

Softwaretechnik Übung: Tools für Speicherzugriffsfehler

Klaus Kusche

1. Echte Speicherzugriffsfehler:

Schreib ein paar C-Programme (oder C++-Programme), in die du bewusst verschiedene Speicherzugriffsfehler einbaust (double **free**, **free** eines Pointer mitten in ein Objekt, Nullpointer-Zugriff, Zugriff auf ein schon freigegebenes lokales oder dynamisches Objekt, Zugriff über die Arraygrenzen bei lokalen, globalen und dynamischen Arrays, uninitialized Variablen oder Pointer, zu lange Eingabe bei einem String, **strcpy** mit zu langer Quelle, ...).

Ich stelle auf meiner Webseite auch fehlerhafte Programme zur Verfügung.

Übersetze diese Programme mit **clang** oder **gcc** mit Address Sanitizer (**-fsanitize=address**), und lass sie laufen.

Lass zum Vergleich die "normal" (ohne Prüfungen!) übersetzten Programme unter Valgrind mit den Plugins **memcheck** und **sgcheck** laufen.

Welche Fehler werden erkannt? Wie brauchbar sind die Fehlermeldungen?

Zum Address Sanitizer ("Asan"):

- Die Funktionalität ist auf drei Teile aufgespalten:

Der eigentliche **AddressSanitizer** soll ungültige Speicherzugriffe erkennen, der **LeakSanitizer** soll nicht mehr freigegebene dynamische Speicherbereiche finden und der **MemorySanitizer** (nur **clang**, nicht **gcc**) soll das Lesen gültiger, aber uninitialisierter Speicherbereiche erkennen.

Unabhängig vom Asan gibt es den **ThreadSanitizer** zur Analyse problematischer paralleler Speicherzugriffe und den **UndefinedBehaviorSanitizer** des **gcc** zur Erkennung von Operationen, die in C / C++ einen undefinierten Effekt haben.

- Die **gcc**- und **clang**-Option **-fno-common** ist nötig, damit Asan auch globale Variablen und Arrays richtig prüft.

-fsanitize-address-use-after-scope erweitert den erzeugten Code beim **clang** um Prüfungen auf Zugriffe auf schon freigegebene lokale Variablen.

Mit **-fno-omit-frame-pointer** enthalten die Fehlermeldungen für Speicherfehler mehr Funktionsaufrufs-Information.

- **ulimit -d unlimited** ist für den Asan nötig, damit er seine 16 TB große Speicher-Gültigkeits-Tabelle anlegen kann.
- Das Verhalten von Asan zur Laufzeit wird durch die Environment-Variable **ASAN_OPTIONS** kontrolliert, z.B.
ASAN_OPTIONS=strict_string_checks=1:detect_stack_use_after_return=1:check_initialization_order=1:strict_init_order=1

Fangen Programme, die mit **gcc** und einer der **-fstack-protector**-Optionen übersetzt wurden, auch etwas ab?

Prüfe die Programme auch *statisch* mit dem statischen Checker von **llvm/clang** (siehe Befehle **scan-build** und **scan-view** !) sowie mit **gcc** und **llvm/clang** mit maximalen Warn- und Optimierungs-Optionen.

Wie der viele Probleme werden erkannt? Wie hilfreich ist der Output?

2. Memory Leaks:

Schau dir an, wie man den in der Glibc *eingebauten* Memory Leak Tracer (https://www.gnu.org/software/libc/manual/html_node/Allocation-Debugging.html) verwendet, und probiere ihn mit einem kleinen Programm aus, das dynamischen Speicher anlegt und freigibt (oder auch nicht!).

Lass dasselbe Programm auch

- mit **gcc** oder **clang** plus *LeakSanitizer*,
- mit **valgrind**
- mit dem *Heap Checker der Google Perftools* laufen.

Womit arbeitet es sich am angenehmsten?

Werden dieselben Probleme gefunden? Welcher Output hilft am besten?