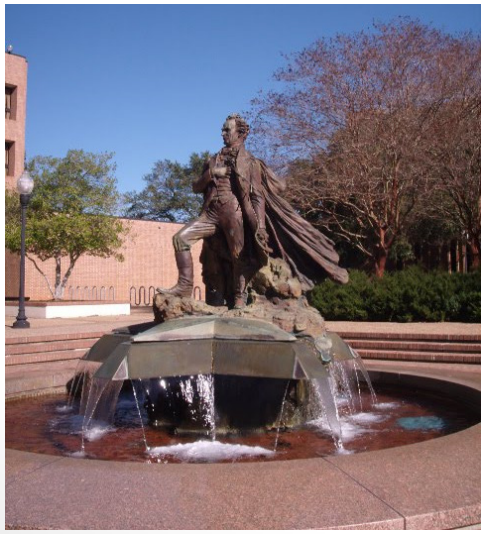
Putting Security Back Into The Code



Dr. David A. Cook
Stephen F. Austin State University
cookda@sfasu.edu

Dr. Eugene W.P. Bingue
U. S. Navy
dr.bingue@gmail.com

Who I am

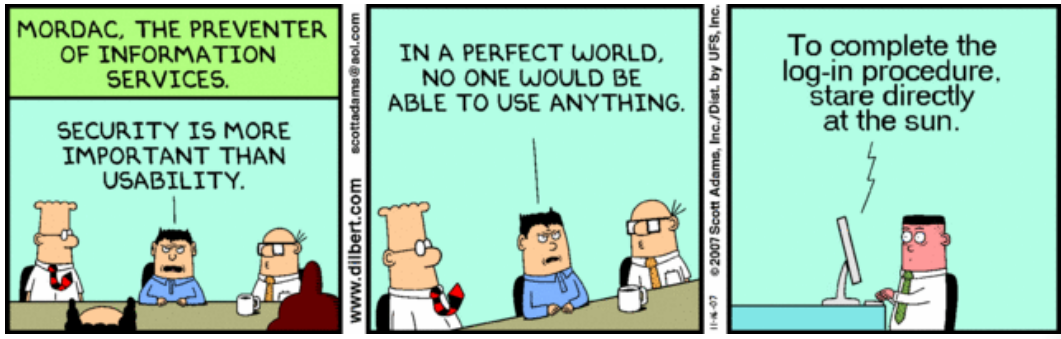
- Retired AF Officer (23 years) where I qualified for the Air Force Specialty Code for Software Engineer. I taught software engineering at the USAF Academy and the Air Force Institute of Technology. I also taught at Keesler AFB (technical training) for 6 years
- Former Consultant for the Software Technology Support Center (12 years)
- Professor of Computer Science, Stephen F. Austin State University
- Columnist for Crosstalk, the Journal of Defense Software Engineering (I write the Backtalk column for every issue)
- ABET Program Evaluator since 1998, and a Commissioner and/or Team Chair since 2007

Who is paying for this trip?

SFA is in Nacogdoches, TX. It is NOT in Austin, nor is it associated with any school of higher learning in Austin.

In fact, it is generally acknowledged by those of us who graduated from Texas A&M that there ARE no schools of higher education in Austin. 😂

What this talk is about



- In 2014, I was approached by 12 students, and they asked me to teach a course about “writing safe and secure code”
- Not one of the students needed the course to graduate, they just wanted to learn more about the topic
- The course involved comparing multiple language, and focused on the “weak areas” inherent in coding

Languages that I used in the course

- C++
- Java
- Ada (which several students had studied as part of a parallel processing course)
  - This course was NOT an attempt to convert students to Ada – the language was used to demonstrate alternate ways of implementing security features

First Topic – Typing and Syntax

- Strong typing has been shown to drastically reduce errors
  - Age is NOT an integer!
  - I has a lower limit of 0 – so use unsigned
  - However, it is limited to 120 or less (for humans!) – so a limit is required
- Solutions
  - Use a language that permits declarations of “integer-like” types with ranges
  - Use a class (and use a setter to verify all assignment and modifications)
- Cost
  - High – but critical to secure coding
  - Part of good design

Typing and Syntax

- Named vs. Typed equivalence is an issue
  - Multiplying a length by a length should not be assignable to a length
  - Moving a length directly to an area is incorrect
- Solution
  - Strongly typed language
  - Use a class to override operators (such as deleting equality in C++)
- Cost
  - High, but critical.
  - Part of good design

