Project 1 Q&A (...) Debuggin’

15-440 Recitation 3

Vijay Vasudevan

Carnegie Mellon University

Announcements

- Project 1 due TOMORROW, before class
  - Thanks for showing up to recitation :)
- Project 1 Updates
  - Updated, more robust ruby testers, server reference binary, and server_prot_tester
  - Download them from the webpage!
  - Andrew unhappy with our 6-length password graph experiments, we've toned them down.

Agenda

- Project 1 Q&A

- Leftover time: Debugging

Overview

- What is debugging?
- Strategies to live (or at least code) by.
- Tools of the trade
  - gdb
  - smart logging
  - electric fence

What is debugging?

You tell me! Everybody writes codes with bugs.

What debugging have you needed to do already for password cracking?

Things to think about:

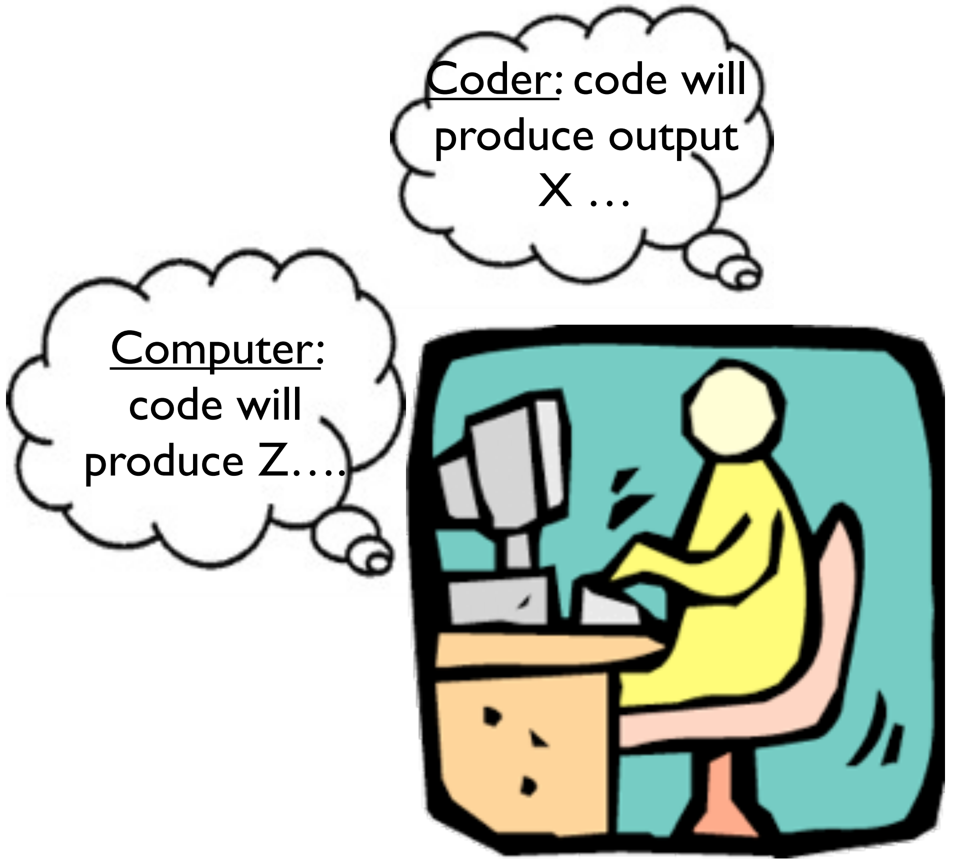
- What caused the bug?
- How did you end up finding it?
- How could you have avoided the bug in the first-place?

Debugging Philosophy

Guiding Steps:

1) Think about why you believe the program should produce the output you expected.

2) Make assertions until you understand how your view differs from the computer's.


Coder: code will produce output X …
Computer: code will produce Z….

