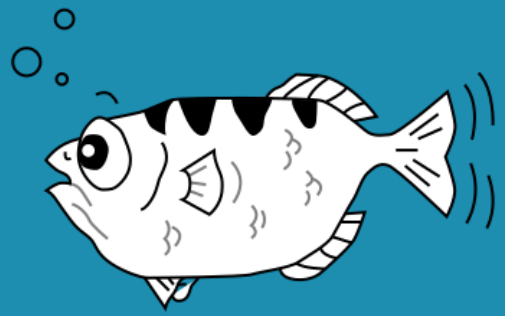
KU LEUVEN

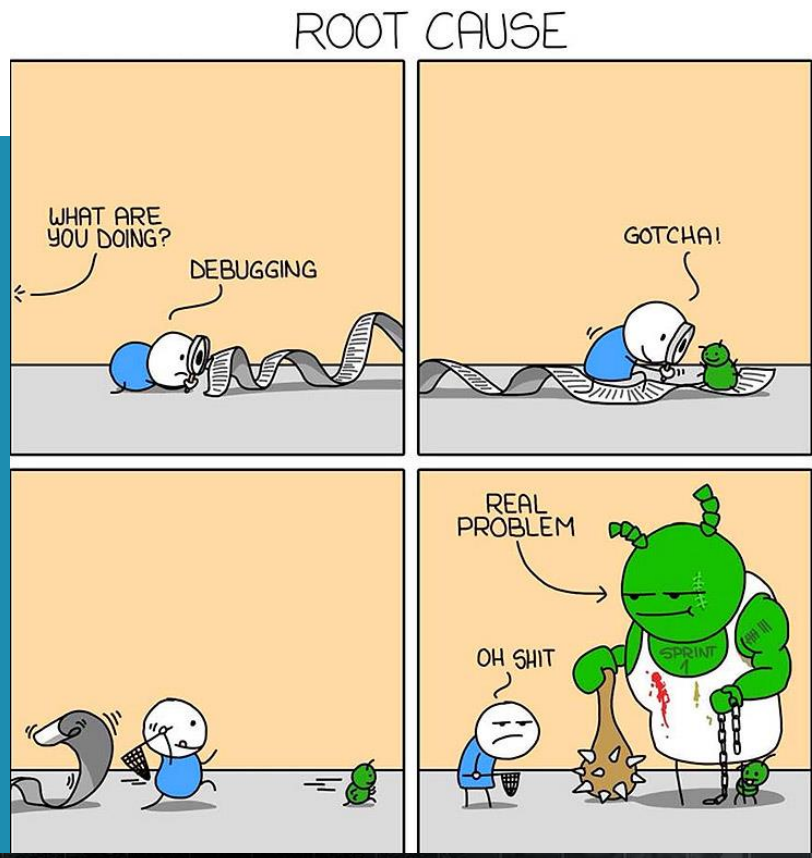


Secure Software Lab 1
Debugging C/C++ with GDB

Ruben Mechelinck

ruben.mechelinck@kuleuven.be

February 2021



Setup (do this before the lab starts!)

- Install VirtualBox or VMWare player
- Get the latest Kali 64-bit VM from:
  https://www.offensive-security.com/kali-linux-vm-vmware-virtualbox-image-download/
- Username & password: kali
- In the VM
  - `sudo apt-get update`
  - `sudo apt-get upgrade` (takes some time!)
  - `sudo apt-get install gdb clang`
  - Install your favorite text editor (e.g. vscode, easiest installation, see: https://code.visualstudio.com/docs/setup/linux#_debian-and-ubuntu-based-distributions

GDB

- GNU Debugger
- No build-in GUI (but there is a text UI)
- Supports: Ada, Go, **C, C++, assembly code, Rust**, Fortran, Pascal, ...

