IDA

INSTITUTE FOR DEFENSE ANALYSES

# Approaches to Cyber-Resilience through Language System Design

David A. Wheeler

November 2018

IDA Non-Standard
NS D-10330

INSTITUTE FOR DEFENSE ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

About This Publication

This work was conducted by the Institute for Defense Analyses (IDA) under contract HQ0034-14-D-0001, Task ITSDPB, for Institute for Defense Analyses. The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgments

For more information:

David A. Wheeler, Project Leader
dwheeler@ida.org, 703-845-6662

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

INSTITUTE FOR DEFENSE ANALYSES

IDA Non-Standard NS D-10330

# Approaches to Cyber-Resilience through Language System Design

David A. Wheeler

## Executive Summary

Current software often does not do what users wish due to defects, including security vulnerabilities. The defects (including vulnerabilities) may be unintentional or intentionally inserted. This presentation argues that it is possible to design, select, and modify our programming systems to reduce the presence or impact of defects or, in some cases, eliminate them entirely. This involves designing or modifying our programming languages, style checkers/enforcers, libraries, frameworks, package managers, and other software development infrastructure to counter these defects.

This presentation makes the argument that it is possible to counter defects by identifying many approaches that can be applied to the programming language specifically and by tooling support for development processes more generally.

Programming languages can be specifically designed to help counter defects. Their syntax can be designed to be unlikely to get wrong and unlikely to be misunderstood, for example, by preventing common mistakes like swapping "=" for "==" and countering misleading (underhanded) use of comments. The semantics of common operations should be clear to developers and avoid "magic" (special cases that do not always work). Programming languages should eliminate or greatly limit undefined behaviors (situations where the program can do anything at all). There has long been a debate between advocates of static and dynamic typing. Their advocates should instead learn from each other: designers of statically typed languages should emphasize speeding code development by reducing what the developer needs to declare or know (e.g., type inference), and designers of dynamically typed languages should find ways to gain some of the benefits of static types (e.g., via optional static typing). Using clear names, supporting pre-conditions/post-conditions/invariants, and good error-handling support can all help counter defects.

More generally, tools can be developed to support development processes, including design, implementation, validation, and sustainment. Tools can help support the secure development principles of Saltzer & Schroeder, such as simple design, fail-safe defaults, least privilege, and psychological acceptability (particularly by supporting "least astonishment"). Libraries can be provided to support input validation. Tools should be designed to counter each of the common types of vulnerabilities identified in the OWASP Top 10 and the CWE/SANS Top 25 (such as SQL injection, cross-site scripting, and buffer overflow). There should be a recommended test framework that measures at least statement and branch coverage. A built-in standard assert capability and support for something like QuickCheck is also valuable. Supporting fuzz testing (e.g., quicker function restarts and easily retrievable "number of branch executions" counts) can also be helpful. It is also valuable to ease the creation of static analysis tools (e.g., providing

libraries/APIs to get data about a program). Formal methods can be valuable as well, but they are also expressed as languages and there should be support for reducing defects in formal specifications. It is usually a bad idea to rewrite whole programs, so the system should instead support incremental improvement over time. Designers should also provide mechanisms to mark interfaces as deprecated and avoid “silent” changes in semantics. Tools should detect obsolete components or those with vulnerabilities and enable easy updates. Finally, tools should be usable when disconnected from the Internet, as that can reduce the attack surface.

The myriad approaches demonstrate that programming systems can be designed to improve resilience. It is best to do it up front, as many of these changes are more difficult to apply later, but some incremental improvements can also be made to existing systems. In conclusion, if we want to reduce software defects, programming system developers must consider countering defects as an important objective.

**Institute for Defense Analyses**

4850 Mark Center Drive • Alexandria, Virginia 22311-1882

# Approaches to Cyber-Resilience through Language System Design

High Integrity Language Technology (HILT) International Workshop on Cyber-Security Interaction with High Integrity, Boston, Massachusetts

2018-11-05

by

David A. Wheeler
PhD, CISSP
dwheeler @ ida.org

## The problem

- Software doesn’t do what users wish due to defects, including security vulnerabilities
  - Especially focus on vulnerabilities
- Defect categories for our purposes:
  - Unintentional defects
    - Security-related defects
    - Non-security-related defects
  - Intentional defects from malicious individual or subversion appearing to be from individual
    - Need to help organization counter underhanded code
  - Malicious organization: Hard to deal with
    - Organization’s goal & might not be considered defects
    - Mechanisms to support independent review can help

