Debugging

Variables

Lecturers: Mirko Jantschke, Pascal Scholz

28. Januar 2019

Contents

Einleitung

It’s not a bug...

Es gibt verschiede Arten von Bugs:

- Compiletime errors
- Runtime errors (*bugs*)

Compiletime errors: Fehler die beim Compilieren auftreten. Können relative einfach behoben werden, da der Compiler auf sie hinweist.

Bugs: Treten während der Laufzeit (runtime) des Programms auf. Sie sind schwer zu finden und könnne schlimme Folgen haben.

… it’s a feature.

Mögliche Auslöser von Bugs:

- Over- /underflow von Variablen
- Division durch Null
- Unendliche Schleifen / Rekursion
- Range excess
- Segmentation fault
- Dereferenzierung von *NULL pointers*
- und viele mehr

Tools für das Debugging

Nützliche und weitverbreitete Tools die zum Debugging eingesetzt werden sind:

- Der Compiler
- valgrind
- GDB
- strace (werden wir nicht ausprobieren, nur Hinweis auf Existenz)
- ...und viele mehr

Im folgenden sollen die Möglichkeiten der genannten Tools kurz angerissen werde.

Debugging mit dem Compiler - am Beispiel der GCC

Nützliche Flags des GCC

Nützliche und weitverbreitete Tools die zum Debugging eingesetzt werden sind:

- -Wunsed (zeigt ungenutze Variable, gilt als Warnung)
- -Wall (zeigt [fast]alle Warnung)
- -Werror (behandelt Warnung als Fehler, so dass nicht compiliert wird)
- -g (default Debugging Informationen)
- und viele mehr, siehe man gcc

Ein Test am Beispiel

Versucht einmal, das Programm der linked list wie folgt zu compilieren:

```
$ gcc -Wall -Werror <Dateiname>.c
```

Wenn ihr den Code nicht mehr habt, kopiert ihn euch hier. Die Ausgabe sollte wie folgt aussehen (wenn es eine Warnung gibt):

```
[                                    ]$ gcc -Wall -Werror dynmamicMemeory.c
dynmamicMemeory.c: In function ‘main’:
dynmamicMemeory.c:29:7: error: unused variable ‘x’ [-Werror=unused-variable]
   int x;
       ^
```

Vorbereitungen für Runtime Debugging

Vorbereitung

Normal kompilierten Programmen beinhalten sinnvollerweise keine bzw. wenige Debugginginformationen. Diese blähen die ausführbare Datei auf und bieten zusätzliche Angriffsfläche, da mit ihnen Reverse-Engeneering möglich ist.

Um Programme mit Debugging-Informationen zu compilieren:

```
$ gcc -g <Dateiname>.c
```

Falls *GDB* noch nicht installiert ist, sollte dies noch getan werden:

```
$ sudo pacman -S gdb         #arch
$ sudo apt-get install gdb   #ubuntu
```

Debugging mit GDB

GDB - Was ist das?

Der GDB oder GNU Debugger ist ein freies und sehr mächtiges Debugging-Werkzeug. Wenn sich ein Programm nicht wie erwartet verhält, kann man mit ihm dessen Abarbeitung Schritt für Schritt nachvollziehen, um herauszufinden, wo im Quelltext Fehler sein könnte.

Generell kann er für alle Fehlerarten (während der Laufzeit) eingesetzt werden.

GDB Befehle

- Wenn gdb ohne ein Programm gestartet wurde kann mit **file** *file_name* eine geladen werden.
- Mit **r[un]** kann das Programm innerhalb von gdb ausgeführt werden.
  Als ersten Schritt sollte das Programm so einmal abgearbeitet werden, um genauere Details zum Fehler zu erhalten. Alternativ kann mit **start** zum Einstiegspunkt (erste Zeile in main) gesprungen werden.
- Haltepunkte (Breakpoints) können mit **b[reak]** *line_number* oder **b[reak]** *function_name* gesetzt werden.
  An der vermuteten Stelle des Bugs im Quelltext sollte begonnen werden.
- Mit **p[rint]** *identifier* können Werte angezeigt werden.
- **w[atch]** *identifier* unterbricht das Programm und gibt den Wert von *identifier* aus, sobald diese Variable geändert wird.

...wenn man den Breakpoint dann einmal erreicht hat.

- **n[ext]** führt nur die nächste Zeile des Prrgramms aus.
- **s[tep]** fürht die nächste Anweisung aus.
- Um das Programm bis zum nächsten Breakpoint ausführen zu lassen, kann **c[ontinue]** genutzt werden.
- Mit **backtrace** oder **bt** können die aufgerufenen Funktionen angezeigt werden.
- Um den Quelltext auszugeben ab eine bestimmten Zeile, kann **l[ist] Startzeilennummer** genutzt werden.