Commands

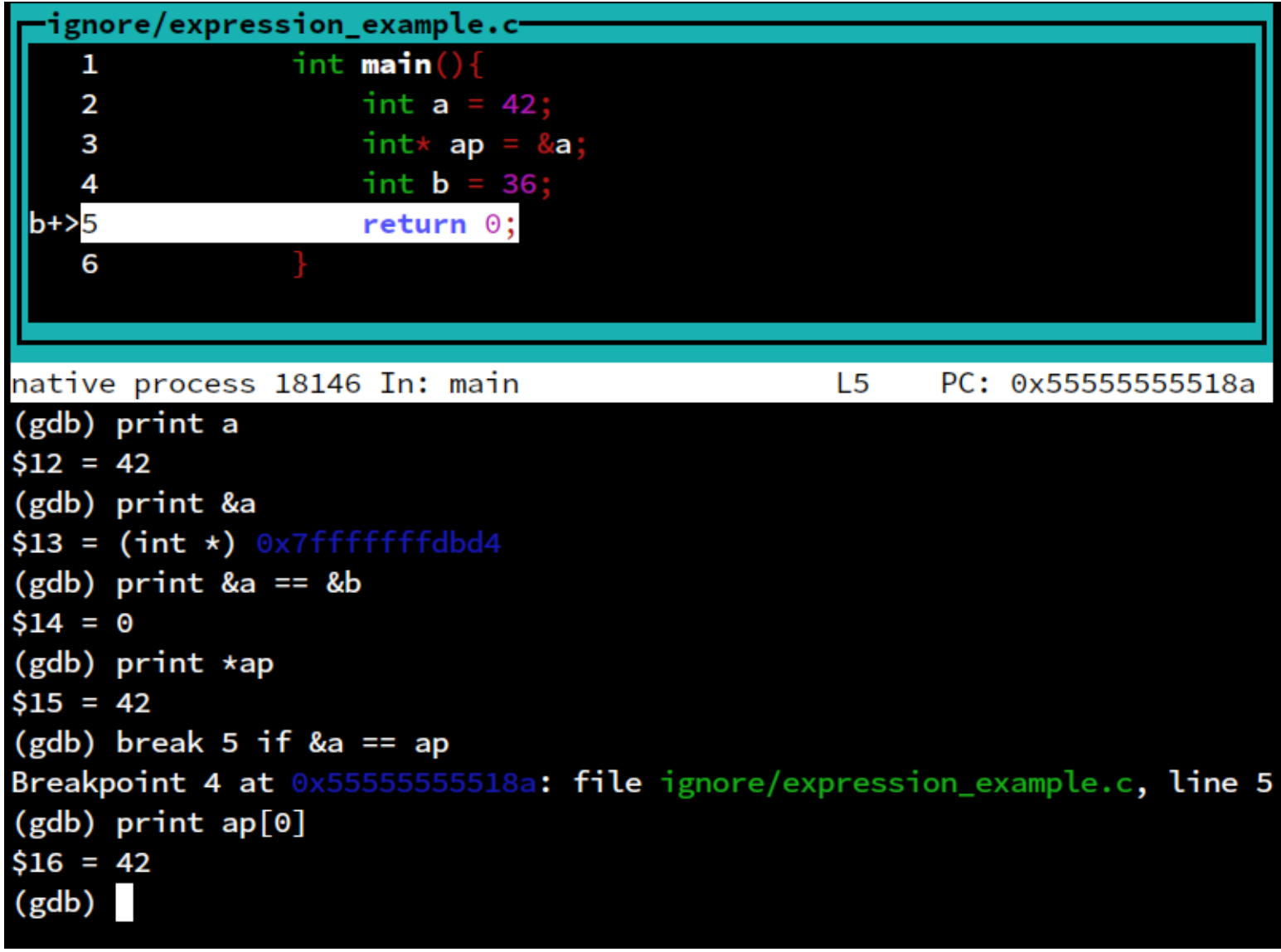
- Compile source code with debug symbols
  - `clang -g <source> -o <program_name>`
- Start the program in GDB with Text UI
  - `gdb -tui --args <program_name> <args...>`
- GDB controls
  - `enter` repeats last command
  - `ctrl+l` refreshes text UI
  - `ctrl+p` previous command in history
  - `ctrl+n` next command in history
  - `ctrl+b` move cursor back in command line
  - `ctrl+f` move cursor forward in command line

GDB Commands

- `run`
  - Start / restart
- `break <line> | <fn> | <mem_addr> | …`
  - Set a breakpoint
- `watch <var_name> | <*mem_location>`
  - Break whenever a variable / value in memory changes
- `info breakpoints`
  - Show all break/watchpoints with their number
- `delete <breakpoint_number>`
  - Delete the break/watchpoint
- `layout asm | src | split`
  - Show the assembly code / source code / both in the text UI
- `next`
  - Run the current **source** instruction, then go to the next
- `nexti`
  - Run the current **assembly** instruction, then go the next
- `step`
  - Step into the current **source** instruction (e.g. step into a called function)
- `stepi`
  - Step into the current **assembly** instruction (e.g. step into a called function)
- `continue`
  - Resume execution until the next break/watchpoint
- `finish`
  - Complete the current function and return to the caller
- `print <var_name> | <*mem_address>`
  - Print the value of a variable / memory location
- `backtrace`
  - Show the call stack
- `up / down`
  - Go up (= examine caller) or down (= examine callee) a stack frame
- `quit`

GDB Commands

- Some commands take an expression in the working language (i.e. C/C++) as an argument, e.g. print ; break if



Lab Instructions

- The following exercises contain **bugs. Find and fix them** (if the source is available)
- For some exercises the **source code file** contains:
  - Examples of working and failing inputs
  - Expected outputs
- Keep the **GDB cheat sheet** at hand!