## The question

- Can we design/select/modify our programming systems to reduce/eliminate presence or impact of defects?
  - Including security defects?
- Yes!
  - We can design/modify programming languages, style checkers/enforcers, libraries/frameworks, package managers, & other infrastructure
  - Personal experiences with Ada back this up
  - Best to do it up front, but often possible to create backwards-compatible improvements
  - To counter unintentional should focus on *common* problems (e.g., OWASP top 10 & CWE/SANS top 25)

## Outline (for rest of talk)

- Programming language traits
  - Syntax, semantics, etc.
- Development processes*
  - Design
  - Implementation
  - Verification (dynamic & static)
  - Sustainment

* Processes are not the same as stages or phases. Processes typically occur simultaneously. See ISO/IEC 12228 for more.

IDA

## Programming language: Syntax (1)

- Make syntax “unlikely to get wrong & unlikely to be misunderstood”
- Example: = vs. == in C/C++
  - Common misunderstanding
  - Attempted 2003 Linux kernel subversion

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
   retval = -EINVAL;
```

  - Can require special style in conditional
    - E.g., “extra” parentheses to warn of unusual use
    - “Yoda conditions” (0==x) partly solve - many consider confusing
    - It’s better to design the language to avoid the problem!
  - Java also uses = and ==, but semantics expressly changed to make this unlikely
    - Java intentionally doesn’t auto-convert to Boolean, so “if (a = b) …” is usually detected at compile-time
    - Java has a different problem: .equals, not ==, for strings

## Programming language: Syntax (2)

- Syntax should be easy to parse with relatively little context
  - Easy for humans
  - Easy for editor syntax colorizing
- Ensure that editor syntax coloring can’t be easily fooled by malicious code
  - Helps counter underhanded code by showing *real* syntactic role (keyword, etc.)

## Programming language: Syntax of comments

- Some publicly available underhanded code exploits comment syntax
- Code in middle of comment line
  - C: /* Is this */code/* or not? */
  - Forbid (by language or linter) code in a line after comment text
- Code masquerading as a comment
  - Underhanded Python example:

```
uid = get_uid_from_db(username) // 100000000 is max # of users…
```

  - Beware of allowing another language’s comment initiator as an operator!

## Programming language: Semantics

- “What common operations do” should be clear to developers
- Beware of excessive magic (weird special cases that don’t always work)
- Example: JavaScript “==”
  - “the == rules are easy to get wrong even if you feel like you're familiar with them” - JavaScript equality game
  - Added “===“ operation, but it looks like “==”
    - A style checker can *mandate* === instead of ==

Source: https://slikts.github.io/js-equality-game/

## Programming language: Undefined constructs

- Sometimes need to give language implementers some “wiggle room”
- Disastrous way: “Entire program may do anything if anywhere in the program something is undefined”
- C does it the disastrous way
  - Huge number of undefined behaviors
  - C standards committee is now one of your most dangerous adversaries if you use C

## Undefined behavior: (Simplified) Linux kernel security flaw #1

- Sample simplified Linux kernel code with a security defect
  - “dev->priv” dereference presumes dev is non-null
  - If “dev” is null, we have undefined behavior
  - C compiler presumed that dev is not null, and threw away the “if (!dev) return” code
- Compiler flag -fno-delete-null-pointer-checks
  - Forces the retention of such checks

```
static void __devexit agnx_pci_remove(struct pci_dev *pdev)
{
  struct ieee80211_hw *dev = pci_get_drvdata(pdev);
  struct agnx_priv *priv = dev->priv;
  if (!dev) return;
  ... do stuff using dev ...
}
```

Source: [Regehr]. https://isc.sans.edu/diary/A+new+fascinating+Linux+kernel+vulnerability/6820 & [Wheeler] https://dwheeler.com/secure-class/presentations/Secure-Software-8-Errors.ppt

## Undefined behavior: (Simplified) Linux kernel security flaw #2

- C standard : when integer is + or - from a pointer
  - result must be a pointer to same object or one past object end
  - otherwise the behavior is undefined
- By this assumption, pointer arithmetic never wraps
  - the compiler can perform algebraic simplification on pointer comparisons
- Both gcc & clang simplify “(buf + size) < buf” to “size < 0”
  - On 32-bit clang will eliminate whole branch (size_t is unsigned to always true)
- Compiler flag -fno-strict-overflow stops check removal

