ISO/IEC JTC 1/SC 22/OWGV N 0189

Proposed rewrite of Clause 5

Date	4 May 2009
Contributed by	Larry Wagoner
Original file name	5_Vulnerability_issues_final.doc
Notes	Closes Action Item #10-05

5 Vulnerability issues

Software vulnerabilities are unwanted characteristics of software that may allow software to behave in ways that are unexpected by a reasonably sophisticated user of the software. The expectations of a reasonably sophisticated user of software may be set by the software's documentation or by experience with similar software. Programmers introduce vulnerabilities into software by failing to understand the expected behaviour (the software requirements), or by failing to correctly translate the expected behaviour into the actual behaviour of the software.

This document does not discuss a programmer's understanding of software requirements. This document does not discuss software engineering issues per se. This document does not discuss configuration management, build environments, code-checking tools, nor software testing. This document does not discuss the classification of software vulnerabilities according to safety or security concerns. This document does not discuss the costs of software vulnerabilities, or the costs of preventing them.

This document does discuss a reasonably competent programmer's failure to translate the understood requirements into correctly functioning software. This document does discuss programming language features known to contribute to software vulnerabilities. That is, this document discusses issues arising from those features of programming languages found to increase the frequency of occurrence of software vulnerabilities. The intention is to provide guidance to those who wish to specify coding guidelines for their own particular use.

A programmer writes source code in a programming language to translate the understood requirements into working software. The programmer selects and codes constructs specified by a programming language with the intention of achieving a written expression of the desired behaviour.

A program's expected behaviour might be stated in a complex technical document, which can result in a complex sequence of features of the programming language. Software vulnerabilities occur when a reasonably competent programmer fails to understand the totality of the effects of the language features combined to construct the software. The overall software may be a very complex technical document itself (written in a programming language whose definition is also a complex technical document).

Humans understand very complex situations by chunking, that is, by understanding pieces in a hierarchical scaled scheme. The programmer's initial choice of the chunk for software is the line of code. (In any particular case, subsequent analysis by a programmer may refine or enlarge this initial chunk.) The line of code is a reasonable initial choice because programming editors display source code lines. Programming languages are

often defined in terms of statements (among other units), which in many cases are synonymous with textual lines. Debuggers may execute programs stopping after every statement to allow inspection of the program's state. Program size and complexity can be estimated by the number of lines of source code (automatically counted without regard to language statements).
The recommendations contained in this Technical Report might also be considered to be code quality issues. Both kinds of issues might be addressed through the use of a systematic development process, use of development/analysis tools and thorough testing.

5.1 Issues arising from incomplete or evolving language specifications

While there are many millions of programmers in the world, there are only several hundreds of authors engaged in designing and specifying those programming languages defined by international standards. The design and specification of a programming language is very different from programming. Programming involves selecting and sequentially combining features from the programming language to (locally) implement specific steps of the software's design. In contrast, the design and specification of a programming language involves (global) consideration of all aspects of the programming language. This must include how all the features will interact with each other, and what effects each will have, separately and in any combination, under all foreseeable circumstances. Thus, language design has global elements that are not generally present in any local programming task.
The creation of the abstractions which become programming language standards therefore involve consideration of issues unneeded in many cases of actual programming. Therefore perhaps these issues are not routinely considered when programming in the resulting language. These global issues may motivate the definition of subtle distinctions or changes of state not apparent in the usual case wherein a particular language feature is used. Authors of programming languages may also desire to maintain compatibility with older versions of their language while adding more modern features to their language and so add what appears to be an inconsistency to the language. For example, some languages may allow a subprogram to be invoked without specifying the correct signature of the subprogram. This may be allowed in order to keep compatibility with earlier versions of the language where such usage was permitted, and despite the knowledge that modern practice demands the signature be specified. Specifically, the programming language C does not require a function prototype be within scope. The programming language Fortran does not require an explicit interface. Thus, language usage is improved by coding standards specifying that the signature be present.
A reasonably competent programmer therefore may not consider the full meaning of every language feature used, as only the desired (local or subset) meaning may correspond to the programmer's immediate intention. In consequence, a subset meaning of any feature may be prominent in the programmer's overall experience.
Further, the combination of features indicated by a complex programming goal can raise the combinations of effects, making a complex aggregation within which some of the effects are not intended**.**