Typing and Syntax

- The “=“ operator causes problems
  - In tests, “==“ required, but “=“ frequently will not even raise a warning
  - Almost guaranteed to cause invalid results
  - When used with pointers, can give invalid results
- Solution
  - Use classes to define “isEqual”
- Cost
  - High, but saves debugging errors later

Topic - Attributes

- Magic numbers are the bane of maintenance programmers

- Age.MinValue or Age.MinValue() provides static or dynamic value.
- Some language provide many attributes (….yeah, Ada)
- They can be implemented easily in classes as either a public attribute (DANGER! Anybody can modify unless made constant!) or a public method with private attribute
- A simple getter will work

Topic – Pointers (access types)

- Java disallows
- C++ gives little checking for pointer type compatibility (because you are really using C for most pointer code)
- Useful to teach how to check validity of pointer reference before using it.
- Orphans and aliased values cause problems – so work to prevent

Topic – using Object-Oriented Programming

- Because it works
- Because it saves time in testing and maintenance
- Because if you can't understand it – you have no business writing real-time or embedded or missing-critical code
- NOTE: I didn't say you had to use it – I said you had to understand it! Learning to code in an OO manner changes the way you write code.

Object-Oriented programming

- Be aware that the lack of good garbage collection (and the total lack of a Destructor method – either implicit or explicit) makes Java problematic for some OO applications.

- Teaching students how to effectively use scope (pushing a new scope, exiting a current scope, referencing values in a higher scope) give tools to make code safer, more secure, and more robust.

Next Topic – Design and Interfaces

- Poorly designed modules and interfaces lead to hacking and loss of conceptual integrity (design compromises)

- Solution
  - As the stakes go up – the need for well designed classes and interfaces increases
  - UML is a great design language – but design MUST be part of the process BEFORE coding
  - Or – design MUST be re-accomplished during each spiral

Design and interfaces

- Part of the solution was making students familiar with all of the uses of classes in their target language (by this time – all were coding in Ada for the uniqueness and simplicity of types, and C++ to learn)

- C++ classes have a lot of options (SEVEN kinds of copy constructors you can use to override "=")

- Public, private, protected need to be well understood
  - Improper use de-localizes points of failure
  - Making everything private can cause problems later when adding children

Design and interfaces continued

- Class-wide variables (static variables)
- Static classes
- Singleton classes
- “Free functions”
  - Not everything needs to be in a class in C++. New developers (those new to OO) tend to create classes when the overhead of a class structure is not needed.
  - A program with free functions can still be a well-designed OO program.

Understand the different types of design

- Architectural (high-level system design)
- Data
- Interface
- Module design (algorithms)
- Modularization (effective modularization of the code)
  - What goes into classes
  - How children and friends are partitioned

- All must be considered
- Each influences the other

Design – modularizing code

- Efficient imports and includes can make writing code MUCH easier
- Children and friends need to be designed earlier
- Poorly designed header files (or header files with executable code) make large-scale design much harder
- In C++ - headers should have guards to prevent multiple including
- Again – DESIGN before CODING

Design is critical for security

- Good design makes better code
- Good design reduces testing
- Good design makes maintenance easier
- Good design localized where security breaches can occur
- Simply having single points for reading/writing files, memory and ports increases security

- Focus on RUME code
  - Reliable
  - Understandable
  - Maintainable and Modifiable
  - Efficient

Topic – Error / Exception Handling

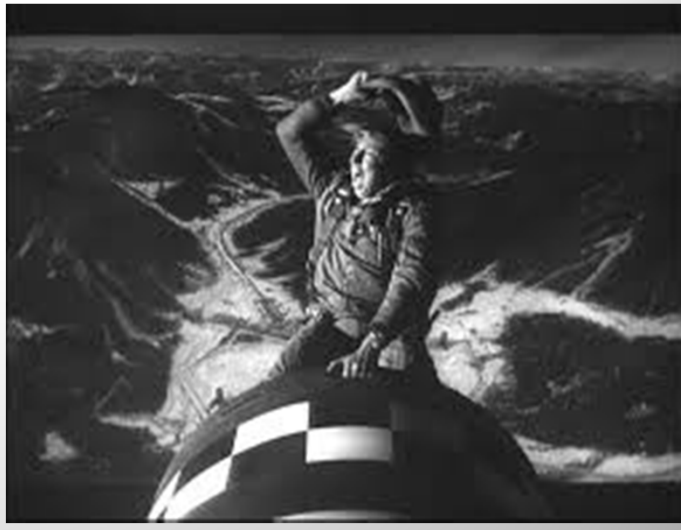
- Once you've though "gee – a problem could occur here" it's not an error anymore, is it? It's just an exception that you either handle or choose to ignore.

