Linux (user space) Debugging

Topics

- How can we find memory problems?
- Cross debugging using GDB and VS Code
- Debugging after a crash with a core file

Tracing Memory:
Valgrind, ASan & mtrace

C's “Safety”

- C does no memory checking on any of:
  - buffer overflows
  - dangling pointers
  - unfreed memory
  - bad pointers
- Need to use extra tools to instrument your program.
  - Instrumentation:

    ..

Valgrind

- Valgrind: a suit of debugging & profiling tools
  - Runs your application in a virtual CPU,
    doing translations for each instruction.
  - Adds a *significant* performance penalty:
    20 – 30 times slower.
- Detects memory errors:
  - .. (not calling free())
  - .. (use after free)
  - Read/write outside of allocated block
  - ..
- (Does not detect stack memory errors)

Valgrind Install

- Install Valgrind on BBB (requires internet access)
  - Our board’s Valgrind (image 2018-01-28) is broken; so install valgrind from newer Debian release. (dependency incorrect, but valgrind works)

    See debugging guide for details.
- Cross-compile your application with -g option.
- Run Valgrind:

```
(bbg)$ valgrind ./mybadapp
(bbg)$ valgrind --leak-check=full \
     --show-reachable=yes ./mybadapp
```

Valgrind Demo

```
(bbg)$ valgrind --leak-check=full --show-reachable=yes ./memleaker

.. normal program output...

==1503== HEAP SUMMARY:
==1503==     in use at exit: 57,344 bytes in 56 blocks
==1503==   total heap usage: 57 allocs, 1 frees, 58,368 bytes allocated
==1503==
==1503== 57,344 bytes in 56 blocks are definitely lost in loss record 1 of 1
==1503==    at 0x48348EC: malloc (vg_replace_malloc.c:263)
==1503==    by 0x104E7: intToString (memleaker.c:16)
==1503==    by 0x1052B: showConvert (memleaker.c:24)
==1503==    by 0x10573: main (memleaker.c:36)
==1503==
==1503== LEAK SUMMARY:
==1503==    definitely lost: 57,344 bytes in 56 blocks
==1503==    indirectly lost: 0 bytes in 0 blocks
==1503==      possibly lost: 0 bytes in 0 blocks
==1503==    still reachable: 0 bytes in 0 blocks
==1503==         suppressed: 0 bytes in 0 blocks
```

Valgrind Sample

Demo this one.

```
(bbg)$ valgrind ./memabuser
```

- funWithVariables(): uninitialized memory
- funWithHeap(): overflow, double free
- funWithStack(): Misses error!
- funWithPointers(): Misses error!

```
(bbg)$ valgrind --leak-check=full  \
    --show-reachable=yes ./memleaker2
```

- Output part:

```
==1561== 1 bytes in 1 blocks are definitely lost in loss record 1 of 11
==1561==    at 0x48348EC: malloc (vg_replace_malloc.c:263)
==1561==    by 0x10753: main (memleaker2.c:48)
```

Valgrind (cont)

- A well-behaved program should

  ..

  - i.e., should have nothing “still reachable”

- If you forget to call pthread_join() on a thread it leaves some memory un-freed.
  - Should join on all spawned threads or else get:

```
136 bytes in 1 blocks are possibly lost in loss record 1 of 1
 at 0x4832C44: calloc (vg_replace_malloc.c:566)
 by 0x40122CB: _dl_allocate_tls (dl-tls.c:297)
 by 0x4855C73: pthread_create@@GLIBC_2.4 (allocatestack.c:585)
 by 0x108D7:   main (demo_thread.c:36)
```

- Can find *some* stack/globals problems with:

```
(bbg)$ valgrind --tool=exp-sgcheck ./mybadapp
```

  - Does not catch all errors.

Valgrind Errors to Ignore

- Valgrind may find errors which originate in code libraries; you may usually ignore these.

```
==832== 8 bytes in 1 blocks are still reachable in loss record 1 of 8
==832==    at 0x4840AA8: calloc (vg_replace_malloc.c:623)
==832==    by 0x489573B: snd_config_update_r
           (in /usr/lib/arm-linux-gnueabihf/libasound.so.2.0.0)
```

- Turn off -pg flag to remove some warnings.
- If getting errors with __udivmoddi4:

```
==852== Use of uninitialised value of size 4
==852==    at 0x12BB2: __udivmoddi4 (in ./myGoodApp)
```

copy code to target and build on target with its gcc.