5.1.1 Compiler Selection

Compiler selection is important to ensure a system operates safely and securely. Compilers are important as they are the intermediary between the human readable source code and the machine readable binary code. This crucial step is often overlooked and compilers, unless coming from a trusted source with digital signature, should be treated as any other commercial off the shelf software that has an unknown pedigree.
Often, developers analyze the source code to detect any code that can negatively impact security or safety. This aims to solve one part of the problem. After the source gets compiled, we need to be sure that the compiler did not insert any logic (maliciously or inadvertently) into the binary that compromises the systems security or safety. This is especially important because this type of vulnerability will be inserted into every piece of software that the compiler processes.
To combat against this, developers of security or safety critical systems should only use compilers from a trusted source with a digital signature. The trusted source should also provide evidence that the compiler is free from anomalous behaviour; similar to the way RTCA's DO-178B defines qualifiable tools. In addition, developers of critical software can perform source to binary traceability to ensure the compiler has not inserted any undesired logic into the binary code.

5.1.2 Issues arising from unspecified behaviour

While every language standard attempts to specify how software written in the language will behave in all circumstances, there will always be some behaviour that is not specified completely. In any circumstance, of course, a particular compiler will produce a program with some specific behaviour (or fail to compile the program at all). Where a programming language construct is insufficiently defined different translators may generate different behaviours from the same source code. The authors of language standards often have an interpretations or defects process in place to treat these situations once they become known, and, eventually, to specify one behaviour. However, the time needed by the process to produce corrections to the language standard is often long, as careful consideration of the issues involved is needed.
When programs are compiled with only one compiler, the programmer may not be aware when behaviour not specified by the standard has been produced. Programs relying upon behaviour not specified by the language standard may behave differently when they are compiled with different compilers. An experienced programmer may choose to use more than one compiler, even in one environment, in order to obtain diagnostics from more than one source. In this usage, any particular compiler must be considered to be a different compiler if it is used with different options (which can give it different behaviour), or is a different release of the same compiler (which may have different default options or may generate different code), or is on different hardware (which may have a different instruction set). In this usage, a different computer may be the same hardware with a different operating system, with different compilers installed, with different software libraries available, with a different release of the same operating system, or with a different operating system configuration.

5.1.3 Issues arising from implementation-defined behaviour

In some situations, a programming language standard may specifically allow compilers to support a range of possible behaviours to a given language feature or combination of features. This may enable a more efficient execution on a wider range of hardware, or enable use of the programming language in a wider variety of circumstances.
In order to allow use on a wide range of hardware, for example, many languages do not specify the amount of storage reserved for language-defined entities such as variables. The degree to which a diligent programmer may obtain information on the amount of storage reserved for entities varies among languages.
The authors of language standards are encouraged to provide lists of all allowed variations of behaviour (as many already do). Such a summary will benefit applications programmers, those who define applications coding standards, and those who make code-checking tools.

5.1.4 Issues arising from undefined behaviour

In some situations, a programming language standard may specify that program behaviour is undefined. While the authors of language standards naturally try to minimize these situations, they may be inevitable when attempting to define software recovery from errors, or other situations recognized as being incapable of precise definition.
An example of undefined behaviour, in many languages, is the use of the value of a variable that has not yet been assigned.

5.2 Issues arising from human cognitive limitations

The authors of programming language standards try to define programming languages in a consistent way, so that a programmer will see a consistent interface to the underlying functionality. Such consistency is intended to ease the programmer's process of selecting language features, by making different functionality available as regular variation of the syntax of the programming language. However, this goal may impose limitations on the variety of syntax used, and may result in similar syntax used for different purposes, or even in the same syntax element having different meanings within different contexts. For example, in the programming language C, a name followed by a parenthesized list of expressions may reference a macro or a function. Likewise, in the programming language Fortran, a name followed by a parenthesized list of expressions may reference an array or a function. Thus, without further knowledge, a semantic distinction may be invisible in the source code.

Any such situation imposes a strain on the programmer's limited human cognitive abilities to distinguish the relationship between the totality of effects of these constructs and the underlying behaviour actually intended during software construction.
Attempts by language authors to have distinct language features expressed by very different syntax may easily result in different programmers preferring to use different subsets of the entire language. This imposes a substantial difficulty to anyone who wants to employ teams of programmers to make whole software products or to maintain software written over time by several programmers. In short, it imposes a barrier to those who want to employ coding standards of any kind. The use of different subsets of a programming language may also render a programmer less able to understand other programmer's code. The effect on maintenance programmers can be especially severe.