THINK ANOMALIES

- "try catch" blocks are critical to writing reliable and secure code.

- Need to be able to catch invalid or erroneous data and force user to re-submit

- Need to be able to gracefully shut down if needed



No Slim Pickens to fix it on the way down

Error Handling

- Few college classes focus on error handling
- Academic environments focus on “run once and then throw away” code
- Should be part of *design process* (both architectural and module)
- Critical to be able to handle common errors, to be able to think of what common errors might occur, and how to gracefully recover from errors
- Setters a necessity for validity checking of inputs

Topic – Subprograms, methods, parameters

- Students did not appreciate the usefulness of default parameters in methods
- Students did not appreciate the use of multiple default constructors for a method
- Students did not appreciate the use of overloading methods (but not overloading operators – which is a bad practice)

Topic – Overloading operators and methods

- Method overloading – GOOD! (In fact, this is one of the cornerstones of good OO development)
- Operator overloading – BAD! (In fact, this is a guaranteed way to increase testing and debugging)
- Overloading "=", or instance, makes the equality operator appear to be the system-supplied one, and it takes a while to figure out to look for errors in the "=" user supplied code

Topic – Parallel processing

- Most students (except those who have taken a formal Operating Systems course) do not understand the dangers of parallelization
- Deadlock and Race Condition are known, but not understood
- Writing threads (which are “light weight parallel processes” is a critical part of understanding
- Threads have no memory needs (no stack) – so easier to parallelize
- Heavy-duty concurrent skills are required to allow concurrent processing of data

Parallel Processing

- The lesson learned is that special training is required to write concurrent code
- Communication with other processes (via parameters or RPC – remote procedure calls) is easy to understand and implement
- Leave the optimization and creation of parallel code to those with sufficient training

Topic – Systems Programming

- Coding down to “bare metal”
- Accessing low-level features of the hardware
- Again – while some languages make it easy to access low-level features, it typically reduces security and makes the resulting code non-transportable
- Well designed interfaces should take care of the low-level requirements

RECAP – topics taught

- Typing and Syntax
- Attributes
- Pointers
- Object-Oriented Programming
- Design
  - Interface design
  - Architectural design
  - Data design
  - Modularization
- Error / Exception handling
- Overloading
- Subprograms, Methods, Parameters
- Parallel Processing (as in “don’t do)
- Systems Programming (ditto)

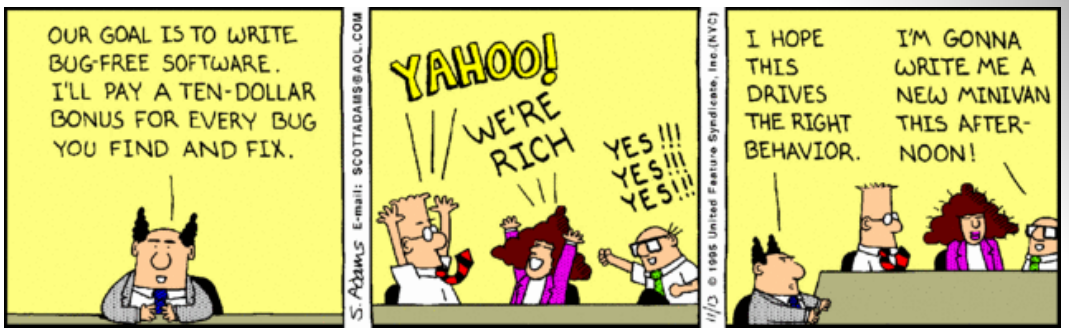
Results of the course

- All 12 students completed the course
- The critiques were amazing – the students said that this course “put it all into perspective”
- Students who also took a senior-level security class (7 of the 12) said that it changed the way they “though of coding”
- All agreed that the course would have been nice integrated into earlier courses, but that their limited knowledge would have prevented them from understanding it “back then”

References used for the course

- Language reference manuals for Java, C++ and Ada
- **C Traps and Pitfalls**, by Andrew Koenig
- **Expert C Programming: Deep C Secrets**, by Peter van der Linden
- **Expert C++,** by Herbert Schildt
- **Java puzzlers**, by Joshua Bloch
- **The Rationale for the Ada Programming Language**, and **An Introduction to Ada** by John Barnes
- **Safe and Secure Software – an invitation to Ada**, by John Barnes, with contributions by Ben Brosgol

THANK YOU!!!



If you have questions…..
comments…..
or want to express a differing opinion…..

David A. Cook

Professor, Dept. of Computer Science

Stephen F. Austin State University

cookda@sfasu.edu