

Übungen zu Systemprogrammierung 1 (SP1)

B1 – Debugging

J. Schedel, C. Erhardt, A. Ziegler, J. Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2015 – 09. bis 13. November 2015

https://www4.cs.fau.de/Lehre/WS15/V_SP1



Agenda

3.1 Haupteingang nach Walhall

3.2 gdb



Agenda

3.1 Haupteingang nach Walhall

3.2 gdb



- Baukasten von Debugging- und Profiling-Werkzeugen
- Für uns relevant: *memcheck*
 - Erkennt Speicherzugriff-Probleme:
 - Nutzung von nicht-initialisiertem Speicher
 - Zugriff auf freigegeben Speicher
 - Zugriff über das Ende von allozierten Speicherbereichen



■ Zugriffe auf nicht allozierten Speicher finden

```
=711= Invalid read of size 4
=711=      at 0x804841B: main (test.c:19)
=711= Address 0x0 is not stack'd, malloc'd or (recently) free'd
=711=
=711= Process terminating with default action of signal 11 (SIGSEGV)
=711= Access not within mapped region at address 0x0
```

- In Zeile 19 wird lesend auf die Adresse 0x0 zugegriffen
- Der Prozess wird auf Grund einer Speicherzugriffsverletzung (SIGSEGV) beendet

```
=787= Invalid write of size 1
=787=      at 0x48DC9EC: memcpy (mc_replace_strmem.c:497)
=787=      by 0x80485A2: test_malloc (test.c:57)
=787=      by 0x80484A8: main (test.c:22)
=787= Address 0x6d1f02d is 0 bytes after a block of size 5 alloc'd
```

- In Zeile 57 wird `memcpy` aufgerufen, welches ein Byte an eine *ungültige* Adresse schreibt



■ Auffinden von nicht freigegebenem Speicher

```
=787= HEAP SUMMARY:  
=787=      in use at exit: 5 bytes in 1 blocks  
=787=    total heap usage: 1 allocs, 0 frees, 5 bytes allocated
```

- Bei Programmende ist noch ein Speicherbereich (Block) belegt
- Während der Programmausführung wurde einmal `malloc()` und keinmal `free()` aufgerufen
- Mit Hilfe der Option `--leak-check=full --show-reachable=yes` wird angezeigt, wo der Speicher angelegt wurde, der nicht freigegeben wurde.

```
=799= 5 bytes in 1 blocks are definitely lost in loss record 1  
=799=    at 0x48DAF50: malloc (vg_replace_malloc.c:236)  
=799=    by 0x8048576: test_malloc (test.c:52)  
=799=    by 0x80484A8: main (test.c:22)
```

- In Zeile 52 wurde der Speicher angefordert
- Im Quellcode Stellen identifizieren, an denen `free()`-Aufrufe fehlen



■ Auffinden uninitialisierten Speichers

```
=799= Use of uninitialised value of size 4
=799=   at 0x4964316: _itoa_word (_itoa.c:195)
=799=   by 0x4967C59: vfprintf (vfprintf.c:1616)
=799=   by 0x496F3DF: printf (printf.c:35)
=799=   by 0x8048562: test_int (test.c:48)
=799=   by 0x8048484: main (test.c:15)
```

- In Zeile 48 wird auf uninitialisierten Speicher zugegriffen
- Mit Hilfe der Option `--track-origins=yes` wird angezeigt, wo der uninitialisierte Speicher angelegt wurde

```
=683= Use of uninitialised value of size 4
=683=   at 0x4964316: _itoa_word (_itoa.c:195)
=683=   by 0x4967C59: vfprintf (vfprintf.c:1616)
=683=   by 0x496F3DF: printf (printf.c:35)
=683=   by 0x8048562: test_int (test.c:48)
=683=   by 0x8048484: main (test.c:15)
=683=   Uninitialised value was created by a stack allocation
=683=   at 0x804846A: main (test.c:10)
```



- Spezialfall: Zugriff auf uninitialisierten Speicher bei Bedingungsprüfungen

```
=683= Conditional jump or move depends on uninitialised value(s)
=683=   at 0x48DC0E7: __GI_strlen (mc_replace_strmem.c:284)
=683=   by 0x496886E: vfprintf (vfprintf.c:1617)
=683=   by 0x496F3DF: printf (printf.c:35)
=683=   by 0x8048562: test_int (test.c:48)
=683=   by 0x8048484: main (test.c:15)
```



Agenda

3.1 Haupteingang nach Walhall

3.2 gdb

gdb: 2015-05-19



Debugger: gdb

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
 - das Programm schrittweise abarbeiten
 - Variablen- und Speicherinhalte ansehen und modifizieren
 - core dumps (Speicherabbilder beim Programmabsturz) analysieren
 - Erlauben von core dumps (in der laufenden Shell): z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Programm sollte Debug-Symbole enthalten
 - mit GCC-Flag `-g` übersetzen
- Aufruf des Debuggers mit `gdb <Programmname>`

gdb: 2015-05-19



Beispiel

```
void initArray(long *array, unsigned int size) {
    int i;
    for ( i=0; i<=size; i++ ) {
        array[i] = 0;
    }
}

int main(int argc, char *argv[]) {
    long *array;
    long buf[7];
    array = buf;

    initArray(buf, sizeof(buf)/sizeof(long));

    while ( array != buf+sizeof(buf)/sizeof(long) ) {
        printf("%ld\n", *array);
        array++;
    }

    exit(EXIT_SUCCESS);
}
```



- Programmausführung beeinflussen
 - Breakpoints setzen:
 - `b [<Dateiname>:]<Funktionsname>`
 - `b <Dateiname>:<Zeilennummer>`
 - Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
 - Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
 - schrittweise Abarbeitung auf Ebene der Quellsprache mit
 - `s` (step: läuft in Funktionen hinein)
 - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
 - Breakpoints anzeigen: `info breakpoints`
 - Breakpoint löschen: `delete breakpoint#`



- Variableninhalte anzeigen/modifizieren
 - Anzeigen von Variablen mit: `p expr`
 - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
 - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
 - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
 - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
 - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
 - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
 - Anzeigen und Löschen analog zu den Breakpoints

