Debugging: GDB und clang-sanitizer


C ist Liebe,
C ist Leben,
C ist segmentation fault, core dumped

Ausblick

Heute lernen wir ...

- was es außer taktisch platzierte *printf* noch für Möglichkeiten der Fehlersuche gibt
- wie ihr gezielter die Ursache einer *Segmentation Violation* findet
- wie ihr subtil falsche Speicherzugriffe findet
- wie ihr Speicherlecks findet

Debugger

Was ist ein Debugger?

- Ein Programm zum Inspizieren anderer Programme zur Laufzeit
  - Programmfluss verfolgen
  - Werte von Speicherbereichen anzeigen
  - Code in das Programm hinzufügen
  - ...
- Extrem nützlich!

Der GDB

- **G**NU **D**e**b**ugger
- Der "standard"-Debugger unter Linux
- Alt (Ursprung 80er Jahre)
- Viel Funktionalität
- TUI-Programm (die meisten IDEs haben Frontends)

GDB installieren

- macOS:

```
$ brew install gdb
```

- Ubuntu/WSL:

```
$ sudo apt install gdb
```

- Fedora:

```
$ sudo dnf install gdb
```

Programm für Debugging vorbereiten

- Eure Programme müssen sog. *Debug-Symbole* enthalten
- Funktionsnamen von internen Funktionen, Zeilen-Nummern, Variablen-Namen etc.
- Es geht auch ohne, ist aber (sehr) schwer
- Ihr müsst Flags and den Compiler übergeben:

```
1 $ cc hallo.c -g -O0 -o hallo
```

Die wichtigsten GDB-Befehle

- *Breakpoint* setzen:

```
(gdb) break main
```

- Ausführung fortsetzen:

```
(gdb) c
```

- Zur nächsten Zeile gehen:

```
(gdb) step
```

- Variable ausgeben:

```
(gdb) p die_variable
```

- Auch mit Dereferenzierung:

```
(gdb) p *der_zeiger
```

Die wichtigsten GDB-Befehle

- Sog. *Backtrace* ausgeben:

```
(gdb) bt
```

- Funktion im Backtrace anwählen:

```
(gdb) frame 5
```

- Liste lokaler Variablen ausgeben:

```
(gdb) info locals
```

- Funktionsargumente ausgeben:

```
(gdb) info args
```

- GDB beenden:

```
(gdb) exit
```

GDB-Demo

Demo: Verkettete Listen

Weiterführende Themen

- GDB ist sehr mächtig, sprengt den Rahmen
- *Remote-Debugging*, *Zeitreisen-Debugging*, *Pretty-Printer*, *Postmortem-Debugging*, Skripte...
- Komplette GDB-Doku via `$ info gdb` oder auf `https://sourceware.org/gdb/current/onlinedocs/gdb.html/`

Zwischen-Zusammenfassung

- Debugger helfen euch dabei, gezielt nach der Ursache eines Fehlers zu suchen
- Programm muss mit den Flags `-g -O0` gebaut werden
- Spicker für häufige Befehle:
  https://gabriellesc.github.io/teaching/resources/GDB-cheat-sheet.pdf

Einschub

Projekt-Zwischenreview

Sanitizer

Was sind Sanitizer?

- Werkzeuge, die bei automatischer Fehlersuche in eurem Programm helfen
- Helfen dabei, auch subtile Fehler zu finden
- Oft in Compilern integriert
- Schwerpunkt auf Speicherfehler

Die Clang-Sanitizer

- Bauen auf dem *clang*-Compiler von Apple auf
- Ihr müsst euer Programm mit *clang* statt *cc* bauen
- Nur auf Linux vollständig unterstützt
- Viele verschiedene, euch interessieren erstmal:
  - *AddressSanitizer* : Verbotene Speicherzugriffe, Speicherlecks
  - *UndefinedBehaviorSanitizer* : Undefiniertes Verhalten
  - *MemorySanitizer* : Nutzung von nicht initialisierten Speicher

Clang-Sanitizer einsetzen

- *clang* mit entsprechenden Flags nutzen:

```
$ clang -g -O0 -fsanitize=address,undefined -o main main.c
```

- Bei mehreren Dateien: Auch beim linken angeben!

```
$ clang -fsanitize=address -o main main.o a.o b.o
```

- Dann Programm wie gewohnt ausführen
- *AddressSanitizer* und *MemorySanitizer* sind nicht kompatibel!

Sanitizer-Demo I

Demo: Ungültiger Schreibzugriff

Sanitizer-Demo II

Demo: Nutzung nach Ablaufen der Lebzeit

Sanitizer-Demo III

Demo: Speicherleck

Sanitizer-Demo IV

Demo: Subtiler Fehler

Zusammenfassung

- Sanitizer helfen euch, vor allem Speicherfehler zu finden
- Finden subtile, oft "symptomfreie" Fehler
- Machen das Programm langsamer, kosten euch aber relativ wenig Zeit
- Weiterführende Dokumentation: `https://clang.llvm.org/docs/index.html`