```
int vsnprintf(char *buf, size_t size, …) {
  char *end;
  /* Reject out-of-range values early... */
  if (WARN_ON_ONCE((int) size < 0)) return 0;
  end = buf + size;
  /* Make sure end is always >= buf */
  if (end < buf) { ... }
  ...
}
```

Linux kernel’s lib/vsprintf.c

Source: [Wang 2012] “Undefined Behavior: What Happened to My Code?” https://people.csail.mit.edu/nickolai/papers/wang-undef-2012-08-21.pdf

IDA

## C undefined behavior: Getting worse

- "Recently we have seen spectacular advances in compiler optimisation."
- "Spectacular in that **large swathes of existing previously-working code** have been discovered, by diligent compilers, to be contrary to the published C standard, and '**optimised' into non-working machine code**."
- "In fact, it turns out that **there is practically no existing C code which is correct** according to said standards (including C compilers themselves)."
- "Real existing code does not conform to the rules now being enforced by compilers."
- "Indeed often it **can be very hard to write new code which does conform to the rules**, even if you know what the rules are and take great care."
- Recommended changing Debian's "default compiler options to favour safety, and provide more traditional semantics."

Source: Ian Jackson (UK), "Subject: Re: Packaging of static libraries", 13 Apr 2016 13:29:05 +0100, https://lwn.net/Articles/684420/

IDA

## Dealing with undefined behavior

- Don’t have totally undefined behavior
  - Bound it in some way (e.g., “implementation may do X or Y…”)
  - Make where it occurs “obvious”
  - Limit where it can be used (“unsafe” mode)
- Backwards-compatible solutions
  - Add options to define the undefined
  - Add tools/style checkers to detect undefined constructs
  - Demand standards bodies bound behavior & justify every undefined behavior

## Static typing: The great debate

- Static vs. dynamic typing expectations (& my experience):
  - Static typing should increase likelihood of compile-time error detection, speed runtime, provide better edit-time information, & ease later documentation
    - Type info does take time to write, debug, maintain
  - Dynamic typing should speed code development/modification (don’t need to identify types)
    - Many “real systems” create test cases that partly duplicate static typing checks, reducing time saved (common in Ruby)
- Learn from each other!
  - Static typing: Emphasize speeding code development by reducing what developer needs to declare/know (e.g., type inference (ML, OCaml, Haskell, Scala, F#, ~C#, ~C++))
    - Wordiness *not* good, *not* the same as clarity (often opposite)
  - Dynamic typing: Find ways to gain (some) static type benefits (e.g., add (optional) static typing (Flow, TypeScript, Python, Common Lisp, …))

## Pre-/post-conditions & invariants

- Have a *standard* way to provide pre-conditions, post-conditions, invariants
  - Useful for tests, formal verification, optimization
- Some languages build in (which is great!)
  - Ada 2012 added, SPARK switched to it
  - Eiffel
- Some languages have de facto standard
  - C: ACSL (via Frama-C)
  - Java: Java Modeling Language (JML)
  - Comment-based systems rarely used for optimization

IDA

## Use clear names & mental models

- Choose names carefully
  - Clear names & mental models *greatly* ease *correct* understanding
  - Often hard to change name later
  - Have & use a single naming convention
    - camelCase / underscore_case, “?” at end, etc.
- Beware: Boolean parameter names
  - Usage often lacks context (only true or false)
  - Instead of “melted” consider more specific name (isMelted, allowMelted, …)

## Error handling

- Real world causes many errors
  - Many programs are *mostly* error handling
  - Need easy-to-use programming mechanisms for handling errors
- Exceptions widely used/understood
  - Scheme finally added them in R6RS
  - So C is the only widely used language lacking them
  - Wide use & understanding helpful
  - Often easy to *not* handle when you should
- Return error codes easily implemented
  - But easily ignored & sometimes overlap data
  - Alternative: Haskell “Maybe a = Nothing | Just a”
    - As efficient as error return, but counters its problems

## Design process (1)

- Language systems should support design
  - E.g., Saltzer & Schroeder principles
- Economy of mechanism (simple design)
  - Provide for common cases
  - Language: Easy/efficient/safe concurrency
  - Libraries: Unicode string, web framework, etc.
- Fail-safe defaults/complete mediation
  - Ease implementation of “deny by default”
- Least common mechanism
  - Minimize sharing with different privileges
