'arrêt spécifié.

Placez un point d'arrêt sur la fonction `main` puis lancez l'exécution. Elle s'interrompt avant le début du code. Expérimentez avec les commandes `next` et `step`. Chacune permet d'avancer l'exécution d'une ligne puis de bloquer l'exécution. Si cette ligne contient un appel de fonction, `step` entre dans le code de cette fonction tandis que `next` l'exécute en entier et passe à la ligne suivante de la fonction courante.

▷ **Question 3:** Pour trouver le problème, interrompez au besoin votre programme (ctrl-C), utilisez la commande `print` pour afficher le contenu de la variable i (`print i`). Vous pouvez également le faire continuer (commande `continue`), et le réinterrompre. Corrigez le problème.

Indice : ce premier bug se trouve ligne 8.

▷ **Question 4:** Maintenant que le programme s'exécute jusqu'à la fin, on constate que l'affichage d'une des cases de `tab` après un arrêt à la ligne 21 indique que l'affectation du tableau ne s'effectue pas correctement, puisqu'elles valent 0 au lieu du 1 attendu. Réexécutez votre programme pas à pas pour comprendre le problème, puis corrigez le.

Indice : ce second bug se trouve ligne 10.

★ Exercice 3: La suite d'outils valgrind

`valgrind` est une suite d'outil fabuleuse pour mettre au point vos programmes. Selon l'outil utilisé, il est possible de détecter la plupart des problèmes liés à la mémoire (outil `memcheck`), d'étudier les effets de cache pour améliorer les performances (avec `cachegrind`), de débugger des programmes multi-threadés (avec `hellgrind`, voir le cours de système en 2A) ou encore d'optimiser les programmes (avec `callgrind`). Nous allons nous intéresser au premier outil, que l'on lance avec `valgrind --tool=memcheck ./prog`

▷ **Question 1:** Lancez valgrind sur le programme `boom` étudié plus tôt. S'affichent de nombreuses lignes commençant par `==<identifiant du processus>==`. Elles sont le fait de `valgrind`.

La cause de l'erreur de segmentation est donnée par le second groupe de ligne :

```
1 ==7585== Invalid write of size 4
2 ==7585==    at 0x40051E: main (bug-boom.c:9)
3 ==7585==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

À la ligne `bug-boom.c:9`, nous écrivons 4 octets (sans doute un entier) à un endroit invalide. En effet, l'adresse `0x0` [où nous tentons d'écrire] n'est ni sur la pile, ni le résultat d'un `malloc` et il n'a pas été free()é récemment. Bien sûr ! La ligne 9 écrit à l'adresse pointé par `p`, mais `p` vaut la valeur `NULL`, qui n'est pas une adresse valide (et on a `NULL=0x0`). `valgrind` localise immédiatement et précisément le problème.

▷ **Question 2:** Lancez maintenant `valgrind` sur le programme `boucle` (après avoir corrigé les deux bugs identifiés dans l'exercice 2). Vous pouvez constater que le programme que l'on croyait corrigé contient encore des problèmes :

```
==7532== Invalid write of size 4
==7532==    at 0x400571: initialise (bug-boucle.c:12)
==7532==    by 0x4005AE: main (bug-boucle.c:20)
==7532==  Address 0x51dd0c0 is 0 bytes after a block of size 40,000 alloc'd
==7532==    at 0x4C29BBE: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==7532==    by 0x40059D: main (bug-boucle.c:19)
```

La ligne `bug-boucle.c:12` tente d'écrire 4 octets à un endroit invalide. De plus, cet endroit est localisé juste après un gros bloc mémoire alloué en `bug-boucle.c:19`. Corrigez ce problème (indice : le bug est en ligne 10).

▷ **Question 3:** Relancez `valgrind` sur le programme `boucle`. À la fin de l'exécution, valgrind affiche :

