ISO/IEC JTC 1/SC 22/OWGV N 0218

On Removing Programming Language Bias from the Vulnerabilities Document

Date	15 August 2009
Contributed by	J-P Rosen
Original file name	AUJ-Vuln-JPR.doc
Notes	Drafted for submission to Ada User

On Removing Programming Language Bias from the Vulnerabilities Document

J-P Rosen
Adalog, 19-21 rue du 8 mai 1945, 94110 ARCUEIL, France; Tel: +33 1 41 24 31 40; email:rosen@adalog.fr

Abstract

ISO/SC22/WG23 is currently working on a document that identifies vulnerabilities in programming languages. The document is structured as a core report which is supposed to be independent of any programming language, and annexes related to the applicability of each vulnerabitlity in specific languages. Unfortunately, the core exhibits in places a bias, generally towards the C/C++ family of languages. This paper identifies those places in the report where the wording or intent was biased by the features of certain programming languages, and suggests improvements to remove them.

Keywords: Ada,C, C++, vulnerabilities.

1 Introduction

ISO/SC22/WG23 is a working group of ISO which has been formed with the goal of producing a technical report (TR) that identifies vulnerabilities in programming languages. The official title of the TR is *Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use* [1]. Since the goal of this TR is clearly to provide guidance in selecting a programming language based on the vulnerabilities that each language may exhibit, it is structured as a general core study which does not refer to any programming language in particular, with language annexes that describe how the vulnerability applies, or not, to the given language. Since the document originates from members of the C/C++ community (although there is strong participation from the Ada community), it is unavoidable that some bias from these languages has crept in the core document. For the sake of simplicity, we'll use the name "C*" to designate the C family of programming languages.

This paper identifies places where this bias has been introduced, and suggests improvements to remove them.

2 Typical C-isms

As noted above, it is very hard to keep the core document totally language agnostic. For example, the description of vulnerabilities often include examples, such examples are very useful, but they have to be given with some programming language. Moreover, vulnerabilities that exist in only one language have to be addressed in the core document, since the language annexes are not supposed to be adding new vulnerabilities.

However, there are cases where the description of the vulnerability reflects a C* approach (as opposed to C* syntax). Conversely, there are vulnerabilities in some languages that do not exist in C* due to the absence of features that correspond to the vulnerability. Not addressing these vulnerabilities is another hint of a C* bias.

For example, *Pointer Arithmetic* [RVG/6.22][1] addresses (rightly) the vulnerabilities caused by arithmetic operations on pointers. However, the recommendations mention pointer arithmetic only as a way of indexing arrays. Although other languages may provide pointer arithmetic, C is the only language where pointer arithmetic is connected to the indexing of arrays.

Another arguable statement is found in *Argument Passing to Library Functions* [TRJ/6.48]. There is no definition of what "library functions" are, but it seems that the intent is to refer to standard libraries provided with the language. The description states:

> *Libraries that supply objects or functions are in most cases not required to check the validity of parameters passed to them.*

Although such a statement might be applicable to some C* libraries, there is no reason to think that it is a general principle that applies to all languages. There is a general vulnerability connected to subprograms that do not check their arguments, but there is no reason to limit this to "libraries".

3 Lack of generality

Some vulnerabilities are related to functionalities that exist in several programming languages, but whose scope and features vary greatly among languages. For example, generics/templates exist in Ada, C++, Eiffel... However, the report describes mainly the C++ view:

"Many languages provide a mechanism that allows objects and/or functions to be defined parameterized by type, and then instantiated for specific types" (*Templates and Generics* [SYM/ 6.25.1])

[1] Each description of a vulnerability is identified with an arbitrary three-letter code. This is intended to make them independent of any renumbering of clauses that may happen during the preparation of the document, but makes it harder to retrieve the place where it is defined. We therefore refer to the vulnerabilities by their code, followed by the corresponding clause number in the version of the document [1] that was current as of June 8th, 2009.

This formulation is clearly too restrictive for Ada, where other entities (such as packages) can be generic, and where the possible parameter kinds include subprograms, constants, variables, and even other packages (through formal instantiations).

Another example can be found in *Likely Incorrect Expression* [KOA/6.32]. This vulnerability is mainly concerned with the unintended use of "=" in place of "==" in expressions. While it is true that this problem is haunting every C programmer, it has no equivalent in other languages. On the other hand, the description only mentions in passing the confusion between "&" and " &&", which does have equivalents in other languages ("and" and "and then" in Ada). Note that this vulnerability should be kept separate from the issue of order of evaluation, which is addressed by *Side-effects and Order of Evaluation* [SAM/6.31].

