L3-MI / PROG
TP#6 : Valgrind & ASan

Pierre Karpman

2024-10-09

Présentation

Le but de ce TP est de vous familiariser avec deux outils d'analyse dynamique et un outil d'analyse statique, Valgrind, ASan et T-Snippet. Ces outils permettent de détecter un certain nombre de bugs (notamment liés à la gestion et aux accès mémoire), et sont complémentaires en ce qu'ils peuvent détecter des bugs difficilement (ou non-) détectables par les autres. Nous les présentons ici très brièvement.

Valgrind

Valgrind [1] est un analyseur dynamique capable de détecter des bugs d'accès mémoire illégaux, de lecture de mémoire non initialisée, de désallocation multiple, etc.. Le principe de base de Valgrind consiste à exécuter le programme que l'on souhaite analyser dans un environnement virtualisé qui maintient ses propres informations sur l'espace mémoire et son contenu.

Valgrind s'utilise typiquement en :

1. compilant son programme avec un niveau d'optimisation « raisonnable » (par ex. `-O1`) et les options facilitant le débuggage (typiquement `-g -fno-omit-frame-pointer`);
2. remplaçant l'invocation du programme `$ prog <options>` (ici depuis un shell) par `$ valgrind prog <options>`.

ASan

L'*AdressSanitizer* "ASan" [2] est un analyseur dynamique capable de détecter des bugs d'accès mémoire illégaux, d'accès à de la mémoire déjà désallouée, etc.. Le principe de base d'ASan consiste à modifier (à « instrumenter ») le programme lors de la compilation afin d'ajouter un environnement de détection et des tests associés.

ASan s'utilise typiquement en :

1. compilant son programme avec un niveau d'optimisation « raisonnable » (par ex. `-O1`) et les options facilitant le débuggage (typiquement `-g -fno-omit-frame-pointer`);
2. ajoutant `-fsanitize=address` aux options de compilation (pour un compilateur compatible, par ex. les versions suffisamment récentes de clang ou gcc), à la fois pour la production de fichiers objets `.o` et pour la phase de *link*.

Par ailleurs, il existe également un *UndefinedBehaviorSanitizer* "UBSan" de fonctionnement similaire, qui s'utilise en ajoutant `-fsanitize=undefined` aux options de compilation.

1. https://valgrind.org/
2. https://github.com/google/sanitizers/wiki/AddressSanitizer

T-Snippet

T-Snippet[3] est un analyseur statique s'utilisant depuis un navigateur web. Il permet aisément d'étudier des fragments de code (des "snippets"), et est spécialisé dans la détection d'UBs. Il fournit aussi une version « développée » du code (la *Analyzer view*) qui permet de le rendre plus explicite, et de détecter du « code mort » (du code qui ne sera *jamais* exécuté).

Comparaison

On peut comparer (très brièvement) ces analyseurs en soulignant les aspects suivants :

— Certains bugs sont mieux détectés avec un outil en particulier. Par exemple ASan ne détecte pas la lecture de mémoire non initialisée, mais il détecte certains accès mémoire illégaux mieux que Valgrind (notamment les « petits » dépassements de tableau).

— ASan est assez rapide (il dégrade peu les performances du programme instrumenté) et s'intègre facilement au processus de compilation. Il nécessite cependant un accès aux sources du programme.

— Le test d'un programme sous Valgrind peut être assez lent, mais est possible pour tout exécutable sans nécessiter un accès aux sources ou une recompilation ; il n'est pas non plus restreint aux programmes écrits en C ou C++.

— Le caractère statique de T-Snippet lui permet de détecter aisément des problèmes qui ne se manifesteraient pas forcément à l'exécution, et l'Analyzer view peut être utile pour comprendre ce que fait « réellement » le programme. L'utilisation est cependant limitée à de petits fragments de code.

Conseils

De façon générale (hors du cadre spécifique de ce TP), on peut formuler les conseils suivants : 1) systématiquement utiliser ASan et UBSan lors du processus de développement (sans attendre la présence visible d'un bug) ; 2) en cas de bug observé non détecté par ASan/UBSan, utiliser Valgrind (mieux vaut dans ce cas désactiver ASan et UBSan) ; 3) T-Snippet est particulièrement approprié pour analyser un fragment de code que vous trouvez suspect.

Exercice 1

Commencez par extraire le programme `rf.c` de l'archive https://miashs-www.u-ga.fr/~davidjer/progl3mi/tp6/pc2024_tp6.tar.bz2.

Q.1 :

1. *Sans exécuter le programme*, pouvez-vous prévoir quel problème risque de survenir à l'exécution ?
2. Vérifiez votre hypothèse en exécutant le programme et en utilisant Valgrind et/ou ASan et/ou gdb.

Exercice 2

Commencez par extraire le programme `mul.c` de l'archive, compilez-le et exécutez-le. Celui-ci est manifestement incorrect.

3. https://tsnippet.trust-in-soft.com/

Q.1 :

1. Trouvez et corrigez le bug dans la fonction `test`.

Q.2 :

1. Trouvez et corrigez le bug dans la fonction `mul64`. Vous pouvez pour cela vous aider de l'Analyzer view de T-Snippet pour détailler ce que fait cette fonction.
2. Vérifiez que votre fonction corrigée passe les tests pour une valeur « raisonnable » de son argument (par exemple 2^{14}).
3. Écrivez une nouvelle fonction de test similaire à `test`, qui teste la correction de `mul64` sur des entrées aléatoires.

Q.3 :

1. À quel algorithme vu en cours (et déjà implémenté !) pourriez-vous rapprocher la fonction `mul64` ?

Exercice 3

Commencez par extraire le programme `minesweeper.c` de l'archive.

Q.1 : En utilisant Valgrind, ASan et UBSan, trouvez et corrigez les (au moins) 6 bugs présents dans ce programme. (On conseille de tester le programme avec (entre autres) des petites tailles de grille "*mapsize*", par exemple 3.)

Exercice 4

Commencez par extraire le programme `prpnbs.c` de l'archive, et compilez-le tout d'abord sans options particulières.

Q.1 :

1. Lancez le programme successivement avec les arguments suivants : 100, $1\,000$, $10\,000$, $100\,000$, $1\,000\,000$, pour la fonction `prpnbs`.
2. Faites de même (avec argument 100) en utilisant une fois Valgrind et une fois ASan. Expliquez les messages d'erreur, et corrigez le bug.

Q.2 :

1. Lancez le programme avec argument $3\,000\,000\,000$ (si vous avez suffisamment de mémoire). Que se passe-t il ?
2. Faites de même avec UBSan, et corrigez le bug (sans changer la signature de la fonction).

Q.3 :

1. Quel est l'algorithme antique implémenté par ce programme, et que fait il ?
2. Expliquez l'intérêt des fonctions `getm2` et `setm2`.
3. Comment pourrait-on diviser l'espace mémoire utilisé par ce programme par un facteur 32 (pour de « grandes » valeurs de `n`) ?
4. Faites une analyse de coût (on pourra utiliser $H_n := \sum_{k=1}^{n} 1/k = \Theta(\log(n))$).