Debugging

Ion Mandoiu
Laurent Michel

Overview

- Compilation
- Two key tools in the trade
    - gdb
    - valgrind
- Techniques
- Demo
- Reading

Valgrind ?

- Can be found at: http://valgrind.org
- Purpose
  - “Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.”
- six production-quality tools:
  - a memory error detector
  - two thread error detectors
  - a cache and branch-prediction profiler
  - a call-graph generating cache
  - branch-prediction profiler
  - a heap profiler.

Valgrind ?

- Can be found at: http://valgrind.org
- Purpose
  - “Valgrind is an inst … ilding dynamic analysis tools. The … automatically detect many memo … g bugs, and profile your progra … Valgrind to build new tools.”

Our Focus for Now

- six production-quality tools:
  - a memory error detector
  - two thread error detectors
  - a cache and branch-prediction profiler
  - a call-graph generating cache
  - branch-prediction profiler
  - a heap profiler.

Memory Errors

- Context
  - Manual memory management languages (C / C++)
- Classes of errors
  - Leaks
  - Buffer overflows (under/over)
  - Uninitialized memory (read without prior write)
  - Improperly matched calls (a C++ issue)

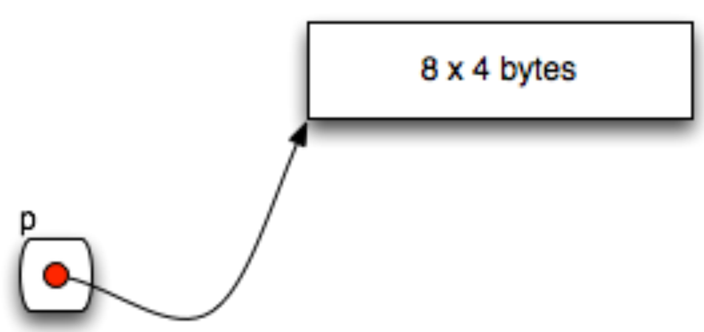
Gdb?

- It won't help directly
  - GDB is about….
    - Control flow errors
    - Simulating the code
- What we need
  - A way to catch the 4 classes
  - That is fast… [low time overhead]
  - That is space efficient… [low space overhead]

Example: Buffer overflows….

- Here is how to catch them….
- Normally when you malloc your mental model is…

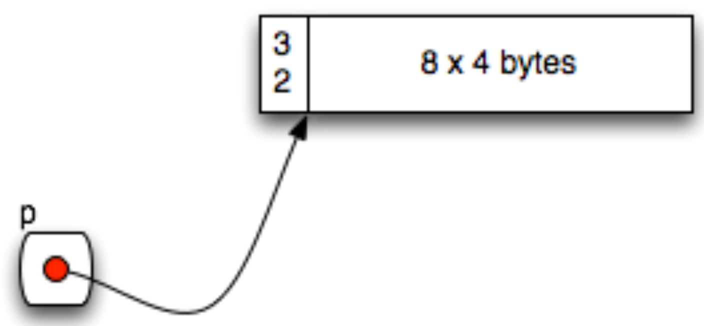
```
int* p = (int*)malloc(sizeof(int)*8);
```



Example: Buffer overflows….

- Here is how to catch them….
- Yet, in reality you get…

```
int* p = (int*)malloc(sizeof(int)*8);
```

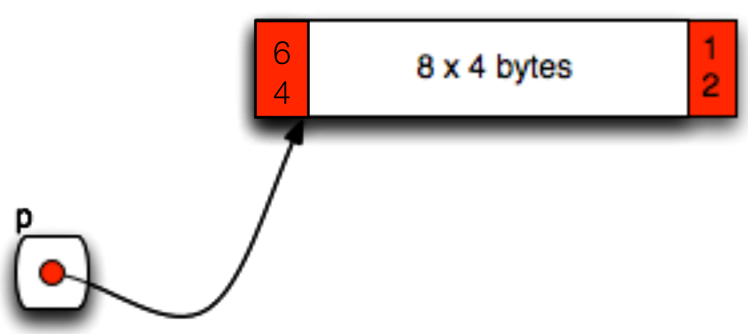


- Why ?