Timing Bugs

- Heisenbug
  - A bug which appears/disappears only when you are debugging
- Valgrind significantly changes the runtime performance of your application
  - May cause false timing related bugs related to performance or driving real-time hardware
  - Your code must be threadsafe:
    even if the timing changes significantly, your code must perform the correct computations and steps

Address Sanitizer (ASan)

- GCC and Clang support Address Sanitizer:
  - ..
- Similar to valgrind except
  - It’s fast!
    Only x2 slowdown vs x20
  - It checks more types of errors
  - It requires compile-time change
    (cannot be run on precompiled binary)

ASan catches:

- Use after free
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

ASan use

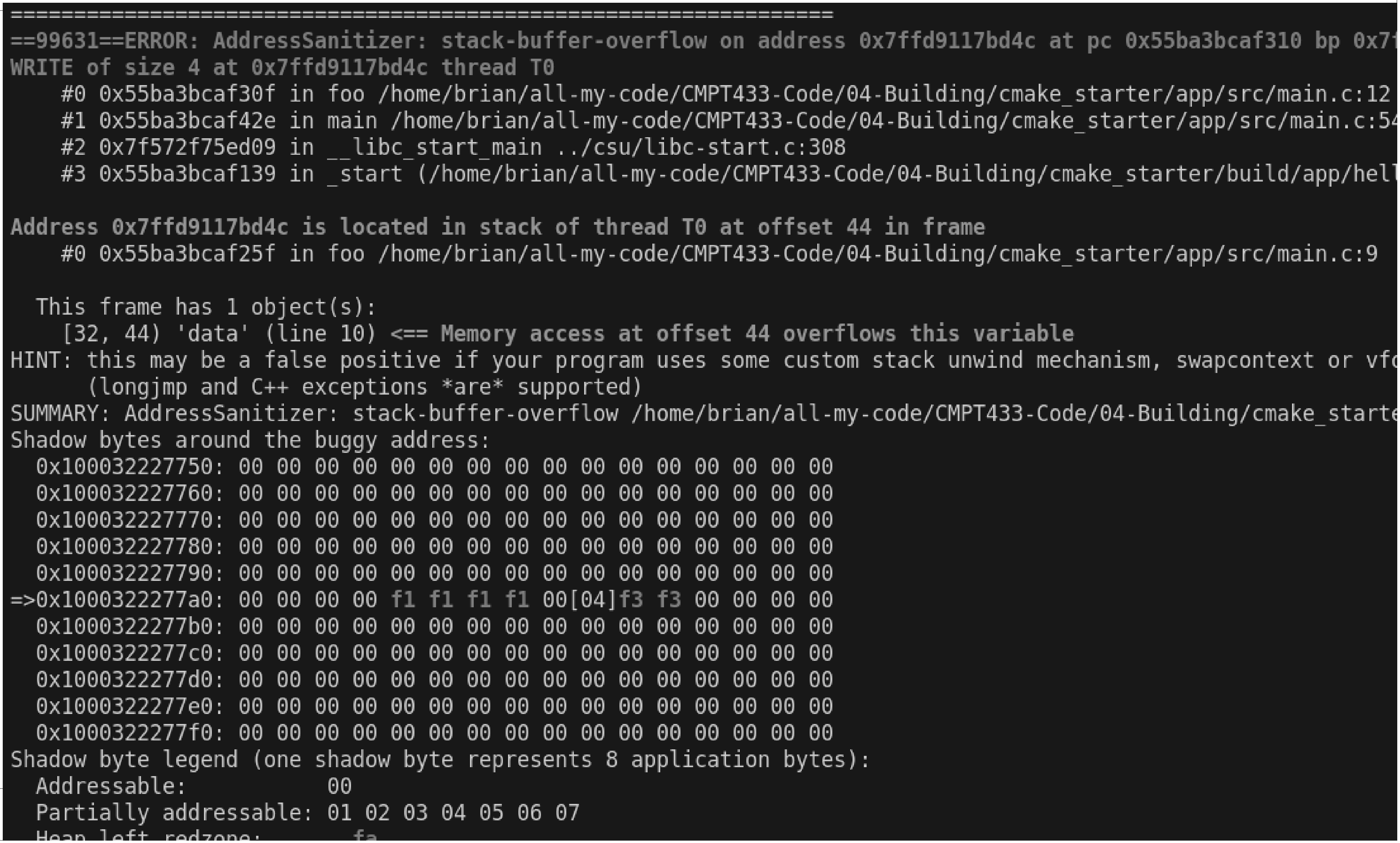
- Enable at compile time in CMakeLists.txt:

```
# Enable address sanitizer
# (Comment this out to make your code faster)
add_compile_options(-fsanitize=address)
add_link_options(-fsanitize=address)
```