- Wenn nur die *Enter-Taste* gedrückt wird, wird der zuletzt eingegebene Befehl ausgeführt.

Bedingte Haltepunkte (breakpoints)

Nachdem ein Breakpoint gesetzt wurde, wird ihm von GDB eine ID zugewiesen.
Diese ID kann genutzt werden, um die Funktionalität von diesem zu erweitern.

- **con[dition]** *breakpoint_ID expression* fügt dem Breakpoint eine Bedingung hinzu:

```
(gdb) br 42
Breakpoint 1 at 0xbada55: file main.c, line 42.
(gdb) condition 1 i@=@@=@3
```

- Strings müssen definiert werden, bevor sie mit **strcmp** genutzt werden können:

```
1 (gdb) br main.c:42
2 Breakpoint 13 at 0xdeadbeef: file main.c, line 42.
3 (gdb) set $string_to_compare = "lolwut"
4 (gdb) cond 13 strcmp ( $stringtocompare, c ) @=@@=@ 0
```

Eine kleine Zusammenfassung

file	Lädt Programm
r[un]	Führt ein Programm aus
start	Führt die erste Instruktion eines Programms aus
b[reak]	Setzt einen Breakpoint
p[rint]	Gibt Variable aus
w[atch]	Bei Veränderung unterbreche und gebe Variable aus.
n[ext]	Führe nächste Zeile aus und unterbreche
s[tep]	Führe nächsten Befehl aus und unterbreche
c[ontinue]	Führt ein Programm bis zum nächsten Breakpoint aus
l[ist]	Gibt Quelltext aus
backtrace / bt	Gibt Aufrufhierarchi der Funktionen aus
q[uit]	Beendet GDB (und das darin ausgeführte Programm)

...und ein Beispiel

Im Beispiel-Programm ist ein kleiner Fehler versteckt. Hier eine Idee, wie man vorgehen könnte:

- Führt das Programm innerhalb von GDB einmal aus, um zu schauen, was der Fehler ist (es passiert nichts, beendet das Programm. Eventuell gibt es einen Hinweis..)
- Mit **s[tep]** kann das Programm schrittweise abgearbeitet werden, wenn man keine Idee hat, wo man anfangen kann. (Bei großen Programmen eventuell unpraktisch)
- Hat man eine verdächtige Stelle gefunden, bietet es sich an den Quelltext mit **l[ist] Startzeilennummer** anzusehen.
- Falls nötig entsprechende Variablen und deren Veränderung mit **p[int] idendifier** ansehen.

EIN HINWEIS: Mit der Tastenkombination `strg+C` kann man laufende Programme (im Terminal) beenden.
Bitte nicht auf die nächste Folie schauen, da steht die Lösung.

Die Auflösung

```
1 int power(int base, int exponent) {
2     int result = 1;
3     int count  = 0;
4     while(count < exponent) % {
5         result *= base;     %
6         ++count;            % }
7     return result;
8 }
```

Wir haben hier eine Endlosschleife. Auf den ersten Blick ist aber alles in Ordnung und sollte funkionieren. Aber warum tut es das nicht?

Es fehlen die geschweiften Klammern nach dem while-Statement. Es wird nur result berechnet, nicht aber count inkrementiert.
$\rightarrow$ Ein Klammernpaar nach dem while-Statement, das nach Zeile 6 geschlossen wird, löst das Problem.

Debugging mit Valgrind

Was ist Valgrind

Valgrind ist ein Framwork, welches verschiede Werkzeuge anbietet. Beispielsweise bietet es Werkzeuge um Bugs im Speicher- oder Threadmangemant zu finden. Wir befassen uns nur mit dem Speichertool, mit welchem wir selbst Fehler in Programmen aufdecken können, die es nicht zum Absturz bringen.

Weitere Informationen zu Valgrind können auf der offiziellen Seite gefunden werden.

Den Memory check von Valgrind verwenden

Nachdem das Programm mit Debugginginformationen kompiliert wurde, kann man valgrind starten und das Programm als Argument übergben:

```
$ valgrind <Programmname>
```

Um weiter Informationen über Memory-Leaks zu erhalten:

```
$ valgrind --leak-check=full <Programmname>
```

Wenn man nun noch mehr über die Quelle der Leaks erfahren möchte:

```
$valgrind --leak-check=full --track-origins=yes <Prog.-name>
```

Mehr Informationen zur Verwendung können auf der offiziellen Seite von Valgrind gefunden werden.

Wir probieren es aus...

Wir haben im letzten Kurs ein Programm mit Speicherbugs geschrieben. Dieses sollten wir eventuell noch fertigstellen, damit ihr ein gutes Beispiel habt.

