Report of Progress of ISO/IEC 24772, Programming Language Vulnerabilities, in ISO/IEC JTC 1/SC 22

John Benito, Convener
Jim Moore, Secretary
ISO/IEC JTC 1/SC 22/WG 23
1 December 2008

This version of the presentation is an update of ISO/IEC JTC 1/SC 22/WG 23 N0140.

The Problem

- Any programming language has constructs that are imperfectly defined, implementation dependent or difficult to use correctly.
- As a result, software programs sometimes execute differently than intended by the writer.
- In some cases, these weaknesses can be exploited by hostile parties, or can lead to failure in anticipated environments.
  - Can compromise safety, security, privacy, dependability or other critical properties.
  - A vulnerability in any program can be used as a springboard to make additional attacks on other programs.

Complicating Factors

- The choice of programming language for a project is not solely a technical decision and is not made solely by software engineers.
- Some vulnerabilities cannot be mitigated by better use of the language but require mitigation by other methods, e.g. review, static analysis.

Planned ISO/IEC 24772

- A type III Technical Report
  - A document containing information of a different kind from that which is normally published as an International Standard
- The report will not contain *normative* statements, but information and suggestions.
- Project is to work on a set of *common mode* failures that occur across a variety of languages
  - However, not all vulnerabilities are common to all languages. Some are manifest themselves in different ways in different languages.
- Annexes to the report will describe how the vulnerabilities relate to specific languages.

Planned ISO/IEC 24772 (continued)

- No single programming language or family of programming languages is to be singled out
  - As many programming languages as possible should be involved
  - Need not be just the languages defined by ISO Standards

Audience

- *Safety*: those developing, qualifying, or maintaining a system where it is critical to prevent behaviour that might lead to loss of human life or human injury, or damage to the environment.
- *Security*: those developing, qualifying, or maintaining a system where it is critical to exhibit security properties of confidentiality, integrity, and availability.
- *Mission-Critical*: those developing, qualifying, or maintaining a system where it is critical to prevent behaviour that might lead to property loss or damage, or economic loss or damage.
- *Modeling and Simulation*: those who are primarily experts in areas other than programming but need to use computation as part of their work [and who] require high confidence in the applications they write and use.

Approach to Identifying Vulnerabilities

- *Empirical approach:* Observe the vulnerabilities that occur in the wild and describe them, e.g. buffer overrun, execution of unvalidated remote content
- *Analytical approach:* Identify potential vulnerabilities through analysis of programming languages
  - This just might help in identifying tomorrow's vulnerabilities.

Desired Outcomes

- Provide guidance to users of programming languages that:
  - Assists them in improving the predictability of the execution of their software even in the presence of an attacker
  - Informs their selection of an appropriate programming language for their job
- Provide feedback to programming language standardization groups, resulting in the improvement of programming language standards.

WG 23 Participants

- National Bodies
  - Canada
  - Germany
  - Italy
  - Japan
  - France
  - United Kingdom
  - USA
- Other Groups
  - RT/SC Java
  - MISRA C/C++
  - CERT
- Language Standards Groups
  - SC 22/WG 9
  - SC 22/WG14
  - SC 22/WG 5, INCITS J3 (Fortran)
  - SC 22/WG 4, INCITS J4 (Cobol)
  - MDC (Mumps)
  - ECMA (C#, C++CLI)

WG 23 (Vulnerabilities) Progress

- Organization:
  - Project was originally assigned to a temporary group, an "other working group" called OWGV.
  - In September 2008, SC 22 created WG 23 to continue the work
- Meetings:
  - E-Mail reflector, Wiki and Web site are used during and between meetings
  - Nine meetings have been held, hosted by six national bodies.
  - Meetings planned through 2009
- Progress through standards process:
  - Working Group Level
    - ✓ Working Draft (WD) – several of them
  - Parent (SC 22) Level
    - ✓ PDTR registration
    - PDTR ballot - repeated until consensus is obtained, typically two or three times
  - Management (JTC 1) Level
    - DTR Ballot
  - Publication by ISO/IEC (Planned in 2009)
- More information: http://aitc.aitcnet.org/isai/

Outline of Current Draft

- Scope
- References
- Terms and Definitions
- Vulnerability Issues
- Programming Language Vulnerabilities
  - (Currently 48 of them)
- Application Vulnerabilities
  - (Currently 18 of them, selected because of relationship to languages)

Vulnerability Template

- The major portion of Technical Report describes vulnerabilities in a generic manner, including:
  - Brief description of application vulnerability
  - Cross-reference to enumerations and other classifications, e.g. CWE
  - Description of failure mechanism, i.e. how coding problem relates to application vulnerability
  - Applicable language characteristics
  - Avoiding or mitigating the vulnerability
  - Implications for standardization
- Annexes will provide language-specific treatments of each vulnerability.
- The following slides provide an example.

Example Description

6.17 Boundary Beginning Violation [XYX]

6.17.1 Description of application vulnerability

A buffer underwrite condition occurs when an array is indexed outside its lower bounds, or pointer arithmetic results in an access to storage that occurs before the beginning of the intended object.

6.17.2 Cross reference

[Cross references to CWE, JSF, MISRA, CERT, etc.]

Continued…

6.17.3 Mechanism of failure

There are several kinds of failures (in some cases an exception may be raised if the accessed location is outside of some permitted range):

- A read access will return a value that has no relationship to the intended value, e.g., the value of another variable or uninitialized storage.
- An out-of-bounds read access may be used to obtain information that is intended to be confidential.
- A write access will not result in the intended value being updated and may result in the value of an unrelated object (that happens to exist at the given storage location) being modified.
- When the array has been allocated storage on the stack an out-of-bounds write access may modify internal runtime housekeeping information (e.g., a functions return address) which might change a programs control flow.

Continued…

6.17.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that do not detect and prevent an array being accessed outside of its declared bounds.
- Languages that do not automatically allocate storage when accessing an array element for which storage has not already been allocated.

Continued…

6.17.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:.

- Use of implementation provided functionality to automatically check array element accesses and prevent out-of-bounds accesses.
- Use of static analysis to verify that all array accesses are within the permitted bounds. Such analysis may require that source code contain certain kinds of information, e.g., that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified.
- Sanity checks should be performed on all calculated expressions used as an array index or for pointer arithmetic.

Some guideline documents recommend only using variables having an unsigned type when indexing an array, on the basis that an unsigned type can never be negative. This recommendation simply converts an indexing underflow to an indexing overflow because the value of the variable will wrap to a large positive value rather than a negative one. Also some language support arrays whose lower bound is greater than zero, so an index can be positive and be less than the lower bound.

In the past the implementation of array bound checking has sometimes incurred what has been considered to be a high runtime overhead (often because unnecessary checks were performed). It is now practical for translators to perform sophisticated analysis that significantly reduces the runtime overhead (because runtime checks are only made when it cannot be shown statically that no bound violations can occur).

Continued…

6.17.6 Implications for standardization

- Languages that use pointer types should consider specifying a standard for a pointer type that would enable array bounds checking, if such a pointer is not already in the standard.

6.17.7 Bibliography

[None]