```
==21800== LEAK SUMMARY:
==21800==    definitely lost: 0 bytes in 0 blocks
==21800==    indirectly lost: 0 bytes in 0 blocks
==21800==      possibly lost: 0 bytes in 0 blocks
==21800==    still reachable: 40,000 bytes in 1 blocks
==21800==         suppressed: 0 bytes in 0 blocks
==21800== Rerun with --leak-check=full to see details of leaked memory
```

Il y a donc un bloc de mémoire (de 40 ko) obtenu par `malloc`, mais jamais restitué au système avec `free`. Ajoutez les options nécessaires pour voir lequel et corrigez le problème.

★ **Exercice 4: le debugger GNU : gdb (utilisation avec les fonctions) (optionnel)**

À télécharger : `http://people.irisa.fr/Martin.Quinson/Teaching/ArcSys/bug-fact.c`

Nous allons maintenant utiliser le debugger avec un autre programme afin d'expérimenter les opérations permettant de trouver les problèmes impliquant des fonctions.

Pile et cadres La commande `backtrace` permet d'afficher la pile d'exécution du processus. Compilez ***bug-fact.c*** (page suivante et dans le dépot) puis chargez fact dans `gdb`. Spécifiez un point d'arrêt sur la ligne 9 (x=1) et lancez l'exécution. Lorsque le processus est stoppé, exécutez `backtrace`.

La liste affichée indique tout d'abord les appels récursifs à fact et termine par main. Les fonctions sont donc listées depuis l'appel le plus imbriqué (regardez la valeur indiquée pour le paramètre n de f pour chaque cadre) vers l'appel le moins imbriqué (donc dans l'ordre inverse de l'ordre chronologique, d'où le nom de la commande).

Chaque ligne constitue ce que l'on appelle un *cadre de pile* (« frame » en anglais). Il est possible de se déplacer dans la pile avec les commandes `up` et `down`, ou directement avec la commande `frame` suivie du numéro de cadre visé.

Affichage de variables et d'expressions La commande `print` permet d'afficher le contenu d'une variable. Placez un point d'arrêt sur `fact` puis ré-exécutez. Utilisez `print n`. La commande `disp` est similaire, mais affiche le résultat à chaque interruption du programme. Exécutez `disp (char)n+65` puis utilisez `cont` plusieurs fois.

On peut de plus modifier des valeurs avec `set variable VAR=EXP` où VAR est le nom de la variable à modifier et EXP l'expression dont le résultat est à lui affecter. Si le nom de la variable à modifier n'entre pas en conflit avec les variables internes de `gdb`, on peut omettre le mot-clé variable.

Conclusion sur gdb. Vous en savez maintenant assez sur `gdb` pour faire vos premiers pas. Il existe cependant de nombreuses fonctionnalités que nous n'avons pas abordé ici comme les *watchpoints* (qui arrêtent l'exécution quand une variable donnée est modifiée), le chargement de fichiers `core` pour débugger un programme après sa mort, la prise de contrôle de processus en cours d'exécution, et bien d'autres encore. `info gdb` pour les détails.

bug-fact.c

```
#include <stdio.h>

int fact(int n) {
   int x = 0;

   if (n > 0) {
      x = n * fact(n - 1);
   } else {
      x = 1;
   }

   return x;
}

int main() {
   int a = 10;
   int b =  0;

   b = fact(a);
   printf("%d!=%d\n", a,b);

   return 0;
}
```

★ **Exercice 5: L'utilisation des warnings est conseillée pour la santé (optionnel)** Les compilateurs peuvent fournir un ensemble d'avertissements durant la compilation afin de pointer du doigt des bugs potentiels dans votre programme. Il s'agit d'avertissements, car cela n'empêche pas de générer un exécutable selon les standards du C. Par défaut, aucun warning n'est activé. Cependant, il est recommandé de toujours activé **au minimum** ces deux options : `-Wall -Wextra`

Le nom de ces options nous laisseraient penser qu'ils activent tout les avertissements de l'univers, mais ce n'est pas le cas. `-Wall` active les avertissements facile à résoudre et qui ont peu de chance de se révéler être de

faux positifs, afin de limiter la gêne de l'utilisateur. `-Wextra` va en activer un peu plus mais risque dans certains cas de gêner sans raison.

D'autres options existent mais ne sont pas activés par ces deux options et se révèlent très utiles pour détecter des erreurs dans un programme. Voici un sous-ensemble non exhaustif [1] :