5.3 Issues arising from a lack of predictable execution

If a reasonably competent programmer has a good understanding of the state of a program after reading source code as far as a particular line of code, the programmer ought to have a good understanding of the state of the program after reading the next line of code. However, some features, or, more likely, some combinations of features, of programming languages are associated with relatively decreased rates of the programmer's maintaining their understanding as they read through a program. It is these features and combinations of features that are indicated in this document, along with ways to increase the programmer's understanding as code is read.

Here, the term understanding means the programmer's recognition of all effects, including subtle or unintended changes of state, of any language feature or combination of features appearing in the program. This view does not imply that programmers only read code from beginning to end. It is simply a statement that a line of code changes the state of a program, and that a reasonably competent programmer ought to understand the state of the program both before and after reading any line of code. As a first approximation (only), code is interpreted line by line.

5.4 Issues arising from portability and interoperability

The representation of characters, the representation of true/false values, the set of valid addresses, the properties and limitations of any (fixed point or floating-point) numerical quantities, and the representation of programmer defined types and classes may vary among hardware, among languages (affecting inter-language software development), and among compilers of a given language. These variations may be the result of hardware differences, operating system differences, library differences, compiler differences, or different configurations of the same compiler (as may be set by environment variables or configuration files). In each of these circumstances, there is an additional burden on the programmer because part of the program's behaviour is indicated by a factor that is not a part of the source code. That is, the program's behaviour may be indicated by a factor that is invisible when reading the source code. Compilation control schemes (IDE projects, make, and scripts) further complicate this situation by abstracting and manipulating the relevant variables (target platform, compiler options, libraries, and so forth).

Many compilers of standard-defined languages also support language features that are not specified by the language standard. These non-standard features are called extensions. For portability, the programmer must be aware of the language standard, and use only constructs with standard-defined semantics. The motivation to use extensions may include the desire for increased functionality within a particular environment, or increased efficiency on particular hardware. There are well-known software engineering techniques for minimizing the ill effects of extensions; these techniques should be a part of any coding standard where they are needed, and they should be employed whenever extensions are used. These issues are software engineering issues and are not further discussed in this document.

Some language standards define libraries that are available as a part of the language definition. Such libraries are an intrinsic part of the respective language and are called

intrinsic libraries. There are also libraries defined by other sources and are called non-intrinsic libraries.

The use of non-intrinsic libraries to broaden the software primitives available in a given development environment is a useful technique, allowing the use of trusted functionality directly in the program. Libraries may also allow the program to bind to capabilities provided by an environment. However, these advantages are potentially offset by any lack of skill on the part of the designer of the library (who may have designed subtle or undocumented changes of state into the library's behaviour), and implementer of the library (who may not have the implemented the library identically on every platform), and even by the availability of the library on a new platform. The quality of the documentation of a third-party library is another factor that may decrease the reliability of software using a library in a particular situation by failing to describe clearly the library's full behaviour. If a library is missing on a new platform, its functionality must be recreated in order to port any software depending upon the missing library**.** The re-creation may be burdensome if the reason the library is missing is because the underlying capability for a particular environment is missing.

Using a non-intrinsic library usually requires that options be set during compilation and linking phases, which constitute a software behaviour specification beyond the source code. Again, these issues are software engineering issues and are not further discussed in this document**.**

Languages need to acknowledge the existence of other languages. Support for inter-language operability permits the implementation of large heterogeneous systems (systems which consist of a mixture of hardware platforms running software implemented using a mixture of compilers/languages).

Some languages define methods of binding to object code written in other common programming languages. Without considering interoperability, problems are encountered such as how to call a C function from Ada where the C function writes to one of its arguments – something that is not permitted in an Ada function because Ada has the concept of parameter modes and functions may only be “in” mode parameters where as procedure parameters may be “in” mode, “out” mode or “in out” mode – Ada solves this problem by treating a C function as if it were a procedure with an extra “out” mode parameter – the return value. Without such provision, it wouldn’t be possible for Ada to interface with C libraries.”

5.6 Inadequate language intrinsic support

Many languages are created to facilitate programming within an application domain. Some languages are specifically designed for programming of business applications, numerical computation or systems programming. Problems can arise when, for example, a language being used to implement a real-time, multi-threaded system lacks key features that are needed such as a way of enforcing mutual exclusion. Such features can be provided by the programming environment in the form of libraries, but the definition of such libraries may be proprietary and inclined to change in later releases. A vendor may even decide to withdraw support entirely for such a library. Also, such a library may not be verified and validated to the same standard as the compiler and the application being developed.