- Bad Code

```
void foo() {
  int data[3];
  for (int i = 0; i <= 3; i++) {
    data[i] = 10;
    printf("Val: %d\n", data[i]);
  }
}
```

ASan Error Report



```
=================================================================
==99631==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffd9117bd4c at pc 0x55ba3bcaf310 bp 0x7f
WRITE of size 4 at 0x7ffd9117bd4c thread T0
    #0 0x55ba3bcaf30f in foo /home/brian/all-my-code/CMPT433-Code/04-Building/cmake_starter/app/src/main.c:12
    #1 0x55ba3bcaf42e in main /home/brian/all-my-code/CMPT433-Code/04-Building/cmake_starter/app/src/main.c:54
    #2 0x7f572f75ed09 in __libc_start_main ../csu/libc-start.c:308
    #3 0x55ba3bcaf139 in _start (/home/brian/all-my-code/CMPT433-Code/04-Building/cmake_starter/build/app/hell

Address 0x7ffd9117bd4c is located in stack of thread T0 at offset 44 in frame
    #0 0x55ba3bcaf25f in foo /home/brian/all-my-code/CMPT433-Code/04-Building/cmake_starter/app/src/main.c:9

  This frame has 1 object(s):
    [32, 44) 'data' (line 10) <== Memory access at offset 44 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfo
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/brian/all-my-code/CMPT433-Code/04-Building/cmake_starte
Shadow bytes around the buggy address:
  0x100032227750: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100032227760: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100032227770: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100032227780: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100032227790: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x1000322277a0: 00 00 00 00 f1 f1 f1 f1 00[04]f3 f3 00 00 00 00
  0x1000322277b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000322277c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000322277d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000322277e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1000322277f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
```

mtrace

- If Valgrind's overhead is too high, can use mtrace:
  - ..
- Usage:
  - In C code:

    ```
    #include <mcheck.h>
    void main() {
        mtrace();      // Call to start trace; can be anywhere
    }
    ```

  - On target, set environment variable for trace file:

    ```
    (bbg)$ export MALLOC_TRACE=/tmp/mallocTrace.txt
    ```

  - Run the program (writes mallocTrace.txt):

    ```
    (bbg)$ ./badapp
    ```

  - Analyze results (on host or target):

    ```
    (host)$ mtrace badapp /tmp/mallocTrace.txt
    ```

mtrace example

```
(bbg)$ export MALLOC_TRACE=/tmp/mallocTrace.txt

(bbg)$ ./memleaker
... program's normal operation....

(bbg)$ mtrace ./memleaker ../mallocTrace.txt
- 0x00012008 Free 58 was never alloc'd 0xb6f7495d

Memory not freed:
-----------------
   Address     Size     Caller
0x022ec7e8    0x400  at 0x4b25c9
0x022ecbf0    0x400  at 0x4b25c9
0x022ecff8    0x400  at 0x4b25c9
```