  - E.g., package installer should avoid using /tmp

IDA

## Design process (2)

- Least privilege
  - Provide basics to minimize privileges granted, time granted
  - Ease dividing software into differently privileged components that communicate with each other & can’t subvert the other
  - Package manager & tools should work with fewer privileges
- Psychological acceptability
  - Design for “least astonishment” (e.g., what does that construct mean? when is the network accessed?)
- Input validation with whitelists
  - Provide easy libraries for input validation (regex for general case, HTML, URL, email address, …)

## Design process (3)

- Selection of programming language & components are key design decisions
- Ease selection of library components
  - Package manager should download securely
    - At least HTTPS
    - Even better: Signatures that can counter repo subversions
  - Ease identification of what’s more/less used
  - See: The Update Framework (TUF)
- Have a standard style guide
  - Nobody likes endless arguments over style
  - Have automated tool(s) that enforce it
  - Unautomated “rules” means “doesn’t matter” – design your language system that way

## Implementation process (1)

- Language/libraries: counter common problems
  - OWASP top 10, CWE/SANS top 25, NSA CAS list
- Buffer overflow
  - Don’t allow or allow only in “unsafe” mode
- Cross-site scripting (XSS)
  - Need templates to “escape correctly by default”
  - Auto-escape (e.g., Ruby on Rails’ SafeBuffer)
    - SafeBuffer = “already properly escaped HTML”
    - Appending String S to SafeBuffer also escapes S
    - Appending SafeBuffer to SafeBuffer just appends
    - Special operation converts String to SafeBuffer unescaped
    - Output generated as a SafeBuffer

## Implementation process (2)

- SQL injection
  - Provide Object-Relational Mapping (ORM) in a common library to eliminate many problems
  - Provide prepared statements
    - These need to work with multiple popular DBMSs
  - Make clear that string concatenation of user data combined with exec is dangerous
- Other injection
  - Provide prepared-statement-like interfaces
  - Have marshal/unmarshal mechanisms safe for untrusted data (vs. XXE, pickle, etc.)

## Verification process: Dynamic (1)

- Have a standard recommended test framework
  - Otherwise people need to decide which one to use
  - Test framework should measure *at least* statement & branch coverage
    - Bad test suites can have good coverage, but bad coverage always means a bad test suite
    - Ruby <2.5 didn't support branch coverage, and even now most Ruby test support tools don't support branch coverage
- Language/key libraries must have good automated tests
  - Or they won't be trustworthy enough for your users
  - "Dogfood" your test framework to make it good

IDA

## Verification process: Dynamic (2)

- Include built-in “assert” (“always true”)
  - Trivial to implement, but painful if there’s no standard way to do it
  - Assert was required by Steelman (Ada requirements) item 10F, yet not added until Ada 2005
  - Enable control of checking assertions & what happens if one fails
- Support QuickCheck-like capability
  - Originally from Haskell – quickly sends in a few random valid values & checks for valid results
  - Many languages have it – make sure yours does

IDA

## Verification process: Dynamic (3)

- Provide supports for fuzz testing
  - Fuzz testing = Send a very large number of inputs & monitor program to detect problems
  - Aided by assertions, pre-/post-conditions, invariants
  - Key need: Fast startup/restart of function/ library/ program so can execute more tests
  - American Fuzzy Lop (AFL) surprisingly powerful by monitoring “number of executions” on each branch
    - Make it easy to get that information!

## Verification process: Static (1)

- Ease creating static analysis tools
  - Provide libraries/APIs to get data about program
  - Organizations report that analyses specialized for their specific target of evaluation (TOE) are especially effective
- Formal methods (FM) support
  - Rigorously define language semantics

IDA

## Verification process: Static (2)

- Beware: FM specifications are written in FM languages & errors can also happen in them!
  - Empty types confuse humans:
    - ∀ X martian(X)→green(X) “All Martians are green”
    - ∀ X martian(X)→¬green(X) “All Martians are not green”
    - Both statements are *true* if there are no Martians
    - Devise notations to counter (e.g., PVS “+” for not empty)*
  - Common mistake: ∃…→… (usually use ∧, not →)
  - English “and” and “or” don’t translate exactly
  - Devise language or style checkers to counter/ detect common errors in FM notation too!