```
-Wunreachable-code -Wwrite-strings -Wcast-align -Wformat=2 -Wformat-security
-Wformat-nonliteral -Wpointer-arith -Wmissing-include-dirs -Wmissing-declarations
-Wmissing-prototypes -Wsign-conversion -Wunused-macros -Wswitch-bool -Wundef
-Wredundant-decls -Wlogical-op -Wdouble-promotion -Wbool-compare
-Wlogical-not-parentheses
```

Prenons l'exemple d'un programme avec un bug (provenant du TP3). On sait que dans l'exemple ci-dessous crashera à l'exécution :

```
1  static void edit(char *s) {
2      s[0] = 'R';
3      s[1] = 'u';
4      s[2] = 's';
5      s[3] = 't';
6  }
7  int main(void) {
8      edit("C   ");
9
10     return 0;
11 }
```

Cela est dû au fait que la chaîne constante `"C   "` est mise dans une section du programme où il est interdit d'écrire [2]. `-Wall -Wextra` ne permettent pas de détecter ce type d'erreur, l'ajout de `-Wwrite-strings` est nécessaire.

L'exercice serait donc de commencer à utiliser ces options avec votre compilateur pour chaque projet C que vous avez en cours (et j'espère par le futur). En plus de repérer des erreurs cela vous apprendra probablement de nouvelles choses sur la magie du C !

Si durant votre compilation il y a des avertissements que vous considérez comme faux, dans un premier temps posez-vous bien la question si cela est un faux positif, et si oui vous pouvez le retirer.

★ **Exercice 6: Il n'y a pas que gcc (optionnel)** GCC n'est pas le seul compilateur libre et open source.

Un de ses célèbres compétiteurs est clang (initialement il s'agissait d'un projet de recherche universitaire !). clang est en majeur partie compatible en terme d'options sur la ligne de commande avec gcc. Tout ce que vous faisiez avec gcc vous pouvez le faire avec clang.

Ici on va s'intéresser à des options que clang a été le premier à offrir : les *sanitizers*.

Ces *sanitizers* ont pour but d'instrumenter (ajouter des instructions supplémentaires) le programme afin de détecter durant l'exécution des comportements indéfinis, une mauvaise utilisation de la mémoire ou encore l'oublie de libérer de la mémoire allouée dynamiquement. Si vous êtes curieux, voici une liste non exhaustive des *sanitizers* de clang :

— `https://clang.llvm.org/docs/AddressSanitizer.html`
— `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
— `https://clang.llvm.org/docs/MemorySanitizer.html`

Attention, comme il s'agit d'une instrumentation il y peu y avoir un coût important en performance, c'est pourquoi on utilise cela que en phase de développement.

clang fournit aussi l'outil `scan-build` : `https://clang-analyzer.llvm.org/scan-build.html` Il permet de lancer un analyseur statique sur votre code source qui va permettre de détecter des bugs potentiels dans celui-ci. Il génère un rapport HTML que vous pouvez lire avec votre navigateur. Voici une liste des vérifications effectuées : `https://clang-analyzer.llvm.org/available_checks.html`

Comme l'exercice précédent, tentez d'utiliser les *sanitizers* et `scan-build` avec vos projets C (ou C++), vous découvrirez peut être des bugs et pensez à les utiliser à l'avenir, cela pourrait vous faire gagner quelques heures de debug.

1. Visitez `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html` pour connaître leurs significations et pourquoi ils n'ont pas été ajouté à `-Wall`
2. Pour être plus précis, toute tentative de modification d'un objet constant est un comportement indéfini (voir `https://en.wikipedia.org/wiki/Undefined_behavior`)