C++ and Programming Language Vulnerabilities

Stephen Michell
Convenor of ISO/IEC/JTC 1/SC 22/WG 23 Programming Language Vulnerabilities
stephen.michell@csagroup.org
stephen.michell@maurya.on.ca

Outline

- History of WG 23
- Edition 3 Outlook
- Compare with Ada
- Changes from Edition 2
- What are programming language vulnerabilities?
- Vulnerability examples
- Documenting C subset of C++
- Working with WG 23

History of WG 23

- ISO/IEC/SC22/WG23 Programming Language Vulnerabilities
- Formed in 2006 to address mistakes in programs that can lead to attacks or faults the can cause application and system failures
- Published first Technical Report TR 24772 Guidance in avoiding programming language vulnerabilities through language selection and use in 2010
  - Identified and documented vulnerabilities in a general way
  - WG 23 was working on programming language specific annexes, but none were ready
- Published edition 2 in 2013
  - Contained annexes for
    - Ada
    - C
    - Python
    - PHP
    - Ruby
    - Spark

Edition 3 Outlook

- Slight name change
  - Dropping “through language selection and use”
- Move the language-specific annexes from the document
- Put separate parts
    - too difficult to keep all language-specific annexes in sync for a single publication cycle
  - Base document becomes 24772-1
  - Ada guidance becomes 24772-2
  - C guidance becomes 24772-3
  - Python guidance becomes 24772-4
  - Fortran guidance becomes 24772-8
- Expect ballot to start in 2018

Edition 3 Outlook (cont)

- Others (Spark, Ruby, PHP, etc) in 1-3 years
- Planning a C++ Part, hopeful
- Rationale
  - C++ is being increasingly used in places where the safety and security of the application matter.
  - C++, like <u>all</u> programming languages, provides capabilities to developers and makes choices that leave applications open to programming errors or attacks that can be detrimental to systems or users that depend on the application.
  - The TR explains the vulnerability, its possible consequences, and ways to avoid it.

Compare with Ada

- Ada is known as a “safe language”
  - Very strong type system
  - Robust compile-time and runtime checking
- Yet TR 24772-2
  - Acknowledges 50 (out of 63) vulnerabilities as having applicability to Ada.
  - Most of the rest are acknowledged if an identified, small set of unsafe features of the language are used.
  - The guidance to users helps to avoid or mitigate the vulnerabilities

Changes from edition 2?

- Added a few new vulnerabilities
  - Deep vs Shallow Copying
  - Violations of the Liskov Substitution Principle
  - Redispatching
  - Polymorphic Variables
  - 3 vulnerabilities (in clause 7) on time-related issues
- Moved some vulnerabilities to clause 7
  - no language-related implications or mitigations in clause 7
  - Fault tolerance and failure strategies
  - Distinguished values in data types
- Added a top-N guidance in clause 5
  - Not a coding standard, but a collation of the most common (and important) guidelines that apply to many vulnerabilities in clauses 6 & 7

What are Programming Language Vulnerabilities?

- Every application program exists on a machine with constrained resources
  - Fixed word size and formats mean that some operations will always overflow, overflow or wrap around
  - And sometimes it is not an error
  - Fixed memory size and or long-lived programs mean that we must reuse memory locations. This leads to reusing variables, releasing and reallocating memory, sharing or reinterpreting data in other contexts

What are Programming Language Vulnerabilities? (cont)

- Every programming language contains features that:
  - Permit data regions to be sized, resized, allocated, destroyed
  - Permit data to be reinterpreted
  - Permit data to be created in one type and arbitrarily changed to another
  - Permit algorithms to be prematurely terminated or arbitrarily extended
  - Permit the arbitrary input, interpretation and output of data over arbitrary I/O channels
- Each expose the application to the risk that the capability will be misused in a way that could adversely affect the system that relies upon the application.

What are Programming Language Vulnerabilities? (cont)

- Programming languages can/have been devised that minimize or mitigate the risks, but cannot eliminate them.
  - Where an attacker has sufficient knowledge of the application, or the OS environment, or the communications protocols, or the timing order of application components, or the order of access to external resources, then attacks can be constructed that will compromise the system.

Example – Buffer Overflow

- A classic vulnerability in C
  - C IO (old style) does not enforce the buffer size of an array (say an array of characters)
  - very often, the developer makes assumptions about input data and does not check dynamically.
  - Unrestricted input can overflow the buffer and write new data or instructions into neighbouring buffers or onto the stack, resulting in program failure or even arbitrary code execution.
- C++ provides mechanisms to mitigate or eliminate the defect.
  - Buffers can be placed into classes and access restricted via methods that can be shown to obey the size limitations of the buffer.
  - Guidance would be to always encapsulate data elements into classes and restrict access to the buffer to methods of the class.

Example – Liskov Substitution Principle

- Present in all object-oriented languages.
- Each method at the root of a hierarchy needs to specify
  - the strongest preconditions that are required to call any extension of the method, a
  - the weakest postcondition that the method or any of its derivatives will deliver.
- Each extension of a method must
  - Weaken the preconditions
  - Strengthen its delivery guarantees the postconditions
- Failure is that implementors do not understand the principle and override a method with one that has stronger preconditions or weaker postconditions
- Guidance is to forbid such strentheneing/weakening, use static analysis tools, etc.

About “safe” languages

- Almost all “real” languages are dramatically “pruned” when used in projects with stringent needs for security or safety
  - Witness JSF coding rules for C++ or MISRA
  - Witness Spark subset of Ada
    - Removes all access types (pointers), generics, polymorphism, generalized tasking, functions with side effects, calendar clock, atomic data
- There is no weakness in documenting the vulnerabilities in a language and recommending approaches and techniques to avoid them

Documenting C subset of C++ - a proposed approach

- TR 24772-3 already fully documents the C subset
  - With a couple of exceptions where C++ changed
- C++ Part must recognize the subset and the way that it is used
  - C++ Part should not restate the C Part, but should reference it.
- Make a simple statement that the appropriate vulnerabilities as documented in TR24772-3 clause 6.x exists in C++
  - Document features that share that vulnerability (if applicable)
  - Document C++ features that mitigate or avoid the vulnerability
  - Recommend that these be used instead
- Where C++ changes a C language feature,
  - Document the impact of the changes on any related C vulnerabilities
  - Document new or changed vulnerabilities

Working with WG 23

- WG 23 is willing to
    - Let WG 21 lead with the documentation of vulnerabilities in C++
    - Lead and ask WG 21 for technical assistance
- How can we work together?