Die Rede ist von der verketteten Liste. Falls ihr den Code nicht mehr habt, findet ihr in hier.
Nun versuchen wir den Fehler zu finden

Möglicher Qutput

```
==15028== HEAP SUMMARY:
==15028==     in use at exit: 68 bytes in 5 blocks
==15028==   total heap usage: 6 allocs, 1 frees, 1,092 bytes allocated
==15028==
==15028== 4 bytes in 1 blocks are definitely lost in loss record 1 of 5
==15028==    at 0x483777F: malloc (vg_replace_malloc.c:299)
==15028==    by 0x109227: main (dynmamicMemeory.c:28)
==15028==
==15028== 64 (16 direct, 48 indirect) bytes in 1 blocks are definitely lost in l
oss record 5 of 5
==15028==    at 0x483777F: malloc (vg_replace_malloc.c:299)
==15028==    by 0x1091D6: append (dynmamicMemeory.c:21)
==15028==    by 0x10923C: main (dynmamicMemeory.c:30)
==15028==
==15028== LEAK SUMMARY:
==15028==    definitely lost: 20 bytes in 2 blocks
==15028==    indirectly lost: 48 bytes in 3 blocks
==15028==      possibly lost: 0 bytes in 0 blocks
==15028==    still reachable: 0 bytes in 0 blocks
==15028==         suppressed: 0 bytes in 0 blocks
==15028==
==15028== For counts of detected and suppressed errors, rerun with: -v
==15028== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Heap Summary

```
==15028== HEAP SUMMARY:
==15028==     in use at exit: 68 bytes in 5 blocks
==15028==   total heap usage: 6 allocs, 1 frees, 1,092 bytes allocated
==15028==
==15028== 4 bytes in 1 blocks are definitely lost in loss record 1 of 5
==15028==    at 0x483777F: malloc (vg_replace_malloc.c:299)
==15028==    by 0x109227: main (dynmamicMemeory.c:28)
==15028==
==15028== 64 (16 direct, 48 indirect) bytes in 1 blocks are definitely lost in l
oss record 5 of 5
==15028==    at 0x483777F: malloc (vg_replace_malloc.c:299)
==15028==    by 0x1091D6: append (dynmamicMemeory.c:21)
==15028==    by 0x10923C: main (dynmamicMemeory.c:30)
```

- Statistiken zum Auf dem Heap allokierten Speicher
- Backtracing der Funktionsaufrufe in denen Speicher verlorengegangen ist

Leak Summary

```
==15028== LEAK SUMMARY:
==15028==    definitely lost: 20 bytes in 2 blocks
==15028==    indirectly lost: 48 bytes in 3 blocks
==15028==      possibly lost: 0 bytes in 0 blocks
==15028==    still reachable: 0 bytes in 0 blocks
==15028==         suppressed: 0 bytes in 0 blocks
```

- Ist die zusammenfassende Statistik
- definitily lost: Speicher auf den von Pointern direkt gezeigt wurde
- indirectly lost: Speicher auf den von Pointer über Pointer indirekt gezeigt wurde.
- Im Falle von Fehlern durch das Verwenden von uninitialisierten Variablen wird zusätzlich gewarnt.

Ebenfalls auf der offiziellen Seite wird noch einmal genau erklärt, was die einzelnen Angaben bedeuten.

Aufgabe

Zum Ausprobieren

Im Beispiel-Programm sind 2 kleine Fehler versteckt, die das Programm zum Absturz bringen. Versucht einmal, diese beiden Fehler mit GDB, valgrind und dem Compiler zu finden. Hier eine Idee, wie man vorgehen könnte:

- Führt das Programm einmal aus, um zu schauen, was der Fehler ist (es sollte abstürzen)
- Lasst euch einen Backtrace anzeigen, um die Funktionsaufrufe nachzuvollziehen.
- Wenn ihr verstanden habt, was nicht funktionieren könnte, lasst euch den Quelltext anzeigen. (Wenn nicht, meldet euch)
- Von hier aus solltet ihr alleine weiter kommen.

Bitte nicht auf die nächste Folie schauen, da steht die Lösung.

Und die Auflösung

Beide Fehler sind in dieser kleinen Funktion:

```
1 void a(int n){
2   if(n = 0)
3     return;
4   a(n);
5 }
```

In main() wird d(), in d() wird c(), in c() wird b() und in d() wird a() aufgerufen. Die Funktion a() ruft sich selbst rekursiv auf. Sie hat folgende Fehler, die zu einer unendlichen Rekursion und damit wegen ausgehenden Stackspeicher zu einem Absturz führen:
In Zeile 2: Kein Vergleich, sondern Zuweisung. Sollte n == 0. sein
In Zeile 4: Kein Dekrementieren um Abbruchbedingungen zu erreichen. Sollte a(n-1) sein.

Noch mehr Bugs...

Im Ordner Materials in diesem Repository befindet sich Ordner 1_* bis 6_* (* ist eine Wildcard) mit verbuggtem Code.

Viel Erfolg...