Progress Report: ISO/IEC 24772, Programming Language Vulnerabilities

By James W. Moore and John Benito
Draft 2, Submitted to Ada User, 16 March 2009

Any programming language has constructs that are imperfectly defined, implementation-dependent, or difficult to use correctly. As a result, software programs sometimes execute in a manner that is different than what was intended by the developer. In some cases, the unintended functionality can be exploited by hostile parties or can lead to failure when used in unanticipated circumstances. The result can be a compromise of safety, security, privacy, dependability or some other critical property. Security vulnerabilities are a particular concern because an adaptive adversary can use a compromise in ***any*** executing program—even a non-critical one—as a springboard to make additional attacks on other programs. This report describes an effort to develop an authoritative account of the known weaknesses in programming languages and how developers might avoid those weaknesses.

Despite the fact that all programming languages have weaknesses, they manifest the weaknesses in different ways and the weaknesses must be mitigated in different ways, sometimes by better use of the language, sometimes by tooling such as static analysis, and sometimes by other methods such as review.

Some will be tempted to dismiss the problem, saying that one should simply use a better programming language. However, this viewpoint would overlook two factors:

- All programming languages have some weaknesses.
- The selection of a programming language for a project is often not a technical decision but is often forced by external concerns.

This article will describe progress on the planned ISO/IEC TR 24772, *Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use*. The project is being conducted in ISO/IEC JTC 1/SC 22/WG 23[1]. The WG has two officers—John Benito, convener and James Moore, secretary—the authors of this article.

The "TR" in the designation of the document means that it is not a standard (a document that prescribes requirements for conformance), but a Technical Report—in this case, a Type 3 Technical Report—a document that provides guidance but not requirements. Therefore, the report will consist of information and recommendations. The report will describe programming language weaknesses in a generic manner that spans a broad selection of languages. However, since

[1] The International Electrotechnical Commission (IEC) develops standards for electrical and electronic devices; the International Organization for Standardization (ISO) develops standards for nearly everything else. They have a Joint Technical Committee (JTC 1) that deals with information technology. One of its subcommittees (SC 22) deals with programming languages. A working group (WG 23) of SC 22 is producing the subject document.

- not all vulnerabilities are present in all languages;
- the ones that are present manifest themselves differently in different languages;
- and mitigation of the manifested vulnerabilities differ among the various languages

there is a need for language-specific material. Therefore, the report will include annexes that are specific to various programming languages. We plan to cooperate with other SC 22 working groups (the ones responsible for the standardized programming languages) to write these annexes. We also hope to obtain annexes for languages standardized by organizations that are outside of ISO/IEC.

Although the information in the Technical Report would be useful for the development of software required to exhibit any critically important property, the report is intended for four specific audiences:

- *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.

The working group has taken two approaches to identifying weaknesses in programming languages. An *empirical* approach has relied on prior efforts that categorize particular classes of vulnerabilities that appear to occur frequently in the wild. This has been particularly helpful in finding security-related weaknesses because large numbers of security weaknesses result from a few identifiable patterns of attack, such as buffer overrun and execution of unvalidated remote content. An *analytical* approach has built on prior efforts that identified weaknesses via *a priori* analysis of particular programming languages. This has been particularly helpful in identifying safety-related weaknesses. We can speculate that it might also be helpful in identifying the security weaknesses of the future as current opportunities become less easily exploitable.

So the report will provide guidance to users of a broad range of programming languages. In some cases, a language-specific annex will provide specific guidance. However, the generic discussions will be useful for users of languages that are not specifically covered. The advice will assist users in improving the predictability of the execution of their software, even in the presence of an attacker or its use in anticipated circumstances. It will also inform their selection of an appropriate programming language for a project, when they have the freedom to make that choice.

The working group also plans an outcome in addition to the report itself. The working group will provide feedback to its sibling working groups, suggesting ways in which the standardized

specification of the programming language might be improved so that predictability of execution would be improved.