Requirements for Debugging

- WHAT program behavior to look for?
  - Sometimes this is nearly free.... (e.g., compiler error, or segfault)
  - Sometimes it is the hardest part.... (e.g., logic bugs, race conditions)
- How to easily expose information to test hypothesis?
  - gdb, logging, strace....

Strategies to Live By...

Debugging is part art,

part science.

You'll improve with experience….

… but we can try to give you a jump-start!

Strategy #1:

Debug with Purpose

Don't just change code and "hope" you'll fix the problem!

Instead, make the bug reproducible, then use methodical "Hypothesis Testing":

```
While(bug) {
    ▪ Ask, what is the simplest input that produces the bug?
    ▪ Identify assumptions that you made about program operation that
      could be false.
    ▪ Ask yourself "How does the outcome of this test/change guide me
      toward finding the problem?"
    ▪ Use pen & paper to stay organized!
}
```

Strategy #2:

Explain it to Someone Else

Often explaining the bug to “someone” unfamiliar with the program forces you to look at the problem in a different way.

Before you actually email the staff...

Write an email to convince them that you have eliminated all possible explanations....

Strategy #3:

Focus on Recent Changes

If you find a NEW bug, ask:

what code did I change recently?

(new code might expose buggy, old code)

This favors:

- writing and testing code incrementally
- using 'svn diff' to see recent changes
- regression testing (making sure new changes don't break old code).

strategy #4:

When in doubt, dump state out

In complex programs, reasoning about where the bug is can be hard, and stepping through in a debugger time-consuming.

Sometimes its easier to just “dump state” and scan through for what seems “odd” to zero in on the problem.

Strategy #5:

Get some distance...

Sometimes, you can be TOO CLOSE to the code to see the problem.

Go for a run, take a shower, whatever relaxes you but let's your mind continue to spin in the background.

Sleep, if your mind is unable to keep spinning.

strategy #6:

Let others work for you!

Sometimes, error detecting tools make certain bugs easy to find. We just have to use them.

Electric Fence or Valgrind:

runtime tools to detect memory errors

Extra GCC flags to statically catch errors:

-Wall, -Wextra, -Wshadow, -Wunreachable-code

Strategy #7: Think Ahead

Bugs often represent your misunderstanding of a software interface.

Once you've fixed a bug:

1) Smile and do a little victory dance....
2) Think about if the bug you fixed might manifest itself elsewhere in your code

   (a quick grep can help).
3) Think about how to avoid this bug in the future

   (maybe coding 36 straight hours before the deadline isn't the most efficient approach....)

Tools of the Trade

Different bugs require different tools:

1) Program crashes with segfault

   -> gdb

2) Hard to reproduce or highly complex bugs

   -> logging & analysis

3) Program hangs waiting for network traffic

   -> tcpdump / ethereal

GDB: Learn to Love it

Run a program, see where it crashes, or stop it in the middle of running to examine program state.

Two ways to run:

- gdb *binary* (to run binary inside of gdb)
- gdb *binary core-file* (to debug crashed program)

GDB Commands

<u>Controlling Execution</u>

- run <cmd-line args>
- break <func>
- step
- next
- control-c

<u>Getting Info</u>

- backtrace
- print <expr>
- info locals
- list
- up/down

GDB Tricks & Tips

- See handout for detailed explanations, and abbreviations
- Remember: always compile with -g (and no optimizations sometimes helps)
- If you're not getting core files, type: 'unlimit coredumpsize' or ulimit -c unlimited

Smart Logging

- Use a debug macro that you can easily turn off to suppress output just by changing one line.

  (example posted online)

  - Don’t litter your code with //printf(...)
- Often smart to create generic log functions like dumpJobMessage() or printTimeoutQueue()

- A tool like 'strace' or 'ktrace' may be able to log easily read information for free!

Electric Fence

Adds run-time checks to your program to find errors related to malloc.

e.g.: writing out of bounds, use after free...

just compile your programs using -lefence

Alternative: Valgrind finds more memory errors, but is VERY slow, and many false positives

Alternative 2: cppCheck and antiC

That’s It!

Questions?

Good luck finishing Project 1!