Example: Buffer overflows….

- Here is how to catch them….
- What happens when you overflow or underflow?

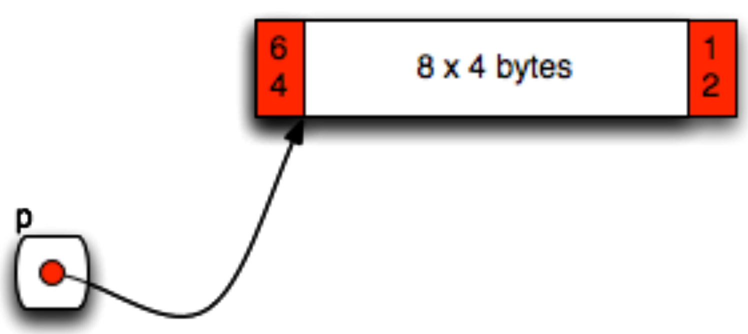
```
int* p = (int*)malloc(sizeof(int)*8);
p[-1] = 64;
p[8]  = 12;
```



Example: Buffer overflows….

- Here is how to catch them….
- What happens when you overflow or underflow?

```
int* p = (int*)malloc(sizeof(int)*8);
p[-1] = 64;
p[8]  = 12;
```

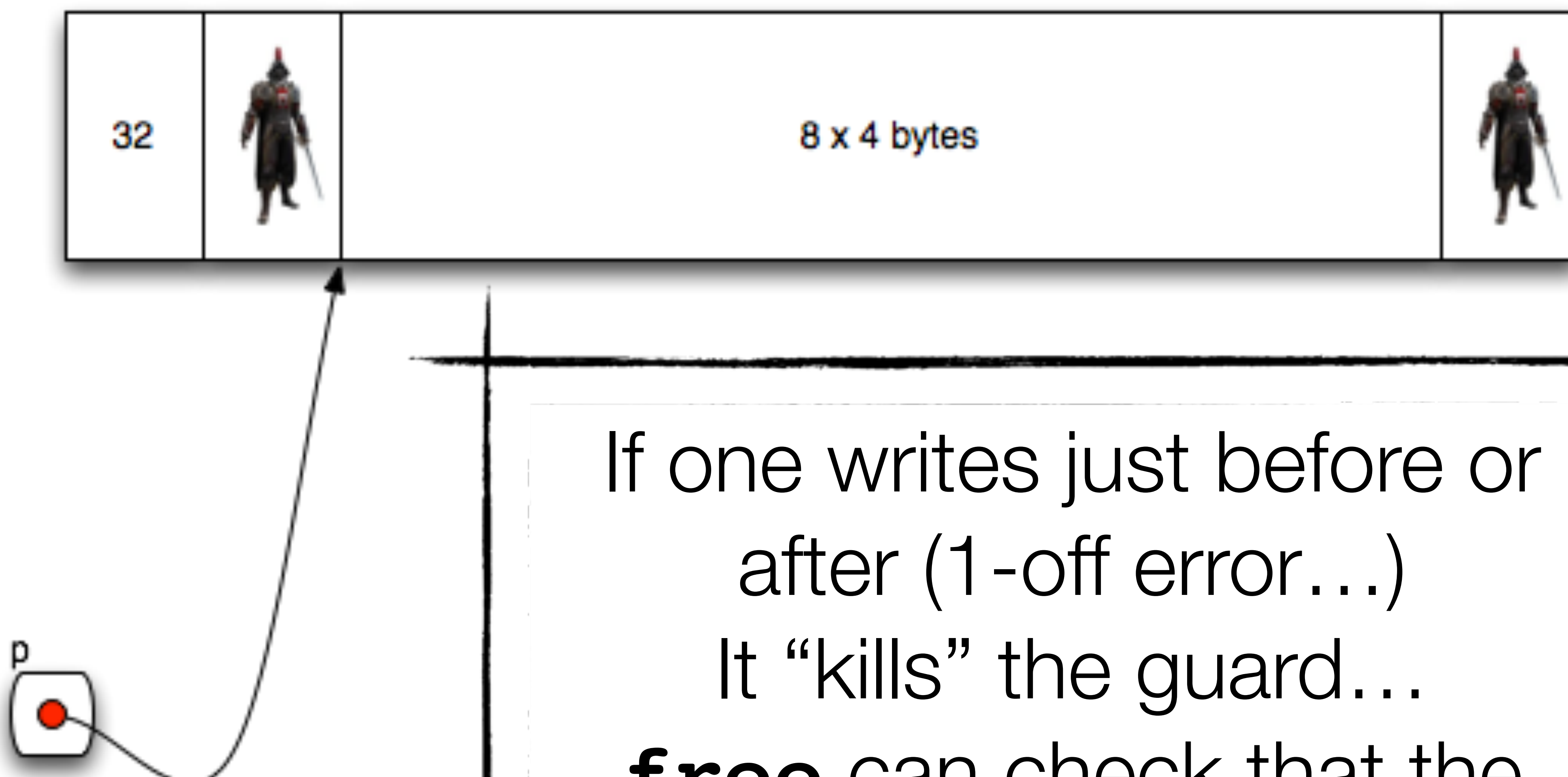


- Try to free now?

```
free(p);
```

How do you catch that ?

- A simple idea….
  - Put Guards around the memory block



If one writes just before or after (1-off error…)
It “kills” the guard…
`free` can check that the guards are present!

How to do that in practice?

- Use a tool that provides a *new* definition of malloc / free
- It *replaces* the stock definition
- It *instruments* free to check the guards
- That *is* what valgrind does!
    - It can catch buffer over and under flow (if buffer was malloc'd)
    - It can catch leaks
    - It can catch uninitialized memory access
    - It can catch misuse of memory APIs (C++ mostly)

Debugging Technique

- Key steps
  - Reproduce the bug
  - Isolate the bug
  - Iteratively
    - Formulate hypothesis
    - Test the hypothesis
    - Whenever test fails, understand why and iterate
    - Whenever test succeed
      - You found it!
      - Revise the implementation