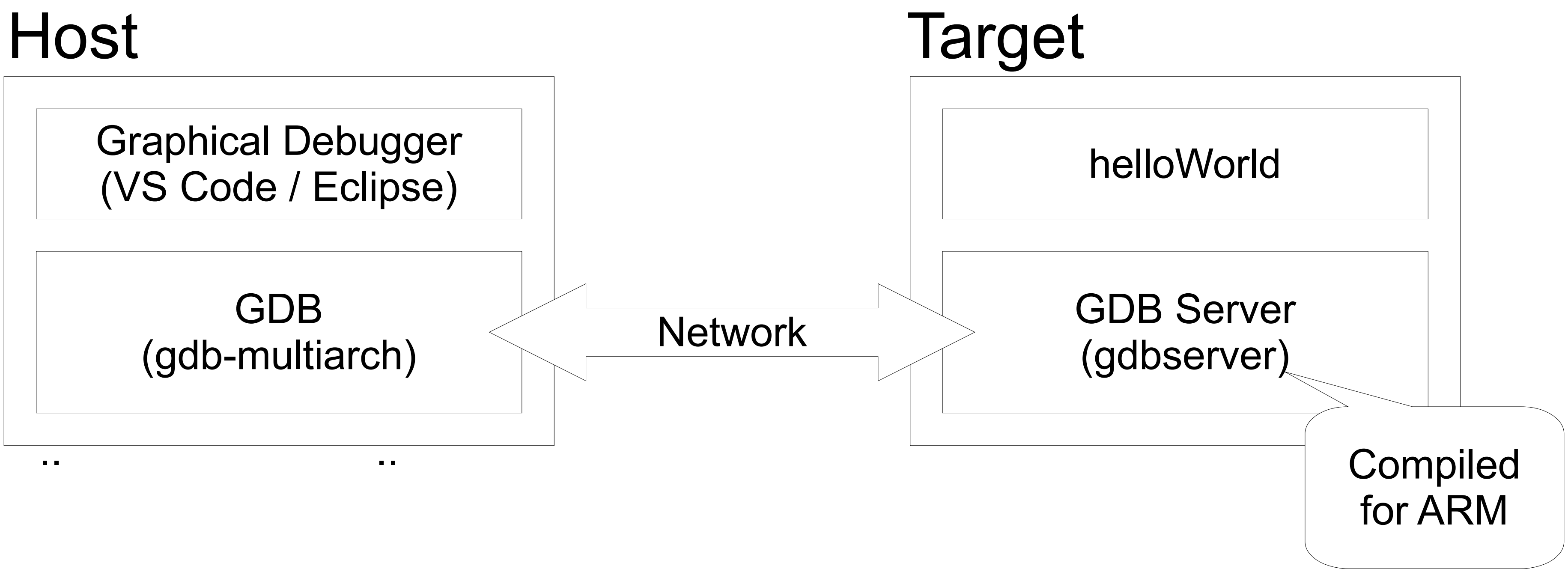
Note: Current BBG image seems not to resolve address to line of code!

GDB

GDB & Debug Symbols

- GDB: GNU debugger
  - Able to read structure of an executable and interactively step through it.
  - ..
    “Symbols” includes:
    - Symbol names: function, variables, parameters
    - Symbol types: return, variable, parameter types
    - File & line numbers for each instruction.
- Build app with debug symbols:
  - GCC: Use -g option:
    `arm-linux-gnueabihf-gcc -g -std=c99 foo.c -o foo`

The Big Picture



- On Target

```
(bbg)$ gdbserver localhost:2001 helloWorld
```

- On Host

```
(host)$ gdb-multiarch -q helloWorld
```

GDB Commands:

- Connect: target remote 192.168.7.2:2001
- View Source:..
- Breakpoints:..
  break main, break test.c:7
- Stepping: run, continue
  step (into), next (over)
  ..
  print <expr>
- Functions: info args, info local,
  ..
- Quit: quit

! Demo badmath.c

VS Code Debugging

- See the Debugging guide for step-by-step on how to setup VS Code (and Eclipse) for cross-debugging.

! Demo VS Code cross debugging badmath.c

Debugging *after* a crash:
Core Dumps

Core Dump

- When a program hits a runtime error, Linux can store its complete state to a core file
  - Enable core file generation:

    ```
    (bbg)$ ulimit -c unlimited
    (bbg)$ ulimit -a                // Display's limit
    ```

  - User can generate core file and send it to developers for later debugging.

Debugging with Core

- Run program on target to generate core file:

```
(bbg)$ ./segfaulter
```

  - When program crashes, it creates a core file in current directory.

May need to run in /tmp if core file is 0 bytes. chhmod a+r on core if cannot read on host.

- Copy to NFS (if not there already)

- On host, open core in cross-debugger:

```
(host)$ cd ~/cmpt433/public/
(host)$ gdb-multiarch ./segfaulter core
```

! Demo: segfaulter.c

Stripping Symbols

- Debug symbols help you debug a program.
- However, they:
  - Make the binary bigger
  - Give away information about your program.
- Can remove the debug symbols after compile:

```
(host)$ cp myApp myApp2
(host)$ arm-linux-gnueabihf-strip myApp2
```

  - Copy myApp2 to target (it's smaller)!
  - When debugging core files generated by a stripped myApp2 on target, can use un-stripped myApp with symbols on host.

Summary

- Tracing memory:
  - Valgrind for a deep check on memory use
  - mtrace for an efficient check on dynamic allocation
- GDB:
  - target runs gdbserver
  - host runs gdb-multiarch
- GDB Commands:
  - target remote, list, info b, b main, continue, bt, step, next, info args, up, down, quit
- Can debug in text or via an IDE
- Debug after a crash with a core file
- Strip a binary to remove symbols