* One idea is the “allsome” quantifier “∀!” - see https://dwheeler.com/essays/allsome.html

IDA

## Sustainment

- Usually a terrible idea to rewrite big programs*
- Support incremental maintenance over time
  - Let people make partial changes over time
  - Learn from the Python 2->3 transition train wreck
- Include standard mechanism to *deprecate* interface with clear compile/test time notification
  - Enables incremental improvement
- Avoid “silent” changes of semantics
  - In general, use new method/name/parameter
- Support refactoring (automatically if possible)
- Package manager should detect components with vulnerabilities / old, enable easy update

*For more, see “If it works, it’s legacy: analysis of legacy code” by David A. Wheeler, https://www.adacore.com/sound-static-analysis-workshop/program/wheeler-talk

IDA

## Desiderata for development tools

- Be usable while disconnected from the Internet
- Be usable in a sandbox (limited privilege)
- Run even where developers aren’t fully trusted
- DO NOT ask for more privileges than clearly needed
- If you don’t need to write, don’t ask for the privilege
- Allow repeatable re-execution (verify results)
  - Time-based heuristics for “giving up” are a problem in static analysis, because it’s not repeatable
- Make it easy to check-in inputs and results (verify results)
- Consider releasing the tools as OSS (reviewability)

IDA

## Conclusions

- Programming language systems *can* be designed to improve resilience
  - Programming language, libraries/frameworks, tools, etc.
- Best to do up front
  - You may be able to make some incremental improvements to existing systems
- But their developers must consider it
  - If you don’t work toward it, you won’t get it

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

| | | |
|---|---|---|
| 1. REPORT DATE (DD-MM-YY)<br>00-11-18 | 2. REPORT TYPE<br>Non-Standard | 3. DATES COVERED (From – To) |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>Approaches to Cyber-Resilience through Language System Design | 5a. CONTRACT NUMBER<br>HQ0034-14-D-0001 |
| | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBERS |
| 6. AUTHOR(S)<br>David A. Wheeler | 5d. PROJECT NUMBER<br>ITSDPB |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES<br>Institute for Defense Analyses<br>4850 Mark Center Drive<br>Alexandria, VA 22311-1882 | 8. PERFORMING ORGANIZATION REPORT NUMBER<br>NS D-10330 |
| 9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br>Institute for Defense Analyses | 10. SPONSOR'S / MONITOR'S ACRONYM<br>IDA |
| | 11. SPONSOR'S / MONITOR'S REPORT NUMBER(S) |

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: David A. Wheeler

14. ABSTRACT

Current software often does not do what users wish due to defects, including security vulnerabilities. The defects (including vulnerabilities) may be unintentional or intentionally inserted. This presentation argues that it is possible to design, select, and modify our programming systems to reduce the presence or impact of defects, and in some cases eliminate them entirely. This involves designing or modifying our programming languages, style checkers/enforcers, libraries, frameworks, package managers, and other software development infrastructure to counter these defects. The presentation does this by identifying many approaches that can be applied to the programming language (including its syntax and semantics) and development processes (specifically design, implementation, verification, and sustainment). The myriad approaches demonstrate that programming systems can be designed to improve resilience. It is best to do it up front, but some incremental improvements can also be made to existing systems. In conclusion, if we want to reduce software defects, programming system developers must consider countering defects as an important objective.

15. SUBJECT TERMS

Software development, programming, defects, security vulnerabilities, software assurance, countering vulnerabilities, countering attack, cybersecurity, cyber security, computer security, software development environment, software engineering environment, programming systems, defect reduction, quality assurance, Ada, undefined behavior, style checkers, cross-site scripting, XSS, SQL injection, injection, package managers, secure software, designing programming languages

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON<br>Institute for Defense Analyses |
|---|---|---|---|---|---|
| a. REPORT<br>Unclassified | b. ABSTRACT<br>Unclassified | c. THIS PAGE<br>Unclassified | Unlimited | 32 | 19b. TELEPHONE NUMBER (Include Area Code) |