Example Demo

Reading on Valgrind

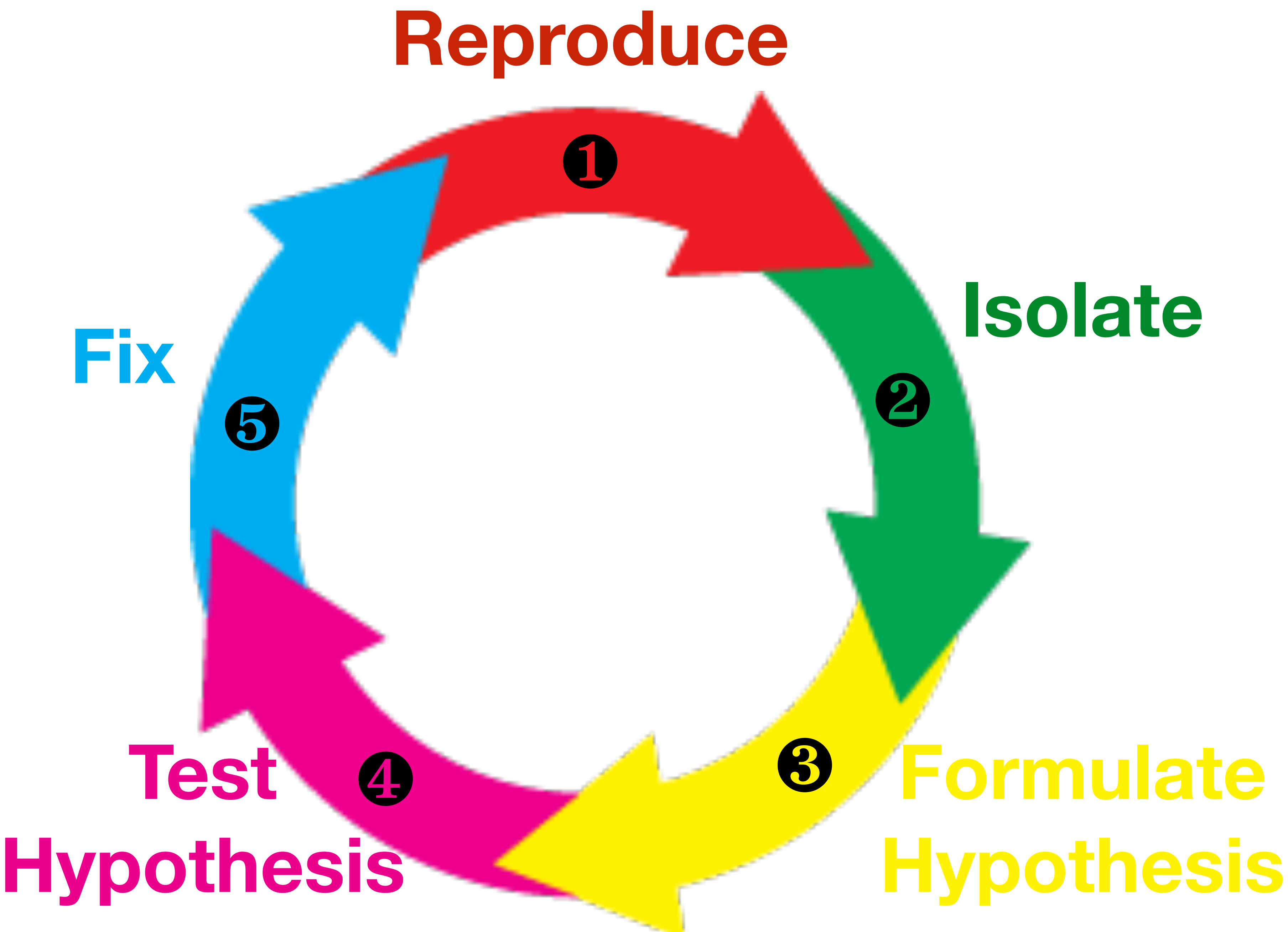
- Chapter 4 of “Developer’s Guide to Debugging”

Debugging Strategies

It is the way!

Caveats

The Process


Reproduce
❶
Isolate
❷
❸
Formulate Hypothesis
Test Hypothesis
❹
Fix
❺

Reproduce

- Debugging is an iterative process
  - First find the **minimal conditions** that *always* trigger the bug
  - That’s the smallest conceivable input
  - That’s the smallest conceivable test program
- Don’t hesitate to change the program to
  - Hard code the input
  - Hard code intermediate results to get rid of irrelevant “paths”
  - Get the smallest input that triggers the bug
  - Eliminate all UI whenever possible
  - The smaller the code, the better!

Isolate

- Once you have 100% reproducibility…
- Fire up the debugger
    - Run the “bad” program on the “bad” input
    - Let it crash
    - Assess
        - Where you are in the code
        - Where you are in the flow
        - What memory state you are in.
        - What is the **symptomatic** cause (not the **underlying** cause)

Formulate Hypothesis

- Based on **symptoms** you must
  - Understand how you got in a pickle to begin with!
  - If you are at the wrong place in the code…
    - A test in the control flow somewhere led you astray
    - Where is it?
  - If your memory state is incorrect
    - Some statement must have written a wrong value.
    - When was that written to ?
- Formulate an hypothetical explanation to answer these questions!

Test Hypothesis

- Once you have an hypothesis…
  - Write down a predicate of the memory state to “catch” the event
  - Add a “breakpoint” or a “watchpoint” to catch the bug in the act
  - Rerun the debug session
- If your break/watch fires….
  - You caught it!
  - Follow the flow to confirm all the way to the crash site
- If the break/watch does not fire….
  - Either your predicate is not expressed correctly
  - Or this was not the cause… try again!

Fix

- Well….
  - Usually (99% of the time) that is the easy part!
- Unless…
  - You are dealing with a design bug.
  - Then you must rethink and…
  - …. be ready to throw away code