4 Left-out features

The introduction of [1] states that, due to the limited amount of resources, vulnerabilities related to some subjects were deliberately postponed. These subjects include:

- Object-oriented language features
- Concurrency
- Numerical analysis
- Scripting languages
- Some issues related to inter-language operability

Although it is understandable that the subject of vulnerabilities is gigantic, and that it is not possible to address them all, the choice of left-out feature is another indication of language bias: all of these features are either not provided by C (even though some of them, and notably object orientation, are provided by other languages of the C family), or related to domains where C is not particularly fit (like numerical analysis). To Ada users, for example, addressing concurrency would seem a much more important topic than syntactic ambiguity!

5 Abstracting the vulnerabilities

It should be understood that the C* bias found in the description of some vulnerabilities does not invalidate the value of the vulnerability; the issue is more on separating the general, high level problem that it addresses (which belongs to the core document) from how it shows in some specific language (which belongs to the language annex). This requires an effort for abstracting the vulnerability.

For example, *String Termination* [CJM/6.16] describes the vulnerability caused by forgetting the null character that terminates a string. The general vulnerability is about using a sentinel value to mark the end of a data structure; there is nothing specific to strings, not even to arrays, here.

Strangely enough, the document distinguishes *Boundary Beginning Violation* [XYX/6.17], *Unchecked Array Indexing* [XYZ/6.18], *Unchecked Array Copying* [XYW/6.19], and *Buffer Overflow* [XZB/6.20]. All these are variants of a single vulnerability: accessing an array outside of its bounds. The origin of this distinction is that in C, it is common practice to allocate arrays in the direction where the stack is growing; therefore, addressing below the array may ruin the return address, while addressing above it does not. This is not even connected to a particular language, but to a specific (although common) implementation technic.

Similarly, there are subtle distinctions between *Type System* [IHN/6.11], *Numeric Conversion Errors* [FLC/6.15], *Pointer Casting and Pointer Type Changes* [HFC/6.21], *Sign Extension Error* [XZI/6.29], and *Type-breaking Reinterpretation of Data* [AMV/6.46]. They are all occurrences of problems with conversions; the only possible distinction could be between semantic-preserving conversions (regular conversions in Ada) and non-semantic-preserving conversions (Unchecked_Conversion in Ada).

The same phenomenon appears with vulnerabilities related to bad pointers: *Pointer Arithmetic* [RVG/6.22], *Null Pointer Dereference* [XYH/6.23], *Dangling Reference to Heap* [XYK/6.24], and *Dangling References to Stack Frames* [DCM/6.40].

In the last two examples, we have clearly single vulnerabilities that can appear, in the C* languages, in various forms; the core should contain only the abstract formulation (incorrect pointer value), leaving the variants to the language annex.

6 Cross-references clauses

The standard vulnerability template includes a "Cross-reference" clause to provide links to other documents addressing the given concerns. All vulnerabilities have links to C or C++ standards, and only those, although Ravenscar and Spark (but not the HRG document) are mentioned in the bibliography.

This is a clear indication that the selection of rules was made from C* documents; although a good starting point, documents from other languages should have been considered right from the start. Otherwise, only C* vulnerabilities will be addressed, especially considering that at this point, it could be argued that it is too late to add new vulnerabilities to the document.

7 Conclusion

There is no doubt that the vulnerabilities identified in the document are real, and do happen in various programming languages, including the C* family of languages. However, the formulation of some of them, and the selection of vulnerabilities, show a strong C* influence in some cases.

We suggest in this paper some improvements to make the formulation more general and applicable to other languages, and identify the parts that should be moved to language specific annexes; we hope that, by following theses advices, the generality and overall quality of the document could be improved.

8 References

[1] ISO/IEC PDTR 24772.2, as of 2009-05-29

[2] Alan Burns, Brian Dobbing and Tullio Vardanega (June 2004). "Guide for the use of the Ada Ravenscar Profile in high integrity systems". ACM SIGAda Ada Letters XXIV (2): 1–74. Now part of annex D of ISO/IEC 8652:1995 with cor. 1 and amdt 1 (Programming language Ada).

[3] John Barnes: "High Integrity Software: The SPARK Approach to Safety and Security"

[4] ISO/IEC TR 15942:2000, Guidance for the Use of Ada in High Integrity Systems.