The project has succeeded in gaining a broad base of participation. Measured in various ways we have participation from a variety of parties and interests. For example, at some level, we have participation from:

- Eight nations: Canada, France, Germany, Italy, Japan, Netherlands, United Kingdom, and USA
- Several programming languages: Ada, C, C++, C#, C++CLI, Cobol, Fortran, Java, MUMPS
- Some organizations with a strong interest in dependable software: the Computer Emergency Response Team (CERT) of Carnegie Mellon University, the US Food and Drug Administration, the US National Security Agency, and the Motor Industry Software Reliability Association

WG 23 has handled this project since its creation in September 2008; previously the work was performed by an ad hoc sub-group of SC 22 with the odd name of OWGV (standing for "other working group – vulnerabilities"). Like any ISO/IEC product, the report will go through a process of ever-widening consensus formation. Working Drafts were written by the working group and its predecessor. A Preliminary Draft Technical Report (PDTR) is currently under review and ballot by the parent, SC22. Comments from that ballot will be resolved and the ballot repeated as necessary until consensus is reached. Finally, the Draft Technical Report (DTR) will be balloted for approval by the grand-parent (JTC 1). Only after approval by at least 75% of the JTC 1 nations will the Technical Report be published—probably in early 2010. That will probably not be the end of the story—evolving attack patterns and the evolution of the language standards will require future revision of the report. Furthermore, there will be a continuing effort to "recruit" additional language-dependent annexes.

The working group conducts its work in person with nine meetings to date, supplemented by a wiki, an email reflector, and a website. The website can be accessed by the public at http://aitc.aitcnet.org/isai/

The current body of the draft document contains seven major sections:

- Scope (an explanation of the intended use of the document)
- References (any other standards that one must use with this one)
- Terms and Definitions
- Symbols (none so far)
- Vulnerability Issues (an explanation of some general concepts0
- Programming Language Vulnerabilities (discussions of the programming language weaknesses)
- Application Vulnerabilities (other vulnerabilities that don't result from programming languages *per se* but which are related to programming language usage)

In the current draft, 48 programming language weaknesses are described as well as 18 application vulnerabilities.

The primary content of the Technical Report's body are the 48 descriptions of programming language weaknesses. All of them follow a uniform outline:

- Brief description of the vulnerability as it occurs in execution
- Cross-reference to enumerations and other classifications, e.g. CERT Coding Guidelines, Common Weakness Enumeration (CWE), Joint Strike Fighter (JSF) Coding Guidelines, MISRA C Coding Guidelines.
- Description of failure mechanism, i.e. how the coding problem leads to a vulnerability in the application
- Applicable language characteristics, i.e. the types of programming languages affected by the weakness
- Avoiding or mitigating the vulnerability, i.e. how one can code to avoid the problem or, in some other way, mitigate its effects
- Implications for standardization, i.e. recommendations for groups creating language standards

This format is best explained by an example from the current draft. Anything appearing in curly brackets {} is our explanation of the intended content rather than the content itself.

6.17 Boundary Beginning Violation [XYX] {Every description is assigned an arbitrary three-letter code. This allows one to reference a description even if subsequent versions of the report are reorganized.}

6.17.1 Description of application vulnerability

{This is intended to be a very brief description of the vulnerability as it occurs in execution.}
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.}

6.17.3 Mechanism of failure

{This description is intended to depict the mechanism of failure connecting the programming language weakness to the vulnerability in the application.}
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 program's control flow.

6.17.4 Applicable language characteristics

{If the report does not contain an annex providing information specific to the language of interest, the reader may consult this section to determine whether the language is likely to have this weakness.}

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.

6.17.5 Avoiding the vulnerability or mitigating its effects

{This section describes, in generic terms, how the problem might be avoided or mitigated. A language-specific annex might contain more specific information.}

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 could 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 languages 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).

6.17.6 Implications for standardization

{This section suggests how language standards might be improved to improve the problem.}

- 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.

Currently, WG 23 has reasonably firm assurances that language-dependent annexes will be provided for Ada, C, and Fortran through cooperation with working groups 9, 14, and 5, respectively, of SC 22. We are hoping to find additional groups with the expertise to write annexes for other standards.