Exercise 1: Fibonacci Numbers (2 bugs)

- Compile source:
  - `clang -g exercise1.c -o exercise1`
- Run:
  - `./exercise1`
- Useful GDB commands:
  - `break <line_number> | <function_name> | …`
  - `next`
  - `print <variable_name> | <*memory_address>`

Exercise 1

- Tips:
  - The crash (segmentation fault) is a good starting point for debugging: run the program in GDB without any breakpoints and let it crash again, put a breakpoint on the crash address (e.g. `break *0x55555555514f`) or line number and restart the program
  - Now you can examine the variables right before the crash by printing them (e.g. `print fib_seq`)

```
exercise1.c
  7
  8          void calc_fib_seq(int n, int* fib_seq){
>9               fib_seq[0] = 1;
 10              fib_seq[1] = 1;
 11
 12              for(int i = 0; i < n; i++){
 13                  int prev_fib = fib_seq[i-1];
 14                  int prev_prev_fib = fib_seq[i-2];
 15                  fib_seq[i] = prev_prev_fib + prev_fib;
 16              }
 17          }
 18
 19          int main(int argc, char** argv) {
 20              int* fib_seq = 0;
native process 9852 In: calc_fib_seq     L9    PC: 0x55555555514f
(gdb) set style address foreground yellow
(gdb) r
Starting program: /home/ruben/Dropbox/Werk/Onderwijs/Veilige_softw
are/Labo_GDB_feb2021/exercise1

Program received signal SIGSEGV, Segmentation fault.
0x000055555555514f in calc_fib_seq (n=10, fib_seq=0x0)
    at exercise1.c:9
(gdb)
```

Exercise 2: “Encryption" (1 bug)

- Compile source:
  - `clang -g exercise2.c -o exercise2`
- Run:
  - See source file
- Useful GDB commands:
  - `step`
  - `finish`
- Tips:
  - There is no crash, run the whole program step by step from the beginning (e.g. by `break main`), print and check the variables

Exercise 3: Bubble Sort (2 bugs)

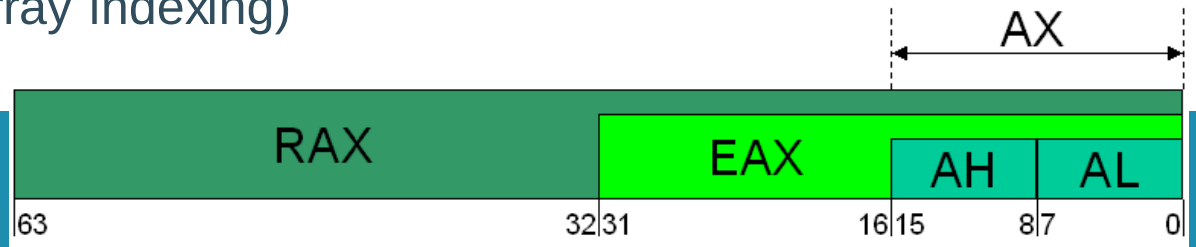
- Compile source:
  - `clang -g exercise3.c -o exercise3`
- Run:
  - `./exercise3`
- Tips:
  - Use the debugger to see what happens to the array and the variables used for indexing them
  - You can print array elements with e.g. `print array[2]`

Exercise 4: Naive String Search (1 bug)

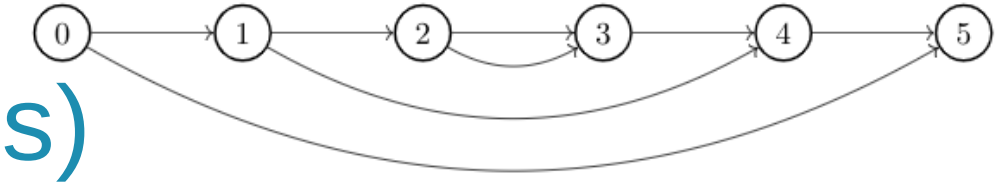
- Compile source (note that no source code is provided for libsearch):
  - `clang -g exercise4.c libsearch.o -o exercise4`
- Run:
  - See source file
- Useful GDB commands:
  - `layout split`
  - `nexti`
  - `stepi`
  - `info register <reg_name>`
  - `break <line_number> | <memory_address> if <condition>`

Exercise 4: Naive String Search (1 bug)

- Tips:
  - Look for the index of the pattern in the text yourself (e.g. vscode has a "col" field in its status bar)
  - Run the outer loop until the loop variable i is at the index of the pattern (e.g. using GDB command: `break <line_number> if i = <index>`)
  - Open the assembly view (`layout asm` or `layout split`), step into the `call` instruction, the arguments 1 to 4 are in edi, rsi, rdx, ecx in that order, look how the arguments are used
  - GDB uses AT&T syntax by default for x86 assembly:
    - `mov %edi,-0x8(%rbp)` --> move the value in register edi to the stack at offset 0x8 from the beginning of the stack frame of this function
    - `movsbl (%rax,%rdx,1),%ecx` --> move 1 byte at address RAX+1*RDX into ECX (e.g. used for array indexing)
  - Remember:



Exercise 5: Graph DFS (2 bugs)



- Compile source:
  - `clang++ -g exercise5.cpp -o exercise5`
- Run
  - `./exercise5`
- Useful GDB commands:
  - `watch <variable_name> | <*memory_address>`
- Tips:
  - If it looks like a value gets magically overwritten, use a watchpoint to find at which program points the variable changes
  - Examine the callers local variables to get more context (using GDB command: `up`)