The use of OOP language features may well be highly appropriate for implementing a GUI but at the same time, other features such as dynamic memory management, heap utilization, inefficient data representation, and dynamic polymorphism are very unsuitable for implementing a solid real-time safety-critical system. If a language is so intrinsically bound to OOP, it is sensible to seek an alternative language for implementing some applications. Conversely, if the problem domain is GUI development, it will often (although not always) be advisable to use a language that has OOP features. Some potential problems may be preventable through the use of stronger types, or the use of controls such as array bounds checking or integer checking to avoid overflows. These stronger restraints on a language have a cost both in performance and in the flexibility to perform certain operations. Language designers must strike a balance between restraints in the language, performance and flexibility causing some languages to lean heavily toward one or more extremes in pursuit of some language attributes. The intrinsic support provided by a language can help considerably in avoiding vulnerabilities, but such support can cause the utility of programming within a particular application domain to diminish.

5.7 Language features prone to erroneous use

Certain language constructs are relatively simple and straightforward to use. Other ones are complex to use or easily misused in a legal, but unintended way. Programmers may use floating point variables and pointers without fully understanding the nuances of the data representation. Rarely needed constructs or constructs that can be substituted for a series of simpler constructs can be used without a complete understanding of the full effects of the constructs.

Syntactic language features that are not intolerant of common typo errors can produce some problems that are notoriously difficult to find. One common example of this is that C permits an unintentional assignment to be performed in a Boolean expression by the accidental use of a single “=” (assignment) instead of the intended “==” test for equality. It then allows the resulting value to be treated as a Boolean.

Mapping of vulnerabilities to sections in chapter 5:

5.1 ~~Lack of knowledge~~ Incomplete or evolving language specifications

6.2 Unspecified Behaviour [BQF]
6.3 Undefined Behaviour [EWF]
6.4 Implementation-defined Behaviour [FAB]
6.5 Deprecated Language Features [MEM]

5.2 Human cognitive limitations

6.7 Choice of Clear Names [NAI]
6.32 Likely Incorrect Expression [KOA]
6.38 Structured Programming [EWD]

5.3 Predictable execution

6.33 Dead and Deactivated Code [XYQ]
6.35 Demarcation of Control Flow [EOJ]
6.43 Returning Error Status [NZN]
6.44 Termination Strategy [REU]
6.48 Dynamically-linked Code and Self-modifying Code [NYY]
6.15 Numeric Conversion Errors [FLC]
6.30 Operator Precedence/Order of Evaluation [JCW]
6.31 Side-effects and Order of Evaluation [SAM]

5.4 Portability and Interoperability

6.8 Choice of Filenames and other External Identifiers [AJN]
6.12 Bit Representations [STR]
6.39 Passing Parameters and Return Values [CSJ]
6.47 Argument Passing to Library Functions [TRJ]

5.6 Inadequate language intrinsic support

6.9 Unused Variable [XYR]
6.11 Type System [IHN]
6.14 Enumerator Issues [CCB]
6.16 String Termination [CJM]
6.17 Boundary Beginning Violation [XYX]
6.18 Unchecked Array Indexing [XYZ]
6.19 Unchecked Array Copying [XYW]
6.20 Buffer Overflow [XZB]
6.23 Null Pointer Dereference [XYH]
6.24 Dangling Reference to Heap [XYK]
6.27 Initialization of Variables [LAV]
6.28 Wrap-around Error [XYY]
6.29 Sign Extension Error [XZI]
6.34 Switch Statements and Static Analysis [CLL]
6.36 Loop Control Variables [TEX]
6.41 Subprogram Signature Mismatch [OTR]
6.45 Type-breaking Reinterpretation of Data [AMV]
6.46 Memory Leak [XYL]

5.7 Language features prone to erroneous use

6.1 Obscure Language Features [BRS]
6.6 Pre-processor Directives [NMP]
6.10 Identifier Name Reuse [YOW]
6.13 Floating-point Arithmetic [PLF]
6.21 Pointer Casting and Pointer Type Changes [HFC]
6.22 Pointer Arithmetic [RVG]
6.25 Templates and Generics [SYM]
6.26 Inheritance [RIP]
6.37 Off-by-one Error [XZH]
6.40 Dangling References to Stack Frames [DCM]
6.42 Recursion [GDL]